МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Острозька академія»
Навчально-науковий інститут інформаційних технологій та бізнесу
Кафедра інформаційних технологій та аналітики даних

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня бакалавра

на тему: «**Розробка застосунку для організації замовлень їжі»**

Виконав: студент 4 курсу, групи КН-42
першого (бакалаврського) рівня вищої освіти
спеціальності 122 Комп'ютерні науки
освітньо-професійної програми «Комп'ютерні науки»
Свирида Андрій Юрійович

Керівник: *викладач кафедри інформаційних технологій та аналітики даних*
Мацевич Денис Володимирович

Рецензент: *кандидат технічних наук, доцент, доцент кафедри прикладної математики Донецького національного університету імені Василя Стуса Загоруйко Любов Василівна*

РОБОТА ДОПУЩЕНА ДО ЗАХИСТУ

Завідувач кафедри інформаційних технологій та аналітики даних _____
(проф., д.е.н. Кривицька О.Р.)

Протокол № 11 від «20 » травня 2026 р.

Острог, 2026

АНОТАЦІЯ
кваліфікаційної роботи
на здобуття освітнього ступеня бакалавра

Тема: Розробка та проєктування системи управління подіями з елементами штучного інтелекту.

Автор: Свирида Андрій Юрійович

Науковий керівник: викладач, фахівець-практик, Денис Володимирович Мацевич

Захищена «……»…… 2026 року.

Пояснювальна записка до кваліфікаційної роботи: *128 с., 1 рис., 12 табл., 5 додатків, 34 джерело.*

Ключові слова: організація спільних обідів, вебзастосунок, групові замовлення, облік балансів, FastAPI, React, PostgreSQL, SQLAlchemy, Redux Toolkit, Nginx, GitHub Actions.

Короткий зміст праці:
Метою кваліфікаційної роботи є проєктування та розробка вебзастосунку «LunchTogether» для автоматизації організації спільних обідів у колективах. Система підтримує створення груп, запрошення учасників, ведення каталогу ресторанів і страв, формування спільних замовлень, розподіл вартості доставки та автоматичне оновлення балансів. Серверна частина реалізована на Python з використанням FastAPI, SQLAlchemy 2.0 і PostgreSQL, а бізнес-логіку винесено у Workflow-класи з доступом до даних через Repository-шар. Клієнтська частина побудована на React, TypeScript, Vite, Redux Toolkit та RTK Query. Для автентифікації використано JWT-токени в HTTP-only cookies, а система прав поєднує глобальні ролі й групові дозволи. Розгортання виконано у промисловій конфігурації з HTTPS через Nginx, systemd-сервісом, резервним копіюванням, Sentry та GitHub Actions. Результатом роботи є готова до експлуатації full-stack система для зручної координації обідів і прозорого обліку витрат між учасниками групи.

_____

(підпис автора)

ANNOTATION
of qualification paper
for bachelor's degree

Title: System Design and Development of an Online Store for Cosmetic Products with Ingredient-Based

Filtering and a Personalized Recommendation System.

Author: Andrii Yuriyovych Svyryda

Scientific advisor: Lecturer, Practicing Specialist, Denys Volodymyrovych Matsevych.

Defended «……»…… 2026.

Explanatory note to the qualification thesis: 128 pages, 1 figure, 12 tables, 5 appendices, 34 references.

Keywords: SHARED LUNCHES, GROUP ORDERS, BALANCE TRACKING, WEB APPLICATION, FASTAPI, REACT, POSTGRESQL, SQLALCHEMY, REDUX TOOLKIT, NGINX, GITHUB ACTIONS.

Abstract:

The aim of this qualification work is to design and develop the "LunchTogether" web application for automating the organization of shared lunches in teams. The system supports group creation, member invitations, restaurant and dish catalog management, shared order creation, delivery cost distribution, and automatic balance updates. The server side is implemented in Python using FastAPI, SQLAlchemy 2.0, and PostgreSQL, with business logic organized into Workflow classes and data access handled through a Repository layer. The client side is built with React, TypeScript, Vite, Redux Toolkit, and RTK Query. Authentication is implemented using JWT tokens stored in HTTP-only cookies, while the permission system combines global roles and group-level permissions. The application is deployed in a production-oriented configuration with HTTPS via Nginx, a systemd service, database backups, Sentry monitoring, and GitHub Actions. The result is a ready-to-use full-stack system for convenient lunch coordination and transparent expense tracking among group members.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

Скорочення	Розшифровка
API	Програмний інтерфейс застосунку
CI/CD	Безперервна інтеграція та доставка
CRUD	Базові операції Create / Read / Update / Delete
ER	Модель «сутність-зв'язок»
JWT	JSON Web Token
ORM	Object-Relational Mapping
REST	Архітектурний стиль веб-API
SPA	Односторінковий веб-застосунок
SQL	Мова структурованих запитів
TLS	Протокол захищеної передачі даних
UI / UX	Інтерфейс та користувацький досвід
VPS	Віртуальний виділений сервер
СУБД	Система управління базою даних

ВСТУП

У сучасному корпоративному середовищі спільні обіди сприяють згуртованості команди, проте їх організація для великих колективів супроводжується значними координаційними витратами: збір замовлень, розподіл вартості доставки та відстеження взаєморозрахунків. Традиційні підходи (месенджери, таблиці) не забезпечують належної автоматизації та зручності.

Актуальність роботи обумовлена попитом на спеціалізовані внутрішні інструменти (internal tools), розвитком асинхронних Python-фреймворків (FastAPI [1]) та типобезпечних фронтенд-рішень (React [2]), що дозволяють створювати надійні додатки з гнучкими системами авторизації.

Об'єктом дослідження є процес розробки веб-додатку для автоматизації спільних обідів. Предметом — реалізація системи «LunchTogether» (FastAPI, PostgreSQL, React, Nginx, GitHub Actions).

Методи дослідження. Використано системний аналіз, SWOT-аналіз, use-case моделювання, UML [32], ER-моделювання та інтеграційне тестування.

Мета роботи — створення повнофункціональної платформи для управління групами, замовленнями та балансами учасників із розгортанням у промисловій конфігурації (HTTPS, CI/CD).

Завдання:

1. Проаналізувати предметну область та існуючі рішення.
2. Сформувати вимоги на основі персон та сценаріїв.
3. Спроєктувати архітектуру, базу даних та REST API.
4. Реалізувати серверну (FastAPI) та клієнтську (React) частини.
5. Розгорнути систему на VPS із автоматизацією (setup.sh, deploy.sh).
6. Виконати тестування та підготувати документацію.

Практичне значення полягає у створенні готової до використання платформи з промисловою конфігурацією, що підтверджує можливість її експлуатації реальними колективами.

Структура роботи. Робота складається з шести розділів, висновків, списку джерел та додатків. Розділ 1 присвячено аналізу та вимогам, розділ 2 — архітектурі, розділи 3-4 — реалізації бекенду та фронтенду, розділ 5 — розгортанню, розділ 6 — тестуванню.

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Проблематика організації спільних обідів

Організація обідів у групах від 10 учасників стикається з проблемами:

- Координація: збір замовлень у месенджерах призводить до втрати повідомлень та помилок.
- Розподіл витрат: ручний розрахунок часток доставки є трудомістким.
- Заборгованості: відсутність обліку взаєморозрахунків породжує фінансові конфлікти.
- Втрата історії: ціни та улюблені страви не зберігаються між замовленнями.
- Делегування: відсутність ролей перевантажує одного організатора.

Ключові сутності: Користувач, Група, Учасник, Запрошення, Ресторан, Замовлення (Initiated, Confirmed, Ordered, Finished), Позиція, Баланс.

1.2 Аналіз існуючих рішень

На ринку існує декілька класів інструментів, які частково покривають описану задачу, проте жоден з них не пропонує комплексного рішення для групової координації замовлень із автоматичним фінансовим обліком.

Месенджери (Telegram, Slack, Microsoft Teams). Дозволяють координувати замовлення у вільному текстовому форматі через групові чати, опитування та закріплені повідомлення. Інструменти універсальні та поширені, але не забезпечують структурованого обліку: позиції замовлення зберігаються як вільний текст, баланси між учасниками не обчислюються, історія губиться у стрічці повідомлень. Месенджери є інструментом комунікації, а не обліку.

Спеціалізовані сервіси доставки їжі (Glovo, Bolt Food, Uber Eats). Орієнтовані на індивідуальні замовлення кінцевого користувача. Хоч у деяких є режим групового замовлення, він обмежується спільним кошиком, а оплата зазвичай

належить одній особі. Сервіси не дозволяють зберегти інформацію про склад групи, історію її замовлень або баланси між її учасниками, оскільки кожне замовлення є самостійною трансакцією.

Електронні таблиці (Google Sheets, Excel). Можуть бути адаптовані для ведення обліку замовлень та балансів, але потребують значних зусиль на налаштування й підтримку. Інтерфейс не оптимізовано для мобільних пристроїв, формули потребують ручного супроводу, відсутні механізми авторизації та валідації введення. Електронна таблиця є інструментом для тих, у кого вже є власне рішення, а не готовою платформою.

Інструменти розподілу витрат (Splitwise, Settle Up). Спеціалізуються на обліку взаємних заборгованостей між людьми, мають мобільні застосунки та підтримують групи. Однак вони не покривають процес замовлення: користувач має все одно вручну зібрати позиції, погодити їх з учасниками і лише потім ввести підсумкову суму у систему обліку. Зв'язок між фактичним замовленням, його структурою та фінансовим записом залишається неявним.

1.2.1 Поглиблений огляд конкретних продуктів

Для уточнення картини нижче детальніше розглянуто чотири продукти, що часто застосовуються при координації спільних замовлень у командах. Кожен опис містить характерні сильні та слабкі сторони у контексті групової координації обідів.

Splitwise (https://www.splitwise.com). Один з найпопулярніших інструментів обліку спільних витрат. Сильні сторони: зрілий мобільний застосунок, підтримка груп та категорій, нагадування про борги, простий експорт у CSV. Слабкі сторони у контексті обіду: відсутність моделі «замовлення зі складом позицій» — користувач вводить лише підсумкову суму; немає каталогу страв чи ресторанів; немає поняття життєвого циклу замовлення (підтвердження, доставка); немає делегування прав між учасниками групи.

Settle Up (https://settleup.io). Чеський аналог Splitwise з акцентом на мінімізацію кількості необхідних переказів між учасниками. Сильні сторони: компактний інтерфейс, підтримка кількох валют, експорт у PDF. Слабкі сторони: ті ж, що й у Splitwise — облік виключно фінансових результатів, без структури замовлень та каталогу страв; немає інтеграції з електронною поштою для запрошень.

Glovo / Bolt Food (https://glovoapp.com, https://food.bolt.eu). Сервіси B2C-доставки їжі з обмеженою функцією «спільного кошика». Сильні сторони: реальний каталог ресторанів і страв, оплата онлайн, доставка. Слабкі сторони у контексті колективного обіду: оплата прив'язана до однієї людини; немає історії групи між замовленнями; немає обліку балансів між учасниками; немає делегування прав; неможливо повторно використати склад попередніх замовлень.

Google Sheets як шаблон (https://docs.google.com/spreadsheets). Поширений «саморобний» інструмент. Сильні сторони: повна гнучкість, низький поріг входу, безкоштовність, доступність із будь-якого пристрою. Слабкі сторони: відсутність структурованої моделі (типи даних, валідації, обмеження), мобільний UX табличного редактора слабкий, формули потребують ручного супроводу, немає механізму авторизації за ролями, кожна нова група починає шаблон з нуля.

Таблиця 1.2.1 узагальнює порівняння цих рішень із цільовою системою «LunchTogether» за ключовими функціональними критеріями.

Таблиця 1.2.1

Скорочений порівняльний аналіз існуючих рішень

Критерій	Месенджери	Сервіси доставки	Таблиці	Splitwise / Settle Up	LunchTogether
Структуроване замовлення	Ні	Частково	Частково	Ні	Так

Критерій	Месенджери	Сервіси доставки	Таблиці	Splitwise / Settle Up	LunchTogether
Фінансовий облік групи	Ні	Ні	Частково	Так	Так
Ролі та дозволи	Ні	Ні	Ні	Обмежено	Так
Історія групи та повторне використання	Ні	Частково	Частково	Ні	Так

Як видно з таблиці, жодне з існуючих рішень не поєднує одночасно всіх необхідних можливостей. Це обґрунтовує необхідність створення спеціалізованої платформи.

1.2.2 SWOT-аналіз цільової системи

Для систематичного огляду внутрішніх та зовнішніх факторів проєкту виконано SWOT-аналіз. Він фіксує сильні та слабкі сторони реалізованої системи, а також можливості та загрози зовнішнього середовища, що впливатимуть на її використання у реальних колективах.

Таблиця 1.2.2

Узагальнений SWOT-аналіз системи «LunchTogether»

Strengths	Weaknesses
Поєднання моделі групового замовлення, балансів і ролей	Немає онлайн-оплати та нативного мобільного застосунку
Типобезпечний стек і готова інфраструктура розгортання	Потрібна SMTP-конфігурація для запрошень
Самостійний хостинг без обов'язкових SaaS-залежностей	На старті система розрахована на один сервер
Opportunities	Threats

Strengths	Weaknesses
Попит на внутрішні офісні інструменти та локальні рішення	Конкуренція B2C-сервісів доставки
Можливість майбутніх інтеграцій з POS/email-доменами	Регуляторні зміни щодо персональних даних

Інтерпретація. Найвиразніша сильна сторона — комбінація моделі замовлення та фінансового обліку, якої не дають аналоги, у поєднанні з промисловою конфігурацією розгортання, готовою до реальної експлуатації. Найбільш чутлива слабкість — відсутність мобільних застосунків та онлайн-оплати, що частково компенсується адаптивною версткою. Серед зовнішніх можливостей особливо значущим є зростання попиту на корпоративні координаційні інструменти, а серед загроз — регуляторний тиск щодо персональних даних, що вже враховано у нефункціональних вимогах (див. розділ 1.6).

1.3 Цільова аудиторія та персони

Для уточнення вимог було сформовано три ключові персони, що відображають типових користувачів системи з різним рівнем відповідальності та доступу.

Персона А. «Олена — власник групи». Тімлід відділу розробки з 12 співробітниками. Двічі-тричі на тиждень організовує замовлення обідів. Має смартфон та робочий ноутбук, користується корпоративним месенджером. Її цілі: швидко зібрати замовлення, мінімізувати час на координацію, точно розподілити вартість доставки, не пам'ятати, хто кому винен. У системі очікує грати роль «Власника групи» з повним доступом до управління учасниками, замовленнями, балансами та налаштуваннями групи.

Персона Б. «Ігор — учасник групи». Розробник, учасник 1–2 робочих груп. Користується сервісом, щоб додавати власні позиції до замовлення, бачити, скільки

винен групі, та позначати улюблені страви. Не має потреби (а часто й бажання) керувати учасниками або редагувати чужі замовлення. У системі грає роль звичайного учасника з обмеженими дозволами.

Персона В. «Марія — заступник власника / супервайзер». Колега Олени, якій делеговано право ініціювати замовлення, коли Олена відсутня. Не повинна керувати учасниками групи або налаштуваннями, але повинна мати можливість запустити цикл замовлення та переглядати загальну аналітику. У системі грає роль «Supervisor Member» — пресет, що автоматично призначає права рівня «Viewer» на всі області плюс «Initiator» на замовлення.

Окремим актором є системний адміністратор — особа з повним доступом до всіх груп та користувачів системи, що використовується для модерування та підтримки.

1.4 Сценарії використання

На основі персон та функціональних областей сформовано діаграму прецедентів використання (див. Рис. 1.4.1).

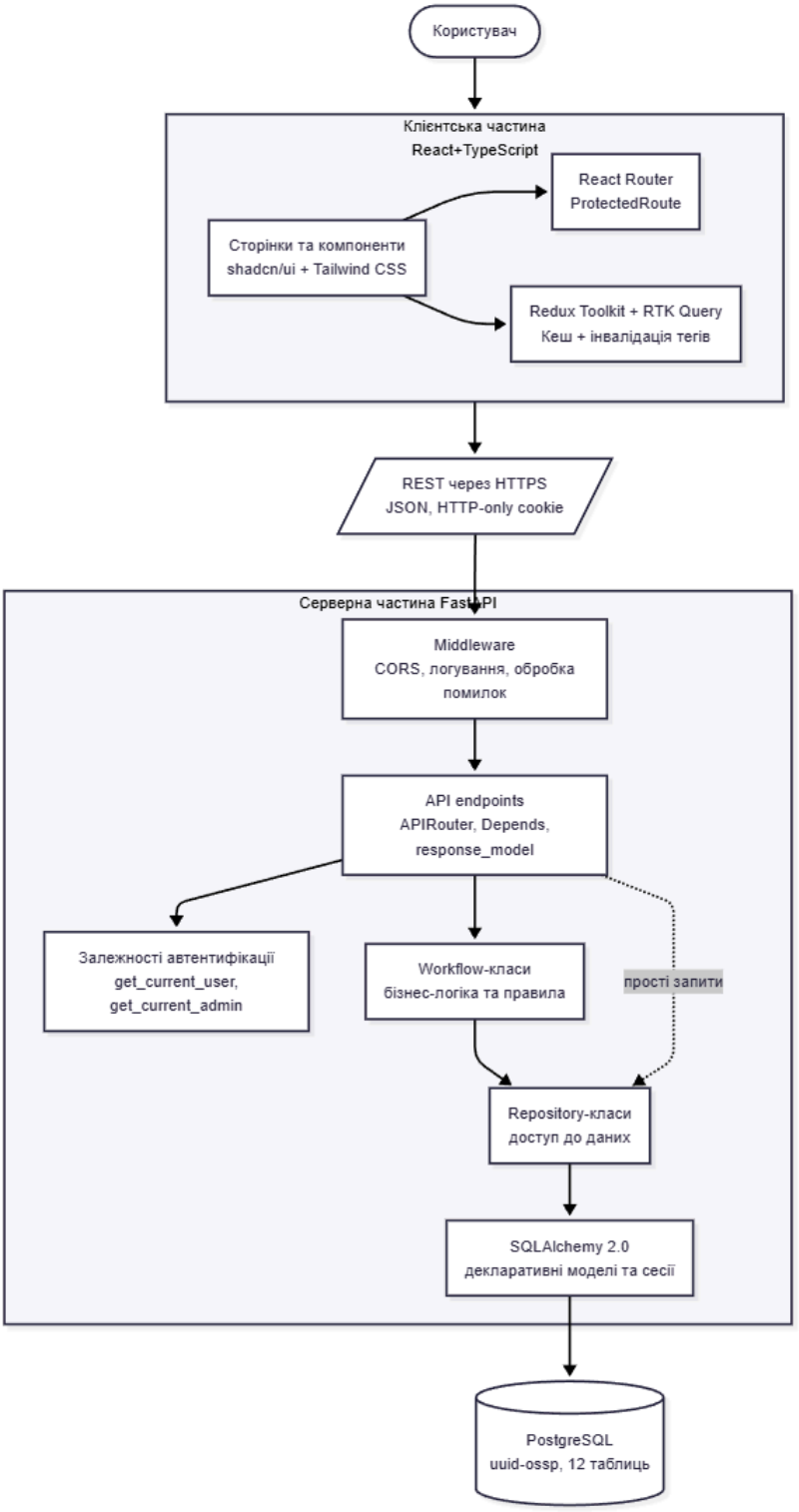

Рис. 1.4.1. Діаграма сценаріїв використання

Джерело: розроблено автором

Діаграма виділяє шість логічних областей сценаріїв:

- Автентифікація — реєстрація, вхід та вихід; доступна гостям та зареєстрованим користувачам.
- Управління групами — створення груп, запрошення учасників, прийняття запрошень, налаштування дозволів, видалення учасників.
- Ресторани та страви — створення каталогу ресторанів у межах групи, додавання страв з цінами, позначення улюблених позицій.
- Замовлення — ініціювання, додавання позицій, підтвердження, вказання вартості доставки, позначення як замовленого, завершення (із автоматичним перерахунком балансів) та скасування.
- Баланси — перегляд балансів учасників, ручне коригування, перегляд історії змін.
- Адміністрування — управління всіма користувачами системи (тільки для актора «Системний адміністратор»).

Кожен прецедент описано через короткий сценарій типу «актор — дія — очікуваний результат», що в подальшому стає основою для функціональних вимог.

1.5 Функціональні вимоги до системи

Результатом аналізу предметної області, персон та сценаріїв використання став формальний перелік функціональних вимог, структурований за дев'ятьма модулями (епіками). Кожна вимога ідентифікована кодом виду FR-X.Y для подальшого використання у матриці трасування (див. розділ 6.6).

Таблиця 1.5.1

Узагальнення функціональних вимог за модулями

Модуль	Коди вимог	Ключовий зміст
Автентифікація	FR-1.1-FR-1.4	Реєстрація, вхід, поточний користувач, вихід

Модуль	Коди вимог	Ключовий зміст
Дозволи	FR-2.1-FR-2.4	Глобальні ролі, групові дозволи, рівні та пресети
Користувачі	FR-3.1-FR-3.3	Адміністрування користувачів і власний профіль
Групи	FR-4.1-FR-4.7	Створення, учасники, ліміти, email-запрошення
Ресторани	FR-5.1-FR-5.3	Ресторани, страви, улюблені позиції
Баланси	FR-6.1-FR-6.3	Поточні баланси, ручні коригування, історія
Замовлення	FR-7.1-FR-7.8	Життєвий цикл, позиції, доставка, завершення
Дашборд групи	FR-8.1-FR-8.3	Аналітика групи, активне замовлення, баланс
Дашборд користувача	FR-9.1-FR-9.3	Особиста аналітика, налаштування та навігація

Деталізований перелік з 37 вимог використовується у матриці трасування розділу 6.6; у цьому розділі залишено лише рівень, потрібний для постановки задачі.

