МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю. М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

перший (бакалаврський)
(рівень вищої освіти)

на тему **Розробка Веб-застосунку для замовлення їжі з використанням фреймворка Next.js**

Виконав: студент 4 курсу, групи 6.1212-пзс
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Програмне забезпечення систем
(код і назва освітньої програми)
А.О. Павленко
(ініціали та прізвище)
Керівник доцент, к.ф.-м.н. Г.П. Коломоєць
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ «Дісітел»
П.О. Лютий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ім. Ю. М. Потебні ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра електроніки, інформаційних систем та програмного забезпечення
Рівень вищої освіти перший (бакалаврський)
Спеціальність 121 Інженерія програмного забезпечення
(код та назва)
Освітня програма Програмне забезпечення систем
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри Тетяна КРИТСЬКА

“ 01 ” березня 2026 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Павленко Анастасії Олександрівній
(прізвище, ім’я, по батькові)

1. Тема роботи Розробка Веб-застосунку для замовлення їжі з використанням фреймворка Next.js

керівник роботи Коломоєць Геннадій Павлович, к.ф.-м.н., доцент
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджена наказом ЗНУ від 23.12.2025 р. № 1830-с

2. Строк подання студентом кваліфікаційної роботи 02.06.2026
3. Вихідні дані бакалаврської роботи
   - комплект нормативних документів ;
   - технічне завдання до роботи.

4. Зміст розрахунково - пояснювальної записки (перелік питань, які потрібно розробити)
   - Огляд та збір літератури стосовно теми кваліфікаційної роботи;
   - Огляд та аналіз існуючих рішень та аналогів;
   - Аналіз сучасних технологій для створення Веб-застосунків;
   - Проєктування та розробка Веб-застосунку.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень) слайдів презентації6.

Консультанти розділів бакалаврської роботи

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

7. Дата видачі завдання 01.03.2026

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів бакалаврської роботи	Примітка
1	Аналіз предметної області	02.03.2026	виконано
2	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником	09.03.2026	виконано
3	Аналіз існуючих методів рішення	15.03.2026	виконано
4	Огляд та збір літератури стосовно теми кваліфікаційної роботи	20.03.2026	виконано
5	Огляд та аналіз існуючих рішень та аналогів	08.04.2026	виконано
6	Вивчення засобів для розробки Веб-застосунку	12.04.2026	виконано
7	Визначення функціоналу застосунку, що розробляється	20.04.2026	виконано
8	Узгодження подальших дій з науковим керівником	21.04.2026	виконано
9	Проєктування бази даних та створення проєкту застосунку	27.04.2026	виконано
10	Проєктування та конструювання застосунку	11.05.2026	виконано
11	Програмна реалізація застосунку та його апробація	25.05.2026	виконано
12	Оформлення звіту	02.06.2026	виконано
13	Оформлення презентації. Отримання рецензій від опонентів	10.06.2026	виконано

Студент _____ _____А.О. Павленко

Керівник роботи _____ _____Г. П. Коломоєць

Нормоконтроль пройдено

Нормоконтролер _____ _____І.А. Скрипник

АНОТАЦІЯ

Сторінок – 105

Рисунків – 29

Джерел – 10

Павленко А. О. Розробка Веб-застосунку для замовлення їжі з використанням фреймворка Next.js: кваліфікаційна робота бакалавра спеціальності 121 Інженерія програмного забезпечення / наук. керівник Г. П. Коломоєць. Запоріжжя : ЗНУ, 2026, 96 с.

У сучасному світі процеси цифровізації охоплюють дедалі більше сфер суспільної діяльності, і сфера організованого колективного харчування не є винятком. Для ефективної координації процесів у сфері шкільного харчування необхідна прозора, гнучка та швидка взаємодія між замовниками (навчальними закладами) та підприємствами-постачальниками готової кулінарної продукції. Традиційні неавтоматизовані методи (паперовий облік, телефонні дзвінки та чати у месенджерах) призводять до виникнення помилок через людський фактор, касових розривів та затримок у логістиці, що є критичним для установ Запорізької області, які функціонують в умовах воєнного стану та підземних шкіл.

Мета кваліфікаційної роботи – створення повностекового вебзастосунку, який автоматизує процес формування, обробки й контролю замовлень харчування, забезпечує централізоване зберігання даних та гнучке відстеження статусів виконання і фінансових розрахунків між контрагентами в межах рольової B2B-моделі.

У ході розробки програмного забезпечення було проведено аналіз предметної області та існуючих аналогів на ринку FoodTech. Було визначено детальні функціональні й нефункціональні вимоги до системи.

Ключові слова: *шкільне харчування, вебзастосунок, B2B-взаємодія, frontend, backend, база даних, Next.js, React 19, Prisma ORM, PostgreSQL*.

ABSTRACT

Pages – 105

Drawings – 29

Sources – 10

Pavlenko A. O. Development of a Web Application for Food Ordering Using the Next.js Framework: qualification work of a bachelor in the specialty 121 “Software Engineering” / scientific supervisor H. P. Kolomoicts. Zaporizhzhia : ZNU, 2026, 105 p.

In the modern world, digitalization processes cover more and more areas of social activity, and the sphere of organized school catering is no exception. For effective coordination of educational catering processes, a transparent, flexible, and fast interaction between customers (educational institutions) and suppliers of ready-made culinary products is required. Traditional non-automated methods (paper records, phone calls, and chats in messengers) lead to errors due to the human factor, payment desynchronization, and logistical delays, which is critical for institutions in the Zaporizhzhia region operating under martial law and within underground schools.

The goal of this project is to create a full-stack web application that automates the process of creating, processing, and monitoring food orders, providing centralized data storage and flexible tracking of order execution and payment statuses between counterparties within a role-based B2B model.

During the development of the application, an analysis of the subject area and existing solutions in the FoodTech market was conducted. Detailed functional and non-functional requirements for the system were identified. As a result, a web application was developed with a clear modular. A reliable relational database.

Keywords: *school catering, web application, B2B interaction, frontend, backend, database, Next.js, React 19, Prisma ORM, PostgreSQL*.

ЗМІСТ

ВСТУП 7
1.1 Особливості організації процесу замовлення їжі 11
1.2 Аналіз проблем та недоліків неавтоматизованих систем обліку 14
1.3 Аналіз існуючих рішень 17
1.4 Постановка задачі 21
1.5 Висновки до розділу 1 23
2 ОПИС ТА ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАСТОСУНКУ 25
2.1 Архітектура веб-застосунку 25
2.2 Дизайн інтерфейсу користувачів 28
2.3 Нефункціональні вимоги 29
2.4 Функціональність застосунку 31
2.4.1 Функціональність для шкіл 34
2.4.2 Функціональність для постачальників 35
2.4.3 Функціональність для адміністратора 36
2.5 Огляд систем управління базами даних 38
2.6 Огляд інструментів серверної розробки 41
2.7 Огляд інструментів розробки інтерфейсу користувача 44
2.8 Огляд бібліотек та фреймворків для стилізації інтерфейсу користувача 47
2.9 Висновки з розділу 2 50
3 СТРУКТУРА ПРОЄКТУ ВЕБ-ЗАСТОСУНКУ ДЛЯ ЗАМОВЛЕННЯ ЇЖІ 52
3.1 Структура бази даних проєкту 52
3.2 Клієнтська частина проєкту 60
3.3 Серверна частина проєкту 66
3.4 Використання застосунку 88
3.4.1 Функціонал адміністратора 90
3.4.2 Функціонал працівника школи 94
3.4.3 Функціонал працівника компанії-постачальника 99
3.5 Висновки з розділу 3 101
ВИСНОВКИ 103
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 105

ВСТУП

Актуальність теми

У сучасних умовах цифровізації суспільства все більшої актуальності набуває автоматизація процесів у сфері харчування та обліку замовлень. Використання веб-застосунків дозволяє значно спростити взаємодію між замовниками та постачальниками, зменшити кількість помилок, пов’язаних із ручною обробкою інформації, а також забезпечити зручний контроль за виконанням замовлень.

Особливо важливою є автоматизація процесів у сфері шкільного харчування, оскільки організація харчування учнів потребує чіткого планування, регулярного формування замовлень, контролю за оплатою та відстеження статусів виконання. В умовах великої кількості користувачів та щоденних замовлень традиційні способи обліку є недостатньо ефективними.

Актуальність даної теми також зумовлена впровадженням реформ системи шкільного харчування в Україні, які пов’язані з оновленням стандартів харчування та формуванням збалансованого меню. Значний внесок у популяризацію реформи зробив український кулінар та громадський діяч Євген Клопотенко, який брав участь у створенні нового шкільного меню та концепції сучасного харчування для навчальних закладів [1]. У зв’язку з цим виникає потреба у створенні інформаційних систем, які дозволяють ефективно організувати процес замовлення та контролю харчування.

Особливої актуальності така система набуває для шкіл Запорізької області, зокрема для підземних шкіл та навчальних закладів, які працюють в умовах воєнного стану. Для таких закладів важливо забезпечити швидке формування замовлень, контроль постачання та зручну взаємодію між школами і постачальниками продуктів харчування. У зв’язку з цим було обрано тему розробки веб-застосунку для замовлення їжі з використанням сучасних веб-технологій на прикладі системи шкільного харчування.

Мета роботи

Метою роботи є розробка веб-застосунку для автоматизації процесу замовлення їжі з використанням сучасних веб-технологій та фреймворка Next.js.

Завдання роботи

Для досягнення поставленої мети необхідно виконати наступні завдання: провести аналіз особливостей процесу замовлення їжі, дослідити сучасні технології розробки веб-застосунків, спроєктувати структуру бази даних системи, реалізувати систему користувачів із різними ролями, розробити функціонал створення та обробки замовлень, реалізувати систему керування статусами замовлень, забезпечити збереження історії змін статусів та оплат, створити адаптивний користувацький інтерфейс.

Об’єкт дослідження

Об’єктом дослідження є процес автоматизації замовлення їжі у сфері організованого харчування.

Предмет дослідження

Предметом дослідження є методи та засоби розробки веб-застосунків для автоматизації процесу замовлення їжі з використанням фреймворка Next.js, бази даних PostgreSQL та ORM Prisma.

Методи дослідження

У роботі були використані такі методи дослідження: аналіз наукової та технічної літератури з тематики веб-розробки та інформаційних систем, аналіз існуючих рішень у сфері автоматизації замовлення їжі, проєктування структури бази даних, моделювання архітектури веб-застосунку, використання сучасних технологій фронтенд та бекенд розробки.

Практичне значення одержаних результатів

Практичне значення одержаних результатів полягає у створенні веб-застосунку, який дозволяє автоматизувати процес замовлення їжі, спростити взаємодію між школами та постачальниками, забезпечити контроль за статусами замовлень та оплатою.

Результати роботи можуть бути використані для впровадження у навчальних закладах, організаціях громадського харчування та інших сферах, де необхідна автоматизація процесу формування та обробки замовлень.

Апробація результатів

Основні положення та результати роботи були представлені у вигляді тез доповіді на науково-практичній конференції студентів, аспірантів і молодих вчених «Молода наука – 2026» [2].

Глосарій

Веб-застосунок — програмна система, доступ до якої здійснюється через веб-браузер.

Frontend — клієнтська частина веб-застосунку, яка відповідає за взаємодію з користувачем та відображення інтерфейсу.

Backend — серверна частина веб-застосунку, яка відповідає за обробку даних та бізнес-логіку.

Next.js — сучасний React-фреймворк для розробки веб-застосунків із підтримкою серверного рендерингу та API.

PostgreSQL — реляційна система управління базами даних.

Prisma — ORM-інструмент для взаємодії застосунку з базою даних.

ORM (Object-Relational Mapping) — технологія для роботи з базою даних через об’єкти мови програмування.

API — інтерфейс взаємодії між програмними компонентами.

База даних — організована структура для зберігання та обробки інформації.

CRUD-операції — базові операції роботи з даними: створення, читання, оновлення та видалення.

Аутентифікація — процес перевірки особи користувача під час входу в систему.

Авторизація — процес визначення прав доступу користувача в системі.

Адаптивний дизайн — підхід до розробки інтерфейсу, який забезпечує коректне відображення застосунку на різних пристроях.

Статус замовлення — поточний етап життєвого циклу замовлення у системі.

Життєвий цикл замовлення — послідовність станів замовлення від створення до завершення або скасування.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Особливості організації процесу замовлення їжі

Автоматизація процесів у сфері колективного та організованого харчування вимагає комплексного системного підходу та детального дослідження архітектури інформаційних потоків предметної області. Основна специфіка розроблюваної системи полягає в тому, що вона функціонує в межах моделі взаємодії B2B (Business-to-Business). На відміну від класичних роздрібних сервісів доставки їжі (модель B2C — Business-to-Consumer), де комунікація відбувається між замовником-фізичною особою та закладом громадського харчування на основі імпульсивних, миттєвих транзакцій, у сфері шкільного харчування взаємодія має характер системних, регламентованих та довгострокових відносин між двома контрагентами: замовником (навчальним закладом) та виконавцем (підприємством-постачальником готової кулінарної продукції або продовольчої сировини).

У межах цієї моделі інформаційні та логістичні потоки координуються через чітку взаємодію трьох ключових суб'єктів системи. Кожен із цих суб'єктів наділений унікальним колом функціональних обов'язків, бізнес-задач та рівнем доступу до даних у межах рольової моделі:

Адміністратор платформи - виконує функції глобального модератора та інфраструктурного контролера системи. Його основними завданнями є реєстрація нових контрагентів (навчальних закладів та організацій постачальників) у базі даних, адміністрування облікових записів, моніторинг цілісності системи, а також ведення загальних системних довідників.

Представник навчального закладу (Користувач) — виступає в ролі безпосереднього замовника, який трансформує щоденну потребу установи в сформовану заяву. Ця роль передбачає постійний моніторинг актуального меню, формування замовлень, гнучке коригування обсягів порцій залежно від

фактичної відвідуваності закладу, а також ініціацію та відстеження статусів оплати.

Постачальник — представник виробничо-логістичного підприємства, що забезпечує безпосереднє приготування, фасування та транспортування страв. Його функціональний контур охоплює актуалізацію щоденного та перспективного меню, обробку та консолідацію отриманих від навчальних закладів заявок для формування виробничих планів харчоблоків, переведення замовлень за етапами технологічного циклу, а також ведення фінансового обліку й верифікацію отриманих платежів.

Специфіка взаємодії накладає суворі вимоги до планування обсягів порцій та часових регламентів. У сфері організованого харчування повністю неприпустимим є формування замовлень «у реальному часі» за фактом виникнення випадкового попиту. Організація харчування учнів жорстко лімітується державними санітарними нормами, графіками навчального процесу та сучасними вимогами щодо калорійності й збалансованості раціону. Точний та випереджальний підрахунок кількості порцій є критично важливим з кількох взаємопов'язаних причин:

Економічна доцільність та суворий бюджетний контроль. Фінансування колективного харчування часто здійснюється за рахунок бюджетних коштів, субвенцій або цільових внесків, що вимагає прозорої звітності. Будь-які помилки в підрахунках призводять до нераціонального використання фінансів.

Мінімізація харчових відходів. Оскільки готові страви для шкільних їдалень мають обмежений термін зберігання і не можуть бути повторно реалізовані наступного дня, постачальник повинен мати підтверджену інформацію про точну кількість порцій ще до початку безпосереднього технологічного процесу приготування та закупівлі сировини.

Дотримання стандартів реформи шкільного харчування. Впровадження оновленого шкільного меню вимагає точного дотримання складних рецептур

і технологічних карт. Постачальник має заздалегідь розраховувати обсяги інгредієнтів відповідно до специфікації замовлених страв.

Додатковим специфічним фактором, що суттєво ускладнює координацію процесу, є необхідність оперативного реагування на форс-мажорні обставини. В умовах організації навчання та функціонування установ у безпекових просторах або підземних школах Запорізького регіону під час воєнного стану, структура та обсяги замовлень можуть динамічно змінюватися під впливом безпекових факторів та фактичної присутності дітей. Це вимагає від архітектури інформаційної системи максимальної гнучкості: представник установи повинен мати можливість оперативно вносити корективи в замовлення на етапі його обробки до моменту фіксації постачальником.

З огляду на це, життєвий цикл замовлення в аналізованому процесі є чітко визначеним, багатокроковим і складається з чітко розмежованих станів, що утворюють логічну машину станів (State Machine). На початковому етапі представник замовника формує чернетку (стан Draft), яка є локальною сутністю і дозволяє накопичувати дані, додавати страви та коригувати їхню кількість без відправки виконавцю. Після фінальної перевірки замовлення публікується в системі (стан Published), стаючи доступним для аналізу постачальнику. Постачальник оцінює власні виробничі потужності, логістичні маршрути та приймає замовлення у роботу (стан Accepted), після чого воно переходить на етап безпосереднього приготування та комплектації (стан In Progress). Завершується логістичний цикл станом Completed, що фіксує успішну доставку та приймання продукції, або, у разі виникнення об'єктивних перешкод, стан змінюється на Cancelled (скасовано).

Паралельно та синхронно із логістичним життєвим циклом рухається фінансовий цикл замовлення, який відображає стан взаєморозрахунків і запобігає виникненню касових розривів чи заборгованостей. З моменту створення та публікації заявка отримує статус неоплаченої (стан Unpaid). Після здійснення банківської транзакції представником установи замовлення переходить у стан оплаченого (стан Paid). Проте, враховуючи специфіку

безготівкових розрахунків між організаціями, фінальною точкою фінансового циклу є стан верифікації (стан Verified), який встановлюється виключно після підтвердження надходження коштів бухгалтерією постачальника.

Таким чином, організація процесу замовлення їжі у сфері колективного харчування характеризується високим рівнем відповідальності, жорсткою часовою прив'язкою, необхідністю точного прогнозування обсягів виробництва продукції та складним супутнім документообігом. Традиційні неавтоматизовані методи фіксації таких процесів (телефонні дзвінки, паперові накладні, чати у месенджерах) мають низьку ефективність в сучасних умовах, що зумовлює необхідність у впровадженні профільних інформаційних B2B-системи.

Впровадження та повноцінна автоматизація процесу замовлення їжі в таких умовах дозволяє отримати низку критично важливих переваг. До них належать: суттєве зменшення кількості помилок під час обробки багатокомпонентних замовлень, значне пришвидшення транзакційної взаємодії між навчальними закладами та постачальниками, централізоване та безпечне зберігання інформації, а також забезпечення прозорості роботи системи. Крім того, автоматизація відкриває можливість легкої адаптації та масштабування системи для різних типів організацій у майбутньому.

1.2 Аналіз проблем та недоліків неавтоматизованих систем обліку

Для глибокого розуміння доцільності розробки та впровадження спеціалізованого вебзастосунку необхідно провести критичний аналіз існуючих підходів до ведення обліку в організаціях, які не використовують профільне програмне забезпечення. Досі поширеною є практика, коли координація процесів між замовниками та постачальниками здійснюється за допомогою так званих «напівкустарних» або застарілих методів: паперових журналів, локальних електронних таблиць (MS Excel, Google Таблиці) та засобів швидкого обміну повідомленнями (месенджери Viber, Telegram).

Попри уявну простоту та низьку вартість впровадження, такі неавтоматизовані підходи мають низку фундаментальних недоліків, що негативно впливають на ефективність бізнес-процесів.

Першою і найбільш критичною проблемою є так званий людський фактор. У неавтоматизованих системах збирання, обробка та консолідація даних виконуються людиною вручну. Коли представник навчального закладу щодня підраховує кількість необхідних порцій для різних категорій користувачів, ризик виникнення арифметичних помилок або механічних описок різко зростає. Складність структури замовлень, яка містить різні категорії страв та велику кількість позицій меню, призводить до того, що на етапі ручного перенесення даних з одного джерела в інше інформація викривляється. Навіть одна випадкова помилка у цифрах на етапі формування замовлення з боку установи трансформується у фінансові втрати та логістичний збій на етапі приготування їжі постачальником.

Другим деструктивним фактором є використання месенджерів як основного каналу передачі замовлень. Попри високу швидкість доставлення повідомлень, месенджери абсолютно не пристосовані для ведення структурованого бізнес-обліку. Інформаційні потоки у спільних чатах мають хаотичний характер. Заявки від різних установ накопичуються в одній стрічці новин, через що постачальнику складно оперативно звести їх у єдиний виробничий план. Більш того, у месенджерах повністю відсутній механізм контролю версій та фіксації станів. Якщо представник установи відредагував або видалив повідомлення із замовленням після дедлайну, постачальник може цього не помітити, що призведе до приготування неправильної кількості страв. Відсутність чітко регламентованої «машини станів» унеможливлює розмежування відповідальності: стає неможливо довести, на якому саме етапі (чернетка, публікація, прийняття в роботу) виникла помилка чи відбулася несанкціонована зміна даних.

Третім слабким місцем є використання локальних таблиць Excel або некоректно налаштованих хмарних таблиць.Локальні файли створюють

проблему децентралізації та розсинхронізації даних. Постачальник та замовник змушені постійно пересилати один одному файли електронною поштою, що породжує конфлікти версій (наприклад, виникають десятки файлів із назвами на кшталт «Меню_v2_остаточне.xlsx»). Окрім цього, в електронних таблицях немає жорсткої валідації типів даних. Користувач може випадково ввести текстове значення у поле для введення кількості порцій, що заблокує автоматичні формули підрахунку. Також електронні таблиці не мають диференційованої системи прав доступу: користувач із доступом до редагування може випадково видалити важливі історичні дані, формули або позиції меню іншої установи.

Окремої уваги заслуговує низька швидкість передачі та обробки інформації, яка стає критичною у специфічних умовах функціонування установ. Для Запорізького регіону, де велика кількість організацій та навчальних закладів вимушені працювати в умовах воєнного стану, безпекових просторів або підземних шкіл, фактор часу має вирішальне значення. Форс-мажорні обставини, зміни безпекової ситуації або раптові коливання кількості присутніх дітей вимагають миттєвої реакції. Традиційні методи обміну даними (дзвінки, листи, Excel) є занадто повільними. Поки файл передається або повідомлення обробляється менеджером постачальника вручну, харчоблок уже запускає процес приготування страв за старими даними. Як наслідок — виникає надлишок нереалізованої продукції або, навпаки, дефіцит порцій.

Наостанок, неавтоматизований підхід унеможливлює ефективний фінансовий та історичний контроль. За відсутності інтегрованої бази даних фінансові цикли замовлень розірвані з логістичними. Стан оплати перевіряється бухгалтерією в ручному режимі через банківські виписки, що займає багато часу. Замовник не бачить у реальному часі, чи отримав постачальник кошти і чи переведено замовлення у статус «Verified». Більш того, ведення паперових журналів чи локальних файлів унеможливлює швидкий аналіз історії замовлень за минулі періоди, ускладнює формування

звітів та блокує можливість стратегічного планування закупівель продовольчої сировини.

Отже, аналіз показує, що ручне керування, месенджери та електронні таблиці неспроможні забезпечити надійність, швидкість, безпеку та цілісність даних, які необхідні для сучасної сфери колективного харчування. Описані проблеми та організаційний біль замовників і постачальників формують чіткий перелік вимог до проектованої інформаційної системи, архітектура якої має усунути зазначені недоліки.

1.3 Аналіз існуючих рішень

Сучасний ринок інформаційних технологій у сфері харчування (FoodTech) та автоматизації послуг можна класифікувати за трьома основними категоріями систем, які частково перекривають окремі задачі автоматизації замовлень.

Сервіси доставки їжі для кінцевих споживачів (B2C-сегмент): типовими представниками цієї категорії є міжнародні та локальні платформи агрегатори, такі як Glovo, Bolt Food або Uber Eats. Основна перевага цих систем — наявність високотехнологічних, адаптивних мобільних інтерфейсів, інтегрованих інтерактивних карт, геопозиціонування та систем миттєвої оплати. Проте архітектура цих платформ розроблена виключно під роздрібного покупця (модель Business-to-Consumer). Вони не підтримують укладання довгострокових корпоративних договорів, групове планування меню, формування консолідованих оптових замовлень від установ на сотні порцій та безготівковий розрахунок між юридичними особами з відстрочкою платежу за актами виконаних робіт. Крім того, висока комісія платформ (часто від 20% до 35%) робить їх економічно неприйнятними для соціальних та бюджетних проектів.

