МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА СИСТЕМИ ОНЛАЙН-ЗАМОВЛЕННЯ ТА ДОСТАВКИ З ВИКОРИСТАННЯМ REACT, EXPRESS.JS ТА POSTGRESQL»**

Виконав: студент 4 курсу, групи 6.1212-пі-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
О.В. Клімов
(ініціали та прізвище)

Керівник доцент кафедри програмної інженерії, PhD Столярова А.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

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

Запоріжжя – 2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія

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

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Клімову Олександру Володимировичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка системи онлайн-замовлення та доставки з використанням React, Express.js та PostgreSQL

керівник роботи Столярова Анастасія Валеріївна, PhD
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 02 » грудня 2025 року № 1741-с

2. Строк подання студентом роботи 01.06.2026 р.

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

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Аналіз предметної області та вимог до системи.
2. Проєктування системи.
3. Реалізація та тестування системи.

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

6. Консультанти розділів роботи

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

7. Дата видачі завдання 09.12.2025 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	09.01.2026	
2.	Збір вихідних даних.	16.01.2026	
3.	Обробка методичних та теоретичних	13.02.2026	
	джерел.		
4.	Розробка першого та другого розділу.	03.04.2026	
5.	Розробка третього розділу.	20.05.2026	
6.	Оформлення та нормоконтроль	25.05.2026	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	15.06.2026	

Студент _____ О.В. Клімов
(підпис) (ініціали та прізвище)

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

Нормоконтроль пройдено

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

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка системи онлайн замовлення та доставки з використанням React, Express.js та PostgreSQL»: 56 с., 18 рис., 6 джерел.

АВТОМАТИЗАЦІЯ, БІЗНЕС-ПРОЦЕС, ВЕБЗАСТОСУНОК, MIDDLEWARE, PRISMA, REDIS, STRIPE.

Об’єкт дослідження – процеси розроблення, інтеграції та забезпечення якості кросплатформного програмного забезпечення для систем масового обслуговування.

Мета роботи: скорочення витрат часу на обробку замовлень та зменшення трудомісткості взаємодії між клієнтами, кухнею і кур’єрами у сфері доставки раціонів харчування шляхом розроблення спеціалізованого вебзастосунку.

Методи дослідження – методи системного аналізу предметної області, методи збору та аналізу вимог, методи об’єктно-орієнтованого проєктування та моделювання, методи конструювання й тестування програмного забезпечення.

У результаті розроблено інформаційну систему, яка автоматизує цикл замовлення та логістики сервісу доставки харчування. Система розроблена мовою TypeScript. Серверний функціонал реалізовано на базі Node.js та Express, а для безпечного управління даними використано PostgreSQL з ORM Prisma. Клієнтську частину побудовано за допомогою React і Redux Toolkit. Функціонал системи розширено інтеграцією сторонніх API: Stripe, Cloudinary та OpenStreetMap. Практичне значення одержаних результатів полягає у створенні готового програмного рішення, здатного оптимізувати бізнес-процеси та скоротити витрати часу на обробку замовлень.

SUMMARY

Bachelor’s qualifying paper “Development of an Online Ordering and Delivery System Using React, Express.js, and PostgreSQL ”: 56 pages, 18 figures, 6 references.

AUTOMATION, BUSINESS PROCESS, MIDDLEWARE, PRISMA, REDIS, STRIPE, WEB APPLICATION.

The object of research is the processes of development, integration, and quality assurance of cross-platform software for mass service systems.

The purpose of the work is to reduce the time spent on order processing and decrease the labor intensity of interaction between customers, the kitchen, and couriers in the field of food ration delivery through the development of a specialized web application.

Research methods include methods of system analysis of the subject area, methods of requirements gathering and analysis, methods of object-oriented design and modeling, as well as methods of software construction and testing.

As a result, an information system that automates the order and logistics cycle of a food delivery service has been developed. The system is developed in the TypeScript language. The server functionality is implemented on the basis of Node.js and Express, and PostgreSQL with Prisma ORM is used for secure data management. The client side is built using React and Redux Toolkit. The functionality of the system is expanded by the integration of third-party APIs such as Stripe, Cloudinary, and OpenStreetMap. The practical significance of the obtained results lies in the creation of a ready-made software solution capable of optimizing business processes and reducing the time spent on order processing.

ЗМІСТ

Завдання на кваліфікаційну роботу……..2
Реферат 4
Summary 5
Вступ..... 8
1 Аналіз предметної області та вимог до системи..... 10
1.1 Опис предметної області 10
1.2 Опис інформаційної системи та її функціональних вимог 10
1.3 Обґрунтування вибору технологій розробки 12
1.4 Порівняння з подібними системами..... 14
1.5 Висновки з розділу 1 15
2 Проєктування системи..... 16
2.1 Концептуальне проєктування 16
2.2Логічне проєктування..... 19
2.3 Фізичне проєктування 23
2.4 Висновки з розділу 2..... 26
3 Реалізація та тестування системи 28
3.1 Опис інструментів розробки 28
3.2 Основні класи та компоненти системи 28
3.3 Інформаційна безпека та захист даних 31
3.3.1 Безпечна автентифікація та авторизація..... 31
3.3.2 Управління доступом на основі ролей 34
3.3.3 Безпека фінансових транзакцій 36
3.3.4 Валідація вхідних даних..... 38
3.4 Реалізація інтерфейсу 41
3.5 Тестування системи 46
3.5.1 Модульне тестування 46
3.5.2 Інтеграційне тестування..... 48
3.5.3 Тестування навантаженням 51

Висновки 54
Перелік посилань..... 56

ВСТУП

У сучасному суспільстві через значне прискорення темпу життя, високу професійну зайнятість та прагнення людей оптимізувати щоденні процеси виникає стійкий дефіцит особистого часу. Це безпосередньо впливає на побутову рутину, де самостійне приготування корисних страв вимагає значних часових витрат на закупівлю якісних інгредієнтів, підрахунок калорій та кулінарну обробку, внаслідок чого споживачі дедалі частіше делегують ці завдання спеціалізованим службам. Як наслідок, стрімко зростає попит на сервіси доставки збалансованих раціонів. Проте існуючі платформи – це переважно універсальні агрегатори, які не враховують специфіку тісної взаємодії клієнта, кухні та кур'єра у моделі «їжа за підпискою».

Розробка спеціалізованого вебзастосунку для автоматизації всього циклу – від онлайн-оплати до завдань кухарям і маршрутів на мапі – значно оптимізує бізнес-процеси та покращить користувацький досвід. Це зумовлює високу актуальність проєкту.

Мета – скорочення витрат часу на обробку замовлень та зменшення трудомісткості взаємодії між клієнтами, кухнею і кур'єрами у сфері доставки раціонів харчування шляхом розроблення спеціалізованого вебзастосунку.

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

- проаналізувати існуючі рішення та специфіку предметної області у сфері сервісів доставки їжі за підпискою;
- спроєктувати архітектуру вебзастосунку, визначити рольову модель та розробити структуру реляційної бази даних;
- розробити серверну частину з використанням платформи Node.js, бази даних PostgreSQL та фреймворків Express і ORM Prisma;
- розробити клієнтську частину з використанням бібліотеки React, інструментів управління станом Redux Toolkit та стилізації Tailwind CSS;

- інтегрувати зовнішні сервіси: платіжну систему, хмарне сховище для медіафайлів та сервіси геокодування з картографією;
- протестувати розроблений вебзастосунок на предмет коректності роботи бізнес-логіки та зручності користувацького інтерфейсу.

Об’єкт дослідження – процеси розроблення, інтеграції та забезпечення якості кросплатформного програмного забезпечення для систем масового обслуговування.

Предмет дослідження – архітектура, функціональні модулі та програмні компоненти full-stack вебзастосунку для автоматизації сервісів масового обслуговування на базі клієнт-серверної технології.

Методи дослідження – методи системного аналізу предметної області, методи збору та аналізу вимог, методи об’єктно-орієнтованого проєктування та моделювання, методи конструювання й тестування програмного забезпечення.

Кваліфікаційна робота складається зі вступу, трьох розділів, висновків, переліку посилань. Загальний обсяг роботи становить 56 сторінок. У переліку посилань 6 використаних джерел.

Перший розділ присвячений системному аналізу предметної області сервісів доставки раціонів, формуванню вимог до системи та обґрунтуванню вибору технологічного стеку. У другому розділі наведено архітектурне проєктування платформи, яке представлене UML-діаграмами варіантів використання, послідовностей та компонентів, а також схемою бази даних. Третій розділ описує практичну реалізацію серверної та клієнтської частин вебзастосунку, впроваджені механізми інформаційної безпеки та захисту даних, а також результати модульного, інтеграційного й навантажувального тестування.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИМОГ ДО СИСТЕМИ

1.1 Опис предметної області

Предметною областю даної роботи є автоматизація процесів функціонування сервісу доставки раціонів харчування за підпискою. У сучасному світі, через швидкий темп життя, спостерігається стійкий попит на послуги, що дозволяють делегувати приготування їжі професійним сервісам.

Особливістю даної області є модель, де компанія не просто доставляє їжу з ресторанів, а сама формує меню, готує та здійснює логістику. Це вимагає чіткої синхронізації між чотирма групами учасників:

– *клієнти*, які купують плани харчування на тривалий термін;
– *адміністратори*, які керують асортиментом та користувачами;
– *кухня*, має знати точну кількість порцій на кожен день;
– *кур'єри*, які потребують оптимізованих маршрутів для доставки.

Існуючі універсальні агрегатори Glovo, Bolt Food зосереджені на разових доставках і не надають інструментів для автоматизації виробничих циклів кухні, яка готує спеціалізовані меню. Тому розробка власної інформаційної системи є необхідною для підприємств, що працюють у ніші здорового або спортивного харчування.

1.2 Опис інформаційної системи та її функціональних вимог

Розроблювана система являє собою багатомодульне рішення, спроєктоване для автоматизації та оптимізації ключових бізнес-процесів сервісу доставки раціонів харчування. Основна концепція системи полягає у створенні цілісного цифрового середовища, яке забезпечує синхронізацію даних між клієнтами, кухнею та службою логістики. Такий підхід дозволяє

суттєво мінімізувати операційні ризики, повністю виключити надлишкову паперову звітність та нівелювати негативний вплив людського фактору на етапах обробки, агрегації та супроводу замовлень.

Для досягнення поставлених цілей система повинна реалізовувати наступні **функціональні вимоги**, розподілені за ролями користувачів.

Основні функції клієнта:

– реєстрація та авторизація в системі, включаючи підтримку OAuth 2.0;
– перегляд меню та детальних описів страв;
– вибір періоду підписки та інтерактивний розрахунок вартості замовлення;
– інтегрована онлайн-оплата замовлень через платіжні шлюзи;
– управління профілем, зміна особистих даних та перегляд історії підписок;
– можливість залишати відгуки про замовлені тарифи.

Функції панелі кухні:

– автоматична агрегація загальної кількості порцій страв на конкретну дату;
– надання інтерфейсу для формування індивідуальних раціонів.

Функції панелі кур'єра:

– перегляд списку призначених замовлень для формування оптимального маршруту;
– візуалізація точок доставки на інтерактивній мапі;
– зміна статусів доставки в режимі реального часу.

Функції консолі адміністратора:

– повне управління каталогом страв та тарифними планами;
– модерація відгуків користувачів для забезпечення якості контенту та зворотного зв'язку;
– керування обліковими записами користувачів системи.

1.3 Обґрунтування вибору технологій розробки

Ефективність, безпека та здатність системи до масштабування безпосередньо залежать від правильного вибору інструментальних засобів на етапі проєктування. З огляду на високі вимоги до швидкодії та надійності розроблюваного сервісу доставки раціонів, було сформовано сучасний технологічний стек.

Основною мовою розробки як для серверної, так і для клієнтської частини було обрано TypeScript. Це строго типізована надмножина мови JavaScript, яка дозволяє виявляти більшість помилок ще на етапі компіляції. Використання однієї мови на всіх рівнях системи дає змогу перевикористовувати типи даних між Frontend та Backend, що значно знижує ризик виникнення помилок під час передачі даних через API та пришвидшує процес розробки [1].

Для побудови користувацького інтерфейсу обрано бібліотеку React. Її компонентний підхід та використання віртуального DOM забезпечують високу швидкість рендерингу. Це важливо для такого елементу системи, як інтерактивна панель кур’єра, де дані можуть оновлюватися в режимі реального часу.

Для керування глобальним станом застосунку інтегровано Redux Toolkit у зв’язці з RTK Query. Це дозволить ефективно реалізувати механізми кешування даних на стороні клієнта, мінімізувати кількість дублюючих мережевих запитів до сервера та забезпечити автоматичну синхронізацію стану інтерфейсу з базою даних [2].

Візуальне оформлення побудовано за допомогою фреймворку Tailwind CSS, що забезпечує створення адаптивного та сучасного дизайну без необхідності написання громіздких кастомних CSS-файлів.

Серверну логіку реалізовано на платформі Node.js. Її архітектура ідеально підходить для створення RESTful API, яке має одночасно обробляти велику кількість запитів від різних користувачів [3].

Для маршрутизації та обробки HTTP-запитів використано фреймворк Express.js, який дозволяє легко підключати проміжні програмні забезпечення для валідації даних та перевірки прав доступу.

Оскільки система працює з фінансовими транзакціями та складною бізнес-логікою, використання NoSQL рішень було визнано недоцільним. Вибір зроблено на користь надійної реляційної системи керування базами даних PostgreSQL, яка гарантує цілісність даних.

