НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут атомної та теплової енергетики

Кафедра цифрових технологій в енергетиці

«До захисту допущено»
Завідувач кафедри
_____ Наталія АУШЕВА
“___”_____2025 р.

Дипломна робота
на здобуття ступеня бакалавр

За освітньою програмою “Цифрові технології в енергетиці”

Спеціальності 122 “Комп’ютерні науки”

на тему: “ Платформа для електронної комерції (backend)”

Виконав: студент 4 курсу, групи ТР-11
Шусть Кирило Євгенійович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник проф. д.т.н, Олексій ШУШУРА _____
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Рецензент проф. каф. ІПЗЕ, проф. д.т.н., Євген ГАВРИЛКО _____
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Н.контроль ст.викладач Ольга БЕСПАЛА _____
(посада, ім’я, ПРІЗВИЩЕ) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ - 2025

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

Рівень вищої освіти – перший (бакалаврський)
спеціальність 122 “Комп’ютерні науки”
Освітньо-професійна програма “Цифрові технології в енергетиці”

ЗАТВЕРДЖУЮ
Завідувач кафедри ЦТЕ
_____ Наталія АУШЕВА
(підпис)
“___” _____2025 р.

ЗАВДАННЯ
на дипломну роботу студенту

Шустю Кирилу Євгенійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи “Платформа для електронної комерції (backend)”
Науковий керівник Шушура Олексій Миколайович, д.т.н, проф.
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від “02” червня 2025 року № 1875-с

2. Термін подання студентом роботи 09.06.2025

3. Вихідні дані до роботи мова програмування Java, фреймворки Spring та Swagger, СКБД PostgreSQL, пошуковий двигун Elasticsearch, Caffeine, середовище розробки IntelliJ Idea, інструмент для тестування Postman

4. Перелік питань, які потрібно розробити _____

1) провести аналіз існуючих веб-систем для електронної комерції в Україні
2) визначити головні функції веб-застосунку, що керує послугами та ресурсами
3) аргументувати вибрані інструменти, технології та алгоритми для реалізації програмного забезпечення
4) побудувати архітектури програмного забезпечення, баз даних та пошукового

двигуна
5) створити прикладний програмний веб-інтерфейс
6) представити процес застосування веб-інтерфейсу

5. Орієнтований перелік ілюстративного матеріалу діаграми, що демонструють роботу алгоритмів веб-інтерфейсу, архітектуру системи, бази даних та пошукового двигуна, взаємодію користувачів із системою, класи програмного забезпечення; приклади роботи з програмним продуктом.

6. Дата видачі завдання 13.09.2024

КАЛЕНДАРНИЙ ПЛАН

/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Затвердження теми роботи	13.09.2024	Виконано
2.	Вивчення та аналіз задачі	17-20.04.25	Виконано
3.	Розробка архітектури та загальної структури системи	21-25.04.25	Виконано
4.	Розробка частин окремих підсистем	25.04-02.05.25	Виконано
5.	Програмна реалізація системи	03-07.05.25	Виконано
6.	Оформлення пояснювальної записки	08-12.05.25	Виконано
7.	Захист програмного продукту	12-16.05.25	Виконано
8.	Передзахист	26-28.05.25	Виконано
9.	Подання готової роботи на кафедру	06.06.2025	Виконано
10.	Захист	16-20.06.2025	Виконано

Студент _____ **Кирило ШУСТЬ**
(підпис) (ім'я, ПРІЗВИЩЕ)

Керівник роботи _____ **Олексій ШУШУРА**
(підпис) (ім'я, ПРІЗВИЩЕ)

АНОТАЦІЯ

Дипломна робота виконана на 57 сторінках, містить 9 ілюстрацій, 1 таблицю, 1 додаток, 20 джерел в переліку посилань.

Мета роботи – створення backend-частини платформи електронної комерції, що реалізує функціональну, захищену та масштабовану інфраструктуру для взаємодії між продавцями та покупцями, з інтеграцією повнотекстового пошуку, перевірки даних та підтримкою модульного розширення системи.

Методи та засоби: мова програмування Java, фреймворки Spring та Swagger, СКБД PostgreSQL, пошуковий двигун Elasticsearch, Caffeine, середовище розробки IntelliJ Idea, інструмент для тестування Postman.

Результат – розроблена серверна частина комплексної платформи електронної комерції, яка забезпечує обробку запитів, управління даними, повнотекстовий пошук за допомогою Elasticsearch. Система орієнтована на прозорість, масштабованість та безпеку, відповідає реальним вимогам українського e-commerce і здатна адаптуватися як до невеликих онлайн-магазинів, так і до великих торгових мереж.

Ключові слова: ЕЛЕКТРОННА КОМЕРЦІЯ, BACKEND, SEARCH, RETAIL, СЕРВЕРНИЙ ЗАСТОСУНОК.

ABSTRACT

The thesis is completed on 57 pages, contains 9 illustrations, 1 table, 1 appendix, 20 sources in the list of references.

The purpose of the work is to create a backend part of an e-commerce platform that implements a functional, secure and scalable infrastructure for interaction between sellers and buyers, with the integration of full-text search, data validation and support for modular system expansion.

Methods and tools: Java programming language, Spring and Swagger frameworks, PostgreSQL database, Elasticsearch search engine, Caffeine, IntelliJ Idea development environment, Postman testing tool.

The result is a developed server part of a comprehensive e-commerce platform that provides query processing, data management, full-text search using Elasticsearch. The system is focused on transparency, scalability and security, meets the real requirements of Ukrainian e-commerce and is able to adapt to both small online stores and large retail chains.

Keywords: E-COMMERCE, BACKEND, SEARCH, SECTION, SERVER APPLICATION.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНИХ УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

СУБД – Система управління базами даних

Кастомний – створений або налаштований відповідно до специфічних потреб, вимог чи побажань

Логування – збирання та збереження інформації про роботу системи у вигляді логів (журналів подій)

Патерн (англ. pattern) – усталене рішення типової задачі програмування, що описує загальний підхід до побудови програмного компонента або архітектури

Стартап (англ. startup) – новостворене підприємство або проєкт, що перебуває на початковому етапі розвитку та зазвичай орієнтоване на інноваційні продукти або технології

Агрегація – процес об'єднання даних з різних джерел або об'єктів з метою подальшої обробки чи аналізу

ВСТУП

Сучасні веб-системи потребують не лише зручного та інтуїтивно зрозумілого інтерфейсу, а й забезпечення високої продуктивності, стійкості до навантаження, масштабованості та надійності. У зв’язку з цим особливої важливості набуває розробка програмних рішень, здатних ефективно функціонувати в умовах високого навантаження, обробляти значні обсяги даних і одночасно забезпечувати безперебійну взаємодію з великою кількістю користувачів.

Дипломна робота спрямована на створення повнофункціональної веб-системи з урахуванням вимог до продуктивності, безпеки та масштабованості. Розробка передбачала побудову багаторівневої архітектури із чітким розподілом обов’язків між модулями, застосування сучасних бекенд-технологій, зокрема Java Spring, і впровадження інфраструктури для повноцінного локального та хмарного розгортання.

У процесі виконання дипломної роботи було реалізовано основну програмну логіку ключових компонентів системи, включно з механізмами обробки, трансформації, зберігання та захисту даних. Особливу увагу приділено питанням безпеки взаємодії користувачів із системою, а також забезпеченню відмовостійкості за допомогою вбудованих засобів моніторингу та логування.

З метою аналізу навантаження та продуктивності до системи інтегровано модулі збору та аналізу метрик, що дозволяє виявляти вузькі місця та здійснювати оптимізацію в реальному часі. Було розроблено інструменти для автоматизованого тестування функціоналу, що забезпечило високу якість програмного коду.

Таким чином, у межах дипломної роботи було не лише продемонстровано глибоке розуміння теоретичних основ розробки веб-систем, а й реалізовано повноцінний програмний продукт, здатний задовольнити сучасні вимоги до надійності, функціональності та масштабованості інформаційних сервісів.

1 ПОСТАНОВКА ЗАДАЧІ СТВОРЕННЯ СИСТЕМИ ОНЛАЙН-КОМЕРЦІЇ

З розвитком цифрових технологій та зміною споживчої поведінки електронна комерція стала невіддільною частиною економіки і продовжує набирати обертів. В Україні ринок e-commerce активно зростає, особливо в умовах трансформації бізнесу в онлайн-просторі. В зв’язку з цим, створення масштабованих, високонадійних і зручних веб-систем, що дозволяють ефективно реалізовувати торговельну діяльність у мережі, набувають популярності. Рішення з високою надійністю вимагає швидкої обробки великого обсягу даних для задовлення потреб користувачів.

1.1 Задачі роботи

Необхідно провести аналіз предметної області з метою виявлення недоліків, обмежень та проблем існуючих систем електронної комерції, які функціонують на території України. На основі результатів цього аналізу доцільно сформулювати функціональні та нефункціональні вимоги до нової веб-системи, ґрунтуючись на актуальних потребах цільової аудиторії. Визначені вимоги слугують основою для того, щоб спроєктувати архітектуру системи, включно з визначенням її основних модулів, взаємодії між ними, а також обраної технологічної стеку (з урахуванням масштабованості, безпеки та продуктивності). Після завершення етапу проєктування постає завдання обрати засоби реалізації системи, які найкраще відповідають поставленим цілям. Завершальним етапом є розробити програмне забезпечення системи та провести його тестування, що дозволяє переконатися в коректності та надійності її функціонування.

Основна мета даного проєкту полягає в розробці функціонального веб-ресурсу для електронної комерції з врахуванням сучасних вимог до продуктивності, безпеки та підтримки модульної архітектури. Вимоги системи – ефективне управління замовленнями, надійна обробка та збереження даних, а також можливість масштабування під зростаючі навантаження.

1.2 Орієнтири розвитку

Орієнтир розвитку – максимальна швидкість обровки запитів при максимально високих навантаженнях на систему. Ціль – створення потужного API-застосунку, який використовуватиметься клієнтським веб-застонком, котрий є презентаційною частиною цієї системи. API має бути чітко структурованим, з підтримкою асинхронного обміну даними, обробкою великої кількості одночасних запитів, а також інтеграцією механізмів кешування для зменшення навантаження на базу даних. Паралельно має реалізовуватись система моніторингу, яка дозволяє оперативно відстежувати продуктивність, виявляти вузькі місця та аналізувати ефективність роботи ключових компонентів. Це дасть змогу своєчасно реагувати на потенційні проблеми та забезпечувати безперервність роботи системи.

Функціональність повинна бути масштабованою: можливість додавання нових модулів або мікросервісів без ризику впливу на вже реалізовану логіку. Така гнучка архітектура дозволить адаптувати систему до мінливих вимог ринку та потреб бізнесу, забезпечуючи її довготривалу життєздатність. Окрім того, це відкриває можливості для поетапного впровадження нових технологій і покращень без необхідності повного перероблення існуючих компонентів.

1.3 Архітектурні принципи системи

Система реалізується з урахуванням принципів модульності, слабкої зв'язаності та високої згуртованості компонентів. Це дозволить підвищити читабельність коду, спростити тестування, обслуговування та подальший розвиток. Впровадження багатошарової архітектури забезпечує логічне розділення відповідальностей між рівнями даних, бізнес-логіки, контролю доступу та шару представлення. Завдяки такому підходу кожен шар може розвиватися і тестуватися незалежно, що підвищує якість і стабільність системи.

Також передбачається використання модульної монолітичної структури, що дозволяє організувати проект у вигляді окремих, чітко визначених модулів із власною відповідальністю. Така архітектура забезпечує легкість навігації в коді та покращує масштабованість.

В подальшому, із зростанням вимог до системи та збільшенням навантаження, можливо здійснити поступовий перехід до мікросервісного підходу, що дозволить досягти більшої гнучкості при розгортанні та масштабуванні окремих сервісів.

Впровадження стандартів розробки, таких як інверсія контролю (IoC), залежності через інтерфейси та застосування патернів проектування, допоможуть знизити зв'язаність між компонентами та покращити їх тестованість. Це створить надійний фундамент для підтримки життєвого циклу програмного забезпечення, дозволяючи оперативно впроваджувати нові функції та виправляти помилки без ризику порушення існуючої логіки.

Особливу увагу приділено також логуванню, моніторингу та обробці винятків, що сприятиме швидкому виявленню та усуненню проблем у роботі системи, забезпечуючи її стабільність та безперервність функціонування.

1.4 Захист та безпека даних

Однією з ключових вимог до сучасної e-commerce системи є захист даних користувачів. У проєкті буде реалізовано просунуту систему аутентифікації та авторизації з використанням JWT-токенів, що дозволить забезпечити безпечний та ефективний обмін інформацією між клієнтом і сервером без необхідності зберігання стану сесії на сервері. Шифрування конфіденційної інформації, такої як паролі та персональні дані, буде виконуватись з використанням сучасних криптографічних алгоритмів, що підвищить загальний рівень безпеки системи.

Обмеження доступу до критичних API-ресурсів буде реалізовано за допомогою механізмів контролю доступу на основі ролей (RBAC), що дозволить гнучко налаштовувати права користувачів залежно від їхніх ролей і рівня повноважень. Такий підхід значно знизить ризик несанкціонованого доступу та допоможе дотримуватися принципу найменших привілеїв.

Окрім того, буде впроваджено захист від поширених веб-атак, серед яких SQL-ін’єкції, які можуть порушити цілісність бази даних, атаки міжсайтового виконання скриптів (XSS), що загрожують безпеці користувацьких сесій, та Cross-Site Request Forgery (CSRF), що можуть використовуватись для виконання небажаних дій від імені користувача. Використання сучасних фреймворків та бібліотек, які автоматично обробляють ці загрози, разом із власними додатковими заходами безпеки, дозволить гарантувати стабільність та надійність платформи.

Впровадження цих заходів не лише захистить персональні дані користувачів, але й зміцнить довіру до системи, що є критично важливим фактором успішної роботи будь-якого e-commerce проєкту.

1.5 Забезпечення цілісності та синхронізації даних

У межах високонавантаженої e-commerce системи критично важливою є гарантія того, що всі модулі працюють з актуальними та синхронізованими даними. У роботі передбачається використання стратегій використання пошукового двигуна для швидкої агрегації результатів на переважно великих вибірках. Дані для пошукового двигуна будуть синхронізуватись з базою даних з певним інтервалом – у разі, якщо дані з бази даних оновлюються, нові дані потрібно додати до пошукового двигуна, щоб впровадити основні функції пошуку на вибірці з актуальною інформацію. Також, планується використання подієво-орієнтованої архітектури для адаптування різних типів даних до одного – більш загального. Це допоможе усунути проблеми з подальшим розширенням модулів системи і прискорить розробку.

Також планується реалізація засобів контролю версій об’єктів та відстеження змін, що дозволить усувати помилки при паралельному редагуванні даних, вчасно виявляти розбіжності між модулями й підтримувати цілісність інформаційного простору всієї системи. Такий підхід гарантує стабільність бізнес-логіки при активній взаємодії користувачів, адміністраторів та зовнішніх сервісів з різних точок доступу.

Окремо буде впроваджено механізми логування та аудиту змін, що дозволить відстежувати історію оновлень всередині застосунку і швидко виявляти можливі проблеми в роботі системи. Це допоможе підтримувати якість даних і полегшить діагностику у разі виникнення помилок або конфліктів під час синхронізації між різними модулями.

2 АНАЛІЗ ІНФОРМАЦІЙНИХ ПЛАТФОРМ ЕЛЕКТРОННОЇ КОМЕРЦІЇ УКРАЇНИ

На ринку електронної комерції України представлені різні платформи зі своїми особливостями.

Rozetka та Prom.ua є універсальними інтернет-магазинами, орієнтованими на масову торгівлю від постачальників.

Olx.ua виконує функцію маркетплейсу для приватних осіб і має риси онлайн-версії «блошиного» ринку.

Hotline.ua спеціалізується на порівнянні цін і характеристик товарів з різних магазинів.

Epicentrk.ua розвиває електронну комерцію в межах своєї офлайн-мережі, зосереджуючись на товарах для ремонту та дому.