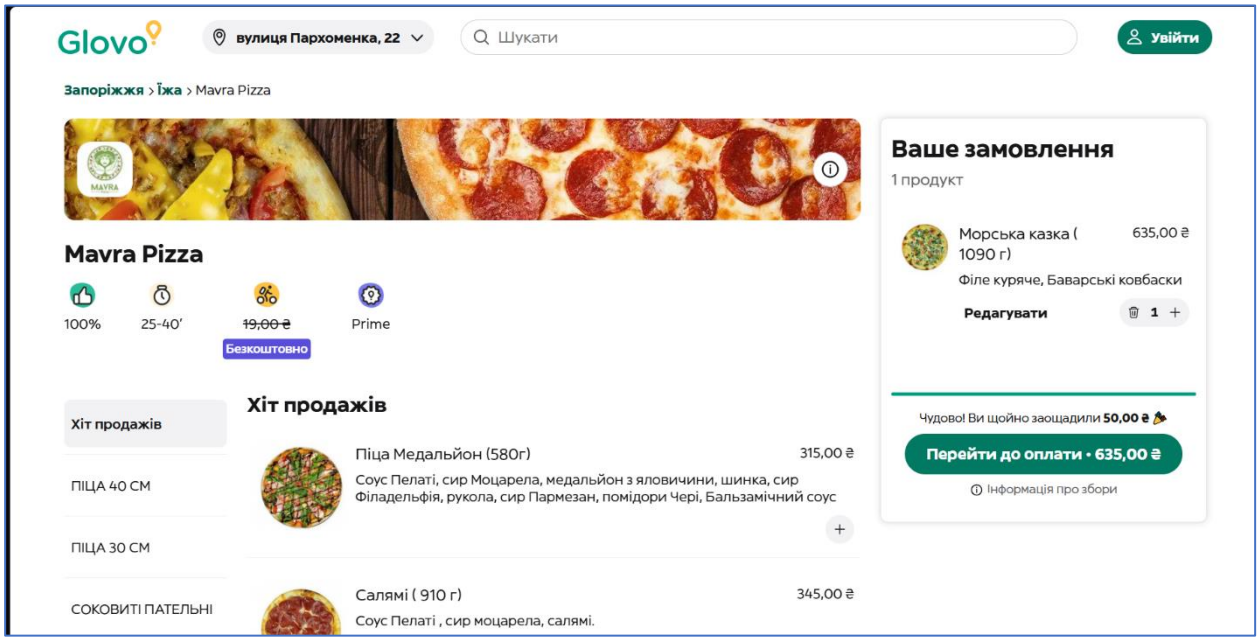


Рисунок *1 — Інтерфейс оформлення роздрібного замовлення в системі B2C-агрегатора (на прикладі Glovo)*

Спеціалізовані системи автоматизації ресторанного бізнесу (POS/ERP-системи): найбільш поширеними рішеннями в цьому сегменті в Україні є Poster, iiko, r_keeper або Syrve. Ці продукти демонструють високу ефективність у задачах внутрішнього операційного обліку: ведення складських залишків, автоматичне списання інгредієнтів за технологічними картами при продажу страв, калькуляція собівартості, фінансовий облік виручки харчоблоку та контроль роботи персоналу. Однак головним недоліком є замкненість архітектури всередині одного підприємства. Вони розраховані на роботу офіціантів, касирів або кухарів. У цих системах відсутній зручний зовнішній кабінет для стороннього клієнта-організації (наприклад, представника школи), який дозволяв би йому дистанційно і самостійно формувати щоденні заявки, контролювати стани замовлень та взаємодіяти з постачальником без прямого контакту з персоналом кафе чи ресторану.

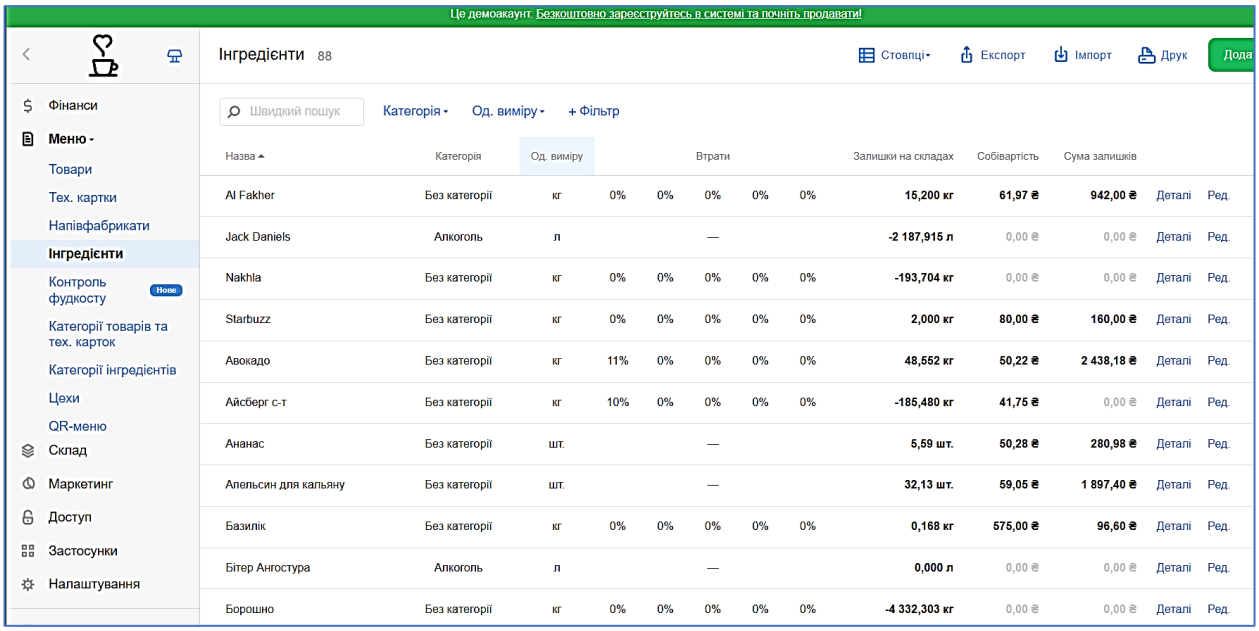


Рисунок *2 — Панель адміністрування та обліку залишків у системі Poster POS*

Спеціалізовані державні, муніципальні та інформаційні портали: в умовах активної реалізації реформи шкільного харчування в Україні було створено державні інформаційні ресурси, яскравим прикладом яких є національний портал «Знаймо». Такі портали містять величезну базу нормативно-правових документів, рекомендовані технологічні картки страв, розроблені за участю провідних кулінарних експертів, та навчальні матеріали для кухарів. Вони мають високу методичну цінність, проте є суто статичними інформаційними веб-ресурсами. Вони повністю позбавлені транзакційного функціоналу — не мають баз даних користувачів для ведення замовлень, не підтримують процеси оформлення заявок, зміни логістичних статусів та моніторингу оплат у реальному часі.



Рисунок 3 — *Головна сторінка національного порталу шкільного харчування «Знаїмо»*

Для систематизації результатів аналізу та наочного доведення необхідності розробки нової системи було сформовано порівняльну матрицю за ключовими інженерними та функціональними критеріями, що відображено у таблиці 1.

Таблиця *1 – Порівняльна таблиця аналогів*

A	B	C	D	E
Критерії порівняння	B2C сервіси (Glovo / Bolt Food)	POS-системи (Poster / iiko)	Інфо-портали (Портал «Знаїмо»)	Проєктована система (School Meal System)
Орієнтація на B2B-модель	Відсутня (лише роздріб B2C)	Часткова (внутрішній облік ресторану)	Відсутня (суто інформаційні дані)	Повна (взаємодія Організація-Постачальни
Рольова модель доступу (Адмін/Замовник/Постачальник)	Відсутня (Клієнт / Кур'єр)	Часткова (Адмін / Касир / Кухар)	Відсутня	Повна (ізольовані кабінети для 3-х ролей)
Багатокроковий життєвий цикл (Draft >Published > Accepted...)	Відсутній (спрощений цикл доставки)	Відсутній (миттєвий продаж на касі)	Відсутній	Повна реалізація логістичної машини станів
Контроль фінансового циклу (Unpaid > Paid > Verified)	Відсутній (миттєвий еквайринг)	Часткова (фіксація факту оплати на касі)	Відсутній	Повна синхронізація логістики з банківським підтвердженням
Оптимізація під слабкий зв'язок (SSR архітектура)	Ні (важкі мобільні додатки / SPA)	Ні (потребують стабільного локального зв'язку)	Часткова (статичний HTML)	Повна (Next.js SSR для швидкого рендерингу)
Оптові замовлення та планування меню	Відсутні	Часткова (калькуляційні карти)	Відсутня (лише перегляд рецептів)	Повна (масштабування порцій під установи)

Проведений порівняльний аналіз наочно демонструє наявність суттєвого функціонального розриву (gap-analysis) між потребами ринку організованого харчування та можливостями наявного програмного забезпечення. Жоден із розглянутих аналогів не здатний комплексно вирішити задачу забезпечення прозорої, швидкої та безпечної взаємодії між замовником-організацією та підприємством-виконавцем, особливо в специфічних або форс-мажорних умовах функціонування.

Таким чином, розробка спеціалізованого веб-застосунку замовлення їжі на прикладі шкільного харчування на базі сучасного технологічного стеку Next.js, є повністю обґрунтованою та актуальною задачею, що дозволить заповнити відкриту нішу в сегменті автоматизації B2B-процесів громадського харчування.

1.4 Постановка задачі

Проведений аналіз предметної області, особливостей організації процесу замовлення їжі та недоліків існуючих неавтоматизованих і частково автоматизованих рішень показав необхідність створення спеціалізованої інформаційної системи, орієнтованої на автоматизацію процесів взаємодії між навчальними закладами та постачальниками харчування.

Основною задачею роботи є розробка веб-застосунку для автоматизації процесу формування, обробки та контролю замовлень їжі у сфері організованого харчування на прикладі шкільних установ. Система повинна забезпечувати централізоване зберігання даних, швидкий обмін інформацією між користувачами та контроль усіх етапів життєвого циклу замовлення.

Проєктований веб-застосунок має реалізовувати модель багатокористувацької системи з підтримкою рольового доступу. У системі передбачено наявність декількох категорій користувачів, кожна з яких виконує власні функції: адміністратор системи, представник навчального закладу, представник організації-постачальника.

Адміністратор повинен мати можливість здійснювати керування обліковими записами користувачів, створювати організації, контролювати коректність роботи системи та забезпечувати підтримку цілісності даних.

Представник навчального закладу повинен мати доступ до функціоналу перегляду меню, формування замовлень, редагування кількості порцій, перегляду історії замовлень та контролю їх поточних статусів.

Представник постачальника повинен мати можливість керування меню, обробки отриманих замовлень, зміни статусів виконання, а також ведення обліку оплат та підтвердження фінансових операцій.

З огляду на специфіку предметної області, важливою задачею є реалізація механізму життєвого циклу замовлення. Кожне замовлення повинно проходити послідовність визначених станів: чернетка, опубліковане, прийняте, в процесі виконання, завершене або скасоване. Такий підхід дозволяє забезпечити контроль етапів виконання та розмежування відповідальності між учасниками процесу.

Крім логістичного циклу, система повинна підтримувати фінансовий цикл замовлення, який включає стани неоплачено, оплачено та підтверджено. Це необхідно для забезпечення прозорості фінансових операцій та контролю взаєморозрахунків між сторонами.

Однією з ключових задач є забезпечення збереження історії змін статусів замовлень та оплат. Наявність журналу змін дозволить здійснювати аудит дій користувачів, відстежувати послідовність обробки замовлень та підвищити надійність системи.

Для реалізації поставлених задач необхідно: спроєктувати структуру бази даних, реалізувати серверну логіку обробки запитів, створити адаптивний користувацький інтерфейс, забезпечити авторизацію та автентифікацію користувачів, реалізувати механізми фільтрації та пошуку замовлень, забезпечити коректну взаємодію між клієнтською та серверною частинами застосунку.

У результаті виконання роботи має бути створений сучасний веб-застосунок, який забезпечить автоматизацію процесу замовлення їжі, зменшення кількості помилок під час обробки заявок, підвищення швидкості взаємодії між організаціями та централізоване керування інформацією в системі.

1.5 Висновки до розділу 1

У першому розділі було проведено аналіз предметної області автоматизації процесу замовлення їжі у сфері організованого харчування. Досліджено особливості взаємодії між навчальними закладами та постачальниками харчування, а також визначено основні функціональні та організаційні вимоги до інформаційної системи.

У ході аналізу встановлено, що процес організації шкільного харчування характеризується високим рівнем відповідальності, необхідністю точного планування кількості порцій, контролем виконання замовлень та супровідним фінансовим обліком. Особливу увагу приділено специфіці функціонування навчальних закладів у сучасних умовах, зокрема необхідності оперативної взаємодії між учасниками процесу в умовах воєнного стану та роботи підземних шкіл Запорізького регіону.

Було проаналізовано недоліки неавтоматизованих підходів до ведення обліку, серед яких використання паперової документації, електронних таблиць та месенджерів. Встановлено, що такі методи не забезпечують належного рівня швидкості, надійності, цілісності та безпеки даних, а також ускладнюють контроль за статусами замовлень і оплат.

Проведений аналіз існуючих програмних рішень показав, що сучасні сервіси доставки їжі, POS-системи та інформаційні портали лише частково покривають потреби сфери організованого харчування. Більшість із них орієнтована або на роздрібного користувача, або на внутрішню автоматизацію

закладів харчування, що не дозволяє забезпечити повноцінну B2B-взаємодію між навчальними закладами та постачальниками.

На основі проведеного аналізу було сформульовано постановку задачі та визначено основні вимоги до розроблюваного веб-застосунку. Встановлено необхідність реалізації системи з підтримкою ролей користувачів, механізмом керування життєвим циклом замовлення, обліком оплат, збереженням історії змін та адаптивним веб-інтерфейсом.

Отже, результати проведеного аналізу підтвердили актуальність і доцільність розробки веб-застосунку для автоматизації процесу замовлення їжі, що стане основою для подальшого проєктування та реалізації інформаційної системи у наступних розділах роботи.

2 ОПИС ТА ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАСТОСУНКУ

2.1 Архітектура веб-застосунку

Для реалізації системи управління шкільним харчуванням обрано повностекову (full-stack) архітектуру в межах єдиного проєкту, що об'єднає клієнтську та серверну частини без потреби підтримувати дві окремі кодові бази. Це дозволить спільно використовувати типи даних і схеми валідації на обох рівнях, зменшить кількість зовнішніх залежностей і спростить розгортання.

Застосунок матиме три чітко розмежовані рівні, зображені на рисунку 4:

- **Клієнтський рівень** — відповідатиме за відображення інтерфейсу, стан форм та локальний UI-стан.
- **Серверний рівень** — виконуватиме бізнес-логіку, перевірку прав доступу, валідацію вхідних даних та роботу з базою даних.
- **Рівень даних** — забезпечуватиме зберігання і цілісність даних через реляційну СУБД із використанням ORM.

Код усередині серверного рівня організовуватиметься за принципом Feature-based архітектури: кожна функціональна область системи матиме власний ізольований модуль із чотирма шарами:

- **серверні дії** — точки входу для мутацій, які ініціює клієнт;
- **сервіси** — бізнес-логіка та звернення до бази даних;
- **схеми** — декларативна валідація вхідних даних;
- **компоненти** — UI відповідного модуля.

Такий поділ дозволить модулям залишатися незалежними один від одного і розширювати систему без торкання вже написаного коду.

Клієнт взаємодіятиме із сервером через серверні дії — функції, що виконуються виключно на сервері і викликаються напряму з клієнтських компонентів, без необхідності проєктувати окремий REST API. Повний

ланцюжок обробки запиту — від взаємодії користувача з формою до отримання відповіді — детально показаний на рисунку 5.

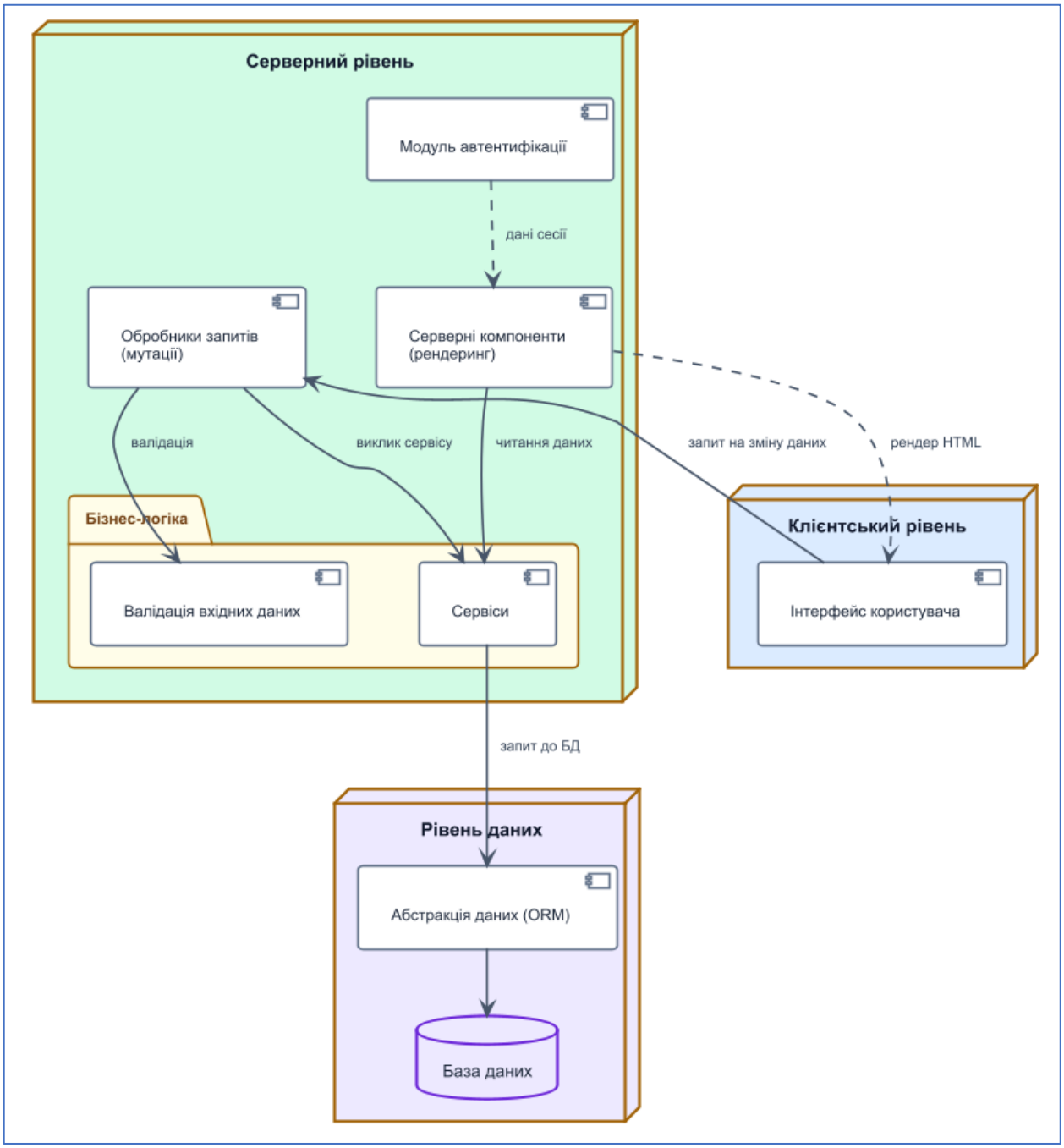


Рисунок *4 — Діаграма компонентів архітектури веб-застосунку*

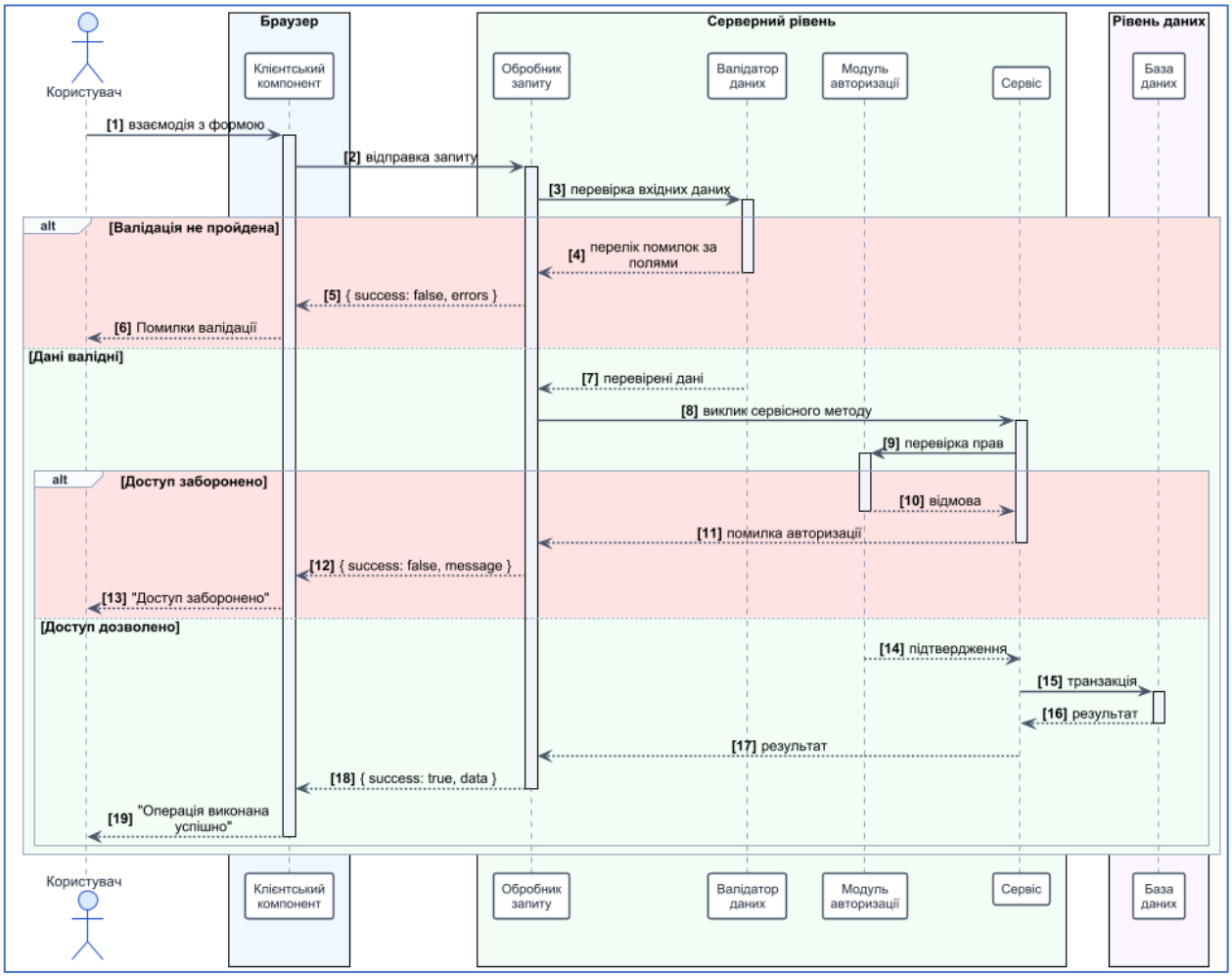


Рисунок *5* — *Діаграма послідовності виконання*

Маршрутизація будуватиметься на основі файлової структури проєкту. Маршрути поділятимуться на відкриті (сторінки автентифікації) і захищені (усі функціональні розділи застосунку). Для доступу до захищених маршрутів потрібна активна сесія — за її відсутності користувача буде перенаправлено на сторінку входу.

Аутентифікацію буде реалізовано на основі JWT: після успішного входу сервер видаватиме підписаний токен із вбудованими даними про роль та організацію користувача. Токен зберігатиметься у захищеному HTTP-cookie і верифікуватиме кожен наступний запит без додаткових звернень до бази даних.

Авторизація будуватиметься за патерном Policy — окремих класів із методами перевірки дозволів для кожної операції. Це зробить логіку прав доступу явною, централізованою та легкою для перевірки, а також виключить її дублювання на рівні представлення.

Бізнес-логіку переходів між статусами замовлень та оплат буде формалізовано у вигляді скінченних автоматів (state machines), що зберігатимуться у доменному шарі модуля замовлень. Це гарантуватиме коректність переходів між станами незалежно від того, звідки надійшов виклик, і зробить правила системи прозорими та передбачуваними.

