НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
В.о. завідувача кафедри
_____ Михайло НОВОТАРСЬКИЙ
(підпис)

“__” _____2025 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

на тему: Веб-застосунок для управління діяльністю мережі магазинів

Виконав : студент __4__ курсу, групи _ІМ-13_
(шифр групи)

_____Кравчук Ілля Володимирович_____
(прізвище, ім’я, по батькові) (підпис)

Керівник __асистент, доктор філософії Шульга М. В._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) ас. Пономаренко А. М_____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент __асистент кафедри БМІ Матвійчук О.В._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2025 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Михайло НОВОТАРСЬКИЙ
(підпис)

“__” _____2025 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Кравчука Іллі Володимировича

1. Тема проєкту *Веб-застосунок для управління діяльністю мережі магазинів*
керівник проєкту Шульга Максим Володимирович, асистент, доктор філософії,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 23 травня 2025 року № 1705-с

2. Термін здачі студентом закінченого проєкту 05.06.2025

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)

Розділ 1. Веб-застосунки для управління діяльністю мережі магазинів.

Розділ 2. Вибір технологій розробки.

Розділ 3. Проєктування та реалізація веб-застосунку.

Розділ 4. Огляд та тестування роботи веб-застосунку.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структурна схема системи, діаграма бази даних (функціональна схема), алгоритм дій користувача в системі (принципова схема).

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Пономаренко А.М.		

7. Дата видачі завдання «27» грудня 2024 р

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	15.12.2024 – 27.12.2024	
2.	*Вивчення та аналіз завдання*	28.12.2024 – 14.01.2025	
3.	*Розробка архітектури та загальної структури системи*	15.01.2025 – 31.01.2025	
4.	*Розробка структур окремих підсистем*	01.02.2025 – 07.03.2025	
5.	*Програмна реалізація системи*	08.03.2025 – 19.04.2025	
6.	*Оформлення пояснювальної записки*	20.04.2025 – 02.06.2025	
7.	*Захист програмного продукту*		
8.	*Передзахист*		
9.	*Захист*	19.06.2025	

Студент-дипломник _____ Ілля КРАВЧУК
(підпис)

Керівник проєкту _____ Максим ШУЛЬГА
(підпис)

АНОТАЦІЯ

У дипломному проєкті розроблено веб-застосунок для централізованого управління мережею магазинів. Проведено аналіз сучасного стану роздрібної торгівлі, виявлено недоліки ручного обліку та сформульовано функціональні й нефункціональні вимоги до системи. На основі огляду існуючих рішень обґрунтовано вибір технологічного стека: Node.js + Express для серверної логіки, PostgreSQL + Prisma для роботи з даними, React, Tailwind CSS, Material UI – для клієнтського інтерфейсу, JWT – для авторизації; контейнеризацію та розгортання реалізовано через Docker / Docker Compose. Запропонована архітектура забезпечує масштабованість, високу продуктивність і безпеку, а впроваджений прототип демонструє можливості оперативного контролю запасів, продажів і аналітики всієї торговельної мережі.

Ключові слова: веб-застосунок, роздрібна торгівля, автоматизація, Node.js, React, PostgreSQL, Docker.

ANNOTATION

The bachelor's project presents a web application that provides centralized management of a retail store network. It analyses the current retail landscape, identifies drawbacks of manual processing, and defines functional and non-functional requirements. After surveying existing solutions, the following technology stack is justified: Node.js + Express for the back-end, PostgreSQL + Prisma for data access, React, Tailwind CSS, Material UI for the front-end, and JWT for authorization; deployment is containerized with Docker / Docker Compose. The proposed architecture delivers scalability, high performance, and security, and the implemented prototype illustrates real-time control of inventory, sales, and analytics across all stores.

Keywords: web application, retail automation, Node.js, React, PostgreSQL, Docker.

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Веб-застосунок для	3		
			управління діяльністю			
			мережі			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Веб-застосунок для	69		
			управління діяльністю			
			мережі			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Веб-застосунок для	1		
			управління діяльністю			
			мережі			
			Структурна схема системи			
	А4	***ІАЛЦ.4672008.005 Д2***	Веб-застосунок для	1		
			управління діяльністю			
			мережі			
			Діаграма бази даних			
			(функціональна схема)			
	А4	***ІАЛЦ.467200.006 Д3***	Веб-застосунок для	1		
			управління діяльністю			
			мережі			
			Алгоритм дій користувача в			
			системі			
			(принципова схема)			
	А4	***ІАЛЦ.467200.007 Д4***	Веб-застосунок для	36		
			управління діяльністю			
			мережі			
			Текст програмного коду			

					ІАЛЦ.467200.001 ОА		
Зм	*Лист*	*№ докум.*	*Підп*	*Дата*			
Розроб		Кравчук І. В.			Веб-застосунок для управління діяльністю мережі магазинів Опис альбому	Літ.	Аркуш 1 / Аркушів 1
Перев		Шульга М. В.					
						КПІ ім. Ігоря Сікорського ФІОТ ІМ-13	

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Веб-застосунок для управління діяльністю мережі*»

Київ – 2025

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Технічне завдання поширюється на розробку «Веб-застосунку для управління мережею магазинів» у межах дипломної роботи бакалавра.

Область застосування – роздрібна торгівля: автоматизація запасів, продажів, лояльності та аналітики у центральному офісі й торгових точках малого та середнього бізнесу.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою є індивідуальне завдання на виконання кваліфікаційної роботи ОС «бакалавр інженерії програмного забезпечення», затверджене кафедрою факультету Інформатики та обчислювальної техніки НТУУ «КПІ ім. Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою роботи є створення веб-застосунку, який об’єднає в одній системі централізоване ведення товарних залишків, контроль продажів і фінансів у режимі реального часу, аналітичні панелі для керівництва та безпечний віддалений доступ персоналу. Таке рішення покликане підвищити оперативність управлінських рішень, мінімізувати людські помилки та скоротити витрати на підтримку IT-інфраструктури торговельної мережі.

4 ДЖЕРЕЛА РОЗРОБКИ

Основою для проєкту слугують відкриті технічні та методичні матеріали з веб-технологій, сучасні стандарти інформаційної безпеки, а також профільна науково-технічна література й галузеві публікації, що описують практики автоматизації роздрібної торгівлі та проєктування багаторівневих інформаційних систем.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Мати простий і зрозумілий веб-інтерфейс.
- Відображати та оновлювати дані про товари, замовлення й клієнтів.
- Формувати короткі звіти про продажі та залишки.
- Забезпечувати вхід до системи за логіном-паролем.
- Надавати стислий опис системи та коротку інструкцію користувача.

5.2. Вимоги до програмного забезпечення

- ОС Linux / Windows / macOS.
- Visual Studio Code.
- Docker Desktop або Docker Engine версії 24 чи вище.
- Сучасний браузер.

5.3. Вимоги до апаратної частини

- Мінімум 2 CPU ядра x86-64 чи ARM.
- RAM від 2 ГБ на клієнті та 4 ГБ на сервері.
- Не менше 20 ГБ вільного дискового простору.
- Стабільне підключення до мережі.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	15.12.2024 – 27.12.2024
Вивчення та аналіз завдання	28.12.2024 – 14.01.2025
Розробка архітектури та загальної структури системи	15.01.2025 – 31.01.2025
Розробка структур окремих частин системи	01.02.2025 – 07.03.2025
Програмна реалізація системи	08.03.2025 – 01.04.2025
Виправлення помилок	02.04.2025 – 25.04.2025
Оформлення пояснювальної записки	26.04.2025 – 02.06.2025

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Веб-застосунок для управління діяльністю мережі*»

Київ – 2025

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

API	(Application Programming Interface) Інтерфейс прикладного програмування
CRM	(Customer Relationship Management) Управління взаємовідносинами з клієнтами
ERP	(Enterprise Resource Planning) Система планування ресурсів підприємства
POS	(Point of Sale) Точка продажу
WMS	(Warehouse Management System) Система управління складом
IT	Інформаційні технології
ПК	Персональний комп’ютер
UI	(User Interface) Користувацький інтерфейс
JWT	(JSON Web Token) Веб-токен у форматі JSON
ORM	(Object-Relational Mapping) Об’єктно-реляційне відображення
REST	(Representational State Transfer) Передавання представлення стану
JSON	(JavaScript Object Notation) Текстовий формат обміну даними
SQL	(Structured Query Language) Структурована мова запитів
Npm	(Node Package Manager) Менеджер пакетів для JavaScript
HTML	(HyperText Markup Language) Мова гіпертекстової розмітки
CSS	(Cascading Style Sheets) Каскадні таблиці стилів
URL	(Uniform Resource Locator) Уніфікований локатор ресурсу
SPA	(Single Page Application) Односторінковий застосунок
MUI	(Material User Interface) Бібліотека інтерфейсних компонентів Material Design для React
UI/UX	(User Interface / User Experience) Користувацький інтерфейс / користувацький досвід

CRUD	(Create, Read, Update, Delete) Створення, читання, оновлення, видалення
HTTP	(HyperText Transfer Protocol) Протокол передавання гіпертексту
CORS	(Cross-Origin Resource Sharing) Механізм спільного доступу до ресурсів з інших джерел

ВСТУП

У сучасних ринкових умовах ефективне управління мережею магазинів є ключовою складовою успішної підприємницької діяльності. З ростом конкуренції та розширенням торговельних мереж актуальним стає питання оперативного та якісного управління усіма аспектами функціонування бізнесу. Особливої актуальності набуває завдання автоматизації управлінських процесів, які традиційно потребують значних часових та фінансових витрат.

Використання сучасних інформаційних технологій дає змогу суттєво підвищити ефективність роботи торговельних мереж. Веб-застосунки в цьому контексті мають низку переваг перед іншими типами інформаційних систем: вони доступні з будь-якої точки світу через Інтернет, не вимагають складного програмного забезпечення на пристроях користувачів, забезпечують оперативну взаємодію різних підрозділів підприємства та полегшують процес прийняття управлінських рішень. Веб-застосунки для управління мережею магазинів дають можливість централізовано керувати запасами, контролювати продажі, відстежувати фінансові показники та аналізувати ефективність роботи магазинів у реальному часі.

Незважаючи на очевидні переваги, далеко не всі торговельні підприємства використовують сучасні веб-рішення, що призводить до зниження їх конкурентоспроможності на ринку. Недостатня автоматизація та інтеграція процесів управління викликають труднощі у оперативному реагуванні на ринкові зміни та потреби клієнтів.

Таким чином, актуальність розробки веб-застосунку для управління мережею магазинів зумовлена необхідністю підвищення ефективності управлінських процесів, покращення контролю за діяльністю торгових точок та забезпечення високого рівня обслуговування клієнтів.

Метою даної дипломної роботи є створення веб-застосунку для ефективного управління діяльністю мережі магазинів, що дозволить оптимізувати процеси прийняття рішень, покращити контроль та управління ресурсами підприємства, а також підвищити загальну ефективність його функціонування.

РОЗДІЛ 1
ВЕБ-ЗАСТОСУНКИ ДЛЯ УПРАВЛІННЯ ДІЯЛЬНІСТЮ МЕРЕЖІ МАГАЗИНІВ

1.1 Загальний опис предметної області

1.1.1 Сучасний стан роздрібної торгівлі

Роздрібна торгівля в сучасному світі є не лише сферою обміну товарів, але і комплексною системою взаємодії між постачальниками, продавцями та кінцевими споживачами. Успішність діяльності компанії в цьому секторі багато в чому залежить від здатності реагувати на швидкі зміни ринку та задовольняти зростаючі вимоги клієнтів. Спостерігається активний розвиток електронної комерції, мобільних платформ, безконтактних платежів, що створює нові виклики для традиційних форматів торгівлі [2].

Значну роль у цьому відіграє цифрова присутність бренду, зокрема функціональність веб-сайту або мобільного застосунку. Покупці очікують зручного сервісу, швидкої доставки, актуального асортименту та можливості взаємодії з брендом у режимі 24/7. У зв'язку з цим, компанії повинні не лише адаптувати свої бізнес-моделі до цифрового середовища, але й активно впроваджувати сучасні технології автоматизації [3].

Сфера роздрібної торгівлі є однією з ключових складових економічного життя суспільства, впливаючи на рівень життя населення, створення робочих місць та розвиток загальної інфраструктури. Протягом останніх десятиліть ця галузь зазнала значних змін, пов'язаних зі стрімким розвитком технологій, глобалізацією економіки та зміною споживацьких звичок. Сучасна торгівля характеризується високою конкуренцією, великою кількістю різноманітних товарних категорій, а також швидким оновленням асортименту товарів. Тому

управління торговельною мережею має бути максимально гнучким, ефективним і адаптивним до мінливих умов зовнішнього середовища.

1.1.2 Проблеми неавтоматизованих торговельних мереж

Відсутність автоматизації в торговельних підприємствах призводить до низки системних проблем. Ручний облік товарів є джерелом неточностей і помилок, особливо в умовах великого обсягу операцій. У разі, коли дані зберігаються в розрізнених таблицях або на різних комп'ютерах без синхронізації, виникає ризик втрати інформації, дублювання записів, помилок у ціноутворенні [4].

Також ускладнюється контроль залишків товарів, відстеження партій, робота зі знижками, поверненнями й взаєморозрахунками з постачальниками. У результаті компанія втрачає гнучкість, знижується ефективність планування, а прийняття рішень часто ґрунтується на застарілій або неповній інформації. Такі умови стримують розвиток підприємства та зменшують його конкурентоспроможність.

Незважаючи на технологічний прогрес, велика кількість підприємств роздрібної торгівлі досі працюють за застарілими схемами управління. Облік товарів часто ведеться вручну або в табличних редакторах, які не дозволяють швидко аналізувати ситуацію на ринку. Відсутність централізованої бази даних призводить до дублювання інформації, помилок у ціноутворенні та проблем із контролем залишків. У таких умовах втрачається гнучкість управління, виникають затримки в поставках і погіршується якість обслуговування клієнтів.

1.1.3 Роль цифрової трансформації у торгівлі

Цифрова трансформація не обмежується лише впровадженням нових ІТ-рішень, це фундаментальна зміна у способі ведення бізнесу. Веб-застосунки у

сфері торгівлі забезпечують єдиний інформаційний простір, де в режимі реального часу можуть взаємодіяти всі відділи компанії: склад, каса, бухгалтерія, логістика, аналітика та керівництво [5]. Це дає змогу оптимізувати процеси закупівель, управління персоналом, обробки замовлень та маркетингових кампаній.

Цифрові інструменти надають можливість глибшого розуміння клієнтів – їх уподобань, поведінки, історії покупок, що стає основою для впровадження персоналізованого підходу. Інтеграція з CRM-системами, використання штучного інтелекту, автоматичне формування звітів та аналітики – усе це підвищує ефективність роботи компанії та дає змогу досягати кращих бізнес-результатів.

Цифрова трансформація стає необхідною умовою для адаптації бізнесу до нових ринкових умов. Веб-застосунки виступають у ролі ключових інструментів цієї трансформації, дозволяючи швидко обробляти великі обсяги даних, автоматизувати рутинні операції, підвищувати рівень клієнтського сервісу. Інформаційні системи забезпечують прозорість усіх бізнес-процесів та формують єдиний інформаційний простір між магазинами, центральним офісом та складом.

1.1.4 Перспективи використання веб-застосунків

Згідно з глобальними аналітичними дослідженнями, до 2030 року понад 80% усіх рітейл-операцій так чи інакше будуть оцифровані. Це відкриває величезні перспективи для підприємств, які вже сьогодні починають інтегрувати веб-застосунки у свою щоденну діяльність. Хмарні рішення, модульна архітектура, розширені API – усе це дає змогу бізнесу масштабуватись без значних витрат [6].

Особливої популярності набувають платформи, що підтримують мобільну аналітику, візуалізацію даних, систему сповіщень і автоматичне оновлення

контенту. Веб-застосунки також дозволяють створювати гнучкі панелі управління для адміністраторів, реалізовувати багаторівневий контроль доступу, інтегрувати чат-ботів та інші сервіси. Усе це формує нову культуру ведення бізнесу – динамічну, прозору, орієнтовану на дані та клієнта.

З огляду на стрімкий розвиток технологій, майбутнє торгівлі безпосередньо пов’язане з інтеграцією інтелектуальних систем управління. Персоналізовані рекомендації, прогнозування попиту за допомогою штучного інтелекту, автоматизація закупівель та логістики – усе це поступово стає стандартом у розвинутих ринках.

1.2 Вимоги до сучасних веб-застосунків

1.2.1 Основні функціональні та нефункціональні вимоги

Сучасні веб-застосунки для управління торговельними мережами повинні відповідати низці важливих вимог, які визначають їхню функціональність, ефективність та конкурентоспроможність на ринку інформаційних технологій. Основними функціональними вимогами є забезпечення обліку товарів, управління замовленнями, ведення клієнтської бази, генерація звітності, управління ціноутворенням та програмами лояльності. Важливо, щоб усі ці функції були пов’язані між собою та працювали в єдиній логіці процесів, без дублювання дій та даних.

Нефункціональні вимоги охоплюють характеристики системи, які не стосуються безпосередньо бізнес-логіки, але суттєво впливають на користувацький досвід. До них належать: надійність (стабільна робота навіть при великій кількості користувачів), продуктивність (швидкий час відгуку), масштабованість (здатність обробляти зростаючі обсяги даних і користувачів), безпека (захист інформації від несанкціонованого доступу), зручність

користування (інтерфейс, інтуїтивна навігація) та супровідність (простота оновлення та підтримки в майбутньому).

1.2.2 Інтерфейс і зручність використання

Один із ключових факторів успішного впровадження веб-застосунку – це якість його інтерфейсу та зручність у використанні. Простий, логічно структурований і візуально привабливий інтерфейс значно підвищує продуктивність користувачів. Працівники можуть швидше орієнтуватися в системі, здійснювати рутинні операції без помилок та ефективно виконувати завдання, що сприяє загальній ефективності бізнесу.

Адаптивність інтерфейсу є важливою складовою – застосунок повинен коректно відображатися та функціонувати на різних пристроях, включаючи мобільні телефони, планшети й настільні ПК. Це особливо актуально для керівників та менеджерів, які потребують швидкого доступу до системи з будь-якої точки та в будь-який момент часу.

Наявність навчальних підказок, інструкцій, зрозумілих повідомлень про помилки – усе це підвищує зручність користування системою, знижує рівень стресу у нових користувачів та сприяє скороченню періоду адаптації. Зручний інтерфейс також покращує взаємодію між різними відділами підприємства, дозволяючи ефективніше обмінюватися інформацією.

Окремої уваги заслуговує візуальна ієрархія елементів інтерфейсу – кнопки, таблиці, графіки, меню повинні бути логічно згруповані та розміщені відповідно до сценаріїв роботи користувача. Таким чином, якісний інтерфейс не лише покращує досвід користування, а й безпосередньо впливає на якість обслуговування клієнтів і швидкість виконання бізнес-процесів.

1.2.3 Продуктивність і безпека

Продуктивність системи – це її здатність швидко виконувати запити користувачів, обробляти великі обсяги даних і надавати результат без затримок. Це особливо важливо в мережах магазинів, де одночасно можуть працювати десятки й сотні користувачів: касири, менеджери, логісти, адміністратори. Повільна система викликає затримки в обслуговуванні клієнтів, помилки в розрахунках, а також знижує ефективність бізнесу загалом.

Безпека веб-застосунку полягає у збереженні цілісності, конфіденційності та доступності інформації. Сучасні рішення повинні включати багаторівневу автентифікацію, шифрування даних у транзиті та на сервері, регулярні резервні копії, обмеження прав доступу відповідно до ролей користувачів. Надійна система безпеки дає змогу не лише зберегти репутацію компанії, але й уникнути фінансових втрат у разі витоку даних.

1.2.4 Інтеграція та масштабованість

Інтеграція веб-застосунку з іншими інформаційними системами є обов’язковою умовою його ефективного використання в корпоративному середовищі. Мова йде про синхронізацію з бухгалтерськими сервісами, CRM-платформами, платіжними системами, постачальниками, а також службами доставки. Важливо, щоб обмін даними відбувався у реальному часі або з мінімальною затримкою, щоб забезпечити актуальність інформації.

Масштабованість означає здатність системи рости разом із бізнесом – без необхідності її повної заміни. Це включає можливість підключення нових користувачів, відкриття додаткових торгових точок, розширення функціональності через модулі або API. Масштабована система дає змогу компанії розвиватися гнучко та ефективно, не вкладаючи ресурси в переробку існуючої інфраструктури.

1.3 Огляд існуючих рішень

Сучасний ринок веб-застосунків для управління торговельними мережами характеризується наявністю великої кількості рішень, що відрізняються за своїми функціональними можливостями, призначенням, масштабом використання та ціновою політикою. На сьогодні існує широкий спектр програмних продуктів, які дозволяють автоматизувати різні аспекти діяльності торговельних підприємств, що значно полегшує управління бізнесом та підвищує ефективність роботи компаній.

Серед представлених рішень на ринку можна виділити декілька основних категорій:

1. CRM-системи (Customer Relationship Management) – веб-застосунки, орієнтовані на управління взаємовідносинами з клієнтами, що дозволяють вести клієнтську базу, керувати програмами лояльності, аналізувати дані щодо покупців.
2. ERP-системи (Enterprise Resource Planning) – комплексні веб-рішення для управління ресурсами підприємства, що інтегрують фінансові, логістичні, кадрові та інші аспекти діяльності компанії.
3. WMS-системи (Warehouse Management System) – рішення для автоматизації складських процесів, які дозволяють контролювати рух товарів, оптимізувати використання складських приміщень та забезпечувати швидке виконання замовлень.
4. POS-системи (Point of Sale) – програмні продукти, які забезпечують автоматизацію процесу продажу в торгових точках, контроль над касовими операціями та оперативне управління товарними запасами.

Кожна з цих категорій має свої особливості та переваги, і вибір конкретного рішення залежить від потреб та специфіки діяльності торговельної мережі. Надалі детальний огляд наявних на ринку рішень дасть змогу окреслити

найрезультативніші підходи та підібрати інструменти, що найкраще відповідатимуть завданням проєкту й специфічним умовам бізнесу.

1.3.1 Salesforce Commerce Cloud

Salesforce Commerce Cloud – хмарна платформа, орієнтована на середній та великий бізнес у сфері електронної комерції. Вона надає змогу користувачам ефективно організовувати цифрову торгівлю, використовуючи потужний набір інструментів для управління товарами, клієнтами, замовленнями та маркетингом. Інтерфейс платформи є сучасним і інтуїтивно зрозумілим, що забезпечує комфортну роботу з великою кількістю даних та налаштувань. Salesforce Commerce Cloud підтримує персоналізацію взаємодії з клієнтами, адаптивне відображення контенту, створення рекламних кампаній, а також інтеграцію з іншими сервісами компанії Salesforce [7]. На рисунку 1.1 представлено головну сторінку інтерфейсу платформи.

