# БАКАЛАВРСЬКА РОБОТА

БР.КІ–07.00.00.000 ПЗ

Група КІ-21-1

Дем’яник Назар

2025

Івано-Франківський Національний Технічний Університет Нафти і Газу
Факультет Інформаційних Технологій
Кафедра комп'ютерних систем і мереж

**Дем'яник Назар Сергійович**

УДК 004.04

# БАКАЛАВРСЬКА РОБОТА

## Розробка web-сервісу для інтернет магазину косметики засобами MySQL,Java та фреймворку Spring

Комп'ютерна інженерія
(назва освітньої програми)

123 - Комп'ютерна інженерія
(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня *Дем'яник Н. С.*
(підпис, прізвище та ініціали здобувача)

Науковий керівник *Бузоверя Надія Геннадіївна, доцент*
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

**Допущено до захисту**

Завідувач кафедри КСМ

к.т.н., професор *Мельничук С. І.*
(посада) (підпис) (дата) (прізвище та ініціали)

**м. Івано-Франківськ – 2025**

**Івано-Франківський Національний Технічний Університет Нафти і Газу**

Інститут *Інформаційних Технологій*

Кафедра *Комп'ютерних систем і мереж*

Освітній рівень *бакалавр*

Спеціальність *123 Комп'ютерна інженерія*

ЗАТВЕРДЖУЮ**:**

Зав. кафедрою КСМ

С.І. Мельничук

*« 05 » травня 2025*року

# З А В Д А Н Н Я

## НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

*Дем'янику Назару Сергійовичу*

(прізвище, ім'я, по батькові)

1. Тема роботи *Розробка web-сервісу для інтернет магазину косметики засобами MySQL,Java та фреймворку Spring* .

керівник роботи *Бузоверя Надія Геннадіївна, доцент*

затверджені наказом закладу вищої освіти від " *05* " *травня 2025 року № 275/7*

2. Термін подання студентом роботи *12 червня 2025 року*

3. Вихідні дані до роботи *Методичні вказівки, технічна література*

4. Зміст пояснювальної записки: *1 Опрацювання предметної області, розгляд наявних прикладів створення схожих сайтів, постановка задачі. 2 Обґрунтування вибору програмного забезпечення для розробки сайту, розробка діаграм взаємодії з користувачем, проєктування основних компонентів вебзастосунку. 3 Розробка структури таблиць бази даних, розробка функцій для обробки даних і створення шляхів для адресації по сторінкам сайту*

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

_____

6. Консультанти розділів роботи

| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | Завдання видав | Завдання прийняв |
| | | | |
| | | | |
| | | | |

7. Дата видачі завдання – 29 січня 2025 року

## КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів бакалаврської роботи | Термін виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | *Підготовка та вибір літератури для написання бакалаврської роботи* | *Березень 2025р* | *виконав* |
| 2 | *Опрацювання предметної області та аналіз аналогів вебдодатків* | *Березень 2025р* | *виконав* |
| 3 | *Розробка структури бази даних, створення діаграм* | *Квітень 2025р* | *виконав* |
| 4 | *Розробка основних компонентів вебдодатку та перевірка працездатності* | *Травень 2025р* | *виконав* |
| 5 | *Оформлення пояснювальної записки* | *Травень 2025р* | *виконав* |
| 6 | *Підготовка до здачі бакалаврської роботи* | *Червень 2025р* | *виконав* |

**Студент** _____ *Дем'яник Н. С.*

**Керівник роботи** _____ *Бузоверя Н.Г.*

# АНОТАЦІЯ

Основним завданням цієї бакалаврської роботи є створення повнофункціонального вебзастосунку для представлення, фільтрації та адміністрування товарів у каталозі інтернет-магазину. Розроблений інтрнет магазин забезпечує зручний перегляд продукції, реєстрацію та авторизацію користувачів, можливість детального ознайомлення з товарами, а також управління асортиментом з боку адміністратора.

У межах реалізації проєкту виконано повний цикл розробки: спроєктовано та реалізовано структуру бази даних (MySQL), розроблено серверну частину з використанням Java, Spring Framework та Hibernate для взаємодії з базою даних. Також реалізовано клієнтську частину (фронтенд) з використанням JavaScript та Tailwind CSS, що забезпечує сучасний адаптивний інтерфейс. Система включає базові механізми безпеки для реєстрації, авторизації та захисту даних користувачів.

Результатом роботи є функціональний вебсайт, який може бути використаний як основа для реального інтернет-магазину з подальшою можливістю масштабування і розширення функціональності.

Ключові слова: ІНФОРМАЦІЙНА СИСТЕМА, ВЕБЗАСТОСУНОК, БАЗА ДАНИХ, JAVA, SPRING, HIBERNATE, JAVASCRIPT, TAILWIND CSS, MYSQL.

# ANNOTATION

The main objective of this bachelor's thesis is the development of a fully functional web application for presenting, filtering, and managing products in an online store catalog. The developed online store provides convenient product browsing, user registration and authentication, detailed product views, as well as product management capabilities for administrators.

Within the scope of the project, a complete development cycle was carried out: the database structure (MySQL) was designed and implemented; the backend was developed using Java, Spring Framework, and Hibernate for interaction with the database. The frontend was built using JavaScript and Tailwind CSS, providing a modern and responsive user interface. The system also includes basic security mechanisms for user registration, authentication, and data protection.

The result of this work is a functional website that can serve as a foundation for a real e-commerce platform with the potential for future scaling and feature expansion.

Keywords: INFORMATION SYSTEM, WEB APPLICATION, DATABASE, JAVA, SPRING, HIBERNATE, JAVASCRIPT, TAILWIND CSS, MYSQL.

**ЗМІСТ**

# ВСТУП

**Актуальність обраної теми.** Сучасна електронна комерція стрімко розвивається, і потреба у якісних, функціональних вебзастосунках для продажу товарів онлайн зростає щодня. Покупці очікують зручної, швидкої та безпечної взаємодії з платформами, які надають можливість перегляду, пошуку, фільтрації та придбання продукції. У зв'язку з цим розробка повнофункціонального вебзастосунку для інтернет-магазину є актуальною прикладною задачею, що дозволяє забезпечити зручний сервіс як для користувачів, так і для адміністраторів платформи. Вебзастосунок, створений у межах даної роботи, дозволяє ефективно керувати каталогом товарів, що особливо важливо у сфері роздрібної торгівлі, включно з ринком косметики.

**Об'єкт дослідження.** Об'єктом дослідження є процес розробки сучасного вебзастосунку для інтернет-магазину.

**Предмет дослідження.** Предметом дослідження є методи та засоби створення інтерактивних клієнт-серверних систем з використанням сучасного стеку технологій..

**Мета та завдання роботи.** Метою даної бакалаврської роботи є створення повноцінного вебзастосунку, що забезпечує функціональну взаємодію між користувачами та адміністраторами інтернет-магазину. Вебзастосунок включає інтерфейс для перегляду та фільтрації товарів, можливості авторизації та реєстрації, а також інструменти для адміністрування асортименту. Для досягнення мети необхідно виконати такі завдання:

1. Спроєктувати та реалізувати структуру бази даних

2. Розробити серверну частину із використанням мови програмування Java та фреймворків Spring та Hibernate,

3. Створити клієнтську частину з використанням JavaScript і Tailwind CSS

4. Реалізувати базові механізмів безпеки, захисту даних користувачів та налагодженню ефективної взаємодії між фронтендом і серверною логікою.

**Методи дослідження.** У ході дослідження було застосовано аналіз

літературних джерел, що охоплював вивчення наукових праць, технічної документації та практичних посібників з розробки веб-додатків із використанням Java, Spring Framework і MySQL. З метою побудови структури програмного забезпечення використовувалося функціональне моделювання, яке дозволило визначити основні модулі та їхню взаємодію. Було застосовано метод об'єктно-орієнтованого проєктування для структуризації коду відповідно до принципів ООП. Також використовувалися методи структурного аналізу для опису логіки взаємодії користувача з системою та обробки даних. Моделювання бази даних дозволило побудувати оптимальну реляційну структуру інформаційного сховища.

**Практична цінність.** Практична цінність роботи полягає у створенні повнофункціонального web-сервісу для інтернет-магазину косметики, який може бути безпосередньо використаний у комерційній діяльності. Реалізоване рішення є готовою платформою, що дозволяє швидко масштабувати функціонал відповідно до потреб бізнесу.

# 1 ОГЛЯД АНАЛОГІВ ІНТЕРНЕТ-МАГАЗИНІВ КОСМЕТИЧНОЇ ПРОДУКЦІЇ ТА ПОСТАНОВКА ЗАДАЧІ

## 1.1 Опрацювання предметної області

Інтернет-магазин – це вебзастосунок, що забезпечує можливість користувачам переглядати товари, застосовувати фільтри, додавати продукти до кошика, оформлювати замовлення, а адміністраторам – керувати товарами, категоріями, замовленнями, клієнтами тощо. Він забезпечує дистанційний продаж товарів або послуг через мережу Інтернет . Такі системи є невід'ємною частиною сучасної електронної комерції та активно розвиваються протягом останніх двох десятиліть.

У сучасному світі онлайн-покупки стали основною формою придбання товарів у багатьох галузях – від побутової техніки до косметики. Ефективність функціонування інтернет-магазину значною мірою залежить від його зручності, швидкодії, надійності та гнучкості в розширенні функціоналу. З огляду на це, створення якісного вебзастосунку вимагає використання сучасних технологій для бекенд- і фронтенд-розробки, належного структурування коду та дотримання норм розробки баз даних.

У процесі розробки проєкту буде використано мову програмування Java, яка є однією з найпопулярніших та надійних мов для серверної розробки. Для побудови серверної частини буде застосовано Spring Framework, що забезпечить масштабовану архітектуру, зручну ін'єкцію залежностей, підтримку REST API та високий рівень безпеки .

Для взаємодії з базою даних планується використання Hibernate – ORM-бібліотеки, яка дозволить працювати з таблицями як з Java-об'єктами, що значно спростить написання SQL-запитів . В якості системи управління базами даних буде обрано MySQL, яка забезпечить стабільність, швидкодію та простоту адміністрування.

Фронтенд-частина буде реалізована з використанням JavaScript та Tailwind CSS – сучасного утилітарного CSS-фреймворку, який забезпечить швидке стилізування компонентів та адаптивність інтерфейсу. Це дозволить створити зручний та інтуїтивно зрозумілий інтерфейс для користувачів.

Структура бази даних буде спроєктована з урахуванням нормалізації до третьої нормальної форми (3НФ), що допоможе уникнути дублювання даних, підвищити цілісність та полегшити подальшу підтримку. Всі сутності – користувачі, товари, замовлення, категорії, атрибути, бренди тощо – будуть розміщені в окремих таблицях із коректно налагодженими зв'язками «один до багатьох» та «багато до багатьох», де це буде необхідно.

У структурі системи буде реалізовано CRUD-операції (Create, Read, Update, Delete) для всіх основних сутностей. Це дозволить адміністраторам гнучко керувати даними магазину – додавати нові товари, редагувати існуючі, переглядати замовлення, видаляти застарілі позиції тощо.

Інтерфейс користувача буде розроблено таким чином, щоб забезпечити швидкий доступ до функцій магазину: зручний перегляд товарів, застосування фільтрів (за брендом, ціною, категорією, наявністю, атрибутами), додавання до кошика, оформлення замовлення.

Також буде реалізовано механізми реєстрації, авторизації та особистого кабінету, де користувач зможе переглядати історію покупок, змінювати особисті дані, відстежувати статуси замовлень тощо. Для забезпечення захисту даних буде використано JWT (JSON Web Token) для авторизації, а обмін запитами з сервером буде здійснюватися через захищене API.

Інтерфейс адміністратора надаватиме доступ до управління всіма аспектами вебзастосунку – перегляд і редагування бази товарів, управління замовленнями, користувачами, брендами, категоріями, атрибутами.

Проєкт буде структуровано за принципами MVC (Model-View-Controller). Моделі описуватимуть сутності та їх взаємозв'язки, контролери оброблятимуть HTTP-запити та викликатимуть відповідні сервіси, а сервіси реалізовуватимуть бізнес-логіку. Такий підхід забезпечить масштабованість проєкту, легкість

тестування та зручність у підтримці. Маршрути будуть побудовані логічно: /products, /categories, /login, /admin/products, що забезпечить зрозумілу навігацію як для користувачів, так і для розробників.

### 1.2 Обґрунтування вибору програмного забезпечення

Для розробки вебзастосунку онлайн-магазину косметики планується використати сучасні, перевірені часом інструменти та технології, що забезпечать високу продуктивність, гнучкість, масштабованість і безпеку. Вибір стеку технологій ґрунтуватиметься на їх актуальності, активній підтримці спільноти, сумісності між собою та відповідності вимогам до функціональності системи.

Основною мовою програмування, якою буде реалізовано вебзастосунок, стане Java – універсальний інструмент, що дозволяє створювати надійні серверні рішення корпоративного рівня. Завдяки платформонезалежності, високій продуктивності та безпеці, Java стане оптимальним вибором для обробки персональних і фінансових даних користувачів.

Java є однією з найпопулярніших мов програмування, яка активно використовується в корпоративних, веб- та мобільних застосунках. Вона забезпечує платформонезалежність завдяки використанню віртуальної машини Java Virtual Machine (JVM), високу продуктивність і безпеку, що критично важливо для вебзастосунків, пов’язаних з обробкою персональних та фінансових даних користувачів.

Java має розвинену екосистему бібліотек і фреймворків, які дозволяють реалізовувати як серверну, так і клієнтську логіку вебдодатків. Завдяки суворій типізації та об'єктно-орієнтованому підходу, Java дозволяє створювати масштабовані та підтримувані рішення.

Для реалізації серверної частини проєкту буде використано фреймворк Spring, зокрема його модуль Spring Boot, що значно спростить конфігурацію та розгортання. Spring дозволить ефективно впроваджувати REST-архітектуру, забезпечувати безпеку, обробку форм і взаємодію з базами даних.

Переваги Spring:

- Інверсія управління (IoC), що забезпечує легке масштабування та тестування коду.

- Модульність: окремі компоненти можна легко замінювати або відключати.

- Підтримка транзакцій, безпеки (Spring Security), інтеграції з іншими фреймворками.

- Швидкий запуск завдяки Spring Boot Starter залежностям [4].

Для об’єктно-реляційного відображення (ORM) буде застосовано Hibernate – потужну бібліотеку, яка забезпечить роботу з реляційною базою даних на рівні об’єктів. Це дозволить зменшити кількість ручного SQL-коду, оптимізувати взаємодію з даними та мінімізувати помилки.

Hibernate:

- Забезпечує зручне керування сутностями через анотації або XML-конфігурації.

- Підтримує зв’язки один-до-багатьох, багато-до-багатьох.

- Автоматично генерує SQL-запити відповідно до опису моделей.

- Оптимізує доступ до даних через кешування та лениву ініціалізацію [5].

У якості системи керування базами даних буде обрано MySQL – продуктивне та надійне рішення, що забезпечить ефективне зберігання інформації про товари, користувачів, замовлення й фільтри. MySQL підтримує складні SQL-запити, транзакції та зовнішні ключі, що важливо для побудови складної структури даних.

Переваги MySQL:

- Висока продуктивність при великій кількості одночасних запитів.

- Зручна робота з таблицями та зв’язками між ними.

- Підтримка реплікації, бекапів та масштабування.

- Інтеграція з Hibernate та Spring Data JPA [6].

Інтерфейс користувача планується створити за допомогою Tailwind CSS – утилітарного CSS-фреймворку, який дозволить швидко й ефективно стилізувати сторінки, забезпечивши сучасний вигляд та адаптивність.

Tailwind CSS:

- Гнучкість і повний контроль над зовнішнім виглядом елементів.

- Мінімізація кастомного CSS.

- Висока продуктивність та швидкість розробки.

- Простота адаптації для мобільних пристроїв [7].

JavaScript буде використано для реалізації інтерактивних функцій: фільтрації товарів, обробки AJAX-запитів та реалізації нескінченного скролу.

Для динамічного відображення даних з боку сервера буде задіяно шаблонізатор Thymeleaf, який інтегрується зі Spring Boot та дозволить зручно формувати HTML-шаблони з динамічними даними.

Його переваги:

- Повна інтеграція з Spring MVC.

- Підтримка циклів, умов, вставок даних з об’єктів.

- Можливість попереднього перегляду шаблонів у браузері навіть без сервера.

- Легка читабельність HTML-шаблонів навіть для нефахівців [8].

Захист персональних даних користувачів, обмеження доступу до адміністративної панелі та реалізація аутентифікації/авторизації буде забезпечена за допомогою Spring Security – потужного та гнучкого інструменту безпеки у світі Java.

Функціональність Spring Security:

- Аутентифікація користувачів (вхід за логіном і паролем).

- Авторизація з розмежуванням прав доступу (користувач, адміністратор).

- Захист від CSRF-атак, XSS тощо.

- Можливість розширення за допомогою JWT, OAuth 2.0 тощо [9].

Розгортання й тестування вебзастосунку буде виконуватись на локальному сервері Apache Tomcat, який є сумісним із Spring Boot і широко використовується для запуску Java-застосунків.

Переваги Tomcat:

- Просте налаштування та використання на локальному середовищі.

- Підтримка сервлетів та JSP.

- Сумісність з усіма Java-застосунками.

- Гнучка система конфігурації.

- Можливість інтеграції з середовищем IntelliJ IDEA [10].

Tomcat дозволяє ефективно протестувати роботу вебдодатку локально перед його перенесенням на хостинг або сервер.

Усю розробку планується виконувати у середовищі IntelliJ IDEA – зручному та потужному IDE, що забезпечує інтеграцію зі Spring, Hibernate, Git та інструментами для роботи з базами даних. Це дозволить швидко писати, тестувати та налагоджувати код.

Переваги IntelliJ IDEA:

- Автоматичне доповнення коду та підсвітка синтаксису.

- Глибока інтеграція з фреймворками Spring, Hibernate, Thymeleaf.

- Вбудовані інструменти для запуску Tomcat і роботи з базами даних.

- Вбудований контроль версій (Git, GitHub).

- Зручний налагоджувач (debugger) та профілювальник продуктивності [11].

Обраний стек технологій дозволяє створити повноцінний, надійний, сучасний та масштабований вебзастосунок для онлайн-магазину косметики. Кожен компонент має чітке призначення в архітектурі проєкту, забезпечує гнучкість у розробці, подальшу підтримку та розвиток системи. Завдяки використанню сучасного середовища розробки та локального сервера, розробка буде проходити швидко та зручно.

### 1.3 Розгляд наявних прикладів інтернет-магазинів

У межах розробки вебзастосунку для інтернет-магазину косметики доцільно проаналізувати наявні приклади подібних систем, щоб виділити їх сильні та слабкі сторони, а також врахувати ці аспекти під час створення власного продукту.

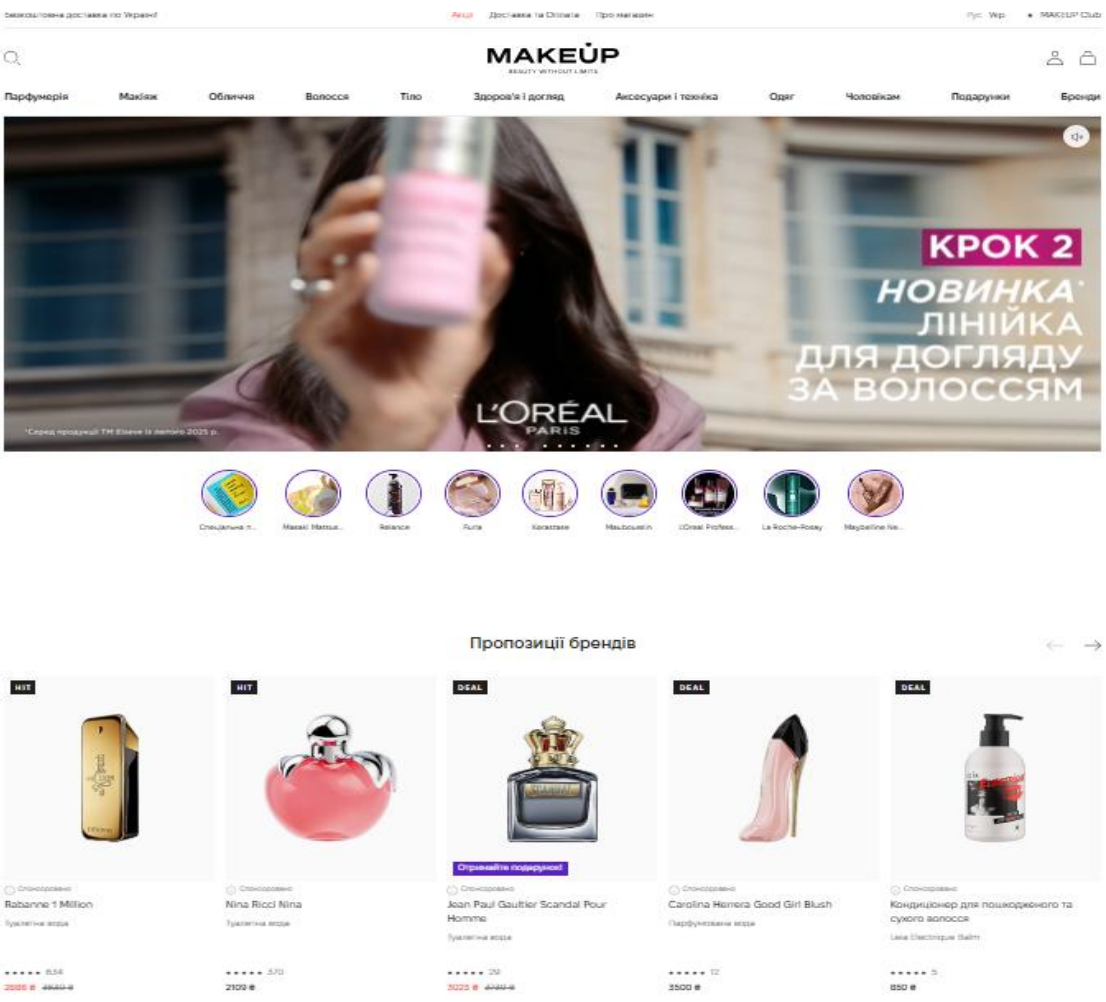


Рисунок 1.1 – Головна сторінка сайту MakeUp.ua

MakeUp.ua — один з найбільших українських інтернет-магазинів косметики та парфумерії. Головна сторінка сайту є інформативною та зручною для користувача [1]. Вона містить банери з акціями, популярні категорії товарів, а також блок із рекомендованими продуктами.

