Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

РОЗРОБКА ВЕБ-ЗАСТОСУНКУ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЙ SPRING BOOT

Текстова частина до курсової роботи
за спеціальністю „Інженерія програмного забезпечення” 121

Керівник курсової роботи
ст.викладач Борозенний С.О.
_____
(підпис)
“____” _____2023 р.
Виконав студент
Марченко В.О.
“____” _____2023 р.

Київ 2024

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Старший викладач

_____ Борозенний С.О.

(підпис)

“____” _____2024 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту Марченко Владиславу Олеговичу факультету інформатики 3 курсу

ТЕМА: Розробка веб-застосунку з використанням технологій Spring Boot

Зміст ТЧ до курсової роботи:

Індивідуальне завдання

Календарний план

Зміст

Анотація

Вступ

1 Аналіз наявних рішень

2 Архітектура застосунку

Висновки

Список використаних джерел

Дата видачі „___” _____ 2023 р. Керівник _____

(підпис)

Завдання отримав _____

(підпис)

Тема: Розробка веб-застосунку з використанням технологій Spring Boot

Календарний план виконання роботи:

№	Назва етапу	Термін виконання	Примітка
1.	Отримання теми курсової	10.04.2024	
2.	Аналіз тематичної літератури	16.04.2024	
3.	Написання програмної частини	29.04.2024	
4.	Оформлення макету текстової частини	05.05.2024	
5.	Написання основної текстової частини	12.05.2024	
6.	Створення презентації	13.05.2024	
7.	Завантаження роботи	14.05.2024	

Студент Марченко В.О.

Керівник Борозенний С.О.

“_____” _____

Зміст

Анотація

Ця курсова робота присвячена розробці веб-застосунку з використанням технологій фреймворку Spring Boot для розробки серверної (Back-end) частини програми. Для Front-end частини застосунку було обрано JavaScript бібліотеку React, із використанням найбільш нових тенденцій. У роботі розглядається весь процес розробки самого веб-застосунку. Особлива увага приділяється етапам планування бази даних та обранню архітектури.

Результатом цієї роботи є функціональний інтернет магазин для продажу та купівлі клієнтами виставлених товарів, зі зручним інтерфейсом, надаючи можливість зручно користуватися запропонованим функціоналом. Дані користувачів і їх облікових записів є захищеними від доступу до даних на серверній стороні і доступні лише авторизованим власникам.

Для створення бази даних, серверної і клієнтської частин використано:

MySQL Server v8.0

Spring Boot v3.2.5

React v18.2.0

Ключові слова: веб, сайт, Spring Boot, React, Redux Toolkit, MySQL, клієнт, сервер, Jwt, аутентифікація, авторизація.

Вступ

У сучасному світі, де інтернет є невід'ємною складовою фактично майже кожної людини у світі, є неабиякий попит на онлайн застосунки абсолютно різного характеру та з різних сфер життя. Це робить веб-застосунки надзвичайно затребуваними не тільки для користувачів, а й для бізнесів, які хочуть вийти на онлайн ринок. Однією з таких сфер є купівля-продаж різних товарів, до якої і відноситься розроблений застосунок. Зі збільшенням попиту на такий формат взаємодії з клієнтами вочевидь з'являється небезпека клієнтських даних, що зумовлюється неякісними продуктами у використанні, що породжує можливість втрати персональної інформації або шахрайства. Для уникнення цих проблем необхідно ставити високий пріоритет забезпеченню надійної системи безпеки у застосунку, що і є основною з ідей у запропонованому продукті.

Задля реалізації продукту, який відповідає всім вимогам використовуються технології, які є актуальними на сьогоднішній день. MySQL є найбільш використовуваною реляційною базою даних серед усіх, будучи дуже гнучкою в проектуванні та використанні. Spring Boot і React є інструментами, які є дуже гнучкими в розробці веб-застосунків через можливості використання сторонніх бібліотек під час розробки. Попри це, вони мають відкриті вихідні коди, що дозволяє їм розвиватися не тільки за рахунок їх розробників, а також завдяки спільноті, які сприяють розвитку цих продуктів, що є гарною практикою для довго перспективної підтримки інструментів на плаву.

Метою виконання цієї роботи було розробити такий застосунок, який відповідав би досить високій планці якості і вирішував би проблеми, перераховані вище, а також пошуку і дослідження сучасних принципів та тенденцій розробки клієнт серверних застосунків, тим самим поглибивши знання у цій сфері розробки, а також закріплення та застосування тих знань, які були здобуті мною за час навчання до виконання.

Робота поділяється на два основних розділи.

Перший розділ присвячено аналізу наявних інструментів розробки, їх порівнянню та огляду запропонованих рішень у існуючих застосунках. Додатково наведено ряд наявних веб-сайтів і проаналізовано стек (набір) програм і технологій, з використанням яких його розроблено.

У другому розділі продемонстровано алгоритм, за яким проходило планування проєкту в цілому, взаємодія його компонентів та обрані рішення для розробки. Продемонстровано структурну схему цілісного застосунку, та розміщенню компонентів усередині нього. Найбільше уваги приділяється обраній архітектурі застосунку, як у клієнтської частини, так і зі сторони

серверної. Таким чином розглядається, як саме впроваджено роботу програми та детально продемонстровано окремі її частини, з поясненнями до окремих рішень та з ілюстраціями коду й прикладами.