Також функціонують локальні гравці, як-от Allo.ua чи Comfy.ua, що поєднують онлайн-торгівлю з роздрібними магазинами.На рисунку 2.1 зображено таблицю трафіку популярних мереж ринку електронної комерції України [1].

№	Сайт	Щомісячний трафік	Частка десктоп/мобільний	Основне джерело трафіку	Зміна MoM / YoY
1	**olx.ua**	72,08 млн	34,09% / 65,91%	Прямий трафік	↑8,7% / ↓5,65%
2	**rozetka.com.ua**	57,76 млн	33,59% / 66,41%	Прямий трафік	↑3,98% / ↓12,07%
3	**prom.ua**	36,85 млн	29,57% / 70,43%	Прямий трафік	↑7,43% / ↓15,67%
4	**tabletki.ua**	27,33 млн	15,22% / 84,78%	Прямий трафік	↓7,55% / ↓15,19%
5	**epicentrk.ua**	22,52 млн	25,45% / 74,55%	Пошуковий трафік	↑8,53% / ↓12,92%
6	**aliexpress.com**	12,06 млн	57,66% / 42,34%	Прямий трафік	↑8,88% / ↑27,8%
7	**makeup.com.ua**	11,32 млн	21,01% / 78,99%	Прямий трафік	↑8,86% / ↓26,78%
8	**comfy.ua**	9,97 млн	26,79% / 73,21%	Прямий трафік	↑21,23% / ↑20,52%
9	**hotline.ua**	9,54 млн	32,52% / 67,48%	Прямий трафік	↓5,09% / ↓4,93%
10	**eva.ua**	7,4 млн	18,69% / 81,31%	Прямий трафік	↓2,27% / ↓12,13%

Рисунок 2.1 – Трафік електронної комерції України за 2025 рік

MoM (Month-over-Month) – це порівняння змін показників у відсотковому вираженні за поточний місяць порівняно з попереднім. Наприклад, якщо у березні 2025 року трафік становив 72,08 млн, а в лютому – приблизно 66,31 млн, то

зростання MoM для olx.ua склало +8,7%. Аналогічно, YoY (Year-over-Year) – це зміна показників за рік: березень 2025 року порівнюється з березнем 2024 року. Наприклад, якщо трафік olx.ua у березні 2024 року був приблизно 76,38 млн, а в березні 2025 – 72,08 млн, то це означає спад YoY на –5,65%.

Базуючись на даних за березень 2025 року серед трьох найпопулярніших рішень, OLX.ua залишається безперечним лідером за щомісячним трафіком – 72,08 млн відвідувань, що демонструє його домінування як платформи для купівлі-продажу між користувачами. Водночас, YoY показує спад на 5,65%, що може свідчити про зниження загального інтересу користувачів порівняно з минулим роком. Ріст MoM на 8,7% вказує на позитивну динаміку протягом останнього місяця, ймовірно завдяки сезонним чинникам або рекламній активності.

Rozetka.com.ua у березні мала 57,76 млн відвідувань – MoM зростання на 3,98% говорить про стабільне покращення місячного трафіку, але водночас спостерігається YoY спад на 12,07%, що може бути тривожним сигналом у контексті конкуренції або зниження купівельної активності.

Prom.ua показує MoM зростання на 7,43%, тобто суттєво покращив свої позиції за останній місяць, хоч і має YoY спад на 15,67%, що також вказує на довготривалий тренд спаду інтересу до майданчика.

Отже, MoM дозволяє швидко відстежити короткострокові коливання трафіку – як позитивні, так і негативні. Це корисно для оцінки ефективності маркетингових кампаній, акцій чи сезонних продажів. Натомість YoY показує довгострокову динаміку й може вказувати на структурні зміни в популярності платформи або зміну поведінки споживачів. У випадку OLX – сильне MoM зростання і слабке YoY падіння свідчать про поточну стабільність, але з потенційним довгостроковим викликом. У Rozetka YoY падіння суттєвіше, ніж у OLX, а в Prom.ua – ще більше, що вказує на потребу адаптації до змін ринку.

2.1 Переваги існуючих платформ

Rozetka – це флагман українського e-commerce, який поєднує риси маркетплейсу та класичного інтернет-магазину. Платформа має високий рівень довіри серед покупців завдяки рокам стабільної роботи. Серед її переваг: відомий бренд і стабільна аудиторія, широкий асортимент товарів – від техніки до продуктів харчування, власна логістика з точками видачі у більшості міст, зручний інтерфейс і мобільний застосунок, потужні маркетингові кампанії, а також система бонусів і акцій для залучення клієнтів.

Prom.ua – один із найбільших маркетплейсів України, що дає можливість продавцям малого та середнього бізнесу запускати власні онлайн-магазини. Його сильні сторони включають велику кількість продавців і товарів, просту реєстрацію та легкий старт, хорошу SEO-оптимізацію для товарів, доступ до послуг типу реклами, CRM та інтеграцій, високий мобільний трафік і систему внутрішніх рейтингів та відгуків, що формують довіру до продавців.

OLX.ua – найбільша платформа оголошень в Україні, яка орієнтується на моделі C2C (користувач для користувача), але також використовується бізнесами. Основні переваги: велика аудиторія користувачів, зручне створення оголошень без технічних знань, можливість безкоштовного розміщення у багатьох категоріях, підтримка мобільного додатку з чатом, охоплення широкого спектру товарів і популярність у містах і регіонах.

2.2 Недоліки існуючих платформ

Rozetka, попри свій масштаб і впізнаваність, може бути складною платформою для малого бізнесу. Серед недоліків – жорсткі вимоги до продавців, серйозна конкуренція, висока комісія та плата за розміщення, обмеження щодо брендування, а також той факт, що Rozetka сама є активним продавцем і може бути прямим конкурентом для партнерів.

Prom.ua має недоліки, пов’язані зі слабшим контролем якості: на платформі трапляються несумлінні продавці або товари сумнівної якості. Окрім цього, інтерфейси магазинів можуть відрізнятись і бути не завжди зручними, велика кількість схожих пропозицій ускладнює вибір, продавці часто повинні самі займатися просуванням, а також немає єдиної системи повернення чи обслуговування – усе залежить від конкретного магазину.

OLX.ua також має свої обмеження, які випливають із його специфіки. Платформа не надає жодних гарантій, що підвищує ризик шахрайства. Висока кількість фейкових оголошень, мінімальні можливості для масштабування бізнесу, слабкий контроль за якістю контенту, обмежені інструменти для просування, відсутність аналітики та маркетингових функцій – усе це робить OLX менш зручним для професійної e-commerce діяльності.

2.3 Переваги очікуваного продукту понад іншими платформами

Проаналізувавши платформи-конкуренти, можна виокремити кілька переваг розроблюваного продукту. Перш за все, пропонується прозора та фіксована комісія без прихованих платежів, що дозволяє знизити витрати для малого бізнесу, на відміну від високих тарифів, характерних для Rozetka. Додатково, передбачена можливість вільного брендування магазину та його візуального оформлення, що сприятиме підвищенню впізнаваності продавця, тоді як на Rozetka такі можливості обмежені.

Система також забезпечує уніфікований і послідовний інтерфейс для всіх продавців, що підвищує зручність і передбачуваність користувацького досвіду. Це контрастує з платформою Prom.ua, де дизайн магазинів суттєво різниться. Інструменти аналітики та звітності, що будуть доступні користувачам, дозволять приймати обґрунтовані бізнес-рішення та ефективно відстежувати продажі – на відміну від OLX.ua, де подібна аналітика відсутня.

Особливу увагу приділено безпеці: обов’язкова верифікація продавців за документами створює довіру та знижує ризики шахрайства, що є недоліком на Prom.ua, де контроль слабший. Крім того, платформа поєднує можливості як роздрібної, так і рітейлерської торгівлі: звичайний користувач зможе продати будь-яку річ без необхідності сплачувати високі тарифи, маючи опцію проходження верифікації.

Підсумовуючи, очікувана платформа вибудована навколо реальних болючих точок українського e-commerce: тут строго відбираються та верифікуються продавці, щоб виключити шахрайство та фейкові оголошення, покупці отримують чіткі гарантійні умови та прозорі правила повернення, а продавці – фіксовану та передбачувану комісію без прихованих платежів.

Візуальне оформлення магазину можна вільно налаштовувати під власний бренд, інтерфейс уніфікований для всіх учасників, а вбудовані інструменти просування знижують потребу у зовнішньому маркетингу. Платформа легко масштабується від невеликих магазинів до великих мереж, має потужну аналітику та єдину службу підтримки для швидкого вирішення будь-яких питань. Завдяки цьому можливо отримати довірчий, безпечний та зручний майданчик, який водночас відповідає базовим очікуванням і вирізняється з-поміж існуючих рішень на ринку.

Додатково до цього, в майбутньому передбачено інтергацію з найпоширенішими платіжними шлюзами, службами доставки, CRM та бухгалтерськими системами без необхідності розробки власних «містків», що скорочує час виходу на ринок. Організація ролей із гнучким доступом до різних модулів підтримує складні внутрішні процеси великих торгових команд, а дорожня карта передбачає поступове впровадження машинного навчання для персоналізованих рекомендацій і прогнозування попиту, що сприятиме оптимізації товарних запасів і підвищенню ефективності продажів.

3 ВИБІР ЗАСОБІВ РОЗРОБКИ

За основу, в якості мови програмування взято Java. В проєкті використовується Spring Boot для створення серверного веб-застосунку з доступом до бази даних і валідацією. В якості бази даних виступає PostgreSQL [2], а також Elasticsearch. Elasticsearch – для пошуку. Caffeine (кеш в пам'яті застосунку для малого об’єму даних) та Redis відповідає за кешування. Також, присутня велика кількість додаткових бібліотек, які зменшують кількість boiler-plate коду. Lombok спрощує написання коду використовуючи специфікації java за допомогою яких він динамічно модифікує класи. PipelinR реалізує патерн [3] медіатора для event-driven архітектури [4]. Для мапінгу даних використовуються ModelMapper і MapStruct.

3.1 Мова програмування Java

Java – це мова програмування та обчислювальна платформа, яка вперше з’явилася у 1995 році завдяки компанії Sun Microsystems. Від свого скромного старту вона виросла до передової технології цифрового світу, слугуючи надійною основою для створення безлічі сервісів і застосунків. Навіть сьогодні багато сучасних цифрових продуктів і інновацій спираються на Java у своїй розробці.

Не дивлячись на те, що сучасні Java-застосунки зазвичай містять вбудоване середовище виконання, ще існують програми та деякі вебсайти, які не працюватимуть без встановленої Java на комп’ютері. Сайт Java.com орієнтований на користувачів, яким досі потрібна Java для запуску робочих програм, особливо тих, що базуються на Java 8. Тим, хто хоче вивчати Java або займається розробкою, варто звернутися до сайту dev.java, а бізнес-користувачам – до oracle.com/java для отримання додаткової інформації.

Java є платформонезалежною, оскільки вихідний код Java компілюється у байт-код, який потім може виконуватись на будь-якій платформі за допомогою

віртуальної машини Java (JVM) [5]. Java також відома як мова WORA (write once, run anywhere – написано один раз, запускається всюди), саме завдяки своїй платформонезалежності. Крім того, розробка більшості Java-додатків відбувається в середовищі Windows, тоді як запуск часто здійснюється на платформі UNIX, і це можливо саме через платформонезалежну природу Java.

Екосистема Java пропонує широкий спектр бібліотек, фреймворків та API, які пришвидшують розробку та скорочують обсяг шаблонного коду. Популярні фреймворки, такі як Spring, Hibernate та Maven, допомагають розробникам створювати все: від простих веб-додатків до масштабних корпоративних систем. Ця багата екосистема робить Java потужним інструментом для вирішення практично будь-яких завдань розробки програмного забезпечення.

3.2 СУБД PostgreSQL

PostgreSQL – це реляційна система управління базами даних з відкритим вихідним кодом, яка поєднує в собі силу реляційної бази даних та об’єктно-орієнтованого програмування.

Об'єктно-орієнтовані функції – це можливості, які дозволяють PostgreSQL керувати даними з більшою складністю та гнучкістю. Деякі з найбільш використовуваних та важливих можливостей:

Налаштовувані типи даних: Користувачі можуть визначати власні типи даних, що дозволяє створювати точніші та складніші структури даних, адаптовані до потреб конкретних програм.

Успадкування таблиць: Таблиці можуть успадковувати властивості з інших таблиць, що полегшує повторне використання структур таблиць та спрощує керування ієрархічними даними [6].

Функції та процедури: PostgreSQL підтримує створення функцій та збережених процедур, які можна писати різними мовами програмування, що покращує здатність бази даних обробляти складні операції.

Підтримка JSON: Широка підтримка типів даних JSON дозволяє PostgreSQL ефективно обробляти напівструктуровані дані, скорочуючи розрив між реляційними та документо-орієнтованими базами даних.

Повнотекстовий пошук: PostgreSQL пропонує надійні можливості повнотекстового пошуку, що дозволяє ефективно виконувати операції пошуку текстових даних.

Не всі бази даних однакові. Вибір бази даних визначає, наскільки швидко та точно можливо зберігати, отримувати та обробляти інформацію, особливо зі зростанням масштабу та складності проектів.

PostgreSQL є потужною системою керування базами даних, яка підходить для широкого спектра застосувань, зокрема в галузі штучного інтелекту та обробки даних. Вона дозволяє ефективно зберігати й керувати великими обсягами інформації, що особливо важливо для складних комерційних систем та побудови сховищ даних. Завдяки підтримці SQL мови запитів, PostgreSQL дає змогу виконувати складні запити, зокрема об’єднання таблиць, підзапити та віконні функції, що значно полегшує пошук і аналіз інформації. Крім того, її можливості з обробки складних структур даних дозволяють гнучко трансформувати та готувати дані безпосередньо в базі. Підтримка користувацьких функцій, збережених процедур і тригерів відкриває можливість виконання складної аналітики на рівні СУБД, без необхідності винесення логіки в окремі сервіси.

Однією з причин популярності PostgreSQL у сфері e-commerce є його широкі можливості для обробки даних. Він дозволяє ефективно управляти інформацією про товари, замовлення, клієнтів, склади тощо. Завдяки підтримці складних запитів і аналітичних функцій, можна швидко формувати звіти, сегментувати аудиторію, відслідковувати тенденції продажів і реалізовувати персоналізовані рекомендації.

Додаткову цінність створюють розширені функції, такі як користувацькі процедури, тригери й уявлення, які дозволяють вбудовувати складну логіку прямо в базу. Це корисно, наприклад, при реалізації бізнес-правил для обліку знижок, розрахунку податків чи синхронізації складів.

Ще однією важливою перевагою PostgreSQL є підтримка транзакційності за принципами ACID [7]. У сфері електронної комерції це критично – кожна транзакція, що пов'язана з покупкою або оновленням залишків на складі, повинна бути оброблена повністю й без помилок, навіть у випадку збоїв або одночасного доступу до даних. PostgreSQL гарантує збереження цілісності інформації, що є основою довіри клієнтів та стабільності бізнесу.

На фоні інших СУБД PostgreSQL вигідно вирізняється своєю розширюваністю. Можливість додавати власні типи даних, функції та модулі дозволяє адаптувати систему до конкретних бізнес-потреб, зокрема для складних облікових моделей, багатомовної підтримки або мультивалютної логіки.

Попри всі переваги, варто зважати, що освоєння PostgreSQL може зайняти трохи більше часу, ніж у випадку з простішими базами, але це швидко окупається завдяки потужності та стабільності платформи. Існує велика спільнота розробників і користувачів, що постійно підтримують і розвивають систему, а також численні готові рішення та плагіни, що спрощують інтеграцію з різними інструментами.

Якщо планується розвиток інтернет-магазину, розширення асортименту, аналітика поведінки клієнтів чи інтеграція з іншими платформами – PostgreSQL стане надійною основою, яка легко масштабується разом із проєктом. Його обирають як стартапи, так і великі гравці на ринку, оскільки він поєднує функціональність комерційних рішень із доступністю open-source. Завдяки підтримці стандартів SQL і можливості використання сучасних технологій, таких як реплікація, кластеризація та резервне копіювання, PostgreSQL забезпечує високу надійність і безпеку збереження даних.

