НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ

І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ПОГОДЖЕНО	**ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ**
Декан факультету Інформаційних технологій	**Завідувач кафедри** Комп'ютерних систем, мереж та кібербезпеки
_____Ігор БОЛБОТ (підпис)	_____Дмитро КАСАТКІН (підпис)
«___»_____2026 р.	«___»_____2026 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

На тему: «Discord-бот для комп'ютерної системи автоматичного обліку замовлень та підтримки клієнтів»

Спеціальність F7 «Комп’ютерна інженерія »

Освітньо-професійна програма «Комп’ютерна інженерія »

Гарант освітньої програми канд. фіз.-мат. наук, доцент	_____ (підпис)	**Євгеній НІКІТЕНКО**
Керівник бакалаврської кваліфікаційної роботи канд. пед. наук, доцент	_____ (підпис)	**Дмитро КАСАТКІН**
Виконав	_____ (підпис)	**Антон СОБОЛЬ**

КИЇВ-2026

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ

І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ЗАТВЕРДЖУЮ
Завідувач кафедри
комп'ютерних систем, мереж та кібербезпеки
канд. пед. наук, доцент_____Дмитро КАСАТКІН
«_____»_____20____ р.

З А В Д А Н Н Я

ДО ВИКОНАННЯ БАКАЛАВРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ

ЗДОБУВАЧУ

Соболю Антону Олександровичу
(прізвище, ім'я, по батькові)

Спеціальність «Комп'ютерна інженерія »

Освітньо-професійна програма «Комп'ютерна інженерія »

Тема кваліфікаційної бакалаврської роботи
«Discord-бот для комп'ютерної системи автоматичного обліку замовлень та підтримки клієнтів»

затверджена наказом ректора НУБіП України від «10» 12 2025 р. № 3072 «С»
Термін подання завершеної роботи на кафедру «2» 06 2026 р.
Вихідні дані до бакалаврської кваліфікаційної роботи

Перелік питань, що підлягають дослідженню:

Перелік графічного матеріалу (за необхідності).

Дата видачі завдання "10" 12 2025 р.

Керівник бакалаврської кваліфікаційної роботи канд. пед. наук, доцент	_____ (підпис)	**Дмитро КАСАТКІН**
Завдання взяв до виконання	_____ (підпис)	**Антон СОБОЛЬ**

Зміст

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ 6
ВСТУП 9
РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИБІР ЗАСОБІВ РЕАЛІЗАЦІЇ 11
1.1 Огляд сучасного стану автоматизації електронної комерції та підтримки клієнтів 11
1.2 Аналіз вимог до комп'ютерної системи 12
1.3 Аналіз та вибір архітектурних рішень 14
1.4 Вибір засобів розробки програмного забезпечення 17
1.5 Вибір та обґрунтування системи керування базами даних 21
РОЗДІЛ 2. ПРОЕКТУВАННЯ КОМП'ЮТЕРНОЇ СИСТЕМИ 25
2.1 Концептуальна модель системи та ролі користувачів 25
2.2 Проектування структури бази даних 29
2.3 Моделювання поведінки системи 32
2.4 Проектування інтерфейсу взаємодії (UI/UX) 35
2.5 Забезпечення безпеки та надійності даних 39
РОЗДІЛ 3. РЕАЛІЗАЦІЯ КОМП'ЮТЕРНОЇ СИСТЕМИ 45
3.1 Розробка структури програмного проекту 45
3.2 Реалізація підключення та пулу з'єднань з базою даних 47
3.3 Розробка ядра бота та динамічного обробника команд 50
3.4 Програмування бізнес-логіки клієнтської частини 54
3.5 Програмування адміністративної панелі та системи підтримки 62
РОЗДІЛ 4. ТЕСТУВАННЯ ТА ДОСЛІДЖЕННЯ КОМП'ЮТЕРНОЇ СИСТЕМИ 68
4.1 Методика проведення тестування 68
4.2 Тестування клієнтського функціоналу 70
4.3 Тестування транзакцій та цілісності даних 75
4.4 Тестування адміністративної панелі та системи підтримки 77
4.5 Аналіз результатів тестування та оцінка продуктивності 81
ВИСНОВКИ 84
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 87
Додаток А скрипт заповнення БД 91
Додаток Б файл db.js 95
Додаток В файл index.js 96
Додаток Г команда /start 99
Додаток Ґ файл deploy-commands.js 100
Додаток Д команда /profile 102

Додаток Е /help 103
Додаток Є Команда /catalog 104
Додаток Ж /product 108
Додаток З Команда /search 110
Додаток Ї Команда /myorders 116
Додаток Й Команда /orderinfo 118
Додаток К Команда /cancelorder 121
Додаток Л Команда /support 124
Додаток М Команда /mytickets 126
Додаток Н Команда /ticket 128
Додаток О Команда /addproduct 131
Додаток П Команда /editproduct 133
Додаток Р Команда /orders 136
Додаток С Команда /setorderstatus 139
Додаток Т Команда /tickets 142
Додаток У Команда /replyticket 145
Додаток Ф Команда /stats 149
Додаток Х Команда /logs 150

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

БД - База даних
БР - Бакалаврська робота
ЕК - Екзаменаційна комісія
ПЗ - Програмне забезпечення (у контексті розробки) / Пояснювальна записка (у контексті документації)
ПП - Приватне повідомлення (Direct Message)
СКБД - Система керування базами даних
ТЗ - Технічне завдання
ACID - Atomicity, Consistency, Isolation, Durability (Атомарність, узгодженість, ізоляція, довговічність)
API - Application Programming Interface (Інтерфейс програмування застосунків)
CRUD - Create, Read, Update, Delete (Створення, читання, оновлення, видалення даних)
ER-діаграма - Entity-Relationship Diagram (Діаграма «сутність-зв'язок»)
I/O - Input/Output (Введення/виведення даних)
ID - Identifier (Унікальний ідентифікатор)
IDE - Integrated Development Environment (Інтегроване середовище розробки)
JSON - JavaScript Object Notation (Текстовий формат обміну даними)
SQL - Structured Query Language (Мова структурованих запитів)
UI - User Interface (Інтерфейс користувача)
UML - Unified Modeling Language (Уніфікована мова моделювання)
UX - User Experience (Користувацький досвід)

РЕФЕРАТ

Пояснювальна записка до дипломної роботи: 122 с., 21 рис., 21 табл., 22 додатки, 47 джерел.

ДИСКОРД-БОТ, ЕЛЕКТРОННА КОМЕРЦІЯ, NODE.JS, MYSQL, АВТОМАТИЗАЦІЯ БІЗНЕС-ПРОЦЕСІВ, КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА, БАЗА ДАНИХ, СИСТЕМА ПІДТРИМКИ.

Об’єкт автоматизації - процес обліку замовлень, взаємодії з клієнтами та супроводу звернень у рамках торгової або сервісної системи.

Мета роботи - створення програмної системи у вигляді Discord-бота, яка забезпечує автоматизацію прийому та обліку замовлень, перегляд каталогу товарів, взаємодію користувачів із системою через Discord, підтримку клієнтів через механізм звернень та адміністрування замовлень і товарів без необхідності вебпанелі на початковому етапі.

Методи дослідження - методи об'єктно-орієнтованого програмування, архітектурне моделювання поведінки системи за допомогою UML-діаграм , методи проектування та нормалізації реляційних баз даних, а також методи модульного та функціонального тестування програмного забезпечення за принципом «чорного ящика».

Отримані результати та їх новизна - спроектовано та програмно реалізовано комплексну клієнт-серверну інформаційну систему електронної комерції у середовищі месенджера Discord. Розроблено нормалізовану до третьої нормальної форми (3NF) реляційну базу даних на базі СКБД MySQL із забезпеченням суворої транзакційної цілісності та захистом від конкурентних конфліктів. Реалізовано дворівневий інтерфейс взаємодії на основі Slash-команд та інтерактивних Embed-повідомлень. Клієнтський модуль забезпечує фонову реєстрацію, пагіновану навігацію каталогом, оформлення замовлень та створення тікетів. Адміністративний модуль забезпечує виконання CRUD-операцій з товарним асортиментом, модерацію замовлень, управління системою підтримки клієнтів та ведення безперервного журналу аудиту (Audit Trail) дій персоналу.

Практичне значення - створений Discord-бот є завершеною, повністю функціональною системою, готовою до впровадження у реальні бізнес-процеси. Використання розробленої системи дозволяє підприємствам малого та середнього бізнесу організувати повноцінний канал продажів і технічної підтримки з мінімальними інфраструктурними та операційними витратами.

ВСТУП

Актуальність теми. Сучасний етап розвитку інформаційних технологій характеризується стрімким переходом електронної комерції у сферу месенджерів та соціальних платформ. Платформа Discord, маючи багатомільйонну активну аудиторію, перетворилася з інструменту для геймерів на потужну екосистему для побудови спільнот та ведення цифрового бізнесу. Для малого та середнього бізнесу розробка повноцінної веб-панелі та інтернет-магазину часто є економічно недоцільною через високі витрати на розробку, хостинг та підтримку інфраструктури. Натомість, автоматизація бізнес-процесів за допомогою спеціалізованих Discord-ботів дозволяє організувати повний цикл взаємодії з клієнтом: від перегляду каталогу до оформлення замовлення та отримання технічної підтримки в єдиному звичному для користувача інтерфейсі. Відсутність готових безкоштовних універсальних рішень, які б поєднували повноцінну реляційну базу даних, систему транзакцій та гнучку підтримку користувачів (тікети) в межах Discord, зумовлює високу актуальність даної роботи.

Мета роботи. Метою розробки є створення програмної системи у вигляді Discord-бота, яка забезпечує автоматизацію прийому та обліку замовлень, перегляд каталогу товарів, взаємодію користувачів із системою через Discord, підтримку клієнтів через механізм звернень та адміністрування замовлень і товарів без необхідності веб-панелі на початковому етапі.

Завдання дослідження. Для досягнення поставленої мети необхідно вирішити такі завдання:

1. Проаналізувати предметну область та обґрунтувати вибір засобів реалізації клієнт-серверної архітектури (Node.js, бібліотека discord.js, реляційна СУБД MySQL).
2. Спроектувати концептуальну модель системи, виокремити категорії користувачів (клієнт, адміністратор, суперкористувач) та визначити їхні права доступу.
3. Розробити реляційну базу даних для надійного збереження інформації про користувачів, товари, замовлення, звернення та журнали дій.
4. Програмно реалізувати клієнтський функціонал: реєстрацію, перегляд каталогу з пагінацією, оформлення замовлення (через SQL-транзакції) та створення звернень.
5. Програмно реалізувати адміністративну панель: управління каталогом, зміну статусів замовлень, відповіді на тікети та збір загальної статистики.

6. Провести тестування програмного продукту на предмет надійності, стійкості до відмов та коректності обробки виняткових ситуацій (наприклад, недостатня кількість товару на складі).

Об’єкт автоматизації. Об’єктом автоматизації є процес обліку замовлень, взаємодії з клієнтами та супроводу звернень у рамках торгової або сервісної системи.

Предмет розробки. Предметом розробки є серверний програмний застосунок, інтегрований із платформою Discord, що працює через Discord API та базу даних.

Практичне значення отриманих результатів. Розроблений програмний продукт є повністю функціональною інформаційною системою, готовою до впровадження в реальні бізнес-процеси. Система забезпечує єдиний канал взаємодії між клієнтом та адміністрацією через Discord-сервер, що суттєво зменшує фінансові витрати на супровід традиційних інтернет-ресурсів. Завдяки використанню транзакцій гарантується цілісність фінансових та складських даних, а накопичена в базі даних інформація створює підґрунтя для подальшого аналізу, звітності та масштабування системи, включно з можливою інтеграцією з повноцінною веб-панеллю в майбутньому.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИБІР ЗАСОБІВ РЕАЛІЗАЦІЇ

1.1 Огляд сучасного стану автоматизації електронної комерції та підтримки клієнтів

Сучасний ринок електронної комерції характеризується перенасиченням пропозицій, що змушує бізнес шукати нові шляхи оптимізації взаємодії з клієнтом. Автоматизація процесів стає не просто перевагою, а необхідністю для виживання торгової системи. Як зазначається у фаховій літературі, «автоматизація бізнес-процесів у комерційній діяльності дозволяє мінімізувати вплив людського фактору, знизити операційні витрати та забезпечити цілодобову підтримку користувачів без залучення значного штату персоналу» [1].

Традиційним рішенням для e-commerce є веб-сайт або складний мобільний застосунок. Проте для невеликих нішевих магазинів, сервісних центрів або цифрових спільнот розгортання повноцінної веб-інфраструктури часто є надлишковим та фінансово обтяжливим. У цьому контексті особливого значення набувають інтерфейси на базі месенджерів, зокрема Discord.

Аналізуючи ринок месенджерів, можна виділити кілька ключових причин, чому Discord-боти стають дієвою альтернативою класичним сайтам для певних ніш:

1. **Низький поріг входження та утримання користувача.** Клієнту не потрібно переходити на сторонні ресурси, реєструватися в нових формах або встановлювати додаткове ПЗ. Взаємодія відбувається у звичному середовищі, де користувач вже проводить значну частину часу.
2. **Інтегрованість у систему спільнот.** Discord дозволяє об’єднувати торговий майданчик з форумом для спілкування. «Ефективність комерційних систем у месенджерах базується на принципі "social commerce", де процес купівлі нерозривно пов’язаний із соціальною взаємодією всередині групи» [2].
3. **Економічна доцільність.** Розробка та підтримка бота потребує значно менше ресурсів, ніж створення адаптивного веб-сайту з особистим кабінетом, оскільки Discord надає готовий UI-фреймворк (кнопки, меню, вікна), систему авторизації та захищені сервери [3].

Безумовно, класичний веб-додаток залишається більш функціональним і зручним для масштабних проєктів завдяки гнучкості інтерфейсу та можливостям SEO-просування. Однак для проєктів, де важливою є пряма, миттєва комунікація адміністрації з клієнтом, Discord-бот виграє за рахунок нативності. Важливою

частиною такої системи є служба підтримки. Традиційні методи через e-mail або телефонні дзвінки втрачають актуальність у гейміфікованих або IT-спільнотах. «Автоматизована система тікетів (support tickets) дозволяє структурувати звернення, зберігати історію листування та оперативно реагувати на запити, що підвищує лояльність клієнтів на 30-40%» [4].

Використання Discord-бота для автоматичного обліку замовлень та підтримки клієнтів є обґрунтованим рішенням для автоматизації процесів у рамках торгової або сервісної системи, оскільки він дозволяє реалізувати:

- централізований прийом замовлень безпосередньо в чаті;
- автоматичне оновлення статусів у реальному часі ;
- прозору систему підтримки через механізм звернень ;
- зберігання даних у реляційній базі для подальшої звітності.

Таким чином, Discord-бот у цій роботі розглядається не як повна заміна веб-технологій, а як спеціалізований інструмент для нішевого бізнесу, що дозволяє досягти мети автоматизації за мінімальних інфраструктурних витрат.

1.2 Аналіз вимог до комп'ютерної системи

Ефективність будь-якої комп'ютерної системи визначається її відповідністю початково проаналізованим вимогам та чітким розподілом функціональних можливостей між різними категоріями користувачів. Згідно з метою розробки, цільова система призначена для організації єдиного каналу взаємодії між клієнтами та адміністрацією через Discord-сервер. Вона є класичною клієнт-серверною інформаційною системою, що функціонує за допомогою бота та реляційної бази даних .

Декомпозиція системи вимагає її поділу на дві основні логічні частини: клієнтську та адміністративну. Цей розподіл обумовлений необхідністю ізоляції бізнес-логіки управління системою від кінцевого споживача.

Вимоги до клієнтської частини системи

Клієнтська частина орієнтована на пересічного користувача Discord, який взаємодіє з ботом для придбання товарів або отримання консультацій . Основний акцент тут робиться на простоту, швидкість реакції та зручність форматування інформації. Система повинна забезпечити такий мінімальний функціонал для клієнта:

- **Автоматична реєстрація.** При першому зверненні (команда /start) система повинна перевірити наявність Discord ID користувача в базі даних та, за його відсутності, створити новий запис для подальшого відстеження активності .
- **Довідковий центр.** Надання інформації про доступні команди та порядок роботи через команду /help .
- **Взаємодія з каталогом.** Користувач повинен мати змогу переглядати список усіх активних товарів (/catalog), здійснювати пошук за назвою (/search) та отримувати розширену інформацію про конкретну позицію, включаючи ціну та кількість на складі (/product) .
- **Оформлення та облік замовлень.** Критичною вимогою є можливість формування замовлення (/order) із вказанням ID товару та кількості . Користувач має право переглядати власні замовлення (/myorders) та детальний чек конкретної операції (/orderinfo) . Також передбачено механізм скасування замовлення (/cancelorder), але виключно до моменту його переходу у стан обробки .
- **Система підтримки.** Клієнт може ініціювати створення тікета (/support) з описом проблеми , а також переглядати історію листування та статус звернень (/mytickets, /ticket) .

Вимоги до адміністративної частини системи

Адміністративний модуль призначений для користувачів із підвищеними правами і є основним інструментом керування життєвим циклом магазину . Доступ до нього має бути жорстко обмежений на рівні інтерфейсу Discord, щоб запобігти виконанню критичних дій звичайними користувачами .

Функціональні вимоги до адміністратора включають:

- **Управління товарним асортиментом.** Можливість додавати нові позиції (/addproduct) та редагувати існуючі, зокрема змінювати ціну або залишок на складі (/editproduct) .
- **Модерація замовлень.** Адміністратор повинен мати змогу переглядати всі замовлення з можливістю фільтрації за статусом (/orders) та змінювати поточний етап обробки замовлення (/setorderstatus) від нового до завершеного чи скасованого .
- **Обробка звернень.** Перегляд усіх тікетів (/tickets), сортування за станом вирішення проблеми та надання зворотного зв'язку клієнтам (/replyticket) із можливістю подальшого закриття тікета .

- **Аудит та аналітика.** Система повинна вести журнал подій (/logs), реєструючи додавання товарів, зміну статусів та відповіді на звернення . Також передбачено команду для отримання загальної статистики роботи системи (/stats), що відображає кількість користувачів, замовлень та активних товарів .

Як зауважується у спеціалізованій літературі, «надійність клієнт-серверної системи безпосередньо залежить від її здатності коректно обробляти виняткові ситуації на рівні серверної логіки» [5]. Тому нефункціональні вимоги до розроблюваної системи передбачають стабільну роботу при неіснуючих ID товарів, недостатній кількості на складі та помилках підключення до бази даних .

Крім того, інтерфейс бота повинен базуватися на сучасних Slash-командах та форматувати відповіді за допомогою Discord Embed-повідомлень, що забезпечить високу читабельність тексту та структурування вихідної інформації . Успішна реалізація вищезазначених вимог є передумовою для створення повноцінної системи автоматизації.

1.3 Аналіз та вибір архітектурних рішень

Проєктування комп'ютерної системи автоматичного обліку замовлень вимагає чіткого визначення архітектурної моделі, яка забезпечить стабільність, масштабованість та ефективну взаємодію між користувачем і сервером. Згідно з характеристиками системи, вона реалізується як клієнт-серверна інформаційна система. Основним вузлом є серверний застосунок, який обробляє запити, взаємодіє з базою даних та платформою Discord через відповідний API.

Обґрунтування клієнт-серверної архітектури

Вибір клієнт-серверної архітектури обумовлений необхідністю централізованого зберігання даних та забезпечення доступу до них багатьох користувачів одночасно. У цій моделі Discord виступає у ролі «тонкого клієнта», який надає інтерфейс користувача, а вся бізнес-логіка та обробка даних зосереджена на сервері Node.js .

- **Порівняння з монолітною локальною архітектурою.** Монолітний підхід передбачає встановлення програми безпосередньо на пристрій кожного користувача з власною копією бази даних. Для торгової системи це є неприпустимим, оскільки відсутність синхронізації між клієнтами призведе до розбіжностей у залишках товарів на складі та неможливості ведення загального обліку.

- **Порівняння з архітектурою Serverless (FaaS).** Безсерверні рішення (наприклад, AWS Lambda) дозволяють виконувати код лише у відповідь на події. Хоча це економічно для малих навантажень, для Discord-бота, що вимагає постійного з’єднання через WebSocket (Gateway) для швидкої реакції на команди, такі рішення є технічно складними в реалізації та можуть спричиняти затримки («холодний старт»), що негативно впливає на UX користувача.

Взаємодія через Discord API: Webhooks vs Gateway

Для обміну даними з Discord існують два основні методи: Webhooks та Gateway API. У межах розробки обрано Gateway API, оскільки він забезпечує постійне двостороннє з’єднання в реальному часі.

- **Порівняння з використанням суто Webhooks.** Вебхуки є одностороннім механізмом, призначеним для надсилання повідомлень у канали без необхідності авторизації бота як повноцінного користувача. Вони ідеально підходять для логування або сповіщень, але не дозволяють реалізувати складні Slash-команди, обробляти натискання кнопок у Embed-повідомленнях або відстежувати статуси користувачів, що є критичним для повноцінної системи підтримки та замовлень.
- **Порівняння з методом Long Polling через REST API.** Даний метод передбачає періодичні запити сервера до Discord для перевірки наявності нових повідомлень. Це створює надлишкове навантаження на мережу та сервер, а також значні затримки у відповіді бота. Gateway API на базі WebSocket, навпаки, миттєво доставляє події серверу, що робить взаємодію з клієнтом максимально плавною.

Вибір платформи та бібліотеки розробки

Для реалізації серверної логіки обрано платформу Node.js та бібліотеку discord.js . Це рішення базується на високій продуктивності рушія V8 та асинхронній моделі I/O, що критично для ботів, які обробляють велику кількість паралельних запитів від користувачів .

- **Порівняння з мовою Python (бібліотека discord.py).** Python є популярним вибором завдяки простоті синтаксису. Проте Node.js демонструє кращу продуктивність у завданнях, пов'язаних із великою кількістю вхідних/вихідних операцій (I/O bound). Крім того, використання JavaScript дозволяє в майбутньому легко інтегрувати серверну частину з вебпанеллю, оскільки обидва компоненти будуть написані на одній мові.

- **Порівняння з мовою Go (бібліотека discordgo).** Мова Go забезпечує надзвичайно високу швидкість виконання та багатопотоковість. Однак її використання потребує складнішого процесу розробки та супроводу («low-level control»). Для завдань автоматизації обліку замовлень, де важлива швидкість написання та гнучкість бізнес-логіки, Node.js є більш збалансованим вибором .

Обрана архітектура дозволяє створити гнучку, надійну систему, яка відповідає всім функціональним вимогам і готова до подальшого масштабування та впровадження додаткових модулів .

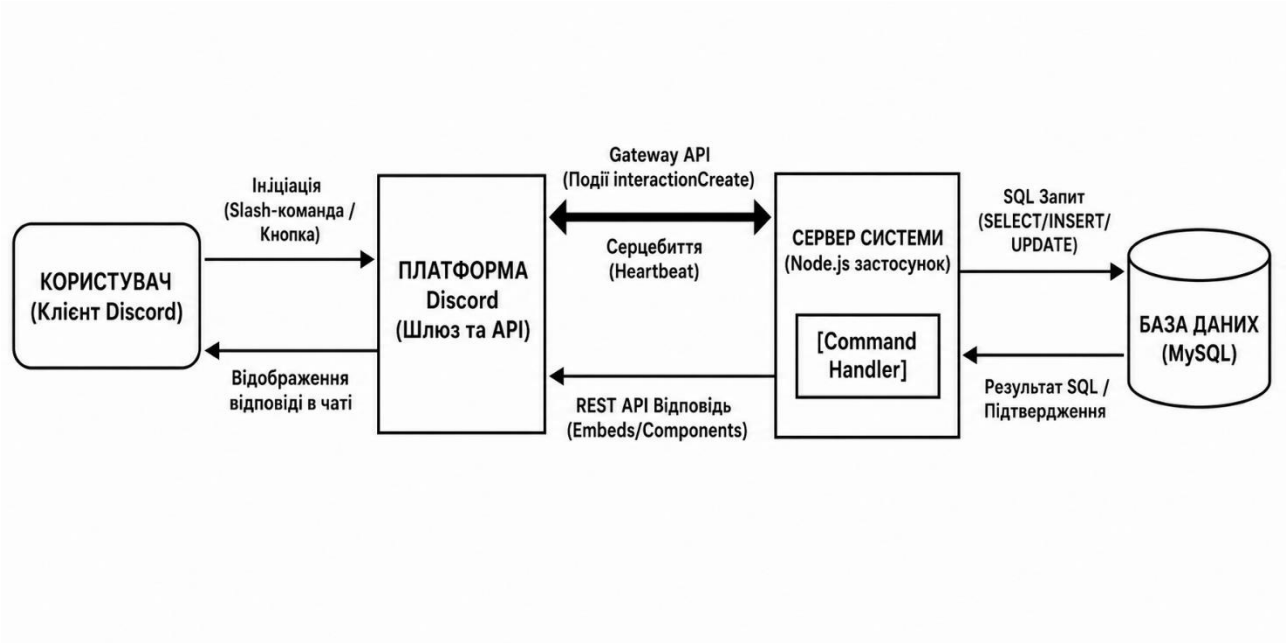

Рисунок 1.1 - Схема клієнт-серверної взаємодії системи через Discord API

Таблиця 1.1 - Порівняльна характеристика протоколів взаємодії

Критерій порівняння	**Discord Gateway (WebSocket)**	**Webhooks**	**REST API (Long Polling)**
Тип з'єднання	Постійне, стійке з'єднання (Keep-alive)	Одноразові HTTP-запити (Stateless)	Періодичні HTTP-запити
Напрямок передачі даних	Двосторонній (Full-duplex)	Односторонній (Сервер -> Discord)	Двосторонній (імітований)
Швидкість реакції на події	Миттєва (Real-time)	Висока (лише для відправки)	Низька (залежить від інтервалу опитування)

Підтримка інтерактивних компонентів (кнопки, меню)	Повна підтримка	Не підтримується (неможливо відстежити натискання)	Підтримується, але з великими затримками
Підтримка Slash-команд	Повна підтримка	Не підтримується	Підтримується
Навантаження на мережу та сервер	Мінімальне (дані передаються лише при настанні події)	Мінімальне	Високе (через постійні «пусті» запити перевірки)
Складність реалізації	Висока (потребує керування станами з'єднання та серцебиттям)	Низька	Середня
Сфера застосування в проєкті	**Основний механізм (ядро бота)**	Альтернативне логування подій	Недоцільно використовувати

1.4 Вибір засобів розробки програмного забезпечення

Розробка надійного та масштабованого програмного забезпечення вимагає ретельного підбору технологічного стеку. Згідно з вимогами до сучасних інтерактивних систем, середовище виконання та супутні бібліотеки повинні забезпечувати асинхронну обробку подій, високу швидкість I/O (введення-виведення) операцій та простоту розгортання [6]. Враховуючи, що система повинна підтримувати постійне з’єднання з Discord Gateway та одночасно опрацьовувати запити до бази даних, вибір базової платформи є критичним етапом проектування.

Обґрунтування вибору середовища Node.js

Як основне середовище виконання серверного коду було обрано платформу Node.js. Цей вибір обґрунтовується тим, що Node.js базується на високопродуктивному рушії V8 від Google та використовує подієво-орієнтовану, неблокуючу модель введення-виведення. Це дозволяє ефективно обробляти тисячі одночасних з'єднань в одному потоці, що ідеально підходить для архітектури чат-ботів . Для підтвердження доцільності цього рішення було проведено аналіз альтернативних середовищ виконання:

- **Порівняння з мовою Python (інтерпретатор CPython).** Python широко використовується для створення ботів завдяки низькому порогу входження та високій читабельності коду. Однак базова реалізація Python має глобальне блокування інтерпретатора (GIL), що суттєво обмежує паралельне виконання потоків. Хоча використання модуля asyncio частково нівелює цю проблему, Node.js є нативно асинхронним із коробки. У задачах, що пов’язані з інтенсивним мережевим обміном даними (I/O bound tasks), Node.js демонструє значно вищу пропускну здатність та менші затримки порівняно з Python [7].
- **Порівняння з платформою Java (Java Virtual Machine).** Java є стандартом для розробки складних Enterprise-рішень завдяки суворій типізації, багатопотоковості та високій стабільності. Проте для розробки Discord-бота використання Java призведе до значного надлишку (overhead) у споживанні оперативної пам'яті. Крім того, Node.js забезпечує швидший цикл розробки (Rapid Application Development) та має легшу інфраструктуру, що дозволяє запустити та масштабувати мікросервіс із меншими апаратними витратами.