Об’єктом дослідження є веб-сайт, повністю написаний за допомогою технологій, наданих фреймворком Spring Boot, бібліотекою React, і з використанням реляційної бази даних MySQL.

Предметом дослідження є процес розробки веб-застосунків за допомогою інструментів, наданих фреймворками, а також обрання правильної архітектури при розробці додатків й використання гарних і корисних практик при написанні коду.

Розділ 1. АНАЛІЗ НАЯВНИХ РІШЕНЬ

1.1 Огляд найбільш популярних інструментів розробки

На сьогоднішній день веб-розробка є чи не найпоширенішої сферою в IT та розробці різноманітних застосунків, тому інструменти та технології неперервно розвиваються, надаючи розробникам можливості для створення продуктів з максимальним спектром. У нашому випадку, мова йде про клієнт-серверні застосунки, і при виборі інструментів для створення клієнтської частини (Front-end) одними з найбільш популярних є JavaScript бібліотека React та фреймворки Angular та Vue.js. Окрім цих трьох, існує ще декілька фреймворків, які інколи використовують розробники, до прикладу, Svelte, але кратно програє іншим варіантам.

У моєму випадку, для розробки клієнтської частини було вибрано фреймворк React, який сумісно посідає перше місце серед розробників у своїй галузі, при чому за статистикою на 2023 рік він випереджає досить сильно, будучи обраним 40% розробників у своїх проєктах, у той час як Angular обирають 17%, а Vue.js – 16% відповідно [1]. React обирає більшість через зручність написання коду компонентами, а також за наявність великої кількості бібліотек та плагінів. Angular у свою чергу приваблює широким функціоналом, надаючи інструменти для роботи з маршрутизацією та іншими частинами застосунків. Vue.js обирають через простоту, і тому він є популярним серед новачків, завдяки чому використовується багато де.

У сфері серверної (Back-end) розробки ситуація цікавіша, адже є трохи більше конкурентоспроможних фреймворків і інструментів, які обирають і вподобають розробники по всьому світі, про що кажуть опитування та статистика. Як вже було сказано, у клієнтській частині є один чіткий лідер із 40% користувачів за використанням. Серед інструментів для серверних додатків також є лідер, за статистикою на 2023 рік, у 42% випадках використовується фреймворк Node.js. За різними даними фреймворк Spring Boot використовують у 12-15% випадків, який випереджають такі інструменти як ASP.NET, Express, і вже згаданий Node.js [2, 7]. Досить цікаво, що за статистикою на листопад 2023 року Spring Boot посідає третє місце після Laravel і Django за кількістю зірок на Git платформах, при цьому маючи приблизно схожі результати, що каже про серйозну базу користувачів.

Spring Boot – це фреймворк для побудови Java-додатків, для подальшого розгортання і використанні у веб розробці. Хоч він і не займає першого місця серед всіх можливих інструментів, для цієї роботи було обрано саме його через високу ефективність, надійність та розширюваність (можливість масштабування).

1.2 Існуючі веб-платформи

На сьогоднішній день React використовують майже у будь-якому куточку мережі для реалізації своїх продуктів. Неможливо заперечувати актуальність, популярність та розповсюдженість даного інструменту, адже з його допомогою функціонує немала кількість гігантів, таких як Facebook, Instagram, Airbnb, тощо. Використання можна зустріти абсолютно в будь-яких сферах: якщо зайти на сайти таких брендів як Volvo, Lamborghini, Twitch, Spotify, завжди у стеку технологій побачимо наявний React як основа клієнтської частини.

З іншого боку, якщо подивитися на готові продукти, які використовують Spring Boot, можна знайти чимало результатів, які повністю або частково виконані за допомогою даного інструменту. Серед найбільших і відомих компаній – це AliExpress, LinkedIn, Alibaba, Ticketmaster та інші, менш відомі, яких також достатньо.

Розділ 2. АРХІТЕКТУРА ЗАСТОСУНКУ

2.1 Структурна схема

Правильно спланована структура веб застосунку є невід'ємною складовою його успішного й ефективного функціонування, особливо якщо ми говоримо про застосунок, який розгорнутий у декілька частин. Чітка архітектура завжди є основою для масштабованості будь-якого продукту, що надзвичайно важливо для розвитку впродовж тривалого періоду та для великих компаній. Першим і основним прикладом є розділення застосунку на клієнтську та серверну частини, що дозволяє відокремити інтерфейс для користувача від бізнес логіки, що надзвичайно спрощує написання коду, якщо мова йде про велику команду розробників. Підтримка такої опції також є однозначно легшою в перспективі.

Ефективна комунікація між клієнтською та серверною частиною, включаючи способи взаємодії з базою даних зі сторони серверу є ключем для якісного результату роботи. Використання REST API для передачі даних між клієнтом і сервером дозволяє забезпечити плавну та безперервну роботу застосунку, навіть при великому навантаженні. Швидка і коректна комунікація серверу з базою даних, як до прикладу запити SQL, дозволяє підтримувати високу швидкодію та ефективність збереження та обробки даних. Структура розробленого застосунку представляє собою React частину, яка комунікує завдяки REST API з частиною Spring Boot (Back-end), яка в свою чергу обробляє отримані дані, відповідає за бізнес-логіку, а також за комунікацію і збереження даних у MySQL реляційну базу даних, яка в свою чергу окремо запущена в мережі, і функціонує незалежно від роботи інших частин застосунку (рисунок 2.1).

