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 проект

на здобуття ступеня бакалавра

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

спеціальності 121 “Інженерія програмного забезпечення”

на тему: Система ведення обліку даних і підтримки процесів малого бізнесу

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

_____Литвинюк Роман Костянтинович_____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____доцент, к.т.н. Волокита А. М._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) асистент, Гончаренко О.О._____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент ___доц., к.т.н., доцент кафедри ІСТ Шимкович В.М._____
(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

Литвинюка Романа Костянтиновича

1. Тема проєкту *Система ведення обліку даних і підтримки процесів малого бізнесу*

керівник проєкту Волокита Артем Миколайович, доцент, к.т.н.,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 23.05.2025 №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. Огляд існуючих аналогів даної тематики.

Розділ 2. Огляд технологічного стеку для розробки системи.

Розділ 3. Архітектура та розробка системи.

Розділ 4. Огляд готової системи.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) логіка роботи системи (принципова схема), діаграма класів (функціональна схема), модель бази даних (структурна схема).

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

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Гончаренко О.О.		

7. Дата видачі завдання 04.02.2025

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	27.01.2025-23.02.2025	
2.	*Вивчення та аналіз завдання*	24.02.2025-16.03.2025	
3.	*Розробка архітектури та загальної структури системи*	17.03.2025-13.04.2025	
4.	*Розробка структур окремих підсистем*	14.04.2025-27.04.2025	
5.	*Програмна реалізація системи*	28.04.2025-25.05.2025	
6.	*Оформлення пояснювальної записки*	14.04.2025-25.05.2025	
7.	*Захист програмного продукту*	02.06.2025	
8.	*Передзахист*	02.06.2025	
9.	*Захист*	17.06.2025	

Студент-дипломник _____ Роман ЛИТВИНЮК
(підпис)

Керівник проєкту _____ Артем ВОЛОКИТА
(підпис)

АНОТАЦІЯ

У даній роботі було розглянуто проблематику автоматизації ведення обліку даних малого бізнесу та підтримки їх бізнес-процесів для керівників, менеджерів та бухгалтерів компаній. Був проведений аналіз схожих аналогів даного рішення та виокремлення цікавих рішень з них. На основі аналізу було розроблено та спроектовано програму на мові програмування Java, використовуючи фреймворк Spring. Розроблена програма надає базовий функціонал підтримки та дослідження бізнес-процесів всередині компанії та надає можливість зберігати облікові дані.

Ключові слова: бізнес, облік, автоматизація обліку, бізнес-процеси, застосунок ведення обліку даних, Java

ANNOTATION

This work addresses the issue of of automating small business accounting data and supporting their business processes for managers, managers and accountants of companies was created. An analysis of similar analogues of this problematic was conducted and interesting solutions were identified with them. Based on the analysis, a program was developed and designed in the Java programming language using the Spring framework. The developed program provides basic functional support and research of business processes within the company and provides the ability to save accounting data.

Keywords: business, accounting, accounting automation, business processes, application of accounting data, Java

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Система ведення обліку	3		
			даних і підтримки процесів			
			малого бізнесу			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Система ведення обліку	71		
			даних і підтримки процесів			
			малого бізнесу			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Система ведення обліку	1		
			даних і підтримки процесів			
			малого бізнесу			
			Логіка роботи системи			
			(принципова схема)			
	А4	***ІАЛЦ.4672008.005 Д2***	Система ведення обліку	1		
			даних і підтримки процесів			
			малого бізнесу			
			Діаграма класів			
			(функціональна схема)			
	А4	***ІАЛЦ.467200.006 Д3***	Система ведення обліку	1		
			даних і підтримки процесів			
			малого бізнесу			
			Модель бази даних			
			(структурна схема)			
	А4	***ІАЛЦ.467200.007 Д4***	Система ведення обліку	30		
			даних і підтримки процесів			
			малого бізнесу			
			Текст програмного коду			

					ІАЛЦ.467200.001 ОА		
Зм	***Лист***	***№ докум.***	***Підп***	***Дата***			
Розроб		Литвинюк Р.К.			*Система ведення обліку даних і підтримки процесів малого бізнесу* Опис альбому	Літ.	Аркуш 1 / Аркушів 1
Перев		Волокита А.М.					
						КПІ ім. Ігоря Сікорського ФІОТ ІМ-13	

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «Система ведення обліку даних і підтримки процесів малого бізнесу»

Київ – 2025

ЗМІСТ

					ІАЛЦ.467200.002 ТЗ		
		№ докум.	Підпис	Дата			
Розробив		Литвинюк Р.К.			**Система ведення обліку даних і підтрики процесів малого бізнесу** Технічне завдання	Літ.	Аркуш 1 / Аркушів 3
Перевірив		Волокита А. М.					
Реценз.		Шимкович В.М.				КПІ ім. Ігоря Сікорського,ФІОТ, ІМ-13	
Н. Контр.		Гончаренко О.О.					
Затвердив							

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку системи ведення обліку даних малого підприємства з підтримкою базових бізнес-процесів а також на подальшу підтримку та вдосконалення розробленої системи.

Областю застосування даної системи є робоча збірка програми, що виконує базові елементи ведення обліку прибутку, складових частин тощо.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою даної роботи є розробка системи для автоматизації рутинних запитів ведення обліковості даних малих підприємств та підкреслення ключових моментів ефективності бізнес-процесів.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки даного дипломного проекту є офіційні документації, публікації та статті в мережі Інтернет на дану тему, науково-технічна література.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Надавати базовий функціонал введення та виведення деяких даних.
- Бути легкою та інтуітивною в розумінні інтерфейсу
- Дозволяти передивлятись статистику з будь-якої точки світу.
- Надати вичерпну та зрозумілу документацію для розробленого продукту.

5.2. Вимоги до програмного забезпечення

- ОС Windows, Mac чи Linux.
- Доступ до браузерів Chrome, Firefox, Microsoft Edge та ін.

5.3. Вимоги до апаратної частини

- ЦП не менше ніж Intel® Core (TM) i5-10500T.
- Стабільне інтернет-з’єднання.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	27.01.2025-23.02.2025
Вивчення та аналіз завдання	24.02.2025-16.03.2025
Розробка архітектури та загальної структури системи	17.03.2025-13.04.2025
Розробка структур окремих частин системи	14.04.2025-27.04.2025
Програмна реалізація системи	28.04.2025-25.05.2025
Виправлення помилок	12.05.2025-30.05.2025
Оформлення пояснювальної записки	14.04.2025-25.05.2025

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Система ведення обліку даних і підтримки процесів малого бізнесу*»

Київ – 2025

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ERP	(Enterprise Resource Planning) Планування ресурсів підприємства
CRM	(Customer Relationship Management) Управління відносинами з клієнтами
AWS	(Amazon Web Services) Платформа хмарного сховища Amazon
PWA	(Progressive Web Application) Поступовий вебзастосунок
UI/UX	(User Interface/Experience) Дизайн користувацького досвіду
API	(Application Programming Interface) Прикладний програмний інтерфейс
ORM	(Object-Relational Mapper) Об'єктно-реляційне відображення
JAR	(Java ARchive) Java-архів
HTTP	(Hypertext Transfer Protocol) Протокол передачі гіпертексту
MVC	(Model-View-Controller) Модель–вигляд–контролер
REST	(Representational State Transfer) Передача репрезентативного стану
JSON	(JavaScript Object Notation) Запис об’єктів JavaScript
XML	(Extensible Markup Language) Розширювана мова розмітки
JWT	(JSON Web Token) Токен доступу на основі JSON
СУБД	Система керування базами даних
SQL	(Structured Query Language) Мова структурованих запитів
JPA	(Java Persistence API) Інтерфейс для Java ORM
CRUD	(Create Read Update Delete) Функції управління даними
ACID	(Atomicity Consistency Isolation Durability) Властивості для надійної роботи транзакцій БД
CLI	(Command Line Interface) Інтерфейс командного рядка

ВСТУП

У сучасних економічних умовах малий бізнес відіграє ключову роль у забезпеченні сталого розвитку, створенні робочих місць та формуванні конкурентного середовища. Проте обмеженість ресурсів, висока динаміка ринку та постійна необхідність адаптації до нових умов вимагають від підприємців ефективних інструментів управління. Одним із критично важливих елементів функціонування малого бізнесу є систематизований облік даних і підтримка операційних процесів, що дозволяють зменшити ручну працю, мінімізувати помилки та забезпечити прозорість у прийнятті рішень.

З огляду на це, впровадження системи, здатної не лише вести облік, але й підтримувати ключові процеси бізнесу, зокрема управління товарами, замовленнями, фінансами, клієнтською базою, є актуальним завданням. Особливого значення це набуває для малих підприємств, яким необхідні доступні, прості у використанні та адаптивні рішення, що не потребують великих фінансових чи технічних ресурсів.

Темою цієї роботи є якраз-таки розробка системи ведення обліку даних і підтримки процесів малого бізнесу, що спрямована на вирішення саме таких завдань. У роботі проаналізовано особливості функціонування аналогів додатків, націлених на таку саму аудиторію малого бізнесу, визначено основні вимоги до облікових систем, запропоновано архітектурну модель рішення, а також реалізовано програмний прототип, що демонструє ключову функціональність.

Метою дослідження є створення універсальної системи, яка забезпечує ефективне управління даними та автоматизацію базових бізнес-процесів малого підприємства. То ж сама робота націлена на спрощення життя керівникам, бухгалтерам та менеджерам малого бізнесу у контексті ведення потрібних даних.

РОЗДІЛ 1. ОГЛЯД ІСНУЮЧИХ АНАЛОГІВ ДАНОЇ ТЕМАТИКИ

Для початку потрібно зрозуміти які взагалі принципи використовують програми для ведення малого та середього бізнесу. Такі програми спрямовані щоб автоматизувати щоденні та рутинні операцій. Такий підхід не лише вивільняє час співробітників для більш пріоритетних завдань, але й суттєво знижує ризик механічних помилок, що в кінцевому підсумку підвищує загальну ефективність та продуктивність усієї компанії.

Одним з ключових аспектів сучасного бізнес-програмного забезпечення є централізація даних та інтеграція різноманітних процесів з великою кількістю сервісів. Замість розрізнених таблиць чи окремих програм, бізнес має єдиний інформаційний простір, в якому і зберігаються введені дані. Вся інформація про фінанси, товари, замовлення та клієнтів зберігається в одній базі даних або тісно пов'язаних модулях, забезпечуючи узгодженість даних, усуваючи їх дублювання та надає керівництву цілісне бачення стану справ у компанії та дозволяючи різним відділам працювати незалежно одне від одного.

Така централізація процесів безпосередньо сприяє покращенню контролю та прозорості бізнес-процесів. Керівники або менеджери отримують доступ до актуальних звітів, аналітики продажів, можуть відстежувати залишки на складах у реальному часі та контролювати виконання завдань. Все зазначене дає можливість швидко реагувати на будь-які зміни, виявляти проблемні зони та приймати зважені, обґрунтовані управлінські рішення на основі точних даних, а не на основі інтуїції.

1.1 Perfectum

Perfectum – українська IT-компанія, що випускає однойменний продукт для ведення бізнесу. Продукт являє собою CRM+ERP систему (контроль над відгуками клієнтів та управління ресурсами підприємства), яка має великий функціонал програм, що автоматично оновлюються та можуть запускатись на будь-якому пристрої. [2]

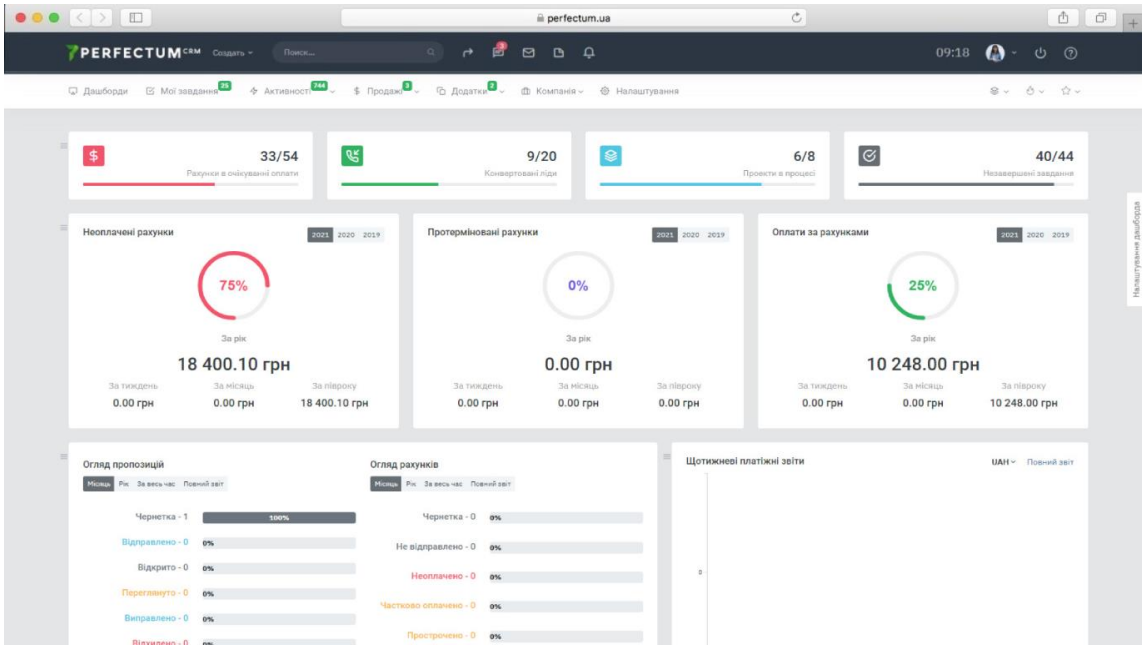


Рисунок 1.1 – Інтерфейс Perfectum

1.1.1 Функціонал, архітектура та безпека

Функціонал даної платформи є досить великим, через що кожна компанія може вибрати необхідні модулі роботи з бізнесом у їх застосунку. У функціонал входить:

1. **«Конструктор бізнес-процесів»**, що дозволяє моделювати та запускати робочі процеси для лідів, замовлень чи завдань.

2. **«Конструктор звітів»**, що дозволяє створювати звіти на базі наявних даних.
3. **«Конструктор нових модулів»**, який може створювати унікальні модулі в системі, не використовуючи програмування.
4. **«Конструктор дашбордів»**, який створює дашборди на базі наявних даних .
5. **«Моделювання бізнес-процесів»**, що являє собою детальний опис та оптимізацією операційної діяльності (блок-схеми).
6. **«Таблиці онлайн»**, які за функціоналом дуже схожі на Google Таблиці але розташовані саме у додатку Perfectum CPR+ERP. [3,4]

Функціональність Perfectum CRM+ERP не обмежується переліченими стандартними та корпоративними додатками, оскільки система має «Конструктор нових модулів». Цей потужний інструмент дозволяє користувачам самостійно, без залучення програмістів та спеціальних технічних знань, створювати абсолютно нові, унікальні розділи та функціональні блоки безпосередньо в системі. Тож якщо ваш бізнес має специфічні завдання, унікальні процеси або потребує обліку особливих даних, які не охоплюються стандартним набором, «Конструктор нових модулів» дозволяє «допрацювати» систему Perfectum саме під ці індивідуальні потреби, надаючи практично необмежені можливості налаштування.

Архітектура системи Perfectum CRM+ERP розроблена з використанням поширених веб-технологій. Клієнтська частина (фронтенд) побудована на основі JavaScript, бібліотеки JQuery, CSS для стилів та HTML для структури сторінок. Для розробки мобільних додатків для Android та iOS використовується кросплатформний фреймворк Flutter. Серверна частина (бекенд) реалізована мовою програмування PHP з використанням фреймворку Codeigniter, який забезпечує логіку роботи системи.

Як інфраструктурну основу для зберігання даних використовується система керування базами даних MariaDB. Веб-сервери, на яких розміщена система, працюють під управлінням операційних систем AlmaLinux або CentOS. Perfectum можна запускати на усіх популярних веб-браузерах, включаючи Chrome, Firefox, Opera, Safari та Edge, а також через нативні мобільні додатки.

Для розширення можливостей та інтеграції Perfectum передбачає взаємодію з низкою додаткових сервісів. До них належать ionCube для захисту коду, Pusher для функціональності сповіщень у режимі реального часу та популярні платформи та сервіси від Google, Amazon, Firebase та Dropbox. Завдяки цьому сервіс має гнучку та функціональну систему, яка може адаптуватися до різних потреб бізнесу. [5]

Perfectum CRM+ERP підходить до питань обробки даних комплексно, орієнтуючись на визнані практики, аналогічні міжнародним стандартам безпеки ISO, тобто ключові файли системних служб не зберігаються у відкритому доступі, а проходять процедуру шифрування. Простіше кажучи, важлива інформація перетворюється на захищений код, що значно ускладнює доступ до неї як для зовнішніх зловмисників, так і для спроб несанкціонованого доступу зсередини компанії.

Система надає адміністраторам спеціальний модуль для налаштування правил доступу. Наприклад, можна дозволити вхід лише з певних IP-адрес комп'ютерів або типів пристроїв, що забезпечує ще один рівень контролю. Також є можливість налаштувати автоматичне завершення робочого сеансу, якщо користувач неактивний протягом певного часу, та зробити шлях до доступу до системи менш очевидним, якщо створюються додаткові перешкоди для небажаних «гостей».

Особлива увага приділяється захисту паролів користувачів. Система дозволяє встановлювати вимоги до їх складності, стимулюючи створення більш надійних та стійких до компрометації комбінацій. Важливо, щоб самі паролі не

зберігалися в прямому, читабельному вигляді. Натомість використовується процес хешування – перетворення пароля на унікальний зашифрований код, який неможливо легко відновити до початкового стану.

Крім того, всі суттєві дії, що відбуваються в системі, остаточно записуються в спеціальний системний журнал. Ця детальна реєстрація операцій служить інструментарієм не лише для аналізу подій «постфактум» у разі інцидентів, але й для проактивного моніторингу безпеки. Адміністратори можуть відстежувати спроби доступу, виявляти незвичайну активність та оперативно реагувати на будь-які загрози, що значно підвищує загальний рівень захисту бізнес-інформації. [6]

1.2 UGLA

UGLA – комплексна ERP система нового покоління, зручна та проста, здатна налагодити ефективну комунікацію між підрозділами компанії та клієнтами, звільняючи час для важливішого. UGLA будує надійний технологічний міст між компанією та клієнтами та є абсолютно українським продуктом. [7]

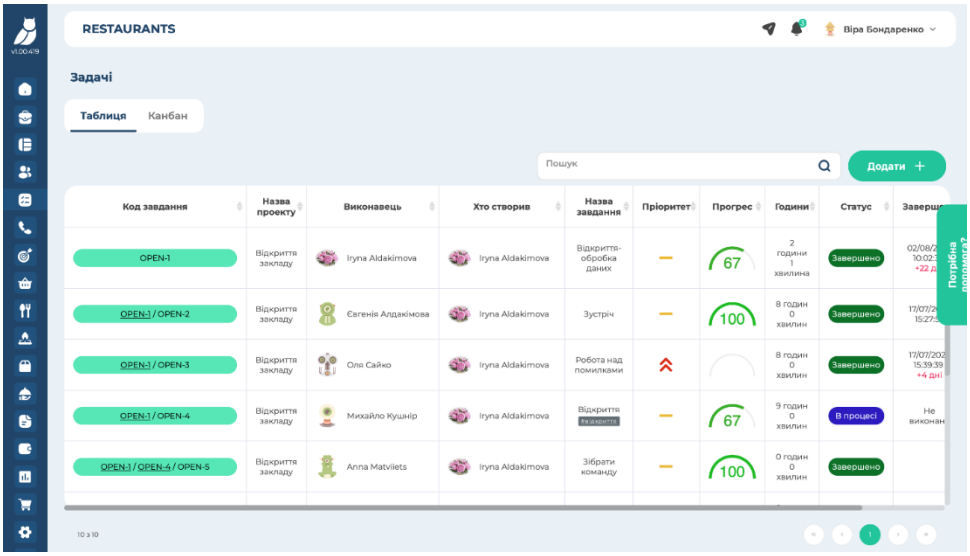


Рисунок 1.2 – Інтерфейс UGLA

1.2.1 Функціонал та архітектура

UGLA ERP поділена на 20+ модулів, які об’єднуються у один комплексний застосунок з функціоналом цих модулів. Ці модулі також явно поділені на CRM та ERP функціонал, створюючи систему для ведення бізнес-процесів. Функціонал UGLA включає в себе:

1. Модуль **«Виробництво»**, який оптимізує взаємодію на всіх етапах виробництва та допомагає підвищити його загальну ефективність для підприємств;
2. Модуль **«CRM»**, який якраз-таки відповідає за можливість ефективно вести облік та аналізувати клієнтську базу, налаштовувати фільтр продажів, а також здійснювати фінансовий і складський облік;
3. Модуль **«Контрагенти»**, схожий за функціоналом на модуль **«CRM»**, але націлений на управління взаємовідносинами з усіма типами партнерів (постачальниками, підрядниками, клієнтами) та на відстеження статусів і місцезнаходження їхніх замовлень у реальному часі;

4. Модуль **«Документообіг»**, що являє собою централізованим сховищем шаблонів договорів різного типу та забезпечує автоматизоване управління всім процесом документообігу;
5. Модулі **«Дашборд»** та **«Мій дашборд»**, що являють собою одну і ту ж саму мету показати всеосяжний огляд діяльності, прибуток та оптимізацію роботи, тільки в першому випадку компанії, а в другому – працівника;
6. Модуль **«Маркетинг»** дозволяє використовувати широкий набір ефективних інструментів для просування бренду, товарів і послуг, доповнюючи їх автоматизацією багатьох маркетингових процесів;
7. Модуль **«Персонал та HR»**, що висвітлює увесь персонал компанії та дозволяє автоматизовано та ефективно адаптувати та навчати нових працівників колективу;

та багато інших модулів, націлених на автоматизацію ведення бізнесу та його процесів. [8]

UGLA має сучасні архітектурні підходи, що забезпечують гнучкість, повсюдну доступність та високий рівень безпеки. Основою системи є хмарна інфраструктуращо базується на серверах AWS, що гарантує надійність, легку масштабованість відповідно до зростаючих потреб бізнесу та доступ до даних з будь-якої точки світу, де є підключення до Інтернету.

Важливим аспектом доступності системи є мобільний доступ, реалізований за допомогою технології PWA. Така технологія дозволяє користувачам комфортно працювати з UGLA на смартфонах та планшетах Android та iOS через веб-браузер, отримуючи досвід, максимально наближений до використання нативних додатків, і водночас завжди маючи найновішу версію системи. При цьому розробники приділяють значну увагу UI/UX елементам, наголошуючи на його максимальній зручності та інтуїтивності з гнучкими налаштуваннями під себе, включаючи багатомовну підтримку.

Як було вказано в розділі до цього, система побудована за модульним принципом, з більш ніж 20 модулями, та надає інструменти кастомізації процесів без прямого програмування, дозволяючи користувачам самостійно вибирати ефективні функції відповідно до унікальних потреб їхнього бізнесу. Для розширення функціональності та взаємодії з іншими корпоративними системами UGLA має задокументований API, який можна підлаштувати під взаємодію з іншими системами. Це відкриває можливості для легкої інтеграції з банківськими системами, торговельними фірмами або іншим спеціалізованим програмним забезпеченням Також на сайті компанії зазначається, що для оптимізації управління бізнес-процесами будуть використовуватися технології штучного інтелекту та алгоритми машинного навчання.

Хоча у відкритих джерелах детальна інформація про внутрішню архітектуру цього застосунку є досить обмеженою і офіційно не розкривається повний технологічний стек, аналіз аккаунтів працівників компанії явно показує, що серверна частина UGLA розроблена переважно на PHP. Тож роблячи висновок щодо архітектури додатку, стек технологій, описаних самою фірмою – хмарна інфраструктура AWS, підтримка PWA та наявність API чітко вказує на те, що UGLA є сучасним, комплексним хмарним сервісом. [7]

1.3 Odoo

Odoo – пакет програм різного функціоналу, які направлені на усі можливі потреби компаній; від бухгалтерського обліку до збирання інформації відгуків клієнтів. Odoo має своє ядро як у будь-якої операційної системи, система та додатки якої побудовані на мовах програмування Python та JavaScript.

