Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Навчально-науковий
інститут електроенергетики
(навчально-науковий інститут)
Факультет інформаційних технологій
(факультет)
Кафедра інформаційних технологій та комп'ютерної інженерії
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня магістра

Здобувача вищої освіти Гордєєва Артема Юрійовича
(ПІБ)
академічної групи 123м-23-1
(шифр)
спеціальності 123 Комп'ютерна інженерія
(код і назва спеціальності)
за освітньо-професійною програмою «Комп'ютерна інженерія»
(офіційна назва)

на тему «Комп'ютерна система складського обліку підприємства «АКС-ЮГ СИСТЕМА» з використанням телеграм-боту»
(назва за наказом ректора)

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
кваліфікаційної роботи	проф. Олевський В.І.			
розділів:				
синтез системи	доц. Бешта Д.О.			
розроблення програмного забезпечення	ас. Панфьорова Я.В.			
Рецензент				
Нормоконтролер	проф. Цвіркун Л.І.			

Дніпро
2024

ЗАТВЕРДЖЕНО:
завідувач кафедри
інформаційних технологій
та комп'ютерної інженерії
(повна назва)
_____ _В.В. Гнатушенко
(підпис) (ініціали, прізвище)

«_____» _____
2024 року

ЗАВДАННЯ
на кваліфікаційну роботу
ступеня магістра
(бакалавра, магістра)

здобувача вищої освіти Гордєєва А.Ю. **академічної групи** 123м-23-1
(прізвище та ініціали) (шифр)

спеціальності 123 Комп'ютерна інженерія

за освітньою-професійною програмою «Комп'ютерна інженерія»
(офіційна назва)

на тему «Комп'ютерна система складського обліку підприємства «АКС-ЮГ СИСТЕМА» з використанням телеграм-боту»,
затверджену наказом ректора НТУ «Дніпровська політехніка» від 17.10. 2024 р. №1388-с

Розділ	Зміст	Термін виконання
Стан питання та постановка завдання	Стан питання полягає у відсутності комп'ютерної системи складського обліку підприємства з використанням телеграм-боту	02.11.2024
Теоретичний	Обґрунтування комп'ютерної системи телеграм-бота та серверної частини	10.11.2024
Синтез системи	Розробка синтезу комп'ютерної системи	17.12.2024
Розроблення програмного забезпечення	Створення телеграм-бота, проєктування бази даних і розробка серверної частини	19.12.2024
Експериментальний розділ	Виконання тестування та аналіз результатів роботи телеграм-бота	22.12.2024

Завдання видано _____ проф. В. І. Олевський
(підпис керівника) (ініціали, прізвище)

Дата видачі 06 вересня 2024 р.

Дата подання до екзаменаційної комісії 24.12.2024 р.

Прийнято до виконання _____ Гордєєв А.Ю.
(підпис здобувача вищої освіти) (ініціали, прізвище)

РЕФЕРАТ

Пояснювальна записка 00 с., 00 с., 00 с., 1 дод., 00 джерел

TYPESCRIPT, REACTJS, NESTJS, NODEJS, JAVASCRIPT, POSTGRESQL, TELEGRAM, ТЕЛЕГРАМ–БОТ.

Об’єкт розробки: Телеграм–бот, для якого буде розроблена серверна та клієнтська частини, спроектована база даних для системи складського обліку підприємства «АКС-ЮГ СИСТЕМА».

Мета розробки: створення інтегрованої комп'ютерної системи складського обліку для підприємства «АКС-ЮГ СИСТЕМА» з використанням телеграм-бота, яка забезпечує автоматизацію обліку товарів, ефективну взаємодію з користувачами, зручний доступ до даних про залишки на складі та управління процесами в реальному часі.

Пояснювальна записка має аналіз існуючих систем складського обліку, в записці описуються їх переваги та недоліки.

Сформульоване завдання дослідження було зроблено на основі вже існуючих даних.

В розділі «Опис програмних засобів та технології проектування» описані обрані технології для реалізації поставленої задачі.

В розділі «Синтез комп’ютерної системи» описані необхідні характеристики для серверу.

В розділі «Проектування бази даних, розробка серверної частини та створення web додатку для телеграм-бота» описано процес проектування бази даних для обліку товарів, розробку серверної частини для обробки запитів, а також створення веб-додатку для інтеграції з телеграм-ботом, що забезпечує автоматизацію складських операцій.

В розділі «Дослідження» описується дослідження створеного програмного забезпечення на помили та його стресостійкість.

ЗМІСТ

Реферат……3
Перелік скорочень, умовних позначень, одиниць, термінів……7
Вступ……8
1 Стан питання та постановка завдання……9
1.1 Актуальність теми……..9
1.2 Аналіз існуючих систем автоматизації складського обліку……..10
1.2.1 Існуючі системи автоматизації складського обліку……...10
1.2.2 Сучасні рішення з використанням телеграм-ботів у автоматизації….13
1.3 Проблеми існуючих систем та необхідність удосконалення……..16
1.4 Постановка завдання дослідження……17
2 Опис програмних засобів та технологій проектування……...19
2.1 Мова SQL та СУБД PostgreSQL……..…19
2.2 pgAdmin як інструмент адміністрування……...20
2.3 Інструмент для розробки VS Code та середовище виконання Node.js..20
2.4 Фреймворк NestJS і мова TypeScript……..…22
2.5 React та JSX……..…22
2.6 Telegram web-api……..…23
2.7 Використані паттерни та архітектура проектування……23
2.7.1 Паттерн MVC……23
2.7.2 Монолітна архітектура……..…24
2.8 Принципи SOLID……25
3 Розділ синтезу комп’ютерної системи……...…27
3.1 Обґрунтування вибору сервера для телеграм-боту……..…27
3.1.1 Обґрунтування вибору хмарного сервера для телеграм-бота……..…29
3.1.2 Основні причини вибору хмарного сервера……..…32
3.1.3 Приклад обраного хмарного сервера……..….32
3.2 Обґрунтування вибору бази даних для збереження інформації про складські операції……..…35

3.2.1 Переваги використання PostgreSQL……...35
3.2.2 Приклад обраної бази даних PostgreSQL……36
3.3 Технічні вимоги до пристроїв операторів складу для використання телеграм-бота……38
3.3.1 Вимоги до операційної системи…….38
3.3.2 Вимоги до апаратних характеристик…….38
3.3.3 Вимоги до інтернет-з'єднання……39
3.3.4 Вимоги до оператора складу……39
3.4 Функціональна схема обміну даними……40
3.5 Висновки синтезу комп'ютерної системи……42
4 Проєктування бази даних, розробка серверної частини і ствонення Web додатку для телеграм-бота……44
4.1 Функціональне призначення програмного забезпечення……44
4.2 Технічні характеристики програмного забезпечення……...…45
4.2.1 Постановка завдання на розробку програмного забезпечення……45
4.2.2 Опис алгоритму роботи програмного забезпечення……..…45
4.2.3 Опис організації вхідних та вихідних даних……..…46
4.3 Опис розробленого програмного забезпечення……46
4.3.1 Опис та проектування бази даних……...…46
4.3.2 Опис серверної частини……..….49
4.3.2.1 Архітектура серверної частини……49
4.3.2.2 Рівень обробки вхідної інформації ……..…50
4.3.2.3 Рівень обробки бізнес-логіки……...…52
4.3.2.4 Рівень доступу до. Інформації з бази даних……54
4.3.3 Опис клієнтської частини……56
4.3.3.1 Архітектура клієтської частини……56
4.3.3.2 Рівень компонентів……56
4.3.3.3 Рівень контейнерів……..…57

4.3.4 Функціональне призначення програмного забезпечення……..…59
4.4 Опис та демонстраційні матеріали роботи функціонування програманого забезпечення……..…59
4.4.1 Функціонування програмного забезпечення зі сторони користувача.60
4.4.1.1 Робота з товарами……..……60
4.4.1.2 Робота з поставками……..65
4.4.1.3 Робота з замовленнями…….……71
4.5 Висновки до розділу розробки програмного забезпечення……76
5 Дослідження……...…77
5.1 Мета дослідження……...…77
5.2 Методологія дослідження……...77
5.3 Тестування обробки помилок web додатку……77
5.3.1 Тестування модуля «Товари» ……..77
5.3.2 Тестування модуля «Замовлення» ……..…81
5.3.3 Тестування модуля «Постачання» ……..85
5.4 Тестування стресостійкості серверної частини……88
5.4.1 Тестування GET запиту продуктів……...…89
5.4.2 Тестування GET запиту продуктів з фільтрацією…….…...89
5.5 Результати дослідження……..…90
Висновки……92
Перелік посилань……...93
Додаток А – UML діаграми..……96
Додаток Б – Текст програми…….……99

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ОДИНИЦЬ, ТЕРМІНІВ

CRM – система управління взаємодією з клієнтами

ERP – система планування ресурсів підприємства

NLP – обробка природної мови

FAQ – часті запитання

CPU – центральний процесор

PNG – растровий формат збереження графічної інформації

SVG – масштабована векторна графіка

PDF – міжплатформенний відкритий формат електронних документів

VPS – віртуальний виділений сервер

ВСТУП

Сучасний світ диктує необхідність автоматизації бізнес-процесів, особливо для компаній, які працюють із великим обсягом даних, як-от склади. Автоматизація дозволяє зменшити кількість ручної роботи, уникнути помилок і значно підвищити ефективність управління. Одним із простих, доступних та водночас потужних інструментів для таких завдань є телеграм-боти. Вони дозволяють швидко отримувати інформацію, взаємодіяти зі складською системою та навіть проводити базову аналітику.

Однак аналіз існуючих рішень показує, що багато систем автоматизації складського обліку є складними у впровадженні, дорогими або не враховують сучасних вимог, таких як мобільність і зручність використання. Такі системи часто не інтегруються з іншими платформами, що ускладнює їхнє використання на малих і середніх підприємствах.

Світові тенденції показують, що автоматизація складів усе частіше спирається на хмарні технології, інтеграцію з мобільними платформами та використання штучного інтелекту. Зокрема, телеграм-боти стають популярними завдяки їхній доступності, простоті впровадження та багатому функціоналу. Вони дозволяють отримувати актуальну інформацію про складські залишки, проводити інвентаризацію чи формувати звіти без складного навчання персоналу.

1 СТАН ПИТАННЯ ТА ПОСТАНОВКА ЗАВДАННЯ

1.1 Актуальність теми

Автоматизація бізнес-процесів стала однією з головних тенденцій сучасного управління, особливо у сфері логістики та складського обліку. Складський облік є ключовим елементом функціонування підприємств, що займаються виробництвом, торгівлею або дистрибуцією товарів. Від його точності, оперативності та ефективності залежить не лише фінансова стабільність компанії, а й її конкурентоспроможність на ринку.

Сучасні виклики управління складськими процесами включають:

а) зростання обсягів даних через збільшення кількості товарів та їхніх характеристик;

б) потребу в оперативному доступі до інформації для прийняття рішень у режимі реального часу;

в) необхідність зниження кількості людських помилок при обліку товарів;

г) підвищення рівня безпеки та захищеності даних.

Ручний облік або використання застарілих систем вже не відповідають вимогам сучасного бізнесу, оскільки вони часто є причиною помилок, затримок та невиправданих витрат. Традиційні методи складського обліку не забезпечують достатньої гнучкості та інтеграції з іншими системами управління підприємством.

З іншого боку, використання телеграм-ботів у складі автоматизованих систем обліку є відносно новим та перспективним напрямом. Телеграм-боти дозволяють:

а) організувати швидкий і зручний доступ до складської інформації через мобільні пристрої;

б) забезпечити інтерактивний режим взаємодії користувачів із системою;

в) автоматизувати окремі процеси, такі як формування звітів, відстеження залишків, оповіщення про зміни в складі тощо;

г) підвищити зручність використання системи для працівників складу, керівництва та інших зацікавлених осіб.

На підприємстві «АКС-ЮГ СИСТЕМА», яке займається дистрибуцією та зберіганням товарів, гостро стоїть проблема оптимізації складського обліку. Поточна система обліку не здатна забезпечити необхідну оперативність і точність, що негативно впливає на ефективність діяльності компанії. Запровадження автоматизованої системи, інтегрованої з телеграм-ботом, дозволить:

а) мінімізувати вплив людського фактора на складські процеси;

б) забезпечити миттєвий доступ до складської інформації;

в) підвищити ефективність управління залишками товарів;

г) скоротити час на виконання облікових операцій.

Таким чином, тема дипломного проєкту є актуальною, оскільки вона спрямована на вирішення важливої проблеми автоматизації складського обліку за допомогою сучасних інформаційних технологій. Реалізація цього проєкту матиме практичну цінність не лише для конкретного підприємства, а й для інших організацій, які прагнуть оптимізувати свої бізнес-процеси.

1.2 Аналіз існуючих систем автоматизації складського обліку

1.2.1 Існуючі системи автоматизації складського обліку

Сьогодні на ринку представлено широкий спектр програмних продуктів, що забезпечують автоматизацію складського обліку. Ці системи спрямовані на оптимізацію процесів управління складськими операціями, підвищення точності обліку та зменшення витрат на обробку даних. Серед популярних рішень можна виділити такі категорії:

а) ERP-системи з модулем складського обліку;

ERP-системи (Enterprise Resource Planning) інтегрують усі основні бізнес-процеси підприємства, включаючи управління складом. Відомими продуктами є:[1]

1) SAP ERP – забезпечує управління всіма аспектами складської діяльності, включаючи контроль запасів, переміщення товарів і логістику. Проте висока вартість та складність впровадження є основними бар'єрами для малих і середніх підприємств;

2) 1С:Підприємство – популярна в Україні система, яка містить модулі для автоматизації складського обліку, зокрема, управління залишками, формування звітів та інтеграцію з бухгалтерським обліком. Вона доступна для малого та середнього бізнесу, але її функціонал часто потребує налаштування під конкретні потреби підприємства;

б) спеціалізовані системи складського обліку (WMS);

Системи управління складом (Warehouse Management System, WMS) розроблені спеціально для автоматизації складських операцій. Основні функції таких систем включають:[2]

1) контроль надходження та відвантаження товарів;
2) оптимізацію розміщення товарів на складі;
3) відстеження залишків і термінів придатності продукції;
4) управління інвентаризацією.

Приклади:

1) Logistix – гнучка система для оптимізації складських процесів, що забезпечує інтеграцію з обладнанням, таким як сканери штрих-кодів;
2) ProWMS – орієнтована на великі склади, забезпечує управління логістичними потоками, планування вантажів і маршрутизацію.

в) хмарні платформи складського обліку;

Останнім часом все більш популярними стають хмарні рішення, які дозволяють використовувати складські програми через інтернет.[3]

1) Zoho Inventory – система для управління запасами та складом із доступом через веб-інтерфейс. Підходить для малого та середнього бізнесу;
2) TradeGecko – платформа, яка інтегрується з інтернет-магазинами, пропонує функції управління запасами та автоматизації замовлень.

г) прості рішення для невеликих складів;

Для малих підприємств часто використовуються прості програми на базі Excel або локальні програми з мінімальним функціоналом. Вони включають:[4?]

1) ведення картотеки товарів.
2) формування звітів про залишки.

Приклади:

1) локальні програми типу "Облік складу" або "Складський асистент".
2) переваги існуючих систем:
3) зменшення кількості помилок завдяки автоматизації;
4) швидке отримання актуальної інформації про запаси;
5) оптимізація складських процесів через інтеграцію із сучасними технологіями, такими як штрих-кодування або RFID.

Недоліки існуючих систем:

1) висока вартість впровадження та обслуговування для малих підприємств;
2) обмежений функціонал у простих рішеннях, що не дозволяє повністю автоматизувати процеси;
3) відсутність адаптації під сучасні комунікаційні технології, такі як інтеграція з месенджерами (наприклад, телеграм-ботами), що важливо для мобільності та зручності використання.

Таким чином, існуючі системи автоматизації складського обліку мають широкий функціонал, але не завжди відповідають потребам підприємств, які шукають бюджетні, мобільні та інтегровані рішення. Це обґрунтовує необхідність створення нових гнучких рішень, таких як використання телеграм-ботів у складі систем складського обліку.

1.2.2. Сучасні рішення з використанням телеграм-ботів у автоматизації

Телеграм-боти активно набувають популярності як інструменти автоматизації бізнес-процесів завдяки їх простоті, доступності та можливостям інтеграції з різними системами. Використання телеграм-ботів для автоматизації складського обліку дозволяє зробити доступ до складської інформації швидким, зручним і мобільним, що особливо актуально в умовах сучасних вимог до управління бізнесом.

Переваги використання телеграм-ботів у складському обліку:

а) мобільність. Телеграм-боти забезпечують доступ до складських даних із будь-якого місця за допомогою смартфона чи комп’ютера, що значно спрощує управління складом у реальному часі;

б) інтерактивність. Бот може реагувати на запити користувачів у режимі реального часу, наприклад:

1) видавати інформацію про залишки товарів;
2) надавати звіти щодо надходжень і відвантажень;
3) повідомляти про критичний рівень запасів або термін придатності продукції.

в) інтеграція з існуючими системами. Телеграм-боти можуть працювати як фронтенд для складських програм або баз даних. Вони дозволяють:

1) виконувати команди для отримання даних зі складського обліку;
2) інтегруватися з іншими системами обліку через API.

г) автоматизація завдань. За допомогою телеграм-ботів можна автоматизувати рутинні операції, такі як формування звітів, оповіщення про зміни в залишках або управління інвентаризацією;

в) масштабованість. Боти можуть легко масштабуватися разом із зростанням обсягу складських даних і кількістю користувачів.

Приклади застосування телеграм-ботів у складському обліку:

а) моніторинг залишків товарів. Телеграм-боти дозволяють швидко отримати дані про залишки на складі. Наприклад, користувач може надіслати команду /залишки, і бот поверне актуальний список товарів із кількістю на складі;

б) повідомлення про критичні події. Бот може автоматично надсилати повідомлення у випадках:

1) досягнення критичного рівня запасів;
2) завершення терміну придатності продукції;
3) несподіваних розходжень у результатах інвентаризації.

в) інтеграція з логістичними процесами. Телеграм-боти можуть використовуватися для управління відвантаженнями та надходженнями товарів. Наприклад, вони можуть надсилати водіям або менеджерам списки товарів для відвантаження чи підтвердження надходження;

г) автоматизація інвентаризації. Бот може допомагати у процесі інвентаризації, надаючи інтерактивний доступ до бази даних товарів і спрощуючи внесення результатів у систему;

д) технологічні можливості телеграм-ботів. Телеграм-боти створюються за допомогою API Telegram, яке надає широкі можливості для розробки:

1) обробка текстових команд користувача.
2) інтеграція з базами даних (наприклад, MySQL, PostgreSQL).
3) підключення до сторонніх сервісів через вебхуки або API.
4) використання клавіатури та меню для спрощення взаємодії.

Недоліки та обмеження телеграм-ботів:

а) безпека. Якщо не приділити достатньо уваги захисту даних, можуть виникнути ризики втрати чи крадіжки інформації. Для цього необхідно впроваджувати методи шифрування даних і контролю доступу;

б) обмежений інтерфейс. Телеграм-боти працюють у текстовому або інтерактивному (кнопковому) режимі, що може бути недостатньо зручним для складних аналітичних операцій;

в) необхідність стабільного інтернет-з’єднання. Для роботи бота потрібен постійний доступ до мережі Інтернет, що може бути проблемою в деяких умовах;

г) перспективи використання телеграм-ботів у складському обліку. Використання телеграм-ботів є перспективним напрямком розвитку систем автоматизації складського обліку, особливо для підприємств малого та середнього бізнесу, які прагнуть знизити витрати на впровадження складних ERP або WMS-систем. Інтеграція бота із системою складського обліку дозволить:

1) скоротити час виконання операцій;
2) підвищити точність обліку;
3) забезпечити мобільність та доступність інформації.

Таким чином, сучасні рішення з використанням телеграм-ботів є ефективним способом покращення автоматизації складських процесів, відповідаючи сучасним вимогам до гнучкості, мобільності та інтерактивності.

1.3 Проблеми існуючих систем та необхідність удосконалення

Автоматизація складського обліку є ключовим фактором підвищення ефективності роботи підприємств. Проте існуючі системи автоматизації, навіть найсучасніші, не завжди задовольняють усі потреби бізнесу. Це обумовлено низкою проблем, які спостерігаються в роботі цих систем, а також зростанням вимог до автоматизації та інтеграції сучасних технологій.

Проблеми існуючих систем складського обліку:

а) висока вартість впровадження та обслуговування:

1) багато ERP-систем або WMS-систем потребують значних фінансових витрат на придбання ліцензій, налаштування програмного забезпечення та навчання персоналу;
2) ця проблема особливо актуальна для малого та середнього бізнесу, який не має достатнього бюджету на впровадження складних систем.

б) складність інтеграції:

1) часто системи не забезпечують зручної інтеграції з іншими програмами чи сервісами, які вже використовує підприємство;
2) інтеграція може вимагати розробки спеціальних модулів або адаптації API, що ускладнює процес впровадження.

в) обмежена функціональність:

1) деякі програми мають вузький набір функцій, який не враховує специфічні вимоги підприємства, наприклад, можливість роботи з мобільними пристроями чи інтеграцію із сучасними месенджерами;
2) відсутність можливостей для віддаленого управління чи автоматизації звітності.

г) відсутність інтерактивності та мобільності:

1) більшість традиційних систем розраховані на роботу в локальній мережі або на стаціонарних пристроях;
2) відсутність мобільного доступу ускладнює швидке прийняття рішень, особливо у разі роботи у розподілених складах або у польових умовах.

ґ) складність у використанні:

1) інтерфейси багатьох систем є занадто складними для рядового користувача, що ускладнює процес навчання персоналу та збільшує кількість помилок під час роботи;
2) висока залежність від технічної підтримки постачальника програмного забезпечення.

д) безпека даних:

1) недостатній рівень захисту інформації або вразливість систем до кібератак можуть призвести до втрати чи крадіжки конфіденційних даних;
2) проблема актуальна для рішень, які не забезпечують багаторівневий захист і резервне копіювання даних.

е) обмеження в аналітичних можливостях:

1) багато систем не забезпечують гнучкого аналізу складських даних, що обмежує можливості для планування та оптимізації процесів.

1.4 Постановка завдання дослідження

Складський облік є одним із найважливіших елементів управління підприємством, оскільки від його ефективності залежить точність даних про запаси, своєчасність постачання та обслуговування клієнтів. Враховуючи актуальні проблеми існуючих систем автоматизації складського обліку та необхідність впровадження сучасних технологій, мета цього дослідження полягає у розробці комп'ютерної системи автоматизації складського обліку з інтеграцією телеграм-бота, яка забезпечить:

а) зручність доступу до складської інформації;

б) високу мобільність користувачів;

в) автоматизацію рутинних процесів, таких як обробка запитів, створення звітів та моніторинг залишків.

Основні завдання дослідження:

а) аналіз існуючих систем автоматизації складського обліку;

1) провести огляд сучасних підходів і технологій, що застосовуються в автоматизації складських процесів.

2) дослідити використання телеграм-ботів у сфері автоматизації.

б) розробка концепції системи;

1) розробити структуру комп'ютерної мережі для забезпечення надійного функціонування складського обліку;

2) визначити вимоги до програмного забезпечення та обладнання, необхідного для впровадження системи.

в) інтеграція телеграм-бота;

1) реалізувати інтеграцію телеграм-бота як інструмента взаємодії з користувачем;

2) забезпечити можливість автоматичної обробки запитів, надання звітів та оповіщення про зміни на складі.

г) розробка технічної документації.

1) скласти технічну документацію, включаючи специфікації системи, архітектуру комп'ютерної мережі та алгоритми роботи бота.

2 ОПИС ПРОГРАМНИХ ЗАСОБІВ ТА ТЕХНОЛОГІЙ ПРОЕКТУВАННЯ

2.1 Мова SQL та СУБД PostgreSQL

Насамперед перед вибором бази даних та СУБД необхідно ознайомитись з цими термінами.

База даних (БД) — це організований за певними правилами набір інформації (наприклад, пости, коментарі, імена (ніки) користувачів тощо), який можна легко редагувати та доповнювати. Класичні БД організовані у вигляді таблиці: кожен рядок виступає як окремий запис, а стовпець — його атрибут. Для виконання запитів використовується мова SQL. А зручне керування всією БД забезпечує система управління базами даних (СУБД) [1].

Система управління базами даних (СУБД) це програмне забезпечення для зберігання та отримання даних користувачів із врахуванням відповідних заходів безпеки. Він складається з групи програм, які маніпулюють базою даних. СУБД приймає запит на дані від програми та дає команду операційній системі надати певні дані. У великих системах СУБД допомагає користувачам та іншому програмному забезпеченню сторонніх виробників зберігати та отримувати дані [2]

Розрізняють наступні моделі бази даних [3]:

а) ієрархічні – у яких інформація у базі даних зберігається як об’єкти, що об’єднані в деревоподібну структуру. У порівнянні з іншими типами СУБД, ієрархічна СУБД має відносно невеликий арсенал операцій з маніпулювання даними (втім, цього достатньо для вирішення більшості завдань);

б) мережеві – СУБД, які як і ієрархічні мають деревоподібну структуру, але відрізняються від ієрархічних СУБД тим, що будь-яка запис-

нащадок може мати відразу кілька предків. Складність побудови такої БД компенсується добрими показниками швидкістю доступу та низькою витратою оперативної пам’яті;

в) реляційна – це інформація, де дані впорядковані, тобто пов’язані між собою певними особливими відносинами. По суті, така база є таблицею, в якій розміщені всі дані. Фізично бази даних – це файли у спеціальному форматі. Для роботи з цими файлами використовується спеціальне програмне забезпечення. Цей софт називається СУБД – система управління базами даних. Оскільки СУБД нерозривно пов’язана з базами даних, ці терміни часто використовуються як синоніми, що, строго кажучи, некоректно.

В якості СУБД для дипломної роботи була обрана PostgreSQL.

PostgreSQL — це потужна та гнучка СУБД, ідеально підходяща для різних типів додатків, від простих веб-сайтів до складних аналітичних систем. Її надійність, розширюваність та багатий набір функцій роблять її популярним вибором серед розробників та компаній по всьому світу [4].

2.2 pgAdmin як інструмент адміністрування

pgAdmin - це платформа з відкритим вихідним кодом для адміністрування та розробки для PostgreSQL і пов'язаних з нею систем управління базами даних. Платформа написана на Python і jQuery і підтримує всі функції PostgreSQL. pgAdmin використовується для будь-яких операцій, починаючи із запису базових SQL-запитів і закінчуючи здійсненням моніторингу ваших баз даних і налаштуванням просунутих архітектур баз даних [5].

