Міністерство освіти і науки України
Університет митної справи та фінансів

Факультет інноваційних технологій
Кафедра комп'ютерних наук та інженерії програмного забезпечення

Кваліфікаційна робота бакалавра

на тему «Проектування та реалізація веб-системи електронної комерції на мікросервісній архітектурі»

Виконав: студент групи ІПЗ21-1
Спеціальність 121 «Інженерія програмного забезпечення»
Панченко К. В.
(прізвище та ініціали)
Керівник к.т.н., доц. Чупілко Т. А.
(науковий ступінь, вчене звання, прізвище та ініціали)
Рецензент Університет митної справи та фінансів
(місце роботи)
доцент кафедри кібербезпеки та інформаційних технологій
(посада)
к.т.н., доц. Савченко Ю. В.
(науковий ступінь, вчене звання, прізвище та ініціали)

Дніпро – 2025

АНОТАЦІЯ

Панченко К. В. Проектування та реалізація веб-системи електронної комерції на мікросервісній архітектурі.

Кваліфікаційна робота на здобуття освітнього ступеня бакалавр за спеціальністю 121 «Інженерія програмного забезпечення». – Університет митної справи та фінансів, Дніпро, 2025.

Пояснювальна записка: 94 с., 25 рисунків, 21 джерело.

Кваліфікаційна робота спрямована на створення веб-додатку онлайн магазину, що займається продажем товарів електронної комерції: акумулятори, інвертори, BMS (Battery Management System), тощо. Функціонал системи включає: створення замовлення, поповнення асортименту товарів, email-розсилка, замовлення консультації в адміністрації сайту, створення коментарів до товарів, пошук пропозицій за критеріями користувача .

У рамках проекту спроектовані та розгорнуті реляційні бази даних для різних мікросервісів, налаштовано брокер повідомлень для асинхронного спілкування між компонентами системи, створений фронтенд та бекенд додатки.

Програмний код створений з використанням мов програмування Python, JavaScript, Java, фреймворків FastAPI, Spring, Angular, та технологій Docker, PostgreSQL, RabbitMQ, Minio.

Також підготовлено документацію, яка включає опис усіх доступних API-ендпоінтів (Application Programming Interface), приклади запитів і відповідей, що полегшує користувачам взаємодію із системою та її інтеграцію в наявні бізнес-процеси.

Додаток спрямований на простий та в той же час зрозумілий UI/UX (User Interface / User Experience), що надає користувачам можливість зробити заказ без додаткових проблем. В той же час, серверний додаток є достатньо гнучким, щоб використовувати його в іншій доменній сфері.

Ключові слова: серверний додаток, брокер повідомлень, мікросервіси, база даних, каталог товарів, фронтенд, розробка API, гарний UX, інфраструктура.

ABSTRACT

Panchenko K. V. Design and Implementation of an E-Commerce Web System Based on Microservice Architecture.

Qualification work for obtaining the bachelor's degree in specialty 121 "Software engineering". – University of Customs and Finance, Dnipro, 2025.

Explanatory note: 94 pages, 25 images, 21 references.

This qualification work is aimed at developing a web application for an online store specializing in e-commerce products such as batteries, inverters, BMS modules, and more. The system's functionality includes order creation, product catalog management, email notifications, requesting consultations with site administrators, product commenting, and searching for offers based on user-defined criteria.

As part of the project, relational databases were designed for various microservices, a message broker was configured for asynchronous communication between system components, and both frontend and backend applications were developed.

The software code was developed using the programming languages Python, JavaScript, and Java; the frameworks FastAPI, Spring, and Angular; and the technologies Docker, PostgreSQL, RabbitMQ, and MinIO.

In addition, documentation was prepared that includes a description of all available API endpoints (Application Programming Interface), along with examples of requests and responses, facilitating user interaction with the system and its integration into existing business processes.

The application focuses on a simple yet intuitive UI/UX (User Interface / User Experience), enabling users to place orders without any additional difficulties. At the same time, the server-side application is flexible enough to be adapted for use in other domains.

Keywords: server-side application, message broker, microservices, database, product catalog, frontend, API development, user-friendly UX, infrastructure.

ЗМІСТ

ВСТУП

Актуальність застосунку для системи електронної комерції полягає у вирішенні проблем, з якими стикаються сучасні онлайн-магазини в галузі енергетичних товарів.

Автоматизація процесів замовлення та управління товарним асортиментом дозволяє значно скоротити операційні витрати та підвищити ефективність обслуговування клієнтів. Застосування сучасних технологій для обробки транзакцій, керування залишками на складі та обміну даними між мікросервісами забезпечує високу точність виконання замовлень, швидку реакцію на запити клієнтів і можливість відстеження статусу кожного замовлення в режимі реального часу. Це дозволяє не лише підвищити рівень обслуговування, а й підтримувати конкурентоспроможність на ринку електронної комерції.

У сучасних умовах стрімкого зростання обсягів онлайн-продажів та підвищених вимог до швидкості й точності обробки замовлень, використання сучасних інформаційних технологій у сфері електронної комерції стає критично необхідним. Розробка автоматизованої системи дозволяє ефективно керувати життєвим циклом замовлення — від його оформлення до доставки — мінімізуючи кількість помилок, скорочуючи час обробки та підвищуючи загальну продуктивність роботи сервісу. Саме тому створення надійного серверного застосунку на базі мікросервісної архітектури має важливе значення для забезпечення стабільності, масштабованості та конкурентоспроможності бізнесу на ринку електронної комерції.

Слід також виділити кілька ключових чинників, що зумовлюють необхідність розробки подібної системи. По-перше, стрімке зростання обсягів електронної комерції. Сучасний ринок динамічно розвивається, що призводить до збільшення кількості замовлень і потребує ефективних інструментів для їх обробки та доставки. Онлайн-магазини повинні забезпечувати швидке та надійне

виконання замовлень, щоб відповідати очікуванням клієнтів. По-друге, виникає потреба в підвищенні ефективності внутрішніх бізнес-процесів. У багатьох випадках саме неефективна організація процесів призводить до зниження продуктивності та зростання витрат. Впровадження автоматизованих рішень на базі мікросервісної архітектури дозволяє оптимізувати ці процеси, знизити витрати та забезпечити стабільну й масштабовану роботу системи.

Сьогодні існує багато рішень для організації електронної комерції, однак більшість із них або надто дорогі для малих і середніх підприємств, або не забезпечують достатньої гнучкості для адаптації під специфічні потреби конкретного онлайн-бізнесу, або мають дуже велику кількість функціоналу, що може бути непотрібною для бізнесу.

Практичне значення цієї дипломної роботи полягає у створенні повноцінного прикладного програмного забезпечення, яке може бути безпосередньо використане в комерційній діяльності для організації продажу товарів через інтернет. Застосунок розроблено із застосуванням сучасних архітектурних підходів, таких як мікросервісна архітектура та предметно-орієнтоване проектування (Domain-Driven Design), що забезпечує високу гнучкість, масштабованість та адаптивність системи до змін бізнес-вимог. Це робить програмний продукт не лише ефективним з точки зору виконання ключових бізнес-процесів, а й придатним до подальшого розвитку та інтеграції з іншими системами.

Метою даної дипломної роботи є автоматизація процесів управління замовленнями в онлайн-магазині, їх створення, розсилка повідомлень про статус замовлення. Основним завданням серверного застосунку є забезпечення можливості отримання деталей замовлень, наповнення каталогу товарів магазину, автоматизація управління бізнес-процесами, пов'язаними з продажом і доставкою товарів, та відображення цього функціоналу у браузері користувача у вигляді фронтенд додатку.

Завданням дипломної роботи є наступне: проаналізувати системні вимоги, щоб визначити функціональні та нефункціональні вимоги до системи, розробити архітектуру системи та бази даних окремих мікросервісів, розробити програмний код та написати конфігураційних файлів для розгортання додатку на сервері з використанням Docker й Docker Compose, і створення документації для розробників системи та технічних користувачів з використанням OpenAPI. В кваліфікаційній роботі також треба створити окремі серверні додатки (мікросервіси), які забезпечуватимуть наступні можливості:

a. Користувачі можуть створювати замовлення на базі каталогу сайту.

b. Користувачі можуть залишати коментарі для товарів сайту.

c. Застосунок автоматизує типові операції, пов'язані з обробкою замовлень, що сприяє зниженню навантаження на працівників та мінімізує ризик виникнення помилок у процесі.

d. Клієнту сайту можуть переглядати товари по категоріям та своїм критеріям пошуку

Об'єктом дослідження є система електронної комерції у сфері реалізації енергетичних товарів, зокрема її інформаційна інфраструктура, архітектурні підходи до організації серверного програмного забезпечення та технічні засоби реалізації.

Предметом дослідження є методи та засоби проектування, розробки й впровадження серверного застосунку для електронної комерції з використанням мікросервісної архітектури, сучасних технологій, а також принципів структуризації програмного коду.

Для розроблення додатку використовуються наступні технології: мова програмування Python, фреймворк для веб-додатків FastAPI, технології розгортання Docker та Docker Compose, реляційна база даних для збереження інформації PostgreSQL, брокер повідомленнь RabbitMQ, документация OpenAPI, JavaScript для створення фронтенд частини, та мова програмування Java. Код реалізовано з дотриманням принципів модульності та масштабованості, що

забезпечує зручність у подальшому розширенні функціоналу та додаванні нових компонентів. Детально описана конфігурація розгортання в середовищі завдяки Docker'у дає змогу швидко та безперешкодно інсталювати систему на будь-якому сумісному сервері.

Впровадження розробленого серверного застосунку в систему електронної комерції очікувано забезпечить відчутний техніко-економічний ефект. По-перше, автоматизація процесів обробки замовлень дозволить зменшити витрати на ручну працю, що позитивно вплине на загальні експлуатаційні витрати платформи. По-друге, підвищення точності та швидкості виконання замовлень сприятиме зростанню рівня задоволеності клієнтів. Крім того, функція розсилки пошти при зміні статусів замовлення дає змогу оперативно виявляти і реагувати на затримки або помилки, що загалом підвищує ефективність роботи всієї системи.

Завдяки застосуванню мікросервісної архітектури та сучасних технологій, забезпечується гнучкість, розширюваність та стійкість програмного рішення до змін у бізнес-логіці та навантаженні.

Інтеграція з зовнішніми сервісами, такими як електронна пошта та служби обміну повідомленнями, забезпечує оперативну комунікацію з клієнтами та адміністрацією, що дозволяє не лише підтримувати високий рівень обслуговування, а й оперативно виявляти та вирішувати проблемні ситуації. Завдяки впровадженню таких функцій, як інформування клієнтів про статус замовлення, можливість залишати коментарі до товарів і запитувати консультації, платформа набуває додаткової цінності як для бізнесу, так і для користувачів.

У процесі виконання роботи були застосовані методи аналізу та обробки інформації, принципи проектування програмного забезпечення, а також підходи до побудови та оптимізації реляційних баз даних. Також були визначені системні вимоги, необхідні для коректного функціонування та розгортання розроблюваного веб-додатку електронної комерції.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ. ПОСТАНОВКА ЗАВДАННЯ

1.1. Аналіз предметної області

Сучасний ринок електронної комерції, зокрема в сфері продажу енергетичних товарів, таких як інвертори, акумулятори, сонячні панелі та системи керування живленням, демонструє стабільне зростання попиту [1]. Онлайн-магазини, що спеціалізуються на таких продуктах, стикаються з низкою викликів, пов'язаних з обробкою великої кількості замовлень, оновленням товарного асортименту, веденням обліку залишків та забезпеченням якісної клієнтської підтримки. У зв'язку з цим виникає об'єктивна потреба в автоматизованій системі, яка здатна ефективно управляти всіма цими аспектами в режимі реального часу.

Один із ключових напрямів автоматизації [2] — це цифрове управління замовленнями, починаючи від їх створення клієнтом до завершення етапу доставки. У процесі обробки замовлення важливо оперативно змінювати його статус, надсилати відповідні повідомлення клієнту та фіксувати коментарі залишені користувачами сайту. Автоматизоване вирішення цих завдань дозволяє не лише знизити навантаження на працівників компанії, а й значно мінімізувати ризики виникнення помилок, пов'язаних із людським фактором.

Окрема увага в системі приділяється управлінню асортиментом товарів. Каталог має бути гнучким і масштабованим: підтримувати велику кількість категорій, характеристик, фільтрів і варіантів пошуку. Це створює додаткову цінність для користувача, який може швидко знайти потрібний товар, а також для адміністратора, який має можливість оперативно оновлювати дані про наявність, опис, ціну та технічні параметри.

Не менш важливим є механізм взаємодії користувача з платформою, що реалізується через коментарі до товарів, можливість поставити запитання або залишити відгук через спеціальну відведену для цього форму на сайті. Такий

функціонал не лише покращує взаємодію між продавцем і покупцем, а й сприяє підвищенню довіри до онлайн-магазину, особливо в галузі складних технічних продуктів. Коментарі та питання дозволяють потенційним покупцям отримати додаткову інформацію про товар, з'ясувати практичні аспекти його використання та прийняти зважене рішення щодо покупки. Для продавця ж ці елементи стають джерелом зворотного зв'язку, що допомагає виявити недоліки, краще розуміти потреби клієнтів і підвищувати якість сервісу.

Система також має інтегруватися з email-розсилкою для оперативного інформування як клієнтів, так і адміністраторів щодо змін у статусі замовлень або виникнення важливих подій. Це дає змогу підвищити прозорість процесів, зменшити кількість звернень до служби підтримки та вчасно реагувати на затримки чи збої в роботі. Крім того, автоматичні повідомлення можуть використовуватися не лише для інформаційних цілей, але й для підвищення залученості користувачів — наприклад, шляхом нагадування про незавершені замовлення, повідомлень про наявність очікуваних товарів або пропозицій щодо схожої продукції. Така комунікація сприяє утриманню клієнтів і зростанню конверсій, що є важливим чинником успішності електронної комерції.

Таким чином, функціональні можливості, орієнтовані на зручність користувача, безпосередньо впливають на якість взаємодії з платформою та загальний рівень довіри до бренду. Забезпечення зручного каналу для комунікації, своєчасне інформування та інтеграція з сучасними засобами оповіщення дозволяють сформувати позитивний досвід користування сервісом і забезпечити його конкурентоспроможність на ринку.

З огляду на зростаючий обсяг даних і навантаження, особливу увагу в предметній області займають вимоги до надійності, продуктивності та масштабованості системи. Вирішенням цих завдань є впровадження мікросервісної архітектури, яка дозволяє ізолювати ключові компоненти, масштабувати їх незалежно одне від одного, швидко адаптуватися до змін бізнес-вимог і зменшувати ризики при оновленні [3].

Важливою складовою предметної області є міжсервісна взаємодія. Для забезпечення цілісності інформації та стабільної роботи системи необхідно передбачити надійний обмін даними між окремими мікросервісами — такими як обробка замовлень, керування каталогом, обробка коментарів, email-нотифікації. Використання брокера повідомлень (наприклад, RabbitMQ) дозволяє гарантувати доставку повідомлень та обробку подій у асинхронному режимі [4].

Застосування такої моделі взаємодії дозволяє зменшити зв'язаність між сервісами, підвищити гнучкість у розвитку кожного з них окремо, а також забезпечити масштабованість усієї системи. У разі виникнення навантаження на окремі компоненти, система зберігає стабільність завдяки можливості обробки повідомлень у черзі. Такий підхід також дозволяє краще управляти помилками — у разі недоступності одного з мікросервісів повідомлення не втрачаються, а залишаються в черзі до моменту їх обробки.

На рівні предметної області важливо врахувати специфіку бізнес-процесів, які мають бути реалізовані у системі. До них належать не лише продаж товарів і оформлення замовлень, але й управління наявністю продукції, аналітика взаємодій із клієнтами, персоналізовані рекомендації, консультаційна підтримка та нотифікації. Усі ці процеси потребують синхронної і асинхронної взаємодії між компонентами системи.

Таким чином, аналіз показав, що предметна область охоплює широкий спектр бізнес-процесів, характерних для електронної комерції, включаючи автоматизацію типових операцій, оптимізацію внутрішніх процесів, обробку великого обсягу даних та забезпечення якісної взаємодії з клієнтами. Успішна реалізація системи вимагає правильного проектування архітектури та вибору сучасних технологій, що відповідають критеріям надійності, гнучкості та масштабованості. Особливу увагу слід приділяти правильному моделюванню домену, структурі сервісів та способам їх взаємодії, що у сукупності дозволяє побудувати стійку та ефективну інформаційну систему для підтримки комерційної діяльності онлайн-магазину.

1.2. Аналіз існуючих рішень

На ринку програмного забезпечення для електронної комерції існує велика кількість готових платформ, які дозволяють швидко запустити онлайн-магазин. Серед найпоширеніших рішень варто виділити такі платформи як Shopify, Magento (Adobe Commerce), WooCommerce, OpenCart, а також Prom.ua для українського сегменту ринку. Кожна з них має свої переваги та недоліки, але жодна не задовольняє повною мірою специфічні потреби бізнесу, або є занадто загальною та ускладненою для онлайн магазину електронної комерції.

Наприклад, Shopify [5] — це популярна SaaS-платформа, яка пропонує швидкий запуск магазину без глибоких технічних знань. Проте вона має обмежену гнучкість у розширенні функціональності, а створення кастомної логіки, зокрема обробки замовлень або інтеграції з окремими сервісами (наприклад, внутрішньою ERP-системою [6] чи спеціалізованим складським обліком), вимагає додаткових витрат, спеціалістів, або використання сторонніх платних додатків. Додатково, через те що Shopify це SaaS-платформа [7], користувачі, а саме адміністрація сайту, є обмеженим лише тим функціоналом, що надає платформа.

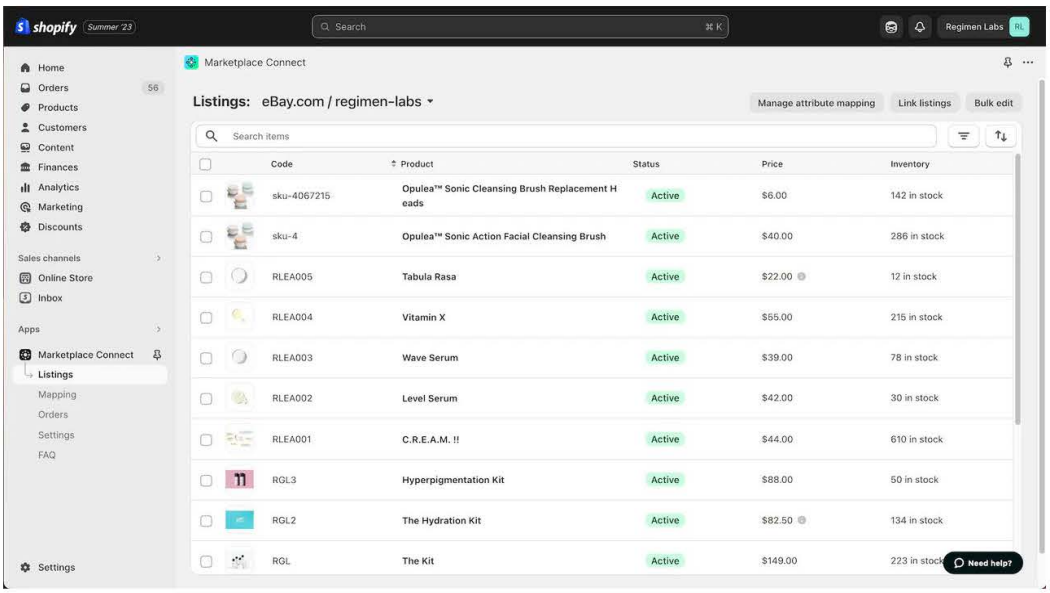

Рисунок 1.1. – Інтерфейс Shopify

Також треба відмітити, що Shopify не є популярним сервісом в Україні - це може бути ключовим моментом при виборі сервісу для створення свого магазину.

Magento (Adobe Commerce) [8] — платформа з відкритим кодом, орієнтована на великі бізнеси. Вона дозволяє створювати складну інфраструктуру, масштабувати систему та інтегрувати з зовнішніми сервісами. Проте налаштування та підтримка Magento є ресурсоємними, що робить її малопридатною для середніх або невеликих компаній. Також потрібно враховувати значне споживання ресурсів серверу та складність навчання персоналу.