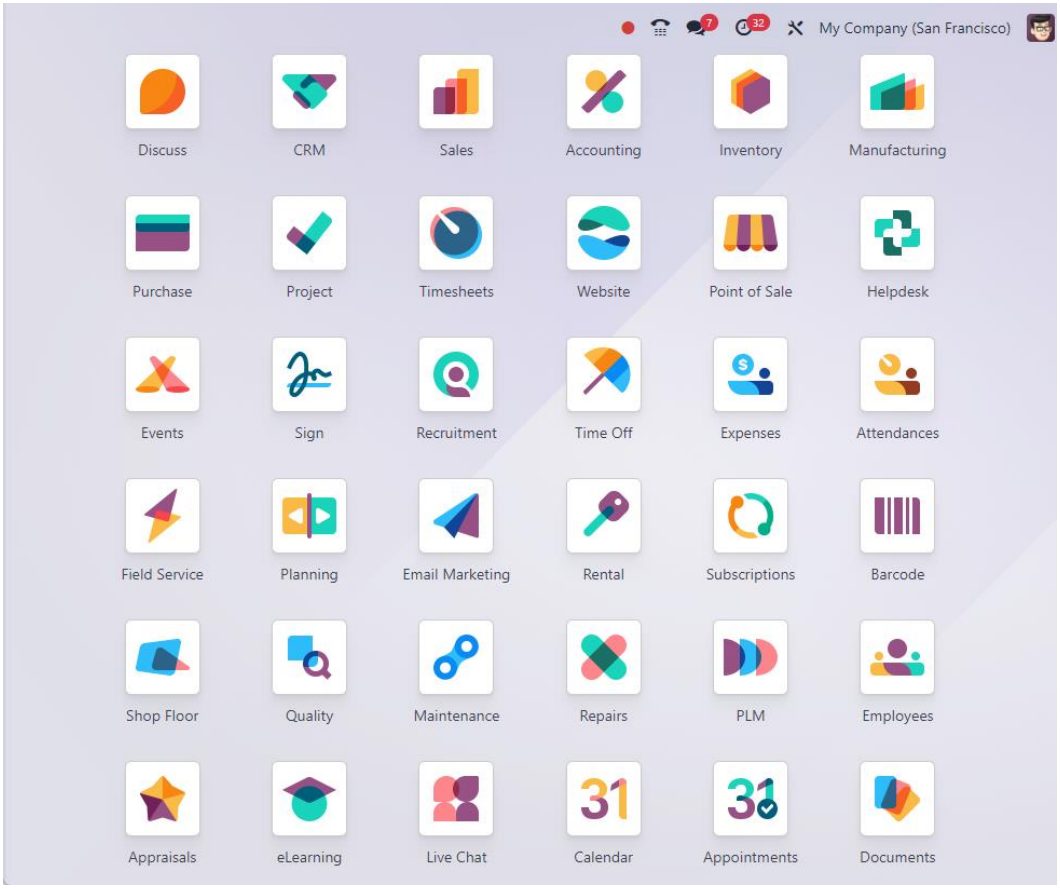


Рисунок 1.3 – Початкова сторінка Odoo

1.3.1 Функціонал та архітектура

Сама Odoo має необмежений потенціал керування бізнесом, яке підходить не тільки малим або середнім компаніям, а й великим бізнесам, які можуть запросити розробити компанію зробити підходящий функціонал саме для них. Якщо брати основний функціонал системи, то він зосереджений на таких аспектах як: управління взаємовідносинами з клієнтами та продажами – додаток **«CRM»** дозволяє ефективно відстежувати потенційних клієнтів на всіх етапах воронки продажів, автоматизувати комунікації та аналізувати ефективність команди. Фінансовий та бухгалтерський облік розкривається у додатку **«Бухгалтерія»**, що надає інструменти для ведення повного циклу облікових операцій, від обробки рахунків-фактур та банківських виписок до формування фінансової звітності. Для операційної діяльності не менш

важливими є ефективне управління інвентарем, де модуль **«Склад»** забезпечує точний облік товарів у реальному часі, управління переміщеннями та автоматизацію поповнення запасів. Крім того, Odoo дозволяє легко створювати та підтримувати веб-сайти й повноцінні інтернет-магазини за допомогою додатку **«Електронна комерція»**, який пропонує інтуїтивний конструктор та повну інтеграцію з продажами і складом, та багато інших програм.

Головними перевагами системи є тісна інтеграція всіх модулів, що забезпечує єдиний потік даних та усуває їх дублювання, а також можливість глибокої адаптації під унікальні бізнес-процеси завдяки широкому вибору готових додатків та гнучким налаштуванням. Odoo прагне бути єдиною платформою, що дозволяє компаніям будь-якого розміру ефективно керувати всіма своїми ресурсами та процесами через зручний веб-інтерфейс. [9]

Технічною основою Odoo є відкритий вихідний код (Open Source), що надає системі значну гнучкість та можливості для модифікації. Переважно система розроблена з використанням мови програмування Python. Її виразно модульна архітектура коду дозволяє легко розширювати функціонал, створювати власні додатки та адаптувати систему під унікальні бізнес-потреби, чому також сприяє наявність тисяч готових модулів від спільноти розробників.

1.3.2 Розробка додатків на Odoo

Також в Odoo є функціонал розробки власних додатків, яким також користуються компанії. Коли стандартний функціонал не повністю задовольняє специфічні потреби компанії, Odoo надає розробникам потужний інструментарій для створення власних модулів. Використовуючи мову Python, вбудований ORM для ефективної роботи з даними та чітко документований API, фахівці можуть проектувати та реалізовувати нові додатки. Ці кастомні модулі безшовно інтегруються з ядром системи та іншими стандартними

додатками, такими як **«Бухгалтерія»** чи **«Склад»**, успадковуючи загальну логіку та інтерфейс. [10]

1.4 Порівняння існуючих рішень

Задля порівняння наведених систем для зручності всі головні аспекти цих програм будуть виведені у таблицю, де буде відмічатись присутність або відсутність того чи іншого функціоналу.

Таблиця 1.1 – Порівняння існуючих аналогів

Характеристика	Perfectum	Ugla	Odoo
Система (Походження, Тип)	CRM+ERP система	CRM+ERP система	ERP-платформа з функціоналом CRM
Основний підхід до Кастомізації/ Розробки	No-Code конструктори (включаючи створення нових унікальних модулів користувачем без програмування).	No-Code інструменти для кастомізації процесів; задокументований API для інтеграцій.	Відкритий вихідний код (Open Source, Python); розробка власних додатків тисячі готових додатків від спільноти.
Ключові технології та інфраструктура	**Backend:** PHP **DB:** MariaDB **Інфраструктура:** Сервери на AlmaLinux/CentOS **Інтеграції:** ionCube, Pusher, Google, Amazon, Firebase, Dropbox.	**Backend:** PHP **DB:** MariaDB **Інфраструктура:** Сервери на AlmaLinux/CentOS **Інтеграції:** ionCube, Pusher, Google, Amazon, Firebase, Dropbox.	**Backend:** Python **Frontend:** HTML, CSS, JavaScript **DB:** PostgreSQL

Продовження таблиці 1.1

Характеристика	Perfectum	Ugla	Odoo
Основні аспекти безпеки	Орієнтація на стандарти ISO, шифрування файлів, модуль контролю доступу (IP, пристрої, сесії), політики паролів (складність, хешування), системний журнал.	Забезпечується високий рівень безпеки завдяки хмарній інфраструктурі AWS.	Працює на серверах Linux з останніми оновленнями, розміщення різних дата-центрів у різних регіонах світу

Підсумувавши результати порівняльної таблиці, можна зробити висновок що абсолютно усі системи ведення обліку даних та бізнес-процесів мають гарний захист, простий та нативний інтерфейс, можливість виведення графів для розуміння ситуації в компанії та її процесах, резервне копіювання даних. Саме це і стане в основі майбутнього додатку по темі.

ВИСНОВОК ДО РОЗДІЛУ 1

Сучасне програмне забезпечення для малого та середнього бізнесу створюється з чітким фокусом на спрощення повсякденних завдань, об'єднанні даних в одному місці та узгоджену роботу всіх бізнес-процесів. Як уже згадувалося раніше, головна мета таких систем – зробити роботу компанії більш ефективною, зменшити кількість помилок, забезпечити прозорість дій і допомогти керівництву приймати рішення, спираючись на точні й актуальні дані. Все це реалізується шляхом формування єдиного інформаційного середовища, де зберігається вся ключова інформація – від фінансів до замовлень, клієнтів і ресурсів.

Аналіз трьох наведених популярних систем Perfectum, UGLA та Odoo наочно показав, як ці завдання вирішуються на практиці, і водночас навів приклад сучасних підходів в дії. Серед помітних трендів – акцент на гнучкість і кастомізацію: наприклад, Perfectum і UGLA надають прості No-Code інструменти, якими може користуватись навіть не технічна людина. Натомість Odoo дозволяє глибоке налаштування для досвідчених розробників завдяки відкритому коду на Python. Важливим є також те, що ці системи використовують сучасні веб-технології, підтримують хмарну інфраструктуру або локальне розгортання та мають мобільний доступ (нативні додатки або PWA). Питання безпеки теж не залишаються поза увагою.

Порівняльний аналіз підкреслив спільні риси систем: інтуїтивно зрозумілий інтерфейс, зручні інструменти для створення звітів і візуалізації даних, можливість резервного копіювання, а також стабільний захист інформації. Отримані результати стали хорошим підґрунтям для створення власного додатку, який поєднуватиме перевірені підходи з орієнтацією на реальні виклики, що постають перед малим бізнесом.

РОЗДІЛ 2. ОГЛЯД ТЕХНОЛОГІЧНОГО СТЕКУ ДЛЯ РОЗРОБКИ СИСТЕМИ

2.1 Вибір мови програмування

При виборі мови програмування даної системи потрібно враховувати багато ключових факторів, головними з яких є безпека, тип і структура даних, масштабованість, стабільність підтримки та можливість візуалізації даних. Варіантом мови для розробки системи була вибрана мова програмування Java, яка відома своєю стресостійкістю до навколишніх негараздів, монолітною структурою та масою бібліотек та фреймворків для реалізації своєї кастомної системи керування бізнес-процесами.

2.1.1 Java

Мова програмування Java є однією з найбільш популярних мов програмування для серверної частини у багатьох фінансових питаннях (банки, системи регулювання ризиків, фінансові установи, державні інституції і т.п.), використання у веб-застосунках (серверна частина програми, REST-сервіси), мобільна розробка (додатки для абсолютно усіх пристроїв, на яких стоїть Android), Big-data програми (обробка великих массивів даних, їх експорт та імпорт). ЇЇ популярність зумовлена розробкою додатків на абсолютно різному спектрі: від великих застосунків, поділених на великі монолітні системи, до невеличких програм на смартфон по типу калькулятора, кроссплатформеність серед багатьох операційних систем, що дозволяє запустити додатки на багатьох платформах та обширною кількістю інструментів різного призначення. [11]

Рисунок 2.1 – Логотип Java

2.1.2 Плюси та мінуси Java

Головними плюсами Java є жорста типізація, при якій потрібно зазначати тип змінних незалежно від того чи зрозуміло що за тип даних буде внесений чи ні, запобігаючи помилки типізації та неправильного порівняння типів. Ткаож у ній присутнє автоматичне керування пам’яттю для звільнення ячейок з непотрібними даними, яке реалізоване за допомогою «колекціювання сміття» (garbage collector), та слідування принципам об’єктно-орієнтованності, що дозволяє створювати великі ланцюги наслідування класів до того моменту, доки не буде реалізований задуманий клас з усіма потрібними методами, що у свою чергу виливається у гнучкіть програмного забезпечення.

Напротивагу плюсам, мінуси в Java бувають різних ступенів. Java доволі повільно працює в порівнянні з деякими мовами програмування, що напряму впливає на відгук та швидкодію програми. Споживання пам’яті також є одним з недоліків, адже сама Java використовує автоматичне керування пам’яті та структурами даних, які не були початково прописані у Java Virtual Machine. Багато яиих мов не можуть похизуватись доступом до низькорівневих функцій комп’ютера, наприклад індексів або комірок пам’яті, на жаль Java не є

виключенням, це також напряму впливає на швидкодію програми адже доступу до реєстів та ручного розподілення пам'яті немає.

Хоча й здається, що Java може бути не цікавою та не практичною мовою програмування, насправді ситуація зворотня. Java є доволі дружньою мовою для новачків та багато професіоналів з цієї мови готові допомогти виправити проблеми або дати пораду.[12] Також багатий інстурментарій Java потребує автоматизації для збору та підкріплення модулів в одне ціле.

2.1.3 Maven

Maven – інструмент автоматизації збирання та управління проектами, що переважно представлені на Java від Apache. Він дозволяє збирати проекти, записуючи потрібні для роботи залежності, компілює та запускає додатки.

Рисунок 2.2 – Логотип Maven

Для того, щоб цей інструмент міг автоматизовувати всю роботу по збиранню додатка, потрібен один єдиний файл, в якому буде зберігатись усі конфігураційні дані – pom.xml. Після виконання команди mvn install, Maven читає pom-файл, завантажує залежності, компілює код та створює фінальний артефакт накшталт JAR. Після цього процесу програму можна буде без проблем запустити та перевірити на працездатність.

Автоматизація процесу збирання програми та відлагодження з компілюванням коду значно спрощують працю над Java-додатками та додатків на інших мовах, за що Maven і використовують. [13]

2.2 Вибір фреймворків та бібліотек

Java-розробка неможлива без цілого арсеналу інструментів, які суттєво спрощують створення додатків і роблять процес роботи ефективнішим. Бібліотеки займають у цьому ключове місце – вони дозволяють розв'язувати типові завдання без зайвого коду. Наприклад, Jackson чи Gson допомагають швидко працювати з JSON, Apache Commons пропонує десятки готових утиліт, а Guava суттєво розширює можливості роботи з колекціями. Логування подій, яке є важливою частиною будь-якого застосунку, зручно реалізовується за допомогою Log4j або SLF4j. Усе це дозволяє зосередитися не на «технічних дрібницях», а на побудові логіки, яка справді вирішує задачі бізнесу.

Ще потужнішим інструментом для розробника є фреймворки. Вони не просто допомагають, а значною мірою формують каркас застосунку, задають стиль і правила, а також економлять купу часу завдяки вже реалізованим рішенням. Класичні приклади – Jakarta EE для корпоративних систем чи Hibernate для роботи з базами даних. Останні роки з'являється дедалі більше легких, гнучких фреймворків на зразок Quarkus або Micronaut. Проте окремо варто згадати Spring – один із найвідоміших і найпотужніших фреймворків у світі Java. Спочатку він пропонував більш легку альтернативу традиційним Java EE-рішенням, але з часом виріс у велику екосистему. Сьогодні, завдяки таким проєктам, як Spring Boot, Spring Data та Spring Security, розробник може побудувати будь-який застосунок – від простого сайту до складної корпоративної системи – швидко, зручно та без втрати якості.

2.2.1 Загальний фреймворк Spring

Spring Framework став справжньою основою для сучасної Java-розробки, адже дозволяє будувати гнучкі, масштабовані та зрозумілі системи. Його

головною ідеєю являється інверсія управління, яка полягає в тому, що об’єкти створює спеціальний контейнер, а не сам розробник. Замість того, щоб компоненти залежали один від одного, Spring сам підставляє потрібні частини — це називається впровадженням залежностей. Такий підхід дозволяє створювати слабкозв’язані модулі, які легко підтримувати та тестувати. Додатково, через аспектно-орієнтоване програмування, можна винести загальні задачі (наприклад, логування чи безпеку) в окремі модулі, не зачіпаючи основний код.

Рисунок 2.3 – Логотип Spring Framework

Spring має модульну структуру, тож ти сам обираєш, що саме використовувати: Spring MVC – для вебінтерфейсів, Spring Data – для роботи з базами даних, Spring Security – для захисту даних. А Spring Boot зробив усе ще зручнішим: він прибирає зайву конфігурацію, дозволяє запускати застосунок як окремий JAR-файл і пропонує готові налаштування для найпоширеніших сценаріїв. Завдяки підтримці анотацій, код залишається компактним і зрозумілим. Уся ця екосистема, включно з Spring Cloud для мікросервісів, допомагає створювати надійні й сучасні Java-додатки швидше й легше, ніж будь-коли. [14]

2.2.2 Spring Boot

Spring Boot є одним із найпопулярніших інструментів у Java-екосистемі, який суттєво полегшує створення додатків на основі Spring. Його мета – забезпечити автоматичне налаштування компонентів та надати зручний старт для розробки, дозволяючи одразу зосередитися на розв’язанні прикладних бізнес-завдань, не витрачаючи час на налаштування інфраструктури. Завдяки Spring Boot, створення повноцінного застосунку більше не вимагає складних файлів конфігурації чи великої кількості додаткового коду.

Основою функціонування Spring Boot є механізм автоконфігурації, який автоматично визначає необхідні налаштування залежно від підключених бібліотек. Також платформа використовує так звані starter-залежності – готові комплекти для поширених сценаріїв використання, таких як веб-розробка, безпека чи робота з базами даних. Також окремою перевагою є можливість запуску застосунків у вигляді JAR-файлів із вбудованим сервером, що спрощує процес розгортання та добре підходить для мікросервісної архітектури.

Окрім зручного старту, Spring Boot пропонує низку можливостей для ефективного керування додатками в робочому середовищі. Зокрема, модуль Spring Boot Actuator забезпечує доступ до внутрішніх показників системи, таких як стан здоров’я застосунку, метрики продуктивності чи параметри конфігурації. Гнучкий механізм зовнішньої конфігурації через .properties, .yml або змінні середовища дозволяє легко адаптувати поведінку програми до різних середовищ — розробки, тестування чи продакшну. Підсумувавши, Spring Boot є універсальним рішенням для побудови сучасних, масштабованих і підтримуваних Java-додатків. [15]

2.2.3 Spring Web MVC

Серед інструментів для розробки веб-додатків на Java Spring Web MVC посідає особливе місце завдяки своїй гнучкості, простоті інтеграції та підтримці сучасних підходів. Він базується на шаблоні Model-View-Controller (MVC), який забезпечує поділ логіки на три складові:

1. Model – відповідає за дані;
2. View – за їх відображення;
3. Контролер – обробляє запити та керує логікою.

Зв'язуючим елементом у Spring MVC є DispatcherServlet – фронт-контролер, який приймає всі HTTP-запити й передає їх відповідним обробникам. За допомогою таких анотацій, як @Controller, @RestController, @GetMapping, @PostMapping, @RequestParam, @PathVariable, @RequestBody, можна швидко та зручно описати маршрути та обробку запитів. Для REST API відповіді автоматично формуються у JSON або XML через вбудовані перетворювачі. [16]

Spring MVC також підтримує різні системи шаблонізації (JSP, Thymeleaf, FreeMarker) і легко інтегрується з іншими модулями, такими як Spring Security чи Spring Data. А завдяки вже зазначеному Spring Boot конфігурація спрощується до мінімуму: більшість налаштувань виконується автоматично. Найвиразнішими перевагами Spring MVC є чітка структура та модульність, гнучке оброблення веб-запитів, зручна робота з даними (введення, валідація, форматування) та легка інтеграція з іншими інструментами Spring. У поєднанні з Spring Boot це рішення дозволяє швидко створювати сучасні, масштабовані веб-додатки та REST API з мінімальними зусиллями. [17]

2.2.4 Spring Security

Spring Security – це ще один значимий фреймворк у світі Spring, який захищає ваш застосунок. Він відповідає за дві головні речі: автентифікацію і авторизацію користувачів додатку

Одна з найзручніших його фішок — це автоматичне налаштування, яке працює разом зі Spring Boot. Якщо просто додали залежність spring-boot-starter-security, Spring сам вмикає базовий захист усіх сторінок вашого сайту. За замовчуванням користувачу буде показано стандартну форму входу, а для зміни форми доступне підключення кастомних HTML-сторінок.

Звісно, у реальних проектах часто потрібен гнучкіший контроль. І тут Spring Security дозволяє все налаштувати вручну: які сторінки відкриті, які вимагають логіну, куди перенаправляти після входу, як виглядає форма авторизації, як обробляється вихід. Для цього використовується спеціальний компонент SecurityFilterChain, через який легко керувати правилами безпеки у коді.

За замовчуванням користувачів можна описати просто в пам'яті (in-memory), прямо у коді, призначивши їм імена, паролі (зашифровані) та ролі. Але якщо потрібно, то можна легко підключити базу даних або зовнішні сервіси, як-от Google або Facebook, для авторизації. Загалом можна відзначити такі його плюси, як:

1. Обмеження доступу до окремих методів у коді через зручні анотації;
2. Вбудований захист від атак на кшталт впровадження шкідливого скрипту у поля додатку;
3. Робота з сучасними протоколами авторизації (OAuth2, JWT, тощо);
4. Засоби для безпечного зберігання паролів і роботи з шифруванням.

Незважаючи на велику кількість функцій, Spring Security добре інтегрується зі Spring Boot і не вимагає багато зусиль на старті. Цей фреймворк

дозволяє побудувати як простий захист, так і складну багаторівневу систему безпеки, залежно від того, що саме потрібно для бажаного застосунку. [18]

2.2.5 Spring Data Jpa

Spring Data JPA являє собою один з багатьох модулів Spring Framework, створений для полегшення роботи із СУБД у застосунках, що базуються на JPA. Основна мета цього модуля полягає у мінімізації шаблонного коду: щоб отримати повноцінний CRUD-функціонал, достатньо оголосити інтерфейс-репозиторій, який успадковує, наприклад, JpaRepository. Під час запуску Spring автоматично згенерує потрібну реалізацію, тим самим скоротивши обсяг коду, підвищивши читабельність і забезпечивши єдиний стиль доступу до даних. Крім того, модуль органічно інтегрується з транзакційною підсистемою Spring і переводить специфічні JPA-винятки в уніфіковану ієрархію DataAccessException, що спрощує обробку помилок.

Фундаментом слугує Java Persistence API – стандартна специфікація ORM у Java, яка розглядає таблиці БД як звичайні Java -сутності (@Entity). Оскільки JPA є лише набором інтерфейсів, для фактичної роботи потрібен провайдер (найуживаніший – Hibernate). JPA також пропонує мову запитів Jakarta Persistence Query Language, подібну до SQL, але орієнтовану на об'єкти, а не на таблиці.

У середовищі Spring Boot підключення стартеру spring-boot-starter-data-jpa та налаштування DataSource практично вичерпує початкові кроки. Далі створюються сутності та інтерфейси-репозиторії, а складніші запити будує або через «розумні» назви методів (findByNameAndStatus), або за допомогою анотації @Query, де можна прописати нативний SQL. [19]

2.2.6 Lombok

Project Lombok – це анотаційна бібліотека, що інтегрується у фазу компіляції й автоматично додає у байт-код стандартні елементи на кшталт геттерів, сетерів, методів toString(), equals(), hashCode() та необхідних конструкторів. Завдяки цьому вихідні файли залишаються лаконічними, а розробник звільняється від повторюваної «ручної» роботи. Підхід не потребує жодних рантайм-залежностей: після компіляції клас містить звичайний Java-код, що спрощує подальшу інтеграцію та аналіз.

Рисунок 2.4 – Логотип Lombok

Найчастіше застосовують узагальнену анотацію @Data, яка одразу формує комплект геттерів, сетерів і базових методів. Для вибіркових ситуацій використовують @Getter та @Setter, а генерацію конструкторів забезпечують @NoArgsConstructor, @RequiredArgsConstructor і @AllArgsConstructor. Додаткові анотації дають змогу розширити модель класу: @Builder створює патерн Builder для зручного конструювання об’єктів, а @Value оголошує незмінний (immutable) клас, роблячи всі поля final і доступними лише для читання.

