МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чорноморський національний університет
імені Петра Могили
Факультет комп'ютерних наук
Кафедра інтелектуальних інформаційних систем

ДОПУЩЕНО ДО ЗАХИСТУ
Завідувач кафедри інтелектуальних
інформаційних систем, д-р техн. наук, проф.
_____Ю. П. Кондратенко
«____»_____2024 р.

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

ІНФОРМАЦІЙНА СИСТЕМА ОБРОБКИ ЗАМОВЛЕНЬ ДЛЯ ІНТЕРНЕТ-МАГАЗИНУ

Спеціальність 122 «Комп'ютерні науки»

122 – КРБ – 401.2010306

Виконав студент 4-го курсу, групи 401
_____ М. С. Євстратьєв
«17» червня 2024 р.

Керівник: д-р техн. наук, доцент
_____І. О. Калініна
«17» червня 2024 р.

Миколаїв – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чорноморський національний університет ім. Петра Могили
Факультет комп'ютерних наук
Кафедра інтелектуальних інформаційних систем

Рівень вищої освіти **бакалавр**
Спеціальність **122 «Комп'ютерні науки»**
(шифр і назва)
Галузь знань **12 «Інформаційні технології»**
(шифр і назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри інтелектуальних інформаційних систем, д-р техн. наук, проф.
_____ Ю. П. Кондратенко
«____» _____2024 р.

З А В Д А Н Н Я
на виконання кваліфікаційної роботи

Видано студенту групи 401 факультету комп'ютерних наук Євстратьєву Максиму Сергійовичу.

1. Тема кваліфікаційної роботи «Інформаційна система обробки замовлень для інтернет-магазину».

Керівник роботи Калініна Ірина Олександрівна, д-р техн. наук, доцент.

Затв. наказом Ректора ЧНУ ім. Петра Могили від «28» грудня 2023 р. № 271

2. Строк представлення кваліфікаційної роботи студентом «17» червня 2024 р.

3. Вхідні (початкові) дані до роботи: експертні оцінки технологій обробки замовлень для підприємств; наявні архітектурні підходи та технології для розробки застосунків для інформаційних систем обробки замовлень.

Очікуваний результат: інформацій система на мікросервісній архітектурі для обробки замовлень для інтернет-магазину.

4. Перелік питань, що підлягають розробці (зміст пояснювальної записки):

- аналіз інформаційних систем обробки замовлень для інтернет-магазинів;

– архітектурні підходи, паттерни проектування та технології, що використовуються для створення інформаційної системи обробки замовлень для інтернет-магазину;

– розробка інформаційної системи.

5. Перелік графічного матеріалу: презентація.

6. Завдання до спеціальної частини: «Проведення оцінки умов праці при роботі за комп'ютером».

7. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис
Спеціальна частина з охорони праці	Алєксєєва А. О., доцент кафедри екології	

Керівник роботи _____д-р техн. наук, доц. І. О. Калініна_____
(наук. ступінь, вчене звання, прізвище та ініціали)

_____
(підпис)

Завдання прийнято до виконання ___Євстратьєв М. С._____
(прізвище та ініціали)

_____
(підпис)

Дата видачі завдання «_14_» _____січня____ 2024 р.

КАЛЕНДАРНИЙ ПЛАН
виконання кваліфікаційної роботи

Тема: Інтелектуальна система обробки замовлень для інтернет-магазину

№	Найменування роботи	Початок	Закінчення	Примітки
1	Подання заяви на затвердження теми та керівників КРБ	10.11.2023	15.11.2023	Виконано
2	Отримання завдання на виконання КРБ	10.01.2024	15.01.2024	Виконано
3	Складання календарного плану роботи на весь період виконання КРБ	16.01.2024	30.01.2024	Виконано
4	Отримання завдання на переддипломну практику	15.04.2024	29.04.2024	Виконано
5	Проходження переддипломної практики, збір та аналіз матеріалів до КРБ	29.04.2024	11.05.2024	Виконано
6	Розробка звіту з переддипломної практики	12.05.2024	15.05.2024	Виконано
7	Виконання КРБ: аналіз існуючих систем обробки замовлень для інтернет-магазинів, огляд існуючих технологій, розробка ПЗ	13.05.2024	22.06.2024	Виконано
8	Перший попередній захист КРБ на засіданні комісії кафедри	27.05.2024	27.05.2024	Виконано
9	Доробка та остаточне оформлення КРБ	28.05.2024	09.06.2024	Виконано
10	Другий попередній захист КРБ на засіданні комісії кафедри	10.06.2024	10.06.2024	Виконано
11	Подання КРБ рецензенту	13.06.2024	13.06.2024	Виконано
11	Подання КРБ, її електронної копії та інших документів (відгуку, рецензії) до захисту	17.06.2024	21.06.2024	Виконано
12	Захист КРБ перед екзаменаційною комісією (ЕК)	24.06.2024	28.06.2024	Виконано

Розробив студент __Євстратьєв М. С._____ _____
(прізвище, ім'я, по батькові студента) (*підпис*)

Керівник роботи ____д-р техн. наук, доц. Калініна І. О.___ _____
(*посада, прізвище, ім'я, по батькові)* *(підпис)*

«_29_» _____01_____ 2024 р.

АНОТАЦІЯ

кваліфікаційної роботи студента групи 401 ЧНУ ім. Петра Могили

Євстратьєва Максима Сергійовича

Тема: «Інформаційна система обробки замовлень для інтернет-магазину»

Дана кваліфікаційна робота присвячена аналізу та розробці інформаційної системи обробки замовлень для інтернет-магазину. Метою роботи є створення ефективної та надійної системи, що дозволить автоматизувати процеси обробки замовлень, зменшити час обробки, підвищити точність і знизити ймовірність помилок.

У роботі розглянуто основні вимоги до інформаційних систем такого типу, проведено аналіз існуючих рішень на ринку та визначено їх відмінності. На основі цього аналізу було розроблено власний проект системи, що використовує мікросервісну архітектуру, взаємодію з іншими сервісами та можливості інтеграції з іншими системами.

Розроблена система складається з декількох частин, серед яких:

– сервіс керування каталогом товарів інтернет-магазину;
– сервіс керування замовленнями;
– сервіс сповіщення у месенджер «Telegram»;
– сервіс аутентифікації;
– сервіс виявлення;
– сервіс конфігурацій;
– застосунок для взаємодії з системою.

Особливу увагу приділено питанням безпеки даних та захисту інформації, а також забезпеченню високої швидкості обробки даних та зручності використання системи як для адміністратора, так і для кінцевого користувача.

Запропонована інформаційна система дозволяє значно покращити процес обробки замовлень в інтернет-магазині, забезпечуючи високу швидкість, точність і безпеку.

Об’єкт роботи – процес розробки системи обробки замовлень в інтернет-магазинах. Це включає всі аспекти прийому, обробки, виконання та

відслідковування замовлень, а також взаємодії між клієнтами та менеджерами магазину.

Предмет роботи – програмні засоби і архітектурні шаблони для створення інформаційної система для обробки замовлень, яка забезпечить автоматизацію процесів прийому, обробки та виконання замовлень.

Метою даної роботи є дослідження архітектурних шаблонів, програмних засобів, методів та алгоритмів для створення інформаційної системи яка виконуватиме операції прийому замовлень від клієнтів, обробляти їх та сповіщати про надходження у вигляді повідомлення для менеджерів інтернет-магазину.

Кваліфікаційна робота бакалавра містить 89 сторінок, 44 рисунки, 1 таблицю, 31 використане джерело та 16 додатків.

Ключові слова: інформаційна система, інтернет-магазин, обробка замовлень, автоматизація, безпека даних, мікросервіс.

ABSTRACT

of a bachelor's degree work of a student of group 401 at Petro Mohyla Black Sea National University

Yevstratiev Maksym

Topic: "Information system for processing orders for an online store"

This qualification work is devoted to the analysis and development of an information system for processing orders for an online store. The goal of the work is to create an efficient and reliable system that will allow automating the processes of processing orders, reduce processing time, increase accuracy and reduce the likelihood of errors.

The paper considers the main requirements for information systems of this type, analyzes existing solutions on the market and identifies their differences. Based on this analysis, a proprietary system project was developed that uses microservice architecture, interaction with other services, and integration capabilities with other systems.

The developed system consists of several parts, including:

– online store product catalog management service;
– order management service;
– "Telegram" messenger notification service;
– authentication service;
– detection service;
– configuration service;
– an application for interaction with the system.

Particular attention is paid to issues of data security and information protection, as well as ensuring high speed of data processing and ease of use of the system for both the administrator and the end user.

The proposed information system allows to significantly improve the process of processing orders in the online store, ensuring high speed, accuracy and security.

The **object** of the work is the process of developing an order processing system in online stores. This includes all aspects of receiving, processing, fulfilling and tracking orders, as well as interactions between customers and store managers.

The **subject** of the work is software tools and architectural templates for creating an information system for processing orders, which will ensure the automation of the processes of receiving, processing and fulfilling orders.

The **purpose** of this work is to study architectural templates, software tools, methods and algorithms for creating an information system that will perform operations of receiving orders from customers, process them and notify about receipt in the form of a message for online store managers.

The bachelor's thesis contains 89 pages, 44 pictures, 1 table, 31 used sources and 16 appendices.

Keywords: information system, online store, order processing, automation, data security, microservice.

ЗМІСТ

ВСТУП

Обрана тема є актуальною в сучасному світі електронної комерції. З поглибленням цифровізації та зростанням кількості покупців, які обирають онлайн-покупки, важливо мати ефективну систему обробки замовлень. Це дозволяє магазинам забезпечити швидке та точне виконання замовлень, зменшити помилки та покращити задоволення клієнтів. Система обробки замовлень може бути розширена шляхом її інтеграції у більші системи, що розширить функціональність та сферу її використання.

У сучасному конкурентному середовищі інтернет-магазини стикаються з численними викликами, такими як висока конкуренція, очікування клієнтів щодо швидкої доставки, забезпечення високого рівня обслуговування клієнтів та оптимізація операційних витрат. Ефективна інформаційна система обробки замовлень може значно вплинути на здатність інтернет-магазину задовольняти ці вимоги. Така система автоматизує процеси прийому, обробки та виконання замовлень, знижуючи ймовірність помилок та підвищуючи загальну ефективність.

Метою даної роботи є розробка інформаційної системи що буде приймати замовлення від клієнта, обробляти їх та сповіщати про надходження у вигляді повідомлення для менеджерів інтернет-магазину. Клієнт повинен мати змогу відслідковувати стан замовлення. Система базуватиметься на технологіях, що мають високий рівень підтримки та гнучкості, що дозволить комбінувати різні інструменти в процесі розробки та забезпечить можливість для подальшого розширення.

Основні завдання включатимуть:

- аналіз існуючих рішень у сфері обробки замовлень;
- створення вимог для інформаційної системи;
- вибір технологій, методів, та архітектурних рішень для системи;
- розробка програмної реалізації системи.

Об’єкт роботи – процес розробки системи обробки замовлень в інтернет-магазинах. Це включає всі аспекти прийому, обробки, виконання та відслідковування замовлень, а також взаємодії між клієнтами та менеджерами магазину.

Предмет роботи – програмні засоби і архітектурні шаблони для створення інформаційної система для обробки замовлень, яка забезпечить автоматизацію процесів прийому, обробки та виконання замовлень.

Метою даної роботи є дослідження архітектурних шаблонів, програмних засобів, методів та алгоритмів для створення інформаційної системи яка виконуватиме операції прийому замовлень від клієнтів, обробляти їх та сповіщати про надходження у вигляді повідомлення для менеджерів інтернет магазину.

1 АНАЛІЗ ІНФОРМАЦІЙНИХ СИСТЕМ ОБРОБКИ ЗАМОВЛЕНЬ ДЛЯ ІНТЕРНЕТ-МАГАЗИНІВ. ПОСТАНОВКА ЗАДАЧІ

1.1 Опис предметної сфери

Інформаційна система обробки замовлень для інтернет-магазину призначена для автоматизації процесу прийому, обробки та виконання замовлень, зроблених клієнтами в інтернет-магазині. Система має забезпечувати ефективне та злагоджене функціонування всіх етапів обробки замовлень, від моменту оформлення замовлення на сайті магазину до його доставки клієнту.

Елементами системи є найпростіші, умовно неподільні частини цілої системи, які не можуть повноцінно функціонувати окремо і визначаються в залежності від визначених задач. Ці елементи можуть бути об’єднаними у підсистеми – складові частини цілісної системи які виконують задачі такі як передача, збереження, забезпечення цілісності та захисту інформації, структуроване відображення інформації, тощо. Відокремлення елементів у підсистеми покращує організацію між ними, забезпечує її надійність та структурну цілісність [1,2].

1.1.1 Класифікація інформаційних систем

Інформаційні системи класифікують за різними ознаками. В залежності від обсягу вирішуваних завдань, організації функціонування та технічних засобів та їх поділяють на класи.

За типом даних, які зберігаються в інформаційних системах визначають фактографічні і документальні. В документальних системах інформація представлена у формі документів такі як описи, тексти, найменування, реферати, тощо. Пошук інформації в таких системах виконується за допомогою семантичних ознак. Після цього знайдені документи надаються клієнту, без додаткової обробки даних.

За рівнем автоматизації інформаційні системи поділяють на автоматичні, автоматизовані та ручні. У автоматичних системах обробка інформації виконується без участі людини, в автоматизованих – часткова участь людини, але основну роль у обробці даних займає комп’ютер. У ручних інформаційних системах людина виконує всі операції, а сучасні технічні засоби обробки інформації відсутні.

За характером обробки даних виділяють інформаційно-розв’язуючі та інформаційно-пошукові інформаційні системи. Інформаційно-пошукові системи здійснюють такі дії як отримання даних, їх систематизація, зберігання та відправлення інформації у відповідь на запит користувача. У таких системах не використовуються складні перетворення даних. Прикладами можуть бути системи обслуговування бібліотек, продаж квитків, бронювання місць, тощо. Інформаційно-розв’язуючі ж системи виконують ще й операції обробки даних з використанням алгоритмів.

1.1.2 Обробка замовлень інтернет-магазинів

Обробкою замовлень інтернет-магазинів є управління та виконання замовлень клієнта. Цей процес може включати:

– автентифікацію та авторизацію клієнта;
– збереження даних про клієнта;
– створення замовлення клієнтом;
– попередня перевірка створеного замовлення;
– збереження та підтвердження замовлення за допомогою оператора;
– інформування клієнта про стан замовлення.

В процесі обробки замовлення збираються необхідні дані про клієнта. Одною з головних вимог до збереження даних це гарантування конфіденційності та захищеності даних від несанкціонованого доступу. Тож під час проектування інформаційної системи для роботи з клієнтами необхідно впевнитися що дані надійно захищені, наприклад, використовуючи хешування чутливих даних такі як паролі, біометричні дані, ключі, тощо.

Отримані від клієнта дані можна використати для покращення користувацького досвіду, розробки маркетингових стратегій і оптимізації розробки продукту. Також ці дані можна використати для попередньої обробки замовлення, такої як фільтрація замовлень від ненадійних клієнтів за допомогою попередньої їх перевірки людиною або алгоритмами інтелектуального аналізу даних.

Управління замовленнями безпосередньо впливає на сприйняття бізнесу або бренду клієнтом. У багатоканальному середовищі клієнти прагнуть отримати бездоганний досвіт взаємодії. Замовник може зробити замовлення онлайн з однієї платформи, але матиме запитання і завершить замовлення за допомогою кол-центру. В процесі виконання замовлення клієнт може використати додаткові канали для взаємодією з замовленням, такий як відслідковування його стану через електронну пошту або через месенджери. У разі виникнення проблем замовник матиме можливість виконати повернення коштів через фізичний канал, наприклад магазин. Кожна точка взаємодії має на меті забезпечити кращий досвід взаємодії з клієнтом та підвищити їх утримання, в наслідок чого збільшити дохід [3, 4].

Тож автоматизація обробки замовлень оптимізує управління ресурсами, спрощує взаємодії між клієнтами та виробниками, зменшує витрати та підвищує точність і надійність даних».

1.1.3 ERP системи

Система обробки замовлень може бути як окремою системою, так і компонентом більшої системи під назвою ERP (Enterprise Resource Planning).

ERP-система – пакетне програмне забезпечення для бізнесу, що дозволяє компанії «автоматизувати та інтегрувати більшість своїх бізнес-процесів; обмінюватися спільними даними та практиками в межах підприємства; створювати та отримувати доступ до інформації в режимі реального часу». (Самнер, 2014).

ERP-системи пов’язують між собою різні бізнес процеси та забезпечують передачу даних між ними. За допомогою збирання спільних даних з декількох джерел ці системи позбавляють дублювання і забезпечують єдність даних з єдиного

джерела. Ці системи мають центральну базу даних для всіх потоків інформації в організаціях, тим самим забезпечуючи збільшуючи гнучкість при зменшенні надлишковості. За допомогою веб технологій, ERP-системи мають можливість інтегрувати інформації не лише в середині організацій, але й з сторонніми партнерами, постачальниками чи клієнтами.

1.2 Огляд та аналіз наявних аналогів систем обробки замовлень

1.2.1 Приклади аналогів систем у світі

На момент 2024 року на ринку представлено різні інтелектуальні системи, що виконують обробку замовлень. Нижче перераховано декілька з них.

1.«IBM Sterling® Order Management» – це інтелектуальна система виконання замовлень, яка спрощує інтеграцію комплексних рішень багатоканальних процесів виконання замовлень, включно з керуваннями запасів та складами у реальному часі, доставку з магазину та купівлю онлайн (рисунок 1.1). Ця платформа пропонує інтуїтивно зрозумілий інтерфейс із доступними функціями та сповіщеннями. Також вона розширює можливості за допомогою інтеграції з різними доповненнями такими як:

– «IBM Sterling® Call Center»;
– «IBM Sterling® Store Engagement»;
– «IBM Sterling® Order Management Supply Chain Resiliency».

Кожне доповнення розроблено для інтеграції з основною системою, оптимізуючи керування електронною комерцією та роботу центрів виконання. «IBM Sterling® Order Management» дає змогу об’єднати усі канали продажів в одній платформі виконання, за допомогою якої можна точно відслідковувати рівень запасів, координувати логістику від сторонніх виконавців та організовувати замовлення, забезпечувати різні варіанти транспортування і керування поверненнями товарів, мінімізуючи витрати на транспортування. За допомогою комбінації цих інструментів можливо перетворити платформу електронної

комерції розумнішими, перетворюючи бізнес на компанію з виконання замовлень [5].

Рисунок 1.1 – Логотип «IBM Sterling® Order Management»

2. «Dynamics 365 Intelligent Order Management» – це система обробки замовлень від компанії Microsoft як автоматизує процес виконання замовлень за допомогою систем з інтегрованим штучним інтелектом та машинним навчанням які використовують дані про запаси, які надходять з різних каналів (рисунок 1.2). Ця система прискорює адаптацію і сприяє своєчасному та ефективному виконанню замовлень. З використанням даної системи у компаній є можливість централізовано керувати усім циклом замовлення від його прийому до виконання [6].

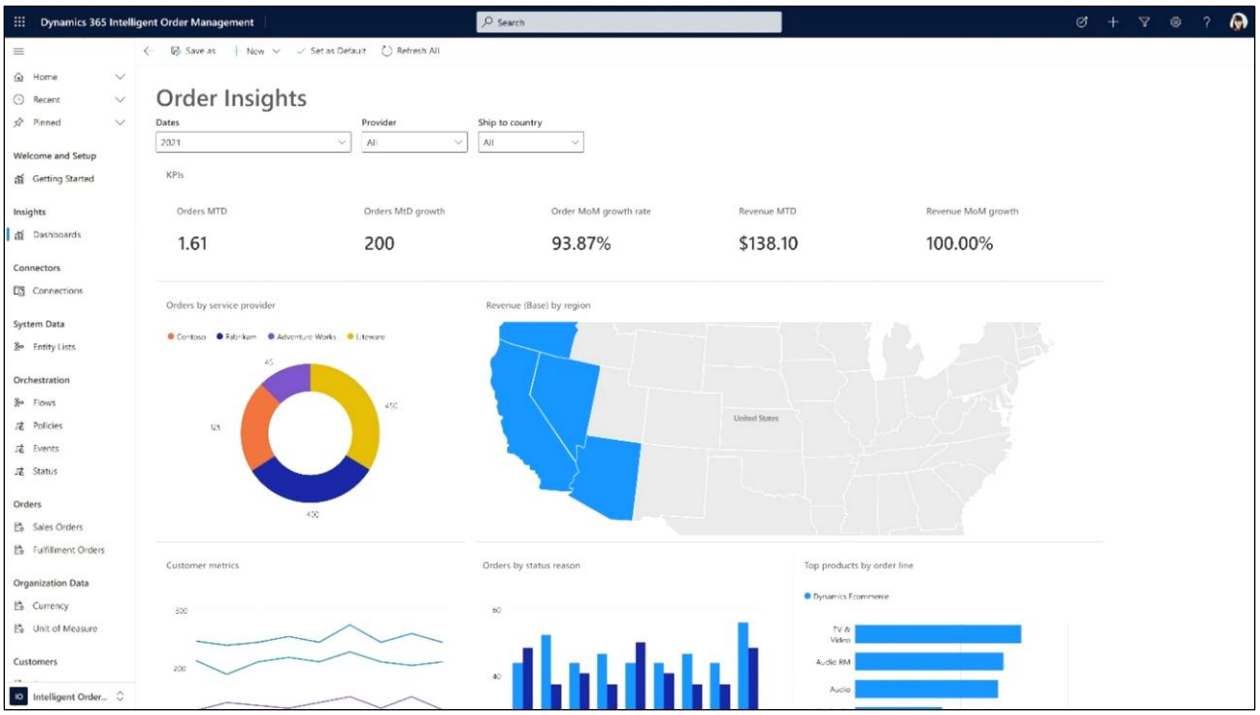

Рисунок 1.2 – Вигляд системи керування замовленнями «Dynamics 365 Intelligent Order Management»

3. «NetSuite Order Management» – це система керування життєвим циклом замовлення від компанії NetSuite, яка надає послуги доступу до застосунків для середнього бізнесу за принципом SaaS – Software as a Service (рисунок 1.3). Основні класи компонентів цієї компанії:

– CRM – керування взаємовідношеннями з клієнтами;
– ERP – керування ресурсами підприємства;
– E-commerce – електронна комерція;
– PSA – керування проєктами.