1.5.1 Постановка задачі за ролями користувачів

Той самий набір функціональних вимог з різних точок зору виглядає по-різному. Для прозорого окреслення меж відповідальності кожної ролі нижче згруповано задачі, які має розв'язувати система з позиції конкретного актора.

Власник групи (Group Owner — пресет admin). Ця роль є точкою входу для робочого колективу у систему. Завдання:

- створення нової групи й налаштування її назви, опису та логотипу;

- запрошення учасників за корпоративним email; видалення тих, хто покинув колектив;
- призначення індивідуальних дозволів (з пресету або точково по п'яти типах);
- ініціювання, ведення та завершення замовлення; ручне коригування балансів;
- доступ до аналітики групи; видалення групи у разі завершення її використання.

Заступник власника (Supervisor Member — пресет supervisor_member). Ця роль розрахована на делегування повсякденних дій без передачі повного контролю. Завдання:

- ініціювання замовлень за відсутності власника;
- перегляд учасників, балансів та аналітики групи без права редагування;
- перегляд каталогу ресторанів і страв.

Звичайний учасник (Member — пресет member). Найчисельніша роль. Завдання:

- додавання, редагування та видалення лише власних позицій у поточному замовленні;
- позначення улюблених страв для швидкого додавання;
- перегляд каталогу ресторанів (читання); перегляд активного замовлення.

Системний адміністратор (System Admin — роль рівня застосунку admin). Не належить жодній групі, але має повний доступ до всіх ресурсів системи. Завдання:

- модерація користувачів (зміна ролі рівня застосунку, активація/деактивація);
- технічна підтримка та відновлення даних з резервних копій;
- доступ до будь-якої групи без проходження стандартних перевірок дозволів.

Таке групування підтверджує, що кожна функціональна вимога з таблиці 1.5.1 закріплена за щонайменше однією роллю-виконавцем, а доступні дії для кожної ролі обмежені дворівневою системою дозволів, описаною у розділі 2.5.

1.6 Нефункціональні вимоги та обмеження

До нефункціональних вимог системи належать характеристики, що визначають якість, надійність та обмеження реалізації.

Безпека. Автентифікація реалізована через JWT-токени, що зберігаються в HTTP-only cookies для захисту від XSS-атак. Паролі зберігаються виключно у вигляді bcrypt-хешів. Усі вхідні дані валідуються на рівні API через Pydantic-схеми [8]. Доступ до даних інших груп обмежений на рівні API — користувач не може отримати ресурси груп, у яких не є учасником (окрім системного адміністратора). У промисловому режимі додаток доступний лише за HTTPS з TLS-сертифікатом Let's Encrypt [11], а пряма передача даних на бекенд (порт 8000) обмежена внутрішнім мережевим інтерфейсом.

Продуктивність. Серверна частина повністю асинхронна (FastAPI на ASGI-сервері Uvicorn [12], SQLAlchemy 2.0 у режимі async [4]), що дозволяє обробляти багато одночасних запитів на стандартній VPS-конфігурації. У продакшені передбачено 4 робочих процеси uvicorn (--workers 4), кеш статичних ресурсів у Nginx [16] та з'єднання proxy_http_version 1.1 для keep-alive.

Підтримуваність та якість коду. Бекенд проходить статичну перевірку через ruff check та ruff format у CI-конвеєрі. Фронтенд проходить перевірку типів TypeScript через tsc --noEmit та збирається у CI. Кодова база організована за чіткою шаровою архітектурою із розділенням відповідальності між API, Workflow та Repository.

Зручність використання. Інтерфейс є адаптивним та працює як на десктопних, так і на мобільних пристроях. Усі форми мають інтегровану валідацію через react-hook-form + zod, повідомлення про помилки виводяться поряд із полями, а тост-сповіщення (бібліотека sonner) інформують користувача про результат дій.

Обмеження масштабу. Максимум 5 груп на одного користувача; максимум 25 учасників у одній групі; максимальний розмір завантаженого файлу — 10 МБ. Ці обмеження зафіксовано як константи у коді (MAX_GROUPS_PER_USER, MAX_UPLOAD_SIZE).

Технологічні обмеження. PostgreSQL [5] як СУБД (через використання типу UUID та розширення uuid-ossp), Python 3.12+ для серверної частини (через сучасний синтаксис типів), Node.js 20+ для фронтенду (вимога Vite 7 [13]). Валюта обліку — українська гривня (₴), без підтримки мультивалютності.

Інфраструктурні вимоги. Підтримується розгортання на виділеному сервері Ubuntu 22.04 LTS [19] з мінімум 2 ядрами CPU, 4 ГБ ОЗП та 40 ГБ диска. Розгортання повністю автоматизовано через скрипти infrastructure/setup.sh (одноразове налаштування сервера) та infrastructure/deploy.sh (повторні оновлення). Резервне копіювання бази даних виконується щодня з ретенцією 30 днів.

Регуляторний контекст обробки персональних даних. Система обробляє обмежений обсяг персональних даних користувачів — email, повне ім'я, хеш пароля, фінансові операції у межах групи. Цей перелік сформовано з дотриманням принципу мінімізації даних Закону України «Про захист персональних даних» № 2297-VI [31]: жодних додаткових ідентифікаторів (номер телефону, дата народження, адреса) не збирається. Користувач надає згоду на обробку під час реєстрації шляхом активної дії (створення облікового запису), а пароль зберігається лише як bcrypt-хеш і не доступний у відкритому вигляді навіть адміністратору. Доступ до власних даних користувач реалізує через сторінку профілю; деактивація облікового запису припиняє можливість використання системи без видалення історичних записів про вже завершені замовлення. У межах роботи передано лише ті обсяги обробки, що необхідні для функціонування корпоративного внутрішнього інструменту, без передачі даних до третіх сторін.

1.7 Висновки до розділу 1

У розділі визначено проблему автоматизації спільних обідів, порівняно основні класи наявних рішень і показано, що вони окремо покривають лише комунікацію, доставку або облік витрат. Сформовано цільові ролі користувачів, ключові сценарії, 37 функціональних вимог у дев'яти модулях та набір нефункціональних вимог щодо безпеки, продуктивності, підтримуваності, інфраструктури й обробки персональних даних. Ці результати є основою для архітектурного проєктування у наступному розділі.

РОЗДІЛ 2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ

2.1 Загальна архітектура та принципи побудови

Система «LunchTogether» побудована за класичною клієнт-серверною архітектурою з чітким розділенням відповідальності між шарами. Клієнтська частина (React SPA [2]) взаємодіє з серверною частиною (FastAPI REST API [1]) через HTTP-запити у форматі JSON. Серверна частина організована шарово: ендпоінти API делегують виконання бізнес-логіки до Workflow-класів, які, у свою чергу, отримують доступ до даних через шар репозиторіїв над SQLAlchemy.

Ключові архітектурні рішення:

- Розділення відповідальності за шарами. Кожен шар має чітко визначений обов'язок: API-шар приймає HTTP-запити, валідує вхідні дані та формує відповіді; Workflow-шар інкапсулює бізнес-правила та оркестрацію кількох операцій; Repository-шар відповідає виключно за доступ до даних і не містить бізнес-логіки; Models-шар описує структуру даних та зв'язки.
- Інверсія залежностей через DI. Усі залежності (репозиторії, workflow, сесії БД, сервіси) ін'єктуються через механізм Depends() фреймворку FastAPI, що спрощує тестування та усуває жорсткі прив'язки між шарами.
- Асинхронність на всіх рівнях. Запити до бази даних, HTTP-обробники, надсилання електронних листів та робота з файлами реалізовано через async/await, що дозволяє ефективно використовувати ресурси сервера.
- Монорепозиторій з двома основними додатками. Серверна та клієнтська частини розташовано в одному репозиторії у директоріях backend/ та frontend/ відповідно, що спрощує спільний CI, тестування контрактів та супровід.

2.1.1 Аналітика інформаційних потоків системи

Для повноти проєктування виокремлено вхідні й вихідні потоки даних та межі системи. Це дозволяє чітко зафіксувати, які зовнішні актори впливають на систему та які зовнішні сервіси є невід'ємною частиною її роботи.

Вхідні потоки (Inputs).

- HTTP-запити користувачів. Усі взаємодії з фронтенду та сторонніх клієнтів виконуються через REST API за префіксом /api. Запити обов'язково містять валідовану Pydantic-схему у тілі (для POST/PATCH/PUT) та JWT-токен у HTTP-only cookie для захищених маршрутів.
- Завантаження файлів. Бінарні дані (логотипи груп) надходять як multipart/form-data з обмеженням 10 МБ. Зберігаються у директорії UPLOAD_DIR через модуль core/storage.py.
- Електронна пошта запрошень. Користувач отримує лист із токеном; перехід за посиланням повертає його у систему через окремий API-маршрут прийняття/відхилення.
- ACME-челенджі Let's Encrypt. HTTP-блок Nginx обслуговує /.well-known/acme-challenge/ для автоматичного отримання й оновлення TLS-сертифіката.

Вихідні потоки (Outputs).

- JSON-відповіді API. Усі ендпоінти повертають структуровані JSON-документи за схемами *Response, автоматично документованими в OpenAPI.
- Статичні файли. Nginx віддає SPA-збірку з frontend/dist/ та завантажені файли з /uploads/ з агресивним кешуванням.
- SMTP-листи. Запрошення нових учасників надсилаються через EmailService (SMTP-клієнт із пакета aiosmtplib).

- Події Sentry [25]. Усі необроблені виключення асинхронно відсилаються до зовнішньої служби моніторингу разом із контекстом запиту.
- Системні журнали. journald та logrotate зберігають журнали systemd-сервісу та Nginx у файловій системі сервера.

Межі системи. Зовнішніми сутностями стосовно ядра застосунку є: користувачі (через браузер), сервер електронної пошти (SMTP-провайдер), центр сертифікації Let's Encrypt, служба моніторингу Sentry та реляційна СУБД PostgreSQL [5]. Внутрішніми компонентами є FastAPI-застосунок, Nginx-проксі, systemd-сервіс, cron-завдання резервного копіювання та підсистема файлового сховища.

2.2 Вибір технологічного стеку

Технологічний стек було обрано з огляду на сучасність, типобезпеку, продуктивність та зрілість екосистеми. Таблиця 2.2.1 узагальнює зроблений вибір і його обґрунтування.

Таблиця 2.2.1

Технологічний стек системи

Шар	Технологія	Обґрунтування вибору
Мова бекенду	Python 3.12	Сучасна підтримка типів (X | None, list[X]), широка стандартна бібліотека, потужні фреймворки. Альтернативи: Node.js (TypeScript), Go.
Фреймворк бекенду	FastAPI [1]	Асинхронний фреймворк із автоматичною генерацією OpenAPI, валідацією через Pydantic, нативною підтримкою DI. Перевершує Flask за швидкістю розробки REST API і Django REST Framework за продуктивністю та типобезпекою.

Шар	Технологія	Обґрунтування вибору
ORM	SQLAlchemy 2.0 [4]	Найзріліший ORM для Python із повноцінною підтримкою async. Декларативні моделі з Mapped/mapped_column забезпечують типобезпеку. Альтернативи: Tortoise ORM (молодша спільнота), SQLModel (тонкий шар над SQLAlchemy, але без переваг для зрілого проєкту).
Міграції	Alembic [7]	Стандартний інструмент міграцій для SQLAlchemy із підтримкою autogenerate.
СУБД	PostgreSQL 16 [5]	Зріла реляційна СУБД з потужною підтримкою типів (UUID через uuid-ossp, Numeric для грошей), транзакцій та індексів. Альтернативи: MySQL/MariaDB (слабша підтримка UUID), SQLite (не для продакшну).
Драйвер БД	asyncpg	Найшвидший асинхронний драйвер для PostgreSQL.
Валідація	Pydantic v2 [8]	Інтегрована з FastAPI, забезпечує автоматичну валідацію та серіалізацію за моделями.
Автентифікація	JWT + bcrypt	Стандартний підхід: bcrypt для хешування паролів, JWT для бездержавних токенів. Зберігаються у HTTP-only cookies для захисту від XSS.
Менеджер пакетів	uv [22]	Сучасний інструмент від Astral, на порядки швидший за pip/poetry, із автоматичним керуванням віртуальним середовищем. Використовується як локально, так і у продакшені.
ASGI-сервер	Uvicorn [12]	Стандартний ASGI-сервер для FastAPI; у промисловій конфігурації запускається із 4 робочими процесами.

Шар	Технологія	Обґрунтування вибору
Лінтер/форматер	Ruff	Швидкий лінтер та форматер для Python, що замінює одразу декілька інструментів (flake8, isort, pyupgrade).
Мова фронтенду	TypeScript 5	Типобезпечна надбудова над JavaScript, що значно знижує кількість runtime-помилок та полегшує рефакторинг.
Бібліотека UI	React 19 [2]	Найпопулярніша бібліотека для побудови SPA, із зрілою екосистемою та підтримкою функціональних компонентів і хуків.
Збирач	Vite 7 [13]	Швидкий збирач із підтримкою HMR (hot module replacement), нативним TypeScript та оптимізованим продакшен-збиранням через esbuild і Rollup.
Управління станом	Redux Toolkit + RTK Query [6]	Офіційний інструмент для Redux із суттєво меншим обсягом шаблонного коду. RTK Query автоматизує кешування серверних даних, інвалідацію за тегами та повторне завантаження.
Маршрутизація	React Router v7 [10]	Стандартна бібліотека маршрутизації для React, із підтримкою захищених маршрутів та createBrowserRouter.
Дизайн-система	shadcn/ui [3] + Radix UI	Колекція доступних, копійованих у проєкт UI-компонентів на базі Radix UI. На відміну від MUI чи Ant Design, компоненти не закриті у бібліотеці, а живуть у коді проєкту, що дозволяє вільну кастомізацію.
Стилізація	Tailwind CSS v4 [9]	Утилітарна CSS-фреймворк, що добре поєднується з shadcn/ui. У версії 4 використовується нативна JIT-компіляція без PostCSS.

Шар	Технологія	Обґрунтування вибору
Форми	react-hook-form + zod	react-hook-form дає продуктивні неконтрольовані форми, zod — типобезпечну валідацію.
Іконки	lucide-react	Зріла бібліотека SVG-іконок із більш ніж 1500 елементів, оптимізована для tree-shaking.

2.3 Проєктування реляційної бази даних

База даних PostgreSQL містить 12 основних таблиць, пов'язаних зовнішніми ключами. Усі таблиці використовують UUID як первинний ключ (для уникнення передбачуваності ідентифікаторів та для незалежності від послідовностей при злитті даних з кількох екземплярів) та мають поля created_at і updated_at для аудиту. Міграції керуються за допомогою Alembic [7]. Таблиця 2.3.1 описує призначення кожної сутності.

Таблиця 2.3.1

Сутності реляційної моделі

Таблиця	Призначення	Ключові поля та особливості
users	Користувачі системи	email (унікальний, індексований), hashed_password, full_name, role, is_active, is_verified
groups	Групи для спільних обідів	name, description, logo_path, owner_id (FK на користувача-власника)
group_members	Учасники груп (зв'язкова таблиця користувач × група)	(user_id, group_id) — унікальна пара (UQ uq_group_member_user_group)
group_member_permissions	Дозволи учасників	permission_type, level; UQ на (group_member_id, permission_type)

Таблиця	Призначення	Ключові поля та особливості
group_invitations	Запрошення до груп	invitee_email, token (унікальний), status, invitee_id (опціонально, якщо вже зареєстрований)
restaurants	Ресторани в межах групи	name, description, group_id
dishes	Страви ресторану	name, detail, price (Numeric(10,2)), restaurant_id
favorite_dishes	Улюблені страви користувача	UQ на (user_id, dish_id), is_favorite
orders	Замовлення	group_id, restaurant_id (SET NULL), restaurant_name (текстовий запасний варіант), initiator_id, status, delivery_fee_total, delivery_fee_per_person
order_items	Позиції замовлення	order_id, user_id, dish_id (опц.), name, detail, price, quantity (за замовчуванням 1)
balances	Баланси користувачів у групах	UQ на (user_id, group_id), amount (Numeric(10,2), за замовчуванням 0)
balance_history	Записи історії змін балансу	amount (підписана сума зміни), balance_after (новий стан), note, change_type (manual/order), order_id (опц.), created_by_id

Стратегії видалення. Більшість зовнішніх ключів використовують ON DELETE CASCADE (наприклад, dishes.restaurant_id → restaurants.id), що забезпечує цілісність при видаленні агрегатних коренів. Винятки: orders.restaurant_id та balance_history.order_id використовують ON DELETE SET NULL, що зберігає історію навіть після видалення посилань. restaurant_name дублює

назву у момент створення, тож історія залишається повною навіть якщо ресторан було видалено.

Грошові типи. Усі поля, що зберігають грошові суми, використовують Numeric(10, 2) для уникнення похибок з плаваючою комою. Десяткова арифметика (decimal.Decimal) використовується і в Python-коді, зокрема в OrderLifecycleWorkflow._handle_finish() для розрахунку часток вартості доставки.

Обмеження даних для ключових сутностей. Таблиця 2.3.2 фіксує конкретні обмеження довжин, форматів і числових діапазонів, синхронізовані між Pydantic-схемами та SQLAlchemy-моделями. Ці обмеження діють як на рівні валідації запитів, так і на рівні структури БД.

Таблиця 2.3.2

Обмеження даних для ключових сутностей

Поле	Тип	Обмеження
users.email	String(255)	Обов'язкове, унікальне, валідація формату через EmailStr Pydantic [8]
users.hashed_password	String(255)	Bcrypt-хеш ($2b$-формат, довжина 60 символів)
users.full_name	String(100)	Обов'язкове, 2–100 символів
groups.name	String(100)	Обов'язкове, 2–100 символів
groups.description	String(500)	Необов'язкове, до 500 символів
groups.logo_path	String(255)	Заповнюється файлом ≤ MAX_UPLOAD_SIZE = 10 МБ (10 485 760 байт)
group_invitations.invitee_email	String(255)	Валідація формату; пара (email, status=PENDING) — унікальна
group_invitations.token	String(64)	Унікальний, генерується через secrets.token_urlsafe(32)

Поле	Тип	Обмеження
restaurants.name	String(100)	Обов'язкове, 1–100 символів
dishes.name	String(100)	Обов'язкове, 1–100 символів
dishes.price	Numeric(10, 2)	≥ 0; до 99 999 999.99
order_items.quantity	Integer	≥ 1, за замовчуванням 1
order_items.price	Numeric(10, 2)	≥ 0; зберігається як знімок ціни на момент додавання позиції
orders.delivery_fee_total	Numeric(10, 2)	≥ 0; перераховується відповідно до delivery_fee_per_person × n_unique_users
balances.amount	Numeric(10, 2)	За замовчуванням 0; підтримує від'ємні значення
balance_history.amount	Numeric(10, 2)	Підписана зміна; запис створюється атомарно з оновленням Balance.amount
Ліміти кількості	константи	MAX_GROUPS_PER_USER = 5, MAX_MEMBERS_PER_GROUP = 25, MAX_UPLOAD_SIZE = 10 МБ

2.4 Проєктування REST API

REST API побудовано на основі FastAPI [1] з використанням APIRouter для модульної організації ендпоінтів. Усі ендпоінти повертають JSON-відповіді на основі Pydantic-схем [8], що автоматично генерують специфікацію OpenAPI (доступну у режимі розробки за адресою /api/docs через Swagger UI та /api/redoc через ReDoc). Усі маршрути організовано під єдиним префіксом /api. Таблиця 2.4.1 наводить ключові ендпоінти у згрупованому вигляді.

