МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»
ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«ОНЛАЙН-МАРКЕТПЛЕЙС ЗАСОБАМИ SPRING FRAMEWORK»

Рівень «бакалавр»
Галузь знань 12 «Інформаційні технології»,
спеціальність 121 «Інженерія програмного забезпечення»

Виконав здобувач 4 курсу
Мустафаєв Тимур Зайірович

Керівник к.т.н., доцент
Трофименко Олена Григорівна

Рецензент к.т.н., доцент
Задерейко Олександр Владиславович

Одеса – 2024

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»

Факультет кібербезпеки та інформаційних технологій
Кафедра інформаційних технологій
Галузь знань 12 Інформаційні технології
Спеціальність 121 «Інженерія програмного забезпечення»
Назва освітньої програми «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри інформаційних технологій
доцент Н.І. Логінова
_____ «___» _____20__ року

З А В Д А Н Н Я
на кваліфікаційну (бакалаврську) роботу
здобувачу Мустафаєву Тимуру Зайіровичу

1. Тема роботи: «Онлайн-маркетплейс засобами Spring Framework»
   Керівник роботи: к.т.н, доцент Трофименко Олена Григорівна
   затверджені наказом НУ «ОЮА» № 809-06 від «02» квітня 2024 року
2. Дата видачі завдання «15» вересня 2023 року.
3. Строк подання здобувачем роботи «20» травня 2024 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Формування функціональних вимог	10.10.2023	
2	Аналіз та вибір засобів розробки	30.10.2023	
3	Розробка архітектури вебзастосунку	20.11.2023	
4	Проєктування бази даних	25.12.2023	
5	Розробка серверної частини	12.02.2024	
6	Розробка клієнтської частини	19.03.2024	
7	Тестування функціональності сайту	09.04.2024	
8	Оформлення звітної частини	19.05.2024	

Здобувач _____
(підпис) (прізвище та ініціали)

Керівник роботи _____
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Мустафаєв Т. З. Онлайн-маркетплейс засобами Spring Framework. – Кваліфікаційна робота бакалавра за спеціальністю 121 «Інженерія програмного забезпечення», освітньою програмою «Інженерія програмного забезпечення».

Кваліфікаційна робота викладена на 106 сторінках основного тексту і 117 сторінках загального тексту, містить 3 розділи, 55 рисунків, 16 таблиць, 1 додаток, 25 використаних джерел.

Мета роботи – розробка платформи онлайн-маркетплейсу для продажу одягу засобами Spring Framework.

Об’єкт дослідження – специфіка розробки онлайн-маркетплейсу та організація контенту на його вебсторінках.

Предмет дослідження – технології Spring Framework для розробки онлайн-маркетплейсу.

У першому розділі виконано аналіз предметної області, аналіз наявних програмних аналогів та вибір засобів розробки. У другому розділі здійснено проєктування вебзастосунку, моделювання програмної системи, розроблено архітектуру програмного застосунку. У третьому розділі описано розробку клієнтської та серверної частини вебзастосунку, виконано тестування та оцінку якості розробленого онлайн-маркетплейсу.

Ключові слова: онлайн-маркетплейс, одяг, мікросервіс, RESTful, Java, Spring, застосунок, шлюз, авторизація, автентифікація, реєстрація.

ABSTRACT

Mustafaiev T. Z. Online marketplace using Spring Framework. – Bachelor's thesis in specialty 121 "Software Engineering", educational program "Software Engineering".

The qualification work is set out on 102 pages of the main text and 129 pages of the general text, contains 3 sections, 55 figures, 16 tables, 1 appendix, 25 references.

The purpose of the work is to develop an online marketplace platform for selling clothes using the Spring Framework.

The object of research is the specifics of developing an online marketplace and organising content on its web pages.

The subject of the study is Spring Framework technologies for developing an online marketplace.

The first section analyses the subject area, analyses existing software analogues, and selects development tools. The second section describes the design of the web application, modelling of the software system, and the development of the software application architecture. The third section describes the development of the client and server parts of the web application, testing and quality assessment of the developed online marketplace.

Keywords: online marketplace, clothing, microservice, RESTful, Java, Spring, application, gateway, authorization, authentication, registration.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ 7
ВСТУП 8
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИБІР ЗАСОБІВ РОЗРОБКИ 10
1.1 Специфіка програмної розробки онлайн-маркетплейсу 10
1.2 Аналіз наявних аналогів 14
1.2.1 Amazon 15
1.2.2 eBay 15
1.2.3 Allegro 16
1.2.4 Etsy 16
1.2.5 AliExpress 17
1.3 Розробка функціональних вимог та виявлення варіантів використання створюваного програмного продукту 18
1.3.1 Функціональні вимоги для неавторизованих користувачів 19
1.3.2 Функціональні вимоги для користувачів у ролі «Покупець» 20
1.3.3 Функціональні вимоги для користувачів у ролі «Продавець» 21
1.3.4 Функціональні вимоги для користувача у ролі «Адміністратор» 22
1.4 Вибір засобів розробки 23
1.4.1 Засоби розробки backend частини 24
1.4.2 Засоби розробки frontend частини 26
1.4.3 Засоби розробки бази даних застосунку 26
1.5 Маркетингові інструменти в створенні та просуванні програмного продукту 29
1.5.1 Карта клієнта 29
1.5.2 Аналіз ринку 30
1.5.3 Аналіз споживачів 31
1.5.4 SMM (соціальні медіа) 31
1.6 Висновки до розділу 1 32
2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ПРОДУКТУ 33
2.1 Інформаційна модель предметної області 33
2.1.1 Сутності моделі 33
2.1.2 Стосунки 34
2.1.3 Сценарії взаємодії 35

2.2 Архітектура програмного застосунку 36
2.2.1 MVC 37
2.2.2 Rest API 38
2.2.3 Архітектура клієнт-сервер 39
2.2.4 Мікросервісна архітектура 40
2.3 Проєктування бази даних 41
2.4 Моделювання програмної системи 42
2.4.1 Діаграми прецендентів 44
2.4.2 Діаграма класів 47
2.4.2 Діаграми активності 48
2.5 Висновки до розділу 2 52
3 РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ 53
3.1 Розробка запитів до бази даних відповідно до функціоналу застосунку 53
3.1.1 Додавання залежностей 54
3.1.2 Налаштування параметрів підключення до бази даних 55
3.1.3 Створення класу сутності 56
3.1.4 Створення інтерфейсу репозиторія 59
3.2 Розробка серверної частини з використанням Spring Framework 60
3.2.1 Eureka Server. 62
3.2.2 Docker контейнери. 64
3.2.3 API Gateway. 68
3.2.4 AWS S3 Storage. 75
3.2.5 RESTful мікросервіси. 83
3.3 Розробка клієнтської частини засобами Spring Framework 86
3.4 Функціонал застосунку 90
3.5 Тестування та оцінка якості застосунку 96
3.5 Висновки до розділу 3 99
ВИСНОВКИ 101
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 103
ДОДАТКИ 105
Додаток А. Фрагменти програмного коду 105

ПЕРЕЛІК СКОРОЧЕНЬ

API – Application Programming Interface, інтерфейс програмування застосунків;

AWS – Amazon Web Services, веб-сервіси Amazon;

B2B – Business-to-Business, платформи різних постачальників, які полегшують транзакції між компаніями;

B2C – Business-to-Consumer, платформи, які з'єднують компанії безпосередньо зі споживачами;

C2C – Consumer-to-Consumer, торгові майданчики від споживача до споживача;

CRUD – Create Read Update Delete, створення читання оновлення видалення;

HTTP – HyperText Transfer Protocol, протокол передачі гіпертексту;

JPA – Java Persistence API, стандартизований інтерфейс для Java ORM фреймворків;

JSON – JavaScript Object Notation, нотація об'єктів JavaScript;

JWKS – JSON Web Key Set, набір ключів веб-токенів JSON;

JWT – JSON Web Token, веб-токен на основі JSON;

MVC – Model-View-Controller, архітектурний шаблон для розробки програмного забезпечення, який допомагає розділити програму на три основні частини: Модель, Подання та Контролер;

MVP – Minimum Viable Product, мінімальний життєздатний продукт. Це тестова версія торгового майданчика, яка знайомить продавців і покупців з основними функціями платформи;

ORM – Object-relational mapping, об'єктно-реляційна проєкція;

REST – Representational State Transfer, репрезентативна передача стану;

S3 – Simple Storage Service, служба простого зберігання;

SDK – Software Development Kit, набір засобів розробки програмного забезпечення;

SMM – Social Media Marketing, маркетинг у соціальних мережах;

URL – Uniform Resource Locator, уніфікований локатор ресурсів;

XML – eXtensible Markup Language, розширювана мова розмітки.

ВСТУП

На сьогодні комерційні компанії масово розвивають надання своїх товарів та послуг через вебпростір. Сучасний ринок одягу повсюдно переходить в онлайн, адже інтернет-магазини дозволяють підприємцям знизити витрати на оренду приміщень, сплату комунальних послуг, оплату зарплатні продавців та іншого персоналу, що в свою чергу впливає на зниження ціни на товар. Нижчі ціни, порівняно з «магазинними», приваблюють більше людей на онлайн-шопінг. Крім того, швидкість та зручність обслуговування є суттєвою перевагою для вибору клієнтом інтернет-обслуговування. Розвиток інтернет-комерції, в свою чергу, призводить до залучення нових ринків у вебпростір, серед яких і ринок одягу.

При цьому не кожен малий підприємець може собі дозволити розробку власного сайту для інтернет-магазину. Малий бізнес потребує економного способу торгівлі в мережі інтернет. Наразі численні дрібні підприємці займаються продажом одягу. Відтак, дослідження та розробка онлайн-маркетплейсу для продажу одягу є актуальними, а тема кваліфікаційної роботи, присвячена розробленню онлайн-маркетплейсу, – перспективною.

Мета роботи – розробка платформи онлайн-маркетплейсу засобами Spring Framework для продажу одягу.

Об’єкт дослідження – специфіка розробки онлайн-маркетплейсу та організація контенту на його вебсторінках.

Предмет дослідження – технології Spring Framework для розробки онлайн-маркетплейсу.

Відповідно до поставленої мети завданнями роботи є:

– аналіз сучасних тенденцій ринку одягу в онлайн-сфері, вивчення популярних моделей бізнесу, особливостей споживчої поведінки та конкурентного середовища;

– вивчення можливостей Spring Framework, дослідження основних компонентів Spring Framework: Spring Boot, Spring Cloud, Spring Security, Spring Data JPA тощо;

– проєктування архітектури онлайн-маркетплейсу, визначення структури застосунку, розробка структури бази даних, логіки бізнес-процесів та зручного інтерфейсу користувача;

– розроблення функціоналу маркетплейсу.

Методами дослідження є:

– вивчення документації Spring Framework та аналіз інших літературних джерел;

– порівняльний аналіз Spring Framework з іншими фреймворками для розробки вебзастосунків.

– вивчення сучасних тенденцій ринку одягу в онлайн-сфері.

Практичне значення отриманих результатів:

– *для підприємців та бізнесу*: підприємці можуть користуватись онлайн-магазином одягу за допомогою функціонального брендплейсу. У сфері онлайн-торгівлі це може бути корисно для тих, хто планує розпочати власний бізнес. Маркетплейс дозволить розширити аудиторію та забезпечить легкий доступ для покупців;

– *для клієнтів:* онлайн-маркетплейси дозволяють клієнтам легко вибирати та купувати одяг в інтернеті. Вони можуть переглядати товари в каталозі, додавати їх до кошика та робити замовлення, не виходячи з дому;

– *для розвитку електронної комерції:* створення онлайн-маркетплейсу сприятиме розвитку електронної комерції в цілому. Це зробить онлайн-торгівлю більш зручною та безпечною, а також збільшить обсяги продажів.

Результати роботи були апробовані у науковій публікації автора:

1. Мустафаєв Т.З. Специфіка веброзробки онлайн-маркетплейсу. *Інформаційне суспільство: проблеми та перспективи* : матер. IX всеукр. наук.-практ. конф. (м. Одеса, 24 травня 2024 р.). Одеса, 2024. (депоновано).

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИБІР ЗАСОБІВ РОЗРОБКИ

1.1 Специфіка програмної розробки онлайн-маркетплейсу

Онлайн-маркетплейси одягу є програмними платформами, які об'єднують продавців і покупців одягу. Вони надають можливість користувачам шукати, переглядати та купувати товари різних брендів і категорій (рис. 1.1) [1].

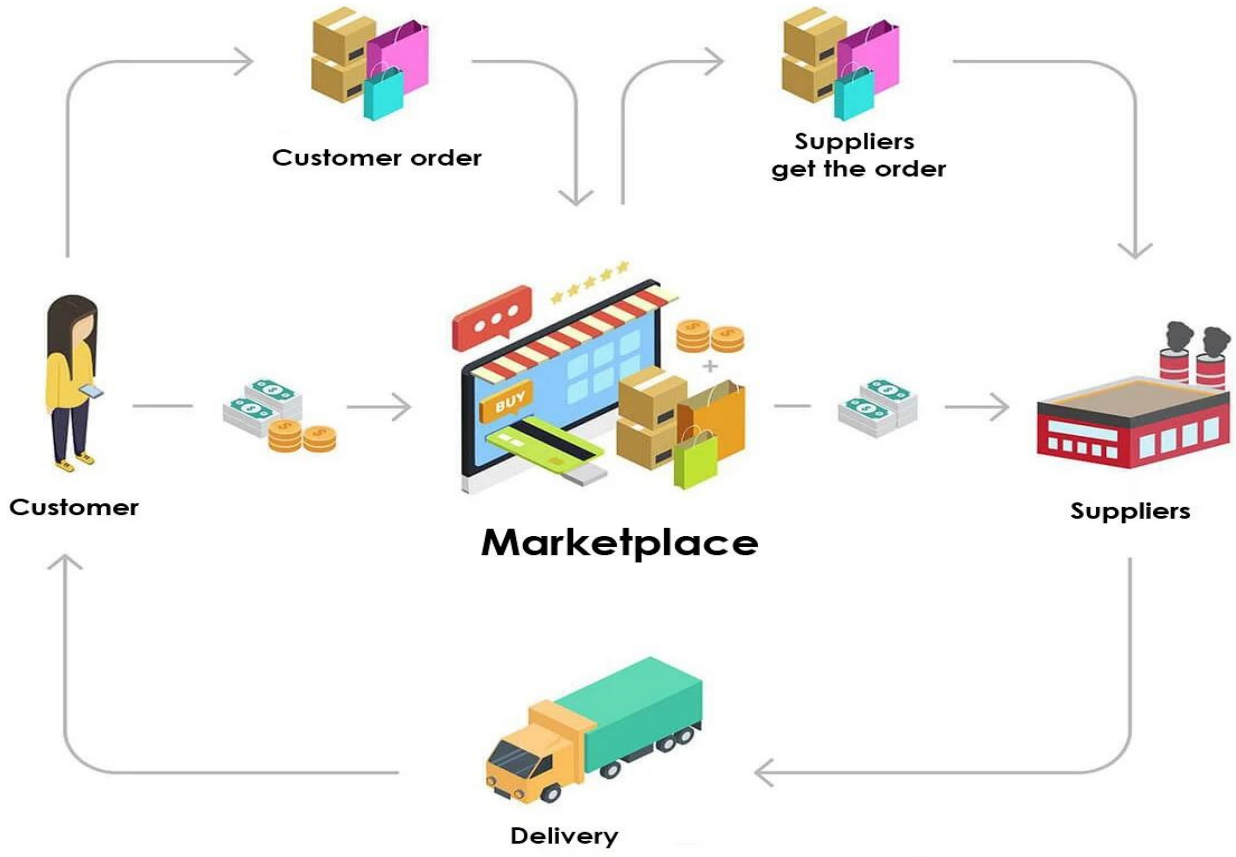


Рисунок 1.1 – Робота Marketplace

Попри те, що запуск нового ринку вимагає зусиль й інвестицій, це дуже прибуткова ніша. Інтернет-ринки настільки подобаються продавцям і покупцям, що майбутній постачальник програмного забезпечення гарантовано матимете багато зацікавлених користувачів. Адже цифрова трансформація триває, і ринок онлайн-комерції продовжуватиме зростати. Тому, навіть у висококонкурентному середовищі нові гравці завжди можуть знайти способи увійти та заповнити нові ніші, щоб створити базу лояльних клієнтів.

За даними досліджень [1] 47% цифрових покупок у всьому світі відбуваються через платформи онлайн-ринків і лише 6% покупців B2B зараз не користуються онлайн-ринками. За прогнозами, протягом наступних 5 років 75% витрат на закупівлі B2B відбуватимуться через онлайн-ринки. Наразі 53% роздрібних торговців використовують інструменти, які допомагають їм продавати через різні канали (Shopify).

Бізнес-модель ринку — це тип бізнес-моделі, де платформа з'єднує покупців і продавців, полегшуючи операції між ними. Платформа зазвичай надає інструменти та послуги для безперебійних і безпечних транзакцій і отримує від цього дохід [2]. Створення торгової платформи надає переваги для всіх сторін: власника платформи, продавців і покупців.

Можливі переваги для **власника** онлайн-маркетплейсу [3, 4]:

– *зручні моделі монетизації*: власники комерційних майданчиків можуть вибирати різні гнучкі моделі доходів: комісії, реєстраційний збір, платна підписка, внесок за реєстрацію лістингу товарів, фіксована плата, що стягується з продавця за кожну транзакцію, чи то поєднання кількох схем;

– *масштабований бізнес, доступ до глобальної ринкової ніші з мільярдами споживачів*: оскільки ринки розширюються, додавання нових продавців відповідає зростанню попиту, зберігаючи баланс між попитом і пропозицією для сталого росту;

– *додатковий дохід від реклами;*

– *інтерес користувачів та інвесторів*: широкий спектр пропозицій привертає інтерес користувачів, а ринкові інвестиції привертають значну увагу інвесторів.

Переваги використання комерційної платформи для **продавців** [3]:

– *постійний потік покупців;*

– *не потрібно перейматися питаннями самореклами*: продавці не потребують незалежного просування, заощаджуючи на розробці, підтримці та витратах на SEO;

– *покращений пошуковий рейтинг*: списки товарів маркетплейсів незмінно мають високі позиції в пошукових результатах, забезпечуючи кращу видимість, ніж окремі онлайн-магазини;

– *певна економія коштів*: не потрібно інвестувати в розробку інтернет-магазину на замовлення, немає потреби мати технічну команду для обслуговування сайту;

– *розширені можливості просування власного торгового бренду*: продавці отримують вигоду від збільшення впізнаваності бренду на торгових платформах, що призводить до більшого потоку клієнтів;

– *доступ до аналітики*: продавці отримують уявлення про перегляди продуктів, покупки та іншу статистику через свої облікові записи на торговому майданчику.

Перевагами для **покупців** онлайн-маркетплейсів є [1, 2]:

– *широкий асортимент продукції*: інтернет-ринки надають покупцям доступ до широкого спектра продуктів в одному місці.

– *зручні варіанти оплати та доставки;*

– *централізований інтерфейс*: система перегляду широкого вибору брендів на одній платформі;

– *надійні платформи*: рейтинги продавців та товарів, наявні відгуки підвищують довіру покупців до ринків, порівняно з окремими фірмами. Крім того, передбачена надійна політика відшкодування;

– *акції та зручність*: ринкові майданчики часто пропонують знижки, залучаючи й утримуючи користувачів, а процес бронювання в один клік підвищує зручність.

Інтернет-ринки протягом останніх років стрімко зростають у доходах і кількості користувачів. Так, до 2024 року продажі B2B можуть сягнути 3,6 трильйонів доларів , а сегмент B2C – 3,5 трильйонів доларів на рік [4].

Щоб створити успішну платформу, зручну для користувачів, важливо врахувати багато різних аспектів роботи онлайн-маркетплейсу для продажу одягу, а саме [2-4]:

– *велика кількість продавців:* завдяки онлайн-майданчикам для продажу одягу безліч продавців мають унікальну можливість просувати свої товари широкій аудиторії. Це можуть бути як великі бренди, так і незалежні дизайнери та дрібні підприємці. У кожного продавця є власний профіль і магазин на торговому майданчику. Це дає їм змогу виставляти товари на продаж, встановлювати ціни та безпосередньо спілкуватися з покупцями;

– *керування товарами:* можливість зручного додавання і редагування товарів, управління запасами і рекламними акціями;

– *керування замовленнями:* на торговому майданчику етапи процесу купівлі охоплюють процеси вибору товару, оформлення замовлення, відстеження доставки та оформлення можливого повернення. Продавці повинні мати можливість керувати замовленнями, оновлювати статуси і спілкуватися з покупцями в зручний для них спосіб, а покупці – бачити оновлення щодо замовлень у режимі реального часу;

– *відгуки та рейтинги* на торгових майданчиках забезпечують прозорість і довіру. Покупці можуть залишати відгуки про продавців і товари, ґрунтуючись на різних критеріях. Це дає змогу продавцям покращувати свої товари та послуги, а новим покупцям – робити більш усвідомлений вибір.

– *асортимент продукції:* широкий вибір товарів – ключова перевага платформи маркетплейса. Покупці можуть знайти різноманітні види одягу, взуття та аксесуарів – від класики до розкоші. Це робить онлайн-майданчик для продажу одягу зручним місцем для покупок для найрізноманітніших покупців.

Окремо слід зазначити важливий аспект розвитку, який стосується не тільки онлайн-маркетів, а й розробки будь-яких масово-орієнтованих застосунків. До цих аспектів належать [5]:

– *зручний інтерфейс користувача (UI):* привабливий і зручний дизайн, що дає змогу покупцям легко знаходити товари та інформацію, які їх цікавлять;

– *користувацький досвід (UX):* навігація сайтом має бути інтуїтивно зрозумілою, а процес вибору і купівлі товару – максимально простим і швидким;

– *масштабованість:* платформа має бути здатна впоратися зі збільшенням кількості користувачів і продуктів без шкоди для продуктивності;

– *безпека:* захист даних користувачів і транзакцій, використання шифрування і дотримання нормативних вимог;

– *пошукова оптимізація та маркетинг:* оптимізація для пошукових систем і використання маркетингових інструментів для залучення нових користувачів;

– *аналітика:* збір і аналіз даних про поведінку користувачів для поліпшення послуг і продуктів;

– *підтримка клієнтів:* системи зворотного зв'язку, чати підтримки та інші інструменти для спілкування з клієнтами.

Якщо ці аспекти враховуються при розробці платформи для онлайн-маркетплейсу, шанси на успішний запуск проєкту, який знайде відгук у аудиторії, набагато вищі. Це пов'язано з тим, що торговельний майданчик не тільки поєднує великі бренди, малі та середні підприємства, а й надає зручний інтерфейс як для покупців, так і для продавців [6].

Зручність використання – основний фактор, що впливає на інтерес користувачів до програмних продуктів. Це один із найважливіших аспектів, на які варто звернути увагу під час розробки. Призначений для користувача інтерфейс має бути зручним та інтуїтивно зрозумілим, щоб покупці не відволікалися на пошук кнопок або конкретних розділів онлайн-майданчика. Вони мають бути зосереджені лише на виборі товарів та їхніх характеристиках.

1.2 Аналіз наявних аналогів

Завдяки високому споживчому попиту на зручні онлайн-покупки цифрові ринки стали найкращим місцем для покупок. 37% постійних покупців роблять там покупки [4]. За статистичними даними досліджень станом на 2023 рік Amazon став найпопулярнішим онлайн-ринком у Європі, реєструючи приблизно 1,3 мільярда відвідувань на місяць [7]. Під час Amazon Prime Day у 2022 році, який проходив з 12 по 13 липня, малий і середній бізнес, який продає на Amazon,

заробив близько трьох мільярдів доларів США. Це значне зростання, порівняно з попереднім роком [8]. eBay посів друге місце серед найпопулярніших онлайн-ринків у Європі з понад 474 мільйонами відвідувань, а на третьому місці Allegro, який привернув близько 199 мільйонів відвідувань.

У роботі виконано аналіз конкурентів – онлайн-маркетплейсів одягу: Amazon, eBay, Allegro, Etsy та AliExpress.

1.2.1 Amazon

Amazon є глобальним гігантом у сфері електронної комерції, пропонуючи широкий асортимент товарів від електроніки до одягу та продуктів харчування. Він відомий своїми інноваційними рішеннями, такими як швидка доставка через Prime, а також можливістю продажу для малого та середнього бізнесу.

Переваги:

- великий вибір товарів;
- швидка доставка через Amazon Prime;
- надійність і довіра серед споживачів;
- розвинена інфраструктура підтримки продавців.

Недоліки:

- висока конкуренція між продавцями;
- високі комісії за продаж;
- залежність від платформи для малих бізнесів.

1.2.2 eBay

eBay є одним із піонерів в області онлайн-торгівлі, відомий своїм аукціонним форматом, а також функцією "купити зараз". Платформа дозволяє продавати як нові, так і вживані товари.

Переваги:

- можливість продажу вживаних товарів;
- велика база користувачів;
- аукціонна модель продажу;
- глобальний охват.

Недоліки:

- платформа менш популярна серед молодого покоління;
- ризики пов'язані з шахрайством;
- складність у використанні для нових продавців.

1.2.3 Allegro

Allegro є провідним онлайн-ринком у Польщі, пропонуючи широкий асортимент товарів і послуг. Платформа зосереджена на наданні зручного досвіду покупок для місцевих споживачів.

Переваги:

- лідер ринку в Польщі;
- інтуїтивно зрозумілий інтерфейс;
- низькі комісії для продавців;
- локалізований контент та обслуговування.

Недоліки:

- обмежений міжнародний охват;
- менший асортимент товарів порівняно з глобальними платформами;
- невелика база користувачів.

1.2.4 Etsy

Etsy спеціалізується на продажу ручної роботи, вінтажних виробів і товарів для творчості. Платформа підтримує малі бізнеси та індивідуальних підприємців, які створюють унікальні продукти.

Переваги:

- нішевий ринок для унікальних товарів;
- підтримка малих бізнесів і творчих підприємців;
- висока лояльність клієнтів;
- відсутність великих брендів, що сприяє меншій конкуренції.

Недоліки:

- обмежений асортимент товарів;
- відносно висока комісія за продаж;

– невелика база користувачів порівняно з великими платформами.

1.2.5 AliExpress

AliExpress, що належить Alibaba Group, є популярним онлайн-ринком для товарів з Китаю. Платформа відома низькими цінами і широким асортиментом товарів.

Переваги:

– низькі ціни на товари;
– широкий асортимент продукції;
– глобальний охват;
– можливість прямого замовлення від виробників.

Недоліки:

– тривалий час доставки;
– проблеми з якістю товарів;
– відсутність належної підтримки клієнтів.

Порівняльний *SWOT*-аналіз конкурентів наведено в табл. 1.1.

Таблиця 1.1 – SWOT-аналіз конкурентів

Конкурент	Сильні сторони	Слабкі сторони	Загрози	Загрози
Amazon	Великий вибір товарів, швидка доставка, довіра споживачів	Висока конкуренція, високі комісії	Розширення асортименту, інновації в доставці	Конкуренція з іншими гігантами, регулювання ринку
eBay	Можливість продажу вживаних товарів, аукціонна модель	Менш популярний серед молоді, ризики шахрайства	Залучення нових продавців, покращення безпеки	Зростання нових платформ, зниження активності користувачів
Allegro	Лідер ринку в Польщі, низькі комісії	Обмежений міжнародний охват, менший асортимент	Розширення на інші ринки, локалізовані послуги	Конкуренція з глобальними гравцями, економічні зміни
Etsy	Підтримка творчих підприємців, нішевий ринок	Обмежений асортимент, висока комісія	Залучення нових креативних продавців, партнерства	Зростання аналогічних платформ, зміни в уподобаннях споживачів
AliExpress	Низькі ціни, широкий асортимент	Тривалий час доставки, проблеми з якістю	Покращення логістики, розширення на нові ринки	Конкуренція з локальними ринками, репутаційні ризики

1.3 Розробка функціональних вимог та виявлення варіантів використання створюваного програмного продукту

Онлайн-маркетплейс, який розробляється в рамках цієї роботи, є платформою, яка сприятиме ефективному спілкуванню між продавцями і покупцями. Ця платформа дозволятиме продавцям збільшити кількість потенційних покупців, які можуть взаємодіяти з товаром, а споживачам легко шукати і купувати потрібні їм елементи гардероба. Така торгова платформа може змінити спосіб, яким люди купують і продають одяг, роблячи процес більш зручним і доступним для всіх.

Щоб забезпечити подібну взаємодію між клієнтами даного сервісу, потрібно забезпечити створення функціоналу, який буде не тільки зручним у використанні, а й матиме практичне застосування в онлайн-маркетплейсі одягу. Визначаючи функціональні вимоги до цього проєкту, варто враховувати специфіку ринку одягу та вимоги сучасного користувача до подібних торговельних майданчиків.