Lombok також надає утилітарні можливості: @NonNull додає перевірки на null, @Cleanup автоматично закриває ресурси через блок try–finally, а логерні анотації (@Slf4j, @Log4j2 тощо) без зайвої конфігурації ін'єктують готовий об'єкт для журналювання. За потреби приховано кидати перевірені винятки

можна скористатися @SneakyThrows, тоді як @Synchronized генерує потокобезпечні блоки без явного виклику synchronized. [20]

Немаловажною утилітою для статичного аналізу чи сумісності з інструментами, що не обробляють анотації, слугує delombok, яка розгортає вихідний код, замінюючи анотації на повноцінні методи. У підсумку Lombok істотно підвищує продуктивність і підтримуваність проєкту, дозволяючи зосередитися на бізнес-логіці, а не на повторюваних технічних конструкціях.

2.3 База даних

Бази даних є невід'ємною складовою сучасних інформаційних систем, забезпечуючи збереження, обробку й доступ до даних, необхідних для роботи програмних продуктів. Залежно від потреб проєкту, існують два основні типи СУБД – реляційними (SQL) та нереляційними (NoSQL).

NoSQL-технології виникли як відповідь на виклики масштабованості й обробки неструктурованих даних, характерні для великих розподілених систем. Такі бази, як MongoDB, Redis та інші, краще справляються з великими обсягами динамічної інформації та загалом орієнтовані на жорстій структуризації, але при цьому часто поступаються в питанні узгодженості та транзакційної цілісності.

На противагу нереляційним, реляційні бази даних (наприклад, MySQL, PostgreSQL, Oracle) забезпечують стабільність, структурованість та суворе дотримання цілісності даних завдяки підтримці прямих зв'язків з даними. Вони використовують таблиці зі схемами, підтримують складні запити мовою SQL і мають розвинену екосистему засобів аналітики, резервного копіювання й адміністрування. [21]

У контексті створення системи для обліку та підтримки бізнес-процесів було прийнято рішення на користь реляційної СУБД. Це обумовлено чіткою структурованістю даних (замовлення, транзакції, товари, користувачі) та наявністю складних взаємозв’язків між ними. Ще варто зазначити, що такі системи гарантують високу надійність, злагодженість операцій та можливість виконання складних запитів, що є критично важливим для корпоративного обліку.

2.3.1 PostgreSQL

На сучасному ринку систем керування базами даних доволі високо розташована реляційна СУБД під назвою PostgreSQL. Це потужна об'єктно-реляційна система з відкритим кодом, яка поєднує в собі своєрідний багатий функціонал та базується на SQL. Завдяки понад 35-річному розвитку, PostgreSQL здобула репутацію стабільної та функціонально багатої платформи для зберігання та обробки даних.

Рисунок 2.5 – Логотип PostgreSQL

PostgreSQL варта розмови через її відповідність стандарту SQL та підтримка ACID-властивостей, що гарантує цілісність і надійність транзакцій. Система також відзначається високою розширюваністю: користувачі можуть створювати власні типи даних, функції та навіть використовувати різні мови програмування без необхідності перекомпіляції бази даних [22].

Також варто навести інформацію про те, що сама система має широку документацію, та як вже було вказано раніше, має відкриту кодову базу, розташовану на GitHub.

2.3.2 pgAdmin

PostgreSQL можна використовувати на будь-якій комп'ютерній системі, яка має відповідні технології та CLI для доступу до даних, але задля зручності адміністрування баз даних на цій системі була створена така утиліта як pgAdmin. Вона дозволяє керувати базами даних через інтерфейс та також має інструмент для роботи з БД у вигляді CLI у одному з діалогових вікон.

Рисунок 2.6 – Логотип pgAdmin

Вбудований редактор для написання та виконання SQL-запитів розширює функціонал адміністрування, підтримучи підсвічування синтаксису, автодоповнення та дозволяючи переглядати результати запитів у зручному табличному форматі. Також pgAdmin надає можливість створення та редагування базових об'єктів бази даних, таких як таблиці, індекси, тригери та функції, без необхідності писати SQL-код вручну за допомогою відкриття діалогового вікна. [23]

Для забезпечення безпеки та контролю доступу, pgAdmin дозволяє керувати ролями користувачів та їхніми правами. Адміністратори можуть створювати нові ролі, призначати їм права доступу до конкретних об'єктів бази даних та налаштовувати параметри автентифікації. Утиліта підтримує функції

резервного копіювання та відновлення баз даних, що дозволяє зберігати дані для критичних ситуацій та швидко відновлювати їх у разі потреби. Інструмент також надає можливості для моніторингу продуктивності бази даних, включаючи перегляд активних процесів, статистики запитів та використання ресурсів. [24]

2.3.3 Інтеграція PostgreSQL з Spring

Для підключення PostgreSQL до Spring Framework необхідно занести його у конфігураційний файл pom.xml від Maven при збірці додатку.

Детальний гайд виглядає таким чином:

1. Потрібно створити Spring Boot проект та вибрати потрібні вам бібліотеки та фреймворки;
2. Додати залежності у файл pom.xml або переконатись, що вони внесені у конфігурацію
3. Налаштувати підключення до бази даних у конфігураційному файлі самого додатку: назву СУБД, користувача та пароль
4. Написати відповідну вашим потребам програму та запустити її: Spring Boot автоматично підключиться до PostgreSQL та у логах буде виведене успішне під'єднання до БД [25]

ВИСНОВОК ДО РОЗДІЛУ 2

Усі раніше наведені мови програмування, інструменти збору проекту та відладки коду, фреймворки та допоміжні бібліотеки в значному обсязі будуть використовуватись у подальшій розробці застосунку. Зібраний технологічний стек буде основою додатку, його безпеки та функціональності для усіх запланованих модулів застосунку.

Якщо взяти вибір мов програмування, також гарними кандидатами є такі мови як JavaScript, C#, Go та інші. В порівнянні з ними Java виокремлюється дуже багатим функціоналом на різні випадки при розробці та кількість нерозв'язаних задач за усі роки існування цієї мови є настільки мало, що можна вважати, що написання коду та помилки всередині є напряму пов'язані від навичок програмування розробника.

Підсумовуючи використання фреймворків та бібліотек, Spring Framework буде основним інструментом для розробки сервісної частини додатку з його модулями Spring Security, Spring Data JPA та інші. Гарною альтернативою також є Java Servlet API, в якому усі відповіді на запити проходять через спеціалізований клас – сервлет. Також для спрощення написання громіздких та повторюваних функцій буде допомагати Project Lombok, в якому усе скоротиться до написання анотацій над класами.

Між вибором СУБД було прийнято рішення розроблювати систему на PostgreSQL, так як усі дані будуть заноситись вручну та вони будуть підпорядковані, тож використанням реляційної та нереляційної баз даних перевагу було надано реляційним. Стабільним аналогом PostgreSQL виступає MySQL, яка була створена засновниками мови програмування Java – Oracle Corp. Але через неможливість зберігати об'єкти в якості даних у MySQL та нестабільним використанням ACID-властивостей, PostgreSQL стала основною реляційною БД для додатку.

РОЗДІЛ 3. АРХІТЕКТУРА ТА РОЗРОБКА СИСТЕМИ

3.1 Загальні відомості

В основі архітектури розробленої системи лежать перевірені принципи багаторівневого підходу (Layered Architecture). Така архітектура дозволяє чітко розділити всю логіку програми на окремі, незалежні функціональні блоки та робить систему гнучкішою, простішою в обслуговуванні та розширенні, а також дає змогу ефективніше контролювати її роботу та доступ до інформації.

Сама система побудована за класичною трирівневою моделлю. Вона складається з рівня зберігання даних (database), рівня серверної логіки (back-end, де відбуваються основні обчислення та операції з внесеними даними) та рівня представлення (front-end, з яким працює користувач). Кожен з цих рівнів має чітко визначені завдання та взаємодіє з іншими через зрозумілі й надійні програмні інтерфейси, що гарантує їх злагоджену роботу.

Рівень зберігання даних функціонує на базі реляційної системи управління базами даних PostgreSQL. Інформація тут зберігається у вигляді впорядкованих таблиць, які відповідають ключовим бізнес-об'єктам (наприклад, клієнтам, замовленням, товарам). Щоб дані залишалися цілісними та узгодженими, таблиці пов'язані між собою спеціальними ключами, що також дає можливість формувати складні запити для аналізу інформації.

Серверна частина (back-end), що є центральним обчислювальним ядром системи, побудована за принципом розподілу відповідальностей. Для цього використано підхід "Controller – Service – Repository".

Контролери (Controller) виконують роль диспетчерів: приймають запити від клієнтського інтерфейсу та скеровують їх для подальшої обробки.

Сервіси (Service) інкапсулюють основну бізнес-логіку: тут реалізовані правила функціонування системи та виконуються ключові операції.

Репозиторії (Repository) забезпечують взаємодію з базою даних. Технологія JPA дозволяє працювати з даними як з програмними об'єктами, абстрагуючись від технічних деталей SQL-запитів. Такий розподіл відповідальностей підвищує зрозумілість коду, гнучкість системи та спрощує її подальше доопрацювання.

Клієнтський рівень (front-end) – це те, що бачить і з чим безпосередньо працює користувач. Він створений на основі HTML-шаблонів. Завдяки шаблонізатору Thymeleaf в Spring, ці сторінки динамічно наповнюються актуальними даними, які надходять із серверної частини. Для забезпечення зручності та інтерактивності системи у деяких шаблонах використовується JavaScript, допомагаючи перевіряти введені дані, оновлювати частини сторінки без її повного перезавантаження та обробляти дії користувача за допомогою REST-функціональності. Для наочного представлення одного з розділів сайту (аналітика) інтегровано бібліотеку Chart.js, що дозволяє показувати дані у вигляді графіків та діаграм прямо на веб-сторінках, роблячи складну інформацію простішою для сприйняття.

Обрана архітектура найбільш підходить під спосіб створення даного проекту. Вона дозволяє легко додавати нові можливості, забезпечує стабільну роботу системи навіть при значних навантаженнях, а також спрощує її підтримку та розвиток у майбутньому.

3.2 Рівень бази даних

Як вже було зазначено в розділі 2, в цій системі використовується СУБД PostgreSQL. PostgreSQL є найпопулярнішою реляційною СУБД для використання у системах зі збереженням даних.

Структура бази даних спроектована таким чином, щоб точно відображати логіку внутрішніх процесів, які підтримує система. Кожна ключова сутність предметної області, така як клієнт, замовлення, товар, склад, фінансова транзакція, була представлена відповідною таблицею. Між цими таблицями встановлені реляційні зв'язки, що моделюють реальні взаємозв'язки між об'єктами: наприклад, кожне замовлення асоціюється з конкретним клієнтом, а товар – з визначеним складом. Така архітектура даних забезпечує їх цілісність, мінімізує дублювання інформації та оптимізує виконання запитів до бази.

База даних в цілому функціонує як централізоване сховище, до якого звертається серверна частина системи при виконанні будь-яких операцій з даними: збереження, оновлення, видалення або їх отримання. Фундаментальним є те, що вся бізнес-логіка системи залежить від цієї структури даних. Відповідно, будь-яка дія користувача в інтерфейсі ініціює операції читання або модифікації даних у базі.

3.2.1 Структура бази даних

Структура бази даних у цій системі розроблена спеціально для охоплення всіх основних об’єктів, з якими може працювати малий бізнес у щоденній роботі. В її основі чіткий поділ на таблиці, кожна з яких відповідає окремій логічній одиниці: користувач, компанія, клієнт, товар, замовлення, платіж тощо.

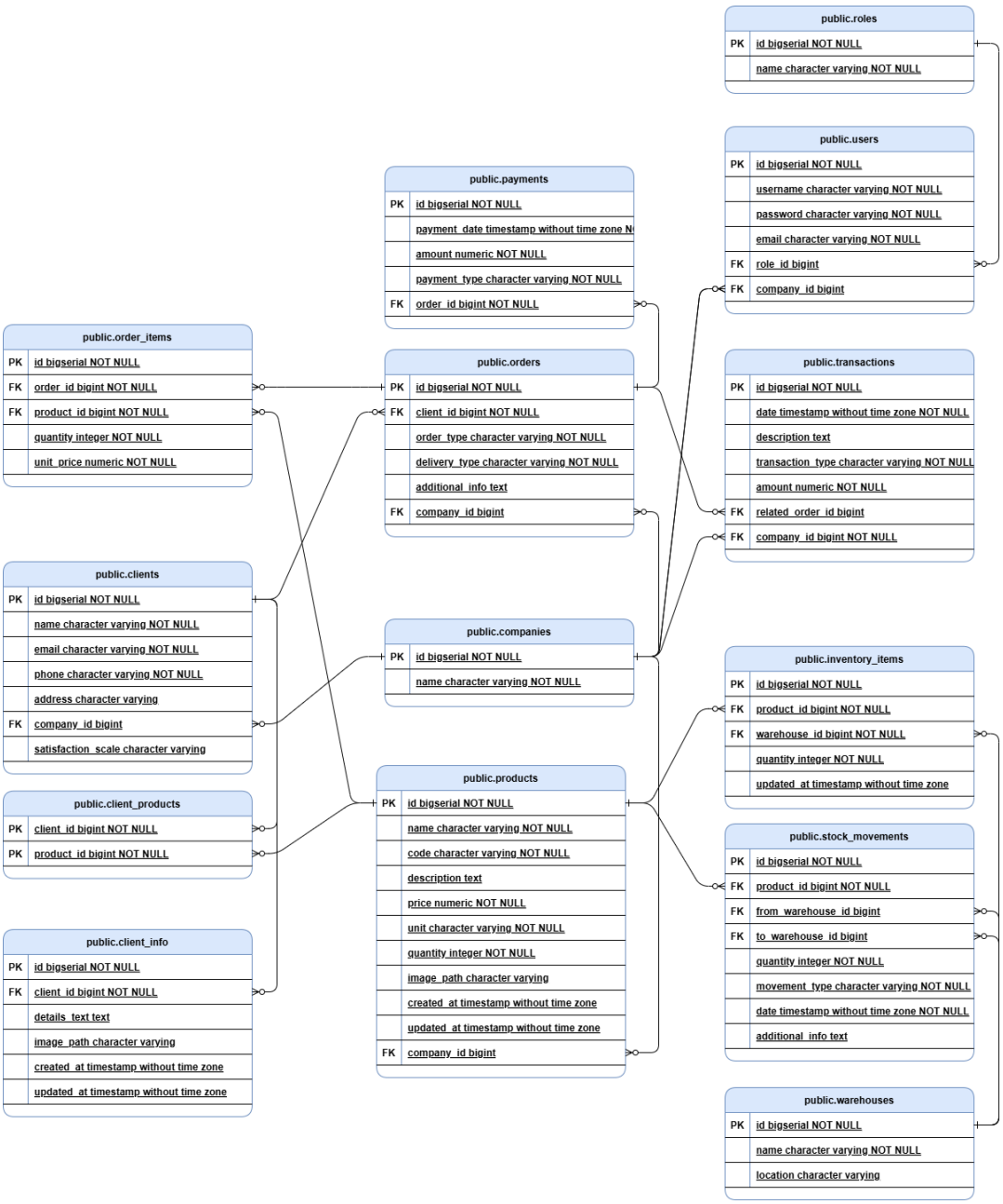


Рисунок 3.1 – Модель бази даних

Користувачі системи зберігаються в таблиці users. Вони мають логін, зашифрований пароль і роль - адміністратор, бухгалтер та простий працівник. Ролі винесені окремо в таблицю roles, щоб у майбутньому можна було легко додати нові або змінити права доступу. Також roles використовують enum у самій БД roletypes, де і збережені типи ролей.

Оскільки система може обслуговувати декілька компаній, є таблиця companies, яка описує організацію – її назву, контактну інформацію, а також пов’язує користувачів із конкретною компанією. Якщо ж користувач не належить компанії або не вказав її при реєстрації, у таблицю вноситься id компанії зі значенням null.

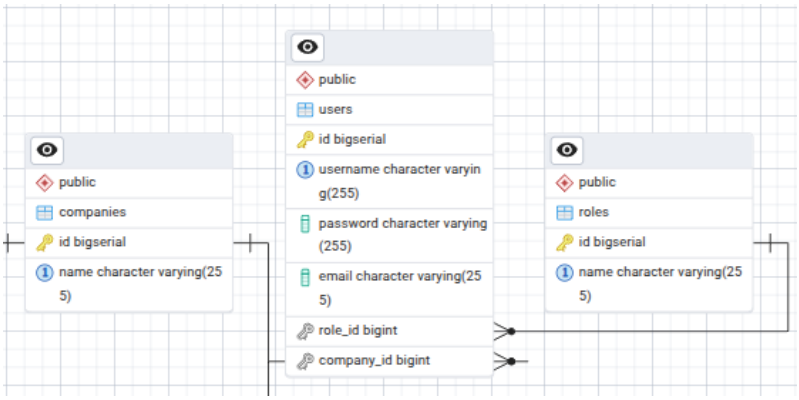


Рисунок 3.2 – Таблиці companies, users, roles

Клієнти, з якими працює бізнес, зберігаються в таблиці clients. Це основні контактні дані: ім’я, телефон, email, а також рівень задоволеності. Додаткові деталі про клієнтів винесено в таблицю client_info, щоби не перевантажувати основну структуру – туди входять коментарі, дати, нотатки тощо.

Товари, які обліковуються на складах, містяться в таблиці products. Вони мають назву, опис, ціну, а також додаткові технічні характеристики, якщо потрібно.

Для складів передбачено окрему таблицю warehouses, де зберігається базова інформація: назва, місцезнаходження тощо. Інформація про те, скільки товару є на кожному складі, зберігається в inventory_items. Саме ця таблиця дозволяє побачити залишки, не змінюючи дані в товарах чи складах напряму. А всі зміни – надходження, переміщення між складами або списання – фіксуються окремо в stock_movements.

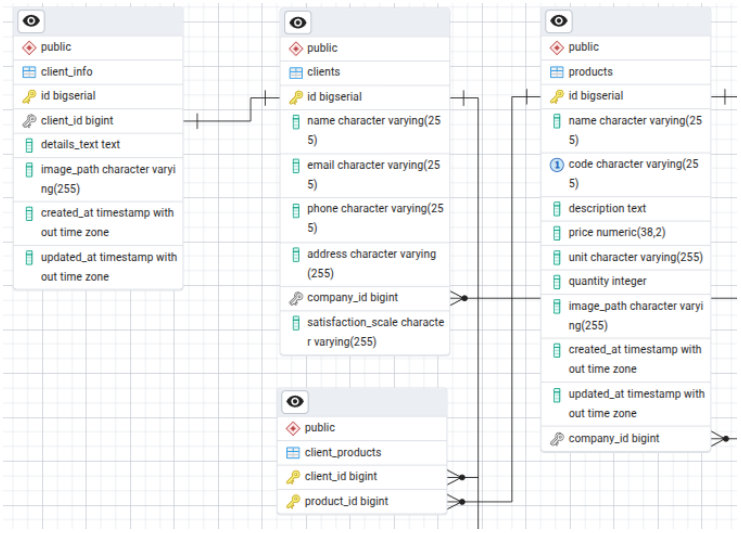


Рисунок 3.3 – Таблиці clients, client_info, products, client_products

Замовлення клієнтів описані в таблиці orders. Тут фіксується, хто і коли зробив замовлення, яка сума, статус виконання тощо. Сам склад кожного замовлення -- тобто перелік товарів і їх кількість – винесено в таблицю order_items.

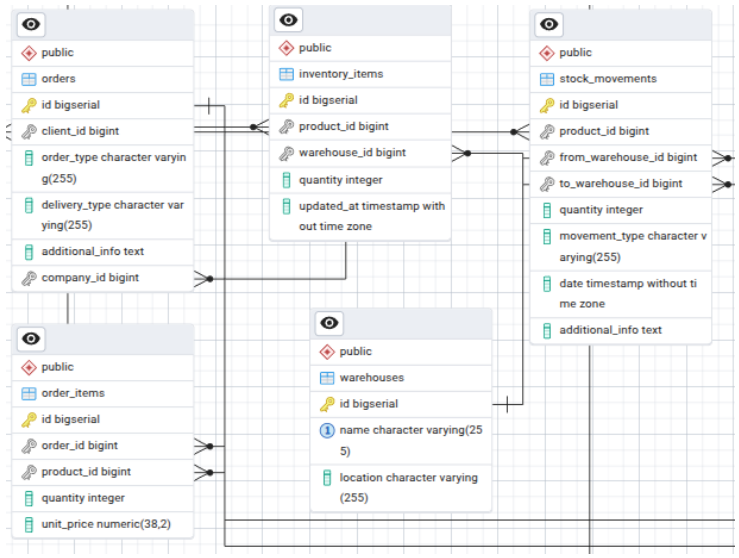


Рисунок 3.4 – Таблиці orders, order_items, inventory_items, warehouses, stock_movements

Також є довідники для типів доставки delivery_types і типів замовлень order_types, які використовуються для фільтрації, аналітики та бізнес-логіки.

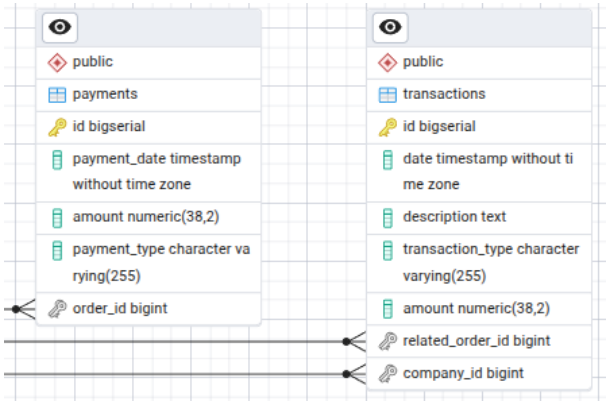


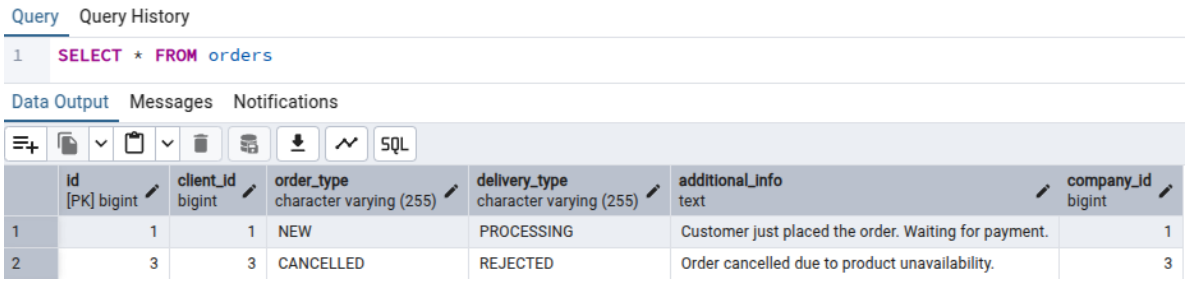
Рисунок 3.5 – Таблиці transactions та payments

Фінансовий блок представлений двома таблицями. У transactions зберігаються загальні записи про прибутки та витрати компанії — з описом, сумою, типом і датою. У payments — інформація про окремі платежі, прив’язані до конкретних замовлень.

3.2.2 Зв’язки між таблицями

Усі таблиці так чи інакше пов’язані між собою, то ж відносини між окремими таблицями є також невід’ємною частиною моделювання логіки БД.

Основним прикладом таких зв’язків є співвідношення між клієнтами та замовленнями: кожне замовлення (orders) має посилання на клієнта (clients), який його створив. Подібним чином таблиця order_items зберігає деталізацію замовлення, пов’язуючи кожен запис із конкретним замовленням і товаром.



	id [PK] bigint	client_id bigint	order_type character varying (255)	delivery_type character varying (255)	additional_info text	company_id bigint
1	1	1	NEW	PROCESSING	Customer just placed the order. Waiting for payment.	1
2	3	3	CANCELLED	REJECTED	Order cancelled due to product unavailability.	3

Рисунок 3.6 – Таблиця orders у pgAdmin4

На рисунку вище видно, що таблиця orders має company_id. В цьому плані запозичені id для таблиці вважаються зовнішніми ключами (FK, Foreign Key), а основні id - первинними (PK, Primary Key).

Схожим чином реалізовані зв'язки між складами та товарами: таблиця inventory_items відображає, скільки одиниць певного товару зберігається на конкретному складі, поєднуючи products і warehouses. А таблиця stock_movements дозволяє фіксувати переміщення товарів між складами або їх надходження/списання, маючи посилання як на товар, так і на склади, між якими відбувається рух.

