МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра інтелектуальних інформаційних систем

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри інтелектуальних інформаційних систем

_____Юрій КОНДРАТЕНКО

«____»_____ 2025 р.

КВАЛІФІКАЦІЙНА РОБОТА

НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВРА

ІНТЕГРОВАНА ПЛАТФОРМА ЕЛЕКТРОННОЇ КОМЕРЦІЇ

Спеціальність 122 Комп'ютерні науки

Освітня програма «Комп'ютерні науки»

Здобувач _____Костянтин СЕРГІЄВ

«___»_____ 20__ р.

Керівник ст. викладач кафедри ІПЗ _____Юрій РЕШЕТНІК

«___»_____ 20__ р.

Миколаїв – 2025

Завдання на виконання кваліфікаційної роботи

Чорноморський національний університет імені Петра Могили
(повне найменування закладу вищої освіти)

Факультет	Комп'ютерних наук
Кафедра	Інтелектуальних інформаційних систем
Рівень вищої освіти	Перший (бакалаврський)
Освітній ступень	Бакалавр
Спеціальність	122 Комп'ютерні науки
Освітня програма	Комп'ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедри інтелектуальних інформаційних систем
_____ Юрій КОНДРАТЕНКО
«____» _____ 20___ р.

ЗАВДАННЯ
на кваліфікаційну роботу здобувача

Сергієв Костянтин Олександрович
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: «Інтегрована платформа електронної комерції».

Керівник роботи: Решетнік Юрій Володимирович ст. викладач кафедри ІПЗ.

(прізвище, ім'я, по батькові, посада, науковий ступінь, вчене звання)

Затверджена наказом ЧНУ ім. Петра Могили від «18» листопада 2025 р. № 321.

2. Строк представлення кваліфікаційної роботи «___» червня 2025 р.

3. Очікуваний результат роботи:

У результаті роботи планується створити інтегровану платформу електронної комерції, яка дозволить користувачам зручно здійснювати онлайн-покупки, керувати своїм кошиком, отримувати персоналізовані знижки, переглядати історію замовлень та оплачувати товари через платіжні сервіси.

4. Перелік питань, що підлягають розробці:

- аналіз сфери електронної комерції;
- формування вимог до платформи електронної комерції;
- реалізація платформи на основі сучасного стеку (React, Node.js, MongoDB);
- забезпечення інтеграції всіх компонентів.

5. Перелік графічних матеріалів: презентація.

Керівник роботи _____ Юрій РЕШЕТНІК
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Здобувач _____ Костянтин СЕРГІЄВ
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Дата видачі завдання «20» грудня 2024 р.

КАЛЕНДАРНИЙ ПЛАН
кваліфікаційної роботи

Тема: Інтегрована платформа електронної комерції.

№	Найменування роботи	Початок	Закінчення	Примітки
1	Отримання завдання на виконання КР	18.12.2024	22.12.2024	виконано
2	Аналіз предметної області та постановка задачі	23.12.2024	30.01.2025	виконано
3	Огляд літературних джерел за темою кваліфікаційної роботи, зокрема огляд стану електронної комерції та аналогічних платформ	31.01.2025	01.03.2025	виконано
4	Вивчення та вибір методів для реалізації платформи	02.03.2025	01.04.2025	виконано
5	Використання отриманих технологій, розробка діаграм, аналіз та тестування платформи	02.04.2025	15.05.2025	виконано
6	Перший попередній захист КР на засіданні комісії кафедри	16.05.2025	16.05.2025	виконано
7	Корегування роботи за результатами попереднього захисту	17.05.2025	29.05.2025	виконано
8	Другий попередній захист КР на засіданні комісії кафедри	30.05.2025	30.05.2025	виконано
9	Доробка та остаточне оформлення КР	31.05.2025	10.06.2025	виконано
10	Подання КР, її електронної копії та інших документів (відгуку, рецензії) до захисту	11.06.2025	17.06.2025	

Керівник роботи _____ Юрій РЕШЕТНІК
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Здобувач _____ Костянтин Сергієв
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Дата складання календарного плану
«30» січня 2025 р.

АНОТАЦІЯ

до кваліфікаційної роботи
здобувача групи 402 ЧНУ ім. Петра Могили

Сергієва Костянтина Олександровича

на тему: **«ІНТЕГРОВАНА ПЛАТФОРМА ЕЛЕКТРОННОЇ КОМЕРЦІЇ»**

Актуальність даної роботи полягає у необхідності впровадження сучасних цифрових рішень для автоматизації процесів електронної комерції, що забезпечує зручну взаємодію між продавцем і покупцем, підвищує ефективність обслуговування клієнтів та сприяє розвитку малого і середнього бізнесу в онлайн-середовищі.

Об'єктом роботи є процес розробки інтегрованої платформи електронної комерції

Предметом роботи є методи та засоби розробки інтегрованих платформ електронної комерції з використанням сучасних підходів до реалізації.

Метою роботи є розробка методів та програмних засобів для підвищення ефективності інтегрованих платформ електронної комерції.

У першому розділі подано аналітичний огляд предметної області, аналіз ринку електронної комерції та постановку задачі. У другому розділі розглянуто архітектурне планування клієнтської частини, реалізацію ролей користувачів, макети інтерфейсу та проєктування бази даних MongoDB. У третьому розділі представлено модельну частину системи – діаграми варіантів використання, послідовності, активностей, класів і компонентів, що відображають внутрішню логіку роботи платформи. Четвертий розділ містить повну реалізацію проєкту: опис клієнтської частини (React), серверної логіки (Node.js + Express), схеми взаємодії з базою даних, функціонал авторизації та кабінету користувача, адміністрування, тестування платформи.

Розроблена платформа ShopManager є повнофункціональним рішенням електронної комерції з підтримкою ролей, обробки замовлень, інтеграцією з платіжними системами та доставкою. У процесі тестування було підтверджено

стабільність роботи та зручність інтерфейсу. Система рекомендована до подальшого розгортання або адаптації для реальних потреб малого онлайн-бізнесу.

Загальний обсяг роботи – 109 сторінок. Кваліфікаційна робота містить 1 додаток, 59 рисунків, 1 таблицю та 50 джерел посилань.

Ключові слова: платформа, електронна комерція, React, Node.js, MongoDB, ShopManager.

ABSTRACT

to the qualification work by the student of the group 402
of Petro Mohyla Black Sea National University

Serhiiev Kostiantyn

“ INTEGRATED E-COMMERCE PLATFORM”

The relevance of this work lies in the need to implement modern digital solutions for automating e-commerce processes, ensuring convenient interaction between sellers and buyers, increasing customer service efficiency, and supporting the development of small and medium-sized businesses in the online environment.

The object of the work is the process of developing an integrated e-commerce platform.

The subject of the work includes methods and tools for designing integrated e-commerce platforms using modern implementation approaches.

The purpose of the work is to develop methods and software tools to improve the efficiency of integrated e-commerce platforms.

The first chapter provides an analytical review of the subject area, an analysis of the e-commerce market, and a problem statement. The second chapter covers the architectural planning of the client side, implementation of user roles, interface mockups, and the design of the MongoDB database. The third chapter presents the system modeling part — use case diagrams, sequence diagrams, activity diagrams, class and component diagrams that reflect the internal logic of the platform. The fourth chapter includes the complete implementation of the project: a description of the client side (React), server logic (Node.js + Express), database interaction schemes, authentication and user account functionality, administration, and platform testing.

The developed ShopManager platform is a fully functional e-commerce solution with support for user roles, order processing, integration with payment systems, and delivery services. During testing, the system's stability and user interface convenience were confirmed. The platform is recommended for further deployment or adaptation to meet the real needs of

small online businesses.

Total length of the thesis – 109 pages. The qualification work contains 1 appendix, 59 figures, 1 table, and 50 references.

Keywords: platform, e-commerce, React, Node.js, MongoDB, ShopManager.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

БД – база даних

ІІС – інтелектуальні інформаційні системи

КР – кваліфікаційна робота

CSS – Cascading Style Sheets

API – Application Programming Interface

ВСТУП

Сучасна електронна комерція є однією з найбільш динамічно розвинутих галузей цифрової економіки, яка безпосередньо впливає на формування глобального ринку товарів і послуг. Попри значні досягнення в цій сфері, зокрема створення масштабованих платформ Amazon, Shopify, Magento, існує чимало викликів, пов’язаних з інтеграцією різних сервісів, обробкою великих обсягів даних, захистом конфіденційної інформації, а також підтримкою стабільної роботи платформи у пікові навантаження.

Актуальність обраної теми зумовлена стрімким розвитком цифрової торгівлі, зростаючими сподіваннями клієнтів щодо швидкості обслуговування та безперервної роботи сервісу. Сучасність потребує рішень, які зможуть реагувати на зміни попиту, інтегруватися з різними каналами продажу та забезпечуватимуть високий рівень безпеки.

Об’єктом роботи є процес розробки інтегрованої платформи електронної комерції

Предмет роботи є методи та засоби розробки інтегрованих платформ електронної комерції з використанням сучасних підходів до реалізації.

Метою роботи є розробка методів та програмних засобів для підвищення ефективності інтегрованих платформ електронної комерції.

Завдання полягає у реалізації комплексу дій, необхідних для досягнення поставленої мети, а саме:

– провести аналіз предметної області та дослідити типові архітектури та функціонал існуючих платформ електронної комерції;

– визначити функціональні вимоги до інтегрованого вебзастосунку з урахуванням ролей користувачів;

– спроєктувати структуру бази даних для зберігання інформації про користувачів, замовлення, товари, знижки та оплату;

– реалізувати фронтенд-частину платформи з використанням React та забезпечити взаємодію з сервером через REST API;

– створити захищену систему реєстрації, входу та авторизації на основі JWT-токенів із розмежуванням прав доступу;

– додати адміністративний функціонал для управління товарами, знижками, замовленнями та клієнтами;

– протестувати роботу основних модулів системи для перевірки її стабільності, зручності інтерфейсу та відповідності технічним вимогам.

1 АНАЛІЗ СУЧАСНОГО СТАНУ СФЕРИ ЕЛЕКТРОННОЇ КОМЕРЦІЇ

1.1 Актуальність сфери електронної комерції

В умовах стрімкого розвитку цифрових технологій електронна комерція перетворилася з допоміжного каналу продажів на повноцінну бізнес-модель, яка охоплює широкий спектр товарів, послуг і споживачів. Присутність у цифровому середовищі стала не просто бажаною, а необхідною умовою для ефективного функціонування сучасного бізнесу. Вебплатформа з інтуїтивно зрозумілим інтерфейсом, підтримкою онлайн-оплати, персоналізованими знижками та інтеграцією з логістичними сервісами вже давно є стандартом для компаній, які прагнуть до масштабування, підвищення якості обслуговування та формування лояльності клієнтів.

Сьогодні очікування споживача сфокусовані на зручності, швидкості, доступності та індивідуальному підході. Клієнт не бажає витрачати час на дзвінки, уточнення наявності товару чи способів доставки — він очікує повністю автоматизованого й прозорого процесу покупки. Це означає, що вебзастосунок має забезпечувати безперебійну обробку дій користувача: від перегляду каталогу до фінальної оплати і отримання сповіщення про доставку.

Незважаючи на активне поширення e-commerce рішень, значна кількість малих та середніх підприємств усе ще використовують застарілі або частково цифровізовані моделі обслуговування. Серед поширених прикладів — використання лише соціальних мереж для приймання замовлень, обробка заявок вручну або обмежене використання сторонніх маркетплейсів. Такий підхід унеможливлює побудову стабільної клієнтської бази, ускладнює масштабування бізнесу й не відповідає сучасним вимогам до автоматизації.

Натомість створення власної інтегрованої платформи електронної комерції дозволяє суттєво підвищити ефективність управління бізнес-процесами. Така система включає низку взаємопов’язаних модулів: управління товарами та знижками, облік клієнтів, обробка замовлень, онлайн-оплата, аналітика, підтримка

доставок та розширена адміністративна панель. Впровадження такої платформи дозволяє не лише покращити внутрішні процеси, а й забезпечити конкурентну перевагу завдяки зростанню рівня сервісу.

З технічної точки зору, особливої актуальності набуває використання сучасних фреймворків і бібліотек, таких як React, які забезпечують гнучкість, високу швидкодію та масштабованість вебінтерфейсу. Використання React дозволяє реалізувати інтерактивні компоненти, динамічні оновлення даних без перезавантаження сторінки та швидку адаптацію до змін у бізнес-логіці. Це важливо як для користувацької зручності, так і для тривалої підтримки проєкту.

Окрім інтерфейсної частини, важливу роль відіграє й серверна логіка, зокрема використання Node.js та Express, які забезпечують гнучку архітектуру REST API, взаємодію з базою даних (наприклад, MongoDB) та інтеграцію зі сторонніми сервісами, такими як платіжні шлюзи (LiqPay) та логістичні API (Нова Пошта). Такий технічний стек дозволяє реалізувати повноцінну систему електронної комерції з високим рівнем надійності та продуктивності.

Підсумовуючи, можна стверджувати, що розробка інтегрованої вебплатформи електронної комерції є необхідним кроком у напрямку цифрової трансформації бізнесу. Вона дозволяє створити повноцінний онлайн-сервіс, який відповідає вимогам сучасного ринку, забезпечує зручну взаємодію з клієнтом, оптимізує процеси продажу та сприяє зростанню прибутковості підприємства. Актуальність цієї теми підтверджується як попитом на готові рішення, так і потребою у гнучких, масштабованих платформах, здатних адаптуватися до динамічного середовища цифрової економіки.

1.2 Сучасні інструменти створення застосунків для електронної комерції

Розробка сучасних вебзастосунків для електронної комерції вимагає застосування ефективних, масштабованих та перевірених інструментів, які забезпечують як зручний інтерфейс для кінцевих користувачів, так і надійну

серверну логіку. У даному проєкті реалізація вебплатформи базується на повноцінному стеку технологій, який включає Node.js, Express, MongoDB, React, а також допоміжні інструменти, такі як JWT для аутентифікації, Docker для контейнеризації та LiqPay API для реалізації електронних платежів.

Одним із ключових компонентів системи є Node.js — середовище виконання JavaScript на сервері. Завдяки асинхронній моделі обробки подій Node.js забезпечує високу продуктивність, що особливо важливо в системах, орієнтованих на обробку численних запитів одночасно. У поєднанні з фреймворком Express.js створюється легкий і гнучкий HTTP-сервер, який дозволяє швидко реалізовувати RESTful API для взаємодії клієнтської частини із серверною логікою.

Для зберігання даних використовується MongoDB — документоорієнтована NoSQL-база даних, яка ідеально підходить для гнучких та динамічних структур. У рамках цього проєкту MongoDB використовується для збереження інформації про користувачів, замовлення, товари, знижки, кошики та інші важливі сутності. Структура бази є масштабованою та легко адаптується до змін у бізнес-логіці.

На стороні клієнта застосовується React — сучасна JavaScript-бібліотека для створення інтерфейсів користувача. Її компонентна архітектура дозволяє розробляти повторно використовувані елементи UI, які легко підтримувати та масштабувати. У проєкті React використовується для реалізації інтуїтивно зрозумілих сторінок — таких як сторінка товару, кошик, реєстрація та кабінет користувача. React забезпечує швидке оновлення вмісту за рахунок використання віртуального DOM, що покращує продуктивність навіть на повільних пристроях.

Ще одним важливим елементом є JWT (JSON Web Tokens) — технологія, що використовується для безпечної авторизації. Вона дозволяє системі генерувати токени доступу, які зберігають інформацію про авторизованого користувача без необхідності постійного звернення до бази даних. Це забезпечує швидку та безпечну взаємодію між клієнтом і сервером.

У рамках платіжної інтеграції використовується LiqPay API — український платіжний сервіс, що дозволяє здійснювати онлайн-оплати банківськими картками

та іншими способами. Це дає змогу користувачам платформи швидко та зручно оплачувати свої замовлення безпосередньо на сайті.

Ще одним важливим технічним рішенням є Docker — платформа для контейнеризації, яка дозволяє ізолювати програму разом з усіма її залежностями. Це спрощує розгортання системи на сервері, підвищує стабільність і безпеку, а також забезпечує повторюваність середовища розробки та продакшн-екземпляра.

Таким чином, поєднання Node.js, Express, MongoDB, React, JWT, LiqPay API та Docker створює потужне середовище для розробки інтегрованої платформи електронної комерції. Ці технології забезпечують надійність, масштабованість, зручність користування та швидку адаптацію до нових вимог, що робить їх ідеальним вибором для реалізації сучасного вебзастосунку.

1.3 Аналіз аналогічних платформ

Розробка будь-якого складного вебзастосунку вимагає попереднього вивчення вже реалізованих рішень у відповідній галузі. Такий аналіз дозволяє визначити, які підходи є ефективними, а які призводять до зниження якості користувацького досвіду чи до технічних обмежень. У сфері електронної комерції особливо важливо звертати увагу на функціональність, швидкодію, стабільність та гнучкість платформи, а не лише на її візуальну привабливість.

Сучасні користувачі очікують від e-commerce систем не просто красивого інтерфейсу, а насамперед зручного функціоналу, що відповідає їхнім потребам. Вони хочуть швидко знайти потрібний товар, додати його до кошика, обрати спосіб доставки, оплатити онлайн і відстежувати статус замовлення — без перешкод, надмірних кліків або помилок на етапі оформлення.

Відомі системи, як-от Shopify, Magento, OpenCart, Prom.ua та інші, вже реалізували багато з цих функцій, що дозволяє використати їхній досвід як орієнтир. Наприклад, Shopify надає інструменти для швидкого запуску магазину без знань програмування, але обмежує можливості кастомізації в безкоштовному тарифі. Magento, своєю чергою, має потужну архітектуру, але потребує глибокої

технічної підготовки для впровадження. У той час як Prom.ua надає просту систему для локального ринку, але значно поступається у функціональності інтегрованим рішенням.

Ключова мета аналізу аналогів полягає не у повторенні чужих рішень, а у виокремленні найефективніших підходів і уникненні поширених помилок. Наприклад, складна навігація, відсутність чіткої ієрархії категорій або надмірна кількість обов'язкових полів при оформленні покупки — усе це часто призводить до втрати клієнтів. Тому у власній розробці варто впроваджувати лише ті функції, які дійсно підвищують зручність, швидкість і зрозумілість для користувача.

Особливістю розроблюваної платформи є її повна функціональна автономність — система не залежить від сторонніх модулів CMS, що дозволяє реалізовувати специфічні бізнес-вимоги: розширену роботу з кошиком, сегментацію клієнтів, гнучкі знижки, глибоку інтеграцію з платіжними сервісами (LiqPay) та логістикою (Нова Пошта).

На відміну від шаблонних рішень, проєкт орієнтований на адаптовану архітектуру з відкритим кодом, що дозволяє масштабувати систему, оновлювати окремі модулі без переробки всієї платформи та використовувати сучасні підходи до побудови backend-логіки (REST API, JWT-автентифікація, компонентна структура React тощо).

Таким чином, функціональний аналіз аналогічних рішень дозволяє чітко сформулювати вимоги до вебплатформи, орієнтуючись на зручність, ефективність та інтеграцію всіх ключових процесів електронної торгівлі в межах єдиної системи.

З метою якісного проєктування функціоналу інтегрованої платформи електронної комерції доцільно проаналізувати приклади вже існуючих вебрішень. Розглянемо три приклади застосунків-аналогів, що охоплюють різні підходи до реалізації базових функцій інтернет-магазину. Основну увагу приділено реалізації логіки користувача, обробці кошика, системі знижок та інтеграції з платіжними й логістичними сервісами.

Shopify — одна з найпоширеніших SaaS-платформ для запуску онлайн-магазинів. Основна її перевага полягає в простоті налаштування й широкому наборі готових компонентів. Користувачі мають змогу створювати товари, керувати замовленнями, встановлювати ціни й акції. На рис. 1.1 представлено головну сторінку керування магазином в «Shopify».

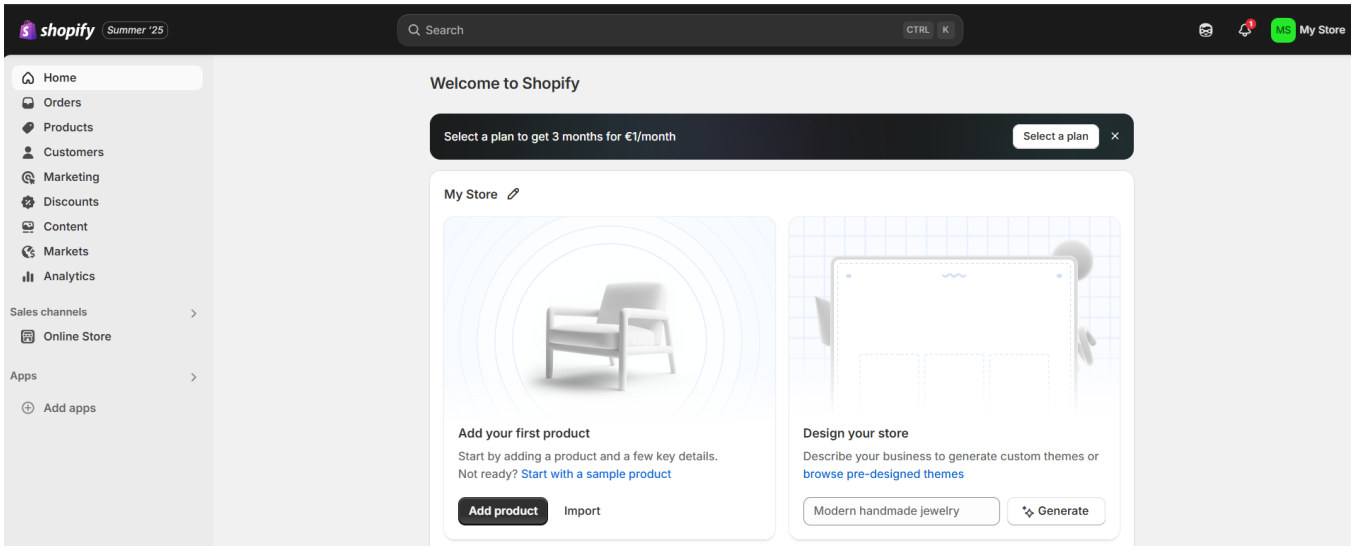

Рисунок 1.1 - Сторінка адмін-панелі «Shopify»

Проте для більшості магазинів, що працюють за специфічною бізнес-логікою або потребують локалізованих інтеграцій (наприклад, українських платіжних систем чи служб доставки), можливостей Shopify виявляється недостатньо. Впровадження нестандартного функціоналу вимагає або придбання додаткових модулів, або використання сторонніх рішень, що здорожчує підтримку платформи.

На відміну від Shopify, розроблювана система спочатку орієнтована на вбудовану інтеграцію з LiqPay та API Нової Пошти, що робить її адаптованою до українського ринку з самого початку. У системі також буде реалізовано повноцінний облік замовлень, розширену логіку кошика й механізми для запуску динамічних знижок.

Тепер розглянемо українську платформу Prom.ua — маркетплейс, орієнтований на малий і середній бізнес в Україні. Платформа дозволяє

підприємцям швидко створити сторінку магазину, завантажити товари та приймати замовлення. Вона має базову інтеграцію з українськими сервісами й підтримує кілька способів оплати, що є її беззаперечною перевагою. На рис. 1.2 зображено вигляд поля вибору оплати замовлення на платформі «Prom.ua».

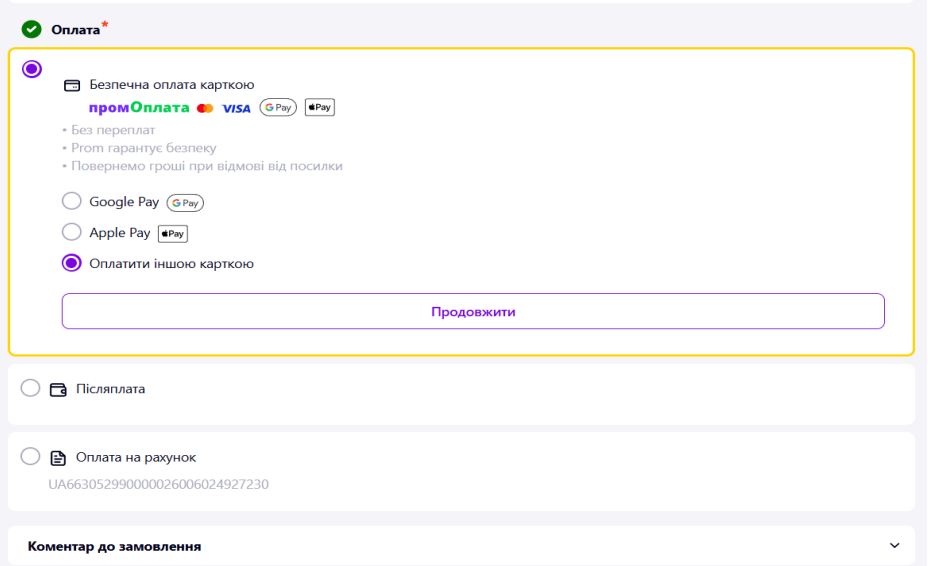

Рисунок 1.2 – Інтеграція платформи Prom.ua з платіжними системами

Серед недоліків Prom.ua — відсутність контролю над кодом та обмежена кастомізація бізнес-логіки. Користувачі не можуть впливати на архітектуру застосунку або розширювати його функціонал під специфічні потреби. Також не всі продавці мають доступ до аналітики в реальному часі або CRM-функцій, що ускладнює масштабування проєктів.