Система «NetSuite Order Management» є частиною компоненту ERP. Ця система автоматизує та керує життєвим циклом замовлень від моменту розміщення його користувачем до доставки у сервіс збуту. Вона забезпечує точне ведення записів, включаючи збір та перевірку замовлень, підтвердження їх відправлення та спілкування з клієнтами. Також система підтримує складні процеси такі як роздільна та прямі відправлення.

Система допомагає відстежувати та керувати потоками інформації зі всіх точок виконання замовлення що забезпечує високу швидкість та ефективність переміщення замовлень [7].

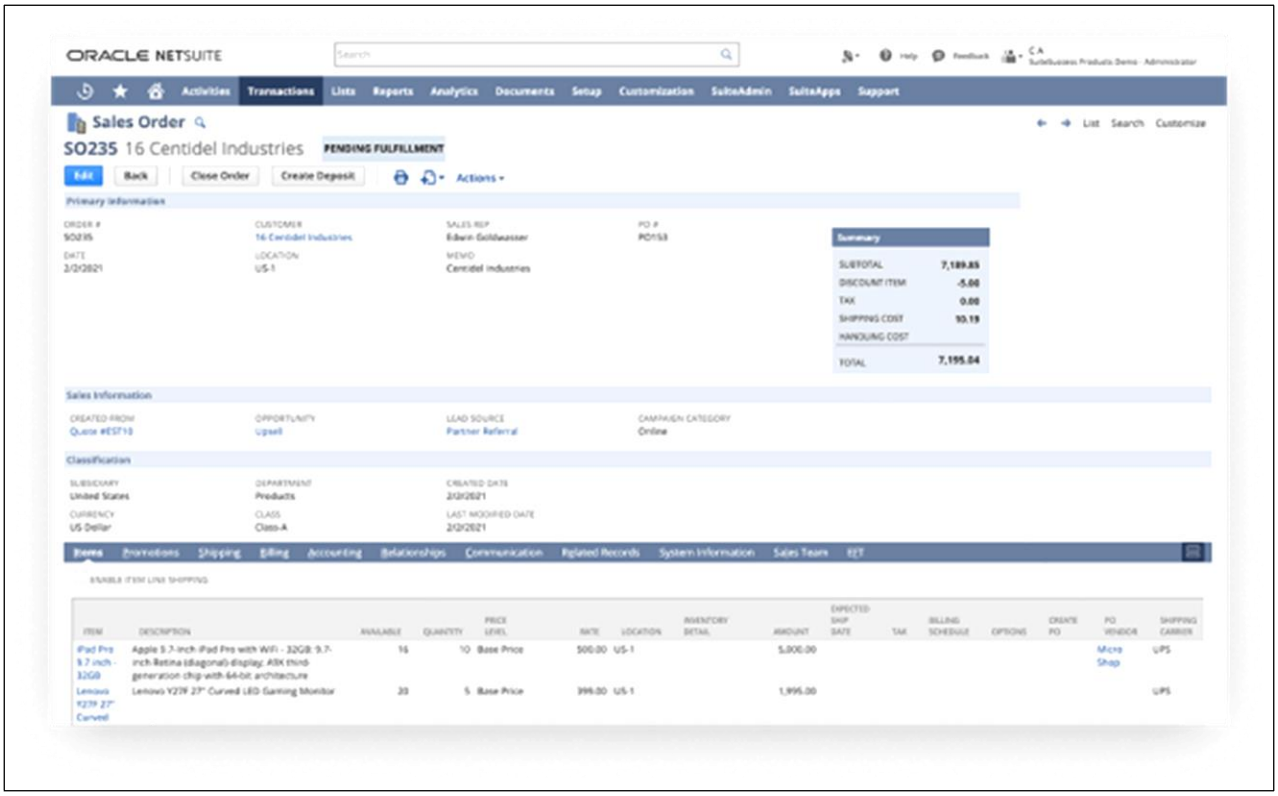

Рисунок 1.3 – Вигляд системи керування замовленнями «NetSuite Order Management

1.2.2 Порівняння систем обробки замовлень

Найбільш демонстративним способом надання порівняння буде подання у вигляді таблиці 1.1.

Для порівняння використаємо наступні критерії:

– характеристики системи;
– функціональність;
– можливість масштабування;
– інтеграція.

Таблиця 1.1 – Порівняльна характеристика наведених систем обробки замовлень

	IBM Sterling® Order Management	**Dynamics 365 Intelligent Order Management**	**NetSuite Order Management**
Характеристики системи	Інтегрована система виконання замовлень з різних каналів, включаючи керування запасами, доставку та електронну комерцію.	Система з інтегрованим штучним інтелектом та машинним навчанням для ефективного виконання замовлень.	Система для життєвого циклу замовлень з доступом через SaaS.
Функціональність	Керування запасами, складами, доставкою, електронна комерція, інтеграція з доповненнями.	Автоматизація процесів виконання замовлень за допомогою штучного інтелекту.	Керування життєвим циклом замовлень, включаючи взаємодію з клієнтами та складні відправлення.
Можливість масштабування	Високий рівень	Високий рівень	Високий рівень
Інтеграція	З доповненнями, такими як IBM Sterling® Call Center, Store Engagement, Order Management Supply Chain Resiliency.	Вбудована інтеграція з іншими рішеннями Microsoft.	Частина компоненту ERP компанії NetSuite.

1.3 Використання архітектури для побудови інформаційної системи

Для побудови програмної частини необхідно обрати ахітектуру. Для проектування подібних систем найчастіше використовують одну з двох архітектур: монолітну або мікросервісну (рисунок 1.4).

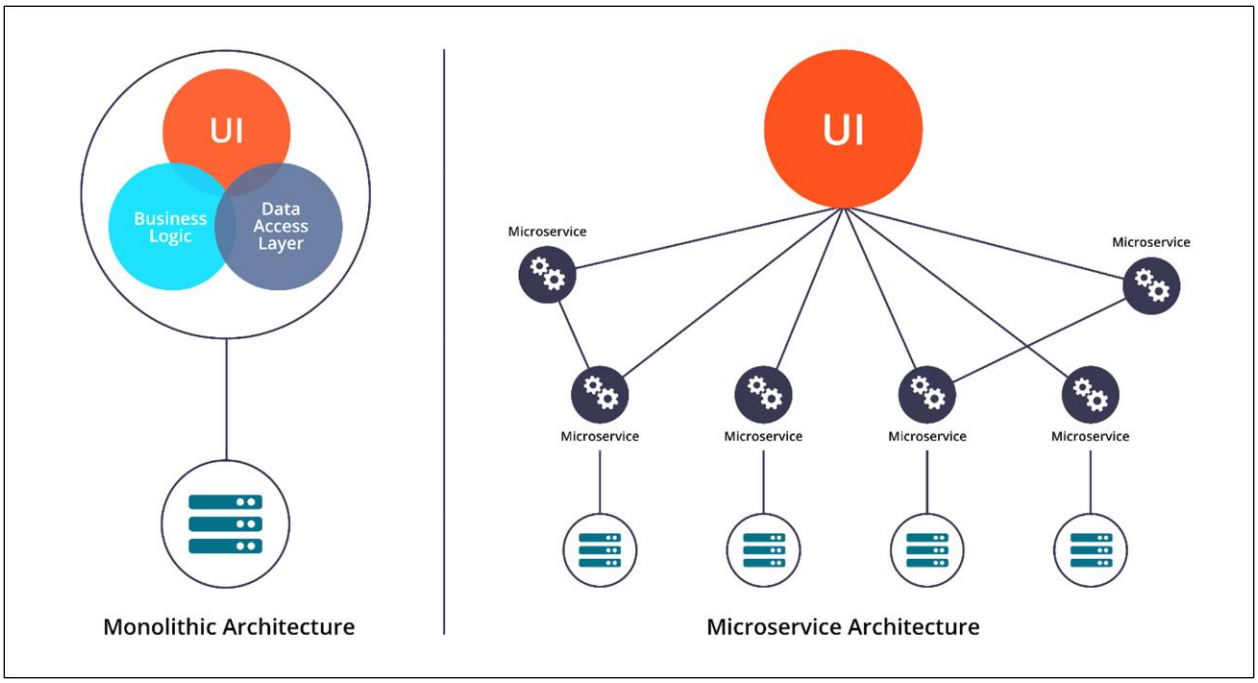

Рисунок 1.4 – Графічне порівняння монолітної та мікросервісної аріітектури

1.3.1 Визначення мікросервісної архітектури

Мікросервіси – це архітектурний стиль розробки програмного забезпечення, де застосунок будується як сукупність невеликих, незалежних сервісів, кожен з яких виконує певну окрему функцію. Одну одиницю часто називають мікросервісом. За допомогою сукупності цих сервісів будується одна, ціла система. Така система вирізняється децентралізованістю, може бути написаною з використанням різних мов програмування та використанням різних технологій для збереження даних. Ці сервіси спілкуються між собою через прості протоколи, такі як HTTP, MQTT або gRPC.

Для пояснення мікросервісної архітектури буде корисно надати визначення її альтернативи, монолітної архітектури.

1.3.2 Поняття монолітної архітектури

Програми, що написані з використанням монолітної архітектури побудовані як єдине ціле. Корпоративні програми часто складаються з трьох основних частин:

інтерфейс користувача на стороні клієнта (що складається з HTML-сторінок і JavaScript, запущених у браузері на комп'ютері користувача), база і програму на стороні сервера. Програма на стороні сервера оброблятиме запити HTTP, виконуватиме логіку домену, отримуватиме та оновлюватиме дані з бази даних, а також вибиратиме та заповнюватиме представлення HTML для надсилання в браузер. Ця серверна програма є монолітом – одним логічним виконуваним файлом. Будь-які зміни в системі передбачають створення та розгортання нової версії серверної програми.

Така архітектура є органічним підходом до створення подібних систем. Вся логіка обробки запитів може бути виконана у одному процесі. Це надає можливість використовувати усі функції мови програмування для розділення програми на складові, такі як структури, функції, класи, простори імен тощо [10,11].

1.4 Постановка задачі

Для розробки інформаційної системи обробки замовлень для інтернет-магазину необхідно використати певні технології та інструменти.

1.4.1 Планування деталей реалізації

Вибір однієї або декількох мов програмування такі як Java, JavaScript, Python та інші. Ці мови можуть використовуватися у поєднанні для написання якіснішої системи, так як кожна мова має свої переваги та недоліки. Для цього необхідно налагодити передачу повідомлень між програмами, що були написані на різних мовах.

Визначення фреймворків та бібліотек що будуть використані для створення системи. Наприклад, для мови Java найпопулярнішим фреймворком є Spring, для мови Python це Django або Flask, для JavaScript це React або Vue. Тож визначення правильних технологій є одною з ключових задач для розробки системи.

Проектування архітектури інформаційної системи обробки замовлень для інтернет-магазину. Існує велика кількість підходів до проектування систем. Одні з

основних це монолітні та мікросервісні архітектури, кожна з яких має переваги і недоліки, тож вибір залежить від технічного завдання розробки системи.

Розробка інтерфейсів програмних застосунків (API). Для організації обміну даних між застосунками використовують API – визначений набір методів для взаємодії між різними компонентами системи. Ці компоненти можуть використовувати різні протоколи та способи для передачі даних.

Вибір системи керування базами даних (СКБД). Існують реляційні та не реляційні бази даних. Реляційні СКБД мають перевагу в швидкості доступу та запису даних. До них відносять такі системи як MySQL MariaDB, PostgreSQL тощо. У той самий час такі системи важко масштабувати. Для вирішення цієї проблеми існують не реляційні СКБД, такі як MongoDB, Couchbase, eXist, MarkLogic та інші [9].

Важливим завданням є розробка правильної структури таблиць для збереження даних. Це дозволить зменшити надлишковість даних в системі що зможе покращити швидкість та надійність системи.

Вибір інтегрованого середовища розробки (integrated development environment або IDE) є також важливим елементом в процесі розробки програмних застосунків. Ці середовища найчастіше складаються з редактора коду, та інструментів для налагодження та компіляції написаних програм. Ці системи можуть підтримувати одну або більше мов програмування. Найвідоміші середовища розробки наразі це «Visual Studio», «Eclipse», різні системи від компанії «JetBrains», такі як «IntelliJ IDEA» для Java розробки, «PyCharm» для розробки на Python, «WebStorm» для розробки веб застосунків на JavaScript тощо.

1.4.2 Вимоги до програмної реалізації системи

Спираючись на план що до розробки системи можна визначити загальну структуру застосунку.

1. Систему буде створено з використанням мікросервісної архітектури. Зазвичай така архітектура складніша за монолітну, де всі компоненти працюють

разом в одному місці, але має суттєві переваги для створення системи обробки замовлень. За допомогою мікросервісної архітектури можливо налагодити просту взаємодію між частинами системи, так званими «мікросервісами» використовуючи RESTful веб-сервіси.

2. Для розробки сервісів буде використано декілька мов програмування, а саме Java, JavaScript та Python та відповідні фреймворки: Spring, Vue та Aiohttp. Це забезпечить найбільш ефективну розробку, тому що кожна мова програмування має свої сильні та слабкі сторони.

3. Кожен сервіс матиме свою окрему базу даних. Це дозволить обирати яку саме базу даних використовувати у мікросервісах, реаляційну чи не реляційну.

Висновки до розділу 1

В цьому розділі було проаналізовано предметну сферу, а саме знайдено публікаці та програмні реалізації що до створення систем для обробки замовлення, розглянуто переваги і недоліки існуючих систем та отримано загальне розуміння що до структури майбутньої інформаційної системи. Було з’ясовано що використання мікросервісної архітектури найкраще підходить для розробки системи, яка зазвичай є складової більшої системи, такої як ERP.

Наявні рішення здебільшого можна використовувати для великих підприємств. Тож є доцільним розробити окрему систему що буде націленою на середній та малий бізнес. У подальшому така система може бути розширена іншими інструментами для розширення області її застосування.

Основними вимогами до фреймворків та мов програмування є забезпечення найнадійніших систем, готових реалізацій та високого рівня підтримки. Це необхідно зменшення ймовірності отримання помилок при розробці, та у разі їх отримання, пришвидшити їх вирішення.

На основі проведеного аналізу можна зробити висновок, що для створення конкурентоспроможної інформаційної системи обробки замовлень необхідно

об’єднати найкращі існуючі практики і спрямовувати розробку на ті ніші, які ще не були зайняти у повній мірі.

2 АРХІТЕКТУРНІ ПІДХОДИ, ПАТТЕРНИ ПРОЕКТУВАННЯ ТА ТЕХНОЛОГІЇ, ЩО ВИКОРИСТОВУЮТЬСЯ ДЛЯ СТВОРЕННЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ ОБРОБКИ ЗАМОВЛЕНЬ ДЛЯ ІНТЕРНЕТ-МАГАЗИНУ

2.1 Java

Java – це широковикористовувана об’єктно-орієнтована мова програмування та програмна платформа, яка працює на мільярдах пристроїв, включаючи ноутбуки, мобільні пристрої, ігрові консолі, медичні пристрої та багато інших. Java заснована на мовах C та C++ [12].

Java – це технологія, яка використовує як мову програмування, так і програмну платформу. Щоб створити застосунок на Java, необхідно завантажити пакет Java Development Kit (JDK), який доступний для Windows, macOS і Linux. В процесі розолюкт програм мовою Java, компілятор перетворює її у Java байт-код. Це набір інструкцій для віртуальної машини Java (JVM), яка є частиною середовища виконання Java (JRE). Байт-код Java працює без змін у системах, які підтримують JVM, тому можна запускати його де завгодно.

Програмна платформа складається з JVM, Java API та середовища розробки. JVM аналізує та виконує або інтерпретує байт-код Java. Java API містить велику бібліотеку, що включає основні об'єкти, мережеві функції та функції безпеки, генерацію розширюваної мови розмітки (XML) та веб-сервіси. Java та її програмна платформа створюють потужні та перевірені технології для розробки корпоративного програмного забезпечення.

Основна філософія, що лежить в основі його створення – взаємодія між різними пристроями – є найвагомішим аргументом на користь Java для нових корпоративних застосунків. Об'єктно-орієнтована архітектура Java дозволяє писати модульні програми, скорочуючи цикл розробки.

Масштабованість платформи є ключовим атрибутом Java. Вона дозволяє використовувати єдину систему для широкого спектру застосувань. Існуючі

настільні програми можуть бути легко адаптовані для роботи на невеликих пристроях з обмеженими ресурсами. Їх можна переносити зі мобільного пристрою на робочий стіл, розробити бізнес-програми для платформи Android, а потім використовувати їх на поточному робочому столі [13].

Станом на 2024 рік, останньою версією є Java22. Java8, 11, 17 і 21 були попередніми версіями lts і досі офіційно підтримуються.

2.2 Spring Framework

Spring Framework – це фреймворк для створення корпоративних застосунків на Java. Він забезпечує всеохоплюючу інфраструктуру для розробки Java-застосунків, полегшуючи управління конфігурацією, життєвим циклом об'єктів, а також забезпечуючи інтеграцію з іншими популярними технологіями та фреймворками [14].

Spring – це багаторівнева архітектура, тому щоразу, коли система електронної комерції розробляється з використанням Spring, вона чітка поділ шарів. Через свою багатошарову архітектуру це дозволяє користувачам обирати, які з його компонентів користувачі можуть використовувати.

2.2.1 Архітектура Spring Framework

Далі перелічено основні компоненти що входять до складу Spring Framework (рисунок 2.1).

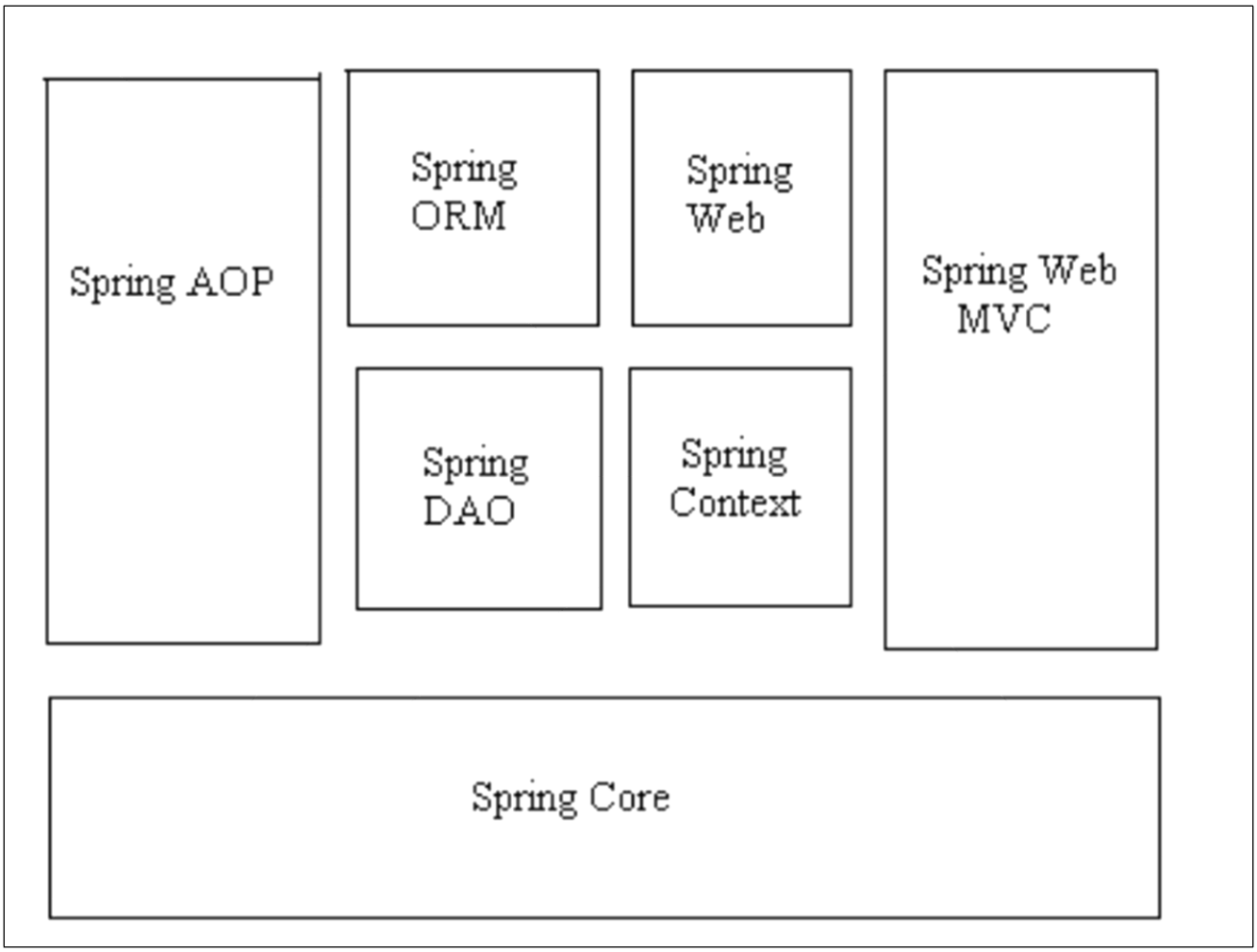

Рисунок 2.1 – Схема архітектури Spring Framework

1. **Spring Core** є основним модулем фреймворка Spring. Він включає базові функції, які є основою для інших модулів. Від містить наступні компоненти:

– IoC Container (Inversion of Control). Контейнер Spring відповідає за управління життєвим циклом об'єктів та їх залежностями. Він реалізує принцип Inversion of Control, де контроль за створенням об'єктів передається контейнеру;

– Dependency Injection (DI). Механізм впровадження залежностей дозволяє автоматично налаштовувати залежності між об'єктами, що полегшує тестування та зменшує зв'язність коду.

2. **Spring Context** розширює можливості модуля Core, забезпечуючи більш високий рівень функціональності та інтеграції. До нього входять наступні компоненти:

– ApplicationContext. Це розширений контейнер IoC, який додає можливості подій, інтернаціоналізації (i18n), та інтеграцію з іншими фреймворками. Він є основним середовищем виконання для Spring-застосунків;

– Enterprise Services: Підтримка JNDI, EJB, JMS та інших корпоративних служб.