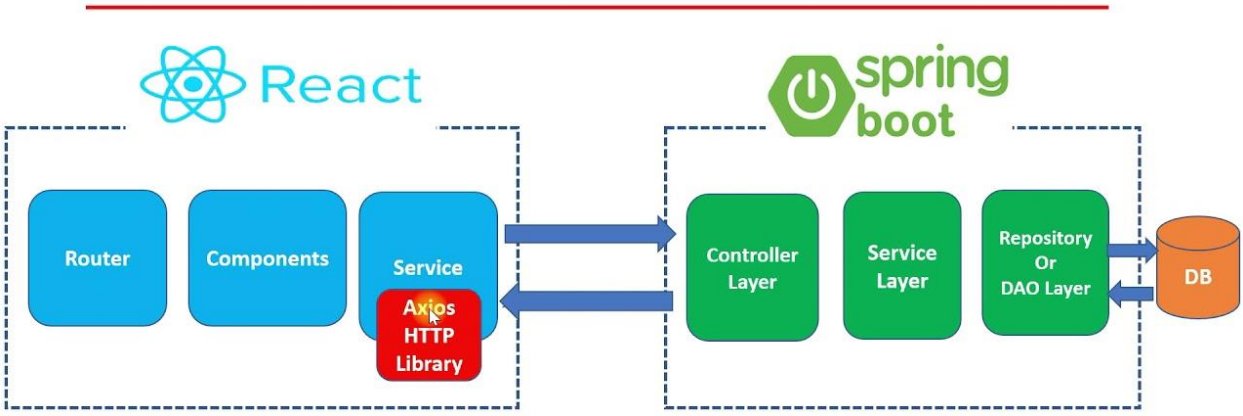


Рисунок 2.1 – Структурна схема застосунку

2.2 Проєктування бази даних

2.2.1 План створення бази даних

Перед тим, як переходити до безпосередньої розробки застосунку, завжди треба спроєктувати базу даних, яка буде основою, на яку все спирається. База даних є першочерговим етапом у розробці, адже її проєктування неможливе лише створенням таблиць, перед цим необхідні кроки – це аналіз галузі, з якою працює застосунок, визначення сутностей, які мають бути присутні в бізнес-логіці продукту, їх зв’язків та взаємодії, а також накладання на ці сутності після їх створення обмежень, що визначають певні особливості предметної галузі. Після даних кроків необхідним також є нормалізація бази даних як мінімум до третьої нормальної форми для запобігання різним проблемам таких як аномалії видалення, редагування, вставки та інших. Останнім формальним кроком буде заповнення бази даних даними, які буде відображати застосунок на клієнтській стороні – це відносно нескладно зробити за допомогою запитів мови SQL.

2.2.2 Аналіз галузі

Дана галузь, або предметна область, з якою має працювати застосунок – це інтернет-магазин для розміщення там для продажу товарів (у нашому випадку меблів), та купівлі цих товарів користувачами шляхом оформлення замовлень на сайті, попередньо зареєструвавшись і увійшовши у свій обліковий запис.

2.2.3 Сутності

Для сутностей створено окремі таблиці для візуалізації полів, які вони будуть містити.

- Сутність користувача або ж його облікового запису, яка зберігає всі його дані, серед яких обов’язкові: ім’я, прізвище, пошта, телефон, пароль, роль; а також необов’язкові: по-батькові та дата народження. Відповідна таблиця з ключем user_id (рисунок 2.2).

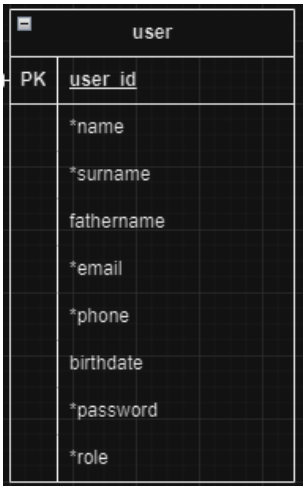


Рисунок 2.2 – Таблиця user

- Сутність категорії, які в подальшому будуть мати підкатегорії, до яких вже будуть відноситися самі товари магазину. Зберігає первинний ключ category_id, а також обов’язкові поля назва, опис і картинка (рисунок 2.3).

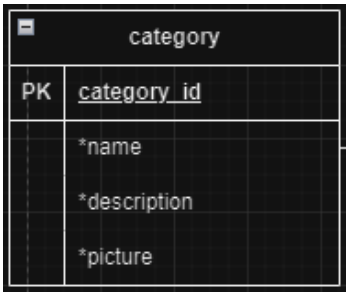


Рисунок 2.3 – Таблиця category

- Сутність підкатегорії, які містять продукти, а також є підкатегорією безпосередньо категорії (category). Містить первинний ключ subcategory_id, а також обов’язкове поле назва, і зовнішній ключ-посилання на категорію (рисунок 2.4).

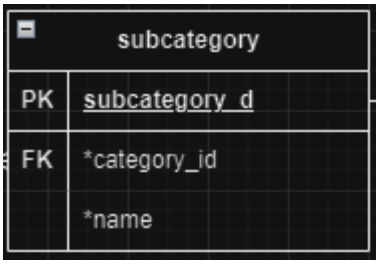


Рисунок 2.4 – Таблиця subcategory

- Сутність картинки продукту, яка відноситься відповідно до конкретного продукту (product). Містить первинний ключ picture_id, а також обов’язкове поле picture, і зовнішній ключ-посилання на продукт (рисунок 2.5).