На відміну від Prom.ua, наша система розробляється як повністю кастомізований програмний продукт з відкритою архітектурою. Передбачається можливість розширення через окремі модулі, гнучке налаштування структури знижок, підтримка покинутих кошиків, e-mail-нагадування та автоматизація взаємодії з клієнтами. На рис. 1.3 зображено головну сторінку платформи «Prom.ua».

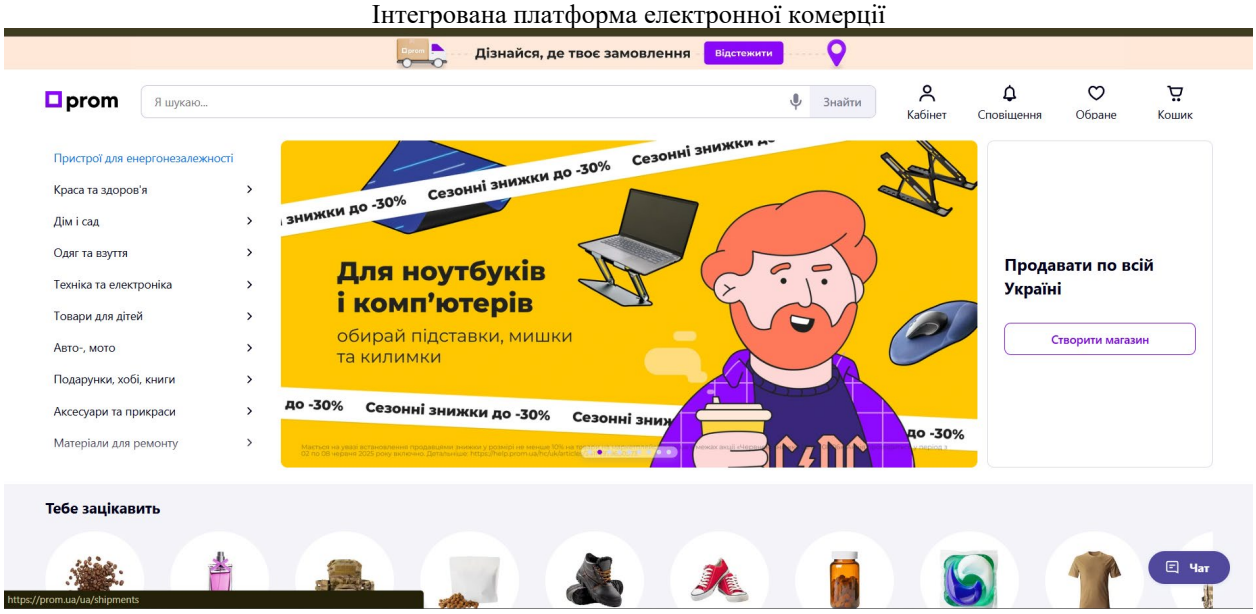

Рисунок 1.3 – Головна сторінка платформи «Prom.ua»

OpenCart — ще одна популярна CMS для електронної комерції з відкритим кодом. Її основною перевагою є гнучкість налаштувань і розширена система модулів. Вона дозволяє реалізовувати складні конфігурації з підтримкою кількох мов, валют, складів тощо.

Однак, OpenCart вимагає від розробника доброго знання PHP і має досить складну структуру, що ускладнює обслуговування й оновлення великих проєктів. Інтеграція з REST API, платіжними шлюзами чи кастомними CRM часто потребує написання власних рішень з нуля, що сповільнює розробку.

На рис. 1.4 зображено вигляд панелі користувача в OpenCart.

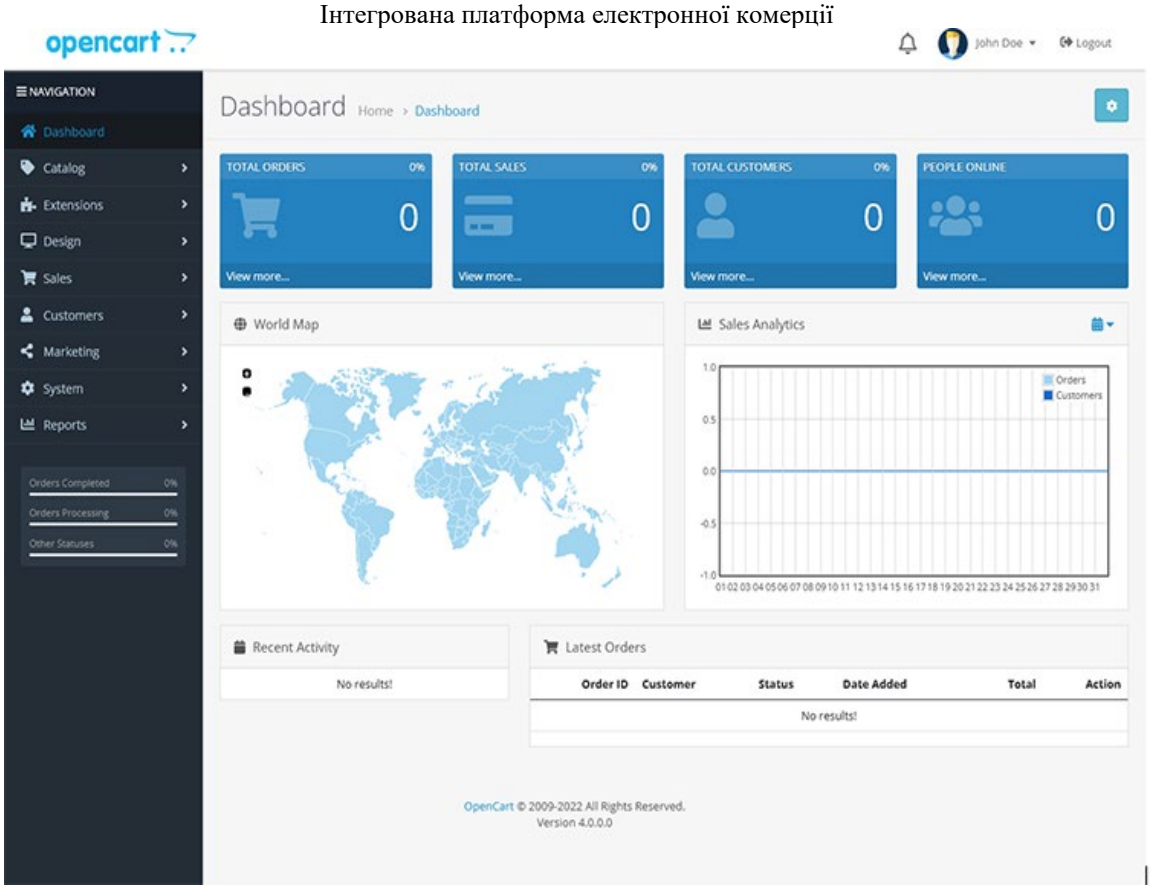

Рисунок 1.4 – Вигляд адмін-панелі користувача «OpenCart»

Аналіз вищенаведених систем показує, що навіть відомі платформи мають обмеження, пов’язані або з закритістю архітектури, або з надмірною складністю адаптації. Це підкреслює доцільність створення власного рішення, орієнтованого на цільові функціональні завдання, такі як динамічні кошики, налаштовувані знижки, інтеграція з локальними сервісами доставки й оплати, а також зручний інтерфейс для клієнтів та адміністраторів.

З результатами аналізу популярних e-commerce платформ можна ознайомитися в таблиці 1.1, де порівняно основні переваги та недоліки кожного рішення. Також подано, як ці аспекти враховані у власному проєкті з розробки інтегрованої вебплатформи електронної комерції.

Таблиця 1.1 – Порівння характеристика платформ електронної комерції

№	Платформа	Недоліки	Переваги	Наша платформа
1	Shopify	Обмежена кастомізація, відсутність локальних інтеграцій	Просте налаштування, готові компоненти, SaaS-модель	Підтримка LiqPay та Нової Пошти
2	Prom.ua	Закритий код, відсутність гнучкого функціоналу, обмежена аналітика	Підтримка українського ринку, інтеграція з оплатою і доставкою	Розширене управління замовленнями, гнучкі знижки,ширша аналітика
3	OpenCart	Високий поріг входу, складна розробка REST API, повільна адаптація	Відкритий код, багатомовність, система модулів	Використано сучасний стек (Node.js, React), REST API, авторизація через JWT, кастомна маршрутизація

Таким чином, сучасна e-commerce платформа має не лише виконувати базові функції продажу, а й забезпечувати гнучке адміністрування, швидку інтеграцію нових модулів, безпечну авторизацію та комфортну взаємодію з кінцевим користувачем. Успішне поєднання технічної функціональності та адаптованого під місцевий ринок підходу створює умови для розробки ефективного, сучасного й масштабованого продукту.

1.4 Постановка задачі

У сучасних умовах цифрової економіки особливої актуальності набувають інструменти, що дозволяють підприємствам малого та середнього бізнесу автоматизувати продаж товарів, обслуговування клієнтів та керування логістичними процесами. Електронна комерція перестала бути лише доповненням до традиційних форм торгівлі — вона стала повноцінною платформою для ведення бізнесу. У зв'язку з цим зростає потреба у створенні ефективних, функціонально насичених і безпечних вебрішень, що можуть забезпечити безперебійну взаємодію з клієнтом на всіх етапах — від вибору товару до його доставки.

Відсутність сучасної інтегрованої платформи, що об'єднує онлайн-каталог, кошик, оплату, доставку, знижки та облікові записи користувачів, значно знижує конкурентоспроможність бізнесу. Без такої системи ускладнюється обробка замовлень, підвищується навантаження на персонал, а рівень довіри з боку клієнтів зменшується. Тому виникає потреба у розробці вебплатформи, яка підтримує:

- структуровану подачу товарів, категорій, цін та знижок;
- реєстрацію користувачів та створення особистого кабінету;
- формування та редагування кошика з урахуванням акцій та купонів;
- можливість здійснення онлайн-оплати через інтеграцію з платіжною системою (LiqPay);
- вибір методу доставки та підключення до API логістичної служби;
- адміністративну панель для керування товарами, замовленнями, знижками та користувачами;
- захищене зберігання персональних даних відповідно до стандартів безпеки.

Технічна реалізація має базуватися на сучасному стеку технологій, зокрема Node.js, Express, MongoDB та React, що дозволяють створювати масштабовані, швидкі та інтерактивні вебзастосунки. Серверна частина забезпечуватиме REST API, автентифікацію через JWT, підключення до бази даних та взаємодію з

зовнішніми сервісами. Клієнтська частина — адаптивний інтерфейс з компонентною структурою, що надає користувачу зручні механізми навігації, оформлення замовлень та взаємодії з системою.

Цільова аудиторія вебплатформи — покупці та адміністратори. Для покупців реалізовується функціонал авторизації, перегляду товарів, управління кошиком, застосування знижок, оформлення доставки та перегляду історії замовлень. Для адміністраторів — доступ до системи управління контентом: створення нових товарів, редагування цін, управління замовленнями, аналітика та контроль за активністю користувачів. Важливо, щоб інтерфейс був інтуїтивно зрозумілим для користувачів із різним рівнем технічної підготовки.

Окрему увагу слід приділити надійності системи: передбачено автентифікацію з використанням JWT, обмеження доступу до адміністративних функцій, резервне копіювання бази даних та можливість її шифрування. Це дозволить мінімізувати ризики втрати чи витоку даних та забезпечити стабільність роботи навіть при високому навантаженні.

Таким чином, у рамках цієї роботи ставиться задача розробити повноцінну вебплатформу електронної комерції, яка відповідатиме сучасним вимогам до безпеки, гнучкості, зручності користування та технічної підтримки. Проєкт сприятиме автоматизації бізнес-процесів, підвищенню лояльності клієнтів та створенню цифрової основи для стабільного зростання компанії у сфері онлайн-торгівлі.

Висновки до розділу 1

У першому розділі було проведено всебічний аналіз предметної області електронної комерції, зокрема досліджено актуальні вимоги до сучасних онлайн-магазинів, що орієнтовані на зручність користувача, безпеку та адаптивність. Визначено ключові потреби потенційних покупців: можливість швидко знаходити товари, зручно додавати їх до кошика, безпечно оформлювати замовлення та контролювати особисту інформацію.

Особливу увагу приділено вивченню аналогічних сервісів та рішень у сфері e-commerce, що дозволило виявити як ефективні підходи до побудови інтерфейсу й структури сайту, так і типові недоліки, які слід уникати при реалізації власного застосунку. До таких недоліків можна віднести перевантаження головної сторінки, відсутність мобільної адаптації, недостатню деталізацію сторінок товарів, а також складну або заплутану систему реєстрації.

На основі отриманих результатів було сформовано загальні функціональні й нефункціональні вимоги до платформи, серед яких: інтуїтивний і легкий у навігації інтерфейс, можливість реєстрації та авторизації користувачів, функціональний кошик, система замовлень із підтримкою онлайн-оплати та інтеграцією з поштовими сервісами доставки.

Крім того, обґрунтовано вибір технологічного стеку для реалізації платформи. Фронтенд частина створюється на базі бібліотеки React, яка забезпечує гнучку, компонентну архітектуру та підтримку адаптивної верстки. Для бекенду обрано Node.js з Express як надійний та масштабований інструмент обробки API-запитів. Зберігання даних здійснюється за допомогою MongoDB, що забезпечує гнучкість роботи з вкладеними структурами та масивами — необхідну для динамічної онлайн-торгівлі.

Таким чином, у першому розділі було закладено концептуальну основу для побудови повноцінного вебзастосунку у сфері електронної комерції. Визначення предметної області, аналіз аналогів та вибір оптимального технологічного підходу стали вихідною точкою для розробки логіки, дизайну та архітектури майбутньої системи.

2 ПРОЄКТУВАННЯ ІНТЕГРОВАНОЇ ПЛАТФОРМИ ЕЛЕКТРОННОЇ КОМЕРЦІЇ

2.1 Формування структурної моделі вебзастосунку

Структура вебзастосунку платформи електронної комерції базується на логіці взаємодії різних типів користувачів із системою, що дозволяє точно визначити їхні потреби та сформувати зручний і адаптований функціонал. В архітектурі чітко виділяються два основні напрями: фронт для кінцевого користувача (покупця) та адміністративна частина для менеджменту та обслуговування платформи.

Користувацький інтерфейс починається з головної сторінки, яка виступає відправною точкою для навігації. Тут розміщено базову інформацію про платформу, список акційних товарів, розділи каталогу, можливість переходу до реєстрації або входу в особистий кабінет. Основна навігація реалізована через меню з доступом до категорій товарів, сторінки “Про нас”, “Доставка і оплата”.

На рис. 2.1 зображено загальну структуру користувацької частини платформи, її головні розділи та взаємозв’язки між компонентами.

Користувач може переглядати каталог продукції, переходити на сторінку конкретного товару, додавати його до кошика та переходити до оформлення замовлення. Після авторизації відкривається доступ до особистого кабінету, де можна переглядати історію покупок.

Інтеграція з платіжною системою LiqPay дозволяє здійснювати оплату замовлень безпосередньо на сайті. Також підключено API служби доставки “Нова Пошта”, завдяки чому користувач може одразу обрати зручне відділення для отримання товару.

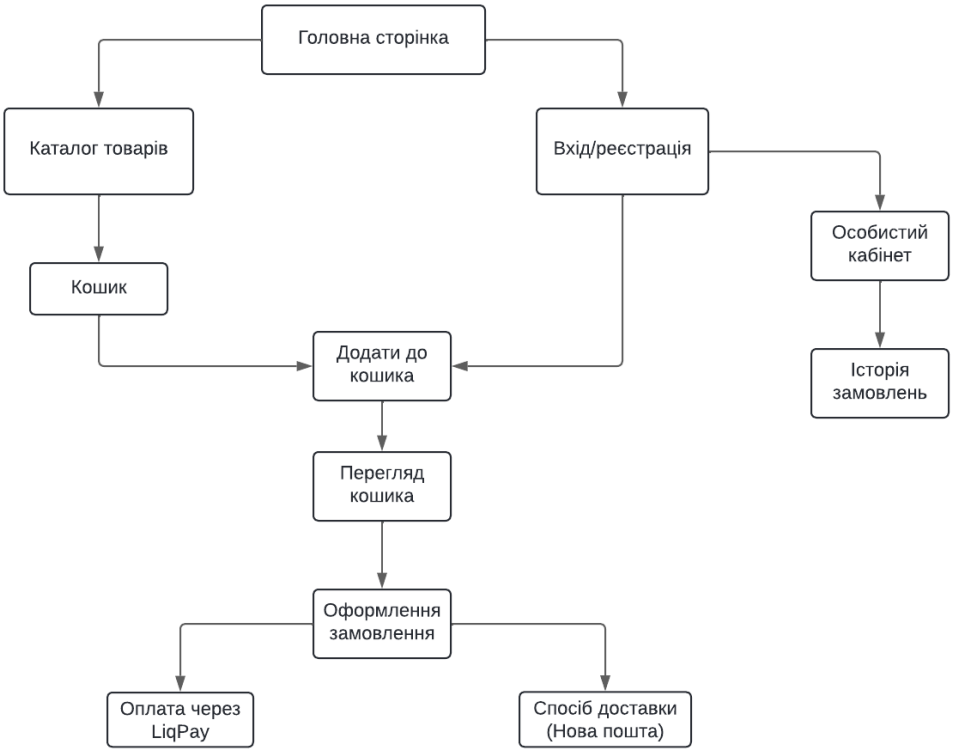

Рисунок 2.1 – Структура користувацької частини платформи

Адміністративна частина вебплатформи є окремим модулем системи з обмеженим доступом, доступ до якого мають лише авторизовані користувачі з відповідними правами — зокрема адміністратори та менеджери. Вхід до адмін-панелі захищено механізмом автентифікації, а доступ до функцій регламентується за допомогою ролей. Це забезпечує безпеку даних та контроль за процесами управління платформою.

Функціональність адміністративного інтерфейсу охоплює кілька ключових напрямів. Перш за все, це керування товарами, що включає створення нових позицій, редагування вже наявних, завантаження графічних матеріалів, призначення знижок і акційних пропозицій. Таким чином адміністратор має змогу оперативно оновлювати вітрину онлайн-магазину відповідно до потреб бізнесу.

Наступним критично важливим блоком є система керування замовленнями, яка дозволяє переглядати статус кожного замовлення, здійснювати підтвердження

або скасування, а також контролювати оплату і логістику. Це забезпечує своєчасну обробку замовлень і комунікацію з клієнтами.

Важливу роль відіграє розділ модерації користувачів, де адміністратор може переглядати профілі клієнтів, змінювати їхні ролі (наприклад, активувати або деактивувати облікові записи). Це сприяє підтриманню порядку й безпеки у системі.

Окрему функціональну групу становить аналітичний модуль, у якому доступні графіки та звіти щодо обсягів продажів, середнього чека, рівня активності користувачів. Такі дані дозволяють керівництву приймати обґрунтовані бізнес-рішення на основі реальної статистики.

Для підтримки лояльності клієнтів реалізовано систему знижок і промокодів, яка дає змогу створювати як загальнодоступні, так і персоналізовані маркетингові пропозиції. Адміністратор може гнучко керувати цим інструментом - умови використання та цільову аудиторію.

Також адміністративний інтерфейс включає блок загальних налаштувань платформи, який охоплює редагування інформаційних сторінок (наприклад, «Про нас», «Контакти», «Політика конфіденційності»), що дозволяє оперативно оновлювати вміст сайту відповідно до змін у структурі або стратегії компанії.

Загалом, адміністраторська частина вебзастосунку забезпечує повноцінне управління всіма ключовими процесами в межах електронної комерції — від товарного обліку до аналітики — і виступає критичним елементом для стабільного функціонування платформи.

На рис. 2.2 представлено логічну структуру адміністративної частини платформи з основними взаємозв’язками модулів.

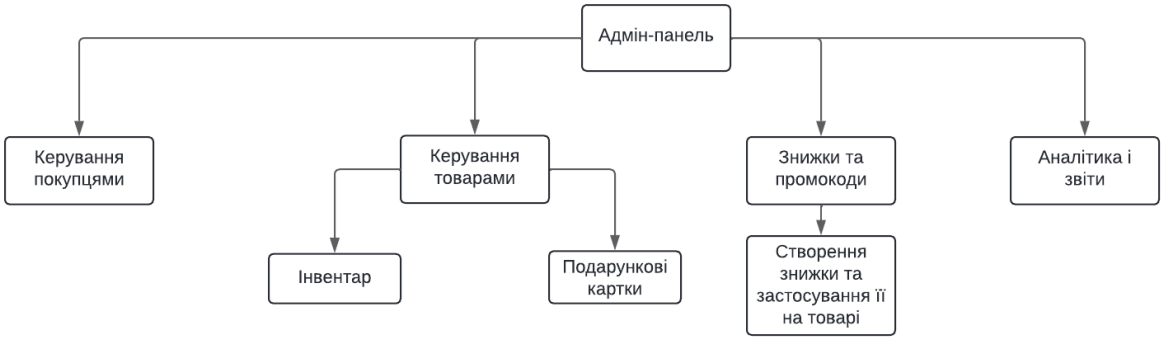

Рисунок 2.2 – логічна структура адміністративної частини платформи

Запропонована архітектура має модульну побудову, що дозволяє легко масштабувати систему, додавати нові компоненти та адаптувати функціонал під різні бізнес-моделі. Структура забезпечує ефективну роботу з великою кількістю користувачів, одночасне обслуговування замовлень та гнучке управління контентом.

Таким чином, розроблена архітектура платформи враховує інтереси як клієнтів, так і адміністраторів системи, підтримує стабільну взаємодію між ними та формує цілісний цифровий простір для реалізації повного циклу електронної комерції.

2.2 Проєктування користувацького інтерфейсу

Дизайн інтерфейсу вебплатформи електронної комерції було розроблено з урахуванням сучасних стандартів UI/UX і орієнтацією на інтуїтивну взаємодію користувача з системою. Особливу увагу приділено зручності навігації, контрастності, структурованості інформації та адаптивності до різних типів пристроїв.

Інтерфейс кожної сторінки створювався окремо з урахуванням її функціонального навантаження, водночас усі елементи дизайну підпорядковано єдиній стилістиці — світлій темі з виразними акцентами, чіткою типографікою та великою кількістю білого простору. В основі стилю лежить принцип мінімалізму,

який сприяє швидкому орієнтуванню користувача навіть без попереднього досвіду взаємодії з платформою.

Головна сторінка виконує роль навігаційного хабу. У верхній частині розміщено назву магазину а також кнопки входу, реєстрації та доступу до кошика, у випадку, якщо ми увійшли в аккаунт. Центральний блок включає банери з актуальними акціями, короткий опис продукту, у випадку, якщо користувач наведеться мишкою на товар. Зовнішній вигляд головної сторінки показано на рис. 2.3.

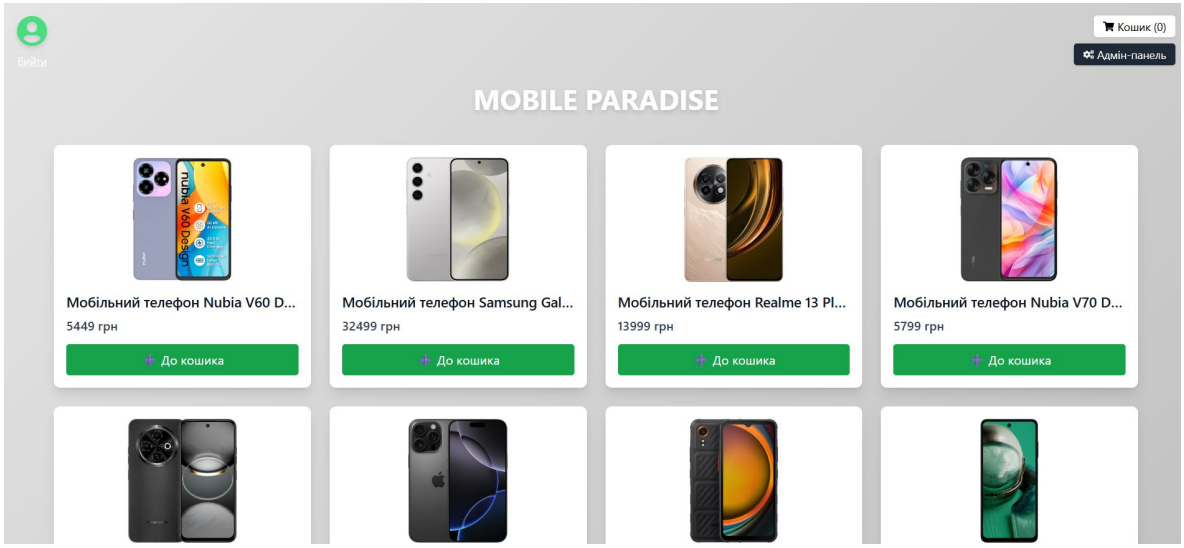

Рисунок 2.3 – Головна сторінка платформи

Особистий кабінет користувача побудовано у мінімалістичному, що полегшує доступ до функціоналу: історія замовлень, зміна контактних даних, управління паролем. Усі елементи оформлено у спокійній кольоровій гамі, з мінімальним використанням іконок та зрозумілими підписами. Зовнішній вигляд сторінки кабінету користувача показано на рис. 2.4.