2.2 Дизайн інтерфейсу користувачів

Інтерфейс застосунку проєктуватиметься за принципами адаптивного дизайну з підходом mobile-first — спочатку для мобільних пристроїв, а потім із розширенням макету під більші екрани. Це забезпечить зручне і коректне відображення незалежно від того, з якого пристрою заходить користувач.

Загальний макет матиме три постійні структурні елементи:

- **Верхня панель** — міститиме кнопку відкриття навігаційного меню на мобільних пристроях, назву застосунку та елементи керування обліковим записом (посилання на профіль і вихід із системи).
- **Бічна навігаційна панель** — на широких екранах відображатиметься постійно, а на мобільних розгортатиметься як накладний елемент поверх основного контенту. Перелік пунктів меню формуватиметься динамічно залежно від ролі та типу організації авторизованого користувача.
- **Область основного контенту** — адаптивний контейнер, що займатиме весь доступний простір між панелями і масштабуватиме внутрішні відступи залежно від ширини екрана.

Навігацію буде спроєктовано так, щоб кожен користувач бачив лише ті розділи, до яких має доступ. Адміністратор матиме доступ до управління організаціями, користувачами, каталогом меню та замовленнями.

Представник школи бачитиме каталог меню, власні замовлення та аналітику. Представник постачальника — опубліковані замовлення, власні замовлення та аналітику.

Для позначення статусів замовлень та оплат використовуватиметься кольорове кодування: кожному статусу відповідатиме окремий колір мітки, що дозволить з першого погляду зрозуміти стан замовлення без читання тексту.

Форми проєктуватимуться з урахуванням принципу миттєвого зворотного зв'язку: помилки валідації відображатимуться безпосередньо під відповідними полями в реальному часі, ще до відправки форми. Це зменшить кількість невдалих спроб і зробить роботу з формами комфортнішою.

Сповіщення про результат виконаних операцій реалізовуватимуться у форматі ненав'язливих toast-повідомлень, що на короткий час з'являтимуться в кутку екрана, не заважаючи поточній роботі.

Доступність забезпечуватиметься через коректну семантичну розмітку та відповідність стандарту ARIA для інтерактивних компонентів — модальних вікон, випадаючих меню, вкладок. Це гарантуватиме правильну роботу з екранними читачами та повноцінну клавіатурну навігацію.

Табличне представлення даних використовуватиметься для відображення реєстрів — організацій, користувачів, позицій меню та замовлень. Таблиці підтримуватимуть пагінацію та фільтрацію, що полегшить роботу з великими наборами записів.

2.3 Нефункціональні вимоги

Нефункціональні вимоги описують якісні характеристики системи — вони не визначають конкретну поведінку, але безпосередньо впливають на надійність, безпеку та зручність використання. Для системи управління шкільним харчуванням сформульовано такий перелік нефункціональних вимог, згрупованих за категоріями.

Безпека:

- Паролі зберігаються виключно у хешованому вигляді з використанням алгоритму bcrypt.
- Сесії реалізовуватимуться через JWT-токени.
- Кожна серверна дія перевірятиме автентифікацію та відповідні права доступу до того, як виконати основну логіку.
- Авторизаційна логіка буде централізована в policy-класах і не дублюватиметься на рівні представлення.

Продуктивність:

- React Server Components формуватимуть HTML на сервері, що мінімізує обсяг JavaScript, який передається клієнту.
- Кешування реалізовуватиметься через теги Next.js, що дозволить точково інвалідувати кеш після мутацій без повного перезавантаження сторінок.
- На полях, що використовуються у фільтрації — orderStatus, paymentStatus, schoolId, supplierId — буде встановлено індекси бази даних.

Надійність:

- Типобезпека забезпечуватиметься на всіх рівнях застосунку: TypeScript — на етапі компіляції, Zod — під час виконання.
- Для кожного замовлення вестиметься повний аудит-трейл із фіксацією автора та часу кожної зміни статусу замовлення і статусу оплати.
- Атомарність операцій над пов'язаними записами гарантуватиметься транзакціями Prisma.

Масштабованість:

- Модульна feature-based архітектура спрощуватиме додавання нових функціональних областей без необхідності змінювати вже написаний код.

- Використання ORM забезпечуватиме незалежність від конкретного постачальника СУБД.
- Composable-підхід при побудові UI-компонентів дозволятиме розширювати дизайн-систему, не порушуючи наявні інтерфейси.

Підтримуваність:

- Єдина структура кожного feature-модуля знижуватиме когнітивне навантаження при навігації по кодовій базі.
- Стандартизований тип відповіді ActionResult для всіх серверних дій забезпечуватиме консистентну обробку результатів на клієнті.
- ESLint та Prettier буде налаштовано для автоматизованого контролю якості та форматування коду.

2.4 Функціональність застосунку

Функціональність застосунку визначається набором операцій, доступних кожній ролі користувача. Система підтримуватиме три ролі: адміністратор, представник школи та представник постачальника. Кожна роль матиме визначений перелік дозволених дій, що забезпечуватиметься на рівні серверної логіки через policy-класи. Загальна карта взаємодії кожної ролі з функціями системи зображена на рисунку 6.

Ключовою частиною бізнес-логіки є управління станами замовлення та оплати. Усі допустимі переходи між станами буде формалізовано у вигляді скінченних автоматів. Діаграма станів замовлення зображена на рисунку 7, діаграма станів оплати — на рисунку 8.

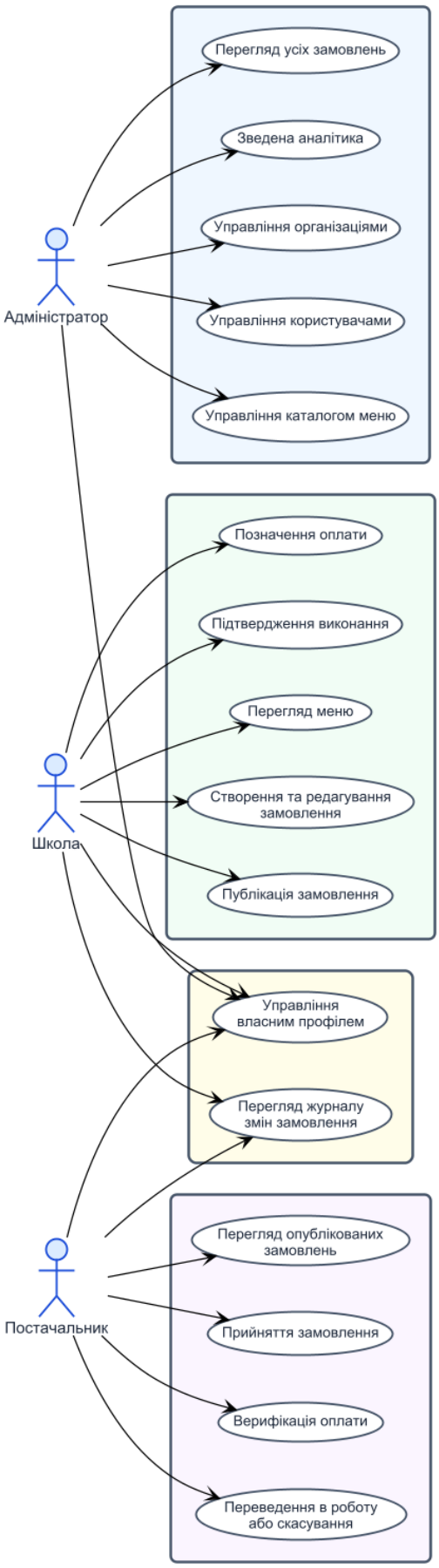


Рисунок *6* — *Діаграма варіантів використання*

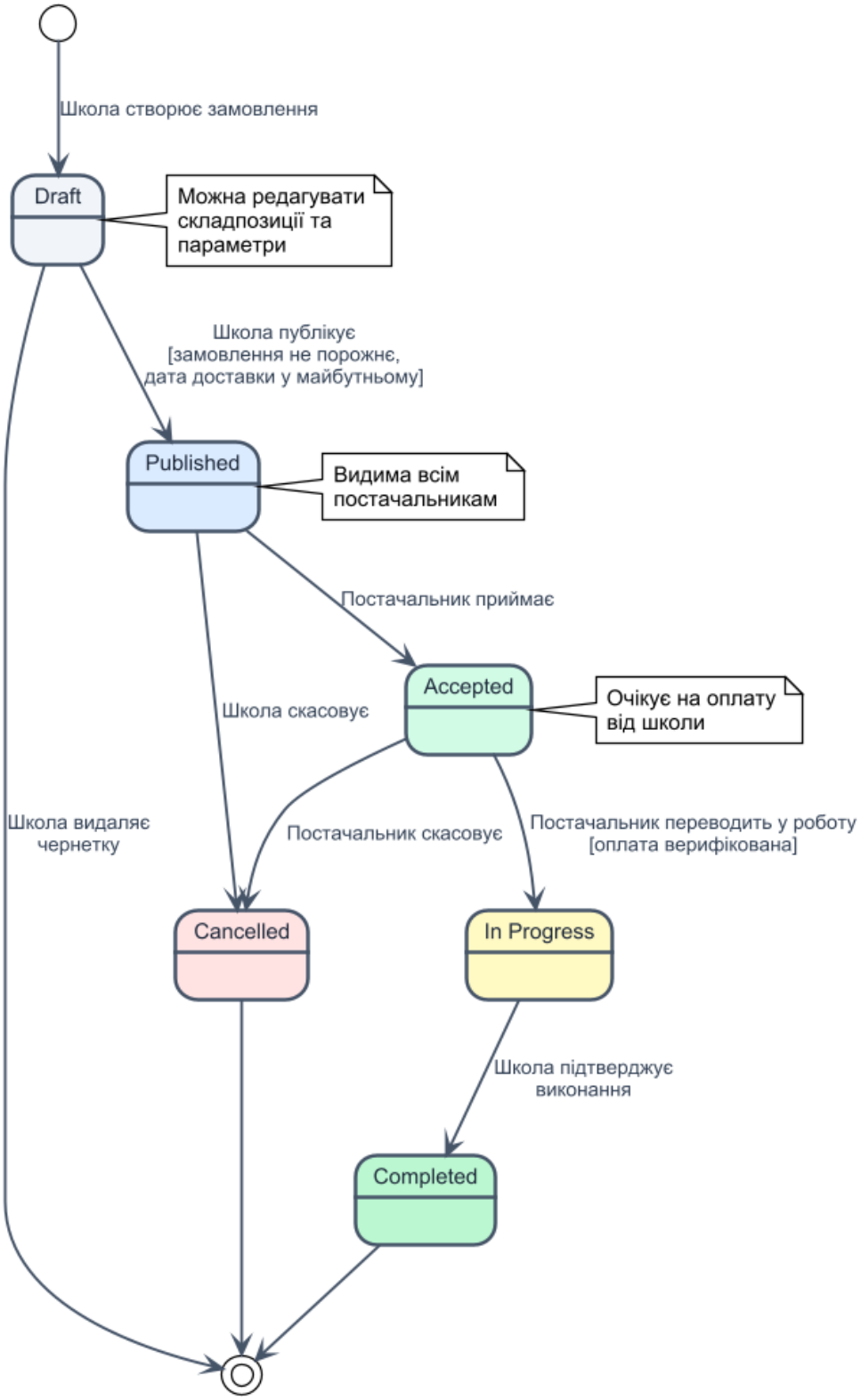


Рисунок *7 — Діаграма станів замовлення*



Рисунок *8 — Діаграма станів оплати*

У підрозділах нижче описано функціональні можливості для кожної категорії користувачів.

2.4.1 Функціональність для шкіл

Представники шкіл є основними ініціаторами процесу замовлення харчування. Після входу в систему вони матимуть доступ до власного розділу, де зосереджено весь цикл управління замовленнями — від перегляду каталогу страв до підтвердження виконання замовлення.

Перегляд меню:

- Перегляд каталогу усіх страв, згрупованих за категоріями.

- Перегляд детальної інформації по кожній позиції: назва, опис, ціна, зображення.

Управління замовленнями:

- Створення нового замовлення зі статусом draft із вибором дати доставки та необов'язкового коментаря.
- Додавання позицій меню до чернетки замовлення із зазначенням кількості.
- Редагування кількості та видалення позицій у чернетці.
- Публікація готового замовлення, після якої воно стає видимим для постачальників.
- Скасування опублікованого замовлення до моменту його прийняття постачальником.
- Перегляд детальної інформації про замовлення: перелік позицій, загальна вартість, поточний статус, дата доставки.

Управління оплатою:

- Позначення замовлення як оплаченого після його прийняття постачальником.

Аналітична панель:

- Перегляд загальної кількості замовлень своєї школи.
- Перегляд кількості активних замовлень та найближчої дати доставки.
- Перегляд загальної суми неоплачених замовлень.

2.4.2 Функціональність для постачальників

Постачальники є виконавцями замовлень у системі. Після входу вони матимуть доступ до переліку замовлень, призначених їхній організації, та інструментів управління процесом виконання. Постачальник не бачитиме замовлень інших постачальників і не зможе редагувати склад або параметри замовлення — ці дані формує виключно школа.

Перегляд замовлень:

- Перегляд усіх опублікованих замовлень.
- Перегляд детальної інформації по кожному замовленню: школа-замовник, дата доставки, склад замовлення, загальна вартість.
- Перегляд власних прийнятих та активних замовлень.

Управління виконанням:

- Прийняття опублікованого замовлення.
- Верифікація отриманої від школи оплати.
- Переведення замовлення у статус in_progress після верифікації оплати, що означає початок виконання.
- Скасування прийнятого замовлення у разі ненадходження оплати.

Аналітична панель:

- Перегляд кількості прийнятих замовлень та замовлень у виконанні.
- Перегляд очікуваних дат доставок.
- Перегляд поточного стану оплат за замовленнями.

2.4.3 Функціональність для адміністратора

Адміністратор матиме найширші права доступу в системі. На відміну від представників шкіл і постачальників, він не прив'язаний до жодної організації та матиме повний доступ до управління всіма ресурсами. Основне призначення адміністратора — первинне налаштування системи (реєстрація організацій і користувачів, формування каталогу харчування) та подальший операційний моніторинг.

Управління організаціями:

- Перегляд повного списку організацій (шкіл та постачальників) з пагінацією та пошуком.
- Створення нових організацій із зазначенням назви, типу (school або supplier), контактного email та телефону.
- Редагування даних існуючих організацій.
- Видалення організацій із системи.

Управління користувачами:

- Перегляд списку всіх користувачів із фільтрацією.
- Створення нових користувачів та прив'язка їх до організацій.
- Редагування даних користувачів: ім'я, email, роль, організація.
- Видалення облікових записів.

Управління каталогом харчування*:*

- Створення, редагування та видалення категорій страв.
- Створення, редагування та видалення позицій меню із зазначенням назви, опису, ціни, зображення та ознаки доступності.
- Управління доступністю позицій меню (активна / неактивна).

Моніторинг замовлень:

- Перегляд усіх замовлень у системі незалежно від організації.
- Фільтрація замовлень за статусом, організацією та діапазоном дат.

Зведена аналітика:

- Перегляд загальної кількості організацій та замовлень у системі.
- Перегляд загальної суми неоплачених замовлень.
- Перегляд розподілу замовлень за статусами.
- Перегляд топ-5 шкіл та топ-5 постачальників за обсягом замовлень.

2.4.4 Додаткові функціональні можливості

Окрім рольово-специфічних функцій, система надаватиме ряд можливостей, доступних усім автентифікованим користувачам — спільну інфраструктуру застосунку, що не прив'язана до жодної конкретної ролі.

Управління профілем:

- Оновлення особистих даних: ім'я, прізвище, адреса електронної пошти.
- Зміна пароля.
- Завантаження та оновлення аватара.

Аудит-трейл замовлень:

- Фіксація кожної зміни статусу замовлення із зазначенням попереднього та нового статусу, ідентифікатора автора та часової позначки.
- Фіксація кожної зміни статусу оплати з аналогічним набором атрибутів.
- Перегляд повної журнальної історії для будь-якого замовлення.

Пагінація та фільтрація:

- Пагінація для всіх реєстрів: організації, користувачі, позиції меню, замовлення.
- Пошук за текстовими полями у відповідних реєстрах.
- Фільтрація замовлень за статусом, типом організації та діапазоном дат.

2.5 Огляд систем управління базами даних

Вибір системи управління базами даних (СУБД) — одне з ключових архітектурних рішень, що безпосередньо впливає на надійність, масштабованість і продуктивність застосунку. Нижче наведено порівняльний аналіз чотирьох поширених СУБД.

MySQL — реляційна СУБД з відкритим вихідним кодом, що використовує SQL для визначення, маніпуляції та запиту даних. Це одна з найпопулярніших баз даних у веб-розробці: вона традиційно входить до стеку LAMP і підтримується переважною більшістю хмарних хостинг-провайдерів.

Переваги:

- Велика спільнота, зріла екосистема та вичерпна документація.
- Проста початкова конфігурація і низький поріг входу для розробників-початківців.
- Широка підтримка з боку хостинг-провайдерів і хмарних платформ.

Недоліки:

- Неповна відповідність стандарту SQL: низка розширених можливостей з'явилася лише у відносно пізніх версіях.

- Обмежена підтримка складних типів даних порівняно з PostgreSQL (JSON, масиви, діапазони).
- Слабша підтримка конкурентних транзакцій і складних аналітичних запитів.

MongoDB — документоорієнтована NoSQL-система, що зберігає дані у форматі BSON (Binary JSON). Вона пропонує гнучку схему даних, де кожен документ може мати довільну структуру, і орієнтована на горизонтальне масштабування та високу пропускну здатність операцій запису.

Переваги:

- Гнучка схема дозволяє змінювати структуру документів без міграцій.
- Зручне зберігання ієрархічних і вкладених структур без потреби в нормалізації.
- Висока швидкість запису завдяки відсутності жорстких реляційних обмежень.

Недоліки:

- Відсутність підтримки зовнішніх ключів і декларативних зв'язків між колекціями на рівні рушія.
- Обмежена підтримка ACID-транзакцій, що охоплюють кілька колекцій.
- Складність забезпечення узгодженості даних між взаємопов'язаними сутностями без реляційних обмежень.

SQLite — вбудована реляційна СУБД, що зберігає всю базу даних в одному файлі та не потребує окремого серверного процесу. SQLite реалізує значну підмножину стандарту SQL і широко застосовується в мобільних застосунках, браузерах та середовищах розробки.

Переваги:

- Не потребує встановлення й налаштування окремого сервера.
- Мінімальні системні вимоги та висока швидкість у однокористувацьких сценаріях.

- Повна переносимість: база даних є звичайним файлом, який легко скопіювати або перенести.

Недоліки:

- Не розрахована на конкурентний багатокористувацький доступ: одночасний запис блокує всю базу даних.
- Суттєві обмеження масштабування: відсутня підтримка шардування, реплікації та клієнт-серверної архітектури.
- Відсутність ряду функцій, притаманних повноцінним СУБД: віконних функцій, паралельних транзакцій тощо.

PostgreSQL — об'єктно-реляційна СУБД з відкритим вихідним кодом, що вирізняється повною відповідністю стандарту SQL, розвиненою підтримкою транзакцій і конкурентності, а також широким набором вбудованих типів даних. PostgreSQL активно розвивається з 1996 року і є одним із найбільш функціонально повних рішень серед вільного програмного забезпечення.

Переваги:

- Повна підтримка ACID-транзакцій, зовнішніх ключів, каскадних операцій і складних обмежень цілісності.
- Тип DECIMAL для точного зберігання грошових значень без похибок округлення, властивих числам із рухомою крапкою.
- Підтримка розширених типів даних: JSON, масиви, діапазони, спеціалізовані типи для геоданих тощо.
- Широкі можливості індексування: B-tree, Hash, GIN, GiST — для оптимізації різних класів запитів.

Недоліки:

- Складніше початкове налаштування порівняно з MySQL або SQLite.

- Вище споживання оперативної пам'яті, що може бути критичним на серверах з обмеженими ресурсами.
- Менша поширеність серед бюджетних хостинг-провайдерів порівняно з MySQL.

З огляду на вищенаведений аналіз PostgreSQL є найбільш обґрунтованим вибором для розроблюваної системи з декількох причин. По-перше, розроблювана система оперує фінансовими даними — цінами та сумами замовлень — де тип DECIMAL(10,2) є стандартним підходом до точного зберігання. По-друге, модель даних є суто реляційною: організації, користувачі, замовлення та журнальні записи пов'язані зовнішніми ключами з каскадними операціями. По-третє, ACID-транзакції є критично важливими для атомарного оновлення статусу замовлення та синхронного запису до журналу змін. Обмеження SQLite у сценаріях конкурентного доступу і відсутність реляційних гарантій у MongoDB виключають ці технології з розгляду для даного застосунку.

2.6 Огляд інструментів серверної розробки

Вибір серверного фреймворку визначає архітектуру взаємодії між клієнтом і сервером, підходи до рендерингу, модель безпеки та складність підтримки кодової бази. Нижче наведено порівняльний аналіз поширених рішень.

Express.js — мінімалістичний і гнучкий веб-фреймворк для Node.js, що надає базові інструменти для побудови веб-застосунків і REST API: маршрутизацію, систему middleware та обробку HTTP-запитів. Express не нав'язує жодної конкретної архітектури й залишає розробнику повну свободу у структуруванні проєкту.

Переваги:

- Мінімалістичність і гнучкість: розробник самостійно обирає всі компоненти архітектури.

- Зріла екосистема з великою кількістю middleware-пакетів для вирішення типових задач.
- Низький поріг входу для розробників, знайомих із Node.js.

Недоліки:

- Відсутність вбудованої структури проєкту: вся архітектура лягає на плечі розробника, що може призводити до неузгодженості у команді.
- Побудова SSR або повностекового застосунку потребує значного ручного налаштування та підключення додаткових бібліотек.
- Спільне використання TypeScript-типів між клієнтською і серверною частинами вимагає окремого налаштування і є менш зручним порівняно з монорепозиторними рішеннями.

Django — повнофункціональний веб-фреймворк для Python, що дотримується принципу «batteries included». Django надає вбудовані інструменти для ORM, міграцій, адміністративної панелі, автентифікації, шаблонізації та безпеки, орієнтуючись на швидку розробку серверно-рендерованих застосунків.

Переваги:

- Широкий набір вбудованих інструментів, що покриває більшість типових потреб веб-застосунку без залучення сторонніх бібліотек.
- Зріла система міграцій і ORM із підтримкою більшості популярних СУБД.
- Надійна конфігурація безпеки за замовчуванням: захист від CSRF, XSS і SQL-ін'єкцій.

Недоліки:

- Використання Python на бекенді разом із TypeScript-фронтендом означає підтримку двох окремих кодових баз різними мовами, що ускладнює розробку й унеможливлює спільне використання типів.
- Монолітна архітектура Django може бути надлишковою для застосунків, що потребують гнучкої API-орієнтованої або мікросервісної моделі.

- Django ORM і система шаблонів мають відчутну криву навчання для розробників без досвіду роботи з Python.

Ruby on Rails — повнофункціональний веб-фреймворк для Ruby, що будується на принципах «convention over configuration» і «don't repeat yourself». Rails надає вбудовані ORM (Active Record), систему міграцій, маршрутизацію та шаблонізацію, роблячи ставку на максимальну швидкість розробки при дотриманні встановлених конвенцій.

Переваги:

- Висока швидкість розробки завдяки жорстким конвенціям і генераторам коду (scaffolding).
- Зріла екосистема бібліотек (gems) для більшості типових задач веб-розробки.
- Active Record — один із найбільш зрілих і функціональних ORM у веб-розробці.

Недоліки:

- Як і Django, потребує окремої серверної кодової бази на Ruby при використанні TypeScript-фронтенду.
- Продуктивність Ruby є нижчою за Node.js при обробці конкурентних запитів, що може бути суттєвим обмеженням для навантажених застосунків.
- Жорсткі конвенції Rails звужують простір для архітектурних рішень, що ускладнює реалізацію нестандартних сценаріїв.

Next.js — повностековий React-фреймворк для Node.js, побудований навколо концепції єдиної кодової бази для клієнтської та серверної логіки. Next.js підтримує кілька стратегій рендерингу (SSR, SSG, ISR, RSC), вбудовану файлову маршрутизацію, Server Actions для обробки мутацій і гнучке кешування на рівні фреймворку.

Переваги:

- Єдина TypeScript-кодова база для клієнтської і серверної частини застосунку дозволяє спільно використовувати типи, схеми валідації та бізнес-логіку без дублювання.
- Server Actions забезпечують виконання мутацій даних виключно на сервері без необхідності писати REST-ендпоінти, що скорочує поверхню атаки.
- React Server Components дозволяють робити запити до бази даних безпосередньо в компонентах — без клієнтських хуків, що суттєво спрощує архітектуру.
- Органічна інтеграція з NextAuth.js — усталеним стандартом автентифікації в екосистемі Next.js.