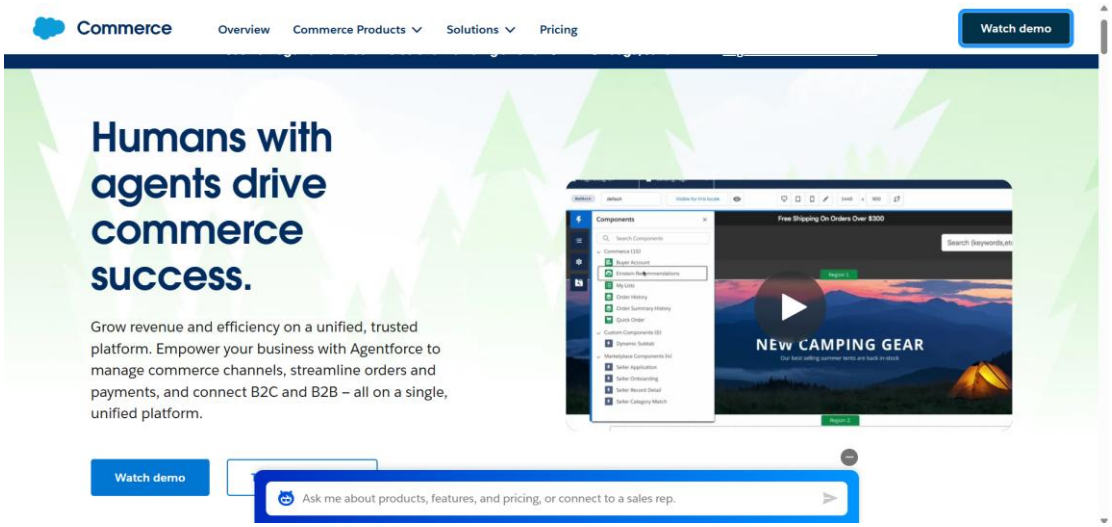


Рисунок 1.1 – Головна сторінка платформи Salesforce Commerce Cloud

Ця платформа дає змогу керувати каталогом товарів, формувати індивідуальні пропозиції, налаштовувати системи знижок і акцій, інтегрувати

CRM-функціонал, підключати платіжні сервіси та створювати аналітичні звіти. Її функціональність адаптується до потреб конкретного бізнесу.

1.3.2 Poster

Poster – спеціалізована система автоматизації для закладів громадського харчування та роздрібних магазинів. Система дає змогу ефективно управляти продажами, складами, фінансами, персоналом та програмами лояльності. Завдяки зручному та зрозумілому інтерфейсу Poster значно спрощує повсякденні бізнес-операції, надаючи менеджменту необхідні інструменти для швидкого ухвалення рішень та оперативного контролю за діяльністю підприємства. Poster відомий своєю простотою інтеграції з різноманітним касовим обладнанням та гнучкістю налаштувань, що дає змогу адаптувати систему під особливості будь-якого бізнесу [8]. На рисунку 1.2 представлено зовнішній вигляд головної сторінки сайту Poster.

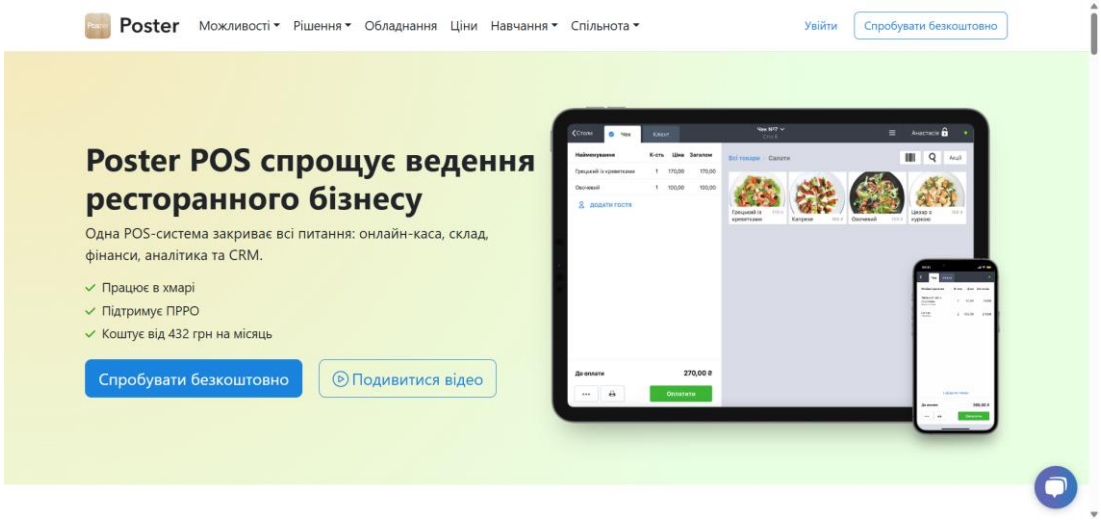


Рисунок 1.2 – Головна сторінка вебсайту Poster

Інтерфейс застосунку Poster забезпечує користувачам доступ до необхідної інформації в реальному часі, дозволяючи швидко реагувати на зміни попиту і пропозиції та оптимізувати товарні запаси.

1.3.3 Torgsoft

Torgsoft – комплексне рішення для автоматизації підприємств роздрібної та оптової торгівлі. Система орієнтована на забезпечення повного контролю над усіма аспектами бізнесу, включаючи продажі, закупівлі, операції зі складськими запасами, фінанси, персонал та звітність. Torgsoft дає змогу комплексно вирішувати управлінські завдання завдяки широким можливостям налаштування й гнучкості системи [9]. На рисунку 1.3 показано інтерфейс програми.

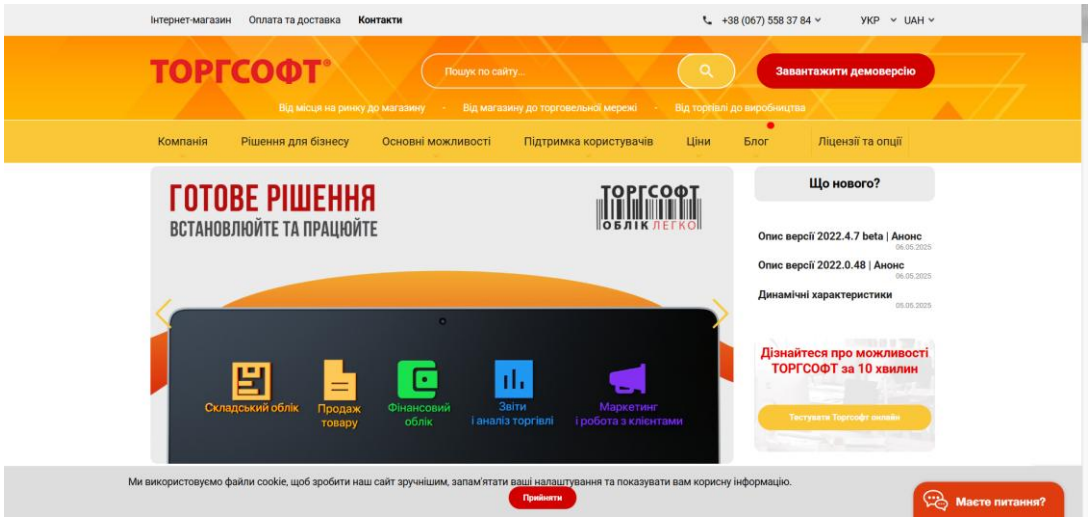


Рисунок 1.3 – Головне вікно веб-застосунку Torgsoft

Платформа дає змогу автоматизувати облік товарних запасів, контролювати продажі та закупівлі, аналізувати фінансові потоки та оптимізувати роботу персоналу. Система легко інтегрується з різним касовим обладнанням та може бути налаштована для роботи з різними типами підприємств і торговельних мереж. Крім того, Torgsoft пропонує користувачам потужні аналітичні інструменти для створення докладних звітів, що дає змогу приймати обґрунтовані управлінські рішення на основі актуальних і точних даних.

1.3.4 Порівняльний аналіз розглянутих рішень

На основі описаних характеристик Salesforce Commerce Cloud, Poster та Torgsoft виконано порівняльний аналіз, покликаний визначити вимоги майбутніх користувачів і сформувати технічні орієнтири подальшої розробки. У таблиці 1.1 наведено зіставлення трьох систем за ключовими параметрами, що впливають на вибір програмного забезпечення для торговельної мережі.

Таблиця 1.1 – Порівняльна характеристика веб-застосунків для управління торговельною мережею

Параметр	Salesforce Commerce Cloud	Poster	Torgsoft
Цільова аудиторія	Середній та великий бізнес	Ресторани, магазини малого бізнесу	Роздрібні та оптові мережі
Основний функціонал	Е-комерція, маркетинг, аналітика	POS-система, склад, персонал	Склад, фінанси, звітність
Локалізація	Міжнародна	Українська	Українська
Модель використання	Хмарна підписка	Хмарна + локальна	ПЗ з ліцензією, локальна
Гнучкість налаштувань	Висока	Середня	Висока
Інтеграції	API, CRM, ERP	POS-обладнання, бухгалтерія	Обладнання, фінанси, CRM

Аналіз показує, що кожне з рішень орієнтоване на свою категорію бізнесу. Salesforce Commerce Cloud надає широку функціональність, але потребує значних ресурсів і знань для впровадження. Poster простіший у використанні та зручний для малого бізнесу, але менш масштабований. Torgsoft поєднує

широкий функціонал з орієнтацією на український ринок, проте поступається в гнучкості хмарних сервісів. Отже, підсумкове порівняння дає змогу швидко оцінити сильні й слабкі сторони кожної платформи та обрати ту, що найточніше відповідає потребам конкретної торговельної мережі.

1.3.5 Недоліки та обмеження наявних платформ

Попри велику кількість функцій, жодне з розглянутих рішень не є універсальним і бездоганним. Salesforce Commerce Cloud часто виявляється надмірно складним і дорогим для невеликих компаній. Його впровадження потребує значного часу, ресурсів і спеціалізованого IT-персоналу. Така система зазвичай виправдана лише для масштабних підприємств із розвиненим IT-відділом.

Poster має низький поріг входу та приваблює користувачів простим інтерфейсом і базовими функціями. Проте виявляється недостатньо потужним для середнього бізнесу, який потребує глибокої аналітики, прогнозування продажів і складної системи звітності. Також Poster обмежений у гнучкості налаштувань і може не підтримувати повноцінну інтеграцію з усіма популярними бізнес-системами.

Torgsoft добре адаптований до українського ринку, має багатий функціонал та підтримує взаємодію з касовим обладнанням. Проте через те, що система є локальною, її важко розгорнути в хмарній інфраструктурі. Це обмежує можливості масштабування та використання в розподілених мережах магазинів, де потрібен швидкий доступ до даних з будь-якої точки.

Таким чином, сучасний ринок потребує більш універсальних рішень, які поєднували б гнучкість, адаптивність і локалізацію – особливо для малого та середнього бізнесу в Україні.

ВИСНОВОКИ ДО РОЗДІЛУ 1

У результаті проведеного аналізу встановлено, що роздрібна торгівля, попри активну цифровізацію, все ще стикається з низкою проблем, пов'язаних із недостатнім рівнем автоматизації та фрагментарним впровадженням ІТ-рішень. У межах розділу досліджено як загальні тенденції розвитку ринку, так і конкретні виклики, з якими стикаються підприємства під час організації ефективної системи управління мережею магазинів.

Зокрема, виявлено, що ручне ведення обліку, відсутність інтегрованих баз даних, нестача аналітичних інструментів та складність масштабування є ключовими стримуючими чинниками для розвитку малих та середніх торговельних компаній. Цифрова трансформація, що передбачає широке впровадження веб-застосунків, виступає не лише як тренд, а як необхідність для збереження конкурентоспроможності.

Веб-застосунки, як показав аналіз, повинні відповідати низці важливих вимог – бути гнучкими, безпечними, масштабованими, мати інтуїтивно зрозумілий інтерфейс і водночас інтегруватися з іншими корпоративними системами.

У ході огляду вже існуючих рішень розглянуто як міжнародні системи (Salesforce), так і локальні (Poster, Torgsoft), кожна з яких має свої переваги та обмеження. Проведене порівняння дозволило зробити висновок, що жодне з них не забезпечує повною мірою потребу малого та середнього бізнесу в Україні.

Отже, на основі проведеного дослідження сформовано обґрунтовану необхідність розробки нового веб-застосунку, який буде адаптований до українського ринку, відповідатиме сучасним технологічним вимогам і забезпечуватиме ефективне управління мережею магазинів. Такий підхід дозволить уникнути обмежень, притаманних типових рішень, і забезпечить

індивідуальну гнучкість, швидке впровадження та оптимізацію бізнес-процесів відповідно до реальних потреб ринку.

РОЗДІЛ 2
ВИБІР ТЕХНОЛОГІЙ РОЗРОБКИ

2.1 Серверна частина

2.1.1 Мова програмування JavaScript

JavaScript – одна з найпопулярніших мов програмування у світі, яка традиційно використовується для розробки клієнтської частини веб-застосунків, а також активно застосовується для створення серверної логіки завдяки платформі Node.js. Її універсальність, велика екосистема та активна спільнота зробили JavaScript стандартом у сучасній веб-розробці.

Основною перевагою JavaScript є можливість реалізовувати як frontend, так і backend логіку в межах одного стеку технологій. Це значно спрощує процес розробки, тестування та підтримки проєкту. Завдяки цьому зменшується кількість контекстних перемикань між мовами програмування.

Мова підтримує об'єктно-орієнтовану, функціональну та подієво-орієнтовану парадигми програмування. Це дає змогу створювати гнучку та масштабовану архітектуру застосунків, відповідно до потреб бізнесу.

JavaScript також має потужну інфраструктуру – наявність численних фреймворків (React, Vue, Angular), серверних бібліотек (Express.js), ORM-рішень (Prisma, Sequelize) та готових пакетів (через npm) забезпечують швидку розробку і мінімізують потребу в написанні великої кількості базового коду вручну.

Для даного проєкту вибір JavaScript є логічним рішенням через його:

- широке використання в індустрії;
- підтримку з боку основних браузерів та серверних середовищ;

- наявність великої кількості навчальних ресурсів та прикладів;

Таким чином, JavaScript забезпечує ефективність розробки, масштабованість та легкість підтримки веб-застосунку для управління мережею магазинів.

2.1.2 Фреймворк Node.js + Express

При створенні серверної частини веб-застосунку для управління мережею магазинів особливу увагу слід приділити обраному фреймворку, який забезпечить надійність, швидкодію, простоту масштабування та активну підтримку спільноти. Після аналізу сучасних рішень, таких як Spring Boot, Django, Laravel, обрано саме Node.js у зв’язці з Express.js – одну з найпопулярніших і перевірених технологій для розробки серверних веб-застосунків [10, 11].

Node.js – це середовище виконання JavaScript на сервері, яке базується на подієвій неблокуючій моделі. На відміну від традиційних серверних рішень, Node.js дає змогу обробляти багато одночасних запитів з мінімальними витратами ресурсів. Це ідеально підходить для проєктів, де важлива висока продуктивність та швидкий відгук – як у випадку з мережею магазинів, де передбачається велика кількість одночасних користувачів (каси, адміністрація, аналітика тощо).

Express.js – це легкий фреймворк, створений спеціально для Node.js, який значно спрощує створення веб-сервера та маршрутизацію. Він надає зручний інтерфейс для налаштування обробників запитів, роботи з middleware, сесіями, cookies та іншими компонентами, що використовуються у сучасних застосунках.

Основні переваги цієї зв’язки:

- Універсальність: можливість використовувати одну мову програмування (JavaScript) як на фронтенді, так і на бекенді;

- Швидка розробка: мінімальна кількість шаблонного коду, велика кількість бібліотек;
- Простота масштабування: можливість горизонтального масштабування з використанням кластерів;
- Активна спільнота: велика кількість відкритих пакетів (npm), рішень і документації;
- Гнучкість: розробник самостійно обирає структуру проєкту, бібліотеки та підхід до організації логіки.

Ця зв'язка також легко інтегрується з базами даних, системами авторизації, інструментами логування, аналітики, обробки платежів тощо. Express.js не накладає обмежень на структуру проєкту, що дає змогу адаптувати архітектуру під специфічні вимоги бізнесу.

У межах розробки веб-застосунку для керування мережею магазинів ця технологія дає змогу ефективно організувати роботу з товарами, замовленнями, аналітичними модулями, а також забезпечити швидкий обмін даними між сервером і клієнтом через REST API. Усе це робить Node.js + Express.js оптимальним вибором для серверної частини даного проєкту.

2.1.3 База даних PostgreSQL

Серед багатьох систем керування базами даних, доступних для використання у веб-розробці, PostgreSQL вирізняється своєю надійністю, функціональністю та стабільністю, що й стало основною причиною її вибору в межах цього проєкту. Створювана онлайн-платформа для адміністрування торговельної мережі має працювати з великою кількістю взаємопов'язаних даних – таких як інформація про товари, клієнтів, замовлення, знижки, точки продажу, працівників тощо. Тому важливою вимогою до бази даних є можливість швидко і безпечно обробляти складні запити, підтримувати

транзакції, забезпечувати цілісність даних та можливість розширення у майбутньому. PostgreSQL повністю задовольняє ці вимоги [12].

На відміну від більшості інших безкоштовних рішень, ця система активно підтримує роботу з великою кількістю одночасних з'єднань, високонавантаженими таблицями та має вбудовані механізми для резервного копіювання, що особливо важливо для бізнесу, де втрата даних може призвести до фінансових збитків. PostgreSQL також надає підтримку складних типів даних, включаючи масиви, JSON, географічну інформацію, що відкриває широкі можливості для розширення функціоналу застосунку в майбутньому.

Ще однією важливою перевагою є гнучка система керування правами доступу: адміністратор може визначити точні межі дозволів для кожного користувача або групи. Це дає змогу гарантувати безпеку даних і мінімізувати ризики несанкціонованого доступу. Також PostgreSQL добре інтегрується з популярними інструментами розробки, включаючи серверне оточення Node.js, з яким працює фреймворк Express. Це робить взаємодію між застосунком і базою даних зручною, швидкою й передбачуваною.

Слід відзначити і відкритий характер PostgreSQL: це програмне забезпечення з відкритим кодом, що дає змогу уникнути ліцензійних витрат, а також гнучко модифікувати систему під конкретні потреби підприємства. Активна світова спільнота користувачів і розробників забезпечує постійну підтримку, випуск оновлень та вдосконалення функцій, що робить PostgreSQL не лише технологічно просунутим, а й перспективним рішенням.

Таким чином, вибір PostgreSQL для реалізації цього веб-застосунку є цілком обґрунтованим з позицій безпеки, стабільності, зручності використання, відкритості та довгострокової підтримки, що особливо важливо у розробці бізнес-орієнтованих систем.

2.1.4 ORM Prisma

Для зручної та безпечної взаємодії з базою даних у межах цього проєкту обрано інструмент Prisma. Це сучасна ORM-технологія (об’єктно-реляційне відображення), яка значно спрощує роботу з базами даних у серверному JavaScript- або TypeScript-застосунку. Prisma дає змогу розробникам описувати структуру бази даних за допомогою зрозумілої декларативної мови, а також генерує зручний клієнт для взаємодії з таблицями через код [13].

Основною перевагою Prisma є простота використання та висока швидкість розробки. Розробник може швидко створювати, оновлювати або видаляти записи в базі даних, не використовуючи сирий SQL-код. Це не лише підвищує безпеку, а й зменшує кількість помилок. Prisma має хорошу інтеграцію з PostgreSQL, яку обрано як основну СУБД у цьому проєкті, а також підтримує автоматичну генерацію міграцій – змін у структурі таблиць.

Ще одним важливим аргументом на користь Prisma є її прозорість і наочність. Весь стан схеми бази даних зберігається в одному файлі, що спрощує командну роботу і контроль версій. Крім того, Prisma пропонує зручний інструмент Prisma Studio – графічний інтерфейс для перегляду та редагування даних, що дуже корисно на етапі тестування.

Таким чином, використання Prisma як ORM-рішення дає змогу зробити роботу з базою даних більш структурованою, безпечною та ефективною, що особливо актуально в межах розробки складного веб-застосунку для управління мережею магазинів.

2.2 Клієнтська частина

2.2.1 Фреймворк React

Для реалізації клієнтської частини веб-застосунку обрано React – одну з найпоширеніших та найпотужніших бібліотек для створення сучасних інтерфейсів. React дає змогу реалізовувати швидкі, інтерактивні та масштабовані веб-додатки завдяки компонентному підходу до розробки [14].

Однією з головних переваг React є здатність формувати високодинамічні інтерфейси: перезавантажується лише та частина сторінки, яка справді змінюється. Це підвищує швидкість роботи й забезпечує плавний, зручний досвід взаємодії з даними – критичний для системи управління мережею магазинів.

Використання React також дає змогу значно спростити підтримку і розширення веб-застосунку завдяки модульній структурі. Кожен елемент інтерфейсу розробляється як окремий компонент, який можна повторно використовувати у різних частинах системи. Це зменшує дублювання коду, підвищує читабельність і полегшує тестування окремих елементів.

Крім того, активна спільнота розробників React забезпечує постійний розвиток інструменту, оновлення безпеки, а також наявність великої кількості навчальних матеріалів та підтримки.

У контексті даного проєкту, React забезпечує всі необхідні можливості для створення ефективного, зручного і гнучкого інтерфейсу користувача, адаптованого до потреб системи управління мережею магазинів.

2.2.2 Стилізація Tailwind CSS

Для стилізації інтерфейсу веб-застосунку обрано Tailwind CSS – сучасну утилітарну CSS-бібліотеку, яка суттєво спрощує процес створення зовнішнього вигляду веб-сторінок. Основна ідея Tailwind полягає у використанні невеликих CSS-класів з чітко визначеними властивостями, які можна комбінувати без написання власного CSS-коду. Це дає змогу розробникам створювати адаптивні та привабливі інтерфейси без потреби створювати окремі стилі для кожного елементу.

Однією з ключових переваг Tailwind CSS є його гнучкість. Завдяки системі утиліт розробник отримує повний контроль над відступами, розмірами, кольорами, типографікою, тінями та іншими параметрами без використання шаблонних класів. Це особливо корисно при створенні інтерфейсів для веб-застосунків з індивідуальними дизайнерськими вимогами.

Ще однією перевагою є можливість швидкого прототипування. Завдяки великій кількості готових утиліт Tailwind дає змогу створювати інтерфейс "на льоту", без потреби перемикатися між CSS-файлом і HTML-структурою. Це підвищує продуктивність роботи та пришвидшує процес розробки [15].

Таким чином, використання Tailwind CSS у цьому проєкті забезпечує швидку, зручну та масштабовану стилізацію веб-застосунку, що є критично важливим при створенні сучасного інтерфейсу для системи управління мережею магазинів.

2.2.3 Компоненти Material UI

Для реалізації компонентної структури інтерфейсу в межах веб-застосунку обрано бібліотеку Material UI – одну з найпопулярніших бібліотек інтерфейсних компонентів, створених для React. Вона базується на принципах дизайн-системи Material Design, розробленої компанією Google, і надає

розробникам набір готових, сучасних, естетично привабливих компонентів, таких як кнопки, таблиці, діалоги, форми, меню, сповіщення тощо.

Однією з основних причин вибору саме Material UI є її глибока інтеграція з React, що дає змогу будувати інтерфейси з повторно використовуваних компонентів. Це значно пришвидшує розробку і полегшує підтримку коду. Завдяки попередньо стилізованим компонентам розробник може сконцентруватися на логіці застосунку, не витрачаючи багато часу на візуальну частину.

Material UI також підтримує кастомізацію на глибокому рівні. Це означає, що навіть попри наявність фірмового стилю Google, дизайн компонентів можна легко адаптувати до потреб конкретного проєкту. Колірна схема, шрифти, відступи, стилі наведення – усе це може бути змінено завдяки вбудованій тематизації.

Крім цього, бібліотека забезпечує повну адаптивність компонентів, що дає змогу створювати коректне відображення інтерфейсу на різних пристроях. Це є важливим аспектом для веб-застосунку, який може використовуватись як у великій торговельній мережі на стаціонарних комп’ютерах, так і на планшетах чи ноутбуках у точках продажу.