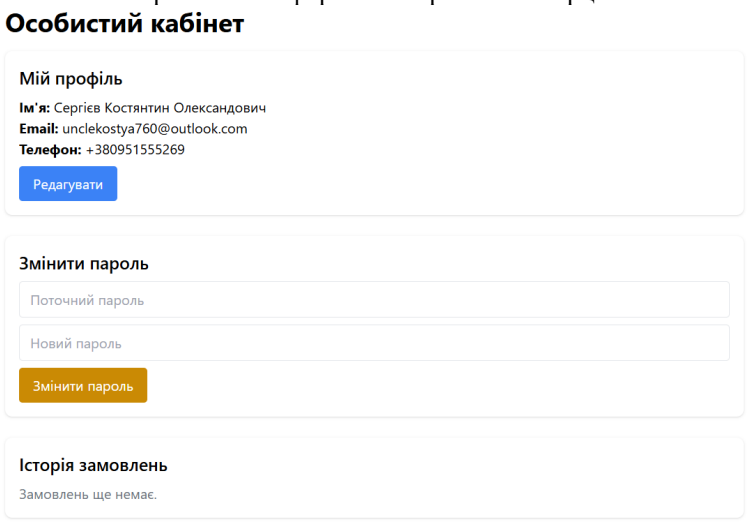

Рисунок 2.4 – Сторінка редагування даних покупця

Кошик у реалізованому вебзастосунку виступає як повноцінна сторінка, що дозволяє користувачеві зручно переглядати та керувати списком вибраних товарів перед оформленням замовлення. Кожна позиція в кошику містить назву товару, його кількість, ціну за одиницю та підсумкову вартість. Користувач має можливість змінювати кількість товарів, а також видаляти непотрібні позиції — усі ці дії миттєво оновлюють фінальну суму до оплати. Сторінка кошику показана на рис. 2.5.

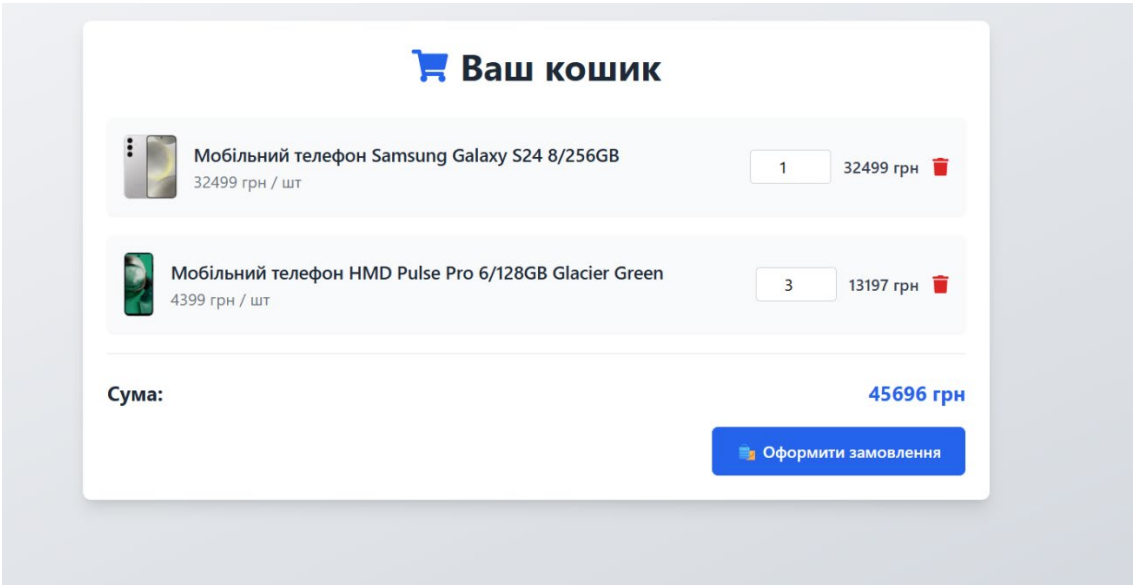

Рисунок 2.5 – Зовнішній вигляд кошику

Інтерфейс реалізовано у вигляді таблиці або вертикального списку, де кожен товар представлено окремим блоком. У правій частині розміщено підсумок: загальна кількість товарів, вартість, а також кнопка переходу до оформлення замовлення. Усі обрахунки виконуються автоматично — при кожній зміні кількості чи видаленні позиції сторінка не перезавантажується, завдяки чому користувацький досвід є максимально динамічним.

Оформлення замовлення представлено у вигляді окремої сторінки з послідовним введенням усієї необхідної інформації. Першим блоком користувач бачить форму, де вводить або підтверджує свої контактні дані: ПІБ, номер телефону, адресу електронної пошти. Поля мають перевірку правильності заповнення та повідомлення про помилки, які відображаються одразу після взаємодії з відповідним полем. Сторінка оформлення замовлення показана на рис. 2.6.

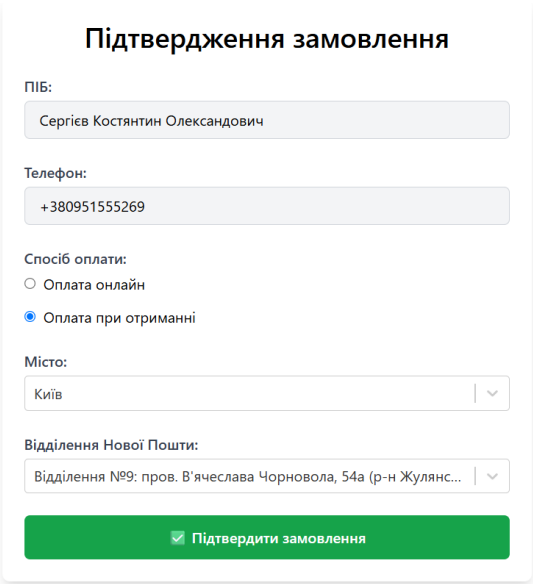

Рисунок 2.6 – Сторінка оформлення замовлення

Усі елементи сторінки оформлення замовлення розміщені компактно, у зручній послідовності, що відповідає логіці користувача. Кожен блок оформлений

в окремому контейнері, візуально розділеному від інших, що покращує сприйняття та навігацію. Кнопка завершення замовлення розміщена в нижній частині форми, а в разі помилки система підсвічує відповідні поля червоним кольором та надає пояснення.

Сайдбар навігації адмін-панелі, який використовується замість традиційного верхнього меню, виконує роль постійного орієнтира для користувача. Він розміщується ліворуч і залишається фіксованим під час прокручування сторінки. У сайдбарі зібрано всі ключові розділи платформи: магазин, замовлення, товари, знижки, аналітика, профіль тощо. Вигляд бічного меню адмін-панелі показано на рис. 2.7.

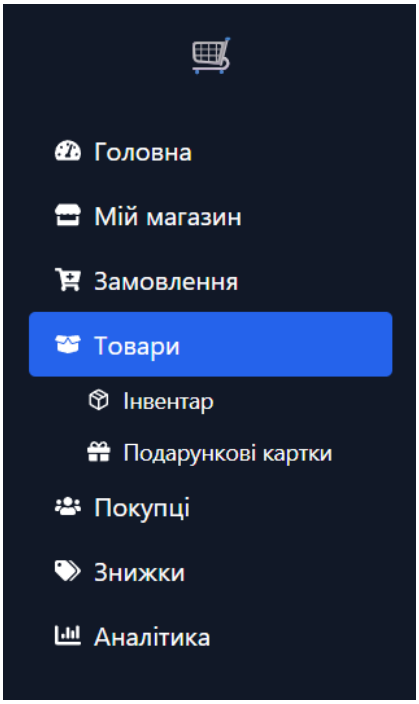

Рисунок 2.7 – Вигляд бічного меню адмін-панелі

Навігаційні елементи мають логічну ієрархію та піктограми, що полегшують візуальне сприйняття. Активна сторінка підсвічується, а вкладені меню автоматично розгортаються. Дизайн витримано в мінімалістичному стилі, що робить інтерфейс універсальним і зручним як для клієнтів, так і для менеджерів.

2.3 Адаптивність інтерфейсу платформи

У сучасному цифровому середовищі користувачі взаємодіють із вебресурсами з широкого спектра пристроїв: від стаціонарних комп’ютерів з великими моніторами до ноутбуків, планшетів і смартфонів з різними розмірами екранів. У зв’язку з цим адаптивний дизайн перестав бути лише бажаною опцією — він перетворився на обов’язкову складову професійної розробки вебінтерфейсів.

Адаптивний дизайн — це підхід до створення вебінтерфейсів, який забезпечує автоматичну зміну структури, зовнішнього вигляду та функціональності сайту залежно від розміру екрана, орієнтації пристрою та технічних можливостей браузера. Метою цього підходу є гарантувати комфортне використання ресурсу незалежно від того, який саме пристрій обрав користувач.

Адаптивність полягає не лише у візуальному масштабуванні елементів, а й у зміні логіки розміщення блоків, перемиканні між горизонтальними та вертикальними форматами, оптимізації шрифтів, кнопок та форм введення. Правильно реалізована адаптивність значно знижує рівень відмов (bounce rate), покращує залучення користувачів та підвищує загальний рівень задоволеності від взаємодії з платформою.

Головна сторінка інтернет-магазину є першою точкою взаємодії користувача з вебзастосунком, тому її адаптивність має критичне значення для збереження уваги відвідувача, зручної навігації та залучення до перегляду товарів. У розробленій платформі електронної комерції головна сторінка реалізована як гнучка візуальна сітка з адаптивною логікою відображення всіх ключових елементів: списку товарів, інформаційного заголовка, кнопок додавання до кошика, а також навігаційних блоків (sidebar, профіль, авторизація тощо). Вигляд головної сторінки на мобільних пристроях показано на рис. 2.8.

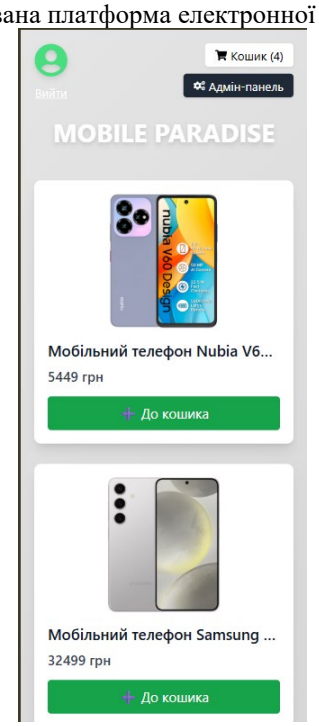

Рисунок 2.8 – Головна сторінка на мобільних пристроях

Основний функціональний блок сторінки — грид-контейнер із товарами, який автоматично перебудовується залежно від ширини вікна браузера. На широких екранах використовується сітка з 4 колонками — товари розміщуються по чотири в ряд, що дозволяє охопити більшу кількість контенту без потреби прокручування. На середніх екранах грид автоматично перебудовується на 2–3 колонки, зберігаючи розміри карток та комфорт читання. На мобільних пристроях використовується одна колонка, і кожна товарна картка займає майже всю ширину екрана, забезпечуючи гарну видимість зображення, назви, ціни та кнопки. Це досягається за допомогою Tailwind CSS-класів, що дозволяють адаптувати сітку під задані брейкпоінти.

Усередині кожної картки товару реалізовано окрему адаптивну структуру:

– зображення товару масштабуються автоматично за допомогою класів TailwindCSS, які забезпечують однакову висоту зображення незалежно від його пропорцій;

– назва товару обрізається за допомогою truncate, що запобігає виходу тексту за межі картки на вузьких екранах;

– ціна та кнопка «До кошика» залишаються у фіксованій позиції внизу картки незалежно від кількості контенту в назві чи описі;

– адаптивна анімація опису з'являється після наведення (або фокусування на елементі), але не активується на мобільних пристроях, що враховує відсутність hover-подій на сенсорних екранах.

Замість традиційного хедера в цьому проєкті реалізовано контекстну навігацію через плаваючі елементи управління, розташовані у верхніх кутах. У лівому верхньому куті розміщено аватар профілю користувача і кнопка «Вийти». Цей блок зменшується на мобільних пристроях до однієї іконки, а решта — приховується або згортається. У правому верхньому куті відображається кнопка переходу до кошика, а також за потреби — кнопка адміністративної панелі. Їхній розмір, відступи та відображення регулюються класами Tailwind. На мобільних пристроях ці кнопки не перекривають контент, оскільки завдяки позиціюванню та адаптивним паддінгам контент починається нижче них.

Секція з пагінацією теж оптимізована: на мобільних пристроях відображається як компактний рядок кнопок з можливістю горизонтальної прокрутки, а на великих екранах усі сторінки показані повністю. Використання класів TailwindCSS дозволяє кнопкам не виходити за межі екрану навіть при великій кількості сторінок.

Сторінка оформлення замовлення є одним із ключових етапів взаємодії користувача з електронною комерцією. Вона завершує ланцюжок прийняття рішення та має бути максимально зручною, інтуїтивно зрозумілою та швидкою у використанні — незалежно від типу пристрою, на якому її відкрито. У даному проєкті сторінка /checkout реалізована з глибоким урахуванням принципів адаптивного дизайну, що гарантує комфортну взаємодію як на широкоформатних моніторах, так і на смартфонах.

Весь контент CheckoutPage розташовано в центральному стовпчику, обмеженому по ширині (max-w-xl) з автоматичним центрованим вирівнюванням (mx-auto) і внутрішнім відступом (p-6). Це забезпечує зручне візуальне читання

контенту, незалежно від ширини екрана, та запобігає розтягуванню форм на великих екранах. Завдяки класам Tailwind, таких як mt-8 mb-16, створено візуальне відокремлення сторінки від загального фону, а також простір для комфортного скролінгу на мобільних пристроях. Зовнішній вигляд сторінки на мобільних пристроях показана на рис. 2.9.

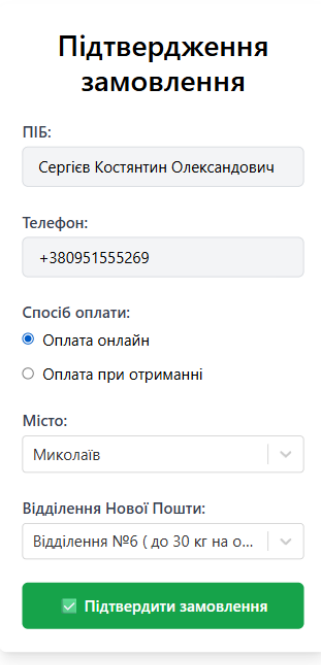

Рисунок 2.9 – Форма підтвердження замовлення на мобільних пристроях

Заголовок «Підтвердження замовлення» реалізовано з адаптивним класом text-3xl, який на мобільних пристроях виглядає досить великим, але не перевантажує екран, а на планшетах і ПК — достатньо помітний. Центрування (text-center) гарантує однакове розміщення на всіх розмірах вікна.

Для вибору міста та відділення Нової Пошти використано компонент Select, який сам по собі підтримує адаптивність — список розгортається поверх інших елементів, а ширина поля налаштовується відповідно до ширини контейнера. На мобільних пристроях випадаючі списки відкриваються у повноекранному або

блоковому режимі (в залежності від браузера), що зручно для сенсорного керування.

Блок вибору способу оплати («Оплата онлайн» або «При отриманні») реалізований як вертикальна група радіокнопок (flex flex-col gap-3). Це дозволяє елементам рівномірно розміщуватись один під одним, не порушуючи структуру на вузьких екранах. Розмітка інтуїтивна і не вимагає горизонтального скролу.

Усі проміжні стани — очікування, обробка платежу, повідомлення про помилки — реалізовані як центровані елементи, які однаково добре виглядають на всіх пристроях. Це створює консистентний користувацький досвід, де незалежно від типу пристрою користувач має чітке розуміння, що відбувається.

Спроєктована платформа електронної комерції демонструє якісну реалізацію адаптивного дизайну, що охоплює всі ключові сторінки — від головної до оформлення замовлення та особистого кабінету. Всі елементи інтерфейсу динамічно підлаштовуються під ширину екрана, зберігаючи логічну структуру, читабельність та функціональність.

2.4 Проєктування структури даних платформи

Для збереження структурованих даних вебплатформи електронної комерції було обрано документоорієнтовану базу даних MongoDB. Такий вибір є обґрунтованим для застосунків, що реалізовані за допомогою JavaScript-стеку (зокрема MERN), оскільки MongoDB чудово поєднується з Node.js та Express і забезпечує зручну роботу з вкладеними об'єктами, високий рівень масштабованості, а також високу швидкодію при обробці великих масивів даних.

База даних була спроєктована відповідно до логіки роботи онлайн-магазину та охоплює основні сутності: користувачів, товари, кошики, замовлення. Кожна колекція відповідає за зберігання окремого шару функціональності сайту та підтримує необхідні взаємозв'язки через ідентифікатори.

Загальна структура включає наступні ключові колекції:

– покупці : ця колекція містить дані про зареєстрованих покупців. Зберігається ім’я покупця, email (унікальний ідентифікатор входу), номер телефону, хешований пароль та булевий прапорець isAdmin, який визначає роль користувача, кошик);

– товари: в цій колекції зберігається повна інформація про асортимент магазину, де кожен товар має унікальний _id, назву (title), поточну ціну (price), опціональну стару ціну (originalPrice), опис, а також шлях до зображення (imageUrl);

– кошик: логіка реалізована через окрему колекцію або поле в користувачеві, що містить масив позицій з посиланнями на товари (productId) та їх кількість (quantity);

– замовлення: ця колекція зберігає усі замовлення, які були сформовані користувачами, структура кожного замовлення включає: ПІБ, email, телефон, перелік товарів (масив з полями title, price, quantity), загальну суму (totalPrice), спосіб оплати (cash або online), статус оплати (isPaid), обрану точку доставки (місто та відділення Нової Пошти), а також дату створення.

Проєктування структури бази даних також передбачає можливість фільтрації, пошуку та зручної генерації аналітики. Наприклад, адміністративна панель дозволяє переглядати статистику замовлень за датами, кількістю та сумами, що реалізується через агрегаційні запити MongoDB.

Усі запити до бази даних здійснюються через Express.js API. На сервері реалізована перевірка прав доступу, автентифікація токенів, перевірка вхідних даних (через валідаційні схеми), а також обробка помилок. Таким чином, структура бази даних не лише є сховищем, а й критично важливою частиною безпечної серверної логіки платформи.

У цілому, проєктована структура є масштабованою і легко розширюваною. За потреби вона може бути доповнена новими колекціями, такими як: повідомлення, оголошення, модерація, повернення товарів тощо. Гнучкість MongoDB дає змогу працювати з вкладеними структурами (наприклад, масив

товарів у замовленні або картки у кошику), що значно спрощує моделювання реальних бізнес-процесів і прискорює доступ до даних.

Висновки до розділу 2

У другому розділі було здійснено поетапне проєктування структури, інтерфейсу та даних інтегрованої платформи електронної комерції. Розробка охоплювала як архітектуру вебзастосунку, так і дизайн користувацького інтерфейсу, що орієнтований на простоту взаємодії, ефективність у навігації та зручність на всіх типах пристроїв — від комп'ютерів до смартфонів.

Значну увагу приділено візуальній структурі сторінок: кожен елемент дизайну створювався з урахуванням цільового функціонального навантаження — від списку товарів до процесу оформлення замовлення. Інтерфейс забезпечує інтуїтивне сприйняття, завдяки чому користувач швидко орієнтується у навігації, фільтрації та оформленні покупки. Усі елементи підтримують адаптивне відображення, що дозволяє працювати з платформою на будь-якому пристрої без втрати зручності чи функціональності.

Особливу увагу приділено реалізації адаптивності інтерфейсу. Кожна сторінка (головна, кошик, оформлення замовлення, профіль, аутентифікація тощо) оптимізована під різні розміри екранів і взаємодії користувача. Упроваджено динамічні блоки, спрощені форми введення, масштабовану верстку та інтуїтивно зрозумілу структуру — усе це значно підвищує якість користувацького досвіду.

Ще одним важливим аспектом стало проєктування структури даних. У роботі було застосовано документоорієнтовану базу даних MongoDB, що ідеально підходить для динамічних JavaScript-застосунків. Визначено основні сутності: користувачі, товари, елементи кошика, замовлення, що пов'язані між собою за допомогою ідентифікаторів і вкладених структур. Такий підхід забезпечує не лише ефективне зберігання, а й надійний контроль доступу, швидку обробку даних і можливість масштабування у майбутньому.

Таким чином, у другому розділі було реалізовано комплексне проєктування майбутнього вебсайту — як з точки зору візуального сприйняття, так і технічної архітектури. Створене рішення поєднує сучасний дизайн, зручність використання та технологічну гнучкість, що закладає надійну основу для повноцінної електронної комерційної платформи.

3 МОДЕЛЮВАННЯ ФУНКЦІОНАЛУ ІНТЕГРОВАНОЇ ПЛАТФОРМИ ЕЛЕКТРОННОЇ КОМЕРЦІЇ

3.1 Взаємодія користувача з платформою: прецеденти та ролі

У цьому підрозділі здійснено моделювання користувацьких сценаріїв взаємодії із системою за допомогою діаграми прецедентів (Use Case Diagram). Така діаграма є важливою складовою візуального аналізу архітектури вебзастосунку, оскільки дозволяє описати поведінку системи з точки зору зовнішніх користувачів (акторів) та окреслити функціональні межі кожної ролі.

У межах інтегрованої платформи електронної комерції, розробленої в межах цього проєкту, було виділено три ключові ролі користувачів:

– гість (незареєстрований користувач);
– зареєстрований користувач (покупець);
– адміністратор.

Гість є початковою точкою взаємодії із сайтом для більшості користувачів.. Діаграма прецедентів для користувача з такою роллю, показана на рис. 3.1.

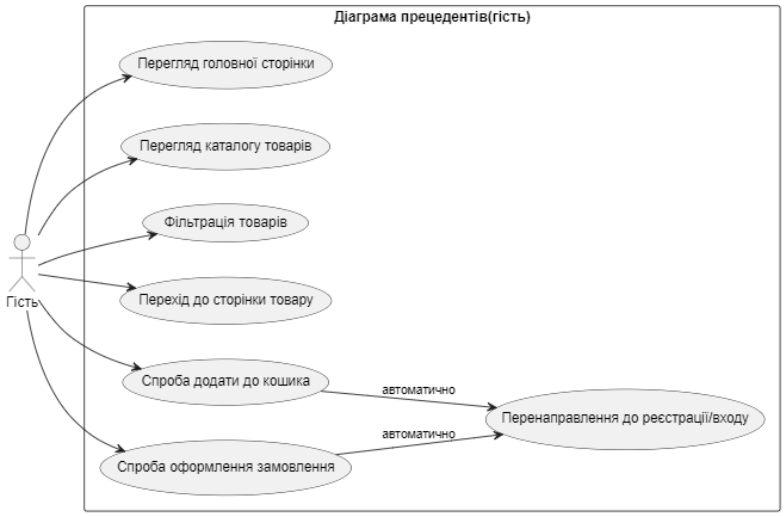

Рисунок 3.1 – Діаграма прецедентів(гість)

У цій ролі він має змогу ознайомитися з публічно доступною частиною функціоналу вебзастосунку. Зокрема, він може переглядати головну сторінку, досліджувати каталог товарів, фільтрувати позиції. У випадку бажання оформити замовлення або при спробі додати товар у кошик, гість буде переадресований до форми реєстрації або авторизації. Таким чином, діапазон дій для цього типу користувача обмежується ознайомчим переглядом та фільтруванням наявних товарів

Зареєстрований користувач отримує розширений доступ до функціоналу системи після проходження процесу реєстрації та авторизації. Діаграма прецедентів для зареєстрованого користувача, показана на рис. 3.2.

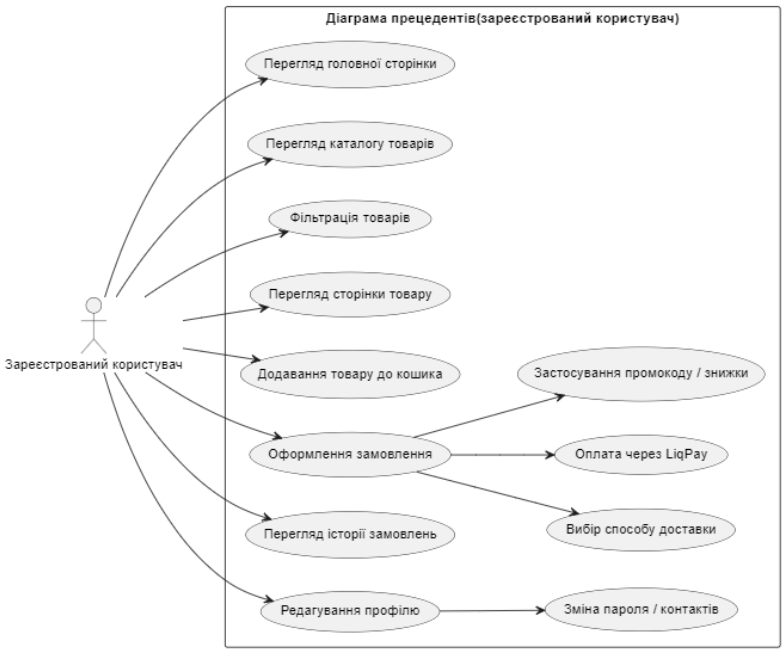

Рисунок 3.2 – Діаграма прецедентів(зареєстрований користувач)

Йому доступні всі можливості гостя, а також персоналізовані сервіси. У своєму особистому кабінеті користувач може переглядати історію попередніх замовлень, змінювати дані профілю (контактну інформацію, пароль, номер телефону). Також він має змогу оформлювати нові замовлення, вибираючи способи доставки, оплачувати товари онлайн через сервіс LiqPay, застосовувати промокоди

або бонусні знижки, отримані в межах маркетингових кампаній. Уся ця діяльність реалізується через зручний і адаптивний інтерфейс, орієнтований на комфортну взаємодію з системою.

