

**Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра інженерії програмного забезпечення**

**КВАЛІФІКАЦІЙНА РОБОТА
ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»**

**РОЗРОБКА ВЕБЗАСТОСУНКУ ДЛЯ БРЕНДУ ОДЯГУ «MINDSET» ІЗ
ВИКОРИСТАННЯМ NEXT.JS ТА NESTJS**

**WEB APPLICATION DEVELOPMENT FOR THE CLOTHING BRAND
“MINDSET” USING NEXT.JS AND NESTJS**

спеціальність 121 «Інженерія програмного забезпечення»
освітня програма «Інженерія програмного забезпечення»

Виконав: здобувач вищої освіти
групи ІПЗ-41
Прополов Сергій Вадимович

Керівник:
Суринович Олена Миколаївна

Кваліфікаційну роботу
допущено до захисту
«__» _____ 20__ р.
Гарант освітньої програми:
к.т.н., доцент
Ліщина Наталія Миколаївна

Луцьк – 2026 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет комп'ютерних та інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Галузь знань: 12 «Інформаційні технології»

Спеціальність: 121 «Інженерія програмного забезпечення»

Освітня програма: «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

«__» _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Прополову Сергію Вадимовичу

(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи «Розробка вебзастосунку для бренду одягу «Mindset» із використанням Next.js та NestJS»
Керівник роботи: децент Суринович О. М.
затверджені наказом закладу вищої освіти від «31» грудня 2025 р. № 541/01-02
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи бакалавра «29» травня 2026 р.
3. Вихідні дані до роботи: предметна область електронної комерції та веброзробки, технології розробки програмного забезпечення (TypeScript, Next.js, NestJS), засоби роботи з базами даних (PostgreSQL, Prisma ORM, Supabase), інтеграція із зовнішніми сервісами (Stripe, Nova Post API), вимоги до вебзастосунків та забезпечення інформаційної безпеки.
4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити): аналіз сучасного стану ринку електронної комерції у сфері роздрібно́ї торгівлі одягом та огляд існуючих платформ, постановка завдання, визначення функціональних та нефункціональних вимог до вебзастосунку, обґрунтування вибору технологічного стеку (Next.js, NestJS, TypeScript, PostgreSQL), проєктування архітектури системи та структури бази даних, практична реалізація клієнтської та серверної частин застосунку, інтеграція платіжної системи Stripe та сервісу доставки Nova Post, обґрунтування принципів захисту інформаційно-комп'ютерної системи, тестування та налагодження програмного забезпечення.
5. Перелік графічного матеріалу: 15 рисунків, 9 лістингів, 1 таблиця.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз предметної області	Суринович О. М.		
Специфікація вимог до розробленої системи	Суринович О. М.		
Розробка об'єкта проектування	Суринович О. М.		
Нормоконтроль	Суринович О. М.		
Гарант ОП	Ліщина Н. М.		
Показник запозичень тексту	%		
Академічна доброчесність	Суринович О. М.		

7. Дата видачі завдання «3» лютого 2026 р.

№	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	Огляд літературних джерел по темі кваліфікаційної роботи бакалавра	до 14.02.2026 р.	
2	Аналіз проблеми розробки та впровадження об'єкту проектування	до 03.03.2026 р.	
3	Обґрунтування вибору шляхів, технологій і засобів вирішення поставленого завдання	до 14.03.2026 р.	
4	Розробка функціонально-структурної схеми роботи об'єкта проектування та проектування бази даних	до 14.04.2026 р.	
5	Практична реалізація об'єкта проектування та розробка бази даних	до 20.04.2026 р.	
6	Тестування та налагодження об'єкта проектування	до 04.05.2026 р.	
7	Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедрі	до 29.05.2026 р.	

Здобувач вищої освіти _____

Сергій ПРОПОЛОВ

Керівник кваліфікаційної роботи _____

Олена СУРИНОВИЧ

АНОТАЦІЯ

Прополов С. В. Розробка вебзастосунку для бренду одягу «Mindset» із використанням Next.js та NestJS. Рукопис. Кваліфікаційна робота бакалавра ОП «Інженерія програмного забезпечення» спеціальності «Інженерія програмного забезпечення». Луцький національний технічний університет. Луцьк, 2026.

Кваліфікаційна робота бакалавра складається зі вступу, трьох розділів, висновків, списку використаних джерел.

У першому розділі проведено дослідження предметної області електронної комерції у сфері роздрібного продажу одягу, проаналізовано існуючі платформи (Shopify, WooCommerce) та виявлено їхні переваги й обмеження, обґрунтовано доцільність власної розробки. У другому розділі визначено функціональні та нефункціональні вимоги до вебзастосунку, здійснено проектування архітектури системи з використанням UML-нотації та обґрунтовано вибір технологічного стеку: Next.js, NestJS, TypeScript, PostgreSQL, Prisma ORM, Stripe та Nova Post API. У третьому розділі описано процес практичної реалізації клієнтської та серверної частин застосунку, розроблено структуру бази даних, розглянуто питання автентифікації, авторизації та захисту персональних даних, а також наведено результати тестування і налагодження програмного забезпечення. У висновках підсумовано отримані результати, підтверджено досягнення мети роботи та повне виконання поставлених завдань.

Ключові слова: вебзастосунок, електронна комерція, Next.js, NestJS, TypeScript, PostgreSQL, Prisma ORM, Stripe, Nova Post, Supabase, REST API, серверний рендеринг.

ABSTRACT

Propolov S. V. Web Application Development for the Clothing Brand “Mindset” Using Next.js and Nestjs. Manuscript. Qualification work for bachelor’s degree in Software Engineering, specialty “Software Engineering”. Lutsk National Technical University. Lutsk, 2026.

The bachelor’s qualification work consists of an introduction, three chapters, conclusions, a list of references.

In the first chapter, a study of the e-commerce domain in the retail clothing segment was conducted. Existing platforms such as Shopify and WooCommerce were analyzed, their advantages and limitations were identified, and the necessity of developing a custom solution was substantiated. In the second chapter, the functional and non-functional requirements for the web application were defined, the system architecture was designed using UML notation, and the technology stack was justified: Next.js, NestJS, TypeScript, PostgreSQL, Prisma ORM, Stripe, and Nova Post API. The third chapter describes the practical implementation of both the client-side and server-side parts of the application, the development of the database structure, the consideration of authentication, authorization, and personal data protection mechanisms, as well as the results of software testing and debugging. The conclusions summarize the results obtained, confirm the achievement of the research objective, and verify the complete fulfillment of all defined tasks.

Keywords: web application, e-commerce, Next.js, NestJS, TypeScript, PostgreSQL, Prisma ORM, Stripe, Nova Post, Supabase, REST API, server-side rendering.

ЗМІСТ

ВСТУП.....	7
РОЗДІЛ 1 АНАЛІЗ РОЗРОБКИ ВЕБЗАСТОСУНКІВ ТА ПОСТАНОВКА ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ	9
1.1 Аналіз сучасного стану проблеми	9
1.2 Постановка завдання на кваліфікаційну роботу бакалавра	11
РОЗДІЛ 2 СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ	12
2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проєктування програмного забезпечення	12
2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання	15
РОЗДІЛ 3 РОЗРОБКА ВЕБЗАСТОСУНКУ ДЛЯ БРЕНДУ ОДЯГУ «MINDSET» ІЗ ВИКОРИСТАННЯМ NEXT.JS ТА NESTJS	20
3.1 Практична реалізація об'єкта проєктування	20
3.2 Розробка бази даних.....	32
3.3 Захист інформаційно-комп'ютерної системи.....	38
3.4 Тестування та налагодження інформаційно-комп'ютерної системи.....	43
ВИСНОВКИ.....	49
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	51
ДОДАТКИ.....	53

ВСТУП

Актуальність теми. Стрімкий розвиток цифрової економіки та поширення інтернет-торгівлі зумовлюють зростаючий попит на сучасні та технологічно досконалі платформи електронної комерції. Особливо помітною ця тенденція є у сфері роздрібної торгівлі одягом, де конкурентні переваги бізнесу значною мірою визначаються якістю цифрового представлення бренду та зручністю процесу здійснення покупок для кінцевого споживача.

Сучасний покупець очікує від онлайн-магазину не лише широкого асортименту товарів, а й інтуїтивно зрозумілого інтерфейсу, швидкого завантаження сторінок, захищеної системи оплати та зручної інтеграції з логістичними сервісами. Відповідно, бізнес-структури, що прагнуть забезпечити стійке зростання на ринку, змушені інвестувати у розробку власних вебплатформ, адаптованих під конкретні операційні потреби.

Наявні на ринку готові рішення для електронної комерції, такі як Shopify або WooCommerce, попри свою поширеність, часто не задовольняють специфічні вимоги нішевих брендів. Вони або надмірно обмежені у можливостях кастомізації, або вимагають значних витрат на ліцензування та підтримку, або не підтримують інтеграцію з українськими сервісами. Це формує об'єктивну потребу у розробці власних вебзастосунків на базі сучасних технологій.

Вирішення цих проблем потребує застосування сучасного технологічного стеку, орієнтованого на продуктивність, масштабованість та зручність супроводу. Використання фреймворків Next.js для клієнтської частини та NestJS для серверної дозволяє побудувати архітектурно виважену систему з чітким розподілом відповідальності між компонентами, підтримкою серверного рендерингу та розвиненою екосистемою сторонніх бібліотек.

Таким чином, тема кваліфікаційної роботи, присвячена розробці вебзастосунку для бренду одягу «Mindset» із використанням Next.js та NestJS, є актуальною, оскільки спрямована на вирішення реальних бізнес-задач у сфері електронної комерції із застосуванням сучасних інструментів веброботи.

Метою кваліфікаційної роботи є проєктування та реалізація вебзастосунку для інтернет-магазину із застосуванням фреймворків Next.js та NestJS та інтеграцією платіжної системи.

Для досягнення поставленої мети необхідно вирішити такі завдання:

- дослідити сучасні підходи до побудови вебзастосунків у сфері електронної комерції;
- визначити функціональні та нефункціональні вимоги до розроблюваної системи;
- обґрунтувати вибір технологічного стеку та архітектурних рішень;
- спроектувати структуру бази даних та взаємодію між компонентами системи;
- реалізувати клієнтську частину застосунку з використанням Next.js та TypeScript;
- розробити серверну частину на базі NestJS із підтримкою REST API;
- інтегрувати систему платежів Stripe та зовнішнє Nova Post API для отримання списку відділень та поштоматів;
- забезпечити механізми автентифікації, авторизації та захисту даних;
- провести тестування та налагодження розробленого програмного продукту.

Об'єктом дослідження є процес розробки повнофункціонального вебзастосунку для підтримки електронної комерції у сфері роздрібної торгівлі одягом.

Предметом дослідження є методи та засоби розробки вебзастосунків з використанням фреймворків Next.js і NestJS та їх інтеграції із зовнішніми сервісами.

Практичне значення роботи полягає у створенні готового до розгортання вебзастосунку, що забезпечує повний цикл електронної комерції – від перегляду каталогу товарів до оформлення та відстеження замовлень – і може бути використаний у реальній комерційній діяльності.

РОЗДІЛ 1

АНАЛІЗ РОЗРОБКИ ВЕБЗАСТОСУНКІВ ТА ПОСТАНОВКА ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

1.1 Аналіз сучасного стану проблеми

Об'єктом розробки у межах кваліфікаційної роботи є вебзастосунок для інтернет-магазину бренду одягу «Mindset», що функціонує за моделлю D2C (Direct-to-Consumer). Предметне середовище охоплює сферу електронної комерції в сегменті авторського одягу – галузь, що характеризується специфічними вимогами до технічної реалізації програмного забезпечення.

На відміну від традиційного ритейлу з дистриб'юторами та роздрібними мережами, модель D2C покладає на програмне забезпечення повний цикл відповідальності за взаємодію з клієнтом: від першого знайомства з брендом до моменту отримання товару. Це принципово змінює вимоги до архітектури системи, її продуктивності та функціональності.

У сегменті авторського одягу візуальна презентація продукту є критично важливою складовою комерційного успіху. Предметне середовище передбачає роботу з великими масивами медіаданих – зображеннями товарів у різних кольорових варіантах та ракурсах – які мають завантажуватися швидко для утримання уваги користувача. Це вимагає впровадження сучасних форматів стиснення зображень та використання хмарних сховищ для оптимальної доставки контенту.

Процеси предметного середовища можна розподілити на декілька ключових напрямів. Ведення каталогу передбачає складну структуру кожної товарної одиниці: категорії, колекції, розмірні сітки та колірні варіації. Обробка та супровід замовлень є ядром системи й охоплює перевірку коректності введених даних, безпечне проведення транзакцій через міжнародні платіжні шлюзи.

Важливим аспектом предметного середовища є управління взаємовідносинами з клієнтами: збирання та модерація відгуків, ведення історії

покупок та збереження персональних налаштувань між сесіями. Це дозволяє бренду формувати базу лояльних клієнтів, що є основою моделі D2C та запорукою довгострокової конкурентоспроможності.

Комплексний підхід до автоматизації охоплює як зовнішню частину, з якою безпосередньо взаємодіє покупець, так і внутрішню адміністративну, призначену для операційного керування бізнес-процесами. Такий підхід дозволяє мінімізувати вплив людського фактора та забезпечити безперебійну роботу магазину в режимі 24/7 без потреби у великому штаті менеджерів.

З метою обґрунтування доцільності власної розробки проведено порівняльний аналіз наявних платформ електронної комерції. Результати порівняння наведено в таблиці 1.1.

