Полтавський університет економіки і торгівлі
Навчально-науковий інститут денної освіти
Форма навчання денна
Кафедра комп'ютерних наук та інформаційних технологій

Допускається до захисту
Завідувач кафедри
_____Олена ОЛЬХОВСЬКА
(підпис)
«_____»_____202_ р.

КВАЛІФІКАЦІЙНА РОБОТА
на тему

«ПРОЄКТУВАННЯ ВЕБ-САЙТУ ДЛЯ ЕЛЕКТРОННОГО МАГАЗИНУ КНИГ З ІНТЕГРАЦІЄЮ ПЛАТІЖНИХ СЕРВІСІВ»

зі спеціальності 122 Комп'ютерні науки
освітня програма «Комп'ютерні науки»
ступеня бакалавр

Виконавець роботи Матвєєнко Дмитро Дмитрович
_____«____» _____202_ р.
(підпис)

Науковий керівник зав. каф, доц, к.ф.-м.н. Ольховська О. В.
_____«____»_____202_ р.
(підпис)

Рецензент

ПОЛТАВА 2026

РЕФЕРАТ

Записка: 102 с., 14 рис., 3 таблиці, 1 додаток, 16 джерел.

ЕЛЕКТРОННИЙ МАГАЗИН, ВЕБ-САЙТ, ІНТЕГРАЦІЯ ПЛАТІЖНИХ СЕРВІСІВ, STRIPE, NEXT.JS, NESTJS, POSTGRESQL, REST API, JWT, DOCKER, ЕЛЕКТРОННА КОМЕРЦІЯ

Об'єктом розробки є веб-сайт електронного магазину книг з підтримкою повного циклу онлайн-продажу — від перегляду каталогу до оплати замовлення через платіжний сервіс.

Предметом розробки є програмна реалізація клієнтської та серверної частин веб-застосунку електронної комерції з інтеграцією платіжного сервісу Stripe на основі технологій Next.js, NestJS та реляційної СУБД PostgreSQL.

Метою роботи є проєктування та реалізація веб-сайту електронного магазину книг, що забезпечує зручний перегляд і пошук товарів, формування кошика, оформлення замовлення з онлайн-оплатою через інтегрований платіжний сервіс та адміністрування магазину.

Результатом роботи стало розроблення веб-застосунку «BookNest» на базі фреймворків Next.js і NestJS з реляційною СУБД PostgreSQL. Реалізовано ключові модулі:

- модуль каталогу — перегляд, фільтрація, сортування та пагінація книг, сторінки категорій та авторів;
- модуль повнотекстового пошуку на основі пошукового рушія Meilisearch з толерантністю до помилок;
- модуль автентифікації — реєстрація, вхід, JWT-токени з ротацією refresh-токенів, скидання пароля;
- модуль кошика — підтримка кошика гостя та авторизованого користувача зі злиттям при вході;
- модуль оформлення замовлення з інтеграцією платіжного сервісу Stripe через Payment Element;

- модуль обробки платіжних подій через webhook Stripe з перевіркою підпису та ідемпотентністю;
- модуль замовлень, відгуків з модерацією, списку бажань та промокодів;
- адміністративна панель — управління книгами, категоріями, авторами, замовленнями, відгуками, користувачами та статистикою.

Особливості: монорепозиторій з розподілом на клієнтський та серверний застосунки, контейнеризація всіх сервісів засобами Docker, двомовний інтерфейс (українська та англійська), автоматичне наповнення каталогу даними з відкритого джерела Open Library, інтерактивна документація API на базі специфікації OpenAPI 3.0.

Проведено модульне тестування сервісного шару та функціональне тестування ключових сценаріїв: реєстрація, додавання книги до кошика, оформлення й оплата замовлення тестовою карткою, обробка платіжного webhook.

«BookNest» може використовуватися як готова основа для запуску інтернет-магазину книг або іншої товарної ніші, а також як навчальний приклад побудови веб-застосунку електронної комерції з інтеграцією платіжних сервісів.

ЗМІСТ

СПИСОК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ, ТЕРМІНІВ

Умовні позначення, символи, скорочення, терміни	Пояснення умовних позначень, скорочень, символів
API	Application Programming Interface — програмний інтерфейс взаємодії застосунків
CDN	Content Delivery Network — мережа доставки вмісту
CRUD	Create, Read, Update, Delete — базові операції над даними
CSP	Content Security Policy — політика безпеки вмісту
CSRF	Cross-Site Request Forgery — міжсайтова підробка запиту
CSS	Cascading Style Sheets — каскадні таблиці стилів
DTO	Data Transfer Object — об'єкт передавання даних
HTML	HyperText Markup Language — мова розмітки гіпертексту
HTTP	HyperText Transfer Protocol — протокол передачі гіпертексту
HTTPS	HyperText Transfer Protocol Secure — захищена версія протоколу HTTP
JSON	JavaScript Object Notation — текстовий формат обміну даними
JWT	JSON Web Token — стандарт токенів для автентифікації (RFC 7519)

ORM	Object-Relational Mapping — об'єктно-реляційне відображення
REST	Representational State Transfer — архітектурний стиль взаємодії
RFC	Request for Comments — серія документів з технічними стандартами
SDK	Software Development Kit — набір засобів розробки
SMTP	Simple Mail Transfer Protocol — протокол передачі електронної пошти
SPA	Single Page Application — односторінковий веб-додаток
SQL	Structured Query Language — мова структурованих запитів
SSR	Server-Side Rendering — рендеринг на стороні сервера
TLS	Transport Layer Security — протокол захищеного передавання даних
UI	User Interface — інтерфейс користувача
URL	Uniform Resource Locator — уніфікований локатор ресурсу
UUID	Universally Unique Identifier — універсальний унікальний ідентифікатор
БД	База даних
СУБД	Система управління базами даних

ВСТУП

Електронна комерція є однією з найдинамічніших галузей сучасної цифрової економіки. Перехід торгівлі в онлайн-простір став незворотним: покупці очікують можливості обрати та придбати товар у будь-який час, з будь-якого пристрою, з прозорим процесом оплати та доставки. Книжковий ринок не є винятком — продаж паперових книг через інтернет-магазини стабільно зростає, оскільки онлайн-формат дозволяє запропонувати покупцеві значно ширший асортимент, ніж фізична книгарня, зручний пошук за жанрами й авторами, відгуки інших читачів та персоналізовані рекомендації.

Ключовим елементом будь-якого інтернет-магазину є інтеграція з платіжними сервісами. Саме механізм онлайн-оплати перетворює каталог товарів на повноцінний інструмент продажу. Сучасні платіжні провайдери, такі як Stripe, надають розробникам програмні інтерфейси, що дозволяють приймати платежі без необхідності самостійно зберігати й обробляти дані банківських карток, делегуючи відповідальність за відповідність стандарту безпеки PCI DSS платіжному сервісу. Це суттєво знижує технічні та юридичні бар'єри для запуску електронного магазину, проте водночас вимагає від розробника правильного проєктування взаємодії: створення платіжного наміру, безпечного підтвердження оплати на стороні клієнта та надійної обробки асинхронних повідомлень про результат платежу.

Розробка веб-сайту електронного магазину книг є комплексною інженерною задачею, що поєднує проєктування зручного користувацького інтерфейсу, побудову серверної логіки з модулями каталогу, кошика, замовлень та автентифікації, проєктування реляційної бази даних, забезпечення інформаційної безпеки та інтеграцію зовнішніх сервісів — платіжного провайдера, пошукового рушія, сховища файлів і поштового сервісу. Якість архітектурних рішень безпосередньо

впливає на швидкодію сайту, безпеку персональних і платіжних даних користувачів та зручність подальшого супроводу системи.

Сучасний підхід до побудови веб-застосунків спирається на чіткий розподіл клієнтської та серверної частин, використання типізованих мов програмування, контейнеризацію сервісів та автоматизацію розгортання. Фреймворк Next.js забезпечує продуктивний рендеринг сторінок на стороні сервера та оптимізацію клієнтського коду, фреймворк NestJS надає модульну архітектуру для побудови серверного REST API, а система контейнеризації Docker дозволяє відтворювано розгортати всю інфраструктуру магазину однією командою. Поєднання цих технологій дає змогу створити продуктивний, безпечний і масштабований інтернет-магазин.

Метою кваліфікаційної роботи є проєктування та реалізація веб-сайту електронного магазину книг з інтеграцією платіжних сервісів, що забезпечує повний цикл онлайн-продажу: перегляд і пошук товарів, формування кошика, оформлення замовлення з онлайн-оплатою та адміністрування магазину.

Для досягнення поставленої мети у роботі визначено такі основні завдання:

- проаналізувати предметну область електронної комерції у книжковій ніші, дослідити типові процеси онлайн-продажу та вимоги до інтернет-магазину;
- провести огляд існуючих українських інтернет-магазинів книг, виконати їх порівняльний аналіз та обґрунтувати доцільність власної розробки;
- дослідити теоретичні засади побудови сучасних веб-застосунків: клієнт-серверну архітектуру, побудову REST API, механізми автентифікації на основі JWT, принципи інтеграції платіжних сервісів;
- обґрунтувати вибір технологічного стеку для клієнтської та серверної частин, СУБД, платіжного провайдера та допоміжних сервісів;
- спроєктувати загальну архітектуру системи, схему реляційної бази даних та структуру програмного коду;
- реалізувати ключові модулі магазину: каталог, пошук, автентифікацію, кошик, оформлення замовлення з інтеграцією Stripe, обробку платіжних webhook, замовлення, відгуки та промокоди;

- розробити адміністративну панель для управління контентом і замовленнями магазину;
- провести модульне та функціональне тестування реалізованих компонентів, оформити інструкцію для користувача та адміністратора.

Об'єктом дослідження є процеси функціонування веб-сайту електронного магазину з онлайн-оплатою товарів.

Предметом дослідження є програмна реалізація клієнтської та серверної частин веб-застосунку електронної комерції з інтеграцією платіжного сервісу Stripe на основі технологій Next.js, NestJS та PostgreSQL.

Методи дослідження. У роботі застосовано методи системного аналізу для дослідження предметної області, методи порівняльного аналізу для оцінювання аналогів і технологій, методи об'єктно-орієнтованого проєктування для побудови архітектури, методи реляційної алгебри та нормалізації для проєктування схеми бази даних, а також практичні методи розробки програмного забезпечення з використанням систем контролю версій та контейнеризації.

Практичне значення одержаних результатів полягає у створенні готового до використання веб-застосунку електронного магазину книг, який може бути впроваджений як основа реального інтернет-магазину, а також у систематизації знань про сучасні підходи до побудови застосунків електронної комерції з інтеграцією платіжних сервісів.

Кваліфікаційна робота складається зі вступу, постановки задачі, трьох розділів, висновків, списку інформаційних джерел та одного додатку.

ПОСТАНОВКА ЗАДАЧІ

Основною задачею кваліфікаційної роботи є проєктування та реалізація веб-сайту електронного магазину книг з інтеграцією платіжних сервісів, що забезпечує безпечне виконання повного циклу онлайн-продажу товарів і відповідає сучасним вимогам до застосунків електронної комерції.

Розроблюваний веб-сайт повинен надавати відвідувачеві зручний інтерфейс для перегляду каталогу книг з фільтрацією за категоріями, авторами, ціною, мовою та наявністю, повнотекстовий пошук, детальні сторінки книг з відгуками, а також можливість сформувати кошик і оформити замовлення з онлайн-оплатою. Зареєстрований користувач повинен мати особистий кабінет з історією замовлень, списком бажань та керуванням адресами доставки. Адміністратор магазину повинен отримати окрему панель для управління асортиментом, замовленнями, відгуками та перегляду статистики продажів.

Функціональні вимоги до системи передбачають наявність модулів каталогу, пошуку, автентифікації, кошика, оформлення замовлення, інтеграції з платіжним сервісом, обробки платіжних подій, замовлень, відгуків, промокодів та адміністрування. Нефункціональні вимоги охоплюють безпеку (хешування паролів алгоритмом Argon2id, автентифікація за стандартом JWT з ротацією refresh-токенів, перевірка підпису платіжних webhook, обмеження частоти запитів, захисні HTTP-заголовки), продуктивність (рендеринг каталогу на стороні сервера, кешування, повнотекстовий пошук) та простоту розгортання (контейнеризація всіх сервісів засобами Docker, автоматичне застосування міграцій бази даних).

Для досягнення поставленої задачі необхідно виконати такі підзадачі:

1. проаналізувати предметну область електронної комерції у книжковій ніші, визначити типові ролі користувачів, бізнес-процеси та вимоги до інтернет-магазину книг;
2. провести огляд існуючих українських інтернет-магазинів книг, виконати

порівняльний аналіз їх функціональних можливостей та скласти таблицю порівняння аналогів;

3. дослідити теоретичні засади побудови сучасних веб-застосунків: клієнт-серверну архітектуру, принципи побудови REST API, механізми автентифікації на основі JWT;
4. дослідити принципи інтеграції платіжних сервісів, модель платіжного наміру Stripe Payment Intent та механізм асинхронної обробки платіжних подій через webhook;
5. обґрунтувати вибір технологічного стеку: фреймворків клієнтської та серверної частин, СУБД, ORM, платіжного провайдера, пошукового рушія, об'єктного сховища та засобів контейнеризації;
6. спроєктувати загальну архітектуру системи з розподілом на клієнтський застосунок, серверне API, фонового обробника черг та інфраструктурні сервіси;
7. спроєктувати реляційну схему бази даних із підтримкою користувачів, книг, категорій, авторів, видавництв, кошика, замовлень, відгуків, промокодів та журналу подій, реалізувати систему версіонування міграцій;
8. реалізувати модуль каталогу з фільтрацією, сортуванням, пагінацією та повнотекстовим пошуком на основі пошукового рушія Meilisearch;
9. реалізувати модуль автентифікації з підтримкою реєстрації, входу, ротації refresh-токенів, скидання пароля та рольового розмежування доступу;
10. реалізувати модуль кошика з підтримкою гостьового та авторизованого режимів і злиттям кошика при вході користувача в систему;
11. реалізувати модуль оформлення замовлення з інтеграцією платіжного сервісу Stripe: створення платіжного наміру, підтвердження оплати через Payment Element, обробку webhook з перевіркою підпису та ідемпотентністю;
12. реалізувати модулі замовлень, відгуків з модерацією, списку бажань та промокодів;
13. розробити адміністративну панель для управління книгами, категоріями, авторами, замовленнями, відгуками, користувачами та перегляду статистики;

14. підготувати конфігурацію Docker для розгортання всіх сервісів системи, написати модульні тести, провести функціональне тестування ключових сценаріїв та оформити інструкцію для користувача й адміністратора.

1. ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1. Аналіз предметної області електронної комерції у книжковій ніші

Електронна комерція (e-commerce) — це сфера економічної діяльності, у межах якої укладання угод, продаж товарів і послуг та проведення розрахунків відбуваються за допомогою інформаційно-комунікаційних технологій, насамперед мережі Інтернет. Інтернет-магазин є програмно-апаратним комплексом, що реалізує повний цикл взаємодії продавця і покупця в онлайн-середовищі: представлення асортименту, оформлення та оплату замовлення, його супровід і доставку. Для книжкової ніші електронна комерція є особливо органічною, оскільки книга — це стандартизований товар з чіткими атрибутами (назва, автор, видавництво, ISBN, рік видання, кількість сторінок), який зручно описувати, каталогізувати та порівнювати в цифровому каталозі.

У типовому інтернет-магазині книг можна виділити три основні ролі учасників: відвідувач (неавторизований гість), зареєстрований покупець та адміністратор магазину. Відвідувач має доступ до публічної частини сайту — перегляду каталогу, пошуку, сторінок книг і відгуків — і може наповнювати кошик ще до реєстрації. Зареєстрований покупець додатково отримує особистий кабінет з історією замовлень, списком бажань, збереженими адресами доставки та можливістю залишати відгуки. Адміністратор працює з закритою панеллю керування, через яку він підтримує актуальність асортименту, обробляє замовлення, модерує відгуки та аналізує показники продажів.

Ключовим бізнес-процесом магазину є оформлення замовлення. Узагальнено цей процес складається з таких етапів: користувач переглядає каталог і додає обрані книги до кошика; переходить до сторінки оформлення замовлення, де вказує контактні дані, адресу та спосіб доставки; обирає спосіб оплати; підтверджує платіж; отримує підтвердження замовлення, а магазин — повідомлення про

необхідність його обробки. Узагальнену схему процесу онлайн-покупки книги наведено на рисунку (див. рис. 1.1).

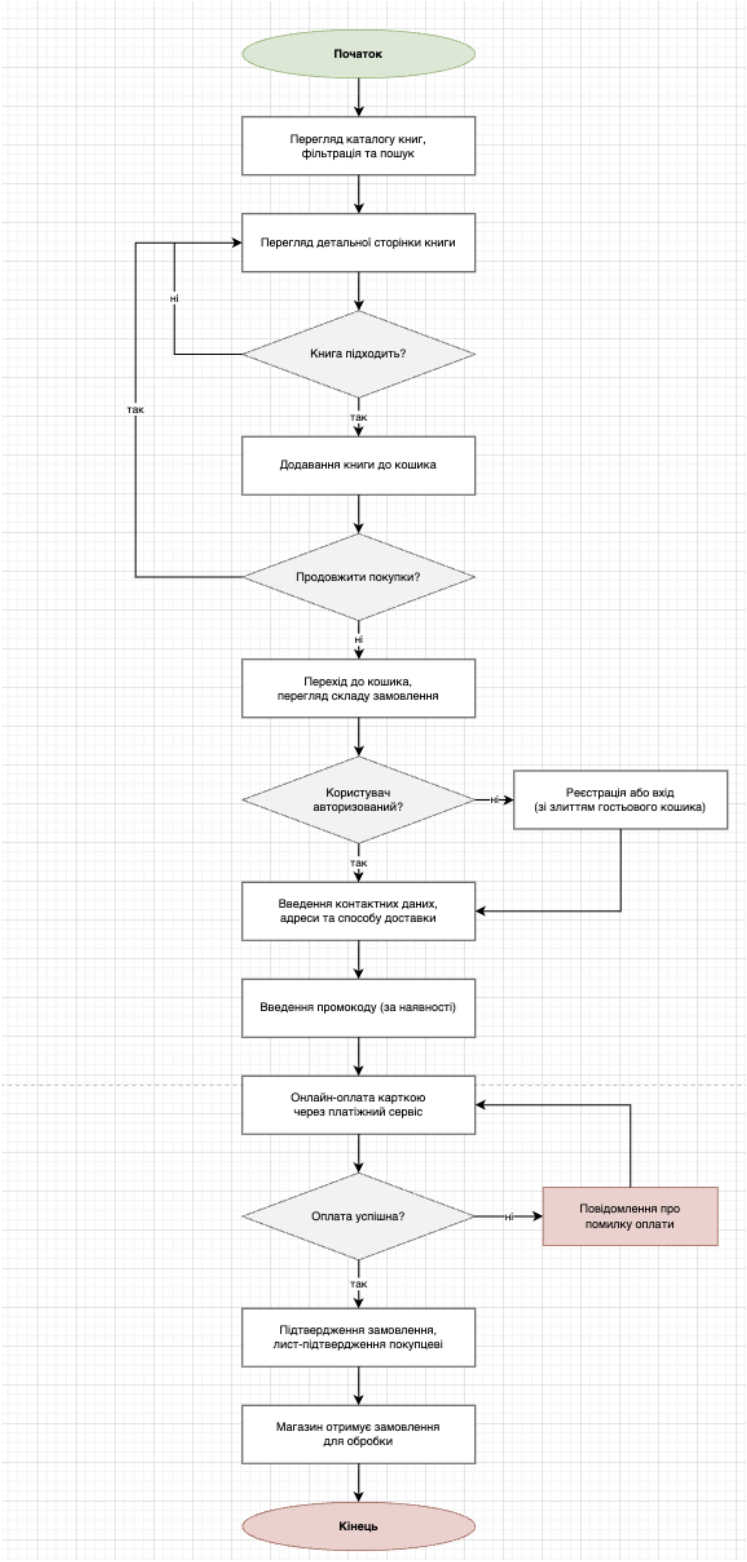

Рисунок 1.1 – Узагальнена схема процесу онлайн-покупки книги в інтернет-магазині

Окрему увагу слід приділити етапу оплати, оскільки саме він є технічно та юридично найскладнішим. Самостійне приймання платежів вимагало б від магазину зберігання й обробки даних банківських карток, що накладає сувору відповідальність за відповідність стандарту безпеки індустрії платіжних карток PCI DSS. Тому переважна більшість сучасних інтернет-магазинів делегує цю функцію спеціалізованим платіжним сервісам, які беруть на себе зберігання карткових даних, перевірку транзакцій та боротьбу із шахрайством, надаючи магазину лише програмний інтерфейс для ініціювання платежу та отримання його результату.

