МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА ВЕБДОДАТКУ ДЛЯ УПРАВЛІННЯ ДОСТАВКОЮ З ВИКОРИСТАННЯМ NUXT.JS»**

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)
І.А. Карнаух
(ініціали та прізвище)

Керівник доцент кафедри програмної інженерії, PhD Столярова А.В.
(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

Карнауху Івану Андрійовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка вебдодатку для управління доставкою з використанням Nuxt.js

керівник роботи Столярова Анастасія Валеріївна, PhD
(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. Аналіз технологій та сучасних підходів у розробці вебсервісів.
2. Постановка задачі та проєктування системи.
3. Реалізація вебдодатку засобами Nuxt.JS.
4. Тестування та оцінка ефективності системи.

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.	28.03.2025	
2.	Збір вихідних даних.	25.04.2025	
3.	Обробка методичних та теоретичних	13.06.2025	
	джерел.		
4.	Розробка першого та другого розділу.	29.08.2025	
5.	Розробка третього та четвертого розділу.	24.11.2025	
6.	Оформлення та нормоконтроль	01.12.2025	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	17.12.2025	

Студент _____ І.А. Карнаух
(підпис) (ініціали та прізвище)

Керівник роботи _____ А.В. Столярова
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

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

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка вебдодатку для управління доставкою з використанням Nuxt.js»: 96 с., 41 рис., 10 джерел, 6 додатків.

БАЗА ДАНИХ, ВЕБДОДАТОК, ВЕБРОЗРОБКА, ДОСТАВКА, ЕЛЕКТРОННА КОМЕРЦІЯ, NODE.JS, NUXT, REST API, SEO, SPA, SSR VUE.JS.

Об’єкт дослідження – процес автоматизації керування доставкою за допомогою вебзастосунків.

Предмет дослідження – архітектурні, функціональні та технологічні особливості розробки вебдодатку для управління доставкою на основі Nuxt.js.

Мета роботи: розробка комплексного вебдодатку для управління процесами доставки.

Методи дослідження – методи збору та аналізу вимог до програмного забезпечення, методи моделювання, методи проєктування, методи конструювання, методи тестування програмного забезпечення.

У процесі розробки вебдодатку було проведено аналіз предметної області доставки, визначено вимоги до системи та обґрунтовано вибір архітектури. Система реалізована за клієнт-серверною моделлю: клієнтська частина створена з використанням Nuxt.js, серверна – з використанням Node.js та REST API. Було виконано проєктування бази даних, реалізовано основні функції управління доставкою та проведено тестування. У результаті розроблено вебдодаток для управління процесами доставки товарів.

SUMMARY

Master’s qualifying paper “Development of a Web Application for Delivery Management Using Nuxt.js”: 96 pages, 41 figures, 10 references, 6 supplements.

DATABASE, DELIVERY, E-COMMERCE, NODE.JS, NUXT, REST API, SEO, SPA, SSR, VUE.JS, WEB APPLICATION, WEB DEVELOPMENT.

The object of the study is the process of automating delivery management using web applications.

The subject of the research is the architectural, functional, and technological aspects of developing a delivery management web application based on Nuxt.js.

The aim of the study is to develop a comprehensive web application for managing delivery processes.

The research methods include requirements gathering and analysis techniques, modeling methods, design methods, software construction techniques, and software testing methods.

During the development of the web application, an analysis of the delivery domain was conducted, system requirements were defined, and the architecture was justified. The system is implemented using a client-server model: the client side is developed with Nuxt.js, and the server side is based on Node.js and REST API. A database schema was designed, key delivery management functions were implemented, and system testing was conducted. As a result, a functional web application for managing goods delivery processes was developed.

ЗМІСТ

Завдання на кваліфікаційну роботу……..…..2
Реферат 4
Summary 5
Скорочення та умовні познаки 8
Вступ..... 9
1 Аналіз технологій та сучасних підходів у розробці вебсервісів 12
1.1 Тенденції розвитку вебдодатків: SPA, SSR, JAMstack 12
1.2 Особливості JavaScript-фреймворків: Vue.js, Nuxt.js, їх переваги та обмеження..... 14
1.3 Аналіз існуючих платформ для управління доставкою..... 18
1.4 Обґрунтування вибору Nuxt.js для реалізації системи 20
2 Постановка задачі та проєктування системи..... 25
2.1 Мета, завдання та вимоги до вебдодатку 25
2.2 Архітектурна модель системи: клієнт-серверна взаємодія 26
2.2.1 Архітектурна модель вебдодатку..... 26
2.2.2 Клієнтська частина системи..... 27
2.2.3 Користувацький інтерфейс та інтерактивна візуалізація 29
2.2.4 Архітектура серверної частини 30
2.2.5 Схема взаємодії компонентів..... 30
2.2.6 Масштабування системи 32
2.3 Прототипування та тестування..... 34
2.4 Схема бази даних та структура об’єктів доставки 35
2.4.1 Концептуальна модель даних 36
2.4.2 Логічна модель бази даних, індекси та оптимізація..... 37
2.4.3 Структура об’єктів доставки 37
2.4.4 Взаємодія між об’єктами..... 39
2.4.5 Сценарії використання бази даних..... 40

3 Реалізація вебдодатку засобами Nuxt.js 41
3.1 Ініціалізація проєкту: конфігурація Nuxt-додатку 41
3.2 Реалізація серверної логіки та взаємодія з базою даних 45
3.2.1 Інтеграція ORM Prisma: налаштування клієнта, генерація типів та Repository Pattern 48
3.2.2 Розробка REST API 50
3.2.3 Сервіси автентифікації 52
3.3 Клієнт і стан 53
4 Тестування та оцінка ефективності системи 63
4.1 Методика тестування та критерії оцінки ефективності 63
4.2 Результати тестування та аналіз ефективності системи 64
Висновки 66
Перелік посилань 68
Додаток А SQL-схема бази даних 69
Додаток Б Індекси 76
Додаток В Основні сценарії використання бази даних 77
Додаток Г Форма створення замовлення з динамічним списком товарів та підтвердженням 79
Додаток Д Сторінка кур’єра з маршрутом і списком замовлень 88
Додаток Е Репозиторії проєкту 90

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

API	Application Programming Interface
AWS	Amazon Web Services
CSRF	Cross-Site Request Forgery
DOM	Document Object Model
ETA	Estimated time of arrival
FAQ	Frequently Asked Questions
GRAP	GreenSock Animation Platform
HMR	Hot-Module Replacement
HTML	HyperText Markup Language
JSON	JavaScript Object Notation
JWT	JSON Web Tokens
NPM	Node Package Manager
ORM	Object Relation Maping
PWA	Progressive Web App
RBAC	Role Based Access Control
SCSS	Sassy Awesome Stylesheets
SPA	Single Page Applications
SQL	Structured Query Language
SSG	Static Site Generation
SSR	Server Side Rendering
XSS	Cross Site Scripting
ТТН	Товарно-транспортна накладна

ВСТУП

У сучасному світі інформаційні технології стрімко інтегруються в усі сфери життя людини, зокрема в логістику та сферу доставки товарів. Одним із найважливіших інструментів для автоматизації таких процесів є вебдодатки, які дозволяють ефективно керувати замовленнями, відстежувати маршрути, взаємодіяти з клієнтами та аналітично обробляти дані.

Сервіси з доставки набули великого значення в Україні, особливо в умовах воєнного стану, коли критично важливою є швидка та надійна доставка гуманітарної допомоги, медичних засобів, продуктів харчування, технічного обладнання тощо. Також зросла потреба у внутрішніх логістичних рішеннях для малого та середнього бізнесу, який вимушено переходить у дистанційний формат або працює в умовах обмежень. У зв’язку з цим розробка інноваційних цифрових рішень у сфері логістики є вкрай **актуальною**.

Одним із сучасних підходів до створення інтерактивних, швидких та масштабованих вебдодатків є використання фреймворку Nuxt.js, який базується на Vue.js і дозволяє реалізовувати як клієнтську, так і серверну логіку. Це сприяє побудові універсальних застосунків із високою продуктивністю та зручною структурою.

Мета дослідження – створення функціонального вебдодатку для управління процесом доставки з використанням фреймворку Nuxt.js.

Досягнення поставленої мети передбачає реалізацію наступних **завдань**:

- проаналізувати існуючі рішення в сфері керування доставкою та визначити їхні недоліки й переваги;
- дослідити можливості фреймворку Nuxt.js для реалізації вебдодатків;
- розробити архітектуру вебдодатку для керування доставкою;
- спроєктувати користувацький інтерфейс та базу даних для збереження інформації про замовлення та маршрути;

– реалізувати основну функціональність вебдодатку відповідно до вимог;
– протестувати вебдодаток у різних сценаріях використання та оцінити його ефективність.

Об’єкт дослідження – процес автоматизації керування доставкою за допомогою вебзастосунків.

Предмет дослідження – архітектурні, функціональні та технологічні особливості розробки вебдодатку для управління доставкою на основі Nuxt.js.

Об’єктом дослідження виступає сама сфера логістичного програмного забезпечення, яка потребує вдосконалення через високі вимоги до точності, швидкості обробки замовлень та взаємодії з клієнтами. Предметом же виступає спосіб реалізації такого програмного забезпечення – від вибору стеку технологій до конкретних рішень інтерфейсу та бази даних, що безпосередньо впливають на зручність і функціональність системи доставки.

Методи дослідження – методи збору та аналізу вимог до програмного забезпечення, методи моделювання, методи проєктування, методи конструювання, методи тестування програмного забезпечення.

Для реалізації вебдодатку було використано мову програмування JavaScript, що дозволяє створити як клієнтську, так і серверну частину застосунку в єдиному середовищі. Як основний фреймворк застосовано Nuxt.js, який забезпечує зручну структуру проєкту, SSR (Server-Side Rendering) та можливість створення SEO-оптимізованих застосунків. Для зберігання інформації про замовлення та маршрути було обрано реляційну систему управління базами даних PostgreSQL, яка забезпечує високу надійність, відповідність ACID-властивостям та підтримує складні запити. Завдяки використанню типізованих таблиць та підтримці зв’язків між сутностями, PostgreSQL дозволяє ефективно структурувати дані, забезпечуючи цілісність і узгодженість інформації в системі доставки. Також були використані REST API для обміну інформацією між модулями системи.

Результатом кваліфікаційної роботи є створений вебдодаток, який дозволяє адміністратору та користувачам керувати процесами доставки,

переглядати замовлення, формувати маршрути, переглядати статуси доставки та вести базу клієнтів. Отримані результати мають як практичне значення (застосування в малому бізнесі, локальній логістиці, гуманітарних місіях), так і теоретичну цінність (як приклад використання Nuxt.js у реальному проєкті з акцентом на логістику).

Кваліфікаційна робота складається зі вступу, чотирьох розділів, висновків, переліку використаних джерел та додатків. Загальний обсяг роботи становить 96 сторінок. Додатки викладено на 28 сторінках. Перелік посилань складається з 10 найменувань.

Перший розділ присвячений аналізу предметної області, сучасних підходів до реалізації логістичних вебрішень. У другому розділі наведено технічне обґрунтування вибору стеку технологій, описано архітектуру, моделі даних, інтерфейс та взаємодію з API. У третьому розділі описано реалізацію вебдодатку та можливі напрямки його вдосконалення. У четвертому розділі описано методики та результати тестування.

У додатках наведено програмну реалізацію, приклади запитів та документацію API.

1 АНАЛІЗ ТЕХНОЛОГІЙ ТА СУЧАСНИХ ПІДХОДІВ У РОЗРОБЦІ ВЕБСЕРВІСІВ

1.1 Тенденції розвитку вебдодатків: SPA, SSR, JAMstack

Сучасна веброзробка характеризується різноманіттям підходів та архітектурних рішень, які постійно еволюціонують відповідно до зростаючих потреб користувачів та бізнесу. Серед найпоширеніших підходів до створення вебдодатків можна виділити SPA (Single Page Applications), SSR (Server-Side Rendering) та JAMstack архітектуру.

Односторінкові додатки (SPA) являють собою вебдодатки, які завантажуються один раз і потім динамічно оновлюють вміст сторінки без повного перезавантаження. Такий підхід забезпечує швидку реакцію інтерфейсу та створює враження роботи з десктопним додатком. SPA використовують JavaScript для маніпуляції DOM-елементами та здійснення асинхронних запитів до серверного API [1].

Переваги SPA:

– швидка взаємодія з користувачем після початкового завантаження;
– зменшення навантаження на сервер через відсутність необхідності постійного запиту повної HTML-сторінки;
– можливість створення складних інтерактивних інтерфейсів;
– чіткий розподіл клієнтської та серверної логіки.

Недоліки SPA:

– проблеми з SEO, оскільки пошукові системи можуть не індексувати контент, що генерується динамічно;
– довге початкове завантаження через необхідність отримання всього JavaScript-коду;
– складнощі з управлінням станом додатку при великих масштабах;
– підвищені вимоги до обчислювальних ресурсів клієнта.

Серверний рендеринг (SSR) – це підхід, при якому формування HTML-сторінки відбувається на сервері перед відправкою клієнту. Це дозволяє вирішити проблеми з SEO, характерні для SPA, та прискорити початкове завантаження сторінки [2].

Переваги SSR:

– кращі показники SEO, оскільки пошукові системи отримують вже сформований HTML;
– швидше відображення початкового контенту (First Contentful Paint);
– менше навантаження на клієнтський пристрій;
– краща продуктивність на слабких пристроях та при повільному інтернет-з’єднанні.

Недоліки SSR:

– збільшення навантаження на сервер;
– складніша розробка та налаштування;
– можливі затримки при переході між сторінками порівняно з SPA.

JAMstack (JavaScript, API, Markup) – сучасний архітектурний підхід до веброзробки [3], що фокусується на розділені фронтенду від серверної логіки. такі складаються з попередньо згенерованого HTML, динамічного JavaScript, та взаємодії із зовнішнім API.

Переваги JAMstack:

– висока продуктивність через статичну доставку контенту через CDN;
– підвищена безпека через відсутність безпосереднього зв’язку з базою даних;
– масштабованість та низькі витрати на хостинг;
– відмінні показники SEO для попередньо згенерованих сторінок.

Недоліки JAMstack:

– обмежені можливості для застосунків, які вимагають значної динамічної взаємодії;
– складність інтеграції з системами, що вимагають частих оновлень даних;

– потенційно довгий час збірки для великих проєктів;
– необхідність додаткових інструментів для реалізації динамічних функцій.

Сучасний підхід до розробки вебдодатків часто поєднує в собі різні архітектурні рішення, використовуючи їх переваги відповідно до конкретних потреб проєкту. Наприклад, гібридні рішення, такі як Nuxt.js для Vue.js, дозволяють поєднувати переваги SPA та SSR, а також підтримують статичну генерацію для реалізації принципів JAMstack.

Іншими важливими тенденціями у веброзробці є:

– прогресивні вебдодатки (PWA), що надають можливості, подібні до нативних мобільних додатків;
– мікрофронтенди, що дозволяють розділити великі додатки на незалежні модулі;
– serverless – архітектура, яка мінімізує необхідність управління серверами;
– WebAssembly, що забезпечує виконання високопродуктивного коду у браузері.

Для розробки системи управління доставкою необхідно обрати підхід, який забезпечить швидку взаємодію з користувачем, гарну індексацію пошуковими системами та можливість масштабування. Гібридний підхід з використанням SSR та елементів SPA видається найбільш оптимальним для такого типу проєктів, і саме тут Nuxt.js демонструє свої переваги [5].

1.2 Особливості JavaScript-фреймворків: Vue.js, Nuxt.js, їх переваги та обмеження

Сучасна розробка вебдодатків значно базується на використанні JavaScript-фреймворків, які спрощують створення інтерактивних і динамічних інтерфейсів. Серед найпопулярніших фреймворків можна виділити Vue.js та

його надбудову Nuxt.js, які зарекомендували себе як ефективні інструменти для розробки різноманітних вебдодатків.

Vue.js – це прогресивний JavaScript-фреймворк [4], розроблений для створення користувацьких інтерфейсів. Він фокусується на шарі представлення і може бути легко інтегрований в існуючі проєкти або використаний для розробки повноцінних SPA.

Розглянемо ключові **особливості Vue.js**.

Реактивність: Vue.js забезпечує автоматичне оновлення DOM при зміні даних. Це реалізується через систему реактивності, яка відстежує зміни в даних і автоматично оновлює пов’язані з ними елементи інтерфейсу.

Компонентна архітектура: фреймворк базується на компонентному підході, де кожен компонент містить власну логіку, шаблон і стилі. Це дозволяє створювати повторно використовувані елементи інтерфейсу та спрощує підтримку коду.

Віртуальний DOM: Vue.js використовує віртуальний DOM для оптимізації оновлень реального DOM, що значно підвищує продуктивність додатків.

Директиви: фреймворк надає вбудовані директиви (наприклад, v-if, v-for, v-bind), які спрощують маніпуляції з DOM і логіку відображення.

Простота вивчення: Vue.js має пологу криву навчання і доступну документацію, що робить його привабливим для початківців.

Гнучкість: Vue.js можна використовувати як для простих проєктів, так і для складних додатків, поступово впроваджуючи нові концепції та інструменти.

Переваги Vue.js:

– невеликий розмір (приблизно 20 КБ після мінімізації і стиснення);
– висока продуктивність завдяки оптимізованому рендерингу;
– детальна і зрозуміла документація;
– гнучкість у використанні – від вбудовування окремих компонентів до повноцінних SPA;

– інтуїтивно зрозумілий синтаксис шаблонів;
– активна спільнота розробників.

Обмеження Vue.js:

– менша екосистема порівняно з React або Angular;
– обмежена підтримка TypeScript в ранніх версіях (покращено у Vue 3);
– відсутність вбудованої підтримки серверного рендерингу;
– недостатня кількість великих корпоративних користувачів порівняно з іншими фреймворками [5].

Розглянемо Nuxt.js та його особливості.

Nuxt.js – це безплатне середовище з відкритим кодом, яке розширює можливості Vue.js, забезпечуючи зручний і гнучкий спосіб створення високопродуктивних, масштабованих вебзастосунків і сайтів [5]. Nuxt.js вирішує багато обмежень чистого Vue.js, додаючи серверний рендеринг, статичну генерацію сайтів, автоматичну маршрутизацію та інші корисні функції.

Наведемо ключові **особливості Nuxt.js**.

Маршрутизація на основі файлів: Nuxt автоматично визначає маршрути, спираючись на структуру каталогу pages/. Це спрощує організацію застосунку та знижує потребу в ручному налаштуванні маршрутів.

Розподіл коду: Nuxt автоматично розділяє код на менші фрагменти (chunks), що допомагає скоротити час завантаження застосунку.

Рендеринг на стороні сервера (SSR): вбудовані можливості SSR дозволяють здійснити рендеринг без необхідності налаштовувати окремий сервер, що покращує SEO та початкову швидкість завантаження.

Підтримка TypeScript: Nuxt дає змогу писати типобезпечний код без необхідності додаткових налаштувань, завдяки автоматично згенерованим типам і файлу tsconfig.json.

Автоматичний імпорт: можливість додавати уніфіковані методи (composables) та компоненти Vue у відповідні каталоги й використовувати їх

без потреби імпортувати, що дозволяє скористатись перевагами tree-shaking та оптимізованих JS-пакетів.

Налаштовані інструменти збірки: для розробки за замовчуванням використовується Vite, який підтримує гарячу заміну модулів (HMR). Для підготовки коду до розгортання на продуктивному середовищі застосовуються необхідні налаштування, що забезпечують ефективність роботи застосунку» [5].

Статична генерація сайтів (SSG): Nuxt.js дозволяє генерувати статичні HTML-файли на етапі збірки, які можна розгорнути на будь-якому статичному хостингу.

Модульна архітектура: модульна архітектура розбиває застосунок на автономні, повторно використовувані одиниці – модулі – які інкапсулюють певну функціональність. Кожен модуль може розроблятись, тестуватись і підтримуватись незалежно. У Nuxt модулі можуть підключатися до життєвого циклу фреймворка, розширювати основні функції або додавати сторонні інтеграції [6].

Переваги Nuxt.js:

– поєднання переваг SPA, SSR та SSG в одному фреймворку;
– покращені показники SEO завдяки серверному рендерингу;
– автоматична маршрутизація, що спрощує структуру проєкту;
– оптимізація продуктивності через автоматичний розподіл коду;
– широкі можливості для налаштування і розширення функціональності;
– вбудована підтримка асинхронних даних та метаданих.

Обмеження Nuxt.js:

– більш складна архітектура порівняно з чистим Vue.js;
– додаткові вимоги до хостингу для SSR-режиму;
– потенційно довший час збірки для великих проєктів в режимі SSG;
– необхідність глибшого розуміння життєвого циклу додатку;
– обмеження при роботі з певними браузерними API в SSR-режимі.

Порівняння Vue.js і Nuxt.js. Вибір між Vue.js та Nuxt.js залежить від вимог і складності проєкту. Vue.js підходить для невеликих та середніх проєктів, де важливо швидко реалізувати інтерфейси. Це варіант для доповнення вже готових рішень, створення односторінкових застосунків або покращення статичних сайтів. Фреймворк також підходить для проєктів, які не вимагають складної серверної логіки або статичної генерації сайтів.

Nuxt.js варто обирати для складних проєктів завдяки підтримці серверного рендерингу, автоматичному розподілу коду та маршрутизації на основі структури файлів. Автоматизація багатьох процесів дозволяє прискорити розробку масштабованих вебзастосунків. Застосування Nuxt.js доцільне й для проєктів з високими вимогами до SEO або швидкості завантаження.

Для системи управління доставкою, яка потребує якісної взаємодії з користувачем, оптимізації для пошукових систем та ефективної роботи з даними, Nuxt.js надає більш повний набір інструментів порівняно з чистим Vue.js.

1.3 Аналіз існуючих платформ для управління доставкою

Сучасний ринок програмного забезпечення для управління доставкою характеризується високою конкуренцією та різноманіттям рішень, що варіюються від спеціалізованих продуктів до комплексних екосистем. Серед міжнародних гравців виділяються платформи AllRide Apps та Shipsy Logistics, що пропонують повний цикл управління автопарком і логістикою, а також SmartRoutes, ключовою перевагою якої є потужні алгоритми маршрутизації. Для сегмента малого та середнього бізнесу популярним є DelivApp, тоді як Trakop фокусується на трекінгу в реальному часі та прогнозуванні, а Water Delivery Solutions закриває потреби вузькопрофільних сервісів доставки води [7]. Вагоме місце займають і українські розробки, адаптовані до

локальних реалій: хмарний сервіс Ant-Logistics забезпечує автоматичну побудову маршрутів та GPS-контроль [10], а Dots Platform надає рішення для FoodTech та Retail з повним технологічним стеком (сайт, додатки, логістика). Аналіз функціональності цих платформ дозволяє виявити їхні сильні та слабкі сторони для формування оптимальних вимог до власної системи проєктування.