2.3 Інструмент для розробки VS Code та середовище виконання Node.js

Інтегроване середовище розробки (IDE) — це програмний застосунок, який допомагає програмістам ефективно розробляти програмний код. Воно підвищує продуктивність розробників, об'єднуючи такі можливості, як

редагування, створення, тестування та упаковка програмного забезпечення в зручному для використання застосунку. [6].

Під час написання дипломної роботи була використана IDE Visual Studio Code. Вона була обрана завдяки її безкоштовності та сумісності з усіма популярними мовами програмування.

Visual Studio Code - це один із найпопулярніших редакторів коду, розроблений корпорацією Microsoft. Він поширюється в безоплатному доступі і підтримується всіма актуальними операційними системами: Windows, Linux і macOS. VS Code являє собою звичайний текстовий редактор з можливістю під'єднання різних плагінів, що дає можливість працювати зі всілякими мовами програмування для розробки будь-якого IT-продукту[7].

Node.js - це потужне середовище виконання для запуску JavaScript-коду поза веб-браузером. Воно дозволяє запускати JavaScript на стороні сервера, що дає змогу розробникам створювати масштабовані, високопродуктивні та керовані подіями застосунки.

Node.js дозволяє розробникам використовувати JavaScript як на стороні клієнта, так і на стороні сервера, забезпечуючи уніфіковану мову та екосистему. Це усуває необхідність перемикання контексту і дозволяє повторно використовувати код між фронтендом і бекендом. Це призводить до підвищення продуктивності та скорочення часу розробки [8].

Node.js було обрано для розробки web-API завдяки його неблокуючій моделі введення/виведення (I/O), яка дає змогу обробляти велику кількість запитів одночасно без затримок. Це забезпечує високу продуктивність веб-додатків, особливо тих, які потребують роботи з численними одночасними з'єднаннями та реалізації функцій у режимі реального часу.

2.4 Фреймворк NestJS і мова TypeScript

Як мову програмування було обрано TypeScript завдяки її повній сумісності з Node.js.

TypeScript – це мова програмування, яка являє собою надмножину JavaScript. Вона додає статичну типізацію і деякі додаткові функції, які допомагають розробникам створювати більш надійні та масштабовані додатки. Однак, незважаючи на свої розширені можливості, TypeScript зберігає сумісність зі стандартами JavaScript, що робить його легко інтегрованим в наявні проекти. У цьому матеріалі ми розглянемо: що таке typescript, які в нього можливості, навіщо він потрібен і для яких проєктів він може бути найкориснішим, переваги та недоліки його використання[9].

Nest (NestJS) — це платформа для створення ефективних, масштабованих серверних програм Node.js. Він використовує прогресивний JavaScript, створений і повністю підтримує TypeScript (все ще дозволяє розробникам кодувати на чистому JavaScript) і поєднує елементи ООП (об’єктно-орієнтоване програмування), FP (функціональне програмування) і FRP (функціональне реактивне програмування)[10]

Nest під капотом використовує надійні HTTP-фреймворки, такі як Express (за замовчуванням), і за необхідності може бути налаштований для роботи з Fastify. Він забезпечує вищий рівень абстракції порівняно зі звичайними фреймворками Node.js (Express/Fastify), але водночас надає доступ до їхніх API. Це дозволяє розробникам використовувати широкий вибір сторонніх модулів, доступних для базової платформи.

2.5 React та JSX

React JS — це відкритий JavaScript-фреймворк, а точніше, бібліотекою JavaScript, яка використовується для розробки інтерфейсів користувача. Він був створений компанією Facebook і швидко набув популярності серед розробників з усього світу. Реакт дозволяє ефективно створювати

застосунки з високою продуктивністю і масштабованістю. Одним з ключових концепцій у React JS є компоненти. Вони представляють собою незалежні блоки коду, які відповідають за рендеринг певної частини користувацького інтерфейсу[11].

React має зрозумілий синтаксис та добре структуровану документацію, що робить його доступним навіть для початківців. Він базується на концепціях JavaScript. Через це він і був обраний в якості інструмента для розробки графічного інтерфейсу.

JSX — це розширення синтаксису JavaScript, яке виглядає як XML. Ви можете використовувати просту трансформацію синтаксису JSX в React [12].

2.6 Telegram web-api

Telegram Bot API — це специфікація цього інтерфейсу, тобто довгий список методів і типів даних, який зазвичай називають довідкою. Він визначає все, що можуть робити боти Telegram. Посилання на нього можна знайти на вкладці «Ресурси» в розділі «Telegram»[13].

2.7 Використані паттерни та архітектура проектування

Патерни проектування – це перевірені часом рішення, які допомагають розробникам створювати надійне, гнучке та масштабоване ПЗ. Вони являють собою шаблони, засновані на загальних принципах і архітектурних рішеннях, які можна використовувати в різних ситуаціях. Їхня роль полягає в тому, що вони допомагають нам проектувати програми більш організовано, роблять код більш перевикористовуваним і спрощують підтримку системи [14].

2.7.1 Паттерн MVC

MVC (Model-View-Controller) — це архітектурний патерн, який розділяє програму на три основні компоненти [15]:

а) Model (Модель):

1) відповідає за роботу з даними, бізнес-логікою та правилами їх обробки;

2) зберігає стан програми та керує ним;

3) вона незалежна від уявлення (View) або контролера (Controller).

б) View (Уявлення):

1) відповідає за відображення інформації користувачу;

2) отримує дані від моделі та показує їх у зручному форматі;

3) реагує на зміни моделі, зазвичай через підписку на події.

в) Controller (Контролер):

1) посередник між моделлю та уявленням;

2) обробляє взаємодію користувача (наприклад, натискання кнопок, введення даних);

3) оновлює модель на основі дій користувача та забезпечує оновлення View.

Але в рамках цієї роботи характеристика View из MVC паттерна не буде використана на стороні web-api тому, що роль візуалізації даних буде виконувати web застосунок в telegram боті.

2.7.2 Монолітна архітектура

Монолітна архітектура (Monolithic Architecture) – це традиційний підхід до розробки програмного забезпечення, при якому весь додаток розробляється як одна єдина технологічна система. Всі компоненти додатку взаємодіють один з одним і розгортання відбувається на одному сервері або групі серверів.

Монолітна архітектура має декілька переваг. Вона зазвичай простіша в розробці та тестуванні, оскільки всі компоненти додатку взаємодіють безпосередньо один з одним, що дозволяє швидко та легко вирішувати проблеми. Крім того, монолітні додатки можуть бути менш складними в

управлінні, оскільки все програмне забезпечення працює на одній технологічній платформі.

Однак, монолітні додатки можуть стати проблемою при масштабуванні, оскільки розширення системи може бути обмеженим через те, що всі компоненти додатку залежать один від одного. Крім того, відносно складна структура монолітних додатків може ускладнити розробку та підтримку.

Отже, монолітна архітектура підходить для невеликих та середніх проектів зі стабільним обсягом функціоналу [16].

За принципом цієї архітектура буде розроблена серверна частина та web додаток для телеграмм боту.

2.8 Принципи SOLID

SOLID - це аббревіатура, яка представляє набір основних принципів проектування об'єктно-орієнтованого програмування. Ці принципи допомагають створювати гнучкі, розширювані та підтримувані програмні системи. Кожна літера SOLID відповідає окремому принципу [19]:

а) Single Responsibility Principle (Принцип єдиного обов'язку): Кожен клас повинен мати лише одну причину для зміни. Це означає, що кожен клас повинен виконувати лише одну конкретну відповідальність або завдання. Це полегшує розуміння, підтримку та зміну коду;

б) Open-Closed Principle (Принцип відкритості/закритості): Класи повинні бути відкритими для розширення, але закритими для змін. Це означає, що повинен бути забезпечений механізм розширення функціональності класу без зміни його вихідного коду. Це досягається за допомогою використання абстракцій, інтерфейсів та поліморфізму;

в) Liskov Substitution Principle (Принцип підстановки Лісков): Об'єкти підкласу повинні бути замінювані об'єктами базового класу без зміни коректності програми. Це означає, що класи-спадкоємці повинні відповідати

контракту, заданому базовим класом, та поводитись так само, як і базовий клас;

г) Interface Segregation Principle (Принцип розділення інтерфейсу): Клієнти не повинні залежати від інтерфейсів, які вони не використовують. Це означає, що інтерфейси повинні бути спеціалізованими та зорієнтованими на потреби конкретних клієнтів. Краще мати кілька малих спеціалізованих інтерфейсів, ніж один загальний;

д) Dependency Inversion Principle (Принцип інверсії залежності): Класи повинні залежати від абстракцій, а не від конкретних реалізацій. Це означає, що високорівневі модулі не повинні залежати від низькорівневих модулів, а обидва типи модулів повинні залежати від абстракцій. Абстракції повинні бути встановлені на рівні взаємодії між компонентами програми.

Ці принципи SOLID надають директиви для проектування програмного забезпечення, що покращують його структуру, модульність та розширюваність. Вони сприяють створенню коду, який легко змінюється, тестується та підтримується протягом тривалого часу.

3 РОЗДІЛ СИНТЕЗУ КОМП’ЮТЕРНОЇ СИСТЕМИ

3.1 Обґрунтування вибору сервера для телеграм-боту

У процесі розробки комп'ютерної системи складського обліку підприємства «АКС-ЮГ СИСТЕМА», важливим етапом є вибір оптимального сервера для розміщення та роботи Telegram-боту. Сервер відіграє ключову роль у забезпеченні стабільної, безперебійної роботи системи, а також у швидкому обміні даними між ботом і користувачами.

Для обґрунтованого вибору сервера враховувалися ключові вимоги до системи. Сервер має забезпечувати високу швидкість обробки даних та мінімальну затримку відповіді, що особливо важливо при одночасній взаємодії кількох користувачів із ботом. Висока надійність та доступність на рівні 99,9% є критичною умовою для забезпечення стабільної роботи системи. Також враховувалася можливість резервного копіювання даних для запобігання їх втраті у разі збоїв. Важливим критерієм стало питання масштабованості, адже у разі збільшення кількості користувачів або функціоналу система повинна мати здатність швидко розширювати ресурси. Крім того, сервер повинен відповідати критеріям безпеки, забезпечуючи надійний захист даних від несанкціонованого доступу.

Аналіз можливих варіантів серверів:

а) хмарні сервери (AWS, Google Cloud, Microsoft Azure). Хмарні сервери є одним із найбільш надійних і продуктивних варіантів для розміщення Telegram-боту. Вони забезпечують високу доступність завдяки розподіленій інфраструктурі та глобальним дата-центрам, що мінімізують простої системи. Основними перевагами хмарних серверів є гнучка масштабованість, можливість автоматичного резервного копіювання даних та висока продуктивність, що дозволяє обробляти велику кількість одночасних запитів. Крім того, провайдери хмарних рішень надають

високий рівень безпеки, включно з шифруванням даних і захистом від несанкціонованого доступу.

Однак, основним недоліком хмарних серверів є відносно висока вартість їх використання у довгостроковій перспективі, особливо при інтенсивному використанні ресурсів. Проте ці витрати компенсуються зниженням витрат на технічне обслуговування інфраструктури, оскільки підтримка повністю покладається на провайдера;

б) виділений фізичний сервер на базі підприємства. Виділений фізичний сервер передбачає встановлення та обслуговування апаратного обладнання безпосередньо на базі підприємства. Основною перевагою такого рішення є повний контроль над ресурсами сервера та можливість локального зберігання даних, що може бути важливим для підприємств із підвищеними вимогами до конфіденційності. Також відсутня залежність від зовнішніх провайдерів.

Проте виділені фізичні сервери мають значні недоліки. По-перше, це високі витрати на закупівлю обладнання, його встановлення та обслуговування. По-друге, підприємство потребує кваліфікованих фахівців для підтримки працездатності системи. Крім того, фізичний сервер є менш гнучким у питанні масштабування ресурсів та може стати вузьким місцем у разі зростання навантаження на систему;

в) віртуальний приватний сервер (VPS) у сторонніх провайдерів (DigitalOcean, Hetzner, Linode). Віртуальний приватний сервер (VPS) є компромісним варіантом між хмарними рішеннями та виділеними фізичними серверами. VPS надає доступ до виділеної частини обчислювальних ресурсів на фізичному сервері провайдера. Основними перевагами VPS є відносно низька вартість, гнучкість у налаштуванні сервера та легка масштабованість. Провайдери VPS також забезпечують інструменти для резервного копіювання даних і мають базові механізми безпеки.

Серед недоліків VPS слід відзначити залежність від провайдера та його технічної підтримки. У разі збоїв у роботі фізичного сервера або неполадок у мережі користувачі можуть зіткнутися з простоєм системи.

На основі аналізу переваг і недоліків було вирішено використовувати хмарні сервери, зокрема платформу AWS (Amazon Web Services). Основною причиною цього вибору є висока продуктивність, надійність та гнучкість хмарних серверів. AWS надає можливість автоматичного масштабування ресурсів у разі зростання навантаження, що є критичним для розвитку системи складського обліку.

3.1.1 Обґрунтування вибору хмарного сервера для телеграм-бота

Основною причиною вибору хмарного сервера для розміщення Telegram-боту є його висока продуктивність та гнучкість. Система складського обліку вимагає швидкої обробки запитів користувачів, а хмарна інфраструктура забезпечує мінімальні затримки завдяки глобально розподіленим дата-центрам. Це дозволяє підприємству обслуговувати велику кількість користувачів одночасно та без збоїв.

Хмарний сервер також гарантує надійність і доступність системи на рівні 99,9%, що є критично важливим для безперебійної роботи складського обліку. Завдяки автоматичному резервному копіюванню та можливостям відновлення даних, система буде захищена від втрат інформації у разі технічних збоїв або інших непередбачуваних обставин.

Ще однією перевагою є масштабованість. У разі збільшення кількості користувачів або додавання нових функціональних можливостей, ресурси хмарного сервера можна легко розширити. Це дозволяє системі зростати разом із потребами підприємства без необхідності в додатковому обладнанні чи перенесенні інфраструктури.

Хмарні платформи, такі як AWS, забезпечують високий рівень безпеки завдяки шифруванню даних та захисту від несанкціонованого

доступу. Це особливо важливо для системи складського обліку, де обробляються конфіденційні дані про товари, клієнтів і фінансові операції.

Переваги хмарних серверів:

а) висока продуктивність та швидкодія. Хмарні сервери забезпечують високу обчислювальну потужність та швидке опрацювання запитів користувачів завдяки використанню сучасного обладнання та технологій розподілених обчислень. Це особливо важливо для Telegram-боту, який обслуговує запити у реальному часі;

б) гнучка масштабованість. Хмарна інфраструктура дозволяє швидко збільшувати або зменшувати ресурси (процесорну потужність, оперативну пам'ять, обсяг сховища) відповідно до навантаження на систему. У разі зростання кількості користувачів або розширення функціоналу системи це дає можливість уникнути простоїв та збоїв;

в) надійність та доступність. Провідні хмарні провайдери, такі як Amazon Web Services (AWS), гарантують доступність сервісів на рівні 99,9% завдяки використанню розподілених дата-центрів і систем автоматичного моніторингу. Це забезпечує безперебійну роботу системи, що є критично важливим для складського обліку, де кожна хвилина простою може призвести до фінансових втрат;

г) резервування та відновлення даних. Хмарні платформи надають можливості для автоматичного резервного копіювання та швидкого відновлення даних. Це знижує ризик втрати інформації в разі технічних збоїв, помилок або несанкціонованих дій користувачів.

ґ) високий рівень безпеки. Хмарні сервери використовують сучасні механізми безпеки, зокрема:

1) Шифрування даних під час зберігання та передачі;

2) Автоматичний захист від DDoS-атак;

3) Багаторівневу аутентифікацію та контроль доступу.

Це особливо важливо для системи складського обліку, де

обробляються конфіденційні дані про товари, клієнтів і фінансові операції.

д) відсутність витрат на фізичне обладнання. Використання хмарних серверів дозволяє підприємству уникнути витрат на придбання, налаштування та обслуговування фізичних серверів. Усі технічні аспекти (охолодження, технічна підтримка, апаратне оновлення) забезпечуються провайдером;

е) глобальна доступність. Хмарні сервери забезпечують доступ до системи з будь–якої точки світу за умови наявності інтернет–з'єднання. Це дозволяє віддаленим працівникам та керівництву підприємства контролювати стан складу у реальному часі;

є) швидка інтеграція з іншими сервісами. Хмарні платформи, такі як AWS, пропонують широкий спектр інтеграцій із базами даних, аналітичними інструментами, системами штучного інтелекту та іншими хмарними сервісами, що спрощує розширення функціональності системи.

Недоліки хмарних серверів:

а) висока вартість. Незважаючи на те, що хмарні сервери дозволяють уникнути витрат на фізичне обладнання, їх обслуговування може бути дорогим при значному навантаженні або тривалому використанні. Для системи з великою кількістю запитів вартість може суттєво зрости через споживання обчислювальних ресурсів та обсяг збережених даних;

б) Залежність від інтернет-з'єднання. Для доступу до хмарного сервера необхідне стабільне інтернет-з'єднання. У разі проблем з мережею система може стати тимчасово недоступною. Це може бути критичним у випадку віддалених регіонів із нестабільним інтернетом;

в) Залежність від провайдера. Підприємство стає залежним від хмарного провайдера, його технічної підтримки та стабільності інфраструктури. У випадку технічних збоїв на стороні провайдера система може зазнати простою;

г) Ризики безпеки даних. Попри високий рівень безпеки, існує ризик потенційного витоку або несанкціонованого доступу до даних через людський фактор або уразливості. Для мінімізації цих ризиків необхідно впроваджувати додаткові заходи безпеки, зокрема двофакторну аутентифікацію, моніторинг доступу тощо;

ґ) Обмежена конфіденційність. Дані зберігаються на серверах стороннього постачальника, що може викликати занепокоєння з приводу конфіденційності. Для систем із чутливою інформацією необхідно ретельно обирати провайдера та використовувати додаткове шифрування даних.

3.1.2 Основні причини вибору хмарного сервера

Вибір хмарного сервера для реалізації комп'ютерної системи складського обліку підприємства «АКС-ЮГ СИСТЕМА» обумовлений його високою продуктивністю, гнучкою масштабованістю та надійністю. Завдяки можливості швидкого розгортання, автоматичного резервування даних та доступності з будь-якої точки світу, хмарний сервер забезпечує стабільну та безперебійну роботу Telegram-боту. Крім того, висока швидкість обробки запитів і сучасні механізми безпеки дозволяють ефективно обслуговувати користувачів та захищати інформацію від несанкціонованого доступу.

Хмарна інфраструктура є економічно вигідним рішенням, адже модель оплати «pay-as-you-go» дозволяє оптимізувати витрати на підтримку системи. У результаті підприємство отримує надійну та технологічно сучасну платформу, яка відповідає всім вимогам продуктивності, безпеки та гнучкості, необхідним для ефективного управління складським обліком.

3.1.3 Приклад обраного хмарного сервера

У процесі розробки комп'ютерної системи складського обліку підприємства «АКС-ЮГ СИСТЕМА» обрав хмарний сервер AWS EC2 (Elastic Compute Cloud) на платформі Amazon Web Services (AWS). Конкретним прикладом є інстанс t3.micro, який відповідає вимогам до

продуктивності, вартості та масштабованості для роботи Telegram-боту системи складського обліку.

Конфігурація обраного сервера включає 1 віртуальний процесор (vCPU), 1 ГБ оперативної пам’яті та до 30 ГБ дискового простору на основі Amazon Elastic Block Store (EBS). Як операційна система використовується Ubuntu 20.04 LTS, оскільки вона є стабільною, безпечною та популярною серед розробників. Для мінімізації затримок запитів сервер було обрано в регіоні EU (Frankfurt), що дозволяє забезпечити швидкий доступ користувачів до системи.

Основною перевагою AWS EC2 є те, що інстанс t3.micro підходить для малих і середніх проєктів завдяки своїй економічності. Зокрема, на початковому етапі використання він доступний безкоштовно протягом 12 місяців у межах програми AWS Free Tier, що є вагомим аргументом для студентського проєкту та малобюджетного підприємства.

AWS EC2 забезпечує високу продуктивність для роботи Telegram-боту, який відповідає за обробку запитів та збереження даних у системі. Зокрема, завдяки можливостям хмарної інфраструктури система здатна обробляти одночасні запити користувачів з мінімальною затримкою. У разі зростання навантаження інстанс можна легко масштабувати до більш продуктивних варіантів, таких як t3.small або t3.medium, що дозволяє забезпечити стабільну роботу системи при збільшенні кількості користувачів.

Надійність сервера є ще однією важливою перевагою AWS. Інфраструктура забезпечує доступність на рівні 99,99%, що мінімізує ризик простою системи складського обліку. Це критично важливо для підприємства, оскільки система повинна працювати безперебійно для оперативної обробки інформації про складські запаси.

Безпека даних на AWS EC2 реалізується на високому рівні завдяки вбудованим інструментам для моніторингу та захисту даних. Налаштування

Security Groups дозволяє контролювати доступ до сервера, а шифрування зберігає дані в безпеці. Це забезпечує конфіденційність інформації про складські операції підприємства та захищає від можливих загроз.

На сервері також легко розгорнути необхідні програмні компоненти. Для роботи з базою даних використовується PostgreSQL, який забезпечує ефективну роботу з великими обсягами даних. Сам Telegram-бот розроблено на мові програмування JavaScript із використанням середовища Node.js та бібліотеки *node-telegram-bot-api*, що дозволяє реалізувати необхідний функціонал для обліку товарів і обміну даними.

Таким чином, хмарний сервер AWS EC2 t3.micro є оптимальним вибором для реалізації Telegram-боту в системі складського обліку підприємства.

Характеристика	Значення
Тип сервера	AWS EC2 (Elastic Compute Cloud)
Інстанс	t3.micro
Процесор (vCPU)	1 віртуальний процесор (vCPU)
Оперативна пам'ять (RAM)	1 ГБ
Дисковий простір	До 30 ГБ (Amazon Elastic Block Store, EBS)
Операційна система	Ubuntu 20.04 LTS
Програмне середовище	Node.js + JavaScript
База даних	PostgreSQL
Регіон	EU (Frankfurt)
Пропускна здатність	5 ГБ/місяць у межах Free Tier
Доступність	99,99%
Безпека	Налаштування Security Groups, шифрування
Вартість	Безкоштовно до 12 місяців (AWS Free Tier)
Масштабованість	Легке масштабування до t3.small, t3.medium
Призначення	Хостинг Telegram-боту для складського обліку

Таблиця 3.1 Характеристики сервера AWS EC2

Таблиця містить основні характеристики обраного хмарного сервера AWS EC2 t3.micro, які відповідають вимогам системи складського обліку підприємства. Сервер забезпечує необхідну продуктивність завдяки 1 vCPU та 1 ГБ оперативної пам'яті, має гнучку масштабованість, стабільну

доступність на рівні 99,99%, а також надає можливість безкоштовного використання протягом 12 місяців у рамках програми AWS Free Tier.

Сервер працює на операційній системі Ubuntu 20.04 LTS, що є стабільним і безпечним рішенням для розгортання середовища Node.js із Telegram-ботом на JavaScript. База даних PostgreSQL використовується для збереження інформації про складські операції.

3.2 Обґрунтування вибору бази даних для збереження інформації про складські операції

У процесі розробки комп'ютерної системи складського обліку підприємства «АКС-ЮГ СИСТЕМА» важливим компонентом є вибір бази даних для зберігання інформації про складські операції. База даних забезпечує надійне збереження, обробку та доступ до даних, що є ключовим для ефективного функціонування системи.

На основі аналізу реляційних баз даних, серед доступних рішень було обрано PostgreSQL.

3.2.1 Переваги використання PostgreSQL

Це обґрунтовується наступними перевагами:

а) надійність та продуктивність. PostgreSQL є однією з найпопулярніших реляційних баз даних з відкритим вихідним кодом. Вона забезпечує високу продуктивність завдяки оптимізованому обробленню SQL-запитів і підтримує роботу з великими обсягами даних, що особливо важливо для складських операцій;

б) гнучкість і розширюваність. PostgreSQL підтримує складні запити, індекси, транзакції та забезпечує високу гнучкість при обробці структурованих даних. Система дозволяє створювати складні схеми баз даних, що є необхідним для ефективного обліку товарів;

в) масштабованість. PostgreSQL дозволяє масштабувати базу даних як вертикально (шляхом збільшення ресурсів сервера), так і горизонтально

(шляхом розподілу даних). Це дозволяє системі легко адаптуватися до зростання обсягів інформації та кількості користувачів;

г) підтримка безпеки. PostgreSQL надає вбудовані механізми для контролю доступу користувачів та захисту даних. Підтримується аутентифікація за допомогою паролів, а також шифрування даних на рівні з'єднання, що відповідає вимогам безпеки для корпоративних систем;

ґ) відкритий вихідний код та безкоштовне використання. PostgreSQL є безкоштовним рішенням з відкритим вихідним кодом, що робить його економічно вигідним варіантом для впровадження у невеликих та середніх підприємствах, таких як «АКС-ЮГ СИСТЕМА»;

д) сумісність із JavaScript. Завдяки своїй підтримці JSON і можливості інтеграції з середовищем Node.js, PostgreSQL є ідеальним рішенням для збереження та обробки даних у системі, де Telegram-бот розроблений на JavaScript.

3.2.2 Приклад обраної бази даних PostgreSQL

Для реалізації комп'ютерної системи складського обліку підприємства «АКС-ЮГ СИСТЕМА» обрано базу даних PostgreSQL як основне рішення для зберігання інформації про складські операції. PostgreSQL відповідає всім критеріям надійності, масштабованості та продуктивності, необхідним для ефективного управління даними у складській системі.