Особливу увагу варто звернути на розміщення фільтрів і швидкий доступ до популярних розділів — це позитивно впливає на користувацький досвід. Дизайн сторінки адаптовано до мобільних пристроїв, що є обов'язковою умовою для сучасного інтернет-магазину. У нашій системі також доцільно передбачити акційні банери, блок із найпопулярнішими категоріями та секцію рекомендаці

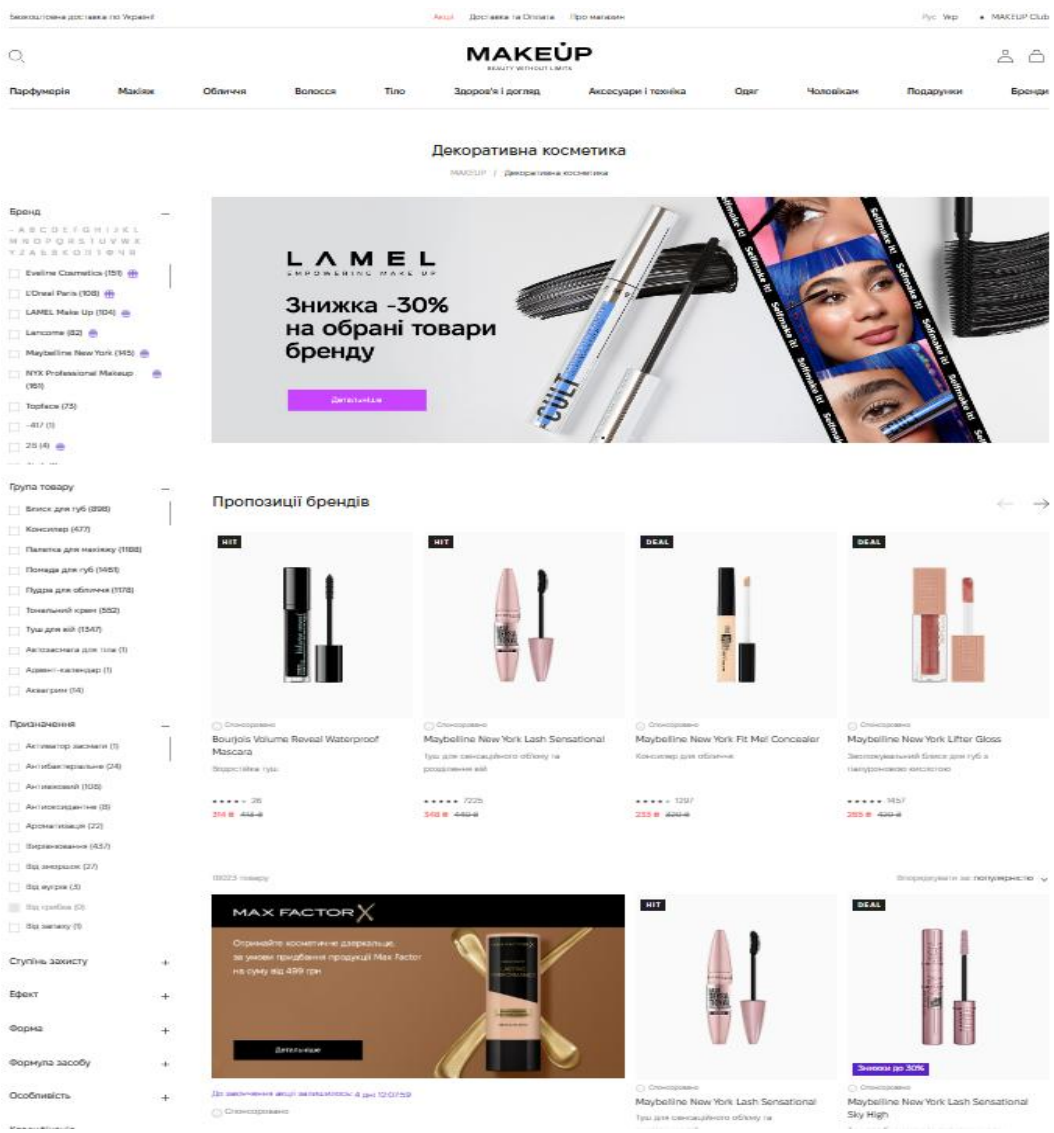


Рисунок 1.2 – Сторінка категорії товарів сайту MakeUp.ua

На сторінці категорії представлено список товарів з можливістю фільтрації за брендом, ціною, наявністю, а також специфічними характеристиками (наприклад, тип шкіри, країна виробництва, об’єм тощо).

Фільтри мають інтуїтивно зрозумілий інтерфейс, їх можна розгорнути або згорнути, що забезпечує зручність навігації. Інформація про кількість товарів за кожним фільтром допомагає користувачу швидко зорієнтуватися.

Важливо впровадити аналогічну багаторівневу фільтрацію, а також функцію «нескінченного скролу» або пагінації для зручного перегляду великої кількості товарів.

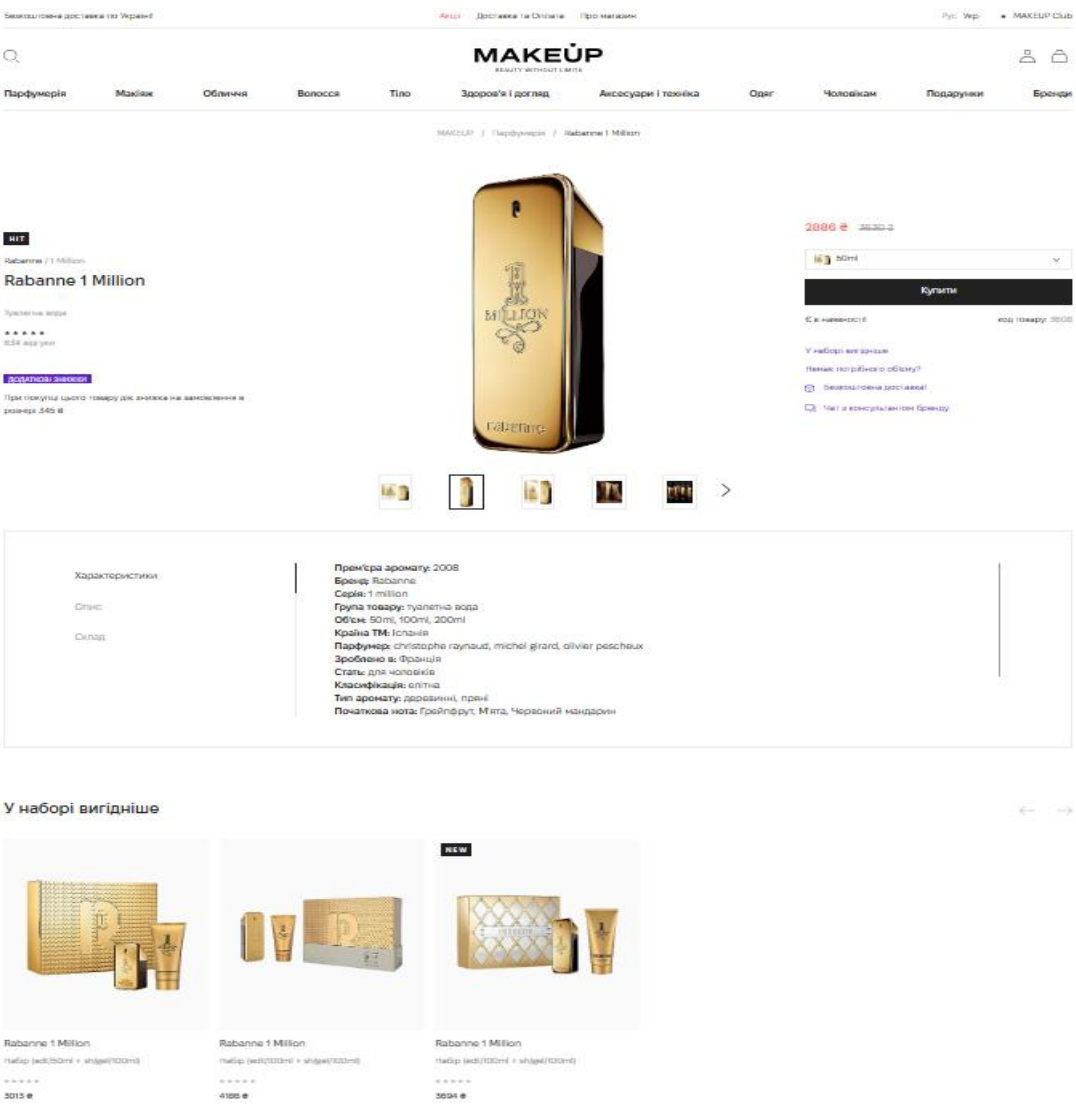


Рисунок 1.3 – Сторінка окремого товару на сайту MakeUp.ua

На сторінці окремого товару детально представлена вся необхідна інформація: назва продукту, бренд, ціна, наявність, опис, інгредієнти, відгуки покупців, а також супутні товари. Присутній зручний інтерфейс для додавання товару до кошика.

При розробці власної системи варто реалізувати структуру сторінки, яка дозволить зручно переглядати як текстову інформацію, так і медіаконтент.

Розглянемо ще один приклад вебсайту з продажу косметики — Parfums.ua.

Аналіз кількох аналогічних вебресурсів дозволяє краще зрозуміти підходи до реалізації інтерфейсів користувача, структури каталогів, функціональності пошуку та презентації товарів. Це необхідно для того, щоб виявити найкращі

практики та врахувати можливі недоліки під час розробки власної інформаційної системи онлайн-магазину косметики.

У той час як сайт MakeUp.ua демонструє повноцінну класичну структуру з розширеним каталогом, фільтрами та деталізацією сторінок, сайт Parfums.ua використовує інші UX-рішення, орієнтовані на швидкість взаємодії з продуктами та зменшення кліків для користувача [2].

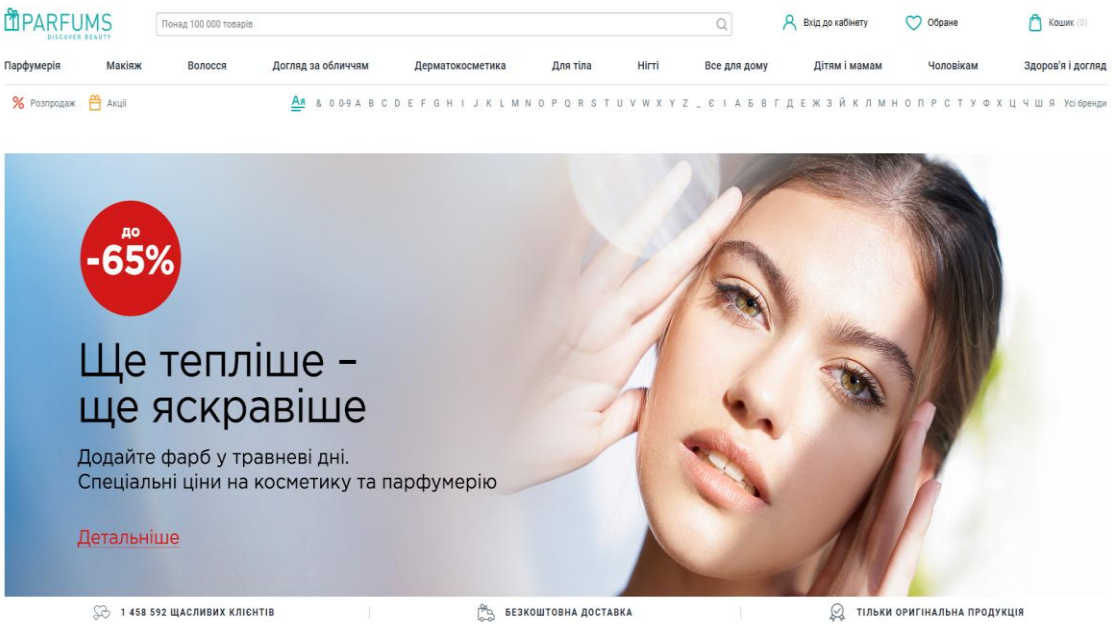


Рисунок 1.4 – Сторінка пошуку товарів на сайті Parfums.ua

На сторінці пошуку користувач бачить інтегрований рядок для введення запиту, а також швидкі підказки за категоріями, брендами або популярними запитами. Важливим моментом є функція автоматичних підказок (автозаповнення), яка з’являється ще до натискання Enter.

Такий підхід значно скорочує час на пошук необхідного товару, а також допомагає зорієнтуватися новим користувачам. У нашій майбутній системі варто врахувати цей UX-підхід як приклад інтелектуального пошуку з динамічними підказками.

Крім сторінки пошуку, інші типові сторінки сайту Parfums.ua – такі як каталоги за категоріями чи окремі сторінки товару – мають доволі класичну структуру, подібну до MakeUp.ua. У категоріях також реалізовано фільтрацію за брендами, ціною, наявністю тощо.

Сторінка товару містить основну інформацію про продукт, опис, характеристики, а також можливі модифікації – наприклад, вибір об’єму (500 мл, 100 мл), форми випуску чи упаковки. Важливо, що для одного і того ж товару доступні різні варіації, що реалізовано через варіативні опції прямо на сторінці, а не окремими товарами. Такий підхід зручний для користувача і дозволяє зменшити дублювання в базі даних, зберігаючи логіку «один товар – кілька варіантів».

У порівнянні з MakeUp.ua, який має більш глибоку деталізацію інтерфейсу, сайт Parfums.ua орієнтується на швидкість і спрощення доступу до основних дій користувача — пошуку, перегляду товару, додавання в кошик.

Окрім українських прикладів, для кращого розуміння сучасних підходів до розробки вебзастосунків у сфері електронної комерції також було проаналізовано міжнародний сайт Sephora.com. Цей ресурс є прикладом високоякісного користувацького інтерфейсу, гнучкої системи фільтрації та персоналізованого досвіду користувача. На головній сторінці акцент зроблено на рекомендації, промо-банери та актуальні пропозиції. Крім того, реалізовано динамічне оновлення фільтрів на сторінках категорій та розширені можливості на сторінках товарів – від вибору модифікації до інтеграції відео й фотоогляді від користувачів.

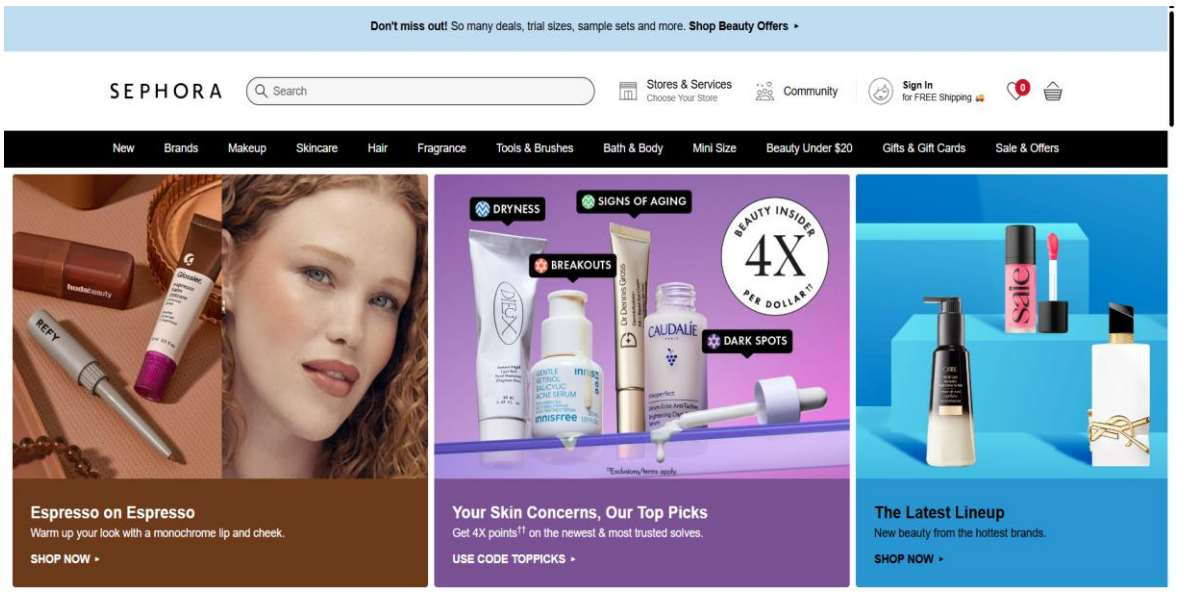


Рисунок 1.5 – Головна сторінка Sephora.com

**Таблиця 1.1 – табличка порівняння переваг та недоліків розглянутих аналогів**

| Назва сайту | Переваги | Недоліки |
|---|---|---|
| MakeUp.ua | – Інтуїтивно зрозумілий інтерфейс<br>– Чітко структурований каталог<br>– Потужна система фільтрів (бренд, ціна, характеристики)<br>– Адаптивний дизайн<br>– Вивід рекомендованих товарів | – Відсутність варіацій товарів в межах однієї сторінки<br>– Дублювання варіантів одного товару як окремих позицій |
| Parfums.ua | – Простий і швидкий доступ до основних функцій<br>– Автоматичні підказки при пошуку<br>– Варіативні опції товарів (об'єм, упаковка тощо)<br>– Зменшення кліків для користувача | – Менш гнучка система фільтрації у порівнянні з MakeUp.ua<br>– Менше контенту на сторінках товарів |
| Sephora.com | – Динамічна фільтрація під категорії<br>– Рекомендації та персоналізація<br>– Інтеграція відео/фотооглядів<br>– Модульність сторінок товарів | – Зайва складність для деяких користувачів<br>– Висока ресурсоємність інтерфейсу (може працювати повільно на слабких пристроях) |

На основі аналізу існуючих аналогів вебзастосунків для продажу косметики – MakeUp.ua, Parfums.ua та Sephora.com – було зроблено низку висновків, які напряму впливають на подальший процес розробки власного рішення.

Вебзастосунок, який ми реалізовуватимемо, має забезпечити зручну взаємодію користувача з каталогом товарів, а також надати гнучкий механізм фільтрації за категоріями, брендами, ціною, наявністю та характеристиками товарів. Як показує приклад Sephora, ефективною є реалізація динамічної фільтрації, яка підлаштовується під вибрану категорію, а також адаптивний інтерфейс, що враховує попередні дії користувача (наприклад, рекомендації чи історію переглядів).

Система буде включати сторінку з детальним описом кожного товару, де окрему увагу буде приділено:можливості вибору модифікацій товару (об'єм, відтінок тощо), характеристикам, інформації про наявність.

Каталог товарів буде реалізовано зі сторінкою категорій, яка включатиме всі ключові фільтри, виведення товарів за нескінченним скролом та сучасний дизайн, адаптований до мобільних пристроїв.

Всі таблиці бази даних будуть створені з урахуванням нормалізації та вимог до цілісності даних – жодне критичне поле (наприклад, назва товару, ціна, кількість, категорія) не зможе залишатись порожнім. Це забезпечить цілісність даних і зручність обробки.

Адміністратори сайту матимуть можливість створювати, редагувати або видаляти товари, категорії, бренди та фільтри, а користувачі – здійснювати пошук, фільтрацію, перегляд і додавання товарів до кошика.

### 1.4 Постановка задачі

Потрібно розробити вебзастосунок для онлайн-магазину косметики, що забезпечить користувачам зручний інтерфейс для перегляду товарів, фільтрації за характеристиками, оформлення замовлень, а також адміністраторам – повноцінне управління асортиментом та структурою магазину.

Розробка охоплює створення повноцінного клієнт-серверного застосунку, де серверна частина буде реалізована мовою програмування Java з використанням фреймворку Spring (Spring Boot, Spring Data, Spring Security тощо). Це дозволить реалізувати надійний бекенд із чітким поділом логіки, безпечну роботу з базою даних, гнучку маршрутизацію та авторизацію користувачів.

Основні завдання:

Проєктування архітектури вебзастосунку з урахуванням розділення на клієнтську і серверну частини, дотриманням принципів REST-архітектури.

Розробка структури бази даних:

- Визначення основних сутностей системи: товари, категорії, бренди, атрибути товарів, користувачі, замовлення, кошик тощо.

- Нормалізація таблиць, встановлення зв’язків між ними (один-до-багатьох, багато-до-багатьох).

- Забезпечення цілісності даних за допомогою обов’язкових полів (NOT NULL), зовнішніх ключів та відповідних обмежень.

- Наповнення бази тестовими даними для перевірки функціональності.

Розробка серверної логіки (бекенду):

- Реалізація CRUD-операцій (створення, читання, оновлення, видалення) для всіх основних сутностей.

- Створення окремих сервісів (Service layer), які відповідатимуть за бізнес-логіку взаємодії з базою.

- Розробка контролерів (Controller layer), які обробляють HTTP-запити, передають дані до сервісів та повертають результат клієнту.

- Налаштування маршрутів (Routes) для логічного переходу між сторінками застосунку (наприклад, /products, /categories, /admin/products/edit, тощо).

- Створення окремих запитів для фільтрації товарів за брендом, ціною, характеристиками, наявністю.

Реалізація системи авторизації та аутентифікації:

- Розмежування прав доступу (звичайний користувач, адміністратор).

- Використання JWT або іншої технології безпечного входу до системи.

- Захист доступу до адміністративних функцій.

Розробка клієнтської частини (фронтенду):

- Проєктування шаблонів сторінок на основі HTML, CSS (використання Tailwind CSS), JavaScript.

- Реалізація логіки взаємодії з бекендом за допомогою AJAX-запитів (fetch/axios).

- Динамічне оновлення сторінок (без перезавантаження) при фільтрації, зміні кількості товарів у кошику тощо.

- Реалізація адаптивного дизайну для мобільних пристроїв.

Реалізація функціональності з боку користувача:

- Перегляд каталогу товарів з можливістю фільтрації.

- Перегляд детальної сторінки товару з описом, характеристиками, фото, модифікаціями (об’єм, колір тощо).

- Додавання товарів у кошик.

- Оформлення замовлення.

Реалізація функціональності з боку адміністратора:

- CRUD-операції над товарами, категоріями, брендами, фільтрами та характеристиками.

- Перегляд та обробка замовлень користувачів.

- Завантаження зображень для товарів.

- Забезпечення валідації форм для уникнення порожніх або некоректних даних.

Перевірка працездатності вебзастосунку:

- Запуск проєкту на локальному сервері для перевірки взаємодії між клієнтом та сервером.

- Тестування всіх HTTP-запитів у Postman (GET, POST, PUT, DELETE).

- Перевірка обробки помилок, валідації та повідомлень на клієнті.

Очікуваним результатом є повнофункціональний вебзастосунок онлайн-магазину косметики, який дозволить як звичайним користувачам зручно переглядати та купувати товари, так і адміністраторам — ефективно управляти вмістом магазину. Система повинна бути масштабованою, безпечною та розробленою з урахуванням сучасних стандартів веброзробки.

# 2 ОБҐРУНТУВАННЯ ВИБОРУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ОСНОВА СТРУКТУРИ ВЕБЗАСТОСУНКУ

## 2.1 Розробка структури бази даних

Для створення бази даних спершу необхідно визначити її структуру та вміст. Переглянувши вимоги в технічному завданню, можемо створити такі таблиці:

Таблиця 2.1 містить інформацію про зареєстрованих користувачів. Вона включає такі дані, як електронна адреса, ім’я, прізвище та пароль. Первинним ключем таблиці є поле id, яке гарантує унікальність кожного користувача. Таблиця має зовнішні зв’язки з такими таблицями: orders, addresses, cart_items, а також user_roles, яка в свою чергу пов'язана з таблицею roles. Це дозволяє реалізувати функціональність багаторівневої авторизації через зв’язок «багато до багатьох».

