МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА ТОРГОВЕЛЬНОГО МАЙДАНЧИКУ З ВИКОРИСТАННЯМ RUBY ON RAILS»**

Виконав: студент 2 курсу, групи 8.1214-іпз-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
А.А. Горевий
(ініціали та прізвище)

Керівник завідувач кафедри програмної інженерії, доцент, к.ф.-м.н. Лісняк А.О.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент доцент кафедри комп'ютерних наук, доцент, к.т.н. Решевська К.С.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Горевому Артуру Андрійовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка торговельного майданчику з використанням Ruby on Rails

керівник роботи Лісняк Андрій Олександрович, к.ф.-м.н., доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 28 » лютого 2025 року № 326-с

2. Строк подання студентом роботи 03.12.2025 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Технічне завдання.
2. Проєктування.
3. Реалізація та тестування.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 07.03.2025 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	14.03.2025	
2.	Збір вихідних даних.	11.04.2025	
3.	Обробка методичних та теоретичних	06.05.2025	
	джерел.		
4.	Розробка першого та другого розділу.	19.08.2025	
5.	Розробка третього розділу.	24.11.2025	
6.	Оформлення та нормоконтроль	01.12.2025	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	17.12.2025	

Студент _____ А.А. Горевий
(підпис) (ініціали та прізвище)

Керівник роботи _____ А.О. Лісняк
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка торговельного майданчику з використанням Ruby on Rails»: 87 с., 54 рис., 2 табл., 22 джерела.

БАЗА ДАНИХ, МАРКЕТПЛЕЙС, BACK-END, FRONT-END, JAVA SCRIPT, MVC, POSTGRESQL, REACTJS, REST API, RUBY, RUBY ON RAILS.

Об’єкт дослідження: розробка торговельного майданчику за допомогю Ruby on rails та React JS.

Мета роботи: розробити торговельний майданчик за допомогю Ruby on rails та React JS.

Методи дослідження: моделювання, проєктування, програмний, аналітичний.

Результати та особливості застосунку: реалізовано стек Ruby on Rails + React + PostgreSQL, підтримано реєстрацію, створення магазину та додавання товарів. Доступний каталог із пошуком, фільтрами й сортуванням, а також кошик для авторизованих (збереження в БД) і гостей (localStorage) з автосинхронізацією після входу. Оформлення замовлення інтегровано з API Нової пошти для пошуку адрес і передбачає генерацію унікального токена для перегляду статусу без авторизації. Оплата через LiqPay реалізована з повним циклом (payload, redirect, callback), зарахуванням на баланс магазину та обліком транзакцій. Налаштовані сповіщення про зміну статусів і реалізовані кабінети продавця та адміністратора.

Переваги рішення: масштабованість і безпека завдяки MVC/REST, окремому фронтенду та API-інтеграціям, а також бізнес-фокус через модулі балансу, транзакцій, категорій і сповіщень для підтримки операційних процесів.

SUMMARY

Master’s qualifying paper “Development of a Trading Platform Using Ruby on Rails”: 87 pages, 54 figures, 2 tables, 22 references.

BACK-END, DATABASE, FRONT-END, JAVA SCRIPT, MARKETPLACE, MVC, POSTGRESQL, REACTJS, REST API, RUBY, RUBY ON RAILS.

The object of the study is the process of development of a trading platform using Ruby on Rails and React JS.

The aim of the study is to develop a trading platform using Ruby on Rails and React JS.

Research methods: modeling, projecting, programming, analytical.

The application delivers a Ruby on Rails + React + PostgreSQL stack with end-to-end core functionality. It supports user registration, store creation, and product onboarding. The solution provides a catalog with search, filtering, and sorting, plus a shopping cart for authorized users (persisted in the database) and guests (localStorage) with automatic synchronization after login. The checkout flow is integrated with the Nova Poshta API for address lookup and generates a unique token to let users track order status without authentication. LiqPay payments are implemented as a full cycle, including payload generation, redirect, and callback handling, with funds credited to the store balance and transactions recorded for accounting. Status change notifications are configured, and both seller and administrator accounts are available.

The solution offers scalability and security through an MVC/REST architecture, a separated frontend, and controlled API integrations. It also supports business operations via balance management, transaction tracking, category management, and notification modules.

ЗМІСТ

Завдання на кваліфікаційну роботу……..2
Реферат 4
Summary 5
Вступ..... 8
1 Технічне завдання 10
1.1 Опис системи..... 10
1.2 Аналіз існуючих рішень 11
1.3 Загальні вимоги 12
1.3.1 Функціональні вимоги..... 12
1.3.2 Нефункціональні вимоги..... 15
1.3.3 Архітектурні вимоги..... 16
1.3.4 Вимоги до безпеки 16
2 Проєктування..... 18
2.1 Діаграма прецедентів..... 18
2.2 Діаграма послідовності..... 23
2.3 Діаграма компонентів..... 26
2.4 Діаграма розгортання 29
2.5 Діаграма класів..... 31
3 Реалізація та тестування 36
3.1 Опис інструментів розробки 36
3.2 Проєктування структури бази даних 37
3.3 Основні класи системи 41
3.4 Реалізація серверної частини торговельного майданчику..... 43
3.4.1 Налаштування бази даних торговельного майданчику 45
3.4.2 Налаштування серверу торговельного майданчику..... 47
3.4.3 Розробка моделей для наших сутностей 47
3.4.5 Функціонал кошику та перенесення з localStorage 50

3.4.6 Система замовлень 52
3.4.7 Інтеграція API Нової Пошти 55
3.4.8 Інтеграція платіжної системи LiqPay 58
3.5 Реалізація клієнтської частини торговельного майданчику 63
3.5.1 Розробка компонента кошику 66
3.5.2 Розробка компонента замовлення 68
3.5.3 Використання API Нової Пошти на клієнті 71
3.5.4 Використання платіжної системи LiqPay на клієнті 73
3.6 Приклади роботи програмного забезпечення 75
3.6.1 Функціонал для ролі «Покупець» 77
3.6.2 Функціонал для ролі «Продавець» 81
3.6.3 Функціонал для ролі «Адміністратор» 84
Висновки 85
Перелік посилань 86

ВСТУП

Електронна комерція є важливим сегментом сучасної економіки, а торговельні онлайн-майданчики – ефективним інструментом взаємодії між продавцями та покупцями. Відомі приклади таких систем – Prom.ua, Rozetka, Olx – демонструють необхідність швидких, зручних і масштабованих рішень для ведення бізнесу онлайн.

Предметною областю є розробка торговельного онлайн-майданчику, що забезпечує купівлю й продаж товарів через вебплатформу. Користувачі можуть зареєструватися, увійти в систему та працювати з основними розділами: каталог товарів, кошик, замовлення, особистий кабінет. Для початку продажів користувач створює магазин у розділі «Особистий кабінет», після чого його роль змінюється на продавця.

Продавець керує магазином у розділі «Мій магазин» та створює товари через форму з полями: назва, опис, ціна, залишок, категорія, зображення. Після збереження товар автоматично з’являється в каталозі. Покупці переглядають каталог, використовують пошук, фільтри й сортування, додають товари в кошик. Для авторизованих користувачів кошик зберігається в базі даних, для неавторизованих – у localStorage з подальшою синхронізацією після входу.

Під час оформлення замовлення користувач заповнює форму (ім’я, email, телефон, адреса доставки) з можливістю використання API Нової пошти для вибору міста та відділення. Система створює замовлення з унікальним номером, надає можливість оплати через LiqPay, а для неавторизованих генерує токен доступу. Після успішної оплати кошти зараховуються на баланс магазину; продавець бачить баланс, історію транзакцій і може ініціювати виведення коштів. Система формує сповіщення про зміну статусів замовлень. Адміністратор має інструменти для керування обліковими записами, ієрархією категорій та перегляду статистики системи.

Метою цієї роботи є створення вебзастосунку торговельного майданчику на базі Ruby on Rails, який дозволяє користувачам реєструвати

магазини, додавати товари, працювати з категоріями та виконувати базові торгові операції.

Для досягнення поставленої мети необхідно виконати такі завдання:

– проаналізувати предметну область та визначити основні процеси;
– дослідити існуючі аналоги та підходи, виявити їх переваги та недоліки;
– сформувати функціональні та нефункціональні вимоги;
– спроєктувати базу даних, описати структуру даних та взаємозв’язки;
– реалізувати серверну та клієнтську частини вебдодатку відповідно до попереднього проєктування.

Об’єкт дослідження – процес розробки торговельного онлайн-майданчику.

Предмет дослідження – методи побудови вебзастосунку з підтримкою ролей та управлінням товарами.

Практична цінність проєкту полягає у створенні гнучкої основи для подальшого розвитку торговельного функціоналу та інтеграції з іншими сервісами.

1 ТЕХНІЧНЕ ЗАВДАННЯ

1.1 Опис системи

Онлайн-торгівля є ключовою та невід’ємною частиною сучасного бізнесу. Однією з основних категорій електронної комерції є торговельні майданчики, які об’єднують покупців та продавців на одній платформі.

Зазвичай продавці створюють окремі інтернет-магазини, що призводить до багатьох проблем, наприклад: висока вартість розробки та підтримки власного сайту, складність привернення покупців, відсутність уніфікованої системи оплати та доставки, необхідність індивідуальної інтеграції з платіжними системами та службами доставки тощо. У зв’язку з цим велику актуальність набуває розробка систем, які дозволяють продавцям легко публікувати товари, а покупцям – зручно їх знаходити та купувати.

Торговельні майданчики стають невід’ємною частиною в забезпеченні ефективної взаємодії між покупцями та продавцями. Знайдеться небагато бізнесів, котрі б не користувалися послугами таких систем. За допомогою них продавці мають можливість публікувати товари, керувати замовленнями, отримувати оплату та виводити кошти, а покупці – переглядати каталог товарів, додавати їх до кошика, оформлювати замовлення з автоматичною інтеграцією доставки та оплати, відстежувати статуси замовлень та отримувати сповіщення.

Однією з таких систем є «Торговельний майданчик». Нова система для ведення онлайн-торгівлі. Розглянемо можливості системи. До основних слід віднести:

– керування магазинами та товарами (створення, редагування, видалення);
– перегляд каталогу товарів з пошуком, фільтрацією за категоріями, ціною, магазинами та сортуванням;

– управління кошиком (для авторизованих та неавторизованих користувачів);
– оформлення замовлень з інтеграцією API Нової Пошти для вибору доставки;
– оплата замовлень через платіжну систему LiqPay;
– відстеження статусів замовлень та отримання сповіщень;
– фінансова система для продавців (баланс, історія транзакцій, виведення коштів).

Консоль адміністратора:

– керування обліковими записами користувачів;
– керування ієрархічною структурою категорій товарів.

1.2 Аналіз існуючих рішень

Prom.ua – український B2C/B2B торговельний майданчик для малого та середнього бізнесу. Дозволяє створювати магазини, додавати товари, використовувати просування.

Сильні сторони: масштабованість, маркетинг-інструменти, підтримка.

Слабкі сторони: платна підписка, обмежена кастомізація.

Rozetka.ua – найбільший український торговельний майданчик із власним складом, доставкою та партнерською платформою.

Сильні сторони: довіра бренду, інтеграція доставки та оплати.

Слабкі сторони: закритість платформи, висока конкуренція серед продавців.

OLX.ua – класифайд-платформа для оголошень у форматі C2C. Товари публікуються напряму без магазинів.

Сильні сторони: простота, швидке розміщення, велике охоплення.

Слабкі сторони: відсутність магазинів, обмежений контроль товарів.

Etsy.com – глобальний майданчик для продажу товарів ручної роботи, вінтажу та крафтових виробів.

Сильні сторони: нішевий ринок, потужний пошук, аналітика.

Слабкі сторони: висока конкуренція, комісії з кожного продажу.

Amazon.com – найбільший у світі e-commerce майданчик. Доступний для масового ринку, має власну логістику та партнерську мережу.

Сильні сторони: логістика, довіра, Prime-модель.

Слабкі сторони: перенасиченість, складна модерація, дорогі комісії.

eBay.com – глобальна платформа для торгів та прямих продажів. Дає змогу продавати як нові, так і вживані товари.

Сильні сторони: підтримка аукціонів, доступ до глобального ринку.

Слабкі сторони: складність виводу коштів, високі комісії.

Shopify.com – конструктор для запуску індивідуальних онлайн-магазинів з можливістю інтеграції в торговельні майданчики.

Сильні сторони: висока кастомізація, готові шаблони.

Слабкі сторони: потребує окремого трафіку, платна основа.

1.3 Загальні вимоги

Опираючись на аналіз існуючих рішень розробляємо загальні вимоги для торговельного майданчику. Загальні вимоги для розробки торговельного майданчику можна розділити на кілька категорій: функціональні вимоги, нефункціональні вимоги, архітектурні вимоги та вимоги до безпеки.

1.3.1 Функціональні вимоги

Функціональні вимоги визначають основні можливості та поведінку торговельного майданчику, які забезпечують його роботу та взаємодію з користувачами. Вони охоплюють повний цикл роботи системи: від реєстрації користувачів та управління контентом до оформлення замовлень, оплати та

фінансових операцій. Нижче наведено детальний опис функціональних вимог, організований за основними модулями системи.

Реєстрація та авторизація користувачів:

– можливість реєстрації нових користувачів;
– авторизація користувачів з використанням email та пароля;
– система ролей: покупець, продавець, адміністратор;
– автоматичне створення сесії після успішної авторизації;
– можливість виходу з системи з очищенням сесії.

Управління магазинами:

– можливість створення магазину для ролі «покупець»;
– автоматична зміна ролі «покупець» на «продавець» після створення магазину;
– можливість перегляду інформації про магазин;
– обмеження: один користувач може мати лише один магазин.

Управління товарами:

– можливість створення, редагування та видалення товарів для продавців;
– завантаження множинних зображень товарів через Active Storage;
– встановлення ціни товару;
– додавання опису товару;
– прив’язка товарів до категорій (ієрархічна структура);
– прив’язка товарів до магазину продавця;
– можливість перегляду детальної інформації про товар;
– пошук товарів за назвою;
– фільтрація товарів за категорією, ціною, магазином;
– сортування товарів (за ціною, датою створення).

Система категорій:

– ієрархічна структура категорій (батьківські та дочірні категорії);
– можливість створення, редагування та видалення категорій для адміністраторів;
– фільтрація товарів за категорією (включаючи підкатегорії).

Кошик покупця:

– можливість додавання товарів до кошика;
– збереження кошика в базі даних для авторизованих користувачів;
– збереження кошика в localStorage для неавторизованих користувачів;
– автоматичне перенесення товарів з localStorage в базу даних при авторизації;
– можливість зміни кількості товарів у кошику;
– можливість видалення товарів з кошика;
– заборона додавання власних товарів до кошика (для продавців).

Оформлення замовлення:

– можливість оформлення замовлення з кошика;
– введення даних доставки (ім’я, прізвище, телефон, email);
– пошук населеного пункту через API Нової Пошти;
– вибір відділення або поштомату через API Нової Пошти;
– можливість оформлення замовлення для неавторизованих користувачів з генерацією access_token;
– збереження стану замовлення;
– можливість перегляду замовлення за access_token (для неавторизованих).

Платіжна система:

– інтеграція з платіжною системою LiqPay;
– генерація платіжної форми для оплати замовлення;
– обробка callback від LiqPay для підтвердження оплати;
– автоматичне оновлення статусу замовлення після успішної оплати;
– нарахування коштів на гаманець продавця після оплати замовлення.

Гаманець продавця та виплати:

– відображення балансу гаманця продавця;
– можливість налаштування реквізитів для виплат;
– можливість створення запиту на виплату коштів;
– історія транзакцій (нарахування від продажів, виплати).

Сповіщення:

– система сповіщень для користувачів та продавців;
– відображення кількості непрочитаних сповіщень;
– можливість позначення сповіщень як прочитаних;
– можливість позначення всіх сповіщень як прочитаних.

Адміністративна панель:

– перегляд списку всіх користувачів;
– управління категоріями товарів.

1.3.2 Нефункціональні вимоги

Нефункціональні вимоги визначають якісні характеристики системи, які впливають на її експлуатацію, продуктивність та користувацький досвід, але не описують конкретні функції. Вони охоплюють аспекти масштабованості, продуктивності, зручності використання, надійності та сумісності, що забезпечують ефективну роботу торговельного майданчику в реальних умовах експлуатації.

Нижче наведено опис нефункціональних вимог.

– масштабованість: система має підтримувати велику кількість користувачів, товарів та замовлень;
– продуктивність: швидкий час завантаження сторінок та відповідей на API-запити;
– зручність користування: інтуїтивно зрозумілий інтерфейс користувача з адаптивним дизайном;
– надійність: забезпечення цілісності даних та обробка помилок;
– сумісність: підтримка сучасних браузерів (Chrome, Firefox, Edge).

1.3.3 Архітектурні вимоги

Архітектурні вимоги визначають технологічний стек, структурні принципи побудови та організацію компонентів системи. Вони забезпечують узгоджену архітектуру, що відповідає сучасним підходам до розробки вебдодатків та сприяє підтримуваності, масштабованості та розширюваності системи.

Нижче наведено опис архітектурних вимог.

Стек технологій:

– Frontend: React;
– Backend: Ruby on Rails;
– база даних: PostgreSQL;
– зберігання файлів: Active Storage;
– Web server: Puma.

Архітектурні принципи:

– RESTful API для комунікації між фронтендом та бекендом;
– розділення відповідальності: окремі контролери для сутностей;
– використання Context API для глобального стану;
– компонентна архітектура на фронтенді;
– використання міграцій для управління схемою бази даних.

1.3.4 Вимоги до безпеки

Вимоги до безпеки визначають механізми захисту системи та даних користувачів від несанкціонованого доступу, втрати інформації та інших загроз. Вони охоплюють аутентифікацію та авторизацію користувачів, захист конфіденційних даних, контроль доступу до ресурсів та валідацію вхідних даних, що забезпечують безпечну експлуатацію торговельного майданчику.

Нижче наведено перелік вимог до безпеки:

– аутентифікація та авторизація: використання cookie-based сесій для аутентифікації користувачів;
– захист даних: шифрування паролів за допомогою bcrypt (has_secure_password);
– контроль доступу: перевірка прав доступу на рівні контролерів (authenticate_user!, authenticate_admin!);
– валідація даних: перевірка вхідних даних на рівні моделей та контролерів;
– безпечне зберігання файлів: використання Active Storage з хешованими шляхами для уникнення передбачуваних URL;
– захист API: перевірка авторизації для захищених ендпоінтів.

2 ПРОЄКТУВАННЯ

Попереднє проєктування застосунку перед стартом розробки є одним із ключових факторів успішної реалізації будь-якого ІТ-проєкту. На цьому етапі формується цілісне бачення кінцевого продукту, фіксуються основні функціональні та нефункціональні вимоги, а також заздалегідь ідентифікуються потенційні ризики та проблемні зони.

У першу чергу, якісне проєктування мінімізує ризик непорозумінь між членами команди та стейкхолдерами, оскільки забезпечує єдине трактування цілей, очікувань і меж проєкту. Це підвищує прозорість комунікації, знижує потребу в доопрацюваннях на пізніх етапах життєвого циклу розробки та, відповідно, скорочує додаткові витрати ресурсів.

Unified Modeling Language (UML) – це уніфікована мова моделювання, тобто стандартна візуальна нотація, яка дає змогу описувати структуру та поведінку програмних і бізнес-систем. Вона належить до загальновживаних (general-purpose) мов моделювання і використовується для специфікації, візуалізації, конструювання та документування артефактів програмного забезпечення, а також для моделювання бізнес-процесів у різних доменах [5].

2.1 Діаграма прецедентів

Діаграма прецедентів фокусується на зовнішній поведінці системи з точки зору акторів. Вона оперує двома базовими сутностями: актором (Actor), який представляє роль користувача чи зовнішньої системи, що взаємодіє із застосунком, та прецедентом (Use Case), який описує окремий сценарій використання – цінну для актора функцію або послугу. Така діаграма показує, «що» повинна робити система для різних категорій користувачів, не заглиблюючись у технічні деталі реалізації.

Виходячи з аналізу існуючих рішень та побудови загальних вимог маємо наступних акторів системи: покупець (неавторизований), покупець (авторизований), продавець, адміністратор. Для визначених акторів побудуємо діаграми використання (див. рис. 2.1 – 2.3).