До функціональних вимог сучасного інтернет-магазину книг належать: структурований каталог з категоріями та фільтрами, повнотекстовий пошук, детальні картки товарів, кошик із підтримкою гостьового режиму, оформлення замовлення з онлайн-оплатою, особистий кабінет покупця, система відгуків і рейтингів, механізм знижок і промокодів та адміністративна панель. До нефункціональних вимог відносять швидкодію (час відгуку сторінок), безпеку персональних і платіжних даних, надійність обробки замовлень, адаптивність інтерфейсу до різних пристроїв та зручність супроводу. Аналіз предметної області показує, що центральним для книжкового інтернет-магазину є поєднання якісно спроєктованого каталогу з надійною та безпечною інтеграцією платіжного сервісу, що й визначає напрям цієї кваліфікаційної роботи.

1.2. Огляд існуючих інтернет-магазинів книг

На українському ринку представлено низку інтернет-магазинів книг, які мають усталену аудиторію та розвинений функціонал. Розгляд найвідоміших із них дозволяє визначити галузеві стандарти користувацького досвіду, типовий набір функцій та виявити можливості для вдосконалення, які буде враховано під час проєктування власної системи.

Yakaboo — найбільший український онлайн-магазин книг із широким асортиментом друкованих та електронних видань, аудіокниг і супутніх товарів. Сайт має розвинену систему категорій, фільтрів і рекомендацій, програму

лояльності, мобільний застосунок та інтеграцію з кількома платіжними та логістичними сервісами. Сильними сторонами є масштаб каталогу і зрілість сервісу; до недоліків можна віднести перевантаженість інтерфейсу великою кількістю банерів і рекламних блоків, що ускладнює зосередження на пошуку конкретної книги.

Книгарня «Є» — інтернет-магазин відомої мережі книгарень з акцентом на українську книгу. Має охайний каталог, зв'язок з фізичними магазинами мережі (перевірка наявності у відділеннях, самовивіз), розділ подій. Водночас функції пошуку та фільтрації менш гнучкі порівняно з найбільшими майданчиками, а робота сайту під навантаженням періодично сповільнюється.

Bookchef — інтернет-магазин однойменного видавництва. Інтерфейс сучасний і лаконічний, добре подані новинки та добірки, проте асортимент здебільшого обмежений продукцією власного й партнерських видавництв, що звужує вибір порівняно з універсальними майданчиками.

Загальний висновок з огляду полягає в тому, що великі майданчики пропонують широкий функціонал, але часто за рахунок переобтяженого інтерфейсу, тоді як менші магазини мають охайніший дизайн, але поступаються в гнучкості пошуку, фільтрації та повноті асортименту. Приклад типового інтерфейсу сторінки каталогу інтернет-магазину книг наведено на рисунку (див. рис. 1.2).

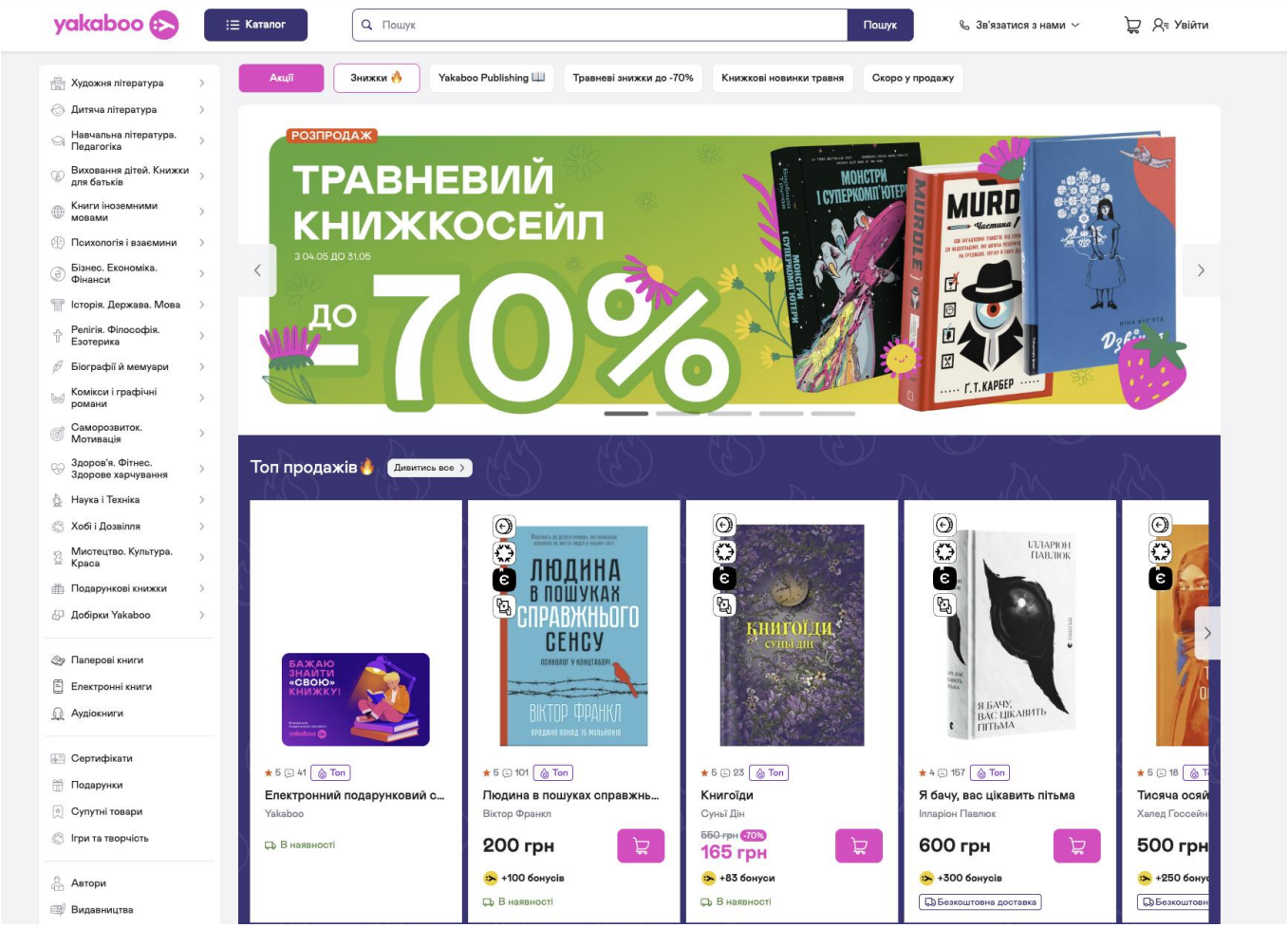

Рисунок 1.2 – Приклад інтерфейсу сторінки каталогу інтернет-магазину книг

Для систематизованого порівняння розглянутих рішень за ключовими критеріями — структурою каталогу, гнучкістю фільтрів, наявністю повнотекстового пошуку, онлайн-оплатою, особистим кабінетом та системою відгуків — складено таблицю 1.1. Розроблюваний у межах цієї кваліфікаційної роботи продукт «BookNest» включено до таблиці для подальшого обґрунтування його місця серед аналогів.

Таблиця 1.1 — Порівняння функціональних можливостей інтернет-магазинів книг

Критерій	**Yakaboo**	**Книгарня «Є»**	**Bookchef**	**BookNest**
Широкий універсальний каталог	+	+	–	+
Гнучкі фільтри каталогу	+	частково	частково	+
Повнотекстовий пошук з толерантністю до помилок	+	–	–	+
Онлайн-оплата карткою	+	+	+	+
Кошик для неавторизованого гостя	+	+	+	+
Особистий кабінет та історія замовлень	+	+	+	+
Система відгуків з модерацією	+	частково	+	+
Промокоди та знижки	+	+	+	+
Двомовний інтерфейс	частково	–	–	+
Лаконічний інтерфейс без перевантаження рекламою	–	+	+	+

Як видно з таблиці 1.1, кожен з аналогів має сильні сторони, проте жоден не поєднує одночасно гнучкого пошуку, охайного двомовного інтерфейсу та сучасної архітектури. Це створює простір для розробки власного рішення, орієнтованого на якісний користувацький досвід та технічну прозорість.

1.3. Обґрунтування доцільності власної розробки

Проведений огляд аналогів свідчить, що готові рішення для книжкової електронної комерції умовно поділяються на дві групи. Перша — великі універсальні майданчики з широким функціоналом, але переобтяженим інтерфейсом та закритою для вивчення архітектурою. Друга — менші спеціалізовані магазини з охайним дизайном, але обмеженими можливостями пошуку, фільтрації та вузьким асортиментом. Окремо існують готові платформи електронної комерції загального призначення, проте їх адаптація під конкретні потреби часто потребує значних зусиль, а внутрішня логіка залишається непрозорою.

Розробка власного веб-сайту електронного магазину книг є доцільною з кількох міркувань. По-перше, власна реалізація дозволяє повністю контролювати архітектуру системи, склад функціональних модулів та схему даних, не успадковуючи зайвої складності готових платформ. По-друге, вона дає змогу свідомо обрати сучасний технологічний стек і спроєктувати чисту інтеграцію з платіжним сервісом, що є центральною темою цієї роботи. По-третє, власний застосунок є цінним як систематизований навчальний та практичний приклад побудови повноцінного застосунку електронної комерції — від проєктування бази даних до обробки асинхронних платіжних подій.

На основі аналізу предметної області та виявлених недоліків аналогів сформульовано вимоги до власного продукту: лаконічний адаптивний двомовний інтерфейс без рекламного перевантаження; структурований каталог з гнучкими фільтрами та повнотекстовим пошуком, толерантним до помилок; кошик з підтримкою гостьового режиму та злиттям при вході; безпечне оформлення замовлення з онлайн-оплатою через інтегрований платіжний сервіс; надійна асинхронна обробка результатів платежу; особистий кабінет покупця та повноцінна адміністративна панель; контейнеризоване розгортання всієї інфраструктури однією командою.

Таким чином, розробка веб-сайту електронного магазину книг «BookNest» з інтеграцією платіжних сервісів є обґрунтованою та актуальною задачею. Подальші розділи роботи присвячено дослідженню теоретичних засад побудови такого застосунку, обґрунтуванню технологічного стеку та практичній реалізації спроєктованої системи.

2. ТЕОРЕТИЧНА ЧАСТИНА

2.1. Архітектура сучасних веб-застосунків електронної комерції

В основі сучасного веб-застосунку лежить клієнт-серверна архітектура, у якій система розподілена на дві взаємодійні частини. Клієнтська частина (frontend) відповідає за представлення інформації та взаємодію з користувачем у браузері, серверна частина (backend) — за бізнес-логіку, роботу з базою даних, автентифікацію та інтеграцію зовнішніх сервісів. Взаємодія між ними відбувається через мережу за протоколом HTTP, як правило у форматі прикладного інтерфейсу REST API, що обмінюється даними у форматі JSON. Такий розподіл забезпечує незалежну розробку, тестування й масштабування кожної частини, а також можливість використання одного й того самого серверного API різними клієнтами — веб-сайтом, мобільним застосунком чи зовнішніми системами.

Узагальнену схему клієнт-серверної архітектури веб-застосунку електронної комерції з допоміжними сервісами наведено на рисунку (див. рис. 2.1). Браузер користувача звертається до клієнтського застосунку, який, своєю чергою, через REST API взаємодіє із серверною частиною; сервер працює з реляційною базою даних, кешем, пошуковим рушієм, об'єктним сховищем файлів та зовнішнім платіжним сервісом.

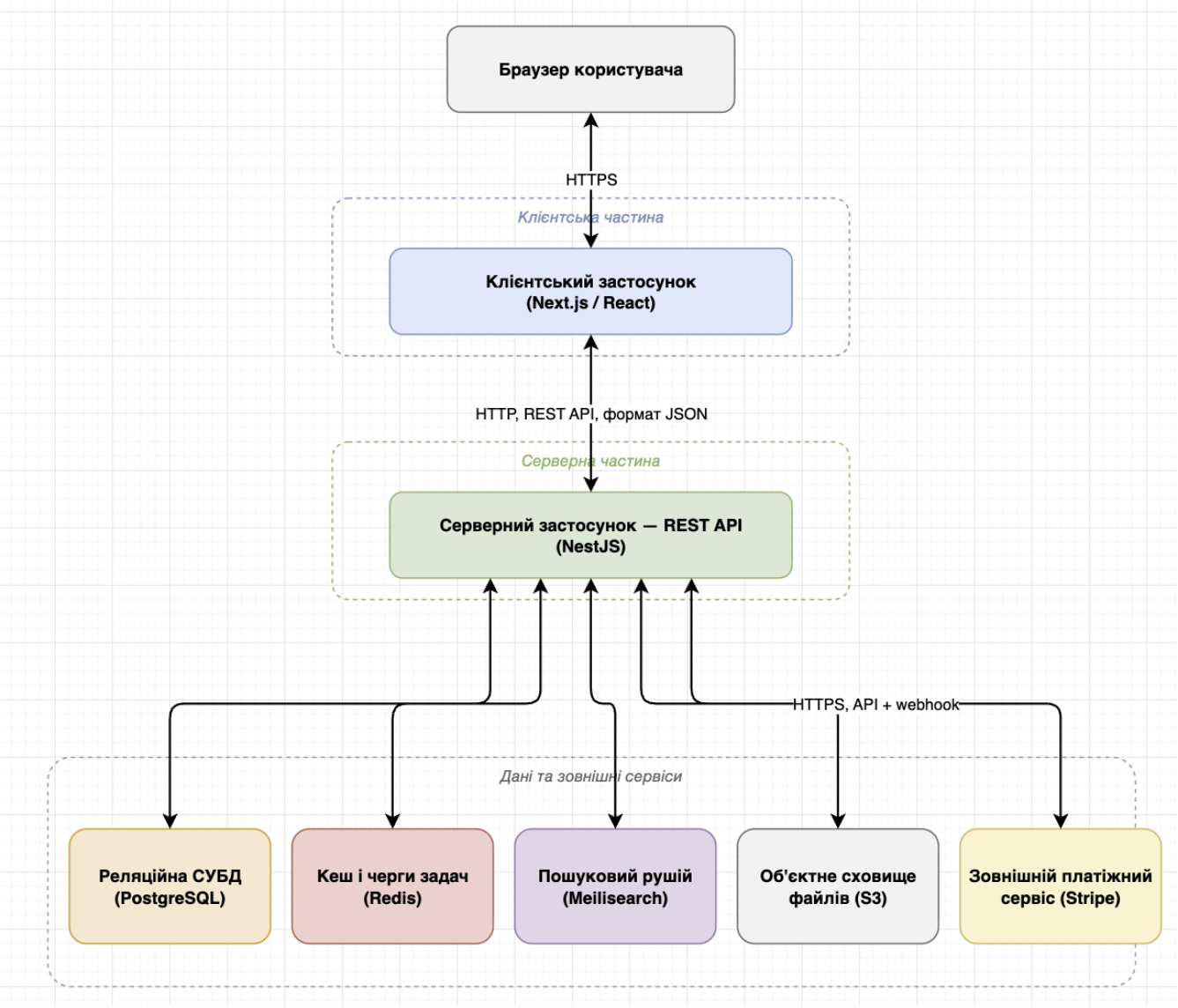

Рисунок 2.1 – Узагальнена клієнт-серверна архітектура веб-застосунку електронної комерції

Для клієнтської частини сучасних веб-застосунків де-факто стандартом стали компонентні бібліотеки та фреймворки, насамперед React, що дозволяють будувати інтерфейс із повторно використовуваних компонентів. Поверх React побудовано фреймворк Next.js, який додає рендеринг сторінок на стороні сервера (Server-Side Rendering), маршрутизацію на основі файлової структури, оптимізацію зображень та поділ коду. Рендеринг на стороні сервера є особливо важливим для інтернет-магазину, оскільки забезпечує швидке відображення каталогу та коректну індексацію сторінок пошуковими системами [1, 3]. Клієнтський код таких застосунків, як правило, пишуть статично типізованою мовою програмування

TypeScript, а оформлення інтерфейсу виконують засобами утилітарних CSS-фреймворків, зокрема TailwindCSS [4, 14].

Серверна частина будується за принципом модульності: код поділяється на окремі модулі за функціональними напрямами (каталог, кошик, замовлення, автентифікація тощо), кожен з яких містить контролери, що приймають HTTP-запити, сервіси з бізнес-логікою та об'єкти передавання даних (DTO) з правилами валідації. Такий підхід реалізовано, зокрема, у фреймворку NestJS. Для роботи з реляційною базою даних застосовують об'єктно-реляційне відображення (ORM), яке дозволяє описувати структуру даних у вигляді типізованих моделей, а запити виконувати безпечними параметризованими викликами замість формування рядків SQL вручну. Окремо виносять інфраструктурні сервіси: кеш та черги задач, повнотекстовий пошук, об'єктне сховище файлів і поштовий сервіс, що взаємодіють із серверною частиною за власними протоколами [2].

2.2. Механізми автентифікації та безпеки веб-застосунків

Автентифікація — це процес підтвердження достовірності особи користувача, авторизація — визначення дозволених йому дій. У веб-застосунках електронної комерції автентифікація захищає особистий кабінет, історію замовлень і платіжні операції, а авторизація розмежовує доступ між звичайним покупцем та адміністратором магазину. Найпоширенішим сучасним підходом до автентифікації в REST API є використання токенів за стандартом JSON Web Token (JWT, RFC 7519). JWT — це компактний самодостатній токен, що містить корисне навантаження (ідентифікатор користувача, роль, час видачі та закінчення дії) і криптографічний підпис, який сервер перевіряє секретним ключем, не звертаючись до бази даних [9].

Поширеною є схема двох токенів: короткоживучого access-токена, який супроводжує кожен запит до захищених ресурсів, та довгоживучого refresh-токена, призначеного лише для отримання нового access-токена. Короткий час життя access-токена обмежує наслідки його можливого витоку, а refresh-токен зберігається

у вигляді хешу в базі даних, що дозволяє відкликати сесію. Важливим механізмом є ротація refresh-токенів: при кожному оновленні старий токен позначається відкликаним, а користувачеві видається новий. Це дозволяє виявляти повторне використання викраденого токена. Схему автентифікації на основі пари access- та refresh-токенів з їх ротацією наведено на рисунку (див. рис. 2.2).