Недоліки:

- Модель рендерингу (Server Components проти Client Components) є досить складною і потребує чіткого розуміння відмінностей та обмежень кожного підходу.
- Менша гнучкість у відступі від конвенцій фреймворку порівняно з Express.js.

Беручи до уваги результати вищенаведеного аналізу, вибір Next.js зумовлений насамперед можливістю побудувати повностековий застосунок в єдиній TypeScript-кодовій базі — без необхідності підтримувати окремі клієнтський і серверний проєкти, як це було б у випадку Django або Rails. Механізм Server Actions і React Server Components принципово спрощують архітектуру порівняно з Express.js, де аналогічна функціональність потребувала б значного обсягу ручного налаштування.

2.7 Огляд інструментів розробки інтерфейсу користувача

Вибір клієнтської бібліотеки або фреймворку визначає модель компонентів, підхід до управління станом і загальну продуктивність

інтерфейсу. Нижче наведено порівняльний аналіз найпоширеніших інструментів.

Angular — комплексний фреймворк для розробки клієнтських застосунків від Google, написаний на TypeScript. Angular реалізує строгу архітектурну модель із модулями, компонентами, сервісами, директивами та системою ін'єкції залежностей, і надає вбудовані засоби маршрутизації, роботи з формами та HTTP-клієнта.

Переваги:

- Комплексне рішення з вбудованими інструментами для більшості задач без потреби у сторонніх бібліотеках.
- Сувора архітектурна модель забезпечує узгодженість кодової бази у великих командах.
- Ефективна система ін'єкції залежностей спрощує тестування та розділення відповідальностей.

Недоліки:

- Висока крива навчання: розробнику потрібно освоїти значний обсяг Angular-специфічних концепцій — модулі, декоратори, DI, RxJS.
- Більший розмір початкового бандлу порівняно з React і Vue, що впливає на час першого завантаження.
- Помітний обсяг шаблонного коду навіть для простих компонентів.

Vue.js — прогресивний фреймворк для побудови інтерфейсів, що поєднує реактивну систему на основі Proxy, компонентну архітектуру та Composition API. Vue розрахований на поступове впровадження: його можна використовувати як для окремих інтерактивних елементів сторінки, так і для повноцінних односторінкових застосунків.

Переваги:

- Плавна крива навчання і зрозумілий синтаксис, що знижує поріг входу для нових розробників.
- Чітка і детальна офіційна документація, яка вважається одним із галузевих еталонів.

- Гнучкість: Vue можна поступово впроваджувати в наявні проєкти без повного переписування.

Недоліки:

- Екосистема бібліотек і компонентів суттєво менша порівняно з React, що обмежує вибір готових рішень.
- Інтеграція Vue з повностековими Node.js-фреймворками (Nuxt.js) є менш органічною порівняно з парою React + Next.js.
- Підтримка TypeScript, попри вдосконалення у Vue 3, залишається менш зрілою порівняно з React та Angular.

React — декларативна JavaScript-бібліотека для побудови інтерфейсів від Meta, заснована на концепції однонаправленого потоку даних і компонентній архітектурі. React 19 вводить React Server Components як нативну концепцію: компоненти рендеряться на сервері й передають готовий HTML клієнту без додаткового JavaScript у браузері.

Переваги:

- Найбільша екосистема бібліотек і компонентів серед усіх клієнтських фреймворків.
- Нативна підтримка React Server Components у React 19 забезпечує ефективний серверний рендеринг.
- Органічна інтеграція з Next.js, де Server Components і Server Actions є першокласними концепціями.
- Широка спільнота і великий ринок розробників.

Недоліки:

- React є бібліотекою, а не фреймворком, тому маршрутизацію, управління станом та інші архітектурні компоненти потрібно обирати й налаштовувати самостійно.
- Велика кількість патернів і підходів — hooks, context, server/client components — потребує глибокого розуміння для коректного застосування.

З урахуванням викладених вище результатів аналізу, вибір React зумовлений насамперед тісною нативною інтеграцією з Next.js, де React Server Components є першокласним механізмом, а не зовнішнім доповненням. Це дозволяє виконувати запити до бази даних безпосередньо у серверних компонентах, суттєво спрощуючи архітектуру порівняно з Angular або Vue, де аналогічний підхід потребував би значно більших інтеграційних зусиль. Найбагатша екосистема бібліотек, своєю чергою, забезпечує зрілі та перевірені рішення для управління формами, серверним і клієнтським станом.

2.8 Огляд бібліотек та фреймворків для стилізації інтерфейсу користувача

Підхід до стилізації інтерфейсу визначає продуктивність розробки, узгодженість дизайн-системи та доступність (accessibility) компонентів. Нижче наведено порівняльний аналіз поширених інструментів.

Bootstrap — один із найстаріших і найвідоміших CSS-фреймворків з відкритим вихідним кодом, що надає готову систему сітки, типографіку та широкий набір компонентів із JavaScript-поведінкою. Bootstrap орієнтований на швидку розробку адаптивних інтерфейсів без написання значного обсягу власних стилів.

Переваги:

- Швидкий старт: готові компоненти — модальні вікна, картки, навігація, форми — доступні одразу без додаткового налаштування.
- Широка документація і велика спільнота, що полегшує пошук рішень.
- Висока сумісність із різними браузерами.

Недоліки:

- Генерує значний обсяг невикористаного CSS, що негативно впливає на розмір бандлу навіть при застосуванні purge-інструментів.

- Застосунки на Bootstrap мають характерний однотипний вигляд, а адаптація під нестандартний дизайн потребує суттєвого перевизначення стилів.
- JavaScript-компоненти Bootstrap маніпулюють DOM безпосередньо, що може конфліктувати з React-моделлю управління DOM.

Material UI (MUI) — комплексна бібліотека React-компонентів, що реалізує специфікацію Material Design від Google. MUI надає широкий набір готових компонентів, розвинену систему тем на основі JavaScript-об'єктів і CSS-in-JS стилізацію через emotion або styled-components.

Переваги:

- Великий набір готових компонентів, що покриває більшість типових потреб інтерфейсу без написання власних.
- Розвинена система тем дозволяє централізовано змінювати кольори, типографіку та відступи.
- Повна підтримка TypeScript з автодоповненням пропсів компонентів.

Недоліки:

- Нав'язує характерну естетику Material Design, відступ від якої потребує значних зусиль з кастомізації.
- CSS-in-JS генерує стилі під час виконання у браузері, що може знижувати продуктивність і ускладнює роботу з React Server Components.
- Великий розмір бібліотеки суттєво збільшує клієнтський бандл навіть при використанні tree-shaking.

Ant Design — корпоративна дизайн-система і бібліотека React-компонентів від Alibaba, орієнтована на розробку бізнес-застосунків. Ant Design пропонує багатий набір складних компонентів для роботи з даними — таблиці, дерева, складні форми, графіки — і особливо популярна в китайськомовній розробницькій спільноті.

Переваги:

- Найбагатший набір компонентів для бізнес-інтерфейсів серед розглянутих бібліотек, зокрема складні таблиці, деревоподібні структури та розширені форми.
- Вбудована підтримка інтернаціоналізації (i18n) для понад 50 мов.
- Детальна документація з великою кількістю прикладів.

Недоліки:

- Нав'язує корпоративну естетику, що ускладнює реалізацію нестандартного або сучасного дизайну.
- Один із найважчих за розміром бандлу варіантів серед розглянутих альтернатив.
- Компоненти активно використовують React Context та ефекти, що суттєво обмежує їх застосування у React Server Components.

Tailwind CSS + Radix UI + shadcn/ui — це стек, де кожен інструмент відповідає за свою чітку зону відповідальності. Tailwind CSS — utility-first CSS-фреймворк, що надає низькорівневі утилітарні класи для стилізації безпосередньо у розмітці. Radix UI — бібліотека безстильових (unstyled) доступних React-примітивів із коректною ARIA-розміткою та складною поведінкою компонентів. shadcn/ui поєднує обидві технології: надає набір компонентів, побудованих на Radix UI і стилізованих через Tailwind CSS, вихідний код яких копіюється безпосередньо до проєкту.

Переваги:

- Tailwind CSS з механізмом purge видаляє невикористані стилі при збірці, забезпечуючи мінімальний розмір CSS-файлу.
- Radix UI забезпечує коректну реалізацію доступності — управління фокусом, ARIA-атрибути, клавіатурна навігація — без необхідності писати цю логіку вручну.

- Копіювання вихідного коду shadcn/ui компонентів до проєкту дає повний контроль над їх реалізацією та усуває залежність від зовнішнього API бібліотеки.
- Відсутність runtime-стилізації: усі стилі є статичними CSS-класами, що повністю сумісно з React Server Components.

Недоліки:

- Utility-first підхід призводить до великої кількості CSS-класів безпосередньо у JSX, що знижує читабельність розмітки.
- Для впевненої роботи з Tailwind CSS необхідно засвоїти повний набір утилітарних класів, що потребує певного часу.

Зважаючи на результати вищенаведеного аналізу, стек Tailwind CSS + Radix UI + shadcn/ui є оптимальним з кількох причин. По-перше, Tailwind CSS з PurgeCSS видаляє невикористані стилі при збірці, забезпечуючи мінімальний розмір CSS-файлу. По-друге, на відміну від Bootstrap та MUI, shadcn/ui не нав'язує фіксованої естетики — CSS-змінні Tailwind дозволяють гнучко змінювати тему. По-третє, Radix UI бере на себе коректну реалізацію доступності — управління фокусом, ARIA-атрибути, клавіатурну навігацію — без необхідності писати цю логіку вручну. По-четверте, копіювання вихідного коду shadcn/ui компонентів до проєкту дає повний контроль над їх реалізацією і знімає залежність від зовнішніх API бібліотеки. По-п'яте, utility-first підхід Tailwind органічно поєднується з компонентною моделлю React: стилі та розмітка знаходяться в одному файлі, що спрощує рефакторинг.

2.9 Висновки з розділу 2

У другому розділі описано та спроєктовано програмний застосунок системи управління шкільним харчуванням.

Застосунок реалізовано як повностекову веб-систему на основі Next.js 16 з архітектурним патерном feature-based, що забезпечує чіткий поділ відповідальностей між модулями і спрощує подальше масштабування

системи. Механізм Server Actions дозволяє виконувати мутації даних безпечно на стороні сервера без необхідності проєктувати окремий API-шар. Ключова бізнес-логіка переходів між станами замовлень та оплат формалізована у вигляді машин станів, а права доступу централізовані в policy-класах — це робить поведінку системи передбачуваною і придатною до верифікації.

Порівняльний аналіз технологій у розділах 2.5–2.8 підтверджує обґрунтованість обраного стеку. PostgreSQL забезпечує точне зберігання фінансових даних і цілісність реляційної моделі через ACID-транзакції та зовнішні ключі. Next.js усуває потребу у двох окремих кодових базах і забезпечує наскрізну типобезпеку між клієнтом і сервером. React у поєднанні з Server Components надає ефективну модель рендерингу з прямим доступом до бази даних на рівні компонентів. Tailwind CSS разом із Radix UI і shadcn/ui формує гнучку, доступну та легко розширювану дизайн-систему без нав'язування фіксованої естетики.

3 СТРУКТУРА ПРОЄКТУ ВЕБ-ЗАСТОСУНКУ ДЛЯ ЗАМОВЛЕННЯ ЇЖІ

3.1 Структура бази даних проєкту

Для зберігання даних застосунку використовується реляційна СУБД PostgreSQL. Взаємодія з базою даних здійснюється через ORM Prisma, який забезпечує типобезпечний доступ до даних і дозволяє описувати схему у вигляді єдиного декларативного файлу. Схему БД наведено у лістингу 1.

Лістинг *1 – Схема бази даних*

```
enum Role {
  employee
  admin

  @@map("role")
}

enum OrgType {
  school
  supplier

  @@map("org_type")
}

enum OrderStatus {
  draft
  published
  accepted
  in_progress
  completed
  cancelled

  @@map("order_status")
}

enum PaymentStatus {
  unpaid
  paid
  verified

  @@map("payment_status")
```

```
}

model Organization {
  id             String   @id
@default(dbgenerated("gen_random_uuid()")) @map("id")
@db.Uuid
  name           String   @unique @map("name")
@db.VarChar(255)
  type           OrgType  @map("type")
  contactEmail   String?  @unique @map("contact_email")
@db.VarChar(255)
  contactPhone   String?  @unique @map("contact_phone")
@db.VarChar(50)
  users          User[]   @relation("OrganizationUsers")
  schoolOrders   Order[]  @relation("SchoolOrders")
  supplierOrders Order[]  @relation("SupplierOrders")
  createdAt      DateTime @default(now())
@map("created_at")
  updatedAt      DateTime @updatedAt @map("updated_at")

  @@map("organizations")
}

model User {
  id                     String                  @id
@default(dbgenerated("gen_random_uuid()")) @map("id")
@db.Uuid
  organizationId         String?
@map("organization_id") @db.Uuid
  organization           Organization?
@relation("OrganizationUsers", fields: [organizationId],
references: [id], onDelete: SetNull)
  email                  String                  @unique
@map("email") @db.VarChar(255)
  passwordHash           String
@map("password_hash") @db.VarChar(255)
  firstName              String?
@map("first_name") @db.VarChar(255)
  lastName               String?
@map("last_name") @db.VarChar(255)
  avatarUrl              String?
@map("avatar_url") @db.Text
  role                   Role
@map("role")
  orderStatusHistories   OrderStatusHistory[]
@relation("OrderStatusHistoryActors")
  orderPaymentHistories  OrderPaymentHistory[]
@relation("OrderPaymentHistoryActors")
```

```
  createdAt              DateTime
@default(now()) @map("created_at")
  updatedAt              DateTime               @updatedAt
@map("updated_at")

  @@index([organizationId])
  @@map("users")
}

model Category {
  id          String     @id
@default(dbgenerated("gen_random_uuid()")) @map("id")
@db.Uuid
  name        String     @unique @map("name")
@db.VarChar(255)
  description String?    @map("description") @db.Text
  menuItems   MenuItem[] @relation("CategoryMenuItems")
  createdAt   DateTime   @default(now()) @map("created_at")
  updatedAt   DateTime   @updatedAt @map("updated_at")

  @@map("categories")
}

model MenuItem {
  id          String      @id
@default(dbgenerated("gen_random_uuid()")) @db.Uuid
  categoryId  String?     @map("category_id") @db.Uuid
  category    Category?   @relation("CategoryMenuItems",
fields: [categoryId], references: [id], onDelete: SetNull)
  name        String      @unique @map("name")
@db.VarChar(255)
  description String?     @map("description") @db.Text
  price       Decimal     @map("price") @db.Decimal(10, 2)
  isAvailable Boolean     @default(true)
@map("is_available")
  imageUrl    String?     @map("image_url") @db.Text
  orderItems  OrderItem[] @relation("OrderItemMenuItems")
  createdAt   DateTime    @default(now())
@map("created_at")
  updatedAt   DateTime    @updatedAt @map("updated_at")

  @@index([categoryId])
  @@map("menu_items")
}

model Order {
  id                     String                  @id
@default(dbgenerated("gen_random_uuid()")) @db.Uuid
```

```
  schoolId              String?
@map("school_id") @db.Uuid
  school                Organization?
@relation("SchoolOrders", fields: [schoolId], references:
[id], onDelete: SetNull)
  supplierId            String?
@map("supplier_id") @db.Uuid
  supplier              Organization?
@relation("SupplierOrders", fields: [supplierId],
references: [id], onDelete: SetNull)
  deliveryDate          DateTime?
@map("delivery_date") @db.Date
  orderStatus           OrderStatus
@default(draft) @map("status")
  paymentStatus         PaymentStatus
@default(unpaid) @map("payment_status")
  totalPrice            Decimal                @default(0)
@map("total_price") @db.Decimal(10, 2)
  comment               String?
@map("comment") @db.Text
  orderItems            OrderItem[]
@relation("OrderOrderItems")
  orderStatusHistories  OrderStatusHistory[]
@relation("OrderStatusHistoryOrders")
  orderPaymentHistories OrderPaymentHistory[]
@relation("OrderPaymentHistoryOrders")
  createdAt             DateTime
@default(now()) @map("created_at")
  publishedAt           DateTime?
@map("published_at")
  updatedAt             DateTime               @updatedAt
@map("updated_at")

  @@index([orderStatus])
  @@index([paymentStatus])
  @@index([schoolId])
  @@index([supplierId])
  @@map("orders")
}

model OrderItem {
  id         String    @id
@default(dbgenerated("gen_random_uuid()")) @map("id")
@db.Uuid
  orderId    String    @map("order_id") @db.Uuid
  order      Order     @relation("OrderOrderItems", fields:
[orderId], references: [id], onDelete: Cascade)
  menuItemId String?   @map("menu_item_id") @db.Uuid
```

```
  menuItem   MenuItem? @relation("OrderItemMenuItems",
fields: [menuItemId], references: [id], onDelete: SetNull)
  quantity   Int       @default(1) @map("quantity")
  price      Decimal   @map("price") @db.Decimal(10, 2)
  addedAt    DateTime  @default(now()) @map("added_at")
  updatedAt  DateTime  @updatedAt @map("updated_at")

  @@index([orderId])
  @@index([menuItemId])
  @@map("order_items")
}

model OrderStatusHistory {
  id             String      @id
@default(dbgenerated("gen_random_uuid()")) @map("id")
@db.Uuid
  orderId        String      @map("order_id") @db.Uuid
  order          Order
@relation("OrderStatusHistoryOrders", fields: [orderId],
references: [id], onDelete: Cascade)
  previousStatus OrderStatus @map("previous_status")
  newStatus      OrderStatus @map("new_status")
  actorId        String?     @map("actor_id") @db.Uuid
  actor          User?
@relation("OrderStatusHistoryActors", fields: [actorId],
references: [id], onDelete: SetNull)
  changedAt      DateTime    @default(now())
@map("changed_at")

  @@index([orderId])
  @@index([actorId])
  @@map("order_status_history")
}

model OrderPaymentHistory {
  id             String        @id
@default(dbgenerated("gen_random_uuid()")) @map("id")
@db.Uuid
  orderId        String        @map("order_id") @db.Uuid
  order          Order
@relation("OrderPaymentHistoryOrders", fields: [orderId],
references: [id], onDelete: Cascade)
  previousStatus PaymentStatus @map("previous_status")
  newStatus      PaymentStatus @map("new_status")
  actorId        String?       @map("actor_id") @db.Uuid
  actor          User?
@relation("OrderPaymentHistoryActors", fields: [actorId],
references: [id], onDelete: SetNull)
```

```
  changedAt      DateTime      @default(now())
@map("changed_at")

  @@index([orderId])
  @@index([actorId])
  @@map("order_payment_history")
}
```

База даних містить вісім таблиць та чотири перелічувальні типи (enum), перелік яких наведено у таблиці 2.

Таблиця 2 — Перелічувальні типи схеми бази даних

Тип	**Значення**
Role	employee, admin
OrgType	school, supplier
OrderStatus	draft, published, accepted, in_progress, completed, cancelled
PaymentStatus	unpaid, paid, verified

Таблиця **organizations** зберігає дані про організації, що беруть участь у системі, — школи та компанії-постачальники. Тип організації визначається полем ***type*** зі значеннями enum ***OrgType***. Одна організація може мати багато користувачів, а також виступати школою-замовником або постачальником у замовленнях.

Таблиця **users** зберігає облікові записи користувачів системи. Кожен користувач належить до певної організації та має одну з двох ролей: ***employee*** або ***admin***. Пароль зберігається виключно у вигляді bcrypt-хешу в полі ***password_hash***.

Таблиця **categories** зберігає категорії позицій меню (наприклад, «Основні страви», «Напої», «Десерти»). Категорії використовуються для групування та фільтрації позицій під час їх відображення.

Таблиця **menu_items** зберігає страви й продукти, доступні для включення до замовлень. Кожна позиція належить до однієї категорії. Поле

is_available дозволяє тимчасово приховати позицію без її видалення. У разі видалення категорії поле ***category_id*** встановлюється в NULL.

Таблиця **orders** є центральною сутністю схеми: вона пов'язує школу з постачальником у межах конкретного замовлення. Поле ***supplier_id*** є необов'язковим — воно заповнюється лише після того, як постачальник прийме замовлення. Статуси замовлення та оплати відстежуються окремими полями з відповідними enum-типами.

Таблиця **order_items** зберігає позиції кожного замовлення — конкретні страви та їх кількість. Ціна фіксується в момент додавання позиції до замовлення, завдяки чому дані залишаються коректними навіть після подальшої зміни ціни в меню. При видаленні замовлення всі його позиції видаляються каскадно.

Таблиця **order_status_history** є журналом аудиту змін статусу замовлення. Кожен запис фіксує попередній і новий статус, час зміни та ідентифікатор користувача, який ініціював зміну. При видаленні замовлення всі пов'язані записи видаляються каскадно.

Таблиця **order_payment_history** аналогічна до **order_status_history**, але призначена для відстеження змін статусу оплати замовлення. Структура таблиці, логіка каскадного видалення та індекси є ідентичними.

Зв'язки між усіма таблицями реалізовано за принципом «один до багатьох». Загальну структуру зв'язків між таблицями відображено на рисунку 9.

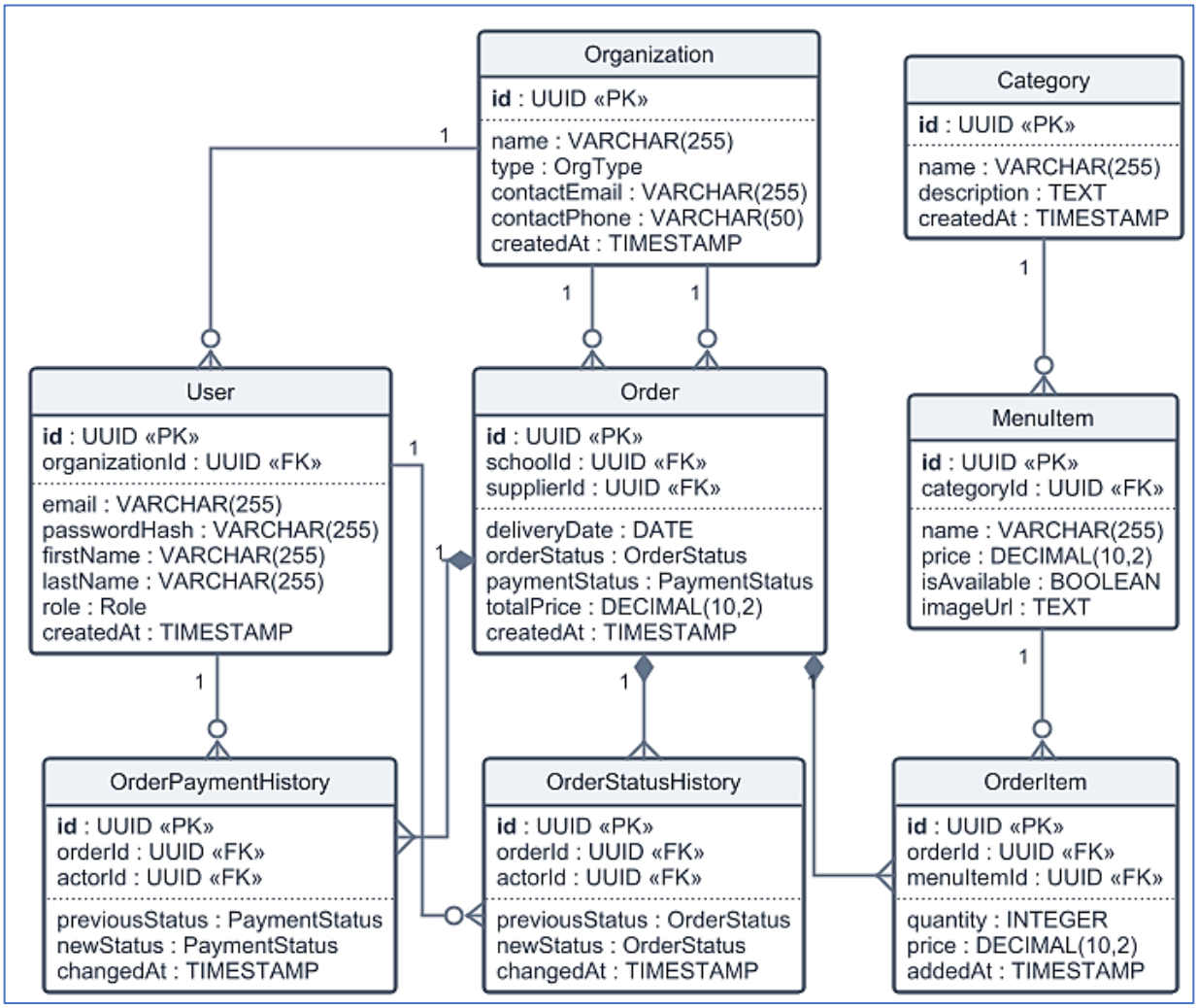


Рисунок *9 — ER-діаграма бази даних*

3.2 Клієнтська частина проєкту

Клієнтська частина застосунку побудована на основі React 19 та Next.js 16 з використанням App Router. Ключовою особливістю підходу є поділ компонентів на серверні та клієнтські. Серверні компоненти виконуються виключно на сервері, мають прямий доступ до функцій отримання даних і не потрапляють до JavaScript-бандлу браузера. Клієнтські компоненти, позначені директивою "use client", виконуються у браузері та використовуються лише там, де необхідна інтерактивність: обробка подій форм, локальний стан, навігаційні хуки. Стилізація реалізована засобами Tailwind CSS у поєднанні з бібліотекою Radix UI, що надає доступні та семантично коректні примітиви інтерфейсу.

