БАКАЛАВРСЬКА РОБОТА

БР. ІП - 90.00.00.000 ПЗ

Група ІП-22-4

Боднар Ростислав

2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Боднар Ростислав Володимирович

(прізвище, ім'я, по батькові)

(індекс)

БАКАЛАВРСЬКА РОБОТА

Розробка онлайн платформи замовлення та доставки їжі

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121– Інженерія програмного забезпечення

(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня Боднар Р. В.

(підпис, ініціали та прізвище здобувача)

Науковий керівник Лютак Ігор Зіновійович, д.т.н, професор

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

Допущено до захисту

Завідувач кафедри

доц. Бандура В. В.

(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

***ЗАТВЕРДЖУЮ*:**
Зав. кафедрою ***ІПЗ***
доц. В.В. Бандура
“ ” 2026 р.

ЗАВДАННЯ

НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Боднару Ростиславу Володимировичу

(прізвище, ім’я, по-батькові)

1. ***Тема проекту* (*роботи*)** ***“Розробка онлайн платформи замовлення та доставки їжі”***

керівник проекту (роботи) Лютак Ігор Зіновійович

затверджені наказом закладу вищої освіти від “ 21 ” квітня 2026 р. № 179/7

2. ***Строк подання студентом проекту* (*роботи*)** 28 травня 2026 р.

3. ***Вихідні дані до проекту* (*роботи*)**

4. ***Зміст розрахунково - пояснювальної записки* (*перелік питань, які потрібно розробити*)**

1. Аналіз предметної області та існуючих рішень
2. Аналіз вимог та постановка задачі проєктування
3. Проєктування архітектури системи
4. Розробка програмного забезпечення сервісу достави їжі
5. Тестування та аналіз результатів

5. ***Перелік графічного матеріалу* (*з точним зазначенням обов’язкових креслень*)**

1. Загальна архітектура системи доставки їжі (рис. 2.1, ст. 28)
2. Component Diagram мікросервісної архітектури (рис. 2.2, ст. 30)
3. Sequence Diagram взаємодії сервісів під час оформлення замовлення (рис. 2.3, ст. 32)
4. Deployment Diagram програмної системи (рис. 2.4, ст. 33)
5. ER-діаграма бази даних (рис. 2.5, ст. 35)

1. ***Консультанти розділів проекту (роботи)***

Розділ	***Консультант***	***Підпис, дата***	
		Завдання видав	***Завдання прийняв***

2. ***Дата видачі завдання*** _____

Керівник _____ _____
(підпис) (розшифровка підпису)

Завдання прийняв до виконання _____ _____
(підпис) (розшифровка підпису)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	***Назва етапів бакалаврського проєкту (роботи)***	***Строк виконання етапів проєкту***	***Примітка***
1	Вибір теми бакалаврського проєкту, аналіз предметної області сервісів доставки їжі, дослідження існуючих платформ та формування вимог до програмної системи	до 28.02.2026	Виконано
2	Аналіз ринку сервісів доставки їжі, огляд існуючих платформ, вибір мікросервісної архітектури та технологічного стеку	до 15.03.2026	Виконано
3	Реалізація серверної частини системи: API Gateway, сервісів користувачів, меню, замовлень, платежів та відстеження доставки, впровадження JWT-автентифікації та авторизації	до 15.04.2026	Виконано
4	Розробка клієнтського вебзастосунку на React, реалізація міжсервісної взаємодії через RabbitMQ, інтеграція Stripe, SignalR та Redis, контейнеризація системи	До 15.05.2026	Виконано
5	Проведення тестування, налаштування CI/CD-конвеєра, підготовка пояснювальної записки та оформлення бакалаврської роботи	до 09.06.2026	Виконано

Студент – дипломник _____ _____
(підпис) (розшифровка підпису)

Керівник роботи _____ _____
(підпис) (розшифровка підпису)

АНОТАЦІЯ

Бакалаврська робота містить 98 сторінки, 24 рисунків, список використаних джерел із 17 найменувань, 1 додаток.

Метою роботи є розробка онлайн платформи замовлення та доставки їжі на основі мікросервісної архітектури.

Об'єкт дослідження: є процеси організації онлайн замовлення та доставки їжі, спрямовані на забезпечення ефективної взаємодії між клієнтами, закладами харчування та кур'єрами.

Предмет дослідження: є методи, технології та інструменти розробки вебсистеми для доставки їжі, включаючи аналіз предметної області, проєктування архітектури системи, структури баз даних, реалізацію міжсервісної взаємодії, онлайн-платежів і відстеження доставки.

Результати дослідження: створено онлайн платформу доставки їжі з використанням ASP.NET Core та React на основі мікросервісної архітектури.

В першому розділі проаналізовано платформи доставки їжі та порівняно монолітну та мікросервісну архітектури.

В другому розділі сформовано функціональні та нефункціональні вимоги до системи й виконано постановку задачі розробки.

В третьому розділі описано проєктування архітектури системи, взаємодії між сервісами та баз даних.

В четвертому розділі розглянуто практичну реалізацію API Gateway, сервісів користувачів, меню, замовлень, платежів і відстеження доставки, фронтенду та міжсервісної взаємодії.

В п’ятому розділі описано тестування системи та аналіз результатів.

Висновок: розробку онлайн платформи доставки їжі успішно реалізовано, забезпечуючи відповідність сучасним вимогам до інформаційних систем.

КЛЮЧОВІ СЛОВА: ДОСТАВКА ЇЖІ, МІКРОСЕРВІСНА АРХІТЕКТУРА, ASP.NET CORE, REACT, REST API, RABBITMQ, REDIS, STRIPE, JWT-АВТЕНТИФІКАЦІЯ, CI/CD.

ABSTRACT

The bachelor's thesis consists of 98 pages, 24 figures, a list of references containing 17 sources, and 1 appendix.

The purpose of the work s to develop an online food ordering and delivery platform based on a microservices architecture.

Object of the study: the processes of organizing online food ordering and delivery aimed at ensuring effective interaction between customers, restaurants, and couriers.

Subject of the study: the methods, technologies, and tools for developing a web system for food delivery, including domain analysis, design of the system architecture and database structure, and implementation of inter-service communication, online payments, and delivery tracking.

Research results: an online food delivery platform has been developed using ASP.NET Core and React based on a microservices architecture.

The first chapter analyzes food delivery platforms, compares monolithic and microservices architectures, and defines system requirements.

The second chapter defines the functional and non-functional requirements for the system and formulates the development task.

The third chapter describes the design of the system architecture, inter-service interaction, and databases.

The fourth chapter examines the practical implementation of the API Gateway, the user, menu, order, payment, and delivery tracking services, the frontend, and inter-service communication.

The fifth chapter describes the system testing and the analysis of the results.

Conclusion: the development of the online food delivery platform has been successfully implemented, ensuring compliance with modern requirements for distributed information systems and high software quality.

KEYWORDS: FOOD DELIVERY, MICROSERVICES ARCHITECTURE, ASP.NET CORE, REACT, REST API, RABBITMQ, REDIS, STRIPE, JWT AUTHENTICATION, CI/CD.

ЗМІСТ

ВСТУП

У сучасному світі стрімкого розвитку інформаційних технологій спостерігається стрімке зростання популярності сервісів онлайн-замовлення та доставки їжі [2]. Користувачі очікують від сервісів доставки можливості швидко переглядати меню різних закладів харчування, оформлювати замовлення та здійснювати онлайн-транзакції та відстежувати статус та місцерозташування замовлення в реальному часу. Водночас для власників закладів харчування та кур'єрських служб важливо автоматизовувати бізнес-процеси, мати централізоване управління стравами та замовленнями, і отримання аналітичної інформації щодо діяльності закладу. У зв'язку зі зростанням кількості користувачів та навантаження на системи доставки виникає необхідність створення масштабованих, надійних та високопродуктивних програмних рішень.

Важливість і сучасність теми полягає у необхідності розробки надійної онлайн платформи для доставки їжі, яка здатна стабільно працювати, підтримувати одночасну взаємодію між користувачами, а також гарантувати і забезпечити безпечні і надійні виконання онлайн-платежів та передачу даних. Мікросервісна архітектура дозволяє підвищити маштабованість, ізольованість та стійкість програмної системи на відміну від монолітного рішення

Метою бакалаврської роботи є розробка онлайн платформи замовлення та доставки їжі на основі мікросервісної архітектури, яка забезпечує ефективну взаємодію між клієнтами, закладами харчування та кур'єрами, підтримує онлайн-платежі, відстеження доставки в режимі реального часу та можливість масштабування системи.

Для досягнення поставленої цілі необхідно виконати ряд завдань:

- провести аналіз сучасних платформ доставки їжі та існуючих підходів до їх реалізації;
- дослідити особливості використання мікросервісної архітектури для побудови масштабованих вебсистем;
- виконати проєктування архітектури програмної системи та взаємодії між сервісами;

- розробити API Gateway та набір мікросервісів для обробки користувачів, меню, замовлень, платежів і відстеження доставки;
- реалізувати систему автентифікації та авторизації користувачів на основі JWT;
- реалізувати механізми асинхронної взаємодії між сервісами за допомогою RabbitMQ;
- реалізувати онлайн-платежі з використанням Stripe API;
- реалізувати систему відстеження кур’єра в режимі реального часу за допомогою SignalR;
- забезпечити контейнеризацію та автоматизацію розгортання програмної системи;
- провести тестування функціональності та оцінити ефективність роботи розробленої системи.

Для реалізації програмного забезпечення використано технології .NET 10, ASP.NET Core Web API, React, PostgreSQL, Redis, RabbitMQ, SignalR, Docker та Stripe API. Архітектура системи побудована на основі мікросервісного підходу, що забезпечує незалежність, ізольованість сервісів, можливість горизонтального масштабування та підвищення стабільності і стійкості системи.

Результатом даної роботи полягає у реалізації працюючої платформи доставки їжі, яка може бути використана для автоматизації роботи сервісів доставки, закладів харчування та кур’єрських служб. Розроблена система підтримує сучасні механізми онлайн-оплати, кешування даних, оновлення в реальному часі та автоматизованого розгортання.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ ДЛЯ СЕРВІСУ ДОСТАВКИ ЇЖІ

1.1 Аналіз сучасних платформ доставки їжі

Онлайн-сервіси доставки їжі стали невід’ємною частиною сучасного світу. Використання мобільних додатків та веб-сайтів дозволило користувачам переглядати меню різних закладів, швидко оформляти замовлення та проводити безпечну онлайн-оплату без необхідності відвідувати заклади харчування. Такий підхід додає зручності користувачам і дозволяє розширити дохід бізнесів без відкриття додаткових закладів харчування.

Стандартна система доставки їжі об’єднує різноманітний функціонал для різних груп користувачів. Клієнти здійснюють пошук закладів та оформлення замовлень, ресторани формують меню та обробляють отримані заявки, а кур'єри відповідають за доставку замовлень кінцевим споживачам. Коректна взаємодія цих користувачів між собою забезпечується онлайн-платформою. Зокрема дана платформа забезпечує обробку замовлень, передачу інформації між сторонами та контроль стану доставки.

Значної популярності набули спеціалізовані платформи доставки їжі, які надають широкий набір функціоналу, включаючи онлайн-оплату, відстеження місцезнаходження кур'єра, управління меню закладів та аналітику діяльності партнерів. Найбільш відомими представниками таких систем є Uber Eats, Glovo, Bolt Food та DoorDash.

Платформа Uber Eats реалізує централізовану систему замовлення їжі, інтегровану з геолокаційними сервісами та системою онлайн-платежів [14]. Основною перевагою даного сервісу є швидке визначення найближчих закладів харчування та оптимізація маршрутів доставки. Крім того, система використовує механізми реального часу для відстеження переміщення кур’єра та оновлення статусу замовлення [13].

Сервіс Glovo орієнтований не лише на доставку їжі, а й на доставку різноманітних товарів. Платформа забезпечує інтеграцію між клієнтами, кур’єрами

та бізнесами через мобільний застосунок і використовує сучасні механізми маршрутизації та геолокації [9]. Однією з ключових особливостей системи є підтримка великої кількості одночасних замовлень та гнучке масштабування інфраструктури [2].

Платформа Bolt Food реалізує функціональність онлайн-замовлень, управління меню, безготівкової оплати та відстеження доставки в реальному часі [13]. Особливістю сервісу є оптимізована система роботи кур'єрів та інтеграція з картографічними сервісами для побудови маршрутів [9].

DoorDash є однією з найбільших платформ доставки їжі у США. Система підтримує рекомендаційні алгоритми, аналітику для ресторанів та механізми персоналізації користувацького досвіду. Крім базового функціоналу доставки, сервіс надає інструменти бізнес-аналітики для партнерських закладів [5].

Аналіз сучасних платформ показує, що більшість із них мають спільний набір функціональних можливостей:

- реєстрація та автентифікація користувачів;
- управління ролями користувачів (клієнт, бізнес, кур'єр);
- перегляд меню та інформації про заклади;
- оформлення замовлень;
- онлайн-оплата банківськими картками;
- система сповіщень та оновлення статусів;
- відстеження доставки в режимі реального часу;

Водночас сучасні платформи здебільшого мають ряд проблем і обмежень. Основними серед них є складність масштабування монолітних систем, високе навантаження на серверну інфраструктуру, необхідність забезпечення стабільності та безперервної роботи сервісів, а також складність інтеграції з зовнішніми платіжними й картографічними системами.

Для вирішення зазначених проблем у сучасних системах дедалі частіше застосовується мікросервісна архітектура. Такий підхід дозволяє розділити систему на незалежні сервіси, кожен з яких виконує окрему функцію та може масштабуватися незалежно від інших компонентів.

На основі проведеного аналізу можна зробити висновок, що сучасна онлайн платформа доставки їжі повинна забезпечувати високу продуктивність, масштабованість, підтримку реального часу, безпечну обробку платежів та зручний інтерфейс користувача. Саме тому для реалізації бакалаврської роботи обрано мікросервісну архітектуру із використанням сучасних технологій веб-розробки, асинхронної взаємодії сервісів та контейнеризації.

1.2 Аналіз функціональних можливостей існуючих систем.

Сучасні онлайн платформи доставки їжі поєднують у собі функції електронної комерції, логістики, систем онлайн-платежів та сервісів геолокації. Основною метою таких систем є забезпечення швидкої та зручної взаємодії між клієнтами, закладами харчування та кур’єрами. З огляду на високу конкуренцію у сфері доставки їжі, сучасні сервіси постійно розширюють набір функціональних можливостей та вдосконалюють користувацький досвід.

Однією з базових функцій більшості платформ є система реєстрації та автентифікації користувачів. Користувачі можуть створювати облікові записи, виконувати авторизацію та керувати персональними даними. У більшості сучасних сервісів також реалізовано підтримку декількох ролей користувачів, серед яких клієнт, представник закладу харчування та кур’єр. Такий підхід дозволяє розмежувати доступ до функціональних можливостей системи та забезпечити окремі інтерфейси для різних категорій користувачів [15].

Важливим компонентом платформ доставки їжі є система управління меню ресторанів. Користувачі мають можливість переглядати список доступних закладів, категорії страв, склад продуктів, ціни, час приготування та рейтинги. Для закладів харчування передбачено функціонал створення, редагування та видалення страв, завантаження фотографій і зміни статусу доступності товарів. Аналогічні можливості реалізовані у сервісах Uber Eats та DoorDash, де ресторани можуть самостійно керувати власним меню через адміністративні панелі [2].

Більшість сучасних платформ підтримують систему оформлення замовлень у декілька етапів. Після вибору товарів користувач формує кошик, вказує адресу

доставки та обирає спосіб оплати. Система автоматично розраховує вартість доставки, загальну суму замовлення та орієнтовний час прибуття кур'єра. Подібний функціонал дозволяє значно спростити процес взаємодії користувача із сервісом та мінімізувати кількість помилок під час оформлення замовлення [2].

Окрему роль у роботі платформ доставки їжі відіграють системи онлайн-платежів. Сучасні сервіси інтегрують платіжні шлюзи для підтримки оплати банківськими картками, цифровими гаманцями та іншими електронними методами. Наприклад, платформи Uber Eats і Glovo використовують механізми безпечної обробки платежів із підтримкою підтвердження транзакцій та повернення коштів у разі скасування замовлення [14]. Реалізація подібного функціоналу вимагає використання захищених каналів передачі даних та відповідності стандартам безпеки.

Однією з ключових особливостей сучасних систем доставки є підтримка відстеження замовлення в режимі реального часу. Після підтвердження замовлення користувач може спостерігати за зміною статусів доставки та переміщенням кур'єра на карті. Для реалізації такого функціоналу використовуються технології геолокації, WebSocket-з'єднання та сервіси побудови маршрутів. Це дозволяє підвищити прозорість роботи сервісу та покращити взаємодію між користувачем і кур'єром [13].

Сучасні платформи також містять аналітичні та адміністративні інструменти. Для представників бізнесу реалізовуються панелі керування, які дозволяють переглядати статистику продажів, популярність страв, кількість замовлень та фінансові показники. Деякі сервіси використовують алгоритми персоналізації та рекомендацій для формування індивідуальних пропозицій користувачам на основі історії замовлень [5].

Разом із великою кількістю функціональних можливостей існуючі системи мають певні недоліки. Значне навантаження на серверну інфраструктуру, потреба у підтримці великої кількості одночасних користувачів та складність масштабування монолітних систем створюють проблеми для подальшого розвитку платформ. Крім того, інтеграція із зовнішніми сервісами платежів, картографії та повідомлень ускладнює підтримку та оновлення програмного забезпечення.

Проведений аналіз показує, що сучасна платформа доставки їжі повинна підтримувати модульну архітектуру, забезпечувати високу швидкодію, масштабованість, безпечну обробку платежів та стабільну роботу механізмів реального часу. Саме ці особливості були враховані під час проєктування та реалізації програмної системи у межах даної бакалаврської роботи.

1.3 Аналіз архітектурних підходів та технологій реалізації онлайн платформи

Під час розробки сучасних вебсистем одним із ключових етапів є вибір архітектурного підходу. Від архітектури програмного забезпечення залежить масштабованість системи, складність підтримки, продуктивність, швидкість розробки та можливість подальшого розширення функціоналу. Для платформ онлайн замовлення та доставки їжі найбільш поширеними підходами є монолітна та мікросервісна архітектури.

Монолітна архітектура передбачає створення програмної системи як єдиного застосунку, у межах якого всі модулі працюють спільно та використовують одну базу даних. У такій системі функціональність користувачів, замовлень, платежів, меню та доставки реалізується в одному програмному середовищі. Подібний підхід тривалий час залишався основним способом побудови вебзастосунків через простоту реалізації та розгортання [4].

Основною перевагою монолітної архітектури є відносно невелика складність початкової розробки. Оскільки всі компоненти системи знаходяться в одному проєкті, процес налагодження, тестування та локального запуску є простішим порівняно з розподіленими системами. Крім того, монолітні системи не потребують складної міжсервісної взаємодії та окремої інфраструктури для передачі повідомлень [4].

Проте зі збільшенням функціональності та кількості користувачів монолітний підхід починає створювати значні труднощі. Будь-яка зміна або оновлення окремого модуля вимагає повторного розгортання всього застосунку. Через тісну залежність компонентів зростає ризик виникнення помилок у різних

частинах системи. Також масштабування монолітного застосунку виконується лише повністю, навіть якщо підвищене навантаження виникає лише на окремий функціональний модуль, наприклад систему обробки платежів або сервіс доставки [2].

Мікросервісна архітектура базується на принципі розділення системи на незалежні сервіси, кожен із яких виконує окрему бізнес-функцію та може розроблятися, розгортатися й масштабуватися незалежно від інших компонентів. У системах доставки їжі зазвичай окремо виділяються сервіси користувачів, меню, замовлень, платежів та відстеження доставки [2].

Однією з головних переваг мікросервісного підходу є можливість незалежного масштабування компонентів системи. Наприклад, у години пікового навантаження можна збільшити кількість екземплярів сервісу замовлень без необхідності масштабування всієї системи. Це дозволяє ефективніше використовувати серверні ресурси та підвищувати продуктивність програмного забезпечення [2].

Ще однією важливою перевагою є підвищення відмовостійкості. У випадку помилки або недоступності одного сервісу інші компоненти системи можуть продовжувати працювати. Наприклад, тимчасова недоступність сервісу аналітики не призведе до повного припинення роботи системи оформлення замовлень. Крім того, мікросервісний підхід спрощує підтримку великих команд розробників, оскільки кожна команда може працювати над окремим сервісом незалежно від інших [2].

Разом із перевагами мікросервісна архітектура має і певні недоліки. Побудова розподіленої системи потребує складнішої інфраструктури, налаштування міжсервісної взаємодії, балансування навантаження та механізмів моніторингу. Додатково виникає необхідність забезпечення узгодженості даних між сервісами та реалізації механізмів асинхронного обміну повідомленнями [11].

Для взаємодії між мікросервісами зазвичай використовуються REST API, брокери повідомлень та системи асинхронної передачі подій. У сучасних системах доставки їжі часто застосовуються RabbitMQ або Apache Kafka для реалізації подій

замовлень, оновлення статусів доставки та синхронізації платежів. Це дозволяє зменшити залежність між сервісами та підвищити стабільність системи [11].

Порівняння монолітної та мікросервісної архітектур наведено у таблиці 1.1.

Таблиця 1.1.

Порівняння монолітної та мікросервісної архітектур

Критерій	**Монолітна архітектура**	**Мікросервісна архітектура**
Структура системи	Єдиний застосунок	Набір незалежних сервісів
Масштабування	Повне масштабування системи	Масштабування окремих сервісів
Розгортання	Одночасне для всієї системи	Незалежне для кожного сервісу
Відмовостійкість	Низька	Вища
Складність розробки	Нижча на початковому етапі	Вища
Підтримка великих проєктів	Ускладнюється з ростом системи	Більш зручна
Продуктивність взаємодії	Вища всередині процесу	Можливі мережеві затримки
Гнучкість технологій	Обмежена	Можливе використання різних технологій

Для розробки онлайн платформи замовлення та доставки їжі було обрано саме мікросервісну архітектуру. Такий підхід дозволяє забезпечити незалежність окремих компонентів системи, підвищити масштабованість та спростити подальший розвиток програмного забезпечення. Крім того, використання окремих сервісів для роботи з користувачами, замовленнями, платежами та відстеженням доставки відповідає сучасним тенденціям побудови високонавантажених вебсистем.

Розробка сучасної онлайн платформи замовлення та доставки їжі потребує використання технологій, які забезпечують високу продуктивність, масштабованість, безпечну обробку даних та підтримку великої кількості одночасних користувачів. Вибір програмних засобів та архітектурних рішень безпосередньо впливає на стабільність роботи системи, швидкість розробки та можливість подальшого розширення функціональності. Крім того, сучасні програмні рішення повинні підтримувати інтеграцію із зовнішніми сервісами, забезпечувати надійний захист персональних даних користувачів та відповідати

вимогам щодо безперервності роботи системи навіть за умов значного навантаження.

Для реалізації серверної частини програмної системи доцільно використовувати платформу .NET та фреймворк ASP.NET Core Web API. Дана технологія забезпечує високу продуктивність, підтримує побудову REST API та дозволяє ефективно реалізовувати мікросервісну архітектуру. ASP.NET Core підтримує кросплатформеність, асинхронну обробку запитів та інтеграцію із сучасними засобами контейнеризації та хмарного розгортання [6]. Завдяки оптимізованому механізму обробки HTTP-запитів і підтримці сучасних протоколів передачі даних дана платформа широко використовується для створення корпоративних вебсистем та високонавантажених сервісів.

Важливою перевагою ASP.NET Core є вбудована підтримка механізмів Dependency Injection, конфігурації middleware та JWT-автентифікації. Це дозволяє реалізовувати безпечні та масштабовані вебсистеми з розмежуванням доступу для різних категорій користувачів. У розробленому проєкті платформа .NET використовується для створення API Gateway та окремих мікросервісів користувачів, замовлень, меню, платежів і відстеження доставки. Додатково використання ASP.NET Core спрощує реалізацію журналювання подій, централізованої обробки винятків та моніторингу роботи сервісів, що позитивно впливає на підтримку та супровід програмного забезпечення протягом його життєвого циклу.

Архітектура системи побудована за принципами мікросервісного підходу, відповідно до якого кожен сервіс відповідає за окрему бізнес-функцію. Такий підхід дозволяє незалежно розробляти, тестувати та розгортати окремі компоненти системи. Крім того, у випадку збільшення навантаження можливо масштабувати лише ті сервіси, які найбільше використовуються користувачами, що забезпечує ефективніше використання обчислювальних ресурсів та спрощує супровід програмного забезпечення.

Для реалізації клієнтської частини системи обрано бібліотеку React. Даний інструмент використовується для побудови динамічних користувацьких інтерфейсів та підтримує компонентний підхід до розробки. React дозволяє

створювати повторно використовувані елементи інтерфейсу, ефективно оновлювати стан сторінок та забезпечувати швидку взаємодію користувача із системою [7]. Використання компонентного підходу значно спрощує розробку великих програмних систем, оскільки окремі елементи інтерфейсу можуть використовуватись у різних частинах застосунку без дублювання коду.

Однією з переваг React є використання Virtual DOM, що зменшує кількість безпосередніх змін у структурі сторінки та покращує продуктивність вебзастосунку. У межах розробленої платформи React використовується для реалізації інтерфейсів клієнтів, закладів харчування та кур'єрів, а також для відображення інформації про замовлення, меню та статус доставки. Додатково використання React дозволяє реалізувати адаптивний інтерфейс користувача, який коректно працює як на персональних комп'ютерах, так і на мобільних пристроях.

Для організації клієнтської логіки застосовуються сучасні підходи до управління станом застосунку, маршрутизації та взаємодії з серверними API. Це забезпечує швидке оновлення інформації без повного перезавантаження сторінок та покращує користувацький досвід під час роботи із системою.

Для зберігання даних у системі використовується PostgreSQL. Дана система керування базами даних є однією з найбільш популярних реляційних СКБД із відкритим кодом та забезпечує підтримку складних SQL-запитів, транзакцій і механізмів забезпечення цілісності даних [8]. PostgreSQL добре підходить для побудови високонавантажених систем, оскільки підтримує індексацію, реплікацію та масштабування баз даних. Додатковою перевагою є підтримка сучасних механізмів резервного копіювання та відновлення даних, що підвищує надійність зберігання інформації.

У розробленій платформі PostgreSQL використовується для зберігання інформації про користувачів, заклади харчування, меню, замовлення та платежі. Для взаємодії з базою даних у проєкті використовується Entity Framework Core, який реалізує об'єктно-реляційне відображення та спрощує роботу з даними [21]. Використання ORM-підходу дозволяє зменшити кількість ручного написання SQL-запитів, підвищити читабельність коду та прискорити процес розробки програмного забезпечення.

Для реалізації механізмів кешування використовується Redis. Даний інструмент дозволяє тимчасово зберігати часто використовувані дані у оперативній пам’яті, що суттєво зменшує навантаження на базу даних та підвищує швидкість відповіді системи [10]. У системах доставки їжі кешування особливо важливе для зберігання інформації про меню, популярні заклади та активні сесії користувачів. Завдяки використанню Redis зменшується кількість звернень до основної бази даних, що позитивно впливає на загальну продуктивність системи.

Одним із ключових компонентів мікросервісної архітектури є система асинхронного обміну повідомленнями. Для реалізації такої взаємодії у проєкті використовується RabbitMQ. Даний брокер повідомлень забезпечує передачу подій між сервісами та дозволяє зменшити залежність компонентів системи [11]. Використання черг повідомлень дозволяє гарантувати доставку важливих бізнес-подій навіть у випадку тимчасової недоступності окремих сервісів.

Використання RabbitMQ дає можливість реалізувати обробку подій створення замовлень, оновлення статусів доставки та синхронізацію платежів без прямого виклику сервісів між собою. Це підвищує відмовостійкість системи та дозволяє окремим сервісам працювати незалежно навіть при високому навантаженні. Такий підхід також спрощує подальше розширення функціональності системи шляхом додавання нових сервісів без необхідності внесення значних змін до вже існуючих компонентів.

Для реалізації функціоналу реального часу використовується SignalR. Технологія забезпечує двосторонній обмін даними між сервером і клієнтом без необхідності постійного надсилання HTTP-запитів [13]. У платформі доставки їжі SignalR використовується для оновлення статусів замовлення та відображення переміщення кур’єра на карті в режимі реального часу. Завдяки цьому користувачі можуть оперативно отримувати інформацію про поточний стан доставки та орієнтовний час прибуття замовлення.

Важливим компонентом системи є підтримка онлайн-платежів. Для цього у проєкті використовується Stripe API. Дана платіжна платформа забезпечує безпечну обробку банківських транзакцій, підтримує інтеграцію з вебзастосунками та дозволяє виконувати підтвердження платежів і повернення коштів [14]. Stripe

надає механізми захисту фінансових операцій та відповідає сучасним міжнародним стандартам безпеки платіжних систем, що робить її надійним рішенням для електронної комерції.

Для контейнеризації програмного забезпечення використовується Docker. Дана технологія дозволяє ізолювати окремі сервіси системи у контейнерах та забезпечує однакове середовище виконання незалежно від операційної системи або серверної інфраструктури [12]. Контейнеризація спрощує процес розгортання, тестування та масштабування програмної системи.

У межах проєкту також використовуються засоби автоматизації CI/CD, що дозволяють автоматично виконувати збірку, тестування та розгортання програмного забезпечення після внесення змін у репозиторій. Це значно спрощує підтримку мікросервісної системи та прискорює процес оновлення застосунку. Автоматизація процесів розробки дозволяє підвищити якість програмного забезпечення, скоротити час випуску нових версій та мінімізувати вплив людського фактора під час розгортання системи.

Окрему увагу під час розробки системи приділено питанням інформаційної безпеки. Для захисту персональних даних користувачів використовуються механізми шифрування переданих даних за допомогою протоколу HTTPS, рольова модель доступу та перевірка автентичності користувачів на основі JWT-токенів. Такий підхід дозволяє забезпечити конфіденційність інформації та запобігти несанкціонованому доступу до ресурсів системи.

Отже, проведений аналіз показує, що використання сучасних технологій .NET, React, PostgreSQL, Redis, RabbitMQ, SignalR, Stripe API та Docker дозволяє створити масштабовану, надійну та високопродуктивну платформу онлайн замовлення та доставки їжі. Обраний стек технологій забезпечує підтримку мікросервісної архітектури, асинхронної взаємодії компонентів та функціоналу реального часу, що є важливими вимогами для сучасних вебсистем. Комплексне використання зазначених технологій створює надійну основу для подальшого розвитку програмного продукту, розширення його функціональних можливостей та забезпечення стабільної роботи в умовах зростання кількості користувачів і транзакцій.

1.4 Висновки по розділу

У першому розділі проведено аналіз предметної області та існуючих рішень у сфері онлайн замовлення та доставки їжі, що дозволило обґрунтувати вибір архітектурного підходу й технологій для реалізації програмної системи.