З огляду на все те, про що було сказано вище, можна зрозуміти, що є два основних способи взаємодії з цим сервісом: у ролі продавця та у ролі покупця. Це ті ролі користувача, чий функціонал розрахований на широку аудиторію. Тобто кожна людина, що має доступ до цього ресурсу, може зареєструватися і отримати доступ до ресурсу в рамках наданих їй прав, на основі заданих нею при створенні облікового запису.

Але, крім тих ролей, при створенні подібного багатокористувацького проєкту, виникає потреба в адмініструванні платформи. Це дуже важливий аспект для кожного застосунку, що зумовлює потребу створення ролі адміністратора в цьому проєкті. Це ключова роль для забезпечення контролю над платформою онлайн-маркетплейсу, адже вона потребує додаткового контролю з боку спеціального користувача, незалежно від того, які і на скільки надійні інструменти були задіяні в розробці даного програмного продукту.

Проаналізувавши всі необхідні ролі користувачів та їхнє значення в цьому проєкті, можна з упевненістю сказати, що при розробці онлайн-маркетплейсу, є обов'язковою наявність таких ролей:

1) *Адміністратор* – у контексті програми «ADMIN»;

2) *Продавець* – у контексті програми «SELLER»;

3) *Покупець* – у контексті програми «USER».

Досвід використання програмного продукту різнитиметься залежно від того, яку роль присвоєно автентифікованому й авторизованому користувачеві.

Ґрунтуючись на цих варіантах використання програмного продукту, можна визначити вимоги до функціоналу застосунку. Залежно від ролі, надаватиметься функціонал, який відповідає потребам цього користувача. А ті можливості, яких потребують користувачі кожної з груп, будуть визначені далі. Варто враховувати той факт, що користувачами створюваного сервісу можуть бути люди, які не зареєстровані або ж не авторизовані. Але цей факт не має перешкоджати ознайомленню з ресурсом.

Розглядати функціональні вимоги для застосунку варто, розмежовуючи їх між собою за допомогою прав доступу до ресурсів. Їх же, своєю чергою, варто розглянути в ієрархічному порядку. Тобто від користувача з меншою кількістю прав і функціоналу до більших.

1.3.1 Функціональні вимоги для неавторизованих користувачів

Функціональні вимоги для неавторизованих користувачів, можуть бути такими:

– *перегляд основного контенту:* неавторизований користувач повинен мати можливість переглядати основний контент платформи: головну сторінку, каталог товарів або послуг, статті та інші інформаційні матеріали;

– *пошук і фільтрація:* користувач може використовувати функцію пошуку і фільтрації для швидкого пошуку інформації, що його цікавить, наприклад, товарів, послуг або статей;

– *перегляд інформації про товари:* користувач може переглядати детальну інформацію про продукти або послуги, включно з описами, зображеннями, цінами та іншими характеристиками;

– *контактна інформація та зворотний зв'язок:* користувач може отримати доступ до контактної інформації та форми зворотного зв'язку для постановки запитань, надсилання коментарів або запитів.

Ці функціональні вимоги забезпечують базовий набір можливостей для неавторизованих користувачів, надаючи їм змогу отримувати дні та взаємодіяти з платформою до того, як вони авторизуються.

1.3.2 Функціональні вимоги для користувачів у ролі «Покупець»

Функціональні вимоги для користувачів у ролі «Покупець» можуть бути такими:

– *перегляд товарів або послуг:* покупці повинні мати можливість переглядати каталог товарів або послуг, подані на платформі;

– *пошук і фільтрація товарів:* покупцям треба надати можливість використовувати функцію пошуку та фільтрації для швидкого знаходження потрібних товарів або послуг;

– *перегляд детальної інформації про товари або послуги*: покупці повинні мати можливість переглядати детальну інформацію про товари або послуги, включно з описами, зображеннями, цінами та характеристиками;

– *додавання товарів до кошика:* покупці повинні мати можливість додавати вибрані товари в кошик для подальшого оформлення замовлення;

– *оформлення замовлення:* покупці повинні мати можливість оформляти замовлення, вказуючи необхідну інформацію, як-от адреса доставки, спосіб оплати та інше;

– *перегляд історії замовлень: п*окупці повинні мати доступ до історії своїх попередніх замовлень для відстеження статусу і повторного замовлення за потреби;

– *залишення відгуків та оцінок:* покупці повинні мати можливість

залишати відгуки та оцінки про куплені товари або послуги для зворотного зв'язку і допомоги іншим покупцям;

– *управління обліковим записом:* покупці повинні мати можливість змінювати свої персональні дані, налаштування доставки та інші параметри свого облікового запису;

– *підписка на новини та акції:* покупці можуть мати можливість підписуватися на розсилку новин або отримувати інформацію про спеціальні пропозиції та акції.

Ці функціональні вимоги забезпечують повний набір можливостей для користувачів у ролі «Покупець», даючи їм змогу зручно переглядати та купувати товари або послуги на платформі.

1.3.3 Функціональні вимоги для користувачів у ролі «Продавець»

Функціональні вимоги для продавців можуть бути такими:

– *керування каталогом товарів або послуг:* продавці повинні мати можливість додавати, редагувати і видаляти товари або послуги зі свого каталогу;

– *перегляд та обробка замовлень:* продавці повинні мати доступ до списку замовлень, що надійшли на їхні товари або послуги, і можливість обробляти їх (підтвердження, скасування, оновлення статусу тощо);

– *керування описами та зображеннями товарів:* продавці повинні мати можливість редагувати описи, зображення та інші характеристики своїх товарів або послуг;

– *керування цінами та знижками:* продавці повинні мати можливість встановлювати ціни на свої товари або послуги, а також надавати знижки та акційні пропозиції;

– *керування складськими запасами:* продавці повинні мати можливість відстежувати рівень запасів своїх товарів або послуг і отримувати повідомлення про потребу поповнення;

– *обробка повернень і скарг:* продавці повинні мати можливість обробляти запити на повернення товарів або послуг, а також вирішувати скарги з боку

покупців;

– *керування обліковим записом:* продавці повинні мати можливість керувати своїм обліковим записом, зокрема змінювати персональні дані, налаштування доставки та інше;

– *аналітика продажів:* продавці повинні мати доступ до статистики продажів своїх товарів або послуг, а також можливість аналізувати дані для прийняття управлінських рішень;

– *керування комунікацією з покупцями:* продавці повинні мати можливість зв'язуватися з покупцями для уточнення деталей замовлень, вирішення проблем або надання підтримки.

Ці функціональні вимоги забезпечать продавцям усі необхідні інструменти для ефективного управління своїм бізнесом на платформі, включно з *керуванням* товарами, замовленнями, комунікацією з клієнтами та аналітикою продажів.

1.3.4 Функціональні вимоги для користувача у ролі «Адміністратор»

Функціональні вимоги для адміністратора можуть бути такими:

– *управління користувачами:* адміністратор має змогу переглядати список усіх користувачів платформи, редагувати їхні дані, а також за потреби блокувати чи видаляти облікові записи;

– *моніторинг та модерація контенту:* адміністратор має можливість перевіряти контент, який додають користувачі (товари, описи, зображення), та модерувати його, усуваючи неприйнятний контент, наприклад, контрафактні товари або матеріали, що порушують авторське право чи то інші вимоги законодавства;

– *управління категоріями та каталогом:* адміністратор має повноваження керувати категоріями товарів чи послуг, додаючи нові, редагуючи наявні та видаляючи застарілі;

– *управління замовленнями:* адміністратор має доступ до переліку всіх замовлень на платформі, може переглядати їхні статуси, вирішувати конфліктні ситуації між продавцями та покупцями, а також надавати необхідну підтримку;

– *налаштування параметрів платформи:* адміністратор має можливість налаштовувати параметри платформи, включаючи налаштування безпеки, оплати, доставки, текстів повідомлень тощо;

– *управління системою безпеки:* адміністратор має можливість управління системою безпеки платформи, встановлення прав доступу користувачів, оновлення програмного забезпечення та моніторингу за підозрілою активністю;

– *генерація звітів та аналітика:* адміністратор має доступ до статистики роботи платформи, включаючи дані про активність користувачів, обсяги продажів, популярність товарів та інші показники, а також можливість генерувати звіти для аналізу.

Ці вимоги надають адміністратору необхідні інструменти для ефективного управління та підтримки роботи онлайн-маркетплейсу, включаючи управління користувачами, контентом, замовленнями та системою безпеки.

1.4 Вибір засобів розробки

Онлайн-маркетплейс одягу – це складна і велика платформа для торгівлі, яка потребує потужних засобів у розробці. Пов'язано це насамперед з його головною особливістю. Ця платформа торгівлі повинна підтримувати різні типи учасників, а архітектура системи має бути розроблена так, щоб кожен тип користувачів мав відповідний доступ та функціонал.

Забезпечення безпеки та конфіденційності даних користувачів є одним з найважливіших аспектів розвитку маркетплейсу, оскільки передбачає захист від злому, зловживань та несанкціонованого доступу до персональних даних користувачів.

Крім того, враховуючи потенційно велику кількість користувачів і продуктів, маркетплейс має бути масштабованим і здатним ефективно працювати при високих навантаженнях. Важливо мати систему рейтингів та відгуків, щоб підвищити довіру до якості продавців та товарів.

Вибір засобів для розробки відбувався з урахуванням цих нюансів роботи платформи онлайн-маркетплейсу одягу.

1.4.1 Засоби розробки backend частини

Під час вибору засобів розробки backend частини застосунку онлайн-маркетплейсу одягу головним фактором було забезпечення безпеки, маштабованості та надійності. Визначивши критерії для цих засобів, було прийнято рішення використовувати популярний і перевірений часом фреймворк. Під час аналізу наявних фреймворків за переліченими вище критеріями, вибір припав на фреймворк Spring. Це сучасний фреймворк, що дає змогу створювати Java-застосунки, зокрема й backend для них.

Починаючи з 2002 року, коли Spring Framework був створений Родом Джонсоном як частина його книжки "Експертне індивідуальне проєктування і розробка J2EE", опублікованої 2003 року. У книзі досліджувалася ідея впровадження залежностей (DI) та інверсія контролю (IoC) з використанням легкого та спрощеного контейнера, що пізніше стане Spring [9]. І нині фреймворк активно розвивається та залишається одним з найпопулярніших загалом, і найпопулярнішим серед Java-розробників зокрема.

Spring поєднує різноманітні фреймворки та підпроєкти, які охоплюють безліч аспектів розробки програмного забезпечення, а сам6: розробка вебзастосунків, робота з БД, побудова мікросервісів та інше.

Під час детального вивчення особливостей у розробці такого великого проєкту як онлайн-маркетплейс було ухвалено рішення використовувати мікросервісну архітектуру. Ця методологія розробки розбиває застосунок на невеликі частини, які відповідають за конкретний функціонал застосунку і працюють незалежно один від одного. Взаємодія між мікросервісами відбувається через API.

З огляду на вибрану методологію розробки застосунку було прийнято рішення використовувати Spring Cloud для управління мікросервісами. Цей фреймворк дозволяє виконувати необхідні дії для розгортання та налаштування мікросервісів (рис. 1.2) [10].

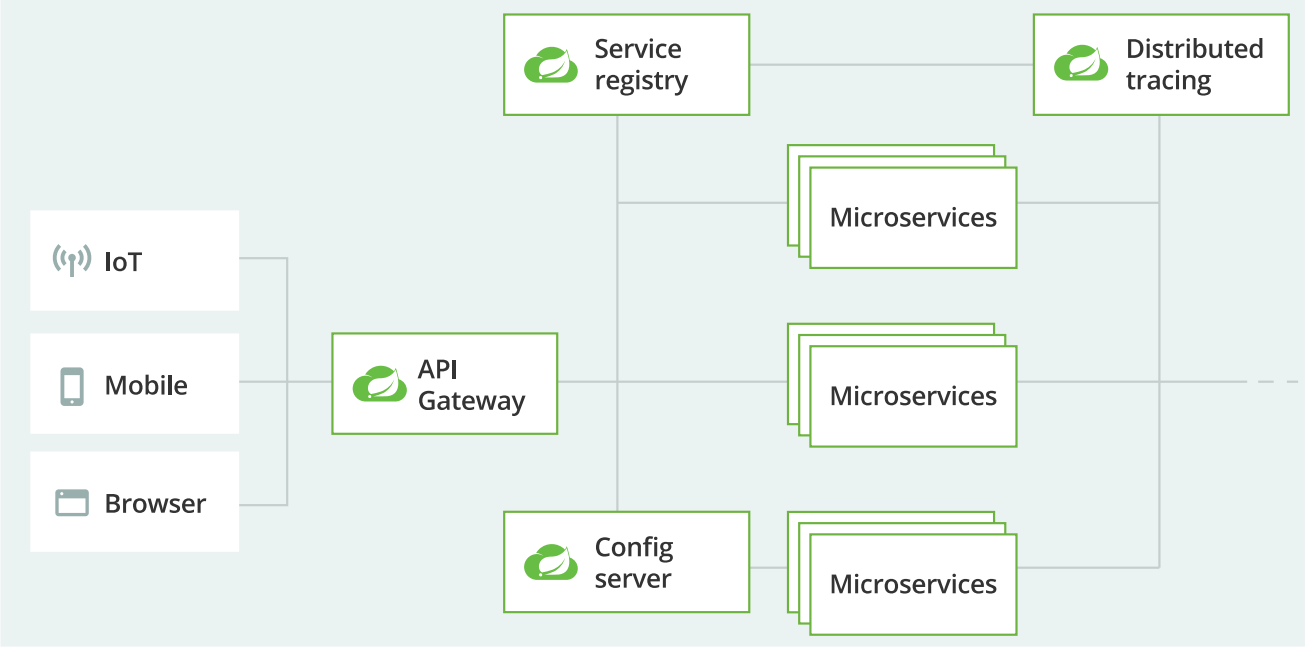


Рисунок 1.2 – Основні аспекти архітектури Spring Cloud

На рис. 1.2 показано схему того, який вигляд може мати мікросервісний застосунок. Ця схема поєднує:

1) різні види інтернет-пристроїв, які можуть взаємодіяти з мікросервісним застосунком;

2) сервіс шлюзу запитів;

3) сервіс виявлення мікросервісів;

4) сервіс управління конфігурацією мікросервісів;

5) сервіс відстеження операцій запитів;

6) мікросервіси.

Це лише приблизна схема мікросервісного застосунку. Наш застосунок матиме такі модулі:

1) Spring Cloud Service Discovery with Netflix Eureka – сервіс реєстрації мікросервісів;

2) Spring Cloud Gateway – шлюз запитів;

3) Keycloak – сервіс авторизації та аунтетифікації;

4) Spring Boot – мікросервіси.

Така схема мікросервісного застосунку дає змогу створити безпечний застосунок, що може маштабуватися.

1.4.2 Засоби розробки frontend частини

Завдяки використанню мікросервісної архітектури, frontend частину створюваного застосунку можна винести в окремий мікросервіс, який, у свою чергу, спілкуватиметься з іншими сервісами через API. Було прийнято рішення використати Spring Boot для створення мікросервісу з frontend частиною застосунку онлайн-маркетплейсу.

Завдяки Spring Boot, можна створити вебзастосунок, який повертатиме HTML сторінки, керуватиме статичним контентом і взаємодіятиме з іншими мікросервісами. Тобто використовуватимуться технології:

- шаблонізатор Thymeleaf;
- мова розмітки Html;
- мова стилів CSS;
- мова програмування JavaScript;
- фреймворк Bootstrap.

Отже, за допомогою Spring Boot буде реалізовано frontend частину мікросервісного застосунку онлайн-маркетплейсу одягу.

1.4.3 Засоби розробки бази даних застосунку

PostgreSQL

Основний компонент, який визначає надійність і ефективність застосунку, – це засоби розробки бази даних. PostgreSQL було вибрано через те, що він має низку добре відомих характеристик, які можуть забезпечити деякі переваги для вашого проєкту.

Популярність PostgreSQL зумовлена його стабільністю і надійністю. При цьому він має високий ступінь відмовостійкості, а також широку систему контролю цілісності даних, що робить його привабливим рішенням для використання в критичних застосунках.

PostgreSQL є одним з найпопулярніших і часто використовуваних інструментів для обробки великих обсягів даних. Він має можливість легко масштабуватися, збільшуючись зі зростанням програми.

Архітектура PostgreSQL дотримується моделі клієнт-сервер. Його основна програма працює як служба, що відповідає за визначення структур даних, зберігання даних і обробку запитів (рис. 1.3) [11].

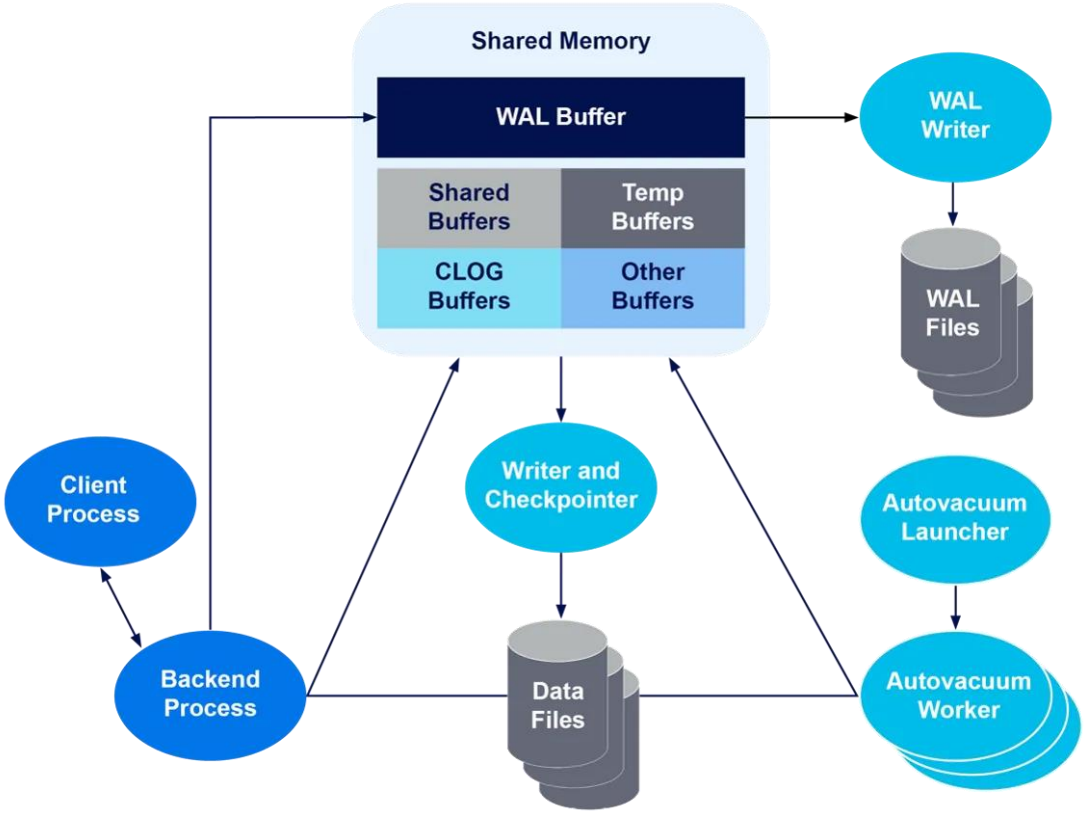


Рисунок 1.3 – Схема PostgreSQL

PostgreSQL має широкий спектр вбудованих функцій та типів даних, що робить його дуже гнучким для різних вимог застосунку.

Все це обґрунтовує використання саме PostgreSQL для розробки платформи онлайн-маркетплейсу одягу.

Spring Data JPA

Spring Data JPA – це популярна платформа для доступу до даних, яка допомагає писати логіку доступу/збереження даних малою кількістю коду. Spring Data JPA – це рівень абстракції над JPA, який зменшує розмір коду, і може використовуватися для ефективного написання логіки з використанням репозиторіїв на основі JPA (рис. 1.4) [12].

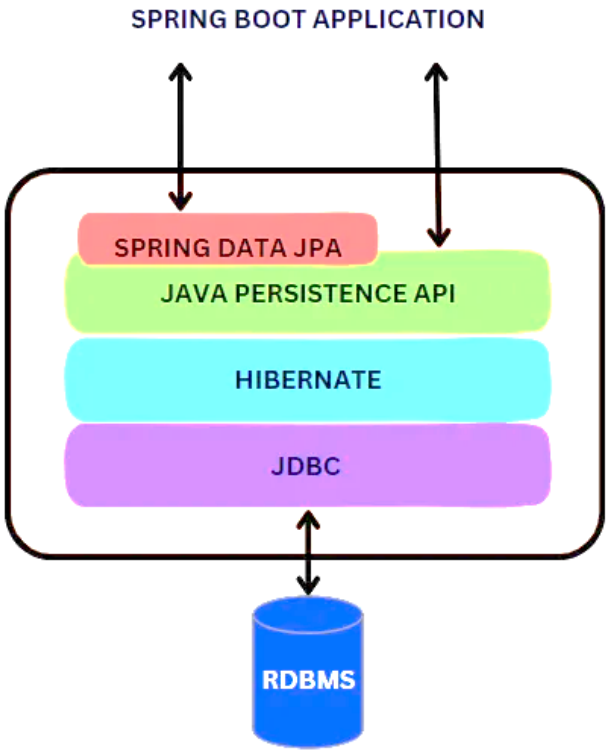


Рисунок 1.4 – Схема роботи Spring Data JPA

Рішення використовувати Spring Data JPA для розробки бази даних у поєднанні з PostgreSQL було ухвалено з декількох причин:

1. *спрощена розробка*: Spring Data JPA надає високорівневий спосіб взаємодії з базою даних, що спрощує розробку та підтримку коду. Він дозволяє розробникам працювати на рівні об'єктів замість SQL-запитів, що полегшує роботу з даними;

2. *універсальність*: підтримує різні типи баз даних, включаючи PostgreSQL, тому застосунок може за потреби легко переноситися на інші бази даних у майбутньому;

3. *інтеграція з іншими модулями Spring*: Spring Data JPA легко інтегрується з іншими модулями Spring (Spring Boot, Spring Security тощо), що дозволяє створювати повноцінні застосунки на основі Spring.

Використання Spring Data JPA разом з PostgreSQL може спростити та прискорити розробку застосунку, забезпечуючи при цьому ефективну роботу з базою даних та високий рівень функціональності.

1.5 Маркетингові інструменти в створенні та просуванні програмного продукту

Впровадження онлайн-маркетплейсу одягу вимагає комплексного підходу до маркетингу, щоб забезпечити успішний запуск та стабільний розвиток бізнесу. Розглянемо ключові маркетингові інструменти, які використовуються в процесі створення та просування програмного продукту.

1.5.1 Карта клієнта

Ключовою особливістю онлайн-маркеплейсу є наявність двох окремих груп користувачів з різними потребами, цілями та очікуваннями. Пов'язано це з тим, що він, за своєю суттю, є платформою, яка об'єднує продавців і покупців, забезпечуючи можливість продажу і купівлі товарів [13]. У нашому випадку – одягу.

У зв'язку з цією особливістю нашого проєкту, буде два види карток клієнта, відповідно до тих груп користувачів, які визначили вище. Тобто картки користувачів для онлайн-маркетплейсу одягу виглядають так:

1. Карта клієнта-продавця:

1) Демографічні характеристики:
   - тип бізнесу (фізична особа, компанія);
   - галузь;
   - географія діяльності.

2) Психографія:
   - цілі (збільшення продажів, розширення аудиторії, збільшення видимості бренду);
   - досвід в онлайн-продажах;
   - уподобання за стилем і типом подання товарів

3) Поведінкові аспекти:
   - платформи, що використовуються для продажу;
   - досвід роботи з онлайн-маркетплейсами;

– потреба в додаткових послугах.

4) Проблеми та потреби:

– труднощі із залученням клієнтів;

– необхідність управління замовленнями.

2. Карта клієнта-покупця:

1) Демографічні характеристики:

– стать;

– вік;

– географія;

– сімейний стан.

2) Психографія:

– стиль життя;

– інтереси;

– уподобання щодо брендів і стилю.

3) Поведінкові аспекти:

– частота покупок онлайн;

– досвід роботи з подібними маркетплейсами.

4) Проблеми та потреби:

– необхідність у широкому асортименті товарів;

– бажання отримати якісний сервіс підтримки.

Отже, розробка двох карт клієнта в діловому стилі дає змогу глибше зрозуміти та задовольнити потреби як продавців, так і покупців, що є ключовим елементом успішного функціонування та розвитку онлайн-маркетплейсу.

1.5.2 Аналіз ринку

Конкуренція на ринку одягу в онлайн-маркетплейсах досить висока. Продавці часто конкурують між собою не тільки за якістю товарів, а й за цінами, що створює вигідне середовище для споживачів.

Завдяки відкритій системі відгуків та оцінок в онлайн-ресурсі кожен продавець прагне отримати хороший рейтинг на платформі. Пов'язано це з підвищенням довіри покупців у разі підтримання високої планки рейтингу.

З огляду на факти, наведені вище, можна зрозуміти, що ринок одягу на онлайн-маркетплейсах є конкурентним, що, своєю чергою, безпосередньо впливає на його розвиток у рамках платформи, завдяки функціоналу та можливостям, які вона надає.

1.5.3 Аналіз споживачів

На сьогодні покоління молодих людей до 35 років становить значну частину користувачів онлайн-маркетплейсів одягу. Це пов'язано з їхньою активною онлайн-присутністю, високою технологічною грамотністю і частішим інтересом до сучасної моди і трендів.

Попри це, з кожним роком середній вік користувачів онлайн-платформ торгових майданчиків одягу в мережі інтернет збільшується. Пов'язано це з тим, що кожна платформа онлайн торгівлі намагається залучати все більше аудиторії, а також з популяризацією онлайн шопінгу.

Отже, на сьогодні сервісами онлайн-покупок більшою мірою користується молоде покоління. Але при цьому збільшується зацікавленість і старшого покоління.

1.5.4 SMM (соціальні медіа)

Соціальний медіа маркетинг є ключовим інструментом для просування онлайн-маркетплейсу одягу в соціальних мережах [14]. Цей вид маркетингу створює спільноту навколо бренду, привертає увагу потенційних клієнтів, спілкується з ними та підтримує взаємодію.

Активна присутність у соціальних мережах необхідна для ефективного просування онлайн-ринку одягу. Соціальні мережі надають можливість взаємодіяти з цільовою аудиторією, просувати свої товари, проводити рекламні кампанії та підтримувати спільноту. SMM дозволяє будувати довгострокові відносини з клієнтами та підвищувати впізнаваність бренду.

1.6 Висновки до розділу 1

У розділі проведено аналіз особливостей розробки онлайн-маркетплейсу продажу одягу. З’ясовано, що він має свою специфіку, яку потрібно враховувати при його розробці. Необхідними складовими при створенні цього програмного продукту є: зручний інтерфейс користувача; масштабованість; захист даних користувачів і транзакцій; збір і аналіз даних про поведінку користувачів для поліпшення послуг і продуктів; підтримка клієнтів тощо.

Проведений SWOT-аналіз наявних застосунків із подібним функціоналом допоміг виявити популярні тенденції, відмітити переваги та недоліки серед наявних на ринку програмних продуктів аналогів Amazon, eBay, Allegro, Etsy та AliExpress. Тим самим аналіз аналогів допоміг визначити функціональні вимоги створюваного застосунку. Саме на основі повного переліку функціоналу та варіантів взаємодії системи з користувачем відбуватиметься подальша розробка застосунку. Сформульований перелік вимог визначив траєкторію процесу створення застосунку.

Вибір засобів розробки застосунку стосувався засобів розробки бази даних, а також розробки backend та frontend частин вебзастосунку. Так, для розробки бази даних обґрунтовано вибір Spring Data JPA разом з PostgreSQL, які можуть прискорити розробку застосунку, забезпечуючи при цьому ефективну роботу з базою даних та високий рівень функціональності. Для розробки backend частини вибрано фреймворк Spring, який забезпечує кросплатформність, надійний рівень безпеки, оптимізацію продуктивності, швидкість роботи, має великий перелік інструментів та функцій для ефективного створення програмного застосунку за короткотривалий період. Для розробки frontend частини онлайн-маркетплейсу вибрано мікросервіс Spring Boot.

Крім того, у розділі розглянуто можливі маркетингові інструменти в створенні та просуванні створюваного програмного продукту.

2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ПРОДУКТУ

2.1 Інформаційна модель предметної області

Інформаційна модель – це концептуальна структура, яка відображає організацію даних і взаємозв'язки між ними в конкретній предметній області або системі. Вона служить як інструмент для опису та аналізу інформації, що обробляється або зберігається у системі, і допомагає у розумінні структури даних та їх функціонування.

Інформаційна модель визначає сутності (або об'єкти), їх атрибути (властивості або характеристики) і взаємозв'язки між цими сутностями. Вона може бути використана як основа для розробки бази даних, проєктування програмного забезпечення або для спрощення розуміння та комунікації між учасниками проєкту.