3. **Spring AOP** (Aspect-Oriented Programming) забезпечує підтримку аспектно-орієнтованого програмування, що дозволяє розділяти поперечні аспекти (cross-cutting concerns), такі як логування, безпека, транзакції.

4. **Spring DAO** (Data Access Object) модуль спрощує доступ до баз даних, забезпечуючи абстракцію над різними технологіями доступу до даних.

5. **Spring ORM** (Object-Relational Mapping) модуль забезпечує інтеграцію з різними ORM фреймворками, такими як Hibernate, JPA (Java Persistence API), JDO (Java Data Objects). Компонент HibernateTemplate спрощує використання Hibernate, забезпечуючи шаблони для виконання основних операцій, а JpaTemplate полегшує використання JPA, інтегруючись з EntityManager та іншими компонентами JPA.

6. **Spring Web** модуль забезпечує базові функції для створення веб-застосунків, включаючи інтеграцію з іншими веб-технологіями. Він містить компонент WebApplicationContext, що розширює ApplicationContext для веб-застосунків, забезпечуючи інтеграцію з веб-контейнерами, такими як сервлети.

7. **Spring Web MVC** (Model-View-Controller) модуль забезпечує підтримку шаблону проектування MVC для розробки веб-застосунків.

2.3 Spring Boot

Spring Boot – це інструмент для спрощення та прискорення розробки застосунків на базі Spring Framework. Він забезпечує швидкий старт проекту, уникнення рутинної конфігурації та інтеграцію з базовим Spring Framework [15]. Основна мета Spring Boot полягає у забезпеченні простого та ефективного способу розробки застосунків шляхом усунення потреби у ручній настройці завдяки

стандартним конфігураціям та автоматичній настройці, що сприяє зручності та швидкості розробки.

Spring Boot має декілька ключових властивостей.

1. Надання автоматичного налаштування застосунків, що базується на структурі та залежностях проекта.

2. Наявність вбудованого серверу для розгортання застосунків. Це позбавляє потреби у налаштуванні окремого сервера.

3. Спрощення управління та додавання залежностей застосунку.

Керування залежностями є критичною складовою будь якого складного проекту, в особливості такого як мікросервісна система. Проведення усіх необхідних налаштувань самотужки не є дуже хорошою практикою, бо чим більше часу буде витрачено на ці дії, тим менше залишається на безпосередню розробку системи. Для вирішення цієї проблеми використовують так звані «стартери» (**Spring Boot starters**) що зображені на рисунку 2.2 [16]. Це набори зручних дескрипторів залежностей, що можуть бути додані до програми. Ці набори надають необхідні бібліотеки для Spring Framework та пов’язаних з ним технологій, що позбавляють потреби переглядати інші приклади коду і прописувати безліч дескрипторів залежностей (рисунок 2.3).

```
dependencies { Edit Starters...
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    implementation 'org.springframework.boot:spring-boot-starter-data-rest'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'org.springframework.boot:spring-boot-starter-web-services'
    implementation 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client'
    implementation 'org.springframework.cloud:spring-cloud-starter-config'

    implementation 'org.mapstruct:mapstruct:1.5.5.Final'
    implementation 'org.mapstruct:mapstruct-processor:1.5.5.Final'

    compileOnly 'org.projectlombok:lombok'
    developmentOnly 'org.springframework.boot:spring-boot-devtools'
    runtimeOnly 'org.postgresql:postgresql'
    annotationProcessor 'org.projectlombok:lombok'
    annotationProcessor 'org.mapstruct:mapstruct-processor:1.5.5.Final'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'
}
```

Рисунок 2.2 – Залежності застосунку зі стартерами

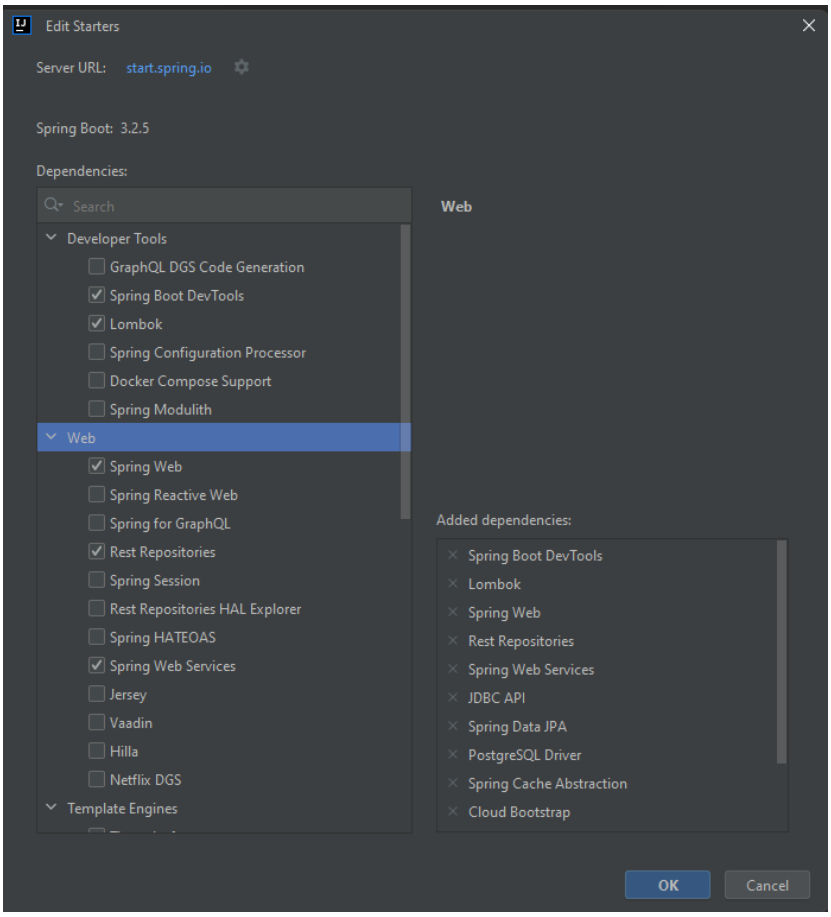

Рисунок 2.3 – Меню керування залежностями у середовищі розробки «Intellij Idea»

2.4 Spring Data

Для збереження даних у застосунках на базі Spring Framework найчастіше використовують Spring Data.

Spring Data – це проект, що спрощує взаємодію з базами даних. Він надає абстракцію над різними технологіями доступу до даних, такими як JPA, MongoDB, Redis тощо. Основна задача цих абстракцій – надання єдиного інтерфейсу для роботи з різними базами даних. Це дозволяє змінювати бази даних у налаштуваннях не змінюючи код. Завдяки цьому збільшується обсяг повторюваного коду, що спрощує реалізації репозиторіїв, що відповідають за доступ до даних.

2.4.1 Об'єктно-реляційне відображення

Об'єктно-реляційним відображенням, також відомим як ORM (Object-Relational Mapping), називають процес перетворення об’єктів парадигми об’єктно орієнтованого програмування у записи в таблицях баз даних. Вона надає можливість взаємодіяти з базами даних без використання мови запитів, наприклад SQL.

В мові програмування Java найбільш популярною реалізацією ORM є Hibernate.

2.4.2 Hibernate

Hibernate – це ORM фреймворк, який полегшує роботу з реляційними базами даних шляхом відображення об'єктно-орієнтованої моделі на реляційну модель бази даних [17]. Основні можливості Hibernate:

– автоматичне відображення об'єктів Java на таблиці баз даних;
– підтримка різних стратегій кешування для оптимізації продуктивності;
– засоби для виконання запитів і маніпуляцій даними за допомогою HQL (Hibernate Query Language) та Criteria API;
– автоматичне керування транзакціями;
– Hibernate використовують у Spring Framework проектах як реалізацію специфікації Spring Data JPA.

2.4.3 Spring JPA

Spring Data JPA – це підпроєкт в рамках Spring Data, який спеціалізується на роботі з JPA (Java Persistence API). JPA – це специфікація Java EE (Java Platform, Enterprise Edition) для управління реляційними даними в Java застосунках [18]. Spring Data JPA робить простішим використання JPA завдяки наданню додаткових можливостей, таких як:

– спрощення створення репозиторіїв та таблиць (сутностей) за допомогою інтерфейсів та абстракцій;

– підтримка автоматичного створення запитів на основі імен методів;

– інтеграція з іншими частинами Spring Framework, такими як Spring MVC і Spring Security;

– підтримка спеціалізованих запитів через JPQL (Java Persistence Query Language) та нативні SQL-запити.

2.4.3.1 JPA Entity

JPA Entity – це об'єкт, що відображає структуру даних в базі даних, а також зв'язки між цими даними, в об'єктно-орієнтованій програмі. Кожен JPA Entity відповідає таблиці в базі даних і може бути збережений, витягнутий, видалений та оновлений за допомогою JPA.

Для створення JPA Entity використовують звичайні класи Java, до яких застосовують відповідну анотацію **@Entity**. На рисунку 2.4 продемонстровано використання анотацій з бібліотеки **jakarta persistance**. Всередині класу використовують анотації **@Column** для позначення назв колонок у майбутній таблиці. Для позначення унікального ключа в реляційній таблиці використовують анотацію **@Id** та **@GeneratedValue** зі вказанням стратегії для автиоматичної генерації ключа під час створення запису.

Відповідна таблиця в базі даних продемонстровано на рисунку 2.5.

```java
package up.pp.atomax.catalog.entities;
import jakarta.persistence.*;
import lombok.*;


@Getter  7 usages  new *
@Setter
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Entity
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)
    private Long id;
    @Column
    private String name;
    @Column
    private String description;
    @Column
    private String manufacturer;

    @Column
    @Enumerated(EnumType.STRING)
    private ProductStatus status;

    @Column
    private Double price;

}
```

Рисунок 2.4 – Використання анотації Entity

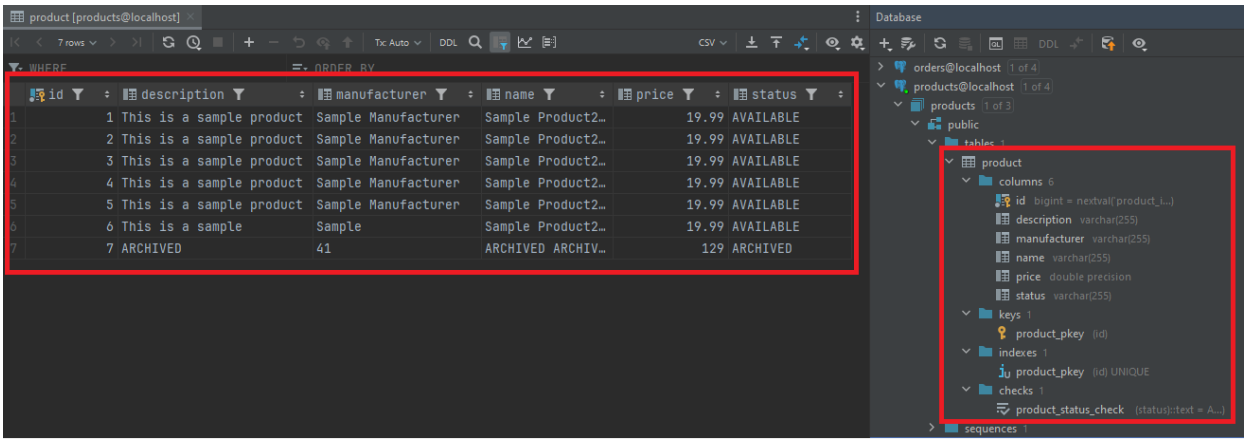

Рисунок 2.5 – Створена відповідна таблиця в реляційній базі даних

2.4.3.2 JPA Repository

Для доступу до даних, можна використовувати **JPA Repository**.

JPA Repository – це інтерфейс, який надає спрощений спосіб взаємодії з базою даних для класів що позначені анотацією **@Entity**. Він надає зручні методи для виконання різних операцій з даними, таких як CRUD (скорочення від «Create, Read, Update, Delete») та деякі інші [19]. JPA Repository є об'єднанням декількох інтерфейсів що зображені на рисунку 2.6.

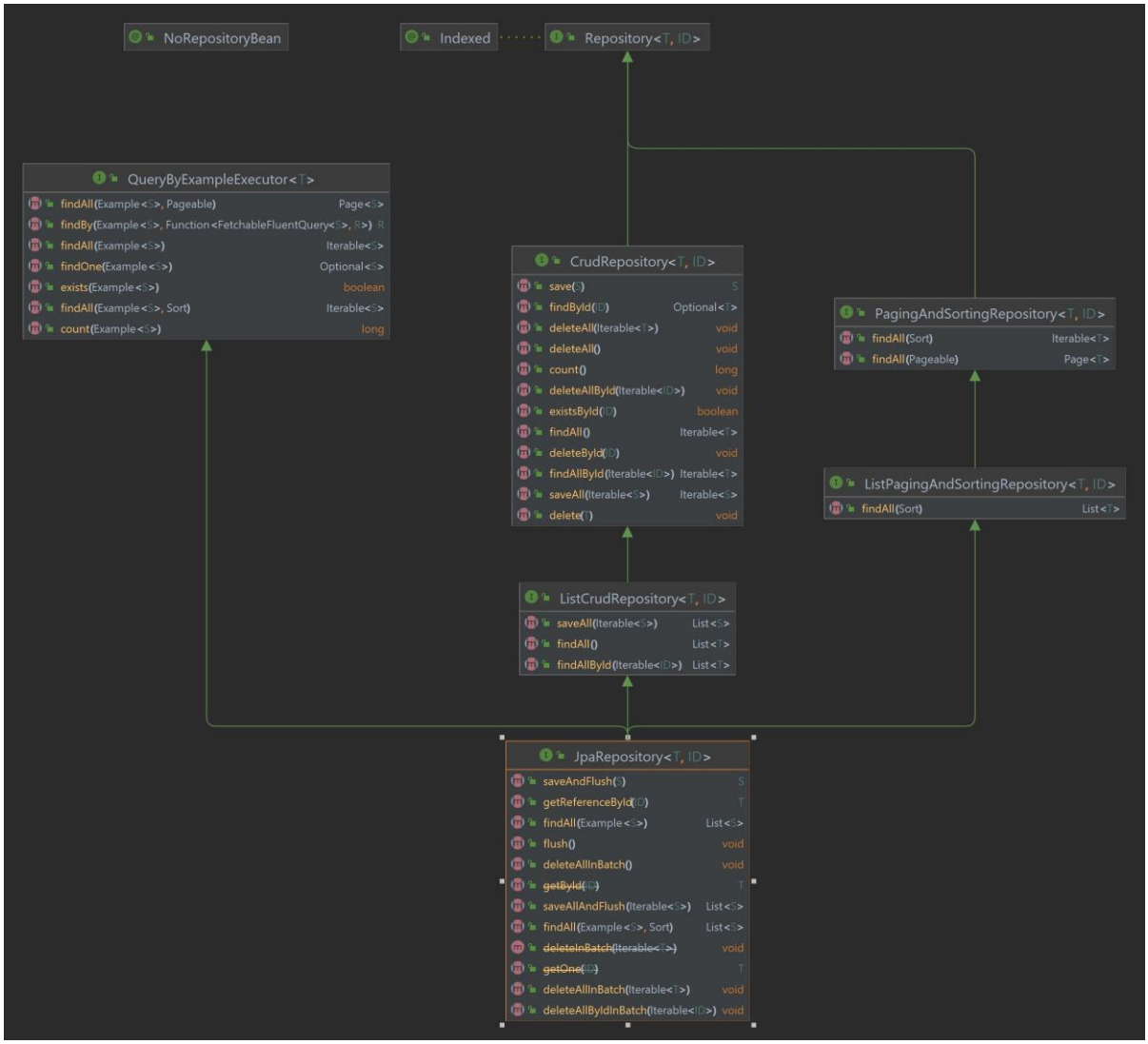

Рисунок 2.6 – Ієрархія інтерфейсів які розширює JPA Repository

Основними перевагами використання JPA Repository є:

– **спрощення доступу до даних**. JPA Repository надає методи для збереження, оновлення, видалення та витягнення даних з бази даних без необхідності написання SQL-запитів вручну;

– **автоматичне створення запитів**. Spring Data JPA може автоматично генерувати SQL-запити на основі імен методів в інтерфейсі репозиторію. Це дозволяє писати менше коду, оскільки багато загальних запитів можуть бути автоматично згенеровані;

– **підтримка власних запитів**. У JPA Repository є можливість визначити власні методи з використанням анотацій Query або @Query;

– **рефакторинг коду**. Використання інтерфейсів репозиторіїв дозволяє зосередитися на бізнес-логіці застосунку та зменшити кількість коду для доступу до даних;

– **підтримка транзакцій**. JPA Repository автоматично управляє транзакціями, забезпечуючи атомарність та цілісність операцій.

2.5 Об’єктно-реляційна база даних. PostgreSQL

Для збереження даних у застосунку було обрано об’єктно-реляційну базу даних (або ORD – object-relational database).

Об’єктно-реляційна база даних – це система керування керування базами даних, що поєднує одночасно реляційну модель та об’єктно-орієнтовану модель бази даних. ORD має підтримку основних компонентів будь-якої об’єктно-орієнтованої моделі в її схемах, а також компонентів мови запитів, таких як об’єкти, класи та успадкування [21].

ORD є посередником між реляційними та об’єктно-орієнтованими базами даних, оскільки він містить аспекти та характеристики обох моделей. В ORD основний підхід базується на реляційній базі даних, а доступ до даних здійснюється за допомогою запитів, написаних мовою запитів, наприклад SQL. Однак ORD має характеристику об’єктної-орієнтованості, оскільки база даних вважається сховищем об’єктів, як правило, для програмного забезпечення, написаного на об’єктно-орієнтованій мові програмування.

Одна з основних задач ORD є закриття прогалини між методами концептуального моделювання даних для реляційних та об’єктно орієнтованих баз

даних, таких як об’єктно-реляційне відображення (ORM) та діаграми зв'язків сутностей (ERD – entity-relationship diagram).

2.5.1 Використання PostgreSQL у Spring застосунку

Прикладом об’єктно-реляційної системи керування базами даних (СКБД) є PostgreSQL. Це база даних з відкритим кодом, що підтримує широкий спектр функцій, таких як транзакції, підзапити, складні запити, зберігання процедур, і багато іншого [22].

Для використання PostgreSQL у Spring застосунку необхідно виконати наступні кроки.

1. Налаштувати залежності. На рисунку 2.7 продемонстровано залежність в системі Gradle.

```
dependencies { Edit Starters...
    implementation 'org.springframework.boot:spring-boot-starter-data-rest'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'org.springframework.cloud:spring-cloud-starter-config'
    implementation 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client'
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'

    compileOnly 'org.projectlombok:lombok'
    developmentOnly 'org.springframework.boot:spring-boot-devtools'
    annotationProcessor 'org.springframework.boot:spring-boot-configuration-processor'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    runtimeOnly 'org.postgresql:postgresql'
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'
}
```

Рисунок 2.7 – Додання залежності для PostgreSQL до Spring застосунку

2. Налаштувати підключення до бази даних у файлі налаштувань застосунку такому як «application.yml», що знаходиться у директорії з ресурсами (рисунок 2.8).

```yaml
spring:
  datasource:
    driver-class-name: org.postgresql.Driver
    password: admin
    url: jdbc:postgresql://localhost:5432/products
    username: postgres
  jpa:
    hibernate:
      ddl-auto: update
      naming:
        physical-strategy: org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
```

Рисунок 2.8 Налаштування підключення до бази даних PostgreSQL у Spring застосунку

2.6 Архітектурний шаблон MVC

Шаблон проектування MVC – це один з найбільш популярних шаблонів для проектування застосунків що взаємодіють з користувачем [23]. Він розділяє застосунок на 3 модулі: модель, відображення та контролер (рисунок 2.9).

1. Модель – призначена для взаємодії з базами даних чи іншими структурами для збереження даних. Ця частина застосунку відповідає за виконання бізнес логіки та забезпечення збереження даних у відповідному форматі що надходять від контролера.

2. Відображення – перетворює отримані данні у форму, що буде зручною користувачу. Це може бути html сторінка у браузері, рендеринг зображення на пристрої, консольний вивід, повідомлення у чат-боті, тощо.

3. Контролер – отримує дані від користувача і викликає відповідні моделі для обробки їх обробки. Контролер відповідає за зв’язок між командою від користувача та моделлю, що може її обробити. Прикладом такого контролеру може бути Spring Controller із Spring Web MVC.

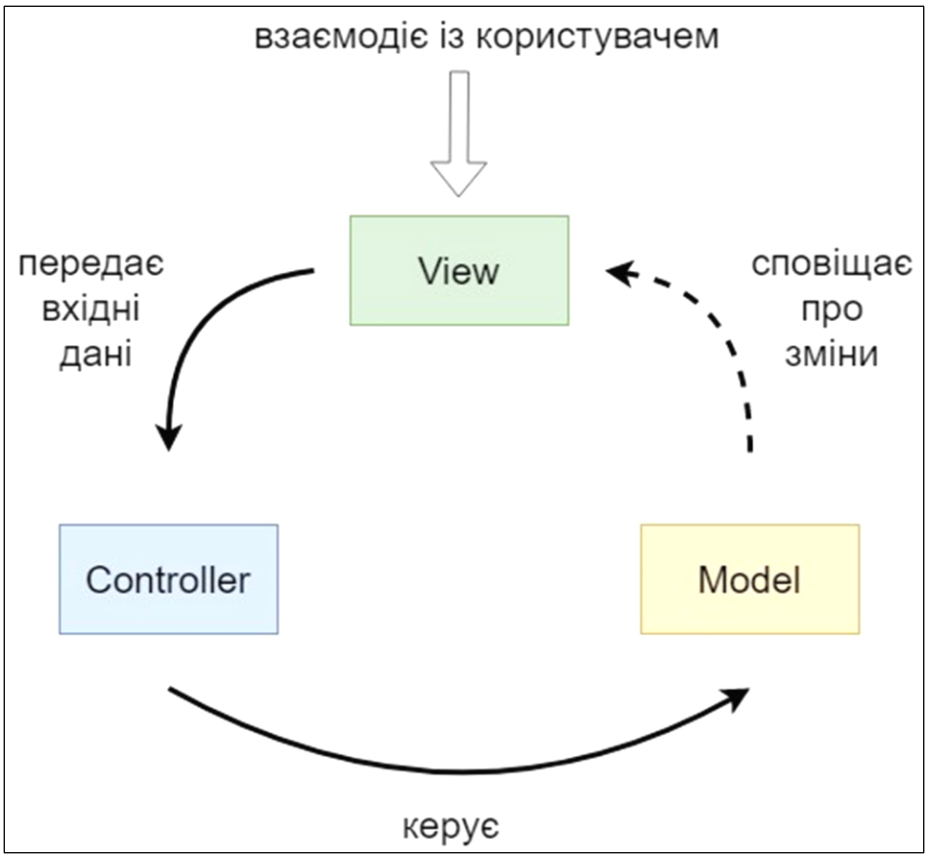

Рисунок 2.9 – Схема шаблону MVC

2.7 Spring Web MVC

Для створення застосунку, з яким можна взаємодіяти за допомогою http запитів у застосунках на Spring Framework використовують Spring Web MVC [24].