У результаті аналізу сучасних платформ доставки їжі, зокрема Uber Eats, Glovo, Bolt Food та DoorDash, визначено типовий набір функціональних можливостей таких систем - реєстрацію та автентифікацію користувачів, розмежування ролей клієнта, закладу й кур'єра, перегляд меню, оформлення замовлень, онлайн-оплату, сповіщення про зміну статусів та відстеження доставки в режимі реального часу. Водночас виявлено характерні для існуючих рішень обмеження, пов'язані зі складністю масштабування, високим навантаженням на серверну інфраструктуру та залежністю від зовнішніх платіжних і картографічних сервісів.

На основі порівняння монолітної та мікросервісної архітектур обґрунтовано доцільність застосування мікросервісного підходу, який забезпечує незалежне масштабування окремих компонентів, підвищену відмовостійкість, гнучкість розробки та зручність подальшого супроводу системи. Проаналізовано сучасні технології реалізації онлайн платформи та обрано технологічний стек, що включає ASP.NET Core для побудови серверних сервісів, React для клієнтської частини, PostgreSQL як основне сховище даних, Redis для кешування, RabbitMQ для асинхронної взаємодії сервісів, SignalR для роботи в режимі реального часу, а також зовнішні сервіси Stripe, Cloudinary та сервіси геолокації.

Отримані результати аналізу створюють основу для формування вимог до програмної системи та постановки задачі розробки, що детально розглядаються в наступному розділі.

Крім того, за результатами дослідження визначено основні вимоги до майбутньої системи щодо продуктивності, надійності, безпеки та зручності використання. Сформовані вимоги та обрані технологічні рішення стали підґрунтям для подальшого проєктування архітектури програмного забезпечення та реалізації функціональних компонентів платформи доставки їжі.

РОЗДІЛ 2. АНАЛІЗ ВИМОГ ТА ПОСТАНОВКА ЗАДАЧІ ПРОЄКТУВАННЯ

Якісне проєктування та реалізація програмної системи неможливі без чіткого визначення вимог до неї. Вимоги формують основу для прийняття архітектурних рішень, вибору технологій, планування розробки та подальшого тестування створеного програмного продукту. На основі проведеного у першому розділі аналізу предметної області, існуючих платформ доставки їжі та сучасних технологій у цьому розділі сформовано функціональні й нефункціональні вимоги до системи, а також виконано постановку задачі розробки. Вимоги поділено на функціональні, що описують безпосередню поведінку та можливості системи, і нефункціональні, що визначають її якісні характеристики й умови експлуатації.

2.1 Функціональні вимоги до сервісу доставки їжі

Функціональні вимоги визначають перелік функцій, які повинна виконувати система, та описують її поведінку у відповідь на дії користувачів. Формування вимог виконувалося на основі аналізу сучасних платформ доставки їжі, дослідження предметної області, типових сценаріїв використання та особливостей реалізованого проєкту. Перед формулюванням конкретних вимог було визначено основні категорії користувачів системи та їхні ролі, оскільки кожна категорія взаємодіє із системою по-різному та потребує власного набору функціональних можливостей.

Система передбачає чотири основні категорії користувачів - клієнтів, представників закладів харчування, кур'єрів та адміністраторів. Розмежування доступу до функцій реалізується на основі ролей користувачів, що визначаються під час автентифікації та кодуються у токені доступу [15]. Опис ролей користувачів системи наведено нижче:

Клієнт - користувач, який здійснює замовлення їжі. Він переглядає заклади харчування та їхнє меню, формує кошик, оформлює та оплачує замовлення, відстежує його стан і місцеположення кур'єра в режимі реального часу, а також переглядає історію власних замовлень.

Представник закладу харчування (бізнес-користувач) - користувач, який керує меню закладу та опрацьовує замовлення. Він створює, редагує та видаляє страви й категорії, керує доступністю позицій, переглядає чергу активних замовлень, змінює їхні статуси та має доступ до аналітичної інформації про діяльність закладу.

Кур'єр - користувач, який виконує доставку замовлень. Він переглядає перелік доступних замовлень, приймає їх до виконання, змінює статус доставки та передає координати свого місцеположення для відображення на карті клієнтові.

На основі визначених ролей сформовано функціональні вимоги, згруповані за окремими функціональними підсистемами програмного забезпечення. Такий поділ відповідає мікросервісній організації системи, у якій кожна підсистема реалізується окремим сервісом зі своєю зоною відповідальності.

Підсистема автентифікації та керування користувачами

Система повинна забезпечувати реєстрацію та автентифікацію користувачів. Користувачі повинні мати можливість створювати облікові записи, виконувати вхід у систему, завершувати сеанс роботи та відновлювати доступ до акаунта. Для забезпечення безпеки доступу необхідно реалізувати механізм автентифікації на основі JWT-токенів, який передбачає видачу токена доступу з обмеженим терміном дії та токена оновлення для продовження сеансу без повторного введення облікових даних [15]. Розмежування прав доступу до функцій системи здійснюється на основі ролей, закодованих у токені. Користувач повинен мати можливість редагувати свій профіль, зокрема завантажувати зображення профілю, а також за потреби перемикатися між кількома обліковими записами різних типів у межах одного користувача.

Підсистема перегляду меню та формування замовлення

Клієнтська частина системи повинна забезпечувати перегляд списку закладів харчування, категорій меню та детальної інформації про страви, включно з їхньою назвою, описом, ціною, зображенням та доступністю. Для зручності користувача необхідно реалізувати пошук і фільтрацію страв за категоріями та іншими атрибутами. Користувач повинен мати можливість формувати кошик товарів, додавати та видаляти позиції, редагувати їхню кількість, а також

оформлювати замовлення із зазначенням адреси доставки та способу оплати. Під час оформлення замовлення система повинна формувати його склад, обчислювати загальну вартість та передавати замовлення на подальшу обробку. Для забезпечення коректної роботи з великими обсягами даних перегляд переліків закладів і страв повинен підтримувати посторінкове виведення (пагінацію).

Підсистема онлайн-оплати

Однією з ключових функціональних вимог є підтримка онлайн-платежів. Система повинна забезпечувати інтеграцію із платіжним сервісом Stripe для безпечної обробки банківських транзакцій та підтвердження платежів [14]. Під час оформлення замовлення для онлайн-оплати формується платіжний намір, а підтвердження платежу здійснюється на стороні клієнта через захищену форму оплати. Окрім онлайн-оплати, система повинна підтримувати готівковий розрахунок під час доставки. Необхідно також реалізувати можливість скасування замовлення та повернення коштів - повного або часткового - у випадках, передбачених бізнес-логікою системи. Для здійснення розрахунків із закладами та виплат кур'єрам система повинна підтримувати модель під'єднаних рахунків платіжного сервісу, що дозволяє автоматично розподіляти кошти між платформою, закладом і кур'єром.

Підсистема керування меню та замовленнями з боку закладу

Для представників закладів харчування система повинна надавати повний набір інструментів управління меню та замовленнями. Бізнес-користувачі повинні мати можливість створювати, редагувати та видаляти страви й категорії, завантажувати зображення страв, змінювати їхню доступність та ціну. Система повинна надавати закладу чергу активних замовлень із можливістю перегляду їхнього складу та послідовної зміни статусів - від прийняття замовлення до позначення його готовності до видачі кур'єру. Окремою важливою вимогою є надання закладу аналітичної інформації про його діяльність, зокрема даних про виручку за стравами та за визначений період, що використовуються для прийняття управлінських рішень.

Підсистема кур'єрської доставки та відстеження

Окрему увагу необхідно приділити функціоналу кур'єрської доставки. Кур'єри повинні отримувати інформацію про доступні для виконання замовлення, приймати їх до доставки, змінювати статус доставки на відповідних етапах та передавати координати свого місцеположення для відображення на карті [13]. Система повинна підтримувати оновлення статусів замовлення та місцеположення кур'єра без перезавантаження сторінки. Для цього застосовується технологія SignalR, що забезпечує двосторонній обмін даними між сервером і клієнтом у режимі реального часу та дозволяє клієнтові отримувати інформацію про підтвердження замовлення, початок доставки, переміщення кур'єра й орієнтовний час прибуття. Для побудови маршрутів доставки та визначення відстані й часу в дорозі система повинна інтегруватися із зовнішнім сервісом маршрутизації, а для перетворення адрес у географічні координати - із сервісом геокодування.

Підсистема міжсервісної взаємодії та сповіщень

Оскільки систему реалізовано на основі мікросервісної архітектури, важливою функціональною вимогою є підтримка асинхронної взаємодії між сервісами. Для передачі подій між окремими компонентами системи необхідно використовувати брокер повідомлень RabbitMQ, який забезпечує слабко зв'язану взаємодію сервісів, зменшує їхню взаємну залежність і підвищує відмовостійкість програмного забезпечення [11]. Зокрема, події життєвого циклу замовлення повинні автоматично передаватися від сервісу замовлень до сервісів платежів і відстеження, що дозволяє узгоджено опрацьовувати замовлення без прямих синхронних викликів. На основі цих подій система повинна формувати своєчасні сповіщення для користувачів про зміну стану замовлення.

Для ілюстрації взаємодії користувача із системою розглянемо типовий сценарій оформлення замовлення. Клієнт переглядає меню обраного закладу, додає страви до кошика та переходить до оформлення замовлення, вказуючи адресу доставки й спосіб оплати. Система формує замовлення, обчислює його вартість і, у разі онлайн-оплати, створює платіжний намір та повертає клієнтові посилання для оплати. Після підтвердження платежу замовлення передається закладу, який готує його та позначає готовим до видачі. Кур'єр приймає замовлення, отримує його в

закладі та виконує доставку, передаючи свої координати, які клієнт бачить на карті в режимі реального часу. Після завершення доставки замовлення позначається виконаним.

2.2 Нефункціональні вимоги до сервісу доставки їжі

Окрім функціональних, система повинна відповідати ряду нефункціональних вимог, які визначають її якісні характеристики, умови експлуатації та обмеження. Нефункціональні вимоги безпосередньо впливають на архітектурні рішення та вибір технологій, оскільки саме вони визначають здатність системи працювати надійно, безпечно та ефективно за реальних умов експлуатації. Нижче розглянуто основні групи нефункціональних вимог до розроблюваної платформи.

Масштабованість

Однією з головних нефункціональних вимог є масштабованість. Платформа повинна підтримувати збільшення кількості користувачів та обробку великої кількості одночасних запитів без значного зниження продуктивності [2]. Для цього система повинна забезпечувати можливість горизонтального масштабування окремих сервісів, тобто збільшення кількості їхніх екземплярів залежно від навантаження. Така вимога безпосередньо зумовлює застосування мікросервісної архітектури, у якій найбільш навантажені сервіси, зокрема сервіс замовлень і сервіс відстеження доставки, можуть масштабуватися незалежно від інших компонентів. Важливою умовою масштабованості є відсутність збереження стану сеансу на стороні сервісу, що забезпечується використанням самодостатніх JWT-токенів.

Продуктивність та швидкодія

Програмна система повинна забезпечувати високу швидкодію та мінімальний час відповіді програмного інтерфейсу. Затримка під час обробки типових запитів користувачів має залишатися прийнятною навіть за умов зростання навантаження. Для оптимізації продуктивності доцільно використовувати Redis як систему кешування часто запитуваних даних [10, 25], що дозволяє зменшити навантаження на основну базу даних та прискорити обробку

популярних запитів, зокрема перегляду меню. Додатково підвищенню продуктивності сприяє посторінкове виведення великих переліків даних та асинхронна обробка тривалих операцій.

Безпека

Важливою вимогою є забезпечення безпеки даних користувачів [28]. Система повинна використовувати захищені механізми передачі інформації з шифруванням каналів зв'язку, шифрування та перевірку токенів автентифікації, а також контроль доступу до програмного інтерфейсу на основі ролей. Окрему увагу необхідно приділити захисту платіжної інформації: обробка даних банківських карток делегується сертифікованому платіжному сервісу, завдяки чому конфіденційні платіжні дані не зберігаються в системі. Персональні дані користувачів повинні оброблятися відповідно до вимог чинного законодавства про захист персональних даних. Взаємодія між внутрішніми сервісами також повинна бути захищеною та доступною лише через єдину точку входу.

Надійність та відмовостійкість

Ще однією важливою характеристикою є відмовостійкість системи [20]. У разі виникнення помилок або тимчасової недоступності окремих сервісів інші компоненти повинні продовжувати роботу. Для цього доцільно використовувати асинхронний обмін повідомленнями через брокер, кешування даних та механізми повторної обробки запитів. Надійність передачі повідомлень між сервісами повинна забезпечуватися застосуванням шаблону надійної доставки подій та механізмів запобігання повторній обробці однакових повідомлень. Система також повинна підтримувати засоби контролю працездатності сервісів, що дозволяють середовищу розгортання своєчасно виявляти й перезапускати компоненти, які перестали відповідати.

Зручність використання та сумісність

Система повинна мати інтуїтивно зрозумілий користувацький інтерфейс, який не потребує спеціального навчання та забезпечує зручну взаємодію для всіх категорій користувачів. Клієнтська частина повинна коректно відображатися в сучасних веб-браузерах та адаптуватися до різних розмірів екрана. Програмний інтерфейс системи повинен відповідати принципам REST, що забезпечує

сумісність із різними клієнтськими застосунками та спрощує подальшу інтеграцію зі сторонніми системами.

Супроводжуваність та переносимість

Система повинна бути зручною у супроводі та подальшому розвитку. Мікросервісна організація програмного забезпечення забезпечує модульність і дозволяє вносити зміни до окремих сервісів без впливу на інші компоненти. Для спрощення розгортання, оновлення та супроводу необхідно застосовувати засоби автоматизації складання й розгортання. Важливою вимогою є також кросплатформеність та можливість контейнеризації системи: для забезпечення стабільного розгортання сервісів використовується Docker, що ізолює компоненти й гарантує однакове середовище виконання незалежно від серверної інфраструктури [12].

2.3 Постановка задачі розробки сервісу доставки

З огляду на проведений аналіз предметної області та сформовані функціональні й нефункціональні вимоги, у межах роботи поставлено задачу розробки програмної системи онлайн замовлення та доставки їжі. Постановка задачі визначає мету розробки, її межі, припущення й обмеження, перелік підзадач та очікувані результати, що відображає практичну спрямованість роботи та орієнтацію на вирішення реальної інженерної проблеми.

Метою розробки є створення масштабованої веб-орієнтованої платформи для замовлення та доставки їжі, яка забезпечує комплексну взаємодію між клієнтами, закладами харчування та кур'єрами й підтримує повний життєвий цикл замовлення - від вибору страв до виконання доставки - із забезпеченням високого рівня доступності, розширюваності, безпеки та зручності використання. Система повинна бути побудована на основі мікросервісної архітектури, що відповідає визначеним вимогам щодо масштабованості та відмовостійкості.

Об'єктом розробки є процеси організації онлайн замовлення та доставки їжі, а предметом - методи, технології та інструменти побудови мікросервісної вебсистеми для автоматизації цих процесів. Межі системи охоплюють клієнтську

частину, єдину точку входу, набір незалежних серверних сервісів та їхні сховища даних, а також інтеграцію із зовнішніми сервісами оплати, зберігання зображень, маршрутизації та геокодування.

Розроблювана система повинна забезпечувати автоматизацію основних бізнес-процесів, пов'язаних із прийманням, обробкою та виконанням замовлень на доставку їжі. Важливою особливістю платформи є підтримка одночасної роботи декількох категорій користувачів, кожна з яких взаємодіє із системою через власний набір функціональних можливостей. Такий підхід дозволяє створити єдине інформаційне середовище для всіх учасників процесу доставки та забезпечити оперативний обмін даними між ними.

Особлива увага приділяється забезпеченню узгодженості даних у розподіленому середовищі, оскільки окремі компоненти системи функціонують як незалежні мікросервіси. Для цього необхідно реалізувати надійні механізми міжсервісної взаємодії, обробки подій та синхронізації інформації між підсистемами.

Припущення та обмеження

Під час постановки задачі прийнято низку припущень та обмежень. Передбачається, що користувачі мають доступ до мережі Інтернет та сучасного веб-браузера. Обробка платежів здійснюється виключно через зовнішній сертифікований платіжний сервіс, тому система не зберігає даних банківських карток. Побудова маршрутів і геокодування виконуються за допомогою зовнішніх сервісів, доступність яких є умовою коректної роботи підсистеми відстеження. Розгортання системи передбачається у хмарному середовищі з можливістю масштабування окремих сервісів.

Перелік підзадач

Для досягнення поставленої мети необхідно вирішити такі підзадачі:

- спроєктувати загальну архітектуру системи та визначити склад і зони відповідальності окремих мікросервісів;
- реалізувати єдину точку входу (API Gateway), що забезпечує маршрутизацію запитів, автентифікацію користувачів та захист взаємодії з внутрішніми сервісами;

- забезпечити реєстрацію, авторизацію, керування профілями та ролями користувачів (клієнт, заклад, кур'єр) із використанням JWT-токенів і механізму оновлення сеансів;
- реалізувати створення та керування меню закладами, а також перегляд і вибір страв користувачами з урахуванням категорій та атрибутів;
- реалізувати механізм формування кошика, створення замовлення, відстеження його стану та управління життєвим циклом замовлення - від підготовки до доставки;
- забезпечити підтримку онлайн-оплат і готівкових розрахунків, включно з інтеграцією із зовнішнім платіжним сервісом Stripe та обробкою повернень коштів;
- забезпечити передавання координат кур'єра й відображення процесу доставки в режимі реального часу засобами SignalR та кешування стану в Redis;
- реалізувати обмін повідомленнями через брокер RabbitMQ для узгодженості даних та обробки подій у розподіленому середовищі;
- спроєктувати структуру баз даних за принципом окремого сховища для кожного сервісу;
- забезпечити контейнеризацію та автоматизоване розгортання компонентів системи;
- провести тестування системи та аналіз отриманих результатів.

Очікувані результати та критерії досягнення мети

Очікуваним результатом розробки є працездатна онлайн платформа замовлення та доставки їжі, що відповідає сформованим функціональним і нефункціональним вимогам. Критеріями досягнення мети вважаються: коректна робота всіх основних бізнес-сценаріїв для кожної категорії користувачів; надійна асинхронна та синхронна взаємодія між сервісами; безпечна обробка онлайн-платежів; своєчасне оновлення інформації про доставку в режимі реального часу; а також підтвердження масштабованості й відмовостійкості системи за результатами тестування. Розв'язання поставлених підзадач забезпечить створення цілісної програмної системи, що відповідає сучасним підходам до побудови розподілених застосунків.

2.4 Висновки по розділу

У другому розділі на основі проведеного у першому розділі аналізу предметної області та існуючих рішень сформовано вимоги до програмної системи та виконано постановку задачі розробки.

Визначено основні категорії користувачів системи - клієнтів, представників закладів харчування, кур'єрів та адміністраторів - і для кожної з них окреслено набір функцій відповідно до ролі. Сформовано функціональні вимоги, згруповані за окремими підсистемами: автентифікації та керування користувачами, перегляду меню й формування замовлення, онлайн-оплати, керування меню та замовленнями з боку закладу, кур'єрської доставки й відстеження, а також міжсервісної взаємодії та сповіщень. Основні функціональні вимоги систематизовано у вигляді таблиці з визначенням їхнього пріоритету, а взаємодію користувача із системою проілюстровано типовим сценарієм оформлення замовлення.

Сформовано нефункціональні вимоги, що визначають якісні характеристики системи - масштабованість, продуктивність, безпеку, надійність і відмовостійкість, зручність використання, сумісність, супроводжуваність та переносимість. Для кожної групи вимог визначено критерії її досягнення, що систематизовано у відповідній таблиці. Саме нефункціональні вимоги значною мірою зумовлюють вибір мікросервісної архітектури та технологічного стека системи.

На основі сформованих вимог виконано постановку задачі розробки: визначено мету, об'єкт і предмет, межі системи, припущення й обмеження, перелік підзадач та очікувані результати з критеріями досягнення мети. Сформовані вимоги та поставлена задача створюють основу для подальшого проєктування архітектури системи, баз даних і взаємодії між сервісами, що детально розглядається в наступному розділі.

РОЗДІЛ 3. ПРОЄКТУВАННЯ АРХІТЕКТУРИ СЕРВІСУ ДОСТАВКИ ЇЖІ

3.1 Загальна архітектура програмної системи.

Під час розроблення сучасних вебплатформ одним із найважливіших етапів є проєктування архітектури програмного забезпечення. Саме архітектурні рішення визначають структуру системи, способи взаємодії між компонентами, можливість масштабування та рівень відмовостійкості платформи. Для реалізації системи онлайн-замовлення та доставки їжі було обрано мікросервісний підхід, який дозволяє розділити програмне забезпечення на незалежні функціональні модулі та забезпечити їх автономну роботу.

Розроблена система являє собою багаторівневу вебплатформу, що складається з клієнтської частини, API Gateway, набору мікросервісів, брокера повідомлень, сервісів кешування та баз даних. Така архітектура дозволяє забезпечити незалежне масштабування окремих компонентів, спростити супровід програмного забезпечення та зменшити зв’язність між модулями системи .

Основними компонентами програмної системи є:

- клієнтський вебзастосунок;
- API Gateway;
- UserService;
- MenuService;
- OrderService;
- PaymentService;
- TrackingService;
- RabbitMQ;
- Redis;
- PostgreSQL;
- SignalR
- зовнішні API та сервіси.

Клієнтська частина системи реалізована із використанням React та забезпечує взаємодію користувачів із платформою. Через вебінтерфейс користувачі можуть переглядати меню закладів харчування, формувати замовлення, здійснювати оплату та відстежувати процес доставки. Окремі інтерфейси передбачені для представників бізнесів та кур'єрів.

Усі HTTP-запити від клієнтського застосунку надходять до API Gateway, який виступає центральною точкою доступу до системи. Gateway забезпечує маршрутизацію запитів до відповідних мікросервісів, виконує перевірку JWT-токенів, централізовану обробку помилок, логування та контроль доступу до ресурсів системи [6].

UserService відповідає за реєстрацію, автентифікацію та авторизацію користувачів. У сервісі реалізовано підтримку ролей клієнтів, бізнесів і кур'єрів. Для захисту доступу використовується JWT-автентифікація, що дозволяє забезпечити безпечну взаємодію між клієнтом і сервером [15].

MenuService забезпечує зберігання та обробку інформації про заклади харчування і страви. Сервіс реалізує функціонал створення, редагування та видалення меню, підтримку категорій товарів і завантаження фотографій страв.

OrderService реалізує основну бізнес-логіку системи. До його функцій належать створення замовлень, керування їх статусами, обробка складу замовлення та взаємодія із сервісами платежів і доставки.

Для виконання онлайн-платежів використовується окремий PaymentService, інтегрований із Stripe API. Виділення платіжної логіки в окремий сервіс дозволяє ізолювати фінансові операції та підвищити безпеку системи [14].

TrackingService реалізує механізми відстеження доставки в режимі реального часу. Для цього використовується SignalR, який забезпечує двосторонній обмін повідомленнями між сервером і клієнтом без необхідності постійного надсилання HTTP-запитів [13].

Для організації асинхронної взаємодії між сервісами використовується RabbitMQ. Брокер повідомлень забезпечує передачу подій між компонентами системи та дозволяє реалізувати event-driven підхід [11, 26]. Для кешування часто

використовуваних даних і зменшення навантаження на базу даних використовується Redis [10].

Для постійного зберігання інформації використовується PostgreSQL. Кожен мікросервіс має власну базу даних або окрему схему, що відповідає принципам незалежності компонентів та мікросервісної архітектури [8].

Програмна система також інтегрується із зовнішніми сервісами. Stripe API використовується для обробки платежів, а геолокаційні сервіси - для побудови маршрутів доставки та визначення координат кур'єрів [30].

Для забезпечення стабільного розгортання та ізоляції сервісів використовується Docker. Контейнеризація дозволяє запускати кожен компонент системи в окремому середовищі та спрощує масштабування платформи [12].

Загальну архітектуру програмної системи наведено на рисунку 3.1.

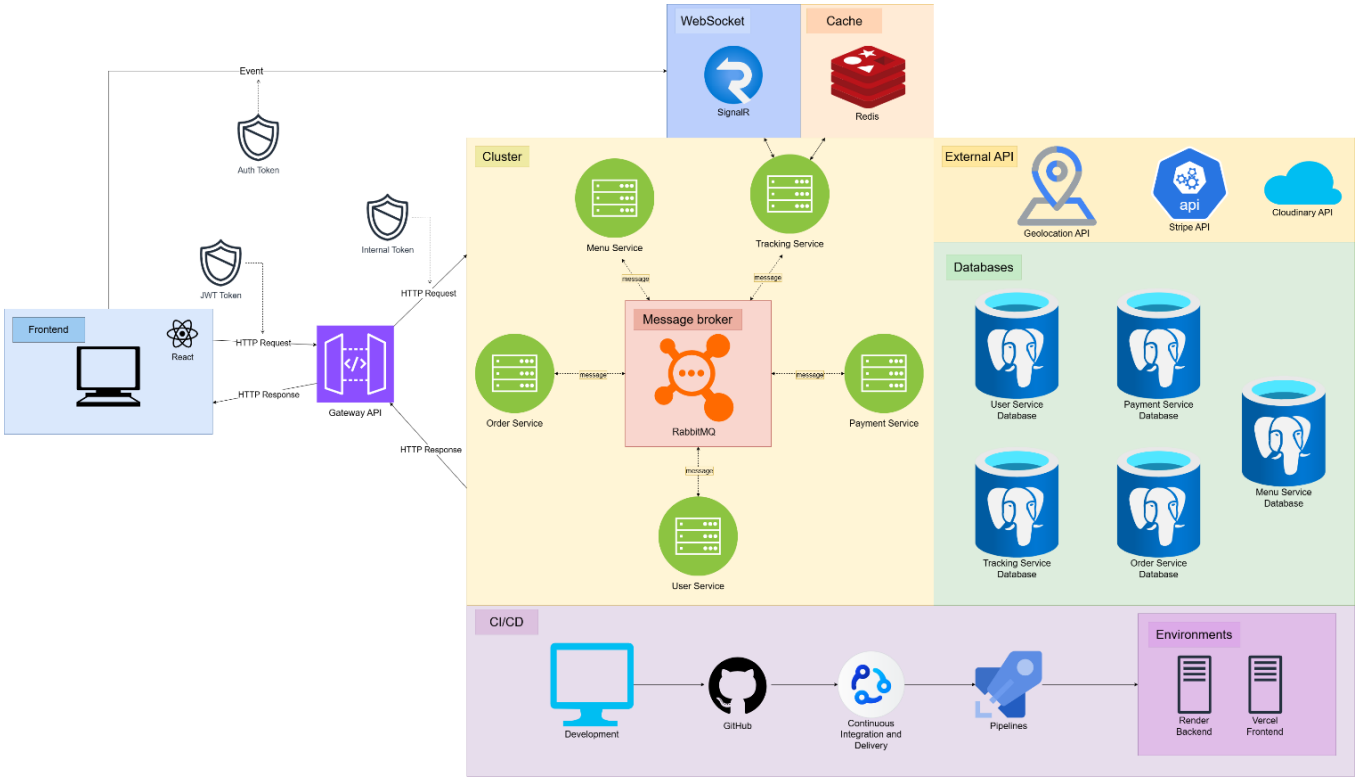


Рисунок 3.1 - Загальна архітектура програмної системи доставки їжі

3.2 Проєктування мікросервісної архітектури та взаємодії між сервісами

Проєктування мікросервісної архітектури

Проєктування мікросервісної архітектури є одним із ключових етапів створення сучасних вебплатформ, оскільки саме архітектурні рішення визначають масштабованість, продуктивність та зручність подальшого супроводу системи.

Для реалізації платформи доставки їжі було обрано мікросервісний підхід, який передбачає поділ програмної системи на незалежні сервіси, кожен з яких відповідає за окрему бізнес-область [18].

Основними сервісами системи є: UserService, MenuService, OrderService, PaymentService, TrackingService, API Gateway.

API Gateway використовується як єдина точка входу до системи та забезпечує маршрутизацію HTTP-запитів від клієнтського застосунку до внутрішніх сервісів. Крім маршрутизації, Gateway виконує централізовану автентифікацію користувачів, перевірку JWT-токенів та контроль доступу до ресурсів платформи.

UserService відповідає за керування обліковими записами користувачів, реєстрацію, автентифікацію та авторизацію. MenuService реалізує функціонал роботи із закладами харчування та меню страв. OrderService забезпечує створення та обробку замовлень, а також керування життєвим циклом доставки. PaymentService виконує обробку онлайн-платежів через Stripe API та облік фінансових операцій. TrackingService реалізує механізми геолокації та відстеження кур’єрів у режимі реального часу. Тобто ці сервіси спроектовані таким чином, що відповідає принципам розподілу доменів предметної області [19].

Для забезпечення асинхронної взаємодії між сервісами використовується RabbitMQ. Передача подій через брокер повідомлень дозволяє зменшити зв’язність між компонентами та забезпечити більш стабільну роботу системи у випадку тимчасової недоступності окремих сервісів [11].

Для підвищення продуктивності системи використовується Redis, який виконує роль кешу для часто використовуваних даних та підтримує механізми роботи в режимі реального часу. Кожен сервіс використовує власну базу даних

PostgreSQL, що відповідає принципу Database per Service та дозволяє забезпечити незалежність компонентів системи [8].

У системі також реалізовано механізми idempotency, retry policy, dead-letter queues та outbox pattern. Використання outbox pattern дозволяє гарантувати коректну передачу подій у RabbitMQ навіть у випадку тимчасових помилок або недоступності брокера повідомлень [3].

Взаємозв'язки між окремими мікросервісами, їхні функціональні залежності та способи комунікації доцільно подати у вигляді UML Component Diagram. На даній діаграмі відображаються основні сервіси системи, API Gateway, RabbitMQ та зовнішні інтеграції, а також показуються синхронні та асинхронні способи взаємодії між компонентами.

UML-діаграму компонентів системи наведено на рисунку 3.2.