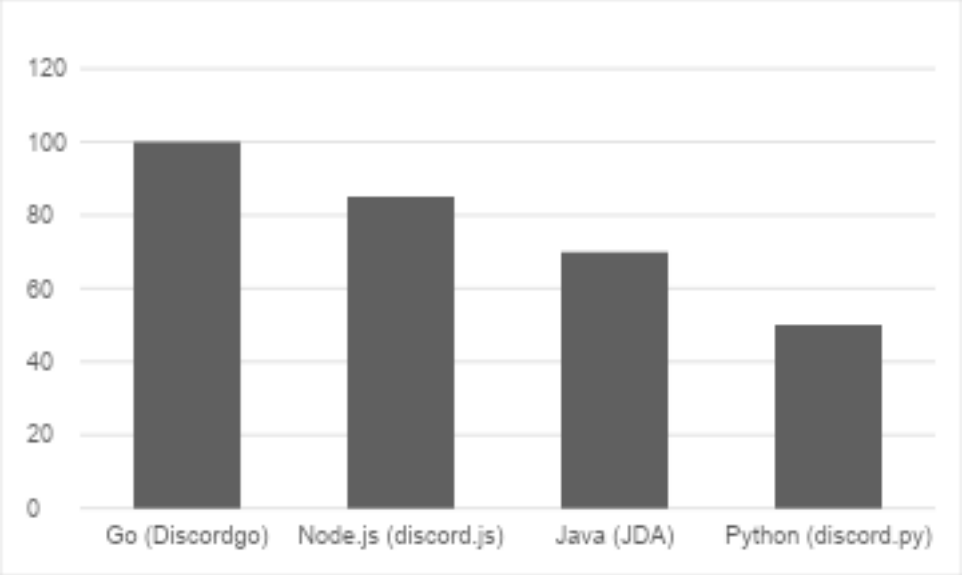

Рисунок 1.2 - Порівняння продуктивності середовищ виконання при обробці WebSocket-з'єднань

Вибір бібліотеки для взаємодії з Discord API

Взаємодія з серверами Discord потребує використання спеціалізованої обгортки (фреймворку) над базовим HTTP/WebSocket API. У якості такої бібліотеки було жорстко визначено discord.js. Ця бібліотека є повністю об'єктно-орієнтованою, підтримує найсучасніші функції платформи (Slash-команди, компоненти

повідомлень, модальні вікна) і має одну з найактивніших екосистем розробників . Альтернативні рішення також були ретельно проаналізовані:

- **Порівняння з бібліотекою discord.py (Python).** Довгий час discord.py була головним конкурентом discord.js. Проте у 2021 році розробка discord.py була тимчасово зупинена її творцем через радикальні зміни в архітектурі Discord API (перехід на Message Content Intents та Slash-команди). Це спричинило кризу в тисячах існуючих проєктів. Бібліотека discord.js в цей же час оперативно інтегрувала всі оновлення, продемонструвавши вищу інституційну стійкість та готовність екосистеми до масштабних змін з боку платформи [8].
- **Порівняння з бібліотекою Discordgo (мова Go).** Discordgo є надзвичайно швидкою та ресурсоефективною обгорткою завдяки компільованій природі мови Go. Але її суттєвим недоліком є менша база готових рішень, слабша документація (порівняно з discord.js) та необхідність написання великої кількості шаблонного (boilerplate) коду для реалізації базового функціоналу. Для системи автоматичного обліку замовлень, де гнучкість бізнес-логіки превалює над економією мікросекунд процесорного часу, discord.js є раціональнішим вибором.

Таблиця 1.2 - Порівняльний аналіз бібліотек для роботи з Discord API

Характеристика	**discord.js (Node.js)**	**discord.py (Python)**	**Discordgo (Go)**	**JDA (Java)**
Модель виконання (Продуктивність)	Асинхронна, подієво-орієнтована (висока)	Синхронна / Асинхронна (середня)	Багатопотокова (дуже висока)	Багатопотокова (висока)
Споживання оперативної пам'яті	Середнє	Середнє	Низьке	Високе (вплив середовища JVM)

Підтримка оновлень Discord API	Оперативна, першочерго ва	Затримуєтьс я (через попередню кризу розробки)	Повільна (слабша підтримка спільнотою)	Стабільна, але консервативн а
Рівень абстракції	Високий (суворе ООП)	Високий (ООП)	Низькорівневи й (ближче до REST)	Високий (суворе ООП)
Наявність готових модулів	Максимальн а	Велика	Обмежена	Середня
Складність розгортання та супроводу	Низька	Низька	Середня	Висока

Вибір допоміжних програмних засобів

Для забезпечення повного циклу розробки та супроводу програмного продукту було обрано низку допоміжних інструментів, рекомендованих методологією сучасної розробки .

1. **Середовище розробки (IDE):** У якості основного редактора коду застосовано Visual Studio Code (VS Code) від Microsoft . Вибір обґрунтовано вбудованою підтримкою середовища Node.js, гнучкою системою налагодження (дебагінгу), інтелектуальним автодоповненням коду IntelliSense та наявною інтеграцією з системами контролю версій (Git) .
2. **Пакетний менеджер:** Управління залежностями здійснюється за допомогою стандартного пакетного менеджера npm, який постачається разом з Node.js . Це дозволяє зручно керувати версіями бібліотек та скриптами запуску.
3. **Додаткові модулі (Middleware):** * Бібліотека dotenv використовується для завантаження змінних середовища з файлу .env. Цей інструмент критично важливий для дотримання вимог безпеки, оскільки дозволяє приховати секретний токен бота та облікові дані бази даних від потрапляння у відкритий код .

Утиліта nodemon обрана для оптимізації процесу розробки. Вона автоматично відстежує зміни у вихідних файлах та перезапускає сервер, що значно прискорює тестування нового функціоналу.
Проведений аналіз дозволяє зробити висновок, що стек технологій "Node.js + discord.js" є оптимальним рішенням. Він повністю задовольняє вимоги до надійності, супровідності та масштабованості системи , гарантуючи при цьому мінімальні витрати часу на розгортання архітектури .

1.5 Вибір та обґрунтування системи керування базами даних

Важливою складовою частиною будь-якої інформаційної системи є база даних (БД), яка відповідає за надійне зберігання, структурування та швидкий доступ до інформації. Оскільки розроблювана система автоматизації обліку замовлень передбачає роботу з великою кількістю взаємопов’язаних сутностей (користувачі, категорії, товари, замовлення, тікети), вибір системи керування базами даних (СКБД) є фундаментальним рішенням, що визначає стабільність усього проєкту. Згідно з характеристиками системи, вона побудована за клієнт-серверною архітектурою, де серверна логіка на базі Node.js повинна забезпечувати цілісність даних при кожній операції.

Порівняльний аналіз реляційних та нереляційних моделей
На етапі проєктування було розглянуто два основні підходи до організації даних: реляційний (SQL) та нереляційний (NoSQL).

- **NoSQL рішення (наприклад, MongoDB).** Нереляційні бази даних пропонують високу гнучкість схеми (schema-less) та горизонтальне масштабування. Проте головним недоліком NoSQL для торгових систем є відсутність жорстких механізмів перевірки зв'язків та складність реалізації складних транзакцій, що охоплюють кілька колекцій. «Для систем, де фінансова точність та логічна цілісність є пріоритетом, використання NoSQL може призвести до появи "сміттєвих" або несинхронізованих даних» [9].
- **Реляційні рішення (SQL).** Реляційна модель базується на математичній теорії множин і забезпечує чітке структурування даних у таблицях із визначеними зв'язками. Це дозволяє використовувати механізми зовнішніх ключів (Foreign Keys) для запобігання видаленню записів, від яких залежать інші об'єкти (наприклад, неможливість видалити товар, який вже є у замовленні).

Таблиця 1.3 - Порівняння СКБД за критеріями надійності та цілісності

Критерій порівняння	**MySQL (InnoDB)**	**PostgreSQL**	**MongoDB (NoSQL)**
Тип моделі даних	Реляційна (таблична)	Об’єктно-реляційна	Документо-орієнтована (BSON)
Підтримка ACID	Повна підтримка транзакцій	Повна (найсуворіша)	Обмежена (залежить від версії)
Цілісність зв’язків (Foreign Keys)	Жорстка перевірка на рівні ядра	Жорстка перевірка	Відсутня (контролюється кодом)
Складність транзакцій	Висока (підтримка COMMIT/ROLLBACK)	Дуже висока	Середня (переважно атомарність документів)
Гнучкість схеми даних	Низька (потребує фіксованої структури)	Низька	Дуже висока (Dynamic Schema)
Механізм блокувань	Блокування на рівні рядка (FOR UPDATE)	Блокування на рівні рядка	Блокування документа
Оптимальність для обліку замовлень	**Висока (найкращий баланс швидкості/надійності)**	Висока	Низька (ризик втрати цілісності зв’язків)

Обґрунтування вибору MySQL

Серед реляційних СКБД для реалізації проєкту було обрано **MySQL**. Це рішення ґрунтується на тому, що архітектура системи вимагає забезпечення суворої цілісності зв’язків між ключовими сутностями. MySQL є класичним інструментом, що гарантує надійне зберігання транзакційних даних та безпроблемно інтегрується з Node.js.

Основними аргументами на користь MySQL є:

1. **Підтримка механізму ACID.** Абревіатура ACID (Atomicity, Consistency, Isolation, Durability) описує набір властивостей, що гарантують надійність обробки транзакцій. Для торгової системи це є критичним: операція

замовлення або повинна виконатися повністю (зменшення залишку на складі + створення запису в таблиці orders), або не виконатися взагалі, якщо на будь-якому етапі виникла помилка .

2. **Продуктивність та поширеність.** MySQL оптимізована для операцій читання та запису, що часто зустрічаються в e-commerce системах. Вона підтримує пул з'єднань, що дозволяє боту обробляти багато запитів одночасно без перевантаження сервера .
3. **Інструментарій.** Використання phpMyAdmin дозволяє візуально контролювати структуру таблиць, виконувати налагоджувальні SQL-запити та створювати резервні копії (дампи) бази даних для передачі замовнику.

Забезпечення цілісності фінансових транзакцій

Особлива увага в роботі приділяється реалізації транзакцій при оформленні замовлення (/order). У цей момент система взаємодіє одночасно з декількома таблицями: users, products, orders та order_items . Використання MySQL дозволяє ініціювати транзакцію (BEGIN TRANSACTION), під час якої перевіряється наявність товару та його ціна. Якщо дані коректні, система фіксує зміни (COMMIT). У разі виникнення збою або виявлення недостатньої кількості одиниць товару на складі, виконується відкат (ROLLBACK), що повертає базу даних до початкового стабільного стану .

«Цілісність даних у реляційних СКБД забезпечується не лише програмним кодом, а й внутрішніми правилами бази: обмеженнями унікальності (UNIQUE), перевірками на порожнечу (NOT NULL) та каскадними діями (ON DELETE CASCADE)» [10]. Всі ці інструменти були впроваджені в архітектуру системи ServiceBridge для автоматичного обліку та аудиту дій .

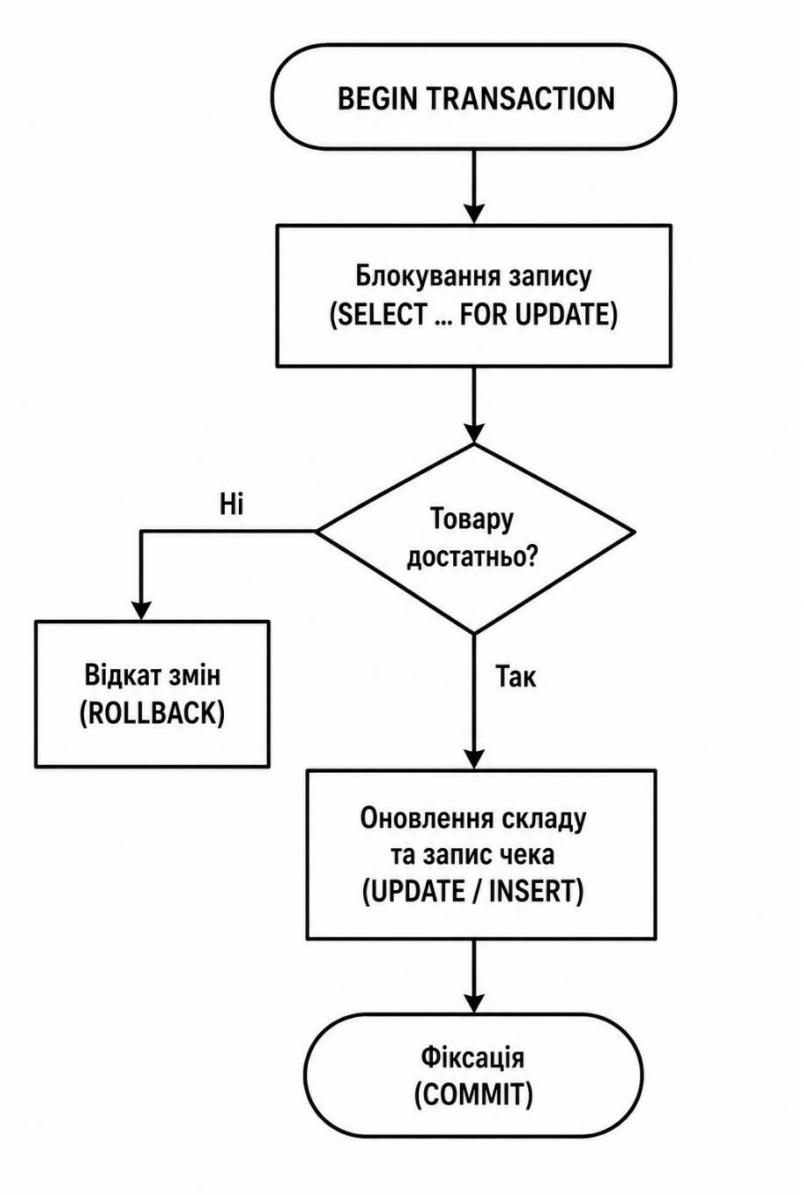

Рисунок 1.3 - Діаграма проходження транзакції в MySQL при оформленні замовлення

Вибрана СКБД MySQL у поєднанні з бібліотекою mysql2 є оптимальним інженерним рішенням, яке повністю відповідає вимогам безпеки та надійності, що були визначені на етапі аналізу предметної області.

РОЗДІЛ 2. ПРОЕКТУВАННЯ КОМП'ЮТЕРНОЇ СИСТЕМИ

2.1 Концептуальна модель системи та ролі користувачів

Проєктування комп’ютерної системи автоматичного обліку замовлень розпочинається з розробки концептуальної моделі, яка визначає архітектурний скелет взаємодії між зовнішніми суб’єктами та внутрішніми процесами бота.

Концептуальна модель системи базується на ієрархічному розподілі доступів та функціональних обов’язків, що критично важливо для забезпечення безпеки та цілісності даних у середовищі Discord .

Центральним елементом моделі є взаємодія акторів із програмним інтерфейсом через систему Slash-команд та інтерактивних компонентів (кнопок, Embed-повідомлень) . Згідно з архітектурними вимогами, у системі жорстко розмежовано три категорії користувачів: Клієнт, Адміністратор та Суперкористувач.

Опис акторів системи та їхніх функціональних ролей

1. **Клієнт (Client).** Це основний користувач системи, який взаємодіє з ботом для отримання послуг або придбання товарів . Концептуально роль клієнта є пасивно-виконавчою: він ініціює запити, на які система відповідає вибіркою з бази даних .

Функції клієнта: реєстрація в базі даних (команда /start), перегляд асортименту товарів з використанням пагінації (/catalog), детальний пошук позицій (/search, /product), формування замовлень (/order), перегляд історії власних операцій (/myorders) та взаємодія зі службою підтримки через створення тікетів (/support) .

2. **Адміністратор (Administrator).** Це користувач із підвищеним рівнем довіри, відповідальний за операційну діяльність торгового майданчика . У концептуальній моделі адміністратор виступає контролюючою ланкою, яка має доступ до модифікації складських залишків та життєвого циклу замовлень .

Функції адміністратора: наповнення каталогу (/addproduct), редагування цінових та кількісних показників товарів (/editproduct), модерація замовлень клієнтів (/orders, /setorderstatus), відповіді на звернення користувачів у системі підтримки (/tickets, /replyticket) та аналіз логів системи .

3. **Суперкористувач (Superuser).** Роль власника системи або технічного директора . Його головним завданням є управління складом адміністративної групи та виконання дій, що мають критичний вплив на роботу програмного комплексу в цілому .

Функції суперкористувача: додавання та видалення адміністраторів (/addadmin, /removeadmin), зміна рівнів доступу та перегляд розширеної статистики продуктивності системи .

Механізм розмежування прав доступу. Для надійної реалізації концептуальної моделі застосовується гібридний метод перевірки прав. Обмеження доступу реалізується як на рівні Discord API (використання PermissionFlagsBits.Administrator), так і на рівні внутрішньої логіки сервера Node.js, де перед виконанням кожної команди система звіряє Discord ID ініціатора з таблицею admins у базі даних MySQL . Це гарантує, що критичні команди, такі як зміна статусу замовлення або видалення товару, будуть недоступні для звичайних користувачів навіть за спроби прямого виклику функції .

Візуалізація моделі. Згідно з методичними вказівками, для повного розкриття архітектури проєктування на даному етапі необхідно додати UML-діаграму варіантів використання (Use Case Diagram), яка наочно продемонструє зв'язки між усіма акторами та ключовими функціями бота. Також доцільно включити таблицю порівняння функціональних можливостей ролей для підкріплення розділу про розмежування прав .

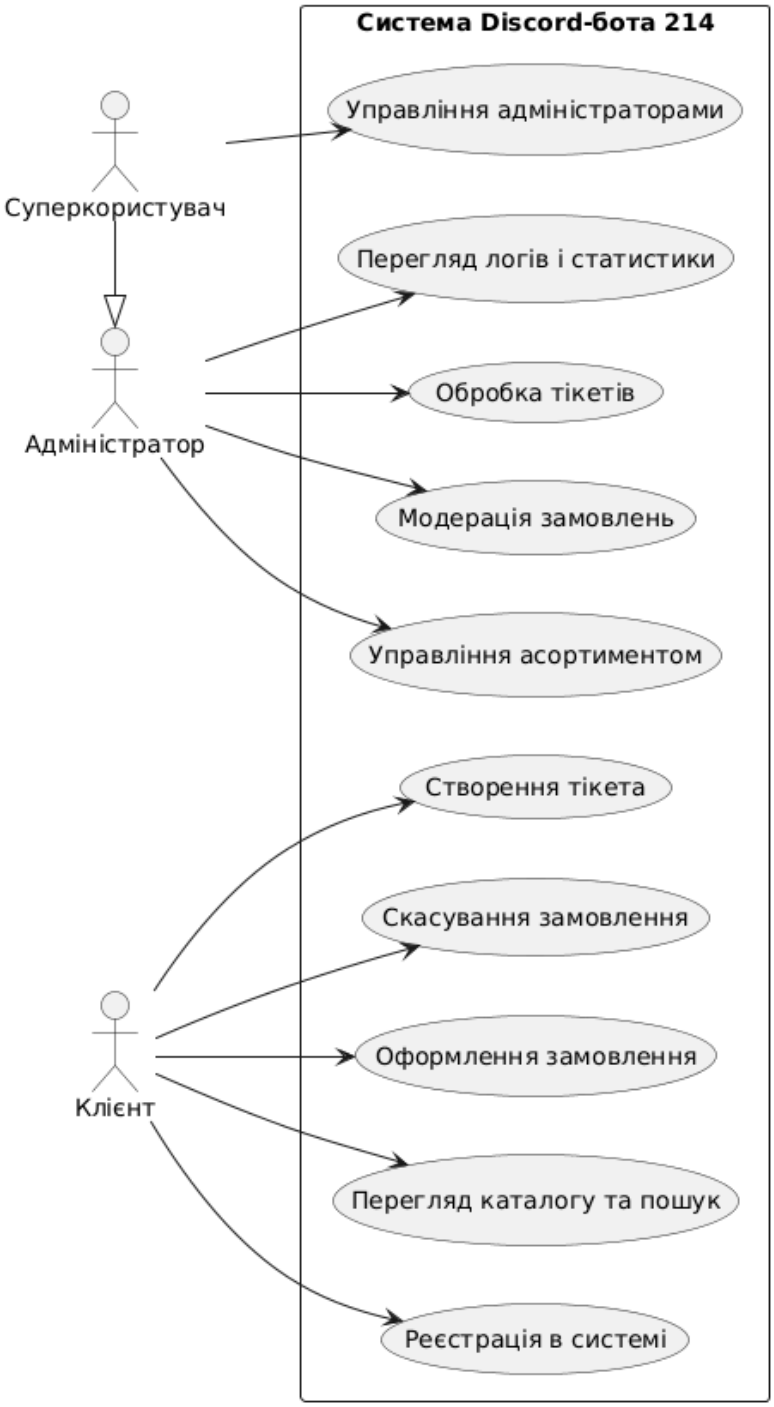

Рисунок 2.1 - UML-діаграма варіантів використання системи (Use Case Diagram)

Таблиця 2.1 - Розподіл прав та функціональних можливостей користувачів

Роль користувача	**Рівень доступу**	**Функціональні можливості та доступні команди**
Клієнт	Базовий	Реєстрація в системі (/start); перегляд асортименту з пагінацією (/catalog); детальний пошук позицій (/search, /product);

		оформлення замовлень (/order); скасування замовлень (/cancelorder); перегляд історії власних операцій (/myorders, /orderinfo); взаємодія зі службою підтримки через створення тікетів (/support) та перегляд їх статусу (/mytickets, /ticket).
Адміністратор	Розширений	Наповнення каталогу (/addproduct); редагування цінових та кількісних показників товарів (/editproduct); модерація замовлень клієнтів (/orders); зміна поточного етапу обробки замовлення (/setorderstatus); перегляд списку тікетів (/tickets); надання відповідей клієнтам та закриття звернень (/replyticket); аналіз логів системи та збір загальної статистики (/logs, /stats).
Суперкористувач	Повний (Root)	Усі функціональні можливості адміністратора; управління складом адміністративної групи (/addadmin, /removeadmin); зміна рівнів доступу; виконання дій, що мають критичний вплив на роботу програмного комплексу в цілому.

2.2 Проектування структури бази даних

Ефективність функціонування комп'ютерної системи автоматичного обліку замовлень та підтримки клієнтів безпосередньо залежить від раціонально спроектованої архітектури бази даних. Для забезпечення цілісності, мінімізації надлишковості та високої швидкості доступу до даних було обрано реляційну модель на базі СКБД MySQL. Вибір зумовлений необхідністю підтримки транзакційності (принципів ACID) та забезпечення суворої відповідності зв'язків між об'єктами системи.

Проектування бази даних включало етап логічного моделювання, результатом якого стало створення схеми з десяти взаємопов'язаних таблиць, що охоплюють усі аспекти бізнес-процесів бота: від реєстрації користувачів до аудиту адміністративних дій.

Опис таблиць бази даних Нижче наведено детальний опис структури кожної з десяти таблиць, реалізованих у системі:

1. **users (Користувачі):** Центральна таблиця системи, призначена для ідентифікації та зберігання даних про клієнтів. Вона містить унікальний Discord ID, ім'я користувача, відображуване ім'я, дату реєстрації, дату останньої активності та статус блокування . Поле id є первинним ключем (PRIMARY KEY) з автоінкрементом.

2. **admins (Адміністратори):** Таблиця для розмежування прав доступу. Вона пов'язана з таблицею users через зовнішній ключ (FOREIGN KEY) user_id. Це дозволяє системі перевіряти наявність адміністративних повноважень у користувача перед виконанням критичних команд .

3. **categories (Категорії):** Використовується для логічного групування товарів. Кожна категорія має унікальну назву та опис. Поле is_active дозволяє адміністратору тимчасово приховувати цілі групи товарів з каталогу.

4. **products (Товари):** Містить детальну номенклатуру товарів, включаючи ціну, опис та актуальну кількість на складі . Таблиця має зовнішній ключ category_id, що посилається на таблицю категорій. Це забезпечує структурований вивід каталогу за фільтрами .

5. **orders (Замовлення):** Зберігає заголовки транзакцій. Вона фіксує ідентифікатор користувача, загальну суму замовлення та поточний статус (new, confirmed, processing, completed, canceled) . Зв’язок із користувачем реалізовано через user_id.

6. **order_items (Позиції замовлення):** Описує конкретний склад кожного замовлення. Це таблиця-посередник, що пов'язує замовлення з товарами. Вона фіксує ціну товару на момент купівлі та його кількість, що важливо для формування чека .

7. **support_tickets (Звернення):** Призначена для організації служби підтримки. Кожен тікет має тему, початкове повідомлення та статус вирішення . Зовнішній ключ user_id ідентифікує автора звернення.

8. **ticket_messages (Повідомлення в тікетах):** Зберігає історію листування між клієнтом та адміністратором у межах конкретного звернення. Таблиця фіксує тип відправника (користувач або адмін), його ID та текст повідомлення . Пов'язана з тікетами через ticket_id.

9. **logs (Журнал дій):** Виконує роль системи аудиту. Вона реєструє тип дії, опис та ідентифікатор користувача, який її вчинив . Це критично важливо для відстеження змін у системі адміністраторами.
10. **order_status_history (Історія статусів замовлень):** Окремий лог для відстеження життєвого циклу замовлень. Вона фіксує старий та новий статуси, час зміни та відповідальну особу .

Для наочного представлення структури та логічних зв'язків між сутностями розроблено ER-діаграму (Entity-Relationship Diagram), що наведена на рисунку 2.2.

Зв’язки та цілісність даних Зв’язки в системі побудовані на основі обмежень цілісності та референсної надійності. Основні типи зв'язків:

- **Один-до-багатьох (1:N):** Один користувач може мати багато замовлень (users -> orders) та багато звернень (users -> support_tickets). Одна категорія може містити багато товарів (categories -> products).
- **Багато-до-багатьох (M:N):** Замовлення та товари пов'язані через розв'язувальну таблицю order_items, що дозволяє одному замовленню містити різні товари, а одному товару бути частиною різних замовлень.

Використання обмеження ON DELETE CASCADE для ключових зв'язків гарантує, що при видаленні тікета автоматично будуть видалені всі пов'язані з ним повідомлення, що запобігає появі "записів-сиріт". У той же час, для таблиці логів використано ON DELETE SET NULL, що дозволяє зберігати інформацію про дію в журналі навіть після видалення профілю користувача.

Нормалізація даних Структура бази даних відповідає вимогам третьої нормальної форми (3NF).

1. **Перша нормальна форма (1NF):** Усі атрибути таблиць є атомарними, кожне поле містить лише одне значення .
2. **Друга нормальна форма (2NF):** База відповідає вимогам 1NF, і всі неключові атрибути функціонально залежать від повного первинного ключа. Наприклад, ціна продукту залежить виключно від його id .
3. **Третя нормальна форма (3NF):** Виключено транзитивні залежності. Дані про категорії винесені в окрему таблицю categories, а не дублюються в кожному рядку товару, що зменшує ризик аномалій при оновленні інформації.

Характеристики полів та типи даних, використані при проектуванні, деталізовано в таблиці 2.2.

Таблиця 2.2 - Характеристики основних полів бази даних

Таблиця	**Поле**	**Тип даних**	**Опис**
users	discord_id	VARCHAR(20)	Унікальний ідентифікатор Discord
products	price	DECIMAL(10,2)	Грошовий тип для точності ціни
orders	status	ENUM(...)	Фіксований набір станів замовлення
support_tickets	message	TEXT	Повнотекстовий опис проблеми

Таким чином, спроектована база даних забезпечує надійний фундамент для роботи Discord-бота, гарантуючи збереження даних та можливість подальшого розширення системи.

2.3 Моделювання поведінки системи

Для повного розуміння принципів функціонування комп'ютерної системи недостатньо лише статичного опису структури бази даних. Життєвий цикл торгової платформи вимагає розробки динамічної моделі поведінки, яка описує бізнес-процеси (алгоритми) переходу системи з одного стану в інший під впливом дій користувачів. Моделювання поведінки здійснюється за допомогою діаграм мови UML, що відповідає сучасним стандартам проектування інформаційних систем .

У цьому підрозділі детально розглядаються два ключові бізнес-процеси: реєстрація та навігація каталогом (з точки зору загального прецеденту взаємодії) та процес оформлення і обробки замовлення (алгоритмічний аспект).

Моделювання варіантів використання (Use Case) Поведінка системи на макрорівні визначається через варіанти використання. Як зазначалося раніше, система має дві основні групи акторів: Клієнти та Адміністратори .

Сценарій взаємодії Клієнта починається з ініціалізації бота командою /start. Обов'язковою умовою для виконання будь-яких подальших дій є наявність запису користувача в базі даних (таблиця users). Якщо запис відсутній, система автоматично виконує реєстрацію у фоновому режимі, після чого користувач отримує доступ до каталогу . Бізнес-процес перегляду каталогу (/catalog)

включає запит до БД на вибірку активних товарів та повернення результату з розбиттям на сторінки (пагінація) .

Адміністратор, своєю чергою, ініціює процеси управління контентом. Сценарій додавання товару (/addproduct) вимагає перевірки прав доступу, валідації введених даних (назва, ціна, залишок) та створення запису в таблицях products та logs .

Алгоритмічне моделювання транзакції замовлення (Activity) Найбільш критичним з точки зору цілісності даних є бізнес-процес оформлення замовлення (/order). Його поведінка строго регламентована та вимагає виконання послідовності кроків з обробкою винятків на кожному етапі .

Алгоритм (Activity) проходження замовлення виглядає наступним чином:

1. **Ініціалізація:** Клієнт викликає команду /order, передаючи ID товару та бажану кількість .
2. **Валідація:** Система перевіряє наявність користувача в базі. Якщо користувача не знайдено (наприклад, він оминув команду /start), генерується виняток USER_NOT_FOUND, і процес переривається.
3. **Блокування (Locking):** Відкривається SQL-транзакція. Виконується запит SELECT ... FOR UPDATE до таблиці products. Це гарантує, що під час обробки цього замовлення інший клієнт не зможе купити цей самий товар, змінюючи його залишок .
4. **Перевірка залишків:** Якщо товару немає або його кількість (stock_quantity) менша за замовлену, генерується виняток NOT_ENOUGH_STOCK, транзакція скасовується (ROLLBACK).
5. **Фіксація операції:** За умови успішної валідації система створює новий запис у таблиці orders зі статусом new. Далі формується запис у таблиці order_items з ціною фіксації .
6. **Модифікація складу:** Відбувається зменшення значення stock_quantity у таблиці товарів на замовлену кількість .
7. **Аудит:** Додається запис у таблицю order_status_history .
8. **Завершення:** Виконується фіксація транзакції (COMMIT), і користувач отримує квитанцію про успішне замовлення у вигляді Discord Embed повідомлення .

Поведінка системи під час скасування замовлення (/cancelorder) є дзеркальним процесом: система перевіряє статус замовлення (скасувати можна лише статуси new або confirmed), змінює статус на canceled та повертає кількість товару назад на склад (UPDATE products SET stock_quantity = stock_quantity + ...) .

Для деталізації цього складного процесу буде наведено діаграму діяльності (Activity Diagram), яка графічно відобразить розгалуження та точки перевірки умов.