Не менш важливо, що Material UI має активну спільноту, регулярні оновлення, підтримку сучасних стандартів JavaScript та доступну документацію. Це робить її надійним вибором не лише для невеликих застосунків, а й для масштабних комерційних проєктів [16].

Отже, вибір Material UI як основної компонентної бібліотеки забезпечує швидку побудову зручного, уніфікованого та функціонального інтерфейсу користувача, що відповідає очікуванням сучасних користувачів і вимогам бізнесу.

2.3 Схема взаємодії компонентів

Архітектура системи базується на трьох узгоджених рівнях: клієнтському, серверному та рівні даних. Користувач працює через браузерний інтерфейс, який формує HTTP-запити з JWT-токеном для автентифікації. Запити потрапляють до REST-API на Node.js / Express, де перевіряються права доступу, виконується бізнес-логіка й формуються звернення до сховища даних.

Наступний етап обробки відбувається на рівні бази: ORM-шар Prisma транслює об'єктні виклики у SQL-операції PostgreSQL і забезпечує атомарність транзакцій. Отримані результати повертаються у форматі JSON тим самим захищеним каналом, що підтримує цілісність і конфіденційність інформації на всіх етапах обміну. Схематичне представлення потоків даних наведено на рисунку 2.1.

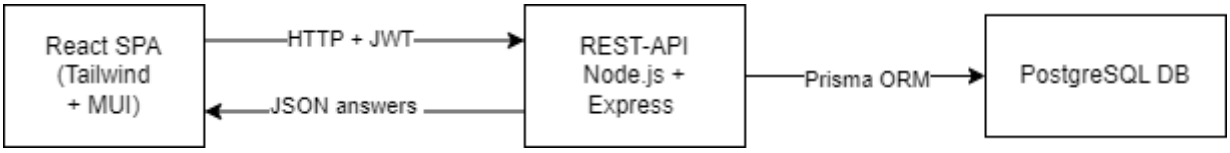


Рисунок 2.1 – Узагальнена взаємодія фронтенду, REST-API та бази даних

Така трирівнева організація дає змогу незалежно масштабувати кожен компонент, централізовано реалізувати політики безпеки й зберігати єдине «джерело правди» у базі даних.

2.4 Інструменти розробки

2.4.1 Редактор Visual Studio Code

У процесі створення веб-застосунку для управління мережею магазинів обрано Visual Studio Code як основне середовище розробки. Це сучасний, легкий та водночас потужний текстовий редактор, який підтримує розробку на

багатьох мовах програмування та активно використовується як у професійному середовищі, так і в освітніх проєктах.

Однією з головних переваг Visual Studio Code є його розширюваність. Завдяки великій кількості плагінів та розширень можна налаштувати середовище під потреби конкретного проєкту. Наприклад, для зручної роботи з JavaScript, TypeScript, React, Node.js чи Prisma існують спеціалізовані плагіни, які надають підсвічування синтаксису, автодоповнення, перевірку типів, навігацію по коду та форматування.

Інтегрований термінал дає змогу запускати команди безпосередньо у вікні редактора, що підвищує ефективність розробки. Крім того, Visual Studio Code має зручний інтерфейс для роботи з системами контролю версій, зокрема Git. Це дає змогу керувати репозиторієм, створювати гілки, переглядати зміни та виконувати злиття без необхідності використовувати сторонні інструменти.

Ще однією важливою перевагою є підтримка відлагодження (debugging) безпосередньо в редакторі. Visual Studio Code дає змогу запускати застосунок у режимі налагодження, додавати точки зупину, переглядати значення змінних та контролювати виконання крок за кроком.

Завдяки простому інтерфейсу, високій швидкості роботи та активній спільноті, Visual Studio Code забезпечує комфортне середовище для ефективної командної та індивідуальної розробки. Саме ці якості роблять його доцільним вибором для реалізації проєкту веб-застосунку.

2.4.2 Контроль версій Git + GitHub

У процесі розробки веб-застосунку використовувалася система контролю версій Git у поєднанні з хмарною платформою GitHub. Це дозволило ефективно відстежувати всі зміни в коді, зберігати історію розробки та організовувати роботу над проєктом у вигляді окремих гілок. Такий підхід дає змогу розробнику працювати над новими функціями незалежно від основної версії

застосунку, а також безпечно повертатися до попередніх варіантів у разі помилок або необхідності перегляду змін.

GitHub, як платформа для розміщення репозиторіїв, доповнює Git можливостями командної взаємодії: надає інтерфейс для перегляду змін, дає змогу створювати pull-запити для перевірки коду перед його злиттям, використовувати issue-трекер для ведення задач і проблем, а також автоматизувати процеси перевірки та розгортання коду. Використання Git і GitHub забезпечує високу якість керування розробкою, стабільність та прозорість змін, що робить ці інструменти важливою складовою сучасного веб-проєкту.

2.4.3 Контейнери Docker та інфраструктура Docker Compose

Для створення стабільного, ізольованого та керованого середовища розробки й розгортання веб-застосунку обрано Docker. Цей інструмент дає змогу «упакувати» застосунок разом із усіма його залежностями – серверною частиною (Node.js + Express), базою даних PostgreSQL чи іншими службами – у відокремлені контейнери. Завдяки цьому середовище розробки повністю ідентичне середовищу продакшну, що істотно знижує ризик помилок, пов’язаних із несумісністю конфігурацій [17]. Docker також дає змогу масштабувати проєкт і додавати нові компоненти, не впливаючи на вже запущені сервіси, а ізоляція контейнерів підвищує безпеку системи.

Для спрощення керування декількома контейнерами використовується Docker Compose. Усі потрібні сервіси – фронтенд, бекенд, база даних PostgreSQL та, за потреби, додаткові інструменти моніторингу – описуються в одному файлі docker-compose.yml. Після цього повноцінне середовище піднімається однією командою «docker compose up», що мінімізує «ефект працює-в-мене» та нівелює відмінності між локальним і бойовим серверами [18].

На рисунку 2.2 показано спрощену схему контейнеризації: центральний демон Docker керує трьома основними контейнерами – клієнтським інтерфейсом, серверною частиною та базою даних.

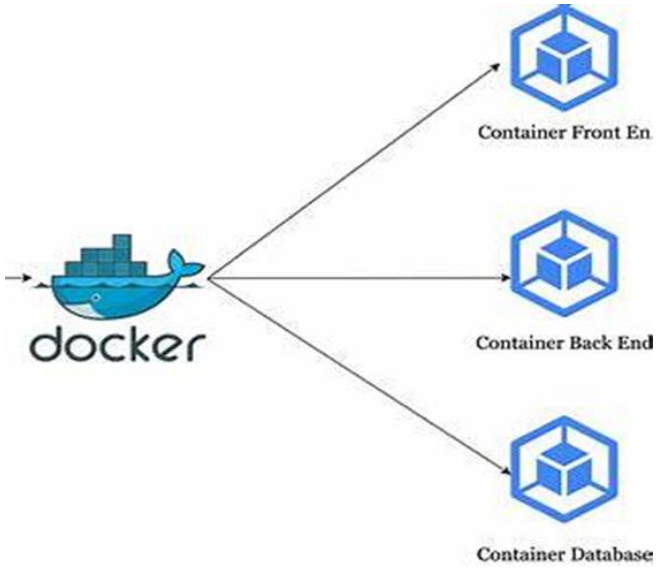


Рисунок 2.2 – Запуск інфраструктури за допомогою Docker Compose

Основні переваги цього підходу:

- простий запуск – новому розробнику достатньо встановити Docker та виконати одну команду;
- уніфіковані залежності й налаштування для розробки, тестування та продакшну;
- можливість легко додавати нові сервіси або змінювати ресурси існуючих без втручання в інші контейнери;
- повна ізоляція компонентів, що спрощує оновлення й підвищує безпеку.

2.5 Додаткові технології

2.5.1 REST API

У процесі розробки веб-застосунку для управління мережею магазинів прийнято рішення використовувати REST API як основний спосіб організації взаємодії між клієнтською (frontend) та серверною (backend) частинами системи. REST – це архітектурний стиль, що базується на принципах простоти, масштабованості та стандартизованої взаємодії через протокол HTTP.

Однією з основних переваг REST є його орієнтація на ресурси, де кожен об’єкт або сутність у системі (наприклад, користувач, товар, замовлення) представлений у вигляді унікального URL-адресу. Операції над цими ресурсами виконуються за допомогою стандартних HTTP-методів:

- GET – для отримання інформації,
- POST – для створення нових записів,
- PUT або PATCH – для оновлення,
- DELETE – для видалення.

Такий підхід робить структуру API зрозумілою для розробників та сприяє кращій підтримці проєкту в майбутньому. REST дає змогу чітко розмежувати логіку клієнтської та серверної частин, завдяки чому застосунок стає більш модульним, зручним для тестування та масштабування.

Використання REST API є також доцільним з точки зору ефективності обміну даними, оскільки запити й відповіді формуються у форматі JSON, що легко обробляється як на фронтенді, так і на сервері. Крім того, REST добре підходить для створення інтерактивних веб-застосунків, які оновлюють дані без повного перезавантаження сторінки.

REST API буде реалізований з використанням фреймворку Express.js, що працює поверх Node.js. Express дає змогу зручно створювати маршрути, обробляти параметри запитів, реалізовувати логіку контролерів та взаємодіяти

з базою даних. У поєднанні з ORM Prisma це забезпечує чисту, зрозумілу та підтримувану серверну архітектуру.

Отже, залучення REST-архітектури повністю виправдане: вона спрощує створення й підтримку сервісу, забезпечує швидкий відгук, залишає простір для подальшого розширення та масштабування. Такий підхід полегшить інтеграцію з зовнішніми системами і надасть єдиний, зручний канал доступу до даних для всіх категорій користувачів.

2.5.2 Авторизація JWT

У рамках розробки веб-застосунку для управління мережею магазинів особливу увагу слід приділити безпеці доступу до даних та автентифікації користувачів. Для реалізації сучасного, зручного та надійного механізму авторизації у даному проєкті використовується технологія JWT.

JWT – це компактний і самодостатній формат передачі інформації між клієнтом і сервером у вигляді токена. Після проходження автентифікації (наприклад, введення логіну та паролю), сервер генерує унікальний токен, який зберігається на клієнті і додається до кожного наступного запиту у заголовку. Сервер, отримуючи такий токен, може швидко ідентифікувати користувача без потреби зберігати стан сесії.

Однією з ключових переваг JWT є те, що він підписується на сервері за допомогою секретного ключа. Це дає змогу перевірити справжність токена та гарантує, що його вміст не було змінено. Токен також може містити корисну інформацію, таку як ID користувача, його роль у системі (наприклад, адміністратор, менеджер, касир), термін дії та інші метадані, що дає змогу більш гнучко керувати доступом до ресурсів.

Завдяки безстанному підходу, JWT чудово працює з REST API. Він дає змогу ефективно організувати захист приватних маршрутів, перевірку прав

доступу, а також спрощує масштабування системи, адже сервер не зберігає жодної інформації про стан сесії [19].

У поєднанні з middleware-функціями Express.js токен авторизації перевіряється автоматично перед кожним запитом до захищених ресурсів, що значно підвищує безпеку та стабільність системи. Крім того, JWT добре підходить для побудови розподілених архітектур та мобільних застосунків, у разі подальшого розвитку системи.

Отже, використання JWT як технології авторизації в даному веб-застосунку забезпечує високий рівень захисту, простоту реалізації та сумісність з обраною архітектурою на основі REST API, що робить його ефективним вибором для проєкту.

.

ВИСНОВОКИ ДО РОЗДІЛУ 2

Узагальнюючи матеріал другого розділу, можна констатувати, що обрано узгоджений стек технологій, на базі якого буде реалізовано розроблювану систему.

На рівні серверної логіки комбінація JavaScript + Node.js + Express забезпечує неблокувальну обробку запитів, єдину мовну парадигму між клієнтом і сервером та спрощує супровід коду. Подієво-орієнтована модель Node.js у поєднанні з гнучкістю Express гарантує високу пропускну здатність, що важливо для багатокористувацького середовища мережі магазинів, де кількість звернень коливається від касових операцій до аналітичних запитів.

Як сховище даних обрано PostgreSQL, що надає повноцінні транзакції, цілісність і підтримку складних типів (JSON, масиви, геодані). Завдяки цьому база витримує високі навантаження та залишається гнучкою для майбутніх бізнес-задач, зокрема інтеграції програм лояльності або просторової аналітики. Prisma ORM піднімає рівень абстракції над SQL, пропонує декларативний опис схеми й автоматичні міграції, підвищуючи безпеку запитів і швидкість упровадження змін.

На фронтенді React слугує «двигуном» інтерфейсу, дозволяючи розбити складну взаємодію на ізольовані компоненти, що легко тестуються та масштабуються. Tailwind CSS прибирає зайві шари стилів, роблячи макети прозорими без переходів між HTML і окремими CSS-файлами, а Material UI забезпечує набір перевірених компонентів із готовими сценаріями доступності та адаптивності. Таке поєднання дає змогу швидко формувати єдиний візуальний стиль у межах вимог бренду торговельної мережі.

Комунікаційний шар реалізовано як REST-службу з JWT-автентифікацією, що надає захищений і розширюваний доступ до ресурсів без потреби зберігати

сесійні дані на сервері. Завдяки цьому сервер легко масштабується горизонтально, а клієнти отримують швидку перевірку прав на кожному запиті.

У частині інструментів Visual Studio Code з багатою екосистемою розширень забезпечує продуктивну роботу, а Git та GitHub фіксують зміни коду й підтримують автоматизоване розгортання. Docker і Docker Compose створюють відтворювані середовища, усувають проблему «працює тільки в мене» та гарантують повторюваність збірок, що критично для оперативного випуску оновлень.

Узгоджене функціонування всіх наведених компонентів формує надійну, масштабовану та легко підтримувану платформу. Отже, розглянутий і обраний стек оптимально відповідає вимогам дипломного проєкту та закладає міцний фундамент для подальшої розробки й еволюції систем.

РОЗДІЛ 3
ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ВЕБ-ЗАСТОСУНКУ

3.1 Архітектура застосунку

3.1.1 Компоненти клієнта та сервера

У реалізації веб-застосунку для управління мережею магазинів використано чітко структурований підхід до побудови клієнтської та серверної частини. Код організовано модульно, з логічним поділом на функціональні блоки, що відповідають за конкретні аспекти роботи системи.

Серверна частина реалізована як RESTful API, що обслуговує всі запити від клієнта. Вона складається з маршрутів, кожен із яких відповідає певній сутності, зокрема магазинам, товарам, працівникам, продажам і профілям користувачів. До кожного маршруту застосовується проміжна обробка запитів через middleware-функції: зокрема, функція authenticate перевіряє дійсність JWT-токена, а authorizeRoles контролює доступ до маршрутів відповідно до ролі користувача. Додаткова логіка реалізується у вигляді окремих функціональних блоків: наприклад, перевірка залишків товару перед продажем або обчислення загальної суми продажів.

Доступ до бази даних здійснюється через Prisma ORM, яка забезпечує зручну та типізовану роботу з таблицями PostgreSQL. Структура моделей у Prisma точно відображає бізнес-логіку застосунку. Завдяки цьому стало можливим підтримувати чисту архітектуру серверної частини, де бізнес-логіка чітко відокремлена від рівня зберігання даних.

Клієнтська частина реалізована у вигляді односторінкового застосунку, побудованого на базі React. Інтерфейс поділений на окремі сторінки, зокрема

сторінку автентифікації та сторінку профілю користувача. Основні функціональні блоки реалізовано у вигляді незалежних компонентів, таких як StoreCard, ProductCard, EmployeeCard та SaleCard. Вони відповідають за візуалізацію, введення даних, модальні діалоги й інтеграцію з API.

На клієнті реалізовано механізм контролю доступу: інтерфейс автоматично адаптується відповідно до ролі користувача, яка зберігається після автентифікації. Наприклад, адміністратор має повний доступ до всіх даних, тоді як касир може взаємодіяти лише з товарами свого магазину. Стан даних керується за допомогою React-хуків (useState, useEffect), а всі запити до серверної частини надсилаються через бібліотеку Axios [20].

Загалом архітектура застосунку побудована з урахуванням принципів розділення відповідальності, модульності та масштабованості, що спрощує підтримку, налагодження та можливе розширення функціональності в майбутньому.

3.1.2 Взаємодія між частинами застосунку

Вся комунікація між клієнтською та серверною частинами здійснюється через HTTP-запити відповідно до RESTful-архітектури. Клієнт надсилає запити у форматі JSON і отримує відповідь у такому ж форматі, що забезпечує уніфікований спосіб обміну даними.

Процес взаємодії починається з автентифікації: користувач вводить логін і пароль, після чого клієнт надсилає POST-запит до маршруту /login. У відповідь сервер повертає токен доступу у форматі JWT, який зберігається у браузері у localStorage. Усі подальші запити до захищених ресурсів супроводжуються заголовком авторизації у форматі Authorization: Bearer <token>.

При отриманні такого запиту сервер виконує перевірку токена за допомогою middleware. Якщо токен дійсний, користувач допускається до

маршруту, доступ до якого також регулюється його роллю. Таким чином, реалізовано як автентифікацію, так і авторизацію доступу.

На клієнті кожна сутність, з якою може взаємодіяти користувач (магазини, товари, працівники, продажі), представлена окремим компонентом. Ці компоненти самостійно виконують запити до API – наприклад, при створенні, оновленні або видаленні елементів. Результати запитів миттєво оновлюються на інтерфейсі без потреби перезавантаження сторінки.

Система також дає змогу адміністратору обирати магазин, з яким він бажає працювати, через відповідний селектор на сторінці профілю. Значення ідентифікатора магазину (storeId) додається до параметрів запиту, що дає змогу отримувати фільтровані дані для кожного конкретного магазину. Для касирів ця логіка автоматизована – вони можуть працювати лише з даними магазину, до якого прив’язані.

Окремий механізм реалізовано для завантаження файлів. Зокрема, користувач може змінити аватарку, надіславши зображення через HTML-форму. Файл передається на сервер у форматі multipart/form-data, зберігається у файловій системі, а посилання на нього записується в базу даних. Сервер надає прямий доступ до файлів через маршрут /uploads.

Завдяки такій взаємодії між частинами система забезпечує повний цикл обробки запитів, швидке оновлення інтерфейсу, гнучкий контроль доступу та захищений обмін даними. Це дає змогу реалізувати надійний, зручний і безпечний інструмент для управління роздрібною мережею.

3.2 Серверна частина

3.2.1 Налаштування Express-сервера

Серверна частина веб-застосунку реалізована за допомогою фреймворку Express, що працює на базі Node.js. Основна мета цієї частини – створення гнучкого й безпечного REST API, через яке клієнтська частина взаємодіє з даними.

На початку проєкту ініціалізується Express-додаток, встановлюється порт для прослуховування, а також підключаються основні middleware-модулі. Для забезпечення обміну даними між фронтендом і бекендом використовується бібліотека cors, яка налаштована так, щоб дозволяти запити з адреси клієнтського інтерфейсу (http://localhost:5173) та передавати куки та заголовки авторизації.

Сервер також конфігуровано на прийом запитів у форматі JSON за допомогою express.json(). Це дає змогу автоматично парсити тіло запитів без додаткових бібліотек.

Для підключення до бази даних PostgreSQL використовується Prisma Client. Ініціалізація Prisma відбувається один раз при запуску сервера через new PrismaClient(), після чого доступ до таблиць бази даних здійснюється через методи prisma.<model>.findMany(), create(), update() тощо.

Окремо реалізовано маршрутизацію HTTP-запитів за типами ресурсів (/stores, /products, /sales, /employees, /profile, /auth) та їхню обробку відповідними функціями. Основні маршрути обгорнуті у middleware для забезпечення перевірки токенів (JWT) та прав доступу, що дає змогу централізовано керувати безпекою всього API.

Усі помилки обробляються централізовано: при недоступності ресурсу повертається статус 404, при помилках на сервері – статус 500, а при порушенні прав доступу – статус 403 або 401.

Окремим компонентом інтегровано multer – middleware для обробки завантаження файлів [21], яке використовується для збереження аватарок користувачів у локальну директорію /uploads. Сервер також відкриває статичний доступ до цієї директорії через маршрут /uploads, що дає змогу клієнту завантажувати зображення профілю.

3.2.2 Реалізація REST API для CRUD-операцій

Для забезпечення повноцінної взаємодії між клієнтом і сервером у веб-застосунку реалізовано REST API, яке підтримує повний набір CRUD-операцій: створення, читання, оновлення та видалення. Це дає змогу ефективно керувати всіма основними сутностями – магазинами, товарами, працівниками, продажами та користувачами.

Кожна сутність має окремий набір маршрутів, які відповідають стандартним HTTP-методам: GET – для отримання даних, POST – для створення, PUT – для оновлення, DELETE – для видалення. Запити обробляються з урахуванням автентифікації та ролей користувачів. Наприклад, створювати нові магазини дозволено лише адміністраторам, а касири можуть працювати лише з товарами власного магазину. На рисунку 3.1 представлено приклад реалізації маршруту POST /stores, який дає змогу створити новий магазин у системі.

```
73 app.post('/stores', authenticate, authorizeRoles(['admin']), async (req, res) => {
74   try {
75     const store = await prisma.store.create({ data: req.body });
76     res.status(201).json(store);
77   } catch (err) {
78     res.status(400).json({ error: 'Failed to create store' });
79   }
80 });
```

Рисунок 3.1 – Код маршруту створення магазину

У цьому прикладі маршрут обгорнутий у middleware authenticate і authorizeRoles(['admin']). Це гарантує, що лише автентифікований користувач з роллю адміністратора може надіслати запит. Об’єкт, що передається у тілі запиту (req.body), зберігається у базі даних через Prisma ORM методом prisma.store.create().

Аналогічно реалізовано маршрути для оновлення (PUT /stores/:id), перегляду (GET /stores та GET /stores/:id) і видалення (DELETE /stores/:id) магазинів. Для інших сутностей, таких як товари чи продажі, також реалізовано повну CRUD-функціональність, але з урахуванням додаткової логіки – наприклад, перевірки залишку товару перед створенням продажу або обмеження доступу касира лише до свого магазину.

API підтримує параметри фільтрації, такі як storeId, що дає змогу динамічно відображати відповідні дані в інтерфейсі адміністратора або касира. Усі відповіді від сервера повертаються у форматі JSON разом зі статусами HTTP, що дає змогу клієнту точно визначати результат операції.

Завдяки дотриманню принципів REST API, використанню middleware для безпеки й ORM для роботи з базою даних, вдалося створити розширюване, типізоване та захищене серверне середовище.

3.2.3 Аутентифікація та авторизація (JWT, ролі)

У системі управління мережею магазинів реалізовано повноцінний механізм автентифікації та авторизації, що забезпечує безпечний доступ до ресурсів. Для цього використовується токен на основі JWT (JSON Web Token), який клієнт отримує після успішного входу в систему.

Під час автентифікації користувач надсилає свої облікові дані (email і пароль) на сервер через маршрут login. У разі успіху сервер формує токен, який містить основні дані про користувача: його id, email, роль і (опційно) storeId,

якщо користувач – касир. Цей токен клієнт зберігає у localStorage та надсилає в заголовках подальших HTTP-запитів як Authorization: Bearer token.

На сервері всі захищені маршрути перевіряються двома проміжними функціями (middleware): authenticate – перевіряє валідність токена, декодує його і додає до об'єкта запиту req.user; authorizeRoles – визначає, чи дозволено користувачу доступ до певного маршруту відповідно до його ролі.