Фреймворк Spring Web MVC розроблено навколо DispatcherServlet (рисунок 2.10), який надсилає запити до обробників із відображеннями обробників що налаштовуються, роздільною здатністю перегляду сторінки, локалізацією та графічною темою сторінки, надає підтримку для завантаження файлів. Обробники базуються на анотаціях @Controller і @RequestMapping, надаючи великий набір методів для обробки запитів від клієнта.

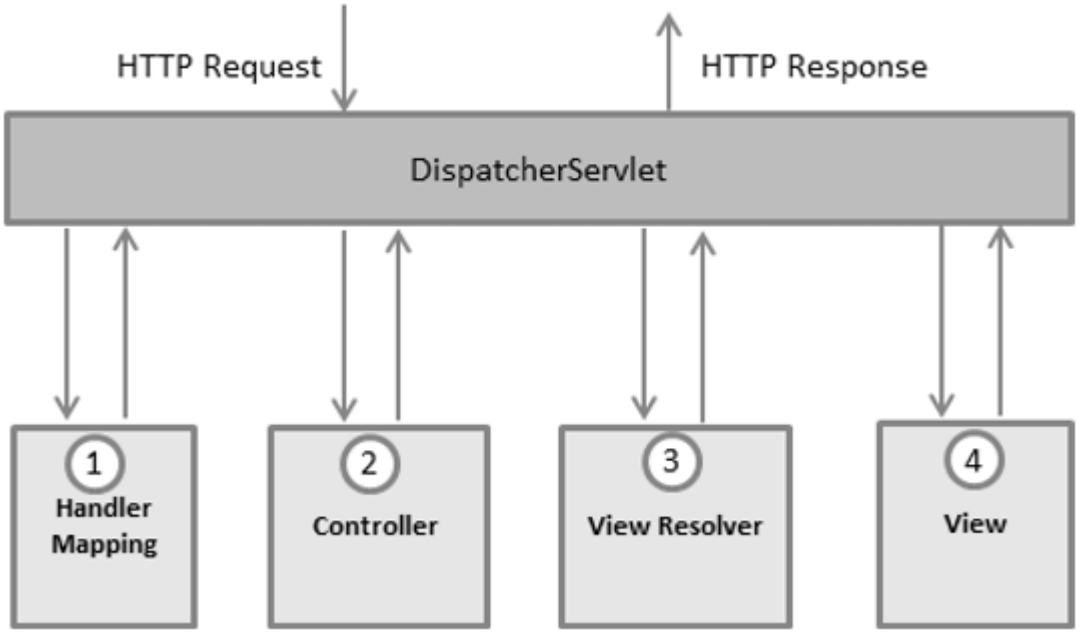

Рисунок 2.10 – Структура Spring MVC

2.7.2 Controller

DispatcherServlet надсилає запити до відповідних класів-контролерів. Для позначення контролеру використовують анотацію @Controller. Разом з нею також додають іншу анотацію під назвою @RequestMapping. Вона потрібна для зв’язування з URL адресою. Ця анотація може бути застосована як до методу, після чого він буде викликатися для обробки вказаної адреси, так і до всього класу, додаючи загальну адресу для усіх методів всередині контролеру.

На рисунку 2.11 зображено контролер з використанням анотацій, зазначеними вище, а також деякими іншими що розширюють попередні анотації додаючи додаткову фільтрацію для запитів.

```java
@AllArgsConstructor  new *
@RequestMapping()
@RestController
public class CartController {

    CartService cartService;
    private final CartMapper cartMapper;


    @GetMapping("/")  new *
    ResponseEntity<?> getShoppingCart(@RequestHeader("X-username") String username) {
        return ResponseEntity.ok().body(cartMapper.toDto(cartService.getCartFromUsername(username)));
    }

    @DeleteMapping("/clear")  new *
    ResponseEntity<?> clearShoppingCart(@RequestHeader("X-username") String username) {
        cartService.clearCart(username);
        return ResponseEntity.status(HttpStatus.NO_CONTENT).build();
    }

    @PutMapping("/")  new *
    public ResponseEntity<String> updateCart(@RequestHeader("X-username") String username, @RequestBody CartDto updatedCartDto) {
        cartService.updateCart(username, updatedCartDto);
        return ResponseEntity.status(HttpStatus.NO_CONTENT).build();

    }

    @GetMapping("/test")  new *
    String test() { return "test"; }

}
```

Рисунок 2.11 – Використання анотацій для створення класу контролера

2.8 Spring Security

В сучасних застосунках що працюють на рівні підприємств важко уявити такі, що не використовують ніяких інструментів для гарантування інформаційної безпеки. Для захисту інформації від несанкціонованого доступу можна використати різні інструменти і технічні засоби. У Spring Framework за це відповідає Spring Security Framework [25].

Spring Security Framework – потужний та гнучкий фреймворк для аутентифікації та контролю за доступом до інформації. Він є стандартом, для забезпечення безпеки у Spring застосунках.

Головна задача цього фреймворку – одночасне надання можливості як для авторизації, так і для аутентифікації користувачів. Spring Security спроектований

таким чином, щоб його легко розширювати та змінювати. Цей фреймворк також надає додатковий захист проти різних атак, таких як CSRF, XSS тощо.

Spring Security підтримує систему фільтрації. Запит, що надходить від користувача повинен спочатку пройти через різні фільтри, так званий «ланцюг фільтрів» (FilterChain), перш ніж бути обробленим. Ці фільтри дають можливість додавати додаткову логіку для перевірки правильності запитів, а також їх модифікації (рисунок 2.12).

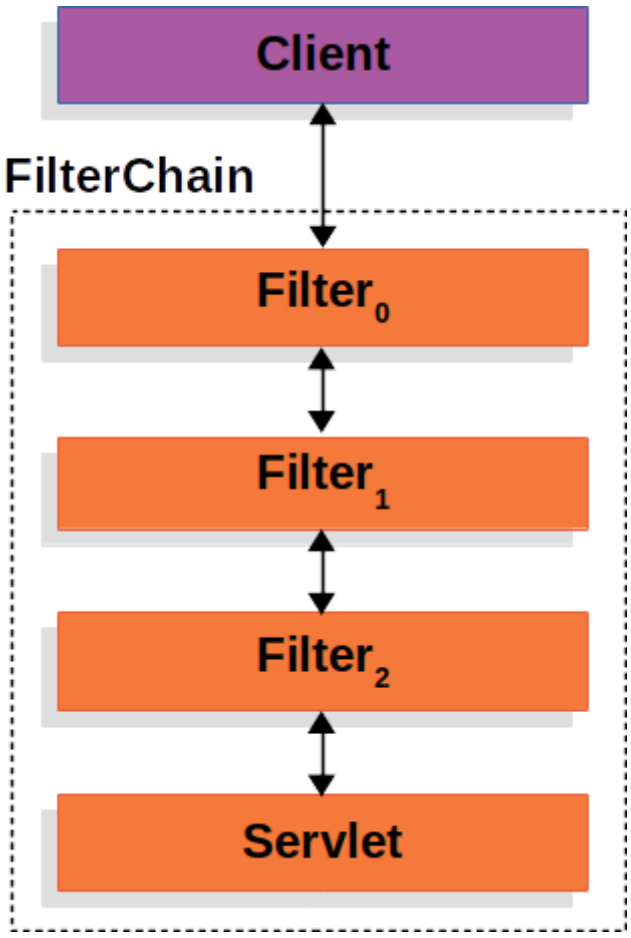

Рисунок 2.12 – FilterChain

2.9 Spring Cloud

Для створення застосунків з використанням мікросервісної архітектури найпопулярнішим рішенням є набір компонентів Spring Cloud. Він може бути

використаний для забезпечення масштабованості, надійності та гнучкості архітектури системи. [26]

Основними елементами з набору компонентів Spring Cloud для вирішення завдання КРБ будуть:

– Spring Cloud Config – дозволяє централізовано управляти конфігураціями мікросервісів. Може використовуватися для зберігання налаштувань у Git репозиторіях, тим самим надається можливість переглядати історію змін в налаштуваннях.

– Spring Cloud Netflix Eureka – сервіс виявлення, який дозволяє мікросервісам реєструватися у ньому та знаходити один одного. Його можна порівняти з реєстром для усіх мікросервісів.

– Spring Cloud Gateway – забезпечує маршрутизацію запитів на основі URL, заголовків, cookies тощо.

2.10 JavaScript

JavaScript – це високорівнева, динамічна, інтерпретована мова програмування, яка широко використовується для створення інтерактивних і динамічних веб-сторінок. Ця мова була створена для використанні у браузері на стороні клієнта. Але зараз її також використовують для написання серверної частини коду. Запускати JavaScript код можна, наприклад, за допомогою Node.js.

2.10.1 Node.js

Node.js – це міжплатформне середовище виконання JavaScript із відкритим кодом, яке підходить майже для будь-якого типу проектів. Node.js використовує рушій V8 JavaScript, основа Google Chrome, поза браузером, що сприяє його високій продуктивності.

Програма Node.js працює в одному процесі без створення нового потоку для кожного запиту. Він включає в себе набір асинхронних примітивів вводу-виводу у

своїй стандартній бібліотеці, які запобігають блокуванню коду JavaScript. Як правило, бібліотеки Node.js розроблені з використанням неблокуючих парадигм.

Коли Node.js виконує операції вводу-виводу, такі як мережевий зв'язок, доступ до бази даних або взаємодія з файловою системою, він не блокує потік, таким чином уникаючи марних циклів процесору. Натомість Node.js відновлює роботу після отримання відповіді.

Це дозволяє Node.js керувати тисячами одночасних з'єднань з одним сервером, усуваючи складність керування паралельними потоками.

Node.js пропонує значну перевагу, оскільки дозволяє мільйонам розробників, які використовують JavaScript для браузерів, також писати серверний код без необхідності вивчати нову мову.

2.10.2 Vue.js

Vue.js – це фреймворк JavaScript з відкритим вихідним кодом model–view–viewmodel (MVVM), розроблений для створення інтерфейсів користувача та односторінкових програм.

Vue.js має поступово адаптовану архітектуру, яка фокусується на декларативному рендерингу та композиції компонентів. Його основна бібліотека зосереджена виключно на рівні пердставлень. Для складніших додатків розширені функції, як-от маршрутизація, керування станом і інструменти побудови, які надаються за допомогою офіційно підтримуваних допоміжних бібліотек і пакетів. Vue.js дозволяє розширювати HTML за допомогою атрибутів HTML, відомих як директиви. Ці директиви, які можуть бути вбудованими або визначеними користувачем та створені для розширення функціональності HTML програмам.

Vue.js також надає реактивну систему зв'язування даних, яка оновлює представлення щоразу, коли змінюється модель, сприяючи покращенню процесу розробки. Цей фреймворк підтримує структуру на основі компонентів, що дозволяє створювати багаторазові компоненти. Це підвищує продуктивність і зручність обслуговування системи. Крім того, Vue.js має активну спільноту та велику

кількість ресурсів, включаючи високий рівень документації, навчальні посібники та плагіни сторонніх розробників, що робить його доступним і надійним вибором як для початківців, так і для досвідчених розробників.

2.11 Python

Python – це інтерпретована, динамічно типізована мова програмування високого рівня, створена Гвідо ван Россумом і вперше випущена в 1991 році. Вона широко використовується завдяки своїй читабельності, простоті синтаксису та великій стандартній бібліотеці [29].

2.11.1 Aiohttp

Aiohttp – це асинхронна бібліотека Python для роботи з HTTP-клієнтами та серверами. Вона дозволяє створювати високопродуктивні веб-програми, використовуючи асинхронне програмування (async/await).

Основні можливості Aiohttp:

- HTTP-клієнт для надсилання HTTP-запитів;
- HTTP-сервер для створення веб-серверів;
- підтримка для роботи з WebSocket-з'єднаннями;
- гнучка система маршрутизації для обробки різних URL-шляхів;
- підтримка сесій та роботи з кукі.

Aiohttp є потужною та гнучкою бібліотекою для створення асинхронних веб-застосунків у Python. Вона дозволяє ефективно обробляти великий обсяг запитів завдяки використанню асинхронного програмування. Незалежно від того, чи потрібен простий HTTP-клієнт, чи повноцінний веб-сервер з підтримкою WebSocket, Aiohttp забезпечить всі необхідні інструменти для розробки [30].

2.11.2 Aiogram

Aiogram – це асинхронна бібліотека Python для розробки ботів для Telegram, яка використовує асинхронні функції. Вона є дуже популярною завдяки своїй простоті та ефективності, забезпечуючи зручний інтерфейс для взаємодії з Telegram API [31].

Основні можливості Aiogram:

– асинхронність: використання async/await для покращення продуктивності;

– маршрутизація: зручна система для обробки повідомлень та команд;

– фільтри: можливість фільтрації вхідних повідомлень за різними критеріями;

– підтримка різних середовищ для зберігання даних під час роботи бота;

– налаштування та конфігурація бота за допомогою спеціальних інструментів.

Aiogram дозволяє швидко та ефективно створювати асинхронні боти. Використання асинхронного програмування значно покращує продуктивність бота, а зручний інтерфейс Aiogram робить процес розробки інтуїтивно зрозумілим.

Висновки до розділу 2

В цьому розділі було розглянуто технології, архітектурні підходи та паттерни проектування, які будуть використані під час розробки інформаційної системи обробки замовлень для інтернет-магазину, а саме технології та фреймворки для створення мікросервісної архітектури на Java та механізми збереження даних. Було розглянуто Spring Framework, як основу back-end застосунку, Vue.js як фреймворк для frontend серверу та мову Python з фреймворками aiogram та aiohttp для взаємодії з системою та месенджером «Telegram».

3 АРХІТЕКТУРНА ТА ПРОГРАМНА РЕАЛІЗАЦІЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ

3.1 Розробка архітектури мікросервісів

Для взаємодії мікросервісів було використано відправлення повідомлень через http. Цей спосіб є доволі популярним і гнучким, тож для цього можна використовувати існуючі бібліотеки.

Загальну структуру розробленого застосунку зображено на рисунку 3.1.

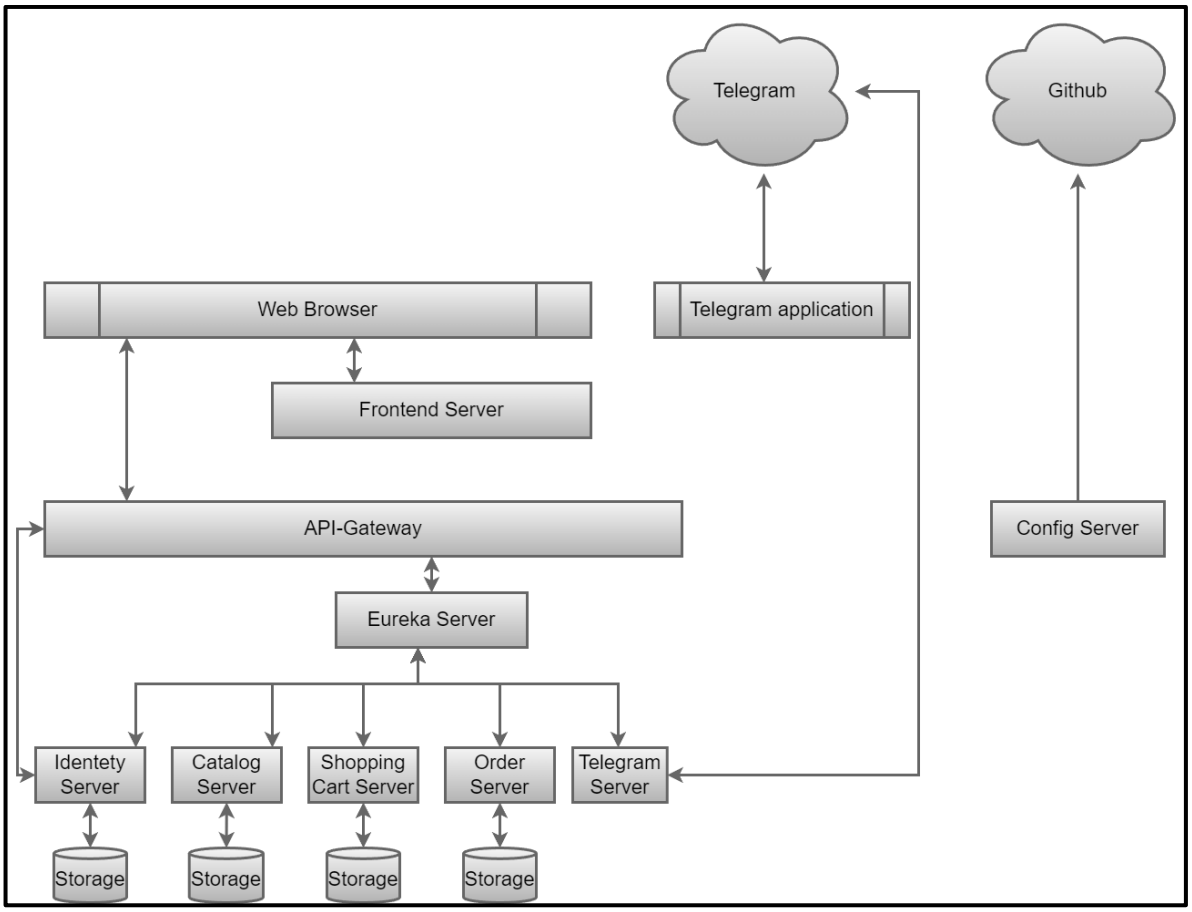

Рисунок 3.1 – Схема мікросервісної архітектури системи обробки замовлень для інтернет-магазину

На цьому рисунку позначено, що основними точками для взаємодії користувача з системою є «Web Browser» (браузер) та «Telegram application» (telegram застосунок).

За допомогою браузеру додавати або видаляти товари з каталогу (необхідна роль адміністратора), додавання товару з каталогу до корзини, зміна кількості товарів у корзині користувача, створення замовлення з вказанням адреси та додаткової інформації що до замовлення, а також реєстрація нового користувача та вхід до системи.

За допомогою застосунку телеграм, з використанням ботів здійснюється підтвердження або відхилення замовлення менеджером.

3.2 Telegram Bot

Месенджер «Telegram» надає можливість взаємодії з користувачами через «ботів» за допомогою Telegram Bot API. Для реєстрації нового бота використовується спеціальний чат @BotFather (рисунок 3.2).

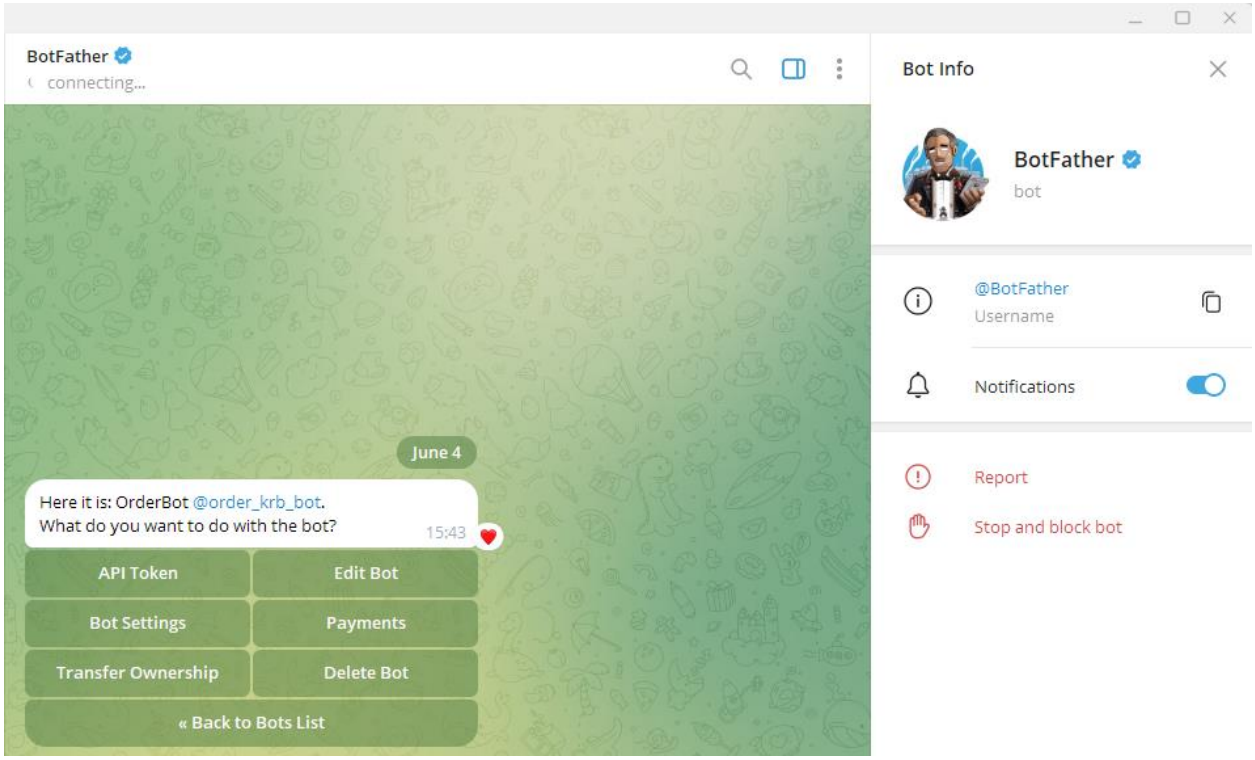

Рисунок 3.2 – Чат для налаштування боту Telegram

Після створення бота надається спеціальний токен для взаємодії з API. Клієнт надсилатиме в застосунок повідомлення, а Telegram сервер надсилатиме їх до створеного сервісу, що відповідає за обробку цих повідомлень. Сам сервіс також має змогу приймати повідомлення та надсилати їх до Telegram серверу, після чого воно буде надіслано до вказаного у повідомленні чату, за допомогою унікального ідентифікатора (chat_id). У застосунку ці чати будуть вказані у конфігураційному файлі.