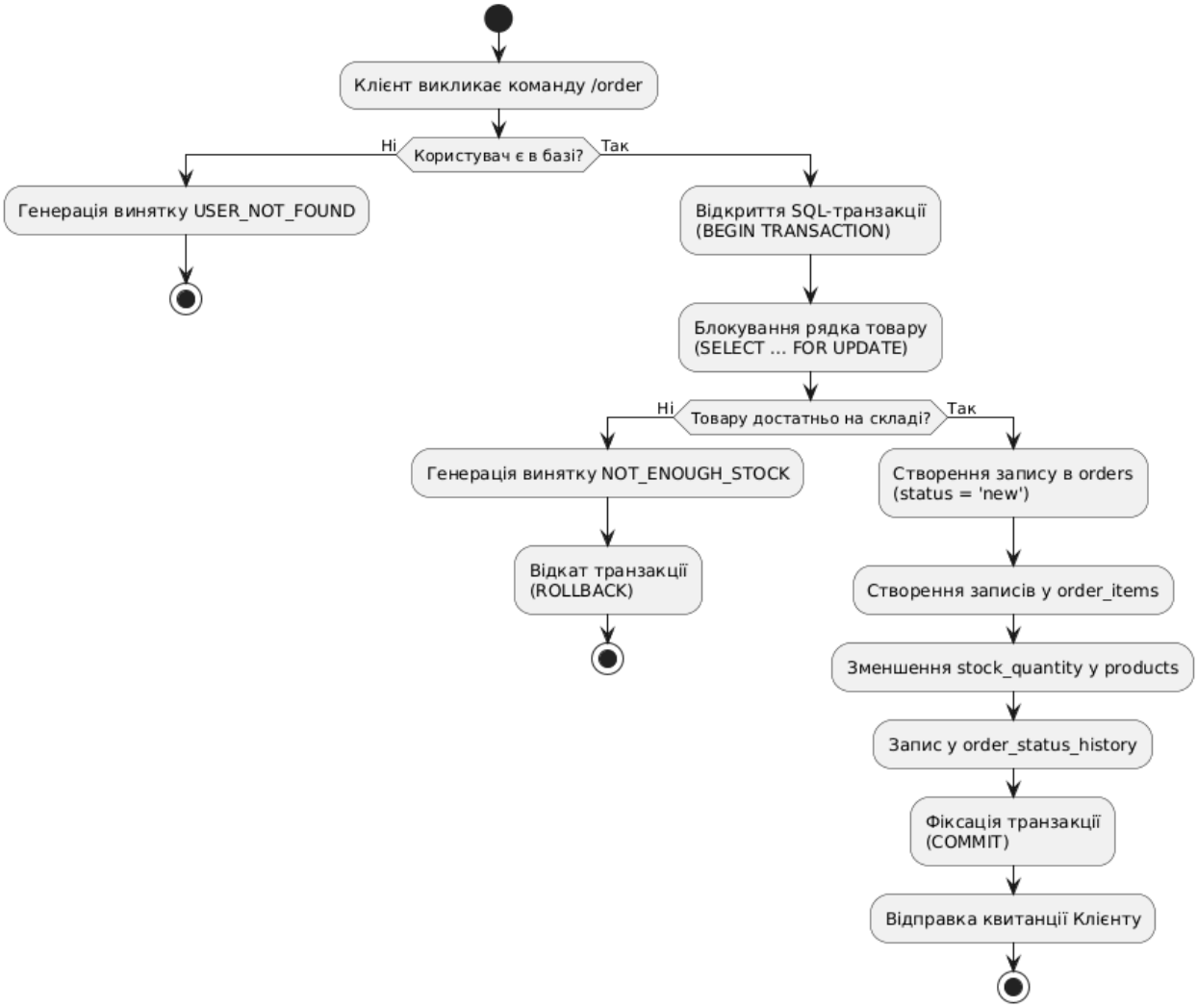

Рисунок 2.3 - UML-діаграма діяльності (Activity Diagram) процесу оформлення замовлення

Моделювання життєвого циклу об'єктів (State Machine)

Поведінка системи також відстежується через зміну станів ключових сутностей, зокрема замовлень та звернень до служби підтримки (тікетів) .

Замовлення проходить через такі стани: нове -> підтверджене (адміністратором) -> в обробці -> завершене. На етапі нове або підтверджене допускається перехід у стан скасоване з ініціативи клієнта . Тікети мають простіший життєвий цикл: відкрите (створене клієнтом) -> в обробці (після першої відповіді адміна) -> закрите (проблема вирішена). Ці зміни станів контролюються адміністратором через команди /setorderstatus та /replyticket, кожна з яких також супроводжується транзакційним логуванням . Таким чином, розроблена динамічна модель повністю описує реакції системи на зовнішні події та гарантує передбачуваність її поведінки у нестандартних ситуаціях.

2.4 Проектування інтерфейсу взаємодії (UI/UX)

Проектування інтерфейсу взаємодії (User Interface, UI) та користувацького досвіду (User Experience, UX) для комп'ютерних систем на базі месенджерів докорінно відрізняється від класичної веб-розробки. Оскільки розробник обмежений рамками платформи Discord, інтерфейс не використовує традиційні HTML/CSS макети, а спирається на нативні компоненти месенджера . Головною метою такого підходу є забезпечення максимальної зрозумілості, передбачуваності системи та мінімізації кількості хибних дій з боку користувача.

Згідно з технічними вимогами, інтерфейс системи побудований на трьох основних китах: Slash-командах (навігація), Embed-повідомленнях (презентація даних) та інтерактивних компонентах (взаємодія) .

Логіка побудови Slash-команд У сучасній архітектурі Discord основним способом взаємодії актора із системою є Slash-команди (команди, що починаються зі знаку /). На відміну від застарілих текстових команд із префіксами, Slash-команди нативно інтегровані в графічний інтерфейс клієнта.

З погляду UX, використання Slash-команд вирішує кілька ключових проблем:

1. **Автоматичне виявлення (Discoverability):** Користувачу не потрібно запам'ятовувати список команд. При введенні символу / система автоматично відображає меню доступних дій із короткими описами .
2. **Валідація на стороні клієнта:** Під час проектування команд (через клас SlashCommandBuilder) задаються жорсткі типи аргументів. Наприклад, для команди /order параметр product_id вимагає введення цілого числа (addIntegerOption), а параметр кількості має обмеження мінімального значення (setMinValue(1)) . Це запобігає відправленню некоректних даних на сервер і зменшує кількість помилок бази даних [15].

3. **Обов'язковість аргументів:** Система не дозволить відправити команду /replyticket, якщо адміністратор не заповнив обов'язкове текстове поле message (setRequired(true)) .

Таблиця 2.3 - Структура ключових Slash-команд та їх параметрів

Назва команди	**Призначення**	**Параметри (Тип, Обмеження)**	**Доступ**
/start	Автоматична реєстрація в системі та ініціалізація сесії	*Без параметрів*	Клієнт
/catalog	Перегляд асортименту товарів із пагінацією	*Без параметрів*	Клієнт
/search	Пошук товару за ключовим словом	query (Рядок, Обов'язковий)	Клієнт
/product	Отримання детальної інформації про товар	id (Ціле число, Обов'язковий)	Клієнт
/order	Транзакційне оформлення замовлення	product_id (Ціле число, Обов'язковий) quantity (Ціле число, Обов'язковий, min: 1)	Клієнт
/cancelorder	Реверсивне скасування замовлення	order_id (Ціле число, Обов'язковий)	Клієнт
/support	Створення звернення до служби підтримки	subject (Рядок, Обов'язковий, max: 100) message (Рядок, Обов'язковий, max: 1000)	Клієнт
/addproduct	Додавання нової товарної позиції	name (Рядок, Обов'язковий) price (Число, Обов'язковий) stock (Ціле число, Обов'язковий) category_id (Ціле число, Обов'язковий) description (Рядок, Необов'язковий)	Адмін
/editproduct	Зміна ціни або складського залишку	product_id (Ціле число, Обов'язковий) new_price (Число, Необов'язковий) new_stock (Ціле число, Необов'язковий)	Адмін

/setorderstatus	Модерація життєвого циклу замовлення	order_id (Ціле число, Обов'язковий) status (Вибір зі списку, Обов'язковий)	Адмін
/replyticket	Відповідь на звернення та зміна його статусу	ticket_id (Ціле число, Обов'язковий) message (Рядок, Обов'язковий) close_ticket (Логічний, Необов'язковий)	Адмін
/logs	Перегляд аудиторського сліду	*Без параметрів*	Адмін
/stats	Агрегована статистика бази даних	*Без параметрів*	Адмін

Форматування інформації через Discord Embeds. Оскільки текстові повідомлення погано підходять для відображення структурованих даних (таких як каталоги товарів або чеки замовлень), виведення всієї бізнес-інформації реалізовано через Embed-повідомлення (об'єкт EmbedBuilder). Це спеціальні блоки, які підтримують розширене форматування: заголовки, поля (fields), зображення (thumbnails) та кольорову ідентифікацію.

Логіка UI/UX передбачає чітке кольорове кодування (Color Coding), яке допомагає користувачеві миттєво зчитати контекст повідомлення:

- **Зелений (0x00FF00, 0x2ECC71):** Успішні операції (реєстрація, успішне оформлення замовлення, вирішення тікета) .
- **Червоний (0xE74C3C):** Критичні дії або помилки (скасування замовлення, закриття тікета) .
- **Помаранчевий/Жовтий (0xFFA500, 0xF1C40F):** Інформаційні панелі (каталог товарів, створення звернення до підтримки) .
- **Синій/Блакитний (0x3498DB, 0x00FFFF):** Детальна інформація про сутності (профіль клієнта, картка товару, статистика системи) .

Для імітації табличного виводу використовуються поля addFields із параметром inline: true, що дозволяє розміщувати дані (наприклад, ціну, категорію та кількість на складі) у кілька колонок на одному рівні .

Інтерактивні компоненти та конфіденційність (Ephemeral messages). Для оптимізації UX при роботі з великими масивами даних (наприклад, виведення десятків товарів) спроектовано систему пагінації каталогу. Замість того, щоб перевантажувати чат довгим текстом, система обмежує вивід до 3 товарів на сторінку та генерує кнопки управління («Попередня», «Наступна») за допомогою класів ActionRowBuilder та ButtonBuilder . Це створює відчуття повноцінного інтерактивного застосунку (SPA) всередині чату.

Важливим аспектом UX є також збереження конфіденційності клієнта. Оскільки бот функціонує у спільних каналах, команди, що містять особисту інформацію (чеки замовлень, створення тікетів, перегляд профілю), проектуються з використанням прапорця ephemeral: true. Це означає, що відповідь бота буде видимою виключно тому користувачеві, який викликав команду, і автоматично зникне після перезавантаження клієнта Discord.

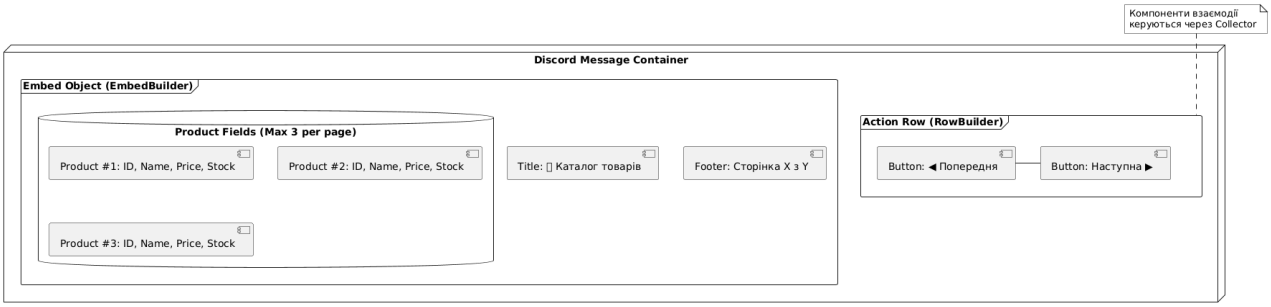

Рисунок 2.5 - Структура інтерактивного меню каталогу з пагінацією та форматуванням Embed

Спроектований таким чином інтерфейс взаємодії повністю відповідає сучасним вимогам до діалогових систем (Conversational UI), поєднуючи надійність захисту даних із простотою навігації [16].

2.5 Забезпечення безпеки та надійності даних

Проектування архітектури безпеки є критично важливим етапом створення будь-якої комерційної комп'ютерної системи. Враховуючи специфіку інтеграції через платформу Discord, розроблений програмний продукт функціонує у відкритому середовищі, що підвищує ризики несанкціонованого доступу та конкурентних конфліктів при обробці даних. Забезпечення безпеки та надійності даних у даній роботі реалізується на трьох взаємопов'язаних рівнях: захист периметра та розмежування доступу, транзакційна цілісність на рівні бази даних та неперервний аудит дій.

Розмежування доступу та захист периметра. Базовим рівнем безпеки є захист конфігураційних даних та облікових записів. Відповідно до вимог безпеки, токен

авторизації бота в Discord API та паролі до бази даних MySQL винесено за межі програмного коду у спеціальний файл змінних оточення .env . Це унеможливлює витік критичної інформації у разі передачі вихідного коду третім особам або розміщення його у публічних репозиторіях.

Оскільки бот перебуває на загальнодоступному сервері, система має надійно блокувати спроби виконання адміністративних команд звичайними клієнтами . Для цього реалізовано дворівневий механізм авторизації:

1. **Обмеження інтерфейсу (Discord Level):** Використовується нативний метод setDefaultMemberPermissions(PermissionFlagsBits.Administrator) під час ініціалізації Slash-команд. Цей прапорець приховує адмін-команди з меню інтерфейсу користувачів, які не мають відповідних серверних прав.
2. **Апаратна авторизація (Server Level):** Навіть якщо зловмисник знайде спосіб обійти клієнтські обмеження Discord і надішле POST-запит безпосередньо на шлюз (Gateway), серверна частина на Node.js примусово звіряє унікальний ідентифікатор користувача (discord_id) з таблицею admins у базі даних . Якщо ID не знайдено, виконання команди миттєво переривається з генерацією винятку, що повністю задовольняє вимоги технічного завдання щодо захисту критичних дій .

Додатковим фактором безпеки є використання параметризованих (підготовлених) запитів при роботі з бібліотекою mysql2. Усі змінні, отримані від користувача (наприклад, пошукові запити або описи тікетів), передаються в базу даних через плейсхолдери ? . Як зазначають фахівці з кібербезпеки, «використання параметризованих запитів на рівні драйвера СУБД є єдиним надійним методом превентивного захисту інформаційних систем від атак типу SQL Injection» [17].

Механізми транзакцій та запобігання стану гонки (Race Conditions). Електронна комерція передбачає паралельну обробку запитів від багатьох клієнтів. Якщо два користувачі одночасно намагатимуться придбати останню одиницю товару на складі, виникне конкурентний конфлікт (Race Condition), що може призвести до від'ємних залишків у базі даних.

Для вирішення цієї проблеми під час розробки команд /order та /cancelorder було імплементовано механізм суворих SQL-транзакцій у поєднанні з песимістичним блокуванням рядків (Pessimistic Row-Level Locking).

Алгоритм транзакційної безпеки замовлення діє за таким принципом:

1. Сервер ініціює створення ізольованого пулу з'єднання та відкриває транзакцію викликом connection.beginTransaction() .
2. Виконується запит SELECT ... FOR UPDATE. Цей оператор фізично блокує запис конкретного товару в таблиці products для інших сесій до моменту завершення поточної транзакції.
3. Проводиться валідація наявності необхідної кількості товару (if (product.stock_quantity < quantity) throw new Error(...)) .
4. Якщо перевірку пройдено, система виконує серію інструкцій INSERT та UPDATE (оновлення залишків, створення запису в orders, створення позицій чека) .
5. Якщо всі операції виконані без збоїв, зміни фіксуються оператором COMMIT.
6. Якщо на будь-якому етапі (навіть через апаратний збій мережі) виникає помилка, спрацьовує блок catch, який виконує оператор ROLLBACK . Усі проміжні зміни скасовуються, база даних повертається до початкового консистентного стану, і товар не списується зі складу помилково. Ця ж логіка застосована при поверненні товарів під час скасування замовлення .

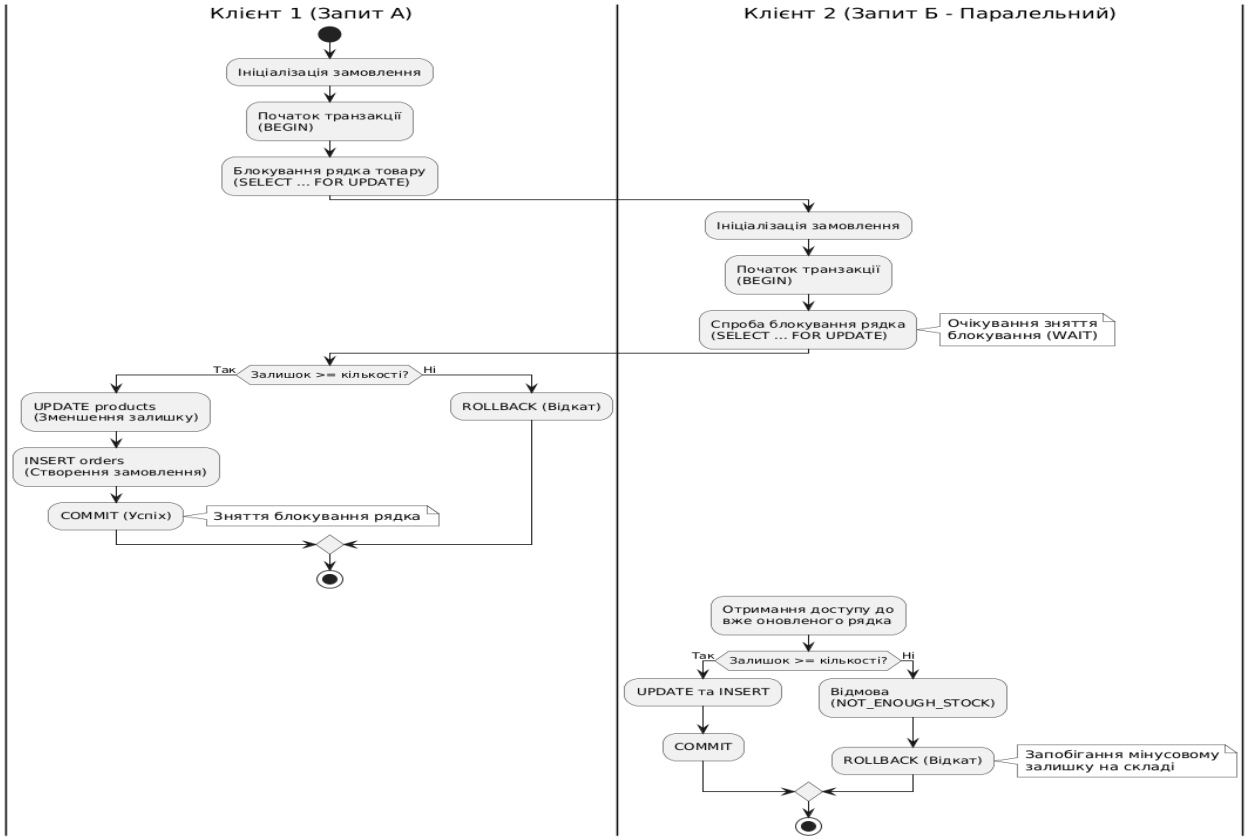

Рисунок 2.6 - Алгоритм обробки транзакції з блокуванням рядка (Pessimistic Locking) при паралельних запитах

Логування та аудит дій адміністратора. Відповідно до принципу підзвітності (Accountability), система не може вважатися захищеною, якщо в ній не реалізовано механізм відстеження дій персоналу. Для виконання вимоги ТЗ щодо журналювання було створено розширену систему аудиту, яка охоплює дві таблиці: logs та order_status_history.

Будь-яка модифікація критичних даних супроводжується обов'язковим записом у журнал.

- Під час додавання товару (/addproduct) або зміни його ціни (/editproduct) у таблицю logs записується тип дії, детальний опис (наприклад, "нова ціна = 700") та ID адміністратора, який вніс зміни .
- Під час переведення замовлення на наступний етап (/setorderstatus) система фіксує перехід у таблиці order_status_history (зберігаючи старий і новий статус), що унеможливлює приховану зміну стану фінансових документів .

«Побудова неперервного аудиторського сліду (Audit Trail) гарантує прозорість бізнес-процесів і дозволяє суперкористувачам або власникам бізнесу оперативно виявляти та розслідувати інциденти внутрішнього шахрайства» [18]. Для

перегляду аудиторського сліду реалізовано команду /logs, що виводить інформацію безпосередньо в інтерфейс Discord .

Впровадження описаних механізмів забезпечує високий рівень відмовостійкості, ізолює систему від ін'єкцій шкідливого коду, усуває загрозу фінансових втрат від конкурентних запитів та забезпечує повний контроль за адміністративними процесами.

Таблиця 2.4 - Механізми забезпечення безпеки на різних рівнях архітектури системи

Рівень архітектури	**Механізм захисту**	**Опис та функціональне призначення**
Discord (Периметр)	Обмеження інтерфейсу (Client-side)	Використання методу setDefaultMemberPermissions для приховування адміністративних команд від звичайних користувачів .
Сервер застосунку (Node.js)	Апаратна авторизація (Server-side)	Примусова перевірка discord_id ініціатора за внутрішньою таблицею admins перед виконанням критичної логіки .
Сервер застосунку (Node.js)	Ізоляція конфігурації	Зберігання токенів API та паролів до СКБД у файлі змінних оточення .env за межами програмного коду .
База даних (MySQL)	SQL-транзакції та ACID	Використання операторів BEGIN, COMMIT та ROLLBACK для гарантування цілісності комплексних операцій .
База даних (MySQL)	Песимістичне блокування рядків	Застосування оператора FOR UPDATE для запобігання стану гонки (Race Condition) при одночасних покупках .
База даних (MySQL)	Параметризовані запити	Передача вхідних даних через плейсхолдери (?) для превентивного захисту від атак типу SQL Injection .
Аудит та моніторинг	Неперервний аудиторський слід	Автоматичне логування всіх дій адміністраторів та змін станів замовлень у таблицях logs та order_status_history .

РОЗДІЛ 3. РЕАЛІЗАЦІЯ КОМП'ЮТЕРНОЇ СИСТЕМИ

3.1 Розробка структури програмного проекту

Реалізація будь-якого масштабованого програмного забезпечення на базі платформи Node.js розпочинається зі створення чіткої ієрархічної структури файлів і каталогів. Зважаючи на те, що розроблювана комп’ютерна система виконує одночасно функції чат-бота, обробника транзакцій та адміністративної панелі, монолітне розміщення коду в одному файлі є категорично неприпустимим . Відсутність структуризації призводить до надмірної зв'язності коду (Spaghetti code), ускладнює процес налагодження та унеможливлює подальше розширення функціоналу.

Для вирішення цієї проблеми в процесі розробки було застосовано архітектурний підхід, заснований на принципах модульного програмування та розділення відповідальності (Separation of Concerns). Це означає, що кожний логічний блок системи винесено в окремий модуль, який експортує свої функції для використання ядром системи [19].

Кореневий каталог та конфігурація. У кореневому каталозі проекту розміщено фундаментальні файли, що забезпечують ініціалізацію та конфігурацію середовища:

- package.json та package-lock.json — файли маніфесту проекту, які керують залежностями (встановленими бібліотеками discord.js, mysql2, dotenv) та містять скрипти для запуску системи (наприклад, npm run dev із використанням nodemon для автоматичного перезапуску при змінах) .
- .env — прихований файл конфігурації оточення. Згідно з вимогами кібербезпеки, у ньому зберігаються конфіденційні дані: токен авторизації бота (DISCORD_TOKEN), ідентифікатори програми та параметри доступу до бази даних MySQL (DB_HOST, DB_USER, DB_PASSWORD) .
- index.js — ядро застосунку (точка входу), яке ініціалізує клієнта Discord, підключає базу даних та динамічно завантажує обробники подій і команд .
- deploy-commands.js — спеціалізований скрипт (реєстратор), призначений виключно для синхронізації локального списку Slash-команд із серверами (API) Discord .

Відокремлення бізнес-логіки (Каталог commands). Найважливішим архітектурним рішенням стало винесення логіки обробки запитів у

спеціалізовану директорію commands. Замість того, щоб прописувати дії для кожної команди всередині події interactionCreate у файлі index.js, ядро бота сканує цю папку та динамічно реєструє всі знайдені модулі. Це відповідає патерну проектування Command (Команда), який інкапсулює запит у вигляді окремого об'єкта [20].

Для забезпечення додаткового рівня абстракції та безпеки, каталог commands жорстко поділено на дві піддиректорії:

1. **Підкаталог user/:** Містить модулі, призначені для взаємодії з клієнтами. Тут розміщено файли, які не вимагають підвищених привілеїв: реєстрація (start.js), навігація по магазину (catalog.js, product.js, search.js), оформлення та управління власними замовленнями (order.js, myorders.js, cancelorder.js), а також створення тікетів підтримки (support.js, mytickets.js) .
2. **Підкаталог admin/:** Ізолює файли з критичною бізнес-логікою, доступ до яких на рівні коду обмежено прапорцем PermissionFlagsBits.Administrator. До цієї групи належать модулі управління складом (addproduct.js, editproduct.js), модерації замовлень (orders.js, setorderstatus.js), обробки скарг (tickets.js, replyticket.js) та інструменти аудиту (stats.js, logs.js) .

«Фізичне розділення клієнтських та адміністративних модулів на рівні файлової системи не лише спрощує навігацію проектом для розробника, але й унеможливлює випадкове об’єднання публічного функціоналу з конфіденційними адміністративними методами під час рефакторингу» [21].

Ізоляція рівня доступу до даних (Каталог database). Усі процеси, пов'язані зі встановленням з'єднання та конфігурацією системи керування базами даних, винесено в каталог database. Файл db.js, що знаходиться всередині, реалізує патерн Singleton (Одинак) для створення єдиного пулу з'єднань (Connection Pool) за допомогою бібліотеки mysql2/promise . Відмова від відкриття нового з'єднання при кожному запиті на користь пулу (з лімітом connectionLimit: 10) значно підвищує пропускну здатність системи в умовах високого навантаження. Модулі з папки commands лише імпортують готовий об'єкт pool для виконання SQL-запитів.

Згідно з рисунком 3.1, також передбачено створення допоміжних каталогів (config, events, services, utils), які формують заділ для майбутнього масштабування проекту та переходу до мікросервісної архітектури, якщо виникне необхідність інтеграції з зовнішньою веб-панеллю.

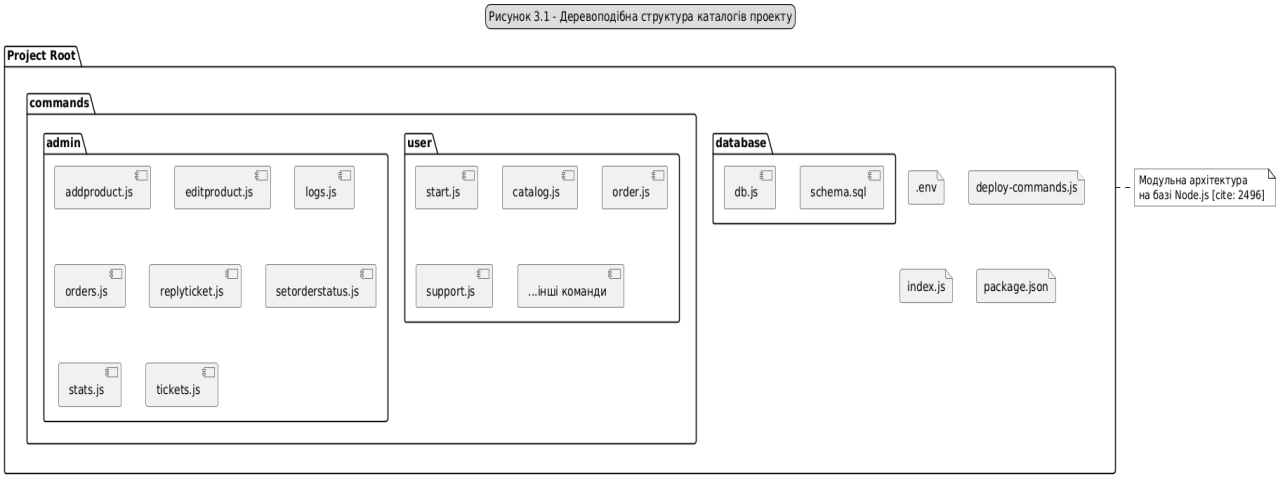

Рисунок 3.1 - Деревоподібна структура каталогів та модулів програмного проекту

Реалізована структура програмного коду забезпечує високу підтримуваність (Maintainability), читабельність та повну відповідність сучасним галузевим стандартам розробки застосунків мовою JavaScript.

3.2 Реалізація підключення та пулу з'єднань з базою даних

Надійність та швидкість взаємодії серверної частини із системою керування базами даних (СКБД) є визначальними чинниками стабільності комп’ютерної системи автоматичного обліку . У розроблюваному застосунку реалізація цього вузла базується на використанні сучасного драйвера mysql2 з підтримкою промісів (Promises), що дозволяє писати асинхронний код у стилі async/await, уникаючи проблеми «callback hell» та забезпечуючи високу читабельність логіки [22].

Конфігурація та безпека (Файл .env)

Першим етапом реалізації підключення є забезпечення безпечного зберігання конфіденційних даних. Відповідно до вимог безпеки, параметри доступу до бази даних (хост, логін, пароль, назва БД) не прописуються безпосередньо у вихідному коді застосунку, а виносяться у файл змінних оточення .env.

Робота з цим файлом забезпечується бібліотекою dotenv. Під час запуску бота вона зчитує змінні та додає їх до глобального об'єкта process.env. Це дозволяє змінювати параметри розгортання системи (наприклад, при переході з локального сервера XAMPP на віддалений VPS) без втручання в основний програмний код [23]. Перелік основних змінних конфігурації БД наведено в таблиці 3.1.

Таблиця 3.1 - Параметри конфігурації підключення до СКБД

Змінна оточення	**Призначення**	**Опис значення**
DB_HOST	Адреса сервера	Зазвичай localhost для локальної розробки
DB_USER	Ім’я користувача	Стандартно root для середовища XAMPP
DB_PASSWORD	Пароль доступу	Секретний ключ користувача бази даних
DB_NAME	Назва бази	Назва створеної схеми (наприклад, discord_store)

Обґрунтування використання Connection Pool. У процесі розробки модуля database/db.js замість поодиноких з’єднань було застосовано механізм пулу з’єднань (Connection Pool) . Традиційний підхід «один запит — одне підключення» є неефективним для Discord-бота, оскільки відкриття та закриття TCP-сесії з MySQL потребує значних часових та обчислювальних ресурсів при кожній команді користувача.