**Таблиця 2.1 – users**

| Ключове поле | Назва стовпця | Тип даних | Довжина | Дозволяє NULL | Вміст |
|---|---|---|---|---|---|
| + | id | BIGINT | | - | ID користувача |
| | email | VARCHAR | 255 | - | Електронна пошта |
| | first_name | VARCHAR | 255 | + | Ім’я користувача |
| | last_name | VARCHAR | 255 | + | Прізвище користувача |
| | password | VARCHAR | 255 | - | Пароль користувача |

Таблиця 2.2 містить перелік можливих ролей користувачів у системі, таких як «адміністратор», «користувач» тощо. Первинним ключем є поле id. Зв’язок із таблицею users реалізовано через проміжну таблицю user_roles, що дозволяє одному користувачу мати декілька ролей і навпаки.

**Таблиця 2.2 – roles**

| Ключове поле | Назва стовпця | Тип даних | Довжина | Дозволяє NULL | Вміст |
|---|---|---|---|---|---|
| + | id | BIGINT | | - | ID ролі |
| | name | ENUM | | - | Назва ролі |

Таблиця 2.3 це проміжна таблиця для реалізації зв'язку «багато до багатьох» між таблицями users і roles. Вона складається з двох зовнішніх ключів: user_id (посилається на users) та role_id (посилається на roles). Разом вони утворюють складовий первинний ключ, який забезпечує унікальність пар користувач-роль..

**Таблиця 2.3 – user_roles**

| Ключове поле | Назва стовпця | Тип даних | Довжина | Дозволяє NULL | Вміст |
|---|---|---|---|---|---|
| + | user_id | BIGINT | | - | ID користувача |
| + | role_id | BIGINT | | - | ID ролі |

Таблиця 2.4 призначена для зберігання адрес користувачів. Вона містить поля для міста, вулиці, номера будинку, квартири та поштового індексу. Зовнішній ключ user_id пов'язує адресу з конкретним користувачем. Це дозволяє кожному користувачу мати одну або декілька адрес.

**Таблиця 2.4 – addresses**

| Ключове поле | Назва стовпця | Тип даних | Довжина | Дозволяє NULL | Вміст |
|---|---|---|---|---|---|
| + | id | BIGINT | | - | ID адреси |
| | apartment | VARCHAR | 255 | + | Квартира |
| | city | VARCHAR | 255 | - | Місто |
| | house_number | VARCHAR | 255 | - | Номер будинку |
| | postal_code | VARCHAR | 255 | - | Поштовий індекс |
| | street | VARCHAR | 255 | - | Вулиця |
| | user_id | BIGINT | | - | ID користувача |

Таблиця 2.5 містить інформацію про замовлення, які здійснили користувачі. Кожен запис має унікальний id (первинний ключ), дату створення, статус і загальну вартість. Зовнішній ключ user_id зв’язує замовлення з користувачем. Таблиця пов’язана з order_items (товари в замовленні) та payments (оплата замовлення).

**Таблиця 2.5 – order**

| Ключове поле | Назва стовпця | Тип даних | Довжина | Дозволяє NULL | Вміст |
|---|---|---|---|---|---|
| + | id | BIGINT | | - | ID замовлення |
| | created_at | DATETIME(6) | | - | Дата створення |
| | status | ENUM | | - | Статус замовлення |
| | total_price | DECIMAL(38,2) | | - | Загальна ціна |
| | delivery_address_id | BIGINT | | | Адреса доставки |
| | user_id | BIGINT | | - | ID користувача |

Таблиця 2.6 зберігає перелік товарів, що входять до складу кожного замовлення. Первинним ключем є id. Зовнішні ключі: order_id (посилається на orders) і product_id (посилається на products). Це дозволяє зберігати кількість одиниць та ціну товару на момент покупки.

**Таблиця 2.6 – order_items**

| Ключове поле | Назва стовпця | Тип даних | Довжина | Дозволяє NULL | Вміст |
|---|---|---|---|---|---|
| + | id | BIGINT | | - | ID запису |
| | price_at_purchase | DECIMAL(38,2) | | - | Ціна під час покупки |
| | quantity | INT | | - | Кількість |
| | order_id | BIGINT | | - | ID замовлення |
| | product_id | BIGINT | | - | ID товару |

В таблиці 2.7 зберігаються дані про оплату замовлень. Кожен платіж має унікальний id (первинний ключ), дату, метод оплати, статус, а також унікальний transaction_id, який гарантує унікальність транзакції. Поле order_id є зовнішнім ключем і зв’язує платіж із відповідним замовленням.

**Таблиця 2.7 – payments**

| Ключове поле | Назва стовпця | Тип даних | Довжина | Дозволяє NULL | Вміст |
|---|---|---|---|---|---|
| + | id | BIGINT | | - | ID платежу |
| | created_at | DATETIME(6) | | - | Дата створення |
| | payment_method | ENUM | | - | Метод оплати |
| | status | ENUM | | - | Статус платежу |
| | transaction_id | VARCHAR | 255 | - | Унікальний ідентифікатор транзакції |
| | order_id | BIGINT | | - | ID замовлення |

Таблиця 2.8 зберігає інформацію про бренди товарів, включаючи назву та країну походження. Поле id є первинним ключем. Зв’язок з products реалізований через зовнішній ключ brand_id.

**Таблиця 2.8 – brands**

| Ключове поле | Назва стовпця | Тип даних | Довжина | Дозволяє NULL | Вміст |
|---|---|---|---|---|---|
| + | id | BIGINT | | - | ID запису |
| | country | VARCHAR | 255 | - | Країна бренду |
| | name | VARCHAR | 255 | - | Назва бренду |

Таблиця 2.9 містить перелік категорій товарів. Використовується ієрархічна структура за допомогою поля parent_id, що дозволяє формувати підкатегорії. Первинним ключем є id, а зовнішній ключ parent_id дозволяє реалізувати деревоподібну структуру. Таблиця має зв’язки з products (через category_id) і category_attribute (через category_id).

**Таблиця 2.9 – categories**

| Ключове поле | Назва стовпця | Тип даних | Довжина | Дозволяє NULL | Вміст |
|---|---|---|---|---|---|
| + | id | BIGINT | | - | ID запису |
| | name | VARCHAR | 255 | - | Назва категорії |
| | parent_id | BIGINT | | + | ID батьківської категорії |

Таблиця 2.10 містить основну інформацію про товари, доступні на сайті: назву, опис, ціну, кількість на складі та дату додавання. Первинним ключем є id. Зовнішні ключі brand_id та category_id зв'язують товар відповідно з брендом і категорією. Ця таблиця також має зв'язки з product_images, order_items, cart_items і attribute_values.

**Таблиця 2.10 – products**

| Ключове поле | Назва стовпця | Тип даних | Довжина | Дозволяє NULL | Вміст |
|---|---|---|---|---|---|
| + | id | BIGINT | | - | ID запису |
| | created_at | DATETIME | | - | Дата створення |
| | description | VARCHAR | 255 | + | Опис товару |
| | name | VARCHAR | | - | Назва товару |
| | price | DECIMAL(38,2) | | - | Ціна |
| | stock_quantity | INT | | - | Кількість на складі |
| | brand_id | BIGINT | | - | ID бренду |
| | category_id | BIGINT | | - | ID категорії |

Таблиця 2.11 призначена для зберігання зображень товарів. Містить id, шлях до зображення (image_url) та зовнішній ключ product_id, який вказує на відповідний запис у таблиці products.

**Таблиця 2.11 – product_images**

| Ключове поле | Назва стовпця | Тип даних | Довжина | Дозволяє NULL | Вміст |
|---|---|---|---|---|---|
| + | id | BIGINT | | - | ID запису |
| | image_url | VARCHAR | 255 | - | Посилання на зображення |
| | product_id | BIGINT | | - | ID товару |

Таблиця 2.12 містить товари, які користувачі додали до свого кошика. Первинний ключ – id. Зовнішні ключі user_id та product_id зв'язують таблицю відповідно з користувачами та товарами. Це дозволяє зберігати інформацію про кількість і дату додавання товару в кошик.

**Таблиця 2.12 – cart_items**

| Ключове поле | Назва стовпця | Тип даних | Довжина | Дозволяє NULL | Вміст |
|---|---|---|---|---|---|
| + | id | BIGINT | | - | ID запису |
| | added_at | DATETIME(6) | | - | Дата додавання |
| | quantity | INT | | - | Кількість |
| | product_id | BIGINT | | - | ID товару |
| | user_id | BIGINT | | - | ID користувача |

Таблиця 2.13 містить перелік усіх можливих атрибутів товарів, наприклад, «колір», «об'єм» тощо. Первинним ключем є id. Ці атрибути застосовуються до категорій через таблицю category_attribute і до товарів через таблицю.

**Таблиця 2.13 – attributes**

| Ключове поле | Назва стовпця | Тип даних | Довжина | Дозволяє NULL | Вміст |
|---|---|---|---|---|---|
| + | id | BIGINT | | - | ID запису |
| | name | VARCHAR | 255 | - | Назва атрибута |

Таблиця 2.14 зберігає конкретні значення атрибутів для кожного товару. Первинний ключ – id. Містить зовнішні ключі attribute_id (зв’язок із таблицею attributes) і product_id (зв’язок із products). Це дозволяє визначити, які саме значення атрибутів має кожен товар.

**Таблиця 2.14 – attribute_values**

| Ключове поле | Назва стовпця | Тип даних | Довжина | Дозволяє NULL | Вміст |
|---|---|---|---|---|---|
| + | id | BIGINT | | - | ID запису |
| | value | VARCHAR | 255 | - | Значення |
| | attribute_id | BIGINT | | | ID атрибута |
| | product_id | BIGINT | | | ID продукту |

Таблиця 2.15 - це допоміжна таблиця для реалізації зв’язку «багато до багатьох» між категоріями (categories) та атрибутами (attributes). Вона складається з зовнішніх ключів category_id та attribute_id. Таким чином можна визначити, які атрибути є актуальними для певної категорії товарів.

Таблиця 2.15 – category_attribute

| Ключове поле | Назва стовпця | Тип даних | Довжина | Дозволяє NULL | Вміст |
|---|---|---|---|---|---|
| + | category_id | BIGINT | | - | ID категорії |
| + | attribute_id | BIGINT | | - | ID атрибута |

Дальше визначимо зв’язки між таблицями за допомогою ключових полів:

Первинний ключ — це одне або кілька полів (стовпців), комбінація значень яких однозначно визначає кожний запис у таблиці[12].

Зовнішній (вторинний) ключ — це одне або кілька полів (стовпців) у таблиці, що містять посилання на поле або поля первинного ключа в іншій таблиці. Зовнішній ключ визначає спосіб об’єднання таблиць[13].

Тепер розглянемо діаграму структуру бази даних:

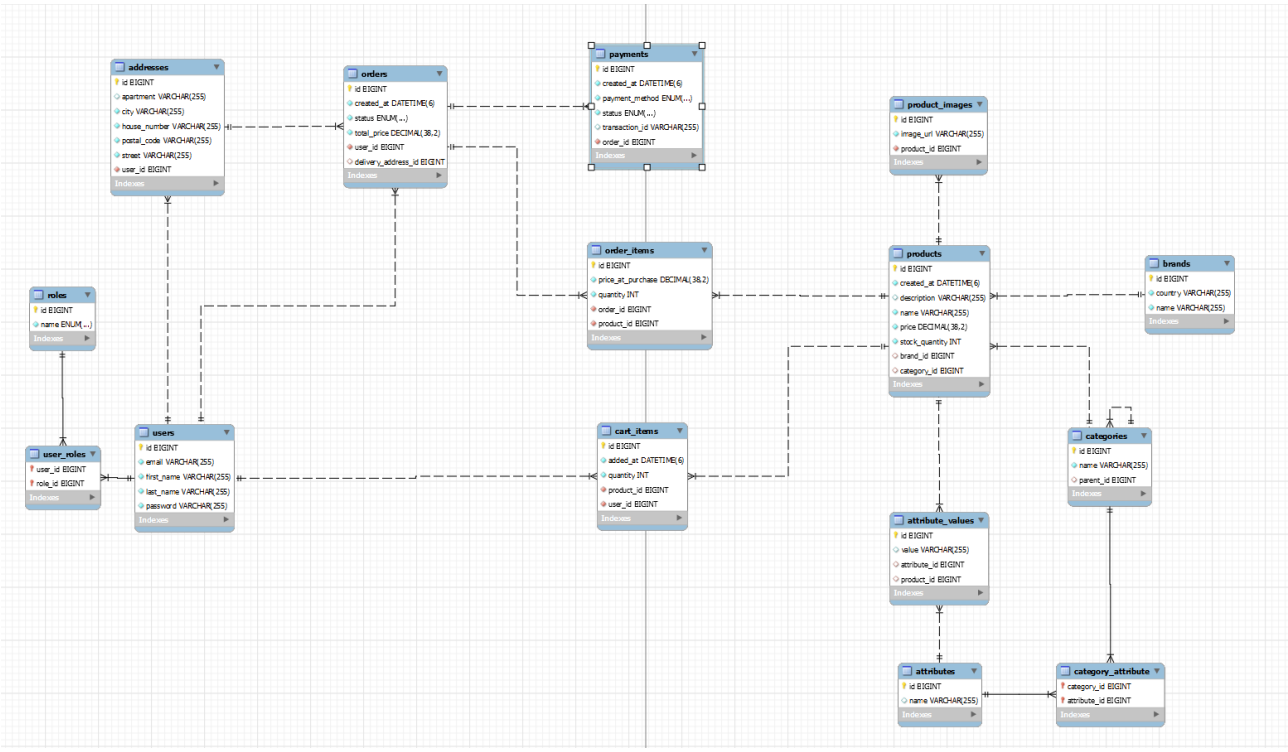


Рисунок 2.1 – Діаграма структури бази даних

Отже, структура бази даних вебзастосунку для онлайн-магазину буде реалізована у вигляді сукупності таблиць, які будуть пов’язані між собою за допомогою первинних та зовнішніх ключів. Первинні ключі (id) у кожній таблиці забезпечуватимуть унікальну ідентифікацію кожного запису. Зовнішні ключі відповідатимуть за встановлення зв’язків між таблицями, що дозволить забезпечити цілісність даних і логічні зв’язки між сутностями.

На прикладеній діаграмі (рис. 2.1) буде зображено повну структуру бази даних. Центральною сутністю системи стане таблиця users, яка зберігатиме інформацію про користувачів. З нею будуть пов’язані таблиці orders (замовлення), addresses (адреси користувачів), cart_items (елементи кошика), а також user_roles, яка реалізуватиме зв’язок користувача з певною роллю (наприклад, адміністратор чи звичайний користувач). Через таблицю user_roles буде реалізовано зв’язок багато-до-багатьох між таблицями users і roles.

Товари (products) будуть пов’язані з таблицями brands (бренди), categories (категорії), product_images (зображення товарів), а також з order_items (елементи замовлень), cart_items (кошик) та attribute_values (значення атрибутів). Кожен

товар належатиме певній категорії та бренду, що буде реалізовано за допомогою зовнішніх ключів category_id і brand_id.

Категорії будуть реалізовані у вигляді деревоподібної структури: поле parent_id посилатиметься на id тієї ж таблиці, що дозволить створювати ієрархічні зв'язки — підкатегорії. Для прив'язки динамічних характеристик товарів до певних категорій буде використано допоміжну таблицю category_attribute, яка встановлюватиме зв'язок багато-до-багатьох між таблицями categories та attributes.

Для зберігання додаткових характеристик товарів, таких як колір, розмір або об'єм, будуть використовуватися таблиці attributes (перелік можливих атрибутів) та attribute_values (конкретні значення атрибутів для товарів). Це дозволить реалізувати гнучку систему фільтрації на основі атрибутів.

Замовлення (orders) будуть пов'язуватися з користувачами, таблицею order_items (перелік придбаних товарів), а також з таблицею payments, яка зберігатиме інформацію про оплату кожного замовлення.

Розроблена модель бази даних відповідатиме вимогам нормалізації, включно з третьою нормальною формою (3НФ), що дозволить уникнути надмірності даних і забезпечить логічну цілісність структури[14].

Таким чином, модель бази даних, яка буде розроблена, відповідатиме вимогам вебзастосунку для онлайн-магазину. Вона забезпечуватиме логічну, нормалізовану структуру з чітко визначеними зв'язками, що дозволить ефективно управляти інформацією про користувачів, товари, замовлення, кошики, атрибути та оплату.

## 2.2 Створення бази даних для інтернет-магазину косметичної продукції

Після того як було визначено структуру бази даних для вебзастосунку інтернет-магазину косметики, можна переходити до безпосереднього її створення. У межах даного проєкту розробляється реляційна база даних, яка зберігатиме всю необхідну інформацію про товари, категорії, користувачів,

замовлення та інші сутності. Для створення та адміністрування цієї бази даних використовується СУБД MySQL разом із інструментом MySQL Workbench, який надає зручне графічне середовище для керування базами даних, виконання SQL-запитів, візуалізації структури таблиць і побудови зв'язків між ними.

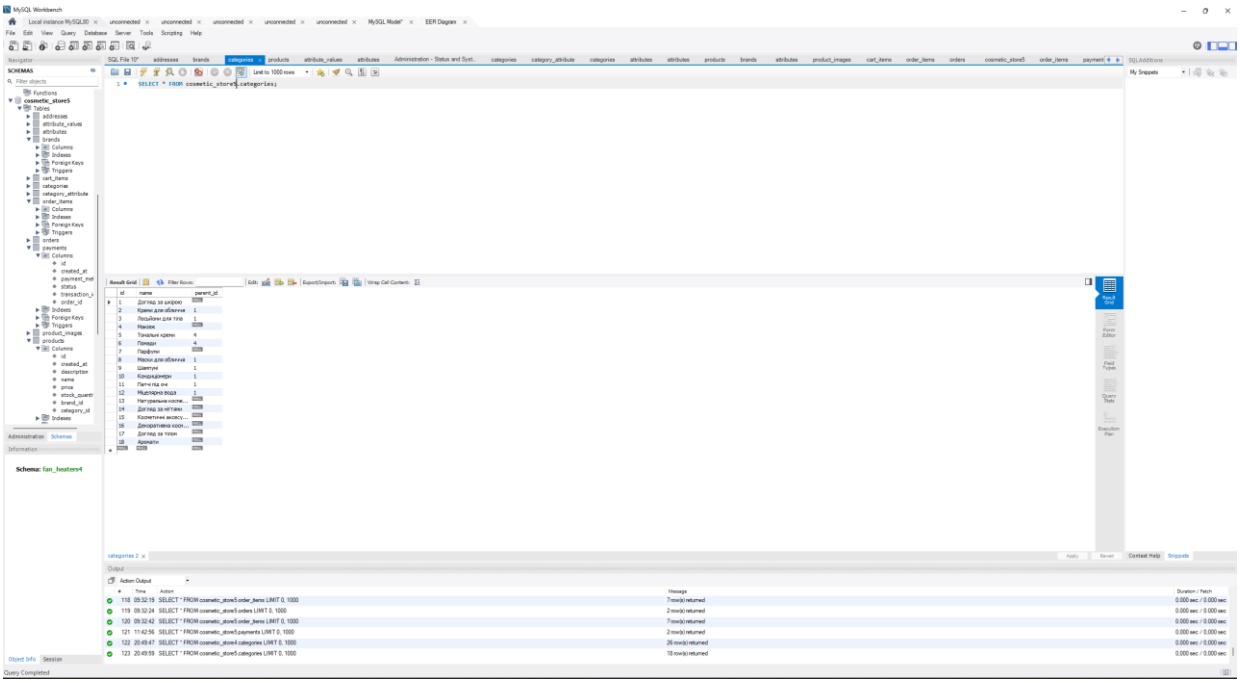

Рисунок 2.2 – Робоче середовище MySQL Workbench

На рисунку 2.2 зображено інтерфейс середовища MySQL Workbench, яке використовується для створення та адміністрування реляційної бази даних вебзастосунку інтернет-магазину косметики. У верхній частині вікна розташована вкладка з відкритою SQL-командою, що виконує запит на отримання всіх записів із таблиці categories. У нижній частині зображення бачимо результати цього запиту – таблиця містить категорії товарів, такі як "Догляд за шкірою", "Макіяж", "Парфумерія" тощо, що будуть використані для класифікації продукції в інтернет-магазині.

У лівій панелі навігації, в межах схеми cosmetic_store, відображено повний список таблиць, які формують структуру бази даних. Серед них можна побачити ключові сутності: products, categories, brands, attributes, product_images, order_items, cart_items, users та інші. Кожна з таблиць відповідає певному

функціональному модулю системи – наприклад, таблиця products зберігає дані про товари, users – про користувачів, а order_items – про склад замовлень.

Для створення таблиць, а також полів і зв'язків між ними, використовується вбудований SQL-редактор, представлений у центральній частині вікна. У ньому розробник має змогу виконувати запити, зокрема CREATE TABLE, що дозволяє поступово реалізовувати структуру, спроєктовану раніше в процесі нормалізації бази даних..

Саме створення таблиць бази даних і її полів виконувалося на вкладці меню SQL, в якій надається аркуш для виконання різних SQL-запитів, в нашому випадку CREATE TABLE.

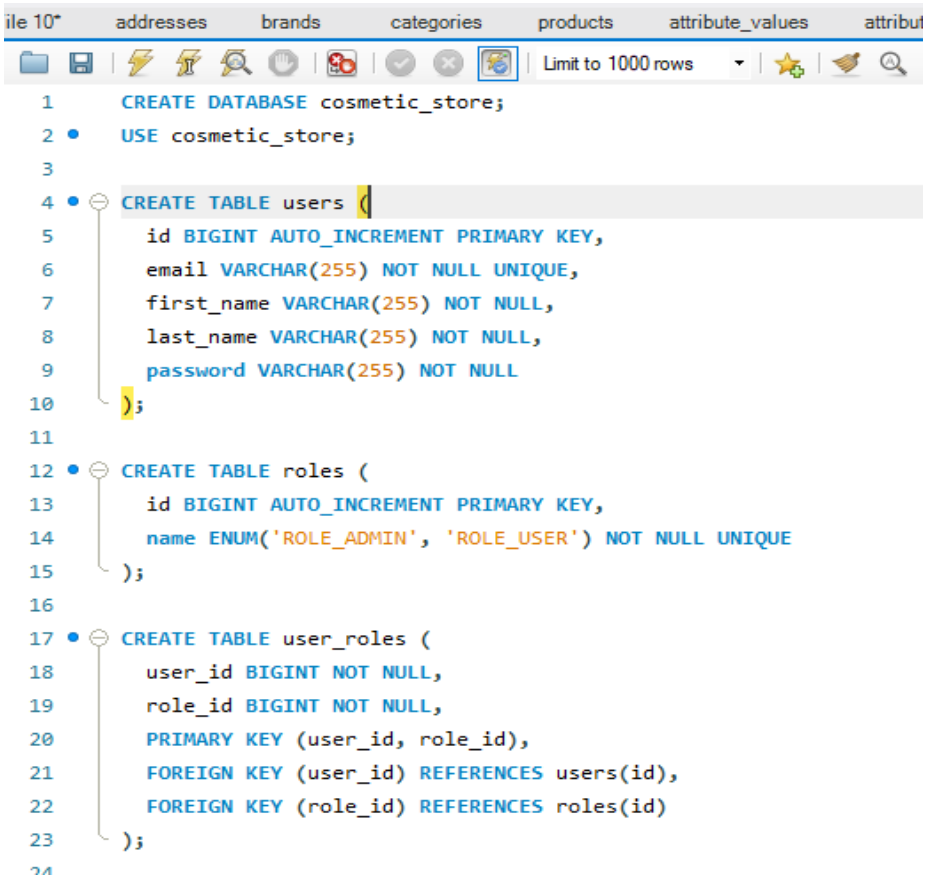


Рисунок 2.3 – Виконання запитів CREATE TABLE

На рисунку 2.3 представлено готові SQL-запити для створення таблиць бази даних інтернет-магазину косметики в середовищі MySQL з використанням MySQL Workbench. Першим кроком є створення самої бази даних.