Усі захищені сторінки застосунку вкладені у серверний компонент ProtectedLayout (наведений в лістингу 2), що формує загальну структуру: заголовок (Header), бічна панель навігації (Sidebar), область основного контенту та підвал (Footer). Також тут підключається система toast-сповіщень Toaster.

Лістинг *2 — Серверний компонент ProtectedLayout*

```
export default async function ProtectedLayout({
  children,
}: {
  children: React.ReactNode;
}) {
  return (
    <div className="relative flex min-h-screen flex-col">
      <Header />
      <div className="flex flex-1">
        <Sidebar />
        <main className="flex-1 bg-background overflow-x-
hidden">
          <div className="container mx-auto p-4 md:p-6 max-
w-[100vw]">
            {children}
          </div>
        </main>
      </div>
```

```
      <Footer />
      <Toaster richColors position="top-center" closeButton
/>
    </div>
  );
}
```

Header, наведений в лістингу 3, є клієнтським компонентом. Він відображає логотип застосунку та спадне меню поточного користувача з його ім'ям, аватаром та кнопкою виходу із системи. Для отримання даних сесії використовується хук useSession. Також Header містить кнопку розгортання бічної панелі на мобільних пристроях.

Лістинг 3 — Клієнтський компонент Header

```
export default function Header() {
  const { data: session } = useSession();
  const setOpen = useSidebarStore((s) => s.setOpen);

  const userInitials = session?.user?.name
    ? session.user.name
        .split(" ")
        .map((n) => n[0])
        .join("")
        .toUpperCase()
    : "Г";

  return (
    <header className="sticky top-0 z-50 w-full border-b
bg-background/95 backdrop-blur supports-backdrop-filter:bg-
background/60">
      <div className="mx-8 flex h-14 items-center justify-
between">
        <Button
          variant="outline"
          size="icon"
          className="md:hidden mr-3"
          aria-label="Відкрити меню"
          onClick={() => setOpen(true)}
        >
          <Menu className="h-5 w-5" />
        </Button>
        <Link href="/" className="hidden md:block">
          <span className="font-bold text-xl text-primary">
            Шкільне харчування
          </span>
        </Link>
```

```
        {session ? (
          <DropdownMenu>
            <DropdownMenuTrigger asChild>
              <Button
                variant="ghost"
                className="relative h-8 w-8 rounded-full"
                aria-label="Відкрити меню користувача"
              >
                <Avatar className="h-8 w-8">
                  <AvatarImage
                    src={session.user?.image ?? undefined}
                    alt="avatar"
                  />

<AvatarFallback>{userInitials}</AvatarFallback>
                </Avatar>
              </Button>
            </DropdownMenuTrigger>
            <DropdownMenuContent className="w-56"
align="end" forceMount>
              <DropdownMenuLabel className="font-normal">
                <div className="flex flex-col space-y-1">
                  <p className="text-sm font-medium
leading-none">
                    {session.user?.name || "Користувач"}
                  </p>
                  <p className="text-xs leading-none text-
muted-foreground">
                    {session.user?.email}
                  </p>
                </div>
              </DropdownMenuLabel>
              <DropdownMenuSeparator />
              <DropdownMenuItem asChild>
                <Link href="/profile">Профіль</Link>
              </DropdownMenuItem>
              <DropdownMenuItem
                className="text-red-600 focus:text-red-600"
                onClick={() => signOut({ callbackUrl:
"/auth/login" })}
              >
                Вийти
              </DropdownMenuItem>
            </DropdownMenuContent>
          </DropdownMenu>
        ) : (
          <Button asChild variant="outline" aria-
label="Увійти в систему">
```

```
            <Link href="/auth/login">Увійти</Link>
          </Button>
        )}
      </div>
    </header>
  );
}
```

Sidebar є клієнтським компонентом і реалізує основну навігацію застосунку. Перелік навігаційних посилань формується динамічно на основі ролі та типу організації поточного користувача: адміністратор бачить розділи управління організаціями, користувачами, категоріями, меню та замовленнями; школа — панель, меню, власні замовлення; постачальник — панель, опубліковані замовлення, власні активні замовлення. Активний пункт меню визначається за допомогою хуку usePathname. Крім навігаційних посилань, бічна панель містить кнопку «Створити замовлення», що безпосередньо викликає серверну дію createOrder — ця кнопка відображається виключно для школи. Стан розгортання бічної панелі на мобільних пристроях зберігається у Zustand-сторі useSidebarStore, наведеному в лістингу 4, з персистенцією у localStorage.

Лістинг *4 — Zustand-стор useSidebarStore*

```
type SidebarState = {
  open: boolean;
  setOpen: (v: boolean) => void;
  toggle: () => void;
};

const useSidebarStore = create<SidebarState>()(
  persist(
    (set) => ({
      open: false,
      setOpen: (v: boolean) => set({ open: v }),
      toggle: () => set((s) => ({ open: !s.open })),
    }),
    {
      name: "sidebar-storage",
    },
  ),
```

```
);

export default useSidebarStore;
```

Усі сторінки застосунку є асинхронними серверними компонентами. Дані отримуються безпосередньо у тілі компонента через виклики функцій запитів — getAllOrders, getMyOrganizationOrders, getMyOrganizationStats, getAllMenuItems тощо. Параметри фільтрації та пагінації передаються через URL-параметри (searchParams), які сторінка отримує як проп і передає у відповідні функції запитів. Такий підхід забезпечує повноцінний серверний рендеринг без необхідності клієнтської бібліотеки для отримання даних.

Усі форми застосунку побудовані на бібліотеці React Hook Form у поєднанні з Zod-схемами для валідації через zodResolver. Інтеграція з Next.js Server Actions реалізована через хук useActionState, що дозволяє відстежувати стан виконання серверної дії та відображати помилки — як загальні, так і на рівні окремих полів. Після успішного виконання дії більшість форм викликає router.push або router.refresh для оновлення серверного стану сторінки.

Форми, що підтримують завантаження файлів (аватар користувача, зображення страви), реалізують попередній перегляд через URL-об'єкти (URL.createObjectURL) та клієнтську валідацію розміру і формату файлу до відправлення на сервер.

LoginForm є клієнтським компонентом і реалізує форму входу з полями email та пароля. Для аутентифікації використовується функція signIn з бібліотеки NextAuth. При успішному вході виконується переспрямування на відповідну панель залежно від ролі користувача. RegisterForm аналогічно реалізує форму реєстрації і викликає серверну дію registerUser. Обидві форми використовують React Hook Form із Zod-валідацією.

Адміністративна панель AdminDashboardPage паралельно отримує дані через getOrganizationsOverview та getAllOrdersStats і відображає шість карток ключових показників, таблицю останніх замовлень та рейтинги організацій. Решта адміністративних сторінок побудовані за єдиним шаблоном: серверний

компонент-сторінка отримує дані та передає їх у клієнтські компоненти фільтрів та таблиці.

OrganizationsTable, UsersTable, CategoriesTable, MenuItemsTable — клієнтські компоненти таблиць із підтримкою видалення записів безпосередньо з таблиці (через виклик відповідної серверної дії з підтвердженням у AlertDialog). Компоненти фільтрів — OrganizationsFilters, UsersFilters, CategoriesFilters, MenuItemsFilters — реалізують пошук та фільтрацію через оновлення URL-параметрів за допомогою useRouter та useSearchParams. MenuCategoryTabs відображає горизонтальну панель вкладок для фільтрації страв за категорією.

SchoolDashboardPage відображає персоналізовану статистику через getMyOrganizationStats: компонент StatsCards з кількістю активних, майбутніх та неоплачених замовлень. Сторінки списку замовлень (SchoolOrdersPage, SchoolOrdersDraftsPage) використовують відповідно OrderTable для табличного подання та OrderRows — для подання у вигляді акордеону з можливістю розгортання деталей.

Ключовим компонентом розділу є SchoolOrderDetailsCard — клієнтський компонент детального перегляду замовлення. Він відображає реквізити замовлення та набір кнопок дій, що визначається поточним статусом: для чернетки доступні редагування, публікація та видалення; для прийнятого та неоплаченого — позначення оплаченим; для замовлення у процесі — підтвердження доставки. Кожна дія з наслідками (видалення, публікація, скасування) захищена підтверджуючим діалогом AlertDialog. Компонент StatusHistory відображає повну хронологію змін статусів замовлення.

SupplierDashboardPage аналогічно до школи використовує getMyOrganizationStats та StatsCards, але відображає лише статуси, релевантні для постачальника. SupplierPublishedOrdersPage отримує список опублікованих замовлень через getPublishedOrders та відображає їх у вигляді акордеону OrderRows.

SupplierOrderDetailsCard є клієнтським компонентом, симетричним до SchoolOrderDetailsCard, але з набором дій постачальника: прийняти опубліковане замовлення; підтвердити оплату; розпочати виконання (лише якщо оплату підтверджено); скасувати прийняте замовлення (лише якщо не оплачено).

ProfilePage відображає дані поточного користувача через ProfileInfo (серверний компонент для перегляду) та надає ProfileForm для їх редагування. ProfileForm підтримує зміну імені, email, пароля та аватара з попереднім переглядом зображення. Після успішного збереження викликається useUpdateSession для синхронізації оновлених даних у JWT-сесії без повторного входу.

OrderTable є клієнтським компонентом, що використовується на сторінках замовлень як адміністратора, так і школи та постачальника. Залежно від переданих пропсів organizationType та role він формує правильне посилання на детальну сторінку замовлення.

Pagination є клієнтським компонентом, що приймає поточну сторінку, загальну кількість сторінок і записів та навігує між ними через оновлення URL-параметра page, зберігаючи при цьому всі інші активні фільтри.

3.3 Серверна частина проєкту

Серверну частину застосунку реалізовано засобами Next.js Server Actions — механізму, за якого серверні функції з директивою "use server" можуть викликатися безпосередньо з клієнтських компонентів. Такий підхід усуває необхідність у явному визначенні REST API і дозволяє повністю контролювати виконання серверного коду в межах єдиної кодової бази. Серверну частину структуровано за принципом Feature-based Architecture: кожен функціональний модуль є самодостатнім і містить власні сервіси, функції запитів та серверні дії. Усі модулі спираються на спільну

інфраструктуру: систему обробки помилок, уніфікований тип результату серверної дії та клієнт бази даних.

Усі серверні дії повертають уніфікований тип ActionResult<T> — об'єднання двох варіантів: успішного (з необов'язковими полями data та message) та помилкового (з рядком помилки, кодом та необов'язковим об'єктом fieldErrors для помилок конкретних полів форми). Такий підхід гарантує єдину структуру відповіді незалежно від модуля. Тип ActionResult<T> наведено в лістингу 5.

Лістинг *5 — Тип результату серверної дії*

```
export type ActionResult<T = unknown> =
  | { success: true; data?: T; message?: string }
  | {
      success: false;
      error: string;
      code?: string;
      fieldErrors?: Record<string, string[]>;
    };
```

Функція getCurrentUser отримує дані поточного користувача з JWT-сесії та повертає об'єкт типу CurrentUser з ідентифікатором, email, роллю, а також ідентифікатором і типом організації. Якщо активна сесія відсутня — генерується UnauthorizedError. Функція викликається на початку кожної серверної дії, що потребує аутентифікації.

Автентифікацію реалізовано за допомогою NextAuth.js з JWT-стратегією сесій та постачальником типу Credentials. Конфігурацію визначено в об'єкті authOptions і вона включає: постачальника, що перевіряє email та пароль відносно хешу в базі даних; JWT callback, що заповнює токен розширеними даними користувача (роль, ідентифікатор і тип організації) під час входу та підтримує їх оновлення; session callback, що відображає поля токена в об'єкт сесії. Конфігурацію NextAuth.js наведено в лістингу 6.

Лістинг *6 — Конфігурація аутентифікації NextAuth.js*

```
export const authOptions: AuthOptions = {
  adapter: PrismaAdapter(prisma),
  providers: [
    CredentialsProvider({
      name: "Email",
      credentials: {
        email: { label: "Email", type: "text" },
        password: { label: "Password", type: "password" },
      },
      async authorize(credentials) {
        const data = loginSchema.parse(credentials);

        const user = await prisma.user.findUnique({
          where: { email: data.email },
          include: { organization: true },
        });

        if (!user) return null;
        const valid = await bcrypt.compare(data.password,
user.passwordHash);
        if (!valid) return null;

        return {
          id: user.id,
          email: user.email,
          name: user.firstName + " " + user.lastName,
          image: user.avatarUrl || null,
          organizationId: user.organizationId,
          organizationType: user.organization?.type ??
null,
          role: user.role,
        };
      },
    }),
  ],
  pages: {
    signIn: "/auth/login",
  },
  session: {
    strategy: "jwt",
    maxAge: 30 * 24 * 60 * 60,
  },
  cookies: {
    sessionToken: {
      name:
        process.env.NODE_ENV === "production"
          ? "__Secure-next-auth.session-token"
```

```
          : "next-auth.session-token",
      options: {
        httpOnly: true,
        sameSite: "lax",
        path: "/",
        secure: process.env.NODE_ENV === "production",
      },
    },
  },
  callbacks: {
    async jwt({
      token,
      user,
      trigger,
      session,
    }: {
      token: JWT;
      user?: User;
      trigger?: "signIn" | "signUp" | "update";
      session?: SessionUpdateData;
    }) {
      if (user) {
        token.id = user.id;
        token.email = user.email!;
        token.name = user.name ?? null;
        token.image = user.image ?? null;
        token.role = user.role;
        token.organizationId = user.organizationId;
        token.organizationType = user.organizationType;
      }

      if (trigger === "update" && session) {
        if (session.name !== undefined) token.name =
session.name;
        if (session.email !== undefined) token.email =
session.email;
        if (session.image !== undefined) token.image =
session.image;
        if (session.role !== undefined) token.role =
session.role;
        if (session.organizationId !== undefined)
          token.organizationId = session.organizationId;
        if (session.organizationType !== undefined)
          token.organizationType =
session.organizationType;
      }

      return token;
    },
```

```
    session({ session, token }: { session: Session; token:
JWT }) {
      if (token?.id) {
        session.user.id = token.id;
        session.user.email = token.email;
        session.user.name = token.name ?? null;
        session.user.image = token.image ?? null;
        session.user.role = token.role;
        session.user.organizationId = token.organizationId
?? null;
        session.user.organizationType =
token.organizationType ?? null;
      }
      return session;
    },
  },
  secret: process.env.NEXTAUTH_SECRET,
};
```

Реєстрація нових користувачів виконується через серверну дію registerUser, яка валідує дані за схемою registerSchema та викликає AuthService.register. Цей метод перевіряє унікальність email, верифікує існування вказаної організації, хешує пароль та за наявності зберігає файл аватара на сервері. Схеми registerSchema та loginSchema описані через Zod і включають перевірку формату email та вимог до складності пароля. Схеми наведено в лістингу 7.

Лістинг *7 — Схеми валідації вхідних даних*

```
export const registerSchema = z
  .object({
    email: z
      .email("Неправильний формат електронної адреси")
      .transform((v) => v.trim().toLowerCase()),
    password: z
      .string()
      .min(8, "Пароль повинен містити щонайменше 8
символів")
      .regex(
        /(?=.*[a-z])(?=.*[A-Z])(?=.*\d)/,
        "Пароль повинен містити великі та малі літери та
цифру",
      ),
```

```
    firstName: z.string().trim().optional(),
    lastName: z.string().trim().optional(),
    organizationId: z.uuid("Неправильний формат ID
організації"),
    confirmPassword: z.string().optional(),
    avatar: z.instanceof(File).optional(),
  })
  .refine(
    (data) => {
      if ("confirmPassword" in data && data.confirmPassword
!== undefined) {
        return data.password === data.confirmPassword;
      }
      return true;
    },
    {
      message: "Паролі не співпадають",
      path: ["confirmPassword"],
    },
  );

export const loginSchema = z.object({
  email: z
    .email("Неправильний формат електронної адреси")
    .transform((v) => v.trim().toLowerCase()),
  password: z
    .string()
    .min(8, "Пароль повинен містити щонайменше 8 символів")
    .regex(
      /(?=.*[a-z])(?=.*[A-Z])(?=.*\d)/,
      "Пароль повинен містити великі та малі літери та
цифру",
    ),
});
```

OrganizationService реалізує повний цикл CRUD для організацій. При створенні та оновленні перевіряється унікальність назви, контактного email і телефону — порушення цих умов призводить до генерації ConflictError. Функція getOrganizationsOverview повертає агреговану статистику для адміністративної панелі: загальну кількість організацій із розбивкою за типом та рейтинги найактивніших шкіл і постачальників за обсягом замовлень.

Серверні дії createOrganization та deleteOrganization доступні виключно адміністраторам. Дія updateOrganization дозволяє адміністратору оновлювати

будь-яку організацію, а рядовому співробітнику — лише власну. Дію updateOrganization наведено в лістингу 8.

Лістинг *8 — Серверна дія updateOrganization*

```
export async function updateOrganization(
  prevState: ActionResult<OrganizationInfo> | null = null,
  formData: FormData | updateOrganizationInput,
): Promise<ActionResult<OrganizationInfo>> {
  try {
    const currentUser = await getCurrentUser();
    if (currentUser.role !== "admin" &&
!currentUser.organizationId) {
      return {
        success: false,
        error: "Відмовлено в доступі",
      };
    }
    const rawData =
      formData instanceof FormData ?
Object.fromEntries(formData) : formData;
    const result =
updateOrganizationSchema.safeParse(rawData);
    if (!result.success) {
      const flattened = z.flattenError(result.error);
      return {
        success: false,
        error: "Перевірте дані",
        fieldErrors: flattened.fieldErrors,
      };
    }
    const data = result.data;
    const organizationId = data.id ??
currentUser.organizationId;
    if (!organizationId) {
      return {
        success: false,
        error: "ID організації не вказано",
      };
    }
    if (
      currentUser.role !== "admin" &&
      currentUser.organizationId !== organizationId
    ) {
      return {
        success: false,
        error: "Відмовлено в доступі",
```

```
      };
    }
    const updatedOrganization = await
OrganizationService.update({
      id: organizationId,
      ...data,
    });
    updateTag("all-orgs");
    updateTag("all-users");
    revalidateTag("org-overview", "max");
    return { success: true, data: updatedOrganization };
  } catch (error: unknown) {
    return {
      success: false,
      error:
        error instanceof Error
          ? error.message
          : "Не вдалося оновити організацію",
    };
  }
}
```

UserService містить два методи. Метод update виконує умовне оновлення профілю: перевіряє унікальність нового email, хешує пароль при його зміні та зберігає файл аватара за наявності. Метод delete видаляє обліковий запис після перевірки його існування.

Функція getAllUsers, наведена в лістингу 9, доступна виключно адміністраторам і підтримує фільтрацію за організацією та іменем. Функція getUserById доступна адміністратору або самому користувачу — в інших випадках генерується AccessDeniedError. Серверна дія updateUser дозволяє адміністратору оновлювати будь-який профіль, а рядовому співробітнику — лише власний.

Лістинг *9 — Функція getAllUsers*

```
export async function getAllUsers(
  data: getAllUsersInput,
  currentUser: CurrentUser,
): Promise<UsersList> {
  if (currentUser.role !== "admin") {
    throw new AccessDeniedError("Access denied");
  }
  const validated = getAllUsersSchema.parse(data);
```

```
  const cacheKeyParts = [
    "all-users",
    validated.name || "no-name",
    validated.organizationId || "no-org",
    validated.page ? String(validated.page) : "1",
    validated.limit ? String(validated.limit) : "10",
  ];

  return unstable_cache(
    async () => {
      console.log(
        `[CACHE MISS/HIT CHECK] ${new Date().toISOString()}
— выполняю реальные запросы к БД`,
      );

      const page = validated.page && validated.page > 0 ?
validated.page : 1;
      const limit =
        validated.limit && validated.limit > 0 ?
validated.limit : 10;
      const skip = (page - 1) * limit;
      const existingOrg = validated.organizationId
        ? await prisma.organization.findUnique({
            where: { id: validated.organizationId },
          })
        : null;
      if (validated.organizationId && !existingOrg) {
        notFound();
      }

      const filters: Prisma.UserWhereInput[] = [];
      if (validated.organizationId) {
        filters.push({ organizationId:
validated.organizationId });
      }
      if (validated.name) {
        filters.push({
          OR: [
            {
              firstName: {
                contains: validated.name,
                mode: "insensitive",
              },
            },
            {
              lastName: {
                contains: validated.name,
                mode: "insensitive",
```

```
              },
            },
          ],
        });
      }

      const [users, total] = await Promise.all([
        prisma.user.findMany({
          where: filters.length > 0 ? { AND: filters } :
undefined,
          skip: skip,
          take: limit,
          select: {
            id: true,
            email: true,
            firstName: true,
            lastName: true,
            role: true,
            avatarUrl: true,
            organization: {
              select: {
                id: true,
                name: true,
                type: true,
                contactEmail: true,
                contactPhone: true,
              },
            },
          },
        }),
        prisma.user.count({
          where: filters.length > 0 ? { AND: filters } :
undefined,
        }),
      ]);
      const totalPages = Math.ceil(total / limit);
      return { users, total, page, totalPages };
    },
    cacheKeyParts,
    {
      revalidate: 300,
      tags: ["all-users"],
    },
  )();
}
```

CategoryService реалізує операції CRUD для категорій страв. При створенні та оновленні перевіряється унікальність назви. Функція getAllCategories підтримує фільтрацію за назвою та пагінацію. Усі мутуючі серверні дії — createCategory, updateCategory, deleteCategory — доступні виключно адміністраторам.

MenuItemService реалізує операції CRUD для страв. При створенні перевіряється унікальність назви та за наявності зберігається зображення. Метод update виконує умовне оновлення: при зміні назви перевіряє унікальність, при зміні категорії — верифікує її існування, при передачі нового зображення — зберігає файл. Ціна зберігається у типі Decimal і конвертується в число при поверненні клієнту. Функція getAllMenuItems підтримує фільтрацію за категорією, назвою та ознакою доступності. Усі мутуючі дії доступні виключно адміністраторам.

OrderItemService керує позиціями конкретного замовлення. Усі три операції — create, update, delete — виконуються в транзакціях Prisma, що гарантує атомарність: одночасно зі зміною позиції перераховується загальна сума замовлення (totalPrice). Кожна операція попередньо перевіряє, що замовлення перебуває у статусі draft, — редагування позицій в опублікованому або прийнятому замовленні не допускається. Арифметичні операції з цінами виконуються через методи типу Decimal, що дозволяє уникнути похибок арифметики з рухомою комою.

OrderService, наведений у лістингу 10, є найскладнішим сервісом системи. Кожен з його методів — create, update, updateOrderStatus, updatePaymentStatus, delete — починається з перевірки прав доступу через OrderPermissionPolicy. Методи updateOrderStatus та updatePaymentStatus додатково валідують допустимість переходу через відповідні машини станів і атомарно оновлюють статус та створюють запис у журналі аудиту.