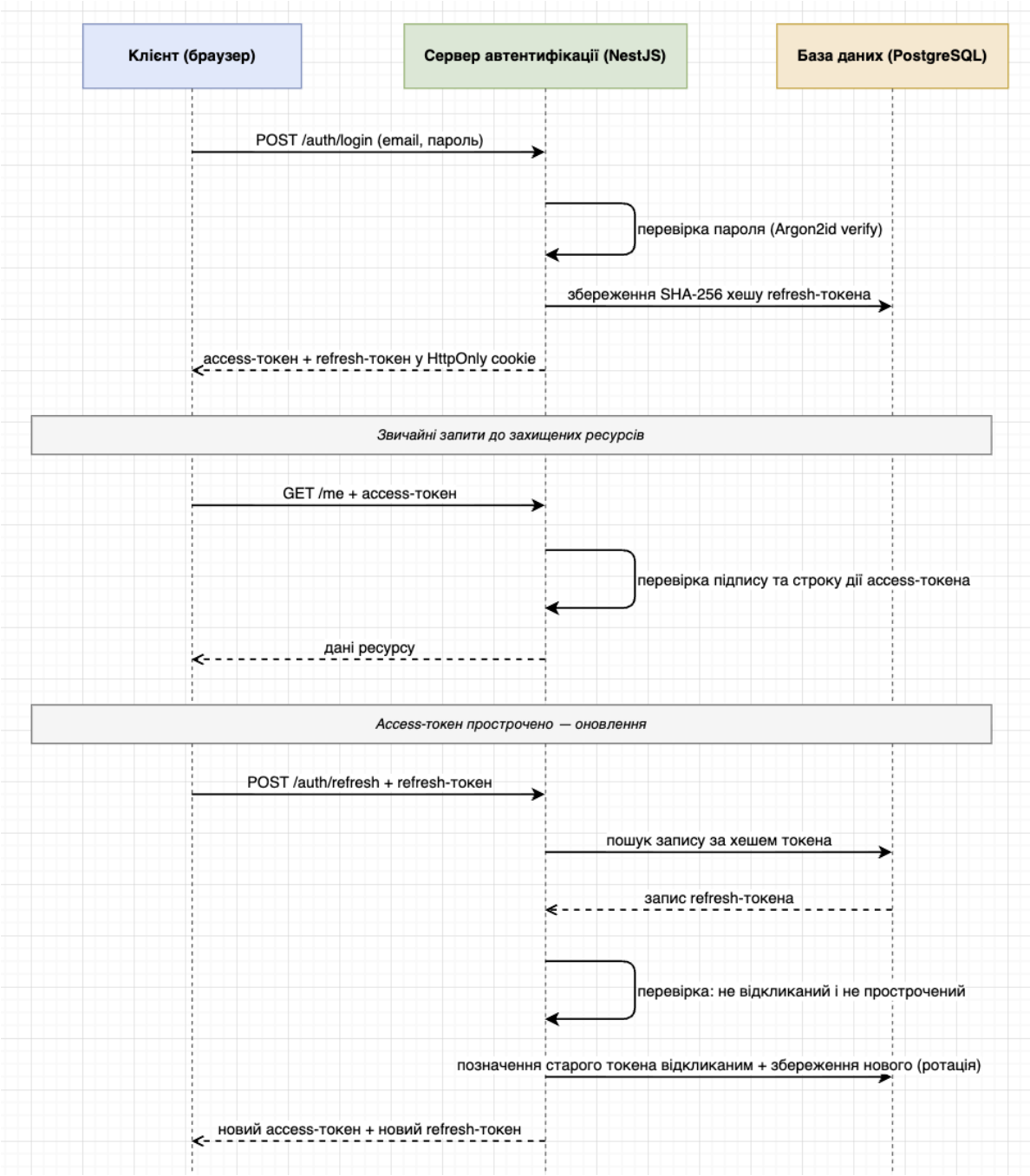

Рисунок 2.2 – Схема автентифікації на основі пари JWT-токенів з ротацією refresh-токена

Окрему роль відіграє надійне зберігання паролів. Паролі ніколи не зберігають у відкритому вигляді — натомість застосовують спеціалізовані алгоритми хешування, стійкі до підбору. Сучасним рекомендованим алгоритмом є Argon2id, який поєднує стійкість до атак з використанням графічних процесорів та спеціалізованого обладнання завдяки налаштовуваним вимогам до обсягу пам'яті, кількості ітерацій та паралелізму. Токени, що передаються користувачеві (refresh-токени, токени скидання пароля), також доцільно зберігати в базі лише у вигляді криптографічного хешу.

Окрім автентифікації, безпека веб-застосунку забезпечується низкою додаткових заходів, узагальнених у рекомендаціях OWASP. До них належать: валідація усіх вхідних даних на сервері, використання параметризованих запитів до бази даних для запобігання SQL-ін'єкціям, екранування вмісту на виході для захисту від міжсайтового скриптингу, передавання токенів у захищених cookie з атрибутами HttpOnly та SameSite, встановлення захисних HTTP-заголовків, обмеження частоти запитів (rate limiting) для протидії підбору паролів і автоматизованим атакам, а також застосування протоколу HTTPS для шифрування трафіку. Сукупність цих заходів утворює багаторівневий захист, у якому компрометація одного механізму не призводить до повного порушення безпеки системи [10].

2.3. Принципи інтеграції платіжних сервісів

Інтеграція платіжного сервісу є центральним технічним елементом інтернет-магазину. Платіжний сервіс (платіжний шлюз) — це посередник між магазином, покупцем і банківською системою, який бере на себе приймання даних картки, проведення транзакції, боротьбу із шахрайством та відповідність стандарту PCI DSS. Завдяки цьому магазин не зберігає й не обробляє повні дані банківських карток, а взаємодіє з платіжним сервісом лише через його програмний інтерфейс. Серед міжнародних платіжних сервісів широко використовується Stripe — провайдер, що надає докладно документований API, офіційні бібліотеки (SDK) для

різних мов програмування та режим тестування, який дозволяє повністю відтворити процес оплати без реальних транзакцій за допомогою тестових карток [7].

Сучасна модель інтеграції Stripe побудована навколо поняття платіжного наміру (Payment Intent) — серверного об'єкта, що представляє намір отримати від покупця певну суму та супроводжує платіж упродовж усього його життєвого циклу. Загальна послідовність взаємодії така: магазин на стороні сервера створює платіжний намір на суму замовлення й отримує його клієнтський секрет (client secret); цей секрет передається клієнтському застосунку, який за допомогою бібліотеки Stripe.js відображає захищену форму введення картки — компонент Payment Element; покупець вводить дані картки безпосередньо в інтерфейс Stripe, не передаючи їх на сервер магазину; Stripe проводить транзакцію та підтверджує платіж. Узагальнену схему потоку оплати через Stripe Payment Intent наведено на рисунку (див. рис. 2.3) [8].

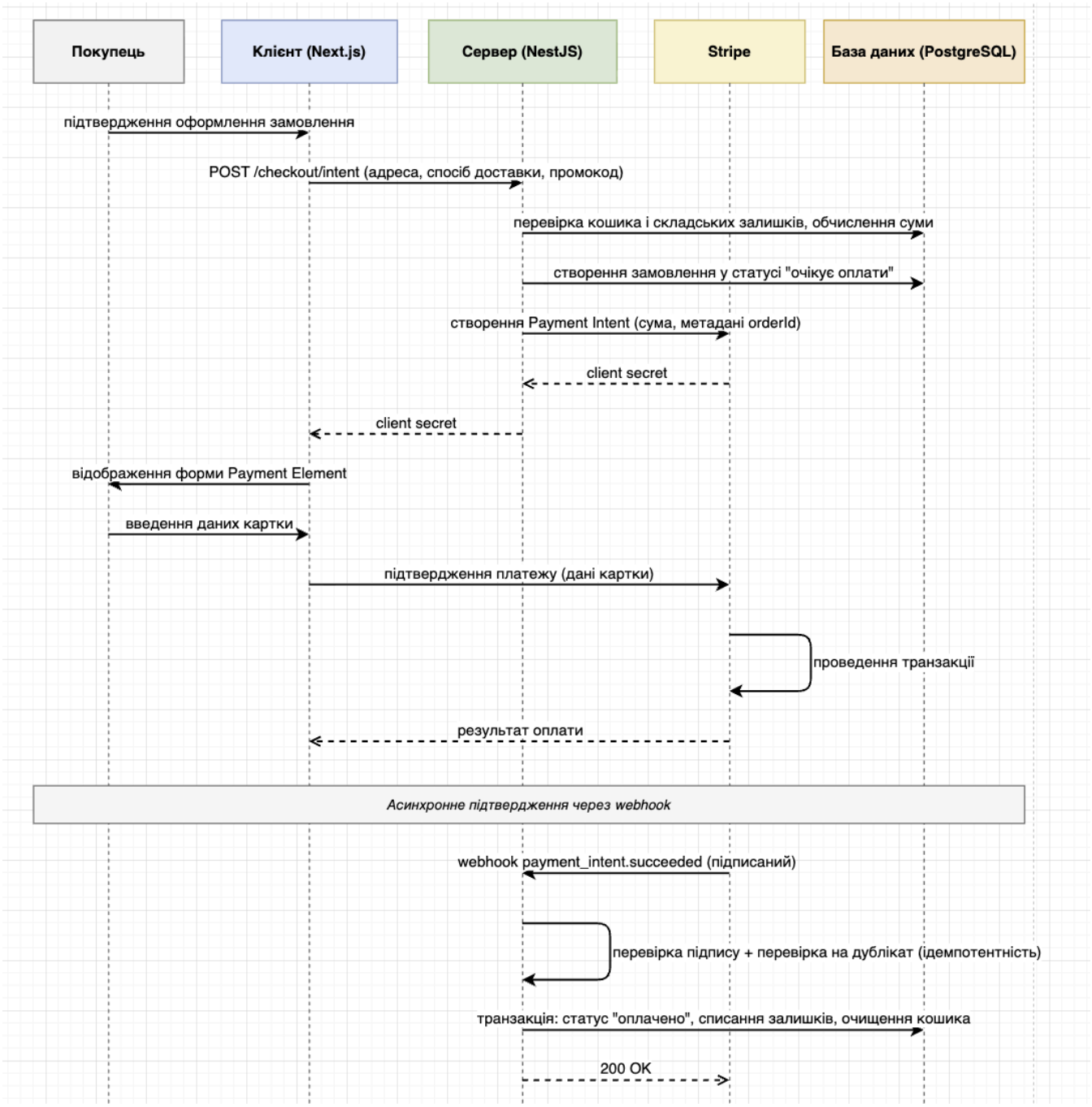

Рисунок 2.3 – Схема потоку оплати замовлення через Stripe Payment Intent

Принципово важливим є те, що остаточним джерелом істини про результат платежу слугує не відповідь, отримана клієнтом, а асинхронне повідомлення від платіжного сервісу — webhook. Webhook — це HTTP-запит, який платіжний сервіс надсилає на заздалегідь визначену адресу магазину при настанні певних подій: успішної оплати, невдалого платежу, повернення коштів. Обробка webhook вимагає дотримання двох ключових вимог. Перша — перевірка криптографічного підпису запиту, що гарантує походження повідомлення саме від платіжного сервісу й

виключає підробку. Друга — ідемпотентність: оскільки платіжний сервіс може повторно надіслати ту саму подію, магазин повинен зберігати ідентифікатори вже оброблених подій і ігнорувати дублікати, щоб уникнути повторного списання товарних залишків чи дублювання замовлень.

Для порівняння можливостей платіжних сервісів, доступних розробникам, складено таблицю 2.1, у якій зіставлено міжнародний сервіс Stripe з популярними українськими провайдерами LiqPay, Fondy та WayForPay за критеріями якості документації, наявності офіційних SDK, моделі інтеграції та режиму тестування.

Таблиця 2.1 — Порівняння платіжних сервісів для інтеграції з інтернет-магазином

Критерій	**Stripe**	**LiqPay**	**Fondy**	**WayForPay**
Докладна англомовна документація	+	частково	+	частково
Офіційні SDK для популярних мов	+	частково	+	частково
Модель платіжного наміру (Payment Intent)	+	–	–	–
Готовий UI-компонент для введення картки	+	+	+	+
Підписані webhook про результат платежу	+	+	+	+
Повноцінний режим тестування	+	+	+	+

Як видно з таблиці 2.1, Stripe вирізняється зрілою моделлю платіжного наміру, якісною документацією та офіційними бібліотеками, що робить його зручним вибором для навчально-практичної розробки. Саме тому в цій роботі для інтеграції платіжних сервісів обрано Stripe у режимі тестування, який дозволяє повністю відтворити потік оплати за допомогою тестових карток без реальних фінансових транзакцій.

2.4. PostgreSQL та контейнеризація як основа інфраструктури магазину

Дані інтернет-магазину — користувачі, книги, категорії, замовлення, відгуки — мають виражену реляційну природу з численними зв'язками між сутностями, тому оптимальним сховищем для них є реляційна система управління базами даних. У роботі обрано PostgreSQL — потужну відкриту об'єктно-реляційну СУБД, що підтримує сувору типізацію, обмеження цілісності, транзакції з властивостями ACID, індекси різних типів, тип даних JSON та має репутацію надійного рішення промислового рівня. Транзакційність PostgreSQL є критично важливою для електронної комерції: операції, що змінюють кілька таблиць одночасно (наприклад, переведення замовлення в статус «оплачено», списання товарних залишків і запис руху складу), повинні виконуватися атомарно — або повністю, або не виконуватися взагалі [5].

Структура бази даних описується у вигляді типізованих моделей за допомогою ORM, а зміни схеми оформлюються у вигляді міграцій — послідовних версіонованих кроків, що дозволяють відтворювано приводити базу до потрібного стану на будь-якому середовищі. Нормалізація схеми до третьої нормальної форми усуває надлишковість даних та аномалії оновлення, тоді як зв'язки «багато-до-багатьох» (наприклад, між книгами й авторами або книгами й категоріями) реалізуються через проміжні таблиці.

Сучасний інтернет-магазин складається не лише з бази даних та серверного застосунку, а й з низки допоміжних сервісів: кешу та сховища черг, пошукового рушія, об'єктного сховища файлів, поштового сервісу та зворотного проксі-сервера. Розгортання такої кількості компонентів вручну є складним і погано відтворюваним процесом. Для його вирішення застосовують контейнеризацію — технологію, що дозволяє упакувати кожен сервіс разом з його залежностями в ізольований контейнер. Інструмент Docker та засіб оркестрації Docker Compose дають змогу описати весь набір сервісів магазину в єдиному декларативному файлі та запустити

повну інфраструктуру однією командою, що забезпечує відтворюваність середовища розробки, тестування й експлуатації [11].

Допоміжними сервісами в архітектурі магазину виступають: Redis — швидке сховище в оперативній пам'яті, що використовується для черг фонових задач; Meilisearch — легкий пошуковий рушій з підтримкою толерантності до помилок у запитах; об'єктне сховище, сумісне з протоколом S3, для зберігання обкладинок книг; та SMTP-сервіс для надсилання листів покупцям [12, 13]. Винесення тривалих операцій (надсилання електронних листів, оновлення пошукового індексу) у фонові черги задач дозволяє серверному API швидко відповідати на запити користувача, не очікуючи завершення цих операцій. Сукупність розглянутих теоретичних засад — клієнт-серверної архітектури, механізмів автентифікації, принципів інтеграції платіжних сервісів та інфраструктурних рішень — становить основу для практичного проєктування і реалізації системи, описаних у наступному розділі.

3. ПРАКТИЧНА ЧАСТИНА

3.1. Загальна архітектура системи та структура проєкту

Розроблений веб-сайт електронного магазину книг отримав назву «BookNest». Систему побудовано як монорепозиторій, що містить два самостійні застосунки — клієнтський (web) та серверний (api) — а також спільну інфраструктурну конфігурацію. Клієнтський застосунок реалізовано на фреймворку Next.js, серверний — на фреймворку NestJS. Усі компоненти системи розгортаються у вигляді контейнерів Docker, описаних у єдиному файлі docker-compose.yml, що дозволяє запустити повну інфраструктуру магазину однією командою.

До складу інфраструктури входять дев'ять сервісів, кожен з яких виконує власну роль у системі. Перелік сервісів та їх призначення наведено в таблиці 3.1.

Таблиця 3.1 — Сервіси інфраструктури системи BookNest

Сервіс	**Призначення**
caddy	Зворотний проксі-сервер, що маршрутизує запити до клієнтського застосунку, API та сховища файлів
web	Клієнтський застосунок на Next.js
api	Серверне REST API на NestJS
worker	Фоновий обробник черг задач (надсилання листів, індексація)
postgres	Реляційна СУБД PostgreSQL — основне сховище даних
redis	Сховище в пам'яті для черг фонових задач
meilisearch	Пошуковий рушій для повнотекстового пошуку книг
minio	Об'єктне сховище, сумісне з S3, для обкладинок книг
mailhog	SMTP-сервіс для приймання та перегляду листів магазину

Загальну архітектуру системи з усіма сервісами та зв'язками між ними наведено на рисунку (див. рис. 3.1). Зовнішні запити надходять на зворотний проксі-сервер Caddy, який розподіляє їх між клієнтським застосунком, серверним API та сховищем файлів. Серверне API взаємодіє з базою даних PostgreSQL, кешем

Redis, пошуковим рушієм Meilisearch, об'єктним сховищем MinIO та зовнішнім платіжним сервісом Stripe.

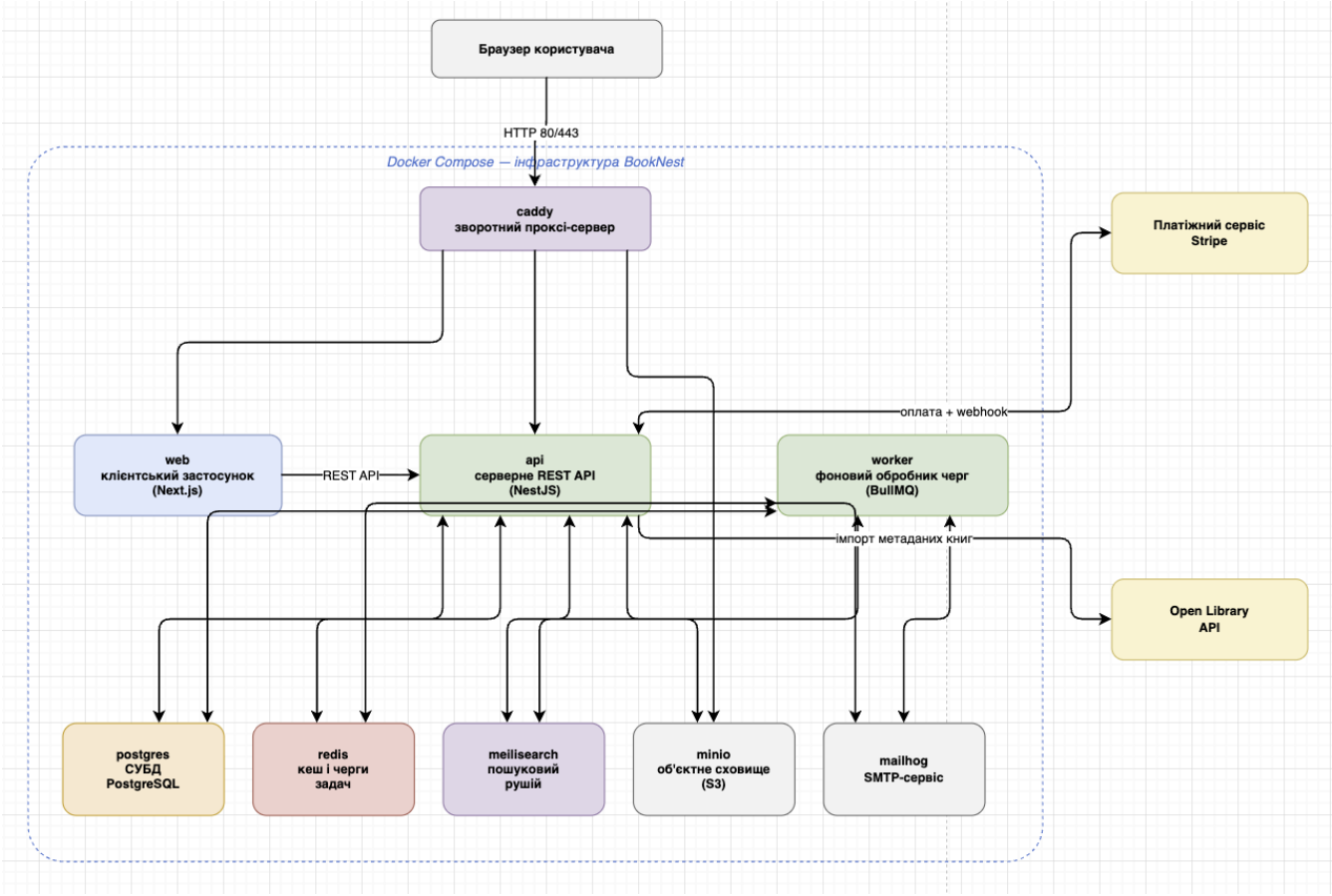

Рисунок 3.1 – Загальна архітектура системи BookNest

Опис інфраструктури у файлі docker-compose.yml має декларативний характер. Як приклад, нижче наведено фрагмент опису сервісу бази даних PostgreSQL.

```
postgres:
image: postgres:16-alpine
restart: unless-stopped
environment:
POSTGRES_USER: ${POSTGRES_USER}
POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
POSTGRES_DB: ${POSTGRES_DB}
volumes:
- postgres_data:/var/lib/postgresql/data
```

```
healthcheck:
test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER} -d
${POSTGRES_DB}"]
interval: 5s
```

Для кожного сервісу задано образ контейнера, змінні середовища, постійні томи для зберігання даних та перевірку працездатності (healthcheck). Механізм healthcheck та залежностей depends_on забезпечує правильний порядок запуску: серверне API стартує лише після того, як база даних, кеш і об'єктне сховище повідомлять про готовність. Сервіс seed виконується одноразово й автоматично наповнює каталог магазину книгами, завантаженими з відкритого джерела Open Library.

Серверний застосунок має модульну структуру: кожен функціональний напрям виділено в окремий модуль (auth, books, cart, checkout, payments, orders, reviews, promo, admin та інші), а спільна логіка винесена у модулі prisma, search, storage та mail. Кожен модуль містить контролер, що описує HTTP-маршрути, сервіс з бізнес-логікою та об'єкти передавання даних з правилами валідації. Клієнтський застосунок організовано за файловою маршрутизацією Next.js: каталог сторінок [locale] містить публічні сторінки магазину, особистий кабінет та адміністративну панель, а спільні елементи інтерфейсу винесено в каталог компонентів.