4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

Загально, архітектуру застосунку розподілено між двома ключовими сервісами – сервіс роздрібної та масштабної торгівлі від постачальників, за котрі відповідають модулі «Flea» та «Retailer», відповідно. Окрім цього, база даних тепер відповідає лише за вставку і видалення даних, за агрегацію та їх пошук тепер відповідає Elasticsearch, що кратно зменшує навантаження на всю систему та прискорює її роботу. Дані для пошуку з певним інтервалом часу завантажуються з бази даних в пошуковий двигун, щоб актуалізувати нові дані від користувачів.

4.1 Система управління товарами маркетплейсів

Розроблена програмна система покликана вирішити завдання централізованого управління товарами для двох окремих, але концептуально схожих за логікою функціонування типів маркетплейсів – так званого блошиного ринку (flea market), який орієнтований на торгівлю товарами між користувачами, що зазвичай є фізичними особами, та традиційного роздрібного ринку (retailer), який охоплює професійних продавців або компанії з розвиненим товарним асортиментом. Основною архітектурною метою при проєктуванні цієї системи було створення спільного каркасу, який дозволяє опрацьовувати товари обох маркетів за єдиною моделлю, збереженням високого ступеня повторного використання коду, при цьому враховуючи специфіку кожного з напрямків. Такий підхід не тільки зменшує складність підтримки системи, але й спрощує процес масштабування нових функцій на обидва напрями, заощаджуючи час та ресурси розробників.

Архітектурно проєкт побудований відповідно до принципів багаторівневої (layered) архітектури [8], де чітко відокремлено різні рівні відповідальності – контролери (presentation layer), фасади й сервіси (business logic layer), об’єкти

доступу до даних (data access layer), DTO і мапери (data transformation layer), а також індексація та пошук (search layer) на основі Elasticsearch. Це забезпечує не тільки чіткість у розподілі обов'язків між компонентами, але й високу тестованість – кожен рівень можна ізольовано перевірити, не залежачи від інших. Наприклад, контролери перевіряються через інтеграційні тести, сервіси – через unit-тести, а Elasticsearch-пошук – через окремі search-тести з фікстурами.

Важливо, що бізнес-логіка для кожного маркету реалізована незалежно одна від одної. Хоча концептуально вони мають спільні операції – створення, отримання, пошук продуктів – кожна з них реалізована окремо, із застосуванням відповідних сервісів (FleaProductService, RetailerProductService), фасадів (FleaProductServiceFacade, RetailerProductServiceFacade), DTO і моделей Elasticsearch (ElasticFleaProduct, ElasticRetailerProduct). Така модульність дозволяє легко адаптувати поведінку в рамках конкретного ринку. Наприклад, для flea товар може бути опублікований лише після перевірки користувача, тоді як для retailer передбачене попереднє модераційне схвалення адміністрацією платформи.

Інтеграція з системою повнотекстового пошуку Elasticsearch є ще одним ключовим аспектом архітектури. Оскільки маркетплейс за своєю суттю передбачає активну взаємодію користувача з системою через пошук, було прийнято рішення реалізувати два окремих індекси – один для flea товарів, інший – для retailer. Це дозволяє з одного боку налаштовувати пошукові алгоритми незалежно для кожного типу продуктів, а з іншого – ізолювати структуру та формат збережених документів. У перспективі така реалізація відкриває шлях до глибокого машинного навчання, наприклад, різних моделей рекомендацій для кожного маркету.

Не менш важливим є підхід до масштабування та майбутньої інтеграції з іншими системами. Усі запити, які приходять до API, фіксуються за допомогою Spring Interceptors, що дозволяє автоматизувати аудит подій, логування дій користувачів, а також впровадження механізмів кешування чи rate limiting у майбутньому. Для конфігурації застосовано підхід externalized configuration: усі параметри (індекси Elasticsearch, URL бази даних, кількість сторінкових

результатів тощо) винесені у файли конфігурації application.yml або передаються через змінні середовища. Завдяки цьому система залишається максимально portable – її легко деплоїти у різні середовища без зміни коду.

Таким чином, система побудована із прицілом на довготривале супроводження та масштабування. Кожен компонент може бути замінений або розширений без впливу на інші, що особливо важливо в умовах активного розвитку бізнес-логіки або інтеграції з новими видами маркетів у майбутньому (наприклад, оптовими або міжнародними продавцями).

4.2 Пошуковий двигун

Інтеграція з Elasticsearch у межах системи управління товарами маркетплейсів Flea та Retailer була реалізована з урахуванням максимальної гнучкості, розширюваності та легкості у підтримці [9]. Основним завданням цього модуля було забезпечити безперебійний, ізольований доступ до окремих індексів для кожного типу товарів, не змішуючи логіку їх взаємодії з пошуковим рушієм. Для цього було створено окрему конфігурацію ElasticSearchConfig, яка служить точкою входу до налаштування всіх Elasticsearch-компонентів у проєкті.

Конфігурація реалізована як клас із анотацією @Configuration, який визначає два окремих ElasticsearchOperations бінів – по одному для кожного ринку. Такий підхід дозволяє точно розділити логіку роботи з індексами flea-products і retailer-products, уникнувши неявного змішування або доступу з помилкового сервісу. Кожен ElasticsearchRestTemplate налаштовується через окремий RestHighLevelClient, який отримує URL хоста з відповідних змінних середовища або YAML-конфігурації, що дає змогу підключатися до різних кластерів або вузлів у залежності від середовища (локального, dev, prod).

Крім базового підключення, в ElasticSearchConfig також реалізовано іменовані біни через @Qualifier – це дозволяє явно вказувати, який саме

ElasticsearchOperations слід використовувати у кожному з сервісів (fleaElasticSearchOperations або retailerElasticSearchOperations).

Створення кастомних EntityMapper-ів, які конвертують Java-об’єкти до форматів, придатних для збереження у вигляді Elasticsearch-документів. Це особливо важливо, якщо об’єкти мають складну вкладену структуру або потребують спеціального перетворення полів (наприклад, перетворення локалізованих назв або дат).

Індекси створюються або перевіряються наявність під час старту застосунку, з використанням IndexOperations. У разі, якщо індекс відсутній – система автоматично ініціалізує його створення, ґрунтуючись на мапінгу класів ElasticFleaProduct і ElasticRetailerProduct. Вони визначені через анотації Spring Data Elasticsearch, як-от @Document, @Field, що дає змогу задати точні типи полів, їхню поведінку при індексації (keyword, text, nested), а також політику зберігання або агрегацій.

Варто зазначити, що з міркувань надійності та відмовостійкості, обидва клієнти можуть бути налаштовані через пул підключень, з використанням кастомних RestClientBuilder налаштувань – наприклад, timeout-и, максимальна кількість паралельних з’єднань, політика ретраїв. Це дозволяє оптимізувати роботу системи при високих навантаженнях або тимчасових перебоях у мережі.

Завдяки такій структурі конфігурації можна легко додати новий тип індексу без порушення існуючої архітектури. Також забезпечується ізоляція бізнес-логіки для flea та retailer Крім того, інтегрувати нові плагіни або шаблони для Elasticsearch (або Opensearch) стає простіше, якщо з’явиться потреба, наприклад, у використанні custom analyzer-ів для пошуку за синонімами або іншими мовами.

Таким чином, ElasticSearchConfig є не лише конфігураційним класом у класичному розумінні Spring, але й справжнім архітектурним шаром, який дозволяє будувати розділену та масштабовану пошукову інфраструктуру, орієнтовану на майбутнє розширення функціональності, включно з впровадженням інтелектуального пошуку, аналітики користувацької поведінки або персоналізованих рекомендацій.

4.3 Особливості пошуку

Інтерфейси репозиторіїв Elasticsearch імплементовано класом реалізації під назвою ElasticSearchFleaProductRepositoryImpl використовуючи клієнтську бібліотеку Elasticsearch. Саме цей клас забезпечує кастомну логіку пошуку товарів для блошиного ринку, адаптовану до особливостей Elasticsearch [10]. Основним методом цього класу є searchByQuery, який реалізує механізм пошуку документів на основі повнотекстових запитів. Для здійснення запитів використовується інструментарій NativeQuery, що дозволяє побудувати складні та гнучкі умови пошуку з урахуванням потреб проєкту. Сам Elasticsearch-клієнт надає можливість низькорівневого контролю за логікою формування запитів та отримання результатів.

Особливістю реалізації є застосування механізму search_after, що є сучасною альтернативою класичному пагінуванню. У випадку великих обсягів даних, коли просте використання from і size ускладнює масштабованість та знижує продуктивність, search_after дозволяє досягти значно кращої ефективності при обході послідовних сторінок результатів. Такий підхід полягає в тому, що запити будуються на основі останнього елемента попередньої вибірки, що забезпечує детермінованість, стабільність порядку та значно пришвидшує обробку, особливо в умовах розподілених або масштабованих індексів. Ще одним важливим аспектом є сортування результатів за унікальним ідентифікатором продукту, що гарантує сталість порядку навіть при одночасному оновленні даних у системі, а також забезпечує коректність наступних сторінок результатів при паралельному зверненні.

Результати, що повертаються з пошуку, не використовуються напряму в контролерах. Замість цього кожен знайдений документ трансформується у DTO (Data Transfer Object), який виступає уніфікованим представленням товару для клієнтської частини. Такий підхід дає змогу ізолювати внутрішню структуру індексу від зовнішнього API, що важливо з точки зору безпеки, гнучкості та еволюції архітектури. DTO може включати лише необхідні для користувача поля,

може форматувати їх відповідним чином (наприклад, форматування цін або назв), а також дозволяє легко адаптувати відповіді під різні потреби фронтенд-клієнтів без зміни внутрішньої моделі.

Таким чином, реалізація репозиторію з використанням NativeQuery, механізму search_after та подальшої трансформації результатів у DTO формує важливу складову масштабованої та ефективної системи пошуку. Вона дозволяє не тільки задовольнити типові запити користувачів, а й підтримує високонавантажені сценарії, де швидкість, точність і гнучкість пошуку відіграють ключову роль у забезпеченні позитивного користувацького досвіду.

4.4 Бізнес-логіка платформи

Одним із ключових елементів системи є шар бізнес-логіки, реалізований у вигляді сервісів. Ці сервіси відповідальні за обробку отриманих даних, реалізацію основних операцій над товарами, а також взаємодію з репозиторіями та зовнішніми компонентами. Вони працюють як своєрідний міст між веб-шаром, що обробляє HTTP-запити, і шаром збереження даних.

Зокрема, для роботи з товарами блошиного ринку реалізовано сервіс FleaProductServiceImpl. Цей клас містить логіку створення, оновлення, отримання і видалення продуктів. Він відповідає за перевірку вхідних даних, застосування бізнес-правил, наприклад, встановлення ринкового типу як FLEA, а також за коректне формування об’єктів для збереження у базі та індексації в Elasticsearch. Важливо, що сервіс бере до уваги специфіку блошиного ринку, наприклад, особливі правила обробки описів або ціноутворення, які можуть відрізнятися від правил роздрібної торгівлі.

Крім того, сервіс відповідає за виклики репозиторіїв, які працюють з реляційною базою даних для довготривалого зберігання, і одночасно з Elasticsearch – для забезпечення швидкого пошуку. Таким чином, дані синхронізуються між двома сховищами, що дозволяє підтримувати цілісність і

актуальність інформації. Сервіс реалізує транзакційність, яка гарантує, що у разі помилки жодні зміни не будуть частково застосовані, що критично для збереження консистентності системи.

Аналогічно працює сервіс для роздрібних товарів RetailerProductServiceImpl. Він має дуже схожу структуру, але містить адаптації, необхідні для особливостей роздрібного ринку, таких як інша логіка ціноутворення, інша категоризація або додаткові атрибути продуктів. Цей підхід дозволяє зберігати загальну архітектурну концепцію, водночас забезпечуючи гнучкість і можливість розвивати окремі напрямки незалежно.

В цілому, бізнес-сервіси виконують роль центрів ухвалення рішень у програмі: вони координують взаємодію між різними компонентами, забезпечують виконання складних операцій і слугують захистом від некоректного використання нижчих рівнів архітектури. Такий поділ надає системі високу модульність і робить її готовою до розширення, адаптації та масштабування.

4.5 Презентаційний шар для клієнтського застосунку

Наступним важливим шаром у системі є фасадні сервіси, які забезпечують спрощений та уніфікований інтерфейс для контролерів. Вони служать своєрідним «прокладним шаром», який інкапсулює виклики бізнес-сервісів і обробляє їх результати, готуючи їх для подальшої трансформації у DTO (Data Transfer Object) або відповіді клієнту [11]. Завдяки фасадам, контролери залишаються максимально легкими та зосередженими на обробці HTTP-запитів, не завантажуючись деталями бізнес-логіки або інтеграції з базою даних.

Фасад FleaProductServiceFacade реалізує основні операції, необхідні для роботи з товарами блошиного ринку. Він координує виклики до сервісного шару, конвертує внутрішні сутності у DTO, які зручні для передачі у веб-шар, а також виконує додаткову обробку, наприклад, агрегацію даних або кешування

результатів. Завдяки цьому фасаду забезпечується більш чиста і підтримувана архітектура, де кожен шар має чіткі завдання та відповідальність.

На рисунку 4.1 зображено принцип роботи контролеру розділу постачальника.

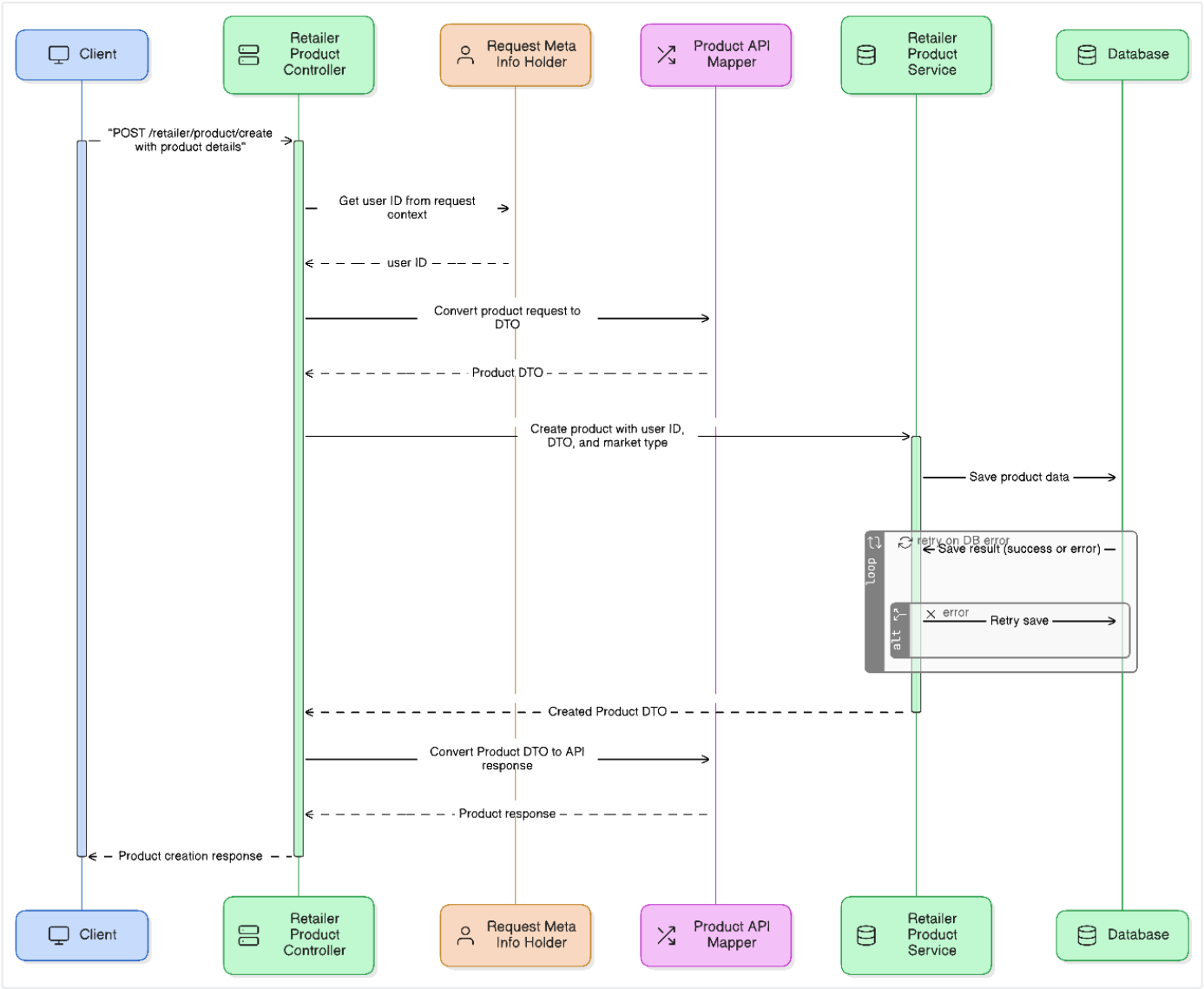

Рисунок 4.1 – Діаграма послідовностей шару представлення постачальника

Подібний фасад [12] для постачальної торгівлі – RetailerProductServiceFacade – виконує ті ж функції, але з урахуванням специфіки ринку retailer. Він відповідає за трансформацію внутрішніх моделей у формати, придатні для зовнішніх запитів, і надає уніфікований інтерфейс для контролерів.