Використання пулу з’єднань надає такі переваги:

1. **Зниження латентності.** Система тримає певну кількість підключень постійно відкритими та готовими до використання, що мінімізує час відгуку бота [24].
2. **Керування навантаженням.** Завдяки параметру connectionLimit: 10, система обмежує максимальну кількість одночасних сесій до бази, що запобігає відмові СКБД при масових запитах від користувачів.
3. **Автоматичне відновлення.** Пул самостійно перевіряє стан з’єднань і відновлює їх у разі розриву з боку сервера.

Програмна реалізація модуля db.js. Модуль підключення реалізований як окремий сервіс, який експортує об'єкт пулу для використання іншими частинами системи. Архітектурно це дозволяє ініціалізувати базу даних один раз при старті ядра (index.js) та гарантувати, що всі Slash-команди будуть використовувати спільні ресурси .

Алгоритм ініціалізації підключення (див. рис. 3.2) передбачає завантаження конфігурації, створення об'єкта pool та виконання тестового запиту для перевірки доступності даних. У разі неможливості з’єднання система ініціює примусову зупинку процесу (process.exit(1)), що запобігає роботі бота в «сліпому» режимі без доступу до даних .

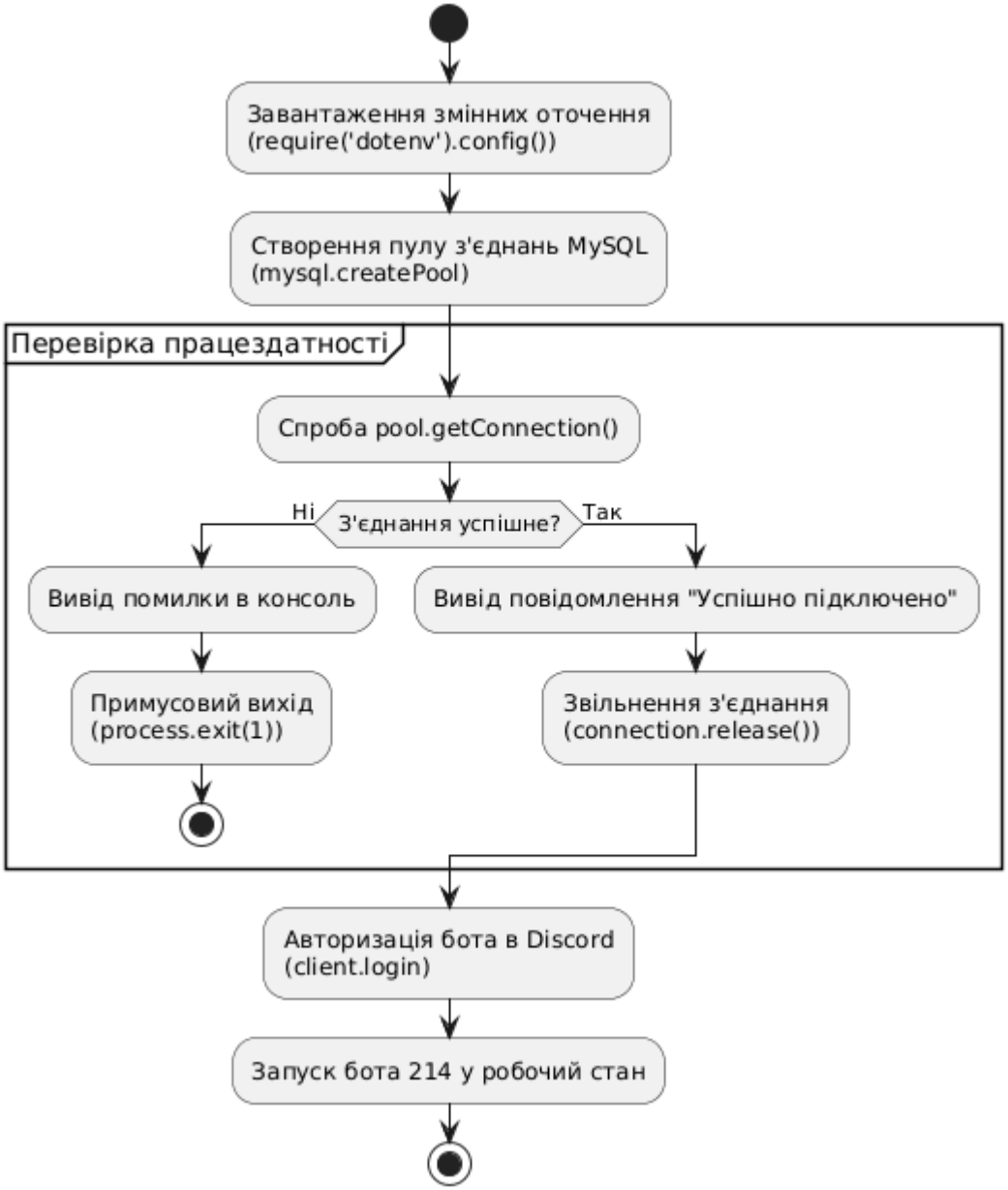

Рисунок 3.2 - Алгоритм ініціалізації підключення до бази даних через Connection Pool

«Використання пулу з’єднань у поєднанні з асинхронними драйверами є індустріальним стандартом при розробці високонавантажених застосунків на Node.js, оскільки це забезпечує оптимальний баланс між продуктивністю та споживанням пам’яті сервера» [25].

3.3 Розробка ядра бота та динамічного обробника команд

Архітектура ядра комп'ютерної системи базується на подієво-орієнтованій моделі, що забезпечується середовищем Node.js та бібліотекою discord.js. Основним завданням цього вузла є стабільна підтримка зв'язку із шлюзом Discord (Gateway), розпізнавання вхідних команд користувача та їх безпечне виконання. Для забезпечення гнучкості системи було розроблено механізм динамічного завантаження модулів, що дозволяє додавати нові функції без рефакторингу основного файлу index.js.

Синхронізація команд через deploy-commands.js. Оскільки сучасні Slash-команди Discord реєструються на стороні серверів платформи, у проекті реалізовано окремий службовий модуль deploy-commands.js. Його функція полягає у скануванні каталогу commands/, витягуванні метаданих команд у форматі JSON та їх передачі через REST API Discord .

Алгоритм роботи реєстратора включає:

1. Ініціалізацію інтерфейсу REST з використанням секретного токена із файлу .env.
2. Рекурсивне (або ієрархічне) зчитування підпапок user/ та admin/ для збору всіх файлів із розширенням .js .
3. Перевірку наявності необхідних методів data та execute у кожному модулі .
4. Надсилання PUT-запиту до маршруту Routes.applicationGuildCommands, що гарантує миттєве оновлення списку команд на цільовому сервері, на відміну від глобальної реєстрації, яка може тривати до однієї години .

Ядро системи та обробка взаємодій (index.js). Центральним компонентом системи є файл index.js, який виконує роль керуючого контролера. Програмна реалізація ядра базується на класі Client, що налаштований із використанням необхідних дозволів (Intents): Guilds (для доступу до структури сервера), GuildMessages та MessageContent .

Ключовим елементом ядра є структура даних Collection, яка є розширеною версією об'єкта Map. У ній зберігаються посилання на всі доступні команди, де ключем є назва команди, а значенням — об'єкт модуля. Це забезпечує складність пошуку $O(1)$, що критично для швидкодії системи.

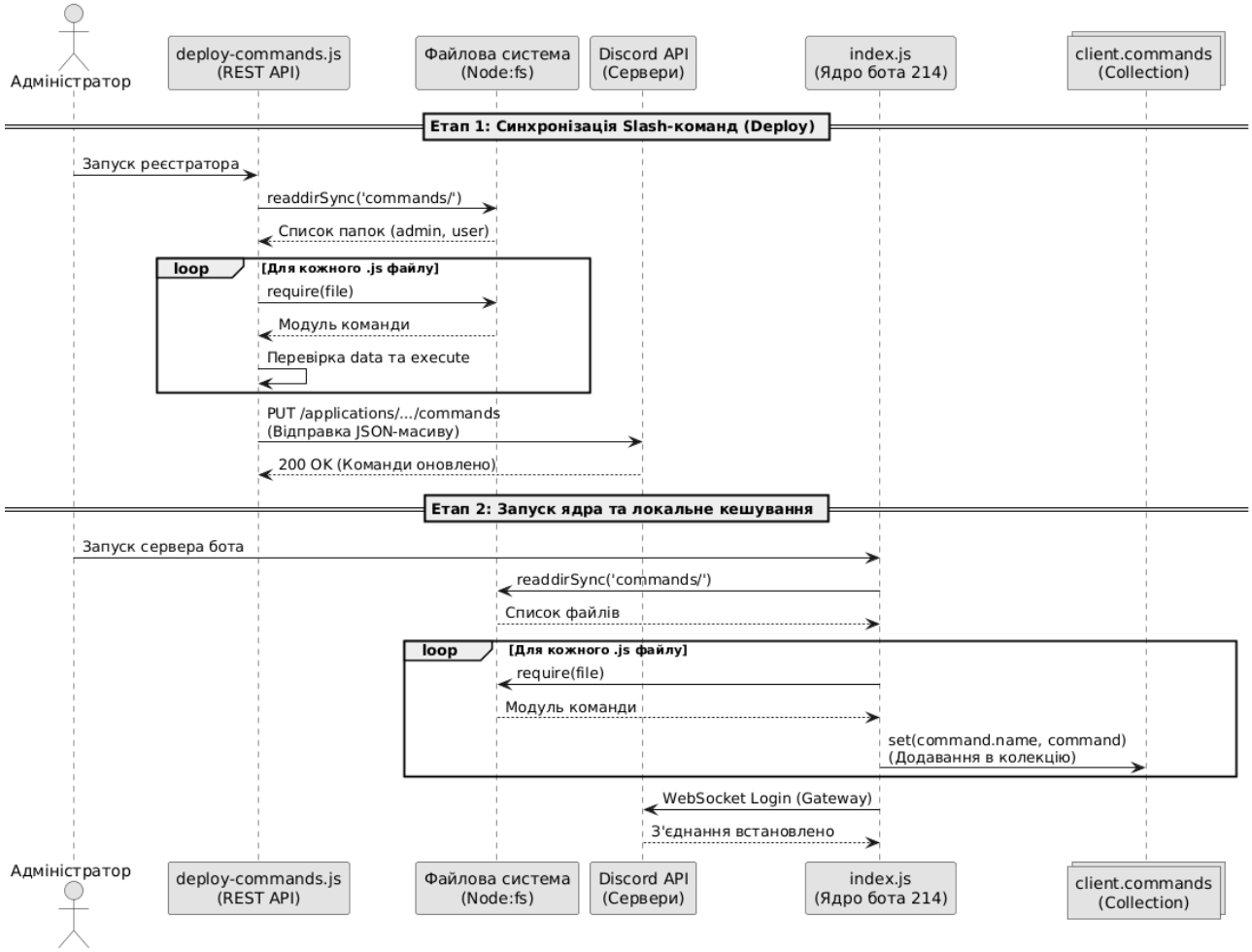

Рисунок 3.3 - Діаграма послідовності завантаження та реєстрації команд при старті системи

Алгоритм динамічного зчитування та виконання команд\. Процес ініціалізації обробника команд (Command Handler) реалізовано як циклічну операцію зчитування файлової системи (див. рис. 3.4).

1. **Сканування:** Програма використовує вбудований модуль node:fs для отримання списку категорій (папок) у директорії commands .
2. **Імпорт:** Для кожного знайденого файлу викликається функція require(), яка завантажує код модуля в оперативну пам'ять.
3. **Реєстрація:** Якщо модуль проходить валідацію, він додається до колекції client.commands.
4. **Слухач подій:** При отриманні події interactionCreate (виклик команди користувачем), ядро перевіряє тип взаємодії isChatInputCommand() та шукає відповідну назву в колекції .

5. **Ізольоване виконання:** Команда запускається всередині блоку try...catch. Це гарантує, що помилка в логіці однієї команди (наприклад, невірний SQL-запит у /stats) не призведе до падіння всього бота .

«Використання паттерна "Command Handler" дозволяє створювати архітектурно чисті системи, де ядро залишається незмінним, а функціональні можливості нарощуються шляхом простого створення нових файлів у відповідних директоріях» [26].

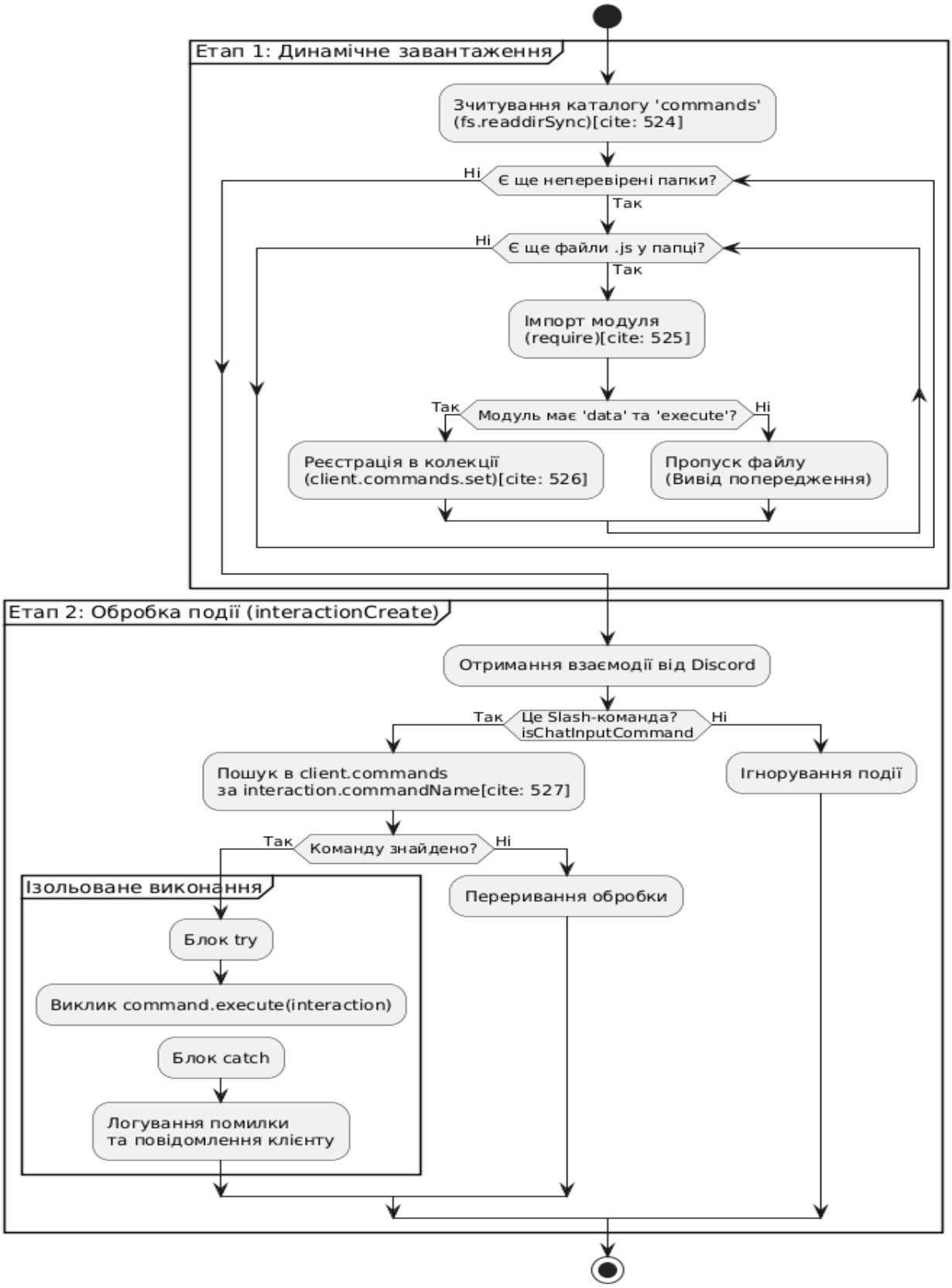

Рисунок 3.4 - Блок-схема алгоритму динамічного завантаження модулів (Command Handler)

Результатом розробки ядра стала стійка до відмов програмна оболонка, яка забезпечує автоматичне підключення до бази даних, авторизацію в Discord та швидку маршрутизацію запитів від користувачів різних рівнів доступу .

3.4 Програмування бізнес-логіки клієнтської частини

Бізнес-логіка клієнтської частини відповідає за безпосередню взаємодію кінцевого користувача з комп'ютерною системою. Згідно з вимогами технічного завдання , вона охоплює процеси реєстрації, пошуку товарів, формування замовлень та створення звернень до служби підтримки. Для реалізації цих завдань було розроблено набір Slash-команд, що зберігаються у директорії commands/user/ . У цьому підрозділі проведено детальний розбір найважливіших алгоритмів та фрагментів коду, які забезпечують виконання цих функцій.

Алгоритм автоматичної реєстрації користувача. Взаємодія із системою розпочинається з виклику команди /start. Оскільки використання окремих форм реєстрації в Discord є незручним для користувачів, було прийнято рішення реалізувати механізм неявної (фонової) реєстрації .

Алгоритм, реалізований у файлі start.js , під час виконання команди витягує унікальний ідентифікатор користувача (interaction.user.id), його ім'я (username) та відображуване ім'я (displayName) безпосередньо з об'єкта взаємодії (Interaction). Далі виконується перевірка наявності цього ідентифікатора у базі даних:

```
// Шукаємо користувача за Discord ID
const [rows] = await pool.query('SELECT * FROM users WHERE discord_id = ?', [discordId]);

if (rows.length === 0) {
  // Якщо користувача немає — реєструємо
  await pool.query(
    'INSERT INTO users (discord_id, username, display_name) VALUES (?, ?, ?)',
    [discordId, username, displayName]
  );
  await interaction.reply(`Вітаю, **${displayName}**! Тебе успішно зареєстровано...`);
```

```
} else {
   // Користувач вже існує
   await interaction.reply(`З поверненням, **${displayName}**!...`);
}
```

Використання параметризованих запитів (знаки питання ? замість прямого вставлення змінних) унеможливлює виконання SQL-ін'єкцій, навіть якщо користувач спробує змінити своє ім'я в Discord на шкідливий SQL-код [27].

Реалізація пагінації в інтерактивному каталозі. Одним із найскладніших елементів клієнтської частини є команда /catalog, реалізована у файлі catalog.js . Зважаючи на те, що Discord має жорсткі ліміти на кількість символів в одному повідомленні (до 4096 символів в описі Embed) та кількість полів (не більше 25), виведення всього асортименту товарів єдиним списком є неможливим [28]. Для вирішення цієї проблеми було спроектовано систему пагінації (сторінкового виведення) з використанням інтерактивних кнопок.

На початку виконання команди скрипт робить запит до таблиці products, поєднуючи її з таблицею categories за допомогою LEFT JOIN, щоб відразу отримати назви категорій . Зчитаний масив об'єктів товарів розбивається на фрагменти по 3 одиниці (itemsPerPage = 3).

Для побудови інтерфейсу керування сторінками використовується об'єкт ActionRowBuilder та компоненти ButtonBuilder:

```
const getButtons = (page) => {
   const row = new ActionRowBuilder();
   row.addComponents(
      new ButtonBuilder()
         .setCustomId('prev_page')
         .setLabel(' ◀ Попередня ')
         .setStyle(ButtonStyle.Primary)
         .setDisabled(page === 1), // Вимкнути на першій сторінці
      new ButtonBuilder()
         .setCustomId('next_page')
         .setLabel('Наступна ▶ ')
```

```
        .setStyle(ButtonStyle.Primary)
        .setDisabled(page === pages) // Вимкнути на останній сторінці
    );
    return row;
};
```

Обробка подій інтерфейсу (Message Component Collector). Для того щоб кнопки реагували на натискання, до повідомлення прив'язується колектор компонентів (createMessageComponentCollector). Цей колектор перехоплює події взаємодії (interaction) саме з цими кнопками протягом встановленого часу (у даному випадку — 60000 мілісекунд, або 1 хвилина).

Важливим аспектом безпеки є перевірка того, хто натиснув кнопку. З огляду на те, що бот функціонує в загальному чаті, необхідно запобігти ситуації, коли один користувач викликає каталог, а інший перемикає йому сторінки. Для цього імплементовано перевірку ідентифікаторів:

```
collector.on('collect', async i => {
    // Перевірка: чи збігається ID того, хто натиснув кнопку, з ID того, хто викликав команду
    if (i.user.id !== interaction.user.id) {
        return i.reply({ content: 'Ви не можете використовувати ці кнопки.', ephemeral: true });
    }

    // Зміна поточної сторінки
    if (i.customId === 'prev_page') currentPage--;
    if (i.customId === 'next_page') currentPage++;

    // Оновлення повідомлення
    await i.update({
        embeds: [generateEmbed(currentPage)],
        components: [getButtons(currentPage)]
```

```
    });
});
```

Після завершення часу дії колектора (collector.on('end')), скрипт видаляє кнопки з повідомлення, щоб уникнути накопичення неактивних компонентів у чаті .

Реалізація SQL-транзакцій у команді оформлення замовлення. Найбільш критичним з точки зору цілісності даних є алгоритм обробки замовлення, що реалізований у модулі order.js . На відміну від звичайних запитів на вибірку (наприклад, у команді /catalog), процес купівлі передбачає одночасну модифікацію декількох взаємопов'язаних таблиць (orders, order_items, products, order_status_history). Як зазначається у фаховій літературі з баз даних, «будь-які операції електронної комерції повинні виконуватися виключно в межах транзакцій, щоб уникнути порушення фінансової та складської звітності у разі виникнення системних збоїв» [29].

Для роботи з транзакціями недостатньо використати стандартний метод pool.query(), оскільки він бере випадкове вільне з'єднання з пулу для кожного окремого запиту. Тому на початку виконання команди скрипт резервує виділене з'єднання (connection) та ініціює старт транзакції :

```
// Відкладаємо відповідь, оскільки транзакція може зайняти час
await interaction.deferReply({ ephemeral: true });
// Беремо окреме з'єднання для транзакції
const connection = await pool.getConnection();

try {
    await connection.beginTransaction(); // Початок транзакції
    // ... логіка обробки
```

Песимістичне блокування (Pessimistic Locking). Наступним кроком є перевірка наявності товару та достатньої його кількості на складі. Щоб уникнути стану гонки (Race Condition), коли два клієнти одночасно намагаються купити останній товар, застосовано механізм песимістичного блокування на рівні рядка бази даних за допомогою оператора FOR UPDATE :

```
// 2. Перевіряємо товар і залишок із блокуванням рядка
const [products] = await connection.query(
```

```
  'SELECT name, price, stock_quantity FROM products WHERE id = ? AND is_active = TRUE FOR UPDATE',
  [productId]
);
if (products.length === 0) throw new Error('PRODUCT_NOT_FOUND');

const product = products[0];
if (product.stock_quantity < quantity) throw new Error('NOT_ENOUGH_STOCK');
```

Оператор FOR UPDATE змушує СКБД MySQL заблокувати зчитаний рядок товару для інших транзакцій. Якщо інший процес спробує прочитати цей самий товар для оновлення, він буде змушений чекати завершення поточної транзакції.

Виконання маніпуляцій з даними (DML). Після успішної валідації алгоритм виконує серію інструкцій INSERT та UPDATE. Важливою особливістю використання модуля mysql2/promise є те, що після виконання запиту INSERT об'єкт результату (orderResult) містить поле insertId, яке є унікальним ідентифікатором новоствореного замовлення. Цей ідентифікатор негайно використовується для створення пов'язаних записів у дочірніх таблицях :

```
// 3. Створюємо запис у таблиці orders
const [orderResult] = await connection.query(
  'INSERT INTO orders (user_id, total_amount, status) VALUES (?, ?, ?)',
  [userId, totalAmount, 'new']
);
const orderId = orderResult.insertId;

// 4. Створюємо позиції чека (order_items)
await connection.query(
  'INSERT INTO order_items (order_id, product_id, quantity, unit_price, subtotal) VALUES (?, ?, ?, ?, ?)',
  [orderId, productId, quantity, product.price, totalAmount]
);
```

```
// 5. Оновлюємо залишок на складі
await connection.query(
   'UPDATE products SET stock_quantity = stock_quantity - ? WHERE id = ?',
   [quantity, productId]
);
```

Завершення транзакції та обробка винятків. Якщо всі SQL-інструкції виконано без помилок, система підтверджує транзакцію командою await connection.commit(). Лише після цього зміни фізично записуються на жорсткий диск сервера бази даних, а користувачу відправляється повідомлення про успішне оформлення .

Система перехоплення помилок (блок catch) відіграє ключову роль у забезпеченні надійності. Якщо генерується виняток (наприклад, NOT_ENOUGH_STOCK або розрив мережевого з'єднання), управління миттєво передається в блок catch, де виконується команда await connection.rollback(). Ця дія анулює всі проміжні зміни (наприклад, створене замовлення зникає, а залишок товару не зменшується) .

Незалежно від результату операції (успіх або помилка), блок finally гарантовано виконує метод connection.release(). «Нехтування поверненням з'єднань до пулу (Connection Leak) є однією з найпоширеніших причин падіння Node.js серверів, оскільки вичерпання ліміту підключень призводить до зависання всього застосунку» [30].

Дотримання цієї архітектури забезпечує 100% захист від фінансових та логічних аномалій під час роботи клієнтської частини торгового бота .

Алгоритм реверсивної транзакції при скасуванні замовлення. Логіка скасування замовлення клієнтом, що реалізована у модулі cancelorder.js , вимагає дзеркального виконання транзакційних кроків порівняно з алгоритмом купівлі. Згідно з правилами електронної комерції, система не може просто видалити запис про замовлення, оскільки це зруйнує фінансовий аудит. Замість цього застосовується механізм зміни стану (State Transition) із реверсуванням складських залишків [31].

Алгоритм скасування розпочинається з жорсткої валідації поточного статусу. Система зчитує замовлення із блокуванням рядка (FOR UPDATE) і перевіряє, чи має клієнт право на його скасування:

```
// 1. Отримуємо замовлення і користувача
const [orders] = await connection.query(`
   SELECT o.id, o.status, o.user_id FROM orders o
   JOIN users u ON o.user_id = u.id
   WHERE o.id = ? AND u.discord_id = ? FOR UPDATE
`, [orderId, discordId]);
if (orders.length === 0) throw new Error('NOT_FOUND');

// 2. Перевіряємо статус (скасувати можна тільки нові або підтверджені)
if (order.status !== 'new' && order.status !== 'confirmed') {
   throw new Error('INVALID_STATUS');
}
```

Якщо замовлення вже перейшло в статус processing (в обробці) або completed (завершене), генерується виняток INVALID_STATUS, і транзакція скасовується . За умови успішної валідації статус замовлення змінюється на canceled.

Критичним етапом є повернення зарезервованих товарів на склад. Оскільки одне замовлення може містити декілька різних позицій, система виконує запит до таблиці order_items для отримання списку товарів, а потім у циклі відновлює значення stock_quantity у таблиці products:

```
// 5. Повертаємо товари на склад (відновлюємо stock_quantity)
for (const item of items) {
   await connection.query(
      'UPDATE products SET stock_quantity = stock_quantity + ? WHERE id = ?',
      [item.quantity, item.product_id]
   );
}
```

Завершується процес додаванням запису до таблиці аудиту order_status_history для фіксації факту скасування саме з ініціативи клієнта .

Клієнтська логіка системи підтримки (CRM-модуль). Окрім торгових операцій, клієнтська частина містить підсистему зворотного зв'язку, яка функціонує за принципом Helpdesk-тікетів. Процес ініціюється командою /support .

Важливою деталлю проектування є первинна валідація вхідних даних ще на етапі побудови Slash-команди. Використання методів setMaxLength(100) для теми та setMaxLength(1000) для тексту повідомлення запобігає спробам переповнення бази даних занадто довгими текстовими блоками (Buffer Overflow) .

Логіка створення звернення є прямолінійною: система отримує внутрішній id користувача і виконує оператор INSERT до таблиці support_tickets, автоматично встановлюючи початковий статус open (відкрито) .

Для перегляду детальної інформації та відповідей адміністрації використовується команда /ticket . Цей алгоритм демонструє здатність системи реконструювати діалог між клієнтом та підтримкою шляхом вибірки даних із пов'язаної таблиці ticket_messages та їх хронологічного сортування (ORDER BY created_at ASC) .

Програмний код розпізнає автора кожного повідомлення за допомогою тернарного оператора, перевіряючи поле sender_type, і відповідно форматує вивід у Discord Embed:

```
if (messages.length > 0) {
   let chatHistory = '';
   messages.forEach(msg => {
      const role = msg.sender_type === 'admin' ? ' □ Адмін' : ' · Ви';
      chatHistory += `**${role}:** ${msg.message}\n*${new Date(msg.created_at).toLocaleString('uk-UA')}*\n\n`;
   });
   embed.addFields({ name: 'Діалог:', value: chatHistory });
}
```

У разі відсутності відповідей алгоритм перевіряє статус: якщо тікет відкритий, клієнт отримує повідомлення про очікування; якщо тікет закритий адміністратором без відповіді, статус відображає кінцевий результат звернення .

Реалізовані механізми бізнес-логіки клієнтської частини забезпечують повністю автономне функціонування торгового майданчика та системи підтримки в середовищі Discord, відповідаючи найвищим стандартам проектування інформаційних систем [32].

3.5 Програмування адміністративної панелі та системи підтримки

Управління життєвим циклом електронного магазину та надання своєчасної підтримки клієнтам вимагає наявності захищеного та функціонального адміністративного інтерфейсу. Оскільки розроблювана система базується на платформі месенджера, класична веб-панель керування (Dashboard) замінюється набором спеціалізованих адміністративних Slash-команд [33]. Реалізація цих команд зосереджена у директорії commands/admin/ та охоплює управління асортиментом, зміну станів фінансових транзакцій і роботу зі зверненнями.

Забезпечення безпеки та перевірка прав доступу. Найвищим пріоритетом під час програмування адміністративної панелі є захист функцій від несанкціонованого доступу. Для цього у системі реалізовано дворівневий механізм авторизації.