3.2. Проєктування схеми бази даних

Схему бази даних спроєктовано як реляційну й описано засобами ORM Prisma у вигляді типізованих моделей. Схема нормалізована до третьої нормальної форми та налічує дев'ятнадцять моделей, що покривають усі предметні сутності магазину: користувачів та їх адреси, токени автентифікації, книги, авторів, категорії, видавництва, кошик, список бажань, замовлення та їх позиції, відгуки, промокоди, рух складських залишків, журнал платіжних подій та журнал аудиту [6].

Центральною сутністю каталогу є книга. Нижче наведено фрагмент опису моделі Book у схемі Prisma.

```
model Book {
id            String       @id @default(cuid())
slug          String       @unique
isbn13        String?      @unique
titleUk       String
titleEn       String
priceUah      Int
oldPriceUah   Int?
stock         Int          @default(0)
language      BookLanguage @default(UK)
rating        Float        @default(0)
isActive      Boolean      @default(true)
authors       BookAuthor[]
categories    BookCategory[]
reviews       Review[]
}
```

Кожна книга має унікальний текстовий ідентифікатор slug для формування зрозумілих адрес сторінок, назву й опис двома мовами, ціну, що зберігається в копійках у цілочисловому полі для уникнення похибок округлення, кількість на складі та ознаку активності. Зв'язки книги з авторами та категоріями реалізовано за моделлю «багато-до-багатьох» через проміжні таблиці BookAuthor і BookCategory.

Замовлення є ключовою транзакційною сутністю магазину. Модель Order містить унікальний номер замовлення, посилання на користувача, статус, суми (проміжну, знижку, вартість доставки й підсумкову), посилання на застосований промокод, адресу доставки у форматі JSON, контактні дані та поля інтеграції з платіжним сервісом — ідентифікатор платіжного наміру Stripe та ідентифікатор транзакції. Життєвий цикл замовлення описується переліком статусів: «очікує», «очікує оплати», «оплачено», «обробляється», «відправлено», «доставлено», «скасовано», «повернено». Позиції замовлення зберігаються в окремій таблиці OrderItem зі знімком назви та ціни книги на момент покупки, що гарантує незмінність історичних даних замовлення навіть у разі подальшої зміни ціни товару.

Для забезпечення ідемпотентної обробки платіжних подій передбачено модель WebhookEvent з унікальним обмеженням на пару «провайдер — зовнішній ідентифікатор події», а для журналювання дій адміністратора — модель AuditLog. Повну схему бази даних у вигляді діаграми «сутність — зв'язок» наведено на рисунку (див. рис. 3.2).

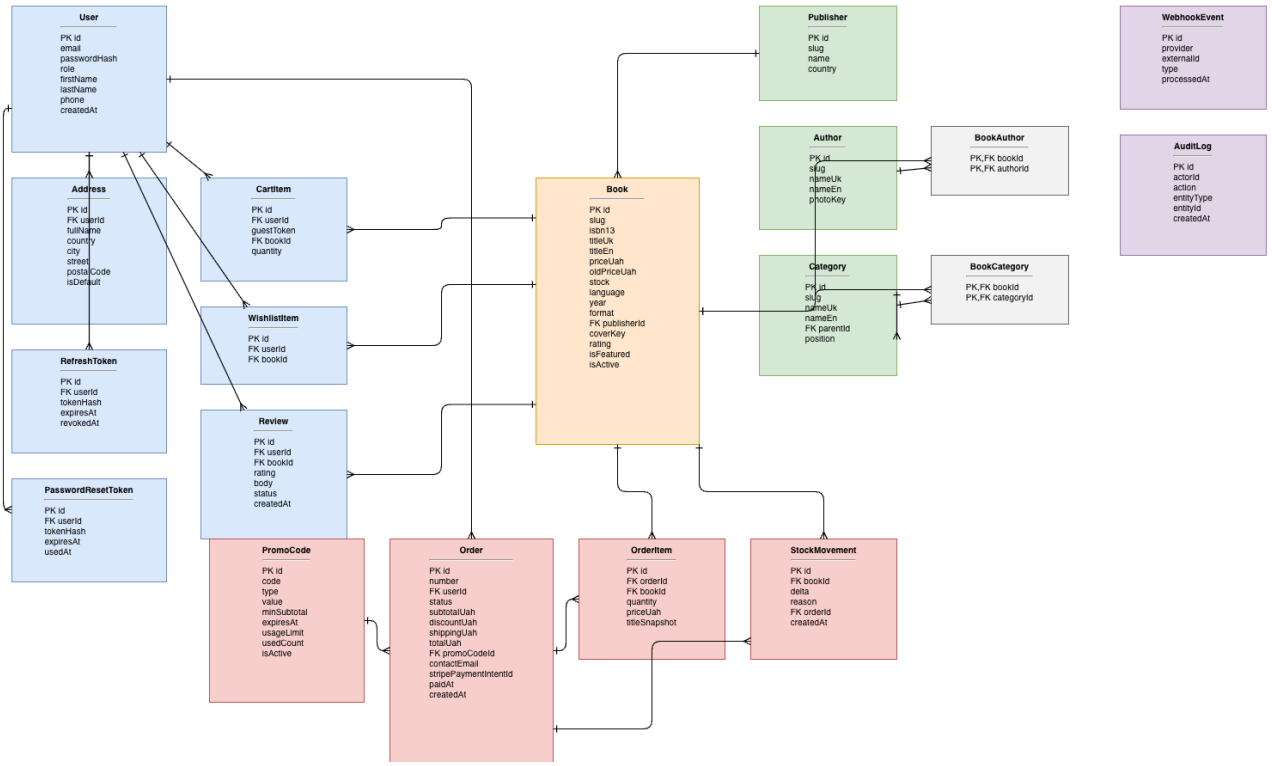

Рисунок 3.2 – Діаграма «сутність – зв'язок» бази даних системи BookNest

Зміни схеми оформлено у вигляді версіонованих міграцій Prisma, що застосовуються до бази автоматично під час розгортання. Початкове наповнення бази (обліковий запис адміністратора, категорії, промокоди, книги та відгуки) виконує окремий сценарій seed, який звертається до відкритого API Open Library для отримання реальних метаданих та обкладинок книг [15].

3.3. Реалізація модуля каталогу та повнотекстового пошуку

Модуль каталогу відповідає за перегляд книг із фільтрацією, сортуванням та пагінацією. Серверний сервіс BooksService формує запит до бази даних на основі набору фільтрів, переданих клієнтом: категорія, автор, діапазон ціни, мова, наявність на складі, ознаки рекомендованих книг і бестселерів. Нижче наведено фрагмент формування умови вибірки.

```
const where: Prisma.BookWhereInput = { isActive: true };
if (filters.category) {
where.categories = { some: { category: { slug: filters.category } } };
}
if (filters.priceMin !== undefined)
where.priceUah = { ...(where.priceUah as object), gte:
filters.priceMin };
if (filters.inStock) where.stock = { gt: 0 };
```

Запит до бази виконується через ORM Prisma параметризованими викликами, що виключає можливість SQL-ін'єкцій. Перелік книг та їх загальна кількість для пагінації отримуються в межах однієї транзакції, після чого результат перетворюється у формат відповіді API з обчисленням повної адреси обкладинки та переліку авторів і категорій кожної книги.

Головна сторінка магазину відображає рекомендовані книги, новинки та бестселери; її зовнішній вигляд наведено на рисунку (див. рис. 3.3). Сторінку каталогу з бічною панеллю фільтрів, сортуванням і пагінацією наведено на рисунку (див. рис. 3.4).

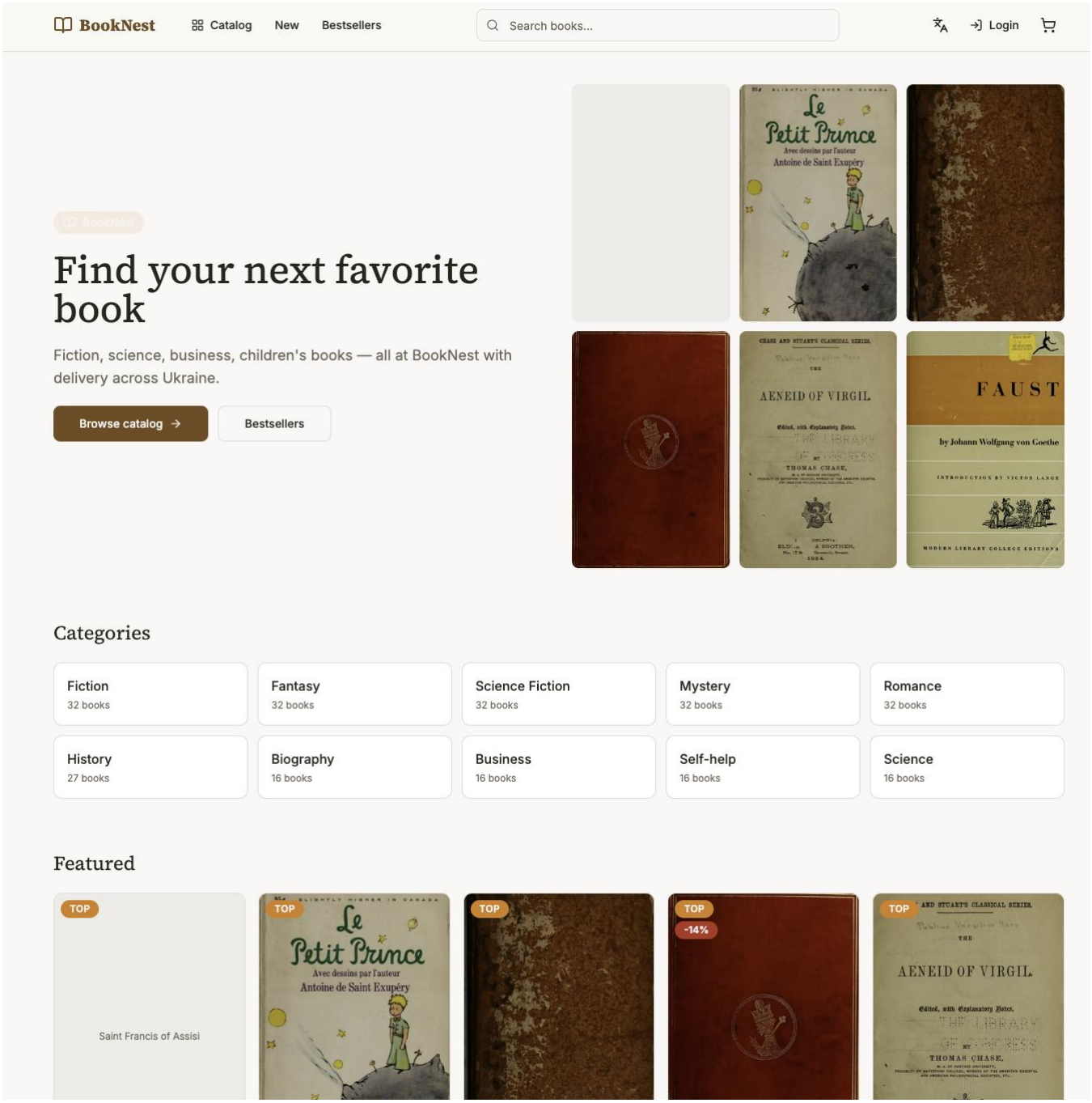

Рисунок 3.3 – Головна сторінка магазину BookNest

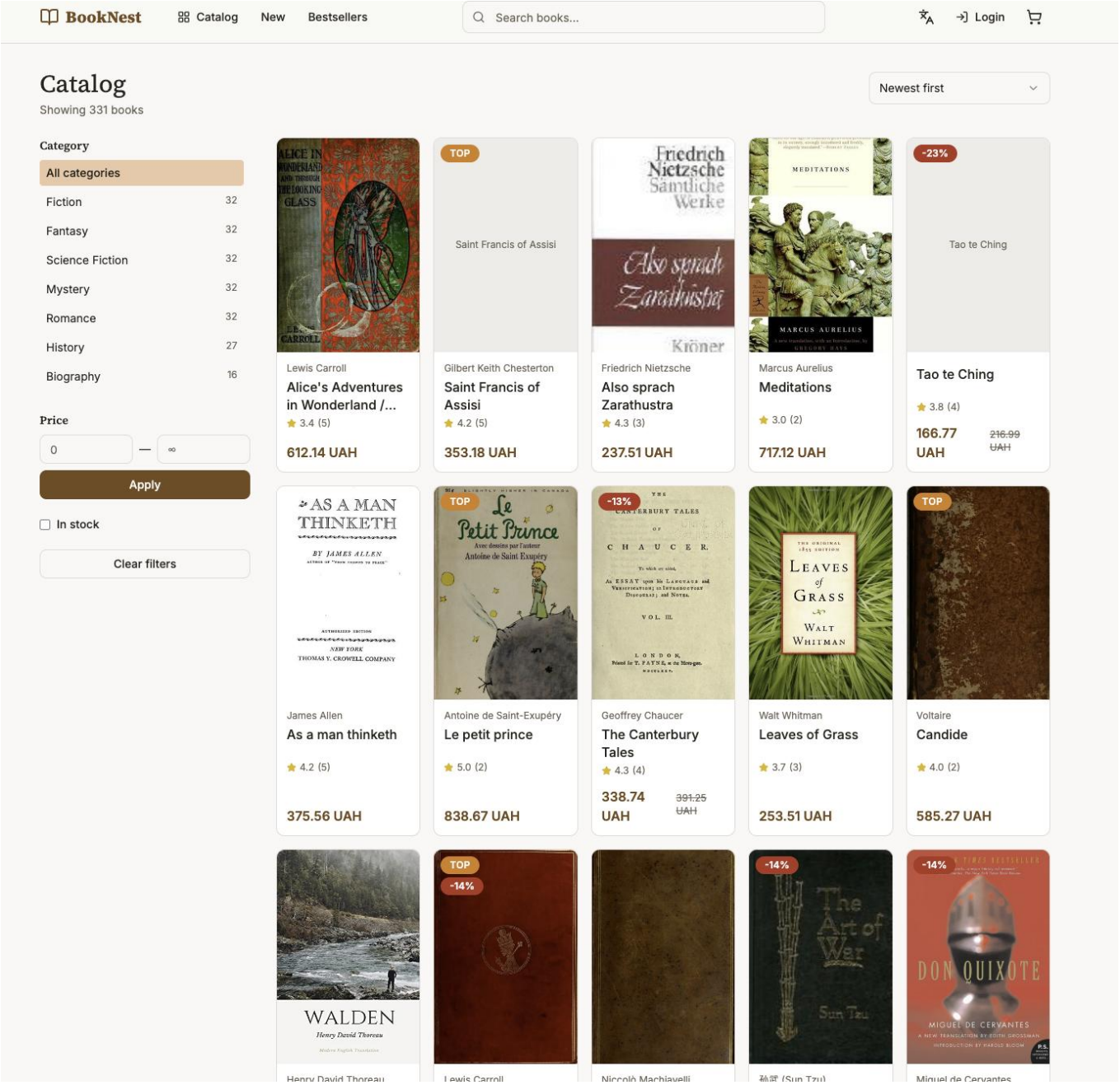

Рисунок 3.4 – Сторінка каталогу з фільтрами та сортуванням

Повнотекстовий пошук реалізовано на основі окремого пошукового рушія Meilisearch, винесеного в самостійний сервіс SearchService. Під час старту системи сервіс створює пошуковий індекс books та налаштовує його: визначає атрибути, за якими ведеться пошук (назви, автори, описи), атрибути для фільтрації й сортування, а також вмикає толерантність до помилок у запитах.

```
await this.client.index(this.index).updateSettings({
searchableAttributes: ['titleUk', 'titleEn', 'authors',
'descriptionUk'],
```

```
filterableAttributes: ['categories', 'language', 'isActive'],
sortableAttributes: ['priceUah', 'rating', 'createdAt'],
typoTolerance: { enabled: true },
});
```

Завдяки толерантності до помилок користувач отримує релевантні результати навіть за наявності одруку в запиті. Наповнення та оновлення пошукового індексу виконується сценарієм seed і фоновим обробником, що дозволяє серверному API не залежати від доступності пошукового рушія під час обробки звичайних запитів. Детальну сторінку книги з описом, ціною, кнопкою додавання до кошика та блоком відгуків наведено на рисунку (див. рис. 3.5).

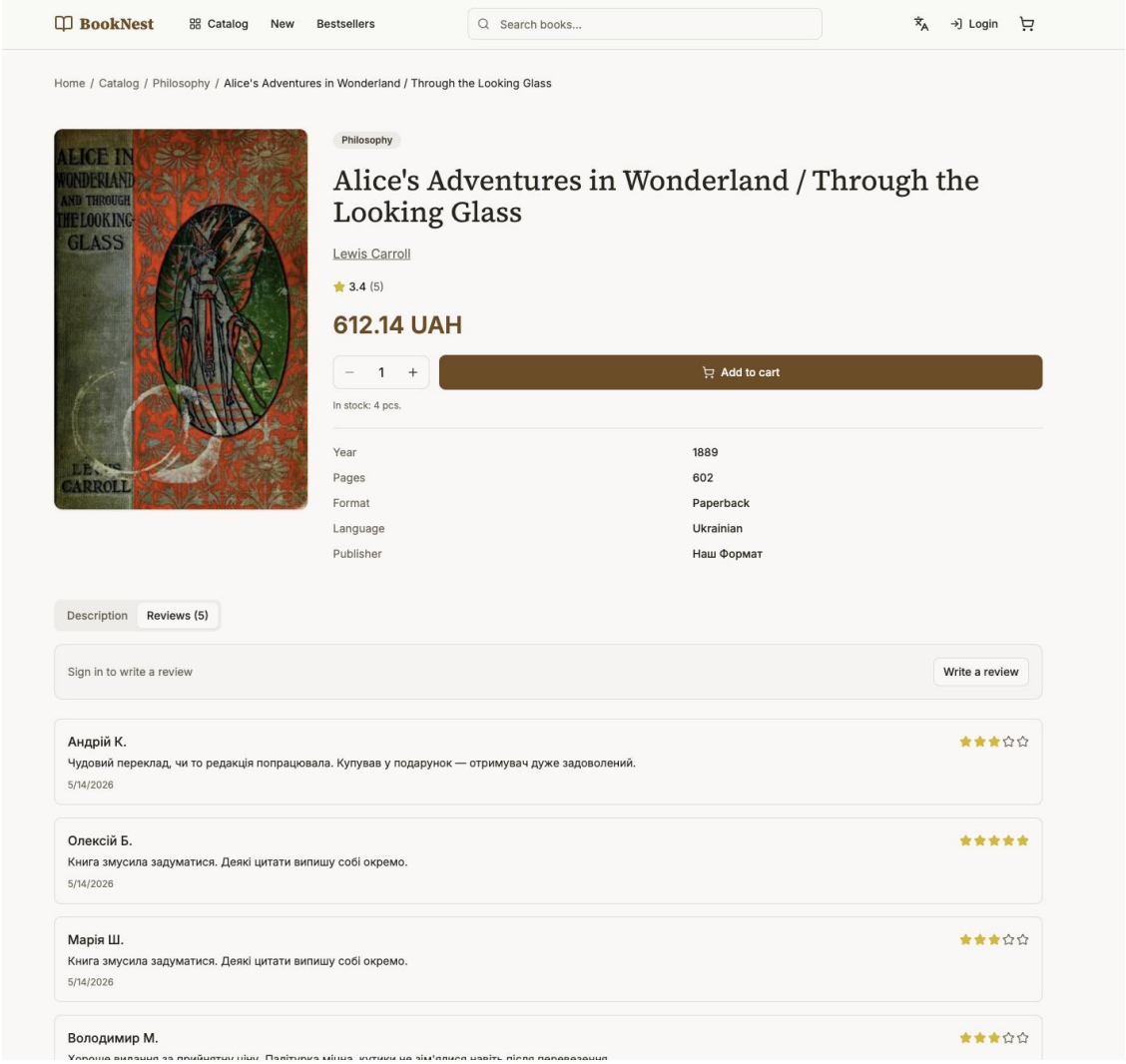

Рисунок 3.5 – Детальна сторінка книги з відгуками

Таким чином, модуль каталогу поєднує гнучку фільтрацію засобами реляційної бази даних з якісним повнотекстовим пошуком, делегованим спеціалізованому рушію, що забезпечує зручний для покупця процес знаходження потрібної книги.