Адміністратор є користувачем із максимальними правами доступу, який відповідає за повне функціонування платформи. Діаграма прецедентів показана на рис. 3.3.

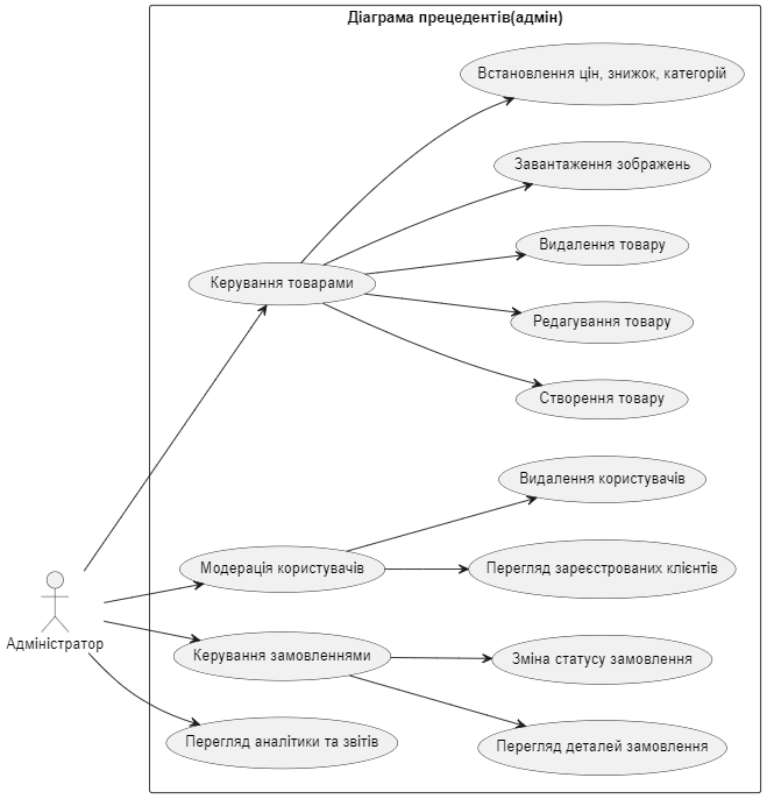

Рисунок 3.3 – Діаграма прецедентів(адмін)

У його повноваження входить керування товарами (створення, редагування, видалення, завантаження зображень, встановлення цін, знижок і категорій), модерація користувачів (перегляд зареєстрованих клієнтів, можливість видалення), обробка та супровід замовлень (зміна статусів, перегляд деталей), а також доступ до аналітичного модуля, де він може переглядати звіти про продажі, кількість активних замовлень, динаміку зростання клієнтської бази тощо.

Побудова окремих діаграм прецедентів для кожної ролі дозволяє структурувати логіку взаємодії, уникнути перетину обов'язків та надати кожному типу користувача індивідуалізований доступ до функцій системи. У наступних підрозділах буде детально представлено PUML-схеми для кожної ролі, які наочно продемонструють обсяг і специфіку доступних дій.

Загалом, діаграма прецедентів стала першим кроком до подальшого глибшого моделювання внутрішніх процесів — зокрема через діаграми послідовностей, активностей, станів і архітектурних компонентів. Такий підхід дозволяє побудувати логічну та масштабовану структуру, орієнтовану на розвиток і гнучке розширення функціоналу платформи.

3.2 Реєстрація клієнта з підтвердженням email: послідовність подій

Діаграма послідовності демонструє кроки взаємодії між користувачем та основними компонентами вебплатформи під час реєстрації нового клієнта з підтвердженням електронної пошти. Цей процес критично важливий для безпеки системи, запобігання створенню фейкових акаунтів і гарантування, що доступ отримує лише підтверджений користувач.

Основні учасники процесу:

- користувач — особа, що ініціює реєстрацію;
- клієнтський інтерфейс (React) — форма на фронтенді, де вводяться дані;
- сервер (Node.js + Express) — обробляє логіку реєстрації;
- сервіс електронної пошти (Gmail + Nodemailer) — надсилає код підтвердження;
- база даних (MongoDB) — тимчасово зберігає код верифікації та після підтвердження створює обліковий запис.

Користувач у клієнтському інтерфейсі (React-компонент CustomerAuthPage.jsx) заповнює реєстраційну форму, де вводить своє ім'я, електронну адресу, телефон (необов'язково) та пароль. Після підтвердження форми клієнт надсилає HTTP-запит методом POST на адресу

/customerAuth/customer/initiate-registration, який потрапляє на сервер Express. Сервер виконує перевірку — чи вже існує користувач із такою електронною адресою у базі MongoDB. Якщо користувача не знайдено, сервер генерує випадковий шестизначний код, хешує пароль за допомогою bcryptjs і формує тимчасовий об'єкт реєстрації, який зберігається у Map pendingRegistrations. Після цього сервер викликає nodemailer, щоб надіслати email з кодом підтвердження на вказану адресу користувача. У відповідь клієнтський інтерфейс показує користувачу інтерфейс введення коду. Діаграма послідовності для процесу реєстрації показана на рис. 3.4.

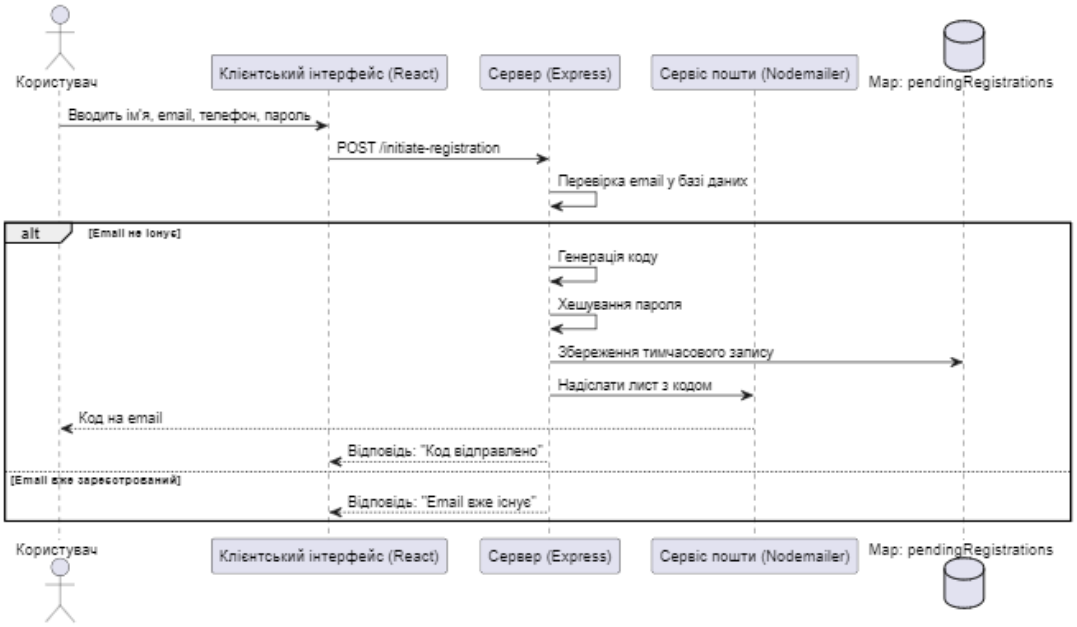

Рисунок 3.4 – Діаграма послідовності для процесу реєстрації

Користувач отримує код підтвердження на свою електронну пошту та вводить його в новій формі. Після цього клієнтський інтерфейс надсилає запит методом POST на маршрут /customerAuth/customer/verify-code, передаючи email і введений код. Сервер зіставляє отриманий код із відповідним записом у Map pendingRegistrations і перевіряє, чи код не протермінований (встановлений термін дії — 10 хвилин). Якщо код дійсний і збігається, сервер створює новий постійний запис у колекції Customer бази MongoDB та видаляє тимчасовий запис з Map.

Діаграма послідовностей для процесу підтвердження електронної пошти показана на рис. 3.5.

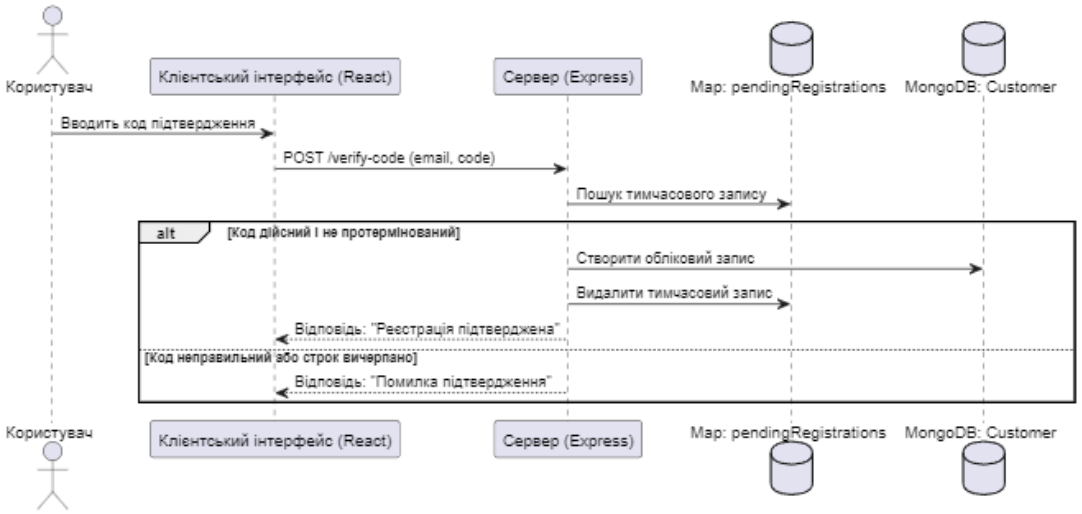

Рисунок 3.5 – Діаграма послідовносте процесу підтвердження електронної пошти

Якщо все пройшло успішно — на клієнт надсилається повідомлення про успішну реєстрацію. У випадку помилки — користувачу повертається відповідне повідомлення: або "Невірний код", або "Термін дії вичерпано".

Для реалізації описаної послідовності було створено компонент CustomerAuthPage.jsx, який керує усім процесом реєстрації покупця: від збору початкових даних до перевірки коду підтвердження. Завдяки логічному розподілу етапів процес побудовано у два кроки:

– спочатку користувач вводить своє ім’я, email, телефон та пароль, після чого система генерує 6-значний код підтвердження і надсилає його на пошту;

– після отримання листа користувач вводить код, який передається на сервер для перевірки. Якщо код дійсний і не протермінований, реєстрація завершується створенням постійного облікового запису в базі MongoDB.

Код, необхідний для реалізації перевірки правильності регістрації наведено на рис. 3.6.

```
try {
  if (isRegister && !isCodeSent) {
    await axios.post("http://localhost:5000/customerAuth/customer/initiate-registration", form);
    setTempEmail(form.email);
    setIsCodeSent(true);
  } else if (isRegister && isCodeSent) {
    await axios.post("http://localhost:5000/customerAuth/customer/verify-code", {
      email: tempEmail,
      code: verificationCode,
    });
    navigate("/customerAuth?justRegistered=true");
  } else {
    const res = await axios.post("http://localhost:5000/customerAuth/customer/login", {
      email: form.email,
      password: form.password,
    });
    localStorage.setItem("customerToken", res.data.token);
    navigate("/");
  }
} catch (err) {
  setError(err.response?.data?.message || "Помилка");
}
};
```

Рисунок 3.6 – Код перевірки правильності реєстрації

Така поетапна реалізація забезпечує надійність системи реєстрації, дозволяє уникнути створення фальшивих облікових записів, сприяє формуванню довіри до платформи, а також дає змогу чітко відслідковувати логіку користувацьких ролей і доступів на наступних етапах.

3.3 Оформлення онлайн-замовлення: логіка користувацької активності

Діаграма активностей (Activity Diagram) відображає етапи взаємодії зареєстрованого користувача з інтерфейсом інтернет-магазину під час створення та підтвердження замовлення. Це один із ключових сценаріїв сайту, оскільки охоплює весь процес покупки — від додавання товарів до кошика до збереження інформації про замовлення в базі даних.

Процес доступний лише після авторизації, яка відбувається за допомогою токена автентифікації (JWT). Оформлення замовлення виконується через інтерактивну сторінку, де користувач бачить список обраних товарів, вводить дані доставки (ПІБ, телефон, місто, відділення), а також підтверджує своє замовлення. Завдяки цьому забезпечується інтуїтивна, безпечна й послідовна взаємодія.

Основна логіка включає такі кроки:

– користувач авторизується у системі (або вже має активний токен);

– переходить на сторінку кошика (/cart), де переглядає додані товари;

– за потреби змінює кількість товарів або видаляє їх;

– натискає кнопку «Оформити замовлення»;

– заповнює форму з контактними та логістичними даними;

– система перевіряє валідність введених полів (використовується валідація на клієнті та сервері);

– після успішної перевірки запит надсилається на сервер (POST /orders/create);

– сервер зберігає замовлення в MongoDB з прив’язкою до користувача;

– користувач бачить повідомлення про успішне замовлення.

На рис. 3.7 зображено діаграму активностей, пов’язаної з оформленням замовлення.

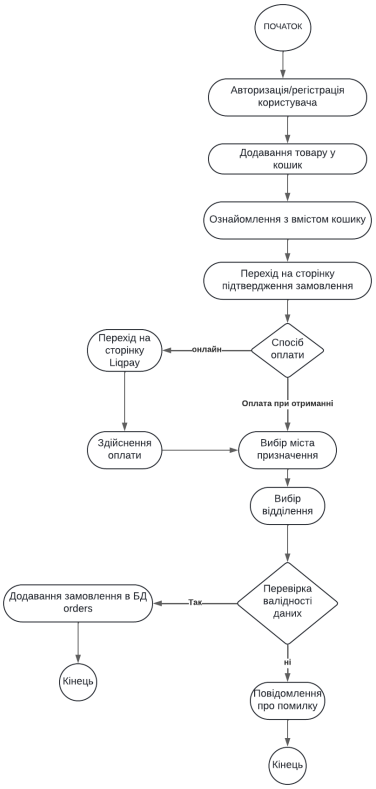

Рисунок 3.7 – Діаграма активностей оформлення замовлення

Компонент CheckoutPage.jsx реалізує повну логіку процесу оформлення замовлення відповідно до діаграми активностей. Його структура охоплює всі основні етапи взаємодії користувача з системою.

На першому етапі виконується отримання даних користувача. Після автентифікації токен зберігається в локальному сховищі браузера , і при завантаженні сторінки виконується запит GET /customerAuth/customer/me. У відповідь система повертає особисті дані користувача, які автоматично підставляються у відповідні поля форми, як показано на рис. 3.8.

```
const handleSubmitOrder = async () => {
  if (!selectedWarehouse || !selectedCity) {
    alert("Оберіть місто та відділення Нової Пошти");
    return;
  }

  try {
    const { data: cartRes } = await axiosInstance.get("/cart", {
      headers: { Authorization: `Bearer ${token}` },
    });

    const items = cartRes.cart.map((item) => ({
      title: item.product.title,
      quantity: item.quantity,
      price: item.product.price,
    }));

    const totalPrice = items.reduce((sum, i) => sum + i.quantity * i.price, 0);
```

Рисунок 3.8 – Метод отримання даних користувача

Далі користувач здійснює вибір способу оплати та доставки. У компоненті реалізована інтерактивна форма, що дозволяє обрати між двома типами оплати: онлайн через LiqPay або готівкою при отриманні. Для доставки використовується інтеграція з API Нової Пошти – користувач обирає місто з підказками та доступне відділення.

Після заповнення усіх необхідних даних натискання кнопки підтвердження ініціює процес збереження замовлення. Спочатку виконується запит до /cart для отримання товарів у кошику. Після формування структури замовлення – надсилається запит POST /orders, у якому міститься уся інформація: контактні дані, перелік товарів, ціна, обраний спосіб оплати та доставки.

У разі вибору онлайн-оплати, компонент забезпечує інтеграцію з платіжною системою LiqPay, як показано на рис. 3.9. Через запит POST /liqpay/create-payment генерується платіжне посилання, після чого користувача автоматично перенаправляє на сторінку оплати.

```
localStorage.setItem(
  "pendingOrder",
  JSON.stringify({
    customerName: customer.name,
    customerEmail: customer.email,
    customerPhone: customer.phone,
    items,
    totalPrice,
    paymentMethod: "online",
    deliveryMethod: "Нова пошта",
    novaPoshtaOffice: selectedWarehouse.label,
    city: selectedCity.label,
    isPaid: true,
  })
);

const { data: paymentData } = await axiosInstance.post(
  "/liqpay/create-payment",
  {
    amount: totalPrice,
    orderInfo: `Замовлення від ${customer.name}`,
    currency: "UAH",
    returnUrl: "http://localhost:3000/checkout/success",
    failUrl: "http://localhost:3000/checkout/fail",
```

Рисунок 3.9 – Метод інтеграції з платіжною системою

Весь процес супроводжується перевірками та обробкою помилок. Якщо користувач не заповнив обов’язкові поля або виникає помилка на сервері, система повідомляє про це через інтерфейс. Також реалізована блокування повторного відправлення форми під час очікування відповіді.

3.4 Структура бізнес-об'єктів: товари, замовлення, клієнти

Діаграма класів (англ. Class Diagram) є ключовим інструментом моделювання об’єктної структури вебзастосунку. Вона дозволяє наочно зобразити основні сутності системи, їх атрибути та зв’язки між об'єктами. Такий підхід є критично важливим на етапі проєктування, оскільки дозволяє зрозуміти взаємодію компонентів ще до створення бази даних або REST API.

У межах розробки клієнтської платформи інтернет-магазину була побудована узагальнена діаграма класів, яка охоплює шість ключових сутностей: Клієнт (Customer), Товар (Product), Кошик (Cart), Замовлення (Order), Подарункова

картка (GiftCard) та Знижка (Discount). Ці класи утворюють каркас системи, на основі якого побудовано базу даних та логіку взаємодії.

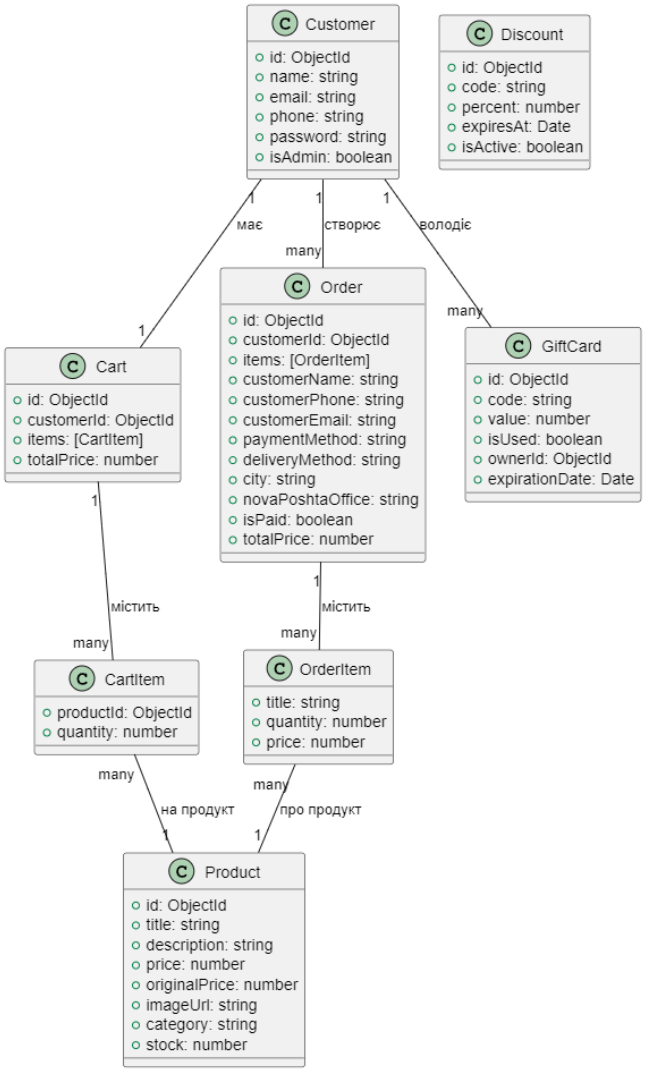

Рисунок 3.10 – Графічне зображення структури і зв'язків БД

Структура системи включає:

– customer – має атрибути: id, name, email, phone, password, isAdmin. Один клієнт може мати кілька замовлень, один активний кошик і бути власником подарункових карток.

– product – містить поля: id, title, description, price, originalPrice, imageUrl, category, stock. Продукти можуть бути частиною багатьох замовлень або додані до кошика клієнта.

– cart – є перехідною сутністю між клієнтом і товарами. Містить список продуктів з кількістю, загальною сумою та прив'язкою до customerId

– order – зберігає дані про підтверджене замовлення: items, customerName, paymentMethod, deliveryMethod, city, novaPoshtaOffice, isPaid, totalPrice. Пов’язаний з клієнтом та може містити кілька товарів.

– giftCard – представляє подарункові сертифікати, прив'язані до клієнта. Містить: code, value, isUsed, ownerId, expirationDate.

– discount – містить інформацію про знижки, доступні для продуктів чи категорій. Має: code, percent, expiresAt, isActive.

Для прикладу покажемо приклад моделі Customer рис. 3.11 та модель Order рис. 3.12.

```
const mongoose = require("mongoose");

const customerSchema = new mongoose.Schema(
  {
    name: { type: String, required: true },
    email: { type: String, required: true, unique: true },
    phone: String,
    password: { type: String, required: true },
    isAdmin: { type: Boolean, default: false },
    cart: [
      {
        productId: { type: mongoose.Schema.Types.ObjectId, ref: "Product" },
        quantity: { type: Number, default: 1 },
      },
    ],
  },
  { timestamps: true }
);


module.exports = mongoose.model("Customer", customerSchema);
```

Рисунок 3.11 – Модель Customer

Модель Customer описує основну сутність користувача — покупця інтернет-магазину. Вона містить обов’язкові поля name, email, password, а також додаткову інформацію: phone і прапорець isAdmin для розмежування доступу. Кожен клієнт має масив cart, у якому зберігаються тимчасово додані товари. Це дозволяє уникнути створення окремої колекції для кошика та пришвидшує операції з ним. Зв’язок із товарами реалізовано через поле productId з типом ObjectId.

```
const mongoose = require("mongoose");

const orderSchema = new mongoose.Schema(
  {
    customerName: { type: String, required: true },
    customerEmail: String,
    customerPhone: String,
    items: [
      {
        title: String,
        quantity: Number,
        price: Number,
      },
    ],
    totalPrice: { type: Number, required: true },
    status: {
      type: String,
      enum: ["Очікує", "Активне", "Виконано"],
      default: "Очікує",
    },
    paymentMethod: {
      type: String,
      enum: ["online", "cash"], // узгоджено з frontend значеннями
      required: true,
    },
    deliveryMethod: {
      type: String,
      enum: ["Нова пошта"],
      default: "Нова пошта",
```

а)

```
    },
    novaPoshtaOffice: String, // Назва/адреса відділенн
    address: String, // Не обов'язкове, залишаємо для м
    city: { type: String },
    paid: {
    type: Boolean,
    default: false,
  },
  },
  { timestamps: true }
);

module.exports = mongoose.model("Order", orderSchema);
```

б)

Рисунок 3.12 – Модель Order(а, б)

Модель Order представляє оформлене замовлення клієнта. Вона містить інформацію про користувача (customerName, customerEmail, customerPhone), список замовлених товарів (items), загальну вартість (totalPrice) і деталі доставки. Підтримується оплата як онлайн, так і при отриманні, що задається через поле paymentMethod. Також є статус виконання (status), логістичні поля (city, novaPoshtaOffice, address) та прапорець paid. Усі замовлення автоматично фіксують дату створення й оновлення через timestamps.

3.5 Архітектура застосунку: модулі фронтенду, бекенду та інтеграцій

Щоб краще усвідомити архітектуру вебзастосунку, було побудовано діаграму компонентів (англ. Component Diagram), яка відображає основні складові системи, зв’язки між ними та потоки даних. Такий формат представлення допомагає чітко окреслити логічну структуру проєкту, спрощує його розробку, масштабування та подальший супровід.

У нашому інтернет-магазині реалізована модель взаємодії за архітектурою клієнт–сервер із використанням підходу SPA (Single Page Application). Основні компоненти включають:

Клієнтську частину (Frontend) – реалізована на React. Вона відповідає за візуальне відображення інтерфейсу, взаємодію з користувачем, маршрутизацію сторінок (товари, кошик, оформлення замовлення, профіль), а також надсилання запитів на сервер через HTTP. Усі ініціативи (додати до кошика, увійти, підтвердити замовлення) надходять саме з цього рівня.

Серверну частину (Backend API) – створена на Node.js із застосуванням Express. Вона приймає та обробляє запити з фронтенду, перевіряє авторизацію, взаємодіє з базою даних, генерує коди підтвердження, створює замовлення, надсилає повідомлення та виступає посередником між клієнтом і зовнішніми сервісами (наприклад, LiqPay або Nova Poshta API).

Базу даних (MongoDB) – використовується для зберігання всіх ключових сутностей: користувачів (customers), замовлень, товарів, знижок, подарункових сертифікатів тощо. Дані організовані у вигляді колекцій із застосуванням бібліотеки Mongoose для опису схем.

Зовнішні сервіси:

– email-сервер (SMTP через nodemailer) — забезпечує надсилання листів із кодами підтвердження під час реєстрації;

– JWT-механізм — використовується для створення токенів автентифікації, які передаються в заголовках HTTP-запитів;

– платіжна система LiqPay — інтегрована для здійснення онлайн-оплати безпосередньо з вебінтерфейсу;

– API Нової Пошти — використовується для динамічного вибору міста та відділення під час оформлення доставки.