Основні функціональні можливості систем управління доставкою. Сучасні системи управління доставкою забезпечують повний цикл логістичних операцій, починаючи з гнучкого менеджменту замовлень, що включає їх створення, редагування, інтеграцію з платформами електронної комерції та розподіл між кур'єрами в ручному чи автоматичному режимах. Ключовою складовою є інтелектуальне планування логістики: програмне забезпечення будує оптимальні маршрути з урахуванням трафіку, вагогабаритних параметрів і часових вікон, дозволяючи візуалізувати рух на карті та перепланувати шлях у реальному часі. Процес виконання прозорий завдяки GPS-моніторингу, який забезпечує миттєве оновлення статусів, автоматичне сповіщення клієнтів та збір доказів вручення (фото або підпис). Для стратегічного управління система надає розгорнуту аналітику щодо витрат і продуктивності персоналу, а технічну гнучкість гарантують можливості автоматизації, експорту даних та інтеграції з іншими сервісами через API на різних платформах.

Архітектурні особливості існуючих систем. Більшість сучасних систем управління доставкою побудовані за принципом клієнт-серверної архітектури з вебінтерфейсом для адміністраторів та мобільними додатками для кур'єрів. При цьому можна виділити кілька підходів до реалізації.

Монолітна архітектура – вся логіка системи реалізована в рамках одного додатку. Перевагами є простота розробки та розгортання, а недоліками – складність масштабування та підтримки при зростанні системи.

Мікросервісна архітектура – система розділена на незалежні сервіси, кожен з яких відповідає за певну функціональність. Переваги – гнучкість,

масштабованість, можливість незалежного оновлення окремих компонентів, а недоліки – складність розробки та налаштування взаємодії між сервісами.

Гібридний підхід – поєднання елементів монолітної та мікросервісної архітектури, де основний функціонал реалізований як моноліт, а окремі компоненти (наприклад, аналітика чи сповіщення) – як мікросервіси.

Технологічний ландшафт сучасних систем управління доставкою базується на використанні передових JavaScript-фреймворків (React, Angular, Vue.js) та технологій PWA для фронтенду, тоді як серверну архітектуру зазвичай реалізують на Node.js, Python, Java або .NET. Інфраструктура таких рішень спирається на хмарні платформи (AWS, Google Cloud), надійні бази даних (PostgreSQL, MongoDB) та інтеграцію з геолокаційними API і сервісами сповіщень.

Проте, незважаючи на технологічну досконаліть, існуючі продукти мають низку суттєвих недоліків, що стримують їх масове впровадження, зокрема у малому бізнесі. Основними бар'єрами є висока вартість ліцензування та складність інтеграції в робочі процеси, що часто посилюється перевантаженим інтерфейсом і недостатньою гнучкістю кастомізації. Додатковими слабкими місцями залишаються слабка оптимізація під мобільні пристрої та обмежені аналітичні можливості, що не дозволяють повною мірою оцінити ефективність логістичних операцій.

Ці недоліки створюють можливості для розробки нової системи управління доставкою, яка буде більш гнучкою, доступною, зручною у використанні та ефективною з точки зору оптимізації бізнес-процесів.

1.4 Обґрунтування вибору Nuxt.js для реалізації системи

Розробка вебдодатку для управління доставкою вимагає ретельного вибору технологічного стеку, який забезпечить необхідну функціональність, продуктивність, масштабованість та зручність у розробці та підтримці. В

цьому підрозділі обґрунтовується вибір Nuxt.js як основного фреймворку для реалізації системи.

Розглянемо **технічні переваги Nuxt.js** для системи управління доставкою.

Серверний рендеринг (SSR) та SEO-оптимізація. Система управління доставкою повинна мати публічну частину, доступну для клієнтів та пошукових систем. Вбудовані можливості SSR у Nuxt.js дозволяють здійснити рендеринг сторінок на сервері без необхідності налаштовувати окремий сервер, що значно покращує індексацію контенту пошуковими системами та початкову швидкість завантаження [5]. Це особливо важливо для сторінок відстеження замовлень, які мають бути доступні клієнтам без авторизації.

Автоматична маршрутизація та структурування проєкту. Nuxt.js автоматично визначає маршрути на основі структури каталогу *pages/*, що спрощує організацію додатку та знижує потребу в ручному налаштуванні маршрутів [5]. Для системи управління доставкою, яка містить багато різних екранів та функціональних модулів, така автоматизація значно спрощує розробку та підтримку.

Оптимізація продуктивності. Nuxt.js автоматично розділяє код на менші фрагменти (chunks), що допомагає скоротити час завантаження додатку [5]. Для системи управління доставкою, яка може містити складні компоненти для відображення карт, маршрутів, аналітики, така оптимізація критично важлива для забезпечення швидкої роботи інтерфейсу.

Модульна архітектура. Модульна архітектура Nuxt.js дозволяє легко додавати або покращувати функції, не порушуючи роботу всієї системи. Це забезпечує гнучкість у розробці та можливість поступового розширення функціональності системи управління доставкою.

Підтримка різних режимів рендерингу. Nuxt.js підтримує різні режими рендерингу: серверний (SSR), статична генерація (SSG) та клієнтський (SPA). Це дозволяє обрати оптимальний підхід для різних частин системи: SSR для публічних сторінок, SSG для статичного контенту (документація, FAQ) та SPA для адміністративної панелі, що потребує динамічної взаємодії.

Інтеграція з API та робота з даними. Nuxt.js надає зручні інструменти для роботи з API та управління даними, включаючи підтримку асинхронних запитів та зручні хуки життєвого циклу. Для системи управління доставкою, яка активно взаємодіє з різними API (геолокація, платіжні системи, SMS-сповіщення), ці можливості є надзвичайно важливими.

Розширені можливості для аутентифікації та авторизації. Система управління доставкою потребує надійної системи аутентифікації та авторизації з різними рівнями доступу для адміністраторів, менеджерів, кур'єрів та клієнтів. Nuxt.js надає гнучкі можливості для реалізації таких систем через middleware та модулі, що спрощує розробку безпечних механізмів контролю доступу.

Переваги Nuxt.js з точки зору розробки та підтримки.

Структурований підхід до організації коду. Nuxt.js пропонує чітку структуру проєкту з визначеними каталогами для різних компонентів системи (pages, components, layouts, store, middleware тощо), що спрощує організацію коду та полегшує орієнтацію нових розробників у проєкті.

Автоматичний імпорт компонентів та composables. Можливість додавати уніфіковані методи (composables) та компоненти Vue у відповідні каталоги й використовувати їх без потреби імпортувати, що дозволяє скористатись перевагами tree-shaking та оптимізованих JS-пакетів [5]. Це значно прискорює розробку та зменшує кількість шаблонного коду.

Підтримка TypeScript. Nuxt.js надає вбудовану підтримку TypeScript без необхідності додаткових налаштувань, завдяки автоматично згенерованим типам і файлу tsconfig.json [5]. Для складної системи управління доставкою використання TypeScript забезпечує кращу типізацію, що зменшує кількість помилок під час розробки та полегшує підтримку коду.

Активна спільнота та екосистема. Nuxt.js має активну спільноту розробників, багато готових модулів та компонентів, а також детальну документацію. Це спрощує вирішення потенційних проблем та прискорює розробку завдяки можливості використання готових рішень.

Порівняння Nuxt.js з альтернативними фреймворками для даного проєкту.

Nuxt.js vs. Next.js (React). Next.js є аналогом Nuxt.js для екосистеми React і також пропонує SSR, SSG та автоматичну маршрутизацію. Однак, для команди з досвідом роботи з Vue.js, Nuxt.js є більш природним вибором, оскільки зберігає всі переваги Vue (простий синтаксис, реактивність, компонентний підхід) і додає до них функціональність серверного рендерингу.

Nuxt.js та чистий Vue.js + Express. Реалізація SSR за допомогою чистого Vue.js та Express вимагає значно більше ручного налаштування та кодування порівняно з Nuxt.js, який надає готові рішення для SSR, маршрутизації, управління станом тощо. Використання Nuxt.js дозволяє зосередитися на бізнес-логіці додатку, а не на налаштуванні інфраструктури.

Nuxt.js та Angular. Angular пропонує повний набір інструментів для розробки складних додатків, але має значно крутішу криву навчання і вимагає більше шаблонного коду. Nuxt.js, базуючись на Vue.js, пропонує більш гнучкий та простий підхід, зберігаючи при цьому всі необхідні можливості для розробки складних систем.

Nuxt.js та традиційні бекенд-фреймворки (Django, Laravel). Традиційні бекенд-фреймворки з шаблонізаторами не забезпечують такого рівня інтерактивності та плавності користувацького інтерфейсу як сучасні JavaScript-фреймворки. Використання Nuxt.js дозволяє створити SPA-подібний досвід з перевагами SSR для SEO та початкової швидкості завантаження.

Враховуючи функціональні вимоги до системи управління доставкою, особливості розробки та підтримки, а також порівняння з альтернативними технологіями, Nuxt.js є оптимальним вибором для реалізації даного проєкту. Він надає всі необхідні інструменти для створення сучасного, швидкого, масштабованого та SEO-оптимізованого додатку, при цьому забезпечуючи зручність розробки та підтримки.

Можливість використання різних режимів рендерингу в рамках одного проєкту дозволяє гнучко підходити до оптимізації різних частин системи, а

модульна архітектура забезпечує легкість розширення функціональності та інтеграції з зовнішніми сервісами.

2 ПОСТАНОВКА ЗАДАЧІ ТА ПРОЄКТУВАННЯ СИСТЕМИ

2.1 Мета, завдання та вимоги до вебдодатку

Розробка вебдодатку для управління доставкою з використанням Nuxt.js спрямована на створення ефективного інструменту для оптимізації логістичних процесів, підвищення якості обслуговування клієнтів та автоматизації рутинних операцій. Для успішної реалізації проєкту необхідно чітко визначити мету, завдання та вимоги до системи.

Головною метою розробки комплексного вебдодатку є автоматизація та оптимізація логістичних процесів, що дозволить мінімізувати операційні витрати, підвищити якість обслуговування та забезпечити прозорість виконання замовлень на всіх етапах. Досягнення цієї мети реалізується через проєктування масштабованої архітектури та створення програмного забезпечення на базі стеку Node.js та Nuxt.js, яке охоплює повний цикл управління доставкою: від реєстрації та обробки замовлень до інтелектуального планування маршрутів і GPS-моніторингу кур’єрів у реальному часі.

Функціональне ядро системи передбачає розмежування прав доступу для адміністраторів, менеджерів, водіїв та клієнтів, забезпечуючи кожного зручними інструментами для взаємодії, включаючи систему сповіщень, інтерактивні карти та аналітичні модулі для оцінки ефективності роботи.

Особлива увага приділяється нефункціональним вимогам, таким як висока продуктивність, захист даних від кіберзагроз, відмовостійкість у режимі 24/7 та адаптивність інтерфейсу під будь-які пристрої.

Результатом має стати створення гнучкого інструменту для кур’єрських служб і бізнесу, який легко інтегрується із зовнішніми сервісами та масштабується відповідно до зростання навантаження.

2.2 Архітектурна модель системи: клієнт-серверна взаємодія

Архітектурна модель вебдодатку для управління доставкою базується на клієнт-серверній взаємодії, яка забезпечує ефективне розділення відповідальності між різними компонентами системи. Правильно спроєктована архітектура є ключовим фактором для забезпечення масштабованості, продуктивності та надійності додатку.

2.2.1 Архітектурна модель вебдодатку

Архітектура системи управління доставкою, представлена на рисунку 2.1, є комплексним рішенням, що базується на взаємодії клієнтської частини, реалізованої на Nuxt.js для забезпечення зручного інтерфейсу користувача, та серверного API, який виконує бізнес-логіку та обробляє запити.

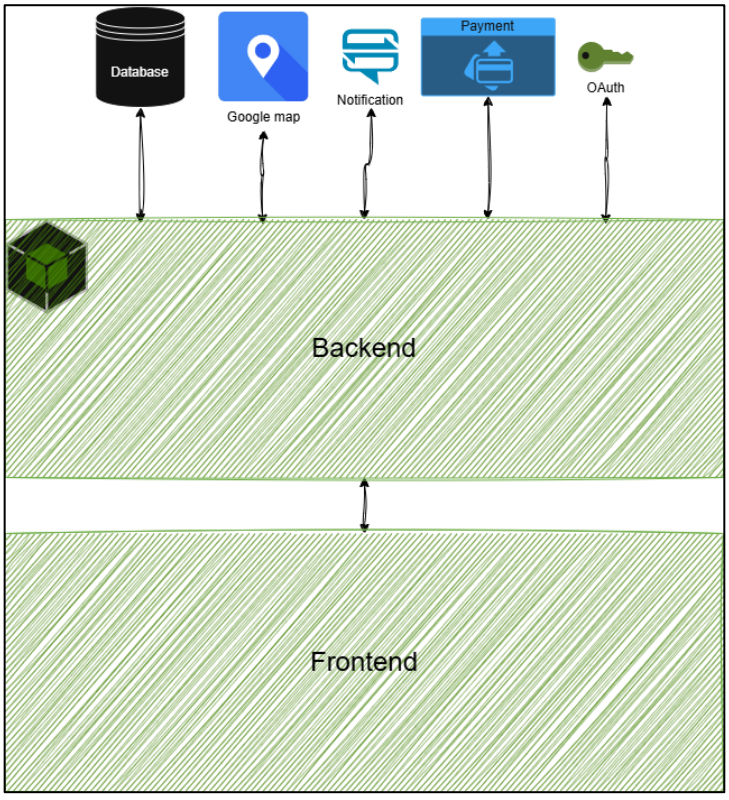


Рисунок 2.1 – Архітектурна модель вебдодатку

Центральним елементом зберігання інформації про замовлення, маршрути та користувачів виступає база даних, тоді як безпека доступу та розмежування прав гарантуються спеціалізованою підсистемою аутентифікації. Функціональність платформи додатково розширюється завдяки інтеграції із зовнішніми сервісами геолокації, оплати та сповіщень, при цьому обмін даними між усіма компонентами здійснюється через захищені HTTPS-запити, переважно у форматі JSON.

2.2.2 Клієнтська частина системи

Клієнтська частина системи побудована на базі фреймворку Nuxt.js, що дозволяє ефективно поєднувати переваги серверного рендерингу (SSR) та гнучкість односторінкових додатків (SPA). Архітектура додатку організована навколо файлової структури сторінок (pages), які автоматично генерують маршрути, та використовує спеціалізовані шаблони (layouts) для розмежування структури інтерфейсу, наприклад, між публічною частиною та адміністративною панеллю.

Візуальна складова формується з багаторазово використовуваних компонентів, таких як форми чи карти, а централізоване управління станом та даними забезпечується сховищем на базі Pinia. Для взаємодії з бекендом виділено окремі сервісні модулі, тоді як контроль доступу та обробку запитів перед рендерингом здійснює проміжне програмне забезпечення (middleware). Додатково система включає статичні ресурси та плагіни, що розширюють базовий функціонал Vue.js і Nuxt.js, наприклад, для валідації даних чи інтеграції картографічних сервісів. Nuxt.js дозволяє використовувати різні режими рендерингу для різних частин додатку:

– SSR (принципову схему наведено на рисунку 2.2): для публічних сторінок, що потребують SEO-оптимізації (відстеження замовлень, публічні сторінки);

– CSR (принципову схему наведено на рисунку 2.3): для адміністративних панелей та частин додатку, що вимагають багатої інтерактивності;
– SSG (принципову схему наведено на рисунку 2.4): для статичних сторінок, таких як документація, FAQ, умови використання.

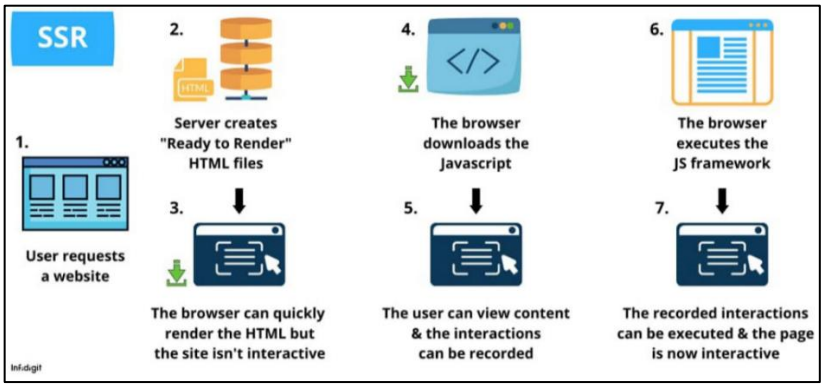


Рисунок 2.2 – Принципова схема SSR [8]

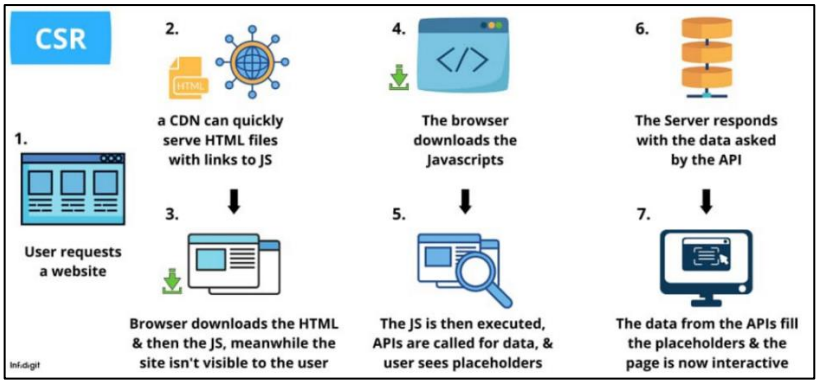


Рисунок 2.3 – Принципова схема CSR [8]

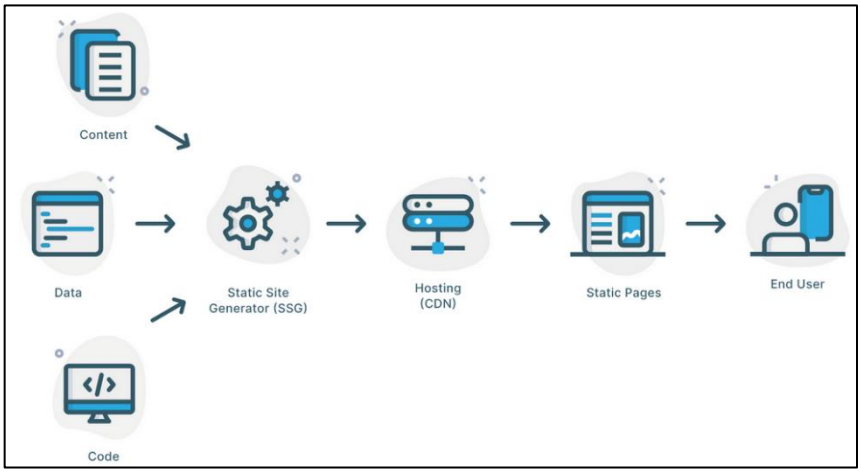


Рисунок 2.4 – Принципова схема SSG [8]

Маршрутизація в Nuxt.js базується на файловій структурі каталогу pages/, що значно спрощує організацію маршрутів. Nuxt.js автоматично генерує маршрути на основі структури файлів та директорій.

Додатково використовуються:

– динамічні маршрути (для сторінок з параметрами, наприклад `/orders/:id`);
– вкладені маршрути (для складної ієрархії сторінок);
– middleware для захисту маршрутів (перевірка авторизації).

Серверна частина системи реалізована як RESTful API, що надає клієнтській частині необхідні дані та функціональність.

2.2.3 Користувацький інтерфейс та інтерактивна візуалізація

Користувацький інтерфейс системи спроєктовано з урахуванням специфічних потреб кожної ролі, забезпечуючи зручність та ефективність на всіх етапах логістичного ланцюжка.

Клієнтська частина представлена мінімалістичним дашбордом, що дозволяє швидко відстежувати рух кур’єра за допомогою карти. Для кур’єрів розроблено мобільний інтерфейс із великими контрастними елементами управління, дозволяючи працювати без зайвих маніпуляцій навіть у складних дорожніх умовах.

Адміністративна панель виступає потужним аналітичним центром, де на інтерактивних картах відображається глобальна картина всіх процесів, доступні інструменти автоматичної оптимізації маршрутів та детальні звіти для прийняття управлінських рішень.

Весь інтерфейс реалізовано з використанням адаптивного дизайну, що гарантує коректне відображення як у спрощеному вертикальному режимі на смартфонах, так і в багатоколонковому форматі на десктопах.

2.2.4 Архітектура серверної частини

Архітектура серверної частини спроєктована за багатошаровим принципом (Layered Architecture), що забезпечує гнучкість, модульність та чіткий розподіл зон відповідальності між компонентами. Обробка даних відбувається послідовно: API-контролери приймають та валідують HTTP-запити, передаючи їх у сервісний шар для виконання бізнес-логіки, після чого репозиторії здійснюють операції з реляційною базою даних, яка підтримує геопросторові об’єкти.

Технічний стек реалізовано на платформі Node.js із використанням ORM Prisma для роботи з даними, Socket.io для комунікації в реальному часі, а також допоміжних утиліт і проміжного програмного забезпечення (middleware). Функціональність системи значно розширена завдяки інтеграції із зовнішніми сервісами: геолокаційними API (OpenStreetMap) для побудови маршрутів і трекінгу, аналітичними інструментами та мультиканальною системою сповіщень (Email, Push).

Захист забезпечується надійними протоколами безпеки, включаючи аутентифікацію через JWT, двофакторну перевірку, а також систему авторизації на основі ролей (RBAC), що гарантує контроль доступу до ресурсів на всіх рівнях.

2.2.5 Схема взаємодії компонентів

Взаємодія компонентів системи, схематично зображена на рисунку 2.5, реалізується через чіткий алгоритм обміну даними між клієнтською та серверною частинами.

Процес ініціюється діями користувача в інтерфейсі, які трансформуються в API-запити до сервера, де проходять перевірку аутентифікації та авторизації через проміжне програмне забезпечення.