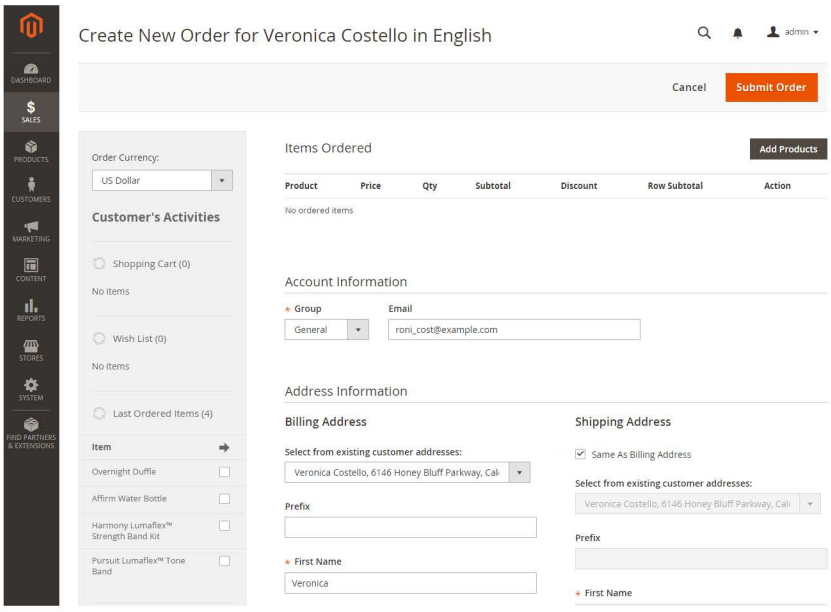

Рисунок 1.2. – Інтерфейс Magento

Magento має дуже багато функціоналу, який може бути зайвим для сайту з продажу електронної комерції. Цей рішення більш гнучке та є загальним для магазинів та e-commerce. Через це, треба наймати додаткових спеціалістів, що будуть адмініструвати та знати як налаштувати та працювати з Magento.

WooCommerce [9], який працює як плагін для WordPress, підходить для малих магазинів з простим асортиментом. Однак при спробі масштабувати

магазин із великою кількістю категорій, фільтрів, технічних характеристик і специфічною логікою доставки, WooCommerce швидко втрачає продуктивність і потребує складних оптимізацій. Крім того, він не передбачає розділення бізнес-логіки на мікросервіси, що ускладнює супровід системи в довгостроковій перспективі.

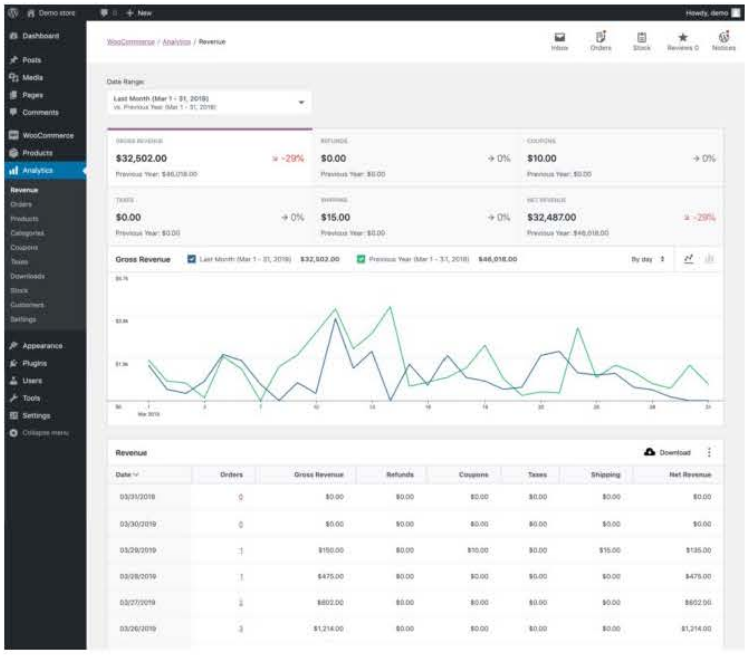

Рисунок 1.3. – Інтерфейс WooCommerce

Серед українських рішень Prom.ua виступає як універсальна платформа для малого бізнесу, однак вона має обмеження в налаштуванні інтерфейсу, слабку інтеграцію з кастомним бекендом і неможливість самостійно керувати архітектурою або базами даних, що робить її малопридатною для кастомізованих рішень зі складною логікою.

Також, Prom.ua є маркетплейсом, що може не підходити для бізнесу, якщо він вимагає створити саме свій сервіс зі своєю бізнес логікою, правилами та картою розвитку додатку. Даний застосунок можна розглядати як SaaS рішення, що, як було вже сказано, може не підходити, якщо функціонал не задовольняє потреби бізнесу.

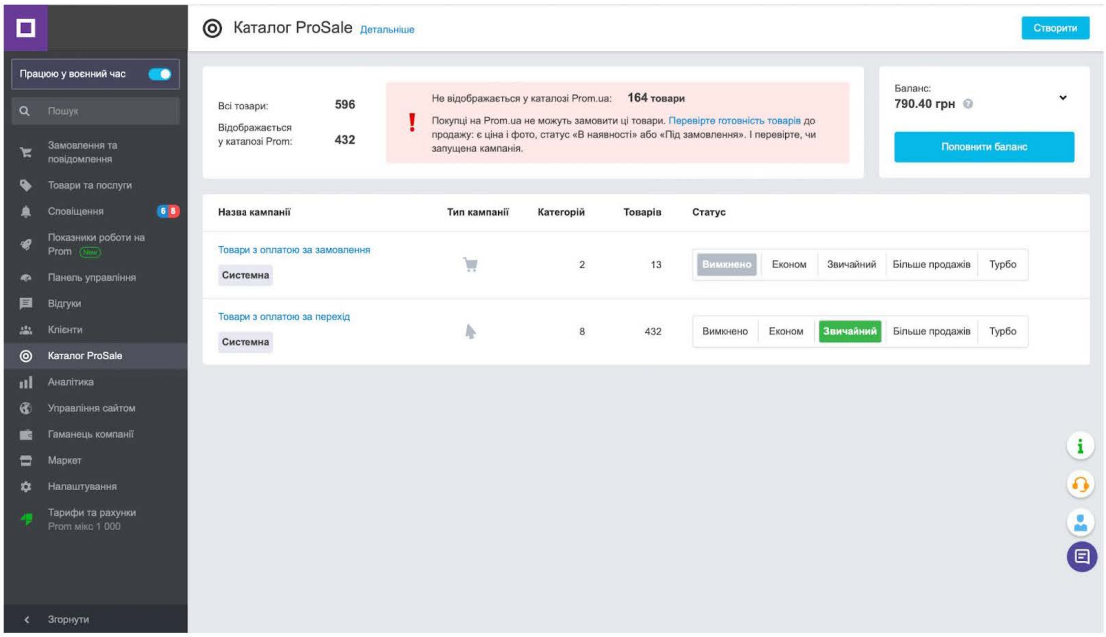

Рисунок 1.4. – Інтерфейс Prom.ua

Окремо слід відзначити, що більшість готових рішень не мають повноцінної підтримки мікросервісної архітектури. Їхній код часто є монолітним, а це ускладнює масштабування окремих функціональних блоків, наприклад, обробку замовлень, оновлення складу чи розсилку повідомлень. Наявність лише API-інтерфейсів [10] не розв'язує повністю проблему гнучкого масштабування та розподілу навантаження. У таких системах будь-яка модифікація або оновлення одного компоненту часто потребує змін у всій структурі додатку, що збільшує ризики виникнення помилок, потребує зупинки системи під час розгортання та ускладнює підтримку. Монолітні рішення також мають обмежені можливості в плані впровадження нових технологій або змін конфігурацій, що є особливо критичним для швидко зростаючих проектів у сфері електронної комерції.

На відміну від перелічених платформ, кастомна розробка серверного застосунку забезпечує повну гнучкість у налаштуванні системи під специфіку бізнесу. Такий підхід дозволяє самостійно визначати структуру бази даних, логіку обробки подій, інтеграцію з зовнішніми API та внутрішніми сервісами, а також організувати надійний і масштабований обмін даними між компонентами

системи. Використання мікросервісної архітектури дозволяє створювати окремі незалежні модулі, кожен з яких може розроблятися, тестуватися, оновлюватися та масштабуватися незалежно від інших. Це спрощує супровід системи, дозволяє оперативно реагувати на зміни в бізнес-логіці та забезпечує стабільність навіть за високого навантаження. Наприклад, при зростанні кількості замовлень можливим є масштабування лише сервісу обробки замовлень без необхідності втручання в інші компоненти, що значно підвищує ефективність використання ресурсів.

Крім того, власна розробка дає змогу глибше контролювати процеси безпеки, впроваджувати власні механізми автентифікації, авторизації та шифрування даних, що особливо актуально для роботи з персональними даними клієнтів та конфіденційною інформацією. Також розробка з нуля дає змогу забезпечити найвищий рівень відповідності специфічним вимогам замовника або внутрішнім політикам організації, що часто неможливо при використанні готових рішень з обмеженою гнучкістю.

Таким чином, аналіз існуючих рішень демонструє, що хоча на ринку доступні численні платформи для організації електронної комерції, жодна з них не пропонує достатньої гнучкості, контрольованості, масштабованості й можливостей кастомізації, які надає власна серверна розробка на основі мікросервісного підходу. Це робить запропоновану архітектуру більш доцільною для реалізації завдань онлайн-магазину енергетичних товарів.

1.3. Постановка завдання дослідження

У сучасних умовах стрімкого розвитку електронної комерції виникає необхідність у створенні високопродуктивних, масштабованих і гнучких інформаційних систем, здатних ефективно обробляти великий обсяг замовлень, забезпечувати інтеграцію з зовнішніми сервісами та відповідати вимогам користувачів щодо швидкості, точності та зручності роботи з платформою.

Особливої актуальності це набуває для спеціалізованих інтернет-магазинів, таких як ті, що займаються продажем енергетичних товарів, де наявність великої кількості категорій, технічних характеристик, індивідуальних запитів клієнтів та складної логіки виконання замовлень потребує продуманої архітектури застосунку.

У сучасних умовах цифрової трансформації бізнесу електронна комерція стала одним із найдинамічніших і найперспективніших секторів економіки. Особливо це стосується галузей, пов'язаних із продажем спеціалізованої продукції, зокрема енергетичних товарів, де клієнти очікують не лише зручного інтерфейсу для здійснення покупок, а й високої швидкості обробки замовлень, точної інформації про наявність товарів, можливості залишати коментарі та оперативно отримувати інформацію про зміну статусу свого замовлення.

Більшість традиційних рішень для організації електронної комерції або надто складні та дорогі для малих і середніх підприємств, або не забезпечують гнучкості, необхідної для адаптації під специфічні бізнес-процеси конкретного магазину. У таких умовах актуальною є розробка власного серверного застосунку, що відповідає сучасним вимогам до стабільності, масштабованості, автоматизації операцій і простоти обслуговування.

Метою дослідження є розробка, впровадження та тестування серверного застосунку для онлайн-магазину, побудованого на основі мікросервісної архітектури, який дозволяє автоматизувати основні бізнес-процеси: обробку замовлень, управління каталогом товарів, обмін повідомленнями між сервісами, та інформування клієнтів про зміну статусу замовлення.

У межах цього дослідження необхідно реалізувати низку взаємопов'язаних завдань, спрямованих на створення ефективного серверного застосунку для системи електронної комерції у сфері енергетичних товарів. Насамперед потрібно провести детальний аналіз предметної області з урахуванням специфіки онлайн-продажу енергетичної продукції та сформувати перелік функціональних вимог до майбутньої системи. Такий аналіз дає змогу врахувати

ключові особливості галузі — зокрема велику кількість категорій товарів, залежність попиту від сезонності, потребу в технічних характеристиках, що впливають на вибір товару, а також важливість своєчасної доставки й підтримки користувачів.

Важливо також обґрунтувати вибір архітектурного підходу, що дозволить реалізувати масштабовану, гнучку й надійну інфраструктуру. У цьому контексті мікросервісна архітектура є найбільш доцільною, адже вона дозволяє розділити систему на незалежні компоненти, кожен з яких відповідає за конкретну бізнес-функцію: управління товарами, обробку замовлень, керування користувачами, обробку коментарів, розсилку повідомлень тощо. Такий підхід забезпечує зручність масштабування, спрощує тестування, підвищує надійність, дозволяє впроваджувати нові функції без шкоди для існуючого функціоналу та значно полегшує обслуговування й підтримку системи.

Подальший етап полягає у проектуванні структури бази даних для зберігання даних про товари, користувачів, замовлення та коментарі. Для цього використовується реляційна система керування базами даних PostgreSQL, яка забезпечує високу продуктивність, підтримує складні запити та транзакції, а також дозволяє реалізовувати складні зв'язки між сутностями. На основі спроєктованої схеми розробляється серверний застосунок, що реалізує REST API для управління основними бізнес-процесами: створення та обробка замовлень, фільтрація товарів за категоріями, керування коментарями тощо. API створює надійний інтерфейс для взаємодії як з користувацьким інтерфейсом, так і з зовнішніми або внутрішніми сервісами.

Для забезпечення асинхронної міжсервісної взаємодії впроваджується механізм обміну повідомленнями за допомогою брокера RabbitMQ — зокрема для автоматичної розсилки електронних листів при зміні статусу замовлення. Це дозволяє досягти високої надійності у взаємодії між сервісами, уникнути блокувань і затримок у роботі системи та покращити користувацький досвід завдяки оперативному інформуванню про зміну статусу покупки. Такий підхід

також відкриває перспективи для масштабування системи в майбутньому — наприклад, шляхом додавання нових сервісів без потреби в модифікації існуючих компонентів.

Одночасно потрібно створити інфраструктуру для розгортання системи за допомогою Docker і Docker Compose, що забезпечить гнучкість, ізоляцію окремих сервісів і простоту налаштування як у локальному, так і у віддаленому середовищі. Завдяки контейнеризації розроблений застосунок легко переносити між середовищами, а процес розгортання значно спрощується, що є критичним для забезпечення безперервної інтеграції та доставки (CI/CD). Кожен мікросервіс розгортається у власному контейнері, що гарантує незалежність виконання та виключає вплив одного сервісу на інший.

Для візуалізації функціоналу розробляється фронтенд-додаток, який забезпечує взаємодію з сервером через HTTP-запити, дозволяючи користувачам переглядати товари, залишати коментарі, оформлювати замовлення та отримувати сповіщення. Важливо забезпечити зручний і інтуїтивно зрозумілий інтерфейс, який відповідає очікуванням сучасних користувачів та сприяє ефективному використанню функціоналу системи. У реалізації фронтенду можуть бути використані сучасні JavaScript-фреймворки, які забезпечують високу продуктивність і зручність розробки.

Завершальним етапом є підготовка повноцінної технічної документації за допомогою OpenAPI (Swagger) та Postman для полегшення тестування, впровадження і подальшої підтримки системи. Документація дозволяє як розробникам, так і стороннім інтеграторам легко ознайомитися з можливостями API, здійснювати перевірку запитів, генерувати клієнтські SDK та забезпечувати подальший розвиток проєкту відповідно до нових бізнес-вимог. Наявність якісної документації — це необхідна умова для ефективного впровадження програмного продукту у виробниче середовище.

Таким чином, у межах дипломної роботи реалізується повний цикл розробки серверного застосунку — від аналізу предметної області та

формування вимог до впровадження, налаштування інфраструктури, реалізації взаємодії сервісів та підготовки супровідної документації. Отриманий результат відповідає сучасним вимогам до інформаційних систем у сфері електронної комерції, зокрема в контексті автоматизації процесів, надійності, масштабованості та якості обслуговування кінцевого користувача.

Об'єктом дослідження в межах цієї роботи є інформаційна система управління замовленнями в електронній комерції, зокрема її серверна частина, яка забезпечує автоматизацію бізнес-процесів, пов'язаних із продажем, обробкою та створенням товарів. Система охоплює управління даними про замовлення, товарний асортимент і комунікацію між окремими сервісами, що є критичними елементами функціонування сучасного онлайн-магазину в галузі енергетичних товарів.

У результаті дослідження очікується створення повноцінного серверного застосунку для онлайн-магазину з енергетичними товарами, який забезпечуватиме обробку замовлень, управління товарним каталогом, фільтрацію продукції за категоріями та обробку коментарів користувачів. Система дозволятиме клієнтам відстежувати статус замовлень у реальному часі, а адміністраторам — ефективно керувати всіма етапами життєвого циклу замовлення. Буде реалізовано зручну інфраструктуру для розгортання за допомогою Docker та Docker Compose, що дасть змогу швидко й безперешкодно запускати систему на будь-якому сумісному сервері.

Окрім цього, буде підготовлено технічну документацію у форматі OpenAPI та колекції запитів у Postman, що спростить тестування, інтеграцію та подальшу підтримку системи. Впровадження API-документації дозволить зацікавленим сторонам — розробникам, тестувальникам, адміністраторам — швидко ознайомитися з можливостями системи, протестувати основні бізнес-функції та інтегрувати сторонні сервіси без додаткових витрат часу на вивчення коду.

Важливою складовою є й забезпечення належного рівня гнучкості та масштабованості. Завдяки мікросервісному підходу окремі компоненти можуть

оновлюватися, масштабуватися або розгортатися незалежно один від одного. Це відкриває широкі можливості для подальшого розвитку системи, наприклад, додавання нових функцій без потреби зупинки всієї платформи або адаптації до змін ринку й бізнес-вимог.

Застосування таких сучасних технологій, як FastAPI для побудови REST API, PostgreSQL для збереження даних, RabbitMQ для обміну повідомленнями та Docker для розгортання, дозволить досягти високого рівня продуктивності, надійності та безпеки. Усі компоненти системи будуть розроблені з урахуванням принципів модульності, повторного використання коду та відкритості до масштабування.

Розробка такого серверного застосунку є актуальним кроком у підвищенні ефективності бізнес-процесів у сфері електронної комерції. Вона сприятиме зменшенню витрат, підвищенню рівня обслуговування клієнтів, забезпеченню стабільності та масштабованості платформи. Запропоноване рішення створює технічне підґрунтя для подальшого зростання онлайн-магазину, адаптації до нових ринків і забезпечення конкурентоспроможності в умовах стрімкого розвитку цифрової торгівлі.

1.4. Висновки до першого розділу

У цьому розділі було проведено ґрунтовний аналіз предметної області електронної комерції в контексті онлайн-продажу енергетичних товарів. Проведений аналіз предметної області показав, що сучасні онлайн-магазини, зокрема в сфері енергетичних товарів, стикаються з численними викликами, пов’язаними з управлінням асортиментом, обробкою замовлень та взаємодією з клієнтами. В умовах стрімкого зростання обсягів електронної комерції та високих очікувань користувачів щодо швидкості й точності обслуговування, впровадження сучасних, масштабованих і гнучких технічних рішень стає критично важливим для збереження конкурентоспроможності бізнесу.

Запропонований серверний застосунок відповідає цим потребам, адже розроблений із використанням мікросервісної архітектури та сучасних технологій, що дозволяє ефективно автоматизувати ключові бізнес-процеси, підвищити продуктивність роботи системи та забезпечити високий рівень обслуговування клієнтів.

Було виявлено, що ринок має багато готових рішень, а саме такі системи як: Shopify, Magento, WooCommerce, Prom.ua, тощо. Проте висока вартість обслуговування, надто загальний набір функцій може не підходити для малих та середніх підприємств. Вибір існуючого рішення може залежати від потреб компанії, розмірів та можливостей у сфері фінансів. Для підприємств, що тільки починають працювати, критично важливо знайти баланс між вартістю, функціональністю, та коштами, що йдуть на підтримку та використання системи. Нерідко бізнесу потрібне індивідуальне рішення, що враховує специфіку продуктів, наприклад — широкий асортимент енергетичного обладнання, потребу в додатковій консультації клієнтів, тощо.

Окрім цього, в ході дослідження були проаналізовані ключові особливості взаємодії користувача з системою та вимоги до функціональності з урахуванням галузевої специфіки. Зокрема, було визначено необхідність реалізації системи коментування товарів, функціоналу запитів консультацій, а також гнучкого управління статусами замовлень із супровідною онлайн комунікацією. Це дає змогу не лише підвищити зручність користування платформою, а й значно поліпшити загальну якість обслуговування, що особливо важливо в умовах конкурентного середовища.

У межах аналізу також розглянуто недоліки монолітної архітектури, характерної для багатьох готових рішень. Така архітектура ускладнює масштабування окремих функціональних модулів. Зі зростанням кількості користувачів і транзакцій такі системи стають менш ефективними, що обмежує їхнє застосування в довгостроковій перспективі.

Розробка власного серверного застосунку відкриває можливості для повної кастомізації функціональності відповідно до потреб бізнесу. Такий підхід дозволяє реалізувати індивідуальну структуру бази даних, гнучку логіку обробки подій, інтеграцію з зовнішніми сервісами.

Таким чином, створення серверного застосунку для онлайн-магазину енергетичних товарів на базі сучасних технологій та мікросервісної архітектури є актуальним кроком до підвищення ефективності бізнесу, конкурентоспроможності на ринку та якості обслуговування клієнтів. Крім того, обрана архітектура дозволяє уникнути проблем, пов'язаних із централізованим зберіганням даних і обробкою великої кількості одночасних запитів, забезпечуючи стабільність роботи навіть при збільшенні навантаження.