Ці команди створюють нову базу даних cosmetic_store та активують її для подальшої роботи.Далі виконуються запити на створення основних таблиць: користувачів, ролей та таблиці для зв’язку між ними.

Важливо зазначити, що перед створенням таблиць необхідно явно вказати базу даних, у якій ці таблиці будуть створюватися, використовуючи команду USE. Якщо цього не зробити, при виконанні запитів виникатиме помилка, оскільки середовище не знатиме, куди зберігати нові таблиці.

Код CREATE TABLE:

Таблиця users - зберігатиме інформацію про ролі користувачів у системі для керування рівнями доступу користувачів.

```
CREATE TABLE users (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  email VARCHAR(255) NOT NULL UNIQUE,
  first_name VARCHAR(255) NOT NULL,
  last_name VARCHAR(255) NOT NULL,
  password VARCHAR(255) NOT NULL
);
```

Таблиця roles - зберігатиме інформацію про професійний досвід працівників університету.

```
CREATE TABLE roles (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  name ENUM('ROLE_ADMIN', 'ROLE_USER') NOT NULL UNIQUE
);
```

Таблиця user_roles– буде використовуватись для реалізації зв’язку багато-до-багатьох між користувачами та ролями. Вона зберігатиме, які ролі має кожен користувач.

```
CREATE TABLE user_roles (
  user_id BIGINT NOT NULL,
```

```
  role_id BIGINT NOT NULL,
  PRIMARY KEY (user_id, role_id),
  FOREIGN KEY (user_id) REFERENCES users(id),
  FOREIGN KEY (role_id) REFERENCES roles(id)
);
```

Таблиця addresses – зберігатиме інформацію про адреси користувачів.

```
CREATE TABLE addresses (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  apartment VARCHAR(255),
  city VARCHAR(255) NOT NULL,
  house_number VARCHAR(255) NOT NULL,
  postal_code VARCHAR(255) NOT NULL,
  street VARCHAR(255) NOT NULL,
  user_id BIGINT NOT NULL,
  FOREIGN KEY (user_id) REFERENCES users(id)
);
```

Таблиця categories - зберігатиме інформацію про категорії товарів.

```
CREATE TABLE categories (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(255) NOT NULL UNIQUE,
  parent_id BIGINT,
  FOREIGN KEY (parent_id) REFERENCES categories(id)
);
```

Таблиця brands - зберігатиме інформацію про бренди продукції.

```
CREATE TABLE brands (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(255) NOT NULL UNIQUE,
  country VARCHAR(255) NOT NULL
);
```

Таблиця products - зберігатиме інформацію про товари, доступні в магазині

```
CREATE TABLE products (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(255) NOT NULL UNIQUE,
  description VARCHAR(255),
  price DECIMAL(38, 2) NOT NULL,
  stock_quantity INT NOT NULL,
  created_at DATETIME NOT NULL,
  brand_id BIGINT,
  category_id BIGINT,
  FOREIGN KEY (brand_id) REFERENCES brands(id),
  FOREIGN KEY (category_id) REFERENCES categories(id)
);
```

Таблиця product_images - зберігатиме інформацію про зображення товарів у магазині.

```
CREATE TABLE product_images (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  image_url VARCHAR(255) NOT NULL,
  product_id BIGINT NOT NULL,
  FOREIGN KEY (product_id) REFERENCES products(id)
);
```

Таблиця attributes - зберігатиме інформацію про різні характеристики або атрибути, які можуть мати товари..

```
CREATE TABLE attributes (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(255) NOT NULL UNIQUE
);
```

Таблиця attribute_values - зберігатиме інформацію про конкретні значення атрибутів для товарів.

```
CREATE TABLE attribute_values (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  value VARCHAR(255),
  attribute_id BIGINT,
  product_id BIGINT,
  UNIQUE (attribute_id, product_id),
  FOREIGN KEY (attribute_id) REFERENCES attributes(id),
  FOREIGN KEY (product_id) REFERENCES products(id)
);
```

Таблиця category_attribute – буде використовуватись для вставнолвення зв’язку між категоріями товарів і атрибутами, які можуть бути застосовані до товарів у цих категоріях.

```
CREATE TABLE category_attribute (
  category_id BIGINT NOT NULL,
  attribute_id BIGINT NOT NULL,
  PRIMARY KEY (attribute_id, category_id),
  FOREIGN KEY (attribute_id) REFERENCES attributes(id),
  FOREIGN KEY (category_id) REFERENCES categories(id)
);
```

Таблиця order - зберігатиме інформацію про замовлення, зроблені користувачами магазину..

```
CREATE TABLE orders (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  created_at DATETIME NOT NULL,
  status   ENUM('NEW',   'PROCESSING',   'SHIPPED',   'DELIVERED',
'CANCELLED') NOT NULL,
  total_price DECIMAL(38, 2) NOT NULL,
  user_id BIGINT NOT NULL,
  delivery_address_id BIGINT NOT NULL,
  FOREIGN KEY (user_id) REFERENCES users(id),
  FOREIGN KEY (delivery_address_id) REFERENCES addresses (id));
```

Таблиця order_items - зберігатиме інформацію про конкретні товари, що входять до складу кожного замовлення.

```
CREATE TABLE order_items (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  quantity INT NOT NULL,
  price_at_purchase DECIMAL(38, 2) NOT NULL,
  order_id BIGINT NOT NULL,
  product_id BIGINT NOT NULL,
  FOREIGN KEY (order_id) REFERENCES orders(id),
  FOREIGN KEY (product_id) REFERENCES products(id)
);
```

Таблиця payments - зберігатиме інформацію про платежі, пов’язані з замовленнями.

```
CREATE TABLE payments (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  created_at DATETIME NOT NULL,
  payment_method ENUM('CARD', 'CASH', 'PAYPAL') NOT NULL,
  status ENUM('PENDING', 'COMPLETED', 'FAILED') NOT NULL,
  transaction_id VARCHAR(255),
  order_id BIGINT NOT NULL UNIQUE,
  FOREIGN KEY (order_id) REFERENCES orders(id)
);
```

Таблиця cart_items - зберігатиме інформацію про платежі, пов’язані з замовленнями.

```
CREATE TABLE cart_items (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  added_at DATETIME NOT NULL,
  quantity INT NOT NULL,
```

```
    user_id BIGINT NOT NULL,
    product_id BIGINT NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users(id),
    FOREIGN KEY (product_id) REFERENCES products(id)
);
```

Коли створено таблиці та встановлено зв’язки потрібно занести записи, щоб вони не були пустими. Оскільки потрібно  перевірити роботу методів CRUD, то додамо тестові дані. Додавання записів виконуємо через запити INSERT INTO.

Заносимо записи у таблицю users:

```
INSERT INTO users (email, first_name, last_name, password) VALUES
('john.doe@example.com', 'John', 'Doe', 'hashed_password_1'),
('jane.smith@example.com', 'Jane', 'Smith', 'hashed_password_2');
```

Заносимо записи у таблицю roles:

```
INSERT INTO roles (name) VALUES
('ROLE_ADMIN'),
('ROLE_USER');
```

Заносимо записи у таблицю user_roles:

```
INSERT INTO user_roles (user_id, role_id) VALUES
(1, 1),  -- John Doe — ROLE_ADMIN
(1, 2),  -- John Doe — ROLE_USER
(2, 2);  -- Jane Smith — ROLE_USER
```

Заносимо записи у таблицю addresses:

```
INSERT INTO addresses (apartment, city, house_number, postal_code,
street, user_id) VALUES
('12B', 'Kyiv', '24', '01001', 'Khreshchatyk', 1),
(NULL, 'Lviv', '15', '79000', 'Shevchenka', 2);
```

Заносимо записи у таблицю categories:

```
INSERT INTO categories (name, parent_id) VALUES
('Косметика', NULL),
('Макіяж', 1),
('Догляд за шкірою', 1);
```

Заносимо записи у таблицю brands:

```
INSERT INTO brands (name, country) VALUES
('Lush', 'UK'),
('Sephora', 'France');
```

Заносимо записи у таблицю products:

```
INSERT INTO products (name, description, price, stock_quantity, created_at, brand_id, category_id) VALUES
('Тональний крем Lush Velvet', 'Легкий тональний крем з натуральними компонентами', 450.00, 120, NOW(), 1, 2),
('Зволожуючий крем Sephora', 'Інтенсивний зволожуючий крем для всіх типів шкіри', 780.00, 80, NOW(), 2, 3);
```

Заносимо записи у таблицю product_images:

```
INSERT INTO product_images (image_url, product_id) VALUES
('https://example.com/images/lush-velvet.jpg', 1),
('https://example.com/images/sephora-moisturizer.jpg', 2);
```

Заносимо записи у таблицю attributes:

```
INSERT INTO attributes (name) VALUES
('Відтінок'),
('Об’єм');
```

Заносимо записи у таблицю attribute_values:

```
INSERT INTO attribute_values (value, attribute_id, product_id) VALUES
('Світлий', 1, 1),
('30 мл', 2, 1),
('Нейтральний', 1, 2),
('50 мл', 2, 2);
```

Заносимо записи у таблицю category_attribute:

```
INSERT INTO category_attribute (category_id, attribute_id) VALUES
(2, 1),
(2, 2),
(3, 1)
(3, 2);
```

Заносимо записи у таблицю orders:

```
INSERT INTO orders (created_at, status, total_price, user_id) VALUES
(NOW(), 'NEW', 1230.00, 1);
```

Заносимо записи у таблицю order_items:

```
INSERT INTO order_items (quantity, price_at_purchase, order_id,
product_id) VALUES
(1, 450.00, 1, 1),
(1, 780.00, 1, 2);
```

Заносимо записи у таблицю payments:

```
INSERT INTO payments (created_at, payment_method, status,
transaction_id, order_id) VALUES
(NOW(), 'PAYPAL', 'COMPLETED', 'txn_cosmetic_001', 1);
```

Заносимо записи у таблицю cart_items:

```
INSERT INTO cart_items (added_at, quantity, user_id, product_id)
VALUES(NOW(), 2, 2, 1);
```

Було створено всі необхідні таблиці для зберігання даних про користувачів, ролі, адреси, категорії, бренди, товари, атрибути, зображення, замовлення, платежі та кошик. Встановлено всі потрібні зв’язки між таблицями через первинні та зовнішні ключі. Після цього додано початкові тестові записи для перевірки роботи бази — користувачі з ролями, категорії та бренди, товари з атрибутами, замовлення, оплати та позиції в кошику. Це дозволяє розпочати подальшу розробку функціоналу магазину з готовою структурою даних і тестовими даними для перевірки.

### 2.3 Розробка діаграм взаємодії з користувачем

Розглянемо один із можливих сценаріїв взаємодії користувача з вебдодатком для інтернет-магазину косметики. Для цього буде побудовано діаграму, яка демонструє конкретну послідовність дій користувача під час перегляду товару та додавання його до кошика. Такий підхід дозволяє проаналізувати логіку роботи окремих функціональних компонентів системи, а також зрозуміти, яким чином реалізується взаємодія між користувачем, серверною частиною та базою даних у рамках певного випадку використання.

Вебзастосунок повинен забезпечувати повноцінну роботу як для звичайного користувача (покупця), так і для адміністратора. Зокрема, користувач може переглядати каталог товарів, використовувати фільтри, переглядати деталі товарів, додавати їх до кошика, оформлювати замовлення та керувати особистим кабінетом. Адміністратор має змогу управляти товарним асортиментом, категоріями, брендами, атрибутами, а також опрацьовувати замовлення.

Авторизація є обов’язковим кроком для отримання доступу до персоналізованих функцій. Після успішного входу користувач отримує доступ до функціоналу відповідно до своєї ролі — покупця або адміністратора.

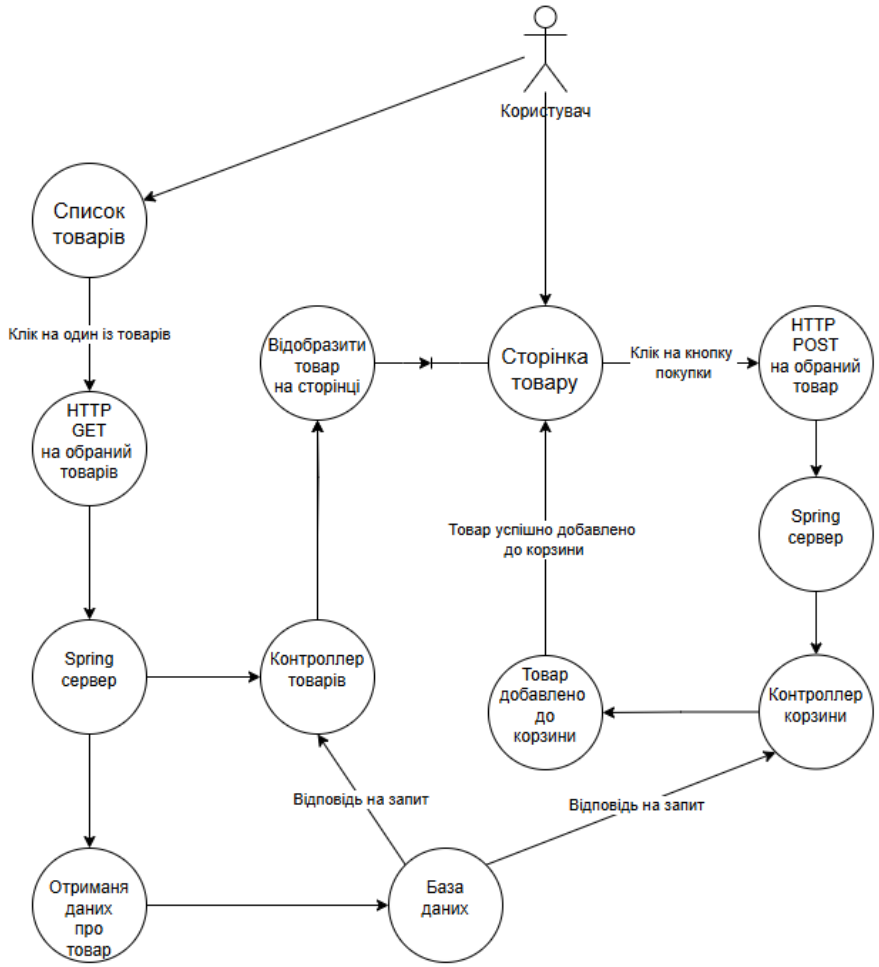


Рисунок 2.4 – Use-case діаграма взаємодії користувача з інтернет-магазином

На діаграмі 2.4 представлено один із ключових сценаріїв взаємодії користувача з інформаційною системою інтернет-магазину косметики — перегляд товару та додавання його до кошика. Діаграма демонструє послідовність запитів і відповідей між користувачем, сторінкою товару, контролерами, сервером та базою даних.

Зокрема, можна побачити, як користувач переходить зі списку товарів на сторінку одного з них, а далі — додає його до кошика. На кожному етапі система надсилає запити до відповідного контролера, який у свою чергу звертається до бази даних та повертає результат для відображення на стороні користувача. Усі

етапи дій виконуються за допомогою HTTP-запитів (GET, POST) у межах архітектури Spring Framework.

З метою наочного відображення процесу обміну даними між користувачем, серверною частиною вебзастосунку та базою даних, побудуємо діаграму послідовності. Вона демонструє, як саме відбувається обробка запитів у системі під час типових дій користувача.

На прикладі сценарію перегляду товару та додавання його до кошика показано, як користувач ініціює запити через інтерфейс вебдодатку, після чого сервер (реалізований з використанням Spring Framework) обробляє ці запити та звертається до бази даних MySQL. База даних виконує відповідні дії — отримання, запис або оновлення інформації — та повертає результат на сервер, який відображає його на стороні клієнта.

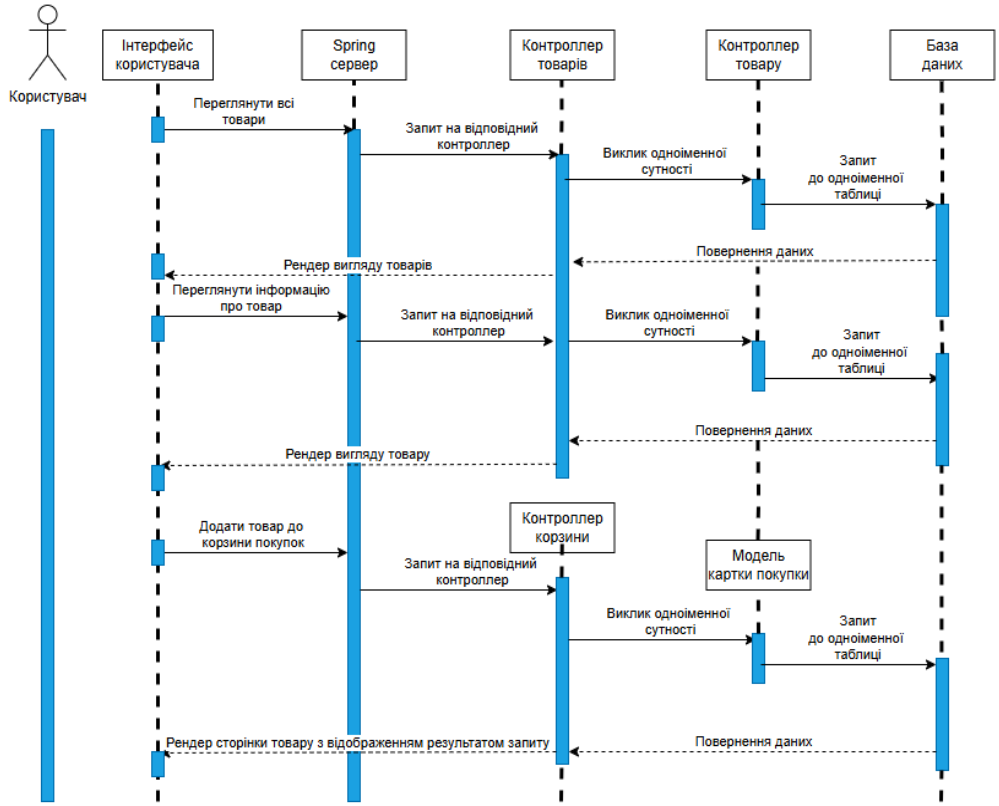


Рисунок 2.5 - Діаграма взаємодії із сервером

На діаграмі послідовності зображено процес взаємодії між користувачем, серверною частиною вебзастосунку та базою даних під час виконання базових операцій у системі. Ця діаграма дозволяє зрозуміти порядок та логіку обміну

повідомленнями між компонентами під час виконання таких дій, як перегляд товару, додавання до кошика або обробка запиту на створення замовлення.

Зокрема, на ній показано, як користувач через інтерфейс вебдодатку ініціює запит (наприклад, на перегляд товару), після чого сервер (Spring) обробляє цей запит і звертається до бази даних (MySQL), отримує відповідні дані, а потім повертає результат користувачу. Цей процес ілюструє типовий механізм роботи клієнт-серверної архітектури вебзастосунку, де кожна дія користувача запускає ланцюжок запитів і відповідей між компонентами системи.

# 3 РОЗРОБКА ФУНКЦІОНАЛУ ВЕБЗАСТОСУНКУ ІНТЕРНЕТ-МАГАЗИНУ КОСМЕТИКИ

## 3.1 Розробка програмного коду

Розроблений вебзастосунок базується на фреймворку Spring Boot, який забезпечує автоматичну конфігурацію, запуск сервера та управління життєвим циклом компонентів. Точкою входу до застосунку є головний клас LushglowApplication, у якому виконується ініціалізація всієї системи:

```
@SpringBootApplication(scanBasePackages = "lushglow.com")
public class LushglowApplication {
	public static void main(String[] args) {
		SpringApplication.run(LushglowApplication.class, args);
	}
}
```

Цей клас виконує наступні функції:

Анотація @SpringBootApplication — є комбінованою анотацією, яка поєднує в собі:

@Configuration — вказує, що клас містить конфігураційні біни.

@EnableAutoConfiguration — дозволяє Spring автоматично налаштовувати залежності на основі бібліотек у pom.xml.

@ComponentScan — дозволяє сканувати пакети на наявність компонентів (@Component, @Service, @Repository, @Controller).

У параметрі scanBasePackages = "lushglow.com" явно вказується базовий пакет, у якому Spring шукатиме компоненти для ініціалізації.

Метод main() — є стандартною точкою входу в Java-програму. В ньому викликається метод SpringApplication.run(...), який запускає Spring Boot застосунок, створює контекст, конфігурує бін-компоненти, ініціалізує сервер ( Tomcat) та готує застосунок до обробки HTTP-запитів.

Таким чином, клас LushglowApplication відповідає за повноцінний запуск усієї інформаційної системи інтернет-магазину косметики.

Файл application.properties у Spring Boot використовується для збереження конфігураційних параметрів застосунку. Саме в ньому налаштовується підключення до бази даних, параметри JPA, властивості безпеки, робота з шаблонізатором та іншими компонентами [15].

У розробленому застосунку для інтернет-магазину косметики конфігураційний файл має наступний вигляд:.

```
spring.application.name=lushglow
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/cosmetic_store5
spring.datasource.username=root
spring.datasource.password=root24547

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLD
ialect
spring.jpa.hibernate.ddl-auto=update

management.endpoints.web.exposure.include=beans

jwt.secret=gZd2Yz8nA2KJ+mCr8KhJknRuPchRM8ArE1tGiDzVb9c=

spring.thymeleaf.cache=false

spring.web.resources.static-
locations=classpath:/static/,file:uploads/
spring.servlet.multipart.max-file-size=5MB
spring.servlet.multipart.max-request-size=5MB;
```

Цей файл є ключовим для налаштування середовища виконання вебзастосунку, і його правильна конфігурація забезпечує стабільну роботу всіх компонентів системи.

Файл pom.xml є основним конфігураційним файлом для керування

залежностями у проєкті, що використовує систему збірки Maven. У ньому вказано всі необхідні бібліотеки, плагіни та властивості, потрібні для коректної роботи застосунку.