Важливо, що фасад працює з сервісами і репозиторіями, не розкриваючи контролерам складних деталей реалізації, що значно спрощує тестування і розширення системи.

Таким чином, фасадні сервіси виступають як адаптери, що знижують залежність між шарами і підвищують модульність системи. Вони є основою для підтримки принципів SOLID [13], зокрема принципу єдиної відповідальності, і сприяють кращій організації коду.

Мапери у цій системі виконують ключову роль у конвертації даних між різними шарами архітектури, забезпечуючи чистоту коду і відокремлення відповідальностей. Вони реалізують трансформацію між DTO (Data Transfer Object), які використовуються у зовнішніх запитах і відповідях, і внутрішніми сутностями, що зберігаються у базі даних або індексах Elasticsearch. Це дозволяє розробникам уникнути прямої залежності контролерів та сервісів від деталей структури бази, а також спрощує адаптацію системи при зміні бізнес-логіки або формату даних.

Конкретно в цьому проєкті використовується набір mapper-ів, які діляться на кілька категорій. Перш за все, є mapper-и для конвертації запитів клієнта – наприклад, ProductApiMapper, що перетворює об’єкти ProductRequest у внутрішні DTO. Ці DTO далі обробляються у сервісах для створення або оновлення записів у базі. У зворотньому напрямку існують mapper-и, що трансформують внутрішні DTO або сутності у відповіді клієнту – наприклад, ProductResponse, який уніфіковано подає інформацію про товар у форматі, зручному для фронтенду.

Також варто зазначити, що mapper-и беруть на себе не тільки просту копіювання полів, а й виконують більш складні операції – наприклад, об’єднання або розділення полів, конвертацію типів, підготовку специфікацій у вигляді JSON або словника, а також форматування даних перед їх відправкою. Це дозволяє централізувати логіку трансформації, знизити дублювання коду та покращити його підтримку.

Для реалізації mapper-ів застосовується бібліотека MapStruct, яка автоматично генерує код на основі описаних інтерфейсів та анотацій. Для

нечутливих до продуктивності місць використано ModelMapper, котрий працює за принципом рефлексії, що допомагає не писати шар маперів вручну, але це може не сильно сповільнити логіку перетворення даних. Завдяки цьому мінімізується кількість ручного коду і знижується ймовірність помилок у процесі конвертації. Водночас, там де потрібна кастомна логіка, можливо дописати методи вручну, що забезпечує гнучкість і можливість налаштувань.

4.6 Безпекова частина системи

Реалізація Role-Based Access Control (RBAC) у наведеній конфігурації безпеки на базі фреймворку Spring Security демонструє глибоко інтегровану архітектуру контролю доступу, яка базується на відокремленні обов’язків, поділі прав і централізації автентифікації через JSON Web Token (JWT). Така модель безпеки є типовою для сучасних вебзастосунків, орієнтованих на безстанні (stateless) API, особливо у контексті мікросервісної архітектури, де масштабованість, розмежування повноважень і дотримання принципу найменших привілеїв (Principle of Least Privilege) мають ключове значення.

Ключовим об'єктом, що відповідає за забезпечення доступу до HTTP-ресурсів, є компонент SecurityFilterChain, який конфігурується за допомогою DSL (domain-specific language) на базі класу HttpSecurity. Саме в цьому місці встановлюються правила авторизації, що визначають, які запити мають бути дозволені без автентифікації, а які вимагають певної ролі або повної перевірки користувача. Фреймворк Spring Security [14] у цьому випадку виступає як посередник між зовнішніми запитами та внутрішньою логікою доступу, автоматизуючи обробку, перевірку і делегування повноважень згідно із заданими політиками.

У конфігурації використовуються кілька важливих фільтрів. Зокрема, JwtFilter бере участь у перехопленні кожного HTTP-запиту з метою перевірки наявності токена у заголовку Authorization (як правило, у форматі Bearer

<токен>). У разі успішної валідації – тобто перевірки цифрового підпису, терміну дії, цілі та джерела токена – фільтр створює об’єкт типу Authentication і реєструє його у контексті безпеки через клас SecurityContextHolder. Саме цей контекст слугує єдиним джерелом прав доступу для всієї подальшої обробки запиту в межах потоку.

Для гостьових (анонімних) запитів передбачено окремий перелік маршрутів GUEST_ALLOWED_RESOURCES, які позначено як загальнодоступні. Це дозволяє не блокувати важливу функціональність, зокрема вхід, вихід або перегляд деяких публічних ресурсів. Дозвіл до таких маршрутів конфігурується через метод permitAll(), який фактично дозволяє запити без вимоги попередньої автентифікації або належності до певної ролі. Це особливо корисно для SPA-додатків, що потребують доступу до авторизаційного API до моменту створення токена.

Окрему категорію ресурсів складають службові ендпоінти /actuator/health/** та /actuator/prometheus, що використовуються для моніторингу працездатності системи (наприклад, з Prometheus, Grafana, або іншими інструментами). Доступ до них дозволений лише при наявності ролі SERVICE. Такий підхід зменшує ризик несанкціонованого моніторингу або збору чутливої системної інформації, яка могла б бути використана для аналізу потенційних векторів атаки. У цьому випадку Spring Security використовує вбудований механізм перевірки ролей, де роль, яка зберігається у JWT, має відповідати значенню, переданому у методі hasRole(String role). Варто зазначити, що Spring автоматично префіксує значення ROLE_ до ролі, якщо використовується hasRole(...), тоді як hasAuthority(...) очікує повне значення ролі без префікса.

Конфігурація також демонструє інтеграцію з механізмами керування CORS (Cross-Origin Resource Sharing). Це важливо у випадках, коли клієнтська частина застосунку розгорнута на іншому домені або порту, ніж серверна логіка. Через компонент CorsConfiguration дозволено всі заголовки, методи та домени, що дозволяє здійснювати запити з будь-якого джерела. Також дозволено передачу cookies (через setAllowCredentials(true)), що є важливою опцією для

авторизаційних механізмів, які покладаються на сесію чи передачу маркерів через куки. Таким чином, забезпечується гнучкість при інтеграції з фронтендом незалежно від середовища його виконання.

Система автентифікації повністю побудована на принципі безстанності, що задається через SessionCreationPolicy.STATELESS. У результаті сервер не зберігає інформацію про сесію користувача, що зменшує ризик атаки через викрадення сесійного ідентифікатора (наприклад, session fixation, session hijacking) та спрощує горизонтальне масштабування серверів. У безстанному середовищі кожен запит має бути самодостатнім з точки зору передачі повноважень – що і забезпечується шляхом ін’єкції JWT у кожен запит.

Іншою важливою частиною конфігурації є об’єднання SecurityFilterChain із користувацькими фільтрами, такими як SessionGuestAuthenticationFilter та RequestMetaInfoFilter. Перший може використовуватись для динамічного створення тимчасового Authentication-контексту для гостей або сесій з анонімними правами. Це дозволяє системі відслідковувати користувацьку активність або надавати обмежені функції навіть без повної автентифікації. Другий фільтр (RequestMetaInfoFilter) може слугувати джерелом додаткових метаданих, таких як IP-адреса, заголовки клієнта або параметри пристрою, що надалі можуть використовуватись як частина логіки RBAC – наприклад, для блокування доступу з певних регіонів або агентів користувача.

Варто також згадати про роль AuthenticationManager, яка полягає у централізованому делегуванні автентифікаційних запитів. У цьому прикладі використовується передача екземпляра AuthenticationManager через ін’єкцію залежності від AuthenticationConfiguration, що дозволяє уникати явного створення власного менеджера. Це дає змогу фреймворку автоматично налаштувати його на основі наявних UserDetailsService, AuthenticationProvider, або DaoAuthenticationProvider.

У більш широкому сенсі, використання RBAC у Spring Security дозволяє поєднувати декларативну конфігурацію з динамічними перевірками. У контролерах або сервісах можуть використовуватись анотації @PreAuthorize,

@Secured, або @RolesAllowed, які дають змогу вказати дозволені ролі безпосередньо у коді методів, але в даній конфігурації все обмежується лише глобальною перевіркою на рівні HTTP-шляху. Такий підхід відповідає принципу централізованого керування політиками доступу, що спрощує їхнє супроводження та аналіз з боку аудиту або безпекових ревізій.

4.7 Інтеграція сторонніх рішень аутентифікації

У представленому механізмі аутентифікації застосовується протокол OAuth 2.0 [15], який є сучасним і широко вживаним стандартом для делегування доступу, що дозволяє стороннім додаткам отримувати обмежений доступ до ресурсів користувача без необхідності передавати паролі. У даному випадку використовується інтеграція з Google як провайдером аутентифікації, що дозволяє користувачам виконувати вхід у систему за допомогою облікового запису Google. Такий підхід підвищує зручність та безпеку, оскільки користувачеві не потрібно створювати окремий логін і пароль для системи, а також дає змогу централізовано керувати аутентифікацією через відомий і довірений провайдер.

Процес підключення Google OAuth2 передбачає декілька ключових етапів, починаючи від налаштування самого OAuth клієнта у консолі Google Cloud Platform, де створюється проект, в якому визначаються параметри OAuth, зокрема ідентифікатор клієнта (Client ID), секрет клієнта (Client Secret) та URI для редіректів. Ці параметри дозволяють серверу коректно взаємодіяти з Google, визначати дозволені джерела запитів і забезпечувати повернення користувача на коректний URL після аутентифікації.

Далі на стороні додатку відбувається конфігурація Spring Security, зокрема підключення залежностей Spring Security OAuth2 Client, налаштування властивостей у application.properties або application.yml, де задаються параметри клієнта Google, зокрема адреса авторизації, адреса отримання токена, і URI для

callback. В цьому процесі важливо врахувати безпекові аспекти, зокрема захист токенів, налаштування HTTPS, правильне управління сесіями користувачів.

На рисунку 4.2 зображено діаграму послідовностей інтеграції Google OAuth2.

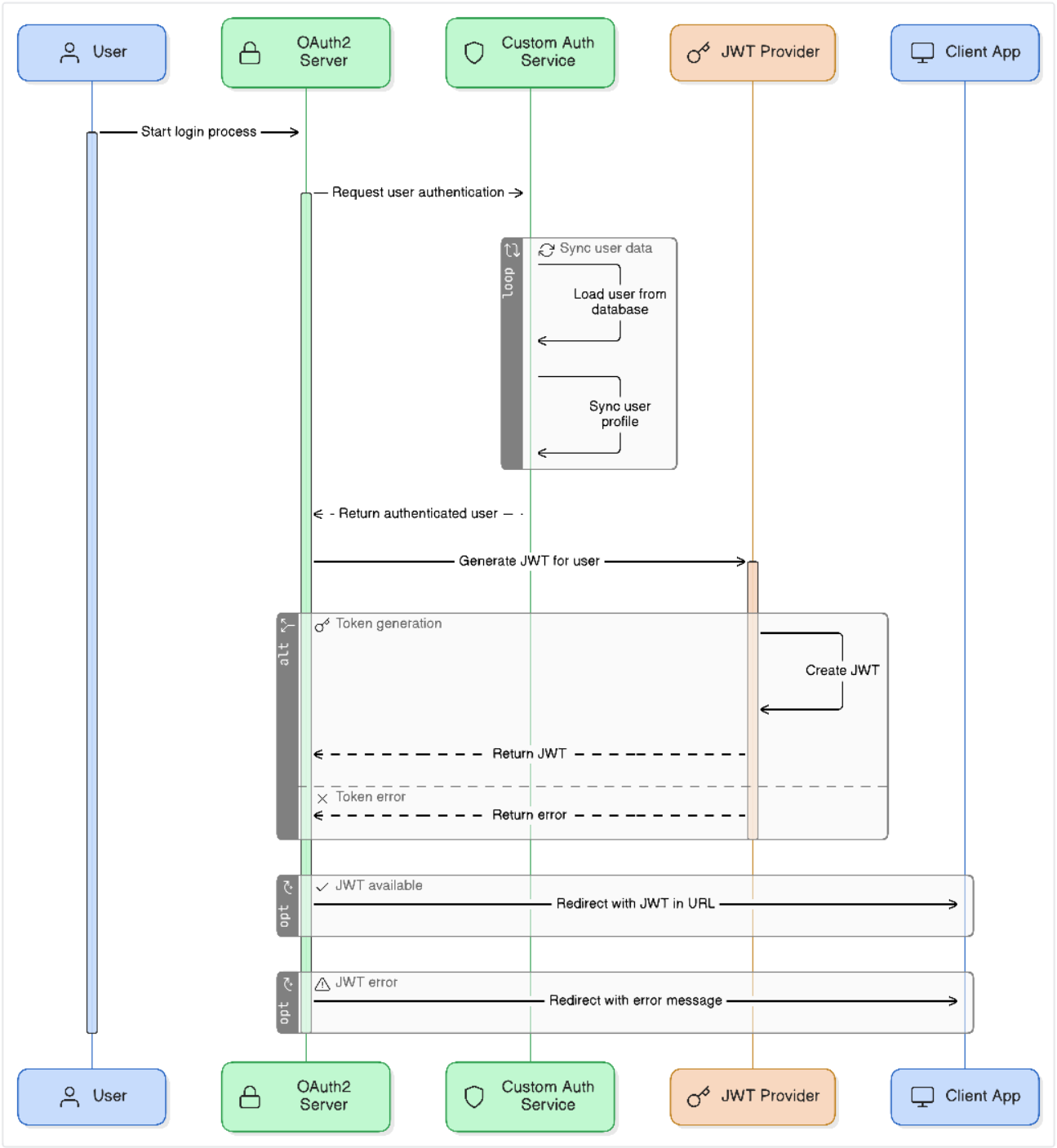

Рисунок 4.2 – Діаграма послідовностей інтеграції Google OAuth2

У коді реалізовано кастомний сервіс CustomOAuth2UserService, що розширює DefaultOAuth2UserService. Цей клас відповідає за отримання та обробку інформації про користувача, яка надходить від провайдера OAuth2 (Google). Основна функція цього класу – метод loadUser, який викликається після успішної аутентифікації провайдером і отримання токена доступу. Він відповідає за завантаження профілю користувача з Google, трансформацію атрибутів у внутрішній формат додатку, а також інтеграцію або оновлення інформації про користувача у базі даних.

Для універсалізації та забезпечення можливості масштабування механізму на інші провайдери OAuth2, реалізовано паттерн Factory – OAuth2UserFactory, який створює об’єкти типу OAuth2UserInfo залежно від провайдера. Це дозволяє ізолювати специфіку обробки атрибутів, що повертаються від Google (наприклад, ідентифікатор користувача, ім’я, прізвище, email), у відповідному класі GoogleOAuth2UserInfo. Такий підхід покращує підтримуваність коду і дозволяє легко додати нові провайдери без суттєвих змін у бізнес-логіці.