3.4. Реалізація модуля автентифікації та безпеки

Модуль автентифікації побудовано за схемою двох токенів JWT. Сервіс AuthService реалізує реєстрацію, вхід, оновлення токенів, вихід, скидання та зміну пароля. При реєстрації пароль користувача хешується алгоритмом Argon2id, після чого створюється обліковий запис і видається пара токенів.

```
const passwordHash = await argon2.hash(dto.password, { type:
argon2.argon2id });
const user = await this.prisma.user.create({
data: {
email: dto.email.toLowerCase(),
passwordHash,
firstName: dto.firstName,
lastName: dto.lastName,
},
});
```

Пароль ніколи не зберігається у відкритому вигляді — у базі даних залишається лише його незворотний хеш. Під час входу введений пароль перевіряється функцією argon2.verify проти збереженого хешу; у разі невдачі повертається уніфіковане повідомлення про помилку без розкриття того, чи існує користувач із вказаною адресою.

Видачу, ротацію та відкликання токенів інкапсульовано в сервісі TokensService. Access-токен підписується секретним ключем і має короткий час життя. Refresh-токен генерується як випадковий рядок, а в базі даних зберігається лише його хеш за алгоритмом SHA-256. Ключовим є метод ротації refresh-токена.

```
async rotate(rawToken: string) {
const tokenHash = createHash('sha256').update(rawToken).digest('hex');
```

```
const record = await this.prisma.refreshToken.findUnique({ where: {
tokenHash } });
if (!record || record.revokedAt || record.expiresAt < new Date())
return null;
await this.prisma.refreshToken.update({
where: { tokenHash }, data: { revokedAt: new Date() },
});
const next = await this.issueRefresh(record.userId);
return { userId: record.userId, ...next };
}
```

При кожному оновленні старий refresh-токен позначається відкликаним, а користувачеві видається новий. Якщо викрадений токен буде використано повторно, ротація поверне відмову, оскільки запис уже позначено відкликаним. Токени передаються клієнту у захищених cookie з атрибутами HttpOnly та SameSite, що унеможливлює доступ до них зі сторонніх скриптів.

Перевірку access-токена реалізовано через стратегію JwtStrategy на основі бібліотеки Passport, а розмежування доступу між покупцем та адміністратором — через RolesGuard, що перевіряє роль користувача в корисному навантаженні токена. На рівні всього застосунку діє низка додаткових захисних заходів: захисні HTTP-заголовки встановлюються бібліотекою Helmet, обмеження частоти запитів реалізовано через ThrottlerGuard (зокрема, не більше п'яти спроб реєстрації за хвилину), усі вхідні дані проходять валідацію через DTO, а звернення до бази виконуються виключно параметризованими запитами. Сукупність цих рішень утворює багаторівневий захист застосунку.

3.5. Реалізація модуля кошика

Модуль кошика спроєктовано так, щоб користувач міг наповнювати кошик ще до реєстрації. Для цього кожній позиції кошика відповідає або ідентифікатор зареєстрованого користувача, або гостьовий токен — випадковий ідентифікатор, що зберігається в браузері відвідувача. Сервіс CartService визначає власника кошика і виконує над ним операції додавання, оновлення кількості, видалення позиції та очищення.

```
async addItem(owner: Owner, bookId: string, quantity: number) {
const book = await this.prisma.book.findUnique({ where: { id: bookId }
});
if (!book || !book.isActive) throw new NotFoundException('Book not
found');
if (book.stock < quantity) throw new BadRequestException('Not enough
stock');
await this.prisma.cartItem.upsert({
where: { userId_bookId: { userId: owner.userId, bookId } },
update: { quantity: { increment: quantity } },
create: { userId: owner.userId, bookId, quantity },
});
}
```

Перед додаванням книги до кошика перевіряється її наявність та достатність складського залишку. Операція upsert додає нову позицію або збільшує кількість уже наявної, що спрощує логіку повторного додавання тієї самої книги.

Окремої уваги потребує сценарій, коли гість, що вже наповнив кошик, виконує вхід або реєстрацію. У цьому випадку спрацьовує механізм злиття кошиків: позиції гостьового кошика переносяться до кошика користувача, а гостьовий кошик очищується.

```
private async mergeGuestCart(userId: string, guestToken: string) {
const guestItems = await this.prisma.cartItem.findMany({ where: {
guestToken } });
for (const item of guestItems) {
```

```
await this.prisma.cartItem.upsert({
where: { userId_bookId: { userId, bookId: item.bookId } },
update: { quantity: { increment: item.quantity } },
create: { userId, bookId: item.bookId, quantity: item.quantity },
});
}
await this.prisma.cartItem.deleteMany({ where: { guestToken } });
}
```

Завдяки механізму злиття користувач не втрачає обраних товарів при вході в систему, що позитивно впливає на конверсію магазину. На стороні клієнта стан гостьового кошика синхронізується з сервером, а кількість товарів у кошику відображається в шапці сайту.

3.6. Реалізація оформлення замовлення та інтеграції з платіжним сервісом Stripe

Оформлення замовлення є центральним сценарієм системи, що поєднує бізнес-логіку магазину з інтеграцією платіжного сервісу. Серверний сервіс CheckoutService обробляє запит на створення платіжного наміру: він отримує вміст кошика користувача, повторно перевіряє активність книг та достатність складських залишків, обчислює проміжну суму, застосовує промокод за наявності, визначає вартість доставки та підсумкову суму замовлення.

```
const subtotal = cartItems.reduce(
(s, it) => s + it.book.priceUah * it.quantity, 0);
let discount = 0;
if (input.promoCode) {
const v = await this.promo.validate(input.promoCode, subtotal);
discount = v.discount;
}
const shipping = subtotal >= FREE_SHIPPING_THRESHOLD ? 0 :
rawShipping;
const total = subtotal - discount + shipping;
```

Після обчислення сум у базі даних створюється замовлення у статусі «очікує оплати» разом з його позиціями, у яких зберігається знімок назви та ціни кожної книги. Лише після цього виконується звернення до платіжного сервісу.

Взаємодію зі Stripe інкапсульовано в сервісі StripeService. Метод createPaymentIntent створює платіжний намір на підсумкову суму замовлення з прив'язкою ідентифікатора замовлення в метаданих платежу.

```
const intent = await this.client.paymentIntents.create({
amount: args.amount,
currency: args.currency,
automatic_payment_methods: { enabled: true },
receipt_email: args.customerEmail,
metadata: { orderId: args.orderId, orderNumber: args.orderNumber },
});
```

Сервіс StripeService додатково підтримує режим імітації: якщо тестові ключі не налаштовано, платіжний намір створюється локально, що дозволяє демонструвати потік оформлення замовлення навіть без підключення до Stripe. Отриманий від платіжного наміру клієнтський секрет передається клієнтському застосунку.

На стороні клієнта на сторінці оформлення замовлення відображається захищена форма введення даних картки — компонент Payment Element з бібліотеки Stripe.js. Дані картки вводяться безпосередньо в інтерфейс Stripe і не передаються на сервер магазину. Сторінку оформлення замовлення з формою Payment Element наведено на рисунку (див. рис. 3.6).

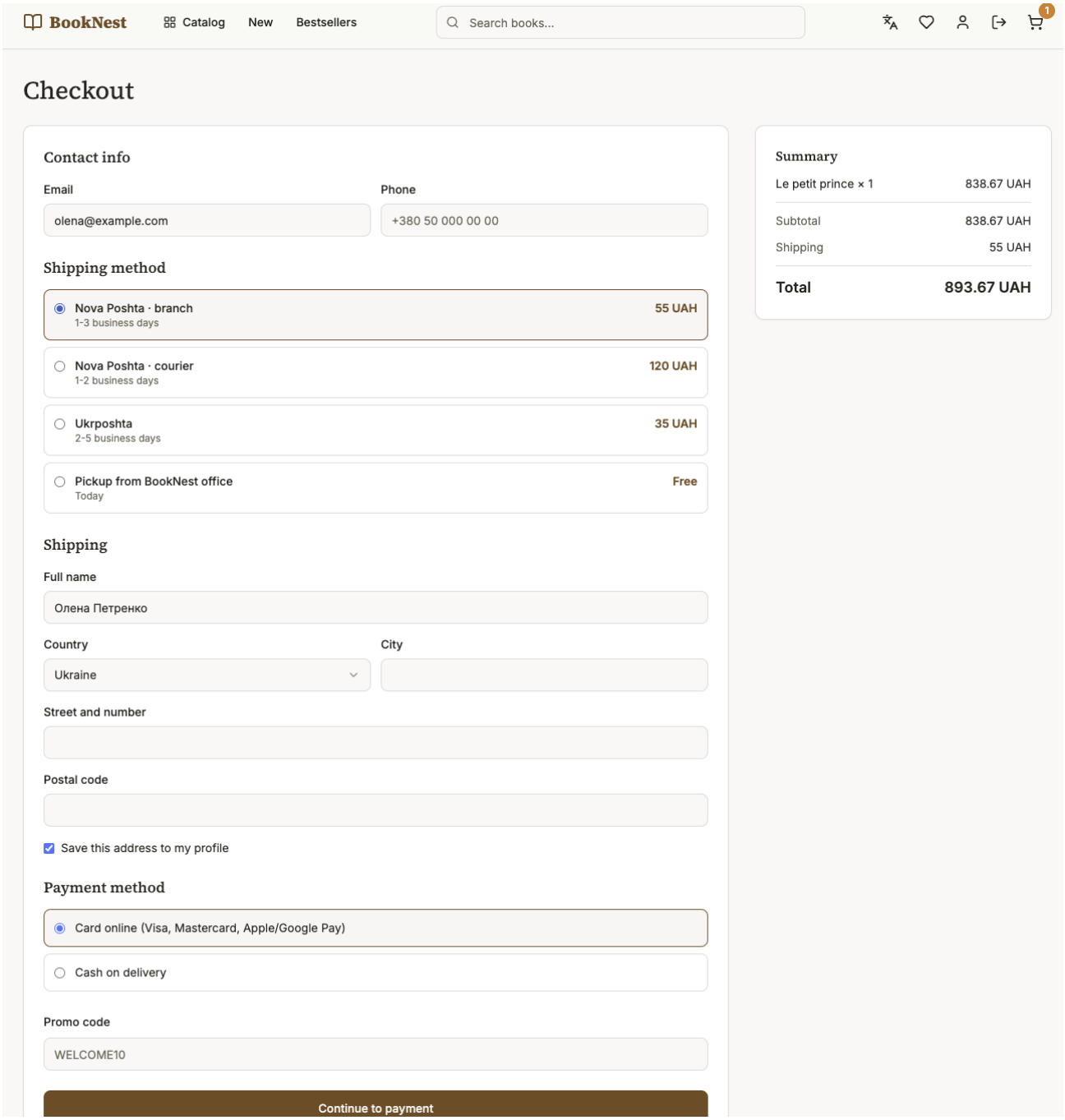

Рисунок 3.6 – Сторінка оформлення замовлення з формою оплати Stripe Payment Element

Остаточним підтвердженням оплати слугує асинхронний webhook від Stripe, який обробляє контролер WebhookController. Обробка побудована з дотриманням двох вимог — перевірки підпису та ідемпотентності.

```
const existing = await this.prisma.webhookEvent.findUnique({
where: { provider_externalId: { provider: 'stripe', externalId:
event.id } },
```

```
});
if (existing) return { received: true, duplicate: true };
await this.prisma.webhookEvent.create({
data: { provider: 'stripe', externalId: event.id, type: event.type,
payload: event },
});
```

Підпис запиту перевіряється методом constructEvent, що гарантує походження повідомлення від Stripe. Ідентифікатор кожної обробленої події зберігається в таблиці WebhookEvent з унікальним обмеженням, тож повторне надходження тієї самої події розпізнається як дублікат і ігнорується. При успішній оплаті в межах однієї транзакції бази даних виконуються списання складських залишків, запис руху складу, переведення замовлення у статус «оплачено», очищення кошика користувача та збільшення лічильника використань промокоду, після чого покупцеві надсилається лист-підтвердження. Атомарність транзакції гарантує, що замовлення не залишиться у неузгодженому стані навіть у разі збою. Сторінку успішного завершення оплати наведено на рисунку (див. рис. 3.7).

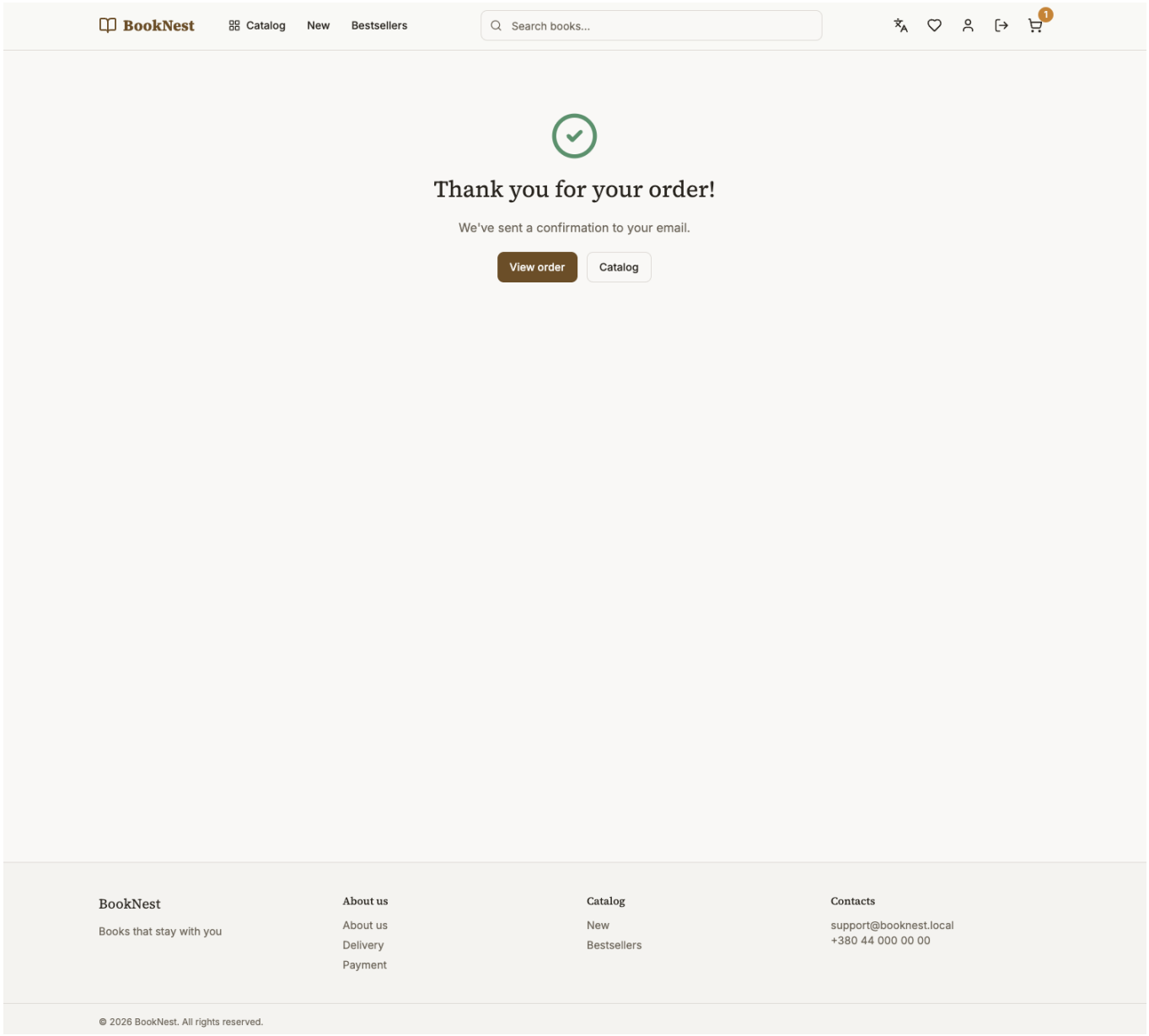

Рисунок 3.7 – Сторінка успішного завершення оплати замовлення

Реалізована інтеграція повністю відповідає сучасній моделі взаємодії з платіжним сервісом: магазин не зберігає даних карток, остаточне рішення про оплату приймається на основі підписаного webhook, а ідемпотентність виключає дублювання замовлень при повторному надходженні подій.

3.7. Адміністративна панель та документація API

Для управління магазином реалізовано окрему адміністративну панель, доступ до якої мають лише користувачі з роллю адміністратора. Панель об'єднує розділи управління книгами, категоріями, авторами, замовленнями, відгуками, користувачами та промокодами, а також головну сторінку зі статистикою. Серверний сервіс AdminService обчислює зведені показники магазину: загальну кількість замовлень, кількість оплачених замовлень, кількість покупців і активних книг, сумарний виторг та перелік останніх замовлень.

```
const revenueAgg = await this.prisma.order.aggregate({
where: { status: { in: ['PAID', 'PROCESSING', 'SHIPPED', 'DELIVERED']
} },
_sum: { totalUah: true },
});
```

Головну сторінку адміністративної панелі зі зведеною статистикою та переліком останніх замовлень наведено на рисунку (див. рис. 3.8). Через розділ управління книгами адміністратор може створювати й редагувати картки книг, завантажувати обкладинки в об'єктне сховище, керувати наявністю та ознаками рекомендованих книг; через розділ замовлень — змінювати їх статуси; через розділ відгуків — модерувати відгуки покупців.

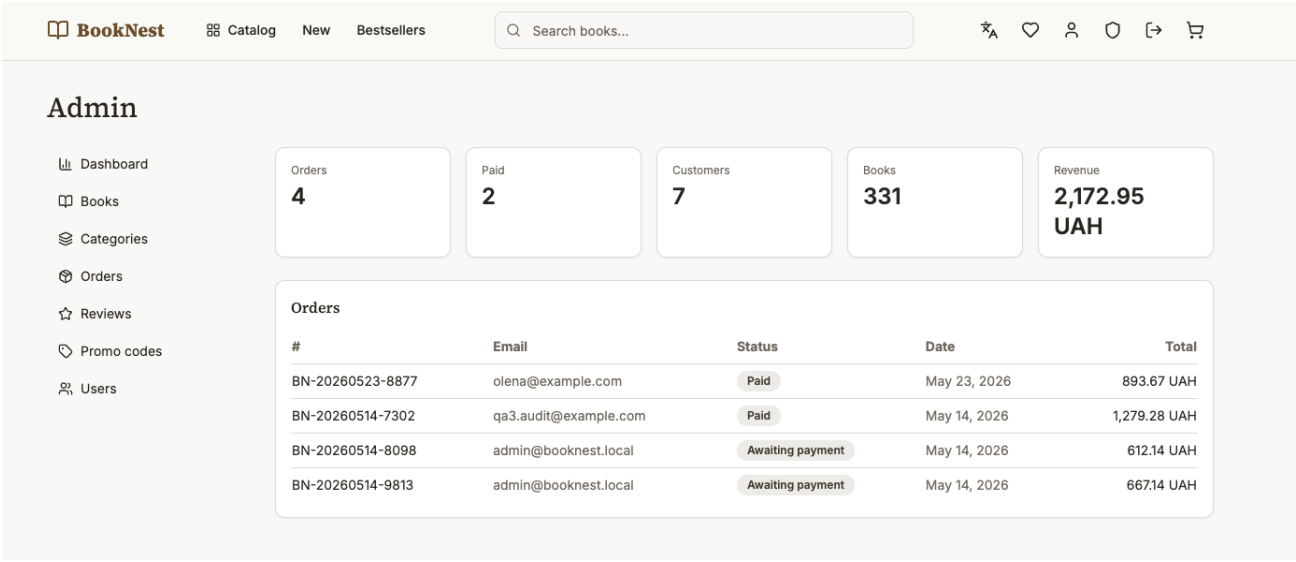

Рисунок 3.8 – Головна сторінка адміністративної панелі зі статистикою

Для всіх маршрутів серверного API автоматично генерується інтерактивна документація на основі специфікації OpenAPI 3.0 засобами модуля @nestjs/swagger. Документація доступна за окремою адресою у вигляді інтерфейсу Swagger UI, який дозволяє розробникам переглядати перелік ендпоінтів, їх параметри та схеми відповідей, а також надсилати тестові запити безпосередньо з браузера. Інтерфейс Swagger UI з документацією API наведено на рисунку (див. рис. 3.9).