Таблиця 1.1 – Порівняльний аналіз платформ електронної комерції

Критерій порівняння	Shopify	WooCommerce	Власна розробка
Гнучкість кастомізації	Обмежена (шаблони+Liquid)	Середня (PHP теми +плагіни)	Повна (будь-яка логіка та UI)
Інтеграція з Nova Post	Відсутня	Часткова (сторонні плагіни)	Вбудована (каскадний вибір відділень)
Вартість використання	\$29-\$299/міс +комісія	Хостинг+платні плагіни	Мінімальна (лише хостинг Railway)
Контроль над кодовою базою	Відсутній	Частковий	Повний
SSR / ISR підтримка	Часткова	Відсутня	Повна (Next.js App Router)
Аналітика в адмін-панелі	Базова + платна	Через плагіни	Вбудована (власні діаграми продажів)
Відгуки лише від покупців	Через плагін	Через плагін	Вбудовано (перевірка через OrderItem)
Масштабованість архітектури	Обмежена платформою	Обмежена WordPress	Горизонтальна (модульний NestJS)

1.2 Постановка завдання на кваліфікаційну роботу бакалавра

Метою кваліфікаційної роботи є проєктування та реалізація повнофункціонального вебзастосунку для інтернет-магазину бренду одягу «Mindset», що забезпечує повний цикл електронної комерції – від перегляду каталогу до оплати та відстеження замовлень.

Для досягнення поставленої мети необхідно вирішити такі завдання:

- дослідити сучасні підходи до побудови вебзастосунків у сфері електронної комерції;
- визначити функціональні та нефункціональні вимоги до розроблюваної системи;
- обґрунтувати вибір технологічного стеку та архітектурних рішень;
- спроектувати структуру бази даних та взаємодію між компонентами системи;
- реалізувати клієнтську частину застосунку з використанням Next.js та TypeScript;
- розробити серверну частину на базі NestJS із підтримкою REST API;
- інтегрувати систему платежів Stripe та зовнішнє Nova Post API для отримання списку відділень та поштоматів;
- забезпечити механізми автентифікації, авторизації та захисту даних;
- провести тестування та налагодження розробленого програмного продукту.

Розроблений застосунок повинен відповідати сучасним стандартам якості вебпродуктів: забезпечувати високу продуктивність, захист персональних даних користувачів, коректну роботу на різних пристроях та зручність подальшого супроводу і розширення.

РОЗДІЛ 2

СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ

2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проєктування програмного забезпечення

Розробка вебзастосунку розпочиналася з детального аналізу потреб кінцевих користувачів та визначення вимог до системи. Відповідно до загальноприйнятої практики, вимоги поділяються на функціональні, що описують поведінку системи, та нефункціональні, що встановлюють обмеження на її характеристики.

До функціональних вимог, орієнтованих на кінцевого покупця, належать:

- перегляд каталогу товарів із можливістю фільтрації за категоріями, колекціями та пошуковим запитом;
- перегляд детальної сторінки товару з фотогалереєю, вибором варіантів та відгуками покупців;
- управління кошиком покупок із збереженням стану між сесіями;
- оформлення замовлення з вибором відділення служби доставки Nova Post та здійсненням оплати через Stripe;
- перегляд історії замовлень в особистому кабінеті;
- реєстрація та авторизація користувача, зокрема через обліковий запис Google;
- можливість залишати та переглядати відгуки на товари.

До функціональних вимог адміністративної панелі належать:

- управління колекціями, категоріями та товарами (створення, редагування, видалення);
- обробка та оновлення статусів замовлень;
- модерація відгуків покупців;
- перегляд аналітики продажів у вигляді інтерактивних діаграм;
- завантаження зображень товарів через хмарне сховище Supabase;
- управління ролями користувачів (адміністратор / покупець).

Для наочного представлення функціональних вимог на рисунку 2.1 наведено діаграму варіантів використання, що відображає взаємодію двох основних акторів системи – покупця та адміністратора – з відповідними функціями застосунку.

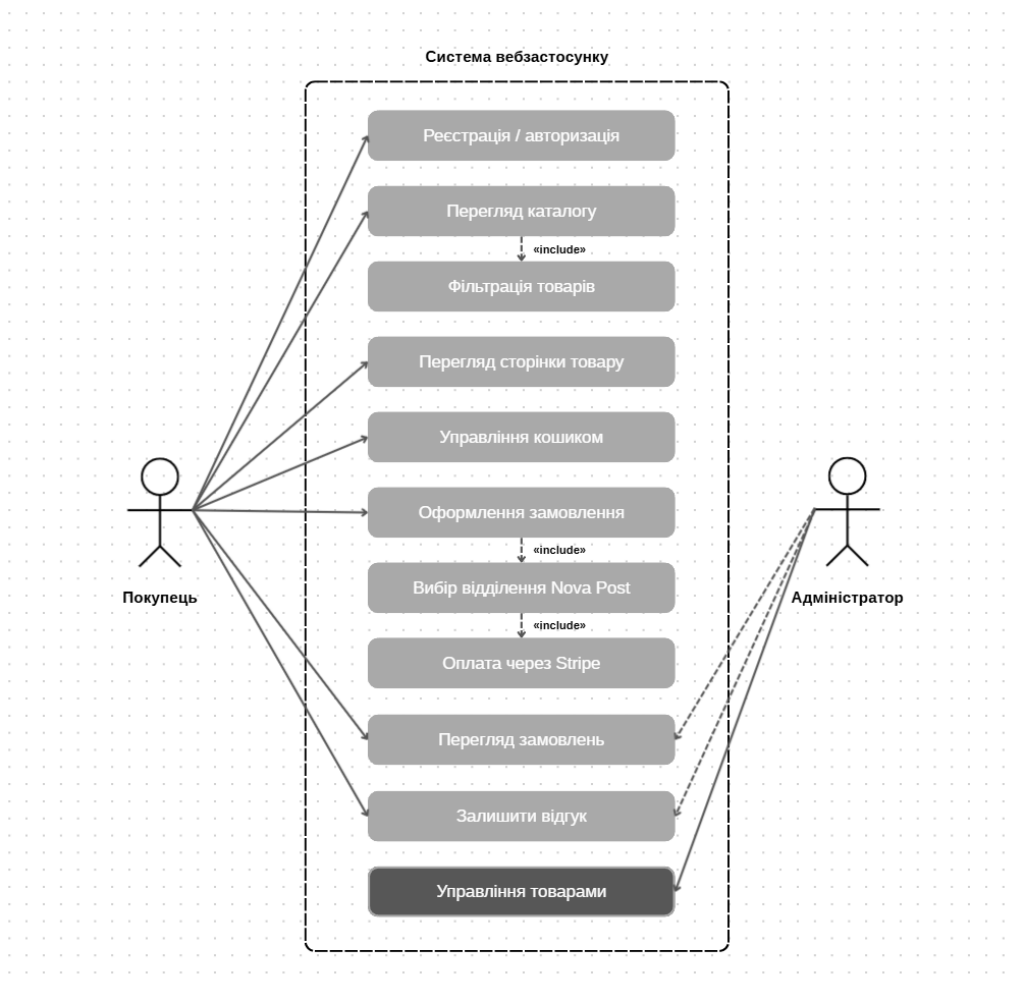


Рисунок 2.1 – Діаграма варіантів використання взаємодії покупця та адміністратора з функціями застосунку

Серед нефункціональних вимог визначено такі:

- продуктивність: час відповіді сервера на API-запити не повинен перевищувати 300 мс за нормального навантаження; клієнтські сторінки мають отримувати високі показники за метриками Core Web Vitals;

- безпека: зберігання паролів у захешованому вигляді, використання JWT-токенів із коротким часом дії та refresh-токенів, захист адміністративних маршрутів;
- масштабованість: архітектура системи має допускати горизонтальне масштабування без суттєвих змін у кодовій базі;
- надійність: коректна обробка помилок на рівні API з відповідними HTTP-статусами та інформативними повідомленнями;
- зручність супроводу: дотримання принципів чистого коду, модульна структура проєкту, наявність документації API через Swagger.

На етапі проєктування визначено трирівневу архітектуру застосунку: клієнтська частина (Next.js), серверна частина (NestJS) та рівень даних (PostgreSQL). Взаємодія між рівнями здійснюється через REST API. Зовнішні інтеграції (Stripe, Supabase, Resend) підключаються виключно на рівні сервера. Детальний вигляд рівнів показано в діаграмі компонентів на рисунку 2.2.

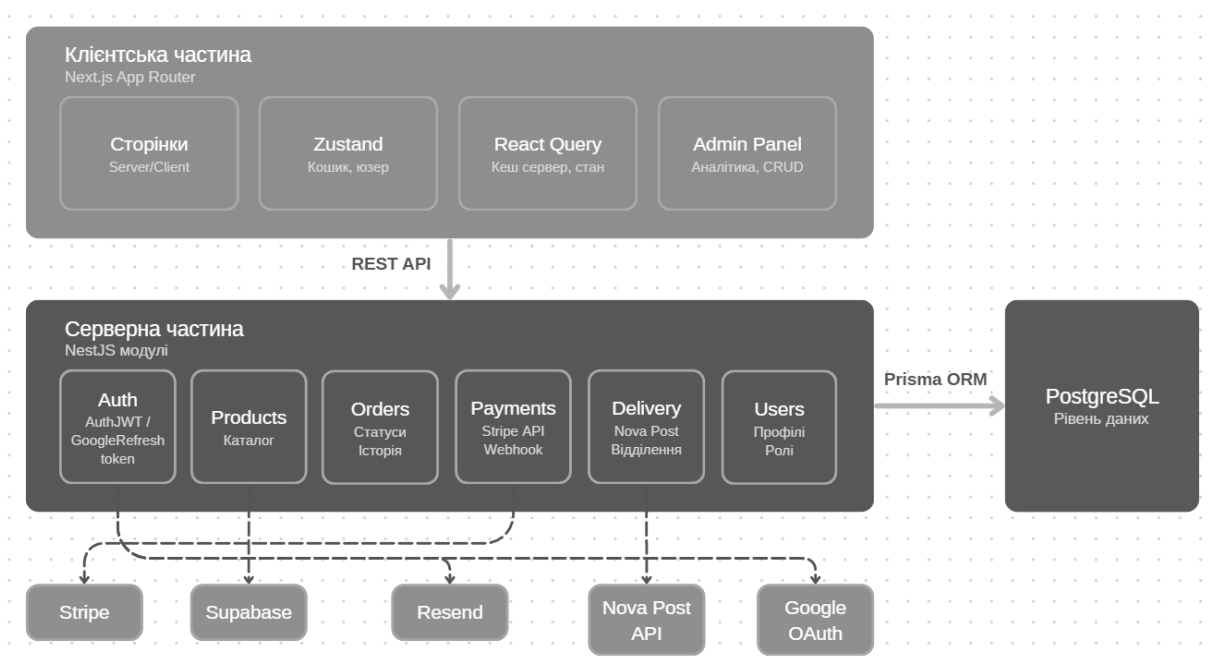


Рисунок 2.2 – Діаграма компонентів трирівневої структури додатку

Клієнтська частина спроектована за принципом компонентної архітектури з розподілом на серверні та клієнтські компоненти відповідно до можливостей Next.js App Router. Глобальний стан застосунку (кошик, дані користувача)

управляється за допомогою Zustand, тоді як серверний стан (дані товарів, замовлень та інше) кешується та інвалідується через React Query.

Серверна частина реалізована як набір модулів NestJS, кожен з яких відповідає за окрему предметну область: автентифікація, товари, замовлення, користувачі, платежі, доставка. Кожен модуль містить контролер, сервіс та необхідні DTO-класи з валідацією через class-validator.

2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання

Вибір технологічного стеку є одним із ключових архітектурних рішень при розробці будь-якого вебзастосунку, оскільки від нього безпосередньо залежать продуктивність системи, зручність її підтримки та можливості масштабування. У процесі вибору інструментів для розробки вебзастосунку враховувалися такі критерії: зрілість технології та активність її спільноти, наявність розвиненої екосистеми бібліотек, підтримка TypeScript та сумісність компонентів стеку між собою.

Next.js 16 обрано як основний фреймворк для побудови клієнтської частини застосунку [1]. Ключовою перевагою Next.js над альтернативними React-фреймворками є підтримка кількох стратегій рендерингу в межах одного проєкту, що дозволяє обирати оптимальний підхід для кожного типу сторінок.

Серверний рендеринг (SSR) застосовується для сторінок, що потребують актуальних даних при кожному. Генерація HTML на стороні сервера забезпечує швидке відображення першого контенту та позитивно впливає на показники пошукової оптимізації, оскільки пошукові роботи отримують повністю сформовану розмітку.

Інкрементальна статична регенерація (ISR) використовується для сторінок каталогу та деталей товарів. Цей підхід дозволяє попередньо генерувати HTML під час збірки та автоматично оновлювати його у фоновому режимі після закінчення заданого інтервалу часу, без необхідності повного перерозгортання

застосунку. Завдяки цьому сторінки товарів завантажуються зі швидкістю статичних файлів, але при цьому завжди відображають актуальні дані.

Клієнтський рендеринг (CSR) застосовується для інтерактивних компонентів, що не потребують індексування пошуковими системами: форм фільтрації, інтерфейсу адміністративної панелі та аналітичних дашбордів.

App Router, введений у Next.js 13 та суттєво вдосконалений у версії 16, забезпечує природний поділ компонентів на серверні та клієнтські. Серверні компоненти виконуються виключно на сервері та не додають JavaScript до клієнтського бандлу, що зменшує його обсяг та покращує показники Core Web Vitals.