Значна увага приділяється безпеці, зокрема створенню та управлінню JWT (JSON Web Token), який виступає як механізм передачі підтвердженої інформації про користувача між клієнтом і сервером. У класі OAuth2SuccessHandler реалізовано логіку, яка після успішної аутентифікації генерує JWT, використовуючи JwtProvider. Токен створюється на основі унікального email користувача, і в ньому містяться необхідні дані, зокрема роль користувача (наприклад, "USER"). Після генерації токен інкапсулюється у URI, куди користувач буде перенаправлений, що дозволяє клієнту прийняти і зберегти токен для подальшої взаємодії із захищеними ресурсами.

Важливо відзначити, що OAuth2SuccessHandler успадковується від SimpleUrlAuthenticationSuccessHandler, що є частиною Spring Security і служить для обробки успішної аутентифікації, зокрема перенаправлення користувача після входу. В даному випадку реалізація кастомізує поведінку перенаправлення, додаючи до URL згенерований токен.

Клас OAuth2UserManagementService реалізує бізнес-логіку управління користувачами, які аутентифікуються через OAuth2. Він забезпечує як реєстрацію нових користувачів, так і оновлення даних існуючих користувачів. Використання анотації @Transactional гарантує цілісність операцій запису в базу даних, забезпечуючи, що у разі помилки всі зміни буде скасовано, що запобігає неконсистентному стану. У цьому класі також використовується паттерн фабрики UserFactory, що відповідає за створення екземплярів користувача на основі інформації з OAuth2.

Реалізація залежностей через Spring Framework забезпечує інверсію контролю (IoC) та впровадження залежностей (Dependency Injection), що підвищує модульність, тестованість і масштабованість системи. Використання анотацій, таких як @Component, @Service, @Autowired, полегшує конфігурацію і автоматичне підключення необхідних компонентів у контекст додатку.

Загалом, підхід базується на сучасних архітектурних принципах, серед яких є розділення відповідальностей, кожен клас виконує чітко визначену функцію, таких як обробка успішної аутентифікації, завантаження даних користувача з провайдера, управління бізнес-логікою користувача, генерація токенів. Також, використання паттернів проектування: фабрика для створення специфічних об’єктів користувача OAuth2, інверсія контролю для управління залежностями. Щодо безпеки, застосування JWT для безпечної передачі інформації про користувача, використання Spring Security для управління доступом.

Масштабованість і розширюваність дає можливість легко додати нові провайдери OAuth2, змінити спосіб створення токена або зберігання даних без великих змін у коді.

Для підключення Google OAuth2 здійснюються кроки, починаючи з реєстрації додатку в Google Cloud, де створюється OAuth-клієнт з налаштуваннями авторизації та редіректу. Потім налаштовується додаток на стороні сервера через конфігурацію Spring Security OAuth2 Client, після чого реалізується сервіс для завантаження і обробки даних користувача (CustomOAuth2UserService), механізм управління користувачами

(OAuth2UserManagementService), генерація JWT (JwtProvider та OAuth2SuccessHandler) та перенаправлення користувача з токеном.

Детальна увага приділяється збереженню і оновленню токенів доступу від провайдера (Google) в базі даних, що дозволяє виконувати додаткові операції, наприклад, оновлення даних користувача або доступ до API Google від імені користувача.

4.8 Модуль керування користувачами

Ця частина системи відповідає за авторизацію та реєстрацію користувачів у додатку, реалізуючи ключові функції безпеки і керування обліковими записами. В основі лежить кілька взаємопов'язаних компонентів, які спільно забезпечують повний цикл автентифікації, реєстрації, активації акаунту, а також відновлення паролю.

В цьому модулі виділено логіку роботи з користувачами, розділено на кілька важливих функціональних напрямів: реєстрація нових облікових записів, підтвердження реєстрації (активація), автентифікація існуючих користувачів (вхід), керування паролем (скидання, оновлення), а також підтримка гостьового режиму з можливістю безшовного переходу до зареєстрованого облікового запису. Ця логіка реалізується через взаємодію між контролерами, сервісним шаром, репозиторіями доступу до бази даних, DTO-класами, доменними сутностями, а також допоміжними компонентами безпеки.

Контролер аутентифікації, або AuthenticationController, виступає як точка входу для HTTP-запитів [16], пов'язаних із входом користувача в систему, ініціалізацією процесу скидання паролю та підтвердженням його зміни. Коли користувач надсилає запит на вхід, контролер приймає об'єкт із даними для авторизації, валідуючи його, і передає до сервісного шару, який реалізує основну бізнес-логіку аутентифікації. Якщо вхід успішний, генерується відповідь, що містить токен доступу, який гарантує подальшу автентифіковану взаємодію

користувача з системою. Окрім цього, після логіну запускається асинхронна операція, яка відправляє команду для перенесення даних гостьового акаунту, що є частиною більшої логіки з обробки користувацьких сесій і переходу від тимчасового гостьового стану до авторизованого. Цей механізм дозволяє плавно зберігати дані користувача, навіть якщо він раніше працював без реєстрації, тим самим покращуючи користувацький досвід.

У частині, що відповідає за реєстрацію користувачів, представлені механізми створення нового акаунту, включно з перевіркою капчі для захисту від ботів та автоматичних реєстрацій. RegistrationController приймає запити з даними нового користувача, валідуючи їх і передаючи в сервісний шар. Тут виконується реальна робота по збереженню інформації про користувача, генерації коду активації, який надсилається на електронну пошту. Цей код дозволяє підтвердити валідність адреси електронної пошти і активувати обліковий запис, уникаючи реєстрації з фальшивими або некоректними даними. Активація здійснюється через спеціальний запит з кодом, що також проходить валідацію і впливає на стан облікового запису в базі даних, переводячи його з неактивного в активний. Такий підхід дозволяє підвищити безпеку системи і запобігти несанкціонованому доступу.

Сервісний шар, представлений реалізацією AuthenticationServiceImpl, є центром бізнес-логіки, що управляє усіма процесами, пов’язаними з аутентифікацією і реєстрацією. Він використовує різні компоненти, включаючи менеджер аутентифікації Spring Security, який здійснює перевірку облікових даних, і JWT-провайдер для генерації токенів доступу, що забезпечують безпечний і зручний спосіб збереження сесій користувачів. У разі невдалого входу через неправильний пароль або відсутність облікового запису генерується відповідне виключення, яке надсилається клієнту із зрозумілим повідомленням про помилку.

При реєстрації користувача відбувається складний процес, що включає перевірку відповідності паролів, підтвердження капчі шляхом звернення до зовнішнього сервісу, збереження нових користувацьких даних у базі, а також

асинхронну генерацію та збереження коду активації. Асинхронність цієї операції дозволяє не блокувати основний потік виконання, підвищуючи продуктивність і швидкість відповіді сервісу. Важливим моментом є суворе дотримання принципів Domain-Driven Design [17], при якому DTO-об’єкти відокремлені від доменної моделі. Це дозволяє зберегти чистоту архітектури, забезпечити розділення відповідальностей і полегшити підтримку та масштабування системи.

Сервіс також підтримує функціонал скидання паролю, що включає генерацію одноразового коду, який відправляється користувачу, і подальшу зміну паролю після введення правильного коду. Перевірка співпадіння паролів при скиданні виконується для забезпечення цілісності введених даних і запобігання помилкам користувача. Важливо, що всі ці операції супроводжуються перевіркою контексту користувача через RequestMetaInfoHolder, який дозволяє отримати метадані поточного запиту, включно з ідентифікатором користувача, що важливо для коректного оновлення даних у базі.

Доменна модель користувача відображена через сутність User, яка містить стандартні поля, що ідентифікують користувача, такі як унікальний ідентифікатор, ім’я, ролі, статус активності, а також коди активації та скидання паролю. Використання UUID як ідентифікатора забезпечує унікальність і масштабованість системи. Поля ролей дозволяють задавати різні рівні доступу, а наявність провайдера аутентифікації відкриває можливість інтеграції з різними зовнішніми системами автентифікації, наприклад, соціальними мережами або корпоративними каталогами.

Система надійно обробляє валідацію вхідних даних, використовуючи анотації валідності з Jakarta Validation API, що дозволяє перехоплювати некоректні дані ще на рівні контролера і повертати відповідні HTTP-коди з описом проблеми. Такий підхід мінімізує ризики виникнення помилок у глибині системи і підвищує загальну безпеку і стабільність роботи.

Таким чином, даний модуль охоплює весь цикл роботи з користувачами від їх створення, активації, аутентифікації до управління паролями, реалізуючи високі стандарти безпеки і зручності. Враховано принципи сучасної розробки,

зокрема застосування асинхронності для неблокуючих операцій, розділення доменної моделі і DTO, а також використання готових механізмів Spring Security, що разом забезпечує гнучкість, надійність і підтримуваність всієї системи.

Також, Дана частина системи реалізує функціональність управління персональною інформацією користувача в модулі авторизації. Основна роль цієї підсистеми полягає в тому, щоб надавати API для отримання, створення та оновлення об’єктів, що містять адресу проживання, номер телефону та пов’язаний ідентифікатор користувача. Вона слугує проміжною ланкою між фронтендом або іншим зовнішнім сервісом і базою даних, в якій зберігається відповідна інформація.

Контролер UserInfoController відповідає за обробку HTTP-запитів, адресованих до шляху /user-info. Для цього визначено три основні методи. Перший дозволяє отримати інформацію про користувача за його ідентифікатором, другий – створити новий запис з прив’язкою до поточного автентифікованого користувача, а третій – оновити вже наявний запис, використовуючи його унікальний ID. Усі вхідні об'єкти запиту трансформуються в DTO-шар за допомогою мапера UserInfoApiMapper, після чого передаються в сервісний шар.

Сервісна реалізація UserInfoServiceImpl інкапсулює бізнес-логіку і забезпечує взаємодію з базою даних через репозиторій. Метод getUserInfo здійснює пошук відповідного об’єкта UserInfo у базі за userId, і в разі його відсутності кидає виняток з HTTP-кодом 404. Метод createUserInfo здійснює мапінг DTO в сутність, зберігає її в базу і повертає відповідь у DTO-форматі. Для оновлення інформації використовується механізм пакетного оновлення через утиліту BatchEntityUpdater, яка в рамках транзакції завантажує об'єкти з бази, selectively оновлює лише ті поля, що були явно задані, та зберігає зміни.

Сутність UserInfo представляє собою таблицю user_info у реляційній базі даних, де зберігаються дані про адресу (вулиця, місто, країна, поштовий індекс), номер телефону та зв’язаний з ними ідентифікатор користувача. Усі поля мають обмеження довжини, що відповідає вимогам до структури таблиці.

Завдяки такій архітектурі, система є чітко розділеною на відповідальні рівні: вхідний (API), логічний (сервісний) та зберігання (репозиторій з сутностями). Це дозволяє легко масштабувати або адаптувати компонент під нові вимоги, включаючи розширення обсягу персональної інформації або інтеграцію з зовнішніми сервісами.

На рисунку 4.2 зображено діаграму послідовностей створення інформації.

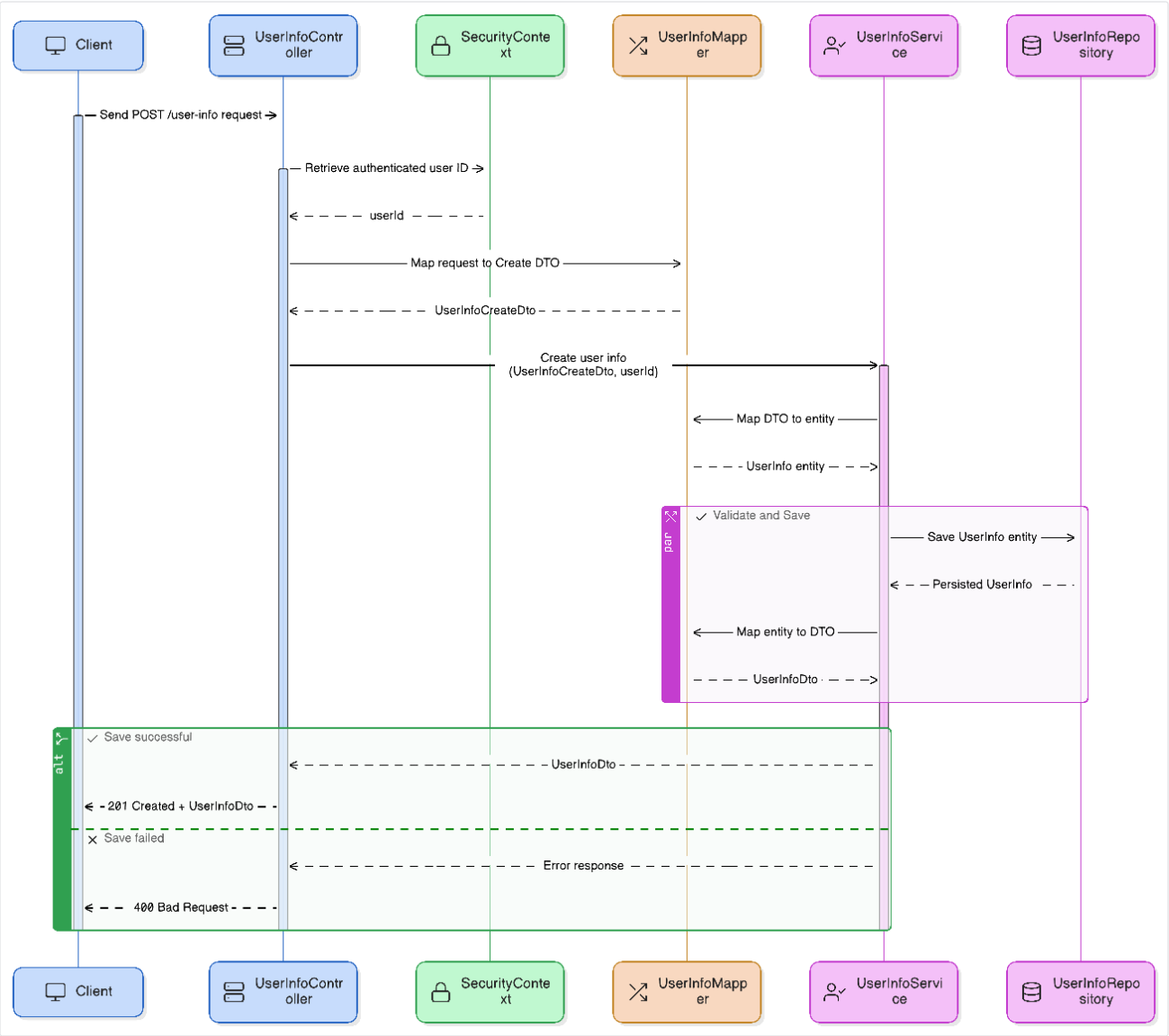

Рисунок 4.2 – Діаграма послідовностей створення інформації користувача

За створення нового запису (котрий вказується в особистому кабінеті користувача) відповідальний UserInfoService, використовуючи метод

createUserInfo, відповідно. При цьому унікальність забезпечується тим, що userId береться з поточного контексту безпеки, а не з клієнтського запиту. Таким чином, навіть якщо клієнт намагається підмінити дані, система проігнорує переданий ID і встановить правильний з контексту автентифікації. Вхідні дані мапляться з об’єкта запиту у DTO, а потім передаються в сервіс для збереження.

На рисунку 4.3 зображено діаграму послідовностей оновлення інформації.