Ці механізми дозволяють ефективно розмежувати права доступу. Наприклад, адміністратор має повний доступ до всіх CRUD-операцій, касир може додавати, редагувати та видаляти товари лише у своєму магазині, а звичайний користувач має доступ лише до перегляду магазинів.

Завдяки такому підходу забезпечується висока гнучкість, безпека та масштабованість системи керування доступом.

3.2.4 Робота з базою даних через Prisma

Для зберігання та обробки даних застосовується база даних PostgreSQL, доступ до якої реалізовано за допомогою ORM Prisma. Prisma дає змогу описувати структуру таблиць, встановлювати зв’язки між ними, а також виконувати CRUD-операції у вигляді типізованих запитів на JavaScript.

Усі моделі описані у файлі schema.prisma. Зокрема, модель User описує структуру облікового запису користувача. Основні поля цієї моделі:

- id – унікальний ідентифікатор користувача, який створюється автоматично.
- email – унікальна адреса електронної пошти, яка використовується для входу в систему.
- password – збережений у хешованому вигляді пароль користувача.
- role – текстове поле, яке визначає роль користувача: admin, cashier або user.

- storeId – опціональне поле, що зберігає ідентифікатор магазину, до якого прив’язаний касир.
- firstName та lastName – персональні дані користувача, які відображаються у профілі.
- avatarUrl – шлях до зображення-аватарки, який використовується на сторінці профілю.

На рисунку 3.2 зображено структуру моделі User у файлі schema.prisma, яка є ключовою для автентифікації, авторизації та персоналізації взаємодії.

```
model User {
  id        Int     @id @default(autoincrement())
  email     String  @unique
  password  String
  role      String  @default("user")
  storeId   Int?
  firstName String?
  lastName  String?
  avatarUrl String?
}
```

Рисунок 3.2 – Структура моделі користувача у Prisma

Кожна роль користувача впливає на його права в системі. Наприклад, касир має обмежений доступ і прив’язаний до конкретного магазину (storeId), у той час як адміністратор не має обмежень і може працювати з усією базою.

Крім базових запитів, Prisma використовується також для складніших операцій: валідації зв’язків, агрегації даних (наприклад, підрахунку суми продажів) та транзакцій.

Таким чином, використання Prisma забезпечує зручну, безпечну та продуктивну роботу з базою даних, що дає змогу підтримувати актуальність та цілісність усіх даних системи.

3.3 Клієнтська частина

Фронтенд веб-застосунку реалізований як односторінковий застосунок з використанням React, Tailwind CSS і Material UI. Його структура забезпечує швидку навігацію, реактивність і чітке розмежування доступу до функціоналу відповідно до ролей користувача.

3.3.1 Сторінка автентифікації

Сторінка автентифікації є першою точкою взаємодії користувача з системою. Вона реалізована як компонент Auth, що забезпечує як реєстрацію нових користувачів, так і вхід до системи. Компонент використовує useState для збереження введених значень (email, пароль, роль, ім’я тощо) та useEffect для динамічного завантаження магазинів у разі вибору ролі «касира».

Валідація пароля реалізована за допомогою регулярного виразу, який перевіряє наявність великої та малої літери, цифри та мінімальну довжину. У разі помилки виводиться відповідне повідомлення. Після успішної реєстрації користувачеві показується підтвердження, а після входу – токен зберігається в localStorage і здійснюється перенаправлення на сторінку профілю.

Усі запити надсилаються до бекенда через axios, що забезпечує взаємодію з маршрутами /login та /register. Дані залежно від ролі містять додаткові поля, як-от storeId для касирів.

Сторінка має адаптивний дизайн із трьома зонами: інформаційний блок, форма автентифікації та ілюстрація. Це дає змогу зробити інтерфейс інтуїтивно зрозумілим та зручним для користувачів.

3.3.2 Профіль користувача

Профіль користувача реалізовано як компонент Profile, який завантажує та відображає персональні дані користувача, перелік доступних магазинів, а також дані про товари, продажі та працівників у залежності від ролі користувача.

При завантаженні сторінки застосовується useEffect, який надсилає запити до сервера з токеном авторизації. Після обробки відповіді у стан компонента записуються відповідні дані. Для адміністраторів доступна повна інформація про магазини, продажі та працівників. Касири працюють лише з товарами, закріпленими за їхнім магазином.

Користувач може змінювати аватар, використовуючи компонент завантаження файлу. Зображення надсилається у форматі multipart/form-data, після чого система оновлює відповідний запис користувача.

Інтерфейс реалізовано адаптивно: блок з інформацією про користувача містить ім’я, email, роль, а також селектор вибору магазину. Основна частина сторінки складається з компонентів-карток: StoreCard, ProductCard, SaleCard і EmployeeCard, які забезпечують перегляд і керування відповідними сутностями. Відображення кожного з блоків контролюється логікою відповідності ролі – наприклад, компоненти SaleCard і EmployeeCard доступні виключно адміністраторам.

Таким чином, профіль користувача є функціональним ядром системи з гнучким контролем доступу, зручною взаємодією та можливістю персоналізації.

3.3.3 Компоненти управління даними

Управління ключовими сутностями системи – магазинами, товарами, працівниками та продажами – реалізовано через окремі компоненти React. Кожен з них використовує типову структуру з таблицею (DataGrid) для

відображення даних, діалогами для редагування або додавання записів, і кнопками для виконання дій. Всі дані отримуються з бекенда через запити axios, з урахуванням авторизації користувача та, за потреби, фільтрації за магазином.

Компонент StoreCard відповідає за управління магазинами і доступний лише для адміністратора. Користувач може додати або змінити назву магазину та його адресу. Компонент ProductCard дає змогу додавати, редагувати або видаляти товари – залежно від ролі користувача та прив'язки до магазину. Наприклад, касир може змінювати лише товари свого магазину, тоді як адміністратор – усі. SaleCard реалізує облік продажів з автоматичним підрахунком вартості, завантаженням списку товарів та перевіркою залишків. Дані про продаж пов'язуються з продуктами, що дає змогу відображати назву товару, ціну, кількість та суму.

Компонент EmployeeCard реалізує функціональність додавання, редагування та видалення працівників магазину. У ньому вказується ім’я, роль (наприклад, cashier або admin) та магазин, до якого працівник належить. Усі компоненти виконують повторне завантаження даних після будь-яких змін, що забезпечує актуальність інформації без перезавантаження сторінки.

Таким чином, система управління забезпечує чітку логіку обробки даних для кожної сутності, адаптовану під роль користувача та специфіку бізнес-процесів.

3.4 Розгортання застосунку

Для зручного розгортання всієї системи використано технології контейнеризації Docker та Docker Compose. Кожна частина застосунку – база даних, серверна частина та клієнт – ізольована у власному контейнері, що полегшує керування, тестування та масштабування. Серверна частина розгортається на базі Node.js, де під час збирання встановлюються необхідні залежності, копіюються файли, генерується Prisma Client і відкривається порт

для обробки запитів до API. Клієнтська частина також автоматизована, що дає змогу швидко запускати її в середовищі розробки з відкритим портом для доступу з браузера.

База даних PostgreSQL функціонує в окремому контейнері. Для збереження даних використано монтування локального тому, а конфігурація сервісів у файлі docker-compose.yml передбачає послідовний запуск усіх компонентів. На рисунку 3.3 наведено приклад конфігурації, що демонструє, як описані вище сервіси взаємодіють між собою за допомогою Docker Compose.

retail-webapp > docker-compose.yml

```yaml
version: "3.9"

services:
  db:
    image: postgres:15-alpine
    environment:
      POSTGRES_USER: retail
      POSTGRES_PASSWORD: retail
      POSTGRES_DB: retail
    volumes:
      - ./dbdata:/var/lib/postgresql/data
    ports:
      - "5432:5432"

  api:
    build: ./api
    environment:
      DATABASE_URL: postgres://retail:retail@db:5432/retail
    ports:
      - "3000:3000"
    depends_on:
      - db

  client:
    build: ./client
    ports:
      - "5173:5173"
    depends_on:
      - api
    volumes:
      - ./client:/app
      - /app/node_modules
    command: npm run dev

```

Рисунок 3.3 – Docker Compose файл для розгортання застосунку

Після запуску команди docker-compose up --build усі компоненти системи автоматично ініціалізуються: база даних запускається першою, за нею стартує сервер, після чого клієнт отримує можливість обробки запитів. Такий підхід дає змогу швидко та безпомилково розгорнути веб-застосунок у будь-якому середовищі з підтримкою Docker.

ВИСНОВОКИ ДО РОЗДІЛУ 3

У третьому розділі детально розглянуто процес проєктування та реалізації веб-застосунку для управління мережею магазинів. Застосунок побудовано на основі сучасних вебтехнологій із чітким розділенням на клієнтську та серверну частини, що взаємодіють через REST API. Архітектура рішення базується на принципах модульності, масштабованості та безпеки, що дає змогу ефективно обробляти запити, керувати даними та розмежовувати доступ згідно з роллю користувача.

Сервер реалізовано з використанням Express.js і Prisma ORM, що забезпечило зручну взаємодію з базою даних PostgreSQL, централізовану автентифікацію, авторизацію та обробку CRUD-запитів. Клієнтська частина побудована як SPA-додаток на базі React із використанням Tailwind CSS та MUI, де кожна сутність реалізована через окремі компоненти з повною інтеграцією в бізнес-логіку застосунку.

Особливу увагу приділено контролю доступу відповідно до ролей користувачів, що реалізовано як на бекенді, так і у фронтенді. Також продемонстровано розгортання системи за допомогою Docker та Docker Compose, що дає змогу швидко запускати застосунок у будь-якому середовищі.

Таким чином, створений застосунок є повноцінною платформою для цифрової трансформації торгівлі та управління торговельною мережею з можливістю подальшого розширення та масштабування.

РОЗДІЛ 4
ОГЛЯД ТА ТЕСТУВАННЯ РОБОТИ ВЕБ-ЗАСТОСУНКУ

4.1 Тестування серверної частини

У процесі розробки серверної частини веб-застосунку значну увагу зосереджено на перевірці її коректності, стабільності та відповідності очікуваній поведінці в умовах реального використання. Це дозволило виявити критичні помилки, уникнути збоїв у взаємодії компонентів і забезпечити надійний захист від несанкціонованих дій. Для моделювання взаємодії з API використано бібліотеку Supertest, яка дає змогу надсилати HTTP-запити безпосередньо до Express-сервера без його запуску на порті [22].

4.1.1 Перевірка повної взаємодії компонентів

Інтеграційні тести перевіряли узгоджену роботу ключових модулів: автентифікації, авторизації, CRUD-операцій з магазинами, товарами, продажами, працівниками та логіки обмеження доступу. Тестування охоплювало:

- створення користувачів з різними ролями;
- вхід із отриманням JWT-токена;
- операції з магазинами, включно з обмеженням прав;
- керування товарами з урахуванням доступу;
- фіксацію продажів із перевіркою оновлення залишку;
- управління працівниками, що виконується адміністраторами;
- перевірку ролей і обмежень доступу.

На рисунку 4.1 показано приклад тесту, де звичайному користувачу заборонено створення продукту.

```
204  // --- Спроба створити продукт як звичайний користувач ---
205  test('Звичайний користувач не може створити продукт', async () => {
206    const res = await request(api)
207      .post('/products')
208      .set('Authorization', `Bearer ${userToken}`)
209      .send({ name: 'fail', price: 1, stock: 1, storeId });
210    expect(res.statusCode).toBe(403);
211  });
```

Рисунок 4.1 – Приклад інтеграційного тесту: заборона створення продукту для звичайного користувача

Цей приклад підтверджує коректність реалізації логіки авторизації: користувач без відповідних прав отримує помилку з кодом 403 (Forbidden).

Ще одним показовим тестом є сценарій створення продажу адміністратором, що демонструє взаємодію між модулем продажів, товарами, магазинами та користувачами. Його представлено на рисунку 4.2.

```
130  // --- Продажі ---
131  test('Створення продажу (адмін)', async () => {
132    const res = await request(api)
133      .post('/sales')
134      .set('Authorization', `Bearer ${adminToken}`)
135      .send({ productId, quantity: 2 });
136    expect(res.statusCode).toBe(201);
137    saleId = res.body.id;
138  });
```

Рисунок 4.2 – Приклад інтеграційного тесту: створення продажу адміністратором

Результати цього тесту засвідчують, що система правильно зберігає інформацію про транзакцію, оновлює залишки товару та враховує права доступу користувача.

4.1.2 Перевірка обробки помилок і виняткових ситуацій

Окрема серія тестів була спрямована на перевірку поведінки API у випадках порушення бізнес-логіки або некоректного вводу. Було змодельовано ситуації, коли:

- відбувається спроба продажу більшої кількості товару, ніж є в наявності;
- запити містять неповні або неправильні дані;
- відсутній або недійсний токен авторизації;
- користувач звертається до ресурсів, до яких не має доступу.

У всіх випадках сервер повертав передбачувані статуси 400, 401 або 403 із поясненнями, що свідчить про якісну реалізацію валідації та обробки виключень. Перевірялися також обмеження доступу згідно з ролями: звичайний користувач чи касир не могли створити магазини або товари. На рисунку 4.3 показано результат повного проходження всіх тестів, запущених у середовищі NODE_ENV=test за допомогою команди npx jest.

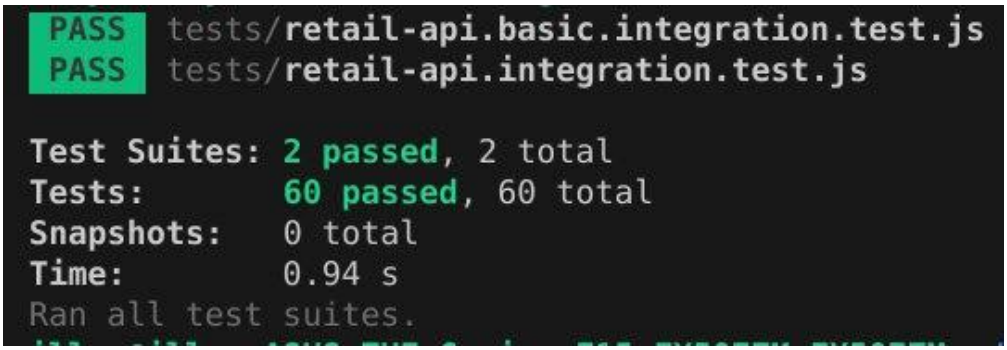


Рисунок 4.3 – Результати проходження всіх тестів

Усього реалізовано 60 тестових випадків, і всі вони успішно пройдені без жодної помилки. Це підтверджує стабільність API, правильність реалізації авторизаційної логіки та стійкість системи до типових і виняткових сценаріїв. Усі тести були реалізовані за допомогою Jest – фреймворку для написання та запуску модульних і інтеграційних тестів у середовищі Node.js [23].

4.2 Огляд функціоналу веб-застосунку

4.2.1 Реєстрація і вхід користувачів

Однією з ключових функцій веб-застосунку є можливість реєстрації та автентифікації користувачів через інтерактивну форму. Сторінка реєстрації та входу реалізована як єдиний компонент, що перемикається між двома режимами – «Реєстрація» і «Вхід». Такий підхід забезпечує зручний користувацький досвід та не потребує перенаправлення між окремими сторінками.

У режимі реєстрації форма стає розширеною та включає кілька обов'язкових полів. Користувач має вказати адресу електронної пошти, яка має бути унікальною, і пароль, що повинен відповідати критеріям безпеки: щонайменше вісім символів, одна велика і одна мала літера, а також хоча б одна цифра. Якщо пароль не відповідає цим вимогам, форма одразу виводить підказку з поясненням. Також необхідно заповнити поля з ім'ям і прізвищем користувача.

Окремо обирається роль у системі – «Користувач», «Касир» або «Адміністратор». У випадку вибору ролі «Касир» додатково з'являється поле для вибору магазину.

Після введення даних і натискання кнопки «Зареєструватись» надсилається POST-запит до /register. Якщо реєстрація успішна, на екрані з'являється повідомлення про успішне створення акаунту (через компонент Snackbar), після чого форма автоматично переключається на режим входу. У випадку помилок (наприклад, коли email вже існує) користувач отримує відповідне повідомлення.

Таким чином, механізм реєстрації дає змогу не лише створити новий обліковий запис, але й одразу задати рівень доступу в системі. Наприклад, адміністратор отримує доступ до всіх функцій, касир – лише до пов'язаного

магазину, а звичайний користувач – до перегляду даних. На рисунку 4.4 показано приклад заповнення форми реєстрації з вибраною роллю «Касир».

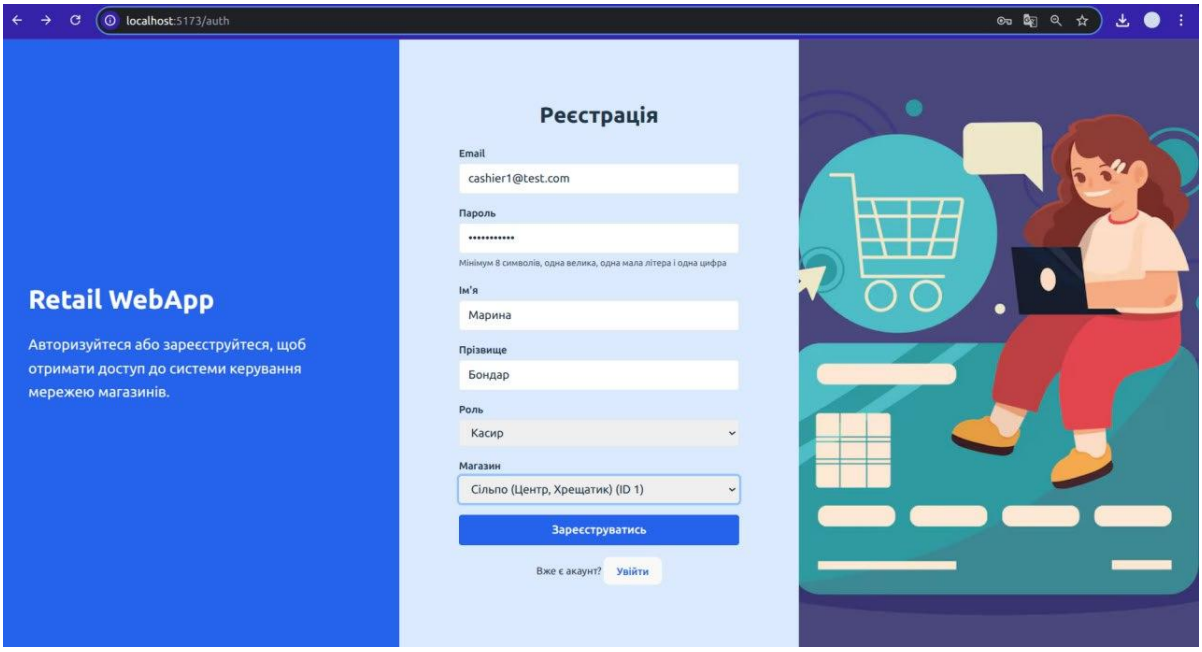


Рисунок 4.4 – Інтерфейс сторінки реєстрації нового користувача

Режим входу містить лише два обов’язкових поля: електронну адресу, яка була використана під час реєстрації, та пароль. Після натискання кнопки «Увійти» надсилається запит до маршруту /login. Якщо облікові дані правильні, сервер повертає токен доступу у форматі JWT, який зберігається в localStorage. Це дає змогу зберегти авторизацію між сесіями та надсилати подальші запити без повторного входу.

У разі помилок (наприклад, неправильний пароль або відсутній акаунт), користувач бачить повідомлення про помилку над формою. Після успішного входу користувач автоматично перенаправляється до сторінки свого профілю, яка вже відображає функціонал відповідно до його ролі. На рисунку 4.5 зображено інтерфейс форми входу з прикладом введених даних.

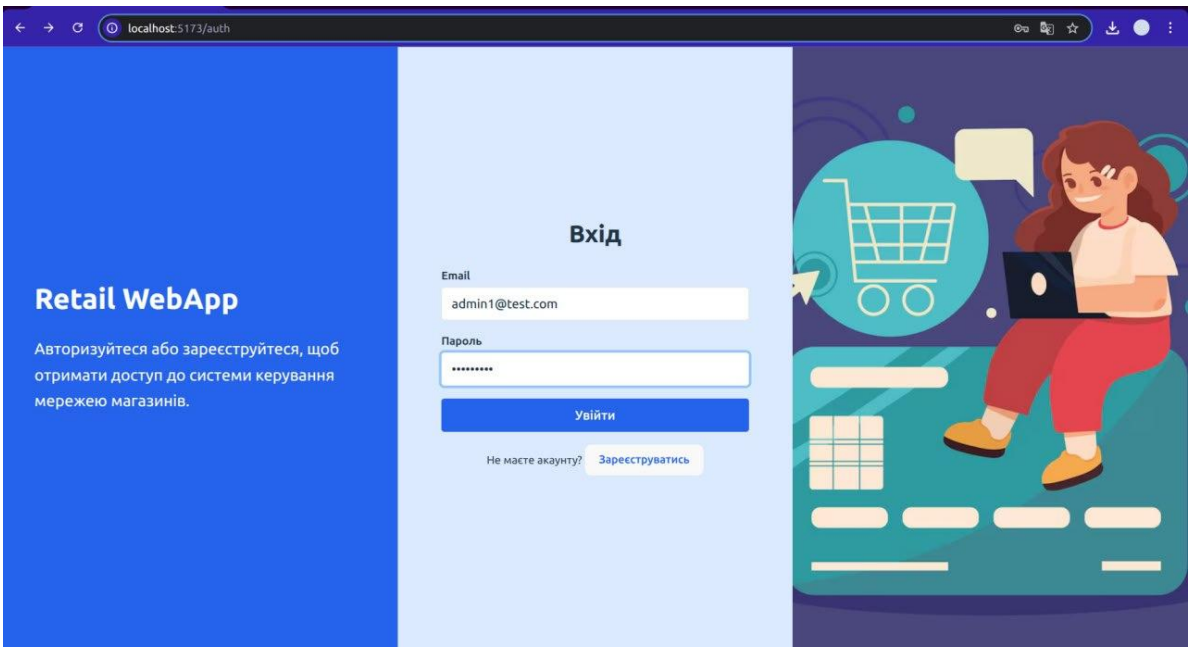


Рисунок 4.5 – Інтерфейс сторінки входу користувача

Цей механізм автентифікації дає змогу забезпечити персоналізований доступ до функціоналу системи відповідно до ролі користувача. Завдяки перевірці даних як на клієнтському, так і на серверному боці, вдалося досягти надійного захисту від некоректного введення, а також запобігти несанкціонованому доступу. Таким чином, сторінка реєстрації та входу не лише виконує базові функції створення та автентифікації облікових записів, а й є важливою складовою загальної системи безпеки та управління доступом у веб-застосунку.

4.2.2 Профіль користувача

Після успішного входу до системи користувач потрапляє на сторінку профілю, яка є основною панеллю керування та взаємодії з даними. Її інтерфейс автоматично адаптується відповідно до ролі – адміністратора, касира або звичайного користувача – забезпечуючи лише дозволені операції.

У верхній частині сторінки відображається персоналізоване вітання, що містить ім’я користувача, адресу електронної пошти, роль у системі, а також

інтерактивну аватарку з можливістю її зміни. Для користувачів також доступний список магазинів із можливістю вибору, що дає змогу перемикати активний контекст. Це зручно для фільтрації даних за конкретним магазином і наведено на рисунку 4.6.