Взаємодія сервера з базою даних здійснюється через ORM-інструмент Prisma. Він забезпечує строгу типізацію запитів до БД, генерує безпечний SQL-код та автоматизує процес створення міграцій при зміні архітектури даних [4].

Для забезпечення максимальної продуктивності та стійкості до навантажень у систему інтегровано Redis – високошвидкісне in-memory сховище структур даних. У межах проєкту воно використовується для зберігання тимчасових даних та кешування результатів найбільш ресурсомістких запитів. Це дозволяє суттєво знизити навантаження на основну базу даних PostgreSQL і мінімізувати час відповіді сервера [5].

Для розширення функціоналу системи застосовано інтеграцію зі сторонніми сервісами.

Stripe API: використовується для еквайрингу та обробки онлайн-платежів. Це знімає з розроблюваної системи відповідальність за зберігання конфіденційних даних банківських карток клієнтів і гарантує відповідність міжнародним стандартам безпеки PCI DSS [6].

Cloudinary: хмарний сервіс, для зберігання та миттєвої оптимізації медіаконтенту, що суттєво зменшує навантаження на основний сервер та пришвидшує завантаження сторінок у клієнтів.

OpenStreetMap: відкритий картографічний сервіс, який застосовується для геокодування адрес доставки та візуалізації маршрутів у панелі кур’єра.

1.4 Порівняння з подібними системами

Для оцінки розроблюваного програмного продукту необхідно провести порівняльний аналіз із існуючими рішеннями. Аналоги можна розділити на дві основні групи: глобальні агрегатори замовлень та спеціалізовані локальні сервіси доставки раціонів.

До першої групи належать такі, як Glovo та Bolt Food. Їхньою головною перевагою є величезна база користувачів та налагоджена логістична мережа. Проте ці системи орієнтовані на разові замовлення з ресторанів і абсолютно не пристосовані до бізнес-процесів виробництва раціонів за підпискою. Вони не мають функціоналу для автоматичного зведення замовлень для кухні на тиждень вперед та не дозволяють гнучко керувати тривалими планами харчування.

Друга група – це локальні спеціалізовані сервіси. Вони пропонують підписки на раціони, проте технічна реалізація багатьох таких проєктів часто обмежується простими сайтами. У таких випадках обробка замовлень, формування завдань для кухарів та координація кур’єрів здійснюються вручну, що суттєво підвищує ризик виникнення помилок та уповільнює масштабування бізнесу.

Переваги розроблюваної системи:

- *автоматизація*: на відміну від локальних конкурентів, система автоматично агрегує дані про всі активні підписки, формуючи чітке завдання для кухні, що мінімізує людський фактор;
- *спеціалізована логістика*: впровадження інтерактивної мапи для кур’єрів на базі OpenStreetMap дозволяє оптимізувати маршрути доставки саме для ранкових виїздів із раціонами;
- *сучасний технологічний стек*: використання Node.js, Prisma та Redis забезпечує високу швидкість обробки транзакцій та кешування даних, що ставить систему на один рівень із великими IT-платформами.

Недоліки:

– *залежність від сторонніх API*: стабільність роботи Stripe або Cloudinary впливає на функціонал оплати та відображення контенту.

Проведений аналіз показав, що наявні на ринку рішення або є занадто універсальними, або недостатньо автоматизованими. Розроблювана система пропонує глибоку інтеграцію бізнес-процесів кухні та логістики, що зекономити ресурси та підвищити якість обслуговування.

1.5 Висновки з розділу 1

У ході аналізу предметної області було з’ясовано, що сервісам доставки раціонів за підпискою потрібна спеціалізована програма, яка зв’яже в одному просторі клієнтів, адміністраторів, кухню та кур’єрів. Популярні агрегатори типу Glovo чи Bolt Food для цього не підходять, бо вони розраховані лише на разові замовлення з ресторанів і не вміють рахувати порції для кухарів на тиждень наперед, а дрібні конкуренти зазвичай не мають такої наведеної автоматизації процесів.

Сформовані вимоги описують систему, яка повністю автоматизує роботу: клієнт зможе легко обрати та оплатити тариф, кухня одразу побачить точну кількість страв на день, а кур’єр отримає готовий маршрут на мапі. Для реалізації цього плану було обрано сучасний та надійний стек технологій. Тепер, після проведеного аналізу та визначення вимог, можна перейти до розділу проєктування.

2 ПРОЄКТУВАННЯ СИСТЕМИ

У даному розділі на основі вимог, сформованих на попередньому етапі аналізу, здійснюється проєктування архітектури інформаційної системи сервісу доставки раціонів. Для моделювання структури та поведінки застосунку застосовуються інструменти UML. Розділ послідовно розкриває логічну модель бази даних, рольову взаємодію акторів, а також компонентну і програмну архітектуру, створюючи технічний план для подальшого етапу практичної реалізації системи.

2.1 Концептуальне проєктування

Для концептуального моделювання функціональних вимог до розроблюваної системи було побудовано діаграму варіантів використання в нотації UML. Дана діаграма дозволяє продемонструвати межі системи, визначити акторів та описати головні сценарії їхньої взаємодії з системою.

На рисунку 2.1 представлена діаграма варіантів використання. На діаграмі представлено базового актора «Гість», який відображає спільні початкові функціональні можливості для всіх користувачів системи. Цей актор має доступ до загальнодоступної інформації: перегляду меню та відгуків, а також доступу до функцій реєстрації та авторизації. Від нього успадковують свої права всі інші спеціалізовані актори системи: «Користувач», «Адмін», «Кухня» та «Кур’єр».

Для актора «Користувач» виділено основний варіант використання – оформлення замовлення. Після того, як користувач авторизується у системі, він отримує можливість сформувати замовлення. Цей процес може бути динамічно доповнений опціональними діями, такими як «Оплата» або «Редагування замовлення». Крім того, користувачу доступний розділ

«Перегляд профілю», з якого він за потреби може перейти до перегляду власної історії замовлень або редагування особистих даних. Для завершення сеансу передбачено функцію виходу з системи.

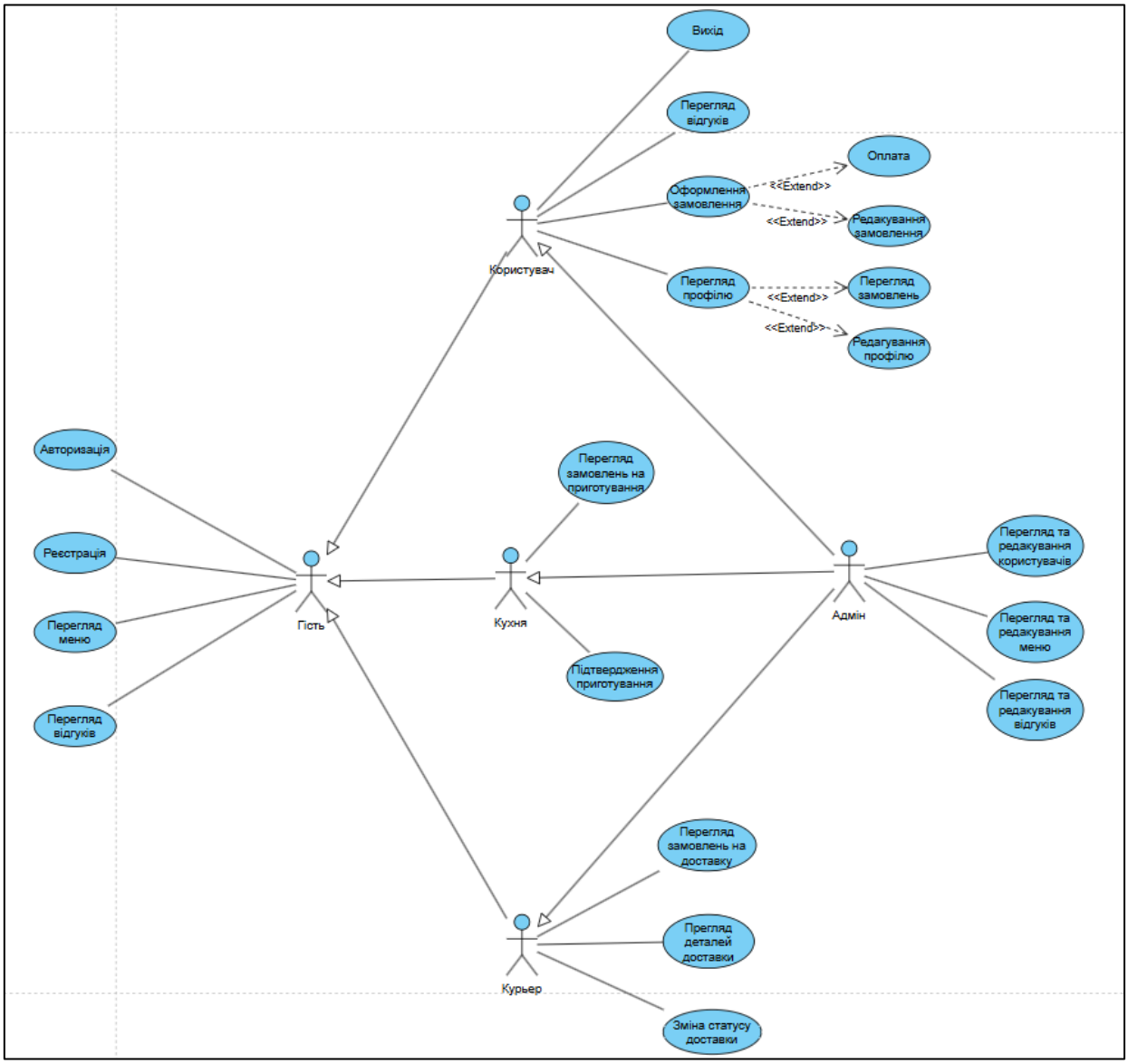


Рисунок 2.1 – Діаграма варіантів використання

Актор «Адмін» безпосередньо є користувачем з розширеними правами на взаємодію з системою. Його основні прецеденти спрямовані на адміністрування та керування контентом платформи. Адміністратор здійснює перегляд та редагування бази користувачів, керування асортиментом, а також модерацію відгуків клієнтів. На кожну з цих дій система надсилає відповідний запит до БД для оновлення або збереження даних.

Для безперебійного функціонування бізнес-логіки виділено акторів персоналу, чиї дії є послідовними. Актор «Кухня» використовує систему для перегляду актуальних замовлень, що надійшли на приготування, та здійснює дію підтвердження після їх готовності. Після цього актор «Кур'єр» отримує можливість переглядати список замовлень, призначених на доставку, детально ознайомлюватися з інформацією про клієнта та адресу, а також змінювати статус доставки в системі в процесі виконання своєї роботи.

Наведемо опис основного прецеденту системи – «Оформлення замовлення».

Призначення: даний варіант використання надає можливість авторизованому користувачу сформувати своє замовлення, відредагувати його деталі та здійснити оплату.

Основний потік подій:

– користувач починає процес оформлення замовлення;
– система відображає деталі замовлення;
– користувач підтверджує правильність введених;
– система пропонує здійснити оплату;
– користувач успішно проводить платіж;
– система зберігає замовлення, оновлює його статус та передає інформацію до модуля «Кухня».

Альтернативні потоки:

а) редагування даних під час оформлення:
   – якщо користувач бажає змінити склад замовлення до його підтвердження, спрацьовує розширення «Редагування замовлення»;
   – система дозволяє змінити параметри та перераховує загальну вартість.

б) відхилення або скасування оплати: якщо транзакція не пройшла успішно, система видає повідомлення про помилку: «Оплату відхилено» і замовлення залишається в статусі очікування.

Передумови: для оформлення замовлення користувач повинен:

– бути зареєстрованим та увійти в систему.
– обрати хоча б один тарифний план або страву з меню.

Постумови: після успішного оформлення та оплати система:

– створює нове замовлення зі статусом «Оплачено» або «В обробці».
– відображає це замовлення в історії.
– відправляє завдання у список замовлень на приготування для актора «Кухня».

Таким чином, побудована діаграма варіантів використання та деталізація базового прецеденту дозволили чітко окреслити функціональні межі системи й формалізувати вимоги користувачів, що закладає основу для подальшого логічного проєктування структури даних.

2.2 Логічне проєктування

На поточному етапі створено логічну модель даних, що відображає архітектуру та бізнес-логіку розроблюваного сервісу. Завдяки побудові деталізованої логічної схеми бази даних, наведеної на рисунку 2.2, вдалося наочно представити структуру застосунку, виокремити його головні сутності та визначити правила їхньої взаємодії.

Детальний аналіз структурних компонентів зображеної схеми, їхніх ключових атрибутів та правил взаємодії у межах виділених сутностей логічної моделі наведено нижче.

Клас User – це центральна сутність системи, яка зберігає облікові дані для всіх учасників робочого процесу.

Атрибути: id, email (електронна адреса для авторизації), password (хешований пароль), name (ім’я користувача) та role (перелічуваний тип для визначення прав доступу: ADMIN, CLIENT, COOK, COURIER).

Зв’язки: користувач-клієнт ініціює створення замовлень (Order), користувач із роллю кур’єра здійснює доставку на конкретні дні (Order_Day),

а будь-який зареєстрований клієнт має право залишати відгуки (Review) на замовлені тарифи.