У проєкті використано такі основні залежності:

Spring Boot Starter Web – для створення вебзастосунку з REST-контролерами;

Spring Boot Starter Data JPA – для роботи з базою даних через ORM (Hibernate);

Spring Boot Starter Thymeleaf – для інтеграції з HTML-шаблонізатором;

Spring Boot Starter Security – для реалізації аутентифікації та авторизації;

MySQL Connector/J – драйвер для підключення до MySQL бази даних;

MapStruct – для автоматичного мапінгу між DTO та сутностями;

JWT (JJWT) – для генерації та валідації токенів при аутентифікації;

Lombok – для зменшення шаблонного коду (гетери, сетери, конструктори тощо);

Validation API – для валідації введених користувачем даних;

Spring Boot DevTools – для автоматичного перезавантаження застосунку під час розробки;

Spring Boot Actuator – для моніторингу стану застосунку.

Також у конфігурації вказано Java-версію 17, а плагін maven-compiler-plugin налаштований на підтримку інтеграції з MapStruct у середовищі Spring.

Цей файл дозволяє швидко керувати зовнішніми бібліотеками, підтримувати проєкт у актуальному стані та забезпечує автоматизовану збірку застосунку.

У процесі реалізації інтернет-магазину косметики було розроблено набір сутностей, які відповідають ключовим елементам предметної області. Кожна з них представляє відповідну таблицю у базі даних та забезпечує зв’язки між об’єктами. Одними із основних класів є класи User, Product та Category.

Клас User відповідає за зберігання та обробку інформації про зареєстрованих користувачів. Він реалізує інтерфейс UserDetails, що дозволяє

інтегрувати користувача з механізмами автентифікації Spring Security.

```
@Entity
@Table(name = "users")
public class User implements UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "first_name", nullable = false)
    private String firstName;
    @Column(name = "last_name", nullable = false)
    private String lastName;
    @Column(name = "email", unique = true, nullable = false)
    private String email;
    @Column(name = "password", nullable = false)
    private String password;
    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(
        name = "user_roles",
        joinColumns = @JoinColumn(name = "user_id"),
        inverseJoinColumns = @JoinColumn(name = "role_id")
    )
    private Set<Role> roles = new HashSet<>();

    // Реалізація методів з UserDetails: getAuthorities(),
getUsername() та ін.
}
```

У класі передбачено можливість призначення кількох ролей одному користувачу через зв'язок @ManyToMany з класом Role. В якості імені користувача використовується електронна адреса (email), що забезпечує унікальність.

Сутність Product представляє товар, доступний для продажу. Вона містить ключові поля, такі як назва, опис, ціна, кількість на складі та дата створення. Продукт зв’язаний із категорією та брендом, а також підтримує множинні зображення та характеристики для гнучкої фільтрації.

```
@Entity
@Table(name = "products")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name="name",nullable = false, unique = true)
    private String name;
    @Column(name = "description")
    private String description;
    @Column(name = "price",nullable = false)
    private BigDecimal price;
    @Column(name = "stock_quantity",nullable = false)
    private int stockQuantity;
    @ManyToOne
    @JoinColumn(name = "category_id")
    private Category category;
    @ManyToOne
    @JoinColumn(name = "brand_id")
    private Brand brand;
    @Column(name = "created_at", nullable = false)
    private LocalDateTime createdAt = LocalDateTime.now();
    @OneToMany(mappedBy = "product", cascade = CascadeType.ALL,
orphanRemoval = true)
    private List<AttributeValue> attributes = new ArrayList<>();
    @OneToMany(mappedBy = "product", cascade = CascadeType.ALL,
orphanRemoval = true, fetch = FetchType.EAGER)
    private List<ProductImage> images = new ArrayList<>();
}
```

Сутність Product має зв'язки з іншими сутностями:

з Category (@ManyToOne) – дозволяє віднести товар до певної категорії;

з Brand – визначає бренд продукту;

з ProductImage – забезпечує підтримку декількох зображень;

з AttributeValue – забезпечує зв’язок з характеристиками, що використовуються у фільтрах.

Сутність Category реалізує ієрархічну структуру категорій товарів. Категорія може містити підкатегорії через поле parentCategory, що дозволяє формувати дерева категорій.

```
@Entity
@Table(name = "categories")
public class Category {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name="name",nullable = false, unique = true)
    private String name;
    @ManyToOne
    @JoinColumn(name = "parent_id")
    private Category parentCategory;
```

Ієрархічний підхід дозволяє створювати зручну систему навігації по товарах — наприклад:

"Догляд за шкірою" → "Очищення" → "Гель для вмивання".

Проєкт також містить інші сутності, згруповані в пакеті entity:

Role — ролі користувачів (ADMIN, USER).

Brand, ProductImage — для представлення брендів і зображень товарів.

Attribute, AttributeValue, CategoryAttribute — реалізація атрибутів та фільтрації.

Order, OrderItem, Payment — для обробки замовлень та оплат.

CartItem, Address — підтримка кошика і доставки.

Для ефективного обміну даними між фронтендом та бекендом у вебзастосунку використовуються об’єкти DTO (Data Transfer Object). DTO дозволяють передавати лише потрібні дані, що зменшує навантаження на мережу та приховує внутрішню реалізацію сутностей. Перетворення між сутностями та DTO здійснюється за допомогою маперів, реалізованих із використанням бібліотеки MapStruct [16].

Об’єкт UserDTO призначений для передачі даних про користувача без розголошення конфіденційної інформації, зокрема паролів. Також передається список ролей користувача у вигляді рядків.

```
public class UserDTO {
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private Set<String> roles;
    // геттери і сеттери
}
```

Mapper — це інтерфейс, що забезпечує автоматичне перетворення між сутністю User та її DTO представленням UserDTO. У нашому проєкті ми використовуємо бібліотеку MapStruct, яка генерує код мапінгу під час компіляції, що значно підвищує продуктивність і зменшує кількість рутинного коду [17].

```
@Mapper(componentModel = "spring")
public interface UserMapper {

    UserMapper INSTANCE = Mappers.getMapper(UserMapper.class);
    @Mapping(target = "roles", source = "roles", qualifiedByName =
"rolesToStrings")
    UserDTO toDto(User user);
    @Mapping(target = "roles", source = "roles", qualifiedByName =
"stringsToRoles")
    User toEntity(UserDTO userDTO);
    @Named("rolesToStrings")
    static Set<String> rolesToStrings(Set<Role> roles) {
        return roles.stream()
                .map(role -> role.getName().name())
                .collect(Collectors.toSet());
    }
    @Named("stringsToRoles")
    static Set<Role> stringsToRoles(Set<String> roleNames) {
```

```
        return roleNames.stream()
                .map(roleName -> new
Role(RoleType.valueOf(roleName))).collect(Collectors.toSet());
    }
}
```

Головні функції UserMapper:

Перетворення сутності User у DTO UserDTO для передачі даних у контролер або клієнтську частину.

Зворотнє перетворення UserDTO у сутність User для збереження або оновлення в базі даних.

Мапінг складного поля roles, де замість об’єктів ролей передається їх рядкове представлення (ім’я ролі), що спрощує роботу з ролями на клієнтській стороні.

Для кожної сутності в проєкті (User, Product, Category тощо) створені відповідні DTO класи та Mapper-и, що забезпечують послідовну і уніфіковану передачу та перетворення даних між рівнями застосунку.

У сучасних вебзастосунках, що побудовані на основі Spring Framework, часто використовується шаблон архітектури Repository–Service–Controller. Цей підхід дозволяє чітко розділити відповідальність між різними компонентами:

Репозиторій (Repository) — забезпечує доступ до бази даних, містить методи для зчитування, збереження, оновлення та видалення даних. Базується на інтерфейсі JpaRepository, який надає стандартні CRUD-операції, а також дозволяє писати власні SQL або JPQL-запити.

Сервіс (Service) — містить бізнес-логіку застосунку, працює з даними через репозиторії.

Контролер (Controller) — обробляє HTTP-запити, взаємодіє з сервісом і повертає дані у вигляді відповіді.

Такий поділ забезпечує масштабованість, зручність підтримки, повторне використання коду та спрощує тестування.

```
@Repository
public interface ProductRepository extends JpaRepository<Product,Long>, JpaSpecificationExecutor<Product> {

    @Query("SELECT DISTINCT p FROM Product p LEFT JOIN FETCH p.images ORDER BY p.createdAt DESC")
    List<Product> findTop4WithImages();
    List<Product> findTop4ByCategoryIdAndIdNotOrderByCreatedAtDesc(Long categoryId, Long excludedProductId);

    List<Product> findTop4ByOrderByStockQuantityAsc();
    List<Product> findTop4ByBrandName(String name);

    @Query("SELECT p FROM Product p WHERE p.category.id = :categoryId ORDER BY p.id ASC")
    List<Product> findByCategoryIdWithPagination(@Param("categoryId") Long categoryId, Pageable pageable);
    List<Product> findByCategoryId(Long categoryId);
    boolean existsByName(String name);
    boolean existsByBrandId(Long brandId);
}
```

Інтерфейс ProductRepository відповідає за взаємодію з таблицею product у базі даних. Він розширює JpaRepository<Product, Long>, що надає базові CRUD-операції, а також JpaSpecificationExecutor<Product>, який дозволяє динамічно будувати складні фільтрації (наприклад, за ціною, брендом, наявністю та динамічними атрибутами).

Крім стандартних методів, у ProductRepository реалізовано кілька користувацьких запитів.

```
@Query("SELECT DISTINCT p FROM Product p LEFT JOIN FETCH p.images ORDER BY p.createdAt DESC")
    List<Product> findTop4WithImages();
```

Цей запит повертає останні чотири товари разом з їхніми зображеннями (JOIN FETCH), використовується для виводу новинок на головній сторінці.

Для кожної основної сутності проєкту (Product, Category, User тощо) створено відповідний інтерфейс репозиторію, що забезпечує всі необхідні операції доступу до даних.

У тришаровій архітектурі вебзастосунку сервісний шар є посередником між контролером (який працює з HTTP-запитами) і репозиторієм (який безпосередньо звертається до бази даних). Такий підхід дозволяє винести всю бізнес-логіку застосунку у спеціалізований шар, що забезпечує кращу організацію коду, його повторне використання, тестування та масштабування.

Сервісний шар реалізує функціональну логіку застосунку, використовуючи методи репозиторіїв. Для кожної сутності, з якою працює система, зазвичай створюється окремий сервіс. Наприклад, для роботи з продуктами реалізується інтерфейс ProductService та його реалізація ProductServiceImpl.

Інтерфейс ProductService визначає набір методів, які можуть бути використані в контролерах, наприклад:

```
public interface ProductService {
    ProductDTO createProduct(ProductDTO productDTO);
    ProductDTO getProductById(Long id);
    List<ProductDTO> getAllProducts();
    ProductDTO updateProduct(Long id, ProductDTO productDTO);
    boolean deleteProduct(Long id);

    List<ProductDTO> getProductsByCategory(Long categoryId);
    List<ProductDTO> getProductsByCategory(Long categoryId, int
offset, int limit);
    List<ProductDTO> getProductsByBrandName(String brandName);
    List<ProductDTO> getNewAddedProducts();
    List<ProductDTO> getTop4LowestStockProducts();
    List<ProductDTO> filterProductsByCategoryAndFilters(Long
categoryId, ProductFilterRequest filters);

}
```

Метод createProduct у класі ProductServiceImpl реалізує повну логіку створення нового товару.

```
    @Override
    @Transactional
    public ProductDTO createProduct(ProductDTO productDTO) {
        if (productRepository.existsByName(productDTO.getName())) {
            throw new RuntimeException("Product with this name already exists");
        }
        Product product = new Product();
        product.setName(productDTO.getName());
        product.setDescription(productDTO.getDescription());
        product.setPrice(productDTO.getPrice());
        product.setStockQuantity(productDTO.getStockQuantity());
        Category category = categoryRepository.findById(productDTO.getCategoryId())
                .orElseThrow(() -> new RuntimeException("Category not found"));
        product.setCategory(category);
        Brand brand = brandRepository.findById(productDTO.getBrandId())
                .orElseThrow(() -> new RuntimeException("Brand not found"));
        product.setBrand(brand);
        List<AttributeValueDto> attributeDtos = productDTO.getAttributes() != null
                ? productDTO.getAttributes()
                : new ArrayList<>();
        Set<Long> seenAttributeIds = new HashSet<>();
        for (AttributeValueDto dto : attributeDtos) {
            if (!seenAttributeIds.add(dto.getAttributeId())) {
                throw new RuntimeException("Duplicate attribute detected: attributeId=" + dto.getAttributeId());
            }
        }

        List<AttributeValue> attributeValues = new ArrayList<>();
        for (AttributeValueDto dto : attributeDtos) {
            Attribute attribute =
```

```
attributeRepository.findById(dto.getAttributeId())
                    .orElseThrow(() -> new RuntimeException("Attribute
not found with id: " + dto.getAttributeId()));
            AttributeValue attributeValue = new AttributeValue();
            attributeValue.setAttribute(attribute);
            attributeValue.setProduct(product);
            attributeValue.setValue(dto.getValue());
            attributeValues.add(attributeValue);
        }
        product.setAttributes(attributeValues);
        Product saved = productRepository.save(product);
        if (productDTO.getNewImages() != null) {
            for (MultipartFile file : productDTO.getNewImages()) {
                if (!file.isEmpty()) {
                    ProductImage image = imageStorageService.
storeImage(file);
                    image.setProduct(saved);
                    productImageRepository.save(image);
                }
            }
        }
        ProductDTO savedDto = productMapper.toDto(saved);

savedDto.setAttributes(productMapper.mapAttributesToDto(saved.getAttribut
es()));
        return savedDto;
    }}
```

Одним з ключових елементів функціональності інтернет-магазину є можливість додавання нових товарів до системи. Для реалізації цієї задачі у сервісному класі ProductServiceImpl створено метод createProduct(ProductDTO productDTO), який приймає об’єкт DTO з усією необхідною інформацією про товар і виконує повний цикл його створення: від перевірки даних до збереження зображень.

Метод реалізовано з використанням анотації @Transactional, що гарантує атомарність усієї операції. Це означає, що всі дії в межах методу або будуть

виконані успішно разом, або у випадку помилки система автоматично відкотить усі зміни, захищаючи цілісність даних у базі.

Перший етап полягає в перевірці наявності товару з такою самою назвою. Це дозволяє уникнути дублювання продуктів, що може призвести до плутанини в інтерфейсі адміністратора або неправильного відображення на сторінці користувача. Якщо продукт з такою назвою вже існує, метод викидає виняток з відповідним повідомленням.

Наступним кроком є створення нового об’єкта сутності Product та заповнення його основних властивостей: назви, опису, ціни, кількості на складі. Ці дані надходять із переданого об'єкта ProductDTO, що дозволяє чітко відокремити зовнішній рівень (DTO) від внутрішньої логіки (сущності бази даних).

Далі виконується прив’язка категорії товару. Для цього з бази даних за ID, переданим у DTO, завантажується відповідна сутність Category. Якщо категорія не знайдена, система також припиняє виконання з помилкою. Аналогічна логіка використовується для визначення бренду продукту.

Окрему увагу в методі приділено роботі з атрибутами товару, такими як колір, об’єм, текстура тощо. Дані про атрибути передаються у вигляді списку спеціальних об’єктів AttributeValueDto. Перед створенням відповідних сутностей AttributeValue система перевіряє, чи немає серед них дублікатів за ID атрибута, що дозволяє уникнути помилкового збереження кількох значень одного і того ж атрибута для одного продукту. Після цього кожен атрибут отримує свій об’єкт AttributeValue, який містить посилання на конкретний атрибут, значення цього атрибута і зв’язок із створеним продуктом.

Після формування всіх даних і зв’язків об’єкт Product з усіма підв’язаними атрибутами зберігається у базу даних за допомогою JPA-репозиторію productRepository.

Останнім кроком у процесі створення є збереження зображень товару. Якщо DTO містить список нових зображень (MultipartFile), кожне з них обробляється спеціальним сервісом imageStorageService, який відповідає за збереження файлів

(наприклад, на сервері чи у файловій системі). Кожне збережене зображення також пов’язується з відповідним продуктом у базі даних через окрему сутність ProductImage.

На завершення метод повертає DTO з усіма збереженими даними продукту, включно з атрибутами. Мапінг між сутністю Product і DTO реалізовано за допомогою ProductMapper, який конвертує об’єкти в зручний для фронтенду формат.

Загалом, цей метод демонструє повноцінний приклад реалізації складної бізнес-логіки на стороні бекенду. Його структура дозволяє підтримувати розширюваність (додавання нових атрибутів, властивостей, перевірок), а також забезпечує стабільність та безпечність з точки зору обробки даних користувача.

Оскільки вся бізнес-логіка створення та обробки продукту вже реалізована у відповідному сервісному шарі, наступним важливим елементом є контролер, який взаємодіє безпосередньо з інтерфейсом користувача. Саме контролер приймає запити від адміністратора, формує моделі для HTML-форм і передає дані у відповідні сервіси. Розглянемо детальніше роботу контролера AdminProductController, який відповідає за адміністрування товарів у вебінтерфейсі.

Контролер AdminProductController відповідає за адміністрування товарів: відображення списку, створення, редагування та видалення продуктів. Він працює з DTO об’єктами, сервісами, репозиторіями та формує модель для HTML-шаблонів за допомогою Thymeleaf.

Контролер позначений анотацією @Controller, що вказує Spring'у на те, що він повинен обробляти HTTP-запити. За допомогою @RequestMapping("/admin/products") всі запити до нього мають починатися з /admin/products.

Контролер використовує кілька сервісів та репозиторіїв:

ProductServiceImpl — основна логіка створення, оновлення, видалення та отримання товарів.

CategoryRepository, BrandRepository — доступ до категорій і брендів для

заповнення форм.

AttributeServiceImpl — отримання списку доступних атрибутів для відображення у формі.

Ці залежності впроваджено через конструктор за допомогою анотації @RequiredArgsConstructor.

```
@GetMapping("/create")
public String createForm(Model model) throws JsonProcessingException
{
    model.addAttribute("product", new ProductDTO());
    model.addAttribute("categories", categoryRepository.findAll());
    model.addAttribute("brands", brandRepository.findAll());
    model.addAttribute("attributesList",
attributeService.getAllAttributes());
    return "admin/products/form";
}
```

Коли адміністратор відкриває сторінку створення нового товару, цей метод передає у форму порожній об’єкт ProductDTO, а також списки всіх доступних категорій, брендів та атрибутів. Таким чином, форма автоматично заповнюється варіантами вибору, що забезпечує зручність введення.

```
@PostMapping("/create")
public String create(@ModelAttribute("product") ProductDTO
productDTO,
@RequestParam(value = "newImages") List<MultipartFile> newImages) {

    productDTO.setNewImages(newImages);
    productService.createProduct(productDTO);
    return "redirect:/admin/products";
}
```

Метод create в контролері обробляє POST-запит, який надходить після заповнення та відправлення форми створення нового товару. За допомогою анотації @ModelAttribute, Spring автоматично формує об’єкт ProductDTO,

наповнюючи його даними, які користувач ввів у HTML-формі. Окремо з форми передаються зображення, які отримуються через параметр newImages типу List<MultipartFile>. Ці файли вручну додаються до DTO за допомогою методу setNewImages(...).

Після цього заповнений DTO передається до методу createProduct(...) у сервісному класі, де вже виконується вся основна логіка: перевірка даних, обробка зображень, прив’язка категорій, брендів, атрибутів і збереження інформації до бази даних. Завершується робота методу перенаправленням на сторінку зі списком товарів. Це забезпечує логічну навігацію і дозволяє адміністратору одразу побачити щойно доданий продукт у загальному переліку.

Шаблон форми для створення або редагування продукту реалізовано за допомогою HTML із використанням шаблонізатора Thymeleaf, що дозволяє динамічно зв’язувати дані з Java-об'єктами. У заголовку сторінки виводиться назва дії — «Редагування продукту» або «Створення продукту» — залежно від того, чи містить об’єкт product ідентифікатор. Форма надсилається методом POST і залежно від того, редагується продукт чи створюється новий, використовується відповідний URL. Зв'язок полів форми з об’єктом ProductDTO забезпечується через атрибут th:field, що автоматично підставляє назви властивостей.

У формі передбачено поля для назви, опису, ціни, кількості на складі, вибору категорії та бренду. Для заповнення випадаючих списків використовується th:each, який перебирає список категорій та брендів, переданих із бекенду. Також реалізовано відображення і редагування атрибутів продукту. Кожен атрибут складається з випадаючого списку для вибору типу атрибута та текстового поля для введення значення. Ці атрибути також зв’язані з DTO і підтримують динамічне додавання нових пар завдяки JavaScript. Початкові атрибути відображаються за допомогою циклу, а нові додаються через кнопку «+ Додати атрибут», яка викликає функцію, що клоную шаблонний блок із прихованого контейнера. Індексація атрибутів забезпечується через змінну attributeIndex, яка підраховує кількість вже доданих атрибутів.

Ще одна важлива частина форми – блок для роботи з фотографіями. Тут

відображаються всі поточні зображення продукту у вигляді мініатюр. Біля кожного зображення є чекбокс, що дозволяє позначити зображення для видалення. Додатково реалізовано можливість завантажити одразу кілька нових фотографій через input type="file" з атрибутом multiple. У нижній частині форми розміщено кнопки для збереження змін та повернення до списку продуктів. Вся форма має адаптивну верстку та приємне оформлення завдяки класам Tailwind CSS: використовуються сітки, відступи, закруглення, тіні та кольори для створення сучасного, легкого для сприйняття інтерфейсу. Вбудований скрипт написаний на JavaScript і відповідає за логіку динамічного додавання атрибутів – встановлює правильні назви для нових полів і додає їх у DOM при натисканні відповідної кнопки.

Сторінка стилізована за допомогою Tailwind CSS – це сучасний CSS-фреймворк, що базується на утилітарних класах. Його використання дозволяє створювати адаптивні, привабливі й водночас легкі інтерфейси без написання додаткових CSS-файлів. У шаблоні форми використовуються класи для керування сіткою (grid, grid-cols-2, gap-4), відступами (p-4, mb-2, mt-6), тінями (shadow, shadow-md), округленням (rounded-lg), кольорами (bg-white, text-gray-700) тощо. Для кнопок застосовуються класи Tailwind, що відповідають за розмір, фон, текст і ховери, наприклад: bg-blue-500 hover:bg-blue-600 text-white font-bold py-2 px-4 rounded. Завдяки Tailwind верстка залишається читабельною, легко змінюється та не вимагає створення окремих CSS-файлів чи класів для кожного елементу.

Обробка завантаження зображень реалізована в класі ImageStorageService, позначеному анотацією @Service. Цей клас відповідає за збереження та видалення файлів на диску. У конструкторі сервісу створюється директорія uploads/images/, якщо вона ще не існує. Метод storeImage приймає об'єкт MultipartFile, генерує унікальне ім’я файлу за допомогою UUID, копіює вміст файлу у вказану директорію, та створює об’єкт ProductImage, де в полі imageUrl зберігається шлях до зображення у форматі /images/назва_файлу.

