

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ**

КВАЛІФІКАЦІЙНА РОБОТА
на тему: «Система менеджменту товарів для оптимізації
мережі магазинів.
Спец-частина: розробка Backend-частини системи»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

*Кваліфікаційна робота містить результати власних досліджень. Використання
ідей, результатів і текстів інших авторів мають посилання
на відповідне джерело*

_____ Валерія ЛАПУЦЬКА
(підпис)

Виконала: здобувачка вищої освіти групи ПД-41

_____ Валерія ЛАПУЦЬКА

Керівник: _____
д-р техн. наук, проф.

_____ Ірина ЗАМРІЙ

Рецензент: _____

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ**

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

Інженерії програмного забезпечення

_____ Ірина ЗАМРІЙ

« ____ » _____ 2026 р.

**ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ**

Лапуцькій Валерії Анатоліївні

1. Тема кваліфікаційної роботи: «Система менеджменту товарів для оптимізації мережі магазинів. Спец-частина: розробка Backend-частини системи»
керівник кваліфікаційної роботи д-р техн. наук, професор Ірина ЗАМРІЙ,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 року № 51.

2. Строк подання кваліфікаційної роботи «29» травня 2026 року.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про архітектурні підходи до розробки корпоративного програмного забезпечення (REST API), опис методів автоматизації складського обліку та управління внутрішньою логістикою мережі магазинів, технічна документація з описом технологій розробки серверної частини (Java, Spring Boot, реляційних баз даних PostgreSQL та протоколів безпеки JWT).

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

1. Аналіз предметної області управління внутрішньою логістикою мережевого ритейлу, дослідження існуючих комерційних інформаційних систем та формування вимог до програмного продукту.

2. Обґрунтування вибору архітектурних підходів, мови програмування, технологій зберігання даних (PostgreSQL) та інструментарію (Spring Boot, Docker) для реалізації серверної частини системи.
 3. Проектування реляційної моделі бази даних, багаторівневої архітектури REST API, моделі безпеки на основі ролей (RBAC) та алгоритмів життєвого циклу замовлень застосунку «LVStore».
 4. Програмна реалізація розробленої бекенд-архітектури, налаштування ізольованого середовища розгортання та комплексне тестування компонентів і бізнес-логіки системи.
5. Перелік графічного матеріалу: *презентація*
1. Аналіз аналогів та комерційних систем.
 2. Вимоги до програмного забезпечення.
 3. Програмні засоби реалізації.
 4. Діаграма класів модуля безпеки.
 5. ER-діаграма реляційної бази даних системи.
 6. UML-діаграми послідовності базових процесів системи.
 7. Тестування REST API (створення замовлення).
 8. Екранні форми.
 9. Апробація результатів дослідження.
6. Дата видачі завдання «23» лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	23.02-28.02.2026	
2	Аналіз та дослідження існуючих аналогів	01.03-07.03.2026	
3	Аналіз технологічних засобів та обґрунтування вибору стеку для реалізації серверної частини системи управління товарообігом	08.03-15.03.2026	
4	Проектування архітектури бази даних, REST API та механізмів безпеки застосунку «LVStore»	16.03-30.03.2026	
5	Програмна реалізація Backend частини системи «LVStore» та налаштування середовища розгортання	31.03-20.04.2026	
6	Тестування програмних модулів, перевірка безпеки та транзакційної бази даних	21.04-28.04.2026	
7	Оформлення роботи: вступ, висновки, реферат	29.04-05.05.2026	

8	Розробка демонстраційних матеріалів	06.05-10.05.2026	
9	Попередній захист роботи	11.05-22.05.2026	

Здобувачка вищої освіти

(підпис)

Валерія ЛАПУЦЬКА

Керівник
кваліфікаційної роботи

(підпис)

Ірина ЗАМРІЙ

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 68 стор., 1 табл., 20 рис., 25 джерел.

Мета роботи – оптимізація процесів контролю товарних запасів та обробки замовлень для вирішення проблеми неузгодженості дій між торговими точками та складами, а також запобігання товарному дефіциту.

Об'єкт дослідження – процес управління товарообігом, формування замовлень та контролю товарних залишків у мережах роздрібно́ї торгівлі.

Предмет дослідження – методи, засоби та технології для побудови автоматизованої інформаційної системи управління мережею магазинів.

Короткий зміст роботи: в роботі проаналізовано предметну область управління внутрішньою логістикою та проблематику складського обліку. Проведено аналіз існуючих комерційних систем (Торгсофт, WooCommerce, Loyverse POS) та виявлено їхні архітектурні недоліки в контексті корпоративного використання. Спроектовано архітектуру та розроблено надійну серверну частину (REST API) системи менеджменту товарів «LVStore». Програмно реалізовані ключові функціональні можливості: жорстке багаторівнєве розмежування прав доступу (RBAC), ізольований контроль товарних залишків, маршрутизація та багатокрокова валідація внутрішніх замовлень. Проведено тестування API-ендпоінтів, бізнес-логіки та транзакційної цілісності бази даних. В роботі використано мову програмування Java, фреймворк Spring Boot для побудови модульного бекенду, реляційну СУБД PostgreSQL із застосуванням Hibernate ORM, технологію JWT для безпечної безстейтової автентифікації, а також інструменти Swagger та Docker.

Сферою використання застосунку є управління внутрішньою логістикою, контроль складського обліку та автоматизація операційних процесів у мережах роздрібно́ї торгівлі.

КЛЮЧОВІ СЛОВА: УПРАВЛІННЯ ТОВАРООБІГОМ, ВНУТРІШНЯ ЛОГІСТИКА, REST API, SPRING BOOT, POSTGRESQL, RBAC, JWT, СКЛАДСЬКИЙ ОБЛІК.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	10
ВСТУП.....	11
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ.....	14
1.1 Роль автоматизації у сучасному мережевому ритейлі	14
1.1.1 Еволюція облікових систем у роздрібній торгівлі	14
1.1.2 Вплив автоматизації на операційну ефективність та логістику.....	15
1.1.3 Архітектурні та технологічні виклики масштабування.....	16
1.2 Особливості процесів управління внутрішньою логістикою та організаційна структура.....	18
1.2.1 Моделювання фізичного та інформаційного руху товарів.....	18
1.2.2 Життєвий цикл внутрішнього замовлення.....	19
1.2.3 Організаційна ієрархія та функціональні ролі персоналу	20
1.3 Проблематика управління товарними залишками та замовленнями	21
1.3.1 Критичні наслідки децентралізованого контролю запасів	21
1.3.2 Комунікаційні розриви та вразливість до людського фактора	22
1.3.3 Архітектурні недоліки локальних інструментів обліку	23
1.4 Аналіз аналогів програмного забезпечення	24
1.4.1 Аналіз програмного комплексу «Торгсофт».....	24
1.4.2 Аналіз платформи електронної комерції WooCommerce	26
1.4.3 Аналіз хмарної системи обліку Loyverse POS	28
1.4.4 Узагальнення результатів аналізу та обґрунтування розробки системи LVStore	30
1.5 Постановка задачі та формування вимог до програмного продукту.....	32
1.5.1 Функціональні вимоги.....	33
1.5.2 Нефункціональні вимоги.....	34
2 ЗАСОБИ РЕАЛІЗАЦІЇ АВТОМАТИЗОВАНОЇ СИСТЕМИ ДЛЯ УПРАВЛІННЯ ТОВАРООБІГОМ.....	36
2.1 Обґрунтування вибору архітектурного підходу (REST) та мови програмування (Java)	37

2.2 Фреймворки екосистеми Spring для побудови бізнес-логіки та безпеки.....	38
2.3 Технології зберігання, об'єктно-реляційного відображення та кешування даних.....	40
2.4 Інструментарій для розробки, тестування, документування та контейнеризації	42
3 ПРОЄКТУВАННЯ АВТОМАТИЗОВАНОЇ СИСТЕМИ ДЛЯ УПРАВЛІННЯ ТОВАРООБІГОМ “LVSTORE”	44
3.1 Структура бази даних та архітектура сутностей	44
3.2 Проєктування архітектурних шарів системи (MVC-патерн)	47
3.3 Проєктування системи безпеки та авторизації	48
3.4 Моделювання бізнес-процесів (сценарії використання).....	50
3.4.1 Адміністрування каталогу.....	51
3.4.2 Життєвий цикл внутрішнього замовлення.....	53
3.5 Проєктування REST API та форматів обміну даними (DTO)	55
4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ	57
4.1 Структура проєкту	57
4.2 Реалізація основного функціоналу та бізнес-логіки.....	60
4.2.1 Реалізація механізму автентифікації та розв'язання токенів.....	61
4.2.2 Реалізація бізнес-логіки управління замовленнями	62
4.2.3 Реалізація REST-контролерів	63
4.3 Взаємодія з базою даних PostgreSQL.....	64
4.4 Розгортання та конфігурація середовища (Docker).....	66
4.5 Тестування системи	68
4.5.1 Модульне та інтеграційне тестування	68
4.5.2 Ручне тестування та документація API	70
4.6 Демонстрація роботи системи	72
ВИСНОВКИ.....	76
ПЕРЕЛІК ПОСИЛАНЬ	79
ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ	82
ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ.....	92

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

СУБД – Система управління базами даних

ACID – Atomicity, Consistency, Isolation, Durability (атомарність, консистентність, ізолюваність, довговічність – набір властивостей транзакцій баз даних)

API – Application Programming Interface (інтерфейс прикладного програмування)

B2B – Business-to-Business (бізнес для бізнесу)

B2C – Business-to-Consumer (бізнес для споживача)

DTO – Data Transfer Object (об'єкт передачі даних)

ER-діаграма – Entity-Relationship diagram (діаграма «сутність-зв'язок»)

ERP – Enterprise Resource Planning (планування ресурсів підприємства)

HTTP – Hypertext Transfer Protocol (протокол передачі гіпертексту)

JSON – JavaScript Object Notation (текстовий формат обміну даними)

JPA – Java Persistence API (специфікація Java для збереження та управління об'єктно-реляційним відображенням)

JVM – Java Virtual Machine (віртуальна машина Java)

JWT – JSON Web Token (відкритий стандарт створення криптографічно підписаних токенів доступу)

MVC – Model-View-Controller (архітектурний патерн «Модель-Вигляд-Контролер»)

ORM – Object-Relational Mapping (об'єктно-реляційне відображення)

RBAC – Role-Based Access Control (управління доступом на основі організаційних ролей)

REST – Representational State Transfer (архітектурний стиль проектування розподілених систем)

SaaS – Software as a Service (програмне забезпечення як послуга)

SQL – Structured Query Language (мова структурованих запитів до реляційних баз даних)

UML – Unified Modeling Language (уніфікована мова графічного опису для моделювання програмного забезпечення)

ВСТУП

Актуальність теми зумовлена постійним зростанням обсягів роздрібної торгівлі та ускладненням процесів управління ланцюгами постачань. В умовах активного розвитку мережевого ритейлу постає гостра необхідність в оптимізації процесів внутрішньої логістики, розподілу товарних запасів та керування замовленнями між центральними складами і торговими точками. Використання універсальних комерційних систем або застарілих монолітних рішень часто призводить до функціонального перевантаження, ускладнює масштабування та створює залежність від закритої інфраструктури постачальників (vendor lock-in). Відсутність спеціалізованих, легковагових інструментів для внутрішнього складського обліку критично ускладнює організацію операційних процесів та підвищує ризики виникнення товарного дефіциту.

Розробка модульної серверної частини (Backend) інформаційної системи дозволяє вирішити ці проблеми, забезпечуючи гнучке, безстейтове та високопродуктивне середовище для управління товарообігом. Вона дозволяє ефективно контролювати залишки ізольовано для кожного підрозділу, автоматизувати маршрутизацію внутрішніх замовлень та впровадити жорстку ієрархію прав доступу персоналу. Враховуючи потребу сучасного бізнесу в надійних кросплатформних рішеннях, розробка такого програмного продукту є надзвичайно актуальною.

Об'єктом дослідження є процес управління товарообігом, формування замовлень та контролю товарних залишків у мережах роздрібної торгівлі.

Предметом дослідження є методи, засоби та технології для побудови автоматизованої інформаційної системи управління мережею магазинів.

Метою роботи є удосконалення процесів контролю товарних запасів та обробки замовлень для вирішення проблеми неузгодженості дій між торговими точками та складами, а також запобігання товарному дефіциту.

Методи дослідження: першим етапом дослідження було вивчення бізнес-процесів управління внутрішньою логістикою, зокрема формування заявок на поповнення запасів та алгоритмів списання товарів. Далі проведено критичний огляд існуючих комерційних засобів автоматизації (Торгсофт, WooCommerce, Loyverse POS) з метою виявлення їхніх архітектурних недоліків та формування точних технічних вимог до майбутньої системи.

На основі сформованих вимог було обрано сучасний технологічний стек для реалізації серверного ядра: мову програмування Java та фреймворк Spring Boot для реалізації бізнес-логіки, реляційну СУБД PostgreSQL із застосуванням Hibernate ORM для управління базами даних, модуль Spring Security та стандарт JWT для безпечної безстейтової автентифікації, а також інструментарій Docker для контейнеризації. Розробку REST API здійснено з урахуванням принципів принципів ACID, масштабованості та ізолюваності. Тестування та документування системи здійснювалося із застосуванням інструментів Postman та Swagger (OpenAPI).

Наукова новизна роботи полягає у проєктуванні спеціалізованої серверної архітектури, яка розв'язує проблему функціонального перевантаження (feature bloat), притаманного універсальним ERP-системам, шляхом фокусування виключно на процесах внутрішньої логістики мережевого ритейлу. Реалізовано функціонал, що дозволяє:

- здійснювати ізолюваний моніторинг товарних залишків у розрізі кожної окремої торгової точки та центрального складу;
- будувати жорстку багаторівневу модель рольового доступу (RBAC) для лінійного персоналу, адміністраторів магазинів та менеджерів складу;
- автоматизувати багатокрокову валідацію внутрішніх замовлень із застосуванням алгоритму зміни статусів (State Machine);
- гарантувати транзакційну цілісність даних під час синхронного нарахування та списання товарів на рівні бази даних;
- забезпечити абсолютну портативність рішення незалежно від апаратного забезпечення завдяки використанню REST-архітектури та контейнеризації.

Платформа забезпечує адаптивну та надійну структуру управління логістичною діяльністю, що оптимізує внутрішні взаємодії без використання зайвих комерційних модулів.

Практична значущість результатів полягає у можливості безпосереднього впровадження розробленого програмного забезпечення «LVStore» у реальні бізнес-процеси мереж роздрібної торгівлі для ефективного менеджменту товарообігу. Система дозволяє не просто автоматизувати рутинні складські операції, а й створити єдиний прозорий інформаційний простір для синхронної взаємодії різних структурних підрозділів підприємства. Завдяки ізольованому контролю залишків та алгоритмізації процесів, розробка мінімізує негативний вплив людського фактора, унеможливорює несанкціоноване втручання в облікові дані та суттєво підвищує швидкість обробки внутрішніх замовлень. Це дозволяє підприємству превентивно запобігати операційним помилкам і критичному товарному дефіциту на полицях фізичних магазинів.

Завдяки використанню актуальних архітектурних підходів, реляційної бази даних та ізольованого середовища виконання, розроблена серверна частина повністю позбавлена проблеми прив'язки до конкретної апаратної інфраструктури. Додаток може бути за лічені хвилини розгорнутий на будь-якому сучасному хмарному хостингу без необхідності закупівлі та налаштування дороговартісного серверного обладнання. Це робить платформу економічно привабливою та високорентабельною для бізнесу, IT-адміністрування гарантує повну готовність системи до миттєвого горизонтального масштабування при збільшенні кількості магазинів або розширенні загальної мережі.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Роль автоматизації у сучасному мережевому ритейлі

Сучасна сфера роздрібної торгівлі (ритейл) є надзвичайно складною організаційною системою, що об'єднує в єдину мережу фізичні магазини, складські приміщення, логістичні маршрути та персонал. В основі успішного функціонування такої мережі лежить ефективний менеджмент товарів – безперервний процес планування, обліку та контролю за рухом продукції на кожному етапі її життєвого циклу всередині компанії.

Сьогодні традиційна модель ведення торговельного бізнесу, що базувалася на паперовому документообігу та ручній інвентаризації, є абсолютно неконкурентоспроможною. На зміну їй прийшла концепція цифрової трансформації (Digital Transformation), яка передбачає глибоку інтеграцію інформаційних технологій у всі сфери діяльності підприємства – від касового обслуговування кінцевого споживача до автоматизації складської логістики. Вирішення проблеми управління товарообігом у мережевому ритейлі пройшло еволюційний шлях від повністю ручних методів обліку до впровадження комплексних розподілених інформаційних систем.

Автоматизація в цьому контексті виступає не просто допоміжним інструментом, а фундаментальним стабілізатором бізнес-процесів, що забезпечує високу швидкість обробки даних, масштабованість інфраструктури та транзакційну цілісність логістичних операцій.

1.1.1 Еволюція облікових систем у роздрібній торгівлі

Історично базова практика управління товарообігом, яка подекуди й досі зустрічається в малому бізнесі, спирається на відмову від спеціалізованого програмного забезпечення. Основними інструментами в такому підході є звичайні електронні таблиці (наприклад, Microsoft Excel), паперові накладні та телефонна

комунікація. Для поодиноких торгових точок з невеликим асортиментом такий метод був прийнятним, однак при масштабуванні бізнесу до рівня мережі з кількома магазинами він виявляє свою абсолютну неефективність.

З точки зору системного аналізу та інженерії програмного забезпечення, використання електронних таблиць та локальних файлів має низку критичних архітектурних недоліків:

- відсутність синхронізації даних у реальному часі, локальні електронні таблиці зазвичай оновлюються періодично (наприклад, наприкінці робочої зміни). Це унеможливорює оперативне реагування на товарний дефіцит та створює явище «розсинхронізації» даних між складом та магазином;

- вразливість до людського фактора, процеси ручного перенесення даних з паперових носіїв до електронних реєстрів неминуче призводять до втрати інформації, дублювання замовлень або помилок у розрахунках;

- відсутність механізмів інформаційної безпеки у файлових системах та електронних таблицях неможливо налаштувати гранулярний (детальний) доступ до даних. Через це будь-який працівник може випадково або навмисно змінити чи видалити критичні фінансові показники [1].

З огляду на ці проблеми, сучасна інженерія програмного забезпечення та теорія проєктування інформаційних систем розглядають управління складськими запасами через призму побудови клієнт-серверних архітектур. Мережевий ритейл потребує відмови від «монолітних» локальних рішень на користь розподілених систем на базі REST API, адже вони дозволяють централізувати базу даних та бізнес-логіку на єдиному сервері, забезпечуючи єдине джерело істини для всіх структурних підрозділів підприємства.

1.1.2 Вплив автоматизації на операційну ефективність та логістику

Ефективний контроль товарообігу є ключовим фактором конкурентоспроможності роздрібної мережі. При ручному, фрагментованому або застарілому управлінні підприємство неминуче стикається з комунікаційними

розривами та помилками ручного обліку. Використання електронних таблиць або паперових журналів генерує неточності безпосередньо через людський фактор. Це регулярно спричиняє розсинхронізацію інформаційних потоків: наприклад, виникають ситуації, коли центральний склад уже відвантажив партію товару, але магазин ще не отримав інформацію про це через затримки в обміні даними. Відсутність автоматизованих алгоритмів контролю призводить до двох критичних логістичних проблем.

Першою з них є товарний дефіцит (out-of-stock) – ситуація, коли попит на певний товар є, але його фізично немає в наявності у магазині. Це призводить не лише до прямої втрати доходу підприємства, але й до стрімкого зниження лояльності покупців, які змушені переходити до конкурентів [2]. Для вирішення цієї проблеми сучасні корпоративні системи використовують модулі автоматизованого моніторингу. Такі програмні компоненти здатні алгоритмічно відстежувати критичний рівень запасів як у конкретних магазинах, так і на центральному складі, дозволяючи системі превентивно реагувати на загрозу товарного дефіциту.

Другою, протилежною за характером проблемою є надлишок залишків (overstock). Акумуляція зайвих товарів на складах призводить до «замороження» оборотного капіталу підприємства, збільшення витрат на зберігання та підвищення ризику псування продукції. Оптимізація цих операційних витрат реалізується шляхом усунення паперового документообігу та автоматизації типових рішень, зокрема через впровадження алгоритмічного сповіщення персоналу про реальну необхідність поповнення запасів.

1.1.3 Архітектурні та технологічні виклики масштабування

Розширення роздрібної мережі ставить перед підприємством нові технологічні виклики, які неможливо вирішити за допомогою застарілих інженерних підходів. Традиційною моделлю розробки облікових систем була монолітна архітектура, де інтерфейс користувача, бізнес-логіка та механізми роботи з даними об'єднані в єдиний виконуваний файл або проєкт. Хоча такий підхід суттєво спрощує початкову розробку та локальне розгортання, зі зростанням кодової бази та збільшенням

кількості територіально розподілених філій він критично ускладнює подальше масштабування та технічну підтримку програмного продукту. Зважаючи на ці обмеження, сучасні наукові та практичні підходи наголошують на необхідності відмови від «монолітних» локальних рішень.

З точки зору сучасної інженерії програмного забезпечення та теорії проектування інформаційних систем, управління складськими запасами розглядається через призму побудови клієнт-серверних архітектур на базі REST API. Такий підхід (API-first design) дозволяє централізувати бізнес-логіку на єдиному сервері, забезпечуючи при цьому повну незалежність клієнтських застосунків, зокрема вебінтерфейсів, мобільних додатків чи спеціалізованих терміналів збору даних. Модульний бекенд на базі REST API є найпопулярнішим рішенням на сучасному ринку корпоративного програмного забезпечення, де серверна частина виступає незалежним централізованим ядром, яке обробляє бізнес-логіку та взаємодіє з базою даних [3]. Спілкування системи з будь-якими зовнішніми клієнтами відбувається через стандартизовані HTTP-запити з використанням універсального формату обміну даними JSON.

Окремим напрямком наукомістких досліджень у галузі проектування корпоративних систем є забезпечення безперервності та безпеки даних. При інтеграції кількох територіально розподілених підрозділів (магазинів і складів) виникає гостра потреба у захисті даних від несанкціонованого доступу та збереженні транзакційної цілісності. Сучасним стандартом розв'язання проблеми безпеки в REST-архітектурах є відмова від класичних серверних сесій на користь безстейтових (stateless) протоколів автентифікації. Використання криптографічно підписаних токенів (наприклад, стандарту JWT) у поєднанні з жорсткою моделлю управління доступом на основі ролей (RBAC) гарантує, що кожна операція в системі буде авторизована та надійно задокументована. Даний підхід дозволяє суттєво знизити навантаження на апаратні ресурси сервера та забезпечує високу відмовостійкість системи при одночасній роботі великої кількості територіально віддалених філій.

1.2 Особливості процесів управління внутрішньою логістикою та організаційна структура

Для того щоб інформаційна система ефективно вирішувала проблеми підприємства, вона не може існувати в ізольованому технологічному вакуумі. Архітектура й логіка роботи повинна бути точним цифровим відображенням реальних фізичних процесів, матеріальних потоків і організаційної ієрархії торговельної мережі [4]. Моделювання цих процесів на етапі аналізу предметної області дозволяє виявити операційні «вузькі місця» та перетворити їх на чіткі алгоритми для подальшої програмної автоматизації. Лише за умови повної відповідності цифрової моделі до реальної структури бізнесу програмний продукт здатний забезпечити коректний контроль товарообігу на кожному етапі життєвого циклу товару всередині компанії.

1.2.1 Моделювання фізичного та інформаційного руху товарів

Організаційна модель сучасного мережевого ритейлу найчастіше будується за радіальною схемою (hub-and-spoke). Центром цієї логістичної системи виступає головний офіс та центральний розподільчий склад, навколо яких розташовані торгові точки (магазини), що безпосередньо взаємодіють із кінцевим споживачем. У такій моделі фізичний «шлях товару» є строго регламентованим і починається з його надходження від зовнішнього постачальника на центральний склад, де продукція ставиться на загальний баланс підприємства. Далі, відповідно до сформованих потреб конкретних торгових точок, товар розподіляється та транспортується до локальних магазинів, де він виставляється на полиці. Фінальним етапом цього ланцюга є продаж товару кінцевому споживачу, що означає його остаточне вибуття з матеріального балансу мережі.

Для забезпечення безперебійності цього процесу взаємодія між вузлами логістичної мережі відбувається у двох зустрічних напрямках. Перший напрямок – інформаційний (знизу вгору, від магазину до центру), який включає передачу аналітичних даних про фактичні продажі, поточні залишки на полицях та ініціацію

запитів на поповнення товару. Другий напрямок – матеріально-управлінський (зверху вниз, від центру до магазину), що передбачає безпосереднє постачання фізичних товарів, оновлення загальної номенклатури у базі даних та директивну зміну цінової політики. Головна проблема неавтоматизованого бізнесу полягає в тому, що на кожному з етапів цих зустрічних потоків дані про товар можуть губитися, тому інформаційна система повинна супроводжувати об'єкт віртуально на кожному його фізичному кроці.

1.2.2 Життєвий цикл внутрішнього замовлення

Основою взаємодії між магазином і розподільчим складом є процес формування внутрішнього замовлення. Для ефективної автоматизації фізична потреба торгової точки в товарі повинна бути трансформована у чіткий цифровий алгоритм. З інженерної точки зору, управління життєвим циклом такого замовлення в інформаційній системі найкраще реалізується через архітектурний патерн кінцевого автомата (State Machine) або патерн зміни станів [5]. У межах розроблюваної системи цей бізнес-процес розбито на чіткі послідовні етапи, кожен з яких супроводжується жорстко регламентованою зміною статусу замовлення:

- формування потреби та створення заявки (процес ініціюється безпосередньо в магазині, після чого замовлення реєструється в базі даних та отримує первинний статус «Нове»);

- затвердження адміністратором (валідація) (для уникнення помилкових або надлишкових замовлень, створена заявка обов'язково потрапляє на перевірку до керівника магазину. Адміністратор оцінює доцільність запиту та підтверджує його, внаслідок чого статус у системі змінюється на «Підтверджене»);

- комплектація на складі та відвантаження (затверджена заявка миттєво стає видимою для менеджерів центрального складу, які перевіряють фізичну наявність необхідного товару, збирають його та готують до відправки. Після того як товар фізично завантажено в транспорт, менеджер складу переводить замовлення у статус «Відправлене»);

– прийом товару та оновлення залишків (коли замовлений товар фізично прибуває до цільового магазину, адміністратор фіксує факт його отримання у системі. Інформаційна система автоматично змінює статус на фінальний – «Отримане», і в цю ж мить транзакційно додає відповідну кількість віртуального товару до балансу залишків цього конкретного магазину).