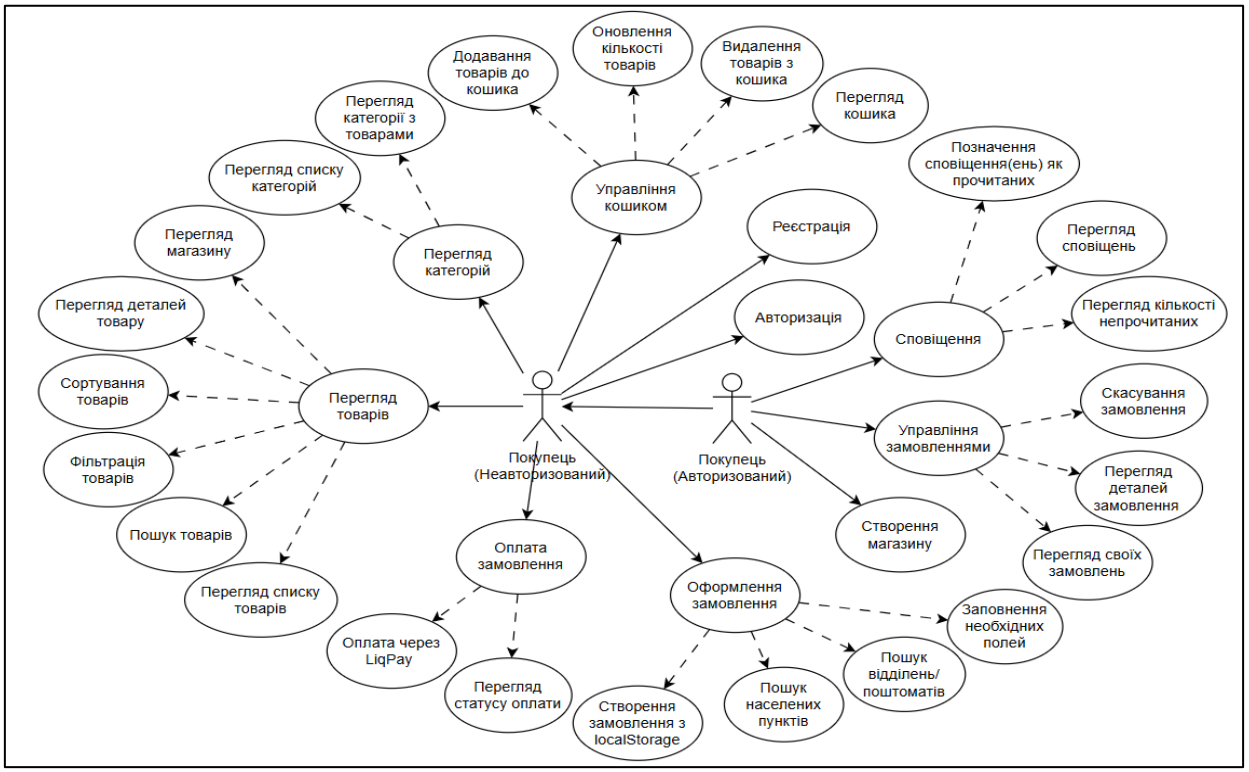


Рисунок 2.1 – Діаграма варіантів використання для актора «Покупець»

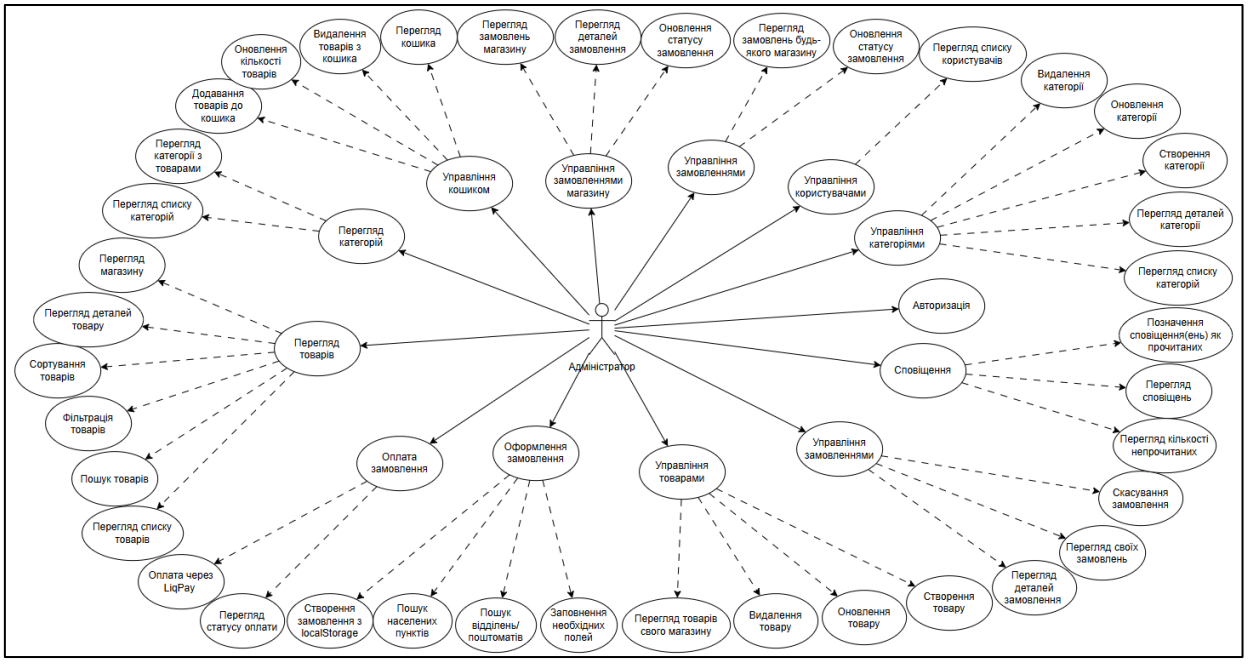


Рисунок 2.2 – Діаграма варіантів використання для актора «Адміністратор»

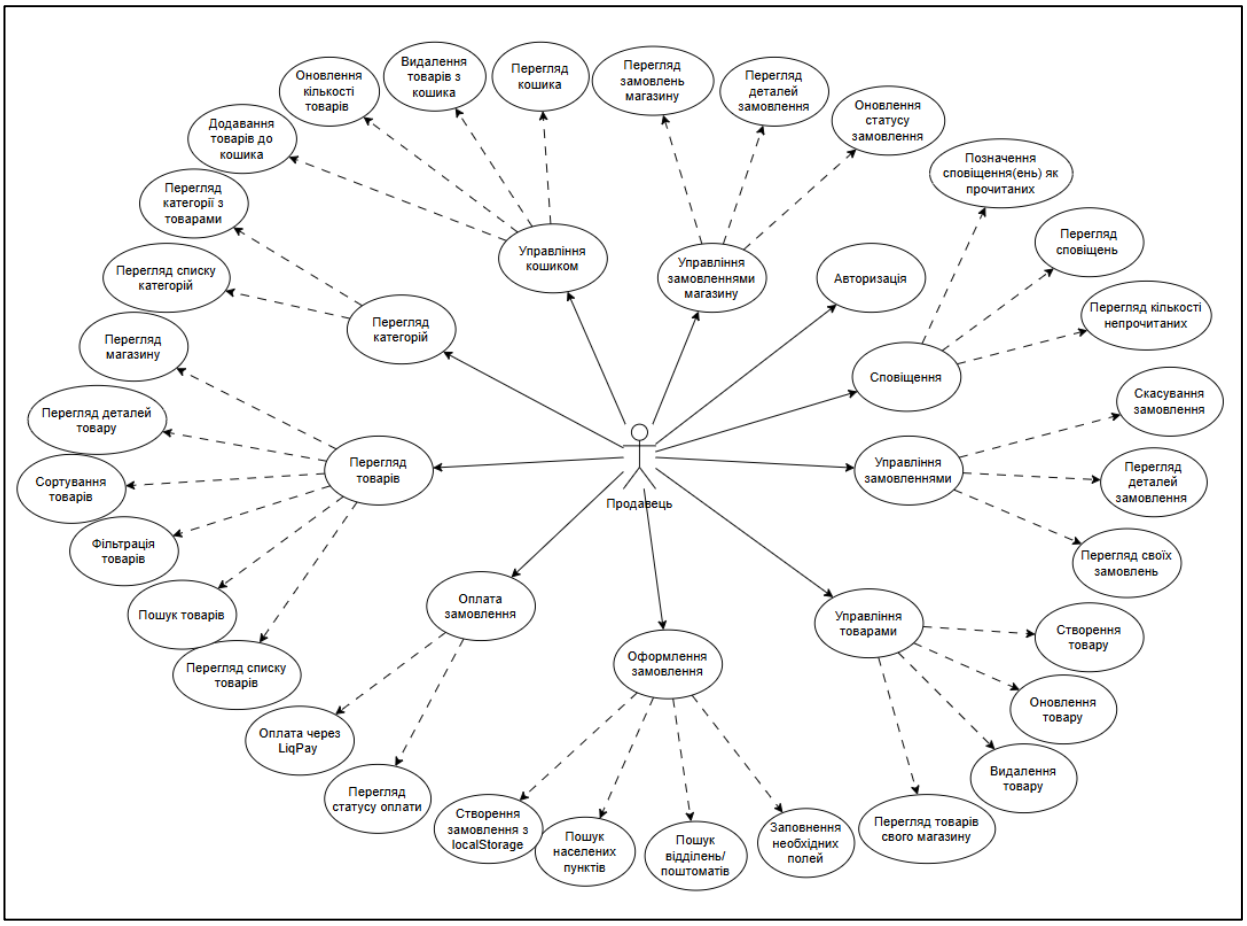


Рисунок 2.3 – Діаграма варіантів використання для актора «Продавець»

Прецедент «Оформлення замовлення».

Призначення: дозволяє користувачу (авторизованому або неавторизованому) оформити замовлення з товарів кошика, вказати дані доставки та контактну інформацію.

Основний потік подій: користувач переходить до оформлення замовлення з кошика. Система відображає форму з полями: ім’я, email, телефон, адреса доставки. Користувач може використати пошук населених пунктів та відділень Нової Пошти через інтеграцію з API. Після заповнення та підтвердження система створює замовлення з унікальним номером. Для неавторизованих користувачів генерується access_token для доступу до замовлення. Система створює сповіщення для покупця та продавця про нове замовлення.

Альтернативний потік подій: якщо кошик порожній, система показує повідомлення про відсутність товарів. Якщо дані валідації не пройдено,

система відображає помилки. Якщо користувач неавторизований, система створює замовлення без user_id та надає access_token для подальшого доступу.

Передумова: у кошику є товари для оформлення замовлення.

Прецедент «Створення товару».

Призначення: дозволяє продавцю додати новий товар до свого магазину з описом, ціною, зображеннями та категорією.

Основний потік подій: продавець переходить до створення товару в панелі управління магазином. Система відображає форму з полями: назва, опис, ціна, залишок на складі, категорія, зображення. Після заповнення та підтвердження система зберігає товар у базі даних.

Альтернативний потік подій: якщо валідація не пройдена (наприклад, відсутня назва або ціна), система відображає помилки. Якщо завантаження зображень не вдалося, система інформує про помилку, але товар може бути збережено без зображень.

Передумова: користувач має роль seller або admin та має створений магазин.

Прецедент «Оплата замовлення через LiqPay».

Призначення: дозволяє користувачу (авторизованому або неавторизованому) оплатити замовлення через платіжну систему LiqPay.

Основний потік подій: користувач переходить до сторінки замовлення та натискає кнопку «Оплатити». Система перевіряє доступ до замовлення (для авторизованих – перевірка user_id або store_id, для неавторизованих – перевірка access_token). Система генерує унікальний liqpay_order_id, формує payload з сумою, описом, result_url та server_url, створює підпис (signature) для безпеки. Система повертає дані для форми оплати LiqPay. Користувач виконує оплату на стороні LiqPay. Після оплати LiqPay надсилає callback на server_url. Система перевіряє підпис, декодує дані, оновлює статус оплати замовлення та автоматично нараховує кошти на баланс магазину через StoreTransaction типу accrual.

Альтернативний потік подій: якщо доступ до замовлення заборонено, система повертає помилку 403 Forbidden. Якщо сума замовлення дорівнює

нулю, система відображає помилку. Якщо підпис у callback не співпадає, система відхиляє запит. Якщо оплата не вдалася, система не оновлює статус замовлення та не нараховує кошти на баланс.

Передумова: існує замовлення зі статусом “pending” та сумою більше нуля. Користувач має доступ до замовлення.

Прецедент «Управління кошиком».

Призначення: дозволяє користувачу додавати товари до кошика, змінювати кількість, видаляти товари та переносити кошик з localStorage в базу даних після авторизації.

Основний потік подій: користувач переглядає товари та натискає «Додати до кошика». Якщо користувач авторизований, система створює або оновлює CartItem у базі даних (якщо товар вже є – збільшує quantity). Якщо користувач неавторизований, система зберігає товар у localStorage. При перегляді кошика система відображає товари з БД (для авторизованих) або з localStorage (для неавторизованих). Після авторизації система автоматично переносить товари з localStorage в БД через endpoint transfer_to_user, фільтруючи товари власного магазину користувача, та очищає localStorage.

Альтернативний потік подій: якщо користувач намагається додати власний товар (як продавець), система блокує дію. Якщо товар не знайдено під час переносу, система пропускає його та продовжує з іншими товарами.

Передумова: користувач переглядає товари на сайті. Для авторизованих користувачів – успішна авторизація в системі.

Прецедент «Перегляд та фільтрація товарів».

Призначення: дозволяє користувачу переглядати список товарів з можливістю пошуку, фільтрації за категоріями, ціною, магазином та сортування.

Основний потік подій: користувач відкриває головну сторінку або сторінку категорії. Система відображає список товарів з базовою інформацією (назва, ціна, зображення, магазин). Користувач може використати пошук за назвою (через параметр query), застосувати фільтри: категорія (включаючи

підкатегорії через self_and_descendant_ids), мінімальна/максимальна ціна, магазин. Користувач може вибрати сортування: за ціною (зростання/спадання) або за датою створення. Система виконує запит до ProductsController з параметрами фільтрації та повертає відфільтрований список товарів.

Альтернативний потік подій: якщо товарів не знайдено за критеріями, система відображає повідомлення «Товари не знайдено». Якщо категорія не існує, система ігнорує фільтр за категорією та показує всі товари. Якщо параметри фільтрації некоректні (наприклад, від'ємна ціна), система ігнорує некоректні параметри.

Передумова: передумови відсутні – функціонал доступний всім користувачам, включаючи неавторизованих.

2.2 Діаграма послідовності

Діаграма послідовності відображає динамічну взаємодію об'єктів у часі – які саме повідомлення надсилаються, в якій черговості та між якими учасниками. Вона деталізує окремі прецеденти, показує порядок виклику операцій і допомагає перевірити узгодженість логіки роботи системи з описаними вимогами [6]. На рисунках 2.4 та 2.5 наведено діаграми послідовностей для основних процесів торгового майданчику.

Опис діаграми послідовності процесу «Пошук адреси доставки». Процес розділено на два етапи: пошук населеного пункту та пошук відділення/поштомату. Це спрощує вибір для користувача та зменшує кількість варіантів на кожному кроці.

Етап 1: пошук населеного пункту. Користувач вводить назву міста, що дозволяє швидко знайти потрібний населений пункт без повного списку. Клієнтська частина відправляє запит на сервер, який виконує пошук через API Нової Пошти. Використання сервера як проксі забезпечує безпеку API-ключа та дозволяє кешувати результати. Серверна частина відправляє запит до Nova

Poshta API з параметрами пошуку, отримує список населених пунктів та повертає їх клієнту у форматі JSON. Клієнтська частина відображає список міст користувачу для вибору.

Етап 2: пошук відділень/поштоматів. Після вибору міста користувач обирає тип доставки (відділення або поштомат), що дозволяє фільтрувати результати за потреби. Клієнтська частина відправляє запит на пошук відділень з параметрами міста та типу доставки. Серверна частина виконує пагінацію для отримання всіх відділень міста, фільтрує їх за типом та повертає клієнту. Клієнтська частина відображає список відділень для вибору.

Двокроковий процес спрощує вибір для користувача та зменшує навантаження на систему. Використання сервера як проксі забезпечує безпеку API-ключа та дозволяє кешувати результати для підвищення продуктивності. Асинхронна обробка запитів дозволяє користувачу продовжувати роботу під час пошуку. Обробка помилок на сервері забезпечує коректну роботу навіть при проблемах з API Нової Пошти.

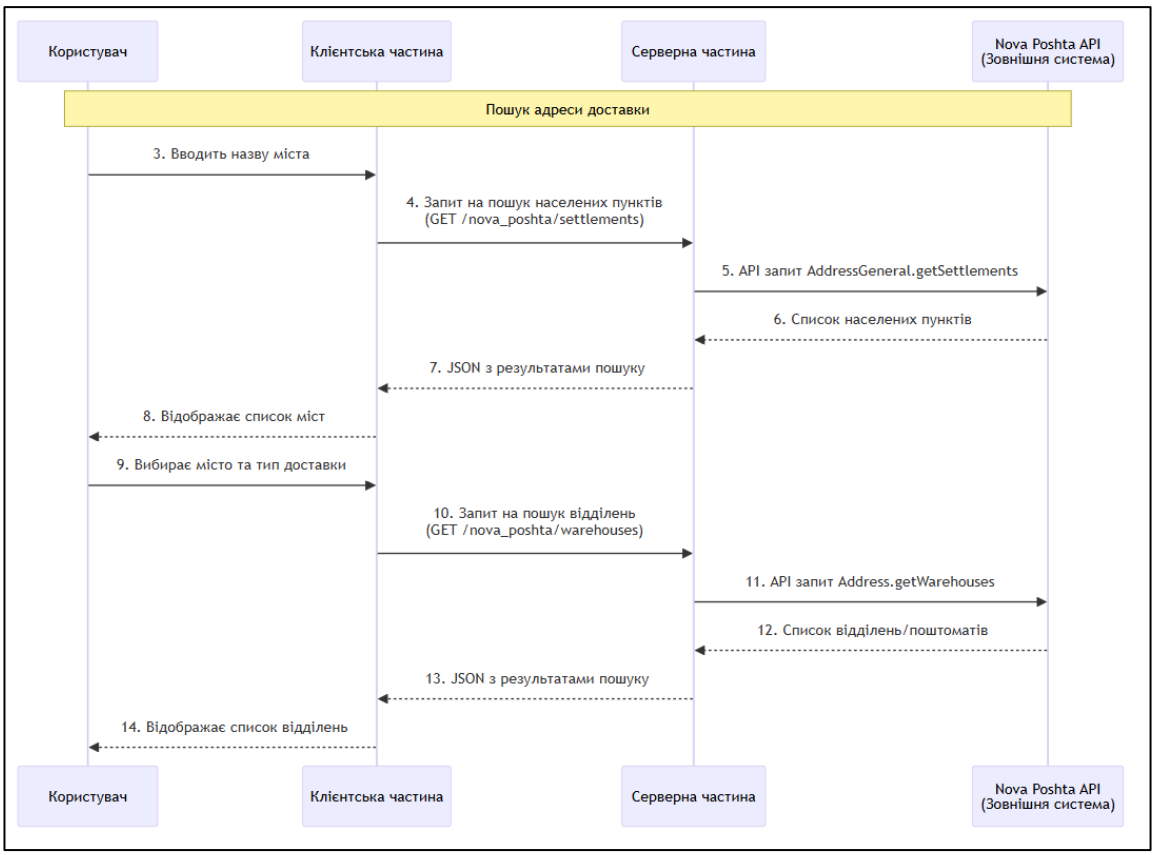


Рисунок 2.4 – Діаграма послідовності процесу «Пошук адреси доставки»

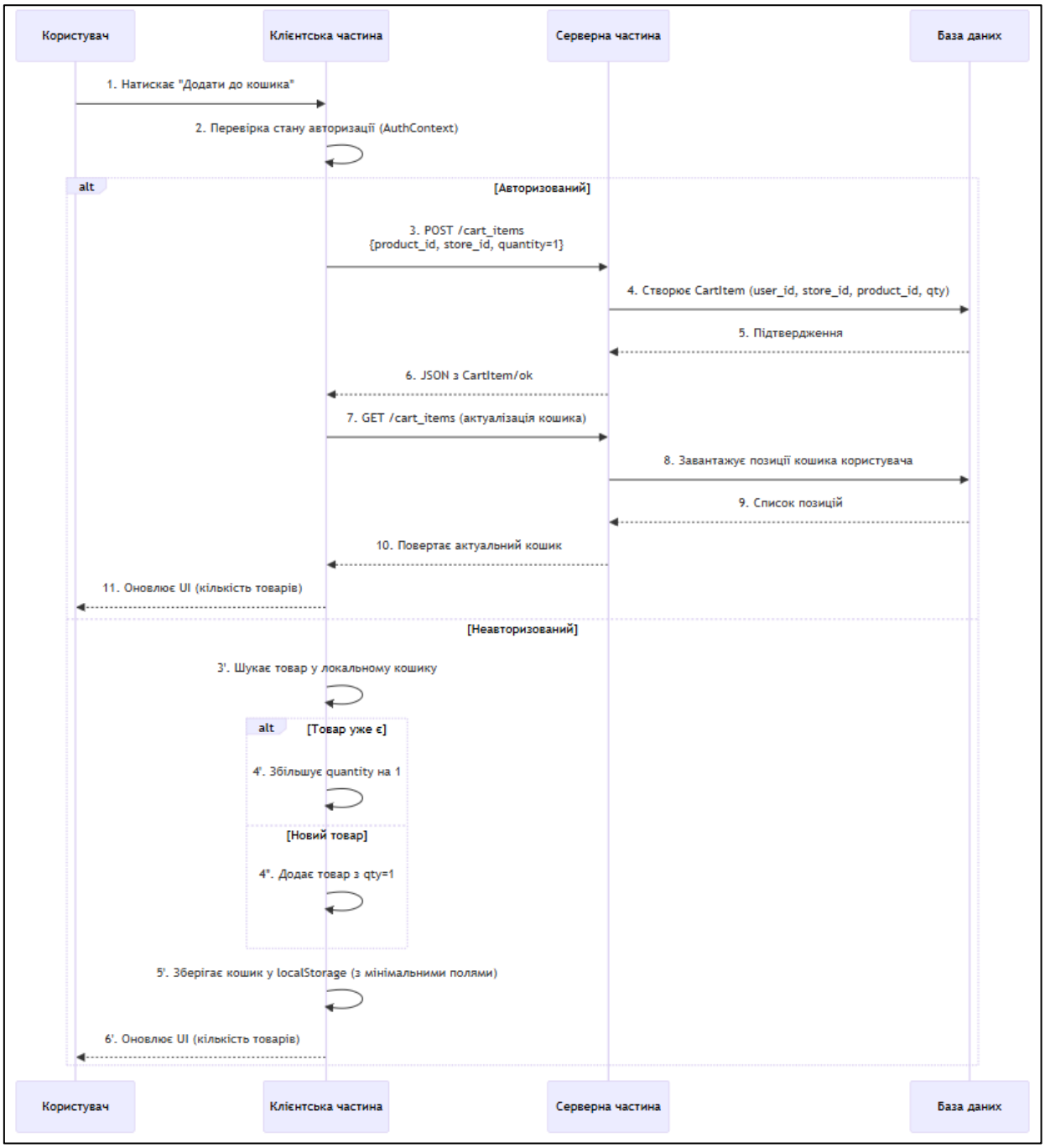


Рисунок 2.5 – Діаграма послідовності процесу «Додавання товару в кошик»

Опис діаграми послідовності процесу «Додавання товару в кошик». Процес підтримує два сценарії залежно від стану авторизації. Це дозволяє неавторизованим користувачам додавати товари без реєстрації, знижуючи бар’єр для покупок.

Сценарій для авторизованого користувача. Користувач натискає кнопку «Додати до кошика», система перевіряє авторизацію через AuthContext. Клієнтська частина відправляє POST запит на сервер з даними

товару. Серверна частина створює запис CartItem в базі даних з прив’язкою до користувача, що забезпечує синхронізацію кошика між пристроями. Після створення клієнтська частина запитує актуальний стан кошика з сервера для синхронізації та відображення оновленої кількості товарів у інтерфейсі.

Сценарій для неавторизованого користувача. Користувач натискає кнопку «Додати до кошика», система визначає відсутність авторизації. Клієнтська частина перевіряє наявність товару в локальному кошику (localStorage). Якщо товар вже є, збільшується його кількість; якщо новий – додається з кількістю 1. Клієнтська частина зберігає кошик у localStorage, використовуючи мінімальні дані (без зображень та великих полів) для оптимізації обсягу зберігання. Інтерфейс оновлюється з оновленою кількістю товарів у кошику.

Підтримка двох сценаріїв покращує користувацький досвід. Авторизовані користувачі мають синхронізований кошик між пристроями через базу даних, що забезпечує доступ до кошика з будь-якого пристрою. Неавторизовані користувачі можуть додавати товари без реєстрації, що зменшує бар’єр для покупок та збільшує конверсію. При авторизації кошик з localStorage автоматично переноситься в базу даних, що забезпечує безперервність досвіду користувача. Використання мінімальних даних в localStorage оптимізує використання сховища браузера та забезпечує швидку роботу з кошиком.

2.3 Діаграма компонентів

Діаграма компонентів відображає статичну архітектуру програмної системи – які саме компоненти її утворюють, як вони організовані та через які інтерфейси взаємодіють. Вона описує фізичну структуру системи, фіксує залежності між модулями й точки інтеграції та дає змогу перевірити узгодженість архітектурних рішень і розподілу відповідальностей із вимогами

до системи [6]. На рисунку 2.6 наведено діаграму компонентів торгового майданчику.