Код файлу ImageStorageService.java:

```
@Service
public class ImageStorageService {

    private final String uploadDir = "uploads/images/";

    public ImageStorageService() {
        try {
            Path uploadPath = Paths.get(uploadDir);
            Files.createDirectories(uploadPath);
        } catch (IOException e) {
            throw new RuntimeException("Could not create upload
directory", e);
        }
    }

    public ProductImage storeImage(MultipartFile file) {
        try {
            String filename = UUID.randomUUID() + "_" +
file.getOriginalFilename();
            Path path = Paths.get(uploadDir + filename);
            Files.copy(file.getInputStream(), path);

            ProductImage image = new ProductImage();
            image.setImageUrl("/images/" + filename);
            return image;
        } catch (IOException e) {
            throw new RuntimeException("Failed to store image", e);
        }
    }

    public void deleteImage(ProductImage image) {
        try {
            Files.deleteIfExists(Paths.get(uploadDir +
Paths.get(image.getImageUrl()).getFileName()));
        } catch (IOException e) {
            // Логування
        }
    }
```

Завантажені файли можна обробляти як статичні ресурси за допомогою Spring MVC, попередньо налаштувавши відповідний ResourceHandler у конфігурації. Метод deleteImage дозволяє видаляти зображення з файлової системи. Для цього з URL зображення видобувається ім’я файлу, після чого викликається Files.deleteIfExists, що безпечно видаляє файл, якщо він існує. Таким чином, забезпечується повний цикл роботи з фотографіями — збереження, відображення та видалення, — що інтегрується з шаблоном форми редагування продукту.

Для забезпечення автентифікації та авторизації у вебзастосунку використано модуль Spring Security, який налаштовано на роботу зі stateless-сесіями через JWT-токени (JSON Web Token). Такий підхід дозволяє реалізувати безпечну систему входу без збереження стану на стороні сервера [18].

JWT-токен генерується за допомогою класу JwtUtil, який використовує секретний ключ та алгоритм підпису HMAC-SHA256. Токен містить ім’я користувача (subject), дату створення та термін дії (1 доба). Метод generateToken() створює новий токен, а методи extractUsername() та validateToken() дозволяють відповідно отримати дані з токена та перевірити його дійсність. Для генерації секретного ключа використовується окремий клас SecretKeyGenerator, який виводить закодований ключ у консоль [19].

JWT зберігається в cookie з іменем jwt на клієнтській стороні. Кожен запит, що надходить до сервера, обробляється фільтром JwtAuthenticationFilter, який є нащадком OncePerRequestFilter. У цьому фільтрі відбувається пошук токена у cookie, його валідація та, у разі успішної перевірки, створення об'єкта UsernamePasswordAuthenticationToken, що вноситься до SecurityContextHolder. Це забезпечує автентифікацію користувача на поточному запиті.

Конфігурація безпеки задана в класі SecurityConfig. Для налаштування фільтрації HTTP-запитів використовується SecurityFilterChain, у якому вказано, що:

доступ до публічних ресурсів (/auth/**, /, /images/**, CSS, JS) дозволений усім;

доступ до URL-адрес, які починаються на /admin/**, дозволений лише користувачам з роллю ADMIN;

усі інші запити вимагають автентифікації.

Обробка винятків налаштована через кастомні редіректи: неавторизовані користувачі перенаправляються на сторінку входу, а у випадку забороненого доступу (403) – на відповідну сторінку помилки.

Сесійну політику встановлено як SessionCreationPolicy.STATELESS, що означає повну відсутність серверної сесії — автентифікація здійснюється виключно на основі JWT.

Для шифрування паролів використовується алгоритм BCrypt (BCryptPasswordEncoder), що забезпечує надійний захист паролів від атак перебором.

Аутентифікація здійснюється через AuthenticationManager, який конфігурується на основі DaoAuthenticationProvider із підключенням власного сервісу CustomUserDetailsService, який реалізує завантаження користувача з бази даних.

Загалом, реалізована система безпеки поєднує сучасні підходи до автентифікації (JWT, cookie, stateless) та гнучке управління доступом, що забезпечує високий рівень захисту для користувачів і адміністративної частини застосунку.

Функціонал бекенду реалізовано таким чином: користувач взаємодіє з вебзастосунком через інтерфейс, переходячи на різні сторінки, кожна з яких має власний URL (маршрут). Залежно від маршруту спрацьовує відповідний контролер, який приймає HTTP-запити від користувача, викликає відповідні методи сервісного шару та забезпечує доступ до даних у базі.

Для кожної таблиці бази даних створено відповідну сутність (Entity), яка відображає її поля у вигляді Java-класів. Для передачі даних між різними рівнями застосунку використовуються DTO (Data Transfer Object), а для конвертації між сутностями та DTO – окремі мапери. Репозиторії реалізовані за допомогою Spring Data JPA та відповідають за роботу з базою даних. У сервісному шарі описано

бізнес-логіку застосунку, а контролери керують потоком даних між користувачем та бекендом.

Кожна сторінка інтерфейсу користувача представлена окремим стилізованим HTML-шаблоном із використанням Thymeleaf і Tailwind CSS, що забезпечує сучасний адаптивний вигляд сайту.

Доступ до окремих частин системи захищено за допомогою Spring Security. Усі запити перевіряються на наявність JWT-токена, що дозволяє забезпечити автентифікацію та авторизацію користувачів. Крім того, визначено ролі (наприклад, адміністратор) та відповідні права доступу до ресурсів. Таким чином, система побудована за архітектурними принципами розділення відповідальностей (controller-service-repository) та забезпечує надійний рівень безпеки, масштабованості та підтримки.

### 3.2 Перевірка працездатності вебпрограми

Коли створено програмний код можна перейти до перевірки працездатності створеного онлайн-магазину косметики. Для цього буде здійснено послідовний огляд основних сторінок та функціональних можливостей вебзастосунку як для звичайного користувача, так і для адміністратора.

Головна сторінка інтернет-магазину є центральним елементом взаємодії користувача з платформою.

Сторінка вебзастосунку має зручну структуру та сучасний дизайн, орієнтований на користувача. У верхній частині розташовано банер-карусель — слайдер, реалізований з використанням бібліотеки Swiper.js. Він показує великі рекламні зображення акційних пропозицій, що автоматично змінюються або перемикаються вручну за допомогою навігаційних стрілок. Це дозволяє ефективно презентувати поточні знижки чи популярні товари.

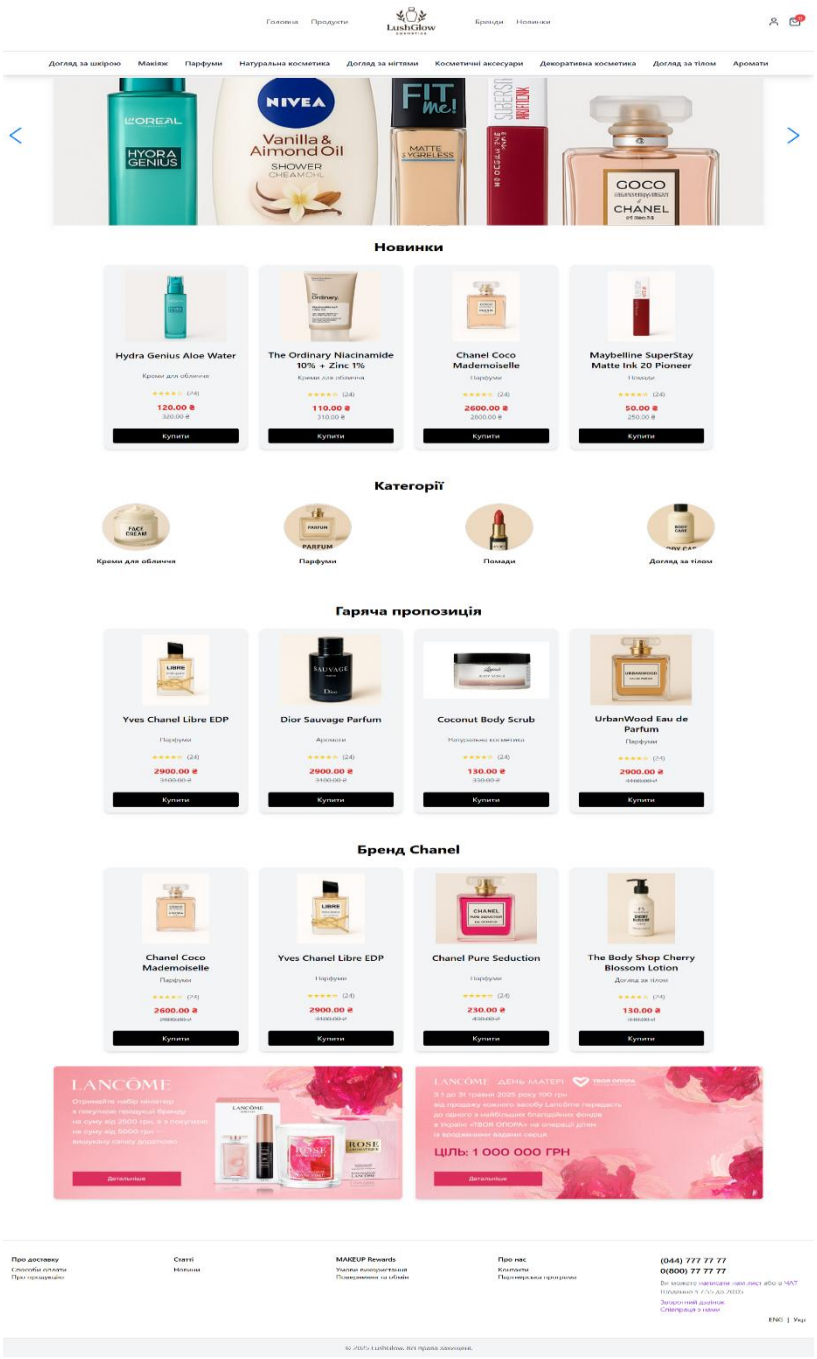


Рисунок 3.1 – Домашня сторінка

Нижче представлена секція новинок, у якій відображаються нові товари, що нещодавно надійшли у продаж. Кожен продукт має окрему картку з фотографією (або зображенням-заглушкою), назвою, категорією, рейтингом, старою та знижковою ціною, а також кнопкою «Купити», яка дозволяє одразу додати товар до кошика.

Далі розташовані популярні категорії у вигляді чотирьох округлих зображень з підписами. При натисканні на одну з категорій користувач

переходить до відповідного розділу товарів. Завдяки адаптивній верстці сторінка зберігає зручність перегляду на мобільних пристроях.

Секція «Гаряча пропозиція» привертає увагу до товарів, кількість яких на складі обмежена. Цей блок стимулює швидке прийняття рішення про покупку та оформлений у тому ж стилі, що й секція новинок.

Окремо виділено секцію, присвячену бренду Chanel. Вона містить добірку товарів цього бренду, отриманих через запит до відповідного сервісу. Це дозволяє зробити акцент на популярних чи преміальних виробниках.

Крім того, на сторінці розміщено кілька рекламних блоків, що займають всю ширину контентної області. У них відображаються графічні банери з текстовими описами акцій або спеціальних пропозицій. Вся верстка побудована з урахуванням принципів респонсивного дизайну, що забезпечує коректне відображення незалежно від типу пристрою.

Після переходу на сторінку певної категорії (наприклад, «Парфуми»), користувачу відображається список товарів, що належать до цієї категорії. Інтерфейс адаптовано до потреб користувача: зверху розміщена навігаційна панель з категоріями, а в основній частині — товари та фільтри для зручного пошуку.

Ліва частина сторінки містить панель фільтрації, яка дозволяє уточнити результати пошуку за такими параметрами, як бренд, ціна, кількість у наявності, аромат та об’єм. Дані фільтри формуються динамічно відповідно до обраної категорії. Користувач може позначити потрібні параметри та натиснути кнопку «Застосувати фільтри», після чого товари автоматично оновлюються відповідно до вибраних критеріїв.

Центральна частина сторінки заповнена картками товарів. Кожна картка містить зображення товару, його назву, категорію, та актуальну ціну. Якщо для товару встановлено знижку, також відображається стара ціна. Завдяки візуально привабливому оформленню користувачу легко зорієнтуватися серед асортименту.

Сторінка має адаптивний дизайн і коректно відображається на різних пристроях, включно зі смартфонами та планшетами.

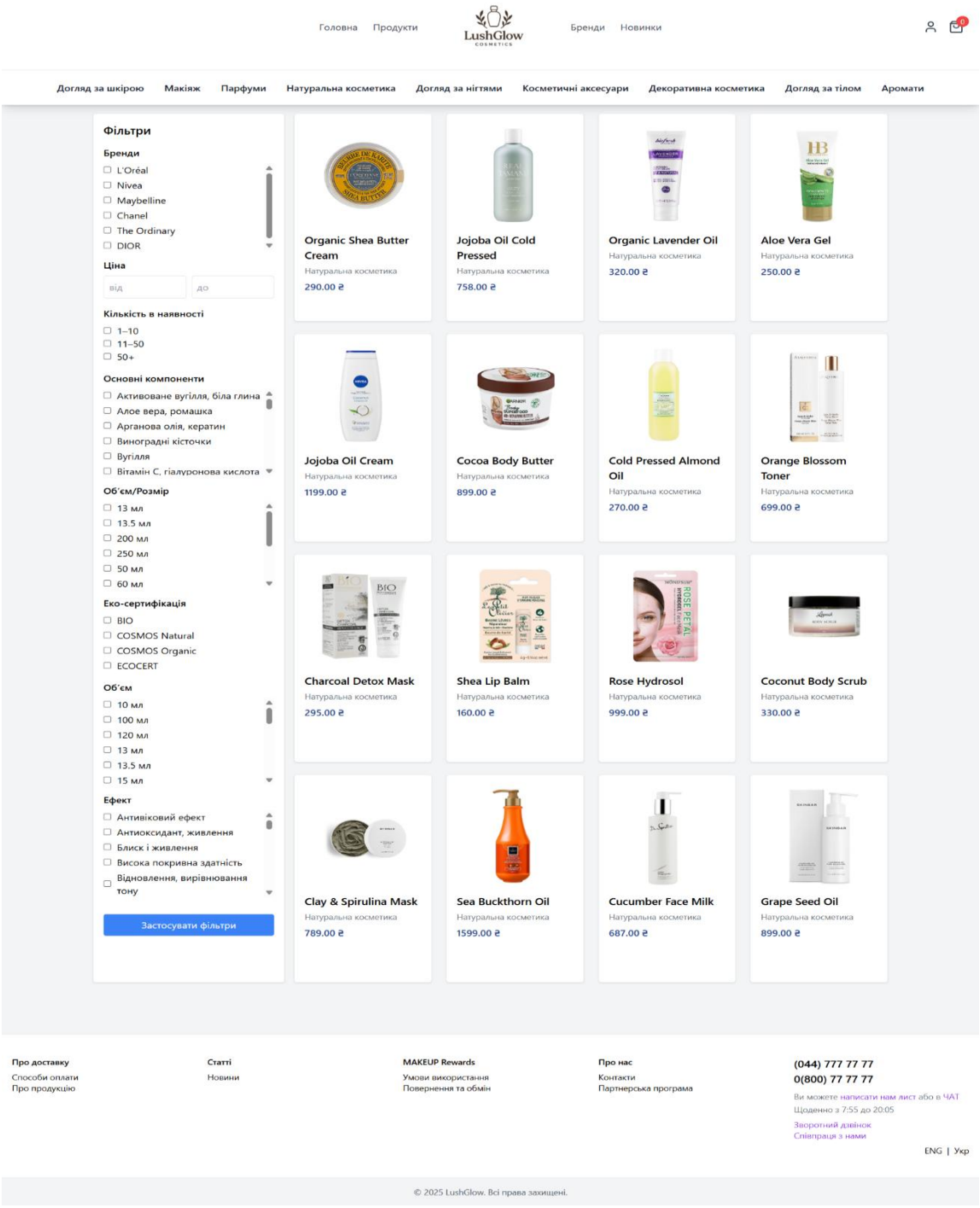


Рисунок 3.2 – Сторінка з товарами у категорії «Натуральна косметика»

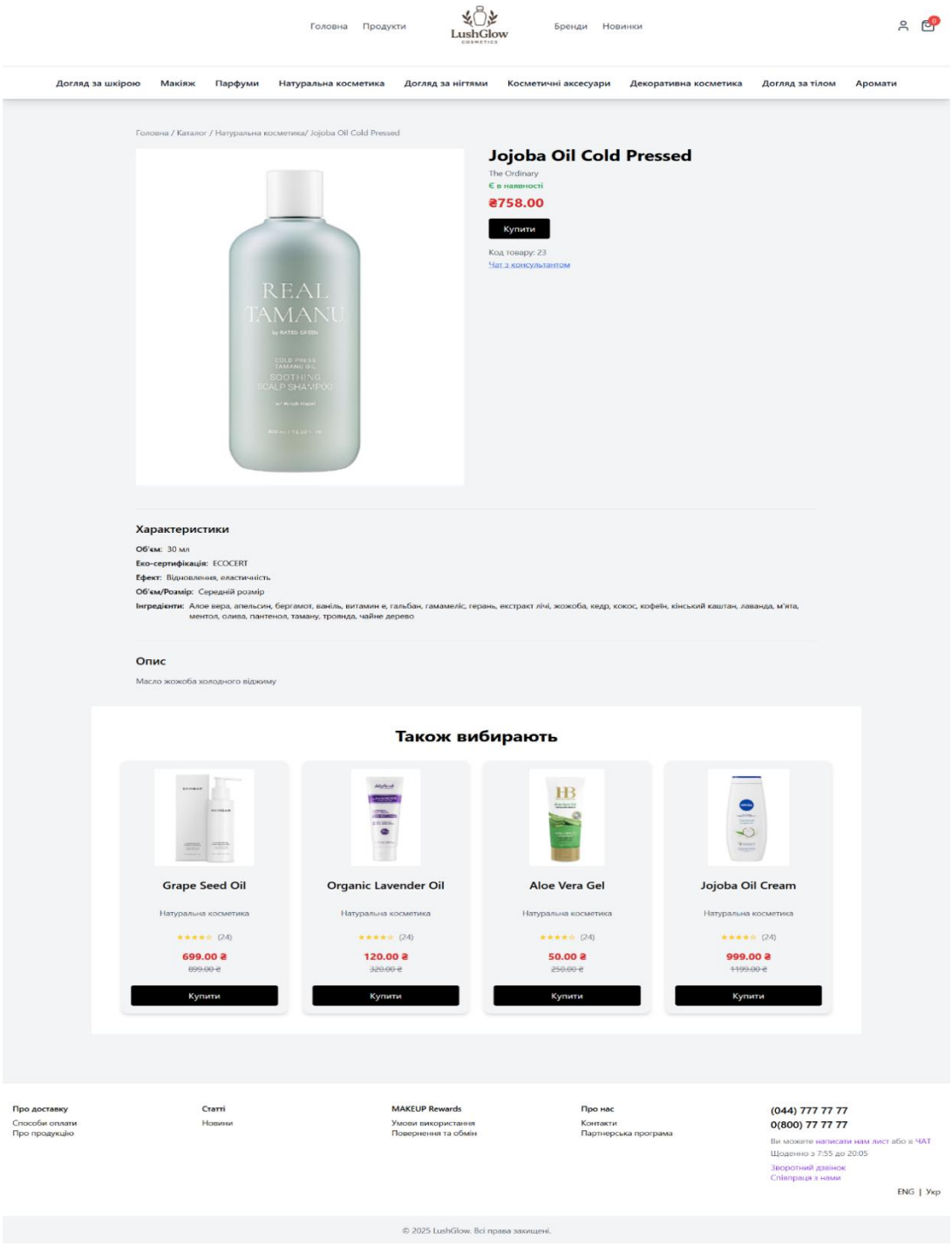


Рисунок 3.3 – Детальна сторінка товару

При натисканні на картку будь-якого товару на сторінці категорії, користувач переходить до детального перегляду обраного продукту. Сторінка містить повну інформацію про товар, включно з великим зображенням, назвою, брендом, статусом наявності, ціною, описом та характеристиками.

У правій частині представлено:назву товару та бренд, інформацію про наявність (зелений індикатор), актуальну ціну з можливістю відображення старої, якщо є знижка, кнопку "Купити", яка додає товар у кошик, якщо користувач не авторизований, при спробі додавання його буде переадресовано на сторінку авторизації, код товару, а також опцію звернутися до онлайн-консультанта.

Нижче розміщено блоки з:

Характеристиками товару, такими як об’єм та аромат.

Описом, який дає коротке уявлення про товар.

У нижній частині сторінки реалізовано секцію «Також вибирають», де відображаються інші популярні чи схожі товари з тієї ж категорії. Це сприяє покращенню користувацького досвіду та збільшує ймовірність придбання додаткових товарів.

У межах реалізації безпеки вебзастосунку було передбачено обмеження доступу до певного функціоналу лише для авторизованих користувачів. Зокрема, якщо користувач не увійшов у систему та намагається:

- додати товар до кошика;
- переглянути вміст кошика;
- перейти до оформлення замовлення;
- або ж отримати доступ до особистого кабінету;
- він автоматично перенаправляється на сторінку логування.

На цій сторінці користувач має змогу ввести свої облікові дані (електронну пошту та пароль) для входу в систему. У випадку, якщо обліковий запис ще не створено, доступне посилання «Зареєструватися», яке веде на відповідну сторінку реєстрації нового користувача.

Таке рішення дозволяє забезпечити належний рівень захисту персональної інформації користувачів, зокрема даних про замовлення та адреси доставки, а також дає змогу здійснювати подальший контроль за виконанням покупок.

На рисунках 3.4 та 3.5 представлено інтерфейси сторінок авторизації та реєстрації.