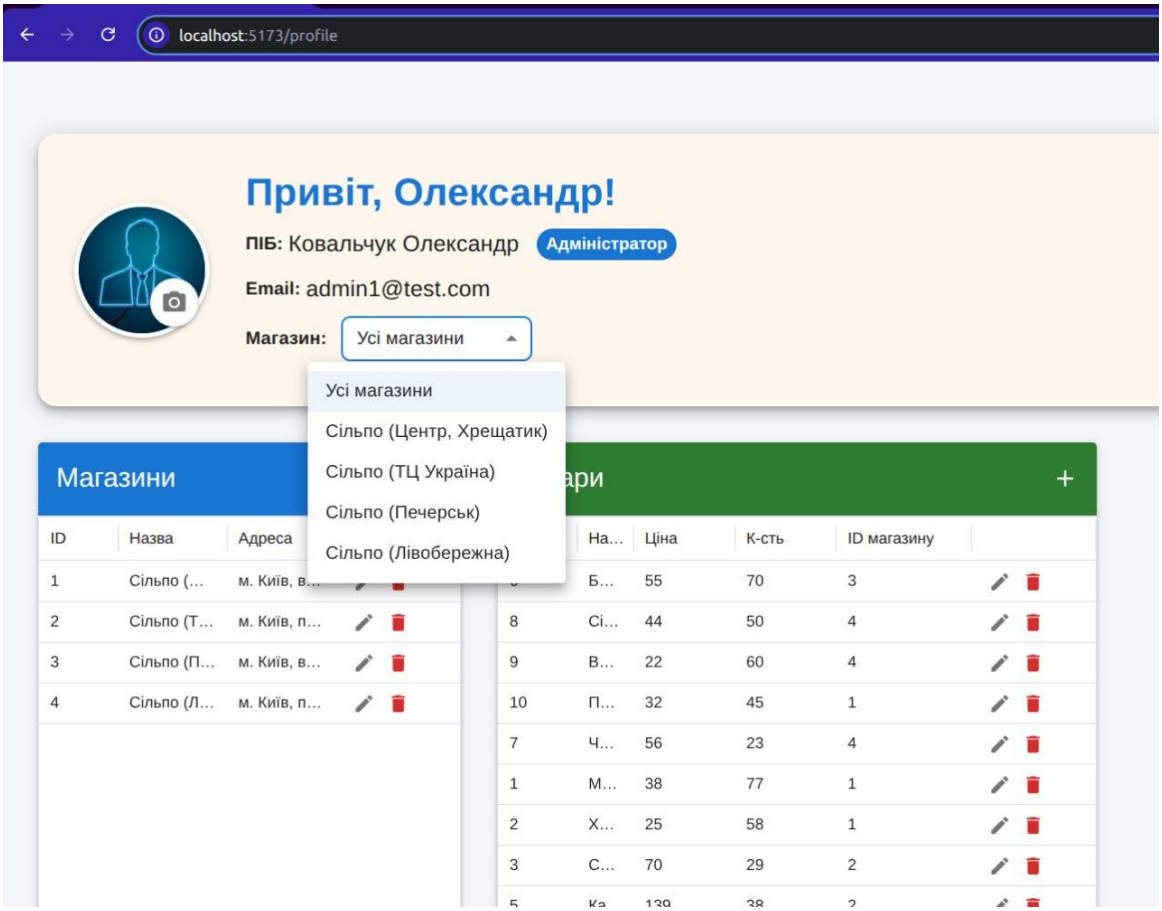


Рисунок 4.6 – Список магазинів з можливістю вибору

Нижче розміщені картки з таблицями для керування інформацією про магазини, товари, продажі та працівників. Залежно від ролі, користувач бачить лише ті компоненти, до яких має дозвіл. Таблиці підтримують функціонал сортування (при натисканні на заголовки стовпців), пагінацію для зручного перегляду великих обсягів даних, а також автоматичну адаптацію до ширини екрана. У кожній таблиці реалізовано можливість виконання дій над рядками – редагування або видалення – через інтуїтивно зрозумілі кнопки.

Адміністратор отримує повний доступ до всіх таблиць. Він може додавати, редагувати та видаляти магазини, товари, керувати продажами та створювати працівників. На рисунку 4.7 зображено таблицю магазинів, яка дає змогу адміністраторам виконувати CRUD-операції. Інтерфейс включає кнопки редагування та видалення для кожного запису, а також кнопку створення нового магазину.

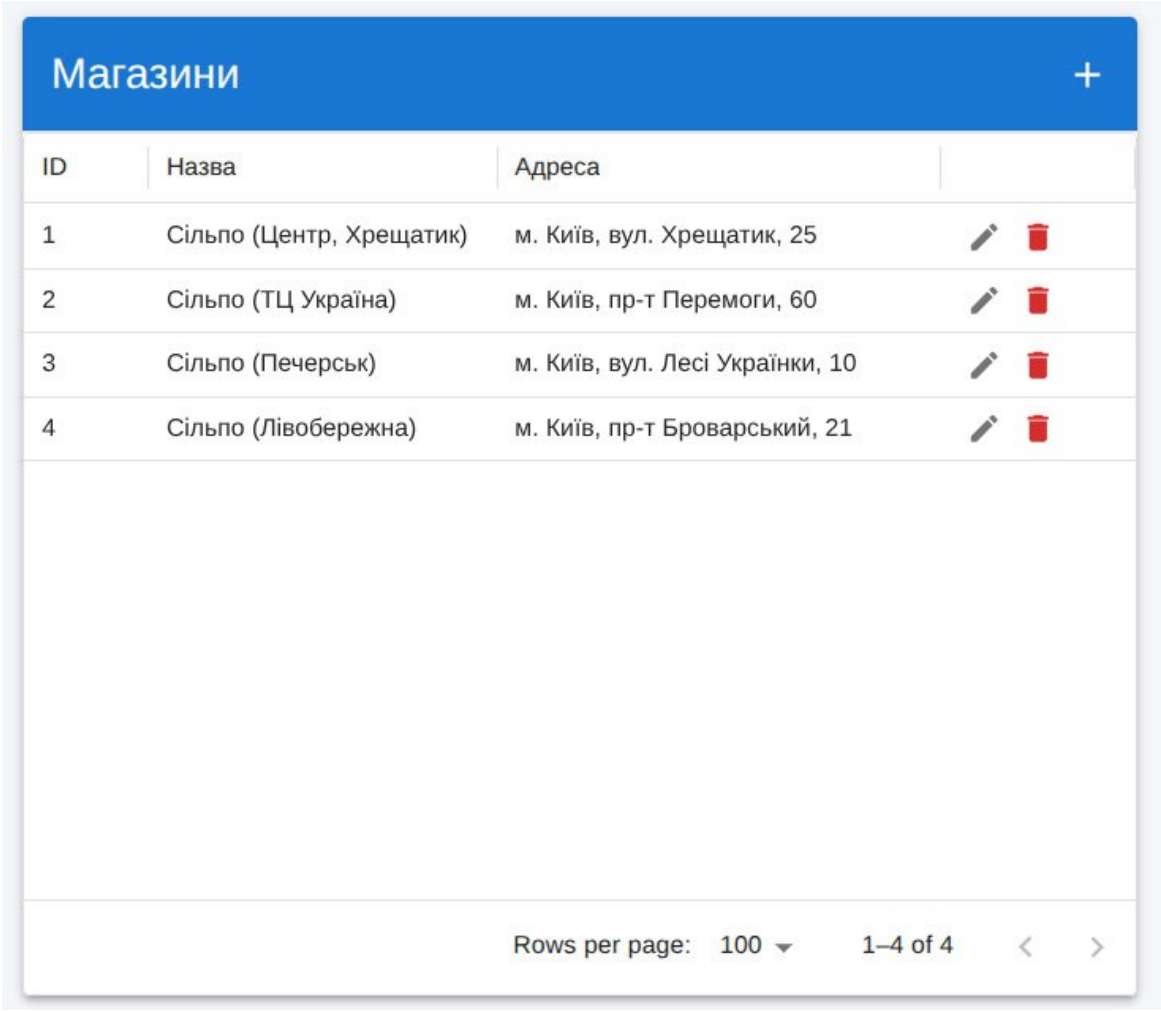


Рисунок 4.7 – Інтерфейс таблиці магазинів у режимі адміністратора

Адміністратор також має змогу здійснювати додавання продажів через спеціальну форму. Вона включає вибір товару із списку, введення кількості, після чого автоматично обчислюється загальна сума. Такий підхід дає змогу швидко формувати нові записи про продажі без потреби ручного підрахунку. На рисунку 4.8 показано діалогове вікно додавання нового продажу.

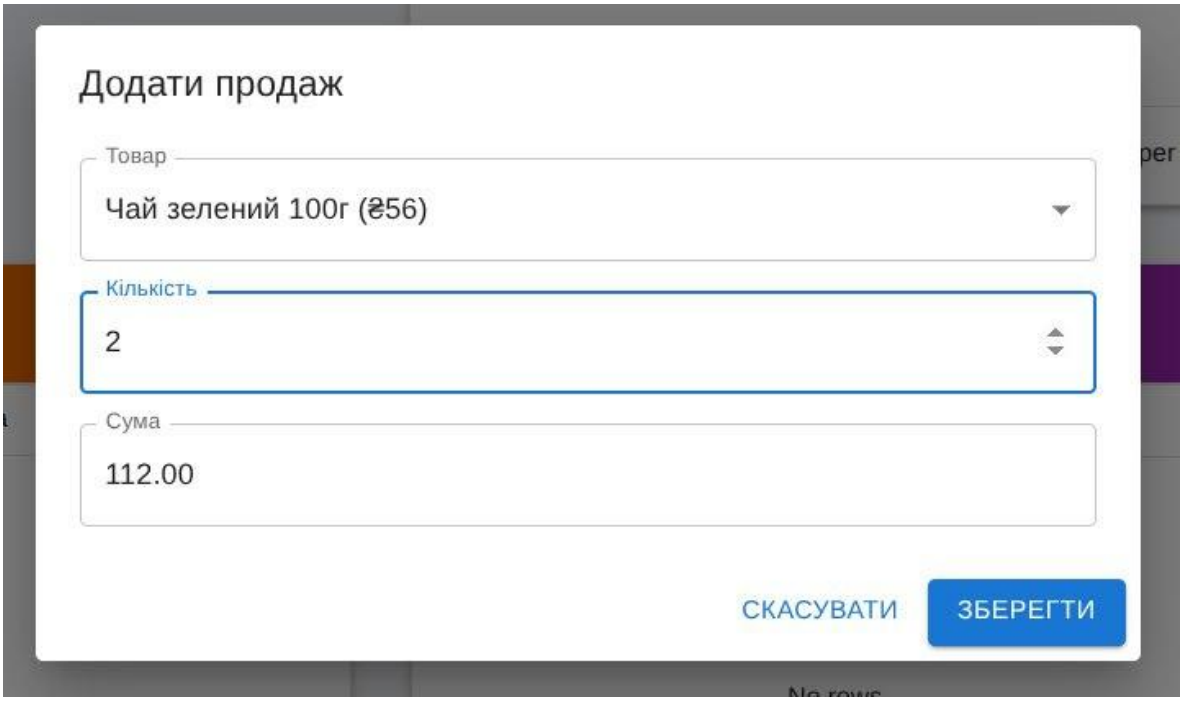


Рисунок 4.8 – Форма додавання нового продажу у режимі адміністратора

На рисунку 4.9 представлена таблиця продажів, де адміністратор має змогу переглядати та редагувати інформацію про кожну транзакцію. Таблиця включає назву товару, його ціну, кількість, залишок на складі та автоматично обчислену суму, а також підсумковий рядок із загальною вартістю.

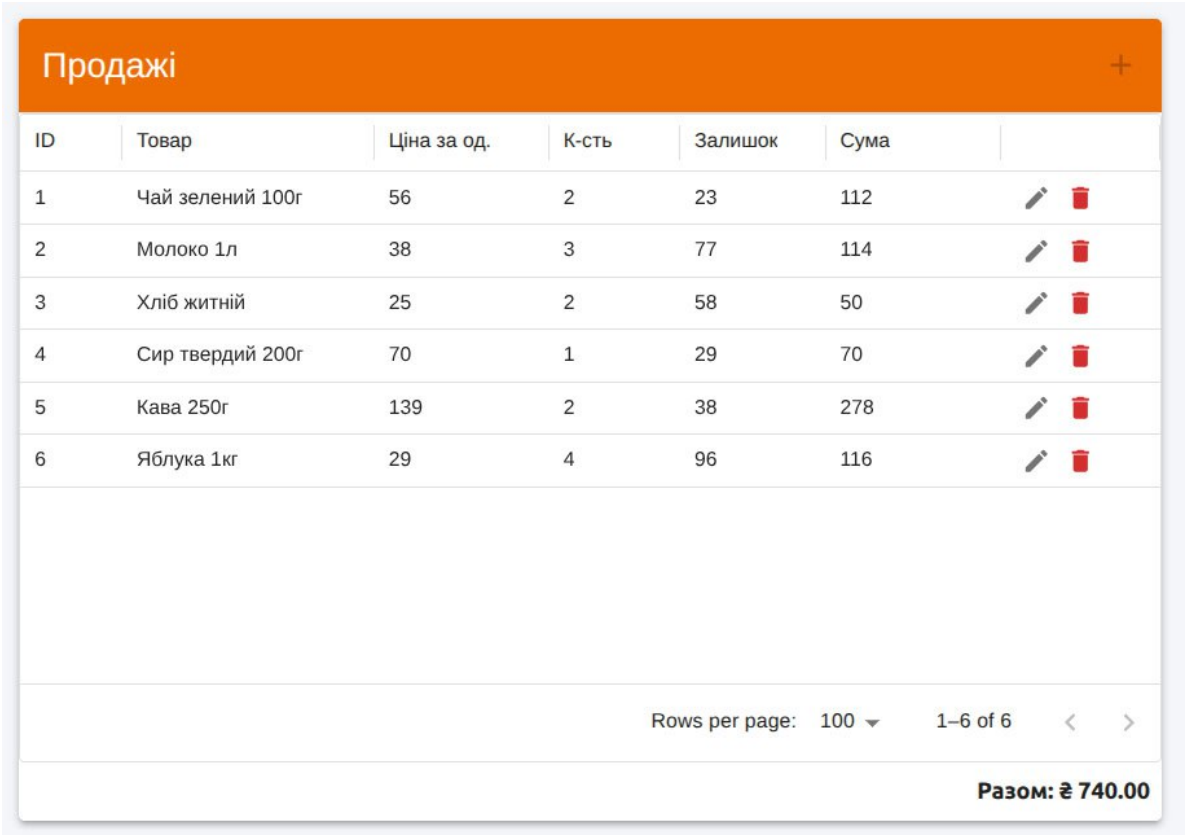


Рисунок 4.9 – Інтерфейс таблиці продажів у режимі адміністратора

Касир має обмежений функціонал: йому доступне лише керування товарами в межах закріпленого за ним магазину. Він бачить таблиці магазинів і товарів, але може редагувати тільки ті товари, що належать до його магазину. На рисунку 4.10 показано приклад інтерфейсу профілю касира, де видно доступні дії та обмеження за роллю.

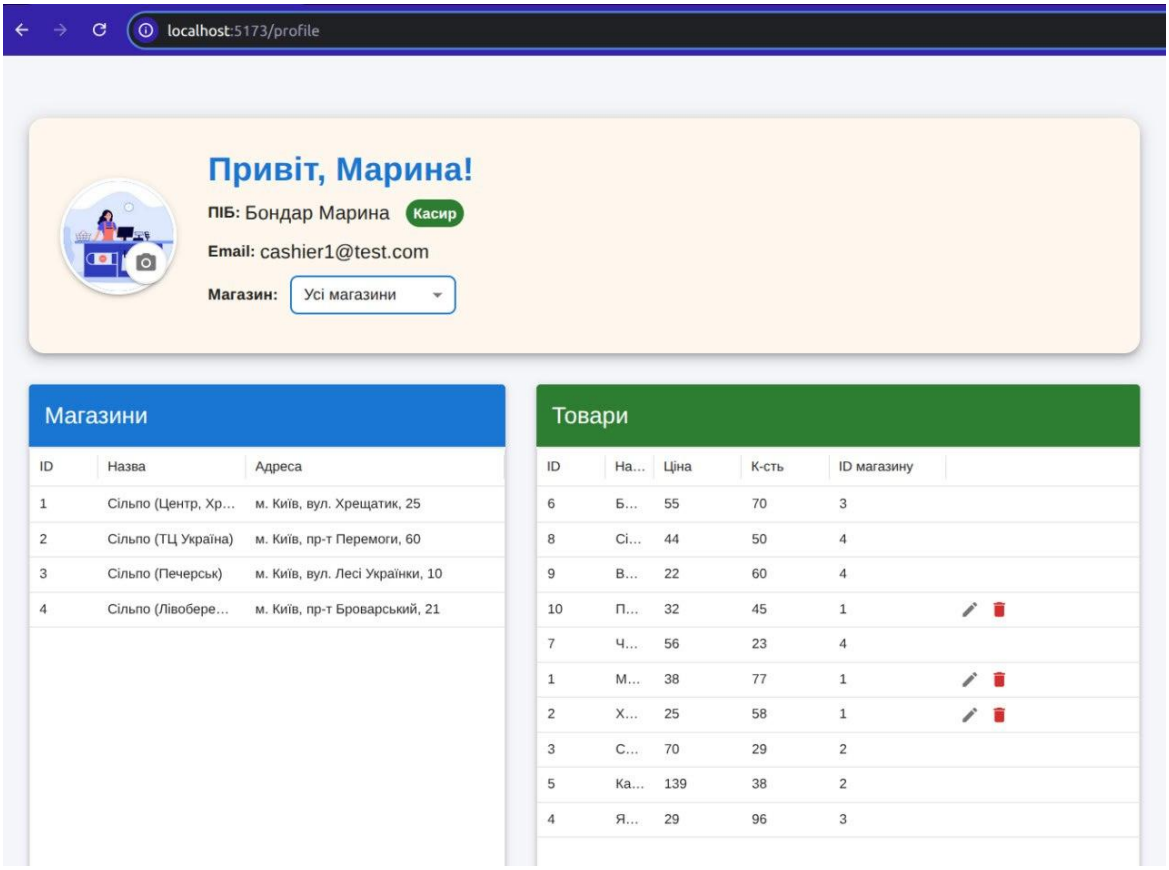


Рисунок 4.10 – Інтерфейс профілю касира

Звичайний користувач має найбільш обмежений функціонал: він може переглядати список магазинів і товари без можливості редагування. Такий рівень доступу забезпечує лише ознайомлення з даними, не даючи змоги впливати на бізнес-логіку застосунку. На рисунку 4.11 наведено вигляд профілю звичайного користувача.

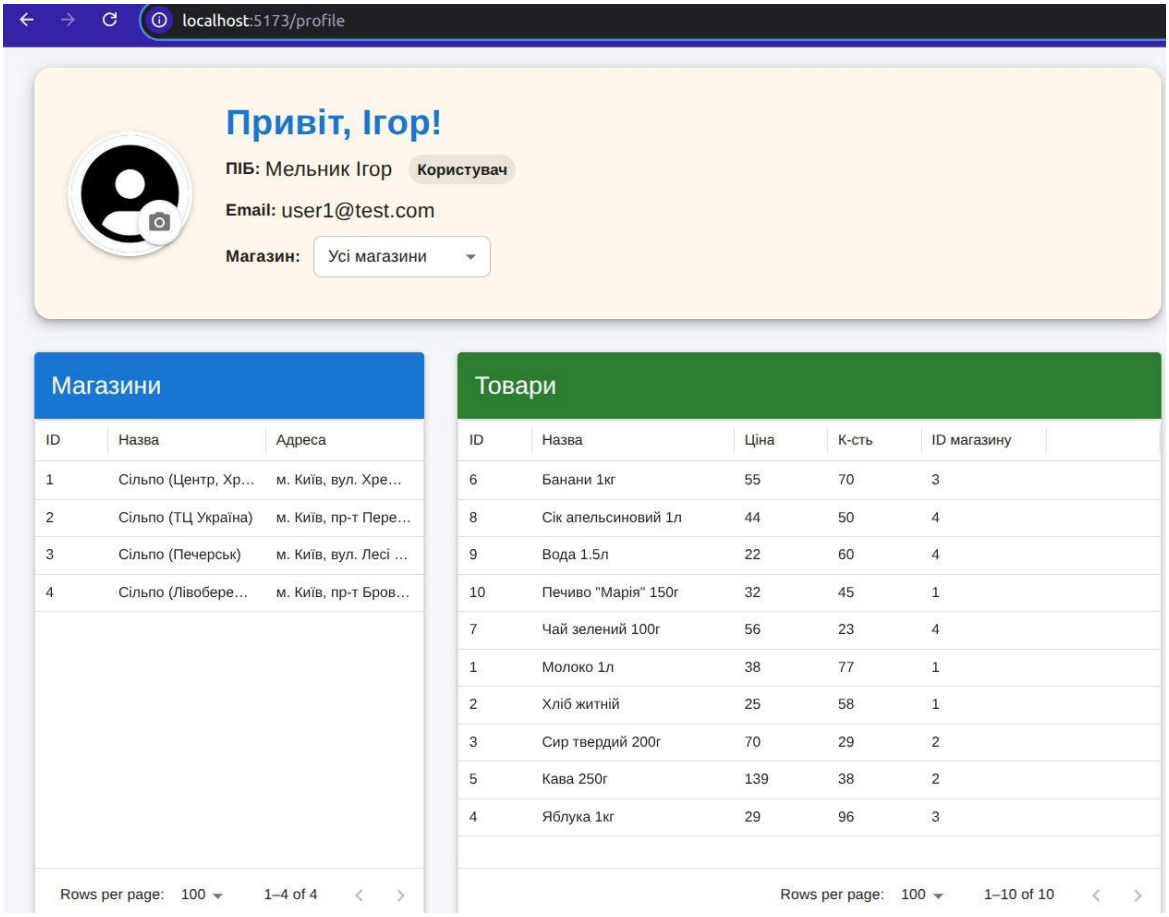


Рисунок 4.11 – Інтерфейс профілю звичайного користувача

Інтерактивна поведінка інтерфейсу, зокрема перемикання магазинів, викликає відповідні API-запити до бекенду, що забезпечує актуальне оновлення вмісту всіх таблиць. Таким чином, сторінка профілю є центральним вузлом взаємодії з даними, а її логіка реалізована з урахуванням ролей, що сприяє зручності, безпеці та ефективності керування мережею магазинів.

ВИСНОВОКИ ДО РОЗДІЛУ 4

У цьому розділі проведено всебічний огляд реалізованого функціоналу веб-застосунку та перевірку його працездатності шляхом інтеграційного тестування. Тестування серверної частини підтвердило правильність реалізації авторизації, обробки помилок і взаємодії між модулями, зокрема у сценаріях, що охоплюють усі ролі користувачів та ключові бізнес-процеси: управління магазинами, товарами, продажами та працівниками.

Завдяки успішно виконаним тестам підтверджено стійкість системи до помилкових запитів, дотримання правил доступу та стабільну роботу API в типових і виняткових ситуаціях. Тестові сценарії також показали, що система правильно обробляє як валідні, так і некоректні дані, забезпечуючи захист від несанкціонованих дій.

Крім того, детально продемонстровано інтерфейс користувача з урахуванням ролей. Інтерфейс є адаптивним, зручним і функціональним: від сторінки автентифікації до профілю користувача з динамічними таблицями. Адміністратори мають повний доступ до всіх модулів і функцій, касири – обмежений доступ до даних свого магазину, а звичайні користувачі – лише до перегляду інформації. Таблиці реалізовано з підтримкою сортування, пагінації, інтерактивного оновлення, а також дій над рядками.