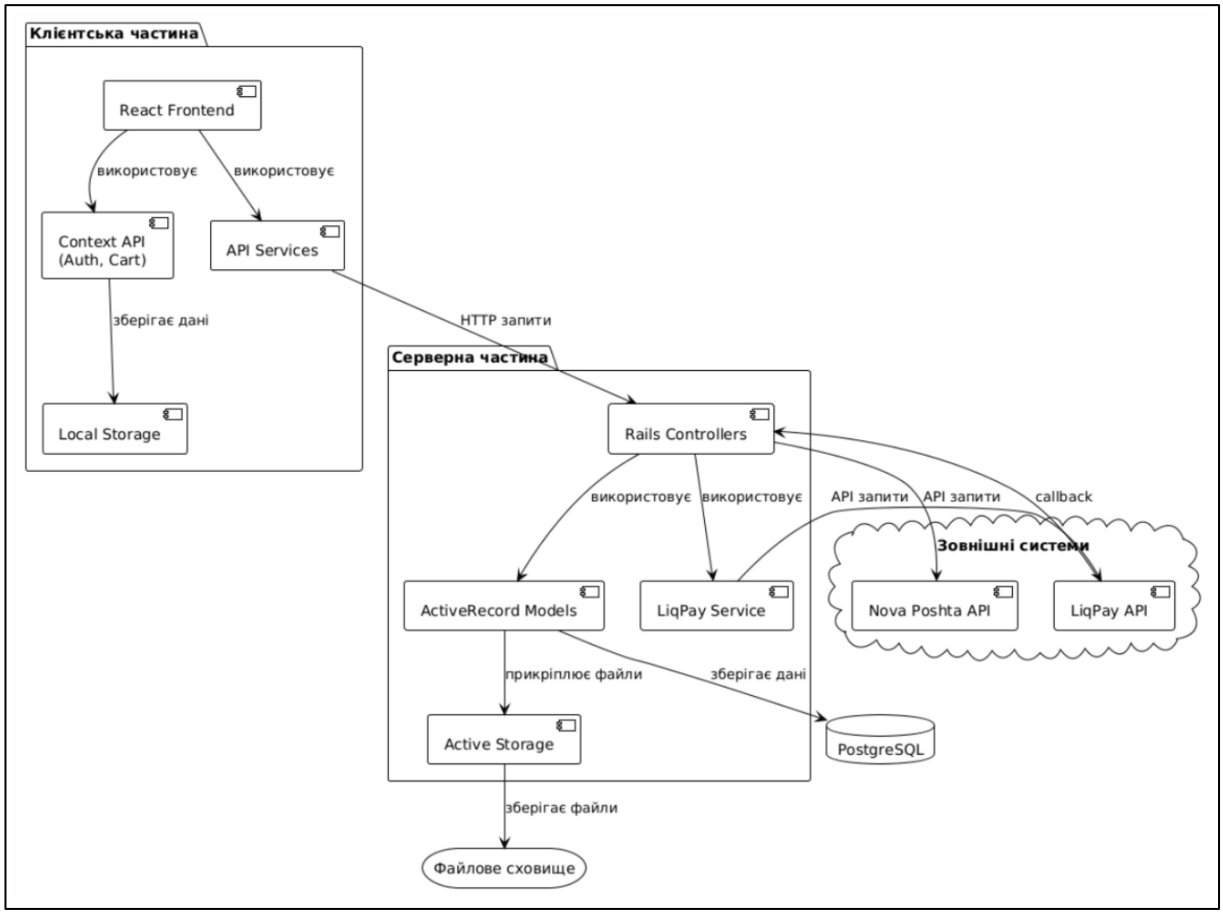


Рисунок 2.6 – Діаграма компонентів

Розподіл відповідальності між пакетами. Клієнтська частина містить компоненти, що виконуються в браузері. React Frontend відповідає за UI та взаємодію з користувачем. Context API зберігає глобальний стан (авторизація, кошик), що дозволяє компонентам отримувати актуальні дані без пропсів. API Services інкапсулює HTTP-запити до сервера, що спрощує зміни в API без змін у компонентах. Local Storage зберігає кошик для неавторизованих користувачів, щоб вони могли додавати товари до авторизації.

Серверна частина містить бізнес-логіку та обробку даних. Rails Controllers обробляють HTTP-запити, виконують валідацію та авторизацію, координують процеси. ActiveRecord Models містять бізнес-правила, валідації та зв’язки між сутностями. LiqPay Service виділено для інтеграції з платіжною

системою, що дозволяє змінювати провайдера без змін у контролерах. Active Storage керує файлами, зокрема зображеннями товарів.

Зв'язки та потоки даних. Frontend > Context > LocalStorage – потік збереження стану: Frontend використовує Context для доступу до стану авторизації та кошика. Context зберігає кошик у LocalStorage для неавторизованих користувачів, щоб зберегти кошик між сесіями.

Frontend > APIServices > Controllers – основний потік обробки запитів: Frontend відправляє запити через APIServices, які інкапсулюють HTTP-логіку та формат даних. Controllers обробляють запити, виконують валідацію та авторизацію перед передачею до Models.

Controllers > Models > DB – потік роботи з даними: Controllers використовують Models для бізнес-логіки та валідації. Models зберігають дані в PostgreSQL через ORM, що забезпечує безпеку та спрощує роботу з даними.

Controllers > LiqPayService > LiqPay API – процес оплати: Controllers використовують LiqPayService для генерації payload та signature. LiqPayService відправляє запити до LiqPay API, який обробляє платежі та повертає callback з результатами.

Models > ActiveStorage > FileStorage – потік роботи з файлами: Models прикріплюють файли через ActiveStorage, який керує зберіганням та доступом до файлів. ActiveStorage зберігає файли у FileStorage, що дозволяє масштабувати зберігання файлів незалежно від БД.

Controllers > NovaPoshta API – пошук адрес доставки: Controllers відправляють запити до Nova Poshta API для пошуку населених пунктів та відділень. Виконується на сервері для безпеки API-ключа та можливості кешування результатів.

LiqPay > Controllers (callback) – зворотний зв'язок від платіжної системи: LiqPay відправляє callback на Controllers з результатами оплати. Controllers оновлюють статус замовлення та нараховують кошти продавцю.

Така архітектура підтримує основні бізнес-процеси торговельного майданчика. Розділення на клієнтську та серверну частини дозволяє

масштабувати компоненти незалежно та зберігати безпеку даних. Виділення Context API забезпечує централізоване управління станом, що спрощує синхронізацію між компонентами. Виділення API Services дозволяє змінювати формат комунікації з сервером без змін у компонентах. Виділення LiqPay Service дозволяє інтегрувати інші платіжні системи без змін у контролерах. Виділення Active Storage дозволяє змінювати спосіб зберігання файлів (локальний диск, S3, Cloudinary) без змін у моделях.

2.4 Діаграма розгортання

Діаграма розгортання показує фізичну архітектуру системи та розподіл програмних компонентів по вузлах (nodes) та артефактах (artifacts). Вона оперує двома основними сутностями: вузлом (Node), який представляє фізичний або логічний обчислювальний ресурс (сервер, пристрій, контейнер), та артефактом (Artifact), який описує фізичну реалізацію програмного компонента (виконуваний файл, бібліотека, база даних). Така діаграма показує, «де» та «як» розгорнута система, відображаючи фізичну інфраструктуру та зв’язки між компонентами в середовищі виконання [6].

На рисунку 2.7 наведено діаграму розгортання торгового майданчику.

Розподіл відповідальності між вузлами. Клієнтський пристрій містить React Frontend, оскільки UI виконується в браузері користувача. Це зменшує навантаження на сервер, дозволяє швидко реагувати на дії користувача та працювати неавторизованим для частини функцій (наприклад, кошик у Local Storage).

Web Server зосереджує серверну логіку в одному вузлі. Rails Application координує роботу компонентів: Controllers обробляють HTTP-запити, Models містять бізнес-правила, LiqPay Service інкапсулює логіку оплати, Active Storage керує файлами. Така організація спрощує розгортання та підтримку.

Database Server виділено окремо для масштабування та безпеки. PostgreSQL зберігає критичні дані (користувачі, замовлення, товари) та забезпечує транзакційність операцій.

File Storage виділено для зображень товарів, оскільки файли потребують іншого підходу до зберігання та обслуговування, ніж структуровані дані.

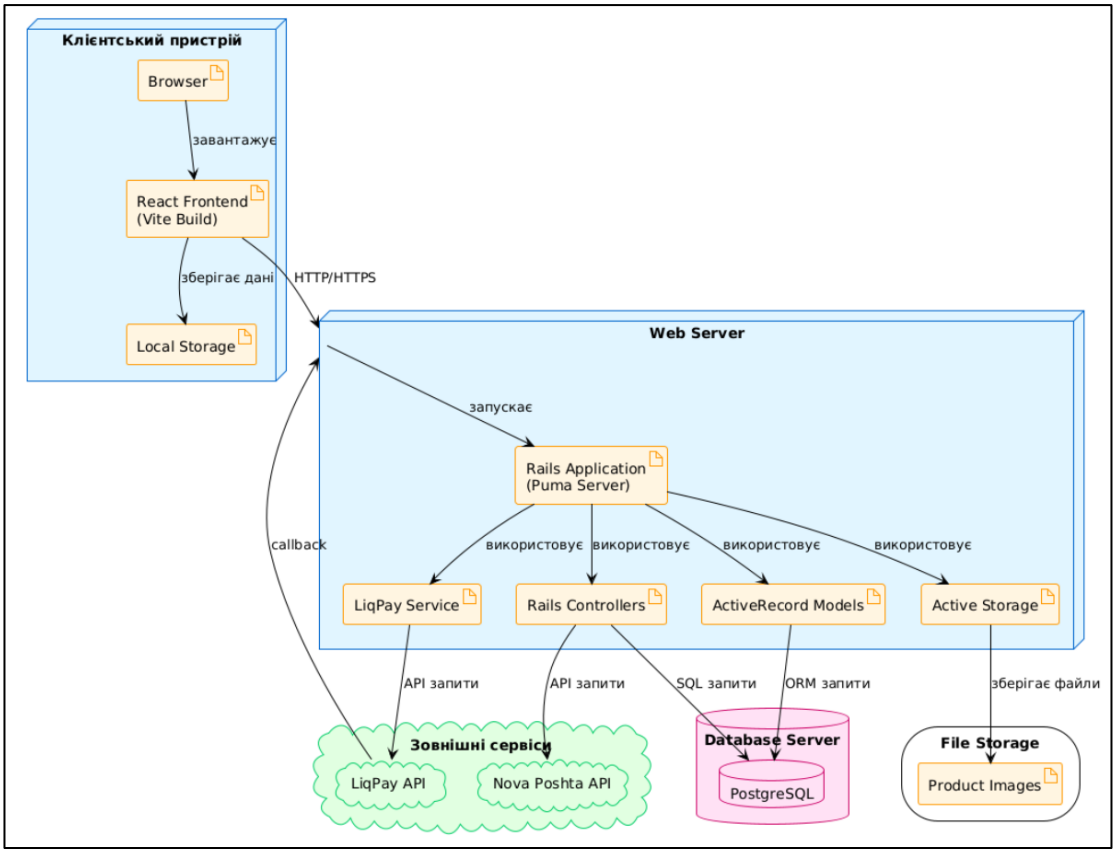


Рисунок 2.7 – Діаграма розгортання

Зв’язки та потоки даних. Browser > ReactApp > WebServer – основний потік: користувач взаємодіє з UI, який відправляє запити на сервер. Використовується HTTP/HTTPS для безпеки та сумісності з мережею.

ReactApp > LocalStorage – збереження кошика для неавторизованих користувачів. Дозволяє зберігати кошик між сесіями без необхідності авторизації.

RailsApp > Controllers > Models > PostgreSQL – обробка запитів: Controllers валідують та авторизують, Models виконують бізнес-логіку, дані зберігаються в БД. ORM спрощує роботу з даними та забезпечує безпеку.

RailsApp > LiqPayService > LiqPay API – процес оплати: сервіс готує дані для оплати, відправляє запит до LiqPay, отримує callback з результатом. Виділення сервісу дозволяє змінювати платіжні провайдери без змін в основних контролерах.

Controllers > NovaPoshta API – пошук адрес доставки. Виконується на сервері для безпеки API-ключа та кешування результатів.

ActiveStorage > File Storage – зберігання зображень товарів. Файли зберігаються окремо від БД для продуктивності та масштабованості.

LiqPay > WebServer (callback) – зворотний зв’язок від платіжної системи. Callback приходить на сервер для оновлення статусу замовлення та нарахування коштів продавцю.

Така архітектура підтримує основні бізнес-процеси: перегляд товарів, додавання до кошика, оформлення замовлень, оплату та доставку. Розділення на вузли дозволяє масштабувати компоненти незалежно: додавати сервери додатків, окремо масштабувати БД та файлове сховище. Виділення зовнішніх сервісів дозволяє інтегрувати сторонні послуги без змін основної системи.

2.5 Діаграма класів

На рисунку 2.8 наведено діаграму класів торговельного майданчика. Для діаграми класів торговельного майданчика створимо детальний опис в таблиці 2.1 з класами, атрибутами та методами, створеними для розробки соцмережі.

Таблиця 2.1 – Опис класів торговельного майданчика

Параметр	**Значення**
Category	
Атрибути	id: integer – ідентифікатор категорії name: string – назва категорії parent_id: integer – ідентифікатор батьківської категорії

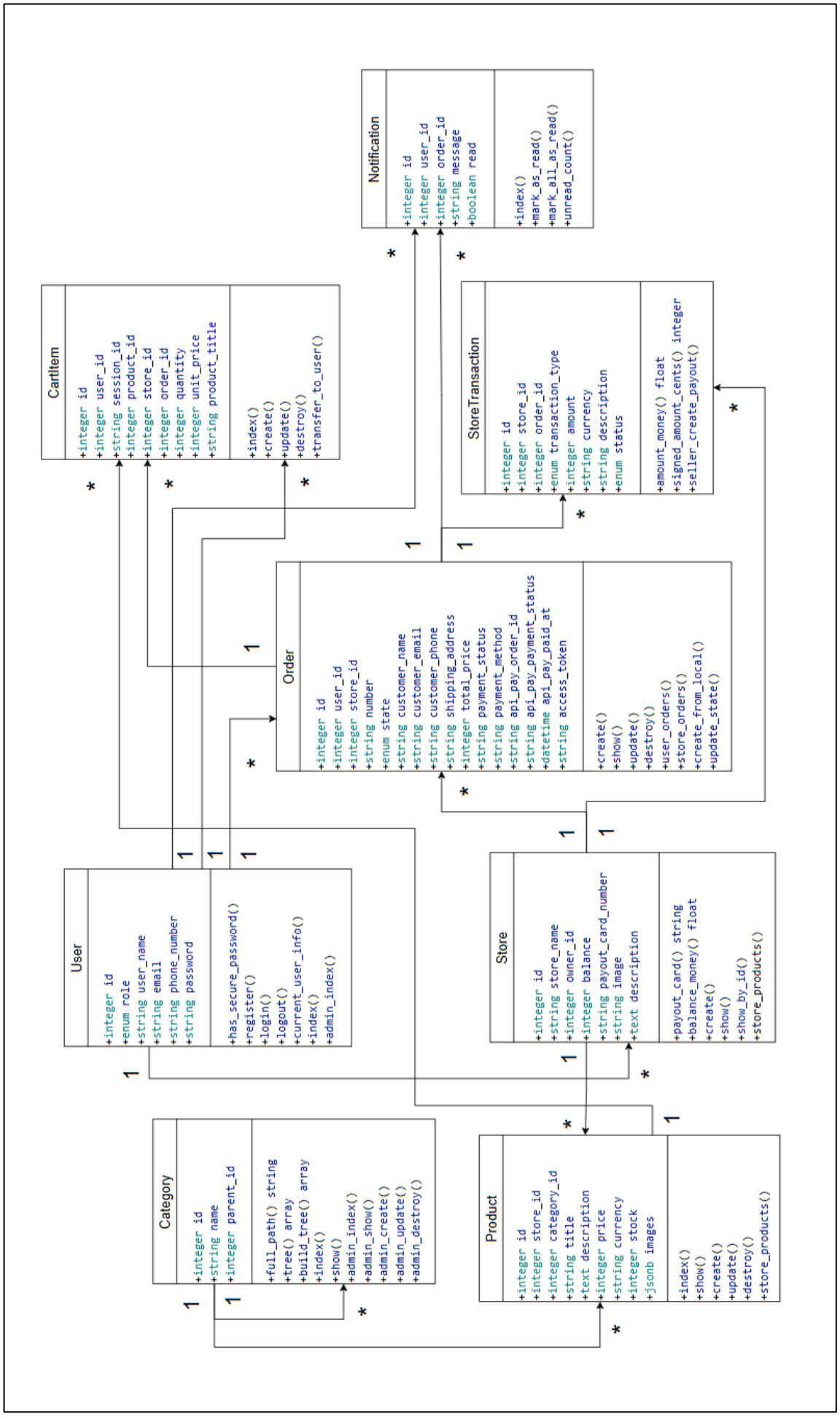


Рисунок 2.8 – Діаграма класів торговельного майданчика

Продовження таблиці 2.1

Параметр	Значення
Category	
Методи	full_path() – повний шлях категорії через ієрархію tree() – отримати всі категорії у вигляді дерева build_tree() – побудувати дерево категорій з вкладеними підкатегоріями index() – список всіх категорій show() – отримати категорію за id admin_index() – список всіх категорій для адміністратора admin_show() – отримати категорію за id для адміністратора admin_create() – створити нову категорію (тільки адмін) admin_update() – оновити категорію (тільки адмін) admin_destroy() – видалити категорію (тільки адмін)
User	
Атрибути	id – ідентифікатор користувача role – роль користувача (buyer, seller, admin) user_name – ім’я користувача email – електронна пошта phone_number – номер телефону password – пароль
Методи	has_secure_password() – метод для безпечного зберігання пароля register() – реєстрація нового користувача login() – авторизація користувача logout() – вихід з системи current_user_info() – отримати інформацію про поточного користувача index() – список користувачів admin_index() – список всіх користувачів для адміністратора
Product	
Атрибути	id – ідентифікатор товару store_id – ідентифікатор магазину, якому належить товар category_id – ідентифікатор категорії товару title – назва товару description – опис товару price – ціна товару currency – валюта stock – кількість товару на складі images – зображення товару
Методи	index() – список товарів show() – отримати товар за id з детальною інформацією create() – створити новий товар (тільки продавець) update() – оновити товар (тільки власник магазину) destroy() – видалити товар (тільки власник магазину) store_products() – список товарів конкретного магазину

Продовження таблиці 2.1

Параметр	Значення
Store	
Атрибути	id – ідентифікатор магазину store_name – назва магазину owner_id – ідентифікатор власника магазину balance – баланс магазину payout_card_number – номер картки для виведення коштів image – зображення магазину description – опис магазину
Методи	payout_card() – отримати замаскований номер картки для виведення balance_money() – отримати баланс у грошовому форматі create() – створити новий магазин show() – отримати інформацію про магазин show_by_id() – отримати магазин за id store_products() – список товарів магазину
Order	
Атрибути	id – ідентифікатор замовлення user_id – ідентифікатор користувача store_id – ідентифікатор магазину number – номер замовлення state – статус замовлення customer_name – ім’я клієнта customer_email – електронна пошта клієнта customer_phone – телефон клієнта shipping_address – адреса доставки total_price – загальна сума замовлення payment_status – статус оплати payment_method – спосіб оплати api_pay_order_id – ідентифікатор замовлення в платіжній системі api_pay_payment_status – статус оплати в платіжній системі api_pay_paid_at – дата та час оплати access_token – токен доступу для перегляду замовлення без авторизації
Методи	create() – створити нове замовлення з кошика show() – отримати замовлення за id або access_token update() – оновити дані замовлення destroy() – видалити замовлення user_orders() – список замовлень користувача store_orders() – список замовлень магазину create_from_local() – створити замовлення з локального сховища (для неавторизованих) update_state() – оновити статус замовлення

Продовження таблиці 2.1

Параметр	Значення
CartItem	
Атрибути	id – ідентифікатор позиції user_id – ідентифікатор користувача session_id – ідентифікатор сесії (для неавторизованих користувачів) product_id – ідентифікатор товару store_id – ідентифікатор магазину order_id – ідентифікатор замовлення quantity – кількість товару unit_price – ціна за одиницю товару product_title – назва товару
Методи	index() – список позицій кошика create() – додати товар до кошика update() – оновити кількість товару в кошику destroy() – видалити товар з кошика transfer_to_user() – перенести товари з кошика сесії до кошика користувача після авторизації
StoreTransaction	
Атрибути	id – ідентифікатор транзакції store_id – ідентифікатор магазину order_id – ідентифікатор замовлення transaction_type – тип транзакції amount – сума транзакції currency – валюта description – опис транзакції status – статус транзакції
Методи	amount_money() – отримати суму транзакції signed_amount_cents() – отримати суму зі знаком seller_create_payout() – створити запит на виведення коштів
Notification	
Атрибути	id – ідентифікатор сповіщення user_id – ідентифікатор користувача order_id – ідентифікатор замовлення message – текст сповіщення read – чи прочитано сповіщення
Методи	index() – список сповіщень поточного користувача mark_as_read() – позначити сповіщення як прочитане mark_all_as_read() – позначити всі сповіщення як прочитані unread_count() – кількість непрочитаних сповіщень

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

3.1 Опис інструментів розробки

Ruby – це динамічна об’єктно-орієнтована мова програмування з простим, лаконічним синтаксисом, що дає змогу писати компактний і читабельний код. Вона підтримує гнучке розширення вбудованих класів і має активну спільноту та зрілу екосистему бібліотек і фреймворків (зокрема Ruby on Rails), що робить Ruby зручним інструментом для швидкої розробки сучасних застосунків [2].

Ruby on Rails – це фреймворк вебзастосунків з відкритим кодом для мови Ruby, який реалізує архітектуру Model–View–Controller та надає готову інфраструктуру для роботи з базою даних, маршрутизацією, шаблонами й бізнес-логікою. Завдяки принципам “convention over configuration” та “don’t repeat yourself” Rails суттєво прискорює створення та масштабування вебсервісів – від MVP до великих SaaS-, e-commerce- та торговельних майданчиків [1].

PostgreSQL – це потужна об’єктно-реляційна система керування базами даних з відкритим кодом, орієнтована на розширюваність, відповідність стандарту SQL та роботу з великими обсягами структурованих даних. Вона підтримує складні типи даних, транзакції з ACID, JSON і частину NoSQL-можливостей, а також індекси, збережені процедури й реплікацію, що робить PostgreSQL популярною альтернативою комерційним СУБД у високонавантажених корпоративних системах, зокрема торговельних майданчиках [10].

Puma – це високопродуктивний багатопроцесний і багатопотоковий HTTP-сервер для Rack-сумісних Ruby-застосунків (у першу чергу Ruby on Rails). Він ефективно завантажує CPU, паралельно обробляє велику кількість HTTP/1.1-запитів та є типовим вебсервером для продакшен-інфраструктури сучасних комерційних онлайн-рішень [16].

Active Storage + image_processing – стандартна зв’язка у Ruby on Rails для завантаження, зберігання та обробки файлів і зображень. Active Storage інтегрується з локальними та хмарними сховищами й прив’язує файли до моделей, а image_processing забезпечує генерацію варіантів зображень (масштабування, кадрування, прев’ю) на базі ImageMagick або libvips, що спрощує побудову галерей, мініатюр і безпечного медіа-функціоналу в вебзастосунках і торговельних майданчиках [17, 18].

React JS – це JavaScript-бібліотека для декларативної розробки користувацьких інтерфейсів, орієнтована насамперед на односторінкові вебзастосунки. Вона базується на компонентному підході, використовує віртуальний DOM для ефективного оновлення інтерфейсу та має потужну екосистему інструментів, що робить React стандартом де-факто для побудови сучасних вебінтерфейсів і SPA-рішень [3, 4].

Axios – це HTTP-клієнт на основі промісів для браузера та Node.js, який уніфікує роботу з GET, POST та іншими типами запитів до серверів і API. Він підтримує перехоплювачі, автоматичну обробку JSON, налаштування таймаутів і скасування запитів, забезпечуючи надійний шар інтеграції між фронтендом і бекенд-сервісами у вебзастосунках [19].

Vite – це сучасний інструмент для розробки та збірки фронтенд-проєктів, що поєднує надшвидкий dev-сервер на базі нативних ES-модулів із продуктивною збіркою для продакшену. Він забезпечує майже миттєвий старт, зручну гарячу заміну модулів (HMR) та має «першокласну» підтримку популярних фреймворків (Vue, React тощо) [20, 21].

3.2 Проєктування структури бази даних

Діаграма «сутність-зв’язок» (Entity Relationship Diagram, ERD) – це різновид блок-схеми, що використовується для моделювання структури даних та логічних зв’язків між ними у межах певної системи. Вона наочно показує,

як «сутності» (наприклад, люди, об’єкти, події чи абстрактні поняття) пов’язані одна з одною, і тому широко застосовується під час проєктування та аналізу реляційних баз даних у програмній інженерії, бізнес-системах, освіті та наукових дослідженнях.

ER-діаграми будуються на основі трьох ключових елементів: сутностей (entities), атрибутів (attributes) та зв’язків (relationships), а також відображають кардинальність – кількісні характеристики відносин між сутностями. На схемі зазвичай використовуються стандартизовані графічні позначення: прямокутники для сутностей, ромби або підписи на лініях для зв’язків, овали або кола для атрибутів тощо.