Характеристика	Опис
Назва	PostgreSQL
Тип бази даних	Реляційна база даних (RDBMS)
Ліцензія	Open-source (BSD ліцензія)
Версія	PostgreSQL 15 (останнє стабільне оновлення)
Механізм обробки запитів	SQL (Structured Query Language), підтримка JSON і JSONB для зберігання неструктурованих даних
Продуктивність	Висока продуктивність при виконанні складних SQL-запитів, індексація для швидкого пошуку
Безпека	Підтримка аутентифікації користувачів, шифрування даних, контроль доступу
Масштабованість	Вертикальна і горизонтальна масштабованість; підтримка реплікації
Інтеграція	Сумісність із JavaScript через Node.js і Sequelize як ORM
Сховище даних	Підтримка великих обсягів даних із можливістю кластеризації
Вартість	Безкоштовне використання (open-source), знижує витрати на розгортання
Інтерфейс управління	pgAdmin, DBeaver, CLI (Command Line Interface) для адміністрування бази

Таблиця 3.2. Характеристики PostgreSQL

PostgreSQL є потужним інструментом для обробки складних структур даних, що дозволяє зберігати інформацію про складські операції у вигляді структурованих таблиць. Завдяки підтримці JSON і JSONB, ця система дозволяє працювати як зі структурованими даними, так і з напівструктурованими форматами, що важливо для інтеграції з Telegram-ботом на JavaScript.

Ця база даних має гнучкий механізм реплікації та можливість масштабування, що дозволяє ефективно обробляти дані при збільшенні кількості товарообігу. Завдяки своїй відкритості та безкоштовності PostgreSQL є економічно вигідним рішенням для невеликих і середніх підприємств, таких як «АКС-ЮГ СИСТЕМА».

3.3 Технічні вимоги до пристроїв операторів складу для використання телеграм-бота

Для забезпечення ефективної роботи комп'ютерної системи складського обліку «АКС-ЮГ СИСТЕМА» з використанням Telegram-бота необхідно, щоб пристрої операторів складу відповідали сучасним апаратним і програмним вимогам. Оскільки Telegram-бот виконує функцію оперативного обміну інформацією та здійснення операцій у реальному часі, технічні характеристики обладнання відіграють важливу роль.

3.3.1 Вимоги до операційної системи

Першочергово важливо забезпечити сумісність операційної системи пристроїв із додатком Telegram. Для мобільних пристроїв оптимальними є операційні системи Android версії 8.0 і вище або iOS версії 12.0 і новіших. Це дозволяє гарантувати стабільну роботу додатка з усіма актуальними функціями, а також безпеку та оновлення. Якщо оператори складу працюють з комп'ютерами чи ноутбуками, необхідна підтримка сучасних настільних операційних систем, таких як Windows 10 або новіші версії та macOS Catalina 10.15 або пізніші версії.

3.3.2 Вимоги до апаратних характеристик

Процесор також відіграє значну роль у забезпеченні стабільної роботи Telegram-бота. Мобільні пристрої повинні мати щонайменше 4-ядерний процесор із тактовою частотою 1.6 ГГц або вище, що забезпечить швидке завантаження даних та відповіді на запити без затримок. Для комп'ютерів та ноутбуків достатнім буде процесор рівня Intel Core i3 або AMD Ryzen 3 і вище, що дає змогу безперебійно працювати із веб-версією Telegram або десктопним клієнтом.

Оперативна пам’ять є ключовим компонентом для швидкої обробки даних і стабільного функціонування пристрою. Для смартфонів і планшетів рекомендованим обсягом є 2 ГБ, проте для більшої надійності та відсутності збоїв краще використовувати пристрої з 4 ГБ оперативної пам’яті. У

випадку ПК чи ноутбуків мінімальним значенням є 4 ГБ, а оптимальним — 8 ГБ, що забезпечить роботу додатків і браузерів одночасно.

3.3.3 Вимоги до інтернет-з’єднання

Для роботи Telegram-бота критично важливим є стабільне Інтернет-з’єднання. Мобільні пристрої повинні підтримувати роботу у мережах 3G, 4G або 5G, оскільки складські оператори часто працюють у динамічному середовищі, де доступ до Wi-Fi може бути обмежений. У випадку стаціонарної роботи на ПК чи ноутбуках необхідне Wi-Fi або Ethernet-з’єднання зі швидкістю не менше 5 Мбіт/с, що дозволить забезпечити миттєву передачу запитів та отримання відповідей від сервера.

3.3.4 Вимоги до оператора складу

Для зручності роботи операторів важливо, щоб пристрої мали достатній розмір та якість дисплея. На мобільних пристроях мінімальний розмір екрана повинен становити 5 дюймів з роздільною здатністю 720p (1280x720), що дозволить коректно відображати текстові та графічні дані. Комп’ютери та ноутбуки мають бути оснащені моніторами з діагоналлю 14 дюймів або більше і роздільною здатністю 1080p (1920x1080), що забезпечить чіткість інтерфейсу та зручність при виконанні складських операцій.

Важливим фактором є достатній обсяг пам'яті для зберігання даних. Мобільні пристрої повинні мати не менше 16 ГБ вільного місця, щоб зберігати кеш даних Telegram і робочі файли. Для комп’ютерів та ноутбуків мінімальним є обсяг 128 ГБ на накопичувачах типу SSD або HDD, щоб мати змогу зберігати тимчасові файли, журнали роботи системи та інші необхідні дані.

Зважаючи на те, що деякі оператори складу можуть працювати з веб-версією Telegram, пристрої повинні підтримувати сучасні веб-браузери, такі як Google Chrome, Mozilla Firefox чи Microsoft Edge, з останніми

оновленнями безпеки. Це забезпечить швидке завантаження інтерфейсу та повну функціональність бота.

3.4 Функціональна схема обміну даними

Функціональна схема обміну даними описує взаємодію між основними компонентами системи складського обліку «АКС-ЮГ СИСТЕМА» з використанням Telegram-бота. У процесі роботи система забезпечує передачу, обробку та збереження даних між клієнтськими пристроями, сервером та базою даних, що дозволяє здійснювати всі операції у режимі реального часу.

Початковий етап взаємодії розпочинається з ініціативи користувача, тобто оператора складу. Оператор використовує мобільний або стаціонарний пристрій, на якому встановлений додаток Telegram, щоб надіслати команду або запит до Telegram-бота. Наприклад, це може бути команда на перевірку залишків товарів на складі, додавання нового товару або створення звіту про відвантаження. Запит оформлюється у простій текстовій або командній формі, що робить роботу з ботом максимально зручною.

Після отримання запиту Telegram-бот виступає як посередник між користувачем та серверною частиною системи. Бот обробляє надіслану команду та перетворює її у формат, який розпізнає серверна частина. Telegram-бот направляє запит на сервер, де відбувається його подальша обробка. У цьому етапі важливу роль відіграє надійність мережевого з'єднання, щоб мінімізувати затримки під час передачі даних.

Хмарний сервер AWS приймає запит від Telegram-бота та виконує необхідну обробку. Залежно від характеру запиту, сервер може звертатися до бізнес-логіки програми для виконання операцій, таких як обчислення, перевірка умов або форматування даних. Якщо для виконання команди потрібна взаємодія з базою даних, сервер ініціює запит до системи управління базами даних PostgreSQL.

База даних PostgreSQL виступає центральним сховищем для всіх даних, пов'язаних зі складським обліком. Серед них – інформація про товари, операції надходження та відвантаження, дані про кількість товарів, а також логи активності користувачів. У разі запиту, наприклад, на перевірку залишків товару сервер виконує SQL-запит до бази даних, отримує необхідну інформацію та передає її назад на сервер для подальшої обробки.

Після отримання відповіді з бази даних сервер формує результат, який повертається до Telegram-бота. Відповідь надійно структурована та адаптована для зручного сприйняття користувачем. Наприклад, інформація про залишки товарів на складі може бути представлена у вигляді таблиці чи короткого текстового повідомлення з переліком найменувань і кількості товарів.

На завершальному етапі Telegram-бот отримує відповідь від сервера та відправляє її оператору складу на клієнтський пристрій. Оператор, зі свого боку, отримує необхідну інформацію у додатку Telegram та може відразу використовувати її для подальшої роботи. Цикл обміну даними завершується, і система готова до прийняття нових запитів від користувача.

Завдяки такій функціональній схемі обміну даними досягається швидка та надійна комунікація між усіма елементами системи. Telegram-бот виступає зручним інтерфейсом для взаємодії з користувачами, сервер AWS забезпечує високу продуктивність та надійність обробки даних, а база даних PostgreSQL гарантує цілісність і доступність інформації. Усі ці компоненти працюють у єдиній зв'язці, забезпечуючи безперебійну роботу системи складського обліку у режимі реального часу.

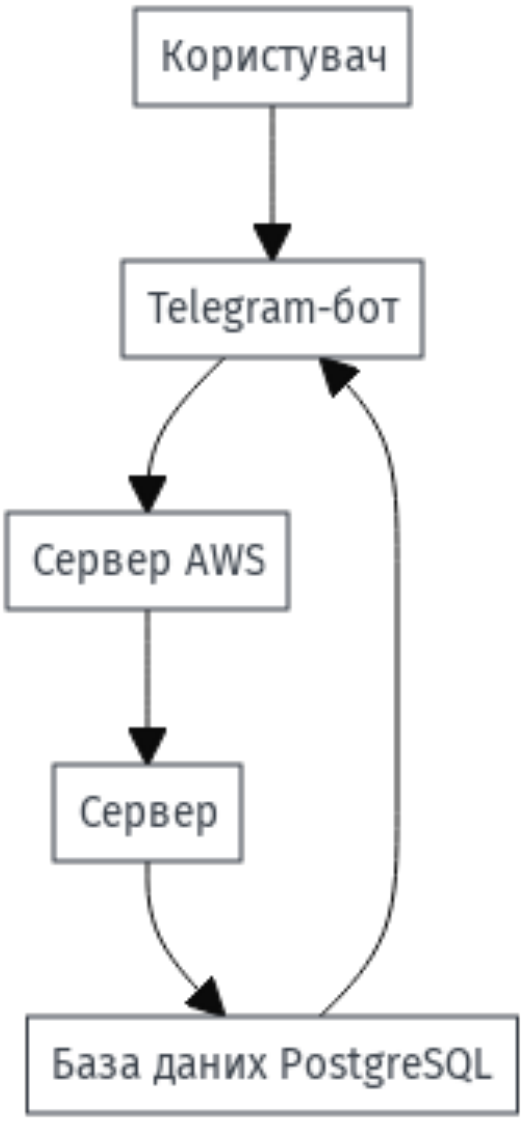

Рисунок 3.1 – Схема відображення основних етапів передачі даних

3.5 Висновки синтезу комп’ютерної системи

У розділі було розроблено ключові аспекти синтезу комп'ютерної системи складського обліку підприємства «АКС-ЮГ СИСТЕМА» з використанням Telegram-бота. Обґрунтовано вибір хмарного сервера AWS як оптимального рішення для забезпечення стабільної, безперебійної та масштабованої роботи системи, що дозволяє оперативно обробляти запити користувачів та захищати дані від несанкціонованого доступу.

Також визначено технічні вимоги до пристроїв операторів складу, які мають гарантувати швидкодію, зручність у використанні та підтримку необхідного програмного забезпечення. Розроблено функціональну схему обміну даними між компонентами системи, яка демонструє логіку взаємодії між Telegram-ботом, сервером AWS та базою даних PostgreSQL. Це забезпечує ефективну передачу даних, високу продуктивність та надійність роботи системи.

4 ПРОЄКТУВАННЯ БАЗИ ДАНИХ, РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ І СТВОРЕННЯ WEB ДОДАТКУ ДЛЯ ТЕЛЕГРАМ-БОТА

4.1 Функціональне призначення програмного забезпечення

Розроблене програмне забезпечення призначене для автоматизації процесів складського обліку та забезпечення зручної взаємодії з користувачами через Telegram-бот. Основними функціональними можливостями розробленого ПЗ є:

Управління замовленнями:

а) можливість створення замовлень через Telegram-бот;

б) відстеження стану замовлень користувачами;

в) формування інформації про товари, що входять до складу замовлень.

Робота з постачаннями:

а) внесення даних про постачання товарів.

б) збереження інформації про вартість поставки, валюту та кількість поставлених товарів.

Облік товарів:

а) додавання нових товарів до бази даних, їх редагування та видалення.

б) збереження ключових характеристик товарів, таких як назва, опис, ціна, кількість, артикул і виробник.

Таким чином, функціональне призначення програмного забезпечення спрямоване на підвищення ефективності облікових процесів, спрощення взаємодії з користувачами та надання актуальної інформації у зручному форматі.

4.2 Технічні характеристики програмного забезпечення

Розроблене програмне забезпечення є сучасним рішенням для автоматизації складського обліку, що включає як серверну, так і клієнтську частини. Основні технічні характеристики ПЗ описані нижче.

Серверна частина була розроблена за допомогою наступних інструментів:

Мова програмування – TypeScript, більш детальна інформація про обрану мову програмування наведена в пункті 2.4.

Фреймворк для серверної частини – NestJS більш детальна інформація про обраний фреймворк наведена в пункті 2.4.

СУБД – PostgreSQL більш детальна інформація про обрану СУБД наведена в пункті 2.1.

4.2.1 Постановка завдання на розробку програмного забезпечення

Метою розробки програмного забезпечення є серверної та клієнтської частини. Програмне забезпечення буде включати в себе такі можливості: створення, оновлення та перегляд товарів, перегляд та створення замовлень та поставок.

Для обробки асинхронних запитів і взаємодії з базою даних використовується асинхронне програмування, яке забезпечує можливість одночасного виконання кількох операцій.

4.2.2 Опис алгоритму роботи програмного забезпечення

Алгортитм роботи програмного забезпечення виглядає наступним чином.

Телеграм бот приймає запит запит від користувача та передає запит на клієнтську частину. Після чого клієнтська частина передає сформаваний запит на серверну частину де його обробляє controller, потім controller передає інформацію з запиту в service в якому викноується бізнес логіка після чого service передає запит на необхідну інформацію з бази даних до repository далі вся зібрана інформація потрапляє до controller звідки йде на

клієнтську частину та передає результат в телеграм бот який відмальвоє результат.

Рисунок 4.1 Схема алгоритму роботи програмного забезпечення

4.2.3 Опис організації вхідних та вихідних даних

В якості вхідних даних для клієнтської частини виступає інформація в форматі JSON яка далі передеється на серверну частину.

Вихідні дані також представляють собою інформацію в форматі JSON, але в тому стилі який буде зрозумілий клієнтській частині.

4.3 Опис розробленого програмного забезпечення

4.3.1 Опис та проектування бази даних

Розглянемо предметну область систем для складського обліку.

Система складського обліку має вести облік для наступних даних:

а) Користувачів (№ користувача, телеграм id користувача, ім’я користувача);

б) Товарів (№ товару, назва товару, опис товару, ціна, кількість, № виробника, артикль);

в) Виробників (№ виробника, ім’я виробника);

г) Замовлень (№ заказу, статус, ціна за замовлення);

д) Товарів в замовленнях (№ товару в замовлені, № номер замовлення, № номер товару, кількість, ціна);

е) Поставок (№ номер поставки, ціна поставки, валюта);

є) Товарів в поставках (№ поставки, № товару, № поставки, кількість, ціна).

Після розгляду предметної області можна вивести наступні сутності для бази даних:

а) Користувачі (user);

б) Товари (product);

в) Виробники (manufacturer);

г) Замовлення (order);

д) Товари в замовлені (orderItem);

е) Поставки (delivery);

є) Товари в поставці (deliveryItem).

База даних була спроектована на основі визначених сутностей із дотриманням нормальних форм і встановленням зв’язків між таблицями.

Процес нормалізації бази даних є методологією, яка спрямована на впорядкування даних у таблицях, щоб зменшити надмірність, залежності між атрибутами та забезпечити структурну цілісність даних. Нормалізація дозволяє уникати аномалій під час додавання, оновлення чи видалення записів, спрощує виконання запитів та оптимізує використання ресурсів зберігання. У цьому процесі застосовуються нормальні форми, які визначають правила побудови таблиць і зв’язків між ними [17].

У проектованій базі даних таблиці було приведено до основних нормальних форм:

а) перша нормальна форма (1НФ**):** усі атрибути таблиць є атомарними, тобто поділеними на найменші логічні одиниці. Наприклад, поля для повного імені (ПІБ) лікарів і пацієнтів розділені на окремі атрибути для прізвища, імені та по батькові. Кожен атрибут містить лише одне значення.

б) друга нормальна форма (2НФ)**:** усі неключові атрибути функціонально залежать лише від повного первинного ключа. Для кожної таблиці введено первинний ключ (наприклад, Id), який

забезпечує однозначну ідентифікацію записів. Частково залежні атрибути були винесені в окремі таблиці.

в) третя нормальна форма (3НФ): кожен неключовий атрибут залежить виключно від первинного ключа, а не від інших неключових атрибутів. Транзитивні залежності були виключені.

г) перевірки унікальності та коректності даних реалізовані на рівні web-api, тому в базі даних вони не описувалися.

При проектуванні бази даних також було визначено наступні типи зв’язків між таблицями [17]:

а) один-до-одного (One-to-One, 1:1): кожен запис однієї таблиці пов'язаний лише з одним записом іншої. Наприклад, кожен пацієнт у медичній системі має одну медичну картку, і навпаки, кожна картка належить лише одному пацієнту.

б) один-до-багатьох (One-to-Many, 1:N): один запис однієї таблиці може бути пов'язаний із кількома записами іншої. Наприклад, лікар може мати одну спеціальність, але одна спеціальність може бути прив’язана до багатьох лікарів.

Такі зв’язки забезпечують коректне зв’язування даних між таблицями, що є основою для їх ефективної обробки[17].

На рисунку 4.2 можна побачити ER діаграмму бази даних (див. рис. 4.2).

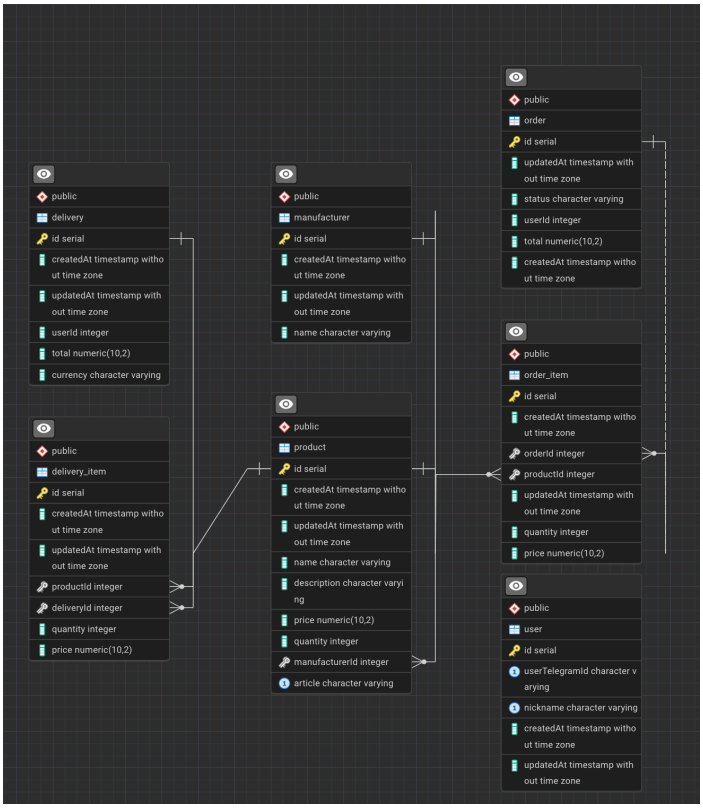

Рисунок 4.2 – ER діаграма бази даних

4.3.2 Опис серверної частини

4.3.2.1 Архітектура серверної частини

Серверна частина була побудована за принципом трьохрівневої архітектури, яка складається з:

а) рівня обробки вхідних даних: Відповідає за прийом та валідацію запитів від клієнтів;

б) рівня бізнес-логіки: Містить основну логіку програми та правила обробки даних;

в) рівня доступу до даних: Забезпечує взаємодію з базою даних чи іншими джерелами даних.

Для зв'язку між цими рівнями використовується механізм впровадження залежностей (Dependency Injection), що дозволяє

зменшити зв'язність між компонентами та підвищити гнучкість системи [15].

4.3.2.2 Рівень обробки вхідної інформації

В NestJS для обробки вхідних даних використовуються контролери. Вони забезпечують обробку запитів від клієнта та формування відповідей. Принцип роботи контролера показано на рисунку 4.3 (див. рис. 4.3).

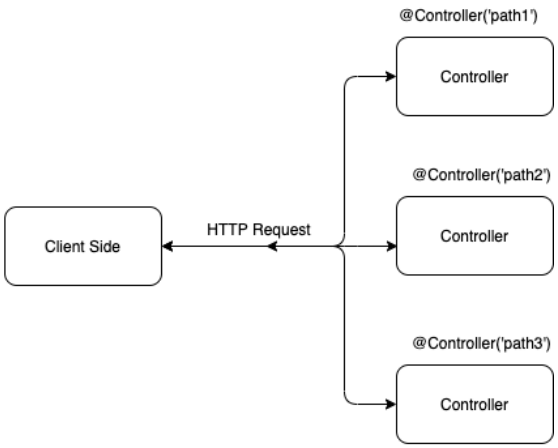

Рисунок 4.3 – принцип роботи контролера

```
@UseGuards(UserExistsGuard)
@Controller('delivery')
export class DeliveryController {
  constructor(private readonly deliveryService: DeliveryService) {}

  @Post()
  async createDelivery(@Body() createDeliveryDto: CreateDeliveryDto) {
    return this.deliveryService.createDelivery(createDeliveryDto);
  }

  @Get()
  async getAllDelivery() {
    return this.deliveryService.getAllDelivery();
  }

  @Get(':id')
  async getDeliveryById(@Param('id') id: string) {
    return this.deliveryService.getDeliveryById(+id);
  }

  @Patch(':id')
  async updateDelivery(
    @Param('id') id: string,
    @Body() updateDeliveryDto: UpdateDeliveryDto,
  ) {
    return this.deliveryService.updateDelivery(+id, updateDeliveryDto);
  }

  @Delete(':id')
  async deleteDelivery(@Param('id') id: string) {
    return this.deliveryService.deleteDelivery(+id);
  }
}
```

Рисунок 4.4 – контролер для поставок (delivery)

Основне завдання контролера - обробляти конкретні запити від додатку. Маршрутизація визначає, який саме контролер обробляє конкретний запит. Зазвичай контролер обробляє декілька маршрутів, кожен з яких може виконувати різні функції. Отримавши запит, контролер аналізує його вміст і передає дані відповідному сервісу для подальшої обробки [10].

В данному прикладі контролер delivery обробляє наступні endpoint`и:

а) створення нової поставки (delivery);

б) отримання всіх поставок;

в) отримання однієї поставки за її id;

г) видалення поставки за її id;

д) оновлення поставки за її id.

4.3.2.3 Рівень обробки бізнес-логіки

Бізнес-логіка - це частина програмного забезпечення, яка визначає правила та процедури, що керують бізнес-процесами організації або додатку. Вона відображає конкретні бізнес-вимоги та правила, які визначають, як додаток повинен взаємодіяти з даними та виконувати певні дії [13].

Вся бізнес-логіка зазвичай обробляється в сервісах.

Ключові особливості сервісу:

а) бізнес-логіка. Сервіси виконують бізнес-логіку додатку, включаючи обробку даних, взаємодію з базами даних, зовнішніми сервісами або іншими джерелами даних. Вони відповідають за виконання певних функцій і обробку бізнес-правил;

б) інкапсульована логіка. Сервіси дозволяють інкапсулювати логіку додатку в окремі модулі. Це допомагає забезпечити чистоту коду і полегшує тестування та підтримку. Сервіси також роблять ваш код більш масштабованим і розширюваним, оскільки ви можете змінювати логіку окремо від інших компонентів;

в) залежності та ін'єкції залежностей. Сервіси можуть мати залежності від інших сервісів або компонентів. NestJS надає механізм ін'єкції залежностей, що дозволяє легко керувати залежностями та перевизначати їх під час тестування. Це дозволяє досягти більшої деталізації та полегшує повторне використання коду;

г) відокремлення від контролера. Сервіси використовуються контролерами для обробки запитів і передачі даних між різними компонентами вашого додатку. Сервіси дозволяють декомпонувати контролер так, щоб він міг зосередитися на обробці HTTP-запитів і передати всю бізнес-логіку сервісам.

```
@Injectable()
export class DeliveryService {
  constructor(
    @InjectRepository(Delivery)
    private readonly deliveryRepository: Repository<Delivery>,
    @InjectRepository(Product)
    private readonly productRepository: Repository<Product>,
    @InjectRepository(DeliveryItem)
    private readonly deliveryItemRepository: Repository<DeliveryItem>,
    @InjectRepository(User)
    private readonly userRepository: Repository<User>,
  ) {}

  async createDelivery(createDeliveryDto: CreateDeliveryDto) {
    const { userId, currency, deliveryItems } = createDeliveryDto;

    const user = await this.userRepository.findOne({
      where: { id: userId },
    });

    let deliveryTotal = 0;

    if (!user) {
      throw new BadRequestException({
        message: `User with id ${userId} is not found`,
      });
    }

    const delivery = await this.deliveryRepository.save({
      userId,
      currency,
    });

    const deliveryItemsPromises = deliveryItems.map(async (deliveryItem) => {
      const deliveryItemPrice = deliveryItem.price * deliveryItem.quantity;

      deliveryTotal += deliveryItemPrice;

      return this.deliveryItemRepository.save({
        ...deliveryItem,
        deliveryId: delivery.id,
        price: deliveryItem.price,
      });
    });
    await Promise.all([deliveryItemsPromises]);
    await this.deliveryRepository.update(delivery.id, {
      total: deliveryTotal,
    });
    return this.getDeliveryById(delivery.id);
  }
```

Рисунок 4.5 – сервіс для роботи з поставками (delivery)

Цей сервіс слугує для роботи поставок (delivery). Він виконує наступні операції:

а) створення поставки;

б) отримання всіх поставок;

в) отримання однієї поставки за її id;

г) видалення поставки за її id;

д) оновлення поставки за її id.

Цими операціями він реалізує роботу бізнес логіки.

4.3.2.4 Рівень доступу до інформації з бази даних

Для доступу до даних використовується ORM (Object-Relational Mapping), зокрема TypeORM. Це один з найпоширеніших ORM для Node.js, який підтримує різноманітні бази даних, такі як PostgreSQL, MySQL, SQLite та інші. TypeORM дозволяє визначати моделі даних, створювати міграції, виконувати запити до бази даних і працювати з реляційними об'єктами.