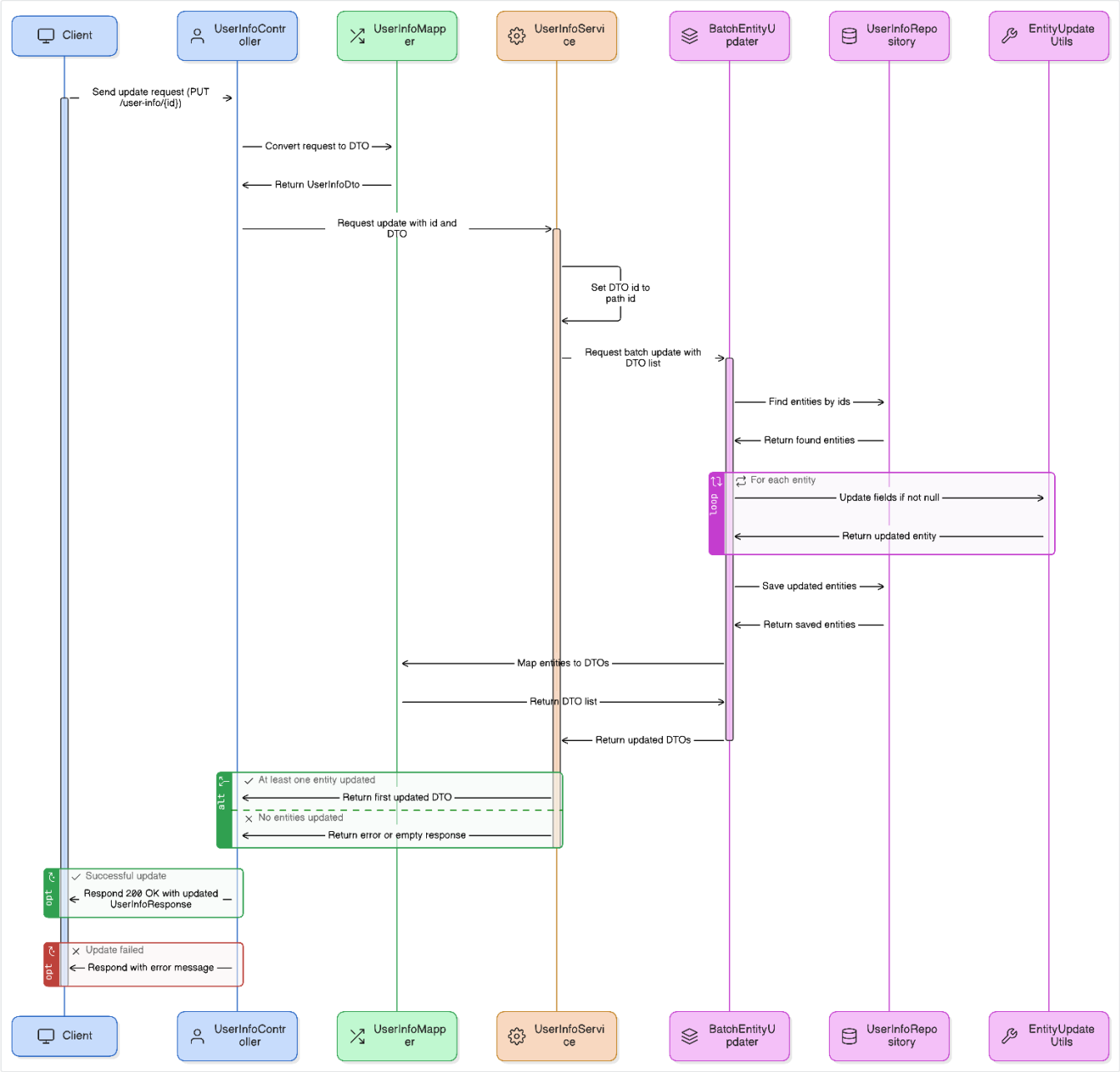

Рисунок 4.3 – Діаграма послідовностей оновлення інформації користувача

Метод updateUserInfo дозволяє оновити наявний запис, використовуючи його ID. Це оновлення не створює новий запис, а змінює існуючий, зберігаючи логічну цілісність даних.

Сервісний клас UserInfoServiceImpl є місцем, де реалізується основна логіка обробки інформації. Метод getUserInfo звертається до репозиторію, щоб знайти об’єкт за userId. Якщо об’єкт не знайдено, сервер надсилає помилку ApiRequestException з кодом 404, що є коректною практикою REST API для інформування клієнта про відсутність ресурсу.

4.8 Децентралізований доступ до мета-інформації користувачів

Дана частина системи відповідає за безпеку на рівні веб-запитів, а саме за механізми фільтрації та обробки HTTP-запитів з метою контролю доступу, ідентифікації користувачів, а також збору метаданих, які можуть використовуватися для аудиту, логування, трасування або статистики. Вся ця функціональність реалізована у вигляді набору спеціалізованих фільтрів, які працюють у певному порядку в ланцюжку обробки запитів, що є частиною Spring Security. Ключовою особливістю є те, що порядок цих фільтрів є критично важливим для коректної роботи всієї системи, оскільки кожен наступний фільтр покладається на результати роботи попереднього.

В основі лежить конфігураційний клас, який визначає порядок створення та підключення кількох фільтрів, що є критичним для коректної роботи механізму безпеки. Порядок оголошення фільтрів у цьому класі прямо впливає на те, у якій послідовності вони будуть виконуватися під час обробки HTTP-запитів, і це важливо, бо деякі фільтри повинні бути ініціалізовані раніше за інші, щоб весь ланцюжок безпеки працював коректно.

Першим у цьому ланцюжку виступає JwtFilter, який займається розпізнаванням і валідацією JWT-токенів, які надсилаються клієнтом в кожному HTTP-запиті. Сам JWT (JSON Web Token) є механізмом аутентифікації, який

дозволяє безпечно передавати інформацію про користувача без необхідності зберігати сесію на сервері. JwtFilter в процесі роботи отримує HTTP-запит, витягує з нього токен, якщо він є, та перевіряє його валідність за допомогою JwtProvider. Якщо токен виявляється дійсним, JwtFilter формує об’єкт Authentication, який потім встановлюється у SecurityContext – центральне сховище інформації про поточного користувача в Spring Security. Якщо токен відсутній або недійсний, JwtFilter не створює автентифікацію, і таким чином запит проходить як анонімний або неавторизований. Крім того, у випадку виникнення помилки валідації токена, цей фільтр відповідає за припинення подальшої обробки запиту і повернення клієнту відповідної HTTP-помилки [18], що сигналізує про недійсний токен.

Другий фільтр, який встановлюється у ланцюжок після JwtFilter, це RequestMetaInfoCreator. Він не відповідає за аутентифікацію, а створює спеціальний об’єкт метаданих для кожного запиту. Його головною задачею є генерація унікального ідентифікатора кореляції (correlation ID), який використовується для відстеження і логування запиту через усі компоненти системи. Такий кореляційний ідентифікатор дозволяє пов’язувати події, логовані на різних рівнях та в різних сервісах, з конкретним HTTP-запитом, що значно полегшує діагностику та аналіз роботи системи. RequestMetaInfoCreator зберігає цей об’єкт метаданих у спеціальному сховищі, яке забезпечує доступ до нього протягом усього часу обробки запиту, і при цьому гарантує безпеку та ізоляцію між потоками.

Наступний фільтр у ланцюжку, RequestMetaInfoFilter, доповнює метадані запиту, які були створені раніше RequestMetaInfoCreator. Він отримує актуальний об’єкт метаданих і встановлює в нього додаткову інформацію, зокрема IP-адресу клієнта, звідки надійшов запит, а також, якщо користувач автентифікований, його унікальний ідентифікатор. Для визначення, чи є користувач автентифікованим, цей фільтр використовує UserInfrastructureService, який взаємодіє з контекстом безпеки та базою даних користувачів. Така додаткова інформація суттєво розширює можливості аудиту та аналізу, дозволяючи зрозуміти не лише хто

виконав дію, а й звідки вона була ініційована. Після завершення обробки запиту цей фільтр очищує контекст метаданих, щоб запобігти випадковому перетину інформації між різними потоками або запитами.

Останнім в черзі фільтрів є SessionGuestAuthenticationFilter, який виконує більш складну роль, ніж просто встановлення контексту безпеки. Цей фільтр відповідає за обробку сесій анонімних або гостевих користувачів, тобто тих, хто не пройшов повну аутентифікацію.

На рисунку 4.4 зображено роботу компоненту керування мета-інформацією.

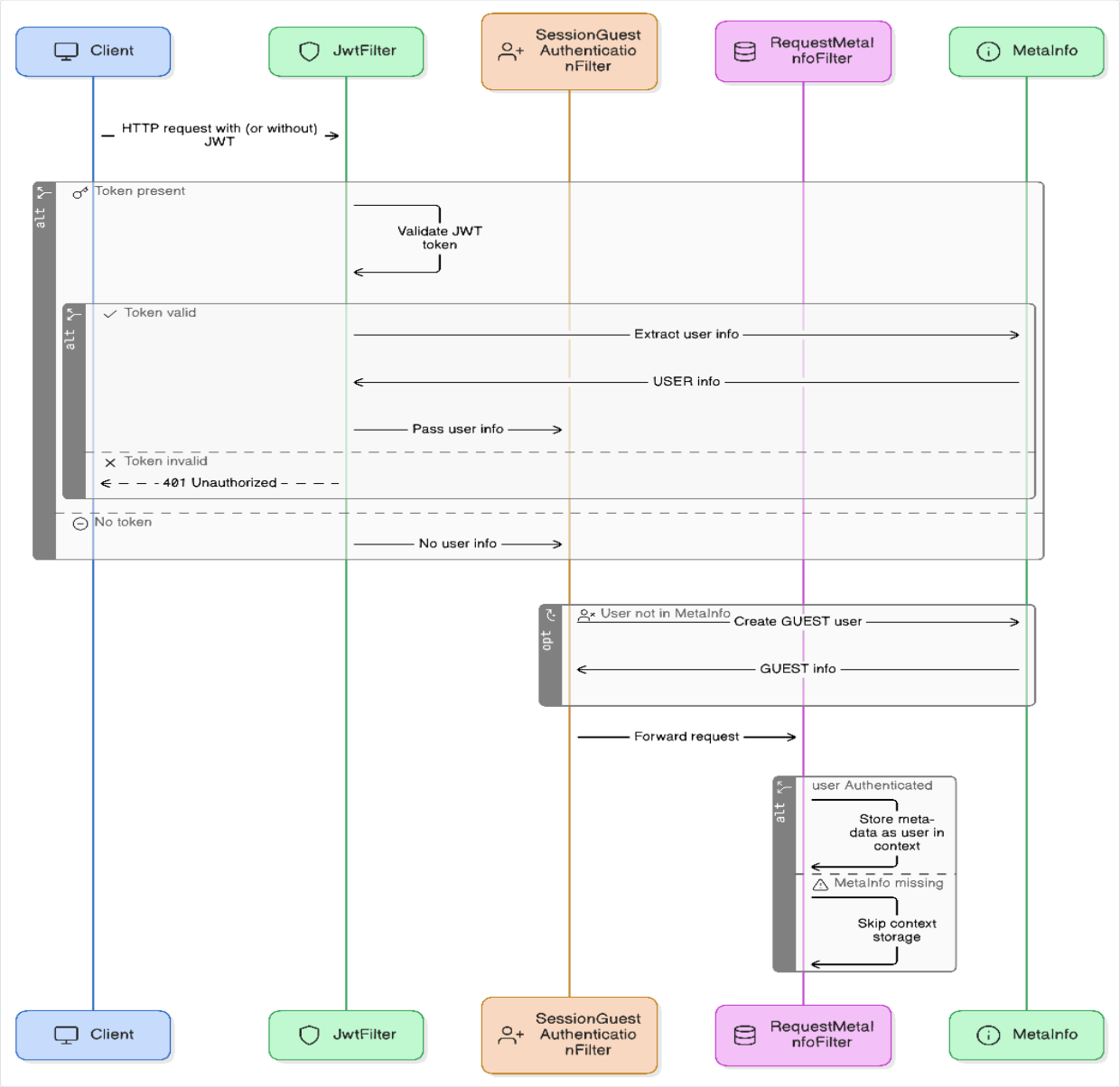

Рисунок 4.4 – Діаграма послідовностей керування мета-інформацією користувача

Його функція полягає в тому, щоб забезпечити гостям унікальну ідентифікацію сесії через спеціальне cookie під назвою SESSION_ID. Якщо у запиті відсутній цей cookie, фільтр генерує новий унікальний ідентифікатор сесії, встановлює його у відповідь клієнту, тим самим ініціюючи сесію гостя. Для кожної сесії створюється унікальна «гостьова» електронна адреса, яка базується на цьому ідентифікаторі сесії, і за допомогою сервісу UserInfrastructureService перевіряється, чи існує користувач з такою електронною адресою в системі. Якщо ні, виконується реєстрація нового гостя в базі даних. Після цього фільтр завантажує інформацію про користувача з UserDetailsService і створює об’єкт Authentication, який встановлюється у SecurityContext, тим самим імітуючи аутентифікацію користувача-гостя.

Цей підхід дозволяє системі відстежувати ідентичність користувача навіть без формальної авторизації, зберігаючи сесію і забезпечуючи персоналізацію і збереження стану між запитами.

Весь цей набір фільтрів об’єднується у конфігуратор, який забезпечує їх правильне підключення у ланцюжок безпеки Spring Security. Саме цей конфігуратор відповідає за додавання фільтрів у визначеному порядку відносно стандартних фільтрів Spring, що гарантує коректне виконання логіки автентифікації і обробки метаданих. Це дуже важливо, оскільки неправильне розташування або затримка у створенні певного фільтра може призвести до того, що він буде пропущений у фільтр-ланцюжку або виконуватиметься не у тому порядку, який необхідний для коректної роботи системи. Щоб уникнути подібних проблем, розробники рекомендують розміщувати оголошення фільтрів у конфігурації у тому ж порядку, у якому вони додаються в JwtAuthenticationConfigurer, або використовувати додаткові механізми для керування життєвим циклом бінів.

Мета-інформацію користувача тепер можливо дістати вздовж всієї системи, використовуючи компонент RequestMetaInfoHolder, котрий утримує мета-інформацію RequestMetaInfo в TreadLocal, який доступний тільки в рамках одного

потоку, що дозволяє уникнути проблем з багатопоточністю [19] в системі, виключаючи несанкціоновану зміну даних.

Таким чином, цей модуль забезпечує комплексну безпекову обробку запитів, починаючи з розпізнавання користувача за JWT, формування унікальних ідентифікаторів для запитів, доповнення метаданих інформацією про клієнта та користувача, і, в разі необхідності, створення гостьових сесій із відповідною автентифікацією. Все це покращує прозорість обробки запитів, підвищує безпеку системи та дозволяє гнучко працювати з користувачами різного статусу в одному додатку.

Зібрані метадані підвищують прозорість і керованість процесів, що відбуваються у системі, сприяють покращенню аудиту, моніторингу та діагностики, що є надзвичайно важливим для сучасних додатків з високими вимогами до безпеки та аналітики. Весь цей функціонал інтегрується з Spring Security, використовуючи механізми фільтрації запитів та контексти безпеки, що дозволяє забезпечити централізований і стандартизований підхід до роботи з веб-застосунком.

В результаті модуль не лише виконує роль проміжного шару між користувачем і бізнес-логікою, а й стає критично важливою частиною архітектури безпеки. Він гарантує, що кожен запит проходить однакову процедуру перевірки й обробки, незалежно від джерела. Це зменшує кількість потенційних вразливостей, полегшує впровадження нових політик доступу та забезпечує масштабованість без втрати контролю над безпековими аспектами.

Крім того, модуль легко адаптується до змін вимог безпеки або інтеграції з новими службами автентифікації. Його гнучкість та розширюваність дають змогу швидко реагувати на нові виклики та загрози. Це робить його ефективним інструментом у побудові надійної, стійкої та прозорої системи.

4.9 Взаємодія баз даних сервісів управління товарами

Архітектура двох баз даних [20] сервісів управління товарами, реалізована за допомогою СУБД PostgreSQL. В першій базі даних, основними структурними компонентами системи є таблиці «orders» (замовлення) та «unified_products» (уніфіковані товари), які взаємодіють через відношення багато-до-одного.

На рисунку 4.5 зображено схему першої бази даних.