Сутність – це визначений об’єкт або поняття, про яке ми зберігаємо дані: клієнт, студент, товар, автомобіль, курс, замовлення. Кожна сутність має набір характеристик і принаймні один ключовий атрибут, що однозначно її ідентифікує (первинний ключ).

Атрибут – це властивість або характеристика сутності, тобто «що ми знаємо» про цей об’єкт. Наприклад, для сутності «Студент» атрибутами можуть бути прізвище, ім’я, дата народження, електронна пошта;

Зв’язок описує, як сутності взаємодіють або асоціюються між собою, і відповідає «дієслову» в описі предметної області. Наприклад, «студент записується на курс», «клієнт робить замовлення». У діаграмі зв’язки показують, які саме сутності пов’язані, та дозволяють зрозуміти бізнес-правила системи.

Таким чином, ER-діаграма дозволяє формалізувати предметну область, структурувати дані, визначити ключі, атрибути та зв’язки між сутностями ще до фізичної реалізації бази даних, що суттєво спрощує подальше проєктування й супровід системи [22].

ER-діаграма є зручним інструментом для візуалізації структури бази даних, тому для торговельного майданчику було розроблено діаграму «сутність-зв’язок».

На рисунку 3.1 наведено ER-діаграму торговельного майданчику, що містить вісім сутностей, опис яких наведено в таблиці 3.1.

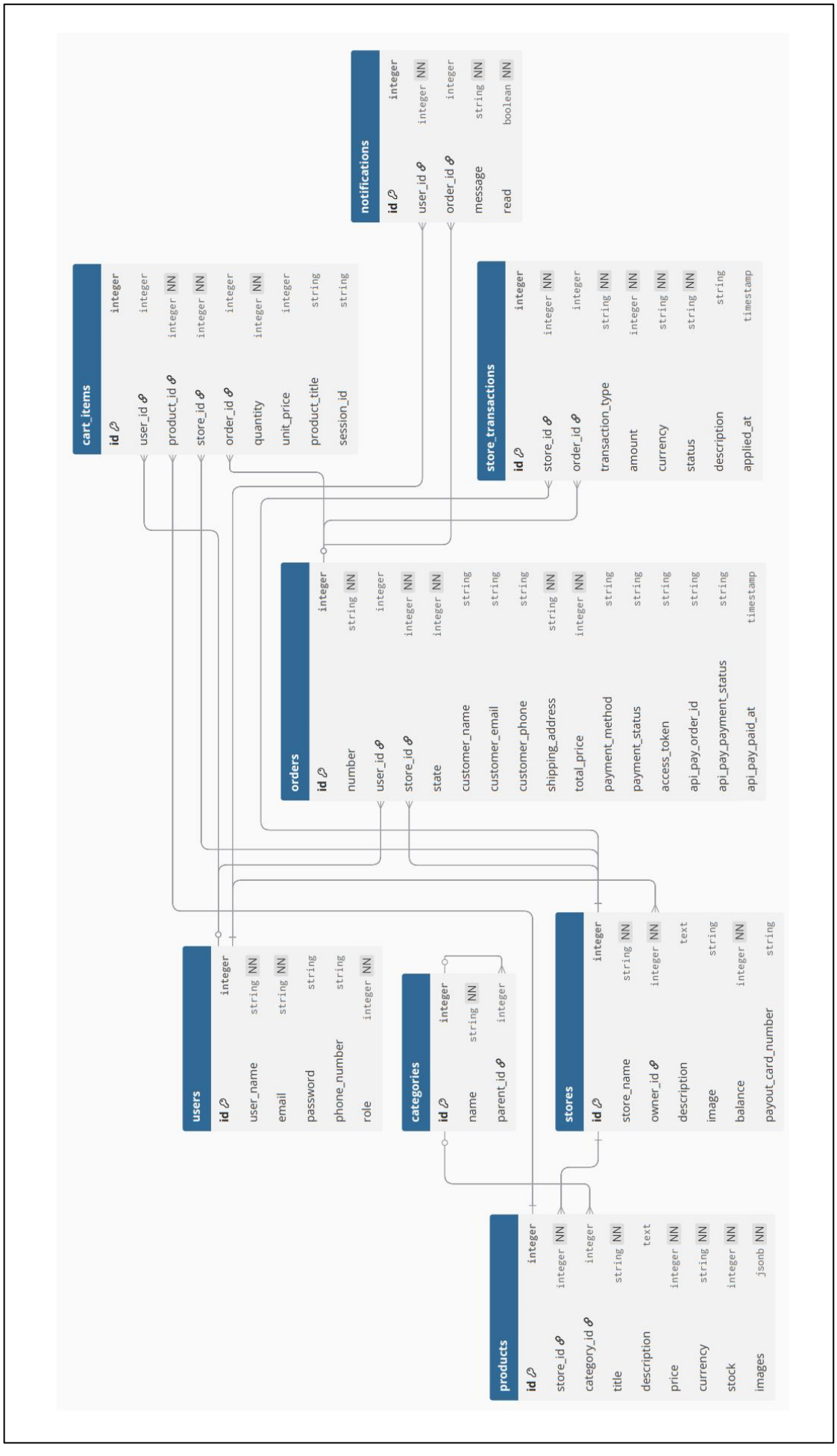


Рисунок 3.1 – Структура торговельного майданчику

Таблиця 3.1 – Опис сутностей ER моделі

Сутність 1	Сутність 2	Тип	Опис
User	Store	один-до-одного	Один користувач може мати тільки один магазин як власник, і один магазин належить тільки одному користувачу
Store	Product	один-до-багатьох	Один магазин може мати багато товарів, кожен товар належить тільки одному магазину.
Category	Product	один-до-багатьох	Одна категорія може містити багато товарів, кожен товар належить одній категорії або може не мати категорії.
Category	Category	один-до-багатьох	Самопосилання: одна категорія може мати багато підкатегорій, кожна підкатегорія має одну батьківську категорію (parent_id).
User	Order	один-до-багатьох	Один користувач може мати багато замовлень, кожне замовлення належить одному користувачу або може бути без користувача (неавторизовані).
Store	Order	один-до-багатьох	Один магазин може отримувати багато замовлень, кожне замовлення належить одному магазину.
Order	CartItem	один-до-багатьох	Одне замовлення може містити багато позицій кошика, кожна позиція кошика належить одному замовленню або не належить жодному (якщо ще просто в кошику).
Product	CartItem	один-до-багатьох	Один товар може бути доданий у кошик багато разів, кожна позиція кошика відповідає одному товару.
Store	CartItem	один-до-багатьох	Кожна позиція кошика належить одному магазину.
User	CartItem	один-до-багатьох	Один користувач може мати багато позицій у кошику, кожна позиція належить одному користувачу або може бути без користувача (через session_id для неавторизованих).
User	Notification	один-до-багатьох	Один користувач може отримувати багато сповіщень, кожне сповіщення належить одному користувачу.
Order	Notification	один-до-багатьох	Одне замовлення може генерувати багато сповіщень (наприклад, зміна статусу), кожне сповіщення може бути пов’язане з одним замовленням або не мати прив’язки.
Store	StoreTransaction	один-до-багатьох	Один магазин може мати багато транзакцій (нарахування та виплати), кожна транзакція належить одному магазину.
Order	StoreTransaction	один-до-багатьох	Одне замовлення може генерувати багато транзакцій (нарахування, виплата), кожна транзакція може бути пов’язана з одним замовленням або не мати прив’язки (загальна виплата).

3.3 Основні класи системи

Так як основними прецедентами системи є «Додати товар до кошика», «Оформити замовлення» та «Створити товар», то основними класами системи будуть ті, що надають можливість створювати, змінювати, видаляти та переглядати дані.

Головними класами зі сторони React є:

– CartContext – відповідає за управління станом кошика (містить методи для додавання товарів до кошика, оновлення кількості, видалення товарів; підтримує збереження кошика в localStorage для неавторизованих користувачів та синхронізацію з базою даних для авторизованих; містить логіку переносу товарів з localStorage в БД при авторизації);

– cartAPI – відповідає за дії, пов’язані з Rails API-сервером для роботи з кошиком (містить запити на створення, видалення, зміну та відображення даних про товари в кошику; містить endpoint для переносу кошика з localStorage в БД);

– Checkout – компонент, що обробляє та відображає форму оформлення замовлення (містить методи для управління відображенням форми введення даних доставки; отримує дані від форми та передає їх у сервіс ordersAPI; інтегрує пошук населених пунктів та відділень Нової Пошти; ініціює оплату через LiqPay);

– ordersAPI – відповідає за дії, пов’язані з Rails API-сервером для роботи з замовленнями (містить запити на створення замовлення для авторизованих та неавторизованих користувачів, оновлення даних замовлення, отримання списку замовлень, оновлення статусу замовлення, ініціацію оплати через LiqPay);

– CreateProduct – компонент, що обробляє та відображає форму створення товару (містить методи для управління відображенням форми; отримує дані від форми та передає їх у сервіс productsAPI; підтримує завантаження зображень через FormData);

– productsAPI – відповідає за дії, пов’язані з Rails API-сервером для роботи з товарами (містить запити на створення, видалення, зміну та відображення даних про товари; підтримує завантаження зображень через multipart/form-data).

Головними класами зі сторони Rails є:

– CartItem – модель, що пов’язана з таблицею «cart_items» (містить список полів для запису до таблиці, їх типи, зв’язки з іншими таблицями; містить валідації на заборону додавання власних товарів продавцем; містить scope in_cart для отримання товарів, які ще не додані до замовлення);

– CartItemsController – групує логіку обробки запитів, пов’язаних з моделлю «CartItem» (містить методи index, create, update, destroy для управління товарами в кошику; містить endpoint transfer_to_user для переносу товарів з localStorage в БД при авторизації);

– Order – модель, що пов’язана з таблицею «orders» (містить список полів для запису до таблиці, їх типи, зв’язки з іншими таблицями; містить enum для статусів замовлення: pending, accepted, fulfilled, canceled; містить методи recalc_totals! для перерахунку суми замовлення та ensure_store_accrual_recorded! для автоматичного нарахування коштів на баланс магазину після оплати);

– OrdersController – групує логіку обробки запитів, пов’язаних з моделлю «Order» (містить методи create, update, show, destroy, update_state для управління замовленнями; містить метод create_from_local для створення замовлення з даних localStorage для неавторизованих користувачів; містить методи user_orders та store_orders для отримання списків замовлень; автоматично створює сповіщення при зміні статусів замовлень);

– Product – модель, що пов’язана з таблицею «products» (містить список полів для запису до таблиці, їх типи, зв’язки з іншими таблицями; використовує Active Storage для зберігання зображень через has_many_attached :images; містить валідації наявності ключових полів та коректності зв’язків);

– ProductsController – групує логіку обробки запитів, пов’язаних з моделлю «Product» (містить методи index, show, create, update, destroy для управління товарами; підтримує фільтрацію товарів за категоріями, ціною, магазинами та сортування; обробляє завантаження зображень через Active Storage).

3.4 Реалізація серверної частини торговельного майданчику

Для початку потрібно створити середовище розробки серверної частини. Для цього створюємо в папці проєкту нову папку «server».

Після цього з використанням програми «Visual Studio Code» та попередньо встановленим Ruby на робочій машині ми ініціалізуємо проєкт за допомогою команди «rails new server --api --database=postgresql».

Команда rails new server --api --database=postgresql використовується для створення нового Rails API проєкту з підтримкою PostgreSQL. Опція --api створює проєкт без view-шаблонів, оскільки ми використовуємо React на фронтенді. Опція --database=postgresql вказує на використання PostgreSQL як бази даних замість SQLite за замовчуванням.

Тепер нам потрібно зробити інсталяцію всіх необхідних бібліотек та інструментів для розробки серверної частини за допомогою команди: «bundle install».

Дана команда встановлює наступні основні пакети в наш проєкт Ruby on Rails:

– bcrypt: це бібліотека, яка допомагає нам хешувати паролі через has_secure_password;
– rack-cors: це middleware, який дозволяє або забороняє cross-origin запити до нашого сервера;
– dotenv-rails: цей модуль завантажує змінні середовища з файлу .env;
– pg: це драйвер для роботи з PostgreSQL базою даних;

– puma: це вебсервер для запуску Rails додатку;
– active_storage: це вбудований в Rails механізм для зберігання файлів та зображень.

Файл config/application.rb є основним файлом конфігурації серверного додатку. Він використовується для налаштування Rails додатку, підключення middleware та маршрутів. Саме через нього ми налаштовуємо cookie-based сесії для автентифікації, підключаємо Active Storage для роботи з файлами та налаштовуємо API-only режим.

На рисунку 3.2 зображено вміст нашого файлу config/application.rb.

```ruby
require_relative "boot"

require "rails/all"

Bundler.require(*Rails.groups)

module Server
  class Application < Rails::Application
    config.load_defaults 8.1

    config.autoload_lib(ignore: %w[assets tasks])

    config.eager_load_paths << Rails.root.join("app/services")

    config.api_only = true

    config.middleware.use ActionDispatch::Cookies
    config.middleware.use ActionDispatch::Session::CookieStore, key: '_marketplace_session'
  end
end
```

Рисунок 3.2 – Вміст файлу config/application.rb серверної частини

Спочатку ми налаштовуємо Rails додаток як API-only, що означає відсутність view-шаблонів та деяких middleware, необхідних для вебдодатків.

Далі ми підключаємо cookie-based сесії для автентифікації користувачів. Це дозволяє зберігати ідентифікатор користувача в зашифрованих cookies після успішної авторизації.

Також ми налаштовуємо Active Storage для роботи з файлами та зображеннями товарів.

3.4.1 Налаштування бази даних торговельного майданчику

Попередньо нам необхідно створити та налаштувати нашу базу даних в PostgreSQL, для цього необхідно виконати наступні кроки:

– встановити PostgreSQL на локальну машину або використати хмарний сервіс;
– створити базу даних для розробки;
– налаштувати підключення в файлі config/database.yml (рис. 3.2);
– виконати міграції для створення таблиць за допомогою команди «rails db:migrate».

Кроки використання ActiveRecord для роботи з базою даних:

– activerecord автоматично створює зв’язки між Ruby об’єктами та таблицями бази даних;
– моделі в Rails автоматично відповідають таблицям в базі даних (наприклад, модель User відповідає таблиці users);
– міграції дозволяють версіонувати зміни в структурі бази даних та застосовувати їх послідовно;
– команда rails db:migrate застосовує всі невиконані міграції до бази даних, створюючи необхідні таблиці та індекси.

Файл config/database.yml (рис. 3.3) використовується для налаштування підключення до бази даних PostgreSQL. Він містить конфігурацію для різних середовищ (development, test, production). В нашому випадку ми використовуємо такі параметри:

– adapter: postgresql – вказує на використання PostgreSQL;
– database: server_development – назва бази даних для розробки;
– username та password – дані для авторизації в базі даних;
– host та port – адреса та порт сервера бази даних.

Файл .env використовується для зберігання змінних середовища, які використовуються в нашому додатку. Це можуть бути конфіденційні дані, такі як ключі API, паролі, секрети, URL-адреси баз даних тощо.

В нашому випадку ми використовуємо такі змінні:

– POSTGRES_PASSWORD – це пароль для підключення до бази даних PostgreSQL;
– NP_API_KEY – це ключ API для інтеграції з Новою Поштою;
– LIQPAY_PUBLIC_KEY та LIQPAY_PRIVATE_KEY – це ключі для інтеграції з платіжною системою LiqPay;
– FRONTEND_URL та BACKEND_URL – це URL-адреси фронтенду та бекенду для налаштування CORS та callback URL.

```yaml
default: &default
  adapter: postgresql
  encoding: unicode
  max_connections: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>


development:
  <<: *default
  database: server_development
  username: postgres
  password: <%= ENV['POSTGRES_PASSWORD'] || '1234' %>
  host: localhost
  port: 5432

test:
  <<: *default
  database: server_test
  username: postgres
  password: <%= ENV['POSTGRES_PASSWORD'] || '1234' %>
  host: localhost
  port: 5432

production:
  primary: &primary_production
    <<: *default
    database: server_production
    username: server
    password: <%= ENV["SERVER_DATABASE_PASSWORD"] %>
  cache:
    <<: *primary_production
    database: server_production_cache
    migrations_paths: db/cache_migrate
  queue:
    <<: *primary_production
    database: server_production_queue
    migrations_paths: db/queue_migrate
  cable:
    <<: *primary_production
    database: server_production_cable
    migrations_paths: db/cable_migrate
```

Рисунок 3.3 – Вміст файлу config/database.yml

3.4.2 Налаштування серверу торговельного майданчику

Далі в файлі config/routes.rb відбувається налаштування маршрутів для нашого API.

Створення маршрутів: Rails.application.routes.draw do визначає блок, в якому описуються всі маршрути додатку.

RESTful маршрути: resources :products, only: [:index, :show, :create, :update, :destroy] створює стандартні RESTful маршрути для ресурсу products, обмежуючи їх тільки необхідними діями.

Кастомні маршрути: get ‘/nova_poshta/settlements’, to: ‘nova_poshta#get_settlements’ створює кастомний маршрут для пошуку населених пунктів через API Нової Пошти.

Namespace маршрути: namespace :seller do ... end створює групу маршрутів з префіксом /seller для функціоналу продавця.

Налаштування middleware: в файлі config/application.rb ми налаштовуємо CORS для дозволу cross-origin запитів від фронтенду, cookie-based сесії для автентифікації та Active Storage для роботи з файлами.

Запуск сервера: rails server або rails s використовується для запуску сервера на порту 3000 за замовчуванням. Сервер автоматично перезавантажується при зміні коду в режимі розробки.

3.4.3 Розробка моделей для наших сутностей

Моделі в Rails використовуються для визначення структури даних та бізнес-логіки. Вони автоматично відповідають таблицям в базі даних через ActiveRecord ORM. Моделі допомагають забезпечити узгодженість даних, встановлюючи визначені поля, валідації та зв’язки між сутностями.

Наприклад в нашому випадку модель User визначає структуру користувача (рис. 3.4). Вона вказує, що кожен користувач повинен мати поля user_name, email, password_digest (для зберігання хешу пароля), role (enum:

buyer, seller, admin), а також автоматично генеровані поля created_at та updated_at.

```ruby
class User < ApplicationRecord
    has_secure_password

    enum :role, { buyer: 0, seller: 1, admin: 2 }

    has_one  :store, foreign_key: :owner_id, dependent: :destroy
    has_many :orders, dependent: :nullify
    has_many :cart_items, dependent: :destroy
    has_many :notifications, dependent: :destroy

    validates :user_name, :email, presence: true
    validates :email, uniqueness: true
    validates :password, length: { minimum: 6 }, if: -> { new_record? || !password.nil? }
end
```

Рисунок 3.4 – Вигляд моделі User в файлі «user.rb»

Файл «user.rb» (рис. 3.4) визначає модель User за допомогою ActiveRecord для PostgreSQL. Модель визначає структуру та поведінку користувача. Опис частин коду в файлі «user.rb».

has_secure_password – це метод Rails, який автоматично додає функціонал для хешування паролів через bcrypt. Він додає методи password=, password_confirmation=, authenticate, а також поле password_digest в базі даних.

enum :role, { buyer: 0, seller: 1, admin: 2 } – визначає enum для ролі користувача з трьома значеннями: покупець, продавець, адміністратор. Значення зберігаються як цілі числа в базі даних для оптимізації.

has_one :store – визначає зв’язок один-до-одного між користувачем та магазином. Користувач може мати один магазин (якщо він продавець).

has_many :orders, :cart_items, :notifications – визначає зв’язки один-до-багатьох між користувачем та його замовленнями, товарами в кошику та сповіщеннями.

validates :user_name, :email, presence: true – валідація наявності обов’язкових полів.

validates :email, uniqueness: true – валідація унікальності email.

validates :password, length: { minimum: 6 } – валідація мінімальної довжини пароля.

Аналогічно, модель CartItem (рис. 3.5) визначає структуру товару в кошику. Вона вказує, що кожен товар в кошику повинен мати зв’язки з користувачем (опційно, для неавторизованих), продуктом, магазином та замовленням (опційно, поки товар в кошику).

```ruby
class CartItem < ApplicationRecord
  belongs_to :user,   optional: true
  belongs_to :product
  belongs_to :store
  belongs_to :order,  optional: true

  scope :in_cart,   -> { where(order_id: nil) }
  scope :for_store, ->(sid) { where(store_id: sid) }

  before_validation :sync_store
  validate :not_own_product
  validate :store_matches_product

  def sync_store
    self.store_id ||= product&.store_id
  end

  def not_own_product
    return unless user && product && product.store.owner_id == user.id
    errors.add(:base, "Can't add your own product")
  end

  def store_matches_product
    return unless product && store_id && store_id != product.store_id
    errors.add(:store_id, 'must match product store')
  end
end
```

Рисунок 3.5 – Вигляд моделі CartItem в файлі «cart_item.rb»

Файл «cart_item.rb» (рис. 3.5) визначає модель CartItem. Опис частин коду:

– belongs_to :user, optional: true – зв’язок з користувачем, опційний для підтримки неавторизованих користувачів;
– belongs_to :product, :store, :order – зв’язки з продуктом, магазином та замовленням;
– scope :in_cart, -> { where(order_id: nil) } – scope для отримання товарів, які ще в кошику (не прив’язані до замовлення);
– before_validation :sync_store – callback, який автоматично встановлює store_id на основі product.store_id перед валідацією;

– validate :not_own_product – валідація, яка забороняє додавати власні товари до кошика (для продавців);
– validate :store_matches_product – валідація, яка перевіряє, що store_id відповідає store_id продукту.

3.4.5 Функціонал кошику та перенесення з localStorage

Система кошику в нашому додатку підтримує два режими роботи: для авторизованих користувачів товари зберігаються в базі даних, а для неавторизованих – в localStorage браузера. При авторизації неавторизованого користувача товари автоматично переносяться з localStorage в базу даних.

Розглянемо детальніше контролер кошику «cart_items_controller.rb» та його методи (рис. 3.6).

```ruby
def create
  return head :unauthorized unless current_user

  attrs = {
    product_id: params[:product_id],
    store_id: params[:store_id],
    quantity: params[:quantity].to_i.positive? ? params[:quantity].to_i : 1
  }

  item = CartItem.in_cart.find_or_initialize_by(
    user_id: current_user.id,
    product_id: attrs[:product_id]
  )

  if item.new_record?
    item.store_id = attrs[:store_id] if attrs[:store_id]
    item.quantity = attrs[:quantity]
  else
    item.quantity = item.quantity.to_i + attrs[:quantity]
  end

  item.save!
  render json: { ok: true, id: item.id }
end
```

Рисунок 3.6 – Метод create в контролері «cart_items_controller.rb»

CartItemsController використовує модель CartItem для взаємодії з базою даних. Він обробляє запити на додавання, оновлення, видалення та отримання товарів з кошика.

Метод create (рис. 3.6) визначає обробку POST-запиту на маршрут /cart_items. Цей контролер використовує метод authenticate_user! для перевірки авторизації (тільки авторизовані користувачі можуть додавати товари в кошик в базі даних). Він знаходить або створює новий CartItem для поточного користувача та продукту. Якщо товар вже є в кошику, він збільшує кількість на вказану величину. Якщо товар новий, він створює новий запис з вказаною кількістю. Після збереження він відправляє відповідь з підтвердженням.

Метод index визначає обробку GET-запиту на маршрут /cart_items. Він завантажує всі товари з кошика поточного користувача, включаючи інформацію про продукти та магазини, та відправляє їх як JSON відповідь.