2.1.1 Сутності моделі

1) Користувач (User):

- userId: унікальний ідентифікатор користувача.
- username: унікальне ім'я користувача.
- userFirstName: ім'я користувача.
- userLastName: прізвище користувача.
- email: електронна пошта користувача.
- userBirthday: дата народження користувача.
- password: пароль користувача.

2) Товар (Product):

- productId: унікальний ідентифікатор товару.
- productName: назва товару.
- productDescription: опис товару.

3) Розмір товару (ProductSize):

- productSizeId: унікальний ідентифікатор розміру товару.
- productSizeName: назва розміру товару.

– productSizeQuantity: кількість товару з цим розміром.

4) Відгук про товар (ProductReview):

– productReviewId: унікальний ідентифікатор відгуку.

– reviewText: текст відгуку.

– rating: оцінка товару.

5) Кошик (Basket):

– basketId: унікальний ідентифікатор кошика.

– user: користувач, якому належить кошик.

6) Товари кошика (BasketItem):

– basketItemId: унікальний ідентифікатор елемента кошика.

– basket: кошик, до якого належить цей елемент.

– product: товар, який представлений цим елементом.

– quantity: кількість товару, представленого цим елементом кошика.

7) Замовлення (Order):

– orderId: унікальний ідентифікатор замовлення.

– orderStatus: статус замовлення.

– user: користувач, який зробив замовлення.

– basket: кошик, який був перетворений на замовлення.

– product: товар, що був замовлений.

– orderQuantity: кількість товару у замовленні.

8) Відгук про продавця (SellerReview):

– sellerReviewId: унікальний ідентифікатор відгуку.

– reviewText: текст відгуку.

– rating: оцінка продавця.

2.1.2 Стосунки

1) Користувач (User):

– Має один кошик (One-to-One: User-Basket).

– Має багато відгуків про продавця (One-to-Many: User-SellerReview).

– Має багато замовлень (One-to-Many: User-Order).

2) Товар (Product):

– Має багато розмірів (One-to-Many: Product-ProductSize).
– Має багато відгуків (One-to-Many: Product-ProductReview).
– Продавець (User) продає багато товарів (One-to-Many: User-Product).

3) Розмір товару (ProductSize):

– Належить одному товару (Many-to-One: ProductSize-Product).

4) Відгук про товар (ProductReview):

– Належить одному товару (Many-to-One: ProductReview-Product).

5) Кошик (Basket):

– Належить одному користувачеві (One-to-One: Basket-User).
– Має багато елементів кошика (One-to-Many: Basket-BasketItem).

6) Товари кошика (BasketItem):

– Належить одному кошику (Many-to-One: BasketItem-Basket).
– Представляє один товар (Many-to-One: BasketItem-Product).

7) Замовлення (Order):

– Має одного користувача (Many-to-One: Order-User).
– Має один кошик (Many-to-One: Order-Basket).
– Має один товар (Many-to-One: Order-Product).

8) Відгук про продавця (SellerReview):

– Належить одному користувачеві (Many-to-One: SellerReview-User).

2.1.3 Сценарії взаємодії

1) Реєстрація нового користувача:

– Користувач заповнює форму реєстрації, вводячи особисті дані та обираючи ім'я користувача та пароль.
– Система перевіряє правильність введених даних та створює новий обліковий запис користувача.

2) Додавання товару до кошика:

– Користувач переглядає сторінку товару та обирає бажаний товар.
– Користувач вказує бажану кількість товару та додає його до кошика.

– Система оновлює кошик користувача та відображає зміни.

3) Оформлення замовлення:

– Користувач переглядає вміст свого кошика та переходить до оформлення замовлення.
– Користувач вводить адресу доставки та обирає метод оплати.
– Система створює нове замовлення та оновлює статус кошика користувача.

4) Написання відгуку про товар:

– Після отримання замовлення користувач має можливість залишити відгук про куплений товар.
– Користувач вказує оцінку товару та пише текстовий відгук.
– Система зберігає відгук та відображає його на сторінці товару.

5) Перегляд історії замовлень:

– Користувач має можливість переглянути історію своїх минулих замовлень.
– Система відображає список замовлень з деталями кожного з них.

6) Взаємодія з обліковим записом користувача:

– Користувач може змінювати свої особисті дані, змінювати пароль, видаляти обліковий запис тощо.
– Система забезпечує функціонал для управління обліковим записом користувача та здійснення необхідних змін.

В результаті аналізу та розробки інформаційної моделі для онлайн-маркетплейсу досягнуто розуміння структури даних та їх взаємозв'язків у системі. Інформаційна модель стала основою для подальшого проєктування бази даних та розробки програмного забезпечення.

2.2 Архітектура програмного застосунку

Архітектура програмного застосунку – це структура та організація програмного забезпечення, яка визначає його складові частини, взаємозв'язки між

ними та принципи їх взаємодії. Це план або концепція, що описує загальний вигляд програми та її компонентів, включаючи структуру, поведінку, взаємодію та властивості кожного компонента.

У сучасному програмуванні архітектура програмного застосунку відіграє важливу роль у забезпеченні того, щоб програмне забезпечення було легко зрозумілим, модульним, масштабованим та підтримуваним. Вона дозволяє розробникам організовувати свій код так, щоб його було легко розширювати та змінювати в майбутньому.

При розробці застосунку онлайн-маркетплейсу одягу були застосовані архітектури програмного забезпечення MVC та Rest API.

2.2.1 MVC

Це архітектурний шаблон, який допомагає в організації структури та логіки програмного застосунку. Він розбиває програму на три основні компоненти: Модель (Model), Подання (View) і Контролер (Controller) (рис. 2.1) [15].

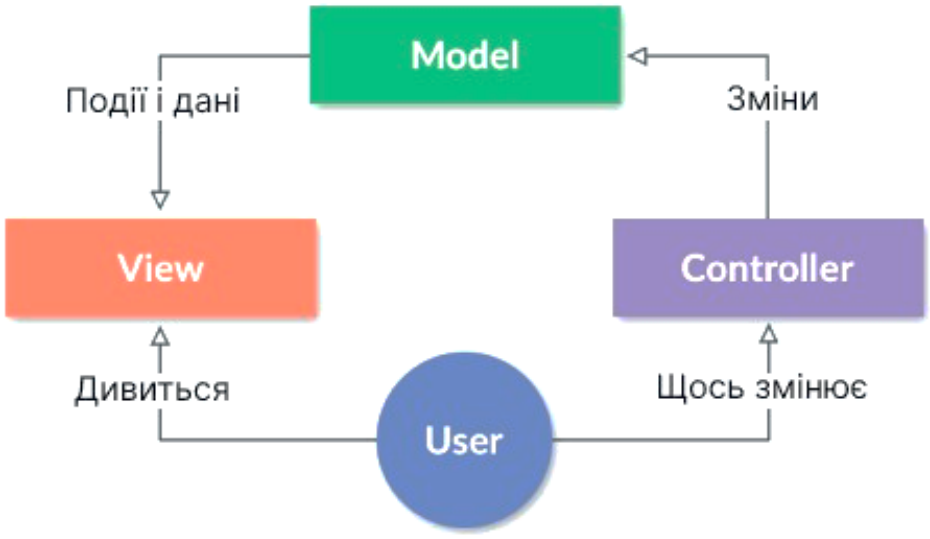


Рисунок 2.1 – Структура MVC

Кожен з цих компонентів виконує свою роль у вебзастосунку. Для розуміння того, за що відповідає кожен з них, варто розглянути їх окремо.

1. *Модель* відповідає за подання даних та бізнес-логіку програми. Також може охоплювати об'єкти даних, логіку операцій над цими даними, валідацію, доступ до бази даних тощо. Головною умовою для моделі є те, що вона не повинна безпосередньо залежати від інтерфейсу користувача або відображення даних.

2. *Подання (Вигляд)* відповідає за подання даних користувачу. Поданням може бути вебсторінка, графічний інтерфейс, екран мобільного застосунку тощо. Також воно отримує дані від моделі і відображає їх для користувача. Подання не має власної бізнес-логіки та повинно бути якомога простішим.

3. *Контролер* відповідає за обробку вхідних запитів користувача та взаємодію з моделлю та поданням. Він приймає вхідні дані від користувача через вебінтерфейс або інших джерел, обробляє їх, ініціює відповідні дії та оновлює стан моделі або подання. Завдяки контролерам вдається відокремити логіку обробки запитів від логіки моделі та відображення, що робить програму більш модульною та підтримуваною [15].

Отже, MVC дозволяє розділити логіку програми на три чітко визначені компоненти, що сприяє покращенню підтримки, розширювання та повторного використання коду. Цей шаблон широко використовується в розробці вебзастосунків, але може бути застосований і в інших типах програмних застосунків.

2.2.2 Rest API

REST API (інші назви RESTful API або RESTful web API) – це інтерфейс прикладного програмування (API), який відповідає принципам архітектурного стилю передачі представницького стану (REST). REST API забезпечує гнучкий, легкий спосіб інтеграції застосунків і підключення компонентів в архітектурі мікросервісів [16].

Розробники можуть розробляти REST API, використовуючи практично будь-яку мову програмування і підтримуючи різні формати даних. Єдина вимога полягає в тому, щоб вони відповідали шести принципам проєктування REST,

відомим як архітектурні обмеження:

1. *Ресурси (Resources):* кожний ресурс повинен мати унікальний ідентифікатор (URI), який ідентифікує його.

2. *HTTP методи:* потрібно використовувати стандартні HTTP методи для виконання операцій над ресурсами, наприклад:

– GET: для отримання даних;

– POST: для створення нового ресурсу;

– PUT або PATCH: для оновлення існуючого ресурсу;

– DELETE: для видалення ресурсу.

3. *Подання (Representation):* клієнт та сервер можуть обмінюватися різними поданнями ресурсів. Найпоширенішими форматами є JSON або XML.

4. *Безстатевість (Statelessness):* кожен запит до сервера повинен містити всю необхідну інформацію для обробки запиту. Сервер не повинен зберігати стан клієнта між запитами. Це сприяє простоті та масштабованості системи.

5. *Гіпермедіа (Hypermedia):* RESTful API може використовувати гіпермедіа, як засіб для навігації та взаємодії з ресурсами, наприклад, використання посилань або вбудованих посилань в поданнях ресурсів.

6. *Кешування (Caching):* використання кешування для підвищення продуктивності та зниження навантаження на сервер. REST дозволяє використовувати стандартні механізми кешування HTTP.

Ці принципи допомагають забезпечити REST API простоту для розуміння, легкість використання та розширювання, а також дотримання стандартів веброзробки.

2.2.3 Архітектура клієнт-сервер

Клієнт-серверна архітектура є системою, яка розміщує, надає та керує більшістю ресурсів і послуг, які запитує клієнт. У цій моделі всі запити та послуги надаються через мережу, її ще називають мережевою обчислювальною моделлю або мережею клієнт-сервер [17]. Ця архітектура є моделлю розподіленої системи, де функціональність програми розділена між двома основними типами

компонентів: клієнтами і серверами. Кожен з цих компонентів виконує свої власні характеристики:

1. Клієнт:

- це програма або пристрій, який взаємодіє з користувачем та відправляє запити на сервер для отримання даних або виконання дій;
- клієнт може бути реалізований у вигляді веббраузера, мобільного застосунку тощо;
- він несе відповідальність за подання інформації користувачу та збирання введених даних.

2. Сервер:

- це програма або пристрій, який обробляє запити від клієнтів та відповідає на них, надаючи необхідні ресурси або послуги;
- сервер може бути фізичним або віртуальним пристроєм, який служить для обробки запитів;
- він несе відповідальність за обробку та зберігання даних, виконання бізнес-логіки та надання відповідей на запити клієнтів.

У сучасному світі клієнт-серверна архітектура широко використовується в різних типах програмних застосунків: від вебзастосунків до мобільних застосунків та хмарних сервісів.

2.2.4 Мікросервісна архітектура

Мікросервісна архітектура – це інноваційна стратегія розробки програмного забезпечення, яка набула великої популярності завдяки своїм властивостям: гнучкості, можливості масштабування та стійкості до відмов. Цей підхід поділяє застосунок на набір малих сервісів, які можуть бути розроблені, розгорнуті та масштабовані незалежно один від одного. Кожен з цих сервісів – це автономний компонент, що виконує специфічну бізнес-функцію і взаємодіє з іншими сервісами через встановлені інтерфейси [18].

Популярність цього метода розробки зростає. Обумовлено це насамперед перевагами цієї архітектури над монолитами. Ось декілька ключових характеристик мікросервісної архітектури, які водночас є його перевагами [18]:

1. *незалежність*: кожен мікросервіс може бути розроблений, випущений та масштабований незалежно від інших. Це означає, що команди можуть працювати над різними сервісами паралельно, без впливу на роботу інших команд;

2. *децентралізація*: у мікросервісній архітектурі немає центрального "моноліту", який керує всіма сервісами. Замість цього кожен сервіс керується окремо, що сприяє гнучкості та швидкості розробки;

3. *спілкування через API*: мікросервіси спілкуються один з одним через визначені API, які дозволяють обмінюватися даними та виконувати функції;

4. *відмовостійкість*: якщо один мікросервіс зазнає збою, це не має впливати на роботу інших сервісів. Це забезпечує високу відмовостійкість системи;

5. *масштабованість*: мікросервіси можна легко масштабувати залежно від потреб, що робить цю архітектуру ідеальною для великих, складних застосунків.

Мікросервісна архітектура може бути використана для побудови великих, складних систем, які легко масштабуються та підтримуються. Однак, вона вимагає значної координації та управління, щоб забезпечити правильну роботу всіх сервісів.

2.3 Проєктування бази даних

Для застосунку, який розробляється у рамках цієї роботи, а точніше онлайн-маркетплейсу одягу, зв'язки між сутностями будуть відповідати зв'язкам між таблицями у базі даних. Тобто до цієї ER-моделі відповідають дані, подані у підрозділі 2.1*"Інформаційна модель предметної області"*, а точніше пп. 2.1.1*"Сутності моделі"* та 2.1.2*"Стосунки"*. Виходячи з цього, ER-модель для застосунку подана на рис. 2.2.

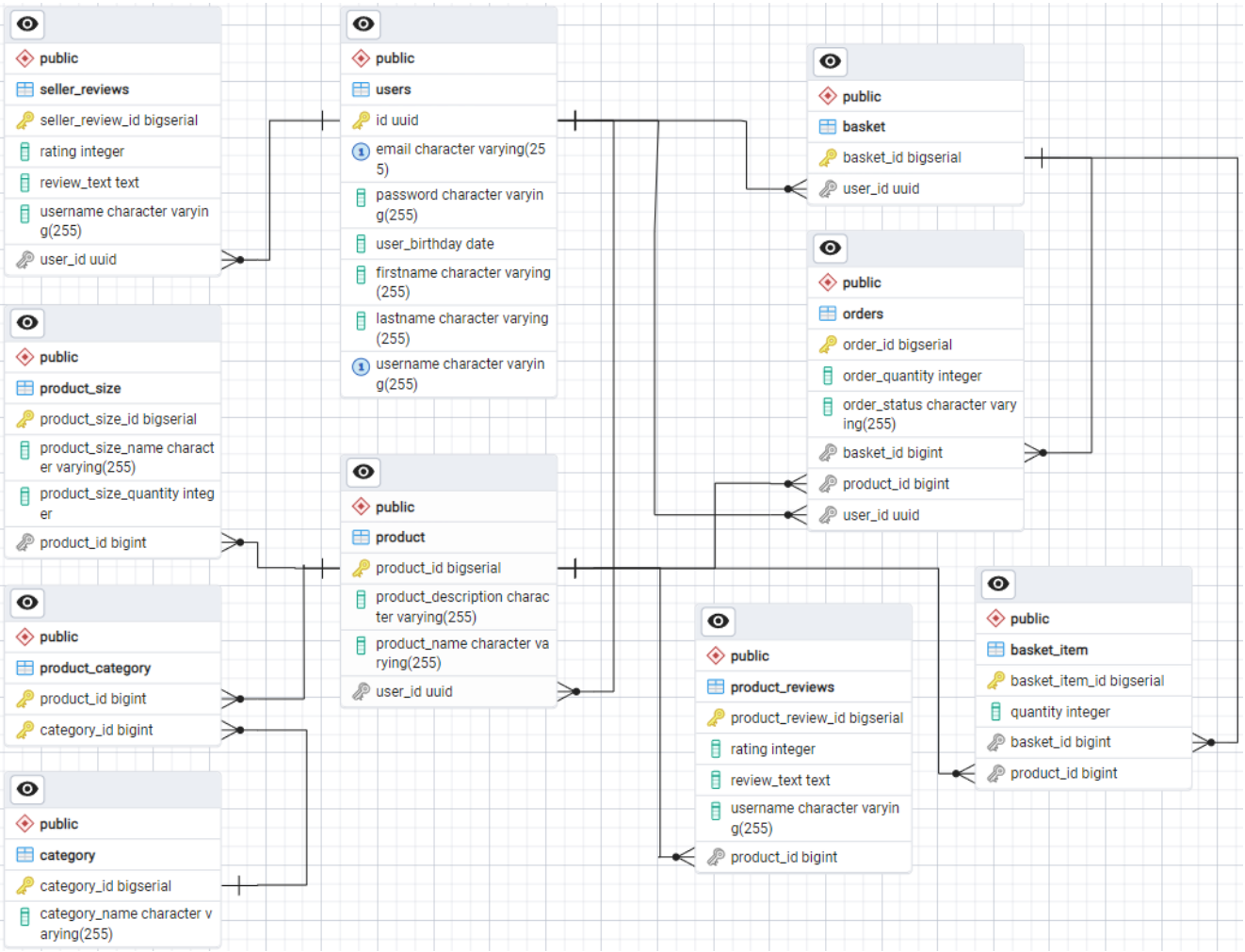


Рисунок 2.2 – ER-діаграма

Усі стовпці у таблицях сформовані коректно та із зауваженнями специфіки області розробки. Також зв’язки, які було сформовано раніше, можна побачити на діаграмі та впевнитись у тому що вона створена коректно.

2.4 Моделювання програмної системи

Сучасні онлайн-майданчики продажу одягу розвиваються значною мірою, завдяки моделюванню програмних систем, яке дає розробникам ефективний інструмент для планування, оцінки та покращення функціональності платформи. Моделювання відіграє вирішальну роль у процесі розробки, коли мова йде про створення онлайн-маркету, оскільки воно дозволяє точно визначити потреби користувачів, бізнес-цілі та технічні характеристики системи.

Онлайн-маркетплейси – це складна програмна платформа, яка з'єднує покупців і продавців для розвитку електронної комерції. При розробці такої системи необхідно враховувати багато факторів, таких як продуктивність, масштабованість, безпека даних та зручність використання. Моделювання програмних систем значно знижує ризики і підвищує якість розробки, допомагаючи оцінити всі ці фактори на ранній стадії проєктування.

У наш час розробники програмного забезпечення використовують UML-діаграми для подання програмних систем. Це пов'язано з тим, що цей інструмент допомагає в аналізі вимог і візуалізації ідей, тим самим є незамінним компонентом моделювання програмних систем. Пропонуючи безліч стилів діаграм, він полегшує організацію та уточнення системних вимог.

Ці діаграми демонструють певні етапи процесу розробки програмного продукту, що дозволяє підвищити якість розробки. Це призводить до покращення комунікації та управління проєктом між командою розробників та зацікавленими сторонами.

UML є важливим інструментом для розробки програмного забезпечення, оскільки дозволяє описувати програмні системи на кожному етапі їх життєвого циклу. При правильному застосуванні UML може забезпечити успішне досягнення цілей проєкту і сприяти систематизованій розробці.

Враховуючи ці нюанси та переваги моделювання програмних систем засобами UML, було прийнято рішення використовувати саме UML-діаграми для проєктування. Для онлайн-маркетплейсу було вибрано для розробки три типи UML-діаграм, а саме:

1) *Use Cases* – діаграма сценаріїв використання системи;

2) *Classes* – діаграма сутностей або об’єктів;

3) *Activity* – діаграма послідовності дій або процесів.

Саме ці діаграми було реалізовано у проєкті онлайн-маркетплейсу одягу.

2.4.1 Діаграми прецендентів

Концепція моделювання діаграм Use Cases дозволяє описати функціональність системи так, як її бачать користувачі сервісу. На цій діаграмі актори мають різні ролі або зовнішні сутності, а сценарії взаємодії користувачів з системою відображаються у варіантах використання. Ці діаграми є важливим інструментом для аналізу системних вимог і специфікацій, оскільки вони допомагають розробникам зрозуміти потреби користувачів і спроєктувати систему так, щоб вона відповідала цим потребам.

У роботі розроблено три Use Cases діаграми для трьох різних ролей користувачів онлайн-маркетплейсу одягу, а саме: Адміністратор, Покупець, Продавець (рис. 2.2-2.4).

1) Адміністратор:

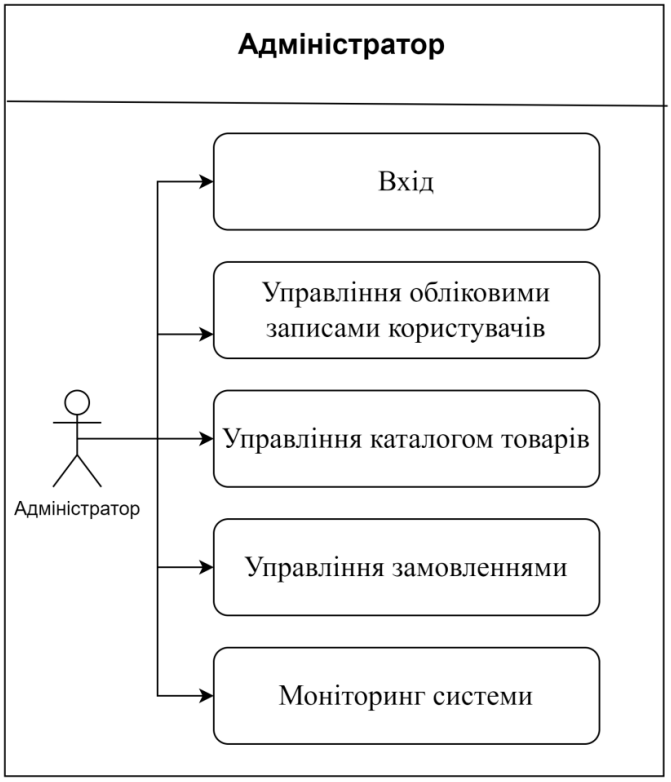


Рисунок 2.2 – Use Cases діаграма для Адміністратора

Ця діаграма показує те, як може користувач може взаємодіяти з програмним продуктом у ролі адміністратора.

Перший сценарій взаємодії з даною роллю є авторизація та автентифікація,

але без можливості зареєструватись. Пов’язано це з міркуваннями безпеки та з відсутністю великої кількості користувачів з такими привілеями.

Другим сценарієм є сценарій “Управління обліковими записами користувачів”. Цей сценарій взаємодії дозволяє проводити певні операції з аккаунтами користувачів так само, як і сценарії: “Управління каталогом товарів” та “Управління замовленнями”. Тобто адміністратор може виконувати з цими даними CRUD операції, якщо це потребується для коректної роботи платформи.

Ще одним сценарієм є “Моніторинг системи”, що дозволяє слідкувати за сервісами, які запущені. Та при виникненні помилки він може бачити, який саме сервіс не працює. Це в свою чергу дозволить виправити ймовірні помилки.

2) Продавець:

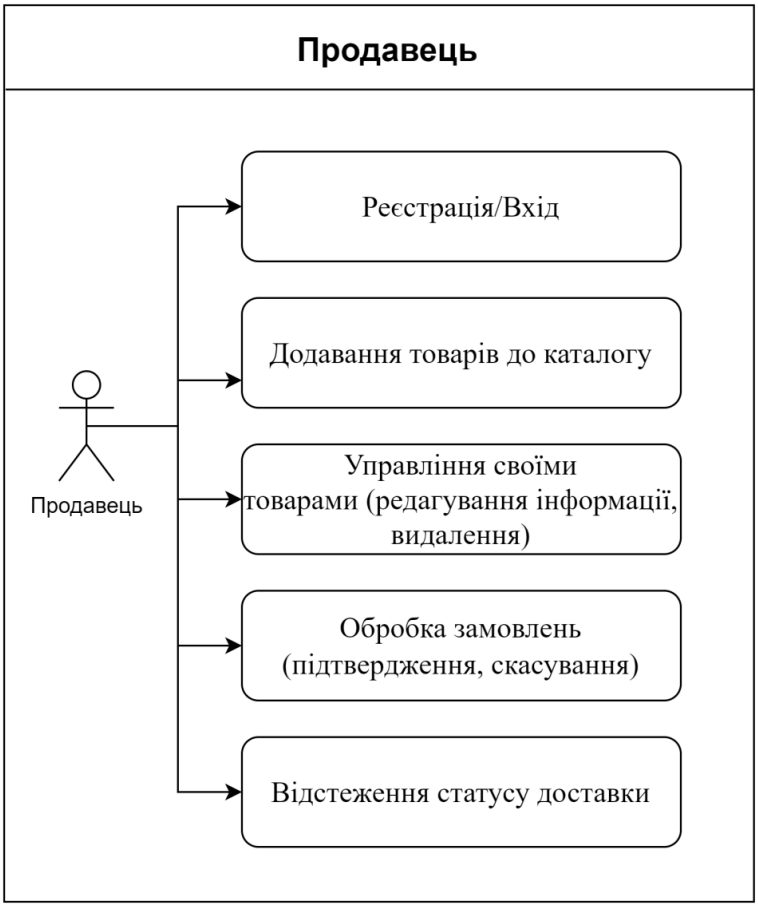


Рисунок 2.3 – Use Cases діаграма для Продавця

Ця діаграма показує, як користувач може взаємодіяти з програмним продуктом у ролі Продавця. Першим етапом взаємодії є процес авторизації та

входу у систему, де користувач має можливість увійти у власний обліковий запис. Наступною функцією є реєстрація нових користувачів, яка вимагає від Продавця заповнити відповідну інформацію та створити обліковий запис.

Після успішної авторизації Продавця, користувач має можливість додавати товари у каталог. Після додавання товарів Продавець може здійснювати редагування, видалення та оновлення інформації про товари.

Після отримання замовлень від покупців Продавець може обробляти їх, виконуючи необхідні дії для прийняття та виконання замовлення. Крім того, Продавець має змогу відстежувати статус доставки замовлень, надаючи корисну інформацію покупцям щодо прогресу доставки їхніх замовлень.

3) Покупець:

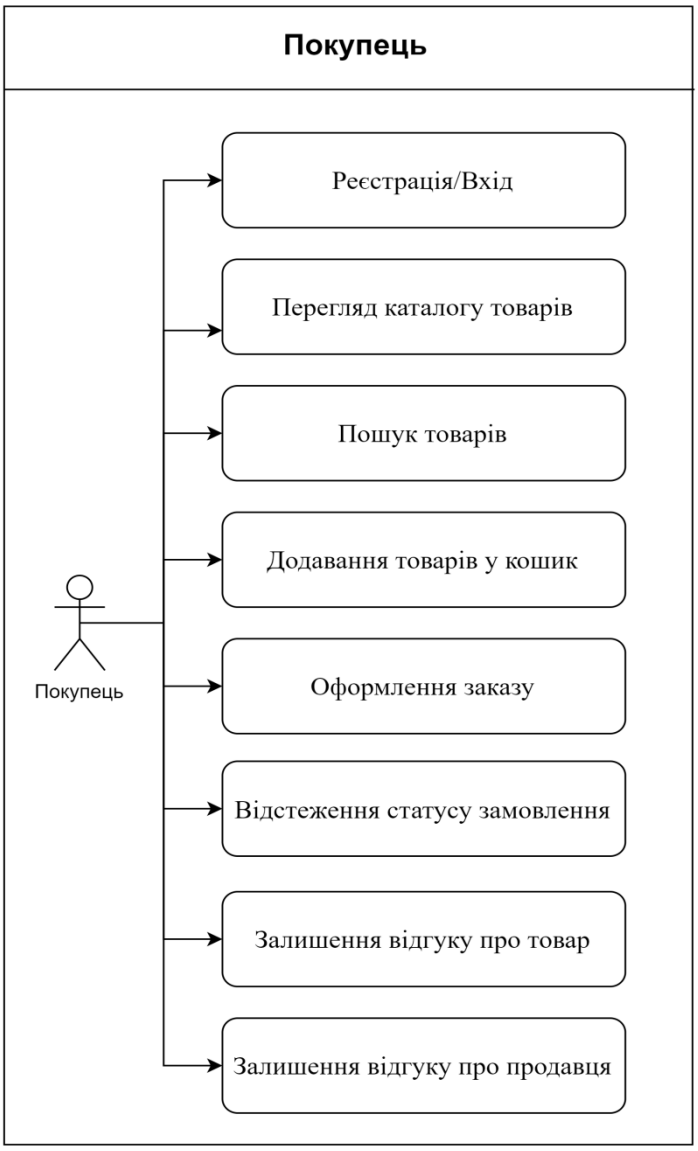


Рисунок 2.3 – Use Cases діаграма для Покупця

Ця діаграма ілюструє взаємодію користувача у ролі Покупця з програмним продуктом. Перший етап – це процес реєстрації або входу в обліковий запис, де користувач має можливість зареєструватися або увійти.