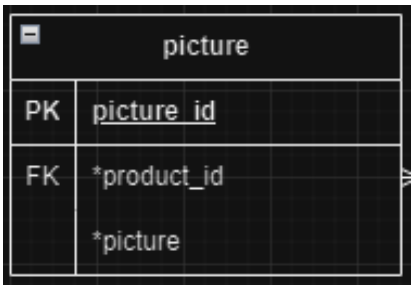


Рисунок 2.5 – Таблиця picture

- Сутність кольору продукту, яка містить первинний ключ color_id, а також обов’язкові поля назва та hex код (рисунок 2.6).

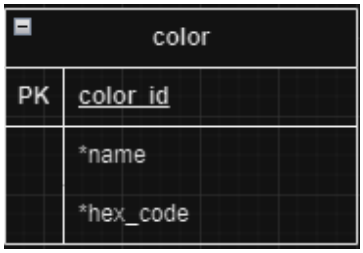


Рисунок 2.6 – Таблиця color

- Сутність матеріалу продукту, яка містить первинний ключ material_id, і також обов’язкове поле назва (рисунок 2.7).

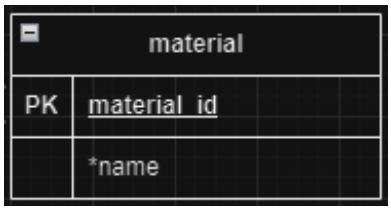


Рисунок 2.7 – Таблиця material

- Сутність продукту, який відноситься до певної підкатегорії (subcategory), має певну кількість картинок (picture), які йому належать, має певну кількість кольорів (color), певну кількість матеріалів (material), а також міститься в майбутньому в замовленнях. Містить первинний ключ product_id, зовнішній ключ-посилання на підкатегорія, а також обов’язкові поля назва, опис, ціна за одиницю, висота, довжина, ширина, вага і опис пакунку (рисунок 2.8).

Рисунок 2.8 – Таблиця product

- Сутність замовлення, яке відноситься до певного користувача (user), а також містить рядки замовлення з інформацією про продукт і кількість. Містить первинний ключ order_id, а також обов’язкові поля дата/час замовлення, повна ціна, статус замовлення (рисунок 2.9).

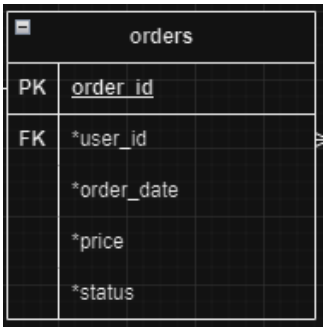


Рисунок 2.9 – Таблиця orders

- Сутність рядок замовлення, або ж таблиця для зв’язку замовлення і продуктів, яка містить складений первинний ключ order_id/product_id, а також обов’язкові поля кількість продукту в замовленні, а також обов’язкові обраний колір товару, і обраний матеріал товару (рисунок 2.10).

Рисунок 2.10 – Таблиця order_product

- Сутність-зв’язок між таблицями продуктів і кольорів, яка містить складений первинний ключ product_id/color_id (рисунок 2.11).

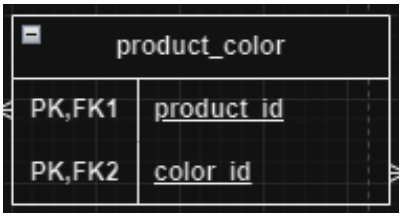


Рисунок 2.11 – Таблиця product_color

- Сутність-зв’язок між таблицями продуктів і матеріалів, яка містить складений первинний ключ product_id/material_id (рисунок 2.12).

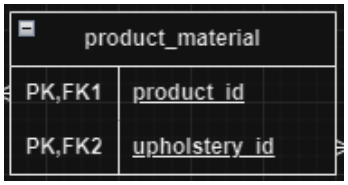


Рисунок 2.12 – Таблиця product_material

2.2.4 Обмеження, зв’язки та нормалізація

Перед тим, як завершити проєктування бази даних необхідно впровадити обмеження щодо наявних сутностей відповідно до предметної області [5]. У нашому випадку основними обмеженнями будуть запроваджені SQL обмеження на довжини рядків і подібні, і неформальні обмеження на формат таких полів, які будуть реалізовані програмно на серверному рівні перед збереженням даних: пошта і телефон користувача – у відповідному форматі, пароль – рядок довжиною рівно 72 символи (закодований за допомогою алгоритму BCrypt), а також hex код кольору повинен бути рядком у рівно шість символів латинського алфавіту або цифр.

Після цього необхідно сутностям, відповідно до їх відношень присвоїти безпосередні зв’язки між таблицями реляційної бази даних. Таким чином, ми будемо мати наступні зв’язки:

- category ONE TO MANY subcategory
- subcategory ONE TO MANY product
- product ONE TO MANY picture
- user ONE TO MANY orders
- product ONE TO MANY product_color
  color ONE TO MANY product_color
- product ONE TO MANY product_material
  material ONE TO MANY product_material
- product ONE TO MANY order_product
  order ONE TO MANY order_product

Де останні шість зв’язків репрезентують три MANY TO MANY зв’язки. Таким чином маємо наступну схему для реляційної бази даних застосунку на даному етапі (рисунок 2.13). Усі зв’язки вказані разом з уточненнями, чи є зв’язок обов’язковий, чи ні.