Таким чином, у результаті тестування та аналізу інтерфейсів можна зробити висновок, що створений веб-застосунок є надійним, зручним у використанні та відповідає функціональним вимогам до систем керування торговельною мережею.

ВИСНОВОКИ

У межах виконання дипломного проєкту розроблено повноцінний веб-застосунок для управління мережею магазинів, що забезпечує автоматизацію ключових бізнес-процесів та сприяє цифровій трансформації торговельної діяльності. У вступі обґрунтовано актуальність теми: зростання конкуренції на ринку роздрібної торгівлі потребує оперативного прийняття рішень, підвищення ефективності управління та скорочення ручної праці шляхом впровадження сучасних інформаційних технологій.

У першому розділі проведено детальний аналіз предметної області, розглянуто існуючі проблеми неавтоматизованих торговельних мереж, роль цифрової трансформації у торгівлі та перспективи використання веб-застосунків. Проведено огляд і порівняльний аналіз популярних рішень (Salesforce, Poster, Torgsoft), виявлено їх сильні й слабкі сторони, що дало змогу обґрунтувати потребу у створенні власного адаптованого рішення для малого та середнього бізнесу в Україні.

У другому розділі обрано оптимальний стек технологій для реалізації системи: серверна частина на базі Node.js та Express, база даних PostgreSQL з ORM Prisma, клієнтська частина з використанням React, Tailwind CSS і Material UI. Також реалізовано REST API із захистом за допомогою JWT, що забезпечує безпечну автентифікацію та авторизацію користувачів. Контейнеризація за допомогою Docker та Docker Compose дозволила створити відтворюване середовище для розробки й розгортання.

У третьому розділі описано архітектуру, реалізацію та функціональність застосунку. Впроваджено повний набір CRUD-операцій для магазинів, товарів, працівників і продажів. Передбачено контроль доступу до функціоналу залежно від ролі користувача (адміністратор, касир, звичайний користувач). Застосунок

підтримує динамічне оновлення даних, адаптивний дизайн, а також завантаження та збереження зображень.

Четвертий розділ присвячено тестуванню системи. Інтеграційні тести охопили всі основні функції та підтвердили коректну взаємодію між модулями, дотримання логіки авторизації, обробку помилок і стабільність системи. Інтерфейс перевірено на відповідність ролям користувачів та зручність використання.

У підсумку, реалізований веб-застосунок повністю відповідає поставленій меті – забезпечити ефективне, надійне та масштабоване рішення для управління мережею магазинів. Застосунок має перспективу подальшого розвитку: інтеграції з аналітичними сервісами, розширення на мобільні платформи, додавання підтримки багатомовності та розгортання у хмарному середовищі. Отримані результати свідчать про доцільність використання обраного підходу для впровадження в реальні торговельні мережі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. GitHub - Illyakravchuk/retail-webapp [Електронний ресурс]. – Режим доступу: https://github.com/Illyakravchuk/retail-webapp.git.
2. Statista. Global retail e-commerce sales 2014–2026 [Електронний ресурс]. – Режим доступу: https://www.statista.com/statistics/379046/worldwide-retail-e-commerce-sales/.
3. Ковальчук В. В., Іванов А. П. Цифрова трансформація вітчизняного ритейлу: виклики і перспективи – 2020. – № 70. – С. 128–132.
4. Тарасов А. Н. Інформаційні системи в економіці. – М.: Фінанси і статистика, 2011. – 312 с.
5. Таненбаум Е. С., ван Стін М. Розподілені системи. Принципи та парадигми. – 2-ге вид. – К.: Діалектика, 2010. – 656 с.
6. ISO/IEC 27001:2022. Information technology – Security techniques – Information security management systems – Requirements. – ISO, 2022.
7. Salesforce Commerce Cloud Documentation [Електронний ресурс]. – Режим доступу: https://developer.salesforce.com/docs/commerce.
8. Poster POS. Офіційна документація [Електронний ресурс]. – Режим доступу: https://joinposter.com.
9. Torgsoft. Система автоматизації торгівлі [Електронний ресурс]. – Режим доступу: https://torgsoft.ua/.
10. Node.js Documentation. – [Електронний ресурс]. – Режим доступу: https://nodejs.org/en/docs.
11. Express.js – Guide & API Reference. – [Електронний ресурс]. – Режим доступу: https://expressjs.com.
12. PostgreSQL 15 Documentation. – PostgreSQL Global Development Group, 2023. – [Електронний ресурс]. – Режим доступу: https://www.postgresql.org/docs/15.

13. Prisma ORM Documentation. – [Електронний ресурс]. – Режим доступу: https://www.prisma.io/docs.
14. React Docs. –Meta Platforms, 2023. – [Електронний ресурс]. – Режим доступу: https://react.dev/learn.
15. Tailwind CSS Documentation. – [Електронний ресурс]. – Режим доступу: https://tailwindcss.com/docs.
16. Material UI Documentation. – [Електронний ресурс]. – Режим доступу: https://mui.com/material-ui/.
17. Docker Documentation. – Docker Inc., 2023. – [Електронний ресурс]. – Режим доступу: https://docs.docker.com.
18. Docker Compose Specification – https://compose-spec.io.
19. Jones M., Bradley J. RFC 7519: JSON Web Token (JWT). – IETF, 2015.
20. Axios Documentation. – [Електронний ресурс]. – Режим доступу: https://axios-http.com.
21. Multer Middleware for handling multipart/form-data. – [Електронний ресурс]. – Режим доступу: https://github.com/expressjs/multer.
22. SuperTest – HTTP assertions for Node.js made easy. – [Електронний ресурс]. – Режим доступу: https://github.com/visionmedia/supertest.
23. Jest – Delightful JavaScript Testing. – [Електронний ресурс]. – Режим доступу: https://jestjs.io/.

ДОДАТОК 1

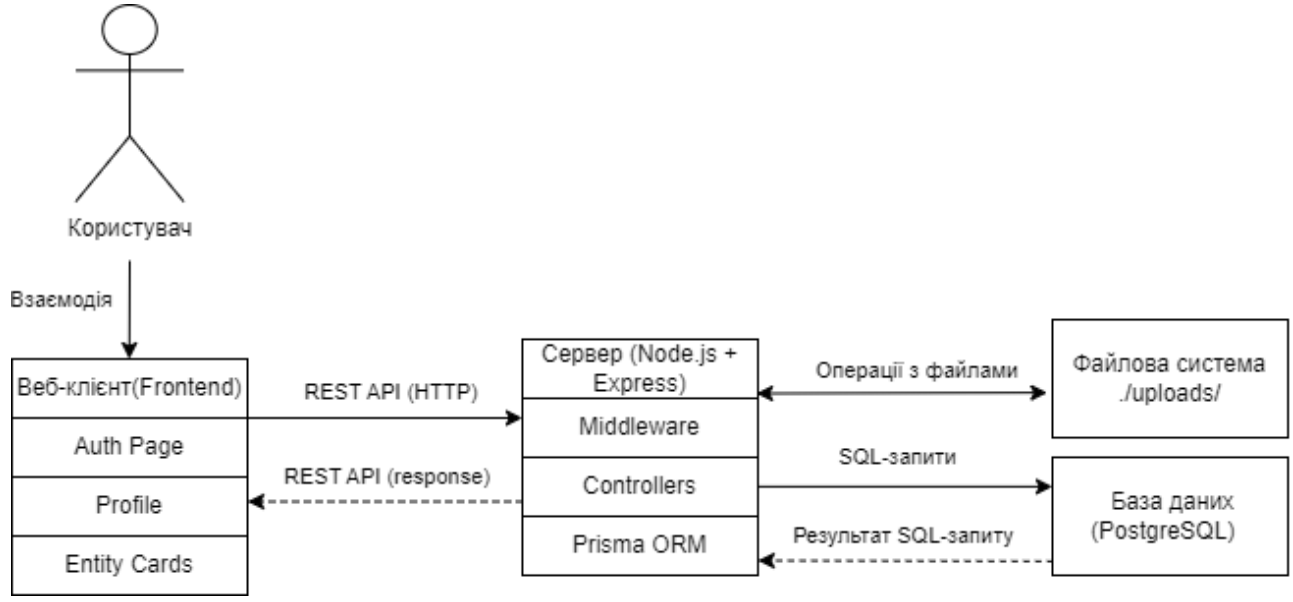
Веб-застосунок для управління діяльністю мережі

Структурна схема системи

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2025



					ІАЛЦ.467200.004 Д1		
		№ докум.	Підпис	Дата			
Розробив		Кравчук І. В.			**Веб-застосунок для управління діяльністю мережі магазинів** Структурна схема системи	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Шульга М. В.					
Н. Контр.		Пономаренко А.М.				НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-13	
Затвердив							

ДОДАТОК 2

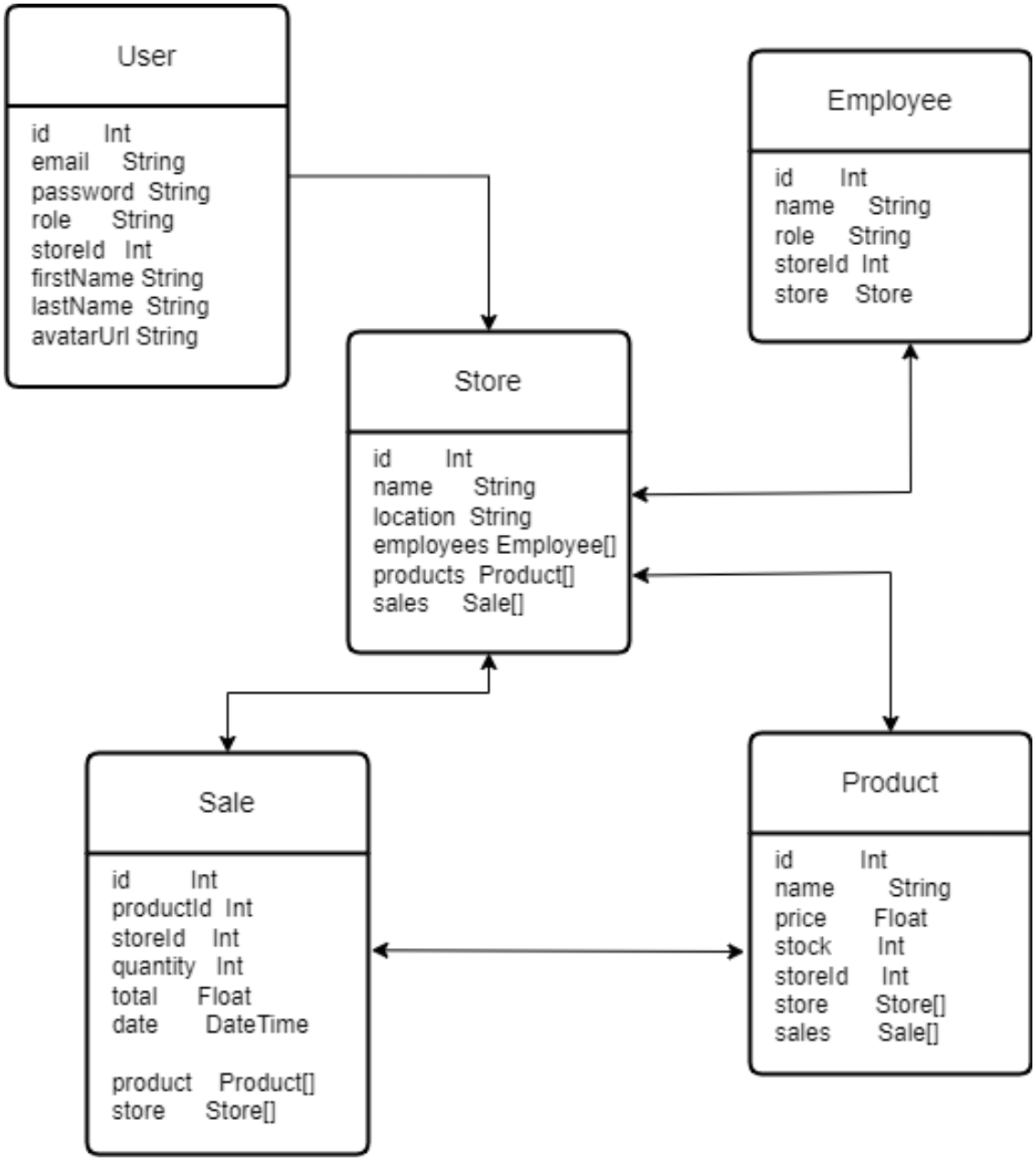
Веб-застосунок для управління діяльністю мережі

Діаграма бази даних (функціональна схема)

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2025



					ІАЛЦ.467200.005 Д2	
		№ докум.	Підпис	Дата		
Розробив		Кравчук І. В.			**Веб-застосунок для управління діяльністю мережі магазинів** Діаграма бази даних (функціональна схема)	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Шульга М. В.				НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-13
Н. Контр.		Пономаренко А.М.				
Затвердив						

ДОДАТОК 3

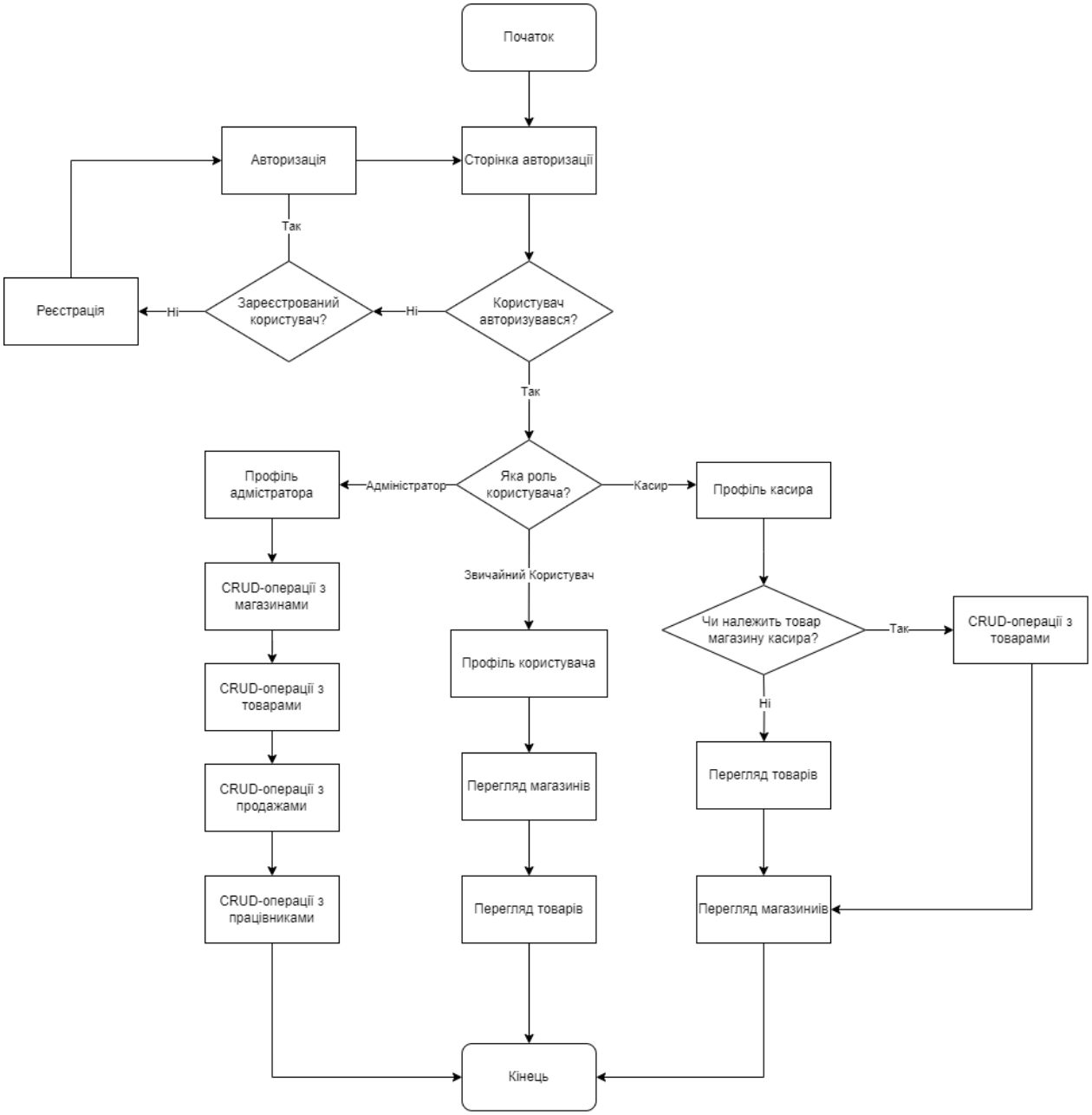
Веб-застосунок для управління діяльністю мережі

Алгоритм дій користувача в системі
(принципова схема)

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2025



					ІАЛЦ.467200.006 Д3	
		№ докум.	Підпис	Дата		
Розробив		Кравчук І. В.			**Веб-застосунок для управління діяльністю мережі магазинів** Алгоритм дій користувача в системі (принципова схема)	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Шульга М. В.				НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-13
Н. Контр.		Пономаренко А.М.				
Затвердив						

ДОДАТОК 4

Веб-застосунок для управління діяльністю мережі

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 36

Київ 2025

```
import 'dotenv/config';
import express from 'express';
import cors from 'cors';
import { PrismaClient } from '@prisma/client';
import bcrypt from 'bcrypt';
import { generateToken } from './utils/jwt.js';
import { authenticate } from './middleware/auth.js';
import { authorizeRoles } from './middleware/roles.js';
import multer from 'multer';
import path from 'path';
import fs from 'fs';

const app = express();
const prisma = new PrismaClient();
const PORT = process.env.PORT || 3000;

// --- CORS та JSON ---
app.use(cors({
  origin: 'http://localhost:5173',
  credentials: true,
}));
app.use(express.json());

// --- Головна сторінка API ---
app.get('/', (_req, res) => {
  res.json({ status: 'ok', message: 'Retail Web API is up!' });
});

// --- Допоміжна утиліта для фільтрації по магазину ---
function byStore(q) {
  return q.storeId ? { storeId: Number(q.storeId) } : {};
}

// --- Магазини ---
app.get('/stores-lite', async (_req, res) => {
  try {
    const stores = await prisma.store.findMany({
      select: { id: true, name: true }
    });
    res.json(stores);
  } catch {
    res.status(500).json({ error: 'Failed to fetch stores' });
  }
});
```