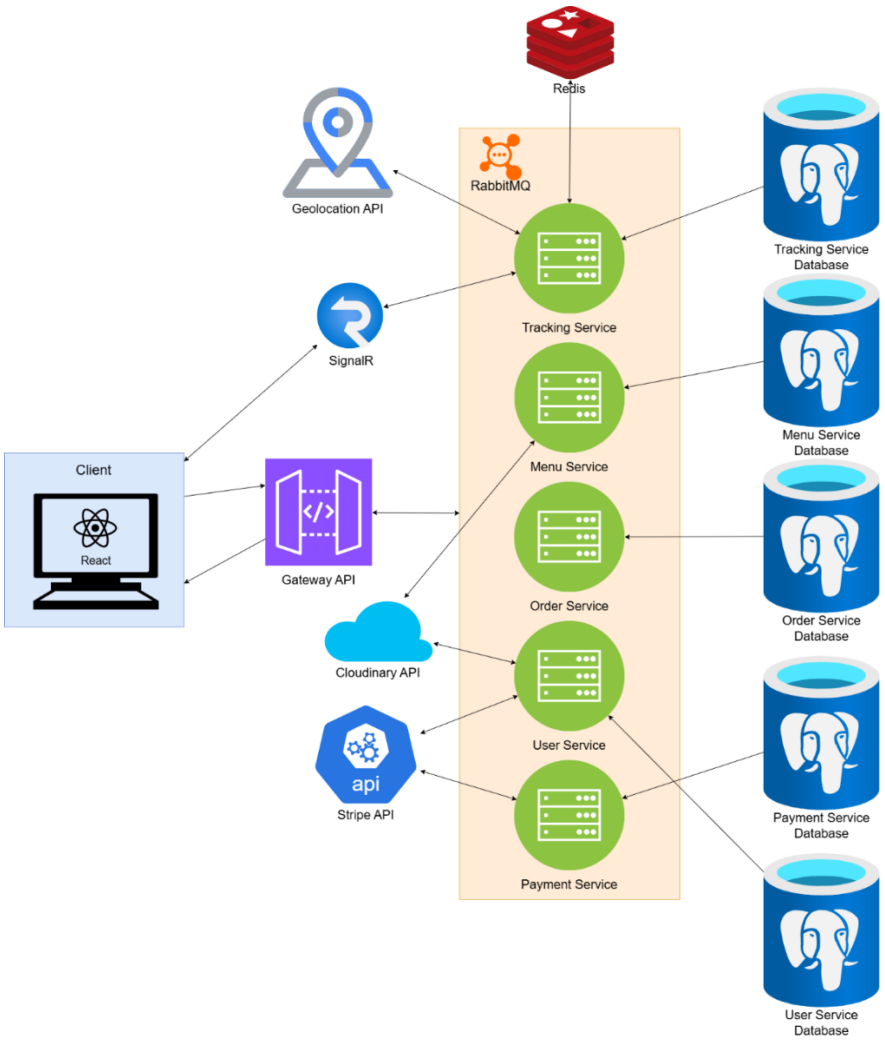


Рисунок 3.2 - UML Component Diagram мікросервісної архітектури

Проєктування взаємодії між сервісами

Під час проєктування системи доставки їжі особливу увагу було приділено організації взаємодії між окремими мікросервісами платформи. Використання мікросервісної архітектури дозволило реалізувати незалежні функціональні модулі, кожен з яких відповідає за окрему бізнес-область: керування користувачами, роботу з меню, обробку замовлень, оплату та відстеження доставки [18].

У розробленій системі взаємодія між сервісами реалізована двома основними способами:

- синхронною взаємодією через REST API;
- асинхронним обміном повідомленнями через RabbitMQ.

Синхронна взаємодія використовується переважно для обробки HTTP-запитів від клієнтського застосунку через API Gateway. Користувач надсилає запит до Gateway, після чого запит маршрутизується до відповідного мікросервісу.

Асинхронна взаємодія використовується для передачі подій між сервісами без прямої залежності між ними. Наприклад, після створення замовлення OrderService публікує подію про створення замовлення у RabbitMQ. Отримавши цю подію, PaymentService створює платіжну сесію через Stripe API. Після успішного підтвердження оплати PaymentService надсилає подію про успішний платіж, яку отримує OrderService для оновлення статусу замовлення.

Для забезпечення надійності передачі повідомлень у системі використовується Outbox Pattern. Події попередньо зберігаються у базі даних сервісу, після чого публікуються у RabbitMQ. Такий підхід дозволяє уникнути втрати повідомлень у випадку тимчасових помилок або недоступності брокера повідомлень [3].

Крім того, у системі реалізовано механізми retry policy, idempotency та dead-letter queues, що дозволяє підвищити стійкість системи до помилок та уникнути дублювання подій.

TrackingService забезпечує оновлення координат кур’єрів у режимі реального часу. Для цього використовується SignalR, який підтримує постійне двостороннє з’єднання між сервером та клієнтом. Кур’єрський застосунок періодично надсилає

координати кур’єра, а клієнти та представники закладів отримують актуальну інформацію щодо місцеположення доставки.

Для детального відображення процесу взаємодії між компонентами системи доцільно використати UML Sequence Diagram. На даній діаграмі можна продемонструвати послідовність взаємодії між клієнтським застосунком, API Gateway, OrderService, PaymentService, RabbitMQ та Stripe API під час оформлення замовлення.

UML-діаграму послідовності взаємодії сервісів наведено на рисунку 3.3.

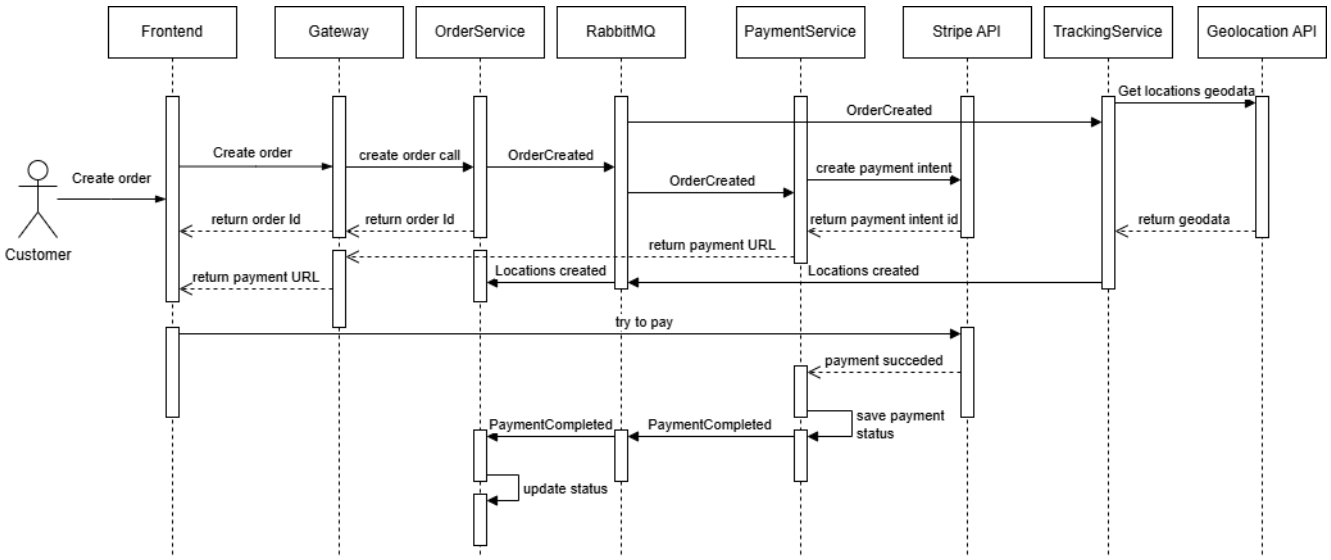


Рисунок 3.3 - UML Sequence Diagram взаємодії сервісів під час оформлення замовлення

Використання UML Deployment Diagram дає можливість відобразити фізичне розташування програмних компонентів та взаємозв’язки між ними на рівні інфраструктури. Діаграма демонструє розподіл сервісів між контейнерами, способи підключення до спільних ресурсів та взаємодію через мережеве середовище Docker. Завдяки цьому можна оцінити структуру розгортання системи, визначити залежності між компонентами та перевірити коректність побудови інфраструктури ще до фактичного розгортання програмного забезпечення. Такий підхід також спрощує подальше масштабування системи та супровід її інфраструктурної частини.

Діаграму розгортання програмної системи наведено на рисунку 3.4.

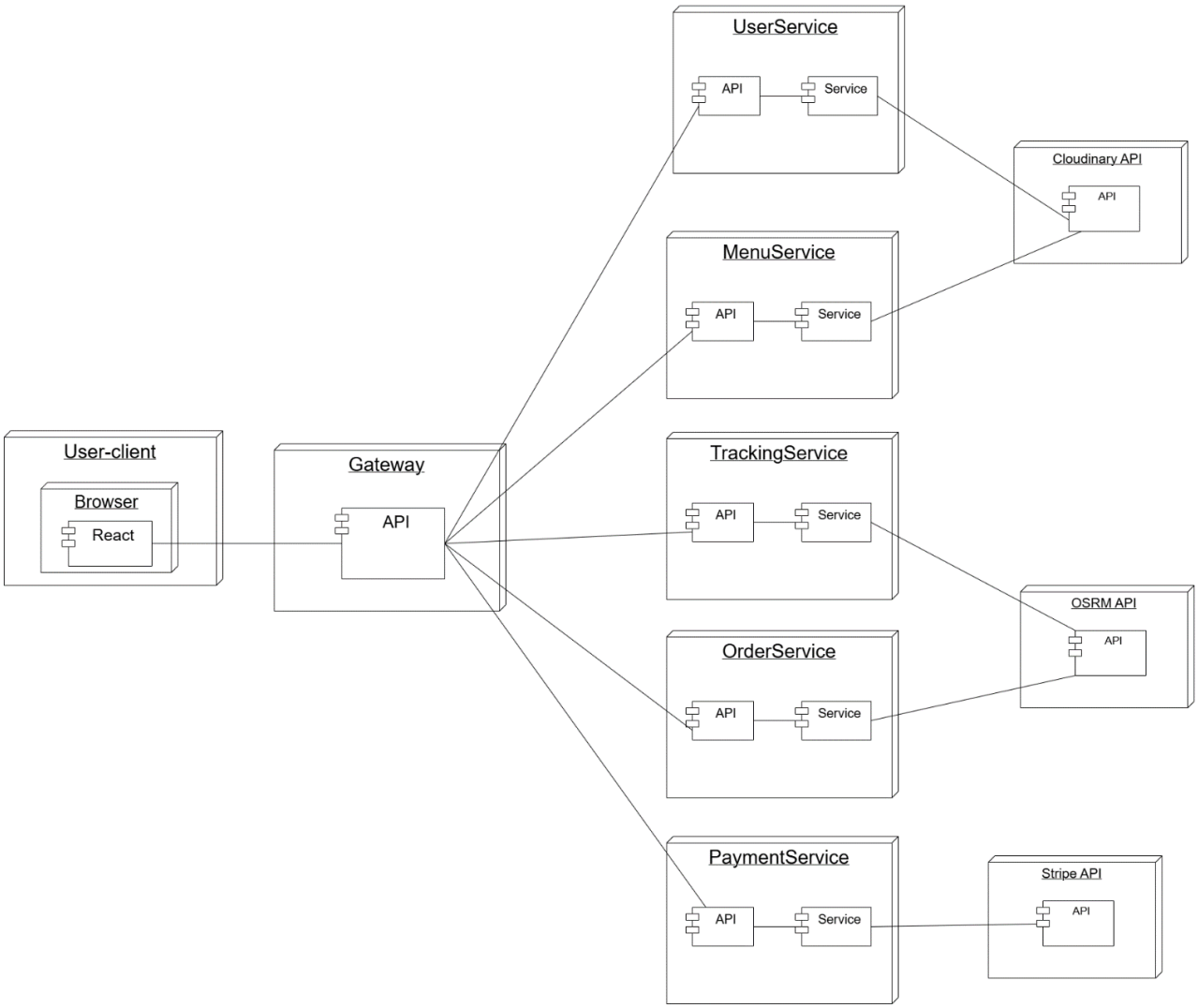


Рисунок 3.4 - UML Deployment Diagram програмної системи

3.3 Проєктування баз даних

Під час розроблення мікросервісної платформи доставки їжі одним із ключових етапів стало проєктування структури зберігання даних. Оскільки система побудована на основі мікросервісної архітектури, кожен сервіс має власну ізольовану базу даних, що відповідає принципу незалежного володіння даними (Database per Service Pattern). Такий підхід забезпечує низьку зв’язаність між сервісами, спрощує масштабування окремих компонентів та дозволяє уникнути прямого доступу одного сервісу до таблиць іншого сервісу [3].

У розробленій системі для зберігання даних використано PostgreSQL, а для сервісу геолокації - PostgreSQL із розширенням PostGIS, яке забезпечує підтримку просторових координат та географічних запитів [9, 29]. Кожен мікросервіс має власний контекст даних та окрему схему зберігання інформації. Це дозволяє реалізувати незалежне оновлення сервісів, а також зменшує ризик порушення цілісності даних під час змін у структурі окремих модулів системи.

У межах системи реалізовано п’ять основних баз даних:

- база даних сервісу користувачів (UserServiceDb);
- база даних сервісу меню (MenuServiceDb);
- база даних сервісу замовлень (OrderServiceDb);
- база даних сервісу платежів (PaymentServiceDb);
- база даних сервісу відстеження та геолокації (TrackingServiceDb).

База даних сервісу користувачів відповідає за зберігання інформації про користувачів, облікові записи різних типів, JWT refresh-токени, Stripe Connect акаунти та історію webhook-подій. У структурі сервісу реалізовано підтримку декількох типів акаунтів: клієнтських, бізнес-акаунтів ресторанів та акаунтів кур’єрів. Також окремо зберігаються дані для авторизації та відновлення сесій користувачів.

База даних сервісу меню містить інформацію про страви, категорії, інгредієнти та медіафайли меню. Для оптимізації продуктивності частина даних кешується за допомогою Redis, однак основним джерелом даних залишається PostgreSQL. У моделі даних реалізовано зв’язки між стравами та інгредієнтами, що дозволяє формувати повну інформацію про меню ресторану.

Сервіс замовлень використовує власну базу даних для зберігання інформації про замовлення, статуси доставки, історію змін, способи оплати та склад замовлення. Кожне замовлення містить набір позицій OrderedDish, у яких зберігається snapshot інформації про страву на момент оформлення замовлення.

У сервісі платежів реалізовано окрему фінансову модель даних, що містить інформацію про платежі, повернення коштів, баланси кур’єрів, payout-операції та історію фінансових транзакцій. Для забезпечення надійності обробки подій у структурі бази даних також реалізовано таблиці processed messages, processed

webhooks та outbox messages, що використовуються для забезпечення ідемпотентності та реалізації Outbox Pattern [3].

База даних сервісу відстеження використовує PostGIS для роботи з географічними координатами. У системі зберігаються координати кур'єрів, адреси закладів та точки доставки замовлень. Просторові типи даних дозволяють ефективно реалізувати обчислення маршрутів, визначення координат на карті та взаємодію із сервісами геолокації.

Для візуалізації структури даних системи доцільно використати ER-діаграми (Entity Relationship Diagram). На ER-діаграмах необхідно відобразити основні сутності кожного сервісу, їх атрибути, первинні та зовнішні ключі, а також взаємозв'язки між таблицями. Загальну ER-діаграму баз даних сервісів наведено на рисунку 3.5.

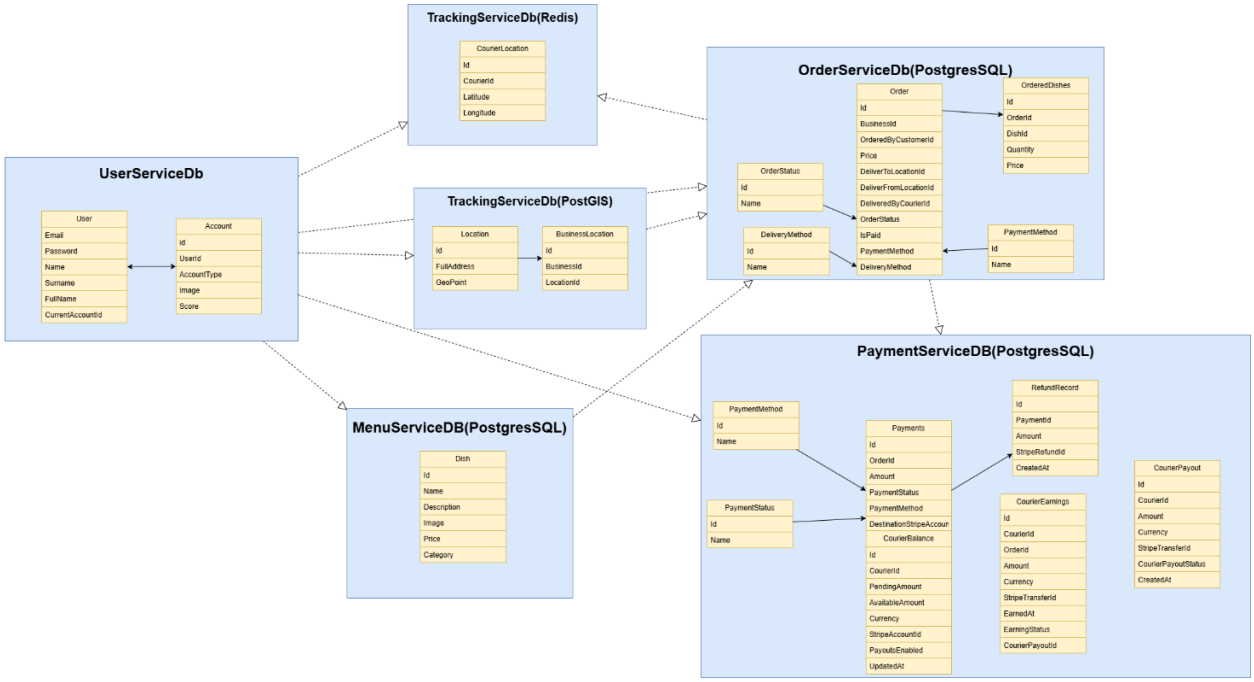


Рисунок 3.5 - ER-діаграма бази даних

Під час проєктування баз даних також враховувались принципи нормалізації даних, мінімізації дублювання інформації та забезпечення транзакційної цілісності. Для роботи з базами даних у всіх сервісах використано Entity Framework Core, який реалізує ORM-підхід та підтримує механізм міграцій для автоматизованого оновлення структури баз даних [6].

3.4 Висновки по розділу

У другому розділі виконано проєктування архітектури програмної системи онлайн замовлення та доставки їжі, що стало основою для подальшої практичної реалізації платформи.

Спроєктовано загальну архітектуру системи, яка являє собою багаторівневу вебплатформу, що складається з клієнтської частини на основі React, API Gateway, набору незалежних мікросервісів, брокера повідомлень, сервісу кешування та баз даних. Визначено основні компоненти системи та їхнє призначення: API Gateway як єдину точку входу, а також сервіси користувачів, меню, замовлень, платежів і відстеження доставки, кожен з яких відповідає за окрему бізнес-область. [18] Такий поділ забезпечує незалежне масштабування компонентів, підвищену відмовостійкість і зручність подальшого супроводу системи.

Детально розглянуто проєктування мікросервісної архітектури та взаємодії між сервісами. Обґрунтовано застосування двох способів взаємодії - синхронного через REST API для обробки запитів клієнта через шлюз та асинхронного обміну подіями через RabbitMQ для слабко зв'язаної комунікації між сервісами. Для забезпечення надійності передачі повідомлень передбачено застосування шаблону вихідної черги (Outbox Pattern), а також механізмів повторних спроб, ідемпотентності та черги недоставлених повідомлень. Окремо спроєктовано підсистему відстеження доставки в режимі реального часу на основі технології SignalR.

Виконано проєктування структури баз даних за принципом окремого сховища для кожного сервісу, що забезпечує низьку зв'язаність компонентів та унеможливлює прямий доступ одного сервісу до даних іншого. Як основне сховище даних обрано PostgreSQL, а для сервісу геолокації - його просторове розширення. Архітектурні рішення подано у вигляді UML-діаграм компонентів, послідовності та розгортання, а також ER-діаграми баз даних.

Отримані під час проєктування результати визначають структуру системи, способи взаємодії її компонентів і моделі даних.

РОЗДІЛ 4. РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ СЕРВІСУ ДОСТАВКИ ЇЖІ

4.1 Реалізація серверної частини системи

Реалізація API Gateway

Компонент API Gateway є центральною точкою входу до програмної системи та забезпечує єдиний інтерфейс взаємодії клієнтського застосунку з усіма мікросервісами платформи. Клієнтська частина ніколи не звертається до внутрішніх сервісів напряму - усі HTTP-запити надходять до Gateway, який виконує автентифікацію користувача, маршрутизацію запитів до відповідного сервісу, перетворення зовнішнього запиту у внутрішній підписаний виклик, обробку помилок та формування уніфікованої відповіді. Такий підхід дозволяє приховати внутрішню топологію системи, централізувати наскрізні задачі безпеки й надійності та зменшити зв’язність між клієнтом і набором сервісів [3].

На відміну від поширених готових рішень, таких як Ocelot або YARP, API Gateway у межах даної роботи реалізовано як власний зворотний проксі (reverse proxy), спроєктований під конкретні потреби платформи. Це дало змогу інтегрувати у шлюз власні механізми внутрішньої автентифікації між сервісами, передавання контексту користувача та формування єдиного формату відповіді без надлишкової конфігурації сторонніх бібліотек. Шлюз реалізовано засобами ASP.NET Core Web API на платформі .NET, що забезпечує високу продуктивність обробки запитів та зручну інтеграцію з механізмами middleware [6].

До основних задач, які виконує API Gateway, належать:

– забезпечення єдиної точки доступу до системи для клієнтського застосунку;

– автентифікація користувача шляхом перевірки JWT-токена, отриманого з cookie або заголовка Authorization;

– перетворення зовнішнього запиту у внутрішній виклик, підписаний за алгоритмом HMAC, що дозволяє сервісам пересвідчитися у походженні запиту саме від шлюзу;

– виконання повторних спроб (retry) з експоненційною затримкою для ідемпотентних запитів у разі тимчасових мережевих помилок;

– налаштування політики CORS на основі дозволеного джерела запитів;

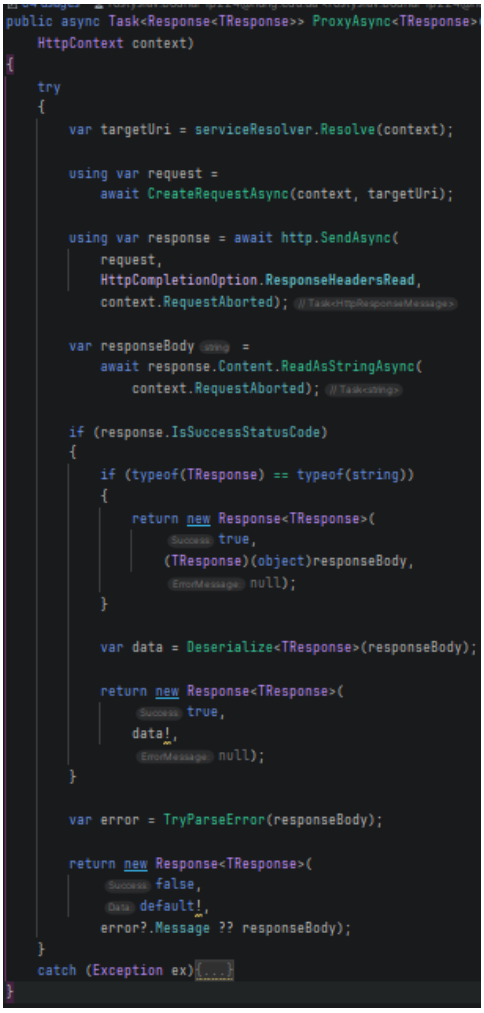
– формування уніфікованої відповіді у вигляді обгортки Response<T> з полями ознаки успіху, даних та повідомлення про помилку.

Маршрутизація запитів та проксіювання

Маршрутизація запитів у шлюзі реалізована на основі контролерів, згрупованих за цільовими сервісами. Кожен контролер позначається атрибутом GatewayService, який визначає, до якого внутрішнього сервісу має бути спрямований запит. Допоміжний клас ServiceResolver зчитує цей атрибут для поточної кінцевої точки та визначає базову адресу відповідного сервісу з конфігурації, що дозволяє уникнути жорсткого прив’язування адрес сервісів у коді контролерів.

Безпосереднє перенаправлення запиту виконує компонент GatewayProxy. Він копіює HTTP-метод, тіло та дозволені заголовки вхідного запиту, додає до них заголовки контексту користувача, переадресовує запит до цільового сервісу. Завдяки цьому кожен метод контролера зводиться фактично до одного виклику проксі, а таблиця маршрутів шлюзу являє собою повний перелік публічних кінцевих точок системи. Реалізацію методу проксіювання наведено на рисунку 3.1.

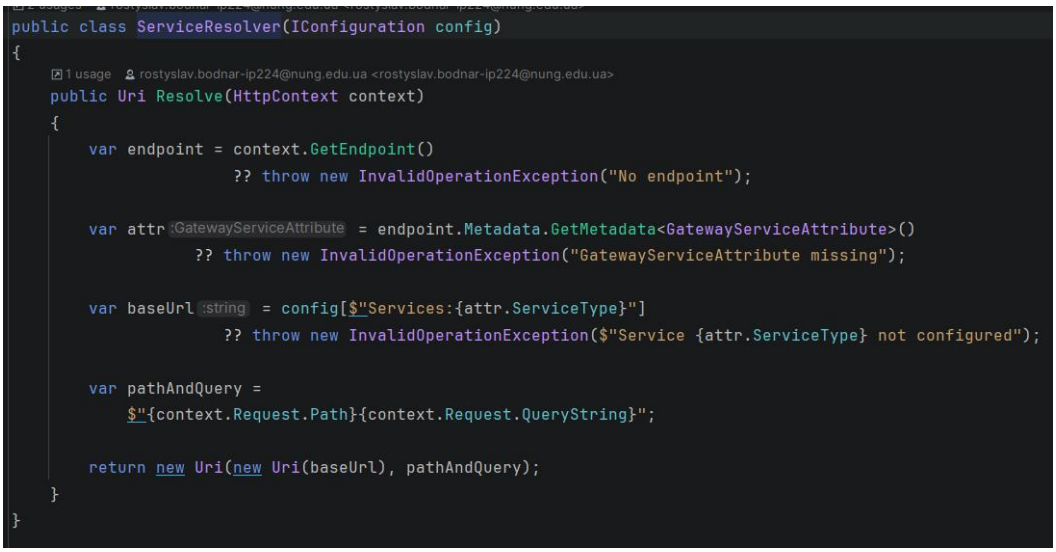
Така архітектура маршрутизації забезпечує розмежування відповідальності між компонентами шлюзу та спрощує розвиток системи. Додавання нового мікросервісу потребує лише створення відповідного контролера та налаштування маршруту в конфігурації. Використання компонента ServiceResolver дозволяє централізовано керувати адресами сервісів і адаптувати систему до різних середовищ розгортання без зміни програмного коду. Компонент GatewayProxy виконує передачу службових заголовків і забезпечує єдиний механізм взаємодії з внутрішніми сервісами. Завдяки цьому спрощується супровід програмного забезпечення та забезпечується однакова поведінка всіх кінцевих точок. Реалізацію методу проксіювання наведено на рисунку 4.1.



```
public async Task<Response<TResponse>> ProxyAsync<TResponse>(
    HttpContext context)
{
    try
    {
        var targetUri = serviceResolver.Resolve(context);

        using var request =
            await CreateRequestAsync(context, targetUri);

        using var response = await http.SendAsync(
            request,
            HttpCompletionOption.ResponseHeadersRead,
            context.RequestAborted);

        var responseBody =
            await response.Content.ReadAsStringAsync(
                context.RequestAborted);

        if (response.IsSuccessStatusCode)
        {
            if (typeof(TResponse) == typeof(string))
            {
                return new Response<TResponse>(
                    true,
                    (TResponse)(object)responseBody,
                    null);
            }

            var data = Deserialize<TResponse>(responseBody);

            return new Response<TResponse>(
                true,
                data!,
                null);
        }

        var error = TryParseError(responseBody);

        return new Response<TResponse>(
            false,
            default!,
            error?.Message ?? responseBody);
    }
    catch (Exception ex){...}
}
```

Рисунок 4.1 - Вміст GatewayProxy.cs (метод проксіювання запиту)

Визначення цільового сервісу на основі атрибута та конфігурації реалізовано у класі ServiceResolver, вміст якого наведено на рисунку 4.2.



```
public class ServiceResolver(IConfiguration config)
{
    1 usage  rostyslav.bodnar-ip224@nung.edu.ua <rostyslav.bodnar-ip224@nung.edu.ua>
    public Uri Resolve(HttpContext context)
    {
        var endpoint = context.GetEndpoint()
                       ?? throw new InvalidOperationException("No endpoint");

        var attr :GatewayServiceAttribute = endpoint.Metadata.GetMetadata<GatewayServiceAttribute>()
                  ?? throw new InvalidOperationException("GatewayServiceAttribute missing");

        var baseUrl :string = config[$"Services:{attr.ServiceType}"]
                     ?? throw new InvalidOperationException($"Service {attr.ServiceType} not configured");

        var pathAndQuery =
            $"{context.Request.Path}{context.Request.QueryString}";

        return new Uri(new Uri(baseUrl), pathAndQuery);
    }
}
```

Рисунок 4.2 - Вміст ServiceResolver.cs (визначення адреси цільового сервісу)

Автентифікація користувача

Автентифікація користувача на рівні шлюзу здійснюється з використанням JWT-токенів, які видаються сервісом користувачів під час входу або реєстрації. У токені зберігаються ідентифікатор користувача, його роль, ідентифікатор та тип активного облікового запису. Перевірка токена налаштовується у файлі Program.cs за допомогою параметрів JwtBearerOptions з валідацією видавця, аудиторії, підпису та терміну дії токена. Шлюз приймає токен доступу як із заголовка Authorization, так і з cookie, що дозволяє підтримувати різні сценарії роботи клієнтського застосунку [15]. Налаштування перевірки JWT-токена наведено на рисунку 4.3.

```
builder.Services.AddAuthentication( configureOptions: options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidIssuer = issuer,

        ValidateAudience = true,
        ValidAudience = audience,

        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new SymmetricSecurityKey(keyBytes),

        ValidateLifetime = true,
        ClockSkew = TimeSpan.FromSeconds(30)
    };

    // IMPORTANT for gateway scenarios (cookies + browser)
    options.Events = new JwtBearerEvents
    {
        OnMessageReceived = context =>
        {
            // support both Authorization header and cookie
            var token :string? = context.Request.Headers["Authorization"].FirstOrDefault();

            if (string.IsNullOrEmpty(token))
                token = context.Request.Cookies["accessToken"];

            if (!string.IsNullOrEmpty(token))
                context.Token = token.Replace("Bearer ", "");

            return Task.CompletedTask;
        }
    };
});
```

Рисунок 4.3 - Вміст Program.cs (налаштування JWT-автентифікації шлюзу)

Внутрішня автентифікація запитів за алгоритмом HMAC

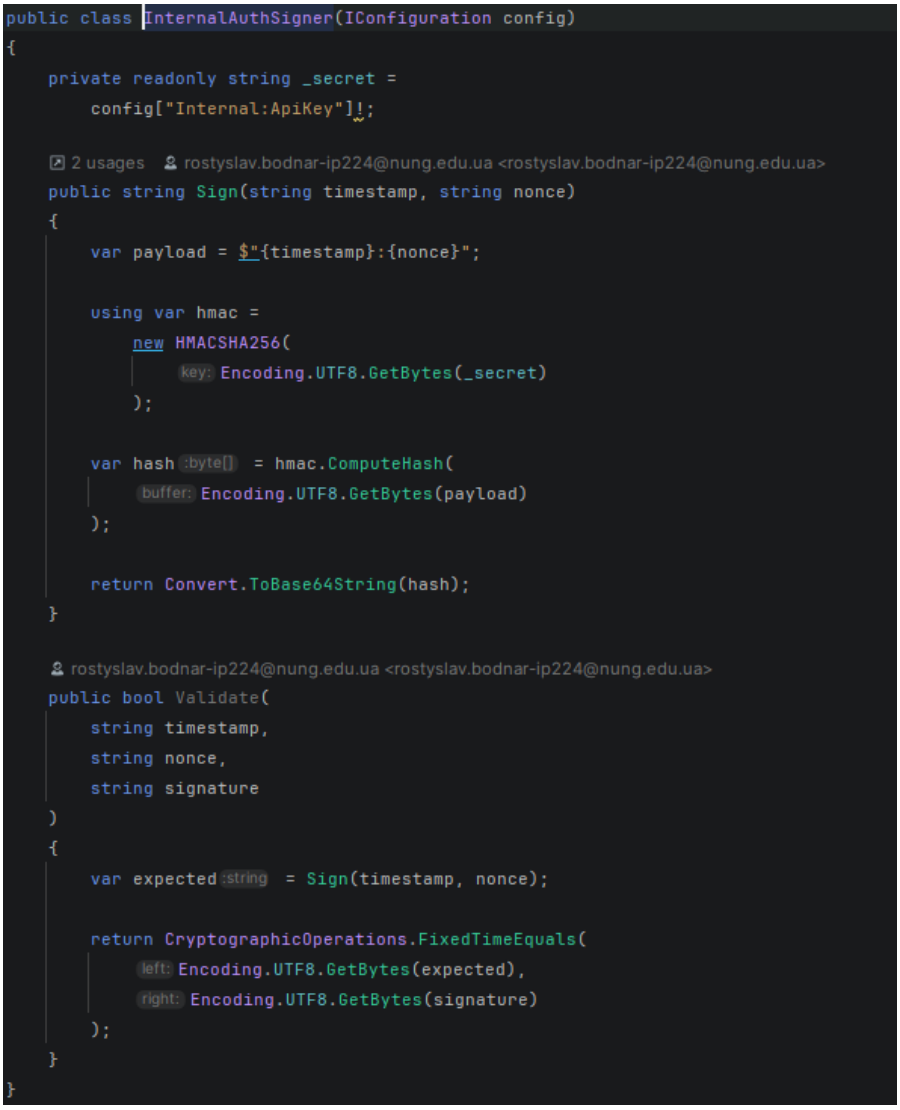
Оскільки внутрішні сервіси не повинні бути доступними напряму, кожен запит, що перенаправляється шлюзом, додатково підписується для підтвердження його походження. Формування підпису виконує клас InternalAuthSigner, який додає до запиту чотири службові заголовки. На стороні кожного сервісу проміжний обробник InternalAuthMiddleware перевіряє наявність та коректність підпису, ключа й часової мітки і відхиляє запит зі статусом 401 у разі їх відсутності або недійсності. Перелік службових заголовків внутрішньої автентифікації наведено в таблиці 4.1.