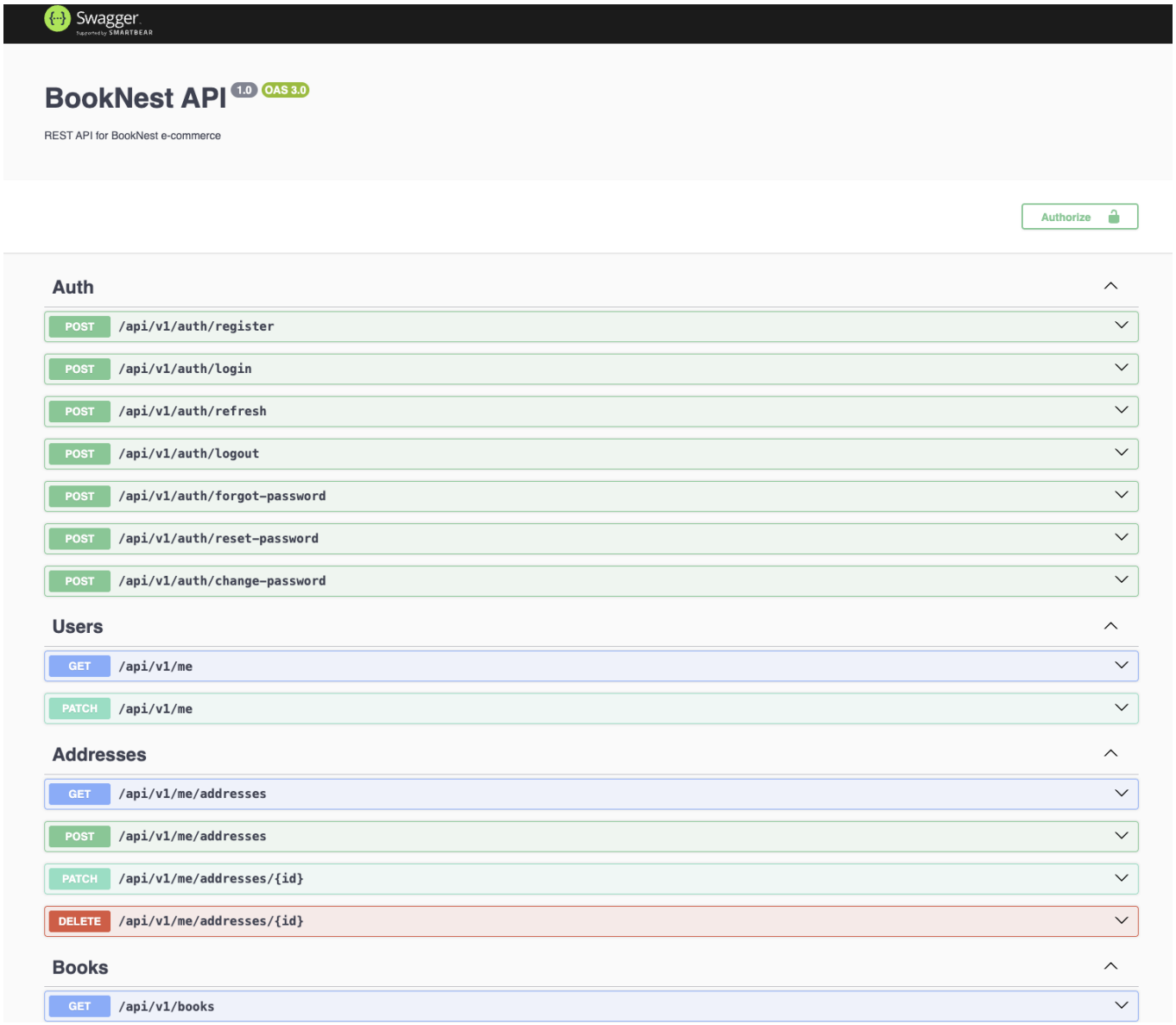

Рисунок 3.9 – Інтерактивна документація API у інтерфейсі Swagger UI

Наявність адміністративної панелі та автоматично згенерованої документації API робить систему придатною як для безпосередньої експлуатації співробітниками магазину, так і для подальшої інтеграції з зовнішніми клієнтськими застосунками.

3.8. Тестування системи та інструкція для користувача

Тестування системи проводилося на двох рівнях. На рівні модульного тестування за допомогою фреймворку Jest перевірялася бізнес-логіка сервісного шару — насамперед обчислення сум замовлення, застосування промокодів та логіка роботи з кошиком. На рівні функціонального тестування вручну перевірялися наскрізні сценарії використання системи.

Ключовим перевіреним сценарієм є повний цикл покупки: реєстрація користувача, додавання книги до кошика, перехід до оформлення замовлення, заповнення адреси доставки, оплата тестовою карткою Stripe з номером 4242 4242 4242 4242, обробка платіжного webhook, переведення замовлення у статус «оплачено» та надходження листа-підтвердження. Окремо перевірялися: коректність злиття гостьового кошика при вході, повторне надходження webhook (перевірка ідемпотентності), застосування промокодів, модерація відгуків та зміна статусів замовлень в адміністративній панелі. Усі перевірені сценарії відпрацювали відповідно до очікуваних результатів.

Інструкція для користувача. Робота з магазином починається з головної сторінки, де представлено рекомендовані книги, новинки та бестселери. Через головне меню або сторінку каталогу користувач переходить до перегляду книг, де може застосувати фільтри за категорією, автором, ціною, мовою та наявністю, скористатися сортуванням і повнотекстовим пошуком у рядку пошуку в шапці сайту. Натиснувши на картку книги, користувач потрапляє на її детальну сторінку з описом, характеристиками та відгуками, звідки може додати книгу до кошика або списку бажань. У кошику можна змінити кількість примірників або видалити позицію. Для оформлення замовлення користувач переходить на сторінку checkout, де вказує контактні дані, адресу та спосіб доставки, за бажанням вводить промокод,

після чого вводить дані картки у захищену форму та підтверджує оплату. Після успішної оплати користувач бачить сторінку підтвердження, а замовлення з'являється в розділі «Мої замовлення» особистого кабінету.

Інструкція для адміністратора. Адміністратор входить у систему зі своїми обліковими даними та отримує доступ до розділу адміністрування. На головній сторінці панелі відображаються зведені показники магазину. У розділі управління книгами адміністратор створює та редагує картки книг, завантажує обкладинки, керує наявністю та ознаками рекомендованих книг і бестселерів. У розділі замовлень переглядає замовлення та змінює їх статуси у міру обробки й відправлення. У розділі відгуків модерує відгуки покупців, схвалюючи або відхиляючи їх. Розділи категорій, авторів, промокодів та користувачів дозволяють підтримувати в актуальному стані відповідні довідники магазину. Розгортання всієї системи виконується однією командою docker compose up після копіювання файлу налаштувань, після чого магазин стає доступним у браузері із вже наповненим каталогом книг.

Результати тестування підтверджують, що реалізована система коректно виконує всі заявлені функції, а ключовий для роботи сценарій оформлення й оплати замовлення з інтеграцією платіжного сервісу працює надійно та відповідає сучасним вимогам безпеки.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи спроєктовано та реалізовано веб-сайт електронного магазину книг «BookNest» з інтеграцією платіжних сервісів, що забезпечує повний цикл онлайн-продажу — від перегляду каталогу до оплати замовлення та його адміністрування.

У результаті роботи було досягнуто таких результатів:

- проаналізовано предметну область електронної комерції у книжковій ніші, визначено ролі учасників, ключові бізнес-процеси та функціональні й нефункціональні вимоги до інтернет-магазину книг;
- виконано огляд та порівняльний аналіз існуючих українських інтернет-магазинів книг, складено таблицю порівняння аналогів та обґрунтовано доцільність власної розробки;
- досліджено теоретичні засади побудови сучасних веб-застосунків: клієнт-серверну архітектуру, принципи побудови REST API, механізми автентифікації на основі JWT та заходи інформаційної безпеки;
- досліджено принципи інтеграції платіжних сервісів, модель платіжного наміру Stripe Payment Intent та механізм асинхронної обробки платіжних подій через webhook з перевіркою підпису та ідемпотентністю;
- обґрунтовано вибір технологічного стеку: фреймворків Next.js і NestJS, СУБД PostgreSQL з ORM Prisma, платіжного сервісу Stripe, пошукового рушія Meilisearch, об'єктного сховища та засобів контейнеризації Docker;
- спроєктовано загальну архітектуру системи з дев'яти контейнеризованих сервісів та реляційну схему бази даних з дев'ятнадцяти моделей, нормалізовану до третьої нормальної форми;
- реалізовано ключові модулі магазину: каталог з фільтрацією та повнотекстовим пошуком, автентифікацію з ротацією refresh-токенів, кошик зі злиттям гостьового та авторизованого режимів, оформлення замовлення з

інтеграцією Stripe, обробку платіжних webhook, замовлення, відгуки та промокоди;

- розроблено адміністративну панель для управління асортиментом, замовленнями, відгуками й користувачами та автоматично згенеровано документацію API на базі специфікації OpenAPI 3.0;
- проведено модульне та функціональне тестування, перевірено наскрізний сценарій оформлення й оплати замовлення тестовою карткою, оформлено інструкцію для користувача та адміністратора.

Розроблена система демонструє сучасний підхід до побудови застосунків електронної комерції: чіткий розподіл клієнтської та серверної частин, типізований код, контейнеризоване розгортання всієї інфраструктури однією командою та безпечну інтеграцію платіжного сервісу, за якої магазин не зберігає даних банківських карток, а остаточне рішення про оплату приймається на основі підписаного й ідемпотентно обробленого webhook.

Практичне значення роботи полягає у створенні готового до використання веб-застосунку, який може бути впроваджений як основа реального інтернет-магазину книг або іншої товарної ніші, а також слугувати систематизованим навчально-практичним прикладом побудови застосунку електронної комерції з інтеграцією платіжних сервісів. Поставлену мету кваліфікаційної роботи досягнуто, усі визначені завдання виконано в повному обсязі.

СПИСОК ІНФОРМАЦІЙНИХ ДЖЕРЕЛ

1. Next.js Documentation. Vercel, Inc. URL: https://nextjs.org/docs
2. NestJS — A progressive Node.js framework. Documentation. URL: https://docs.nestjs.com
3. React Documentation. Meta Open Source. URL: https://react.dev/
4. TypeScript Documentation. Microsoft. URL: https://www.typescriptlang.org/docs/
5. PostgreSQL 16 Documentation. The PostgreSQL Global Development Group, 2023. URL: https://www.postgresql.org/docs/16/index.html
6. Prisma ORM Documentation. Prisma Data, Inc. URL: https://www.prisma.io/docs
7. Stripe API Reference. Stripe, Inc. URL: https://docs.stripe.com/api
8. Stripe Payments — Payment Intents API. Stripe, Inc. URL: https://docs.stripe.com/payments/payment-intents
9. Jones M., Bradley J., Sakimura N. RFC 7519: JSON Web Token (JWT). IETF, 2015. URL: https://datatracker.ietf.org/doc/html/rfc7519
10. OWASP Top Ten 2021. The OWASP Foundation, 2021. URL: https://owasp.org/Top10/
11. Docker Documentation. Docker, Inc. URL: https://docs.docker.com/
12. Redis Documentation. Redis Ltd. URL: https://redis.io/docs/latest/
13. Meilisearch Documentation. Meili SAS. URL: https://www.meilisearch.com/docs
14. Tailwind CSS Documentation. Tailwind Labs Inc. URL: https://tailwindcss.com/docs
15. Open Library Developers Center — APIs. Internet Archive. URL: https://openlibrary.org/developers/api
16. Laudon K. C., Traver C. G. E-Commerce 2021: Business, Technology, and Society. 16th ed. Pearson Education, 2021. 912 p.
17. Chaffey D., Ellis-Chadwick F. Digital Marketing: Strategy, Implementation and Practice. 8th ed. Pearson, 2022.
18. Sommerville I. Software Engineering. 10th ed. Pearson, 2023.

19. Pressman R. S., Maxim B. R. Software Engineering: A Practitioner’s Approach. 9th ed. New York : McGraw-Hill Education, 2020. 880 p.
20. Fowler M. Refactoring: Improving the Design of Existing Code. 2nd ed. Boston : Addison-Wesley Professional, 2021. 448 p.
21. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sebastopol : O’Reilly Media, 2022. 616 p.