Моделі даних описуються за допомогою Entity. Entity — це клас, що представляє модель даних або таблицю в базі даних. Кожен об'єкт типу Entity відповідає конкретному запису в таблиці і містить поля, що відповідають колонкам цієї таблиці [18]. Entity використовується для опису структури даних та взаємодії з базою даних через TypeORM.

```
@Entity()
export class Delivery {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  userId: number;

  @CreateDateColumn()
  createdAt: Date;

  @Column({ type: 'decimal', precision: 10, scale: 2, nullable: true })
  total: number;

  @Column({ enum: CurrencyEnum })
  currency: CurrencyEnum;

  @UpdateDateColumn()
  updatedAt: Date;

  @OneToMany(() => DeliveryItem, (deliveryItem) => deliveryItem.delivery)
  deliveryItems: DeliveryItem[];
}
```

Рисунок 4.6 – модель для поставок (delivery)

Для взаємодії з базою даних в NestJS використовується репозиторій (Repository).

Repository виступає як посередник між сервісами додатку і базою даних, і надає зручний спосіб управління даними.

Основні функції та призначення Repository в NestJS [18]:

а) запити до бази даних. Repository забезпечує методи для виконання запитів до бази даних. Це можуть бути методи для зчитування одного або кількох записів, створення нового запису, оновлення існуючого запису або видалення запису з бази даних. Репозиторій надає зручний інтерфейс для виконання цих операцій без прямого використання SQL або драйверів бази даних;

б) управління сутностями. Repository забезпечує методи для збереження, оновлення та видалення сутностей. Він дозволяє легко створювати, зчитувати та модифікувати дані в базі даних за допомогою об'єктів Entity. Repository допомагає забезпечити цілісність даних та зв'язків між таблицями;

в) розширення функціональності. Репозиторій може містити додаткові методи для виконання специфічних операцій з даними, які не покриваються стандартними методами. Наприклад, ви можете додати метод для фільтрації, сортування або пошуку даних за певними критеріями;

г) транзакційна підтримка. Репозиторій може підтримувати операції в межах транзакцій. Це дозволяє групувати декілька операцій в одну транзакцію, яка виконується атомарно, тобто або всі операції успішно.

```
@InjectRepository(Delivery)
private readonly deliveryRepository: Repository<Delivery>,
```

Рисунок 4.7 – приклад репозиторію (repository) для поставок (delivery).

4.3.3 Опис клієнтської частини

4.3.3.1 Архітектура клієнтської частини

Web додаток був побудований згідно Container/Component архітектури.

Використання патерну Container/Component дає змогу відокремити логіку функціонування додатка від логіки формування його візуального представлення. Це дає змогу поліпшити структуру додатка, розділити відповідальність за виконання різних завдань між різними компонентами [24].

4.3.3.2 Рівень компонентів

Компоненти — це незалежні фрагменти коду, які можна багаторазово використовувати. Компоненти дозволяють розділити інтерфейс користувача на незалежні частини, придатні до повторного використання, і сприймати їх як такі, що функціонують окремо один від одного [25].

```
const ActionDialog = ({
  open,
  onClose,
  onConfirm,
  title,
  description,
  confirmButtonText,
  cancelButtonText,
}) => {
  const {t} = useTranslation();

  return (
    <Dialog open={open} onClose={onClose}>
      <DialogTitle>{t(title)}</DialogTitle>
      <DialogContent>
        <DialogContentText>{t(description)}</DialogContentText>
      </DialogContent>
      <DialogActions>
        <Button onClick={onClose} color="primary">
          {t(cancelButtonText)}
        </Button>
        <Button onClick={onConfirm} color="secondary" variant="contained">
          {t(confirmButtonText)}
        </Button>
      </DialogActions>
    </Dialog>
  );
};

ActionDialog.propTypes = {
  open: PropTypes.bool.isRequired,
  onClose: PropTypes.func.isRequired,
  onConfirm: PropTypes.func.isRequired,
  title: PropTypes.string.isRequired,
  description: PropTypes.string.isRequired,
  confirmButtonText: PropTypes.string.isRequired,
  cancelButtonText: PropTypes.string.isRequired,
};

export default ActionDialog;
```

Рисунок 4.8 – Компонента Dialog

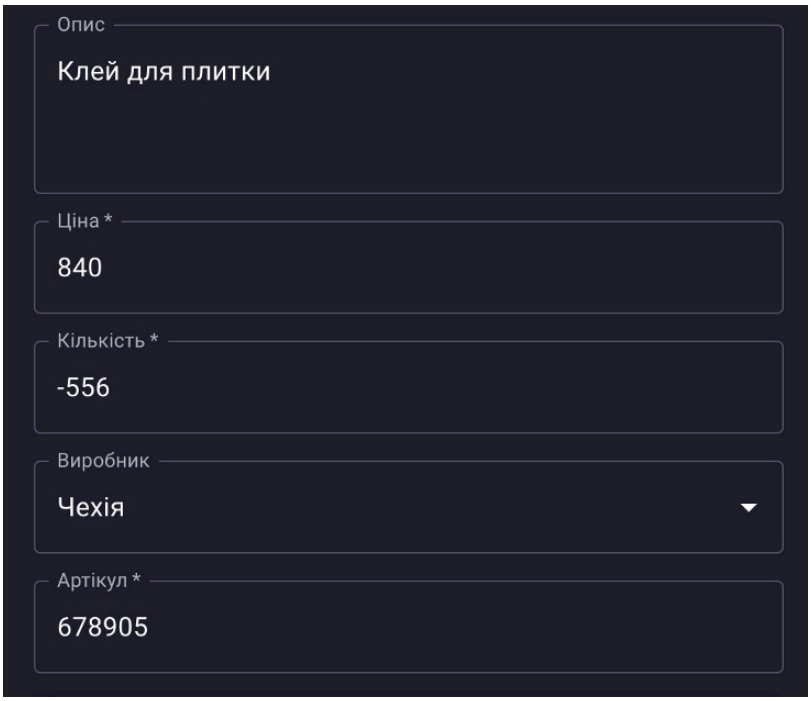

Рисунок 4.9 – Компонента Dialog в web додатку

4.3.3.3 Рівень контейнерів

Container відповідає за [26]:

а) логіку та управління станом;

б) взаємодію з зовнішніми джерелами даних;

в) передачу даних та функцій обробки в презентаційні компоненти.

Основні характеристики Container:

а) не відповідає за відображення UI напряму;

б) включає логіку бізнесу;

в) має доступ до стану (state).

```
const CreateDelivery = () => {
  const navigate = useNavigate();
  const handleSubmit = async (e, values) => {
    e.preventDefault();

    const {deliveryItems, currency} = values
    const filteredProducts = deliveryItems.filter(p => p.product && p.quantity)
    if (!filteredProducts?.length) {
      toast.error('Заповніть інформацію про товари');
      return;
    }

    if (!currency) {
      toast.error('Заповніть поле "Валюта"');
      return;
    }

    try {
      await axiosProvider.post(deliveriesResource,
        {
          deliveryItems: filteredProducts.map(item => ({
            productId: item.product.id,
            price: +item.product.price,
            quantity: +item.quantity,
          })),
          currency: currency.id,
          userId: getUserTelegramId(),
        })
      navigate(`/${deliveriesResource}`);
    } catch (e) {
      toast.error(e.response?.data?.message || e.message);
    }
  }

  return (
    <DeliveryForm
      handleSubmit={handleSubmit}
      values={{
        deliveryItems: [{product: null, quantity: ''}]
      }}
      actionTranslate="actions.create"
      titleTranslate="deliveries.form.title"
    />
  );
};

export default CreateDelivery;
```

Рисунок 4.9 – Контейнер Delivery для створення поставок

4.3.4 Функціональне призначення програмного забезпечення

Функціональне призначення програмного забезпечення полягає в автоматизації процесів збору, обробки та зберігання даних, що забезпечує ефективну взаємодію між користувачем і системою. Програма розроблена для вирішення конкретних задач, пов’язаних із оптимізацією бізнес-процесів, управлінням даними та їх аналізом. Її основною метою є створення зручного інтерфейсу для користувачів, який дозволяє виконувати всі необхідні операції швидко та з мінімальними витратами ресурсів.

Система має забезпечувати надійну роботу з базою даних, включаючи обробку запитів. Особливу увагу приділено масштабованості та гнучкості програмного забезпечення, що дозволяє легко адаптувати його до змін у бізнес-вимогах або збільшення обсягу даних. Завдяки впровадженню сучасних технологій, забезпечується висока продуктивність і стабільність роботи системи, що є ключовими факторами для задоволення потреб користувачів.

Програмне забезпечення також покликане інтегруватися з іншими системами, використовуючи стандартизовані інтерфейси для обміну даними, що дозволяє розширювати його функціонал і підвищувати ефективність роботи організації.

4.4 Опис та демонстраційні матеріали роботи функціонування програманого забезпечення

У цьому розділі представлено детальний опис роботи програмного забезпечення, а також демонстраційні матеріали, які ілюструють його функціонування. Розкрито принципи взаємодії користувача із системою, описано основні сценарії використання, включаючи виконання ключових функцій.

Також розглянуто інтерфейси користувача, їх зручність та відповідність поставленим завданням. Наведено приклади роботи програми, зокрема скріншоти інтерфейсу і ключові результати виконання

операцій. Показано, як розроблене рішення вирішує поставлені задачі, зокрема в контексті автоматизації процесів, які раніше виконувались вручну.

4.4.1 Функціонування програмного забезпечення зі сторони користувача

Основна взаємодія користувача з програмним забезпеченням відбувається через графічний інтерфейс.

Далі розглянемо використання основного функціоналу програмного забезпечення через графічний інтерфейс.

4.4.1.1 Робота з товарами

Програмне забезпечення надає користувачу змогу взаємодіяти з товарами, а саме: пошук товару за артикулом товару та за назвою товару, видалення товару, оновлення товару, створення товару.

Далі будуть наведені скріншоти роботи графічного інтерфейсу з вище перерахованим функціоналом.

Функціонал перегляду товарів.

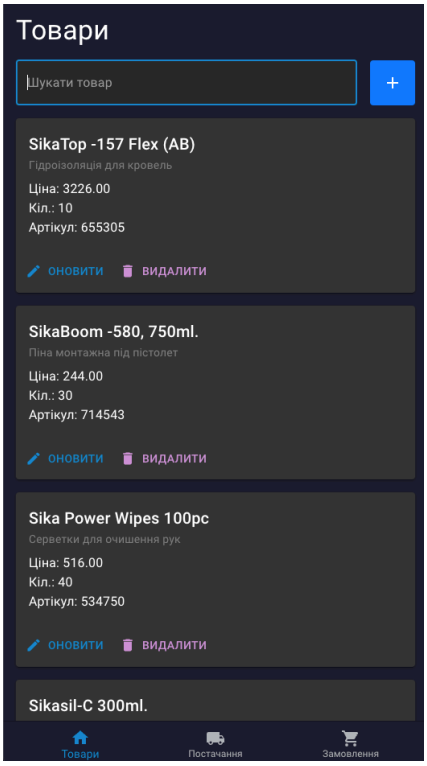

Рисунок 4.10 – сторінка товарів з списком товарі

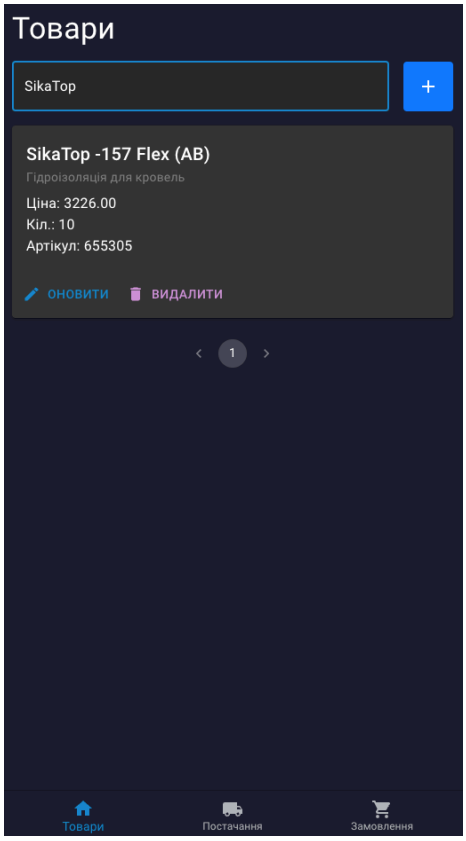

Рисунок 4.11 - Пошук товару за назвою

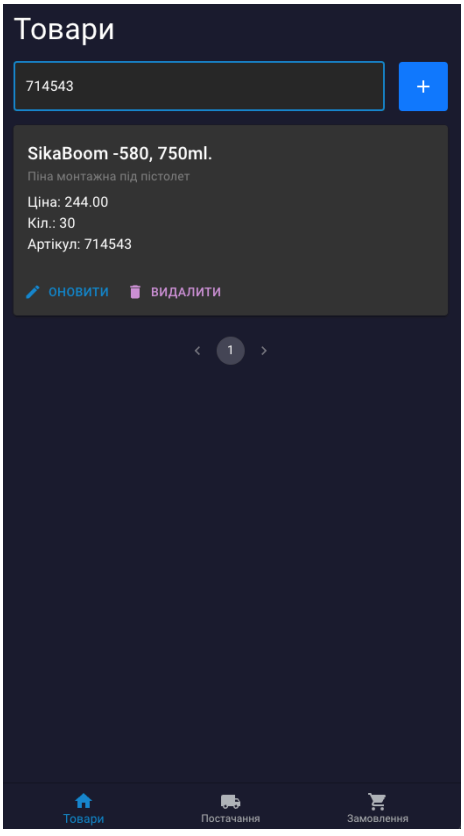

Рисунок 4.12 - Пошук товару за артикулом

Функціонал створення товару.

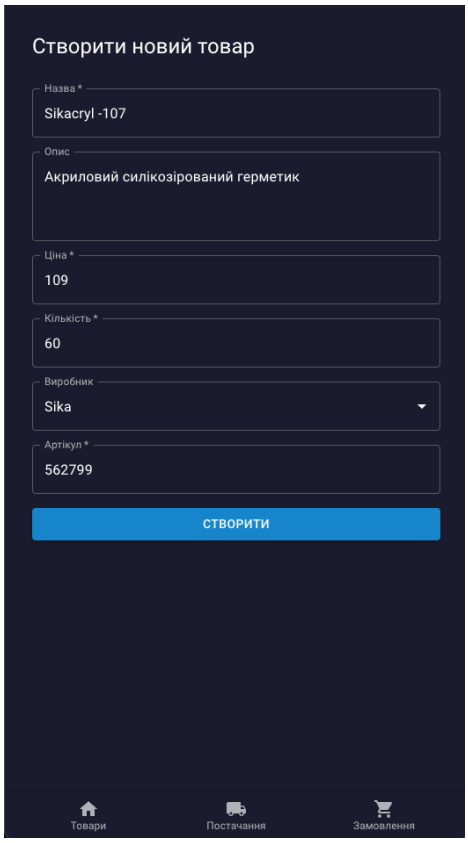

Рисунок 4.13- Форма створення товару

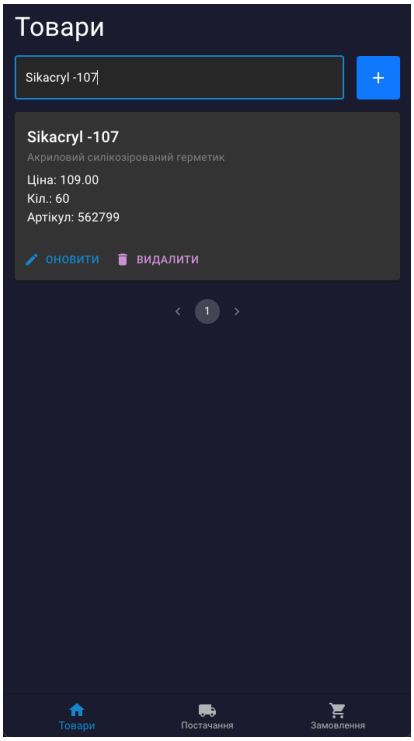

Рисунок 4.14 – Створений товар у списку товарів

Функціонал видалення товару

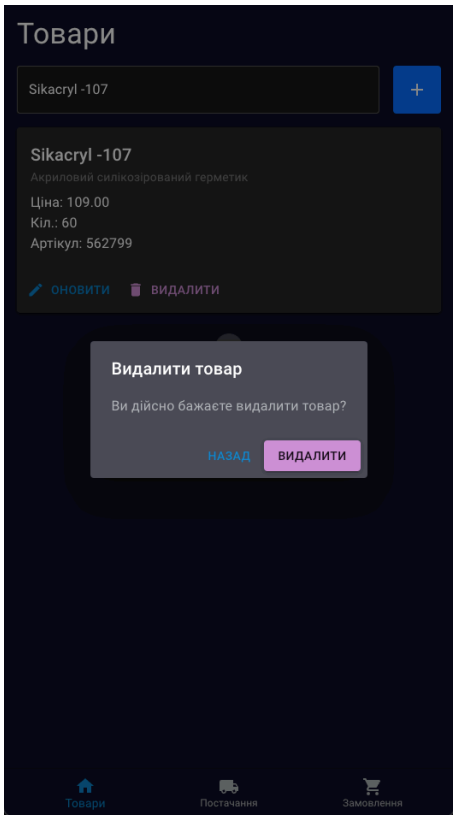

Рисунок 4.15 – Вікно для видалення товару

Після видалення товару його неможливо знайти через пошук.

Рисунок 4.16 – пошук видаленого товару

Функціонал оновлення товару.

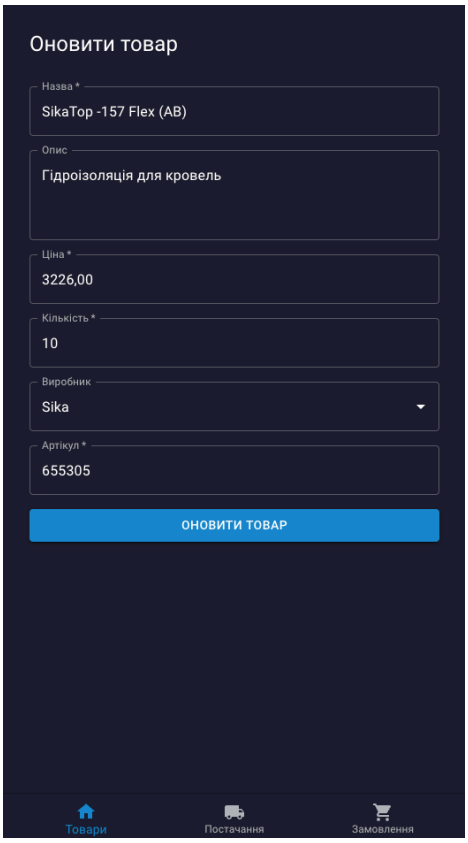

Рисунок 4.17 – вікно оновлення товару

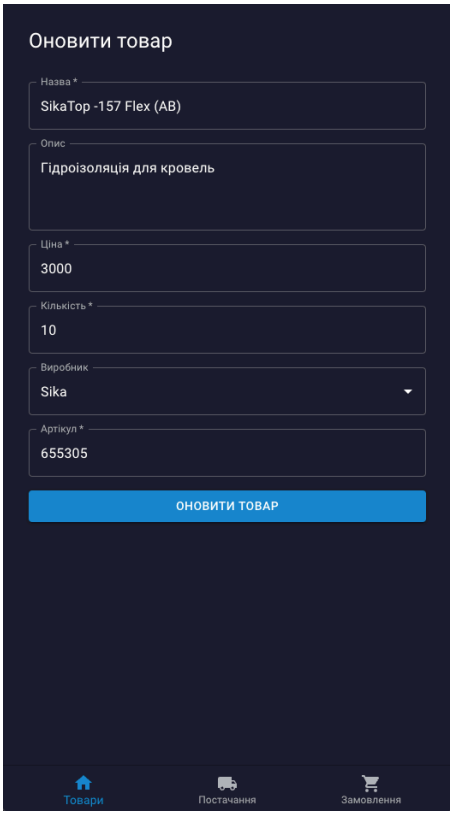

Рисунок 4.18 – оновлення ціни товару

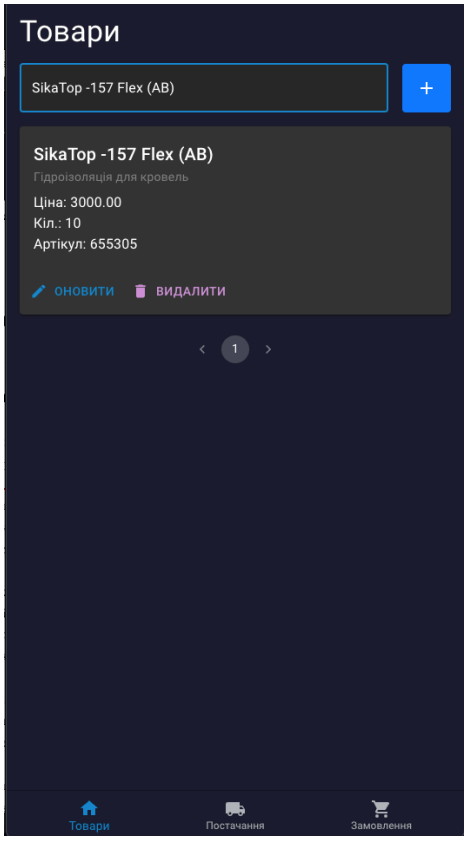

Рисунок 4.19 – товар з оновленою ціною

4.4.1.2 Робота з поставками

Програмне забезпечення надає користувачу змогу взаємодіяти з поставками, а саме: пошук поставки за її номером, видалення створення, оновлення та закриття поставки. Далі будуть наведені скріншоти роботи графічного інтерфейсу з вище перерахованим функціоналом.

Функціонал перегляду поставок.

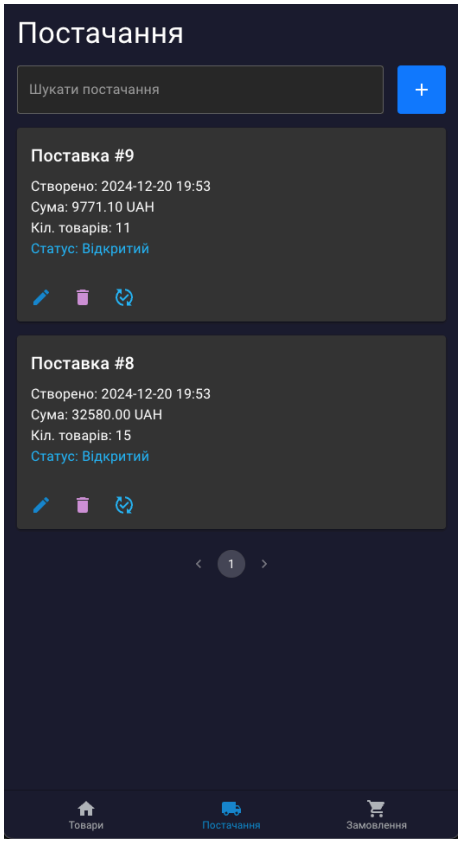

Рисунок 4.20 – список всіх поставок

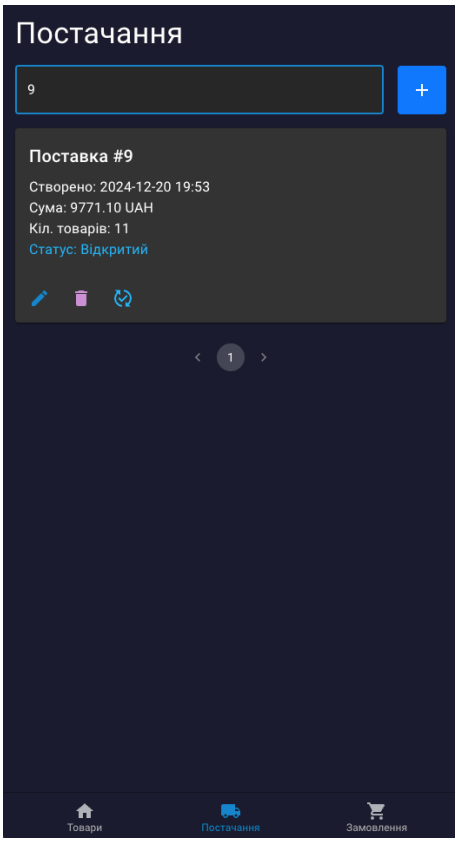

Рисунок 4.21 – пошук поставки за номером

Функціонал створення поставок.

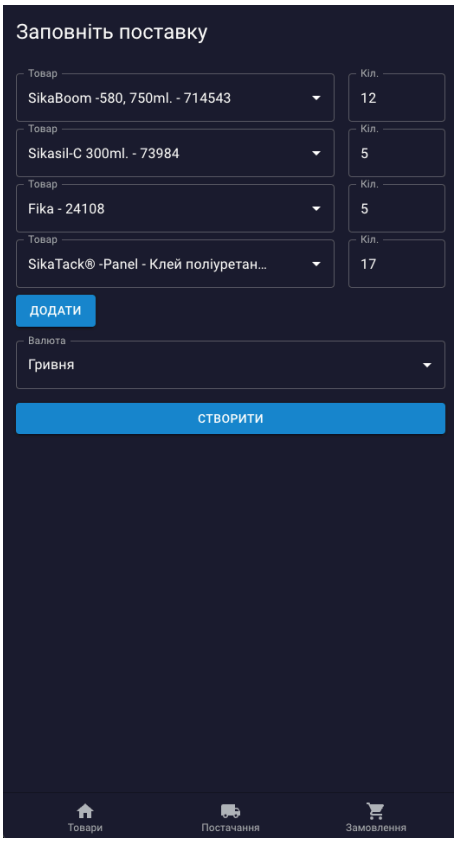

Рисунок 4.22 – форма створення поставки

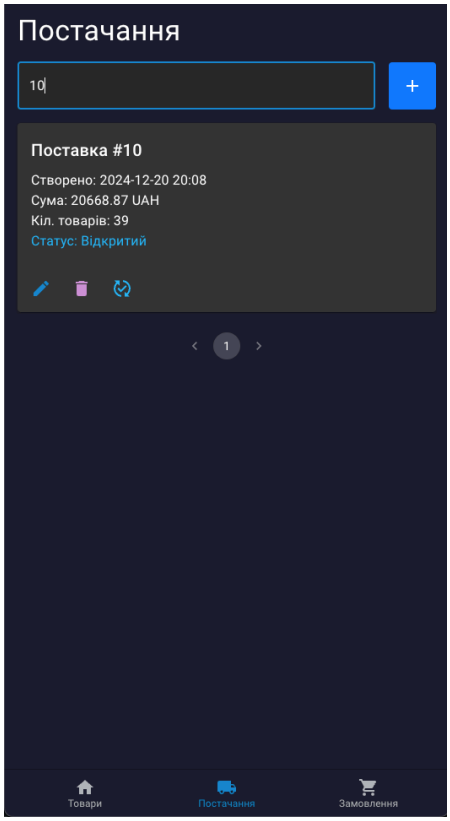

Рисунок 4.23 – створена поставка

Функціонал оновлення поставки.