1.2.3 Організаційна ієрархія та функціональні ролі персоналу

Для створення ефективної інформаційної системи необхідно, щоб її архітектура, зокрема механізми авторизації та контролю доступу, точно відображала реальну організаційну структуру підприємства. Типова мережа роздрібної торгівлі має чітко виражену ієрархічну побудову, яка базується на делегуванні повноважень та розподілі зон відповідальності. Щоб інформаційна система працювала безперебійно, вона має підлаштовуватися під реальну ієрархію підприємства, що в контексті інженерії програмного забезпечення є базисом для побудови рольової моделі управління доступом (Role-Based Access Control) [6]. Відповідно до організаційної структури мережі, виділяються три ключові посади з чітко розмежованими зонами інформаційної та матеріальної відповідальності:

– лінійний продавець (рівень торгового залу), виступає матеріально відповідальною особою на місцях, зона контролю якого обмежується виключно фізичним торговим залом конкретного магазину. До його прямих обов'язків входить обслуговування покупців, візуальний контроль за наповненістю вітрин та стелажів, а також своєчасне виявлення нестачі певних позицій для ініціації процесу внутрішньої закупівлі.

– адміністратор магазину (рівень локального управління) є керівником торгової точки, який відповідає за рентабельність об'єкта та роботу лінійного персоналу. Організаційно він виступає «фільтром» між своїм магазином та центральним офісом, а його головна функція в логістичному ланцюгу полягає у фінансовому та раціональному контролі. Він має оцінити доцільність замовлень, які роблять продавці, затвердити їх у межах бюджету магазину, а згодом – організувати фізичний прийом машини з товаром від логістів.

– менеджер складу (рівень центральної логістики) – це співробітник розподільчого центру, який не має прив'язки до жодного з магазинів і взаємодіє виключно з гуртовими партіями товарів. Організаційно він керує центральними запасами підприємства, а його завдання полягає у фізичному виконанні узгоджених заявок: пошук потрібних палет чи коробок на складських стелажах, організація комплектації (збірки) замовлення та передача товару транспортному відділу для доставки в роздрібну мережу.

1.3 Проблематика управління товарними залишками та замовленнями

1.3.1 Критичні наслідки децентралізованого контролю запасів

Децентралізований підхід до управління запасами означає, що кожна торгова точка або регіональний склад веде власну, ізольовану базу даних чи локальний реєстр. Головним критичним наслідком такої організації обліку є повна відсутність «єдиного джерела істини» (Single Source of Truth) для всього підприємства. За таких умов керівництво мережі та центральний логістичний відділ не мають можливості аналізувати глобальну картину товарообігу в режимі реального часу [7].

Фрагментованість інформаційних потоків призводить до того, що регулярна процедура інвентаризації та звірки залишків перетворюється на тривалий та ресурсомісткий процес ручної агрегації розрізнених файлів від різних магазинів. Крім того, інформаційна ізоляція фактично паралізує горизонтальну взаємодію між структурними підрозділами мережі. Наприклад, одна торгова точка не може оперативно перевірити наявність необхідної позиції в найближчому магазині для того, щоб перенаправити клієнта або ініціювати швидке внутрішнє переміщення товару.

Як наслідок, в умовах децентралізованого контролю всі управлінські та логістичні рішення приймаються на основі даних, які втратили свою актуальність ще на етапі їх збору та консолідації, що робить неможливим ефективне масштабування роздрібною мережі.

1.3.2 Комунікаційні розриви та вразливість до людського фактора

Використання неформалізованих каналів обміну інформацією, таких як телефонні узгодження, побутові месенджери чи паперові накладні, руйнує цілісність логістичного ланцюга підприємства. Основний інженерний недолік такого підходу полягає у відсутності безперервного цифрового сліду (audit trail) транзакцій. Коли своєчасна передача критичних даних щодо переміщення матеріальних цінностей залежить від уважності чи пам'яті конкретного працівника, процеси стають максимально вразливими до людського фактора, що унеможлиблює побудову надійної системи звітності.

Найбільш деструктивним наслідком таких комунікаційних розривів є феномен інформаційної розсинхронізації – утворення тривалого часового лагу (затримки) між виконанням реальної фізичної дії та її відображенням у системі обліку. Класичним проявом цієї проблеми є ситуація, коли необхідну партію товару вже фізично скомплектовано та відвантажено з центрального розподільчого центру в транспортний засіб, але на інформаційному рівні цільовий магазин цього ще не бачить. Через те, що відповідальна особа не внесла дані до реєстрів миттєво, утворюється операційна "сліпа зона". Під час цього періоду підприємство оперує хибними даними: фізичний об'єкт перебуває в стані транзиту, проте логічно він не зафіксований належним чином ані за відправником, ані за отримувачем.

В інфраструктурі, де відсутній прозорий механізм автоматичної зміни статусів замовлень (від ініціації до фінального отримання), подібна розсинхронізація системно провокує аномалії у вигляді втрати або дублювання заявок. Не маючи можливості в режимі реального часу відстежити поточний етап обробки свого запиту, адміністратор магазину може хибно припустити, що первинну заявку було проігноровано. Це спонукає його генерувати повторний документ на поповнення тих самих позицій, що генерує штучне навантаження на склад, подвійну комплектацію та, зрештою, логістичний колапс через надлишкове постачання на локальний об'єкт.

1.3.3 Архітектурні недоліки локальних інструментів обліку

Використання локальних електронних таблиць (наприклад, Microsoft Excel) як ядра корпоративної системи обліку є фундаментальною архітектурною помилкою. Попри свою гнучкість для потреб малого бізнесу, такі інструменти не є повноцінними системами керування базами даних (СКБД) і концептуально не призначені для роботи в розподілених мережах. Спроба масштабувати їх під логістичні завдання стикається з двома критичними технологічними обмеженнями.

Першою і найвагомішою проблемою є неможливість забезпечення одночасної транзакційної роботи кількох користувачів через відсутність механізмів конкурентного доступу (concurrency control) та базових принципів ACID (Atomicity, Consistency, Isolation, Durability). Електронні таблиці спираються на файлову систему зберігання, що зазвичай призводить до блокування всього документа під час його редагування одним користувачем. У реаліях мережевої торгівлі, коли кілька магазинів та склад намагаються синхронно оновити залишки, це неминуче призводить до створення конфліктних копій файлу або втрати інформації. Без строгих ACID-гарантій система не здатна забезпечити транзакційну цілісність, тобто гарантувати, що списання товару зі складу та його нарахування на баланс магазину відбудуться як єдина та безпомилкова операція [8].

Другою критичною вразливістю є повна відсутність спеціалізованих механізмів інформаційної безпеки. Інструменти локального обліку оперують поняттям доступу до файлу в цілому, що унеможливлює налаштування гранулярних (деталізованих) прав для різних категорій працівників. За таких умов технічно неможливо побудувати жорстку багаторівневу модель рольового доступу (Role-Based Access Control, RBAC). Як наслідок, лінійний продавець, отримавши доступ до файлу для фіксації локальних продажів, фактично отримує можливість переглядати, редагувати або навіть безповоротно видаляти критичні фінансові дані всього підприємства. Відсутність програмних обмежень та ізольованого середовища виконання робить корпоративну інформацію абсолютно незахищеною від випадкового пошкодження або цілеспрямованого втручання з боку персоналу.

1.4 Аналіз аналогів програмного забезпечення

Сучасний ринок інформаційних технологій пропонує широкий спектр програмних рішень для управління торгівлею та логістикою – від простих локальних утиліт для малого бізнесу до глобальних хмарних ERP-систем. Однак, незважаючи на таку різноманітність, вибір оптимального інструменту для конкретної мережі магазинів залишається складною задачею. Кожне готове рішення має власну архітектуру, цінову політику та рівень гнучкості, що не завжди відповідає специфічним бізнес-процесам конкретного підприємства.

1.4.1 Аналіз програмного комплексу «Торгсофт»

Першим об'єктом дослідження обрано програмний комплекс «Торгсофт» – одне з найпоширеніших на вітчизняному ринку десктопних рішень для автоматизації роздрібної торгівлі [9]. Дана система позиціонується розробниками як універсальний інструмент класу «все в одному», що забезпечує комплексне управління складом, ведення фінансового обліку, роботу з фіскальним обладнанням та взаємодію з клієнтською базою. З архітектурної точки зору, програма є класичним монолітним додатком типу «товстий клієнт», орієнтованим на встановлення в межах локальних мереж підприємства.

Розглядаючи переваги даного програмного забезпечення, доцільно відзначити високий рівень надійності та безпеки обробки комерційної інформації. Відповідно до визначених критеріїв аналізу, система володіє вбудованою жорсткою багаторівневою рольовою моделлю доступу. Цей механізм дозволяє системним адміністраторам детально налаштовувати права для кожної категорії персоналу, що ефективно захищає базу даних від несанкціонованого втручання або випадкового видалення критичних записів некомпетентними працівниками. Крім того, завдяки тривалому терміну присутності на ринку, система демонструє високу стабільність роботи при виконанні базових облікових операцій.

Разом з тим, детальний аналіз користувацького інтерфейсу та функціонального наповнення на рисунку 1.1 виявляє низку суттєвих недоліків у контексті завдань, що стосуються виключно внутрішньої логістики мережі.

Реалізація № 7 від 28/05/2021

Файл Вид Допомога

ШК/Арт. № Диск. Дата продажу 28.05.2021 23

Реалізація товару в роздріб № 7 від 28/05/2021

Інформація про клієнта

Клієнт <Клієнта не визначено> БОРГ Сума для розрахунку знижки

Знижка Ост. придбання Поч. сума

Придбання Повернення Зниження

Останній чек: До оплати 0,00 Отримано 0,00 Решта 0,00 Товари чека

Назва товару	Штрих-код	Кількість	Ціна за прайсом	Ціна	Сума	% знижки	Вид знижки	Артикул	Підготовка това
Ножиці	2973090000404	1	13,50	13,50	13,50	0%		230584	

Артикулів 1 Кількість 1,00 Сума 13,50

Оплатити (F6) Прийняти замовл. Видати замовлення Вибір зі складу Видати з каси Товарний звіт Звіт по касі

Повна сума 13,50грн Сума до оплати 13,50грн

Немає знижки 0% Фото (Ctrl+N)

Сума знижки 0,00грн

Передплата 0,00грн

Запитувати кількість товару

Рис.1.1 Екранна форма робочого місця у системі «Торгсофт»

Екранна форма додатку критично перевантажена елементами управління, більшість з яких орієнтована на безпосереднє обслуговування роздрібних покупців, розрахунків знижок та проведення касових операцій. Для лінійного персоналу торгового залу або менеджерів логістичного центру, чийм першочерговим завданням є системний моніторинг залишків та формування заявок на поповнення, такий інтерфейс є надмірно складним та вимагає тривалого часу на навчання. Спеціалізація на внутрішній логістиці в цій системі реалізована лише частково, оскільки складський модуль виступає другорядною підсистемою загального фінансово-касового комплексу. Це призводить до того, що процеси автоматизації внутрішніх замовлень між структурними підрозділами мережі не є повноцінно оптимізованими. Як доводять сучасні дослідження ергономіки корпоративного програмного забезпечення [10], надмірне когнітивне навантаження (cognitive load) у

неспеціалізованих інтерфейсах не лише критично збільшує час адаптації персоналу, але й експоненціально підвищує ризик операційних помилок при виконанні рутинних складських завдань.

Найбільш критичним технологічним недоліком комплексу «Торгсофт» є його архітектурна прив'язка до операційної системи Windows та повна відсутність портативності. Монолітна структура десктопного додатка унеможливорює використання сучасних підходів до ізольованого розгортання, зокрема технології контейнеризації. У випадку масштабування торгової мережі, ІТ-спеціалістам необхідно здійснювати ручне фізичне встановлення та налаштування програмного забезпечення на кожному новому робочому місці. Такий підхід суттєво знижує гнучкість інформаційної інфраструктури підприємства, збільшує витрати на технічну підтримку та робить неможливою швидку міграцію серверної частини на сучасні хмарні платформи. Академічна спільнота підтверджує [11], що підтримка таких застарілих (legacy) монолітних систем генерує значний технічний борг, тоді як перехід до API-орієнтованих архітектур із використанням контейнеризації є ключовою умовою для забезпечення масштабованості та відмовостійкості ІТ-інфраструктури сучасного бізнесу.

1.4.2 Аналіз платформи електронної комерції WooCommerce

Другим об'єктом дослідження виступає WooCommerce – одне з найпопулярніших у світі відкритих веб-орієнтованих рішень, яке функціонує як розширення для систем управління контентом (CMS) [12]. На відміну від попереднього аналога, дана платформа від самого початку проєктувалася як інструмент для електронної комерції (e-commerce), орієнтований на сектор B2C (Business-to-Consumer). З технологічної точки зору, система побудована на стеку PHP та MySQL, що забезпечує їй базовий рівень портативності та можливість розгортання на більшості сучасних хмарних хостингів.

До безумовних переваг даного програмного забезпечення належить його кросплатформність: оскільки інтерфейс реалізовано у вигляді веб-додатка, користувачі можуть отримати доступ до системи з будь-якого пристрою через

стандартний браузер. Згідно з результатами порівняльного аналізу, платформа забезпечує частковий захист даних та пропонує базові інструменти для управління товарною номенклатурою. Зокрема, система здатна відстежувати загальну кількість товарів на віртуальному складі (статус «In stock») та фіксувати зміни залишків під час здійснення онлайн-продажів.

Однак, незважаючи на популярність рішення, аналіз його архітектури та користувацького інтерфейсу на рисунку 1.2 виявляє критичну невідповідність вимогам корпоративної логістики мережевого підприємства.

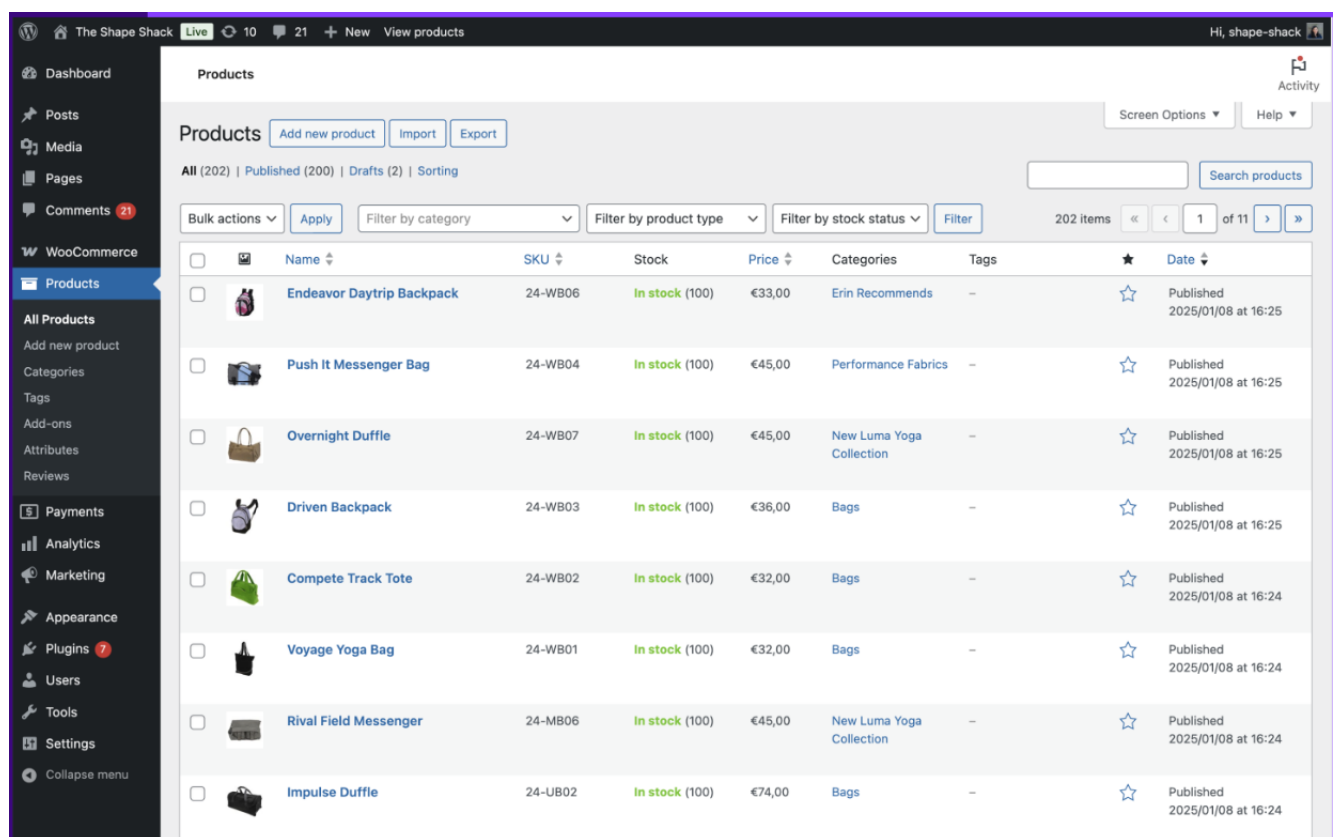


Рис. 1.2 Екранна форма панелі управління запасами у платформі
WooCommerce

Як видно з головного навігаційного меню на рисунку 1.2, архітектура системи сфокусована на зовнішніх продажах: наявні розділи «Маркетинг» (Marketing), «Зовнішній вигляд» (Appearance) та система коментарів. Для завдань внутрішньої складської логістики цей функціонал є абсолютно надлишковим. Спеціалізація на

внутрішніх процесах тут реалізована лише частково. У базовій комплектації платформи повністю відсутній механізм автоматизації внутрішніх замовлень: лінійний продавець не має вбудованого інструментарію для формування заявки на переміщення товару з центрального розподільчого центру до свого конкретного магазину.

Найголовнішим недоліком WooCommerce у контексті корпоративного використання є відсутність жорсткої багаторівневої рольової моделі (RBAC), адаптованої під складську ієрархію. Базова система пропонує лише стандартизовані ролі для управління контентом вітрини. Щоб налаштувати гранулярний доступ (розділити права продавців, адміністраторів магазинів та логістів) або додати функціонал внутрішніх накладних, підприємство змушене інтегрувати десятки сторонніх модулів (плагінів). Такий підхід – використання плагінної архітектури від різних розробників – критично знижує стабільність системи, підвищує ризик конфліктів у коді та робить базу даних вразливою до втрати або пошкодження інформації при оновленнях. Сучасні дослідження в галузі інженерії програмного забезпечення підтверджують небезпеку такого підходу. Зокрема, академічний аналіз вразливостей CMS-систем [13] доводить, що надмірна залежність від екосистеми сторонніх розширень експоненціально збільшує поверхню кібератак та є головним фактором порушення цілісності даних і виникнення критичних конфліктів сумісності.

1.4.3 Аналіз хмарної системи обліку Loyverse POS

Третім об'єктом дослідження є Loyverse POS – сучасна хмарна інформаційна система, яка постачається за моделлю SaaS (Software as a Service) та орієнтована переважно на автоматизацію касових операцій (Point of Sale) у сфері малого бізнесу [14]. На відміну від десктопного «Торгсофт» або плагінної архітектури WooCommerce, дане рішення від самого початку створювалося з фокусом на мобільність та використання на портативних пристроях (планшетах, смартфонах).

Розглядаючи переваги даного програмного засобу, необхідно відзначити сучасний, мінімалістичний та інтуїтивно зрозумілий користувацький інтерфейс. Як

наочно демонструє екранна форма управління запасами на рисунку 1.3, система не перевантажена зайвими елементами управління, що суттєво зменшує поріг входження для лінійного персоналу.

Важливою функціональною перевагою є наявність вбудованої системи візуального сповіщення про критичний рівень залишків товарів. Крім того, відповідно до критеріїв порівняльного аналізу, система забезпечує надійний захист даних від випадкового видалення, оскільки всі транзакції та зміни залишків автоматично синхронізуються та резервуються на хмарних серверах постачальника послуги.

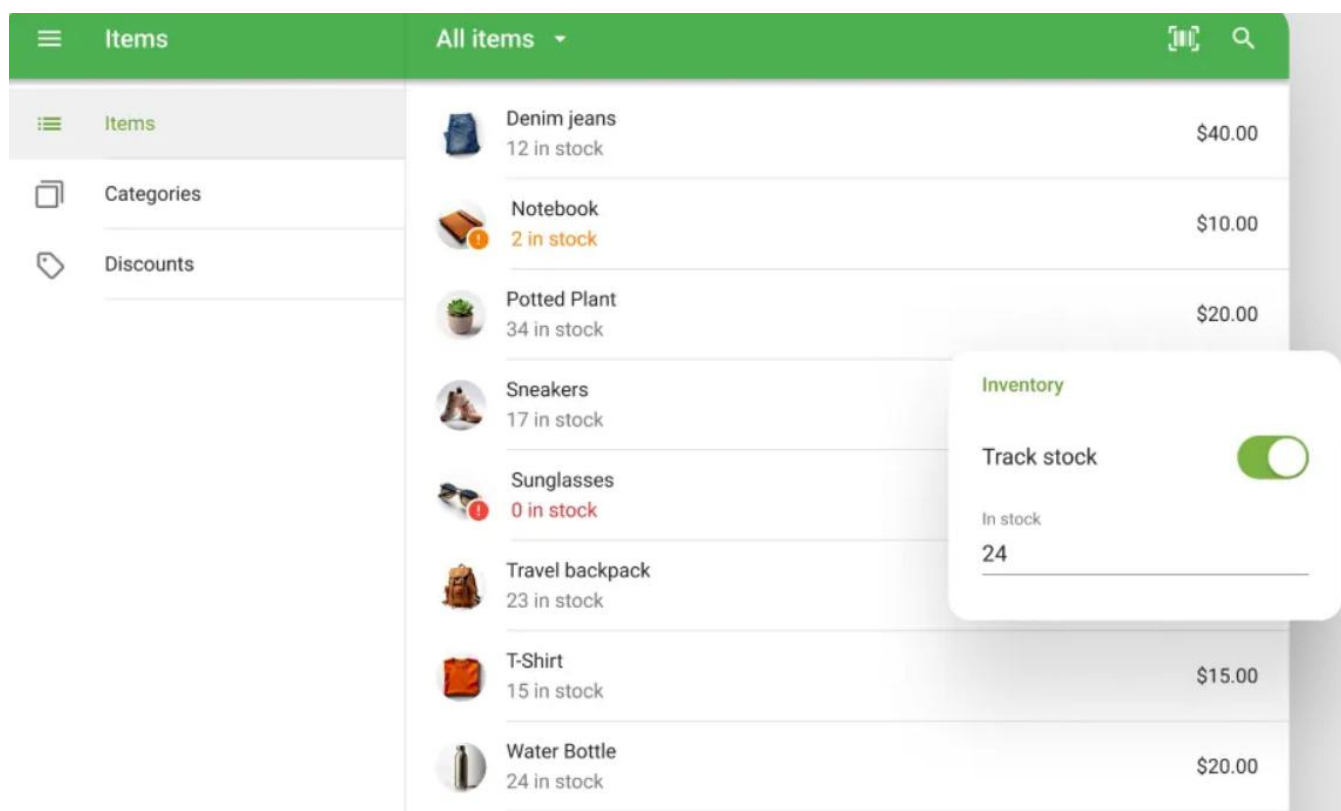


Рис. 1.3 Екранна форма моніторингу залишків у системі Loyverse POS

Проте, незважаючи на зручність інтерфейсу, функціонал системи виявляється недостатнім для забезпечення повноцінного управління товарообігом розгалуженої торгової мережі. Спеціалізація на внутрішній логістиці та механізми автоматизації замовлень реалізовані в системі лише частково. Як видно з інтерфейсу на рисунку 1.3, управління запасами зводиться до простого перемикача «Відстежувати запаси»

(Track stock) та можливості ручного коригування кількості. Програма не підтримує комплексного життєвого циклу внутрішнього замовлення: продавець не може сформулювати запит, який пройде етапи затвердження адміністратором та подальшого відвантаження менеджером складу. Ситуація ускладнюється відсутністю жорсткої багаторівневої рольової моделі (RBAC) – зазвичай такі системи мають лише базові ролі «Власник» та «Касир», що не відповідає складній організаційній структурі мережевого ритейлу.

Найбільш критичним інженерним недоліком Loyverse POS є повна відсутність портативності та ізолюваності розгортання. Оскільки продукт є пропрієтарним хмарним SaaS-рішенням, підприємство не є власником серверної частини (бекенду) і не має прямого доступу до архітектури бази даних. Це створює класичну проблему технологічної прив'язки до вендора (vendor lock-in). Підприємство позбавлене можливості розгорнути систему на власних ізолюваних серверах, запакувати її в Docker-контейнери або інтегрувати з власними нестандартними аналітичними модулями, що робить такі рішення неприйнятними для побудови незалежної корпоративної інформаційної інфраструктури.

1.4.4 Узагальнення результатів аналізу та обґрунтування розробки системи LVStore

Узагальнення результатів дослідження існуючих програмних продуктів дозволяє систематизувати їхні технічні та функціональні характеристики. Для об'єктивної оцінки відповідності розглянутих аналогів специфічним вимогам управління мережевим ритейлом сформовано порівняльну матрицю ключових показників таблиця 1.1.

Аналіз наведених показників свідчить, що більшість готових ринкових рішень орієнтовані переважно на обслуговування кінцевого споживача (B2C) або зосереджені на загальному фінансово-касовому обліку. Зокрема, критерії «Спеціалізація на внутрішній логістиці» та «Автоматизація внутрішніх замовлень» реалізовані в системах Торгсофт, WooCommerce та Loyverse POS лише частково, або ж потребують встановлення сторонніх модулів. Натомість архітектура розробленої

системи LVStore від самого початку спроектована для задоволення потреб виключно внутрішньої логістики. Це забезпечує повний та безперебійний цикл автоматизації руху товарів між центральним складом та магазинами без перевантаження інтерфейсу зайвим функціоналом.