Таблиця 2.4.1

Основні ендпоінти REST API

Метод	Шлях	Призначення
GET	/api/health	Перевірка справності бекенду (health check)
POST	/api/auth/register	Реєстрація користувача
POST	/api/auth/login	Вхід у систему із встановленням HTTP-only cookie
POST	/api/auth/logout	Вихід із системи із очищенням cookie
GET	/api/auth/me	Інформація про поточного користувача
GET	/api/users	Список усіх користувачів (тільки Admin)
PATCH	/api/users/{user_id}	Зміна ролі/активності користувача (Admin) або профілю
GET	/api/groups	Список груп користувача (Admin бачить усі)
POST	/api/groups	Створення групи
GET	/api/groups/{group_id}	Деталі групи
PATCH	/api/groups/{group_id}	Оновлення групи
DELETE	/api/groups/{group_id}	Видалення групи
GET	/api/groups/{group_id}/members	Список учасників
POST	/api/groups/{group_id}/members	Додавання учасника напряму
DELETE	/api/groups/{group_id}/members/{user_id}	Видалення учасника
PATCH	/api/groups/{group_id}/members/{user_id}/permissions	Зміна дозволів учасника
POST	/api/groups/{group_id}/invitations	Створення запрошення за email

Метод	Шлях	Призначення
POST	/api/groups/invitations/{token}/accept	Прийняття запрошення
POST	/api/groups/invitations/{token}/decline	Відхилення запрошення
GET	/api/groups/invitations/mine	Список моїх вхідних запрошень
GET	/api/groups/invitations/by-token/{token}	Попередній перегляд запрошення (публічний)
GET	/api/groups/{group_id}/invitations	Список вихідних запрошень групи
DELETE	/api/groups/{group_id}/invitations/{invitation_id}	Скасування вихідного запрошення
GET	/api/groups/{group_id}/restaurants	Список ресторанів групи
POST	/api/groups/{group_id}/restaurants	Створення ресторану
GET	/api/groups/{group_id}/restaurants/{restaurant_id}	Деталі ресторану зі стравами
POST	/api/groups/{group_id}/restaurants/{rid}/dishes	Додавання страви
GET	/api/groups/{group_id}/orders	Список замовлень групи
POST	/api/groups/{group_id}/orders	Створення замовлення
GET	/api/groups/{group_id}/orders/active	Активне замовлення групи
GET	/api/groups/{group_id}/orders/{order_id}	Деталі замовлення зі списком позицій
POST	/api/groups/{group_id}/orders/{order_id}/status	Перехід у новий стан життєвого циклу
POST	/api/groups/{group_id}/orders/{order_id}/delivery-fee	Установка вартості доставки
GET	/api/groups/{group_id}/orders/{order_id}/items	Список позицій

Метод	Шлях	Призначення
POST	/api/groups/{group_id}/orders/{order_id}/items	Додавання позиції
PATCH	/api/groups/{group_id}/orders/{order_id}/items/{iid}	Редагування позиції
DELETE	/api/groups/{group_id}/orders/{order_id}/items/{iid}	Видалення позиції
POST	/api/groups/{group_id}/orders/favorites/{dish_id}	Перемикання улюбленої страви
GET	/api/groups/{group_id}/balances	Баланси учасників
POST	/api/groups/{group_id}/balances/adjust	Ручне коригування балансу
GET	/api/groups/{group_id}/balances/{user_id}/history	Історія змін балансу
GET	/api/groups/{group_id}/analytics	Аналітика групи
GET	/api/users/me/analytics	Аналітика поточного користувача

Принципи проєктування ендпоінтів:

- Шляхи для ресурсів, що належать групі, мають префікс /groups/{group_id}/..., що чітко відображає ієрархію власності.
- Методи HTTP використовуються відповідно до семантики REST: GET для читання, POST для створення та виконання дій, PATCH для часткового оновлення, DELETE для видалення. Зміна стану замовлення вирішена через POST /status (а не PATCH), оскільки це бізнес-перехід, а не довільна зміна поля.
- Усі захищені ендпоінти вимагають коректного JWT у cookie. У разі відсутності або некоректності токена API повертає HTTP 401 з полем detail: "Not authenticated".

- Помилки повертаються як JSON-об'єкти із полем detail за стандартом FastAPI; для типів помилок використовується чітка ієрархія HTTP-кодів (400 для валідації, 401 для автентифікації, 403 для дозволів, 404 для відсутніх ресурсів, 409 для конфліктів, 422 для семантичних помилок).

2.5 Дворівнева система прав та дозволів

Система прав реалізована на двох рівнях, що поєднують жорсткий глобальний контроль та гнучке делегування у межах груп.

Рівень 1 — рівень додатку. Кожен користувач має одну з двох ролей: Admin або User. Адміністратор має повний доступ до всіх ресурсів системи (усі групи, усі користувачі, усі замовлення), обходить більшість перевірок дозволів і використовується для обслуговування та модерації. Звичайний користувач (User) має доступ лише до тих груп, у яких він є учасником.

Рівень 2 — рівень групи. Для кожного учасника групи (GroupMember) у таблиці group_member_permissions зберігаються індивідуальні дозволи. П'ять типів дозволів охоплюють основні функціональні області:

Таблиця 2.5.1

Типи дозволів і доступні рівні

Тип дозволу	Доступні рівні	Семантика рівнів
members	Editor / Viewer / None	Editor — повне керування учасниками (запрошення, видалення, зміна дозволів); Viewer — перегляд; None — приховано
orders	Editor / Initiator / Participant	Editor — модифікація будь-яких замовлень; Initiator — створення власних замовлень; Participant — додавання лише своїх позицій

Тип дозволу	Доступні рівні	Семантика рівнів
balances	Editor / Viewer / None	Editor — ручне коригування балансів; Viewer — перегляд; None — приховано
analytics	Viewer / None	Viewer — перегляд аналітики групи; None — приховано
restaurants	Editor / Viewer	Editor — CRUD ресторанів і страв; Viewer — лише перегляд

Пресети ролей. Для зручності призначення дозволів реалізовано три пресети у структурі GROUP_ROLE_PRESETS, що одночасно встановлюють рівні всіх п'яти типів дозволів. Це уникає ручного виставлення 5 значень для кожного учасника та забезпечує консистентність призначень.

Таблиця 2.5.2

Пресети ролей у групі

Пресет	Members	Orders	Balances	Analytics	Restaurants
admin	Editor	Editor	Editor	Viewer	Editor
supervisor_ member	Viewer	Initiator	Viewer	Viewer	Viewer
member	None	Participant	None	None	Viewer

Алгоритм перевірки дозволу. При обробці запиту, що потребує дозволу, виконується така послідовність:

1. Чи є користувач системним адміністратором (current_user.is_admin)? Якщо так — дозвіл надається без подальших перевірок.
2. Чи є користувач учасником цільової групи? Запит GroupMemberRepository.get_membership(user_id, group_id). Якщо ні — ForbiddenError.

3. Чи має учасник потрібний тип дозволу з достатнім рівнем? Метод GroupMember.get_permission(permission_type) повертає рівень, який порівнюється з необхідним. Якщо рівень нижчий — ForbiddenError.

Спеціальні випадки: для замовлень додатково перевіряється, чи є користувач ініціатором конкретного замовлення (тоді він може керувати ним навіть з рівнем Participant). Власник групи захищений від модифікації іншими учасниками — навіть Editor не може видалити власника або змінити його дозволи.

2.6 Архітектурні шаблони: Repository, Workflow, Dependency Injection

Серверна частина побудована з використанням трьох взаємодоповнюючих архітектурних шаблонів, що забезпечують чітке розділення відповідальності та полегшують тестування.

Repository Pattern. Шаблон Repository інкапсулює доступ до бази даних, виокремлюючи його у окремий шар. Узагальнений базовий клас BaseRepository[ModelType] надає стандартний набір CRUD-операцій (get_by_id, get, get_multi, create, update, delete), які успадковуються конкретними репозиторіями (UserRepository, GroupRepository, OrderRepository тощо). Конкретні репозиторії додають специфічні методи запитів, але не містять бізнес-логіки — лише операції з даними. Це дозволяє:

- замінювати джерело даних (наприклад, для тестів) без зміни бізнес-логіки;
- легко знаходити всі місця, де відбувається доступ до конкретної таблиці;
- уніфікувати такі деталі, як пагінація та фільтрація.

Workflow Pattern. Workflow-класи інкапсулюють бізнес-логіку. Кожен workflow відповідає за один бізнес-сценарій (наприклад, CreateGroupWorkflow, OrderLifecycleWorkflow, AdjustBalanceWorkflow, InviteWorkflow) і складається з:

- Pydantic-моделей {Name}Input та {Name}Output, що задають типобезпечні вхідні дані та результат;

- класу {Name}Workflow з конструктором, який приймає необхідні репозиторії, та одним або кількома методами (execute, transition тощо);
- усіх бізнес-перевірок (ліміти, дозволи, валідність переходів) усередині методів workflow.

Завдяки цьому шару бізнес-правила зосереджені в одному місці, легко переглядаються рецензентом коду та можуть бути перевикористані з різних точок входу (наприклад, з REST API та з командного скрипту). Workflow-класи зберігають стійкість до зміни деталей реалізації API чи бази даних.

Dependency Injection. Усі залежності — від сесії бази даних до конкретних workflow — ін'єктуються через механізм Depends() фреймворку FastAPI. Фабричні функції зосереджено у єдиному модулі backend/app/dependencies.py, де описано три рівні фабрик:

1. Фабрики репозиторіїв — отримують AsyncSession через Depends(get_db) і повертають конкретний репозиторій.
2. Фабрики сервісів — повертають об'єкти без залежностей від БД (наприклад, EmailService).
3. Фабрики Workflow — приймають уже сконструйовані репозиторії та сервіси і повертають готовий до використання workflow.

Ендпоінти просто декларують свої залежності у сигнатурі функції, не дбаючи про їх створення. Це усуває жорсткі прив'язки, спрощує мокування для тестів та робить структуру залежностей видимою.

2.7 Алгоритмічне забезпечення системи

Серед бізнес-логіки системи є кілька алгоритмів, що мають нетривіальну структуру та визначають коректність ключових сценаріїв. Нижче формалізовано три з них; їхні детальні реалізації наведено у вихідному коді репозиторію та у Додатку Б.

2.7.1 Алгоритм перевірки дворівневих дозволів

Призначення. Гарантувати, що користувач має право виконати дію над ресурсом групи відповідного типу та з достатнім рівнем дозволу.

Вхід: current_user: User, group_id: UUID, required_permission: PermissionType, required_level: PermissionLevel.

Вихід: True — дозвіл наданий; інакше — виключення ForbiddenError.

Кроки:

1. Якщо current_user.is_admin == True — повернути True без подальших перевірок (системний адміністратор обходить групові дозволи).
2. Звернутися до group_member_repository.get_membership(current_user.id, group_id). Якщо результат None — повернути ForbiddenError("Not a member of this group").
3. Отримати рівень дозволу через membership.get_permission(required_permission). Якщо результат — None (тобто дозвіл взагалі не призначений) — повернути ForbiddenError.
4. Порівняти отриманий рівень із required_level за внутрішньою ієрархією (Editor > Initiator > Viewer > Participant > None). Якщо рівень нижчий за необхідний — повернути ForbiddenError.
5. Спеціальний випадок для замовлень: якщо required_permission == Orders та current_user.id == order.initiator_id, то рівень підвищується до Initiator навіть за призначеного Participant.
6. Окремо забезпечується захист власника групи: будь-яка спроба змінити дозволи або видалити користувача із роллю Group Owner відхиляється з ForbiddenError, окрім випадку, коли запит виконується самим власником або системним адміністратором.

Алгоритм реалізовано у workflow-класах та обробниках ендпоінтів, що використовують виклики _check_permission(...); конкретна точка перевірки залежить від типу дії (детальніше — у розділі 3.7).

2.7.2 Алгоритм валідації переходу станів замовлення

Призначення. Не допустити переведення замовлення у недопустимий стан життєвого циклу.

Вхід: current_status: OrderStatus, target_status: OrderStatus, current_user: User, order: Order.

Вихід: True — перехід валідний; інакше — ValidationError (HTTP 422) або ForbiddenError (HTTP 403).

Кроки:

1. Перевірити дозвіл на дію: користувач повинен бути ініціатором замовлення, мати рівень Orders:Editor або бути системним адміністратором. Інакше — ForbiddenError.
2. Звернутися до константи VALID_TRANSITIONS: dict[OrderStatus, list[OrderStatus]]:
   - INITIATED → {CONFIRMED, CANCELLED}
   - CONFIRMED → {ORDERED, CANCELLED}
   - ORDERED → {FINISHED, CANCELLED}
   - FINISHED → {}, CANCELLED → {} — термінальні стани.
3. Якщо target_status не входить до набору допустимих переходів із current_status — повернути ValidationError("Cannot transition from {current} to {target}").
4. Якщо target_status == FINISHED — викликати приватний метод _handle_finish(order) (див. наступний підрозділ) у межах поточної асинхронної сесії БД.

5. Оновити поле status об'єкта Order, виконати flush та повернути результат.

Така табличне формулювання робить набір переходів декларативним і легким для аудиту; додавання нового стану зводиться до зміни одного словника.

2.7.3 Алгоритм перерахунку балансів при завершенні замовлення

Призначення. Атомарно оновити баланси всіх учасників замовлення, зафіксувати журнал змін і синхронізувати каталог страв ресторану.

Вхід: order: Order у стані ORDERED, активна AsyncSession.

Вихід: Оновлені записи у таблицях balances, balance_history та (за наявності restaurant_id) dishes. У разі помилки — повний rollback транзакції.

Кроки:

1. Отримати всі позиції замовлення через order_item_repository.list_by_order(order.id).
2. Згрупувати позиції за user_id та обчислити суму Σ (price × quantity) для кожного учасника.
3. Якщо order.delivery_fee_per_person встановлено, додати його до суми кожного учасника.
4. Для кожного учасника:
   - отримати запис у balances для пари (user_id, group_id) або створити його (balance_repository.get_or_create);
   - зменшити balance.amount на обчислену суму;
   - створити запис у balance_history із полями amount = -total, balance_after = balance.amount, note = f"Order #{order.id.hex[:8]}", change_type = ORDER, order_id = order.id, created_by_id = current_user.id.
5. Якщо замовлення прив'язане до ресторану (order.restaurant_id is not None), для кожної позиції замовлення:

- якщо у каталозі ресторану існує страва із тією самою назвою — оновити її ціну на ціну позиції;
- інакше — створити нову страву з назвою, описом і ціною з позиції.

6. Усі зміни виконуються в межах однієї AsyncSession без явного commit: оскільки сесію керує DI-залежність get_db, її коміт або відкат відбувається на рівні обробника ендпоінта. Це забезпечує атомарність всього перерахунку: якщо хоча б один з кроків падає, ніяких часткових змін у БД не залишається.

2.8 Висновки до розділу 2

У розділі спроєктовано шарову клієнт-серверну архітектуру з поділом на React SPA, FastAPI REST API, Workflow, Repository та PostgreSQL. Обґрунтовано технологічний стек, реляційну модель з 12 сутностей, REST API за дев'ятьма функціональними областями та дворівневу систему прав. Окремо формалізовано три критичні алгоритми: перевірку дозволів, валідацію переходів станів замовлення та атомарний перерахунок балансів при завершенні замовлення.

РОЗДІЛ 3 РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ

3.1 Структура серверного додатку

Серверна частина розташована у директорії backend/. Уся прикладна логіка винесена в пакет backend/app/, структурований за функціональними шарами. На верхньому рівні пакета знаходяться технічні модулі, що не належать до жодного з функціональних шарів: точка входу main.py, конфігурація config.py, налаштування підключення до бази даних database.py та реєстр фабрик залежностей dependencies.py. Функціональні шари винесено у вкладені пакети.

Шарова організація гарантує, що верхні шари не імпортуються нижчими, що запобігає циклічним залежностям.

Таблиця 3.1.1

Пакети серверного додатку

Пакет/модуль	Призначення
app/main.py	Точка входу, фабрика create_app(), реєстрація middleware, ініціалізація Sentry та логування
app/config.py	Конфігурація через pydantic-settings з підтримкою .env
app/database.py	Налаштування AsyncEngine, фабрики сесій, dependency get_db()
app/dependencies.py	Усі DI-фабрики: репозиторіїв, сервісів, workflow та функцій автентифікації
app/models/	SQLAlchemy-моделі (user, group, restaurant, order, balance, спільний base, enums)
app/schemas/	Pydantic-схеми для запитів та відповідей API, згруповані за ресурсами
app/repositories/	Repository-класи, успадковані від BaseRepository[ModelType]
app/workflows/	Workflow-класи, згруповані за ресурсами: user/, group/, order/, balance/
app/api/	Ендпоінти REST API із поділом за ресурсами та центральним router.py
app/core/	Кросрізні утиліти: безпека (security.py), виключення (exceptions.py), middleware, email, storage

3.2 Конфігурація FastAPI та проміжне програмне забезпечення

Точка входу серверного додатку — модуль backend/app/main.py. Він виконує дві задачі: налаштовує глобальні аспекти середовища виконання (логування, інтеграцію Sentry) та оголошує фабричну функцію create_app(), що повертає сконфігурований екземпляр FastAPI.

Перед створенням додатку модуль ініціалізує конфігурацію логування з рівнем DEBUG у середовищі розробки та INFO у промисловому, та (за наявності

SENTRY_DSN у конфігурації) ініціалізує SDK Sentry [25] із прив'язкою до поточного середовища (development/production) та коефіцієнтом сампл-рейту трасування 1.0 у розробці і 0.1 у продакшені.

Фабрика create_app() створює екземпляр FastAPI з метаданими (назва, опис, версія), вмикає Swagger UI (/api/docs) та ReDoc (/api/redoc) лише у середовищі розробки і реєструє три шари проміжного програмного забезпечення (middleware), порядок яких має значення.

Таблиця 3.2.1

Middleware FastAPI-додатку

Middleware	Призначення	Порядок реєстрації
RequestLoggingMiddleware	Логує метод, шлях, статус-код та тривалість виконання кожного запиту	1 (внутрішнє)
ErrorHandlingMiddleware	Перехоплює необроблені виключення, логує їх із трасою стека та повторно піднімає	2 (середнє)
CORSMiddleware	Дозволяє CORS-запити з фронтенду (cors_origins зі змінних середовища), передає credentials: true	3 (зовнішнє)

У FastAPI середній стек middleware виконується у зворотному до реєстрації порядку: першим зареєстрований middleware стає найбільш внутрішнім, останній — найбільш зовнішнім. CORSMiddleware реєструється останнім, аби обробляти запити preflight OPTIONS ще до того, як вони потраплять до інших шарів. Реалізації власних middleware (RequestLoggingMiddleware та ErrorHandlingMiddleware) описано в backend/app/core/middleware.py.

Після реєстрації middleware фабрика підключає головний API-роутер (api_router з модуля backend/app/api/router.py) та оголошує health-check ендпоінт

GET /api/health, що повертає {"status": "ok"} і використовується балансувальниками та системами моніторингу.

Зведена таблиця версій і параметрів виконання. Таблиця 3.2.2 фіксує ключові версії компонентів та параметри процес-моделі серверної частини, узгоджені з реальними значеннями у скриптах розгортання та конфігурації systemd (див. розділ 5). Ця таблиця слугує орієнтиром для збігу версій між середовищем розробки та промисловим розгортанням.

Таблиця 3.2.2

Версії та параметри виконання серверної частини

Компонент / параметр	Значення	Місце фіксації
Python	3.12	infrastructure/setup.sh (крок 3), .github/workflows/ci.yml
FastAPI	0.115.x (мажорна — 0)	backend/pyproject.toml
Pydantic	v2 (2.7.x і новіше)	backend/pyproject.toml
SQLAlchemy	2.0.x	backend/pyproject.toml
Alembic	1.13.x	backend/pyproject.toml
asyncpg	0.29.x	backend/pyproject.toml
PostgreSQL	16	infrastructure/setup.sh (крок 5), infrastructure/scripts/backup-db.sh
Node.js	20	infrastructure/setup.sh (крок 4), .github/workflows/ci.yml
uv	актуальний від Astral [22]	infrastructure/setup.sh (кроки 8 і 10)
Uvicorn worker process	--workers 4, --host 127.0.0.1, --port 8000	infrastructure/systemd/lunchtogether-backend.service

Компонент / параметр	Значення	Місце фіксації
Внутрішня адреса API	127.0.0.1:8000	systemd-юніт; Nginx upstream backend
Зовнішня адреса	https://<DOMAIN> (443)	infrastructure/nginx/lunchtogether.conf
Час життя JWT	30 хвилин (JWT_ACCESS_TOKEN_EXPIRE_MINUTES=30)	infrastructure/setup.sh (крок 13), .env
Максимальний розмір файлу	10 МБ (MAX_UPLOAD_SIZE=10485760)	.env, Nginx client_max_body_size

3.3 Моделі даних SQLAlchemy

Моделі даних використовують декларативний стиль SQLAlchemy 2.0 [4] із типізованими атрибутами Mapped[T] та функцією mapped_column(). Усі моделі успадковуються від спільного абстрактного класу BaseModel (визначеного в backend/app/models/base.py), який забезпечує три універсальні поля для кожної таблиці: id (UUID, первинний ключ, генерується через uuid.uuid4), created_at та updated_at (TIMESTAMP WITH TIME ZONE, заповнюються базою через func.now(), поле updated_at додатково має onupdate). Це гарантує консистентну стратегію ідентифікації та аудиту для всіх сутностей.

Енумерації, що використовуються в моделях, винесено у backend/app/models/enums.py. У реляційній моделі вони зберігаються як String(...) із додатковими валідаціями на рівні застосунку. Це більш гнучкий підхід порівняно з вбудованими ENUM PostgreSQL, оскільки спрощує міграції при додаванні нових значень.

Кожна доменна модель оголошена в окремому модулі під app/models/. Таблиця 3.3.1 описує ключові моделі та їх характерні особливості.