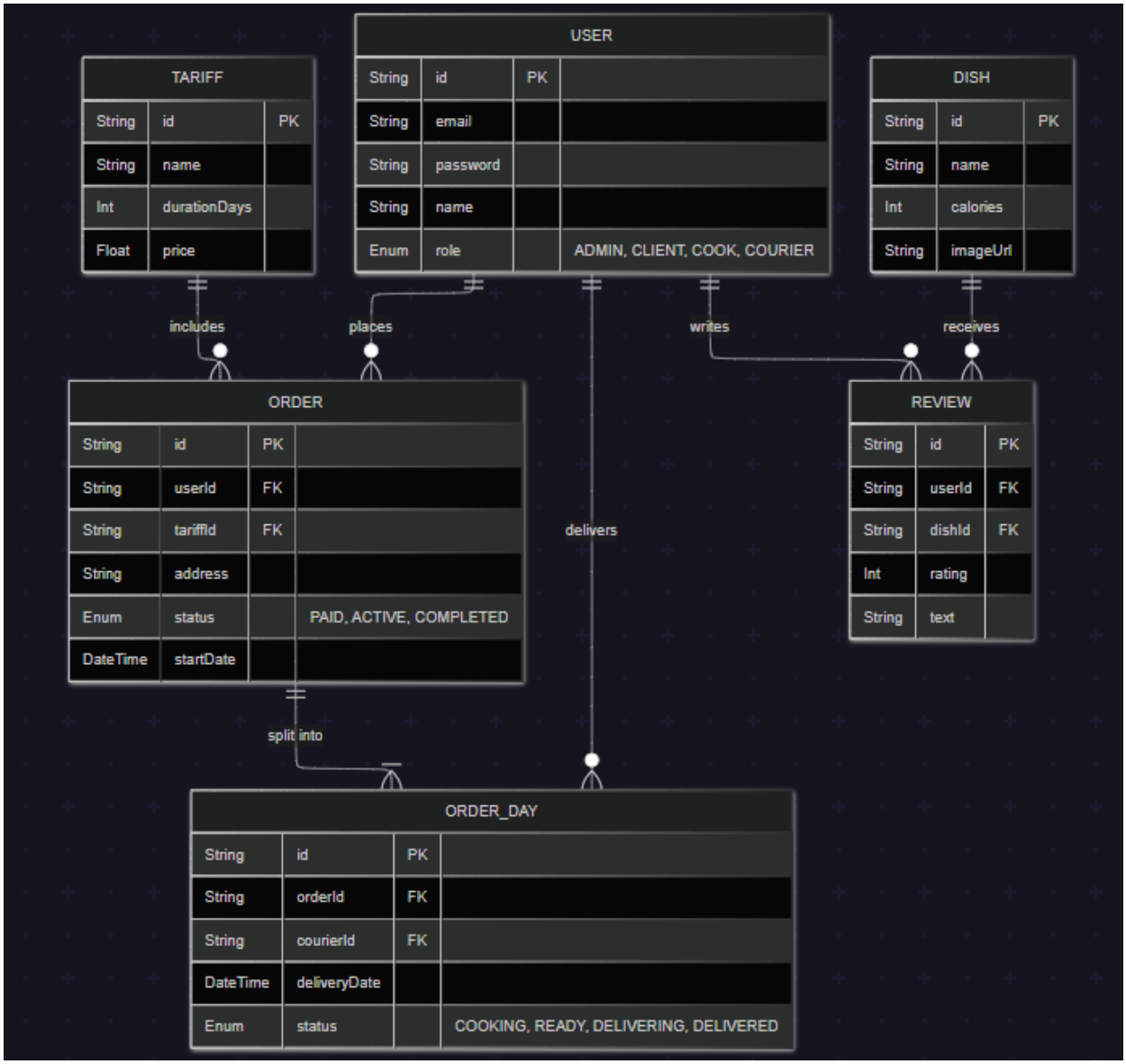


Рисунок 2.2 – Схема бази даних

Клас Tariff – описує доступні підписки на харчування.

Атрибути: id, name (назва тарифу), durationDays (тривалість підписки в днях) та price (вартість тарифу).

Зв'язки: ця сутність пов'язана з багатьма замовленнями (Order), оскільки користувачі оформлюють підписки на основі конкретних тарифних планів, та з відгуками (Review), що дозволяє клієнтам оцінювати якість конкретного обраного тарифу.

Клас Dish – містить каталог кулінарних позицій.

Атрибути: id (UUID), name (назва страви), calories (енергетична цінність) та imageUrl (посилання на зображення страви).

Зв'язки: сутність відображає асортиментний перелік, який використовується при формуванні меню для тарифів.

Клас Order – відображає факт оформлення підписки клієнтом.

Атрибути: id, userId (посилання на клієнта), tariffId (посилання на обраний тариф), address (адреса доставки), status (поточний стан оплати або активності) та startDate (дата початку виконання підписки).

Зв'язки: замовлення належить користувачу, який його оформив, та прив'язане до конкретного тарифу. Для оптимізації логістики замовлення автоматично розбивається на окремі дні доставки Order_Day.

Клас Order_Day – деталізує процес виконання підписки по днях.

Атрибути: id, orderId (посилання на головне замовлення), courierId (посилання на кур'єра, відповідального за доставку цього дня), deliveryDate (дата доставки) та status (поточний стан виконання, наприклад: COOKING, READY, DELIVERING, DELIVERED).

Зв'язки: сутність пов'язує загальне замовлення з конкретним кур'єром, який закріплюється за кожною окремою доставкою.

Клас Review – забезпечує механізм зворотного зв'язку від клієнтів.

Атрибути: id, userId (ідентифікатор автора), tariffId (ідентифікатор оціненого тарифу), rating (числова оцінка) та text (текстовий коментар).

Зв'язки: кожен відгук однозначно належить користувачу, який його написав, та вказує на конкретний тариф, що був придбаний.

Наступним кроком для розуміння роботи системи розроблена діаграма послідовностей, яка є одним із ключових інструментів поведінкового моделювання методології UML, який допомагає аналізувати та розуміти динаміку роботи системи в часі. Цей графічний засіб відображає хронологічний порядок обміну повідомленнями між та внутрішніми компонентами системи в межах виконання конкретного бізнес-сценарію.

На рисунку 2.3 представлена діаграма послідовностей для процесу оформлення замовлення та оплати.

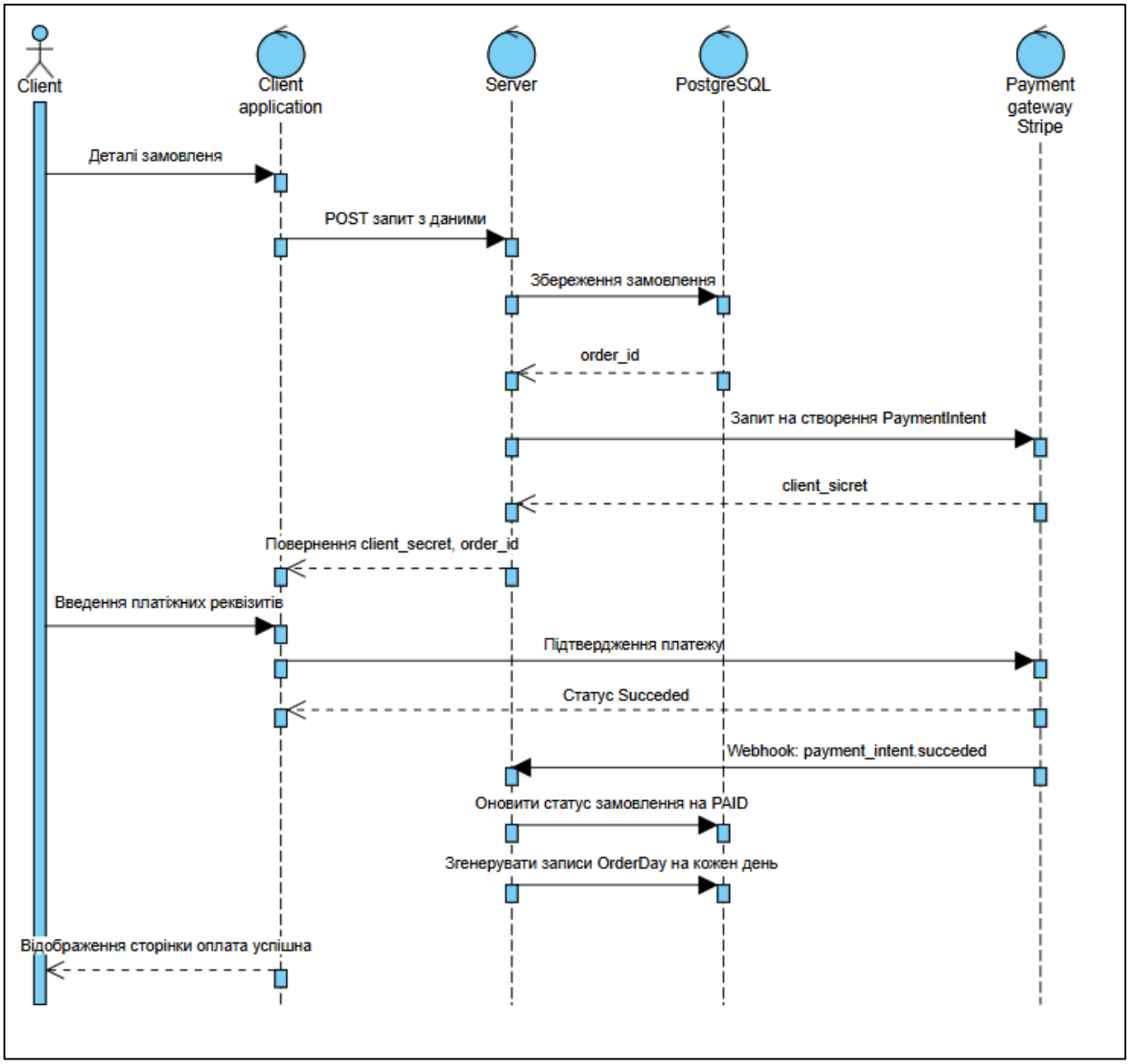


Рисунок 2.3 – Діаграма послідовностей процесу замовлення

Процес оформлення замовлення розпочинається з вибору користувачем тарифного плану, визначення тривалості підписки та введення адреси доставки, після чого ініціюється створення замовлення шляхом надсилання даних на сервер, який реєструє нову транзакцію в базі даних зі статусом «Очікує оплати».

На наступному етапі сервер звертається до платіжного шлюзу Stripe для ініціалізації платежу через створення PaymentIntent, повертаючи клієнтському

застосунку токен транзакції, за допомогою якого користувач безпечно вводить реквізити картки безпосередньо у системі Stripe. Отримавши підтвердження успішної оплати, платіжний шлюз надсилає відповідний статус до застосунку та паралельно ініціює асинхронний Webhook-запит на сервер, що дозволяє системі автоматично оновити статус замовлення на «Оплачено» та згенерувати графік щоденних доставок у базі даних, завершуючи алгоритм перенаправленням користувача на сторінку успішного оформлення підписки. Цей процес охоплює як клієнтську, так і серверну частини логіки, включаючи взаємодію зі стороннім API, забезпечуючи безпечну оплату та повну автоматизацію подальшої логістики.

Описана логічна структура сутностей та алгоритми їхньої динамічної взаємодії повністю регламентують збереження й обмін інформацією в системі, що дозволяє перейти до етапу її фізичного проєктування та програмної реалізації.

2.3 Фізичне проєктування

Фізичне проєктування є завершальним етапом архітектурного моделювання, на якому логічна структура системи трансформується у фізичні програмні компоненти, модулі та підсистеми. Головною метою цього етапу є визначення того, як саме програмне забезпечення буде реалізоване на рівні коду, як його складові частини будуть розподілені між клієнтським та серверним середовищами, та як вони взаємодіятимуть через визначені інтерфейси. Для візуалізації фізичної архітектури розроблюваного застосунку та зв’язків між його основними блоками використано діаграму компонентів.

На рисунку 2.4 зображено діаграму компонентів інформаційної системи доставки раціонів.

Архітектура системи побудована за принципом багаторівневого клієнт-серверного розподілу і розділена на **чотири** ключові зони відповідальності.