TypeScript застосовується як мова програмування на обох рівнях застосунку. Статична типізація дозволяє виявляти помилки на етапі компіляції, покращує якість автодоповнення в IDE та є особливо цінною при роботі з API-контрактами між клієнтською та серверною частинами. Спільне використання типів TypeScript через пакети або монорепозиторій дозволяє гарантувати узгодженість структур даних на обох рівнях [2].

TailwindCSS 4 використовується як утилітарний CSS-фреймворк. На відміну від традиційних підходів до стилізації, Tailwind пропонує набір атомарних CSS-класів, що застосовуються безпосередньо у JSX-розмітці компонентів. Це усуває необхідність підтримки окремих CSS-файлів, запобігає конфліктам стилів між компонентами та значно прискорює процес верстки адаптивних інтерфейсів. Четверта версія TailwindCSS принесла суттєве покращення продуктивності збірки завдяки повністю переписаному рушію обробки стилів [3].

Shadcn/ui обрано як бібліотеку UI-компонентів. На відміну від традиційних бібліотек компонентів, Shadcn/ui не є npm-пакетом – її компоненти копіюються безпосередньо до кодової бази проєкту, що забезпечує повний контроль над їхньою реалізацією та стилізацією. Компоненти побудовані на базі Radix UI – бібліотеки примітивів із повною підтримкою доступності (ARIA-атрибути, навігація з клавіатури) [4].

Для управління глобальним станом застосунку обрано Zustand – легковагову бібліотеку управління станом [5]. На відміну від Redux, Zustand не потребує значного шаблонного коду та природно інтегрується з React Hooks. Стан кошика покупок зберігається у Zustand із синхронізацією до localStorage для збереження між сесіями.

React Query використовується для управління серверним станом: кешування відповідей API, відстеження стану завантаження та автоматичної інвалідації кешу після мутацій [6]. Це дозволяє уникнути надлишкових запитів до сервера та забезпечує коректне оновлення інтерфейсу після змін даних.

Для відображення аналітичних даних у розділі адміністративної панелі використовується бібліотека Recharts, побудована на базі D3.js [7]. Вона надає набір декларативних React-компонентів для побудови різних типів діаграм із підтримкою адаптивного відображення та кастомізації.

NestJS обрано як основний фреймворк для побудови серверної частини [8]. Ключовою особливістю NestJS є модульна архітектура, що нагадує підходи, прийняті у Angular та Spring Boot. Кожен функціональний модуль є самодостатнім і містить контролер для обробки HTTP-запитів, сервіс із бізнес-логікою та репозиторій або пряме звернення до ORM. Вбудована підтримка впровадження залежностей (Dependency Injection) забезпечує слабе зв'язування компонентів, що спрощує їх тестування та заміну.

NestJS побудований поверх Express.js, проте надає додатковий рівень абстракції через декоратори TypeScript. Це дозволяє декларативно описувати маршрути, параметри запитів, заголовки та ролі доступу безпосередньо у коді контролерів.

Prisma ORM обрано для роботи з реляційною базою даних [9]. Основними перевагами Prisma є: автоматична генерація типізованого клієнта на основі схеми, що виключає невідповідність між структурою БД та типами TypeScript; зручний декларативний синтаксис запитів без написання сирого SQL; система міграцій, що дозволяє відтворювати структуру бази даних на будь-якому

середовищі; а також вбудований інструмент Prisma Studio для візуального перегляду та редагування даних у процесі розробки.

PostgreSQL обрано як систему управління базами даних [10]. Ця реляційна СУБД є одним із найпоширеніших рішень у корпоративній розробці завдяки своїй надійності, підтримці транзакцій із рівнем ізоляції `SERIALIZABLE`, розвиненій системі індексів та можливості роботи з JSON-полями через тип `JSONB`, що використовується для зберігання масивів розмірів та варіантів товарів.

Stripe API обрано як платіжну систему завдяки її широкому поширенню, детальній документації та надійній інфраструктурі [11]. Інтеграція реалізована через механізм Checkout Sessions: сервер створює сесію оплати та повертає клієнту URL для редиректу на захищену сторінку оплати Stripe. Після завершення транзакції Stripe надсилає подію на вебхук-ендпоінт сервера, де вона верифікується та обробляється – статус замовлення оновлюється у базі даних. Така схема виключає необхідність обробки платіжних даних на власному сервері, що суттєво спрощує відповідність вимогам стандарту PCI DSS.

Nova Post API інтегровано для автоматизації вибору відділення доставки в процесі оформлення замовлення [12]. Покупець вводить назву населеного пункту, застосунок надсилає запит до API та відображає перелік доступних відділень. Обрана адреса зберігається разом із замовленням і відображається в адміністративній панелі для обробки.

Supabase Storage використовується як хмарне об'єктне сховище для зображень товарів [13]. Завантаження файлів відбувається на сервері через Multer, після чого файл передається до Supabase через офіційний SDK. Отримане публічне посилання зберігається у базі даних і використовується клієнтською частиною для відображення зображень.

Resend забезпечує відправлення транзакційних електронних листів: підтвердження реєстрації, сповіщення про зміну статусу замовлення та відновлення пароля [14]. Використання Resend гарантує високу доставлюваність листів та надає інструменти для моніторингу відправлень. На відміну від

Nodemailer, який потребує власного SMTP-сервера, Resend надає REST API з офіційним SDK для Node.js, що спрощує інтеграцію у NestJS. Порівняно з SendGrid, Resend має простіший API без надлишкової конфігурації та безкоштовний план із достатнім лімітом для MVP-проєкту.

У процесі розробки клієнтської частини застосовано компонентно-орієнтований підхід, при якому інтерфейс розбивається на ієрархію незалежних, повторно використовуваних компонентів. Кожен компонент відповідає за власну логіку відображення та не має прямої залежності від інших компонентів на тому ж рівні ієрархії.

Серверна частина побудована із застосуванням принципів доменно-орієнтованого проєктування, де кожен модуль відповідає окремій предметній області системи. Бізнес-логіка зосереджена у сервісах та ізольована від деталей транспортного рівня (HTTP-контролерів).

Для контролю версій використовується Git із стратегією feature-branch: кожна нова функціональність розробляється у окремій гілці та інтегрується до основної через механізм pull request. Розгортання застосунку здійснено на платформі Railway, яка забезпечує автоматичний CI/CD-пайплайн, де кожне злиття до основної гілки автоматично ініціює процес збірки та розгортання нової версії застосунку у продуктивному середовищі [15].

РОЗДІЛ 3

РОЗРОБКА ВЕБЗАСТОСУНКУ ДЛЯ БРЕНДУ ОДЯГУ «MINDSET» ІЗ ВИКОРИСТАННЯМ NEXT.JS ТА NESTJS

3.1 Практична реалізація об'єкта проєктування

Розробка вебзастосунку «Mindset» здійснювалася у форматі монорепозиторію, що містить дві незалежні директорії: `client` для клієнтської частини та `server` для серверної. Така організація проєкту забезпечує логічне розмежування між рівнями застосунку, водночас зберігаючи єдину кодову базу під спільним контролем версій. Це спрощує синхронізацію змін між клієнтом і сервером та прискорює локальну розробку, оскільки обидві частини запускаються в межах одного проєкту.

Клієнтська частина побудована за методологією Feature-Sliced Design (FSD) – архітектурним підходом, що передбачає організацію коду за функціональними доменами, а не за технічними типами файлів [16]. Кодова база поділена на кілька рівнів: `app` (маршрутизація та конфігурація), `features` (функціональні модулі), `shared` (спільні утиліти, UI-компоненти та хуки) та `store` (глобальний стан).

Маршрутизація реалізована засобами Next.js App Router. Застосовано підхід групування маршрутів через `route groups`: група (`shop`) об'єднує всі публічні сторінки магазину, а група (`admin`) – сторінки адміністративної панелі. Це дозволяє використовувати різні `layout`-компоненти для кожної з груп без впливу на структуру URL.

Маршрутна структура клієнтської частини охоплює такі розділи: головна сторінка магазину, динамічні сторінки колекцій та категорій, сторінки окремих товарів з підсторінкою відгуків, сторінка кошика, сторінка оформлення замовлення, сторінки результату оплати, особистий кабінет покупця з підрозділом замовлень, сторінки авторизації (вхід, реєстрація, відновлення пароля, скидання пароля, верифікація email, OAuth callback) та розділ обраних товарів. Окремою групою розміщена адміністративна панель із підрозділами:

головна сторінка, управління колекціями та категоріями, атрибутами товарів, замовленнями, відгуками, статистикою та обліковим записом адміністратора.

Директорія features містить самодостатні модулі для кожного функціонального домену: auth, products, categories, collections, orders, reviews, checkout, а також підмодулі для роботи зі стороною магазину та адміністративної частини, де зосереджено управління кольорами, розмірами та типами товарів. Кожен модуль містить власні підпапки: api для функцій запитів, components для React-компонентів, hooks для React Query хуків, modals для діалогових вікон та types для TypeScript-інтерфейсів.

Для того, щоб використовувати хуки React Query потрібно ініціалізувати його як показано в лістингу 3.1 та приєднати до кореню проекту як зображено в лістингу 3.2.

Лістинг 3.1 – Код створення обгортки «квері провайдеру» та «квері клієнта»
для проекту [6]

```
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      staleTime: 60 * 1000, // час
      retry: 1, // кількість повторних спроб
    },
  },
});
export function QueryProvider({ children }: { children: ReactNode }) {
  return (
    <QueryClientProvider client={queryClient}> // провайдер
      {children}
      <ReactQueryDevtools initialIsOpen={false} /> // компонент для
відображення даних в devtools
    </QueryClientProvider>
  );
}
```

кінець лістингу 3.1

Лістинг 3.2 – Огортання проекту в «квері провайдер»

```

export default async function RootLayout({
  children,
}: Readonly<{ children: React.ReactNode }>) {
  const { user, accessToken } = await getCurrentUserSSR();

  return (
    <html
      lang="en"
      className={` ${roboto.className}  ${ballet.variable}
${perandory.variable}`}
    >
      <body className="relative bg-black">
        <QueryProvider> // квері провайдер
          <Toaster theme="dark" position="bottom-center" />
          <AuthStoreInitializer
            user={user}
            accessToken={accessToken || null}
          />
          {children}
        </QueryProvider>
      </body>
    </html>
  );
}

```

кінець лістингу 3.2

Модальні вікна реалізовано за допомогою механізму «порталів» через `createPortal` з `react-dom`, що дозволяло відкривати та показувати модальні вікна в будь-якому місці додатку. Код модального вікна з інформацією про категорію зображено в додатку А.

Директорія `shared` містить компоненти та утиліти, що використовуються в різних модулях: спільні UI-компоненти, зокрема кнопки, поля введення, скелетони завантаження, селектори та обгортки для форм і модальних вікон; компоненти макету сторінки – шапка, підвалина, навігація; діаграми для аналітики; іконки; а також утиліти та допоміжні функції.

Глобальний стан застосунку управляється трьома Zustand-сторями: `useCartStore` для кошика покупок, `useFavoritesStore` для списку обраних товарів

та `userStore` для даних авторизованого користувача. Стани кошика та обраних товарів персистуються у `localStorage`, що дозволяє зберігати їх між сесіями навіть для неавторизованих користувачів.

Для обробки завантажувальних станів реалізовано розгалужену систему `skeleton`-компонентів. Для кожного основного UI-блоку існує відповідний скелетон. Це забезпечує коректне відображення інтерфейсу під час завантаження даних без використання спінерів, що покращує сприйняття швидкодії застосунку.

Для SEO-оптимізації реалізовано файли `sitemap.ts` та `robots.ts`, що генеруються динамічно на основі даних із API. Також додано ендпоінт `/api/revalidate`, який дозволяє примусово інвалідувати ISR-кеш конкретних сторінок після внесення змін через адміністративну панель.

Загальна структура папок клієнтської частини зображена на рисунку 3.1.