Після входу в систему Покупець має можливість переглядати каталог товарів та ознайомитись з асортиментом та пошуком товару за категорією або іншими критеріями.

Після знаходження потрібного товару Покупець може додати його у кошик для подальшого оформлення замовлення. Після додавання товарів у кошик Покупець може перейти до оформлення замовлення.

Після оформлення замовлення Покупець може відстежувати його статус, перевіряючи інформацію про прогрес доставки. Крім того, після отримання замовлення він може залишити відгук про товар, виражаючи свою думку та досвід використання товару, а також залишити відгук про продавця, оцінюючи його роботу та обслуговування.

2.4.2 Діаграма класів

UML-діаграма класів (Classes) використовується для візуального подання класів у програмній системі. Вона складається з блоків, які показують класи та пов'язані з ними атрибути. Ці відношення ілюструють взаємозв'язки та взаємодію між класами. За допомогою UML діаграми розробники можуть визначити, які класи потрібно створити для досягнення певної функціональності, а також дізнатися більше про складові частини і те, як вони працюють разом всередині програмних систем.

На рис. 2.4 подана розроблена діаграма класів для створюваного програмної платформи онлайн-маркетплейсу для зберігання та обробки її даних. Вона складається з десяти класів, які забезпечують функціонування та взаємодію з різними користувачами.

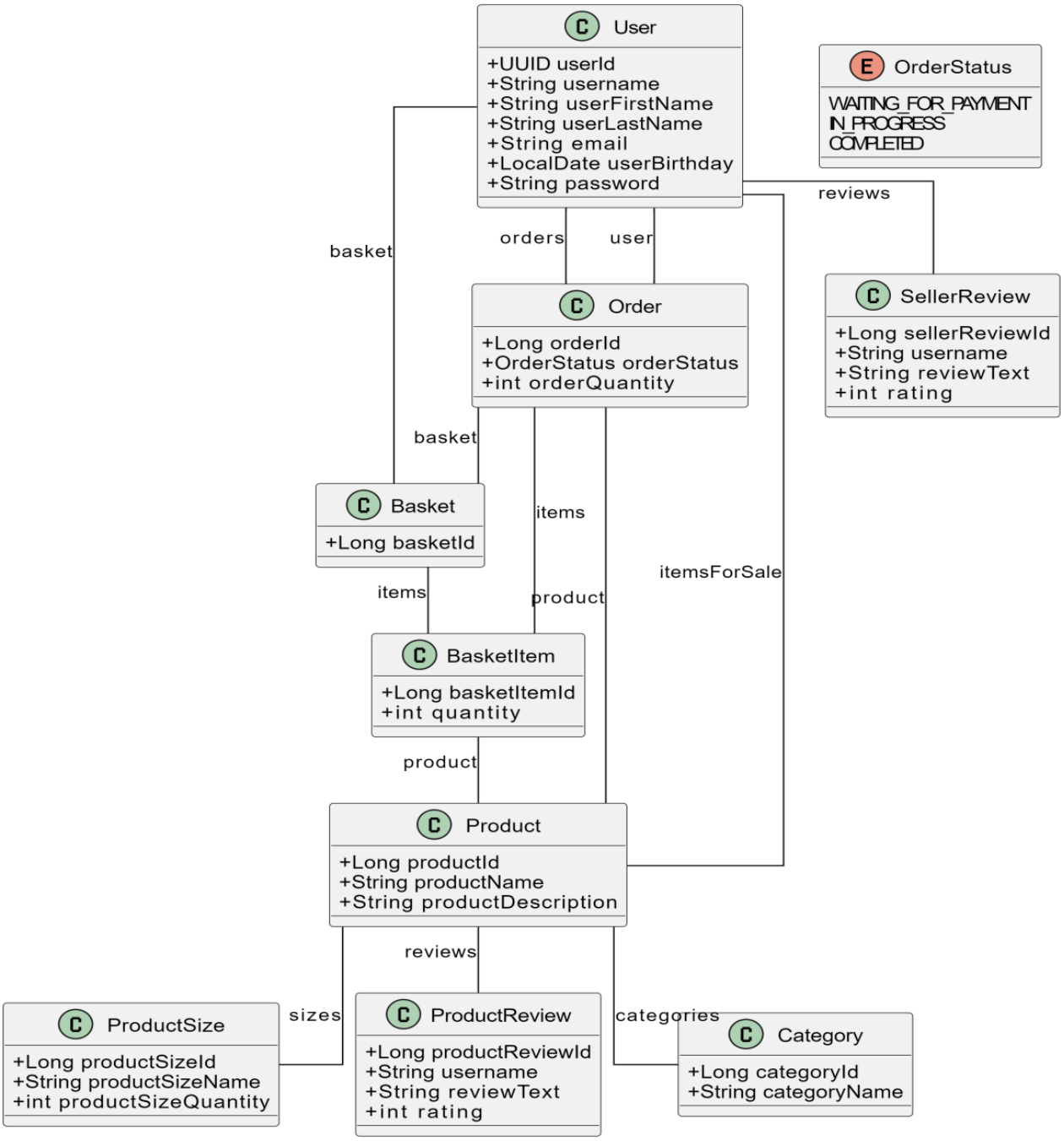


Рисунок 2.4 – Діаграма класів застосунку

2.4.2 Діаграми активності

Діаграма активності (Activity), як інструмент моделювання UML, використовується для зображення послідовності дій та процесів, які є у програмному продукті. Ця діаграма допомагає візуалізувати та проаналізувати процеси, які можуть відбуватися в системі.

Це схоже на ідею моделювання Use Cases, де актори представляють ролі або зовнішні сутності, а сценарії взаємодії з системою відображаються у варіантах використання. Вона дозволяє відобразити послідовність дій або робочих процесів, які відбуваються в системі, і допомагає розробникам зрозуміти, як ці процеси пов’язані один з одним.

Як і у діаграмі Use Case, створюючи цю діаграму клієнтів сервісу розглянуто у декількох ролях, які можуть мати користувачі сервісу. Але в Activity вказано можливі варіанти того, як можуть себе повести користувачі.

У роботі розроблено три діаграми активності для трьох різних ролей користувачів онлайн-маркетплейсу одягу, а саме: Адміністратор, Покупець, Продавець (рис. 2.5-2.7).

1) Activity для Адміністратора:

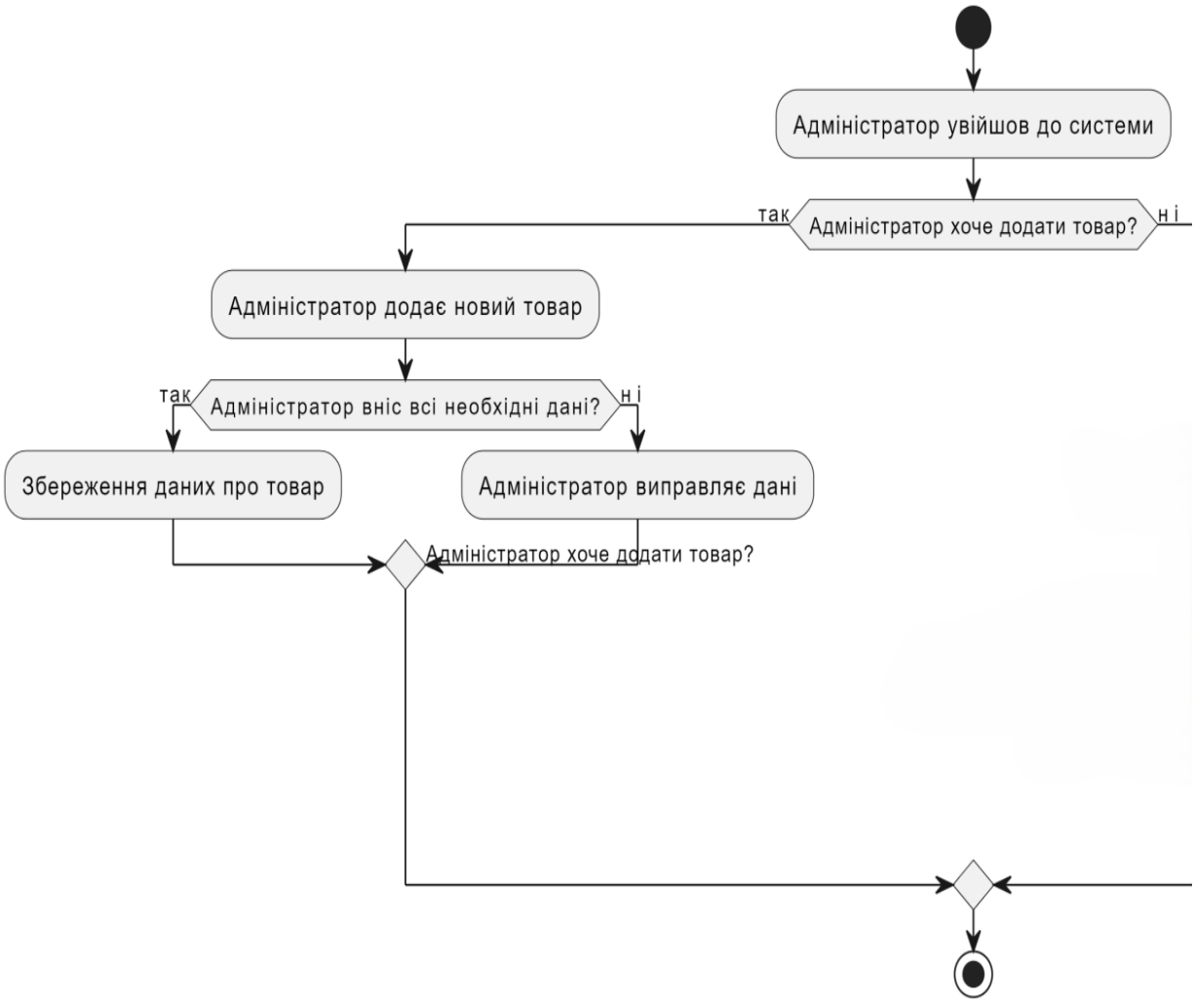


Рисунок 2.5 – UML діаграма активності для Адміністратора ч.1

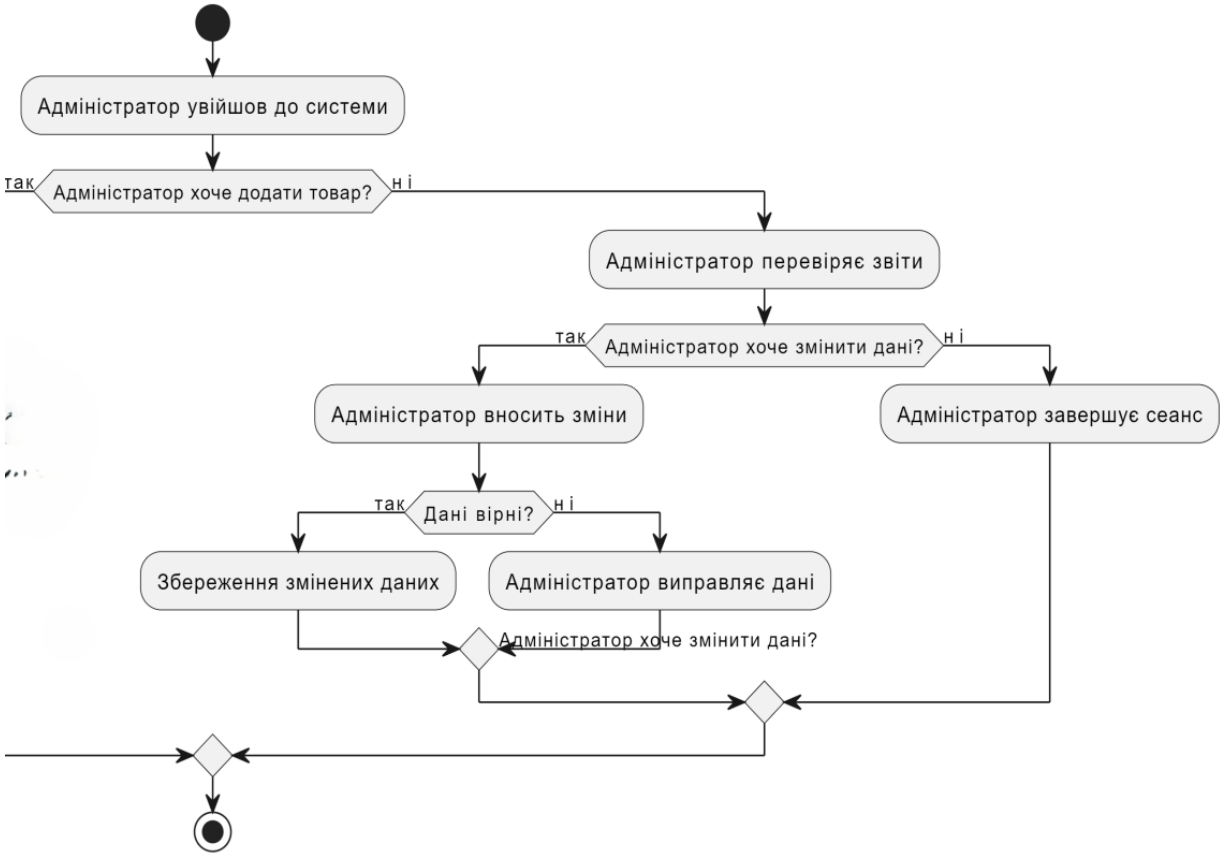


Рисунок 2.6 – UML діаграма Activity для Адміністратора ч.2

2) Activity для Продавця:

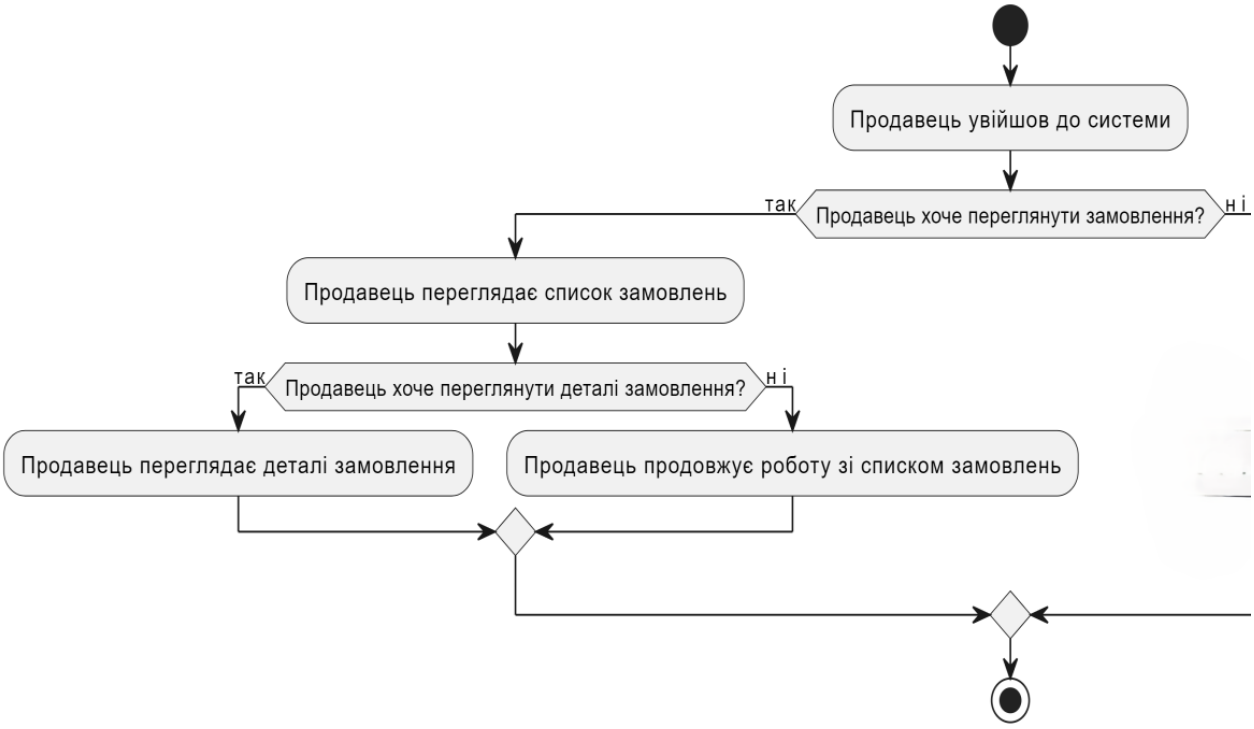


Рисунок 2.7 – UML діаграма Activity для Продавця ч.1.

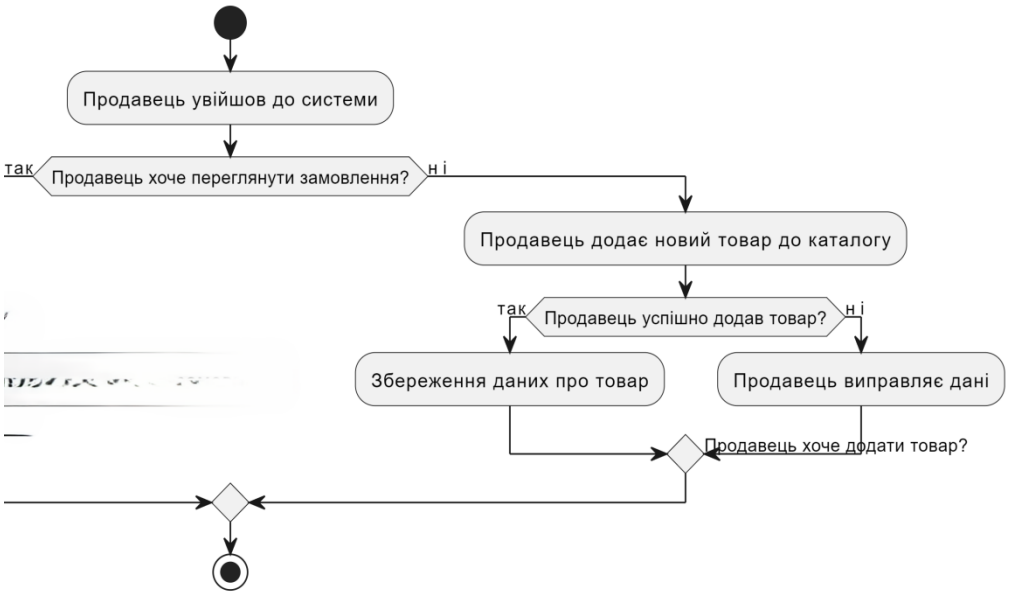


Рисунок 2.8 – UML діаграма Activity для Продавця ч.2

3) Activity для Покупця:

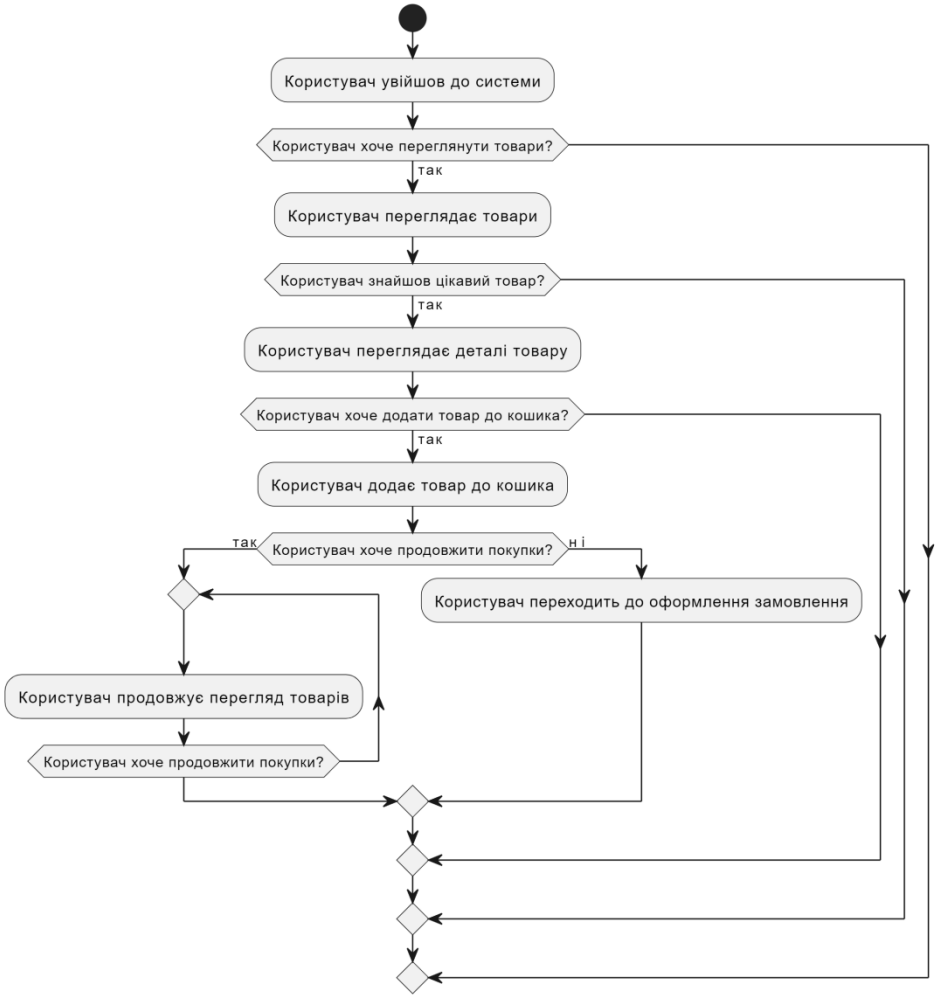


Рисунок 2.9 – UML-діаграма Activity для Покупця

2.5 Висновки до розділу 2

У розділі розглянуто ключові аспекти проєкування та моделювання онлайн-маркетплейсу одягу. Зокрема, розроблено інформаційну модель предметної області, описано архітектуру програмного застосунку, а також розглянуто підходи до проєкування бази даних.

Проаналізовано інструменти проєктування програмного забезпечення, які було використано у цьому розділі. Створено ефективну, масштабовану та легко підтримувану модель системи для онлайн-маркетплейсу. Це сприятиме досягненню поставлених цілей та забезпечить високу якість розробки програмного застосунку.

3 РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ

Spring framework використовується для розробки як серверної, так і клієнтської частин програмного продукту. Для серверної частини використовуються інструменти: Spring Cloud, Spring Boot, Spring MVC або Spring Data, які дозволяють зручно створювати RESTful сервіси, робити взаємодію з базою даних та забезпечувати ефективне управління ресурсами.

У клієнтській частині використовується Spring framework для створення вебінтерфейсу, зазвичай за допомогою інструментів Spring Web та Thymeleaf. Це дозволяє забезпечити безпеку та зручність взаємодії користувачів з програмним продуктом через веббраузер.

Такий підхід з використанням Spring framework дозволяє розробникам детально розкрити технічні аспекти проєкту та забезпечити його ефективну функціональність, максимально використовуючи можливості платформи Java для створення високоякісних програмних рішень.

3.1 Розробка запитів до бази даних відповідно до функціоналу застосунку

У цьому застосунку робота з базами даних виконується за допомогою фреймворка Spring Data, який базується на Spring Framework. Завдяки цьому фреймворку, можна працювати з різними базами даними, а саме: MySQL, MongoDB, PostgreSQL тощо. Взаємодія з цими джерелами даних відбувається, завдяки інтерфейсам та шаблонам доступу до даних, які надає цей фреймворк.

Spring Data має численні засоби для полегшення взаємодії з базами даних. Ось список найважливіших з них [19]:

- репозиторійно-орієнтоване програмування;
- генерація запитів на основі імен методів репозиторію;
- конфігурація бази даних;
- налаштування репозиторію та власні запити;
- інтеграція з іншими частинами Spring.

Для цього проєкту було використано модуль Spring Data JPA, який забезпечує інтеграцію Java Persistence API (JPA) з екосистемою Spring. Цей модуль забезпечує ефективний, зручний, стандартизований спосіб роботи з реляційними базами даних у Spring-застосунках.

Для використання Spring Data JPA, треба спочатку визначитись з базою даних, яка буде використовуватись. В нашому випадку це PostgreSQL, яка покриє всі потреби застосунку.

Для створення сервісу, який може працювати з БД за допомогою Spring Data JPA, треба виконати певні кроки, які будуть розглянуті на прикладі одного з мікросервісів застосунку.

3.1.1 Додавання залежностей

Першим кроком є додавання потрібних залежностей у файл buid.gradle (рис. 3.1).

```
implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
implementation ('mustafaiev.tymur.diploma:model-library')
runtimeOnly 'org.postgresql:postgresql'
```

Рисунок 3.1 – Залежності для роботи з Spring Data JPA

На цьому рисунку видно три залежності, які імпортуються для проєкту:

1) *implementation 'org.springframework.boot:spring-boot-starter-data-jpa'* – ця залежність включає у проєкт Spring Boot Starter для роботи з JPA, яка надає усі необхідні компоненти для роботи;

2) *implementation ('mustafaiev.tymur.diploma:model-library')* – ця залежність є бібліотекою сутностей, яка була створена для того, щоб не доводилось кожного разу їх створювати для кожного сервісу застосунку;

3) runtimeOnly 'org.postgresql:postgresql' – ця залежність включає до проєкту JDBC-драйвер для PostgreSQL. Він потрібен для того, щоб застосунок міг підключатися та взаємодіяти з БД. Оскільки ця залежність вказана як runtimeOnly, драйвер буде увімкнено тільки на етапі виконання, а не на етапі компіляції.

3.1.2 Налаштування параметрів підключення до бази даних

Spring Data JPA дозволяє легко налаштовувати підключення до бази даних через властивості, визначені в application.properties або application.yml, та автоматично підключає необхідні компоненти.

```
# DB connect settings
spring.datasource.url=jdbc:postgresql://localhost:5432/marketplace
spring.datasource.username=keycloak
spring.datasource.password=keycloak

# JPA settings
spring.jpa.hibernate.ddl-auto=update
spring.datasource.driver-class-name=org.postgresql.Driver
```

Рисунок 3.2 – Налаштування у файлі application.properties для Spring Data JPA

На цьому рисунку можна побачити налаштування підключення до БД:

– *spring.datasource.url* вказує *URL* бази даних *PostgreSQL*, до якої буде підключатися застосунок. У даному випадку БД розміщена на локальному хості (localhost) і працює на порту 5432, а назва бази даних — “marketplace”;

– *spring.datasource.username* вказує ім'я користувача, який буде використовуватися для підключення до бази даних;

– *spring.datasource.password* вказує пароль для користувача, що підключається до бази даних.

Налаштування JPA:

– *spring.jpa.hibernate.ddl-auto* визначає, як *Hibernate* повинен керувати схемою бази даних. Значення update означає, що Hibernate буде автоматично оновлювати схему бази даних під час запуску програми, відповідно до змін у моделях сутностей;

– *spring.datasource.driver-class-name* вказує клас драйвера для *JDBC*, який буде використовуватися для підключення до бази даних. У даному випадку використовується драйвер *PostgreSQL*.

3.1.3 Створення класу сутності

Щоб створити сутність для використання в *JPA*, необхідно виконати кілька ключових етапів. Ці етапи розглянемо на прикладі створення сутності з назвою “Product”.

1) Оголошення сутності:

Клас *“Product”* оголошується як сутність, використовуючи анотацію *@Entity*. Це вказує *JPA*, що цей клас повинен бути збережений в базі даних як таблиця.

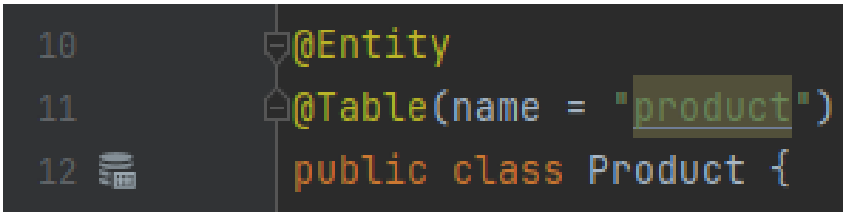


Рисунок 3.3 – Оголошення сутності

Анотація *@Table* визначає, що сутність буде відображатися на таблицю *“product”* в базі даних.

2) Поля класу сутності.

Всі поля класу *“Product”*, які було оголошено, є полями таблиці “product” у базі даних.

Рисунок 3.4 – Поля класу сутності

На рис. 3.4 можна побачити те, що над полями класу також можна додавати деякі анотації, які будуть впливати на властивості колонки у таблиці. У цьому коді задані властивості для поля *“productId”*:

– *@Id* : вказує на те, що що поле є первинним ключем;

– *@GeneratedValue(strategy = GenerationType.IDENTITY):* Ця анотація вказує на те, що значення *“productId”* буде генеруватися автоматично. А значення GenerationType.IDENTITY вказує на те, що ідентичні значення будуть генеруватися базою даних.