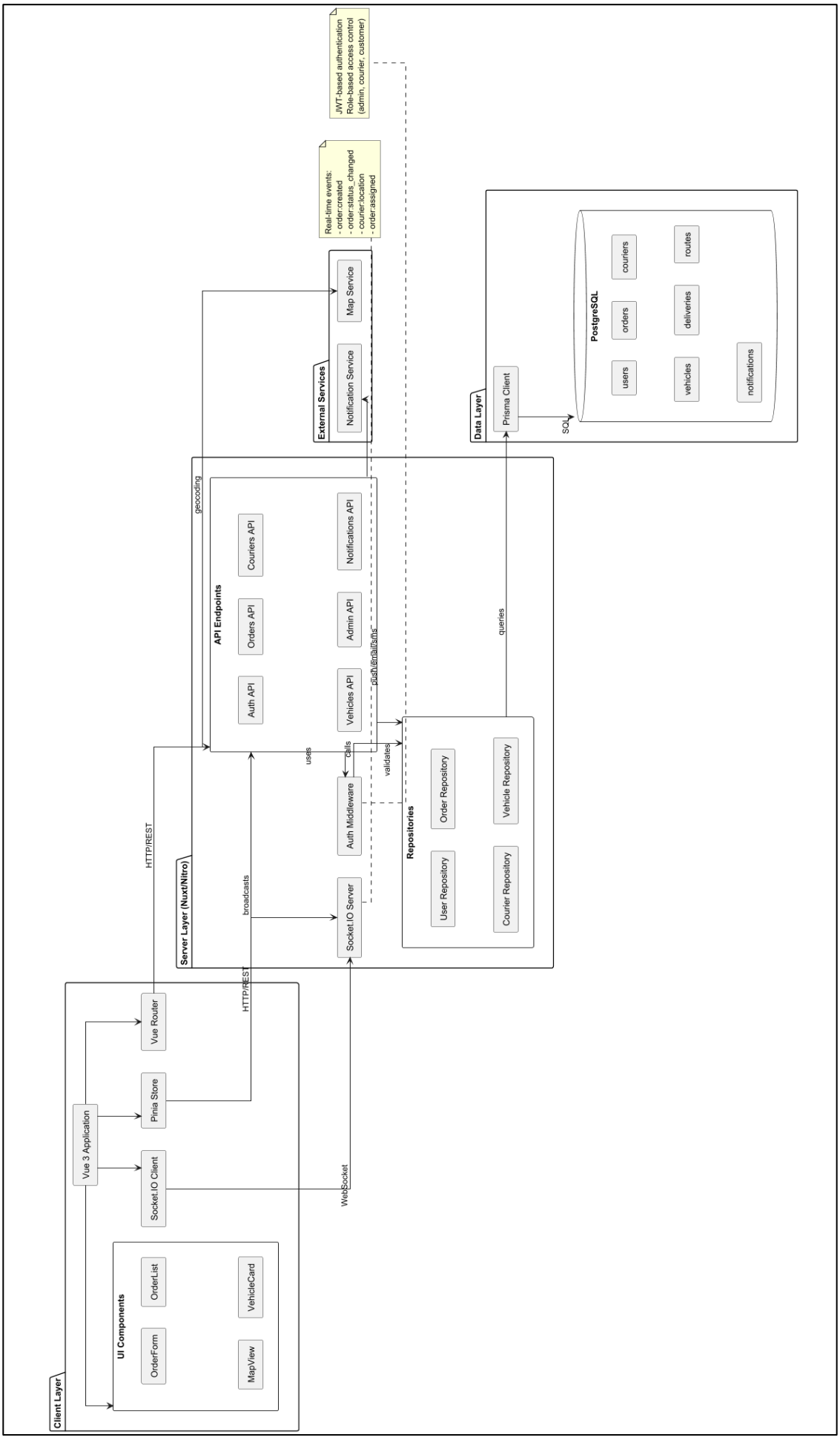


Рисунок 2.5 – Діаграма компонентів

Після верифікації контролери делегують виконання бізнес-логіки сервісам, котрі через репозиторії здійснюють необхідні транзакції з базою даних, а сформована відповідь повертається клієнту для візуалізації результатів. Окремий механізм передбачено для динамічних даних, таких як статус доставки чи геолокація кур'єра: завдяки технології WebSocket сервер автоматично надсилає оновлення в реальному часі, забезпечуючи актуалізацію інтерфейсу без необхідності перезавантаження сторінки.

2.2.6 Масштабування системи

Архітектура системи, схему розгортання якої наведено на рисунку 2.6, спроєктована з урахуванням гнучкого масштабування, що дозволяє адаптувати обчислювальні потужності до змін навантаження. Горизонтальне розширення досягається шляхом розгортання кластера API-серверів під управлінням балансувальника навантаження, використання механізмів шардування бази даних та інтеграції Redis для ефективного кешування, що суттєво знижує навантаження на основне сховище. Водночас вертикальне масштабування передбачає комплексний підхід: від нарощування апаратних ресурсів серверів до програмної оптимізації, яка включає вдосконалення алгоритмів обробки даних та SQL-запитів. Стратегія подальшого розвитку системи орієнтована на перехід до мікросервісної архітектури, де незалежні компоненти, розгорнуті в контейнерах Docker під управлінням Kubernetes, взаємодіятимуть асинхронно через черги повідомлень, такі як RabbitMQ або Kafka, забезпечуючи максимальну гнучкість та відмовостійкість.

Описана архітектурна модель забезпечує надійну основу для розробки масштабованого, продуктивного та безпечного вебдодатку для управління доставкою. Використання Nuxt.js для клієнтської частини дозволяє поєднати переваги серверного та клієнтського рендерингу, забезпечуючи відмінний користувацький досвід та оптимальні умови для розробки та підтримки системи.

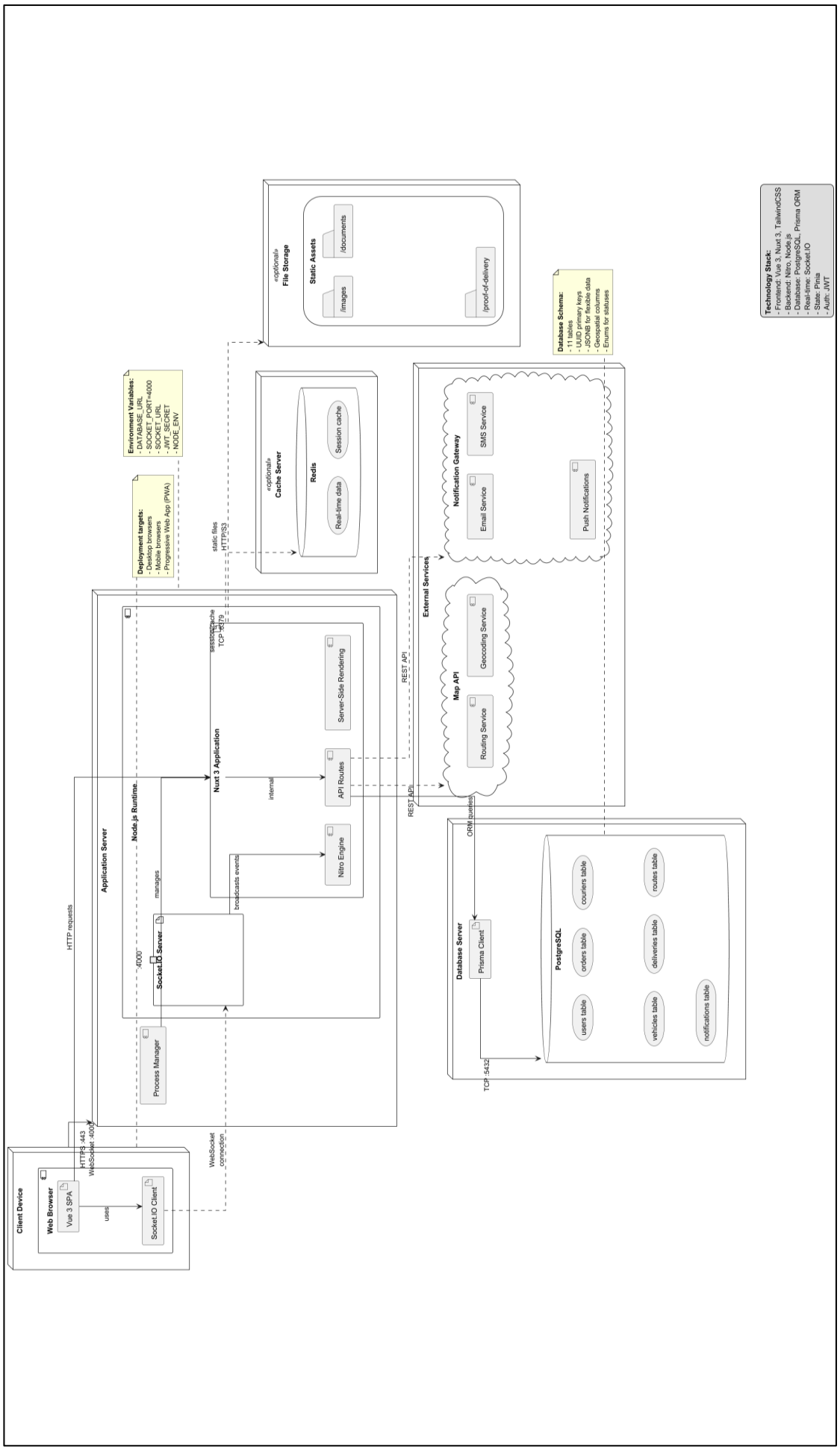


Рисунок 2.6 – Діаграма розгортання

2.3 Прототипування та тестування

Розробка UI/UX-концепції системи, етапи якої відображено на діаграмі активностей (рис. 2.7 та 2.8), базується на послідовному ітеративному підході: від створення низькорівневих схем (wireframes) для затвердження структури та потоків взаємодії до реалізації інтерактивних і високорівневих прототипів із повним візуальним оформленням.

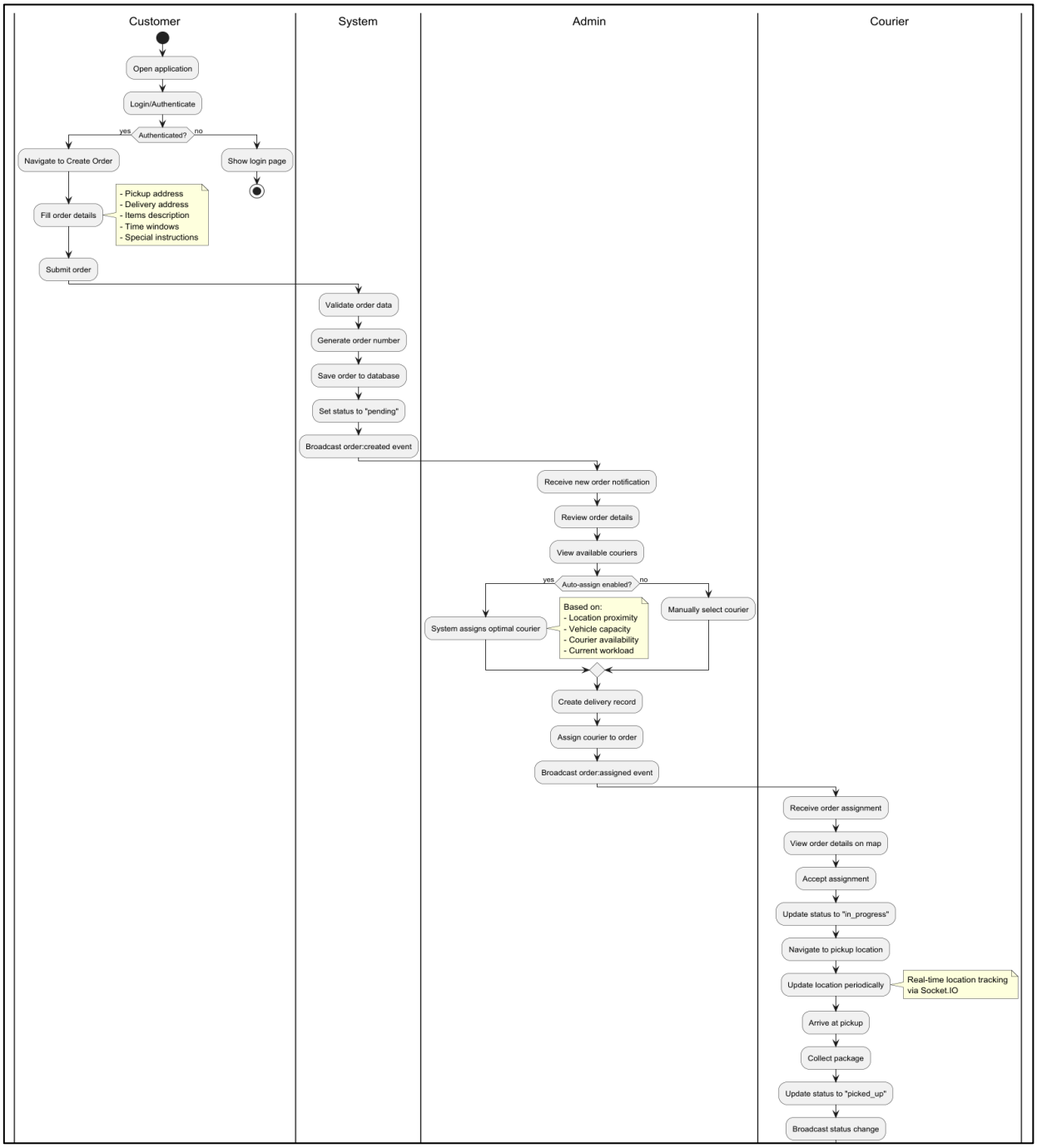


Рисунок 2.7 – Діаграма активностей (1 частина)

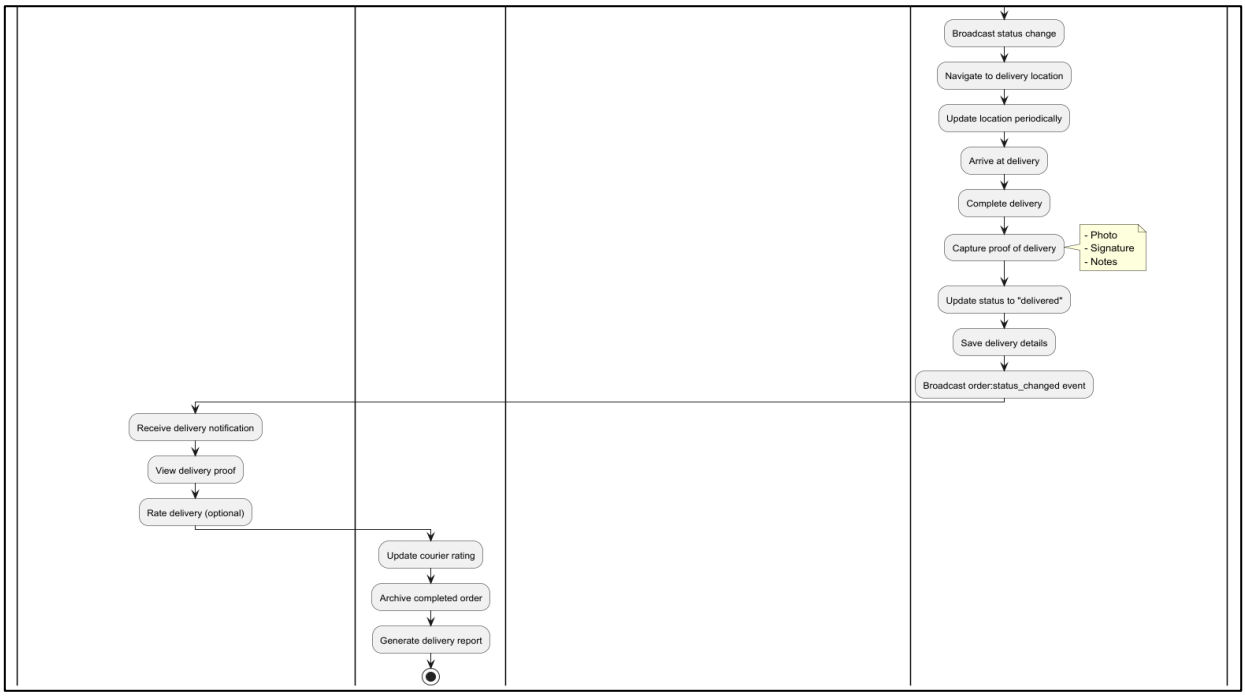


Рисунок 2.8 – Діаграма активностей (2 частина)

Якість рішень забезпечується комплексною стратегією тестування, що включає перевірку сценаріїв на реальних користувачах, A/B-експерименти, аналіз теплових карт та оцінку доступності інтерфейсу. Для підвищення ергономічності та емоційного відгуку в систему інтегровано функціональні анімації.

2.4 Схема бази даних та структура об’єктів доставки

Проєктування ефективної схеми бази даних є ключовим етапом розробки системи управління доставкою, оскільки саме від структури даних залежить надійність зберігання інформації, швидкість обробки запитів та можливість масштабування системи в майбутньому.

У рамках проєкту для реалізації збереження інформації про замовлення, маршрути та клієнтів було обрано систему управління базами даних PostgreSQL. Основними причинами вибору цієї СУБД стали:

– підтримка складних транзакцій, необхідних для забезпечення цілісності даних під час оновлення замовлень та маршрутів;

– можливість використання як класичних реляційних моделей, так і зберігання структурованих даних у форматі JSON;
– підтримка геопросторових типів даних, що є важливою функцією для логістичних систем;
– висока надійність та масштабованість у рамках сучасних вебзастосунків.

Таким чином, PostgreSQL повністю відповідає функціональним вимогам системи та забезпечує стабільну основу для подальшого розвитку додатку.

2.4.1 Концептуальна модель даних

Концептуальна модель даних визначає основні сутності системи та зв’язки між ними. Для системи управління доставкою виділено наступні основні сутності:

– користувачі (Users) – особи, що мають доступ до системи різних рівнів;
– ролі (Roles) – типи користувачів з різними правами доступу;
– клієнти (Customers) – замовники послуг доставки;
– замовлення (Orders) – запити на доставку від клієнтів;
– доставки (Deliveries) – процеси виконання замовлень;
– кур’єри (Couriers) – виконавці доставок;
– транспортні засоби (Vehicles) – засоби для здійснення доставки;
– маршрути (Routes) – шляхи доставки;
– точки маршруту (RoutePoints) – проміжні точки на маршруті;
– статуси (Statuses) – можливі стани замовлень та доставок;
– сповіщення (Notifications) – повідомлення користувачам системи;
– журнали (Logs) – записи про дії користувачів та системні події.

ER діаграму бази даних наведено на рисунку 2.9 та розміщено за посиланням: https://liambx.com/erd/p/github.com/bloodsbro/delivery-app/blob/master/prisma/schema.prisma?showMode=ALL_FIELDS.

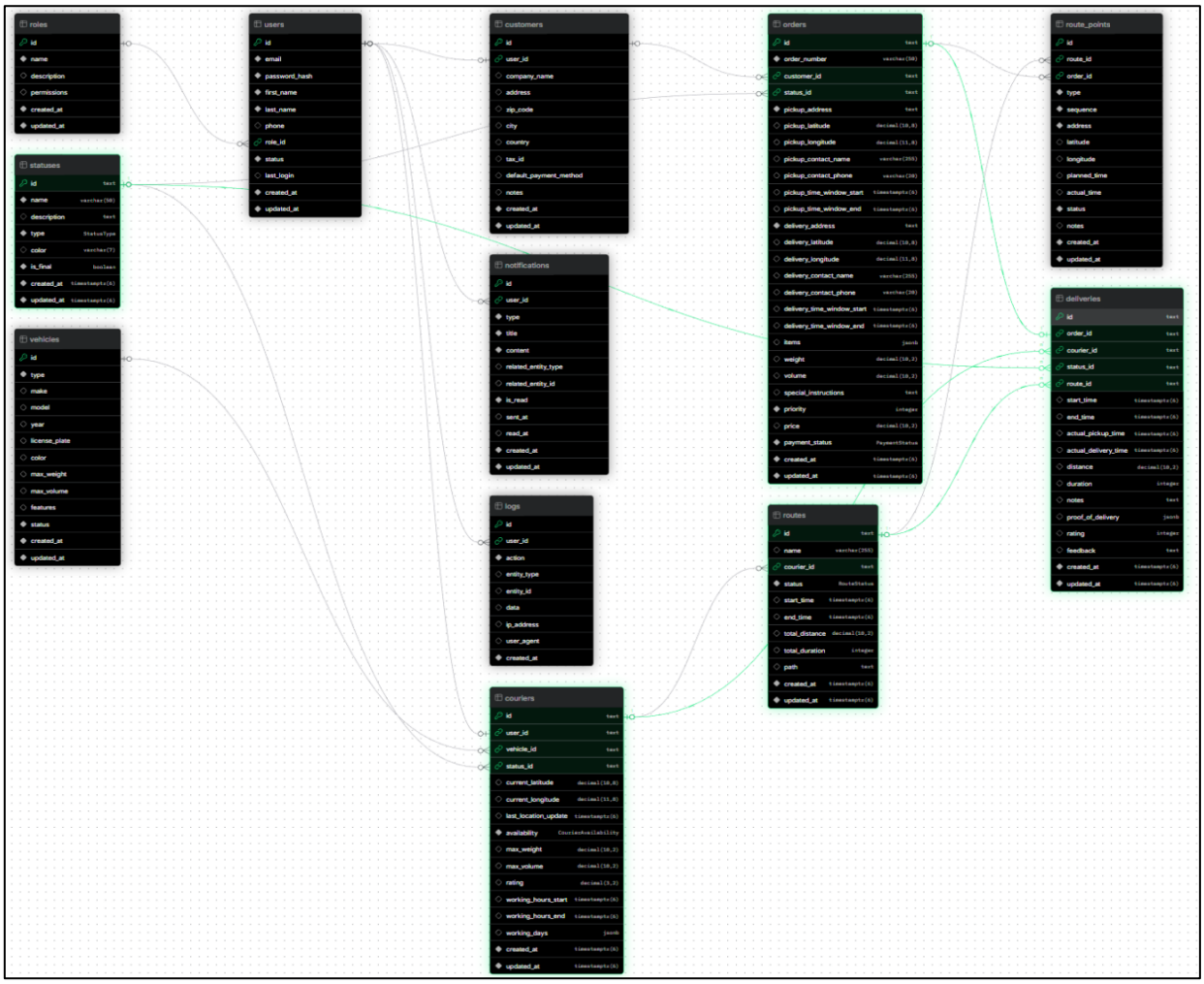


Рисунок 2.9 – ER-діаграма бази даних

2.4.2 Логічна модель бази даних, індекси та оптимізація

На основі концептуальної моделі розроблена логічна модель бази даних, яка включає таблиці, їх атрибути та зв’язки між ними та наведена у додатку А.