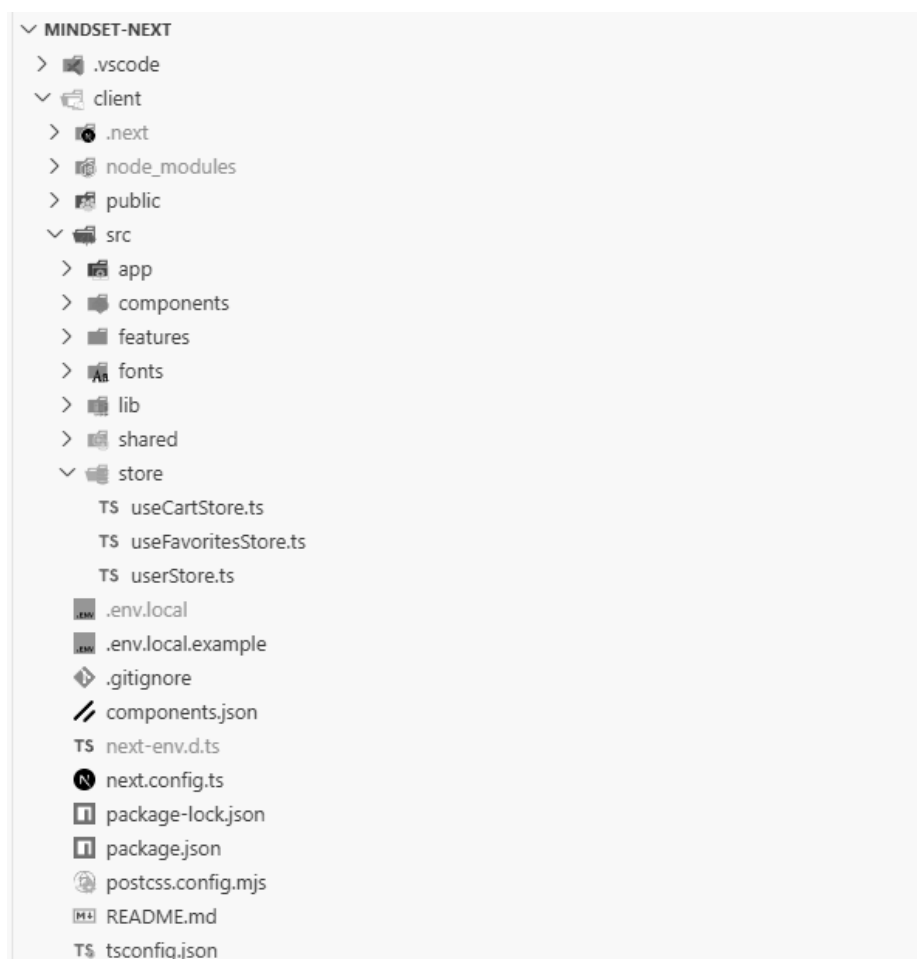


Рисунок 3.1 – Структура папок клієнтської частини застосунку

Сторінка оформлення замовлення є одним із ключових функціональних компонентів застосунку. Вона реалізована як клієнтський компонент і побудована на базі бібліотеки React Hook Form, що забезпечує ефективне управління станом форми та валідацію полів без зайвих перерендерів [17].

Форма оформлення замовлення охоплює два блоки: дані одержувача та адреса доставки. Блок даних одержувача містить поля для введення повного імені (з валідацією формату «Прізвище Ім'я По батькові» латинськими літерами), номера телефону у форматі українського оператора та електронної пошти. Для авторизованих користувачів ці поля автоматично попередньо заповнюються даними із профілю після завантаження даних користувача.

Блок адреси доставки реалізований через каскадні селектори Nova Post: область, місто, відділення. Дані для кожного рівня завантажуються послідовно – список міст підвантажується після вибору області, список відділень – після вибору міста. Кожен крок завантаження супроводжується відповідним станом, що блокує наступний селектор до завершення запиту. Візуальний вигляд форми показано на рисунку 3.2.

MINI

PLACEING AN ORDER

SHIPPING DETAILS

FULL NAME*

Petrenko Petro Petrovych

PHONE NUMBER*

+380XXXXXXXX

E-MAIL*

propolovchesnochok@gmail.com

DELIVERY ADDRESS

REGION*

Choose region*

CITY*

Choose city*

POST OFFICE*

Choose post office*

The data you enter is used to automatically create an express invoice in the Nova Post system.

Please check the address before confirming the order. If you change your number or city, update your account information or contact [technical support](#).

Рисунок 3.2 – Візуальний вигляд форми створення замовлення

У разі помилки при завантаженні відображається повідомлення про помилку, а користувач отримує сповіщення через toast-нотифікацію бібліотеки Sonner. Приклад вигляду нотифікацій показано на рисунку 3.3.

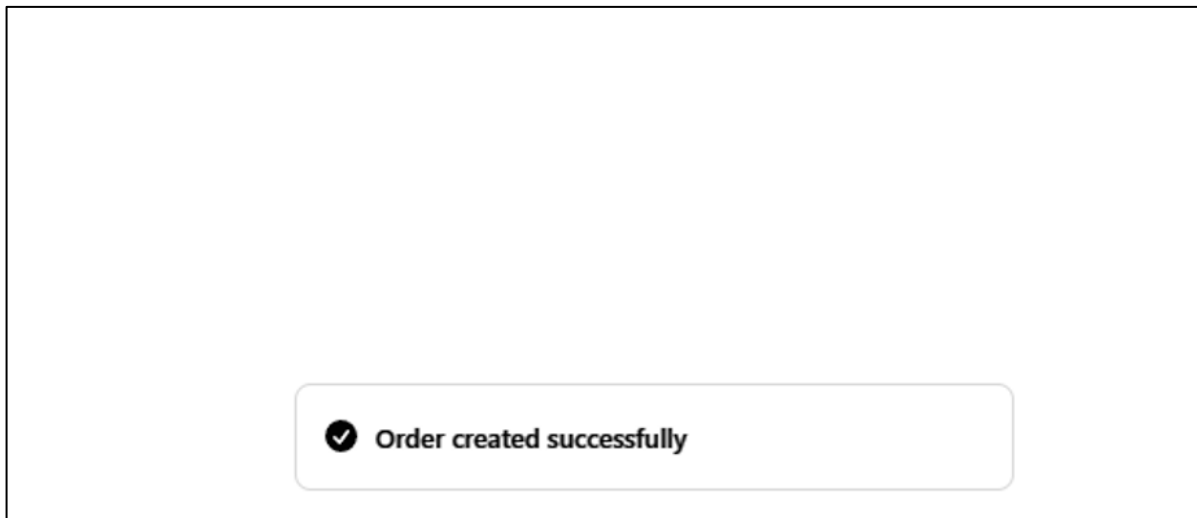


Рисунок 3.3 – Приклад вигляду нотифікації створеного замовлення

Права частина сторінки відображає підсумок замовлення: перелік товарів із зображеннями, назвами, розмірами та цінами, а також загальну суму. Нижче розміщено вибір способу оплати: картою через Stripe або готівкою при отриманні (cash on delivery). Спосіб оплати реалізований через компонент Controller бібліотеки React Hook Form для коректної інтеграції кастомного UI з логікою форми.

При підтвердженні замовлення зі способом оплати Stripe виконується двоетапний процес: спочатку на сервер надсилається запит на створення замовлення у базі даних, після чого з отриманим ідентифікатором замовлення виконується запит до ендпоінту `/stripe/checkout` для ініціації платіжної сесії. У відповідь сервер повертає URL сторінки оплати Stripe, на яку виконується редирект користувача через `window.location.href`. При виборі оплати готівкою замовлення одразу зберігається зі статусом очікування без переходу до платіжної системи.

Процес оплати в застосунку реалізовано у вигляді послідовного багатокрокового сценарію. Ініціювання замовлення відбувається з кошика

покупок: користувач переглядає обрані товари та підтверджує намір оформити замовлення (рис. 3.4).

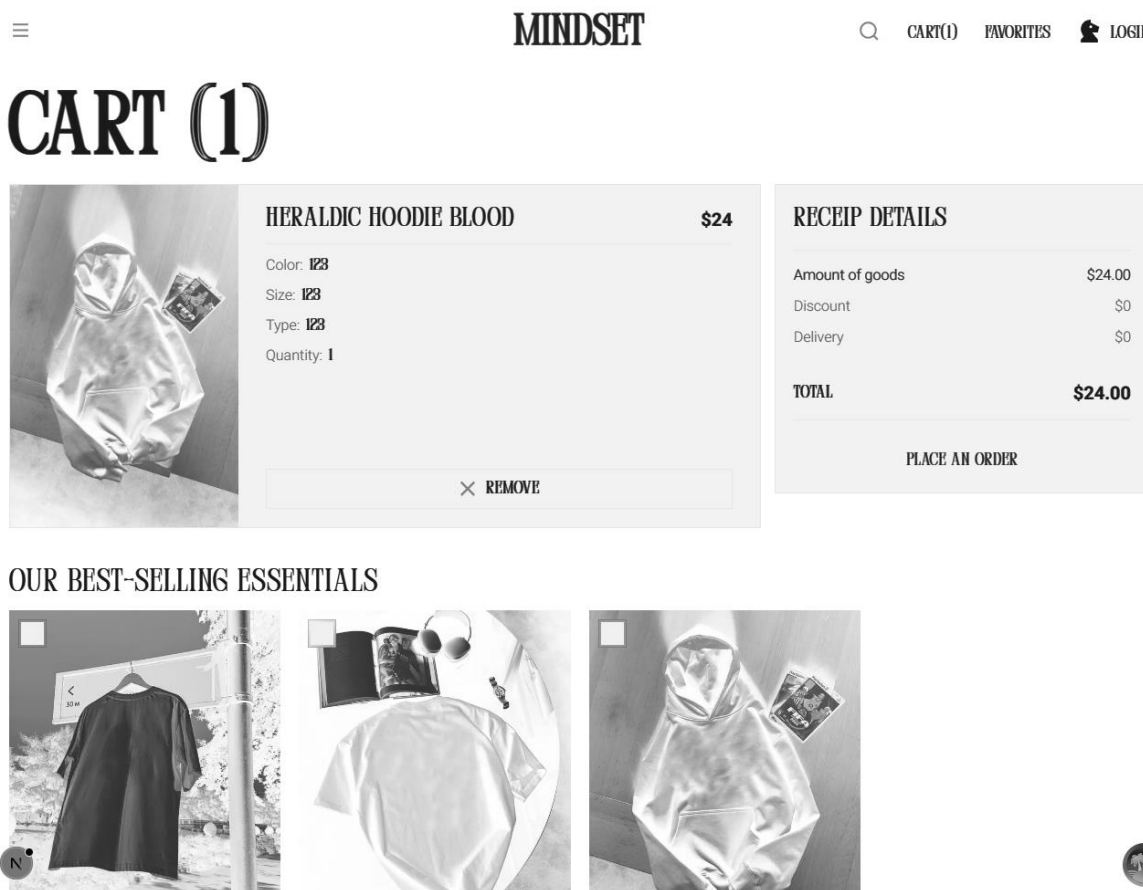


Рисунок 3.4 – Кошик користувача з доданими товарами

На наступному етапі користувач заповнює форму оформлення замовлення (рис. 3.5, ліва частина), де вказує контактні дані та обирає спосіб доставки. Для визначення адреси отримання реалізовано каскадний вибір: спочатку обирається область, потім населений пункт, після чого система автоматично підвантажує через Nova Post API перелік доступних відділень або поштоматів у вказаному місті. Це дозволяє мінімізувати кількість помилок при введенні адреси та гарантує коректність даних для подальшої обробки замовлення.

Після заповнення форми користувачу пропонується обрати спосіб оплати: онлайн через платіжну систему Stripe або готівкою під час отримання товару (рис. 3.5, права частина).

SHIPPING DETAILS

FULL NAME*

Petrenko Petro Petrovych

PHONE NUMBER*

+380XXXXXXXXXX

E-MAIL*

petro@gmail.com

The data you enter is used to automatically create an express invoice in the Nova Post system.

Please check the address before confirming the order. If you change your number or city, update your account information or contact technical support.

DELIVERY ADDRESS

REGION*

Volyńska

CITY*

Luts'k

POST OFFICE*

Viddilennia №1: vul. Ivana Korsaka, 1 (poru)

ORDER INFORMATION

Name	Color	Size	Type	Price	Quantity	Sum
Heraldic hoodie blood	123	123	123	24 \$	1	24 \$
Sum of goods:						24 \$
Delivery:						According to carrier tariffs
Total payable:						24 \$

PAYMENT OPTION*

Pay with card
(Secure with Stripe)

Cash on delivery

CONFIRM ORDER



Heraldic collection

Home
Cart
Favorites

Our policy
Refund
FAQ

☎ +38 (097) 00 00 000
✉ job.tenshi@gmail.com

У разі вибору оплати при отриманні замовлення одразу створюється у базі даних із відповідним статусом, що підтверджує його прийняття в обробку без попередньої транзакції.

У разі вибору онлайн-оплати відбувається наступне: система створює замовлення та Stripe Checkout Session на сервері, після чого користувача автоматично перенаправляють на захищену сторінку оплати Stripe. Етапи заповнення та візуальний вигляд Stripe Checkout зображено на рисунках 3.6-3.8. Там необхідно заповнити платіжні реквізити та підтвердити транзакцію. Якщо оплата пройшла успішно користувача перекидає на сторінку сайту магазину з вікном успішної оплати (рис. 3.9), Stripe надсилає подію на вебхук-ендпоінт сервера, де вона верифікується та обробляється: статус замовлення оновлюється на «paid», а користувач отримує на електронну пошту підтвердження із деталями замовлення (рис. 3.10). У разі невдалої транзакції – через відхилення картки або скасування оплати користувачем – замовленню присвоюється відповідний статус і користувач отримує повідомлення про помилку.

← Mindset sandbox Sandbox

Heraldic hoodie blood
\$24.00

Pay with link

OR

Contact information

Email
email@example.com

Payment method

Card

Card information

1234 1234 1234 1234

MM / YY CVC

Cardholder name
Full name on card

Country or region
Belgium

☐ Save my information for faster checkout
Pay securely at Mindset sandbox and everywhere is accepted.

Pay

Powered by | Terms Privacy

Рисунок 3.6 – Пуста сторінка Stripe Checkout

← Mindset sandbox Sandbox

Heraldic hoodie blood
\$24.00

Pay with link

OR

Contact information

Email
job.tenshi@gmail.com

Payment method

Card

Card information

4242 4242 4242 4242

04 / 44 444

Cardholder name
Serhii Prokolov

Country or region
Belgium

☐ Save my information for faster checkout
Pay securely at Mindset sandbox and everywhere is accepted.

Pay

Powered by | Terms Privacy

Рисунок 3.7 – Сторінка Stripe Checkout з даними карти та країною користувача

Mindset sandbox **Sandbox**

Heraldic hoodie blood
\$24.00

Pay with link

OR

Contact information

Email
job.tenshi@gmail.com

Payment method

Card

Card information

4242 4242 4242 4242

04 / 44 444

Cardholder name
Serhii Propolov

Country or region
Belgium

Save my information for faster checkout
Pay securely at Mindset sandbox and everywhere [Link](#) is accepted.