Також при розробці інших сутностей були використані такі анотації:

– *@Enumerated(EnumType.STRING)* використовується для збереження значень перерахування (enum) в базі даних як рядків;

– *@Column()* використовується для визначення властивостей полів бази даних: назва, чи може бути *NULL*, дозвіл на оновлення тощо;

– *@Email* забезпечує валідацію, що значення поля є допустимою email адресою;

– *@NotNull* забезпечує валідацію того, що значення поля не є *NULL*.

3) Стосунки між сутностями.

```
3 usages
@ManyToMany
@JoinTable(
        name = "product_category",
        joinColumns = @JoinColumn(name = "product_id"),
        inverseJoinColumns = @JoinColumn(name = "category_id")
)

private Set<Category> categories = new HashSet<>();

3 usages
@OneToMany(mappedBy = "product")
private List<BasketItem> basketItems = new ArrayList<>();

3 usages
@ManyToOne
@JoinColumn(name = "user_id")
private User seller;
```

Рисунок 3.3 – Стосунки між сутностями

На рис. 3.3 видно те, які стосунки з іншими сутностями є в сутності *“Product”*, а саме стосунки:

– *@ManyToMany*: ця анотація вказує на стосунок «багато до багатьох» між сутністю “*Product”* та сутністю *“Category”*. Він реалізований через проміжну таблицю product_category, яка зберігає зв'язки між ідентифікаторами продуктів та категорій;

– *@OneToMany*: ця анотація вказує на стосунок «один до багатьох» між сутністю Product та сутністю BasketItem. Поле *mappedBy = "product"* вказує на те, що зв'язок між *“Product”* та *“BasketItem”* визначений полем *product* у класі *“BasketItem”*;

– *@ManyToOne*: ця анотація вказує на стосунок «багато до одного» між сутністю *“Product”* та сутністю *“User”*. *@JoinColumn(name = "user_id")* вказує на поле в таблиці Product, яке містить ідентифікатор користувача, що є продавцем.

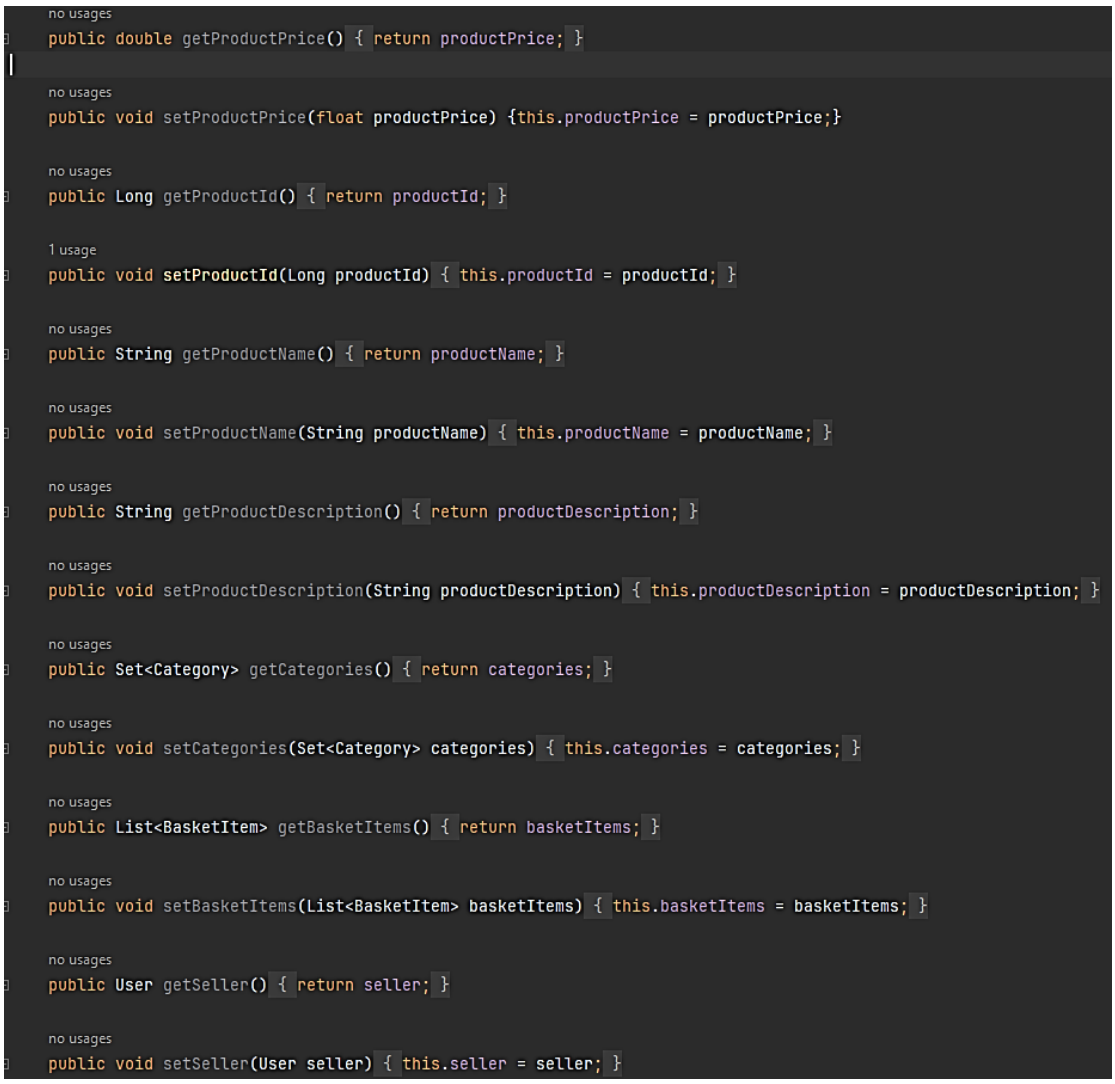
Окрім цих анотацій для налаштування стосунків, які є в сутності *“Product”, “@ManyToOne”, “@ManyToMany”, “@JoinTable”* та інші.

4) Конструктори.

```
no usages
39  public Product(Long productId, String productName,
40                 String productDescription, float productPrice,
41                 Set<Category> categories, List<BasketItem> basketItems,
42                 User seller) {
43      this.productId = productId;
44      this.productName = productName;
45      this.productDescription = productDescription;
46      this.productPrice = productPrice;
47      this.categories = categories;
48      this.basketItems = basketItems;
49      this.seller = seller;
50  }
51
52  public Product() {
53
54  }
```

Рисунок 3.4 – Конструктор з усіма аргументами та без

5) Геттери і сеттери.



```
no usages
public double getProductPrice() { return productPrice; }

no usages
public void setProductPrice(float productPrice) {this.productPrice = productPrice;}

no usages
public Long getProductId() { return productId; }

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

no usages
public String getProductName() { return productName; }

no usages
public void setProductName(String productName) { this.productName = productName; }

no usages
public String getProductDescription() { return productDescription; }

no usages
public void setProductDescription(String productDescription) { this.productDescription = productDescription; }

no usages
public Set<Category> getCategories() { return categories; }

no usages
public void setCategories(Set<Category> categories) { this.categories = categories; }

no usages
public List<BasketItem> getBasketItems() { return basketItems; }

no usages
public void setBasketItems(List<BasketItem> basketItems) { this.basketItems = basketItems; }

no usages
public User getSeller() { return seller; }

no usages
public void setSeller(User seller) { this.seller = seller; }
```

Рисунок 3.5 – Геттери та сеттери сутності

Сутності такого виду коректно створюють таблиці у БД та можуть використовуватись у застосунку. Важливим є факт того, що назви полів сутності при створенні таблиці змінюються. А точніше замінює camelCase на under_score.

3.1.4 Створення інтерфейсу репозиторія

Інтерфейс репозиторія можна створити за допомогою JpaRepository, який надається Spring Data JPA. Цей інтерфейс вже містить реалізацію стандартних CRUD-операцій для роботи з базою даних. На рис. 3.6 показано, як створити інтерфейс репозиторія за допомогою *JpaRepository*.

```java
import mustafaiev.tymur.diploma.modellibrary.model.Product;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;

3 usages
public interface ProductRepository extends JpaRepository<Product, Long> {
    1 usage
    List<Product> findAllByProductName(String productName);
}
```

Рисунок 3.6 – Інтерфейс репозиторія

На рис. 3.6 видно анотацію *@Repository,* яка позначає інтерфейс *ProductRepository* як *Spring* компонент, що підлягає автоматичному скануванню та ініціалізації. *ProductRepository* розширює інтерфейс *JpaRepository*, який приймає на вхід сутність Product та унікальний ідентифікатор типу Long. Метод *findByProductName* використовує стандартне іменування Spring Data JPA для генерації запиту до бази даних, приймаючи назву товару як параметр та повертаючи список товарів.

Завдяки інструментам, які надає *Spring Data JPA*, було розроблено репозиторії для кожної сутності створюваного проєкту. Крім того, як у прикладі з репозиторієм для сутності *“Product”* для деяких сутностей було розроблено додаткові запити до БД.

3.2 Розробка серверної частини з використанням Spring Framework

Серверна частина застосунку онлайн-маркетплейсу одягу написана з використанням мікросервісної архітектури, реалізованої засобами фреймворку Spring. При розробці застосунку було використано такі технології, як *Spring Cloud, Spring Boot, Spring Web* та інші, що дозволило створити гнучку, масштабовану та ефективну серверну частину.

Критично важливим для мікросервісної архітектури, є мати потужний інструмент, який буде надавати функціонал балансування навантаження, конфігурації сервісів, виявлення сервісів та багато іншого. Таким інструментом є

Spring Cloud, використання якого забезпечує високу продуктивність, надійність та масштабованість застосунку.

Самі ж мікросервіси написані з використанням Spring Boot, який забезпечує швидкий старт розробки та мінімізує необхідність ручної конфігурації завдяки вбудованим засобам автонастроювання. Також використовується *Spring Web* для обробки *HTTP*-запитів та створення *RESTful API*, що дозволяє легко інтегрувати клієнтську частину з серверною.

У мікросервісах, які передбачають в собі роботу з БД, використовується *Spring Data JPA*, принцип роботи якого було описано у попередньому підрозділі. Цей інструмент значно спрощує роботу з базами даних, дозволяючи зосередитись на бізнес-логіці, а не на низькорівневих операціях з даними.

Для автоматизації збірки проєктів, використовується *Gradle*. Його часто можна побачити в застосунках з мікросервісною архітектурою. Він дозволяє зручно керувати залежностями, збирати та розгортати кожен мікросервіс окремо, забезпечуючи гнучкість та ефективність у роботі з великою кількістю невеликих і незалежних компонентів. Його можна налаштувати для автоматизації процесів збірки, тестування та розгортання кожного мікросервісу, що робить його незамінним інструментом у розробці масштабованих мікросервісних застосунків.

При розробці будь-якого застосунку, який орієнтований на велику аудиторію та дозволяє створювати облікові записи, постає питання захисту даних користувача та системи. Одним з ключових аспектів безпеки є автентифікація та авторизація користувачів, а також захист ресурсів застосунку від несанкціонованого доступу.

У випадку з застосунком, який розроблюється в рамках цієї роботи і реалізовує мікросервісну архітектуру та побудованому з використанням Spring Cloud, для вирішення цих завдань було принято рішення використовувати не дефолтну реалізацію безпеки, використовуючи сессії.

Зв’язано це з тим, що ця модель реалізації безпеки не підходить для мікросервісних застосунків оскільки порушує один з найголовніших принципів мікросервісної архітектури це незалежність сервісів. Бо кожен запит до сервісу

повинен містити в собі всю необхідну інформацію для його обробки, не зберігаючи стану між сервісами.

Взявши до уваги усі нюанси реалізації безпеки у мікросервісах. Було прийнято рішення використовувати *JSON Web Token* (*JWT*) який дозволяє покращити безпеку та спростити управління доступом. Використання цих токенів дає можливість легше інтегруватися з іншими системами, включаючи зовнішні служби автентифікації та авторизації, такі як *OAuth2* і *OpenID Connect*, а також з системою управління ідентифікацією *Keycloak*.

Використання OAuth2 забезпечує механізм автентифікації та авторизації, який дозволяє користувачам безпечно входити в систему, використовуючи сторонні сервіси автентифікації. У свою чергу, Keycloak виступає як сервер ідентифікації та авторизації, забезпечуючи централізоване управління користувачами, ролями та доступами.

Використовуючи усі ті технології, які були представлені вище, був розроблено мікросервісний застосунок, кожну частину якого буде розглянуто далі.

3.2.1 Eureka Server

Eureka Server– це компонент Spring Cloud, який використовується для реєстрації та керування мікросервісами. Він дозволяє мікросервісам реєструватись у центральному реєстрі, щоб їх можна було знайти іншими мікросервісами за їхніми іменами. Також Eureka Server надає механізм виявлення відмов мікросервісів і автоматичного їхнього відновлення, що робить його важливою складовою для побудови надійних і відмовостійких мікросервісних систем.

Розробка цього сервісу, як і інших, які будуть розглянуті у подальшому, починається з визначення потрібних залежностей для проєкту. Які в свою чергу задаються у файлі build.gradle.

```
dependencies {
    implementation 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-server'
```

Рисунок 3.7 – Залежності для Eureka Server

Для цього сервісу нам потрібна лише одна залежність, а саме *'org.springframework.cloud:spring-cloud-starter-netflix-eureka-server'*.

Далі потрібно анотувати клас *EurekaServerApplication,* який є основним класом з методом main для запуску застосунку, двома анотаціями:

– *@SpringBootApplication*: вказує, що цей клас є основним класом Spring Boot застосунку. Вона об'єднує три анотації: *@Configuration, @EnableAutoConfiguration і @ComponentScan*, що дозволяє автоматично налаштовувати застосунок, визначаючи конфігураційні класи, включаючи залежності та скануючи компоненти.

– *@EnableEurekaServer*: вказує, що цей застосунок є Eureka Server, компонентом Spring Cloud, який використовується для реєстрації та керування мікросервісами в розподіленій архітектурі. Вона активує функціональність Eureka Server у Spring Boot застосунку.

Рисунок 3.8 – Головний клас Spring Boot застосунку

Під час запуску методу main викликається метод run класу *SpringApplication*, що ініціалізує та запускає *Spring Boot* застосунок.

Також для коректної роботи цього сервісу варто додати певні налаштування у файлі application.properties:

Рисунок 3.9 – Конфігураційний файл застосунку

На цьому зображенні видно такі параметри:

– *server.port=8761*: Встановлює порт, на якому працюватиме Eureka Server;

– *spring.application.name=eserver*: Встановлює ім'я застосунку як *“eserver”*.

– *eureka.client.register-with-eureka=false*: Вимикає автоматичну реєстрацію Eureka Server в самому собі.

– *eureka.client.fetch-registry=false*: Вимикає витягування реєстрації інших мікросервісів.

– *eureka.client.service-url.defaultZone=http://localhost:8761/eureka*: Встановлює URL-адресу за замовчуванням для Eureka-сервера;

– *logging.level.com.netflix.eureka=OFF*: Вимикає логування для пакета *com.netflix.eureka*;

– *logging.level.com.netflix.discovery=OFF*: Вимикає логування для пакета *com.netflix.discovery*;

– *logging.pattern.console=%C{1.} [%-5level] %d{HH:mm:ss}– %msg%n*: Встановлює шаблон для виведення повідомлень в консоль. У цьому шаблоні використовуються рівень логування, дата та час, а також текст повідомлення.

Цей сервіс показує лише зареєстровані мікросервіси, налаштування для цього буде продемонстровано у подальшому.

3.2.2 Docker контейнери

Для того, щоби створити та використовувати Keycloak клієнта для розробленого застосунку. Було прийнято рішення розгорнути *Keycloak* у *Docker* контейнері зв’язавши його за базою даних *PostgreSQL* яка використовується у застосунку. Також для зручного перегляду інформації з БД був створений контейнер з графічним клієнтом *pgAdmin*.

Розглядаючи *Keycloak*, БД та графічний клієнту, треба більш досконало розкрити значення технології завдяки якої було розгорнуто ці засоби, а саме *Docker*. Який представляє собою контейнерну платформу програмного забезпечення, створену для розроблення, надсилання та запуску застосунків із використанням контейнерних технологій.

Контейнер, є стандартною програмною одиницею, яка упаковує код і всі його залежності, тому застосунки можуть швидко і надійно запускатися з одного середовища в інше. Контейнери непостійні і створюються з образів, які є не змінною одиницею що містять набір інструкцій для створення контейнера, який може працювати на платформі *Docker* [20].

Взявши до уваги усі переваги та особливості розгортання інструментів розробки у Docker контейнерах, для використання у розроблюваному застосунку прийнято рішення використовувати docker-compose файл, який містить в собі налаштування Docker-образів та їх взаємодії один з одним.

В свою чергу, *Docker Compose* є інструментом для визначення та запуску багатоконтейнерних застосунків. Compose спрощує керування всім стеком застосунків, дозволяючи легко керувати сервісами, мережами та томами в одному зрозумілому конфігураційному файлі *YAML*. Потім за допомогою однієї команди, створюються і запускаються всі сервіси з конфігураційного файлу [21].

Саме для розгортання потрібних для розроблюваного застосунку зв’язаних контейнерів був створений docker-compose.yaml файл, який має вигляд:

1. Налаштування контейнеру *Keycloak* (рис. 3.10).

```yaml
services:
  keycloak_svr:
    image: quay.io/keycloak/keycloak
    volumes:
      - ./jar:/opt/keycloak/providers
    command: start-dev
    environment:
      KC_DB: postgres
      KC_DB_URL_HOST: postgres_svr
      KC_DB_URL_DATABASE: keycloak
      KC_DB_USERNAME: keycloak
      KC_DB_PASSWORD: keycloak
      KC_DB_SCHEMA: public
      KEYCLOAK_ADMIN: admin
      KEYCLOAK_ADMIN_PASSWORD: admin
    ports:
      - "8180:8080"
    depends_on:
      postgres_svr:
        condition: service_healthy
    networks:
      network_sso:
        ipv4_address: "10.0.0.2"

```

Рисунок 3.10 – Налаштування Keycloak контейнеру

На цій ділянці коду (рис. 3.10) налаштовується *Keycloak*, завдяки таким рядкам коду:

1) *services*: розділ, у якому описані всі контейнери, які будуть запущені;

2) *keycloak_svr*: назва для контейнеру у якому буде розгорнуто *Keycloak*;

3) *image*: вказує образ Docker, який буде використаний для запуску контейнера;

4) *volumes:– ./jar:/opt/keycloak/providers*: монтує локальний каталог ./jar в контейнер для провайдерів *Keycloak*;

5) command: start-dev: виконує команду *start-dev* для запуску *Keycloak* в режимі розробки.

6) *environment*: описує змінні середовища для налаштування, саме для Keycloak, юуло використано такі змінні:

- *KC_DB*: тип бази даних;
- *KC_DB_URL_HOST*: ім'я хосту бази даних PostgreSQL;
- *KC_DB_URL_DATABASE*: назва бази даних;
- *KC_DB_USERNAME*: ім'я користувача для підключення до бази даних;
- *KC_DB_PASSWORD*: пароль для підключення до бази даних;
- *KC_DB_SCHEMA*: схема бази даних;
- *KEYCLOAK_ADMIN*: ім'я адміністратора *Keycloak*;
- *KEYCLOAK_ADMIN_PASSWORD*: пароль адміністратора *Keycloak*;

7) *ports:– "8180:8080"*: прокидання порту 8080 всередині контейнера на порт 8180 на локальному хості;

8) *depends_on: postgres_svr: condition: service_healthy*: вказує, що контейнер Keycloak буде запущений тільки після того, як контейнер *“postgres_svr”* буде успішно запущено;

9) networks: network_sso: ipv4_address: "10.0.0.2": статична IP-адреса для контейнера *Keycloak* в мережі *network_sso.*

2. Налаштування контейнеру *Postgres*:

```
25    postgres_svr:
26      image: postgres
27      command: postgres -c 'max_connections=200' && postgres -c 'shared_buffers=24MB'
28      environment:
29        # Environment Variables expecially for Postgres
30        POSTGRES_DB: keycloak
31        POSTGRES_USER: keycloak
32        POSTGRES_PASSWORD: keycloak
33      healthcheck:
34        test: "exit 0"
35      ports:
36        - "5432:5432"
37      networks:
38        network_sso:
39          ipv4_address: "10.0.0.3"
```

Рисунок 3.11 – Налаштування Postgres контейнеру

Налаштування контейнерів є достатньо схожим між собою, бо головними етапами його налаштування в нашому випадку є задання образу, змінних, порту та IP-адреса для контейнера в мережі *network_sso*. Усі ці налаштування є у кожного контейнера і відрізняються лише значення. Але варто зауважити що кожен з контейнерів має свій набір змінних, які можуть впливати на його роботу. Тому далі ми будемо розглядати саме змінні кожного контейнера.

У випадку з контейнером *postgres_svr* такими змінними є:

- *POSTGRES_DB*: назва бази даних;
- *POSTGRES_USER*: ім'я користувача для бази даних;
- *POSTGRES_PASSWORD*: пароль для бази даних.

3. Налаштування контейнеру *pgAdmin*:

```
40    pgadmin:
41      image: dpage/pgadmin4
42      environment:
43        PGADMIN_DEFAULT_EMAIL: admin@example.com
44        PGADMIN_DEFAULT_PASSWORD: admin
45        PGADMIN_LISTEN_PORT: 5050
46      ports:
47        - "5050:5050"
48      networks:
49        network_sso:
50          ipv4_address: "10.0.0.4"
```

Рисунок 3.12 – Налаштування pgAdmin контейнеру

На частині коду який на зображенні, для контейнеру pgadmin задані такі змінні:

- *PGADMIN_DEFAULT_EMAIL*: email адміністратора pgAdmin;
- *PGADMIN_DEFAULT_PASSWORD*: пароль адміністратора pgAdmin;
- *PGADMIN_LISTEN_PORT*: порт, на якому pgAdmin буде слухати *HTTP*-запити.

4. Налаштування мережі *Docker*:

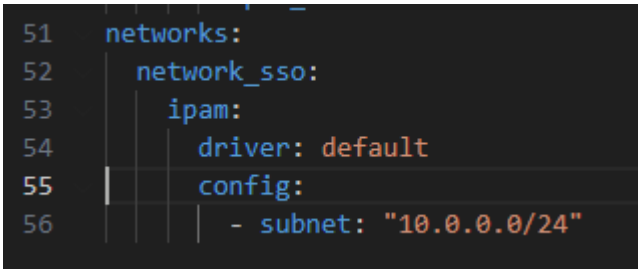


Рисунок 3.13 – Налаштування Docker мережі

Розділ networks (рис. 3.13) описує мережі, використовувані контейнерами. Мережа *network_sso* має налаштування *ipam* з драйвером за замовчуванням для управління *IP*-адресами та конфігурацією, яка задає підмережу з діапазоном *IP*-адрес від 10.0.0.0 до 10.0.0.255.

3.2.3 API Gateway

Цей сервіс представляє собою шлюз API, що виступає як єдина точка входу для клієнтів, які звертаються до нашого застосунку, та маршрутизує запити до мікросервісів. Реалізован він у нашому випадку за допомогою фреймворку Spring Cloud Gateway за допомогою якого можна реалізувати такий функціонал:

– *Маршрутизація запитів*: Забезпечує маршрутизацію HTTP-запитів до відповідних мікросервісів.

– *Фільтри*: Дозволяє додавати різноманітні фільтри для обробки запитів та відповідей.

– *Балансування навантаження*: Інтеграція з іншими компонентами Spring Cloud для реалізації балансування навантаження між екземплярами мікросервісів.

– *Безпека*: Підтримує інтеграцію з механізмами аутентифікації та авторизації, такими як OAuth2, для забезпечення захищеного доступу до мікросервісів.

– *Моніторинг і журналювання:* Дозволяє відстежувати трафік і вести журнал запитів та відповідей, що спрощує діагностику та управління системою.

Для розробки цього сервісу було використано залежності (рис. 3.14).

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-webflux'
    implementation 'org.springframework.cloud:spring-cloud-starter-gateway'
    implementation 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client'
    implementation 'org.springframework.boot:spring-boot-starter-oauth2-resource-server'
    implementation 'org.springframework.boot:spring-boot-starter-security'
    implementation 'org.springframework.security:spring-security-oauth2-client'
```

Рисунок 3.14 – Залежності для API Gateway

На рис. 3.14 зображенні такі залежності:

– *'org.springframework.boot:spring-boot-starter-webflux'*: Це залежність для використання Spring WebFlux, який забезпечує підтримку реактивного програмування в Spring.

– *'org.springframework.cloud:spring-cloud-starter-gateway'*: ця залежність додає підтримку Spring Cloud Gateway;

– *'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client'*: ця залежність дозволяє застосунку працювати як Eureka клієнт.

– *'org.springframework.boot:spring-boot-starter-oauth2-resource-server'*: ця залежність додає підтримку OAuth2 Resource Server.

– *'org.springframework.boot:spring-boot-starter-security'*: ця залежність додає базову підтримку Spring Security.

– *'org.springframework.security:spring-security-oauth2-client'*: ця залежність додає підтримку OAuth2 Client.

Якщо правильно налаштувати шлюз використовуючи ці залежності, то ми на виході отримаємо сервіс, який пропускає через себе усі запити направляє їх до відповідних мікросервісів. Окрім цього буде перенаправляти неавторизованих користувачів, які хочуть отримати доступ до захищеного ресурсу на сторінку входу.

Для того, щоб реалізувати подібний функціонал, в першу чергу, треба задати налаштування для застосунку за допомогою файлу *“application.properties”*.

```
server.port=8765


spring.main.web-application-type=reactive

logging.level.org.springframework=INFO
logging.level.org.springframework.security=DEBUG

spring.application.name=api-gateway

eureka.client.service-url.defaultZone=http://localhost:8761/eureka

logging.pattern.console=%C{1.} [%-5level] %d{HH:mm:ss} - %msg%n

spring.cloud.gateway.discovery.locator.enabled=true

spring.cloud.gateway.discovery.locator.lower-case-service-id=true

logging.level.org.springframework.cloud.gateway=DEBUG

spring.cloud.gateway.default-filters[0]=TokenRelay

```

Рисунок 3.15 – Налаштування для API Gateway ч.1

Цей код конфігурації задає налаштування для API-шлюзу, який працює з Spring Cloud Gateway. Ось докладний опис кожного з цих параметрів:

– *server.port=8765*: Встановлює порт, на якому буде працювати API-шлюз.

– *spring.main.web-application-type=reactive*: вказує, що застосунок має використовувати реактивний стек Spring.

– *logging.level.org.springframework=INFO*: Встановлює рівень логування для пакета *org.springframework* на *INFO*.

– *logging.level.org.springframework.security=DEBUG*: встановлює рівень логування для пакета org.springframework.security на DEBUG.

– *spring.application.name=api-gateway*: встановлює ім'я застосунку як api-gateway.

– *eureka.client.service-url.defaultZone=http://localhost:8761/eureka*: вказує URL сервера Eureka, до якого буде підключатися API-шлюз для реєстрації та виявлення інших мікросервісів.

– *logging.pattern.console=%C{1.} [%-5level] %d{HH:mm:ss}– %msg%n*: визначає шаблон для виводу логів у консоль.

– *spring.cloud.gateway.discovery.locator.enabled=true*: увімкнено автоматичне виявлення сервісів, зареєстрованих у Eureka, і їх маршрутизацію через шлюз.

– *spring.cloud.gateway.discovery.locator.lower-case-service-id=true*: вказує, що ідентифікатори сервісів мають бути перетворені на нижній регістр.

– *logging.level.org.springframework.cloud.gateway=DEBUG*: встановлює рівень логування для пакета org.springframework.cloud.gateway на DEBUG.

– *spring.cloud.gateway.default-filters[0]=TokenRelay*: встановлює фільтр TokenRelay за замовчуванням. Цей фільтр забезпечує перенаправлення OAuth2 токенів аутентифікації від клієнта до цільового сервісу, що корисно для реалізації єдиного входу та передачі авторизаційної інформації між сервісами.