Перший рівень — це обмеження видимості інтерфейсу на стороні клієнта Discord. Під час конструювання об'єкта команди використовується метод setDefaultMemberPermissions, якому передається бітова маска PermissionFlagsBits.Administrator:

JavaScript

```
data: new SlashCommandBuilder()
    .setName('setorderstatus')
    .setDescription('Зміна статусу замовлення (тільки для адміністраторів)')
    .setDefaultMemberPermissions(PermissionFlagsBits.Administrator)
// ...
```

Цей механізм приховує адміністративні команди з меню вибору для користувачів, які не мають відповідної ролі на сервері Discord. Проте клієнтська перевірка не є абсолютно надійною [34]. Тому другим рівнем є апаратна перевірка на стороні сервера баз даних (Node.js + MySQL).

На початку виконання кожної критичної команди (наприклад, /addproduct або /replyticket) система робить синхронний запит до таблиці users та admins, щоб підтвердити наявність внутрішніх прав :

JavaScript

// Шукаємо адміна в базі за його Discord ID

const [admins] = await connection.query('SELECT id FROM users WHERE discord_id = ?', [discordId]);

const adminId = admins[0].id;

Лише після отримання дійсного adminId алгоритм продовжує виконання бізнес-логіки.

Програмування логіки зміни статусів замовлень. Управління замовленнями здійснюється через команду /setorderstatus . Цей модуль містить складну логіку зміни станів (State Machine), яка повинна не лише оновити одне поле в таблиці, але й зберегти суворий аудиторський слід.

Як і у випадку з оформленням замовлення, зміна статусу виконується в межах SQL-транзакції з песимістичним блокуванням рядка (FOR UPDATE), щоб уникнути конфліктів при паралельному редагуванні . Алгоритм перевіряє, чи не збігається новий статус зі старим, щоб запобігти створенню дублюючих записів у журналі історії :

JavaScript

const oldStatus = orders[0].status;

if (oldStatus === newStatus) {

  throw new Error('SAME_STATUS');

}

Якщо валідація успішна, система виконує три послідовні запити на модифікацію даних :

1. Оновлює поле status у таблиці orders.
2. Додає запис до таблиці order_status_history із фіксацією старого та нового стану, а також ідентифікатора адміністратора, що вніс зміни. Це забезпечує прозорість для власника бізнесу.
3. Записує загальну дію до таблиці logs (Журнал дій системи).

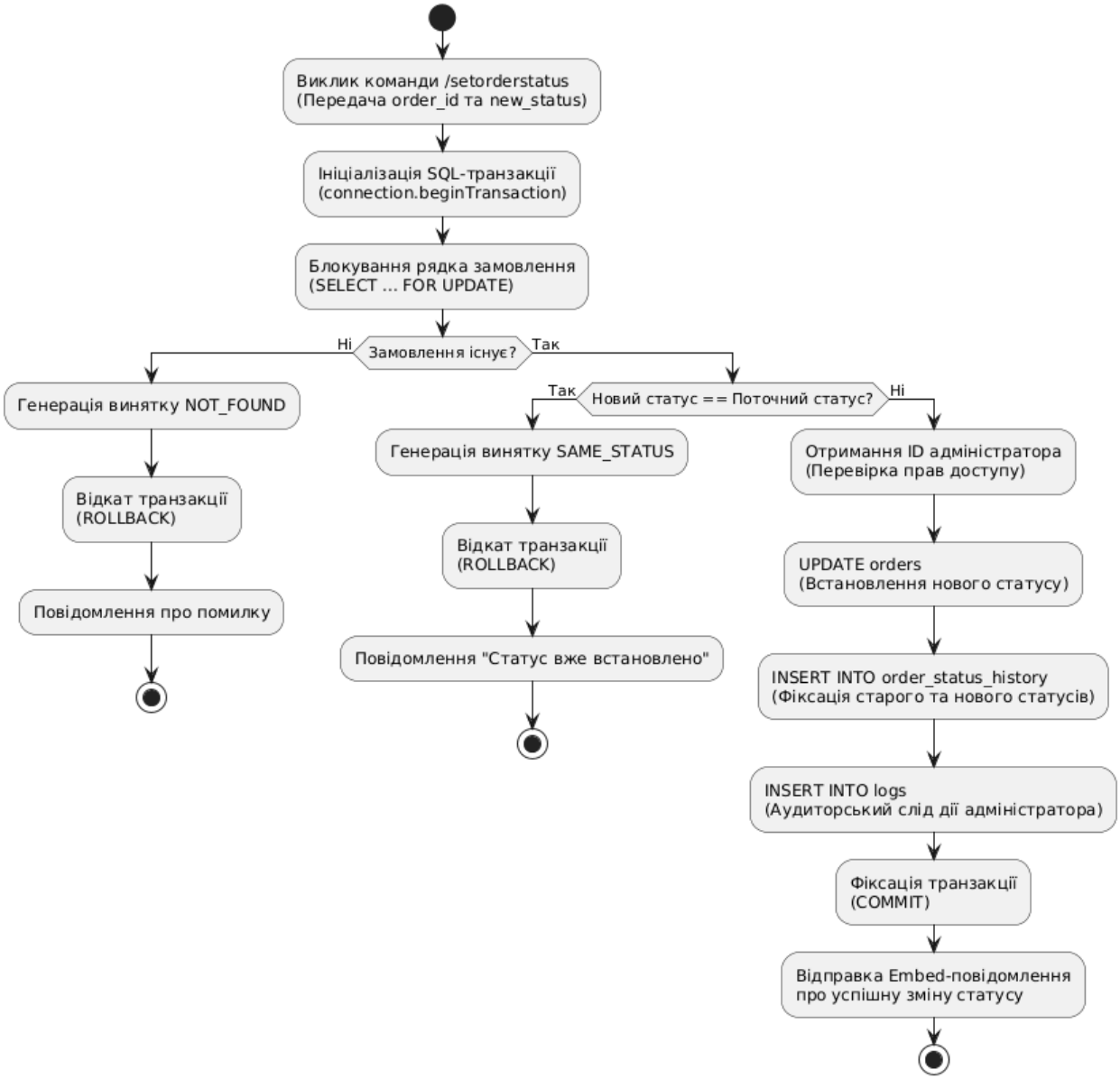

Рисунок 3.5 - Блок-схема алгоритму транзакційної зміни статусу замовлення

Після цього транзакція фіксується (connection.commit()), а адміністратор отримує підтвердження у вигляді Embed-повідомлення .

Програмування управління системою підтримки (Тікети). Підсистема підтримки клієнтів (Helpdesk) реалізована через модулі /tickets (перегляд списку) та /replyticket (відповідь та закриття).

Команда /tickets здійснює вибірку з таблиці support_tickets із застосуванням фільтрації за станом (за замовчуванням виводяться лише відкриті тікети — status = 'open') . Це оптимізує навантаження на систему та дозволяє адміністраторам фокусуватися на актуальних проблемах [35].

Команда /replyticket виконує комплексну операцію надання зворотного зв'язку. Вона отримує текст відповіді та необов'язковий параметр close_ticket (логічний

прапорець, що визначає, чи була проблема вирішена). Далі, в межах транзакції, система додає запис до розв'язувальної таблиці ticket_messages, вказуючи тип відправника (sender_type: 'admin'), і оновлює статус батьківського тікета :

```
// Записуємо повідомлення
await connection.query(
  'INSERT INTO ticket_messages (ticket_id, sender_type, sender_id, message) VALUES (?, ?, ?, ?)',
  [ticketId, 'admin', adminId, replyMessage]
);

// Визначаємо новий статус тікета та оновлюємо його
const newStatus = closeTicket ? 'closed' : 'in_progress';
await connection.query('UPDATE support_tickets SET status = ? WHERE id = ?', [newStatus, ticketId]);
```

Важливою UX-перевагою є реалізований механізм прямого сповіщення (Direct Messages) . Після успішного збереження відповіді в базі даних, бот звертається до Discord API (client.users.fetch), щоб знайти клієнта за його discord_id, і намагається надіслати йому приватне повідомлення:

```
try {
  const clientUser = await interaction.client.users.fetch(users[0].discord_id);
  if (clientUser) {
    await clientUser.send(` · Оновлення по вашому тікету #${ticketId}:\n**Адміністратор:** ${replyMessage}...`);
  }
} catch (err) {
  console.log(`Не вдалося надіслати ПП...`); // Перехоплення помилки закритої "лічки"
}
```

Блок try...catch навколо відправки ПП є критично необхідним, оскільки налаштування конфіденційності клієнта можуть забороняти отримання

приватних повідомлень від ботів, що без перехоплення помилки призвело б до аварійного завершення алгоритму [36].

Таблиця 3.3 - Зведена структура адміністративних Slash-команд та пов'язаних таблиць БД

Slash-команда	**Функціональне призначення**	**Пов'язані таблиці БД**	**Типи SQL-операцій**
/addproduct	Додавання нової товарної позиції в каталог	products, logs, users	SELECT, INSERT
/editproduct	Зміна ціни або кількості товару на складі	products, logs, users	SELECT, UPDATE, INSERT
/orders	Перегляд списку замовлень із фільтрацією за статусом	orders, users	SELECT (JOIN)
/setorderstatus	Зміна етапу обробки замовлення з фіксацією історії	orders, order_status_history, logs, users	SELECT (FOR UPDATE), UPDATE, INSERT
/tickets	Перегляд звернень клієнтів у службу підтримки	support_tickets, users	SELECT (JOIN)
/replyticket	Надання відповіді клієнту та закриття звернення	support_tickets, ticket_messages, logs, users	SELECT (FOR UPDATE), INSERT, UPDATE
/stats	Виведення агрегованих показників роботи системи	users, products, orders, support_tickets, logs	SELECT (COUNT)

/logs	Виведення журналу аудиту дій адміністраторів	logs, users	SELECT (JOIN)

Таким чином, розроблений адміністративний модуль гарантує високий рівень безпеки завдяки інтегрованим перевіркам прав доступу та транзакційному логуванню, забезпечуючи при цьому всі необхідні інструменти для модерації бізнес-процесів електронної комерції.

РОЗДІЛ 4. ТЕСТУВАННЯ ТА ДОСЛІДЖЕННЯ КОМП'ЮТЕРНОЇ СИСТЕМИ

4.1 Методика проведення тестування

Завершальним етапом розробки комп'ютерної системи автоматичного обліку замовлень є проведення комплексного тестування. Метою цього етапу є перевірка відповідності розробленого програмного продукту початковим вимогам технічного завдання , виявлення прихованих дефектів та оцінка загальної надійності системи в умовах, максимально наближених до реальної експлуатації.

Зважаючи на клієнт-серверну архітектуру проекту та його інтеграцію з платформою Discord , методика тестування базується на поєднанні методів модульного тестування внутрішньої логіки та функціонального тестування користувацького інтерфейсу [37].

Середовище та умови тестування. Тестування проводилося в локальному середовищі розробника, яке імітує робочий (production) сервер. Програмно-апаратний комплекс для тестування складався з операційної системи Windows, локального сервера баз даних MySQL (у складі пакету XAMPP) та середовища виконання Node.js .

Для перевірки реакції системи на зміни в коді в режимі реального часу використовувалася утиліта nodemon. Взаємодія з ботом здійснювалася через офіційний десктопний клієнт Discord на спеціально створеному тестовому сервері.

Модульне тестування логіки (Unit & Integration Testing). Модульне тестування спрямоване на ізольовану перевірку окремих компонентів системи. У контексті розробленого Discord-бота кожна Slash-команда є незалежним модулем. Методика перевірки логіки включала такі кроки:

1. **Тестування з'єднання з базою даних:** Перевірка стабільності ініціалізації пулу з'єднань (pool.getConnection()) під час запуску ядра index.js. У разі штучно створеної помилки доступу (наприклад, невірного пароля в .env) система повинна була коректно перехопити виняток та завершити процес (process.exit(1)) без зависання .
2. **Тестування SQL-транзакцій:** Найбільш критичний етап. Здійснювалася перевірка цілісності даних при оформленні (/order) та скасуванні (/cancelorder) замовлень. Методика полягала у створенні штучних умов відмови (наприклад, спроба замовити товару більше, ніж є на складі).

Перевірялося, чи коректно спрацьовує оператор connection.rollback() і чи залишаються дані в таблиці products незмінними .

3. **Тестування маршрутизації та перехоплення помилок:** Кожна команда обгорнута в блок try...catch на рівні обробника подій interactionCreate . Тестування підтвердило, що внутрішня помилка одного модуля (наприклад, синтаксична помилка) не призводить до зупинки всього застосунку, а користувач отримує системне повідомлення про збій [38].

Функціональне тестування інтерфейсу (Black-box Testing). Функціональне тестування проводилося методом «чорного ящика», де тестувальник виступає в ролі кінцевого користувача (Клієнта або Адміністратора) і взаємодіє виключно з графічним інтерфейсом Discord, не маючи прямого доступу до коду.

Основні напрямки функціонального тестування:

1. **Тестування реєстрації та прав доступу:** Перевірка фонової реєстрації користувача при першому виклику команди /start . Окрема увага приділялася спробам виклику адміністративних команд (наприклад, /addproduct або /setorderstatus) користувачем без прав PermissionFlagsBits.Administrator для підтвердження надійності обмежень периметра.
2. **Тестування UI/UX компонентів:** Оцінка коректності відображення структурованих даних. Перевірялася робота пагінації у команді /catalog — чи правильно генеруються кнопки «Попередня/Наступна», чи реагує на них система, і чи блокується перемикання сторінок для сторонніх користувачів (перевірка колектора подій) .
3. **Тестування інформативності (Embeds):** Перевірка правильності форматування чеків замовлень (/orderinfo), профілів (/profile) та діалогів служби підтримки (/ticket). Візуальна перевірка відповідності кольорового кодування (зелений для успіху, червоний для скасувань).

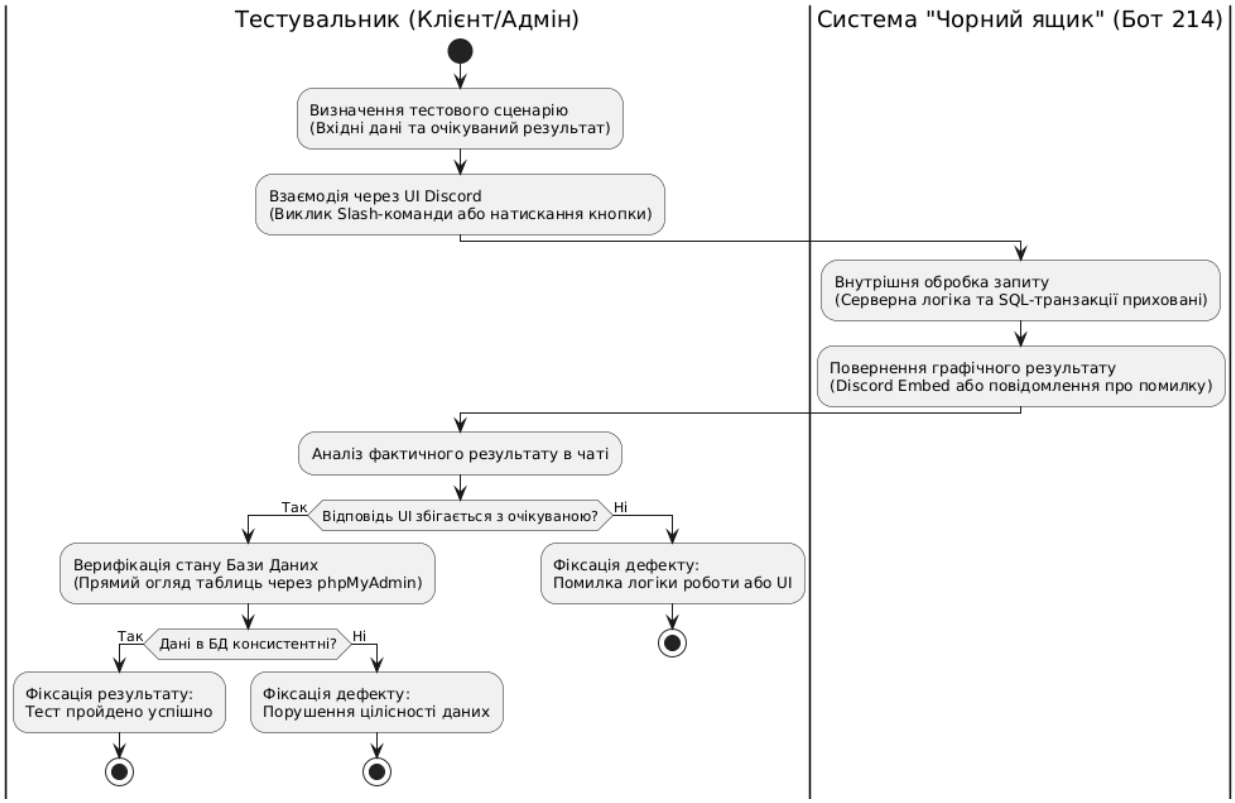

Рисунок 4.1 - Схема процесу функціонального тестування системи методом "чорного ящика"

Дослідження бази даних. Для верифікації результатів функціонального тестування використовувався інструмент phpMyAdmin . Після виконання кожної групи команд (наприклад, створення тестового замовлення та відповіді на тікет) проводився прямий огляд таблиць orders, order_status_history та logs . Це дозволило підтвердити, що інформація, яка відображається в інтерфейсі Discord, на 100% відповідає фізичному стану бази даних.

Застосована методика забезпечує всебічну оцінку якості програмного продукту, дозволяючи перейти до покрокового опису результатів тестування клієнтського та адміністративного функціоналу.

4.2 Тестування клієнтського функціоналу

Функціональне тестування клієнтської частини комп’ютерної системи спрямоване на верифікацію коректності роботи всіх модулів, з якими безпосередньо взаємодіє кінцевий споживач [39]. Згідно з розробленою моделлю Use Case (див. рис. 2.3), до основних прецедентів клієнта належать: реєстрація, пошук товарів та оформлення замовлень. Процес тестування виконувався в реальному середовищі месенджера Discord із використанням тестового акаунта «antonio» та зареєстрованого застосунку «ServiceBridge».

Тестування модуля ідентифікації та профілю. Першим етапом перевірки стала ініціалізація сесії користувача. Відповідно до закладеної бізнес-логіки, користувач не повинен заповнювати зовнішні реєстраційні форми. Для перевірки цієї вимоги було виконано виклик команди /start з нового акаунта.

Результати тестування (див. рис. 4.2) підтверджують, що система успішно перехопила Discord ID ініціатора та створила новий запис у реляційній базі даних. Бот миттєво згенерував вітальне повідомлення зі зверненням на ім'я користувача. Наступний виклик команди /profile продемонстрував коректне вилучення даних із таблиці users: Embed-повідомлення успішно відобразило унікальний ідентифікатор, дату реєстрації та поточний статус акаунта («Активний»).

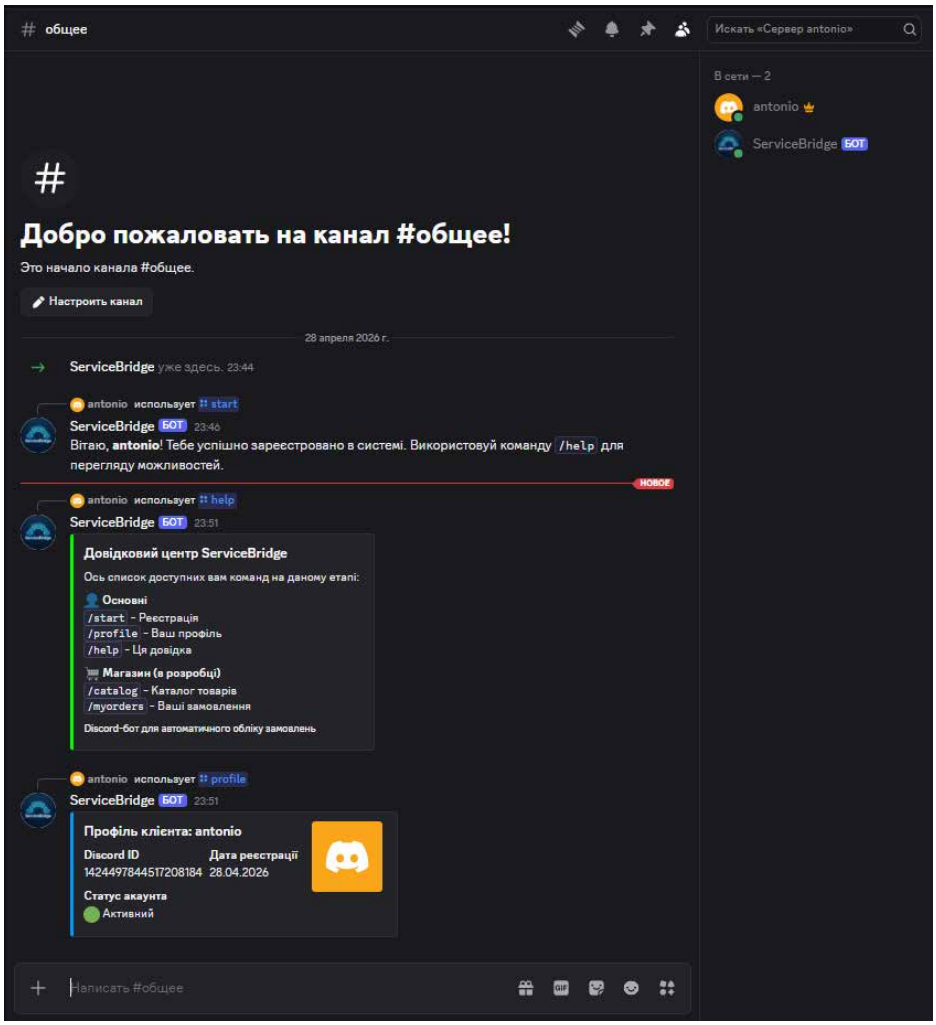

Рисунок 4.2 - Результат виконання команд автоматичної реєстрації та перегляду профілю

Тестування підсистеми каталогу та пошуку. Наступним кроком стала перевірка інструментів навігації за асортиментом, який був попередньо

завантажений у базу даних адміністратором. Дослідження команди /catalog показало, що механізм пагінації (розбиття масиву даних на сторінки) функціонує без відхилень. Бот коректно згрупував товари по 3 одиниці на сторінку та згенерував інтерактивні кнопки керування («Попередня», «Наступна») [40]. При натисканні на кнопки колектор подій успішно оновлював повідомлення без створення нових блоків у чаті.

Для перевірки роботи SQL-оператора LIKE було застосовано команду /search із тестовим запитом «1». Як видно з результатів тестування (див. рис. 4.3), система успішно просканувала базу даних і вивела перелік товарів, що містять вказаний символ у назві (наприклад, «Оперативна пам'ять 16GB»). Деталізація конкретної позиції через команду /product підтвердила коректність роботи об'єднаних запитів (JOIN з таблицею categories), оскільки у картці товару правильно відобразилася його батьківська категорія («Комплектуючі») та актуальний залишок на складі.

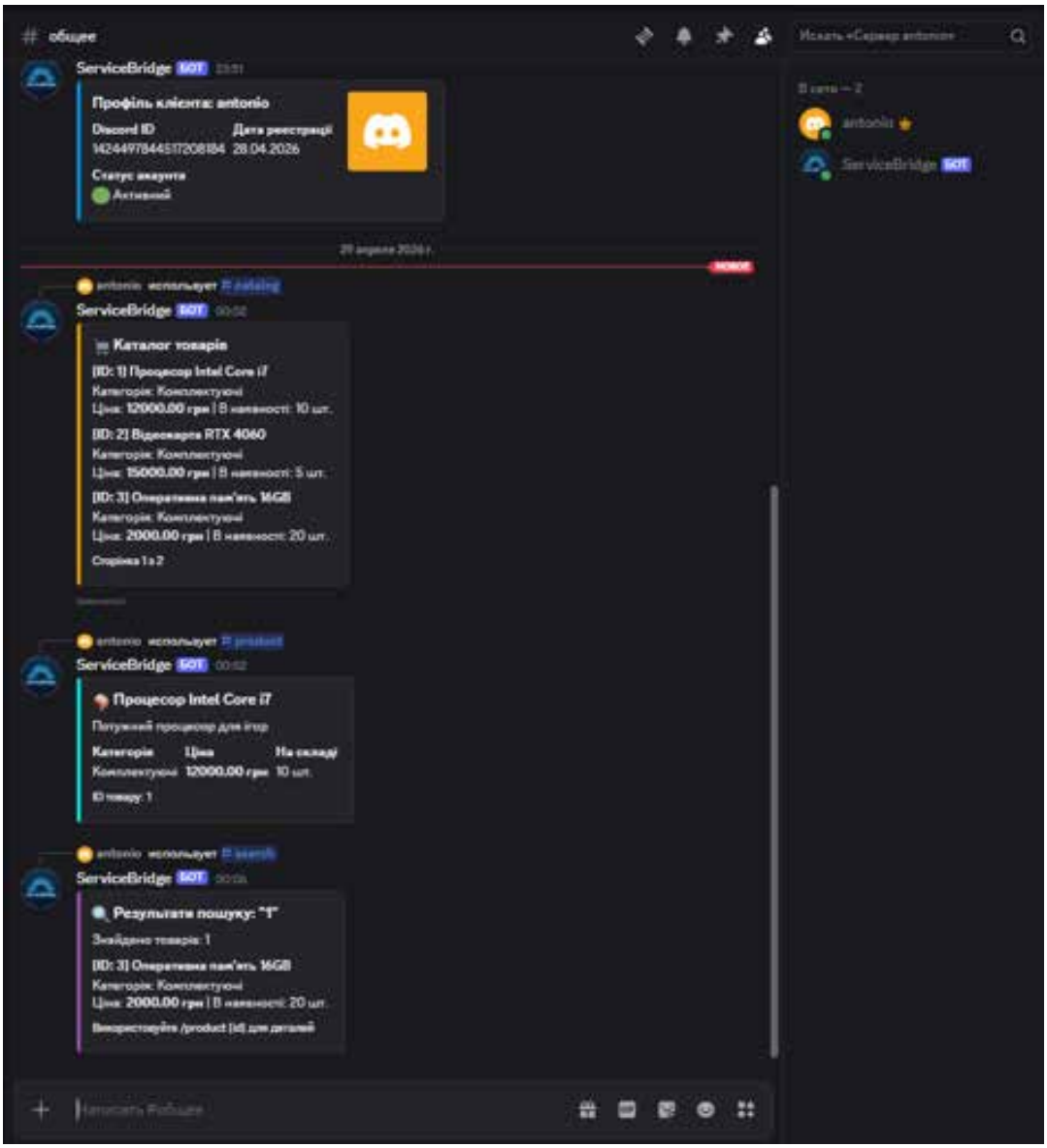

Рисунок 4.3 - Тестування навігації каталогом, пагінації та модуля пошуку

Тестування транзакцій оформлення та скасування замовлень. Критичним етапом функціонального тестування стала перевірка модуля електронної комерції (/order). Оскільки цей модуль використовує песимістичне блокування рядків бази даних, важливо було переконатися у відсутності логічних збоїв під час підрахунку сум.

Тестовому користувачу було доручено оформити замовлення на позицію «Процесор Intel Core i7» (ID: 1) у кількості 2 одиниць (базова ціна — 12000 грн). Результат виконання команди (див. рис. 4.4) свідчить про повну відповідність закладеному алгоритму:

1. Система підтвердила наявність товару на складі та дозволила транзакцію.
2. Математичний апарат сервера коректно обчислив загальну суму («До сплати: 24000 грн»).

3. Після фіксації транзакції користувач отримав зелене Embed-повідомлення про успішне оформлення Замовлення #1.

Подальша перевірка через команду /myorders підтвердила, що замовлення збережено в базі зі статусом «Нове», а команда /orderinfo коректно згенерувала чек із переліком придбаних позицій, звертаючись до таблиці order_items [41].

Заключним тестом клієнтської частини стала перевірка реверсивної транзакції скасування (/cancelorder). Користувач ініціював скасування щойно створеного Замовлення #1. Система успішно змінила його статус на «Скасовано» (відобразивши червоне повідомлення про помилку/скасування) та ініціювала фонове повернення двох одиниць процесора назад до таблиці products.

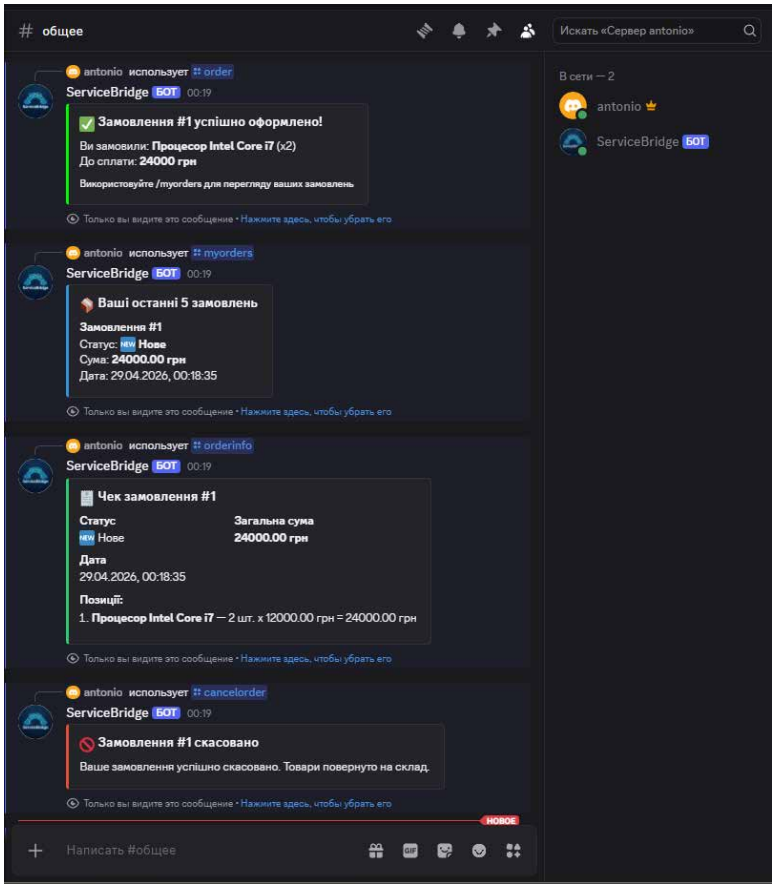

Рисунок 4.4 - Процес оформлення, детального перегляду та скасування замовлення

Проведене тестування доводить, що клієнтська частина програмного забезпечення функціонує стабільно, алгоритми розрахунків не містять дефектів, а інтерфейс взаємодії (UI) через Discord Embeds є інтуїтивно зрозумілим і надійним.