Метод transfer_to_user (рис. 3.7) визначає обробку POST-запиту на маршрут /cart_items/transfer_to_user. Цей метод використовується для перенесення товарів з localStorage в базу даних після авторизації користувача.

```ruby
def transfer_to_user
  return head :unauthorized unless current_user

  return render json: { ok: true, transferred: 0 } unless params[:cart_items].is_a?(Array)

  transferred = 0
  params[:cart_items].each do |item_data|
    product_id = item_data[:product_id] || item_data[:id]
    store_id = item_data[:store_id]
    quantity = item_data[:quantity].to_i

    next unless product_id && store_id && quantity > 0

    product = Product.find_by(id: product_id)
    next if product && product.store.owner_id == current_user.id

    item = CartItem.in_cart.find_or_initialize_by(
      user_id: current_user.id,
      product_id: product_id
    )

    if item.new_record?
      item.store_id = store_id
      item.quantity = quantity
    else
      item.quantity += quantity
    end

    item.save!
    transferred += 1
  rescue => e
    Rails.logger.error "Error transferring cart item: #{e.message}"
    next
  end

  render json: { ok: true, transferred: transferred }
end
```

Рисунок 3.7 – Метод transfer_to_user в контролері «cart_items_controller.rb»

Метод transfer_to_user (рис. 3.7) отримує масив товарів з localStorage (params[:cart_items]). Для кожного товару він перевіряє наявність обов’язкових полів (product_id, store_id, quantity). Він також перевіряє, чи не намагається користувач додати власні товари (якщо користувач є продавцем). Потім він знаходить або створює CartItem для користувача та продукту. Якщо товар вже є в кошику, він додає кількість до існуючої. Якщо товар новий, він створює новий запис. Після обробки всіх товарів він відправляє відповідь з кількістю перенесених товарів.

Цей процес перенесення забезпечує безшовний перехід від неавторизованого до авторизованого стану, зберігаючи товари користувача в кошику.

3.4.6 Система замовлень

Система замовлень дозволяє користувачам оформлювати замовлення з кошика, зберігати інформацію про доставку та відстежувати статуси замовлень. Система підтримує як авторизованих користувачів, так і неавторизованих (через access_token).

Розглянемо детальніше контролер замовлень «orders_controller.rb» та його методи (рис. 3.8).

OrdersController використовує модель Order для взаємодії з базою даних. Він обробляє запити на створення, оновлення, перегляд та видалення замовлень.

Метод create (рис. 3.8) визначає обробку POST-запиту на маршрут /orders. Цей контролер використовує метод authenticate_user! для перевірки авторизації. Він отримує store_id та cart_item_ids з параметрів запиту. Він знаходить всі товари з кошика користувача для вказаного магазину.

Якщо кошик порожній, він відправляє повідомлення про помилку з кодом статусу 422. Якщо товари є, він створює нове замовлення з тимчасовим

номером ‘00000000’, зберігає його в базі даних, оновлює номер замовлення на основі ID (формат: 8 цифр з ведучими нулями), прив’язує товари до замовлення (встановлює order_id для CartItem), зберігає snapshot ціни та назви продукту, перераховує загальну суму замовлення та створює сповіщення для користувача та продавця. Після цього він відправляє відповідь з ID, номером та store_id замовлення.

```ruby
class OrdersController < ApplicationController
  def create
    return head :unauthorized unless current_user

    store_id = params[:store_id]
    cart_item_ids = params[:cart_item_ids] || []

    items = CartItem.in_cart.where(user_id: current_user.id, store_id: store_id)

    items = items.where(id: cart_item_ids) if cart_item_ids.present?

    return render json: { error: 'Empty cart for this store' }, status: 422 if items.blank?

    order = Order.new(
      user: current_user,
      store_id: store_id,
      state: :pending,
      number: '00000000'
    )
    order.save!

    order.update_column(:number, format('%08d', order.id))

    items.find_each do |ci|
      ci.update!(
        order_id: order.id,
        unit_price_cents: ci.product.price_cents,
        product_title_snapshot: ci.product.title
      )
    end
```

Рисунок 3.8 – Метод create в контролері «orders_controller.rb»

Метод create_from_local (рис. 3.9) визначає обробку POST-запиту на маршрут /orders/create_from_local. Цей метод використовується для створення замовлення з localStorage для неавторизованих користувачів.

Метод create_from_local отримує store_id, cart_items (масив товарів з localStorage) та дані доставки (customer_name, customer_email, customer_phone, shipping_address) з параметрів запиту. Він перевіряє наявність товарів у кошику. Якщо кошик порожній, він відправляє повідомлення про помилку. Він нормалізує адресу доставки (перетворює об’єкт в рядок).

```ruby
def create_from_local
  store_id = params[:store_id]
  cart_items_data = params[:cart_items] || []
  return render json: { error: 'Empty cart items' }, status: 422 if cart_items_data.blank?

  shipping_address = normalize_shipping_address(params[:shipping_address])
  store = Store.find_by(id: store_id)
  return render json: { error: 'Store not found' }, status: 422 unless store

  begin
    order = Order.new(
      store_id: store_id.to_i,
      state: :pending,
      customer_name: params[:customer_name].to_s.strip,
      customer_email: params[:customer_email].to_s.strip,
      customer_phone: params[:customer_phone].to_s.strip,
      shipping_address: shipping_address,
      access_token: SecureRandom.hex(32),
      number: '00000000'
    )
    order.save!
    order.update_column(:number, format('%08d', order.id))
  rescue ActiveRecord::RecordInvalid => e
    return render json: { error: e.record.errors.full_messages.join(', ') }, status: 422
  rescue => e
    return render json: { error: "Failed to create order: #{e.message}" }, status: 500
  end

  created_items = 0
  cart_items_data.each do |item_data|
    product_id = item_data[:product_id] || item_data[:id]
    quantity = item_data[:quantity].to_i
    quantity = 1 if quantity <= 0
    next unless product_id && quantity > 0

    product = Product.find_by(id: product_id)
    next unless product

    cart_item = CartItem.new(
      order_id: order.id,
      product_id: product_id,
      store_id: product.store_id,
      quantity: quantity,
      unit_price_cents: product.price_cents,
      product_title_snapshot: product.title
    )
    cart_item.save(validate: false)
    created_items += 1
  end

  if created_items == 0
    order.destroy!
    return render json: { error: 'No valid items to create' }, status: 422
  end

  order.recalc_totals!
  create_seller_notification(order)

  render json: {
    id: order.id,
    number: order.number,
    store_id: order.store_id,
    access_token: order.access_token
  }
rescue => e
  render json: { error: e.message }, status: 500
end
```

Рисунок 3.9 – Метод create_from_local в контролері «orders_controller.rb»

Метод create_from_local перевіряє існування магазину. Потім він створює нове замовлення без user_id (для неавторизованих), генерує унікальний access_token для доступу до замовлення (SecureRandom.hex(32)), встановлює тимчасовий номер ‘00000000’, зберігає замовлення в базі даних та оновлює номер на основі ID. Далі він створює CartItem записи для кожного товару з localStorage, зберігаючи snapshot ціни та назви продукту. Після створення всіх товарів він перераховує загальну суму замовлення та створює сповіщення для продавця. Він відправляє відповідь з ID, номером, store_id та access_token замовлення (для збереження в localStorage).

Метод update_state визначає обробку PATCH-запиту на маршрут /orders/:id/update_state. Цей метод дозволяє змінювати статус замовлення. Користувач (як покупець) може скасовувати свої замовлення тільки якщо статус «Нове» (pending -> canceled). Продавець може змінювати статуси замовлень свого магазину (pending -> accepted, accepted -> fulfilled, тощо). При зміні статусу створюється сповіщення для користувача.

3.4.7 Інтеграція API Нової Пошти

Інтеграція з API Нової Пошти дозволяє користувачам шукати населені пункти та відділення/поштомати для доставки замовлень. API Нової Пошти надає RESTful інтерфейс для пошуку адрес та відділень.

Розглянемо детальніше контролер Нової Пошти «nova_poshta_controller.rb» та його методи (рис. 3.10). NovaPoshtaController використовує Net::HTTP для відправки HTTP-запитів до API Нової Пошти. Він обробляє запити на пошук населених пунктів та відділень.

Метод get_settlements (рис. 3.10) визначає обробку GET-запиту на маршрут /nova_poshta/settlements. Цей контролер отримує query (пошуковий запит) з параметрів запиту. Якщо query порожній, він відправляє порожній масив. Він отримує API ключ з змінних середовища (ENV[‘NP_API_KEY’]). Якщо ключ відсутній, він відправляє повідомлення про помилку з кодом

статусу 500. Потім він формує POST-запит до API Нової Пошти з необхідними параметрами (apiKey, modelName: ‘AddressGeneral’, calledMethod: ‘getSettlements’, methodProperties: { FindByString: query, Page: 1, Limit: 150 }). Він відправляє запит через HTTPS, обробляє відповідь, парсить JSON та відправляє дані клієнту. Якщо виникає помилка, він логує її та відправляє повідомлення про помилку.

```ruby
def get_settlements
  query = params[:query] || ''
  return render json: { data: [] }, status: :ok if query.blank?

  api_key = ENV['NP_API_KEY']
  unless api_key
    return render json: { error: 'Nova Poshta API key not configured' }, status: :internal_server_error
  end

  begin
    uri = URI('https://api.novaposhta.ua/v2.0/json/')
    request_body = {
      apiKey: api_key,
      modelName: 'AddressGeneral',
      calledMethod: 'getSettlements',
      methodProperties: {
        FindByString: query,
        Page: 1,
        Limit: 150,
        Warehouse: '1'
      }
    }

    http = Net::HTTP.new(uri.host, uri.port)
    http.use_ssl = true
    http.verify_mode = OpenSSL::SSL::VERIFY_NONE

    request = Net::HTTP::Post.new(uri.path)
    request['Content-Type'] = 'application/json'
    request.body = request_body.to_json

    response = http.request(request)

    if response.code == '200'
      data = JSON.parse(response.body)
      render json: { data: data['data'] || [] }, status: :ok
    else
      render json: { error: 'Nova Poshta API error' }, status: :bad_gateway
    end
  rescue => e
    render json: { error: 'Failed to fetch settlements', message: e.message }, status: :internal_server_error
  end
end
```

Рисунок 3.10 – Метод get_settlements в контролері «nova_poshta_controller.rb»

Метод get_warehouses (рис. 3.11) визначає обробку GET-запиту на маршрут /nova_poshta/warehouses. Цей метод отримує city (назва міста), query (пошуковий запит для фільтрації) та category (тип доставки: branch або postomat) з параметрів запиту.

```ruby
def get_warehouses
  city = params[:city].to_s.strip
  query = params[:query].to_s.strip
  category = params[:category].to_s.strip.downcase
  category = 'branch' if category.blank?

  return render json: { data: [] }, status: :ok if city.blank?

  api_key = ENV['NP_API_KEY']
  unless api_key
    return render json: { error: 'Nova Poshta API key not configured' }, status: :internal_server_error
  end

  begin
    uri = URI('https://api.novaposhta.ua/v2.0/json/')
    limit = 100
    max_pages = 20
    page = 1
    warehouses = []

    loop do
      request_body = {
        apiKey: api_key,
        modelName: 'Address',
        calledMethod: 'getWarehouses',
        methodProperties: {
          CityName: city,
          Page: page,
          Limit: limit
        }
      }

      http = Net::HTTP.new(uri.host, uri.port)
      http.use_ssl = true
      http.verify_mode = OpenSSL::SSL::VERIFY_NONE

      request = Net::HTTP::Post.new(uri.path)
      request['Content-Type'] = 'application/json'
      request.body = request_body.to_json

      response = http.request(request)
      return render json: { error: 'Nova Poshta API error' }, status: :bad_gateway unless response.code == '200'

      data = JSON.parse(response.body)
      page_data = data['data'] || []
      warehouses.concat(page_data)

      break if page_data.length < limit || page >= max_pages
      page += 1
    end

    warehouses = if category == 'postomat'
                   warehouses.select { |w| w['CategoryOfWarehouse'] == 'Postomat' }
                 else
                   warehouses.reject { |w| w['CategoryOfWarehouse'] == 'Postomat' }
                 end

    if query.present?
      query_normalized = query.strip
      filtered_by_number = warehouses.select do |w|
        number = w['Number'].to_s.strip
        number_int = number.to_i
        query_int = query_normalized.to_i
        number == query_normalized || number_int == query_int ||
        number.include?(query_normalized) || number.start_with?(query_normalized)
      end

      if filtered_by_number.any?
        render json: { data: filtered_by_number }, status: :ok
      else
        filtered_by_description = warehouses.select do |w|
          (w['Description'] || '').downcase.include?(query_normalized.downcase)
        end
        render json: { data: filtered_by_description }, status: :ok
      end
    else
      render json: { data: warehouses }, status: :ok
    end
  rescue => e
    render json: { error: 'Failed to fetch warehouses', message: e.message }, status: :internal_server_error
  end
end
```

Рисунок 3.11 – Метод get_warehouses в контролері «nova_poshta_controller.rb»

Метод get_warehouses (рис. 3.11) перевіряє наявність назви міста. Якщо місто порожнє, він відправляє порожній масив. Він отримує API ключ з змінних середовища. Потім він виконує пагінацію для отримання всіх відділень (до 20 сторінок по 100 записів), оскільки API Нової Пошти повертає результати по сторінках. Для кожної сторінки він формує POST-запит до API з параметрами (apiKey, modelName: ‘Address’, calledMethod: ‘getWarehouses’, methodProperties: { CityName: city, Page: page, Limit: 100 }). Він збирає всі результати в один масив. Після отримання всіх відділень він фільтрує їх за типом (branch або postomat) на основі параметра category. Якщо вказано query, він додатково фільтрує відділення за номером або описом (спочатку шукає за номером, потім за описом). Після фільтрації він відправляє відфільтровані дані клієнту.

Ця інтеграція забезпечує зручний пошук адрес доставки для користувачів без необхідності вручну вводити повну адресу.

3.4.8 Інтеграція платіжної системи LiqPay

Інтеграція з платіжною системою LiqPay дозволяє користувачам оплачувати замовлення через безпечний платіжний шлюз. LiqPay надає API для створення платіжних форм та обробки callback-запитів після оплати.

Розглянемо детальніше сервіс LiqPay «liqpay/client.rb» та контролер «liqpay_controller.rb» (рис. 3.12).

Сервіс Liqpay::Client містить логіку для генерації payload та підпису для LiqPay API. Він використовує Base64 для кодування даних та SHA1 для створення підпису.

Метод checkout_payload (рис. 3.12) створює payload для платіжної форми LiqPay. Він приймає параметри замовлення (order, amount, description, result_url, server_url, currency, sandbox). Він формує об’єкт params з необхідними полями (version: 3, public_key, action: ‘pay’, amount, currency, description, order_id, result_url, server_url, sandbox). Він кодує params в Base64

(encode), створює підпис для закодованих даних (signature_for) та повертає об’єкт з url, data (закодовані дані) та signature (підпис).

```ruby
require "base64"
require "digest"
require "json"

module Liqpay
  class Client
    CHECKOUT_URL = "https://www.liqpay.ua/api/3/checkout".freeze

    class << self
      def checkout_payload(order:, amount:, description:, result_url:, server_url:, currency: "UAH", sandbox: true)
        params = {
          version: 3,
          public_key: public_key,
          action: "pay",
          amount: amount,
          currency: currency,
          description: description,
          order_id: order.liqpay_order_id,
          result_url: result_url,
          server_url: server_url,
          sandbox: sandbox ? 1 : 0
        }
        encoded_data = encode(params)
        {
          url: CHECKOUT_URL,
          data: encoded_data,
          signature: signature_for(encoded_data)
        }
      end

      def decode(encoded_data)
        json = Base64.strict_decode64(encoded_data)
        JSON.parse(json)
      rescue JSON::ParserError
        {}
      end

      def signature_for(encoded_data)
        raw_signature = Digest::SHA1.digest("#{private_key}#{encoded_data}#{private_key}")
        Base64.strict_encode64(raw_signature)
      end

      private

      def encode(params)
        Base64.strict_encode64(params.to_json)
      end

      def public_key
        ENV.fetch("LIQPAY_PUBLIC_KEY")
      end

      def private_key
        ENV.fetch("LIQPAY_PRIVATE_KEY")
      end
    end
  end
end
```

Рисунок 3.12 – Сервіс LiqPay Client в файлі «liqpay/client.rb»

Метод signature_for створює підпис для безпеки транзакції. Він використовує формулу: SHA1(private_key + data + private_key), де data – це

закодовані в Base64 параметри платежу. Підпис кодується в Base64 та повертається.

Метод decode декодує дані з Base64 та парсить JSON для отримання параметрів платежу з callback-запиту.

Розглянемо контролер LiqPay «liqpay_controller.rb» та його методи (рис. 3.13).

```ruby
class LiqpayController < ApplicationController
  before_action :set_order, only: :checkout

  def checkout
    return unless authorize_order_access!(@order)

    ensure_liqpay_order_id!(@order)

    @order.recalc_totals! if @order.total_cents.nil?

    amount = (@order.total_cents || 0).to_f / 100
    if amount <= 0
      return render json: { error: "Сума замовлення має бути більшою за нуль" }, status: :unprocessable_entity
    end

    description = "Оплата замовлення #{@order.number}"
    payload = Liqpay::Client.checkout_payload(
      order: @order,
      amount: format("%.2f", amount),
      description: description,
      result_url: result_url(@order),
      server_url: server_url,
      currency: "UAH",
      sandbox: sandbox_mode?
    )

    render json: payload
  end
```

Рисунок 3.13 – Метод checkout в контролері «liqpay_controller.rb»

Метод checkout (рис. 3.13) визначає обробку POST-запиту на маршрут /orders/:id/liqpay. Цей контролер використовує before_action :set_order для завантаження замовлення. Він перевіряє доступ до замовлення через метод authorize_order_access! (перевіряє, чи користувач є власником замовлення або має access_token). Він забезпечує наявність liqpay_order_id для замовлення (генерує його, якщо відсутній, у форматі: номер замовлення + випадковий hex). Він перераховує загальну суму замовлення, якщо вона не встановлена. Він перевіряє, чи сума більша за нуль. Потім він викликає Liqpay::Client.checkout_payload для генерації payload з параметрами (order,

amount, description, result_url, server_url, currency: ‘UAH’, sandbox: sandbox_mode?). Він відправляє payload клієнту для відображення платіжної форми.

Метод callback (рис. 3.14) визначає обробку POST-запиту на маршрут /liqpay/callback. Цей метод обробляє callback-запит від LiqPay після оплати.

```ruby
def callback
  data = params[:data]
  signature = params[:signature]

  unless data.present? && signature.present?
    return head :bad_request
  end

  expected_signature = Liqpay::Client.signature_for(data)
  unless signature.is_a?(String) &&
         expected_signature.is_a?(String) &&
         signature.bytesize == expected_signature.bytesize &&
         ActiveSupport::SecurityUtils.secure_compare(signature, expected_signature)
    return head :forbidden
  end

  payload = Liqpay::Client.decode(data)
  if payload.blank? || payload["order_id"].blank?
    return head :unprocessable_entity
  end

  order = Order.find_by(liqpay_order_id: payload["order_id"])
  return head :not_found unless order

  status = payload["status"]
  successful = successful_status?(status)

  update_attrs = {
    liqpay_payment_status: status,
    payment_status: successful ? "Сплачено" : order.payment_status
  }
  update_attrs[:liqpay_paid_at] = Time.current if successful
  order.update_columns(update_attrs)

  if successful
    begin
      order.ensure_store_accrual_recorded!
    rescue => e
      # Store accrual error handling
    end
  end

  head :ok
rescue JSON::ParserError => e
  head :unprocessable_entity
rescue => e
  head :internal_server_error
end
```

Рисунок 3.14 – Метод callback в контролері «liqpay_controller.rb»

Метод callback (рис. 3.14) отримує data (закодовані дані) та signature (підпис) з параметрів запиту. Він перевіряє наявність обох параметрів. Якщо параметри відсутні, він відправляє код статусу 400 (Bad Request). Він декодує

data через Liqpay::Client.decode для отримання payload. Він перевіряє підпис через ActiveSupport::SecurityUtils.secure_compare для безпечного порівняння (захист від timing attacks). Якщо підпис не співпадає, він відправляє код статусу 403 (Forbidden). Він знаходить замовлення за liqpay_order_id з payload. Якщо замовлення не знайдено, він відправляє код статусу 404 (Not Found). Він визначає, чи оплата успішна, на основі статусу з payload (success, sandbox, test вважаються успішними). Він оновлює поля замовлення (liqpay_payment_status, payment_status, liqpay_paid_at якщо успішно). Якщо оплата успішна, він викликає order.ensure_store_accrual_recorded! для автоматичного нарахування коштів на баланс магазину продавця. Після обробки він відправляє код статусу 200 (OK).

Метод ensure_store_accrual_recorded! в моделі Order (рис. 3.15) забезпечує автоматичне нарахування коштів на баланс магазину після успішної оплати замовлення.

```ruby
def ensure_store_accrual_recorded!
  return unless store
  return unless payment_status.to_s == 'Сплачено'
  return unless total_cents.to_i.positive?

  StoreTransaction.transaction do
    store.with_lock do
      already_recorded = StoreTransaction.exists?(
        store_id: store_id,
        order_id: id,
        transaction_type: StoreTransaction.transaction_types[:accrual]
      )
      unless already_recorded
      StoreTransaction.create!(
          store: store,
          order: self,
          transaction_type: :accrual,
          amount_cents: total_cents,
          description: "Нарахування за замовлення #{number}",
        status: :accrued,
        metadata: { order_number: number }
        )
      end
    end
  end
end
```

Рисунок 3.15 – Метод ensure_store_accrual_recorded! в моделі «order.rb»

Метод ensure_store_accrual_recorded! (рис. 3.15) перевіряє наявність магазину, статус оплати (‘Сплачено’) та позитивну суму замовлення. Він

використовує транзакцію бази даних для атомарності операції. Він блокує запис магазину для уникнення race conditions. Він перевіряє, чи не було вже створено транзакцію нарахування для цього замовлення. Якщо транзакції немає, він створює нову StoreTransaction з типом :accrual, сумою total_cents, описом та статусом :accrued. Після створення транзакції автоматично спрацьовує callback apply_to_store_balance_if_needed в моделі StoreTransaction, який оновлює баланс магазину.

Ця інтеграція забезпечує безпечну обробку платежів та автоматичне нарахування коштів продавцям після успішної оплати замовлень.

3.5 Реалізація клієнтської частини торговельного майданчику

Для початку потрібно створити середовище розробки клієнтської частини. Для цього створюємо в папці проєкту нову папку «client».

Після цього ми створюємо проєкт клієнтської частини за допомогою команди «npm create vite@latest . -- --template react».

Команда npm create vite@latest . -- --template react використовується для створення нового проєкту React з використанням Vite в поточній директорії.

Vite – це сучасний інструмент для розробки фронтенд-додатків, який забезпечує швидку збірку та гарячу заміну модулів (HMR) під час розробки.

Опція --template react вказує на використання шаблону React для створення проєкту.

Точка (.) в кінці команди вказує, що новий проєкт React має бути створений в поточній директорії.

Тепер нам потрібно зробити інсталяцію всіх необхідних бібліотек та інструментів для розробки клієнтської частини за допомогою одної команди: «npm i axios react-router-dom».

Дана команда встановлює два пакети npm в наш проєкт:

а) axios: цей пакет є HTTP-клієнтом для браузера та Node.js, який дозволяє легко виконувати HTTP-запити до сервера – він підтримує

проміси, автоматичну трансформацію JSON, перехоплення запитів та відповідей, та інші корисні функції;

б) react-router-dom: це бібліотека для маршрутизації в React-додатках, яка дозволяє створювати односторінкові додатки (SPA) з навігацією між різними сторінками без перезавантаження сторінки.

Файл App.jsx є головним компонентом нашого React-додатку. Він використовує React Router для маршрутизації та Context API для управління глобальним станом (авторизація, кошик).