На основі проведеного аналізу можна стверджувати, що власне розроблене рішення має значний потенціал не лише як інструмент автоматизації внутрішніх бізнес-процесів, а й як платформа для масштабування діяльності. Завдяки використанню відкритих стандартів, сучасних фреймворків та хмарної інфраструктури, система здатна адаптуватися до змін ринку, підтримувати зростаючу кількість користувачів і забезпечувати високий рівень надійності та безпеки. Таким чином, запропонований підхід не лише відповідає сучасним технологічним вимогам, а й створює фундамент для стійкого розвитку підприємства в цифровому середовищі.

Сформульовано мету дослідження — створення серверного застосунку на основі мікросервісної архітектури, який дозволяє автоматизувати обробку замовлень, управляти товарним асортиментом, забезпечувати ефективну взаємодію між сервісами та підтримувати масштабовану інфраструктуру. У межах дослідження визначено архітектурні та технічні підходи, які забезпечать реалізацію надійного, гнучкого та зручного у розгортанні застосунку. Було обґрунтовано вибір інструментів і технологій, які забезпечують баланс між продуктивністю, простотою у впровадженні та довгостроковою підтримкою.

РОЗДІЛ 2. ПРОЕКТУВАННЯ ДОДАТКУ ДЛЯ ВЕБ-СИСТЕМИ ЕЛЕКТРОННОЇ КОМЕРЦІЇ

2.1. Інструментальні засоби реалізації додатку

У цьому розділі розглядаються інструментальні засоби, що використовуються під час розробки застосунку для онлайн-магазину енергетичних товарів. Правильний вибір технологій та середовища розробки відіграє ключову роль у створенні ефективного, масштабованого та стабільного програмного продукту. Від цього залежить не лише швидкість і зручність процесу розробки, а й функціональність, надійність та підтримуваність системи в майбутньому. Сучасні інструменти мають забезпечувати повний цикл розробки: від написання коду та тестування — до профілювання, налагодження та розгортання мікросервісів у продакшн-середовищі.

Для розробки та тестування додатку використовуються наступні технології: Git, Docker, Docker Compose, Python, Java, Bash, JavaScript, Nginx, Minio та PostgreSQL. Кожен із цих інструментів забезпечує необхідний функціонал та має свої переваги, що сприяють ефективності та зручності розробки.

Розглянемо більш детально кожен з них.

Git [11] — це система контролю версій, яка дає змогу ефективно керувати історією змін у коді. Вона є стандартом у сучасній розробці програмного забезпечення, забезпечуючи зручну роботу в команді, відкат до попередніх версій, ведення гілок і злиття змін. Завдяки Git розробники можуть працювати паралельно, не заважаючи одне одному, а також зберігати надійність і повторюваність процесу розгортання.

Саме Git використовувався для контроля версій додатку. Якщо якась зміна призводила до несправної роботи застосунку, завжди можна було перейти на минулу версію, яка була стабільною та корректную у експлуатації.

Docker [12] — це платформа для контейнеризації, яка дозволяє розробникам упакувати застосунок з усіма залежностями в окремий контейнер. Це гарантує однакову роботу додатку в будь-якому середовищі — від локальної машини до хмарного сервера. Docker спрощує розгортання, масштабування і тестування застосунків. Разом з Docker використовувався Docker Compose [13]. Він, у свою чергу, є інструментом, який дозволяє описати багатокомпонентні Docker-додатки у вигляді конфігураційного yaml файлу, зазвичай це docker-compose.yml або compose.yaml. Це особливо зручно при роботі з мікросервісною архітектурою, оскільки дозволяє одночасно запускати всі необхідні сервіси (сервер, база даних, брокер повідомлень тощо) однією командою.

Сам по собі контейнер у Docker — це ізольоване середовище, в якому виконується застосунок разом із усіма необхідними залежностями, конфігураціями та системними бібліотеками. Його можна уявити як "легку віртуальну машину", але без зайвої надбудови операційної системи, що робить контейнер швидшим та легшим. [14]

Контейнери створюються на основі Docker-образів, які описують, що саме має бути всередині контейнера (код, бібліотеки, середовище тощо). Під час запуску контейнер використовує ресурси ядра хостової ОС (операційна система), але ізольовано від інших процесів.

Python — одна з найпопулярніших мов програмування, яка широко використовується для серверної розробки. Завдяки простому синтаксису, великій кількості бібліотек та активній спільноті Python чудово підходить для розробки REST API, автоматизації, обробки даних і мікросервісів. У контексті цього проекту використовується фреймворк FastAPI для побудови високопродуктивного серверного застосунку.

Java — мова програмування, яка добре зарекомендувала себе у створенні високонавантажених, надійних і масштабованих систем. Вона може використовуватись для окремих сервісів, де потрібна висока продуктивність, або для інтеграції з існуючими Java-компонентами.

На Java також було написано декілько мікросервісів, що взаємодіють між собою.

Bash — мова скриптів для автоматизації в Unix-подібних системах. [15] Використовується для написання скриптів розгортання, CI/CD (Continuous integration / Continuous delivery), управління середовищем, налаштування серверів. Дає змогу автоматизувати повторювані рутинні завдання.

Завдяки Bash було автоматизовано рутині задачі, такі як: розгортання додатку, запуску тестів, збірки Docker-образів, встановлення останніх змін в проекті. Також, мовою Bash встановлювалися змінні середовища, що використовувалися між мікросервісами.

JavaScript — ключова мова для створення інтерактивного фронтенду. Вона використовується у браузері для реалізації динамічної взаємодії з сервером через HTTP-запити (наприклад, у React, Angular, Vue чи інших фреймворках). У цьому проекті JavaScript забезпечує комунікацію з серверним API і візуалізацію інтерфейсу користувача.

Далі, для взаємодії між сервером та фронтендом використовувався Nginx — це високопродуктивний веб-сервер і зворотній проксі, який використовується для маршрутизації запитів, балансування навантаження, кешування та підвищення безпеки. [16] У проекті Nginx використовується для маршрутизації трафіку між фронтендом і сервером, а також для обслуговування статичних ресурсів.

Для автоматизації тестування, використовувався Pytest — фреймворк для тестування в Python, який дозволяє створювати як юніт так і інтеграційні тести. Його використання дозволяє виявити помилки на ранніх етапах і забезпечити стабільність застосунку.

В якості сервісу для збереження даних використовується PostgreSQL — потужна реляційна система управління базами даних з відкритим кодом. Вона підтримує складні запити, транзакції, індекси, збережені процедури та

масштабування. [17] У PostgreSQL зберігається інформації про товари, замовлення і коментарі.

PostgreSQL була обрана як основна реляційна система управління базами даних для реалізації дипломного проєкту через поєднання високої надійності, багатих функціональних можливостей і відкритої ліцензії. На відміну від інших реляційних СКБД (Система Керування Базами Даних), PostgreSQL демонструє повну відповідність стандартам SQL та забезпечує розширену підтримку складних запитів, включаючи підзапити, представлення (views) та матеріалізовані представлення. Це дозволяє реалізовувати бізнес-логіку без значних компромісів у продуктивності чи гнучкості структури даних. Завдяки своїй архітектурі та реалізації механізмів збереження транзакцій (ACID) PostgreSQL гарантує консистентність і цілісність даних навіть у разі одночасного доступу численних сервісів або виходу з ладу окремих компонентів системи.

Для збереження зображень можна було використовувати й PostgreSQL, але це не саме оптимальне рішення для цього, тому в якості сервісу для збереження статики використовується Minio — це високопродуктивне об'єктне сховище з відкритим кодом, яке сумісне з Amazon S3 API. [18] Воно використовується для зберігання великих обсягів неструктурованих даних, таких як зображення, відео, резервні копії, журнали, документи та інші файли. Minio ідеально підходить для локального або приватного використання, коли потрібен контроль над даними без залежності від хмарних провайдерів.

2.2. Архітектурні патерни та підходи програмування

Використання патернів програмування під час розробки серверного додатку для онлайн-магазину з енергетичними товарами має критичне значення для забезпечення масштабованості, підтримуваності та розширюваності системи. Застосування перевірених архітектурних підходів та принципів

дозволяє уникнути хаотичного зростання коду, зменшує ризики помилок, спрощує тестування та полегшує залучення нових розробників до проекту.

Одним із ключових підходів став DDD (Domain-Driven Design) — підхід, що зосереджується на моделюванні бізнес-логіки відповідно до реальних процесів предметної області [19]. У контексті електронної комерції це дозволяє розмежувати логіку, пов'язану з товарами, замовленнями та комментарями, і створити чіткі межі відповідальності між модулями. Завдяки DDD система стає більш гнучкою до змін, які виникають внаслідок еволюції бізнесу. Основна ідея полягає в тому, що структура та мова програмного коду мають відображати реальні бізнес-концепти, що дозволяє створювати максимально відповідне рішення реальним потребам.

У проекті, DDD був застосований для наступного:

a. Чіткого розмежування модулів за бізнес-функціями: управління товарами, оформлення замовлень, обробка коментарів, email-розсилка тощо.

b. Побудови моделі предметної області на основі реальних термінів, що використовуються в електронній комерції та доменній зоні застосунку.

c. Визначення обмежених контекстів для уникнення двозначностей у системі. Кожен компонент системи має свої границі та зону відповідальності.

d. Створення чистої й логічної структури проекту, де бізнес-логіка зосереджена у відповідних агрегатах і сервісах в рамках конкретного піддомену.

Впровадження принципу інверсії залежностей (Dependency Injection) допомогло підвищити модульність та тестованість системи. Замість жорсткої залежності одного компонента від іншого, залежності передаються зовні — це дозволяє легко змінювати, розширювати або ізолювати компоненти при тестуванні.

Також, в усіх компонентах системи код писався притримуючись принципів SOLID. Принципи SOLID — це набір з п'яти ключових принципів об'єктно-орієнтованого програмування, які сприяють написанню чистого, гнучкого, масштабованого та підтримуваного коду. У дипломному проекті ці принципи

були активно застосовані для забезпечення стабільності системи, легкості її супроводу та розширення. Вони виступають основою якісного об'єктно-орієнтованого дизайну: допомагають будувати класи, які мають лише одну відповідальність, відкриті до розширення, але закриті до змін, легко інтегруються з іншими компонентами системи, не створюючи жорстких зв'язків, і поводяться передбачувано у спадкуванні. Це критично важливо у мікросервісній архітектурі, де багато компонентів взаємодіють між собою [20].

Також, слід відзначити активне використання принципу DRY (Don't Repeat Yourself). Цей принцип був послідовно дотриманий при проектуванні сервісів, бізнес-логіки та допоміжних модулів. Замість дублювання коду, функціональність повторного використання реалізовувалася через створення спільних бібліотек, модулів утиліт та абстракцій. Наприклад, загальні механізми валідації, надсилання повідомлень або взаємодії з базою даних були винесені в окремі модулі, що використовуються одночасно кількома сервісами. Це дозволило зменшити обсяг коду, підвищити його читабельність і знизити ризик помилок при внесенні змін.

Деякі компоненти, зокрема модулі взаємодії з брокером повідомлень RabbitMQ, сервіси для роботи з email-нотифікаціями або клієнти для REST API, були реалізовані як окремі бібліотеки в незалежних репозиторіях. Таке рішення дозволило не лише повторно використовувати готові рішення у різних мікросервісах, а й забезпечити їх автономне тестування, версіонування та підтримку. Це особливо важливо в умовах розширення проєкту, коли кількість сервісів збільшується, а вимоги до модульності та підтримуваності стають критичними.

Крім того, в архітектурі проєкту застосовувались шаблони та підходи, притаманні концепції Domain-Driven Design (DDD), яка дозволяє структурувати код згідно з бізнес-логікою предметної області. Завдяки використанню DDD були виділені чіткі межі між доменними моделями, сервісами, контролерами та інфраструктурними компонентами. Це дало змогу зменшити взаємозалежності

між модулями, покращити тестованість системи та забезпечити чітке розділення відповідальностей.

Також було реалізовано принцип інверсії залежностей (Dependency Injection), що дозволило спростити налаштування залежностей між компонентами, зробити код більш гнучким і таким, що легко тестується. Усі залежності конфігурувалися через фабрики або контейнер залежностей, що сприяло зменшенню жорсткого зв'язування компонентів.

Таким чином, застосування програмних принципів і патернів, таких як Domain-Driven Design (DDD), SOLID, Dependency Injection і DRY (Don't Repeat Yourself), стало фундаментом для побудови якісної, масштабованої та підтримуваної архітектури серверного застосунку в межах дипломного проекту. Їх використання дозволило не лише структурувати систему згідно з потребами реального бізнесу, але й зробити розробку прозорою, розширювальною та придатною до змін. Враховуючи динамічний розвиток вимог у сфері електронної комерції, саме така архітектурна та технічна база є запорукою довгострокової ефективності та життєздатності інформаційної системи.

2.3. Проектування графічного інтерфейсу

Проектування графічного інтерфейсу є одним із ключових аспектів розробки будь-якого програмного продукту, зокрема й серверного застосунку для онлайн-магазину. Добре продуманий інтерфейс забезпечує інтуїтивну та зручну взаємодію користувача з системою, що суттєво впливає на загальне враження від сервісу і рівень задоволеності клієнтів. В умовах електронної комерції, де користувачі очікують швидкого і простого доступу до інформації про товари, можливості оформлення замовлень та перегляду статусів, якісний графічний інтерфейс стає критично важливим для утримання клієнтів і підвищення конверсії.

Процес проектування графічного інтерфейсу включає в себе розробку зрозумілої навігації, логічної структури сторінок, а також оптимізацію взаємодії з користувачем — від швидкості завантаження до відгуку на дії. Особливу увагу приділяють простоті, доступності та естетиці інтерфейсу, адже зручний та привабливий дизайн допомагає мінімізувати помилки користувача, скоротити час на виконання операцій і підвищити лояльність до платформи.

У контексті розробки застосунку, графічний інтерфейс користувача реалізується у вигляді окремого фронтенд-додатку. Саме цей компонент є точкою дотику між користувачем і всіма сервісами, які працюють у фоновому режимі, зокрема тими, що відповідають за управління товарами, оформлення замовлень, комунікацію з клієнтом тощо. Фронтенд забезпечує взаємодію з бекендом за допомогою API-запитів, використовуючи протоколи HTTP або HTTPS.

Фронтенд-додаток має критичне значення, оскільки від його якості залежить перше враження користувача про платформу. Інтерфейс повинен бути не лише привабливим, а й інтуїтивно зрозумілим та логічно структурованим. Важливо також забезпечити швидкий відгук системи на дії користувача, візуальний зворотний зв'язок, а також легкий доступ до основних функцій — перегляду товарів, фільтрації за категоріями, додавання до кошика, оформлення замовлення, перегляду історії покупок тощо. Чітка ієрархія елементів, використання стандартних шаблонів UI-поведінки, а також використання кольорів і шрифтів, що відповідають брендингу магазину, також позитивно впливають на загальне враження.

При створенні такого інтерфейсу важливо дотримуватись принципів UX/UI-дизайну, спираючись на сучасні фреймворки (наприклад, React, Vue або Angular), що дозволяють реалізовувати динамічні та інтерактивні вебсторінки з мінімальним часом очікування. У рамках дипломного проекту був використаний Angular, що забезпечило гнучкість у верстці, стильовій уніфікації та зручність у здійсненні запитів до бекенд-сервісів.

Фронтенд-додаток, як частина мікросервісної архітектури, спілкується із сервером через HTTP-запити, тому його коректна робота тісно пов'язана з якісною документацією API. У цьому контексті документація API є невід'ємною частиною сучасної розробки серверних застосунків, особливо в мікросервісних архітектурах. Документація API — це детальний опис усіх доступних методів, форматів запитів і відповідей, правил автентифікації, а також прикладів використання сервісу.

Наявність якісної документації забезпечує прозорість та зрозумілість інтеграції між серверами, фронтендом, а також зовнішніми системами, які можуть використовувати API. У дипломному проекті для цього була використана специфікація OpenAPI (Swagger), яка дозволяє автоматично генерувати документацію з коду, включно з описом моделей даних, валідаційних правил та прикладів запитів і відповідей. Крім того, створено колекції запитів у Postman для ручного тестування, демонстрації можливостей системи та перевірки коректності реалізації кожного з сервісів.

Добре задокументований API дозволяє розробникам швидко орієнтуватися у функціоналі сервісу, спрощує тестування та підтримку, прискорює процес розробки і знижує ризик виникнення помилок при взаємодії між різними компонентами системи. Крім того, документація служить важливим інструментом для забезпечення якості, оскільки вона допомагає стандартизувати способи використання сервісів і забезпечує єдину точку правди для всіх учасників проекту. Це особливо актуально у розподіленій команді розробників або при підключенні сторонніх інтеграторів до системи.

Крім технічного змісту, документація має бути зручною у користуванні. Важливо, щоб вона була оновленою, структурованою та доповненою прикладами запитів, які можна протестувати в інтерактивному режимі. Саме такий підхід дозволяє зменшити поріг входу для нових учасників команди та полегшує підтримку системи в довгостроковій перспективі.

Таким чином, проектування якісного графічного інтерфейсу та створення повноцінної API-документації — це два критично важливі напрямки, які безпосередньо впливають на успішність та ефективність роботи застосунку. Вони підвищують користувацький досвід, забезпечують надійну інтеграцію між усіма компонентами системи, сприяють масштабованості та підтримуваності програмного забезпечення. Їх правильна реалізація є важливою умовою створення конкурентоспроможного цифрового продукту на сучасному ринку електронної комерції.

2.4. Проектування бази даних

Проектування бази даних є одним з ключових етапів розробки будь-якої складної інформаційної системи, особливо в контексті мікросервісної архітектури, яку реалізовано у даному дипломному проекті. Основна мета — забезпечити надійне, структуроване та ізольоване зберігання даних для кожного окремого сервісу, що підвищує безпеку, масштабованість і зручність обслуговування системи.

У нашому випадку використовується реляційна система управління базами даних PostgreSQL, яка відзначається високою продуктивністю, підтримкою складних запитів, транзакцій, індексації, розширень (наприклад, для повнотекстового пошуку) та є відкритим ПЗ з активною спільнотою [21]. PostgreSQL чудово підходить для структурованих даних, де важлива цілісність і зв’язки між об'єктами, що відповідає потребам кожного з мікросервісів.

Архітектурно база даних спроектована відповідно до принципів database per service, що означає: кожен мікросервіс має власну ізольовану базу даних, до якої має доступ лише його власний користувач. Це підвищує безпеку, запобігає випадковому або несанкціонованому доступу між сервісами, та зменшує кількість потенційних точок відмови в системі.

У системі реалізовано чотири основні мікросервіси, кожен з яких має окрему базу даних:

a. Мікросервіс каталогу — відповідає за зберігання інформації про енергетичні товари та їхні категорії. Тут проєктуються таблиці, які описують характеристики товарів, їхню доступність, зв'язки з категоріями, а також, за потреби, додаткові атрибути (наприклад, бренд, ціна, параметри енергоспоживання тощо).

b. Мікросервіс коментарів — містить дані про коментарі користувачів до конкретних товарів. Це дозволяє реалізувати функцію зворотного зв'язку, підвищити довіру до товарів, а також полегшити іншим клієнтам процес прийняття рішень. База містить таблицю з коментарями, їх датами створення, текстом комментарів та данними користувачів, що писали коментар.

c. Мікросервіс замовлень — містить інформацію про замовлення, статусами обробки, контактами клієнтів тощо. Тут важлива підтримка транзакцій та цілісності, особливо при зміні статусів або оплаті.

d. Мікросервіс identity and access — зберігає облікові записи адміністраторів, а також іншу автентифікаційну інформацію. Забезпечення безпеки тут є пріоритетом, оскільки дані мають критичне значення для управління доступом до адміністративної частини платформи.

У кожної бази даних є свій відведений користувач, який має доступ лише до цієї бази даних. Цей користувач використовується для виконання запитів в окремих мікросервісах. Таким чином, навіть якщо дані доступа до однієї бази даних будуть скомпроментовані, зловмисник не буде мати доступ до інших баз даних.

Такий підхід до проектування дозволяє ефективно масштабувати кожен мікросервіс незалежно від інших, спрощує оновлення й розгортання системи, та зменшує ризики виникнення конфліктів при зміні структури даних. Крім того, ізоляція баз даних забезпечує вищий рівень безпеки та стабільності, що особливо

важливо в електронній комерції, де обробляються конфіденційні й критично важливі дані.

2.5. Проектування інфраструктурної частини

Проектування інфраструктурної частини є критично важливим аспектом побудови сучасного серверного застосунку, особливо при використанні мікросервісної архітектури. У межах даного дипломного проекту інфраструктура була побудована з урахуванням принципів ізоляції, автоматизації, масштабованості та простоти розгортання.

Усі компоненти системи, включаючи мікросервіси, бази даних, брокери повідомлень та інші допоміжні сервіси, було контейнеризовано за допомогою Docker. Це дозволило ізолювати кожен сервіс в окремому середовищі, що значно спрощує керування залежностями, усуває конфлікти між компонентами, а також забезпечує стабільність виконання незалежно від середовища розгортання.