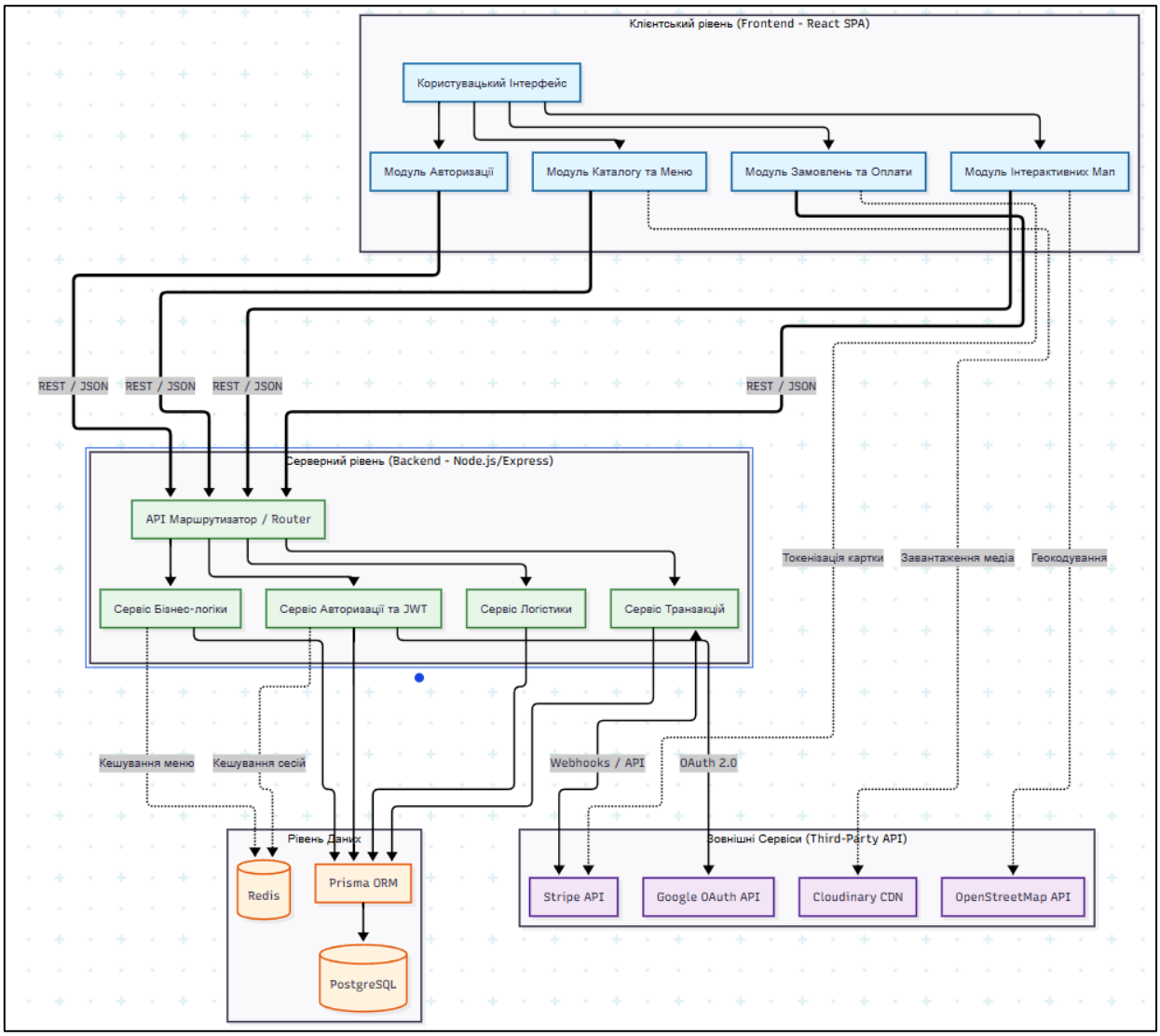


Рисунок 2.4 – Діаграма послідовностей процесу замовлення

Клієнтський рівень (Frontend – React SPA): реалізований як односторінковий застосунок, що працює в браузері користувача. Включає загальний користувацький інтерфейс, який агрегує роботу чотирьох незалежних модулів:

– модуль «Авторизації»;

– модуль «Каталогу» та «Меню»;

– модуль «Замовлень» та «Оплати»;

– модуль «Інтерактивних мап».

Взаємодія цих модулів із серверною частиною здійснюється асинхронно за допомогою протоколу HTTP(S) з використанням архітектурного стилю REST та обміном даними у форматі JSON.

Серверний рівень (Backend – Node.js/Express): ядро системи, що інкапсулює всю бізнес-логіку. Точкою входу для всіх клієнтських запитів є Router, який виконує первинну обробку та делегує задачі відповідним вузькоспеціалізованим сервісам:

– сервісу «Авторизації» та «JWT»;
– сервісу «Бізнес-логіки»;
– сервісу «Логістики»;
– сервісу «Транзакцій».

Рівень даних: забезпечує надійне збереження та швидке вилучення інформації. Сервіси бекенду не звертаються до основної реляційної бази даних PostgreSQL напряму, а використовують проміжний компонент – Prisma ORM. Це гарантує строгу типізацію та захист від SQL-ін’єкцій. Крім того, на цьому рівні розгорнуто in-memory сховище Redis, яке отримує дані від сервісів авторизації та бізнес-логіки для забезпечення швидкого кешування сесій та меню відповідно.

Зовнішні сервіси: діаграма також ілюструє інтеграцію системи з хмарними сервісами для делегування специфічних завдань:

– Google OAuth API – взаємодіє з сервісом авторизації за протоколом OAuth 2.0;
– Stripe API – отримує запити на токенізацію картки від клієнтського модуля оплат та надсилає асинхронні Webhooks до серверного сервісу транзакцій;
– Cloudinary CDN – використовується модулем каталогу для швидкого завантаження оптимізованих медіафайлів;
– OpenStreetMap API – забезпечує геокодування та картографічні дані для модуля інтерактивних мап.

Таким чином, розроблена діаграма компонентів деталізує фізичний розподіл модулів інформаційної системи та взаємодію із зовнішніми сервісами, що повністю завершує етап архітектурного проєктування платформи й дозволяє перейти до її безпосередньої програмної реалізації.

2.4 Висновки з розділу 2

У другому розділі було проведено комплексне архітектурне та системне проєктування інформаційної системи для автоматизації сервісів масового обслуговування. Процес моделювання реалізовано шляхом послідовного переходу від абстрактних концептуальних вимог до конкретних фізичних і програмних компонентів із застосуванням методології UML та реляційного підходу до побудови баз даних.

На основі аналізу отриманих результатів архітектурного моделювання було сформувано цілісний технічний базис та підтверджено спроможність системи до подальшого масштабування.

Концептуальне проєктування за допомогою діаграми варіантів використання забезпечило чітке розмежування ролей у системі та визначення їхніх зон відповідальності. Деталізація специфікації основного прецеденту «Оформлення замовлення» дозволила формалізувати алгоритми взаємодії між користувацькими та службовими інтерфейсами програми.

Спроєктована схема бази даних гарантує високий рівень нормалізації та цілісності інформації. Виокремлення центральних сутностей, визначення їхніх атрибутів та встановлення зв’язків через механізми первинних і зовнішніх ключів забезпечує ефективне керування даними та мінімізує надлишковість на рівні збереження даних.

Аналіз динамічної поведінки системи на діаграмі послідовностей підтвердив технічну можливість реалізації наскрізних бізнес-процесів. Інтеграція зі стороннім платіжним API Stripe за допомогою технології асинхронних вебхуків та обміну об’єктами PaymentIntent забезпечує високу безпеку фінансових транзакцій і виключає можливість проходження несанкціонованих запитів на серверному рівні.

Фізична архітектура застосунку, представлена діаграмою компонентів, базується на принципах багаторівневого клієнт-серверного розподілу. Поділ платформи на ізольований клієнтський рівень (Frontend), ядро бізнес-логіки

(Backend) та оптимізований рівень даних дозволяє досягти високої швидкості відклику системи за рахунок впровадження механізмів кешування в Redis та Prisma ORM.

Інтеграційний потенціал системи підтверджено моделюванням інтерфейсів взаємодії з хмарними сервісами (Google OAuth, Stripe, Cloudinary, OpenStreetMap), що дозволяє делегувати специфічні обчислювальні завдання без перевантаження власної інфраструктури застосунку.

Таким чином, розроблена проєктна документація, графічні моделі та схеми взаємодії утворюють завершений технічний план для переходу до етапу безпосередньої програмної реалізації full-stack вебзастосунку.

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ

3.1 Опис інструментів розробки

Для реалізації програмного продукту було обрано мову TypeScript. Серверну частину реалізовано за допомогою платформи Node.js та фреймворку Express.

Prisma ORM – це інструмент об’єктно-реляційного відображення для Node.js, який забезпечує строгу типізацію та безпечну взаємодію з реляційною базою даних PostgreSQL.

Redis – використовується як сховище для кешування сесій та ресурсомістких запитів до бази даних, що значно підвищує продуктивність серверної частини.

Клієнтську частину розроблено на базі бібліотеки React.

Redux Toolkit та RTK Query – інструменти управління глобальним станом застосунку та асинхронними запитами, що автоматизують кешування даних на стороні клієнта.

Для реалізації специфічного функціоналу інтегровано сторонні сервіси: Stripe для обробки фінансових транзакцій, Cloudinary для зберігання медіафайлів та OpenStreetMap для геокодування.

3.2 Основні класи та компоненти системи

Оскільки основним прецедентом розроблюваної системи є оформлення замовлення, то основними модулями та компонентами системи будуть ті, що надають можливість формувати кошик, ініціювати транзакції та створювати графік доставок.

Головними компонентами зі сторони клієнтського застосунку React є:

– OrderApi – сервіс на базі RTK Query, що відповідає за дії, пов’язані з REST API сервером, який містить запити на ініціалізацію платежу, створення замовлення та відображення історії підписок;
– CheckoutComponent – обробляє та відображає дані замовлення, містить методи для управління формою введення адреси, взаємодіє з віджетом Stripe Elements для безпечного введення реквізитів та передає токен оплати у сервіс orderApi.

Головними модулями зі сторони сервера Node.js є:

– Order та OrderDay – моделі Prisma, що пов’язані з відповідними таблицями в PostgreSQL, вони містять опис полів замовлення, розклад по днях, їх типи та зв’язки з користувачами й тарифами;
– OrderController – групує бізнес-логіку обробки запитів, пов’язаних із генерацією підписок та валідацією платежів через Stripe.

Нижче на рисунку 3.1 наведено приклади функцій клієнтського сервісу та серверного контролеру.

```
import { apiSlice } from '../api/apiSlice';

export const orderApiSlice = apiSlice.injectEndpoints({
  endpoints: (builder) => ({
    createOrder: builder.mutation<any, any>({
      query: (data) => ({
        url: '/order',
        method: 'POST',
        body: data,
      }),
      invalidatesTags: ['Order']
    }),
```

Рисунок 3.1 – RTK Query mutation для створення замовлення

У створеному слайсі orderApiSlice оголошено мутацію createOrder, яка виконує асинхронний HTTP POST-запит за маршрутом /order, передаючи дані замовлення у тілі запиту body: data. Використання властивості invalidatesTags: [‘Order’] забезпечує автоматичне очищення кешу для цієї сутності, що змушує

інтерфейс миттєво оновити дані та відобразити актуальний список замовлень після успішної транзакції.

На рисунку 3.2 наведено код контроллера, який виконує обробку цього запиту.

```
class OrderController {

  create = async (req: Request, res: Response, next: NextFunction) => {
    try {
      if (!req.user) {
        throw ApiError.forbidden("Користувач не авторизований");
      }
      const dto = new CreateOrderDto(req.body);
      const order = await orderService.create(req.user.id, dto);
      return res.json(order);
    } catch (e) {
      next(e);
    }
  }
```

Рисунок 3.2 – Метод create контролеру OrderController

Тут використовується асинхронний метод create класу OrderController. Алгоритм його роботи наступний:

- *перевірка авторизації*: система перевіряє наявність об’єкта req.user – якщо запит неавторизований, генерується виключення ApiError.forbidden;
- *валідація даних*: вхідні дані запиту req.body трансформуються в об’єкт передачі даних CreateOrderDto, що гарантує їхню правильну структуру;
- *делегування бізнес-логіки*: контролер передає ідентифікатор користувача та DTO-об’єкт до сервісного шару ordersService.create, де відбувається безпосередня взаємодія з базою даних та генерація графіку доставок;
- *обробка помилок*: весь процес інкапсульовано у блок try...catch, що дозволяє безпечно перехоплювати будь-які виняткові ситуації та передавати їх до глобального обробника помилок через функцію next(e).

3.3 Інформаційна безпека та захист даних

Враховуючи специфіку розроблюваної інформаційної системи, яка обробляє персональні дані користувачів, їхні фізичні адреси доставки та фінансові транзакції, питання інформаційної безпеки є одним із найбільш пріоритетних на всіх етапах проєктування та розробки. Архітектура безпеки вебзастосунку охоплює як клієнтський, так і серверний рівні, а також шар бази даних.

3.3.1 Безпечна автентифікація та авторизація

Процес підтвердження особи користувача та розмежування прав доступу в розробленій системі є першим і найважливішим ешелоном захисту. Враховуючи архітектуру розділеного застосунку, класична сесійна автентифікація на основі кукі-файлів була визнана неефективною, оскільки вона створює додаткове навантаження на сервер через необхідність збереження стану сесій та погано масштабується.

Замість цього було впроваджено сучасний децентралізований механізм на базі стандарту JSON Web Token. Головна перевага JWT полягає в його бездержавності: серверу не потрібно зберігати інформацію про активні сесії в оперативній пам’яті чи базі даних для кожного запиту. Уся необхідна інформація про користувача зашифрована всередині самого токена, який підписується надійною криптографічною сигнатурою на стороні сервера за допомогою секретного ключа JWT_ACCESS_SECRET.

Для забезпечення максимального рівня безпеки в системі реалізовано двокомпонентну модель токенів.

Access Token *(токен доступу)*, має короткий життєвий цикл. Він надсилається клієнтом у кожному HTTP-запиті в заголовку Authorization як Bearer token. Короткий термін дії гарантує, що навіть у разі перехоплення токена зловмисником, час його використання буде суворо обмеженим.

Refresh Token *(токен оновлення),* він має тривалий життєвий цикл і використовується виключно для отримання нового Access Token без необхідності повторного введення логіна та пароля користувачем. Для захисту від крадіжки через міжсайтовий скриптинг (XSS), Refresh Token зберігається на стороні клієнта у захищених кукі-файлах із прапорцями HttpOnly, Secure та SameSite=Strict. Це робить неможливим доступ до токена через JavaScript-код браузера.