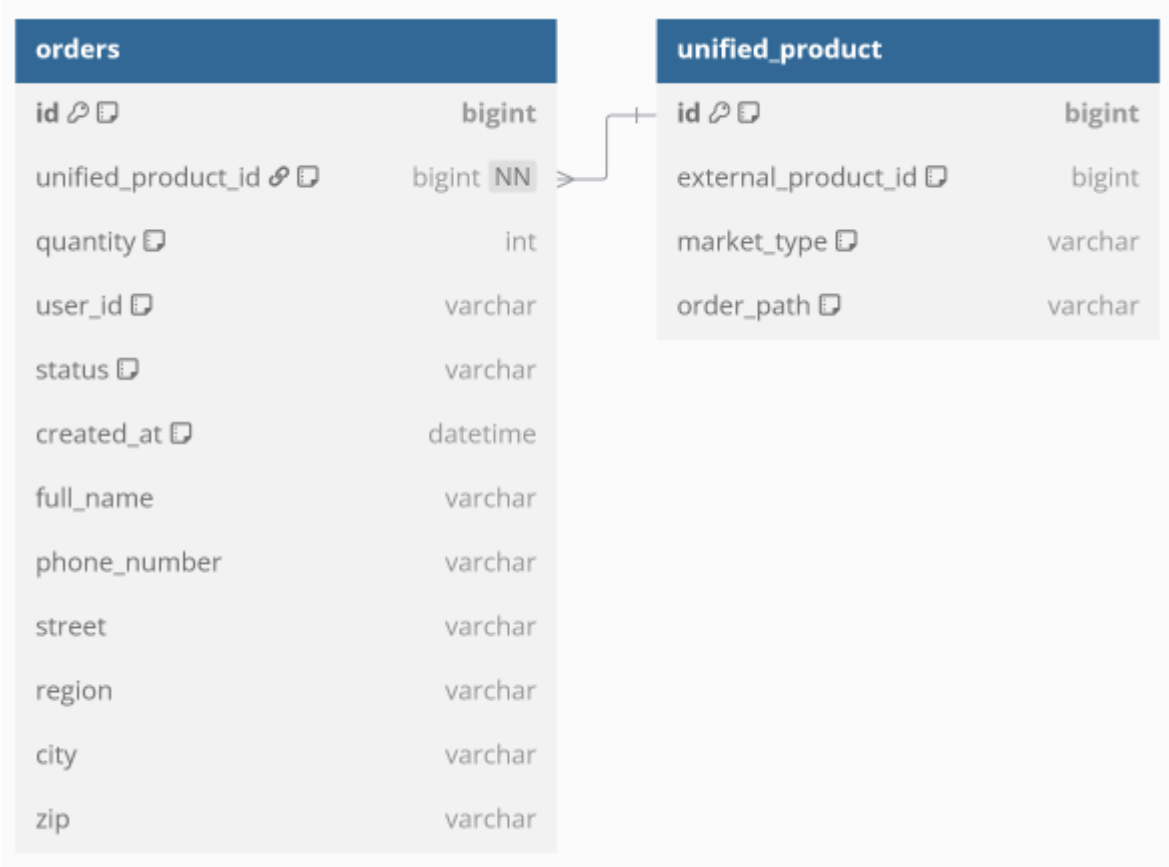

Рисунок 4.5 – Схема бази даних замовлень

Кожен запис у таблиці «orders» містить поля для ідентифікатора замовлення (id), кількості товару (quantity), даних користувача (user_id), статусу замовлення (status), часу створення (created_at), а також інформації про доставку (адреса, регіон, місто, індекс). Ключовим елементом є зв’язок між «orders» та «unified_products» через поле «unified_product_id», що вказує на уніфікований запис товару.

Таблиця «unified_products» виконує роль централізованого каталогу, об’єднуючого товари з різних джерел. Вона містить поле «market_type», яке розрізняє два типи товарів: «RETAILER» (від постачальників) та «FLEA» (від звичайних користувачів). Ця класифікація дозволяє системі диференціювати логіку обробки залежно від типу товару.

Для інтеграції з зовнішніми системами використовується поле «external_product_id», що посилається на відповідні записи в таблицях «retailer_products» (товари постачальників) та «flea_product» (товари користувачів). Важливо зазначити, що ці дві таблиці знаходяться в іншій базі даних (база даних для товарів постачальників та користувачів), що вимагає використання механізмів міжсервісної взаємодії для отримання деталей товарів.

На рисунку 4.6 зображено таблицю товарів постачальників другої бази даних.

Рисунок 4.6 – Таблиця товарів постачальників

Структура таблиць «flea_products» та «retailer_products» має схожість: обидві містять поля для назви товару (name), категорії (category), опису (description), зображення (image_url), ціни (price), кількості (count), ідентифікатора

користувача (user_id) та специфікацій у форматі JSON (specifications). Однак вони відрізняються контекстом використання: «flea_products» відноситься до внутрішнього каталогу сервісу, тоді як «retailer_products» інтегрується з зовнішніми системами постачальників. Використання типу даних JSON для поля «specifications» дозволяє гнучко зберігати різноманітні технічні параметри товарів без жорсткої схемы.

На рисунку 4.7 зображено таблицю третьої бази даних.

Рисунок 4.7 – Таблиця товарів користувачів

Архітектура системи передбачає розділення відповідальностей: сервіс управління замовленнями не включає повну інформацію про товари, а лише посилається на зовнішні джерела через «external_product_id». Це підкреслює модульність системи та необхідність синхронізації даних між різними компонентами. Обмеження на два типи товарів («RETAILER» та «FLEA») спрощує логіку обробки, але може вимагати розширення при додаванні нових категорій. Загалом, представлена структура бази даних відображає баланс між уніфікацією та гнучкістю, що є критичним для масштабованих систем управління товарами.

5 РОБОТА КОРИСТУВАЧА З СИСТЕМОЮ

В ролі користувача виступає клієнтський застосунок. Сам клієнтський застонук – frontend частина системи. Розробник клієнтськогї частини може перейти до розділу документації щодо вхідних точок системи – API.

На рисунку 5 наведено фрагмент інтерфейсу доступного розробникам, котрі можуть дослідити API системи.

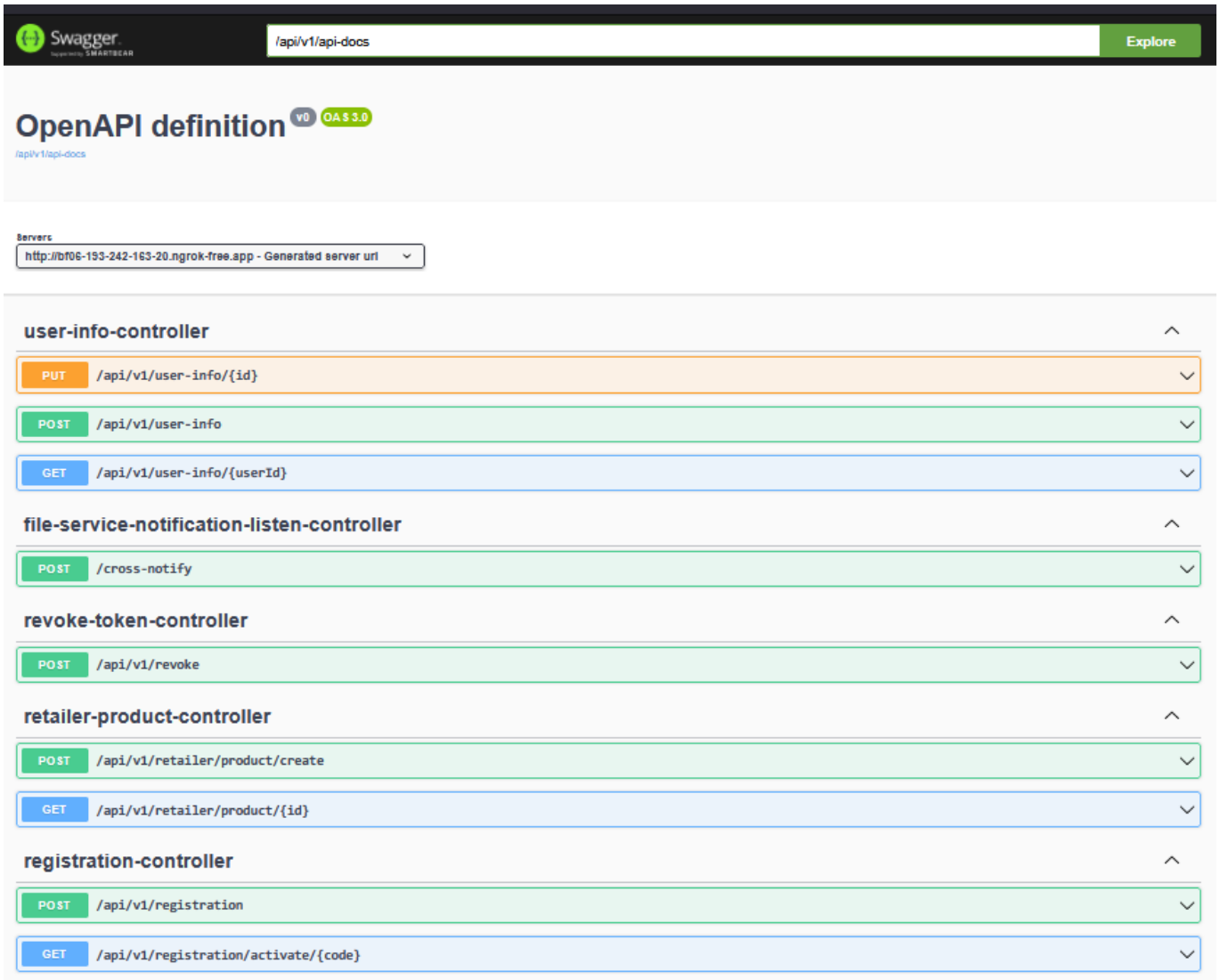

Рисунок 5.1 – Swagger документація маркетплейс системи

У системі реалізовано функціональність авторизації та реєстрації користувачів, яка є критично важливою частиною бекенду. Для взаємодії фронт-

енд розробника з цією частиною передбачено набір REST API ендпоінтів, які дозволяють здійснювати всі необхідні дії з боку клієнта. Фронт-енд взаємодіє із сервером за допомогою HTTP-запитів, обробляючи відповідні JSON-структури.

Після відкриття сайту користувач має можливість перейти на сторінку входу або реєстрації. На сторінці реєстрації фронт-енд формує HTTP POST запит до ендпоінту /api/auth/register, передаючи у тілі запиту об'єкт, що містить ім’я, електронну пошту, пароль та інші необхідні поля. Бекенд проводить валідацію, створює обліковий запис у базі даних і повертає відповідь із підтвердженням успішної реєстрації. Якщо користувач вже існує, у відповідь надсилається повідомлення про помилку.

На сторінці входу фронт-енд надсилає POST запит на /api/auth/login, передаючи у тілі запиту логін і пароль. Якщо облікові дані вірні, бекенд повертає токен авторизації (JWT), який фронт-енд має зберігати у пам’яті (наприклад, у localStorage чи sessionStorage) для подальшого використання. Усі подальші запити до захищених частин API повинні містити цей токен у заголовку Authorization з префіксом Bearer (або без нього, система розпізнає відповідний префікс і потребу в його зміні).

Після отримання токена фронт-енд може використовувати його для запитів, наприклад, для отримання інформації про користувача, звертаючись до /api/users/me. Бекенд, за допомогою фільтра безпеки, перевіряє токен, розпізнає користувача та повертає відповідні дані.

У разі, якщо токен недійсний або закінчився, сервер поверне статус 401, і фронт-енд має переадресувати користувача на сторінку входу. У разі успішної авторизації фронт-енд може надалі виконувати запити до інших захищених ендпоінтів, таких як перегляд історії замовлень, редагування профілю, керування кошиком тощо.

Використання токену потрібне тільки для захищених ендпоінтів, наприклад, для доступу до функціоналу постачальника треба мати відповідний акаунт, і відповідні права доступу.

Фронт-енд також має самостійно реалізовувати механізм виходу з системи шляхом видалення токена з пам’яті, що призведе до припинення доступу до захищених ресурсів.

Уся взаємодія побудована відповідно до стандартної схеми авторизації через JWT, що дозволяє розробникам фронт-енду легко інтегрувати логіку автентифікації та забезпечити захист даних.

У системі реалізовано два основних типи торгових майданчиків: розділ для рітейлерів та блошиний ринок. Обидва ринки мають відмінності у доступі до функціональності, але працюють через загальний набір API, який фронт-енд може викликати, не турбуючись про внутрішню реалізацію логіки.

Для створення товарів у рітейлерській частині системи необхідна наявність підтвердженого облікового запису продавця. Такий обліковий запис створюється через спеціальний процес реєстрації, після чого адміністрація платформи або автоматизований механізм підтвердження активує статус рітейлера. Тільки після цього користувач отримує доступ до ендпоінтів створення та керування товарами рітейлера, таких як POST /api/retailer/products, PUT /api/retailer/products/{id}, DELETE /api/retailer/products/{id} тощо. Якщо непідтверджений користувач намагається звернутись до цих маршрутів, система повертає помилку з відповідним статусом доступу.

На відміну від цього, блошиний ринок доступний усім зареєстрованим користувачам без необхідності підтвердження продавця. Будь-хто може створювати свої товари через публічний ендпоінт POST /api/fleamarket/products, вказуючи мінімально необхідну інформацію. Тут не проводиться попередня модерація облікового запису, що значно спрощує і пришвидшує додавання оголошень. Це дозволяє реалізувати концепцію відкритого обміну товарами між звичайними користувачами без бар’єрів у вигляді додаткових перевірок.

Незважаючи на логічну розділеність системи на два типи ринку, оформлення замовлення в системі реалізовано централізовано через один ендпоінт: POST /api/orders. Цей ендпоінт приймає загальну структуру замовлення, яка включає ідентифікатор товару, кількість, контактну інформацію та спосіб

доставки. Система сама визначає тип товару (рітейлерський чи блошиний) і запускає відповідну внутрішню бізнес-логіку обробки замовлення. Завдяки цьому фронт-енд може мати єдину логіку оформлення замовлення, не розділяючи товари за джерелом, і просто підставляти необхідні дані у загальний формат запиту.

Таким чином, для фронт-енд розробника система надає просту у використанні і передбачувану API-структуру: необхідно лише дотримуватись правил доступу до певних ендпоінтів залежно від типу облікового запису і характеру торгового майданчика. Усі перевірки, маршрутизація та бізнес-правила обробляються на бекенді, що зменшує складність реалізації з боку клієнта.

ВИСНОВКИ

В результаті виконання роботи здійснено дослідження ринку існуючих рішень у сфері електронної комерції, що дозволило виявити типові проблеми українського e-commerce: недовіра до продавців, складність з поверненнями, непрозорі умови роботи та нестача інструментів для ефективного просування. Це стало основою для формування вимог до архітектури майбутньої платформи.

На основі глибокого аналізу українського ринку електронної комерції було виявлено ключові обмеження наявних рішень: слабкий контроль якості продавців, недостатню прозорість умов для покупців, обмежені можливості масштабування платформ, а також відсутність гнучких інструментів для пошуку та просування. Відштовхуючись від цих недоліків, сформульовано чіткі функціональні та нефункціональні вимоги до системи, які включали безпечність, надійність, масштабованість, гнучку модульну архітектуру та сучасний користувацький досвід.

Відповідно до цих вимог було спроєктовано архітектуру з чітким поділом модулів, де окремий акцент зроблено на індексації та пошуку даних за допомогою Elasticsearch. Вибір технологічного стеку враховував потребу в масштабованості та стабільності – рішення дозволяє платформі працювати однаково ефективно як на рівні малого інтернет-магазину, так і великої торгової мережі.

Таким чином, усі поставлені задачі реалізовано. Система поєднує технічну ефективність з реальним вирішенням актуальних бізнес-проблем українського e-commerce, що формує довіру до платформи як з боку покупців, так і з боку продавців.

Перспективи подальшого розвитку охоплюють інтеграцію з зовнішніми платіжними та логістичними сервісами, розширення системи аналітики для бізнес-користувачів, а також впровадження рекомендаційних механізмів на основі машинного навчання для персоналізованого досвіду користувачів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Top websites in Ukraine (Retail Industry). URL: https://sv.semrush.com/trending-websites/ua/retail (дата звернення 15.04.2025).