Оркестрація контейнерів здійснюється за допомогою Docker Compose, що дало змогу визначити у єдиному YAML-файлі конфігурацію всіх сервісів, їхні взаємозв'язки, змінні середовища, порти, томи для зберігання даних тощо. Це значно полегшує процес запуску всієї системи як у середовищі розробки, так і на продакшн-серверах.

У рамках розробки серверного застосунку з мікросервісною архітектурою важливо забезпечити гнучке та ефективне керування процесом запуску окремих сервісів і всієї системи загалом. Для цього реалізовано підхід перевикористання docker-compose файлів із урахуванням різних режимів запуску.

Кожен компонент системи (мікросервіс або інфраструктурний сервіс) має власний Docker-образ, який збирається окремо, та окремий docker-compose.yml файл, у якому описано, як саме запускати цей компонент із відповідними залежностями (наприклад, базою даних, змінними середовища, портами, томами тощо). Це дозволяє запускати кожен мікросервіс ізольовано, що особливо зручно

для розробки, тестування або налагодження конкретного модуля без потреби запускати всю систему.

Для автоматизації типових операцій, таких як збірка образів, запуск окремих сервісів або всієї системи, перезапуск, перевірка логів та інші дії, використовуються Bash-скрипти. У скриптах реалізовано логіку, яка забезпечує швидкий запуск проєкту в один рядок, що зменшує кількість рутинних операцій при розробці. Це також дозволяє уникати помилок, пов'язаних із ручним введенням команд.

Конфігурація системи реалізована через змінні середовища (environment variables), які дозволяють гнучко керувати налаштуваннями без необхідності змінювати вихідний код. Зокрема, таким способом задаються порти, URL-адреси сервісів, облікові дані для баз даних, секрети доступу, тощо. Це полегшує розгортання в різних середовищах.

Для різних компонентів системи, таких як мікросервісу коментарів, замовлень, авторизації, каталогу та фронтенд додатку були створенні свої зміни середовища, які використовуються тільки в контейнерах цих систем. Таким чином, створенні компоненти системи є легким для конфігурації та розгортанні в різних середовищах з різним набіром вхідних параметрів.

Такий підхід до проєктування інфраструктури забезпечує високу повторюваність розгортання, мінімізує людський фактор, сприяє швидкій інтеграції нових сервісів у систему та дозволяє масштабувати застосунок відповідно до навантаження. В результаті створюється стабільне, гнучке та легке в обслуговуванні середовище для розробки й експлуатації серверного застосунку.

2.6. Висновки до другого розділу

У цьому розділі було розглянуто ключові технічні аспекти, що лежать в основі реалізації застосунку для онлайн-магазину з енергетичними товарами на

базі мікросервісної архітектури. Проведено огляд обраних інструментальних засобів і технологій, зокрема Docker, Docker Compose, Git, PostgreSQL, MinIO, Bash, JavaScript, Python та інших, які забезпечують стабільність, масштабованість і гнучкість системи. Обґрунтовано доцільність використання шаблонів проєктування та архітектурних підходів, таких як DDD, SOLID, DRY та Dependency Injection, що дозволяє досягти високої якості коду та спрощує підтримку системи.

Особливу увагу приділено питанню проектування бази даних — кожен мікросервіс має власну ізольовану БД (база даних) та окремого користувача, що гарантує безпечність і незалежність сервісів. Також було описано підхід до проектування інфраструктури з використанням контейнеризації, bash-скриптів для автоматизації запуску, та перевикористання docker-compose файлів у різних режимах роботи системи. Це дозволяє швидко запускати як окремі модулі, так і всю систему повністю, що є критично важливим під час розробки, тестування та розгортання.

Крім того, наголошено на важливості якісної API-документації для забезпечення зрозумілості та зручності взаємодії з системою, а також розглянуто значення графічного інтерфейсу користувача, реалізованого у вигляді фронтенд-додатку. Інтерфейс дозволяє ефективно взаємодіяти з усіма основними функціональними можливостями системи, зокрема з переглядом товарів, здійсненням замовлень, залишенням коментарів, запитами консультацій, тощо. Реалізація фронтенду як односторінкового застосунку дозволила забезпечити високу швидкодію та інтерактивність, що позитивно впливає на користувацький досвід. У розробці інтерфейсу також було дотримано принципів компонентного підходу, що забезпечує гнучкість та зручність при внесенні змін або доповнень у функціональність.

Також було приділено увагу до інфраструктурної частини додатку. Розробка інфраструктурної частини системи відіграє вирішальну роль у забезпеченні стабільної та масштабованої роботи всієї архітектури

мікросервісного застосунку. Було зазначено, що завдяки використанню Docker і Docker Compose вдалося досягти ізольованого розгортання кожного окремого компонента, що дозволяє легко запускати як окремі сервіси, так і всю систему цілком. Застосування Bash-скриптів для автоматизації основних дій — таких як збірка образів, запуск сервісів та конфігурація через змінні середовища — значно спрощує обслуговування й підтримку системи. Крім того, гнучка структура docker-compose-файлів дає змогу перевикористовувати конфігурації залежно від режиму запуску, забезпечуючи зручність як для розробників, так і для адміністраторів. Усе це робить інфраструктурне рішення ефективним, надійним та придатним до подальшого масштабування і розвитку.

Загалом, реалізація технічної частини проєкту охоплює повний цикл розробки сучасного мікросервісного веб-застосунку — від архітектурного проектування до запуску в ізольованому середовищі. Всі елементи системи були реалізовані відповідно до сучасних стандартів індустрії, що дозволяє не лише ефективно використовувати створений програмний продукт у поточному вигляді, але й без проблем масштабувати та модифікувати його в майбутньому залежно від нових бізнес-вимог.

Базуючись на вимогах, для розробки системи електронної комерції необхідно середовище з встановленою операційною системою Linux, наявністю не менше 8 ГБ оперативної пам’яті, підтримкою Docker, Python, Java, Node.js, PostgreSQL, а також стабільним доступом до мережі для взаємодії між мікросервісами та зовнішніми клієнтами.

РОЗДІЛ 3. РОЗРОБКА ДОДАТКУ ЕЛЕКТРОННОЇ КОМЕРЦІЇ

3.1. Розробка графічного інтерфейсу

Розробка графічного інтерфейсу користувача, а саме frontend додатку, є невід'ємною частиною сучасних веб-застосунків, оскільки саме вона забезпечує зручну та інтуїтивно зрозумілу взаємодію користувача з системою. У даному дипломному проекті для створення фронтенд-додатку використовується Angular — один із найпотужніших і найпоширеніших фреймворків для розробки веб-додатків.

Angular — це фронтенд-фреймворк з відкритим кодом, який розробляється та підтримується компанією Google. Його основними перевагами є:

a. Компонентна архітектура: дозволяє розділити інтерфейс на багато незалежних, багаторазово використовуваних блоків, що полегшує розробку і тестування.

b. Двостороннє зв'язування даних (two-way data binding): зміни в моделі миттєво відображаються в інтерфейсі та навпаки.

c. Масштабованість: Angular добре підходить як для невеликих додатків, так і для корпоративних систем.

Фронтенд-додаток спілкується з трьома незалежними мікросервісами через REST API:

a. Сервіс каталогу — надає доступ до інформації про товари та категорії. Фронтенд додаток витягує ці дані зі сервісу, дозволяє фільтрувати отриманні продукти за критеріями: ціна, категорія та назвою продукту.

b. Сервіс замовлень — фронтенд застосунок спілкується з цим сервісом лише для створення нового товару або запиту консультації в адміністрації сайту.

c. Сервіс коментарів — фронтенд зчитує коментарі для конкретного товару та також надсилає запити на створення нового коментаря.

Цей розподіл дозволяє дотримуватись принципу розділення відповідальностей, покращує масштабованість системи та спрощує супровід кожного окремого сервісу.

Для зберігання зображень товарів використовується MinIO — об'єктне сховище, сумісне з Amazon S3. Воно ідеально підходить для зберігання мультимедійних файлів та їх інтеграції з веб-додатками.

Комунікація між фронтендом та MinIO здійснюється через проксі-сервер Nginx. Фронтенд додаток дістає зображення та відображає їх у браузері в користувача: зображення категорії чи товару.

Таке рішення дозволяє ефективно керувати доступом до ресурсів та підвищує безпеку системи.

Фронтенд-додаток також контейнеризовано за допомогою Docker, що забезпечує:

a. Простий флоу розгортання у будь-якому середовищі;

b. Незалежність від локального оточення розробника;

c. Можливість включити додаток у загальну інфраструктуру мікросервісної системи за допомогою docker-compose.

У Docker контейнері працює невеликий nodejs сервер, що віддає фронтенд додаток, при першому потраплянні на сайт.

Для полегшення розробки та інтеграції з бекендом використовується попередньо створена документація API, згенерована за допомогою OpenAPI та Postman. Це дозволяє:

a. Легко ознайомитись зі структурами запитів та відповідей;

b. Тестувати API до завершення розробки серверної логіки;

c. Швидко інтегрувати нові функції.

Розглянемо детальніше які функції було реалізовано в фронтенд додатку.

Як було вже написано, одна з важливих функцій в інтернет магазині, це форма зворотнього зв'язку. На фото знизу наведена ця форма (рис. 3.1.):

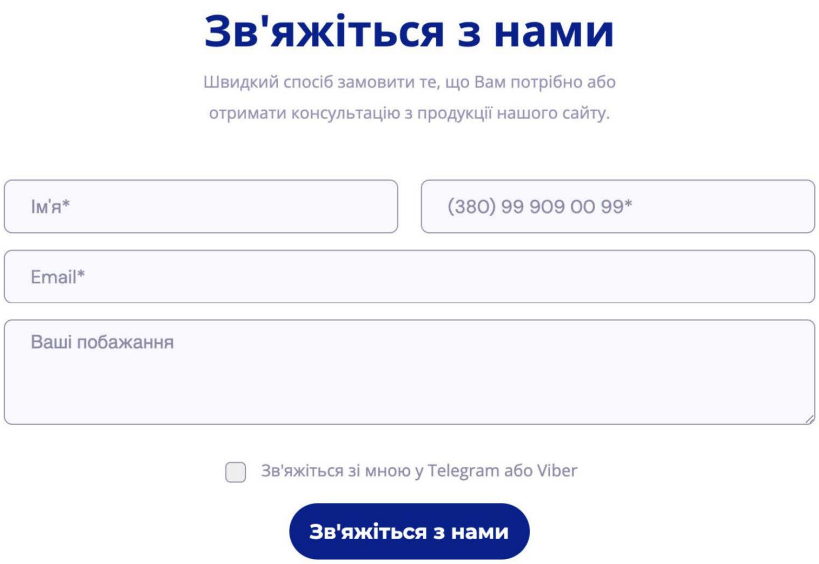

Рисунок 3.1. - Форма зворотнього зв’язку

Далі, на головній сторінці є список усіх категорій (рис. 3.2.). Він є динамічним та розширювальним, бо уся інформація про існуючі категорії надходить з бекенду, а саме мікросервісу каталогу.

Рисунок 3.2. - Список існуючих категорій товару

При переході по категорії гибрідних інверторів, ми потрапляємо на список усіх товарів з цієї категорії (рис. 3.3.):

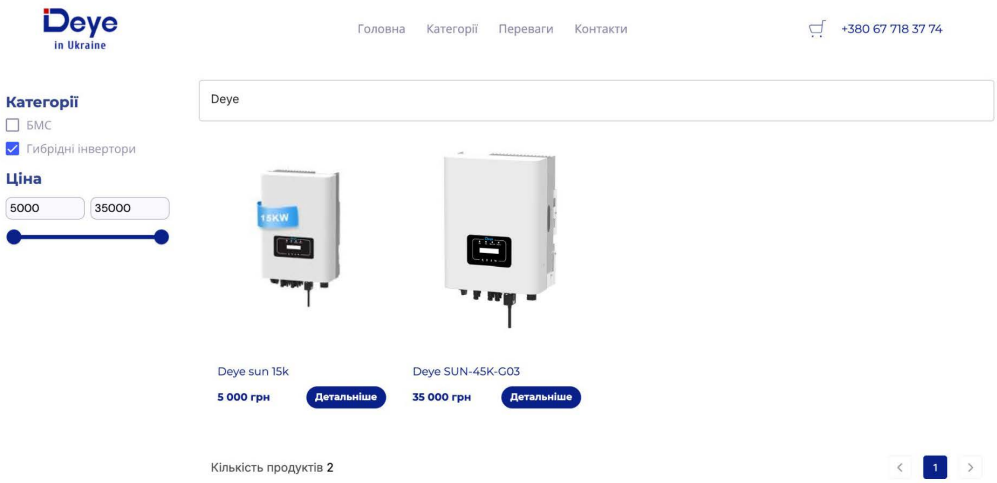

Рисунок 3.3. - Список товарів в категорії

Як бачимо на фото, ми відфільтрували товари по назві Deye. Якщо спробуємо змінити цю назву на Deye SUN-45K-G03, то отримаємо лише один товар (рис. 3.4.).

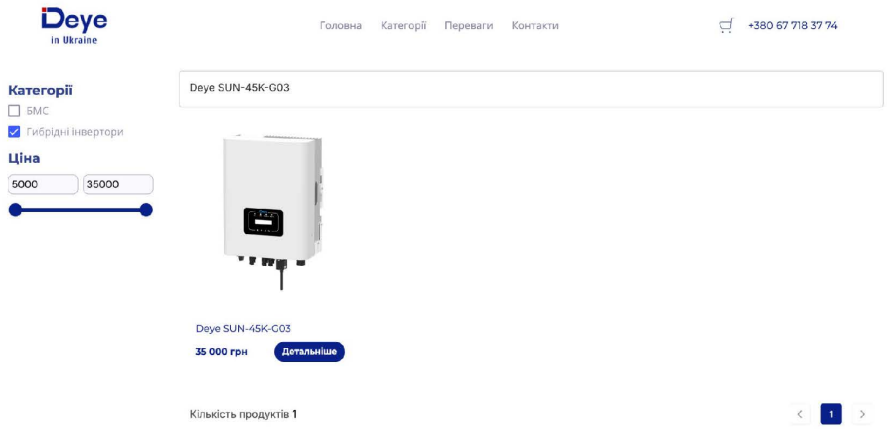

Рисунок 3.4. - Фільтрація за назвою товару

Перейдемо на один з товарів. Нас зустріне сторінка з детальною інформацією по продукту та відгуками до товару (рис. 3.5. та 3.6.).

Рисунок 3.5. - Опис товару

Далі, під описом товару, знаходиться секція коментарів, де користувач може переглядати чужі та залишати свої коментарі:

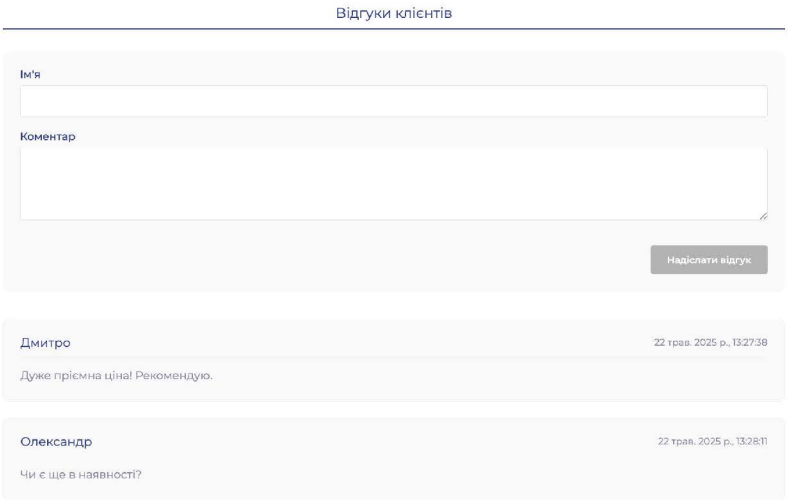

Рисунок 3.6. - Відгуки клієнтів

Також товар в магазині може мати декілька світлин, які можна переглядати (рис. 3.7.):

Рисунок 3.7. - Одна зі світлин товару

Так як це інтернет магазин, ми можемо додавати товару у кошик та замовляти їх. Ось як це виглядає (рис. 3.8.):

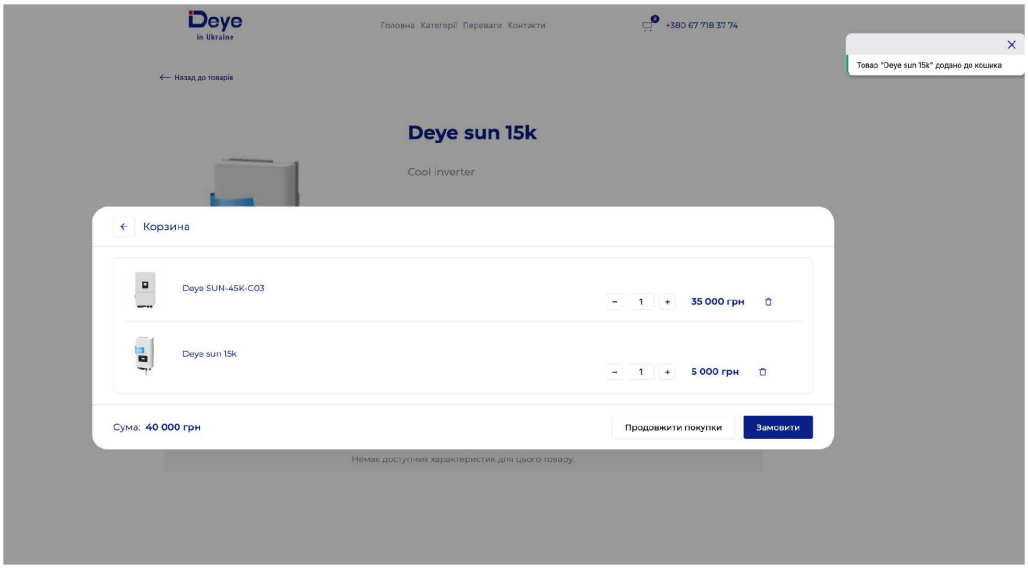

Рисунок 3.8. - Кошик покупця

При натисканні на кнопку замовити, з’являється ще один поп-ап, де ми мусимо ввести свої дані для оформлення замовлення (рис. 3.9.). Ці дані передаються разом з замовленими продуктами на бекенд частину:

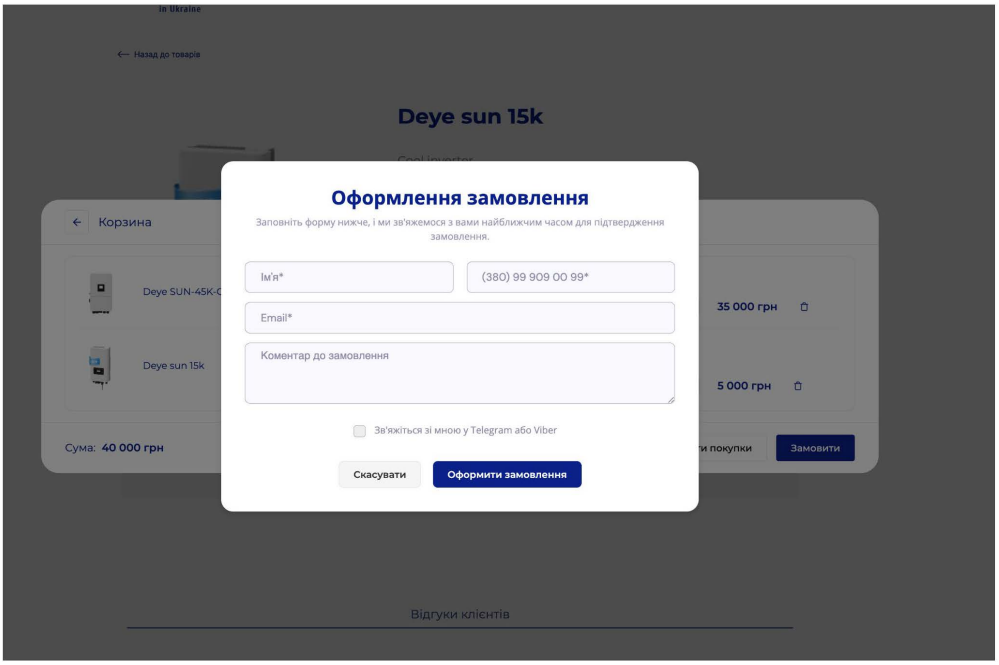

Рисунок 3.9. - Персональні дані покупця при замовленні

Таким чином було продемонстровано основний функціонал фронтенд додатку.

Розробка графічного інтерфейсу з використанням Angular забезпечує гнучкий, продуктивний та масштабований підхід до побудови сучасного веб-додатку. Завдяки інтеграції з REST API мікросервісів, а також використанню MinIO для зберігання зображень та Docker для деплойменту, інтерфейс залишається узгодженим, надійним та готовим до подальшого розширення функціональності.