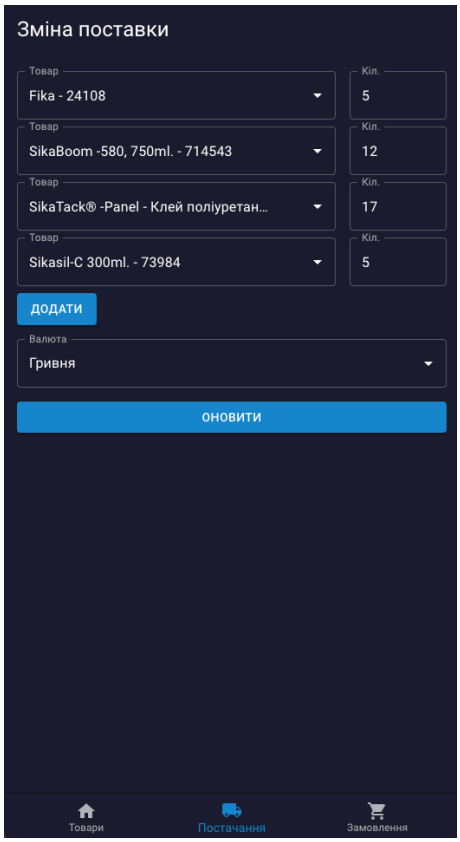

Рисунок 4.24 – форма оновлення поставки

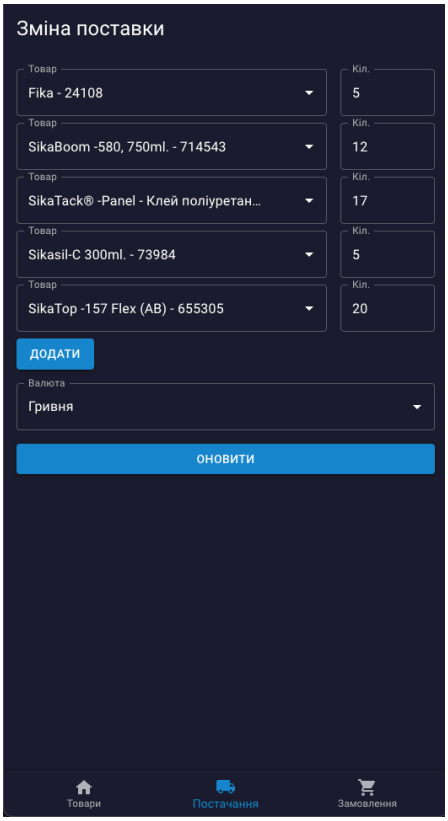

Рисунок 4.25 – додавання нового товару в поставку

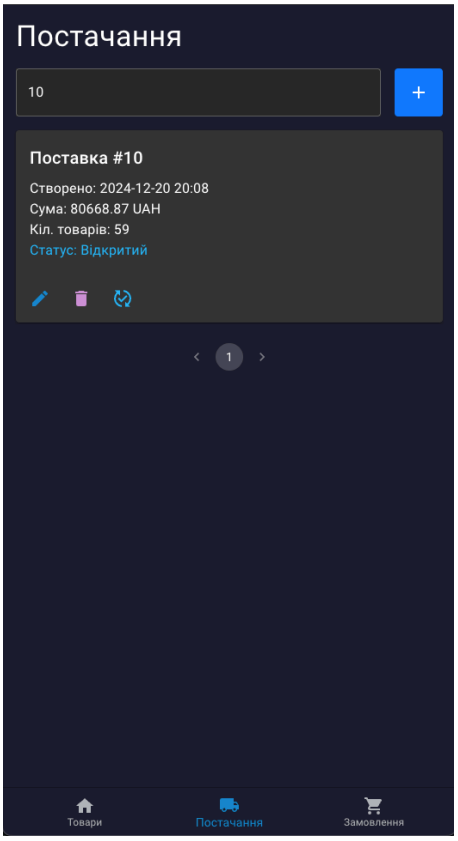

Рисунок 4.26 – оновлена поставка

Функціонал закриття поставки.

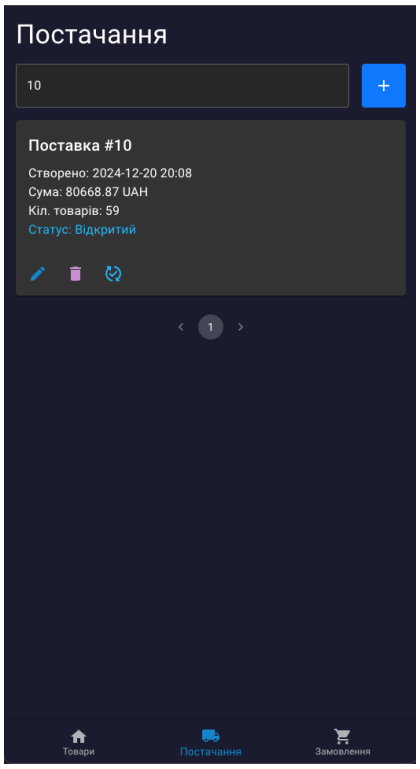

Рисунок 4.27 – поставка з «відкритим» статусом

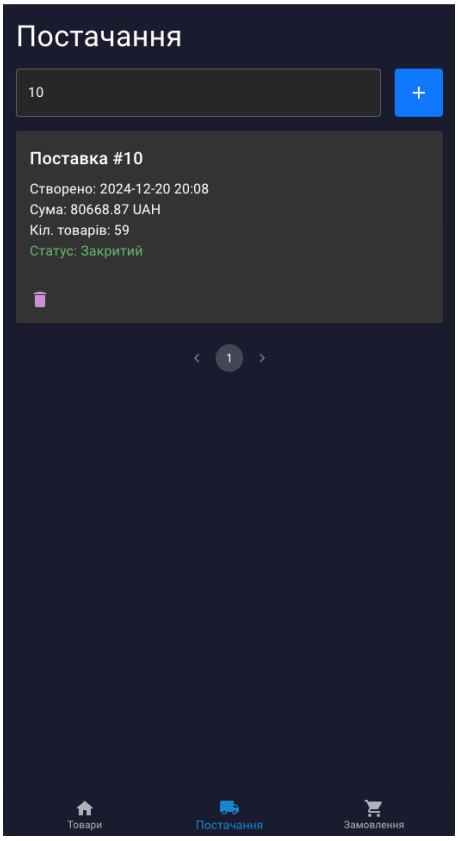

Рисунок 4.28 – поставка з «закритим» статусом

Функціонал видалення поставки.

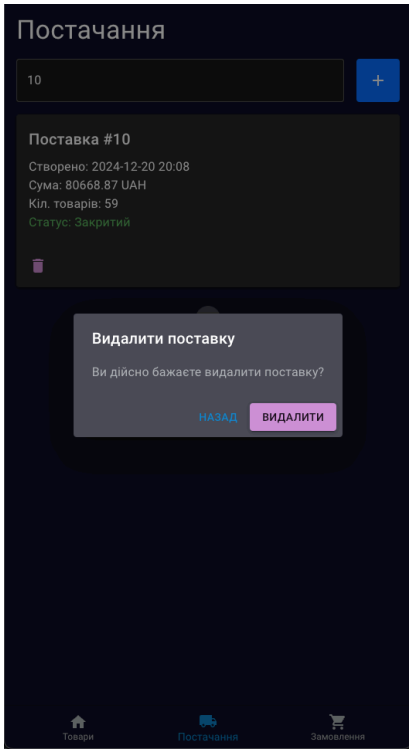

Рисунок 4.29 – вікно видалення поставки

Після видалення поставки її неможливо знайти через пошук.

Рисунок 4.30 – пошук видаленої поставки

4.4.1.3 Робота з замовленнями

Програмне забезпечення надає користувачу змогу взаємодіяти замволенняи, а саме: пошук замовлення за його номером, видалення створення, оновлення статусу замовлення. Далі будуть наведені скріншоти роботи графічного інтерфейсу з вище перерахованим функціоналом.

Функціонал перегляду замовлень.

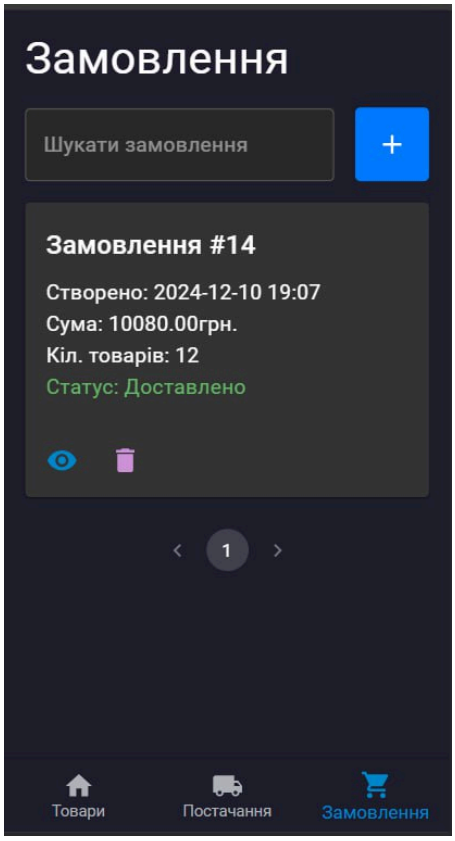

Рисунок 4.31 – сторінка замовлень

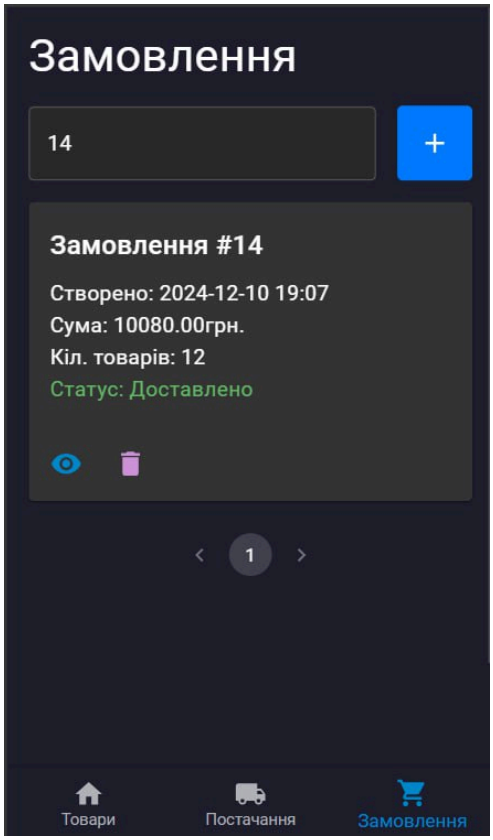

Рисунок 4.32 – пошук замовлення за його номером

Функціонал створення замовлення.

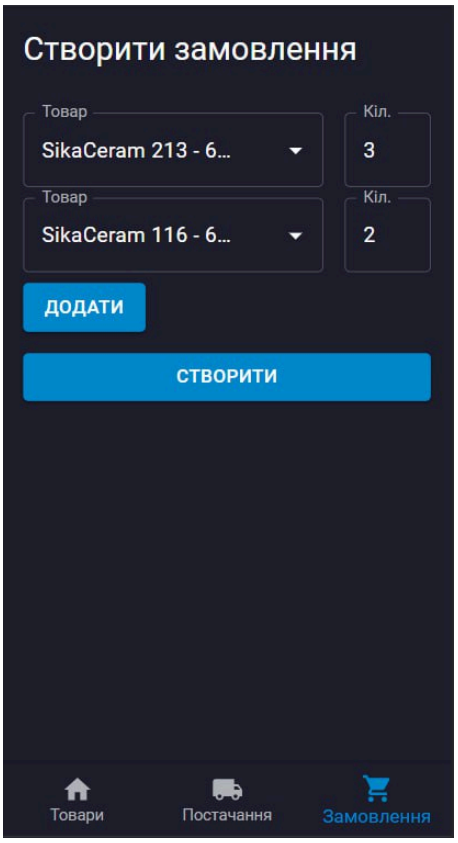

Рисунок 4.33 – форма створення замовлення

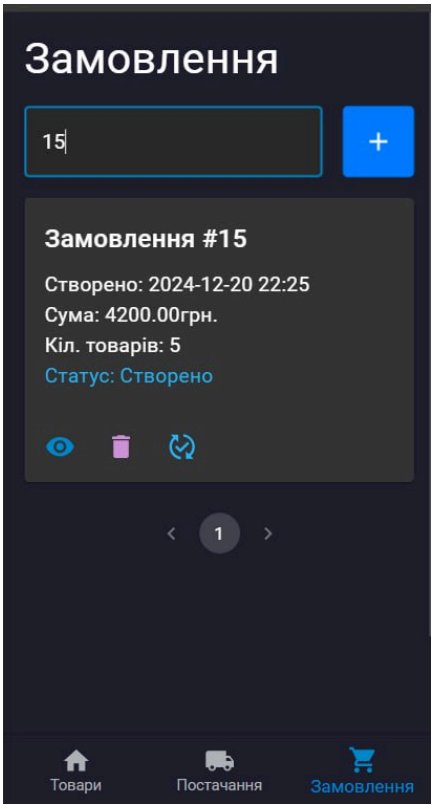

Рисунок 4.34 – створене замовлення

Функціонал оновлення статусу замовлення.

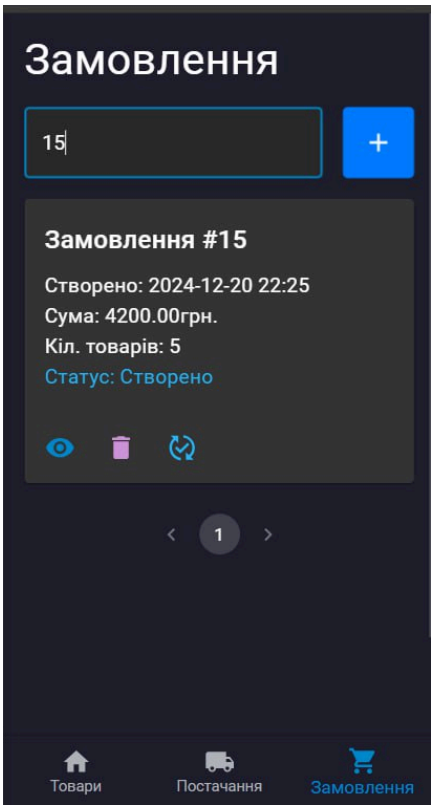

Рисунок 4.35 – замовлення з статусом «Створено»

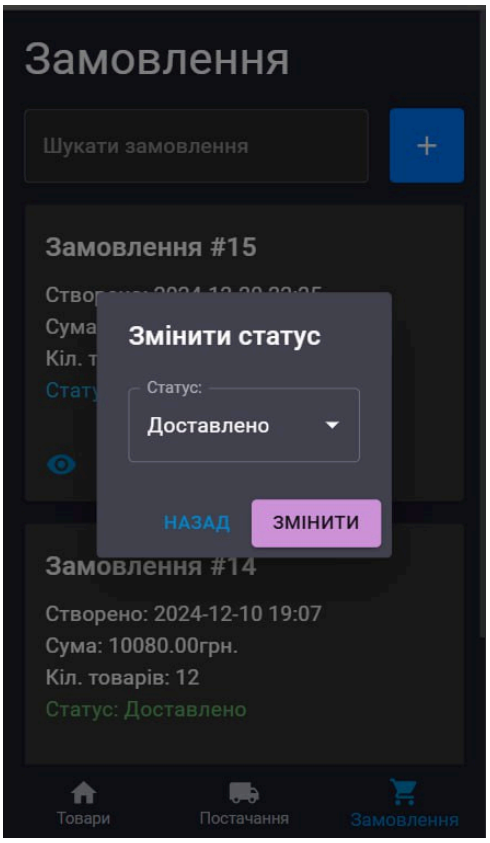

Рисунок 4.36 – вікно оновлення статусу замовлення

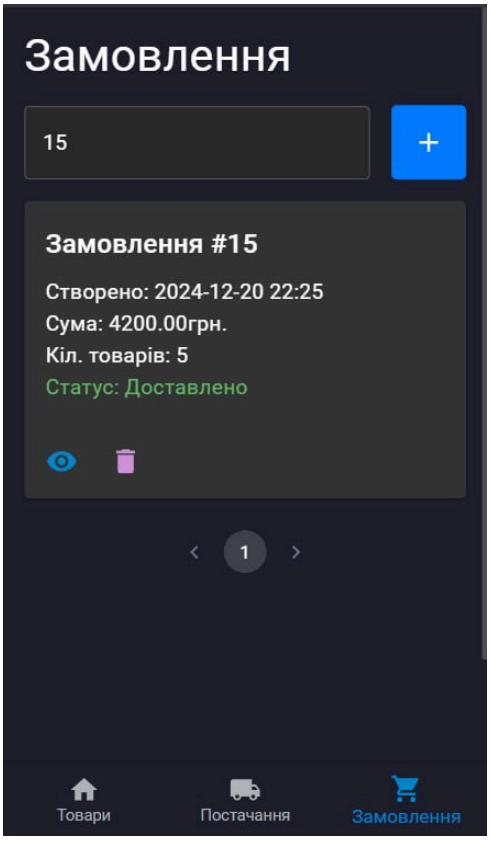

Рисунок 4.37 – замовлення з оновленим статусом «Доствлено»

Функціонал оновлення статусу замовлення.

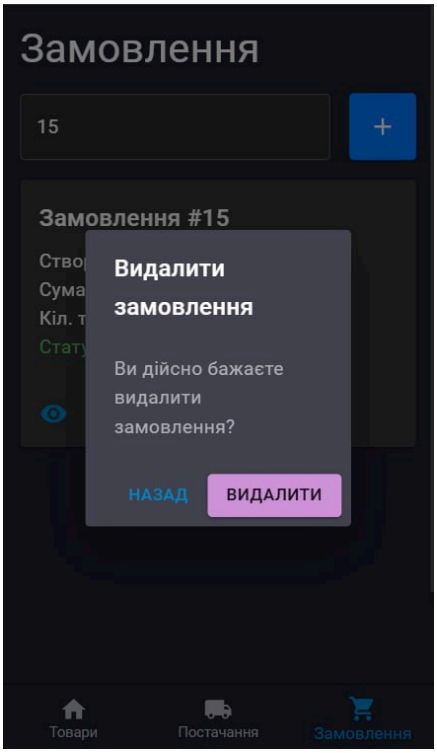

Рисунок 4.38 – вікно для видалення замовлення

Після видалення замовлення його неможливо знайти через пошук.

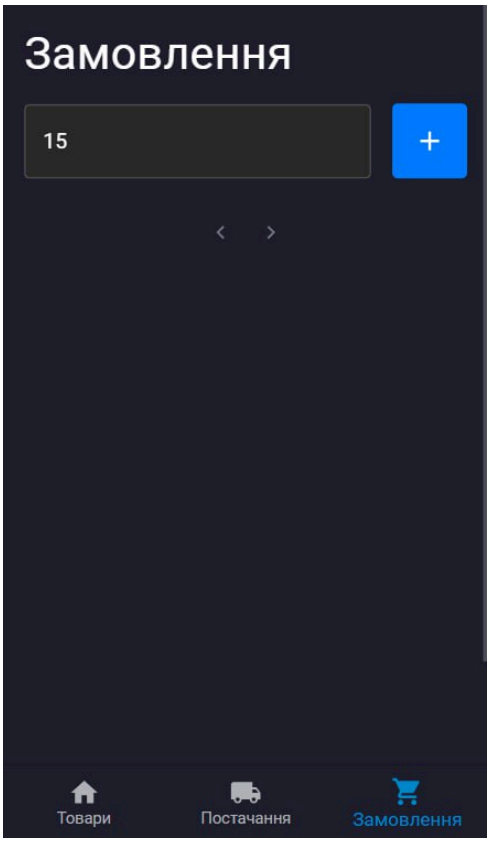

Рисунок 4.39 - головна сторінка після видалення замовлення

4.5 Висновки до розділу розробки програмного забезпечення

У даному розділі було описано процес розробки програмного забезпечення для вирішення поставлених задач. Проведено проєктування архітектури системи, яка забезпечує її модульність, масштабованість та відповідність сучасним вимогам до продуктивності й надійності. Розроблений функціонал відповідає заданим вимогам та охоплює всі основні процеси, необхідні для забезпечення ефективної роботи системи. Проведено тестування, яке підтвердило коректність і стабільність роботи програмного забезпечення. Реалізоване рішення готове до інтеграції в робоче середовище й подальшого розширення функціональних можливостей.

5 ДОСЛІДЖЕННЯ СИСТЕМИ

5.1 Мета дослідження

Метою дослідження є перевірка поведінки React-додатку при заповненні даних у формах, особливо при некоректному введенні та неповному заповненні та дослідження стресостійкості серверної частини за допомого відправки одночасної відправки великої кількості запитів на сервер. Дослідження мало на меті оцінити ефективність валідації та обробки помилок та стрейсостійкості серверної частини.

5.2 Методологія дослідження

Методологія дослідження - це систематична структура, яка використовується для вирішення дослідницької проблеми шляхом застосування найкращих і найбільш доцільних методів для проведення дослідження, узгоджених з метою і завданнями вашого дослідження [20].

В якості методолгії дослідження web додатку був вибраний емпіричний метод, а саме ручне тестування.

Емпіричний метод - метод наукового дослідження навколишньої реальності шляхом досвіду за допомогою експериментів та спостережень [21].

Ручне тестування це - це тип тестування програмного забезпечення, при якому тестовий кейс виконується тестувальником вручну без допомоги будь-яких автоматизованих інструментів [22].

5.3 Тестування обробки помилок web додатку

Web додаток складається з трьох модулів, а саме:

а) товари;

б) замовлення;

в) постачання.

5.3.1 Тестування модуля «Товари»

Модуль «Товари» складається з наступних полей:

а) назва (обов’язкове поле);

б) опис;

в) ціна (обов’язкове поле);

г) кількість (обов’язкове поле);

д) виробник;

е) артікул (обов’язкове поле).

Та наступних кнопок, а саме:

а) створити.