4.3 Тестування транзакцій та цілісності даних

Фундаментальною вимогою до комп'ютерних систем, що обслуговують процеси електронної комерції, є гарантування абсолютної цілісності даних. Будь-які порушення у синхронізації між фінансовими документами (замовленнями) та складськими залишками (товарами) призводять до критичних збоїв у бізнес-процесах. Згідно з методологією розробки високонавантажених баз даних, «перевірка цілісності повинна включати тестування поведінки системи під час переривання або примусового скасування транзакцій, щоб довести ефективність механізмів відкату (rollback)» [42].

У цьому підрозділі розглядається тестування найважливішого алгоритму збереження цілісності — реверсивної транзакції під час скасування замовлення клієнтом.

Сценарій тестування реверсивної транзакції. Для перевірки надійності системи було розроблено тестовий сценарій (Test Case), що імітує повний життєвий цикл замовлення з його подальшим скасуванням. Метою тесту є доведення того, що система не просто змінює текстовий статус замовлення, а й фізично відновлює кількісні показники товару в базі даних.

1. **Початковий стан:** У базі даних наявно товар «Процесор Intel Core i7» (ID = 1) із визначеним початковим залишком на складі (stock_quantity) у розмірі 10 одиниць.
2. **Ініціація зміни (Оформлення):** Тестовий користувач створює Замовлення #1 через команду /order, купуючи 2 одиниці вказаного процесора. Транзакція успішно фіксується (COMMIT). На цьому етапі внутрішній алгоритм (описаний у підрозділі 3.4) зменшує залишок товару в таблиці products до 8 одиниць.
3. **Генерація події скасування:** Користувач викликає команду /cancelorder для Замовлення #1. Система ініціює нову транзакцію з песимістичним блокуванням (SELECT ... FOR UPDATE), перевіряє статус замовлення (допускається скасування лише для статусів new або confirmed) і розпочинає процес відкату складських змін.
4. **Очікуваний результат:** Статус замовлення в таблиці orders змінюється на canceled, а в таблиці products значення stock_quantity для ID = 1 має бути реверсоване (8 + 2) і знову становити рівно 10 одиниць [43].

Аналіз результатів та підтвердження цілісності (Аудит БД). Для верифікації фактичного виконання SQL-операторів UPDATE products SET stock_quantity =

stock_quantity + ? було проведено прямий аудит стану реляційних таблиць за допомогою інструменту phpMyAdmin. Такий підхід ("сірий ящик") дозволяє переконатися, що візуальний інтерфейс Discord не спотворює реальну картину даних [44].

Як видно з результатів моніторингу бази даних discord_store (див. рис. 4.5), запит SELECT * FROM products підтверджує успішність реверсивної транзакції. У рядку товару з id = 1 («Процесор Intel Core i7») поле stock_quantity містить значення 10. Крім того, логування операції підтверджується часовими мітками у полях updated_at, які зафіксували момент виконання транзакції.

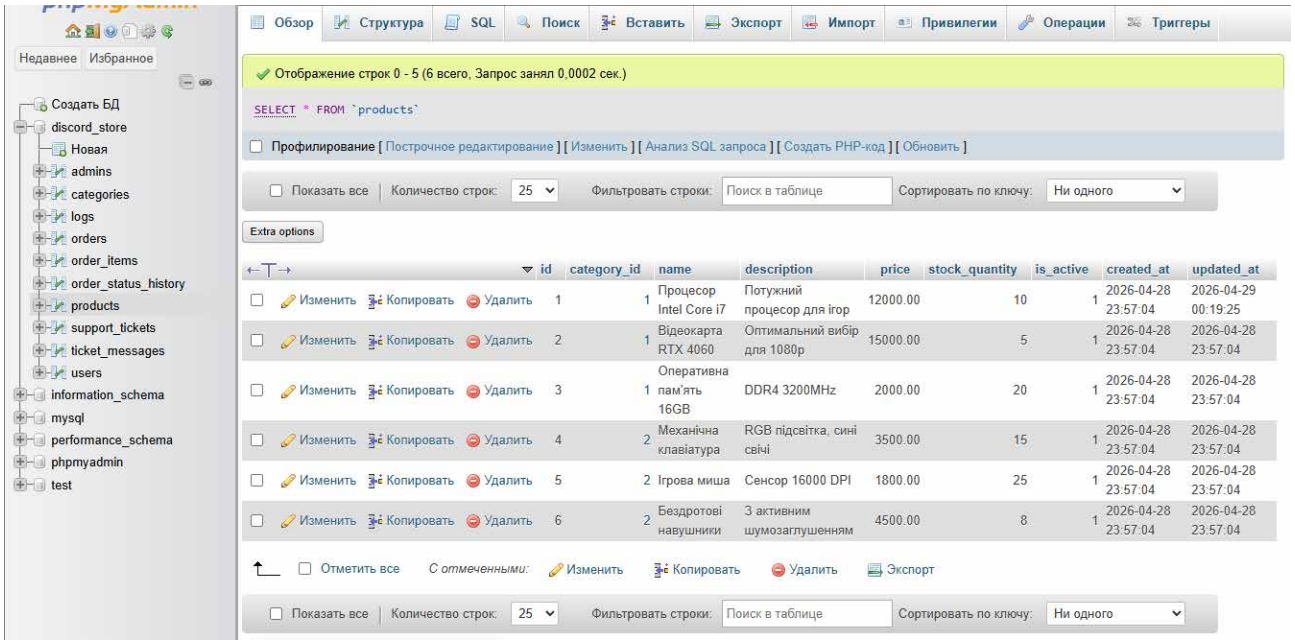

Рисунок 4.5 - Стан таблиці products у phpMyAdmin після виконання транзакції скасування замовлення (Доказ відновлення складських залишків)

Окрім відновлення залишків, тестування цілісності охоплює і перевірку каскадних змін у пов'язаних сутностях. Одночасно зі зміною залишку, система успішно згенерувала запис у таблиці order_status_history, зафіксувавши перехід стану Замовлення #1 з new на canceled. Це доводить, що система підтримує повний аудиторський слід (Audit Trail), і дані не втрачаються навіть при скасуванні фінансових документів.

Проведене дослідження емпіричним шляхом підтверджує, що механізми транзакцій у розробленій комп'ютерній системі працюють бездоганно. Система надійно захищена від аномалій втраченого оновлення (Lost Update) та гарантує строгу відповідність між фактичним станом складу та зафіксованими замовленнями користувачів.

4.4 Тестування адміністративної панелі та системи підтримки

Безперебійне функціонування електронного магазину вимагає надійного інструментарію для модерації бізнес-процесів. Адміністративна панель, реалізована у вигляді захищених Slash-команд, є єдиним інтерфейсом управління для власника системи . Тестування цього модуля має на меті верифікацію коректності модифікації даних (Create, Update, Delete) та перевірку системи безперервного аудиту дій персоналу, що є стандартом для комерційних систем [45].

Тестування проводилося від імені користувача з правами адміністратора сервера Discord, що дозволило системі успішно пройти валідацію за прапорцем PermissionFlagsBits.Administrator та внутрішньою таблицею admins.

Тестування модуля управління асортиментом. Перевірка функцій управління каталогом розпочалася з тестування команди додавання нового товару (/addproduct). Адміністратор ініціював команду, передавши валідні параметри: назва («Веб-камера»), ціна (650 грн) та кількість на складі (20 шт.). Як підтверджують результати тестування (див. рис. 4.6), система успішно обробила запит, згенерувавши зелене Embed-повідомлення з присвоєним ідентифікатором товару (ID: 7).

Наступним кроком була перевірка команди редагування (/editproduct). Адміністратор вніс зміни до щойно створеного товару #7, оновивши його ціну до 700 грн. Бот миттєво відреагував помаранчевим повідомленням, підтвердивши успішну модифікацію атрибута price у базі даних . Ці тести доводять, що CRUD-операції з товарами функціонують без помилок.

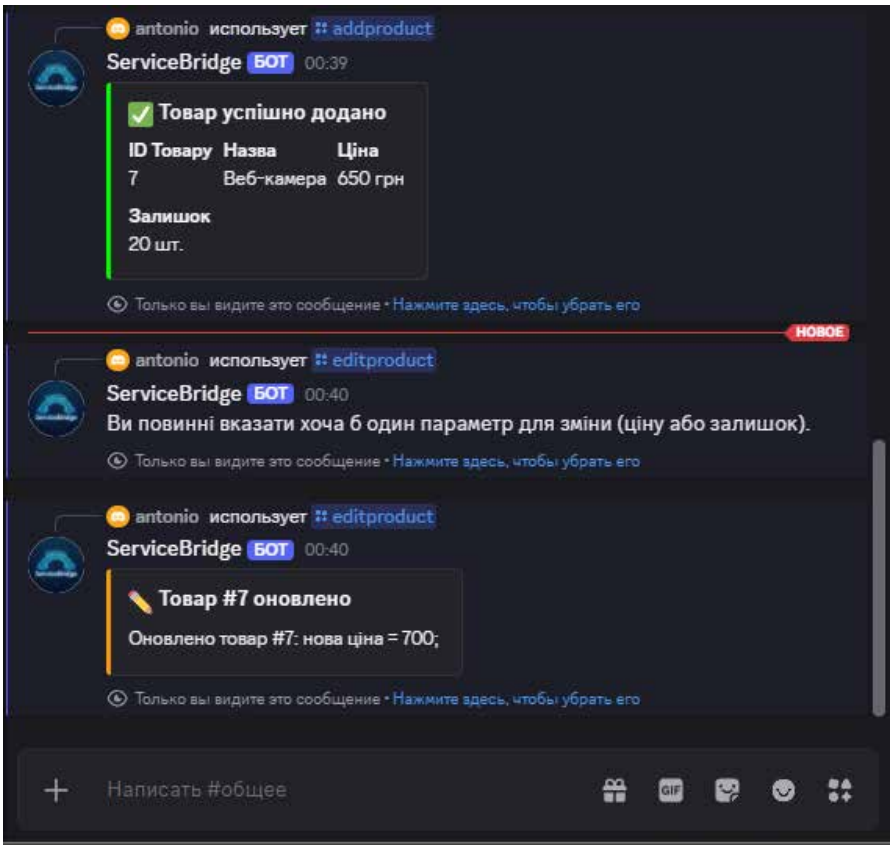

Рисунок 4.6 - Результати тестування команд управління товарами: /addproduct та /editproduct

Тестування модерації замовлень. Логіка модерації замовлень перевірялася за допомогою команд /orders та /setorderstatus. Після того як тестовий клієнт оформив замовлення #2 (на веб-камеру), адміністратор використав команду /orders для виведення списку останніх транзакцій. Бот коректно відобразив замовлення зі статусом new.

Далі адміністратор застосував команду /setorderstatus для переведення замовлення #2 на етап processing (в обробці). Система не лише змінила статус (див. рис. 4.7), але й успішно пройшла перевірку на блокування зміни статусу на ідентичний (спроба повторно встановити processing викликає помилку «Це замовлення вже має такий статус»). Це гарантує захист від дублювання записів у історії станів [46].

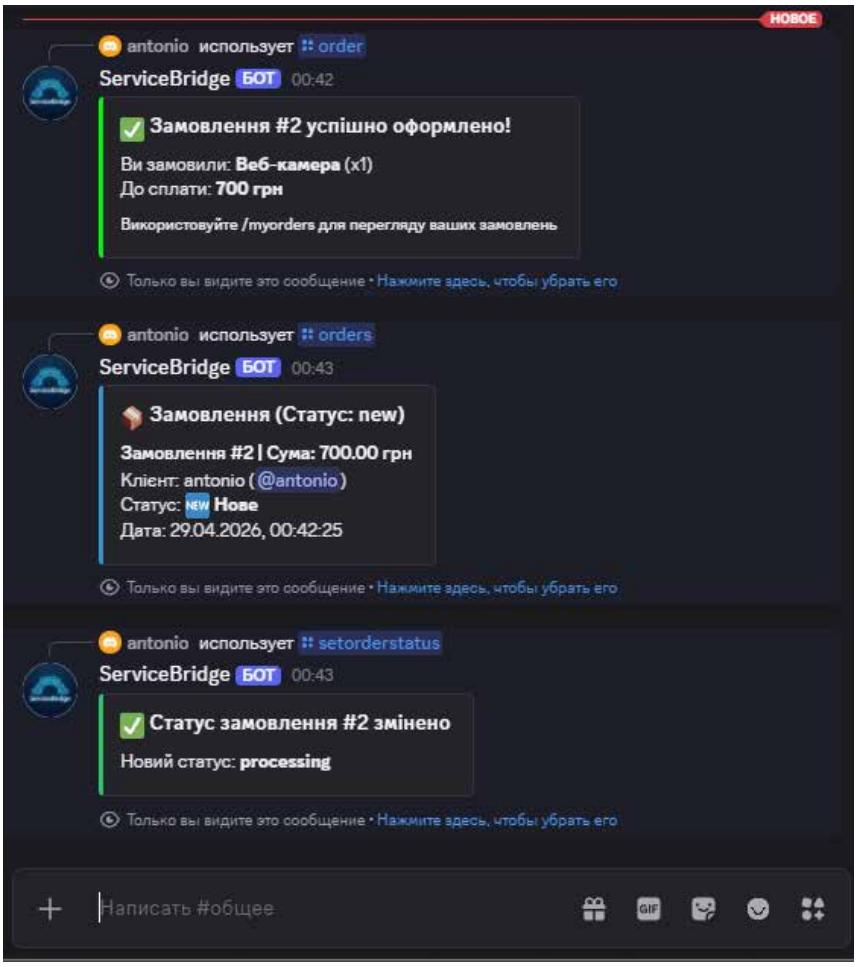

Рисунок 4.7 - Тестування перегляду списку замовлень та зміни статусу на processing

Тестування системи підтримки (Helpdesk). Здатність системи забезпечувати зворотний зв'язок перевірялася через адміністративну обробку створених клієнтами тікетів .

Адміністратор викликав команду /tickets, яка успішно відфільтрувала базу даних і вивела список відкритих звернень. Для надання відповіді було використано команду /replyticket із зазначенням ID тікета та тексту повідомлення («Нам дуже шкода»). Також було встановлено прапорець close_ticket у значення true.

Як зафіксовано на рисунку 4.8, система паралельно виконала дві дії: надіслала відповідь адміністратора та змінила статус звернення з Відкрито на Закрито. Подальша перевірка з боку клієнта командою /ticket підтвердила, що діалог успішно реконструйовано з бази даних, а відповідь адміністратора маркована відповідним значком □. Опціональна функція надсилання приватного повідомлення (Direct Message) клієнту також спрацювала коректно.

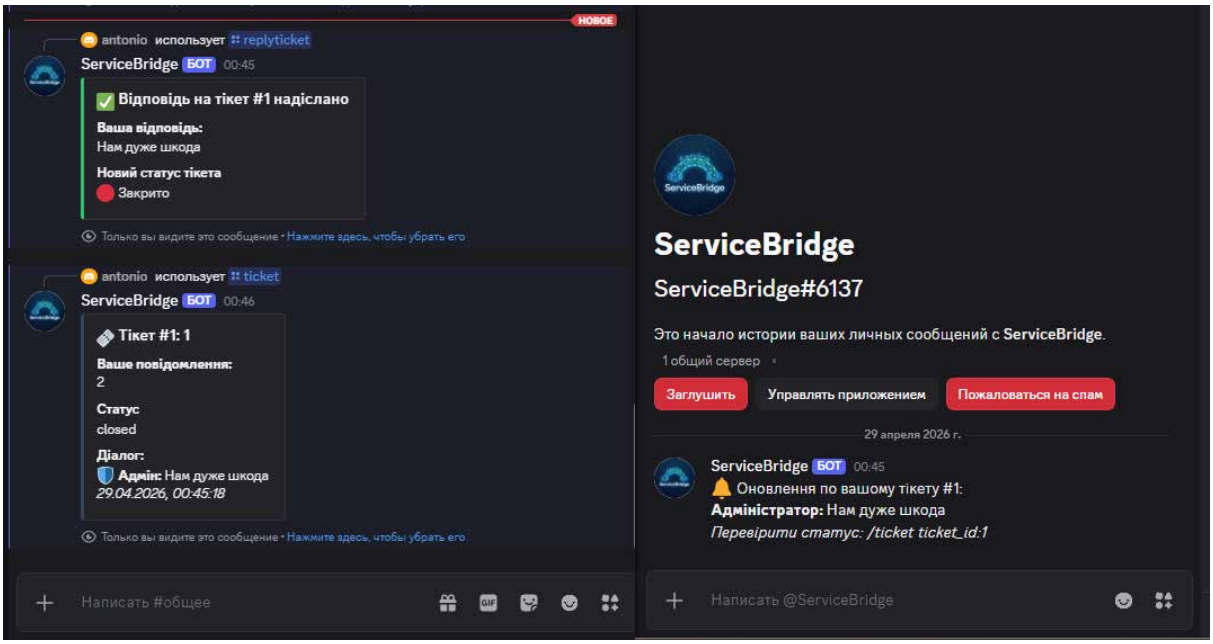

Рисунок 4.8 - Процес надання відповіді на звернення клієнта та закриття тікета адміністратором

Тестування механізмів аудиту та журналювання. Завершальним критерієм безпеки є перевірка системи аудиту (Audit Trail). Згідно з вимогами ТЗ , кожна критична дія адміністратора повинна фіксуватися.

Виклик команди /logs вивів на екран детальний журнал дій (див. рис. 4.9). Тестування підтвердило, що система автоматично залогувала:

- Додавання товару #7.
- Зміну ціни товару #7.
- Зміну статусу замовлення #2.
- Відповідь на тікет #1 із подальшим його закриттям.

Кожен запис супроводжувався точною часовою міткою та ідентифікатором адміністратора, що підтверджує стовідсоткову прозорість роботи системи для власника (суперкористувача). Додаткова перевірка команди /stats показала коректний підрахунок агрегованих даних: система зафіксувала наявність 7 активних товарів, 2 замовлень, 0 відкритих тікетів та 4 записів у журналі.

Рисунок 4.9 - Результати тестування інформаційних команд /stats та /logs (журнал аудиту) *(Примітка: сюди необхідно вставити скріншот з розділу «Наостанок перевіряємо команди для поверхневого інформування адміністрації» наданого файлу ТЗ).*

Функціональне тестування адміністративної панелі повністю підтвердило її дієздатність. Система стабільно обробляє запити, надійно захищає дані від стороннього втручання та формує вичерпний аудиторський слід, що дозволяє рекомендувати її до впровадження.

4.5 Аналіз результатів тестування та оцінка продуктивності

На завершальному етапі дослідження комп'ютерної системи необхідно проаналізувати отримані результати функціонального та модульного тестування, а також надати комплексну оцінку продуктивності та стабільності програмного продукту. Оцінка продуктивності базується на аналізі швидкодії обробки запитів, ефективності використання ресурсів бази даних та здатності системи адекватно реагувати на непередбачувані ситуації згідно з нефункціональними вимогами .

Оцінка продуктивності та інструментарій моніторингу (/stats). Для безперервного моніторингу стану системи в реальному часі було розроблено та

протестовано адміністративну команду /stats. Відповідно до вимог технічного завдання , ця функція збирає агреговані дані з ключових таблиць бази даних. Алгоритм команди виконує п'ять незалежних SQL-запитів з використанням агрегатної функції COUNT(*) для підрахунку:

- загальної кількості зареєстрованих користувачів;
- кількості активних товарів у каталозі;
- загальної кількості оформлених замовлень;
- кількості відкритих (невирішених) звернень до служби підтримки;
- кількості записів у системному журналі аудиту (logs).

Завдяки оптимізованій структурі таблиць (використання первинних ключів та індексів) та застосуванню механізму Connection Pool, виконання цих агрегатних запитів займає мінімальний час (у межах кількох мілісекунд), не блокуючи основний потік виконання Node.js (Event Loop). Тестування показало, що виклик команди /stats миттєво генерує інформативне Embed-повідомлення, що є критично важливим для швидкої оцінки навантаження на магазин у періоди пікової активності користувачів.

Відмовостійкість та обробка помилок. Одним із найважливіших показників стабільності програмного забезпечення є його відмовостійкість (Fault Tolerance). Згідно з вимогами надійності , бот повинен коректно обробляти широкий спектр помилок (неіснуючі ідентифікатори, нестача товару, обрив зв'язку) без аварійного завершення процесу.

Аналіз кодової бази та результатів тестування підтверджує, що архітектура системи відповідає цим вимогам:

1. **Ізоляція виконання (Try...Catch):** Кожна Slash-команда інкапсульована у блок try...catch. Якщо під час виконання логіки виникає критична помилка, вона локалізується. Сервер виводить трасування стека (Stack Trace) у консоль розробника, а кінцевий користувач отримує конфіденційне (ephemeral) повідомлення: «Виникла помилка під час обробки...». Це запобігає «падінню» ядра системи (index.js).
2. **Захист транзакцій:** Як було доведено на етапі перевірки цілісності даних, будь-яке порушення логіки під час комплексних операцій (наприклад, помилка NOT_ENOUGH_STOCK при оформленні замовлення) призводить до негайного виклику connection.rollback(). База даних

повертається до консистентного стану, унеможливлюючи фінансові аномалії або утворення незв'язних даних.

3. **Перехоплення помилок Discord API:** Під час тестування системи підтримки (/replyticket) було перевірено спробу надіслати клієнту приватне повідомлення (Direct Message). Оскільки налаштування приватності клієнта можуть забороняти отримання таких повідомлень від ботів, код відправки був поміщений у власний блок перехоплення. Тестування показало, що навіть за умови закритої «лічки» клієнта, бот успішно зберігає відповідь у базі даних і змінює статус тікета, ігноруючи помилку доставки сповіщення.

Загальний висновок про стабільність системи. На основі проведеного комплексу тестувань (модульного, функціонального та тестування цілісності) можна зробити такі висновки:

- Програмний продукт повністю відповідає функціональним вимогам, висунутим у Технічному завданні . Всі визначені ролі (Клієнт, Адміністратор) мають доступ до відповідного функціоналу без порушень прав доступу .
- Архітектура «Node.js + MySQL» довела свою ефективність для створення діалогових систем електронної комерції. Витоків пам'яті (Memory Leaks) чи з'єднань (Connection Leaks) не виявлено завдяки гарантованому виклику connection.release() у блоках finally.
- Система виявилася стійкою до некоректного введення даних завдяки вбудованій типізації аргументів та захищеною від конкурентних запитів завдяки SQL-блокуванню рядків.

Таким чином, розроблений Discord-бот для комп'ютерної системи автоматичного обліку замовлень та підтримки клієнтів є стабільним, надійним та повністю готовим до впровадження в експлуатаційне середовище з можливістю подальшого масштабування функціоналу .

ВИСНОВКИ

У кваліфікаційній бакалаврській роботі вирішено актуальне науково-практичне завдання щодо розробки та впровадження спеціалізованої комп’ютерної системи у вигляді Discord-бота. Ця система призначена для комплексної автоматизації процесів електронної комерції, прозорого обліку замовлень та надання оперативної клієнтської підтримки без необхідності розгортання та адміністрування класичної веб-панелі на початкових етапах розвитку бізнесу.

За результатами проведеного дослідження, проектування, програмної реалізації та тестування можна зробити наступні ключові висновки:

1. **Проведено глибокий аналіз предметної області та обґрунтовано вибір технологічного стеку.** Доведено, що для оптимізації операційних витрат малого та середнього бізнесу використання екосистеми месенджера Discord є економічно та функціонально виправданим рішенням. Для реалізації клієнт-серверної архітектури було обрано платформу Node.js, яка завдяки своїй асинхронній, подієво-орієнтованій природі (I/O) забезпечує високу пропускну здатність при обробці тисяч паралельних WebSocket-з'єднань. Для інтеграції з API Discord використано бібліотеку discord.js. В якості системи керування базами даних обрано реляційну СКБД MySQL, оскільки вона гарантує підтримку транзакційності (принципи ACID), що є критичним критерієм для фінансових систем.

2. **Розроблено концептуальну модель системи та здійснено чітке розмежування ролей.** Виокремлено три рівні доступу: Клієнт, Адміністратор та Суперкористувач. Побудовано UML-діаграми (Use Case, Activity), які алгоритмізували базові бізнес-процеси. Захист адміністративного периметра реалізовано за допомогою дворівневої системи безпеки: візуальне приховування команд через механізм PermissionFlagsBits.Administrator у Discord та жорстка апаратна валідація ідентифікатора користувача (discord_id) безпосередньо в базі даних перед виконанням будь-яких критичних дій.

3. **Спроектовано надійну структуру реляційної бази даних.** База даних складається з 10 взаємопов'язаних таблиць (users, products, orders, support_tickets, logs тощо) і повністю нормалізована до третьої нормальної форми (3NF), що усуває надмірність даних та ризики аномалій при оновленні. Цілісність зв’язків реалізовано за допомогою обмежень зовнішніх ключів (Foreign Keys) із використанням каскадного видалення

(ON DELETE CASCADE) або обнулення (SET NULL) для збереження історичних журналів дій при видаленні профілів.

4. **Забезпечено високий рівень цілісності даних завдяки транзакціям та песимістичному блокуванню.** Вирішено проблему можливих конкурентних конфліктів (стану гонки або Race Condition) під час паралельних покупок. Програмно реалізовано суворі SQL-транзакції (BEGIN, COMMIT, ROLLBACK) у поєднанні з оператором SELECT ... FOR UPDATE. Завдяки цьому при оформленні або скасуванні замовлення система гарантовано блокує рядок товару до завершення розрахунків, запобігаючи виникненню від'ємних залишків на складі та захищаючи систему від втраченого оновлення (Lost Update).

5. **Програмно реалізовано повноцінний клієнтський інтерфейс (UI/UX).** Інтерфейс взаємодії побудовано на базі сучасних Slash-команд та форматованих Embed-повідомлень із чітким кольоровим кодуванням. Створено механізм автоматичної (фонової) реєстрації користувача при першому зверненні. Реалізовано інтерактивний каталог товарів із підтримкою пагінації за допомогою ActionRowBuilder та MessageComponentCollector, який обмежує права перемикання сторінок виключно для ініціатора команди. Також впроваджено функцію пошуку позицій та можливість скасування замовлень клієнтом до початку їх обробки.

6. **Створено захищену адміністративну панель та систему підтримки (Helpdesk).** Адміністративний модуль забезпечує виконання CRUD-операцій з товарним асортиментом та модерацію замовлень із вбудованою системою перевірки статусів (State Machine). Система підтримки (тікети) дозволяє клієнтам створювати звернення, а адміністрації — оперативно відповідати на них, причому відповіді автоматично фіксуються в БД і додатково дублюються користувачеві в приватні повідомлення (Direct Messages). Будь-яка дія адміністратора (редагування ціни, зміна статусу, закриття тікета) автоматично записується в неперервний системний журнал (Audit Trail) для подальшого контролю власником.

7. **Проведено комплексне тестування розробленого програмного забезпечення.** Методиками модульного (Unit) та функціонального (Black-box) тестування підтверджено безперебійну роботу динамічного обробника команд (Command Handler) та пулу з'єднань бази даних. Емпірично доведено працездатність алгоритмів обробки помилок: система успішно перехоплює винятки (наприклад, недостатню кількість товару або

закриті приватні повідомлення клієнта) без аварійного завершення роботи ядра сервера. Аудит БД підтвердив, що реверсивні транзакції працюють коректно, і при скасуванні замовлення товари автоматично та точно повертаються на віртуальний склад.

Підсумок. Створений Discord-бот є завершеним, автономним програмним продуктом, який повністю відповідає всім висунутим у технічному завданні функціональним та нефункціональним вимогам. Практичне значення роботи полягає в тому, що розроблена система може бути безпосередньо впроваджена в робочі процеси реального малого бізнесу, суттєво зменшуючи операційні витрати на обробку клієнтських запитів. Архітектура застосунку відрізняється високою масштабованістю: модульна структура та ізоляція бізнес-логіки створюють міцний фундамент для майбутнього розширення. Перспективами розвитку даної системи є інтеграція платіжних шлюзів для безпосередньої оплати товарів, додавання функціоналу "кошика" (cart) для групових замовлень, а також розробка супутньої повноцінної веб-панелі управління, яка буде використовувати існуючу структуру бази даних MySQL.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Гриценко В. П. Автоматизація бізнес-процесів у сучасній торгівлі : навч. посіб. Київ : КНТЕУ, 2019. 156 с.

2. Мельник О. В. Соціальна комерція: нові вектори розвитку цифрового ринку. Економіка та суспільство. 2021. № 25. URL: http://economyandsociety.in.ua

3. Discord Developer Documentation. Documentation for Discord's API and Developer tools. URL: https://discord.com/developers/docs

4. Іванов С. С. Системи підтримки клієнтів у месенджерах: аналіз ефективності. Комп'ютерні системи та мережі. 2022. Т. 12, № 2. С. 45-52.

5. Мельник А. О. Проектування високонавантажених інформаційних систем : підручник. Львів : Видавництво Львівської політехніки, 2020. 288 с.

6. Кантор І. М. Сучасний JavaScript для розробників : посіб. Київ : КПІ, 2021. 412 с.

7. Tilkov I., Vinoski S. Node.js: Using JavaScript to Build High-Performance Network Programs. IEEE Internet Computing. 2010. Vol. 14, No. 6. P. 80-83.

8. Сміт Д., Джонсон А. Аналіз екосистеми ботів у месенджерах. Software Engineering Journal. 2022. Т. 5, № 3. С. 112-118.

9. Кочетов А. О. Системи керування базами даних : підручник. Київ : ВПЦ "Київський університет", 2019. 320 с.

10. Garcia-Molina H., Ullman J. D., Widom J. Database Systems: The Complete Book. 2nd ed. Pearson Education, 2014. 1248 p.

11. UML Distilled: A Brief Guide to the Standard Object Modeling Language. 3rd Edition / Fowler M. — Addison-Wesley, 2018. — 208 p.