Таблиця 1.1

Порівняльна характеристика програмних рішень для управління
товарообігом

Показник	Торгсофт	WooCommerce	Loyverse POS
Спеціалізація на внутрішній логістиці	Частково	Частково	Частково
Автоматизація внутрішніх замовлень	Частково	Ні	Частково
Жорстка багаторівнева рольова модель	Так	Ні	Ні
Захист даних від випадкового видалення	Так	Частково	Так
Портативність та ізольованість розгортання	Ні	Частково	Ні

Обмеженість універсальних облікових систем у розв'язанні вузькоспеціалізованих завдань є предметом активних дискусій у сучасній інженерії. Як зазначається в актуальних дослідженнях корпоративної цифровізації [15], комерційні ERP-комплекси та платформи загального призначення переважно орієнтовані на фінансово-адміністративний менеджмент, через що вони часто не здатні ефективно обробляти специфічні операційні процеси компанії. Саме тому

розробка кастомних (custom-built) програмних рішень, що мають модульну архітектуру та спеціалізуються на конкретних логістичних потребах підприємства, визнається значно ефективнішим підходом порівняно зі спробами адаптації готового «коробкового» (off-the-shelf) програмного забезпечення.

З точки зору корпоративної безпеки, критичним фактором для мережевого підприємства є наявність жорсткої багаторівневої ролівої моделі. Дослідження показує, що вебплатформа WooCommerce та хмарний сервіс Loyverse POS не здатні забезпечити гранулярне розмежування доступу для складної ієрархії персоналу (продавець, адміністратор, логіст). І хоча комплекс Торгсофт частково задовольняє цю вимогу, розроблений бекенд LVStore пропонує більш сучасний підхід: він гарантує найвищий рівень захисту даних від випадкового видалення чи несанкціонованого втручання завдяки інтеграції передових алгоритмів безпеки на рівні API. Визначальним недоліком досліджуваних комерційних систем є їхня технологічна негнучкість. Десктопні рішення (Торгсофт) позбавлені портативності через прив'язку до конкретної операційної системи, а SaaS-продукти (Loyverse POS) створюють залежність бізнесу від закритої хмарної інфраструктури постачальника. Як зазначається в сучасних дослідженнях архітектури корпоративного програмного забезпечення [16], уникнення явища жорсткої прив'язки до постачальника послуг (vendor lock-in) є одним із головних викликів для сучасного бізнесу. Проект LVStore концептуально розв'язує цю проблему: використання принципів REST API та контейнеризації забезпечує абсолютну портативність та ізольованість розгортання. Серверна частина може бути оперативно розгорнута на будь-яких потужностях незалежно від апаратного забезпечення.

1.5 Постановка задачі та формування вимог до програмного продукту

На основі проведеного аналізу предметної області, виявлених логістичних проблем та критичної оцінки існуючих комерційних аналогів, можна сформулювати кінцеву мету розробки. Вона полягає у створенні спеціалізованої серверної частини (Backend) інформаційної системи «LVStore», призначеної для оптимізації процесів

контролю товарних запасів та безперебійної обробки замовлень між торговими точками та складами підприємства. Для досягнення цієї мети розроблюваний програмний продукт повинен відповідати чітко регламентованому переліку вимог.

1.5.1 Функціональні вимоги

Функціональні вимоги визначають безпосередню поведінку розроблюваної системи, структуру даних та алгоритми бізнес-логіки, які вона повинна реалізовувати для забезпечення безперебійної роботи мережі:

- автентифікація та авторизація адже система повинна забезпечувати захищений процес входу за допомогою унікальних облікових даних (логін та пароль) із подальшою видачею токена доступу. Кожен запит до серверної частини має супроводжуватися перевіркою цього токена для ідентифікації користувача. Авторизаційний модуль повинен блокувати будь-які спроби несанкціонованого доступу, гарантуючи, що функціонал системи доступний виключно верифікованим працівникам підприємства;

- розмежування прав доступу (Role-Based Access Control), програмний продукт повинен підтримувати жорстку ієрархічну рольову модель, яка диференціює повноваження на три ключові рівні. Рівень «Продавець» передбачає обмежений доступ (читання глобального каталогу та право ініціації заявок лише для своєї торгової точки). Рівень «Адміністратор магазину» розширює ці права можливістю управління локальним персоналом, фінансового затвердження (валідації) створених продавцями заявок та фіксації прийому товару. Рівень «Менеджер складу» надає глобальний доступ до моніторингу всіх розподільчих центрів мережі, а також право змінювати статуси виконання консолідованих замовлень;

- моніторинг товарної номенклатури та залишків для забезпечення оперативного доступу до єдиного централізованого каталогу товарів (Single Source of Truth), де зберігаються уніфіковані характеристики кожної позиції (назва, категорія, ціна, унікальний штрихкод). Система повинна дозволяти користувачам у режимі реального часу переглядати фактичні кількісні залишки об'єктів, причому

область видимості жорстко обмежується правами доступу: працівники бачать лише баланс свого структурного підрозділу, тоді як логістичний центр має доступ до глобальної картини запасів усієї мережі;

– обробка та менеджмент замовлень для підтримки повного та безперервного життєвого циклу внутрішнього переміщення товарів на основі алгоритму кінцевого автомата (State Machine). Процес включає такі етапи: створення запиту лінійним продавцем (статус «Нове»), перевірка доцільності керівником точки (статус «Підтверджене»), резервування товару та його фізичне відвантаження на розподільчому центрі (статус «Відправлене»), і нарешті, фактичний прийом партії в цільовому магазині (статус «Отримане»). Система повинна строго контролювати послідовність цих переходів;

– автоматизований транзакційний контроль запасів (реалізація механізмів нарахування та списання товарно-матеріальних цінностей на рівні бази даних. У момент, коли замовлення переходить у статус «Відправлене», система повинна автоматично списувати вказану кількість одиниць із балансу центрального складу. Відповідно, при набутті статусу «Отримане» алгоритм має миттєво додати ці позиції на баланс цільового магазину. Такий алгоритмічний контроль виключає можливість дублювання записів та дозволяє превентивно реагувати на загрозу товарного дефіциту).

1.5.2 Нефункціональні вимоги

Нефункціональні вимоги визначають атрибути якості програмного продукту, встановлюючи строгі критерії щодо його продуктивності, безпеки, масштабованості та архітектурної стабільності в умовах корпоративного використання:

– продуктивність та транзакційна надійність – система повинна забезпечувати високу швидкість обробки даних, при якій час виконання стандартних REST API запитів до серверної частини не перевищує 200 мілісекунд. Оскільки система оперує критичними логістичними даними, на рівні реляційної бази даних (PostgreSQL) гарантується суворе виконання принципів ACID (Атомарність, Консистентність, Ізольованість, Довговічність). Це архітектурне рішення унеможливорює втрату,

дублювання або часткове збереження інформації у разі виникнення апаратних чи програмних збоїв;

– безпека та криптографічний захист, архітектура безпеки має будуватися на сучасній концепції безстейтових сесій (Stateless), що знімає навантаження з оперативної пам'яті сервера. Процес автентифікації повинен бути захищений стандартом JWT (JSON Web Token) із застосуванням надійного асиметричного криптографічного алгоритму RSA-2048, який використовує пари приватних та публічних ключів для безкомпромісного захисту підпису токенів. Додатково, захист бази даних від розповсюджених кібератак (зокрема SQL-ін'єкцій) має забезпечуватися автоматично на рівні використання сучасного ORM-фреймворку (Hibernate);

– сумісність, інтеграція та ізольоване розгортання, взаємодія між клієнтськими додатками та серверною частиною повинна відбуватися виключно через стандартизований архітектурний стиль REST API з використанням універсального формату обміну даними JSON. Це гарантує повну незалежність бекенду від типу клієнтського інтерфейсу (фронтенду). Крім того, серверна частина має проектуватися з урахуванням хмарно-орієнтованої архітектури та бути повністю ізольованою у Docker-контейнері, що дозволить швидко розгортати та безперешкодно масштабувати систему на будь-яких серверах;

– оптимізація навантаження для забезпечення відмовостійкості системи під час пікових навантажень (наприклад, при масовому перегляді залишків популярних товарів одночасно з кількох магазинів) архітектура повинна передбачати механізм зниження навантаження на основну базу даних. Ця вимога реалізується через інтеграцію системи In-Memory кешування, що дозволить миттєво віддавати дані, які найчастіше запитуються, з оперативної пам'яті сервера.

2 ЗАСОБИ РЕАЛІЗАЦІЇ АВТОМАТИЗОВАНОЇ СИСТЕМИ ДЛЯ УПРАВЛІННЯ ТОВАРООБІГОМ

Розробка сучасного корпоративного застосунку вимагає ретельно підбраного технологічного стеку, здатного забезпечити виконання як функціональних, так і нефункціональних вимог проєкту. У межах створення серверної частини (Backend) автоматизованої системи управління товарообігом «LVStore» було використано низку перевірених індустріальних технологій та інструментів, що сприяли побудові надійної, безпечної та масштабованої архітектури.

Основу платформи становить об'єктно-орієнтована мова програмування Java та фреймворк Spring Boot, які забезпечують гнучкий і швидкий спосіб створення модульного RESTful API. Для реалізації надійної безстейтової (stateless) автентифікації та авторизації застосовано модуль Spring Security у поєднанні зі стандартом JWT (JSON Web Token), що дозволило впровадити жорстке розмежування прав доступу (RBAC) відповідно до ієрархії персоналу торговельної мережі.

У якості основної системи збереження логістичних та фінансових даних обрано реляційну СУБД PostgreSQL, яка гарантує виконання принципів ACID для транзакційних складських операцій. Взаємодія між об'єктно-орієнтованим кодом та базою даних реалізована через ORM-фреймворк Hibernate. Для оптимізації навантаження та кешування частих запитів до каталогу товарів інтегровано In-Memory сховище Redis. Безпосереднє проєктування та написання коду здійснювалося в сучасному інтегрованому середовищі розробки IntelliJ IDEA.

Для тестування розроблених API-ендпоінтів використовувалася платформа Postman, що забезпечило перевірку коректності HTTP-запитів на всіх етапах реалізації бізнес-логіки. Автоматизоване документування програмних інтерфейсів реалізовано за допомогою специфікації Swagger (OpenAPI). Завершальним етапом інженерного циклу стала підготовка системи до розгортання: проєкт було

контейнеризовано за допомогою технології Docker, що гарантує його портативність та незалежність від апаратного забезпечення цільового сервера.

У наступних підрозділах детально розглядається кожен із компонентів обраного стеку, їхня архітектурна роль та особливості інтеграції у процес створення програмного забезпечення для системи «LVStore».

2.1 Обґрунтування вибору архітектурного підходу (REST) та мови програмування (Java)

Проектування сучасних корпоративних систем (Enterprise applications) вимагає чіткого розділення програмного продукту на клієнтську та серверну частини. Історично традиційним підходом була монолітна архітектура, де інтерфейс користувача, бізнес-логіка та механізми роботи з даними об'єднані в єдиний виконуваний файл. Хоча такий підхід спрощує початкову розробку, зі зростанням кодової бази він критично ускладнює подальше масштабування та технічну підтримку. З іншого боку, мікросервісна архітектура, яка декомпозує систему на десятки незалежних дрібних сервісів, є надлишковою для систем середнього розміру, оскільки створює надмірну складність інфраструктури та підвищує витрати на її адміністрування.

Зважаючи на ці фактори, для розробки інформаційної системи «LVStore» було обрано концепцію модульного бекенду на базі архітектурного стилю REST (Representational State Transfer), що є оптимальним компромісним рішенням на сучасному ринку корпоративного програмного забезпечення. Такий підхід (API-first design) дозволяє централізувати базу даних та складну бізнес-логіку на єдиному незалежному сервері, забезпечуючи при цьому повну незалежність клієнтських застосунків. Взаємодія між модульним бекендом та будь-якими зовнішніми клієнтами (вебінтерфейсами, мобільними додатками або терміналами збору даних) відбувається через стандартизовані HTTP-запити з використанням універсального формату обміну даними JSON. Це гарантує, що у разі необхідності оновлення або повної заміни клієнтського інтерфейсу, серверна логіка залишиться недоторканою.

У якості основної мови програмування для реалізації модульного REST API було обрано об'єктно-орієнтовану мову Java. На сьогоднішній день Java є беззаперечним галузевим стандартом для розробки високонавантажених корпоративних рішень у секторі B2B (Business-to-Business). Вибір цієї мови обґрунтовується двома критичними інженерними перевагами.

Першою перевагою є суворі статична типізація мови Java. Вона дозволяє розробнику виявляти переважну більшість архітектурних помилок та несумісностей типів даних ще на етапі компіляції коду, що є фундаментально важливим для фінансових та складських систем, де помилка під час виконання (runtime error) може призвести до спотворення даних про залишки товарів.

Другою, не менш важливою перевагою є абсолютна кросплатформність мови, яка забезпечується віртуальною машиною JVM (Java Virtual Machine). Завдяки JVM написаний компільований байт-код відокремлюється від апаратного забезпечення, що дозволяє виконувати розроблений додаток на серверах під управлінням операційних систем Linux, Windows або macOS без жодної необхідності переписування вихідного коду [17]. Це повністю нівелює проблему прив'язки системи до конкретної інфраструктури, що була виявлена під час аналізу комерційних аналогів у попередньому розділі.

2.2 Фреймворки екосистеми Spring для побудови бізнес-логіки та безпеки

Для прискорення процесу розробки серверної частини та уникнення написання надлишкового шаблонного коду базовим каркасом було обрано фреймворк Spring Boot. На відміну від класичного фреймворку Spring, який традиційно вимагає складного налаштування масивних XML-конфігурацій, Spring Boot базується на інженерній парадигмі «Convention over Configuration» (угода дорожче за конфігурацію). Він автоматично налаштовує більшість базових компонентів системи та містить під капотом вбудований вебсервер (Apache Tomcat), що позбавляє необхідності зовнішнього конфігурування сервера додатків для розгортання REST API. Крім того, завдяки потужному механізму впровадження

залежностей (Dependency Injection), фреймворк забезпечує слабку зв'язність (loose coupling) між ключовими архітектурними шарами системи – контролерами, сервісами та репозиторіями. Це робить програмний код модульним, легко тестованим та придатним до подальшої безпечної модифікації без ризику руйнування існуючої бізнес-логіки.

Оскільки однією з критичних функціональних вимог до системи «LVStore» є суворе багаторівневе розмежування прав доступу персоналу (RBAC), для управління безпекою було інтегровано спеціалізований модуль Spring Security. Зважаючи на те, що сучасні REST-орієнтовані архітектури повинні відповідати принципу безстейтовості (stateless), розробка передбачала повну відмову від використання класичних серверних сесій (Session ID). Замість них механізм автентифікації та подальшої авторизації був побудований на базі відкритого індустріального стандарту JWT (JSON Web Token).

Інтеграція стандарту JWT дозволяє безпечно упаковувати та передавати ідентифікаційні дані користувача, включно з його актуальною організаційною роллю (наприклад, лінійний продавець чи менеджер складу), у вигляді компактного криптографічно підписаного рядка. Головна інженерна перевага такого підходу в контексті корпоративної розробки полягає у критичній оптимізації навантаження: серверна частина не зберігає стану або інформації про авторизованих користувачів у своїй оперативній пам'яті (RAM). Натомість при кожному вхідному HTTP-запиті фільтри Spring Security лише алгоритмічно верифікують криптографічний підпис токена (за допомогою секретного ключа), перевіряючи його валідність та зчитуючи дозволені права доступу до конкретного ендпоінту. Це архітектурне рішення не лише суттєво економить апаратні ресурси сервера, але й робить бекенд ідеально підготовленим до миттєвого горизонтального масштабування. Додатково, застосування надійного асиметричного криптографічного алгоритму RSA-2048 для формування цифрового підпису токенів гарантує високу цілісність даних, унеможливлюючи несанкціоновану підробку або модифікацію прав доступу користувача на боці клієнтського застосунку.

2.3 Технології зберігання, об'єктно-реляційного відображення та кешування даних

Основою будь-якої надійної облікової системи є сховище даних. Оскільки розроблювана платформа оперує критично важливою логістичною інформацією (рух товарів, зміна статусів замовлень, оновлення залишків), до підсистеми зберігання висуваються суворі вимоги щодо узгодженості. Для забезпечення принципів ACID (атомарність, консистентність, ізольованість, довговічність) при виконанні фінансових та складських операцій була обрана потужна реляційна система управління базами даних PostgreSQL. Її архітектура відрізняється високою надійністю та повноцінною підтримкою складних зв'язків між таблицями (наприклад, між магазинами, складами, категоріями товарів та історією транзакцій), що є необхідним для побудови єдиної корпоративної моделі даних.

Взаємодія між об'єктно-орієнтованим кодом мови Java та реляційною БД відбувається за допомогою спеціалізованого ORM-фреймворку (Object-Relational Mapping) Hibernate. Цей інструмент автоматизує рутинні процеси, перетворюючи класичні Java-об'єкти у відповідні реляційні таблиці та автоматично генеруючи оптимізовані SQL-запити. Використання Hibernate не лише пришвидшує розробку, але й забезпечує фундаментальний рівень безпеки: механізми фреймворку під капотом використовують параметризовані запити, що надійно усуває вразливості системи до атак типу SQL-ін'єкцій.

Зважаючи на специфіку роботи мережевого ритейлу, система регулярно стикається з великою кількістю однотипних запитів на читання (наприклад, масовий перегляд загального каталогу товарів продавцями з різних торгових точок). Для зниження навантаження на основну реляційну базу даних під час таких частих запитів було прийнято рішення інтегрувати систему кешування в оперативній пам'яті (In-Memory) Redis [18].

Архітектурна логіка інтеграції сервісу кешування в загальну систему обробки запитів наочно продемонстрована на рисунку 2.1.

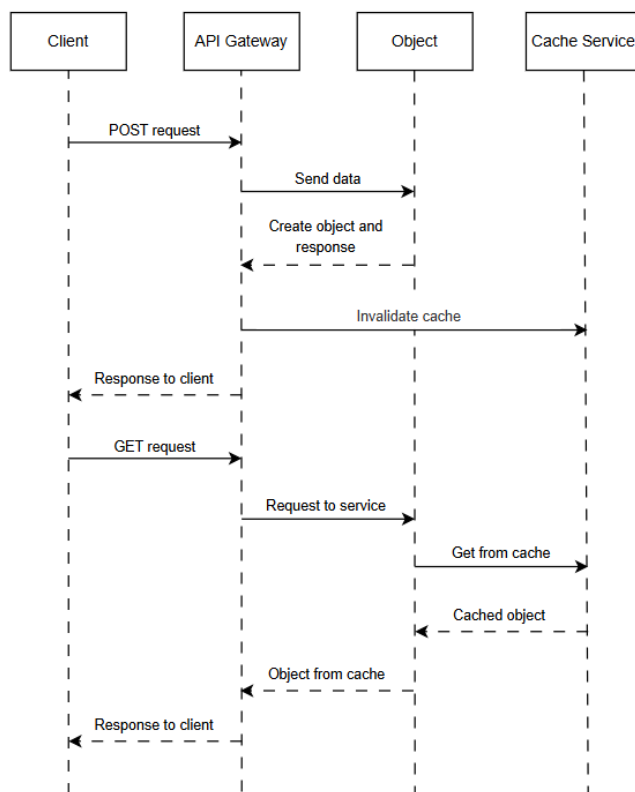


Рис. 2.1 UML-діаграма послідовностей процесу обробки та кешування запитів

Як видно з наведеної діаграми послідовностей, життєвий цикл обробки інформації чітко розділено залежно від типу HTTP-запиту:

1. Операції читання (GET request), коли клієнт запитує дані (наприклад, список товарів), API Gateway перенаправляє запит до сервісу кешування (Cache Service). Якщо дані вже збережені в оперативній пам'яті (Get from cache), вони миттєво повертаються клієнту у вигляді кешованого об'єкта (Cached object), повністю оминаючи звернення до дискової підсистеми PostgreSQL. Це зменшує час відповіді до кількох мілісекунд.

2. Операції запису (POST request) це коли клієнт ініціює зміну стану системи (наприклад, створює нове замовлення або оновлює ціну товару), система спочатку зберігає ці оновлені дані в основній базі. Відразу після успішного збереження генерується команда інвалідації кешу (Invalidate cache). Це критично важливий архітектурний крок, який примусово видаляє застарілі дані з пам'яті Redis. Завдяки

цьому гарантується, що при наступному запиті (GET) клієнт гарантовано отримає щойно оновлену та консистентну інформацію з основної бази.

2.4 Інструментарій для розробки, тестування, документування та контейнеризації

Безпосередній процес написання вихідного коду серверної частини здійснювався в сучасному інтегрованому середовищі розробки (IDE) IntelliJ IDEA. Вибір цього інструменту зумовлений його глибокою інтеграцією з екосистемою Java та Spring, а також наявністю розширених інтелектуальних інструментів для автодоповнення, рефакторингу та глибокого синтаксичного аналізу коду, що суттєво підвищує загальну продуктивність інженера.

Оскільки розробка бекенду (REST API) зазвичай випереджає створення клієнтського графічного інтерфейсу (фронтенду), критично важливим етапом є імітація та тестування мережових запитів. Для ручного тестування HTTP-запитів, налаштування авторизаційних заголовків (передачі JWT-токенів) та перевірки коректності обробки бізнес-логіки використовувалася платформа Postman. Цей інструмент дозволив створювати колекції тестових запитів та повноцінно імітувати життєвий цикл замовлень ще до етапу готовності візуальної частини системи.

Для забезпечення ефективної взаємодії між розробниками серверної та клієнтської частин у проєкт було впроваджено специфікацію Swagger (OpenAPI). Ця спеціалізована бібліотека автоматично сканує Java-код (аналізуючи анотації контролерів) та генерує актуальну інтерактивну вебдокументацію всіх доступних кінцевих точок (ендпоінтів). Це дозволяє розробникам клієнтських додатків миттєво тестувати запити та отримувати вичерпну інформацію про структуру очікуваних JSON-даних без необхідності вивчати вихідний код бекенду.

Фінальним та одним із найважливіших етапів архітектурного проєктування є забезпечення повної портативності та надійності розробленого рішення. Для цього було застосовано технологію контейнеризації Docker. Вона дозволяє запакувати

додаток разом із необхідним середовищем виконання (Java Runtime Environment), конфігураційними файлами та залежностями у повністю ізольований контейнер. Це інженерне рішення розв'язує класичну проблему прив'язки до постачальника послуг (vendor lock-in) та гарантує ідентичність роботи програми локально, на тестовому сервері та у продуктивному середовищі (Production).

Архітектурна схема ізольованого розгортання компонентів системи за допомогою Docker наведена на рисунку 2.2.

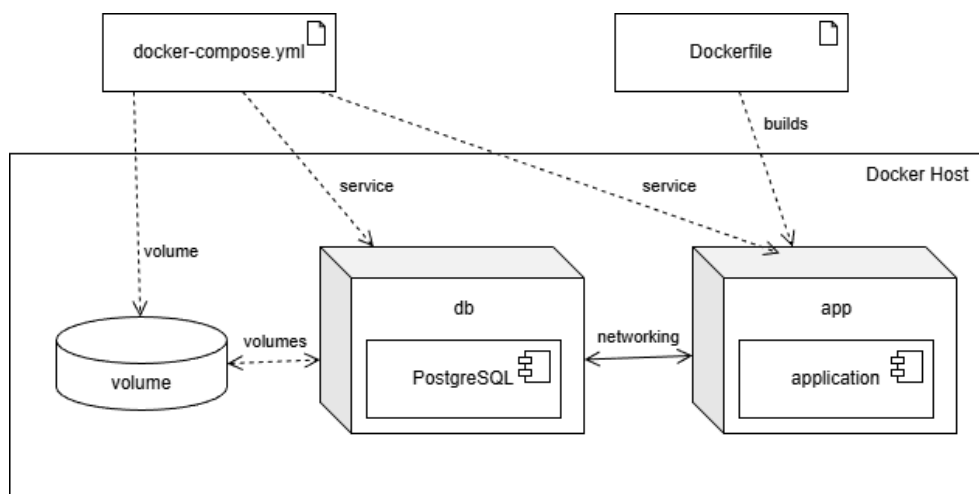


Рис. 2.2 Схема контейнеризованого розгортання системи за допомогою Docker

Як наочно продемонстровано на схемі, інфраструктура розгортання (Docker Host) управляється за допомогою оркестратора `docker-compose.yml`, який декларативно описує взаємодію двох ізольованих сервісів. Перший сервіс (`app`) – це безпосередньо розроблений Spring Boot додаток, образ якого автоматично збирається (`builds`) на основі інструкцій з файлу `Dockerfile` [19]. Другий сервіс (`db`) – це контейнер із системою управління базами даних PostgreSQL. Взаємодія між сервісами відбувається через ізольовану віртуальну мережу (`networking`), що гарантує безпеку даних. Для забезпечення довговічності (`persistence`) логістичної інформації у разі перезапуску чи оновлення бази даних, схема передбачає використання іменованого тому (`volume`), який зберігає файли СУБД незалежно від життєвого циклу самого контейнера [20].

3 ПРОЄКТУВАННЯ АВТОМАТИЗОВАНОЇ СИСТЕМИ ДЛЯ УПРАВЛІННЯ ТОВАРООБІГОМ "LVSTORE"

Проектування програмного забезпечення є критичною стадією життєвого циклу розробки, яка визначає внутрішню організацію системи, принципи її функціонування та механізми взаємодії між її компонентами. Розробка серверної частини інформаційної системи «LVStore» базується на формуванні надійної мікроархітектури, що передбачає строге узгодження обраних технологічних рішень із бізнес-вимогами щодо транзакційної цілісності, безпеки, масштабованості та підтримуваності корпоративного продукту.

У контексті автоматизації управління товарообігом, де фундаментально важливо забезпечити безперебійну обробку замовлень та жорстке розмежування прав доступу між торговими точками і складами, побудова внутрішньої структури платформи спирається на класичний патерн багаторівневої (layered) архітектури. Зокрема, основою для створення надійного бекенду виступає архітектурна модель Controller-Service-Repository. Компонентна організація застосунку за цим принципом гарантує чіткий поділ відповідальності між логічними модулями та підтримує абсолютну незалежність шарів. Обрана модель проектування забезпечує ізоляцію механізмів доступу до бази даних від бізнес-логіки та маршрутизації клієнтських запитів, в результаті спрощуючи інтеграційне тестування та дозволяючи масштабувати систему, якщо торгівельна мережа почне розширюватись.