ДОДАТОК А

```
// ===== docker-compose.yml — опис інфраструктури системи =====
services:
postgres:
image: postgres:16-alpine
restart: unless-stopped
environment:
POSTGRES_USER: ${POSTGRES_USER}
POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
POSTGRES_DB: ${POSTGRES_DB}
volumes:
- postgres_data:/var/lib/postgresql/data
healthcheck:
test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER} -d
${POSTGRES_DB}"]
interval: 5s
timeout: 5s
retries: 20

redis:
image: redis:7-alpine
restart: unless-stopped
command: ["redis-server", "--appendonly", "yes"]
volumes:
- redis_data:/data
healthcheck:
test: ["CMD", "redis-cli", "ping"]
interval: 5s
timeout: 5s
retries: 10

meilisearch:
image: getmeili/meilisearch:v1.11
```

```
restart: unless-stopped
environment:
MEILI_MASTER_KEY: ${MEILI_MASTER_KEY}
MEILI_ENV: production
MEILI_NO_ANALYTICS: "true"
volumes:
- meilisearch_data:/meili_data

minio:
image: minio/minio:RELEASE.2024-12-18T13-15-44Z
restart: unless-stopped
command: server /data --console-address ":9001"
environment:
MINIO_ROOT_USER: ${S3_ACCESS_KEY}
MINIO_ROOT_PASSWORD: ${S3_SECRET_KEY}
volumes:
- minio_data:/data
healthcheck:
test: ["CMD", "mc", "ready", "local"]
interval: 5s
timeout: 5s
retries: 20

minio-init:
image: minio/mc:RELEASE.2024-11-21T17-21-54Z
depends_on:
minio:
condition: service_healthy
environment:
S3_ACCESS_KEY: ${S3_ACCESS_KEY}
S3_SECRET_KEY: ${S3_SECRET_KEY}
S3_BUCKET: ${S3_BUCKET}
entrypoint: ["/bin/sh", "-c"]
command:
- |
set -e;
```

```
mc alias set local http://minio:9000 "$$S3_ACCESS_KEY"
"$$S3_SECRET_KEY";
mc mb --ignore-existing local/$$S3_BUCKET;
mc anonymous set download local/$$S3_BUCKET;
echo "minio bucket ready";

mailhog:
image: mailhog/mailhog:v1.0.1
restart: unless-stopped
ports:
- "8025:8025"

api:
build:
context: .
dockerfile: apps/api/Dockerfile
restart: unless-stopped
environment:
NODE_ENV: ${NODE_ENV}
PORT: "4000"
APP_URL: ${APP_URL}
DATABASE_URL: ${DATABASE_URL}
REDIS_URL: ${REDIS_URL}
JWT_ACCESS_SECRET: ${JWT_ACCESS_SECRET}
JWT_REFRESH_SECRET: ${JWT_REFRESH_SECRET}
JWT_ACCESS_TTL: ${JWT_ACCESS_TTL}
JWT_REFRESH_TTL: ${JWT_REFRESH_TTL}
STRIPE_SECRET_KEY: ${STRIPE_SECRET_KEY}
STRIPE_PUBLIC_KEY: ${STRIPE_PUBLIC_KEY}
STRIPE_WEBHOOK_SECRET: ${STRIPE_WEBHOOK_SECRET}
MEILI_HOST: ${MEILI_HOST}
MEILI_MASTER_KEY: ${MEILI_MASTER_KEY}
S3_ENDPOINT: ${S3_ENDPOINT}
S3_PUBLIC_URL: ${S3_PUBLIC_URL}
S3_ACCESS_KEY: ${S3_ACCESS_KEY}
S3_SECRET_KEY: ${S3_SECRET_KEY}
```

```
S3_BUCKET: ${S3_BUCKET}
S3_REGION: ${S3_REGION}
SMTP_HOST: ${SMTP_HOST}
SMTP_PORT: ${SMTP_PORT}
SMTP_FROM: ${SMTP_FROM}
ADMIN_EMAIL: ${ADMIN_EMAIL}
ADMIN_PASSWORD: ${ADMIN_PASSWORD}
ADMIN_FIRST_NAME: ${ADMIN_FIRST_NAME}
ADMIN_LAST_NAME: ${ADMIN_LAST_NAME}
SEED_BOOK_COUNT: ${SEED_BOOK_COUNT}
depends_on:
postgres:
condition: service_healthy
redis:
condition: service_healthy
meilisearch:
condition: service_started
minio-init:
condition: service_completed_successfully
healthcheck:
test: ["CMD", "wget", "--spider", "--quiet",
"http://localhost:4000/api/v1/health"]
interval: 10s
timeout: 5s
retries: 30
start_period: 30s

worker:
build:
context: .
dockerfile: apps/api/Dockerfile
restart: unless-stopped
command: ["node", "dist/worker.js"]
environment:
NODE_ENV: ${NODE_ENV}
DATABASE_URL: ${DATABASE_URL}
```

```
REDIS_URL: ${REDIS_URL}
MEILI_HOST: ${MEILI_HOST}
MEILI_MASTER_KEY: ${MEILI_MASTER_KEY}
SMTP_HOST: ${SMTP_HOST}
SMTP_PORT: ${SMTP_PORT}
SMTP_FROM: ${SMTP_FROM}
APP_URL: ${APP_URL}
depends_on:
api:
condition: service_healthy

seed:
build:
context: .
dockerfile: apps/api/Dockerfile
restart: "no"
command: ["node", "dist/seed.js"]
environment:
NODE_ENV: ${NODE_ENV}
DATABASE_URL: ${DATABASE_URL}
MEILI_HOST: ${MEILI_HOST}
MEILI_MASTER_KEY: ${MEILI_MASTER_KEY}
S3_ENDPOINT: ${S3_ENDPOINT}
S3_PUBLIC_URL: ${S3_PUBLIC_URL}
S3_ACCESS_KEY: ${S3_ACCESS_KEY}
S3_SECRET_KEY: ${S3_SECRET_KEY}
S3_BUCKET: ${S3_BUCKET}
S3_REGION: ${S3_REGION}
ADMIN_EMAIL: ${ADMIN_EMAIL}
ADMIN_PASSWORD: ${ADMIN_PASSWORD}
ADMIN_FIRST_NAME: ${ADMIN_FIRST_NAME}
ADMIN_LAST_NAME: ${ADMIN_LAST_NAME}
SEED_BOOK_COUNT: ${SEED_BOOK_COUNT}
depends_on:
api:
condition: service_healthy
```

```
web:
build:
context: .
dockerfile: apps/web/Dockerfile
args:
NEXT_PUBLIC_APP_URL: ${NEXT_PUBLIC_APP_URL}
NEXT_PUBLIC_API_URL: ${NEXT_PUBLIC_API_URL}
NEXT_PUBLIC_CDN_URL: ${NEXT_PUBLIC_CDN_URL}
NEXT_PUBLIC_STRIPE_PUBLIC_KEY: ${NEXT_PUBLIC_STRIPE_PUBLIC_KEY}
restart: unless-stopped
environment:
NODE_ENV: ${NODE_ENV}
PORT: "3000"
HOSTNAME: "0.0.0.0"
NEXT_PUBLIC_APP_URL: ${NEXT_PUBLIC_APP_URL}
NEXT_PUBLIC_API_URL: ${NEXT_PUBLIC_API_URL}
NEXT_PUBLIC_CDN_URL: ${NEXT_PUBLIC_CDN_URL}
NEXT_PUBLIC_STRIPE_PUBLIC_KEY: ${NEXT_PUBLIC_STRIPE_PUBLIC_KEY}
INTERNAL_API_URL: http://api:4000/api/v1
depends_on:
api:
condition: service_healthy

caddy:
image: caddy:2-alpine
restart: unless-stopped
ports:
- "80:80"
- "443:443"
volumes:
- ./infra/caddy/Caddyfile:/etc/caddy/Caddyfile:ro
- caddy_data:/data
- caddy_config:/config
depends_on:
- web
```

```
- api
- minio

volumes:
postgres_data:
redis_data:
meilisearch_data:
minio_data:
caddy_data:
caddy_config:


// ===== apps/api/prisma/schema.prisma — схема бази даних =====
generator client {
provider = "prisma-client-js"
}

datasource db {
provider = "postgresql"
url      = env("DATABASE_URL")
}

enum Role {
CUSTOMER
ADMIN
}

enum OrderStatus {
PENDING
AWAITING_PAYMENT
PAID
PROCESSING
SHIPPED
DELIVERED
CANCELLED
REFUNDED
```

```
}

enum ReviewStatus {
PENDING
APPROVED
REJECTED
}

enum PromoType {
PERCENT
FIXED
}

enum BookLanguage {
UK
EN
MIXED
}

enum BookFormat {
HARDCOVER
PAPERBACK
}

model User {
id              String         @id @default(cuid())
email           String         @unique
passwordHash    String
role            Role           @default(CUSTOMER)
firstName       String
lastName        String
phone           String?
emailVerifiedAt DateTime?
createdAt       DateTime       @default(now())
updatedAt       DateTime       @updatedAt
addresses       Address[]
```

```
orders          Order[]
reviews         Review[]
wishlist        WishlistItem[]
cartItems       CartItem[]
refreshTokens   RefreshToken[]
resetTokens     PasswordResetToken[]
}

model RefreshToken {
id        String   @id @default(cuid())
userId    String
user      User     @relation(fields: [userId], references: [id],
onDelete: Cascade)
tokenHash String   @unique
expiresAt DateTime
revokedAt DateTime?
createdAt DateTime @default(now())

@@index([userId])
}

model PasswordResetToken {
id        String   @id @default(cuid())
userId    String
user      User     @relation(fields: [userId], references: [id],
onDelete: Cascade)
tokenHash String   @unique
expiresAt DateTime
usedAt    DateTime?
createdAt DateTime @default(now())

@@index([userId])
}

model Address {
id         String   @id @default(cuid())
```

```
userId     String
user       User     @relation(fields: [userId], references: [id],
onDelete: Cascade)
fullName   String
country    String
city       String
street     String
postalCode String
phone      String?
isDefault  Boolean  @default(false)
createdAt  DateTime @default(now())
updatedAt  DateTime @updatedAt

@@index([userId])
}

model Category {
id         String        @id @default(cuid())
slug       String        @unique
nameUk     String
nameEn     String
icon       String?
position   Int           @default(0)
parentId   String?
parent     Category?     @relation("CategoryTree", fields: [parentId],
references: [id])
children   Category[]    @relation("CategoryTree")
books      BookCategory[]
}

model Author {
id       String       @id @default(cuid())
slug     String       @unique
nameUk   String
nameEn   String
bioUk    String?
```

```
bioEn    String?
photoKey String?
books    BookAuthor[]
}

model Publisher {
id      String  @id @default(cuid())
slug    String  @unique
name    String
country String?
books   Book[]
}

model Book {
id            String         @id @default(cuid())
slug          String         @unique
isbn13        String?        @unique
titleUk       String
titleEn       String
descriptionUk String
descriptionEn String
priceUah      Int
oldPriceUah   Int?
stock         Int            @default(0)
language      BookLanguage   @default(UK)
pages         Int?
year          Int?
format        BookFormat     @default(PAPERBACK)
publisherId   String?
publisher     Publisher?     @relation(fields: [publisherId],
references: [id])
coverKey      String?
rating        Float          @default(0)
reviewCount   Int            @default(0)
isFeatured    Boolean        @default(false)
isBestseller  Boolean        @default(false)
```

```
isActive      Boolean        @default(true)
createdAt     DateTime       @default(now())
updatedAt     DateTime       @updatedAt
authors       BookAuthor[]
categories    BookCategory[]
reviews       Review[]
cartItems     CartItem[]
orderItems    OrderItem[]
wishlist      WishlistItem[]
stockMoves    StockMovement[]

@@index([isFeatured])
@@index([isBestseller])
@@index([isActive])
}

model BookAuthor {
bookId   String
authorId String
book     Book   @relation(fields: [bookId], references: [id],
onDelete: Cascade)
author   Author @relation(fields: [authorId], references: [id],
onDelete: Cascade)

@@id([bookId, authorId])
}

model BookCategory {
bookId     String
categoryId String
book       Book     @relation(fields: [bookId], references: [id],
onDelete: Cascade)
category   Category @relation(fields: [categoryId], references: [id],
onDelete: Cascade)

@@id([bookId, categoryId])
```

```
}

model CartItem {
id          String   @id @default(cuid())
userId      String?
user        User?    @relation(fields: [userId], references: [id],
onDelete: Cascade)
guestToken String?
bookId      String
book        Book     @relation(fields: [bookId], references: [id],
onDelete: Cascade)
quantity    Int      @default(1)
createdAt   DateTime @default(now())
updatedAt   DateTime @updatedAt

@@unique([userId, bookId])
@@unique([guestToken, bookId])
@@index([userId])
@@index([guestToken])
}

model WishlistItem {
id         String   @id @default(cuid())
userId     String
user       User     @relation(fields: [userId], references: [id],
onDelete: Cascade)
bookId     String
book       Book     @relation(fields: [bookId], references: [id],
onDelete: Cascade)
createdAt DateTime @default(now())

@@unique([userId, bookId])
}

model Order {
id                    String      @id @default(cuid())
```

```
number                String      @unique
userId                String
user                  User        @relation(fields: [userId],
references: [id])
status                OrderStatus @default(PENDING)
subtotalUah           Int
discountUah           Int         @default(0)
shippingUah           Int         @default(0)
totalUah              Int
promoCodeId           String?
promoCode             PromoCode?  @relation(fields: [promoCodeId],
references: [id])
shippingAddressJson   Json
contactEmail          String
contactPhone          String?
stripePaymentIntentId String?     @unique
stripeChargeId        String?
paidAt                DateTime?
shippedAt             DateTime?
deliveredAt           DateTime?
cancelledAt           DateTime?
createdAt             DateTime    @default(now())
updatedAt             DateTime    @updatedAt
items                 OrderItem[]
stockMoves            StockMovement[]

@@index([userId])
@@index([status])
}

model OrderItem {
id            String @id @default(cuid())
orderId       String
order         Order  @relation(fields: [orderId], references: [id],
onDelete: Cascade)
bookId        String
```

```
book          Book   @relation(fields: [bookId], references: [id])
quantity      Int
priceUah      Int
titleSnapshot String
}

model Review {
id        String       @id @default(cuid())
userId    String
user      User         @relation(fields: [userId], references: [id],
onDelete: Cascade)
bookId    String
book      Book         @relation(fields: [bookId], references: [id],
onDelete: Cascade)
rating    Int
body      String
status    ReviewStatus @default(PENDING)
createdAt DateTime     @default(now())
updatedAt DateTime     @updatedAt

@@unique([userId, bookId])
@@index([bookId])
@@index([status])
}

model PromoCode {
id           String    @id @default(cuid())
code         String    @unique
type         PromoType
value        Int
minSubtotal  Int       @default(0)
expiresAt    DateTime?
usageLimit   Int?
usedCount    Int       @default(0)
isActive     Boolean   @default(true)
createdAt    DateTime  @default(now())
```

```
orders       Order[]
}

model StockMovement {
id        String   @id @default(cuid())
bookId    String
book      Book     @relation(fields: [bookId], references: [id],
onDelete: Cascade)
delta     Int
reason    String
orderId   String?
order     Order?   @relation(fields: [orderId], references: [id])
createdAt DateTime @default(now())

@@index([bookId])
}

model WebhookEvent {
id           String   @id @default(cuid())
provider     String
externalId   String
type         String
processedAt  DateTime @default(now())
payload      Json

@@unique([provider, externalId])
}

model AuditLog {
id         String   @id @default(cuid())
actorId    String?
action     String
entityType String
entityId   String?
payload    Json?
createdAt  DateTime @default(now())
```

```
@@index([actorId])
@@index([entityType, entityId])
}


// ===== apps/api/src/main.ts — точка входу серверного застосунку
=====
import { NestFactory } from '@nestjs/core';
import { ValidationPipe } from '@nestjs/common';
import { DocumentBuilder, SwaggerModule } from '@nestjs/swagger';
import { json, raw } from 'express';
import cookieParser from 'cookie-parser';
import helmet from 'helmet';
import { AppModule } from './app.module';
import { HttpExceptionFilter } from './common/filters/http-
exception.filter';

async function bootstrap() {
const app = await NestFactory.create(AppModule, { bufferLogs: false
});

app.use('/api/v1/webhooks/stripe', raw({ type: '*/*' }));
app.use(json({ limit: '2mb' }));
app.use(cookieParser());
app.use(
helmet({
contentSecurityPolicy: false,
crossOriginResourcePolicy: { policy: 'cross-origin' },
}),
);

app.enableCors({
origin: [process.env.APP_URL ?? 'http://localhost',
'http://localhost:3000'],
credentials: true,
```

```
});

app.setGlobalPrefix('api/v1');

app.useGlobalPipes(
new ValidationPipe({
whitelist: true,
transform: true,
forbidNonWhitelisted: false,
transformOptions: { enableImplicitConversion: true },
}),
);

app.useGlobalFilters(new HttpExceptionFilter());

const config = new DocumentBuilder()
.setTitle('BookNest API')
.setDescription('REST API for BookNest e-commerce')
.setVersion('1.0')
.addBearerAuth()
.build();
const doc = SwaggerModule.createDocument(app, config);
SwaggerModule.setup('api/docs', app, doc);

const port = Number(process.env.PORT ?? 4000);
await app.listen(port, '0.0.0.0');
}

bootstrap();


// ===== apps/api/src/app.module.ts — кореневий модуль =====
import { Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';
import { ThrottlerModule, ThrottlerGuard } from '@nestjs/throttler';
import { APP_GUARD } from '@nestjs/core';
```

```
import { BullModule } from '@nestjs/bullmq';
import { PrismaModule } from './prisma/prisma.module';
import { AuthModule } from './auth/auth.module';
import { UsersModule } from './users/users.module';
import { BooksModule } from './books/books.module';
import { CategoriesModule } from './categories/categories.module';
import { AuthorsModule } from './authors/authors.module';
import { CartModule } from './cart/cart.module';
import { WishlistModule } from './wishlist/wishlist.module';
import { OrdersModule } from './orders/orders.module';
import { CheckoutModule } from './checkout/checkout.module';
import { PaymentsModule } from './payments/payments.module';
import { ReviewsModule } from './reviews/reviews.module';
import { PromoModule } from './promo/promo.module';
import { SearchModule } from './search/search.module';
import { StorageModule } from './storage/storage.module';
import { MailModule } from './mail/mail.module';
import { AdminModule } from './admin/admin.module';
import { HealthModule } from './health/health.module';

@Module({
imports: [
ConfigModule.forRoot({ isGlobal: true }),
ThrottlerModule.forRoot([{ ttl: 60_000, limit: 120 }]),
BullModule.forRootAsync({
useFactory: () => ({
connection: {
url: process.env.REDIS_URL || 'redis://redis:6379',
},
}),
}),
PrismaModule,
StorageModule,
SearchModule,
MailModule,
AuthModule,
```

```
UsersModule,
BooksModule,
CategoriesModule,
AuthorsModule,
CartModule,
WishlistModule,
PromoModule,
OrdersModule,
CheckoutModule,
PaymentsModule,
ReviewsModule,
AdminModule,
HealthModule,
],
providers: [{ provide: APP_GUARD, useClass: ThrottlerGuard }],
})
export class AppModule {}


// ===== apps/api/src/auth/auth.service.ts — сервіс автентифікації
=====
import { BadRequestException, ConflictException, Injectable,
UnauthorizedException } from '@nestjs/common';
import * as argon2 from 'argon2';
import { PrismaService } from '../prisma/prisma.service';
import { TokensService } from './tokens.service';
import { MailService } from '../mail/mail.service';
import { ChangePasswordDto, ForgotPasswordDto, LoginDto, RegisterDto,
ResetPasswordDto } from './dto/auth.dto';

@Injectable()
export class AuthService {
constructor(
private prisma: PrismaService,
private tokens: TokensService,
private mail: MailService,
```

```
) {}

async register(dto: RegisterDto) {
const existing = await this.prisma.user.findUnique({ where: { email:
dto.email.toLowerCase() } });
if (existing) throw new ConflictException({ message: 'Email already
registered', code: 'EMAIL_EXISTS' });
const passwordHash = await argon2.hash(dto.password, { type:
argon2.argon2id });
const user = await this.prisma.user.create({
data: {
email: dto.email.toLowerCase(),
passwordHash,
firstName: dto.firstName,
lastName: dto.lastName,
},
});
if (dto.guestToken) await this.mergeGuestCart(user.id,
dto.guestToken);
const tokens = await this.issueTokens(user.id, user.role, user.email);
await this.mail.sendWelcome(user.email, user.firstName);
return { user: this.publicUser(user), ...tokens };
}

async login(dto: LoginDto) {
const user = await this.prisma.user.findUnique({ where: { email:
dto.email.toLowerCase() } });
if (!user) throw new UnauthorizedException({ message: 'Invalid
credentials', code: 'INVALID_CREDENTIALS' });
const ok = await argon2.verify(user.passwordHash, dto.password);
if (!ok) throw new UnauthorizedException({ message: 'Invalid
credentials', code: 'INVALID_CREDENTIALS' });
if (dto.guestToken) await this.mergeGuestCart(user.id,
dto.guestToken);
const tokens = await this.issueTokens(user.id, user.role, user.email);
return { user: this.publicUser(user), ...tokens };
```

```
}

async refresh(rawRefresh: string) {
const result = await this.tokens.rotate(rawRefresh);
if (!result) throw new UnauthorizedException('Invalid refresh token');
const user = await this.prisma.user.findUnique({ where: { id:
result.userId } });
if (!user) throw new UnauthorizedException();
const accessToken = this.tokens.signAccess(user.id, user.role,
user.email);
return {
user: this.publicUser(user),
accessToken,
refreshToken: result.token,
refreshExpiresAt: result.expiresAt,
};
}

async logout(rawRefresh?: string) {
if (rawRefresh) await this.tokens.revoke(rawRefresh);
return { ok: true };
}

async forgotPassword(dto: ForgotPasswordDto) {
const user = await this.prisma.user.findUnique({ where: { email:
dto.email.toLowerCase() } });
if (user) {
const token = await this.tokens.issueResetToken(user.id);
await this.mail.sendPasswordReset(user.email, user.firstName, token);
}
return { ok: true };
}

async resetPassword(dto: ResetPasswordDto) {
const userId = await this.tokens.consumeResetToken(dto.token);
```

```
if (!userId) throw new BadRequestException({ message: 'Invalid or
expired token', code: 'RESET_TOKEN_INVALID' });
const passwordHash = await argon2.hash(dto.password, { type:
argon2.argon2id });
await this.prisma.user.update({ where: { id: userId }, data: {
passwordHash } });
await this.prisma.refreshToken.updateMany({ where: { userId,
revokedAt: null }, data: { revokedAt: new Date() } });
return { ok: true };
}

async changePassword(userId: string, dto: ChangePasswordDto) {
const user = await this.prisma.user.findUnique({ where: { id: userId }
});
if (!user) throw new UnauthorizedException();
const ok = await argon2.verify(user.passwordHash,
dto.currentPassword);
if (!ok) throw new BadRequestException({ message: 'Current password is
incorrect', code: 'PASSWORD_INCORRECT' });
const passwordHash = await argon2.hash(dto.newPassword, { type:
argon2.argon2id });
await this.prisma.user.update({ where: { id: userId }, data: {
passwordHash } });
return { ok: true };
}

private async issueTokens(userId: string, role: string, email: string)
{
const accessToken = this.tokens.signAccess(userId, role, email);
const { token, expiresAt } = await this.tokens.issueRefresh(userId);
return { accessToken, refreshToken: token, refreshExpiresAt: expiresAt
};
}

private publicUser(u: any) {
return {
```

```
id: u.id,
email: u.email,
role: u.role,
firstName: u.firstName,
lastName: u.lastName,
phone: u.phone,
createdAt: u.createdAt,
};
}

private async mergeGuestCart(userId: string, guestToken: string) {
const guestItems = await this.prisma.cartItem.findMany({ where: {
guestToken } });
for (const item of guestItems) {
await this.prisma.cartItem.upsert({
where: { userId_bookId: { userId, bookId: item.bookId } },
update: { quantity: { increment: item.quantity } },
create: { userId, bookId: item.bookId, quantity: item.quantity },
});
}
await this.prisma.cartItem.deleteMany({ where: { guestToken } });
}
}


// ===== apps/api/src/auth/tokens.service.ts — сервіс токенів =====
import { Injectable } from '@nestjs/common';
import { JwtService } from '@nestjs/jwt';
import { createHash, randomBytes } from 'crypto';
import { PrismaService } from '../prisma/prisma.service';

@Injectable()
export class TokensService {
constructor(private prisma: PrismaService, private jwt: JwtService) {}

signAccess(userId: string, role: string, email: string): string {
```

```
return this.jwt.sign(
{ sub: userId, role, email },
{
secret: process.env.JWT_ACCESS_SECRET,
expiresIn: Number(process.env.JWT_ACCESS_TTL ?? 900),
},
);
}

async issueRefresh(userId: string): Promise<{ token: string;
expiresAt: Date }> {
const token = randomBytes(48).toString('hex');
const tokenHash = createHash('sha256').update(token).digest('hex');
const ttl = Number(process.env.JWT_REFRESH_TTL ?? 2_592_000);
const expiresAt = new Date(Date.now() + ttl * 1000);
await this.prisma.refreshToken.create({
data: { userId, tokenHash, expiresAt },
});
return { token, expiresAt };
}

async rotate(rawToken: string): Promise<{ userId: string; token:
string; expiresAt: Date } | null> {
const tokenHash = createHash('sha256').update(rawToken).digest('hex');
const record = await this.prisma.refreshToken.findUnique({ where: {
tokenHash } });
if (!record || record.revokedAt || record.expiresAt < new Date())
return null;
await this.prisma.refreshToken.update({
where: { tokenHash },
data: { revokedAt: new Date() },
});
const next = await this.issueRefresh(record.userId);
return { userId: record.userId, ...next };