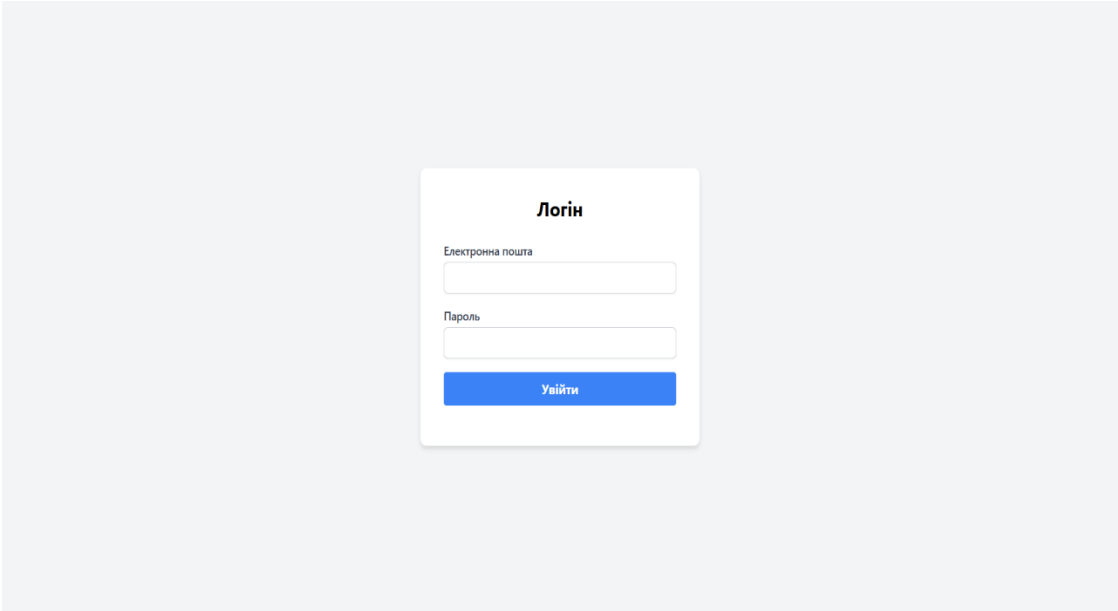


Рисунок 3.4 – Сторінка логування

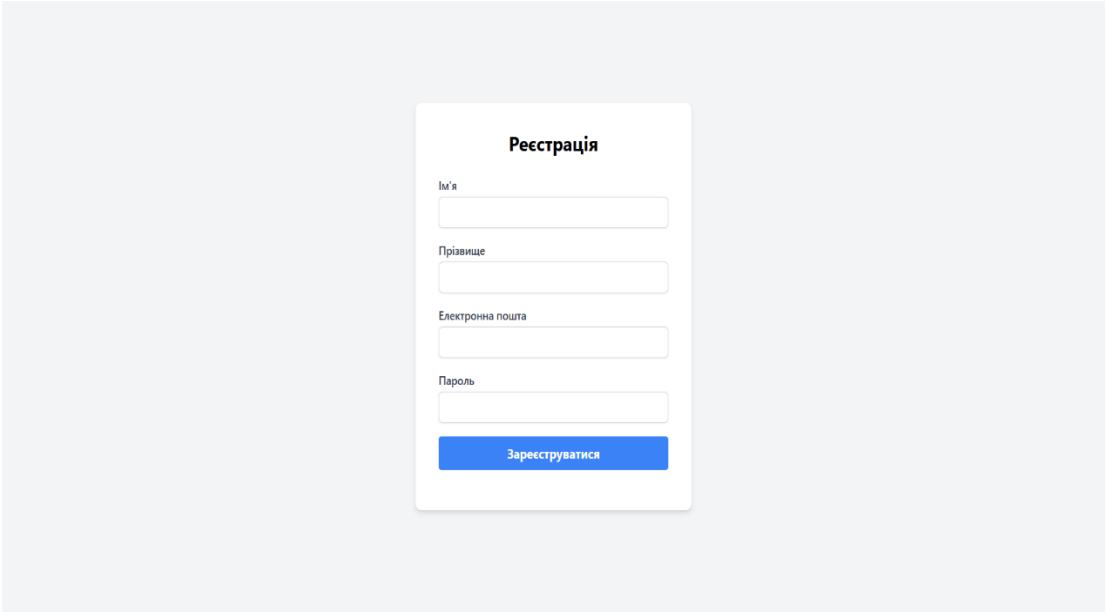


Рисунок 3.5 – Сторінка рейстрації

Після успішного входу в систему або реєстрації нового облікового запису користувач отримує повний доступ до функціоналу вебзастосунку, зокрема можливість додавати товари до кошика та оформлювати замовлення.

Для цього на сторінці товару передбачено кнопку «Купити». При її натисканні обраний товар додається до кошика користувача. Після цього на сайті

з’являється повідомлення про успішне додавання товару, яке інформує користувача про виконану дію.

Крім того, у правому верхньому куті сайту, де розташована іконка кошика, оновлюється лічильник – він відображає загальну кількість товарів, які наразі знаходяться в кошику. Таким чином, користувач одразу бачить, що товар було додано, і скільки товарів він планує придбати.

На рисунку 3.6 зображено приклад інтерфейсу після додавання товару до кошика.

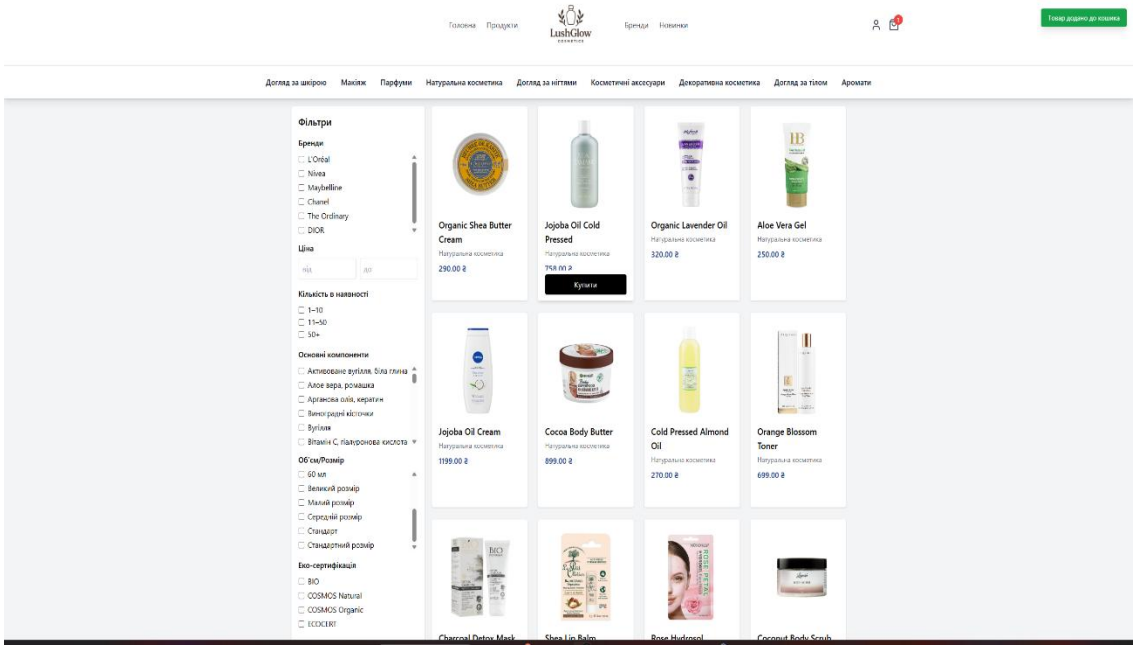


Рисунок 3.6 – Повідомлення про успішне додавання товару та оновлення лічильника кошика

Після додавання товарів до кошика користувач має можливість переглянути їх, змінити кількість або видалити непотрібні позиції перед оформленням покупки. Для цього в інтерфейсі сайту передбачена іконка кошика, розташована у верхній частині сторінки. Натискання на неї переадресовує користувача на сторінку «Кошик».

На сторінці кошика відображаються всі товари, які були додані користувачем. Для кожної позиції доступні наступні функції:

- Перегляд інформації про товар (назва, зображення, ціна).
- Зміна кількості одиниць товару для замовлення.

- Видалення товару з кошика.

- Автоматичне оновлення загальної суми в залежності від кількості товарів.

У нижній частині сторінки кошика розташована кнопка «Оформити замовлення». При її натисканні користувач переходить на сторінку підтвердження замовлення.

На рисунку 3.7 зображено приклад інтерфейсу кошика.

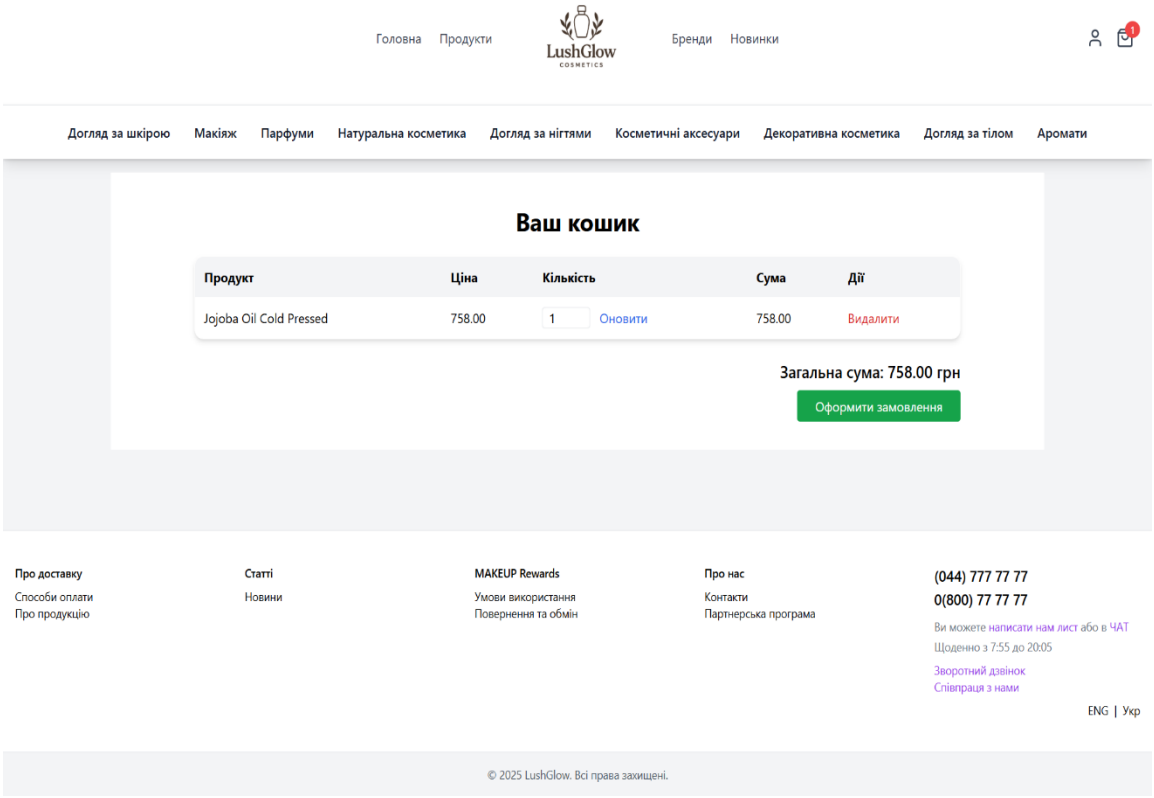


Рисунок 3.7 – Сторінка кошика з доданими товарами та функціями редагування

Сторінка оформлення замовлення відображає всі товари з кошика, але без можливості редагування. Мета цього етапу — підтвердження замовлення та заповнення персональних даних, необхідних для доставки.

На сторінці оформлення замовлення користувач повинен:

- Ввести свої контактні дані, зокрема: ім’я, прізвище, номер телефону, електронну пошту.

- Зазначити адресу доставки (місто, вулиця, номер будинку/квартири, поштовий індекс тощо).

- Вибрати спосіб оплати (наприклад, накладений платіж, оплата онлайн, готівкою при отриманні).

Після заповнення всіх полів користувач натискає кнопку «Підтвердити замовлення».

У результаті:

- Замовлення передається в систему для обробки.

- Кошик автоматично очищується.

- Користувач отримує повідомлення про успішне оформлення замовлення.

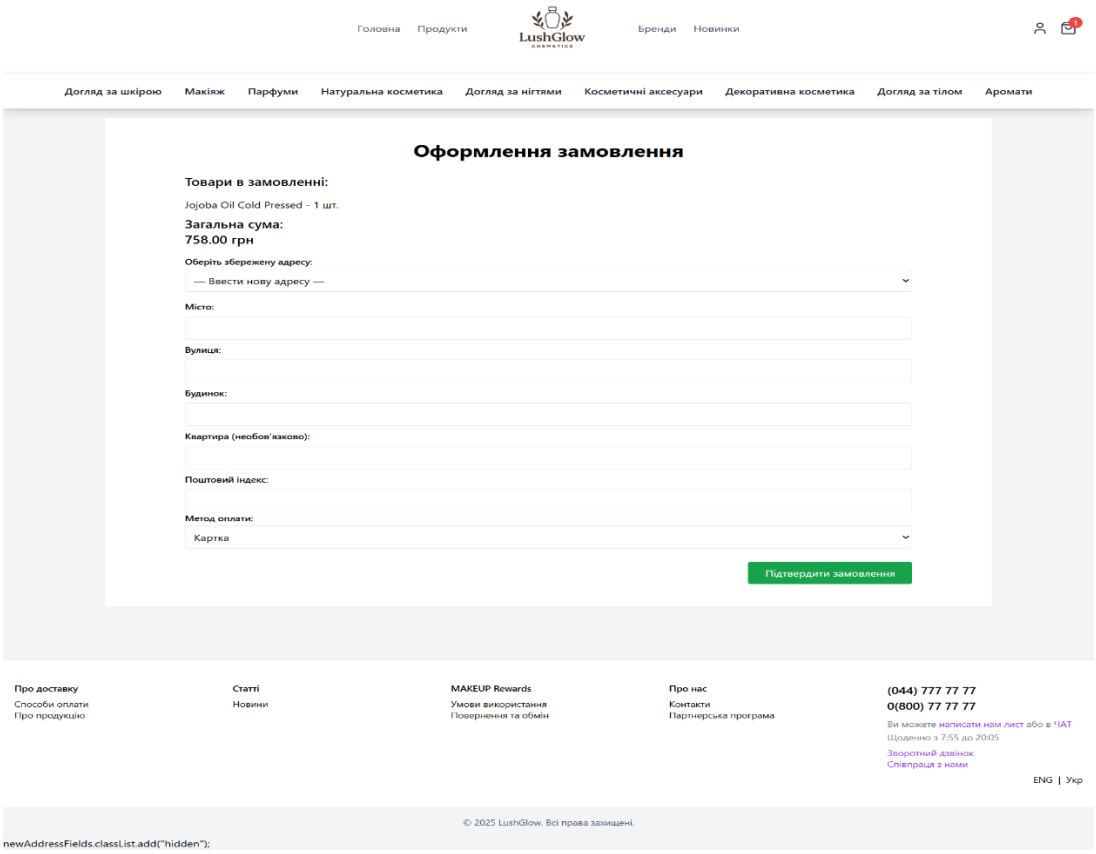


Рисунок 3.8 – Сторінка оформлення замовлення

На рисунку 3.8 представлено приклад сторінки оформлення замовлення.

У системі передбачено розмежування прав доступу для звичайних користувачів і адміністраторів. Якщо користувач, який увійшов у систему, має роль адміністратора, він отримує доступ до адміністративної панелі, де доступні функції управління контентом сайту.

Адмін має можливість:

Додавати нові товари, категорії, бренди;

Редагувати або видаляти існуючі записи;

Переглядати та керувати замовленнями користувачів;

Перевіряти наявність помилок при редагуванні даних.

Для прикладу розглянемо функціонал управління брендами, що реалізує повний набір CRUD-операцій:

- Створення бренду – адміністратор може ввести назву нового бренду в спеціальній формі.

- Перегляд брендів – відображається список усіх наявних брендів у таблиці з відповідними кнопками для редагування та видалення.

- Редагування бренду – у випадку помилки або змін назви адміністратор може відкрити форму редагування.

Видалення бренду – передбачена можливість видалити бренд, якщо він не використовується в жодному товарі.

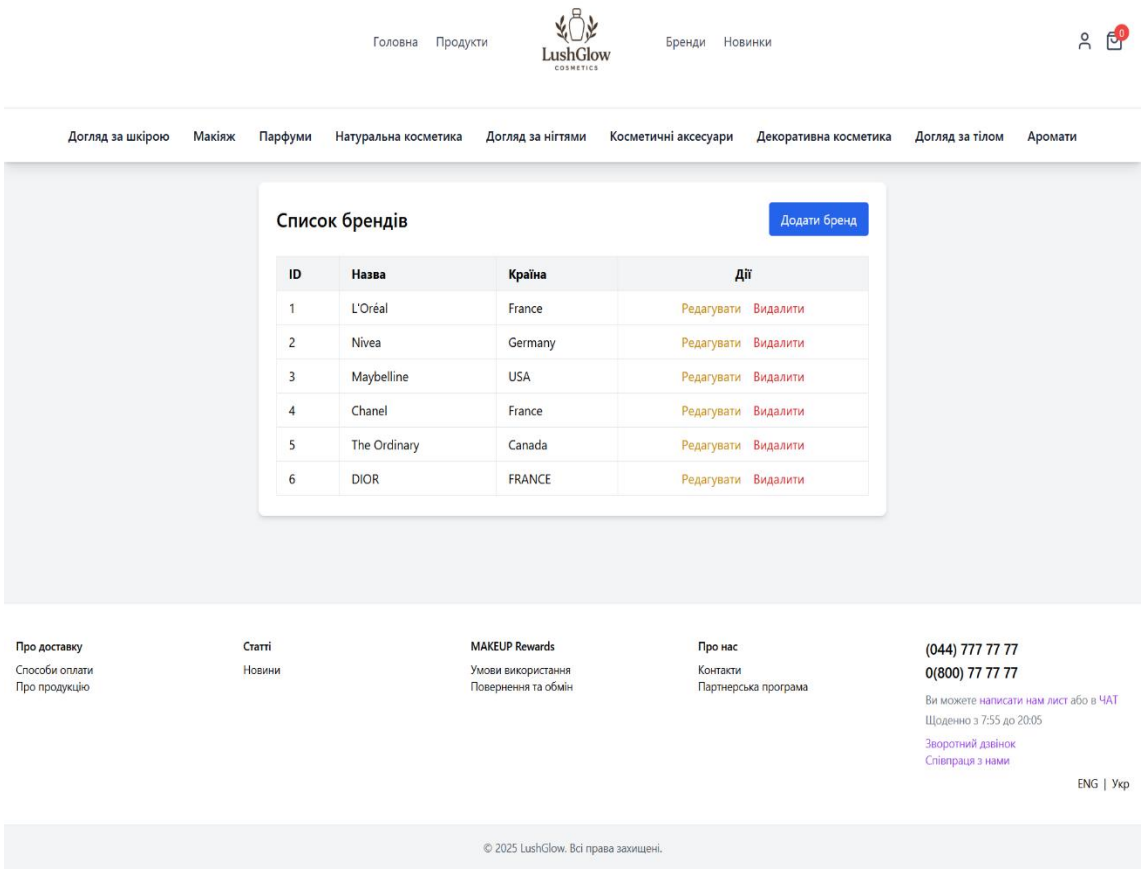


Рисунок 3.9 – Сторінка адмін-панелі: список брендів з можливістю редагування та видалення

Система реалізує перевірку унікальності імені бренду. Якщо адміністратор спробує додати бренд, який уже існує в базі даних, система виведе повідомлення про помилку, і запис не буде створено. Це дозволяє уникнути дублювання інформації та зберегти коректну структуру даних.

Крім того, під час спроби видалити бренд, який вже прив’язаний до одного чи кількох товарів, система також виведе повідомлення про неможливість видалення, доки не буде видалено або змінено відповідні товари.

# ВИСНОВКИ

У ході виконання бакалаврської роботи було розроблено вебзастосунок для інтернет-магазину косметики, що дозволяє користувачам переглядати асортимент товарів, здійснювати фільтрацію за категоріями, брендами, ціною, наявністю та іншими характеристиками, переглядати детальну інформацію про продукти, додавати товари до кошика, оформлювати замовлення, а також користуватися особистим кабінетом. Для адміністратора реалізовано функціонал управління товарами та замовленнями.

На початковому етапі було проведено опрацювання предметної області, зокрема аналіз роботи сучасних інтернет-магазинів косметики та подібних онлайн-платформ. Визначено основні функціональні вимоги до майбутньої інформаційної системи, що дозволило сформувати чітку структуру проєкту.

У процесі роботи виконано обґрунтування вибору програмного забезпечення. Як основні технології було обрано Java, Spring Boot, Hibernate, MySQL, Tailwind CSS і Thymeleaf. Для забезпечення безпеки аутентифікації та авторизації було використано Spring Security з JWT.

Після цього виконано проєктування та розробку структури бази даних, що охоплює сутності для зберігання інформації про товари, категорії, користувачів, замовлення, атрибути, значення атрибутів тощо. Базу даних було нормалізовано відповідно до вимог до зберігання структурованої інформації.

Далі було здійснено розробку програмного коду серверної та клієнтської частини системи. Створено головний клас застосунку, налаштовано середовище конфігурації, реалізовано DTO, сервіси, контролери, схеми фільтрації товарів, інтерфейс користувача тощо. Особливу увагу приділено реалізації фільтрації товарів за характеристиками, а також безпечній роботі з обліковими записами користувачів.

На завершальному етапі проведено перевірку працездатності інформаційної системи, що включає демонстрацію роботи основних сторінок: головної сторінки, каталогу товарів, фільтрації, детальної сторінки товару, процесу реєстрації,

додавання до кошика, оформлення замовлення та адміністративної панелі. Результати перевірки підтвердили правильну роботу всіх ключових функцій системи.

Таким чином, мету бакалаврської роботи було досягнуто – створено повноцінний вебзастосунок інтернет-магазину косметики з функціональністю, що відповідає сучасним вимогам до подібних вебзастосунків . Розроблений застосунок може бути використаний як основа для подальшого комерційного проєкту.

# СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ


1. MAKEUP — інтернет-магазин косметики. URL: https://makeup.com.ua/ua/ (дата звернення: 25.02.2025).

2. PARFUMS.UA — інтернет-магазин парфумерії та косметики. URL: https://parfums.ua/ua (дата звернення: 25.02.2025).

3. Sephora — офіційний сайт. URL: https://www.sephora.com/ (дата звернення: 25.02.2025).

4. Spring Framework Documentation | Spring. URL: https://docs.spring.io/spring-framework/docs/current/reference/html/ (дата звернення: 20.03.2025).

5. Hibernate ORM Documentation | Hibernate. URL: https://hibernate.org/orm/documentation/ (дата звернення: 25.03.2025).

6. MySQL :: MySQL Documentation. URL: https://dev.mysql.com/doc/ (дата звернення: 25.03.2025).

7. Tailwind CSS Documentation. URL: https://tailwindcss.com/docs (дата звернення: 12.04.2025).

8. Thymeleaf Documentation. URL: https://www.thymeleaf.org/documentation.html (дата звернення: 12.04.2025).

9. Spring Security Documentation. URL: https://docs.spring.io/spring-security/ (дата звернення: 24.04.2025).

10. Apache Tomcat® – Welcome. URL: https://tomcat.apache.org/ (дата звернення: 24.04.2025).

11. IntelliJ IDEA – офіційний сайт | JetBrains. URL: https://www.jetbrains.com/idea/ (дата звернення: 25.04.2025).

12. SQL PRIMARY KEY – W3Schools. URL: https://www.w3schools.com/sql/sql_primarykey.asp (дата звернення: 25.04.2025).

13. SQL FOREIGN KEY – W3Schools. URL: https://www.w3schools.com/sql/sql_foreignkey.asp (дата звернення: 25.04.2025).

14. Database Normalization – W3Schools.in. URL: https://www.w3schools.in/dbms/database-normalization (дата звернення: 25.04.2025).

15. Spring Boot Documentation | Spring. URL: https://docs.spring.io/spring-boot/docs/current/reference/html/ (дата звернення: 26.04.2025).

16. Java DTO Pattern – Baeldung. URL: https://www.baeldung.com/java-dto-pattern (дата звернення: 30.04.2025).

17. MapStruct — Java Bean Mapping. URL: https://mapstruct.org/ (дата звернення: 30.04.2025).

18. Spring Security Reference | Spring. URL: https://docs.spring.io/spring-security/reference/index.html (дата звернення: 10.05.2025).