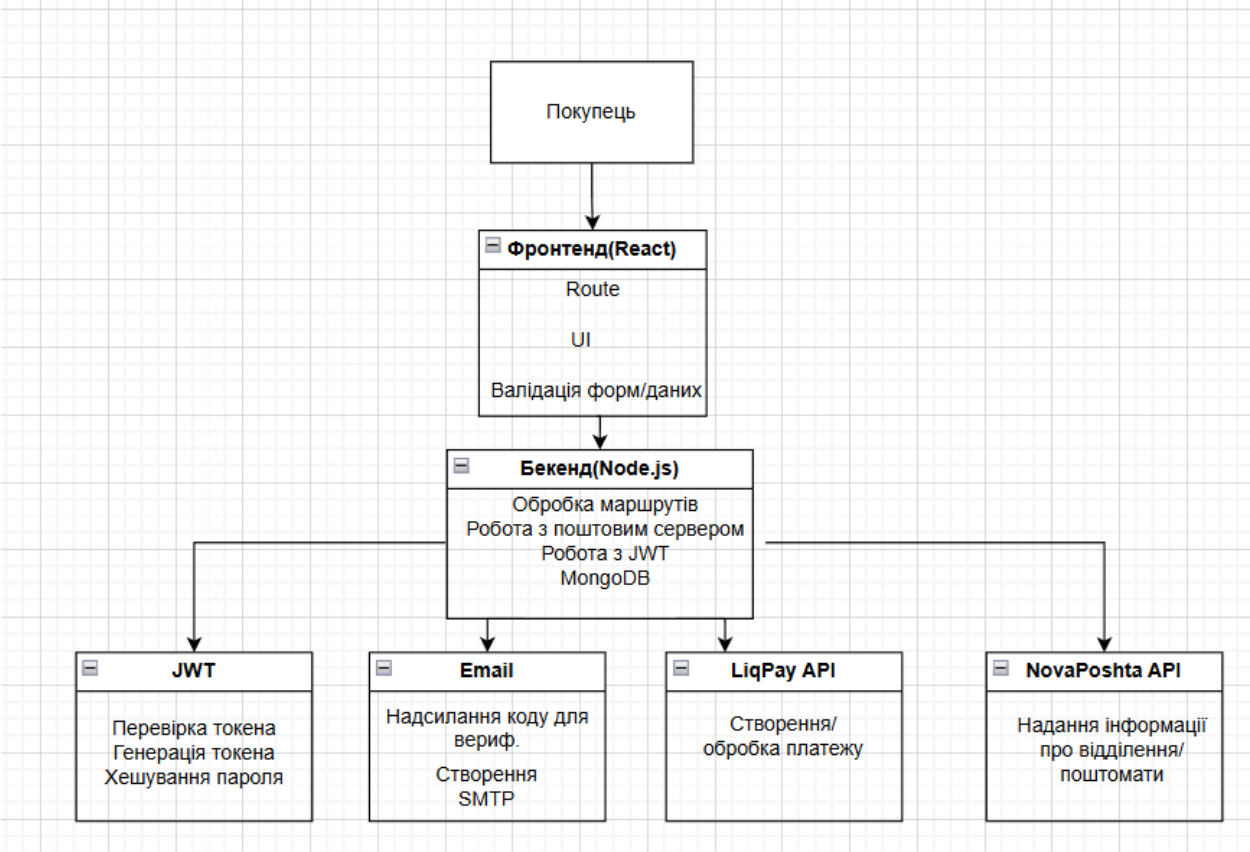

Рисунок 3.13 - Діаграма компонентів платформи

Такий розподіл компонентів дає змогу забезпечити масштабованість, розділення обов’язків, гнучкість у розширенні функціоналу (наприклад, додавання аналітики, підтримки подарункових карток, адміністрування тощо) без негативного впливу на решту системи. Ця архітектура також полегшує тестування, відлагодження та майбутній супровід.

Висновки до розділу 3

У цьому розділі було детально розглянуто процес реалізації ключових компонентів вебзастосунку, що забезпечують повноцінну взаємодію між користувачем, інтерфейсом, серверною логікою та базою даних.

Розроблено функціонал реєстрації користувача з підтвердженням електронної пошти, що підвищує безпеку та запобігає створенню фейкових облікових записів. Запроваджено механізм обробки JWT-токенів для авторизації та захисту доступу до приватних маршрутів. Також реалізовано інтерфейс оформлення замовлення, який включає вибір способу доставки, оплату онлайн (інтеграція з LiqPay) або при отриманні, що відповідає сучасним вимогам до e-commerce платформ.

Було створено діаграми послідовності, активностей, класів і компонентів, що допомогли структуровано представити логіку роботи системи, взаємозв’язки між об’єктами та залежності між модулями.

Загалом, розділ демонструє комплексний підхід до реалізації вебзастосунку: від клієнтської логіки до API та зовнішніх сервісів, забезпечуючи масштабованість, безпеку та зручність користувача.

4 РЕАЛІЗАЦІЯ ІНТЕГРОВАНОЇ ПЛАТФОРМИ ЕЛЕКТРОННОЇ КОМЕРЦІЇ ТА ЇЇ ТЕСТУВАННЯ

4.1 Розробка клієнтської частини платформи

Клієнтська частина розробленого вебзастосунку створена з використанням бібліотеки React, яка забезпечує гнучку побудову інтерфейсу користувача на основі компонентного підходу. Завдяки React вдалося реалізувати адаптивний, інтерактивний та розширюваний UI, що ефективно реагує на дії користувачів та зміни стану програми.

Клієнтська частина вебзастосунку реалізована у вигляді односторінкового застосунку (SPA). Основною технологією обрано бібліотеку React, що дозволяє створити гнучкий інтерфейс, який динамічно оновлюється без перезавантаження сторінки. Уся логіка маршрутизації, компонування сторінок і завантаження даних реалізована на фронтенді і показана на рис. 4.1.

Кореневий компонент App.js виступає як "контейнер маршрутизації" — саме тут відбувається підключення основного навігаційного компонента, визначення доступних шляхів через react-router-dom, і підключення глобального стилю або компонента Header. Початкове завантаження сторінки обмежується мінімальним HTML, а далі всі переходи між сторінками реалізуються динамічно, без звернення до сервера для кожної нової сторінки.

Файлова структура клієнта логічно розділена на папки: pages, components, styles, lib, routes. Це дає змогу ізолювати бізнес-логіку, візуальні компоненти та маршрути окремо, що позитивно впливає на читабельність і масштабованість.

```
function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/auth" element={<LoginPage />} />
        <Route path="/register" element={<RegisterPage />} />
        <Route element={<Layout />}>
          <Route path="/dashboard" element={<DashboardPage />} />
          <Route path="/add-product" element={<AddProductPage />} />
          <Route path="/products" element={<AllProductsPage />} />
          <Route path="/edit-product/:id" element={<EditProductPage />} />
          <Route path="/products/inventory" element={<InventoryPage />} />
          <Route path="/customers" element={<CustomersPage />} />
          <Route path="/customers/add" element={<AddCustomerPage />} />
          <Route path="/discounts" element={<DiscountsPage />} />
          <Route path="/discounts/create" element={<CreateDiscountPage />} />
          <Route path="/orders" element={<OrdersPage />} />
          <Route path="/orders/drafts" element={<CreateOrderDraftPage />} />
          <Route path="/orders/abandoned" element={<AbandonedOrdersPage />} />
          <Route path="/products/gift-cards" element={<GiftCardCreatePage />} />
          <Route path="/gift-cards/send" element={<GiftCardSend />} />
        </Route>
        <Route path="/" element={<StorePage />} />
        <Route path="/customerAuth" element={<CustomerAuthPage />} />
        <Route path = "/cart" element={<CartPage />} />
        <Route path="/checkout" element = {<CheckoutPage/>} />
        <Route path ="/checkout/success" element = {<CheckoutSuccessPage/>} />
```

Рисунок 4.1 – Реалізація маршрутизації на фронтенді

Клієнтський інтерфейс вебзастосунку побудований на основі чіткої маршрутизації за допомогою бібліотеки react-router-dom. Це дозволяє формувати логіку переходів між сторінками залежно від URL без перезавантаження, що є ключовою перевагою архітектури SPA.

Усі основні сторінки винесені в окремі React-компоненти, розміщені в директорії src/pages/. Зокрема, кожна сторінка відповідає за окремий аспект користувацької взаємодії:

– storePage.jsx – відображення товарів;
– cartPage.jsx – робота з кошиком;
– checkoutPage.jsx – оформлення замовлення;
– customerAuthPage.jsx – автентифікація клієнтів;
– customerProfile.jsx – перегляд особистих даних користувача.

Окрема увага в клієнтській частині була приділена процесу автентифікації користувачів. Цей функціонал реалізовано у компоненті CustomerAuthPage.jsx, який охоплює як вхід, так і реєстрацію з підтвердженням електронної пошти через код.

Реалізація автентифікації у клієнтській частині вебзастосунку побудована на основі дворівневого підходу: спочатку реєстрація, а потім підтвердження email-

користувача за допомогою унікального коду. Це дозволяє забезпечити більшу надійність і захист від несанкціонованих реєстрацій.

На першому етапі — реєстрації — користувач заповнює форму, вводячи своє ім’я, електронну пошту, номер телефону та пароль. Після підтвердження введених даних через інтерфейс клієнта (React-компонент CustomerAuthPage.jsx), відправляється HTTP-запит методом POST на маршрут /customerAuth/customer/initiate-registration. Цей запит передає введені дані на сервер. У відповідь бекенд перевіряє, чи не зайнята вказана електронна адреса, і якщо ні — генерує випадковий шестизначний код. Даний код надсилається на пошту користувача за допомогою сервісу Nodemailer.

Другий етап — підтвердження коду. Користувач отримує лист із кодом на вказану адресу і вводить цей код у відповідне поле форми. Після цього з клієнта відправляється новий HTTP-запит методом POST на маршрут /customerAuth/customer/verify-code, у якому передається email та код. Якщо код збігається з тим, що було збережено на сервері у тимчасовому сховищі, і він ще не протермінований — створюється повноцінний обліковий запис у базі даних MongoDB. Після підтвердження користувач може здійснити вхід у систему.

Метод, який обробляє дані та відправляє код, показано на рис. 4.2.

```
try {
  if (isRegister && !isCodeSent) {
    await axios.post("http://localhost:5000/customerAuth/customer/initiate-registration", form);
    setTempEmail(form.email);
    setIsCodeSent(true);
  } else if (isRegister && isCodeSent) {
    await axios.post("http://localhost:5000/customerAuth/customer/verify-code", {
      email: tempEmail,
      code: verificationCode,
    });
    navigate("/customerAuth?justRegistered=true");
```

Рисунок 4.2 – Метод для афтентифікації

Для входу в систему використовується стандартна форма авторизації (email + пароль). Після успішного входу клієнтський токен зберігається в localStorage, що дозволяє здійснювати подальші запити з авторизацією.

Форма має зручний вигляд, динамічно змінює свій стан (реєстрація/підтвердження/вхід), відображає помилки валідації або відповіді від сервера.

Завдяки цій реалізації система захищена від фіктивних email-реєстрацій, а також має гнучкий механізм керування сесіями користувача.

У клієнтській частині вебзастосунку реалізовано повноцінний функціонал додавання товарів до кошика, редагування його вмісту та подальшого оформлення замовлення. Основна логіка сконцентрована у React-компонентах CartPage.jsx та CheckoutPage.jsx.

```
useEffect(() => {
  if (!token) return navigate("/auth");

  axiosInstance
    .get("/cart", {
      headers: { Authorization: `Bearer ${token}` },
    })
    .then((res) => setCartItems(res.data.cart))
    .catch(() => alert("Не вдалося завантажити кошик"));
}, [token, navigate]);
```

Рисунок 4.3 – Метод для відображення товарів кошика

На сторінці кошика (CartPage.jsx) відображаються всі товари, які користувач попередньо додав до корзини. Метод, який забезпечує відображення товарів кошика, показано на рис. 4.3. Для кожного товару можна змінити кількість або повністю видалити його. Дані про кошик зберігаються на сервері в об’єкті, прив’язаному до конкретного користувача, що дозволяє зберігати інформацію навіть після оновлення сторінки або зміни пристрою. Обмін даними між клієнтом і сервером відбувається за допомогою axios через маршрути /cart, /cart/add, /cart/remove, тощо.

При переході до оформлення замовлення (CheckoutPage.jsx) користувачу пропонується вибрати спосіб оплати (онлайн або при отриманні) та спосіб доставки (через інтеграцію з API «Нової Пошти» – вибір міста та відділення). Якщо вибрано

онлайн-оплату, то через маршрут /liqpay/create-payment ініціюється взаємодія з LiqPay API, і користувач перенаправляється на сторінку оплати. Усі введені дані про доставку та товари зберігаються у вигляді замовлення на сервері через запит POST /orders.

Після завершення оформлення клієнту відображається сповіщення про успішну операцію, метод показано на рис. 4.4, а у випадку вибору оплати при отриманні — замовлення зберігається без етапу онлайн-оплати.

```
useEffect(() => {
  const createOrderFromStorage = async () => {
    const stored = localStorage.getItem("pendingOrder");
    if (!stored) return;

    try {
      const orderData = JSON.parse(stored);

      await axios.post("http://localhost:5000/orders", orderData, {
        headers: { Authorization: `Bearer ${token}` },
      });

      await axios.post("http://localhost:5000/cart/clear", null, {
        headers: { Authorization: `Bearer ${token}` },
      });

      localStorage.removeItem("pendingOrder");
    } catch (err) {
      console.error("❌ Помилка створення замовлення після оплати:", err);
      alert("Замовлення не було збережене. Зверніться до підтримки.");
    }
  };
```

Рисунок 4.4 – Метод успішного створення замовлення у разі оплати

Реалізація персоналізованого інтерфейсу в системі досягається завдяки гнучкому керуванню ролями користувачів. Усі клієнти системи мають базовий профіль, однак при потребі їм можуть надаватись додаткові можливості, наприклад, права адміністратора.

Після успішної автентифікації дані про поточного користувача зчитуються з сервера (запит /customerAuth/customer/me) та зберігаються у стані за допомогою хуку useState. На основі отриманої інформації, зокрема властивості isAdmin, в

інтерфейсі автоматично оновлюється доступний функціонал. Наприклад, користувач-адміністратор бачить посилання до адмін-панелі, а звичайний клієнт – лише вкладки "Профіль", "Кошик", "Замовлення" тощо.

Перевірка ролі здійснюється безпосередньо в компонентах через умовний рендеринг. Це забезпечує безпечне та надійне відображення лише дозволених дій та інформації, що зменшує ризики несанкціонованого доступу.

4.2 Розробка серверної частини платформи

При створенні серверної частини нашого вебсайту ми дотримувалися архітектурного шаблону MVC (Model-View-Controller). Цей підхід дозволяє ефективно організувати код, розділити логіку, відповідальність та забезпечити масштабованість проєкту.

У контексті серверної частини сайту, ми працюємо з такими ключовими компонентами:

– models (Моделі) — описують структуру даних, які зберігаються в базі даних MongoDB;

– controllers (Контролери) — містять бізнес-логіку та обробляють запити, які надходять із клієнтської частини;

– routes (Маршрути) — визначають, які контролери викликати при надходженні запитів за певними адресами;

– middleware (Проміжні обробники) — реалізують додаткову логіку, таку як авторизація, перевірка прав доступу, обробка помилок;

– config (Конфігурація) — підключення до бази даних та налаштування середовища.

Першим етапом розробки серверної частини було створення та налаштування Node.js-проєкту. Ми використовували середовище Node.js разом з фреймворком Express.js, який спрощує створення веб-сервера, та бібліотекою Mongoose, яка забезпечує взаємодію з MongoDB.

Файл server.js — це основна точка входу до нашої серверної частини. Його головне завдання — запустити сервер, налаштувати базові middleware, підключити базу даних MongoDB та активувати маршрути для взаємодії з API. Вигляд server.js показано на рис. 4.5.

```
const mongoose = require('mongoose');
const express = require('express');
const cors = require('cors');
const path = require('path');
require('dotenv').config();

const connectDB = require('./config/db');
connectDB();

const app = express();
const PORT = process.env.PORT || 5000;

app.use(cors({
  origin: 'http://localhost:3000',
  credentials: true
}));
app.use(express.json());

const cookieParser = require('cookie-parser');
app.use(cookieParser());
app.use('/api/auth', require('./routes/auth'));
app.use('/products', require('./routes/products'));
app.use('/customers', require('./routes/customers'));
app.use('/discounts', require('./routes/discounts'));
app.use('/orders', require('./routes/orders'));
app.use('/gift-cards', require('./routes/giftCards'));
app.use('/store-settings', require('./routes/storeSettings'));
app.use('/customerAuth/customer', require('./routes/customerAuth'));
```

Рисунок 4.5 – Вигляд файлу server.js

Ретельне структурування server.js — це запорука масштабованості та підтримуваності всього застосунку.

Для збереження даних нашого вебсайту ми обрали базу даних MongoDB, яка є документоорієнтованою і не потребує жорсткої структури таблиць, як у традиційних реляційних системах. Це дозволяє швидко зберігати, змінювати та масштабувати інформацію, яка надходить від користувачів або формується в процесі роботи сайту. З MongoDB зручно працювати у зв’язці з Node.js, особливо якщо використовується бібліотека Mongoose. Саме її ми використали для встановлення зв’язку з базою даних, організації моделей та виконання запитів.

Підключення до MongoDB реалізовано через окремий модуль config/db.js, який ми створили для того, щоб ізолювати логіку підключення від основного коду сервера. Це робить проєкт структурованішим і легшим у підтримці. У цьому файлі

використовується асинхронна функція, яка виконує підключення до бази через mongoose та виводить повідомлення про успішне з'єднання або помилку. Задля безпеки рядок підключення не прописується прямо в коді — він зберігається у .env файлі, а зчитується за допомогою бібліотеки dotenv. Такий підхід дозволяє уникати публікації конфіденційних даних у відкритих репозиторіях.

На рис. 4.6. наведено, те, як виглядає метод підключення до БД.

```
const mongoose = require('mongoose');

const connectDB = async () => {
  try {
    const conn = await mongoose.connect(process.env.MONGO_URI);


    console.log(`MongoDB підключено: ${conn.connection.host}`);
  } catch (error) {
    console.error(`Помилка підключення: ${error.message}`);
    process.exit(1);
  }
};

module.exports = connectDB;
```

Рисунок 4.6 – Метод для підключення до БД

Сам MongoDB зберігає дані у вигляді документів, які за своєю структурою дуже схожі на звичайні JSON-об’єкти. Наприклад, дані про товар можуть зберігатися у форматі, де кожен документ містить поля з назвою товару, його ціною, наявністю, категорією тощо. Завдяки цій гнучкості, ми можемо легко адаптувати структуру бази під нові вимоги, не змінюючи саму логіку або не створюючи складні міграції, як це часто буває у реляційних базах.

Таким чином, підключення до MongoDB стало надійною основою для подальшої розробки функціоналу, пов’язаного зі збереженням замовлень, користувачів, товарів, налаштувань магазину та іншої важливої інформації.

Після того, як сервер та база даних були успішно налаштовані, наступним кроком стало створення логіки, яка дозволяє обробляти запити користувачів до серверної частини сайту. Цей процес складається з двох важливих етапів:

визначення маршрутів і реалізація контролерів, які відповідають за виконання конкретних дій.

У вебсайті кожна дія, що виконується через інтерфейс — наприклад, перегляд товарів, оформлення замовлення чи авторизація — супроводжується HTTP-запитом до певної адреси. Ці адреси і є маршрутами. Для їхньої обробки ми використовували Express.js, який дозволяє створювати чисті та логічно структуровані API.

Кожен файл із маршрутами у проєкті відповідає за окремий функціональний блок. Наприклад, всі запити, пов’язані з продуктами, описані у файлі routes/products.js. У ньому ми лише вказуємо, який запит має викликати яку функцію з контролера. Сама логіка того, що має відбуватись у відповідь на запит, винесена у відповідні контролери — окремі файли з логікою. На рис. 4.7 наведено метод, який обробляє запит на отримання списку всіх товарів.

```
router.get("/", getAllProducts);
```

Рисунок 4.7 – Метод оброблення запиту на отримання списку товарів

Як видно з цього прикладу, ми імпортуємо функцію getAllProducts із контролера продуктів і прив’язуємо її до HTTP-запиту GET на маршрут /. Коли клієнт надсилає такий запит, сервер викликає відповідну функцію з контролера, яка виконує всю потрібну роботу — звертається до бази, отримує дані, обробляє їх і відправляє відповідь.

Контролери — це своєрідні «мозки» застосунку. У них зосереджена основна логіка: як обробити дані, які поля вибрати, чи потрібно авторизувати користувача, і як виглядатиме відповідь. Вони також взаємодіють з моделями — структурами, які ми визначили раніше через Mongoose.

Вигляд функції getAllProducts в контроллері показано на рис. 4.8.

```
const getAllProducts = async (req, res) => {
  try {
    const products = await Product.find().sort({ createdAt: -1 });
    res.status(200).json(products);
  } catch (err) {
    console.error(err);
    res.status(500).json({ message: "Помилка при завантаженні товарів" });
  }
};
```

Рисунок 4.8 – Контроллер для обробки GET-запиту

Тут видно, що за допомогою методу Product.find() ми отримуємо всі товари з колекції, після чого надсилаємо їх у відповідь клієнту. Якщо під час виконання сталася помилка — ми повертаємо статус 500 і повідомлення про невдачу. Така логіка може бути набагато складнішою — наприклад, із перевіркою прав доступу, фільтрами або посторінковим виведенням.

Перевага такої побудови в тому, що маршрути залишаються максимально «чистими», а вся логіка обробки даних відокремлена у спеціальних функціях, які легко підтримувати, тестувати та масштабувати. Вони не тільки зменшують дублювання коду, а й дозволяють швидко знаходити проблеми або додавати нову функціональність.

Таким чином, маршрути та контролери формують основу взаємодії клієнта з сервером. Коли користувач клікає по кнопці на сайті або надсилає форму, браузер відправляє запит на один із маршрутів, де вже контролер вирішує, що потрібно зробити — і все це відбувається буквально за частки секунди.

Для збереження та організації інформації на сервері ми використовуємо базу даних MongoDB, яка дозволяє працювати з даними у формі документів. Однак для того, щоб ці документи мали чітко визначену структуру і відповідали логіці нашого застосунку, ми застосували бібліотеку Mongoose. Вона дозволяє створювати так звані моделі — своєрідні шаблони, що описують, які саме поля має містити кожен документ у базі, якого вони мають бути типу, чи є обов’язковими тощо.

Кожна така модель відповідає окремій сутності в нашій системі. Наприклад, для товарів ми створили модель Product, для замовлень — Order, для користувачів — User. Кожна з моделей описана у відповідному файлі в директорії models.

Уявімо, що потрібно зберігати інформацію про товар. Він має назву, опис, ціну, категорію, наявність на складі, а також посилання на зображення. Усе це ми визначаємо у моделі, яка створюється за допомогою методу mongoose.Schema. Приклад моделі товару показано на рис. 4.9:

```
const mongoose = require("mongoose");

const productSchema = new mongoose.Schema({
  title: { type: String, required: true },
  description: String,
  price: { type: Number, required: true },
  originalPrice: Number,
  quantity: { type: Number, default: 0 },
  imageUrl: String,
}, { timestamps: true });

module.exports = mongoose.model("Product", productSchema);
```

Рисунок 4.9 – Модель товару

У цьому прикладі ми вказали, що кожен товар обов’язково повинен мати назву та ціну, а також може мати статус наявності (за замовчуванням — так) і зображення. Коли сервер отримує запит на створення нового товару, він буде перевіряти, чи відповідають дані цій моделі. Якщо, наприклад, не вказано ціну або передано рядок замість числа — запит буде відхилено.

Моделі не лише задають структуру документів, а й надають зручні методи для роботи з базою даних. Ми можемо легко створювати нові об’єкти, зберігати їх у базі, оновлювати, шукати за фільтрами або видаляти. Усе це виконується з використанням методів, які надає Mongoose, як-от find(), save(), updateOne() тощо.

Завдяки використанню моделей ми досягаємо важливого ефекту: наші дані стають передбачуваними. Кожна сутність має чітку структуру, а отже — ми

можемо будувати надійну логіку, не переймаючись тим, що користувач або інша частина системи відправить щось несподіване чи неправильне.

У нашому проєкті моделей досить багато: ми створили окремі для клієнтів, замовлень, подарункових сертифікатів, знижок, налаштувань магазину та інших елементів. Це дозволяє підтримувати чисту архітектуру, де кожна частина системи працює з чітко визначеними структурами і не заважає іншим.

Отже, моделі в MongoDB через Mongoose — це не просто формальність, а необхідний інструмент для того, щоб гарантувати стабільність, надійність і логічність роботи з даними. Вони лежать в основі усіх операцій, пов’язаних із базою, і є критично важливими для роботи будь-якого сучасного Node.js-застосунку.

На завершальному етапі створення серверної частини вебсайту важливо не лише реалізувати базову функціональність, а й забезпечити її стабільність, безпеку та здатність працювати з зовнішніми сервісами. У нашому проєкті ми реалізували кілька важливих інтеграцій, які дозволяють системі працювати як повноцінний онлайн-магазин. Серед них — підключення платіжного сервісу LiqPay, організація доставки через API «Нової Пошти», підтримка завантаження зображень та підготовка до деплою з використанням Docker.