Processing

Powered by stripe | [Terms](#) [Privacy](#)

Рисунок 3.8 – Сторінка Stripe Checkout з статусом обробки даних

PAYMENT SUCCESSFUL

Thank you for your purchase. Your order has been created.

Order number: **cmpnswf7c0000uf9864mg3nlv**

CONTINUE SHOPPING

Рисунок 3.9 – Сторінка сайту магазину з вікном успішної оплати та номером замовлення

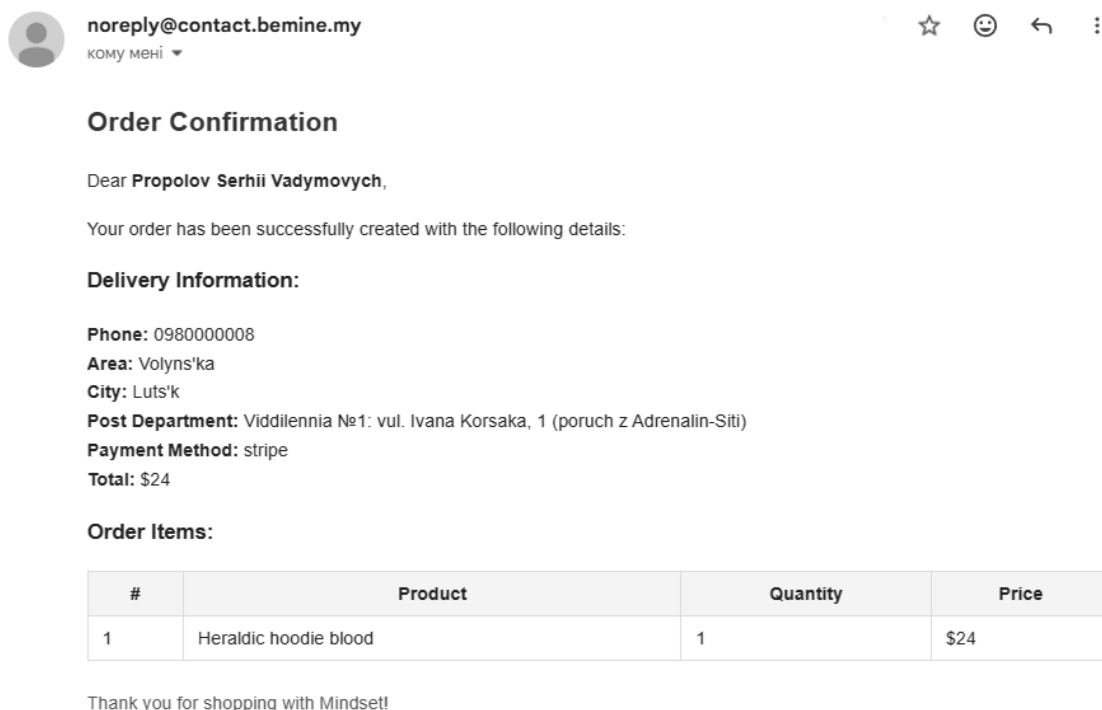


Рисунок 3.10 – Приклад повідомлення яке приходить на пошту у разі створення замовлення

Серверна частина організована як набір функціональних NestJS-модулів, згрупованих у дві кореневі групи: `admin` та `shop`. Такий поділ відображає різні ролі користувачів і відповідні набори дозволів.

Група модулів `admin` містить: `admin-categories`, `admin-collections`, `admin-products`, `admin-orders`, `admin-reviews`, `admin-colors`, `admin-sizes`, `admin-types`, `admin-user`, `admin-recent-actions` та `admin-todo`. Кожен модуль надає повний набір CRUD-операцій для відповідної сутності та захищений перевіркою ролі `ADMIN` на рівні `guard`. Реалізацію «гварду» ролей показано в лістингу 3.3.

Лістинг 3.3 – «Гвард» для перевірки ролей в авторизованих користувачів [8]

```
@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private reflector: Reflector) {}
  canActivate(context: ExecutionContext): boolean | Promise<boolean> |
  Observable<boolean> {
    const requiredRoles =
    this.reflector.getAllAndOverride<Role[]>(ROLES_KEY, [
```

```

        context.getHandler(),
        context.getClass(),
    ]);
    if (!requiredRoles) return true;
    const request = context.switchToHttp().getRequest<Request & {
user: AuthenticatedUser }>();
    const user = request.user;
    return requiredRoles.some((role) => user.role === role);
}
}

```

кінець лістингу 3.3

Група модулів shop охоплює: shop-products, shop-collections, shop-orders, shop-reviews, shop-cart, shop-favorites, shop-user, shop-user-address. Ці модулі реалізують функціональність для публічних сторінок магазину та особистого кабінету покупця.

Окрім функціональних груп, реалізовано ряд інфраструктурних модулів: auth – автентифікація та авторизація, files – завантаження зображень, nova-post – інтеграція з API Нової Пошти, stripe – обробка платежів, supabase – клієнт хмарного сховища, search – пошук товарів по магазину, revalidate – інвалідація ISR-кешу Next.js, prisma – провайдер клієнта бази даних.

Модуль auth реалізує чотири стратегії Passport.js [18]: local для авторизації за логіном та паролем, jwt-access та jwt-refresh для перевірки JWT-токенів, а також google для OAuth 2.0. Модуль містить власні декоратори @Public() для позначення публічних маршрутів та @Roles() для обмеження доступу за роллю, які застосовуються у комбінації з відповідними guards. Окремий сервіс clean-up.service.ts відповідає за очищення протермінованих refresh-токенів із бази даних. Код «стратегій» зображено в лістингах в додатку Б.

Модуль admin-recent-actions реєструє останні дії адміністратора у системі, що дозволяє відстежувати активність у адмін-панелі. Модуль admin-todo реалізує особистий список задач адміністратора безпосередньо в панелі керування.

Модуль search забезпечує повнотекстовий пошук по каталогу товарів із підтримкою пошуку за назвою, описом та атрибутами. Результати пошуку

відображаються у модальному вікні ShopSearchModal на клієнтській частині з динамічним підвантаженням у процесі введення запиту.

Модуль `revalidate` надає захищений ендпоінт, що приймає запити від адміністративної панелі та ініціює примусову ревалідацію відповідних ISR-сторінок Next.js через виклик `/api/revalidate` з секретним ключем. Завдяки цьому зміни у каталозі товарів, колекціях або категоріях одразу відображаються на публічних сторінках без необхідності перерозгортання застосунку.

Модуль `email` реалізує відправлення транзакційних електронних листів через сервіс Resend. `EmailService` ін'єктує `ConfigService` для отримання API-ключа (`RESEND_API_KEY`) та адреси відправника (`MAIL_FROM`) зі змінних середовища. Сервіс реалізує чотири методи відправлення: верифікація електронної пошти після реєстрації, вітальний лист із підтвердженням створення облікового запису, скидання пароля за токеном та підтвердження замовлення з переліком товарів і деталями доставки. Усі методи обгорнуті в блок `try/catch` з централізованою обробкою помилок через `handleEmailError`. Код сервісу наведено в додатку В.

3.2 Розробка бази даних

База даних вебзастосунку побудована на основі PostgreSQL та управляється через Prisma ORM. Схема даних описана у файлі `schema.prisma` і є єдиним джерелом істини для структури бази даних – на її основі автоматично генерується типізований клієнт Prisma, що використовується у сервісах серверної частини.

Для коректної інтеграції Prisma у модульну архітектуру NestJS реалізовано окремий `PrismaModule` із позначкою `@Global()`. Це означає, що модуль реєструється один раз на рівні кореневого `AppModule` і стає доступним для впровадження у будь-який інший модуль застосунку без повторного імпорту.

PrismaService розширює клас PrismaClient і реалізує інтерфейс OnModuleInit, завдяки чому підключення до бази даних встановлюється автоматично під час ініціалізації модуля через метод \$connect(). Такий підхід гарантує, що з'єднання з базою даних буде готове до першого запиту і правильно управлятиметься впродовж усього життєвого циклу застосунку. Код підключення модулю та сервісу призми до проекту зображено в лістингах 3.4-3.6.

Лістинг 3.4 – Модуль prisma [9]

```
@Global
@Module({
  providers: [PrismaService], exports: [PrismaService],
}) // модуль призми з провайдером і експортом свого сервісу
export class PrismaModule {}
```

кінець лістингу 3.4

Лістинг 3.5 – Сервіс prisma [9]

```
@Injectable()
export class PrismaService extends PrismaClient implements OnModuleInit {
  async onModuleInit() {
    await this.$connect(); // підключення Prisma сервісу
  }
}
```

кінець лістингу 3.5

Лістинг 3.6 – Файл main.ts з доданим модулем prisma до загального додатку [9]

```
@Module({
  imports: [
    ConfigModule.forRoot({isGlobal: true}), AdminModule, // конфіг
    PrismaModule, //Prisma модуль
    AuthModule, FilesModule, ShopModule, NovaPostModule,
    StripeModule, RevalidateModule, SearchModule, // інші модулі
  ],})
export class AppModule {}
```

кінець лістингу 3.6

Схема містить шістнадцять моделей, що охоплюють усі предметні області застосунку.

Модель User є центральною сутністю системи. Вона зберігає ідентифікатор у форматі CUID, електронну пошту (унікальну), хешований пароль, необов'язковий номер телефону, ім'я користувача, роль (admin або user), а також ідентифікатор Google-акаунту для OAuth-авторизації. Для підтримки верифікації email передбачені поля verificationToken та verificationTokenExpires, а для відновлення пароля – resetPasswordToken та resetPasswordTokenExpires. Хешований refresh-токен зберігається безпосередньо у записі користувача у полі hashedRefreshToken, що усуває необхідність окремої таблиці токенів. Модель пов'язана з кошиком, обраними товарами, адресою доставки, замовленнями, відгуками, голосуваннями за відгуки, нотатками та журналом останніх дій.

Модель Collection представляє колекцію одягу. Вона містить назву, унікальний шлях (path) для формування URL, опис, необов'язкове зображення-банер та прапорець видимості isVisible, що дозволяє керувати публікацією колекції без її видалення. Колекція пов'язана з категоріями відношенням «один до багатьох».

Модель Category підпорядкована колекції та містить аналогічний набір полів. Унікальність категорії забезпечується складеним унікальним індексом за парою [collectionId, path], що дозволяє мати однакові шляхи у різних колекціях. Категорія пов'язана з товарами відношенням «один до багатьох».

Модель Product є основною комерційною сутністю. Вона містить назву, URL-шлях, поточну ціну (price) та необов'язкову стару ціну (oldPrice) для відображення знижки, текстовий опис та склад виробу, прапорці доступності (isAvailable) і видимості (isVisible), головне зображення (banner) та масив посилань на додаткові фотографії (images). Зв'язок із атрибутами реалізовано через три проміжні моделі: ProductToColor, ProductToSize та ProductToType – що дозволяє гнучко асоціювати довільну кількість кольорів, розмірів і типів з

кожним товаром. Унікальність товару у межах категорії гарантується складеним індексом [categoryId, path].

Атрибути товарів виділені в окремі довідникові моделі. Модель Color зберігає назву кольору та його HEX-код (обидва поля унікальні). Модель Size зберігає назву розміру. Модель Type зберігає тип виробу (наприклад, «футболка», «худі»). Зв'язок між товаром та атрибутами реалізований через моделі ProductToColor, ProductToSize та ProductToType – кожна з них має складений первинний ключ з пари відповідних ідентифікаторів і використовує каскадне видалення, щоб автоматично прибирати записи при видаленні товару або атрибуту.

Модель Order фіксує повну інформацію про замовлення: персональні дані одержувача (fullName, phoneNumber, email), адресу доставки у вигляді трьох полів – область (area), місто (city), відділення Нової Пошти (postDepartment). Статус замовлення відстежується через перелічувальний тип OrderStatus зі значеннями pending, paid, shipped, delivered, cancelled. Платіжна інформація зберігається у полях paymentStatus, paymentMethod (перелік PaymentMethod: stripe або cod), а також ідентифікаторах Stripe: stripeSessionId, stripePaymentIntentId та stripeCustomerId. Загальна сума замовлення зберігається у полі total у мінімальних одиницях валюти (центах). Замовлення може бути створене без авторизації – поле userId є необов'язковим.

Модель OrderItem є рядком замовлення та зберігає кількість товару, а також конкретні значення атрибутів на момент покупки – колір, розмір та тип – у вигляді рядків. Це забезпечує незмінність даних замовлення навіть при подальшій зміні каталогу. З моделлю OrderItem пов'язана модель Review відношенням «один до одного» – це означає, що покупець може залишити відгук лише на конкретну придбану позицію замовлення, що виключає можливість залишення відгуків без факту покупки.

Модель CartItem зберігає позиції кошика авторизованого користувача на стороні сервера. Унікальність позиції визначається складеним індексом за п'ятьма полями: [userId, productId, size, type, color], що дозволяє зберігати один і той самий товар у різних варіантах атрибутів як окремі позиції.

Модель Favorite реалізує список обраних товарів користувача. Унікальний індекс [userId, productId] виключає дублювання одного товару у списку.

Модель ShippingAddress зберігає збережену адресу доставки користувача, пов'язану відношенням «один до одного». Вона містить деталізовані поля: країна, область, місто, поштовий індекс, вулиця, будинок та квартира.