Лістинг *10 — Сервіс OrderService*

```
export const OrderService = {
```

```
  async create(user: CurrentUser) {
    const permission =
OrderPermissionPolicy.canCreateOrder(user);
    if (!permission.allowed) {
      throw new Error(permission.reason || "Відмовлено в
доступі");
    }
    const existingOrg = await
prisma.organization.findUnique({
      where: { id: user.organizationId },
    });
    if (!existingOrg) {
      throw new Error("Організація не знайдена");
    }
    const order = await prisma.order.create({
      data: {
        schoolId: user.organizationId,
        deliveryDate: null,
        orderStatus: "draft",
        paymentStatus: "unpaid",
        totalPrice: 0,
        comment: null,
      },
      select: {
        id: true,
        deliveryDate: true,
        comment: true,
        orderStatus: true,
        paymentStatus: true,
        totalPrice: true,
        createdAt: true,
        publishedAt: true,
        school: {
          select: {
            id: true,
            name: true,
          },
        },
        supplier: {
          select: {
            id: true,
            name: true,
          },
        },
      },
    });

    return {
      ...order,
```

```
      totalPrice: Number(order.totalPrice),
    };
  },

  async update(data: updateOrderInput, user: CurrentUser) {
    const permission =
OrderPermissionPolicy.canUpdateOrder(user);
    if (!permission.allowed) {
      throw new Error(permission.reason || "Відмовлено в
доступі");
    }
    const order = await prisma.order.findUnique({
      where: { id: data.id },
    });
    if (!order) {
      throw new Error("Замовлення не знайдено");
    }
    if (order.orderStatus !== "draft") {
      throw new Error("Неможливо оновити замовлення, яке не
є чернеткою");
    }
    if (user.organizationId !== order.schoolId) {
      throw new Error(
        "Неможливо змінити замовлення, яке не належить
вашій школі",
      );
    }

    const updateData: Record<string, unknown> = {};
    if (data.deliveryDate !== undefined) {
      updateData.deliveryDate = data.deliveryDate;
    }
    if (data.comment !== undefined) {
      updateData.comment = data.comment ?? null;
    }
    if (Object.keys(updateData).length === 0) {
      throw new Error("Немає даних для оновлення");
    }

    const updatedOrder = await prisma.order.update({
      where: { id: data.id },
      data: updateData,
      select: {
        id: true,
        deliveryDate: true,
        comment: true,
        orderStatus: true,
        paymentStatus: true,
        totalPrice: true,
```

```
        createdAt: true,
        publishedAt: true,
        school: {
          select: {
            id: true,
            name: true,
          },
        },
        supplier: {
          select: {
            id: true,
            name: true,
          },
        },
      },
    });

    return {
      ...updatedOrder,
      totalPrice: Number(updatedOrder.totalPrice),
    };
  },

  async updateOrderStatus(data: updateOrderStatusInput,
user: CurrentUser) {
    const order = await prisma.order.findUnique({
      where: { id: data.id },
      include: {
        orderItems: {
          select: { id: true },
        },
      },
    });
    if (!order) {
      throw new Error("Замовлення не знайдено");
    }
    const permission =
OrderPermissionPolicy.canChangeStatus(user, order);
    if (!permission.allowed) {
      throw new Error(permission.reason || "Відмовлено в
доступі");
    }
    const { isSchool, isSupplier } = permission;
    const { allowed, reason, newSupplierId } =
canTransitionOrderStatus(
      order,
      data.status,
      isSchool,
      isSupplier,
```

```
      isSupplier ? user.organizationId : undefined,
    );
    if (!allowed) {
      throw new Error(reason || "Недопустима зміна статусу
замовлення");
    }
    const updatedOrder = await prisma.$transaction(async
(tx) => {
      const updated = await tx.order.update({
        where: { id: data.id },
        data: { orderStatus: data.status, supplierId:
newSupplierId },
        select: {
          id: true,
          deliveryDate: true,
          comment: true,
          orderStatus: true,
          paymentStatus: true,
          totalPrice: true,
          createdAt: true,
          publishedAt: true,
          school: {
            select: {
              id: true,
              name: true,
            },
          },
          supplier: {
            select: {
              id: true,
              name: true,
            },
          },
        },
      });

      await tx.orderStatusHistory.create({
        data: {
          orderId: data.id,
          previousStatus: order.orderStatus,
          newStatus: data.status,
          actorId: user.id,
        },
      });

      return updated;
    });

    return {
```

```
      ...updatedOrder,
      totalPrice: Number(updatedOrder.totalPrice),
    };
  },

  async updatePaymentStatus(data: updatePaymentStatusInput,
user: CurrentUser) {
    const order = await prisma.order.findUnique({
      where: { id: data.id },
    });
    if (!order) {
      throw new Error("Замовлення не знайдено");
    }
    const permission =
OrderPermissionPolicy.canChangeStatus(user, order);
    if (!permission.allowed) {
      throw new Error(permission.reason || "Відмовлено в
доступі");
    }
    const { isSchool, isSupplier } = permission;
    const { allowed, reason } = canTransitionPaymentStatus(
      order.paymentStatus,
      data.status,
      isSchool,
      isSupplier,
      order.orderStatus,
    );
    if (!allowed) {
      throw new Error(reason || "Недопустима зміна статусу
оплати");
    }
    const updatedOrder = await prisma.$transaction(async
(tx) => {
      const updated = await tx.order.update({
        where: { id: data.id },
        data: { paymentStatus: data.status },
        select: {
          id: true,
          deliveryDate: true,
          comment: true,
          orderStatus: true,
          paymentStatus: true,
          totalPrice: true,
          createdAt: true,
          publishedAt: true,
          school: {
            select: {
              id: true,
              name: true,
```

```
            },
          },
          supplier: {
            select: {
              id: true,
              name: true,
            },
          },
        },
      });

      await tx.orderPaymentHistory.create({
        data: {
          orderId: data.id,
          previousStatus: order.paymentStatus,
          newStatus: data.status,
          actorId: user.id,
        },
      });

      return updated;
    });

    return {
      ...updatedOrder,
      totalPrice: Number(updatedOrder.totalPrice),
    };
  },

  async delete(data: deleteOrderInput, user: CurrentUser) {
    const order = await prisma.order.findUnique({
      where: { id: data.id },
    });
    if (!order) {
      throw new Error("Замовлення не знайдено");
    }
    if (order.orderStatus !== "draft") {
      throw new Error("Лише чернетки можна видаляти");
    }
    const permission =
OrderPermissionPolicy.canDeleteOrder(user, order);
    if (!permission.allowed) {
      throw new Error(permission.reason || "Відмовлено в
доступі");
    }
    await prisma.order.delete({
      where: { id: data.id },
    });
  },
```

```
};
```

Рівень запитів надає сім функцій: getAllOrders, getAllOrdersStats, getMyOrganizationOrders, getMyOrganizationStats, getPublishedOrders, getOrderById та getOrderHistory. Усі функції застосовують кешування з теговою інвалідацією: після будь-якої мутуючої дії відповідні теги інвалідуються через revalidateTag, що змушує наступний запит звернутися до бази даних.

Клас OrderPermissionPolicy, наведений у лістингу 11, централізує всі правила доступу до замовлень у вигляді статичних методів, кожен з яких повертає { allowed: boolean; reason?: string }. Адміністратор має доступ до всіх замовлень; школа — лише до власних; постачальник — до опублікованих та прийнятих ним замовлень. Право на створення та редагування замовлень надається виключно школі.

Лістинг *11 — Клас OrderPermissionPolicy*

```
export class OrderPermissionPolicy {
  static canViewAllOrders(user: CurrentUser) {
    if (user.role !== "admin") {
      return {
        allowed: false,
        reason: "Відмовлено в доступі",
      };
    }
    return { allowed: true };
  }

  static canViewOrderById(user: CurrentUser, order: Order)
{
    const isSchoolMember =
      user.organizationType === "school" &&
      order.schoolId === user.organizationId;
    const isSupplierMember =
      user.organizationType === "supplier" &&
      order.supplierId === user.organizationId;
    const isPublishedOrderForSupplier =
      user.organizationType === "supplier" &&
      order.orderStatus === "published" &&
      !order.supplierId;
```

```
    const isAdmin = user.role === "admin";
    if (!isSchoolMember && !isSupplierMember &&
!isPublishedOrderForSupplier && !isAdmin) {
      return {
        allowed: false,
        reason: "Відмовлено в доступі",
      };
    }
    return { allowed: true };
  }

  static canViewPublishedOrders(user: CurrentUser) {
    if (
      user.role !== "employee" ||
      user.organizationType !== "supplier" ||
      !user.organizationId
    ) {
      return {
        allowed: false,
        reason: "Тільки постачальники можуть переглядати
опубліковані замовлення",
      };
    }
    return { allowed: true };
  }

  static canViewOrganizationData(user: CurrentUser) {
    if (
      user.role !== "employee" ||
      !user.organizationId ||
      !user.organizationType
    ) {
      return {
        allowed: false,
        reason: "Ви не маєте права переглядати ці дані",
      };
    }
    return { allowed: true };
  }

  static canCreateOrder(user: CurrentUser) {
    if (
      user.role !== "employee" ||
      user.organizationType !== "school" ||
      !user.organizationId
    ) {
      return {
        allowed: false,
        reason: "Відмовлено в доступі",
```

```
      };
    }
    return { allowed: true };
  }

  static canUpdateOrder(user: CurrentUser) {
    if (
      user.role !== "employee" ||
      user.organizationType !== "school" ||
      !user.organizationId
    ) {
      return {
        allowed: false,
        reason: "Відмовлено в доступі",
      };
    }
    return { allowed: true };
  }

  static canChangeStatus(
    user: CurrentUser,
    order: Order,
  ):
    | { allowed: true; isSchool: boolean; isSupplier:
boolean }
    | { allowed: false; reason: string } {
    if (user.role !== "employee") {
      return {
        allowed: false,
        reason: "Тільки працівники можуть змінювати статус
замовлення",
      };
    }
    const isSchool =
      user.organizationType === "school" &&
      user.organizationId === order.schoolId;
    const isSupplier =
      user.organizationType === "supplier" &&
      user.organizationId === order.supplierId;
    const isSupplierViewingPublished =
      user.organizationType === "supplier" &&
      order.orderStatus === "published" &&
      !order.supplierId;
    if (!isSchool && !isSupplier &&
!isSupplierViewingPublished) {
      return {
        allowed: false,
        reason: "Ви не маєте права змінювати статус цього
замовлення",
```

```
      };
    }

    return { allowed: true, isSchool, isSupplier:
isSupplier || isSupplierViewingPublished };
  }

  static canDeleteOrder(user: CurrentUser, order: Order) {
    if (
      user.role !== "employee" ||
      user.organizationType !== "school" ||
      user.organizationId !== order.schoolId
    ) {
      return {
        allowed: false,
        reason: "Відмовлено в доступі",
      };
    }
    return { allowed: true };
  }
}
```

Функція canTransitionOrderStatus, наведена в лістингу 12, реалізує скінченний автомат переходів статусу замовлення. Вона перевіряє допустимість переходу з поточного стану, наявність відповідних прав у виконавця та виконання бізнес-умов переходу. Функція canTransitionPaymentStatus аналогічно реалізує автомат переходів статусу оплати.

Лістинг *12 — Функція canTransitionOrderStatus*

```
export function canTransitionOrderStatus(
  order: Order & { orderItems: { id: string }[] },
  to: OrderStatus,
  isSchool: boolean,
  isSupplier: boolean,
  supplierId?: string,
): { allowed: boolean; reason?: string; newSupplierId?:
string } {
  let allowed = false;
  let reason = "";
  let newSupplierId: string | undefined;

  switch (order.orderStatus) {
    case "draft":
      if (to === "published" && isSchool) {
```

```
        if (order.orderItems.length === 0) {
          reason = "Замовлення порожнє — додайте хоча б
один товар";
          break;
        }
        if (!order.deliveryDate) {
          reason = "Вкажіть дату доставки";
          break;
        }
        if (order.deliveryDate <= new Date()) {
          reason = "Дата доставки повинна бути в
майбутньому";
          break;
        }
        allowed = true;
      } else reason = "From 'draft' only → published by
school";
      break;

    case "published":
      if (to === "accepted" && isSupplier) {
        allowed = true;
        newSupplierId = supplierId;
      } else if (to === "cancelled" && isSchool) allowed =
true;
      else
        reason =
          "From 'published': 'accepted' only by supplier,
'cancelled' only by school";
      break;

    case "accepted":
      if (
        to === "in_progress" &&
        isSupplier &&
        order.paymentStatus === "verified"
      )
        allowed = true;
      else if (
        to === "cancelled" &&
        isSupplier &&
        order.paymentStatus === "unpaid"
      )
        allowed = true;
      else
        reason =
          "From 'accepted' only → in_progress (verified) or
cancelled (unpaid) by supplier";
      break;
```

```
    case "in_progress":
      if (to === "completed" && isSchool) allowed = true;
      else reason = "From 'in_progress' only → completed by
school";
      break;

    case "completed":
    case "cancelled":
      reason = "Final status — cannot be changed";
      break;

    default:
      reason = "Unknown current status";
  }

  return { allowed, reason, newSupplierId };
}
```

3.4 Використання застосунку

Після успішного входу до системи користувач потрапляє на сторінку, визначену його роллю та типом організації. Застосунок підтримує три ролі: адміністратор (admin), співробітник школи (employee з організацією типу school) та співробітник компанії-постачальника (employee з організацією типу supplier). Навігаційне меню бічної панелі формується динамічно відповідно до ролі поточного користувача: кожна роль бачить лише ті розділи, до яких має доступ.

Форма входу (зображена на рисунку 10) приймає email та пароль. Форма реєстрації (зображена на рисунку 11) дозволяє новим користувачам створити обліковий запис із вибором організації зі списку наявних. Усі поля форм проходять клієнтську валідацію за допомогою Zod-схем ще до відправлення запиту на сервер.

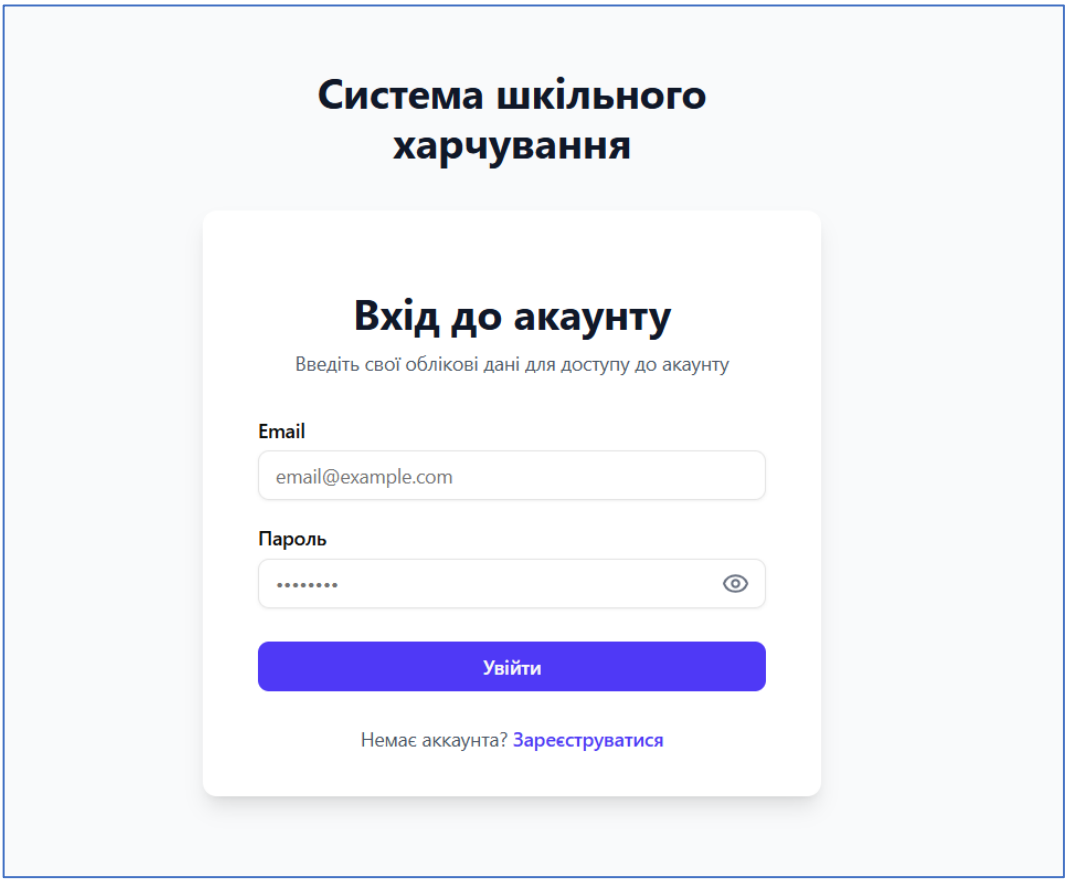


Рисунок *10 — Сторінка входу до системи*

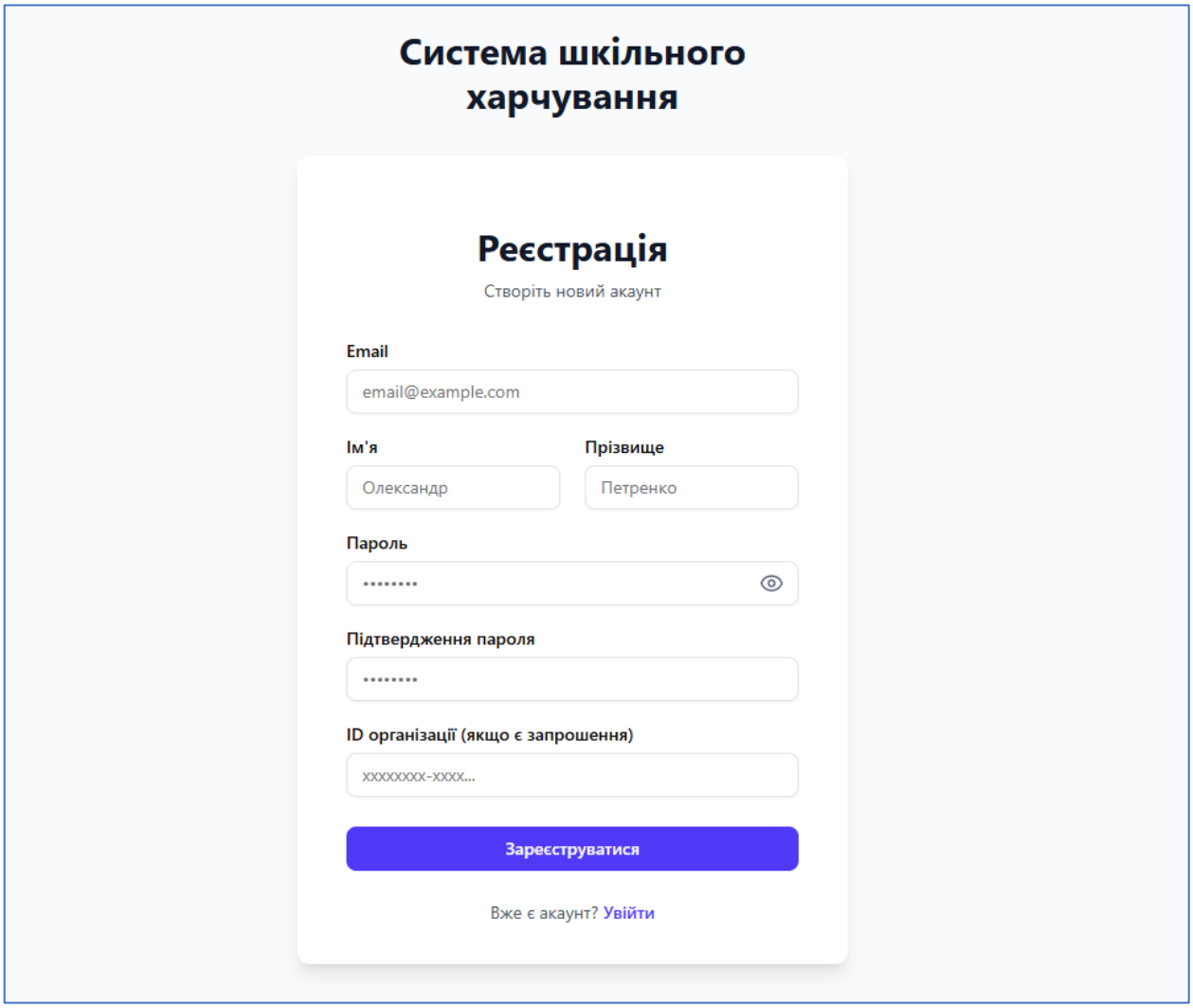


Рисунок *11 — Сторінка реєстрації нового користувача*

3.4.1 Функціонал адміністратора

Адміністратор має повний доступ до всіх розділів системи та виконує функції налаштування й моніторингу. Після входу він потрапляє на сторінку адміністративної панелі, що відображає зведену статистику по всьому застосунку.

Сторінка «Адміністративна панель» містить шість карток показників (зображених на рисунку 12): загальна кількість організацій (із розбивкою на школи та постачальників), кількість активних замовлень (зі статусами published та accepted), загальна сума незакритих і несплачених замовлень, кількість замовлень у процесі виконання, кількість завершених та кількість

скасованих замовлень. Нижче розташовані таблиця п'яти останніх замовлень із посиланнями на їх детальний перегляд, а також рейтинги п'яти найактивніших шкіл і постачальників за загальним обсягом замовлень у гривнях (зображено на рисунку 13).

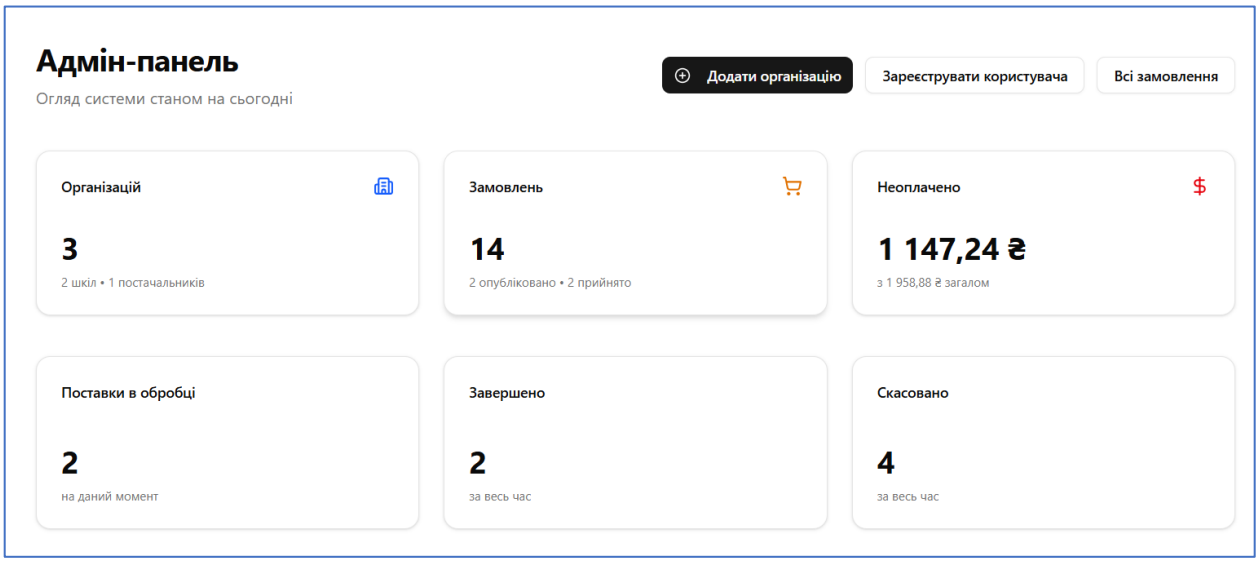


Рисунок *12 — Адміністративна панель: картки статистики*

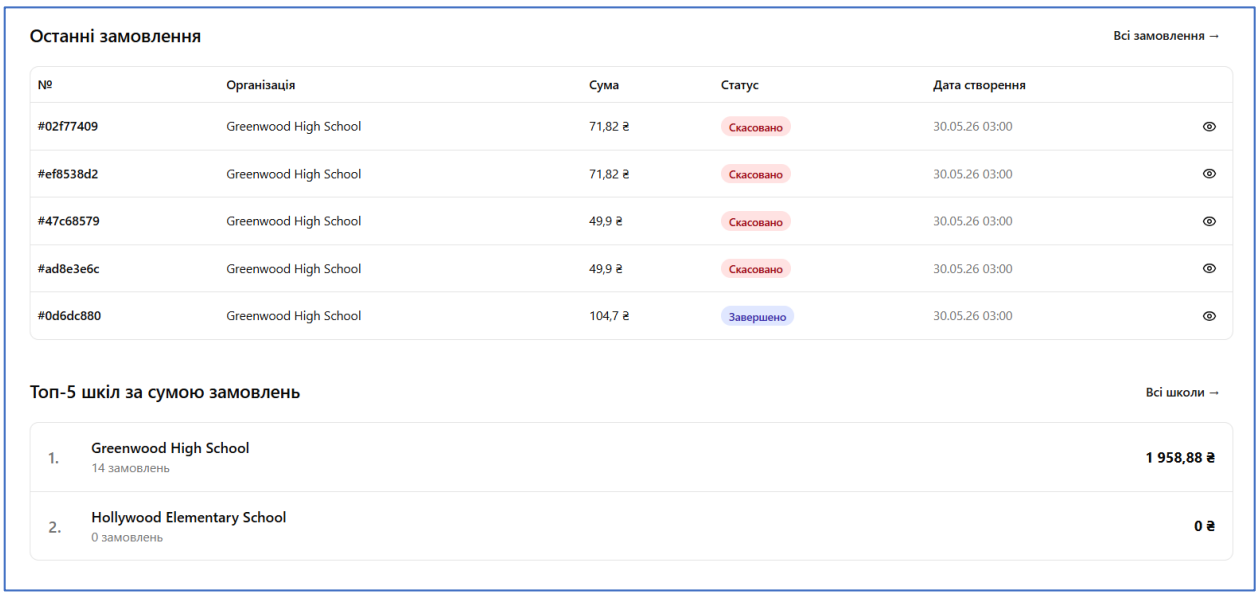


Рисунок *13 — Адміністративна панель: таблиця останніх замовлень та рейтинги організацій*

Сторінка «Управління організаціями» надає таблицю всіх організацій у системі з фільтрацією за типом (school/supplier) та пошуком за назвою (зображено на рисунку 14). Адміністратор може додати нову організацію, вказавши назву, тип, контактний email та телефон. Для редагування існуючої організації використовується окрема сторінка.