Користувачі (users) пов'язані з таблицею roles – для визначення прав доступу – а також із companies, щоб закріпити їхню належність до певної організації. Водночас самі компанії можуть мати багато клієнтів, замовлень і транзакцій, тому більшість ключових таблиць також містить зв'язок із companies.

У фінансовому модулі таблиця payments пов'язана з orders, вказуючи, за яке саме замовлення був здійснений платіж. Таблиця transactions, хоч і працює незалежно від конкретних замовлень, може містити зв'язок із orders у випадку, якщо витрата або дохід напряму стосується певного замовлення.

У самій БД також наведено приклад проміжної таблиці client_producs – ця таблиця слугує для зв'язку між клієнтом та товаром, який він вибирає. Такі таблиці також є прикладом для уникнення дублювання полів та зберігання зв'язків окремо.

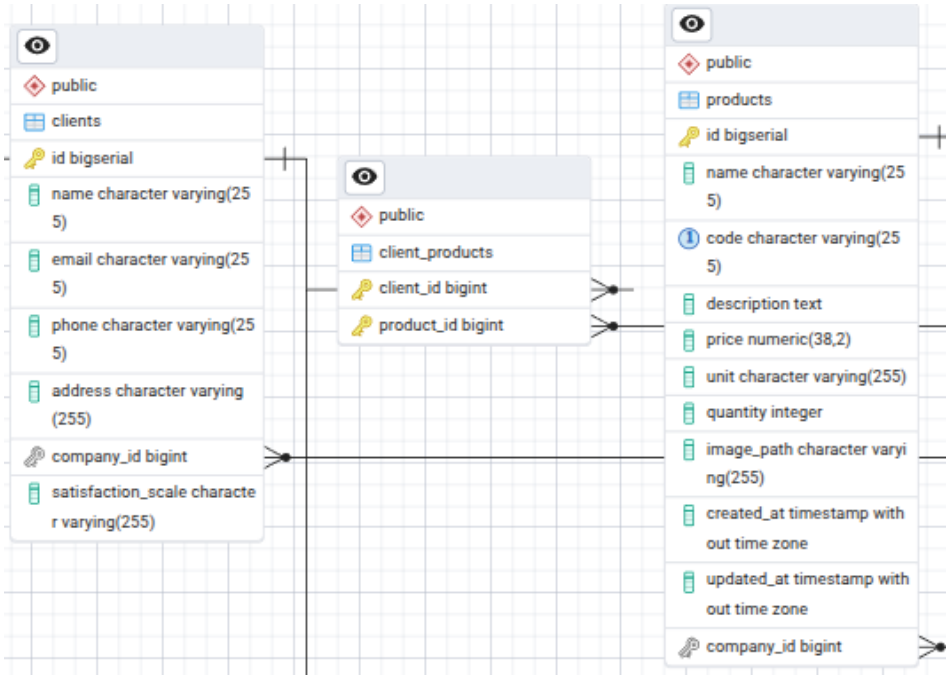


Рисунок 3.7 – Зв’язок таблиць products та clients через client_products

3.2.3 Під'єднання до Spring

Для підключення бази даних до серверної частини програми використовується стандартний механізм, наданий фреймворком Spring Boot. Ключові параметри з'єднання, такі як тип JDBC-драйвера, URL-адреса бази даних, ім'я користувача та пароль, конфігуруються у файлі application.properties або application.yml. Spring Boot при правильному налаштуванні автоматично ініціалізує пул з'єднань та встановлює підключення до БД.

Після успішного налаштування конфігурації, Spring Boot бере на себе управління життєвим циклом транзакцій та забезпечує доступ до таблиць бази даних через механізм репозиторіїв. Для організації взаємодії з конкретною таблицею достатньо створити відповідний інтерфейс репозиторію, а бізнес-логіку, пов'язану з обробкою даних цієї таблиці, реалізувати у відповідному сервісному компоненті.

Окремо варто відзначити, що транзакції можна оголошувати просто через анотацію @Transactional, без зайвого коду. Це дозволяє чітко контролювати, які методи мають виконуватись як одна логічна операція. Також використання Spring Data JPA анотацій для реалізації шару репозиторіїв значно скорочує обсяг коду, необхідного для виконання типових операцій з даними, адже фреймворк може автоматично генерувати реалізації багатьох методів доступу до даних на основі імен методів в інтерфейсах репозиторіїв або за допомогою анотації @Query.

3.3 Серверний рівень

Серверний рівень повністю реалізований за допомогою фреймворку Spring Boot і використовується виключно у цьому середовищі. Він відповідає за виконання бізнес-логіки, взаємодію з базою даних та обробку запитів від клієнтської частини. Структура побудована за принципом поділу на контролери, сервіси та репозиторії, забезпечуючи зрозумілу архітектуру та спрощуючи подальший розвиток системи.

3.3.1 Сутності

Сутності являють собою основу всієї логіки роботи з даними у серверній частині системи. У даному проєкті кожна сутність є Java-класом, який відображає таблицю в базі даних. Усі сутності використовуються у межах Spring Boot із JPA, дозволяючи автоматично зв’язувати поля класів із відповідними колонками таблиць. Таким чином, доступ до записів у базі здійснюється не через SQL-запити, а безпосередньо через об’єкти.

Кожна сутність позначається анотацією @Entity, а назва таблиці явно задається через @Table(name = "..."). Поля сутності, які мають зберігатися в

базі, описуються через @Column, а первинні ключі — через @Id та @GeneratedValue, найчастіше з параметром strategy = GenerationType.IDENTITY, що відповідає автоінкременту.

Для організації зв’язків між таблицями застосовуються такі анотації, як:

1. @ManyToOne представляє зв’язок «багато до одного», наприклад, багато клієнтів належать до однієї компанії;
2. @OneToMany є зв’язком «один до багатьох» – одна компанія має багато клієнтів;
3. @OneToOne – прямий зв’язок «один до одного», для прикладу, клієнт і його додаткова інформація;
4. @ManyToMany є двостороннім зв’язком, який реалізується через проміжну таблицю, наприклад, як вже було зазначено, зв’язок клієнтів із продуктами через таблицю client_products;
5. Анотація @JoinColumn вказує, яка саме колонка є зовнішнім ключем;
6. Анотація @JoinTable використовується для побудови проміжної таблиці у випадку @ManyToMany.

Також у самих класах є і допоміжні анотації валідації – @NotBlank, @NotNull, @Email, які перевіряють правильність введених даних ще до збереження; @CreationTimestamp і @UpdateTimestamp автоматично фіксують час створення або останнього оновлення об’єкта.

Для повного розгляду сутностей буде приведена сутність Product. Вона являється однією з сутностей у даному проекті, пов’язуючись з реальними товарами. Product описує такі дані, як назву, унікальний код, опис, ціну, одиницю виміру, залишок на складах та зображення самого продукту.

```java
@Entity
@Table(name = "products")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false, unique = true)
    private String code;

    @Column(columnDefinition = "TEXT")
    private String description;

    @Column(nullable = false)
    private BigDecimal price;

    @Column(nullable = false)
    private String unit;

    @Column(nullable = false)
    private Integer quantity;

    private String imagePath;

    @ManyToMany(mappedBy = "products")
    private List<Client> clients;

    @CreationTimestamp
    @Column(updatable = false)
    private LocalDateTime createdAt;

    @UpdateTimestamp
    private LocalDateTime updatedAt;

    @ManyToOne
    @JoinColumn(name = "company_id", nullable = false)
    private Company company;
```

Рисунок 3.8 – Java-клас сутності Product

Розібравши цей клас, можна сказати що в ньому присутні майже всі потрібні анотації, що використовуються у інших сутностях.

@Entity позначає для Spring Boot, що клас є сутністю, @Table прив’язує його до таблички “products”, @Id та @GeneratedValue відповідають за автоматичне створення унікального ID, @Column(nullable = false) зобов’язує заповнювати поле, @ManyToOne та @ManyToMany вказує на зв’язки з іншими таблицями,@CreationTimeStamp та @UpdateTimeStamp фіксують час створення та модифікації.

Інші сутності в системі – такі як Client, Order, User, InventoryItem, Transaction, Payment – побудовані за тією ж логікою: вони містять набір полів,

які відображають структуру відповідної таблиці, та містять зв'язки з іншими об'єктами системи.

3.3.2 Рівень репозиторіїв

Репозиторії в системі відповідають за взаємодію з базою даних і реалізовані як інтерфейси, що наслідують JpaRepository<T, ID>. Це стандартний інструмент у Spring Data JPA, який забезпечує повний набір CRUD-операцій без потреби в написанні SQL-запитів. Анотація @Repository дозволяє Spring автоматично виявити ці інтерфейси, обробляти винятки на рівні БД та використовувати їх як компоненти.

Окрім вбудованих методів, у дані інтерфейси було імплементовано декільна нових методів для корректної роботи застосунку – наприклад, findByEmail(), existsByPhone(), findByCompanyId() тощо.

Прикладом у даному рівні сервісного застосунку буде інтерфейс ClientRepo, який використовує звичайні методи з JpaRepository та імплементує власні медоти для подальшого застосування у сервісах.

```
@Repository  6 usages  romalytv *
public interface ClientRepo extends JpaRepository<Client, Long> {

    List<Client> findByCompanyId(Long companyId);  1 usage  romalytv

    Optional<Client> findByEmail(String email);  1 usage  romalytv

    List<Client> findByNameContainingIgnoreCase(String name);  1 usage  romalytv

    boolean existsByEmail(String email);  1 usage  new *

    boolean existsByPhone(String phone);  1 usage  romalytv
}
```

Рисунок 3.9 – Java-клас репозиторія ClientRepo

В цьому репозиторії описано декілька власних методів, такі як findCompanyId(), findByEmai(), findByNameContainingIgnoreCase() що знаходить клієнта по ID компанії, е-мейлу або імені, existsByEmail() та

existsByPhone(), що повертають значення true при присутності е-мейлу або телефону та false якщо навпаки

3.3.3 Рівень сервісів

Класи, що відповідають за сервісний рівень у Spring позначають анотацією @Service, що дозволяє Spring автоматично підключати їх через механізм залежностей. У більшості випадків сервіси взаємодіють із відповідними репозиторіями, обробляють отримані дані й готують результат для передачі контролерам.

Структурно майже всі сервіси в системі мають спільний підхід, наприклад усередині класів оголошуються фінальні поля з відповідними репозиторіями. Також багато сервісів мають методи стандартних CRUD-операції (findAll, save, deleteById, findById), дозволяючи ефективно взаємодіяти з даними. Крім того, у сервісах часто реалізуються додаткові методи для валідації (наприклад, перевірка наявності email або телефону), а також обробки винятків через orElseThrow(...), коли очікувані дані не знайдено.

```
@Service  5 usages  romalytv
public class InventoryItemService {

    private final InventoryItemRepo inventoryItemRepository;  6 usages

    public InventoryItemService(InventoryItemRepo inventoryItemRepository) {  romalytv
        this.inventoryItemRepository = inventoryItemRepository;
    }

    public List<InventoryItem> findAll() { return inventoryItemRepository.findAll(); }

    public List<InventoryItem> findByWarehouseId(Long warehouseId) {  1 usage  romalytv
        return inventoryItemRepository.findByWarehouseId(warehouseId);
    }

    public InventoryItem save(InventoryItem item) { return inventoryItemRepository.save(item); }

    public Optional<InventoryItem> findById(Long id) { return inventoryItemRepository.findById(id); }

    public void delete(Long id) { inventoryItemRepository.deleteById(id); }
}
```

Рисунок 3.10 – Java-клас сервісу InventoryItemService

Також існують деякі методи, які мають доволі комплексний підхід. Для створення графіків на сторінці з аналітикою було створено два методи, один з яких виконує підрахунок кількості клієнтів за шкалою задоволеності getFeedbackStats(), а другий обчислює прибутки та витрати за обраний період часу getProfitVsExpence()

```
public Map<String, Long> getFeedbackStats() {  1 usage   romalytv
    List<Client> clients = clientRepo.findAll();

    long veryHappy = clients.stream()
            .filter( Client c -> c.getSatisfactionScale() == SatisfactionScale.VERY_HAPPY)
            .count();

    long happy = clients.stream()
            .filter( Client c -> c.getSatisfactionScale() == SatisfactionScale.HAPPY)
            .count();

    long neutral = clients.stream()
            .filter( Client c -> c.getSatisfactionScale() == SatisfactionScale.NEUTRAL)
            .count();

    long unhappy = clients.stream()
            .filter( Client c -> c.getSatisfactionScale() == SatisfactionScale.UNHAPPY)
            .count();

    long veryUnhappy = clients.stream()
            .filter( Client c -> c.getSatisfactionScale() == SatisfactionScale.VERY_UNHAPPY)
            .count();

    return Map.of(
            k1: "veryHappy", veryHappy,
            k2: "happy", happy,
            k3: "neutral", neutral,
            k4: "unhappy", unhappy,
            k5: "veryUnhappy", veryUnhappy
    );
}
```

Рисунок 3.11 – Метод getFeedbackStats()

3.3.4 Рівень контролерів

Контролери в даній системі відіграють роль посередників між користувацькими запитами та бізнес-логікою, реалізованою на сервісному рівні. Вони відповідають за обробку HTTP-запитів, виклик відповідних методів сервісів, підготовку моделей даних і передачу їх до шаблонів представлення. Реалізовані контролери дотримуються принципів MVC-підходу та маркуються анотацією @Controller, що дозволяє Spring обробляти їх як компоненти обробки веб-запитів.

Для кожного функціонального модуля (аналітика, клієнти, замовлення, транзакції, склад) створено окремі контролери з відповідною маршрутизацією через @RequestMapping. Вхідні параметри отримуються через анотації @RequestParam, @PathVariable або @ModelAttribute, а результати взаємодії з сервісами додаються до об’єкта Model і передаються до шаблонів, створених за допомогою Thymeleaf.

```
@PostMapping("/save")  romalytv
public String saveClient(@Valid @ModelAttribute Client client,
                         BindingResult result,
                         Model model) {

    if (client.getId() == null) {
        if (clientService.emailExists(client.getEmail())) {
            result.rejectValue( field: "email",  errorCode: "error.client",  defaultMessage: "Email вже існує");
        }
        if (clientService.phoneExists(client.getPhone())) {
            result.rejectValue( field: "phone",  errorCode: "error.client",  defaultMessage: "Телефон вже використовується");
        }
    }

    if (result.hasErrors()) {
        model.addAttribute( attributeName: "satisfactionScale", SatisfactionScale.values());
        return "clients/client-form";
    }

    clientService.saveClient(client);
    return "redirect:/clients";
}
```

Рисунок 3.12 – Метод post-запиту /save контролера ClientController

У контролері ClientController існує приклад для звичайної логіки перевірки унікальності даних перед додаванням у БД. Цей метод перевіряє, чи введені поля е-мейлу та телефону вже задіяні у інших клієнтів та забороняє подальше створення нового клієнта, а також при помилці реєстрації перезапускає сторінку створення клієнта з вже внесеними даними та виводить помилки.

Також у системі присутні й контролери, які працюють з декількома сервісами одночасно, як AnalyticsController. В нього передаються різні дані для подальшого створення графіків у клієнтській частині – шаблон

3.3.5 Автентифікація та розподіл по ролям

Кожен користувач має унікальний логін, електронну адресу, пароль, а також прив'язку до ролі (Role) та компанії (Company), що дозволяє контролювати доступ до різних частин системи відповідно до прав користувача. Саме у SecurityConfig зосереджена більша частина функціоналу автентифікації користувачів та захисту паролей.

При шифруванні паролів використовується алгоритм Bcrypt, який є криптографічно стійким та стійким до атак перебору. BCrypt використовує так званий salt – випадкову послідовність байтів, яка додається до паролю перед хешуванням. Це унеможливлює створення універсальних хеш-таблиць. Навіть якщо два користувачі вводять один і той самий пароль, хеші в базі даних будуть різними, оскільки кожен з них отримає власний унікальний salt.

Контроль доступу до сторінок здійснюється через конфігурацію SecurityFilterChain де для кожного URL-адресного шаблону вказується, яка роль має право доступу. В прописаному конфігі ADMIN має повний доступ до всіх модулів, ACCOUNTANT – до замовлень, транзакцій, аналітики а EMPLOYEE – лише до перегляду аналітики, клієнтів та інших вкладок.

```
.authorizeHttpRequests( AuthorizationManagerRequestMat... auth -> auth
        .requestMatchers("/register", "/api/register", "/login", "/css/**", "/js/**").permitAll()
        .requestMatchers("/dashboard", "/logout").authenticated()
        .requestMatchers("/clients/**").hasAnyRole( ...roles: "ADMIN", "ACCOUNTANT", "EMPLOYEE")
        .requestMatchers("/warehouse/**").hasRole("ADMIN")
        .requestMatchers("/orders/**").hasAnyRole( ...roles: "ADMIN", "ACCOUNTANT", "EMPLOYEE")
        .requestMatchers("/transactions/**").hasAnyRole( ...roles: "ADMIN", "ACCOUNTANT")
        .requestMatchers("/analytics/**").hasAnyRole( ...roles: "ADMIN", "ACCOUNTANT", "EMPLOYEE")
        .anyRequest().authenticated()
)
```

Рисунок 3.13 – Розподіл посилань на ролі

Маршрути, які не потребують автентифікації (реєстрація, вхід, ресурси CSS/JS), дозволено всім користувачам через .permitAll(). Усі інші запити вимагають автентифікації або перевірки ролі. У випадку порушення доступу користувача переадресовують на /access-denied.

3.4 Клієнтський рівень

Представницький рівень реалізовано за допомогою шаблонізатора Thymeleaf, який формує HTML-сторінки на основі даних з контролерів. Шаблони містять форми, таблиці та блоки з умовною логікою. Для динаміки та інтерактивності використовуються JavaScript і бібліотека Chart.js, зокрема для візуалізації аналітики у вигляді графіків прямо на клієнтській стороні.

3.4.1 Шаблони представлення Thymeleaf

Шаблони реалізовані за допомогою Thymeleaf, який є серверним шаблонізатором, що інтегрується зі Spring Boot і дозволяє формувати HTML-сторінки на основі переданих моделей. Структура шаблонів відповідає логіці розділів додатку: кожен модуль (клієнти, замовлення, транзакції, склад, аналітика) має власний підкаталог із наборами сторінок для перегляду, створення, редагування та деталізації.

Використання атрибутів th:text, th:each, th:if, th:field, th:href забезпечує повноцінне динамічне наповнення HTML-контенту. У рисунку 3.14 шаблон client-view.html будує таблицю з переліком клієнтів та керуючими кнопками.

```
<table class="table table-bordered table-hover align-middle eroma-table">
  <thead class="table-light">
  <tr>
    <th>Ім'я</th>
    <th>Email</th>
    <th>Телефон</th>
    <th>Рівень оцінки</th>
    <th>Компанія</th>
    <th>📋</th>
  </tr>
  </thead>
  <tbody>
  <tr th:each="client : ${clients}">
    <td th:text="${client.name}">Ім'я</td>
    <td th:text="${client.email}">Email</td>
    <td th:text="${client.phone}">Телефон</td>
    <td th:text="${client.satisfactionScale.displayName}"
        th:classappend="${client.satisfactionScale.cssClass}">
    </td>
    <td>
      <a th:if="${client.company != null}"
         th:href="@{/clients/company/{id}(id=${client.company.id})}"
         class="eroma-link"
         style="...">
        <span th:text="${client.company.name}">Назва компанії</span>
      </a>
      <span th:if="${client.company == null}" class="text-muted">Компанії немає</span>
    </td>

    <td>
      <div class="btn-group">
        <a th:href="@{/clients/details/{id}(id=${client.id})}" class="btn btn-outline-info btn-sm">Деталі</a>
        <a th:href="@{/clients/edit/{id}(id=${client.id})}" class="btn btn-outline-primary btn-sm">✏️</a>
        <a th:href="@{/clients/delete/{id}(id=${client.id})}" class="btn btn-outline-danger btn-sm"
           onclick="return confirm('Видалити клієнта?')">🗑️</a>
      </div>
    </td>
  </tr>
  </tbody>
</table>
```

Рисунок 3.14 – Таблиця у client-view.html

Оформлення шаблонів базується на Bootstrap 5, створюючи адаптивний і візуально цілісний інтерфейс. Для кожного модуля також використовуються окремі кастомні стилі, які підключаються через файли на кшталт eroma-clients.css або eroma-inventory.css. Навігаційна панель із фіксацією у верхній частині сторінки також була додана до кожного шаблону у цілях спрощення перемикання між розділами, забезпечуючи зручність у користуванні системою.

3.4.2 Використання JavaScript у шаблонах

У межах реалізації вебінтерфейсу JavaScript використовується для обробки базових динамічних дій на стороні клієнта. Зокрема, в окремих шаблонах реалізовано надсилання форм за допомогою fetch() без перезавантаження сторінки, відображення повідомлень про результат операції, підтвердження видалення об’єктів, а також оновлення окремих елементів сторінки в режимі реального часу.

Використання JavaScript у проєкті є допоміжним інструментом, що забезпечує базову інтерактивність при роботі з деякими сторінками, таких як реєстрація register.html. Усі скрипти виконуються окремо від шаблонів Thymeleaf, дотримуючись розділення логіки та представлення.

3.4.3 Імплементування JavaScript у шаблони Thymeleaf

Для імплементування в шаблони Thymeleaf код Javascript’а був приєднаний напряму до шаблону з HTML. [26] Яскравим прикладом імплементації є додавання коду до сторінки з аналітикою, враховуючи що код був написаний на основі бібліотеки Chart.js.

Імплементування JavaScript у шаблони Thymeleaf дозволило органічно поєднати серверну логіку з клієнтським інтерфейсом без потреби в окремих запитах до API, що значно спростило роботу з динамічними даними, особливо в модулі аналітики. Весь необхідний обсяг інформації формується на бекенді та одразу вбудовується у сторінку, що прискорює відображення графіків і знижує навантаження на систему.

```
<script>
    function drawLineChart(canvasId, data) {  Show usages  romalytv
        const ctx = document.getElementById(canvasId).getContext('2d');
        new Chart(ctx, {
            type: 'line',
            data: {
                labels: data.labels,
                datasets: [
                    {
                        label: 'Прибуток',
                        data: data.profit,
                        borderColor: 'green',
                        backgroundColor: 'rgba(0, 128, 0, 0.2)',
                        fill: true,
                        tension: 0.3
                    },
                    {
                        label: 'Витрати',
                        data: data.expense,
                        borderColor: 'red',
                        backgroundColor: 'rgba(255, 0, 0, 0.2)',
                        fill: true,
                        tension: 0.3
                    }
```

Рисунок 3.15 – JavaScript-код у шаблоні analytics.html

Візуалізація статистичних даних здійснюється за допомогою бібліотеки Chart.js, яка виводить графіки на основі даних, що були попередньо підготовлені в контролері й передані у вигляді змінних Thymeleaf. Для цього використовується спеціальна конструкція th:inline="javascript", яка дозволяє інтегрувати серверні об'єкти безпосередньо у JavaScript-код.

ВИСНОВОК ДО РОЗДІЛУ 3

У цьому розділі було детально розглянуто основний функціонал створеної системи з програмної точки зору. Особливу увагу приділено структурі всієї програми, яка чітко поділена на рівні: база даних, серверна логіка та клієнтська частина. На основі наданих фрагментів реалізації було показано, як кожен рівень взаємодіє з іншими і забезпечує узгоджену логіку роботи всієї системи.

У базі даних було реалізовано сутності, що відповідають основним об’єктам системи ведення обліку даних і підтримки процесів малого бізнесу: Client, Order, Product, Warehouse, Transaction, Company, User та пов’язані довідники. Вони позначені анотаціями JPA для відображення в реляційній базі. Складні зв’язки між таблицями, такі як @ManyToOne, @OneToMany, @ManyToMany, забезпечують логічну узгодженість і дозволяють ефективно агрегувати дані для обліку, аналітики й відображення. Сутності доповнені валідаційними анотаціями, що дозволяє одразу фільтрувати некоректні вхідні дані на рівні моделі.