3.1 Структура бази даних та архітектура сутностей

Інформаційна система «LVStore» використовує надійну реляційну модель бази даних, яка спроектована для забезпечення транзакційної цілісності та оптимізації складських і фінансових операцій. Архітектура бази даних складається з сімох основних таблиць (сутностей): User, Store, Warehouse, Product, Stock, WarehouseStock та Order. Кожна з цих таблиць виконує ізольовану бізнес-функцію,

а між ними встановлено строгі логічні зв'язки (зовнішні ключі), що дозволяють ефективно організувати управління товарними залишками та взаємодією між структурними підрозділами мережі.

Фізична структура реляційної моделі та зв'язки між її сутностями наочно відображені на ER-діаграмі (Entity-Relationship diagram), наведеній на рисунку 3.1.

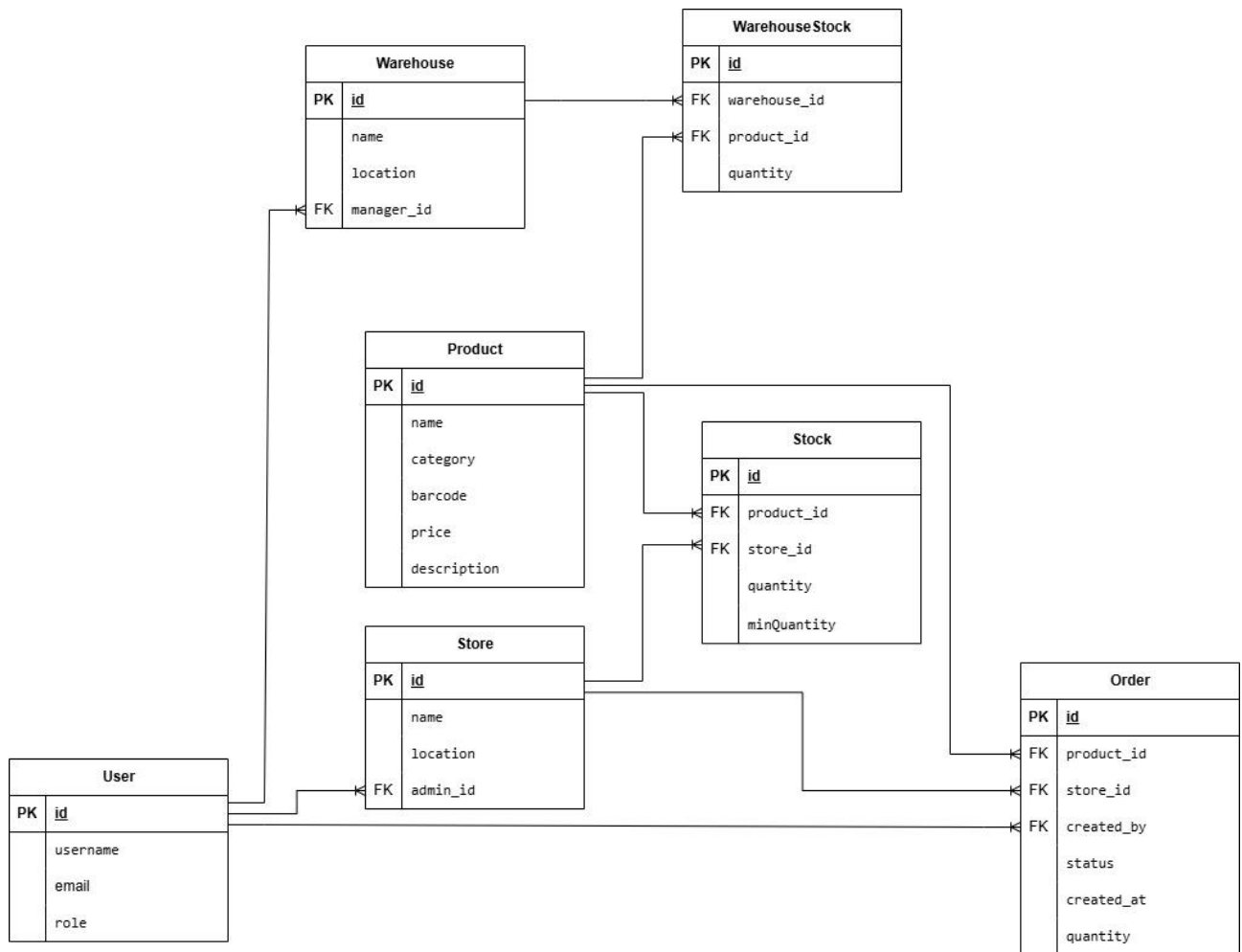


Рис. 3.1 ER-діаграма реляційної бази даних системи «LVStore»

Згідно з наведеною моделлю, архітектура даних базується на наступних ключових сутностях:

– сутність User (користувач), зберігає ідентифікаційні дані персоналу системи: логін, електронну пошту та системну роль (наприклад, адміністратор, менеджер складу, продавець). Первинним ключем є поле id. Ця сутність є ключовим учасником управління системою, оскільки користувач може бути пов'язаний із магазинами як

адміністратор (через поле `admin_id` у таблиці `Store`), зі складами як менеджер (через поле `manager_id` у таблиці `Warehouse`), а також виступати ініціатором замовлень (через поле `created_by` у таблиці `Order`);

- сутність `Store` (магазин) представляє фізичну торгову точку мережі та містить атрибути назви й місцезнаходження. Магазин організаційно закріплений за одним адміністратором (об'єктом `User`). Сутність має прямі зв'язки з таблицями `Stock` (локальні залишки товарів) та `Order` (історія замовлень);

- сутність `Warehouse` (склад) описує центральний або регіональний розподільчий центр, де зберігаються гуртові партії товарів. Кожен склад організаційно підпорядковується своєму менеджеру (об'єкту `User`) та безпосередньо пов'язаний із таблицею `WarehouseStock` для обліку наявності продукції;

- сутність `Product` (Товар) є центральним вузлом системи, що представляє товарну номенклатуру. Вона містить базові характеристики об'єкта: назву, категорію, унікальний штрихкод, ціну та текстовий опис. Сутність `Product` пов'язана із таблицями локальних залишків (`Stock`), складських запасів (`WarehouseStock`) та історії руху (`Order`);

- сутність `Stock` (Залишки в магазині) містить інформацію про фактичну кількість конкретного товару в конкретному магазині, поєднуючи сутності `Store` та `Product`. Критично важливою архітектурною особливістю цієї таблиці є наявність поля `minQuantity` (мінімально допустимий залишок), що використовується алгоритмами системи для превентивного формування сповіщень про загрозу товарного дефіциту;

- сутність `WarehouseStock` (Запаси на складі) є концептуальним аналогом таблиці `Stock`, але акумулює дані про кількість кожного товару на рівні складських приміщень. Має зовнішні ключі на таблиці `Product` та `Warehouse`;

- сутність `Order` (Замовлення) агрегує дані про внутрішні замовлення на переміщення товарів. Таблиця фіксує статус життєвого циклу заявки, кількість товару, дату створення, а також ідентифікатори цільового товару, магазину-отримувача та користувача, який ініціював процес.

Усі реляційні зв'язки між сутностями на рівні коду реалізовані через асоціації типу «багато до одного» (ManyToOne) адже він дозволяє підтримувати гнучку та нормалізовану структуру бази даних: наприклад, багато замовлень може бути створено одним користувачем, а один універсальний товар може одночасно перебувати на балансі кількох магазинів і складів.

3.2 Проєктування архітектурних шарів системи (MVC-патерн)

Для забезпечення високого рівня підтримуваності, масштабованості та зручності тестування, архітектура серверної частини «LVStore» побудована на принципах багаторівневої (layered) архітектури. Такий підхід базується на чіткому розділенні відповідальності між окремими компонентами системи, що дозволяє ізолювати бізнес-логіку від механізмів обробки мережових запитів та логіки доступу до бази даних. Програмний код платформи логічно поділено на три основні функціональні рівні: контролери, сервіси та репозиторії.

Рівень контролерів відповідає виключно за прийом вхідних HTTP-запитів від клієнтських додатків, первинну валідацію вхідних параметрів та налаштування маршрутизації (роутингу). Контролери не містять складної бізнес-логіки; їхня головна функція полягає у правильному делегуванні завдань сервісному рівню та формуванні коректних відповідей із відповідними HTTP-статусами (наприклад, 200 OK, 201 Created).

Сервісний рівень є центральним ядром системи, де зосереджена вся бізнес-логіка. Сервіси координують взаємодію між різними сутностями платформи, перевіряють внутрішні стани об'єктів (наприклад, валідують статус замовлення перед його підтвердженням) та викликають необхідні методи репозиторіїв для збереження або оновлення даних.

Рівень репозиторіїв забезпечує безпосередній доступ до реляційної бази даних PostgreSQL. Використання технології Spring Data JPA дозволяє реалізовувати стандартні CRUD-операції без написання прямих SQL-запитів, повністю інкапсулюючи логіку взаємодії з таблицями.

Головна інженерна перевага такої трирівневої архітектури полягає у суворому дотриманні принципів об'єктно-орієнтованого проєктування SOLID. Завдяки розмежуванню логіки, розробник має можливість модифікувати один шар системи без ризику порушити роботу інших. Наприклад, за необхідності зміни бази даних оновлення торкнуться лише рівня репозиторіїв, тоді як бізнес-логіка сервісів залишиться незмінною. Крім того, модульність архітектури критично спрощує процес тестування: сервісний рівень можна ізольовано перевіряти за допомогою імітаційних Mock-об'єктів (наприклад, підміняючи OrderRepository), не звертаючись до реальної бази даних. Це також створює ідеальне середовище для подальшого впровадження інструментів кешування або аналітики безпосередньо на рівні сервісів.

На рівні програмного коду об'єктно-орієнтована модель системи логічно згрупована за ключовими бізнес-сутностями: замовлення (Order), товари (Product), запаси (Stock), магазини (Store), користувачі (User) та склади (Warehouse). Взаємодія між компонентами базується на концепції делегування повноважень та впровадження залежностей. Контролери (наприклад, OrderController, ProductController) не взаємодіють з базою даних безпосередньо, а передають управління відповідним сервісам (OrderService, ProductService).

Своєю чергою, сервіси містять складну логіку перехресної взаємодії. Наприклад, OrderService не лише працює з OrderRepository для збереження транзакцій, але й активно взаємодіє з ProductService, StockService та UserService. Це дозволяє під час створення замовлення одночасно перевіряти фізичну наявність товару та верифікувати права доступу користувача. Репозиторії ж є кінцевими ланками цієї ієрархії, надаючи стандартизовані методи доступу до бази даних.

3.3 Проєктування системи безпеки та авторизації

Забезпечення надійного захисту корпоративних даних та суворе розмежування прав доступу (RBAC) є критичними вимогами до системи «LVStore». Для вирішення цих завдань було спроектовано спеціалізований модуль безпеки,

який реалізує безстейтову (stateless) архітектуру на основі відкритих стандартів OAuth2 та JWT (JSON Web Token) із використанням куки-токенів для безпечної передачі даних [21].

Архітектурна організація та взаємодія компонентів цього модуля детально відображена на діаграмі класів, що зображена на рисуюнок 3.2.

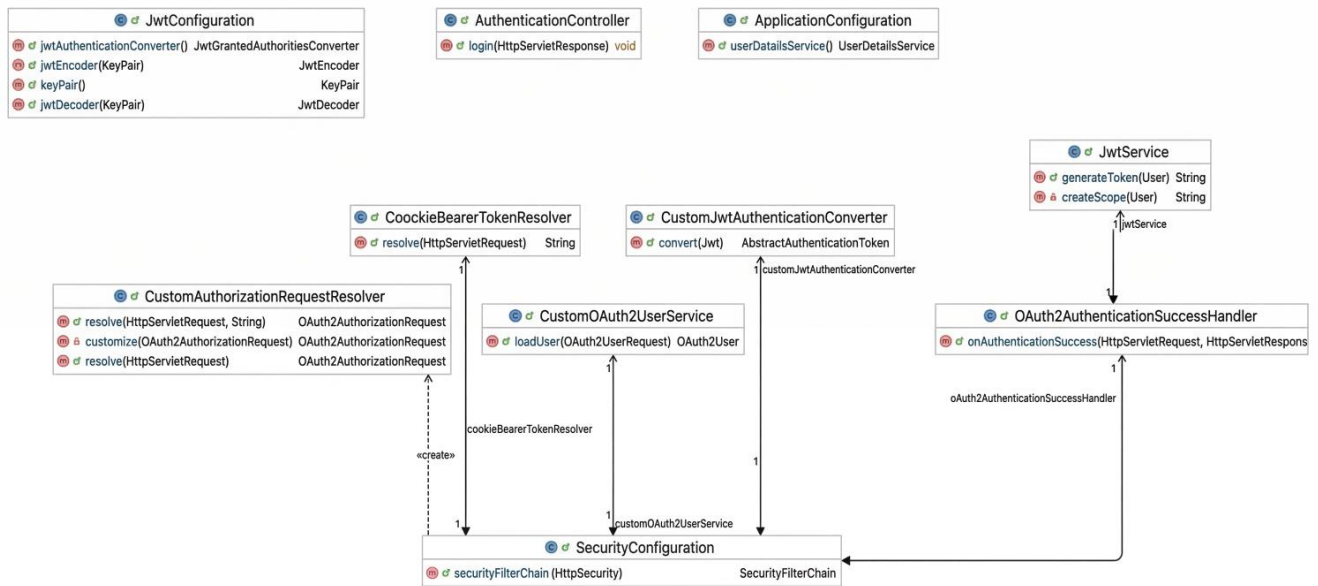


Рис. 3.2 Діаграма класів модуля безпеки системи аутентифікації та авторизації

Представлена діаграма охоплює конфігураційні класи, обробники автентифікації, сервіси та утиліти, які спільно реалізують комплексний механізм безпечного входу, генерації токенів та налаштування фільтрів безпеки:

- конфігурація безпеки (`SecurityConfiguration`) є основним класом безпекової конфігурації платформи. Він містить метод `securityFilterChain()`, який налаштовує ланцюг фільтрів Spring Security та інтегрує всі інші компоненти безпеки через механізм залежностей. Цей клас виступає центральним вузлом, маючи асоціації з обробниками токенів та сервісами користувачів;

- криптографія та генерація токенів (`JwtConfiguration`, `JwtService`) обробка токенів винесена в окремі модулі. `JwtConfiguration` надає методи для створення енкодера (`jwtEncoder()`) та декодера (`jwtDecoder()`) на базі згенерованої ключової

пари (`keyPair()`). Своєю чергою, `JwtService` реалізує безпосередню генерацію JWT-токена через метод `generateToken()`. Його приватний метод `createScope()` автоматично формує область доступу (`scope`) на основі організаційних ролей працівника;

– інтеграція протоколу OAuth2 для підтримки авторизації через сторонніх провайдерів використовується `CustomOAuth2UserService`, який виконує завантаження та мапінг профілів автентифікованих користувачів. Одночасно з цим `CustomAuthorizationRequestResolver` генерує кастомізовані запити для процесу OAuth2-авторизації;

– обробка успішного входу (`OAuth2AuthenticationSuccessHandler`) після підтвердження особи системою викликається метод `onAuthenticationSuccess()`. Цей клас має пряму залежність від `JwtService`, через яку він ініціює створення унікального токена та безпечно передає його клієнту (через збереження у файлах куки);

– верифікація доступу, `CookieBearerTokenResolver`, а також `CustomJwtAuthenticationConverter` при кожному наступному мережевому запиті `CookieBearerTokenResolver` автоматично витягує JWT-токен із куки клієнта. Далі `CustomJwtAuthenticationConverter` конвертує розшифрований JWT у стандартизований об'єкт безпеки Spring (`AbstractAuthenticationToken`), надаючи системі підтверджену інформацію про дозволені дії користувача;

– додатково інфраструктура підтримується класом `ApplicationConfiguration`, який через `userDetailsService()` завантажує деталі користувача з бази даних, та `AuthenticationController`, який відповідає за ініціацію процесу ручної авторизації.

3.4 Моделювання бізнес-процесів (сценарії використання)

Комплексне проектування програмного забезпечення вимагає дослідження не лише його статичної архітектури, а й динамічних аспектів функціонування під час виконання реальних бізнес-процесів. Для формалізації поведінкових характеристик інформаційної системи «LVStore» було застосовано об'єктно-орієнтований підхід до

моделювання. За допомогою сценаріїв використання (Use Cases) та UML-діаграм послідовностей (Sequence Diagrams) деталізовано алгоритми та хронологію взаємодії кінцевих користувачів із внутрішніми програмними компонентами платформи – контролерами, сервісами та репозиторіями. Відповідно до специфіки розробленої системи управління товарообігом, ключові операційні процеси логічно структуровані у два базові напрями: адміністрування товарної номенклатури та комплексне управління життєвим циклом внутрішніх замовлень [22].

3.4.1 Адміністрування каталогу

Процес управління каталогом є базовим для функціонування всієї системи, оскільки він формує номенклатурну базу для подальших операцій.

Сценарій 1: додавання нового товару, метою цього процесу є внесення нової товарної позиції до глобального каталогу. Ініціатором виступає адміністратор магазину, який через графічний інтерфейс передає системі такі параметри, як назва, категорія, штрихкод, ціна та кількість товару. Динаміку цього процесу відображено на рисунку 3.3.

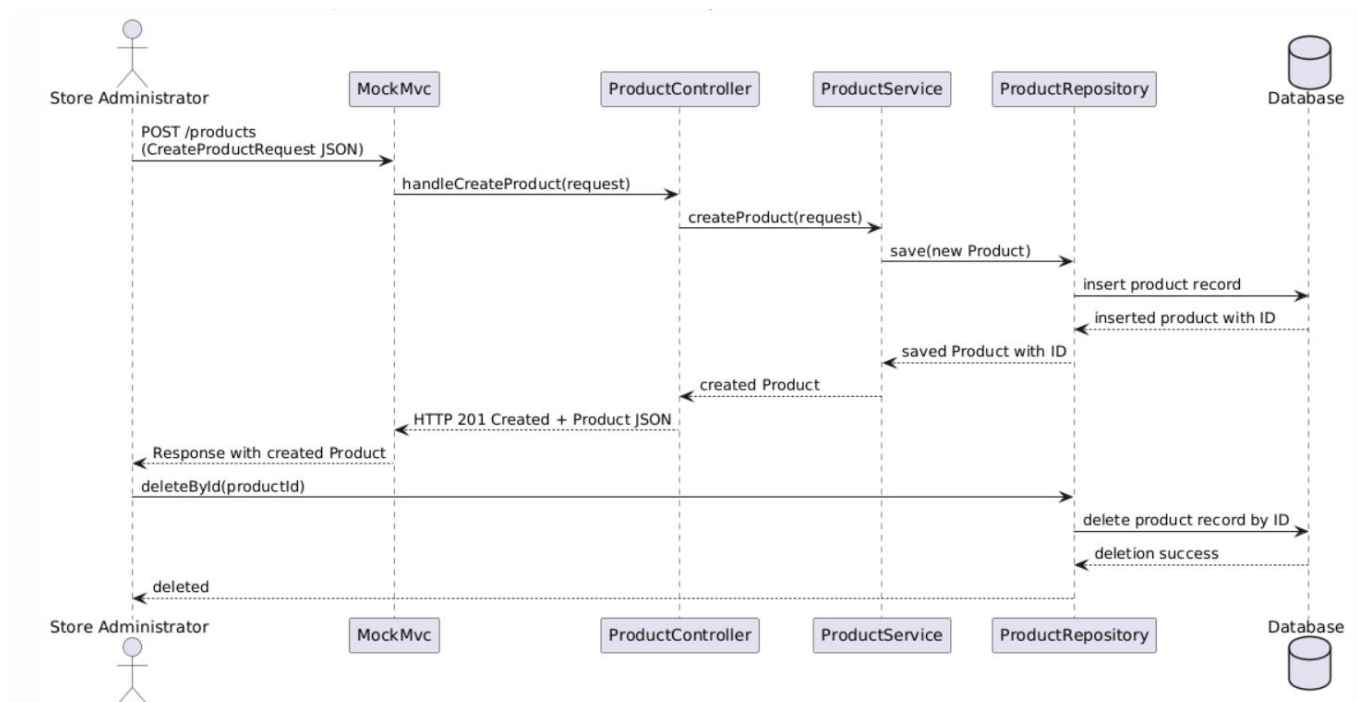


Рис. 3.3 UML-діаграма послідовності процесу додавання нового товару

Згідно з діаграмою, HTTP-запит (POST) обробляється ProductController, який делегує дані до ProductService для валідації. Після успішної перевірки викликається метод збереження в ProductRepository, який генерує SQL-запит (INSERT) до бази даних. У разі введення некоректних даних (наприклад, від'ємної ціни) система перериває транзакцію та повертає повідомлення про помилку.

Сценарій 2: оновлення інформації про товар. Цей сценарій дозволяє адміністратору редагувати існуючі характеристики товару (назву, ціну або категорію), щоб підтримувати актуальність каталогу, зображено на рисунку 3.4..

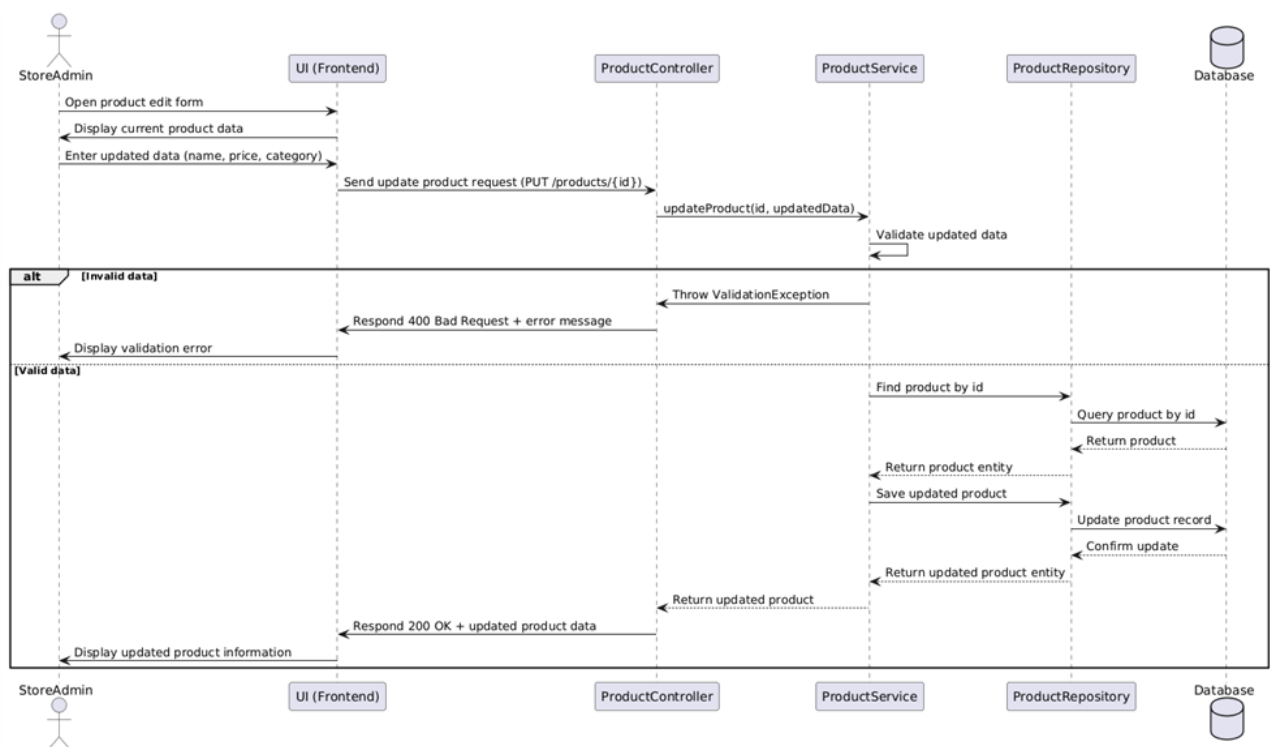


Рис. 3.4 UML-діаграма послідовності процесу оновлення інформації про товар

Запит на оновлення (PUT) супроводжується обов'язковою перевіркою валідності нових даних на рівні сервісу. Якщо дані некоректні, генерується виняток ValidationException із поверненням статусу 400 Bad Request. Якщо ж дані валідні, система знаходить товар за ідентифікатором, оновлює його поля та фіксує зміни у базі даних (UPDATE), після чого оновлена інформація стає доступною для всіх користувачів.

3.4.2 Життєвий цикл внутрішнього замовлення

Життєвий цикл замовлення є центральним механізмом системи «LVStore». Він передбачає послідовну зміну статусів заявки, координує роботу різних категорій персоналу та безпосередньо коригує стан складських запасів.

Сценарій 3: створення замовлення, ініціатором виступає продавець, мета якого – поповнити запаси магазину при досягненні мінімального порогу залишків. Процес зображено на рисунку 3.5.