Таблиця 4.1.

Заголовки внутрішньої автентифікації запитів

HTTP-заголовок	**Призначення**
X-Internal-Timestamp	Час формування запиту (Unix-секунди); запит відхиляється, якщо він старший за п’ять хвилин
X-Internal-Nonce	Випадкове значення для кожного запиту; разом із часовою міткою захищає від повторного відтворення запитів (replay-атак)
X-Internal-Signature	HMAC-підпис, обчислений на основі часової мітки та nonce із використанням спільного секретного ключа
X-Internal-Key	Спільний внутрішній ключ, що звіряється за допомогою CryptographicOperations.FixedTimeEquals для захисту від атак за часом виконання

Окрім підпису, шлюз передає у внутрішні сервіси контекст автентифікованого користувача за допомогою заголовків X-Internal-UserId, X-Internal-Role, X-Internal-AccountId та X-Internal-AccountType. На стороні сервісу проміжний обробник UserContextMiddleware відтворює з цих заголовків об’єкт контексту користувача та надає його через механізм впровадження залежностей. Такий підхід дозволяє кожному сервісу мати доступ до інформації про користувача без повторної перевірки токена. Реалізацію формування підписаних заголовків наведено на рисунку 3.4.



```
public class InternalAuthSigner(IConfiguration config)
{
    private readonly string _secret =
        config["Internal:ApiKey"]!;

    2 usages  rostyslav.bodnar-ip224@nung.edu.ua <rostyslav.bodnar-ip224@nung.edu.ua>
    public string Sign(string timestamp, string nonce)
    {
        var payload = $"{timestamp}:{nonce}";

        using var hmac =
            new HMACSHA256(
                key: Encoding.UTF8.GetBytes(_secret)
            );

        var hash :byte[]  = hmac.ComputeHash(
            buffer: Encoding.UTF8.GetBytes(payload)
        );

        return Convert.ToBase64String(hash);
    }

    rostyslav.bodnar-ip224@nung.edu.ua <rostyslav.bodnar-ip224@nung.edu.ua>
    public bool Validate(
        string timestamp,
        string nonce,
        string signature
    )
    {
        var expected :string  = Sign(timestamp, nonce);

        return CryptographicOperations.FixedTimeEquals(
            left: Encoding.UTF8.GetBytes(expected),
            right: Encoding.UTF8.GetBytes(signature)
        );
    }
}
```

Рисунок 4.4 - Вміст InternalAuthSigner.cs (формування HMAC-заголовків)

Для захисту від повторного відтворення запитів використовується поєднання часової мітки та одноразового значення nonce, а перевірка ключа виконується методом FixedTimeEquals із бібліотеки криптографічних операцій, що унеможливлює визначення ключа за часом виконання порівняння. Окремі групи кінцевих точок навмисно звільнені від внутрішньої автентифікації, зокрема методи входу та реєстрації, перевірки стану сервісу та обробники webhook-подій, які мають власні механізми перевірки підпису [14].

Надійність взаємодії та обробка помилок

Для підвищення стійкості системи до тимчасових збоїв у шлюзі реалізовано політику повторних спроб на основі бібліотеки Polly. Повторне виконання застосовується лише до ідемпотентних HTTP-методів (GET, HEAD, OPTIONS) і виконується до трьох разів з експоненційним зростанням затримки у разі

мережевих помилок або відповідей із кодом 5xx. Це дозволяє автоматично відновлювати взаємодію за короткочасної недоступності сервісів без втручання користувача [3]. Налаштування політики повторних спроб наведено на рисунку 4.5.

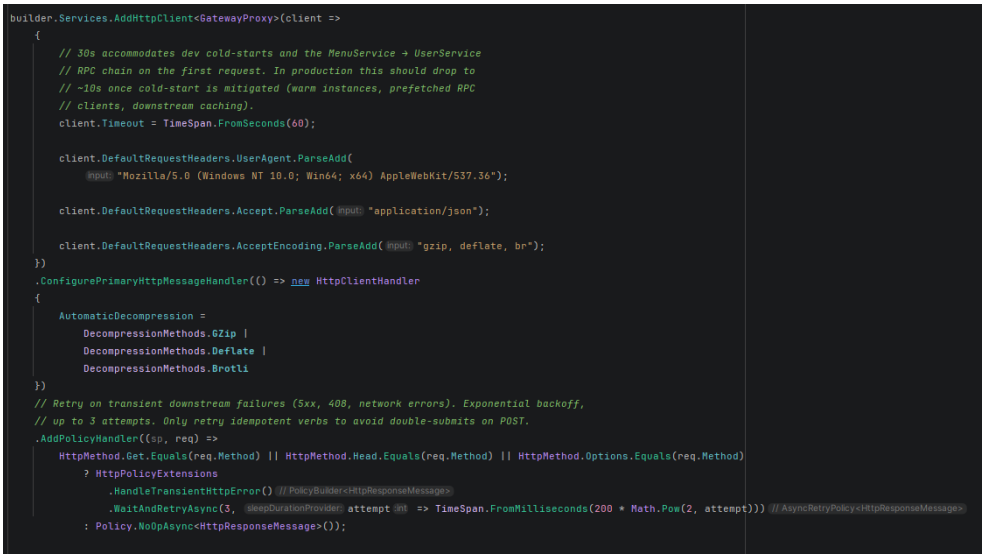


Рисунок 4.5 - Вміст Program.cs (політика повторних спроб Polly)

Централізована обробка помилок реалізована у проміжному обробнику ExceptionHandlingMiddleware, який перехоплює необроблені винятки та перетворює їх на стандартну відповідь у форматі обгортки Response<T> з повідомленням про помилку. Завдяки цьому клієнтський застосунок завжди отримує відповідь єдиного формату незалежно від того, який саме внутрішній сервіс обробляв запит, що спрощує опрацювання помилок на стороні клієнта та робить поведінку системи передбачуваною.

Таким чином, реалізований API Gateway виконує роль єдиної захищеної точки входу до мікросервісної платформи доставки їжі та поєднує у собі функції маршрутизації, автентифікації користувачів, внутрішньої автентифікації запитів між сервісами, забезпечення надійності взаємодії та уніфікованої обробки помилок. Застосовані рішення дозволяють приховати внутрішню структуру системи, підвищити рівень безпеки міжсервісної взаємодії та забезпечити стабільну й передбачувану роботу платформи за умов розподіленої архітектури.

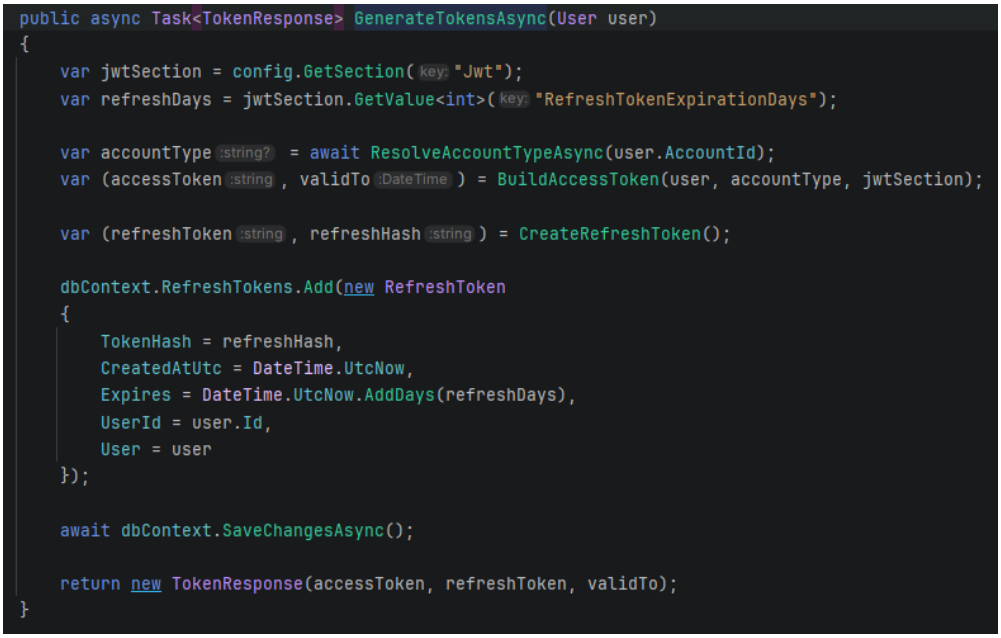
Реалізація сервісу користувачів

Сервіс користувачів (UserService) відповідає за керування обліковими записами, реєстрацію, автентифікацію та авторизацію користувачів платформи. Цей сервіс є джерелом видачі JWT-токенів, які надалі використовуються для перевірки прав доступу на рівні API Gateway та інших сервісів. Окрім базових операцій автентифікації, UserService реалізує підтримку декількох типів облікових записів, керування профілями, завантаження зображень та інтеграцію зі Stripe Connect для онбордингу бізнесів і кур'єрів.

Доменна модель сервісу побудована навколо сутності користувача, що успадковує клас IdentityUser із бібліотеки ASP.NET Core Identity, та сутності облікового запису. Облікові записи реалізовано за принципом єдиної таблиці зі стовпцем-дискримінатором (Table-per-Hierarchy), що дозволяє зберігати в одній таблиці три типи акаунтів - клієнтський, бізнес-акаунт ресторану та акаунт кур'єра. Один користувач може мати декілька облікових записів різних типів, а поле активного акаунта визначає, від імені якої ролі він взаємодіє із системою. Додатково в моделі передбачено сутності refresh-токенів, записів про виплати та оброблених webhook-подій [6].

Автентифікація та керування сесіями

Процес автентифікації реалізовано у контролері автентифікації, який надає кінцеві точки реєстрації, входу, оновлення та виходу. Під час реєстрації засобами менеджера користувачів ASP.NET Identity створюється новий запис користувача, для якого автоматично формується клієнтський обліковий запис, що дозволяє кожному новому користувачеві одразу користуватися сервісом. Після успішного входу або реєстрації клієнт отримує об'єкт відповіді, який містить токен доступу, refresh-токен та час завершення дії токена доступу. Видачу токенів виконує окремий сервіс TokenService, який формує JWT із потрібним набором полів (ідентифікатор користувача, роль, ідентифікатор та тип активного акаунта). Реалізацію формування токена наведено на рисунку 4.6.



```
public async Task<TokenResponse> GenerateTokensAsync(User user)
{
    var jwtSection = config.GetSection( key: "Jwt");
    var refreshDays = jwtSection.GetValue<int>( key: "RefreshTokenExpirationDays");

    var accountType :string? = await ResolveAccountTypeAsync(user.AccountId);
    var (accessToken :string , validTo :DateTime ) = BuildAccessToken(user, accountType, jwtSection);

    var (refreshToken :string , refreshHash :string ) = CreateRefreshToken();

    dbContext.RefreshTokens.Add(new RefreshToken
    {
        TokenHash = refreshHash,
        CreatedAtUtc = DateTime.UtcNow,
        Expires = DateTime.UtcNow.AddDays(refreshDays),
        UserId = user.Id,
        User = user
    });

    await dbContext.SaveChangesAsync();

    return new TokenResponse(accessToken, refreshToken, validTo);
}
```

Рисунок 4.6 - Вміст TokenService.cs (формування JWT-токена)

Refresh-токен зберігається у базі даних та додатково передається клієнту у вигляді захищеного cookie з ознакою HttpOnly, що унеможливлює доступ до нього зі сторони клієнтських сценаріїв та підвищує безпеку сесії. Оновлення сесії виконується кінцевою точкою refresh, яка перевипускає обидва токени, а вихід із системи відкликає відповідний refresh-токен та очищує cookie. Кінцеві точки автентифікації, перевірки стану сервісу та обробники webhook-подій навмисно звільнені від внутрішньої автентифікації між сервісами [15].

Керування обліковими записами та профілем

Сервіс надає функціонал створення, оновлення та видалення облікових записів кожного типу, а також перегляду профілю користувача разом із переліком усіх його акаунтів. Створення та оновлення акаунтів реалізовано із підтримкою передавання файлів (multipart), що дозволяє завантажувати аватар користувача, логотип закладу або зображення посвідчення кур'єра. Зберігання та доставку зображень забезпечує зовнішній сервіс Cloudinary, інтеграцію з яким реалізовано у класі CloudinaryService [16].

Окремою важливою операцією є перемикання активного облікового запису. Оскільки роль та контекст користувача закодовані безпосередньо в JWT-токені, зміна активного акаунта супроводжується перевипуском токена з оновленими

полями ролі та ідентифікатора акаунта. Реалізацію перемикання активного облікового запису наведено на рисунку 4.7.

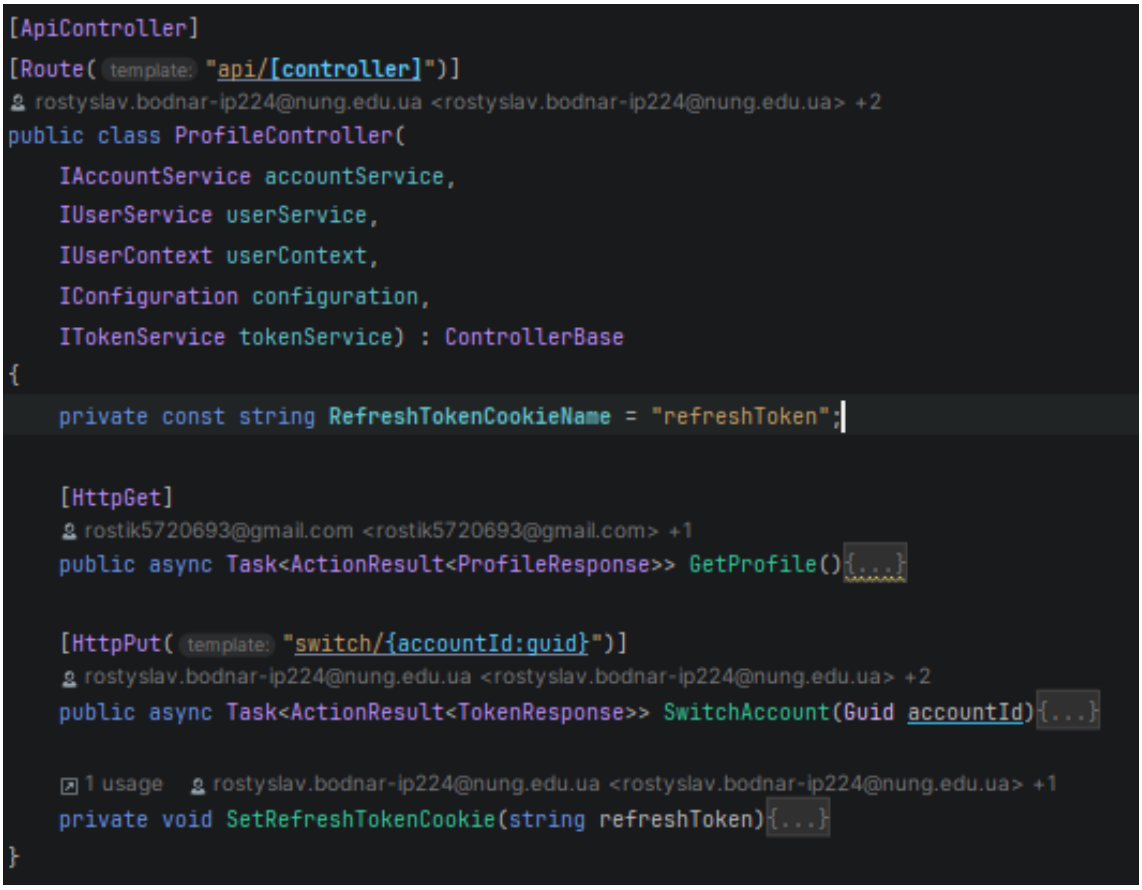


Рисунок 4.7 - Вміст ProfileController.cs (перемикання активного акаунта)

Інтеграція зі Stripe Connect

Для забезпечення можливості прийому платежів та виплат бізнеси й кур'єри проходять процедуру онбордингу у платіжній системі Stripe за моделлю Stripe Connect. Створення під'єднаних акаунтів та формування посилань для проходження онбордингу виконує сервіс StripeConnectService. Оскільки ця операція може потребувати повторних спроб, її винесено у фоновий процес StripeAccountProvisioningWorker, який періодично опрацьовує акаунти, що очікують ініціалізації, та зберігає отримані ідентифікатори. Зміни стану під'єднаних акаунтів і подій балансу опрацьовуються двома окремими контролерами webhook-подій, а для уникнення повторної обробки одних і тих самих подій використовується таблиця оброблених webhook-подій [14].

Взаємодія з іншими сервісами

UserService бере участь у міжсервісній взаємодії як у синхронному, так і в асинхронному режимах. Для синхронних запитів інших сервісів реалізовано набір RPC-споживачів, що виконують пошук облікових записів - отримання акаунта за ідентифікатором, отримання бізнес-, клієнтського або кур'єрського акаунта, а також пакетні варіанти цих запитів. Своєю чергою, під час створення акаунтів сервіс сам звертається до сервісу відстеження через RPC-клієнт для отримання інформації про локації закладів. У межах асинхронної взаємодії сервіс публікує події, зокрема подію створення облікового запису та подію завершення виплати кур'єру. Загальну схему взаємодії сервісу користувачів з іншими сервісами наведено на рисунку 3.8.

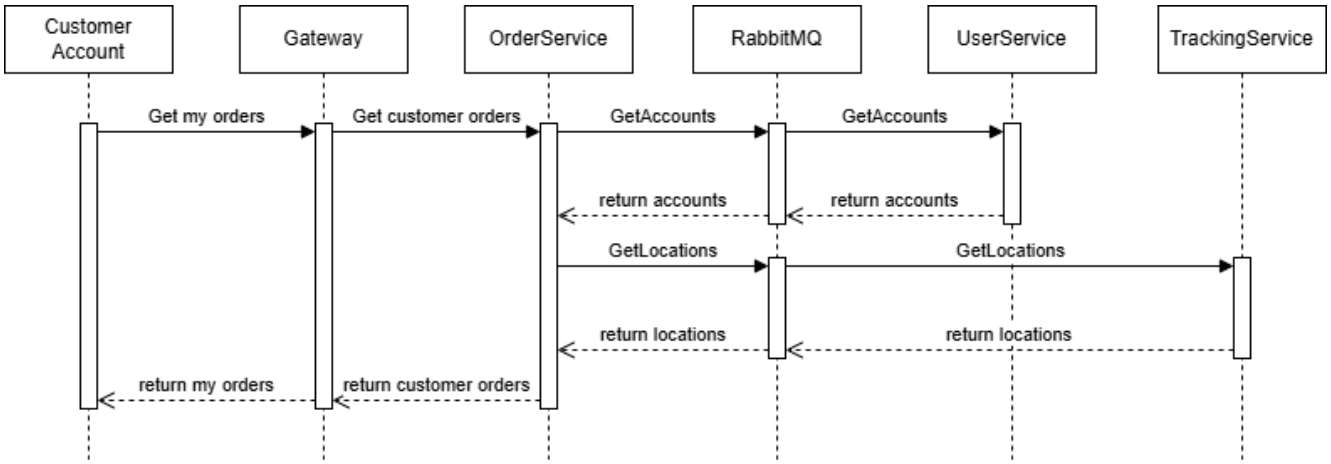


Рисунок 4.8 – Діаграма взаємодії сервісу користувачі при отриманні замовлення

Для постійного зберігання даних сервіс використовує СКБД PostgreSQL у поєднанні з Entity Framework Core та сховищами ASP.NET Identity. Міграції структури бази даних застосовуються автоматично під час запуску сервісу, а для контейнерів-міграторів передбачено окремий режим запуску, що дозволяє уникнути одночасного внесення змін до структури бази даними кількома екземплярами сервісу. Коректність роботи бізнес-логіки сервісу перевірено за допомогою модульних тестів, реалізованих із використанням бібліотек xUnit, Moq та FluentAssertions і провайдера бази даних у пам'яті.

Таким чином, сервіс користувачів реалізує повний цикл роботи з обліковими записами - від реєстрації та автентифікації до керування профілями, інтеграції з платіжною системою та забезпечення міжсервісної взаємодії. Винесення логіки автентифікації та керування користувачами в окремий сервіс відповідає принципам мікросервісної архітектури та забезпечує централізоване й безпечне керування доступом у межах усієї платформи.

Реалізація сервісу меню

Сервіс меню (MenuService) відповідає за зберігання та обробку інформації про страви, їхні категорії та інгредієнти. Він реалізує функціонал перегляду меню для клієнтів і повного керування стравами з боку бізнес-користувачів, а також забезпечує завантаження зображень страв та доповнення інформації про страви даними про відповідний заклад. Виділення роботи з меню в окремий сервіс відповідає принципу єдиної відповідальності та дозволяє масштабувати й оновлювати цей компонент незалежно від інших частин системи.

Доменна модель сервісу складається із сутностей категорії, страви та інгредієнта, а також службової сутності ключа ідемпотентності. Кожна страва належить до однієї з вбудованих категорій, перелік яких реалізовано у вигляді переліку (enum) і містить п’ятнадцять значень - напої, супи, салати, піца, бургери, паста, суші, десерти, сніданки, страви гриль, гарніри, соуси, веганські страви, дитяче меню та спеціальні пропозиції. Між стравами та інгредієнтами реалізовано зв’язок, що дозволяє формувати повну інформацію про склад страви.

Перегляд та керування стравами

Сервіс надає окремі кінцеві точки для отримання повного переліку страв, перегляду конкретної страви, отримання страв для клієнта та страв у межах окремого закладу. Отримання списку страв реалізовано із серверною пагінацією: сервіс повертає не лише набір елементів, а й службові заголовки відповіді з інформацією про поточну сторінку, її розмір, загальну кількість елементів та сторінок. Це дозволяє ефективно працювати з великими обсягами даних та зменшує навантаження на клієнтську частину.

Для бізнес-користувачів реалізовано повний цикл керування стравами - створення, оновлення та видалення позицій меню. Операції створення та оновлення

підтримують передавання файлів (multipart) для завантаження зображень страв, які зберігаються та доставляються через сервіс Cloudinary. Щоб уникнути повторного створення однакових страв у разі повторного надсилання запиту через нестабільне з'єднання, операція створення захищена механізмом ідемпотентності: спеціальний фільтр перевіряє ключ ідемпотентності та зберігає його у відповідній таблиці. Реалізацію створення страви наведено на рисунку 4.9.

```
[HttpPost( template: "create")]
[Idempotent]
[Consumes( contentType: "multipart/form-data")]
rostik5720693@gmail.com <rostik5720693@gmail.com> +1
public async Task<IActionResult> CreateDish([FromForm] CreateDishRequest request)
{
    var result DishResponse  = await dishService.CreateDishAsync(request);
    return Ok(result);
}
```

Рисунок 4.9 - Вміст DishController.cs (метод створення страви)

Валідація вхідних даних

Для перевірки коректності вхідних даних у сервісі застосовано бібліотеку FluentValidation, інтегровану через механізм автоматичної валідації. Валідатори, описані для відповідних моделей запитів, автоматично виконуються під час надходження запиту до контролера, що дозволяє відокремити логіку перевірки даних від бізнес-логіки та централізувати правила валідації. У разі невідповідності даних установленим правилам сервіс повертає відповідь із описом помилок ще до виконання основної логіки обробки запиту [6].

Кешування та взаємодія з іншими сервісами

Оскільки інформація про меню є одними з найчастіше запитуваних даних, для зменшення навантаження на базу даних та прискорення обробки запитів у сервісі реалізовано кешування за допомогою Redis із використанням клієнта StackExchange.Redis. Найбільш популярні запити на отримання страв обслуговуються з кешу, тоді як основним джерелом даних залишається PostgreSQL [10].

Для формування повної інформації про страву разом із даними про заклад, якому вона належить, сервіс меню звертається до сервісу користувачів через

механізм RPC поверх RabbitMQ за допомогою клієнта UserServiceRpcClient. Своєю чергою, MenuService сам надає RPC-споживачів для отримання інформації про страви іншими сервісами - отримання окремої страви, набору страв та пакетного запиту страв. Такий підхід дозволяє сервісам обмінюватися даними без прямої залежності від HTTP-адрес одне одного та без порушення принципу ізоляції баз даних [3]. Загальну схему взаємодії сервісу меню з іншими сервісами наведено на рисунку 4.10.

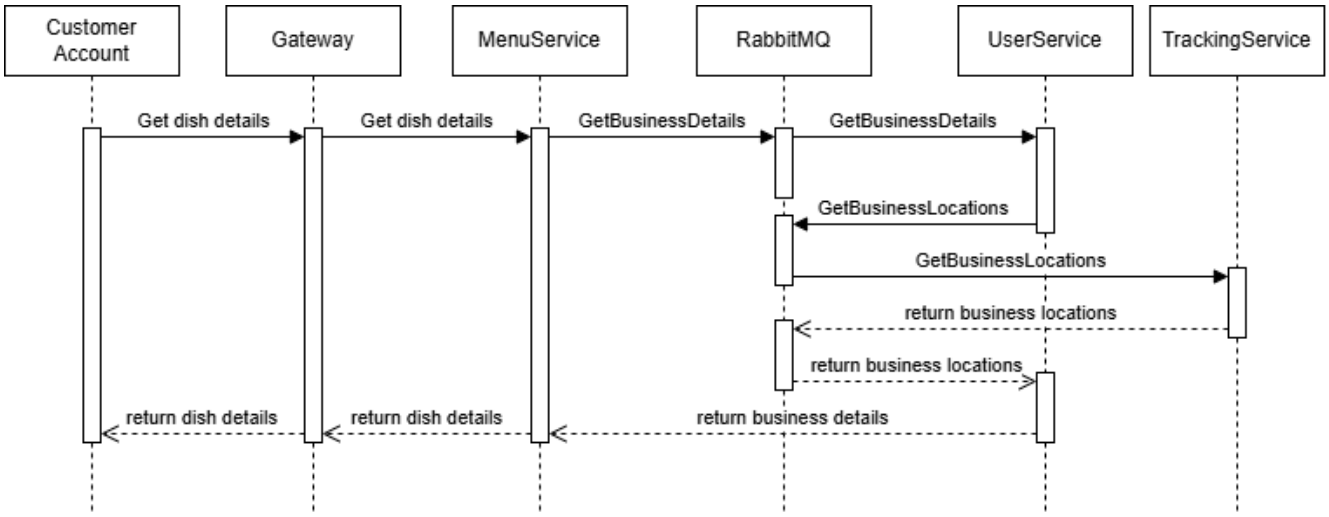


Рисунок 4.10 – Діаграма взаємодії сервісу меню при отриманні деталей страви

Для постійного зберігання даних сервіс використовує СКБД PostgreSQL та Entity Framework Core, а міграції структури бази даних застосовуються автоматично під час запуску. Поєднання реляційного сховища для надійного зберігання даних та кешу Redis для прискорення читання дозволяє забезпечити як цілісність інформації про меню, так і високу швидкодію під час обслуговування клієнтських запитів.

Отже, сервіс меню реалізує повний функціонал роботи зі стравами та категоріями, поєднуючи серверну пагінацію, валідацію вхідних даних, ідемпотентне створення позицій, кешування та міжсервісну взаємодію через RPC. Реалізовані рішення забезпечують зручне керування меню з боку закладів харчування та швидкий перегляд страв клієнтами, що є одними з ключових сценаріїв роботи платформи доставки їжі.

Реалізація сервісу замовлень

Сервіс замовлень (OrderService) реалізує основну бізнес-логіку платформи, пов’язану з оформленням та обробкою замовлень. Він відповідає за створення замовлень, керування їхнім життєвим циклом, формування історії замовлень для різних ролей користувачів, надання аналітичної інформації для закладів та координацію взаємодії із сервісами платежів і відстеження доставки. Саме навколо сервісу замовлень будується наскрізний сценарій роботи системи - від формування кошика до завершення доставки.