На рівні репозиторіїв було використано JpaRepository – всі інтерфейси успадковують цей тип і надають готові методи findAll, findById, save, deleteById. Додатково в окремих модулях, як-от транзакції або замовлення, створені методи на основі запитів до полів (findByWarehouseFromIdOrWarehouseToId(), findByCompanyId(), тощо), що дозволяє зменшити обсяг ручного SQL-коду. Репозиторії залишаються вузькоспеціалізованими, передаючи логіку обробки на рівень сервісів.

У серверній логіці кожен функціональний модуль має відповідний сервісний клас – наприклад, ClientService, OrderService, TransactionService, UserService – який відповідає за реалізацію бізнес-логіки та обробку запитів до бази даних. Ці сервіси отримують необхідні залежності через конструктор і

забезпечують чисту реалізацію логіки, зокрема в UserService реалізовано реєстрацію користувача з шифруванням пароля за допомогою BCryptPasswordEncoder. Також у серверній частині відіграють контролери, які реалізують зв'язок між HTTP-запитами клієнта та відповідними сервісами. Вони позначені анотацією @Controller або @RestController, обробляють запити через @GetMapping, @PostMapping, @RequestParam, @PathVariable, і передають дані в шаблони через об’єкт Model. Для кожного модуля створено окремий контролер, що підтримує принципи модульності та спрощує підтримку системи.

На клієнтському рівні шаблони HTML створені з використанням Thymeleaf. У прикладах з register.html та analytics.html видно, як реалізується повноцінна форма з валідацією та графіки за допомогою Chart.js. Дані передаються з бекенду безпосередньо в JavaScript через th:inline="javascript". Окремий скрипт у шаблоні обробляє запити до API реєстрації (fetch("/api/register")), що дозволяє оновлювати інтерфейс без перезавантаження сторінки.

РОЗДІЛ 4. ОГЛЯД ГОТОВОЇ СИСТЕМИ

Цей розділ присвячений огляду функціоналу системи зі сторони користувача, та можливості, що надає веб-додаток.

4.1 Сторінки входу та реєстрації

Функціональність авторизації та реєстрації в системі реалізована у вигляді двох окремих сторінок: сторінки входу (/login) та сторінки реєстрації (/register). Користувачі, які вже мають обліковий запис, можуть увійти в систему, вказавши ім’я користувача та пароль. Після успішної автентифікації вони автоматично перенаправляються до панелі керування (/dashboard).

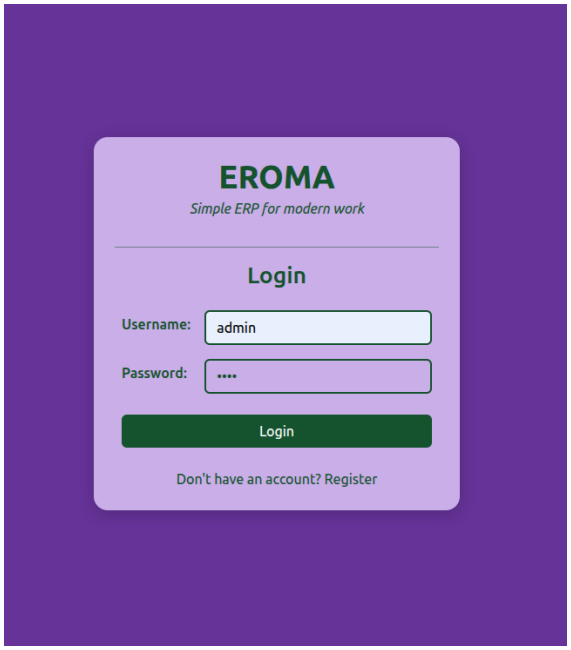


Рисунок 4.1 – Сторінка логіну

Реєстраційна форма відправляє дані через JavaScript (fetch API) на бекенд за маршрутом /api/register, де обробляється відповідним сервісом. У разі помилок (наприклад, вже існуючого логіна) повертається повідомлення без перезавантаження сторінки.

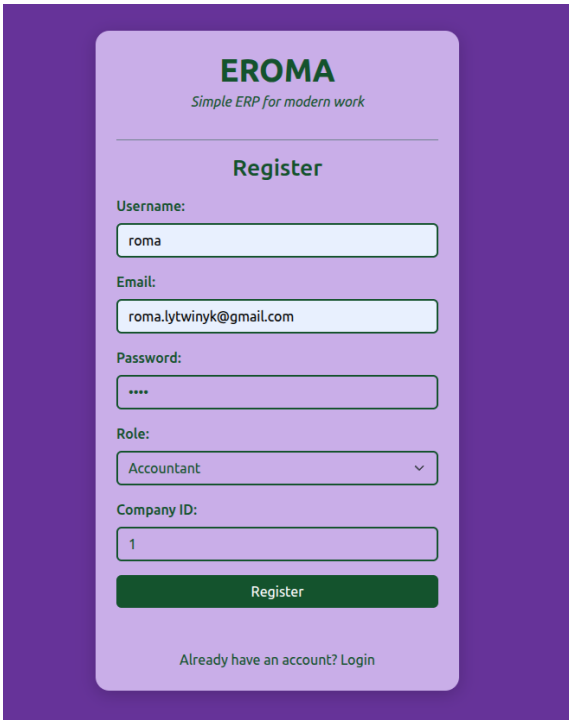


Рисунок 4.2 – Сторінка реєстрації

Після успішного входу користувач переходить до основної панелі керування, яка служить зручним навігаційним центром. На ній відображається коротка статистика по основних модулях системи: кількість клієнтів, стан складу та останнє замовлення. Панель адаптується під права доступу користувача: наприклад, бухгалтер бачить лише частину функціоналу, а адміністратор – повний доступ до всіх розділів.

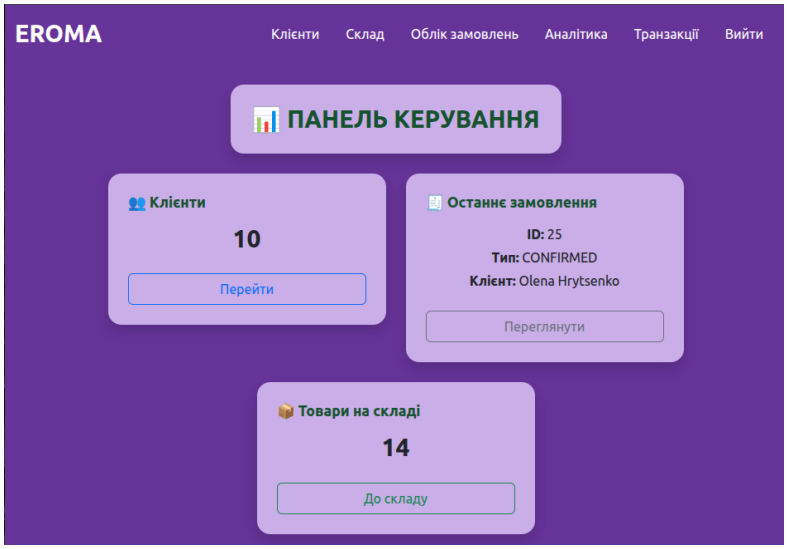


Рисунок 4.3 – Головна сторінка

4.2 Сторінка Клієнти

Розділ клієнтів надає користувачам можливість керувати всіма зареєстрованими клієнтами компанії в єдиному інтерфейсі. Основна сторінка цього модуля реалізує табличне представлення, де зібрано ключову інформацію: ім’я, email, телефон, рівень задоволеності та назва компанії. Таблиця інтерактивна — кожен запис містить кнопки для перегляду детальної інформації, редагування або видалення, що значно спрощує роботу з клієнтською базою.

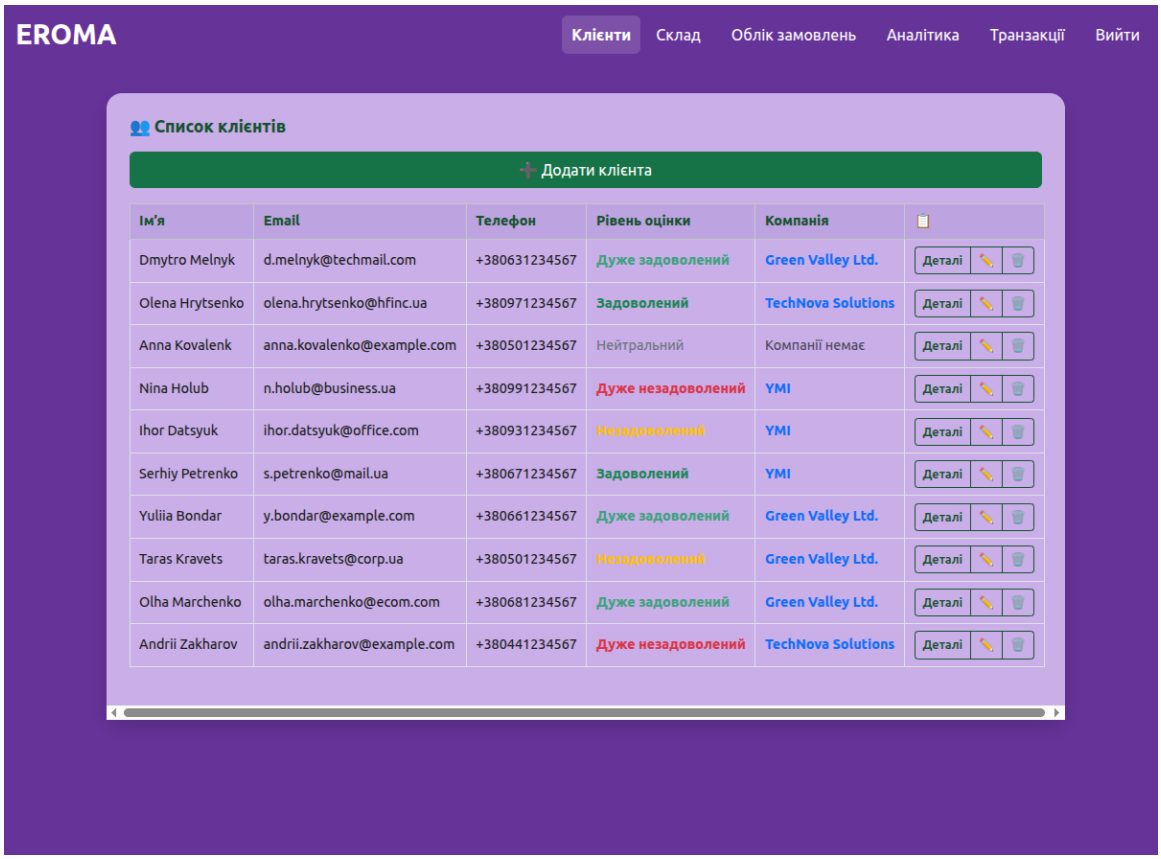


Рисунок 4.4 – Сторінка «Клієнти»

Процес додавання нового клієнта реалізовано через окрему форму, яка доступна натисканням кнопки "Додати клієнта". Вона передбачає заповнення всіх основних полів та вибір рівня задоволеності зі списку. Після збереження

нового запису користувач автоматично повертається на головну таблицю з оновленими даними, що демонструє інтерактивну взаємодію без потреби в повному оновленні інтерфейсу.

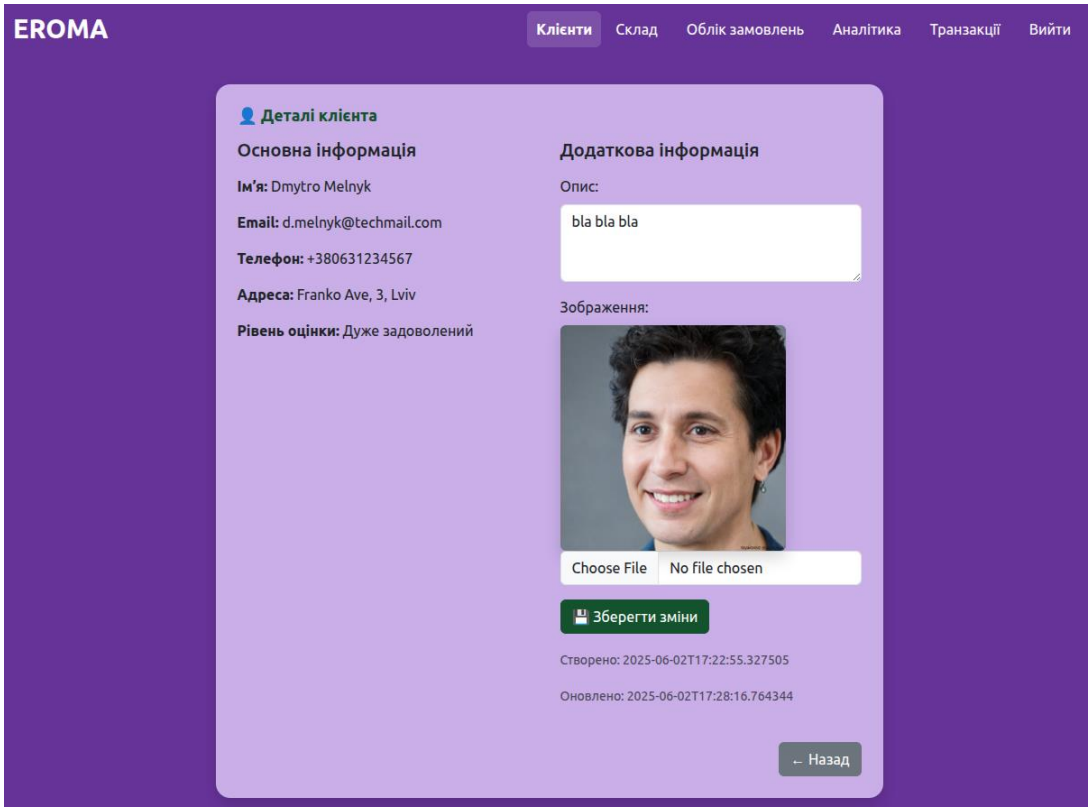


Рисунок 4.5 – Додаткова інформація про клієнтів

Також можна переглянути детальну інформацію про кожного клієнта, включаючи додаткові поля, зображення та час створення/оновлення. Таким чином, модуль "Клієнти" поєднує інтуїтивний інтерфейс із повним набором CRUD-операцій (створення, читання, оновлення, видалення), забезпечуючи зручне адміністрування клієнтської бази.

4.3 Сторінка Склад

У сторінці "Склад" реалізовано повноцінний облік залишків товарів у межах компанії. Інтерфейс сторінки дозволяє переглядати список наявних товарів із зазначенням складу, кількості та часу останнього оновлення. Ці дані можна безпосередньо редагувати: у кожному рядку є поле для зміни кількості та кнопка збереження, яка відправляє зміни на сервер. Такий підхід забезпечує швидку інвентаризацію без необхідності переходу на окрему сторінку редагування.

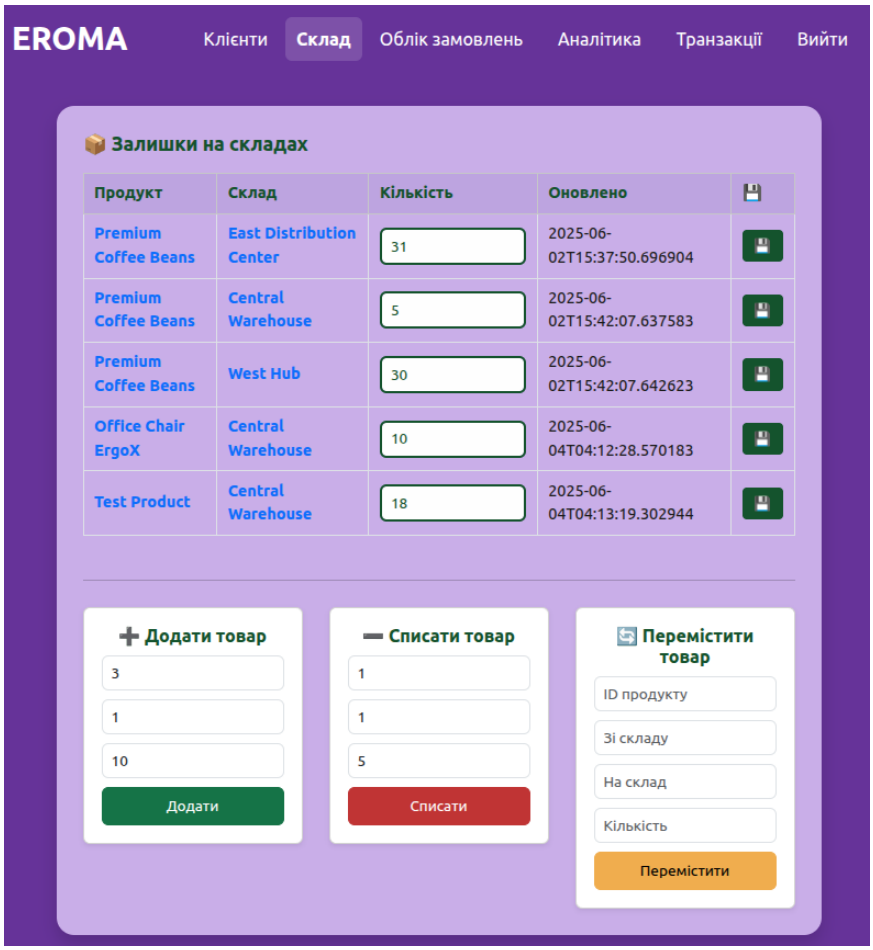


Рисунок 4.6 – Сторінка «Склад»

Також окрім перегляду, реалізовано три ключові дії: додавання, списання та переміщення товарів. У відповідних секціях користувач може ввести ID

товару, складу (або складів для переміщення) і кількість. Після натискання кнопки (Додати, Списати, Перемістити) система обробляє запит і змінює кількісні показники у базі.

4.4 Сторінка Облік замовлень

«Облік замовлень» має функціонал перегляду, створення, редагування та видалення замовлень. Головна таблиця відображає перелік усіх замовлень із зазначенням клієнта, компанії, статусу оплати (наприклад, NEW, CONFIRMED, PAID) та типу доставки (Shipped, Processing тощо). Для кожного замовлення передбачено кнопки для перегляду деталей, редагування або його видалення.

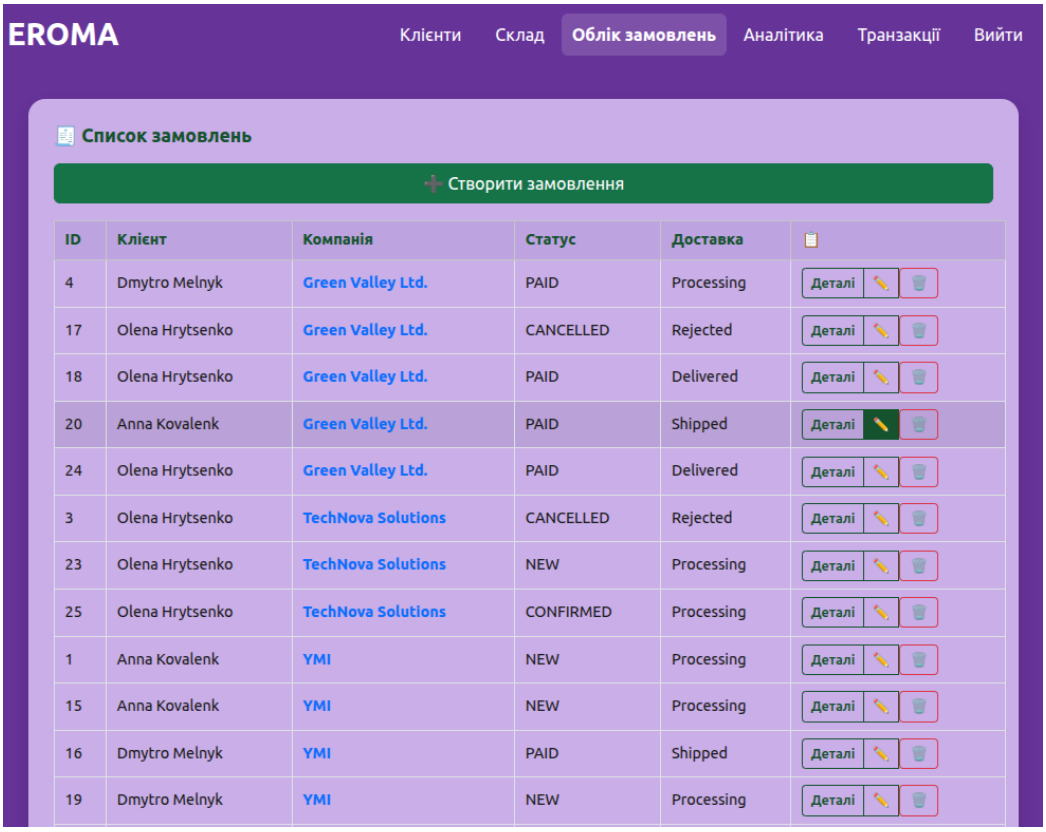


Рисунок 4.7 – Сторінка «Облік замовлень»

При відкритті детального перегляду замовлення користувач бачить усю ключову інформацію про замовлення: ім’я клієнта, назву компанії, статус оплати, доставку, а також список товарів із кількістю, ціною та підсумковою сумою.

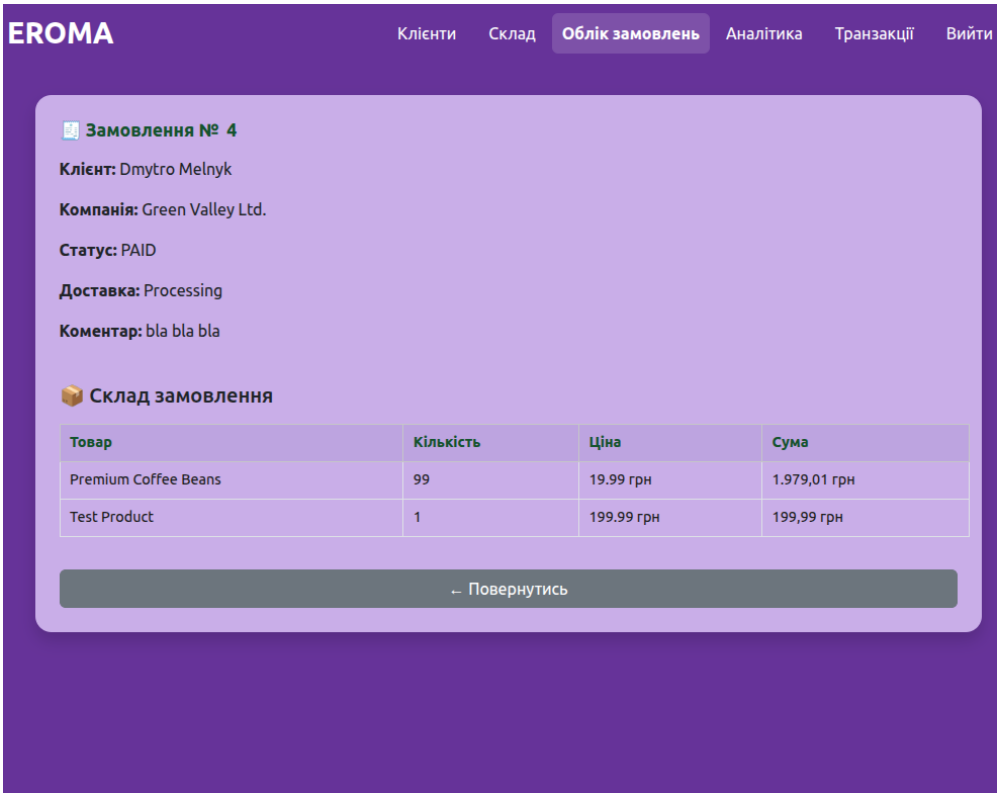


Рисунок 4.8 – Сторінка додаткової інформації замовлення

4.5 Сторінка Транзакції

У розділі «Транзакції» представлено інтерфейс для обліку фінансових операцій компанії. Головна таблиця відображає ключову інформацію: тип транзакції (доходи або витрати), суму, опис, дату, компанію та пов’язане замовлення (за потреби). Кожен запис можна швидко змінити або видалити за допомогою відповідних кнопок, що спрощує актуалізацію фінансових даних.