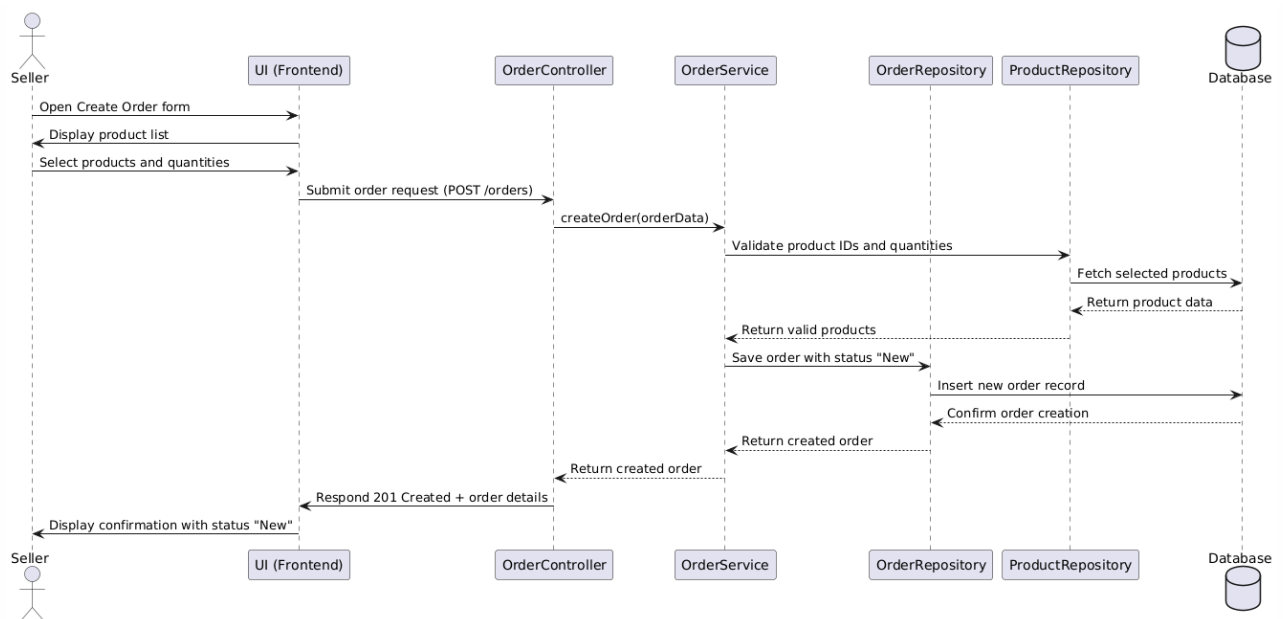


Рис. 3.5 UML-діаграма послідовності процесу створення замовлення

Продавець формує запит, вказуючи перелік товарів та їхню кількість. OrderService валідує ідентифікатори товарів через ProductRepository і, у разі успіху, створює новий запис у таблиці замовлень із початковим статусом «Нове». Відповідь сервера повертається зі статусом 201 Created.

Сценарій 4: підтвердження створеного замовлення потребує фінансової та логістичної валідації з боку керівництва.

Адміністратор магазину отримує список усіх нових замовлень, аналізує їх і натискає кнопку підтвердження. OrderService перевіряє поточний стан заявки та змінює її статус на «Підтверджене» (Confirmed), після чого замовлення стає видимим для працівників складу. Зображено на рисунку 3.6

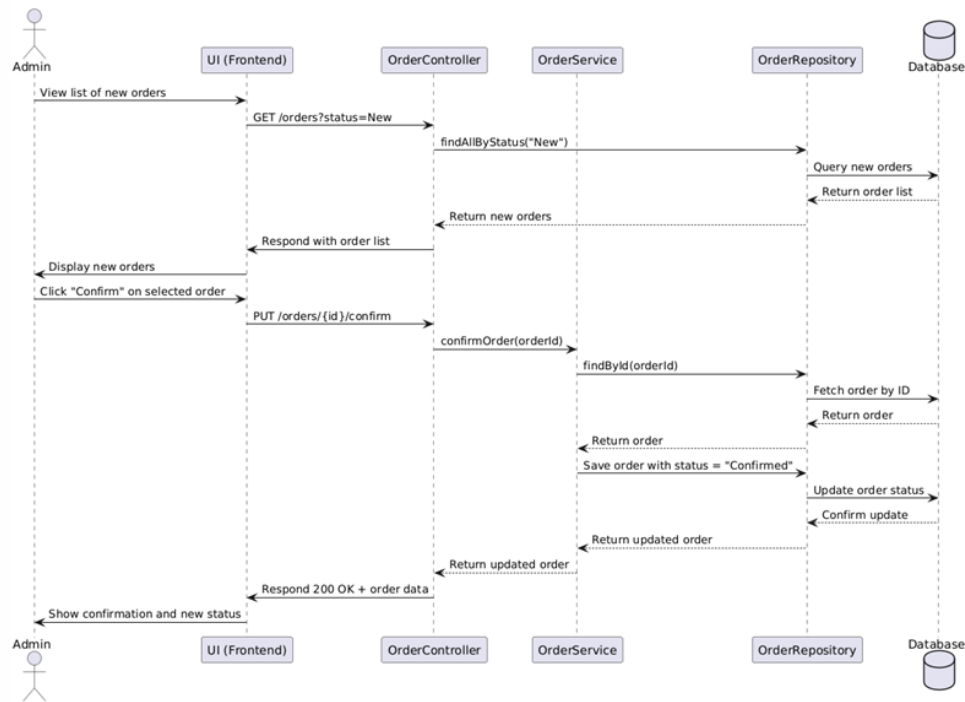


Рис. 3.6 UML-діаграма послідовності процесу підтвердження замовлення

Сценарій 5: виконання замовлення, відповідальність переходить до менеджера складу, який має фізично зібрати та відвантажити товар, процес зображено на рисунку 3.7.

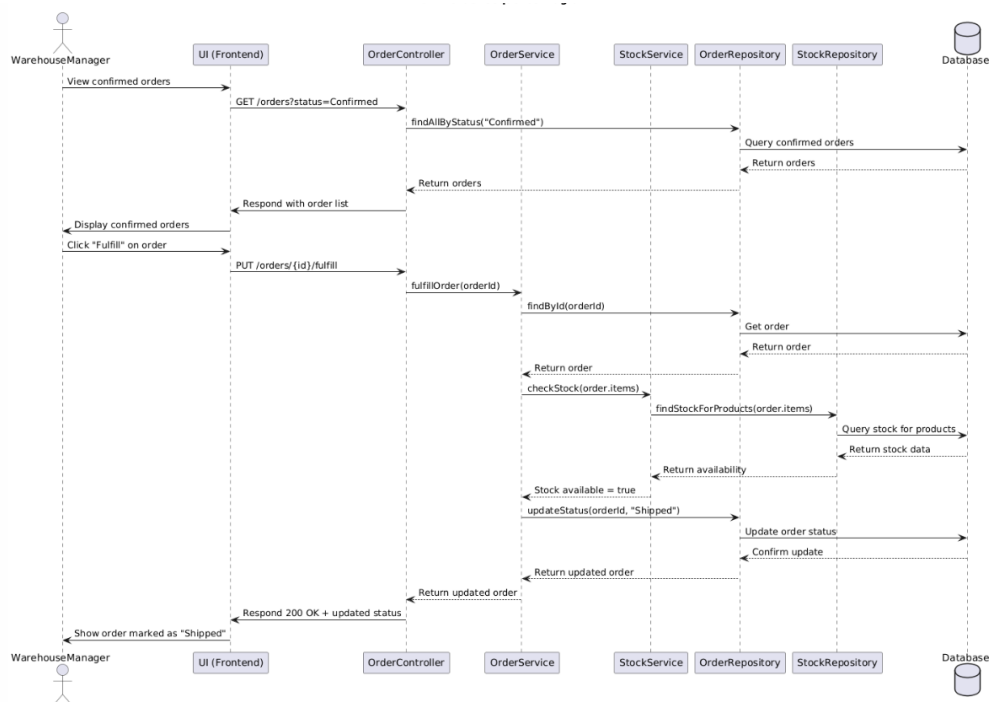


Рис. 3.7 UML-діаграма послідовності процесу виконання замовлення

При спробі змінити статус на «Відправлене» (Shipped), система (через StockService) проводить критично важливу перевірку: чи достатньо запитуваного товару на балансі складу. Лише за умови підтвердження наявності (Stock available = true) статус замовлення успішно оновлюється.

Сценарій 6: отримання товару в магазині. Фінальний етап життєвого циклу замовлення, на якому товар фактично надходить до цільової торгової точки, відображено на рисунку 3.8.

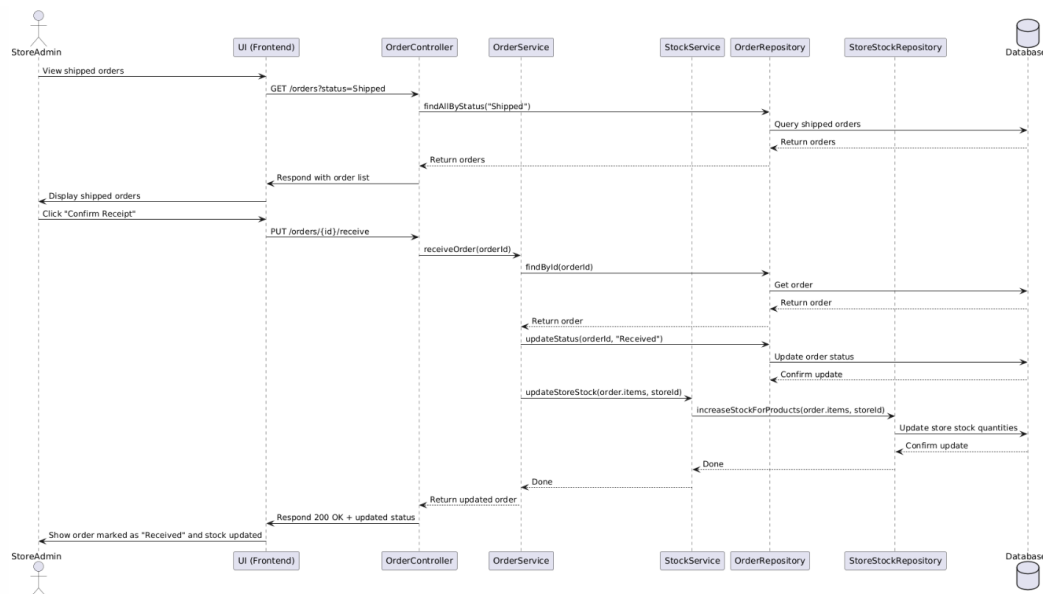


Рис. 3.8 UML-діаграма послідовності процесу отримання товару у магазині

Коли адміністратор магазину підтверджує отримання, система змінює статус заявки на «Отримане» (Received). Логіка цього етапу полягає у тому, що OrderService автоматично викликає метод increaseStockForProducts() у StockService, який миттєво нараховує отримані одиниці товару на баланс магазину в базі даних (таблиця Stock), гарантуючи цілісність обліку без ручного втручання.

3.5 Проєктування REST API та форматів обміну даними (DTO)

Взаємодія між розробленою серверною частиною та будь-якими зовнішніми клієнтами (вебінтерфейсом чи мобільним додатком) здійснюється через

спроектований програмний інтерфейс REST API. Архітектура інтерфейсу побудована на принципах ресурсно-орієнтованого підходу: кожен бізнес-об'єкт системи (товар, замовлення, магазин) виступає як окремий ресурс, доступ до якого здійснюється через уніфіковані ідентифікатори URI за допомогою стандартних HTTP-методів (GET, POST, PUT, DELETE).

Критично важливим архітектурним рішенням на етапі проєктування інтерфейсу стало впровадження патерну DTO (Data Transfer Object). З міркувань безпеки та оптимізації продуктивності, система ніколи не передає клієнту і не приймає від нього «сирі» об'єкти (сутності) [23], які безпосередньо відображаються на таблиці бази даних (наприклад, сутність Order або User). Замість цього на рівні контролерів використовуються спеціалізовані DTO-класи, які містять лише ті поля, що необхідні для конкретної операції.

Наприклад, при створенні нового замовлення клієнт відправляє об'єкт `CreateOrderRequest`, який містить лише ідентифікатори товарів та їхню кількість. Використання DTO забезпечує наступні переваги:

1. Інкапсуляція бази даних приховує від зовнішнього користувача внутрішню структуру реляційних таблиць та назви колонок.
2. Зменшення обсягу трафіку адже передаються лише релевантні дані без надлишкової інформації (наприклад, при запиті списку товарів не передається повна історія їх поставок).
3. Запобігання вразливостям: захищає систему від атак типу Mass Assignment, коли зловмисник намагається передати в JSON додаткові поля (наприклад, змінити свою роль чи баланс), які автоматично збереглися б у базі даних без використання DTO-прошарку.

Додатково проєктування API передбачає строгу стандартизацію відповідей сервера. Успішні операції читання повертають статус 200 OK, успішне створення ресурсу – 201 Created. У разі порушення бізнес-логіки (наприклад, спроба замовити товар, якого немає на складі) або помилки валідації вхідних DTO-об'єктів, система генерує виняток та повертає статус 400 Bad Request з описом помилки у форматі JSON. Для несанкціонованих запитів сервіс безпеки автоматично формує відповіді 401 Unauthorized або 403 Forbidden.

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

Практичне конструювання серверної частини інформаційної системи «LVStore» базувалося на попередньо затверджених архітектурних рішеннях та функціональних специфікаціях. Ключовим пріоритетом було впровадження передових інженерних практик, які гарантують високу гнучкість, надійний захист даних та здатність платформи до подальшого масштабування.

Завдяки застосуванню принципів чистого коду та багаторівневої структуризації на базі інструментарію Spring, вдалося досягти глибокої інкапсуляції бізнес-логіки. Такий підхід дозволив повністю відокремити алгоритми обробки замовлень від механізмів маршрутизації та роботи з базою даних [24]. Зведення зв'язності між програмними компонентами до мінімуму зробило кодову базу стійкою до модифікацій, що значно спрощує її подальший супровід без необхідності глобальної перебудови ядра системи.

Життєвий цикл розробки мав яскраво виражений ітеративний характер: імплементація кожного нового модуля супроводжувалася обов'язковою верифікацією його працездатності. Для підтвердження стабільності створеного REST API було застосовано комплексне тестування мережевих ендпоінтів, а для гарантії безперебійного функціонування у цільовому середовищі – впроваджено сучасні технології контейнеризованого розгортання.

4.1 Структура проєкту

У розробленій системі «LVStore» застосовано стандартизовану ієрархію директорій, характерну для фреймворку Spring Boot та системи автоматизації збирання Maven. Логічний поділ Java-класів на пакети (packages) суворо відповідає парадигмі трирівневої архітектури, яка була спроектована та обґрунтована у попередньому розділі.

На рисунку 4.1 наведено дерево каталогів розробленого серверного застосунку в інтегрованому середовищі розробки IntelliJ IDEA.

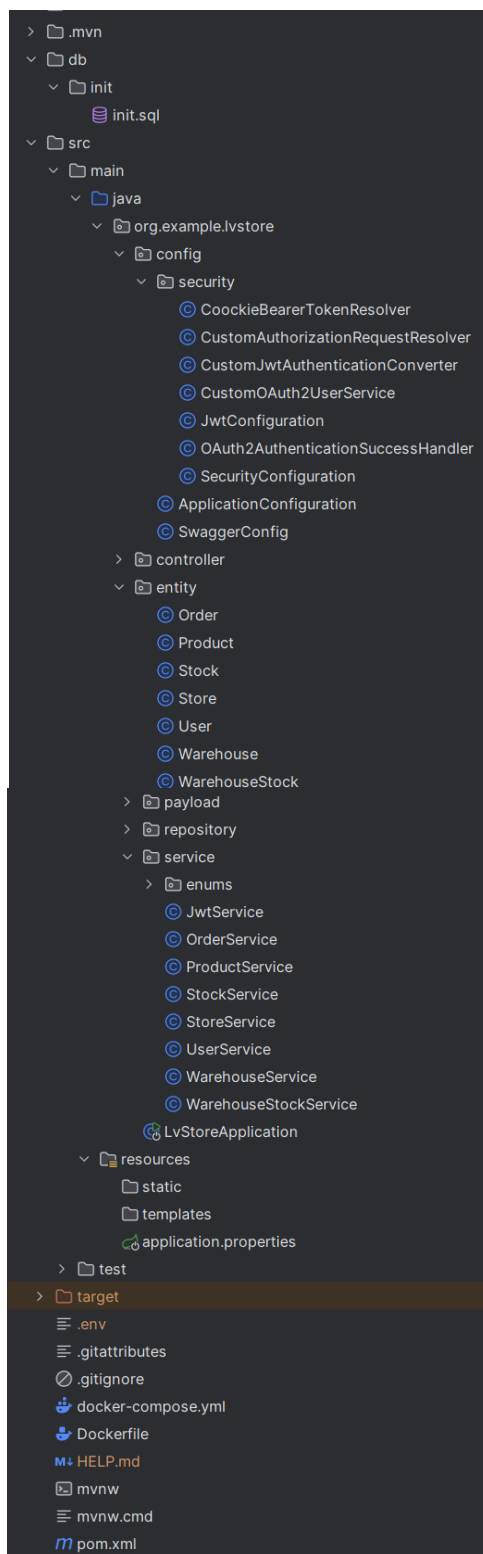


Рис. 4.1 Структура директорій та логічних пакетів проєкту в середовищі IntelliJ IDEA

Як видно з наведеної структури, кореневим пакетом застосунку є `org.example.lvstore`, у якому розміщено головний клас ініціалізації Spring контексту – `LvStoreApplication`. Весь інший програмний код розгорнуто та структуровано за наступними функціональними підпакетами:

- `config`, містить класи глобальної конфігурації системи. Зокрема, тут розміщено клас `ApplicationConfiguration` (базові налаштування бінів) та `SwaggerConfig`, який відповідає за автоматичну генерацію інтерактивної документації API (OpenAPI);

- `config.security`, глибоко спеціалізований підпакет, що інкапсулює всю логіку інформаційної безпеки. Тут розміщено центральний клас `SecurityConfiguration` (налаштування ланцюга фільтрів доступу), компоненти для роботи з JWT-токенами (`JwtConfiguration`, `CustomJwtAuthenticationConverter`, `CookieBearerTokenResolver`), а також класи для інтеграції авторизації за протоколом OAuth2 (`CustomOAuth2UserService`, `OAuth2AuthenticationSuccessHandler`);

- `controller`, шар представлення (роутингу). Включає REST-контролери, які відповідають за прийом HTTP-запитів від клієнтських додатків, первинну валідацію вхідних параметрів та повернення сформованих JSON-відповідей;

- `service`, центральне ядро бізнес-логіки. Містить класи (наприклад, `OrderService`, `ProductService`, `StockService`, `UserService`), які реалізують складні алгоритми обробки даних, розрахунки, перевірку наявності товарів та координацію взаємодії між різними сутностями. У цьому ж пакеті виділено підпакет `enums` для зберігання перелічень (наприклад, статусів замовлення);

- `repository`, шар доступу до даних (Data Access Layer). Містить інтерфейси, що розширюють можливості Spring Data JPA. Вони забезпечують безпосередню взаємодію з реляційною базою даних та містять як стандартні CRUD-методи, так і кастомні запити;

- `entity`, об'єктно-реляційна модель (ORM). Пакет включає Java-класи (`Order`, `Product`, `Stock`, `Store`, `User`, `Warehouse`, `WarehouseStock`), які за допомогою анотацій Hibernate відображаються на відповідні таблиці в базі даних PostgreSQL;

– payload, пакет, що виконує концептуальну роль DTO (Data Transfer Objects). Він агрегує класи, які описують формати вхідних запитів (Requests) та вихідних відповідей (Responses). Використання цього пакету дозволяє надійно ізолювати сутності бази даних (entity) від зовнішнього API, запобігаючи витоку конфіденційної інформації та атакам типу Mass Assignment.

Окрім безпосередньо Java-коду, архітектура проєкту включає низку важливих конфігураційних файлів та допоміжних директорій:

– директорія resources, яка містить файл application.properties, у якому задаються критичні налаштування середовища (параметри підключення до бази даних, порти сервера, параметри Hibernate);

– директорія db/init, що містить скрипт init.sql, який використовується для автоматичного створення схем та первинного наповнення бази даних при першому запуску;

– кореневі файли інфраструктури – pom.xml відповідає за управління залежностями бібліотек через Maven. Файли Dockerfile та docker-compose.yml містять інструкції для контейнеризації застосунку та розгортання ізольованої інфраструктури разом із СУБД. Файл .env використовується для безпечного зберігання локальних змінних середовища (наприклад, секретних ключів JWT).

4.2 Реалізація основного функціоналу та бізнес-логіки

Одним з етапів розробки стало написання програмного коду для ключових бізнес-модулів платформи. Усі компоненти розроблялися з орієнтацією на ізоляцію відповідальності та забезпечення надійної і безперебійної роботи системи навіть в умовах потенційно некоректних запитів від клієнта. Розробка охоплювала три основні вектори: налаштування механізмів безпеки, реалізацію бізнес-логіки життєвого циклу замовлень та створення REST API для керування товарним каталогом.

4.2.1 Реалізація механізму автентифікації та розв'язання токенів

Як було спроектовано в архітектурному розділі, система безпеки спирається на безстейтову модель з використанням JWT (JSON Web Token). Оскільки система працює як єдине ціле з потенційно різними типами клієнтів (браузер або мобільний застосунок), виникла потреба в гнучкому механізмі отримання токена доступу (Bearer Token).

Для вирішення цього завдання було реалізовано клас `CookieBearerTokenResolver`, який імплементує стандартний інтерфейс `Spring Security BearerTokenResolver`. Цей компонент діє як фільтр-перехоплювач, він аналізує кожен вхідний HTTP-запит і намагається витягти JWT-токен спочатку зі стандартного заголовка `Authorization`, а у разі його відсутності – сканує масив файлів `Cookie` на наявність токена з ім'ям `token`.

```
@Component
public class CookieBearerTokenResolver implements BearerTokenResolver {
    @Override
    public String resolve(HttpServletRequest request) {
        String authHeader = request.getHeader("Authorization");
        if(StringUtils.hasText(authHeader)                &&
authHeader.startsWith("Bearer ")) {
            return authHeader.substring(7);
        }
        if (request.getCookies() != null) {
            for (Cookie cookie : request.getCookies()) {

                if (cookie.getName().equals("token")) {
                    return cookie.getValue();
                }
            }
        }
        return null;
    }
}
```

Використаний гібридний підхід робить архітектуру безпеки універсальною, одночасно підтримує традиційні API-запити (з передачею токенів у заголовок) та безпечні браузерні сесії (через HTTP-only куки).

4.2.2 Реалізація бізнес-логіки управління замовленнями

Модуль OrderService не лише виконує CRUD-операції з базою даних, а й координує взаємодію із сервісами ProductService, StoreService та StockService для перевірки цілісності даних.

Критично важливою функцією цього сервісу є метод shipOrder(), який продемонстрований на наведеному коді нижче, він відповідає за фізичне відвантаження товару. Згідно з бізнес-правилами, перед відправкою замовлення система повинна жорстко перевірити дві умови: поточний статус заявки має бути CONFIRMED та на відповідному складі повинна бути достатня кількість товару.

```
public Order shipOrder(Long id) {
    Order order = getOrderById(id);
    if (order.getStatus() != OrderStatus.CONFIRMED) {
        throw new IllegalStateException("Only CONFIRMED orders can be shipped");
    }
    boolean inStock = stockService.isInStock(order.getProduct().getId(),
order.getStore().getId(), order.getQuantity());
    if (!inStock) {
        throw new IllegalStateException("Not enough stock to ship order");
    }
    stockService.decreaseStock(order.getProduct().getId(),
order.getStore().getId(), order.getQuantity());

    order.setStatus(OrderStatus.SHIPPED);
    return orderRepository.save(order);
}
```

На лістингу видно, що якщо перевірка методу `stockService.isInStock()` повертає `false`, транзакція негайно переривається шляхом генерації винятку `IllegalStateException`. В такому випадку у разі успіху сервіс ініціює метод `decreaseStock()` для списання товару із залишків, після чого фіксує новий статус замовлення. Такий код гарантує унеможливлення товарного дефіциту через помилки обробки.

4.2.3 Реалізація REST-контролерів

Шар представлення у системі реалізований за допомогою REST-контролерів, які делегують усю обчислювальну логіку відповідним сервісам. Прикладом такої архітектури є клас `ProductController`, що обслуговує ендпоінти для керування товарним каталогом.

```
@RestController
@RequiredArgsConstructor
@RequestMapping("/products")
@SecurityRequirement(name = "bearerAuth")
public class ProductController {
    private final ProductService productService;

    @PostMapping
    public ResponseEntity<Product> createProduct(@RequestBody
CreateProductRequest request) {
        try {
            return
ResponseEntity.status(HttpStatus.CREATED).body(productService.createProduct(re
quest));
        } catch (IllegalArgumentException e) {
            return ResponseEntity.status(HttpStatus.CONFLICT).build();
        }
    }
    // ... інші методи ...

    @PutMapping
    public ResponseEntity<Product> updateProduct(@RequestBody
UpdateProductRequest request) {
        try {
```

```

        return
        ResponseEntity.ok(productService.updateProduct(request));
    } catch (NoSuchElementException e) {
        return ResponseEntity.notFound().build();
    } catch (IllegalArgumentException e) {
        return ResponseEntity.status(HttpStatus.CONFLICT).build();
    }
}
}

```

Архітектурною особливістю цього контролера є використання об'єктів `CreateProductRequest` та `UpdateProductRequest` (патерн DTO) для захисту від надлишкових даних. Також варто відзначити правильну обробку HTTP-статусів через блоки `try-catch`, у разі спроби створити товар з існуючим штрихкодом генерується помилка, яку контролер коректно трансформує у статус 409 Conflict, а у випадку відсутності товару для оновлення — повертається статус 404 Not Found. Це повністю відповідає міжнародним стандартам розробки REST API.

4.3 Взаємодія з базою даних PostgreSQL

Для забезпечення надійного зберігання, швидкого доступу та суворої транзакційної цілісності інформації, у системі «LVStore» використовується потужна реляційна система управління базами даних PostgreSQL. Взаємодія між об'єктно-орієнтованим програмним кодом - Java та реляційними таблицями бази даних побудована на концепції ORM, де ключовим провайдером виступає фреймворк Hibernate.

З метою оптимізації процесу розробки та мінімізації низькорівневого коду, доступ до даних реалізовано через модуль Spring Data JPA. Цей інструмент дозволяє абстрагуватися від написання прямих SQL-запитів, натомість пропонуючи використання стандартизованих інтерфейсів-репозиторіїв. Такий підхід забезпечує автоматичну генерацію SQL-коду під час виконання програми.