Спочатку ми імпортуємо необхідні модулі та компоненти, включаючи BrowserRouter, Routes, Route, Navigate з react-router-dom, AuthProvider та CartProvider з контекстів, а також всі компоненти сторінок (рис. 3.16).

```
import { BrowserRouter as Router, Routes, Route, useLocation, Navigate } from 'react-router-dom';
import { AuthProvider } from './contexts/AuthContext';
import { CartProvider } from './contexts/CartContext';
import Navbar from './components/Navbar/Navbar';
import Home from './components/Home/Home';
import Login from './components/Login/Login';
import Register from './components/Register/Register';
import StoreLayout from './components/Store/StoreLayout/StoreLayout';
import StoreOrdersList from './components/Store/StoreOrdersList/StoreOrdersList';
import StoreView from './components/Store/StoreView/StoreView';
import ProductsList from './components/Store/ProductsList/ProductsList';
import CreateProduct from './components/Store/CreateProduct/CreateProduct';
import EditProduct from './components/Store/EditProduct/EditProduct';
import ProductView from './components/Product/ProductView/ProductView';
import Cart from './components/Cart/Cart';
import Checkout from './components/Checkout/Checkout';
import OrderView from './components/OrderView/OrderView';
import UserCabinet from './components/UserCabinet/UserCabinet';
import OrdersList from './components/UserCabinet/OrdersList';
import Notifications from './components/UserCabinet/Notifications';
import StoreWallet from './components/Store/StoreWallet/StoreWallet';
import StoreAccount from './components/Store/StoreAccount/StoreAccount';
import AdminLayout from './components/Admin/AdminLayout/AdminLayout';
import UsersList from './components/Admin/UsersList/UsersList';
import CategoriesList from './components/Admin/CategoriesList/CategoriesList';
import styles from './App.module.css';
```

Рисунок 3.16 – Імпорти та підключення провайдерів в App.jsx

AuthProvider та CartProvider використовуються для надання глобального стану авторизації та кошика для всіх компонентів в додатку. BrowserRouter компонент з react-router-dom використовується для реалізації маршрутизації в додатку.

В компоненті AppContent визначається логіка відображення Navbar залежно від поточного маршруту та визначення всіх маршрутів додатку (рис. 3.17).

```
function AppContent() {
  const location = useLocation();
  const isStoreCabinetPage = location.pathname.startsWith('/cabinet/store');
  const isAdminCabinetPage = location.pathname.startsWith('/cabinet/admin');
  const isCheckoutPage = location.pathname.includes('/checkout');
  const showNavbar = !isStoreCabinetPage && !isAdminCabinetPage && !isCheckoutPage;

  return (
    <div className={styles.app}>
      {showNavbar && <Navbar />}
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/home" element={<Home />} />
        <Route path="/login" element={<Login />} />
        <Route path="/register" element={<Register />} />
        <Route path="/cart" element={<Cart />} />
        <Route path="/product" element={<Navigate to="/" replace />} />
        <Route path="/product/:id" element={<ProductView />} />
        <Route path="/store" element={<Navigate to="/" replace />} />
        <Route path="/store/:id" element={<StoreView />} />
        <Route path="/order/:id/checkout" element={<Checkout />} />
        <Route path="/order/:id" element={<OrderView />} />
        <Route path="/cabinet/user" element={<UserCabinet />}>
          <Route index element={<OrdersList />} />
          <Route path="orders" element={<OrdersList />} />
          <Route path="orders/:id" element={<OrderView />} />
          <Route path="notifications" element={<Notifications />} />
        </Route>
        <Route path="/cabinet/store" element={<StoreLayout />}>
          <Route index element={<Navigate to="/cabinet/store/orders" replace />} />
          <Route path="orders" element={<StoreOrdersList />} />
          <Route path="orders/:id" element={<OrderView />} />
          <Route path="product" element={<ProductsList />} />
          <Route path="product/create" element={<CreateProduct />} />
          <Route path="product/edit/:id" element={<EditProduct />} />
          <Route path="account" element={<StoreAccount />} />
          <Route path="wallet" element={<StoreWallet />} />
          <Route path="notifications" element={<Notifications />} />
        </Route>
        <Route path="/cabinet/admin" element={<AdminLayout />}>
          <Route index element={<Navigate to="/cabinet/admin/users" replace />} />
          <Route path="users" element={<UsersList />} />
          <Route path="categories" element={<CategoriesList />} />
        </Route>
      </Routes>
    </div>
  );
}
```

Рисунок 3.17 – Визначення маршрутів в AppContent

Navbar не відображається на сторінках кабінету та checkout для кращого користувацького досвіду. Всі маршрути визначені всередині компонента Routes, включаючи головну сторінку, сторінки авторизації, кошик, товари, магазини, замовлення та кабінети користувача, продавця та адміністратора.

3.5.1 Розробка компонента кошику

Компонент кошику Cart.jsx відображає товари, додані користувачем до кошика, дозволяє змінювати кількість товарів, видаляти товари та оформлювати замовлення. Компонент підтримує роботу як для авторизованих користувачів (товари зберігаються в базі даних), так і для неавторизованих (товари зберігаються в localStorage). Компонент Cart використовує useCart для отримання функцій та стану кошика, useAuth для отримання інформації про користувача та useNavigate для навігації (рис. 3.18).

```
import { useState } from 'react';
import { useCart } from '../../contexts/CartContext';
import { useAuth } from '../../contexts/AuthContext';
import { useNavigate } from 'react-router-dom';
import { getImageUrl } from '../../utils/imageUtils';
import styles from './Cart.module.css';

const Cart = () => {
  const { cartItems, loading, getGroupedCartItems, updateQuantity, removeFromCart, createOrder } = useCart();
  const navigate = useNavigate();
  const groupedItems = getGroupedCartItems();
  const [processingOrder, setProcessingOrder] = useState(null);

  const { user } = useAuth();
```

Рисунок 3.18 – Ініціалізація компонента Cart з використанням контекстів

Функція handleOrderClick викликається при натисканні кнопки «Оформити замовлення» для групи товарів одного магазину. Вона перевіряє, чи користувач авторизований, та отримує відповідні ідентифікатори товарів (cart_item.id для авторизованих, product.id для неавторизованих) (рис. 3.19).

Якщо користувач авторизований, функція завантажує товари з бази даних та отримує cart_item.id для кожного товару. Якщо користувач неавторизований, вона використовує product.id з localStorage. Потім вона викликає createOrder для створення замовлення в localStorage та перенаправляє користувача на сторінку оформлення замовлення.

В рендерингу компонента Cart відображається список товарів, згрупованих за магазинами. Для кожної групи відображається назва магазину, список товарів з можливістю зміни кількості та видалення, загальна сума та кнопка «Оформити замовлення» (рис. 3.20).

```
const handleOrderClick = async (storeId, groupItems) => {
  try {
    setProcessingOrder(storeId);
    let cartItemIds = [];

    if (user) {
      const { cartAPI } = await import('../../api/cart');
      const dbItems = await cartAPI.getCartItems();
      cartItemIds = groupItems
        .map(item => {
          const dbItem = dbItems.find(db => db.product.id === item.id);
          return dbItem?.id;
        })
        .filter(id => id !== undefined);
    } else {
      cartItemIds = groupItems.map(item => item.id);
    }

    const order = await createOrder(storeId, cartItemIds);
    navigate(`/order/${order.number}/checkout`);
  } catch (error) {
    console.error('Error creating order:', error);
    alert('Помилка при створенні замовлення');
  } finally {
    setProcessingOrder(null);
  }
};
```

Рисунок 3.19 – Функція handleOrderClick для створення замовлення

```
return (
  <div className={styles.container}>
    <h1 className={styles.title}>Кошик</h1>

    <div className={styles.groups}>
      {groupedItems.map((group) => (
        <div key={group.storeId} className={styles.storeGroup}>
          <div className={styles.storeHeader}>
            <h2 className={styles.storeName}>{group.storeName}</h2>
            <span className={styles.itemsCount}>{group.items.length} {group.items.length === 1 ? 'товар' : 'товарів'}</span>
          </div>

          <div className={styles.itemsList}>
            {group.items.map((item) => (
              <div key={item.id} className={styles.cartItem}>
                {item.images && item.images.length > 0 && (
                  <img src={getImageUrl(item.images[0])} alt={item.title} className={styles.itemImage} onClick={() => navigate(`/product/${item.id}`)} />
                )}
                <div className={styles.itemInfo}>
                  <h3 className={styles.itemTitle} onClick={() => navigate(`/product/${item.id}`)}>
                    {item.title}
                  </h3>
                  {item.description && (
                    <p className={styles.itemDescription}>
                      {item.description.length > 100
                        ? `${item.description.substring(0, 100)}...`
                        : item.description}
                    </p>
                  )}
                  <div className={styles.itemPrice}>
                    {formatPrice(item.price_cents)} грн
                  </div>
                </div>
                <div className={styles.itemControls}>
                  <div className={styles.quantityControls}>
                    <button onClick={() => updateQuantity(item.id, item.quantity - 1)} className={styles.quantityButton}>-</button>
                    <span className={styles.quantity}>{item.quantity}</span>
                    <button onClick={() => updateQuantity(item.id, item.quantity + 1)} className={styles.quantityButton}>+</button>
                  </div>
                  <div className={styles.itemTotal}>
                    {formatPrice(item.price_cents * item.quantity)} грн
                  </div>
                  <button className={styles.removeButton} onClick={() => removeFromCart(item.id)}>Видалити</button>
                </div>
              </div>
            ))}
          </div>

          <div className={styles.groupFooter}>
            <div className={styles.groupTotal}>
              <span className={styles.totalLabel}>Всього:</span>
              <span className={styles.totalAmount}> {formatPrice(calculateGroupTotal(group.items))} грн</span>
            </div>
            <button className={styles.orderButton} onClick={() => handleOrderClick(group.storeId, group.items)} disabled={processingOrder === group.storeId
              {processingOrder === group.storeId ? 'Обробка...' : 'Оформити замовлення'}
            </button>
          </div>
        </div>
      ))}
    </div>
  </div>
);
```

Рисунок 3.20 – Рендеринг списку товарів, згрупованих за магазинами

Компонент використовує CartContext для управління станом кошика. CartContext забезпечує синхронізацію між базою даних (для авторизованих користувачів) та localStorage (для неавторизованих користувачів), автоматичне перенесення товарів з localStorage в базу даних при авторизації та оптимізацію даних перед збереженням в localStorage.

3.5.2 Розробка компонента замовлення

Компонент замовлення OrderView.jsx відображає детальну інформацію про замовлення, включаючи статус замовлення, дані користувача, адресу доставки, список товарів та загальну суму. Компонент підтримує різні режими відображення залежно від ролі користувача (покупець, продавець, адміністратор) та дозволяє змінювати статус замовлення (для продавців) або скасовувати замовлення (для покупців).

Компонент OrderView використовує useParams для отримання ID замовлення з URL, useNavigate для навігації, useLocation для визначення поточного маршруту, useAuth для отримання інформації про користувача та useState для управління станом (рис. 3.21).

```
import { useState, useEffect } from 'react';
import { useParams, useNavigate, useLocation } from 'react-router-dom';
import { useAuth } from '../../contexts/AuthContext';
import { ordersAPI } from '../../api/orders';
import { getImageUrl } from '../../utils/imageUtils';
import PopUp from '../PopUp/PopUp';
import styles from './OrderView.module.css';

const OrderView = () => {
  const { id } = useParams();
  const navigate = useNavigate();
  const location = useLocation();
  const { user, loading: authLoading } = useAuth();
  const [order, setOrder] = useState(null);
  const [loading, setLoading] = useState(true);
  const [accessDenied, setAccessDenied] = useState(false);
  const [updatingState, setUpdatingState] = useState(false);
  const [showCancelPopup, setShowCancelPopup] = useState(false);

  const isUserCabinetRoute = location.pathname.startsWith('/cabinet/user/orders');
  const isStoreCabinetRoute = location.pathname.startsWith('/cabinet/store/orders');
```

Рисунок 3.21 – Ініціалізація компонента OrderView з хуками

Функція loadOrder викликається при монтуванні компонента або зміні ID замовлення. Вона перевіряє доступ до замовлення на основі ролі користувача та маршруту. Для неавторизованих користувачів вона отримує access_token з localStorage (рис. 3.22).

```
const loadOrder = async () => {
  try {
    setLoading(true);
    setAccessDenied(false);

    if (isUserCabinetRoute && !user) {
      navigate('/cabinet/user/orders');
      return;
    }

    if (isStoreCabinetRoute && (!user || (user.role !== 'seller' && user.role !== 'admin'))) {
      navigate('/cabinet/store/orders');
      return;
    }

    try {
      let accessToken = null;
      if (!isUserCabinetRoute && !isStoreCabinetRoute) {
        const orderTokens = JSON.parse(localStorage.getItem('order_tokens') || '{}');
        accessToken = orderTokens[id];
      }

      const orderData = await ordersAPI.getOrder(id, accessToken);

      if (isStoreCabinetRoute) {
        setOrder(orderData);
        return;
      }

      if (user && !accessToken && orderData.user_id && orderData.user_id !== user.id) {
        if (user.role !== 'seller' && user.role !== 'admin') {
          setAccessDenied(true);
          setLoading(false);
          return;
        }
      }

      if (!user && !accessToken) {
        setAccessDenied(true);
        setLoading(false);
        return;
      }

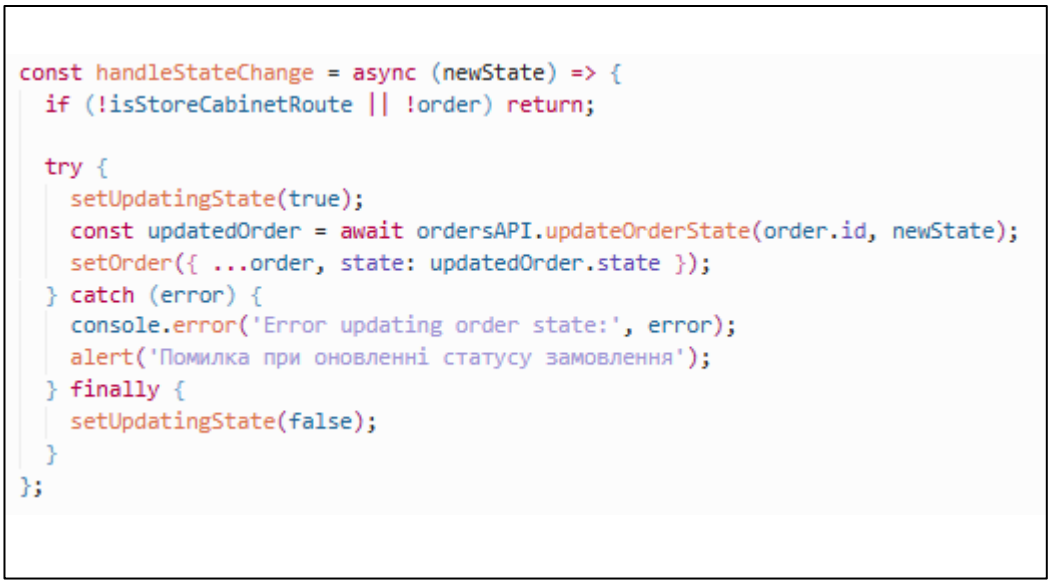
      setOrder(orderData);
    } catch (apiError) {
      if (apiError.response?.status === 403 || apiError.response?.status === 401) {
        setAccessDenied(true);
        setLoading(false);
        return;
      }

      if (isUserCabinetRoute) {
        navigate('/cabinet/user/orders');
      } else if (isStoreCabinetRoute) {
        navigate('/cabinet/store/orders');
      } else {
        navigate('/cart');
      }
    }
  } catch (error) {
    console.error('Error loading order:', error);
    setAccessDenied(true);
    setLoading(false);
  } finally {
    setLoading(false);
  }
};
```

Рисунок 3.22 – Функція loadOrder з перевіркою доступу

Потім вона викликає ordersAPI.getOrder для завантаження даних замовлення з сервера. Якщо доступ дозволено, вона встановлює дані замовлення в стан. Якщо доступ заборонено, вона встановлює accessDenied в true.

Функція handleStateChange викликається при зміні статусу замовлення (тільки для продавців). Вона викликає ordersAPI.updateOrderState для оновлення статусу замовлення на сервері та оновлює локальний стан після успішного оновлення (рис. 3.23).



```
const handleStateChange = async (newState) => {
  if (!isStoreCabinetRoute || !order) return;

  try {
    setUpdatingState(true);
    const updatedOrder = await ordersAPI.updateOrderState(order.id, newState);
    setOrder({ ...order, state: updatedOrder.state });
  } catch (error) {
    console.error('Error updating order state:', error);
    alert('Помилка при оновленні статусу замовлення');
  } finally {
    setUpdatingState(false);
  }
};
```

Рисунок 3.23 – Функція handleStateChange для оновлення статусу замовлення

Функція handleCancelConfirm викликається при підтвердженні скасування замовлення (тільки для покупців, якщо статус замовлення «Нове»). Вона викликає ordersAPI.updateOrderState для зміни статусу на «canceled» та оновлює локальний стан.

В рендерингу компонента OrderView відображається номер замовлення, статус замовлення, статус оплати, дані користувача, адреса доставки, список товарів та загальна сума замовлення. Для продавців відображається dropdown для зміни статусу замовлення. Для покупців відображається кнопка «Скасувати замовлення» (якщо статус «Нове»).

3.5.3 Використання API Нової Пошти на клієнті

Інтеграція з API Нової Пошти на клієнті дозволяє користувачам шукати населені пункти та відділення/поштомати для доставки замовлень безпосередньо в інтерфейсі оформлення замовлення. Інтеграція реалізована в компоненті Checkout.jsx з використанням автодоповнення (autocomplete) та debounce для оптимізації кількості запитів до API.

Визначаються стани для управління пошуком: settlements (список населених пунктів), showSettlementsDropdown (відображення dropdown), loadingSettlements (індикатор завантаження), branches (список відділень), postomats (список поштоматів), warehousesCache (кеш для результатів пошуку) (рис. 3.24).

```
const createEmptyDeliveryData = () => ({ ...baseDeliveryData });

const Checkout = () => {
  const { id } = useParams();
  const navigate = useNavigate();
  const { user } = useAuth();
  const { getGroupedCartItems } = useCart();

  const [order, setOrder] = useState(null);
  const [loading, setLoading] = useState(true);
  const [submitting, setSubmitting] = useState(false);
  const [formData, setFormData] = useState({
    customer_name: '',
    customer_email: '',
    customer_phone: '',
  });
  const [deliveryType, setDeliveryType] = useState('branch'); // 'branch', 'postomat', 'courier'
  const [deliveryData, setDeliveryData] = useState(() => createEmptyDeliveryData());
  const [errors, setErrors] = useState([]);
  const [showCancelPopup, setShowCancelPopup] = useState(false);
  const [settlements, setSettlements] = useState([]);
  const [showSettlementsDropdown, setShowSettlementsDropdown] = useState(false);
  const [loadingSettlements, setLoadingSettlements] = useState(false);
  const settlementsTimeoutRef = useRef(null);
  const cityInputRef = useRef(null);
  const dropdownRef = useRef(null);
  const [branches, setBranches] = useState([]);
  const [showBranchesDropdown, setShowBranchesDropdown] = useState(false);
  const [loadingBranches, setLoadingBranches] = useState(false);
  const branchesTimeoutRef = useRef(null);
  const branchInputRef = useRef(null);
  const branchDropdownRef = useRef(null);
  const selectingSettlementRef = useRef(false);
  const [postomats, setPostomats] = useState([]);
  const [showPostomatsDropdown, setShowPostomatsDropdown] = useState(false);
  const [loadingPostomats, setLoadingPostomats] = useState(false);
  const postomatTimeoutRef = useRef(null);
  const postomatInputRef = useRef(null);
  const postomatDropdownRef = useRef(null);
  const selectingPostomatRef = useRef(false);
  const [warehousesCache, setWarehousesCache] = useState({});
  const resetDeliveryData = () => {
    setDeliveryData(createEmptyDeliveryData());
  };
```

Рисунок 3.24 – Визначення станів для пошуку через API Нової Пошти

Функція handleDeliveryDataChange викликається при зміні значення в полі введення (місто, відділення, поштомат). Вона використовує debounce (затримка 300 мс) для оптимізації кількості запитів до API. При введенні назви міста вона викликає novaPoshtaAPI.getSettlements для пошуку населених пунктів (рис. 3.25).

При введенні назви відділення або поштомату функція спочатку перевіряє кеш, і якщо результатів немає, викликає novaPoshtaAPI.getWarehouses для пошуку відділень/поштоматів. Результати зберігаються в кеші для подальшого використання (рис. 3.26).

```
settlementsTimeoutRef.current = setTimeout(async () => {
  try {
    setLoadingSettlements(true);
    const response = await novaPoshtaAPI.getSettlements(value);
    const allSettlements = response.data || [];
    // Фільтруємо дублікати
    const uniqueSettlements = filterUniqueSettlements(allSettlements);
    setSettlements(uniqueSettlements);
    setShowSettlementsDropdown(uniqueSettlements.length > 0);
  } catch (error) {
    console.error('Error fetching settlements:', error);
    setSettlements([]);
  } finally {
    setLoadingSettlements(false);
  }
}, 300);
```

Рисунок 3.25 – Функція пошуку населених пунктів з debounce

```
branchesTimeoutRef.current = setTimeout(async () => {
  try {
    let warehousesList = warehousesCache[cacheKey];
    if (!warehousesList) {
      setLoadingBranches(true);
      const response = await novaPoshtaAPI.getWarehouses({
        city: trimmedCity,
        category: 'branch',
      });
      warehousesList = response.data || [];

      if (warehousesList && warehousesList.length > 0) {
        setWarehousesCache(prev => ({
          ...prev,
          [cacheKey]: warehousesList,
        }));
      }
    }

    const filtered = filterWarehousesByQuery(warehousesList || [], searchValue);
    setBranches(filtered);
    setShowBranchesDropdown(filtered.length > 0);
  } catch (error) {
    console.error('Error fetching warehouses:', error);
    setBranches([]);
    setShowBranchesDropdown(false);
  } finally {
    setLoadingBranches(false);
  }
}, 300);
```

Рисунок 3.26 – Функція пошуку відділень з кешуванням

Функція handleSettlementSelect викликається при виборі населеного пункту зі списку. Вона встановлює вибраний населений пункт в стан deliveryData, зберігає cityRef (посилання на населений пункт в API Нової Пошти) та очищає поля відділення/поштомату для нового пошуку.

В рендерингу компонента Checkout відображається поле введення для пошуку населеного пункту з автодоповненням. При введенні тексту відображається dropdown зі списком населених пунктів (до 5 результатів). Після вибору населеного пункту відображається поле введення для пошуку відділення або поштомату з автодоповненням. Результати пошуку кешуються для зменшення кількості запитів до API.

3.5.4 Використання платіжної системи LiqPay на клієнті

Інтеграція з платіжною системою LiqPay на клієнті дозволяє користувачам оплачувати замовлення через безпечний платіжний шлюз. Інтеграція реалізована в компоненті Checkout.jsx з використанням динамічного створення форми для відправки даних до LiqPay.

Функція submitLiqpayForm викликається для створення та відправки форми оплати до LiqPay. Вона приймає paymentData (об’єкт з url, data та signature) з сервера, перевіряє всі необхідні поля, створює HTML-форму динамічно та додає приховані поля для data та signature (рис. 3.27).

Потім вона додає форму до document.body та викликає form.submit() для автоматичної відправки форми, що перенаправляє користувача на сторінку оплати LiqPay.