Модель Review зберігає відгук покупця: текстовий зміст, числову оцінку (від 1 до 5), ім'я та email відправника, масив зображень, прапорець модерації isApproved, лічильники корисності (isHelpful, isNotHelpful), а також поля для відповіді адміністратора – adminReply та adminReplyAt. Голосування за корисність відгуку реалізовано через модель ReviewVote з унікальним обмеженням [userId, reviewId], що запобігає повторному голосуванню.

Модель Todo реалізує особистий список задач адміністратора. Кожна задача має назву та пріоритет – перелічувальний тип TodoPriority зі значеннями low, medium, high.

Модель RecentAction відображає останні дії, виконані адміністратором у системі. Кожен запис містить текстовий опис дії та часову мітку створення.

Візуальний вигляд відношень бази даних зображено на рисунку 3.11.

prx prisma migrate deploy, що забезпечує відтворюваність структури бази даних на будь-якому сервері – від локального середовища розробника до виробничого розгортання на Railway.

3.3 Захист інформаційно-комп'ютерної системи

Забезпечення безпеки реалізовано як комплексна багаторівнева система, що охоплює автентифікацію користувачів, авторизацію доступу до ресурсів, захист даних при зберіганні та передачі, а також захист від типових вебуразливостей. Такий підхід відповідає сучасним стандартам розробки захищених вебдодатків.

Система автентифікації побудована на основі JWT (JSON Web Token) з використанням пари токенів – access-токена та refresh-токена. Така схема є компромісом між безпекою та зручністю використання: короткий час життя access-токена обмежує вікно можливої атаки у разі його перехоплення, тоді як refresh-токен дозволяє прозоро поновлювати сесію без повторного введення облікових даних.

Access-токен має короткий термін дії і передається у кожному авторизованому запиті до API. Він містить у своєму payload ідентифікатор користувача та його роль, що дозволяє серверу авторизувати запит без звернення до бази даних. Після закінчення терміну дії access-токен стає недійсним і клієнт автоматично виконує запит на його оновлення.

Refresh-токен має тривалий термін дії та передається і зберігається виключно в HTTP-only cookie. Такий спосіб зберігання унеможливорює доступ до токена через JavaScript у браузері, що захищає від атак типу XSS (Cross-Site Scripting). На стороні сервера у полі hashedRefreshToken запису користувача зберігається не сам токен, а його хеш. Це означає, що навіть у разі компрометації бази даних злоумисник не зможе використати отримані дані для отримання доступу до облікового запису. При виході користувача із системи значення

hashedRefreshToken обнуляється, що фактично інвалідує всі активні сесії на рівні сервера.

Реалізовано два механізми автентифікації. Перший – локальна автентифікація за логіном та паролем з використанням стратегії Passport.js local. Другий – OAuth 2.0 авторизація через Google з використанням стратегії google. При першому вході через Google автоматично створюється обліковий запис користувача з прив'язаним googleId. Для JWT-токенів реалізовано дві окремі стратегії: jwt-access та jwt-refresh, кожна з яких витягує токен із відповідного cookie та верифікує його підпис і термін дії.

Підтримується повний цикл відновлення доступу до облікового запису. При запиті на скидання пароля генерується одноразовий токен, що зберігається у базі даних разом із часом його дії. Посилання з цим токеном надсилається на електронну пошту користувача. Після використання токен анулюється. Аналогічно реалізовано верифікацію електронної пошти при реєстрації через verificationToken та verificationTokenExpires.

Доставка посилань із токенами здійснюється через сервіс Resend. При верифікації email формується URL виду /auth/verify?token=..., при скиданні пароля – /auth/reset-password?token=.... Посилання вбудовуються в HTML-шаблон листа та надсилаються через resend.emails.send(). Такий підхід гарантує надійну доставлюваність та забезпечує захищений транзит чутливих токенів виключно через особистий email користувача.

Авторизація реалізована через систему Guards у NestJS. За замовчуванням усі маршрути серверної частини є захищеними та вимагають дійсного access-токена. Для публічних ендпоінтів (сторінки каталогу, деталі товарів, форма входу) використовується декоратор @Public(), що позначає маршрут як загальнодоступний і пропускає перевірку токена. Код декоратора що позначає маршрут як загальнодоступний зображено в лістингу 3.7.

Лістинг 3.7 – Код декоратора що позначає маршрут в контроллерах як загальнодоступний [8]

```
import { SetMetadata } from '@nestjs/common';
export const IS_PUBLIC_KEY = 'IS_PUBLIC'; // визначення ключа
export const Public = () => SetMetadata(IS_PUBLIC_KEY, true);
```

кінець лістингу 3.7

Для адміністративних маршрутів застосовується додатковий рівень захисту через декоратор `@Roles()` у поєднанні з `RolesGuard`. Цей `guard` перевіряє роль користувача, витягнуту з `payload access`-токена, та відхиляє запити від користувачів із роллю `user` до адміністративних ендпоінтів. Таким чином, навіть авторизований покупець не може отримати доступ до функцій управління товарами, замовленнями чи користувачами.

Захист адміністративної панелі на клієнтській частині реалізований на рівні серверних компонентів `Next.js`. `Layout`-компонент групи (`admin`) виконує перевірку поточної сесії користувача та його ролі безпосередньо на сервері під час рендерингу. У разі відсутності авторизації або недостатності прав виконується серверний редирект на сторінку входу. Це унеможливорює навіть короткочасне відображення адміністративного інтерфейсу неавторизованим користувачам, на відміну від клієнтських перевірок.

Паролі користувачів зберігаються у базі даних виключно у вигляді хешу, отриманого за допомогою бібліотеки `bcrypt` [19]. Алгоритм `bcrypt` є адаптивним: параметр `cost factor` визначає обчислювальну складність операції хешування, що дозволяє підвищувати стійкість до брутфорс-атак зі зростанням обчислювальних потужностей. Навіть у разі повного витоку бази даних злоумисник не зможе відновити оригінальні паролі в прийнятний час завдяки навмисно повільному алгоритму хешування та унікальній солі для кожного запису. При авторизації пароль, введений користувачем, порівнюється з хешем через функцію `bcrypt.compare()`, яка ніколи не розкриває оригінальне значення. Код хешування пароллю зображено в лістингу 3.8.

Лістинг 3.8 – Код хешування паролю за допомогою бібліотеки bcrypt [19]

```
const hashedPassword = await bcrypt.hash(createUserDto.password, 10);
```

кінець лістингу 3.8

Обробка платіжних даних повністю делегована платформі Stripe, що є свідомим архітектурним рішенням з точки зору безпеки. Жодні дані платіжних карток не проходять через сервери застосунку та не зберігаються у базі даних. Клієнт перенаправляється на захищену сторінку Stripe Checkout, яка обслуговується інфраструктурою Stripe та сертифікована за стандартом PCI DSS рівня 1.

Для підтвердження успішних платежів використовується механізм вебхуків. Після завершення транзакції Stripe надсилає подію на ендпоінт `/stripe/webhook` серверної частини. Кожна така подія підписана секретним ключем вебхука (`STRIPE_SECRET_WEBHOOK_KEY`), і сервер верифікує підпис перед обробкою події за допомогою методу `stripe.webhooks.constructEvent()`. Це унеможлиблює підроблення платіжних подій зловмисниками, оскільки без знання секретного ключа коректний підпис згенерувати неможливо. Код сервісу stripe зображено в додатку Г.

Усі дані, що надходять до серверної частини від клієнта, проходять обов'язкову валідацію на рівні DTO-класів (Data Transfer Objects). Для декларативного опису правил валідації використовуються декоратори бібліотеки `class-validator` [20]: `@IsString()`, `@IsEmail()`, `@IsNotEmpty()`, `@MinLength()`, `@IsEnum()` тощо. Трансформація вхідних даних до потрібних типів здійснюється через `class-transformer`. Глобально підключений `ValidationPipe` у NestJS автоматично перевіряє відповідність вхідних об'єктів DTO перед передачею у контролер. У разі невідповідності запит відхиляється із HTTP-статусом 400 Bad Request та детальним переліком порушень валідації, не досягаючи рівня бізнес-логіки. Приклад `dto` зображено в додатку Д.

Міжсайтові запити контролюються через налаштування CORS (Cross-Origin Resource Sharing) у NestJS. Перелік дозволених джерел запитів обмежено

доменом клієнтської частини, що вказується через змінну середовища `FRONTEND_URL`. Таким чином, API-запити від сторонніх доменів автоматично відхиляються на рівні протоколу браузера. Підключення CORS відбувається в файлі `main.ts` на сервері. Код підключення зображено в лістингу 3.9.

Лістинг 3.9 – Код підключення CORS політики з посиланнями та доступними методами та передачею `credential` [8]

```
app.enableCors({
  origin: [
    frontendUrl, // посилання на фронтенд частину
    https://mindset-client-production.up.railway.app // запасне
  ],
  methods: "GET,HEAD,PUT,PATCH,POST,DELETE", // методи доступу
  credentials: true, // дозвіл доступу до куки
});
```

кінець лістингу 3.9

Усі чутливі параметри конфігурації – ключі Stripe, секрети JWT, облікові дані бази даних, ключ Supabase, API-ключ Nova Post, налаштування поштового сервісу – зберігаються виключно у змінних середовища та ніколи не потрапляють до кодової бази репозиторію. Для кожного середовища розгортання (локальне та виробниче) підтримуються окремі набори змінних. На платформі Railway змінні середовища налаштовуються безпосередньо у панелі керування проектом та автоматично передаються у контейнер при розгортанні. Зображення розгортання та налаштування проекту на Railway відображено на рисунку 3.12.

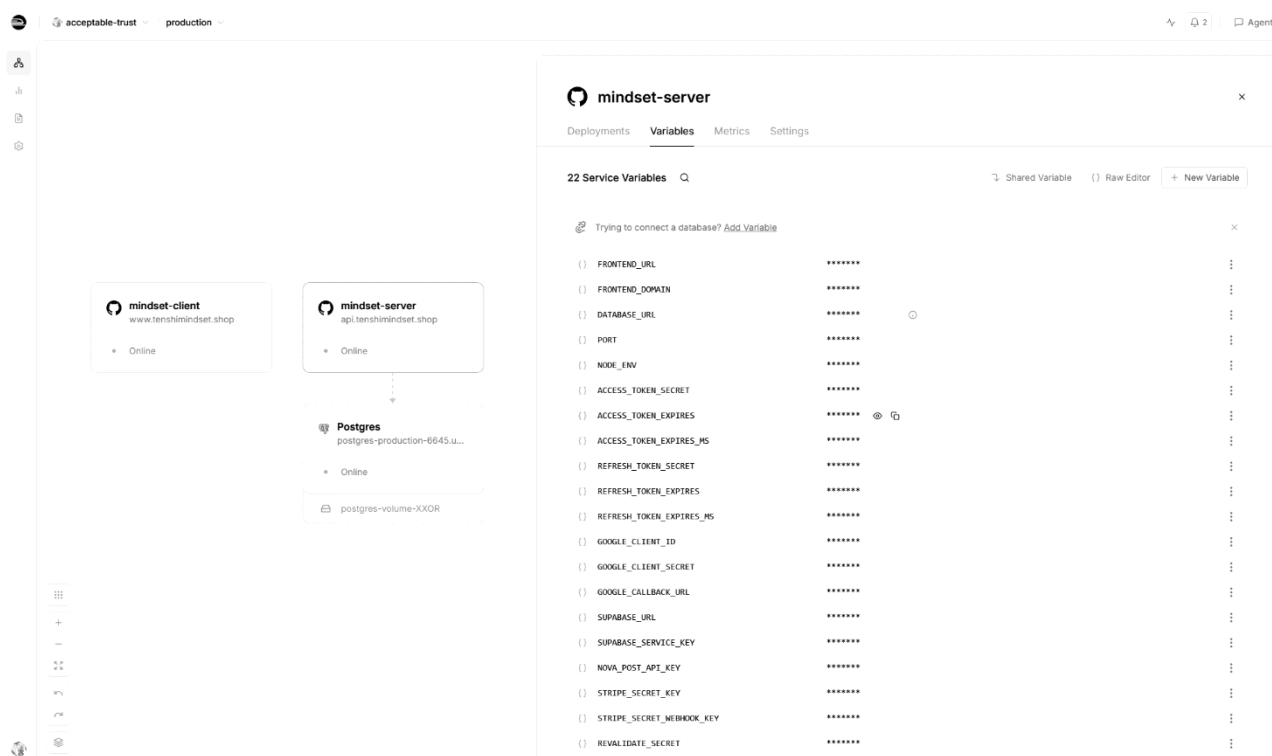


Рисунок 3.12 – Дешбоард проекту з змінними середовища розгорнутого на Railway хостингу

3.4 Тестування та налагодження інформаційно-комп'ютерної системи

Тестування розробленого вебзастосунку проводилося систематично на кількох рівнях із метою перевірки коректності функціонування кожного компонента системи, відповідності поставленим функціональним вимогам та стійкості до граничних і некоректних вхідних даних. Процес тестування охоплював клієнтську частину, серверне API, платіжну систему, інтеграцію із зовнішніми сервісами та механізми безпеки.

Функціональне тестування інтерфейсу виконувалося методом ручного проходження сценаріїв використання у браузері Google Chrome. Для кожного функціонального модуля було визначено перелік тестових сценаріїв, що охоплюють як типові, так і граничні випадки.