Однією з ключових інтеграцій є підключення платіжного шлюзу LiqPay. Для цього ми створили окремі маршрути та контролери, які відповідають за генерацію платіжних форм, обробку відповіді від LiqPay та прийом webhook-сповіщень про статус транзакції. Це дає змогу покупцеві оплачувати замовлення онлайн, а магазину — отримувати підтвердження в автоматичному режимі. У файлі liqpayRoutes.js реалізовано маршрут, який на запит клієнта формує платіжну сторінку на основі переданих параметрів. Окремо, в контролері liqpayWebhookController.js, ми реалізували обробку callback-запитів, які надсилає LiqPay після кожної транзакції. Вигляд методу для підключення LiqPay API показано на рис. 4.10.

```
const LIQPAY_PRIVATE_KEY = process.env.LIQPAY_PRIVATE_KEY;
function generateSignature(data) {
  const shasum = crypto.createHash("sha1");
  shasum.update(LIQPAY_PRIVATE_KEY + data + LIQPAY_PRIVATE_KEY);
  const sha1Result = shasum.digest("hex");
  return Buffer.from(sha1Result, "hex").toString("base64");
}
const createPayment = (req, res) => {
  const { amount, orderInfo, currency, returnUrl, failUrl } = req.body;

  const paymentData = {
    public_key: LIQPAY_PUBLIC_KEY,
    version: "3",
    action: "pay",
    amount,
    currency,
    description: orderInfo,
    order_id: Date.now().toString(),
    sandbox: 1,
    result_url: returnUrl,
    server_url: failUrl,
  };

  const jsonStr = JSON.stringify(paymentData);
  const data = Buffer.from(jsonStr).toString("base64");
  const signature = generateSignature(data);

  res.json({ data, signature, paymentUrl: `https://www.liqpay.ua/api/3/checkout?data=${encodeURIComponent(data)}&signature=${encodeURIComponent(sign
```

Рисунок 4.10 – Метод для підключення та роботи з LiqPay

Ще однією важливою функцією є можливість оформлення доставки. Ми реалізували інтеграцію з API «Нової Пошти», що дозволяє автоматично отримувати перелік відділень, підтверджувати доставку та формувати ТТН без необхідності ручного введення. Це реалізовано через модуль novaPostController.js, а логіка взаємодії з API винесена у відповідний маршрут. Вигляд методу для тримання поштових відділень Нової пошти, відповідно до обраного міста показано на рис. 4.11.

```
const getWarehouses = async (req, res) => {
  const { cityRef } = req.query;

  try {
    const response = await axios.post("https://api.novaposhta.ua/v2.0/json/", {
      apiKey: process.env.NOVAPOSHTA_API_KEY,
      modelName: "AddressGeneral",
      calledMethod: "getWarehouses",
      methodProperties: {
        SettlementRef: cityRef, // <--- ключовий момент
        Page: "1",
        Limit: "50",
        Language: "UA",
      },
    });

    const warehouses = response.data.data.map((wh) => ({
      label: wh.Description,
      value: wh.Ref,
    }));

    res.json(warehouses);
  } catch (err) {
    console.error("Помилка отримання відділень:", err.message);
    res.status(500).json({ message: "Не вдалося знайти відділення" });
  }
};
```

Рисунок 4.11 – Метод отримання поштових відділень з NovaPoshtaAPI

Таким чином, інтеграція сторонніх сервісів і продумана архітектура забезпечують повноцінну роботу інтернет-магазину. Кожна реалізована можливість — від прийому оплат до автоматизації доставки — робить систему більш зручною для користувача і готовою до реального використання.

Завершуючи опис розробки серверної частини, можна впевнено сказати, що вона є надійним технологічним фундаментом усього вебсайту. Завдяки поєднанню сучасних інструментів — Node.js, Express, MongoDB і зовнішніх API — нам вдалося реалізувати систему, яка не лише ефективно обробляє дані, але й легко масштабується, адаптується до змін та інтегрується з ключовими бізнес-сервісами. Уся серверна архітектура побудована з урахуванням реальних потреб онлайн-магазину, забезпечуючи стабільну роботу, безпеку даних і зручну взаємодію з користувачем.

4.3 Ключові функції платформи: робота з товарами, клієнтом та транзакціями

Будь-який сучасний вебсайт, який має справу з персональними замовленнями, доступом до кабінету або історією покупок, вимагає реалізації системи авторизації. У проєкті ShopManager це питання вирішене шляхом створення повноцінного модуля автентифікації, який включає в себе як серверну логіку реєстрації нового користувача, так і механізм входу, перевірки токена, а також захисту закритих маршрутів. Усе це побудовано на основі стандартного підходу з використанням JWT (JSON Web Token), який сьогодні є одним із найпоширеніших і найзручніших способів захисту API.

На рівні маршрутизації за авторизацію відповідають два окремі файли — auth.js та customerAuth.js. Перший переважно обробляє загальні запити для входу або реєстрації адміністративних користувачів, тоді як другий зосереджений на клієнтах — звичайних користувачах інтернет-магазину. Обидва маршрути використовують логіку, яка описана в контролері authController.js, де реалізовано перевірку облікових даних, генерацію токена та повернення відповіді.

JWT-токени створюються після успішної авторизації користувача. У токен записується його унікальний ідентифікатор, і цей токен передається клієнту. Надалі кожен запит до захищених маршрутів має містити цей токен у заголовках. На сервері є middleware — спеціальна проміжна функція, яка перевіряє, чи є токен у запиті, і чи дійсний він. Якщо все в порядку, запит проходить далі, і сервер знає, хто саме надіслав його. Вигляд коду, який відповідає за створення токена показано на рис. 4.12.

```
const token = jwt.sign(
  { userId: user._id, username: user.username },
  process.env.JWT_SECRET,
  { expiresIn: '7d' }
);
```

Рисунок 4.12 – Функція створення токену

Коли користувач проходить реєстрацію або вхід, система викликає цю функцію і передає створений токен назад у відповідь. Надалі цей токен стає «ключем» до його особистого простору на сайті. Саме на основі цього ключа можна дізнатися, хто робить запит, і дати йому доступ до історії покупок, замовлень, даних профілю тощо.

Паролі користувачів, звичайно ж, не зберігаються у відкритому вигляді. Перед тим як записати їх у базу, ми хешуємо їх за допомогою бібліотеки bcryptjs. Це гарантує, що навіть у випадку витоку бази даних зловмисник не зможе просто прочитати паролі користувачів. Також при кожному вході, введений користувачем пароль порівнюється не з рядком у базі, а з його хешем. Така практика є стандартною в розробці безпечних вебсистем.

Окрему роль у захисті ресурсів відіграє middleware, наприклад, authMiddleware.js. Саме там перевіряється токен, і якщо він дійсний — у req.user додається інформація про користувача, що авторизувався. Усі інші частини

системи, які потребують авторизації, можуть розраховувати на те, що користувач уже перевірений і його дані доступні.

Отже, реалізація авторизації у ShopManager забезпечує повноцінну і безпечну ідентифікацію користувача, захищає приватні маршрути, і дозволяє зручно будувати подальшу логіку, пов’язану з персоналізацією. Усе це виконується за допомогою простих і ефективних інструментів — JSON Web Token, bcryptjs та middleware, які гарантують одночасно і безпеку, і зручність використання.

Після того як користувач зареєструвався або авторизувався в системі, наступним логічним кроком є надання йому доступу до особистого кабінету. У межах ShopManager особистий кабінет — це простір, де користувач може переглядати та оновлювати свої персональні дані, змінювати контактну інформацію, переглядати історію покупок або взаємодіяти з іншими модулями, пов’язаними з його активністю на сайті.

З технічного боку, доступ до інформації особистого кабінету реалізовано через модель Customer.js, яка описує, які саме поля має клієнт у базі даних. Це може бути ім’я, прізвище, електронна пошта, номер телефону, адреса доставки тощо. Усі ці поля доступні для читання і, за потреби, редагування. Маршрути, які відповідають за обробку таких запитів, зосереджені у файлі customers.js, а логіка обробки розміщена в customerController.js.

Коли клієнт заходить на сторінку свого кабінету, фронтенд надсилає GET-запит на спеціальний маршрут, наприклад /api/customers/profile. Цей запит супроводжується JWT-токеном, який ідентифікує користувача. На сервері токен розшифровується за допомогою middleware, і якщо він дійсний, сервер отримує ID користувача з токена та завантажує відповідні дані з бази. Вигляд методу для отримання профілю користувача показано на рис. 4.13.

```
const getProfile = async (req, res) => {
  try {
    const customer = await Customer.findById(req.customer._id).select("-password");
    if (!customer) return res.status(404).json({ message: "Користувача не знайдено" });
    res.json({ customer });
  } catch (err) {
    res.status(500).json({ message: "Помилка при отриманні профілю" });
  }
};
```

Рисунок 4.13 – Метод для отримання профілю користувача

У цьому прикладі спочатку виконується пошук у базі за ID, який був витягнутий із токена. Поле пароля, звичайно, не включається у відповідь — воно виключається за допомогою .select('-password'). У випадку, якщо користувача з таким ID не існує, повертається відповідь із помилкою. Зовнішній вигляд сторінки особистого кабінету користувача показано на рис. 4.14.

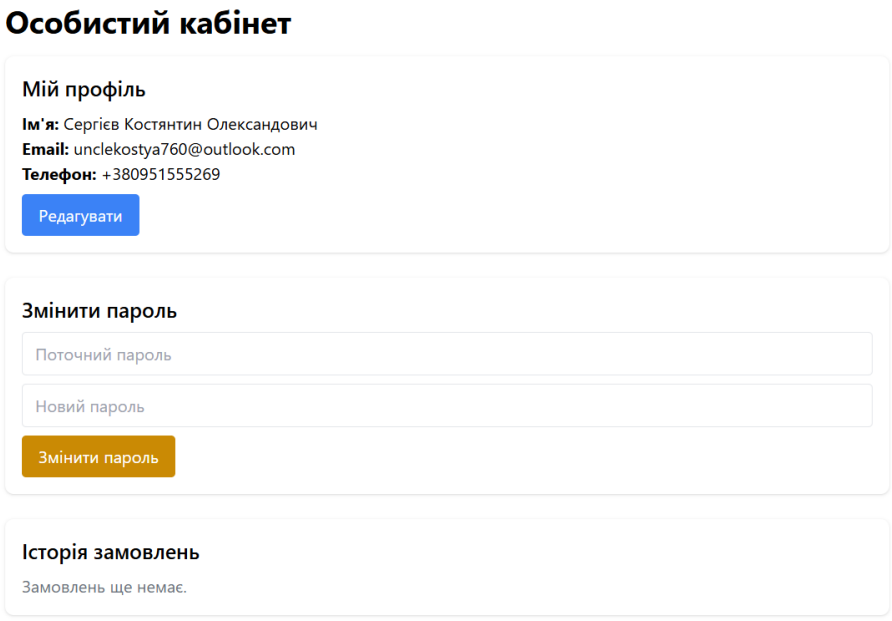

Рисунок 4.14 – Сторінка особистого кабінету користувача

Крім перегляду, клієнт також може змінювати свою контактну інформацію. Для цього реалізовано PUT-запит на той самий маршрут, але вже з оновленими даними у тілі запиту. Сервер перевіряє, чи користувач має право вносити ці зміни,

і зберігає нові значення в базі. Завдяки цій логіці користувач самостійно керує своїм обліковим записом без участі адміністратора.

Таким чином, реалізація особистого кабінету в ShopManager дозволяє кожному зареєстрованому клієнту мати доступ до своїх даних, вносити зміни, перевіряти замовлення або історію покупок. Це — обов'язковий компонент сучасного e-commerce застосунку, який підвищує рівень довіри й зручності для кінцевого користувача.

У кожному інтернет-магазині користувацька активність зводиться до ключового процесу — додавання товарів у кошик та формування замовлення. У системі ShopManager ця логіка реалізована за допомогою двох взаємопов'язаних модулів: обробки кошика та обробки замовлень. Вони тісно співпрацюють між собою, але відповідають за різні частини бізнес-процесу.

Першим кроком у цьому процесі є наповнення кошика. Коли користувач переглядає товари, він має змогу додавати їх у свій віртуальний кошик. Кожне таке додавання викликає відповідний маршрут у cartRoutes.js, який обробляється контролером cartController.js. Система зберігає інформацію про те, які саме товари обрані, у якій кількості та до якого користувача вони належать. Оскільки для кожного клієнта створюється індивідуальний кошик, ми можемо відстежувати вміст для кожного зареєстрованого облікового запису окремо.

Коли клієнт вирішує оформити замовлення, на сцену виходить другий модуль — модуль замовлень, що базується на orders.js та orderController.js. В момент підтвердження покупки дані з кошика переносяться у новий документ у базі даних. У ньому зберігається повна інформація про замовлення: список товарів, їх кількість, сума, спосіб оплати, статус замовлення, дані клієнта, а також у деяких випадках — логістична інформація.

Щоб замовлення не залишалося відірваним від користувача, кожне з них прив'язується до ID клієнта. Це забезпечує можливість у майбутньому переглядати історію покупок, відстежувати статус або повторювати замовлення. У структурі

бази MongoDB модель Order.js передбачає поле для збереження зв’язку з користувачем, що робить її частиною особистого простору клієнта.

Реалізація маршруту створення замовлення виглядає стандартно: POST-запит надсилає дані на сервер, де вони зберігаються як новий документ у колекції замовлень. Якщо все пройшло успішно, сервер повертає клієнту ID замовлення та підтвердження. Далі це замовлення може змінювати статус (наприклад, з "очікує оплату" на "оплачено"), і ці зміни відображаються в інтерфейсі клієнта або адміністратора.

Усю цю логіку супроводжують додаткові перевірки, наприклад, наявність товарів у базі, коректність введених даних і валідність знижок, якщо вони застосовуються. Але в центрі цього процесу завжди знаходиться зв’язок між клієнтом і його діями: те, що обрано в кошику, переходить у повноцінне замовлення, з усіма супутніми наслідками.

Зовнішній вигляд сторінки кошику показано на рис. 4.15.

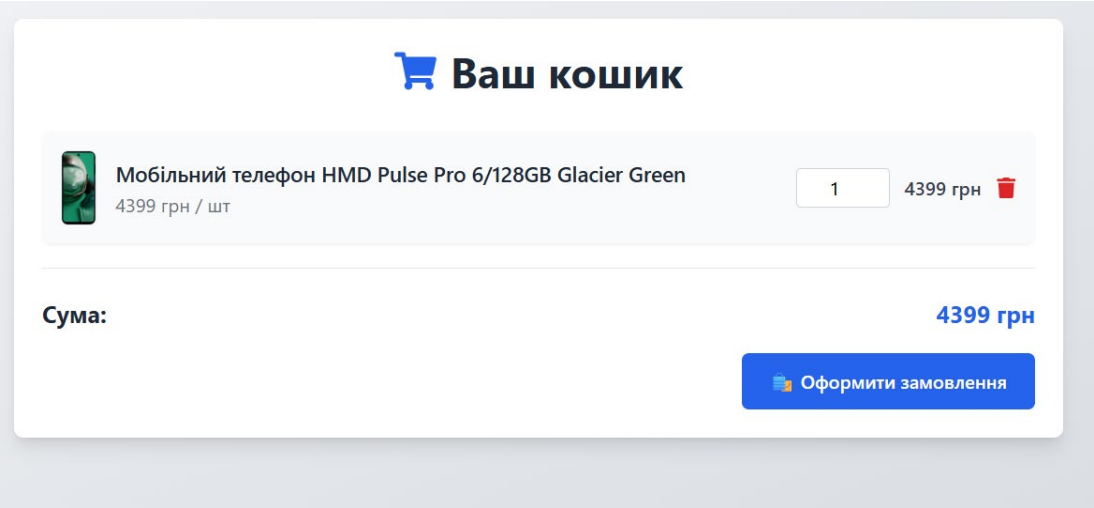

Рисунок 4.15 – Сторінка кошику покупця

Завдяки такій структурі ShopManager забезпечує цілісний, передбачуваний і стабільний механізм покупки, який об'єднує дії користувача від першого кліку на кнопку «додати в кошик» до фінального підтвердження замовлення. Це ключовий

елемент всієї системи, який поєднує в собі логіку обробки даних, збереження історії та підготовку до оплати й доставки.

Сучасний онлайн-магазин неможливо уявити без можливості здійснювати електронні платежі та використовувати систему знижок. У ShopManager ці два напрямки тісно інтегровані у загальний цикл оформлення замовлення. Їх мета — зробити процес покупки максимально зручним, швидким і гнучким для кінцевого користувача. Саме на цьому етапі відбувається обробка оплати через зовнішній платіжний сервіс, а також перевірка і застосування знижок чи подарункових сертифікатів.

У проєкті реалізовано інтеграцію з LiqPay — одним із найбільш поширених сервісів для прийому онлайн-платежів в Україні. Робота з ним організована через модулі liqpayRoutes.js, liqpayWebhookRoutes.js та контролер liqCon.js. Коли користувач переходить до етапу оплати, сервер генерує спеціальний HTML-код або посилання, яке відкриває платіжну форму LiqPay. У ній вже зашиті сума, опис замовлення, унікальний номер і ключі доступу, які підтверджують справжність запиту.

Важливою частиною цієї інтеграції є обробка зворотного зв’язку від платіжної системи. Після того як користувач здійснив оплату, LiqPay надсилає webhook-запит на спеціальний маршрут. У цьому запиті міститься інформація про те, чи вдалася транзакція. Сервер приймає ці дані, перевіряє їх достовірність і змінює статус відповідного замовлення у базі даних. Таким чином, навіть якщо користувач випадково закрив сторінку або не дочекався відповіді — інформація про оплату все одно буде врахована.

Паралельно з оплатою, ShopManager підтримує систему знижок та подарункових карт. Вона реалізована через окремі маршрути в discounts.js та giftCards.js. Кожна знижка або сертифікат зберігається у базі як окремий документ із власним унікальним кодом, сумою знижки, терміном дії та статусом (активний, використаний, прострочений). Коли користувач вводить код знижки під час

оформлення замовлення, сервер перевіряє його дійсність — чи ще активний код, чи не був використаний раніше, і чи відповідає сумі замовлення.

Вигляд методу для створення знижки у коді показано на рис. 4.16.

```
const createDiscount = async (req, res) => {
  const { percentage, productId } = req.body;
  if (!percentage || !productId) {
    return res.status(400).json({ message: "Необхідно вказати всі поля" });
  }
  try {
    const product = await Product.findById(productId);
    if (!product) {
      return res.status(404).json({ message: "Товар не знайдено" });
    }

    if (!product.originalPrice) {
      product.originalPrice = product.price;
    }
    const discountFactor = (100 - percentage) / 100;
    product.price = Math.round(product.originalPrice * discountFactor);
    await product.save();
    const newDiscount = new Discount({ percentage, productId });
    await newDiscount.save();
    res.status(201).json({ message: "Знижку застосовано", discount: newDiscount });
  } catch (err) {
    console.error(err);
    res.status(500).json({ message: "Помилка при створенні знижки" });
  }
};
```

Рисунок 4.16 – Метод для створення знижки

Після успішної перевірки відповідна сума знижки вираховується із загальної вартості замовлення. Якщо ж код недійсний або вже використаний — користувач отримує зрозуміле повідомлення про помилку. Такий механізм дозволяє легко створювати маркетингові акції, стимулювати повторні покупки та надавати персоналізовані бонуси. Те, як знижка виглядає на головній сторінці магазину, показано на рис. 4.17.

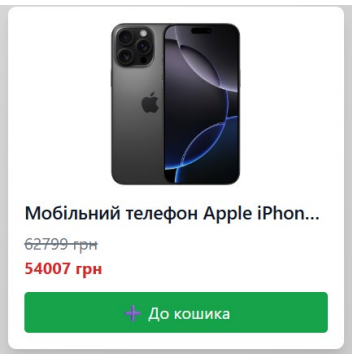

Рисунок 4.17 – Вигляд знижки на сторінці магазину

Впровадження платежів і знижок у ShopManager значно розширює функціональність системи та наближає її до повноцінного e-commerce рішення. Користувач отримує можливість не лише обрати товар, але й зручно оплатити його в кілька кліків, а також скористатися перевагами лояльності, що сприяє повторним замовленням і підвищенню довіри до платформи.

Адміністративна панель — це центральний інструмент для управління всім функціоналом магазину з боку персоналу або власника. У системі ShopManager вона не має окремого інтерфейсу на рівні API, однак уся логіка, яка відповідає за контроль над товарами, замовленнями, клієнтами, знижками та іншими модулями, реалізована у вигляді спеціально захищених маршрутів і контролерів. Саме ці серверні можливості лежать в основі адмін-панелі та забезпечують повний контроль над життєвим циклом роботи магазину.

Доступ до адміністративних функцій суворо обмежений. При створенні користувача в системі він може мати роль "admin", і тільки такі користувачі мають право змінювати товари, переглядати аналітику, керувати знижками або редагувати налаштування магазину. Авторизація на рівні адміністратора реалізується за тим же принципом, що і в звичайних користувачів, але додатково кожен запит до закритих маршрутів перевіряється через middleware, яке перевіряє роль користувача. Якщо роль не відповідає "admin", запит блокується, навіть якщо токен дійсний.

Управління товарами — один із найважливіших блоків адміністративної панелі. Через відповідні маршрути, описані у products.js, адміністратор може створювати нові товари, редагувати вже існуючі, змінювати ціни, категорії, опис і зображення. Оновлення даних реалізовано як стандартні PUT- і DELETE-запити. Весь функціонал розподілено по контролері productController.js, де кожна дія реалізована окремою функцією: наприклад, створення нового товару, видалення, пошук за ID або оновлення. Вигляд сторінки з товарами показано на рис. 4.18.

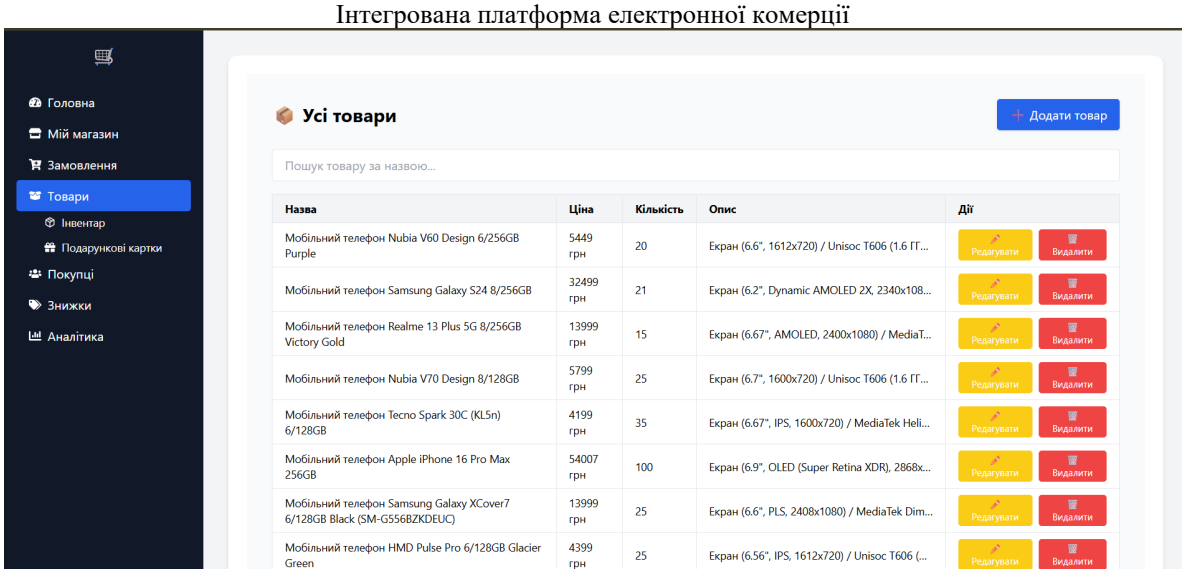

Рисунок 4.18 – Сторінка з товарами в адмін-панелі

Крім товарів, адмін має доступ до замовлень усіх клієнтів. Це дозволяє відстежувати їхній статус, бачити, які товари були замовлені, коли здійснена оплата, який спосіб доставки обрано, це показано на рис. 4.19. Відповідні запити обробляються в orderController.js. Тут же можна змінювати статус замовлення — наприклад, вручну позначити його як «виконано», «оплачено» або «відправлено», якщо обробка проводиться вручну.

Рисунок 4.19 – Відстеження активних замовлень зі сторони адміна

Також до функціоналу адміністративної панелі входить керування знижками та подарунковими сертифікатами. Через відповідні інтерфейси адміністратор може створювати знижкові коди з відсотковим значенням, обмеженням по терміну дії та статусом активності. Всі такі об’єкти зберігаються у колекції Discount, а взаємодія з ними відбувається через окремі маршрути у discountRoutes.js. Сторінка перегляду наявних знижок показана на рис. 4.20.