Паралельно з цим, для покращення користувацького досвіду та підвищення безпеки, інтегровано протокол OAuth 2.0 через Google API. Цей підхід дозволяє користувачам авторизуватися за допомогою свого існуючого Google-акаунта. При цьому пароль користувача взагалі не передається в розроблювану систему, а сервер взаємодіє лише з довіреним сертифікатом від компанії Google, отримуючи підтверджені дані про профіль клієнта.

На рисунку 3.3 представлено програмну реалізацію Middleware на серверному рівні, яке відповідає за перевірку Access Token та верифікацію прав доступу користувача.

Розглянемо детально алгоритм роботи представленого модуля системи безпеки. На самому початку функції реалізовано перевірку методу запиту: якщо це попередній запит типу OPTIONS, який браузери надсилають для перевірки політики CORS, система одразу передає керування далі за допомогою функції next(), уникаючи непотрібних криптографічних перевірок.

Основна логіка авторизації інкапсульована у блок try...catch для безпечної обробки винятків. Процес верифікації складається з *послідовних етапів*.

Спочатку система перевіряє наявність заголовка authorization в об’єкті HTTP-запиту. Якщо заголовок відсутній, генерується помилка доступу за допомогою кастомного класу обробки помилок ApiError.forbidden, і запит блокується.

У разі наявності заголовка, з нього вилучається JWT-токен доступу. Рядок розбивається за символом пробілу, оскільки стандарт передбачає

передачу токена у форматі Bearer <token>. Якщо сам токен порожній, також повертається помилка.

```
export default function (req: Request, res: Response, next: NextFunction) {
    if (req.method === "OPTIONS") {
        next();
    }
    try {
        const autorizationHeader = req.headers.authorization;
        if (!autorizationHeader) {
            return next(ApiError.forbidden('Користувач не авторизований (немає заголовка)'))
        }

        const accessToken = autorizationHeader.split(' ')[1];
        if (!accessToken) {
            return next(ApiError.forbidden('Користувач не авторизований (немає заголовка)'))
        };

        const userData = tokenService.validateAccessToken(accessToken);

        if (!userData) {
            return next(ApiError.forbidden('Користувач не авторизований (токен невалідний)'));
        }

        req.user = userData as UserDto;

        next();

    } catch (e) {
        return next(ApiError.forbidden('Користувач не авторизований'));
    }
}
```

Рисунок 3.3 – Код Middleware для перевірки реєстрації

Далі відбувається валідація вилученого токена. Для цього Middleware звертається до спеціалізованого сервісу tokenService.validateAccessToken(), який здійснює перевірку сигнатури за допомогою секретного ключа. Якщо токен виявляється підробленим, невалідним або його термін дії вичерпано, сервіс не повертає дані, що призводить до блокування запиту з відповідним повідомленням.

Після успішної дешифрації та повної валідації, дані про користувача токена приводяться до структурованого типу UserDto та ін’єктується безпосередньо в об’єкт запиту: req.user = userData. Це робить інформацію про поточного клієнта доступною для всіх наступних контролерів у ланцюжку обробки.

На фінальному етапі викликається функція next(), яка дозволяє серверу перейти до виконання основної бізнес-логіки ендпоінту.

Завершується конструкція блоком catch. Він перехоплює будь-які непередбачувані помилки під час виконання коду та гарантовано повертає клієнту статус заборони доступу.

Такий підхід недозволяє витік технічної інформації через системні помилки та забезпечує безперебійну роботу механізму авторизації.

3.3.2 Управління доступом на основі ролей

Автентифікація користувача є лише першим кроком у забезпеченні безпеки інформаційної системи. Після того як особа успішно підтвердила свою ідентичність за допомогою JWT-токена, система повинна чітко визначити, які саме дії їй дозволено виконувати в межах платформи. Для вирішення цього завдання було впроваджено архітектурний шаблон Управління доступом на основі ролей.

Основна мета полягає в реалізації принципу найменших привілеїв. Згідно з цим принципом, кожному актору надається доступ виключно до тих ресурсів і функцій, які критично необхідні для виконання його обов’язків. Це дозволяє запобігти як випадковим помилкам персоналу, так і навмисним спробам несанкціонованого доступу до конфіденційних даних.

У розробленій системі бізнес-логіка передбачає чіткий розподіл на чотири системні ролі, які вже були описані у попередніх розділах.

Оскільки клієнтський застосунок функціонує в браузері користувача, будь-які перевірки ролей на фронтенді є виключно елементом UI/UX і не

можуть вважатися надійним захистом. Справжнє розмежування прав реалізовано на рівні Backend (Node.js/Express) через ланцюжок Middleware. Після того, як попередній Middleware успішно перевірив токен і записав дані користувача в req.user, у гру вступає модуль перевірки ролей.

На рисунку 3.4 представлено програмну реалізацію проміжного програмного забезпечення checkRole, яке динамічно обмежує доступ до маршрутів API на основі ролей.

```
export default function (role: string) {
  return function (req: Request, res: Response, next: NextFunction) {
    if (req.method === "OPTIONS") {
      next();
    }

    try {
      if (!req.user) {
        return next(ApiError.forbidden('Користувач не авторизований'));
      }

      if (req.user.role !== role) {
        return next(ApiError.forbidden('Немає доступу (потрібна роль ' + role + ')'));
      }

      next();
    } catch (e) {
      return next(ApiError.forbidden('Помилка перевірки прав'));
    }
  };
}
```

Рисунок 3.4 – Код Middleware для перевірки ролі

Розглянемо детально алгоритм роботи представленого модуля системи безпеки. Функція приймає строковий параметр role і повертає саму функцію-обробник HTTP-запиту. Таке архітектурне рішення забезпечує максимальну гнучкість, дозволяючи захищати різні ендпоінти API єдиним модулем, просто передаючи йому потрібну роль під час налаштування маршрутизатора.

Розглянемо, як саме працює цей код. Насамперед, як було і в попередньому прикладі, система перевіряє тип запиту: якщо це службовий

запит браузера, він пропускається далі без перевірок, щоб не блокувати нормальне завантаження сайту. Основна логіка обгорнута в захисний блок try...catch для уникнення збоїв. Спочатку програма перевіряє, чи взагалі існують дані про користувача. Якщо їх немає – це означає, що людина не пройшла авторизацію, тому система одразу зупиняє процес і видає помилку про відсутність доступу.

Якщо ж користувач авторизований, починається найголовніше – перевірка його прав. Система бере фактичну роль користувача та порівнює її з тією, яка вимагається для виконання дії. Якщо ролі не збігаються, доступ миттєво блокується з повідомленням про нестачу прав. А якщо все добре – сервер дозволяє виконати потрібну операцію. Крім того, якщо під час цих перевірок станеться якась непередбачувана технічна помилка, спрацює механізм catch. Він просто повідомить клієнту про збій перевірки прав, не розкриваючи жодних внутрішніх секретів роботи сервера. Такий підхід робить захист системи надійним і водночас дуже зручним у налаштуванні.

3.3.3 Безпека фінансових транзакцій

Оскільки розроблювана інформаційна система передбачає оформлення платних підписок на послуги доставки раціонів, обробка фінансових транзакцій є одним із найбільш критичних векторів атаки. Зберігання та обробка даних банківських карток користувачів вимагає проходження складної процедури сертифікації на відповідність міжнародному стандарту безпеки.

Для уникнення цих ризиків та забезпечення максимального рівня безпеки фінансових операцій, архітектуру системи було побудовано на принципі делегування відповідальності. Процес еквайрингу повністю інтегровано з платіжним шлюзом Stripe. Завдяки цьому розроблювана система функціонує без збереження жодних конфіденційних платіжних даних на власній базі PostgreSQL.

Безпека процесу оплати забезпечується **двома** ключовими механізмами.

Токенізація на стороні клієнта. Коли користувач ініціює оплату, сервер створює об’єкт PaymentIntent у системі Stripe і отримує унікальний одноразовий токен. Цей токен передається на клієнтський застосунок, де за допомогою бібліотеки Stripe Elements генерується захищена форма для введення реквізитів. Дані з цієї форми шифруються і відправляються безпосередньо на сервери Stripe, оминаючи бекенд розроблюваної системи.

Криптографічна верифікація Webhooks. Найбільш вразливим місцем у таких інтеграціях є момент, коли сервер отримує повідомлення про успішну оплату для того, щоб оновити статус замовлення. Якщо сервер сліпо довірятиме вхідним HTTP-запитам, зловмисник зможе надіслати фальшивий запит і безкоштовно активувати тариф. Для запобігання цьому використовується механізм перевірки цифрового підпису вебхуків.

На рисунку 3.5 представлено реалізацію серверного контролера, який безпечно обробляє асинхронні повідомлення від платіжної системи Stripe.

```
        webhook = async (req: Request, res: Response, next: NextFunction) => {
            const sig = req.headers['stripe-signature'];
            const endpointSecret = process.env.STRIPE_WEBHOOK_SECRET;
            let event;
            try {
                event = stripe.webhooks.constructEvent(req.body, sig as string, endpointSecret
as string);
            } catch (err: any) {
                return res.status(400).send(`Webhook Error: ${err.message}`);
            }

            if (event.type === 'checkout.session.completed') {
                const session = event.data.object as Stripe.Checkout.Session;
                const orderId = session.metadata?.orderId;

                if (orderId) {
                    await orderService.confirmPayment(orderId, session.id);
                }
            }

            res.json({ received: true });
        }
```

Рисунок 3.5 – Код контролеру для системи Stripe

Робота цього модуля розпочинається з вилучення криптографічного підпису із заголовка HTTP-запиту stripe-signature та отримання секретного ключа вебхука з конфігурації середовища. Критично важливим етапом є використання бібліотечного методу stripe.webhooks.constructEvent усередині захисного блоку try...catch. Ця функція здійснює верифікацію цифрового підпису запиту. Якщо підписи не збігаються, система перехоплює виняток, логує помилку та негайно повертає статус 400 Bad Request, ефективно відкидаючи підроблену транзакцію.

У разі успішної криптографічної перевірки, безпечно розпаковані дані обробляються залежно від типу події. У даному фрагменті коду система перевіряє настання події 'checkout.session.completed', яка сигналізує про те, що клієнт успішно завершив процес оплати та кошти були списані. З об’єкта події вилучається сесія Checkout, з метаданих якої отримується унікальний ідентифікатор замовлення. Ці метадані були попередньо закладені сервером під час генерації посилання на оплату, що дозволяє тепер точно ідентифікувати, за яке саме замовлення надійшли кошти.

На фінальному етапі, переконавшись у наявності ідентифікатора замовлення, контролер делегує виконання бізнес-логіки сервісному шару, викликаючи асинхронний метод orderService.confirmPayment(). Цей метод відповідає за взаємодію з базою даних та оновлення статусу відповідного замовлення на оплачене. Після успішного виконання всіх операцій сервер обов’язково надсилає платіжній системі відповідь у форматі JSON: { received: true }. Це є необхідним технічним підтвердженням для Stripe, що вебхук був успішно доставлений та опрацьований, що запобігає повторному автоматичному надсиланню цього ж сповіщення.

3.3.4 Валідація вхідних даних

У будь-якій інформаційній системі, що взаємодіє з користувачами, неконтрольовані вхідні дані є основним вектором потенційних атак та

найчастішою причиною порушення цілісності бази даних. Базове правило веббезпеки полягає в тому, що сервер ніколи не повинен довіряти даним, отриманим від клієнта. Тому строгий механізм валідації (перевірки) вхідних даних є обов'язковим бар'єром, який захищає бізнес-логіку застосунку від некоректних параметрів та запобігає аварійному завершенню роботи сервера.

Для забезпечення максимальної надійності у розробленій системі реалізовано комплексний **дворівневий підхід** до валідації.

Клієнтська валідація, яка відповідає переважно за користувацький досвід. Вона перевіряє правильність заповнення форм, наприклад, форматування електронної пошти, складність пароля або обов'язковість заповнення адреси, ще до відправки запиту на сервер. Це дозволяє миттєво надавати користувачу візуальні підказки про помилки та суттєво зменшує кількість невалідних мережевих запитів.

Серверна валідація, що виступає головним, остаточним механізмом захисту. Оскільки зловмисники можуть легко обійти клієнтську валідацію, відправляючи HTTP-запити безпосередньо до API, наприклад, через Postman, сервер зобов'язаний самостійно перевіряти кожен байт вхідної інформації.

Далі на рисунку 3.6 представлено програмну реалізацію проміжного програмного забезпечення для серверної валідації вхідних даних з використанням бібліотеки типізації та валідації Zod.

Розглянемо детально алгоритм роботи цього захисного модуля. Функція validate приймає як аргумент попередньо описану схему валідації і повертає асинхронний обробник запиту для фреймворку Express. Це дозволяє легко підключати цей модуль до будь-якого маршруту, передаючи відповідну схему, наприклад, схему реєстрації чи схему створення замовлення.

Виконання коду проходить через наступні логічні **етапи**, які описано нижче.

Ініціалізація перевірки. Всередині блоку try викликається асинхронний метод schema.parseAsync(). На відміну від простої перевірки лише тіла запиту, цей метод перевіряє одразу три ключові компоненти HTTP-запиту: body, query та params. Такий підхід гарантує повну перевірку всього вхідного контексту.

Успішна валідація. Якщо всі надіслані клієнтом дані повністю відповідають суворим правилам, описаним у переданій схемі, тобто правильні типи даних, дотримання лімітів довжини рядків, коректність email тощо, то функція викликає next(), дозволяючи серверу безпечно перейти до виконання основної бізнес-логіки.