Таблиця 3.3.1

Доменні моделі та їх характерні поля

Модель	Файл	Ключові поля та зв'язки
User	models/user.py	email (унікальний, індексований), hashed_password, full_name, role, is_active, is_verified, navigate_to_active_order; обчислювана властивість is_admin
Group	models/group.py	name, description, logo_path, owner_id; зв'язки до членів, запрошень, ресторанів, замовлень та балансів
GroupMember	models/group.py	Унікальна пара (user_id, group_id); колекція permissions із lazy="joined" для уникнення N+1; метод get_permission(type) повертає рівень дозволу
GroupMemberPermission	models/group.py	Унікальна пара (group_member_id, permission_type), поля permission_type, level
GroupInvitation	models/group.py	invitee_email, унікальний token, status, опціональний invitee_id
Restaurant	models/restaurant.py	name, description, group_id; зв'язок до страв
Dish	models/restaurant.py	name, detail, price (Numeric(10,2)), restaurant_id
Order	models/order.py	group_id, опціональний restaurant_id (SET NULL), restaurant_name (страховий текстовий запис), initiator_id, status, delivery_fee_total, delivery_fee_per_person
OrderItem	models/order.py	order_id, user_id, опціональний dish_id (SET NULL), name, detail, price, quantity

Модель	Файл	Ключові поля та зв'язки
FavoriteDish	models/order.py	Унікальна пара (user_id, dish_id), прапор is_favorite
Balance	models/balance.py	Унікальна пара (user_id, group_id), amount (Numeric(10,2)) із початковим значенням 0
BalanceHistory	models/balance.py	amount (підписана зміна), balance_after, note, change_type (manual/order), опціональні order_id та created_by_id

Реєстрація всіх моделей виконується через імпорт у файлі backend/app/models/__init__.py. Це гарантує, що Alembic при автогенерації міграцій бачитиме всі сутності.

3.4 Шаблон Repository для доступу до даних

Шар доступу до даних реалізовано через узагальнений базовий клас BaseRepository[ModelType], оголошений у backend/app/repositories/base.py. Завдяки Generic[ModelType] і параметризації типом моделі, успадковані репозиторії автоматично отримують типобезпечні CRUD-операції без дублювання коду. Конструктор приймає клас моделі та асинхронну сесію AsyncSession.

Таблиця 3.4.1

Методи BaseRepository

Метод	Призначення
_base_query()	Захищений метод, що повертає базовий SELECT * запит. Може бути перевизначений у нащадках для додавання JOIN-ів за замовчуванням
get_by_id(entity_id: uuid.UUID)	Повертає один запис за первинним ключем або None

Метод	Призначення
get(**filters)	Повертає один запис за довільними фільтрами (kwargs накладаються як WHERE)
get_multi(page, page_size, **filters)	Повертає пагінований список через PaginatedResponse (з полями items, total, page, page_size, total_pages)
create(data: dict)	Створює та повертає новий запис; виконує flush та refresh для отримання згенерованих значень
update(entity_id, data: dict)	Оновлює запис за ідентифікатором; ігнорує поля зі значенням None, що дозволяє реалізувати часткові оновлення
delete(entity_id)	Видаляє запис; повертає True/False залежно від наявності

Конкретні репозиторії додають специфічні методи запитів. Наприклад, UserRepository.get_by_email(email) повертає користувача за email; GroupRepository.count_by_owner(user_id) повертає кількість груп, що належать користувачу (використовується для контролю ліміту 5 груп); GroupMemberRepository.get_membership(user_id, group_id) повертає членство користувача у конкретній групі разом із завантаженими дозволами; OrderRepository.get_active_for_group(group_id) повертає поточне активне замовлення групи (тобто замовлення у нетермінальному стані).

Принципова заборона: репозиторії не повинні містити бізнес-логіку. Перевірка прав, ліміти, валідація переходів між станами — усе це належить шару Workflow. Репозиторії лише надають доступ до даних і не приймають бізнес-рішень.

3.5 Бізнес-логіка у Workflow-класах

Шар Workflow-класів зосереджує всю бізнес-логіку та правила застосунку. Структура одного workflow стандартизована: окремий файл містить дві Pydantic-моделі ({Name}Input та {Name}Output) та клас {Name}Workflow із методом-операцією. Залежності (репозиторії, сервіси) приймаються через конструктор. Таблиця 3.5.1 систематизує ключові workflow серверного додатку.

Таблиця 3.5.1

Workflow-класи бекенду

Workflow	Файл	Вхід	Бізнес-правила, що інкапсулюються
RegisterWorkflow	workflows/user/register.py	data: UserCreate	Перевірка унікальності email; хешування пароля через bcrypt; створення користувача
LoginWorkflow	workflows/user/login.py	data: UserLogin	Пошук користувача за email; перевірка пароля через bcrypt; перевірка is_active; генерація JWT-токена
CreateGroupWorkflow	workflows/group/create.py	data: GroupCreate, current_user	Перевірка ліміту 5 груп на користувача (адміністратор обходить); створення групи; додавання творця як учасника з пресетом admin

Workflow	Файл	Вхід	Бізнес-правила, що інкапсулюються
ManageMembersWorkflow	workflows/group/manage_members.py	group_id, user_id, current_user	Перевірка прав керування учасниками; захист власника групи від модифікації; призначення/зняття дозволів
InviteWorkflow	workflows/group/invite.py	group_id, email, current_user	Перевірка членства запрошувача; перевірка унікальності pending-запрошення; генерація токена secrets.token_urlsafe(32); надсилання email через SMTP
InviteWorkflow.accept	workflows/group/invite.py	token, current_user	Валідація токена; перевірка статусу PENDING; перевірка email-збігу; перевірка ліміту 25 учасників; додавання користувача з пресетом member
CreateOrderWorkflow	workflows/order/create.py	group_id, data: OrderCreate, current_user	Перевірка прав на створення (Editor або Initiator); заборона на одночасні активні замовлення; розв'язання ресторану (існуючий чи

Workflow	Файл	Вхід	Бізнес-правила, що інкапсулюються
			новий); створення замовлення в стані INITIATED
OrderLifecycleWorkflow.transition	workflows/order/lifecycle.py	order_id, new_status, current_user	Перевірка існування замовлення; перевірка ролі (ініціатор/Editor/Admin); валідація переходу за VALID_TRANSITIONS; виклик _handle_finish при переході в FINISHED
OrderLifecycleWorkflow.set_delivery_fee	workflows/order/lifecycle.py	order_id, data, current_user	Перевірка стану (не FINISHED/CANCELLED); перерахунок суми per-person ↔ total на основі кількості унікальних учасників
AdjustBalanceWorkflow	workflows/balance/adjust.py	group_id, data: BalanceAdjustment, current_user	Перевірка дозволу Balances:Editor; перевірка членства цільового користувача; оновлення балансу; створення запису в BalanceHistory з

Workflow	Файл	Вхід	Бізнес-правила, що інкапсулюються
			change_type=manual

Особливо складним є workflow життєвого циклу замовлення — OrderLifecycleWorkflow, що визначений у файлі backend/app/workflows/order/lifecycle.py. Він керує переходами між чотирма станами та автоматично оновлює баланси при завершенні замовлення. Уривки реалізації OrderLifecycleWorkflow.transition та приватного методу _handle_finish наведено у Додатку Б (див. Лістинг Б.1, Б.2).

Логіка перерахунку балансів при завершенні замовлення. Метод _handle_finish() (private) виконує таку послідовність дій:

1. Отримати всі позиції замовлення.
2. Для кожного унікального користувача-учасника обчислити суму його позицій як price × quantity.
3. Якщо встановлено delivery_fee_per_person, додати її до суми кожного учасника.
4. Для кожного учасника отримати або створити запис у Balance для пари (user, group).
5. Зменшити Balance.amount на отриману суму та зберегти.
6. Створити запис у BalanceHistory із полями: amount = -total (підписана зміна), balance_after, note = "Order #...8", change_type = order, order_id.
7. Якщо замовлення прив'язане до ресторану, синхронізувати каталог страв: для кожної позиції замовлення оновити ціну існуючої страви (за збігом назви) або створити нову страву із поточною ціною.

Така стратегія перерахунку балансів виконується атомарно у межах однієї транзакції БД, оскільки управляється єдиною сесією AsyncSession, отриманою через DI.

Валідація переходів. Константа `VALID_TRANSITIONS` визначає допустимі переходи у вигляді словника `OrderStatus → list[OrderStatus]`:

- INITIATED → CONFIRMED, CANCELLED
- CONFIRMED → ORDERED, CANCELLED
- ORDERED → FINISHED, CANCELLED
- FINISHED та CANCELLED — термінальні стани, переходів немає.

Спроба недопустимого переходу повертає HTTP 422 з повідомленням "Cannot transition from {current} to {new}".

3.6 API-маршрутизація та структура ендпоінтів

Ендпоінти REST API організовано за допомогою механізму APIRouter фреймворку FastAPI, що дозволяє модульно групувати маршрути за ресурсами. Кожен файл у директорії backend/app/api/ визначає власний router з префіксом та тегом, що групують ендпоінти у Swagger UI. Усі рутери збираються разом у центральному модулі backend/app/api/router.py під єдиним префіксом /api.

Таблиця 3.6.1

Модулі API-роутерів

Модуль	Префікс роутера	Теги	Ендпоінти
api/auth.py	/auth	auth	POST /register, POST /login, POST /logout, GET /me

Модуль	Префікс роутера	Теги	Ендпоінти
api/users.py	/users	users	Список користувачів, оновлення, особистий профіль, аналітика користувача
api/groups.py	/groups	groups	CRUD груп, члени, запрошення, дозволи
api/restaurants.py	/groups/{group_id}/restaurants	restaurants	CRUD ресторанів, CRUD страв
api/orders.py	/groups/{group_id}/orders	orders	CRUD замовлень, активне замовлення, переходи статусу, доставка, позиції, улюблені страви
api/balances.py	/groups/{group_id}/balances	balances	Перегляд балансів, коригування, історія
api/analytics.py	/groups/{group_id}/analytics та /users/me/analytics	analytics	Аналітика групи та користувача

Структура типового ендпоінта. Усі ендпоінти дотримуються спільного шаблону:

1. Сигнатура містить аргументи маршруту (наприклад, group_id: uuid.UUID), тіло запиту (Pydantic-модель), а також залежності через Depends() — поточного користувача, репозиторії, workflow.
2. Простіші ендпоінти (читання списку, отримання за ідентифікатором) звертаються безпосередньо до репозиторію.
3. Складніші ендпоінти (зі складною валідацією та побічними ефектами) делегують виконання workflow-класу через виклик workflow.execute(Input(...)) чи специфічного методу.
4. Для відповідей використовується response_model із Pydantic-схеми, що автоматично виконує валідацію та документує контракт у OpenAPI.

Окремою задачею є захист від міжрезурсних звернень: маршрути ресурсів, що належать групі (наприклад, замовлення з ID X у групі G), завжди перевіряють, що поточний користувач є членом цієї групи (або системним адміністратором). Це робиться на початку обробника через виклик group_member_repository.get_membership(current_user.id, group_id).

3.7 Автентифікація та авторизація

Автентифікація реалізована через JWT-токени, що зберігаються у HTTP-only cookies. Така схема обрана з кількох міркувань: cookies автоматично надсилаються браузером, що спрощує клієнтський код; прапор httponly блокує доступ до cookie з JavaScript, ефективно протидіючи XSS-атакам; прапор secure забезпечує передачу cookie лише через HTTPS; samesite="lax" обмежує надсилання cookie сторонніми сайтами, ускладнюючи CSRF.

Видача токена при вході. Ендпоінт POST /api/auth/login приймає email та пароль, виконує LoginWorkflow, отримує JWT-токен і викликає response.set_cookie("access_token", value=token, httponly=True, secure=True, samesite="lax", max_age=30*60) (термін дії — 30 хвилин). Тіло відповіді містить лише дані користувача (UserResponse), сам токен браузер зберігає у cookie і прозоро надсилає при наступних запитах.

Генерація токена. Функція create_access_token у модулі backend/app/core/security.py формує JWT із двома полями: sub (UUID користувача як рядок) та exp (час закінчення). Підпис виконується алгоритмом HS256 за допомогою бібліотеки python-jose. Секретний ключ та алгоритм керуються через конфігурацію.

Залежність автентифікації get_current_user. Визначена у backend/app/dependencies.py, ця асинхронна функція виконує наступну послідовність:

1. Витягти access_token з cookie запиту (FastAPI декларативно, через Cookie()).
2. Якщо cookie відсутній — підняти AuthError("Not authenticated") (HTTP 401).
3. Викликати decode_access_token(token), що повертає subject (UUID як рядок) або None у разі помилки/закінчення.
4. Якщо токен невалідний — AuthError("Invalid or expired token").
5. Сконвертувати subject у uuid.UUID. Якщо некоректний — AuthError("Invalid token payload").
6. Звернутися до UserRepository.get_by_id(user_id). Якщо користувача немає — AuthError("User not found").
7. Перевірити user.is_active. Якщо ні — AuthError("User account is deactivated").
8. Повернути об'єкт User.

Усі захищені ендпоінти просто декларують у сигнатурі current_user: User = Depends(get_current_user), отримуючи готовий об'єкт користувача.

Залежність get_current_admin. Розширює попередню, додаючи перевірку current_user.role == UserRole.ADMIN. У разі невідповідності — ForbiddenError("Admin access required") (HTTP 403). Використовується для адміністративних ендпоінтів.

Захист на рівні дозволів. Авторизація на рівні групи реалізована не як окрема залежність, а як частина бізнес-логіки workflow-класів та обробників ендпоінтів. Це обґрунтовано тим, що правила надто різноманітні, щоб винести їх у декларативну форму: для деяких дій достатньо членства, для інших — конкретного рівня дозволу певного типу, для третіх — ролі ініціатора замовлення.

3.8 Валідація даних та обробка помилок

Pydantic-схеми. Уся валідація вхідних та вихідних даних виконується через Pydantic v2 [8]. Схеми згруповано за ресурсами у директорії backend/app/schemas/ із спільним базовим модулем schemas/base.py (де визначено PaginatedResponse та

MessageResponse). Кожен ресурс має набір схем — *Create, *Update, *Response, — кожна з яких обмежує допустимі поля для відповідного контексту. Завдяки інтеграції з FastAPI:

- вхідні JSON-тіла автоматично десеріалізуються та валідуються;
- невалідні запити повертають HTTP 422 із детальним описом помилок (у форматі OpenAPI);
- вихідні відповіді автоматично серіалізуються відповідно до response_model, відсікаючи зайві поля (захист від випадкового витоку даних).

Ієрархія виключень. У модулі backend/app/core/exceptions.py визначено власну ієрархію виключень, що успадковується від HTTPException. Таблиця 3.8.1 описує її.

Таблиця 3.8.1

Ієрархія прикладних виключень

Виключення	HTTP-код	Сценарій використання
AppException	—	Базовий клас для всіх прикладних виключень
ValidationError	422	Семантична помилка валідації, що не покривається Pydantic (бізнес-валідація)
AuthError	401	Помилка автентифікації (відсутній токен, невалідний токен, неактивний акаунт)
ForbiddenError	403	Помилка авторизації (недостатньо прав для дії)
NotFoundError	404	Запитуваний ресурс не існує
ConflictError	409	Конфлікт стану (наприклад, email вже зайнятий, активне замовлення вже існує)

Усі workflow та ендпоінти використовують лише виключення з цієї ієрархії; пряме використання HTTPException заборонено для забезпечення консистентності повідомлень про помилки.

Логування помилок. ErrorHandlingMiddleware перехоплює всі необроблені виключення, логує їх через стандартний logging із трасою стека та повторно піднімає, щоб FastAPI повернув коректну відповідь. У промисловому режимі ці винятки також автоматично відсилаються до Sentry [25].

3.9 Керування схемою бази даних через Alembic

Зміни схеми бази даних керуються інструментом Alembic [7], інтегрованим зі SQLAlchemy. Конфігурація знаходиться у директорії backend/alembic/ із файлами env.py (асинхронний skeleton, що зчитує Base.metadata з app.models) та versions/ (файли міграцій).

Робочий процес створення міграції:

1. Розробник вносить зміни до моделей у app/models/.
2. Виконує uv run alembic revision --autogenerate -m "опис змін". Alembic порівнює поточну схему з декларованими моделями та генерує файл міграції з парою функцій upgrade() та downgrade().
3. Розробник переглядає згенерований файл, при потребі коригує його (наприклад, додає міграцію даних), фіксує у git.
4. Застосування міграції відбувається командою uv run alembic upgrade head.

У промисловому розгортанні застосування міграцій виконується не системним сервісом, а скриптом розгортання infrastructure/deploy.sh (див. розділ 5.11). Це свідома архітектурна декомпозиція: міграції — частина процесу деплою, а не серверного процесу, що дозволяє контролювати порядок виконання та діагностувати помилки до перезапуску сервісу.

3.10 Завантаження файлів та статичні ресурси

Серверна частина підтримує завантаження довільних бінарних ресурсів (зокрема — логотипів груп). Параметри завантаження задаються у конфігурації:

UPLOAD_DIR (директорія зберігання, типово /var/www/lunchtogether/uploads у продакшені) та MAX_UPLOAD_SIZE (10 МБ за замовчуванням). Логіка збереження файлів зосереджена у модулі backend/app/core/storage.py, де реалізовано асинхронні операції через aiofiles.

При завантаженні файл отримує унікальне ім'я (на основі UUID), зберігається на диск у конфігурованій директорії, а в моделі записується відносний шлях (logo_path). Файли видаються користувачам через локацію /uploads/ Nginx-конфігурації (див. розділ 5.4) з кешуванням 1 рік (expires 1y), оскільки кожен файл має унікальне імя і не змінюється у часі.

Обсяг обробки файлів обмежено зберіганням і віддачею через /uploads/ без додаткових перетворень (зміни розміру чи конвертації формату), що відповідає сформованим у розділі 1.6 функціональним межам системи.

3.11 Висновки до розділу 3

У розділі реалізовано серверну частину «LunchTogether» як асинхронний REST API на FastAPI з SQLAlchemy-моделями, репозиторіями, Workflow-шаром, JWT-автентифікацією в HTTP-only cookies, Pydantic-валідацією, Alembic-міграціями та інтеграцією Sentry. Найважливіша бізнес-логіка винесена у Workflow-класи, зокрема життєвий цикл замовлення з атомарним оновленням балансів і каталогу страв.

РОЗДІЛ 4 РОЗРОБКА КЛІЄНТСЬКОЇ ЧАСТИНИ

4.1 Структура React-додатку

Клієнтська частина розташована у директорії frontend/ і реалізована як одностороннє веб-застосунок (SPA) на React 19 з TypeScript. Структуру коду побудовано за принципом «глобальні ресурси + функціональні модулі»:

- На глобальному рівні (frontend/src/) знаходяться спільні для всього додатку речі: маршрутизатор, Redux-сховище, RTK Query API-клієнти, типи, константи, переюзні хуки, утиліти та компоненти спільного шару (Layout, ProtectedRoute, ErrorBoundary). Тут же лежать примітиви shadcn/ui у components/ui/.
- На рівні модулів (frontend/src/modules/) кожен функціональний розділ додатку (auth, dashboard, group, restaurant, order, balance, user) має свій набір сторінок (pages/), специфічних компонентів (components/), модульних хуків (hooks/) та модульних типів (types/).

Завдяки такому розподілу нові функціональні модулі додаються без втручання у структуру існуючих, а спільний рівень містить лише дійсно переюзні елементи.

Імпортна модель. Усі імпорти з глобального рівня виконуються через alias @/ (налаштований у tsconfig.json та vite.config.ts). Це робить імпорти короткими, незалежними від глибини файлу та полегшує рефакторинг. Усередині модуля можна використовувати відносні шляхи, а зовнішні споживачі завжди ходять через @/.

4.2 Інструментарій розробки фронтенду

Інструментарій було обрано з огляду на швидкість збирання, типобезпеку та сумісність із сучасним екосистемним стеком React.

Збирач Vite 7 [13]. Замість Webpack чи Create React App, проєкт використовує Vite — швидкий збирач, що базується на esbuild для розробки та Rollup для продакшен-збирання. Vite надає миттєвий старт сервера розробки та оновлення модулів через HMR без перезавантаження сторінки. Команди:

- npm run dev — запускає сервер розробки на порту 5173.
- npm run build — спочатку перевіряє типи через tsc, потім збирає продакшен-бандл у frontend/dist/.
- npm run type-check — лише перевірка типів без збирання (використовується у CI).
- npm run preview — попередній перегляд продакшен-збірки локально.

TypeScript 5. Уся клієнтська кодова база типобезпечна. Конфігурація tsconfig.json містить строгий режим (strict: true) і кілька суворіших перевірок (noUnusedLocals, noUnusedParameters, noImplicitReturns). У CI запускається tsc --noEmit, що блокує merge, якщо у коді з'являються типові помилки.

Стандарт компонентів. Усі компоненти реалізовано як функціональні React-компоненти із іменованим експортом (export function ComponentName(...)). Виняток складає лише ErrorBoundary, реалізований як класовий компонент через специфіку React API для error boundaries.

4.3 Управління станом: Redux Toolkit та RTK Query

Управління серверним станом реалізовано через RTK Query [6] — інструмент Redux Toolkit для автоматичного кешування, інвалідації та повторного завантаження даних з REST API. На відміну від ручного кодування useEffect + fetch, RTK Query дає декларативний підхід із автоматичним керуванням кешем та інвалідацією за тегами.

Базовий API-клієнт. Модуль frontend/src/store/api/baseApi.ts визначає основу для всіх запитів:

- fetchBaseQuery із baseUrl із env.apiBaseUrl та credentials: "include", що автоматично передає HTTP-only cookie з JWT при кожному запиті;
- кастомний baseQueryWithReauth, що обробляє 401-відповіді: при отриманні detail: "Not authenticated" він очищає Redux-стан користувача (auth/clearUser) і виконує перехід на сторінку входу через window.location.href = ROUTES.LOGIN. Винятком є самі ендпоінти автентифікації (login, register, logout, me), для яких 401 не викликає редирект (інакше сторінка входу зациклювалася б);
- createApi із набором тегових типів (Auth, User, Group, GroupMember, Restaurant, Dish, Order, OrderItem, Balance, Analytics), що використовуються для інвалідації кешу.