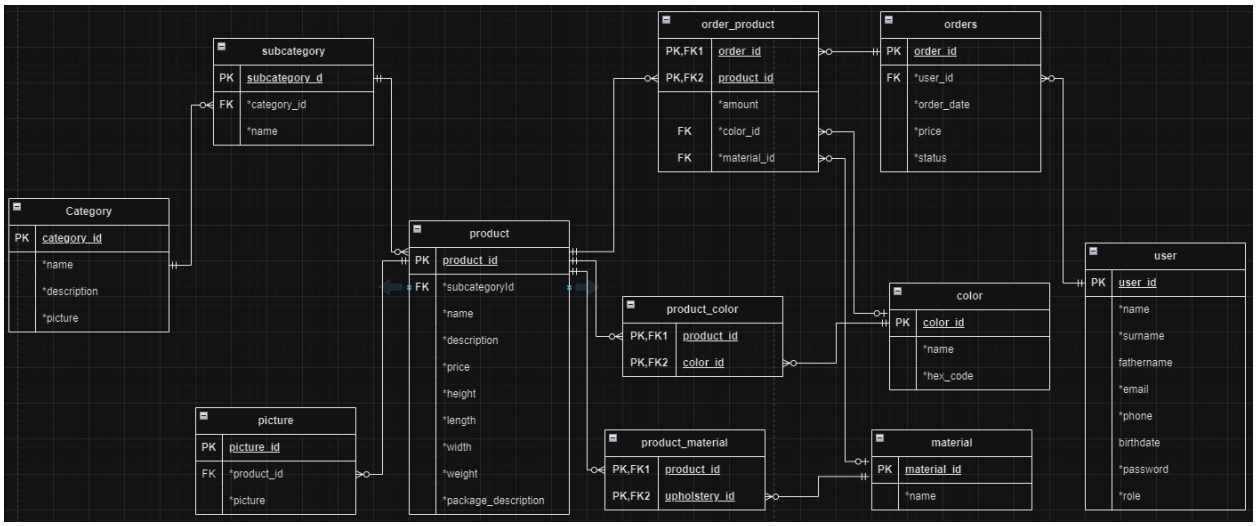


Рисунок 2.13 – Кінцева схема реляційної бази даних

Якщо спробувати провести нормалізацію даної схеми бази даних відповідно до всіх норм по черзі від першої нормальної форми до третьої, то виявиться, що дана схема вдало спроєктована і відповідає як мінімум третій нормальній формі [3], чого достатньо для відсутності аномалій і проблем у роботі 99% застосунків, а отже робимо висновок, що це можна вважати кінцевою схемою бази даних, яка може бути використана для використання у роботі майбутнього застосунку. Останній крок, який робиться – це розгортання відповідної бази даних за допомогою MySQL Server і безпосередньо запитів SQL для створення таблиць.

2.3 Back-end

2.3.1 Конфігурація

Щоб надалі працювати над написанням коду серверної частини за допомогою інструментів Spring Boot, важливо правильно налаштувати наше середовище та провести конфігурацію.

По-перше, необхідним є підключення до проєкту потрібних бібліотек і залежностей, як показано на рисунку 2.14. Spring Boot має велику кількість можливостей розширювати проєкти через систему збірки. Наразі у Spring Boot можна використовувати дві основні збірки для Java-проєктів – Maven та Gradle. Через них можна в конфігураційному файлі проєкту додавати ті залежності, бібліотеки яких надаватимуть необхідні інструменти для роботи в самому застосунку. У нашому випадку використовується система збірки Maven, через яку ми додаємо необхідні для серверної частини інструменти.

Серед них маємо такі як Spring Boot Data JPA, який дозволяє створювати класи мовою Java, які функціонують як репозиторії, або ж те, за допомогою чого ми можемо комунікувати з відповідною репозиторію таблицею з сутностями у реляційній базі даних SQL. Надалі це буде однією з ключових одиниць в архітектурі серверної частини. Окрім цього, надається не менш необхідна можливість створювати Java-класи, які відповідатимуть відповідній сутності в реляційній базі даних, які будуть взаємодіяти як єдине ціле з репозиторіями.

Надалі також одна з основних залежностей – це Spring Boot Web [4]. Не включивши її в структуру проєкту, жоден проєкт не матиме можливості коректно і повноцінно взаємодіяти з Rest запитами. Які надходять до завтосунку зовні і обробляти їх.

Spring Boot Validation. Даний модуль надає в користування інструменти, завдяки яким можна гнучко і зручно налаштовувати вже згадані раніше обмеження предметної області до будь-якої з визначених сутностей, що відносяться до бази даних шляхом Java-анотацій. Для нас ця можливість необхідна для правильного функціонування і збереження коректних даних, а також уникнення проблем, викладених у розділі 2.2.4.

Lombok. Підключення даного інструменту не є необхідним і вирішальним кроком, у даному випадку це надає нам інтсрументи, завдяки яким ми можемо полегшити для себе написання того коду, який не потребує якихось знань галузі, і що можна покласти на анотації, які надаються при підключенні Lombok.