Задокументовані API гарантують ефективну співпрацю між фронтендом і бекендом, що критично важливо для якісного кінцевого продукту. При розробці фронтенд додатку, це API документація пришвидшувала створення нового функціоналу, який був пов'язаний на взаємодії між фронтенд та бекенд компонентами системи.

3.2. Розробка баз даних мікросервісів

Розробка бази даних є критично важливою складовою створення будь-якої інформаційної системи, особливо у випадку розподілених застосунків із мікросервісною архітектурою. У межах цього дипломного проекту реалізовано підхід, згідно з яким кожен мікросервіс має власну окрему базу даних, що відповідає принципу Database per Service. Це дозволяє забезпечити кращу ізоляцію даних, незалежність мікросервісів та спрощує масштабування й супровід системи

Під час проектування схем БД дотримано принципів нормалізації — це процес організації структур даних таким чином, щоб зменшити надлишковість і забезпечити цілісність інформації. Основні форми нормалізації, які використовуються:

a. Перша нормальна форма (1NF): усі значення в таблиці атомарні, тобто кожне поле містить єдине значення, а не список або масив.

b. Друга нормальна форма (2NF): забезпечує відокремлення атрибутів, що не залежать від первинного ключа в складених ключах.

c. Третя нормальна форма (3NF): унеможливлює транзитивні залежності між неключовими атрибутами.

Застосування цих нормалізацій дозволяє:

a. Зменшити дублювання даних у таблицях;

b. Полегшити оновлення, видалення та додавання інформації;

c. Покращити підтримку цілісності даних;

У системі використовується чотири основні мікросервіси, і кожен з них має власну реляційну базу даних PostgreSQL:

a. Мікросервіс каталогу - містить таблиці для зберігання інформації про товари, їх характеристики та категорії.

b. Мікросервіс коментарів - зберігає коментарі користувачів до товарів.

c. Мікросервіс замовлень - відповідає за зберігання даних про замовлення та статуси обробки.

d. Мікросервіс авторизації (identity & access) - містить інформацію про облікові записи користувачів.

Для кожної БД створено окремого користувача PostgreSQL, який має доступ лише до своєї БД, що є важливим заходом безпеки — у разі компрометації одного з сервісів зловмисник не отримає доступу до інших даних.

Розглянемо більш детальніше кожну базу даних для мікросервісів.

Спочатку, розглянемо схему БД сервіса замовлень (рис. 3.10.):

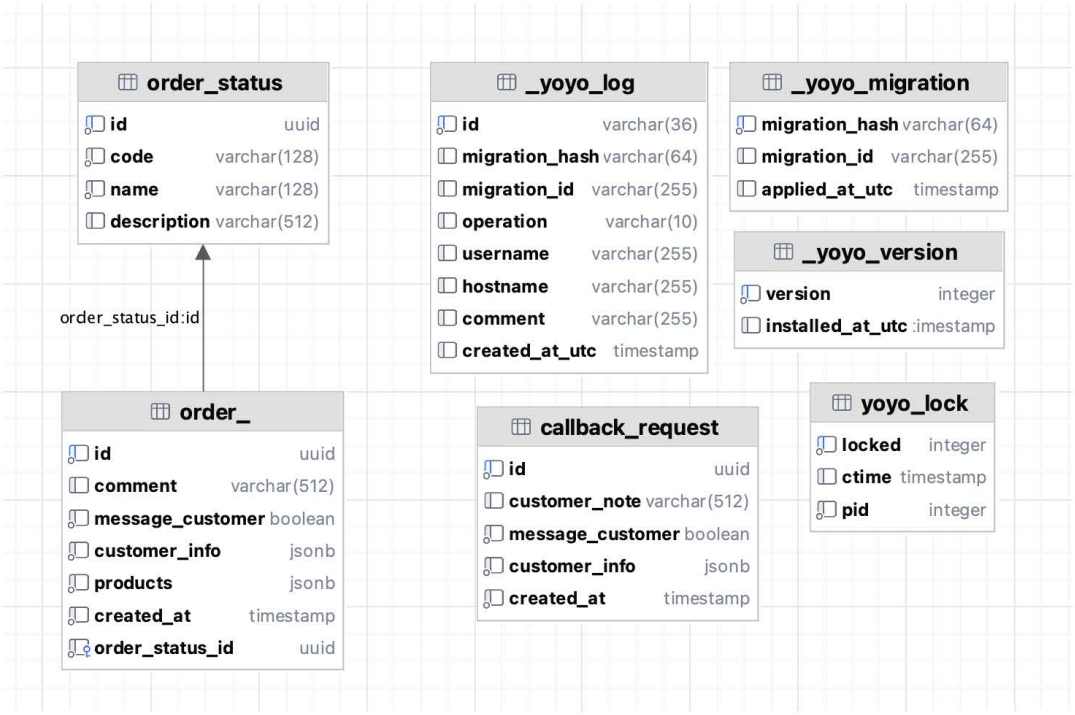

Рисунок 3.10. - Схема бази даних мікросервіса замовлень

Як бачимо сервіс має 7 таблиць:

a. order_status - таблиць існуючих статусів замовлень, наприклад: Done, New, Processing, тощо. На дану таблицю посилається таблиця order_.

b. order_ - таблиця для замовлень. Коли користувач надсилає запит на створення нового замовлення, ми зберігаємо цю інформацію в себе, для того, щоб в майбутньому обробити цю інформацію.

c. callback_request - таблиця для збереження запитів на консультацію. Користувач може запитувати в адміністрації сайту консультацію й ця інформація також має бути збереженою.

d. _yoyo_log, _yoyo_migration, _yoyo_version, yoyo_lock - це службові таблиці, які використовуються утилітою для міграції баз даних yoyo. Дані з цих таблиць ніяк не використовуються для обробки бізнес логіки.

Далі, розглянемо БД мікросервісу каталогу:

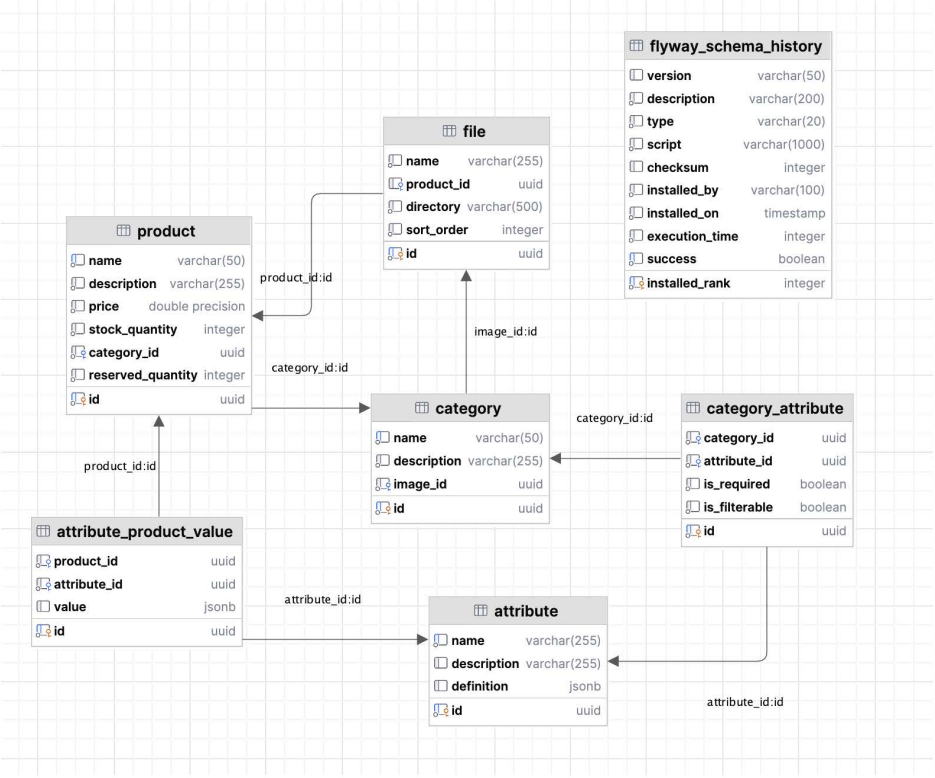

Рисунок 3.11. - Схема бази даних мікросервіса каталогу

Опишемо таблиці:

a. product – таблиця з інформацією про товари. Містить назву, опис, ціну, кількість на складі, зарезервовану кількість та посилання на категорію. Є основною таблицею для представлення товарів у каталозі.

b. category – таблиця з категоріями товарів. Використовується для логічного групування товарів. Має зв’язок із зображенням категорії через поле image_id.

c. attribute – таблиця з атрибутами, які можуть мати товари (наприклад, колір, розмір тощо). Зберігає опис і структуру атрибуту у вигляді JSON.

d. attribute_product_value – таблиця зі значеннями атрибутів для кожного конкретного товару. Зберігає відповідність між товаром і атрибутом, а також значення цього атрибуту (наприклад: "red", "XL").

e. category_attribute – таблиця, яка вказує, які атрибути притаманні кожній категорії. Містить також інформацію, чи є атрибут обов'язковим (is_required) і чи можна його використовувати для фільтрації (is_filterable).

f. file – таблиця для зберігання метаданих про файли (наприклад, зображення). Файли можуть належати товарам або категоріям.

g. flyway_schema_history – службова таблиця, яка використовується Flyway для контролю версій бази даних. Не бере участі в бізнес-логіці, але необхідна для обліку міграцій.

Наступний мікросервіс буде сервіс авторизації або IAC (Identity and Access). Даний мікросервіс має доволі просту схему бази даних. Далі, якщо функціонал буде розширюватися, база даних стане набагато більшою, але на поточний момент, цей сервіс лише веде облік наявних користувачів, а саме адміністраторів сайту. Схема БД на рисунку 3.12.:

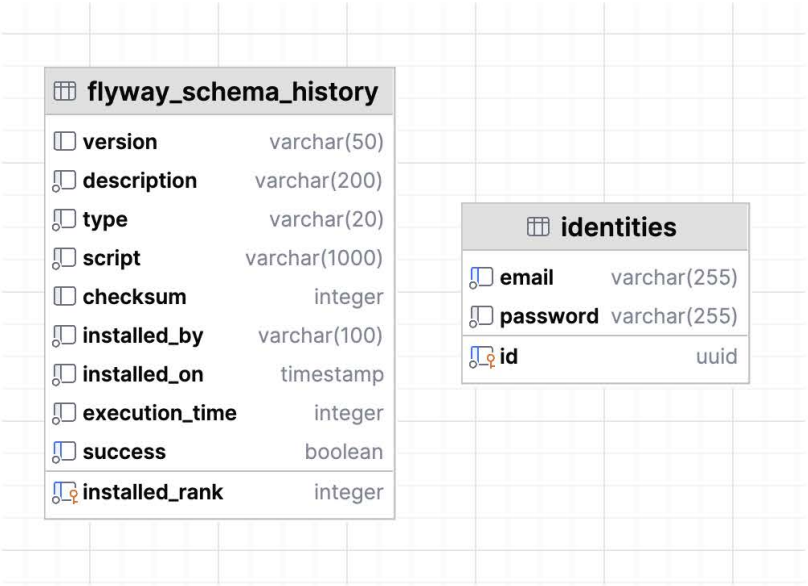

Рисунок 3.12. - Схема бази даних мікросервіса IAC

Опишемо таблиці:

a. flyway_schema_history - службова таблиця, яка використовується Flyway для контролю версій бази даних. Не бере участі в бізнес-логіці, але необхідна для обліку міграцій.

b. identities - таблиця, де зберігаються усі облікові дані адміністраторів сайту.

Останній мікросервіс та його схема БД (рис. 3.13.) - сервіс коментарів. Сам по собі сервіс є невеликим. Його відповідальність лише зберігати та надавати можливість робити CRUD-операції (Create, Read, Update, Delete) над коментарями. Тому схема БД також невелика.

Рисунок 3.12. - Схема бази даних мікросервіса коментарів

a. comment – таблиця для збереження користувацьких коментарів до товарів. Містить автора коментаря, сам текст, дату створення та ключ товару (product_id), до якого належить коментар.

b. alembic_version – службова таблиця, яка використовується системою міграцій Alembic для відстеження поточної версії схеми бази даних. Не використовується у бізнес-логіці.

У процесі проектування бази даних для мікросервісної архітектури онлайн-магазину було реалізовано важливі принципи ізоляції, масштабованості та безпеки. Кожен мікросервіс використовує власну базу даних з окремим користувачем, що дозволяє зменшити ризики перехресного доступу, запобігти проблемам з цілісністю даних і полегшує супровід системи. Такий підхід відповідає патерну Database per Service, який є одним із ключових у побудові мікросервісів та допомагає уникати "вузьких місць", пов'язаних з централізованим сховищем даних.

Було розроблено окремі схеми для мікросервісів каталогу, замовлень, коментарів та авторизації. Схеми відповідають нормальним формам баз даних, що дозволяє уникати надлишковості даних, підвищує цілісність інформації та покращує продуктивність запитів. Для кожної предметної області спроектовано власні таблиці з урахуванням вимог бізнес-логіки: товари, категорії, атрибути, замовлення, статуси, запити на консультацію, коментарі, користувачі тощо. Чітке визначення відповідальності за дані в кожному сервісі дозволяє більш гнучко управляти схемою та оптимізувати індексацію для найпоширеніших сценаріїв використання.

Застосування сучасних інструментів міграцій, таких як Flyway, Yoyo та Alembic, забезпечує контрольоване та відстежуване оновлення структури БД, що є критично важливим у середовищах із безперервною інтеграцією та частими релізами. Автоматизація міграцій зменшує людський фактор і ризики виникнення помилок під час розгортання оновлень, а також дозволяє швидко відкотитися до попередньої версії у разі виявлення проблем.

Активне використання UUID як первинних ключів значно спрощує інтеграцію між мікросервісами, особливо у розподілених системах із паралельною обробкою даних. Завдяки цьому досягається унікальність записів

без необхідності централізованого генератора ідентифікаторів, що особливо важливо при горизонтальному масштабуванні та крос-сервісних запитах.

Для забезпечення безпеки реалізовано розподіл привілеїв на рівні бази даних, мінімізацію доступу сервісів лише до необхідних їм таблиць та операцій, а також шифрування конфіденційної інформації. Крім того, інтеграція з системами контролю доступу та моніторингу дозволяє оперативно виявляти підозрілі активності й захищати систему від несанкціонованого доступу.

У контексті масштабованості, кожен мікросервіс може незалежно вибирати найбільш оптимальний тип бази даних — реляційну, NoSQL або іншу, відповідно до своїх функціональних потреб. Це створює додаткову гнучкість і дозволяє використовувати найкращі інструменти для конкретних завдань, наприклад, кешування, обробки великих обсягів даних або підтримки аналітики.

Таким чином, база даних не лише забезпечує надійний фундамент для всієї системи, а й підтримує високу модульність, гнучкість у розвитку функціоналу, а також відповідає принципам безпечної, масштабованої та стабільної мікросервісної архітектури. Такий підхід сприяє ефективній взаємодії між сервісами, спрощує технічне обслуговування і створює умови для безперебійного росту онлайн-магазину, забезпечуючи його готовність до майбутніх викликів та нових бізнес-вимог.

3.3. Розробка інфраструктурної частини додатку

Інфраструктурна частина онлайн-магазину з енергетичними товарами побудована з урахуванням принципів автоматизації, ізоляції та гнучкості, які є критично важливими для сучасних мікросервісних систем. Весь стек інфраструктури базується на Docker та Docker Compose, що дозволяє забезпечити однакове середовище запуску на будь-якому етапі розробки, тестування та продакшену.

Кожен компонент системи — мікросервіси, база даних, сховище файлів, веб-інтерфейс тощо — упакований у окремий Docker-образ. Це забезпечує:

a. Ізольованість: сервіси працюють незалежно один від одного.

b. Швидкий деплой: запуск і оновлення додатку зводиться до запуску контейнерів.

c. Портативність: однакове середовище на будь-якій машині.

Використовується docker-compose для опису конфігурації всієї системи або окремих її частин. Кожен мікросервіс має власний docker-compose.yml, що дозволяє запускати сервіси окремо для локальної розробки або тестування. Також існує docker-compose файл, який об'єднує сервіси, що використовуються кількома сервісами. Цей сервіс містить брокер повідомлень RabbitMQ, базу даних PostgreSQL, Nginx, Minio, Grafana та Prometheus для моніторингу мікросервісів.

Розглянемо детальніше інфраструктурну частину додатку на прикладі інфраструктуру мікросервіса замовлень та спільної інфраструктури. Це є важливим, бо саме те, як зберігаються ці файли, впливає на те, як треба розробляти скрипти запуску та Docker-образи для їх спільного та коректного використання разом.

На рисунку 3.13. можна побачити структуру файлів в цій частині системи:

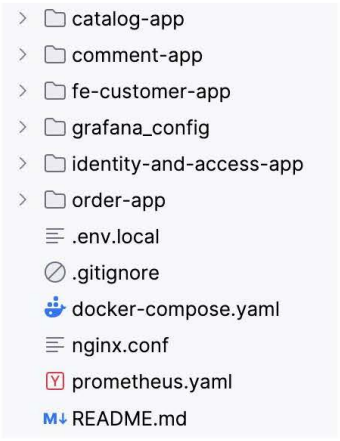

Рисунок 3.13. - Файлова структура інфраструктуру проекту

Для кожного мікросервіса чи частини проекту заведена своя директорії, де зберігаються конфігураційні файлі мікросервіса. Як вже було сказано, розглянемо інфраструктуру мікросервіса замовлень. Перейдемо в order-app діректорію (рис. 3.14.):

Рисунок 3.14. - Файлова структура інфраструктуру проекту

Ми маємо декілька файлів:

a. .env.local - файл зі змінними середовища та скриптами для локальної розробки.

b. .env.local.test - файл зі змінними середовища для локального тестування додатку з автоматизованими тестами.

c. docker-compose.test.yaml - файл для розгортання інфраструктури для запуску тестів.

d. docker-compose.yaml - файл для розгортання інфраструктури в локальному середовищі.

e. Dockerfile - файл для створення Docker-образу мікросервісу замовлень. Використовується багатоетапна збірка для того, щоб оптимізувати фінальній образ.

Багатоетапна збірка — це підхід у Docker, який дозволяє створювати оптимізовані, малі за розміром образи, використовуючи кілька послідовних етапів у одному Dockerfile. Кожен етап має власне ізольоване середовище, і лише потрібні артефакти передаються з одного етапу в інший.

Цей підхід використовується під час створення Docker-образу для сервісу замовлень нашого застосунку.

Розглянемо як виглядає цей образ (рис. 3.15. та 3.16.):

```
ARG GROUP_ID
ARG USER_ID
ARG INSTALL_LOCAL_SHARED_KERNEL


FROM python:3.13 AS dependencies
COPY --from=order-app requirements/requirements.web.txt requirements/requirements.queue.txt .
RUN pip install --no-cache-dir -r requirements.web.txt --prefix=web-service-venv && \
    pip install --no-cache-dir -r requirements.queue.txt --prefix=queue-consumer-venv


FROM python:3.13-slim AS base-app
ARG GROUP_ID
ARG USER_ID
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1
ENV PYTHONPATH="/app"
WORKDIR /app
RUN groupadd -g $GROUP_ID appgroup && \
    useradd -u $USER_ID -g $GROUP_ID appuser && \
    chown appuser /app


FROM base-app AS backend-app
ARG INSTALL_LOCAL_SHARED_KERNEL
COPY --from=dependencies /web-service-venv /usr/local
COPY --from=order-app ./src .
COPY --from=shared-kernel . /shared_kernel/
RUN ($INSTALL_LOCAL_SHARED_KERNEL && pip install -e /shared_kernel/) || true
USER appuser
```

Рисунок 3.15. - Образ сервісу замовлень

Наступним кроком є створення вже фінальних образів, які будуть безпосередньо використовуватися при роботі застосунку:

```
FROM base-app AS queue-consumer
ARG INSTALL_LOCAL_SHARED_KERNEL
COPY --from=dependencies /queue-consumer-venv /usr/local
COPY --from=order-app ./src .
COPY --from=shared-kernel . /shared_kernel/
RUN ($INSTALL_LOCAL_SHARED_KERNEL && pip install -e /shared_kernel/) || true
USER appuser


FROM base-app AS migration-service
ENV PATH="/app/scripts:${PATH}"
COPY --from=order-app requirements/requirements.migration.txt .
RUN mkdir ./scripts && \
    echo '#!/bin/bash\nyoyo apply -d postgres://${DATABASE_USERNAME}:${DATABASE_PASSWORD}@${DATABASE_HOST}:${DATABASE_PORT}/${DATABASE_DATABASE}' > /app/scripts/migra
    echo '#!/bin/bash\nyoyo rollback -d postgres://${DATABASE_USERNAME}:${DATABASE_PASSWORD}@${DATABASE_HOST}:${DATABASE_PORT}/${DATABASE_DATABASE}' > /app/scripts/mi
    chmod +x ./scripts/migration-run ./scripts/migration-rollback && \
    pip install --no-cache-dir -r requirements.migration.txt && \
    rm requirements.migration.txt
WORKDIR /app/migrations
COPY --from=order-app ./src/infrastructure/database/relational/migrations/ .
USER appuser
```

Рисунок 3.16. - Образ сервісу замовлень

На перших двох етапах ми завантажуємо та встановлюємо усі необхідні залежності та додаємо користувача з під якого буде працювати додаток.