}
```

```
async revoke(rawToken: string): Promise<void> {
const tokenHash = createHash('sha256').update(rawToken).digest('hex');
await this.prisma.refreshToken.updateMany({
where: { tokenHash, revokedAt: null },
data: { revokedAt: new Date() },
});
}

async issueResetToken(userId: string): Promise<string> {
const token = randomBytes(32).toString('hex');
const tokenHash = createHash('sha256').update(token).digest('hex');
const expiresAt = new Date(Date.now() + 3600 * 1000);
await this.prisma.passwordResetToken.create({
data: { userId, tokenHash, expiresAt },
});
return token;
}

async consumeResetToken(rawToken: string): Promise<string | null> {
const tokenHash = createHash('sha256').update(rawToken).digest('hex');
const rec = await this.prisma.passwordResetToken.findUnique({ where: {
tokenHash } });
if (!rec || rec.usedAt || rec.expiresAt < new Date()) return null;
await this.prisma.passwordResetToken.update({
where: { tokenHash },
data: { usedAt: new Date() },
});
return rec.userId;
}
}


// ===== apps/api/src/auth/jwt.strategy.ts — стратегія перевірки JWT
=====
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { PassportStrategy } from '@nestjs/passport';
```

```
import { ExtractJwt, Strategy } from 'passport-jwt';
import { Request } from 'express';

const fromCookie = (req: Request): string | null => {
if (req && req.cookies && req.cookies['access_token']) return
req.cookies['access_token'];
return null;
};

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy, 'jwt') {
constructor() {
super({
jwtFromRequest: ExtractJwt.fromExtractors([fromCookie,
ExtractJwt.fromAuthHeaderAsBearerToken()]),
ignoreExpiration: false,
secretOrKey: process.env.JWT_ACCESS_SECRET ||
'dev_secret_change_me_change_me_change_me',
});
}

async validate(payload: any) {
if (!payload?.sub) throw new UnauthorizedException();
return { id: payload.sub, role: payload.role, email: payload.email };
}
}


// ===== apps/api/src/common/guards/roles.guard.ts — захисник ролей
=====
import { CanActivate, ExecutionContext, ForbiddenException, Injectable
} from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { ROLES_KEY } from '../decorators/roles.decorator';

@Injectable()
```

```
export class RolesGuard implements CanActivate {
constructor(private reflector: Reflector) {}

canActivate(ctx: ExecutionContext): boolean {
const required = this.reflector.getAllAndOverride<string[]>(ROLES_KEY,
[
ctx.getHandler(),
ctx.getClass(),
]);
if (!required || required.length === 0) return true;
const req = ctx.switchToHttp().getRequest();
const user = req.user;
if (!user || !required.includes(user.role)) {
throw new ForbiddenException('Insufficient permissions');
}
return true;
}
}


// ===== apps/api/src/books/books.service.ts — сервіс каталогу книг
=====
import { Injectable, NotFoundException } from '@nestjs/common';
import { Prisma } from '@prisma/client';
import { PrismaService } from '../prisma/prisma.service';
import { SearchService } from '../search/search.service';

export interface BookFilters {
category?: string;
author?: string;
search?: string;
priceMin?: number;
priceMax?: number;
language?: 'UK' | 'EN' | 'MIXED';
inStock?: boolean;
isFeatured?: boolean;
```

```
isBestseller?: boolean;
sort?: 'newest' | 'price_asc' | 'price_desc' | 'rating' | 'title';
page?: number;
limit?: number;
}

@Injectable()
export class BooksService {
constructor(private prisma: PrismaService, private search:
SearchService) {}

async list(filters: BookFilters) {
const page = Math.max(1, Number(filters.page) || 1);
const limit = Math.min(60, Math.max(1, Number(filters.limit) || 24));
const skip = (page - 1) * limit;

const where: Prisma.BookWhereInput = { isActive: true };
if (filters.category) {
where.categories = { some: { category: { slug: filters.category } } };
}
if (filters.author) {
where.authors = { some: { author: { slug: filters.author } } };
}
if (filters.priceMin !== undefined) where.priceUah = {
...(where.priceUah as object), gte: filters.priceMin };
if (filters.priceMax !== undefined) where.priceUah = {
...(where.priceUah as object), lte: filters.priceMax };
if (filters.language) where.language = filters.language;
if (filters.inStock) where.stock = { gt: 0 };
if (filters.isFeatured) where.isFeatured = true;
if (filters.isBestseller) where.isBestseller = true;
if (filters.search) {
where.OR = [
{ titleUk: { contains: filters.search, mode: 'insensitive' } },
{ titleEn: { contains: filters.search, mode: 'insensitive' } },
];
```

```
}

let orderBy: Prisma.BookOrderByWithRelationInput = { createdAt: 'desc'
};
switch (filters.sort) {
case 'price_asc':
orderBy = { priceUah: 'asc' };
break;
case 'price_desc':
orderBy = { priceUah: 'desc' };
break;
case 'rating':
orderBy = { rating: 'desc' };
break;
case 'title':
orderBy = { titleUk: 'asc' };
break;
}

const [items, total] = await this.prisma.$transaction([
this.prisma.book.findMany({
where,
orderBy,
skip,
take: limit,
include: {
authors: { include: { author: true } },
categories: { include: { category: true } },
},
}),
this.prisma.book.count({ where }),
]);

return {
items: items.map(this.format),
page,
```

```
limit,
total,
totalPages: Math.ceil(total / limit),
};
}

async getBySlug(slug: string) {
const book = await this.prisma.book.findUnique({
where: { slug },
include: {
authors: { include: { author: true } },
categories: { include: { category: true } },
publisher: true,
},
});
if (!book || !book.isActive) throw new NotFoundException('Book not
found');
return this.format(book);
}

async getRelated(slug: string, limit = 8) {
const book = await this.prisma.book.findUnique({
where: { slug },
include: { categories: true },
});
if (!book) return [];
const categoryIds = book.categories.map((c) => c.categoryId);
const related = await this.prisma.book.findMany({
where: {
id: { not: book.id },
isActive: true,
categories: { some: { categoryId: { in: categoryIds } } },
},
take: limit,
orderBy: { rating: 'desc' },
include: {
```

```
authors: { include: { author: true } },
categories: { include: { category: true } },
},
});
return related.map(this.format);
}

featured(limit = 8) {
return this.prisma.book
.findMany({
where: { isActive: true, isFeatured: true },
take: limit,
orderBy: { createdAt: 'desc' },
include: { authors: { include: { author: true } }, categories: {
include: { category: true } } },
})
.then((rows) => rows.map(this.format));
}

bestsellers(limit = 8) {
return this.prisma.book
.findMany({
where: { isActive: true, isBestseller: true },
take: limit,
orderBy: { rating: 'desc' },
include: { authors: { include: { author: true } }, categories: {
include: { category: true } } },
})
.then((rows) => rows.map(this.format));
}

newest(limit = 8) {
return this.prisma.book
.findMany({
where: { isActive: true },
take: limit,
```

```
orderBy: { createdAt: 'desc' },
include: { authors: { include: { author: true } }, categories: {
include: { category: true } } },
})
.then((rows) => rows.map(this.format));
}

async searchAutocomplete(q: string, limit = 8) {
if (!q || q.length < 2) return [];
return this.search.searchBooks(q, limit);
}

format = (book: any) => ({
id: book.id,
slug: book.slug,
isbn13: book.isbn13,
titleUk: book.titleUk,
titleEn: book.titleEn,
descriptionUk: book.descriptionUk,
descriptionEn: book.descriptionEn,
priceUah: book.priceUah,
oldPriceUah: book.oldPriceUah,
stock: book.stock,
language: book.language,
pages: book.pages,
year: book.year,
format: book.format,
coverKey: book.coverKey,
coverUrl: this.coverUrl(book.coverKey),
rating: book.rating,
reviewCount: book.reviewCount,
isFeatured: book.isFeatured,
isBestseller: book.isBestseller,
authors: (book.authors || []).map((a: any) => ({
id: a.author.id,
slug: a.author.slug,
```

```
nameUk: a.author.nameUk,
nameEn: a.author.nameEn,
})),
categories: (book.categories || []).map((c: any) => ({
id: c.category.id,
slug: c.category.slug,
nameUk: c.category.nameUk,
nameEn: c.category.nameEn,
})),
publisher: book.publisher
? { id: book.publisher.id, slug: book.publisher.slug, name:
book.publisher.name }
: null,
createdAt: book.createdAt,
});

private coverUrl(key?: string | null): string | null {
if (!key) return null;
const base = process.env.S3_PUBLIC_URL || 'http://localhost/cdn';
return `${base}/${key}`;
}
}


// ===== apps/api/src/search/search.service.ts — сервіс пошуку =====
import { Injectable, Logger, OnModuleInit } from '@nestjs/common';
import { MeiliSearch } from 'meilisearch';

@Injectable()
export class SearchService implements OnModuleInit {
private readonly logger = new Logger(SearchService.name);
private client: MeiliSearch;
private readonly index = 'books';

constructor() {
this.client = new MeiliSearch({
```

```
host: process.env.MEILI_HOST || 'http://meilisearch:7700',
apiKey: process.env.MEILI_MASTER_KEY,
});
}

async onModuleInit() {
try {
await this.client.health();
await this.ensureIndex();
} catch (e: any) {
this.logger.warn(`Meilisearch not reachable on boot: ${e?.message}`);
}
}

async ensureIndex() {
try {
await this.client.createIndex(this.index, { primaryKey: 'id' });
} catch {}
await this.client.index(this.index).updateSettings({
searchableAttributes: ['titleUk', 'titleEn', 'authors',
'descriptionUk', 'descriptionEn'],
filterableAttributes: ['categories', 'language', 'isActive'],
sortableAttributes: ['priceUah', 'rating', 'createdAt'],
typoTolerance: { enabled: true, minWordSizeForTypos: { oneTypo: 4,
twoTypos: 8 } },
});
}

async indexBook(book: {
id: string;
slug: string;
titleUk: string;
titleEn: string;
descriptionUk: string;
descriptionEn: string;
priceUah: number;
```

```
rating: number;
coverKey?: string | null;
language: string;
isActive: boolean;
authors: string[];
categories: string[];
createdAt: Date;
}) {
await this.client.index(this.index).addDocuments([
{
...book,
createdAt: book.createdAt.getTime ? book.createdAt.getTime() :
Number(book.createdAt),
},
]);
}

async indexMany(books: any[]) {
if (books.length === 0) return;
await this.client.index(this.index).addDocuments(
books.map((b) => ({
...b,
createdAt: b.createdAt?.getTime ? b.createdAt.getTime() :
Number(b.createdAt) || Date.now(),
})),
);
}

async removeBook(id: string) {
try {
await this.client.index(this.index).deleteDocument(id);
} catch {}
}

async searchBooks(query: string, limit = 10) {
try {
```

```
const r = await this.client.index(this.index).search(query, {
limit,
filter: ['isActive = true'],
attributesToHighlight: ['titleUk', 'titleEn'],
});
return r.hits;
} catch (e: any) {
this.logger.warn(`Search failed: ${e?.message}`);
return [];
}
}
}


// ===== apps/api/src/cart/cart.service.ts — сервіс кошика =====
import { BadRequestException, Injectable, NotFoundException } from
'@nestjs/common';
import { PrismaService } from '../prisma/prisma.service';

interface Owner {
userId?: string;
guestToken?: string;
}

@Injectable()
export class CartService {
constructor(private prisma: PrismaService) {}

private where(owner: Owner) {
if (owner.userId) return { userId: owner.userId };
if (owner.guestToken) return { guestToken: owner.guestToken };
throw new BadRequestException('Cart owner not specified');
}

async get(owner: Owner) {
const items = await this.prisma.cartItem.findMany({
```

```
where: this.where(owner),
include: { book: { include: { authors: { include: { author: true } } }
} },
orderBy: { createdAt: 'asc' },
});
const cdn = process.env.S3_PUBLIC_URL || 'http://localhost/cdn';
const mapped = items.map((it) => ({
id: it.id,
quantity: it.quantity,
book: {
id: it.book.id,
slug: it.book.slug,
titleUk: it.book.titleUk,
titleEn: it.book.titleEn,
priceUah: it.book.priceUah,
oldPriceUah: it.book.oldPriceUah,
stock: it.book.stock,
coverKey: it.book.coverKey,
coverUrl: it.book.coverKey ? `${cdn}/${it.book.coverKey}` : null,
authors: it.book.authors.map((a) => ({
id: a.author.id,
slug: a.author.slug,
nameUk: a.author.nameUk,
nameEn: a.author.nameEn,
})),
},
}));
const subtotal = mapped.reduce((s, x) => s + x.book.priceUah *
x.quantity, 0);
return { items: mapped, subtotal, count: mapped.reduce((s, x) => s +
x.quantity, 0) };
}

async addItem(owner: Owner, bookId: string, quantity: number) {
const book = await this.prisma.book.findUnique({ where: { id: bookId }
});
```

```
if (!book || !book.isActive) throw new NotFoundException('Book not
found');
if (quantity < 1) throw new BadRequestException('Quantity must be >=
1');
if (book.stock < quantity) throw new BadRequestException('Not enough
stock');

if (owner.userId) {
await this.prisma.cartItem.upsert({
where: { userId_bookId: { userId: owner.userId, bookId } },
update: { quantity: { increment: quantity } },
create: { userId: owner.userId, bookId, quantity },
});
} else if (owner.guestToken) {
await this.prisma.cartItem.upsert({
where: { guestToken_bookId: { guestToken: owner.guestToken, bookId }
},
update: { quantity: { increment: quantity } },
create: { guestToken: owner.guestToken, bookId, quantity },
});
}
return this.get(owner);
}

async updateItem(owner: Owner, itemId: string, quantity: number) {
if (quantity < 1) throw new BadRequestException('Quantity must be >=
1');
const item = await this.prisma.cartItem.findUnique({ where: { id:
itemId }, include: { book: true } });
if (!item) throw new NotFoundException();
if (owner.userId ? item.userId !== owner.userId : item.guestToken !==
owner.guestToken) {
throw new NotFoundException();
}
if (item.book.stock < quantity) throw new BadRequestException('Not
enough stock');
```

```
await this.prisma.cartItem.update({ where: { id: itemId }, data: {
quantity } });
return this.get(owner);
}

async removeItem(owner: Owner, itemId: string) {
const item = await this.prisma.cartItem.findUnique({ where: { id:
itemId } });
if (!item) throw new NotFoundException();
if (owner.userId ? item.userId !== owner.userId : item.guestToken !==
owner.guestToken) {
throw new NotFoundException();
}
await this.prisma.cartItem.delete({ where: { id: itemId } });
return this.get(owner);
}

async clear(owner: Owner) {
await this.prisma.cartItem.deleteMany({ where: this.where(owner) });
return this.get(owner);
}
}


// ===== apps/api/src/checkout/checkout.service.ts — сервіс оформлення
замовлення =====
import { BadRequestException, Injectable } from '@nestjs/common';
import { PrismaService } from '../prisma/prisma.service';
import { StripeService } from '../payments/stripe.service';
import { PromoService } from '../promo/promo.service';

interface ShippingAddress {
fullName: string;
country: string;
city: string;
street: string;
```

```
postalCode: string;
phone?: string;
method?: string;
methodLabel?: string;
paymentMethod?: string;
}

interface CreateIntentInput {
userId: string;
email: string;
shipping: ShippingAddress;
contactPhone?: string;
promoCode?: string;
}

const FREE_SHIPPING_THRESHOLD = 150_000;
const SHIPPING_PRICES: Record<string, number> = {
np_branch: 5500,
np_courier: 12000,
ukr_poshta: 3500,
pickup: 0,
};

function genNumber() {
const d = new Date();
const ymd = `${d.getFullYear()}${String(d.getMonth() + 1).padStart(2,
'0')}${String(d.getDate()).padStart(2, '0')}`;
const rand = Math.floor(Math.random() * 9000 + 1000);
return `BN-${ymd}-${rand}`;
}

@Injectable()
export class CheckoutService {
constructor(
private prisma: PrismaService,
private stripe: StripeService,
```

```
private promo: PromoService,
) {}

async createIntent(input: CreateIntentInput) {
const cartItems = await this.prisma.cartItem.findMany({
where: { userId: input.userId },
include: { book: true },
});
if (cartItems.length === 0) throw new BadRequestException('Cart is
empty');

for (const it of cartItems) {
if (!it.book.isActive) throw new BadRequestException(`Book
${it.book.titleUk} is no longer available`);
if (it.book.stock < it.quantity) {
throw new BadRequestException(`Not enough stock for
"${it.book.titleUk}"`);
}
}

const subtotal = cartItems.reduce((s, it) => s + it.book.priceUah *
it.quantity, 0);

let discount = 0;
let promoId: string | undefined;
if (input.promoCode) {
const v = await this.promo.validate(input.promoCode, subtotal);
discount = v.discount;
promoId = v.promo.id;
}

const methodKey = input.shipping.method &&
SHIPPING_PRICES[input.shipping.method] !== undefined
? input.shipping.method
: 'np_branch';
const rawShipping = SHIPPING_PRICES[methodKey];
```

```
const shipping = subtotal >= FREE_SHIPPING_THRESHOLD ? 0 :
rawShipping;
const total = subtotal - discount + shipping;

const order = await this.prisma.order.create({
data: {
number: genNumber(),
userId: input.userId,
status: 'AWAITING_PAYMENT',
subtotalUah: subtotal,
discountUah: discount,
shippingUah: shipping,
totalUah: total,
promoCodeId: promoId,
shippingAddressJson: input.shipping as any,
contactEmail: input.email,
contactPhone: input.contactPhone ?? null,
items: {
create: cartItems.map((it) => ({
bookId: it.bookId,
quantity: it.quantity,
priceUah: it.book.priceUah,
titleSnapshot: it.book.titleUk,
})),
},
},
});

const intent = await this.stripe.createPaymentIntent({
amount: total,
currency: 'uah',
orderId: order.id,
orderNumber: order.number,
customerEmail: input.email,
});
```

```
await this.prisma.order.update({
where: { id: order.id },
data: { stripePaymentIntentId: intent.id },
});

return {
orderId: order.id,
orderNumber: order.number,
clientSecret: intent.client_secret,
total,
subtotal,
discount,
shipping,
};
}
}


// ===== apps/api/src/payments/stripe.service.ts — сервіс інтеграції
зі Stripe =====
import { Injectable, Logger } from '@nestjs/common';
import Stripe from 'stripe';

@Injectable()
export class StripeService {
private readonly logger = new Logger(StripeService.name);
private client: Stripe | null;

constructor() {
const key = process.env.STRIPE_SECRET_KEY;
if (!key || key.startsWith('sk_test_replace')) {
this.logger.warn('Stripe secret key not configured — payments will run
in mock mode');
this.client = null;
} else {
```

```
this.client = new Stripe(key, { apiVersion: '2024-11-20.acacia' as
Stripe.LatestApiVersion });
}
}

get stripe(): Stripe | null {
return this.client;
}

async createPaymentIntent(args: {
amount: number;
currency: string;
orderId: string;
orderNumber: string;
customerEmail: string;
}): Promise<{ id: string; client_secret: string }> {
if (!this.client) {
const id = `pi_mock_${args.orderId}`;
return { id, client_secret: `${id}_secret_mock` };
}
const intent = await this.client.paymentIntents.create({
amount: args.amount,
currency: args.currency,
automatic_payment_methods: { enabled: true },
receipt_email: args.customerEmail,
metadata: { orderId: args.orderId, orderNumber: args.orderNumber },
});
return { id: intent.id, client_secret: intent.client_secret as string
};
}

constructEvent(rawBody: Buffer, signature: string): Stripe.Event |
null {
const secret = process.env.STRIPE_WEBHOOK_SECRET;
if (!this.client || !secret || secret.startsWith('whsec_replace')) {
try {
```

```
return JSON.parse(rawBody.toString('utf8')) as Stripe.Event;
} catch {
return null;
}
}
return this.client.webhooks.constructEvent(rawBody, signature,
secret);
}

async retrieveIntent(id: string) {
if (!this.client) {
return { id, status: 'succeeded' } as any;
}
return this.client.paymentIntents.retrieve(id);
}
}


// ===== apps/api/src/payments/webhook.controller.ts — обробник
webhook Stripe =====
import { BadRequestException, Body, Controller, Headers, HttpCode,
Logger, Param, Post, Req } from '@nestjs/common';
import { Request } from 'express';
import { OrderStatus } from '@prisma/client';
import { PrismaService } from '../prisma/prisma.service';
import { StripeService } from './stripe.service';
import { MailService } from '../mail/mail.service';
import { Public } from '../common/guards/jwt-auth.guard';

@Public()
@Controller('webhooks')
export class WebhookController {
private readonly logger = new Logger(WebhookController.name);

constructor(
private prisma: PrismaService,
```

```
private stripeSvc: StripeService,
private mail: MailService,
) {}

@Post('stripe')
@HttpCode(200)
async stripe(@Req() req: Request, @Headers('stripe-signature') sig:
string) {
const raw = (req as any).body as Buffer;
let event: any;
try {
event = this.stripeSvc.constructEvent(raw, sig || '');
} catch (e: any) {
this.logger.error(`Webhook signature failed: ${e.message}`);
throw new BadRequestException('Invalid signature');
}
if (!event) throw new BadRequestException('Invalid event');

const existing = await this.prisma.webhookEvent.findUnique({
where: { provider_externalId: { provider: 'stripe', externalId:
event.id } },
});
if (existing) return { received: true, duplicate: true };

await this.prisma.webhookEvent.create({
data: { provider: 'stripe', externalId: event.id, type: event.type,
payload: event },
});

switch (event.type) {
case 'payment_intent.succeeded':
await this.handlePaymentSucceeded(event.data.object);
break;
case 'payment_intent.payment_failed':
await this.handlePaymentFailed(event.data.object);
break;
```

```
case 'charge.refunded':
await this.handleRefunded(event.data.object);
break;
}

return { received: true };
}

@Post('stripe/mock/:orderId')
@HttpCode(200)
async mockSuccess(@Param('orderId') orderId: string) {
const order = await this.prisma.order.findUnique({ where: { id:
orderId } });
if (!order) throw new BadRequestException('Order not found');
await this.handlePaymentSucceeded({
id: order.stripePaymentIntentId ?? `pi_mock_${orderId}`,
metadata: { orderId: order.id, orderNumber: order.number },
latest_charge: `ch_mock_${orderId}`,
});
return { ok: true };
}

private async handlePaymentSucceeded(intent: any) {
const orderId = intent?.metadata?.orderId as string | undefined;
if (!orderId) return;
const order = await this.prisma.order.findUnique({ where: { id:
orderId }, include: { items: true, user: true } });
if (!order) return;
if (order.status === OrderStatus.PAID || order.status ===
OrderStatus.PROCESSING) return;

await this.prisma.$transaction(async (tx) => {
for (const it of order.items) {
await tx.book.update({
where: { id: it.bookId },
data: { stock: { decrement: it.quantity } },
```

```
});
await tx.stockMovement.create({
data: { bookId: it.bookId, delta: -it.quantity, reason: 'ORDER_PAID',
orderId: order.id },
});
}
await tx.order.update({
where: { id: order.id },
data: {
status: OrderStatus.PAID,
paidAt: new Date(),
stripeChargeId: typeof intent.latest_charge === 'string' ?
intent.latest_charge : null,
},
});
await tx.cartItem.deleteMany({ where: { userId: order.userId } });
if (order.promoCodeId) {
await tx.promoCode.update({
where: { id: order.promoCodeId },
data: { usedCount: { increment: 1 } },
});
}
});

await this.mail.sendOrderPaid(order.contactEmail, order.number,
order.totalUah);
}

private async handlePaymentFailed(intent: any) {
const orderId = intent?.metadata?.orderId as string | undefined;
if (!orderId) return;
await this.prisma.order.update({
where: { id: orderId },
data: { status: OrderStatus.CANCELLED, cancelledAt: new Date() },
});
}
```

```
private async handleRefunded(charge: any) {
const intentId = charge?.payment_intent as string | undefined;
if (!intentId) return;
const order = await this.prisma.order.findUnique({ where: {
stripePaymentIntentId: intentId } });
if (!order) return;
await this.prisma.order.update({
where: { id: order.id },
data: { status: OrderStatus.REFUNDED },
});
}
}
```