Приклад отримання повідомлення від бота та відправлення повідомлення до нього зображено на рисунку 3.3. Отримане повідомлення складається з ім’я користувача, унікальних ідентифікаторів товару, їх кількості, адреса замовлення та додаткової інформації. В кінці повідомлення зазначено 2 команди для прийняття та відхилення замовлення. Ці команди виділені синім кольором, складаються з назви операції та номеру замовлення. Після натискання на одну з команд відправляється повідомлення на сервер і обробляється відповідним методом.

Рисунок 3.3 – Повідомлення у чат боті

Сервіс що приймає на надсилає повідомлення до телеграм боту було написано за допомогою мови Python, бібліотеки aiogram та aiohttp. Aiogram реєструє адресу, що відповідатиме за отримання повідомлень на сервері Telegram (webhook).

У конфігураційному файлі вказано адреса серверу ngrok, який відповідає пересилання повідомлень до локального серверу (рисунок 3.4). Після розгортання

застосунку на зовнішньому ресурсі з виділеною адресою цей етап можна пропустити.

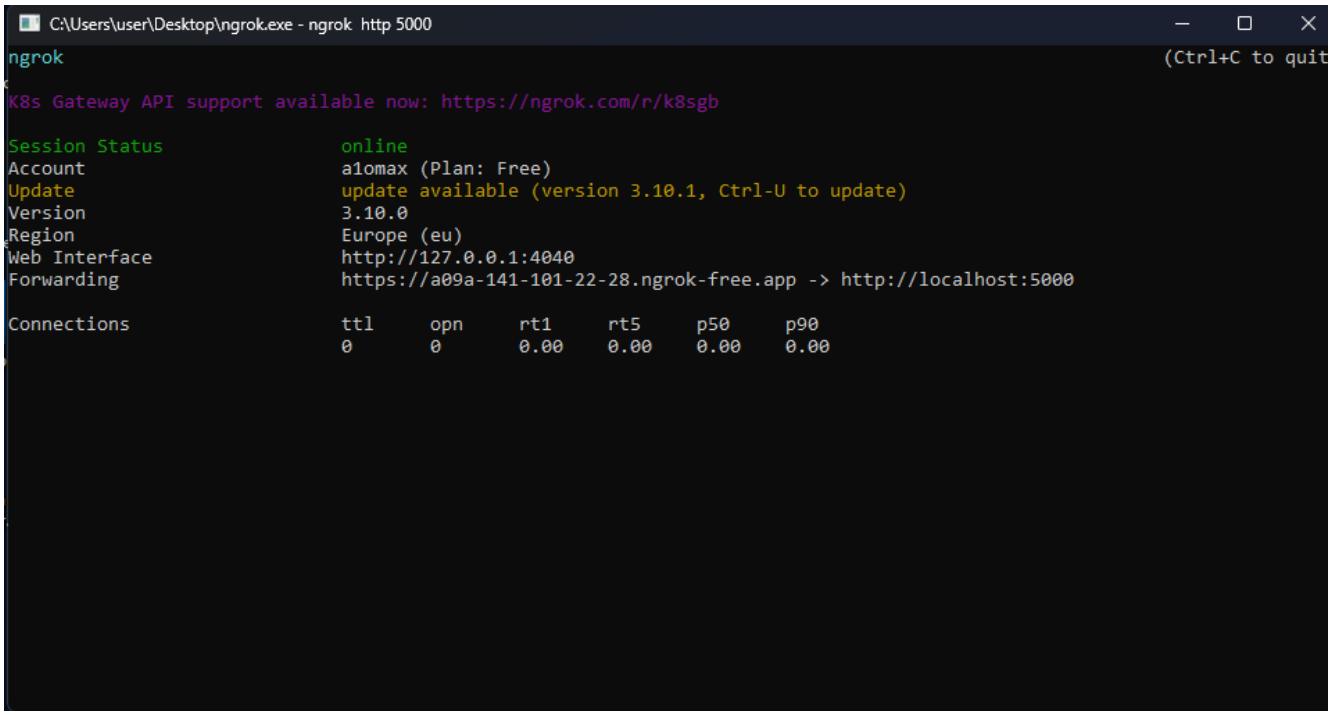

Рисунок 3.4 – Сервер Ngrok

Цей сервіс також приймає http запити на відправку повідомлень до боту від інших сервісів. В цьому застосунку реалізовано лише сповіщення про створення нового замовлення. Цю функціональність в подальшому можна легко розширювати.

3.3 Сервіс виявлення Eureka Server

Використання http потребує, щоб сервери (мікросервіси) знали шлях за яким треба відправляти запити. Для вирішення проблеми жорсткої прив’язки до адрес та можливість їх динамічної зміни було використано Eureka Server, сервіс виявлення мікросервісів (Discovery Service). Цей сервер «реєструє» у пам’яті інформацію про інші мікросервіси. Після цього вони мають можливість отримувати інформацію про адреси інших сервісів і відправляти їм повідомлення. За допомогою цього

зникає потреба у майбутній зміні конфігурацій при розгортанні на інших платформах.

Щоб створити цей сервіс достатньо імпортувати необхідні залежності, та за допомогою конфігураційного файлу та анотацій цей сервер створюється автоматично. Адміністратор також може переглянути інформацію про зареєстровані сервіси можна за URL, що було вказано в налаштуваннях.

На рисунку 3.5 продемонстровано сторінку Eureka Server, на якій позначено інформацію про стан серверу, а також про стан зареєстрованих у ньому сервісів.

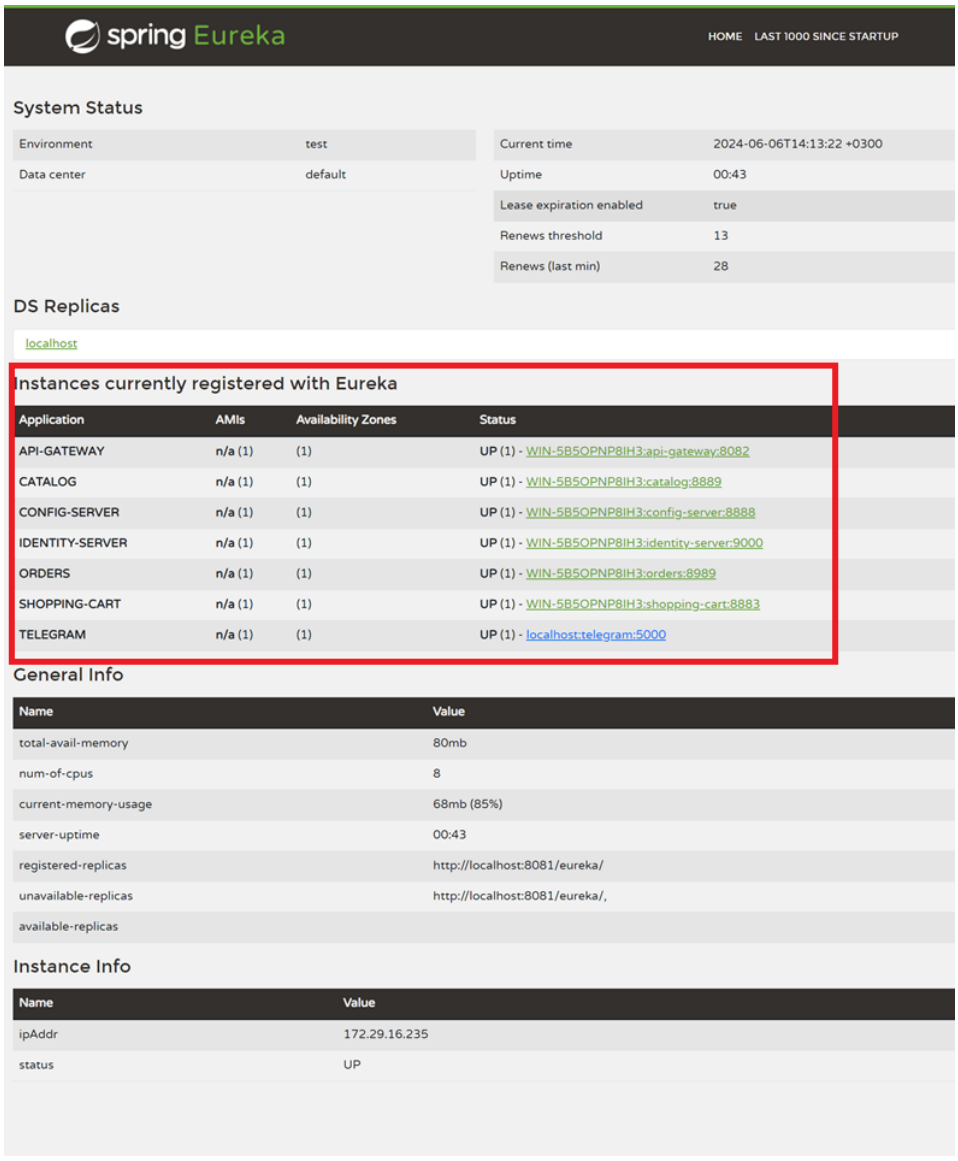

Рисунок 3.5 – Інформацій про стан Eureka Server

3.4 Frontend Server

Frontend Server відповідає за відображення інформації у браузері. Його було виконано з використанням мови JavaScript, Node JS та фреймворку Vue.

Цей серверний застосунок не є частиною мікросервісної архітектури, він створений для зручної взаємодії користувача з системою. Він використовує JavaScript для відправлення запитів до API-Gateway. Це спрощує розробку застосунків, що взаємодіють з системою обробки замовлень.

3.4.1 Сторінки для авторизації

За допомогою фреймворку було створено домашню сторінку (рисунок 3.6), що має посилання на сторінки для реєстрації (рисунок 3.7) та входу до системи для користувача (рисунок 3.8).

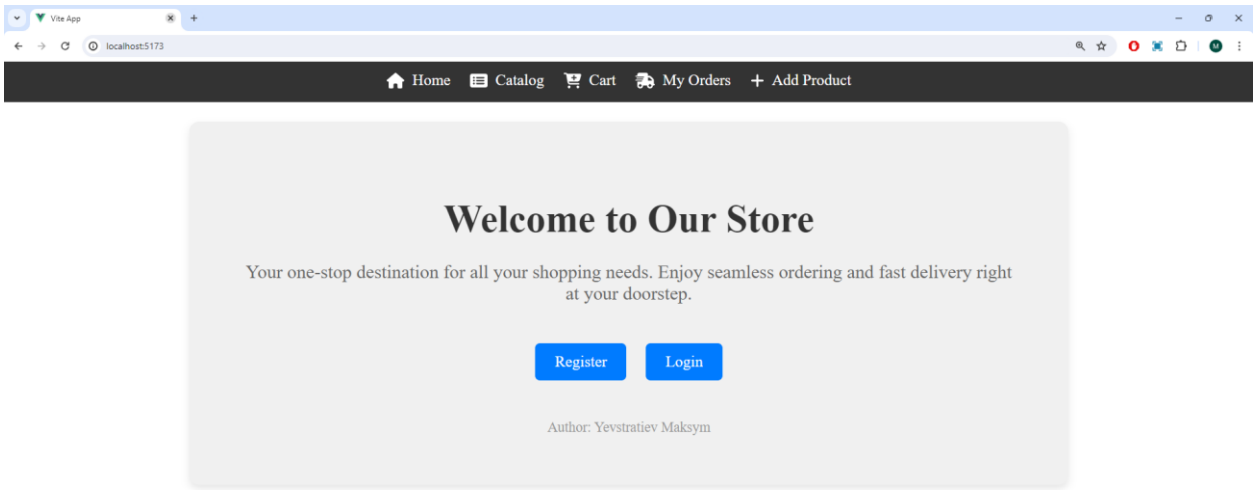

Рисунок 3.6 – Головна сторінка

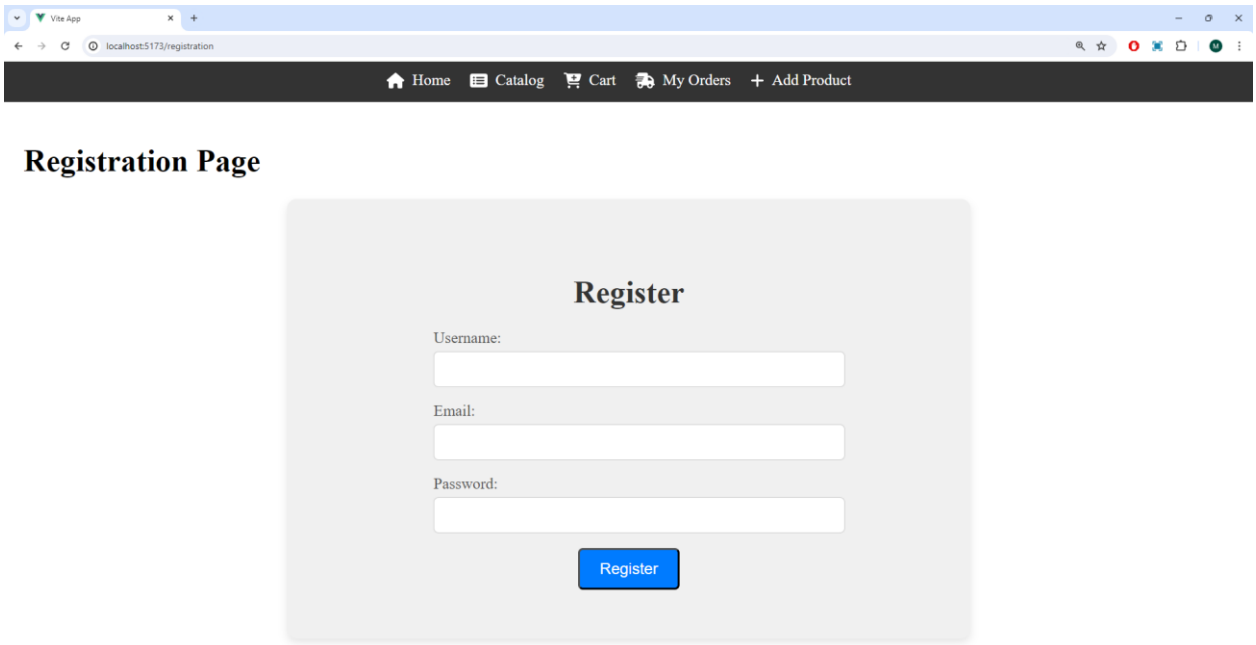

Рисунок 3.7 – Сторінка реєстрації користувача

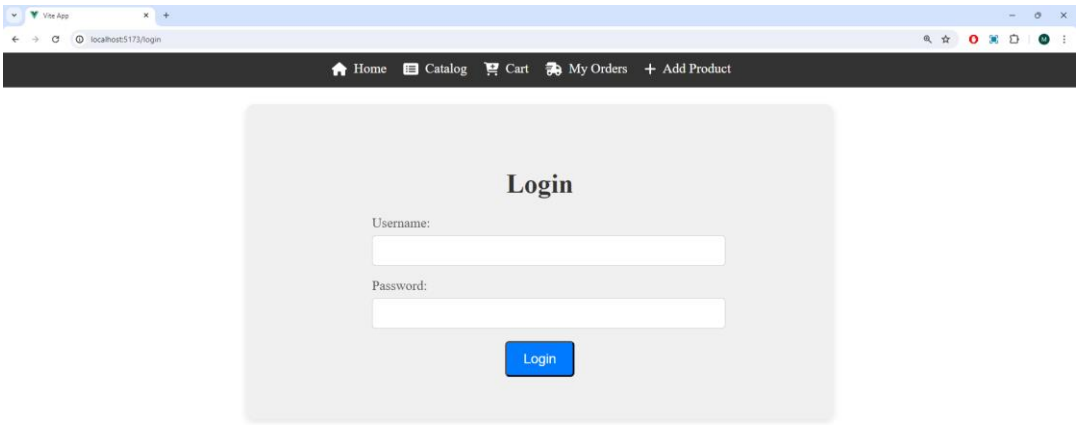

Рисунок 3.8 – Сторінка входу у систему

Форми реєстрації та входу, після натиснення на кнопку, відправляють запит до серверу, після чого отримують відповідь у вигляді JWT токену (рисунок 3.9). Цей токен зберігається у localStorage браузера клієнта (рисунок 3.10). Після цього цей токен додається до запитів (рисунок 3.11), що надходять до API Gateway.

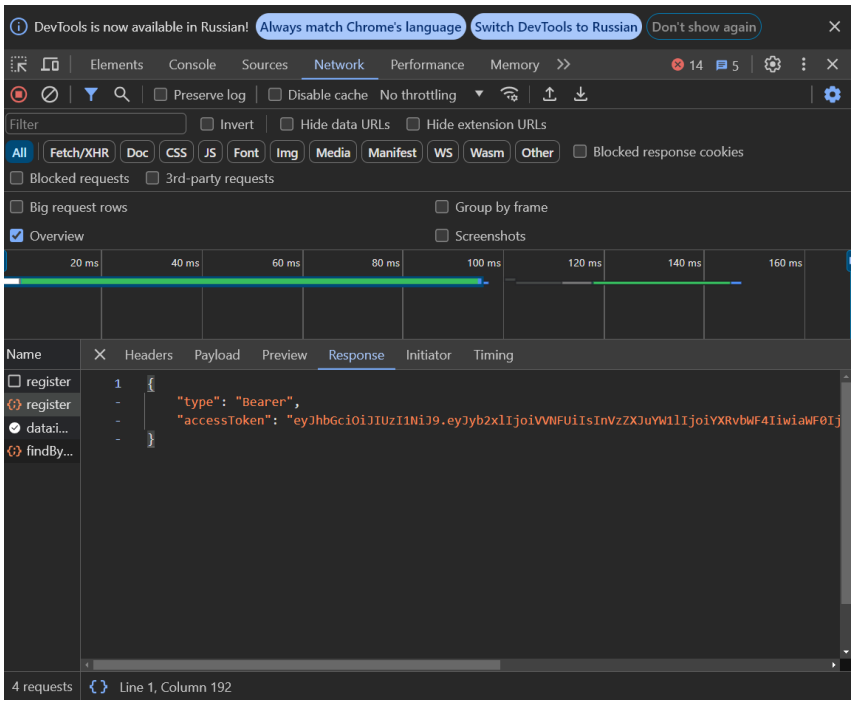

Рисунок 3.9 – Відповідь від сервера з токеном

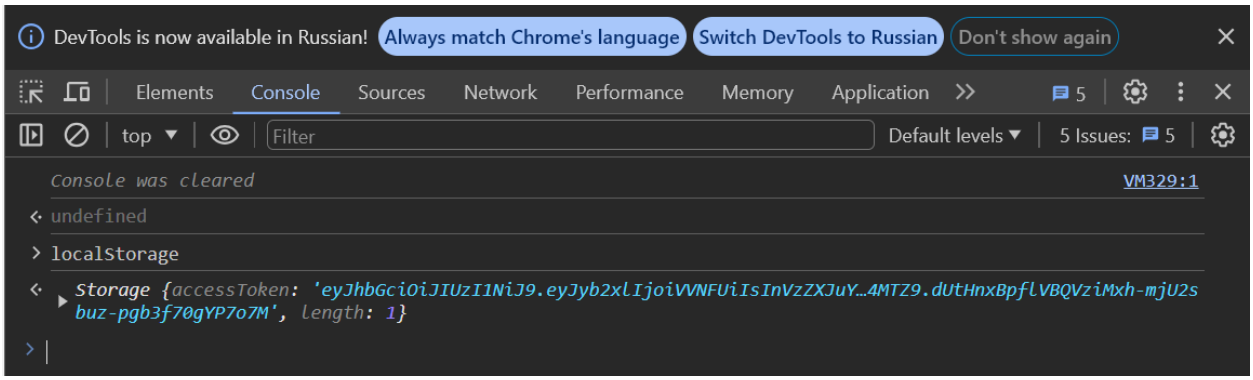

Рисунок 3.10 – Збережений токен в localStorage

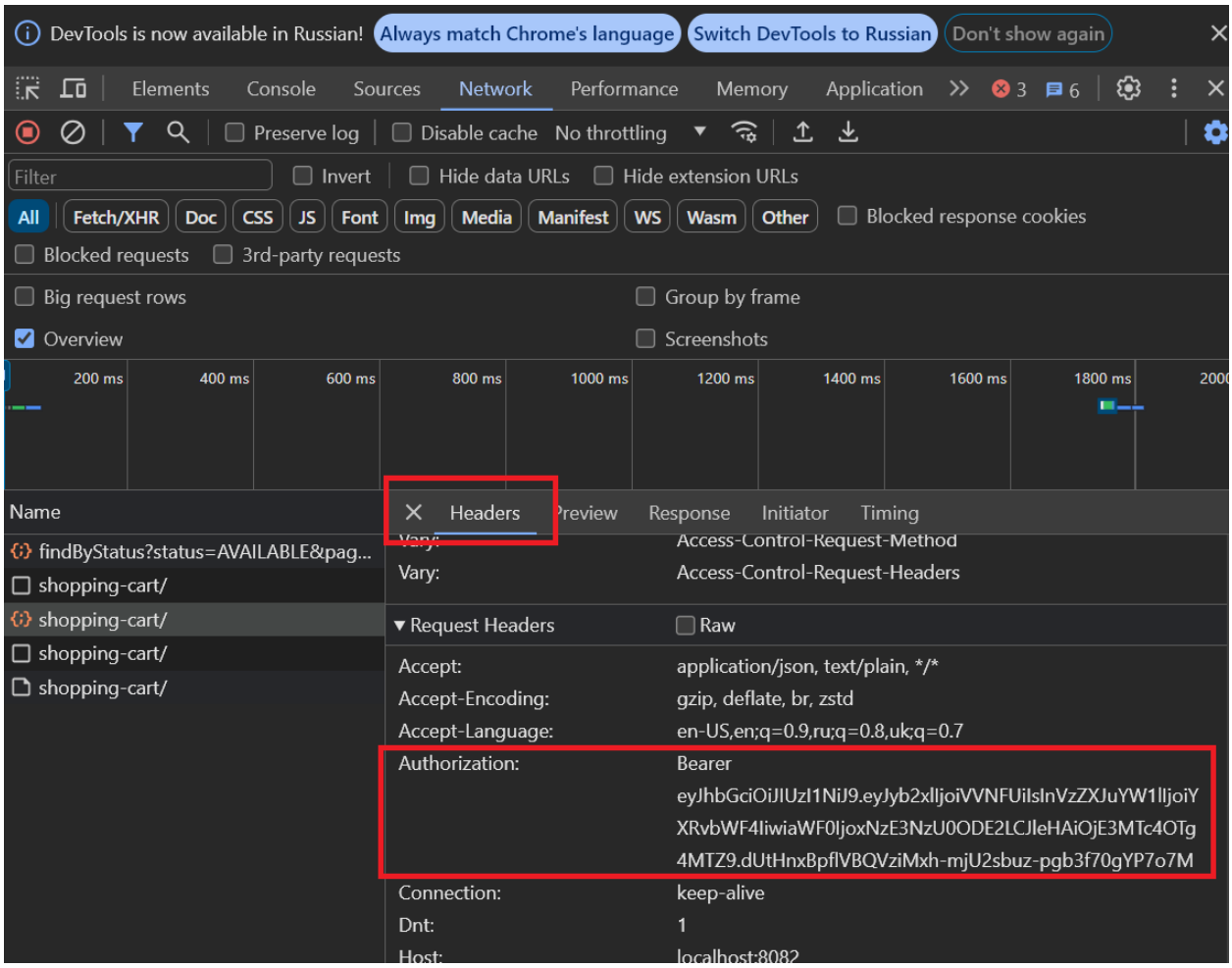

Рисунок 3.11 – Запит до серверу з використанням Authorization Header

3.4.2 Каталог товарів

Авторизовані та неавторизовані користувачі можуть переглядати каталог товарів (рисунок 3.12).