Наочним прикладом архітектурної реалізації шару доступу до даних є компонент `OrderRepository`, який відповідає за маніпуляції з об'єктами замовлень.

```
package org.example.lvstore.repository;
import org.example.lvstore.entity.Order;
import org.example.lvstore.service.enums.OrderStatus;
import org.springframework.data.jpa.repository.JpaRepository;
import java.util.List;
public interface OrderRepository extends JpaRepository<Order, Long> {
    List<Order> findByStoreId(Long storeId);
    List<Order> findByStatus(OrderStatus status);
}
```

На наведеному лістингу, розроблений інтерфейс успадковується від базового контракту `JpaRepository<Order, Long>`. У генериках вказується тип цільової сутності (`Order`) та тип її первинного ключа (`Long`). Завдяки цьому успадкуванню репозиторій автоматично (з коробки) отримує повний набір стандартних CRUD-операцій: `save()`, `findById()`, `findAll()`, `deleteById()` та інші, без необхідності їх явної реалізації розробником.

Окрім базових операцій, Spring Data JPA підтримує потужний механізм створення похідних запитів (`Derived Query Methods`) – генерацію запитів на основі лексичного аналізу назви методу. У доданому лістингу задекларовано два кастомні методи:

1. `findByStoreId(Long storeId)` – під час виклику цього методу Hibernate автоматично формує та виконує SQL-запит на кшталт `SELECT * FROM orders WHERE store_id = ?`, повертаючи список усіх замовлень, що належать конкретному магазину.

2. `findByStatus(OrderStatus status)` – аналогічно перетворюється на запит фільтрації замовлень за їхнім життєвим циклом (наприклад, пошук усіх заявок зі статусом `NEW`), перетворюючи Java-перелічення (`Enum`) на відповідний строковий або числовий тип бази даних.

Використаний підхід не лише значно підвищує читабельність програмного коду, але й гарантує високий рівень інформаційної безпеки. Spring Data JPA під

капотом використовує параметризовані запити (Prepared Statements), що автоматично екранує вхідні дані та повністю унеможлиблює вразливості типу SQL-ін'єкцій. Крім того, абстрагування від специфічних SQL-команд гарантує, що система не прив'язана жорстко до PostgreSQL і, за потреби, зможе мігрувати на іншу реляційну СУБД лише шляхом зміни налаштувань у файлі `application.properties`.

4.4 Розгортання та конфігурація середовища (Docker)

Для забезпечення безперебійної роботи, швидкого розгортання та ізоляції програмного середовища від особливостей операційної системи хоста було використано технологію контейнеризації – Docker [25]. Використання контейнерів гарантує, що застосунок працюватиме ідентично як на локальній машині розробника, так і на віддаленому сервері, вирішуючи класичну проблему конфлікту залежностей.

Процес пакування серверної частини (бекенду) реалізовано за допомогою багатокрокової збірки (Multi-stage build), що описана у файлі `Dockerfile` та наведена нижче

```
FROM maven:3.9.5-eclipse-temurin-17 AS build
WORKDIR /app
COPY pom.xml .
COPY src ./src
RUN mvn clean package -DskipTests
FROM openjdk:17-jdk-slim
WORKDIR /app
COPY --from=build /app/target/*.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Цей підхід складається з двох етапів, спочатку на базі образу `maven:3.9.5-eclipse-temurin-17` відбувається компіляція вихідного коду та збирання виконуваного `.jar` файлу. На другому етапі використовується полегшений образ `openjdk:17-jdk-slim`, куди копіюється лише зібраний артефакт. Завдяки цьому вдається суттєво

зменшити кінцевий розмір Docker-образу та підвищити безпеку (за рахунок відсутності інструментів збірки у продакшн-контейнері).

Оскільки архітектура платформи вимагає одночасної роботи кількох взаємопов'язаних сервісів (самого застосунку, бази даних PostgreSQL та інструменту адміністрування), їх запуск оркеструється за допомогою утиліти Docker Compose. Відповідна конфігурація наведена у файлі `docker-compose.yml`:

```
version: '3.8'
services:
  app:
    build: .
    ports:
      - "8080:8080"
    depends_on:
      - db
    environment:
      - SPRING_DATASOURCE_URL=${POSTGRES_URL}
      - SPRING_DATASOURCE_USERNAME=${POSTGRES_USERNAME}
      - SPRING_DATASOURCE_PASSWORD=${POSTGRES_PASSWORD}
      - CLIENT_ID=${CLIENT_ID}
    networks:
      - app-network
  db:
    image: postgres:15
    environment:
      POSTGRES_DB: ${POSTGRES_DB}
      POSTGRES_USER: ${POSTGRES_USERNAME}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
    ports:
      - "5432:5432"
    volumes:
      - pgdata:/var/lib/postgresql/data
      - ./db/init:/docker-entrypoint-initdb.d
    networks:
      - app-network
```

Дана конфігурація об'єднує всі сервіси в єдину віртуальну мережу `app-network`, дозволяючи взаємодіяти між собою за внутрішніми іменами хостів (наприклад, застосунок звертається до бази даних за хостом `db`).

Використано важливі конфігурації:

1. Паролі та ключі доступу (наприклад, `CLIENT_SECRET`) не зберігаються у відкритому вигляді у файлі конфігурації, а передаються через змінні середовища (файл `.env`), що є стандартом безпеки.

2. Директива `depends_on` гарантує, що контейнер `app` почне запуск лише після старту контейнера `db`.

3. Том `./db/init:/docker-entrypoint-initdb.d` дозволяє автоматично виконати стартовий SQL-скрипт (`init.sql`) під час першого підняття контейнера бази даних, що одразу створює необхідну структуру таблиць.

4. Використання іменованого тому `pgdata` гарантує, що інформація у базі даних не зникне після зупинки чи видалення контейнера `db`.

4.5 Тестування системи

Надійність, стабільність та коректність функціонування системи «LVStore» забезпечується через комплексну стратегію багаторівневого тестування. Процес верифікації програмного забезпечення був органічно інтегрований у життєвий цикл розробки та охоплював модульні (Unit) перевірки бізнес-логіки, інтеграційне тестування взаємодії компонентів, а також ручне тестування API-ендпоінтів.

4.5.1 Модульне та інтеграційне тестування

Для перевірки коректності роботи окремих класів та методів в ізольованому середовищі застосовано модульне тестування з використанням фреймворків JUnit 5 та Mockito. Цей підхід дозволив протестувати бізнес-логіку сервісного шару (зокрема, `OrderService`, `ProductService`), імітуючи відповіді від бази даних за допомогою Mock-об'єктів. Це гарантує незалежність тестів від зовнішньої

інфраструктури дозволяючи ефективно перевіряти поведінку системи у граничних станах.

Нижче наведено приклад модульного тесту з класу `OrderServiceTest`, який перевіряє логіку відвантаження замовлення. Тест підтверджує, що за умов наявності товару на складі статус заявки успішно змінюється на `SHIPPED`, а кількість залишків коректно зменшується завдяки виклику методу `stockService.decreaseStock()`.

```
@Test
void
shipOrder_shouldUpdateStatusAndDecreaseStock_whenStockAvailable() {
    Order order = createOrder(OrderStatus.CONFIRMED);

    when(orderRepository.findById(1L)).thenReturn(Optional.of(order));
    when(stockService.isInStock(10L, 20L, 5)).thenReturn(true);
    when(orderRepository.save(any()))
    .thenReturn(inv.getArgument(0));
    Order result = orderService.shipOrder(1L);

    verify(stockService).decreaseStock(10L, 20L, 5);
    assertEquals(OrderStatus.SHIPPED, result.getStatus());
}
```

Для перевірки наскрізної роботи системи (від отримання HTTP-запиту до збереження запису в реляційну базу даних) були реалізовані інтеграційні тести. Вони ініціалізують повноцінний тестовий контекст Spring Boot та симулюють реальні клієнтські запити до REST-контролерів за допомогою інструменту `MockMvc`. Особливістю інтеграційного тестування у даному проєкті є перевірка рольової моделі доступу (RBAC).

Продемонстровано інтеграційний тест із класу `OrderControllerIntegrationTests`, який імітує запит від користувача з роллю `STORE_ADMINISTRATOR` на створення нового замовлення. Тест перевіряє, чи сервер коректно обробляє вхідний JSON-об'єкт, чи повертає статус `201 Created` та чи правильно формує зв'язки у базі даних між товаром, магазином та ініціатором.

```

@Test
@WithMockUser(authorities = "STORE_ADMINISTRATOR")
void createOrder_Success_AsStoreAdministrator() throws Exception {
    // Ініціалізація тестових даних: admin, store, product
    CreateOrderRequest request = new CreateOrderRequest(
        OrderStatus.NEW.name(),
        5,
        LocalDateTime.now(),
        productId,
        storeId,
        userId );

    String response = mockMvc.perform(post("/orders")
        .contentType(MediaType.APPLICATION_JSON)

        .content(objectMapper.writeValueAsString(request)))
        .andExpect(status().isCreated())
        .andExpect(jsonPath("$.status").value("NEW"))
        .andExpect(jsonPath("$.quantity").value(5))
        .andReturn().getResponse().getContentAsString();

    Order created = objectMapper.readValue(response, Order.class);

    assertThat(created.getProduct().getId()).isEqualTo(product.getId());