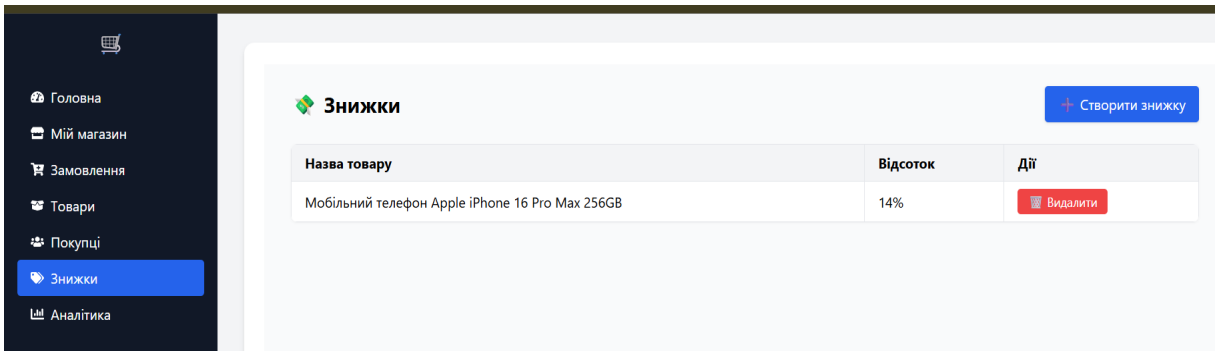

Рисунок 4.20 – Сторінка перегляду наявних знижок

Адміністративна панель оформлена в єдиному стилі з рештою інтерфейсу, проте кожен її компонент доступний лише після перевірки прав доступу. Це забезпечує як зручність для працівників магазину, так і захист від несанкціонованого втручання.

4.4 Тестування ефективності та працездатності реалізованої функціональності

Перший етап функціонального тестування зосереджено на перевірці механізму реєстрації нового клієнта, що є критично важливим для будь-якого e-commerce застосунку. У платформі ShopManager цей процес реалізовано у два етапи: введення основних даних (ім’я, email, телефон, пароль) та підтвердження через код, надісланий на електронну пошту. Такий підхід забезпечує не лише зручність користувача, а й захист від створення фейкових акаунтів.

У ході тестування послідовно перевірялась робота клієнтського інтерфейсу, коректність валідації форм, реакція на неправильні або дубльовані email-адреси, генерація коду підтвердження на сервері, а також його надсилання через SMTP-сервер. Завершальним етапом була перевірка створення облікового запису після введення коду, з подальшою авторизацією користувача. Вигляд заповненої форми показано на рис. 4.21.

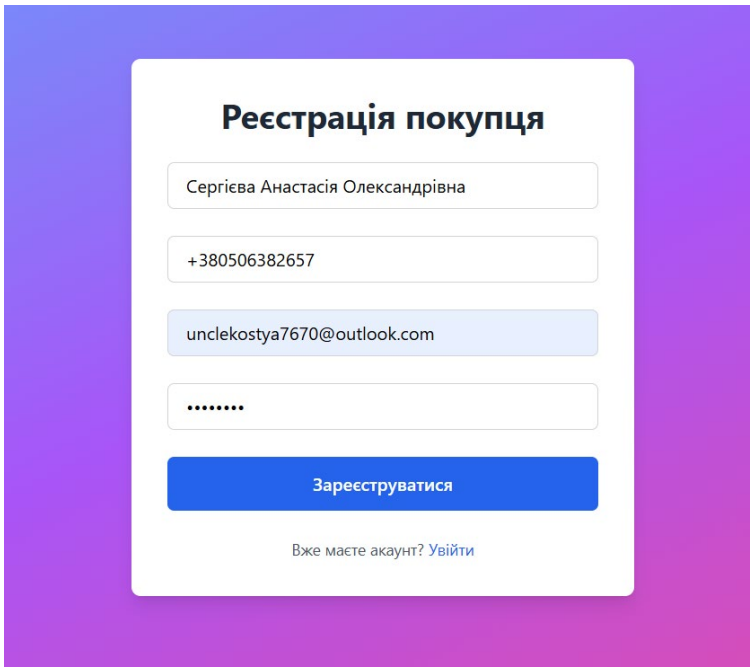

Рисунок 4.21 – Вигляд заповненої форми

Усі перевірки були успішно пройдені: дані зберігаються коректно в локальному серидовищі, натискання кнопки зареєструватися перекидає нас на сторінку підтвердження пошти. Під час завантаження цієї сторінки, на пошту вказану користувачем надсилається відповідний код, показано на рис. 4.22

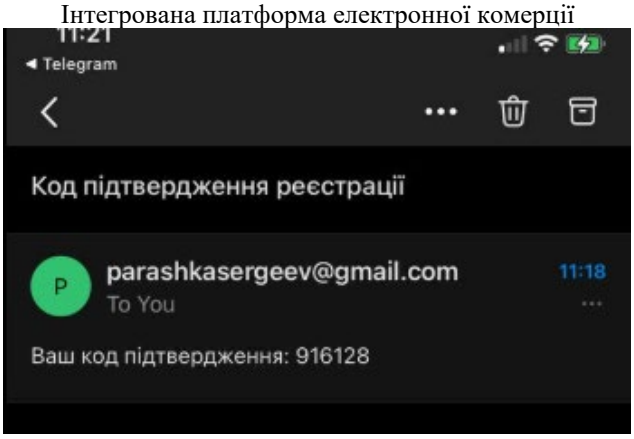

Рисунок 4.22 – Код отриманий користувачем

На рис. 4.23 показано, що введення коду, отриманого з листа дійсно підтверджує реєстрацію.

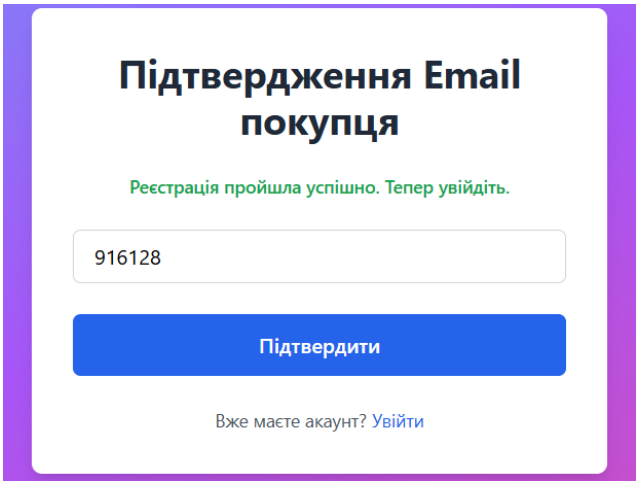

Рисунок 4.23 – Правильне введеня коду

На другому етапі тестування було перевірено процес входу до системи, зокрема — обробку логіна користувача, створення та збереження JWT-токена, а також доступ до захищених маршрутів на основі цього токена. Після успішної реєстрації або наявності раніше створеного акаунта користувач переходить до форми входу, де вводить email і пароль. У відповідь на запит сервер перевіряє дані, генерує токен та надсилає його клієнту для подальших запитів.

Додатково була протестована реакція системи на некоректний пароль, як показано на рис. 4.24. Також важливою частиною стало тестування доступу до маршруту /customerAuth/customer/me, який має повертати дані лише у випадку наявності дійсного токена. Усі запити, де токен був відсутній або неправильний, блокувалися, як і передбачено middleware-функцією.

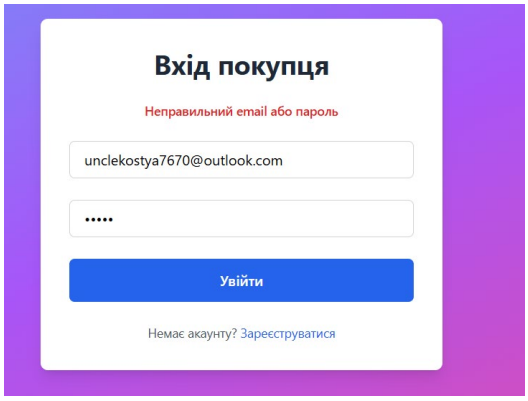

Рисунок 4.24 – Реакція системи на некоректний пароль

Результати показали, що механізм автентифікації повністю функціонує згідно з очікуваннями: користувачі без токена не мають доступу до персональних даних, логін працює стабільно, а авторизований користувач одразу отримує доступ до свого профілю та магазину.

Наступним етапом було тестування сторінки профілю клієнта — особистого кабінету, де відображаються персональні дані користувача, його історія замовлень, а також можливість оновити контактну інформацію. Після авторизації фронтенд надсилає запит типу GET /customerAuth/customer/me, у відповідь на який сервер повертає знеособлені дані користувача без пароля. Ці дані автоматично заповнюють відповідні поля на сторінці профілю. Редагування користувацьких даних показано на рис. 4.25.

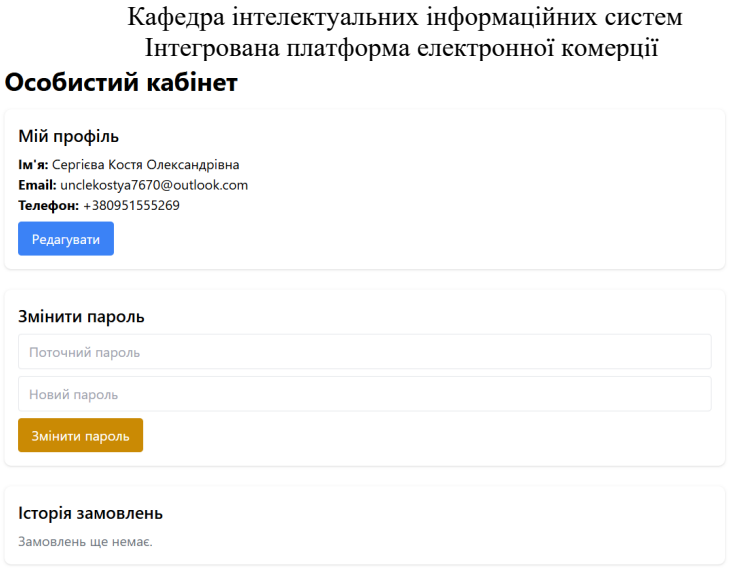

Рисунок 4.25 – Сторінка редагування користувацьких даних

Окремо перевірялась функція оновлення інформації користувача. Коли клієнт редагує свої дані (наприклад, ім’я або номер телефону) та надсилає PUT-запит, система не тільки зберігає нову інформацію в базі, а й одразу оновлює локальний стан на фронтенді. У випадку помилок – наприклад, порожнього обов’язкового поля — інтерфейс виводить попередження без перезавантаження сторінки.

У результаті перевірки функціональність особистого кабінету показала високу стабільність і зручність. Дані завжди актуальні, редагування проходить плавно, а захист інформації забезпечується правильною перевіркою токена. Інтерфейс коректно оновлюється після змін, що підтверджує правильну взаємодію фронтенда та бекенда.

У цьому кроці було протестовано модуль додавання та редагування товарів у кошику користувача. Основна мета — переконатися, що додавання, зміна кількості та видалення товарів працює без збоїв як у локальному стані React-додатку, так і на рівні бази даних (MongoDB). Для прикладу, додамо до кошика товар під назвою «Мобыльний телефон Realme 13 Plus» 2 рази. Результат отримуємо на рис. 4.26

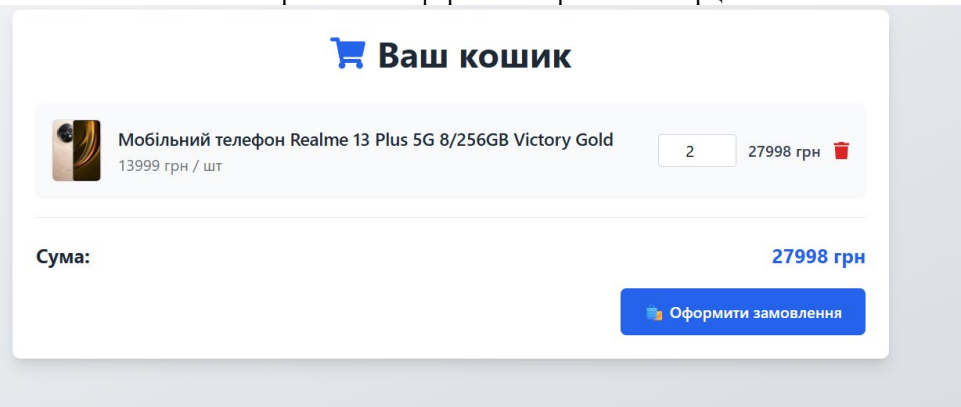

Рисунок 4.26 – Сторінка кошику

Після натискання кнопки «Додати в кошик» на сторінці продукту, клієнтський застосунок надсилає запит до POST /cart, який додає товар до масиву cart у документі поточного користувача. Якщо товар уже існує в кошику, замість дублювання змінюється лише його кількість. Перевірялась також можливість змінювати кількість вручну на сторінці CartPage.jsx — при цьому відправляється PUT-запит із новим значенням, і одразу оновлюється інтерфейс.

Функція «Видалити» викликає DELETE-запит, який оновлює сервер і синхронізує стан кошика на клієнті. Сторінка повністю динамічна й не потребує перезавантаження після кожної дії, що підвищує зручність користувача. Результати тестування показали повну відповідність очікуваному функціоналу: кошик працює стабільно, товари додаються, змінюються та видаляються правильно. Усі зміни відображаються в інтерфейсі негайно, без затримок або помилок. Це свідчить про ефективну інтеграцію frontend- і backend-логіки в частині керування кошиком.

На цьому етапі перевірялась коректність процесу завершення покупки: від заповнення форми доставки до генерації замовлення в базі даних. Користувач переходить на сторінку CheckoutPage.jsx, де заповнює контактні дані, вибирає спосіб доставки (Нова Пошта) та оплату (готівка або LiqPay). Сторінка заповнення даних для замовлення показана на рис. 4.27

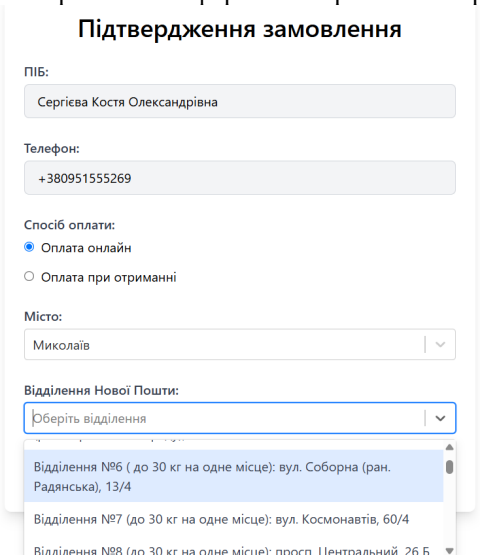

Рисунок 4.27 – Сторінка оформлення замовлення

Після підтвердження форми на сервер надсилається POST-запит на /orders, у якому міститься вся інформація з кошика, включно з товарами, їх кількістю, загальною сумою та даними клієнта. Для оплати онлайн активується генерація платіжного посилання через маршрут /liqpay/create-payment. У відповідь користувач автоматично перенаправляється до платіжної сторінки LiqPay з усіма даними замовлення. Сторінка для оплати онлайн, на яку користувача закидує при оформленні замовлення, показано на рис. 4.28.

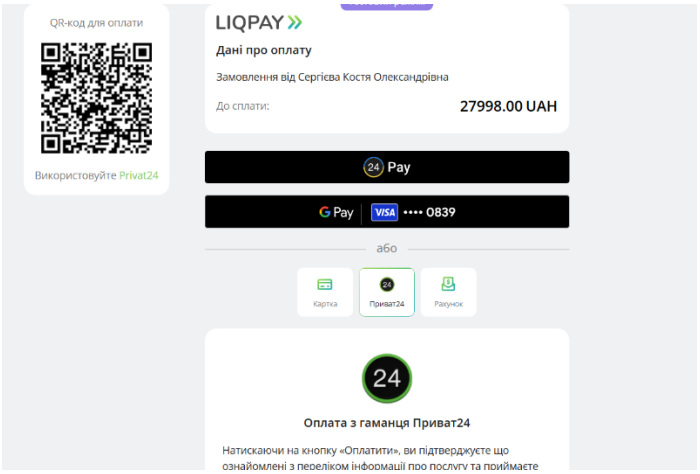

Рисунок 4.28 – Сторінка онлайн-оплати

У випадку вибору оплати готівкою замовлення створюється одразу з позначкою isPaid: false, і після збереження в базі клієнт отримує підтвердження у вигляді повідомлення. Також тестувались випадки з помилками: наприклад, якщо не вказане місто чи відділення, сервер повертає відповідну помилку, яку інтерфейс відображає через повідомлення.

Усі перевірки підтвердили: механізм оформлення працює коректно. Дані передаються на сервер без втрат, відображення помилок реалізовано зручно, інтеграція з LiqPay працює стабільно, а підтвердження замовлення з'являється миттєво. Це свідчить про готовність checkout-модуля до реального використання.

Адміністративна частина платформи передбачає повний контроль над асортиментом інтернет-магазину. Для цього реалізовано функціональність додавання, редагування та видалення товарів через захищені маршрути POST, PUT, DELETE у productRoutes.js. Відповідна логіка описана в productController.js.

Адміністратор, після входу в систему з обліковими даними з правами isAdmin: true, отримує доступ до сторінки товарів. Тут йому доступна форма, показана на рис. 4.29, для створення нового товару з введенням назви, опису, ціни, зображення та кількості на складі.

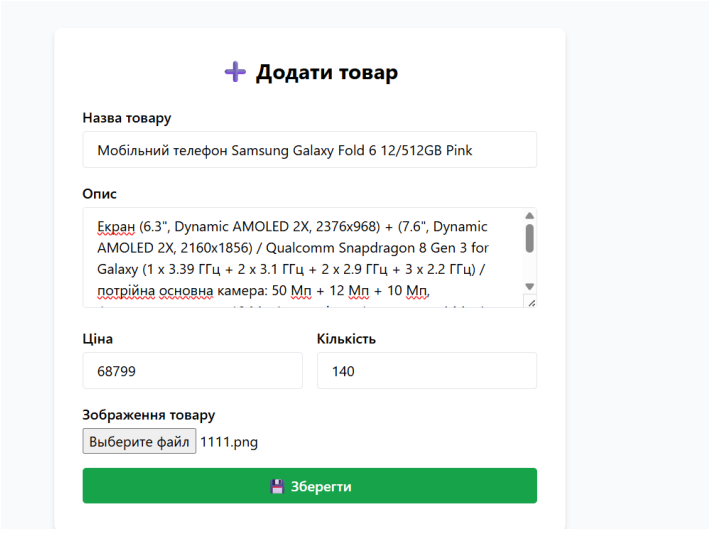

Рисунок 4.29 – Додавання нового товару

Для редагування використовуються вже наявні товари, які можна оновити натисканням на кнопку «Редагувати», що відкриває форму з попередньо заповненими даними. Доданий товар, як видно на рис. 4.30, тепер знаходиться на сторінці магазину.

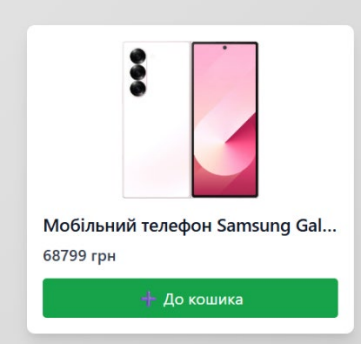

Рисунок 4.30 – Товар на сторінці магазину

Усі запити до API виконуються коректно, товари зберігаються в базі даних або оновлюються відповідно до змін. Інтерфейс не допускає пустих полів, неправильної ціни чи некоректного зображення. Завдяки цій реалізації адміністративна панель дозволяє ефективно та безпечно керувати асортиментом платформи.

Однією з найважливіших частин e-commerce платформи є модуль керування замовленнями, показаний на рис. 4.31. У ShopManager адміністратор має доступ до всіх створених замовлень користувачів через відповідний інтерфейс. Сторінка замовлень реалізована таким чином, що відображає список усіх заявок із ключовими параметрами: ім’я клієнта, сума замовлення, статус оплати, спосіб доставки та дата створення.

Рисунок 4.31 – Сторінка керування замовленнями

Система чудово обробляє запити керування замовленнями. Зміна статусу замовлення миттєво фіксуються у базі, а відображення в інтерфейсі оновлюється автоматично. Завдяки цьому модуль замовлень виконує свою функцію надійно, забезпечуючи адміністраторам повний контроль над етапами виконання покупок.

Інтеграція механізму знижок у платформі ShopManager дозволяє в майбутньому впроваджувати програми лояльності, проводити акції та персоналізувати взаємодію з клієнтами. Для адміністраторів реалізовано окремий інтерфейс, який дозволяє створювати, редагувати та деактивувати знижки.

Процес створення знижки на товар показано на рис. 4.32.

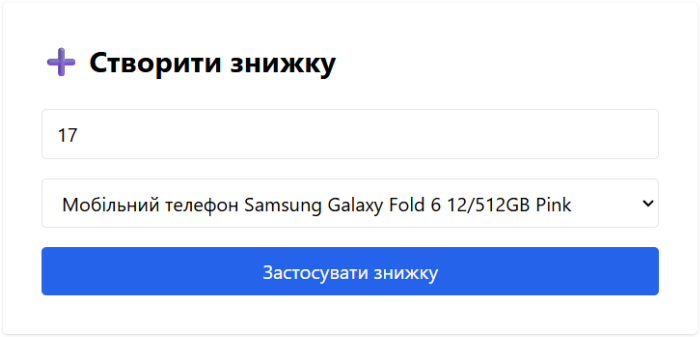

Рисунок 4.32 – Сторінка створення знижки

Усі функції системи знижок працюють надійно: нові знижки створюються й оновлюються коректно, як показано на рис. 4.33, сума в замовленні коригується правильно.

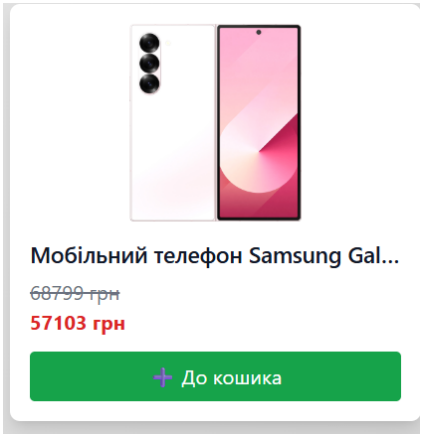

Рисунок 4.33 – Знижка на сторінці магазину

У результаті проведеного тестування функціональності вебзастосунку ShopManager було перевірено ключові аспекти роботи як клієнтської частини, так і адміністративної панелі. Особливу увагу приділено коректності процесів автентифікації, обробці кошика, створенню замовлень, роботі зі знижками та оплатою, а також обмеженню доступу до захищених маршрутів. Усі ці дії були протестовані у типовому сценарії взаємодії кінцевого користувача з інтерфейсом, а також з точки зору адміністратора магазину.

Кожен етап було протестовано як вручну у браузері, так і з використанням DevTools (перевірка HTTP-запитів, відповідей сервера, обробка помилок). Всі функції показали стабільну та передбачувану поведінку, з правильним виведенням повідомлень, обробкою токенів, збереженням даних у базі та коректною взаємодією між фронтендом і бекендом. Завдяки продуманій структурі коду та чітко реалізованій логіці, платформа продемонструвала надійну працездатність у всіх ключових точках взаємодії.

Таким чином, проведене тестування підтвердило відповідність реалізованого функціоналу технічним вимогам, а також довело готовність системи до подальшого розгортання та використання в умовах реального середовища.

Висновки до розділу 4

У цьому розділі було детально розглянуто етапи реалізації інтегрованої платформи електронної комерції, що охоплює як клієнтську, так і серверну частину вебзастосунку. Клієнтська частина побудована з використанням бібліотеки React, що забезпечило гнучкий, динамічний та адаптивний інтерфейс, орієнтований на зручність користувача. Застосування компонентного підходу дозволило ефективно розподілити функціонал між окремими модулями системи, а використання React Router забезпечило швидку та зручну навігацію без перезавантаження сторінок.

Серверна частина була реалізована на основі Node.js з Express, з урахуванням безпечної авторизації через JWT, обробки запитів REST API, взаємодії з MongoDB та зовнішніми сервісами (оплата, поштові сервіси, перевірка знижок). Було реалізовано повноцінну систему ролей, де адміністратор має окремі інструменти для керування товарами, замовленнями, знижками та аналітикою, тоді як користувачі взаємодіють із функціоналом відповідно до своїх прав доступу.

Окрему увагу приділено тестуванню функціональності. Кожен ключовий модуль, як-то реєстрація, кошик, обробка замовлень, оплата через LiqPay, застосування знижок і подарункових сертифікатів, був перевірений на практиці. Результати тестування засвідчили коректну роботу основних сценаріїв, стабільність застосунку та відповідність реалізованого функціоналу поставленим завданням.

Таким чином, реалізована система відповідає вимогам сучасного електронного магазину та забезпечує повний цикл взаємодії користувача — від реєстрації до оплати й обробки замовлення.

ВИСНОВКИ

У результаті виконання КР було успішно реалізовано повнофункціональну інтегровану платформу електронної комерції, орієнтовану на потреби сучасного онлайн-магазину. Проєкт охоплює повний цикл життєдіяльності користувача — від реєстрації та входу в особистий кабінет до здійснення покупки, обробки замовлення та взаємодії з адміністративною панеллю. Архітектура вебзастосунку побудована за принципом SPA із використанням технологій React на клієнтській частині та Node.js/Express на сервері, що дозволило досягти високої швидкості та зручності взаємодії.

У ході розробки було спроєктовано і реалізовано ряд ключових функцій: механізм автентифікації з підтвердженням email, система обліку користувачів і ролей, управління товарами, кошиком, замовленнями, знижками та подарунковими сертифікатами. Адміністративна панель дозволяє контролювати бізнес-процеси та оновлювати дані в реальному часі, а клієнтський інтерфейс забезпечує інтуїтивну навігацію і підтримку різних сценаріїв взаємодії. Проведене тестування підтвердило стабільність роботи основних модулів і відповідність функціональності поставленим вимогам.