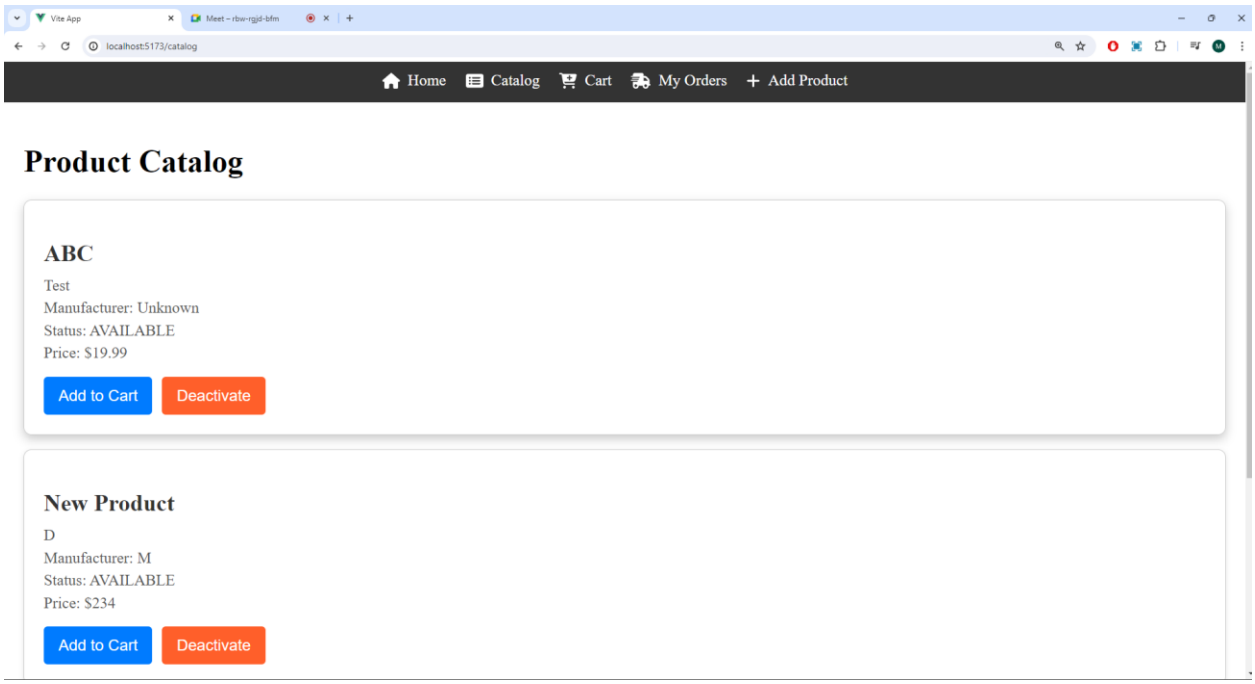

Рисунок 3.12 – Сторінка каталогу

Користувачі мають можливість скористатися кнопкою «Add to Cart», яка додає товар до корзини. Адміністратор, окрім того, може скористатися кнопкою «Deactivate». Натиснення на цю кнопку змінює статус товару на архівований і не буде показуватися користувачам. Видалення товарів з бази не є рекомендованим, тому що в такому разі інші сутності, які спираються на id товару, не зможуть функціонувати правильно.

3.4.3 Корзина товарів

Додані то корзини товари можна переглянути на сторінці «Cart», що зображена на рисунку 3.13. Дані про стан корзини одразу надсилаються до серверу після будь якої зміни. Це дозволить користуватися сайтом на будь якому пристрої з одного акаунту і мати спільну корзину.

Збільшити, зменшити та вилучити товар з корзини можна виконати за допомогою клавіш «+», «-» та «Remove» відповідно. За допомогою чек-боксу можна виділити необхідні товари, і після натискання на клавішу «Create Order»

з’являється popup-форма для додаткової інформації про замовлення, а саме: деталі та адреса замовлення (рисунок 3.14).

Коли замовлення сформовано користувачем, текст замовлення формується у текстовий формат і відправляється у телеграм (рисунок 3.15).

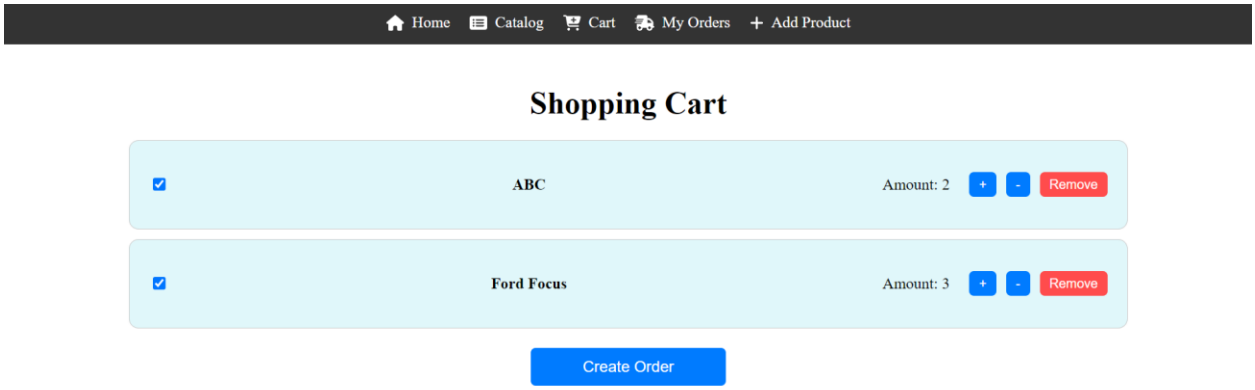

Рисунок 3.13 – Корзина користувача

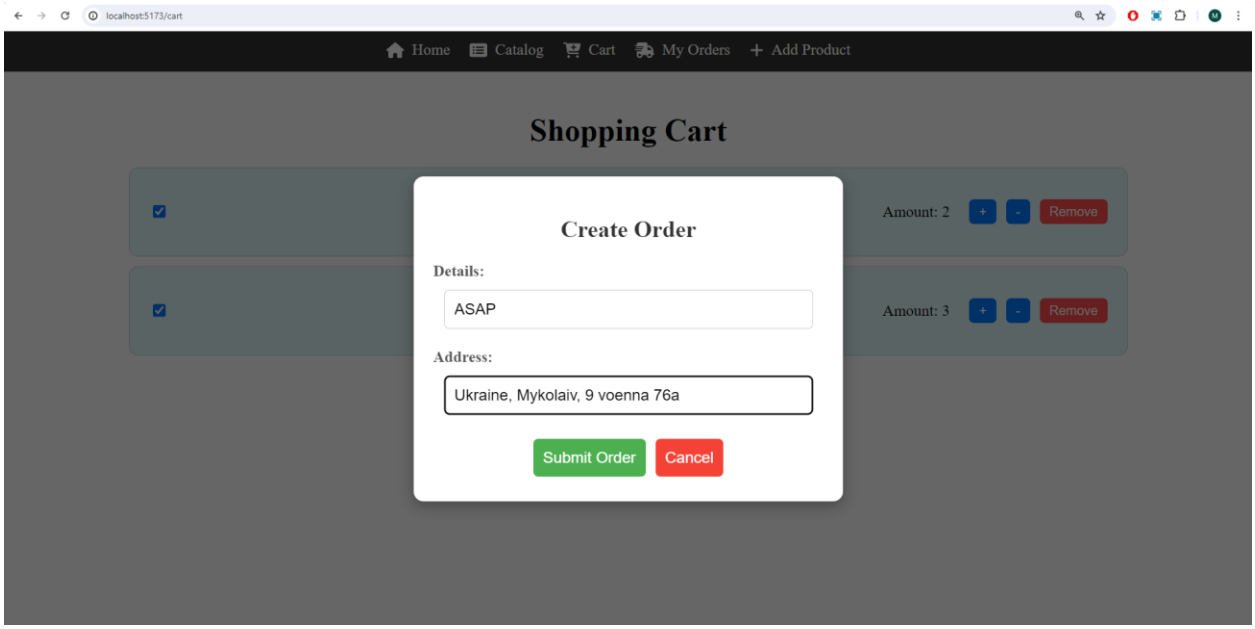

Рисунок 3.14 – Popup форма для замовлення

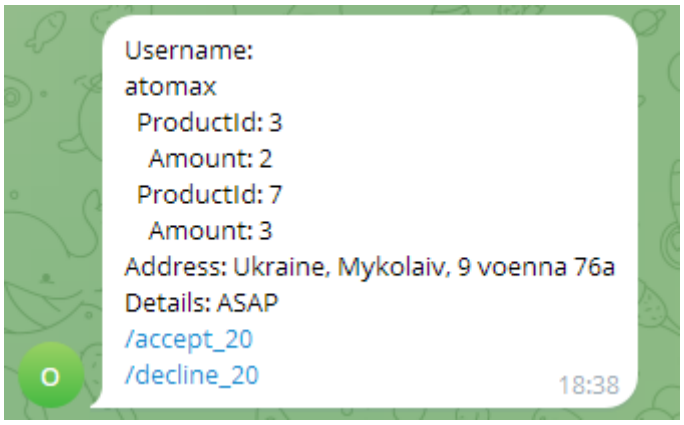

Рисунок 3.15 – Створене замовлення у телеграм боті

3.4.4 Перегляд замовлень

Після формування замовлення воно записується в сховище і користувач має можливість переглядати їх. Для цього створено сторінку «My Orders» (рисунок 3.16). Можна переглядати замовлення із застосуванням фільтру статусу замовлення (рисунок 3.17).

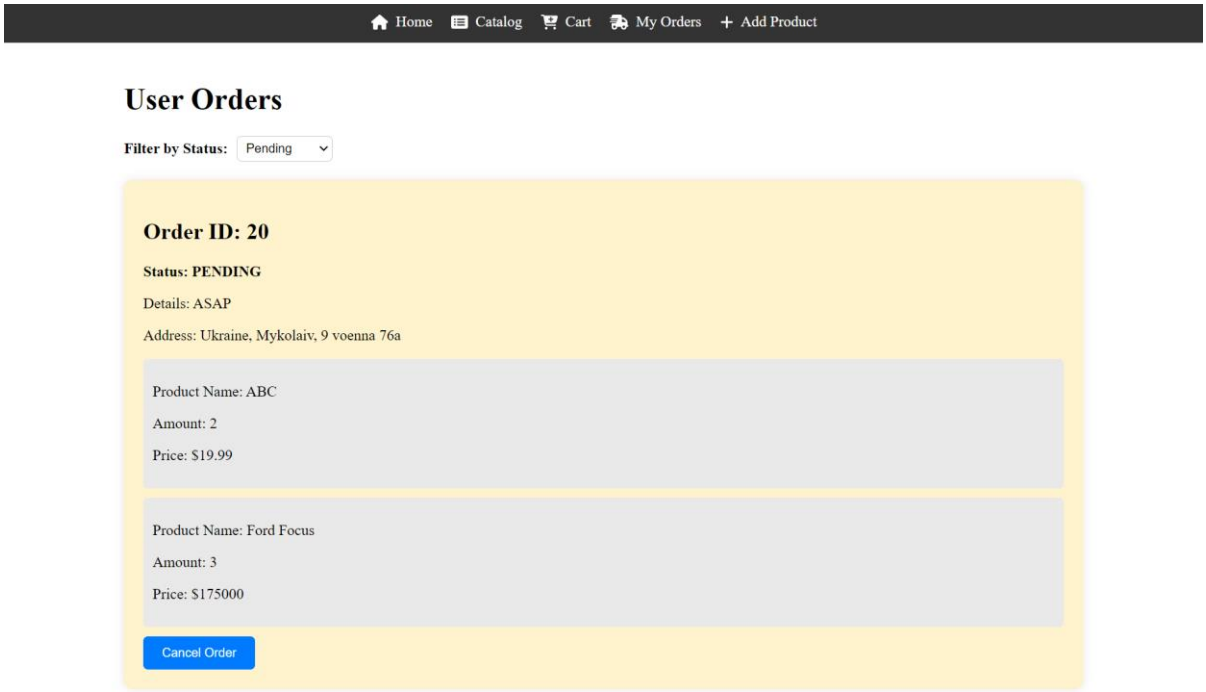

Рисунок 3.16 – Сторінка з замовленнями користувача

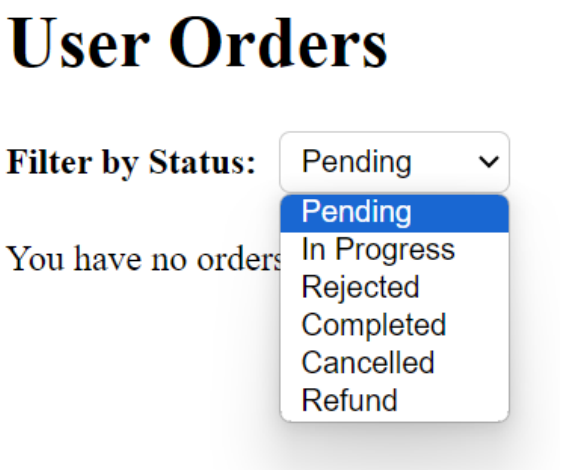

Рисунок 3.17 – Фільтр замовлень за статусом

Замовлення, що мають статус «Pending», тобто такі, які ще не встигли прийняти на виконання, можна відмінити за допомогою кнопки «Cancel Order». На рисунку 3.18 зображено приклад списку відхилених користувачем замовлень.

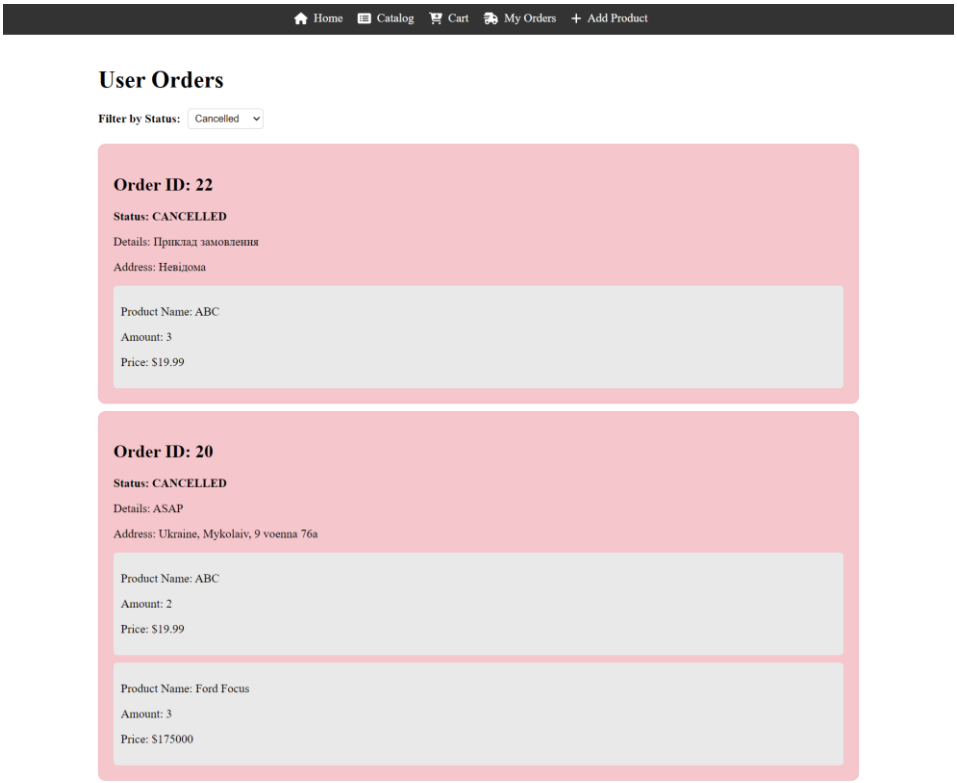

Рисунок 3.18 – Відхилені замовлення

3.4.5 Додавання товару до каталогу

Для зручності наповнення сайту контентом було створено форму «Add Product» (рисунок 3.19). Відправка цієї форми потребує прав адміністратора. Створений запис у каталозі продемонстровано на рисунку 3.20.

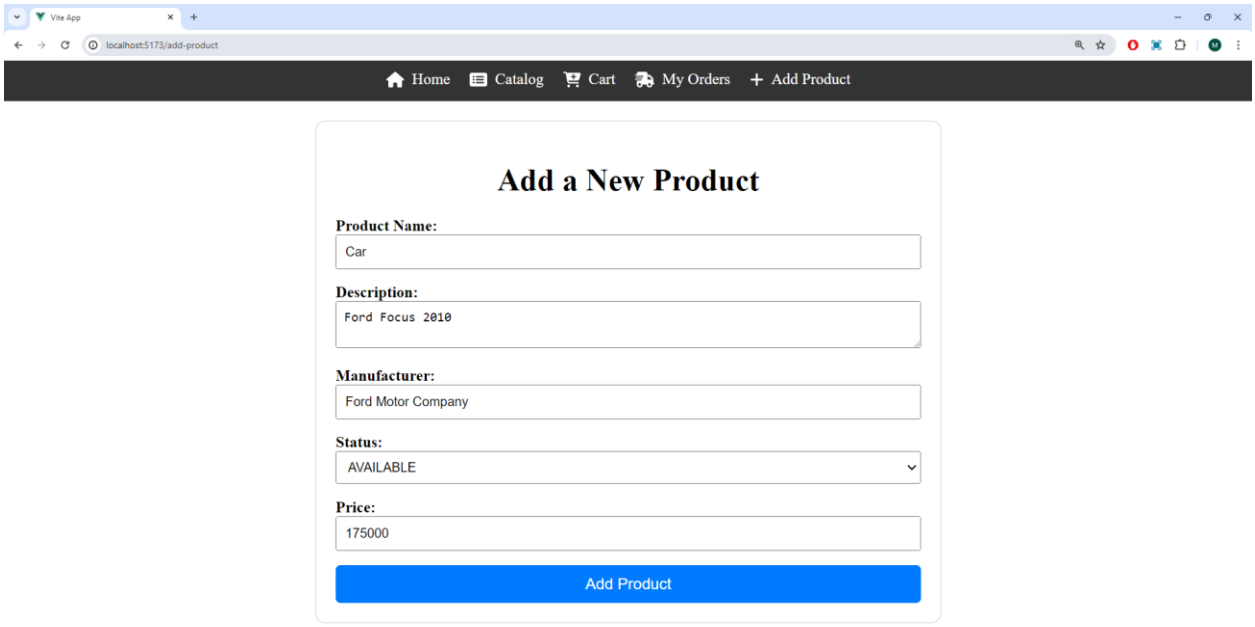

Рисунок 3.19 – Форма створення запису про продукт у каталозі

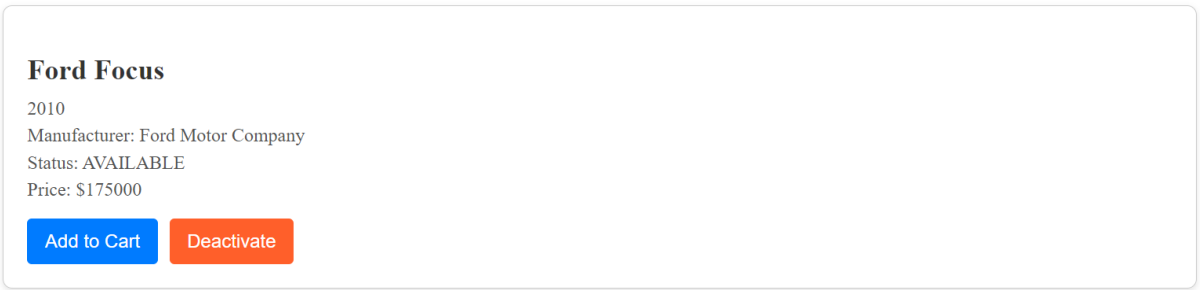

Рисунок 3.20 – Створений новий продукт у каталозі за допомогою форми

3.5 API-Gateway

Після отримання запиту API-Gateway виконує фільтрацію запитів, у якому перевіряє підпис токена, та, якщо він дійсний, дістає ім’я користувача та його роль

з токену. Це ім’я користувача додається до запиту що надійшов як X-username header, після чого перенаправляється на мікросервіс що може обробити запит. Це дає змогу не перевіряти токен у кожному сервісі, а просто використати username, що надійшов. Це накладає додаткове обмеження на сервіси, тому що використовувати їх без API-Gateway можна лише всередині локального серверу, інашке можливе отримання несанкціонованого доступу до даних. Перевагою цього підходу є легка взаємодія між самими серверами.