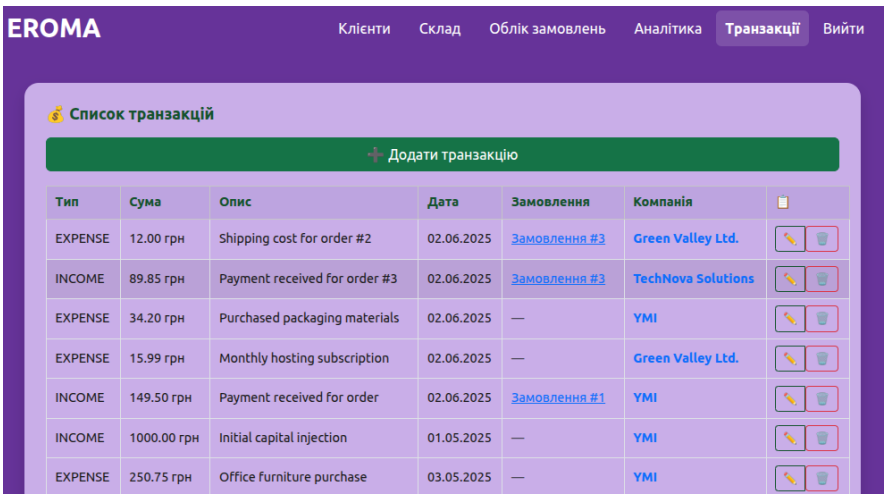


Рисунок 4.9 – Вкладка «Транзакіці»

Форма редагування транзакції надає інтерфейс для зміни основних параметрів фінансової операції: типу (дохід або витрата), суми, опису, компанії та, за потреби, пов’язаного замовлення.

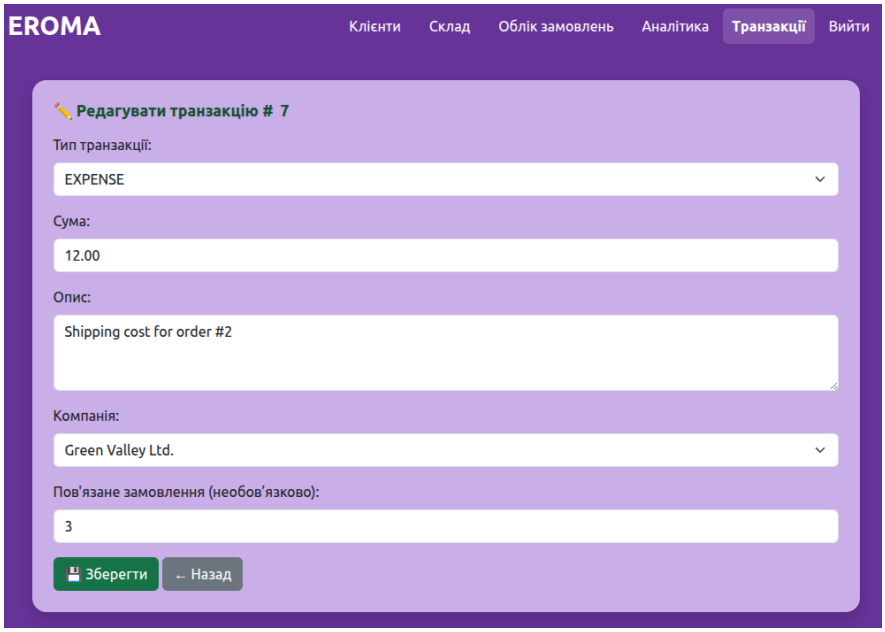


Рисунок 4.10 – Форма редагування транзакції

4.6 Сторінка Аналітика

Вкладка "Аналітика" у вебзастосунку слугує для візуального представлення ключових показників компанії. На ній реалізовано два основні типи аналітики: динаміка прибутків і витрат за певні періоди часу (тиждень, місяць, рік) та розподіл клієнтських оцінок задоволеності. Діаграми побудовані на основі бібліотеки Chart.js і оновлюються відповідно до обраного часового проміжку. Для побудови графіків дані передаються з бекенду у вигляді серій значень, що дозволяє відобразити як зміну фінансових потоків, так і загальну картину клієнтської лояльності.

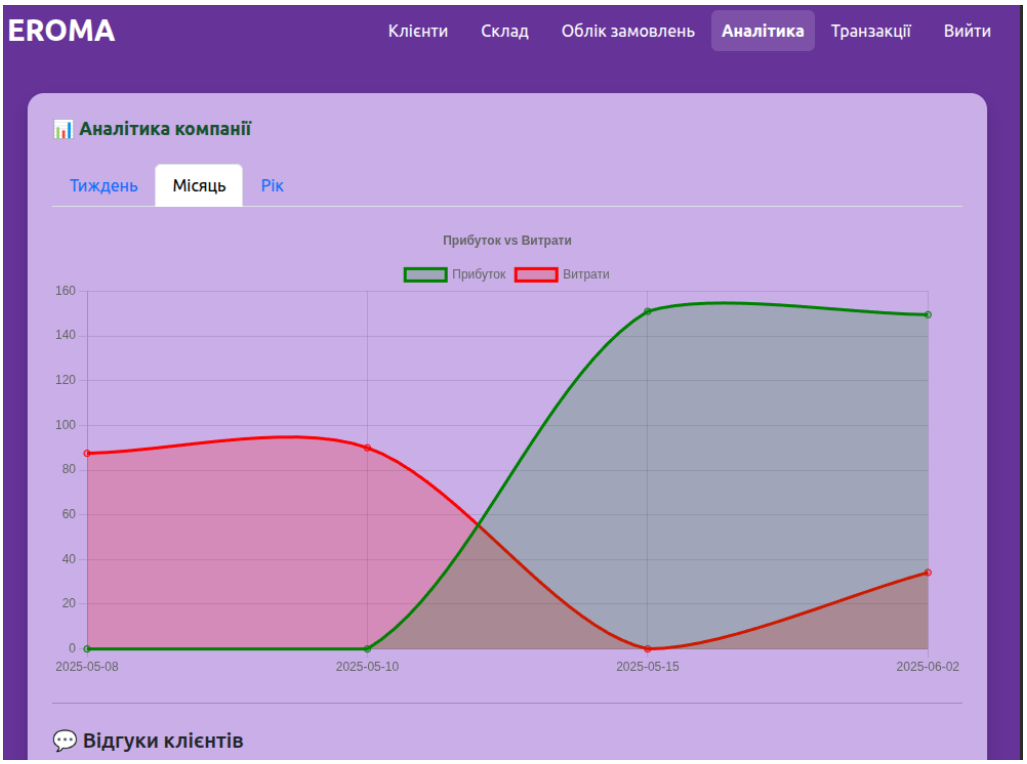


Рисунок 3.11 – Сторінка «Аналітика», графік витрат та прибутків

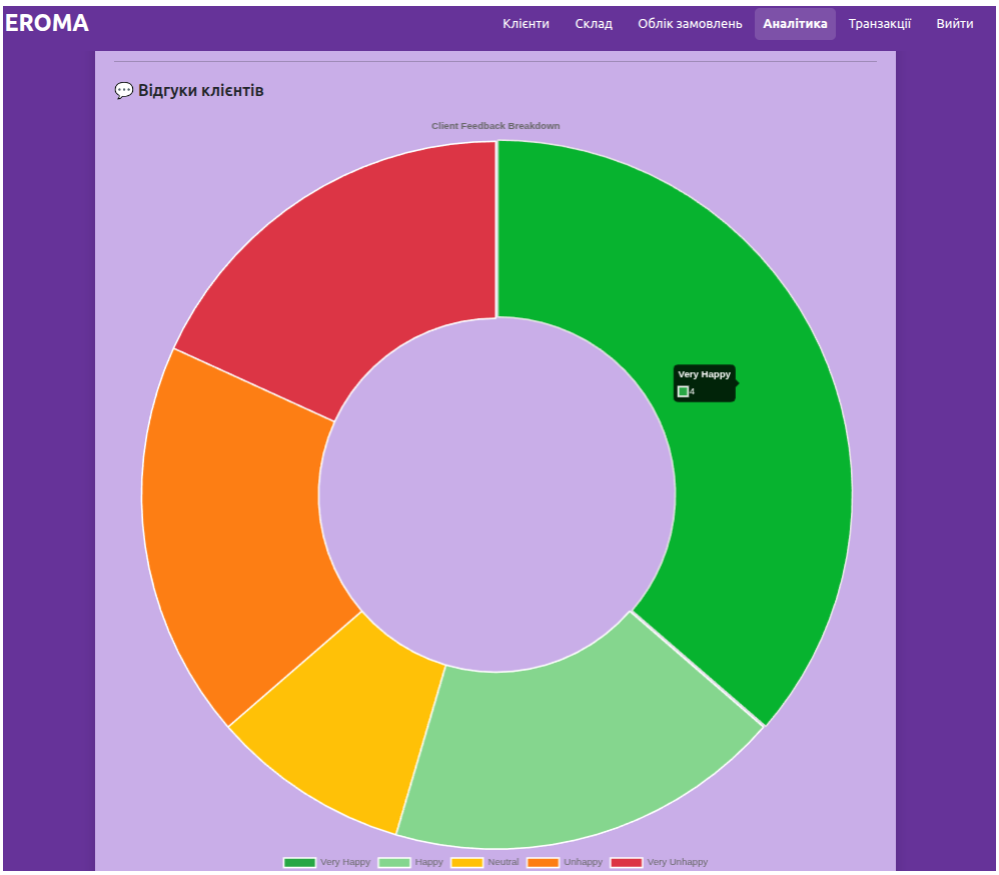


Рисунок 3.12 – Сторінка «Аналітика», кругова діаграма статусу задоволення клієнтів

При наведенні на окремі точки на графіках або кольору на круговій діаграмі можна побачити детальну інформацію про кількість наданих або відданих грошей та кількість клієнтів даного стану задоволеності послугами.

ВИСНОВОК ДО РОЗДІЛУ 4

Цей розділ містить детальний огляд користувацького інтерфейсу веб-додатку, аналізуючи структуру та принципи організації взаємодії. Процедури автентифікації та реєстрації реалізовані як стандартні точки входу, що надають доступ до функціоналу системи, а також до початкової сторінки, де висвітлено окремі модулі останніх подій. Ця початкова сторінка виконує роль центральної панелі керування, з якої користувач може швидко перейти до основних розділів системи: управління клієнтами, складським обліком, замовленнями, транзакціями та аналітикою.

Робота з даними в основних модулях, як-от управління клієнтською базою чи облік замовлень, побудована на використанні інтерактивних табличних представлень. Такий підхід дозволяє користувачеві виконувати повний спектр CRUD-операцій – створення, читання, оновлення та видалення сутностей. Застосування JavaScript, зокрема для асинхронної взаємодії з серверною частиною, забезпечує оновлення даних в інтерфейсі без повного перезавантаження сторінки, що є одним з факторів, котрі впливають на загальний досвід користування системою.

Окрім базових операцій, інтерфейс надає розширений набір інструментів для виконання більш спеціалізованих завдань, таких як управління складськими процесами та ведення фінансового обліку. Аналітичні можливості системи представлені модулем "Аналітика", де за допомогою бібліотеки Chart.js здійснюється візуалізація ключових бізнес-метрик. Воно пертворює коремо розташовані числові дані і переносить їх на графіки, надаючи користувачам інструмент для аналізу процесів, який може бути використаний для підготовки інформації в ході прийняття управлінських рішень.

ВИСНОВКИ

Метою цієї дипломної роботи було створення ергономічної та інтуативно-зрозумілої системи, орієнтовану на потреби малого бізнесу. Застосунок мав забезпечити операції над декількома головними потребами бізнес-звітності – стеження за рівнем задоволенності клієнтів та ведення їх обліку; перегляд та додавання продуктів на склади, можливість їх перевезення до клієнтів або на інші склади; вести звітність щодо обліковості замовлень та змінення їх статусів; змога перевіряти транзакції до компанії, закупівлі або продажі продуктів і т.п.; зручний перегляд деяких аспектів звітності, таких як підрахунок кількості статусів всіх користувачів та перегляд фінансових прибутків або відбутків на фіксований термін часу.

У першому розділі було проаналізовано сучасні підходи до побудови систем більшого та рівного масштабів бізнесу та проведено порівняльний огляд популярних платформ. Цей аналіз дозволив окреслити ключові функціональні та архітектурні рішення, що виявилися корисними для власної реалізації: модульність, підтримка інтерактивного інтерфейсу, безпека, адаптація до різних ролей користувача. Виявлені переваги сучасних систем стали відправною точкою для формування вимог до майбутнього застосунку.

Другий розділ був присвячений технічному обґрунтуванню вибору інструментів розробки. У ньому було визначено технологічний стек, основу якого склали Java, Spring Boot, PostgreSQL, Thymeleaf і бібліотеки для фронтенду. Вибір на користь Java та Spring пояснюється їх стабільністю, широким поширенням та наявністю потужного інструментарію для побудови масштабованих веб-застосунків. PostgreSQL обрано як основну СУБД завдяки підтримці реляційних зв’язків і надійності в роботі з транзакційними даними.

У третьому розділі описано внутрішню структуру програми та реалізацію її основних компонентів. Створено набір сутностей для бази даних, які

охоплюють всі ключові об'єкти доменної моделі (клієнти, замовлення, товари, склади, транзакції, компанії, користувачі тощо). За допомогою Spring Data JPA реалізовано рівень доступу до даних, тоді як сервісний рівень відповідає за бізнес-логіку. Контролери забезпечують взаємодію між клієнтськими запитами та серверною обробкою, а фронтенд-частина — через Thymeleaf — дає змогу візуалізувати інформацію та працювати з нею інтерактивно.

У четвертому розділі було проведено огляд готової реалізації з точки зору користувача. Інтерфейс системи охоплює всі ключові модулі: сторінки входу та реєстрації, клієнти, склад, замовлення, транзакції та аналітика. Реалізовані функції забезпечують виконання повного циклу роботи з даними, включаючи створення, перегляд, редагування й видалення. Інтерфейс зручний, адаптивний до ролей користувачів та побудований з урахуванням сучасних вимог до взаємодії.

У результаті реалізовано повнофункціональну систему ведення обліку даних та підтримки процесів малого бізнесу з усіма базовими модулями, необхідними для ведення внутрішнього обліку в компанії. Система є гнучкою, модульною та придатною до подальшого розвитку, зручна у використанні як для адміністратора, так і для бухгалтера чи аналітика.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Архітектура і модулі ERP системи - A4 Company. *A4 Company*. URL: https://a4.com.ua/arhitektura-i-moduli-erp-sistemi/ (дата звернення: 02.05.2025).
2. Perfectum CRM+ERP. *Perfectum CRM+ERP*. URL: https://perfectum.ua/ua/ (дата звернення: 17.05.2025).
3. Perfectum Додатки. *Perfectum CRM+ERP*. URL: https://perfectum.ua/ua/functional/enterprise-applications (дата звернення: 17.05.2025).
4. Perfectum No-Code інструменти. *Perfectum CRM+ERP*. URL: https://perfectum.ua/ua/functional/no-code (дата звернення: 17.05.2025).
5. Perfectum Архітектура. *Perfectum CRM+ERP*. URL: https://perfectum.ua/ua/architecture (дата звернення: 17.05.2025).
6. Perfectum Безпека. *Perfectum CRM+ERP*. URL: https://perfectum.ua/ua/security (дата звернення: 17.05.2025).
7. Ugla | Presentation. *Ugla ERP*. URL: https://present.ugla.ua/en (дата звернення: 17.05.2025).
8. UGLA | Модулі. *UGLA*. URL: https://ugla.ua/moduli/ (дата звернення: 17.05.2025).
9. Індекс документації Odoo. *Odoo*. URL: https://www.odoo.com/uk_UA/page/docs (дата звернення: 17.05.2025).
10. Функції Odoo Studio. *Odoo*. URL: https://www.odoo.com/uk_UA/app/studio-features (дата звернення: 17.05.2025).
11. Чому Java – найпопулярніша мова програмування у світі. *Курси Java Львів | Комплексний курс Java від Java Academy*.

URL: https://java.lviv.ua/chomu-java-najpopulyarnisha-mova-programuvannya-u-sviti (дата звернення: 17.05.2025).

12. Що таке Java і де вона використовується. *GoIT*.
URL: https://goit.global/ua/articles/shcho-take-java-i-de-vona-vykorystovuietsia (дата звернення: 17.05.2025).
13. Beekeeper R. Все, що ви хотіли знати про Maven. *JavaRush*.
URL: https://javarush.com/ua/groups/posts/uk.3119.vse-jsho-vi-khotli-znati-pro-maven---java-proekt-vd-a-do-ja (дата звернення: 17.05.2025).
14. Spring Framework. *Spring Framework*.
URL: https://spring.io/projects/spring-framework (date of access: 17.05.2025).
15. Spring boot: структура проекту та розробка веб-додатків. *FoxmindEd*.
URL: https://foxminded.ua/spring-boot/ (дата звернення: 17.05.2025).
16. Spring Web MVC :: Spring Framework. *Spring | Home*.
URL: https://docs.spring.io/spring-framework/reference/web/webmvc.html (дата звернення: 17.05.2025).
17. GeeksforGeeks. Spring - MVC Framework - GeeksforGeeks. *GeeksforGeeks*.
URL: https://www.geeksforgeeks.org/spring-mvc-framework/ (дата звернення: 17.05.2025).
18. Securing a Web Application. *Spring Framework*.
URL: https://spring.io/guides/gs/securing-web (дата звернення: 17.05.2025).
19. Spring Data JPA | Baeldung. *Baeldung*.
URL: https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa (дата звернення: 17.05.2025).
20. Lombok. *Project Lombok*.
URL: https://projectlombok.org/features/ (дата звернення: 17.05.2025).

21. Типи баз даних: особливості, відмінності та приклади. *DOU.ua*. URL: https://dou.ua/lenta/articles/types-of-databases/ (дата звернення: 17.05.2025).
22. PostgreSQL: About. *PostgreSQL*. URL: https://www.postgresql.org/about/ (дата звернення: 17.05.2025).
23. FAQ. *pgAdmin - PostgreSQL Tools*. URL: https://www.pgadmin.org/faq/ (дата звернення: 17.05.2025).
24. pgAdmin 4 9.3 documentation. *pgAdmin - PostgreSQL Tools*. URL: https://www.pgadmin.org/docs/pgadmin4/latest/getting_started.html (дата звернення: 17.05.2025).
25. Artur. Додаємо БД PostgreSQL до RESTful сервісу на Spring Boot. *JavaRush*. URL: https://javarush.com/ua/groups/posts/uk.2579.dodamo-bd-postgresql-do-restful-servsu-na-spring-boot-chastina-1 (дата звернення: 17.05.2025).
26. GitHub - romalytv/eroma. *GitHub*. URL: https://github.com/romalytv/eroma (дата звернення: 20.05.2025).

ДОДАТОК А

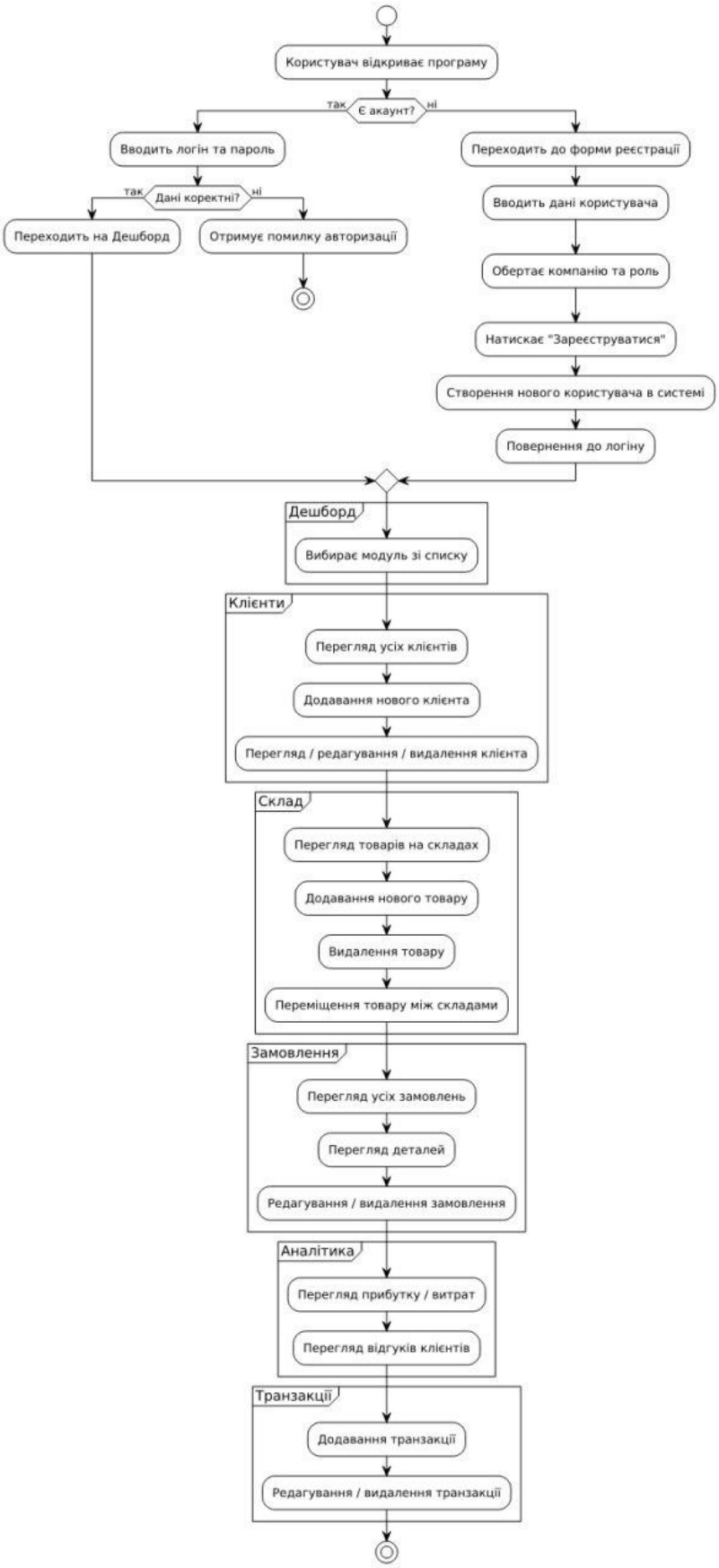
Система ведення обліку даних і підтримки процесів малого бізнесу

Логіка роботи системи (принципова схема)

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2025 р



					ІАЛЦ.467200.004 Д1		
		№ докум.	Підпис	Дата			
Розробив		Литвинюк Р.К.			**Система ведення обліку даних і підтримки процесів малого бізнесу** Логіка роботи системи (принципова схема)	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Волокита А. М.					
Реценз.		Шимкович В.М.				КПІ ім. Ігоря Сікорського, ФІОТ,ІМ-13	
Н. Контр.		Гончаренко О.О.					
Затвердив							