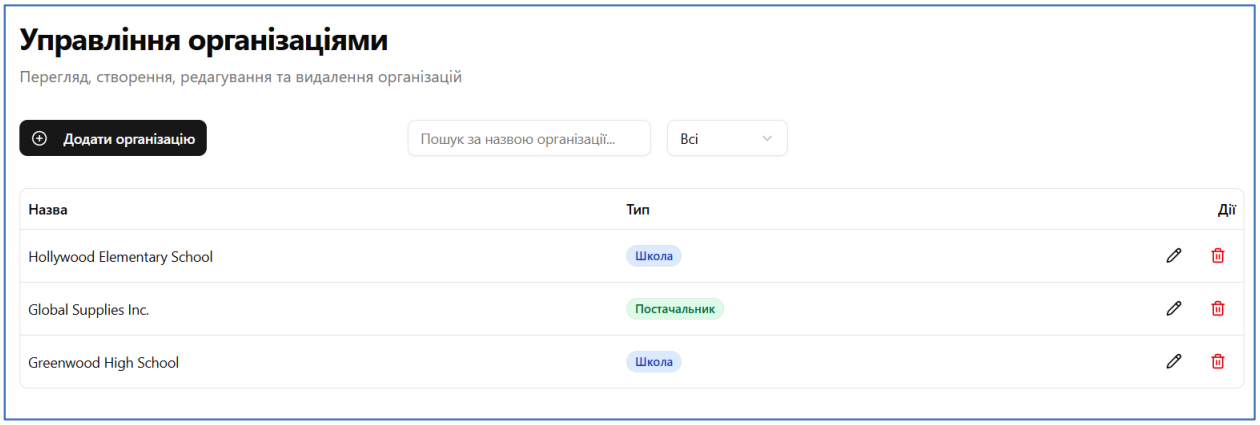


Рисунок *14 — Список організацій із фільтрами та таблицею*

Сторінка «Управління користувачами» відображає список усіх користувачів системи з прив'язкою до організації та роллю (зображено на рисунку 15. Адміністратор може зареєструвати нового користувача, вказавши його ім'я, прізвище, email, пароль, роль та організацію.

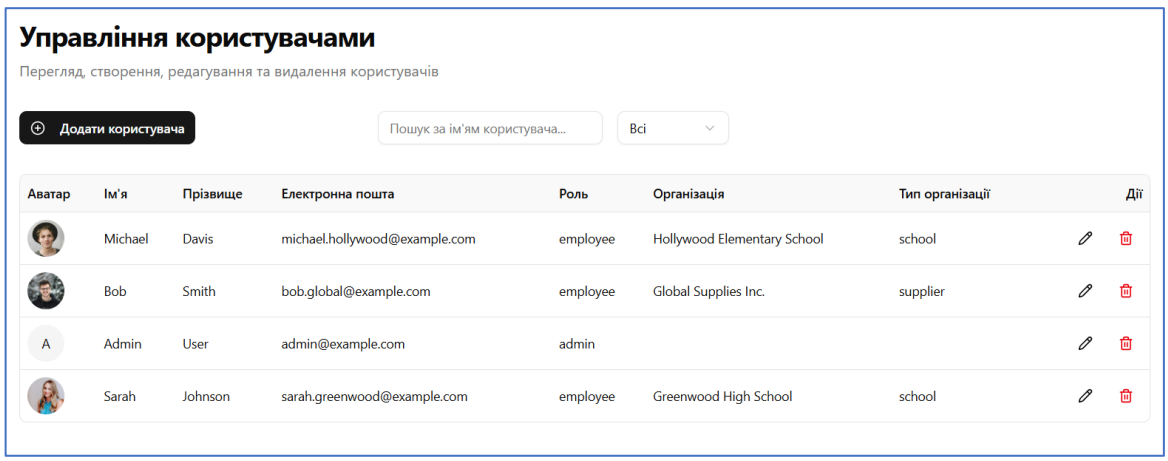


Рисунок *15 — Список користувачів системи*

Сторінка «Управління категоріями» дозволяє створювати та редагувати категорії страв (зображено на рисунку 16).

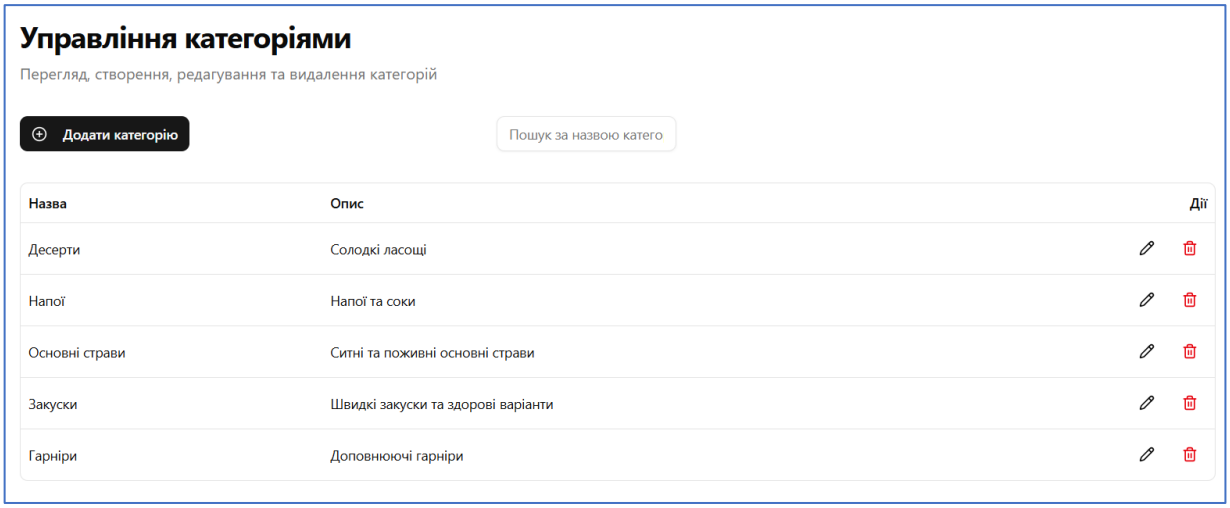


Рисунок *16 — Список категорій*

Сторінка «Управління пунктами меню» надає повне управління стравами: назва, опис, категорія, ціна, наявність та URL зображення (зображено на рисунку 17).

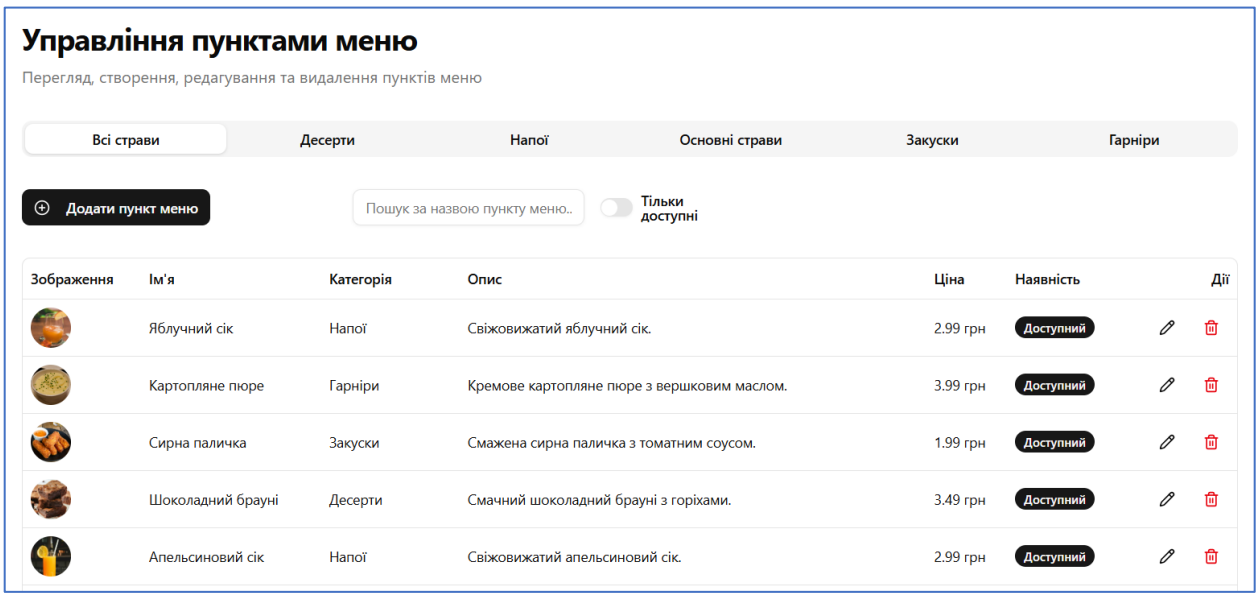


Рисунок *17 — Список позиції меню з фільтрацією за категорією*

Сторінка «Усі замовлення», зображена на рисунку 18, відображає усі замовлення системи з фільтрами за статусом замовлення, статусом оплати та діапазоном дат. Адміністратор має доступ до детального перегляду кожного замовлення з повною інформацією про позиції, статуси та їх хронологію.

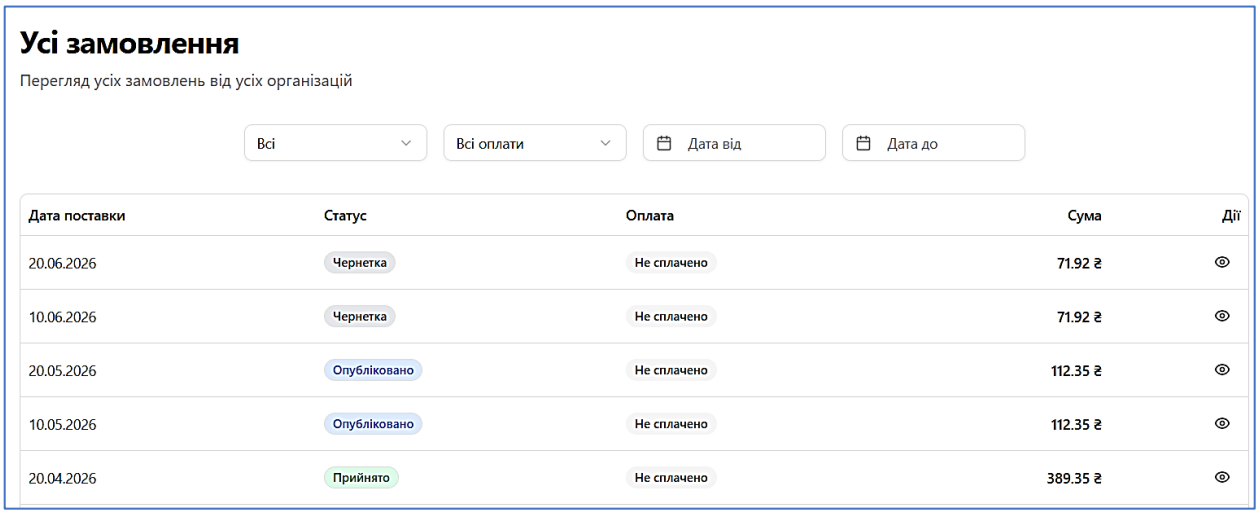


Рисунок *18 — Список усіх замовлень системи з фільтрами*

3.4.2 Функціонал працівника школи

Співробітник школи використовує систему для формування та відстеження замовлень харчування. Після входу він потрапляє на сторінку приладової панелі школи.

Сторінка «Дашборд» відображає персоналізовану статистику: кількість поточних замовлень за статусами, найближчі заплановані доставки та нещодавно завершені замовлення (зображено на рисунку 19). Звідси можна швидко перейти до створення нового замовлення або перегляду наявних.

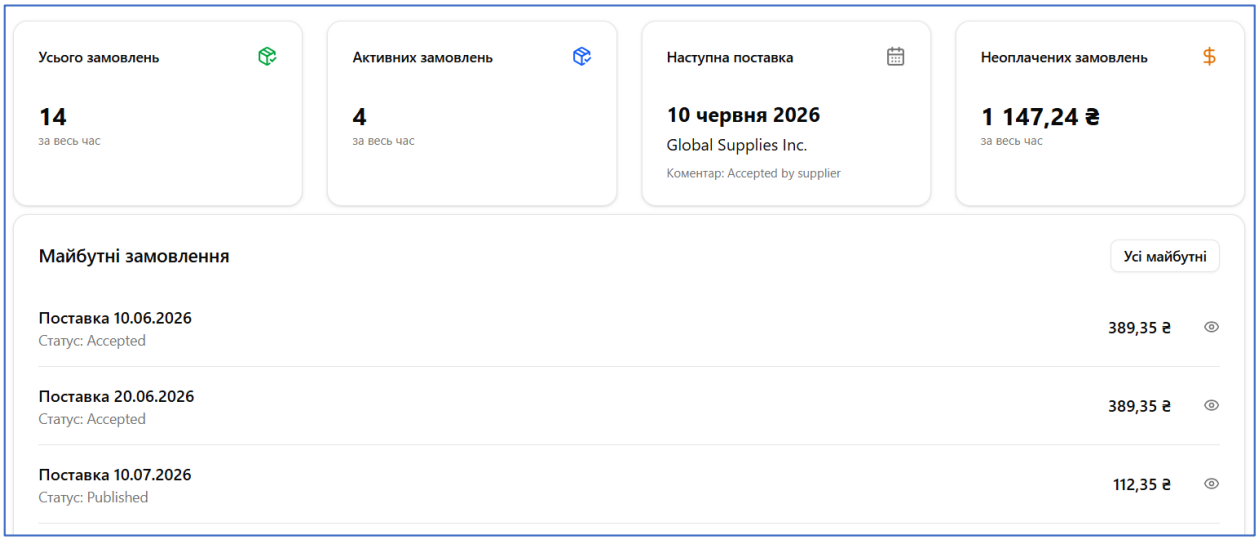


Рисунок *19 — Дашборд школи зі статистикою та найближчими доставками*

Сторінка «Меню» надає можливість ознайомитися з наявними стравами, згрупованими за категоріями (зображено на рисунку 20). Кожна позиція відображає назву, опис, категорію та ціну. Цей розділ є виключно довідковим.

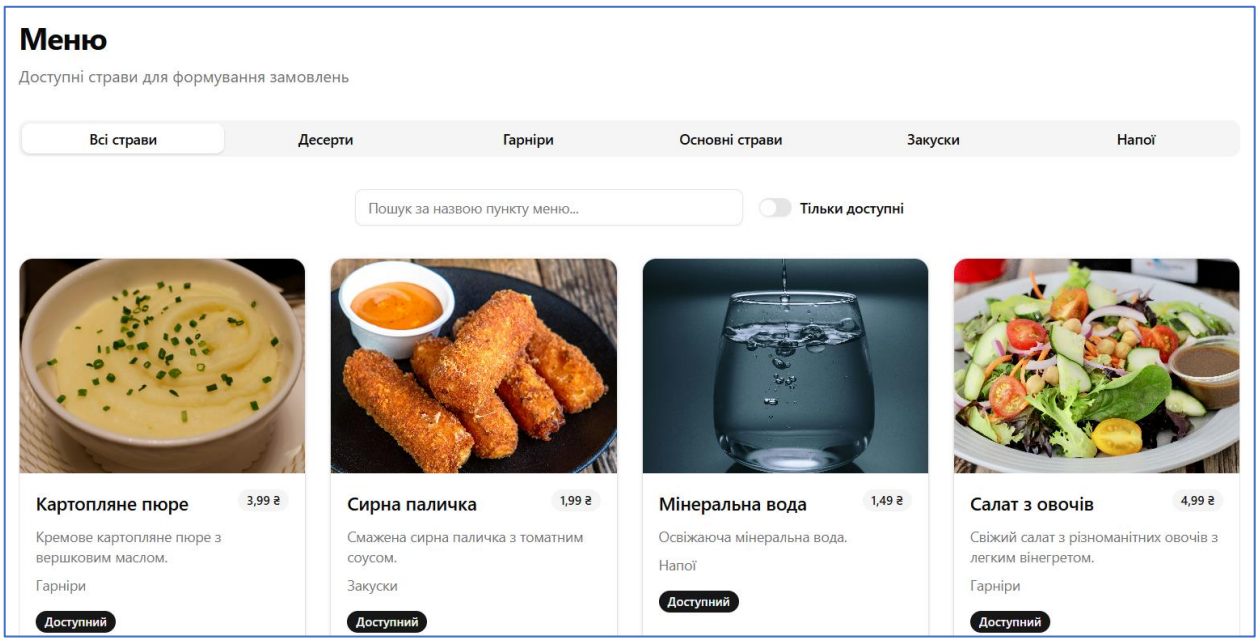


Рисунок *20 — Сторінка перегляду меню із розбивкою за категоріями*

Чернеткові замовлення доступні на **сторінці «Чорновики»**, зображеній на рисунку 21, а загальний перелік замовлень на **сторінці «Мої замовлення»**, зображеній на рисунку 22. Для створення нового замовлення достатньо натиснути відповідну кнопку — система автоматично формує чернетку та відкриває сторінку редагування.

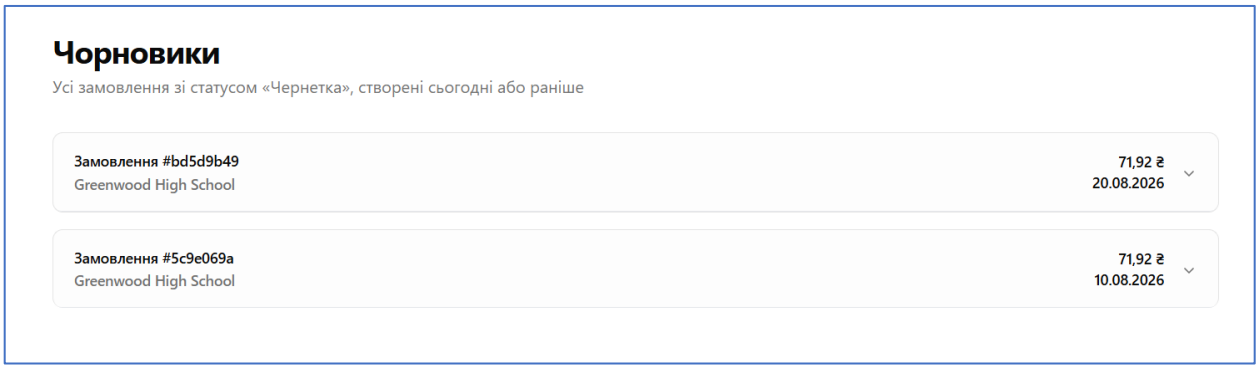


Рисунок *21 — Список чернеткових замовлень школи*

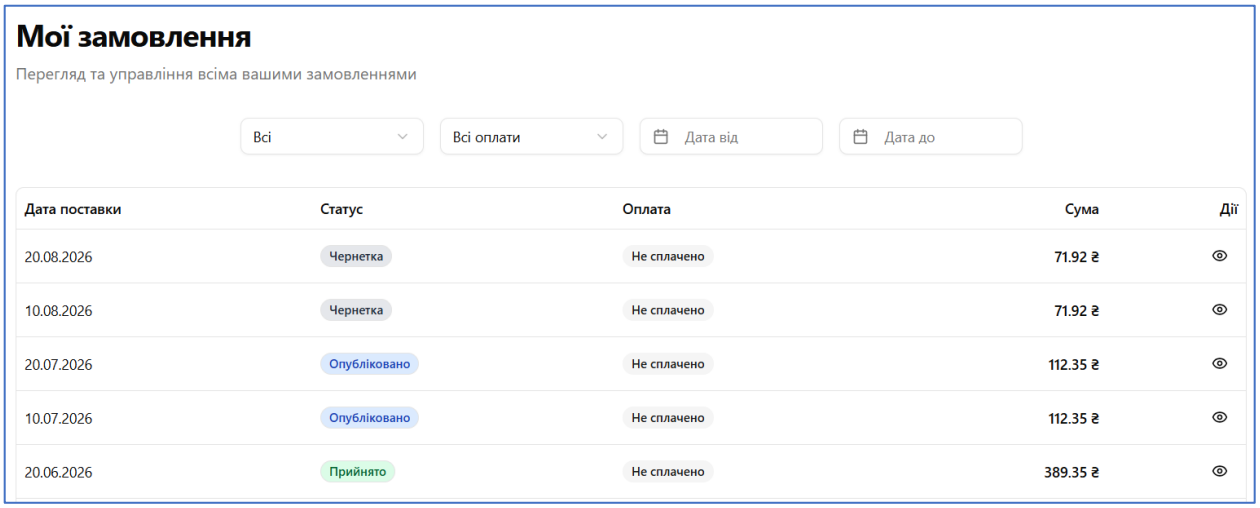


Рисунок *22 — Список усіх замовлень школи*

Сторінка «Редагування замовлення», зображена на рисунку 23, дозволяє: вказати дату доставки, додати страви з меню (форма вибору страви та кількості), видалити доданих позиції, додати коментар до замовлення. Після формування переліку страв співробітник публікує замовлення — система

перевіряє наявність хоча б однієї позиції та майбутню дату доставки. Опубліковане замовлення стає доступним постачальникам для ознайомлення та прийняття.