Наступні три залежності можна об’єднати в одну групу – Spring Boot Security, Spring Security Test та Auth0 Java Jwt. Як можна здогадатися, при під’єднанні до проєкту цих трьох інструментів, надаються можливості для конфігурації та подальшого використання в застосунку механізмів авторизації та аутентифікації користувачів з великою кількістю можливих налаштувань [6]. У нашому випадку дані функції є обов’язковими, враховуючи всі цілі, які були поставлені перед нами.

2.3.2 Під’єднання до бази даних

Другим важливим кроком є вказати серверу необхідні параметри для того, щоб в подальшому він міг під’єднатися до створеної бази даних в мережі, та стягувати або передавати дані туди.

По-перше, для цього необхідно підключити останній необхідний нам інструмент у конфігурацію застосунку MySQL Java Connector (рисунок 2.14). Після цього ми можемо задавати серверу параметри для підключення бази даних, і з допомогою цього інструменту йому буде зрозуміло, як це зробити.

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.33</version>
</dependency>
```

Рисунок 2.14 – Підключення залежності (Maven)

Останнім кроком конфігурації серверної частини буде прописання усіх необхідних параметрів і налаштувань у спеціально призначеному файлі в ресурсах проєкту під назвою application.yml (іноді application.properties). У нашому випадку це будуть параметри підключення до бази даних, назва застосунку, а також секретний ключ для генерації в майбутньому Jwt токенів для аутентифікації користувачів (рисунок 2.15).

```yaml
spring:
  application:
    name: shop-springboot
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    password: pass
    url: jdbc:mysql://localhost:3306/furnituredb
    username: root
  jpa:
    open-in-view: true

security:
  jwt:
    token:
      secret-key: "jwtsecret"
```

Рисунок 2.15 – Конфігураційний файл

2.3.3 Визначення архітектури

Вибір правильної архітектури звичайно не завжди є обов'язковим для потрібного функціонування застосунку в цілому, проте цей етап є надзвичайно важливим при розробці будь-якого програмного забезпечення через низку інших причин, і може значно вплинути на багато аспектів у перспективі майбутнього [8].

У цій роботі, конкретно серверна частина була побудована на архітектурі під назвою "Model-View-Controller" (MVC). MVC – це шаблон проєктування коду в програмі, який розділяє її на три основні частини з відповідними назвами [9]. У нашому випадку було обрано триматися архітектури, яка дуже часто використовується розробниками при роботі над серверною частиною (Back-end). Трьома основними складовими програми будуть контролери (Controller), сервіси (Service) і вже згадані раніше репозиторії (Repository) [10]. У даному випадку контролери відповідатимуть за обробку та відправлення HTTP-запитів, тобто за зв'язок з зовнішнім світом, а саме з клієнтською частиною та обміном з нею даними. Сервіси будуть структурними одиницями, до яких оброблені дані передаватимуть контролери, і в яких відбуватиметься основна бізнес-логіка застосунку, тобто основна програмна частина буде саме там. Репозиторії відповідатимуть за комунікацію з базою даних і передачу туди даних. У архітектурі запропонованого застосунку також буде четверта складова – моделі (Model). Моделям відповідатимуть класи-сутності, які не нестимуть конкретної бізнес-логіки, а лише відповідатимуть за представлення даних, або ж просто сутностей, які використовуватимуться в додатку.

По-перше, розділення нашої логіки в додатку на окремі компоненти є гарною практикою, тому що це дозволяє підтримувати чистоту коду для розробника, що робить легшим робити зміни в коді та підлагоджувати його при потребі. Розділення завдань на компоненти відповідає принципам SOLID, що є гарною практикою в дизайні і об’єктно-орієнтованому програмуванні в цілому [11].

По-друге, така архітектура спрощує для розробників процес тестування додатку, адже оскільки компоненти розділені разом з логікою, тестування можливо проводити окремих компонентів, або їх груп, що дозволяє і логіку тестувати окремо і незалежно від інших частин коду.

Окрім зазначеного, архітектура MVC дозволяє зручно масштабувати додаток без нагромадження і зосередження значної частини коду в одному місці, адже для нового функціоналу при потребі можна не додавати його в існуючі компоненти, а навпаки – створити окремі компоненти саме під потрібні особливості, і таким чином розширювати програмне забезпечення не вглиб, а вшир.

Таким чином ми маємо архітектуру, яка виглядає гарно структурованою та дає низку переваг для застосунку через послідовність дій і відсутність хаосу: дані, які приходять ззовні у вигляді HTTP-запитів, передаються від контролерів до сервісів, де відбуваються основні дії з ними, результати яких передаються до репозиторіїв, і потім безпосередньо зберігаються в базі даних.

2.3.4 Впровадження архітектури

Після того, як були прийняті всі рішення щодо архітектури, можна приступати до створення класів для впровадження обраного дизайну таким чином, щоб триматися плану. У нашому випадку у нас заплановано провадити вісім повноцінних сутностей, і ще три сутності, які відповідають за Many to Many зв’язки між іншими, і вочевидь відповідну бізнес-логіку, яка відповідатиме за основні операції над сутностями. Таким чином, повноцінний шар типу Controller-Service-Repository потрібно впровадити для кожної повноцінної сутності, і як мінімум модель і репозиторій для сутностей-зв’язків, для яких скоріше не буде потребуватися частина контролерів, і можливо сервісів [12].

2.3.5 Модель

Розглянемо основні частини на прикладі повноцінної сутності категорій (category), про які згадували раніше. Будемо йти від низу до верху, щоб прослідкувати за залежностями між структурними компонентами між собою, тож почати необхідно з моделі – компоненту, який узагалі не залежить від інших частин архітектури і є самостійним класом, який не несе бізнес-логіки, і є представленням даних, у нашому випадку клас-сутність Category у коді застосунку (рисунок 2.16) відповідатиме таблиці category у базі даних MySQL.