Функція handleSubmit викликається при підтвердженні оформлення замовлення. Вона валідує форму, створює або оновлює замовлення в базі даних, викликає ordersAPI.createLiqpayPayment для отримання payload від сервера та викликає submitLiqpayForm для відправки форми до LiqPay (рис. 3.28).

```
const submitLiqpayForm = (paymentData) => {
  if (!paymentData || !paymentData.data || !paymentData.signature || !paymentData.url) {
    throw new Error('Некоректна відповідь LiqPay');
  }

  const form = document.createElement('form');
  form.method = 'POST';
  form.action = paymentData.url;
  form.acceptCharset = 'utf-8';

  const dataInput = document.createElement('input');
  dataInput.type = 'hidden';
  dataInput.name = 'data';
  dataInput.value = paymentData.data;
  form.appendChild(dataInput);

  const signatureInput = document.createElement('input');
  signatureInput.type = 'hidden';
  signatureInput.name = 'signature';
  signatureInput.value = paymentData.signature;
  form.appendChild(signatureInput);

  document.body.appendChild(form);
  form.submit();
};
```

Рисунок 3.27 – Функція submitLiqpayForm для створення форми оплати

```
const handleSubmit = async (e) => {
  if (e) {
    e.preventDefault();
  }

  const validationErrors = validateForm();
  if (validationErrors.length > 0) {
    setErrors(validationErrors);
    return;
  }

  try {
    setSubmitting(true);
    setErrors([]);

    let orderIdForPayment = order?.id || id;
    let accessTokenForOrder = null;

    const shippingAddress = buildShippingAddress();

    let createdOrder;
    if (user) {
      createdOrder = await prepareAuthorizedOrder(shippingAddress);
      orderIdForPayment = createdOrder.id;
      accessTokenForOrder = null;

      if (createdOrder.number && createdOrder.number !== id) {
        navigate(`/order/${createdOrder.number}/checkout`, { replace: true });
      }
    } else {
      createdOrder = await prepareUnauthorizedOrder(shippingAddress);
      orderIdForPayment = createdOrder.id;
      accessTokenForOrder = createdOrder.access_token || null;

      if (createdOrder.number && createdOrder.number !== id) {
        navigate(`/order/${createdOrder.number}/checkout`, { replace: true });
      }
    }

    if (!user) {
      const orderTokens = JSON.parse(localStorage.getItem('order_tokens') || '{}');
      if (!accessTokenForOrder) {
        accessTokenForOrder = orderTokens[orderIdForPayment];
      }
    }

    const paymentData = await ordersAPI.createLiqpayPayment(orderIdForPayment, accessTokenForOrder);
    submitLiqpayForm(paymentData);
  } catch (error) {
    console.error('Error initiating LiqPay payment:', error);
    const errorMessage = error.response?.data?.error || error.message || 'Не вдалося ініціювати оплату LiqPay';
    setErrors([errorMessage]);
  } finally {
    setSubmitting(false);
  }
};
```

Рисунок 3.28 – Функція handleSubmit для ініціалізації оплати

В рендерингу компонента Checkout відображається розділ «Метод оплати (LiqPay)» з описом процесу оплати. При натисканні кнопки «Підтвердити та оплатити замовлення» викликається handleSubmit, яка ініціалізує процес оплати та перенаправляє користувача на сторінку оплати LiqPay.

Після успішної оплати LiqPay перенаправляє користувача на result_url, який веде на сторінку перегляду замовлення, де користувач може перевірити статус оплати. Сервер отримує callback від LiqPay на server_url та автоматично оновлює статус оплати замовлення в базі даних.

3.6 Приклади роботи програмного забезпечення

При відкритті торговельного майданчику на користувача чекає головна сторінка з каталогом товарів (рис. 3.29), де він може переглядати товари, фільтрувати їх за категоріями, ціною та магазинами, а також додавати товари до кошика. Для повноцінного використання функціоналу користувач має або зареєструватися, перейшовши на сторінку реєстрації (рис. 3.30), або ввійти з даними існуючого облікового запису, перейшовши на сторінку авторизації (рис. 3.31).

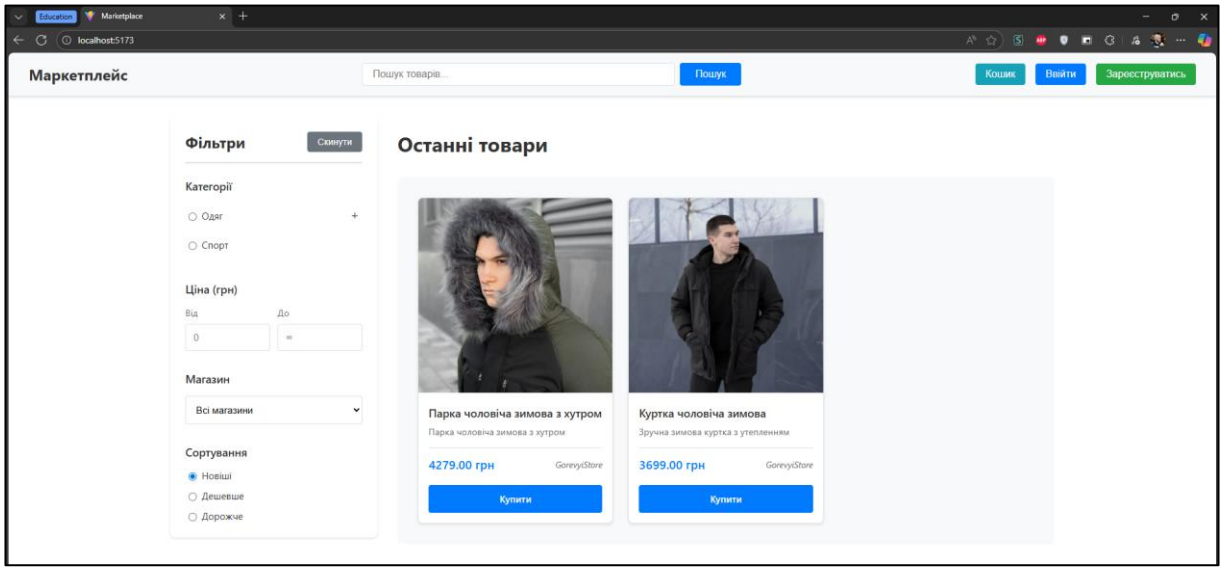


Рисунок 3.29 – Вигляд головної сторінки торговельного майданчику

На головній сторінці відображається каталог товарів з можливістю фільтрації за категоріями, ціною та магазинами. Користувач може переглядати товари, додавати їх до кошика та переходити на сторінку детального перегляду товару.

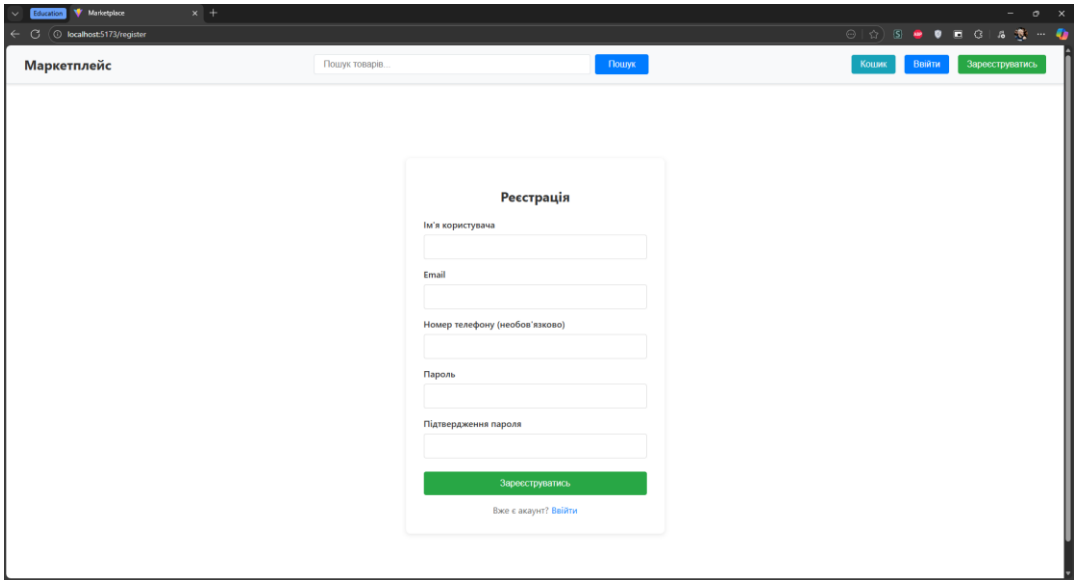


Рисунок 3.30 – Вигляд сторінки реєстрації

На сторінці реєстрації користувач має заповнити форму з полями: ім’я користувача, email та пароль. Після успішної реєстрації користувач автоматично авторизується в системі та отримує роль «покупець».

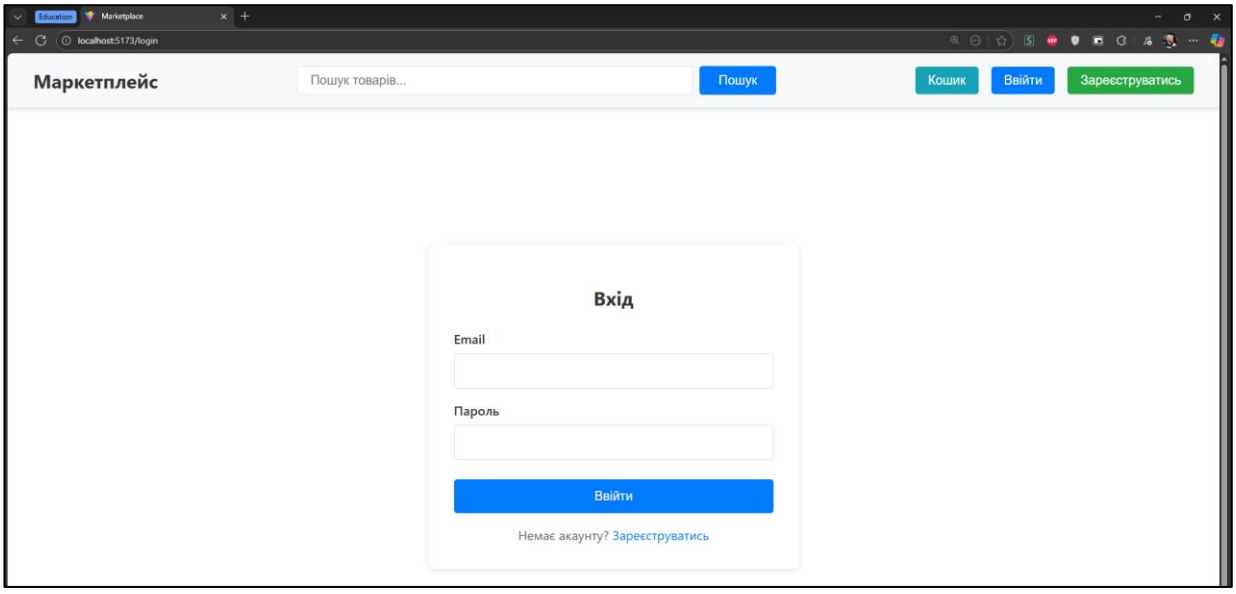


Рисунок 3.31 – Вигляд сторінки авторизації

На сторінці авторизації користувач вводить email та пароль для входу в систему. Після успішної авторизації система створює сесію та перенаправляє користувача на головну сторінку.

На навігаційній панелі «Navbar» (рис. 3.32), відображаються посилання на головну сторінку, кошик (з кількістю товарів), особистий кабінет та кнопка виходу з системи. Для продавців та адміністраторів додатково відображаються посилання на кабінет продавця та адмін-панель відповідно.

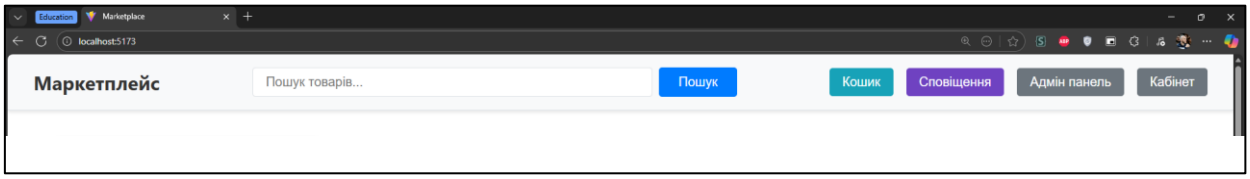


Рисунок 3.32 – Вигляд навігаційної панелі

3.6.1 Функціонал для ролі «Покупець»

Покупець може додавати товари до кошика з головної сторінки (рис. 3.33) або зі сторінки детального перегляду товару. При натисканні кнопки «Додати до кошика» товар додається до кошика, а кількість товарів у кошику оновлюється в навігаційній панелі. Для авторизованих користувачів товари зберігаються в базі даних, для неавторизованих – в localStorage браузера.

Після додавання товарів до кошика покупець може перейти до кошика (рис. 3.34), де відображаються всі товари, згруповані за магазинами. Користувач може змінювати кількість товарів, видаляти товари з кошика та оформлювати замовлення для кожного магазину окремо.

При натисканні кнопки «Оформити замовлення» для групи товарів одного магазину користувач перенаправляється на сторінку оформлення замовлення (рис. 3.35), де має заповнити форму з даними для доставки: ім’я, email, телефон та адресу доставки.

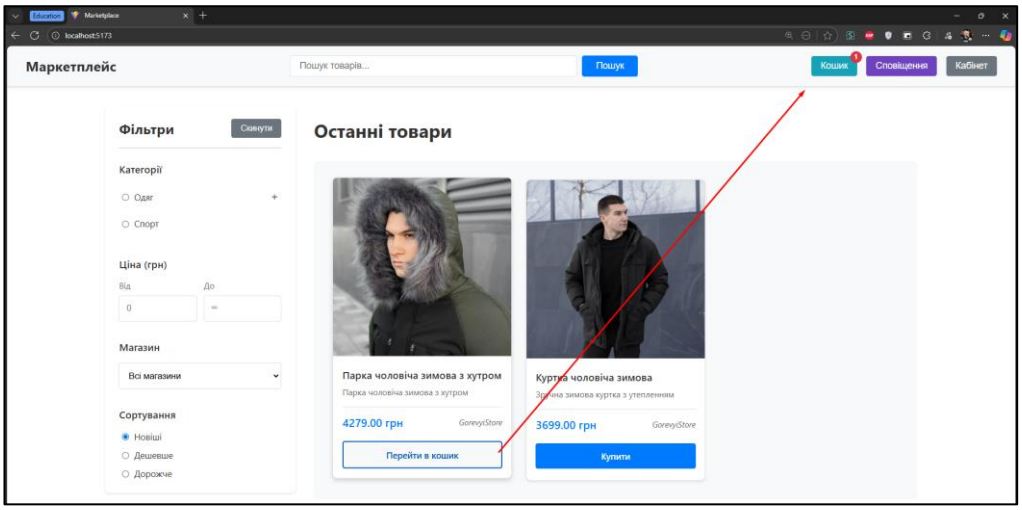


Рисунок 3.33 – Додавання товару до кошика

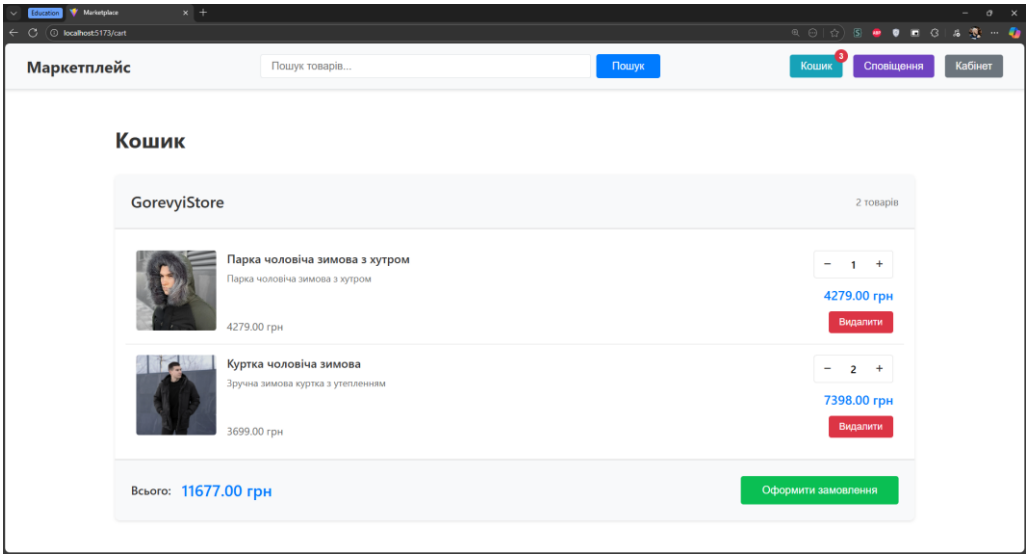


Рисунок 3.34 – Вигляд кошика з товарами

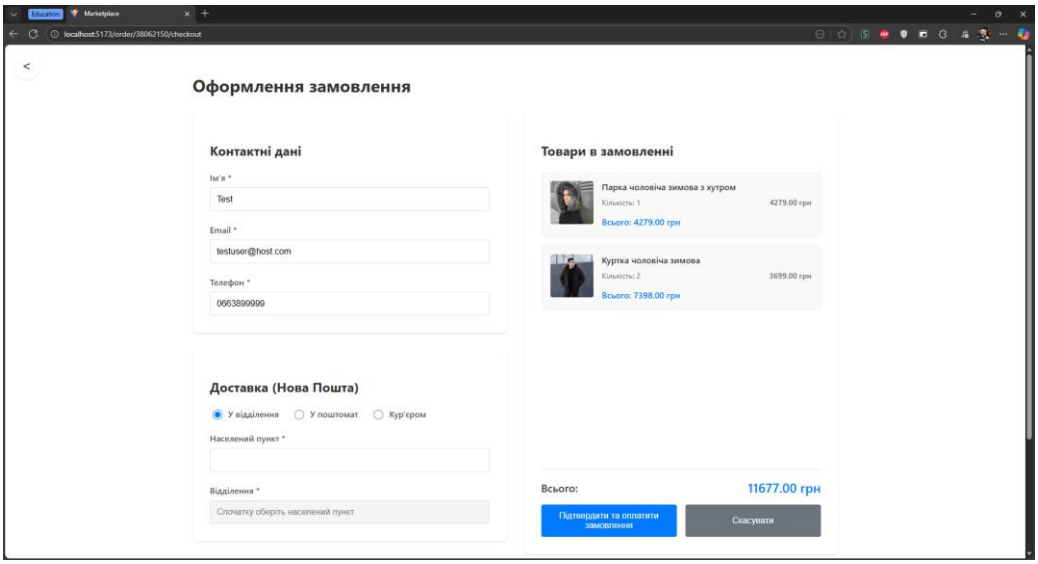


Рисунок 3.35 – Форма оформлення замовлення

На сторінці оформлення замовлення користувач може обрати тип доставки: відділення Нової Пошти, поштомат або кур’єрська доставка. При виборі відділення або поштомату користувач має спочатку знайти населений пункт через пошук (рис. 3.36). Система використовує API Нової Пошти для пошуку населених пунктів з автодоповненням. При введенні назви міста відображається випадаючий список з результатами пошуку.

Після вибору населеного пункту користувач може шукати відділення або поштомат (рис. 3.37). Система використовує API Нової Пошти для пошуку відділень/поштоматів з можливістю фільтрації за номером або описом. Результати пошуку кешуються для зменшення кількості запитів до API.

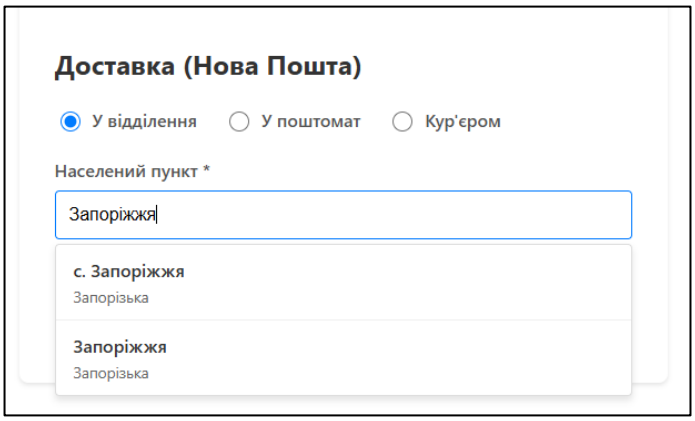


Рисунок 3.36 – Пошук населеного пункту через API Нової Пошти

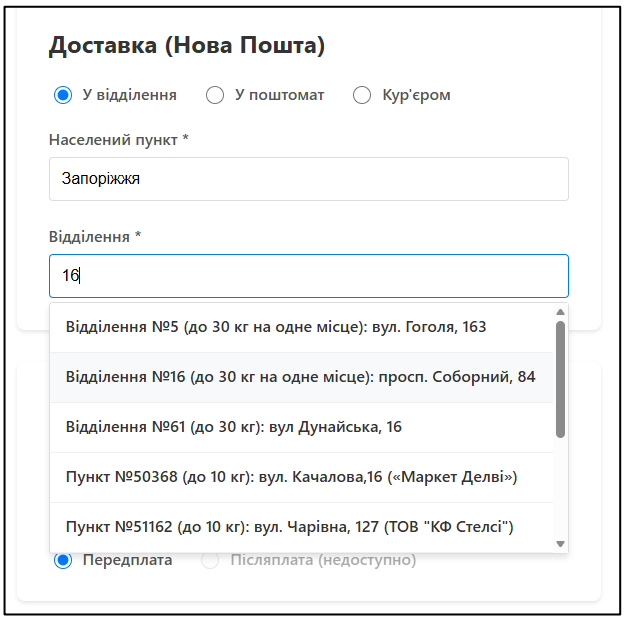


Рисунок 3.37 – Пошук відділення Нової Пошти

Після заповнення всіх даних для доставки та вибору відділення/поштомату користувач натискає кнопку «Підтвердити та оплатити замовлення». Система створює замовлення в базі даних та ініціалізує оплату через LiqPay (рис. 3.38). Користувач перенаправляється на сторінку оплати LiqPay, де може виконати оплату замовлення.

Після успішної оплати користувач перенаправляється на сторінку перегляду замовлення (рис. 3.39), де може перевірити статус замовлення та оплати. Статус оплати автоматично оновлюється на сервері після отримання callback від LiqPay.