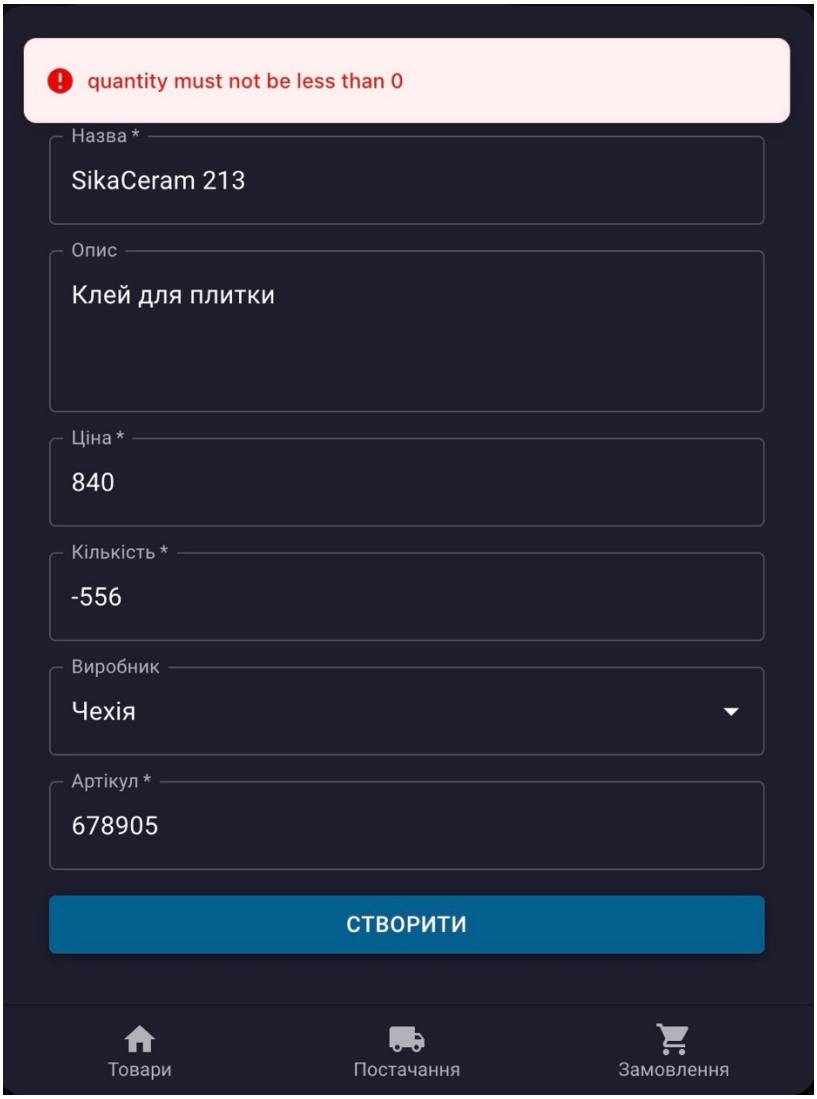

Рисунок 5.1 – Введена негативна кількість товару

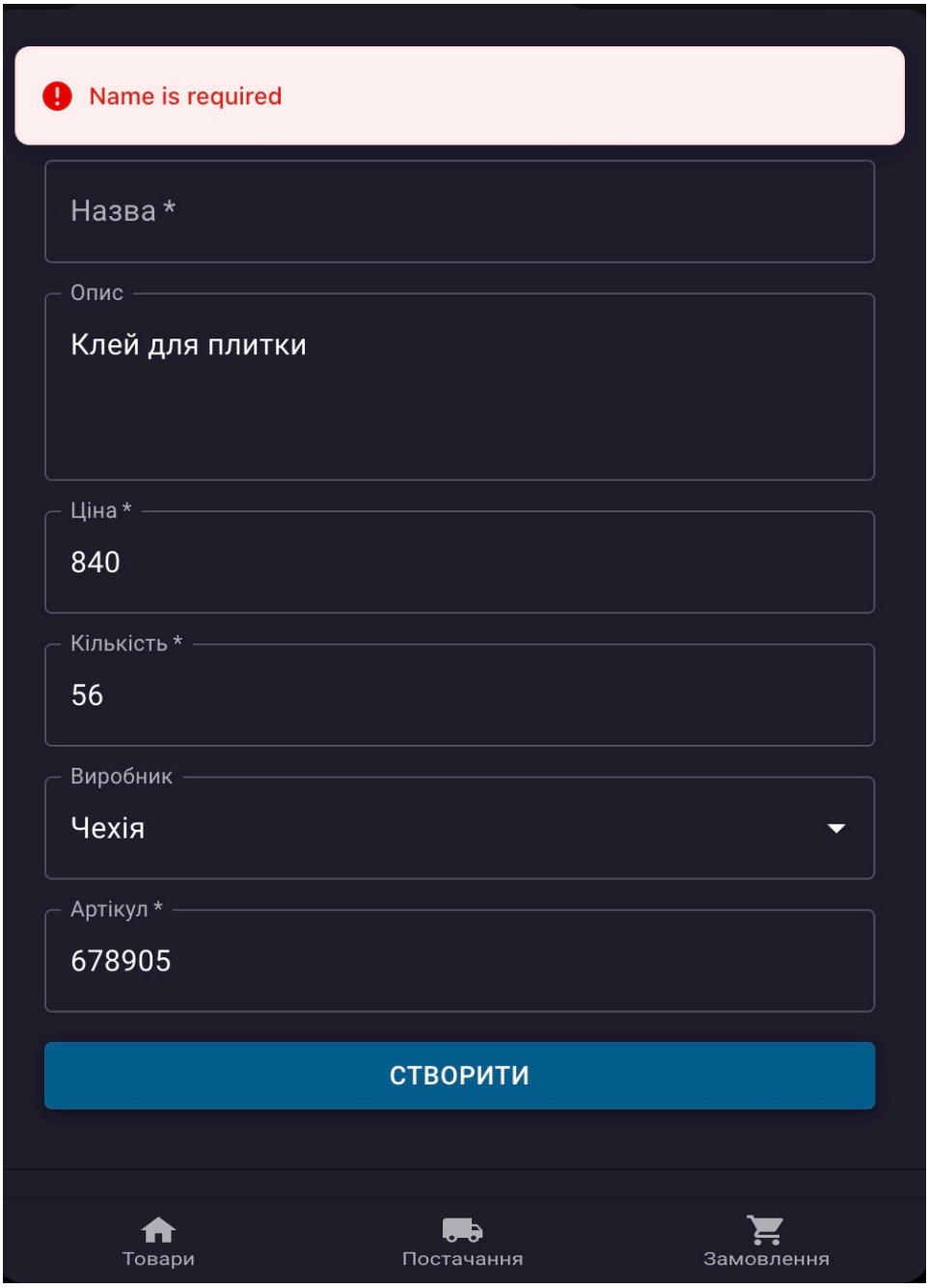

Рисунок 5.2 – Поле «Назва» не було заповнене

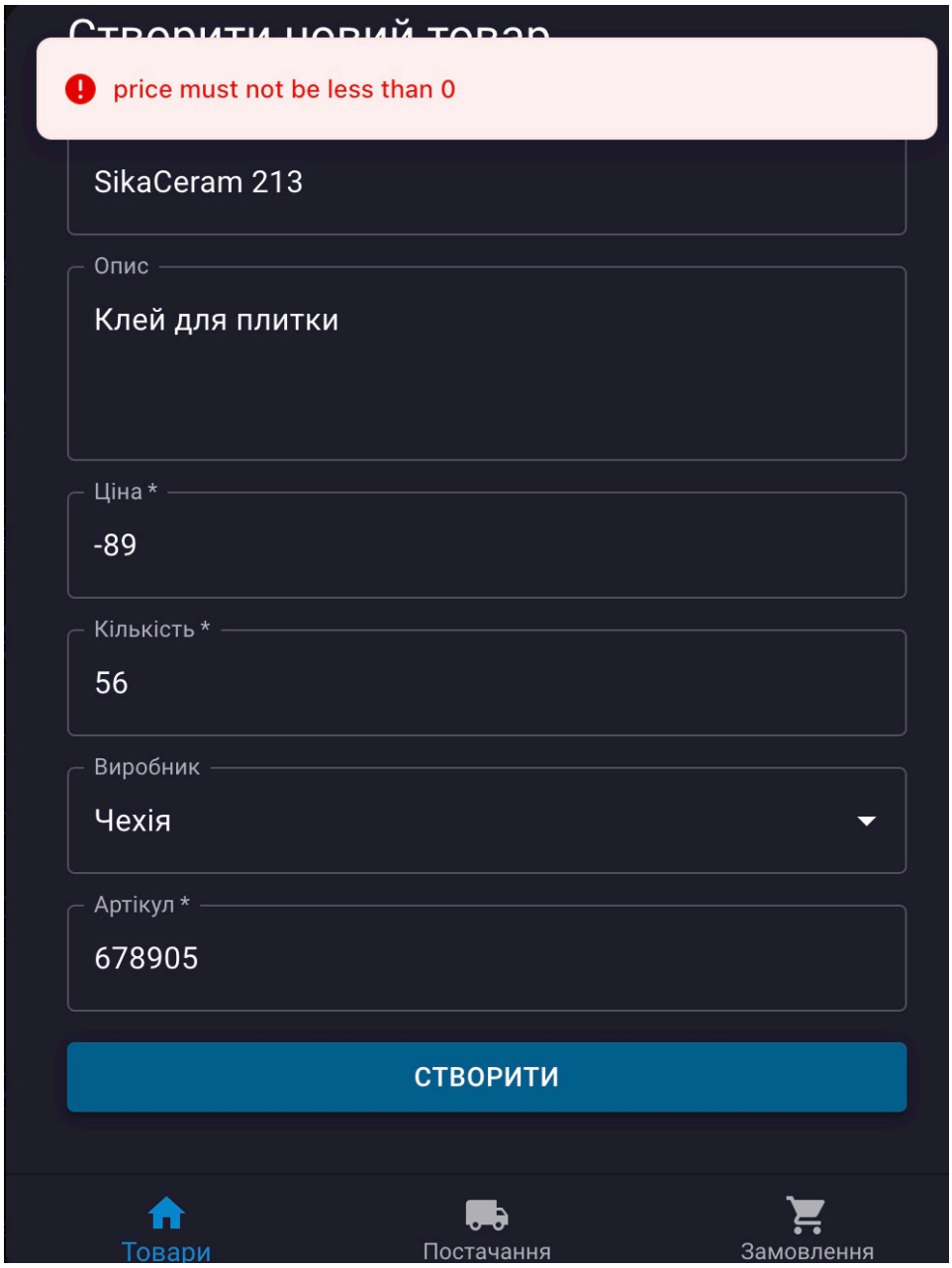

Рисунок 5.3 – Поле «Артікул» не було заповнене

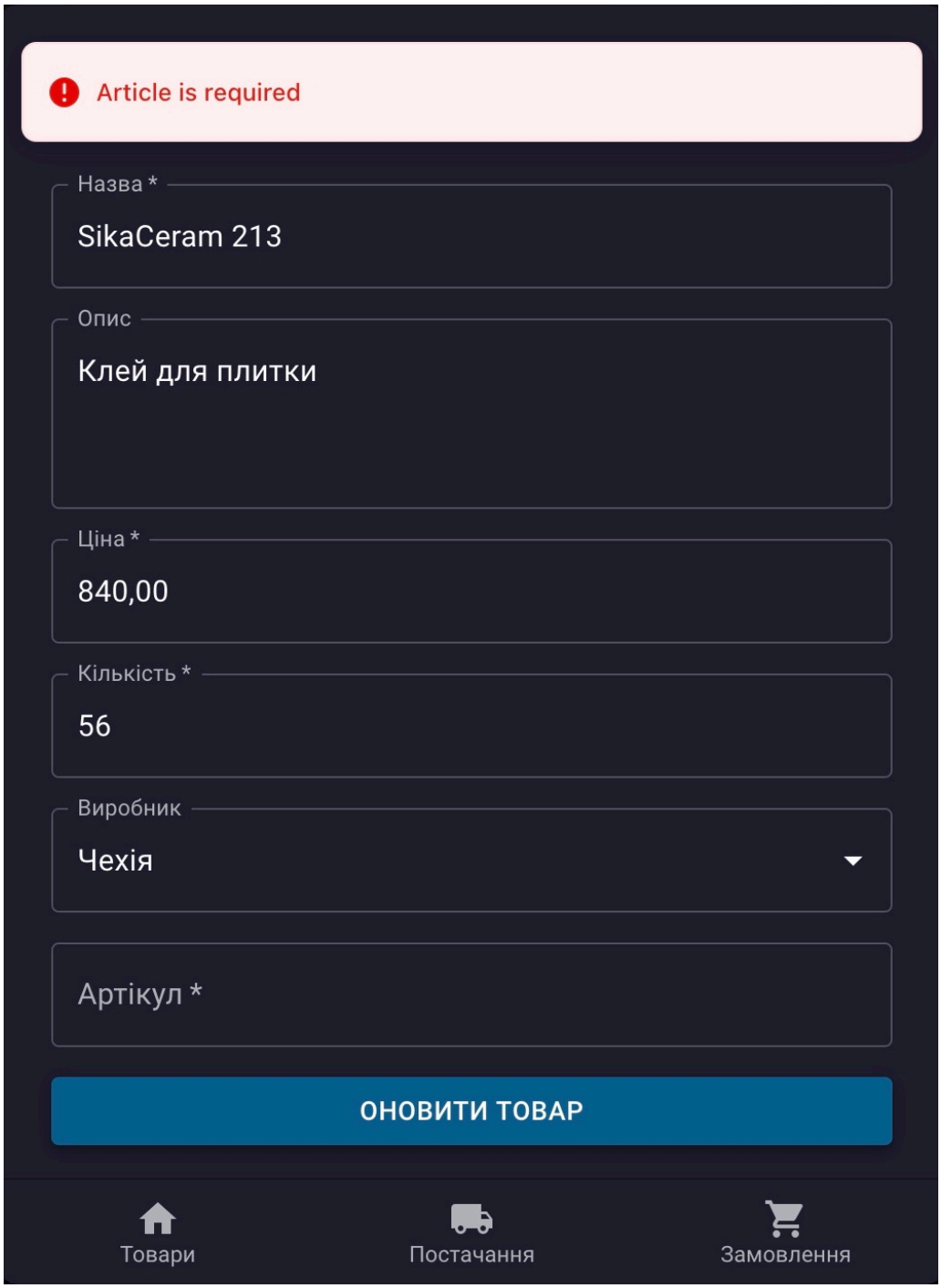

Рисунок 5.4 – Введена негативна ціна товару

5.3.2 Тестування модуля «Замовлення»

Модуль «Замовлення» складається з наступних полей:

а) товар (обов’язкове поле);

б) кількість (обов’язкове поле).

Та наступних кнопок, а саме:

а) додати;

б) створити.

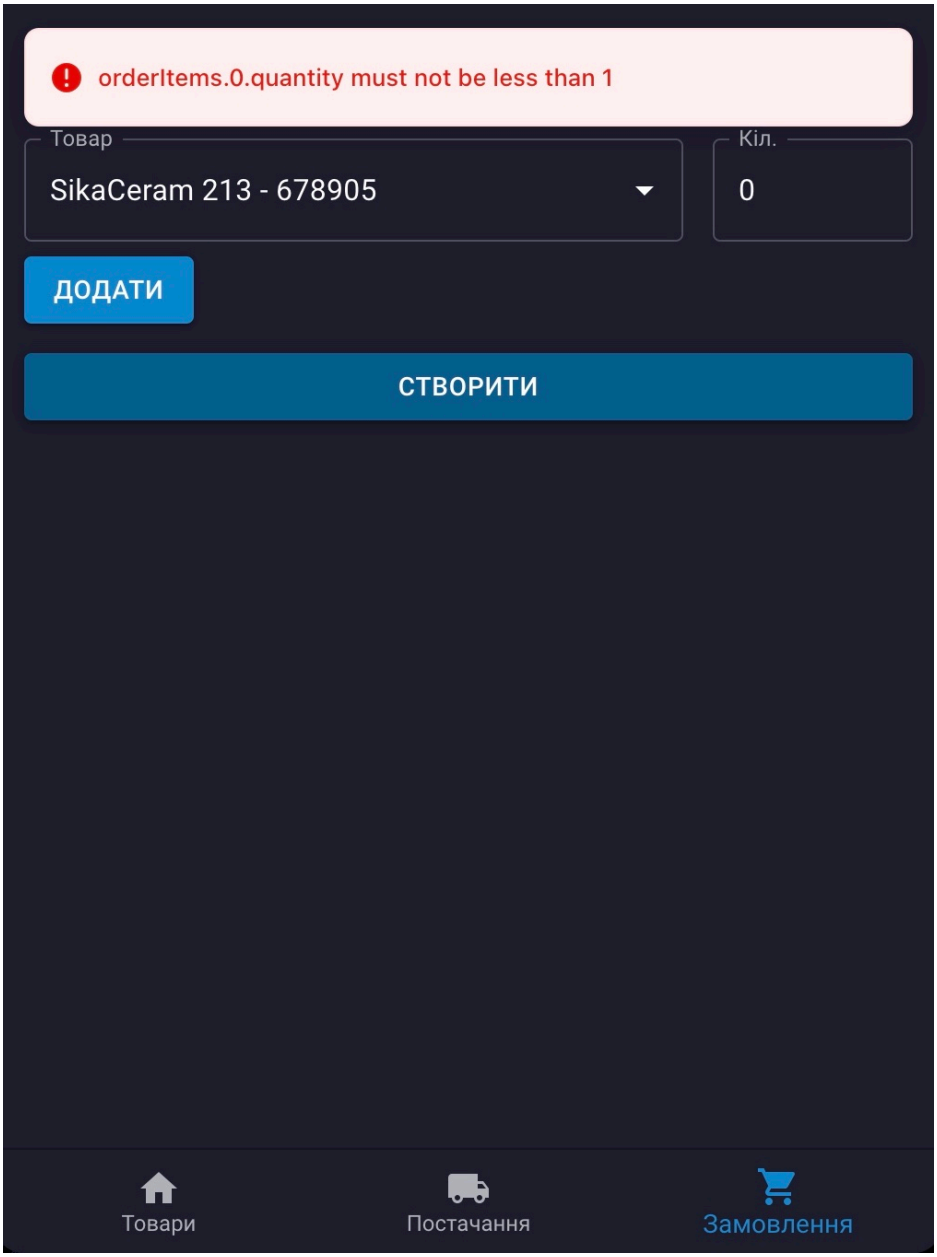

Рисунок 5.5 – Введена кількість товару нуль

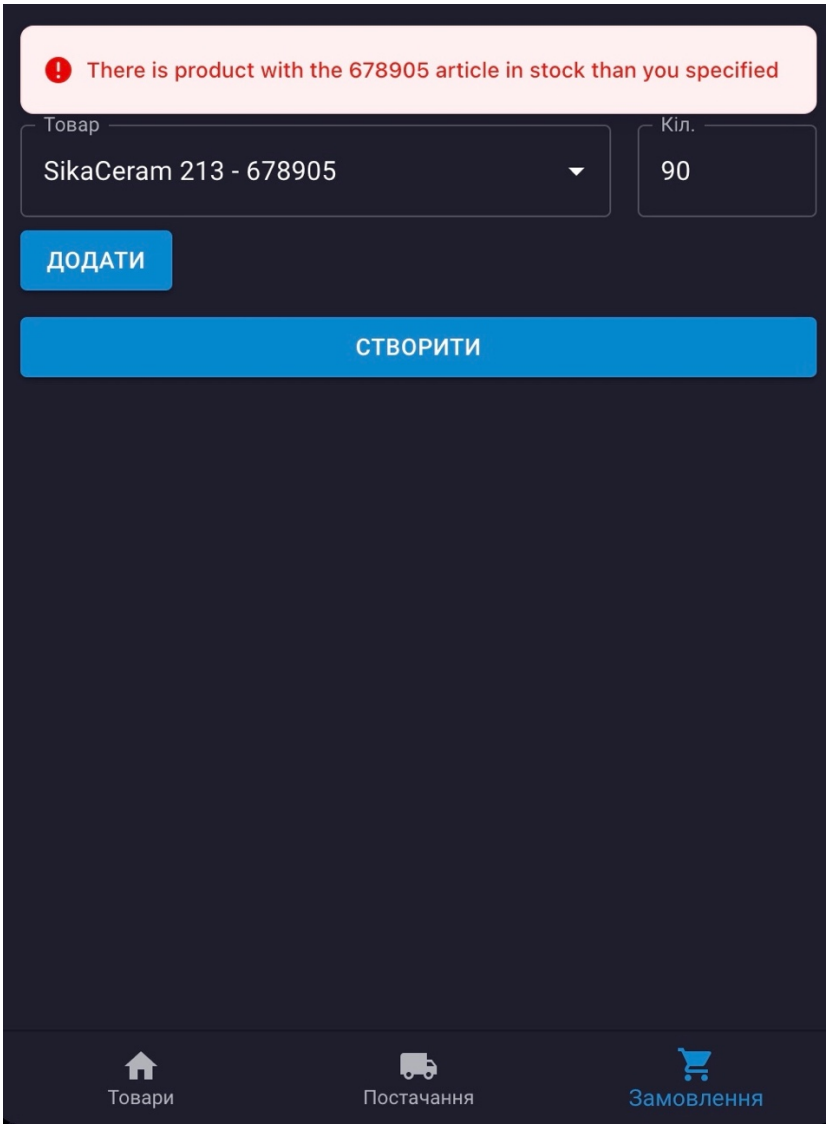

Рисунок 5.6 – Введена кількість товару більша ніж кількість на складі

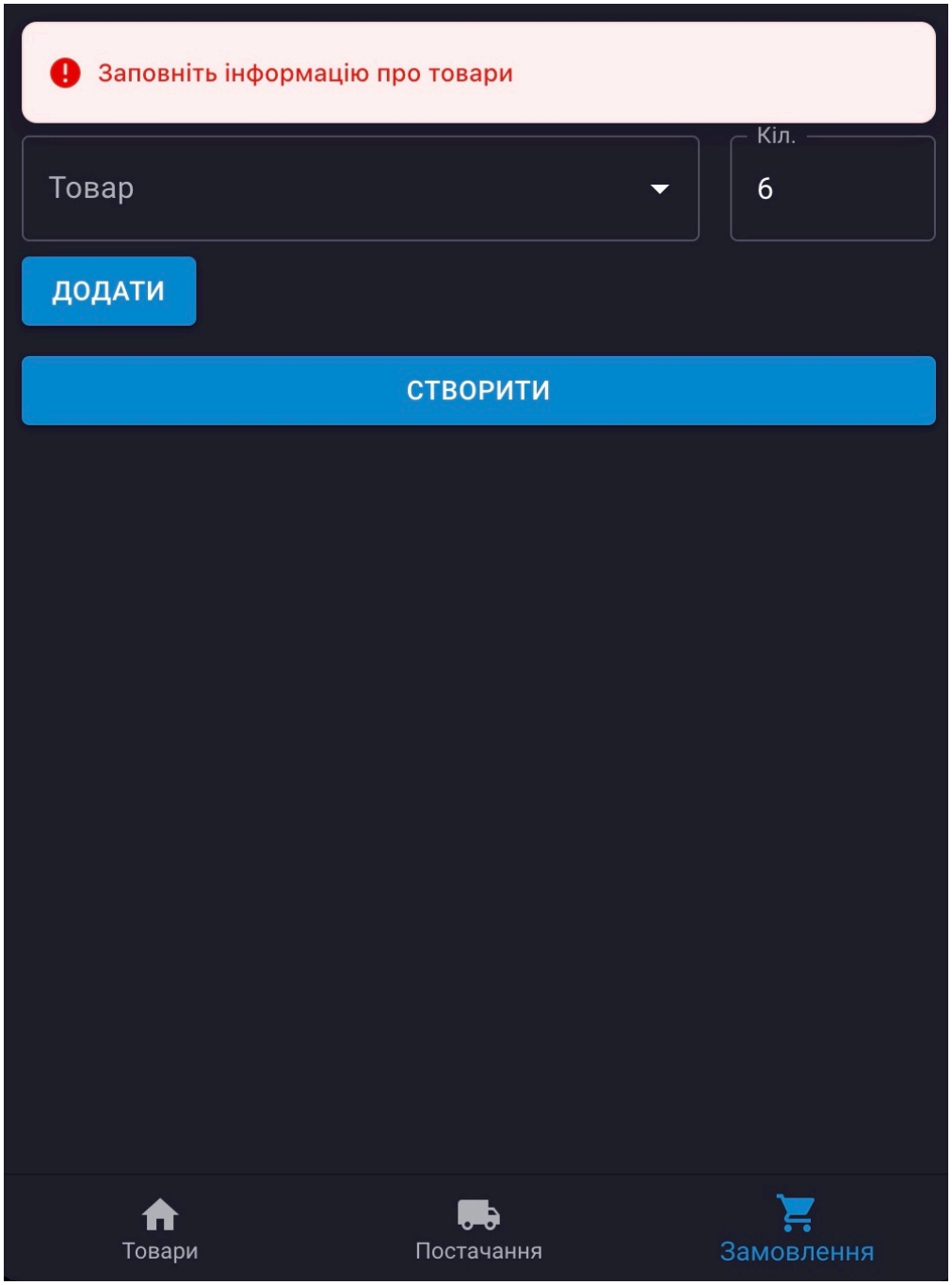

Рисунок 5.7 – інформація про назву товару не була заповнена

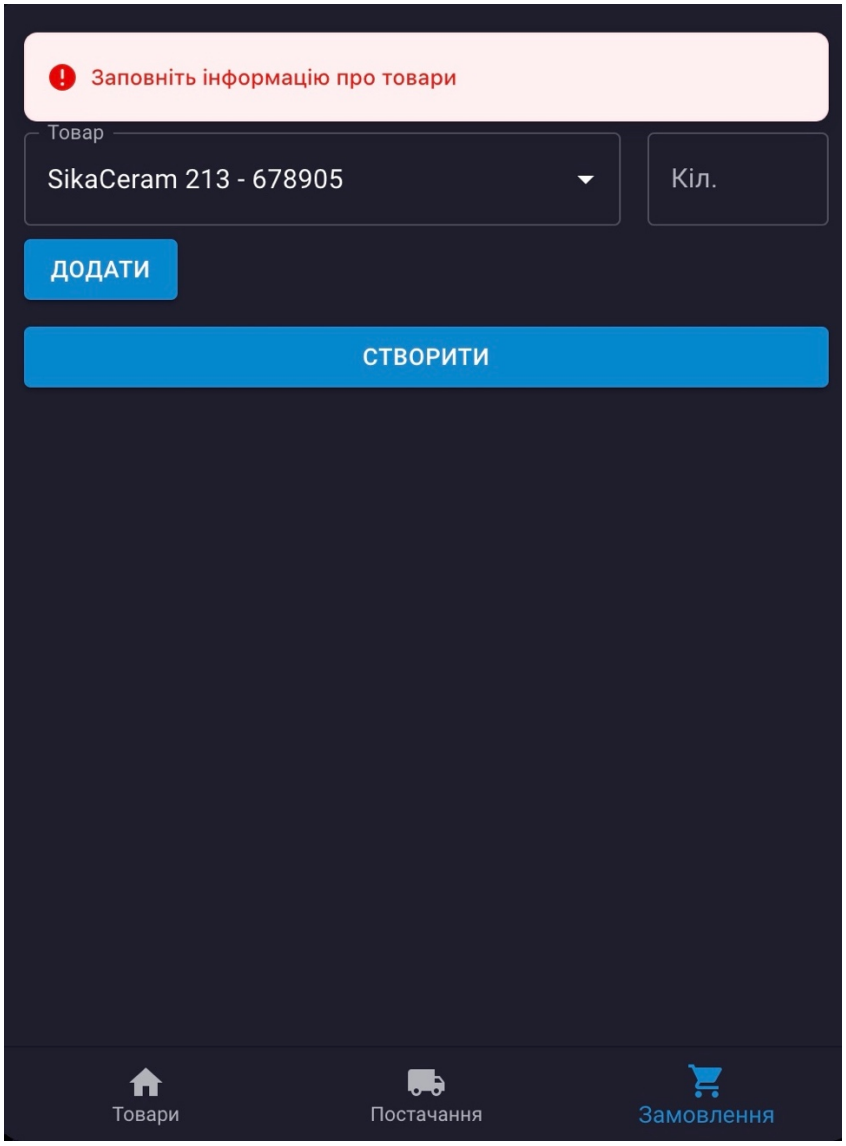

Рисунок 5.8 – інформація про кількість товару не була заповнена

5.3.3 Тестування модуля «Постачання»

Модуль «Постачання» складається з наступних полей:

а) товар (обов’язкове поле);

б) валюта (обов’язкове поле);

в) кількість (обов’язкове поле).

Та наступних кнопок, а саме:

а) додати;

б) створити.