На останніх трьох етапах ми створюємо образи, які використовуються при роботі додатку, а саме:

a. backend-app - бекенд додаток, який обробляє HTTP повідомлення

b. queue-consumer - додаток, що обробляє повідомлення, які надходять з брокера повідомлень.

c. migration-service - сервіс, який запускає файли міграції при запуску мікросервіса.

Далі, розглянемо як запускається сам додаток завдяки docker compose. В додатку А наведений compose файл для запуску мікросервіса замовлень.

Базова конфігурація представлена через шаблонні блоки YAML (& та <<), що дозволяє уникнути дублювання і забезпечує централізоване управління змінними середовища та параметрами збірки. Для кожного з етапів у Dockerfile визначено окремі цілі збірки (target), такі як backend-app, queue-consumer, migration-service, що дозволяє чітко розділити процеси обробки HTTP-запитів, повідомлень з черги та запуску міграцій бази даних.

Сервіс order-backend-app відповідає за запуск бекенд додатку. Він забезпечує взаємодію з клієнтською частиною додатку та іншими мікросервісами через REST API. Параметри конфігурації передаються через змінні середовища, які задаються в .env.local файлі. Для зручності локальної розробки використовується режим автоматичної перезагрузки при зміні коду (--reload).

Сервіс order-queue-consumer запускає споживача повідомлень із черги RabbitMQ, реалізованого на базі бібліотеки FastStream. Цей компонент обробляє фонові задачі, пов'язані із замовленнями. Він використовує окремий канал у системі обміну повідомленнями та також залежить від успішного запуску міграцій.

Сервіс order-migration-service забезпечує виконання міграцій бази даних до запуску інших компонентів. У конфігурації визначено окремий об'єм з міграціями, що дозволяє запускати команди управління схемою бази даних незалежно.

Для локального тестування поштових повідомлень використовується контейнер smtp-server на базі Mailpit. Він забезпечує тестовий SMTP-сервер з веб-інтерфейсом, через який можна переглядати надіслані листи.

Усі сервіси функціонують у межах зовнішньої мережі deye-web-network, яка об'єднує мікросервіси в загальне середовище, забезпечуючи міжсервісну взаємодію.

Останнім для цього мікросервіса треба розглянути файл зі змінними середовища та скриптами, які знаходяться в .env.local. Лістинг коду знаходиться в додатку Б.

Файл зі змінними середовища та Bash-скриптами для мікросервісу замовлень відіграє ключову роль у забезпеченні гнучкої конфігурації та автоматизації запуску, зупинки й обслуговування мікросервісу в процесі локальної розробки та розгортання. Він поділений на кілька логічних блоків, кожен з яких відповідає за окремий аспект налаштування або автоматизації.

У блоці Infrastructure Config визначаються порти, які мають бути відкриті у Docker для доступу до HTTP-сервісу замовлень (DOCKER_ORDER_APPLICATION_EXPOSE_PORT) та SMTP-сервера (DOCKER_SMTP_SERVER_EXPOSE_PORT). Також задаються UID і GID для створення образів у системах із різною конфігурацією користувачів, що дозволяє уникнути проблем з доступом до файлів при мапінгу об'ємів у Docker.

Також файл описує основні параметри запуску додатку: порт, в якому працює HTTP-сервер (ORDER_APPLICATION_PORT), назву, версію та режим налагодження додатку (APPLICATION_DEBUG, APPLICATION_VERSION, APPLICATION_TITLE). Окремо задається конфігурація бази даних: хост, порт, ім'я БД, ім'я користувача та пароль.

Для взаємодії з RabbitMQ задаються відповідні змінні: ім'я користувача, пароль, хост, порт, ім'я черги для замовлень (RABBITMQ_ORDER_QUEUE) та конфігурація черги резервування товарів у мікросервісі каталозі.

Також включено параметри для роботи з JWT (JWT_PUBLIC_KEY, JWT_ALGORITHM), що використовуються для перевірки авторизації запитів, та параметри сервісів авторизації і каталогу (IAC_HOST, CATALOG_SERVICE_HOST), а також SMTP-сервера для надсилання email-повідомлень.

Файл містить набір функцій для автоматизованої роботи з інфраструктурою та мікросервісом:

a. order-up — повністю запускає інфраструктуру та мікросервіс замовлень.

b. order-down — зупиняє усі запущені сервіси, пов'язані з замовленнями.

c. app-build — виконує збірку Docker-образу мікросервісу замовлень та очищає проміжні образи.

d. app-up — виконує збірку та запуск мікросервісу, а також чекає завершення міграцій.

e. app-down — зупиняє лише мікросервіс, без впливу на інфраструктуру.

f. migration-run — запускає скрипт для застосування міграцій бази даних.

g. migration-rollback — виконує відкочування останньої міграції.

Функції реалізовані у вигляді окремих підпроцесів та використовують змінну _FUNCTIONS_LOCATION для визначення абсолютного шляху до директорії, з якої були викликані, що дозволяє уникнути проблем з навігацією при виклику скриптів з інших місць.

Таким чином реалізована усі інші мікросервіси.

Для загальної інфраструктури використовується також compose файл, що знаходиться в корені директорії. Лістинг коду цього compose файлу знаходиться в додатку В.

Цей Docker Compose файл забезпечує запуск основної інфраструктури, яка підтримує роботу мікросервісного вебзастосунку. Серед основних компонентів – брокер повідомлень RabbitMQ, база даних PostgreSQL, зворотний проксі-сервер Nginx, об'єктне сховище MinIO, система збору метрик Prometheus, система моніторингу Grafana, а також службовий контейнер `resources-setup`, що відповідає за ініціалізацію ключових ресурсів.

Контейнер RabbitMQ використовує образ з підтримкою веб-інтерфейсу керування (rabbitmq:4.0.5-management). Він відкриває два порти: для менеджмент-консолі (15672) і для обміну повідомленнями за протоколом AMQP (5672). У середовищі задаються облікові дані та параметри логування. Контейнер зберігає свої дані у томі rabbitmq_data і має вбудований механізм перевірки доступності сервісу.

PostgreSQL виконує роль централізованої системи зберігання даних. Для запуску використовується останній офіційний образ. Контейнер приймає облікові дані з середовища і зберігає дані в окремому томі database_data. Здоров'я сервісу перевіряється за допомогою команди pg_isready.

Nginx виконує роль проксі-сервера, який спрямовує запити до інших сервісів, зокрема MinIO. Конфігурація монтується з локального файлу nginx.conf, а сам контейнер стартує лише після того, як MinIO буде визнано "здоровим". Веб-інтерфейс Nginx доступний на порту 9999.

MinIO реалізує об'єктне сховище, сумісне з S3, та запускається з консоллю керування. Контейнер слухає два порти: для основного API (9000) та для панелі адміністратора (9001). Доступ до сервісу захищений за допомогою пари ключ/секрет, а всі дані зберігаються у томі minio_data. Контейнер має healthcheck, який перевіряє доступність HTTP-інтерфейсу.

Grafana забезпечує візуалізацію метрик із сервісів. Вона запускається з доступом на порт 3000 і автоматично конфігурується через локальні файли, що описують дашборди та джерела даних. Облікові дані адміністратора передаються через змінні середовища.

Prometheus збирає метрики з інфраструктури і працює на порту 9090. Він завантажує конфігурацію зі змонтованого локального файлу `prometheus.yml` та запускається з параметром, що вказує шлях до цієї конфігурації.

Окремий службовий контейнер resources-setup виконує скрипт на базі Alpine Linux, який налаштовує MinIO, RabbitMQ і PostgreSQL. У цьому скрипті створюється бакет у MinIO, конфігуруються черги в RabbitMQ, включно з чергами для обробки помилок (dead-letter), та здійснюється створення баз даних і користувачів у PostgreSQL для кожного з мікросервісів (каталог, коментарі, ідентифікація та замовлення). Цей контейнер виконується лише після успішного запуску RabbitMQ, бази даних і MinIO.

Усі сервіси працюють у межах зовнішньої Docker-мережі deye-web-network, що дозволяє забезпечити міжконтейнерну комунікацію. Для RabbitMQ, PostgreSQL та MinIO використано окремі іменовані томи (rabbitmq_data, database_data, minio_data), що гарантує збереження даних між перезапусками. Така конфігурація дозволяє швидко розгорнути повноцінну інфраструктуру застосунку як у середовищі розробки, так і для тестування або CI.

3.4. Розробка мікросервісів

Розробка бекенд-частини додатку реалізована з використанням мікросервісної архітектури, що передбачає поділ системи на незалежні компоненти, кожен з яких виконує окрему бізнес-функцію. Загалом у системі присутні чотири мікросервіси: сервіс замовлень (Order Service), сервіс коментарів (Comment Service), сервіс каталогу товарів (Catalog Service) та сервіс ідентифікації та доступу (Identity and Access Service).

Мікросервіси замовлень і коментарів реалізовані мовою програмування Python з використанням сучасного асинхронного фреймворку FastAPI. FastAPI дозволяє будувати високопродуктивні REST API з підтримкою валідації вхідних даних на основі моделей Pydantic, а також забезпечує автоматичну генерацію OpenAPI-документації. Ці сервіси використовують PostgreSQL як основну СУБД і RabbitMQ для обміну повідомленнями. У них реалізовано черги обробки подій, зокрема, підтвердження або відхилення замовлень, а також асинхронну обробку коментарів із використанням dead-letter queues для підвищення надійності. Для доступу до бази даних застосовано SQLAlchemy з підтримкою останньої версії API (ORM 2.0), що дозволяє описувати моделі у вигляді класів, а також використовувати декларативний підхід до зв'язків між таблицями та сирі запити до БД.

Інші два сервіси — каталог товарів і сервіс автентифікації — реалізовано на Java із застосуванням фреймворку Spring Boot. У каталозі товарів реалізовано логіку створення, редагування та отримання інформації про товари. Сервіс ідентифікації відповідає за реєстрацію користувачів, автентифікацію за допомогою JWT-токенів та авторизацію. Spring Security використовується для контролю доступу до ресурсів, а також реалізації механізмів автентифікації.

Міжсервісна комунікація відбувається як синхронно (через HTTP API), так і асинхронно (через RabbitMQ). Обмін повідомленнями дозволяє розвантажити сервіси, підвищити відмовостійкість системи та забезпечити масштабованість. Сервіси замовлень та каталогу логують ключові події, а також експортують метрики до Prometheus. Метрики доступні для візуалізації в Grafana.

Дані кожного мікросервісу зберігаються у власній базі PostgreSQL. Такий підхід відповідає принципу "розділеної бази даних" (database per service), що покращує ізольованість сервісів та спрощує масштабування. Початкове створення баз даних і користувачів здійснюється автоматизованим скриптом у контейнері `resources-setup`, що виконується під час ініціалізації інфраструктури.

У процесі розробки бекенд-частини нашого застосунку ми керувалися підходом Domain-Driven Design (DDD), що став основою для моделювання бізнес-логіки та визначення кордонів мікросервісів. Поділ на чотири окремі сервіси — Order, Comment, Catalog та Identity and Access — був здійснений саме згідно з концепцією bounded contexts, що є центральним поняттям у DDD.

DDD передбачає, що складна бізнес-система має бути розділена на кілька предметних областей (domains), кожна з яких має свою термінологію, правила та логіку. У нашому випадку кожен мікросервіс представляє окремий bounded context, який ізольовано реалізує свою предметну область і взаємодіє з іншими через чітко визначені контракти.

Order Service реалізує предметну область обробки замовлень — створення, підтвердження, скасування. Ця логіка відокремлена від будь-якої інформації про користувачів або товари, що дозволяє сервісу бути зосередженим виключно на процесі замовлення.

Також даний сервіс передбачає запит на консультацію в працівників сайту. Усі інші мікросервіси слухають на евенти які йдуть з цього сервісу.

Comment Service відповідає за створення та видалення коментарів до товарів. Коментарі — це окрема модель взаємодії користувачів з продуктом, і вона має свою поведінку, правила та цикли життя. Важливо, що цей сервіс не має прямого доступу до сервісу товарів — натомість використовується ідентифікатор товару як частина контексту.

Catalog Service реалізує управління товарами — створення, оновлення, структурування за категоріями, а також резервування товару для замовлень. Саме в межах цього bounded contexty формується уявлення про продукт, його характеристики та наявність.

Identity and Access Service — це окремий контекст, який несе відповідальність за автентифікацію, аутентифікацію користувачів та видачу JWT-токенів. У DDD це називається supporting domain яка хоча й не є

безпосередньо бізнес-критичною, але є необхідною для узгодженої взаємодії в системі.

Усі сервіси спілкуються між собою через асинхронні повідомлення або REST API, не порушуючи межі контекстів. Це дозволяє кожному сервісу бути незалежною мовною та технічною одиницею: деякі з них реалізовані на Python (де це було доцільніше через гнучкість у розробці), інші — на Java (де важлива була сталість, стабільність і безпека на рівні фреймворку).

DDD також дозволив нам чітко структурувати код всередині кожного сервісу. Відповідно до патернів DDD, бізнес-логіка реалізована в доменних моделях, які не змішані з інфраструктурним кодом або API-адаптерами. Наприклад, сервіси коментарів і замовлень у Python використовують окремі модулі для доменних сутностей, сервісів, репозиторіїв і use-caseів. Це не лише полегшує підтримку та розвиток системи, але й робить її масштабованою у довгостроковій перспективі.

Таким чином, використання DDD дозволило нам:

a. визначити чіткі границі відповідальності між сервісами;

b. ізолювати моделі й термінологію кожного домену;

c. забезпечити узгодженість термінів між командою розробки й замовником;

d. реалізувати незалежне масштабування та розгортання сервісів.

Це стратегічне рішення значно підвищило керованість розробкою, зрозумілість архітектури та якість коду, що є особливо критично в умовах зростаючої складності бізнесу.

Розглянемо на діаграмах, які взаємозв'язки між мікросервісами у контексті DDD (рис. 3.17.):

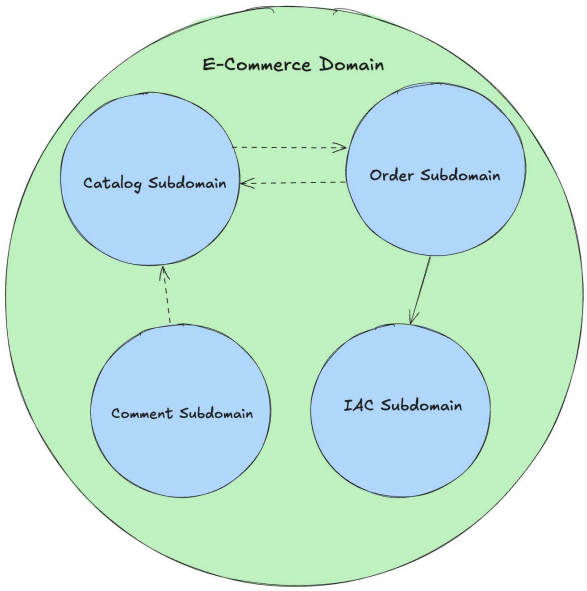

Рисунок 3.17. - Context Map домену

Неявна лінія позначає, що субдомени зацікавленні в інтеграційних подіях, які відбуваються в тому чи іншому сервісі. Наприклад, ми бачимо лінію, що пов'язує сервіс коментарів та каталогу. Це значить, що сервіс коментарів зацікавлений в тій чи іншій події, що відбувається в контексті сервісу каталогу. У випадку цих двох сервісів, це подія видалення продукта. Коли видаляється продукт, то й коментарі, що були створенні до нього, також мають бути видаленні.

Явна лінія, це коли один субдомен напряму звертається до іншого субдомену, щоб дістати якусь інформацію. Наприклад, сервіс замовлень та сервіс авторизації.

Для того, щоб створити замовлення, сервісу замовлень необхідно витягнути електронну пошту працівників магазину, щоб надіслати їм повідомлення. Вся ця інформація зберігається в сервісу IAC. Тому, сервіс замовлень робить синхронний HTTP запит на отримання цієї інформації.

Розглянемо субдомен замовлень, та як ця логіка DDD перенеса в код.

Спочатку, треба сформувати та описати домені сущності, що знаходяться в субдомені замовлень (рис. 3.18.):

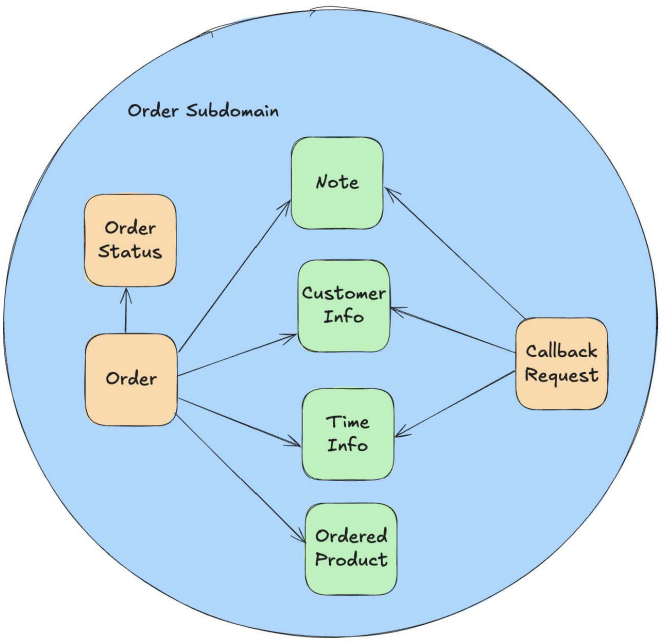

Рисунок 3.18. - Опис субдомену замовлень

У межах субдомену замовлень, зображеного на діаграмі, реалізовано поділ доменної моделі на сутності (entities) та об'єкти значення (value objects) відповідно до принципів Domain-Driven Design (DDD). Такий поділ сприяє точному моделюванню предметної області та забезпечує чітке розмежування відповідальності у межах бізнес-логіки.

Сутності в цьому контексті представляють об’єкти з унікальною ідентичністю, життєвий цикл яких відстежується протягом усієї взаємодії із системою. Основною сутністю субдомену є Order (Замовлення), яка містить унікальний ідентифікатор, інформацію про клієнта, перелік товарів, поточний статус, дату створення, а також часові мітки змін

Об'єкти значення використовуються для опису елементів, які не мають власної ідентичності, але відіграють важливу роль у бізнес-логіці.

Кожна сутність у межах субдомену містить тільки ті методи, які стосуються її поведінки, відповідно до принципу єдиної відповідальності. Це забезпечує високий рівень ізоляції логіки, дозволяє уникати дублікації коду та

спрощує тестування. Для керування життєвим циклом замовлень у системі передбачено використання агрегату (aggregate root), яким виступає сутність Order. Вона відповідає за цілісність всієї внутрішньої структури замовлення, забезпечуючи, щоб усі інваріанти залишалися валідними після будь-яких змін.

Окремо варто відзначити використання доменних подій (domain events), які сигналізують про важливі зміни стану замовлення. Це дозволяє побудувати слабко зв'язану архітектуру, у якій зміни в одній частині системи можуть спричинити реакцію інших компонентів без прямої залежності між ними — наприклад, надсилання електронного листа, оновлення складу або ініціація оплати. Такі події є ключовим механізмом у побудові реактивних мікросервісів.

Таким чином, застосування концепцій DDD у субдомені замовлень дозволило створити модель, яка відповідає реальним бізнес-процесам, є гнучкою до змін і здатною до масштабування. Завдяки чіткому поділу між сутностями та об'єктами значення, виділенню агрегатів та обробці доменних подій система залишається прозорою, передбачуваною та легкою в супроводі.

Розглянемо більш детальніше ці сутності на рисунку 3.18.

Сутності на діаграмі позначені помаранчевим кольором. Вони мають сталу ідентичність протягом усього життєвого циклу та можуть змінювати свій стан. Головною сутністю є Order, яка інкапсулює всю бізнес-логіку, пов'язану із замовленням. Вона пов'язана з іншими об'єктами в моделі, зокрема містить посилання на статус замовлення, інформацію про клієнта, час, замовлені товари та нотатки. Order є агрегатним коренем для цієї частини доменної моделі, що означає централізоване управління узгодженістю даних усередині агрегату.

Сутність Order Status відображає поточний стан замовлення (наприклад, нове, опрацьовується, виконане тощо) та є частиною агрегату Order. Callback Request також є сутністю, яка моделює запит клієнта на зворотний дзвінок, і має власну ідентичність, оскільки може існувати окремо.