12. Розенберг Д., Скотт К. Применение объектного моделирования с использованием UML и анализ прецедентов / Пер. с англ. — М.: ДМК Пресс, 2012. — 160 с.

13. Конноллі Т., Бегг К. Бази даних: проектування, реалізація та супровід. Теорія та практика. 4-е вид. — М.: Вільямс, 2017. — 1440 с.

14. Дейт К. Дж. Вступ до систем баз даних. 8-е вид. — М.: Вільямс, 2016. — 1328 с.

15. Shevat E. Designing Bots: Creating Conversational Experiences. — O'Reilly Media, 2017. — 314 p.

16. Кузьменко О. В. Проектування людино-машинних інтерфейсів : навч. посібник. — Харків : ХНУРЕ, 2021. — 205 с.

17. Clarke J. SQL Injection Attacks and Defense. 2nd Edition. — Syngress, 2012. — 564 p.

18. Андрєєв О. В. Основи інформаційної безпеки корпоративних мереж : навч. посіб. — Одеса : НУ «ОМА», 2021. — 215 с.

19. Мартін Р. Чиста архітектура: мистецтво розроблення програмного забезпечення. — Харків: Фабула, 2019. — 368 с.

20. Гамма Е., Хелм Р., Джонсон Р., Вліссідес Дж. Патерни проектування: багаторазове використання об'єктно-орієнтованого програмного забезпечення. — СПб.: Пітер, 2020. — 368 с.

21. Фрімен Е., Робсон Е. Head First. Патерни проектування. — Київ: O'Reilly, 2022. — 656 с.

22. Teixeira P. Professional Node.js: Building Real-World Scalable Web Applications. — Wrox, 2012. — 408 p.

23. Хавербеке М. Виразний JavaScript. Сучасне програмування. 3-є вид. — СПб.: Пітер, 2019. — 480 с.

24. Сміт Р. MySQL. Оптимізація продуктивності. 3-є вид. — М.: Символ-Плюс, 2017. — 520 с.

25. Касеро М. Node.js. Шаблони проектування. 2-е вид. — СПб.: Пітер, 2021. — 450 с.

26. Haverbeke M. Eloquent JavaScript: A Modern Introduction to Programming. 3rd Edition. — No Starch Press, 2018. — 472 p.

27. Маккінні Е. Python і аналіз даних. — Київ: КІС, 2020. — 400 с. (Прим.: загальні принципи захисту від ін'єкцій).

28. Офіційна документація Discord API: Embed Limits. URL: https://discord.com/developers/docs/resources/channel#embed-object-embed-limits (дата звернення: 09.05.2026).

29. Корнел П., Сарка Д. Основи баз даних. T-SQL і програмування транзакцій. — Київ: ДІА, 2018. — 512 с.

30. Холмс Д. Стек MEAN. Розробка вебзастосунків на Node.js, MongoDB та Express. — СПб.: Пітер, 2020. — 496 с.

31. Бредлі Е. Розробка систем електронної комерції: архітектура та бізнес-процеси. — Київ: Техніка, 2021. — 288 с.

32. Грінберг П. CRM зі швидкістю світла. Соціальні стратегії для взаємодії з клієнтами. — СПб.: Символ-Плюс, 2019. — 528 с.

33. Соммервілл І. Інженерія програмного забезпечення. 10-те вид. — Київ: Видавництво "ПВ", 2018. — 856 с.

34. Бхаргава О. Основи кібербезпеки: Захист інформаційних систем. — Харків: ХНУРЕ, 2020. — 310 с.

35. Томпсон Д. Проектування архітектури API для мікросервісів. — Львів: Компакт, 2021. — 250 с.

36. Офіційна документація Discord API: User Object and DMs. URL: https://discord.com/developers/docs/resources/user (дата звернення: 09.05.2026).

37. Майєрс Г., Баджетт Т., Сандлер К. Мистецтво тестування програмного забезпечення. 3-тє вид. — Київ: Діалектика, 2017. — 272 с.

38. IEEE Standard for Software and System Test Documentation. IEEE Std 829-2008. — New York: IEEE, 2008. — 150 p.

39. Канер Ц., Фолк Д., Нгуєн Е. Тестування програмного забезпечення. Базовий курс. — Київ: Діалектика, 2018. — 430 с.

40. Ніксон Р. Створення вебзастосунків на основі сучасних технологій. — СПб.: Пітер, 2021. — 500 с.

41. Ельмазрі Р., Нават Ш. Основи систем баз даних. 7-ме вид. — М.: Вільямс, 2017. — 1376 с.

42. Оуенс Б. Теорія реляційних баз даних. Транзакції та цілісність. — Київ: Основи, 2019. — 340 с.

43. Date C. J. Database Design and Relational Theory: Normal Forms and All That Jazz. — O'Reilly Media, 2012. — 280 p.

44. Петров В. М. Забезпечення відмовостійкості інформаційних систем. — Харків: ХНУРЕ, 2021. — 215 с.

45. Льюїс В. Основи адміністрування та аудиту баз даних. — Київ: Професіонал, 2020. — 380 с.

46. Ковальчук О. М. Методи верифікації та валідації програмних комплексів електронної комерції. Сучасні інформаційні системи. 2022. Т. 6, № 4. С. 112-120.

47. Відео урок I Tried Coding a Discord Bot in 6 Different Programming Languages https://youtu.be/C_luMMsOtlA?si=CohtTHLKsR6pvcgc

Додаток А скрипт заповнення БД

```
CREATE DATABASE IF NOT EXISTS discord_store CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;

USE discord_store;


-- 1. Користувачі

CREATE TABLE users (

   id INT AUTO_INCREMENT PRIMARY KEY,

   discord_id VARCHAR(20) UNIQUE NOT NULL,

   username VARCHAR(255) NOT NULL,

   display_name VARCHAR(255),

   registered_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

   last_activity_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,

   is_blocked BOOLEAN DEFAULT FALSE

);


-- 2. Адміністратори

CREATE TABLE admins (

   id INT AUTO_INCREMENT PRIMARY KEY,

   user_id INT NOT NULL,

   role_level INT DEFAULT 1,

   created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

   FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE

);


-- 3. Категорії

CREATE TABLE categories (
```

```
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    is_active BOOLEAN DEFAULT TRUE
);

-- 4. Товари
CREATE TABLE products (
    id INT AUTO_INCREMENT PRIMARY KEY,
    category_id INT,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    price DECIMAL(10, 2) NOT NULL,
    stock_quantity INT NOT NULL DEFAULT 0,
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    FOREIGN KEY (category_id) REFERENCES categories(id) ON DELETE SET
NULL
);

-- 5. Замовлення
CREATE TABLE orders (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    status ENUM('new', 'confirmed', 'processing', 'completed', 'canceled') DEFAULT
'new',
```

```
    total_amount DECIMAL(10, 2) NOT NULL DEFAULT 0.00,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);

-- 6. Позиції замовлення (конкретні товари в чеку)
CREATE TABLE order_items (
    id INT AUTO_INCREMENT PRIMARY KEY,
    order_id INT NOT NULL,
    product_id INT NOT NULL,
    quantity INT NOT NULL,
    unit_price DECIMAL(10, 2) NOT NULL,
    subtotal DECIMAL(10, 2) NOT NULL,
    FOREIGN KEY (order_id) REFERENCES orders(id) ON DELETE CASCADE,
    FOREIGN KEY (product_id) REFERENCES products(id) ON DELETE
CASCADE
);

-- 7. Звернення (Тікети)
CREATE TABLE support_tickets (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    subject VARCHAR(255) NOT NULL,
    message TEXT NOT NULL,
    status ENUM('open', 'in_progress', 'closed') DEFAULT 'open',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
```

```
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);

-- 8. Повідомлення в тікетах
CREATE TABLE ticket_messages (
    id INT AUTO_INCREMENT PRIMARY KEY,
    ticket_id INT NOT NULL,
    sender_type ENUM('user', 'admin') NOT NULL,
    sender_id INT NOT NULL,
    message TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (ticket_id) REFERENCES support_tickets(id) ON DELETE
CASCADE
);

-- 9. Журнал дій (Логи)
CREATE TABLE logs (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT,
    action_type VARCHAR(100) NOT NULL,
    description TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE SET NULL
);

-- 10. Історія статусів замовлень
```

```
CREATE TABLE order_status_history (
    id INT AUTO_INCREMENT PRIMARY KEY,
    order_id INT NOT NULL,
    old_status ENUM('new', 'confirmed', 'processing', 'completed', 'canceled'),
    new_status ENUM('new', 'confirmed', 'processing', 'completed', 'canceled') NOT
NULL,
    changed_by INT,
    changed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (order_id) REFERENCES orders(id) ON DELETE CASCADE,
    FOREIGN KEY (changed_by) REFERENCES users(id) ON DELETE SET NULL
);
```

Додаток Б файл db.js

```
const mysql = require('mysql2/promise');
require('dotenv').config();

// Створюємо пул з'єднань
const pool = mysql.createPool({
    host: process.env.DB_HOST,
    user: process.env.DB_USER,
    password: process.env.DB_PASSWORD,
    database: process.env.DB_NAME,
    waitForConnections: true,
    connectionLimit: 10,
    queueLimit: 0
});

module.exports = pool;
```

Додаток В файл index.js

```
require('dotenv').config();
const fs = require('node:fs');
const path = require('node:path');
const { Client, GatewayIntentBits, Collection } = require('discord.js');
const pool = require('./database/db.js');

const client = new Client({
    intents: [
        GatewayIntentBits.Guilds,
        GatewayIntentBits.GuildMessages,
        GatewayIntentBits.MessageContent
    ]
});

client.commands = new Collection();

// Автоматичне завантаження команд з папок
const foldersPath = path.join(__dirname, 'commands');
const commandFolders = fs.readdirSync(foldersPath);

for (const folder of commandFolders) {
    const commandsPath = path.join(foldersPath, folder);
    const commandFiles = fs.readdirSync(commandsPath).filter(file =>
file.endsWith('.js'));
    for (const file of commandFiles) {
```

```
        const filePath = path.join(commandsPath, file);
        const command = require(filePath);
        if ('data' in command && 'execute' in command) {
            client.commands.set(command.data.name, command);
        } else {
            console.log(`[ПОПЕРЕДЖЕННЯ] Команда у ${filePath} не має обов'язкових властивостей "data" або "execute".`);
        }
    }
}

// Слухач команд
client.on('interactionCreate', async interaction => {
    if (!interaction.isChatInputCommand()) return;

    const command = interaction.client.commands.get(interaction.commandName);

    if (!command) {
        console.error(`Команду ${interaction.commandName} не знайдено.`);
        return;
    }

    try {
        await command.execute(interaction);
    } catch (error) {
        console.error(error);
        if (interaction.replied || interaction.deferred) {
```

```
            await interaction.followUp({ content: 'Виникла помилка при виконанні цієї команди!', ephemeral: true });
        } else {
            await interaction.reply({ content: 'Виникла помилка при виконанні цієї команди!', ephemeral: true });
        }
    }
});

async function init() {
    try {
        const connection = await pool.getConnection();
        console.log('✅ База даних: успішно підключена.');
        connection.release();

        await client.login(process.env.DISCORD_TOKEN);
        console.log(`✅ Бот ${client.user.tag}: запущений і готовий до роботи.`);
    } catch (error) {
        console.error('❌ Критична помилка під час запуску:');
        console.error(error);
        process.exit(1);
    }
}

init();
```

Додаток Г команда /start

```
const { SlashCommandBuilder } = require('discord.js');
const pool = require('../../database/db.js');

module.exports = {
    data: new SlashCommandBuilder()
        .setName('start')
        .setDescription('Початок роботи з ботом та реєстрація в системі'),

    async execute(interaction) {
        const discordId = interaction.user.id;
        const username = interaction.user.username;
        const displayName = interaction.user.displayName;

        try {
            // Робимо запит до БД: шукаємо користувача
            const [rows] = await pool.query('SELECT * FROM users WHERE discord_id
= ?', [discordId]);

            if (rows.length === 0) {
                // Якщо користувача немає — додаємо в базу
                await pool.query(
                    'INSERT INTO users (discord_id, username, display_name) VALUES
(?, ?, ?)',
                    [discordId, username, displayName]
                );
```

```
            await interaction.reply(`Вітаю, **${displayName}**! Тебе успішно зареєстровано в системі. Використовуй команду \`/help\` для перегляду можливостей.`);
        } else {
            // Якщо вже є в базі
            await interaction.reply(`З поверненням, **${displayName}**! Ти вже зареєстрований у нашій системі. Переглянь каталог: \`/catalog\``);
        }
    } catch (error) {
        console.error('Помилка БД при реєстрації:', error);
        await interaction.reply({ content: 'Виникла внутрішня помилка бази даних.', ephemeral: true });
    }
  },
};
```

Додаток Ґ файл deploy-commands.js

```
require('dotenv').config();
const { REST, Routes } = require('discord.js');
const fs = require('node:fs');
const path = require('node:path');

const commands = [];
const foldersPath = path.join(__dirname, 'commands');
const commandFolders = fs.readdirSync(foldersPath);

for (const folder of commandFolders) {
  const commandsPath = path.join(foldersPath, folder);
```

```
    const commandFiles = fs.readdirSync(commandsPath).filter(file =>
file.endsWith('.js'));
    for (const file of commandFiles) {
        const filePath = path.join(commandsPath, file);
        const command = require(filePath);
        if ('data' in command && 'execute' in command) {
            commands.push(command.data.toJSON());
        }
    }
}

const rest = new REST().setToken(process.env.DISCORD_TOKEN);

(async () => {
    try {
        console.log(`Початок оновлення ${commands.length} (/) команд...`);

        // Відправляємо команди на конкретний сервер (Guild) для миттєвого
оновлення
        const data = await rest.put(
            Routes.applicationGuildCommands(process.env.CLIENT_ID,
process.env.GUILD_ID),
            { body: commands },
        );

        console.log(`✅ Успішно завантажено ${data.length} (/) команд.`);
    } catch (error) {
        console.error(error);
```

```
    }
})();
```

Додаток Д команда /profile

```
const { SlashCommandBuilder, EmbedBuilder } = require('discord.js');
const pool = require('../../database/db.js');

module.exports = {
    data: new SlashCommandBuilder()
        .setName('profile')
        .setDescription('Перегляд вашого профілю у системі'),

    async execute(interaction) {
        const discordId = interaction.user.id;

        try {
            const [rows] = await pool.query('SELECT * FROM users WHERE discord_id = ?', [discordId]);

            if (rows.length === 0) {
                return interaction.reply({ content: 'Вас ще немає в базі. Використайте спочатку команду `/start`.', ephemeral: true });
            }

            const user = rows[0];

            // Створюємо красиве Embed-повідомлення
            const profileEmbed = new EmbedBuilder()
```

```
            .setColor(0x0099FF)
            .setTitle(`Профіль клієнта: ${user.display_name}`)
            .addFields(
                { name: 'Discord ID', value: user.discord_id, inline: true },
                { name: 'Дата реєстрації', value: new
Date(user.registered_at).toLocaleDateString('uk-UA'), inline: true },
                { name: 'Статус акаунта', value: user.is_blocked ? '· Заблокований' : '·
Активний', inline: false }
            )
            .setThumbnail(interaction.user.displayAvatarURL());

        await interaction.reply({ embeds: [profileEmbed] });

    } catch (error) {
        console.error('Помилка БД при отриманні профілю:', error);
        await interaction.reply({ content: 'Помилка при отриманні даних.',
ephemeral: true });
    }
  },
};
```

Додаток Е /help

```
const { SlashCommandBuilder, EmbedBuilder } = require('discord.js');

module.exports = {
  data: new SlashCommandBuilder()
    .setName('help')
    .setDescription('Довідка по командах бота'),
```

```
  async execute(interaction) {
    const helpEmbed = new EmbedBuilder()
      .setColor(0x00FF00)
      .setTitle('Довідковий центр ServiceBridge')
      .setDescription('Ось список доступних вам команд на даному етапі:')
      .addFields(
        { name: '· Основні', value: '`/start` - Реєстрація\n`/profile` - Ваш профіль\n`/help` - Ця довідка' },
        { name: '· Магазин (в розробці)', value: '`/catalog` - Каталог товарів\n`/myorders` - Ваші замовлення' }
      )
      .setFooter({ text: 'Discord-бот для автоматичного обліку замовлень' });

    await interaction.reply({ embeds: [helpEmbed] });
  },
};
```

Додаток Є Команда /catalog

```
const { SlashCommandBuilder, EmbedBuilder, ActionRowBuilder, ButtonBuilder, ButtonStyle, ComponentType } = require('discord.js');
const pool = require('../../database/db.js');

module.exports = {
  data: new SlashCommandBuilder()
    .setName('catalog')
    .setDescription('Перегляд каталогу товарів'),

  async execute(interaction) {
```

```
    await interaction.deferReply(); // Відкладаємо відповідь, бо запит до БД може зайняти час

    try {
        // Отримуємо всі активні товари
        const [products] = await pool.query(`
            SELECT p.id, p.name, p.price, p.stock_quantity, c.name as category_name
            FROM products p
            LEFT JOIN categories c ON p.category_id = c.id
            WHERE p.is_active = TRUE
        `);

        if (products.length === 0) {
            return interaction.editReply('Каталог наразі порожній.');
        }

        const itemsPerPage = 3;
        const pages = Math.ceil(products.length / itemsPerPage);
        let currentPage = 1;

        const generateEmbed = (page) => {
            const start = (page - 1) * itemsPerPage;
            const end = page * itemsPerPage;
            const currentProducts = products.slice(start, end);

            const embed = new EmbedBuilder()
                .setColor(0xFFA500)
```

```
            .setTitle('· Каталог товарів')
            .setFooter({ text: `Сторінка ${page} з ${pages}` });

        currentProducts.forEach(product => {
            embed.addFields({
                name: `[ID: ${product.id}] ${product.name}`,
                value: `Категорія: ${product.category_name || 'Без категорії'}\nЦіна: **${product.price} грн** | В наявності: ${product.stock_quantity} шт.`,
                inline: false
            });
        });

        return embed;
    };

    const getButtons = (page) => {
        const row = new ActionRowBuilder();

        row.addComponents(
            new ButtonBuilder()
                .setCustomId('prev_page')
                .setLabel('◀ Попередня')
                .setStyle(ButtonStyle.Primary)
                .setDisabled(page === 1),
            new ButtonBuilder()
                .setCustomId('next_page')
                .setLabel('Наступна ▶')
```

```
                .setStyle(ButtonStyle.Primary)
                .setDisabled(page === pages)
            );
            return row;
        };

        const response = await interaction.editReply({
            embeds: [generateEmbed(currentPage)],
            components: pages > 1 ? [getButtons(currentPage)] : []
        });

        if (pages === 1) return;

        // Створюємо колектор для обробки натискань кнопок
        const collector = response.createMessageComponentCollector({
componentType: ComponentType.Button, time: 60000 });

        collector.on('collect', async i => {
            if (i.user.id !== interaction.user.id) {
                return i.reply({ content: 'Ви не можете використовувати ці кнопки.',
ephemeral: true });
            }

            if (i.customId === 'prev_page') currentPage--;
            if (i.customId === 'next_page') currentPage++;

            await i.update({
                embeds: [generateEmbed(currentPage)],
```

```
                components: [getButtons(currentPage)]
            });
        });

        collector.on('end', () => {
            response.edit({ components: [] }).catch(() => {});
        });

    } catch (error) {
        console.error('Помилка при завантаженні каталогу:', error);
        await interaction.editReply('Виникла помилка при завантаженні каталогу.');
    }
  },
};
```

Додаток Ж /product

```
const { SlashCommandBuilder, EmbedBuilder } = require('discord.js');
const pool = require('../../database/db.js');

module.exports = {
  data: new SlashCommandBuilder()
    .setName('product')
    .setDescription('Перегляд детальної інформації про товар')
    .addIntegerOption(option =>
      option.setName('id')
        .setDescription('ID товару')
        .setRequired(true)),
```

```
    async execute(interaction) {
        const productId = interaction.options.getInteger('id');

        try {
            const [rows] = await pool.query(`
                SELECT p.*, c.name as category_name
                FROM products p
                LEFT JOIN categories c ON p.category_id = c.id
                WHERE p.id = ? AND p.is_active = TRUE
            `, [productId]);

            if (rows.length === 0) {
                return interaction.reply({ content: 'Товар з таким ID не знайдено або він недоступний.', ephemeral: true });
            }

            const product = rows[0];

            const embed = new EmbedBuilder()
                .setColor(0x00FFFF)
                .setTitle(`· ${product.name}`)
                .setDescription(product.description || 'Опис відсутній')
                .addFields(
                    { name: 'Категорія', value: product.category_name || 'Без категорії', inline: true },
                    { name: 'Ціна', value: `**${product.price} грн**`, inline: true },
```

```
                { name: 'На складі', value: `${product.stock_quantity} шт.`, inline: true }
            )
            .setFooter({ text: `ID товару: ${product.id}` });

        await interaction.reply({ embeds: [embed] });

    } catch (error) {
        console.error('Помилка при пошуку товару:', error);
        await interaction.reply({ content: 'Виникла помилка при пошуку товару.', ephemeral: true });
    }
  },
};
```

Додаток З Команда /search

```
const { SlashCommandBuilder, EmbedBuilder } = require('discord.js');
const pool = require('../../database/db.js');

module.exports = {
  data: new SlashCommandBuilder()
    .setName('search')
    .setDescription('Пошук товару за назвою')
    .addStringOption(option =>
      option.setName('query')
        .setDescription('Введіть назву товару для пошуку')
        .setRequired(true)),
```

```
    async execute(interaction) {
        const searchQuery = interaction.options.getString('query');

        try {
            // Використовуємо LIKE для пошуку часткового збігу (%текст%)
            const [rows] = await pool.query(`
                SELECT p.id, p.name, p.price, p.stock_quantity, c.name as category_name
                FROM products p
                LEFT JOIN categories c ON p.category_id = c.id
                WHERE p.is_active = TRUE AND p.name LIKE ?
                LIMIT 10
            `, [`%${searchQuery}%`]);

            if (rows.length === 0) {
                return interaction.reply({
                    content: `За запитом **"${searchQuery}"** нічого не знайдено. Спробуйте інше слово.`,
                    ephemeral: true
                });
            }

            const embed = new EmbedBuilder()
                .setColor(0x9B59B6)
                .setTitle(`· Результати пошуку: "${searchQuery}"`)
                .setDescription(`Знайдено товарів: ${rows.length}`)
                .setFooter({ text: 'Використовуйте /product [id] для деталей' });
```

```
            rows.forEach(product => {
                embed.addFields({
                    name: `[ID: ${product.id}] ${product.name}`,
                    value: `Категорія: ${product.category_name || 'Без категорії'}\nЦіна: **${product.price} грн** | В наявності: ${product.stock_quantity} шт.`,
                    inline: false
                });
            });

            await interaction.reply({ embeds: [embed] });

        } catch (error) {
            console.error('Помилка при пошуку товару:', error);
            await interaction.reply({ content: 'Виникла помилка під час обробки пошукового запиту.', ephemeral: true });
        }
    },
};
```

Додаток И Команда /order

```
const { SlashCommandBuilder, EmbedBuilder } = require('discord.js');
const pool = require('../../database/db.js');

module.exports = {
    data: new SlashCommandBuilder()
        .setName('order')
        .setDescription('Оформлення замовлення')
        .addIntegerOption(option =>
```

```
        option.setName('product_id').setDescription('ID товару з
каталогу').setRequired(true))
        .addIntegerOption(option =>

option.setName('quantity').setDescription('Кількість').setRequired(true).setMinValue
(1)),

    async execute(interaction) {
        const discordId = interaction.user.id;
        const productId = interaction.options.getInteger('product_id');
        const quantity = interaction.options.getInteger('quantity');

        await interaction.deferReply({ ephemeral: true }); // Відповідь бачить лише
покупець

        const connection = await pool.getConnection(); // Беремо окреме з'єднання
для транзакції

        try {
            await connection.beginTransaction(); // Початок транзакції

            // 1. Шукаємо внутрішній ID користувача
            const [users] = await connection.query('SELECT id FROM users WHERE
discord_id = ?', [discordId]);
            if (users.length === 0) throw new Error('USER_NOT_FOUND');
            const userId = users[0].id;

            // 2. Перевіряємо товар і залишок
```

```
        const [products] = await connection.query('SELECT name, price,
stock_quantity FROM products WHERE id = ? AND is_active = TRUE FOR
UPDATE', [productId]);
        if (products.length === 0) throw new Error('PRODUCT_NOT_FOUND');


        const product = products[0];
        if (product.stock_quantity < quantity) throw new
Error('NOT_ENOUGH_STOCK');


        const totalAmount = product.price * quantity;


        // 3. Створюємо запис у таблиці orders
        const [orderResult] = await connection.query(
          'INSERT INTO orders (user_id, total_amount, status) VALUES (?, ?, ?)',
          [userId, totalAmount, 'new']
        );
        const orderId = orderResult.insertId;


        // 4. Створюємо запис у таблиці order_items
        await connection.query(
          'INSERT INTO order_items (order_id, product_id, quantity, unit_price,
subtotal) VALUES (?, ?, ?, ?, ?)',
          [orderId, productId, quantity, product.price, totalAmount]
        );


        // 5. Оновлюємо залишок на складі
        await connection.query(
```

```
            'UPDATE products SET stock_quantity = stock_quantity - ? WHERE id =
?',
            [quantity, productId]
        );

        // 6. Записуємо історію статусу
        await connection.query(
            'INSERT INTO order_status_history (order_id, new_status, changed_by)
VALUES (?, ?, ?)',
            [orderId, 'new', userId]
        );

        await connection.commit(); // Підтверджуємо транзакцію!

        const embed = new EmbedBuilder()
            .setColor(0x00FF00)
            .setTitle(`✅ Замовлення #${orderId} успішно оформлено!`)
            .setDescription(`Ви замовили: **${product.name}** (x${quantity})\nДо
сплати: **${totalAmount} грн**`)
            .setFooter({ text: 'Використовуйте /myorders для перегляду ваших
замовлень' });

        await interaction.editReply({ embeds: [embed] });

    } catch (error) {
        await connection.rollback(); // Якщо помилка - скасовуємо всі зміни в БД!
        console.error('Помилка транзакції замовлення:', error);
```

```
            if (error.message === 'USER_NOT_FOUND') return
interaction.editReply('Вас немає в базі. Напишіть `/start`.');
            if (error.message === 'PRODUCT_NOT_FOUND') return
interaction.editReply('Товар не знайдено або він недоступний.');
            if (error.message === 'NOT_ENOUGH_STOCK') return
interaction.editReply('На складі недостатньо товару.');


            await interaction.editReply('Виникла критична помилка при оформленні
замовлення.');
        } finally {
            connection.release(); // Обов'язково повертаємо з'єднання в пул
        }
    },
};
```

Додаток Ї Команда /myorders

```
const { SlashCommandBuilder, EmbedBuilder } = require('discord.js');
const pool = require('../../database/db.js');

module.exports = {
    data: new SlashCommandBuilder()
        .setName('myorders')
        .setDescription('Перегляд списку ваших замовлень'),

    async execute(interaction) {
        const discordId = interaction.user.id;

        try {
            const [rows] = await pool.query(`
```

```
            SELECT o.id, o.status, o.total_amount, o.created_at
            FROM orders o
            JOIN users u ON o.user_id = u.id
            WHERE u.discord_id = ?
            ORDER BY o.created_at DESC
            LIMIT 5
        `, [discordId]);

        if (rows.length === 0) {
            return interaction.reply({ content: 'У вас ще немає замовлень.',
ephemeral: true });
        }

        const embed = new EmbedBuilder()
            .setColor(0x3498DB)
            .setTitle('· Ваші останні 5 замовлень');

        const statusMap = {
            'new': '· Нове',
            'confirmed': '✅ Підтверджене',
            'processing': '☐ В обробці',
            'completed': '· Завершене',
            'canceled': '❌ Скасоване'
        };

        rows.forEach(order => {
```

```
            const date = new Date(order.created_at).toLocaleString('uk-UA');
            embed.addFields({
                name: `Замовлення #${order.id}`,
                value: `Статус: **${statusMap[order.status] || order.status}**\nСума: **${order.total_amount} грн**\nДата: ${date}`,
                inline: false
            });
        });

        await interaction.reply({ embeds: [embed], ephemeral: true });

    } catch (error) {
        console.error('Помилка при отриманні замовлень:', error);
        await interaction.reply({ content: 'Помилка при завантаженні списку замовлень.', ephemeral: true });
    }
  },
};
```

Додаток Й Команда /orderinfo

```
const { SlashCommandBuilder, EmbedBuilder } = require('discord.js');
const pool = require('../../database/db.js');

module.exports = {
  data: new SlashCommandBuilder()
    .setName('orderinfo')
    .setDescription('Перегляд деталей конкретного замовлення')
    .addIntegerOption(option =>
```

```
        option.setName('order_id').setDescription('Номер замовлення').setRequired(true)),

    async execute(interaction) {
        const discordId = interaction.user.id;
        const orderId = interaction.options.getInteger('order_id');

        try {
            // Перевіряємо, чи існує замовлення і чи належить воно цьому користувачу
            const [orders] = await pool.query(`
                SELECT o.* FROM orders o
                JOIN users u ON o.user_id = u.id
                WHERE o.id = ? AND u.discord_id = ?
            `, [orderId, discordId]);

            if (orders.length === 0) {
                return interaction.reply({ content: 'Замовлення не знайдено, або воно вам не належить.', ephemeral: true });
            }

            const order = orders[0];

            // Отримуємо позиції замовлення
            const [items] = await pool.query(`
                SELECT oi.*, p.name
                FROM order_items oi
                JOIN products p ON oi.product_id = p.id
```

```
            WHERE oi.order_id = ?
        `, [orderId]);

        const statusMap = {
            'new': '· Нове',
            'confirmed': '✅ Підтверджене',
            'processing': '☐ В обробці',
            'completed': '· Завершене',
            'canceled': '❌ Скасоване'
        };

        const embed = new EmbedBuilder()
            .setColor(0x2ECC71)
            .setTitle(`· Чек замовлення #${order.id}`)
            .addFields(
                { name: 'Статус', value: statusMap[order.status] || order.status, inline:
true },
                { name: 'Загальна сума', value: `**${order.total_amount} грн**`,
inline: true },
                { name: 'Дата', value: new Date(order.created_at).toLocaleString('uk-
UA'), inline: false }
            );

        let itemsList = '';
        items.forEach((item, index) => {
            itemsList += `${index + 1}. **${item.name}** — ${item.quantity} шт. x
${item.unit_price} грн = ${item.subtotal} грн\n`;
```

```
        });

        embed.addFields({ name: 'Позиції:', value: itemsList || 'Помилка завантаження товарів' });

        await interaction.reply({ embeds: [embed], ephemeral: true });

      } catch (error) {
        console.error('Помилка при отриманні деталей замовлення:', error);
        await interaction.reply({ content: 'Виникла помилка під час пошуку деталей.', ephemeral: true });
      }
  },
};
```