```java
@Entity
@Getter
@Setter
@NoArgsConstructor
@Table(name = "category")
public class Category {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long categoryId;

    @Column(unique = true, nullable = false, length = 20)
    private String name;

    @Column(nullable = false, length = 150)
    private String description;

    @Column(nullable = false)
    private byte[] picture;

    @OneToMany(mappedBy = "category")
    private Set<Subcategory> subcategories = new HashSet<>();

    public Category(String name, String description) {
        this.name = name;
        this.description = description;
    }

    public int trendingIndex() {
        int index = 0;
        for (Subcategory subcategory : subcategories) {
            for (Product product : subcategory.getProducts()) {
                index += product.trendingIndex();
            }
        }
        return index;
    }
}
```

Рисунок 2.16 – Приклад моделі

Необхідними деталями для реалізації цього класу в даному випадку будуть анотація @Entity, яку надає нам Spring Boot JPA разом з @Table для вказування, що даний клас є представленням таблиці category з сутностями категорій в базі даних, для подальшого зв’язку через репозиторій. Наступною необхідною частиною будуть поля класу, при чому необхідно повністю визначити всі ті поля, які були вказані при створенні таблиці в базі даних, з відповідними типами для коректної конвертації. Одне з цих полів, а саме первинний ключ необхідно позначити відповідною анотацією для його

ідентифікації. Назви полів повинні або відповідати напряму колонці в базі даних, або назву колонки вказувати вручну напряму, якщо назва поля названа інакше. Останніми обов'язковими частинами є методи для роботи з класом і його приватними полями (Getters, Setters), і конструктори – це саме та частина коду, яку ми можемо покласти на раніше згаданий Lombok, завдяки наданим нам анотаціям для цього (див. рисунок 2.16).

За потреби запровадити до сутності обмеження, викликані предметною областю, про що було згадано в пункті 2.2.4, було використано інструменти надані Spring Boot Validation, а саме анотаціями, які потребують від полів сутностей бути відповідного формату і лише такого формату. У випадку категорій таких обмежень немає, це продемонстровано наглядно на прикладі полів пошти та номеру телефону сутності користувача (рисунок 2.17).

```
@Column(unique = true, nullable = false, length = 50)
@Email(message = "INVALID USER EMAIL FORMAT")
private String email;

@Column(unique = true, nullable = false, length = 15)
@Pattern(regexp = "^\\d{10,15}$", message = "INVALID USER PHONE FORMAT")
private String phone;
```

Рисунок 2.17 – Приклад обмеження полів сутності

При наявності у сутності зв'язків з іншими, а в нашому випадку, і в сутності категорій, і у всіх інших такі наявні, необхідно за потреби додатково вказувати поле з відповідними сутностями, з якими наявні зв'язки відповідного типу, в випадку категорій – це One to Many з сутностями підкатегорій (див. рисунок 2.16).

І остання частина, яка в майбутньому може бути потрібна для вже бізнес-логіки програми, це допоміжні методи класу, у випадку категорій, це один метод trending, який підраховує індекс категорії в рейтингу популярних на момент запиту категорій із усіх (див. рисунок 2.16).

2.3.6 Репозиторій

Наступною структурною одиницею в ланцюгу архітектури для кожної незалежної частини програми є репозиторій – клас, який залежить від раніше згаданої моделі, і відповідає за зв'язок з таблицею в базі даних, яка відповідає саме тій сутності, яка вказана як перший тип, за яким типізований репозиторій, і від якої він залежить. Класи-репозиторії мають найменше написаного коду, лише необхідну анотацію @Service, щоб програма

ідентифікувала правильно відповідні класи, бо основні операції над даними вже наявні в інтерфейсах, які нам надає Spring Boot JPA, в нашому випадку це інтерфейс CrudRepository (рисунок 2.18), який дозволяє виконувати базові операції додавання, редагування, видалення та знаходження даних в таблиці реляційної бази даних, до якої під'єднаний застосунок.

```java
package com.naukma.shopspringboot.category;

import com.naukma.shopspringboot.category.model.Category;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface CategoryRepo extends CrudRepository<Category, Long> {

}
```

Рисунок 2.18 – Приклад репозиторію

При необхідності є можливість додавати нові методи, які будуть відповідати за нові операції, і під капотом відповідати іншим SQL запитам до цієї самої таблиці, але більш складним, при чому генерація відповідного SQL запиту відбувається в залежності від назви створеного методу, які мають певні правила номенклатури. Такі можливості було використано, до прикладу, у репозиторії для сутності користувача (рисунок 2.19).