Зеленим кольором на діаграмі позначено об’єкти значення. Ці об’єкти не мають власної ідентичності, ідентифікуються повністю за своїми атрибутами та є незмінними. До них належать Note, Customer Info, Time Info та Ordered Product.

Note містить текстові примітки до замовлення або запиту. Customer Info представляє контактні та персональні дані клієнта. Time Info містить інформацію про час створення замовлення. Ordered Product описує характеристики конкретного товару в замовленні.

На рисунку 3.19. можна побачити, як завдяки Python-класам, їх атрибутам та методам можна перенести ці домені сутності в код:

```
@dataclass(kw_only=True, slots=True)  Kirill Panchenko
class Order(Entity, EventMixin):
    __hash__ = Entity.__hash__

    status: OrderStatus
    ordered_products: list[OrderedProduct]
    customer_personal_info: CustomerPersonalInformation
    customer_note: Note | None
    message_customer: bool
    time_info: TimeInfo

    @classmethod  Kirill Panchenko
    def new(
        cls,
        customer_note: str | None,
        message_customer: bool,
        customer_personal_info: CustomerPersonalInformation,
        ordered_products: list[OrderedProduct],
        status: OrderStatus,
    ) -> "Order":...

    def reserve_ordered_products(self) -> None:...

    def start_processing( 4 usages  Kirill Panchenko
        self,
        order_status: OrderStatus,
    ) -> None:...

    def mark_as_failed_for_products_reservation( 3 usages  Kirill Panchenko
        self,
        order_status: OrderStatus,
    ) -> None:...

    def complete( 1 usage  Kirill Panchenko
        self,
        order_status: OrderStatus,
    ) -> None:...

    def set_status( 3 usages  Kirill Panchenko
        self,
        order_status: OrderStatus,
    ) -> None:...
```

Рисунок 3.19. - Сутність Order

Такий розподіл між сутностями та об’єктами значення дає змогу досягти високої когерентності моделі, зменшити дублювання коду та полегшити підтримку цілісності доменних інваріантів.

Для того щоб зробити код більш масштабованим та легким в майбутній підтримці та розширені, дані сутності та об’єкти значення були перенесенні в кодову базу додатку.

Як бачимо, сутність Order на малюнку має атрибути та описану поведінку, що відображає основні дії, які робляться над замовленням, наприклад:

a. зарезервувати товари в цьому замовленні

b. почати обробку цього замовлення

c. позначити як необробленим через те що не вдалося зарезервувати товари

d. виконати замовлення

Цю сутність потім використовується в інших компонентах - чито в сервісах, репозиторіях, тощо.

По тій ж самій логіці зроблені й інші компоненти системи в доменному рівні. На рисунку 3.20. наведені інші шари програми:

Рисунок 3.20. - Шари додатку сервісу замовлень

Структура проєкту, представлена на зображенні, реалізує підхід багатошарової архітектури відповідно до концепцій предметно-орієнтованого

проєктування (DDD — Domain-Driven Design). Кожен із шарів у цій архітектурі виконує чітко визначені функції, що дозволяє досягнути високого рівня розділення обов'язків, зниження зв'язності між компонентами та забезпечення незалежності бізнес-логіки від технічних аспектів реалізації. Основу архітектури становить доменний шар, у якому зосереджена основна бізнес-логіка системи. У межах цього шару реалізовано сутності, агрегати, об'єкти значення та доменні сервіси. Наприклад, у папці domain/order описано всі бізнес-об'єкти, що належать до замовлень. Доменна модель повністю ізольована від інфраструктурних залежностей та технологій.

Застосунковий шар (Application) відповідає за координацію виконання сценаріїв використання системи, які представлені у вигляді окремих команд. Кожна команда реалізує конкретну прикладну операцію, наприклад створення замовлення або запит на зворотний дзвінок. Команди взаємодіють із доменною моделлю для досягнення цілей користувача, але не містять жодної власної бізнес-логіки. Таким чином, застосунковий шар виступає в ролі посередника між зовнішніми інтерфейсами та предметною областю. Ось як виглядає цей шар (рис. 3.21.):

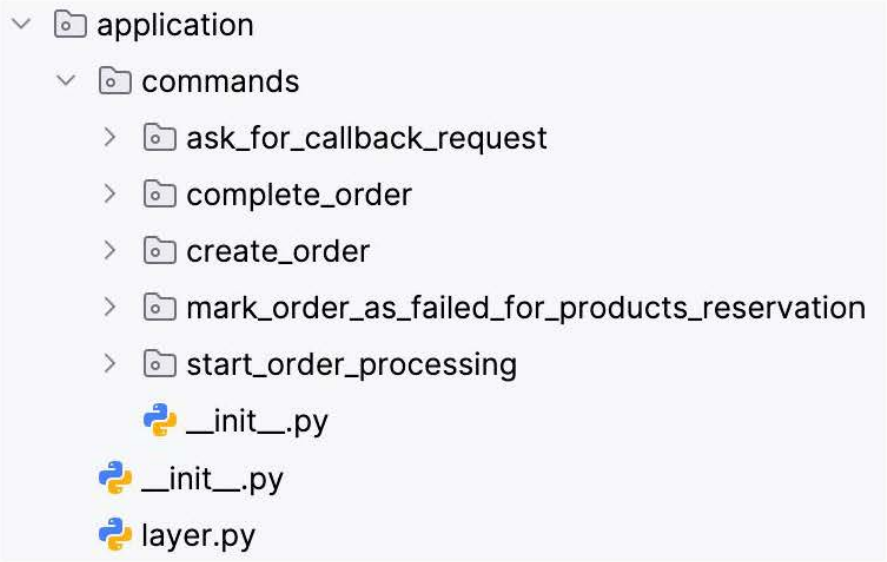

Рисунок 3.21. - Application шар

Інфраструктурний шар (Infrastructure) реалізує взаємодію з зовнішніми ресурсами та технічними засобами, необхідними для функціонування системи. Він містить реалізації доступу до бази даних, інтеграції з брокерами повідомлень, зовнішніми API, системами моніторингу, а також компоненти для конфігурації. Усі залежності з цього шару інжектуються в застосунковий або доменний шар через абстракції, що дозволяє уникати жорсткого зв'язування. Таким чином, зміна або заміна інфраструктурних компонентів не вимагає модифікації бізнес-логіки.

Інтерфейсний шар (Interface) забезпечує взаємодію зовнішніх клієнтів із системою. У цьому шарі розміщено веб-інтерфейс, що реалізує обробку HTTP-запитів, а також інтерфейси черг, які забезпечують асинхронну обробку повідомлень. Шар інтерфейсу делегує обробку запитів застосунковому шару, не містячи у собі бізнес-логіки, що дозволяє легко змінювати або масштабувати інтерфейсні компоненти без впливу на внутрішню структуру додатку.

За такими ж принципами були розроблені й інші компоненти системи. Саме DDD стало ключовим інструментом, для того щоб розділити обов'язки в різні системи та спроектувати ці системи надійно та для подальшого розширення.

3.5. Висновки до третього розділу

У результаті розробки програмного забезпечення для системи електронної комерції було створено повноцінний багатокомпонентний веб-додаток, який охоплює фронтенд, бекенд та інфраструктурну частини. Уся розробка проводилась із дотриманням принципів модульності, масштабованості та з урахуванням сучасних підходів до архітектури програмного забезпечення.

Фронтенд частина системи реалізована як односторінковий веб-застосунок (SPA) з використанням сучасного JavaScript-фреймворку Angular, що забезпечує високу продуктивність, гнучкість розробки та зручну інтерактивну взаємодію

користувача з інтерфейсом. Завдяки Angular вдалося реалізувати динамічне оновлення вмісту без перезавантаження сторінки, організувати чітку модульну структуру коду, ефективну маршрутизацію та двосторонню прив'язку даних, що значно покращує користувацький досвід і спрощує підтримку та масштабування системи. Клієнтський інтерфейс дозволяє користувачам переглядати товари, оформлювати замовлення, залишати коментарі, а також запитувати консультацію в адміністрації сайту. Було забезпечено інтерактивність інтерфейсу, асинхронну взаємодію з сервером. Окрему увагу приділено зручності користування та швидкодії. Завдяки використанню компонентного підходу та керування станом інтерфейсу було досягнуто високої гнучкості у розробці та супроводі клієнтської частини.

Бекенд частина реалізована на основі мікросервісної архітектури, що забезпечило гнучке розділення відповідальності між різними підсистемами. Було створено чотири окремі мікросервіси: два реалізовані мовою Python (обробка замовлень та керування коментарями), інші два — мовою Java. Поділ на сервіси здійснювався згідно з принципами предметно-орієнтованого проектування (DDD), де кожен мікросервіс реалізує власну підмодель бізнес-логіки. Для реалізації бекенду застосовувались шари домену, застосунку, інфраструктури та інтерфейсу, що дозволило ізолювати бізнес-логіку від технічних деталей реалізації та забезпечити чисту архітектуру. Застосування DDD дало змогу зберігати відповідність між вимогами бізнесу та реалізацією на технічному рівні, а також спростити процес впровадження змін у майбутньому.

Кожен мікросервіс мав власну базу даних, що дозволяє уникнути надмірної зв'язаності між сервісами та забезпечити незалежне масштабування. Для організації асинхронної взаємодії між мікросервісами використовувалась система обміну повідомленнями RabbitMQ, що дозволило реалізувати реактивну модель обробки подій. Для зберігання файлів було використано об'єктне сховище MinIO, яке забезпечує швидкий доступ та надійне зберігання мультимедійного контенту, пов'язаного з товарами або відгуками користувачів.

Інфраструктурна частина включає налаштування середовища виконання, зокрема бази даних PostgreSQL, брокерів повідомлень, інструментів моніторингу та зовнішніх клієнтів. Взаємодія між сервісами здійснюється як синхронно через REST API, так і асинхронно через систему обміну повідомленнями. Для забезпечення стабільної роботи системи було реалізовано обробку подій, логування, спостереження за метриками та використано інструменти контейнеризації для розгортання. Усі компоненти системи були упаковані в Docker-контейнери, що дозволило забезпечити їх незалежний запуск, спрощене розгортання та гнучке масштабування. Для централізованого збору логів застосовано сучасні стекові рішення, які дають змогу проводити діагностику та моніторинг у режимі реального часу. За допомогою Prometheus та Grafana було реалізовано збір і візуалізацію метрик, що дало змогу оперативно виявляти проблеми в роботі системи.

Загалом, у межах розробки було реалізовано взаємопов'язаний набір компонентів, які формують повноцінну систему електронної комерції, здатну до подальшого масштабування та розвитку. Обрана архітектура та технічні рішення забезпечують гнучкість, розширюваність та підтримуваність програмного продукту. Завдяки мікросервісному підходу система є стійкою до відмов окремих компонентів і здатна розвиватись у відповідь на зміну бізнес-вимог. Розроблений додаток може використовуватись як платформа для продажу енергетичних товарів, із можливістю подальшої інтеграції додаткових функцій, таких як платіжні шлюзи, система лояльності або CRM-модулі. Робота демонструє приклад успішної реалізації сучасних інженерних підходів у розробці прикладних веб-систем.

ВИСНОВКИ

У межах виконання дипломної роботи було розроблено серверний застосунок системи електронної комерції, побудований на мікросервісній архітектурі та реалізований із використанням сучасних підходів до проектування, програмування та інфраструктурного забезпечення. Розроблена система дозволяє ефективно забезпечувати ключові бізнес-процеси, пов'язані з продажем енергетичних товарів, включаючи обробку замовлень, зберігання та відображення інформації про товари, роботу з коментарями клієнтів, а також підтримку запитів на зворотний зв'язок. У результаті створено масштабований, підтримуваний і гнучкий у розгортанні програмний продукт, який може слугувати основою для подальшого розвитку електронного бізнесу.

Розробка системи передбачала поділ на фронтенд, бекенд та інфраструктурну частини. Фронтенд частина реалізована як односторінковий застосунок (SPA), що забезпечує швидку та інтерактивну взаємодію з користувачем. Клієнтський інтерфейс дозволяє зручно переглядати каталог товарів, фільтрувати та сортувати їх за категоріями, переглядати відгуки інших користувачів, додавати власні коментарі, формувати замовлення та ініціювати запити на консультацію. Завдяки застосуванню сучасних технологій вдалося реалізувати високий рівень динамічності інтерфейсу, забезпечити повторне використання компонентів, а також ефективно керувати станом додатку. Це, у свою чергу, підвищує зручність використання системи та створює позитивний досвід для кінцевого користувача.

Бекенд частина системи була спроєктована згідно з мікросервісною архітектурною моделлю, яка передбачає поділ усієї системи на незалежні сервіси, кожен з яких відповідає за певну частину предметної області. Застосування підходу предметно-орієнтованого проектування (Domain-Driven Design, DDD) дозволило логічно виділити окремі субдомени, зокрема обробку замовлень, керування коментарями, взаємодію з продуктами та обробку запитів

на зворотний зв'язок. Такий підхід уможливив більш гнучке масштабування системи, полегшення тестування окремих її компонентів, а також підвищення ізольованості бізнес-логіки від інфраструктурних залежностей. Два з мікросервісів реалізовані мовою програмування Python із використанням відповідних фреймворків для створення API, обробки запитів і взаємодії з базами даних. Інші сервіси, що обробляють інформацію про товари та категорії, були реалізовані мовою Java, що дозволило ефективно застосувати об'єктно-орієнтовані підходи й засоби екосистеми Spring для побудови надійних веб-сервісів.

Кожен із мікросервісів розроблено за принципами чистої архітектури з виокремленням окремих шарів: шар застосунку, що містить обробники команд та сценарії виконання бізнес-функціоналу; доменний шар, у якому описані сутності, value object-и, доменні події та сервіси; інфраструктурний шар, що забезпечує взаємодію з базою даних, зовнішніми сервісами та чергами повідомлень; а також інтерфейсний шар, відповідальний за прийом зовнішніх запитів або повідомлень. Така багатошарова структура дозволила досягти високого рівня декомпозиції, спростити тестування та підвищити гнучкість системи.

Інфраструктурна частина системи охоплює взаємодію з базами даних, брокерами повідомлень, зовнішніми клієнтами та сервісами моніторингу. Для зберігання структурованих даних було використано реляційні бази даних, що дозволило забезпечити цілісність та консистентність інформації. Для асинхронної взаємодії між мікросервісами застосовувався брокер повідомлень, який дозволяє будувати подієво-орієнтовану архітектуру та реагувати на зміну стану системи без прямого зв'язку між сервісами. Це позитивно вплинуло на масштабованість і відмовостійкість системи. Система логування, моніторингу й сповіщень була запроваджена з метою забезпечення прозорості у роботі сервісів, виявлення проблем на ранніх етапах та полегшення обслуговування. Застосування контейнеризації (наприклад, Docker) дозволило уніфікувати

середовище розгортання, спростити деплоймент і прискорити процес тестування.

З точки зору бізнесу розроблений додаток створює низку переваг. По-перше, автоматизація процесу замовлення товарів та організація зворотного зв'язку з клієнтами дозволяє скоротити час обробки звернень і покращити якість обслуговування. По-друге, впровадження коментарів і відгуків клієнтів дає змогу формувати репутацію товарів та підвищувати довіру до платформи. По-третє, завдяки чіткому розподілу обов'язків між компонентами та можливості масштабування окремих частин системи у відповідь на зміну навантаження, забезпечується стабільність і готовність до зростання кількості користувачів. Крім того, система дозволяє збирати дані про активність користувачів, що відкриває можливості для аналітики, маркетингових досліджень та прийняття бізнес-рішень на основі фактичної інформації.

У процесі виконання дипломної роботи за спеціальністю 121 «Інженерія програмного забезпечення» було реалізовано низку спеціальних (фахових) компетентностей, визначених Стандартом вищої освіти України для першого (бакалаврського) рівня цієї спеціальності .

Зокрема, у роботі відображено здатність застосовувати знання та навички для розв'язання складних спеціалізованих задач і практичних проблем у сфері інженерії програмного забезпечення. Це включає вміння аналізувати вимоги до програмного забезпечення, проектувати архітектуру систем, реалізовувати програмні компоненти з використанням сучасних мов програмування та технологій, а також забезпечувати якість і надійність програмних продуктів.

Крім того, у дипломному проєкті було продемонстровано здатність до використання сучасних засобів управління проєктами та систем контролю версій, що відповідає вимогам до професійної діяльності в галузі програмної інженерії.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ринок електронної комерції в Україні: сучасний стан та тенденції повоєнного відновлення – ppeu.stu.cn.ua. URL: http://ppeu.stu.cn.ua/article/view/293162 (дата звернення: 17.05.2025).

2. What is Business Automation? Softwareag URL: https://www.softwareag.com/en_corporate/resources/process-management/article/business-automation.html (дата звернення: 15.05.2025).

3. Horizontal Vs. Vertical Scaling: Which Should You Choose? Cloudzero. URL: https://www.cloudzero.com/blog/horizontal-vs-vertical-scaling/ (дата звернення: 16.05.2025).

4. Asynchronous Communication Between Microservices: Flask, RabbitMQ, and Docker. Medium. URL: https://medium.com/@33p.peu/asynchronous-communication-between-microservices-flask-rabbitmq-and-docker-b8ca7fad48c0 (дата звернення: 17.05.2025).

5. What Is Shopify and How Does It Work? Shopify. URL: https://www.shopify.com/blog/what-is-shopify (дата звернення: 15.05.2025).

6. Що таке ERP? Oracle. URL: https://www.oracle.com/ua/erp/what-is-erp/ (дата звернення: 15.05.2025).

7. What is SaaS (software as a service)? Techtarget. URL: https://www.techtarget.com/searchcloudcomputing/definition/Software-as-a-Service (дата звернення: 16.05.2025).

8. Magento Platform Overview. Adwservice URL: https://adwservice.com.ua/en/magento-platform-overview (дата звернення: 17.05.2025).

9. WooCommerce | Wordpress. URL: https://wordpress.org/plugins/woocommerce/ (дата звернення: 17.05.2025).

10. Application Programming Interface (API): що це, як і де працює. GoIT. URL: https://goit.global/ua/articles/application-programming-interface-api-shcho-tse-iak-i-de-pratsiuie/ (дата звернення: 21.05.2025).

11. Git. Git-cms. URL: https://git-scm.com/ (дата звернення: 20.05.2025).

12. Що таке Docker і навіщо він? Qagroup. URL: https://qagroup.com.ua/publications/shcho-take-docker-i-navishcho-vin/ (дата звернення: 18.05.2025).

13. Docker compose: опис та використання. Freehost. URL: https://freehost.com.ua/ukr/faq/articles/docker-compose/ (дата звернення: 19.05.2025).

14. What is a container? Docker. URL: https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-a-container/ (дата звернення: 19.05.2025).

15. Що таке bash в Linux? Гайд по написанню bash-скриптів. Acode. URL: https://acode.com.ua/bash-in-linux/ (дата звернення: 21.05.2025).

16. Що таке Nginx? Чим nginx краще за інші веб-сервери. Freehost. URL: https://freehost.com.ua/ukr/faq/wiki/chto-takoe-nginx/ (дата звертання: 21.05.2025).

17. PostgreSQL About. Postgresql. URL: https://www.postgresql.org/about/ (дата звернення: 21.05.2025).

18. MinIO Object Storage for Linux. Minio. URL: https://min.io/docs/minio/linux/index.html (дата звернення: 21.05.2025).

19. Що таке Domain-Driven Design та на якому етапі варто його впроваджувати в продукт. Dou. URL: https://dou.ua/forums/topic/39874/ (дата звернення: 21.05.2025).

20. SOLID: The First 5 Principles of Object Oriented Design | DigitalOcean. URL: https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design (дата звернення: 21.05.2025).