Для оптимізації продуктивності бази даних та забезпечення швидкого доступу до даних створено індекси, які наведено у додатку Б.

2.4.3 Структура об’єктів доставки

Об’єкти доставки в системі представлені як складні структури даних, що об’єднують інформацію з різних таблиць (див. рис. 2.10).

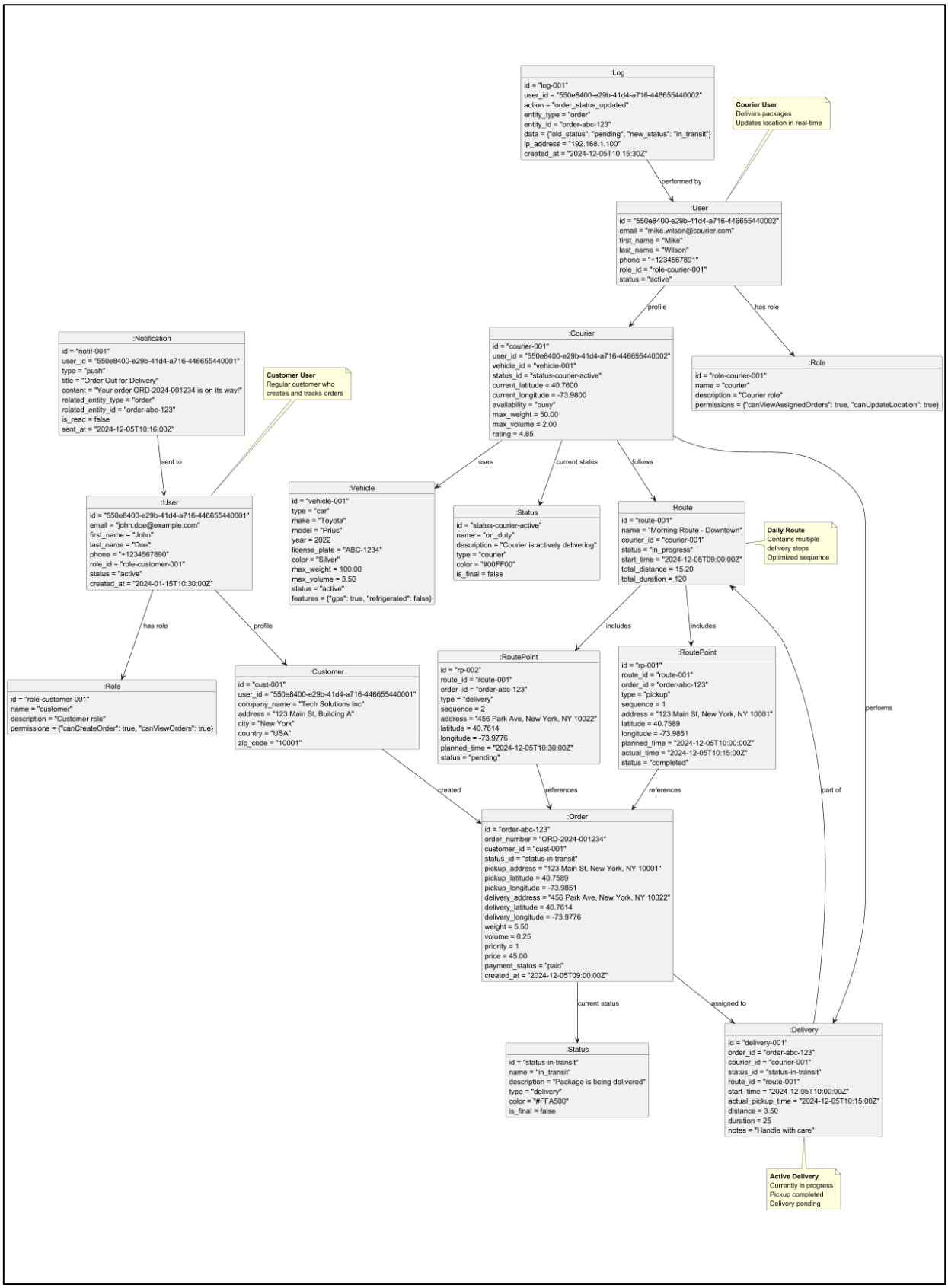


Рисунок 2.10 – Діаграма об’єктів

Центральним елементом архітектури є об’єкт *Замовлення (Order)*, який акумулює вичерпну інформацію про логістичний запит: від ідентифікаторів та

контактних даних клієнта до специфічних параметрів вантажу, адресних точок забору й призначення, допустимих часових вікон та фінансових деталей.

Безпосередній процес виконання фіксується в об’єкті *Доставка (Delivery)*, що пов’язує замовлення з виконавцем, зберігає хронологію подій (час початку, вручення), фактичний трек переміщення та результати роботи, включаючи докази доставки й відгуки.

Логістична складова моделюється через *Маршрут (Route)*, який визначає геометрію шляху, загальні метрики відстані й тривалості та складається з упорядкованої послідовності *Точок маршруту (RoutePoint)* – конкретних локацій із плановими та фактичними часовими мітками.

Виконавцем операцій виступає *Кур’єр (Courier)*, профіль якого містить персональні дані та динамічну інформацію про поточне місцезнаходження і статус доступності в системі.

2.4.4 Взаємодія між об’єктами

Життєвий цикл обробки замовлення, відображений на діаграмі взаємодії (рис. 2.11), розпочинається з ініціації заявки клієнтом, після чого система автоматично реєструє її зі статусом «Нове» та додає до черги на опрацювання.

На етапі логістичного планування відбувається закріплення замовлення за конкретним кур’єром, що супроводжується генерацією об’єкта доставки, побудовою маршруту через точки забору й вручення та зміною статусу на «Призначено».

Процес фізичного виконання передбачає послідовну фіксацію кур’єром етапів забору вантажу та його передачі одержувачу, під час чого статус динамічно змінюється на «В процесі доставки» і врешті «Доставлено».

Завершується цикл моніторингом та аналітикою, де система агрегує дані про виконані операції для розрахунку KPI кур’єрів та подальшої оптимізації маршрутних схем.

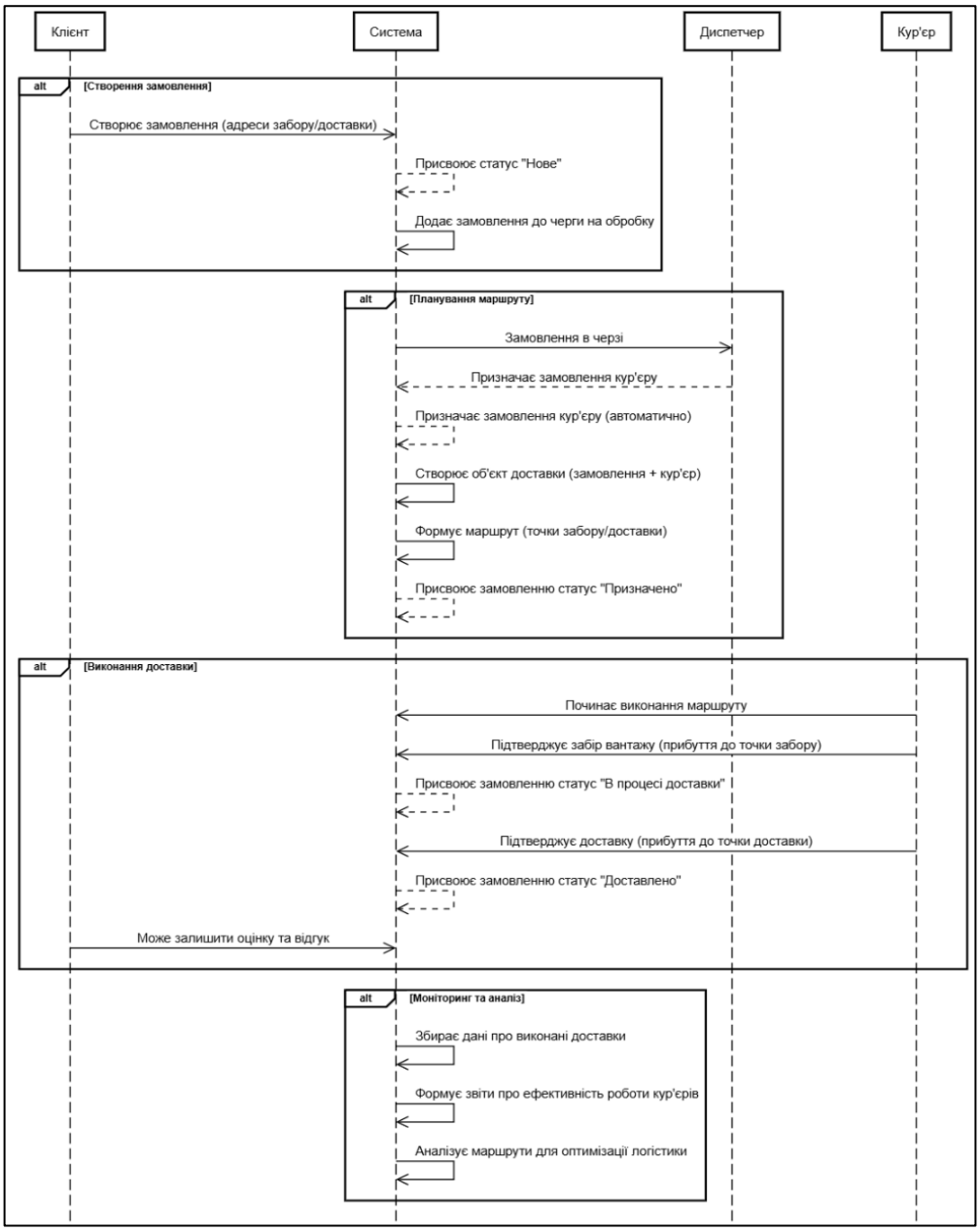


Рисунок 2.11 – Діаграма взаємодії

2.4.5 Сценарії використання бази даних

Основні сценарії використання бази даних в системі управління доставкою наведено у додатку В.

3 РЕАЛІЗАЦІЯ ВЕБДОДАТКУ ЗАСОБАМИ NUXT.JS

3.1 Ініціалізація проєкту: конфігурація Nuxt-додатку

Ініціалізація та правильна конфігурація проєкту на Nuxt.js є важливим етапом, який закладає основу для подальшої розробки вебдодатку для управління доставкою. У цьому підрозділі розглянуто процес створення проєкту, налаштування базової конфігурації та підготовку середовища розробки.

Ініціалізацію нового проєкту реалізовано за допомогою офіційного інструментарію, що дозволило автоматично налаштувати базову архітектуру на основі стабільної версії Nuxt 3 з використанням пакетного менеджера npm. Конфігурація розробки передбачає застосування мови TypeScript та універсального режиму рендерингу (SSR), а візуальне оформлення базується на гнучкому CSS-фреймворку Tailwind CSS. Для розширення функціональних можливостей до системи інтегровано низку спеціалізованих модулів: @pinia/nuxt для централізованого управління станом, @nuxt/content для роботи зі статичними матеріалами та документацією, а також @nuxtjs/i18n для підтримки багатомовності. Забезпечення якості коду та контроль за цілісністю проєкту здійснюються через налаштування лінтерів ESLint і Prettier, середовища тестування Jest та системи контролю версій Git.

До проєкту було додано додаткові залежності, які будуть використовуватися в проєкті для системи управління доставкою (рис. 3.1).

```
        cd delivery-management-app
        npm install @nuxtjs/auth-next mapbox-gl @vueuse/core chart.js vue-chartjs
date-fns lodash-es
```

Рисунок 3.1 – Встановлення залежностей

Базова структура та конфігурація проєкту наведена на рисунку 3.2.

```
       ├── .nuxt/                 # Автоматично генерований код (git-ignored)
       ├── assets/                # Статичні ресурси, які потребують обробки (SCSS,
шрифти)
       ├── components/            # Vue-компоненти
       ├── composables/              # Повторно використовувані функції з поточним
контекстом
       ├── content/               # Markdown та JSON файли для nuxt/content
       ├── layouts/               # Шаблони сторінок
       ├── middleware/            # Проміжне програмне забезпечення для маршрутів
       ├── pages/                # Сторінки додатку (автоматично стають маршрутами)
       ├── plugins/               # Плагіни Vue.js та Nuxt.js
       ├── public/               # Статичні файли, доступні за URL (фавікон, роботи)
       ├── server/                # Серверні роути API та middleware
       ├── stores/                # Pinia stores для управління станом
       ├── utils/                 # Утиліти та допоміжні функції
       ├── .eslintrc.js           # Конфігурація ESLint
       ├── .gitignore             # Git ignore файл
       ├── .prettierrc            # Конфігурація Prettier
       ├── app.vue                # Головний компонент додатку
       ├── nuxt.config.ts         # Конфігурація Nuxt.js
      ├── package.json           # Залежності і скрипти
      ├── tailwind.config.js     # Конфігурація Tailwind CSS
      └── tsconfig.json          # Конфігурація TypeScript
```

Рисунок 3.2 – Структура проєкту

Після ініціалізації критично важливим етапом є налаштування файлів конфігурації, які визначають поведінку додатку, підключені модулі та інструменти збірки.

Налаштування nuxt.config.ts. Файл **nuxt.config.ts** є головним файлом конфігурації. У розроблюваному додатку він налаштований для використання Tailwind CSS v4, Prisma ORM, Nuxt UI та інших допоміжних модулів. Вміст файлу наведено на рисунку 3.3.

Управління залежностями (package.json). Файл **package.json** фіксує список усіх бібліотек та скриптів автоматизації. На рисунку 3.4 наведено повний перелік залежностей, що використовуються для реалізації функціоналу системи доставки.

```
import tailwindcss from "@tailwindcss/vite";

// https://nuxt.com/docs/api/configuration/nuxt-config
export default defineNuxtConfig({
  compatibilityDate: '2025-05-15',
  devtools: { enabled: true },

  modules: [
    '@nuxt/eslint',
    '@nuxt/fonts',
    '@nuxt/image',
    '@nuxt/test-utils',
    '@nuxt/ui',
    '@prisma/nuxt',
    '@pinia/nuxt',
  ],

  experimental: {
    componentIslands: true,
  },

  css: [
    // 'vue-good-table-next/dist/vue-good-table-next.css',
    '~/assets/css/main.css',
  ],

  vite: {
    plugins: [
      tailwindcss(),
    ],
  },
})
```

Рисунок 3.3 – Конфігурація nuxt.config.ts

```
{
  "name": "nuxt-app",
  "private": true,
  "type": "module",
  "scripts": {
    "build": "nuxt build",
```

```
    "dev": "nuxt dev",
    "generate": "nuxt generate",
    "preview": "nuxt preview",
    "postinstall": "nuxt prepare",
      "prisma:erd": "npx @liam-hq/cli erd build --format prisma --input
prisma/schema.prisma"
  },
  "dependencies": {
    "@headlessui/vue": "^1.7.23",
    "@nuxt/eslint": "^1.4.1",
    "@nuxt/fonts": "^0.11.4",
    "@nuxt/image": "^1.10.0",
    "@nuxt/test-utils": "^3.19.1",
    "@nuxt/ui": "^3.1.3",
    "@pinia/nuxt": "^0.11.1",
    "@prisma/client": "6",
    "@prisma/extension-accelerate": "^2.0.1",
    "@prisma/nuxt": "^0.3.0",
    "@tailwindcss/vite": "^4.1.8",
    "@types/bcrypt": "^6.0.0",
    "bcrypt": "^6.0.0",
    "eslint": "^9.28.0",
    "nuxt": "^3.17.5",
    "pinia": "^3.0.3",
    "prisma": "6",
    "ts-node": "^10.9.2",
    "typescript": "^5.8.3",
    "uuid": "^13.0.0",
    "vue": "^3.5.16",
    "vue-router": "^4.5.1"
  },
  "devDependencies": {
    "autoprefixer": "^10.4.21",
    "postcss": "^8.5.4",
    "tailwindcss": "^4.1.8"
  },
  "prisma": {
    "seed": "bun prisma/seed.ts"
  }
}
```

Рисунок 3.4 – Залежності проєкту (package.json)

Аналіз основних залежностей:

– Nuxt, vue, vue-router – ядро додатку;
– UI/UX: @nuxt/ui та @headlessui/vue забезпечують доступні та стильні компоненти інтерфейсу;
– Database: prisma (версія 6) та @prisma/client використовуються як ORM для роботи з базою даних (додано скрипт prisma:erd для генерації діаграми зв’язків сутностей (ERD));
– Security: bcrypt використовується для хешування паролів користувачів;
– Development: налаштовано eslint для перевірки коду та seed скрипт для наповнення бази даних тестовими даними.

Ця конфігурація створює надійний фундамент для масштабованої системи управління доставкою.

3.2 Реалізація серверної логіки та взаємодія з базою даних

Оскільки проєкт використовує стек Nuxt.js + Node.js + Prisma, шар роботи з даними є фундаментом всієї системи. Серверна частина реалізована безпосередньо в Nuxt (через рушій Nitro), а взаємодія з базою даних PostgreSQL відбувається через ORM Prisma. Це забезпечує типізацію даних на всіх рівнях (end-to-end type safety) та спрощує управління складними зв’язками між сутностями доставки.

Проєктування схеми бази даних (prisma/schema.prisma). Схема бази даних визначає структуру даних, типи полів та зв’язки між таблицями. У файлі prisma/schema.prisma описана реляційна модель, яка покриває всі аспекти логістики: від користувачів до маршрутів доставки.

Основні групи моделей. Система управління доступом базується на центральній сутності User, яка зберігає облікові дані з хешованими паролями та статусом активності, і пов’язана з моделлю Role, що визначає права

доступу. Для забезпечення гнучкості та розділення відповідальності логіка аутентифікації відокремлена від бізнес-сутностей: профілі Customer та Courier мають зв’язок «один-до-одного» з користувачем, де кур’єрський профіль додатково містить робочі параметри, такі як рейтинг, геолокація та закріплений транспорт.

Ядром бізнес-логіки виступає модель Order, що агрегує інформацію про адреси, часові вікна, параметри вантажу та перелік товарів, тоді як життєвий цикл замовлення керується через окрему таблицю Status для різних контекстів (замовлення, доставка, кур’єр). Логістична складова реалізована через взаємодію моделей Vehicle для обліку транспортних засобів та Delivery для фіксації процесу виконання, причому архітектура підтримує побудову складних маршрутів завдяки сутностям Route та RoutePoint, що дозволяє оптимізувати логістику шляхом об’єднання кількох точок забору та доставки в один рейс.

На рисунку 3.5 наведено концептуальну схему зв’язків (ERD).

Автоматичне наповнення даними (prisma/seed.ts). Для автоматизованого розгортання та налаштування середовища розробки реалізовано скрипт prisma/seed.ts, який забезпечує наповнення бази даних початковою інформацією, використовуючи бібліотеку bcrypt для безпечного хешування паролів. Процес ініціалізації, або сідінгу, розпочинається зі створення фундаментальних системних довідників, що включають фіксовані ролі (адміністратор, кур’єр, клієнт) та повний набір статусів життєвого циклу замовлень. На наступному етапі генеруються тестові облікові записи з уніфікованими даними для зручності відладки, а також формуються ключові бізнес-сутності, такі як профілі клієнтів і транспортні засоби. Завершується процедура моделюванням реалістичних робочих сценаріїв шляхом створення тестових замовлень, які прив’язані до конкретних користувачів і містять варіативні статуси та географічні координати.

Приклад фрагмента коду, що створює замовлення з прив’язкою до клієнта та статусу наведено на рисунку 3.6.

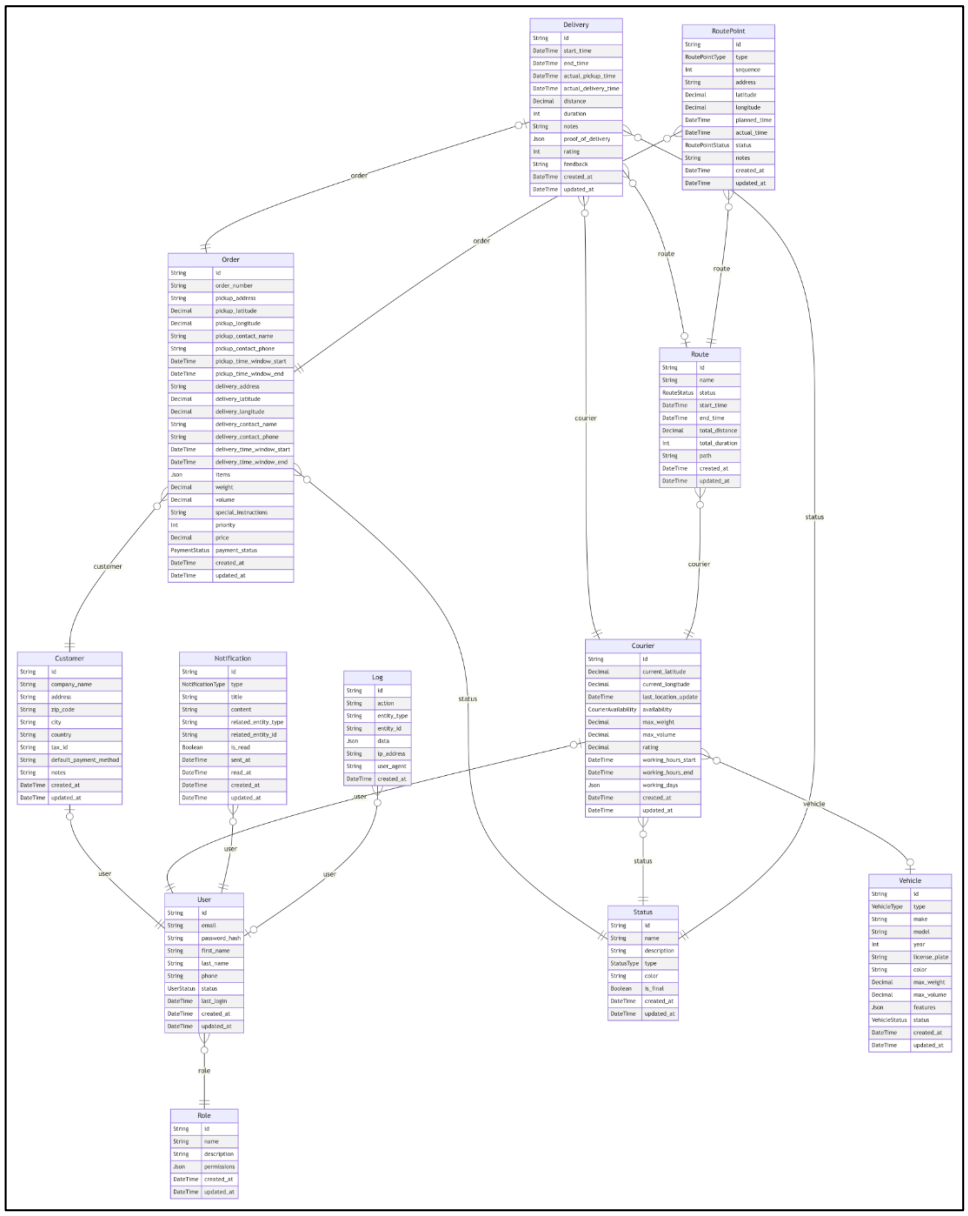


Рисунок 3.5 – ERD-схема бази даних

```
    await prisma.order.upsert({
      where: { order_number: 'ORD-2024-001' },
      update: {},
      create: {
        order_number: 'ORD-2024-001',
        customer_id: customer1.id,
        status_id: 'status-pending',
        pickup_address: 'вул. Хрещатик, 1, Київ',
        pickup_latitude: 50.4501,
        pickup_longitude: 30.5234,
        // ...інші поля
        items: JSON.stringify([
           { name: 'Документи', quantity: 1, weight: 0.1 }
        ]),
        payment_status: 'pending'
      }
    });
```

Рисунок 3.6

Застосування змін. Щоб активувати описану структуру бази даних та наповнити її, у проєкті використовуються наступні команди (визначені у package.json).