Доменна модель сервісу складається із сутності замовлення, позицій замовлення, переліків статусу замовлення, способу доставки та способу оплати, а також службових сутностей ключа ідемпотентності та повідомлення вихідної черги (outbox). Кожна позиція замовлення зберігає знімок (snapshot) інформації про страву на момент оформлення - назву, ціну та кількість. Такий підхід дозволяє зберегти історичні дані замовлення незмінними навіть після подальшого редагування або видалення страви рестораном, що є важливим для коректного відображення історії та фінансового обліку.

Життєвий цикл замовлення

Стан замовлення описується переліком статусів, який реалізує машину станів із такими значеннями: підготовка, готове, передане на доставку, отримане кур’єром, доставлене та скасоване. Бізнес-користувач послідовно переводить замовлення зі стану підготовки до стану готовності, після чого кур’єр приймає замовлення, позначає його отримання та завершення доставки. На будь-якому активному етапі замовлення може бути скасоване клієнтом або закладом. Зміна статусу виконується відповідною кінцевою точкою, яка перевіряє допустимість переходу між станами. Для різних ролей користувачів передбачено окремі запити: перелік активних замовлень клієнта, черга відкритих замовлень закладу, доступні та активні замовлення кур’єра, а також історія замовлень для кожної ролі. Життєвий цикл замовлення та допустимі переходи між його станами наведено на рисунку 4.11.

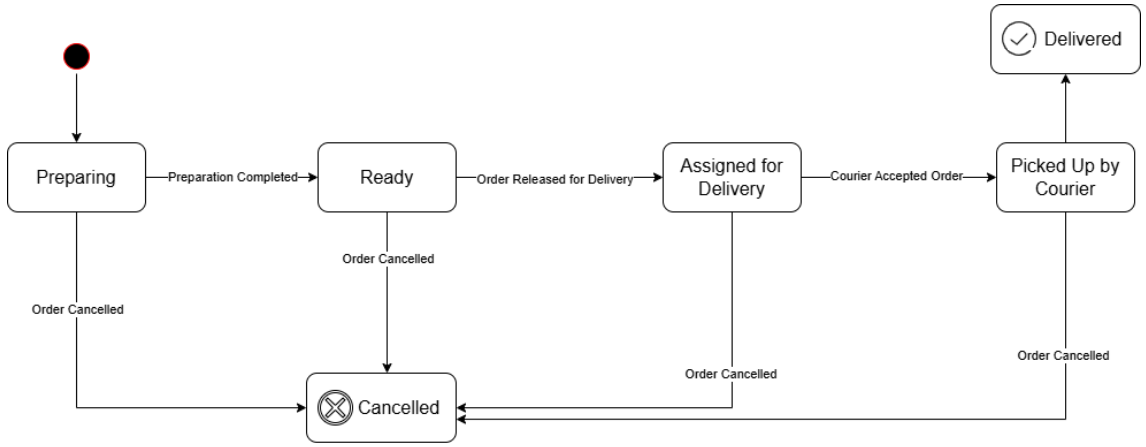


Рисунок 4.11 - Діаграма станів життєвого циклу замовлення

Забезпечення надійності обробки замовлень

Оскільки створення замовлення є критичною операцією, у сервісі реалізовано механізм ідемпотентності: спеціальний фільтр перевіряє ключ ідемпотентності та зберігає його у відповідній таблиці, що унеможливлює повторне створення замовлення в разі повторного надсилання запиту. Такий підхід дозволяє уникнути дублювання записів у базі даних та забезпечує коректну обробку запитів навіть за умов нестабільного мережевого з'єднання або повторних спроб відправлення запиту клієнтом.

Для надійної передачі подій іншим сервісам застосовано шаблон вихідної черги (Outbox Pattern). Події замовлення записуються у таблицю вихідних повідомлень у межах тієї самої транзакції бази даних, що й зміна стану замовлення, після чого фоновий процес-публікатор періодично зчитує ці повідомлення та надсилає їх до брокера RabbitMQ. Завдяки використанню єдиної транзакції досягається узгодженість між даними предметної області та повідомленнями, які мають бути доставлені іншим сервісам. Це усуває ризик виникнення ситуацій, коли зміни в базі даних були успішно збережені, але відповідна подія не була опублікована.

Крім того, застосування шаблону Outbox Pattern підвищує відмовостійкість системи та сприяє реалізації принципів асинхронної взаємодії між мікросервісами. Такий підхід гарантує, що подія не буде втрачена навіть у випадку тимчасової недоступності брокера повідомлень [3]. Реалізацію шаблону вихідної черги наведено на рисунку 4.12.

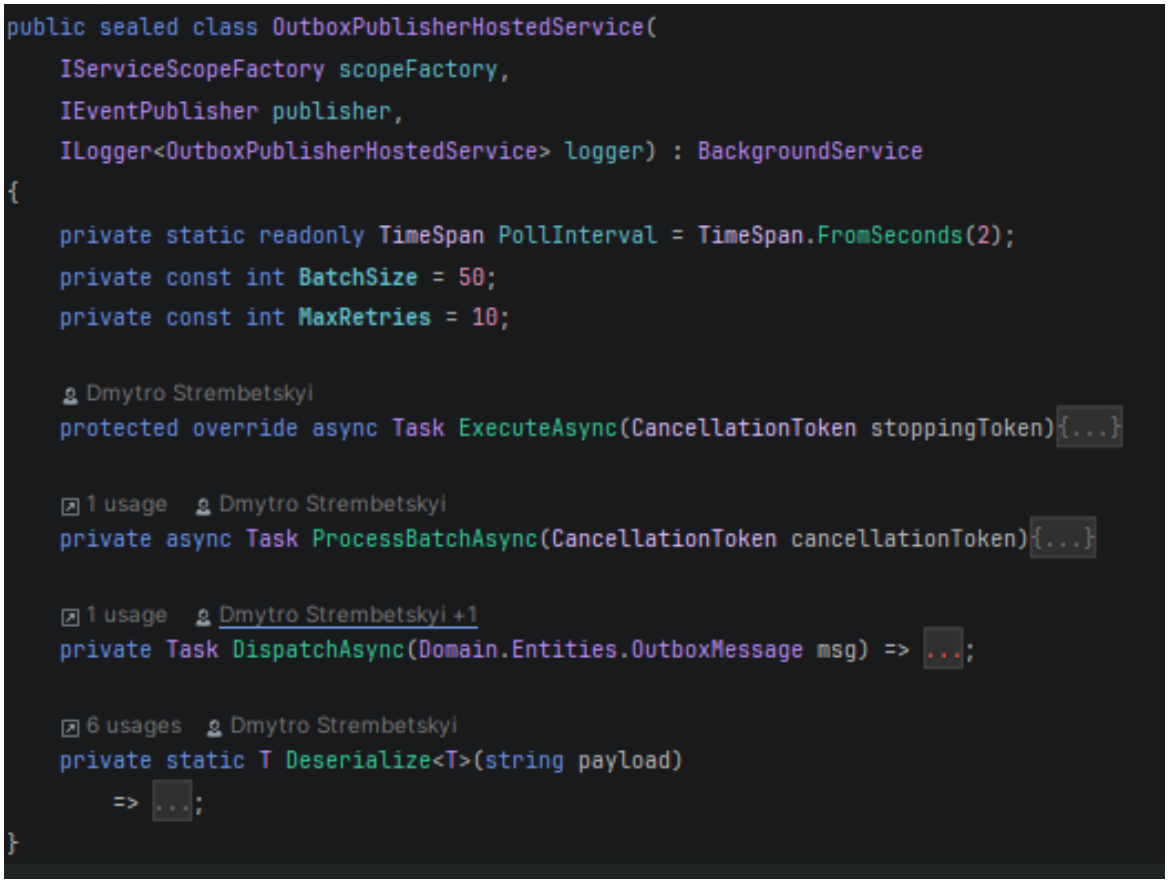


Рисунок 4.12 - Реалізація шаблону Outbox у сервісі замовлень

Додатково у сервісі реалізовано фоновий процес контролю часу очікування оплати, який скасовує замовлення, оплата за якими не була підтверджена протягом установленого часу. Перевірка вхідних даних здійснюється засобами бібліотеки FluentValidation через механізм автоматичної валідації, що дозволяє відхиляти некоректні запити ще до виконання основної бізнес-логіки.

Взаємодія з іншими сервісами

Сервіс замовлень виступає основним джерелом подій у системі. Він публікує події створення замовлення, зміни статусу, отримання та доставки замовлення, його скасування, а також події, пов’язані з розрахунком за готівкову оплату. Ці події споживаються сервісом платежів та сервісом відстеження доставки, що забезпечує узгоджену роботу системи без прямих синхронних викликів між сервісами. Своєю чергою, сервіс замовлень споживає повідомлення про успішне виконання платежу для оновлення статусу попередньо оплачених замовлень. Загальну послідовність взаємодії під час оформлення замовлення наведено на діаграмі послідовності у підрозділі 3.3 (рисунок 3.3).

Окремо сервіс відповідає за видачу короткочасних токенів доступу до відстеження. Для кожного замовлення може бути сформований окремий JWT-токен, обмежений ідентифікатором конкретного замовлення, який згодом перевіряється вузлом реального часу сервісу відстеження.

Для постійного зберігання даних сервіс використовує СКБД PostgreSQL та Entity Framework Core, а міграції структури бази даних застосовуються автоматично під час запуску. Таким чином, сервіс замовлень реалізує повний життєвий цикл замовлення із забезпеченням ідемпотентності, надійної передачі подій через шаблон вихідної черги та узгодженої взаємодії з іншими сервісами, що відповідає сучасним підходам до побудови розподілених систем.

Реалізація сервісу платежів

Сервіс платежів (PaymentService) відповідає за обробку фінансових операцій платформи - створення та підтвердження платежів, обробку готівкових розрахунків, повернення коштів, а також облік і виплату винагороди кур'єрам. Винесення платіжної логіки в окремий сервіс дозволяє ізолювати роботу з фінансовими даними, підвищити рівень безпеки системи та зосередити в одному компоненті всю інтеграцію із зовнішньою платіжною платформою Stripe.

Доменна модель сервісу містить сутності платежу, статусу та способу оплати, запису про повернення коштів, а також групу сутностей фінансового обліку кур'єрів - балансу, нарахувань та виплат. Окремо реалізовано службові сутності оброблених повідомлень, оброблених webhook-подій, повідомлень вихідної черги та повідомлень черги «мертвих» листів, які використовуються для забезпечення ідемпотентності й надійної обробки подій. Архітектурно сервіс побудований за полегшеним варіантом шаблону CQRS: кожна операція описується окремою командою та відповідним обробником команди - створення платежу, створення платіжного наміру Stripe, прийому готівкового розрахунку, скасування платежу та повернення коштів.

Онлайн-оплата через Stripe

Інтеграцію з платіжною системою інкапсульовано у сервісі StripeService, який є обгорткою над офіційним SDK Stripe для платформи .NET. Параметри інтеграції (секретний ключ, секрет webhook, країна за замовчуванням та адреси

панелі) зчитуються з конфігурації. Для виконання онлайн-оплати обробник команди створення платіжного наміру формує об’єкт PaymentIntent із зазначенням комісії платформи та призначенням переказу на під’єднаний акаунт відповідного закладу за моделлю Stripe Connect. Клієнтська частина підтверджує платіж за допомогою бібліотеки Stripe.js. Реалізацію створення платіжного наміру наведено на рисунку 4.13.

```
public class CreateStripePaymentIntentCommandHandler(IPaymentRepository repo, IStripeService stripe)
{
    rostik5720693@gmail.com <rostik5720693@gmail.com> +1
    public async Task Handle(CreateStripePaymentIntentCommand cmd, CancellationToken ct = default)
    {
        var payment = await repo.GetByIdAsync(cmd.PaymentId, ct);
        if (payment is null) return;

        if (payment.Method != PaymentMethod.Online)
            throw new InvalidOperationException("PaymentIntent is applicable only for Online payments.");

        // Якщо PI вже створено — нічого не робимо (ідемпотентність)
        if (!string.IsNullOrWhiteSpace(payment.StripePaymentIntentId))
            return;

        // Destination-charge path when the business is onboarded with Stripe Connect.
        // Falls back to a plain platform charge when DestinationStripeAccountId is null.
        var result = payment.FundsFlow == FundsFlow.Destination
                     && !string.IsNullOrWhiteSpace(payment.DestinationStripeAccountId)
            ? await stripe.CreateDestinationPaymentIntentAsync(
                payment,
                payment.DestinationStripeAccountId!,
                ct: ct) // Task<StripePaymentIntentResult>
            : await stripe.CreatePaymentIntentAsync(payment, ct);

        payment.SetStripeSecrets(result.PaymentIntentId, result.ClientSecret);

        // Виставити TTL на підтвердження (наприклад, 15 хв)
        payment.SetExpiration( expiresAt: DateTime.UtcNow.AddMinutes(15));

        await repo.SaveChangesAsync(ct);
    }
}
```

Рисунок 4.13 - Вміст CreateStripePaymentIntentCommandHandler.cs (створення платіжного наміру)

Окрім онлайн-оплати, сервіс підтримує готівковий розрахунок під час доставки, який обробляється відповідною командою та узгоджується з боку кур’єра. Реалізовано також скасування платежу та повне або часткове повернення коштів із можливістю передавання ключа ідемпотентності для уникнення повторного виконання операції.

Обробка подій Stripe та webhook-повідомлень

Зміни стану платежів на стороні Stripe надходять у вигляді webhook-повідомлень, які опрацьовує окремий контролер. Перед обробкою виконується перевірка підпису повідомлення, а для уникнення повторної обробки однієї й тієї самої події використовується таблиця оброблених webhook-подій. Сервіс опрацьовує події успішного виконання та невдалого виконання платіжного наміру, його скасування, успішного повернення коштів та події, пов’язані з оскарженням платежів. Для виконання операцій, які мають здійснюватися поза основним потоком обробки запиту (зокрема узгодження стану платіжних намірів зі Stripe), реалізовано фоновий процес StripeTaskProcessor з обмеженою кількістю повторних спроб та визначеним вікном очікування.

Облік та виплати винагороди кур’єрам

Для роботи з винагородою кур’єрів у сервісі реалізовано окрему фінансову модель, що включає баланс кур’єра, нарахування за виконані замовлення та записи про виплати. Фоновий процес виплат періодично агрегує нарахування та формує перекази коштів на під’єднані акаунти кур’єрів через Stripe. Після завершення виплати публікується відповідна подія, яку споживають сервіс користувачів та сервіс замовлень для оновлення власних даних.

Надійність обробки повідомлень

Сервіс платежів є подієво-орієнтованим: він споживає події створення, скасування та доставки замовлення, а також події успішного виконання платежу. Для надійної передачі вихідних подій застосовано шаблон вихідної черги (Outbox Pattern): фоновий процес-публікатор зчитує повідомлення з вихідної черги та надсилає їх до брокера RabbitMQ, а в разі повторних невдалих спроб повідомлення переноситься до черги «мертвих» листів для подальшого аналізу [27]. Підключення до брокера повідомлень обгорнуто політикою повторних спроб на основі бібліотеки Polly, а під час обробки повідомлень обмежено кількість одночасно отримуваних повідомлень та задано допустиму кількість повторних спроб перед перенесенням повідомлення до черги «мертвих» листів [3]. Додатково реалізовано фоновий процес скасування платежів, час очікування яких перевищив допустиме значення.

Для постійного зберігання даних сервіс використовує СКБД PostgreSQL та Entity Framework Core, а міграції структури бази даних застосовуються автоматично під час запуску. Отже, сервіс платежів реалізує безпечну та надійну обробку фінансових операцій платформи, поєднуючи інтеграцію зі Stripe, шаблон CQRS, механізми ідемпотентності, шаблон вихідної черги та фонові процеси опрацювання подій, що забезпечує стійку роботу платіжної підсистеми навіть за умов розподіленої архітектури та можливих тимчасових збоїв.

Реалізація сервісу відстеження доставки

Сервіс відстеження доставки (TrackingService) реалізує функціонал визначення місцеположення кур'єрів та відображення процесу доставки в режимі реального часу. Він відповідає за зберігання адрес закладів і точок доставки, опрацювання географічних координат, побудову маршрутів, а також за передавання актуальної інформації про переміщення кур'єра клієнтам і закладам без перезавантаження сторінки. Саме цей сервіс забезпечує одну з ключових особливостей сучасних платформ доставки - прозоре відстеження стану замовлення на карті.

Доменна модель сервісу складається із сутностей локації та локації закладу. Для зберігання географічних даних використовується СКБД PostgreSQL із розширенням PostGIS, яке забезпечує підтримку просторових типів даних та виконання географічних запитів. Це дозволяє ефективно зберігати координати, обчислювати відстані та працювати з адресами закладів і точками доставки. Поточний стан відстеження кожного замовлення - останнє місцеположення кур'єра, етап та статус доставки - зберігається у вигляді знімка (snapshot) у кеші Redis, що забезпечує швидкий доступ до актуальних даних та зменшує навантаження на основне сховище [9].

Взаємодія в режимі реального часу

Для двостороннього обміну даними між сервером та клієнтом використовується технологія SignalR, реалізована у вигляді вузла реального часу (hub) відстеження кур'єра. Вузол надає клієнтам набір методів підписки та передавання даних: підписку на замовлення користувача відповідно до його ролі, підписку на конкретне замовлення з отриманням початкового знімка стану,

надсилання координат кур’єром та оновлення етапу доставки. Залежно від ролі користувача та переданих у токені даних клієнт долучається до відповідних груп розсилки - груп клієнта, закладу, кур’єра, групи доступних замовлень або групи конкретного замовлення. У відповідь сервер розсилає клієнтам повідомлення про оновлення знімка стану, зміну місцеположення кур’єра та зміну статусу замовлення.

Передавання координат кур’єром супроводжується перевіркою прив’язки до конкретного замовлення, а оновлення етапу доставки виконується з контролем допустимості переходів - зокрема, перехід до етапу руху до клієнта неможливий до підтвердження отримання замовлення в закладі. Оновлення статусів також ініціюється асинхронно: сервіс споживає подію зміни статусу замовлення, опубліковану сервісом замовлень, та розсилає відповідну інформацію всім зацікавленим групам клієнтів.

Автентифікація та обмеження навантаження

Особливістю вузла реального часу є підтримка двох схем автентифікації. Клієнт може під’єднатися як за основним JWT-токеном користувача, так і за короткочасним токеном відстеження, обмеженим конкретним замовленням, який видається сервісом замовлень. Це дозволяє надавати доступ до відстеження окремого замовлення без розкриття повного контексту користувача. Для захисту від надмірного навантаження до вузла реального часу та HTTP-кінцевих точок застосовано проміжний обробник обмеження частоти запитів за алгоритмом ковзного вікна.

Побудова маршрутів та геокодування

Для побудови маршрутів доставки сервіс використовує зовнішній сервіс маршрутизації OSRM. Сервіс RoutingService формує запит до OSRM та отримує інформацію про відстань, орієнтовну тривалість і геометрію маршруту, яка використовується для відображення шляху руху кур’єра на карті. Для перетворення адрес у координати та зворотного перетворення координат в адреси застосовується сервіс геокодування, що звертається до зовнішнього геолокаційного API.

Окрім роботи в режимі реального часу, сервіс надає повний набір операцій керування адресами та локаціями закладів, а також RPC-споживачів для отримання

інформації про локації закладів іншими сервісами. Таким чином, сервіс відстеження доставки реалізує комплексний функціонал роботи з геопросторовими даними та взаємодії в режимі реального часу, поєднуючи просторове сховище PostGIS, кешування стану в Redis, двосторонній обмін даними через SignalR та інтеграцію із зовнішніми сервісами маршрутизації й геокодування.

3.2 Реалізація клієнтської частини та взаємодії компонентів

Реалізація фронтенд частини системи

Клієнтська частина системи реалізована у вигляді односторінкового вебзастосунку (Single Page Application) із використанням бібліотеки React. Застосунок забезпечує взаємодію всіх категорій користувачів із платформою - клієнтів, представників закладів харчування та кур'єрів - і реалізує перегляд меню, формування замовлень, онлайн-оплату та відстеження доставки в режимі реального часу. Для збирання та запуску застосунку використовується інструмент Vite, а маршрутизація між сторінками реалізована засобами бібліотеки React Router без перезавантаження сторінки [7].

Компоненти інтерфейсу та сторінки реалізовано мовою JSX, тоді як клієнти для роботи з API та моделі даних описано мовою TypeScript, що дозволяє забезпечити статичну типізацію під час взаємодії із серверною частиною. Стилізація виконана за допомогою CSS-модулів, що забезпечує ізоляцію стилів на рівні окремих компонентів. Глобальний стан автентифікації та профілю користувача зберігається у контексті користувача, а токен доступу - у локальному сховищі браузера.

Взаємодія із серверною частиною

Уся взаємодія клієнтського застосунку із серверною частиною здійснюється виключно через API Gateway за допомогою бібліотеки axios. У застосунку налаштовано окремий екземпляр HTTP-клієнта з базовою адресою шлюзу, який автоматично передає cookie з refresh-токеном та додає до кожного запиту заголовок авторизації з токеном доступу. Перехоплювач відповідей розгортає стандартну обгортку відповіді з полями ознаки успіху, даних та повідомлення про помилку, а

у разі виникнення помилки виводить відповідне повідомлення користувачеві через сервіс сповіщень. Загальну схему взаємодії клієнтського застосунку із серверною частиною системи через API Gateway наведено у підрозділі 2.1 (рисунок 2.1).

Доступ до приватних сторінок застосунку обмежено за допомогою компонента захищеного маршруту, який перевіряє наявність токена доступу та не дозволяє неавторизованим користувачам відкривати сторінки, що потребують автентифікації.

Інтерфейси користувачів та функціонал реального часу

Інтерфейс застосунку адаптується до ролі користувача: реалізовано окремі домашні сторінки та бічні панелі для клієнтів, закладів і кур'єрів. Клієнт має змогу переглядати заклади та страви з пошуком і фільтрацією, формувати кошик та оформлювати замовлення, представник закладу - керувати меню та опрацьовувати замовлення, а кур'єр - приймати доступні замовлення та виконувати доставку. Для представників бізнесу реалізовано інформаційну панель із показниками діяльності та графіками, побудованими засобами бібліотеки recharts.

Для відстеження доставки в режимі реального часу клієнтський застосунок використовує клієнтську бібліотеку SignalR, яка під'єднується до вузла реального часу сервісу відстеження та отримує оновлення знімків стану, координат кур'єра й статусів замовлення. Відображення карти, адрес доставки та позначки кур'єра реалізовано засобами бібліотек leaflet та react-leaflet. У межах функціоналу відстеження реалізовано модальне вікно живого відстеження замовлення, обчислення орієнтовного часу прибуття з періодичним перерахунком, а також передавання координат кур'єром із використанням геолокаційного API браузера.

Для виконання онлайн-оплати у застосунку інтегровано бібліотеки Stripe, які забезпечують відображення вбудованої форми оплати та проходження процедури онбордингу за моделлю Stripe Connect. Збирання продакшн-версії застосунку виконується засобами Vite, а розгортання здійснюється на платформі Vercel із резервним публікуванням на GitHub Pages.

Отже, фронтенд частина системи реалізує повноцінний користувацький інтерфейс платформи доставки їжі з підтримкою різних ролей користувачів, захищеної взаємодії із серверною частиною через API Gateway, онлайн-оплати та

відстеження доставки в режимі реального часу. Застосування компонентного підходу, статичної типізації клієнтів API та сучасних бібліотек роботи з картами й платежами забезпечує зручність використання, швидкодію та підтримуваність клієнтської частини застосунку.

Реалізація взаємодії через RabbitMQ

Однією з ключових особливостей мікросервісної архітектури є організація взаємодії між незалежними сервісами. У розробленій системі взаємодія між компонентами здійснюється трьома способами: синхронною взаємодією клієнта із системою через API Gateway за протоколом HTTP, синхронними міжсервісними запитами за моделлю RPC поверх RabbitMQ та асинхронним обміном подіями через брокер повідомлень RabbitMQ. Останні два способи реалізовано саме засобами RabbitMQ, який виступає основою міжсервісної комунікації та дозволяє зменшити зв’язність компонентів системи [11].

Асинхронний обмін подіями

Для асинхронної взаємодії між сервісами використовується обмінник типу topic, через який публікуються події життєвого циклу замовлення та інші доменні події. Сервіс-видавець публікує подію, не маючи відомостей про конкретних споживачів, а зацікавлені сервіси підписуються на відповідні події та опрацьовують їх незалежно. Зокрема, після створення замовлення сервіс замовлень публікує подію створення, яку споживають сервіс платежів (для ініціалізації платежу) та сервіс відстеження (для створення початкового знімка стану). Аналогічно опрацьовуються події зміни статусу, отримання, доставки та скасування замовлення, а також події завершення виплати кур’єру. Такий підхід дозволяє реалізувати подієво-орієнтовану взаємодію без прямих синхронних викликів між сервісами.

Для уникнення повторної обробки тих самих повідомлень у разі їх повторної доставки використовується таблиця оброблених повідомлень, у якій фіксуються ідентифікатори вже опрацьованих подій. Це забезпечує ідемпотентність обробки вхідних подій та запобігає дублюванню результатів.

Синхронні запити за моделлю RPC

Окрім асинхронного обміну подіями, у системі реалізовано синхронні міжсервісні запити за моделлю віддаленого виклику процедур (RPC) поверх RabbitMQ із використанням механізму прямої зворотної відповіді (direct reply-to). Такий підхід застосовується для запитів, що потребують негайної відповіді, без прив'язки до HTTP-адрес сервісів. Наприклад, сервіс меню звертається до сервісу користувачів для отримання інформації про заклад, якому належить страва, а сервіс користувачів - до сервісу відстеження для отримання локацій закладів. Сервіс користувачів реалізує RPC-споживачів отримання облікових записів, а сервіс меню - отримання страв, зокрема й у пакетному режимі. Базовий клас RPC-споживача та спільний опис топології черг забезпечують ідемпотентне оголошення черг з узгодженим іменуванням.

Усі контракти подій та RPC-повідомлень винесено в окремий спільний пакет, що публікується у реєстрі пакетів GitHub Packages та використовується всіма сервісами. Це гарантує узгодженість структур даних під час взаємодії та унеможливлює розбіжності контрактів між сервісами. Загальну схему взаємодії компонентів через брокер повідомлень наведено на діаграмі компонентів у підрозділі 3.2 (рисунок 3.2). Таким чином, використання RabbitMQ дозволяє реалізувати як асинхронний обмін подіями, так і синхронні міжсервісні запити, що забезпечує гнучку, слабко зв'язану та відмовостійку взаємодію між компонентами системи.

Реалізація кешування та механізмів надійності

Для забезпечення високої продуктивності та стабільної роботи платформи в умовах розподіленої архітектури у системі реалізовано механізми кешування часто використовуваних даних та комплекс заходів підвищення надійності. Ці механізми дозволяють зменшити навантаження на бази даних, прискорити обробку запитів та гарантувати коректну роботу системи навіть за умов тимчасових збоїв окремих компонентів.

Кешування даних

Для кешування у системі використовується Redis - високопродуктивне сховище даних типу «ключ - значення», що зберігає інформацію в оперативній

пам’яті. Кешування застосовується у двох сервісах. У сервісі меню в кеші зберігаються найбільш популярні запити на отримання страв, що дозволяє обслуговувати їх без звернення до основної бази даних, тоді як PostgreSQL залишається основним джерелом даних. У сервісі відстеження доставки в кеші зберігаються знімки поточного стану доставки кожного замовлення - останнє місцеположення кур’єра, етап та статус, - що забезпечує швидкий доступ до актуальних даних під час роботи в режимі реального часу [10].

Механізми надійності та відмовостійкості

Для підвищення стійкості системи до тимчасових збоїв реалізовано низку механізмів. Взаємодія API Gateway із сервісами для ідемпотентних запитів захищена політикою повторних спроб з експоненційним зростанням затримки на основі бібліотеки Polly. Підключення кожного сервісу до брокера повідомлень також обгорнуто політикою повторних спроб, а для самого з’єднання увімкнено механізми автоматичного відновлення з’єднання й топології черг та періодичну перевірку активності з’єднання.

Для надійної передачі подій між сервісами застосовано шаблон вихідної черги (Outbox Pattern), реалізований у сервісах замовлень та платежів. Події зберігаються в базі даних у межах тієї самої транзакції, що й зміна стану, після чого надсилаються до брокера фоновим процесом. У разі вичерпання дозволеної кількості повторних спроб повідомлення переноситься до черги «мертвих» листів для подальшого аналізу. Для запобігання повторному виконанню операцій та дублюванню результатів використовуються ключі ідемпотентності - під час створення замовлень, виконання платіжних команд та обробки webhook-повідомлень. Додатково реалізовано захист внутрішніх запитів від повторного відтворення за допомогою часового вікна та порівняння ключа за сталий час, а також обмеження частоти запитів до сервісу відстеження [13].

Для контролю стану сервісів кожен з них надає три кінцеві точки перевірки працездатності: перевірку життєздатності процесу, її явний варіант та перевірку готовності, що додатково контролює доступність бази даних і брокера повідомлень. Ці кінцеві точки використовуються середовищем розгортання для моніторингу стану сервісів та керування їх готовністю до приймання запитів.

Таким чином, поєднання кешування даних у Redis із механізмами повторних спроб, автоматичного відновлення з'єднань, шаблону вихідної черги, ідемпотентності та перевірок працездатності забезпечує високу продуктивність і відмовостійкість розробленої системи.

3.3 Реалізація процесів розгортання та супроводу системи

Завершальним етапом розробки програмної системи є забезпечення її контейнеризації та автоматизації процесів збирання, тестування й розгортання. Оскільки система складається з великої кількості незалежних сервісів, ручне розгортання кожного з них було б трудомістким та схильним до помилок. Тому в проєкті реалізовано контейнеризацію всіх компонентів засобами Docker та автоматизований конвеєр неперервної інтеграції і неперервного розгортання (CI/CD).

Контейнеризація сервісів

Кожен сервіс системи має власний файл опису образу Docker, побудований за принципом багатоетапної збірки: на першому етапі виконується відновлення залежностей та компіляція застосунку, а на другому формується полегшений образ для запуску, що містить лише необхідні для роботи файли. Такий підхід зменшує розмір кінцевого образу та підвищує безпеку розгортання. Для локального запуску всієї системи передбачено файл оркестрації Docker Compose, який піднімає всі сервіси, бази даних, брокер повідомлень та кеш в єдиній мережі, а також окремий файл для запуску лише інфраструктурних компонентів під час гібридної розробки.

Для застосування міграцій структури баз даних використовуються окремі контейнери-мігратори, що дозволяє уникнути одночасного внесення змін до структури баз кількома екземплярами сервісів. Такий підхід забезпечує контрольоване та послідовне оновлення схеми даних незалежно від середовища виконання. Завдяки використанню Docker усі компоненти системи можуть бути швидко розгорнуті на будь-якому сервері або хмарній платформі, що спрощує супровід та подальше масштабування програмної системи.