Тестування модуля автентифікації включало перевірку реєстрації нового користувача з валідними даними та отримання листа для підтвердження email; спробу реєстрації з уже існуючою адресою електронної пошти та очікувану

появу повідомлення про помилку; вхід з коректними обліковими даними та подальший редирект на головну сторінку; вхід із невірним паролем та відображення відповідного повідомлення; авторизацію через Google OAuth з перевіркою коректного створення або входу в існуючий обліковий запис; а також перевірку повного циклу відновлення пароля – від запиту до встановлення нового.

Тестування каталогу товарів передбачало перевірку відображення товарів із застосуванням різних комбінацій фільтрів: за колекцією, категорією, кольором, розміром та типом. Перевірялася коректність формування URL search params при виборі фільтрів та збереження стану фільтрів при поверненні на сторінку через кнопку «назад». Окремо перевірялася функція пошуку по каталогу через модальне вікно – відображення результатів у реальному часі при введенні запиту та відсутність результатів при введенні неіснуючого товару.

Тестування сторінки товару включало перевірку відображення галереї зображень із каруселлю, коректного вибору атрибутів (колір, розмір, тип), відображення середнього рейтингу та переліку схвалених відгуків. Перевірялася неможливість додавання товару до кошика без вибору всіх обов'язкових атрибутів.

Тестування кошика охоплювало додавання товарів з різними атрибутами як одним записом чи окремими позиціями, зміну кількості товарів, видалення позицій, а також збереження стану кошика між сесіями браузера завдяки localStorage. Для авторизованих користувачів додатково перевірялася синхронізація кошика між пристроями через серверне зберігання.

Тестування особистого кабінету включало перевірку відображення історії замовлень із деталями кожного, можливість зміни основних даних профілю, зміну та встановлення пароля для OAuth-користувачів, управління збереженою адресою доставки, а також видалення облікового запису із підтвердженням.

Тестування адміністративної панелі передбачало перевірку повного циклу CRUD-операцій для кожної сутності: колекцій, категорій, товарів, атрибутів (кольорів, розмірів, типів), замовлень та відгуків. Для товарів окремо

перевірялося завантаження зображень із відображенням попереднього перегляду та їх коректне збереження у Supabase Storage. Тестувалося оновлення статусу замовлення та відображення змін у особистому кабінеті покупця. Перевірялася модерація відгуків – схвалення, відхилення та відповідь адміністратора. Тестування розділу статистики включало перевірку відображення діаграм продажів, реєстрацій користувачів, розподілу замовлень за категоріями та колекціями.

Тестування API-ендпоінтів серверної частини виконувалося за допомогою інструменту Postman. Для кожного ендпоінта сформовано набір тестових запитів, що охоплюють різні сценарії. Візуальний вигляд та запит в Postman показано на рисунку 3.13.

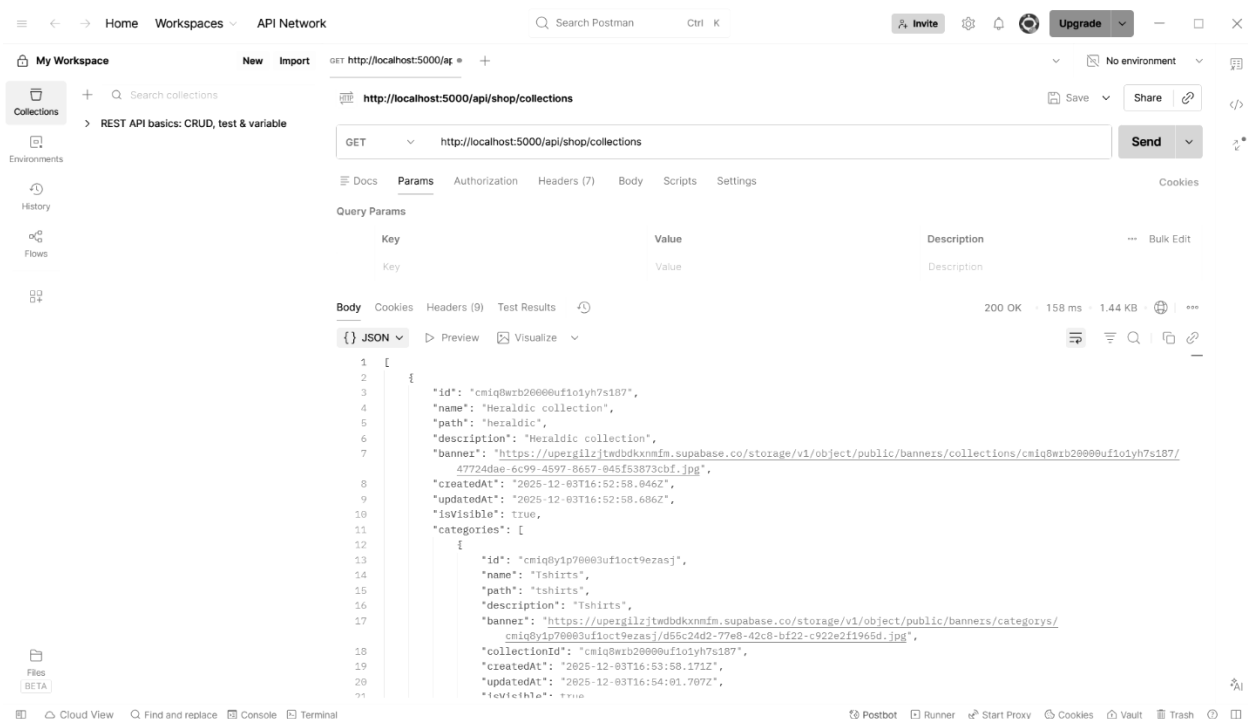


Рисунок 3.13 – Візуальний вигляд та запит в додатку Postman

Перевірялася коректність HTTP-статусів відповідей: 200 або 201 для успішних операцій, 400 для запитів із невалідними даними, 401 для неавторизованих запитів до захищених маршрутів, 403 для запитів від користувачів із недостатніми правами доступу, 404 для запитів до неіснуючих

ресурсів. Для кожного статусу перевірялася структура тіла відповіді та наявність інформативного повідомлення про помилку.

Окремо тестувалася робота системи валідації: надсилалися запити з відсутніми обов'язковими полями, полями невірної типу, значеннями, що не відповідають форматним обмеженням (некоректний email, занадто короткий пароль тощо). У всіх випадках сервер коректно повертав статус 400 з переліком конкретних порушень валідації.

Тестування механізмів авторизації включало спроби звернення до адміністративних ендпоінтів без токена, з протермінованим токеном та з токеном користувача з роллю user. В усіх випадках сервер коректно відхиляв запити. Перевірялася також коректна ротація токенів: після оновлення refresh-токена старий ставав недійсним і не міг бути використаний повторно.

Тестування ендпоінтів фільтрації передбачало перевірку коректності результатів при різних комбінаціях параметрів запиту, а також поведінку при передачі некоректних значень параметрів (від'ємні числа, рядки замість чисел).

Тестування інтеграції зі Stripe проводилося у тестовому режимі платформи, що надає повноцінне середовище для перевірки платіжних сценаріїв без реальних транзакцій. Stripe надає набір тестових номерів карток для відтворення різних сценаріїв.

Перевірявся сценарій успішної оплати: після підтвердження замовлення та оплати тестовою картою 4242 4242 4242 4242 вебхук коректно отримувала подія `checkout.session.completed`, статус замовлення у базі даних змінювався на `paid`, а користувач перенаправлявся на сторінку `/order/success`. Перевірявся сценарій відхилення картки: при використанні тестової картки 4000 0000 0000 0002 Stripe повертав помилку, користувач залишався на сторінці оплати Stripe з відповідним повідомленням, а замовлення зберігало статус `pending`. Окремо тестувалася коректність верифікації підпису вебхука – запит із некоректним підписом відхилявся сервером.

Тестування інтеграції з API Нової Пошти включало перевірку коректності каскадного завантаження даних у формі оформлення замовлення. Перевірялося

відображення повного переліку областей при відкритті форми, коректне завантаження міст після вибору конкретної області, відображення актуального переліку відділень після вибору міста. Тестувалася поведінка при скиданні вибору – зміна вибраної області коректно очищала вибране місто та відділення і завантажувала нові дані. Перевірялася коректна обробка помилок при недоступності API – відображення повідомлення про помилку та можливість повторної спроби.

Тестування модуля email включало: отримання листа верифікації після реєстрації з коректним посиланням; отримання вітального листа після підтвердження email; отримання листа підтвердження замовлення після успішної оплати через Stripe-вебхук – лист містив повне ім'я одержувача, дані доставки (область, місто, відділення Нової Пошти), спосіб оплати, перелік товарів із кількістю та ціною (рис. 3.10); отримання листа скидання пароля з коректним токеном та його анулювання після використання.

Перевірялася коректність формування метаданих сторінок – тегів title, description та Open Graph для сторінок товарів, категорій та колекцій. Тестувалася доступність файлів sitemap.xml та robots.txt. За допомогою інструменту Lighthouse у Chrome DevTools проводився аналіз показників продуктивності, зокрема метрик Core Web Vitals: Largest Contentful Paint, Cumulative Layout Shift та Interaction to Next Paint. Перевірялася коректна робота механізму ISR – після внесення змін через адміністративну панель та виклику ревалідації оновлені дані відображалися на публічних сторінках без перерозгортання застосунку.

Налагодження клієнтської частини виконувалося із використанням вбудованих інструментів Chrome DevTools. Вкладка Network використовувалася для аналізу мережових запитів, перевірки заголовків, тіл запитів та відповідей, а також часу виконання запитів. Вкладка Application використовувалася для перевірки стану localStorage, зокрема коректності збереження та відновлення кошика та списку обраних товарів. Для налагодження стану Zustand-сторів використовувалося розширення Redux DevTools, що підтримує Zustand через

відповідний middleware. Інтерфейс та функціонал розширення зображено на рисунку 3.14.

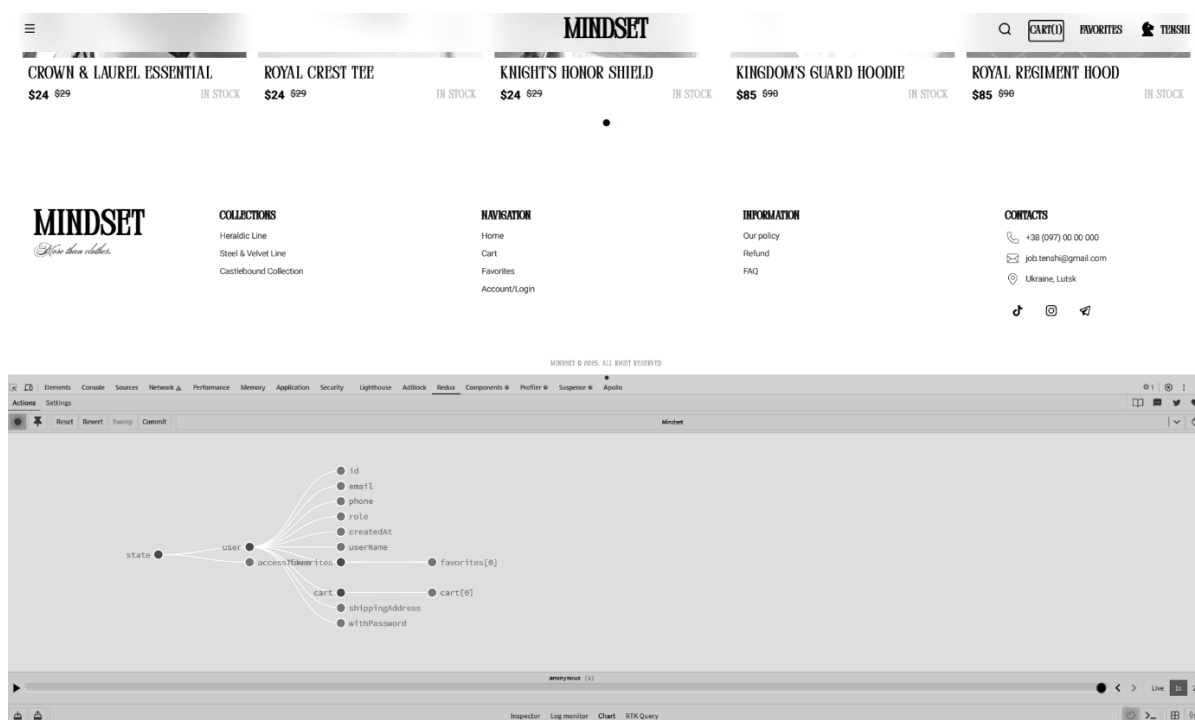


Рисунок 3.14 – Інтерфейс та функціонал розширення Redux Devtools

Налагодження серверної частини здійснювалося через вбудовану систему логування NestJS, що виводить детальну інформацію про отримані запити, виконані SQL-запити Prisma та виникаючі виключення.

Виявлені у ході тестування дефекти фіксувалися та усувалися в ітеративному режимі: після виправлення кожної помилки відповідний сценарій проходив повторне тестування для підтвердження коректності виправлення та відсутності регресій у суміжній функціональності.

За результатами проведеного тестування встановлено, що розроблений вебзастосунок відповідає визначеним функціональним та нефункціональним вимогам. Всі основні сценарії використання функціонують коректно, механізми захисту надійно відхиляють неавторизовані та некоректні запити, інтеграції з платіжною системою та сервісом доставки працюють відповідно до очікуваної поведінки. Застосунок розгорнутий у виробничому середовищі та доступний за адресою www.tenshimindset.shop.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було розроблено повнофункціональний вебзастосунок для інтернет-магазину бренду одягу «Mindset» із використанням фреймворків Next.js та NestJS. У процесі роботи виконано всі поставлені завдання.