21. Feature Matrix. Postgresql. URL: https://www.postgresql.org/about/featurematrix/ (дата звернення: 21.05.2024).

ДОДАТОК А

```
name: order

x-app-setup: &app-setup
  build: &app-build
    args:
      GROUP_ID: "${DOCKER_ORDER_IMAGE_GROUP_ID}"
      USER_ID: "${DOCKER_ORDER_IMAGE_USER_ID}"
      INSTALL_LOCAL_SHARED_KERNEL:
"${DOCKER_INSTALL_LOCAL_SHARED_KERNEL}"
    dockerfile: Dockerfile
    context: .
    additional_contexts:
      - order-app=../app
      - shared-kernel=../../dw_shared_kernel
  environment: &app-env
    APPLICATION_TITLE: ${APPLICATION_TITLE}
    APPLICATION_DEBUG: ${APPLICATION_DEBUG}
    APPLICATION_VERSION: ${APPLICATION_VERSION}
    DATABASE_USERNAME: ${DATABASE_USERNAME}
    DATABASE_PASSWORD: ${DATABASE_PASSWORD}
    DATABASE_HOST: ${DATABASE_HOST}
    DATABASE_PORT: ${DATABASE_PORT}
    DATABASE_DATABASE: ${DATABASE_DATABASE}
    RABBITMQ_USERNAME: ${RABBITMQ_USERNAME}
    RABBITMQ_PASSWORD: ${RABBITMQ_PASSWORD}
    RABBITMQ_HOST: ${RABBITMQ_HOST}
    RABBITMQ_PORT: ${RABBITMQ_PORT}
```

```
      RABBITMQ_CATALOG_RESERVATION_QUEUE:
${RABBITMQ_CATALOG_RESERVATION_QUEUE}
      SMTP_HOST: ${SMTP_HOST}
      SMTP_PORT: ${SMTP_PORT}
      SMTP_TIMEOUT: ${SMTP_TIMEOUT}
      SMTP_USE_TLS: ${SMTP_USE_TLS}
      SMTP_SENDER_EMAIL: ${SMTP_SENDER_EMAIL}
      IAC_HOST: ${IAC_HOST}
      IAC_PORT: ${IAC_PORT}
      IAC_API_KEY: ${IAC_API_KEY}
      CATALOG_SERVICE_HOST: ${CATALOG_SERVICE_HOST}
      CATALOG_SERVICE_PORT: ${CATALOG_SERVICE_PORT}
    volumes:
      - ../app/src:/app
  services:
    order-backend-app:
      <<: *app-setup
      image: startupcorp/order-backend-app
      container_name: order-backend-app
      build:
        <<: *app-build
        target: backend-app
      environment:
        <<: *app-env
        JWT_PUBLIC_KEY: ${JWT_PUBLIC_KEY}
        JWT_ALGORITHM: ${JWT_ALGORITHM}
      ports:
```

```
      -
'${DOCKER_ORDER_APPLICATION_EXPOSE_PORT}:${ORDER_APPLICATI
ON_PORT}'
    entrypoint: [
      "uvicorn", "interface.web.app:web_app",
      "--host", "0.0.0.0",
      "--port", "${ORDER_APPLICATION_PORT}",
      "--factory",
      "--reload",
      "--reload-dir", ".",
    ]
    depends_on:
      order-migration-service:
        condition: service_completed_successfully
  order-queue-consumer:
    <<: *app-setup
    image: startupcorp/order-queue-consumer
    container_name: order-queue-consumer
    build:
      <<: *app-build
      target: queue-consumer
    entrypoint: [
      "faststream", "run",
      "--factory",
      "interface.queue.app:queue_app",
    ]
    environment:
      <<: *app-env
      RABBITMQ_ORDER_QUEUE: ${RABBITMQ_ORDER_QUEUE}
```

```
    depends_on:
      order-migration-service:
        condition: service_completed_successfully
  order-migration-service:
    <<: *app-setup
    image: startupcorp/order-migration-service
    container_name: order-migration-service
    build:
      <<: *app-build
      target: migration-service
    command: "migration-run"
    volumes:
      - ../app/src/infrastructure/database/relational/migrations/:/app/migrations
  smtp-server:
    image: axllent/mailpit
    container_name: order-smtp-server
    ports:
      - "${DOCKER_SMTP_SERVER_EXPOSE_PORT}:8025"
networks:
  default:
    name: deye-web-network
    external: true
```

ДОДАТОК Б

```
## =====================================
## ==== INFRASTRUCTURE CONFIGURATION ====
## =====================================
## ==== DOCKER COMPOSE EXPOSE PORT
export DOCKER_ORDER_APPLICATION_EXPOSE_PORT=8020
export DOCKER_SMTP_SERVER_EXPOSE_PORT=8025
## ==== DOCKER BUILD
export DOCKER_ORDER_IMAGE_GROUP_ID="$(test "$(id -u)" -lt 1000 && echo 1000 || id -u)"
export DOCKER_ORDER_IMAGE_USER_ID="$(test "$(id -g)" -lt 1000 && echo 1000 || id -g)"
export DOCKER_INSTALL_LOCAL_SHARED_KERNEL=false
## ==== ORDER APPLICATION
export ORDER_APPLICATION_PORT=8020
## ===================================
## ==== APPLICATION CONFIGURATION ====
## ===================================
## ==== APPLICATION
export APPLICATION_DEBUG=1
export APPLICATION_VERSION=0.0.1
export APPLICATION_TITLE="Order Microservice"
## ==== DATABASE
export DATABASE_HOST=database
export DATABASE_PORT=5432
export DATABASE_DATABASE=order_
export DATABASE_USERNAME=order_user
export DATABASE_PASSWORD=order_password
```

```
## ==== RABBITMQ
export RABBITMQ_USERNAME=dev
export RABBITMQ_PASSWORD=devdev
export RABBITMQ_HOST=rabbitmq
export RABBITMQ_PORT=5672
export RABBITMQ_ORDER_QUEUE=order.queue
export RABBITMQ_CATALOG_RESERVATION_QUEUE='{"NAME": "catalog.reservation", "EXCHANGE": "deye_web.direct"}'
## ==== JWT
export JWT_PUBLIC_KEY="
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAprbm7KfT9x
cYGiOvYCyZ
UqnObuIIlQNJDJ7xUk0QaKPVaSLBKM5+eHtRfesj+xjtDv1cpm+GWqtvRV
9QskYm
9Qnfhx2mSu0OdbdXsdYbF/IGuxPdN9J6pM4ZFeRb9Ta41AHvnazTJPvsGQ
K/uBRD
+QUKFq7ttrxUuo8540d6HThbww+mniu8u/RTFd2NFnKWGQvyo67JaO/CO
NTcwp11
eT+3+qRMH0JXJYHfsK2A4VYZSTYdf2ZB3q6+LtToLsXkd//1Bm5nf/03/2
XfaT/m
J7q4A0OIwqs0JspWBrPrP/wBCZ7mKRbjaBQFrczYksVE+Z/acAlTD3rlh09v
QTww
BwIDAQAB
-----END PUBLIC KEY-----
"
export JWT_ALGORITHM=RS256
## ==== CATALOG SERVICE
export CATALOG_SERVICE_HOST=catalog-backend-app
```

```
export CATALOG_SERVICE_PORT=8080
## ==== IAC SERVICE
export IAC_HOST=iac-backend-app
export IAC_PORT=8080
export IAC_API_KEY=H35EFIKWNDVK22AQJPWO1DUJQ9U1FJ
## ==== SMTP
export SMTP_HOST=smtp-server
export SMTP_PORT=1025
export SMTP_TIMEOUT=5
export SMTP_USE_TLS=false
export SMTP_SENDER_EMAIL=deye.ukraine@gmail.com
## ====================================
## =======     FUNCTIONS     =======
## ====================================
export _FUNCTIONS_LOCATION=$( cd -- "$( dirname -- "${BASH_SOURCE[0]}" )" &> /dev/null && pwd )
# Starts the microservice and infrastructure part.
order-up() (
   cd "$_FUNCTIONS_LOCATION" && \
   (cd ../../../infrastructure/ && source .env.local && infra-up) && \
   app-up
)
# Stops the microservice.
order-down() (
   cd "$_FUNCTIONS_LOCATION" && \
   app-down
   (cd ../../../infrastructure/ && source .env.local && infra-down)
)
```

```
# Builds the microservice docker image.
app-build() (
    cd "$_FUNCTIONS_LOCATION" && \
    docker compose build
    docker image prune -f > /dev/null
)
# Starts only the microservice.
app-up() (
    cd "$_FUNCTIONS_LOCATION" && \
    app-build
    docker compose up -d
    docker compose wait order-migration-service &> /dev/null
    docker compose rm -fv > /dev/null
)
# Stops only the microservice.
app-down() (
    cd "$_FUNCTIONS_LOCATION" && \
    docker compose down
)
# Runs database migration files.
migration-run() (
    cd "$_FUNCTIONS_LOCATION" && \
    docker compose run --rm order-migration-service migration-run
)
# Downgrades the latest database migration.
migration-rollback() (
    cd "$_FUNCTIONS_LOCATION" && \
    docker compose run --rm order-migration-service migration-rollback
)
```

ДОДАТОК В

```
name: infrastructure
services:
  rabbitmq:
    container_name: rabbitmq
    image: rabbitmq:4.0.5-management
    hostname: rabbitmq
    ports:
      - '${DOCKER_RABBITMQ_EXPOSE_PORT}:15672'
      - '${DOCKER_RABBITMQ_EXPOSE_PORT_AMQP}:5672'
    restart: always
    environment:
      - "RABBITMQ_DEFAULT_USER=${RABBITMQ_USERNAME}"
      - "RABBITMQ_DEFAULT_PASS=${RABBITMQ_PASSWORD}"
      - "RABBITMQ_SERVER_ADDITIONAL_ERL_ARGS=-rabbit log_levels
[{connection,error},{default,error}]"
    volumes:
      - rabbitmq_data:/var/lib/rabbitmq
    healthcheck:
      test: rabbitmq-diagnostics check_port_connectivity
      interval: 1s
      timeout: 10s
      retries: 15
  database:
    container_name: database
    image: postgres:latest
    environment:
      - "POSTGRES_USER=${POSTGRES_USERNAME}"
      - "POSTGRES_PASSWORD=${POSTGRES_PASSWORD}"
```

```
      - "POSTGRES_DB=postgres"
    ports:
      - '${DOCKER_DB_EXPOSE_PORT}:5432'
    volumes:
      - database_data:/var/lib/postgresql/data/
    healthcheck:
      test: [ "CMD-SHELL", "pg_isready -U ${POSTGRES_USERNAME} -d
postgres" ]
      interval: 1s
      timeout: 10s
      retries: 15
  nginx:
    container_name: nginx
    image: nginx:1.24-alpine
    ports:
      - '${DOCKER_NGINX_EXPOSE_PORT}:9999'
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
      - /tmp/nginx_cache:/tmp/nginx_cache
    depends_on:
      minio:
        condition: service_healthy
  minio:
    container_name: minio
    image: quay.io/minio/minio
    ports:
      - "${DOCKER_MINIO_EXPOSE_PORT_UI}:9000"
      - "${DOCKER_MINIO_EXPOSE_PORT_CONSOLE}:9001"
    environment:
```

```
      - MINIO_ROOT_USER=${MINIO_ACCESS_KEY}
      - MINIO_ROOT_PASSWORD=${MINIO_SECRET_KEY}
    command: server /data --console-address ":9001"
    volumes:
      - minio_data:/data
    healthcheck:
      test: "curl -k -f http://localhost:9001/minio/health/live || exit 1"
      interval: 1s
      timeout: 15s
      retries: 30
  grafana:
    image: grafana/grafana
    ports:
      - "${DOCKER_GRAFANA_EXPOSE_PORT}:3000"
    volumes:
      -
./grafana_config/dashboards.yaml:/etc/grafana/provisioning/dashboards/dashboards.y
aml
      - ./grafana_config/dashboards:/var/lib/grafana/dashboards
      - ./grafana_config/datasources/:/etc/grafana/provisioning/datasources/
    environment:
      - GF_SECURITY_ADMIN_USER=${GRAFANA_USERNAME}
      - GF_SECURITY_ADMIN_PASSWORD=${GRAFANA_PASSWORD}
  prometheus:
    container_name: prometheus
    image: prom/prometheus
    ports:
      - "${DOCKER_PROMETHEUS_EXPOSE_PORT}:9090"
    command:
```

```
    - '--config.file=/etc/prometheus/prometheus.yml'
  volumes:
    - ./prometheus.yaml:/etc/prometheus/prometheus.yml
  restart: always
 resources-setup:
  container_name: resources-setup-script
  image: alpine
  entrypoint: [ "sh","-c" ]
  command:
    - |
      apk -q add curl python3 postgresql-client minio-client
      mcli alias set myminio http://minio:9000 ${MINIO_ACCESS_KEY}
${MINIO_SECRET_KEY}
      mcli mb myminio/${MINIO_BUCKET_NAME}
      mcli anonymous set download myminio/${MINIO_BUCKET_NAME}
      curl --no-progress-meter -s -o /usr/local/bin/rabbitmqadmin
http://${SETUP_RABBITMQ_HOST}:${SETUP_RABBITMQ_PORT}/cli/rabbitm
qadmin
      chmod +x /usr/local/bin/rabbitmqadmin
      alias rabbitmqadmin="rabbitmqadmin --
host=${SETUP_RABBITMQ_HOST} --port=${SETUP_RABBITMQ_PORT} --
username=${SETUP_RABBITMQ_USERNAME} --
password=${SETUP_RABBITMQ_PASSWORD}"
      alias psql="PGPASSWORD=${SETUP_DB_PASSWORD} psql -U
${SETUP_DB_USERNAME} -h ${SETUP_DB_HOST} -p ${SETUP_DB_PORT} -
d postgres"
      rabbitmqadmin declare exchange
name=${SETUP_RABBITMQ_EXCHANGE_NAME} type=direct durable=true
```

```
        rabbitmqadmin -q declare queue
name=${SETUP_RABBITMQ_COMMENT_PRODUCT_QUEUE} durable=true
        rabbitmqadmin -q declare queue
name=${SETUP_RABBITMQ_DEAD_LETTER_COMMENT_PRODUCT_QUEU
E} durable=true
        rabbitmqadmin -q declare queue
name=${SETUP_RABBITMQ_CATALOG_RESERVATION_QUEUE}
durable=true
        rabbitmqadmin -q declare queue
name=${SETUP_RABBITMQ_DEAD_LETTER_CATALOG_RESERVATION_Q
UEUE} durable=true
        rabbitmqadmin -q declare queue
name=${SETUP_RABBITMQ_ORDER_RESERVATION_QUEUE} durable=true
        rabbitmqadmin -q declare queue
name=${SETUP_RABBITMQ_DEAD_LETTER_ORDER_RESERVATION_QUE
UE} durable=true
        rabbitmqadmin declare binding \
          source=${SETUP_RABBITMQ_EXCHANGE_NAME} \
        destination=${SETUP_RABBITMQ_COMMENT_PRODUCT_QUEUE}
\
    routing_key=${SETUP_RABBITMQ_COMMENT_PRODUCT_QUEUE}
        rabbitmqadmin declare binding \
          source=${SETUP_RABBITMQ_EXCHANGE_NAME} \
    destination=${SETUP_RABBITMQ_DEAD_LETTER_COMMENT_PRODU
CT_QUEUE} \
    routing_key=${SETUP_RABBITMQ_DEAD_LETTER_COMMENT_PROD
UCT_QUEUE}
        rabbitmqadmin declare binding \
          source=${SETUP_RABBITMQ_EXCHANGE_NAME} \
```

```
destination=${SETUP_RABBITMQ_CATALOG_RESERVATION_QUEUE} \
routing_key=${SETUP_RABBITMQ_CATALOG_RESERVATION_QUEUE}
        rabbitmqadmin declare binding \
          source=${SETUP_RABBITMQ_EXCHANGE_NAME} \
    destination=${SETUP_RABBITMQ_DEAD_LETTER_CATALOG_RESERV
ATION_QUEUE} \
    routing_key=${SETUP_RABBITMQ_DEAD_LETTER_CATALOG_RESER
VATION_QUEUE}
        rabbitmqadmin declare binding \
          source=${SETUP_RABBITMQ_EXCHANGE_NAME} \
    destination=${SETUP_RABBITMQ_ORDER_RESERVATION_QUEUE} \
    routing_key=${SETUP_RABBITMQ_ORDER_RESERVATION_QUEUE}
        rabbitmqadmin declare binding \
          source=${SETUP_RABBITMQ_EXCHANGE_NAME} \
    destination=${SETUP_RABBITMQ_DEAD_LETTER_ORDER_RESERVAT
ION_QUEUE} \
    routing_key=${SETUP_RABBITMQ_DEAD_LETTER_ORDER_RESERVA
TION_QUEUE}
        rabbitmqadmin declare policy \
          name="DLXCommentProductQueue" \
    pattern="^${SETUP_RABBITMQ_COMMENT_PRODUCT_QUEUE}$" \
          definition='{"dead-letter-exchange":
"${SETUP_RABBITMQ_EXCHANGE_NAME}",          "dead-letter-routing-key":
"${SETUP_RABBITMQ_DEAD_LETTER_COMMENT_PRODUCT_QUEUE}"}' \
          apply-to=queues
        rabbitmqadmin declare policy \
          name="DLXReservationQueue" \
```

```
        pattern="^${SETUP_RABBITMQ_CATALOG_RESERVATION_QUEUE}$
" \
            definition='{"dead-letter-exchange":
"${SETUP_RABBITMQ_EXCHANGE_NAME}",          "dead-letter-routing-key":
"${SETUP_RABBITMQ_DEAD_LETTER_CATALOG_RESERVATION_QUEUE
}"}' \
            apply-to=queues
          rabbitmqadmin declare policy \
            name="DLXOrderQueue" \
        pattern="^${SETUP_RABBITMQ_ORDER_RESERVATION_QUEUE}$" \
            definition='{"dead-letter-exchange":
"${SETUP_RABBITMQ_EXCHANGE_NAME}",          "dead-letter-routing-key":
"${SETUP_RABBITMQ_DEAD_LETTER_ORDER_RESERVATION_QUEUE}"}
' \
            apply-to=queues
          psql -c "SELECT 1 FROM pg_database WHERE datname =
'${SETUP_DB_CATALOG_DATABASE}'" \
            | grep -q 1 || psql -c "CREATE DATABASE
${SETUP_DB_CATALOG_DATABASE}"
          psql -c "SELECT 1 FROM pg_roles WHERE rolname =
'${SETUP_DB_CATALOG_USERNAME}'" \
            | grep -q 1 || psql -c "CREATE USER
${SETUP_DB_CATALOG_USERNAME}          WITH          PASSWORD
'${SETUP_DB_CATALOG_PASSWORD}'"
          psql -c "ALTER DATABASE ${SETUP_DB_CATALOG_DATABASE}
OWNER TO ${SETUP_DB_CATALOG_USERNAME}"
          psql -c "GRANT ALL PRIVILEGES ON DATABASE
${SETUP_DB_CATALOG_DATABASE}                                TO
${SETUP_DB_CATALOG_USERNAME}"
```

```
        psql -c "SELECT 1 FROM pg_database WHERE datname =
'${SETUP_DB_COMMENT_DATABASE}'" \
        | grep -q 1 || psql -c "CREATE DATABASE
${SETUP_DB_COMMENT_DATABASE}"
        psql -c "SELECT 1 FROM pg_roles WHERE rolname =
'${SETUP_DB_COMMENT_USERNAME}'" \
        | grep -q 1 || psql -c "CREATE USER
${SETUP_DB_COMMENT_USERNAME} WITH PASSWORD
'${SETUP_DB_COMMENT_PASSWORD}'"
        psql -c "ALTER DATABASE
${SETUP_DB_COMMENT_DATABASE} OWNER TO
${SETUP_DB_COMMENT_USERNAME}"
        psql -c "GRANT ALL PRIVILEGES ON DATABASE
${SETUP_DB_COMMENT_DATABASE} TO
${SETUP_DB_COMMENT_USERNAME}"
        psql -c "SELECT 1 FROM pg_database WHERE datname =
'${SETUP_DB_IDENTITY_AND_ACCESS_DATABASE}'" \
        | grep -q 1 || psql -c "CREATE DATABASE
${SETUP_DB_IDENTITY_AND_ACCESS_DATABASE}"
        psql -c "SELECT 1 FROM pg_roles WHERE rolname =
'${SETUP_DB_IDENTITY_AND_ACCESS_USERNAME}'" \
        | grep -q 1 || psql -c "CREATE USER
${SETUP_DB_IDENTITY_AND_ACCESS_USERNAME} WITH PASSWORD
'${SETUP_DB_IDENTITY_AND_ACCESS_PASSWORD}'"
        psql -c "ALTER DATABASE
${SETUP_DB_IDENTITY_AND_ACCESS_DATABASE} OWNER TO
${SETUP_DB_IDENTITY_AND_ACCESS_USERNAME}"
```

```
            psql -c "GRANT ALL PRIVILEGES ON DATABASE
${SETUP_DB_IDENTITY_AND_ACCESS_DATABASE} TO
${SETUP_DB_IDENTITY_AND_ACCESS_USERNAME}"
            psql -c "SELECT 1 FROM pg_database WHERE datname =
'${SETUP_DB_ORDER_DATABASE}'" \
             | grep -q 1 || psql -c "CREATE DATABASE
${SETUP_DB_ORDER_DATABASE}"
            psql -c "SELECT 1 FROM pg_roles WHERE rolname =
'${SETUP_DB_ORDER_USERNAME}'" \
             | grep -q 1 || psql -c "CREATE USER
${SETUP_DB_ORDER_USERNAME} WITH PASSWORD
'${SETUP_DB_ORDER_PASSWORD}'"
            psql -c "ALTER DATABASE ${SETUP_DB_ORDER_DATABASE}
OWNER TO ${SETUP_DB_ORDER_USERNAME}"
            psql -c "GRANT ALL PRIVILEGES ON DATABASE
${SETUP_DB_ORDER_DATABASE} TO
${SETUP_DB_ORDER_USERNAME}"
        depends_on:
          rabbitmq:
            condition: service_healthy
          database:
            condition: service_healthy
          minio:
            condition: service_healthy
      networks:
        default:
          name: deye-web-network
          external: true
```