Міграція схеми: створення таблиць у PostgreSQL. Команду наведено на рисунку 3.7.

```
npx prisma migrate dev --name init
```

Рисунок 3.7 – Створення міграції

Генерація клієнта: оновлення node_modules для типізованого доступу. Команду наведено на рисунку 3.8.

```
npx prisma generate
```

Рисунок 3.8 – Генерація типів

Наповнення даними: запуск скрипта seed.ts. Команду наведено на рисунку 3.9.

```
bun prisma/seed.ts  # або npx prisma db seed
```

Рисунок 3.9 – Виконання скрипту сидінгу

3.2.1 Інтеграція ORM Prisma: налаштування клієнта, генерація типів та Repository Pattern

Відповідно до архітектури серверної частини, визначеної в п. 2.2.4, взаємодія з базою даних ізолюється в шарі репозиторіїв (Repositories). Це дозволяє відокремити бізнес-логіку (Services) від деталей реалізації запитів до БД. Використання Prisma ORM у поєднанні з TypeScript забезпечує сувору типізацію на цьому рівні.

Налаштування клієнта Prisma (Singleton Pattern). Для роботи з базою даних необхідно ініціалізувати екземпляр PrismaClient. У середовищі Node.js, особливо при використанні механізмів «гарячого перезавантаження» (HMR) під час розробки в Nuxt, важливо використовувати паттерн Singleton. Це запобігає вичерпанню ліміту підключень до бази даних через створення численних екземплярів клієнта при кожній зміні коду.

Створений файл конфігурації клієнта PrismaORM наведено на рисунку 3.10.

```
import { PrismaClient } from '@prisma/client'

const prismaClientSingleton = () => {
  return new PrismaClient()
}

declare const globalThis: {
  prismaGlobal: ReturnType<typeof prismaClientSingleton>;
} & typeof global;

const prisma = globalThis.prismaGlobal ?? prismaClientSingleton()

export default prisma

if (process.env.NODE_ENV !== 'production') globalThis.prismaGlobal = prisma
```

Рисунок 3.10 – Файл конфігурації клієнта PrismaORM

Такий підхід гарантує, що весь додаток використовує одне з’єднання з базою даних, а логування запитів у режимі розробки допомагає відстежувати SQL-запити.

Генерація типів. Однією з головних переваг Prisma є автоматична генерація TypeScript-типів на основі файлу **schema.prisma**. Це дозволяє уникнути розбіжностей між структурою БД та кодом. Команду генерації типів наведено на рисунку 3.11.

Ця команда створює типи в node_modules/.prisma/client, які автоматично підхоплюються IDE. Наприклад, це дозволяє отримати готові типи User, Order, Delivery, а також типи для створення (Prisma.UserCreateInput) та оновлення даних.

```
npx prisma generate
```

Рисунок 3.11 – Команда для генерації типів PrismaORM

Реалізація паттерну Repository. Логіка роботи із замовленнями включає створення записів з глибокою вкладеністю (клієнт, статус, доставка), перевірку наявності статусів та оновлення даних. Функції реалізовані в модульному стилі.

Управління клієнтами (Customers). Цей модуль реалізує логіку «знайти або створити» (Find or Create), яка автоматично створює пов'язані сутності User та Role перед створенням запису Customer.

Управління транспортом та кур'єрами. Функції для адміністративної панелі (CRUD для транспорту) та логіка призначення замовлення на кур'єра, яка включає роботу з таблицею Delivery та оновлення статусів.

Така структура гарантує дотримання принципів чистої архітектури завдяки чіткому розподілу обов'язків у ланцюжку обробки даних. Процес ініціюється на рівні API Handler, який приймає вхідний HTTP-запит і передає керування сервісному шару, де зосереджено валідацію даних та виконання основної бізнес-логіки. Наступним етапом є взаємодія з базою даних через репозиторій, який виконує необхідні запити за допомогою ORM Prisma, забезпечуючи у підсумку повернення суворо типізованого результату.

3.2.2 Розробка REST API

Архітектура маршрутів: кожен ендпоінт – файл у server/api/** , який експортує defineEventHandler і працює з event (HTTP-запит/відповідь) та повертає JSON. Доступ до БД ізольовано у репозиторіях server/repositories/** .

Валідація вхідних даних: для валідації вхідних даних використовується модуль Joi:

– логін: перевірка наявності полів email та пароля наведено на рисунку 3.12;

– створення замовлення: перевірка наявності обов’язкових полів наведено на рисунку 3.13;
– призначення кур’єра: валідація orderId та courierId наведено на рисунку 3.14.

```
      const schema = Joi.object({
          email: Joi.string().trim().lowercase().email().required(),
          password: Joi.string().min(6).required(),
        })
        const { error } = schema.validate(body)
      —       if (error) throw createError({ statusCode: 400, statusMessage:
'Invalid credentials' })
```

Рисунок 3.12 – Перевірка наявності полів email та пароля

```
            const schema = Joi.object({
              customerName: Joi.string().trim().min(2).required(),
              customerAddress: Joi.string().trim().min(3).required(),
              customerPhone:
            Joi.string().trim().pattern(/^[+]?\d{10,15}$/).required(),
              items: Joi.array().items(Joi.object({
              name: Joi.string().trim().min(1).required(),
              quantity: Joi.number().integer().min(1).required(),
              price: Joi.number().min(0).required() })).min(1).required(),
              deliveryLat: Joi.number().optional(),
              deliveryLng: Joi.number().optional(),
            })
            const { error } = schema.validate(body)
            if (error) throw createError({ statusCode: 400, statusMessage: 'Invalid
            input' })
```

Рисунок 3.13 – Перевірка наявності обов’язкових полів

```
       const schema = Joi.object({ orderId: Joi.string().trim().required(),
courierId: Joi.string().trim().required() })
         const { error } = schema.validate(body)
      —       if (error) throw createError({ statusCode: 400, statusMessage:
'Invalid input' })
```

Рисунок 3.14 – Валідація orderId та courierId

Обробка помилок: уніфікована через createError({ statusCode, statusMessage }), наприклад 400 для невірних даних, 401/403 для доступу, 404 для неіснуючих ресурсів.

Приклади контролерів:

– контролер список замовлень наведено на рисунку 3.15;

– оновлення статусу замовлення наведено на рисунку 3.16;

– пошук клієнтів за телефоном наведено на рисунку 3.17.

```
import { toFrontOrder } from '~/server/utils/orders'
import { findMany } from '~/server/repositories/orders'

export default defineEventHandler(async () => {
  const list = await findMany()
  return list.map(toFrontOrder)
})
```

Рисунок 3.15 – Список замовлень

```
if (body.status) await updateStatusByFront(orderId, body.status)

  const updated = await findById(orderId)
  return toFrontOrder(updated)
```

Рисунок 3.16 – Оновлення статусу замовлення

```
export default defineEventHandler(async (event) => {
  const me = await currentUser(event)
  if (!me) throw createError({statusCode: 401, statusMessage: 'Unauthorized'})
  const q = (getQuery(event).q as string)?.trim()
  const schema = Joi.object({ q: Joi.string().min(3).required() })
  const { error } = schema.validate({ q })
  if (error) return []
  return await searchCustomersByPhone(q!)
})
```

Рисунок 3.17 – Пошук клієнтів за телефоном

3.2.3 Сервіси автентифікації

Підсистема безпеки додатку побудована на надійних криптографічних стандартах, де процедура автентифікації використовує алгоритм bcrypt для безпечного порівняння хешів паролів під час входу в систему. Механізм управління сесіями реалізовано через HMAC-підписані кукі-токени, які формуються з додаванням часових міток та унікальних одноразових чисел

(nonce) і захищаються алгоритмом SHA256, причому логіка генерації, валідації підписів та маніпуляцій з кукі інкапсульована в окремих серверних утилітах.

Захист API-маршрутів забезпечується комплексно: базовий рівень перевіряє наявність активної сесії, повертаючи помилку 401 неавторизованим користувачам, тоді як глибший рівень реалізує рольову модель доступу (RBAC) безпосередньо в контролерах, обмежуючи виконання специфічних операцій лише для відповідних ролей, таких як адміністратор або кур’єр.

3.3 Клієнт і стан

Pinia Stores:

– useAuthStore: login/logout/me; заклики до /api/auth/*, зберігання user наведено на рисунку 3.18;
– useOrderStore: список/створення/оновлення статусу наведено на рисунку 3.19;
– useVehicleStore: список/CRUD наведено на рисунку 3.20.

```
       export const useAuthStore = defineStore('auth', {
         state: () => ({ user: null as User | null, loading: false }),
         actions: {
           async me() {
             const { data } = await useFetch<User>('/api/auth/me')
             this.user = data.value || null
           },
           async login(email: string, password: string) {
             const { data } = await useFetch<User>('/api/auth/login', { method: 'POST',
body: { email, password } })
             this.user = data.value || null
           },
           async logout() {
             await useFetch('/api/auth/logout', { method: 'POST' })
             this.user = null
           },
           async adminCreateUser(payload: { email: string; password: string; firstName?:
string; lastName?: string; phone?: string; role?: string }) {
             const { data } = await useFetch('/api/admin/users', { method: 'POST', body:
payload })
             return data.value
           }
         }
       })
```

Рисунок 3.18 – Сховище user store

```
    state: (): OrderState => ({
        orders: [],
      }),
```

Рисунок 3.19 – Сховище order store

```
    export const useVehicleStore = defineStore('vehicles', {
      state: (): VehicleState => ({
        vehicles: [],
      }),
      actions: {
        async fetchVehicles() {
          const { data, error } = await useFetch<Vehicle[]>('/api/vehicles');
          if (data.value) {
            this.vehicles = data.value;
          }
          if (error.value) {
              console.error('Помилка при завантаженні транспортних засобів:',
    error.value);
            throw error.value;
          }
        },
        async createVehicle(payload: Partial<Vehicle>) {
           const { data, error } = await useFetch<Vehicle>('/api/vehicles', {
    method: 'POST', body: payload })
          if (error.value) throw error.value
          if (data.value) this.vehicles.unshift(data.value)
          return data.value
        },
        async updateVehicle(id: string, payload: Partial<Vehicle>) {
          const { data, error } = await useFetch<Vehicle>(`/api/vehicles/${id}`,
    { method: 'PUT', body: payload })
          if (error.value) throw error.value
          const idx = this.vehicles.findIndex(v => v.id === id)
          if (idx !== -1 && data.value) this.vehicles[idx] = data.value
          return data.value
        },
        async deleteVehicle(id: string) {
           const { error } = await useFetch(`/api/vehicles/${id}`, { method:
    'DELETE' })
          if (error.value) throw error.value
          this.vehicles = this.vehicles.filter(v => v.id !== id)
        }
      },
    });
```

Рисунок 3.20 – Сховище vehicle store

Лейаути. Організація клієнтського інтерфейсу базується на використанні універсального шаблону layouts/default.vue, який забезпечує відображення глобальної навігації з адаптивними пунктами меню залежно від ролі користувача, футера та логіки завершення сесії. Відсутність виокремленого лейауту для адміністративної панелі компенсується надійною системою захисту маршрутів, реалізованою через клієнтське проміжне програмне забезпечення (middleware/auth.ts). Цей механізм здійснює централізований контроль доступу, аналізуючи метадані сторінок, визначені через definePageMeta, та автоматично перенаправляє неавторизованих користувачів на сторінку входу або головну, що дозволяє ефективно захищати специфічні розділи для адміністраторів і кур'єрів без ускладнення структури шаблонів.

Огляд інтерфейсу та сторінок. Структура вебдодатку чітко сегментована відповідно до ролей користувачів та бізнес-процесів. Публічна частина представлена головною сторінкою, де реалізовано механізм відстеження посилок за ТТН без необхідності авторизації, тоді як доступ до захищених розділів здійснюється через сторінку входу з автоматичним перенаправленням залежно від ролі.

Адміністративний інтерфейс є найбільш розгалуженим: він включає інструменти для створення замовлень із валідацією та геокодуванням, панель керування статусами й маршрутами, а також окремі модулі для розподілу замовлень між кур'єрами, менеджменту користувачів, перегляду системних логів та управління парком транспортних засобів через інтерактивну карту.

Для кур'єрів передбачено персональний кабінет, оптимізований для роботи зі смартфону, що відображає список призначених доставок, маршрутні полілайни та дозволяє оновлювати статуси виконання.

Усі авторизовані користувачі також мають доступ до налаштувань власного профілю для зміни контактних даних та безпеки.

Ключові модулі та робота з даними. Центральним елементом візуалізації логістики виступає картографічний модуль, побудований на базі

Leaflet та OpenStreetMap. Він інтегрований безпосередньо в компоненти інтерфейсу і забезпечує відображення маркерів, побудову маршрутів методом «найближчого сусіда» та візуалізацію руху транспортних засобів.

Процес управління замовленнями автоматизовано через компонент OrderForm, який поєднує динамічний розрахунок вартості, автозаповнення даних клієнтів та геокодування адрес за допомогою сервісу Nominatim. Повний вміст файлу наведено у додатку Г. Дані про локацію кур'єра зчитуються через Browser Geolocation API та синхронізуються з сервером, забезпечуючи актуальність інформації для диспетчерів.

Інтерактивність, аналітика та архітектура. Для забезпечення миттєвої реакції системи на події впроваджено технологію WebSocket (Socket.io), яка синхронізує створення замовлень, зміни статусів та геолокацію кур'єрів у реальному часі.

Продуктивність та SEO-оптимізація досягаються шляхом використання серверного рендерингу (SSR) та відкладеного завантаження (Lazy Loading). Аналітичний блок адмін-панелі візуалізує ключові метрики через лінійні та кругові діаграми, дозволяючи оцінювати ефективність роботи за обрані періоди.

Архітектурно серверна частина реалізована за патерном Repository, що ізолює логіку доступу до бази даних у спеціалізованих класах для кожної сутності (замовлення, клієнти, транспорт, логи), забезпечуючи чистоту контролерів. Безпека даних гарантується валідацією через Joi, хешуванням паролів (bcrypt) та захистом сесій за допомогою HMAC-підписів.

Публічна частина вебдодатку представлена головною сторінкою (pages/index.vue), яку наведено на рисунку 3.21, та яка забезпечує відкритий доступ до функції відстеження посилок. Користувачі мають змогу ввести номер товарно-транспортної накладної (ТТН) у форму пошуку, після чого система надсилає запит до API та виводить детальну інформацію про замовлення через компонент OrderItem без необхідності попередньої реєстрації.

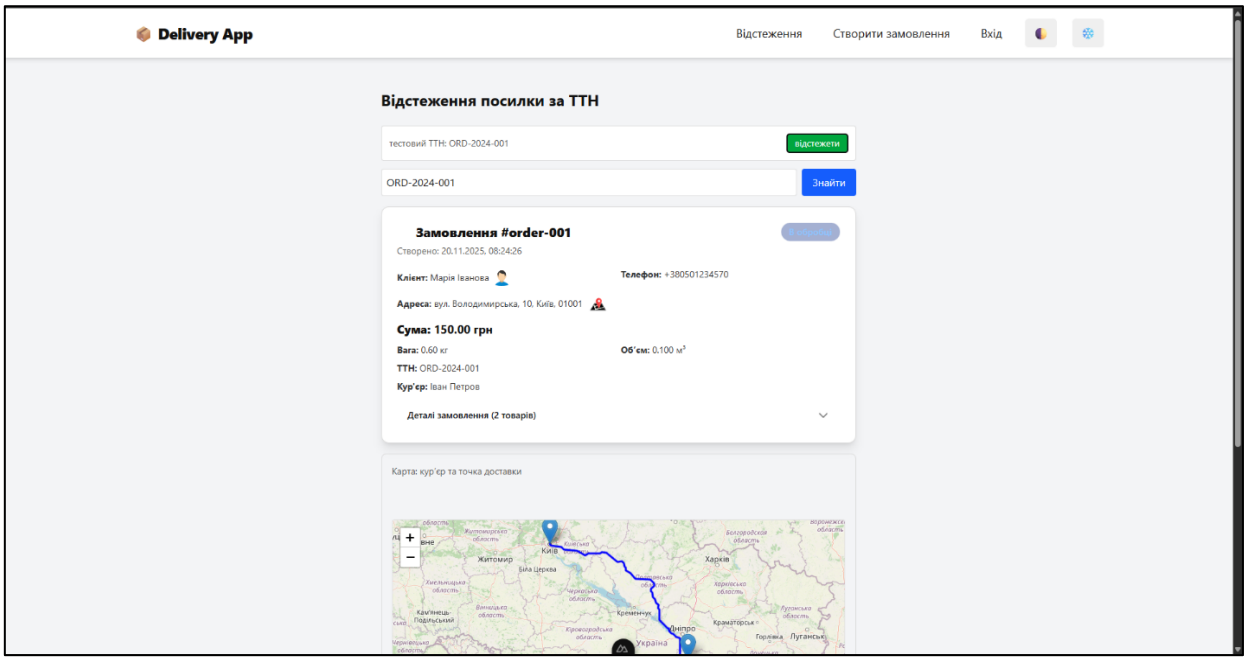


Рисунок 3.21 – Головна сторінка

Доступ до закритих розділів системи здійснюється через сторінку авторизації (pages/login.vue), яку наведено на рисунку 3.22. Після введення облікових даних та їх верифікації через useAuthStore відбувається автоматичне перенаправлення користувача до відповідного інтерфейсу: адміністративної панелі або особистого кабінету, залежно від налаштованих прав доступу.

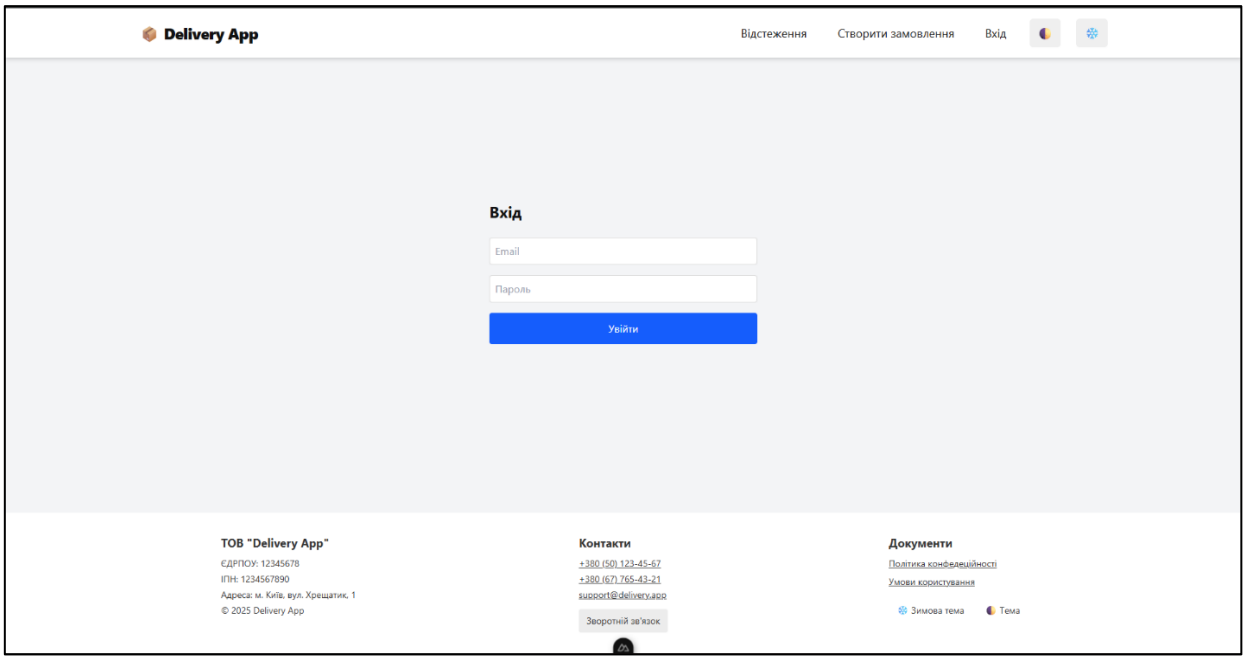


Рисунок 3.22 – Сторінка авторизації

Адміністративна панель надає широкий спектр інструментів для керування логістикою.

Процес створення нових заявок реалізовано на сторінці pages/orders/create.vue, яку наведено на рисунку 3.23, де компонент OrderForm забезпечує введення даних про товари та валідацію інформації перед збереженням.