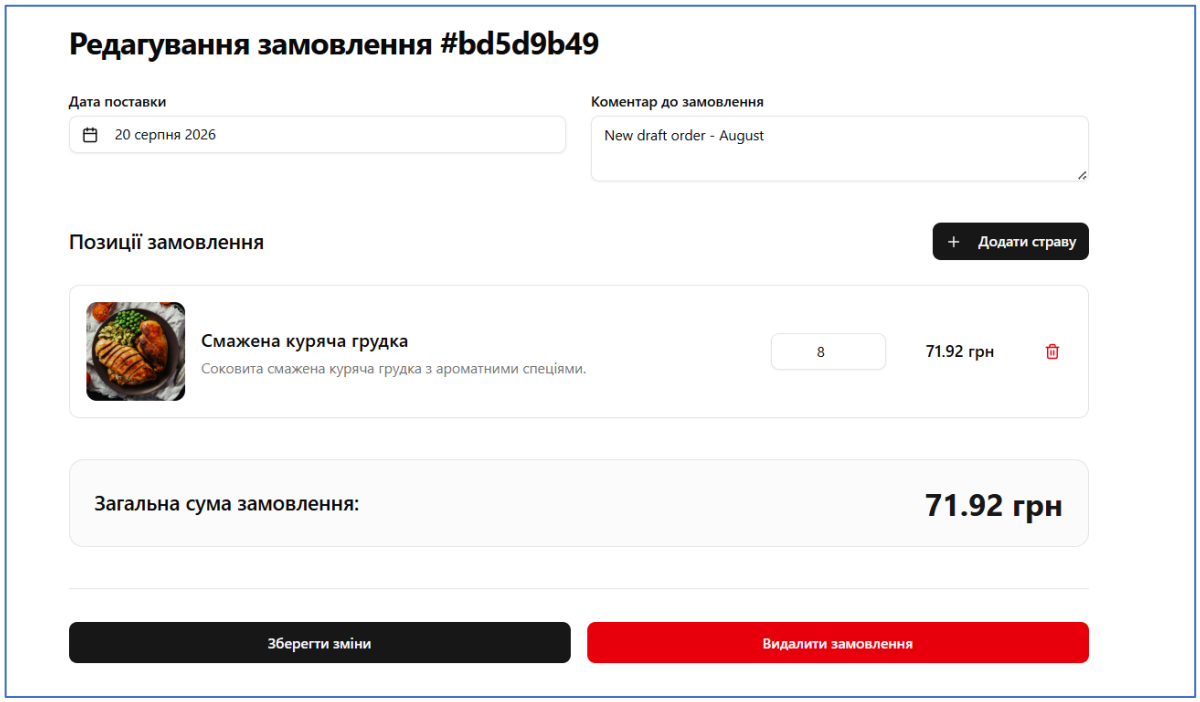


Рисунок *23 — Сторінка редагування замовлення: додавання страв та встановлення дати доставки*

Сторінка «Замовлення», зображена на рисунках 24 і 25, відображає повний перелік позицій, поточний статус замовлення та оплати, дані постачальника (після прийняття) та повну хронологію змін статусів.

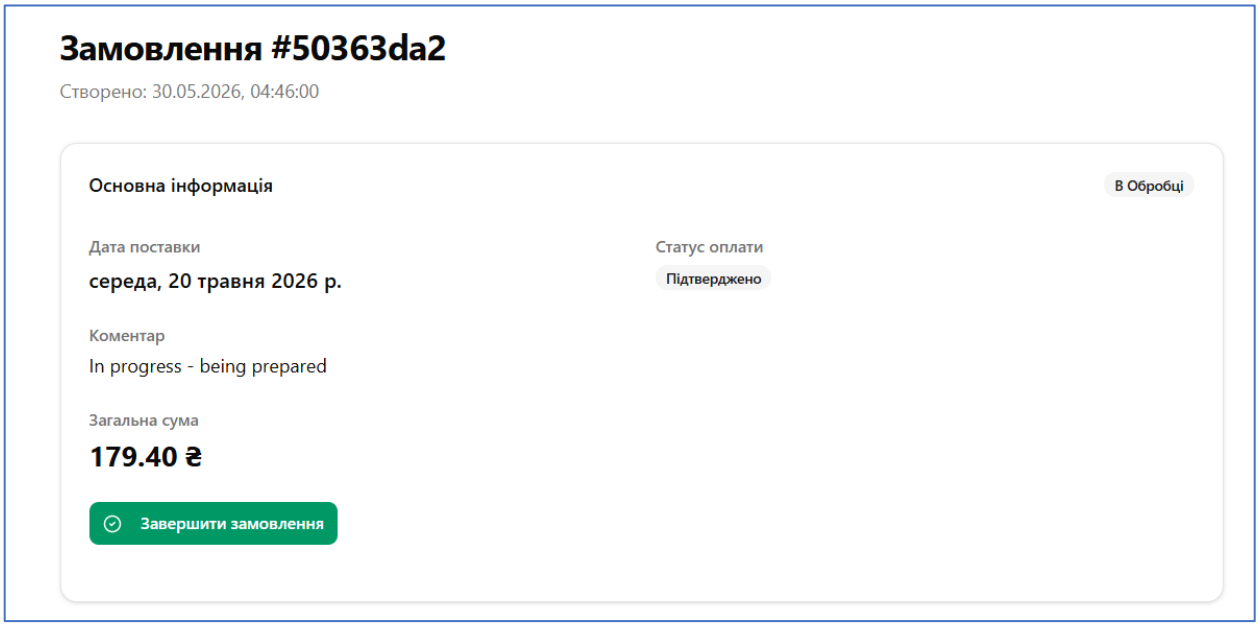


Рисунок *24 — Детальна сторінка замовлення: опис замовлення*

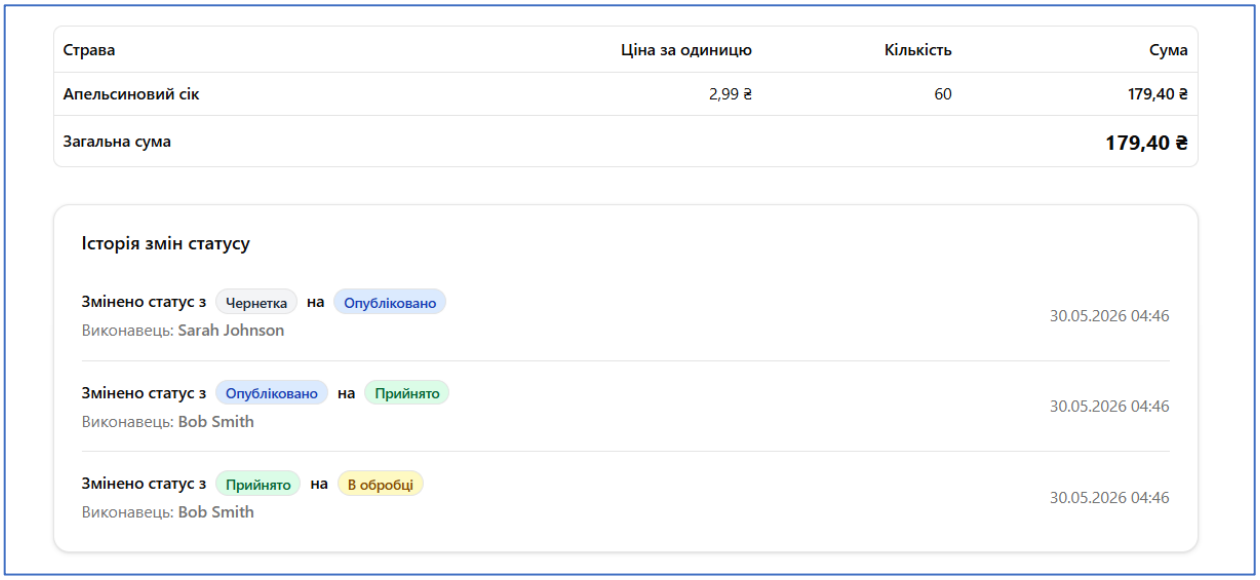


Рисунок *25 — Детальна сторінка замовлення: перелік позицій та хронологія статусів*

3.4.3 Функціонал працівника компанії-постачальника

Співробітник постачальника відповідає за прийняття замовлень від шкіл та управління процесом їх виконання. Після входу він потрапляє на панель постачальника.

Сторінка «Дашборд», зображена на рисунку 26, відображає статистику по замовленнях, що перебувають у роботі: кількість прийнятих, поточних та виконаних замовлень, загальний обсяг у гривнях.

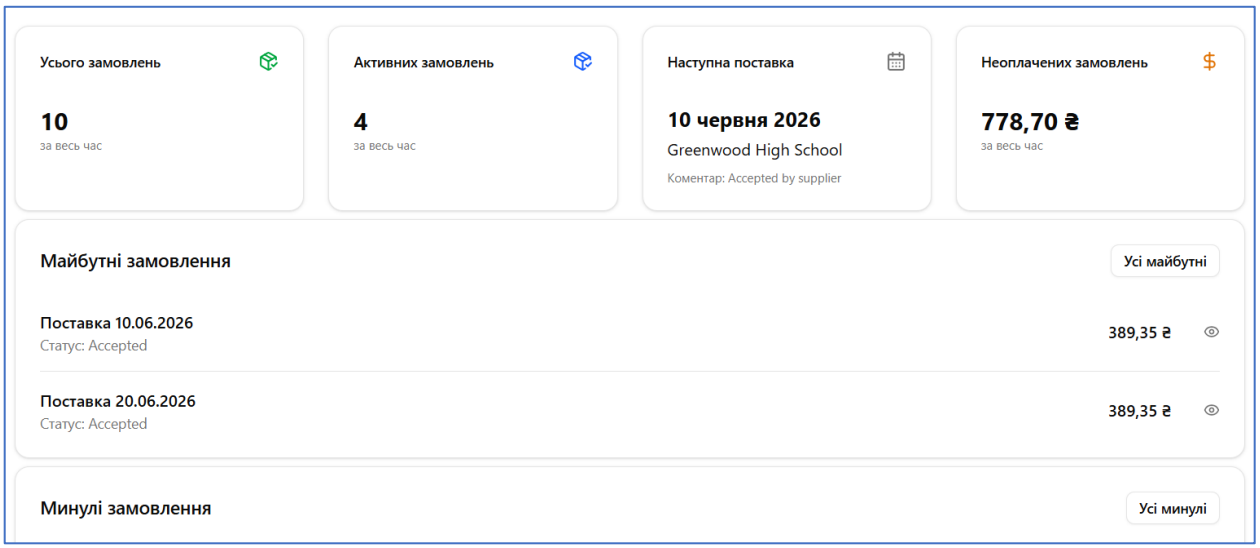


Рисунок *26 — Дашборд постачальника зі статистикою активних замовлень*

Сторінка «Опубліковані замовлення», зображена на рисунку 27, відображає усі опубліковані школами замовлення, що ще не прийняті жодним постачальником. Список включає дату доставки, назву школи, кількість позицій та загальну суму. Постачальник може переглянути деталі замовлення та прийняти його.

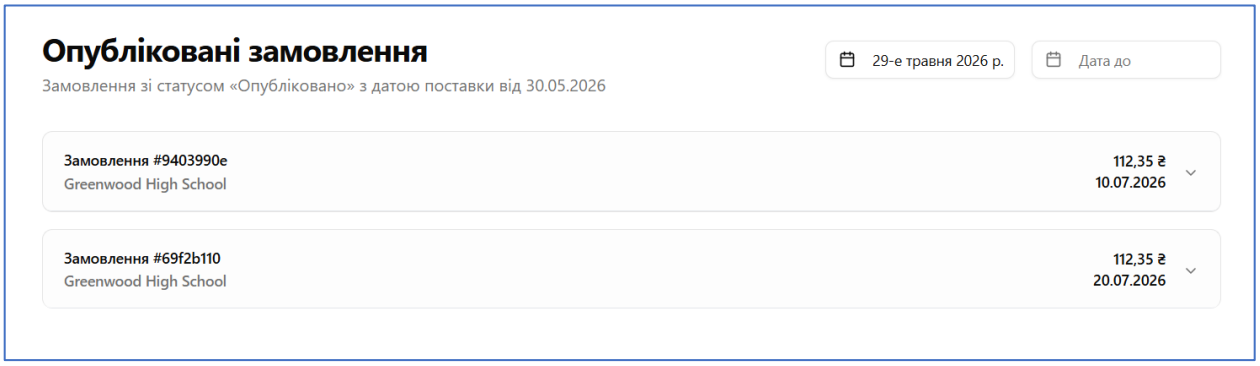


Рисунок *27 — Список опублікованих замовлень, доступних для прийняття*

На **сторінці «Замолення»**, зображеній на рисунку 28, постачальник бачить повний перелік замовлених страв та їх кількість. Натискання кнопки «Прийняти» переводить замовлення зі статусу published у accepted та прив'язує постачальника до замовлення. Після цього замовлення зникає зі **сторінки «Опубліковані замовлення»** та з'являється на **сторінці «Мої замовлення»**.

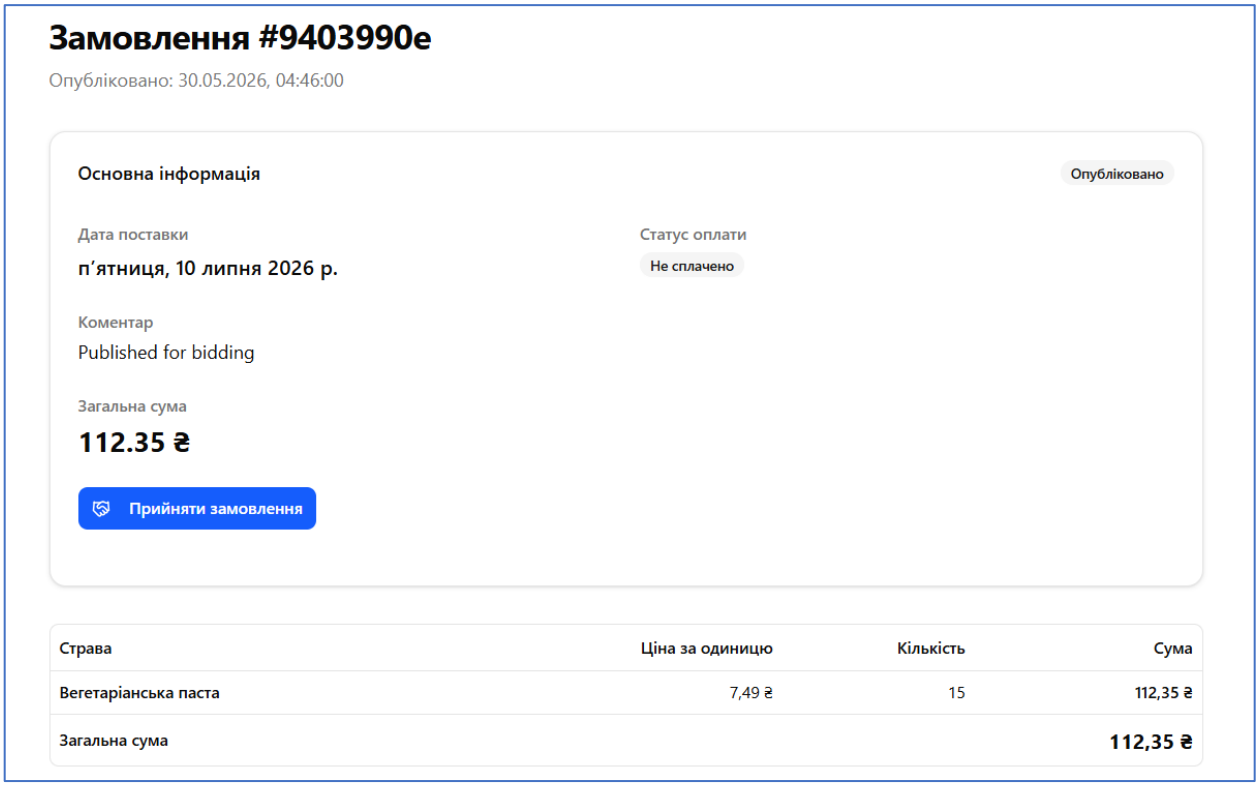


Рисунок *28 — Деталі опублікованого замовлення з кнопкою прийняття*

Сторінка «Мої замовлення», зображена на рисунку 29, відображає замовлення постачальника в усіх активних статусах: accepted, in_progress, completed, cancelled. Для кожного замовлення доступні дозволені дії залежно від його поточного статусу.

Після прийняття замовлення постачальник отримує можливість переводити його по ланцюжку статусів: коли школа позначить замовлення як оплачене, постачальник може перевести статус у in_progress (розпочато виготовлення/підготовку). Завершивши виконання, він підтверджує оплату. Школа в свою чергу завершує замовлення.

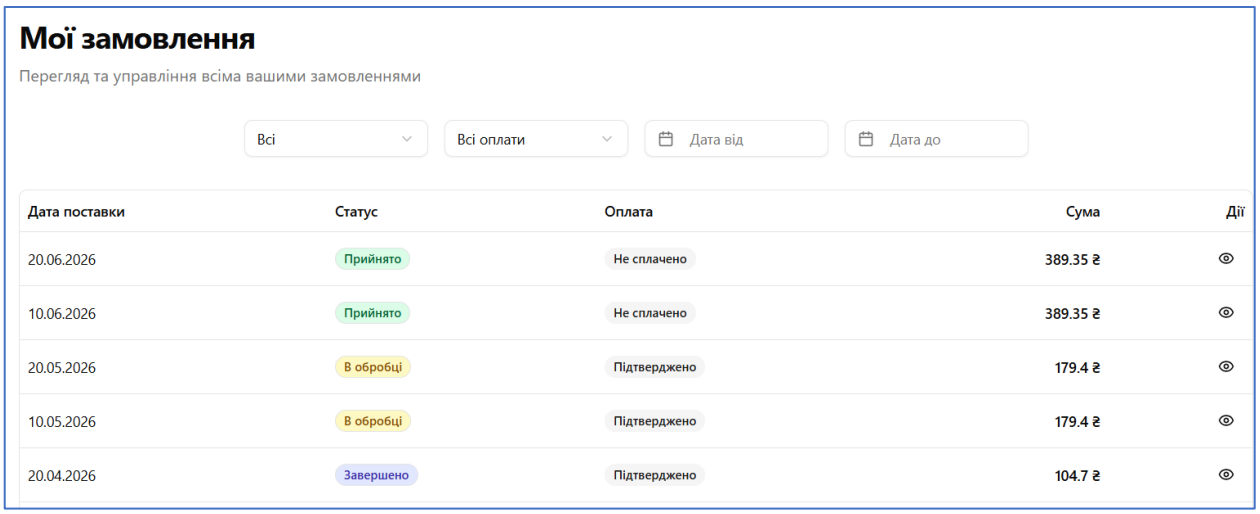


Рисунок *29 — Список власних замовлень постачальника з фільтрами та статусами*

3.5 Висновки з розділу 3

У цьому розділі здійснено детальний аналіз структури та реалізації веб-застосунку для замовлення шкільного харчування.

База даних застосунку спроектована на основі PostgreSQL та описана за допомогою ORM Prisma. Схема містить вісім моделей, пов'язаних відносинами типу «один до багатьох», чотири перелічувальні типи для

кодування станів сутностей, а також таблиці аудиторського журналювання для відстеження змін статусів замовлень і оплат. Індексування ключових полів забезпечує прийнятну продуктивність під час фільтрації та пошуку.

Клієнтську частину реалізовано на базі React 19 та Next.js 16 з App Router, що дозволяє поєднувати серверний рендеринг з інтерактивними клієнтськими компонентами. Стилізацію виконано засобами Tailwind CSS у поєднанні з доступними примітивами Radix UI. Форми контролюються бібліотекою React Hook Form із синхронною Zod-валідацією на стороні клієнта.

Серверну частину побудовано на основі Next.js Server Actions за багаторівневою архітектурою, що включає рівень серверних дій, сервісний рівень, рівень запитів із кешуванням та доменний рівень. Доменну логіку виражено через скінченні автомати переходів для статусів замовлення та оплати, а розмежування доступу централізовано в класі OrderPermissionPolicy. Аутентифікацію реалізовано через NextAuth.js із JWT-стратегією; паролі зберігаються виключно у вигляді bcrypt-хешів.

Застосунок підтримує три ролі користувачів із чітко розмежованим функціоналом: адміністратор налаштовує систему та веде моніторинг; школа формує й публікує замовлення та відстежує їх виконання; постачальник приймає замовлення і керує процесом їх виконання. Таке розмежування забезпечує відповідність системи реальним бізнес-процесам у сфері організації шкільного харчування.

ВИСНОВКИ

1. Під час аналізу предметної області було досліджено специфіку B2B-взаємодії між школами та постачальниками харчування. На основі аналізу недоліків ручного обліку обґрунтовано доцільність автоматизації процесів формування та відстеження замовлень, що є особливо актуальним для навчальних закладів Запорізького регіону, які працюють в умовах воєнного стану.
2. На основі аналізу проблем неавтоматизованих підходів (людський фактор, розсинхронізація даних, хаотичність месенджерів та електронних таблиць) обґрунтовано необхідність створення профільного веб-застосунку. Встановлено високу актуальність такої системи в умовах реформи шкільного харчування в Україні та специфіки функціонування установ Запорізького регіону в безпекових просторах і підземних школах під час воєнного стану. Було сформовано функціональні, нефункциональні та технічні вимоги для системи управління шкільним харчуванням. Визначено рольову модель доступу для трьох категорій користувачів (адміністратор, представник школи, постачальник). Процеси зміни станів замовлень та взаєморозрахунків формалізовано у вигляді логічних машин станів (скінченних автоматів), що призначені для покращення взаємодії, підвищення швидкості транзакцій, забезпечення прозорості та розмежування відповідальності між контрагентами. Спроєктовано архітектуру вебзастосунку, яка визначає основні компоненти системи та їхню взаємодію між собою. Розроблено архітектуру для різних частин застосунку, зокрема клієнтську частину, серверну частину та базу даних. Обрано повностековий (full-stack) підхід із модульною feature-based структурою.
3. Спроєктовано реляційну структуру бази даних у СУБД PostgreSQL, яка містить 8 взаємопов'язаних таблиць та журнал аудиту змін, що забезпечує цілісність, надійність, типобезпеку та атомарність фінансових транзакцій.

Було реалізовано серверну частину застосунку на базі фреймворка Next.js. Безпечне виконання бізнес-логіки, обробка мутацій та взаємодія з базою даних забезпечуються за допомогою механізму Next.js Server Actions та React Server Components без необхідності проєктування окремого REST API. Доступ до даних та операцій жорстко розмежовано на рівні сервісів через централізовану політику прав (OrderPermissionPolicy). Безпеку облікових записів реалізовано через NextAuth.js із криптографічним хешуванням паролів за алгоритмом bcrypt. Серверна частина забезпечує високу продуктивність завдяки механізмам гнучкого кешування з теговою інвалідацією.

4. Реалізовано клієнтську частину застосунку на базі React 19 з використанням App Router, яка забезпечує інтуїтивно зрозумілий та адаптивний інтерфейс користувача. Завдяки стеку Tailwind CSS, Radix UI та компонентів shadcn/ui, інтерфейс є повністю доступним (відповідність стандартам ARIA), адаптованим під мобільні пристрої за принципом mobile-first та надає швидкий відгук користувачеві завдяки валідації форм у реальному часі.

5. Клієнтська частина забезпечує зручний досвід використання застосунку для кожної ролі, надаючи інформативні аналітичні панелі, динамічні інтерактивні таблиці з пагінацією та фільтрацією, а також зручний покроковий конструктор замовлень харчування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Нове шкільне харчування: збірник рецептур страв для закладів освіти / Є. Клопотенко. Київ, 2021.
2. Павленко.А. Розробка Веб-застосунку для замовлення їжі з використанням фреймворка Next.js. Збірник наукових праць студентів, здобувачів PhD, докторантів і молодих вчених, університетська конференція «МОЛОДА НАУКА-2026» Том V «Секції Інженерного навчально-наукового інституту ім. Ю.М. Потебні Запорізького національного університету» : м. Запоріжжя, 20–24 квітня 2026 р. Запоріжжя, 2026. С. 382–384
3. Next.js Documentation. Production React Framework / Vercel Inc. URL: https://nextjs.org/docs.
4. React v19 Documentation. Web Components & Framework Hooks / Meta Platforms Inc. URL: https://react.dev.
5. Prisma ORM Documentation. Object-Relational Mapping for TypeScript / Prisma Data Platform Inc. URL: https://www.prisma.io/docs.
6. PostgreSQL 16 Documentation / The PostgreSQL Global Development Group. URL: https://www.postgresql.org/docs/16/index.html.
7. NextAuth.js. Authentication for Next.js Applications / Vercel Inc. URL: https://next-auth.js.org.
8. Tailwind CSS. Utility-First CSS Framework / Tailwind Labs Inc. URL: https://tailwindcss.com/docs.
9. Radix UI. Unstyled Accessible Primitive React Components / WorkOS Inc. URL: https://www.radix-ui.com.
10. Zod. TypeScript-first schema validation with static type inference / C. Colker. URL: https://zod.dev.

Декларація

академічної доброчесності
здобувача ступеня вищої освіти ЗНУ

Я, Павленко Анастасія Олександрівна, студентка 4 курсу, форми навчання денної, Інженерного навчально-наукового інституту, спеціальності 121 Інженерія програмного забезпечення, адреса електронної пошти se1212-16@stu.zsea.edu.ua, – підтверджую, що написана мною кваліфікаційна робота на тему «Розробка Веб-застосунку для замовлення їжі з використанням фреймворка Next.js» відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений.

- Заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії

- Згоден на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-системи, а також на архівування моєї роботи в базі даних цієї системи.

Дата 07.06.2026 _____ Павленко А.О.
(підпис) (студент)

Дата 07.06.2026 _____ Коломоєць Г.П.
(підпис) (керівник)