Додаток К Команда /cancelorder

```
const { SlashCommandBuilder, EmbedBuilder } = require('discord.js');
const pool = require('../../database/db.js');

module.exports = {
  data: new SlashCommandBuilder()
    .setName('cancelorder')
    .setDescription('Скасування замовлення')
    .addIntegerOption(option =>
      option.setName('order_id').setDescription('Номер замовлення').setRequired(true)),

  async execute(interaction) {
    const discordId = interaction.user.id;
```

```
const orderId = interaction.options.getInteger('order_id');

await interaction.deferReply({ ephemeral: true });

const connection = await pool.getConnection();

try {
    await connection.beginTransaction();

    // 1. Отримуємо замовлення і користувача
    const [orders] = await connection.query(`
        SELECT o.id, o.status, o.user_id
        FROM orders o
        JOIN users u ON o.user_id = u.id
        WHERE o.id = ? AND u.discord_id = ? FOR UPDATE
    `, [orderId, discordId]);

    if (orders.length === 0) throw new Error('NOT_FOUND');

    const order = orders[0];

    // 2. Перевіряємо статус (скасувати можна тільки нові або підтверджені)
    if (order.status !== 'new' && order.status !== 'confirmed') {
        throw new Error('INVALID_STATUS');
    }

    // 3. Змінюємо статус замовлення
```

```
        await connection.query('UPDATE orders SET status = ? WHERE id = ?', ['canceled', orderId]);

        // 4. Отримуємо всі товари з цього замовлення
        const [items] = await connection.query('SELECT product_id, quantity FROM order_items WHERE order_id = ?', [orderId]);

        // 5. Повертаємо товари на склад (відновлюємо stock_quantity)
        for (const item of items) {
          await connection.query(
            'UPDATE products SET stock_quantity = stock_quantity + ? WHERE id = ?',
            [item.quantity, item.product_id]
          );
        }

        // 6. Записуємо лог в історію статусів
        await connection.query(
          'INSERT INTO order_status_history (order_id, old_status, new_status, changed_by) VALUES (?, ?, ?, ?)',
          [orderId, order.status, 'canceled', order.user_id]
        );

        await connection.commit();

        const embed = new EmbedBuilder()
          .setColor(0xE74C3C)
          .setTitle(`· Замовлення #${orderId} скасовано`)
```

```
            .setDescription('Ваше замовлення успішно скасовано. Товари повернуто на склад.');

        await interaction.editReply({ embeds: [embed] });

    } catch (error) {
        await connection.rollback();

        if (error.message === 'NOT_FOUND') return interaction.editReply('Замовлення не знайдено.');
        if (error.message === 'INVALID_STATUS') return interaction.editReply('Це замовлення вже не можна скасувати (воно в обробці або завершене).');

        console.error('Помилка транзакції скасування:', error);
        await interaction.editReply('Виникла критична помилка під час скасування.');
    } finally {
        connection.release();
    }
  },
};
```

Додаток Л Команда /support

```
const { SlashCommandBuilder, EmbedBuilder } = require('discord.js');
const pool = require('../../database/db.js');

module.exports = {
  data: new SlashCommandBuilder()
    .setName('support')
```

```
        .setDescription('Створення звернення до служби підтримки')
        .addStringOption(option =>
            option.setName('subject').setDescription('Тема звернення').setRequired(true).setMaxLength(100))
        .addStringOption(option =>
            option.setName('message').setDescription('Опишіть вашу проблему').setRequired(true).setMaxLength(1000)),

    async execute(interaction) {
        const discordId = interaction.user.id;
        const subject = interaction.options.getString('subject');
        const message = interaction.options.getString('message');

        try {
            // Отримуємо внутрішній ID користувача
            const [users] = await pool.query('SELECT id FROM users WHERE discord_id = ?', [discordId]);
            if (users.length === 0) return interaction.reply({ content: 'Спочатку пропишіть `/start`.', ephemeral: true });

            const userId = users[0].id;

            // Створюємо тікет
            const [result] = await pool.query(
                'INSERT INTO support_tickets (user_id, subject, message, status) VALUES (?, ?, ?, ?)',
                [userId, subject, message, 'open']
            );
```

```
        const ticketId = result.insertId;

        const embed = new EmbedBuilder()
            .setColor(0xF1C40F)
            .setTitle(`· Тікет #${ticketId} створено`)
            .addFields(
                { name: 'Тема', value: subject },
                { name: 'Статус', value: '· Відкрито' }
            )
            .setDescription('Адміністратори отримали ваше повідомлення і скоро нададуть відповідь.')
            .setFooter({ text: 'Перегляд усіх ваших тікетів: /mytickets' });

        await interaction.reply({ embeds: [embed], ephemeral: true });

    } catch (error) {
        console.error('Помилка при створенні тікета:', error);
        await interaction.reply({ content: 'Не вдалося створити звернення.', ephemeral: true });
    }
  },
};
```

Додаток М Команда /mytickets

```
const { SlashCommandBuilder, EmbedBuilder } = require('discord.js');
const pool = require('../../database/db.js');
```

```
module.exports = {
  data: new SlashCommandBuilder()
    .setName('mytickets')
    .setDescription('Перегляд ваших звернень у підтримку'),

  async execute(interaction) {
    const discordId = interaction.user.id;

    try {
      const [rows] = await pool.query(`
        SELECT t.id, t.subject, t.status, t.created_at
        FROM support_tickets t
        JOIN users u ON t.user_id = u.id
        WHERE u.discord_id = ?
        ORDER BY t.created_at DESC
      `, [discordId]);

      if (rows.length === 0) {
        return interaction.reply({ content: 'Ви ще не створювали звернень.',
ephemeral: true });
      }

      const statusMap = {
        'open': '· Відкрито',
        'in_progress': '☐ В обробці',
        'closed': '· Закрито'
      };
```

```
        const embed = new EmbedBuilder()
            .setColor(0x9B59B6)
            .setTitle('· Ваші звернення');

        rows.forEach(ticket => {
            embed.addFields({
                name: `Тікет #${ticket.id}: ${ticket.subject}`,
                value: `Статус: ${statusMap[ticket.status] || ticket.status}\nДата: ${new
Date(ticket.created_at).toLocaleDateString('uk-UA')}`,
                inline: false
            });
        });

        await interaction.reply({ embeds: [embed], ephemeral: true });

    } catch (error) {
        console.error('Помилка при отриманні тікетів:', error);
        await interaction.reply({ content: 'Помилка завантаження списку.',
ephemeral: true });
    }
  },
};
```

Додаток Н Команда /ticket

```
const { SlashCommandBuilder, EmbedBuilder } = require('discord.js');
const pool = require('../../database/db.js');
```

```
module.exports = {
  data: new SlashCommandBuilder()
    .setName('ticket')
    .setDescription('Деталі конкретного звернення')
    .addIntegerOption(option =>
      option.setName('ticket_id').setDescription('ID тікета').setRequired(true)),

  async execute(interaction) {
    const discordId = interaction.user.id;
    const ticketId = interaction.options.getInteger('ticket_id');

    try {
      // Перевіряємо власника тікета
      const [tickets] = await pool.query(`
        SELECT t.* FROM support_tickets t
        JOIN users u ON t.user_id = u.id
        WHERE t.id = ? AND u.discord_id = ?
      `, [ticketId, discordId]);

      if (tickets.length === 0) return interaction.reply({ content: 'Тікет не знайдено.', ephemeral: true });

      const ticket = tickets[0];

      // Завантажуємо відповіді (якщо вони є)
      const [messages] = await pool.query(
        'SELECT * FROM ticket_messages WHERE ticket_id = ? ORDER BY created_at ASC',
```

```
            [ticketId]
        );

        const embed = new EmbedBuilder()
            .setColor(0x34495E)
            .setTitle(`· Тікет #${ticket.id}: ${ticket.subject}`)
            .setDescription(`**Ваше повідомлення:**\n${ticket.message}`)
            .addFields({ name: 'Статус', value: ticket.status, inline: true });

        if (messages.length > 0) {
            let chatHistory = '';
            messages.forEach(msg => {
                const role = msg.sender_type === 'admin' ? '□ Адмін' : '· Ви';
                chatHistory += `**${role}:** ${msg.message}\n*${new
Date(msg.created_at).toLocaleString('uk-UA')}*\n\n`;
            });
            embed.addFields({ name: 'Діалог:', value: chatHistory });
        } else if (ticket.status === 'open') {
            embed.addFields({ name: 'Діалог:', value: 'Відповідей поки немає.
Очікуйте.' });
        }

        await interaction.reply({ embeds: [embed], ephemeral: true });

    } catch (error) {
        console.error('Помилка при отриманні деталей тікета:', error);
        await interaction.reply({ content: 'Помилка завантаження даних.',
ephemeral: true });
```

```
    }
  },
};
```

Додаток О Команда /addproduct

```
const { SlashCommandBuilder, PermissionFlagsBits, EmbedBuilder } =
require('discord.js');
const pool = require('../../database/db.js');

module.exports = {
  data: new SlashCommandBuilder()
    .setName('addproduct')
    .setDescription('Додавання нового товару (тільки для адміністраторів)')
    .setDefaultMemberPermissions(PermissionFlagsBits.Administrator)
    .addStringOption(option => option.setName('name').setDescription('Назва
товару').setRequired(true))
    .addNumberOption(option =>
option.setName('price').setDescription('Ціна').setRequired(true))
    .addIntegerOption(option => option.setName('stock').setDescription('Кількість
на складі').setRequired(true))
    .addIntegerOption(option => option.setName('category_id').setDescription('ID
категорії (1 - Комплектуючі, 2 - Периферія)').setRequired(true))
    .addStringOption(option => option.setName('description').setDescription('Опис
товару').setRequired(false)),

  async execute(interaction) {
    const name = interaction.options.getString('name');
    const price = interaction.options.getNumber('price');
    const stock = interaction.options.getInteger('stock');
    const categoryId = interaction.options.getInteger('category_id');
```

```
    const description = interaction.options.getString('description') || 'Опис відсутній';
    const discordId = interaction.user.id;

    try {
      // Шукаємо адміна в базі
      const [users] = await pool.query('SELECT id FROM users WHERE discord_id = ?', [discordId]);
      if (users.length === 0) return interaction.reply({ content: 'Вас немає в базі.', ephemeral: true });
      const adminId = users[0].id;

      // Додаємо товар
      const [result] = await pool.query(
        'INSERT INTO products (category_id, name, description, price, stock_quantity, is_active) VALUES (?, ?, ?, ?, ?, TRUE)',
        [categoryId, name, description, price, stock]
      );

      const productId = result.insertId;

      // Пишемо лог
      await pool.query(
        'INSERT INTO logs (user_id, action_type, description) VALUES (?, ?, ?)',
        [adminId, 'add_product', `Додано товар #${productId}: ${name}`]
      );

      const embed = new EmbedBuilder()
```

```
            .setColor(0x00FF00)
            .setTitle('✅ Товар успішно додано')
            .addFields(
                { name: 'ID Товару', value: productId.toString(), inline: true },
                { name: 'Назва', value: name, inline: true },
                { name: 'Ціна', value: `${price} грн`, inline: true },
                { name: 'Залишок', value: `${stock} шт.`, inline: true }
            );

        await interaction.reply({ embeds: [embed], ephemeral: true });

    } catch (error) {
        console.error('Помилка при додаванні товару:', error);
        await interaction.reply({ content: 'Помилка бази даних.', ephemeral: true });
    }
  },
};
```

Додаток П Команда /editproduct

```
const { SlashCommandBuilder, PermissionFlagsBits, EmbedBuilder } =
require('discord.js');
const pool = require('../../database/db.js');

module.exports = {
  data: new SlashCommandBuilder()
    .setName('editproduct')
    .setDescription('Редагування ціни або залишку товару (тільки для
адміністраторів)')
```

```
        .setDefaultMemberPermissions(PermissionFlagsBits.Administrator)
        .addIntegerOption(option => option.setName('product_id').setDescription('ID товару').setRequired(true))
        .addNumberOption(option => option.setName('new_price').setDescription('Нова ціна').setRequired(false))
        .addIntegerOption(option => option.setName('new_stock').setDescription('Новий залишок на складі').setRequired(false)),

    async execute(interaction) {
        const productId = interaction.options.getInteger('product_id');
        const newPrice = interaction.options.getNumber('new_price');
        const newStock = interaction.options.getInteger('new_stock');
        const discordId = interaction.user.id;

        if (newPrice === null && newStock === null) {
            return interaction.reply({ content: 'Ви повинні вказати хоча б один параметр для зміни (ціну або залишок).', ephemeral: true });
        }

        try {
            // Перевіряємо товар
            const [products] = await pool.query('SELECT name FROM products WHERE id = ?', [productId]);
            if (products.length === 0) return interaction.reply({ content: 'Товар не знайдено.', ephemeral: true });

            const [users] = await pool.query('SELECT id FROM users WHERE discord_id = ?', [discordId]);
```

```
        const adminId = users[0].id;

        let updateFields = [];
        let updateValues = [];
        let logDesc = `Оновлено товар #${productId}:`;

        if (newPrice !== null) {
            updateFields.push('price = ?');
            updateValues.push(newPrice);
            logDesc += ` нова ціна = ${newPrice};`;
        }
        if (newStock !== null) {
            updateFields.push('stock_quantity = ?');
            updateValues.push(newStock);
            logDesc += ` новий залишок = ${newStock};`;
        }

        updateValues.push(productId);
        const updateQuery = `UPDATE products SET ${updateFields.join(', ')}
WHERE id = ?`;

        await pool.query(updateQuery, updateValues);

        // Пишемо лог
        await pool.query('INSERT INTO logs (user_id, action_type, description)
VALUES (?, ?, ?)', [adminId, 'edit_product', logDesc]);

        const embed = new EmbedBuilder()
```

```
            .setColor(0xF39C12)
            .setTitle(`□ Товар #${productId} оновлено`)
            .setDescription(logDesc);

        await interaction.reply({ embeds: [embed], ephemeral: true });

    } catch (error) {
        console.error('Помилка при редагуванні товару:', error);
        await interaction.reply({ content: 'Помилка бази даних.', ephemeral: true });
    }
  },
};
```

Додаток Р Команда /orders

```
const { SlashCommandBuilder, PermissionFlagsBits, EmbedBuilder } =
require('discord.js');
const pool = require('../../database/db.js');

module.exports = {
  data: new SlashCommandBuilder()
    .setName('orders')
    .setDescription('Перегляд останніх замовлень (тільки для адміністраторів)')
    .setDefaultMemberPermissions(PermissionFlagsBits.Administrator)
    .addStringOption(option =>
      option.setName('status')
        .setDescription('Фільтр за статусом')
        .addChoices(
          { name: 'Нове', value: 'new' },
```

```
                { name: 'Підтверджене', value: 'confirmed' },
                { name: 'В обробці', value: 'processing' },
                { name: 'Завершене', value: 'completed' },
                { name: 'Скасоване', value: 'canceled' }
            )
            .setRequired(false)),

    async execute(interaction) {
        const statusFilter = interaction.options.getString('status');

        try {
            let query = `
                SELECT o.id, o.status, o.total_amount, o.created_at, u.display_name,
u.discord_id
                FROM orders o
                JOIN users u ON o.user_id = u.id
            `;
            let queryParams = [];

            if (statusFilter) {
                query += ' WHERE o.status = ?';
                queryParams.push(statusFilter);
            }

            query += ' ORDER BY o.created_at DESC LIMIT 10';

            const [rows] = await pool.query(query, queryParams);
```

```
        if (rows.length === 0) {
            return interaction.reply({ content: 'Замовлень не знайдено.', ephemeral:
true });
        }

        const embed = new EmbedBuilder()
            .setColor(0x3498DB)
            .setTitle(statusFilter ? `· Замовлення (Статус: ${statusFilter})` : '· Останні
10 замовлень');

        const statusMap = {
            'new': '· Нове',
            'confirmed': '✅ Підтверджене',
            'processing': '☐ В обробці',
            'completed': '· Завершене',
            'canceled': '❌ Скасоване'
        };

        rows.forEach(order => {
            embed.addFields({
                name: `Замовлення #${order.id} | Сума: ${order.total_amount} грн`,
                value: `Клієнт: ${order.display_name}
(<@${order.discord_id}>)\nСтатус: **${statusMap[order.status] ||
order.status}**\nДата: ${new Date(order.created_at).toLocaleString('uk-UA')}`,
                inline: false
            });
```

```
        });

        await interaction.reply({ embeds: [embed], ephemeral: true });

    } catch (error) {
        console.error('Помилка при завантаженні замовлень:', error);
        await interaction.reply({ content: 'Помилка бази даних.', ephemeral: true });
    }
  },
};
```

Додаток С Команда /setorderstatus

```
const { SlashCommandBuilder, PermissionFlagsBits, EmbedBuilder } =
require('discord.js');
const pool = require('../../database/db.js');

module.exports = {
  data: new SlashCommandBuilder()
    .setName('setorderstatus')
    .setDescription('Зміна статусу замовлення (тільки для адміністраторів)')
    .setDefaultMemberPermissions(PermissionFlagsBits.Administrator)
    .addIntegerOption(option => option.setName('order_id').setDescription('ID
замовлення').setRequired(true))
    .addStringOption(option =>
      option.setName('status')
        .setDescription('Новий статус')
        .addChoices(
          { name: 'Нове', value: 'new' },
```

```
                { name: 'Підтверджене', value: 'confirmed' },
                { name: 'В обробці', value: 'processing' },
                { name: 'Завершене', value: 'completed' },
                { name: 'Скасоване', value: 'canceled' }
            )
            .setRequired(true)),

    async execute(interaction) {
        const orderId = interaction.options.getInteger('order_id');
        const newStatus = interaction.options.getString('status');
        const discordId = interaction.user.id;

        const connection = await pool.getConnection();

        try {
            await connection.beginTransaction();

            // Отримуємо поточний статус та ID користувача
            const [orders] = await connection.query('SELECT status, user_id FROM
orders WHERE id = ? FOR UPDATE', [orderId]);
            if (orders.length === 0) throw new Error('NOT_FOUND');

            const oldStatus = orders[0].status;

            if (oldStatus === newStatus) {
                throw new Error('SAME_STATUS');
            }
```

```
        const [admins] = await connection.query('SELECT id FROM users WHERE discord_id = ?', [discordId]);
        const adminId = admins[0].id;

        // Оновлюємо статус в таблиці orders
        await connection.query('UPDATE orders SET status = ? WHERE id = ?', [newStatus, orderId]);

        // Пишемо історію статусів
        await connection.query(
          'INSERT INTO order_status_history (order_id, old_status, new_status, changed_by) VALUES (?, ?, ?, ?)',
          [orderId, oldStatus, newStatus, adminId]
        );

        // Пишемо загальний лог
        await connection.query(
          'INSERT INTO logs (user_id, action_type, description) VALUES (?, ?, ?)',
          [adminId, 'change_order_status', `Замовлення #${orderId}: змінено статус з ${oldStatus} на ${newStatus}`]
        );

        await connection.commit();

        const embed = new EmbedBuilder()
          .setColor(0x2ECC71)
          .setTitle(`✅ Статус замовлення #${orderId} змінено`)
```

```
            .setDescription(`Новий статус: **${newStatus}**`);

        await interaction.reply({ embeds: [embed], ephemeral: true });

    } catch (error) {
        await connection.rollback();

        if (error.message === 'NOT_FOUND') return interaction.reply({ content: 'Замовлення не знайдено.', ephemeral: true });
        if (error.message === 'SAME_STATUS') return interaction.reply({ content: 'Це замовлення вже має такий статус.', ephemeral: true });

        console.error('Помилка при зміні статусу:', error);
        await interaction.reply({ content: 'Виникла помилка під час зміни статусу.', ephemeral: true });
    } finally {
        connection.release();
    }
  },
};
```

Додаток Т Команда /tickets

```
const { SlashCommandBuilder, PermissionFlagsBits, EmbedBuilder } = require('discord.js');
const pool = require('../../database/db.js');

module.exports = {
  data: new SlashCommandBuilder()
    .setName('tickets')
```

```
        .setDescription('Перегляд звернень користувачів (тільки для адміністраторів)')
        .setDefaultMemberPermissions(PermissionFlagsBits.Administrator)
        .addStringOption(option =>
            option.setName('status')
                .setDescription('Фільтр за статусом (за замовчуванням: Відкриті)')
                .addChoices(
                    { name: 'Відкриті', value: 'open' },
                    { name: 'В обробці', value: 'in_progress' },
                    { name: 'Закриті', value: 'closed' }
                )
                .setRequired(false)),

    async execute(interaction) {
        // Якщо фільтр не вказано, показуємо відкриті
        const statusFilter = interaction.options.getString('status') || 'open';

        try {
            const [rows] = await pool.query(`
                SELECT t.id, t.subject, t.status, t.created_at, u.display_name, u.discord_id
                FROM support_tickets t
                JOIN users u ON t.user_id = u.id
                WHERE t.status = ?
                ORDER BY t.created_at ASC
                LIMIT 10
            `, [statusFilter]);
```

```
        if (rows.length === 0) {
            return interaction.reply({ content: `Тікетів зі статусом "${statusFilter}" не знайдено.`, ephemeral: true });
        }

        const statusMap = {
            'open': '· Відкриті',
            'in_progress': '☐ В обробці',
            'closed': '· Закриті'
        };

        const embed = new EmbedBuilder()
            .setColor(0x9B59B6)
            .setTitle(`· Список тікетів (${statusMap[statusFilter]})`);

        rows.forEach(ticket => {
            embed.addFields({
                name: `Тікет #${ticket.id}: ${ticket.subject}`,
                value: `Від: ${ticket.display_name} (<@${ticket.discord_id}>)\nСтворено: ${new Date(ticket.created_at).toLocaleString('uk-UA')}`,
                inline: false
            });
        });

        await interaction.reply({ embeds: [embed], ephemeral: true });
```

```
        } catch (error) {
            console.error('Помилка при завантаженні тікетів:', error);
            await interaction.reply({ content: 'Помилка бази даних.', ephemeral: true });
        }
    },
};
```

Додаток У Команда /replyticket

```
const { SlashCommandBuilder, PermissionFlagsBits, EmbedBuilder } =
require('discord.js');
const pool = require('../../database/db.js');

module.exports = {
    data: new SlashCommandBuilder()
        .setName('replyticket')
        .setDescription('Відповідь на звернення користувача (тільки для
адміністраторів)')
        .setDefaultMemberPermissions(PermissionFlagsBits.Administrator)
        .addIntegerOption(option => option.setName('ticket_id').setDescription('ID
тікета').setRequired(true))
        .addStringOption(option => option.setName('message').setDescription('Текст
відповіді').setRequired(true))
        .addBooleanOption(option =>
option.setName('close_ticket').setDescription('Закрити тікет після цієї
відповіді?').setRequired(false)),

    async execute(interaction) {
        const ticketId = interaction.options.getInteger('ticket_id');
        const replyMessage = interaction.options.getString('message');
        const closeTicket = interaction.options.getBoolean('close_ticket') || false;
```

```
const discordId = interaction.user.id;

const connection = await pool.getConnection();

try {
  await connection.beginTransaction();

  // Перевіряємо, чи існує тікет і чи не закритий він
  const [tickets] = await connection.query('SELECT status, user_id FROM support_tickets WHERE id = ? FOR UPDATE', [ticketId]);
  if (tickets.length === 0) throw new Error('NOT_FOUND');
  if (tickets[0].status === 'closed') throw new Error('ALREADY_CLOSED');

  // Отримуємо ID адміна
  const [admins] = await connection.query('SELECT id FROM users WHERE discord_id = ?', [discordId]);
  const adminId = admins[0].id;

  // Записуємо повідомлення
  await connection.query(
    'INSERT INTO ticket_messages (ticket_id, sender_type, sender_id, message) VALUES (?, ?, ?, ?)',
    [ticketId, 'admin', adminId, replyMessage]
  );

  // Визначаємо новий статус тікета
  const newStatus = closeTicket ? 'closed' : 'in_progress';
```

```
        // Оновлюємо статус тікета
        await connection.query('UPDATE support_tickets SET status = ? WHERE id = ?', [newStatus, ticketId]);

        // Пишемо в лог
        await connection.query(
          'INSERT INTO logs (user_id, action_type, description) VALUES (?, ?, ?)',
          [adminId, 'reply_ticket', `Відповідь на тікет #${ticketId}. Статус змінено на: ${newStatus}`]
        );

        await connection.commit();

        const embed = new EmbedBuilder()
          .setColor(0x2ECC71)
          .setTitle(`✅ Відповідь на тікет #${ticketId} надіслано`)
          .setDescription(`**Ваша відповідь:**\n${replyMessage}`)
          .addFields({ name: 'Новий статус тікета', value: newStatus === 'closed' ? '🔒 Закрито' : '🔓 В обробці' });

        await interaction.reply({ embeds: [embed], ephemeral: true });

        // ОПЦІОНАЛЬНО: Спроба надіслати приватне повідомлення клієнту (може впасти, якщо у клієнта закрита лічка)
        try {
          const [users] = await pool.query('SELECT discord_id FROM users WHERE id = ?', [tickets[0].user_id]);
          const clientUser = await interaction.client.users.fetch(users[0].discord_id);
```

```
            if (clientUser) {
                await clientUser.send(`· Оновлення по вашому тікету #${ticketId}:\n**Адміністратор:** ${replyMessage}\n*Перевірити статус: /ticket ticket_id:${ticketId}*`);
            }
        } catch (err) {
            console.log(`Не вдалося надіслати ПП користувачу по тікету #${ticketId}. Лічка закрита.`);
        }

    } catch (error) {
        await connection.rollback();

        if (error.message === 'NOT_FOUND') return interaction.reply({ content: 'Тікет не знайдено.', ephemeral: true });
        if (error.message === 'ALREADY_CLOSED') return interaction.reply({ content: 'Цей тікет вже закрито. Ви не можете на нього відповісти.', ephemeral: true });

        console.error('Помилка при відповіді на тікет:', error);
        await interaction.reply({ content: 'Критична помилка БД.', ephemeral: true });
    } finally {
        connection.release();
    }
  },
};
```

Додаток Ф Команда /stats

```
const { SlashCommandBuilder, PermissionFlagsBits, EmbedBuilder } =
require('discord.js');

const pool = require('../../database/db.js');


module.exports = {

   data: new SlashCommandBuilder()

      .setName('stats')

      .setDescription('Статистика системи (тільки для адміністраторів)')

      .setDefaultMemberPermissions(PermissionFlagsBits.Administrator),


   async execute(interaction) {

      try {

         // Робимо всі запити паралельно для швидкості

         const [userCount] = await pool.query('SELECT COUNT(*) as count FROM
users');

         const [productCount] = await pool.query('SELECT COUNT(*) as count
FROM products WHERE is_active = TRUE');

         const [orderCount] = await pool.query('SELECT COUNT(*) as count FROM
orders');

         const [ticketCount] = await pool.query('SELECT COUNT(*) as count FROM
support_tickets WHERE status = "open"');

         const [logCount] = await pool.query('SELECT COUNT(*) as count FROM
logs');


         const embed = new EmbedBuilder()

            .setColor(0x3498DB)

            .setTitle('· Статистика системи ServiceBridge')

            .addFields(
```

```
                { name: '· Користувачі', value: userCount[0].count.toString(), inline:
true },
                { name: '· Активні товари', value: productCount[0].count.toString(),
inline: true },
                { name: '· Всього замовлень', value: orderCount[0].count.toString(),
inline: true },
                { name: '· Відкриті тікети', value: ticketCount[0].count.toString(),
inline: true },
                { name: '· Записів у журналі', value: logCount[0].count.toString(),
inline: true }
            )
            .setTimestamp();

        await interaction.reply({ embeds: [embed], ephemeral: true });
    } catch (error) {
        console.error('Помилка статистики:', error);
        await interaction.reply({ content: 'Помилка завантаження статистики.',
ephemeral: true });
    }
  }
};
```

Додаток Х Команда /logs

```
const { SlashCommandBuilder, PermissionFlagsBits, EmbedBuilder } =
require('discord.js');
const pool = require('../../database/db.js');

module.exports = {
  data: new SlashCommandBuilder()
    .setName('logs')
```

```
        .setDescription('Перегляд журналу дій адміністраторів (тільки для адміністраторів)')
        .setDefaultMemberPermissions(PermissionFlagsBits.Administrator),

    async execute(interaction) {
        try {
            const [rows] = await pool.query(`
                SELECT l.action_type, l.description, l.created_at, u.display_name
                FROM logs l
                LEFT JOIN users u ON l.user_id = u.id
                ORDER BY l.created_at DESC
                LIMIT 10
            `);

            if (rows.length === 0) {
                return interaction.reply({ content: 'Журнал порожній.', ephemeral: true });
            }

            const embed = new EmbedBuilder()
                .setColor(0x95A5A6)
                .setTitle('□ Журнал останніх дій (Logs)');

            rows.forEach(log => {
                const date = new Date(log.created_at).toLocaleString('uk-UA');
                embed.addFields({
                    name: `[${date}] ${log.action_type}`,
                    value: `**Адмін:** ${log.display_name || 'Система'}\n**Дія:** ${log.description}`,
```

```
                inline: false
            });
        });

        await interaction.reply({ embeds: [embed], ephemeral: true });
    } catch (error) {
        console.error('Помилка логів:', error);
        await interaction.reply({ content: 'Помилка завантаження журналу.', ephemeral: true });
    }
  }
};
```