Модулі API. Для кожного функціонального ресурсу створено окремий API-модуль у frontend/src/store/api/ (authApi, userApi, groupApi, restaurantApi, orderApi, balanceApi, analyticsApi). Усі вони розширюють базовий API через baseApi.injectEndpoints(...). Кожен ендпоінт декларує:

- query функцію, яка повертає URL та опціонально метод/тіло;
- providesTags (для запитів) — список тегів, що цей запит «постачає» у кеш;
- invalidatesTags (для мутацій) — список тегів, кеш яких треба інвалідувати після успіху.

Принцип «список + сутність». Кеш RTK Query організовано так, що для кожного типу сутності використовується спеціальний ID "LIST" для списків та індивідуальні ID для конкретних сутностей. Це дозволяє інвалідувати тільки ті частини кеша, що дійсно змінилися — наприклад, після створення замовлення інвалідуються теги { type: "Order", id: "LIST" } та { type: "Order", id: "ACTIVE" }, але деталі інших замовлень не оновлюються.

Slice'и Redux. Slice'и (authSlice, userSlice) використовуються лише для клієнтського стану: інформація про поточного користувача (для швидкого доступу

без HTTP-запиту), UI-стани. Серверні дані не дублюються у slice'ах — за них відповідає кеш RTK Query.

Типізовані хуки. Файл frontend/src/hooks/useAppDispatch.ts та useAppSelector.ts експортують типізовані обгортки навколо useDispatch/useSelector, що ловлять помилки на етапі компіляції.

4.4 Маршрутизація та захищені маршрути

Маршрутизація реалізована через React Router v7 [10] із використанням API createBrowserRouter (на відміну від декларативної <BrowserRouter><Routes>, цей підхід дозволяє декларативно описати дерево маршрутів як об'єкт та використовувати data-router-функції, доступні у v7). Файл frontend/src/routes/index.tsx визначає всі маршрути додатку.

Кореневий маршрут обгортає всі дочірні елементи у компоненти ErrorBoundary та Layout (із Header + Sidebar + Footer). Кожен захищений маршрут також обгорнуто у компонент ProtectedRoute, який перенаправляє неавтентифікованих користувачів на сторінку входу.

Перелік маршрутів наведено в константах frontend/src/constants/routes.ts. Таблиця 4.4.1 узагальнює їх.

Таблиця 4.4.1

Маршрути клієнтського додатку

Маршрут	Сторінка	Призначення
/	UserDashboardPage	Головна сторінка користувача з аналітикою
/login	LoginPage	Вхід
/register	RegisterPage	Реєстрація

Маршрут	Сторінка	Призначення
/profile	ProfilePage	Профіль користувача
/settings	SettingsPage	Налаштування
/users	UserListPage	Адмін: список усіх користувачів
/users/:id	UserDetailPage	Адмін: деталі користувача
/groups	GroupListPage	Список груп користувача
/groups/:groupId	GroupDetailPage	Дашборд групи
/groups/:groupId/members	GroupMembersPage	Учасники групи
/groups/:groupId/restaurants	RestaurantListPage	Список ресторанів
/groups/:groupId/restaurants/:restaurantId	RestaurantDetailPage	Деталі ресторану зі стравами
/groups/:groupId/orders	OrderListPage	Список замовлень
/groups/:groupId/orders/:orderId	OrderDetailPage	Деталі замовлення
/groups/:groupId/balances	BalancesPage	Баланси учасників
*	NotFoundPage	404

Компонент ProtectedRoute (frontend/src/components/common/ProtectedRoute/ProtectedRoute.tsx) перевіряє наявність поточного користувача у Redux-стані. Якщо користувач не автентифікований — компонент здійснює навігаційний редирект на /login через <Navigate to={ROUTES.LOGIN} replace />. Це гарантує, що жодна захищена сторінка не рендериться без валідної сесії.

Завантаження при старті. Кореневий компонент App (frontend/src/App.tsx) при монтуванні виконує useGetCurrentUserQuery(), що звертається до /api/auth/me із cookie-токеном. Поки запит виконується — відображається повноекранний спінер; після його завершення (успіх або помилка) рендериться RouterProvider. Це усуває

ефект «миготіння» інтерфейсу, коли спочатку показується сторінка для гостя, а потім швидко перемикається на сторінку для автентифікованого користувача.

4.5 Двопанельна навігаційна модель

Бічна навігація додатку має оригінальну двопанельну структуру, натхненну Discord. Зовнішня вузька панель показує іконки (домашня кнопка та список груп користувача), а внутрішня — підрозділи відповідного контексту. При зміні маршруту навігація автоматично перемикається.

Реалізація знаходиться у файлі frontend/src/components/common/Layout/Sidebar.tsx. Логіка визначення контексту базується на двох допоміжних функціях:

- isHomeContext(pathname) — повертає true, якщо поточний шлях належить до домашніх (/, /profile, /settings, /users/...).
- isGroupContext(pathname) — повертає true, якщо шлях починається з /groups/....

На основі цих функцій компонент рендерить різний набір елементів навігації:

- Домашній контекст — підрозділи Home, Profile, Settings; для адміністратора додається Manage Users.
- Контекст групи — підрозділи Dashboard, Members, Restaurants, Orders, Balances; обраний activeGroupId витягується з параметра маршруту через useParams або обчислюється з location.pathname.

Іконкова панель завжди показує домашню кнопку, список груп користувача (через useGetGroupsQuery()) та кнопку «+», що відкриває діалог створення нової групи. Активна іконка візуально виділяється змінами форми (з кругу на квадрат із заокругленими кутами) та фоном.

4.6 Дизайн-система shadcn/ui та Tailwind CSS

Інтерфейс побудовано на бібліотеці компонентів shadcn/ui [3] — наборі копійованих у проєкт UI-компонентів на базі Radix UI Primitives. На відміну від класичних бібліотек (MUI, Ant Design), де компоненти живуть як зовнішні залежності, shadcn/ui-компоненти імпортуються у проєкт у вигляді вихідних файлів у директорію frontend/src/components/ui/. Це дає кілька переваг:

- повний контроль над стилями та поведінкою (можна змінити будь-який рядок);
- відсутність runtime-залежностей від важких бібліотек компонентів;
- сумісність із Tailwind CSS — стилі живуть у тих самих файлах, що й розмітка;
- доступність (a11y) забезпечена Radix UI Primitives (керування фокусом, screen-reader, клавіатура).

Перелік використаних shadcn/ui-компонентів:

Таблиця 4.6.1

Використані shadcn/ui компоненти

Компонент	Призначення
Button	Кнопки з варіантами default, outline, destructive тощо
Card	Контейнери карткового стилю
Dialog	Модальні діалоги (створення групи, замовлення)
Input	Поля введення тексту
Label	Підписи до полів
Form	Інтеграція з react-hook-form через провайдер контексту
Alert	Повідомлення про помилки
Popover	Розкривні підказки (база для Combobox)

Компонент	Призначення
Combobox	Власна реалізація: випадаючий список із пошуком та опцією створення нового елемента
Sonner	Тост-сповіщення

Tailwind CSS v4 [9]. Стилізація реалізована утилітарним підходом через Tailwind. У версії 4 використовується нова інтеграція через плагін @tailwindcss/vite, що усуває потребу у PostCSS та значно прискорює збирання. Класи компонуються в JSX через утиліту cn() (із пакету class-variance-authority + clsx + tailwind-merge), що дозволяє коректно об'єднувати умовні класи без конфліктів специфічності.

Іконки. Усі іконки беруться з пакету lucide-react — оптимізованої колекції SVG-іконок із tree-shaking. Приклади використання: Home, LayoutDashboard, ShoppingCart, Wallet, Users, UtensilsCrossed, Plus, Settings, User, ShieldCheck.

4.6.1 Кольори, типографіка та доступність

Палітра кольорів задана у вигляді CSS-змінних у файлі frontend/src/index.css у форматі HSL, що дозволяє декларативно перемикати тему й перевизначати окремі ролі для конкретних компонентів. Усі утиліти Tailwind звертаються до цих змінних (наприклад, bg-primary, text-foreground, border-border), що усуває жорстко прописані шістнадцяткові коди у JSX. Базові ролі: background / foreground — фон та основний текст; primary — акцентний колір (фірмова бузково-синя гама); secondary — приглушений акцент для другорядних кнопок; muted — поверхні низького контрасту; destructive — небезпечні дії (видалення, помилки); border та input — кольори меж і полів.

Типографіку реалізовано за принципом «один шрифтовий стек на всю систему»: вільний sans-serif із системних шрифтів (-apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, "Helvetica Neue", Arial, sans-serif), що забезпечує нативний вигляд на основних платформах без додаткового завантаження. Розміри текстів узгоджено через утиліти Tailwind (text-xs, text-sm,

text-base, text-lg, text-xl, text-2xl, text-3xl), що формують консистентну вертикальну ритміку.

Доступність (WCAG 2.1 [33]) забезпечена кількома взаємодоповнюючими механізмами:

- усі інтерактивні елементи (Button, Dialog, Combobox, Popover, Form) побудовано на Radix UI Primitives [14], які за замовчуванням підтримують клавіатурну навігацію, керування фокусом, ARIA-атрибути та режим screen-reader;
- кольорові пари (текст на фоні) у базовій темі мають коефіцієнт контрасту, що відповідає рівню AA WCAG;
- фокус-кільце видиме на всіх інтерактивних елементах за рахунок утиліт focus-visible:ring-2 focus-visible:ring-ring;
- усі поля форм мають програмно пов'язаний Label через атрибути htmlFor/id, що автоматично забезпечує react-hook-form + shadcn FormField;
- сповіщення sonner мають коректні ARIA-ролі (role="status" для інформаційних, role="alert" для помилок).

4.7 Сторінки додатку за функціональними модулями

Фронтенд поділено на функціональні модулі, кожен з яких відповідає окремій групі користувацьких сценаріїв:

Модуль	Основні сторінки	Призначення
auth	LoginPage, RegisterPage	Вхід, реєстрація, первинна валідація форм
dashboard	UserDashboardPage, SettingsPage	Загальна аналітика користувача та персональні налаштування
group	GroupListPage, GroupDetailPage, GroupMembersPage	Список груп, дашборд групи, учасники та запрошення

Модуль	Основні сторінки	Призначення
user	ProfilePage, UserListPage, UserDetailPage	Профіль і адміністративне керування користувачами
restaurant	RestaurantListPage, RestaurantDetailPage	Каталог ресторанів і страв у межах групи
order	OrderListPage, OrderDetailPage	Створення замовлення, позиції, статуси, доставка
balance	BalancesPage	Поточні баланси, ручні коригування та історія
invitations	PendingInvitesPage, AcceptInvitationPage	Вхідні запрошення та прийняття токена з email

Для ілюстрації інтерфейсу достатньо показати не всі екрани, а ключові контрольні точки: реєстрацію/вхід, дашборд групи, сторінку учасників із запрошенням, список ресторанів, створення замовлення, деталі замовлення у двох станах, сторінку балансів і сторінку прийняття запрошення. Решта екранів описані текстово, оскільки повторюють ті самі UI-патерни: таблиці, діалоги, inline-валідацію, тости та захищену навігацію.

4.8 Власні компоненти та форми

Власний компонент Combobox (frontend/src/components/ui/combobox.tsx) — це не лише стандартний випадаючий список, а спеціалізована форма введення, що поєднує:

- пошук серед існуючих опцій із підсвічуванням збігів;
- опцію «Create …» для створення нового елемента просто з вводу (відображається лише якщо введений текст не збігається з жодною з опцій);
- клавіатурну навігацію через стрілки + Enter;
- доступність на основі Radix Popover та cmdk.

Компонент використовується при виборі ресторану при створенні замовлення, що дозволяє швидко або вибрати існуючий ресторан, або одразу створити новий, не виходячи з діалогу.

Шаблон форми. Усі форми в додатку дотримуються єдиного шаблону, описаного у конвенціях проєкту:

1. Схема валідації описується через zod (у frontend/src/utils/validation.ts для переюзних схем або локально в модулі).
2. Логіка форми (стан, обробка submit, серверні помилки, прапор isSubmitting) виноситься в окремий хук use<Name>Form.
3. Компонент рендерить shadcn-обгортки Form, FormField, FormControl, FormMessage, що автоматично пов'язують react-hook-form із UI.

Цей підхід усуває дублювання boilerplate-коду між формами та чітко розділяє відповідальність: схема валідації, логіка та подання живуть у різних місцях.

4.9 Обробка помилок та користувацький досвід

Глобальний ErrorBoundary. Будь-яка неперехоплена помилка у дереві компонентів React зловлюється класовим компонентом ErrorBoundary (frontend/src/components/common/ErrorBoundary/ErrorBoundary.tsx), що відображає дружнє повідомлення про помилку замість «білого екрану». Це особливо важливо для SPA, де runtime-помилка може зламати весь додаток.

Стани завантаження та помилки. Усі сторінки, що виконують HTTP-запити, обробляють три стани відповідно до конвенцій проєкту:

- Loading — відображається спінер animate-spin rounded-full border-4 border-primary border-t-transparent.
- Error — відображається shadcn-компонент Alert з варіантом destructive та повідомленням помилки.

- Empty — централізований muted-текст на кшталт «Немає замовлень. Створіть перше замовлення, щоб розпочати».

Тост-сповіщення. Бібліотека sonner використовується для коротких сповіщень про успішні дії та помилки. Тости з'являються у правому нижньому куті, автоматично зникають через кілька секунд та не блокують роботу. Приклади: «Group created», «Order finished — balances updated», «Failed to invite user».

Реакція на 401. Як описано у розділі 4.3, кастомний baseQueryWithReauth автоматично виходить з сесії при отриманні 401 від будь-якого ендпоінту (крім самих auth-ендпоінтів). Це гарантує, що користувач, чий токен закінчився, не побачить розсипаних на сторінці помилок, а буде акуратно перенаправлений на сторінку входу.

4.10 Висновки до розділу 4

У розділі 4 реалізовано клієнтську частину додатку «LunchTogether» у вигляді одностороннього веб-додатку (SPA) на React 19 з TypeScript та Vite 7. Кодову базу організовано за принципом «глобальні ресурси + функціональні модулі», де кожний з восьми модулів (auth, dashboard, group, user, restaurant, order, balance, invitations) має чітко відокремлені сторінки, компоненти та модульні хуки.

Реалізовано шар управління серверним станом на основі Redux Toolkit з RTK Query [6] із автоматичною інвалідацією кешу за тегами, кастомним обробником 401-відповідей для безшовного відновлення сесії, типобезпечними хуками для всіх запитів. Усі захищені маршрути загорнуто у компонент ProtectedRoute, а маршрутизація реалізована через нову API React Router v7 [10]. Реалізовано оригінальну двопанельну бічну навігацію із автоматичним перемиканням контексту залежно від поточного маршруту.

Для побудови інтерфейсу використано бібліотеку shadcn/ui [3] поверх Radix UI Primitives та Tailwind CSS v4 [9], що дало повний контроль над стилями при

збереженні доступності та консистентності. Реалізовано власний компонент Combobox із функцією створення нового елемента «на льоту», що значно покращує UX діалогу створення замовлення. Усі форми побудовано за єдиним шаблоном з валідацією zod та інтеграцією react-hook-form. Реалізовано глобальний ErrorBoundary, стандартизовану обробку станів (Loading/Error/Empty) та тост-сповіщення на основі sonner. Клієнтська частина повністю функціональна та готова до розгортання у промисловому середовищі, що описано у розділі 5.

РОЗДІЛ 5 РОЗГОРТАННЯ ТА ЕКСПЛУАТАЦІЯ СИСТЕМИ

5.1 Вибір середовища розгортання

Для розгортання системи «LunchTogether» обрано виділений сервер VPS під управлінням Ubuntu Server 22.04 LTS [19]. Цей вибір обґрунтовано кількома міркуваннями:

- Повний контроль над середовищем. На відміну від platform-as-a-service рішень (Heroku, Render, Railway), VPS дає необмежений доступ до системних компонентів — Nginx, systemd, PostgreSQL, cron — що дозволяє відтворити промислову інфраструктуру у формі, наближеній до реальних умов експлуатації корпоративних веб-додатків.
- Передбачувана вартість. VPS із гарантованими ресурсами (CPU, RAM, диск) має фіксовану місячну вартість незалежно від навантаження, що зручно для академічного проєкту.
- LTS-версія Ubuntu. Версія 22.04 LTS [19] має тривалу підтримку (до 2027 року для звичайних оновлень), стабільні пакети та широку документацію.
- Сумісність із обраним стеком. Усі компоненти стеку (Python 3.12, PostgreSQL 16, Nginx, systemd) офіційно підтримуються Ubuntu і доступні через стандартні репозиторії (із додаванням ppa:deadsnakes/ppa для Python 3.12).

Рекомендовані характеристики сервера для одного робочого колективу (до 5 груп × 25 учасників, ~100 активних користувачів) такі: 2 ядра CPU, 4 ГБ ОЗП, 40 ГБ SSD-диска. Ці характеристики є достатніми для базового навантаження проєкту і відповідають типовому тарифу VPS у цінових діапазонах 7–15 €/міс. серед європейських провайдерів.

Уся послідовність налаштування сервера автоматизована у скрипті infrastructure/setup.sh, що приймає шість параметрів (домен, пароль БД, JWT-секрет, Sentry DSN, email для SSL, URL репозиторію) та виконує одноразове налаштування «з нуля». Решта розділу описує кожен з його кроків та артефактів, що створюються.

Топологія розгортання. Розгортання системи має чотири взаємодіючі рівні: (1) клієнтський браузер користувача → (2) Nginx-проксі на 443/TCP, що термінує TLS і обслуговує SPA та статичні файли → (3) внутрішній uvicorn-процес FastAPI на 127.0.0.1:8000 з 4 робочими процесами → (4) PostgreSQL, що слухає лише на localhost:5432. Поруч із застосунком у тій самій VM працюють cron-завдання резервного копіювання БД (щодоби) та поновлення TLS-сертифіката (щодоби о 3:00).

5.2 Реєстрація домену та налаштування DNS

Перед запуском скрипта розгортання зареєстровано доменне ім'я та налаштовано DNS-записи, що вказують на IP-адресу VPS. Як реєстратор обрано Namecheap [34] — комерційного реєстратора, акредитованого ICANN, що надає у складі базової послуги панель керування DNS-записами («Advanced DNS») та можливість підключення додаткових сервісів (зокрема корпоративної електронної пошти, див. підрозділ 5.2.1). Перевагами Namecheap для цього проєкту є невелика річна вартість домену в зоні .food, відсутність прихованих платежів за зміну DNS-серверів, безкоштовний захист WHOIS-приватності та інтегрована служба Private Email, що дозволяє розгорнути корпоративну пошту на тому самому домені без окремого SMTP-провайдера.

Налаштовані DNS-записи. У панелі Domain List → Manage → Advanced DNS для домену lunchtogether.food створено наступні записи (Таблиця 5.2.1).

Таблиця 5.2.1

DNS-записи домену lunchtogether.food

Тип	Host	Значення	TTL	Призначення
A	@	<IP-адреса VPS>	Automatic	Апекс-домен lunchtogether.food вказує на сервер застосунку
A	www	<IP-адреса VPS>	Automatic	Піддомен www.lunchtogether.food вказує на той самий сервер
MX	@	mx1.privateemail.com (priority 10)	Automatic	Основний поштовий маршрут Namecheap Private Email
MX	@	mx2.privateemail.com (priority 10)	Automatic	Резервний поштовий маршрут
TXT	@	v=spf1 include:spf.privateemail.com ~all	Automatic	SPF-політика: автентифікація відправників
TXT	default._domainkey	DKIM-ключ, виданий Namecheap	Automatic	DKIM-підпис вихідних листів
TXT	_dmarc	v=DMARC1; p=none; rua=mailto:postmaster@lunchtogether.food	Automatic	DMARC-політика моніторингу
CNAME	autodiscover	privateemail.com.	Automatic	Автоматичне налаштування поштових клієнтів

Обидва імені A-записів — апекс-домен та піддомен www — використовуються у nginx-конфігурації та у запиті TLS-сертифіката, тому обидва мають бути присутні (див. розділи 5.4 та 5.5). Записи MX, SPF, DKIM і DMARC потрібні для коректної доставки вихідних запрошень із системи (див. підрозділ 5.2.1).

Зміна DNS-записів пропагується глобально протягом часу від кількох хвилин до кількох годин залежно від TTL у попередній конфігурації, тому процедуру виконано за добу до встановлення TLS-сертифіката. Перевірку коректності записів проведено за допомогою команди dig із локальної машини:

dig +short lunchtogether.food A
dig +short www.lunchtogether.food A
dig +short lunchtogether.food MX
dig TXT lunchtogether.food +short
Відповіді мають збігатися зі значеннями таблиці 5.2.1.

5.2.1 Корпоративна електронна пошта через Namecheap Private Email

Для надсилання запрошень новим учасникам груп (див. розділ 3.5 та клас EmailService у backend/app/core/email.py) потрібен надійний SMTP-сервер, який підписує вихідні листи DKIM та збігається з SPF-політикою домену. Для цього обрано та підключено сервіс Namecheap Private Email [34] — комерційне поштове рішення на базі OpenXchange, що пропонується Namecheap як додаткова послуга до зареєстрованого домену.

Обґрунтування вибору. Альтернативні варіанти мали суттєві обмеження:

- SMTP від хмарних провайдерів (Mailgun, SendGrid, Amazon SES) — потужні, але потребують додаткової верифікації домену, окремого облікового запису поза Namecheap і часто мають обмеження безкоштовного тарифу, що не дозволяє створити повноцінну поштову скриньку.