```
33  # /image/**
34  spring.cloud.gateway.routes[2].id=image
35  spring.cloud.gateway.routes[2].uri=http://localhost:8888
36  spring.cloud.gateway.routes[2].predicates[0]=Path=/image/**
37
38  #storage-amazon-s3
39  spring.cloud.gateway.routes[3].id=storage
40  spring.cloud.gateway.routes[3].uri=lb://storage-amazon
41  spring.cloud.gateway.routes[3].predicates[0]=Path=/image-service/**
42
43  spring.cloud.gateway.routes[4].id=order
44  spring.cloud.gateway.routes[4].uri=lb://order-service
45  spring.cloud.gateway.routes[4].predicates[0]=Path=/order/**
46
47  spring.cloud.gateway.routes[5].id=basket
48  spring.cloud.gateway.routes[5].uri=lb://basket-service
49  spring.cloud.gateway.routes[5].predicates[0]=Path=/basket/**
50
51  spring.cloud.gateway.routes[6].id=review
52  spring.cloud.gateway.routes[6].uri=lb://review-service
53  spring.cloud.gateway.routes[6].predicates[0]=Path=/review/**
54
55  spring.cloud.gateway.routes[7].id=category
56  spring.cloud.gateway.routes[7].uri=lb://category-service
57  spring.cloud.gateway.routes[7].predicates[0]=Path=/category/**
58
59  spring.cloud.gateway.routes[8].id=product
60  spring.cloud.gateway.routes[8].uri=lb://product-service
61  spring.cloud.gateway.routes[8].predicates[0]=Path=/product/**
```

Рисунок 3.16 – Налаштування для API Gateway ч.2

Розглянемо шматок коду з картинки, який повторюється:

– *spring.cloud.gateway.routes[3].id=storage*: Встановлює ідентифікатор для цього маршруту як storage.

– *spring.cloud.gateway.routes[3].uri=lb://storage-amazon*: вказує URI для цільового сервісу. Префікс lb:// вказує на те, що API-шлюз має використовувати балансування навантаження для цього сервісу.

– *spring.cloud.gateway.routes[3].predicates[0]=Path=/image-service/***: вказує на предикат маршруту, який визначає умову для відповідності запитів цьому маршруту. У цьому випадку, будь-який запит, URL якого починається з /image-service/, буде спрямований на сервіс storage-amazon.

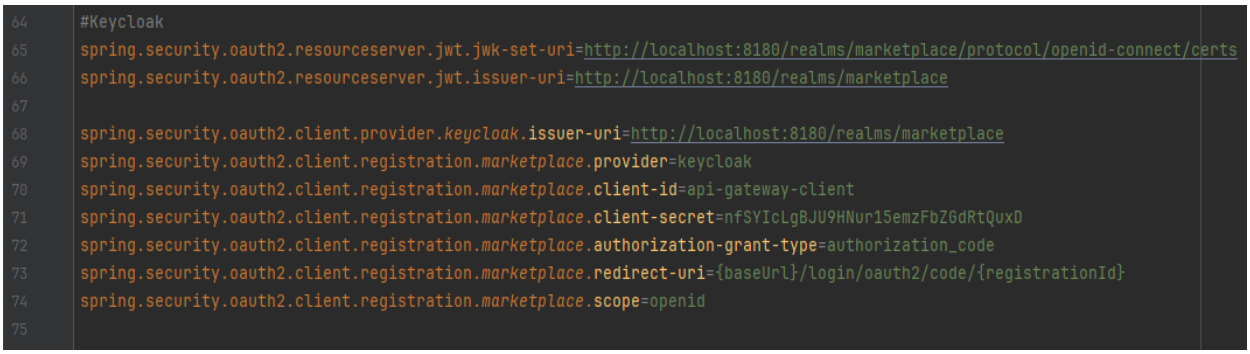


Рисунок 3.17 – Налаштування OAuth2

Цей фрагмент конфігурації (рис. 3.13) налаштовує інтеграцію *Spring Security* з *Keycloak* для забезпечення автентифікації та авторизації в мікросервісному застосунку за допомогою *OAuth2* у сервісі шлюзу *API Gateway*.

Ці налаштування включають у себе дві частини налаштування *OAuth2*, а точніше *Resources Server* та *OAuth2 Client*. Перший представляє собою сервер ресурсів який обробляє аутентифіковані запити після того, як застосунок отримав токен доступу [22].

В свою чергу *OAuth2 Client* це застосунок, що робить запити до захищеного ресурсу від імені власника ресурсу та з його дозволу [23].

Ось опис кожного параметра:

– *spring.security.oauth2.resourceserver.jwt.jwk-set-uri*: вказує *URI* для отримання *JWKS*, який використовується для перевірки підписів *JWT*-токенів;

– *spring.security.oauth2.resourceserver.jwt.issuer-uri*: вказує *URI* видавця токенів, який буде використовуватися для перевірки автентичності *JWT*-токенів;

– *spring.security.oauth2.client.provider.keycloak.issuer-uri*: встановлює URI видавця токенів для Keycloak;

– s*pring.security.oauth2.client.registration.marketplace.provider*: встановлює ім'я провайдера;

– *spring.security.oauth2.client.registration.marketplace.client-id*: вказує ID клієнта, зареєстрованого в Keycloak;

– *spring.security.oauth2.client.registration.marketplace.client-secret*: вказує секретний ключ клієнта, який використовується для автентифікації з *Keycloak*;

– *spring.security.oauth2.client.registration.marketplace.authorization-grant-type*: визначає тип гранту авторизації, який буде використовуватися для отримання токенів;

– *spring.security.oauth2.client.registration.marketplace.redirect-uri*: вказує URI для переадресації після авторизації;

– *spring.security.oauth2.client.registration.marketplace.scope*: визначає *scopes*, які будуть запитані у *Keycloak*.

Всі ці налаштування разом допомагають забезпечити безпеку і контроль доступу в застосунку, побудованому на основі OAuth2, забезпечуючи зручний і захищений спосіб управління автентифікацією і авторизацією користувачів і застосунків.

Для розуміння того, що було налаштовано для надання авторизації та автентифікації застосунку у файлі application.properties, треба детальніше розглянути деякі терміни, які використовувались у конфігурації *OAuth2*. А саме терміни:

1) *Client Secret* – це важливий секретний код, який відомий лише застосунку та серверу авторизації, генерується та знаходиться цей код у розділі *“Credentials”* створеного для застосунку клієнта *Keycloak*.

2) *Authorization Grant Types*– методи або процеси, які клієнтські програми використовують для запиту токену доступу від сервера авторизації, в OAuth називаються типами грантів. Кожен тип гранту створено для різних обставин, потреб безпеки та можливостей клієнта [24].

3) *Scope*– це механізм в OAuth2 для обмеження доступу програми до облікового запису користувача. Застосунок може запросити один або декілька діапазонів, а токен доступу, виданий йому, буде мати дані у відповідністю з обраним *Scope* [25].

Наступним етапом після задання налаштувань у файлі *application.properties* є створення конфігураційного *Java* файлу. В нашому проєкті він називається *SecurityConfig.java* (рис. 3.18).

```
@Configuration
@EnableWebFluxSecurity
public class SecurityConfig {

    @Bean
    public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
        http.authorizeExchange(auth -> auth
                        .pathMatchers("/shop/*").permitAll()
                        .anyExchange().authenticated())
                .oauth2Login( Customizer.withDefaults())
                .oauth2ResourceServer((oauth2) -> oauth2.jwt(Customizer.withDefaults()));
        http.csrf(ServerHttpSecurity.CsrfSpec::disable);
        return http.build();
    }

}
```

Рисунок 3.18 – Файл конфігурації SecurityConfig

Код, який міститься на зображенні вище, представляє собою налаштування безпеки для шлюзу, який використовує *Spring WebFlux*. Він конфігурує аутентифікацію та авторизацію, використовуючи *OAuth2* і *JWT* для захисту ресурсів.

Використання для шлюзу *Spring Cloud Gateway* саме конфігурації безпеки як для *Spring WebFlux* застосунку зв’язано з тим, що він є реактивним, тому що через нього водночас може проходити безліч запитів.

Враховуючи це було написано конфігураційний *Java* файл, який було анотовано анотацією @Configuration та @EnableWebFluxSecurity, які вказують на

те що клас є конфігураційним та активує підтримку налаштування безпеки для реактивних застосунків. У самому класі було задано такі налаштування:

1) http.authorizeExchange(auth– > auth

.pathMatchers("/shop/").permitAll()*

.anyExchange().authenticated()):

- цей код визначає, що запити, які відповідають шаблону шляху /shop/*, дозволені для всіх без аутентифікації;
- усі інші запити повинні бути автентифіковані.

2) *.oauth2Login(Customizer.withDefaults())*:

- включає підтримку аутентифікації через OAuth2, що дозволяє користувачам входити до системи. У нашому випадку використовуючи зовнішнього постачальника *Keycloak*.

3) *.oauth2ResourceServer(oauth2– > oauth2.jwt(Customizer.withDefaults()))*:

- ця частина конфігурації включає підтримку ресурсного сервера, який перевіряє *JWT* для автентифікації запитів.

4) http.csrf(ServerHttpSecurity.CsrfSpec::disable):

- Цей рядок відключає захист *CSRF*, бо він не потрібен при використанні токенів доступу. Бо вірогідність такої атаки на застосунок де використовуються *JWT* близька нулю.

5) return http.build():

- цей метод будує та повертає конфігурацію безпеки для застосунку.

Так, було розроблено сервіс шлюзу API Gateway для мікросервісного застосунку, розробленого у рамках цієї роботи.

3.2.4 AWS S3 Storage

Мікросервіс з назвою *“storage-amazon-s3”* було створено для роботи з файлами зображень в *AWS S3*. Він включає в себе функціональність завантаження файлів, їх зберігання у *S3* та отримання файлів за запитом. Він може бути легко інтегрований з іншими мікросервісами системи, забезпечуючи централізоване сховище для зображень та зручний доступ до них.

У рамках цієї роботи було створено такий мікросервіс, аспектами створення якого є використання Spring Boot та *AWS S3*. Використовуючи ці технології, був створений вебзастосунок, який буде розглянуто далі.

Як і для усіх інших, першим кроком для створення цього сервісу є додавання певних залежностей, які будуть використані при розробці (рис. 3.19).

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client'
    implementation 'software.amazon.awssdk:s3:2.23.2'
    implementation 'com.amazonaws:aws-java-sdk-core:1.12.636'
```

Рисунок 3.19 – Файл з залежностями для "storage-amazon-s3"

На цьому зображенні показано такі залежності:

- *org.springframework.boot:spring-boot-starter-web*: Ця залежність додає базовий функціонал Spring Boot для створення вебзастосунків;
- *org.springframework.cloud:spring-cloud-starter-netflix-eureka-client*: включає компонент *Netflix Eureka Client* для реєстрації мікросервісу в *Eureka Server*;
- *software.amazon.awssdk:s3*: додає *AWS SDK* для *Java*, який надає *API* для взаємодії з *Amazon S3*. Він дозволяє здійснювати операції зберігання та керування об'єктами в хмарному сховищі S3.
- *com.amazonaws:aws-java-sdk-core*: ця залежність також додає *AWS SDK* основні бібліотеки, які необхідні для взаємодії з AWS-хмарними сервісами.

Наступним кроком є задання потрібних налаштувань у файлі *application.properties*, який має такий вигляд:

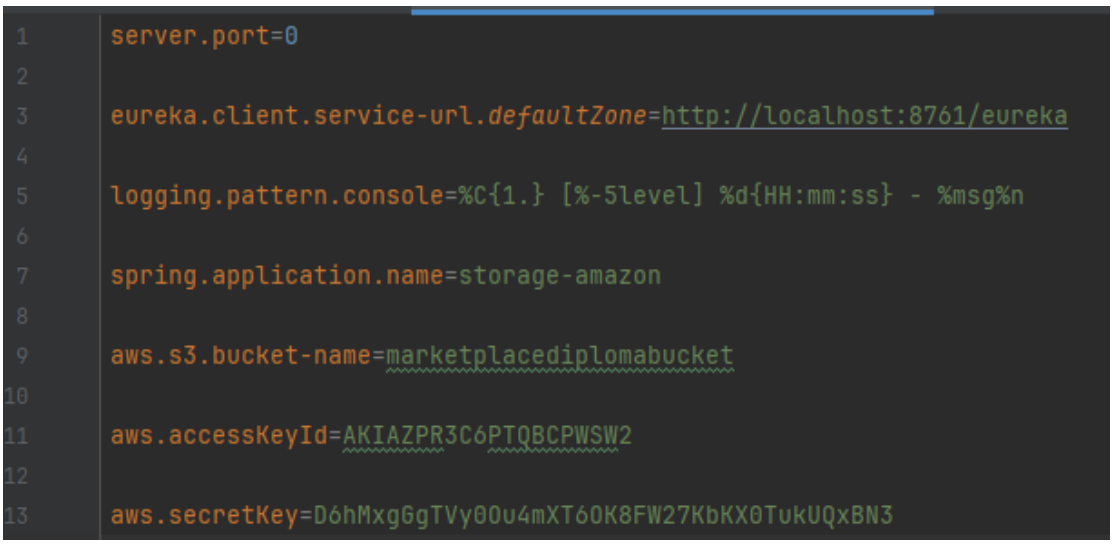
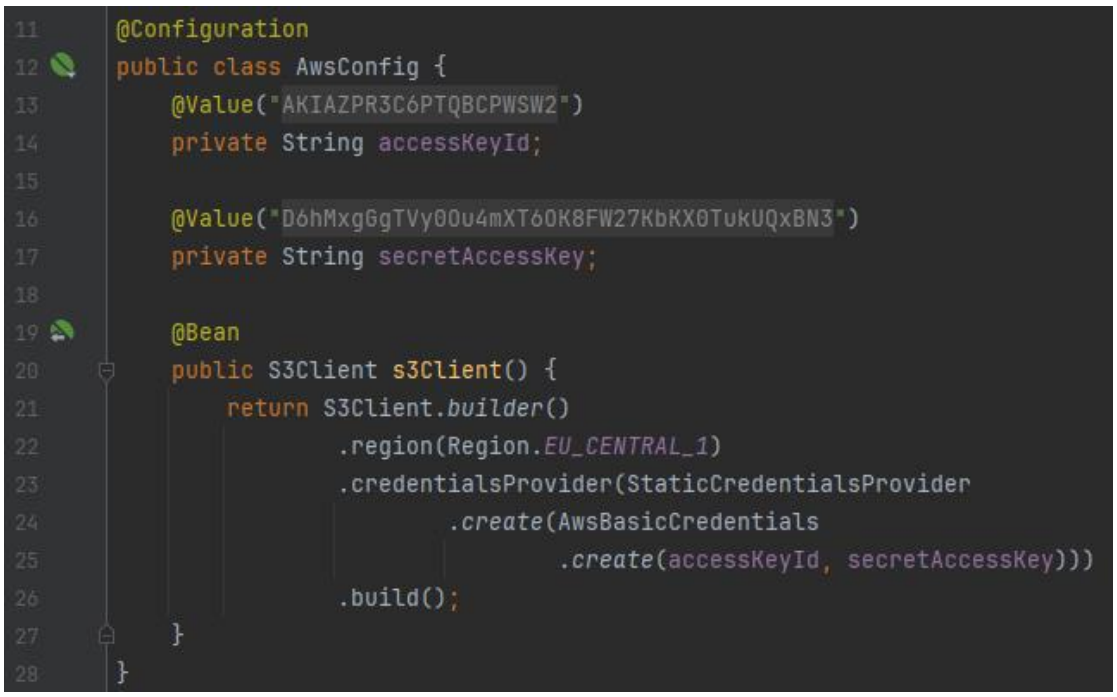


Рисунок 3.19 – Файл з налаштуваннями для "storage-amazon-s3"

На цьому зображенні видно такі налаштування:

- *server.port=0*: це налаштування вказує *Spring Boot* вибрати вільний порт для запуску застосунку. Значення *"0"* означає, що порт буде вибрано автоматично;
- *eureka.client.service-url.defaultZone*: вказує на *URL*-адресу реєстрації у *Eureka*-сервері;
- *logging.pattern.console*: визначає формат виведення журналу в консоль;
- s*pring.application.name*: ім'я застосунку Spring Boot яке використовується для ідентифікації застосунку в середовищі мікросервісів та в реєстрі *Eureka*;
- *aws.s3.bucket-name*: цей параметр вказує назву кошика *S3* в сервісі *Amazon S3*, з яким буде взаємодіяти сервіс;
- aws.accessKeyId та aws.secretKey: Ці параметри вказують ключі доступу *AWS*, які використовуються для автентифікації застосунку в сервісі *Amazon S3*.

Далі було створено конфігураційний *Java* файл (рис. 3.20).



```
@Configuration
public class AwsConfig {
    @Value("AKIAZPR3C6PTQBCPWSW2")
    private String accessKeyId;

    @Value("D6hMxgGgTVy0Ou4mXT6OK8FW27KbKX0TukUQxBN3")
    private String secretAccessKey;

    @Bean
    public S3Client s3Client() {
        return S3Client.builder()
                .region(Region.EU_CENTRAL_1)
                .credentialsProvider(StaticCredentialsProvider
                        .create(AwsBasicCredentials
                                .create(accessKeyId, secretAccessKey)))
                .build();
    }
}
```

Рисунок 3.20 – Файл Java конфігурації для "storage-amazon-s3"

У коді на рис. 3.20 можна побачити вилучення значень з файлу *application.properties* за допомогою анотацій *@Value*. Ці дані, які було вилучено для використання у створенні біна, що є складовою частиною застосунку та може використовуватися іншими бінів через механізм залежностей. Методи, які розробляються в якості біна, треба помітити анотацією *@Bean*, для використання у застосунку.

В свою чергу бін, який створюється у цьому файлі, створює та налаштовує клієнта *S3* з використанням облікових даних, що зберігаються у властивостях програми. А виконується це завдяки коду:

- *S3Client.builder()*: створює новий екземпляр будівельника для клієнта *S3*;
- *.region(Region.EU_CENTRAL_1)*: вказує, що клієнт *S3* буде працювати в регіоні *EU_CENTRAL_1* (Франкфурт);
- *.credentialsProvider(StaticCredentialsProvider.create(AwsBasicCredentials.create(accessKeyId, secretAccessKey)))*: налаштовує клієнт для використання облікових даних, наданих у конфігураційному файлі. *AwsBasicCredentials.create(accessKeyId, secretAccessKey)* створює об'єкт з

обліковими даними, а *StaticCredentialsProvider.create()* обертає їх у провайдер, який буде використовуватися клієнтом *S3* для автентифікації;

- *.build()*: будує та повертає конфігурований екземпляр *S3Client*.

Наступним кроком було створення сервісного класу, в якому написані методи для завантаження та отримання зображень зі сховища. Такі класи мають помітку *@Service*, саме як і клас *S3Service* (рис. 3.21) для роботи з зображеннями.

```
17      @Service
18      public class  S3Service {
            4 usages
19          private final S3Client amazonS3Client;
20
21          @Value("${aws.s3.bucket-name}")
22          private String bucketName;
23
24          public S3Service(S3Client amazonS3Client) {
25              this.amazonS3Client = amazonS3Client;
26          }
```

Рисунок 3.21 – Файл класу S3Service

Також на цьому зображенні можна побачити об’явлення двух змінних, які будуть використовуватись у коді. Одна з них є *S3Client* яка також додана до конструктору та оголошена як *final*. Зв’язано це з тим, що *Spring* ініціалізує змінні через конструктор, а оголошення полів як *final* дає гарантію що усі залежності будуть корректно ініціалізовані при створенні екземпляру класу.

Цей клас має три методи для взаємодії зі сховищем зображень, а саме:

1) uploadFile (рис. 3.22).

```
public void uploadFile(String key, MultipartFile file) throws IOException {
    amazonS3Client.putObject(PutObjectRequest.builder()
                    .bucket(bucketName)
                    .key(key)
                    .build(),
            RequestBody.fromBytes(file.getBytes()));
}
```

Рисунок 3.22 – Метод uploadFile *класу S3Service*

На цьому зображенні показано метод для завантаження зображення на сервер *S3*, а виконується це такому:

- *PutObjectRequest.builder()*: створює новий об'єкт запиту на завантаження файлу в S3.
- *.bucket(bucketName)*: вказує ім'я бакету, до якого завантажується файл, яке задано у змінній *bucketName*;
- *.key(key)*: встановлює шлях для об'єкта в S3. Цей ключ визначає, де саме в бакеті буде зберігатися файл;
- *RequestBody.fromBytes(file.getBytes())*: перетворює вміст файлу в масив байтів та створює з нього тіло запиту, у цьому коді саме метод file.getBytes() отримує байтове представлення вмісту файлу.

2) uploadSomeFiles (рис. 3.23).

```
public void uploadSomeFiles(int id, MultipartFile[] files) throws IOException {
    String key;
    int num=1;
    for (MultipartFile file : files) {
        key = "id_" + id + "/" + num+".png";

        amazonS3Client.putObject(PutObjectRequest.builder()
                        .bucket(bucketName)
                        .key(key)
                        .build(),
                RequestBody.fromBytes(file.getBytes()));
        num++;
    }
}
```

Рисунок 3.23 – Метод uploadFile *класу S3Service*

Цей метод, представлений на зображенні вище, виконує ті ж самі операції, що і попередній. Але у випадку з цим методом, він приймає на вхід масив файлів, та зберігає його у сховищі.

3) getFile (рис. 3.24).

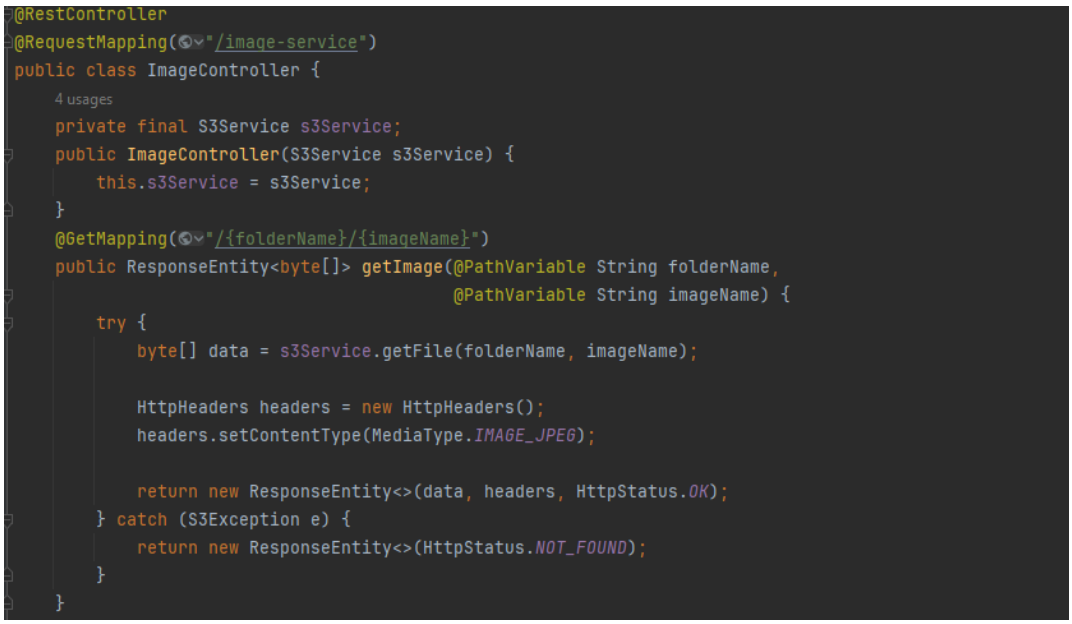
```
public byte[] getFile(String folderName, String fileName) throws S3Exception {
    GetObjectRequest getObjectRequest = GetObjectRequest.builder()
            .bucket(bucketName)
            .key(folderName + "/" + fileName)
            .build();

    ResponseBytes<GetObjectResponse> getObjectResponse = amazonS3Client.getObjectAsBytes(getObjectRequest);
    return getObjectResponse.asByteArray();
}
```

Рисунок 3.24 – Метод uploadFile *класу S3Service*

Метод, програмний код якого можна побачити на цьому зображенні, виконує запит до серверу зображень та повертає масив байтів. Він приймає на вхід дві змінні, які є назвою директорії та самого файлу. Завдяки цим даним виконуються такі методи програмного коду:

- *GetObjectRequest.builder()*: створює новий об'єкт запиту для завантаження файлу з *S3*;
- *.bucket(bucketName)*: встановлює ім'я бакету, з якого буде завантажено файл;
- *.key(folderName + "/" + fileName)*: встановлює шлях до файлу в *S3*, який формується завдяки об'єднання назви папки та файлу.
- *amazonS3Client.getObjectAsBytes(getObjectRequest)*: виконує запит на завантаження об'єкта з *S3* й отримує відповідь у вигляді *ResponseBytes<GetObjectResponse>*;
- *getObjectResponse.asByteArray()*: перетворює вміст відповіді у масив байтів і повертає його.

Використовуючи цей клас-сервіс, було створено клас-контроллер, який надає *REST API* для роботи з зображеннями, що зберігаються у *Amazon S3* (рис. 3.25).



```
@RestController
@RequestMapping("/image-service")
public class ImageController {
    4 usages
    private final S3Service s3Service;
    public ImageController(S3Service s3Service) {
        this.s3Service = s3Service;
    }
    @GetMapping("/{folderName}/{imageName}")
    public ResponseEntity<byte[]> getImage(@PathVariable String folderName,
                                           @PathVariable String imageName) {
        try {
            byte[] data = s3Service.getFile(folderName, imageName);

            HttpHeaders headers = new HttpHeaders();
            headers.setContentType(MediaType.IMAGE_JPEG);

            return new ResponseEntity<>(data, headers, HttpStatus.OK);
        } catch (S3Exception e) {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    }
```

Рисунок 3.25 – Клас-контролер для роботи з зображеннями

На цьому зображенні можна побачити клас *ImageController*, який помічено двома анотаціями:

- *@RestController*: вказує, що цей клас є *REST*-контролером, що дозволяє обробляти *HTTP*-запити і повертати дані у вигляді *JSON* або інших форматів;
- *@RequestMapping("/image-service")*: встановлює базовий шлях для всіх кінцевих точок у цьому контролері.

Також можна побачити оголошення змінної сервісу, розробленого раніше, операції з якою будуть виконуватись у методах, один з яких є на цьому зображенні. А саме метод *getImage*, який виконує *Get*-запрос, для вилучення зображення. Досягається це завдяки додання анотації *@GetMapping("/{folderName}/{imageName}")*, яка обробляє GET-запити за вказаним шляхом, де *folderName* і *imageName* є змінними шляху. Метод викликає *s3Service.getFile* для отримання зображення з *S3* і повертає його з відповідними заголовками. У випадку помилки повертає статус 404 (Not Found).

Також є ще два методи, для поботи з файлами, але вони вже обробляють POST-запити, а саме для завантаження на сервер зображень, як одного, так і декількох.

```java
@PostMapping("/upload")
public String uploadFile(@RequestParam("file") MultipartFile file) {
    try {
        String key = "uploads/" + System.currentTimeMillis() + "_" + file.getOriginalFilename();
        s3Service.uploadFile(key, file);

        return "File uploaded successfully!";
    } catch (Exception e) {
        e.printStackTrace();
        return "Failed to upload the file.";
    }
}
@PostMapping("/upload-many")
public String handleFileUpload(@RequestParam("productId") int productId,
                               @RequestParam("files") MultipartFile[] files) throws IOException {
    System.out.println(productId);
    s3Service.uploadSomeFiles(productId,files);
    return "Files uploaded successfully!";
}
```

Рисунок 3.26 – Методи з POST-запитами

На цьому зображенні можна побачити обробку файлів методами сервісу, які було створено раніше. А саме, методами *s3Service.uploadFile* та *s3Service.uploadSomeFiles*, які дозволяють завантажувати один та декілька файлів відповідно.

Тим самим був розроблений мікросервіс для завантаження зображень на сервер *S3* та отримання зображень.

3.2.5 RESTful мікросервіси

Мікросервіси, які були побудовані за допомогою використання архітектурного стилю *REST*, можуть називатися *RESTful*. Кожен мікросервіс у системі реалізує *REST API* для взаємодії з іншими мікросервісами і клієнтами, відповідає за певну функціональність і використовує принципи *REST* для комунікації. Основними характеристиками *RESTful* мікросервісів є використання стандартних *HTTP*-методів для взаємодії один з одним та унікальних *URL* для ідентифікації ресурсів. Слід зауважити, що обмін даними між мікросервісами проходить за допомогою стандартних форматів даних таких як *JSON* або *XML*. Також для *RESTful* мікросервісів важливим фактором є безстанність, завдяки якої кожен запит містить всю необхідну інформацію для його обробки, що робить їх менш залежними один від одного.

У рамках цієї роботи також були розроблені RESTful мікросервіси, розробку котрих буде розглянути далі. А саме буде розглянуто один з мікросервісів, який працює у застосунку. Було прийнято рішення більш детально розглянути мікросервіс *product-service*, частину коду роботи котрого було розглянуто раніше, а саме у розділі *3.1* де було розглянуто роботу з базами даних.

Виходячи з цього, для побудування *RESTful* мікросервіса потрібно написати клас-сервіс та клас-контролер з тими *CRUD*-операціями, котрі були реалізовані за допомогою *JpaRepository*.