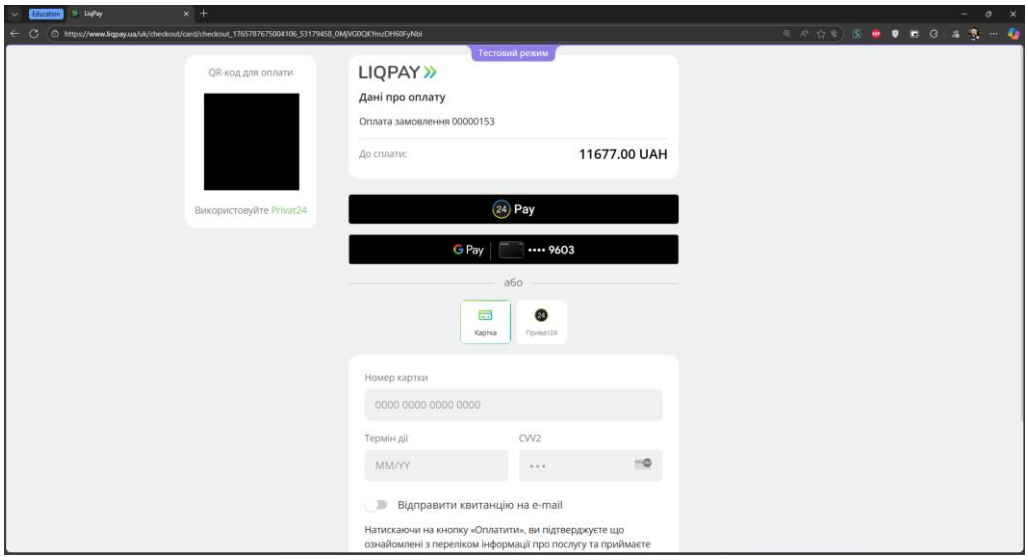


Рисунок 3.38 – Ініціалізація оплати через LiqPay

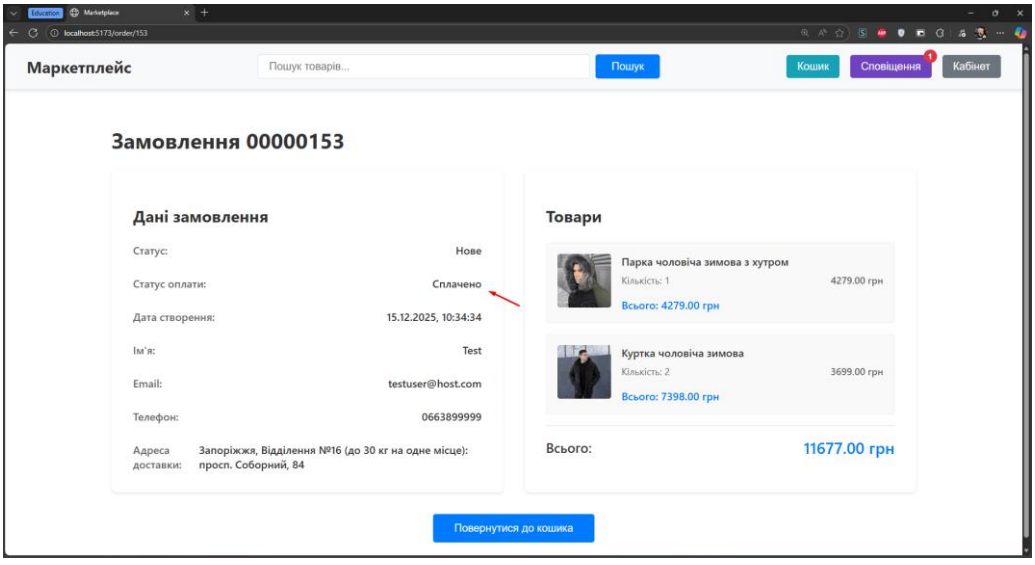


Рисунок 3.39 – Вигляд замовлення після оплати

Покупець може переглядати сповіщення про зміну статусу замовлень через бічне меню сповіщень (рис. 3.40). Сповіщення відображаються при створенні нового замовлення, зміні статусу замовлення та оплаті замовлення. Користувач може позначити сповіщення як прочитані або позначити всі сповіщення як прочитані.

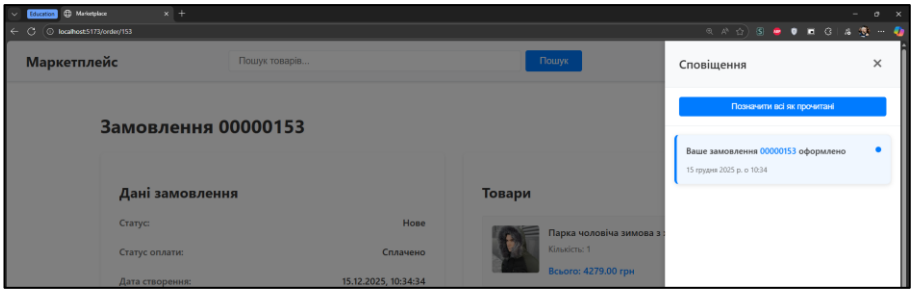


Рисунок 3.40 – Вигляд сповіщень покупця

3.6.2 Функціонал для ролі «Продавець»

Користувач з роллю «покупець» може створити магазин, перейшовши в особистий кабінет та натиснувши кнопку «Створити магазин» (рис. 3.41). При створенні магазину користувач має заповнити форму з назвою магазину. Після успішного створення магазину роль користувача автоматично змінюється на «продавець».

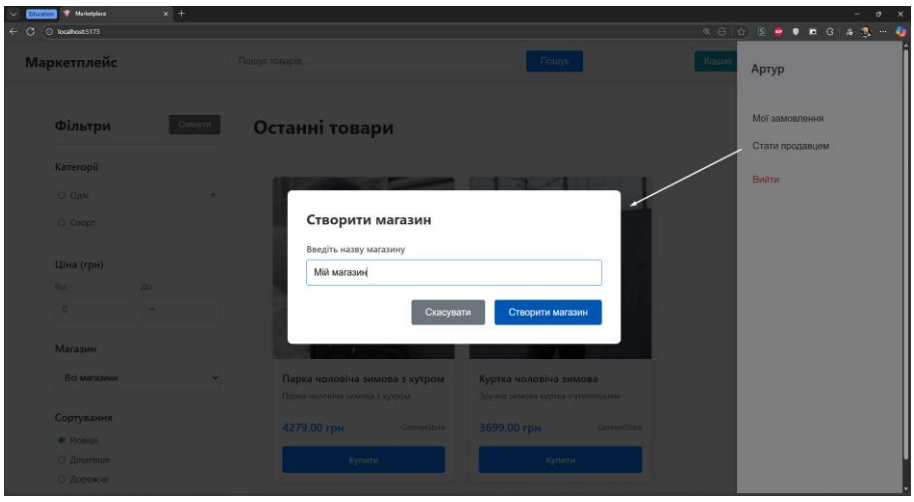


Рисунок 3.41 – Форма створення магазину

Продавець може додавати товари в свій магазин через кабінет продавця (рис. 3.42). На сторінці створення товару продавець має заповнити форму з полями: назва товару, опис, ціна, кількість на складі, категорія та завантажити зображення товару (до 5 зображень). Після збереження товар відображається в каталозі та доступний для покупців.

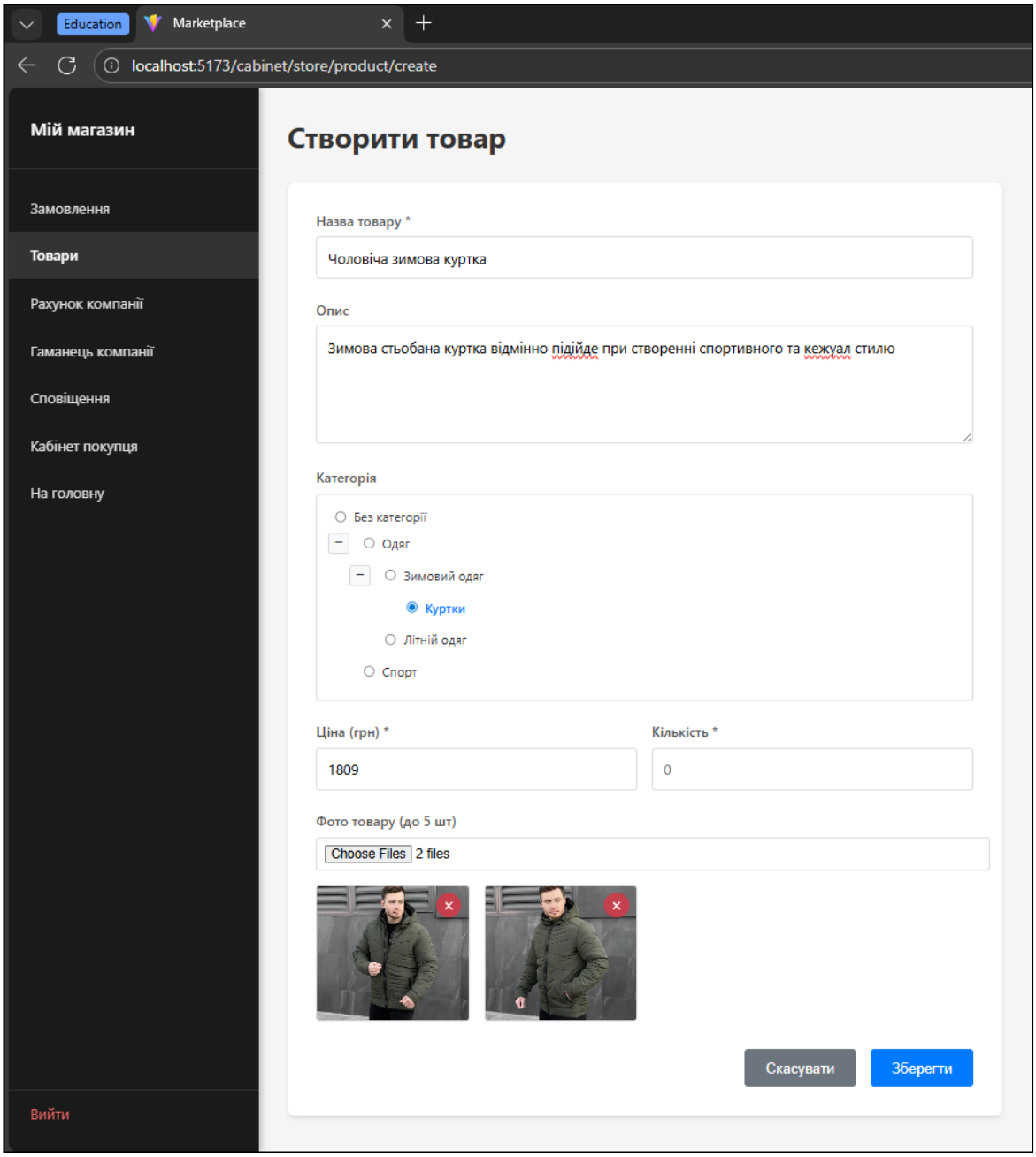


Рисунок 3.42 – Форма додавання товару в магазин

Продавець може переглядати всі замовлення свого магазину в кабінеті продавця (рис. 3.43). На сторінці замовлень відображається список замовлень

з інформацією про номер замовлення, дату створення, статус, клієнта та загальну суму. Продавець може змінювати статус замовлення: «Нове» > «Прийнято» > «Виконано» або скасувати замовлення.

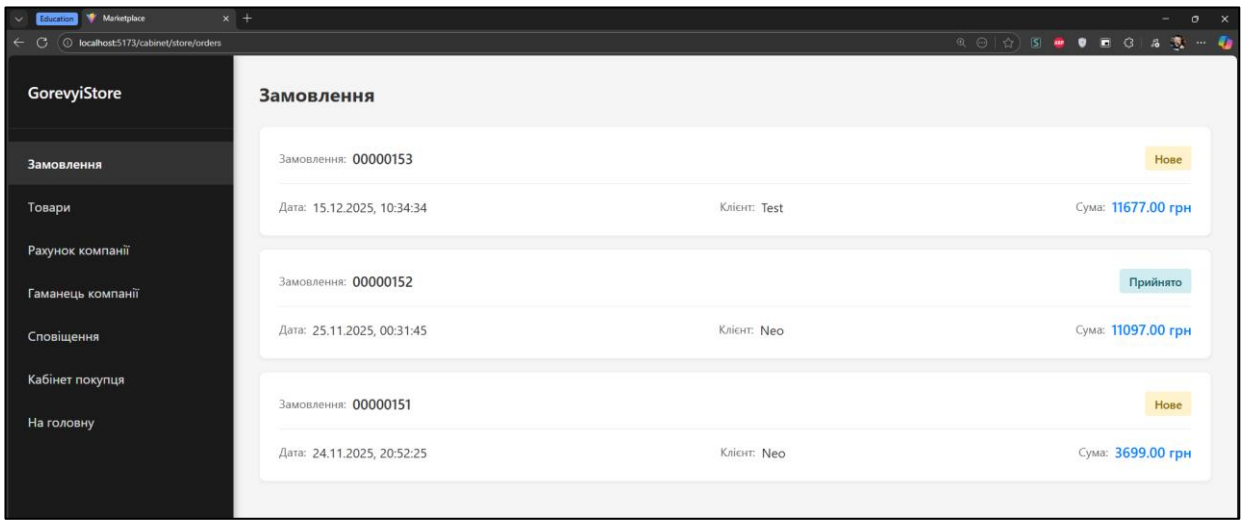


Рисунок 3.43 – Список замовлень продавця

При натисканні на замовлення продавець переходить на сторінку детального перегляду замовлення (рис. 3.44), де може переглянути повну інформацію про замовлення: дані клієнта, адресу доставки, список товарів, статус оплати та змінити статус замовлення.

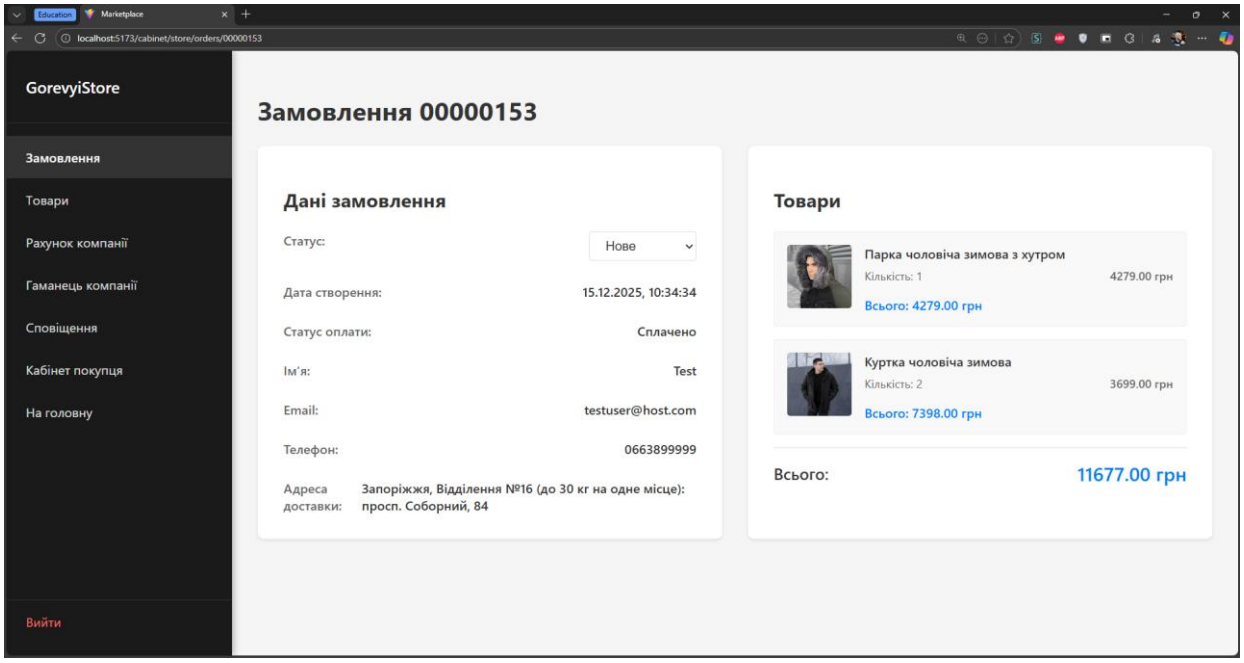


Рисунок 3.44 – Детальний перегляд замовлення продавця

3.6.3 Функціонал для ролі «Адміністратор»

Адміністратор може додавати нові категорії товарів через адмін-панель (рис. 3.45). На сторінці управління категоріями відображається дерево категорій з можливістю створення нових категорій, редагування існуючих та видалення категорій. При створенні нової категорії адміністратор має вказати назву категорії та оберіть батьківську категорію (якщо це підкатегорія).

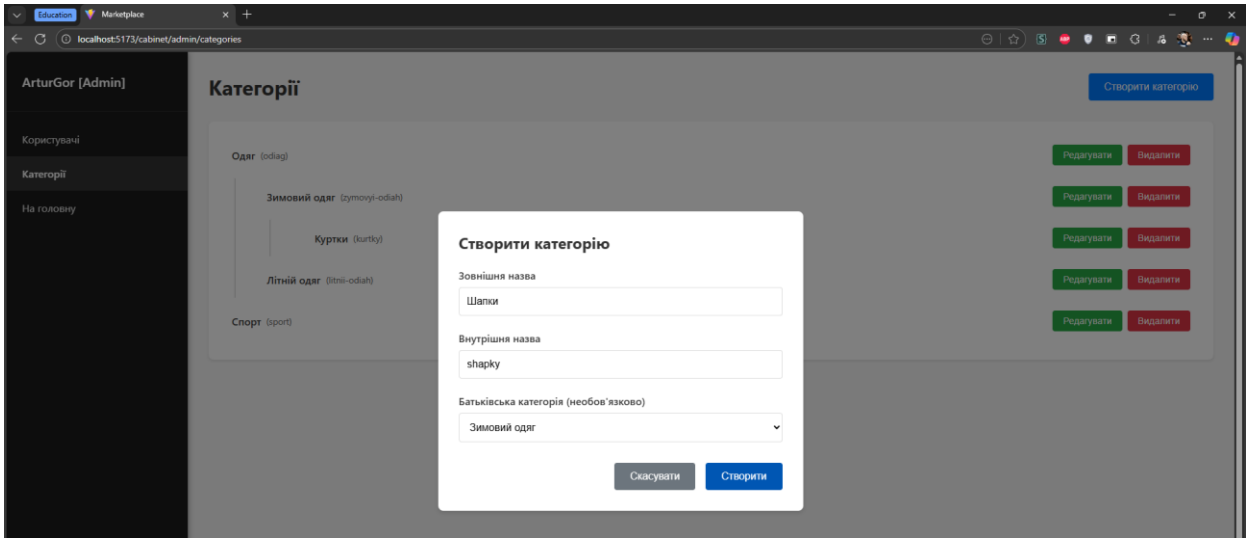


Рисунок 3.45 – Управління категоріями в адмін-панелі

Адміністратор також має доступ до перегляду списку всіх користувачів системи (рис. 3.46) та може переглядати інформацію про користувачів, включаючи їх роль, дату реєстрації та кількість замовлень.

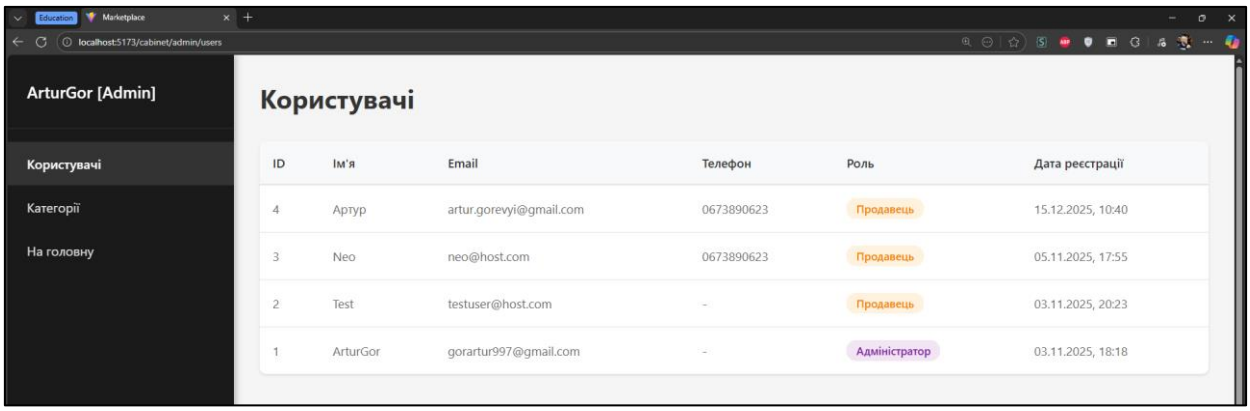


Рисунок 3.46 – Список користувачів в адмін-панелі

ВИСНОВКИ

В результаті кваліфікаційної роботи створено повнофункціональний вебзастосунок торговельного онлайн-майданчику на Ruby on Rails, React/Vite та PostgreSQL.

Реалізовано повний цикл електронної комерції: реєстрація користувачів, відкриття магазинів, додавання товарів із категоріями, ціною, зображеннями, пошук/фільтри/сортування каталогу.

Кошик працює для авторизованих (БД) і гостей (localStorage) з автосинхронізацією після входу.

Оформлення замовлень включає валідацію даних, пошук адреси через API Нової пошти, генерацію унікального номера та токена для неавторизованих.

Платежі інтегровані з LiqPay: формування payload, redirect, callback, зарахування коштів на баланс магазину з обліком транзакцій і можливістю виведення.

Система сповіщень інформує про створення та зміну статусів замовлень.

Адміністратор керує ієрархією категорій та користувачами.

Спроєктовано БД, ключові модулі (MVC, REST API, контексти React), діаграми класів і ER.

Отримане рішення є гнучкою основою для подальшого розширення функціоналу та інтеграцій із зовнішніми сервісами.

ПЕРЕЛІК ПОСИЛАНЬ

1. Hartl M. Ruby on Rails Tutorial: Learn Web Development with Rails : 7th ed. Boston : Addison-Wesley Professional, 2022. 896 p.

2. Ruby S., Copeland D. B., Thomas D. Agile Web Development with Rails 6. Raleigh, NC : Pragmatic Bookshelf, 2020. 496 p.

3. Banks A., Porcello E. Learning React: Modern Patterns for Developing React Apps : 2nd ed. Sebastopol : O'Reilly Media, 2020. 350 p.

4. Thomas M. T. React in Action. Shelter Island, NY : Manning Publications, 2018. 360 p.

5. Fowler M. UML Distilled: A Brief Guide to the Standard Object Modeling Language : 3rd ed. Boston : Addison-Wesley, 2004. 208 p.

6. Booch G., Rumbaugh J., Jacobson I. The Unified Modeling Language User Guide : 2nd ed. Boston : Addison-Wesley, 2005. 496 p.

7. Fernandez O., Dohmen L., Aadland T. H. The Rails 7 Way : 1st ed. Boston : Addison-Wesley Professional, 2023. 750 p.

8. Connolly T., Begg C. Database Systems: A Practical Approach to Design, Implementation, and Management : 6th ed. Harlow : Pearson Education Limited, 2015. 1440 p.

9. Jin B., Sahni S., Shevat A. Designing Web APIs: Building APIs That Developers Love. Sebastopol : O'Reilly Media, 2018. 232 p.

10. Obe R. O., Hsu L. S. PostgreSQL: Up and Running: A Practical Guide to the Advanced Open Source Database : 3rd ed. Sebastopol : O'Reilly Media, 2017. 314 p.

11. Торговельний майданчик: що це таке та як ефективно продавати на онлайн-платформах? URL: https://wezom.com.ua/ua/blog/yak-prosuvati-biznes-na-marketpleysah (дата звернення: 28.04.2025).

12. What are the 2 Main Roles in eCommerce? URL: https://riseuplabs.com/what-are-the-two-main-roles-in-e-commerce/ (дата звернення: 15.04.2025).

13. Key Features of B2C eCommerce Platform. URL: https://www.yo-kart.com/blog/key-features-of-a-b2c-ecommerce-platform/ (дата звернення: 07.04.2025).

14. Що таке платіжні системи? URL: https://shelfy.com.ua/newsroom/shho-take-platizhni-sistemi/ (дата звернення: 23.05.2025).

15. Які переваги має Ruby над іншими мовами програмування? URL: https://rubydevelopers.org/t/ruby/150 (дата звернення: 11.06.2025).

16. Puma – A Fast, Concurrent Web Server for Ruby & Rack. URL: https://puma.io/ (дата звернення: 29.07.2025).

17. Active Storage default variant processor changed to :vips. URL: https://guides.rubyonrails.org/upgrading_ruby_on_rails.html (дата звернення: 13.08.2025).

18. Active Storage Overview (What is Active Storage?) URL: https://guides.rubyonrails.org/active_storage_overview.html (дата звернення: 02.09.2025).

19. Що таке Axios? URL: https://axios-http.com/uk/docs/intro (дата звернення: 19.10.2025).

20. Як Vite завоював світ веброзробки? URL: https://dev.ua/blogs/posts/tarabanov-1736410707 (дата звернення: 18.06.2025).

21. Vue.js Guide. URL: https://ua.vuejs.org/guide/scaling-up/tooling.html (дата звернення: 11.06.2025).

22. What is an Entity Relationship Diagram (ERD)? URL: https://www.lucidchart.com/pages/er-diagrams (дата звернення: 15.06.2025).