- Локальний SMTP-сервер на VPS (Postfix із власною конфігурацією) — ускладнює DKIM-керування, потребує підтримки чорних/білих списків і часто маркується як спам великими поштовими провайдерами через відсутність репутації IP-адреси VPS.
- Безкоштовні поштові сервіси (Gmail SMTP) — не дозволяють використовувати власний домен як «From» без додаткової верифікації, а тарифні обмеження SMTP-сесій непридатні для надсилання запрошень із сервера.

Namecheap Private Email пропонує: окрему поштову скриньку на власному домені (noreply@lunchtogether.food), вбудоване DKIM-підписування, готову SPF-політику, веб-інтерфейс для перегляду надісланих листів, SMTP/IMAP-доступ для застосунку та фіксовану річну вартість (~12 €/рік за одну скриньку), що відповідає вимозі передбачуваної вартості з розділу 1.6.

Створена поштова скринька. У панелі Namecheap → Private Email активовано тарифний план Starter та створено скриньку noreply@lunchtogether.food із власним паролем. Скринька використовується винятково для системних повідомлень, тому її іменування відповідає семантичній конвенції «no-reply» — відповіді на ці листи не очікуються.

Інтеграція з застосунком. Доступ до SMTP-сервера Namecheap налаштовано через змінні оточення, що зчитуються pydantic-settings у класі Settings (backend/app/config.py, див. поля з префіксом smtp_*) та передаються бібліотеці aiosmtplib всередині EmailService.send_email(). Параметри з'єднання наведено у Таблиці 5.2.2.

Таблиця 5.2.2

Параметри SMTP-з'єднання Namecheap Private Email

Змінна .env	Значення	Призначення
SMTP_HOST	mail.privateemail.com	Хост SMTP-сервера Namecheap Private Email
SMTP_PORT	587	Порт STARTTLS-з'єднання (відкритий вихідний на VPS)
SMTP_USER	noreply@lunchtogether.food	Повна адреса поштової скриньки як логін
SMTP_PASSWORD	<пароль скриньки>	Зберігається лише у .env на сервері (доступ chmod 600)
SMTP_FROM_EMAIL	noreply@lunchtogether.food	Поле From у листах, що збігається з логіном
SMTP_FROM_NAME	LunchTogether	Дружнє ім'я відправника
SMTP_USE_TLS	true	Активує STARTTLS після TCP-handshake

Тіло листа-запрошення формується у методі EmailService.send_invitation_email(): підставляються ім'я запрошувача, назва групи та посилання-токен виду https://lunchtogether.food/invitations/accept?token=<...>. HTML-шаблон листа разом із кодом наведено у Додатку Д.

Перевірка доставки. Після підключення Private Email виконано контрольну перевірку: відправлено запрошення на тестову скриньку Gmail та проконтрольовано, що (1) лист потрапляє у «Inbox», а не «Spam»; (2) у заголовках листа поля Authentication-Results містять spf=pass та dkim=pass; (3) у відповіді SMTP-сервера повертається 250 2.0.0 OK. Цей контроль підтверджує коректність налаштування DNS-записів MX, SPF і DKIM зі стандартної таблиці 5.2.1.

Розмежування з іншими секретами. Пароль до поштової скриньки зберігається лише у файлі .env на VPS (з правами 600 та власником — обліковим записом застосунку), не комітиться в репозиторій і не передається у CI як секрет, оскільки робочі процеси ci.yml і deploy.yml (див. розділ 5.10) не надсилають листи.

У .env.example для розробників наведено заглушку (SMTP_PASSWORD=changeme), а локальна розробка ведеться з налаштуванням ENVIRONMENT=development, за якого EmailService логує помилки SMTP без падіння запиту (див. метод send_email у backend/app/core/email.py).

5.3 PostgreSQL на сервері

На етапі 12 скрипта setup.sh ставиться пакет postgresql postgresql-contrib, після чого виконується одноразова ініціалізація бази даних та користувача через psql з-під системного користувача postgres:

1. CREATE DATABASE lunchtogether — створює базу даних.
2. CREATE USER lunchtogether WITH ENCRYPTED PASSWORD '<пароль>' — створює окремого користувача БД, що має доступ лише до власної бази.
3. GRANT ALL PRIVILEGES ON DATABASE lunchtogether TO lunchtogether та ALTER DATABASE ... OWNER TO ... — встановлюють володіння.
4. Перехід у новостворену БД та GRANT ALL ON SCHEMA public — права на схему public.
5. CREATE EXTENSION IF NOT EXISTS "uuid-ossp" — встановлює розширення для генерації UUID, потрібне моделям SQLAlchemy із UUID-первинними ключами.

Захист мережевого рівня. Параметр listen_addresses у /etc/postgresql/*/main/postgresql.conf виставляється у значення 'localhost'. Це означає, що сервер БД приймає з'єднання лише через локальний інтерфейс і недоступний з зовнішньої мережі. Бекенд звертається до БД через localhost:5432, що повністю усуває ризик прямого мережевого доступу до бази даних із зовні.

Сервіс PostgreSQL вмикається через systemctl restart postgresql та позначається systemctl enable postgresql для автоматичного запуску при перезавантаженні системи.

5.4 Реверс-проксі Nginx

Усі вхідні HTTP/HTTPS-запити обробляються веб-сервером Nginx [16], що виконує роль реверс-проксі та сервера статичних файлів. Шаблон конфігурації знаходиться у файлі infrastructure/nginx/lunchtogether.conf та містить два server-блоки: для портів 80 (HTTP) та 443 (HTTPS). Конфігурація розміщується у /etc/nginx/sites-available/lunchtogether із заміною плейсхолдерів DOMAIN_PLACEHOLDER та APP_DIR_PLACEHOLDER, після чого створюється симлінк у /etc/nginx/sites-enabled/. Стандартний конфіг за замовчуванням вимикається через видалення симлінка default.

HTTP-блок (порт 80) виконує дві задачі. По-перше, обслуговує локацію /.well-known/acme-challenge/ з кореня /var/www/certbot/ — це потрібно для ACME HTTP-01 челенджу Certbot при отриманні та автоматичному оновленні TLS-сертифіката (див. розділ 5.5). По-друге, всі інші запити перенаправляються на HTTPS через return 301 https://$host$request_uri. Це жорсткий редирект, що гарантує — користувачі не потраплять на незахищене з'єднання випадково.

HTTPS-блок (порт 443) є основним і обслуговує всі решта запитів:

- TLS-конфігурація. Сертифікати беруться зі стандартного шляху Certbot: /etc/letsencrypt/live/<domain>/fullchain.pem та privkey.pem. Дозволено лише сучасні протоколи TLSv1.2 і TLSv1.3; шифри обмежено HIGH:!aNULL:!MD5.
- Заголовки безпеки. Додаються чотири заголовки безпеки HTTP: X-Frame-Options: SAMEORIGIN (захист від clickjacking), X-Content-Type-Options: nosniff (заборона MIME-снифінгу), X-XSS-Protection: 1; mode=block (legacy-XSS-захист), Strict-Transport-Security: max-age=31536000; includeSubDomains (HSTS на 1 рік).
- Обмеження тіла запиту. client_max_body_size 10M — узгоджено з конфігурацією бекенду (MAX_UPLOAD_SIZE).

- API-проксі. Локація /api/ проксує запити на бекенд (proxy_pass http://backend, де upstream backend посилається на 127.0.0.1:8000) з повним набором proxy-заголовків (X-Real-IP, X-Forwarded-For, X-Forwarded-Proto, Host) та з підтримкою keep-alive через proxy_http_version 1.1. Таймаути встановлено достатньо щедрими (proxy_read_timeout 300s) для обробки потенційно довгих запитів.
- Статичні файли uploads. Локація /uploads/ обслуговує завантажені користувачем файли безпосередньо з диска з агресивним кешуванням (expires 1y, Cache-Control: public, immutable), оскільки кожен файл має унікальне ім'я.
- Фронтенд SPA. Локація / обслуговує статичний продакшен-збір React-додатку з директорії frontend/dist/ із директивою try_files $uri $uri/ /index.html, що забезпечує клієнтську маршрутизацію — будь-який запит до неіснуючої статичної сторінки повертає index.html, після чого React Router відображає правильну сторінку. Окремо налаштовано тривалий кеш для асетів за регулярним виразом (\.(js|css|png|jpg|jpeg|gif|ico|svg|woff|woff2|ttf|eot)$).

Після формування конфігурації виконується nginx -t для перевірки синтаксису, systemctl restart nginx для застосування та systemctl enable nginx для автозапуску.

5.5 HTTPS з Let's Encrypt та Certbot

Сертифікат TLS отримується автоматично через службу Let's Encrypt [11] за допомогою інструменту Certbot із плагіном certbot-nginx. Запит підписується через ACME HTTP-01 челендж: Certbot створює тимчасовий файл у /var/www/certbot/.well-known/acme-challenge/, Let's Encrypt запитує його за HTTP та переконується, що домен дійсно належить заявнику.

Проблема «курка і яйце». При першому запуску виникає нетривіальна ситуація: nginx-конфігурація містить директиви ssl_certificate /etc/letsencrypt/live/<domain>/fullchain.pem та ssl_certificate_key

/etc/letsencrypt/live/<domain>/privkey.pem, але самі файли ще не існують (Certbot не запускався). Це призводить до того, що nginx -t фейлить, nginx не стартує, тож Certbot не може використати HTTP-челендж (бо немає кому обслуговувати /.well-known/acme-challenge/).

Розв'язання — самопідписаний bootstrap. У скрипті setup.sh (крок 14) перед першим запуском nginx генерується короткостроковий самопідписаний сертифікат на ту саму пару шляхів:

1. Створюється директорія /etc/letsencrypt/live/<domain> (хоча в норміальній схемі це робить Certbot).
2. Через openssl req -x509 -nodes -newkey rsa:2048 -days 1 ... створюється цифровий ключ та самопідписаний сертифікат із терміном дії 1 день.
3. Конфігурація nginx проходить перевірку (nginx -t), nginx стартує і обслуговує /.well-known/acme-challenge/.

Виклик Certbot з webroot-плагіном. Замість --nginx (який виконує nginx -t як пред-перевірку, що знову призведе до помилки після видалення dummy-сертифіката) використовується certbot certonly --webroot -w /var/www/certbot -d <domain> -d www.<domain> --non-interactive --agree-tos -m <email>. Цей режим вимагає лише, щоб nginx обслуговував HTTP-челендж із /var/www/certbot, що вже забезпечено HTTP-блоком конфігурації. Перед викликом Certbot директорії live/<domain> та archive/<domain> видаляються (rm -rf), щоб Certbot створив чистий набір символічних посилань. Важливо: nginx залишається запущеним і використовує самопідписаний сертифікат, що вже завантажений у пам'ять процесу.

Завершення налаштування TLS. Після успішного запиту Certbot записує справжні сертифікати у /etc/letsencrypt/live/<domain>/fullchain.pem та privkey.pem. Команди nginx -t && systemctl reload nginx змушують nginx перечитати сертифікати з диска. З цього моменту вебсайт доступний за HTTPS із валідним сертифікатом, прийнятим усіма основними браузерами.

Автоматичне оновлення. Сертифікати Let's Encrypt мають термін дії 90 днів, тому необхідне автоматичне оновлення. Crontab-завдання 0 3 * * * certbot renew --quiet --post-hook 'systemctl reload nginx' запускається щодня о 3:00 і виконує оновлення тих сертифікатів, що мають менше ніж 30 днів до закінчення. Після успішного оновлення hook systemctl reload nginx забезпечує перечитування нових сертифікатів.

5.6 Systemd-сервіс бекенду

Бекенд запускається як системний сервіс через systemd [18], що забезпечує автозапуск при перезавантаженні системи, перезапуск при падінні процесу та централізоване логування через journald. Шаблон unit-файлу знаходиться у infrastructure/systemd/lunchtogether-backend.service; скрипт setup.sh копіює його у /etc/systemd/system/, замінюючи плейсхолдери APP_USER_PLACEHOLDER та APP_DIR_PLACEHOLDER на справжні значення.

Таблиця 5.6.1

Параметри systemd-юніту

Секція	Параметр	Значення / призначення
[Unit]	Description	Описова назва сервісу для логів та статусу
[Unit]	After / Requires	network.target postgresql.service — сервіс стартує після мережі та PostgreSQL
[Service]	Type	simple — процес-головний, без форку
[Service]	User / Group	lunchtogether / www-data — непривілейований обліковий запис
[Service]	WorkingDirectory	/var/www/lunchtogether/backend
[Service]	EnvironmentFile	/var/www/lunchtogether/backend/.env — секрети та конфігурація

Секція	Параметр	Значення / призначення
[Service]	ExecStart	uv run uvicorn app.main:app --host 127.0.0.1 --port 8000 --workers 4
[Service]	Restart / RestartSec	always / 10 — автоматичний перезапуск через 10 с після падіння
[Service]	NoNewPrivileges	true — заборона підвищення привілеїв
[Service]	PrivateTmp	true — ізольований /tmp для процесу
[Service]	StandardOutput / Error	journal — логи перехоплюються journald та доступні через journalctl
[Install]	WantedBy	multi-user.target — сервіс запускається на звичайному рівні (не graphical)

Чому міграції не у ExecStartPre. Хоча інтуїтивно зручно прив'язати запуск міграцій до старту сервісу через ExecStartPre=alembic upgrade head, у конфігурації цей крок свідомо винесено у deploy.sh. Причина: якщо PostgreSQL короткочасно недоступна (наприклад, при її окремому перезапуску), сервіс не зможе перезапуститися, що погіршує доступність. Окрім того, міграції — це разова дія при оновленні версії, а не дія, що повторюється на кожному рестарті.

Сервіс активується командами systemctl daemon-reload, systemctl enable lunchtogether-backend (для автозапуску) та згодом systemctl start lunchtogether-backend (виконує deploy.sh).

Логи та діагностика. Поточний стан сервісу видно командою systemctl status lunchtogether-backend. Реальний журнал — journalctl -u lunchtogether-backend -f (стрім логів). Усі помилки виключень бекенду логуються ErrorHandlingMiddleware та автоматично надсилаються до Sentry [25] (див. розділ 5.9).

5.7 Автоматизовані резервні копії бази даних

Резервне копіювання БД виконується щодоби о 2:00 через cron-завдання, що викликає скрипт infrastructure/scripts/backup-db.sh (встановлений у /usr/local/bin/backup-lunchtogether-db). Алгоритм скрипта:

1. Зчитати пароль БД зі статичного секретного файлу /etc/lunchtogether/backup.env (формат PGPASSWORD=...). Файл має права 600 і належить root, що захищає його від читання непривілейованими процесами.
2. Створити директорію /var/backups/lunchtogether/ (якщо ще не існує).
3. Сформувати ім'я файлу з timestamp: lunchtogether_YYYYMMDD_HHMMSS.sql.gz.
4. Виконати pg_dump -U lunchtogether -h localhost lunchtogether | gzip > <file> — створення стиснутого SQL-дампу.
5. Виставити права 600 на створений файл.
6. Видалити дампи, старші за 30 днів (find ... -mtime +30 -delete).

Чому секрети в окремому файлі? Прямий запис пароля у crontab чи у скрипт у /usr/local/bin/ створював би його повсюдну доступність. Виділений секретний файл із суворими правами доступу відокремлює секрет від коду, що дозволяє ротувати пароль БД без зміни самого скрипта.

Відновлення з резервної копії. Скрипт infrastructure/scripts/restore-db.sh приймає шлях до файлу дампу, після інтерактивного підтвердження зупиняє сервіс бекенду, виконує gunzip + psql для відновлення, потім стартує бекенд знову. Це гарантує консистентність — бекенд не намагається виконувати запити до БД у момент відновлення.

5.8 Ротація логів

Хоча journald (system logs systemd-сервісу) має власну стратегію ротації, окремо налаштовано ротацію логів додатку, що пишуться у директорію

/var/www/lunchtogether/logs/ (зокрема, nginx-логи nginx-access.log та nginx-error.log). Конфігурація logrotate [20] створюється у /etc/logrotate.d/lunchtogether та має такі параметри:

Таблиця 5.8.1

Параметри ротації логів

Параметр	Значення	Призначення
daily	—	Щоденна ротація
missingok	—	Не помилка, якщо файлу немає
rotate 14	—	Зберігати 14 ротованих копій
compress	—	Стискати старі копії gzip-ом
delaycompress	—	Не стискати найсвіжішу ротовану копію (для зручності читання)
notifempty	—	Не ротувати порожні файли
create	0640 lunchtogether lunchtogether	Створювати нові файли з правами 640
sharedscripts	—	Виконати post-скрипт один раз для всіх файлів у блоці

Такий профіль дозволяє зберігати журнал додатку та nginx-доступу за приблизно два тижні без надмірного споживання дискового простору.

5.9 Моніторинг помилок з Sentry

Помилки, що виникають у промисловому середовищі, важливо помічати та діагностувати без затримки. Для цього підключено сервіс Sentry [25], що збирає трекбеки виключень, групує їх за схожістю, надає вебзастосунок для перегляду й нотифікації при появі нових типів помилок.

Інтеграція з FastAPI. У файлі backend/app/main.py перед створенням додатку викликається sentry_sdk.init() з такими параметрами:

- dsn — береться зі змінної середовища SENTRY_DSN; якщо порожня — Sentry не ініціалізується (зручно для локальної розробки).
- environment — встановлюється у development або production залежно від ENVIRONMENT, що дозволяє у Sentry бачити окремо помилки з різних середовищ.
- traces_sample_rate — у розробці 1.0 (100% трасування), у продакшені 0.1 (10% запитів сампл). Це баланс між видимістю та накладними витратами.

Sentry SDK для Python автоматично прив'язується до FastAPI через ASGI-інтеграцію (з пакету sentry-sdk[fastapi]), перехоплюючи всі необроблені виключення з ендпоінтів, middleware та сесій бази даних.

Користь у експлуатації. Завдяки Sentry адміністратор не залежить виключно від journalctl для виявлення проблем. Будь-яке несподіване виключення фіксується разом із повним контекстом (HTTP-запит, заголовки, тіло, ідентифікатор користувача, версія додатку, версія Python, оточення), що скорочує час діагностики у багато разів. Це принципова частина забезпечення промислової якості.

5.10 CI з GitHub Actions

Безперервна інтеграція та доставка реалізована у GitHub Actions [17] двома незалежними робочими процесами у директорії .github/workflows/. Перший —

ci.yml — запускається при кожному пуші у гілки main та develop, а також при відкритті pull request до цих гілок; він перевіряє якість коду. Другий — deploy.yml — запускається при пуші у гілку main або вручну через workflow_dispatch і виконує повне розгортання на промисловому сервері через SSH; його опис наведено у розділі 5.11.

Таблиця 5.10.1

Робочі процеси CI/CD

Workflow	Тригери	Задачі (jobs)	Призначення
ci.yml	push і pull_request у main та develop	backend-lint, frontend-build	Статичний аналіз та перевірка тестів
deploy.yml	push у main та workflow_dispatch	deploy	SSH-розгортання на VPS із health-check

Таблиця 5.10.2

Кроки робочого процесу ci.yml

Задача (job)	Кроки
backend-lint	1. actions/checkout@v4 2. `setup-python@v5` з версією 3.12 3. Встановлення `uv` через офіційний скрипт 4. `uv sync --dev` (із dev-залежностями) 5. `uv run ruff check .` — статичний аналіз 6. `uv run ruff format --check .` — перевірка форматування
frontend-build	1. actions/checkout@v4 2. `setup-node@v4` з версією 20 та кешуванням npm 3. `npm ci` 4. `npm run type-check` — `tsc --noEmit` 5. `npm run build` — повне продакшен-збирання

Конвеєр CI гарантує, що жодна зміна не потрапляє у захищені гілки без проходження автоматичних перевірок. Найбільш типові помилки (неконсистентне форматування, помилки типів TypeScript, синтаксичні помилки, що пройшли локальну перевірку) виявляються до того, як вони доходять до рецензента або користувача. Перелік перевірок безпосередньо відповідає тому, що зафіксовано у файлі .github/workflows/ci.yml; повний YAML наведено у Додатку Д (Лістинг Д.2).

5.11 Скрипт розгортання deploy.sh

Власне розгортання нової версії на сервері виконується скриптом infrastructure/deploy.sh, що запускається з-під root (або через sudo) у директорії розгортання. Алгоритм скрипта:

1. Pull нової версії коду. Якщо у директорії є git-репозиторій, виконується sudo -u lunchtogether git pull.
2. Оновлення бекенду:
    - cd $APP_DIR/backend
    - sudo -u lunchtogether uv sync — синхронізація залежностей із pyproject.toml. uv автоматично створює віртуальне середовище, якщо потрібно.
    - sudo -u lunchtogether uv run alembic upgrade head — застосування непримінених міграцій.
    - systemctl restart lunchtogether-backend — перезапуск сервісу.
    - sleep 2 та перевірка systemctl is-active. Якщо сервіс не активний — вивід journalctl -u lunchtogether-backend -n 50 та exit 1 (скрипт зупиняється з помилкою).
3. Оновлення фронтенду:
    - cd $APP_DIR/frontend
    - sudo -u lunchtogether npm ci --production=false — чисте встановлення залежностей із package-lock.json.