Перехоплення помилок. Якщо дані містять помилки або небезпечні ін'єкції, метод parseAsync негайно генерує виняток, який перехоплюється блоком catch.

Форматування та деталізація помилок. Система перевіряє, чи є помилка саме результатом валідації Zod. Якщо так, виконується парсинг масиву помилок error.issues. За допомогою методів map та join, система збирає детальний опис для кожної помилки, об'єднуючи шлях до некоректного поля та саму причину відхилення.

Блокування запиту. Сформований текстовий рядок з описом усіх знайдених помилок передається у глобальний обробник через кастомний клас ApiError.badRequest. Клієнт миттєво отримує HTTP-статус 400 Bad Request зі зрозумілим повідомленням, наприклад: body.password – пароль має містити мінімум 8 символів, а виконання контролера повністю блокується.

```
export const validate = (schema: z.ZodType) =>
  async (req: Request, res: Response, next: NextFunction) => {
    try {
      await schema.parseAsync({
        body: req.body,
        query: req.query,
        params: req.params,
      });
      next();
    } catch (error) {
      if (error instanceof ZodError) {
        const errorMessages = error.issues
          .map((issue) => `${issue.path.join('.')} - ${issue.message}`)
          .join(', ');

        return next(ApiError.badRequest(`Помилка валидації: ${errorMessages}`));
      }

      next(error);
    }
  };
```

Рисунок 3.6 – Код Middleware для валідації вхідних даних

Такий архітектурний підхід повністю відокремлює логіку валідації від бізнес-логіки, робить код чистим, безпечним та гарантує, що до бази даних потрапить виключно правильна інформація.

3.4 Реалізація інтерфейсу

Після реалізації основних компонентів та налаштування архітектури взаємодії з сервером було розроблено візуальний інтерфейс клієнтського застосунку. Ключовим етапом життєвого циклу замовлення є взаємодія користувача зі сторінкою оформлення підписки.

На рисунку 3.7 наведено реалізацію інтерфейсу оформлення підписки.

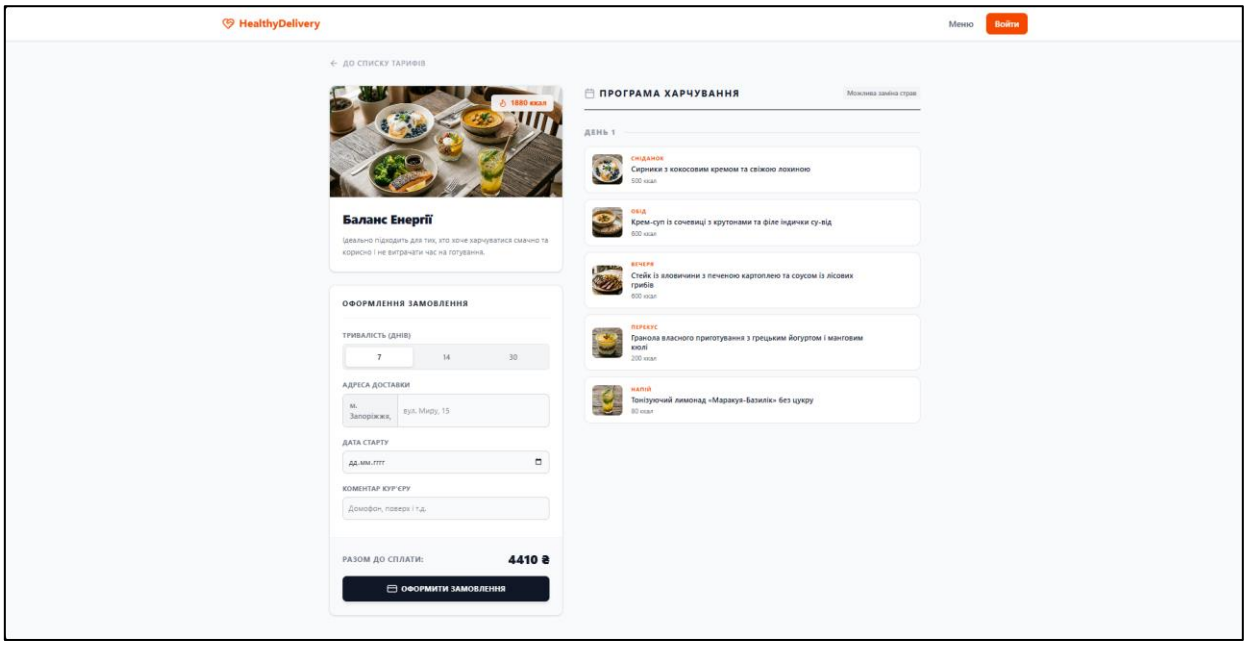


Рисунок 3.7 – Інтерфейс оформлення підписки

На цій сторінці користувач має змогу перевірити обраний тарифний план, налаштувати деталі підписки та вказати логістичні дані. Інтерфейс сторінки умовно розділено на три логічні блоки:

– *блок інформації про раціон*: відображає назву обраного меню, його енергетичну цінність та базову вартість за один день;

– *блок конфігурації підписки*: дозволяє користувачу обрати тривалість доставки – завдяки використанню реактивного стану React, при зміні кількості днів система без перезавантаження сторінки перераховує підсумкову суму замовлення;
– *блок контактних даних та логістики*: містить форму для введення точної адреси доставки, номеру телефону та, за потреби, додаткових коментарів для кур’єра.

Для забезпечення цілісності даних та уникнення помилок на сервері, форма оснащена механізмом клієнтської валідації. Якщо користувач залишає обов’язкові поля порожніми або вводить некоректні дані, система підсвічує відповідні елементи форми червоним кольором та виводить попереджувальні текстові підказки, тимчасово блокуючи можливість переходу до наступного кроку.

Після успішного заповнення всіх необхідних полів та натискання кнопки підтвердження, зібрані дані інкапсулюються в об’єкт та передаються до методу createOrder сервісу orderApi для запису в базу даних та підготовки до етапу оплати.

Далі було розроблено захищену зону для адміністрування системи. Однією з ключових сторінок є панель управління тарифними планами, яка дозволяє адміністратору гнучко налаштовувати асортимент сервісу.

На рисунку 3.8 наведено реалізацію інтерфейсу сторінки управління тарифами в панелі адміністратора.

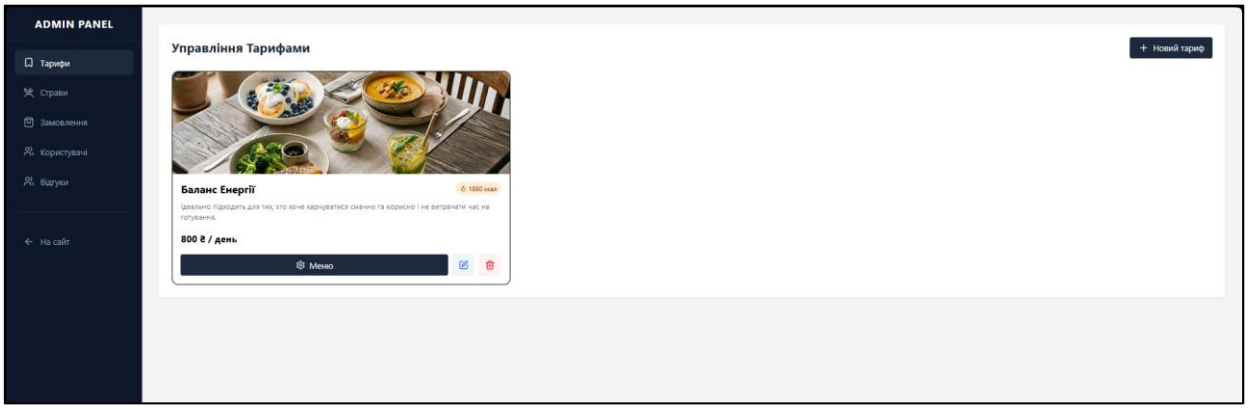


Рисунок 3.8 – Інтерфейс адміна

На цій сторінці адміністратор має змогу переглядати існуючі тарифні плани, редагувати їхні властивості та створювати нові. Інтерфейс сторінки умовно розділено на такі логічні блоки:

– *бокова навігаційна панель* забезпечує швидкий доступ до основних модулів управління системою, що дозволяє зручно перемикатися між робочими процесами;
– *верхня панель дій* містить заголовок поточного розділу та головну кнопку «Новий тариф» для ініціалізації форми створення нової підписки;
– *блок списку тарифів* відображає існуючі підписки у вигляді інтерактивних карток – кожна картка містить візуальне зображення, назву, показник добової калорійності, стислий опис та базову вартість за день;
– *блок керування контентом* розміщений у нижній частині кожної картки і включає кнопку переходу до редагування детального «Меню» для цього тарифу, а також іконки для швидкого редагування параметрів або видалення тарифу з бази даних.

Для оптимізації роботи адміністратора дані підвантажуються асинхронно за допомогою інструменту RTK Query, що забезпечує миттєве оновлення інформації на екрані при внесенні змін до асортименту без необхідності повного перезавантаження сторінки.

Для забезпечення логістичних процесів та зручної взаємодії з персоналом розроблено спеціалізований інтерфейс – панель кур’єра (див. рис. 3.9). Цей модуль призначений для управління маршрутами та відстеження процесу доставки у режимі реального часу.

На цій сторінці кур’єр може переглядати призначені йому завдання на обраний день та звітувати про їх виконання. Інтерфейс сторінки умовно розділено на три логічні блоки:

– *блок вибору дати* містить інтерактивний календар, за допомогою якого співробітник може переглядати свої маршрутні листи як на поточний робочий день, так і на майбутні чи минулі періоди;

– *блок статистики виконання*, це динамічна панель із прогрес-баром, яка відображає загальну кількість призначених доставок, кількість замовлень, що знаходяться «У дорозі», та кількість вже «Доставлених» раціонів – це дозволяє миттєво оцінити ступінь виконання робочої зміни;
– *робоча область списку доставок* відображає детальний перелік адрес, контактних даних клієнтів та коментарів до замовлень на обрану дату.

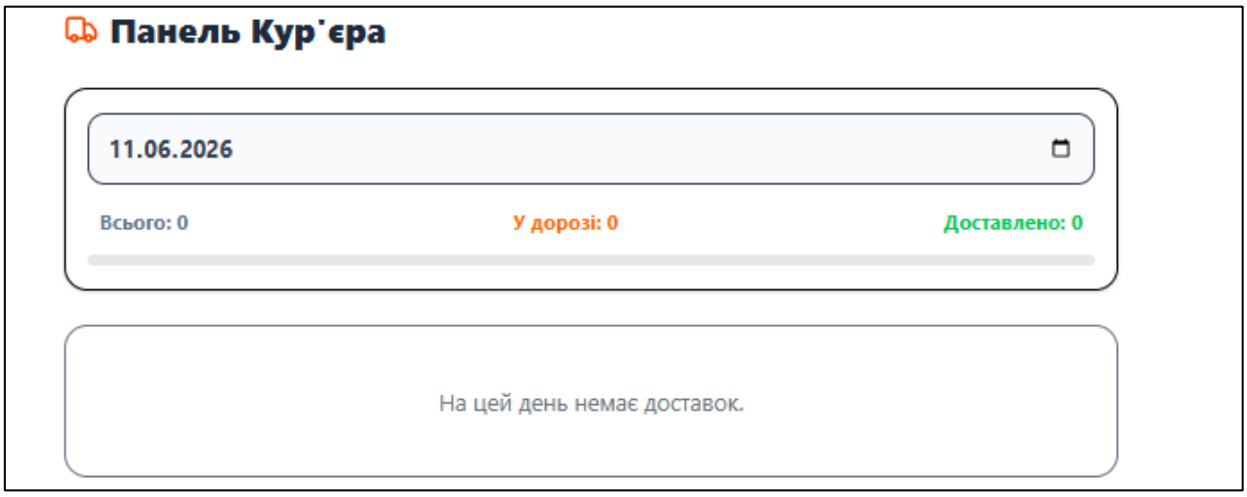


Рисунок 3.9 – Інтерфейс кур’єра

Управління станом у цьому модулі побудовано таким чином, що при зміні статусу окремої доставки кур’єром, прогрес-бар автоматично оновлюється, а відповідний PATCH-запит миттєво відправляється на бекенд для синхронізації змін із загальною базою даних.

Для автоматизації виробничих процесів приготування їжі та зведення замовлень було спроєктовано термінал кухні, який слугує основним робочим інструментом для кухарів та пакувальників сервісу.

Далі на рисунку 3.10 наведено реалізацію інтерфейсу терміналу кухні.

Термінал надає персоналу кухні чітку інформацію про обсяги виробництва. Інтерфейс сторінки розділено на наступні блоки:

– *блок управління зміною* містить заголовок модуля та зручний компонент вибору дати, що дозволяє кухарям бачити план

приготування страв на завтрашній або будь-який інший обраний день;

– *блок «Список страв»* автоматично підраховує та виводить загальну кількість необхідних порцій кожної окремої страви на день – цей модуль позбавляє персонал необхідності вручну вираховувати загальну кількість страв із різних підписок, повністю усуваючи ризик людської помилки при плануванні закупівель та готуванні;
– *блок «Замовлення до роботи»*, це інтерактивна область для відстеження процесу пакування готових раціонів по пакетах для конкретних клієнтів.