19. Introduction to JSON Web Tokens | JWT.io. URL: https://jwt.io/introduction (дата звернення: 12.05.2025).

# ДОДАТКИ

# ДОДАТОК А

## Код серверної частини web-сервісу для інтернет-магазину косметики

**Product.java**

```
package lushglow.com.entity;
import jakarta.persistence.*;
import java.math.BigDecimal;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;

@Entity
@Table(name = "products")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

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

    @Column(name = "description")
    private String description;

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

    @Column(name = "stock_quantity",nullable = false)
    private int stockQuantity;

    @ManyToOne
    @JoinColumn(name = "category_id")
    private Category category;
    @ManyToOne
```

```
   @JoinColumn(name = "brand_id")
    private Brand brand;

    @Column(name = "created_at", nullable = false)
    private LocalDateTime createdAt = LocalDateTime.now();

    @OneToMany(mappedBy = "product", cascade = CascadeType.ALL,
orphanRemoval = true)
    private List<AttributeValue> attributes = new ArrayList<>();

    @OneToMany(mappedBy = "product", cascade = CascadeType.ALL,
orphanRemoval = true,fetch = FetchType.EAGER)
    private List<ProductImage> images = new ArrayList<>();

    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    public BigDecimal getPrice() {
        return price;
    }
    public void setPrice(BigDecimal price) {
```

```
  this.price = price;
}
public int getStockQuantity() {
    return stockQuantity;
}
public void setStockQuantity(int stockQuantity) {
    this.stockQuantity = stockQuantity;
}
public Category getCategory() {
    return category;
}
public void setCategory(Category category) {
    this.category = category;
}
public Brand getBrand() {
    return brand;
}
public void setBrand(Brand brand) {
    this.brand = brand;
}
public LocalDateTime getCreatedAt() {
    return createdAt;
}

public void setCreatedAt(LocalDateTime createdAt) {
    this.createdAt = createdAt;
}
public List<AttributeValue> getAttributes() {
    return attributes;
}
public void setAttributes(List<AttributeValue> attributes) {
    this.attributes = attributes;
}
public List<ProductImage> getImages() {
    return images;
}
```

```
    public void setImages(List<ProductImage> images) {
        this.images = images;
    }
}
```

**ProductDTO.java**

```
package lushglow.com.dto;
import org.springframework.web.multipart.MultipartFile;

import java.math.BigDecimal;
import java.util.List;

public class ProductDTO {

    private Long id;
    private String name;
    private String description;
    private BigDecimal price;
    private Integer stockQuantity;
    private Long categoryId;
    private String categoryName;
    private Long brandId;
    private String brandName;
    private List<ProductImageDTO> images;
    private List<AttributeValueDto> attributes;
    private List<Long> removedImageIds;
    private List<MultipartFile> newImages;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }
```

```
public BigDecimal getPrice() {
    return price;
}

public void setPrice(BigDecimal price) {
    this.price = price;
}

public Integer getStockQuantity() {
    return stockQuantity;
}

public void setStockQuantity(Integer stockQuantity) {
    this.stockQuantity = stockQuantity;
}

public Long getCategoryId() {
    return categoryId;
}

public void setCategoryId(Long categoryId) {
    this.categoryId = categoryId;
}

public String getBrandName() {
    return brandName;
}

public void setBrandName(String brandName) {
    this.brandName = brandName;
}
public Long getBrandId() {
    return brandId;
}

public void setBrandId(Long brandId) {
    this.brandId = brandId;
}

public String getCategoryName() {
    return categoryName;
}

public void setCategoryName(String categoryName) {
    this.categoryName = categoryName;
}

public List<ProductImageDTO> getImages() {
    return images;
}

public void setImages(List<ProductImageDTO> images) {
    this.images = images;
}
```

```
    public List<AttributeValueDto> getAttributes() {
        return attributes;
    }

    public void setAttributes(List<AttributeValueDto> attributes) {
        this.attributes = attributes;
    }

    public List<Long> getRemovedImageIds() {
        return removedImageIds;
    }

    public void setRemovedImageIds(List<Long> removedImageIds) {
        this.removedImageIds = removedImageIds;
    }

    public List<MultipartFile> getNewImages() {
        return newImages;
    }

    public void setNewImages(List<MultipartFile> newImages) {
        this.newImages = newImages;
    }
}
```

**ProductServiceImpl.java**

```
lushglow.com.service.impl;

import jakarta.transaction.Transactional;
import lushglow.com.dto.AttributeValueDto;
import lushglow.com.dto.ProductDTO;
import lushglow.com.dto.ProductFilterRequest;
import lushglow.com.dto.ProductImageDTO;
import lushglow.com.entity.*;
import lushglow.com.mapper.ProductMapper;
import lushglow.com.repository.*;
import lushglow.com.service.ProductService;
import lushglow.com.tools.ImageStorageService;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.stereotype.Service;
import jakarta.persistence.criteria.Predicate;
import org.springframework.web.multipart.MultipartFile;

import java.math.BigDecimal;
import java.util.*;
import java.util.stream.Collectors;

@Service
public class ProductServiceImpl implements ProductService {
    private final ProductRepository productRepository;
    private final ProductMapper productMapper;
    private final AttributeRepository attributeRepository;
    private final CategoryRepository categoryRepository;
```

```
    private final BrandRepository brandRepository;

    private final AttributeValueRepository attributeValueRepository;

    private final ImageStorageService imageStorageService;

    private ProductImageRepository productImageRepository;

    public ProductServiceImpl(ProductRepository productRepository,
                              ProductMapper productMapper,
                              AttributeRepository attributeRepository,
                              CategoryRepository categoryRepository,
                              BrandRepository           brandRepository,
AttributeValueRepository  attributeValueRepository,  ImageStorageService
imageStorageService, ProductImageRepository productImageRepository) {
        this.productRepository = productRepository;
        this.productMapper = productMapper;
        this.attributeRepository = attributeRepository;
        this.categoryRepository = categoryRepository;
        this.brandRepository = brandRepository;
        this.attributeValueRepository = attributeValueRepository;
        this.imageStorageService = imageStorageService;
        this.productImageRepository = productImageRepository;
    }
    @Override
    @Transactional
    public ProductDTO createProduct(ProductDTO productDTO) {

        if (productRepository.existsByName(productDTO.getName())) {
            throw new RuntimeException("Product with this name already
exists");
        }


        Product product = new Product();
        product.setName(productDTO.getName());
        product.setDescription(productDTO.getDescription());
        product.setPrice(productDTO.getPrice());
        product.setStockQuantity(productDTO.getStockQuantity());


        Category                        category                        =
categoryRepository.findById(productDTO.getCategoryId())
                .orElseThrow(()  ->  new  RuntimeException("Category  not
found"));
        product.setCategory(category);


        Brand brand = brandRepository.findById(productDTO.getBrandId())
                .orElseThrow(()   ->   new   RuntimeException("Brand   not
found"));
        product.setBrand(brand);

        List<AttributeValueDto>              attributeDtos              =
productDTO.getAttributes() != null
```

```
                ? productDTO.getAttributes()
                : new ArrayList<>();

        Set<Long> seenAttributeIds = new HashSet<>();
        for (AttributeValueDto dto : attributeDtos) {
            if (!seenAttributeIds.add(dto.getAttributeId())) {
                throw new RuntimeException("Duplicate attribute detected:
attributeId=" + dto.getAttributeId());
            }
        }

        List<AttributeValue> attributeValues = new ArrayList<>();

        for (AttributeValueDto dto : attributeDtos) {
            Long attributeId = dto.getAttributeId();
            String value = dto.getValue();

            Attribute                    attribute                    =
attributeRepository.findById(attributeId)
                    .orElseThrow(()  ->  new  RuntimeException("Attribute
not found with id: " + attributeId));

            AttributeValue attributeValue = new AttributeValue();
            attributeValue.setAttribute(attribute);
            attributeValue.setProduct(product);
            attributeValue.setValue(value);

            attributeValues.add(attributeValue);
        }

        product.setAttributes(attributeValues);
        Product saved = productRepository.save(product);

        if (productDTO.getNewImages() != null) {
            for (MultipartFile file : productDTO.getNewImages()) {
                if (!file.isEmpty()) {
                    // 1) Зберігаємо файл на диск і отримуємо ProductImage
                    ProductImage                image                    =
imageStorageService.storeImage(file);
                    image.setProduct(saved);
                    // 2) Зберігаємо у БД
                    productImageRepository.save(image);
                }
            }
        }

        ProductDTO savedDto = productMapper.toDto(saved);

savedDto.setAttributes(productMapper.mapAttributesToDto(saved.getAttribu
tes()));
        return savedDto;
    }

    @Override
    @Transactional
```

```
    public ProductDTO updateProduct(Long id, ProductDTO productDTO) {
        Product existing = productRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Product not
found"));


        existing.setName(productDTO.getName());
        existing.setDescription(productDTO.getDescription());
        existing.setPrice(productDTO.getPrice());
        existing.setStockQuantity(productDTO.getStockQuantity());


        Category category =
categoryRepository.findById(productDTO.getCategoryId())
                .orElseThrow(() -> new RuntimeException("Category not
found"));
        existing.setCategory(category);


        Brand brand = brandRepository.findById(productDTO.getBrandId())
                .orElseThrow(() -> new RuntimeException("Brand not
found"));
        existing.setBrand(brand);

        List<AttributeValueDto> attributeDtos =
productDTO.getAttributes() != null
                ? productDTO.getAttributes()
                : new ArrayList<>();

        Map<Long, AttributeValue> existingAttributeMap =
existing.getAttributes().stream()
                .collect(Collectors.toMap(av ->
av.getAttribute().getId(), av -> av));

        List<AttributeValue> finalAttributes = new ArrayList<>();

        for (AttributeValueDto dto : attributeDtos) {
            Long attributeId = dto.getAttributeId();
            String value = dto.getValue();

            Attribute attribute =
attributeRepository.findById(attributeId)
                    .orElseThrow(() -> new RuntimeException("Attribute
not found with id: " + attributeId));

            AttributeValue attributeValue;

            if (existingAttributeMap.containsKey(attributeId)) {
                // Оновлюємо існуючий
                attributeValue = existingAttributeMap.get(attributeId);
                attributeValue.setValue(value);
            } else {
                // Новий
                attributeValue = new AttributeValue();
                attributeValue.setAttribute(attribute);
```

```
                attributeValue.setProduct(existing);
                attributeValue.setValue(value);
            }

            finalAttributes.add(attributeValue);
        }

        existing.getAttributes().clear();
        existing.getAttributes().addAll(finalAttributes);

        if (productDTO.getRemovedImageIds() != null) {
            for (Long imageId : productDTO.getRemovedImageIds()) {
                ProductImage                    image                    =
productImageRepository.findById(imageId).orElse(null);
                if (image != null) {
                    imageStorageService.deleteImage(image);
                    productImageRepository.deleteById(imageId);
                }
            }
        }
        if (productDTO.getNewImages() != null) {
            for (MultipartFile file : productDTO.getNewImages()) {
                if (!file.isEmpty()) {
                    ProductImage               savedImage               =
imageStorageService.storeImage(file);
                    savedImage.setProduct(existing);
                    productImageRepository.save(savedImage);
                    existing.getImages().add(savedImage);
                }
            }
        }

        Product saved = productRepository.save(existing);

        ProductDTO dto = productMapper.toDto(saved);

dto.setAttributes(productMapper.mapAttributesToDto(saved.getAttributes()
));

        return dto;
    }

    @Override
public List<ProductDTO> getTop4LowestStockProducts() {
    List<Product>                          products                         =
productRepository.findTop4ByOrderByStockQuantityAsc();

    List<ProductDTO> productDTOs = products.stream()
            .map(product -> {
                ProductDTO dto = productMapper.toDto(product);

                // Додаємо тільки зображення
                List<ProductImageDTO>              images               =
productMapper.mapImagesToDto(product.getImages());
                dto.setImages(images);
```

```
                dto.setAttributes(Collections.emptyList());

                return dto;
            })
            .collect(Collectors.toList());

    return productDTOs;
}
    @Override
    public List<ProductDTO> getNewAddedProducts() {
        List<Product> products = productRepository.findTop4WithImages();

        List<Product> last4 = products.stream()
                .sorted(Comparator.comparing(Product::getCreatedAt))
                .limit(4)
                .collect(Collectors.toList());

        List<ProductDTO> productDTOs = last4.stream()
                .map(product -> {
                    ProductDTO dto = productMapper.toDto(product);
                    List<AttributeValueDto>          attributes          =
productMapper.mapAttributesToDto(product.getAttributes());
                    dto.setAttributes(attributes);

                    List<ProductImageDTO>             images             =
productMapper.mapImagesToDto(product.getImages());
                    System.out.println(product.getImages());
                    dto.setImages(images);

                    return dto;
                })
                .collect(Collectors.toList());

        return productDTOs;
    }
@Override
public List<ProductDTO> getProductsByBrandName(String brandName) {
    List<Product>                          products                          =
productRepository.findTop4ByBrandName(brandName);

    List<ProductDTO> productDTOs = products.stream()
            .map(product -> {
                ProductDTO dto = productMapper.toDto(product);
                List<ProductImageDTO>               images              =
productMapper.mapImagesToDto(product.getImages());
                dto.setImages(images);

                dto.setAttributes(Collections.emptyList());

                return dto;
            })
            .collect(Collectors.toList());

    return productDTOs;
}
```

```
    @Override
    public ProductDTO getProductById(Long id) {
        Product product = productRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Product not
found"));
        ProductDTO dto = productMapper.toDto(product);
        List<AttributeValueDto> attributes =
productMapper.mapAttributesToDto(product.getAttributes());
        dto.setAttributes(attributes);
        List<ProductImageDTO> images =
productMapper.mapImagesToDto(product.getImages());
        System.out.println(product.getImages());
        dto.setImages(images);
        return dto;
    }

    @Override
    public List<ProductDTO> getAllProducts() {
        return productRepository.findAll().stream().map(p -> {
            ProductDTO dto = productMapper.toDto(p);

dto.setAttributes(productMapper.mapAttributesToDto(p.getAttributes()));
            return dto;
        }).collect(Collectors.toList());
    }
    @Override
    public boolean deleteProduct(Long id) {
        if (!productRepository.existsById(id)) {
            return false;
        }
        productRepository.deleteById(id);
        return true;
    }


    @Override
    public List<ProductDTO> getProductsByCategory(Long categoryId) {
        return productRepository.findByCategoryId(categoryId).stream()
                .map(product -> {
                    ProductDTO productDTO = productMapper.toDto(product);
                    List<AttributeValueDto> attributes =
productMapper.mapAttributesToDto(product.getAttributes());
                    productDTO.setAttributes(attributes);
                    List<ProductImageDTO> images =
productMapper.mapImagesToDto(product.getImages());
                    productDTO.setImages(images);
                    return productDTO;
                })
                .collect(Collectors.toList());
    }

    @Override
    public List<ProductDTO> getProductsByCategory(Long categoryId, int
offset, int limit) {
        Pageable pageable = PageRequest.of(offset / limit, limit);
```

```
     return  productRepository.findByCategoryIdWithPagination(categoryId,
pageable).stream()
                .map(product -> {
                    ProductDTO productDTO = productMapper.toDto(product);
                    List<AttributeValueDto>          attributes          =
productMapper.mapAttributesToDto(product.getAttributes());
                    productDTO.setAttributes(attributes);
                    List<ProductImageDTO>             images             =
productMapper.mapImagesToDto(product.getImages());
                    productDTO.setImages(images);
                    return productDTO;
                })
                .collect(Collectors.toList());
    }

    @Override
    public    List<ProductDTO>    filterProductsByCategoryAndFilters(Long
categoryId, ProductFilterRequest filters) {
        List<Product> products = productRepository.findAll((root, query,
cb) -> {
            List<Predicate> predicates = new ArrayList<>();


            predicates.add(cb.equal(root.get("category").get("id"),
categoryId));

            if        (filters.getBrands()        !=        null        &&
!filters.getBrands().isEmpty()) {

predicates.add(root.get("brand").get("name").in(filters.getBrands()));
            }

            if        (filters.getMinPrice()        !=        null        &&
!filters.getMinPrice().isEmpty()) {

predicates.add(cb.greaterThanOrEqualTo(root.get("price"),             new
BigDecimal(filters.getMinPrice())));
            }
            if        (filters.getMaxPrice()        !=        null        &&
!filters.getMaxPrice().isEmpty()) {
                predicates.add(cb.lessThanOrEqualTo(root.get("price"),
new BigDecimal(filters.getMaxPrice())));
            }

            if        (filters.getStock()        !=        null        &&
!filters.getStock().isEmpty()) {
                List<Predicate> stockPredicates = new ArrayList<>();
                for (String stock : filters.getStock()) {
                    switch (stock) {
                        case                  "1-10"                  ->
stockPredicates.add(cb.between(root.get("quantity"), 1, 10));
                        case                 "11-50"                  ->
stockPredicates.add(cb.between(root.get("quantity"), 11, 50));
                        case                  "50+"                   ->
stockPredicates.add(cb.greaterThanOrEqualTo(root.get("quantity"), 51));
```

```
                    }
                }
                predicates.add(cb.or(stockPredicates.toArray(new
Predicate[0])));
            }

            return cb.and(predicates.toArray(new Predicate[0]));
        });

        if (filters.getAttributes() != null &&
!filters.getAttributes().isEmpty()) {
            products = products.stream()
                    .filter(product -> {
                        Map<String, List<String>> attrMap =
filters.getAttributes();
                        Map<String, List<String>> productAttrs = new
HashMap<>();

                        for (AttributeValue value :
product.getAttributes()) {
                            String name = value.getAttribute().getName();
                            productAttrs.putIfAbsent(name, new
ArrayList<>());

productAttrs.get(name).add(value.getValue());
                        }

                        return attrMap.entrySet().stream()
                                .allMatch(entry -> {
                                    String attrName = entry.getKey();
                                    List<String> selectedValues =
entry.getValue();
                                    List<String> productValues =
productAttrs.get(attrName);
                                    return productValues != null &&

productValues.stream().anyMatch(selectedValues::contains);
                                });
                    })
                    .toList();
        }

        return products.stream()
                .map(product -> {
                    ProductDTO productDTO = productMapper.toDto(product);
                    List<AttributeValueDto> attributes =
productMapper.mapAttributesToDto(product.getAttributes());
                    productDTO.setAttributes(attributes);
                    List<ProductImageDTO> images =
productMapper.mapImagesToDto(product.getImages());
                    productDTO.setImages(images);
                    return productDTO;
                })
                .toList();
    }
```

```
public      List<ProductDTO>      getLatestProductsFromSameCategory(Long
categoryId, Long excludedProductId) {
        List<Product>                        products                        =
productRepository.findTop4ByCategoryIdAndIdNotOrderByCreatedAtDesc(categ
oryId, excludedProductId);

        return  products.stream()
                .map(product -> {
                    ProductDTO productDTO = productMapper.toDto(product);
                    List<ProductImageDTO>            images            =
productMapper.mapImagesToDto(product.getImages());
                    productDTO.setImages(images);
                    return productDTO;
                })
                .toList();
    }


}
```

**ProductRepository.java**

```
package lushglow.com.repository;
import lushglow.com.entity.Product;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.EntityGraph;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.JpaSpecificationExecutor;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;


import java.util.List;

@Repository
public interface ProductRepository extends JpaRepository<Product,Long>,
JpaSpecificationExecutor<Product> {

    @Query("SELECT DISTINCT p FROM Product p LEFT JOIN FETCH p.images ORDER
BY p.createdAt DESC")
    List<Product> findTop4WithImages();
    List<Product>   findTop4ByCategoryIdAndIdNotOrderByCreatedAtDesc(Long
categoryId, Long excludedProductId);

    List<Product> findTop4ByOrderByStockQuantityAsc();
    List<Product> findTop4ByBrandName(String name);

    @Query("SELECT p FROM Product p WHERE p.category.id = :categoryId ORDER
BY p.id ASC")
    List<Product>     findByCategoryIdWithPagination(@Param("categoryId")
Long categoryId, Pageable pageable);
    List<Product> findByCategoryId(Long categoryId);
    boolean existsByName(String name);
    boolean existsByBrandId(Long brandId);}
```

**ProductMapper.java**

```
package lushglow.com.mapper;

import lushglow.com.dto.AttributeValueDto;
import lushglow.com.dto.ProductDTO;
import lushglow.com.dto.ProductImageDTO;
import lushglow.com.entity.Attribute;
import lushglow.com.entity.AttributeValue;
import lushglow.com.entity.Product;
import lushglow.com.entity.ProductImage;
import org.mapstruct.Mapper;
import org.mapstruct.Mapping;
import org.mapstruct.factory.Mappers;
import org.springframework.stereotype.Component;

import java.util.List;
import java.util.stream.Collectors;

@Mapper(componentModel = "spring")
@Component
public interface ProductMapper {


    @Mapping(source = "category.id", target = "categoryId")
    @Mapping(source = "brand.id", target = "brandId")
    @Mapping(source = "brand.name", target = "brandName")
    @Mapping(source = "category.name", target = "categoryName")
    @Mapping(target = "attributes", ignore = true)
    @Mapping(target = "images", ignore = true)     images
    ProductDTO toDto(Product product);

    @Mapping(source = "categoryId", target = "category.id")
    @Mapping(source = "brandId", target = "brand.id")
    @Mapping(source = "brandName", target = "brand.name")
    @Mapping(source = "categoryName", target = "category.name")
    @Mapping(target = "attributes", ignore = true)
    @Mapping(target = "images", ignore = true)
    Product toEntity(ProductDTO productDto);

    default                                          List<AttributeValueDto>
mapAttributesToDto(List<AttributeValue> attributes) {
        if (attributes == null) return null;
        return attributes.stream().map(attr -> {
            AttributeValueDto dto = new AttributeValueDto();
            dto.setAttributeId(attr.getAttribute().getId());
            dto.setAttributeName(attr.getAttribute().getName());
            dto.setValue(attr.getValue());
            return dto;
        }).collect(Collectors.toList());
    }

    defaultList<AttributeValue>
zmapAttributesToEntity(List<AttributeValueDto>       attributeValueDtos,
Product product) {
```

```
        if (attributeValueDtos == null) return null;
        return attributeValueDtos.stream().map(dto -> {
            AttributeValue attributeValue = new AttributeValue();
            Attribute attribute = new Attribute();
            attribute.setName(dto.getAttributeName());
            attributeValue.setAttribute(attribute);
            attributeValue.setValue(dto.getValue());
            attributeValue.setProduct(product);
            return attributeValue;
        }).collect(Collectors.toList());
    }
    default    List<ProductImageDTO>    mapImagesToDto(List<ProductImage>
images) {
        if (images == null) return null;
        return images.stream().map(image -> {
            ProductImageDTO dto = new ProductImageDTO();
            dto.setId(image.getId());
            dto.setImageUrl(image.getImageUrl());
            dto.setProductId(image.getProduct()      !=      null      ?
image.getProduct().getId() : null);
            return dto;
        }).collect(Collectors.toList());
    }

    default  List<ProductImage>  mapImagesToEntity(List<ProductImageDTO>
dtos, Product product) {
        if (dtos == null) return null;
        return dtos.stream().map(dto -> {
            ProductImage image = new ProductImage();
            image.setId(dto.getId());
            image.setImageUrl(dto.getImageUrl());
            image.setProduct(product);
            return image;
        }).collect(Collectors.toList());
    }
```