					ІАЛЦ.467200.007 Д4			
		№ докум.	Підпис	Дата				
Розробив		Кравчук І. В.			**Веб-застосунок для управління діяльністю мережі магазинів** Текст програмного коду	Літ.	Аркуш	Аркушів
Перевірив		Шульга М. В.					1	36
						НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-13		
Н. Контр.		Пономаренко А.М.						
Затвердив								

```
app.get('/stores', authenticate, async (_req, res) => {
  try {
    const isAdmin = _req.user.role === 'admin';
    const stores  = await prisma.store.findMany(
      isAdmin
        ? { include:{ employees:true, products:true, sales:true } }
        : { select :{ id:true, name:true } }
    );
    res.json(stores);
  } catch (err) {
    res.status(500).json({ error: 'Failed to fetch stores' });
  }
});

app.get('/stores/:id', authenticate, authorizeRoles(['admin']), async (req, res) => {
  try {
    const store = await prisma.store.findUnique({
      where: { id: Number(req.params.id) },
      include: { employees: true, products: true, sales: true }
    });
    if (!store) return res.status(404).json({ error: 'Store not found' });
    res.json(store);
  } catch (err) {
    res.status(500).json({ error: 'Failed to fetch store' });
  }
});

app.post('/stores', authenticate, authorizeRoles(['admin']), async (req, res) => {
  try {
    const store = await prisma.store.create({ data: req.body });
    res.status(201).json(store);
  } catch (err) {
    res.status(400).json({ error: 'Failed to create store' });
  }
});

app.put('/stores/:id', authenticate, authorizeRoles(['admin']), async (req, res) => {
  try {
    const updated = await prisma.store.update({
      where: { id: Number(req.params.id) },
      data: req.body
    });
    res.json(updated);
  } catch (err) {
    res.status(400).json({ error: 'Failed to update store' });
  }
});

app.delete('/stores/:id', authenticate, authorizeRoles(['admin']), async (req, res) => {
  try {
    await prisma.store.delete({ where: { id: Number(req.params.id) } });
```

```
 res.json({ message: 'Store deleted' });
  } catch (err) {
   res.status(400).json({ error: 'Failed to delete store' });
  }
});

// --- Працівники ---
app.get('/employees', authenticate, authorizeRoles(['admin']), async (_req, res) => {
  try {
   const list = await prisma.employee.findMany({ where: byStore(_req.query) });
   res.json(list);
  } catch {
   res.status(500).json({ error: 'Failed to fetch employees' });
  }
});

app.get('/employees/:id', authenticate, authorizeRoles(['admin']), async (req, res) => {
  try {
   const emp = await prisma.employee.findUnique({ where: { id: Number(req.params.id) }});
   if (!emp) return res.status(404).json({ error: 'Employee not found' });
   res.json(emp);
  } catch {
   res.status(500).json({ error: 'Failed to fetch employee' });
  }
});

app.post('/employees', authenticate, authorizeRoles(['admin']), async (req, res) => {
  try {
   const emp = await prisma.employee.create({ data: req.body });
   res.status(201).json(emp);
  } catch {
   res.status(400).json({ error: 'Failed to create employee' });
  }
});

app.put('/employees/:id', authenticate, authorizeRoles(['admin']), async (req, res) => {
  try {
   const emp = await prisma.employee.update({
    where: { id: Number(req.params.id) },
    data:  req.body
   });
   res.json(emp);
  } catch {
   res.status(400).json({ error: 'Failed to update employee' });
  }
});

app.delete('/employees/:id', authenticate, authorizeRoles(['admin']), async (req, res) => {
  try {
   await prisma.employee.delete({ where: { id: Number(req.params.id) }});
   res.json({ message: 'Employee deleted' });
  } catch {
```

```
  res.status(400).json({ error: 'Failed to delete employee' });
  }
});

// --- Продукти ---
app.get('/products', authenticate, async (_req, res) => {
  try {
    const list = await prisma.product.findMany({ where: byStore(_req.query) });
    res.json(list);
  } catch {
    res.status(500).json({ error: 'Failed to fetch products' });
  }
});

app.get('/products/:id', authenticate, async (req, res) => {
  try {
    const prod = await prisma.product.findUnique({ where: { id: Number(req.params.id) }});
    if (!prod) return res.status(404).json({ error: 'Product not found' });
    res.json(prod);
  } catch {
    res.status(500).json({ error: 'Failed to fetch product' });
  }
});

app.post('/products', authenticate, authorizeRoles(['admin', 'cashier']), async (req, res) => {
  try {
    if (req.user.role === 'cashier') {
      if (+req.body.storeId !== +req.user.storeId)
        return res.status(403).json({ error: 'Cashier – only own store' });
    }
    const prod = await prisma.product.create({ data:req.body });
    res.status(201).json(prod);
  } catch {
    res.status(400).json({ error: 'Failed to create product' });
  }
});

app.put('/products/:id', authenticate, authorizeRoles(['admin', 'cashier']), async (req, res) => {
  try {
    if (req.user.role === 'cashier') {
      const prod = await prisma.product.findUnique({ where:{ id:+req.params.id } });
      if (!prod || +prod.storeId !== +req.user.storeId)
        return res.status(403).json({ error:'Cashier – only own store' });
    }
    const updated = await prisma.product.update({
      where:{ id:Number(req.params.id) }, data:req.body
    });
    res.json(updated);
  } catch {
    res.status(400).json({ error: 'Failed to update product' });
  }
});
```

```
app.delete('/products/:id', authenticate, authorizeRoles(['admin', 'cashier']), async (req, res) => {
  try {
    if (req.user.role === 'cashier') {
      const prod = await prisma.product.findUnique({ where:{ id:+req.params.id } });
      if (!prod || +prod.storeId !== +req.user.storeId)
        return res.status(403).json({ error:'Cashier – only own store' });
    }
    await prisma.product.delete({ where:{ id:Number(req.params.id) } });
    res.json({ message:'Product deleted' });
  } catch {
    res.status(400).json({ error: 'Failed to delete product' });
  }
});

// --- Продажі ---
app.get('/sales', authenticate, authorizeRoles(['admin']), async (req, res) => {
  try {
    const list = await prisma.sale.findMany({
      where: byStore(req.query),
      include:{ product:true, store:true }
    });
    res.json(list);
  } catch {
    res.status(500).json({ error: 'Failed to fetch sales' });
  }
});

// Агрегована сума по магазину
app.get('/sales/summary', authenticate, authorizeRoles(['admin']), async (req,res)=>{
  try{
    const sum = await prisma.sale.aggregate({
      where: byStore(req.query),
      _sum : { total:true }
    });
    res.json({ total: sum._sum.total ?? 0 });
  }catch{ res.status(500).json({ error:'Failed to fetch summary' }); }
});

// Один продаж
app.get('/sales/:id', authenticate, authorizeRoles(['admin']), async (req,res)=>{
  const sale = await prisma.sale.findUnique({
    where:{ id:+req.params.id },
    include:{ product:true, store:true }
  });
  sale ? res.json(sale)
       : res.status(404).json({ error:'Sale not found' });
});

// Додати продаж
app.post('/sales', authenticate, authorizeRoles(['admin']), async (req,res)=>{
  const { productId, quantity } = req.body;
  console.log("[POST /sales] Body:", req.body);
```

```
  if (!productId || !quantity) {
    console.log("[POST /sales] Missing productId or quantity");
    return res.status(400).json({ error:'productId & quantity required' });
  }

  const product = await prisma.product.findUnique({ where:{ id:+productId } });
  console.log("[POST /sales] Found product:", product);

  if(!product)
    return res.status(400).json({ error:'product not found' });

  if(product.stock < quantity) {
    console.log(`[POST /sales] Not enough stock: have ${product.stock}, need ${quantity}`);
    return res.status(400).json({ error:'not enough stock' });
  }

  const sale = await prisma.sale.create({
    data:{
      productId : product.id,
      storeId   : product.storeId,
      quantity  : +quantity,
      total     : product.price * quantity
    },
    include:{ product:true, store:true }
  });

  await prisma.product.update({
    where:{ id:product.id },
    data :{ stock:{ decrement: quantity } }
  });

  res.status(201).json(sale);
});

// Оновити продаж
app.put('/sales/:id', authenticate, authorizeRoles(['admin']), async (req,res)=>{
  const { quantity } = req.body;
  if(!quantity) return res.status(400).json({ error:'quantity required' });

  const old = await prisma.sale.findUnique({ where:{ id:+req.params.id } , include:{ product:true }});
  if(!old) return res.status(404).json({ error:'Sale not found' });

  const diff = quantity - old.quantity;
  if(old.product.stock < diff)
     return res.status(400).json({ error:'not enough stock' });

  await prisma.product.update({
    where:{ id:old.productId },
    data :{ stock:{ decrement: diff } }
  });

  const updated = await prisma.sale.update({
    where:{ id:old.id },
```

```
  data :{ quantity:+quantity, total: old.product.price * quantity },
   include:{ product:true, store:true }
  });

  res.json(updated);
});

app.delete('/sales/:id', authenticate, authorizeRoles(['admin']), async (req,res)=>{
  const sale = await prisma.sale.findUnique({ where:{ id:+req.params.id }});
  if(!sale) return res.status(404).json({ error:'Sale not found' });

  await prisma.product.update({
   where:{ id:sale.productId },
   data :{ stock:{ increment: sale.quantity } }
  });

  await prisma.sale.delete({ where:{ id:sale.id }});
  res.json({ message:'Sale deleted' });
});

// --- Аутентифікація ---
app.post('/register', async (req, res) => {
  const {
   email,
   password,
   role = 'user',
   storeId,
   firstName,
   lastName,
  } = req.body;

  if (!email || !password || !firstName || !lastName)
   return res.status(400).json({ error: 'email, password, firstName та lastName обов’язкові' });

  if (role === 'cashier') {
   if (!storeId)
    return res.status(400).json({ error: 'storeId is required for cashier' });

   const storeExists = await prisma.store.findUnique({
    where: { id: Number(storeId) },
    select: { id: true }
   });
   if (!storeExists)
    return res.status(400).json({ error: 'storeId not found' });
  }

  if (role !== 'cashier' && storeId)
   return res.status(400).json({ error: 'storeId allowed only for cashier' });

  const hashed = await bcrypt.hash(password, 10);

  const data = { email, password: hashed, role, firstName, lastName };
```

```
 if (role === 'cashier') data.storeId = Number(storeId);

  try {
    const createdUser = await prisma.user.create({ data });

    // Якщо роль касир - одразу додаємо у employee
    if (role === 'cashier') {
      await prisma.employee.create({
        data: {
          name: `${lastName} ${firstName}`,
          role: 'cashier',
          storeId: Number(storeId),
        }
      });
    }

    res.status(201).json({ message: 'User created' });
  } catch (err) {
    res.status(400).json({ error: 'Email already exists' });
  }
});

app.post('/login', async (req, res) => {
  const { email, password } = req.body;
  try {
    const user = await prisma.user.findUnique({ where: { email } });
    if (!user || !(await bcrypt.compare(password, user.password)))
      return res.status(401).json({ error: 'Invalid credentials' });

    res.json({ token: generateToken(user) });
  } catch {
    res.status(500).json({ error: 'Login failed' });
  }
});

app.get('/profile', authenticate, async (req, res) => {
  const userFromDb = await prisma.user.findUnique({
    where: { id: req.user.id },
    select: {
      id: true,
      email: true,
      role: true,
      storeId: true,
      firstName: true,
      lastName: true,
      avatarUrl: true
    }
  });
  res.json({ user: userFromDb });
});

// --- Завантаження та віддача аватарки ---
```

```
const uploadDir = './uploads/';
if (!fs.existsSync(uploadDir)) fs.mkdirSync(uploadDir);

const storage = multer.diskStorage({
  destination: (req, file, cb) => cb(null, uploadDir),
  filename: (req, file, cb) => {
    const ext = path.extname(file.originalname);
    cb(null, `avatar-${req.user.id}${ext}`);
  }
});
const upload = multer({ storage });

app.post('/profile/avatar', authenticate, upload.single('avatar'), async (req, res) => {
  if (!req.file) return res.status(400).json({ error: 'No file uploaded' });

  const avatarUrl = `/uploads/${req.file.filename}`;
  await prisma.user.update({
    where: { id: req.user.id },
    data: { avatarUrl }
  });
  res.json({ avatarUrl });
});

app.use('/uploads', express.static(uploadDir));

// --- Запуск сервера ---
app.listen(PORT, () => console.log(`API listening on ${PORT}`));

import jwt from 'jsonwebtoken';

const JWT_SECRET = process.env.JWT_SECRET || 'supersecret';
const JWT_EXPIRES = '7d';

export function generateToken(user) {
  return jwt.sign(
    { id: user.id, email: user.email, role: user.role, storeId:user.storeId ?? null},
    JWT_SECRET,
    { expiresIn: JWT_EXPIRES }
  );
}

export function verifyToken(token) {
  return jwt.verify(token, JWT_SECRET);
}

import { verifyToken } from '../utils/jwt.js';

export function authenticate(req, res, next) {
  const header = req.headers.authorization;
  if (!header?.startsWith('Bearer ')) {
    return res.status(401).json({ error: 'No token provided' });
  }
```

```
  try {
    const payload = verifyToken(header.slice(7));
    req.user = payload;
    next();
  } catch {
    res.status(401).json({ error: 'Invalid token' });
  }
}

export const authorizeRoles = (allowedRoles = []) => {
    return (req, res, next) => {
      if (!req.user || !allowedRoles.includes(req.user.role)) {
        return res.status(403).json({ error: 'Access denied' });
      }
      next();
    };
  };

generator client {
  provider      = "prisma-client-js"
  binaryTargets = ["native", "linux-musl-openssl-3.0.x"]
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model User {
  id        Int      @id @default(autoincrement())
  email     String   @unique
  password  String
  role      String   @default("user")
  storeId   Int?
  firstName String?
  lastName  String?
  avatarUrl String?
}

model Store {
  id        Int      @id @default(autoincrement())
  name      String
  location  String
  employees Employee[]
  products  Product[]
  sales     Sale[]
}

model Employee {
  id       Int    @id @default(autoincrement())
  name     String
  role     String
```

```
 storeId  Int
  store    Store  @relation(fields: [storeId], references: [id])
}

model Product {
  id          Int     @id @default(autoincrement())
  name        String
  price       Float
  stock       Int
  storeId     Int
  store       Store   @relation(fields: [storeId], references: [id])
  sales       Sale[]
}

model Sale {
  id         Int      @id @default(autoincrement())
  productId  Int
  storeId    Int
  quantity   Int
  total      Float
  date       DateTime @default(now())

  product    Product  @relation(fields: [productId], references: [id])
  store      Store    @relation(fields: [storeId], references: [id])
}

FROM node:20-alpine

WORKDIR /usr/src/app

# Встановлення залежностей
COPY package*.json ./
RUN npm ci --omit=dev

# Копіюємо решту файлів (включаючи schema.prisma)
COPY . .

# Генеруємо Prisma Client під час збірки
RUN npx prisma generate

EXPOSE 3000
CMD ["npm", "start"]

import React from "react";
import ReactDOM from "react-dom/client";
import App from './App.jsx';
import './index.css';
import { BrowserRouter } from "react-router-dom";

ReactDOM.createRoot(document.getElementById("root")).render(
  <React.StrictMode>
    <BrowserRouter>
```

```
 <App />
   </BrowserRouter>
  </React.StrictMode>
);



import { Routes, Route, Navigate } from "react-router-dom";
import Auth from "./pages/Auth";
import Profile from "./pages/Profile";

function App() {
  return (
    <Routes>
      <Route path="/auth" element={<Auth />} />
      <Route path="/profile" element={<Profile />} />
      <Route path="*" element={<Navigate to="/auth" />} />
    </Routes>
  );
}

export default App;

import React, { useEffect, useState } from "react";
import axios from "axios";
import AuthGraphic from "../assets/auth-graphic.png";
import Snackbar from "@mui/material/Snackbar";
import MuiAlert from "@mui/material/Alert";

const Alert = React.forwardRef(function Alert(props, ref) {
  return <MuiAlert elevation={6} ref={ref} variant="filled" {...props} />;
});

// Перевірка валідності паролю
function isValidPassword(password) {
  return /^(?=.*[a-z])(?=.*[A-Z])(?=.*\d).{8,}$/.test(password);
}

export default function Auth() {
  const [isLogin, setIsLogin] = useState(true);
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");
  const [role, setRole] = useState("user");
  const [stores, setStores] = useState([]);
  const [storeId, setStoreId] = useState("");
  const [error, setError] = useState("");
  const [firstName, setFirstName] = useState("");
  const [lastName, setLastName] = useState("");
  const [snack, setSnack] = useState({ open: false, message: "", severity: "success" });

  useEffect(() => {
    if (isLogin || role !== "cashier") return;
```

```
(async () => {
    try {
      const { data } = await axios.get("http://localhost:3000/stores-lite");
      setStores(data);
    } catch {
      setError("Не вдалося отримати перелік магазинів");
    }
  })();
 }, [isLogin, role]);

 const handleSubmit = async (e) => {
  e.preventDefault();
  setError("");

  if (!isLogin && role === "cashier" && !storeId)
   return setError("Для касира оберіть магазин");

  if (!isLogin && !isValidPassword(password)) {
   setError("Пароль має містити мінімум 8 символів, одну велику, одну малу літеру і одну цифру");
   return;
  }

  const url = isLogin
   ? "http://localhost:3000/login"
   : "http://localhost:3000/register";

  const payload = isLogin
   ? { email, password }
   : role === "cashier"
   ? { email, password, role, storeId: Number(storeId), firstName, lastName }
   : { email, password, role, firstName, lastName };

  try {
   const { data } = await axios.post(url, payload);

   if (isLogin) {
    localStorage.setItem("token", data.token);
    window.location.href = "/profile";
   } else {
    setSnack({
     open: true,
     message: "Реєстрація успішна! Тепер увійдіть.",
     severity: "success",
    });
    setIsLogin(true);
    setPassword("");
   }
  } catch {
   setError("Невірні дані або email вже використовується");
  }
 };
```

```
        return (
          <div className="min-h-screen w-full flex bg-gray-100">
            {/* Ліва синя колонка */}
            <div className="w-1/3 bg-blue-600 text-white p-10 flex items-center justify-center">
              <div>
                <h2 className="text-4xl font-extrabold mb-6 leading-tight">
                  Retail WebApp
                </h2>
                <p className="text-xl leading-relaxed">
                    Авторизуйтеся або зареєструйтеся, щоб отримати доступ до системи керування мережею магазинів.
                </p>
              </div>
            </div>

            {/* Середня колонка з формою */}
            <div className="w-1/3 bg-blue-100 p-10 flex items-center justify-center">
              <div className="w-full max-w-sm">
                <h2 className="text-3xl font-bold mb-6 text-center">
                  {isLogin ? "Вхід" : "Реєстрація"}
                </h2>

                {error && <p className="text-red-500 mb-4 text-center">{error}</p>}

                <form onSubmit={handleSubmit} className="space-y-4">
                  {/* email */}
                  <div>
                    <label className="block text-sm font-medium mb-1">Email</label>
                    <input
                      type="email"
                      required
                      value={email}
                      onChange={(e) => setEmail(e.target.value)}
                      className="w-full border rounded px-3 py-2 focus:outline-none focus:ring focus:ring-blue-300"
                    />
                  </div>
                  {/* password */}
                  <div>
                    <label className="block text-sm font-medium mb-1">Пароль</label>
                    <input
                      type="password"
                      required
                      value={password}
                      onChange={(e) => setPassword(e.target.value)}
                      className="w-full border rounded px-3 py-2 focus:outline-none focus:ring focus:ring-blue-300"
                    />
                    {!isLogin && (
                      <span className="text-xs text-gray-600">
                        Мінімум 8 символів, одна велика, одна мала літера і одна цифра
                      </span>
                    )}
                  </div>
```

```
{/* Додаткові поля лише при реєстрації */}
        {!isLogin && (
         <>
          <div>
           <label className="block text-sm font-medium mb-1">Ім'я</label>
           <input
            type="text"
            required
            value={firstName}
            onChange={(e) => setFirstName(e.target.value)}
            className="w-full border rounded px-3 py-2 focus:outline-none focus:ring focus:ring-blue-300"
           />
          </div>
          <div>
           <label className="block text-sm font-medium mb-1">Прізвище</label>
           <input
            type="text"
            required
            value={lastName}
            onChange={(e) => setLastName(e.target.value)}
            className="w-full border rounded px-3 py-2 focus:outline-none focus:ring focus:ring-blue-300"
           />
          </div>
          <div>
           <label className="block text-sm font-medium mb-1">Роль</label>
           <select
            value={role}
            onChange={(e) => {
             setRole(e.target.value);
             setStoreId("");
            }}
            className="w-full border rounded px-3 py-2 focus:outline-none focus:ring focus:ring-blue-300"
           >
            <option value="user">Користувач</option>
            <option value="cashier">Касир</option>
            <option value="admin">Адміністратор</option>
           </select>
          </div>
          {role === "cashier" && (
           <div>
            <label className="block text-sm font-medium mb-1">Магазин</label>
            <select
             value={storeId}
             onChange={(e) => setStoreId(e.target.value)}
             required
             className="w-full border rounded px-3 py-2 focus:outline-none focus:ring focus:ring-blue-300"
            >
             <option value="">- Оберіть магазин -</option>
             {stores.map((s) => (
              <option key={s.id} value={s.id}>
               {s.name} (ID&amp;nbsp;{s.id})
              </option>
             ))}
```

```
            </select>
                </div>
              )}
            </>
          )}
          {/* Кнопка */}
          <button
            type="submit"
            className="w-full bg-blue-600 text-white py-2 rounded hover:bg-blue-700 transition"
          >
            {isLogin ? "Увійти" : "Зареєструватись"}
          </button>
        </form>
        {/* Перемикач форми */}
        <p className="mt-4 text-center text-sm text-gray-700">
          {isLogin ? "Не маєте акаунту?" : "Вже є акаунт?"}{" "}
          <button
            type="button"
            onClick={() => setIsLogin(!isLogin)}
            className="text-blue-600 hover:underline"
          >
            {isLogin ? "Зареєструватись" : "Увійти"}
          </button>
        </p>
      </div>
    </div>
    {/* Права ілюстрація */}
    <div className="w-1/3">
      <img
        src={AuthGraphic}
        alt="Retail illustration"
        className="w-full h-full object-cover"
      />
    </div>
    {/* Snackbar pop-up */}
    <Snackbar
      open={snack.open}
      autoHideDuration={3500}
      onClose={() => setSnack({ ...snack, open: false })}
      anchorOrigin={{ vertical: "bottom", horizontal: "center" }}
    >
      <Alert
        severity={snack.severity}
        onClose={() => setSnack({ ...snack, open: false })}
      >
        {snack.message}
      </Alert>
    </Snackbar>
  </div>
 );
}
```

```
import { useEffect, useState, useRef } from "react";
import axios from "axios";
import {
  Box, Grid, CircularProgress, MenuItem, Select,
  Typography, Avatar, Chip, IconButton, Tooltip, Paper, Snackbar
} from "@mui/material";
import CameraAltIcon from '@mui/icons-material/CameraAlt';
import MuiAlert from '@mui/material/Alert';

import StoreCard from "../components/StoreCard";
import ProductCard from "../components/ProductCard";
import SaleCard from "../components/SaleCard";
import EmployeeCard from "../components/EmployeeCard";

const ROLE_UA = {
  admin: "Адміністратор",
  cashier: "Касир",
  user: "Користувач"
};

export default function Profile() {
  const [user, setUser] = useState(null);
  const [stores, setStores] = useState([]);
  const [activeId, setActiveId] = useState(null);
  const [products, setProducts] = useState([]);
  const [sales, setSales] = useState([]);
  const [employees, setEmployees] = useState([]);
  const [loading, setLoading] = useState(true);
  const [uploading, setUploading] = useState(false);

  const [snackbar, setSnackbar] = useState({
    open: false,
    message: "",
    severity: "error"
  });

  const showSnackbar = (message, severity = "error") => {
    setSnackbar({ open: true, message, severity });
  };

  const fileInputRef = useRef();
  const token = localStorage.getItem("token");
  const auth = { headers: { Authorization: `Bearer ${token}` } };

  useEffect(() => {
    if (!token) return;
    (async () => {
      try {
        const { data: { user } } = await axios.get("http://localhost:3000/profile", auth);
        setUser(user);
        const { data: allStores } = await axios.get("http://localhost:3000/stores", auth);
        setStores(allStores);
        // Для admin - перший магазин активний, для інших - усі магазини
```

```
 if (user.role === "admin") setActiveId(allStores[0]?.id ?? null);
      else setActiveId(0);
    } catch (e) {
      console.error(e);
    } finally {
      setLoading(false);
    }
  })();
}, []);

useEffect(() => {
  if (!user) return;
  const query = activeId && activeId !== 0
    ? { params: { storeId: activeId }, ...auth }
    : auth;
  (async () => {
    try {
      const promises = [axios.get("http://localhost:3000/products", query)];
      if (user.role === "admin") {
        promises.push(