Отже, поставлені в роботі цілі були досягнуті. Розроблений застосунок є масштабованим, безпечним та зручним у використанні. Його структура дозволяє легко доповнювати платформу новими модулями, інтегрувати зовнішні сервіси та адаптувати під потреби реального бізнесу, що свідчить про успішність реалізації даного програмного рішення.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Karczewski D. What Are The Best Frontend Frameworks To Use In 2023? Ideamotive. URL: https://www.ideamotive.co/blog/best-frontend-frameworks (Last accessed: 29.04.2025).

2. Безверхий О., Куценко О. Ефективність застосування бібліотеки React. Інформаційні технології та суспільство. 2022. Т. 2, № 4. С. 13–19. DOI: 10.32689/maup.it.2022.2.2.

3. React. Official documentation. URL: https://reactjs.org/docs/getting-started.html (Last accessed: 29.05.2025).

4. Lauer P., Zintel M. React – Einführung in ein Framework. University of Applied Sciences Kaiserslautern. 2020. URL: https://www.researchgate.net/publication/357684856_React_-_Einfuhrung_in_ein_Framework (Last accessed: 29.05.2025).

5. Jain R., Shrivastava V., Pandey A., Sharma A. Modern Web Development using CSS & HTML. International Journal of Emerging Science and Engineering. 2024. Vol. 12, No. 6. P. 13–16. DOI: 10.35940/ijese.G2574.12060524.

6. Attardi J. Modern CSS: Master the Key Concepts of CSS for Modern Web Development. Apress: Berkeley, CA, USA. Jan 2020. p. 289. DOI: 10.1007/978-1-4842-6294-8. ISBN: 978-1-4842-6293-1.

7. Tailwind CSS. Responsive design – Core concepts. URL: https://tailwindcss.com/docs/responsive-design (Last accessed: 29.05.2025).

8. Master Responsive Design with Tailwind CSS Utility Classes. URL: https://tailgrids.com/blog/learn-tailwind-css-mastering-responsive-design (Last accessed: 29.05.2025).

9. React.js Best Practice (Most Secure) Role-Based User Authentication (StackOverflow). URL: https://stackoverflow.com/questions/71852652/react-js-best-practice-most-secure-role-based-user-authentication-jwt-comp (Last accessed: 29.05.2025).

10. Implementing Authentication with JWT Tokens in React and Express.js. URL: https://dev.to/devloker/authentication-with-jwt-tokens-in-react-and-expressjs-5o9 (Last accessed: 29.05.2025).

11. How to implement JWT authentication with React and Node.js. URL: https://medium.com/@simonsruggi/how-to-implement-jwt-authentication-with-react-and-node-js-5d8bf3e718d0 (Last accessed: 29.05.2025).

12. Best Practices for Securing JWT Tokens in React Applications. URL: https://medium.com/@myfacesproduction/best-practices-for-securing-jwt-tokens-in-react-applications-cc9f63b4dbc0 (Last accessed: 29.05.2025).

13. JWT authentication: Best Practices and When to Use It. LoginRadius. URL: https://www.loginradius.com/blog/engineering/guest-post/jwt-authentication-best-practices-and-when-to-use (Last accessed: 29.05.2025).

14. Secure Your REST API with JSON Web Tokens (JWT). Medium. URL: https://medium.com/@louistrinh/secure-your-rest-api-with-json-web-tokens-jwt-fb899d2642ff (Last accessed: 29.05.2025).

15. How to secure a REST API using JWT authentication. LogRocket Blog. URL: https://blog.logrocket.com/secure-rest-api-jwt-authentication (Last accessed: 29.05.2025).

16. JWT authentication: Basics and best practices. Medium. URL: https://medium.com/@darshana-edirisinghe/jwt-security-concerns-f79e63ff4871 (Last accessed: 29.05.2025).

17. Guidelines to build a secure JWT authentication process? (StackOverflow). URL: https://stackoverflow.com/questions/65946041/guidelines-to-build-a-secure-jwt-authentication-process (Last accessed: 29.05.2025).

18. Best Practices for REST API security: Authentication and authorization. StackOverflow Blog. URL: https://stackoverflow.blog/2021/10/06/best-practices-for-authentication-and-authorization-for-rest-apis (Last accessed: 29.05.2025).

19. Salting and Hashing Passwords with bcrypt.js. Medium. URL: https://medium.com/@arunchaitanya/salting-and-hashing-passwords-with-bcrypt-js-a-comprehensive-guide-f5e31de3c40c (Last accessed: 29.05.2025).

20. Hashing in Action: Understanding bcrypt. Auth0 Blog. URL: https://auth0.com/blog/hashing-in-action-understanding-bcrypt (Last accessed: 29.05.2025).

21. Password hashing in Node.js with bcrypt. LogRocket Blog. URL: https://blog.logrocket.com/password-hashing-node-js-bcrypt (Last accessed: 29.05.2025).

22. Bcrypt at 25: A Retrospective on Password Security. USENIX. URL: https://www.usenix.org/publications/loginonline/bcrypt-25-retrospective-password-security (Last accessed: 29.05.2025).

23. How to Handle Passwords Safely with BcryptJS. DigitalOcean. URL: https://www.digitalocean.com/community/tutorials/how-to-handle-passwords-safely-with-bcryptjs-in-javascript (Last accessed: 29.05.2025).

24. Password Hashing using bcrypt. Medium. URL: https://medium.com/@bhupendra_Maurya/password-hashing-using-bcrypt-e36f5c655e09 (Last accessed: 29.05.2025).

25. Password Security 101 - Understanding How Hashing Works. mannhowie.com. URL: https://mannhowie.com/password-security (Last accessed: 29.05.2025).

26. Data Modeling: Sample E-Commerce System with MongoDB. InfoQ. URL: https://www.infoq.com/articles/data-model-mongodb (Last accessed: 29.05.2025).

27. Inclusion of e-commerce workflow with NoSQL DBMS – ResearchGate. URL: https://www.researchgate.net/publication/316828937_Inclusion_of_e-commerce_workflow_with_NoSQL_DBMS_MongoDB_document_store (Last accessed: 29.05.2025).

28. Building a MongoDB NoSQL E-Commerce Data Model. HackerNoon. URL: https://hackernoon.com/building-a-mongodb-nosql-e-commerce-data-model-fn8135bc (Last accessed: 29.05.2025).

29. MongoDB e-Commerce Database Model. ResearchGate. URL: https://www.researchgate.net/publication/300350857_MongoDB_e-Commerce_Database_Model (Last accessed: 29.05.2025).

30. MongoDB: A Case Study for e-fashion retailers. Western Oregon University. URL: https://people.wou.edu/~hsamkari18/files/Professional%20Project%20IS642.pdf (Last accessed: 29.05.2025).

31. Unified Modeling Language Explained. MongoDB. URL: https://www.mongodb.com/resources/basics/unified-modeling-language (Last accessed: 29.05.2025).

32. How to draw UML diagram for NoSQL like MongoDB? StackOverflow. URL: https://stackoverflow.com/questions/38616623/how-to-draw-uml-diagram-for-nosql-like-mongodb (Last accessed: 29.05.2025).

33. From Class Diagram to Node.js API Server with Loopback.io. Medium. URL: https://medium.com/nycdev/from-class-diagram-to-node-js-api-server-with-loopback-io-859d8d1f4759 (Last accessed: 29.05.2025).

34. Node.js UML design discussion. StackOverflow. URL: https://stackoverflow.com/q/30686871 (Last accessed: 29.05.2025).

35. NodeMDA: UML to code generation. GitHub. URL: https://github.com/joelkoz/NodeMDA (Last accessed: 29.05.2025).

36. NodeUML Visual Studio extension. Marketplace. URL: https://marketplace.visualstudio.com/items?itemName=joelkoz.nodeuml (Last accessed: 29.05.2025).

37. LiqPay SDK for NodeJS. GitHub. URL: https://github.com/liqpay/sdk-nodejs (Last accessed: 29.05.2025).

38. LiqPay Payment Integration guide. Reintech.io. URL: https://reintech.io/blog/custom-checkout-experience-liqpay-payment-api (Last accessed: 29.05.2025).

39. "Welcome to LiqPay!"—Doc overview. URL: https://www.liqpay.ua/en/doc (Last accessed: 29.05.2025).

40. LiqPay sandbox token payments. URL: https://www.liqpay.ua/en/doc/api/internet_acquiring/token (Last accessed: 29.05.2025).

41. Nova Poshta integration for online stores. Weblium Help Center. URL: https://help.weblium.com/en-us/article/connecting-nova-poshta-to-an-online-store-1ky38zd (Last accessed: 29.05.2025).

42. NovaPoshta Global services. URL: https://novaposhtaglobal.ua/en/ (Last accessed: 29.05.2025).

43. Nova Poshta Simple Delivery module. CS-Cart Marketplace. URL: https://marketplace.cs-cart.com/nova-poshta-simple-delivery.html (Last accessed: 29.05.2025).

44. Nova Poshta API overview. NovaPost. URL: https://novapost.com/en-ua/for-business/cooperation/integration/index.html (Last accessed: 29.05.2025).

45. Secure API Development Best Practices – OAuth2 and JWT. Conviso AppSec. URL: https://blog.convisoappsec.com/en/secure-api-development-best-practices-oauth2-and-jwt (Last accessed: 29.05.2025).

46. liqpay/sdk-nodejs examples on CodeSandbox. URL: https://codesandbox.io/examples/package/liqpay (Last accessed: 29.05.2025).

47. Best practices for responsive classes (Tailwind). URL: https://www.reddit.com/r/tailwindcss/comments/retza1/best_practices_for_responsive_classes (Last accessed: 29.05.2025).

48. Mastering Responsive Design with Tailwind CSS. Medium. URL: https://medium.com/@harutyunabgaryann/mastering-responsive-design-with-tailwind-css-essential-tips-and-tricks-5128da2b5df9 (Last accessed: 29.05.2025).

49. How to create MongoDB schema diagram using DbSchema. URL: https://twinkal189.medium.com/how-to-create-mongodb-schema-diagram-using-dbschema-b9419a7bcc89 (Last accessed: 29.05.2025).

50. Software Ideas Modeler. UML Diagrams tool. URL: https://www.softwareideas.net/c/1058/UML (Last accessed: 29.05.2025).

ДОДАТОК А

Файл customerAuth.js

```
const express = require("express");
const bcrypt = require("bcryptjs");
const jwt = require("jsonwebtoken");
const nodemailer = require("nodemailer");
const Customer = require("../models/Customer");
const protectCustomer = require("../middleware/protectCustomer");
const router = express.Router();
const pendingRegistrations = new Map();
const transporter = nodemailer.createTransport({
  service: "Gmail",
  auth: {
    user: process.env.EMAIL_USER,
    pass: process.env.EMAIL_PASS,
  }
});
router.post("/initiate-registration", async (req, res) => {
  const { name, email, phone, password } = req.body;
  if (!name || !email || !password) {
    return res.status(400).json({ message: "Усі поля обов’язкові" });
  }
  try {
    const existing = await Customer.findOne({ email });
    if (existing) {
      return res.status(400).json({ message: "Email вже зареєстрований" });
    }
    const code = Math.floor(100000 + Math.random() * 900000).toString();
    const hashedPassword = await bcrypt.hash(password, 10);
    pendingRegistrations.set(email, {
      name,
      email,
      phone,
      password: hashedPassword,
      code,
      expires: Date.now() + 10 * 60 * 1000 // 10 хвилин
    });
    await transporter.sendMail({
      from: "your_email@gmail.com",
      to: email,
      subject: "Код підтвердження реєстрації",
      text: `Ваш код підтвердження: ${code}`
    });
    res.json({ message: "Код відправлено на email" });
  } catch (err) {
    console.error(err);
    res.status(500).json({ message: "Помилка сервера" });
  }
});
router.post("/login", async (req, res) => {
  const { email, password } = req.body;
  try {
    const customer = await Customer.findOne({ email });
    if (!customer) {
      return res.status(401).json({ message: "Неправильний email або пароль" });
```

```
    }
    const isMatch = await bcrypt.compare(password, customer.password);
    if (!isMatch) {
      return res.status(401).json({ message: "Неправильний email або пароль" });
    }
    const token = jwt.sign({ id: customer._id }, process.env.JWT_SECRET ||
"default_secret", {
      expiresIn: "7d",
    });
    res.json({
      message: "Вхід успішний",
      token,
      customer: {
        id: customer._id,
        name: customer.name,
        email: customer.email,
      },
    });
  } catch (err) {
    console.error(err);
    res.status(500).json({ message: "Помилка входу" });
  }
});
router.post("/verify-code", async (req, res) => {
  const { email, code } = req.body;
  const data = pendingRegistrations.get(email);
  if (!data) {
    return res.status(400).json({ message: "Реєстрацію не знайдено або час вийшов" });
  }
  if (data.code !== code) {
    return res.status(400).json({ message: "Невірний код" });
  }
  if (Date.now() > data.expires) {
    pendingRegistrations.delete(email);
    return res.status(400).json({ message: "Термін дії коду вичерпано" });
  }
  try {
    const newCustomer = new Customer({
      name: data.name,
      email: data.email,
      phone: data.phone,
      password: data.password
    });
    await newCustomer.save();
    pendingRegistrations.delete(email);
    res.status(201).json({ message: "Реєстрація підтверджена" });
  } catch (err) {
    console.error(err);
    res.status(500).json({ message: "Помилка збереження користувача" });
  }
});
router.get("/me", protectCustomer, async (req, res) => {
  try {
    const customer = await Customer.findById(req.customer.id).select("-password");
    if (!customer) {
      return res.status(404).json({ message: "Користувача не знайдено" });
    }
    res.json({ customer });
  } catch (err) {
```

```
    console.error(err);
    res.status(500).json({ message: "Помилка сервера" });
  }
});
module.exports = router;
```

Файл CustomerAuthPage.jsx

```
import { useState, useEffect } from "react";
import axios from "axios";
import { useNavigate, useLocation } from "react-router-dom";
export default function CustomerAuthPage() {
  const [isRegister, setIsRegister] = useState(false);
  const [isCodeSent, setIsCodeSent] = useState(false);
  const [verificationCode, setVerificationCode] = useState("");
  const [tempEmail, setTempEmail] = useState("");
  const [form, setForm] = useState({
    name: "",
    email: "",
    phone: "",
    password: "",
  });
  const [error, setError] = useState("");
  const [successMessage, setSuccessMessage] = useState("");
  const navigate = useNavigate();
  const location = useLocation();
  useEffect(() => {
    const params = new URLSearchParams(location.search);
    if (params.get("justRegistered") === "true") {
      setSuccessMessage("Реєстрація пройшла успішно. Тепер увійдіть.");
    }
  }, [location.search]);
  const handleChange = (e) => {
    setForm({ ...form, [e.target.name]: e.target.value });
  };
  const handleSubmit = async (e) => {
    e.preventDefault();
    setError("");
    setSuccessMessage("");
    try {
      if (isRegister && !isCodeSent) {
        await axios.post("http://localhost:5000/customerAuth/customer/initiate-
registration", form);
        setTempEmail(form.email);
        setIsCodeSent(true);
      } else if (isRegister && isCodeSent) {
        await axios.post("http://localhost:5000/customerAuth/customer/verify-code", {
          email: tempEmail,
          code: verificationCode,
        });
        navigate("/customerAuth?justRegistered=true");
      } else {
        const res = await axios.post("http://localhost:5000/customerAuth/customer/login",
{
          email: form.email,
          password: form.password,
        });
        localStorage.setItem("customerToken", res.data.token);
```

```
        navigate("/");
      }
    } catch (err) {
      setError(err.response?.data?.message || "Помилка");
    }
  };
  return (
    <div className="min-h-screen flex items-center justify-center bg-gradient-to-br from-
blue-400 via-purple-500 to-pink-500 px-4">
      <form
        onSubmit={handleSubmit}
        className="bg-white p-8 rounded-lg shadow-lg max-w-md w-full space-y-6"
        aria-label={isRegister ? "Форма реєстрації" : "Форма входу"}
      >
        <h2 className="text-3xl font-bold text-center text-gray-800">
          {isRegister ? (isCodeSent ? "Підтвердження Email" : "Реєстрація") : "Вхід"}
покупця
        </h2>
        {successMessage && (
          <p className="text-green-600 text-center text-sm font-
medium">{successMessage}</p>
        )}
        {error && (
          <p className="text-red-600 text-center text-sm font-medium">{error}</p>
        )}
        {!isCodeSent && isRegister && (
          <>
            <input
              type="text"
              name="name"
              placeholder="ПІБ"
              value={form.name}
              onChange={handleChange}
              className="w-full border border-gray-300 px-4 py-2 rounded-md"
              required
            />
            <input
              type="tel"
              name="phone"
              placeholder="Телефон"
              value={form.phone}
              onChange={handleChange}
              className="w-full border border-gray-300 px-4 py-2 rounded-md"
            />
            <input
              type="email"
              name="email"
              placeholder="Email"
              value={form.email}
              onChange={handleChange}
              className="w-full border border-gray-300 px-4 py-2 rounded-md"
              required
            />
            <input
              type="password"
              name="password"
              placeholder="Пароль"
              value={form.password}
              onChange={handleChange}
```

```
            className="w-full border border-gray-300 px-4 py-2 rounded-md"
            required
          />
        </>
      )}
      {isCodeSent && isRegister && (
        <input
          type="text"
          name="verificationCode"
          placeholder="6-значний код"
          value={verificationCode}
          onChange={(e) => setVerificationCode(e.target.value)}
          className="w-full border border-gray-300 px-4 py-2 rounded-md"
          required
        />
      )}
      {!isRegister && (
        <>
          <input
            type="email"
            name="email"
            placeholder="Email"
            value={form.email}
            onChange={handleChange}
            className="w-full border border-gray-300 px-4 py-2 rounded-md"
            required
          />
          <input
            type="password"
            name="password"
            placeholder="Пароль"
            value={form.password}
            onChange={handleChange}
            className="w-full border border-gray-300 px-4 py-2 rounded-md"
            required
          />
        </>
      )}
      <button
        type="submit"
        className="w-full bg-blue-600 hover:bg-blue-700 text-white py-3 rounded-md font-
semibold"
      >
        {isRegister ? (isCodeSent ? "Підтвердити" : "Зареєструватися") : "Увійти"}
      </button>
      <p className="text-center text-sm text-gray-600">
        {isRegister ? "Вже маєте акаунт?" : "Немає акаунту?"} {" "}
        <button
          type="button"
          className="text-blue-600 hover:underline"
          onClick={() => {
            setIsRegister(!isRegister);
            setError("");
            setSuccessMessage("");
            setIsCodeSent(false);
          }}
        >
          {isRegister ? "Увійти" : "Зареєструватися"}
        </button>
```

```
        </p>
      </form>
    </div>
  );
}
```

Файл CheckoutPage.jsx

```
import { useEffect, useState } from "react";
      import { useNavigate } from "react-router-dom";
      import axios from "axios";
      import { axiosInstance } from "../../lib/axios";
      import Select from "react-select";
      export default function CheckoutPage() {
        const [customer, setCustomer] = useState(null);
        const [paymentMethod, setPaymentMethod] = useState("cash");
        const [city, setCity] = useState("Київ");
        const [cityOptions, setCityOptions] = useState([]);
        const [selectedCity, setSelectedCity] = useState(null);
        const [warehouses, setWarehouses] = useState([]);
        const [selectedWarehouse, setSelectedWarehouse] = useState(null);
        const [isProcessingPayment, setIsProcessingPayment] = useState(false);
        const token = localStorage.getItem("customerToken");
        const navigate = useNavigate();
        useEffect(() => {
          if (token) {
            axiosInstance
              .get("/customerAuth/customer/me", {
                headers: { Authorization: `Bearer ${token}` },
              })
              .then((res) => setCustomer(res.data.customer))
              .catch(() => alert("Помилка отримання даних користувача"));
          }
        }, [token]);
        useEffect(() => {
          if (city.trim()) {
            axiosInstance
              .get("/novaposhta/cities", { params: { name: city } })
              .then((res) => {
                const options = res.data.map((c) => ({ label: c.label, value: c.value
}));
                setCityOptions(options);
              })
              .catch(() => setCityOptions([]));
          }
        }, [city]);
        useEffect(() => {
          if (selectedCity?.value) {
            axiosInstance
              .get("/novaposhta/warehouses", {
                params: { cityRef: selectedCity.value },
              })
              .then((res) => {
                const options = res.data.map((w) => ({ value: w.value, label: w.label
}));
                setWarehouses(options);
              })
```

```
      .catch(() => alert("Не вдалося завантажити відділення"));
  }
}, [selectedCity]);
const handleSubmitOrder = async () => {
  if (!selectedWarehouse || !selectedCity) {
    alert("Оберіть місто та відділення Нової Пошти");
    return;
  }
  try {
    const { data: cartRes } = await axiosInstance.get("/cart", {
      headers: { Authorization: `Bearer ${token}` },
    });
    const items = cartRes.cart.map((item) => ({
      title: item.product.title,
      quantity: item.quantity,
      price: item.product.price,
    }));
    const totalPrice = items.reduce((sum, i) => sum + i.quantity * i.price, 0);
    if (paymentMethod === "online") {
      setIsProcessingPayment(true);
      localStorage.setItem(
        "pendingOrder",
        JSON.stringify({
          customerName: customer.name,
          customerEmail: customer.email,
          customerPhone: customer.phone,
          items,
          totalPrice,
          paymentMethod: "online",
          deliveryMethod: "Нова пошта",
          novaPoshtaOffice: selectedWarehouse.label,
          city: selectedCity.label,
          isPaid: true,
        })
      );
      const { data: paymentData } = await axiosInstance.post(
        "/liqpay/create-payment",
        {
          amount: totalPrice,
          orderInfo: `Замовлення від ${customer.name}`,
          currency: "UAH",
          returnUrl: "http://localhost:3000/checkout/success",
          failUrl: "http://localhost:3000/checkout/fail",
        },
        { headers: { Authorization: `Bearer ${token}` } }
      );
      return (window.location.href = paymentData.paymentUrl);
    }

    await axiosInstance.post(
      "/orders",
      {
        customerName: customer.name,
        customerEmail: customer.email,
        customerPhone: customer.phone,
        items,
        totalPrice,
        paymentMethod,
        deliveryMethod: "Нова пошта",
```

```
            novaPoshtaOffice: selectedWarehouse.label,
            city: selectedCity.label,
            isPaid: false,
          },
          {
            headers: { Authorization: `Bearer ${token}` },
          }
        );
        await axiosInstance.post("/cart/clear", null, {
          headers: { Authorization: `Bearer ${token}` },
        });
        alert("Замовлення успішно створено!");
        navigate("/");
      } catch (err) {
        console.error(err);
        alert("Не вдалося створити замовлення");
      } finally {
        setIsProcessingPayment(false);
      }
    };
    return (
      <div className="p-6 max-w-xl mx-auto bg-white shadow-lg rounded-lg mt-8 mb-16">
        <h1 className="text-3xl font-semibold mb-6 text-center">Підтвердження
замовлення</h1>
        {customer ? (
          <form className="space-y-6" onSubmit={(e) => e.preventDefault()}>
            <div>
              <label className="block mb-1 font-medium text-gray-700">ПІБ:</label>
              <input
                type="text"
                value={customer.name}
                disabled
                className="w-full border border-gray-300 rounded-md px-4 py-2 bg-
gray-100"
              />
            </div>
            <div>
              <label className="block mb-1 font-medium text-gray-
700">Телефон:</label>
              <input
                type="text"
                value={customer.phone || ""}
                disabled
                className="w-full border border-gray-300 rounded-md px-4 py-2 bg-
gray-100"
              />
            </div>

            <div>
              <label className="block mb-1 font-medium text-gray-700">Спосіб
оплати:</label>
              <div className="flex flex-col gap-3">
                <label className="flex items-center gap-2">
                  <input
                    type="radio"
                    name="paymentMethod"
                    value="online"
                    checked={paymentMethod === "online"}
                    onChange={() => setPaymentMethod("online")}
```

```
                    />
                    <span>Оплата онлайн</span>
                  </label>
                  <label className="flex items-center gap-2">
                    <input
                      type="radio"
                      name="paymentMethod"
                      value="cash"
                      checked={paymentMethod === "cash"}
                      onChange={() => setPaymentMethod("cash")}
                    />
                    <span>Оплата при отриманні</span>
                  </label>
                </div>
              </div>
              <div>
                <label className="block mb-1 font-medium text-gray-700">Місто:</label>
                <Select
                  options={cityOptions}
                  value={selectedCity}
                  onInputChange={(inputValue) => setCity(inputValue)}
                  onChange={(option) => {
                    setSelectedCity(option);
                    setCity(option.label);
                  }}
                  placeholder="Введіть назву міста"
                  isSearchable
                  className="text-gray-700"
                  classNamePrefix="select"
                  noOptionsMessage={() => "Місто не знайдено"}
                />
              </div>
              <div>
                <label className="block mb-1 font-medium text-gray-700">Відділення
Нової Пошти:</label>
                <Select
                  options={warehouses}
                  value={selectedWarehouse}
                  onChange={setSelectedWarehouse}
                  placeholder="Оберіть відділення"
                  isSearchable
                  className="text-gray-700"
                  classNamePrefix="select"
                />
              </div>
              <button
                type="button"
                onClick={handleSubmitOrder}
                disabled={isProcessingPayment}
                className={`w-full mt-6 py-3 rounded-md font-semibold text-white ${
                  isProcessingPayment ? "bg-green-400 cursor-not-allowed" : "bg-green-
600 hover:bg-green-700"
                } transition`}
              >
                {isProcessingPayment ? "Перенаправлення на оплату..." : "✅ Підтвердити
замовлення"}
              </button>
            </form>
          ) : (
```

```
          <p className="text-center text-gray-600">Завантаження даних…</p>
        )}
      </div>
    );
  }
```