Також API-Gateway, дістаючи роль користувача, такі як ADMIN або USER (адміністратор, користувач), виконує роль сервера аутентифікації. У коді конфігурації записані адреси, доступ до яких потребує ролі адміністратора, користувача, або можуть бути отриманим не авторизованим користувачем (рисунок 3.21).

```
public static final List<String> openApiEndpoints = List.of(
        "/auth/register",
        "/auth/token",
        "/eureka",
        "/catalog/products/search/"
);

1 usage
public static final List<String> adminApiEndpoints = List.of(
        "/orders/admin",
        "/catalog/products/admin/add"
);
```

Рисунок 3.21 – Відкриті та адміністративні маршрути

3.6 Identity Server

Для збереження даних про користувачів, такі як данні для їх ідентифікації, роль та додаткова інформація, було створено мікросервіс, якому було дано назву Identity Server.

Інформація, яка зберігається в таблиці бази даних, що належить цьому мікросервісу продемонстровано на рисунку 3.22. В таблиці містяться наступні поля:

- id – унікальний ідентифікатор запису;
- email – електрона адреса;
- password – пароль у вигляди хешу;
- role – роль користувача;
- username – унікальне ім’я користувача.

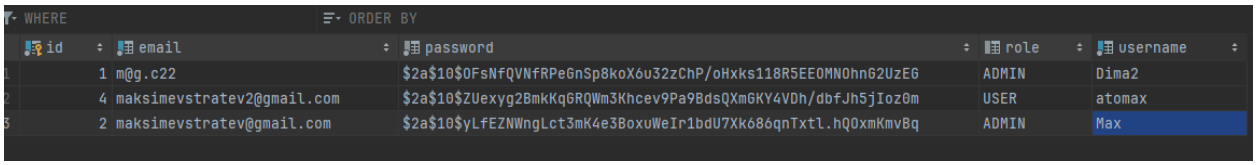

Рисунок 3.22 – Таблиця бази даних Identity Server

При реєстрації нового користувача, пароль зберігається у форматі хешу. При аутентифікації користувача пароль проходить хешування тією самою функцією що і при реєстрації, тому результати повинні бути ідентичними.

Також Identity Server відповідає за видачу та перевірку **JWT** (JSON Web Token). Для цього була використана бібліотека **jjwt-api**.

3.7 Config Server

Для збереження налаштувань мікросервісів в одному місці використовується технологія **Spring Cloud Netflix.** За допомогою неї можна створити сервіс, який звертатиметься до зовнішнього джерела з налаштуваннями для сервісів. Це дозволяє використати систему контролю версій GIT для збереження та відслідковування змін, що відбулися. Як репозиторій для налаштувань було обрано Github. На рисунку 3.23 зображено файли налаштувань у репозиторії. Назви каталогів відповідають назвам застосунку, а файл під назвою «application-dev.yaml» містить у собі загальні конфігурації для всіх мікросервісів.

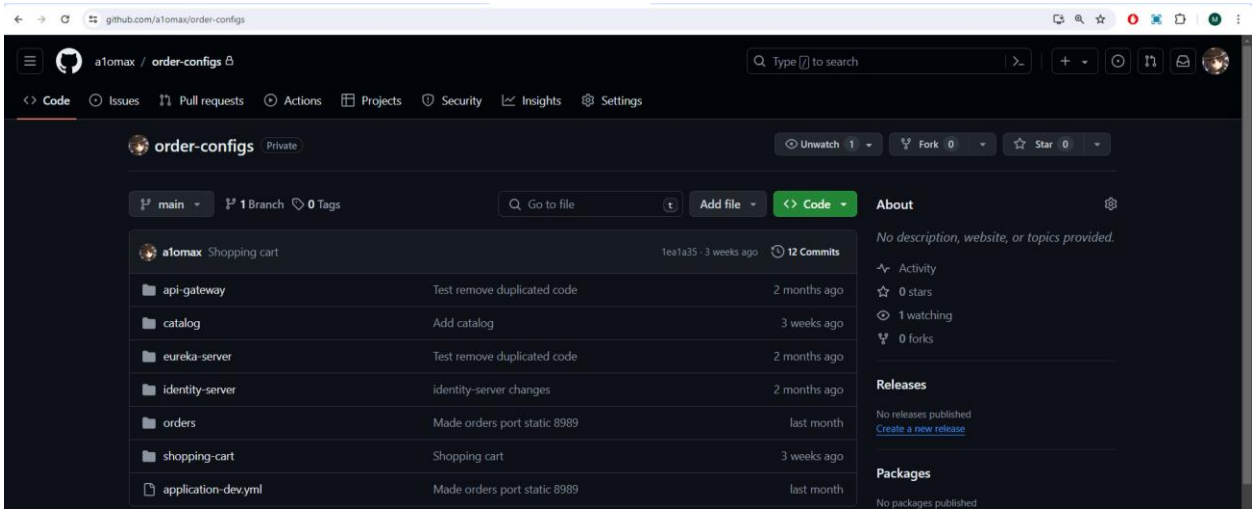

Рисунок 3.23 – Файли конфігурацій у репозиторії GitHub

3.8 Catalog

За операції з товарами, а саме збереження, зміну та відправку інформації про них відповідає мікросервіс **Catalog**. Ця інформація може бути отримана у вигляді json відповіді як клієнтом, так і іншим сервісом. Так, сервіс Orders виконує перевірку чи існує товар надсилаючи запит до Catalog.

Таблиця catalog (рисунок 3.24) містить дані про товари та складається з наступних полів:

– id – унікальний ідентифікатор запису;
– name – опис товару;
– description – опис товару;
– manufacturer – виробник товару;
– status – стан товару;
– price – ціна товару.

WHERE ORDER BY id DESC

	id	description	manufacturer	name	price	status
1	8	2	3	1	4	AVAILABLE
2	7	2010	Ford Motor Company	Ford Focus	175000	AVAILABLE
3	6	f	f	Cucumber	1	AVAILABLE
4	5	241	41	123	51	ARCHIVED
5	4	D	M	New Product	234	AVAILABLE
6	3	Test	Unknown	ABC	19.99	AVAILABLE
7	2	Test		Cucumber	19.99	ARCHIVED
8	1	Oleksii	Sample Manufacturer	Orange	19.99	ARCHIVED

Рисунок 3.24 – Таблиця catalog

3.9 Orders

Обробку створених замовлень виконує мікросервіс Orders

У базі даних цього мікросервісу міститься дві таблиці: order та orderproduct.

Таблиця order (рисунок 3.25) містить дані про замовлення складається з наступних полів:

– id – унікальний ідентифікатор запису;
– address – адреса замовлення;
– details – деталі замовлення;
– status – стан замовлення;
– username – унікальне ім’я користувача.

WHERE ORDER BY username

	id	address	details	status	username
1	19	Mykolaiv	my order	IN_PROGRESS	atomax
2	22	Невідома	Приклад замовлення	CANCELLED	atomax
3	20	Ukraine, Mykolaiv, 9 voenna 76a	ASAP	CANCELLED	atomax
4	15	2	1	CANCELLED	Dima2
5	17	2	1	IN_PROGRESS	Dima2
6	10	net	da	CANCELLED	Dima2
7	18	i	yjg	IN_PROGRESS	Dima2
8	16	b	a	CANCELLED	Dima2
9	11	net	da	CANCELLED	Dima2
10	14	2	1	IN_PROGRESS	Dima2

Рисунок 3.25 – Таблиця orders

Таблиця orderproduct (рисунок 3.26) містить дані про товари в замовленні і складається з наступних полів:

– id – унікальний ідентифікатор запису;
– amount – кількість товару;
– price – ціна;
– product_id – унікальний ідентифікатор товару з каталогу;
– order_id – унікальний ідентифікатор замовлення.

	id	amount	price	product_id	order_id
1	39	3	19.99	3	22
2	37	3	175000	7	20
3	36	2	19.99	3	20
4	35	3	19.99	3	19
5	34	1	234	4	19
6	33	4	19.99	3	18
7	32	3	19.99	3	17
8	31	1	19.99	3	16
9	30	1	19.99	1	15
10	29	5	19.99	2	15
11	28	1	19.99	1	14
12	27	5	19.99	2	14
13	22	1	19.99	2	11
14	21	3	19.99	1	11
15	20	1	19.99	2	10
16	19	3	19.99	1	10

Рисунок 3.26 – Таблиця orderproduct

3.10 Shopping Cart

Авторизовані користувачі повинні мати можливість додавати товари до корзини перед створення замовлення. Для цього створено мікросервіс Shopping Cart. Для кожного користувача в таблиці створюється корзина, до якої можна додавати товари. Реалізація працює за наступним принципом: додавання відбувається на frontend сервері. Спочатку відбувається отримання наявної корзини

користувача, потім до цієї відповіді сервера додається нові товари або змінюється їх кількість, після цього весь вміст корзини передається у мікросервіс для оновлнння вмісту корзини. Перевагою такої реалізації є низька складність, а недоліком – можливість великої кількості даних у запитах.

Цей мікросервіс має дві таблиці: cart та cart_product.

Таблиця cart (рисунок 3.27) містить інформацію про корзини користувачів і складається з наступних полів:

– id – унікальний ідентифікатор запису;
– username – унікальне ім’я користувача.

Рисунок 3.27 – Таблиця cart

Таблиця cart_product (рисунок 3.28) містить інформацію про товари в корзинах і складається з наступних полів:

– cart_id – унікальний ідентифікатор корзини;
– product_id – унікальний ідентифікатор продукту;
– amount – кількість продукту.

Рисунок 2.28 – Таблиця cart_product

Ця таблиця не має колонки з власним унікальним ідентифікатором, її унікальний ключ є складеним з полів cart_id та product_id. Це зроблено для того, аби виключити можливість існування двох однакових записів про товар у корзині.

Висновки до розділу 3

Отже, в цьому розділі було розглянуто програмну реалізацію інформаційної системи обробки замовлень для інтернет-магазину, а саме мікросервісну реалізацію. Ця система була розділена на декілька мікросервісів, кожен з яких виконує свою роль і має окремі таблиці в базі даних. Також було розлянуто реалізацію frontend застосунку на Vue.js, за допомогою якого здійснюється взаємодія з системою як користувачами, так і менеджерами інтернет-магазину.

ВИСНОВКИ

В результаті виконання кваліфікаційної роботи бакалавра було проведено детальний аналіз існуючих рішень на ринку інформаційних систем обробки замовлень, що дозволило виділити їх переваги та недоліки. Було виявлено, що більшість сучасних систем використовують мікросервісну архітектуру для підвищення масштабованості та надійності.

На основі проведеного аналізу було розроблено власний проект інформаційної системи обробки замовлень, яка використовує мікросервісну архітектуру. Система включає кілька сервісів, таких як сервіс керування каталогом товарів, сервіс керування замовленнями, сервіс корзини користувача, сервіс сповіщення у месенджер «Telegram», сервіс аутентифікації, сервіс виявлення та сервіс конфігурацій, а також frontend-застосунок з використанням мови JavaScript.

Розроблена система дозволяє автоматизувати процеси обробки замовлень, що значно знижує час на їх обробку, підвищує точність та зменшує ймовірність помилок. Це досягається за рахунок інтеграції різних сервісів та використання сучасних технологій обробки даних.

Особливу увагу було приділено питанням безпеки даних та захисту інформації. Використання сучасних методів шифрування та аутентифікації забезпечує високий рівень захисту інформації від несанкціонованого доступу.

Система розроблена таким чином, щоб бути зручною як для адміністраторів, так і для кінцевих користувачів. Інтуїтивно зрозумілий інтерфейс та висока швидкість обробки даних сприяють підвищенню ефективності роботи.

Отже, що розроблена інформаційна система обробки замовлень для інтернет-магазину відповідає сучасним вимогам ринку та має високу функціональність, що дозволяє її ефективно використовувати в умовах реального бізнесу.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Sebastian K Boell. What is an Information System? 48th Hawaii International Conference on System Sciences. 2015.

2. Коваленко О. С., Добровська Л. М. Проектування інформаційних систем. навч. посіб. Київ, 2020. 25-27 с.

3. Order Processing. URL: https://dealhub.io/glossary/order-processing/ (дата звернення: 09.05.2024).

4. Order Management. URL: https://www.ibm.com/topics/order-management (дата звернення: 09.05.2024).

5. IBM Sterling Order Management. URL: https://www.ibm.com/products/order-management/ (дата звернення: 09.05.2024).

6. Intelligent Order Management URL: https://dynamics.microsoft.com/ru-ru/intelligent-order-management/overview/ (дата звернення: 09.05.2024).

7. Order Management Systems Defined: What Is an OMS? URL: https://www.netsuite.com/portal/resource/articles/erp/what-is-oms.shtml (дата звернення: 09.05.2024).

8. M. A. Valashani, A. M. Abukari, *Journal of Applied Intelligent Systems & Information Sciences.* 2020. Vol. 1, № 2, P. 70-90. DOI: 10.22034/JAISIS.2020.103704

9. D. McCreary, A. Kelly. Making Sence of NoSQL. A guide for managers and the rest of us. 2014. URL: https://www.bigdata.ir/wp-content/uploads/2016/08/5FB45AB6A5AEEC2E405B214983F9A04B.pdf (дата звернення: 09.05.2024).

10. Security in Microservices Architectures. URL: https://www.sciencedirect.com/science/article/pii/S1877050921003719 (дата звернення: 09.05.2024).

11. Nadareishvili I., Mitra R., McLarty M., Amundsen M. Microservice Architecture: Aligning Principles, Practices, and Culture 2016.

12. Introduction to Java. URL: https://www.oracle.com/java/technologies/introduction-to-java.html

13. How Java works. URL: https://www.ibm.com/topics/java#How+Java+works. (дата звернення: 09.05.2024).

14. Spring Framework. URL: https://docs.spring.io/spring-framework/reference/ (дата звернення: 09.05.2024).

15. Spring Boot. URL: https://foxminded.ua/spring-boot/ (дата звернення: 09.05.2024).

16. Spring Boot Starters. URL: https://www.baeldung.com/spring-boot-starters (дата звернення: 09.05.2024).

17. JPA & Hibernate. URL: https://www.baeldung.com/learn-jpa-hibernate (дата звернення: 09.05.2024).

18. JPA Entities. URL: https://www.baeldung.com/jpa-entities (дата звернення: 09.05.2024).

19. JPA Repository. URL: https://docs.spring.io/spring-data/jpa/reference/repositories/definition.html (дата звернення: 09.05.2024).

20. Spring Data REST. URL: https://spring.io/projects/spring-data-rest (дата звернення: 09.05.2024).

21. Object relational database. URL: https://www.techopedia.com/definition/8714/object-relational-database-ord (дата звернення: 09.05.2024).

22. PostgreSQL URL: https://www.postgresql.org/about/ (дата звернення: 09.05.2024).

23. High modifiability MVC framework with combined spring framework and model translator. URL: https://lexitron.nectec.or.th/public/NCIT_2010_Bangkok%20_Thailand/index_files/papers/31-p048.pdf (дата звернення: 09.05.2024).

24. Spring Web MVC. URL: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html (дата звернення: 09.05.2024).

25. Spring Security. URL: https://spring.io/projects/spring-security (дата звернення: 09.05.2024).

26. Spring Cloud. URL: https://spring.io/projects/spring-cloud (дата звернення: 09.05.2024).

27. Vue.js. URL: https://vuejs.org/guide/introduction.html (дата звернення: 09.05.2024).

28. Node.js. URL: https://nodejs.org/en/learn/getting-started/introduction-to-nodejs (дата звернення: 09.05.2024).

29. General Python FAQ. URL: https://docs.python.org/3/faq/general.html#what-is-python (дата звернення: 09.05.2024).

30. Aiohttp. URL: https://docs.aiohttp.org/en/stable/index.html (дата звернення: 09.05.2024).

31. Aiogram. URL: https://pypi.org/project/aiogram/ (дата звернення: 09.05.2024).

ДОДАТОК А

Лістинг коду класу фільтру AuthenticationFilter мікросервісу api-gateway

```
@Component
@AllArgsConstructor
public class AuthenticationFilter extends AbstractGatewayFilterFactory<AuthenticationFilter.Config> {
    @Autowired
    private RouteValidator validator;
    @Autowired
    private RestTemplate restTemplate;

    @Autowired
    private JwtService jwtService;

    @Autowired
    private DiscoveryClient discoveryClient;

    public AuthenticationFilter() {
        super(AuthenticationFilter.Config.class);
    }

    @Override
    public GatewayFilter apply(Config config) {
        return ((exchange, chain) -> {
            if (exchange.getRequest().getMethod().equals(HttpMethod.OPTIONS)) {
                return chain.filter(exchange);
            }

            if (validator.isSecured.test(exchange.getRequest())) {
                if (!exchange.getRequest().getHeaders().containsKey(HttpHeaders.AUTHORIZATION)) {
                    throw new RuntimeException("Missing Authorization header");
                }
                String authHeader = exchange.getRequest().getHeaders().getFirst("Authorization");

                String token;
                if (authHeader == null || !authHeader.startsWith("Bearer ")) {
                    throw new RuntimeException("Invalid token");
                }
                token = authHeader.substring(7);
                validate(token);

                Map<String, String> payloadClaims = jwtService.decodeJwtPayload(token);

                if (!validator.authorize(payloadClaims.get("role"), exchange.getRequest().getURI().getPath())){
                    throw new RuntimeException("Not authorized");
                }


                payloadClaims.forEach((key, value) -> {
                    exchange.getRequest().mutate().header("X-" + key, value);
                });

            }
            return chain.filter(exchange);
        });
    }
```

```
    private void validate(String token) {
        String identityServiceUrl = getIdentityServiceUrlFromEureka();


        try {
            restTemplate.getForObject(identityServiceUrl + "/auth/validate?token=" + token, String.class);
        } catch (Exception e) {
            e.printStackTrace();
            throw new RuntimeException("Invalid token");
        }
    }
    private String getIdentityServiceUrlFromEureka() {
        List<ServiceInstance> instances = discoveryClient.getInstances("IDENTITY-SERVER");
        if (instances.isEmpty()) {
            throw new RuntimeException("No instances available for IDENTITY-SERVER");
        }
        return instances.getFirst().getUri().toString();
    }


    public static class Config {

    }
}
```

ДОДАТОК Б

Лістинг коду класу RouteValidator мікросервісу api-gateway

```
@Component
public class RouteValidator {

   public static final String ADMIN_ROLE_NAME = "ADMIN";

   public static final List<String> openApiEndpoints = List.of(
         "/auth/register",
         "/auth/token",
         "/eureka",
         "/catalog/products/search/"
   );

   public static final List<String> adminApiEndpoints = List.of(
         "/orders/admin",
         "/catalog/products/admin/add"
   );

   public Predicate<ServerHttpRequest> isSecured =
         request -> openApiEndpoints.stream()
               .noneMatch(uri -> request.getURI().getPath().contains(uri));

   public boolean authorize(String role, String requestPath) {
      for (String endpoint : adminApiEndpoints) {
         if (requestPath.startsWith(endpoint)) {
            return role.equals(ADMIN_ROLE_NAME);
         }
      }
      return true;
   }
}
```

ДОДАТОК В

Лістинг коду класу сервісу JwtService мікросервісу api-gateway

```
@AllArgsConstructor
@Service
public class JwtService {


    public Map<String, String> decodeJwtPayload(String jwtToken) {

        String payload = jwtToken.split("\\.")[1];

        byte[] bytes = Base64.getDecoder().decode(payload);

        String utf8String = new String(bytes, StandardCharsets.UTF_8);


        Gson gson = new Gson();

        return gson.fromJson(utf8String,  new TypeToken<Map<String, String>>(){}.getType());
    }
}
```

ДОДАТОК Г

Лістинг коду класу контролера ProductController мікросервісу catalog

```
@RestController
@RequestMapping("/products")
@AllArgsConstructor
public class ProductController {
    private ProductService productService;

    @GetMapping("/search/by-ids")
    @ResponseBody
    public ResponseEntity<List<Product>> getProductsByIds(@RequestParam List<String> ids) {
        return ResponseEntity.status(200).body(productService.getAllByIdIn(ids));
    }


    @PatchMapping("/{id}/change-status")
    @ResponseBody
    ResponseEntity<?> changeProductStatus(@PathVariable("id") Long orderId, @RequestParam("status") String status) {
        productService.changeProductStatus(orderId, ProductStatus.valueOf(status));
        return ResponseEntity.status(204).build();
    }

    @PostMapping("/admin/add")
    @ResponseBody
    ResponseEntity<?> addProduct(@RequestBody Product product){
        return ResponseEntity.status(201).body(productService.saveProduct(product));
    }
}
```

ДОДАТОК Д

Лістинг коду класу сервісу ProductService мікросервісу catalog

```
@AllArgsConstructor
@Service
public class ProductService {
   ProductRepository productRepository;

   public Product getProduct(Long productId){
      return productRepository.findById(productId).orElseThrow(()->new EntityNotFoundException("Product not
found"));

   }
   public void changeProductStatus(Long productId, ProductStatus productStatus){
      Product product = getProduct(productId);
      product.setStatus(productStatus);
      productRepository.save(product);
   }
   public List<Product> getAllByIdIn(List<String> ids){
      return productRepository.findAllByIdIn(ids);
   }

   public Product saveProduct(Product product){
      return productRepository.save(product);
   }
}
```

ДОДАТОК Е

Лістинг коду класів моделі Product та перелічення ProductStatus мікросервісу catalog

```
@Getter
@Setter
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Entity
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)
    private Long id;
    @Column
    private String name;
    @Column
    private String description;
    @Column
    private String manufacturer;

    @Column
    @Enumerated(EnumType.STRING)
    private ProductStatus status;

    @Column
    private Double price;

}
```

```
public enum ProductStatus {
    AVAILABLE,
    OUT_OF_STOCK,
    DISCONTINUED,
    ARCHIVED
}
```