          axios.get("http://localhost:3000/sales", query),
          axios.get("http://localhost:3000/employees", query)
        );
      }
      const [prRes, slRes, emRes] = await Promise.all(promises);
      setProducts(prRes.data);
      setSales(slRes?.data ?? []);
      setEmployees(emRes?.data ?? []);
    } catch (e) {
      console.error(e);
    }
  })();
}, [activeId, user]);

const handleAvatarChange = async (e) => {
  const file = e.target.files?.[0];
  if (!file) return;
  setUploading(true);
  const formData = new FormData();
  formData.append('avatar', file);

  try {
    await axios.post('http://localhost:3000/profile/avatar', formData, {
      headers: {
        Authorization: `Bearer ${token}`,
        'Content-Type': 'multipart/form-data'
      }
    });
    const { data: { user } } = await axios.get("http://localhost:3000/profile", auth);
    setUser(user);
  } catch (err) {
    showSnackbar("Помилка при завантаженні аватарки", "error");
  } finally {
```

```
 setUploading(false);
  }
};

if (loading || !user)
  return (
    <Box sx={{ minHeight: "100vh", display: "flex", alignItems: "center", justifyContent: "center" }}>
      <CircularProgress />
    </Box>
  );

const showFirstName = user.firstName?.trim() || "";
const showLastName = user.lastName?.trim() || "";
const avatarUrl = user.avatarUrl
  ? (user.avatarUrl.startsWith("http") ? user.avatarUrl : `http://localhost:3000${user.avatarUrl}`)
  : undefined;

return (
  <Box sx={{
    bgcolor: "#f5f6fa",
    minHeight: "100vh",
    py: { xs: 2, md: 8 },
    px: { xs: 0, md: 4 },
  }}>
    {/* ------- profile header ---------- */}
    <Paper elevation={6} sx={{
      mb: 4,
      borderRadius: 4,
      p: { xs: 2, md: 4 },
      display: "flex",
      alignItems: "center",
      gap: 4,
      bgcolor: "#fff8ee",
      flexWrap: { xs: "wrap", md: "nowrap" },
    }}>
      {/* Аватар і кнопка */}
      <Box position="relative" display="flex" flexDirection="column" alignItems="center">
        <Avatar
          src={avatarUrl}
          sx={{
            bgcolor: "primary.main",
            width: 120, height: 120, fontSize: 60,
            boxShadow: 3,
            border: "4px solid #fff",
          }}
        >
          {(showFirstName[0] || user.email[0] || "U").toUpperCase()}
        </Avatar>
        <Tooltip title="Змінити аватар">
          <IconButton
            size="medium"
            sx={{
              position: "absolute",
```

```
          bottom: 10,
          right: 10,
          bgcolor: "#fff",
          boxShadow: 2,
          "&:hover": { bgcolor: "grey.200" },
          border: "2px solid #eee"
        }}
        component="span"
        onClick={() => fileInputRef.current?.click()}
        disabled={uploading}
      >
        <CameraAltIcon fontSize="medium" color="action" />
      </IconButton>
    </Tooltip>
    <input
      ref={fileInputRef}
      type="file"
      accept="image/*"
      style={{ display: "none" }}
      onChange={handleAvatarChange}
    />
  </Box>
  {/* Інформація про користувача */}
  <Box flex={1} minWidth={0}>
    <Typography variant="h4" fontWeight={700} color="primary.main" mb={1}>
      {`Привіт, ${showFirstName || user.email || "користувачу"}!`}
    </Typography>
    <Box mb={1} display="flex" alignItems="center">
      <Typography variant="subtitle1" fontWeight={600}>ПІБ: </Typography>
      <Typography variant="h6" sx={{ display: "inline", mr: 2 }}>
        {(showLastName + " " + showFirstName).trim() || "-"}
      </Typography>
      <Chip
        label={ROLE_UA[user.role] || user.role}
        color={
          user.role === "admin" ? "primary" :
            user.role === "cashier" ? "success" : "default"
        }
        size="small"
        sx={{ fontWeight: 600, fontSize: 15, height: 28 }}
      />
    </Box>
    <Box mb={1} display="flex" alignItems="center">
      <Typography variant="subtitle1" fontWeight={600}>Email: </Typography>
      <Typography variant="h6" sx={{ display: "inline" }}>{user.email}</Typography>
    </Box>
    <Box mb={1} display="flex" alignItems="center">
      <Typography variant="subtitle1" fontWeight={600}>Магазин: </Typography>
      <Select
        size="small"
        value={activeId}
        onChange={e => setActiveId(Number(e.target.value))}
```

```
          sx={{
           bgcolor: "#fff",
           borderRadius: 2,
           minWidth: 170,
           maxWidth: 280,
           fontSize: 16,
           ml: 1,
          }}
         >
          <MenuItem value={0}>Усі магазини</MenuItem>
          {stores.map(s => (
           <MenuItem key={s.id} value={s.id}>{s.name}</MenuItem>
          ))}
         </Select>
        </Box>
       </Box>
      </Paper>

      {/* ------- cards ---------- */}
      <Grid container spacing={4}>
       {/* Всі бачать магазини (read only для cashier/user) */}
       <Grid item xs={12}>
        <StoreCard
         rows={stores}
         setRows={setStores}
         auth={auth}
         userRole={user.role}
        />
       </Grid>
       {/* Товари бачать всі */}
       <Grid item xs={12}>
        <ProductCard
         rows={products}
         setRows={setProducts}
         auth={auth}
         storeId={activeId === 0 ? null : activeId}
         userRole={user.role}
         userStore={user.storeId}
        />
       </Grid>
       {/* Інші картки - лише для адміна */}
       {user.role === "admin" && (
        <>
         <Grid item xs={12}>
          <SaleCard rows={sales} setRows={setSales} auth={auth} storeId={activeId} />
         </Grid>
         <Grid item xs={12}>
          <EmployeeCard rows={employees} setRows={setEmployees} auth={auth} storeId={activeId} />
         </Grid>
        </>
       )}
```

```
      </Grid>
      <Snackbar
        open={snackbar.open}
        autoHideDuration={3000}
        onClose={() => setSnackbar({ ...snackbar, open: false })}
        anchorOrigin={{ vertical: 'top', horizontal: 'center' }}
      >
        <MuiAlert
          onClose={() => setSnackbar({ ...snackbar, open: false })}
          severity={snackbar.severity}
          sx={{ width: '100%' }}
        >
          {snackbar.message}
        </MuiAlert>
      </Snackbar>
    </Box>
  );
}

import { useState } from "react";
import axios from "axios";
import {
  Card, CardHeader, IconButton, Divider,
  Box, Dialog, DialogTitle, DialogContent, DialogActions,
  Button, TextField
} from "@mui/material";
import { DataGrid } from "@mui/x-data-grid";
import AddIcon from "@mui/icons-material/Add";
import EditIcon from "@mui/icons-material/Edit";
import DeleteIcon from "@mui/icons-material/Delete";

export default function StoreCard({ rows, setRows, auth, userRole }) {
  const [openForm, setOpenForm] = useState(false);
  const [openDel, setOpenDel] = useState(false);
  const [cur, setCur] = useState({ id: null, name: "", location: "" });

  const refetch = async () => {
    const { data } = await axios.get("http://localhost:3000/stores", auth);
    setRows(data);
  };
  const reset = () => setCur({ id: null, name: "", location: "" });

  const handleSave = async () => {
    const payload = { name: cur.name, location: cur.location };
    if (cur.id === null) {
      await axios.post("http://localhost:3000/stores", payload, auth);
    } else {
      await axios.put(`http://localhost:3000/stores/${cur.id}`, payload, auth);
    }
    await refetch();
    setOpenForm(false);
    reset();
  };
```

```
const handleDelete = async () => {
  await axios.delete(`http://localhost:3000/stores/${cur.id}`, auth);
  await refetch();
  setOpenDel(false);
  reset();
};

// Тільки для адміна дії над магазинами
const columns = [
  { field: "id", headerName: "ID", width: 70 },
  { field: "name", headerName: "Назва", flex: 1 },
  { field: "location", headerName: "Адреса", flex: 1 },
  ...(userRole === "admin"
    ? [{
      field: "actions",
      headerName: "",
      width: 110,
      renderCell: ({ row }) => (
        <>
          <IconButton size="small" onClick={() => { setCur(row); setOpenForm(true); }}>
            <EditIcon fontSize="small" />
          </IconButton>
          <IconButton size="small" color="error" onClick={() => { setCur(row); setOpenDel(true); }}>
            <DeleteIcon fontSize="small" />
          </IconButton>
        </>
      )
    }]
    : [])
];

return (
  <Card sx={{ height: 420, display: "flex", flexDirection: "column", boxShadow: 3 }}>
    <CardHeader
      title="Магазини"
      action={
        userRole === "admin" && (
          <IconButton
            onClick={() => { reset(); setOpenForm(true); }}
            sx={{ color: "#fff" }}
          >
            <AddIcon />
          </IconButton>
        )
      }
      sx={{ bgcolor: "primary.main", color: "#fff" }}
    />
    <Divider />
    <Box sx={{ flexGrow: 1 }}>
      <DataGrid
        rows={rows}
        columns={columns}
```

```
          density="compact"
          disableRowSelectionOnClick
        />
      </Box>

      {/* Діалоги (тільки для адміна) */}
      {userRole === "admin" && (
       <>
        <Dialog open={openForm} onClose={() => setOpenForm(false)}>
         <DialogTitle>
          {cur.id ? "Редагувати магазин" : "Додати магазин"}
         </DialogTitle>
         <DialogContent
          sx={{
           display: "flex",
           flexDirection: "column",
           gap: 2,
           mt: 1,
           overflow: "visible",
          }}
         >
          <TextField
           label="Назва"
           value={cur.name}
           onChange={e => setCur({ ...cur, name: e.target.value })}
          />
          <TextField
           label="Адреса"
           value={cur.location}
           onChange={e => setCur({ ...cur, location: e.target.value })}
          />
         </DialogContent>
         <DialogActions>
          <Button onClick={() => setOpenForm(false)}>Скасувати</Button>
          <Button variant="contained" onClick={handleSave}>Зберегти</Button>
         </DialogActions>
        </Dialog>

        <Dialog open={openDel} onClose={() => setOpenDel(false)}>
         <DialogTitle>Видалити магазин?</DialogTitle>
         <DialogActions>
          <Button onClick={() => setOpenDel(false)}>Hi</Button>
          <Button color="error" variant="contained" onClick={handleDelete}>Так</Button>
         </DialogActions>
        </Dialog>
       </>
      )}
     </Card>
    );
   }
```

```
import { useEffect, useState } from "react";
import axios from "axios";
import {
  Card, CardHeader, Divider, Box, IconButton,
  Dialog, DialogTitle, DialogContent, DialogActions,
  TextField, Button, MenuItem
} from "@mui/material";
import { DataGrid } from "@mui/x-data-grid";
import AddIcon from "@mui/icons-material/Add";
import EditIcon from "@mui/icons-material/Edit";
import DeleteIcon from "@mui/icons-material/Delete";

export default function SaleCard({ rows, setRows, auth, storeId }) {
  const [products, setProducts] = useState([]);
  const [openForm, setOpenForm] = useState(false);
  const [openDel, setOpenDel] = useState(false);

  const blank = { id: null, productId: "", quantity: "", total: 0 };
  const [cur, setCur] = useState(blank);

  useEffect(() => { if (storeId) refreshData(); }, [storeId]);
  const reset = () => setCur(blank);

  async function handleSave() {
    const productId = Number(cur.productId);
    const quantity = Number(cur.quantity);
    const body = { productId, quantity };

    try {
      if (cur.id == null) {
        await axios.post("http://localhost:3000/sales", body, auth);
      } else {
        await axios.put(`http://localhost:3000/sales/${cur.id}`, body, auth);
      }
      await refreshData();
      setOpenForm(false);
      reset();
    } catch (e) {
      console.error("[Save Sale ERROR]", e);
    }
  }

  async function handleDelete() {
    try {
      await axios.delete(`http://localhost:3000/sales/${cur.id}`, auth);
      await refreshData();
      setOpenDel(false);
      reset();
    } catch (e) {
      console.error("[Delete Sale ERROR]", e);
    }
  }
```

```
async function refreshData() {
  try {
    const [prodRes, saleRes] = await Promise.all([
      axios.get("http://localhost:3000/products", { params: { storeId }, ...auth }),
      axios.get("http://localhost:3000/sales", { params: { storeId }, ...auth }),
    ]);
    setProducts(Array.isArray(prodRes.data) ? prodRes.data : []);
    setRows(Array.isArray(saleRes.data) ? saleRes.data : []);
  } catch (e) {
    console.error("[Refresh ERROR]", e);
  }
}

const getProduct = (id) => products.find((p) => p.id === id);

const flatRows = rows.map((row) => {
  const product = row.product || {};
  return {
    ...row,
    productName: product.name ?? "-",
    unitPrice: typeof product.price === "number" ? product.price : null,
    stock: product.stock ?? "-",
    total:
      typeof product.price === "number" && typeof row.quantity === "number"
        ? product.price * row.quantity
        : null,
  };
});

// Колонки таблиці
const columns = [
  { field: "id", headerName: "ID", width: 70 },
  { field: "productName", headerName: "Товар", flex: 1 },
  { field: "unitPrice", headerName: "Ціна за од.", width: 120 },
  { field: "quantity", headerName: "К-сть", width: 90 },
  { field: "stock", headerName: "Залишок", width: 100 },
  { field: "total", headerName: "Сума", width: 120 },
  {
    field: "actions",
    headerName: "",
    width: 110,
    sortable: false,
    filterable: false,
    renderCell: (params) => {
      const row = params?.row;
      if (!row) return null;
      return (
        <>
          <IconButton size="small" onClick={() => { setCur(row); setOpenForm(true); }}>
            <EditIcon fontSize="small" />
          </IconButton>
          <IconButton size="small" color="error" onClick={() => { setCur(row); setOpenDel(true); }}>
```

```
            <DeleteIcon fontSize="small" />
          </IconButton>
        </>
      );
    },
  },
];

// Загальна сума по таблиці
const grandTotal = flatRows.reduce(
  (sum, r) =>
    typeof r.unitPrice === "number" && typeof r.quantity === "number"
      ? sum + r.unitPrice * r.quantity
      : sum,
  0
);

return (
  <Card sx={{ height: 420, display: "flex", flexDirection: "column", boxShadow: 3 }}>
    <CardHeader
      title="Продажі"
      action={
        <IconButton
          sx={{ color: "#fff" }}
          disabled={!storeId}
          onClick={() => { reset(); setOpenForm(true); }}
        >
          <AddIcon />
        </IconButton>
      }
      sx={{ bgcolor: "warning.main", color: "#fff" }}
    />
    <Divider />

    {/* Таблиця продажів */}
    <Box sx={{ flexGrow: 1 }}>
      <DataGrid
        rows={flatRows}
        columns={columns}
        density="compact"
        disableRowSelectionOnClick
        getRowId={(r) => r?.id}
      />
    </Box>

    {/* Сумарна вартість */}
    <Box sx={{ p: 1, textAlign: "right", fontWeight: "bold" }}>
      Разом: ₴ {grandTotal.toFixed(2)}
    </Box>

    {/* Діалог створення/редагування */}
    <Dialog open={openForm} onClose={() => setOpenForm(false)} fullWidth maxWidth="sm">
```

```
<DialogTitle>{cur.id ? "Редагувати продаж" : "Додати продаж"}</DialogTitle>
<DialogContent
 sx={{
  display: "flex",
  flexDirection: "column",
  gap: 2,
  mt: 1,
  overflow: "visible"
 }}
>
 <TextField
  select
  label="Товар"
  value={cur.productId}
  onChange={(e) => {
   const pid = +e.target.value;
   const price = getProduct(pid)?.price ?? 0;
   setCur((c) => ({ ...c, productId: pid, total: price * (c.quantity || 0) }));
  }}
 >
  {products.map((p) => (
   <MenuItem key={p.id} value={p.id}>
    {p.name} (₴{p.price})
   </MenuItem>
  ))}
 </TextField>
 <TextField
  type="number"
  label="Кількість"
  value={cur.quantity}
  onChange={(e) => {
   const qty = +e.target.value;
   const price = getProduct(+cur.productId)?.price ?? 0;
   setCur((c) => ({ ...c, quantity: qty, total: price * qty }));
  }}
 />
 <TextField
  label="Сума"
  value={cur.total.toFixed(2)}
  InputProps={{ readOnly: true }}
 />
</DialogContent>
<DialogActions>
 <Button onClick={() => setOpenForm(false)}>Скасувати</Button>
 <Button
  variant="contained"
  onClick={handleSave}
  disabled={!+cur.productId || !+cur.quantity}
 >
  Зберегти
 </Button>
</DialogActions>
```

```
      </Dialog>

      {/* Діалог підтвердження видалення */}
      <Dialog open={openDel} onClose={() => setOpenDel(false)}>
        <DialogTitle>Видалити запис про продаж?</DialogTitle>
        <DialogActions>
          <Button onClick={() => setOpenDel(false)}>Ні</Button>
          <Button color="error" variant="contained" onClick={handleDelete}>Так</Button>
        </DialogActions>
      </Dialog>
    </Card>
  );
}

import { useEffect, useState } from "react";
import axios from "axios";
import {
  Card, CardHeader, Divider, Box, IconButton,
  Dialog, DialogTitle, DialogContent, DialogActions,
  TextField, Button
} from "@mui/material";
import { DataGrid } from "@mui/x-data-grid";
import AddIcon    from "@mui/icons-material/Add";
import EditIcon   from "@mui/icons-material/Edit";
import DeleteIcon from "@mui/icons-material/Delete";

export default function ProductCard({
  rows, setRows,
  auth,
  storeId,
  userRole,
  userStore
}) {

  const canEditRow = (sid) =>
    userRole === "admin" ||
    (userRole === "cashier" && +sid === +userStore);

  const canAdd = userRole === "admin" ||
              (userRole === "cashier" && +storeId === +userStore);

  // Шаблон товару
  const blank = {
    id: null,
    name: "",
    price: "",
    stock: "",
    storeId: userRole === "cashier" ? userStore : (storeId || 1)
  };

  const [cur, setCur] = useState(blank);
  const [openForm, setOpenForm] = useState(false);
  const [openDel, setOpenDel] = useState(false);
```

```
useEffect(() => { refetch(); }, [storeId]);
async function refetch() {
  const q = storeId ? { params: { storeId }, ...auth } : auth;
  const { data } = await axios.get("http://localhost:3000/products", q);
  setRows(data);
}

const reset = () => setCur(blank);

async function handleSave() {
  const body = {
    name: cur.name,
    price: +cur.price,
    stock: +cur.stock,
    storeId: +cur.storeId
  };
  if (cur.id === null)
    await axios.post("http://localhost:3000/products", body, auth);
  else
    await axios.put(`http://localhost:3000/products/${cur.id}`, body, auth);

  await refetch();
  setOpenForm(false);
  reset();
}

async function handleDelete() {
  await axios.delete(`http://localhost:3000/products/${cur.id}`, auth);
  await refetch();
  setOpenDel(false);
  reset();
}

// Колонки DataGrid
const columns = [
  { field: "id",      headerName: "ID",          width: 70 },
  { field: "name",    headerName: "Назва",       flex: 1 },
  { field: "price",   headerName: "Ціна",        width: 90 },
  { field: "stock",   headerName: "К-сть",       width: 90 },
  { field: "storeId", headerName: "ID магазину", width: 120 },
  {
    field: "actions",
    headerName: "",
    width: 110,
    renderCell: ({ row }) => canEditRow(row.storeId) && (
      <>
        <IconButton size="small"
          onClick={() => { setCur(row); setOpenForm(true); }}>
          <EditIcon fontSize="small" />
        </IconButton>
        <IconButton size="small" color="error"
          onClick={() => { setCur(row); setOpenDel(true); }}>
```

```
          <DeleteIcon fontSize="small" />
        </IconButton>
      </>
    )
  }
];

// UI
return (
  <Card sx={{ height: 420, display: "flex", flexDirection: "column", boxShadow: 3 }}>
    <CardHeader
      title="Товари"
      action={canAdd && (
        <IconButton sx={{ color: "#fff" }}
          onClick={() => { reset(); setOpenForm(true); }}>
          <AddIcon />
        </IconButton>
      )}
      sx={{ bgcolor: "success.main", color: "#fff" }}
    />
    <Divider />
    <Box sx={{ flexGrow: 1 }}>
      <DataGrid
        rows={rows}
        columns={columns}
        density="compact"
        disableRowSelectionOnClick
      />
    </Box>

    {/* Форма додавання/редагування */}
    <Dialog open={openForm} onClose={() => setOpenForm(false)} fullWidth maxWidth="sm">
      <DialogTitle>{cur.id ? "Редагувати товар" : "Додати товар"}</DialogTitle>
      <DialogContent sx={{
        display: "flex", flexDirection: "column", gap: 2, mt: 1, overflow: "visible"
      }}>
        <TextField
          label="Назва"
          value={cur.name}
          onChange={e => setCur({ ...cur, name: e.target.value })}
        />
        <TextField
          label="Ціна"
          type="number"
          value={cur.price}
          onChange={e => setCur({ ...cur, price: e.target.value })}
        />
        <TextField
          label="Кількість"
          type="number"
          value={cur.stock}
          onChange={e => setCur({ ...cur, stock: e.target.value })}
```

```
          />
          <TextField
            label="ID магазину"
            type="number"
            value={cur.storeId}
            disabled={userRole === "cashier"}
            onChange={e => setCur({ ...cur, storeId: e.target.value })}
          />
        </DialogContent>
        <DialogActions>
          <Button onClick={() => setOpenForm(false)}>Скасувати</Button>
          <Button variant="contained" onClick={handleSave}>Зберегти</Button>
        </DialogActions>
      </Dialog>

      {/* Діалог видалення */}
      <Dialog open={openDel} onClose={() => setOpenDel(false)}>
        <DialogTitle>Видалити товар?</DialogTitle>
        <DialogActions>
          <Button onClick={() => setOpenDel(false)}>Hi</Button>
          <Button color="error" variant="contained" onClick={handleDelete}>Так</Button>
        </DialogActions>
      </Dialog>
    </Card>
  );
}

import { useState } from "react";
import axios from "axios";
import {
  Card, CardHeader, Divider, Box, IconButton,
  Dialog, DialogTitle, DialogContent, DialogActions,
  TextField, Button
} from "@mui/material";
import { DataGrid } from "@mui/x-data-grid";
import AddIcon    from "@mui/icons-material/Add";
import EditIcon   from "@mui/icons-material/Edit";
import DeleteIcon from "@mui/icons-material/Delete";

export default function EmployeeCard({ rows, setRows, auth, storeId }) {
  const [openForm, setOpenForm] = useState(false);
  const [openDel, setOpenDel] = useState(false);
  const [cur, setCur] = useState({ id: null, name: "", role: "", storeId: "" });

  const query = { params: { storeId }, ...auth };
  const refetch = async () => {
    const { data } = await axios.get("http://localhost:3000/employees", query);
    setRows(data);
  };

  const reset = () => setCur({ id: null, name: "", role: "", storeId: "" });

  const handleSave = async () => {
```

```
  const body = {
    name: cur.name,
    role: cur.role,
    storeId: Number(cur.storeId)
  };
  if (cur.id === null) {
    await axios.post("http://localhost:3000/employees", body, auth);
  } else {
    await axios.put(`http://localhost:3000/employees/${cur.id}`, body, auth);
  }
  await refetch();
  setOpenForm(false);
  reset();
};

const handleDelete = async () => {
  await axios.delete(`http://localhost:3000/employees/${cur.id}`, auth);
  await refetch();
  setOpenDel(false);
  reset();
};

// Колонки для DataGrid
const columns = [
  { field: "id",       headerName: "ID",        width: 70 },
  { field: "name",     headerName: "Ім'я",      flex: 1 },
  { field: "role",     headerName: "Роль",      width: 120 },
  { field: "storeId",  headerName: "ID магаз.", width: 110 },
  {
    field: "actions",
    headerName: "",
    width: 110,
    renderCell: ({ row }) => (
      <>
        <IconButton size="small" onClick={() => { setCur(row); setOpenForm(true); }}>
          <EditIcon fontSize="small" />
        </IconButton>
        <IconButton size="small" color="error"
          onClick={() => { setCur(row); setOpenDel(true); }}>
          <DeleteIcon fontSize="small" />
        </IconButton>
      </>
    )
  }
];

// UI
return (
  <Card sx={{
    height: 420,
    display: "flex",
    flexDirection: "column",
```

```
 boxShadow: 3
}}>
 <CardHeader
  title="Працівники"
  action={
   <IconButton
    onClick={() => { reset(); setOpenForm(true); }}
    sx={{ color: "#fff" }}>
    <AddIcon />
   </IconButton>
  }
  sx={{ bgcolor: "#9c27b0", color: "#fff" }}
 />
 <Divider />

 <Box sx={{ flexGrow: 1 }}>
  <DataGrid
   rows={rows}
   columns={columns}
   density="compact"
   disableRowSelectionOnClick
  />
 </Box>

 {/* Форма додавання/редагування */}
 <Dialog open={openForm} onClose={() => setOpenForm(false)} fullWidth maxWidth="sm">
  <DialogTitle>{cur.id ? "Редагувати працівника" : "Додати працівника"}</DialogTitle>
  <DialogContent sx={{
   display: "flex", flexDirection: "column", gap: 2, mt: 1, overflow: "visible"
  }}>
   <TextField
    label="Ім'я"
    value={cur.name}
    onChange={e => setCur({ ...cur, name: e.target.value })}
   />
   <TextField
    label="Роль (admin / cashier / ...)"
    value={cur.role}
    onChange={e => setCur({ ...cur, role: e.target.value })}
   />
   <TextField
    label="ID магазину"
    type="number"
    value={cur.storeId}
    onChange={e => setCur({ ...cur, storeId: e.target.value })}
   />
  </DialogContent>
  <DialogActions>
   <Button onClick={() => setOpenForm(false)}>Скасувати</Button>
   <Button variant="contained" onClick={handleSave}>Зберегти</Button>
  </DialogActions>
 </Dialog>
```

```
      {/* Діалог видалення */}
      <Dialog open={openDel} onClose={() => setOpenDel(false)}>
        <DialogTitle>Видалити працівника?</DialogTitle>
        <DialogActions>
          <Button onClick={() => setOpenDel(false)}>Ні</Button>
          <Button color="error" variant="contained" onClick={handleDelete}>Так</Button>
        </DialogActions>
      </Dialog>
    </Card>
  );
}
```

```
FROM node:20-alpine
WORKDIR /app

COPY package*.json ./
RUN npm install

COPY . .

EXPOSE 5173
CMD ["npm", "run", "dev"]
```

```
version: "3.9"

services:
  db:
    image: postgres:15-alpine
    environment:
      POSTGRES_USER: retail
      POSTGRES_PASSWORD: retail
      POSTGRES_DB: retail
    volumes:
      - ./dbdata:/var/lib/postgresql/data
    ports:
      - "5432:5432"

  api:
    build: ./api
    environment:
      DATABASE_URL: postgres://retail:retail@db:5432/retail
    ports:
      - "3000:3000"
    depends_on:
      - db

  client:
    build: ./client
    ports:
      - "5173:5173"
    depends_on:
      - api
    volumes:
      - ./client:/app
```

```
    - /app/node_modules
    command: npm run dev
```