Приклад файлу опису образу Docker наведено на рисунку 4.14.

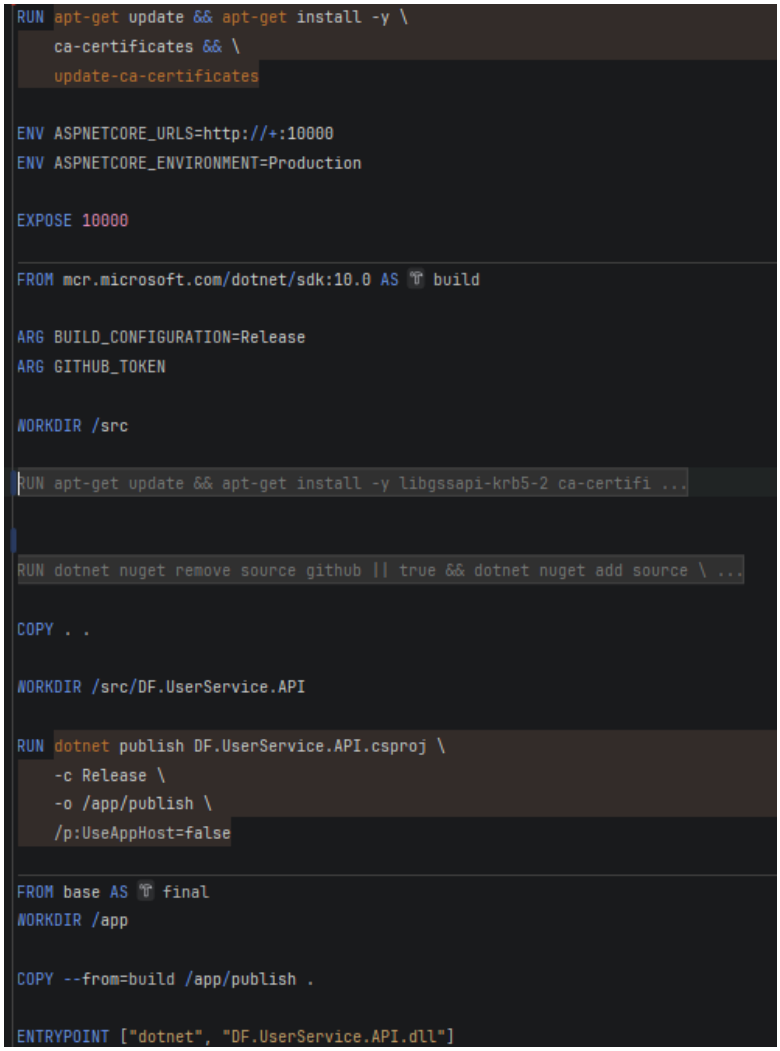


Рисунок 4.14 - Вміст Dockerfile сервісу (багатоетапна збірка образу)

Автоматизація збирання та розгортання

Процеси неперервної інтеграції та розгортання реалізовано засобами GitHub Actions. Для кожного сервісу та клієнтської частини створено окремий конвеєр, який запускається лише за наявності змін у відповідній частині репозиторію завдяки фільтрації за шляхами. Конвеєри серверних сервісів складаються з трьох послідовних етапів. На першому етапі виконується відновлення залежностей, збирання застосунку та запуск тестів. На другому етапі формується образ Docker, який публікується у реєстрі контейнерів GitHub Container Registry з відповідними мітками, причому публікація образу виконується лише для основної гілки репозиторію. На третьому етапі ініціюється розгортання сервісу шляхом виклику відповідного гачка розгортання, після чого середовище розгортання завантажує оновлений образ. Реалізацію конвеєра CI/CD наведено на рисунку 4.15.

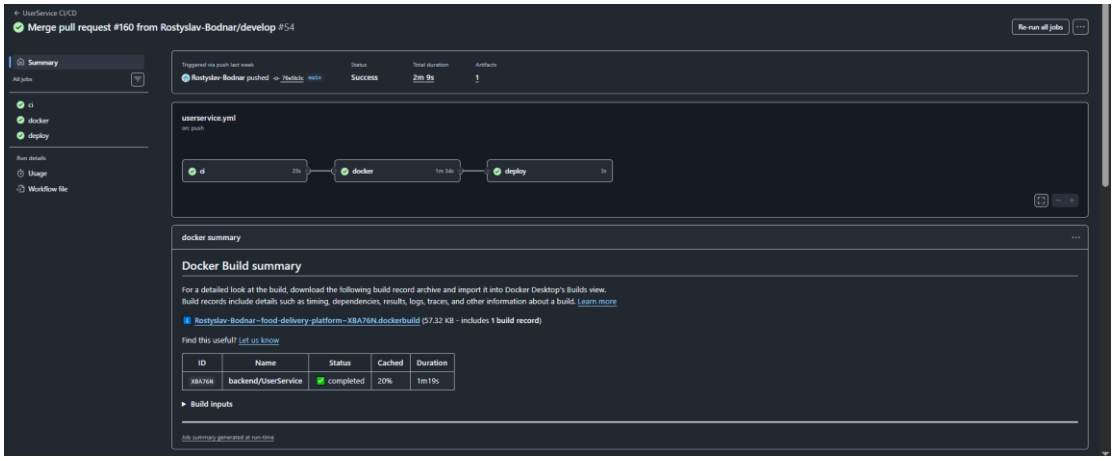


Рисунок 4.15 - Конфігурація конвеєра CI/CD у GitHub Actions

Конвеєр клієнтської частини виконує встановлення залежностей, збирання продакшн-версії застосунку та її розгортання на платформі Vercel із резервним публікуванням на GitHub Pages. Для уникнення одночасного виконання застарілих запусків конвеєрів використовуються групи паралельності, що скасовують попередні запуски для тієї самої гілки.

Розгортання компонентів системи виконується у хмарних середовищах. Загальну топологію розгортання наведено в таблиці 4.2.

Таблиця 4.2

Топологія розгортання компонентів системи

Компонент системи	**Середовище розгортання**
Клієнтський вебзастосунок	Vercel (резервно - GitHub Pages)
API Gateway та мікросервіси	Render (Docker-образи з реєстру GHCR)
Бази даних PostgreSQL (5 шт.)	Керована PostgreSQL - окрема база на кожен сервіс
Кеш Redis	Хмарний Redis (з підтримкою TLS)
Брокер повідомлень RabbitMQ	Хмарний брокер RabbitMQ
Зберігання зображень	Cloudinary
Обробка платежів	Stripe Connect

Отже, реалізована контейнеризація сервісів та автоматизований конвеєр CI/CD забезпечують узгоджене середовище виконання, автоматичне збирання, тестування й розгортання компонентів системи, а також спрощують її подальший супровід та масштабування.

4.4 Висновки по розділу

У третьому розділі описано практичну реалізацію програмного забезпечення онлайн платформи замовлення та доставки їжі, у межах якої розроблено всі основні компоненти системи відповідно до спроєктованої мікросервісної архітектури.

Реалізовано власний API Gateway, який виконує роль єдиної точки входу до системи та забезпечує маршрутизацію запитів, автентифікацію користувачів за допомогою JWT-токенів, підписування внутрішніх викликів за алгоритмом HMAC і централізовану обробку помилок із формуванням уніфікованої відповіді. Розроблено п'ять незалежних мікросервісів, кожен з яких відповідає за окрему предметну область: сервіс користувачів реалізує реєстрацію, автентифікацію, керування обліковими записами та інтеграцію зі Stripe Connect; сервіс меню - роботу зі стравами й категоріями з підтримкою пагінації, валідації та кешування; сервіс замовлень - повний життєвий цикл замовлення із застосуванням ідемпотентності та шаблону вихідної черги; сервіс платежів - обробку онлайн-оплат, повернень коштів і виплат кур'єрам через Stripe; сервіс відстеження доставки - визначення місцеположення кур'єра й оновлення стану доставки в режимі реального часу.

Клієнтську частину системи реалізовано у вигляді односторінкового вебзастосунку на основі React із рольовими інтерфейсами клієнтів, закладів харчування та кур'єрів, взаємодією із серверною частиною через API Gateway та відстеженням доставки в режимі реального часу засобами SignalR. Реалізовано асинхронну взаємодію між сервісами через брокер повідомлень RabbitMQ, а також механізми синхронних запитів за моделлю RPC.

Окрему увагу приділено забезпеченню надійності та масштабованості системи. Для цього використано механізми кешування, контейнеризацію сервісів за допомогою Docker та автоматизований конвеєр CI/CD для збирання і розгортання програмного забезпечення. Реалізовані рішення забезпечують стабільну взаємодію компонентів системи, спрощують її супровід та створюють основу для подальшого масштабування відповідно до зростання навантаження.

РОЗДІЛ 5. ТЕСТУВАННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

Після завершення розробки програмної системи важливим етапом є її тестування та аналіз отриманих результатів. Тестування дозволяє переконатися у відповідності системи сформованим вимогам, виявити та усунути дефекти, а також оцінити її продуктивність, надійність і масштабованість. З огляду на мікросервісну архітектуру платформи, тестування охоплює як окремі сервіси та їхні програмні інтерфейси, так і взаємодію між компонентами системи, роботу із зовнішніми сервісами та поведінку системи за умов навантаження.

5.1 Тестування функціональності та програмного інтерфейсу системи

Тестування функціональності системи

Функціональне тестування спрямоване на перевірку відповідності поведінки системи функціональним вимогам, сформованим на етапі аналізу. Метою цього виду тестування є підтвердження того, що кожна функціональна можливість платформи працює коректно та відповідає очікуванням користувачів. Тестування виконувалося за методом «чорної скриньки» на основі сценаріїв використання для кожної ролі клієнта, представника закладу харчування та кур’єра [17].

Для проведення функціонального тестування було сформовано набір тестових сценаріїв, що охоплюють основні бізнес-процеси системи: реєстрацію та автентифікацію користувачів, перемикання між обліковими записами, перегляд меню із пошуком та фільтрацією, формування кошика й оформлення замовлення, зміну статусів замовлення закладом, прийом і доставку замовлення кур’єром, а також скасування замовлення. Кожен сценарій передбачав виконання послідовності дій та порівняння фактичного результату з очікуваним.

Додатково було перевірено коректність взаємодії між мікросервісами через API Gateway, зокрема обробку авторизаційних токенів, підпис внутрішніх запитів та повернення уніфікованих відповідей. Окрему увагу приділено обробці помилкових ситуацій і некоректних вхідних даних для підтвердження стабільності системи в умовах нештатної роботи.

Перелік основних тестових сценаріїв та результати їх виконання наведено в таблиці 5.1.

Таблиця 5.1.

Результати функціонального тестування

Тестовий сценарій	**Очікуваний результат**	**Статус**
Реєстрація та вхід користувача	Створення акаунта, видача JWT-токена	Пройдено
Перемикання активного облікового запису	Перевипуск токена з новою роллю	Пройдено
Перегляд меню з пошуком і фільтром	Відображення відповідних страв	Пройдено
Формування кошика та оформлення замовлення	Створення замовлення зі статусом «підготовка»	Пройдено
Зміна статусу замовлення закладом	Коректний перехід між станами	Пройдено
Прийом і доставка замовлення кур’єром	Оновлення статусу до «доставлене»	Пройдено
Скасування замовлення	Переведення замовлення у стан «скасоване»	Пройдено

За результатами функціонального тестування всі основні сценарії використання було виконано успішно, що підтверджує відповідність реалізованої системи сформованим функціональним вимогам.

Тестування REST API

Оскільки взаємодія клієнтської частини із серверною здійснюється через програмний інтерфейс REST, окрему увагу було приділено тестуванню кінцевих точок API. Метою цього етапу є перевірка коректності обробки запитів, відповідності кодів статусу HTTP, формату відповіді та механізмів валідації й автентифікації. Для тестування API використовувалися інтерактивна документація Swagger, згенерована для сервісів на основі специфікації OpenAPI, а також інструмент Postman для формування та надсилання запитів [6, 23].

Під час тестування перевірялися такі аспекти: повернення коректних кодів статусу для успішних та помилкових запитів, відповідність структури відповіді уніфікованій обгортці, наявність заголовків пагінації для запитів зі списками, спрацювання валідації вхідних даних, а також відхилення неавторизованих запитів. Окрім ручного тестування, для сервісу користувачів реалізовано модульні тести

бізнес-логіки із використанням бібліотек xUnit, Moq та FluentAssertions і провайдера бази даних у пам’яті [1].

Додатково було проведено перевірку сценаріїв некоректної взаємодії з API, зокрема передачі неповних або некоректних параметрів, що дозволило оцінити стійкість системи до помилок введення. Також перевірено коректність роботи механізмів авторизації при доступі до захищених ресурсів та узгодженість відповідей між різними сервісами системи.

Результати тестування основних кінцевих точок API наведено в таблиці 5.2.

Таблиця 5.2.

Результати тестування кінцевих точок REST API

Метод та кінцева точка	**Очікуваний код**	**Результат**
POST /api/auth/register	200 OK	Відповідає
POST /api/auth/login	200 OK	Відповідає
GET /api/dish/customer	200 OK (з пагінацією)	Відповідає
POST /api/dish/create	200 OK	Відповідає
POST /api/order/create	200 OK	Відповідає
Запит без токена авторизації	401 Unauthorized	Відповідає
Запит із некоректними даними	400 Bad Request	Відповідає

Результати виконання модульних тестів сервісу користувачів на рисунку 5.1.

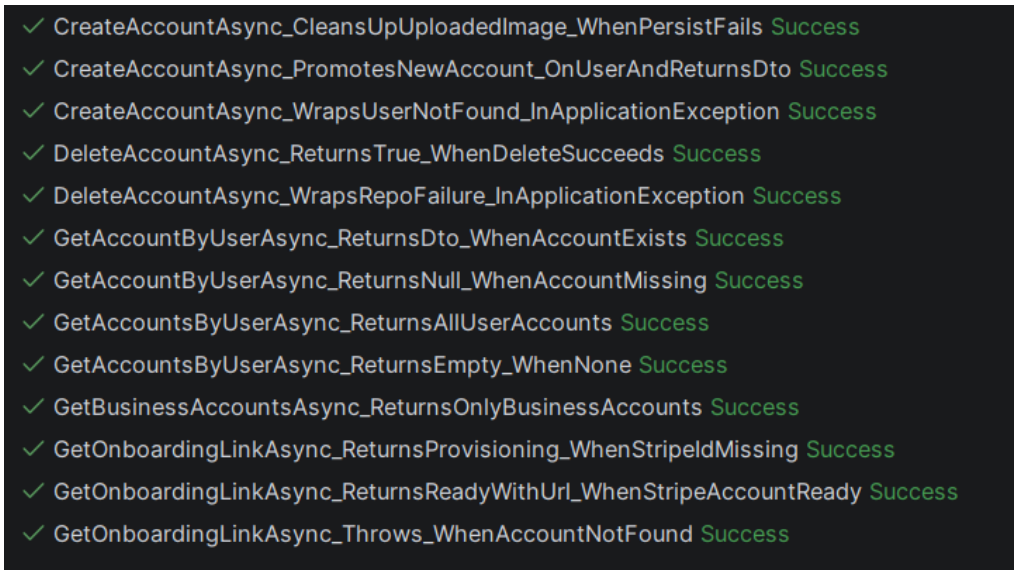


Рисунок 5.1 - Результати виконання модульних тестів сервісу користувачів

Результати тестування підтвердили коректну роботу програмного інтерфейсу системи, відповідність кодів статусу, формату відповіді та спрацювання механізмів валідації й автентифікації.

5.2 Тестування взаємодії компонентів системи

Тестування взаємодії компонентів системи.

Однією з ключових особливостей мікросервісної архітектури є взаємодія між незалежними сервісами, тому окремий етап тестування присвячено перевірці коректності цієї взаємодії. Метою інтеграційного тестування є підтвердження правильності асинхронного обміну подіями через брокер RabbitMQ, синхронних запитів за моделлю RPC, а також роботи механізмів забезпечення надійності - шаблону вихідної черги та ідемпотентності обробки повідомлень [3].

Під час тестування перевірялося, що після створення замовлення відповідна подія публікується у брокер та коректно опрацьовується сервісами платежів і відстеження. Окремо перевірялася стійкість системи до повторної доставки повідомлень: повторне надходження однієї й тієї самої події не призводило до дублювання результатів завдяки таблиці оброблених повідомлень. Для моделювання збоїв тимчасово зупинявся брокер повідомлень, після чого перевірялося, що події зберігаються у вихідній черзі та надсилаються після відновлення з’єднання. Стан черг та контролювався через панель керування RabbitMQ. Результати тестування взаємодії сервісів наведено в таблиці 5.3.

Таблиця 5.3.

Результати тестування взаємодії мікросервісів

Тестовий сценарій	**Результат**
Публікація події створення замовлення та її обробка сервісами	Подію успішно опрацьовано
Повторна доставка події (перевірка ідемпотентності)	Дублювання відхилено
Недоступність брокера повідомлень	Подію збережено та надіслано після відновлення
Синхронний RPC-запит інформації про страву та акаунт	Отримано коректну відповідь

Приклад моніторингу стану черг повідомлень через панель керування RabbitMQ наведено на рисунку 5.2.

Overview					Messages			Message rates		
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
rvchlohy	OrderCancelledEvent.queue	classic	D Args default rvchlohy-max-length	running	0	0	0	0.00/s	0.00/s	0.00/s
rvchlohy	OrderCancelledEvent.queue.dlq	classic	D Args default rvchlohy-max-length	running	0	0	0			
rvchlohy	OrderCancelledEvent.queue.retry	classic	D DLX DLK Args default rvchlohy-max-length	running	0	0	0			
rvchlohy	OrderCreatedEvent.queue	classic	D Args default rvchlohy-max-length	running	0	0	0	0.00/s	0.00/s	0.00/s
rvchlohy	OrderCreatedEvent.queue.dlq	classic	D Args default rvchlohy-max-length	running	0	0	0			
rvchlohy	OrderCreatedEvent.queue.retry	classic	D DLX DLK Args default rvchlohy-max-length	running	0	0	0			
rvchlohy	OrderDeliveredEvent.queue	classic	D Args default rvchlohy-max-length	running	0	0	0	0.00/s	0.00/s	0.00/s
rvchlohy	OrderDeliveredEvent.queue.dlq	classic	D Args default rvchlohy-max-length	running	0	0	0			
rvchlohy	OrderDeliveredEvent.queue.retry	classic	D DLX DLK Args default rvchlohy-max-length	running	0	0	0			
rvchlohy	PaymentSucceededEvent.queue	classic	D Args default rvchlohy-max-length	running	0	0	0	0.00/s	0.00/s	0.00/s
rvchlohy	PaymentSucceededEvent.queue.dlq	classic	D Args default rvchlohy-max-length	running	0	0	0			
rvchlohy	PaymentSucceededEvent.queue.retry	classic	D DLX DLK Args default rvchlohy-max-length	running	0	0	0			
rvchlohy	menu.dlq	classic	D Args default rvchlohy-max-length	running	0	0	0			
rvchlohy	menu.getdish	classic	DLX DLK Args default rvchlohy-max-length	running	0	0	0			
rvchlohy	menu.getdishes	classic	DLX DLK Args default rvchlohy-max-length	running	0	0	0			
rvchlohy	menu.getdishesbatch	classic	DLX DLK Args default rvchlohy-max-length	running	0	0	0	0.00/s	0.00/s	0.00/s
rvchlohy	order.dlq	classic	D Args default rvchlohy-max-length	running	0	0	0			
rvchlohy	orders.courier-payout-completed	classic	D DLX DLK Args default rvchlohy-max-length	running	0	0	0			
rvchlohy	orders.locationscreated	classic	D DLX DLK Args default rvchlohy-max-length	running	0	0	0	0.00/s	0.00/s	0.00/s
rvchlohy	orders.paymentsucceeded	classic	D DLX DLK Args default rvchlohy-max-length	running	0	0	0	0.00/s	0.00/s	0.00/s
rvchlohy	tracking.dlq	classic	D Args default rvchlohy-max-length	running	0	0	0			
rvchlohy	tracking.getbusinesslocations	classic	DLX DLK Args default rvchlohy-max-length	running	0	0	0			
rvchlohy	tracking.getbusinesslocationsbatch	classic	DLX DLK Args default rvchlohy-max-length	running	0	0	0	0.00/s	0.00/s	0.00/s
rvchlohy	tracking.getlocations	classic	DLX DLK Args default rvchlohy-max-length	running	0	0	0			
rvchlohy	tracking.getlocationsbatch	classic	DLX DLK Args default rvchlohy-max-length	running	0	0	0	0.00/s	0.00/s	0.00/s
rvchlohy	tracking.order-cancelled	classic	D DLX DLK Args default rvchlohy-max-length	running	0	0	0	0.00/s	0.00/s	0.00/s
rvchlohy	tracking.order-courier-paid	classic	D DLX DLK Args default rvchlohy-max-length	running	0	0	0			
rvchlohy	tracking.order-created	classic	D DLX DLK Args default rvchlohy-max-length	running	0	0	0	0.00/s	0.00/s	0.00/s
rvchlohy	tracking.order-delivered	classic	D DLX DLK Args default rvchlohy-max-length	running	0	0	0	0.00/s	0.00/s	0.00/s
rvchlohy	tracking.order-picked-up	classic	D DLX DLK Args default rvchlohy-max-length	running	0	0	0	0.00/s	0.00/s	0.00/s
rvchlohy	tracking.order-status-changed	classic	D DLX DLK Args default rvchlohy-max-length	running	0	0	0	0.00/s	0.00/s	0.00/s
rvchlohy	user.dlq	classic	D Args default rvchlohy-max-length	running	0	0	0			
rvchlohy	user.getaccount	classic	DLX DLK Args default rvchlohy-max-length	running	0	0	0			
rvchlohy	user.getbusinessaccountsbatch	classic	DLX DLK Args default rvchlohy-max-length	running	0	0	0	0.00/s	0.00/s	0.00/s
rvchlohy	user.getbussinessaccount	classic	DLX DLK Args default rvchlohy-max-length	running	0	0	0	0.00/s	0.00/s	0.00/s
rvchlohy	user.getcourieraccount	classic	DLX DLK Args default rvchlohy-max-length	running	0	0	0	0.00/s	0.00/s	0.00/s
rvchlohy	user.getcourieraccountsbatch	classic	DLX DLK Args default rvchlohy-max-length	running	0	0	0	0.00/s	0.00/s	0.00/s
rvchlohy	user.getcustomeraccount	classic	DLX DLK Args default rvchlohy-max-length	running	0	0	0	0.00/s	0.00/s	0.00/s
rvchlohy	user.getcustomeraccountsbatch	classic	DLX DLK Args default rvchlohy-max-length	running	0	0	0			

Рисунок 5.2 - Панель керування RabbitMQ зі станом черг повідомлень

Результати тестування підтвердили коректність асинхронної та синхронної взаємодії між сервісами, а також ефективність реалізованих механізмів забезпечення надійності обробки повідомлень.

Тестування системи онлайн-платежів.

Оскільки система працює з фінансовими операціями, окрему увагу приділено тестуванню підсистеми онлайн-платежів. Метою цього етапу є перевірка коректності створення та підтвердження платежів, обробки повернень коштів і готівкових розрахунків, а також правильності опрацювання webhook-повідомлень платіжної системи. Тестування виконувалося у тестовому режимі Stripe із

використанням спеціальних тестових карток, що дозволяють імітувати різні результати оплати без реальних фінансових операцій [14].

Під час тестування перевірялися сценарії успішної оплати, відмови у проведенні платежу, скасування платежу, повного та часткового повернення коштів, а також готівкового розрахунку під час доставки. Окремо перевірялася обробка webhook-повідомлень про зміну стану платежу та ідемпотентність їх опрацювання за допомогою таблиці оброблених подій. Також підтверджено коректність формування платіжного наміру із зазначенням комісії платформи та переказу коштів на під'єднаний акаунт закладу за моделлю Stripe Connect.

Результати тестування підсистеми платежів наведено в таблиці 5.4.

Таблиця 5.4.

Результати тестування системи онлайн-платежів

Тестовий сценарій	Очікуваний результат	Статус
Успішна оплата тестовою карткою	Платіж підтверджено	Пройдено
Оплата карткою з відмовою	Платіж відхилено	Пройдено
Скасування платежу	Платіж скасовано	Пройдено
Повне повернення коштів	Кошти повернено повністю	Пройдено
Часткове повернення коштів	Повернено зазначену суму	Пройдено
Готівковий розрахунок під час доставки	Розрахунок зафіксовано	Пройдено

Приклад тестування онлайн-оплати у тестовому режимі Stripe наведено на рисунку 5.3.

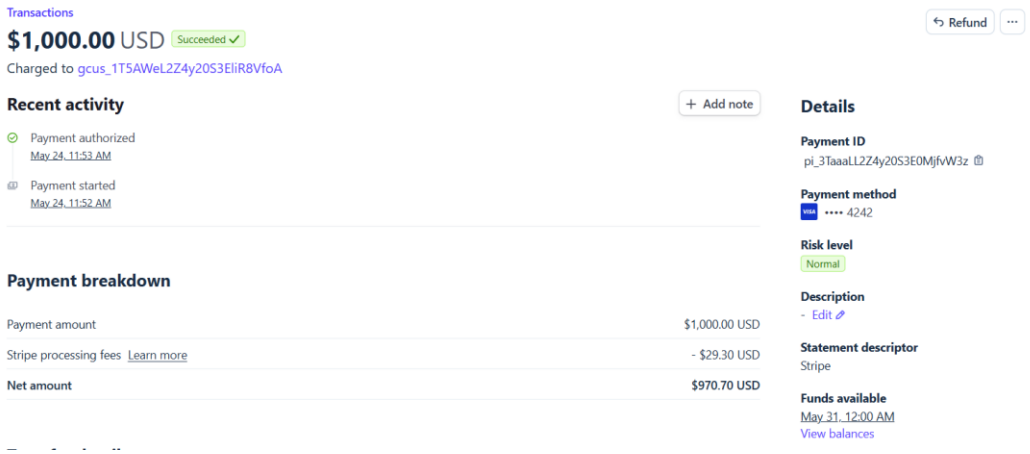


Рисунок 5.3 - Тестування онлайн-оплати у тестовому режимі Stripe

Результати тестування підтвердили коректну та безпечну роботу підсистеми платежів, включно з обробкою повернень коштів, готівкових розрахунків та webhook-повідомлень платіжної системи.

Тестування системи відстеження доставки.

Підсистема відстеження доставки забезпечує взаємодію в режимі реального часу, тому її тестування має свої особливості. Метою цього етапу є перевірка коректності встановлення з'єднання з вузлом реального часу, передавання координат кур'єра, отримання клієнтами оновлень у режимі реального часу, контролю переходів між етапами доставки та побудови маршрутів [13, 24].

Під час тестування перевірялося встановлення з'єднання з вузлом реального часу за основним токеном користувача та за короткочасним токеном відстеження, прив'язаним до конкретного замовлення. Імітувалося періодичне надсилання координат кур'єром, після чого перевірялося, що клієнти отримують оновлення місцеположення та знімків стану доставки. Окремо перевірявся контроль допустимості переходів між етапами доставки, побудова маршруту через зовнішній сервіс маршрутизації, а також відновлення з'єднання після його тимчасового розриву. Додатково оцінювалася поведінка системи при втраті пакетів даних і короткочасних затримках у мережі, що дозволило більш повно перевірити її стійкість до реальних умов експлуатації. Також було перевірено коректність синхронізації станів доставки між усіма підключеними клієнтами.

Результати тестування підсистеми відстеження наведено в таблиці 5.5.

Таблиця 5.5.

Результати тестування системи відстеження доставки

Тестовий сценарій	Результат
Підключення до вузла реального часу за токеном	З'єднання встановлено
Надсилання координат кур'єра	Клієнт отримує оновлення
Перехід між етапами доставки	Коректний перехід
Спроба недопустимого переходу етапу	Перехід відхилено
Відновлення з'єднання після розриву	З'єднання відновлено автоматично
Побудова маршруту доставки	Маршрут сформовано коректно

Приклад відстеження переміщення кур’єра на карті в режимі реального часу наведено на рисунку 5.4.

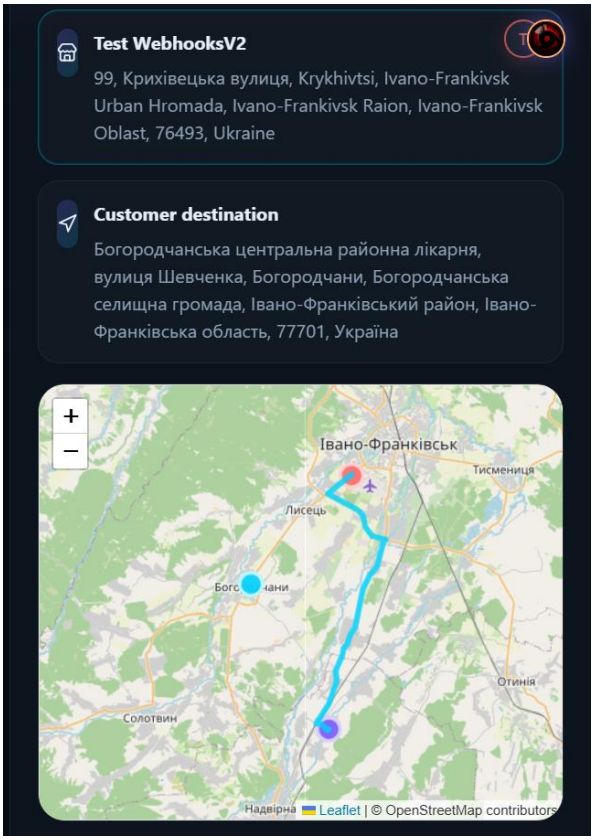


Рисунок 5.4 - Відстеження кур’єра на карті в режимі реального часу

Результати тестування підтвердили коректну роботу підсистеми відстеження доставки, своєчасне оновлення даних у режимі реального часу та надійність з’єднання між сервером і клієнтом.

5.3 Аналіз отриманих результатів

За результатами проведеного тестування виконано узагальнений аналіз відповідності розробленої системи поставленим вимогам. Функціональне тестування підтвердило коректну роботу всіх основних бізнес-сценаріїв платформи, тестування програмного інтерфейсу - відповідність кодів статусу, формату відповіді та механізмів валідації й автентифікації, а інтеграційне тестування - надійність асинхронної та синхронної взаємодії між сервісами. Тестування підсистем онлайн-платежів і відстеження доставки засвідчило безпечну

обробку фінансових операцій та коректну роботу механізмів реального часу, а навантажувальне тестування - прийнятну продуктивність і здатність системи до масштабування.

Узагальнені результати перевірки відповідності системи сформованим вимогам наведено в таблиці 5.7.

Таблиця 5.7.