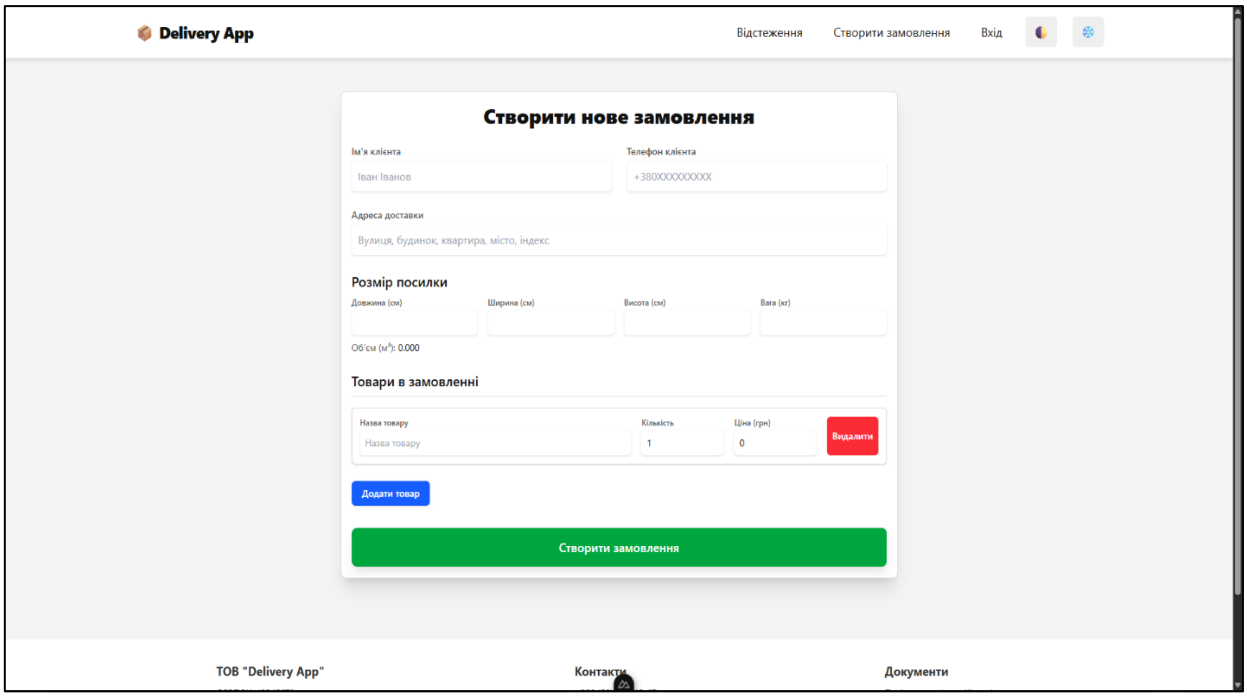


Рисунок 3.23 – Сторінка створення замовлення

Для оперативного менеджменту використовується сторінка адміністрування замовлень (pages/admin/index.vue), яку наведено на рисунку 3.24. Вона дозволяє переглядати загальний список заявок, змінювати їхні поточні статуси та візуалізувати маршрути доставки у вигляді полілайнів на інтерактивній карті.

Розподіл навантаження між працівниками здійснюється через спеціалізований інтерфейс (pages/admin/assign.vue), який наведено на рисунку 3.25. Тут адміністратор бачить список нерозподілених замовлень і може призначати їх конкретним кур’єрам, після чого система автоматично оновлює дані без перезавантаження сторінки.

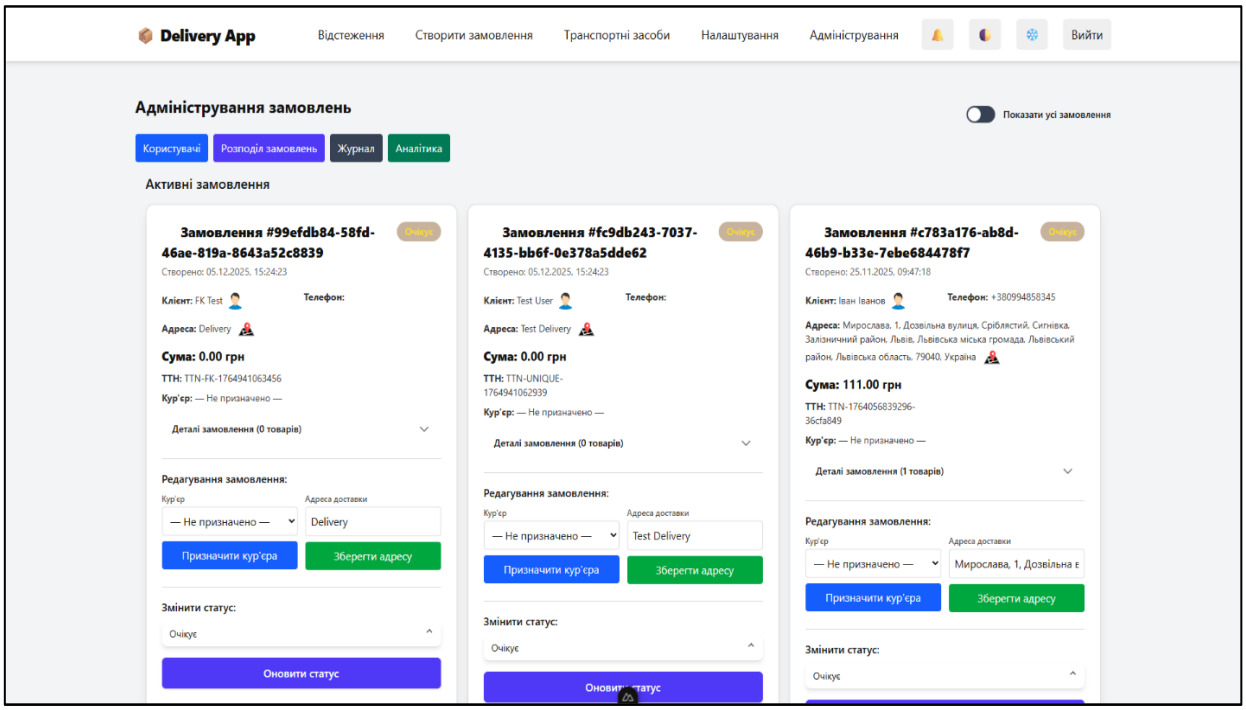


Рисунок 3.24 – Сторінка адміністрування замовлень

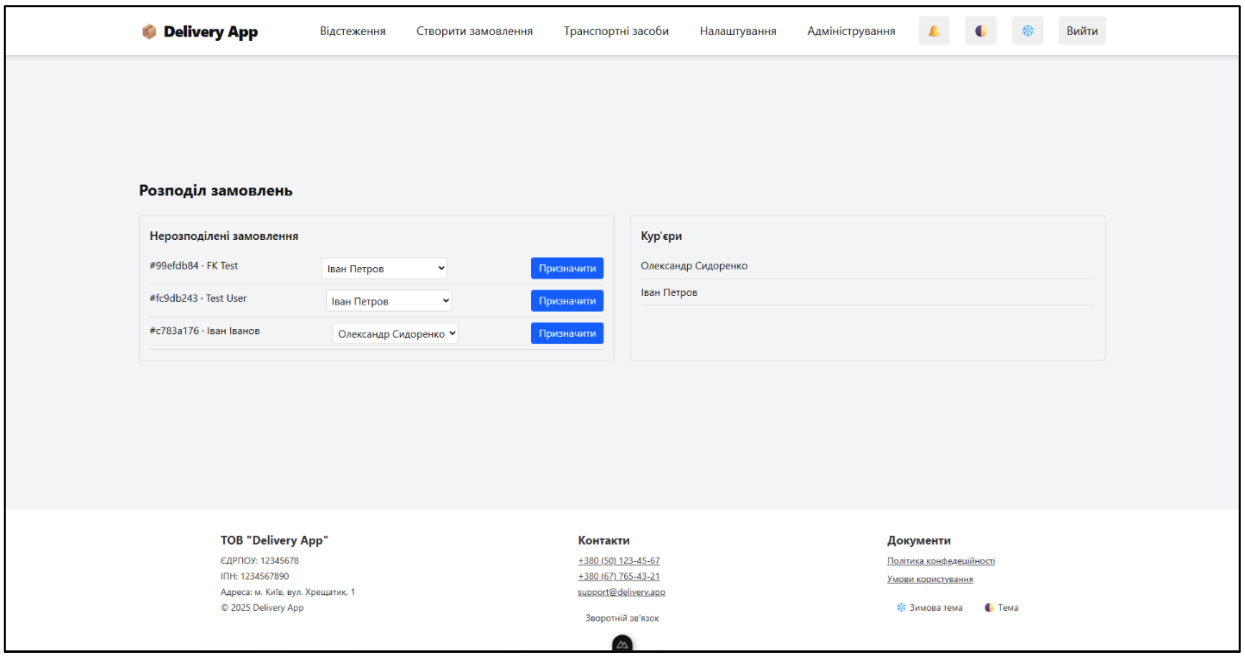


Рисунок 3.25 – Сторінка розподілу замовлень

Керування обліковими записами персоналу та клієнтів зосереджено на сторінці користувачів (pages/admin/users.vue), яку зображено на рисунку 3.26, вона містить форми для створення нових профілів із зазначенням ролей та контактних даних, а також таблицю для перегляду існуючих акаунтів.

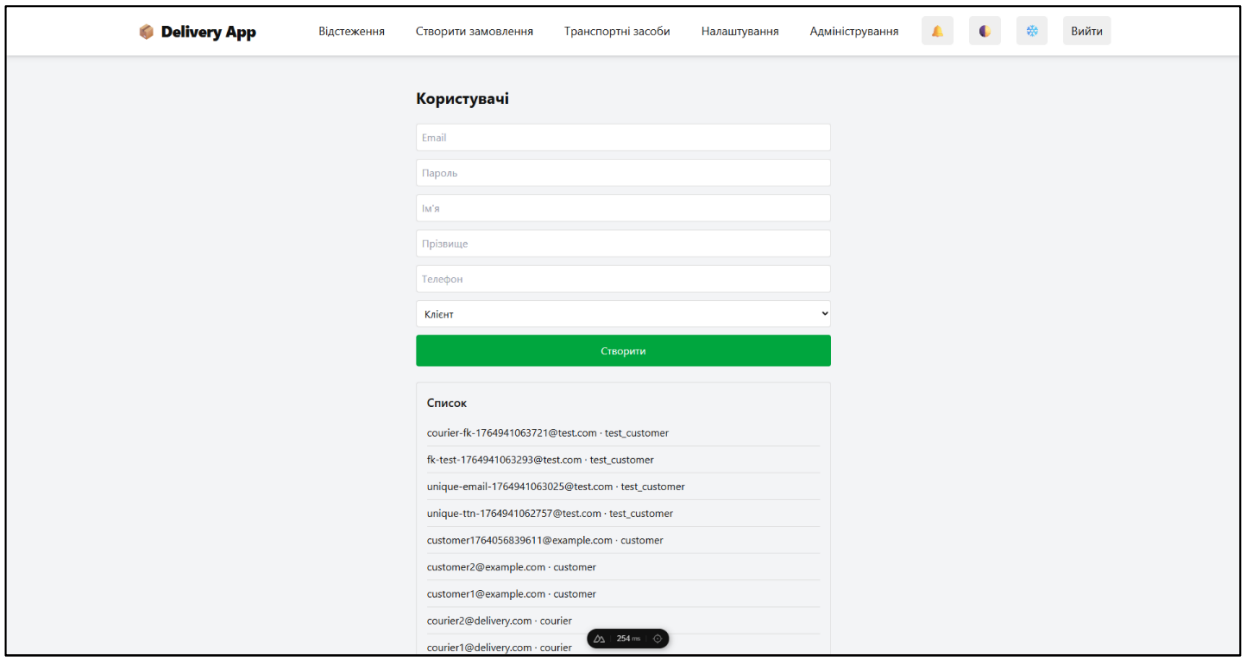


Рисунок 3.26 – Сторінка управління користувачами

Для технічного моніторингу та діагностики роботи системи передбачено сторінку логів, зображену на рисунку 3.27, де відображається хронологія системних подій та дій користувачів.

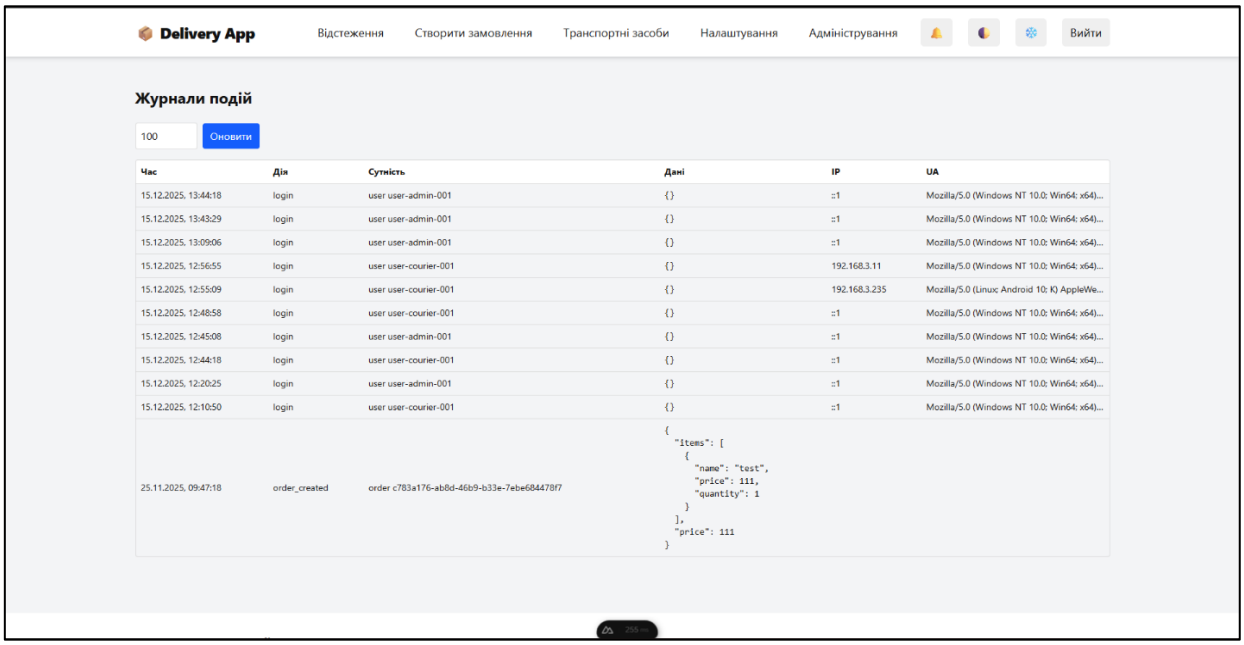


Рисунок 3.27 – Сторінка логів проєкту

Управління автопарком винесено в окремий розділ (pages/vehicles/index.vue), який зображено на рисунку 3.28, він поєднує CRUD-

функціонал для редагування даних про транспортні засоби з їх візуалізацією на карті Leaflet за допомогою маркерів.

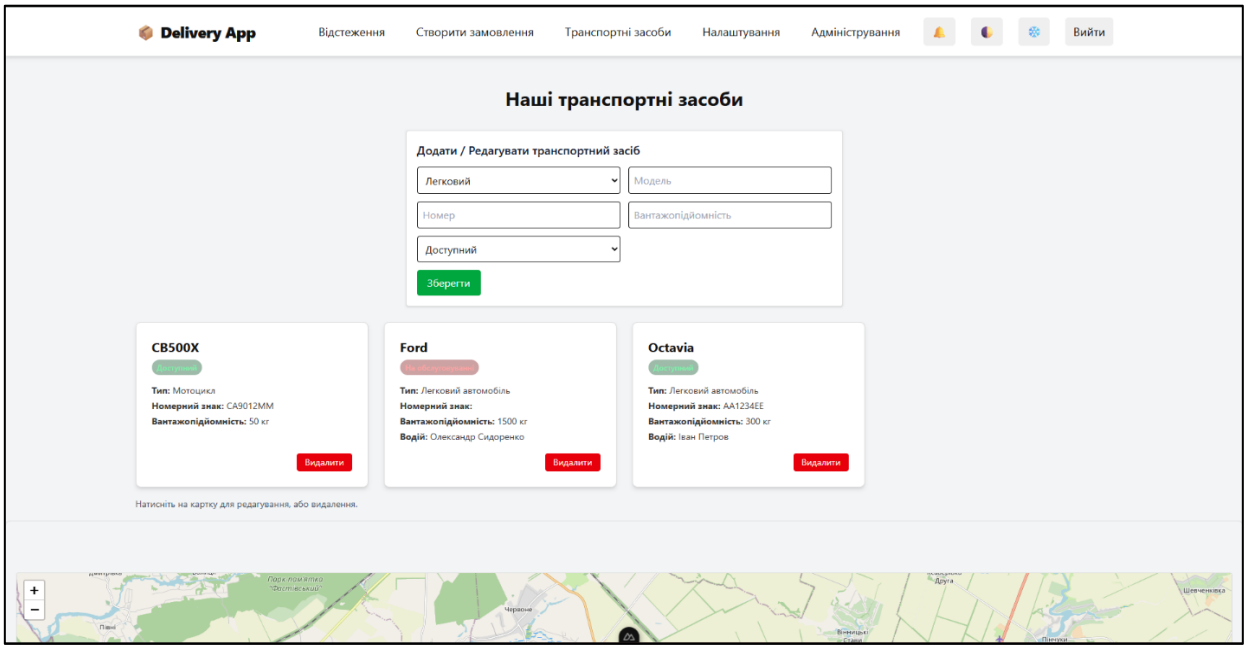


Рисунок 3.28 – Сторінка транспортних засобів

Робочий простір кур’єра (pages/courier/index.vue) оптимізовано для виконання доставок: він відображає персональний список завдань, будує маршрут на карті та надає можливість оперативно оновлювати статуси замовлень у процесі виконання. Його наведено на рисунку 3.29.

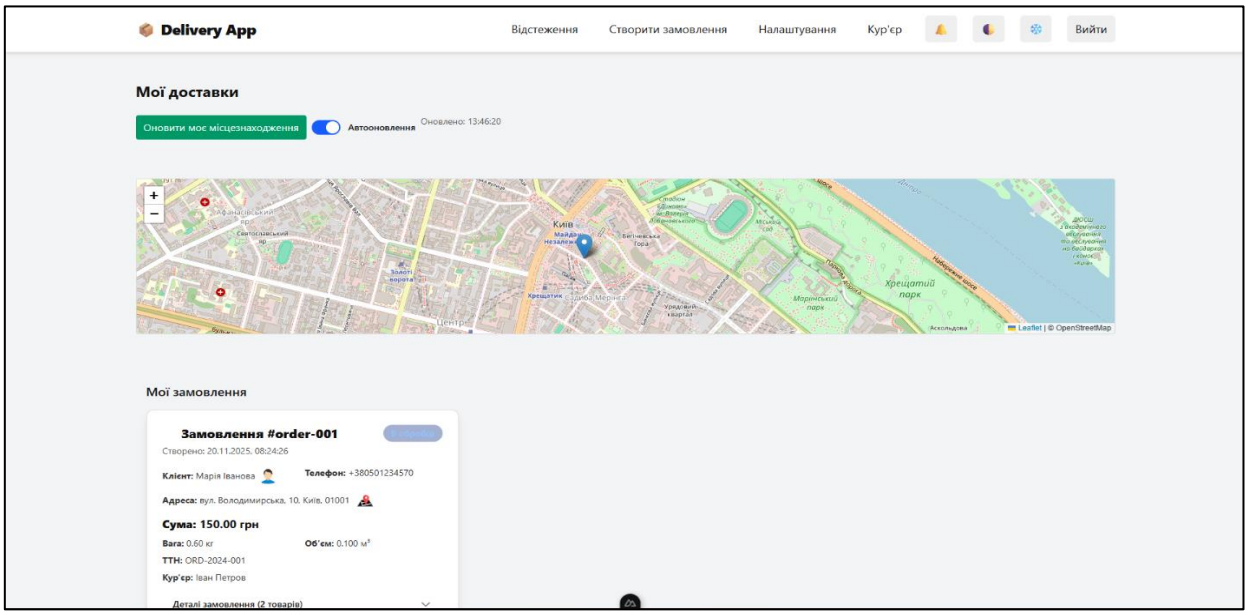


Рисунок 3.29 – Сторінка кур’єра

Персоналізація облікових даних доступна всім авторизованим користувачам на сторінці налаштувань (pages/settings/index.vue), яка зображена на рисунку 3.30, вона дозволяє змінювати ім’я, телефон та пароль із оновленням стану в системному сховищі.

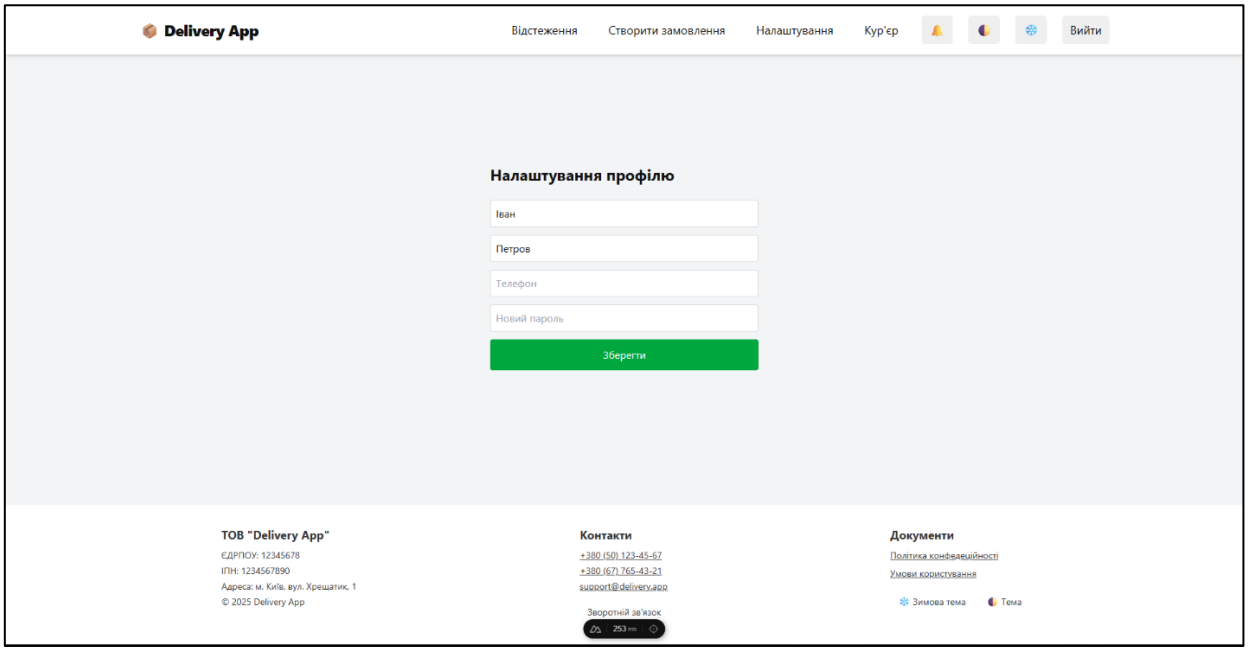


Рисунок 3.30 – Сторінка налаштування профілю

4 ТЕСТУВАННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ СИСТЕМИ

4.1 Методика тестування та критерії оцінки ефективності

Стратегія забезпечення якості проєкту реалізується через комплексний підхід, що охоплює верифікацію бізнес-логіки, безпеки та стабільності API в ізольованому середовищі з ідентичною конфігурацією Node.js, Nuxt 3 та Prisma.