    assertThat(created.getStore().getId()).isEqualTo(store.getId());
    assertThat(created.getCreatedBy().getId()).isEqualTo(userId);
}

```

4.5.2 Ручне тестування та документація API

Окрім автоматизованих тестів, працездатність HTTP-маршрутів перевірялася вручну за допомогою платформи Postman. Одним із ключових етапів такої перевірки стало тестування процесу генерації нового внутрішнього замовлення. На рисунку 4.2 наведено результати виконання HTTP-запиту POST на маршрут /orders.

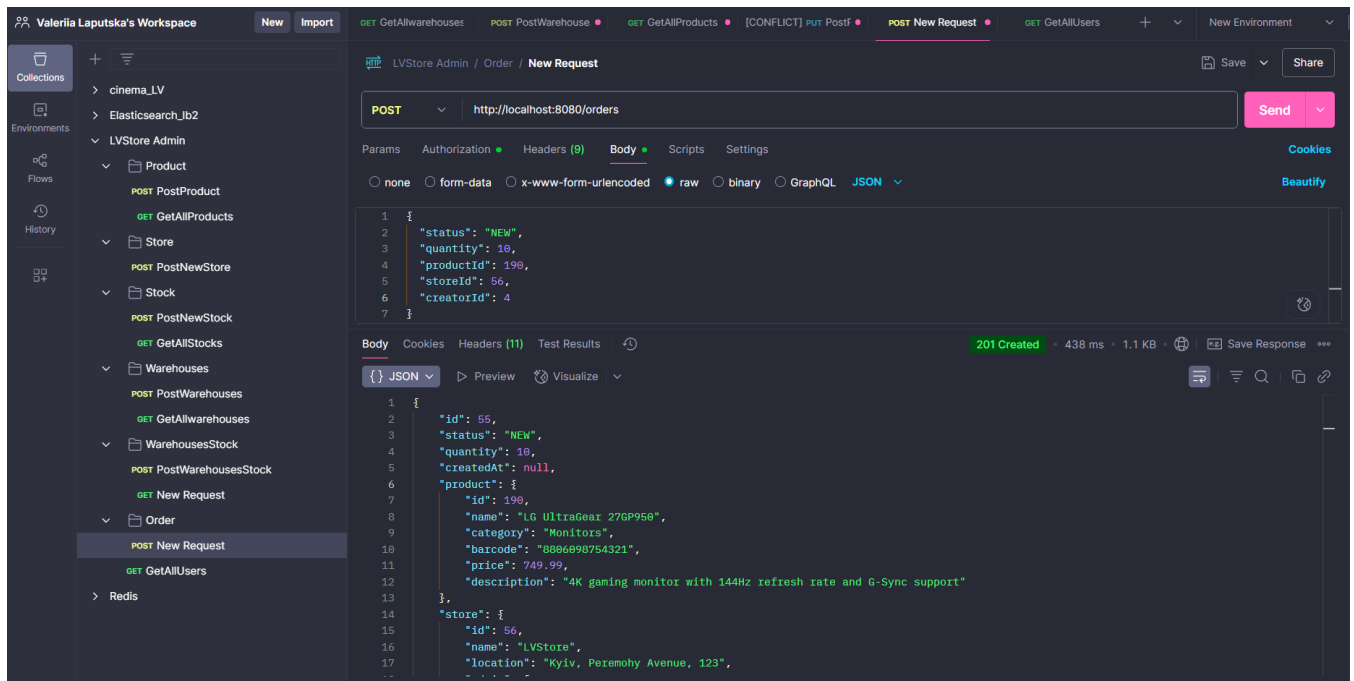


Рис. 4.2 Результати тестування ендпоінту створення замовлення у середовищі Postman

Як видно з результатів тестування, система успішно приймає вхідний JSON-об'єкт із параметрами замовлення. Бекенд коректно опрацьовує запит, валідує дані та повертає статус-код 201 Created, що свідчить про успішне створення транзакції. У тілі відповіді сервер повертає деталізований об'єкт створеного замовлення зі згенерованим первинним ключем.

Додатково, для забезпечення прозорості документації розробленого REST API та спрощення процесу майбутньої інтеграції з клієнтською частиною (фронтом), у систему було впроваджено відкритий стандарт OpenAPI за допомогою бібліотеки Swagger. Інтерактивний веб-інтерфейс, зображений на рисунку 4.3, дозволяє візуально оцінити структуру доступних ендпоінтів, переглянути схеми DTO-об'єктів та тестувати HTTP-запити безпосередньо з вікна браузера. Swagger UI повністю підтримує авторизацію через JWT-токени, доходяючи тестувати захищені маршрути. Завдяки такій автоматизації розробники клієнтських додатків отримують актуальну технічну специфікацію в режимі реального часу, що суттєво пришвидшує процес налагодження взаємодії між компонентами платформи.

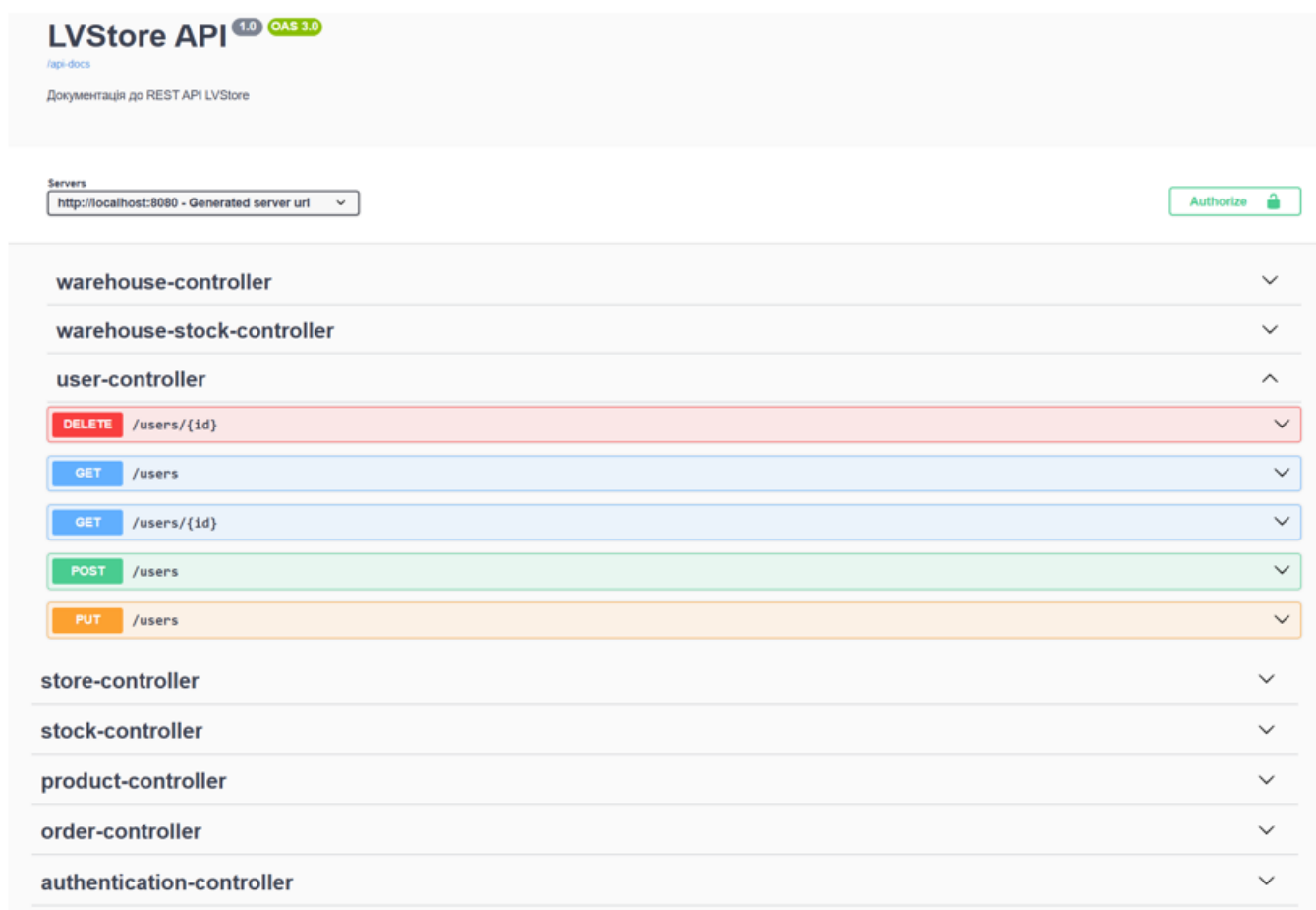


Рис. 4.3 Інтерактивна документація REST API за допомогою інтерфейсу Swagger UI

4.6 Демонстрація роботи системи

Клієнтська частина виступає у ролі кінцевого споживача програмних інтерфейсів, взаємодіючи з сервером за допомогою асинхронних HTTP-запитів, обмінюючись даними у форматі JSON та використовуючи механізми JWT-токенів для авторизації мережесесій.

На рисунку 4.4 зображено стартову форму авторизації користувача. Під час спроби входу клієнтський додаток формує об'єкт із введеними обліковими даними (Email та Password) і відправляє POST-запит на захищений маршрут бекенду. Окрім стандартної автентифікації за паролем, інтерфейс також передбачає можливість інтеграції протоколу OAuth2 для швидкого входу через обліковий запис Google. У

разі успішної перевірки на стороні сервера, бекенд генерує та повертає токен доступу, який дозволяє клієнту виконувати подальші захищені запити.

Рис. 4.4 Екранна форма авторизації користувача в системі «LVStore»

Після успішної авторизації візуальне відображення інтерфейсу та доступний функціонал динамічно адаптуються відповідно до системної ролі користувача (проектування яких було описано в розділі 3).

На рисунку 4.5 наведено інтерфейс панелі «Товари» з точки зору користувача з базовою роллю «Продавець». Ця сторінка формується на основі масивів даних, отриманих від серверного ProductController. Інтерфейс містить зручну таблицю з детальним описом товарної номенклатури (назва, категорія, ціна), а також функціонал пошуку за унікальним штрихкодом стандарту EAN-13. З цієї сторінки продавець має можливість ініціювати створення нової заявки за допомогою кнопки «Замовити».

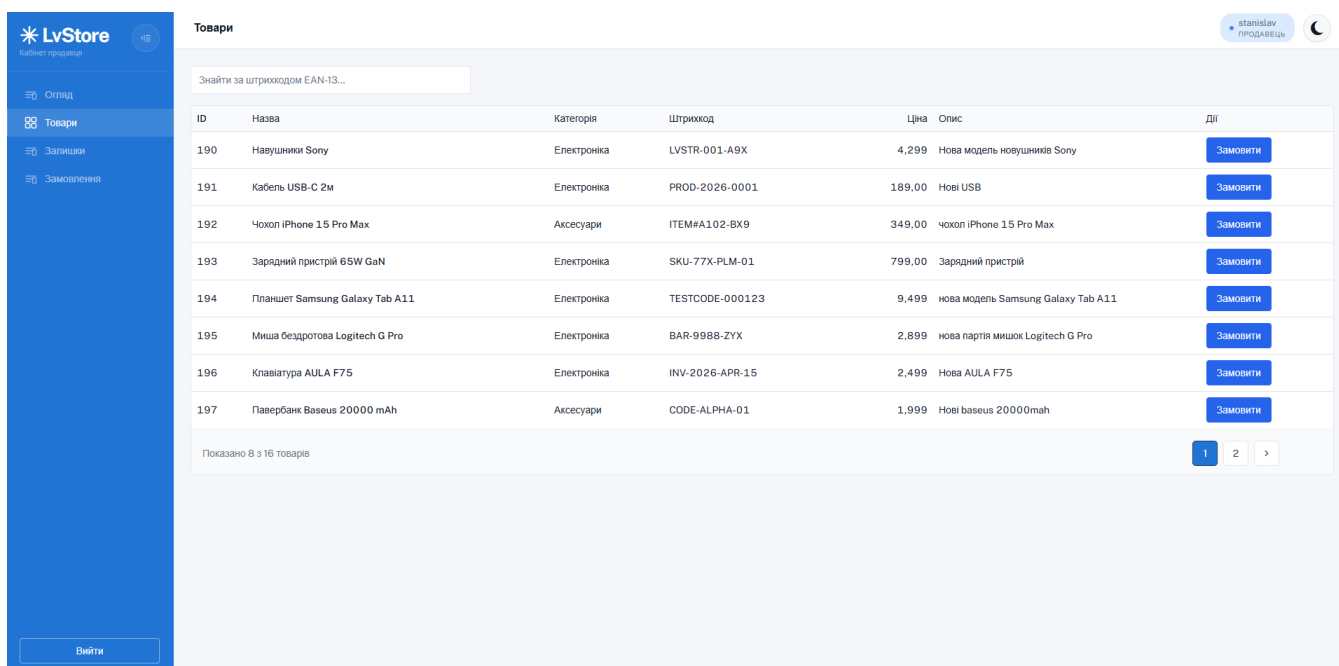


Рис. 4.5 Інтерфейс перегляду товарного каталогу (кабінет продавця)

Центральним бізнес-процесом системи є управління життєвим циклом внутрішніх замовлень. На рисунку 4.6 продемонстровано розширений інтерфейс панелі логістики, доступний для користувача з роллю «Адміністратор». Таблиця виводить вичерпну інформацію про всі сформовані заявки у системі. Візуальна кольорова індикація статусів (NEW, CONFIRMED, SHIPPED, RECEIVED, CANCELLED) дозволяє адміністратору швидко оцінювати стан логістики. За допомогою кнопок дій (Підтвердити, Скасувати, Змінити, Отримано) адміністратор здійснює управління життєвим циклом кожного замовлення. Кожне натискання кнопки ініціює відповідний PUT-запит до бекенду, який виконує транзакційну перевірку наявності товару та автоматично коригує записи у базі даних PostgreSQL.

Для забезпечення ефективної роботи логістичних вузлів мережі передбачено окремий функціонал для ролі «Менеджер складу». На рисунку 4.7 зображено інтерфейс управління складами. На цій сторінці менеджер може додавати нові складські приміщення до системи, призначати відповідальних осіб, змінювати локації та видаляти неактуальні записи. Важливо відзначити, що бокове навігаційне меню для цієї ролі містить специфічні розділи («Складські залишки», «Залишки

магазинів»), доступ до яких суворо контролюється на рівні серверних фільтрів безпеки.

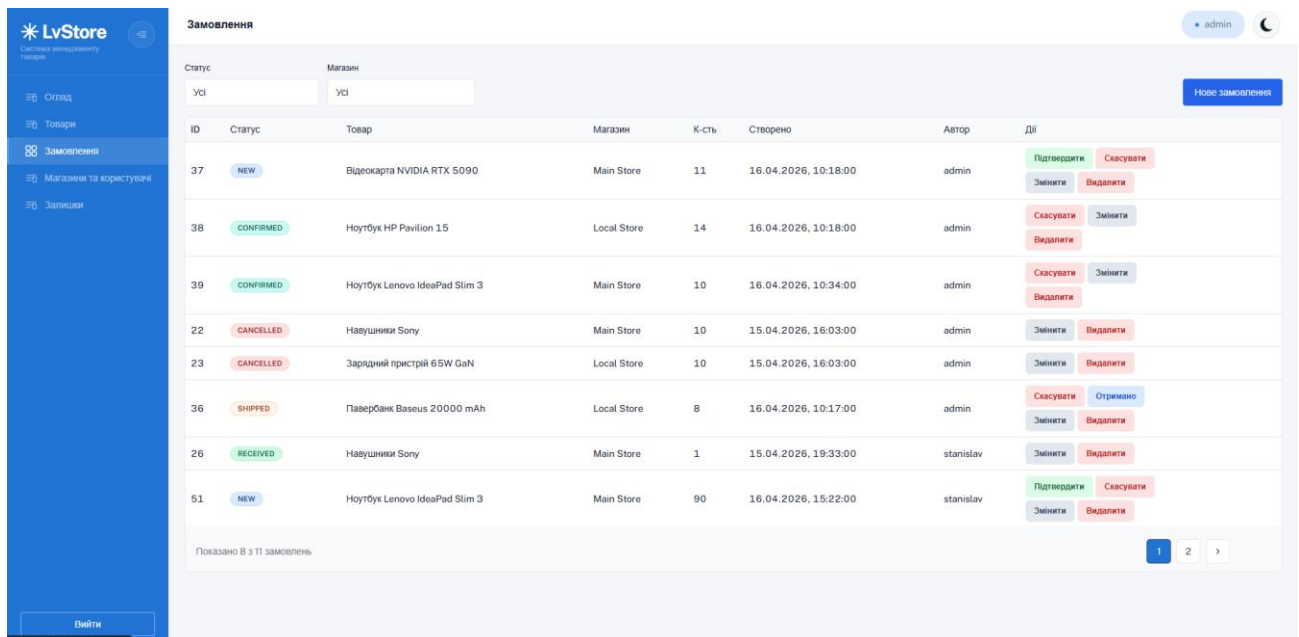


Рис. 4.6 Екранна форма управління статусами замовлень
(кабінет адміністратора)

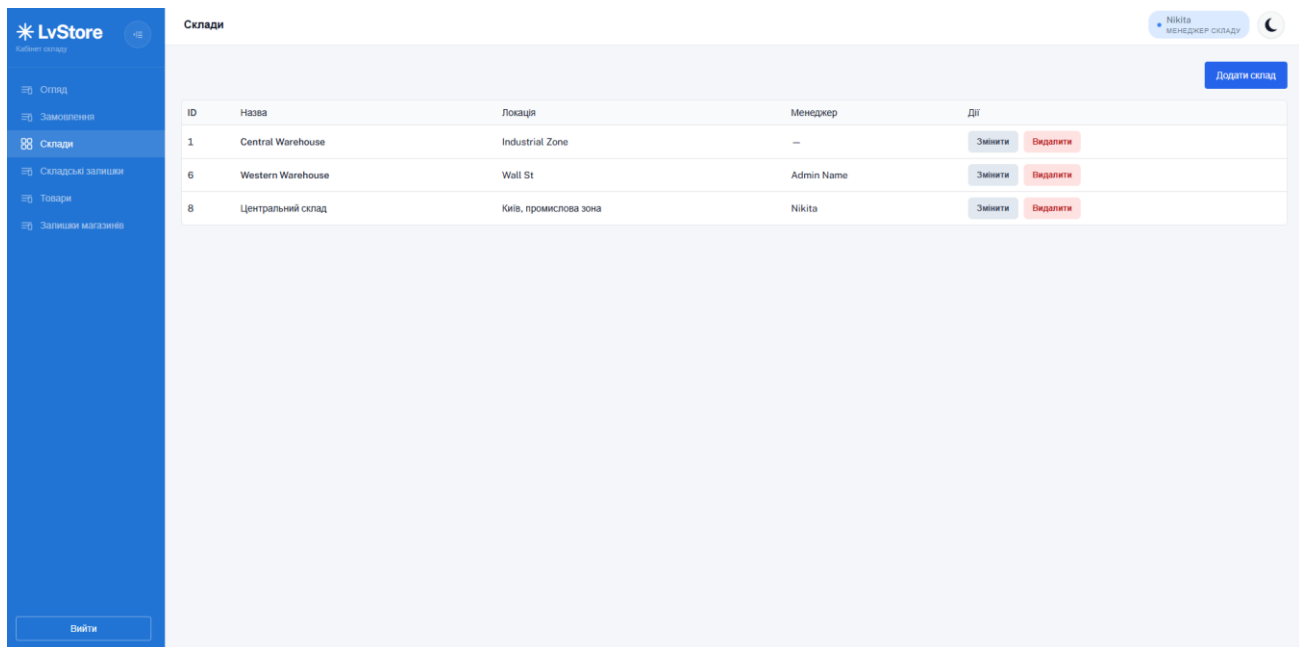


Рис 4.7 Інтерфейс управління логістичними вузлами
(кабінет менеджера складу)

ВИСНОВКИ

Розроблена та впроваджена серверна архітектура автоматизованої платформи «LVStore», призначеної для комплексного управління торговельною мережею. Створений програмний продукт дозволив перевести в цифровий формат ключові бізнес-процеси ритейлу: від обліку товарної номенклатури до контролю складських залишків та маршрутизації внутрішніх замовлень. Впровадження цієї системи мінімізує рутинне навантаження на персонал, виключає помилки людського фактора та створює єдиний, захищений інформаційний простір для взаємодії між складами і точками продажу.

У процесі досягнення поставленої мети було успішно вирішено такий комплекс завдань:

- здійснено глибоке дослідження предметної галузі сучасного ритейлу. Вивчено специфіку логістичної взаємодії між дистрибуційними центрами та роздрібними філіями. На основі виявлених «вузьких місць» в існуючих підходах до управління товарообігом, було сформульовано чіткі технічні та бізнес-вимоги до проектування власного програмного рішення;

- обґрунтовано та затверджено оптимальний технологічний стек. Для розробки серверного ядра використано мову Java та фреймворк Spring Boot. Сховищем даних виступила реляційна СУБД PostgreSQL. У якості допоміжних інструментів задіяно середовище IntelliJ IDEA, систему контролю версій Git/GitHub, а також Docker для контейнеризації. Механізми безпеки побудовано на базі Spring Security з використанням стандарту JWT та протоколу OAuth2;

- розроблено стійку архітектуру програмного забезпечення. Систему спроектовано за принципами слабкої зв'язності компонентів (трирівнева структура з використанням DTO-патернів). Крім того, побудовано нормалізовану реляційну ER-модель бази даних та впроваджено гнучку рольову матрицю (RBAC), яка суворо розмежовує інформаційні привілеї системних адміністраторів, продавців та менеджерів складу;

– програмно реалізовано повнофункціональний REST API. Розроблено захищені ендпоінти для керування каталогом продукції, користувачами, а також складною логікою життєвого циклу замовлень. Імплементовано алгоритми транзакційного списання та резервування товарних одиниць, що гарантує цілісність логістичних даних завдяки технологіям Hibernate та Spring Data JPA;

– забезпечено контроль якості коду через багаторівневе тестування. Алгоритмічну точність ключових сервісів підтверджено модульними тестами (JUnit 5, Mockito), а наскрізну обробку мережевих запитів та правил авторизації – інтеграційними перевірками (MockMvc). Додатково проведено мануальне тестування API за допомогою платформи Postman та згенеровано інтерактивну документацію за стандартом OpenAPI (Swagger UI);

– сформовано інфраструктуру для промислового розгортання проєкту. Застосунок та його базу даних запаковано у незалежні Docker-контейнери, оркестрація яких виконується через конфігурацію Docker Compose. Це забезпечує кросплатформенність рішення, його ізолюваність від хост-системи та готовність до неперервної інтеграції та доставки.

Впровадження розробленого рішення забезпечує підприємство прозорим інструментом для моніторингу товарних потоків у реальному часі, що дозволяє суттєво мінімізувати операційні ризики та підвищити точність управлінських рішень. Використання сучасних стандартів безпеки та технологій контейнеризації гарантує високу надійність і портативність системи, роблячи її готовою до негайного розгортання в умовах реальної бізнес-інфраструктури.

Окрім того, закладений API-first підхід створює надійний фундамент для подальшої інтеграції платформи в ширшу екосистему корпоративних сервісів, забезпечуючи її тривалу експлуатаційну цінність.

Робота пройшла апробацію. За її результатами було опубліковано наступні тези доповідей:

1. Лапуцька В.А., Замрій І.В. Кешування запитів у Spring Boot застосунку з використанням Redis. Матеріали VI Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24 квітня 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С. 203-205.

2. Лапуцька В.А., Замрій І.В. Впровадження контейнеризованого середовища для покращення масштабованості та відмовостійкості вебзастосунків. Матеріали II Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С. 77-79.

3. Лапуцька В.А., Замрій І.В. Механізми гарантування безпеки на основі розподілу ролей у Spring Boot застосунку на базі OAuth2 та JWT. Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 23 квітня 2026 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2026. С. 233-235.

ПЕРЕЛІК ПОСИЛАНЬ

1. Мазур Н., Ніколашин А. Ринок консалтингових послуг України та світу в умовах змін. Економіка та суспільство. 2021. № 24. URL: <https://doi.org/10.32782/2524-0072/2021-24-15>.

2. Пархаєва Н. В. Ризики комерційної діяльності та антикризове управління підприємством роздрібною торгівлі. Актуальні проблеми сталого розвитку. 2026. Т. 3, № 3. С. 191–199. URL: [https://doi.org/10.60022/3\(3\)-23s](https://doi.org/10.60022/3(3)-23s).

3. Integrating Spring Boot Java APIs with React and Angular Applications: A Comprehensive Guide. International Research Journal of Modernization in Engineering Technology and Science. 2025. URL: <https://doi.org/10.56726/irjmets68360>.

4. Digital twin paradigm: A systematic literature review / C. Semeraro et al. Computers in Industry. 2021. Vol. 130. P. 103469. URL: <https://doi.org/10.1016/j.compind.2021.103469>.

5. Temporal: Beyond State Machines for Reliable Distributed Applications. Durable Execution Solutions | Temporal. URL: <https://temporal.io/blog/temporal-replaces-state-machines-for-distributed-applications>.

6. Li K., Min B.-W. Improvement of Permission Management System Based on RBAC(Role-Based Access Control) Model. International JOURNAL OF CONTENTS. 2023. Vol. 19, no. 4. P. 86–97. URL: <https://doi.org/10.5392/ijoc.2023.19.4.086>

7. PEak: A Single Source of Truth for Hardware Design and Verification / C. Donovan et al. ACM Transactions on Embedded Computing Systems. 2024. URL: <https://doi.org/10.1145/3703456>.

8. Architectures in relational databases: An analytical study of SQL-based data models and ACID principles | Deep Science Publishing. URL: https://doi.org/10.70593/978-93-7185-191-6_1.

9. Програма для обліку товару Торгсофт [Електронний ресурс] // Офіційний сайт компанії «Торгсофт». – Режим доступу до ресурсу: <https://torgsoft.ua>.

10. Sousa, S., et al. The Impact of Interface Complexity on User Performance and Error Rates in Enterprise Systems // IEEE Access. 2022. Vol. 10. P. 45120-45132. DOI: <https://doi.org/10.1109/ACCESS.2022.3168123>.
11. Migration Strategies from Monolithic Applications to Cloud-Native Applications: A Systematic Review / J. Vacacela Betancourt et al. Revista Politécnica. 2026. Vol. 57, no. 1. P. 123–138. URL: <https://doi.org/10.33333/rp.vol57n1.10>.
12. WooCommerce - Open Source ecommerce Platform. WooCommerce. URL: <https://woocommerce.com/>.
13. Al-Tawil M., et al. Security Analysis and Vulnerability Assessment of Content Management Systems and Third-Party Plugins // International Journal of Advanced Computer Science and Applications. 2022. Vol. 13, No. 5. P. 542-550. DOI: <https://doi.org/10.14569/IJACSA.2022.0130564>.
14. Loyverse - Easy-to-Use Point of Sale. Loyverse. URL: <https://loyverse.com>.
15. Towards Digital Transformation in SMEs: A Custom Software Solution for Shopfloor–ERP Integration // Machines. 2024. Vol. 13, Iss. 11. P. 1002. DOI: <https://doi.org/10.3390/machines13111002>.
16. Rahman, M. A., et al. A Comprehensive Survey on Cloud-Native Architecture and Microservices // IEEE Access. 2022. Vol. 10. P. 78241-78265. DOI: <https://doi.org/10.1109/ACCESS.2022.3193498>.
17. Hiwale P. R. Review On Cross-Platform Mobile Application Development. International Journal for Research in Applied Science and Engineering Technology. 2022. Vol. 10, no. 1. P. 1433–1439. URL: <https://doi.org/10.22214/ijraset.2022.40004>.
18. Zhijie Li, Design of Ordering System based on Spring Boot Framework, International Core Journal of Engineering, 2022. Vol. 8, no. 5. URL: [https://doi.org/10.6919/ICJE.202205_8\(5\).0074](https://doi.org/10.6919/ICJE.202205_8(5).0074)
19. Hernandez L., Rios C. E. U. Docker Optimization of an Automotive Sector Virtual Server Infrastructure. Knowledge Engineering and Data Science. 2024. Vol. 7, no. 1. P. 71. URL: <https://doi.org/10.17977/um018v7i12024p71-85>.
20. Optimizing PostgreSQL for High-Volume Insurance Transactions & Secure Backup and Restore Strategies for Databases. International Journal of Emerging Trends in

Computer Science and Information Technology. 2023. Vol. 4. URL: <https://doi.org/10.63282/3050-9246.ijetcsit-v4i1p112>.

21. Zahir R. M., Esenalieva G. Authentication Approaches in Spring Security. Preprints.org. 2025. URL: https://www.preprints.org/frontend/manuscript/f696d4daa6be2f1af1f3a6b4078238b1/download_pub. DOI: <https://doi.org/10.20944/preprints202501.0899.v1>.

22. Siewe F., Ngounou G. M. On the Execution and Runtime Verification of UML Activity Diagrams. Software. 2025. Vol. 4, no. 1. P. 4. URL: <https://doi.org/10.3390/software4010004>.

23. Pardede C., Sihombing W., Nainggolan W. Comparative Study of Manual and Generated Data Transfer Object Implementation Performance. Journal of Applied Informatics and Computing. 2025. Vol. 9, no. 5. P. 2912–2919. URL: <https://doi.org/10.30871/jaic.v9i5.10818>.

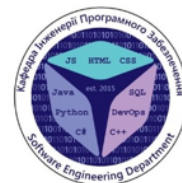
24. Lano K., Yassipour Tehrani S. Introduction to Software Architecture. Cham : Springer Nature Switzerland, 2023. URL: <https://doi.org/10.1007/978-3-031-44143-1>.

25. Роль контейнеризації та віртуалізації на рівні операційної системи в розвитку хмарно орієнтованих додатків / а. Подвиженко та ін. *Information technology and society*. 2025. № 1 (16). С. 183–188. Url: <https://doi.org/10.32689/maup.it.2025.1.23>.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ



ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ



Система менеджменту товарів для оптимізації мережі магазинів.
Спец-частина: розробка Backend-частини системи

Виконала студентка 4 курсу
групи ПД-41
Лапуцька Валерія Анатоліївна
Керівник роботи
д-р.техн.наук., проф., зав. кафедри ІПЗ Замрій Ірина Вікторівна

Київ – 2026

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи – удосконалення процесів контролю товарних запасів та обробки замовлень для вирішення проблеми неузгодженості дій між торговими точками та складами, а також запобігання товарному дефіциту.

Об'єкт дослідження – процес управління товарообігом, формування замовлень та контролю товарних залишків у мережах роздрібної торгівлі.

Предмет дослідження – методи, засоби та технології для побудови автоматизованої інформаційної системи управління мережею магазинів.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Проаналізувати предметну область управління внутрішньою логістикою мережевого ритейлу та дослідити існуючі комерційні аналоги, щоб виявити їхні функціональні недоліки та сформулювати обґрунтовані технічні вимоги до розробки.
2. Обґрунтувати вибір архітектурного підходу (REST), мови програмування та технологічного стеку з метою забезпечення високої продуктивності та надійності серверної частини (Backend).
3. Спроекувати реляційну модель бази даних, модульну архітектуру програмних шарів та багаторівневу модель розмежування прав доступу (RBAC) для гарантування транзакційної цілісності даних, ізоляції бізнес-логіки та захисту системи від несанкціонованого доступу.
4. Розробити алгоритми управління життєвим циклом замовлень на основі патерну State Machine та забезпечити їх програмну реалізацію.
5. Здійснити контейнеризацію застосунку для забезпечення портативності та провести комплексне тестування розробленого REST API.

3

АНАЛІЗ АНАЛОГІВ

Показник	Торгсофт	WooCommerce	Loyverse POS	LVStore
Спеціалізація на внутрішній логістиці	Частково	Частково	Частково	Так
Автоматизація внутрішніх замовлень	Частково	Ні	Частково	Так
Жорстка багаторівнева рольова модель	Так	Ні	Ні	Так
Захист даних від випадкового видалення	Так	Частково	Так	Так
Портативність та ізолюваність розгортання	Ні	Частково	Ні	Так

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

1. Автентифікація та авторизація користувачів (вхід у систему та надання доступу до функцій відповідно до ролі користувача).
2. Розмежування прав доступу за стандартом RBAC (Role-Based Access Control: продавець, адміністратор магазину, менеджер складу).
3. Моніторинг товарної номенклатури та залишків (забезпеченням оперативного доступу до актуальної інформації про кількість та стан товарів для всіх авторизованих ролей).
4. Обробка та менеджмент замовлень (створення внутрішніх замовлень на поповнення запасів, перевірка реальної наявності на складі та зміна статусів виконання).
5. Контроль товарних запасів (моніторинг залишків у магазинах та на складах, відстеження критичного рівня запасів).

Нефункціональні вимоги:

1. Продуктивність (час обробки стандартних REST API запитів до бази даних не повинен перевищувати 200 мілісекунд. База даних гарантує виконання принципів ACID).
2. Безпека (сесії користувачів є Stateless. Аутентифікація захищена JWT-токеном із використанням асиметричного криптографічного алгоритму RSA-2048. Захист від SQL-ін'єкцій забезпечується на рівні ORM (Hibernate)).
3. Сумісність та інтеграція (взаємодія з клієнтом відбувається виключно через REST API у стандартизованому форматі JSON. Серверна частина ізольована у Docker-контейнері для масштабування).
4. Реалізація In-Мемогу кешування для зниження навантаження на базу даних.

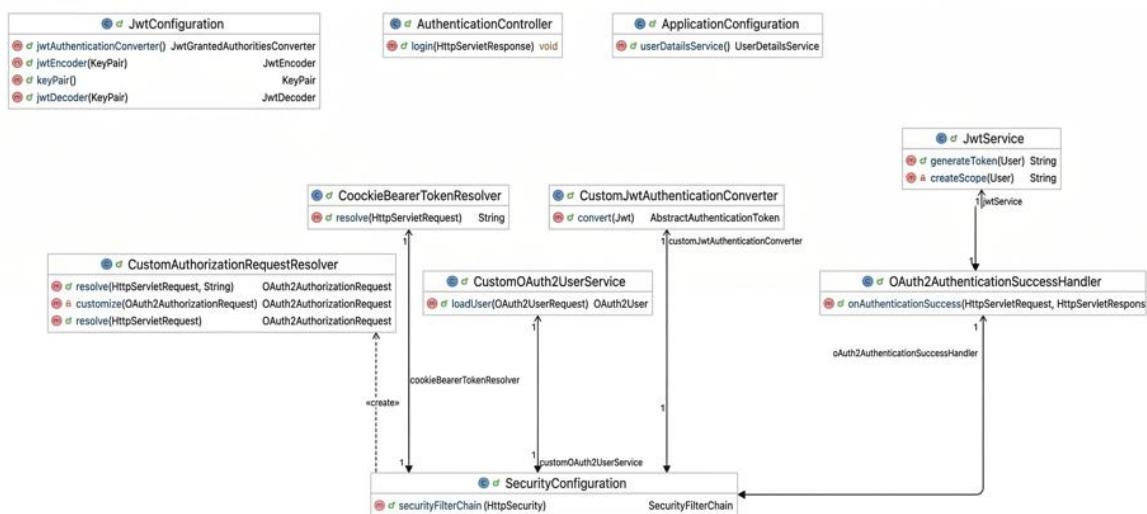
5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ



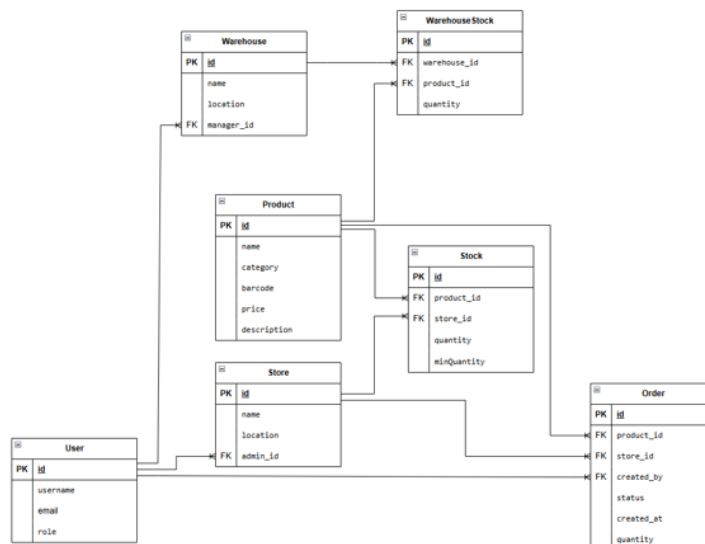
6

ДІАГРАМА КЛАСІВ МОДУЛЯ БЕЗПЕКИ



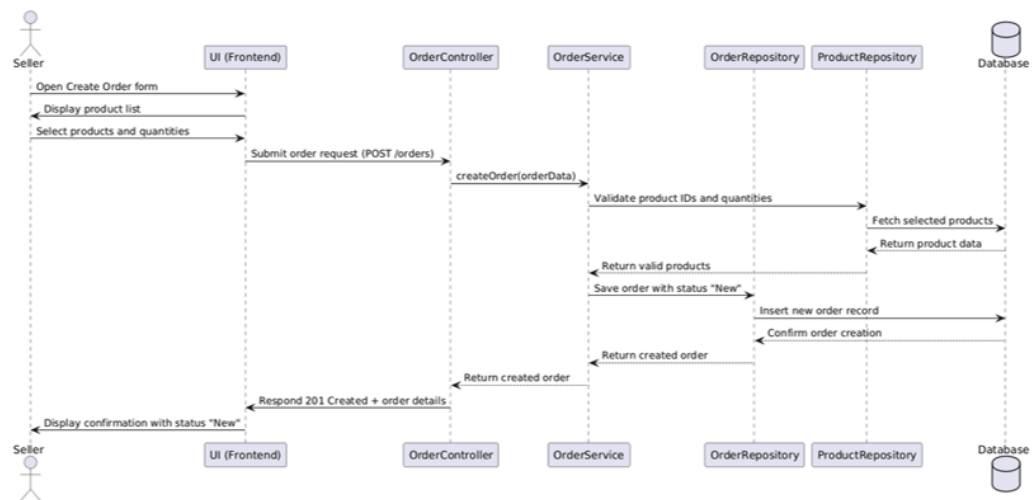
7

ER-ДІАГРАМА РЕЛЯЦІЙНОЇ БАЗИ ДАНИХ СИСТЕМИ



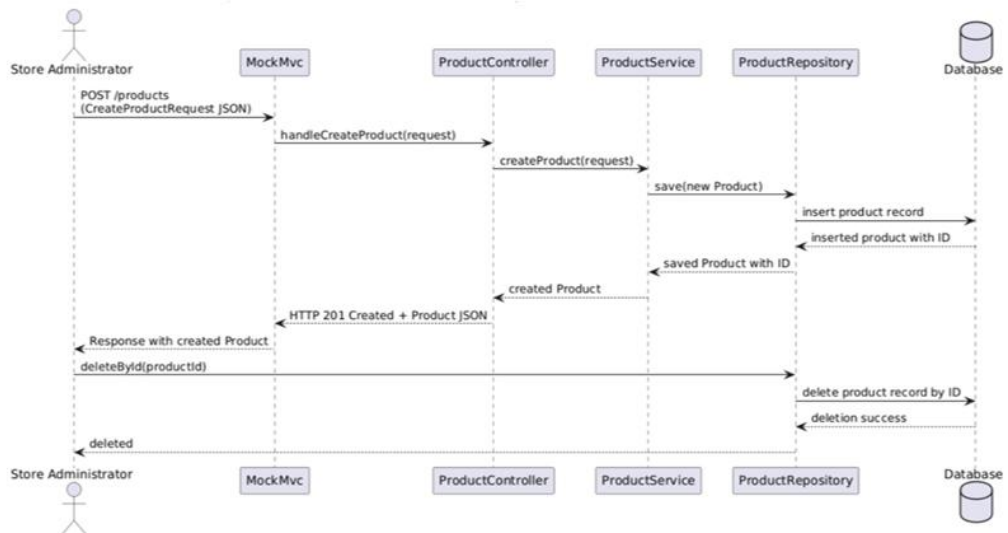
8

UML-ДІАГРАМА ПОСЛІДОВНОСТІ ПРОЦЕСУ СТВОРЕННЯ ЗАМОВЛЕННЯ



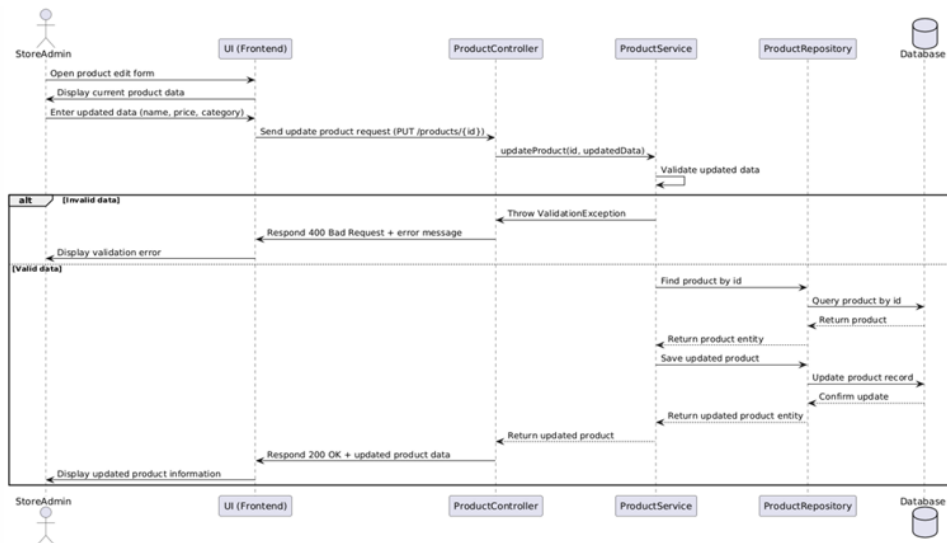
9

UML-ДІАГРАМА ПОСЛІДОВНОСТІ ПРОЦЕСУ ДОДАВАННЯ НОВОГО ТОВАРУ



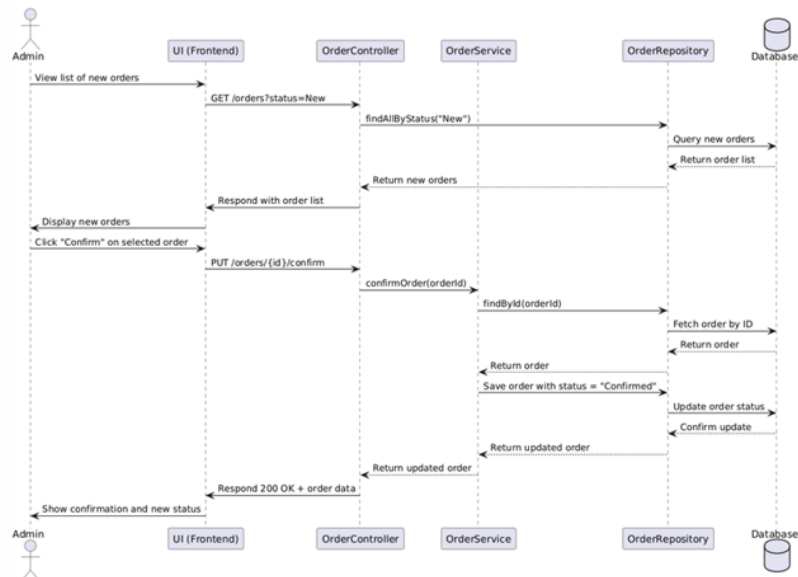
10

UML-ДІАГРАМА ПОСЛІДОВНОСТІ ПРОЦЕСУ ОНОВЛЕННЯ ІНФОРМАЦІЇ ПРО ТОВАР



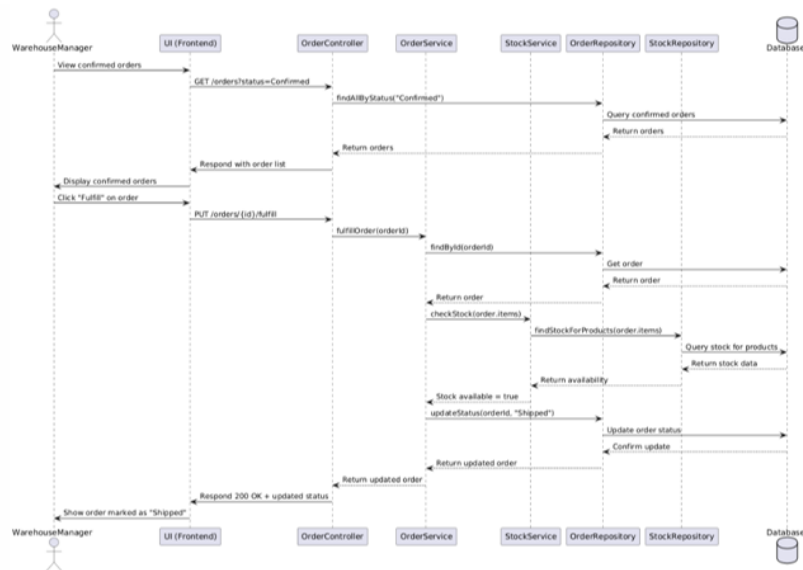
11

UML-ДІАГРАМА ПОСЛІДОВНОСТІ ПРОЦЕСУ ПІДТВЕРДЖЕННЯ ЗАМОВЛЕННЯ



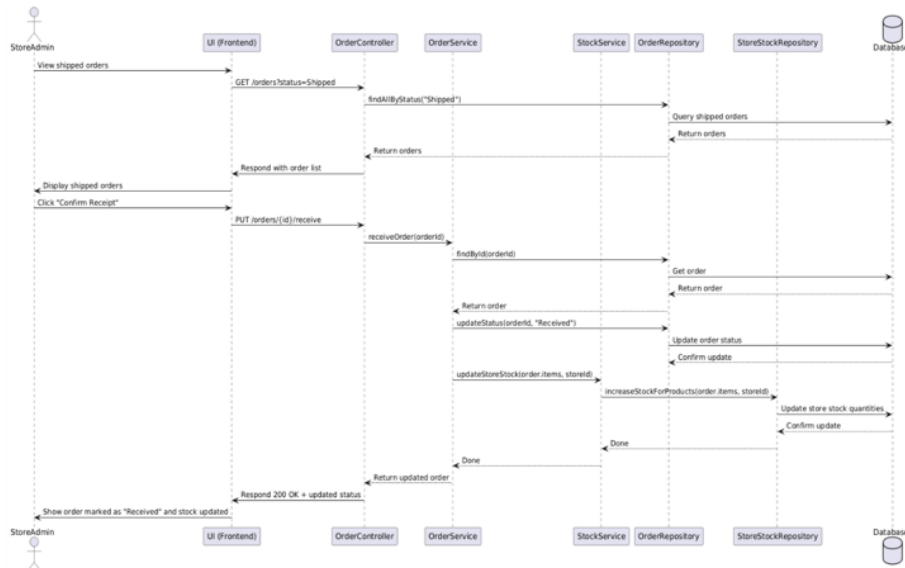
12

UML-ДІАГРАМА ПОСЛІДОВНОСТІ ПРОЦЕСУ ВИКОНАННЯ ЗАМОВЛЕННЯ



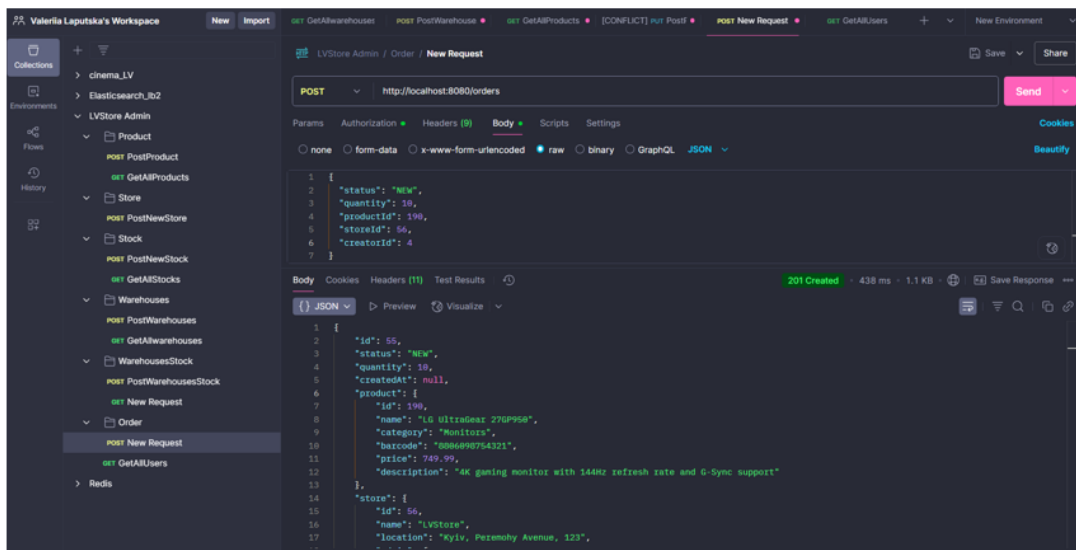
13

UML-ДІАГРАМА ПОСЛІДОВНОСТІ ПРОЦЕСУ ОТРИМАННЯ ТОВАРУ У МАГАЗИНІ



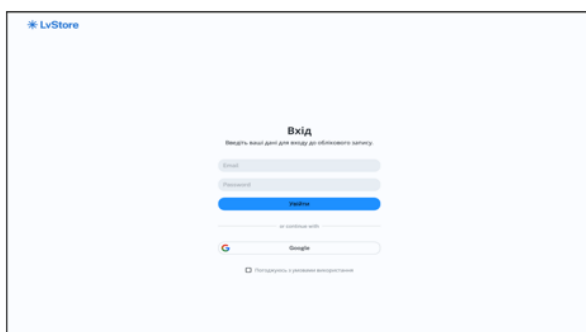
14

ТЕСТУВАННЯ REST API (СТВОРЕННЯ ЗАМОВЛЕННЯ)

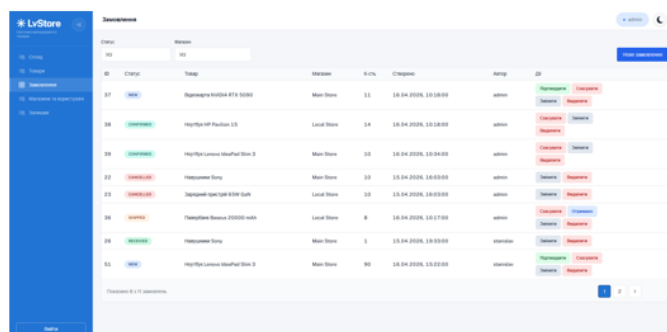


15

ЕКРАННІ ФОРМИ



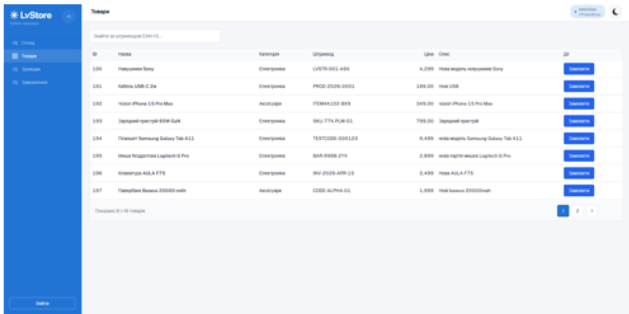
Авторизації користувача в системі «LVStore»



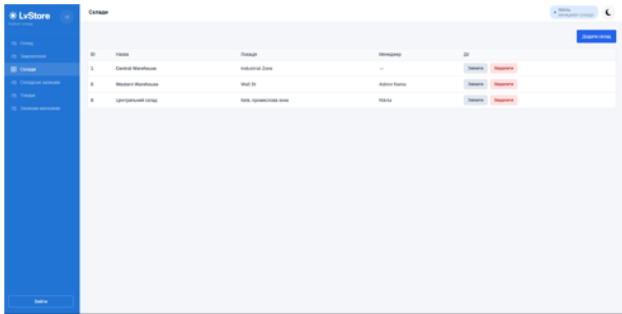
Управління замовленнями (роль: адміністратор)

16

ЕКРАННІ ФОРМИ



Каталог товарів (роль: продавець)



Управління складами (роль: менеджер складу)

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

Тези доповідей:

- 1. Лапуцька В.А., Замрій І.В. Кешування запитів у Spring Boot застосунку з використанням Redis. Матеріали VI Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24 квітня 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С. 203-205.
- 2. Лапуцька В.А., Замрій І.В. Впровадження контейнеризованого середовища для покращення масштабованості та відмовостійкості вебзастосунків. Матеріали II Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С. 77-79.
- 3. Лапуцька В.А., Замрій І.В. Механізми гарантування безпеки на основі розподілу ролей у Spring Boot застосунку на базі OAuth2 та JWT. Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 23 квітня 2026 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2026. (подано до друку).

ВИСНОВКИ

1. На основі проведеного дослідження специфіки логістики мережевого ритейлу та виявлених «вузьких місць» в управлінні товарообігом, сформовано чіткі технічні й бізнес-вимоги до розроблюваної системи.
2. На основі аналізу сучасних технічних засобів обрано оптимальний технологічний стек (Java, Spring Boot, PostgreSQL, Spring Security з JWT/OAuth2), що дозволило спроектувати нормалізовану ER-модель бази даних, впровадити строгу рольову матрицю доступу (RBAC) та побудувати стійку трирівневу архітектуру (DTO-патерни).
3. На основі спроектованої архітектури розроблено повнофункціональний REST API із захищеними ендпоінтами для керування замовленнями та імплементовано алгоритми транзакційного контролю залишків за допомогою Hibernate і Spring Data JPA.
4. За результатами проведеного багаторівневого тестування (JUnit 5, Mockito, MockMvc, Postman) підтверджено коректність роботи API, а шляхом розгортання середовища в ізольованих Docker-контейнерах забезпечено портативність, відмовостійкість та готовність системи до масштабування.

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