Відповідність системи сформованим вимогам

Вимога до системи	**Стан реалізації**
Реєстрація та автентифікація користувачів	Виконано
Підтримка ролей користувачів	Виконано
Перегляд та керування меню	Виконано
Оформлення та обробка замовлень	Виконано
Онлайн-оплата та повернення коштів	Виконано
Відстеження доставки в режимі реального часу	Виконано
Асинхронна взаємодія між сервісами	Виконано
Безпека даних та контроль доступу	Забезпечено
Контейнеризація та автоматизація розгортання	Виконано

Проведений аналіз показав, що всі функціональні та нефункціональні вимоги, сформовані на етапі аналізу предметної області, реалізовано в повному обсязі. Розроблена система є працездатною, надійною та масштабованою платформою онлайн замовлення та доставки їжі. Отримані результати підтверджують досягнення мети роботи та коректність обраних архітектурних і технологічних рішень.

4.4 Висновки по розділу

У четвертому розділі виконано комплексне тестування розробленої онлайн платформи замовлення та доставки їжі, що дозволило підтвердити її відповідність сформованим функціональним і нефункціональним вимогам.

Функціональне тестування, проведене за методом «чорної скриньки» на основі сценаріїв використання для всіх ролей користувачів, підтвердило коректну

роботу основних бізнес-процесів системи - реєстрації та автентифікації, перегляду меню, оформлення й обробки замовлень, зміни статусів і доставки. Тестування програмного інтерфейсу REST із використанням Swagger, Postman та модульних тестів на основі xUnit засвідчило відповідність кодів статусу HTTP, формату відповіді, а також коректне спрацювання механізмів валідації вхідних даних та автентифікації.

Інтеграційне тестування підтвердило правильність взаємодії між мікросервісами - асинхронного обміну подіями через RabbitMQ та синхронних запитів за моделлю RPC, а також ефективність реалізованих механізмів надійності, зокрема шаблону вихідної черги та ідемпотентності обробки повідомлень. Тестування підсистеми онлайн-платежів у тестовому режимі Stripe засвідчило коректну та безпечну обробку платежів, повернень коштів, готівкових розрахунків і webhook-повідомлень, а тестування підсистеми відстеження доставки - надійну роботу механізмів реального часу та своєчасне оновлення даних про переміщення кур'єра.

За результатами проведеного тестування виконано узагальнений аналіз отриманих результатів, який показав, що всі основні функціональні можливості системи працюють коректно, а реалізовані механізми взаємодії, безпеки та надійності відповідають поставленим вимогам. Це підтверджує працездатність розробленої платформи та коректність обраних архітектурних і технологічних рішень, а отже - досягнення мети бакалаврської роботи.

Окрему увагу під час тестування було приділено перевірці роботи системи в умовах виникнення помилок і нестандартних сценаріїв використання. Було перевірено обробку некоректних вхідних даних, відсутність необхідних прав доступу, спроби виконання несанкціонованих операцій, а також відновлення роботи після тимчасової недоступності окремих сервісів. Отримані результати підтвердили стабільність функціонування платформи та коректність реалізованих механізмів обробки виняткових ситуацій. Це свідчить про високу якість розробленого програмного забезпечення та практичну придатність створеної платформи для використання в реальних умовах.

ВИСНОВКИ

У бакалаврській роботі розроблено онлайн-платформу замовлення та доставки їжі на основі мікросервісної архітектури, що забезпечує взаємодію між клієнтами, закладами харчування та кур’єрами, підтримує онлайн-оплати та відстеження доставки в режимі реального часу. Поставлену мету роботи досягнуто, а всі визначені завдання виконано.

У процесі виконання роботи проведено аналіз предметної області та сучасних сервісів доставки їжі, визначено їхні переваги й недоліки, а також обґрунтовано вибір мікросервісної архітектури. На основі проведеного аналізу сформовано функціональні та нефункціональні вимоги до програмної системи.

Під час проєктування розроблено архітектуру системи, що складається з незалежних сервісів користувачів, меню, замовлень, платежів і відстеження доставки, об’єднаних через API Gateway. Спроєктовано структуру баз даних, механізми автентифікації та авторизації, а також взаємодію між компонентами системи.

У практичній частині реалізовано основні компоненти платформи: API Gateway, набір мікросервісів, JWT-автентифікацію, асинхронну взаємодію через RabbitMQ, інтеграцію зі Stripe для онлайн-оплат, SignalR для відстеження доставки та Redis для кешування даних. Клієнтську частину реалізовано як односторінковий вебзастосунок на базі React.

Для автоматизації розгортання виконано контейнеризацію сервісів за допомогою Docker та налаштовано CI/CD-конвеєр на основі GitHub Actions. Проведене тестування підтвердило коректну роботу основних функцій системи та надійність міжсервісної взаємодії.

Практичне значення роботи полягає у створенні працездатної платформи доставки їжі, яка може бути використана для автоматизації діяльності сервісів доставки та закладів харчування. Отримані результати підтверджують ефективність застосованих архітектурних і технологічних рішень та можуть бути основою для подальшого розвитку системи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. **Sommerville I.** Software Engineering. – 10th ed. – Pearson, 2015.

2. **Newman S.** Building Microservices: Designing Fine-Grained Systems. – 2nd ed. – O'Reilly Media, 2021.

3. **Richardson C.** Microservices Patterns. – Manning Publications, 2018.

4. **Fowler M.** Patterns of Enterprise Application Architecture. – Addison-Wesley, 2002.

5. **Kleppmann M.** Designing Data-Intensive Applications. – O'Reilly Media, 2017.

6. **ASP.NET Core Documentation.** [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/aspnet/core, вільний. – Дата звернення: 28.02.2026.

7. **React Documentation.** [Електронний ресурс]. – Режим доступу: https://react.dev, вільний. – Дата звернення: 05.03.2026.

8. **PostgreSQL Documentation.** [Електронний ресурс]. – Режим доступу: https://www.postgresql.org/docs, вільний. – Дата звернення: 10.03.2026.

9. **PostGIS Documentation.** Spatial and Geographic Objects for PostgreSQL. [Електронний ресурс]. – Режим доступу: https://postgis.net/documentation, вільний. – Дата звернення: 12.03.2026.

10. **Redis Documentation.** [Електронний ресурс]. – Режим доступу: https://redis.io/docs, вільний. – Дата звернення: 14.03.2026.

11. **RabbitMQ Documentation.** [Електронний ресурс]. – Режим доступу: https://www.rabbitmq.com/documentation.html, вільний. – Дата звернення: 15.03.2026.

12. **Docker Documentation.** [Електронний ресурс]. – Режим доступу: https://docs.docker.com, вільний. – Дата звернення: 18.03.2026.

13. **SignalR Documentation.** [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/aspnet/core/signalr/introduction, вільний. – Дата звернення: 22.10.2025.

14. **Stripe API Documentation.** [Електронний ресурс]. – Режим доступу: https://docs.stripe.com/api, вільний. – Дата звернення: 22.03.2026.

15. **JWT Introduction.** [Електронний ресурс]. – Режим доступу: https://jwt.io/introduction, вільний. – Дата звернення: 12.04.2026.

16. **Cloudinary Documentation.** [Електронний ресурс]. – Режим доступу: https://cloudinary.com/documentation, вільний. – Дата звернення: 27.04.2026

17. **ISO/IEC 25010:2011.** Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models.

18. **Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software**. – Addison-Wesley, 2003.

19. **Vernon V. Implementing Domain-Driven Design.** – Addison-Wesley, 2013.

20. **Nygard M. Release It!: Design and Deploy Production-Ready Software**. – 2nd ed. – Pragmatic Bookshelf, 2018.

21. **Microsoft. Entity Framework Core Documentation**. [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/ef/core, вільний. – Дата звернення: 15.04.2026.

22. **Microsoft. ASP.NET Core Authentication and Authorization Documentation.** [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/aspnet/core/security, вільний. – Дата звернення: 15.04.2026.

23. **OpenAPI Specification.** [Електронний ресурс]. – Режим доступу: https://swagger.io/specification, вільний. – Дата звернення: 20.04.2026.

24. **Microsoft. SignalR Hubs Documentation**. [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/aspnet/core/signalr/hubs, вільний. – Дата звернення: 20.04.2026.

25. **Microsoft. Distributed Caching in ASP.NET Core**. [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/aspnet/core/performance/caching/distributed, вільний. – Дата звернення: 21.04.2026.

26. **Fowler M. Event-Driven Architecture**. [Електронний ресурс]. – Режим доступу: https://martinfowler.com/articles/201701-event-driven.html, вільний. – Дата звернення: 22.04.2026.

27. **Microsoft. Background Tasks with Hosted Services in ASP.NET Core**. [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/aspnet/core/fundamentals/host/hosted-services, вільний. – Дата звернення: 23.04.2026.

28. **OWASP Foundation. OWASP Top 10 Web Application Security Risks**. [Електронний ресурс]. – Режим доступу: https://owasp.org/www-project-top-ten, вільний. – Дата звернення: 25.04.2026.

29. **PostgreSQL Geographic Information Systems**. [Електронний ресурс]. – Режим доступу: https://postgis.net/workshops, вільний. – Дата звернення: 26.04.2026.

30. **Google Maps Platform Documentation**. [Електронний ресурс]. – Режим доступу: https://developers.google.com/maps/documentation, вільний. – Дата звернення: 26.04.2026.