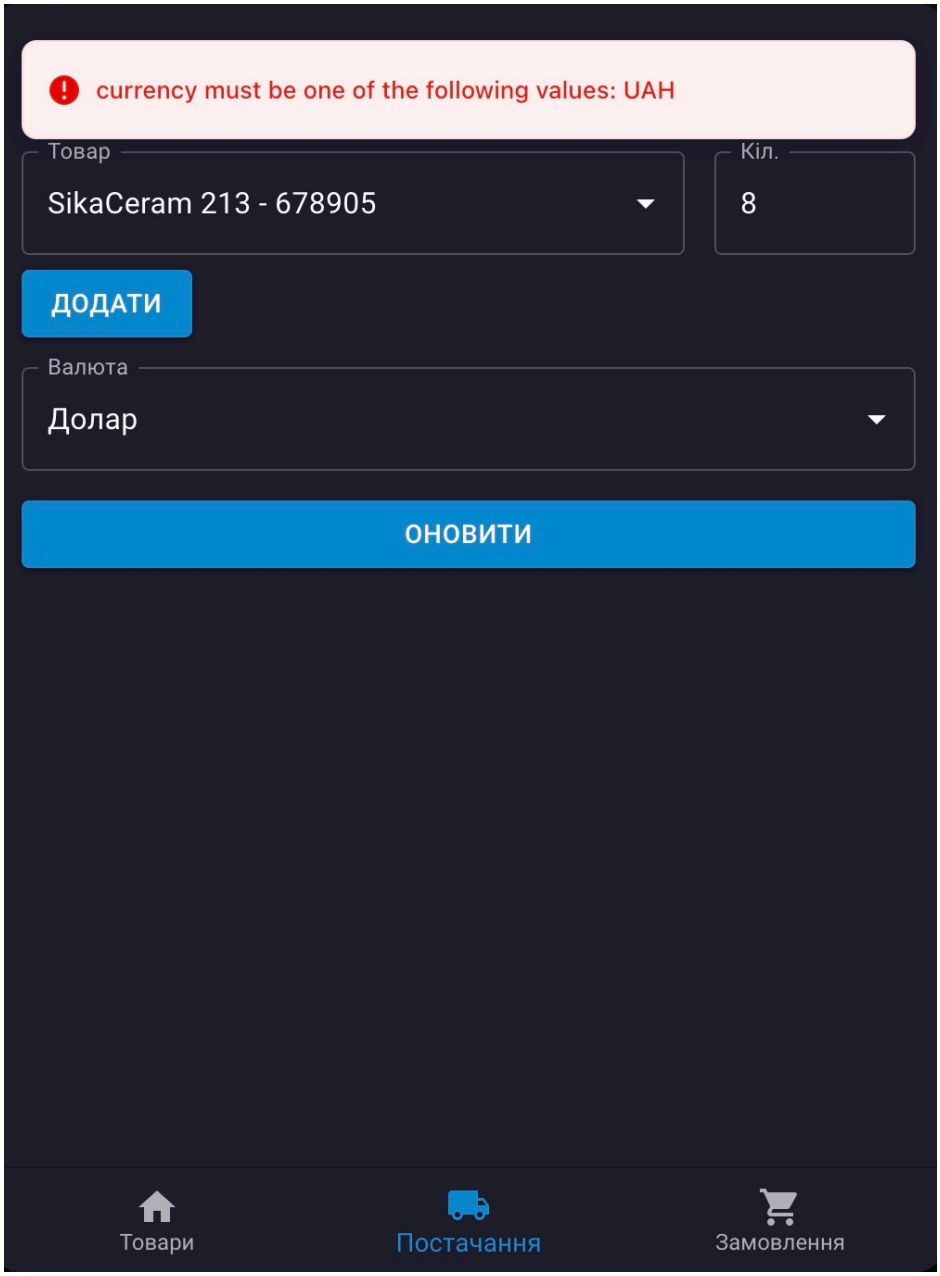

Рисунок 5.9 - Товар був створений в інший валюті

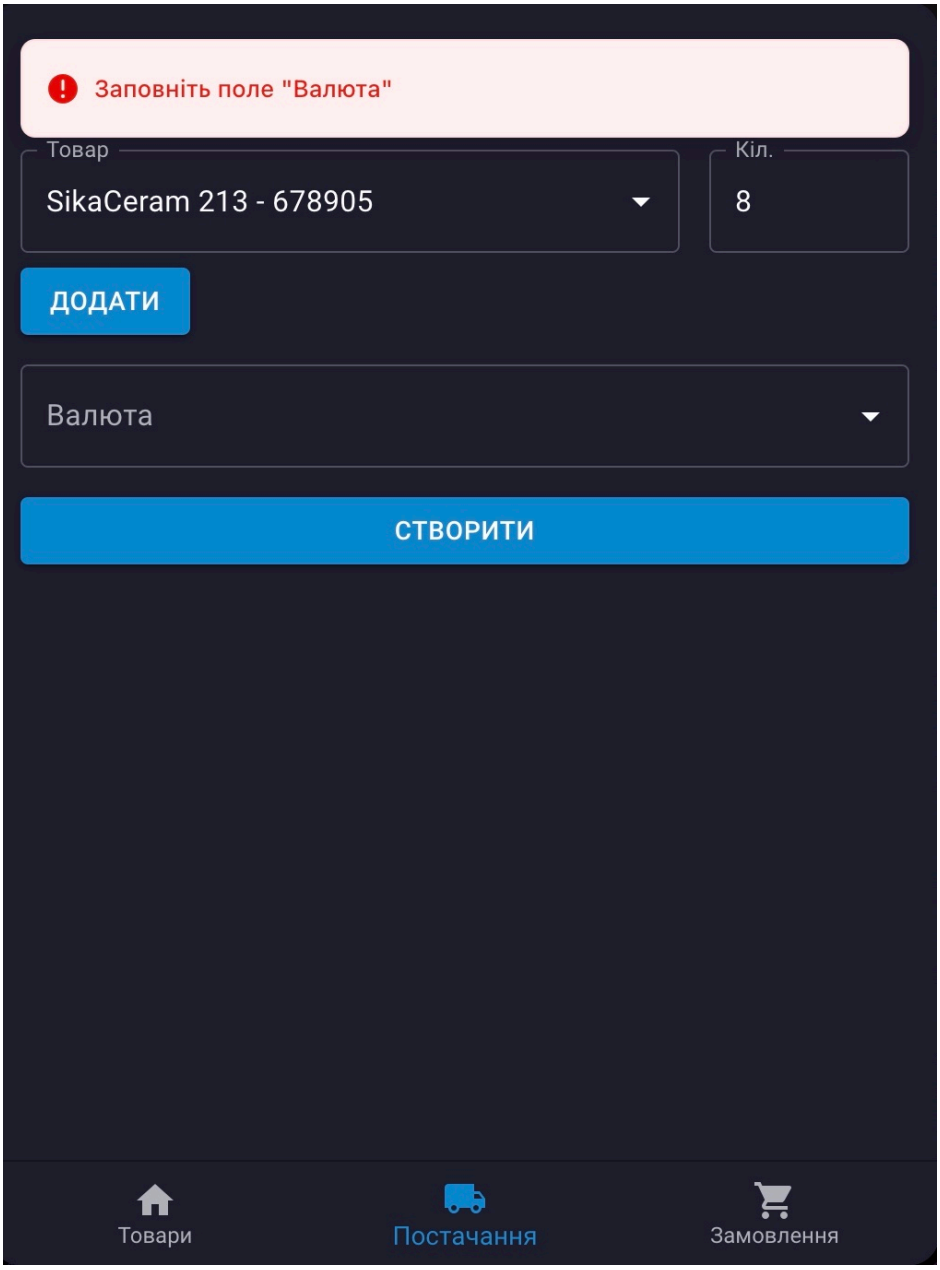

Рисунок 5.10 – не було заповнене поле «Валюта»

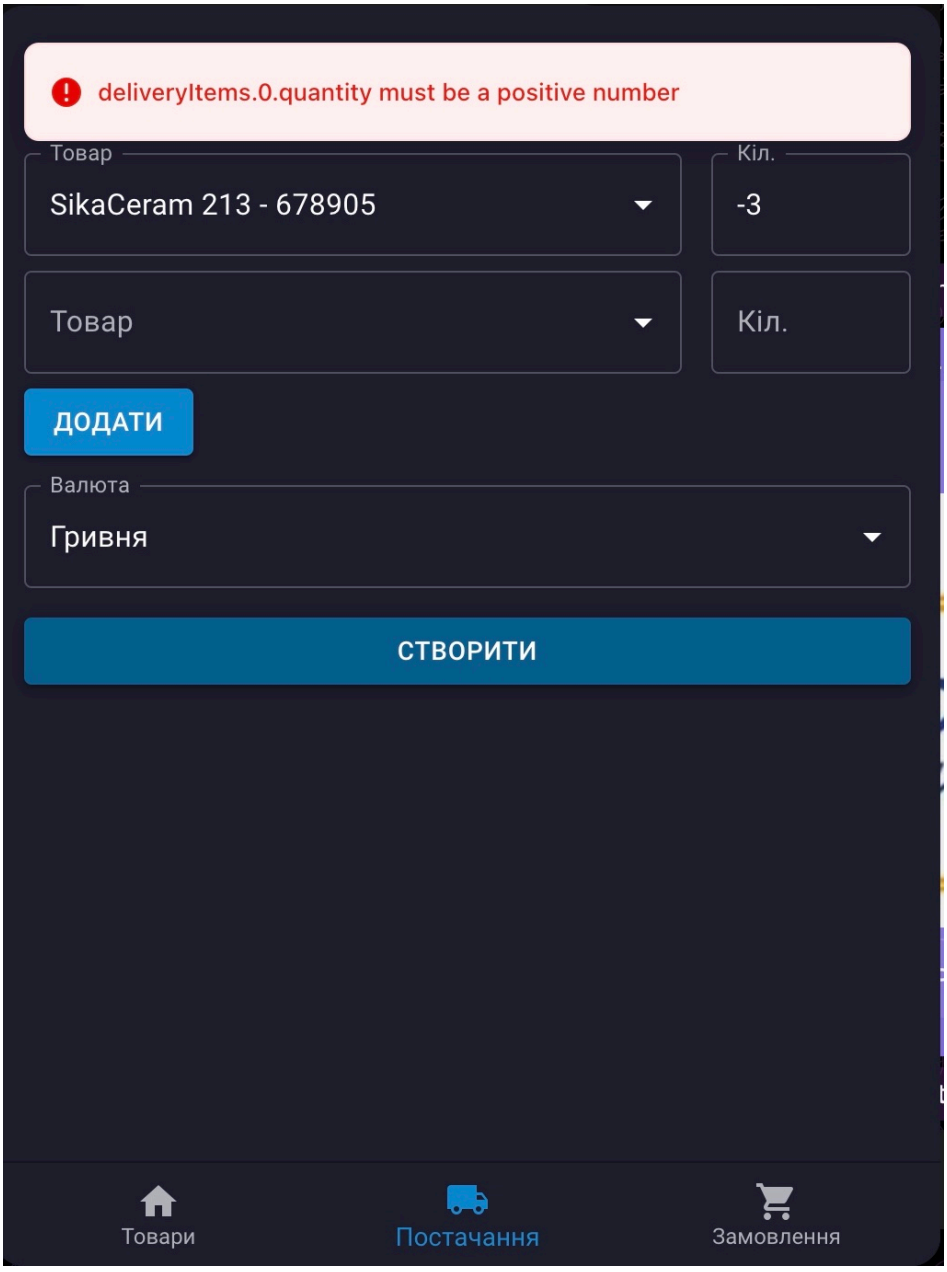

Рисунок 5.11 – Введена негативна кількість товару

5.4 Тестування стресостійкості серверної частини

Тестування стресостійкості серверної частини було проведено за допомогою інструмента «autocannon».

Autocannon - це інструмент для порівняльного аналізу HTTP/1.1, написаний на NodeJS, який широко використовується для вимірювання продуктивності програми. За допомогою autocannon ми можемо моделювати кілька запитів на секунду для навантажувального тестування для застосунку [23].

5.4.1 Тестування GET запиту продуктів

```
autocannon -c 100 -d 10 -p 3 http://localhost:3000/products
```

Цей запит виконує велику кількість GET-запитів для отримання списку продуктів.

Параметри запиту:

а) -c 100 - 100 одночасних з'єднань;

б) -d 10 - тривалість тесту 10 секунд;

в) -p 3 - збільшення кількості запитів на 3 кожну секунду.

```
autocannon -c 100 -d 10 -p 3 http://localhost:3000/products
Running 10s test @ http://localhost:3000/products
100 connections with 3 pipelining factor


┌─────────┬──────┬──────┬───────┬──────┬─────────┬─────────┬────────┐
│ Stat    │ 2.5% │ 50%  │ 97.5% │ 99%  │ Avg     │ Stdev   │ Max    │
├─────────┼──────┼──────┼───────┼──────┼─────────┼─────────┼────────┤
│ Latency │ 63 ms│ 69 ms│ 93 ms │104 ms│ 72 ms   │ 10.42 ms│ 190 ms │
└─────────┴──────┴──────┴───────┴──────┴─────────┴─────────┴────────┘
┌───────────┬─────────┬─────────┬─────────┬─────────┬─────────┬─────────┬─────────┐
│ Stat      │ 1%      │ 2.5%    │ 50%     │ 97.5%   │ Avg     │ Stdev   │ Min     │
├───────────┼─────────┼─────────┼─────────┼─────────┼─────────┼─────────┼─────────┤
│ Req/Sec   │ 3301    │ 3301    │ 4079    │ 4475    │ 4126.7  │ 344.41  │ 3300    │
├───────────┼─────────┼─────────┼─────────┼─────────┼─────────┼─────────┼─────────┤
│ Bytes/Sec │ 2.71 MB │ 2.71 MB │ 3.35 MB │ 3.67 MB │ 3.39 MB │ 283 kB  │ 2.71 MB │
└───────────┴─────────┴─────────┴─────────┴─────────┴─────────┴─────────┴─────────┘

Req/Bytes counts sampled once per second.
# of samples: 10

42k requests in 10.03s, 33.9 MB read
```

Рисунок 5.12 – результати виконання запитів для отримання продуктів

5.4.2 Тестування GET запиту продуктів з фільтрацією

```
autocannon -c 50 -d 15 -p 5
http://localhost:3000/products?sortBy=id&sortOrder=DESC
```

Цей запит виконує велику кількість GET-запитів для отримання списку продуктів відсортованих за id за спаданням.

Параметри запиту:

а) -c 50 - 50 одночасних з'єднань;

б) -d 15 - тривалість тесту 15 секунд;

в) -p 3 - збільшення кількості запитів на 5 кожну секунду.

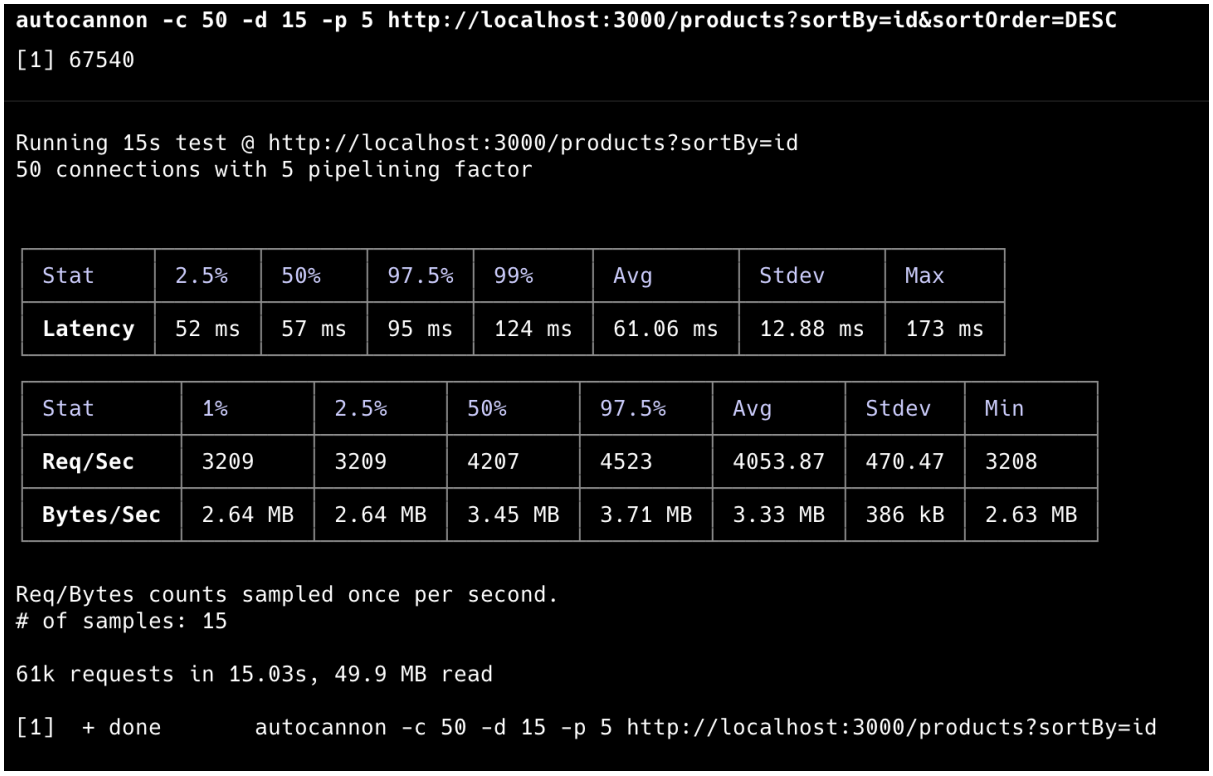

Рисунок 5.13 – результати виконання запитів для отримання продуктів з фільтрацією

5.5 Результати дослідження

Результати тестування показують, що API добре справляється з високими навантаженнями. Затримка, навіть у найбільш навантажених випадках, залишається в межах допустимих значень для більшості сучасних веб-застосунків. Продуктивність API, що дозволяє обробляти більше 4000 запитів на секунду, а також його здатність ефективно працювати з великими обсягами даних, свідчать про високу стресостійкість системи. Тому можна зробити висновок, що API здатне забезпечити стабільну роботу навіть при значних навантаженнях. Однак для забезпечення ще більшої надійності можна розглянути додаткові заходи оптимізації на рівні інфраструктури або кешування.

Перевірка веб-додатку на помилки для неправильно заповнених даних показала, що система коректно обробляє всі типи введення, що можуть призвести до помилок. Система надає чіткі і зрозумілі повідомлення про помилки, що робить взаємодію з додатком зручною і ефективною. Веб-

додаток добре справляється з обробкою помилок введення, що забезпечує високу якість користувацького досвіду та знижує ймовірність помилок при введенні даних.

ВИСНОВКИ

У ході розробки комп'ютерної системи складського обліку підприємства «АКС-ЮГ СИСТЕМА» з використанням Telegram-бота були реалізовані технічні та програмні рішення, спрямовані на оптимізацію управління складськими операціями та підвищення ефективності роботи підприємства. Система забезпечує автоматизацію обліку товарів, мінімізує ризики людських помилок, дозволяє оперативно отримувати інформацію про стан складу та виконувати ключові операції в режимі реального часу. Використання Telegram-бота як інтерфейсу взаємодії значно спрощує доступ користувачів до системи, роблячи її зручною та зрозумілою навіть для некваліфікованих операторів.

Застосування хмарного сервера AWS гарантує надійну обробку запитів, захист даних і масштабованість системи, що дозволяє її адаптувати до зростаючих потреб підприємства. Вибір бази даних PostgreSQL забезпечує ефективне збереження, обробку та аналіз великого обсягу даних, пов'язаних зі складськими операціями.

Розроблена система є сучасним, інтегрованим і ефективним інструментом для управління складським обліком, який відповідає актуальним технологічним і бізнес-вимогам. Її впровадження сприятиме зниженню витрат, підвищенню продуктивності та покращенню якості управління ресурсами підприємства.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Що таке база даних та для чого вона потрібна. Дата оновлення 09.08.2024 URL: https://cityhost.ua/uk/blog/scho-take-baza-danih-ta-dlya-chogo-vona-potribna.html#what-is-a-database-and-why-is-it-needed (дата звернення 07.12.2024)

2. Що таке СУБД (система керування базами даних)? Застосування, типи та приклад. Дата оновлення 28.06.2024 URL: https://www.guru99.com/uk/what-is-dbms.html (дата звернення 03.12.2024)

3. СУБД: які бувають, як вибрати. Дата оновлення 29.08.2022 URL: https://highload.today/uk/subd-yaki-buvayut-yak-vibrati/ (дата звернення 12.12.2024)

4. База даних PostgreSQL: інформація та короткий гайд. Дата оновлення 26.06.2024 URL: https://itproger.com/ua/news/baza-dannih-postgresql-informatsiya-i-kratkiy-gayd (дата звернення 14.12.2024)

5. Встановлення та налаштування pgAdmin 4 в режимі сервера. Дата оновлення 24.01.2020 URL: https://www.digitalocean.com/community/tutorials/how-to-install-configure-pgadmin4-server-mode-ru (дата звернення 08.12.2024)

6. Що таке інтегроване середовище розробки (IDE). Дата оновлення URL: https://aws.amazon.com/ru/what-is/ide/ (дата звернення 12.12.2024)

7. Що такое VisualStudioCode. Дата оновлення 04.12.2023 URL: https://timeweb.com/ua/community/articles/chto-takoe-visual-studio-code (дата звернення 04.12.2024)

8. Що таке NodeJs. Дата оновлення 04.08.2023 URL: https://devzone.org.ua/post/shcho-take-nodejs-osnovy-servernoyi-rozrobky-na-javascript (дата звернення 16.12.2024)

9. Що такое TypeScript і навіщо він потрібен. Дата оновлення 05.09.2023 URL: https://foxminded.ua/typescript/ (дата звернення 02.12.2024)

10. Вступ URL: https://docs.nestjs.com/ (дата звернення 01.12.2024)

11. Що таке React JS? Як почати вивчати Реакт? Навички для react developer. Дата оновлення (11.07.20230 URL: https://cases.media/en/article/sho-take-react-js-yak-pochati-vivchati-reakt-navichki-dlya-react-developer?srsltid=AfmBOora2GXIoJ-av6UPXhjntk-NqAglmsKn0aWLHX28B-7Cffm5FU52 (дата звернення 17.12.2024)

12. JSX в глибину 05.08.202 URL: https://devzone.org.ua/post/jsx-v-hlybynu (дата звернення 11.12.2024)

13. BotApi. Дата оновлення 10.06.2024 URL: https://grammy.dev/uk/guide/api (дата звернення 10.12.2024)

14. Про патерни проектування. Дата оновлення 14.02.2024 URL: https://foxminded.ua/paterny-proektuvannia/ (дата звернення 11.12.2024)

15. «Патерни проектування» / Е. Гамма, Р. Хелм, Р. Джонсон, Дж. Влісідес – Print2Print 2020. 460 с.

16. Монолітна архітектура ПЗ. URL: https://qalight.ua/baza-znaniy/shho-take-monolitna-arhitektura/ (дата звернення 15.12.2024)

17. PostgreSQL. Основи мови SQL: учбовий посібник / Є. П. Моргунов; під ред. Є. В. Рогова, П. В. Лузанова. — СПб.: БХВ-Петербург, 2018. - 336 с.

18. TypeORM URL: https://typeorm.io/ (дата звернення 13.12.2024)

19.«Чиста архітектура» / Роберт Мартін – Фабула 2019. 416 с.

20. Методологія досліджень. Дата оновлення 15.07.2022 URL: https://mindthegraph.com/blog/uk/ (дата звернення 09.12.2024)

21. Емпіричний метод URL: https://vue.gov.ua/ Емпіричний_метод (дата звернення 09.12.2024)

22. Ручне тестування URL: https://www.zaptest.com/uk (дата звернення 14.12.2024)

23. Навантажувальне тестування API NodeJS за допомогою Autocannon. Дата оновлення 13.02.2024 URL: https://alfa-brain.com/blog/345d7e8c-fc14-43d3-bd61-9636633dea49 (дата звернення 16.12.2024)

24. Навчальний курс із React. Дата оновлення 08.04.2019 URL: https://habr.com/ru/companies/ruvds/articles/446206/ (дата звернення 07.12.2024)

25. Компоненти і пропси URL: https://uk.legacy.reactjs.org/docs/components-and-props.html (дата звернення 03.12.2024)

26. Presentational and Container Components. Дата оновлення 23.03.2015 URL: https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0 (дата звернення 02.12.2024)