ДОДАТОК Б

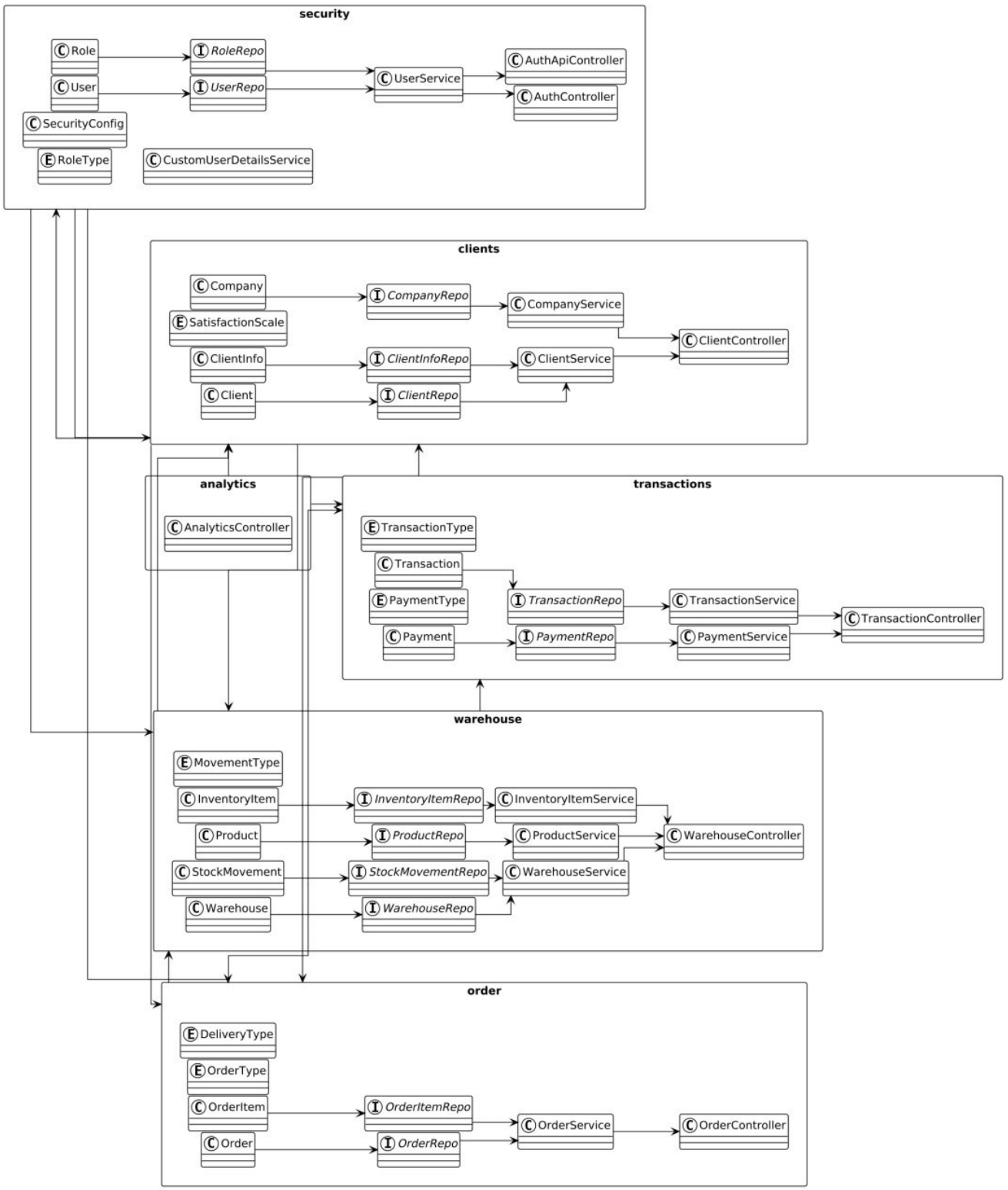
Система ведення обліку даних і підтримки процесів малого бізнесу

Діаграма класів (функціональна схема)

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2025 р



					ІАЛЦ.467200.005 Д2		
		№ докум.	Підпис	Дата			
Розробив		Литвинюк Р.К.			**Система ведення обліку даних і підтримки процесів малого бізнесу** Діаграма класів (функціональна схема)	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Волокита А. М.					
Реценз.		Шимкович В.М.				КПІ ім. Ігоря Сікорського, ФІОТ,ІМ-13	
Н. Контр.		Гончаренко О.О.					
Затвердив							

ДОДАТОК В

Система ведення обліку даних і підтримки процесів малого бізнесу

Модель бази даних (структурна схема)

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2025 р

					ІАЛЦ.467200.006 Д3	
		№ докум.	Підпис	Дата		
Розробив		Литвинюк Р.К.			**Система ведення обліку даних і підтримки процесів малого бізнесу**	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Волокита А. М.				
Реценз.		Шимкович В.М.			Модель бази даних (структурна схема)	КПІ ім. Ігоря Сікорського, ФІОТ,ІМ-13
Н. Контр.		Гончаренко О.О.				
Затвердив						

ДОДАТОК Г

Система ведення обліку даних і підтримки процесів малого бізнесу

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 30

Київ 2025 р

Client.java

```
import com.kpi.diploma.model.entity.types.DeliveryType
import com.kpi.diploma.model.entity.types.SatisfactionScale;
import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;

import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;

import java.util.List;

@Getter
@Setter
@Entity
@Table(name = "clients")
public class Client {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank(message = "Ім'я не може бути порожнім")
    @Column(nullable = false)
    private String name;

    @Email(message = "Email має бути коректним")
    @NotBlank(message = "Email не може бути порожнім")
    @Column(nullable = false)
    private String email;

    @NotBlank(message = "Номер телефону не може бути порожнім")
    @Column(nullable = false)
    private String phone;

    private String address;

    @Enumerated(EnumType.STRING)
    @Column(name = "satisfaction_scale")
    private SatisfactionScale satisfactionScale;


    @ManyToOne
    @JoinColumn(name = "company_id")
    private Company company;

    @ManyToMany
    @JoinTable(
            name = "client_products",
            joinColumns = @JoinColumn(name = "client_id"),
            inverseJoinColumns = @JoinColumn(name = "product_id")
    )

    @OneToOne(mappedBy = "client", cascade = CascadeType.ALL, orphanRemoval = true)

    private ClientInfo clientInfo;

    public Client() {}
```

```
    public Client(String name, String email, String phone, String address, SatisfactionScale
satisfactionScale, Company company) {
        this.name = name;
        this.email = email;
        this.phone = phone;
        this.address = address;
        this.satisfactionScale = satisfactionScale;
        this.company = company;
    }
}
```

ClientInfo.java

```
import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;

import org.hibernate.annotations.CreationTimestamp;
import org.hibernate.annotations.UpdateTimestamp;

import java.time.LocalDateTime;

@Getter
@Setter
@Entity
@Table(name = "client_info")
public class ClientInfo {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToOne
    @JoinColumn(name = "client_id", nullable = false, unique = true)
    private Client client;

    @Column(columnDefinition = "TEXT")
    private String detailsText;

    private String imagePath;

    @CreationTimestamp
    @Column(updatable = false)
    private LocalDateTime createdAt;

    @UpdateTimestamp
    private LocalDateTime updatedAt;

    public ClientInfo() {}

    public ClientInfo(Client client, String detailsText, String imagePath) {
        this.client = client;
        this.detailsText = detailsText;
        this.imagePath = imagePath;
    }
}
```

Company.java

```
import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;

import java.util.ArrayList;
import java.util.List;

import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;

import java.util.List;

import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;

import java.util.List;

@Getter
@Setter
@Entity
@Table(name = "companies")
public class Company {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String name;

    @OneToMany(mappedBy = "company", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<User> employees;

    @OneToMany(mappedBy = "company", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Client> clients;

    @OneToMany(mappedBy = "company", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Product> products;

    @OneToMany(mappedBy = "company", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Order> orders;

    public Company() {}

    public Company(String name) {
        this.name = name;
    }
}
```

InventoryItem.java

```
import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;
import org.hibernate.annotations.UpdateTimestamp;

import java.time.LocalDateTime;

@Getter
@Setter
@Entity
```

```
@Table(name = "inventory_items")
public class InventoryItem {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "product_id", nullable = false)
    private Product product;

    @ManyToOne
    @JoinColumn(name = "warehouse_id", nullable = false)
    private Warehouse warehouse;

    @Column(nullable = false)
    private Integer quantity;

    @UpdateTimestamp
    private LocalDateTime updatedAt;

    public InventoryItem() {}

    public InventoryItem(Product product, Warehouse warehouse, Integer quantity) {
        this.product = product;
        this.warehouse = warehouse;
        this.quantity = quantity;
    }
}
```

Order.java

```
import com.kpi.diploma.model.entity.types.DeliveryType;
import com.kpi.diploma.model.entity.types.OrderType;
import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;

import java.util.List;

@Getter
@Setter
@Entity
@Table(name = "orders")
public class Order {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "client_id", nullable = false)
    private Client client;

    @Enumerated(EnumType.STRING)
    @Column(name = "order_type", nullable = false)
    private OrderType orderType;

    @Enumerated(EnumType.STRING)
    @Column(name = "delivery_type", nullable = false)
    private DeliveryType deliveryType;

    @Column(columnDefinition = "TEXT")
    private String additionalInfo;
```

```
    @OneToMany(mappedBy = "order", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<OrderItem> items;

    @ManyToOne
    @JoinColumn(name = "company_id", nullable = false)
    private Company company;


    public Order() {}

    public Order(Client client, OrderType orderType, DeliveryType deliveryType, String
additionalInfo) {
        this.client = client;
        this.orderType = orderType;
        this.deliveryType = deliveryType;
        this.additionalInfo = additionalInfo;
    }
}
```

OrderItem.java

```
import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;

import java.math.BigDecimal;

@Getter
@Setter
@Entity
@Table(name = "order_items")
public class OrderItem {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "order_id", nullable = false)
    private Order order;

    @ManyToOne
    @JoinColumn(name = "product_id", nullable = false)
    private Product product;

    @Column(nullable = false)
    private Integer quantity;


    @Column(name = "unit_price", nullable = false)
    private BigDecimal unitPrice;

    public OrderItem() {}

    public OrderItem(Order order, Product product, Integer quantity, BigDecimal unitPrice) {
        this.order = order;
        this.product = product;
        this.quantity = quantity;
        this.unitPrice = unitPrice;
    }
}
```

Payment.java

```
import com.kpi.diploma.model.entity.types.PaymentType;
import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;

import java.math.BigDecimal;
import java.time.LocalDateTime;

import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;

import java.math.BigDecimal;
import java.time.LocalDateTime;

@Getter
@Setter
@Entity
@Table(name = "payments")
public class Payment {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "payment_date", nullable = false)
    private LocalDateTime paymentDate;

    @Column(nullable = false)
    private BigDecimal amount;

    @Enumerated(EnumType.STRING)
    @Column(name = "payment_type", nullable = false)
    private PaymentType paymentType;

    @ManyToOne
    @JoinColumn(name = "order_id", nullable = false)
    private Order order;

    public Payment() {}

    public Payment(LocalDateTime paymentDate, BigDecimal amount, PaymentType paymentType,
Order order) {
        this.paymentDate = paymentDate;
        this.amount = amount;
        this.paymentType = paymentType;
        this.order = order;
    }
}
```

Product.java

```
import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;

import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;
import org.hibernate.annotations.CreationTimestamp;
import org.hibernate.annotations.UpdateTimestamp;

import java.math.BigDecimal;
```

```
import java.time.LocalDateTime;
import java.util.List;

@Getter
@Setter
@Entity
@Table(name = "products")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false, unique = true)
    private String code;

    @Column(columnDefinition = "TEXT")
    private String description;

    @Column(nullable = false)
    private BigDecimal price;

    @Column(nullable = false)
    private String unit;

    @Column(nullable = false)
    private Integer quantity;

    private String imagePath;

    @ManyToMany(mappedBy = "products")
    private List<Client> clients;

    @CreationTimestamp
    @Column(updatable = false)
    private LocalDateTime createdAt;

    @UpdateTimestamp
    private LocalDateTime updatedAt;

    @ManyToOne
    @JoinColumn(name = "company_id", nullable = false)
    private Company company;


    public Product() {}

    public Product(String name, String code, String description, BigDecimal price, String
unit, Integer quantity, String imagePath) {
        this.name = name;
        this.code = code;
        this.description = description;
        this.price = price;
        this.unit = unit;
        this.quantity = quantity;
        this.imagePath = imagePath;
    }
}
```

Role.java

```
import com.kpi.diploma.model.entity.types.RoleType;
import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;
import java.util.List;

@Setter
@Getter
@Entity
@Table(name = "roles")
public class Role {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Enumerated(EnumType.STRING)
    @Column(nullable = false, unique = true)
    private RoleType name;

    @OneToMany(mappedBy = "role", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<User> users;

    public Role() {}

    public Role(RoleType name) {
        this.name = name;
    }

    public Role(RoleType name, List<User> users) {
        this.name = name;
        this.users = users;
    }
}
```

StockMovement.java

```
import com.kpi.diploma.model.entity.types.MovementType;
import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;

import java.time.LocalDateTime;

@Getter
@Setter
@Entity
@Table(name = "stock_movements")
public class StockMovement {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "product_id", nullable = false)
    private Product product;

    @ManyToOne
    @JoinColumn(name = "from_warehouse_id")
    private Warehouse fromWarehouse;

    @ManyToOne
```

```
    @JoinColumn(name = "to_warehouse_id")
    private Warehouse toWarehouse;

    @Column(nullable = false)
    private Integer quantity;

    @Enumerated(EnumType.STRING)
    @Column(name = "movement_type", nullable = false)
    private MovementType movementType;

    @Column(nullable = false)
    private LocalDateTime date;

    @Column(columnDefinition = "TEXT")
    private String additionalInfo;

    public StockMovement() {}

    public StockMovement(Product product, Warehouse fromWarehouse, Warehouse toWarehouse,
                         Integer quantity, MovementType movementType, LocalDateTime date,
String additionalInfo) {
        this.product = product;
        this.fromWarehouse = fromWarehouse;
        this.toWarehouse = toWarehouse;
        this.quantity = quantity;
        this.movementType = movementType;
        this.date = date;
        this.additionalInfo = additionalInfo;
    }
}
```

Transaction.java

```
import com.kpi.diploma.model.entity.types.TransactionType;
import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;

import java.math.BigDecimal;
import java.time.LocalDateTime;

@Getter
@Setter
@Entity
@Table(name = "transactions")
public class Transaction {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private LocalDateTime date;

    @Column(columnDefinition = "TEXT")
    private String description;

    @Enumerated(EnumType.STRING)
    @Column(name = "transaction_type", nullable = false)
    private TransactionType transactionType;

    @Column(nullable = false)
    private BigDecimal amount;

    @ManyToOne
```

```
    @JoinColumn(name = "related_order_id")
    private Order relatedOrder;

    @ManyToOne
    @JoinColumn(name = "company_id", nullable = false)
    private Company company;

    public Transaction() {}

    public Transaction(LocalDateTime date, String description, TransactionType
transactionType,
                       BigDecimal amount, Order relatedOrder, Company company) {
        this.date = date;
        this.description = description;
        this.transactionType = transactionType;
        this.amount = amount;
        this.relatedOrder = relatedOrder;
        this.company = company;
    }
}
```

User.java

```
import jakarta.persistence.*;
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
import lombok.Getter;
import lombok.Setter;

@Setter
@Getter
@Entity
@Table(name = "users")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String username;

    @Column(nullable = false)
    private String password;

    @Column(nullable = false)
    @Email(message = "Email має бути коректним")
    @NotBlank(message = "Email не може бути порожнім")
    private String email;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "role_id")
    private Role role;

    @ManyToOne
    @JoinColumn(name = "company_id")
    private Company company;

    public User() {}

    public User(Long id, String username, String password, String email, Role role, Company
company) {
        this.id = id;
        this.username = username;
        this.password = password;
```

```
        this.email = email;
        this.role = role;
        this.company = company;
    }
}
```

Warehouse.java

```
import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;

import java.util.List;

@Getter
@Setter
@Entity
@Table(name = "warehouses")
public class Warehouse {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String name;

    private String location;

    @OneToMany(mappedBy = "warehouse", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<InventoryItem> inventoryItems;

    public Warehouse() {}

    public Warehouse(String name, String location) {
        this.name = name;
        this.location = location;
    }
}
```

DeliveryType.java

```
import lombok.Getter;

@Getter
public enum DeliveryType {
    PROCESSING("Processing"),
    SHIPPED("Shipped"),
    DELIVERED("Delivered"),
    REJECTED("Rejected");

    private final String displayName;

    DeliveryType(String displayName) {
        this.displayName = displayName;
    }

}
```

MovementType.java

```
package com.kpi.diploma.model.entity.types;

public enum MovementType {
    IN,
```

```
    OUT,
    TRANSFER
}
```

OrderType.java

```
public enum OrderType {
    NEW,
    CONFIRMED,
    PAID,
    CANCELLED
}
```

PaymentType.java

```
public enum PaymentType {
    CASH,
    CARD,
    BANK_TRANSFER,
    PAYPAL,
    OTHER
}
```

RoleType.java

```
public enum RoleType {
    ADMIN,
    ACCOUNTANT,
    EMPLOYEE
}
```

SatisfactionScale.java

```
import lombok.Getter;

@Getter
public enum SatisfactionScale {
    VERY_HAPPY("Дуже задоволений", "text-green-soft fw-bold"),
    HAPPY("Задоволений", "text-success fw-bold"),
    NEUTRAL("Нейтральний", "text-secondary"),
    UNHAPPY("Незадоволений", "text-warning fw-bold"),
    VERY_UNHAPPY("Дуже незадоволений", "text-danger fw-bold");

    private final String displayName;
    private final String cssClass;

    SatisfactionScale(String displayName, String cssClass) {
        this.displayName = displayName;
        this.cssClass = cssClass;
    }

}
```

TransactionType.java

```
public enum TransactionType {
    INCOME,
    EXPENSE
}
```

ClientInfoRepo.java

```
import com.kpi.diploma.model.entity.ClientInfo;
import org.springframework.data.jpa.repository.JpaRepository;
```

```
import java.util.Optional;

public interface ClientInfoRepo extends JpaRepository<ClientInfo, Long> {
    Optional<ClientInfo> findByClientId(Long clientId);
}
```

ClientRepo.java

```
import com.kpi.diploma.model.entity.Client;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.List;
import java.util.Optional;

@Repository
public interface ClientRepo extends JpaRepository<Client, Long> {

    List<Client> findByCompanyId(Long companyId);

    Optional<Client> findByEmail(String email);

    List<Client> findByNameContainingIgnoreCase(String name);

    boolean existsByEmail(String email);

    boolean existsByPhone(String phone);
}
```

CompanyRepo.java

```
import com.kpi.diploma.model.entity.Company;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.List;
import java.util.Optional;

@Repository
public interface CompanyRepo extends JpaRepository<Company, Long> {

    Optional<Company> findByName(String name);

    boolean existsByName(String name);

    List<Company> findByNameContainingIgnoreCase(String namePart);
}
```

InventoryItemRepo.java

```
import com.kpi.diploma.model.entity.InventoryItem;
import com.kpi.diploma.model.entity.Product;
import com.kpi.diploma.model.entity.Warehouse;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.Collection;
import java.util.List;
import java.util.Optional;

@Repository
public interface InventoryItemRepo extends JpaRepository<InventoryItem, Long> {
    Optional<InventoryItem> findByWarehouseIdAndProductId(Long warehouseId, Long productId);
    List<InventoryItem> findByWarehouseId(Long warehouseId);
```

```
    List<InventoryItem> findByWarehouse(Warehouse warehouse);
}
```

OrderItemRepo.java

```
import com.kpi.diploma.model.entity.OrderItem;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
public interface OrderItemRepo extends JpaRepository<OrderItem, Long> {
    List<OrderItem> findByOrderId(Long orderId);
}
```

OrderRepo.java

```
import com.kpi.diploma.model.entity.Order;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
public interface OrderRepo extends JpaRepository<Order, Long> {
    List<Order> findByCompanyId(Long companyId);
}
```

PaymentRepo.java

```
import com.kpi.diploma.model.entity.Payment;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
public interface PaymentRepo extends JpaRepository<Payment, Long> {
    List<Payment> findByOrderId(Long orderId);
}
```

ProductRepo.java

```
import com.kpi.diploma.model.entity.Product;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.List;
import java.util.Optional;

@Repository
public interface ProductRepo extends JpaRepository<Product, Long> {

    Optional<Product> findByName(String name);

    List<Product> findByCompanyId(Long companyId);
}
```

RoleRepo.java

```
import com.kpi.diploma.model.entity.Role;
import com.kpi.diploma.model.entity.types.RoleType;
import org.springframework.data.jpa.repository.JpaRepository;
```

```
import java.util.Optional;

public interface RoleRepo extends JpaRepository<Role, Long> {

    Optional<Role> findByName(RoleType role);
}
```

StockMovementRepo.java

```
import com.kpi.diploma.model.entity.StockMovement;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
public interface StockMovementRepo extends JpaRepository<StockMovement, Long> {
    List<StockMovement> findByFromWarehouse_IdOrToWarehouse_Id(Long fromId, Long toId);
}
```

TransactionRepo.java

```
import com.kpi.diploma.model.entity.Transaction;
import com.kpi.diploma.model.entity.types.TransactionType;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import java.time.LocalDateTime;
import java.util.List;

@Repository
public interface TransactionRepo extends JpaRepository<Transaction, Long> {
    List<Transaction> findByCompanyId(Long companyId);

    @Query("SELECT COALESCE(SUM(t.amount), 0) FROM Transaction t WHERE t.transactionType =
:type AND t.date BETWEEN :from AND :to")
    double sumAmountByTypeBetween(@Param("type") TransactionType type,
                                  @Param("from") LocalDateTime from,
                                  @Param("to") LocalDateTime to);

    List<Transaction> findByDateBetween(LocalDateTime from, LocalDateTime to);

    List<Transaction> findByCompanyIdAndDateBetween(Long companyId, LocalDateTime start,
LocalDateTime end);
}
```

UserRepo.java

```
import com.kpi.diploma.model.entity.User;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

public interface UserRepo extends JpaRepository<User, Long> {

    Optional<User> findByUsername(String username);

    boolean existsByUsername(String username);

}
```

WarehouseRepo.java

```
import com.kpi.diploma.model.entity.InventoryItem;
import com.kpi.diploma.model.entity.Warehouse;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.List;
import java.util.Optional;

@Repository
public interface WarehouseRepo extends JpaRepository<Warehouse, Long> { }
```

ClientService.java

```
import com.kpi.diploma.model.entity.Client;
import com.kpi.diploma.model.entity.ClientInfo;
import com.kpi.diploma.model.entity.types.SatisfactionScale;
import com.kpi.diploma.repo.ClientInfoRepo;
import com.kpi.diploma.repo.ClientRepo;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Map;

@Service
@RequiredArgsConstructor
public class ClientService {

    private final ClientRepo clientRepo;
    private final ClientInfoRepo clientInfoRepo;

    public List<Client> getAllClientsByCompany(Long companyId) {
        return clientRepo.findByCompanyId(companyId);
    }

    public List<Client> getAllClients() {
        return clientRepo.findAll();
    }

    public Client getClient(Long id) {
        return clientRepo.findById(id)
                .orElseThrow(() -> new IllegalArgumentException("Клієнта не знайдено з id: "
+ id));
    }

    public Client saveClient(Client client) {
        return clientRepo.save(client);
    }

    public void deleteClient(Long id) {
        clientRepo.deleteById(id);
    }

    public ClientInfo getClientInfo(Long clientId) {
        return clientInfoRepo.findByClientId(clientId).orElse(null);
    }

    public ClientInfo saveClientInfo(ClientInfo info) {
        return clientInfoRepo.save(info);
    }

    public void deleteClientInfo(Long id) {
        clientInfoRepo.deleteById(id);
```

```
    }

    public boolean emailExists(String email) {
        return clientRepo.existsByEmail(email);
    }

    public boolean phoneExists(String phone) {
        return clientRepo.existsByPhone(phone);
    }

    public Map<String, Long> getFeedbackStats() {
        List<Client> clients = clientRepo.findAll();

        long veryHappy = clients.stream()
                .filter(c -> c.getSatisfactionScale() == SatisfactionScale.VERY_HAPPY)
                .count();

        long happy = clients.stream()
                .filter(c -> c.getSatisfactionScale() == SatisfactionScale.HAPPY)
                .count();

        long neutral = clients.stream()
                .filter(c -> c.getSatisfactionScale() == SatisfactionScale.NEUTRAL)
                .count();

        long unhappy = clients.stream()
                .filter(c -> c.getSatisfactionScale() == SatisfactionScale.UNHAPPY)
                .count();

        long veryUnhappy = clients.stream()
                .filter(c -> c.getSatisfactionScale() == SatisfactionScale.VERY_UNHAPPY)
                .count();

        return Map.of(
                "veryHappy", veryHappy,
                "happy", happy,
                "neutral", neutral,
                "unhappy", unhappy,
                "veryUnhappy", veryUnhappy
        );
    }


}
```