ДОДАТОК А

Демонстрація результатів веброзробки

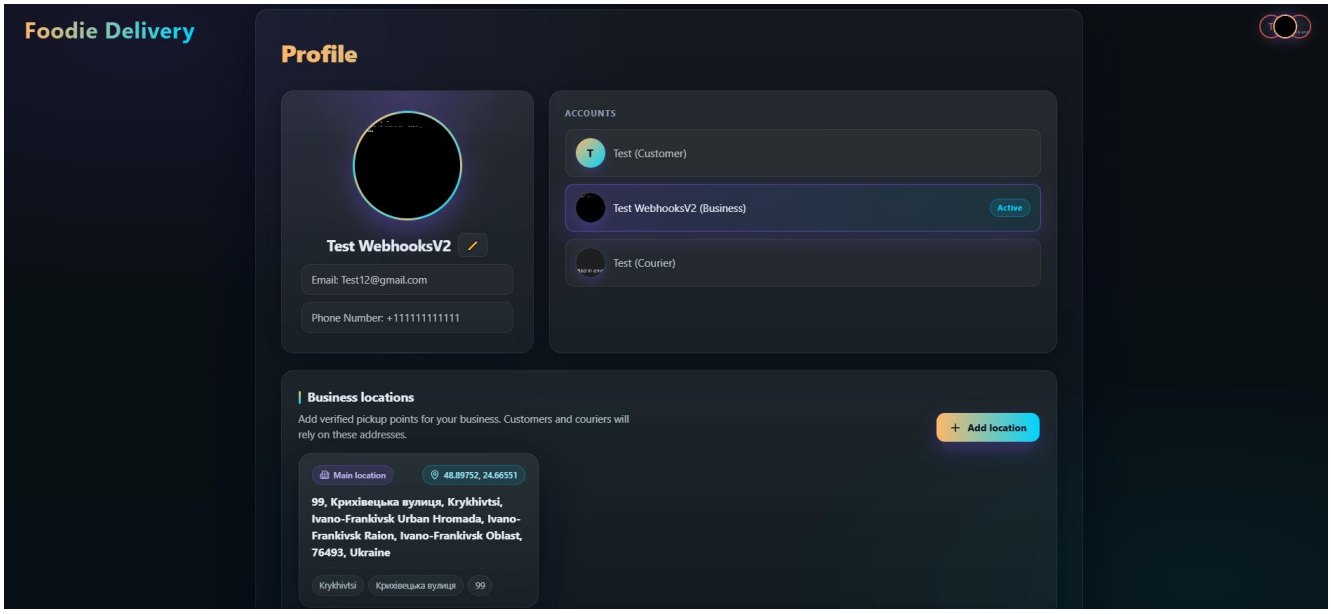


Рисунок А.1 – Сторінка профілю бізнес акаунту

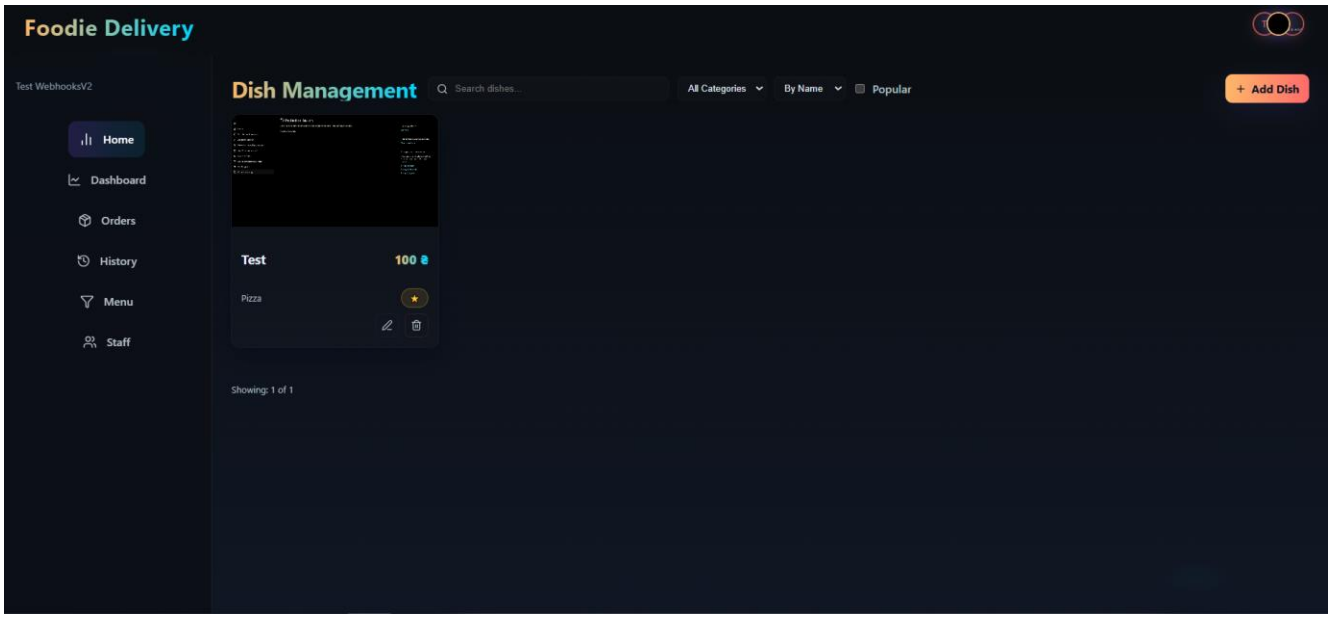


Рисунок А.2 – Сторінка страв бізнес акаунту

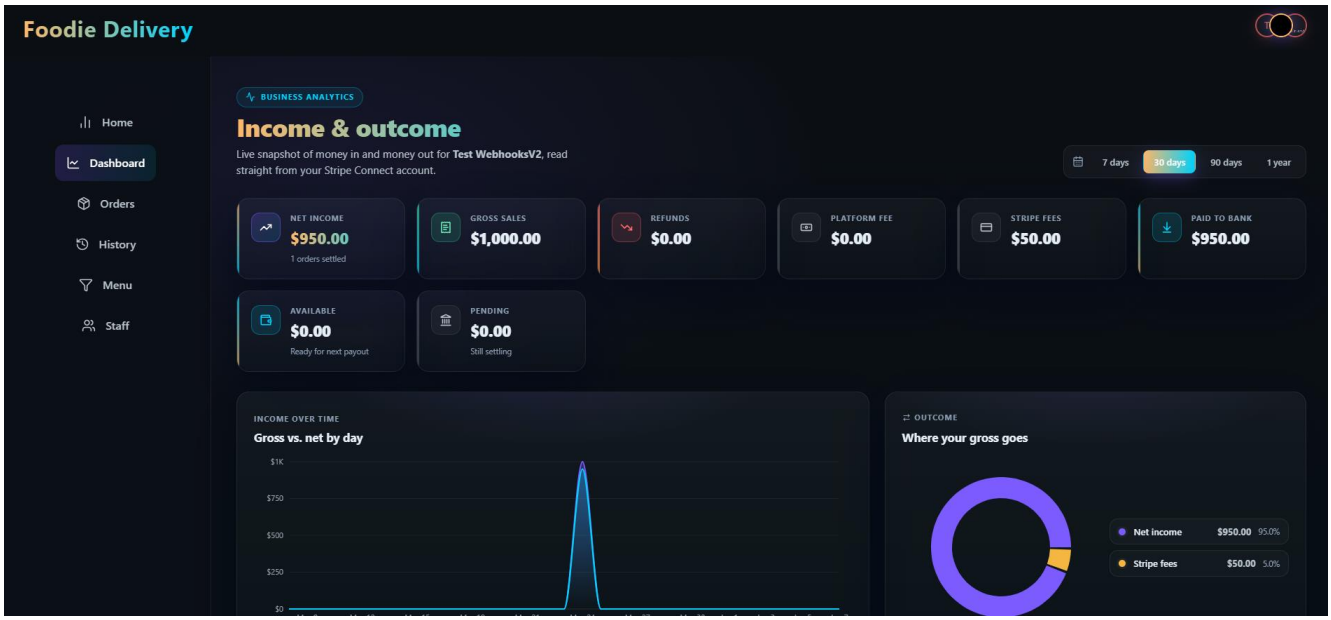


Рисунок А.3 – Сторінка менеджменту транзакцій бізнесу

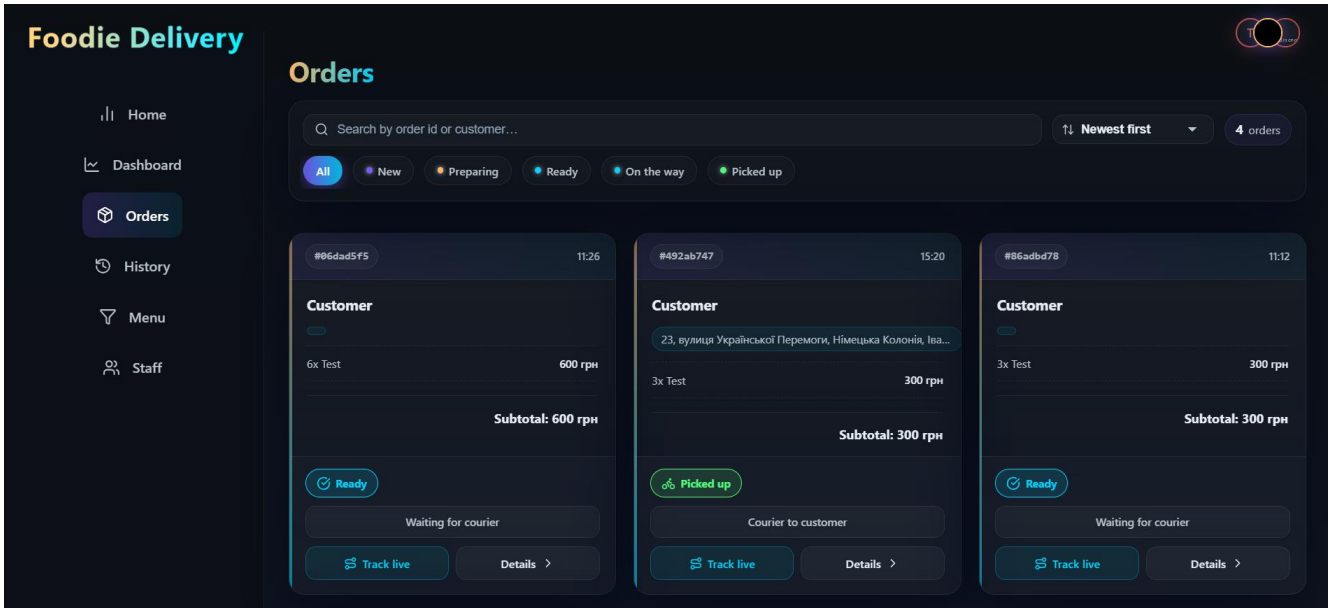


Рисунок А.4 – Сторінка активних замовлень бізнесу

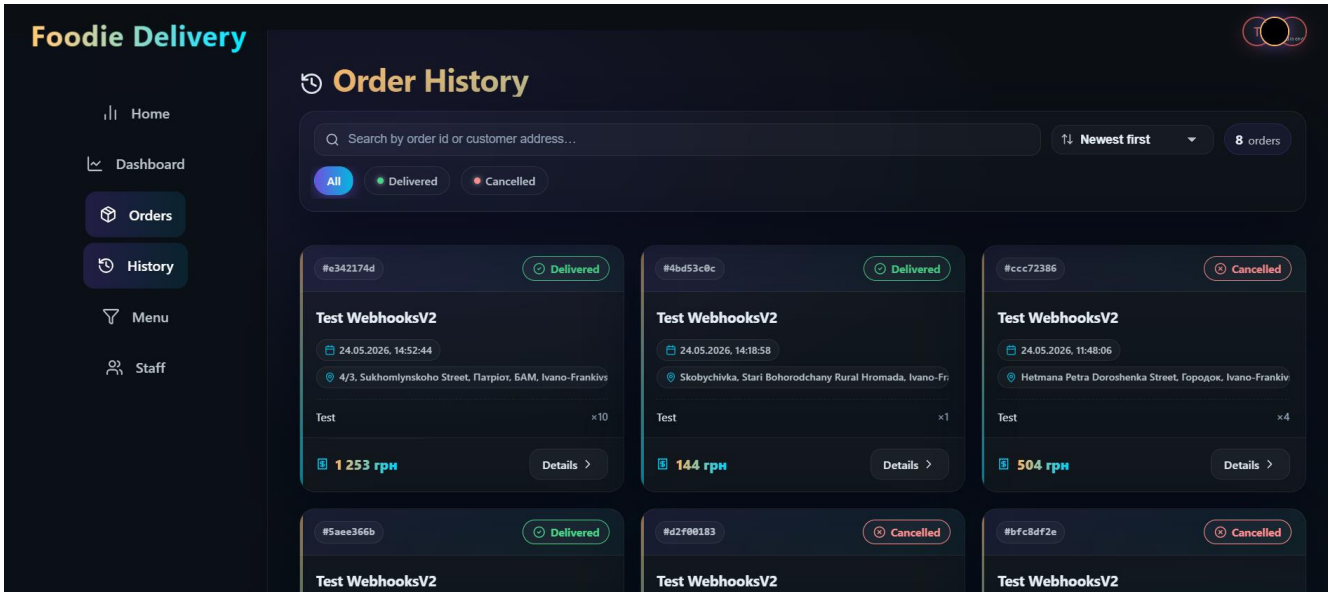


Рисунок А.5 – Сторінка історії замовлень бізнесу

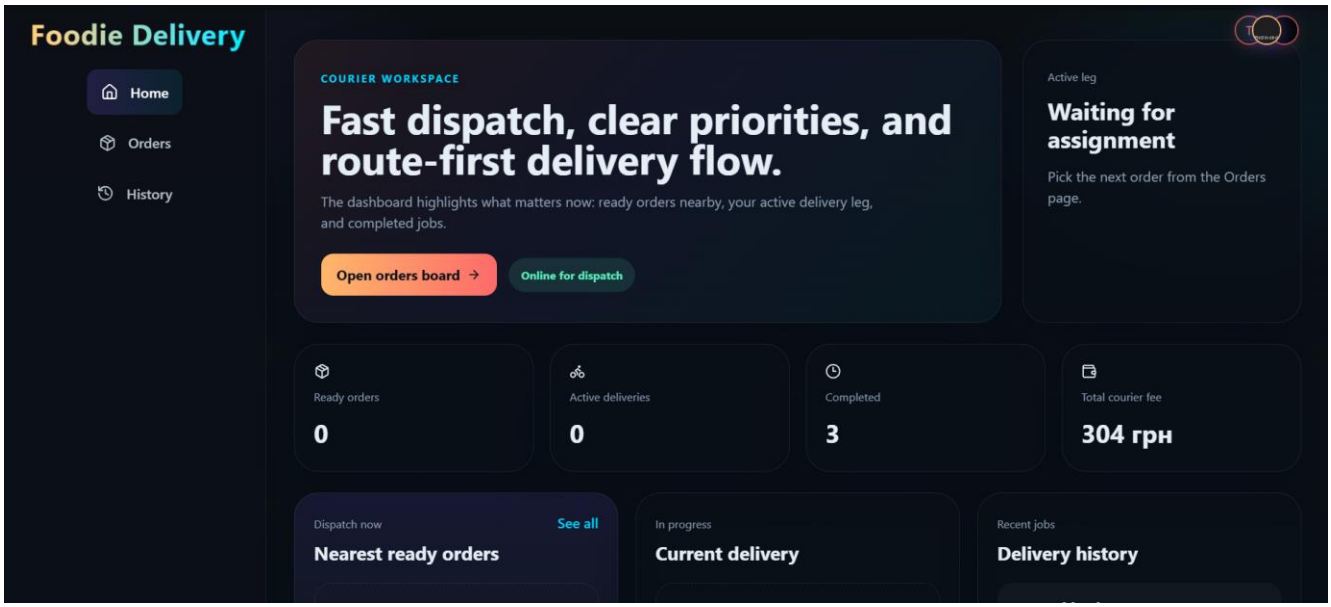


Рисунок А.6 – Головна сторінка кур’єр-акаунту

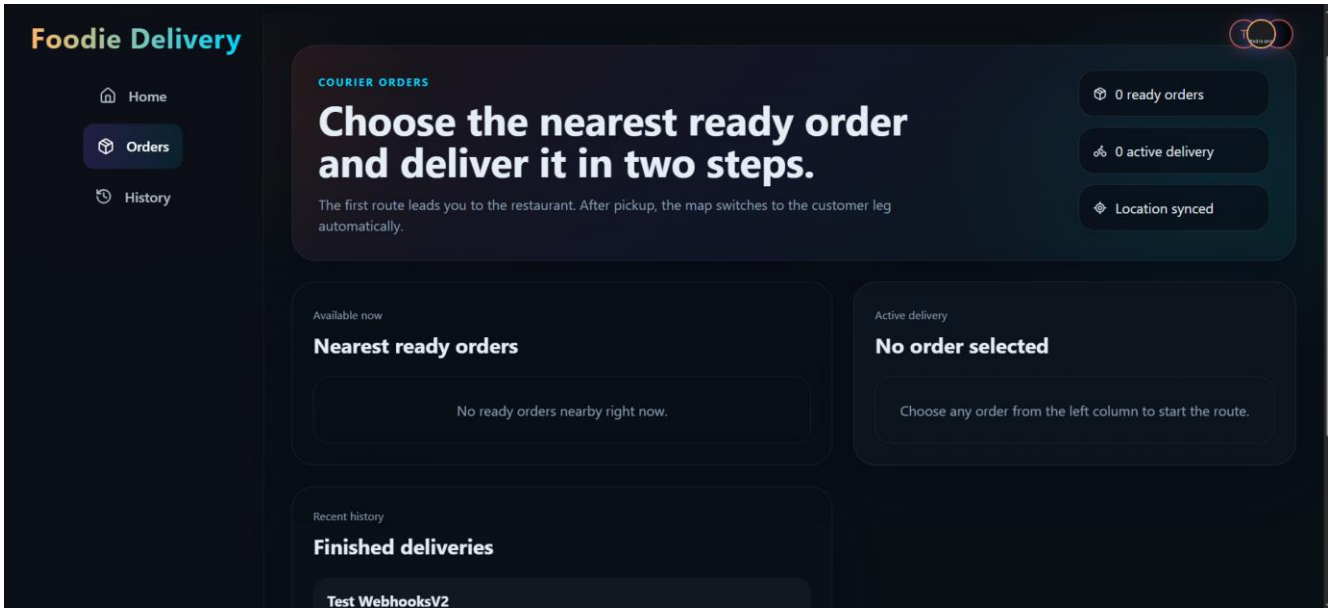


Рисунок А.7 – Сторінка замовлень кур’єр-акаунту

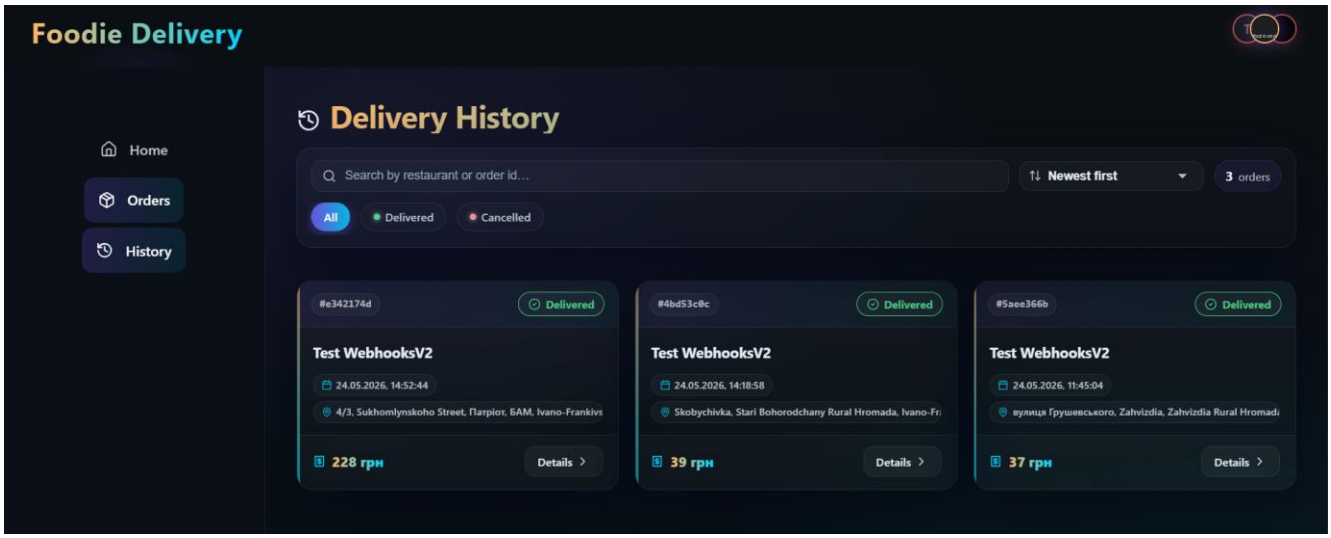


Рисунок А.8 – Сторінка історії доставлених замовлень кур’єр-акаунту

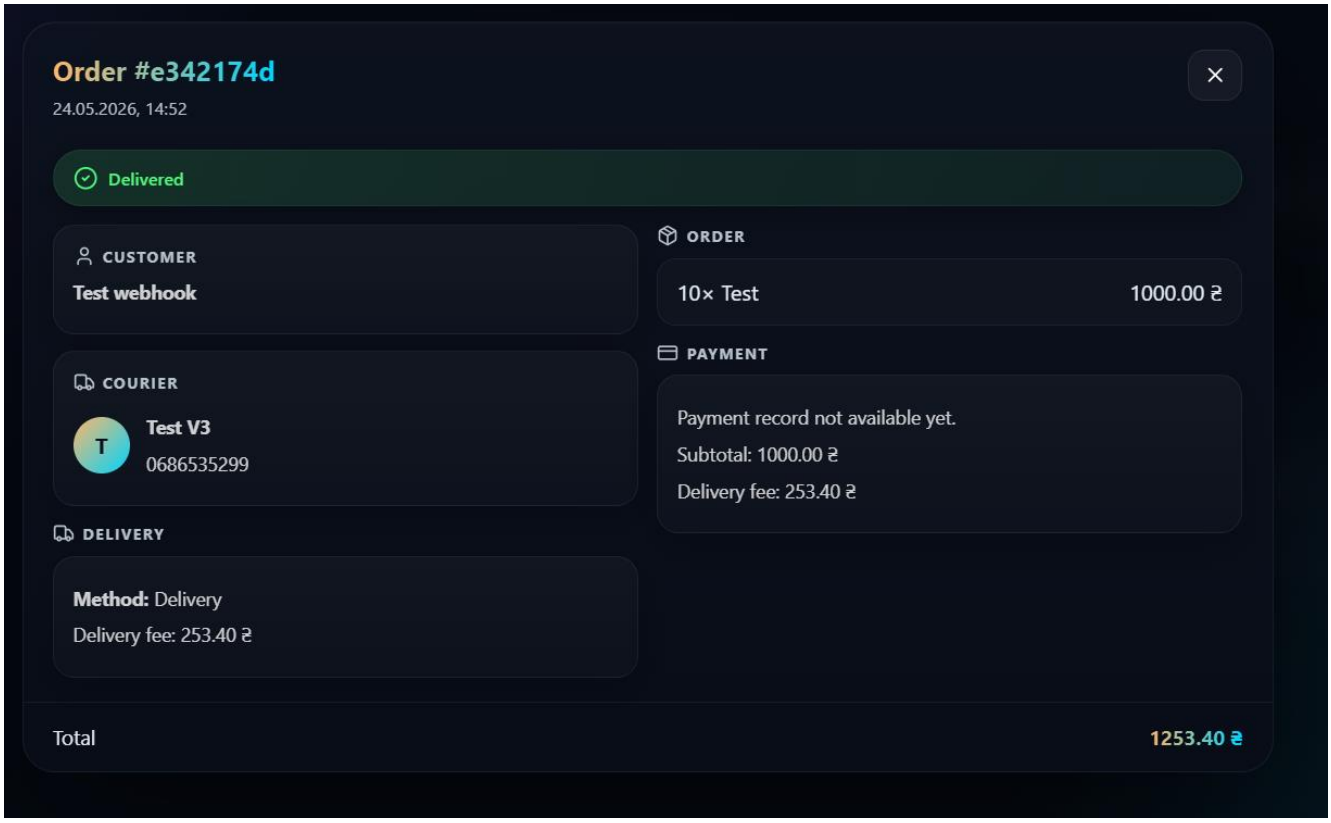


Рисунок А.9 – Форма з деталями замовлення

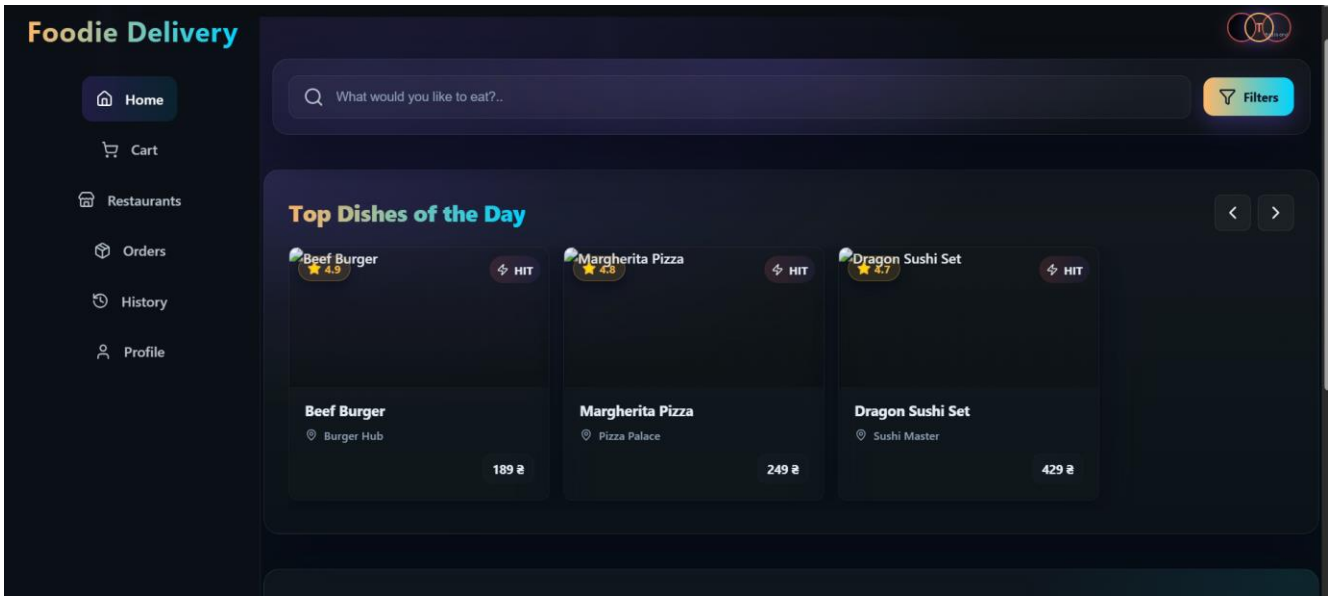


Рисунок А.10 – Головна сторінка клієнт-акаунту

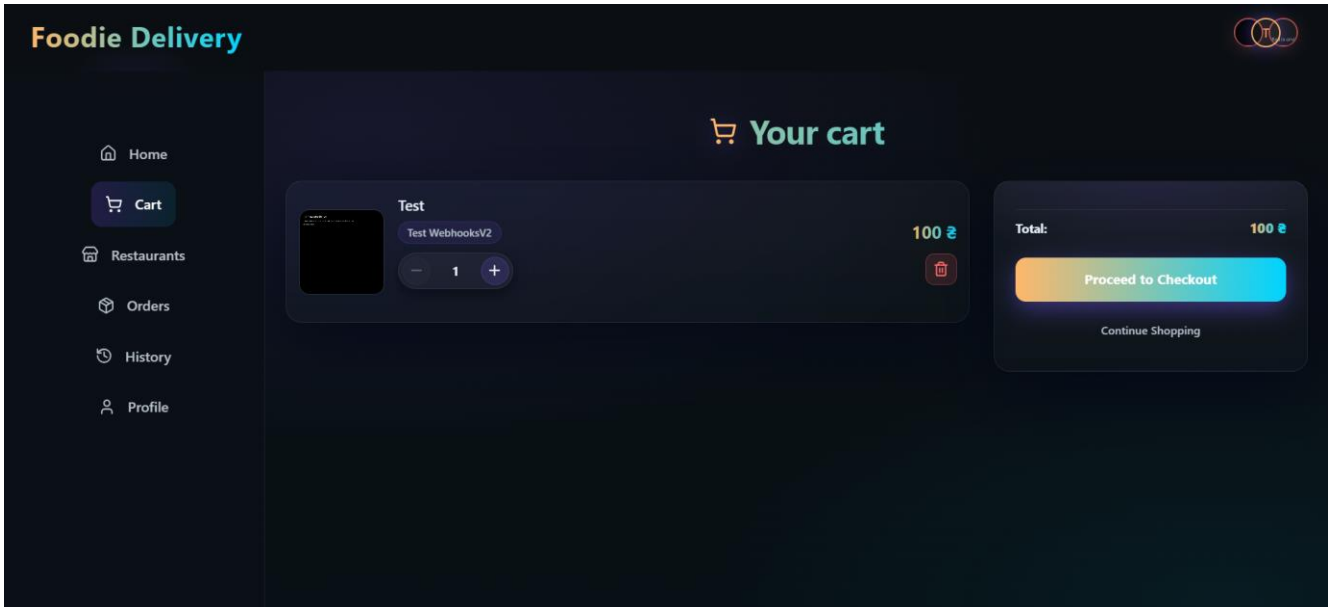


Рисунок А.11 – Сторінка кошику клієнт-акаунту

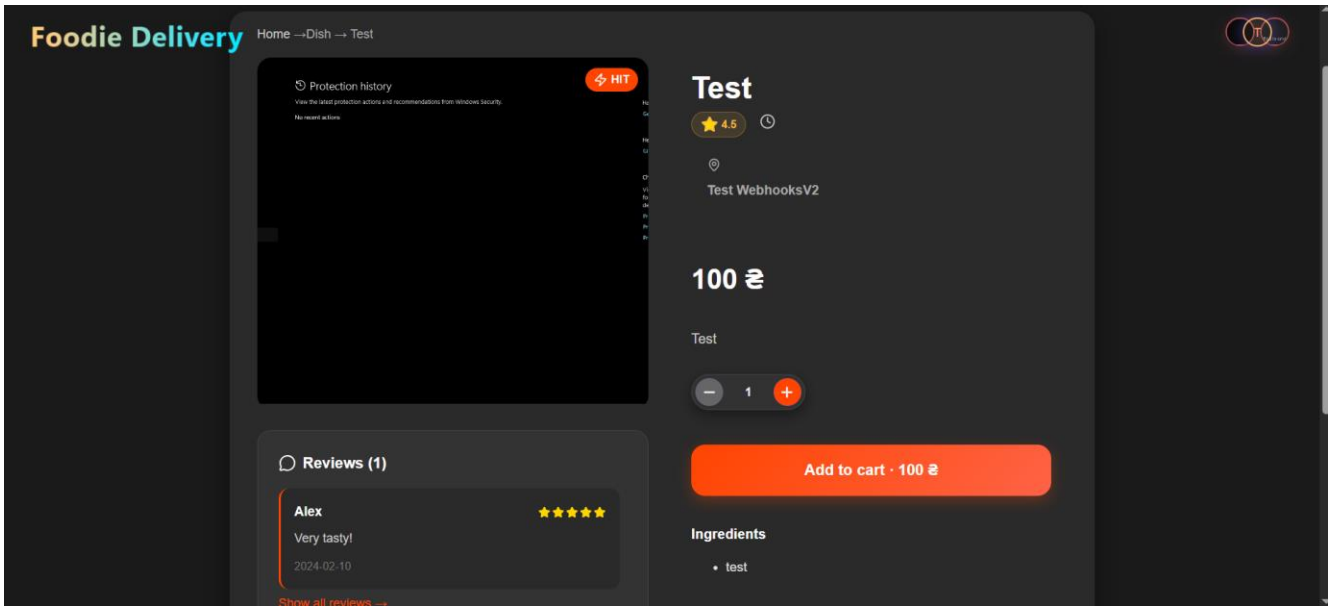


Рисунок А.12 – Сторінка деталей страви клієнт-акаунту

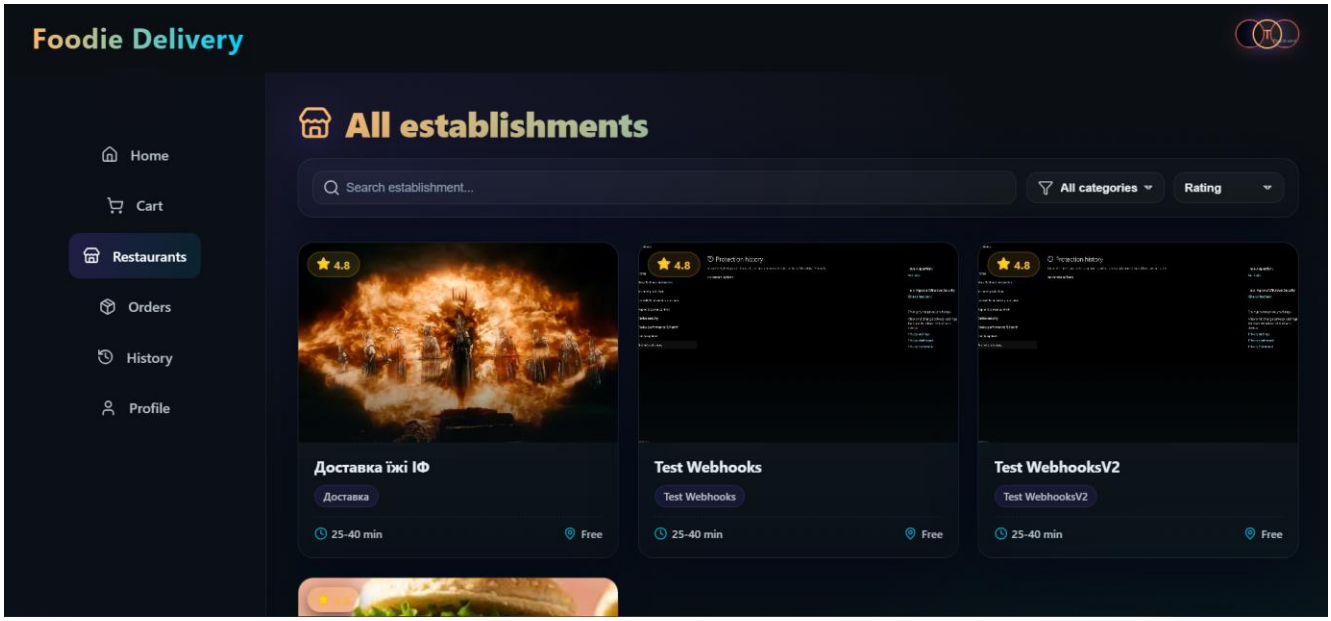


Рисунок А.12 – Сторінка списку закладів харчування клієнт-акаунту

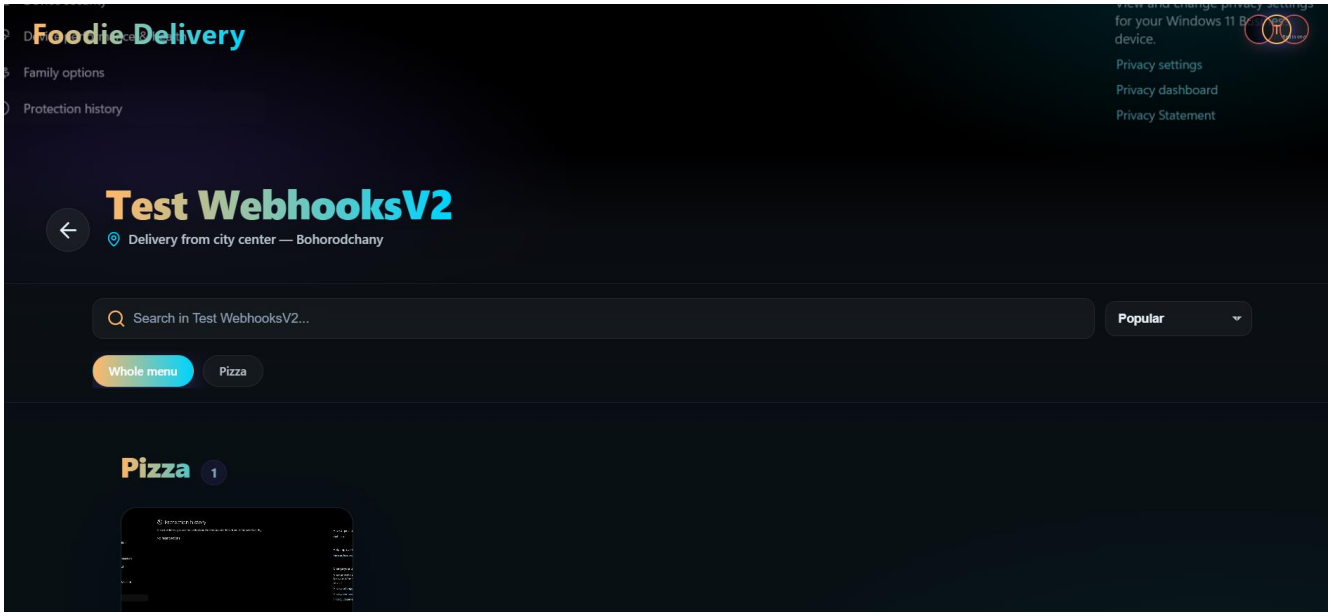


Рисунок А.13 – Сторінка деталей закладу харчування клієнт-акаунту

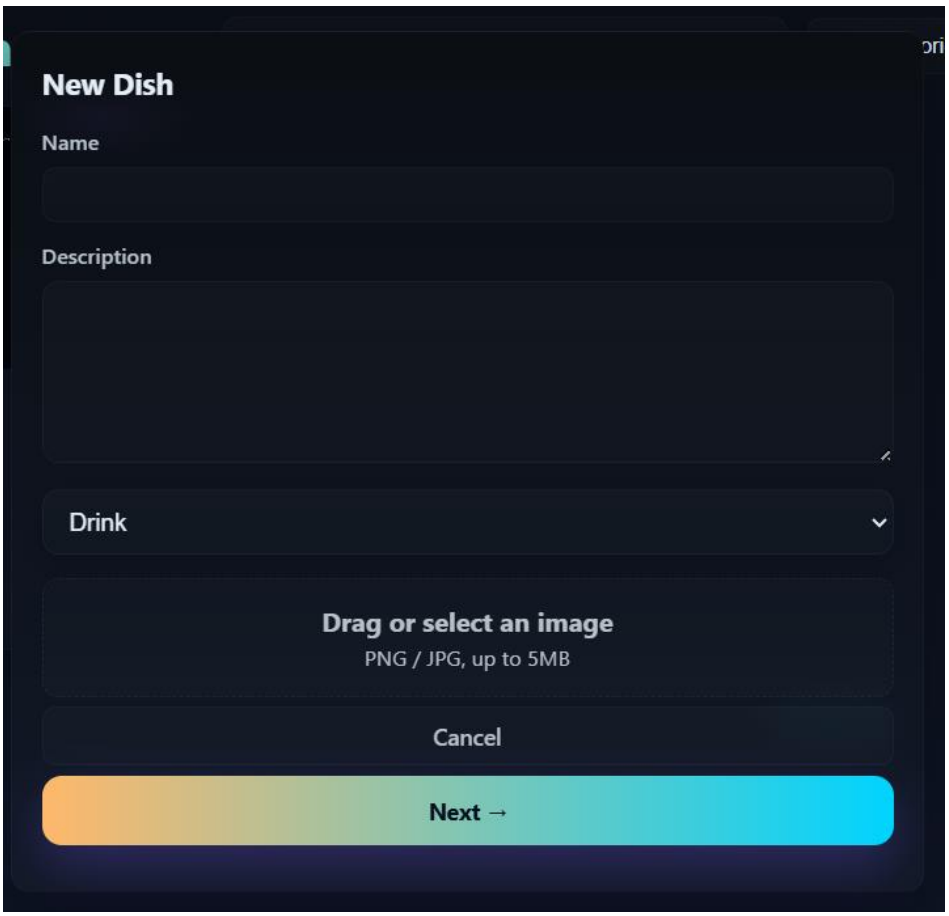


Рисунок А.14 – Форма створення страви бізнес-акаунту

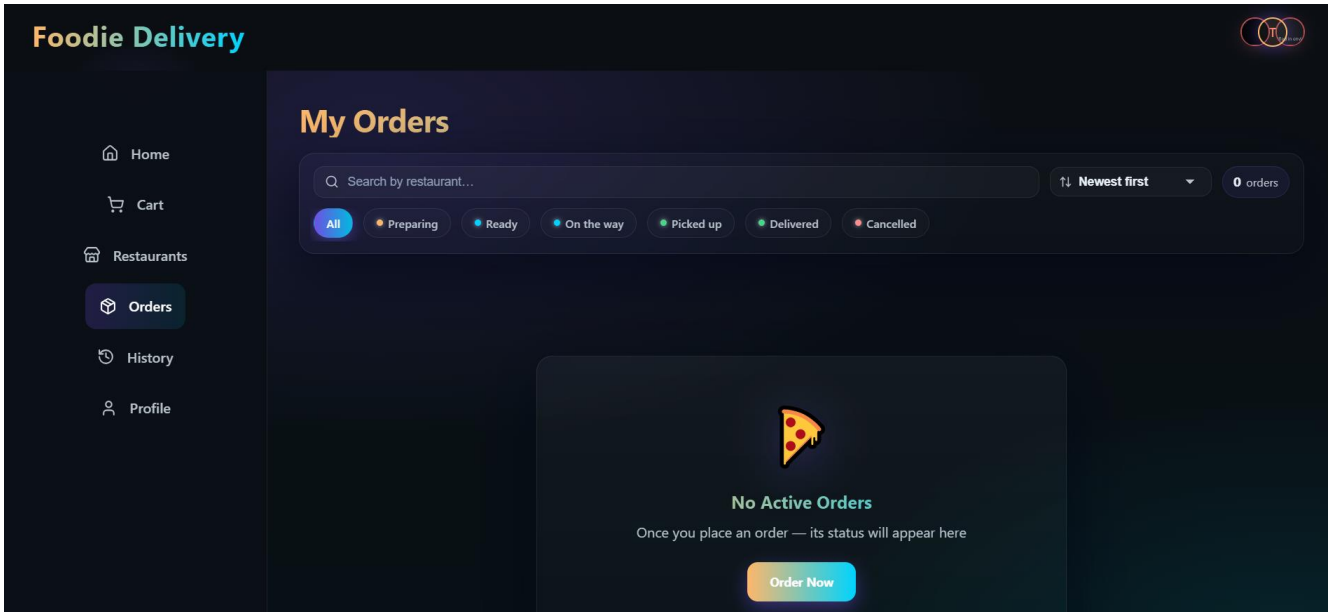


Рисунок А.15 – Сторінка активних замовлень клієнт-акаунту

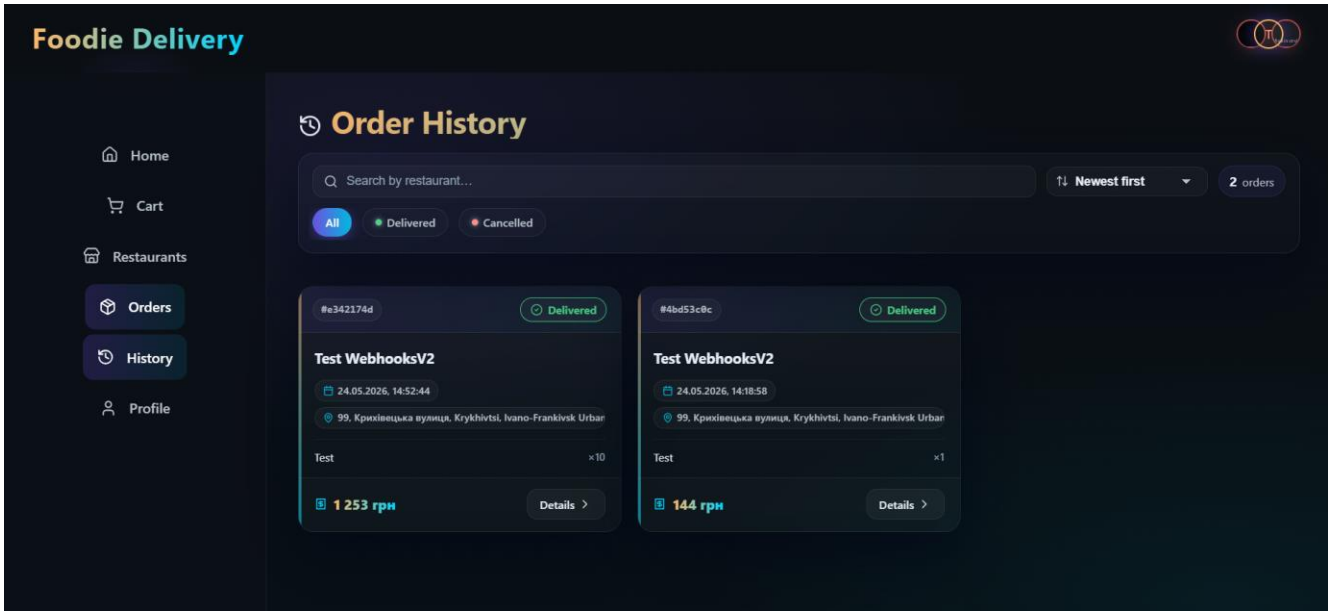


Рисунок А.16 – Сторінка історії замовлень клієнт-акаунту

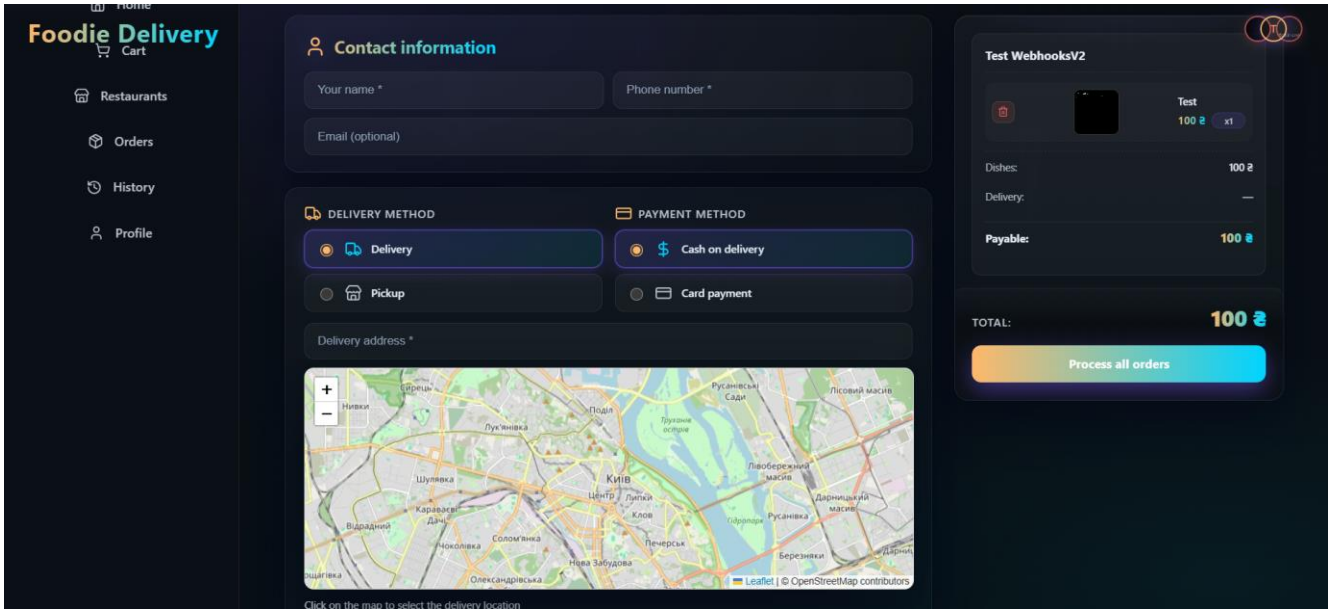


Рисунок А.17 – Сторінка оформлення замовлень клієнт-акаунту

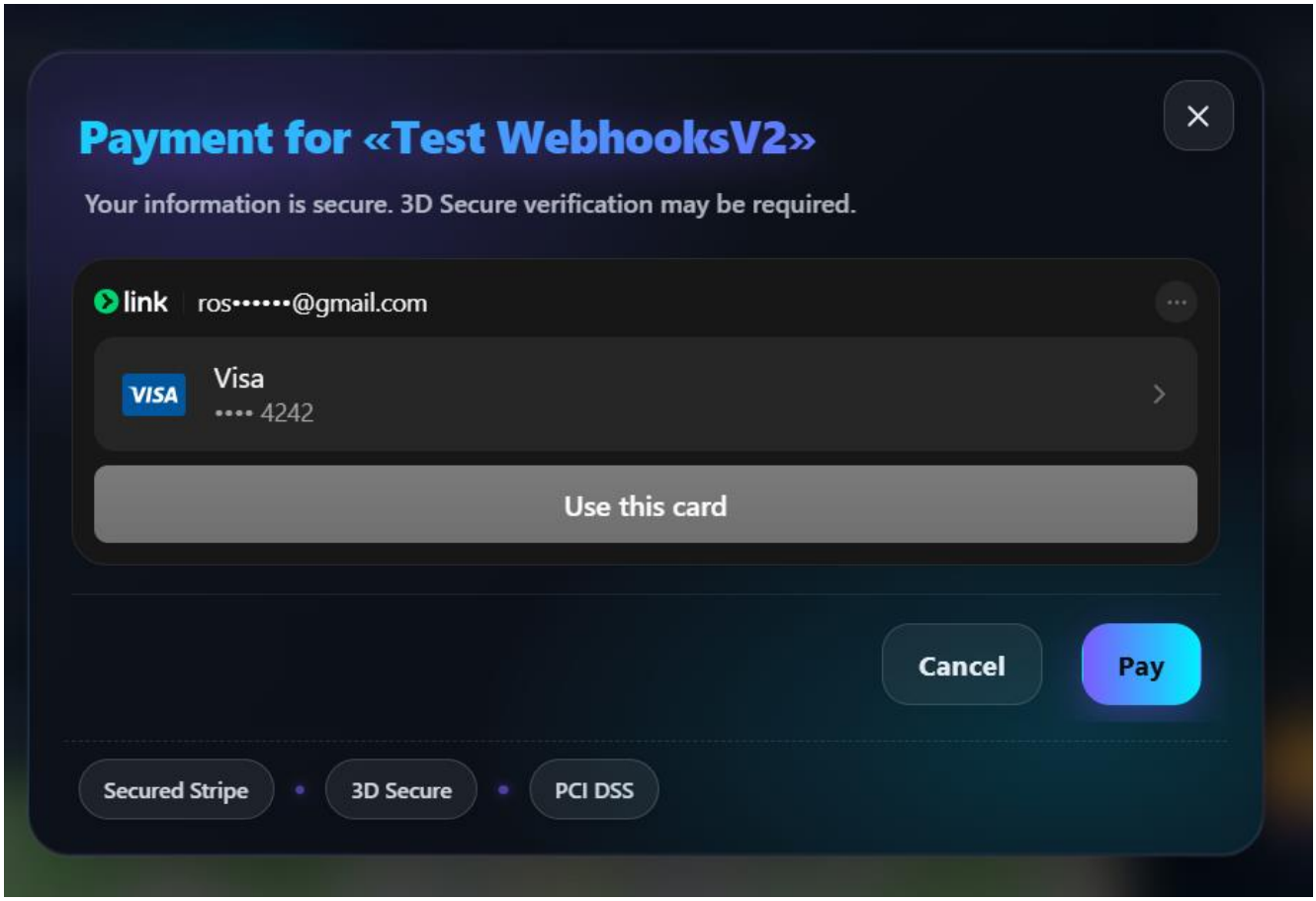


Рисунок А.18 – Форма оплати замовлення клієнт-акаунту

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: " Програмна реалізація прототипу облікової освітньої системи "

Обсяг пояснювальної записки: 98 аркушів

Дата закінчення дипломної роботи червня 2026р.

Підпис студента _____