- sudo -u lunchtogether npm run build — повне продакшен-збирання у frontend/dist/.
- chown -R lunchtogether:www-data $APP_DIR/frontend/dist та chmod -R 755 — приведення прав, що Nginx (як www-data) міг читати файли.
- nginx -t && systemctl reload nginx — перевантаження конфігурації Nginx (без зупинки), що змушує його перечитати файли SPA.

Філософія zero-downtime. Скрипт організовано так, що між зупинкою старого процесу та запуском нового відбувається мінімум часу. Nginx буферизує запити на короткий період, тому користувачі практично не помічають оновлення. Якщо новий бекенд не стартує — старий процес уже зупинений, але деталі помилки одразу виводяться в журнал, тож проблему легко локалізувати.

Автоматизація розгортання через deploy.yml. Виклик deploy.sh із локальної консолі підтримується як резервний механізм; основним каналом доставки на промисловий сервер є workflow .github/workflows/deploy.yml. Він запускається на кожен push у гілку main (а також може бути викликаний вручну через workflow_dispatch) і виконує таку послідовність дій:

1. Налаштування SSH-з'єднання. Із секрета SSH_PRIVATE_KEY створюється приватний ключ ~/.ssh/id_rsa з правами 600; із секретів SERVER_HOST і SERVER_USER витягуються адреса й користувач, перевіряється DNS-розв'язання, виконується ssh-keyscan для занесення хоста до known_hosts.
2. Розгортання серверної частини. Через ssh на віддаленому хості виконується git fetch && git reset --hard origin/main, uv sync, uv run alembic upgrade head від імені облікового запису lunchtogether, після чого systemctl restart lunchtogether-backend. Якщо systemctl is-active --quiet lunchtogether-backend повертає не «active» — виводиться останні 50 рядків journalctl і workflow завершується з помилкою.

3. Розгортання клієнтської частини. На сервері виконується npm ci --include=dev, npm run build, потім права на frontend/dist/ приводяться до lunchtogether:www-data із 755, після чого виконується nginx -t && systemctl reload nginx.
4. Health check. Із runner-а виконується curl https://$DOMAIN/api/health і перевіряється HTTP-код 200. Якщо health check не пройдено — workflow завершується з помилкою.

Для роботи deploy.yml у налаштуваннях репозиторію визначено чотири секрети: SSH_PRIVATE_KEY, SERVER_HOST, SERVER_USER, DOMAIN. Усі рядкові значення проходять tr -d '[:space:]' перед використанням, щоб уникнути проблем із прихованими символами у разі копіювання-вставки через UI GitHub. Повний YAML наведено у Додатку Д (Лістинг Д.3).

5.12 Висновки до розділу 5

У розділі описано промислове розгортання «LunchTogether» на VPS: домен і DNS, SMTP для запрошень, PostgreSQL, Nginx, HTTPS через Let's Encrypt, systemd-сервіс, резервне копіювання, logrotate, Sentry, CI та SSH-доставку через GitHub Actions. Налаштування зведені у скрипти setup.sh і deploy.sh, тому нове середовище можна відтворити з мінімальною кількістю ручних дій і з контрольованими точками перевірки.

РОЗДІЛ 6 ТЕСТУВАННЯ ТА ВАЛІДАЦІЯ СИСТЕМИ

6.1 Стратегія тестування

Для системи «LunchTogether» обрано стратегію інтеграційного тестування, що покриває основні бізнес-сценарії на рівні «API + база даних». Така стратегія є компромісом між трьома альтернативами:

- Юніт-тести — швидкі, але мокування репозиторіїв і сесій БД ховає більшість потенційних регресій, що виникають саме на стиках шарів (наприклад, у роботі SQLAlchemy із транзакціями, JSON-серіалізації Pydantic, інтеграції FastAPI Depends).
- End-to-end (e2e) тести через браузер — найповніше покривають користувацький досвід, але вимагають утримання повноцінного оточення (Selenium / Playwright + БД + бекенд + фронтенд), повільні та крихкі.
- Інтеграційні тести API + БД — дають гарне співвідношення «вартість/користь»: тестують реальні запити SQL, реальну серіалізацію JSON, перевіряють інтеграцію між шарами і при цьому виконуються швидко (десятки тестів за лічені секунди при використанні транзакційного rollback).

Підхід «test-first для критичних шляхів». Замість покриття всього коду тестами заради метрики coverage обрано підхід, де тестами покриваються бізнес-критичні шляхи: повний життєвий цикл замовлення, перерахунок балансів, перевірки дозволів, ліміти на групи та учасників, токени запрошень. Це забезпечує, що головні гарантії системи (фінансова коректність, безпека дозволів) залишаються стабільними при наступних доробках коду.

Структура викладу. Цей розділ описує програму тестування системи: інструментарій, структуру тестового середовища, перелік інтеграційних сценаріїв із чіткими передумовами та очікуваними результатами, шаблон ручних тест-кейсів та матрицю трасування між функціональними вимогами та конкретними тестовими сценаріями. У файлі backend/pyproject.toml до розділу [dependency-groups]

зафіксовано dev-залежності pytest>=8, pytest-asyncio>=0.23 та httpx>=0.27, які формують середовище виконання тестів. Тестова програма має статус робочого артефакту проєкту — кожен сценарій із таблиць §6.4 ідентифікований кодом T-<MODULE>-<N> та трасується до вихідного коду відповідного workflow або ендпоінта.

6.2 Інструменти автоматизованого тестування

Pytest 8 [26]. Стандартний інструмент тестування Python із підтримкою фікстур, параметризації, маркерів та плагінів. Тестові файли іменуються test_*.py, функції — test_*, класи — Test*. Декларативний стиль фікстур (з декоратором @pytest.fixture) дозволяє переюзні чисті налаштування.

Pytest-asyncio 0.23 [26]. Плагін, що дозволяє писати тести у форматі async def, які працюють із асинхронним кодом FastAPI / SQLAlchemy. Конфігурується режимом asyncio_mode = "auto" (усі async-функції тестів автоматично запускаються в event loop) або через декоратор @pytest.mark.asyncio для опт-ін.

httpx 0.27 [27]. Сучасний HTTP-клієнт для Python із підтримкою async. Через httpx.AsyncClient із transport=ASGITransport(app=app) тести можуть виконувати справжні HTTP-запити безпосередньо у пам'яті проти ASGI-додатку FastAPI, без запуску мережевого сервера. Це найшвидший та найточніший спосіб інтеграційного тестування.

Тестова PostgreSQL. Хоча SQLite був би швидшим, його використання у тестах розходиться з продакшен-схемою (зокрема, типи UUID, Numeric, поведінка ON DELETE CASCADE). Для уникнення розбіжностей використовується справжня PostgreSQL, або через окрему БД на тестовому сервері, або через docker-compose із pg_isready health-check. У GitHub Actions використовується service container з PostgreSQL 16 (див. розділ 5.10).

6.3 Структура тестового середовища

Тестова інфраструктура розташована у директорії backend/tests/ і складається з:

- conftest.py — кореневий файл з усіма фікстурами:
  - engine — async-двигун, прив'язаний до тестової БД (URL через змінну середовища TEST_DATABASE_URL).
  - tables — фікстура session-рівня, що створює всі таблиці на початку прогону та видаляє в кінці (через Base.metadata.create_all/drop_all).
  - db — фікстура function-рівня, що відкриває нову сесію БД для кожного тесту і відкочує всі зміни в кінці (через savepoint або повний rollback зовнішньої транзакції). Це забезпечує ізоляцію тестів.
  - client — фікстура AsyncClient з ASGITransport, із override get_db через app.dependency_overrides (щоб тести використовували db-сесію з фікстури).
  - factory_user — фікстура-фабрика, що створює користувача в БД із параметрами або значеннями за замовчуванням. Повертає об'єкт User.
  - factory_group — фабрика груп, що приймає власника як параметр.
  - factory_group_with_members — комбінована фабрика, що створює групу з заданим набором учасників та пресетами ролей.
  - auth_user_client, admin_client — варіанти client, що вже виконали логін (отримали cookie) — для зручного написання захищених тестів.
- tests/integration/ — директорія з тестовими файлами, групованими за модулями (по одному файлу на функціональну область — див. 6.4).
- tests/fixtures/ — додаткові тестові дані (зразкові ресторани, страви, патерни для параметризації).

Команда локального запуску тестового набору:

cd backend

uv run pytest -v

В CI відповідні кроки виконуються у задачі backend-lint файлу .github/workflows/ci.yml разом із PostgreSQL у вигляді service container, що піднімається на час прогону.

6.4 Програма інтеграційних тестів

Інтеграційні тести згруповано за функціональними областями. Замість повного переліку кожної тестової функції у тексті роботи наведено зведену таблицю; детальні назви сценаріїв містяться безпосередньо у файлах backend/tests/integration/test_*.py.

Таблиця 6.4.1

Зведення інтеграційних тестів

Файл тестів	Основна область перевірки	Приклади критичних сценаріїв
test_auth.py	Автентифікація	Реєстрація, логін, JWT-cookie, /me, logout
test_users.py	Користувачі	Адмінські дії, власний профіль, захист від самодеактивації
test_groups.py	Групи	Створення, ліміти, видимість груп, видалення
test_invitations.py	Запрошення	Токени, email-відправка, accept/decline, ліміт 25 учасників
test_permissions.py	Дозволи	Пресети ролей, точкова зміна дозволів, заборонені дії
test_restaurants.py	Ресторани	CRUD ресторанів і страв, улюблені позиції

Файл тестів	Основна область перевірки	Приклади критичних сценаріїв
test_order_lifecycle.py	Замовлення	Створення, допустимі й заборонені переходи, позиції, завершення
test_delivery_fee.py	Доставка	Розподіл вартості та врахування доставки у балансах
test_balances.py	Баланси	Ручні коригування, історія, lazy-створення балансу
test_analytics.py	Аналітика	Метрики групи й користувача, перевірка доступу

Найбільша увага приділена життєвому циклу замовлення, системі дозволів і балансам, оскільки саме ці частини визначають фінансову коректність та безпеку доступу. Уривки реалізації репрезентативних тестів наведено у Додатку Д.

6.5 Ручне функціональне тестування

Ручне тестування використовується для перевірки UX-сценаріїв, які недоцільно автоматизувати без повного e2e-середовища. Для кожного сценарію фіксуються ID, передумови, кроки, очікуваний результат, статус і за потреби один скріншот проблемного або ключового стану.

Приклади ручних сценаріїв:

- MT-LOGIN-1. Вхід із правильними реквізитами та перевірка домашньої навігації.
- MT-INVITE-1. Email-запрошення, перехід за токеном і приєднання до групи.
- MT-ORDER-FULL. Повний цикл замовлення до Finished із перевіркою балансів.
- MT-BALANCE-ADJ. Ручне коригування балансу та перегляд історії.
- MT-RESP-1. Перевірка адаптивної верстки на мобільній ширині.

6.6 Матриця трасування вимог

Матриця трасування пов'язує функціональні вимоги (з розділу 1.5) з конкретною реалізацією та тестами. Це інструмент валідації повноти реалізації та визначення покриття.

Таблиця 6.6.1

Скорочена матриця трасування вимог

Група вимог	Реалізація	Основні тести
FR-1.* Автентифікація	RegisterWorkflow, LoginWorkflow, get_current_user, /api/auth/*	test_auth.py
FR-2.* Дозволи	GroupMemberPermission, GROUP_ROLE_PRESETS, permission checks	test_permissions.py
FR-3.* Користувачі	api/users.py, профіль і адмінські операції	test_users.py
FR-4.* Групи	CreateGroupWorkflow, ManageMembersWorkflow, запрошення	test_groups.py, test_invitations.py
FR-5.* Ресторани	api/restaurants.py, FavoriteDishRepository	test_restaurants.py
FR-6.* Баланси	Balance, BalanceHistory, AdjustBalanceWorkflow	test_balances.py
FR-7.* Замовлення	CreateOrderWorkflow, OrderLifecycleWorkflow, api/orders.py	test_order_lifecycle.py, test_delivery_fee.py
FR-8.* Аналітика групи	api/analytics.py, активне замовлення, баланс	test_analytics.py, test_balances.py
FR-9.* Дашборд користувача	GET /api/users/me/analytics, Sidebar, settings	test_analytics.py, ручний MT-NAV-CONTEXT

Усі 37 функціональних вимог мають відповідну реалізацію та закріплений за нею автоматизований чи ручний тестовий сценарій. Це забезпечує повноту покриття системи відносно специфікації.

6.7 Керівництво користувача

Типовий шлях користувача складається з шести кроків: реєстрація або вхід, створення чи прийняття групи, запрошення учасників, налаштування ресторанів і страв, проходження життєвого циклу замовлення, перегляд або коригування балансів. Для ілюстрації цього сценарію достатньо кількох ключових екранів: дашборд групи, сторінка учасників, створення замовлення, деталі активного замовлення, сторінка балансів і прийняття запрошення.

Користувач повинен враховувати основні обмеження системи: одна активна заявка на групу, максимум 5 груп на користувача, максимум 25 учасників у групі, розмір логотипу до 10 МБ і завершення сесії після закінчення строку JWT. При порушенні бізнес-правил API повертає пояснювальне повідомлення, а фронтенд показує його у формі inline-помилки, alert або toast-сповіщення.

6.8 Висновки до розділу 6

У розділі описано програму контролю якості системи: інтеграційне тестування рівня API + БД, структуру тестового середовища, зведення тестів за функціональними областями, ручні UX-сценарії та матрицю трасування вимог. Покриття зосереджене на критичних для системи гарантіях: автентифікації, дозволах, життєвому циклі замовлення, перерахунку балансів і аналітиці.

ВИСНОВКИ

У межах дипломної роботи розроблено веб-додаток «LunchTogether» для організації спільних обідів у колективах. Робота охопила аналіз предметної області, постановку вимог, архітектурне проєктування, реалізацію серверної та клієнтської частин, розгортання на VPS і програму тестування.

Система реалізована на FastAPI, PostgreSQL, SQLAlchemy, React, TypeScript і Vite. Архітектура поділена на API, Workflow, Repository та Models, що спрощує підтримку бізнес-логіки та тестування. Ключовими інженерними рішеннями стали дворівнева система прав, життєвий цикл замовлення з контрольованими переходами станів, автоматичний перерахунок балансів і повторне використання каталогу ресторанів та страв.

Клієнтська частина побудована як SPA з функціональними модулями, RTK Query для серверного стану, захищеними маршрутами, двопанельною навігацією, адаптивним інтерфейсом і типовими станами Loading/Error/Empty. Інфраструктура включає Nginx, HTTPS через Let's Encrypt, systemd, резервне копіювання, logrotate, Sentry та CI/CD на GitHub Actions.

Практичний результат підтверджено інтеграційними тестами за основними функціональними областями та ручними UX-сценаріями. Створена система є завершеним full-stack рішенням, яке можна розгорнути у реальному середовищі та використовувати як внутрішній інструмент для групової координації обідів.

Основні характеристики продукту: 12 доменних моделей, 11 Workflow-класів, близько 45 REST-ендпоінтів, 8 клієнтських модулів, 5 типів групових дозволів, 3 пресети ролей, автоматизоване розгортання та тестова програма, прив'язана до 37 функціональних вимог.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. FastAPI Documentation. URL: https://fastapi.tiangolo.com/ (дата звернення: 10.05.2026).
2. React Documentation. URL: https://react.dev/ (дата звернення: 10.05.2026).
3. shadcn/ui Documentation. URL: https://ui.shadcn.com/docs (дата звернення: 10.05.2026).
4. SQLAlchemy 2.0 Documentation. URL: https://docs.sqlalchemy.org/en/20/ (дата звернення: 10.05.2026).
5. PostgreSQL 16 Documentation. URL: https://www.postgresql.org/docs/16/ (дата звернення: 10.05.2026).
6. Redux Toolkit and RTK Query Documentation. URL: https://redux-toolkit.js.org/ (дата звернення: 10.05.2026).
7. Alembic Database Migration Tool. URL: https://alembic.sqlalchemy.org/en/latest/ (дата звернення: 10.05.2026).
8. Pydantic v2 Documentation. URL: https://docs.pydantic.dev/latest/ (дата звернення: 10.05.2026).
9. Tailwind CSS v4 Documentation. URL: https://tailwindcss.com/docs (дата звернення: 10.05.2026).
10. React Router v7 Documentation. URL: https://reactrouter.com/ (дата звернення: 10.05.2026).
11. Let's Encrypt — Free SSL/TLS Certificates. URL: https://letsencrypt.org/ (дата звернення: 10.05.2026).
12. Uvicorn — ASGI server implementation. URL: https://www.uvicorn.org/ (дата звернення: 10.05.2026).
13. Vite — Next Generation Frontend Tooling. URL: https://vitejs.dev/ (дата звернення: 10.05.2026).
14. Radix UI Primitives. URL: https://www.radix-ui.com/primitives (дата звернення: 10.05.2026).

15. Lucide — beautiful & consistent icons. URL: https://lucide.dev/ (дата звернення: 10.05.2026).
16. Nginx Documentation. URL: https://nginx.org/en/docs/ (дата звернення: 10.05.2026).
17. GitHub Actions Documentation. URL: https://docs.github.com/en/actions (дата звернення: 10.05.2026).
18. systemd.service — Service unit configuration. URL: https://www.freedesktop.org/software/systemd/man/systemd.service.html (дата звернення: 10.05.2026).
19. Ubuntu Server 22.04 LTS Documentation. URL: https://ubuntu.com/server/docs (дата звернення: 10.05.2026).
20. logrotate(8) — Linux man page. URL: https://man7.org/linux/man-pages/man8/logrotate.8.html (дата звернення: 10.05.2026).
21. Certbot Documentation. URL: https://eff-certbot.readthedocs.io/en/stable/ (дата звернення: 10.05.2026).
22. uv — A fast Python package and project manager (Astral). URL: https://docs.astral.sh/uv/ (дата звернення: 10.05.2026).
23. Ruff — extremely fast Python linter and code formatter. URL: https://docs.astral.sh/ruff/ (дата звернення: 10.05.2026).
24. TypeScript Handbook. URL: https://www.typescriptlang.org/docs/handbook/ (дата звернення: 10.05.2026).
25. Sentry SDK for Python. URL: https://docs.sentry.io/platforms/python/ (дата звернення: 10.05.2026).
26. Pytest Documentation. URL: https://docs.pytest.org/en/stable/ (дата звернення: 10.05.2026).
27. HTTPX — A next generation HTTP client for Python. URL: https://www.python-httpx.org/ (дата звернення: 10.05.2026).

28. pg_dump — PostgreSQL backup utility. URL: https://www.postgresql.org/docs/16/app-pgdump.html (дата звернення: 10.05.2026).
29. bcrypt Documentation. URL: https://pypi.org/project/bcrypt/ (дата звернення: 10.05.2026).
30. python-jose — JSON Object Signing and Encryption for Python. URL: https://python-jose.readthedocs.io/en/latest/ (дата звернення: 10.05.2026).
31. Закон України «Про захист персональних даних» від 01.06.2010 № 2297-VI (зі змінами). URL: https://zakon.rada.gov.ua/laws/show/2297-17 (дата звернення: 10.05.2026).
32. OMG Unified Modeling Language (OMG UML) Specification, version 2.5.1. Object Management Group, 2017. URL: https://www.omg.org/spec/UML/2.5.1/ (дата звернення: 10.05.2026).
33. Web Content Accessibility Guidelines (WCAG) 2.1. W3C Recommendation, 2018. URL: https://www.w3.org/TR/WCAG21/ (дата звернення: 10.05.2026).
34. Namecheap — Domain Registration and Private Email Service. URL: https://www.namecheap.com/ (дата звернення: 12.05.2026).

ДОДАТОК А

У цьому додатку наведено уривки реалізації двох ключових моделей реляційної схеми — Order і OrderItem — та фрагмент моделі User із обчислюваним полем is_admin. Повний код знаходиться у директорії backend/app/models/.