2. Silberschatz A., Sudarshan S., Korth H. F. Database systems concepts. 5th ed. McGraw-Hill Science/Engineering/Math, 2005. 1168 p.

3. Grand, M. Java Enterprise Design Patterns Vol. 3: Patterns in Java. Wiley & Sons, Incorporated, John, 2009. 496 p.

4. Schmidt M.T., Hutchison B., Lambros P., Phippen R. Service-Oriented Architecture and Event-Driven Architecture: J2EE Integrated Solutions. Boston: Addison-Wesley Professional, 2008. 272 p.

5. Steele G. L. Java Language Specification. Pearson Technology Group Canada, 2014. 792 p.

6. Rumbaugh J. The unified modeling language reference manual. 2th ed. Boston : Addison-Wesley, 2005. 721 c.

7. Navathe S. B., Elmasri R. Database systems: models, languages, design, and application programming. Pearson Education, Limited, 2010. 1200 p.

8. System architecture and system design. New York, N.Y : McGraw-Hill, 1989. 494 p.

9. Gheorghe R., Hinman M. L., Russo R. Elasticsearch in Action. Manning Publications Co. LLC, 2015, 496 p.

10. Kuc B. R. Mastering ElasticSearch. Packt Publishing, 2013. 386 p.

11. Walls C. Spring in Action. 6th ed. Manning Publications Co. LLC, 2018. 520 p.

12. Design patterns: Elements of reusable object-oriented software / ed. by G. Erich. Reading, Mass : Addison-Wesley, 1995. 395 p.

13. SOLID Design Principles: Design Principles. Independently Published, 2019. 25 p.

14. Spring Security in Action. Manning Publications Company, 2020. 550 p.

15. Moore J. OAuth 2.0: Introduction to API Security with OAuth 2.0. CreateSpace Independent Publishing Platform, 2016. 118 p.

16. HTTP: Pocket Reference. O'Reilly, 2000. 80 p.

17. Domain-Driven Design Distilled. Addison-Wesley Professional, 2016. 176 p.

18. HTTP developer's handbook. Indianapolis, Ind : Sams, 2003. 282 p.

19. Carver R. H., Tai K.-C. Modern Multithreading: Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs. Wiley & Sons, Incorporated, John, 2007. 480 p.

20. Wolff E. Practical Microservices. Apress, 2017. 310 p.

ДОДАТОК А

АПРОБАЦІЯ РОБОТИ

УКР.НТУУ”КПІ ім. Ігоря Сікорського”_ІАТЕ_ЦТЕ_ТР-11

Аркушів 4

Київ – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

СУЧАСНІ ПРОБЛЕМИ НАУКОВОГО ЗАБЕЗПЕЧЕННЯ ЕНЕРГЕТИКИ

Матеріали XXII Міжнародної
науково-практичної конференції
молодих вчених і студентів
м. Київ, 22–25 квітня 2025 року

ТОМ 2

Київ- 2025

UDC 004.8:004.93:519.6:6

BS's programme 4th year Shust K.E.
BS's programme 4th year Bratchykova N.V.
[1] Prof., doc.eng.sc. Shushura O.M.
https://scholar.google.com.ua/citations?user=beUGko0AAAAJ&hl=uk&oi=sra

[1] Igor Sikorsky Kyiv Polytechnic Institute

DEVELOPMENT OF AN E-COMMERCE SYSTEM FOR ONLINE TRADING

Statement of the problem and its relevance. With the active development of e-commerce, online platforms are becoming a key tool for the sale of goods and services. The main challenges in creating such systems are ensuring a convenient user interaction with the platform, automating order processing, and the scalability of the architecture. Special attention should be paid to the organization of the seller's cabinet, which must provide functionality for adding products, viewing sales statistics, and managing orders.

The relevance of this work is determined by the need to create a universal e-commerce system that combines functionality for both buyers and sellers, considering modern approaches to software architecture design..

Analysis of the latest research. Currently, there are many ready-made solutions for creating online stores, such as Shopify, WooCommerce, and OpenCart. Most of them are focused on creating monolithic applications or use outdated architectural approaches.

Modern approaches to building web applications, such as feature-sliced design on the frontend and module-driven architecture on the backend, significantly improve scalability, responsibility distribution, and code maintainability.

In [1], a modular architecture approach for the backend based on Spring Boot is presented, which provides flexible integration of new features. Research [2] describes the concept of the feature-sliced approach in frontend development for better structuring of business logic.

Goal formulation. The aim of this work is to create a comprehensive e-commerce system that provides a seamless and efficient experience for both buyers and sellers. Buyers will be able to browse and search for products, add items to their cart, place orders, make secure payments, and track their order history. Meanwhile, sellers will have access to a dedicated management panel that allows them to list and update products, monitor inventory, analyze sales performance, and receive detailed reports on customer behavior and market trends. Additionally, the system will incorporate modern architectural principles to ensure scalability, security, and ease of maintenance.

The main part.

The developed system consists of two independent parts:

1. Frontend – implemented using React with the feature-sliced design approach, which divides functionality into domains (features, entities, widgets, shared).
2. Backend – built on Spring Boot based on the module-driven architecture, where each functional module is responsible for a specific business process.

The system provides comprehensive functionality for both users and sellers. It includes user registration and authorization, allowing users to create accounts and protect personal data. Registered users can view their order history, manage personal information by updating contact details and profile settings. For convenient product search, the system implements keyword search and product filtering by categories. The system also offers product recommendations based on the user's previous purchases.

Sellers gain access to a personal dashboard where they can add, edit, and delete products, as well as view sales statistics to analyze their business performance.

The system features a convenient order processing mechanism, including the selection of payment and delivery methods. To enhance user interaction, a feedback and product rating functionality is implemented. The seller's dashboard includes tools for monitoring product availability and managing prices..

Figure 1 shows the overall system workflow diagram:

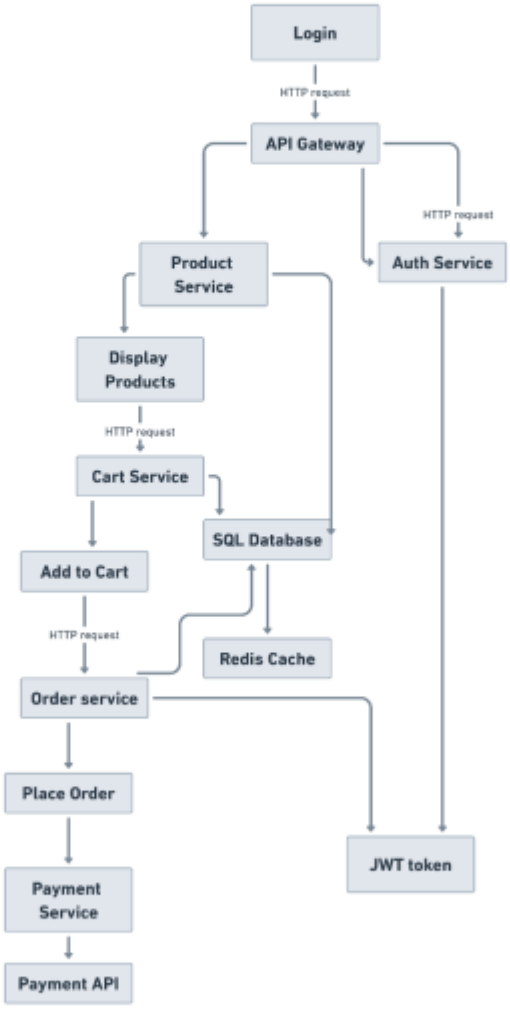

Figure 1 – Generalized diagram of the online trading system

Key points include:

1. Frontend (React, Redux, React Query)
   a) Feature-sliced structure;
   b) Interaction through Axios;

c) Query caching through Redux;
d) Authorization via JWT.

2. Backend (Spring Boot + Microservices Architecture)
   a) Security Layer for access contro;
   b) Separate services for authentication, products, cart, orders, and payments;
   c) Database (SQL);
   d) Redis for product caching;
   e) API protection through JWT.
   f) Integration with third-party payment services.
3. Main interaction scenarios:
   a) Authorization (Login);
   b) Retrieving the product list;
   c) Adding a product to the cart;
   d) Placing an order;
   e) Payment confirmation.

The e-commerce system is built on a client-server architecture with interaction through a REST API. The React frontend uses a feature-sliced approach, Axios for HTTP requests, and caching via Redux. Authorization is implemented using JWT tokens.

The Spring Boot backend follows a modular architecture, where each service is independently responsible for core functionalities such as authentication, product management, cart handling, order processing, and payments. This modularity allows for easy scalability and flexibility in extending or modifying services. Data is stored in an SQL database, providing reliable and structured storage, while Redis is used for caching products to enhance the speed and efficiency of data retrieval. API security is maintained through JWT, protecting against unauthorized access and ensuring secure data transmission.

The system is designed to handle key scenarios such as user authorization, product viewing, cart management, order placement, and payment processing. It ensures smooth user interactions by validating input data, storing necessary information in the database, and updating order statuses in real time. Additionally, the modular approach allows for continuous improvement and the seamless integration of new features as the e-commerce platform evolves.

Conclusions. The e-commerce system ensures secure and efficient interaction between users and the platform through modern technologies. It supports registration, authorization, product search and filtering, as well as personal data management. To enhance user experience, product recommendations, order history viewing, and a seller's personal dashboard for managing products and sales statistics are implemented. Integration with payment services and API protection through JWT make the system fully functional for online commerce.

References:

1. DDD Bounded Contexts and Java Modules. URL: https://www.baeldung.com/java-modules-ddd-bounded-contexts (access date 14.02.2025).

2. Feature-Sliced Design. Architectural methodology for frontend projects. URL: https://feature-sliced.design/ (access date 20.02.2025).

ДОДАТОК Б

ТЕКСТ ПРОГРАМНОГО МОДУЛЯ

«КОНФІГУРАЦІЯ БЕЗПЕКОВОЇ ЧАСТИНИ ЗАСТОСУНКУ ТА ЕНДОІНТУ РЕЄСТРАЦІЇ КОРИСТУВАЧІВ»

УКР.НТУУ”КПІ ім. Ігоря Сікорського”_ІАТЕ_ЦТЕ_ТР-11

Аркушів 8

Київ – 2025

Програмні засоби:

- мова програмування – Java;
- фреймворк – Spring Boot (з модулями Spring Web та Spring Security);
- бібілотека структурного патерну Mediator – Pipelinr;
- бібліотека для валідації вхідних даних – Jakarta Validation;
- бібліотека для створення шаблонного коду – Lombok.

```
package com.blueprint.authorization.input;


import an.awesome.pipelinr.Pipeline;
import
com.blueprint.authorization.model.command.GuestAccountDataTransitionCommand;
import com.blueprint.authorization.service.AuthenticationService;
import com.blueprint.security.model.pojo.RequestMetaInfo;
import com.blueprint.security.util.RequestMetaInfoHolder;
import com.http.paths.ApiRouteConstants;
import com.blueprint.common.model.dto.LoginRequest;
import com.blueprint.common.model.dto.PasswordResetRequest;
import com.blueprint.common.model.dto.auth.AuthenticationResponse;
import jakarta.validation.Valid;
import lombok.RequiredArgsConstructor;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
```

```
import java.util.concurrent.CompletableFuture;


@RestController
@RequiredArgsConstructor
@RequestMapping(ApiRouteConstants.API_V1_AUTH)
public class AuthenticationController {

    private final AuthenticationService authenticationService;
    private final Pipeline mediator;


    @PostMapping(ApiRouteConstants.LOGIN)
    public ResponseEntity<AuthenticationResponse> login(@RequestBody @Valid
LoginRequest request) {
        AuthenticationResponse authenticationResponse =
authenticationService.login(request);


        //RequestMetaInfoHolder is thread-local, and there is a try to access it from a
different thread than the one it was originally set on.
        RequestMetaInfo requestMetaInfo =
RequestMetaInfoHolder.getCurrentClientRequestMetaInfo();
        CompletableFuture.runAsync(() ->
            mediator.send(new
GuestAccountDataTransitionCommand(request.getUsername(), requestMetaInfo))
```

```
        );

        return ResponseEntity.ok(authenticationResponse);
    }

    @GetMapping(ApiRouteConstants.FORGOT_EMAIL)
    public ResponseEntity<String> forgotPassword(@PathVariable String email) {
        return ResponseEntity.ok(authenticationService.sendPasswordResetCode(email));
    }

    @PostMapping(ApiRouteConstants.RESET)
    public ResponseEntity<String> passwordReset(@RequestBody @Valid
PasswordResetRequest passwordReset) {
        return ResponseEntity.ok(authenticationService.passwordReset(passwordReset));
    }
}
```

```
package com.blueprint.security.config;

//import com.marketplace.common.model.constant.SecurityConstants;
import com.blueprint.security.JwtFilter;
import com.blueprint.security.JwtProvider;
import com.blueprint.security.domain.RequestMetaInfoCreator;
```

```
import com.blueprint.security.domain.RequestMetaInfoFilter;

import com.blueprint.security.domain.SessionGuestAuthenticationFilter;

import com.blueprint.security.service.impl.UserInfrastructureService;

import com.http.paths.ApiRouteConstants;

import com.blueprint.common.model.constant.Role;

import com.blueprint.security.JwtAuthenticationConfigurer;

import lombok.RequiredArgsConstructor;

import org.springframework.boot.autoconfigure.security.servlet.PathRequest;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.core.annotation.Order;

import
org.springframework.security.authentication.AnonymousAuthenticationProvider;

import org.springframework.security.authentication.AuthenticationManager;

import
org.springframework.security.config.annotation.authentication.configuration.Authentica
tionConfiguration;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import
org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigure
r;

import org.springframework.security.config.http.SessionCreationPolicy;
```

```
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.web.SecurityFilterChain;
import
org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilt
er;
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

import java.util.List;



@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfig implements WebMvcConfigurer{

    public static final String[] GUEST_ALLOWED_RESOURCES = {
            ApiRouteConstants.LOGIN,
            ApiRouteConstants.LOGOUT,
            ApiRouteConstants.API_V1 + "/**",
            ApiRouteConstants.USER_ROLE,
    };
    private static final String[] AUTH_METRICS = {
```

```
        "/actuator/health/**", "/actuator/prometheus"
    };
    //TODO: connect securityConstants
    //private final SecurityConstants securityConstants;
    private final JwtAuthenticationConfigurer jwtAuthenticationConfigurer;

    @Bean
    @Order(2)
    public SecurityFilterChain securityFilterChain(HttpSecurity http,
                                                   JwtFilter jwtFilter,
                                                   RequestMetaInfoCreator requestMetaInfoCreator,
                                                   SessionGuestAuthenticationFilter
sessionGuestAuthenticationFilter,
                                                   RequestMetaInfoFilter requestMetaInfoFilter) throws
Exception {

        http
                .csrf(AbstractHttpConfigurer::disable)
                .cors(cors -> cors.configurationSource(request -> {
                    var config = new CorsConfiguration();
                    config.addAllowedOriginPattern("*"); //to make cookies work with
setAllowCredentials
                    config.setAllowedMethods(List.of("*"));
```

```
                config.setAllowedHeaders(List.of("*"));
                config.setMaxAge(3600L);
                config.setAllowCredentials(true); //for cookie transfer
                return config;
            }))
            .sessionManagement(management -> management.sessionCreationPolicy(SessionCreationPolicy.STATELESS)) //doesn't remember the user
            .authorizeHttpRequests((authorizeHttpRequests) ->
                    authorizeHttpRequests

//.requestMatchers(CLIENT_ALLOWED_AUTHORIZED_RESOURCES).hasAnyAuthority("USER")

.requestMatchers(GUEST_ALLOWED_RESOURCES).permitAll()

.requestMatchers(AUTH_METRICS).hasRole(Role.SERVICE.toString())

//.requestMatchers(securityConstants.getAllowedResources()).permitAll() //allow access to static resources
                        .requestMatchers(
                            PathRequest
                                .toStaticResources()
                                .atCommonLocations()
                        ).permitAll()
```

```
                .anyRequest().authenticated()
            )
            .apply(jwtAuthenticationConfigurer)
            ;

        return http.build();
    }

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

}
```