Досліджено сучасні підходи до побудови вебзастосунків у сфері електронної комерції. Проведено порівняльний аналіз готових платформ – Shopify та WooCommerce – та встановлено їхню обмеженість щодо кастомізації, інтеграції з українськими сервісами та масштабованості, що обґрунтувало доцільність розробки власного рішення.

Визначено функціональні та нефункціональні вимоги до розроблюваної системи. Сформульовано вимоги окремо для покупця та адміністратора, а також встановлено обмеження щодо продуктивності, безпеки, масштабованості та зручності супроводу, які слугували основою для прийняття архітектурних рішень.

Обґрунтовано вибір технологічного стеку та архітектурних рішень. Обрано трирівневу архітектуру з розподілом на клієнтську частину (Next.js 16), серверну (NestJS) та рівень даних (PostgreSQL), що забезпечує чіткий розподіл відповідальності та можливість горизонтального масштабування.

Спроектовано структуру бази даних та взаємодію між компонентами системи. Розроблено реляційну схему PostgreSQL з управлінням через Prisma ORM, що охоплює сутності товарів, замовлень, користувачів, відгуків та платежів. Взаємодію між компонентами описано через REST API з документацією у Swagger.

Реалізовано клієнтську частину застосунку з використанням Next.js та TypeScript. Застосовано компонентну архітектуру з розподілом на серверні та клієнтські компоненти відповідно до можливостей App Router, підтримку SSR та ISR для оптимізації продуктивності, а також Zustand і React Query для управління станом.

Розроблено серверну частину на базі NestJS із підтримкою REST API. Реалізовано модульну структуру з окремими модулями для автентифікації, товарів, замовлень, користувачів, платежів та доставки. Кожен модуль містить контролер, сервіс та DTO-класи з валідацією через class-validator.

Інтегровано систему платежів Stripe та Nova Post API. Реалізовано повний платіжний цикл через Stripe з обробкою webhook-подій для підтвердження оплати, а також інтеграцію з Nova Post API для отримання актуального списку відділень та поштоматів під час оформлення замовлення.

Забезпечено механізми автентифікації, авторизації та захисту даних. Реалізовано JWT-автентифікацію з refresh-токенами, авторизацію через Google OAuth, хешування паролів, розмежування доступу за роллю користувача та захист адміністративних маршрутів.

Проведено тестування та налагодження розробленого програмного продукту. Виконано функціональне тестування основних сценаріїв використання – оформлення замовлення, обробка платежів, адміністрування – та усунуто виявлені недоліки. Застосунок розгорнуто у продуктивному середовищі та доступний за адресою tenshimindset.shop.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Next.js by Vercel. The React Framework. Next.js by Vercel - The React Framework. URL: <https://nextjs.org/> (дата звернення: 17.02.2026).
2. JavaScript With Syntax For Types. TypeScript: JavaScript With Syntax For Types. URL: <https://www.typescriptlang.org/> (дата звернення: 17.02.2026).
3. Tailwind CSS. Rapidly build modern websites without ever leaving your HTML. URL: <https://tailwindcss.com/> (дата звернення: 17.02.2026).
4. The Foundation for your Design System - shadcn/ui. The Foundation for your Design System. shadcn/ui. URL: <https://ui.shadcn.com/> (дата звернення: 17.02.2026).
5. Zustand. Zustand. URL: <https://zustand-demo.pmnd.rs/> (дата звернення: 03.03.2026).
6. TanStack Query. TanStack. The open-source application stack for the web. URL: <https://tanstack.com/query/latest> (дата звернення: 03.03.2026).
7. Recharts. URL: <https://recharts.github.io/> (дата звернення: 03.03.2026).
8. NestJS. A progressive Node.js framework. NestJS, a progressive Node.js framework. URL: <https://nestjs.com/> (дата звернення: 22.03.2026).
9. Prisma. Serverless Postgres, Type-Safe ORM, and Database Tools. Prisma. URL: <https://www.prisma.io/> (дата звернення: 22.03.2026).
10. PostgreSQL. PostgreSQL. URL: <https://www.postgresql.org/> (дата звернення: 07.04.2026).
11. Documentation. Stripe-Documentation. URL: <https://docs.stripe.com/> (дата звернення: 10.04.2026).
12. Nova Post API. URL: <https://api.novapost.com/developers/index.html> (дата звернення: 21.04.2026).
13. Supabase Docs. Supabase. The Postgres Development Platform. URL: <https://supabase.com/docs> (дата звернення: 24.04.2026).
14. Introduction. Resend. Email for developers. URL: <https://resend.com/docs/introduction> (дата звернення: 23.04.2026).

15. Railway Docs. Railway Docs. URL: <https://docs.railway.com/> (дата звернення: 03.05.2026).

16. Welcome. Feature-Sliced Design. Welcome. Feature-Sliced Design. URL: <https://feature-sliced.design/> (дата звернення: 17.03.2026).

17. React Hook Form - performant, flexible and extensible form library. React Hook Form - performant, flexible and extensible form library. URL: <https://react-hook-form.com/> (дата звернення: 15.03.2026).

18. Passport.js. Passport.js. URL: <https://www.passportjs.org/> (дата звернення: 10.04.2026).

19. GitHub. Kelektiv/node.bcrypt.js: bcrypt for NodeJs. GitHub. URL: <https://github.com/kelektiv/node.bcrypt.js#readme> (дата звернення: 11.04.2026).

20. GitHub. Typestack/class-validator: Decorator-based property validation for classes. GitHub. URL: <https://github.com/typestack/class-validator#readme> (дата звернення: 22.03.2026).

ДОДАТКИ


```

        />
        <InfoField
            label="Number of products"
            value={products?.length}
        />
    </div>
    <InfoModalBanner image={banner || ""} />
</FormFillingWrapper>
<FormButtonsWrapper>
    <MonoButtonUnderlined onClick={onClose}>
        Close
    </MonoButtonUnderlined>
</FormButtonsWrapper>
</ModalWrapper>
);
return createPortal(modalContent, document.body);
}

```

ДОДАТОК Б

Код «стратегій» JWT для Passport.js

```

@Injectable()

export class JwtAccessStrategy extends PassportStrategy(Strategy, "jwt-
access") {
  constructor(
    private readonly configService: ConfigService,
    private readonly authService: AuthService
  ) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      secretOrKey:
configService.get<string>("ACCESS_TOKEN_SECRET")!,
    });
  }

  async validate(payload: { sub: string }) {
    const user = await this.authService.validateJwtUser(payload.sub);
    if (!user) {
      throw new UnauthorizedException("User not found");
    }
    return user;
  }
}

@Injectable()

export class JwtRefreshStrategy extends PassportStrategy(Strategy, "jwt-
refresh") {
  constructor(
    private configService: ConfigService,

```

```

        private authService: AuthService
    ) {
        super({
            jwtFromRequest: extractRefreshTokenFromCookie,
            ignoreExpiration: false,
            secretOrKey:
configService.get<string>("REFRESH_TOKEN_SECRET")!,
            passReqToCallback: true,
        });
    }

    async validate(req: Request, payload: { sub: string }) {
        const refreshToken = String(req.cookies?.refreshToken);

        if (!refreshToken) {
            throw new UnauthorizedException("Refresh token is missing");
        }

        const user = await
this.authService.validateRefreshToken(payload.sub, refreshToken);

        return { id: user.id, refreshToken }; // Return userId and the
token
    }
}

```

ДОДАТОК В

Код email.service.ts з запитами на створення та відправку повідомлень на пошту користувачам

```
@Injectable()
export class EmailService {
  private readonly resend: Resend;
  private readonly fromEmail: string;
  constructor(private readonly configService: ConfigService) {
    const apiKey = this.configService.get<string>("RESEND_API_KEY");
    if (!apiKey) {
      throw new Error("RESEND_API_KEY is missing in
configuration");
    }
    this.resend = new Resend(apiKey);
    this.fromEmail = this.configService.get<string>("MAIL_FROM") ||
"onboarding@resend.dev";
  }
  private handleEmailError(error: unknown, context: string): void {
    console.error(`Error in ${context}:`, error);
  }
  async sendRegistrationEmail(email: string) {
    const msg = {to: email, from: this.fromEmail, subject: "Account
successfully created", html: `
      <div style="font-family: Arial, sans-serif; line-height:
1.6;">
        <p>Your account at <strong>Mindset</strong> has been
successfully created.</p>
        <p>You can now log in using your email and password.</p>
        <p style="margin-top: 20px; color: #555;">
```

If you did not register this account, please ignore this email.

```

        </p>
    </div>
    `,
};
try {
    await this.resend.emails.send(msg);
    console.log(`Registration email sent to ${email}`);
} catch (error) {
    this.handleEmailError(error, "sendRegistrationEmail");
    throw error;
}
}

async sendOrderCreatedEmail(order: Order & { items: (OrderItem & {
product: Product })[] }) {
    const itemsHtml = order.items
        .map((item, index) => `
            ТЕКСТ ПОВІДОМЛЕННЯ
        `,
    );
    try {
        await this.resend.emails.send(msg);
        console.log(`Order confirmation email sent to
${order.email}`);
    } catch (error) {
        this.handleEmailError(error, "sendOrderCreatedEmail");
        throw error;
    }
}
}
}

```

ДОДАТОК Г

Код сервісу Stripe для оплати користувачів та обробки вебхуку

```
@Injectable()
export class StripeService {
  private stripe: Stripe;
  constructor(
    private readonly prisma: PrismaService,
    private readonly configService: ConfigService,
  ) {
    this.stripe = new Stripe(configService.get("STRIPE_SECRET_KEY")!,
    { apiVersion: "2025-10-29.clover" });
    async createCheckoutSession(body: { orderId: string }) {
      const order = await this.prisma.order.findUnique({
        where: { id: body.orderId },
        include: { items: { include: { product: true } } },
      });
      if (!order) throw new NotFoundException("Order not found");
      const lineItems = order.items.map((item) => ({
        price_data: {
          currency: "usd",
          product_data: {
            name: item.product.name,
          },
          unit_amount: item.product.price * 100,
        },
        quantity: item.quantity,
      }));
      const session = await this.stripe.checkout.sessions.create({
        payment_method_types: ["card"],
        mode: "payment",
        line_items: lineItems,
        success_url:
        `${process.env.FRONTEND_URL}/order/success?session_id={CHECKOUT_SESSION_ID}`,
        cancel_url: `${process.env.FRONTEND_URL}/order/cancel`,
        metadata: { orderId: order.id },
      });
      await this.prisma.order.update({
        where: { id: order.id },
        data: { stripeSessionId: session.id },
      });
      return { url: session.url };
    }
  }
}
```

```

    }
    async handleWebhook(req: Request & { body: Buffer }, signature: string)
    {
        const webhookSecret: string =
        this.configService.get("STRIPE_SECRET_WEBHOOK_KEY")!;
        let event: Stripe.Event;
        try {
            event = this.stripe.webhooks.constructEvent(req.body,
signature, webhookSecret);
        } catch (err: unknown) {
            const error = err as Error;
            console.error("Webhook signature failed:", error.message);
            return;
        }
        if (event.type === "checkout.session.completed") {
            const session = event.data.object;
            await this.updateOrderAfterPayment(session);
        }
    }

    private async updateOrderAfterPayment(session:
Stripe.Checkout.Session) {
        const orderId = session.metadata?.orderId;
        const order = await this.prisma.order.findUnique({
            where: { id: orderId },
        });
        if (!order) throw new NotFoundException("Order not found");
        try {
            await this.prisma.order.update({
                where: { id: orderId },
                data: {
                    status: OrderStatus.paid,
                    paymentStatus: PaymentStatus.paid,
                    stripePaymentIntentId:
                        typeof session.payment_intent === "string"
                            ? session.payment_intent
                            : (session.payment_intent?.id ?? null),
                    stripeCustomerId:
                        typeof session.customer === "string"
                            ? session.customer
                            : (session.customer?.id ?? null),
                },
            });
        } catch (err) {console.error("Failed to update order:", err)}
    }
}

```

ДОДАТОК Д

Код DTO для створення замовлення

```
export class CreateOrderItemDto {  
    @IsString()  
    @NotEmpty()  
    productId: string;  
  
    @IsNumber()  
    @NotEmpty()  
    quantity: number;  
  
    @IsString()  
    color: string;  
  
    @IsString()  
    size: string;  
  
    @IsString()  
    type: string; }  

```

```
export class CreateOrderDto {  
    @IsString()  
    @NotEmpty()  
    fullName: string;  
  
    @IsString()  
    @NotEmpty()  
    phoneNumber: string;  
  
    @IsString()  
    @NotEmpty()  
    email: string;  
  
    @IsString()  
    @NotEmpty()  
    area: string;  
  
    @IsString()  
    @NotEmpty()  
    city: string;  
  
    @IsString()  

```

```
@IsNotEmpty()  
postDepartment: string;  
  
@IsString()  
@IsNotEmpty()  
paymentMethod: PaymentMethod;  
  
@IsNumber()  
@IsOptional()  
@Type(() => Number)  
total?: number;  
  
@IsArray()  
@ValidateNested({ each: true })  
@Type(() => CreateOrderItemDto)  
items: CreateOrderItemDto[];  
  
@IsString()  
@IsOptional()  
userId?: string;  
  
@IsEnum(OrderStatus)  
@IsOptional()  
status?: OrderStatus;  
}
```