ДОДАТОК Ж

Лістинг коду класу сервісу CartService мікросервісу shopping-cart

```
@Service
@Transactional
@AllArgsConstructor
public class CartService {

   private CartRepository cartRepository;
   private final CartProductMapper cartProductMapper;
   private CartProductRepository cartProductRepository;

   public Cart createCartForUsername(String username) {
      return cartRepository.save(Cart.builder().username(username).build());
   }

   public Cart getCartFromUsername(String username) {
      Optional<Cart> cart = cartRepository.findByUsername(username);
      return cart.orElseGet(() -> createCartForUsername(username));
   }

   public void clearCart(String username) {
      Cart cart = getCartFromUsername(username);
      List<CartProduct> cartProductList = cart.getCartProducts();
      if (cartProductList != null) {
         cartProductRepository.deleteAll(cartProductList);
      }
      cartRepository.save(cart);
   }
   public void updateCart(String username, CartDto updatedCartDto) {
      Cart cart = getCartFromUsername(username);
      clearCart(username);

      for (CartProductDto cartProductDto : updatedCartDto.getCartProducts()) {
         CartProduct cartProduct = cartProductMapper.toEntity(cartProductDto);
         cartProduct.setCart(cart);
         cartProductRepository.save(cartProduct);

      }
   }
}
```

ДОДАТОК И

Лістинг коду класів моделей CartProduct та Cart, та id-класу CartProductId мікросервісу shopping-cart

```
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Entity
@Table(name = "cart")
public class Cart {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)
    private Long id;

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

    @OneToMany(mappedBy = "cart")
    private List<CartProduct> cartProducts = new ArrayList<>();

}


@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Entity
@IdClass(CartProductId.class)
@Table(name = "cart_product")
public class CartProduct {

    @Id
    @Column(name = "product_id", nullable = false)
    private Long productId;

    @Id
    @ManyToOne(optional = false)
    @JoinColumn(name = "cart_id", nullable = false)
    private Cart cart;

    @Column(name = "amount", nullable = false)
    private Integer amount;
}

public class CartProductId implements Serializable {
    private Long productId;
    private Long cart;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
```

```
        CartProductId that = (CartProductId) o;
        return Objects.equals(productId, that.productId) && Objects.equals(cart, that.cart);
    }

    @Override
    public int hashCode() {
        return Objects.hash(productId, cart);
    }

    public CartProductId() {
    }

    public CartProductId(Long productId, Long cart) {
        this.productId = productId;
        this.cart = cart;
    }

    public Long getProductId() {
        return productId;
    }

    public void setProductId(Long productId) {
        this.productId = productId;
    }

    public Long getCart() {
        return cart;
    }

    public void setCart(Long cart) {
        this.cart = cart;
    }

}
```

ДОДАТОК К

Лістинг коду класів контролерів OrderController та AdminOrderController мікросервісу orders

```
@AllArgsConstructor
@RestController
@RequestMapping("/user")
public class OrderController {

    OrderService orderService;
    private final OrderMapper orderMapper;

    @GetMapping("/get/all")
    @ResponseBody
    ResponseEntity<?> getAllOrders(@RequestHeader("X-username") String username, @RequestParam(value = "status",
required = false) String status) {
        List<OrderDto> orderList = orderService.getAllOrdersByUsernameAndStatus(username, status)
                .stream().map(orderMapper::toDto).collect(Collectors.toList());

        return ResponseEntity.ok().body(orderList);
    }

    @ResponseStatus(HttpStatus.CREATED)
    @PostMapping("/create")
    @ResponseBody
    OrderDto createOrder(@RequestHeader("X-username") String username, @RequestBody OrderCreateRequest
orderCreateRequest) {
        Order order = orderService.save(username, orderCreateRequest);
        return orderMapper.toDto(order);
    }

    @GetMapping("/get/{id}")
    @ResponseBody
    ResponseEntity<?> getOrder(@RequestHeader("X-username") String username, @PathVariable("id") long id) {
        return ResponseEntity.ok().body(orderMapper.toDto(orderService.getOrderByUsernameAndId(username, id)));
    }

    @PostMapping("/cancel")
    @ResponseBody
    ResponseEntity<?> cancelOrder(@RequestHeader("X-username") String username,
                                  @RequestBody OrderCancelRequest orderCancelRequest) {
        orderService.cancelOrder(username, orderCancelRequest.getId());
        return ResponseEntity.status(204).build();
    }
}


@AllArgsConstructor
@RequestMapping("/admin")
@RestController
public class AdminOrderController {

    OrderService orderService;
    private final OrderMapper orderMapper;

    @GetMapping("/get/all")
```

```
    @ResponseBody
    ResponseEntity<?> getOrders(@RequestBody Order order
    ) {
        List<Order> orderList = orderService.getOrders(order);
        return ResponseEntity.ok().body(orderList);
    }

    @GetMapping("/get/{id}")
    @ResponseBody
    ResponseEntity<?> getOrder(@PathVariable("id") long id) {
        return ResponseEntity.ok().body(orderService.getOrderById(id));
    }

    @PatchMapping("/{id}/cancel")
    @ResponseBody
    ResponseEntity<?> cancelOrder(@PathVariable("id") Long orderId) {
        return ResponseEntity.ok().body(
                orderMapper.toDto(orderService.cancelOrderById(orderId))
        );
    }

    @PatchMapping("/{id}/accept")
    @ResponseBody
    ResponseEntity<?> acceptOrder(@PathVariable("id") Long orderId) {
        return ResponseEntity.ok().body(
                orderMapper.toDto(orderService.acceptOrderById(orderId))
        );
    }
}
```

ДОДАТОК Л

Лістинг коду класів моделей Order та OrderProduct, та перелічення OrderStatus мікросервісу orders

```
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
@Builder
@Entity
@Table(name="`order`")
public class Order {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)
    private Long id;

    @Singular
    @OneToMany(mappedBy = "order", orphanRemoval = true, cascade = CascadeType.ALL)
    private List<OrderProduct> orderProducts = new ArrayList<>();


    @Column(name="details", nullable = true)
    private String details;

    @Column(name = "username", nullable = false, length = 64)
    private String username;

    @Enumerated(EnumType.STRING)
    @Column(name = "status", nullable = false)
    private OrderStatus status;

    @Column(name = "address", nullable = false)
    private String address;

}
```

```
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
@Entity
@Builder
public class OrderProduct {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)
    private Long id;

    @Column(name = "product_id", nullable = false)
    private Long product_id;

    @Column(name = "amount", nullable = false)
```

```
    private int amount;

    @Column(name="price", nullable = false)
    private double price;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "order_id")
    private Order order;

    @Override
    public final boolean equals(Object o) {
        if (this == o) return true;
        if (o == null) return false;
        Class<?> oEffectiveClass = o instanceof HibernateProxy ? ((HibernateProxy)
o).getHibernateLazyInitializer().getPersistentClass() : o.getClass();
        Class<?> thisEffectiveClass = this instanceof HibernateProxy ? ((HibernateProxy)
this).getHibernateLazyInitializer().getPersistentClass() : this.getClass();
        if (thisEffectiveClass != oEffectiveClass) return false;
        OrderProduct that = (OrderProduct) o;
        return getId() != null && Objects.equals(getId(), that.getId());
    }

    @Override
    public final int hashCode() {
        return this instanceof HibernateProxy ? ((HibernateProxy)
this).getHibernateLazyInitializer().getPersistentClass().hashCode() : getClass().hashCode();
    }
}



public enum OrderStatus {
    PENDING,
    IN_PROGRESS,
    REJECTED,
    COMPLETED,
    CANCELLED,
    REFUND
}
```

ДОДАТОК М

Лістинг коду класів сервісів OrderService та EurekaClientService мікросервісу orders

```
@AllArgsConstructor
@Service
public class OrderService {

    private EurekaClientService eurekaClientService;
    private OrderRepository orderRepository;

    private OrderProductRepository orderProductRepository;


    public Order save(Order order) {
        return orderRepository.save(order);
    }

    @Transactional
    public Order save(String username, OrderCreateRequest request) {
        Order order = save(Order.builder()
                .status(OrderStatus.PENDING)
                .username(username)
                .details(request.getDetails())
                .address(request.getAddress())
                .build());


        ProductResponseDto[] responseDtos = eurekaClientService.getProducts(
                request.getOrderProductList().stream().map(OrderProductDto::getProductId).toList()
        );


        List<OrderProduct> orderProducts = mergeProductLists(List.of(responseDtos),
                request.getOrderProductList(), order);


        orderProductRepository.saveAll(orderProducts);

        order.setOrderProducts(orderProducts);
        orderRepository.save(order);

        eurekaClientService.sendNewTelegramMessage(createOrderMessage(order), order.getId());

        return order;
    }

    private String createOrderMessage(Order order) {
        return "Username: \n" + order.getUsername() +
                order.getOrderProducts().stream().map(e ->
                        "\n  ProductId: " + e.getProduct_id() + "\n    Amount: " + e.getAmount()).collect(Collectors.joining())
                + "\nAddress: " + order.getAddress()
                + "\nDetails: " + order.getDetails();
    }

    private static List<OrderProduct> mergeProductLists(List<ProductResponseDto> productResponseList,
                                                        List<OrderProductDto> orderProductList,
```

```
                                        Order order
    ) {
        Map<Long, ProductResponseDto> productResponseMap = productResponseList.stream()
                .collect(Collectors.toMap(ProductResponseDto::getId, product -> product));

        return orderProductList.stream()
                .map(orderProduct -> {
                    ProductResponseDto productResponse = productResponseMap.get(orderProduct.getProductId());
                    if (productResponse != null) {
                        return OrderProduct.builder()
                                .product_id(orderProduct.getProductId())
                                .amount(orderProduct.getAmount())
                                .price(productResponse.getPrice())
                                .order(order)

                                .build();
                    } else {
                        return null;
                    }
                })
                .filter(Objects::nonNull)
                .collect(Collectors.toList());
    }


    public List<Order> getAllOrdersByUsernameAndStatus(String username, String status) {
        return orderRepository.findByIdAndDetailsLikeAndUsernameLikeAndStatus(
                null, null, username,
                status == null ? null : OrderStatus.valueOf(status));
    }

    public Order getOrderByUsernameAndId(String username, Long id) {
        return orderRepository.findByUsernameAndId(username, id).orElseThrow(() -> new
EntityNotFoundException("Order not found"));
    }

    public Order cancelOrder(String username, Long id) {
        Order order = getOrderByUsernameAndId(username, id);
        /*if (order.getStatus() != OrderStatus.PENDING) {
            throw new RuntimeException("Order status is not PENDING");
        }*/
        order.setStatus(OrderStatus.CANCELLED);
        return orderRepository.save(order);
    }


    public List<Order> getOrders(Order order) {
        return orderRepository.findByIdAndDetailsLikeAndUsernameLikeAndStatus(
                order.getId(), order.getDetails(), order.getUsername(), order.getStatus());
    }

    public Order getOrderById(Long id){
        return orderRepository.findById(id).orElseThrow(()->new EntityNotFoundException("Order not found"));
    }

    public Order cancelOrderById(Long id){
        Order order = getOrderById(id);
        order.setStatus(OrderStatus.CANCELLED);
        return orderRepository.save(order);
    }
```

```
    public Order acceptOrderById(Long id){
        Order order = getOrderById(id);
        order.setStatus(OrderStatus.IN_PROGRESS);
        return orderRepository.save(order);
    }


}


@AllArgsConstructor
@Service
public class EurekaClientService {

    private RestTemplate restTemplate;
    private DiscoveryClient discoveryClient;

    public String getServerUrl(String serverId) {
        List<ServiceInstance> instances = discoveryClient.getInstances(serverId);
        if (instances.isEmpty()) {
            throw new RuntimeException("No instances available for " + serverId);
        }
        return instances.getFirst().getUri().toString();
    }

    public ProductResponseDto[] getProducts(List<Long> ids){
        String catalogServerUrl = getServerUrl("catalog");
        return restTemplate.getForObject(catalogServerUrl
                +"/products/search/by-ids?ids="
                + ids.stream().map(String::valueOf).collect(Collectors.joining(",")),

                ProductResponseDto[].class);

    }

    public void sendNewTelegramMessage(String message, Long orderId){
        String url = getServerUrl("telegram");
        restTemplate.postForObject(url, Map.of("message", message, "orderId", orderId), String.class);
    }
}
```

ДОДАТОК Н

Лістинг коду класів налаштувань AuthConfig та CustomUserDetails мікросервісу identity-server

```
@Configuration
@EnableWebSecurity
public class AuthConfig {

    @Bean
    public UserDetailsService userDetailsService(CustomUserDetailsService customUserDetailsService) {
        return customUserDetailsService;
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http
                .csrf(AbstractHttpConfigurer::disable)
                .authorizeHttpRequests(authorize -> authorize
                        .requestMatchers("/auth/register", "/auth/token", "/auth/validate")
                        .permitAll())
                .build();
    }
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration config) throws Exception {
        return config.getAuthenticationManager();
    }

    @Bean
    public AuthenticationProvider authenticationProvider(UserDetailsService userDetailsService) {
        DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();
        authProvider.setUserDetailsService(userDetailsService);
        authProvider.setPasswordEncoder(passwordEncoder());
        return authProvider;
    }
}


public class CustomUserDetails implements UserDetails {

    private final String username;
    private final String password;

    @Enumerated(EnumType.STRING)
    private Role role;

    public CustomUserDetails(UserCredential userCredential) {
        this.username = userCredential.getUsername();
        this.password = userCredential.getPassword();
        this.role = userCredential.getRole();
    }

    @Override
```

```
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return List.of(new SimpleGrantedAuthority(role.name()));
    }

    @Override
    public String getPassword() {
        return password;
    }

    @Override
    public String getUsername() {
        return username;
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }
}
```

ДОДАТОК П

Лістинг коду класу контролера AuthController мікросервісу identity-server

```
@RestController
@RequestMapping("/auth")
@AllArgsConstructor
public class AuthController {

    private final AuthService authService;

    private final UserCredentialService userCredentialService;
    private final AuthenticationManager authenticationManager;
    private final UserCredentialMapper userCredentialMapper;


    @PostMapping("/register")
    public ResponseEntity<?> addNewUser(@RequestBody RegisterDto registerDto) {
        UserCredential userCredential= userCredentialService.saveUser(userCredentialMapper.toEntity(registerDto));

        return new ResponseEntity<>(new JwtResponse(authService.generateToken(userCredential)), HttpStatus.OK); //
todo: refactor it
    }

    @PostMapping("/token")
    public ResponseEntity<?> getToken(@RequestBody AuthRequest authRequest) {

        Authentication authenticate = authenticationManager.authenticate(new
UsernamePasswordAuthenticationToken(authRequest.getUsername(), authRequest.getPassword()));
        if (authenticate.isAuthenticated()) {
            return new ResponseEntity<>(new JwtResponse(authService
                    .generateToken(
                            userCredentialService.findByUsername(authRequest.getUsername())
                    )), HttpStatus.OK);

        } else {
            return new ResponseEntity<>(new BadCredentialsException("Invalid username or password"),
HttpStatus.UNAUTHORIZED);
        }

    }

    @GetMapping("/validate")
    public ResponseEntity<String> validateToken(@RequestParam String token) {
        authService.validateToken(token);

        return ResponseEntity.ok("Token is valid");
    }
}
```

ДОДАТОК Р

Лістинг коду класу моделі UserCredential та перелічення Role мікросервісу identity-server

```
@AllArgsConstructor
@NoArgsConstructor
@Entity
@Getter
@Setter
public class UserCredential {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(nullable = false)
    private Long id;
    @Column(nullable = false, unique = true, length = 64)
    private String username;
    @Column(nullable = false, unique = true, length = 64)
    private String email;
    @Column(nullable = false)
    private String password;

    @Enumerated(EnumType.STRING)
    @Column
    private Role role = Role.USER;

}


public enum Role {
    USER,
    ADMIN
}
```

ДОДАТОК С

Лістинг коду класів сервісів AuthService, CustomUserDetailsService, JwtService та UserCredentialService мікросервісу identity-server

```
@Service
@AllArgsConstructor
public class AuthService {
    private JwtService jwtService;


    public String generateToken(UserCredential userCredential) {

        return jwtService.generateToken(userCredential.getUsername(), userCredential.getRole().name());
    }

    public void validateToken(String token) {
        jwtService.isTokenValid(token);
    }

}



@AllArgsConstructor
@Service
public class CustomUserDetailsService implements UserDetailsService {
    private final UserCredentialService userCredentialService;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        UserCredential credential = userCredentialService.findByUsername(username);
        return new CustomUserDetails(credential);
    }
}



@AllArgsConstructor
@Service
public class JwtService {

    public static final String jwtSigningKey =
"5367566B59703373367639792F423F4528482B4D6251655468576D5A71347437";

    public String generateToken(String username, String role) {
        Map<String, Object> claims = new HashMap<>();
        claims.put("role", role);
        claims.put("username", username);

        return createToken(claims, username);
    }


    private String createToken(Map<String, Object> extraClaims, String userName) {
        return Jwts.builder()
```

```
            .claims(extraClaims)
            .issuedAt(new Date(System.currentTimeMillis()))
            .expiration(new Date(System.currentTimeMillis() + 100000 * 60 * 24))
            .signWith(getSigningKey(), SignatureAlgorithm.HS256)
            .compact();
    }

    public void isTokenValid(String token) {
        Jwts.parser().setSigningKey(getSigningKey()).build().parseSignedClaims(token);
    }

    public String extractUsername(String token) {
        return extractClaim(token, Claims::getSubject);
    }

    private <T> T extractClaim(String token, Function<Claims, T> claimsResolvers) {
        final Claims claims = extractAllClaims(token);
        return claimsResolvers.apply(claims);
    }
    private boolean isTokenExpired(String token) {
        return extractExpiration(token).before(new Date());
    }

    private Date extractExpiration(String token) {
        return extractClaim(token, Claims::getExpiration);
    }

    private Claims extractAllClaims(String token) {
        return Jwts.parser().setSigningKey(getSigningKey()).build().parseSignedClaims(token).getPayload();
    }

    private Key getSigningKey() {
        byte[] keyBytes = Decoders.BASE64.decode(jwtSigningKey);
        return Keys.hmacShaKeyFor(keyBytes);
    }
}

@AllArgsConstructor
@Service
public class UserCredentialService {
    private PasswordEncoder encoder;
    private UserCredentialRepository repository;

    public UserCredential findByUsername(String username) {
        return repository.findByUsername(username).orElseThrow(()->new RuntimeException("User not found with
username: " + username) );
    }

    public UserCredential saveUser(UserCredential credential) {
        credential.setPassword(encoder.encode(credential.getPassword()));
        credential.setRole(Role.USER);
        return repository.save(credential);
    }
}
```

ДОДАТОК Т

Лістинг коду класів файлу bot.py python сервісу

```
import logging
import sys

from aiohttp import web

from aiogram import Bot, Dispatcher, Router, F
from aiogram.client.default import DefaultBotProperties
from aiogram.enums import ParseMode
from aiogram.filters import Filter
from aiogram.types import Message
from aiogram.webhook.aiohttp_server import SimpleRequestHandler, setup_application
from py_eureka_client.eureka_client import EurekaClient


class OrderCommandFilter(Filter):

    def __init__(self, command_start_name: str) -> None:
        self.command_start_name = command_start_name

    async def __call__(self, message: Message) -> bool:
        return message.text.startswith(self.command_start_name)


def create_router(eureka_client_instance: EurekaClient):
    router = Router()

    def get_order_id_from_message(message: Message) -> str:
        return message.text.split("_")[1]

    @router.message(OrderCommandFilter("/accept"))
    async def accept_command_handler(message: Message) -> None:
        order_id = get_order_id_from_message(message)
        res = await eureka_client_instance.do_service("orders", f"/admin/{order_id}/accept", method="PATCH")
        print(res)

    @router.message(OrderCommandFilter("/decline"))
    async def decline_command_handler(message: Message) -> None:
        order_id = get_order_id_from_message(message)
        res = await eureka_client_instance.do_service("orders", f"/admin/{order_id}/cancel", method="PATCH")
        print(res)


class TelegramBot:
    def __init__(self,
                 token: str,
                 webhook_url: str,
                 webhook_path: str,
                 webhook_secret: str,
                 web_server_port: int,
                 web_server_host: str,
                 telegram_chat_ids: list = None,
```

```
        ) -> None:

    self.token = token
    self.webhook_url = webhook_url
    self.webhook_path = webhook_path
    self.webhook_secret = webhook_secret
    self.web_server_host = web_server_host
    self.web_server_port = web_server_port

    if telegram_chat_ids is None:
        self.telegram_chat_ids = []
    else:
        self.telegram_chat_ids = telegram_chat_ids

async def on_startup(self, bot: Bot) -> None:

    await bot.set_webhook(f"{self.webhook_url}{self.webhook_path}",
                          secret_token=self.webhook_secret)

def initialize(self, eureka_client_instance) -> None:

    router = create_router(eureka_client_instance)

    bot = Bot(token=self.token, default=DefaultBotProperties(parse_mode=ParseMode.HTML))
    # Dispatcher is a root router
    dp = Dispatcher()
    dp.include_router(router)
    dp.startup.register(self.on_startup)
    async def handle_info(request):
        return web.Response(text="Info from python!")

    def create_commands(order_id: int):
        return "\n/accept_" + str(order_id) + "\n/decline_" + str(order_id)

    async def handle_message(request: web.Request) -> web.Response:
        value = await request.json()
        for telegram_id in self.telegram_chat_ids:
            await bot.send_message(chat_id=telegram_id, text=value["message"] + create_commands(
                value["orderId"]), )
        return web.Response(status=204)
    app = web.Application()
    app.router.add_get("/info", handle_info)
    app.router.add_post("", handle_message)

    webhook_requests_handler = SimpleRequestHandler(
        dispatcher=dp,
        bot=bot,
        secret_token=self.webhook_secret,
    )
    webhook_requests_handler.register(app, path=self.webhook_path)
    setup_application(app, dp, bot=bot)
    web.run_app(app, host=self.web_server_host, port=self.web_server_port)
if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO, stream=sys.stdout)
```