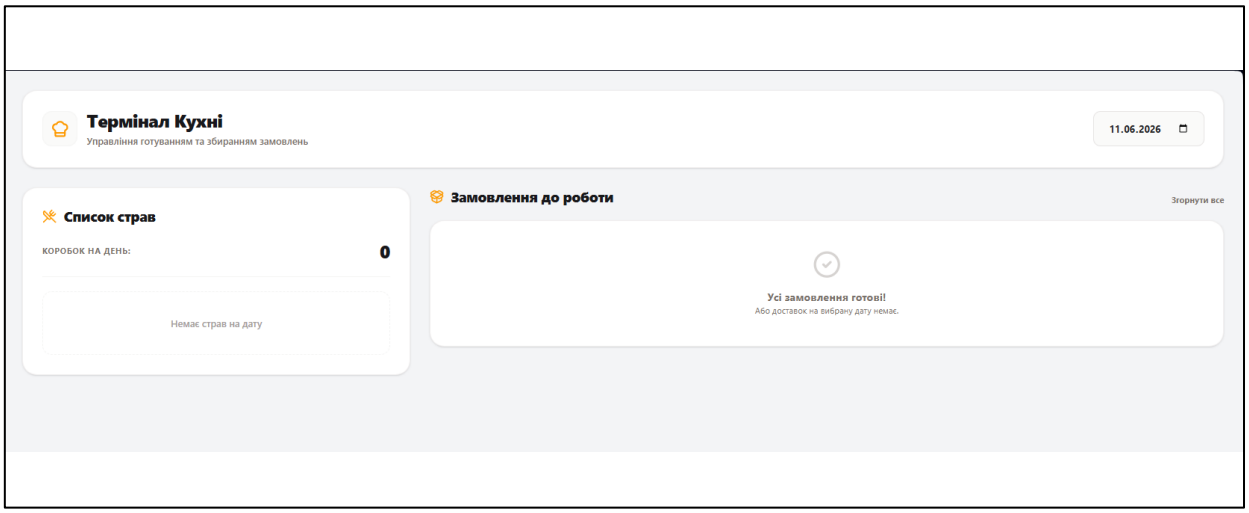


Рисунок 3.10 – Інтерфейс кухні

Агрегація даних про кількість необхідних порцій виконується виключно на стороні сервера баз даних за допомогою складних запитів інструменту Prisma ORM. Це дозволяє гарантувати абсолютну точність розрахунків та суттєво зменшує навантаження на процесор клієнтського пристрою, на якому відкрито термінал кухні.

Таким чином, успішна реалізація адаптивних користувацьких та службових інтерфейсів забезпечила повноцінне візуальне втілення закладеної бізнес-логіки full-stack вебзастосунку, що дозволяє перейти до етапу комплексного тестування його працездатності.

3.5 Тестування системи

Далі наведено тестування системи, а саме процесу замовлення. Для проведення тестування обрано фреймфрок Jest, який є гарним рішенням для тестування TypeScript коду, на якому написана система.

Jest обрано за його простоту, швидкість та роботу з коробки, що пришвидшує тестування. Також він має Mock функції, які є важливими для юніт-тестів та генерацію звітів про покриття коду тестами.

3.5.1 Модульне тестування

Модульне тестування дозволяє перевірити коректність роботи кожної окремої функцій додатку в ізольованому середовищі. Сенс у тому, щоб писати тести для кожної нетривіальної функції чи сервісу. Що дозволяє виявляти помилки при розширенні функціоналу.

В межах створеної системи особлива увага приділяється тестуванню серверних сервісів, щоб перевірити коректність оплати та створення замовлень.

Так як при модульному тестуванні треба уникнути звернення до бази даних, було створено та налаштовано mock об’єкт для Prisma ORM.

На рисунку 3.11 наведено код модульного тесту для сервісу обробки замовлень.

На представленому рисунку зображено процес модульного тестування методу create контролера OrderController з використанням фреймворку Jest. Тестовий набір обгорнутий у блок describe і містить два ізольованих сценарії, які перевіряють основну бізнес-логіку створення підписки та обробку виняткових ситуацій.

У блоці попередніх налаштувань beforeEach для кожного тесту ініціалізуються макети об’єктів запиту, відповіді та наступного middleware.

Після виконання кожного тесту в блоці afterEach викликається команда jest.clearAllMocks(), яка очищує історію викликів макетів для забезпечення повної незалежності тестів один від одного.

```
describe('OrderController - Створення замовлення', () => {
  let mockRequest: any;
  let mockResponse: any;
  let mockNext: any;

  beforeEach(() => {
    mockResponse = {
      json: jest.fn(),
    };
    mockNext = jest.fn();
  });
  afterEach(() => {
    jest.clearAllMocks();
  });
  it('повинен успішно створити замовлення, якщо користувач авторизований', async
() => {
    mockRequest = {
      user: { id: 'user-123', email: 'test@mail.com', role: 'USER' },
      body: { tariffId: 'tariff-1', address: 'м. Запоріжжя, вул. Соборна, 1' }
    };
    const expectedOrder = { id: 'order-999', ...mockRequest.body, userId: 'user-123' };
    (orderService.create as any).mockResolvedValue(expectedOrder);
    await orderController.create(mockRequest, mockResponse, mockNext);
    expect(orderService.create).toHaveBeenCalledWith('user-123', expect.any(Object));
    expect(mockResponse.json).toHaveBeenCalledWith(expectedOrder);
    expect(mockNext).not.toHaveBeenCalled();
  });
  it('повинен передавати помилку (ApiError.forbidden), якщо користувач не
авторизований', async () => {
    mockRequest = {
      user: undefined,
      body: { tariffId: 'tariff-1', address: 'вул. Тестова' }
    };
    await orderController.create(mockRequest, mockResponse, mockNext);
    expect(orderService.create).not.toHaveBeenCalled();
    expect(mockNext).toHaveBeenCalledWith(expect.any(ApiError));
    const errorPassedToNext = (mockNext as any).mock.calls[0][0];
    expect(errorPassedToNext.message).toBe('Користувач не авторизований');
  });
});
```

Рисунок 3.11 – Код модульного тестування

Перший тестовий сценарій перевіряє поведінку системи за умови успішної авторизації клієнта. У макеті запиту моделюються валідні дані

авторизованого користувача та тіло запиту з ідентифікатором тарифу та адресою доставки. За допомогою методу mockResolvedValue підміняється реальна відповідь сервісу orderService.create, яка повертає очікуваний об’єкт замовлення expectedOrder. Наприкінці тесту за допомогою тверджень expect перевіряється, що:

- сервіс створення замовлень був викликаний із правильним ID користувача;
- контролер успішно відправив клієнту відповідь у форматі JSON із даними створеного замовлення;
- функція переходу до наступного middleware mockNext не викликалася, що свідчить про відсутність помилок під час виконання операції.

Другий тестовий сценарій моделює ситуацію, коли неавторизований користувач намагається надіслати запит на створення підписки (поле user має значення undefined). Тест перевіряє захисний механізм контролера. Завдяки налаштованим очікуванням доведено, що в такому випадку бізнес-сервіс orderService.create взагалі не викликається, а керування потоком виконання передається далі через функцію mockNext, яка отримує об’єкт винятку класу ApiError. Останнє твердження тесту перевіряє текстове повідомлення помилки, підтверджуючи коректність роботи системи безпеки та обробника помилок на рівні контролера.

3.5.2 Інтеграційне тестування

На відміну від модульних тестів, які перевіряють компоненти в повній ізоляції, інтеграційне тестування призначене для перевірки коректності взаємодії кількох модулів системи одночасно. Головною метою цього етапу є валідація процесів, що охоплюють маршрутизацію HTTP-запитів, роботу Middleware, механізми валідації вхідних даних, а також реальну взаємодію з базою даних PostgreSQL.

Для автоматизації інтеграційних перевірок API-ендпоінтів було використано бібліотеку Supertest у поєднанні з тестовим середовищем Jest. Це дозволяє імітувати реальні мережеві запити до сервера та аналізувати HTTP-відповіді.

На рисунку 3.12 наведено фрагмент коду інтеграційного тесту для API керування замовленнями.

```
describe('Інтеграційне тестування API Замовлень', () => {
  const secretKey = process.env.JWT_ACCESS_SECRET || 'random_secret_key';
  const validToken = jwt.sign(
    { id: '123e4567-e89b-12d3-a456-426614174000', email: 'test@mail.com', role: 'USER' },
    secretKey,
    { expiresIn: '15m' }
  );
  it('повинен дійти до БД і повернути 400, якщо передано неіснуючий ID тарифу', async () => {
    const orderData = {
      tariffId: "123e4567-e89b-12d3-a456-426614174000",
      days: 7,
      startDate: "2026-06-01",
      address: "м. Запоріжжя, вул. Соборна, 1",
      comment: "Код домофону 123"
    };
    const response = await request(app)
      .post('/api/order')
      .set('Authorization', `Bearer ${validToken}`)
      .send(orderData);

    expect(response.status).toBe(400);
    expect(response.body).toHaveProperty('message');
  });
  it('повинен повернути статус 400, якщо адреса не з Запоріжжя', async () => {
    const invalidOrderData = {
      tariffId: "123e4567-e89b-12d3-a456-426614174000",
      days: 7,
      startDate: "2026-06-01",
      address: "м. Київ, вул. Хрещатик, 1"
    };
    const response = await request(app)
      .post('/api/order')
      .set('Authorization', `Bearer ${validToken}`)
      .send(invalidOrderData);
    expect(response.status).toBe(400);
    expect(response.body.message).toContain('лише в межах м. Запоріжжя');
  });
  it('повинен повернути статус 401 або 403, якщо немає токена', async () => {
    const response = await request(app)
      .post('/api/order')
      .send({ address: "м. Запоріжжя" });

    expect([401, 403]).toContain(response.status);
  });
});
```

Рисунок 3.12 – Код інтеграційного тестування

На представленому рисунку зображено структуру інтеграційного тесту. На початку блоку здійснюється підготовка до автентифікації користувача: за допомогою бібліотеки jsonwebtoken та секретного ключа з конфігурації process.env.JWT_ACCESS_SECRET динамічно генерується валідний токен доступу, який містить корисне навантаження з терміном дії 15 хвилин. Цей токен використовується в наступних сценаріях для авторизації запитів через HTTP-заголовок Authorization.

Тестовий набір містить **три** ключові сценарії, що перевіряють стійкість системи до передачі некоректних даних та роботу механізмів безпеки.

Перший сценарій перевіряє поведінку сервера у випадку передачі неіснуючого або неправильного ідентифікатора тарифного плану (tariffId). Об'єкт orderData містить згенерований випадковий UUID. Запит відправляється методом POST на ендпоінт /api/order з передачею створеного JWT-токена. Перевірка результату за допомогою методів toBe(400) та toHaveProperty('message') доводить, що система успішно обробляє помилку на рівні бази даних або сервісу, повертаючи клієнту статус 400 Bad Request замість аварійного завершення роботи бекенду.

Другий сценарій валідує критично важливу бізнес-вимогу – обмеження географії обслуговування. У макеті даних invalidOrderData вказується адреса, що не належить до цільового регіону. Тест виконує аналогічний POST-запит і перевіряє, що впроваджений у систему Zod-валідатор перехоплює запит, повертає статус 400 та надсилає чітке текстове повідомлення: «лише в межах м. Запоріжжя».

Третій сценарій спрямований на перевірку захищеності маршруту від неавторизованих користувачів. У цьому тесті POST-запит виконується взагалі без встановлення заголовка авторизації та передачі токена. Кінцеве твердження expect([401, 403]).toContain(response.status) гарантує, що система безпеки заблокує доступ до створення замовлення та поверне статус 401 Unauthorized або 403 Forbidden.

Результати виконання цього набору тестів демонструють високу

надійність розробленого REST API, коректність відпрацювання шарів валідації даних та захищеність системи від несанкціонованого доступу.

3.5.3 Тестування навантаженням

Для перевірки стійкості архітектури розробленого вебзастосунку до пікових навантажень та визначення лімітів продуктивності серверної частини було проведено навантажувальне тестування. Головною метою цього етапу є дослідження поведінки системи при одночасній роботі великої кількості користувачів, виявлення можливих затримок у базі даних PostgreSQL, а також перевірка ефективності налаштованого кешування Redis.

Об’єктом дослідження виступив найважливіший для клієнтської частини ендпоінт – отримання детальної інформації та меню конкретного тарифного плану за його ідентифікатором за допомогою HTTP-запиту: GET /api/tariffs/50cc2e5e-e9e5-4ac9-b87b-f89ea3234e90.

Сценарій та конфігурація тесту. Для імітації реальних умов експлуатації було налаштовано групу віртуальних потоків зі специфічними параметрами розгону:

- кількість потоків: 100 віртуальних користувачів;
- період розгону: 10 секунд, тобто система додавала по 10 нових користувачів щосекунди для плавного підвищення навантаження;
- кількість циклів: 5 ітерацій для кожного потоку;
- загальний обсяг трафіку: 500 успішно згенерованих та надісланих HTTP-запитів;

На рисунку 3.13 продемонстровано точні параметри конфігурації HTTP-запиту, що був надісланий тестовим сервером JMeter.

На представленому рисунку видно, що запит повністю відповідає специфікації REST API нашого бекенду. Параметри адресації вказують на локальний хост та мережевий порт сервісу 5000. У полі Path прописано точний

шлях до ресурсу в базі даних із префіксом /api/tariffs/ та передачею валідного рядка унікального ідентифікатора UUID. Метод запиту визначено як GET.