Лістинг А.1. Модель Order (backend/app/models/order.py).

```
class Order(BaseModel):
    __tablename__ = "orders"

    group_id: Mapped[uuid.UUID] = mapped_column(
        UUID(as_uuid=True),
        ForeignKey("groups.id", ondelete="CASCADE"),
        nullable=False,
    )
    restaurant_id: Mapped[uuid.UUID | None] = mapped_column(
        UUID(as_uuid=True),
        ForeignKey("restaurants.id", ondelete="SET NULL"),
        nullable=True,
    )
    restaurant_name: Mapped[str | None] = mapped_column(
        String(255),
        nullable=True,
    )
    initiator_id: Mapped[uuid.UUID] = mapped_column(
        UUID(as_uuid=True),
        ForeignKey("users.id", ondelete="CASCADE"),
        nullable=False,
    )
    status: Mapped[str] = mapped_column(
        String(20),
        default=OrderStatus.INITIATED,
        nullable=False,
    )
    delivery_fee_total: Mapped[Decimal | None] = mapped_column(
```

```
        Numeric(10, 2),
        nullable=True,
    )
    delivery_fee_per_person: Mapped[Decimal | None] =
mapped_column(
        Numeric(10, 2),
        nullable=True,
    )

    # Relationships
    group: Mapped["Group"] = relationship(  # noqa: F821
        "Group",
        back_populates="orders",
    )
    restaurant: Mapped["Restaurant | None"] = relationship(  #
noqa: F821
        "Restaurant",
    )
    initiator: Mapped["User"] = relationship(  # noqa: F821
        "User",
        foreign_keys=[initiator_id],
    )
    items: Mapped[list["OrderItem"]] = relationship(
        "OrderItem",
        back_populates="order",
        cascade="all, delete-orphan",
    )
```

Лістинг А.2. Модель OrderItem (там же).

```
class OrderItem(BaseModel):
    __tablename__ = "order_items"

    order_id: Mapped[uuid.UUID] = mapped_column(
        UUID(as_uuid=True),
        ForeignKey("orders.id", ondelete="CASCADE"),
        nullable=False,
```

```
    )
    user_id: Mapped[uuid.UUID] = mapped_column(
        UUID(as_uuid=True),
        ForeignKey("users.id", ondelete="CASCADE"),
        nullable=False,
    )
    name: Mapped[str] = mapped_column(
        String(255),
        nullable=False,
    )
    detail: Mapped[str | None] = mapped_column(
        String(500),
        nullable=True,
    )
    price: Mapped[Decimal] = mapped_column(
        Numeric(10, 2),
        nullable=False,
    )
    dish_id: Mapped[uuid.UUID | None] = mapped_column(
        UUID(as_uuid=True),
        ForeignKey("dishes.id", ondelete="SET NULL"),
        nullable=True,
    )
    quantity: Mapped[int] = mapped_column(
        Integer,
        default=1,
        nullable=False,
    )

    # Relationships
    order: Mapped["Order"] = relationship(
        "Order",
        back_populates="items",
    )
    user: Mapped["User"] = relationship(  # noqa: F821
        "User",
```

```
    )
    dish: Mapped["Dish | None"] = relationship(  # noqa: F821
        "Dish",
    )
```

Лістинг А.3. Фабрика додатку та реєстрація middleware (backend/app/main.py, уривок).

```
def create_app() -> FastAPI:
    app = FastAPI(
        title="LunchTogether API",
        description="API for organizing regular lunches with
colleagues",
        version="0.1.0",
        docs_url="/api/docs" if settings.is_development else
None,
        redoc_url="/api/redoc" if settings.is_development else
None,
    )

    # Custom middleware (added first = innermost, runs after
CORS)
    app.add_middleware(RequestLoggingMiddleware)
    app.add_middleware(ErrorHandlingMiddleware)

    # CORS (added last = outermost, intercepts preflight OPTIONS
before anything else)
    app.add_middleware(
        CORSMiddleware,
        allow_origins=settings.cors_origins,
        allow_credentials=True,
        allow_methods=["*"],
        allow_headers=["*"],
    )

    # Include routers
```

```
    app.include_router(api_router)

    # Health check
    @app.get("/api/health")
    async def health_check() -> dict[str, str]:
        return {"status": "ok"}

    return app
```

ДОДАТОК Б

У цьому додатку наведено реалізацію TransitionOrderWorkflow (backend/app/workflows/order/transition.py), що інкапсулює переходи станів замовлення та автоматичний перерахунок балансів учасників.

Лістинг Б.1. Реалізація TransitionOrderWorkflow

```
VALID_TRANSITIONS = {
    OrderStatus.INITIATED: [OrderStatus.CONFIRMED,
OrderStatus.CANCELLED],
    OrderStatus.CONFIRMED: [OrderStatus.ORDERED,
OrderStatus.CANCELLED],
    OrderStatus.ORDERED: [OrderStatus.FINISHED,
OrderStatus.CANCELLED],
}


class TransitionOrderInput(BaseModel):
    model_config = ConfigDict(arbitrary_types_allowed=True)

    order_id: uuid.UUID
    new_status: str
    current_user: User


class TransitionOrderOutput(BaseModel):
    order: OrderResponse


class TransitionOrderWorkflow:
    def __init__(
        self,
        order_repository: OrderRepository,
        order_item_repository: OrderItemRepository,
        group_member_repository: GroupMemberRepository,
```

```
        balance_repository: BalanceRepository,
        balance_history_repository: BalanceHistoryRepository,
        dish_repository: DishRepository,
    ):
        self.order_repository = order_repository
        self.order_item_repository = order_item_repository
        self.group_member_repository = group_member_repository
        self.balance_repository = balance_repository
        self.balance_history_repository =
balance_history_repository
        self.dish_repository = dish_repository

    async def execute(self, input_data: TransitionOrderInput) ->
TransitionOrderOutput:
        user = input_data.current_user

        order = await
self.order_repository.get_by_id(input_data.order_id)
        if order is None:
            raise NotFoundError(detail="Order not found")

        membership = await
self.group_member_repository.get_membership(user.id,
order.group_id)
        is_initiator = order.initiator_id == user.id
        is_editor = membership and
membership.get_permission(PermissionType.ORDERS) ==
OrdersScope.EDITOR

        if not is_initiator and not is_editor and not
user.is_admin:
            raise ForbiddenError(detail="Only the order
initiator or an editor can change order status")

        current_status = OrderStatus(order.status)
        new_status = OrderStatus(input_data.new_status)
```

```
        allowed = VALID_TRANSITIONS.get(current_status, [])
        if new_status not in allowed:
            raise ValidationError(detail=f"Cannot transition
from {current_status.value} to {new_status.value}")

        if new_status == OrderStatus.FINISHED:
            await self._handle_finish(order)

        updated = await self.order_repository.update(
            order.id,
OrderStatusInternalUpdate(status=new_status.value)
        )

        return
TransitionOrderOutput(order=OrderResponse.model_validate(updated
))

    async def _handle_finish(self, order) -> None:
        items = await
self.order_item_repository.get_items_for_order(order.id)
        if not items:
            return

        user_totals: dict[uuid.UUID, Decimal] = {}
        for item in items:
            user_totals.setdefault(item.user_id,
Decimal("0.00"))
            user_totals[item.user_id] += item.price *
(item.quantity or 1)

        if order.delivery_fee_per_person:
            for uid in user_totals:
                user_totals[uid] +=
order.delivery_fee_per_person
```

```
        for uid, total in user_totals.items():
            balance = await
self.balance_repository.get_by_user_and_group(uid,
order.group_id)
            if balance is None:
                balance = await self.balance_repository.create(
                    BalanceInternalCreate(user_id=uid,
group_id=order.group_id)
                )
            new_amount = balance.amount - total
            await self.balance_repository.update(balance.id,
BalanceAmountUpdate(amount=new_amount))

            await self.balance_history_repository.create(
                BalanceHistoryInternalCreate(
                    balance_id=balance.id,
                    amount=-total,
                    balance_after=new_amount,
                    note=f"Order #{str(order.id)[:8]}",
                    change_type=BalanceChangeType.ORDER,
                    order_id=order.id,
                )
            )

        if order.restaurant_id:
            for item in items:
                existing_dish = await
self.dish_repository.get_by_name_and_restaurant(item.name,
order.restaurant_id)
                if existing_dish:
                    if existing_dish.price != item.price:
                        from app.schemas.restaurant import
DishUpdate

                        await
self.dish_repository.update(existing_dish.id,
```

```
DishUpdate(price=item.price))
                else:
                    await self.dish_repository.create(
                        DishInternalCreate(
                            name=item.name,
                            detail=item.detail,
                            price=item.price,
                            restaurant_id=order.restaurant_id,
                        )
                    )
```

ДОДАТОК В

Лістинг В.1. Шаблон конфігурації Nginx (infrastructure/nginx/lunchtogether.conf).

```
# Backend upstream
upstream backend {
    server 127.0.0.1:8000;
}

# HTTP to HTTPS redirect
server {
    listen 80;
    listen [::]:80;
    server_name DOMAIN_PLACEHOLDER www.DOMAIN_PLACEHOLDER;

    location /.well-known/acme-challenge/ {
        root /var/www/certbot;
    }

    location / {
        return 301 https://$host$request_uri;
    }
}

# HTTPS server
server {
    listen 443 ssl http2;
    listen [::]:443 ssl http2;
    server_name DOMAIN_PLACEHOLDER www.DOMAIN_PLACEHOLDER;

    # SSL configuration (will be managed by Certbot)
    ssl_certificate
/etc/letsencrypt/live/DOMAIN_PLACEHOLDER/fullchain.pem;
    ssl_certificate_key
/etc/letsencrypt/live/DOMAIN_PLACEHOLDER/privkey.pem;
```

```
    # SSL security settings
    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_ciphers HIGH:!aNULL:!MD5;
    ssl_prefer_server_ciphers on;

    # Security headers
    add_header X-Frame-Options "SAMEORIGIN" always;
    add_header X-Content-Type-Options "nosniff" always;
    add_header X-XSS-Protection "1; mode=block" always;
    add_header Strict-Transport-Security "max-age=31536000;
includeSubDomains" always;

    # Logging
    access_log APP_DIR_PLACEHOLDER/logs/nginx-access.log;
    error_log APP_DIR_PLACEHOLDER/logs/nginx-error.log;

    # Max upload size
    client_max_body_size 10M;

    # Backend API (includes /api/health, /api/docs in dev, and
all api_router routes)
    location /api/ {
        proxy_pass http://backend;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For
$proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_cache_bypass $http_upgrade;
        proxy_read_timeout 300s;
        proxy_connect_timeout 75s;
    }
```

```
    # Uploaded files
    location /uploads/ {
        alias APP_DIR_PLACEHOLDER/uploads/;
        expires 1y;
        add_header Cache-Control "public, immutable";
    }

    # Frontend static files
    location / {
        root APP_DIR_PLACEHOLDER/frontend/dist;
        try_files $uri $uri/ /index.html;

        # Cache static assets
        location ~*
\.(js|css|png|jpg|jpeg|gif|ico|svg|woff|woff2|ttf|eot)$ {
            expires 1y;
            add_header Cache-Control "public, immutable";
        }
    }
}
```

Лістинг B.2. Systemd-юніт бекенду (infrastructure/systemd/lunchtogether-backend.service).

```
[Unit]
Description=LunchTogether Backend Service
After=network.target postgresql.service
Requires=postgresql.service

[Service]
Type=simple
User=APP_USER_PLACEHOLDER
Group=www-data
WorkingDirectory=APP_DIR_PLACEHOLDER/backend
EnvironmentFile=APP_DIR_PLACEHOLDER/backend/.env
```

```
# Start the application
# Note: database migrations are run by the deploy pipeline
(deploy.sh / deploy.yml),
# not as ExecStartPre, so the service can still (re)start even
if the DB is briefly
# unavailable at boot.
ExecStart=/home/APP_USER_PLACEHOLDER/.local/bin/uv run uvicorn
app.main:app --host 127.0.0.1 --port 8000 --workers 4

# Restart policy
Restart=always
RestartSec=10

# Security
NoNewPrivileges=true
PrivateTmp=true

# Logging
StandardOutput=journal
StandardError=journal
SyslogIdentifier=lunchtogether-backend

[Install]
WantedBy=multi-user.target
```

ДОДАТОК Г

Повний перелік маршрутів реалізовано у директорії backend/app/api/. У тексті додатку залишено згрупований вигляд, достатній для розуміння структури API; детальний контракт доступний через OpenAPI-документацію FastAPI у середовищі розробки.

Таблиця Г.1

Групи REST-ендпоінтів

Група	Префікс	Основні операції	Доступ
Auth	/api/auth	register, login, logout, me	публічний / AUTH
Users	/api/users	список, профіль, роль, активність, аналітика	AUTH / ADMIN
Groups	/api/groups	CRUD груп, учасники, дозволи	AUTH / MEMBER / OWNER
Invitations	/api/groups/.../invitations	створення, список, accept, decline, cancel	AUTH / EDITOR(members)
Restaurants	/api/groups/{group_id}/restaurants	CRUD ресторанів і страв	MEMBER / EDITOR(restaurants)
Orders	/api/groups/{group_id}/orders	CRUD замовлень, статуси, доставка, позиції, favorites	MEMBER / INITIATOR / EDITOR(orders)
Balances	/api/groups/{group_id}/balances	список балансів, коригування, історія	VIEWER / EDITOR(balances)
Analytics	/api/groups/{group_id}/analytics, /api/users/me/analytics	аналітика групи та користувача	AUTH / VIEWER(analytics)

Група	Префікс	Основні операції	Доступ
Health	/api/health	перевірка доступності сервісу	публічний

Усі маршрути, що належать групі, мають group_id у шляху та перевіряють членство або адміністративний доступ. Для помилок використовується єдина JSON-структура з полем detail і стандартними HTTP-кодами 401, 403, 404, 409 та 422.

ДОДАТОК Д

Лістинг Д.1. Приклад інтеграційного тесту з тестового набору (tests/integration/test_order_lifecycle.py, скорочено).

```
class TestFinishBalanceSideEffects:
    async def test_finishing_order_with_items_debits_balance(
        self, client: AsyncClient, factory_user, factory_group,
factory_order, auth_client, db
    ):
        owner = await
factory_user(email="debit_own@example.com")
        member = await
factory_user(email="debit_mem@example.com")
        group = await factory_group(owner)
        from app.core.permissions import GROUP_ROLE_PRESETS
        from app.models.enums import GroupRole
        from app.repositories.group import
GroupMemberPermissionRepository, GroupMemberRepository
        from app.schemas.internal import
GroupMemberInternalCreate

        member_repo = GroupMemberRepository(db)
        perm_repo = GroupMemberPermissionRepository(db)
        m = await
member_repo.create(GroupMemberInternalCreate(user_id=member.id,
group_id=group.id))
        presets = GROUP_ROLE_PRESETS[GroupRole.MEMBER]
        await perm_repo.set_permissions(m.id, {pt.value: level
for pt, level in presets.items()})
        await db.commit()

        order = await factory_order(group, owner, items=[(owner,
"Pizza", Decimal("10.00"))])
        ac = await auth_client(owner)
        await _transition(ac, group.id, order.id, "confirmed")
```

```
        await _transition(ac, group.id, order.id, "ordered")
        resp = await _transition(ac, group.id, order.id,
"finished")
        assert resp.status_code == 200

        # Check balance was debited for owner
        bal_repo = BalanceRepository(db)
        balance = await bal_repo.get_by_user_and_group(owner.id,
group.id)
        assert balance is not None
        assert balance.amount == Decimal("-10.00")

        # Check history record
        hist_repo = BalanceHistoryRepository(db)
        histories = await
hist_repo.get_history_for_balance(balance.id)
        assert any(h.change_type == "order" for h in histories)

    async def
test_finishing_order_with_no_items_leaves_balances_unchanged(
        self, client: AsyncClient, factory_user, factory_group,
factory_order, auth_client, db
    ):
        owner = await factory_user(email="noop_own@example.com")
        group = await factory_group(owner)
        order = await factory_order(group, owner, items=None)
        ac = await auth_client(owner)
        await _transition(ac, group.id, order.id, "confirmed")
        await _transition(ac, group.id, order.id, "ordered")
        resp = await _transition(ac, group.id, order.id,
"finished")
        assert resp.status_code == 200

        # No balance record should exist
        from sqlalchemy import select
```

```
        from app.models.balance import Balance

        result = await db.execute(
            select(Balance).where(Balance.user_id == owner.id,
Balance.group_id == group.id)
        )
        balance = result.scalars().first()
        assert balance is None
```

Лістинг Д.2. Робочий процес CI (.github/workflows/ci.yml).

```yaml
name: CI

on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main, develop ]

jobs:
  backend-test:
    name: Backend Tests
    runs-on: ubuntu-latest

    services:
      postgres:
        image: postgres:16
        env:
          POSTGRES_USER: postgres
          POSTGRES_PASSWORD: postgres
          POSTGRES_DB: test
        ports:
          - 5432:5432
        options: >-
          --health-cmd pg_isready
          --health-interval 5s
```

```
          --health-timeout 5s
          --health-retries 10

    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.12'

      - name: Install uv
        run: curl -LsSf https://astral.sh/uv/install.sh | sh

      - name: Install dependencies (incl. dev group)
        working-directory: ./backend
        run: |
          export PATH="$HOME/.local/bin:$PATH"
          uv sync --dev

      - name: Run tests
        working-directory: ./backend
        run: |
          export PATH="$HOME/.local/bin:$PATH"
          uv run pytest -v
        env:
          TEST_DATABASE_URL:
postgresql+asyncpg://postgres:postgres@localhost:5432/test
          DATABASE_URL:
postgresql+asyncpg://postgres:postgres@localhost:5432/test
          JWT_SECRET_KEY: test-secret
          UPLOAD_DIR: /tmp/uploads
          ENVIRONMENT: test

  backend-lint:
    name: Backend Lint
```

```
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.12'

      - name: Install uv
        run: curl -LsSf https://astral.sh/uv/install.sh | sh

      - name: Install dependencies (incl. dev group)
        working-directory: ./backend
        run: |
          export PATH="$HOME/.local/bin:$PATH"
          uv sync --dev

      - name: Run ruff linting
        working-directory: ./backend
        run: |
          export PATH="$HOME/.local/bin:$PATH"
          uv run ruff check .

      - name: Run ruff formatting check
        working-directory: ./backend
        run: |
          export PATH="$HOME/.local/bin:$PATH"
          uv run ruff format --check .

  frontend-build:
    name: Frontend Build & Type Check
    runs-on: ubuntu-latest

    steps:
```

```
      - uses: actions/checkout@v4

      - name: Set up Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'
          cache-dependency-path: ./frontend/package-lock.json

      - name: Install dependencies
        working-directory: ./frontend
        run: npm ci

      - name: TypeScript check
        working-directory: ./frontend
        run: npm run type-check

      - name: Build
        working-directory: ./frontend
        run: npm run build

      # ESLint and test steps intentionally omitted: no ESLint
config or test
      # runner is set up yet. Re-enable after adding
eslint.config.js and a
      # test framework (e.g. vitest):
      # - run: npm run lint
      # - run: npm run test
```

Лістинг Д.3. Робочий процес деплоя (.github/workflows/deploy.yml).

```
name: Deploy

on:
  push:
    branches: [ main ]
```

```
  workflow_dispatch:

jobs:
  deploy:
    name: Deploy to Production
    runs-on: ubuntu-latest
    environment: production

    steps:
      - uses: actions/checkout@v4

      - name: Setup SSH
        env:
          SSH_PRIVATE_KEY: ${{ secrets.SSH_PRIVATE_KEY }}
          SERVER_HOST: ${{ secrets.SERVER_HOST }}
        run: |
          mkdir -p ~/.ssh
          printf '%s\n' "$SSH_PRIVATE_KEY" > ~/.ssh/id_rsa
          chmod 600 ~/.ssh/id_rsa

          # Trim whitespace/newlines the secret picker might
have let through.
          SERVER_HOST="$(printf '%s' "$SERVER_HOST" | tr -d
'[:space:]')"
          if [ -z "$SERVER_HOST" ]; then
            echo "ERROR: SERVER_HOST secret is empty" >&2
            exit 1
          fi

          # Verify it resolves before ssh-keyscan complains
cryptically.
          if ! getent hosts "$SERVER_HOST" >/dev/null; then
            echo "ERROR: SERVER_HOST does not resolve.
Length=${#SERVER_HOST}." >&2
            echo "Make sure the secret is a bare hostname or IP
(no http://, no /, no port)." >&2
```

```
            exit 1
          fi

          ssh-keyscan -H "$SERVER_HOST" >> ~/.ssh/known_hosts

      - name: Deploy Application
        env:
          SERVER_USER: ${{ secrets.SERVER_USER }}
          SERVER_HOST: ${{ secrets.SERVER_HOST }}
        run: |
          SERVER_HOST="$(printf '%s' "$SERVER_HOST" | tr -d
'[:space:]')"
          SERVER_USER="$(printf '%s' "$SERVER_USER" | tr -d
'[:space:]')"
          ssh -o BatchMode=yes "$SERVER_USER@$SERVER_HOST" <<
'ENDSSH'
            set -e

            # Navigate to application directory
            cd /var/www/lunchtogether

            echo "Pulling latest code..."
            sudo -u lunchtogether git fetch origin
            sudo -u lunchtogether git reset --hard origin/main

            echo "Deploying backend..."
            cd backend
            sudo -u lunchtogether
/home/lunchtogether/.local/bin/uv sync
            sudo -u lunchtogether
/home/lunchtogether/.local/bin/uv run alembic upgrade head
            sudo systemctl restart lunchtogether-backend

            # Wait and check if backend started successfully
            sleep 3
            if ! systemctl is-active --quiet
```

```
lunchtogether-backend; then
              echo "Backend failed to start!"
              sudo journalctl -u lunchtogether-backend -n 50
              exit 1
            fi

            echo "Backend deployed successfully"

            echo "Deploying frontend..."
            cd ../frontend
            sudo -u lunchtogether npm ci --include=dev
            sudo -u lunchtogether npm run build

            # Make sure nginx (running as www-data) can traverse
the dist tree.
            sudo chown -R lunchtogether:www-data dist
            sudo chmod -R 755 dist

            echo "Reloading nginx..."
            sudo nginx -t
            sudo systemctl reload nginx

            echo "Frontend deployed successfully"
          ENDSSH

      - name: Health Check
        env:
          DOMAIN: ${{ secrets.DOMAIN }}
        run: |
          sleep 5
          DOMAIN="$(printf '%s' "$DOMAIN" | tr -d '[:space:]')"
          response=$(curl -s -o /dev/null -w "%{http_code}"
"https://$DOMAIN/api/health" || true)
          if [ "$response" = "200" ]; then
            echo "Health check passed"
          else
```

```
          echo "Health check failed (HTTP $response)"
          exit 1
        fi

    - name: Notify Deployment
      if: always()
      run: |
        if [ "${{ job.status }}" = "success" ]; then
          echo "Deployment successful!"
        else
          echo "Deployment failed!"
        fi
```