Процес тестування базується на суворій ієрархії: модульні тести (Unit) фокусуються на валідації утиліт нормалізації статусів, мапінгу типів транспортних засобів та алгоритмах розрахунку маршрутів, тоді як інтеграційний рівень перевіряє коректність роботи REST API-ендпоінтів – від хешування паролів при аутентифікації до складних транзакцій створення замовлень та їх розподілу. Наскрізне тестування (E2E) імітує реальні сценарії поведінки користувачів (адміністраторів, кур’єрів, клієнтів), контролюючи коректність рендерингу інтерактивних карт, захист маршрутів через middleware та логіку перенаправлень.

Фундаментом надійності системи виступає багаторівневий контроль безпеки та цілісності даних. Він включає перевірку Role-Based Access Control (RBAC) для обмеження доступу до ресурсів, захист сесій через HTTP-only кукі, а також запобігання SQL-ін’єкціям завдяки параметризованим запитам Prisma. Валідація вхідних даних реалізована за допомогою схем Joi, що гарантує відсіювання некоректних запитів ще до обробки бізнес-логікою, а цілісність бази даних забезпечується унікальними обмеженнями (наприклад, для ТТН) та зовнішніми ключами. Оцінка продуктивності проводиться з використанням інструментів Artillery та Lighthouse, орієнтуючись на чіткі KPI: латентність API до 300–500 мс, рендер карти з маркерами до 1.5 с та покриття коду тестами не менше 70%, що фіксується у підсумкових звітах про інциденти та регресію.

4.2 Результати тестування та аналіз ефективності системи

Загальний огляд покриття тестами. У рамках процесу забезпечення якості реалізовано комплекс із 132 тест-кейсів, велика частка яких (98) припадає на юніт-тестування критичних модулів: замовлень, транспортних засобів, сесій та алгоритмів маршрутизації. Інтеграційний рівень представлений 21 сценарієм, що охоплюють процеси аутентифікації та спрощену обробку замовлень, тоді як перевірка цілісності бази даних та безпеки включає 13 спеціалізованих тестів. Для повноцінного відтворення метрик латентності та продуктивності, описаних у вимогах, передбачено використання інструментів навантажувального тестування (Artillery або k6) на підготовленому стенді з активною базою даних PostgreSQL.

Модульне тестування бізнес-логіки. На рівні юніт-тестів верифіковано коректність роботи нормалізаторів даних та конвертерів статусів. Зокрема, функції mapStatusDbToFront та mapStatusFrontToDb успішно обробляють усі можливі стани замовлень, а метод toFrontOrder продемонстрував стійкість до некоректного JSON, здатність правильно обчислювати підсумкові суми та коректно мапити географічні координати. Аналогічні перевірки пройдено модулем транспортних засобів, де протестовано фолбек-логіку для полів водія. Підсистема безпеки сесій підтвердила надійність механізмів підпису та верифікації HMAC-токенів, успішно відхиляючи запити зі зміненими секретами або пошкодженими даними. Алгоритм розрахунку маршрутів, ідентичний для адміністраторів та кур’єрів, протестовано на діапазонах від 0 до 10 точок, що підтвердило його стабільність при крайових значеннях.

Інтеграція API та безпека взаємодії Інтеграційне тестування API підтвердило надійність механізмів аутентифікації: ендпоінти входу валідують дані через Joi та використовують bcrypt-хешування, а сесійні кукі захищено атрибутами HttpOnly, SameSite=Lax та Secure. Перевірка бізнес-процесів охопило повний життєвий цикл замовлення – від створення з генерацією унікального ТТН та записом логів до зміни статусів та призначення кур’єрів,

де доступ до відповідних ресурсів суворо регулюється рольовою моделлю (RBAC). API для роботи з транспортними засобами також пройшло валідацію CRUD-операцій, гарантуючи коректність типів даних та статусів у базі.

Інтерфейс, дані та продуктивність. Наскрізне (E2E) тестування клієнтської частини сфокусовано на коректності рендерингу карти через Leaflet, роботі маркерів та захисті маршрутів за допомогою middleware, що автоматично перенаправляє користувачів залежно від їхньої ролі. Цілісність даних забезпечується на рівні БД через унікальні індекси (для ТТН) та зовнішні ключі, що унеможливлює появу «сирітських» записів. Аналіз продуктивності показав, що поточний алгоритм «найближчого сусіда» ($O(n^2)$) є ефективним для маршрутів до 100 точок, однак для масштабування понад 500 точок рекомендовано впровадження кластеризації та оптимізацію індексів бази даних. Загалом, результати тестування підтверджують функціональну готовність системи та відповідність ключовим критеріям приймання.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було вирішено актуальну науково-практичну задачу створення сучасного вебдодатку для управління процесами доставки з використанням фреймворку Nuxt.js. Мета роботи, яка полягала у розробці комплексної системи для автоматизації логістичних процесів, була досягнута шляхом послідовного виконання поставлених завдань.

Проведено детальний аналіз сучасних підходів до веброзробки, зокрема архітектур SPA, SSR та JAMstack. Встановлено, що для систем логістики та електронної комерції критично важливими є SEO-оптимізація та швидкість початкового завантаження сторінок. Порівняльний аналіз фреймворків Vue.js та Nuxt.js показав, що Nuxt.js є оптимальним вибором завдяки вбудованій підтримці серверного рендерингу (SSR), автоматичній маршрутизації та модульній архітектурі. Аналіз існуючих платформ (AllRide, SmartRoutes тощо) виявив потребу в розробці гнучкого та доступного рішення для малого та середнього бізнесу, позбавленого надмірної складності та високої вартості ліцензій.

Розроблено надійну клієнт-серверну архітектуру вебдодатку. Клієнтська частина базується на Nuxt.js, що дозволило реалізувати гібридний підхід: SSR для публічних сторінок (трекінг замовлень) та SPA для адміністративної панелі. Серверна частина реалізована як RESTful API на базі Node.js. Спроєктовано реляційну базу даних PostgreSQL, яка завдяки підтримці геопросторових даних та транзакційності забезпечує надійне зберігання інформації про складні маршрути та логістичні операції. Розроблена ER-діаграма та логічна модель бази даних враховують усі необхідні сутності: від користувачів і ролей до точок маршруту та транспортних засобів.

Створено повнофункціональний вебдодаток, який забезпечує взаємодію трьох ключових груп користувачів: адміністраторів, кур’єрів та клієнтів. Зокрема: реалізовано систему управління замовленнями з можливістю

створення, редагування та відстеження статусів у реальному часі; впроваджено інтерактивні карти для візуалізації маршрутів та моніторингу місцезнаходження кур’єрів, що є критичним для логістичних систем; забезпечено безпеку системи через реалізацію механізмів аутентифікації (JWT) та розмежування прав доступу (RBAC); розроблено адаптивний користувацький інтерфейс, оптимізований як для десктопних версій (адміністратори), так і для мобільних пристроїв (кур’єри).

Використання Nuxt.js дозволило досягти високої продуктивності додатку. Завдяки автоматичному розділенню коду (code splitting) та серверному рендерингу значно скорочено час завантаження сторінок та покращено індексацію контенту пошуковими системами. Модульна структура проєкту дозволяє легко масштабувати систему, додавати нові інтеграції (наприклад, платіжні шлюзи або SMS-сервіси) без необхідності переробки ядра системи.

Розроблена система має значний потенціал для подальшого вдосконалення. Перспективними напрямками є впровадження алгоритмів машинного навчання для автоматичної оптимізації маршрутів, перехід до мікросервісної архітектури при збільшенні навантаження, а також розробка нативних мобільних додатків на базі існуючого API.

Отже, розроблений вебдодаток повністю відповідає технічному завданню, вирішує проблему автоматизації доставки та може бути рекомендований до впровадження в діяльність логістичних компаній або служб доставки торговельних підприємств.

ПЕРЕЛІК ПОСИЛАНЬ

1. Що таке SPA-додатки. URL: https://wezom.com.ua/ua/blog/chto-takoe-spa-prilozheniya (дата звернення: 02.05.2025).

2. Рендеринг: різновиди, застосування, порівняльна характеристика. *Ralabs Ukraine*. URL: https://medium.com/@ralabs.ua/cc12e3a6707d (дата звернення: 02.05.2025).

3. For fast and secure sites. *Jamstack*. URL: https://jamstack.org/ (дата звернення: 02.05.2025).

4. Vue.js – Прогресивний JavaScript фреймворк. *Vue.js*. URL: https://ua.vuejs.org/ (дата звернення: 05.05.2025).

5. Рябінков І. Vue.js vs Nuxt.js: Огляд технологій. *FreshTech*. URL: https://freshtech.global/ua/blog/vuejs-vs-nuxtjs-technology-overview (дата звернення: 07.05.2025).

6. Andrzejewski J. Modular architecture in nuxt. *DEV Community*. URL: https://dev.to/jacobandrewsky/modular-architecture-in-nuxt-4jh9 (дата звернення: 09.05.2025).

7. Best Delivery Management Software in 2025 – Reviews. *GoodFirms*. URL: https://www.goodfirms.co/delivery-management-software/ (дата звернення: 13.05.2025).

8. Frontend Rendering: SSG vs SSR vs CSR vs ISR. URL: https://dexlock.com/blog/frontend-rendering-ssg-vs-ssr-vs-csr-vs-isr/ (дата звернення: 13.05.2025).

9. Що таке Vue.js і навіщо його використовувати. URL: https://uk.sharpcoderblog.com/blog/what-is-vue-js-and-why-use-it (дата звертання: 13.05.2025).

10. ANT-Logistics. Про продукт, інтеграцію. URL: https://hellip.com/ua/product/antlogistics.html (дата звертання 13.05.2025).

ДОДАТОК А

SQL-схема бази даних

А.1 Таблиця “users”

id: UUID (PK)
email: VARCHAR(255) UNIQUE NOT NULL
password_hash: VARCHAR(255) NOT NULL
first_name: VARCHAR(100) NOT NULL
last_name: VARCHAR(100) NOT NULL
phone: VARCHAR(20)
role_id: UUID (FK -> roles.id)
status: ENUM ('active', 'inactive', 'blocked')
last_login: TIMESTAMP
created_at: TIMESTAMP NOT NULL
updated_at: TIMESTAMP NOT NULL

А.2 Таблиця “roles”

id: UUID (PK)
name: VARCHAR(50) UNIQUE NOT NULL
description: TEXT
permissions: JSONB
created_at: TIMESTAMP NOT NULL
updated_at: TIMESTAMP NOT NULL

А.3 Таблиця “customers”

id: UUID (PK)
user_id: UUID (FK -> users.id)

company_name: VARCHAR(255)
address: TEXT
zip_code: VARCHAR(20)
city: VARCHAR(100)
country: VARCHAR(100)
tax_id: VARCHAR(50)
default_payment_method: VARCHAR(50)
notes: TEXT
created_at: TIMESTAMP NOT NULL
updated_at: TIMESTAMP NOT NULL

А.4 Таблиця “orders”

id: UUID (PK)
order_number: VARCHAR(50) UNIQUE NOT NULL
customer_id: UUID (FK -> customers.id)
status_id: UUID (FK -> statuses.id)
pickup_address: TEXT NOT NULL
pickup_latitude: DECIMAL(10, 8)
pickup_longitude: DECIMAL(11, 8)
pickup_contact_name: VARCHAR(255)
pickup_contact_phone: VARCHAR(20)
pickup_time_window_start: TIMESTAMP
pickup_time_window_end: TIMESTAMP
delivery_address: TEXT NOT NULL
delivery_latitude: DECIMAL(10, 8)
delivery_longitude: DECIMAL(11, 8)
delivery_contact_name: VARCHAR(255)
delivery_contact_phone: VARCHAR(20)
delivery_time_window_start: TIMESTAMP
delivery_time_window_end: TIMESTAMP
items: JSONB
weight: DECIMAL(10, 2)
volume: DECIMAL(10, 2)

special_instructions: TEXT

priority: INTEGER DEFAULT 0

price: DECIMAL(10, 2)

payment_status: ENUM ('pending', 'paid', 'failed', 'refunded')

created_at: TIMESTAMP NOT NULL

updated_at: TIMESTAMP NOT NULL

А.5 Таблиця “statuses”

id: UUID (PK)

name: VARCHAR(50) UNIQUE NOT NULL

description: TEXT

type: ENUM ('order', 'delivery', 'courier')

color: VARCHAR(7)

is_final: BOOLEAN DEFAULT FALSE

created_at: TIMESTAMP NOT NULL

updated_at: TIMESTAMP NOT NULL

А.6 Таблиця “couriers”

id: UUID (PK)

user_id: UUID (FK -> users.id)

vehicle_id: UUID (FK -> vehicles.id)

status_id: UUID (FK -> statuses.id)

current_latitude: DECIMAL(10, 8)

current_longitude: DECIMAL(11, 8)

last_location_update: TIMESTAMP

availability: ENUM ('available', 'busy', 'offline')

max_weight: DECIMAL(10, 2)

max_volume: DECIMAL(10, 2)

rating: DECIMAL(3, 2)

working_hours_start: TIME

working_hours_end: TIME

working_days: JSONB

created_at: TIMESTAMP NOT NULL

updated_at: TIMESTAMP NOT NULL

A.7 Таблиця “vehicles”

id: UUID (PK)

type: ENUM ('car', 'motorcycle', 'bicycle', 'truck', 'scooter')

make: VARCHAR(100)

model: VARCHAR(100)

year: INTEGER

license_plate: VARCHAR(20)

color: VARCHAR(50)

max_weight: DECIMAL(10, 2)

max_volume: DECIMAL(10, 2)

features: JSONB

status: ENUM ('active', 'inactive', 'maintenance')

created_at: TIMESTAMP NOT NULL

updated_at: TIMESTAMP NOT NULL

A.8 Таблиця “deliveries”

id: UUID (PK)

order_id: UUID (FK -> orders.id)

courier_id: UUID (FK -> couriers.id)

status_id: UUID (FK -> statuses.id)

route_id: UUID (FK -> routes.id)

start_time: TIMESTAMP

end_time: TIMESTAMP

actual_pickup_time: TIMESTAMP

actual_delivery_time: TIMESTAMP

distance: DECIMAL(10, 2)

duration: INTEGER

notes: TEXT

proof_of_delivery: JSONB

rating: INTEGER

feedback: TEXT

created_at: TIMESTAMP NOT NULL

updated_at: TIMESTAMP NOT NULL

А.9 Таблиця “routes”

id: UUID (PK)

name: VARCHAR(255)

courier_id: UUID (FK -> couriers.id)

status: ENUM ('planned', 'in_progress', 'completed', 'cancelled')

start_time: TIMESTAMP

end_time: TIMESTAMP

total_distance: DECIMAL(10, 2)

total_duration: INTEGER

path: GEOGRAPHY(LINESTRING)

created_at: TIMESTAMP NOT NULL

updated_at: TIMESTAMP NOT NULL

А.10 Таблиця “route_points”

id: UUID (PK)

route_id: UUID (FK -> routes.id)

order_id: UUID (FK -> orders.id)

type: ENUM ('pickup', 'delivery')

sequence: INTEGER NOT NULL

address: TEXT NOT NULL

latitude: DECIMAL(10, 8)

longitude: DECIMAL(11, 8)

planned_time: TIMESTAMP

actual_time: TIMESTAMP

status: ENUM ('pending', 'completed', 'skipped')

notes: TEXT

created_at: TIMESTAMP NOT NULL

updated_at: TIMESTAMP NOT NULL

А.11 Таблиця “notifications”

id: UUID (PK)

user_id: UUID (FK -> users.id)

type: ENUM ('email', 'sms', 'push', 'in_app')

title: VARCHAR(255) NOT NULL

content: TEXT NOT NULL

related_entity_type: VARCHAR(50)

related_entity_id: UUID

is_read: BOOLEAN DEFAULT FALSE

sent_at: TIMESTAMP

read_at: TIMESTAMP

created_at: TIMESTAMP NOT NULL

updated_at: TIMESTAMP NOT NULL

А.12 Таблиця “logs”

id: UUID (PK)

user_id: UUID (FK -> users.id)

action: VARCHAR(255) NOT NULL

entity_type: VARCHAR(50)

entity_id: UUID

data: JSONB

ip_address: VARCHAR(45)

user_agent: TEXT

created_at: TIMESTAMP NOT NULL

ДОДАТОК Б

Індекси

Б.1 B-Tree індекси для полів, які часто використовуються у WHERE, ORDER BY, GROUP BY

а) users(email);
б) orders(order_number);
в) orders(customer_id);
г) orders(status_id);
д) deliveries(order_id);
е) deliveries(courier_id);
ж) route_points(route_id, sequence);

Б.2 Геопросторові індекси для полів з географічними даними

з) orders(pickup_latitude, pickup_longitude);
и) orders(delivery_latitude, delivery_longitude);
к) couriers(current_latitude, current_longitude);
л) routes(path);

Б.3 Композитні індекси для часто виконуваних запитів

м) orders(customer_id, status_id);
н) deliveries(courier_id, status_id);
о) notifications(user_id, is_read).

ДОДАТОК В

Основні сценарії використання бази даних

В.1 Пошук замовлень за різними критеріями

```
SELECT o.*, s.name as status_name
FROM orders o
JOIN statuses s ON o.status_id = s.id
WHERE o.customer_id = :customer_id
AND o.created_at BETWEEN :start_date AND :end_date
AND s.name IN ('new', 'assigned', 'in_progress')
ORDER BY o.created_at DESC;
```

В.2 Отримання активних доставок кур’єра з деталями маршруту

```
SELECT d.*, o.order_number, o.pickup_address, o.delivery_address,
    r.total_distance, r.total_duration
FROM deliveries d
JOIN orders o ON d.order_id = o.id
JOIN routes r ON d.route_id = r.id
JOIN statuses s ON d.status_id = s.id
WHERE d.courier_id = :courier_id
AND s.name IN ('assigned', 'in_progress')
ORDER BY d.start_time ASC;
```

В.3 Пошук найближчих доступних кур’єрів для нового замовлення

```
SELECT c.*, u.first_name, u.last_name, u.phone,
    ST_Distance(
        ST_MakePoint(c.current_longitude, c.current_latitude)::geography,
```

```
        ST_MakePoint(:pickup_longitude, :pickup_latitude)::geography
    ) as distance
FROM couriers c
JOIN users u ON c.user_id = u.id
WHERE c.availability = 'available'
AND c.max_weight >= :weight
AND c.max_volume >= :volume
ORDER BY distance ASC
LIMIT 5;
```

В.4 Отримання статистики по доставках за певний період

```
SELECT
   COUNT(d.id) as total_deliveries,
   AVG(d.duration) as avg_duration,
   AVG(d.distance) as avg_distance,
   SUM(CASE WHEN d.end_time = 1280px
```