InventoryItemService.java

```
import com.kpi.diploma.model.entity.InventoryItem;
import com.kpi.diploma.repo.InventoryItemRepo;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;

@Service
public class InventoryItemService {

    private final InventoryItemRepo inventoryItemRepository;

    public InventoryItemService(InventoryItemRepo inventoryItemRepository) {
        this.inventoryItemRepository = inventoryItemRepository;
    }

    public List<InventoryItem> findAll() {
        return inventoryItemRepository.findAll();
```

```
    }

    public List<InventoryItem> findByWarehouseId(Long warehouseId) {
        return inventoryItemRepository.findByWarehouseId(warehouseId);
    }

    public InventoryItem save(InventoryItem item) {
        return inventoryItemRepository.save(item);
    }

    public Optional<InventoryItem> findById(Long id) {
        return inventoryItemRepository.findById(id);
    }

    public void delete(Long id) {
        inventoryItemRepository.deleteById(id);
    }
}
```

OrderService.java

```
import com.kpi.diploma.model.entity.*;
import com.kpi.diploma.model.entity.types.DeliveryType;
import com.kpi.diploma.model.entity.types.OrderType;
import com.kpi.diploma.repo.ClientRepo;
import com.kpi.diploma.repo.OrderItemRepo;
import com.kpi.diploma.repo.OrderRepo;
import com.kpi.diploma.repo.ProductRepo;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

@Service
@RequiredArgsConstructor
public class OrderService {

    private final OrderRepo orderRepo;
    private final OrderItemRepo orderItemRepo;
    private final ClientRepo clientRepo;
    private final ProductRepo productRepo;

    public List<Order> getAllOrdersByCompany(Long companyId) {
        return orderRepo.findByCompanyId(companyId);
    }

    public Optional<Order> getOrderById(Long id) {
        return orderRepo.findById(id);
    }

    public Order createOrder(Long clientId,
                             OrderType orderType,
                             DeliveryType deliveryType,
                             String additionalInfo,
                             List<Long> productIds,
                             List<Integer> quantities) {

        Client client = clientRepo.findById(clientId)
                .orElseThrow(() -> new IllegalArgumentException("Клієнт не знайдений"));

        Company company = client.getCompany();
        if (company == null) {
```

```
            throw new IllegalStateException("Клієнт не прив’язаний до компанії");
        }

        Order order = new Order();
        order.setClient(client);
        order.setOrderType(orderType);
        order.setDeliveryType(deliveryType);
        order.setAdditionalInfo(additionalInfo);
        order.setCompany(company);

        List<OrderItem> items = new ArrayList<>();
        for (int i = 0; i < productIds.size(); i++) {
            Product product = productRepo.findById(productIds.get(i))
                    .orElseThrow(() -> new IllegalArgumentException("Товар не знайдено"));

            Integer quantity = quantities.get(i);
            BigDecimal unitPrice = product.getPrice();

            OrderItem item = new OrderItem(order, product, quantity, unitPrice);
            items.add(item);
        }

        order.setItems(items);

        return orderRepo.save(order);
    }

    public void deleteOrder(Long id) {
        orderRepo.deleteById(id);
    }

    public void updateOrder(Long orderId,
                            Long clientId,
                            OrderType orderType,
                            DeliveryType deliveryType,
                            String additionalInfo,
                            List<Long> productIds,
                            List<Integer> quantities) {

        Order existingOrder = orderRepo.findById(orderId)
                .orElseThrow(() -> new IllegalArgumentException("Замовлення не знайдено"));

        Client client = clientRepo.findById(clientId)
                .orElseThrow(() -> new IllegalArgumentException("Клієнта не знайдено"));

        Company company = client.getCompany();
        if (company == null) {
            throw new IllegalStateException("Клієнт не має компанії");
        }

        existingOrder.setClient(client);
        existingOrder.setOrderType(orderType);
        existingOrder.setDeliveryType(deliveryType);
        existingOrder.setAdditionalInfo(additionalInfo);
        existingOrder.setCompany(company);

        List<OrderItem> updatedItems = new ArrayList<>();
        for (int i = 0; i < productIds.size(); i++) {
            Product product = productRepo.findById(productIds.get(i))
                    .orElseThrow(() -> new IllegalArgumentException("Товар не знайдено"));
            Integer quantity = quantities.get(i);
            BigDecimal unitPrice = product.getPrice();

            updatedItems.add(new OrderItem(existingOrder, product, quantity, unitPrice));
        }
```

```
        existingOrder.getItems().clear();
        existingOrder.getItems().addAll(updatedItems);

        orderRepo.save(existingOrder); // CascadeType.ALL оновить OrderItem
    }

}
```

TransactionService.java

```
import com.kpi.diploma.model.entity.Transaction;
import com.kpi.diploma.model.entity.types.TransactionType;
import com.kpi.diploma.repo.TransactionRepo;
import org.springframework.stereotype.Service;

import java.math.BigDecimal;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.temporal.ChronoUnit;
import java.util.*;
import java.util.stream.Collectors;

@Service
public class TransactionService {

    private final TransactionRepo transactionRepo;

    public TransactionService(TransactionRepo transactionRepo) {
        this.transactionRepo = transactionRepo;
    }

    public List<Transaction> findAll() {
        return transactionRepo.findAll();
    }

    public List<Transaction> findByCompanyId(Long companyId) {
        return transactionRepo.findByCompanyId(companyId);
    }

    public Optional<Transaction> findById(Long id) {
        return transactionRepo.findById(id);
    }

    public Transaction save(Transaction transaction) {
        return transactionRepo.save(transaction);
    }

    public void deleteById(Long id) {
        transactionRepo.deleteById(id);
    }

    public void updateTransaction(Long id, Transaction updatedTransaction) {
        Transaction existing = transactionRepo.findById(id)
                .orElseThrow(() -> new IllegalArgumentException("Транзакція не знайдена"));

        existing.setTransactionType(updatedTransaction.getTransactionType());
        existing.setAmount(updatedTransaction.getAmount());
        existing.setDescription(updatedTransaction.getDescription());
        existing.setCompany(updatedTransaction.getCompany());
        existing.setRelatedOrder(updatedTransaction.getRelatedOrder());
        existing.setDate(LocalDateTime.now());

        transactionRepo.save(existing);
    }
```

```
    public Map<String, List<Object>> getProfitVsExpense(String period, Long companyId) {
        LocalDateTime now = LocalDateTime.now();
        LocalDateTime start;

        switch (period.toUpperCase()) {
            case "WEEK":
                start = now.minusDays(7);
                break;
            case "MONTH":
                start = now.minusMonths(1);
                break;
            case "YEAR":
                start = now.minusYears(1);
                break;
            default:
                throw new IllegalArgumentException("Invalid period: " + period);
        }

        List<Transaction> transactions =
transactionRepo.findByCompanyIdAndDateBetween(companyId, start, now);

        Map<String, List<Object>> result = new HashMap<>();
        List<String> labels = new ArrayList<>();
        List<BigDecimal> profits = new ArrayList<>();
        List<BigDecimal> expenses = new ArrayList<>();

        Map<LocalDate, List<Transaction>> grouped = transactions.stream()
                .collect(Collectors.groupingBy(t -> t.getDate().toLocalDate()));
        grouped = new TreeMap<>(grouped);

        for (Map.Entry<LocalDate, List<Transaction>> entry : grouped.entrySet()) {
            labels.add(entry.getKey().toString());

            BigDecimal profit = entry.getValue().stream()
                    .filter(t -> t.getTransactionType() == TransactionType.INCOME)
                    .map(Transaction::getAmount)
                    .reduce(BigDecimal.ZERO, BigDecimal::add);

            BigDecimal expense = entry.getValue().stream()
                    .filter(t -> t.getTransactionType() == TransactionType.EXPENSE)
                    .map(Transaction::getAmount)
                    .reduce(BigDecimal.ZERO, BigDecimal::add);

            profits.add(profit);
            expenses.add(expense);
        }

        result.put("labels", new ArrayList<>(labels));
        result.put("profit", new ArrayList<>(profits));
        result.put("expense", new ArrayList<>(expenses));

        return result;
    }

}
```

UserService.java

```
import com.kpi.diploma.model.entity.Company;
import com.kpi.diploma.model.entity.User;
import com.kpi.diploma.model.entity.Role;
import com.kpi.diploma.model.entity.types.RoleType;
import com.kpi.diploma.repo.CompanyRepo;
import com.kpi.diploma.repo.RoleRepo;
```

```
import com.kpi.diploma.repo.UserRepo;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;

@Service
public class UserService {

    private final UserRepo userRepository;
    private final RoleRepo roleRepository;
    private final CompanyRepo companyRepository;
    private final PasswordEncoder passwordEncoder;

    public UserService(UserRepo userRepository,
                       RoleRepo roleRepository,
                       CompanyRepo companyRepository,
                       PasswordEncoder passwordEncoder) {
        this.userRepository = userRepository;
        this.roleRepository = roleRepository;
        this.companyRepository = companyRepository;
        this.passwordEncoder = passwordEncoder;
    }

    public void register(String username, String rawPassword, String email, String roleName,
Long companyId) {
        Role role = roleRepository.findByName(RoleType.valueOf(roleName))
                .orElseThrow(() -> new RuntimeException("Роль не знайдена"));

        Company company = companyRepository.findById(companyId)
                .orElseThrow(() -> new RuntimeException("Компанія не знайдена"));

        if (userRepository.existsByUsername(username)) {
            throw new RuntimeException("Користувач з таким іменем вже існує");
        }

        User user = new User();
        user.setUsername(username);
        user.setPassword(passwordEncoder.encode(rawPassword));
        user.setEmail(email);
        user.setRole(role);
        user.setCompany(company);

        userRepository.save(user);
    }
}
```

WarehouseService.java

```
import com.kpi.diploma.model.entity.InventoryItem;
import com.kpi.diploma.model.entity.Product;
import com.kpi.diploma.model.entity.StockMovement;
import com.kpi.diploma.model.entity.Warehouse;
import com.kpi.diploma.model.entity.types.MovementType;
import com.kpi.diploma.repo.InventoryItemRepo;
import com.kpi.diploma.repo.ProductRepo;
import com.kpi.diploma.repo.StockMovementRepo;
import com.kpi.diploma.repo.WarehouseRepo;
import jakarta.transaction.Transactional;
import org.springframework.stereotype.Service;

import java.time.LocalDateTime;
import java.util.List;
import java.util.Optional;

@Service
```

```
public class WarehouseService {

    private final InventoryItemRepo inventoryItemRepo;
    private final StockMovementRepo stockMovementRepo;
    private final ProductRepo productRepo;
    private final WarehouseRepo warehouseRepo;

    public WarehouseService(InventoryItemRepo inventoryItemRepo,
                            StockMovementRepo stockMovementRepo,
                            ProductRepo productRepo,
                            WarehouseRepo warehouseRepo) {
        this.inventoryItemRepo = inventoryItemRepo;
        this.stockMovementRepo = stockMovementRepo;
        this.productRepo = productRepo;
        this.warehouseRepo = warehouseRepo;
    }

    public List<InventoryItem> getAllInventory() {
        return inventoryItemRepo.findAll();
    }

    public Optional<InventoryItem> getInventoryItem(Long id) {
        return inventoryItemRepo.findById(id);
    }

    public void updateInventoryQuantities(List<Long> ids, List<Integer> quantities) {
        for (int i = 0; i < ids.size(); i++) {
            InventoryItem item = inventoryItemRepo.findById(ids.get(i)).orElseThrow();
            item.setQuantity(quantities.get(i));
            inventoryItemRepo.save(item);
        }
    }

    @Transactional
    public void processInMovement(Long productId, Long warehouseId, Integer quantity) {
        Product product = productRepo.findById(productId).orElseThrow();
        Warehouse warehouse = warehouseRepo.findById(warehouseId).orElseThrow();

        InventoryItem item = inventoryItemRepo
                .findByWarehouse(warehouse).stream()
                .filter(i -> i.getProduct().getId().equals(productId))
                .findFirst()
                .orElse(new InventoryItem(product, warehouse, 0));

        item.setQuantity(item.getQuantity() + quantity);
        inventoryItemRepo.save(item);

        StockMovement movement = new StockMovement(
                product,
                null,
                warehouse,
                quantity,
                MovementType.IN,
                LocalDateTime.now(),
                "Додавання товару"
        );
        stockMovementRepo.save(movement);
    }

    @Transactional
    public void processOutMovement(Long productId, Long warehouseId, Integer quantity) {
        Product product = productRepo.findById(productId).orElseThrow();
        Warehouse warehouse = warehouseRepo.findById(warehouseId).orElseThrow();

        InventoryItem item = inventoryItemRepo
```

```
                .findByWarehouse(warehouse).stream()
                .filter(i -> i.getProduct().getId().equals(productId))
                .findFirst()
                .orElseThrow(() -> new IllegalStateException("Товар не знайдено на
складі"));

        if (item.getQuantity() < quantity) {
            throw new IllegalStateException("Недостатньо товару для списання");
        }

        item.setQuantity(item.getQuantity() - quantity);
        inventoryItemRepo.save(item);

        StockMovement movement = new StockMovement(
                product,
                warehouse,
                null,
                quantity,
                MovementType.OUT,
                LocalDateTime.now(),
                "Списання товару"
        );
        stockMovementRepo.save(movement);
    }

    @Transactional
    public void processTransfer(Long productId, Long fromWarehouseId, Long toWarehouseId,
Integer quantity) {
        InventoryItem fromItem =
inventoryItemRepo.findByWarehouseIdAndProductId(fromWarehouseId, productId)
                .orElseThrow(() -> new IllegalArgumentException("Product not found in from-
warehouse"));

        if (fromItem.getQuantity() < quantity) {
            throw new IllegalArgumentException("Not enough stock in from-warehouse");
        }

        fromItem.setQuantity(fromItem.getQuantity() - quantity);
        inventoryItemRepo.save(fromItem);

        InventoryItem toItem =
inventoryItemRepo.findByWarehouseIdAndProductId(toWarehouseId, productId)
                .orElse(new InventoryItem(
                        fromItem.getProduct(),
                        warehouseRepo.findById(toWarehouseId).orElseThrow(),
                        0
                ));

        toItem.setQuantity(toItem.getQuantity() + quantity);
        inventoryItemRepo.save(toItem);

        StockMovement movement = new StockMovement(
                fromItem.getProduct(),
                fromItem.getWarehouse(),
                toItem.getWarehouse(),
                quantity,
                MovementType.TRANSFER,
                LocalDateTime.now(),
                "Auto transfer"
        );
        stockMovementRepo.save(movement);
    }

    public Optional<Warehouse> findById(Long id) {
        return warehouseRepo.findById(id);
```

```
    }
}
```

AuthApiController.java

```
import com.kpi.diploma.service.UserService;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api")
public class AuthApiController {

    private final UserService userService;

    public AuthApiController(UserService userService) {
        this.userService = userService;
    }

    @PostMapping("/register")
    public ResponseEntity<String> handleRegistration(
            @RequestParam String username,
            @RequestParam String password,
            @RequestParam(required = false) String email,
            @RequestParam String role,
            @RequestParam Long companyId
    ) {
        try {
            userService.register(username, password, email, role, companyId);
            return ResponseEntity.ok("Користувач успішно зареєстрований");
        } catch (RuntimeException e) {
            return ResponseEntity.badRequest().body("Помилка: " + e.getMessage());
        }
    }
}
```

AnalyticsController.java

```
import com.kpi.diploma.service.ClientService;
import com.kpi.diploma.service.TransactionService;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
@RequestMapping("/analytics")
public class AnalyticsController {

    private final TransactionService transactionService;
    private final ClientService clientService;

    public AnalyticsController(TransactionService transactionService, ClientService
clientService) {
        this.transactionService = transactionService;
        this.clientService = clientService;
    }

    @GetMapping
    public String showAnalyticsPage(@RequestParam(name = "companyId", required = false) Long
companyId, Model model) {
```

```
        if (companyId == null) {
            companyId = 1L; // тимчасовий ID або з твоєї бази
        }

        model.addAttribute("weeklyData", transactionService.getProfitVsExpense("WEEK",
companyId));
        model.addAttribute("monthlyData", transactionService.getProfitVsExpense("MONTH",
companyId));
        model.addAttribute("yearlyData", transactionService.getProfitVsExpense("YEAR",
companyId));

        model.addAttribute("feedbackStats", clientService.getFeedbackStats());

        return "analytics/analytics-view";
    }
}
```

ClientController.java

```
import com.kpi.diploma.model.entity.Client;
import com.kpi.diploma.model.entity.ClientInfo;
import com.kpi.diploma.model.entity.Company;
import com.kpi.diploma.model.entity.types.SatisfactionScale;
import com.kpi.diploma.service.ClientService;
import com.kpi.diploma.service.CompanyService;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.*;

import jakarta.validation.Valid;
import org.springframework.web.multipart.MultipartFile;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.UUID;

@Controller
@RequestMapping("/clients")
@RequiredArgsConstructor
public class ClientController {

    private final ClientService clientService;
    private final CompanyService companyService;

    @GetMapping
    public String listClients(Model model) {
        Long companyId = 1L;
        model.addAttribute("clients", clientService.getAllClients());
        return "clients/client-view";
    }

    @GetMapping("/new")
    public String showCreateForm(Model model) {
        model.addAttribute("client", new Client());
        model.addAttribute("satisfactionScale", SatisfactionScale.values());
        return "clients/client-form";
    }

    @PostMapping("/save")
    public String saveClient(@Valid @ModelAttribute Client client,
                             BindingResult result,
```

```
                        Model model) {

    if (client.getId() == null) {
        if (clientService.emailExists(client.getEmail())) {
            result.rejectValue("email", "error.client", "Email вже існує");
        }
        if (clientService.phoneExists(client.getPhone())) {
            result.rejectValue("phone", "error.client", "Телефон вже використовується");
        }
    }

    if (result.hasErrors()) {
        model.addAttribute("satisfactionScale", SatisfactionScale.values());
        return "clients/client-form";
    }

    clientService.saveClient(client);
    return "redirect:/clients";
}

@GetMapping("/edit/{id}")
public String showEditForm(@PathVariable Long id, Model model) {
    Client client = clientService.getClient(id);
    model.addAttribute("client", client);
    model.addAttribute("satisfactionScale", SatisfactionScale.values());
    return "clients/client-form";
}

@GetMapping("/delete/{id}")
public String deleteClient(@PathVariable Long id) {
    clientService.deleteClient(id);
    return "redirect:/clients";
}

@GetMapping("/details/{id}")
public String viewClientDetails(@PathVariable Long id, Model model) {
    Client client = clientService.getClient(id);
    ClientInfo info = clientService.getClientInfo(id);

    model.addAttribute("client", client);
    model.addAttribute("info", info);

    return "clients/client-details";
}

@PostMapping("/details/save")
public String updateClientInfo(@RequestParam Long clientId,
                               @RequestParam String detailsText,
                               @RequestParam("image") MultipartFile image) {
    Client client = clientService.getClient(clientId);

    ClientInfo info = clientService.getClientInfo(clientId);
    if (info == null) {
        info = new ClientInfo();
        info.setClient(client);
    }

    info.setDetailsText(detailsText);

    if (!image.isEmpty()) {
        try {
            String filename = UUID.randomUUID() + "_" + image.getOriginalFilename();
            Path path = Paths.get("src/main/resources/static/uploads/" + filename);
            Files.createDirectories(path.getParent());
            Files.write(path, image.getBytes());
```

```
                info.setImagePath("/uploads/" + filename);
            } catch (IOException e) {
                e.printStackTrace(); // продумай логування
            }
        }

        clientService.saveClientInfo(info);
        return "redirect:/clients/details/" + clientId;
    }

    @GetMapping("/company/{id}")
    public String viewCompany(@PathVariable Long id, Model model) {
        Company company = companyService.getCompany(id);
        model.addAttribute("company", company);
        return "clients/company-view"; // можна також назвати company-detail
    }
}
```

analytics-view.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Аналітика | EROMA</title>
    <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css">
    <link rel="stylesheet" th:href="@{/css/eroma-clients.css}">
    <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
</head>
<body class="eroma-bg">

<nav class="navbar fixed-top" style="background-color: rebeccapurple;">
    <div class="container-fluid d-flex justify-content-between align-items-center">
        <a href="/dashboard" class="navbar-brand eroma-title text-white mb-0" style="text-
decoration: none;">EROMA</a>
        <ul class="nav">
            <li class="nav-item"><a class="nav-link text-white eroma-link"
href="/clients">Клієнти</a></li>
            <li class="nav-item"><a class="nav-link text-white eroma-link"
href="/warehouse">Склад</a></li>
            <li class="nav-item"><a class="nav-link text-white eroma-link"
href="/orders">Облік замовлень</a></li>
            <li class="nav-item"><a class="nav-link eroma-link active-link"
href="/analytics">Аналітика</a></li>
            <li class="nav-item"><a class="nav-link text-white eroma-link"
href="/transactions">Транзакції</a></li>
            <li class="nav-item"><a class="nav-link text-white eroma-link"
href="/logout">Вийти</a></li>
        </ul>
    </div>
</nav>

<div class="container py-5 mt-5">
    <div class="card eroma-card p-4 shadow">

        <h4 class="eroma-section mb-4">📊 Аналітика компанії</h4>

        <ul class="nav nav-tabs mb-3" id="periodTabs">
            <li class="nav-item"><a class="nav-link active" data-bs-toggle="tab"
href="#week">Тиждень</a></li>
            <li class="nav-item"><a class="nav-link" data-bs-toggle="tab"
href="#month">Місяць</a></li>
            <li class="nav-item"><a class="nav-link" data-bs-toggle="tab"
href="#year">Рік</a></li>
```

```
        </ul>

        <div class="tab-content">
            <div class="tab-pane fade show active" id="week">
                <canvas id="weeklyChart"></canvas>
            </div>
            <div class="tab-pane fade" id="month">
                <canvas id="monthlyChart"></canvas>
            </div>
            <div class="tab-pane fade" id="year">
                <canvas id="yearlyChart"></canvas>
            </div>
        </div>

        <hr class="my-4">

        <h5 class="mb-3">💬 Відгуки клієнтів</h5>
        <canvas id="feedbackChart" width="400" height="300"></canvas>

    </div>
</div>

<script th:inline="javascript">
    /*<![CDATA[*/
    const weeklyData = /*[[${weeklyData}]]*/ {};
    const monthlyData = /*[[${monthlyData}]]*/ {};
    const yearlyData = /*[[${yearlyData}]]*/ {};
    const feedbackStats = /*[[${feedbackStats}]]*/ {};
    /*]]>*/
</script>

<script>
    function drawLineChart(canvasId, data) {
        const ctx = document.getElementById(canvasId).getContext('2d');
        new Chart(ctx, {
            type: 'line',
            data: {
                labels: data.labels,
                datasets: [
                    {
                        label: 'Прибуток',
                        data: data.profit,
                        borderColor: 'green',
                        backgroundColor: 'rgba(0, 128, 0, 0.2)',
                        fill: true,
                        tension: 0.3
                    },
                    {
                        label: 'Витрати',
                        data: data.expense,
                        borderColor: 'red',
                        backgroundColor: 'rgba(255, 0, 0, 0.2)',
                        fill: true,
                        tension: 0.3
                    }
                ]
            },
            options: {
                responsive: true,
                plugins: {
                    legend: { position: 'top' },
                    title: { display: true, text: 'Прибуток vs Витрати' }
                }
            }
        });
```

```
    }

    drawLineChart('weeklyChart', weeklyData);
    drawLineChart('monthlyChart', monthlyData);
    drawLineChart('yearlyChart', yearlyData);

    new Chart(document.getElementById("feedbackChart"), {
        type: 'doughnut',
        data: {
            labels: ['Very Happy', 'Happy', 'Neutral', 'Unhappy', 'Very Unhappy'],
            datasets: [{
                data: [
                    feedbackStats.veryHappy,
                    feedbackStats.happy,
                    feedbackStats.neutral,
                    feedbackStats.unhappy,
                    feedbackStats.veryUnhappy
                ],
                backgroundColor: [
                    '#28a745', '#85d68e', '#ffc107', '#fd7e14', '#dc3545'
                ],
                hoverOffset: 10
            }]
        },
        options: {
            responsive: true,
            plugins: {
                legend: { position: 'bottom' },
                title: { display: true, text: 'Client Feedback Breakdown' }
            }
        }
    });
</script>

<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js"></script>
</body>
</html>
```