Взявши до уваги все те, що було сказано раніше, було розроблено клас-сервіс з назвою *ProductService* (рис. 3.26).

```
@Service
public class ProductService {
    9 usages
    private final ProductRepository productRepository;
    public ProductService(ProductRepository productRepository) { this.productRepository = productRepository; }
    1 usage
    public List<Product> getAllProducts() { return productRepository.findAll(); }

    1 usage
    public Product getProductById(Long id) {
        return productRepository.findById(id)
                .orElseThrow(() -> new EntityNotFoundException("Product not found with id: " + id));
    }
    1 usage
    public Product createProduct(Product product) { return productRepository.save(product); }
    1 usage
    public Product updateProduct(Long id, Product updatedProduct) {
        if (!productRepository.existsById(id)) {
            throw new EntityNotFoundException("Product not found with id: " + id);
        }
        updatedProduct.setProductId(id);
        return productRepository.save(updatedProduct);
    }
    1 usage
    public void deleteProduct(Long id) {
        if (!productRepository.existsById(id)) {
            throw new EntityNotFoundException("Product not found with id: " + id);
        }
        productRepository.deleteById(id);
    }
    1 usage
    public List<Product> findAllByProductName(String productName) {
        if (productName == null) {
            throw new IllegalArgumentException("Product name cannot be null");
        }
        return productRepository.findAllByProductName(productName);
    }
}
```

Рисунок 3.26 – Клас ProductService

Клас *ProductService*, який представлено на зображенні, необхідний для організації бізнес-логіки і взаємодії з базою даних у контексті роботи з

продуктами. Він дозволяє інкапсулювати операції над продуктами, що в свою чергу передбачає розміщення логіки, яка визначає правила та поведінку бізнесу, у спеціальні класи або компоненти, окремі від інших частин програми, таких як контролери. Це робить код більш організованим, зручним для тестування і підтримки.

Також цей клас-сервіс забезпечує правильну обробку і передачу помилок на вищі рівні, наприклад до контролера. Зокрема обробляє можливі виключення, такі як *EntityNotFoundException*, для випадків, коли продукт не знайдено.

Наступним кроком є створення класу *ProductController*, який представляє собою *REST*-контролером, який обробляє *HTTP*-запити, пов'язані з продуктами. Взаємодія з котрими виконується через методи класу *ProductService*.

```java
@RestController
@RequestMapping("/products")
public class ProductController {

    7 usages
    private final ProductService productService;

    @Autowired
    public ProductController(ProductService productService) { this.productService = productService; }

    @GetMapping("/")
    public List<Product> getAllProducts() { return productService.getAllProducts(); }

    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(@PathVariable Long id) {
        try {
            Product product = productService.getProductById(id);
            return new ResponseEntity<>(product, HttpStatus.OK);
        } catch (EntityNotFoundException e) {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    }
```

Рисунок 3.27 – Клас ProductController

На рис. 3.27 видно частину коду класу контролера *ProductController* де видно *GET*-методи, один з котрих повертає усі продукти, а інший шукає його за значенням id та повертає, або відповідає статусом 404, якщо продукт не знайдено. Таким же чином реалізовані і інщі *CRUD*-операції, які є у файлі ProductService.

Весь код, що був представлений вище, реалізує *RESTful* мікросервіс який буде працювати у застосунку онлайн-маркетплейсу одягу з товарами, які знаходяться у *БД*. Як і цей мікросервіс, інші також побудовані таким чином.

3.3 Розробка клієнтської частини засобами Spring Framework

При розробці клієнтської частини застосунку було використано Spring Boot та Spring Web для розробки вебзастосунку який повертає HTML сторінки. В свою чергу ці сторінки є динамічними завдяки шаблонізатору Thymeleaf, який використовується у мікросервісі клієнтської частини.

В свою чергу Thymeleaf є серверним шаблонізатором для Java-застосунків, завдяки якому генерується статичний контент такий як HTML, CSS, JS та інші. Він добре підходить для створення динамічних вебсторінок.

Для використання Thymeleaf та створення клієнтського застосунку було використано такі залежності:

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client'
    implementation 'org.springframework.boot:spring-boot-starter-oauth2-client'
    implementation 'org.springframework.security:spring-security-oauth2-resource-server'
    implementation ('mustafaiev.tymur.diploma:model-library')
```

Рисунок 3.28 – Залежності для клієнтської частини

З тих залежностей, що видно на зображені, новою є залежність *'org.springframework.boot:spring-boot-starter-thymeleaf'*, яка як раз таки дає можливість налаштовувати і працювати з *Thymeleaf*.

Для налаштування шаблонізатору та мікросервісу в цілому було використано, як і раніше, файл *application.properties*:

```
server.port=8888

eureka.client.service-url.defaultZone=http://localhost:8761/eureka

logging.pattern.console=%C{1.} [%-5level] %d{HH:mm:ss} - %msg%n

spring.application.name=web-site

spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.suffix=.html
spring.thymeleaf.enabled=true

spring.security.oauth2.resourceserver.jwt.issuer-uri=http://localhost:8180/realms/marketplace
spring.security.oauth2.resourceserver.jwt.jwk-set-uri=http://localhost:8180/realms/marketplace/protocol/openid-connect/certs

spring.datasource.url=jdbc:postgresql://localhost:5432/marketplace
spring.datasource.username=keycloak
spring.datasource.password=keycloak

spring.jpa.hibernate.ddl-auto=update
spring.datasource.driver-class-name=org.postgresql.Driver
```

Рисунок 3.29 – Налаштування для клієнтської частини

На цьому зображені можна побачити налаштування, які раніше вже використовувались для інших мікросервісів. А саме налаштування для роботи з *БД*, *Eureka Server* та *OAuth2*. З цього коду варто виділити налаштування для *Thymeleaf*:

- *spring.thymeleaf.prefix=classpath:/templates/*: це налаштування вказує, де шукати шаблони *Thymeleaf* у вашому проєкі. В даному випадку шаблони будуть шукатися у шляху класу всередині “*src/main/resources/templates/”*;
- *spring.thymeleaf.suffix=.html*: це налаштування визначає розширення файлів шаблонів;
- spring.thymeleaf.enabled=true: вмикає використання Thymeleaf у застосунку.

Після цих налаштувань можна буде використовувати Thymeleaf у HTML файлах у середені яких також можна буде використовувати статичний контент, такий як CSS та JS файли. Статичний контент буде зберігатися у директорії “*src/main/resources/static”,* як видно на зображенні нижче (рис. 3.30).

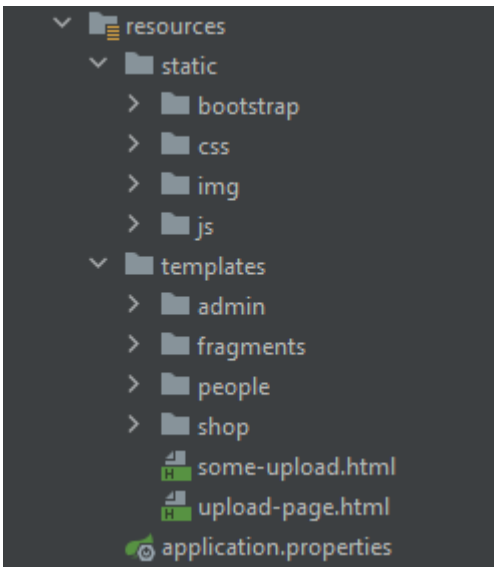


Рисунок 3.30 – Директорії

Виконавши всі ці налаштування у клієнтському застосунку, можна використовувати Thymeleaf та статичний контент у HTML файлах.

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">

<head>
  <meta charset="ISO-8859-1">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0 , maximum-scale=1.0 , user-scalable = 0">
  <link rel="stylesheet" href="" th:href="@{/css/categories.css}"/>
  <link rel="stylesheet" href="" th:href="@{/css/fragments/header.css}"/>
  <link rel="stylesheet" href="" th:href="@{/css/fragments/footer.css}"/>
  <link rel="stylesheet" href="" th:href="@{//code.jquery.com/ui/1.13.2/themes/base/jquery-ui.css}">

  <title>Categories</title>
</head>
<body>
<div class="wrapper">
  <div th:replace="fragments/header.htm :: header"></div>
  <div th:replace="fragments/header.htm :: menubox"></div>
```

Рисунок 3.31 – Приклад використання Thymeleaf ч.1

Цей фрагмент коду демонструє імпортування та використання шаблонізатору у HTML файлі. А саме видно імпортування статичного контенту стилів за ендпоінтами, та вставлення блоків інтерфейсу зі сторонніх .htm файлів які містять блоки коду який повторюється у сторінках. А шаблонізатор дозволяє вносити правки лише в одному файлі, а вони будуть застосовуватись автоматично для файлів у якому є відповідний код.

```
<div class="main">
  <div class="main__container">
      <div class="sales__body">
        <div class="sales__items">
          <div th:if="${error}">
            <p th:text="${error}"></p>
          </div>
          <div class="sales__item" th:each="prod:${products}">
            <div class="item__image">
              <img th:src="@{/image-service/{folderName}/1(folderName=${prod.getProductId()})}" alt="Product Image">
            </div>
            <div class="item__text">
              <div class="item__name" th:text="${prod.getProductName()}">
                Костюм для девочки
              </div>
              <div class="item__price">
                <div class="defoult__price" th:text="${prod.getProductPrice()}">599₽</div>
<!--                <div class="sale__price">719₽</div>-->
              </div>
              <div class="item__buttons">
                <a href="#"><div class="buttons__view">Смотреть</div></a>
                <a href="#">
                  <div class="buttons__basket">В корзину</div>
                  <img class="button__basket-image" th:src="@{/img/products/basket-plus.svg}" alt="">
                </a>
              </div>
```

Рисунок 3.32 – Приклад використання Thymeleaf ч.2

На цьому ж зображенні можно побачити взаємодію шаблонізатору з тими даними, які контролер передає у файл з запитом. А саме з даними товарів з БД, яки представлені у вигляді масиву. Також виконується запит на отримання зображення для цього товару з сервісу по роботи зі сховищем S3.

В свою чергу контролер, який передає ці дані у файл має вигляд на рис. 3.33.

```
@Controller
@RequestMapping("/shop")
public class ShopController {

    2 usages
    private final RestTemplate restTemplate;

    public ShopController(RestTemplate restTemplate) { this.restTemplate = restTemplate; }

    @GetMapping("/products")
    public String getAllProducts(Model model) {
        ResponseEntity<List<Product>> response = restTemplate.exchange(
                url: "http://localhost:8765/products/",
                HttpMethod.GET,
                requestEntity: null,
                new ParameterizedTypeReference<>() {}
        );
        if (response.getStatusCode().is2xxSuccessful()) {
            List<Product> products = response.getBody();
            model.addAttribute( attributeName: "products", products);
        } else {
            model.addAttribute( attributeName: "error", attributeValue: "Could not retrieve products");
        }
        return "shop/categories";
    }
```

Рисунок 3.33 – Контроллер ShopController

Цей фрагмент коду демонструє те, як передаються дані продуктів, до вебсторінки. А саме те, як вони вилучаються з сервісу, який працює з даними продуктів, по ендпоінту *GET*-запиту. Розглянемо більш детальніше цей код:

- *public String getAllProducts(Model model)*: метод повертає рядок, який є назвою сторінки, що буде відображено. Параметр *Model model* використовується для передачі даних з контролера до шаблону *Thymeleaf*;
- *ResponseEntity<List<Product>> response = restTemplate.exchange(...)*: restTemplate використовується для виконання *HTTP* запиту, а метод exchange виконує запит типу *GET* до *URL*;
- *if(response.getStatusCode().is2xxSuccessful())* перевіряє, чи був запит успішним.
- *List<Product> products = response.getBody()*: витягує список продуктів з відповіді;
- *model.addAttribute("products", products)*: додає список продуктів до моделі, щоб передати їх на сторінку;
- *model.addAttribute("error", "Could not retrieve products")*: якщо відповідь не є успішною, додається атрибут з ключем "error" і повідомленням про помилку до моделі;
- *return "shop/categories"*: повертає назву шаблону, який буде використаний для відображення даних.

Виконавши усі налаштування та написавши потрібний для реалізації технологій, використаних у цьому мікросервісі. Було розроблено клієнтський сервіс засобами Spring Framework.

3.4 Функціонал застосунку

Застосунок, розроблений у рамках цієї роботи реалізує увесь запланований функціонал, потрібний для онлайн-маркетплейсу. Цей функціонал дозволяє об’єднати та використовувати мікросервіси, які були розроблені, у користувацькому інтерфейсі.

При переході до застосунку, користувача зустрічає головна сторінка (рис. 3.34), яка представляє собою шапку сторінки, яка містить певні посилання на інші сторінки сайту та логотип. Також сторінка має в собі карусель з інформацією та посиланнями на каталог з певними товарами.

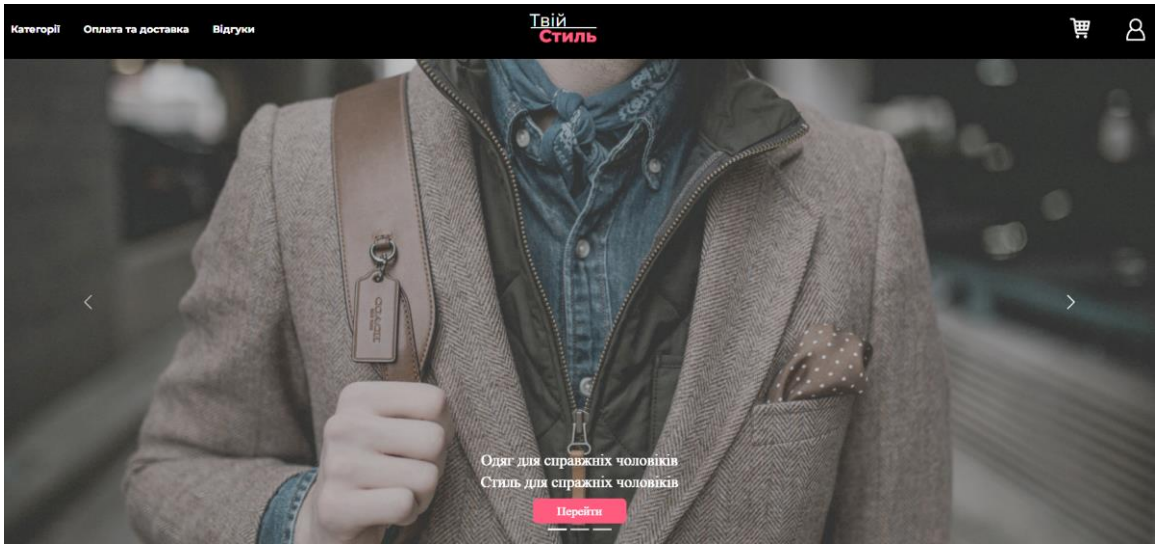


Рисунок 3.34 – Головна сторінка застосунку

Користувач може переходити на потрібні йому розділи застосунку, але деякі з них захищені та потребують певних прав доступу. А незареєстровані користувачі мають доступ до обмеженого списку сторінок. Однією з тих, до яких він має доступ, є сторінка з товарами (рис. 3.35).

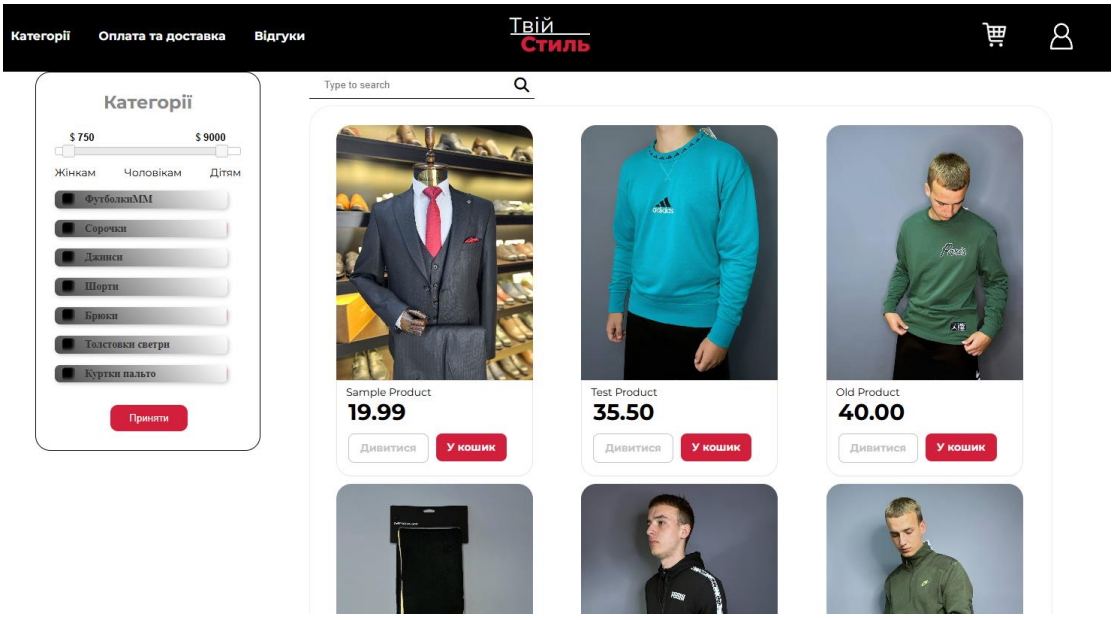


Рисунок 3.35 – Сторінка каталогу товарів

Незареєстрований користувач може передивлятись, фільтрувати та шукати товари. Також він може переходити на сторінку вибраного товару (рис. 3.36).

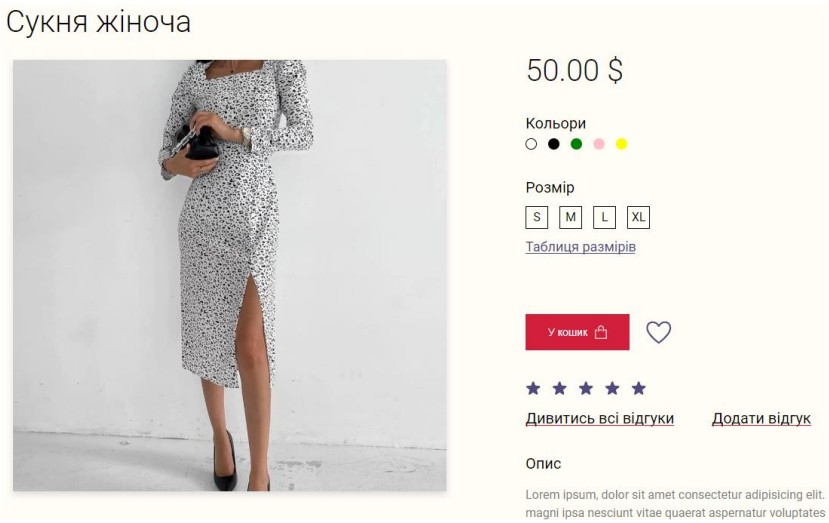


Рисунок 3.36 – Сторінка товару

Ця сторінка подає інформацію та зображення вибраного товару. Як і у попередніх випадках, функціонал цієї сторінки є обмеженим для незареєстрованих користувачів. Обмеження цієї сторінки полягає у тому, що з функцій додавання у кошик, додавання відгуку та перегляду всіх відгуків продавця цього товару, цей користувач має доступ лише до останнього (рис. 3.37).

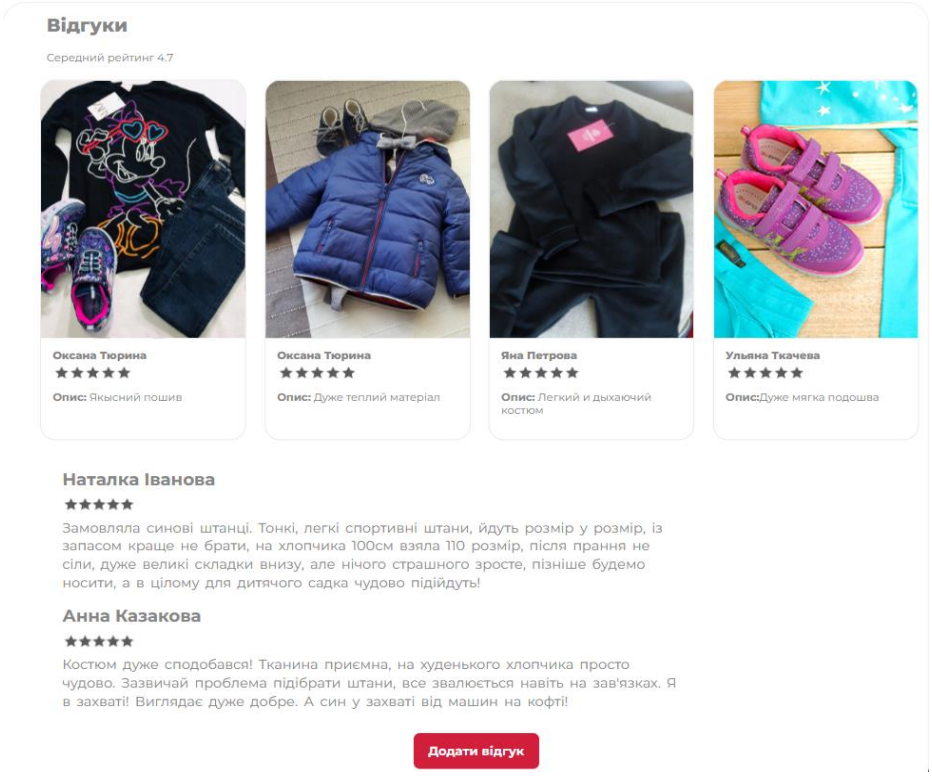


Рисунок 3.37 – Сторінка відгуків

Зареєстровані користувачі можуть надавати відгуки як про товари, так і про продавців. Якщо незареєстрований користувач схоче перейти на захищену сторінку або отримати доступ до певного функціоналу, то його перенаправить до сторінки входу *Keycloak* (рис. 3.38).

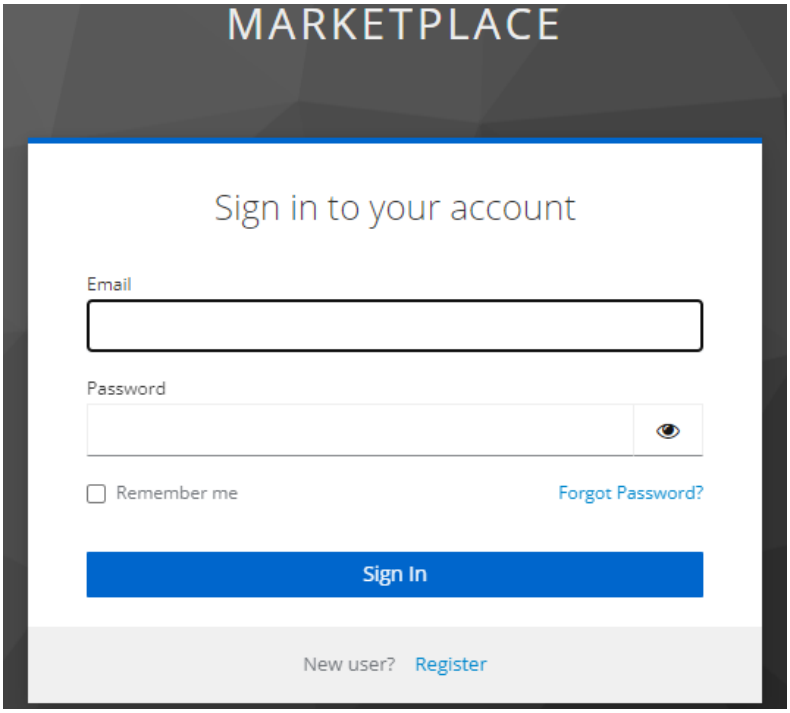


Рисунок 3.38 – Сторінка входу

Ця форма входу до застосунку містить два поля для введення даних: пошта користувача та пароль. Також ця форма надає можливості відновлення пароля та збереження пошти користувача для зручного повторного входу.

Крім того, є посилання на форму реєстрації користувача для випадків, коли користувач, який хоче отримати доступ до ресурсу, не має аккаунту у застосунку.

На рис. 3.39 показано форму реєстрації, на яку потрапляє користувач, переходячи за посиланням зі сторінки входу. Ця форма містить поля: пошта, пароль, повтор пароля, ім’я та прізвище. Всі поля форми є обов’язковими для створення облікового запису. Ще однією можливістю є повернення до сторінки входу.

Виконавши автентифікацію і авторизацію у застосунку, користувач отримує доступ до сторінок, відповідно за наданими правами, закріпленими за його акаунтом. Деякі з цих сторінок буде розглянуто далі.

На відміну від інших функцій, функція додавання товарів на рис. 3.40 потребує не лише автентифікації, а й авторизації у ролі продавця. Тобто користувач у ролі покупця не зможе отримати доступ до цієї сторінки та функціоналу.

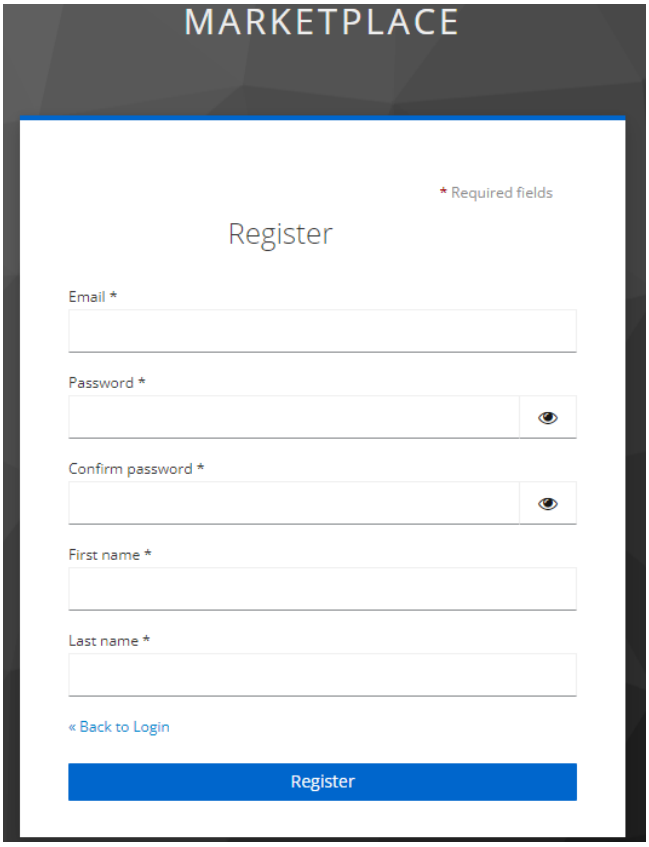


Рисунок 3.39 – Сторінка реєстрації

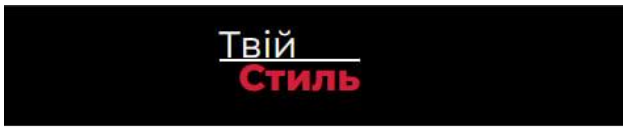


Рисунок 3.40 – Форма створення товару

Сама форма містить всі необхідні поля для створення нового продукту. Такими полями є назва товару, категорії, опису та статі. Після заповнення полів та відправки форми, користувача перенаправляє до сторінки додавання зображень товару (рис. 3.41).

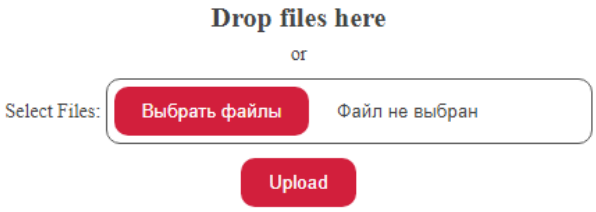


Рисунок 3.41 – Форма додання зображень

Завдяки цій формі додавання зображень для створеного товару, продавець завершує створення картки товару, який буде продаватись на платформі онлайн-маркетплейсу одягу.

Також є ще сторінка для користувача авторизованого з роллю адміністратора (рис. 3.42).

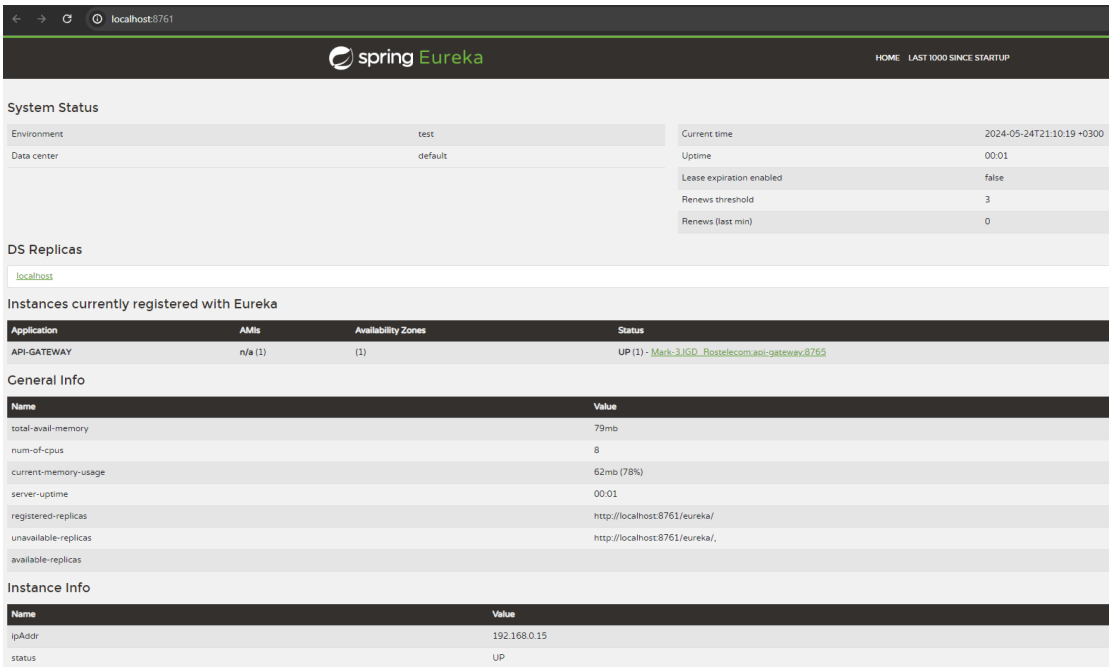


Рисунок 3.42 – Сторінка адміністративної консолі Eureka Server

Ця сторінка дозволяє моніторити запущені мікросервіси, а саме передивлятись стан системи, головну інформацію та інформацію про екземпляри.