ДОДАТОК А – UML ДІАГРАМИ

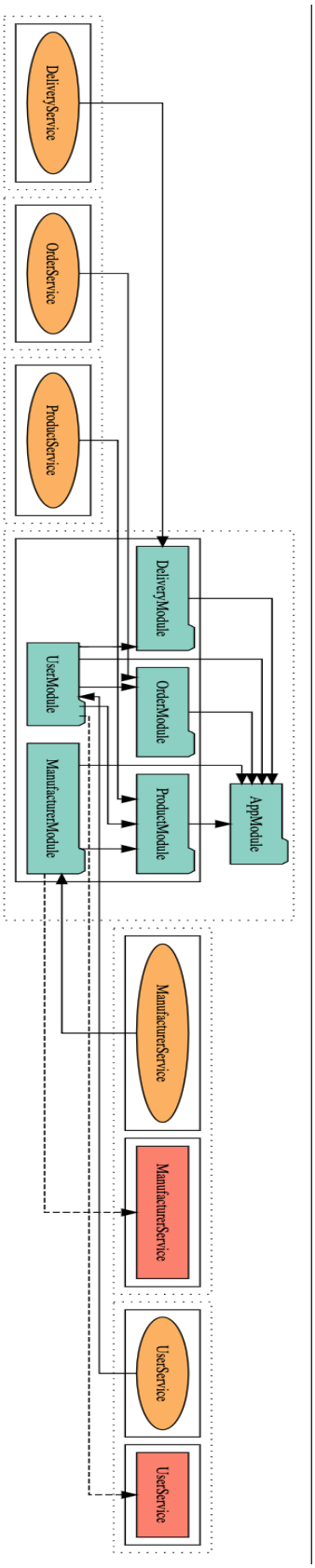

Рисунок А.1 – діаграма серверної частини

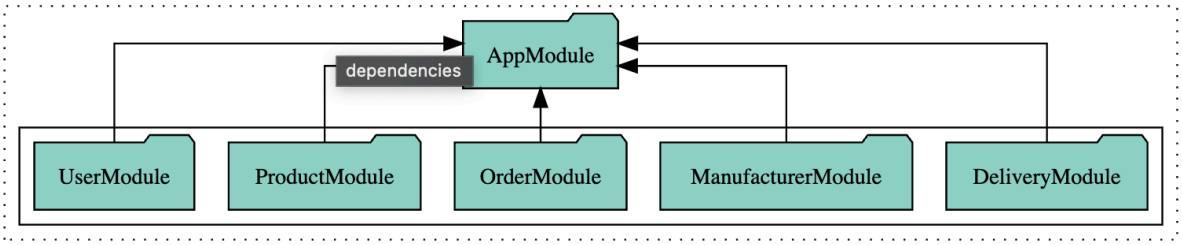

Рисунок А.2 – діаграма app module

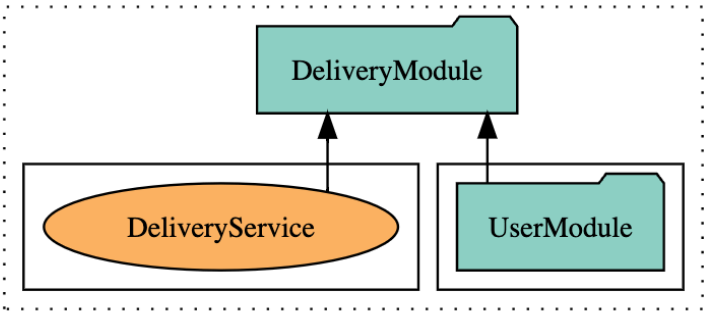

Рисунок А.3 – діаграма delivery module

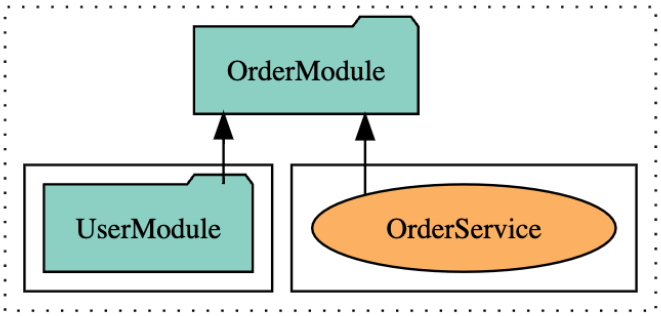

Рисунок А.4 – діаграма order module

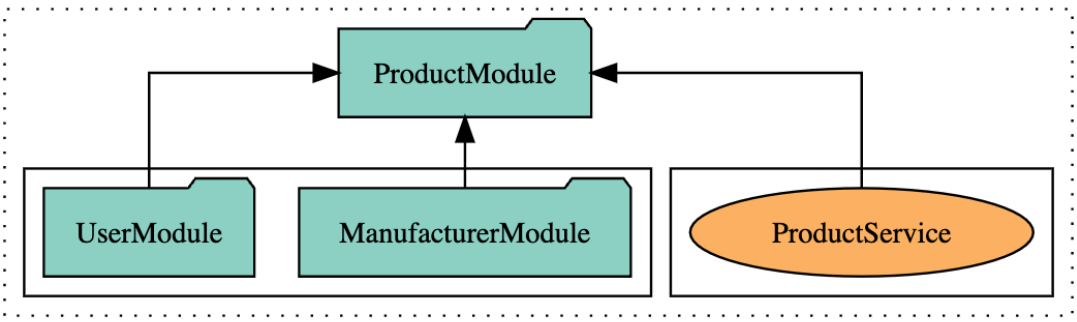

Рисунок А.5 – діаграма product module

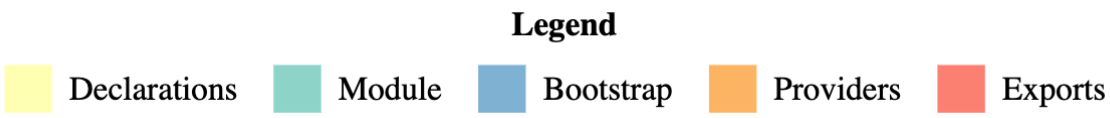

Рисунок А.6 – легенда до діаграм

ДОДАТОК Б

ТЕКСТ ПРОГРАМИ РОБОТИ ТЕЛЕГРАМ–БОТА

```
import React from 'react';
import {axiosProvider} from "../../axiosProvider";
import {productsResource} from "../../constants";
import {useNavigate} from "react-router-dom";
import {getUserTelegramId} from "../../utils/telegramUtils";
import {toast} from "sonner";
import ProductForm from "./Form";

const CreateProduct = () => {
  const navigate = useNavigate();

  const handleSubmit = async (e, values) => {
    e.preventDefault();
    const {
      name,
      price,
      quantity,
      article,
      manufacturerId,
      description,
    } = values;

    if (!name) {
      toast.error('Name is required');
      return;
    }
    if (!price) {
      toast.error('Price is required');
      return;
    }
    if (!article) {
      toast.error('Article is required');
      return;
    }
    if (!manufacturerId) {
      toast.error('Manufacturer is required');
      return;
    }

    try {
      await axiosProvider.post(productsResource,
      {
        name,
        article,
        description,
        price: +price,
```

```
        quantity: +(quantity || 0),
        manufacturerId: +(manufacturerId || 1),
        userTelegramId: getUserTelegramId()
      })
      navigate(`/${productsResource}`);
    } catch (e) {
      toast.error(e.response?.data?.message || e.message);
    }
  };

  return (
    <ProductForm
      values={{
        name: '',
        description: '',
        price: '',
        quantity: '',
        manufacturerId: '',
        article: '',
      }}
      titleTranslate="products.create.title"
      actionTranslate="actions.create"
      handleSubmit={handleSubmit}
    />
  );
};

export default CreateProduct;

import React from 'react';
import {CircularProgress,
} from '@mui/material';
import {axiosProvider} from "../../axiosProvider";
import {productsResource} from "../../constants";
import {useNavigate, useParams} from "react-router-dom";
import {toast} from "sonner";
import {useTranslation} from "react-i18next";
import ProductForm from "./Form";
import useEntity from "../../hooks/useEntity";

const EditProduct = () => {
  const {t} = useTranslation();
  const navigate = useNavigate();
  const {id} = useParams();
  const {entity: product, loading} = useEntity(id,
productsResource);
  const handleSubmit = async (e, values) => {
    e.preventDefault();
    const {
      name,
      price,
```

```
    quantity,
    article,
    manufacturerId,
    description,
  } = values;

  if (!name) {
    toast.error('Name is required');
    return;
  }
  if (!price) {
    toast.error('Price is required');
    return;
  }
  if (!article) {
    toast.error('Article is required');
    return;
  }
  if (!manufacturerId) {
    toast.error('Manufacturer is required');
    return;
  }

  try {
    await axiosProvider.patch(`/${productsResource}/${id}`,
      {
        name,
        article,
        description,
        price: +price,
        quantity: +(quantity || 0),
        manufacturerId: +manufacturerId,
      })
    navigate(`/${productsResource}`);
  } catch (e) {
    toast.error(e.response?.data?.message || e.message);
  }
};

if (loading) {
  return <CircularProgress />
}

if (!product) {
  toast.error(t('products.edit.error_msg_not_found'));
  navigate(`/${productsResource}`);
  return null
}

return (
  <ProductForm
```

```
      handleSubmit={handleSubmit}
      values={product}
      titleTranslate={"products.edit.title"}
      actionTranslate="products.edit.title"
    />
  );
};

export default EditProduct;

import {
  Box,
  Button,
  Dialog, DialogActions,
  DialogContent, DialogContentText,
  DialogTitle,
  Grid,
  TextField,
  Typography
} from "@mui/material";
import Autocomplete, {createFilterOptions} from
"@mui/material/Autocomplete";
import React, {useEffect, useMemo, useState} from "react";
import useManufacturer from "../../hooks/useManafacturer";
import {useTranslation} from "react-i18next";
import {axiosProvider} from "../../axiosProvider";
import {manufacturersResource} from "../../constants";
import {toast} from "sonner";

const filter = createFilterOptions();

const ProductForm = ({actionTranslate, handleSubmit,
titleTranslate, values}) => {
  const [manufacturerKey, setManufacturerKey] = useState('');
  const {t} = useTranslation();
  const {handleSearch, manufacturers, loading} =
useManufacturer();
  const [currentManufacturer, setCurrentManufacturer] =
useState('');
  const [formValues, setFormValues] = useState({...values});
  const selectedManufacturer = useMemo(() => {
    return manufacturers.find((manufacturer) =>
manufacturer.id === formValues.manufacturerId) || null;
  }, [loading, currentManufacturer])

  const handleCreateManufacturer = async (manufacturerName) =>
{
    try {
      const {data} = await
axiosProvider.post(`/${manufacturersResource}`, {name:
manufacturerName})
```

```
      setFormValues({...formValues, manufacturerId:
data.payload?.id});
    } catch (e) {
      setFormValues({...formValues, manufacturerId: null});
      toast.error(e.response?.data?.message || e.message);
    } finally {
      handleSearch('');
      setManufacturerKey(Date.now());
    }
  }

  const handleChange = (e) => {
    const { name, value } = e.target;
    setFormValues({ ...formValues, [name]: value });
  };

  const handleChangeManufacturer = async (e, value) => {
    if (value?.id === -1) {
      await handleCreateManufacturer(value.value);
      return;
    }

    setCurrentManufacturer(value?.id);

    setFormValues({ ...formValues, manufacturerId: value?.id
});
  };

  return (
    <Grid container justifyContent="center">
      <Box
        component="form"
        onSubmit={handleSubmit}
        noValidate
        sx={{
          display: 'flex',
          flexDirection: 'column',
          gap: 2,
          width: '100%',
          maxWidth: 600,
          padding: '16px',
          color: '#fff',
        }}
      >
        <Typography variant="h5" sx={{ marginBottom: 2 }}>
          {t(titleTranslate)}
        </Typography>
        <TextField
          label={t('products.create.name')}
          variant="outlined"
          name="name"
```

```
            defaultValue={formValues.name}
            value={formValues.name}
            onChange={handleChange}
            fullWidth
            required
          />
          <TextField
            label={t('products.create.description')}
            variant="outlined"
            name="description"
            value={formValues.description}
            onChange={handleChange}
            fullWidth
            multiline
            rows={3}
          />
          <TextField
            label={t('products.create.price')}
            variant="outlined"
            name="price"
            value={formValues.price}
            onChange={handleChange}
            fullWidth
            type="number"
            inputProps={{ step: '0.01' }}
            required
          />
          <TextField
            label={t('products.create.quantity')}
            variant="outlined"
            name="quantity"
            value={formValues.quantity}
            onChange={handleChange}
            fullWidth
            type="number"
            required
          />
          <Autocomplete
            key={manufacturerKey}
            filterOptions={(options, params) => {
              const filtered = filter(options, params);

              const { inputValue } = params;
              const isExisting = options.some((option) =>
inputValue === option.name);
              if (inputValue !== '' && !isExisting) {
                filtered.push({
                  id: -1,
                  value: inputValue,
                  name: `${t('actions.add')} "${inputValue}"`,
                });
```

```
            }

            return filtered;
          }}
          loading={loading}
          options={manufacturers}
          getOptionLabel={(option) => option.name}
          onInputChange={handleSearch}
          onChange={handleChangeManufacturer}
          value={selectedManufacturer}
          renderInput={(params) => (
            <TextField {...params}
label={t('products.create.manufacturer')} variant="outlined"
fullWidth/>
          )}
        />
        <TextField
          label={t('products.create.article')}
          variant="outlined"
          name="article"
          value={formValues.article}
          onChange={handleChange}
          fullWidth
          required
        />
        <Button
          variant="contained"
          color="primary"
          onClick={(e) => handleSubmit(e, formValues)}
          fullWidth
        >
          {t(actionTranslate)}
        </Button>
      </Box>
    </Grid>
  );
};

export default ProductForm;

import React, { useEffect, useState } from 'react';
import {
  Box,
  Grid,
  TextField,
  Button,
  Pagination,
  Typography,
  Card,
  CardContent,
  CardActions,
```

```
  IconButton
} from '@mui/material';
import AddIcon from '@mui/icons-material/Add';
import EditIcon from '@mui/icons-material/Edit';
import DeleteIcon from '@mui/icons-material/Delete';
import {axiosProvider} from "../../axiosProvider";
import {productsResource} from "../../constants";
import {useNavigate} from "react-router-dom";
import {toast} from "sonner";
import Dialog from "../../Components/Dialog";
import {useTranslation} from "react-i18next";

const ProductList = () => {
  const {t} = useTranslation();
  const [stateDeleteDialog, setStateDeleteDialog] = useState({
    open: false,
    id: null,
  });
  const [products, setProducts] = useState([]);
  const [search, setSearch] = useState('');
  const [page, setPage] = useState(1);
  const [totalPages, setTotalPages] = useState(1);
  const navigate = useNavigate();
  const itemsPerPage = 10;

  // Fetch products from the server
  const fetchProducts = async () => {
    try {
      const {data} = await
axiosProvider.get(`/${productsResource}`, {
        params: { offset: (page - 1) * itemsPerPage, search,
limit: itemsPerPage},
      });

      setProducts(data.payload.products);
      const totalPages = Math.ceil(data.payload.totalCount /
itemsPerPage);
      setTotalPages(totalPages);
    } catch (error) {
      console.error('Error fetching products:', error);
    }
  };

  useEffect(() => {
    fetchProducts();
  }, [page, search]);

  const handleDelete = async () => {
    if (!stateDeleteDialog.id) {
      toast.error(t('products.list.error_msg_before_delete'));
      return;
```

```
    }

    try {
      await
axiosProvider.delete(`/${productsResource}/${stateDeleteDialog
.id}`);
      await fetchProducts();
      setStateDeleteDialog({open: false})
    } catch (e) {
      setStateDeleteDialog({open: false})
      toast.error(`Error deleting product:
${e.response?.data?.message || e.message}`);
    }
  };

  return (
    <>
      <Box sx={{color: '#fff', minHeight: '100vh' }}>
        <Typography variant="h4" mb={2}>
          {t('products.name')}
        </Typography>

        <Box mb={2} display="flex" justifyContent="space-
between">
          <TextField
            variant="outlined"
            placeholder={t('products.list.placeholder')}
            value={search}
            onChange={(e) => setSearch(e.target.value)}
            sx={{backgroundColor: '#2a2a2a', color: '#fff',
flex: 1, marginRight: 2}}
          />
          <IconButton
            variant="contained"
            sx={{backgroundColor: '#007bff', width: '56px',
borderRadius: '4px'}}
            onClick={() => navigate(`create`)}
          >
            <AddIcon/>
          </IconButton>
        </Box>

        <Grid container spacing={2}>
          {products.map((product) => (
            <Grid item xs={12} sm={6} md={4} key={product.id}>
              <Card sx={{ backgroundColor: '#2a2a2a', color:
'#fff' }}>
                <CardContent>
                  <Typography
variant="h6">{product.name}</Typography>
                  <Typography variant="body2" color="gray">
```

```
                    {product.description ||
t('products.list.empty_description')}
                  </Typography>
                  <Typography variant="body1" mt={1}>
                    {`${t('products.list.price')}:
${product.price}`}
                  </Typography>
                  <Typography
variant="body1">{`${t('products.list.quantity')}:
${product.quantity}`}</Typography>
                  <Typography
variant="body1">{`${t('products.list.article')}:
${product.article}`}</Typography>
                </CardContent>
                <CardActions>
                  <Button
                    startIcon={<EditIcon />}
                    color="primary"
                    onClick={() =>
navigate(`/${productsResource}/${product.id}`)}
                  >
                    {t('actions.update')}
                  </Button>
                  <Button
                    startIcon={<DeleteIcon />}
                    color="secondary"
                    onClick={() => setStateDeleteDialog({open:
true, id: product.id})}
                  >
                    {t('actions.delete')}
                  </Button>
                </CardActions>
              </Card>
            </Grid>
          ))}
        </Grid>

        <Box mt={3} display="flex" justifyContent="center">
          <Pagination
            count={totalPages}
            page={page}
            onChange={(event, value) => setPage(value)}
            sx={{
              button: { color: '#fff' },
              '& .MuiPaginationItem-root': { color: '#fff' },
            }}
          />
        </Box>
      </Box>
      <Dialog
        onClose={() => setStateDeleteDialog({open: false})}
```

```
        onConfirm={handleDelete}
        description="products.list.delete_desc"
        title="products.list.delete_title"
        open={stateDeleteDialog.open}
        confirmButtonText="actions.delete"
        cancelButtonText="actions.back"
      />
    </>
  );
};

export default ProductList;

import {BadRequestException, Injectable, NotFoundException,}
from '@nestjs/common';
import {CreateDeliveryDto} from './dto/create-delivery.dto';
import {InjectRepository} from '@nestjs/typeorm';
import {Brackets, Repository} from 'typeorm';
import {Delivery} from './entities/delivery.entity';
import {Product} from 'src/product/entities/product.entity';
import {DeliveryItem} from './entities/deliveryItem.entity';
import {User} from 'src/user/entities/user.entity';
import {DeliveryStatusEnum} from "./enum/delivery-
status.enum";
import {GetAllDeliveryDto} from "./dto/get-all-delivey.dto";
import {UpdateDeliveryDto} from "./dto/update-delivery.dto";

@Injectable()
export class DeliveryService {
  constructor(
    @InjectRepository(Delivery)
    private readonly deliveryRepository: Repository<Delivery>,
    @InjectRepository(Product)
    private readonly productRepository: Repository<Product>,
    @InjectRepository(DeliveryItem)
    private readonly deliveryItemRepository:
Repository<DeliveryItem>,
    @InjectRepository(User)
    private readonly userRepository: Repository<User>,
  ) {}

  async createDelivery(createDeliveryDto: CreateDeliveryDto) {
    const { userId, currency, deliveryItems } =
createDeliveryDto;

    let deliveryTotal = 0;

    const delivery = await this.deliveryRepository.save({
      status: DeliveryStatusEnum.Open,
      userId,
      currency,
```

```
    });

    const deliveryItemsPromises = deliveryItems.map(async
(deliveryItem) => {
      const deliveryItemPrice = deliveryItem.price *
deliveryItem.quantity;

      deliveryTotal += deliveryItemPrice;

      return this.deliveryItemRepository.save({
        ...deliveryItem,
        deliveryId: delivery.id,
        price: deliveryItem.price,
      });
    });

    await Promise.all([deliveryItemsPromises]);

    await this.deliveryRepository.update(delivery.id, {
      total: deliveryTotal,
    });

    return this.getDeliveryById(delivery.id);
  }

  async getAllDelivery(query: GetAllDeliveryDto) {
    const { search, limit = 10, offset = 0, sortOrder =
'DESC', sortBy = 'id' } = query;

    const queryBuilder =
this.deliveryRepository.createQueryBuilder('delivery');
    queryBuilder
      .leftJoinAndSelect('delivery.deliveryItems',
'deliveryItems')
      .leftJoinAndSelect('deliveryItems.product', 'product')

    if (search) {
      queryBuilder.andWhere(
        new Brackets((qb) => {
          qb.where('CAST(delivery.id AS TEXT) ILIKE :search',
{
            search: `%${search}%`,
          })
        }),
      );
    }

    const totalCount = await queryBuilder.getCount();
    let deliveries: Delivery[] = [];

    if (totalCount) {
```

```
      deliveries = await queryBuilder
        .skip(offset)
        .take(limit)
        .orderBy(`delivery.${sortBy}`, sortOrder)
        .getMany();
    }

    return {
      payload: {
        deliveries,
        totalCount,
      },
      status: 'success',
      message: 'Deliveries found',
    };
  }

  async updateDelivery(id: number, updateDto:
UpdateDeliveryDto) {
    const delivery = await this.getDeliveryById(id);

    if (delivery.status === DeliveryStatusEnum.Close) {
      throw new BadRequestException({
        message: `Delivery with id: ${id} is already closed`
      })
    }

    await Promise.all(delivery.deliveryItems.map(async
(deliveryItem) => {
      await
this.deliveryItemRepository.delete(deliveryItem.id);
    }))

    let deliveryTotal = 0;
    const deliveryItemsPromises =
updateDto.deliveryItems.map(async (deliveryItem) => {
      const deliveryItemPrice = deliveryItem.price *
deliveryItem.quantity;

      deliveryTotal += deliveryItemPrice;

      return this.deliveryItemRepository.save({
        ...deliveryItem,
        deliveryId: delivery.id,
        price: deliveryItem.price,
      });
    });

    await Promise.all([deliveryItemsPromises]);
```

```
    await this.deliveryRepository.update(id, {currency:
updateDto.currency, total: deliveryTotal});

    const updatedDelivery = await this.getDeliveryById(id);

    return {
      payload: updatedDelivery,
      status: 'success',
      message: `Delivery with id ${id} updated`,
    };
  }

  async closeDelivery(id: number) {
    const delivery = await this.getDeliveryById(id);

    if (delivery.status === DeliveryStatusEnum.Close) {
      throw new BadRequestException({
        message: `Delivery with id: ${id} is already closed`
      })
    }

    await this.deliveryRepository.update(id, {status:
DeliveryStatusEnum.Close});

    await Promise.all(delivery.deliveryItems.map(async (item)
=> {
      await this.productRepository.update(item.productId,
{quantity: item.product.quantity + item.quantity})
    }))
  }

  async getDeliveryById(id: number) {
    const delivery = await this.deliveryRepository.findOne({
      where: { id },
      relations: ['deliveryItems', 'deliveryItems.product'],
    });

    if (!delivery) {
      throw new NotFoundException({
        message: `Delivery with id ${id} is not found`,
      });
    }

    return delivery;
  }

  async deleteDelivery(id: number) {
    await this.getDeliveryById(id);

    return this.deliveryRepository.delete(id);
  }
```

```
}

import {
  BadRequestException,
  Injectable,
  NotFoundException,
} from '@nestjs/common';
import { CreateProductReqDto } from
'./dto/createProduct.req.dto';
import { UpdateProductReqDto } from
'./dto/updateProduct.req.dto';
import { InjectRepository } from '@nestjs/typeorm';
import { Product } from './entities/product.entity';
import {Brackets, Not, Repository} from 'typeorm';
import { ManufacturerService } from
'src/manufacturer/manufacturer.service';
import { GetAllProductsReqDto } from
'./dto/getAllProducts.req.dto';

@Injectable()
export class ProductService {
  constructor(
    @InjectRepository(Product)
    private readonly productRepository: Repository<Product>,
    private readonly manufacturerService: ManufacturerService,
  ) {}

  async createProduct(dto: CreateProductReqDto) {
    await
this.manufacturerService.getManufacturerById(dto.manufacturerI
d);

    const existProduct = await this.productRepository.exists({
      where: {article: dto.article}
    });

    if (existProduct) {
      throw new BadRequestException({
        message: `Product with article ${dto.article} is
already exist`,
      })
    }

    const product = await this.productRepository.save(dto);

    const { payload: createdProduct } = await
this.getProductById(product.id);

    return {
      payload: createdProduct,
      status: 'success',
```

```
      message: 'Product created',
    };
  }

  async getAllProducts(query: GetAllProductsReqDto) {
    const { search, limit, offset, sortOrder = 'DESC', sortBy
= 'id' } = query;

    const queryBuilder =
this.productRepository.createQueryBuilder('product');

    if (search) {
      queryBuilder.andWhere(
        new Brackets((qb) => {
          qb.where('product.name ILIKE :search', {
            search: `%${search}%`,
          }).orWhere('product.article ILIKE :search', {
            search: `%${search}%`,
          });
        }),
      );
    }

    const totalCount = await queryBuilder.getCount();
    let products: Product[] = [];

    if (totalCount) {
      products = await queryBuilder
        .skip(offset)
        .take(limit)
        .orderBy(sortBy, sortOrder)
        .getMany();
    }

    return {
      payload: {
        products,
        totalCount,
      },
      status: 'success',
      message: 'Products found',
    };
  }

  async getProductById(id: number) {
    const product = await this.productRepository.findOne({
      where: { id },
      relations: ['manufacturer'],
    });

    if (!product) {
```

```
      throw new NotFoundException({
        status: 'error',
        message: `Product with id ${id} not found`,
      });
    }

    return {
      payload: product,
      status: 'success',
      message: 'Product found',
    };
  }

  async updateProduct(id: number, dto: UpdateProductReqDto) {
    const { article } = dto;
    await this.getProductById(id);

    const isArticleExist = await
this.productRepository.exists({
      where: { article , id: Not(id)},
    });

    if (isArticleExist) {
      throw new BadRequestException({
        status: 'error',
        message: `Product with article: ${article} already
exist`,
      });
    }

    await this.productRepository.update(id, dto);

    const { payload: updatedProduct } = await
this.getProductById(id);

    return {
      payload: updatedProduct,
      status: 'success',
      message: `Product with id ${id} updated`,
    };
  }

  async deleteProduct(id: number) {
    await this.getProductById(id);

    await this.productRepository.delete(id);

    return {
      status: 'success',
      message: `Product with id ${id} deleted`,
    };
```

```
  }
}

import {BadRequestException, Injectable, NotFoundException,}
from '@nestjs/common';
import {CreateOrderDto, OrderItemDto} from './dto/create-
order.dto';
import {UpdateOrderDto} from './dto/update-order.dto';
import {InjectRepository} from '@nestjs/typeorm';
import {Brackets, Repository} from 'typeorm';
import {Order} from './entities/order.entity';
import {OrderItem} from './entities/orderItem.entity';
import {Product} from 'src/product/entities/product.entity';
import {GetAllOrdersDto} from "./dto/get-all-orders.dto";
import {OrderStatuses} from "./enums/orderStatuses.enum";

@Injectable()
export class OrderService {
  constructor(
    @InjectRepository(Order)
    private readonly orderRepository: Repository<Order>,
    @InjectRepository(OrderItem)
    private readonly orderItemRepository:
Repository<OrderItem>,
    @InjectRepository(Product)
    private readonly productRepository: Repository<Product>,
  ) {}

  async validateBeforeCreate(orderItems: OrderItemDto[]) {
    const quantityProductMapping = {};

    for (const orderItem of orderItems) {
      const product = await this.productRepository.findOne({
        where: { id: orderItem.productId },
        select: ['quantity', 'article'],
      });

      if (quantityProductMapping[product.article]) {
        quantityProductMapping[product.article].quantity =
product.quantity - orderItem.quantity;
        continue;
      }

      quantityProductMapping[product.article] = {quantity:
product.quantity - orderItem.quantity};
    }

    Object.keys(quantityProductMapping).map((article) => {
      if (quantityProductMapping[article].quantity < 0) {
        throw new BadRequestException(`There is product with
the ${article} article in stock than you specified`)
```

```
      }
    });

  }

  async create(createOrderDto: CreateOrderDto) {
    const { orderItems, userId } = createOrderDto;

    await this.validateBeforeCreate(orderItems);

    const order = await this.orderRepository.save({
      status: OrderStatuses.CREATED,
      userId,
    });

    let orderTotal = 0;

    const orderItemsPromises = orderItems.map(async
(orderItem) => {
      const product = await this.productRepository.findOne({
        where: { id: orderItem.productId },
        select: ['price', 'quantity', 'article', 'id'],
      });

      await this.productRepository.update(product.id,
{quantity: product.quantity - orderItem.quantity});

      const orderItemPrice = product.price *
orderItem.quantity;

      orderTotal += orderItemPrice;

      return this.orderItemRepository.save({
        ...orderItem,
        orderId: order.id,
        price: product.price * orderItem.quantity,
      });
    });

    await Promise.all(orderItemsPromises);

    await this.orderRepository.update(order.id, { total:
orderTotal });

    return this.findOne(order.id);
  }

  async findAll(query: GetAllOrdersDto) {
    const { search, limit = 10, offset = 0, sortOrder =
'DESC', sortBy = 'id' } = query;
```

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ДНІПРОВСЬКА ПОЛІТЕХНІКА»

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ТЕЛЕГРАМ-БОТУ СКЛАДСЬКОГО ОБЛІКУ ПІДПРИЕМСТВА

Текст програми
804.02070743.24004–01 12 01
Листів 19

2024

АНОТАЦІЯ

Розроблений Telegram-бот для складського обліку підприємства є інструментом автоматизації управління складськими операціями. Він забезпечує інтерактивну взаємодію між користувачами та системою складського обліку, надаючи зручний доступ до інформації про залишки товарів, операції з постачання та відвантаження.