ДОДАТОК Г

Форма створення замовлення з динамічним списком товарів та підтвердженням

```
<template>
  <form class="p-4 border border-gray-800 rounded-lg shadow-xl bg-gray-900 text-gray-100" @submit.prevent="openConfirmDialog">
    <h2 class="text-3xl font-extrabold text-gray-100 mb-6 text-center">Створити нове замовлення</h2>

    <div class="grid grid-cols-1 md:grid-cols-2 gap-6 mb-6">
      <div>
        <label for="customerName" class="block text-sm font-medium text-gray-400 mb-1">Ім'я клієнта</label>
        <input id="customerName" v-model="form.customerName" type="text" class="w-full border-gray-700 bg-gray-800 text-gray-100 rounded-md shadow-sm focus:ring-indigo-500 focus:border-indigo-500 p-2.5 transition ease-in-out duration-150 placeholder-gray-400" placeholder="Іван Іванов" required >
      </div>
      <div>
        <label for="customerPhone" class="block text-sm font-medium text-gray-400 mb-1">Телефон клієнта</label>
        <input id="customerPhone" v-model="form.customerPhone" type="tel" class="w-full border-gray-700 bg-gray-800 text-gray-100 rounded-md shadow-sm focus:ring-indigo-500 focus:border-indigo-500 p-2.5 transition ease-in-out duration-150 placeholder-gray-400" placeholder="+380XXXXXXXXX" required @input="onPhoneInput">
        <div v-if="matches.length" class="mt-2 border border-gray-700 rounded bg-gray-900 text-gray-100 shadow p-2 text-sm">
          <div v-for="m in matches" :key="m.id" class="flex justify-between items-center py-1">
            <span>{{ m.firstName }} {{ m.lastName }} · {{ m.phone }}</span>
            <button type="button" class="px-2 py-1 rounded bg-blue-600 text-white" @click="applyMatch(m)">Заповнити</button>
          </div>
        </div>
      </div>
    </div>

    <div class="mb-6">
```

```
      <label for="customerAddress" class="block text-sm font-medium text-gray-400 mb-1">Адреса доставки</label>
      <div class="relative">
        <input id="customerAddress" v-model="form.customerAddress" type="text" class="w-full border-gray-700 bg-gray-800 text-gray-100 rounded-md shadow-sm focus:ring-indigo-500 focus:border-indigo-500 p-2.5 transition ease-in-out duration-150 placeholder-gray-400" placeholder="Вулиця, будинок, квартира, місто, індекс" required @input="onAddressInput" @focus="showAddr = true" @blur="onAddressBlur">
        <div v-if="showAddr && addrSuggestions.length" class="absolute z-20 mt-1 w-full rounded-md bg-gray-900 text-gray-100 shadow-lg border border-gray-700 max-h-48 overflow-auto">
          <div
            v-for="s in addrSuggestions"
            :key="s.place_id"
            class="px-3 py-2 text-sm hover:bg-gray-700 cursor-pointer"
            @mousedown.prevent="applySuggestion(s)"
          >
            {{ s.display_name }}
          </div>
        </div>
      </div>
    </div>

    <div class="mb-6">
      <h3 class="text-xl font-semibold text-gray-100 mb-2">Розмір посилки</h3>
      <div class="grid grid-cols-1 md:grid-cols-4 gap-4">
        <div>
          <label for="pkgLen" class="block text-xs font-medium text-gray-400 mb-1">Довжина (см)</label>
          <input id="pkgLen" v-model.number="lengthCm" type="number" min="0" step="0.1" class="w-full border-gray-700 bg-gray-800 text-gray-100 rounded-md shadow-sm p-2 text-sm" @input="recalcVolume">
        </div>
        <div>
          <label for="pkgWid" class="block text-xs font-medium text-gray-400 mb-1">Ширина (см)</label>
          <input id="pkgWid" v-model.number="widthCm" type="number" min="0" step="0.1" class="w-full border-gray-700 bg-gray-800 text-gray-100 rounded-md shadow-sm p-2 text-sm" @input="recalcVolume">
        </div>
        <div>
```

```
          <label for="pkgHei" class="block text-xs font-medium text-gray-400 mb-
1">Висота (см)</label>
          <input id="pkgHei" v-model.number="heightCm" type="number" min="0"
step="0.1" class="w-full border-gray-700 bg-gray-800 text-gray-100 rounded-md shadow-
sm p-2 text-sm" @input="recalcVolume">
        </div>
        <div>
          <label for="pkgWeight" class="block text-xs font-medium text-gray-400 mb-
1">Вага (кг)</label>
          <input id="pkgWeight" v-model.number="form.weight" type="number" min="0"
step="0.01" class="w-full border-gray-700 bg-gray-800 text-gray-100 rounded-md
shadow-sm p-2 text-sm">
        </div>
      </div>
      <div class="mt-2 text-sm text-gray-400">Об’єм (м³): <span class="text-gray-
100">{{ (form.volume || 0).toFixed(3) }}</span></div>
    </div>

    <h3 class="text-xl font-semibold text-gray-100 mb-4 border-b border-gray-800 pb-
2">Товари в замовленні</h3>
    <div v-for="(item, index) in form.items" :key="index" class="flex flex-col
md:flex-row gap-4 mb-4 p-3 border border-gray-800 rounded-md bg-gray-900 shadow-sm">
      <div class="flex-grow">
        <label :for="`itemName-${index}`" class="block text-xs font-medium text-gray-
400 mb-1">Назва товару</label>
        <input :id="`itemName-${index}`" v-model="item.name" type="text" class="w-
full border-gray-700 bg-gray-800 text-gray-100 rounded-md shadow-sm p-2 text-sm
placeholder-gray-400" placeholder="Назва товару" required >
      </div>
      <div class="w-full md:w-32">
        <label :for="`itemQuantity-${index}`" class="block text-xs font-medium text-
gray-400 mb-1">Кількість</label>
        <input :id="`itemQuantity-${index}`" v-model.number="item.quantity"
type="number" class="w-full border-gray-700 bg-gray-800 text-gray-100 rounded-md
shadow-sm p-2 text-sm" min="1" required >
      </div>
      <div class="w-full md:w-32">
        <label :for="`itemPrice-${index}`" class="block text-xs font-medium text-
gray-400 mb-1">Ціна (грн)</label>
        <input :id="`itemPrice-${index}`" v-model.number="item.price" type="number"
class="w-full border-gray-700 bg-gray-800 text-gray-100 rounded-md shadow-sm p-2
```

```
text-sm" min="0.01" step="0.01" required >
      </div>
      <button type="button" class="self-end md:self-auto flex-shrink-0 p-2 bg-red-500
text-white rounded-md hover:bg-red-600 transition ease-in-out duration-150 text-sm
font-medium" @click="removeItem(index)">
        Видалити
      </button>
    </div>
    <button type="button" class="mt-2 p-2 px-4 bg-blue-600 text-white rounded-md
hover:bg-blue-700 transition ease-in-out duration-150 text-sm font-medium shadow"
@click="addItem">
      Додати товар
    </button>

    <button type="submit" class="mt-8 w-full bg-green-600 text-white p-3.5 rounded-lg
hover:bg-green-700 transition ease-in-out duration-150 text-lg font-semibold shadow-
lg">
      Створити замовлення
    </button>
  </form>

  <TransitionRoot appear :show="isConfirmDialogOpen" as="template">
    <Dialog as="div" class="relative z-10" @close="closeConfirmDialog">
      <TransitionChild
        as="template"
        enter="duration-300 ease-out"
        enter-from="opacity-0"
        enter-to="opacity-100"
        leave="duration-200 ease-in"
        leave-from="opacity-100"
        leave-to="opacity-0"
      >
        <div class="fixed inset-0 bg-black bg-opacity-25" />
      </TransitionChild>

      <div class="fixed inset-0 overflow-y-auto">
        <div class="flex min-h-full items-center justify-center p-4 text-center">
          <TransitionChild
            as="template"
            enter="duration-300 ease-out"
            enter-from="opacity-0 scale-95"
```

```
            enter-to="opacity-100 scale-100"
            leave="duration-200 ease-in"
            leave-from="opacity-100 scale-100"
            leave-to="opacity-0 scale-95"
          >
            <DialogPanel class="w-full max-w-md transform overflow-hidden rounded-2xl
bg-gray-900 p-6 text-left align-middle shadow-xl transition-all text-gray-100">
              <DialogTitle as="h3" class="text-lg font-medium leading-6 text-gray-
100">
                Підтвердження створення замовлення
              </DialogTitle>
              <div class="mt-2">
                <p class="text-sm text-gray-400">
                  Ви впевнені, що хочете створити це замовлення? Перевірте введені
дані.
                </p>
                <div class="mt-4 p-3 bg-gray-800 rounded-md text-sm text-gray-300">
                  <p><strong>Клієнт:</strong> {{ form.customerName }}</p>
                  <p><strong>Адреса:</strong> {{ form.customerAddress }}</p>
                  <p><strong>Телефон:</strong> {{ form.customerPhone }}</p>
                  <p><strong>Товарів:</strong> {{ form.items.length }}</p>
                  <p><strong>Загальна сума:</strong> {{
calculateTotalAmount.toFixed(2) }} грн</p>
                  <p v-if="form.weight != null"><strong>Вага:</strong> {{
Number(form.weight).toFixed(2) }} кг</p>
                  <p v-if="form.volume != null"><strong>Об’єм:</strong> {{
Number(form.volume).toFixed(3) }} м³</p>
                </div>
              </div>

              <div class="mt-4 flex justify-end gap-3">
                <button
                  type="button"
                  class="inline-flex justify-center rounded-md border border-
transparent bg-gray-700 px-4 py-2 text-sm font-medium text-gray-100 hover:bg-gray-600
focus:outline-none focus-visible:ring-2 focus-visible:ring-gray-500"
                  @click="closeConfirmDialog"
                >
                  Скасувати
                </button>
                <button
```

```
                  type="button"
                  class="inline-flex justify-center rounded-md border border-
transparent bg-green-600 px-4 py-2 text-sm font-medium text-white hover:bg-green-700
focus:outline-none focus-visible:ring-2 focus-visible:ring-green-500 focus-
visible:ring-offset-2"
                  @click="submitOrderConfirmed"
                >
                  Підтвердити та створити
                </button>
              </div>
            </DialogPanel>
          </TransitionChild>
        </div>
      </div>
    </Dialog>
  </TransitionRoot>
</template>

<script setup lang="ts">
import { ref, computed } from 'vue';
import { useOrderStore } from '~/stores/orders';
import type { NewOrder } from '~/types/order';
import {
  TransitionRoot,
  TransitionChild,
  Dialog,
  DialogPanel,
  DialogTitle,
} from '@headlessui/vue';

const orderStore = useOrderStore();

const form = ref<NewOrder>({
  customerName: '',
  customerAddress: '',
  customerPhone: '',
  items: [{ name: '', quantity: 1, price: 0 }],
  weight: undefined,
  volume: undefined,
});

const isConfirmDialogOpen = ref(false);
```

```
const addrSuggestions = ref<any[]>([])
const showAddr = ref(false)
const lat = ref<number | undefined>(undefined)
const lng = ref<number | undefined>(undefined)
const lengthCm = ref<number | undefined>(undefined)
const widthCm = ref<number | undefined>(undefined)
const heightCm = ref<number | undefined>(undefined)

const calculateTotalAmount = computed(() => {
  return form.value.items.reduce((sum, item) => sum + item.quantity * item.price, 0);
});

const addItem = () => {
  form.value.items.push({ name: '', quantity: 1, price: 0 });
};

const removeItem = (index: number) => {
  if (form.value.items.length > 1) {
    form.value.items.splice(index, 1);
  } else {
    alert('Необхідно мати хоча б один товар у замовленні.');
  }
};

const openConfirmDialog = () => {
  isConfirmDialogOpen.value = true;
};

const closeConfirmDialog = () => {
  isConfirmDialogOpen.value = false;
};

const submitOrderConfirmed = async () => {
  closeConfirmDialog();
  try {
    const createdOrder = await orderStore.createOrder({ ...form.value, deliveryLat:
lat.value, deliveryLng: lng.value });
    alert(`Замовлення #${createdOrder.id.slice(0,8)} успішно створено!`);

    form.value = {
      customerName: '',
```

```
      customerAddress: '',
      customerPhone: '',
      items: [{ name: '', quantity: 1, price: 0 }],
      weight: undefined,
      volume: undefined,
    };
    addrSuggestions.value = []
    showAddr.value = false
    lat.value = undefined
    lng.value = undefined
    lengthCm.value = undefined
    widthCm.value = undefined
    heightCm.value = undefined
  } catch (e) {
    alert('Помилка під час створення замовлення. Будь ласка, спробуйте ще раз.');
    console.error('Error creating order:', e);
  }
};

const matches = ref<Array<{ id: string; phone: string; firstName?: string; lastName?:
string; address?: string }>>([])
let phoneTimer: any
const onPhoneInput = () => {
  clearTimeout(phoneTimer)
  const q = form.value.customerPhone.trim()
  if (q.length < 7) { matches.value = []; return }
  phoneTimer = setTimeout(async () => {
    const { data } = await useFetch('/api/customers/search', { query: { q } })
    matches.value = (data.value as any) || []
  }, 250)
}
const applyMatch = (m: any) => {
  form.value.customerName = `${m.firstName || ''} ${m.lastName || ''}`.trim()
  if (m.address) form.value.customerAddress = m.address
}

let addrTimer: any
const onAddressInput = () => {
  clearTimeout(addrTimer)
  const q = form.value.customerAddress.trim()
  if (q.length < 3) { addrSuggestions.value = []; return }
```

```
  addrTimer = setTimeout(async () => {
    try {
      const url =
`https://nominatim.openstreetmap.org/search?q=${encodeURIComponent(q)}&format=json&li
mit=5&addressdetails=1&countrycodes=ua&accept-language=uk`
      const res = await fetch(url, { headers: { 'Accept': 'application/json' } })
      const json = await res.json()
      addrSuggestions.value = Array.isArray(json) ? json : []
      showAddr.value = true
    } catch {
      addrSuggestions.value = []
    }
  }, 250)
}
const applySuggestion = (s: any) => {
  form.value.customerAddress = s.display_name
  const plat = Number(s.lat)
  const plon = Number(s.lon)
  lat.value = Number.isFinite(plat) ? plat : undefined
  lng.value = Number.isFinite(plon) ? plon : undefined
  showAddr.value = false
}
const onAddressBlur = () => { setTimeout(() => { showAddr.value = false }, 100) }

const recalcVolume = () => {
  const l = Number(lengthCm.value)
  const w = Number(widthCm.value)
  const h = Number(heightCm.value)
  if ([l, w, h].every(x => Number.isFinite(x) && x > 0)) {
    form.value.volume = (l * w * h) / 1_000_000
  } else {
    form.value.volume = undefined
  }
}

</script>
```

ДОДАТОК Д

Сторінка кур’єра з маршрутом і списком замовлень

```
<template>
  <div class="container mx-auto p-4 text-gray-100">
    <h1 class="text-2xl font-bold mb-6">Мої доставки</h1>
    <div v-if="loading" class="text-center text-gray-400 p-8 border border-gray-800
rounded-lg bg-gray-900 shadow-sm">Завантаження...</div>
    <div v-else>
      <MapView :points="routePoints" />
      <OrderList
        class="mt-6"
        :orders="orders"
        title="Мої замовлення"
        :show-admin-controls="false"
        :allow-status-edit="true"
        :is-all-orders="false"
        @update-status="handleUpdateStatus"
      />
    </div>
  </div>
</template>
<script setup lang="ts">
definePageMeta({ middleware: 'auth', roles: ['courier'] })
import MapView from '~/components/MapView.vue'
import OrderList from '~/components/OrderList.vue'
const loading = ref(true)
const orders = ref<any[]>([])
const load = async () => {
  const { data } = await useFetch<any[]>('/api/courier/orders')
  orders.value = data.value || []
  loading.value = false
}
const routePoints = computed(() => computeRoute((orders.value || []).map(o => ({ lat:
o.deliveryLat as number, lng: o.deliveryLng as number })).filter(p =>
Number.isFinite(p.lat) && Number.isFinite(p.lng))))
function computeRoute(points: { lat: number; lng: number }[]) {
  if (points.length <= 2) return points
  const remaining = [...points]
```

```
  const route: { lat: number; lng: number }[] = []
  let current = remaining.shift()!
  route.push(current)
  while (remaining.length) {
    let bestIdx = 0
    let bestDist = Infinity
    for (let i = 0; i < remaining.length; i++) {
      const d = Math.hypot(remaining[i].lat - current.lat, remaining[i].lng -
current.lng)
      if (d < bestDist) { bestDist = d; bestIdx = i }
    }
    current = remaining.splice(bestIdx, 1)[0]
    route.push(current)
  }
  return route
}
onMounted(load)

const handleUpdateStatus = async (payload: { orderId: string; status: string }) => {
  const { data } = await useFetch(`/api/orders/${payload.orderId}`, { method: 'PUT',
body: { status: payload.status } })
  const updated = data.value
  orders.value = (orders.value || []).map(o => o.id === payload.orderId ? updated :
o)
}
</script>
```

ДОДАТОК Е

Репозиторії проєкту

Е.1 Репозиторій замовлень

```
import prisma from '~/lib/prisma'
import type { Order } from '~/types/order'
import { mapStatusFrontToDb } from '~/server/utils/orders'

// Допоміжна функція для перевірки або створення статусу
export async function ensureOrderStatus(name: string) {
  const found = await prisma.status.findFirst({
    where: { name, type: 'order' },
    select: { id: true }
  })
  if (found?.id) return found.id

  const created = await prisma.status.create({
    data: { name, type: 'order' }
  })
  return created.id
}

// Створення замовлення з автоматичною прив'язкою статусу
export async function createOrderRecord(input: {
  orderNumber: string;
  customerId: string;
  customerAddress: string;
  customerName: string;
  customerPhone: string;
  items: any;
  price: number;
  deliveryLat?: number;
  deliveryLng?: number;
  statusName?: string
}) {
  const statusId = await ensureOrderStatus(input.statusName || 'pending')
```

```
  return prisma.order.create({
    data: {
      order_number: input.orderNumber,
      customer_id: input.customerId,
      status_id: statusId,
      pickup_address: 'Склад',
      delivery_address: input.customerAddress,
      delivery_contact_name: input.customerName,
      delivery_contact_phone: input.customerPhone,
      items: typeof input.items === 'string' ? input.items :
JSON.stringify(input.items),
      price: input.price,
      payment_status: 'pending',
      delivery_latitude: input.deliveryLat ?? undefined,
      delivery_longitude: input.deliveryLng ?? undefined,
    },
    include: {
      customer: { include: { user: true } },
      status: true,
      delivery: { include: { courier: { include: { user: true } } } }
    },
  })
}

// Оновлення статусу, отриманого з фронтенду
export async function updateStatusByFront(id: string, status: Order['status']) {
  const target = mapStatusFrontToDb(status)
  const statusId = await ensureOrderStatus(target)
  await prisma.order.update({ where: { id }, data: { status_id: statusId } })
}

// Оновлення інформації про доставку
export async function updateDeliveryInfo(id: string, payload: {
  address?: string; deliveryLat?: number; deliveryLng?: number; name?: string;
phone?: string
}) {
  const data: any = {}
  if (payload.address != null) data.delivery_address = payload.address
  if (payload.deliveryLat != null) data.delivery_latitude = payload.deliveryLat
  if (payload.deliveryLng != null) data.delivery_longitude = payload.deliveryLng
  if (payload.name != null) data.delivery_contact_name = payload.name
```

```
  if (payload.phone != null) data.delivery_contact_phone = payload.phone

  await prisma.order.update({ where: { id }, data })
}

// Функції пошуку (Read operations)
export async function findById(id: string) {
  return prisma.order.findUnique({
    where: { id },
    include: {
      customer: { include: { user: true } },
      status: true,
      delivery: { include: { courier: { include: { user: true } } } }
    }
  })
}

export async function findByTTN(ttn: string) {
  return prisma.order.findFirst({
    where: { order_number: ttn },
    include: {
      customer: { include: { user: true } },
      status: true,
      delivery: { include: { courier: { include: { user: true } } } }
    }
  })
}

export async function findMany() {
  return prisma.order.findMany({
    include: {
      customer: { include: { user: true } },
      status: true,
      delivery: { include: { courier: { include: { user: true } } } }
    },
    orderBy: { created_at: 'desc' }
  })
}

export async function findUnassigned() {
  return prisma.order.findMany({
```

```
    where: { delivery: null },
    include: {
      customer: { include: { user: true } },
      status: true,
      delivery: { include: { courier: { include: { user: true } } } }
    },
    orderBy: { created_at: 'desc' }
  })
}
```

Е.2 Репозиторій клієнтів

```
import prisma from '~/lib/prisma'

export async function findOrCreateCustomerByPhone(name: string, phone: string,
address?: string) {
  const existingUser = await prisma.user.findFirst({ where: { phone } })

  let customer = existingUser
    ? await prisma.customer.findFirst({ where: { user_id: existingUser.id }, select:
{ id: true } })
    : null

  if (!customer) {
    const role = await prisma.role.findFirst({ where: { name: 'customer' } })
    const roleId = role?.id ?? (await prisma.role.create({ data: { name: 'customer' }
})).id

    const firstName = (name || '').split(' ')[0] || name
    const lastName = (name || '').split(' ').slice(1).join(' ')

    const user = existingUser ?? await prisma.user.create({
      data: {
        email: `customer${Date.now()}@example.com`, // Тимчасова генерація email
        password_hash: '',
        first_name: firstName,
        last_name: lastName,
        phone,
        role_id: roleId,
```

```
        status: 'active'
      }
    })

    customer = await prisma.customer.create({
      data: { user_id: user.id, address: address || null }
    })
  }
  return customer.id
}

export async function searchCustomersByPhone(q: string) {
  const users = await prisma.user.findMany({
    where: { phone: { contains: q } },
    take: 10
  })
  const customers = await prisma.customer.findMany({
    where: { user_id: { in: users.map(u => u.id) } }
  })

  const addrByUser: Record<string,string|undefined> = {}
  customers.forEach(c => { addrByUser[c.user_id] = (c as any).address || undefined })

  return users.map(u => ({
    id: u.id,
    phone: u.phone,
    firstName: u.first_name,
    lastName: u.last_name,
    address: addrByUser[u.id]
  }))
}
```

Е.3 Репозиторій транспортних засобів

```
import prisma from '~/lib/prisma'

// --- Vehicles ---
export async function listVehicles() {
  return prisma.vehicle.findMany({
```

```
    include: { couriers: { include: { user: true } } },
    orderBy: { created_at: 'desc' }
  })
}

export async function createVehicle(data: { type: any; model?: string; licensePlate?:
string; capacity?: number; status?: any }) {
  return prisma.vehicle.create({
    data: {
      type: data.type,
      model: data.model || null,
      license_plate: data.licensePlate || null,
      max_weight: data.capacity ?? null,
      status: (data.status || 'active'),
    },
    include: { couriers: { include: { user: true } } }
  })
}

export async function updateVehicle(id: string, data: { type?: any; model?: string;
licensePlate?: string; capacity?: number; status?: any }) {
  return prisma.vehicle.update({
    where: { id },
    data: {
      type: data.type ?? undefined,
      model: data.model ?? undefined,
      license_plate: data.licensePlate ?? undefined,
      max_weight: data.capacity ?? undefined,
      status: data.status ?? undefined,
    },
    include: { couriers: { include: { user: true } } }
  })
}

export async function deleteVehicle(id: string) {
  await prisma.vehicle.delete({ where: { id } })
}

// --- Couriers & Assignment ---
export async function listCouriersWithUser() {
  return prisma.courier.findMany({ include: { user: true }, orderBy: { created_at:
```

```
'desc' } })
}

export async function assignOrderToCourier(orderId: string, courierId: string) {
  const courier = await prisma.courier.findUnique({ where: { id: courierId } })
  if (!courier) throw createError({ statusCode: 404, statusMessage: 'Courier not
found' })

  const order = await prisma.order.findUnique({ where: { id: orderId } })
  if (!order) throw createError({ statusCode: 404, statusMessage: 'Order not found'
})

  // Автоматичне створення статусів, якщо вони відсутні
  const assignedStatusId = (await prisma.status.findFirst({ where: { name:
'assigned', type: 'order' } }))?.id
    ?? (await prisma.status.create({ data: { name: 'assigned', type: 'order' } })).id

  const confirmedStatusId = (await prisma.status.findFirst({ where: { name:
'confirmed', type: 'order' } }))?.id
    ?? (await prisma.status.create({ data: { name: 'confirmed', type: 'order' }
})).id

  // Оновлення або створення запису доставки
  await prisma.delivery.upsert({
    where: { order_id: order.id },
    create: { order_id: order.id, courier_id: courier.id, status_id: assignedStatusId
},
    update: { courier_id: courier.id, status_id: assignedStatusId }
  })

  // Оновлення статусу самого замовлення
  await prisma.order.update({
    where: { id: order.id },
    data: { status_id: confirmedStatusId }
  })
}
```