Відтак, можна з впевненістю сказати, що реалізованого функціоналу цілком достатньо для онлайн-маркетплейсу одягу.

3.5 Тестування та оцінка якості застосунку

Для тестування програмного застосунку онлайн-маркетплейсу одягу були створені тест-кейси, завдяки яким було перевірено функціонал додатку. Ці тести формують сценарії взаємодії з програмою, містять набір умов, на основі котрих виконується тестування.

Для мануального тестування застосунку було розроблено такі тест-кейси:

1) Перевірка роботи реєстрації у застосунку:

- відправка коректних даних для реєстрації:

Таблиця 3.1 – test case 1

Кроки	Очікуваний результат
1. Зайти на сторінку з реєстрацією 2. Відправити форму з коректними даними нового користувача	У базі даних з’явиться новий користувач, користувачеві буде видано токен доступу

- відправка даних для реєстрації існуючого користувача:

Таблиця 3.2 – test case 2

Кроки	Очікуваний результат
1. Зайти на сторінку з реєстрацією 2. Відправити форму з вказанням Email, що вже зареєстрований	З’явиться попередження про те, що введений Email вже використовується

- відправка не повної форми з даними для реєстрації:

Таблиця 3.3 – test case 3

Кроки	Очікуваний результат
1. Зайти на сторінку з реєстрацією 2. Відправити форму не вказавши Email або пароль користувача	З’явиться попередження про відсутність того чи іншого елемента у відправленій формі

2) Перевірка авторизації та автентифікації:

- відправка коректних даних для авторизації:

Таблиця 3.4 – test case 4

Кроки	Очікуваний результат
1. Зайти на сторінку з авторизацією 2. Ввести коректні дані	Користувач отримає токен доступу, та дозволи за роллю

- відправка некоректних даних для авторизації:

Таблиця 3.5 – test case 5

Кроки	Очікуваний результат
1. Зайти на сторінку з авторизацією 2. Ввести некоректні дані логіну	З’явиться попередження про неправильно введені дані, токен доступу не буде видано користувачеві

- відправка не повних даних для авторизації:

Таблиця 3.6 – test case 6

Кроки	Очікуваний результат
1. Зайти на сторінку з авторизацією 2. Відправити форму не вказавши логін або пароль користувача	З’явиться попередження про відсутність того чи іншого елемента у відправленій формі, токен доступу не буде видано користувачеві

3) Перевірка пошуку товарів за назвою:

- відправка назви товарів які є у БД для пошуку:

Таблиця 3.7 – test case 7

Кроки	Очікуваний результат
1. Ввести у форму пошуку назву товару який міститься у БД	Перенаправить на сторінку з результатами пошуку

- відправка назви товарів яких немає у БД для пошуку:

Таблиця 3.8 – test case 8

Кроки	Очікуваний результат
1. Ввести у форму пошуку назву товару який не міститься у БД	Перенаправить на сторінку з повідомленням що товар не було знайдено

4) Перевірка додавання товару до кошику:

Таблиця 3.9 – test case 9

Кроки	Очікуваний результат
1. Натиснути кнопку додавання до кошику товару	Товар буде додано до кошику

5) Перевірка додавання відгуку до товару:

- відправка повних даних відгуку про товар:

Таблиця 3.10 – test case 10

Кроки	Очікуваний результат
1. Зайти на сторінку товару 2. Відправити форму вказавши коректні дані	Відгук буде збережено

- відправка не повних даних відгуку про товар:

Таблиця 3.11 – test case 11

Кроки	Очікуваний результат
1. Зайти на сторінку товару 2. Відправити форму не вказавши рейтинг або рецензію про товар	З'явиться попередження про відсутність того чи іншого елемента у відправленій формі

6) Перевірка додавання відгуку до продавця:

- відправка повних даних відгуку про продавця:

Таблиця 3.12 – test case 12

Кроки	Очікуваний результат
1. Зайти на сторінку товару 2. Відправити форму вказавши коректні дані	Відгук буде збережено

- відправка не повних даних відгуку про продавця:

Таблиця 3.13 – test case 13

Кроки	Очікуваний результат
1. Зайти на сторінку товару 2. Відправити форму не вказавши рейтинг або рецензію про товар	З'явиться попередження про відсутність того чи іншого елемента у відправленій формі

На основі цих тестів проведено мануальне тестування розробленого застосунку онлайн-маркетплейсу одягу, сформований чек-ліст (табл. 3.14) та баг-репорт (табл. 3.15).

Таблиця 3.14 – Чек-ліст тестів

ID	NAME	Status
1	Відправка коректних даних для реєстрації	PASSED
2	Відправка даних для реєстрації існуючого користувача	PASSED
3	Відправка не повної форми з даними для реєстрації	PASSED
4	Відправка коректних даних для авторизації	PASSED
5	Відправка некоректних даних для авторизації	PASSED
6	Відправка не повних даних для авторизації	PASSED
7	Відправка назви товарів які є у БД для пошуку	PASSED
8	Відправка назви товарів яких немає у БД для пошуку	FAILED
9	Перевірка додавання товару до кошику	PASSED
10	Відправка повних даних відгуку про товар	PASSED
11	Відправка не повних даних відгуку про товар	PASSED
12	Відправка повних даних відгуку про продавця	PASSED
13	Відправка не повних даних відгуку про продавця	PASSED

Таблиця 3.15 – Баг-репорт

Назва багу	Неможливо додати відгук про продавця з неповними даними.
Опис	При спробі відправити форму відгука про продавця без вказання рейтингу або коментаря система не дозволяє виконати дію, проте не надає користувачу відповідного повідомлення про помилку.
Кроки	1. Зайти на сторінку товару 2. Перейти до розділу відгуків про продавця 3. Відправити форму відгука, не вказавши рейтинг або коментар
Фактичний результат	Форма не відправляється, проте користувач не отримує інформацію про причину невдачі.
Очікуваний результат	Система повинна відобразити повідомлення про помилку, яке вказуватиме на необхідність заповнення обов'язкових полів.
Пріоритет	Середній – ця проблема може призвести до негативного досвіду користувача і негативного враження від продукту.

Проведено ручне тестування основних функціональних модулів онлайн-маркетплейсу одягу, який було розроблено у рамках цієї роботи. Це допомогло створити чек-ліст для систематизації процесу перевірки та виявити недоліки. Також підготовлено звіт про виявлені несправності, що дозволило виправити виявлені в ході тестування проблеми.

3.5 Висновки до розділу 3

У розділі детально проаналізовано структуру та функціональні можливості створюваного програмного забезпечення з використанням Spring framework, як основного інструмента розробки. Зокрема, увага приділяється опису модулів, підпрограм та блоків, що формують основу продукту. Реалізації концепції онлайн-маркеплейсу виконується через технічні компоненти застосунку, кожен з яких відповідає певній функціональності.

Розроблено застосунок реалізує мікросервісну архітектуру за допомогою інструментів, які містить в собі Spring. Мікросервісну архутектуру застосунку поділено на декілька частин, кожна з яких виконуватиме своє завдання.

Для поділу програми на складові частини мікросервісного застосунку використано технології: Spring Cloud, Spring Boot, Spring Data та інші. Це дозволило створити сучасний застосунок, перевагами якого є масштабованість, модульність та безпека, що є важливими аспектами сучасного застосунку. Всі

переваги цього застосунку реалізовано за допомогою потужних інструментів фреймворка Spring.

Розроблено 6 тестових випадків та 13 тест-кейсів. На основі цих тестів проведено мануальне тестування розробленого застосунку онлайн-маркетплейсу одягу, сформований чек-ліст та баг-репорт. Проведене тестування застосунку дозволило його покращити та виправити помилки, допущені під час розробки програмного продукту. Тобто тести, виконані над застосунком, позитивно вплинули на якість кінцевого продукту.

ВИСНОВКИ

Розроблений у роботі програмний продукт онлайн-маркетплейсу одягу надає змогу взаємодіяти продавцям одягу та покупцям як зручний майданчик для купівлі-продажу. Такий складний проєк було реалізовано завдяки використання потужних засобів фреймворку Spring.

У першому розділі проведено аналіз предметної області. У рамках цього розглянуто специфіку розробки онлайн маркетплейсу, що дало розуміння того, як потрібно вибудовувати логіку роботи подібного майданчика. Наступним кроком був розгляд ймовірних конкурентів та аналогів, виявлення їх переваг і недоліків. Логічним завершенням цього було створення SWOT-аналізу конкурентів, що дало змогу розглянути конкурентів більш детально. Аналіз ринку та ймовірних конкурентів дав зрозуміти актуальність розробки майданчику для інтернет-комерції, зокрема онлайн-маркетплейсу. Крім того, у першому розділі розглянуто вимоги до функціоналу онлайн-маркетплейсу одягу, разом з якими було виявлено варіанти використання цього програмного продукту. Засобами розробки для backend, frontend та бази даних застосунку вибрано:

– backend: Spring Cloud, Spring Boot, Spring Web;
– frontend: Spring Boot, Spring Web, Thymeleaf, HTML, CSS, JS, Bootstrap;
– база даних: PostgreSQL, Spring Data JPA.

Важливим аспектом, який було розглянуто у роботі, є маркетингова стратегія, що дозволить просувати та розвивати програмний продукт. При цьому розглянуто його просування через соціальні мережи.

У другому розділі розглянуто архітектурні стилі, які були використані у розробці, а саме: мікросервісна архітектура, MVC та REST API, переваги яких дозволили розобити сучасний вебзастосунок. Під час моделювання програмної системи реалізовано численні UML-діаграми, завдяки яким легше розробляти конкретний функціонал та таблиці БД. Пов’язано це з тим, що використовуючи їх у подальшій розробці застосунку, легше визначити те, як має виглядати БД чи які функції має забезпечувати той чи інший мікросервіс.

У третьому розділі описано етапи розробки і написання програмного коду, який реалізує весь функціонал онлайн-маркетплейсу для продажу одягу. Розроблено численні запити до бази даних відповідно до функціоналу застосунку. Серверна частина застосунку онлайн-маркетплейсу одягу написана з використанням мікросервісної архітектури, реалізованої засобами фреймворка Spring. Серверна частина застосунку онлайн-маркетплейсу одягу написана з використанням мікросервісної архітектури, реалізованої засобами фреймворку Spring. В роботі виконано мануальне тестування розробленого застосунку онлайн-маркетплейсу одягу, сформований чек-ліст та баг-репорт. Проведене тестування застосунку дозволило його покращити та виправити помилки, допущені під час розробки програмного продукту. Тобто тести, виконані над застосунком, позитивно вплинули на якість кінцевого продукту.

Результатом роботи є якісний розроблений програмний продукт – сучасний онлайн-маркетплейс одягу. Головною особливістю та водночас перевагою його є використання для розробки фреймворка Spring.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. How to Build a Marketplace Website: The Definitive Guide. URL: https://smarttek.solutions/blog/how-to-build-a-marketplace/

2. Bashir A. Marketplace Business Model. URL: https://www.linkedin.com/pulse/marketplace-business-model-bashir-ahmed

3. How to Build a Marketplace Website in 15 Steps: Ultimate Guide. URL: https://www.codica.com/blog/how-to-build-online-marketplace/

4. Paratsii T. How to build an online marketplace: steps, features, and cost. URL: https://leobit.com/blog/how-to-build-an-online-marketplace-steps-features-and-cost/

5. Трофименко О.Г., Пастернак Ю.Ю., Манаков С.Ю., Лобода Ю.Г. Автоматизація тестування вебсайтів електронної комерції. *Сучасна спеціальна техніка.* 2021. № 2(65). С. 46-59. DOI: https://doi.org/10.36486/mst2411–3816.2021.2(65).5.

6. Мустафаєв Т.З. Специфіка веброзробки онлайн-маркетплейсу. *Інформаційне суспільство: проблеми та перспективи* : матер. IX всеукр. наук.-практ. конф. (м. Одеса, 24 травня 2024 р.). Одеса, 2024. (депоновано).

7. Shopping locations that shoppers worldwide prefer to shop at for first-time and repeat purchases as of July 2019. URL: https://www.statista.com/statistics/897678/online-shopper-preference-for-first-time-repeat-purchases-platform-global/

8. Global Amazon Prime Day sales by small and medium sized businesses selling on Amazon from 2018 to 2022. URL: https://www.statista.com/statistics/1029873/annual-amazon-prime-day-sales/

9. Spring Framework, History, and Its Structure. URL: https://dev.to/mrjeandev/spring-framework-history-and-its-structure-361

10. Quiña-Mera A., Landeta D., Rea-Peñafiel M., Guevara-Vega C. Quality Evaluation of a Spring Cloud Microservices Architecture Implementation. 2022. DOI: 10.1007/978-981-16-4126-8_66.

11. Understanding the Fundamentals of PostgreSQL Architecture URL: https://instaclustr.medium.com/understanding-the-fundamentals-of-postgresql-architecture-98ce3672376f

12. Spring Data JPA. URL: https://medium.com/@aprayush20/spring-data-jpa-with-spring-boot-9e3b261e4cc7

13. Top 7 Challenges in eCommerce Website Development & How to Overcome Them. URL: https://smarttek.solutions/blog/challenges-in-ecommerce-development/

14. What is SMM? And why is it crucial for your company? URL: https://sites.google.com/view/socialmediamarketplace/home

15. Jailia M., Kumar A., Agarwal M., Sinha I. Behavior of MVC (Model View Controller) based Web Application developed in PHP and .NET framework. International Conference on ICT in Business Industry & Government (ICTBIG'16). 2016. P. 1-5. DOI:10.1109/ICTBIG.2016.7892651. URL: https://ieeexplore.ieee.org/document/7892651

16. What is a REST API? URL: https://www.ibm.com/topics/rest-apis

17. What is Client-Server Architecture? Everything You Should Know. URL: https://www.simplilearn.com/what-is-client-server-architecture-article

18. Scaling Microservices for Optimal Performance: Strategies and Best Practices. URL: https://www.atmosly.com/blog/scaling-microservices-for-optimal-performance

19. Introduction to Spring Data URL: https://hyperskill.org/learn/step/16299

20. What is Docker? URL: https://www.oracle.com/ua/cloud/cloud-native/container-registry/what-is-docker/

21. Docker Compose overview. URL: https://docs.docker.com/compose/

22. The Resource Server URL: https://www.oauth.com/oauth2-servers/the-resource-server/

23. The OAuth 2.0 Authorization Framework URL: https://datatracker.ietf.org/doc/html/rfc6749#section-1.1

24. Grant Types URL: https://developers.miniorange.com/docs/oauth-joomla/grant-types

25. OAuth Scopes URL: https://oauth.net/2/scope/

ДОДАТКИ

Додаток А. Фрагменти програмного коду

1. api-gateway:

1) application.properties:

```
server.port=8765

spring.main.web-application-type=reactive

logging.level.org.springframework=INFO
logging.level.org.springframework.security=DEBUG

spring.application.name=api-gateway

eureka.client.service-url.defaultZone=http://localhost:8761/eureka

logging.pattern.console=%C{1.} [%-5level] %d{HH:mm:ss} - %msg%n

spring.cloud.gateway.discovery.locator.enabled=true

spring.cloud.gateway.discovery.locator.lower-case-service-id=true

logging.level.org.springframework.cloud.gateway=DEBUG

spring.cloud.gateway.default-filters[0]=TokenRelay

#WEB-SITE
spring.cloud.gateway.routes[0].id=static-resources
spring.cloud.gateway.routes[0].uri=http://localhost:8888
spring.cloud.gateway.routes[0].predicates[0]=Path=/css/**, /img/**, /js/**,
/bootstrap/**, /shop/**

# /account/**
spring.cloud.gateway.routes[1].id=account
spring.cloud.gateway.routes[1].uri=http://localhost:8888
spring.cloud.gateway.routes[1].predicates[0]=Path=/account/**

# /image/**
spring.cloud.gateway.routes[2].id=image
spring.cloud.gateway.routes[2].uri=http://localhost:8888
spring.cloud.gateway.routes[2].predicates[0]=Path=/image/**

#storage-amazon-s3
spring.cloud.gateway.routes[3].id=storage
spring.cloud.gateway.routes[3].uri=lb://storage-amazon
spring.cloud.gateway.routes[3].predicates[0]=Path=/image-service/**

spring.cloud.gateway.routes[4].id=order
spring.cloud.gateway.routes[4].uri=lb://order-service
spring.cloud.gateway.routes[4].predicates[0]=Path=/order/**

spring.cloud.gateway.routes[5].id=basket
spring.cloud.gateway.routes[5].uri=lb://basket-service
spring.cloud.gateway.routes[5].predicates[0]=Path=/basket/**

spring.cloud.gateway.routes[6].id=review
spring.cloud.gateway.routes[6].uri=lb://review-service
spring.cloud.gateway.routes[6].predicates[0]=Path=/review/**
```

```
spring.cloud.gateway.routes[7].id=category
spring.cloud.gateway.routes[7].uri=lb://category-service
spring.cloud.gateway.routes[7].predicates[0]=Path=/category/**

spring.cloud.gateway.routes[8].id=product
spring.cloud.gateway.routes[8].uri=lb://product-service
spring.cloud.gateway.routes[8].predicates[0]=Path=/product/**


#Keycloak
spring.security.oauth2.resourceserver.jwt.jwk-set-
uri=http://localhost:8180/realms/marketplace/protocol/openid-connect/certs
spring.security.oauth2.resourceserver.jwt.issuer-
uri=http://localhost:8180/realms/marketplace

spring.security.oauth2.client.provider.keycloak.issuer-
uri=http://localhost:8180/realms/marketplace
spring.security.oauth2.client.registration.marketplace.provider=keycloak
spring.security.oauth2.client.registration.marketplace.client-id=api-gateway-
client
spring.security.oauth2.client.registration.marketplace.client-
secret=nfSYIcLgBJU9HNur15emzFbZGdRtQuxD
spring.security.oauth2.client.registration.marketplace.authorization-grant-
type=authorization_code
spring.security.oauth2.client.registration.marketplace.redirect-
uri={baseUrl}/login/oauth2/code/{registrationId}
spring.security.oauth2.client.registration.marketplace.scope=openid
```

2) SecurityConfig.java:

```
@Configuration
@EnableWebFluxSecurity
public class SecurityConfig {

    @Bean
    public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity
http) {
        http.authorizeExchange(auth -> auth
                        .pathMatchers("/shop/*").permitAll()
                        .anyExchange().authenticated())
                .oauth2Login( Customizer.withDefaults())
                .oauth2ResourceServer((oauth2) ->
oauth2.jwt(Customizer.withDefaults()));
        http.csrf(ServerHttpSecurity.CsrfSpec::disable);
        return http.build();
    }
}
```

2. eureka-server:

1) application.properties:

```
server.port=8761

spring.application.name=eserver

eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
eureka.client.service-url.defaultZone=http://localhost:8761/eureka

logging.level.com.netflix.eureka=OFF
logging.level.com.netflix.discovery=OFF

logging.pattern.console=%C{1.} [%-5level] %d{HH:mm:ss} - %msg%n
```

2) EurekaServerApplication:

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }
}
```

3. product-service:

1) application.properties:

```
spring.application.name=product-service

server.port=0

eureka.client.service-url.defaultZone=http://localhost:8761/eureka

logging.pattern.console=%C{1.} [%-5level] %d{HH:mm:ss} - %msg%n


# DB connect settings
spring.datasource.url=jdbc:postgresql://localhost:5432/marketplace
spring.datasource.username=keycloak
spring.datasource.password=keycloak
```

```
# JPA settings
spring.jpa.hibernate.ddl-auto=update
spring.datasource.driver-class-name=org.postgresql.Driver

spring.security.oauth2.resourceserver.jwt.issuer-
uri=http://localhost:8180/realms/marketplace
spring.security.oauth2.resourceserver.jwt.jwk-set-
uri=http://localhost:8180/realms/marketplace/protocol/openid-connect/certs
```

2) ProductRepository.java:

```
package mustafaiev.tymur.diploma.productservice.repository;

import mustafaiev.tymur.diploma.modellibrary.model.Product;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;

public interface ProductRepository extends JpaRepository<Product, Long> {
    List<Product> findAllByProductName(String productName);
}
```

3) ProductService.java:

```
@Service
public class ProductService {
    private final ProductRepository productRepository;
    public ProductService(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }
    public List<Product> getAllProducts() {
        return productRepository.findAll();
    }

    public Product getProductById(Long id) {
        return productRepository.findById(id)
                .orElseThrow(() -> new EntityNotFoundException("Product not found
with id: " + id));
    }
    public Product createProduct(Product product) {
        return productRepository.save(product);
    }
    public Product updateProduct(Long id, Product updatedProduct) {
        if (!productRepository.existsById(id)) {
            throw new EntityNotFoundException("Product not found with id: " + id);
```

```
        }
        updatedProduct.setProductId(id);
        return productRepository.save(updatedProduct);
    }
    public void deleteProduct(Long id) {
        if (!productRepository.existsById(id)) {
            throw new EntityNotFoundException("Product not found with id: " + id);
        }
        productRepository.deleteById(id);
    }
    public List<Product> findAllByProductName(String productName) {
        if (productName == null) {
            throw new IllegalArgumentException("Product name cannot be null");
        }
        return productRepository.findAllByProductName(productName);
    }
}
```

4) ProductController.java:

```
@RestController
@RequestMapping("/products")
public class ProductController {

    private final ProductService productService;

    @Autowired
    public ProductController(ProductService productService) {
        this.productService = productService;
    }

    @GetMapping(value = "/", produces = "application/json")
    public List<Product> getAllProducts() {
        return productService.getAllProducts();
    }

    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(@PathVariable Long id) {
        try {
            Product product = productService.getProductById(id);
            return new ResponseEntity<>(product, HttpStatus.OK);
        } catch (EntityNotFoundException e) {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
```

```
    }
    @PostMapping("/")
    public Product createProduct(@RequestBody Product product) {
        return productService.createProduct(product);
    }

    @PutMapping("/{id}")
    public ResponseEntity<Product> updateProduct(@PathVariable Long id,
@RequestBody Product updatedProduct) {
        try {
            Product product = productService.updateProduct(id, updatedProduct);
            return new ResponseEntity<>(product, HttpStatus.OK);
        } catch (EntityNotFoundException e) {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteProduct(@PathVariable Long id) {
        try {
            productService.deleteProduct(id);
            return new ResponseEntity<>(HttpStatus.NO_CONTENT);
        } catch (EntityNotFoundException e) {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    }

    @GetMapping("/byName")
    public ResponseEntity<List<Product>> findProductsByName(@RequestParam(name =
"name") String productName) {
        try {
            List<Product> products =
productService.findAllByProductName(productName);
            return new ResponseEntity<>(products, HttpStatus.OK);
        } catch (IllegalArgumentException e) {
            return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
        }
    }
}
```

5) SecurityConfig.java:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests(authorize -> authorize
                        .requestMatchers("/products/**").permitAll()
                        .anyRequest().authenticated())
                .oauth2ResourceServer((oauth2) ->
oauth2.jwt(Customizer.withDefaults()));
        return http.build();
    }
}
```

4. storage-amazon-s3:

1) AwsConfig.java:

```
@Configuration
public class AwsConfig {
    @Value("${aws.accessKeyId}")
    private String accessKeyId;

    @Value("${aws.secretKey}")
    private String secretAccessKey;

    @Bean
    public S3Client s3Client() {
        return S3Client.builder()
                .region(Region.EU_CENTRAL_1)
                .credentialsProvider(StaticCredentialsProvider
                        .create(AwsBasicCredentials
                                .create(accessKeyId, secretAccessKey)))
                .build();
    }
}
```

2) S3Service.java:

```
@Service
public class  S3Service {
    private final S3Client amazonS3Client;

    @Value("${aws.s3.bucket-name}")
    private String bucketName;

    public S3Service(S3Client amazonS3Client) {
        this.amazonS3Client = amazonS3Client;
    }

    public void uploadFile(String key, MultipartFile file) throws IOException {
        amazonS3Client.putObject(PutObjectRequest.builder()
                        .bucket(bucketName)
                        .key(key)
                        .build(),
                RequestBody.fromBytes(file.getBytes()));
    }

    public void uploadSomeFiles(int id, MultipartFile[] files) throws IOException
{
        String key;
        int num=1;
        for (MultipartFile file : files) {
            key = "id_" + id + "/" + num+".png";
            amazonS3Client.putObject(PutObjectRequest.builder()
                            .bucket(bucketName)
                            .key(key)
                            .build(),
                    RequestBody.fromBytes(file.getBytes()));
            num++;
        }
    }
    public byte[] getFile(String folderName, String fileName) throws S3Exception {
        GetObjectRequest getObjectRequest = GetObjectRequest.builder()
                .bucket(bucketName)
                .key("id_" + folderName + "/" + fileName + ".png")
                .build();
        ResponseBytes<GetObjectResponse> getObjectResponse =
amazonS3Client.getObjectAsBytes(getObjectRequest);
        return getObjectResponse.asByteArray();
      }
}
```

3) ImageController.java:

```
@RestController
@RequestMapping("/image-service")
public class ImageController {
    private final S3Service s3Service;
    public ImageController(S3Service s3Service) {
        this.s3Service = s3Service;
    }
    @GetMapping("/{folderName}/{imageName}")
    public ResponseEntity<byte[]> getImage(@PathVariable String folderName,
                                           @PathVariable String imageName) {
        try {
            byte[] data = s3Service.getFile(folderName, imageName);

            HttpHeaders headers = new HttpHeaders();
            headers.setContentType(MediaType.IMAGE_JPEG);

            return new ResponseEntity<>(data, headers, HttpStatus.OK);
        } catch (S3Exception e) {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    }
    @PostMapping("/upload")
    public String uploadFile(@RequestParam("file") MultipartFile file) {
        try {
            String key = "uploads/" + System.currentTimeMillis() + "_" +
file.getOriginalFilename();
            s3Service.uploadFile(key, file);

            return "File uploaded successfully!";
        } catch (Exception e) {
            e.printStackTrace();
            return "Failed to upload the file.";
        }
    }
    @PostMapping("/upload-many")
    public String handleFileUpload(@RequestParam("productId") int productId,
                                   @RequestParam("files") MultipartFile[] files)
throws IOException {
        System.out.println(productId);
        s3Service.uploadSomeFiles(productId,files);
        return "Files uploaded successfully!";}}
```

5. web-site:

1) AppConfig.java:

```
package mustafaiev.tymur.diploma.website.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@Configuration
public class AppConfig {
    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

2) ShopController.java:

```
@Controller
@RequestMapping("/shop")
public class ShopController {

    private final RestTemplate restTemplate;

    public ShopController(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    @GetMapping("/products")
    public String getAllProducts(Model model) {
        ResponseEntity<List<Product>> response = restTemplate.exchange(
                "http://localhost:8765/products/",
                HttpMethod.GET,
                null,
                new ParameterizedTypeReference<>() {}
        );
        if (response.getStatusCode().is2xxSuccessful()) {
            List<Product> products = response.getBody();
            model.addAttribute("products", products);
        } else {
            model.addAttribute("error", "Could not retrieve products");
        }
        return "shop/categories";
```

```
    }

    @GetMapping("/main")
    public       String       mainPage(@RequestHeader("Authorization")       String
authorizationHeader){
        if (authorizationHeader != null && authorizationHeader.startsWith("Bearer
")) {
            String token = authorizationHeader.substring(7);
            System.out.println("Token: " + token);
        } else {
            System.out.println("Токен не найден");
        }

        return "shop/main";}

    @GetMapping("/payInfo")
    public String payInfoPage(){return "shop/pay";}

    @GetMapping("/test")
    public String testPage(){return "shop/testPage";}

    @GetMapping("/about")
    public String aboutPage(){return "shop/about";}

    @GetMapping("/reviews")
    public String reviewsPage(){return "shop/reviews";}

}
```