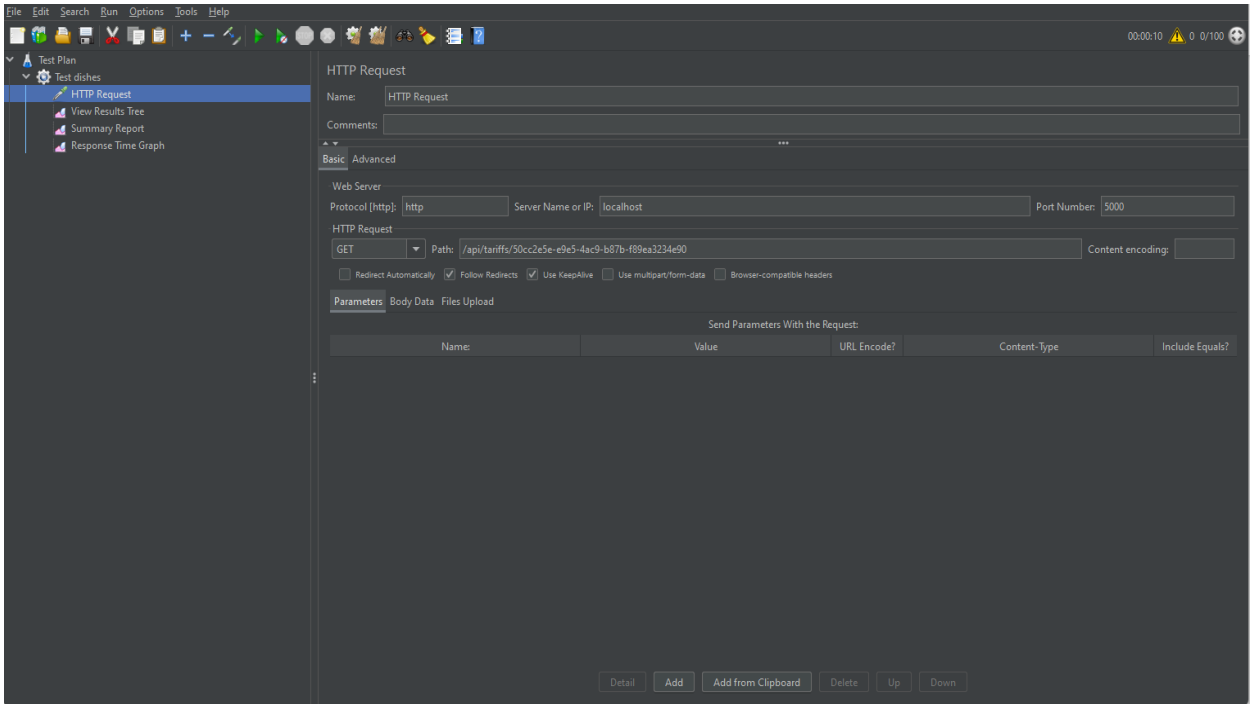


Рисунок 3.13 – Параметри конфігурації HTTP-запиту

Після запуску тестового сценарію фіксація результатів відбувалася за допомогою компонента View Results Tree.

На рисунку 3.14 наведено візуалізацію успішно виконаних запитів.

Аналіз інтерфейсу результатів показує, що всі надіслані HTTP-запити пройшли успішно, про що свідчить зелений маркер біля кожного елемента списку. У детальній аналітиці обраного потоку Test dishes 1-1 зафіксовано такі метрики продуктивності бекенд-платформи:

– Response code: 200 / Response message: OK – сервер безпомилково розпізнав усі вхідні запити та повернув стандартний статус успішного виконання;
– ContentType: application/json; charset=utf-8 – підтверджує, що бекенд коректно опрацював запит до бази даних і віддав валідну структуру даних JSON, яка містить опис та меню тарифу, готову для рендерингу на клієнтській стороні;

– Load time: 6 ms / Latency: 6 ms – час повної відповіді сервера на запит склав лише 6 мілісекунд, що є надзвичайно високим показником швидкодії.

Такий низький час затримки доводить ефективність розробленої архітектури: завдяки інтеграції ORM-інструменту Prisma, правильному індексуванню таблиць у PostgreSQL та кешуванню шару Redis, серверна частина не витрачає ресурси на повторні важкі SQL-вибірки, а миттєво віддає збережену структуру даних із пам’яті.

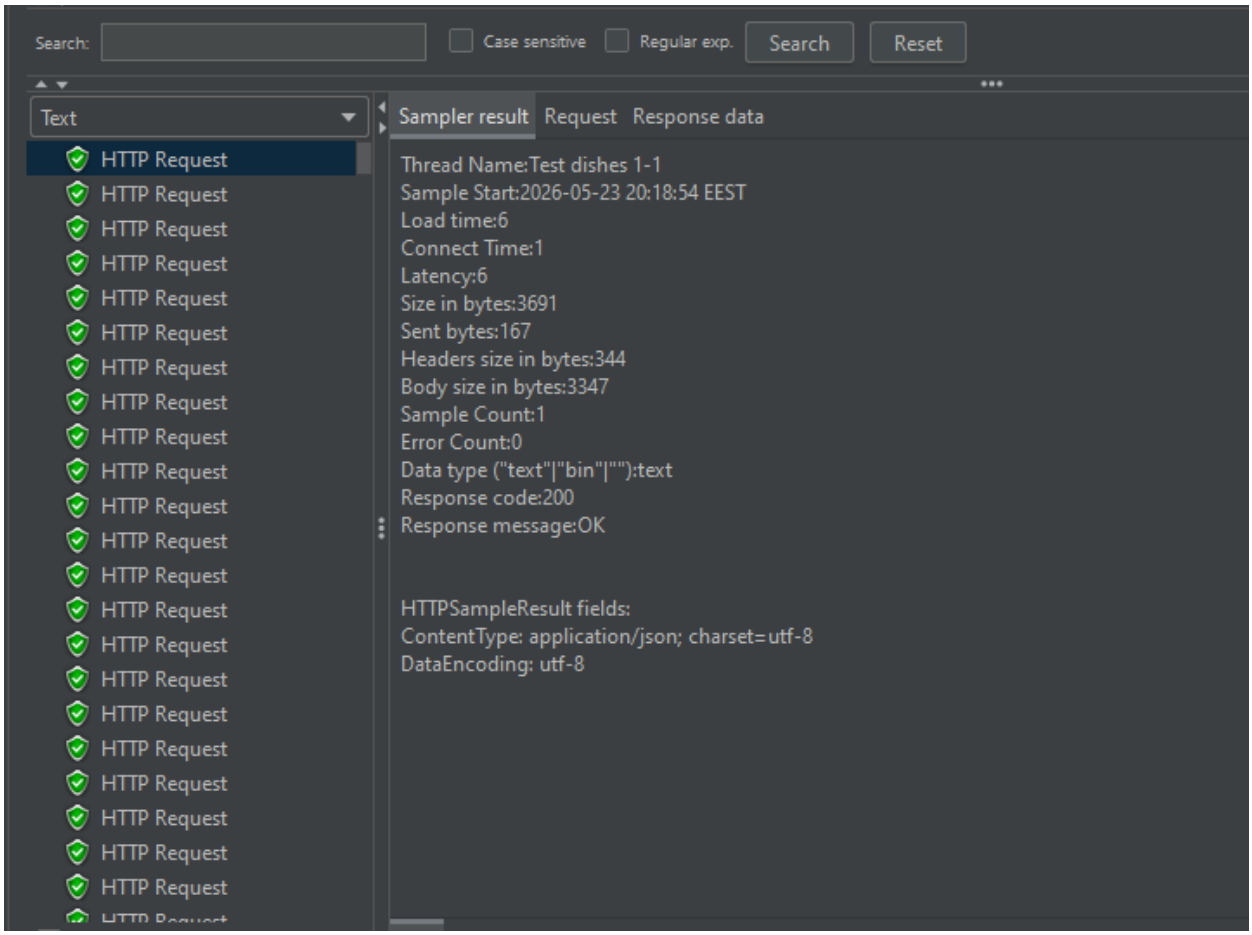


Рисунок 3.14 – Результат HTTP-запитів

Загальний відсоток помилок під час пікового навантаження склав 0.00%, що повністю підтверджує стабільність розробленого програмного забезпечення, його готовність до релізу та здатність стабільно обслуговувати реальних користувачів онлайн-сервісу.

ВИСНОВКИ

У кваліфікаційній роботі бакалавра вирішено актуальне завдання – розроблено спеціалізовану інформаційну систему онлайн-замовлення та доставки збалансованих раціонів харчування. Розроблений програмний продукт дозволяє повністю автоматизувати цикл взаємодії між клієнтами, кухнею, кур'єрами та адміністраторами. Для забезпечення високої продуктивності та надійності серверну частину реалізовано строго типізованою мовою TypeScript на платформі Node.js з використанням фреймворку Express.js, бази даних PostgreSQL та високошвидкісного in-memory сховища Redis. Клієнтську частину побудовано на базі бібліотеки React та інструментарію Redux Toolkit, що гарантує високу інтерактивність і плавність роботи користувацького інтерфейсу.

У першому розділі роботи проведено системний аналіз предметної області сервісів доставки їжі за підпискою. Вивчено існуючі аналоги, такі як глобальні агрегатори та локальні сервіси, і виявлено їхні недоліки у контексті комплексного управління раціонами на тривалий період. На основі цього сформульовано чіткі функціональні вимоги для кожної групи користувачів системи: клієнтів, адміністраторів, персоналу кухні та логістики. Здійснено порівняльний аналіз архітектурних підходів та науково обґрунтовано вибір сучасного стеку технологій, зокрема використання інструменту Prisma ORM для безпечного управління реляційними даними.

У другому розділі спроєктовано логічну та фізичну архітектуру системи. За допомогою інструментарію UML розроблено концептуальну модель у вигляді діаграми варіантів використання, яка візуалізує права та обов'язки акторів платформи. Спроєктовано нормалізовану реляційну схему бази даних, що відображає складні зв'язки між профілями користувачів, тарифами, кулінарними стравами та щоденним графіком доставок. За допомогою діаграми послідовностей змодельовано бізнес-процес оформлення замовлення та безпечної інтеграції з платіжним шлюзом Stripe. Крім того, побудована

діаграма компонентів детально відобразила багаторівневу клієнт-серверну структуру вебзастосунку з підключенням сторонніх API-сервісів.

У третьому розділі продемонстровано практичну програмну реалізацію розробленого застосунку. Створено ключові модулі: реактивний клієнтський інтерфейс із функцією кастомізації меню та динамічним ціноутворенням, захищені REST API ендпоінти, а також систему автоматичної генерації завдань для кухні та логістики по днях. Особливу увагу приділено інформаційній безпеці: впроваджено JWT-автентифікацію, строге рольове розмежування доступу, багаторівневу валідацію вхідних даних за допомогою бібліотеки Zod та безпечну обробку фінансових транзакцій через цифрові підписи Webhooks. Для перевірки надійності системи проведено комплексне тестування: модульне та інтеграційне тестування бізнес-логіки за допомогою фреймворку Jest та бібліотеки Supertest , а також навантажувальне тестування бекенду за допомогою Apache JMeter. Отримані результати підтвердили коректність обробки запитів, захищеність від несанкціонованого доступу, нульовий відсоток помилок та високу швидкодію сервера під час пікових навантажень.

Практичне значення одержаних результатів полягає у створенні готового до розгортання програмного рішення, здатного суттєво оптимізувати бізнес-процеси, мінімізувати вплив людського фактору та скоротити операційні витрати компаній у сфері виробництва й доставки раціонів. Надалі розроблену систему доцільно масштабувати шляхом створення кросплатформового мобільного застосунку, впровадження системи push-сповіщень для координації кур’єрів, розширення модуля фінансової аналітики для адміністраторів, а також інтеграції алгоритмів штучного інтелекту для формування персоналізованих дієтичних рекомендацій.

Підсумовуючи вищезазначене, можна стверджувати, що поставлена мета кваліфікаційної роботи досягнута повною мірою, а розроблений програмний продукт є цілісним, стабільним і повністю готовим до впровадження та комерційної експлуатації.

ПЕРЕЛІК ПОСИЛАНЬ

1. TypeScript Documentation. URL: https://www.typescriptlang.org/docs/ (дата звернення: 26.03.2026).

2. RTK Query Overview. *Redux Toolkit*. URL: https://redux-toolkit.js.org/rtk-query/comparison (дата звернення: 28.03.2026).

3. About Node.js. URL: https://nodejs.org/en/about (дата звернення: 30.03.2026).

4. Korkmaz S. Інструменти об’єктно-реляційного відображення (ORM) та зв’язки з базами даних. 2025. *Hostragons Blog*. URL: https://www.hostragons.com/uk/блог/інструменти-orm-для-реляційного-відобра/ (дата звернення: 04.04.2026).

5. Redis на практичних прикладах. 2020. *DevZone*. URL: https://devzone.org.ua/post/redis-na-praktychnykh-prykladakh (дата звернення: 05.04.2026).

6. Відкрити stripe для бізнесу: прийом платежів онлайн без обмежень. Як відкрити stripe для бізнесу: повне керівництво. *Locman International Corporate Service.* URL: https://locman.com.ua/stripe-dlya-biznesu-pidklyuchennya-veryfikacziya-ta-pochatok-roboty/ (дата звернення: 05.04.2026).