```
package org.example.lvstore.controller;

import
io.swagger.v3.oas.annotations.security.SecurityRequirement;
import lombok.RequiredArgsConstructor;
import org.example.lvstore.entity.WarehouseStock;
import org.example.lvstore.payload.warehousestock.CreateWarehouseStockRequest;
import org.example.lvstore.payload.warehousestock.UpdateWarehouseStockRequest;
import org.example.lvstore.service.WarehouseStockService;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/warehouse-stocks")
@RequiredArgsConstructor
@SecurityRequirement(name = "bearerAuth")
public class WarehouseStockController {

    private final WarehouseStockService warehouseStockService;

    @GetMapping
    public List<WarehouseStock>
    getAllWarehouseStocks() {
        return
        warehouseStockService.getAllWarehouseStocks();
    }

    @GetMapping("/{id}")
    public WarehouseStock
    getWarehouseStockById(@PathVariable Long id) {
        return
        warehouseStockService.getWarehouseStockById(id);
    }

    @PostMapping
    public WarehouseStock
    createWarehouseStock(@RequestBody
    CreateWarehouseStockRequest request) {
        return
        warehouseStockService.createWarehouseStock(request);
    }

    @PutMapping
    public WarehouseStock
    updateWarehouseStock(@RequestBody
    UpdateWarehouseStockRequest request) {
        return
        warehouseStockService.updateWarehouseStock(request);
    }
}
```

```
@DeleteMapping("/{id}")
public void deleteWarehouseStock(@PathVariable
Long id) {
    warehouseStockService.deleteWarehouseStock(id);
}

@GetMapping(params = "warehouseId")
public List<WarehouseStock>
getStocksByWarehouse(@RequestParam Long
warehouseId) {
    return
    warehouseStockService.getWarehouseStocksByWarehouseId(warehouseId);
}

@GetMapping(params = "productId")
public List<WarehouseStock>
getStocksByProduct(@RequestParam Long productId) {
    return
    warehouseStockService.getWarehouseStocksByProductId(productId);
}
}
```

```
package org.example.lvstore.controller;

import
io.swagger.v3.oas.annotations.security.SecurityRequirement;
import lombok.RequiredArgsConstructor;
import org.example.lvstore.entity.Warehouse;
import org.example.lvstore.payload.warehouse.CreateWarehouseRequest;
import org.example.lvstore.payload.warehouse.UpdateWarehouseRequest;
import org.example.lvstore.service.WarehouseService;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequiredArgsConstructor
@RequestMapping("/warehouses")
@SecurityRequirement(name = "bearerAuth")
public class WarehouseController {

    private final WarehouseService warehouseService;

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
}
```

```

    public ResponseEntity<Warehouse>
createWarehouse(@RequestBody
CreateWarehouseRequest request) {
    return
ResponseEntity.ok(warehouseService.createWarehouse(r
equest));
}

    @GetMapping
    public ResponseEntity<List<Warehouse>>
getAllWarehouses() {
    return
ResponseEntity.ok(warehouseService.getAllWarehouses(
));
}

    @GetMapping("/{id}")
    public ResponseEntity<Warehouse>
getWarehouseById(@PathVariable Long id) {
    return
ResponseEntity.ok(warehouseService.getWarehouseById
(id));
}

    @PutMapping
    public ResponseEntity<Warehouse>
updateWarehouse(@RequestBody
UpdateWarehouseRequest request) {
    try {
    return
ResponseEntity.ok(warehouseService.updateWarehouse(r
equest));
    } catch (RuntimeException e) {
    return ResponseEntity.notFound().build();
    }
}

    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void deleteWarehouse(@PathVariable Long id)
{
    warehouseService.deleteWarehouse(id);
}
}

```

```
package org.example.lvstore.controller;
```

```

import
io.swagger.v3.oas.annotations.security.SecurityRequirem
ent;
import lombok.RequiredArgsConstructor;
import org.example.lvstore.entity.User;
import
org.example.lvstore.payload.user.CreateUserRequest;
import
org.example.lvstore.payload.user.UpdateUserRequest;
import org.example.lvstore.service.UserService;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

```

```
import java.util.List;
```

```

@RestController
@RequiredArgsConstructor
@RequestMapping("/users")
@SecurityRequirement(name = "bearerAuth")
public class UserController {

    private final UserService userService;

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public ResponseEntity<User>
createUser(@RequestBody CreateUserRequest
createUserRequest) {
    return
ResponseEntity.ok(userService.createUser(createUserReq
uest));
}

    @GetMapping
    public ResponseEntity<List<User>> getAllUsers() {
    return
ResponseEntity.ok(userService.getAllUsers());
}

    @GetMapping("/{id}")
    public ResponseEntity<User>
getUserById(@PathVariable Long id) {
    return
ResponseEntity.ok(userService.getUserById(id));
}

    @GetMapping(params = "username")
    public ResponseEntity<User>
getUserByUsername(@RequestParam String username) {
    return
ResponseEntity.ok(userService.getUserByUsername(user
name));
}

    @PutMapping
    public ResponseEntity<User>
updateUser(@RequestBody UpdateUserRequest
updateUserRequest) {
    try {
    User updatedUser =
userService.updateUser(updateUserRequest);
    return ResponseEntity.ok(updatedUser);
    } catch (RuntimeException e) {
    return ResponseEntity.notFound().build();
    }
}

    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void deleteUser(@PathVariable Long id) {
    userService.deleteUser(id);
}
}

```

```

package org.example.lvstore.controller;

import
io.swagger.v3.oas.annotations.security.SecurityRequirem
ent;
import lombok.RequiredArgsConstructor;
import org.example.lvstore.entity.Store;
import
org.example.lvstore.payload.store.CreateStoreRequest;
import
org.example.lvstore.payload.store.UpdateStoreRequest;
import org.example.lvstore.service.StoreService;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequiredArgsConstructor
@RequestMapping("/stores")
@SecurityRequirement(name = "bearerAuth")
public class StoreController {

    private final StoreService storeService;

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public ResponseEntity<Store>
createStore(@RequestBody CreateStoreRequest request)
{
    return
ResponseEntity.ok(storeService.createStore(request));
}

    @GetMapping
    public ResponseEntity<List<Store>> getAllStores() {
        return
ResponseEntity.ok(storeService.getAllStores());
    }

    @GetMapping("/{id}")
    public ResponseEntity<Store>
getStoreById(@PathVariable Long id) {
        return
ResponseEntity.ok(storeService.getStoreById(id));
    }

    @PutMapping
    public ResponseEntity<Store>
updateStore(@RequestBody UpdateStoreRequest
request) {
        try {
            return
ResponseEntity.ok(storeService.updateStore(request));
        } catch (RuntimeException e) {
            return ResponseEntity.notFound().build();
        }
    }

    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)

```

```

    public void deleteStore(@PathVariable Long id) {
        storeService.deleteStore(id);
    }
}

package org.example.lvstore.controller;

import
io.swagger.v3.oas.annotations.security.SecurityRequirem
ent;
import lombok.RequiredArgsConstructor;
import org.example.lvstore.entity.Store;
import
org.example.lvstore.payload.store.CreateStoreRequest;
import
org.example.lvstore.payload.store.UpdateStoreRequest;
import org.example.lvstore.service.StoreService;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequiredArgsConstructor
@RequestMapping("/stores")
@SecurityRequirement(name = "bearerAuth")
public class StoreController {

    private final StoreService storeService;

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public ResponseEntity<Store>
createStore(@RequestBody CreateStoreRequest request)
{
    return
ResponseEntity.ok(storeService.createStore(request));
}

    @GetMapping
    public ResponseEntity<List<Store>> getAllStores() {
        return
ResponseEntity.ok(storeService.getAllStores());
    }

    @GetMapping("/{id}")
    public ResponseEntity<Store>
getStoreById(@PathVariable Long id) {
        return
ResponseEntity.ok(storeService.getStoreById(id));
    }

    @PutMapping
    public ResponseEntity<Store>
updateStore(@RequestBody UpdateStoreRequest
request) {
        try {
            return
ResponseEntity.ok(storeService.updateStore(request));
        } catch (RuntimeException e) {

```

```

        return ResponseEntity.notFound().build();
    }
}

@DeleteMapping("/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deleteStore(@PathVariable Long id) {
    storeService.deleteStore(id);
}
}

package org.example.lvstore.controller;

import
io.swagger.v3.oas.annotations.security.SecurityRequirem
ent;
import lombok.RequiredArgsConstructor;
import org.example.lvstore.entity.Stock;
import
org.example.lvstore.payload.stock.CreateStockRequest;
import
org.example.lvstore.payload.stock.UpdateStockRequest;
import org.example.lvstore.service.StockService;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequiredArgsConstructor
@RequestMapping("/stocks")
@SecurityRequirement(name = "bearerAuth")
public class StockController {

    private final StockService stockService;

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public ResponseEntity<Stock>
createStock(@RequestBody CreateStockRequest request)
{
    return
ResponseEntity.ok(stockService.createStock(request));
}

    @GetMapping
    public ResponseEntity<List<Stock>> getAllStocks() {
        return
ResponseEntity.ok(stockService.getAllStocks());
    }

    @GetMapping("/{id}")
    public ResponseEntity<Stock>
getStockById(@PathVariable Long id) {
        return
ResponseEntity.ok(stockService.getStockById(id));
    }

    @PutMapping

```

```

    public ResponseEntity<Stock>
updateStock(@RequestBody UpdateStockRequest
request) {
        try {
            return
ResponseEntity.ok(stockService.updateStock(request));
        } catch (RuntimeException e) {
            return ResponseEntity.notFound().build();
        }
    }

    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void deleteStock(@PathVariable Long id) {
        stockService.deleteStock(id);
    }

    @GetMapping(params = "storeId")
    public ResponseEntity<List<Stock>>
getStocksByStore(@RequestParam Long storeId) {
        return
ResponseEntity.ok(stockService.getStocksByStoreId(stor
eId));
    }

    @GetMapping(params = "productId")
    public ResponseEntity<List<Stock>>
getStocksByProduct(@RequestParam Long productId) {
        return
ResponseEntity.ok(stockService.getStocksByProductId(p
roductId));
    }

    @GetMapping("/critical")
    public ResponseEntity<List<Stock>>
getCriticalStocks(@RequestParam Long storeId) {
        return
ResponseEntity.ok(stockService.getCriticalStocksByStor
eId(storeId));
    }
}

package org.example.lvstore.controller;

import
io.swagger.v3.oas.annotations.security.SecurityRequirem
ent;
import lombok.RequiredArgsConstructor;
import org.example.lvstore.entity.Product;
import
org.example.lvstore.payload.product.CreateProductReque
st;
import
org.example.lvstore.payload.product.UpdateProductRequ
est;
import org.example.lvstore.service.ProductService;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.NoSuchElementException;

```

```

@RestController
@RequiredArgsConstructor
@RequestMapping("/products")
@SecurityRequirement(name = "bearerAuth")
public class ProductController {

    private final ProductService productService;

    @PostMapping
    public ResponseEntity<Product>
createProduct(@RequestBody CreateProductRequest
request) {
    try {
        return
ResponseEntity.status(HttpStatus.CREATED).body(prod
uctService.createProduct(request));
    } catch (IllegalArgumentException e) {
        return
ResponseEntity.status(HttpStatus.CONFLICT).build();
    }
}

    @GetMapping
    public ResponseEntity<List<Product>>
getAllProducts() {
        return
ResponseEntity.ok(productService.getAllProducts());
    }

    @GetMapping("/{id}")
    public ResponseEntity<Product>
getProductById(@PathVariable Long id) {
    try {
        return
ResponseEntity.ok(productService.getProductById(id));
    } catch (NoSuchElementException e) {
        return ResponseEntity.notFound().build();
    }
}

    @GetMapping(params = "barcode")
    public ResponseEntity<Product>
getProductByBarcode(@RequestParam String barcode) {
    try {
        return
ResponseEntity.ok(productService.getProductByBarcode(
barcode));
    } catch (NoSuchElementException e) {
        return ResponseEntity.notFound().build();
    }
}

    @PutMapping
    public ResponseEntity<Product>
updateProduct(@RequestBody UpdateProductRequest
request) {
    try {
        return
ResponseEntity.ok(productService.updateProduct(request
));
    } catch (NoSuchElementException e) {
        return ResponseEntity.notFound().build();
    }
}

```

```

    } catch (IllegalArgumentException e) {
        return
ResponseEntity.status(HttpStatus.CONFLICT).build();
    }
}

    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void deleteProduct(@PathVariable Long id) {
        productService.deleteProduct(id);
    }
}

package org.example.lvstore.controller;

import
io.swagger.v3.oas.annotations.security.SecurityRequirem
ent;
import lombok.RequiredArgsConstructor;
import org.example.lvstore.entity.Order;
import
org.example.lvstore.payload.order.CreateOrderRequest;
import
org.example.lvstore.payload.order.UpdateOrderRequest;
import org.example.lvstore.service.OrderService;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;
@RestController
@RequiredArgsConstructor
@RequestMapping("/orders")
@SecurityRequirement(name = "bearerAuth")
public class OrderController {

    private final OrderService orderService;

    @PostMapping
    public ResponseEntity<Order>
createOrder(@RequestBody CreateOrderRequest
createOrderRequest) {
        return
ResponseEntity.status(HttpStatus.CREATED).body(order
Service.createOrder(createOrderRequest));
    }

    @GetMapping
    public ResponseEntity<List<Order>> getAllOrders() {
        return
ResponseEntity.ok(orderService.getAllOrders());
    }

    @GetMapping("/{id}")
    public ResponseEntity<Order>
getOrderById(@PathVariable Long id) {
        return
ResponseEntity.ok(orderService.getOrderById(id));
    }

    @GetMapping(params = "storeId")

```

```

    public ResponseEntity<List<Order>>
getOrdersByStoreId(@RequestParam Long storeId) {
    return
ResponseEntity.ok(orderService.getOrdersByStoreId(storeId));
}
    @GetMapping(params = "status")
    public ResponseEntity<List<Order>>
getOrdersByStatus(@RequestParam String status) {
    return
ResponseEntity.ok(orderService.getOrdersByStatus(status));
}

    @PutMapping
    public ResponseEntity<Order>
updateOrder(@RequestBody UpdateOrderRequest
updateOrderRequest) {
    try {
        Order updatedOrder =
orderService.updateOrder(updateOrderRequest);
        return ResponseEntity.ok(updatedOrder);
    } catch (RuntimeException e) {
        return ResponseEntity.notFound().build();
    }
}
    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void deleteOrder(@PathVariable Long id) {
        orderService.deleteOrder(id);
    }

    @PutMapping("/{id}/confirm")
    public ResponseEntity<Order>
confirmOrder(@PathVariable Long id) {
    return
ResponseEntity.ok(orderService.confirmOrder(id));
}

    @PutMapping("/{id}/cancel")
    public ResponseEntity<Order>
cancelOrder(@PathVariable Long id) {
    return
ResponseEntity.ok(orderService.cancelOrder(id));
}

    @PutMapping("/{id}/ship")
    public ResponseEntity<Order>
shipOrder(@PathVariable Long id) {
    return
ResponseEntity.ok(orderService.shipOrder(id));
}

    @PutMapping("/{id}/deliver")
    public ResponseEntity<Order>
deliverOrder(@PathVariable Long id) {
    return
ResponseEntity.ok(orderService.markAsDelivered(id));
}
}
package org.example.lvstore.controller;

```

```

import jakarta.servlet.http.HttpServletResponse;
import lombok.RequiredArgsConstructor;
import
org.springframework.web.bind.annotation.GetMapping;
import
org.springframework.web.bind.annotation.RestController;

import java.io.IOException;

@RestController
@RequiredArgsConstructor
public class AuthenticationController {

    @GetMapping("/login")
    public void login(HttpServletResponse response)
throws IOException {

        response.sendRedirect("/oauth2/authorization/google");

    }
}

```