```java
@Repository
public interface UserRepo extends CrudRepository<User, Long> {
    Optional<User> findByPhoneEquals(String phone);

    Optional<User> findByEmailEquals(String email);
}
```

Рисунок 2.19 – Приклад додаткових методів у репозиторії

2.3.7 Сервіс

Третьою структурною одиницею після репозиторію йде сервіс – клас, анотований як @Service, і який не є обов'язковим в функціонуванні програми в цілому, але створений для розподілення бізнес-логіки від контролерів в окреме місце, щоб залишити на контролерах лише обов'язки обробки запитів. Таким чином, всередині сервісів містяться основні операції над даними, що по суті формує майже всю бізнес-логіку застосунку. Прийнято проєктувати код так, щоб сервіс був залежний від відповідного репозиторію, або ж від інших сервісів інших частин застосунку, якщо необхідно пов'язати

певні аспекти бізнес-логіки, або звернутися в певний момент для збереження стягнення або збереження відмінних від основної сутності, до якої прив'язана частина програми. У випадку категорій, у сервісі реалізовані методи, які відповідають за базові Crud операції, використовуючи репозиторій, а також такі методи, які відповідають за отримання картинки категорії, отримання всіх категорій списком, отримання підкатегорій, які їй належать, а також отримання списку перших N категорій у рейтингу за індексом популярності (рисунок 2.20).

```
@Service
public class CategoryService {
    private final CategoryRepo categoryRepo;

    public CategoryService(CategoryRepo categoryRepo) { this.categoryRepo = categoryRepo; }

    // BUSINESS LOGIC

    public byte[] getCategoryPicture(Long categoryId) {...}

    public List<CategoryDTO> getAllCategories() {...}

    public List<SubcategoryDTO> getSubcategoriesByCategory(Long categoryId) {...}

    public List<CategoryDTO> getTrendingCategories(Integer size) {...}

    // CRUD OPERATIONS

    public Set<Category> findAll() {...}

    public Optional<Category> findById(Long id) { return categoryRepo.findById(id); }

    public Category save(Category category) { return categoryRepo.save(category); }

    public void deleteById(Long id) { categoryRepo.deleteById(id); }

    public void delete(Category category) { categoryRepo.deleteById(category.getCategoryId()); }

    public void deleteAll() { categoryRepo.deleteAll(); }
}
```

Рисунок 2.20 – Приклад сервісу

2.3.8 Контролер

Останньою згаданою частиною архітектури йдуть контролери – класи, анотовані як @Controller, або ж в нашому випадку @RestController – розширеною версією звичайного контролера, і які відповідають за обмін інформацією з зовнішніми клієнтами через мережу за допомогою REST API та HTTP-запитів.

З обов'язкового в контролері – лише згадана анотація, а окрім неї вже можна впроваджувати безпосередньо потрібний код. Перше, що є абсолютно в усіх контролерах застосунку – залежність від відповідного сервісу для подальшої передачі даних із запитів. Окрім цього, оскільки кожен контролер відповідатиме за свою окрему нішу в функціоналі застосунку, а саме за свою частину API, яку надає наш сервер для клієнтів, кожен з них анотований своїм власним URL шляхом за допомогою анотацій, наданих Spring Boot, за

яким до його методів можна буде доступитися. Кожен з таких методів, також має окремий шлях, який додається до базового, таким чином формуючи унікальний шлях для кожного з них. У прикладі з контролером категорій (рисунок 2.21), реалізовано чотири методи, кожен з яких посилає дані по ланцюгу до відповідного методу бізнес-логіки уже в сервісі, таким чином створюючи для кожного шматку функціоналу чітко визначений шлях, уникаючи масових колізій серед викликів методів, що також полегшує роботу для відлагодження, тестування та масштабування коду.

```
@RestController
@RequestMapping("/api/categories")
public class CategoryController {
    private final CategoryService categoryService;

    public CategoryController(CategoryService categoryService) { this.categoryService = categoryService; }

    @GetMapping("/{categoryId}/picture")
    public ResponseEntity<byte[]> getCategoryPicture(@PathVariable("categoryId") Long categoryId) {...}

    @GetMapping("")
    public ResponseEntity<List<CategoryDTO>> getAllCategories() {...}

    @GetMapping("/{categoryId}/subcategories")
    public ResponseEntity<List<SubcategoryDTO>> getSubcategoriesByCategory(@PathVariable("categoryId") Long categoryId) {...}

    @GetMapping("/trending")
    public ResponseEntity<List<CategoryDTO>> getTrendingCategories(@RequestParam(defaultValue = "4") Integer size) {...}
}
```

Рисунок 2.21 – Приклад контролеру

2.3.9 Data Transfer Objects

Додатковими компонентами в архітектурі застосунку, хоч і не обов’язковими, є класи типу Data Transfer Object (DTO), які, як і моделі, відповідають за представлення даних, проте йдеться саме про дані, якими застосунок обмінюється з клієнтами [13, 14]. Приклад такого класу для сутності категорій продемонстровано на рисунку 2.22.

```
public record CategoryDTO(
        Long categoryId,
        String name,
        String description
) {
}
```

Рисунок 2.22 – Приклад DTO

Першою функцією таких класів є ефективний обмін даних між застосунком і клієнтом, уникаючи передачі представлення зв’язності даних для зовнішніх програм, і передаючи лише ті дані, які потрібні. Таку практику втілено і в реалізований застосунок, адже контролери і приймають, і повертають саме типи даних DTO (див. рисунок 2.21). Це і є другою основною функцією таких класів – передається лише та частина інформації,

яка зазначена всередині них [13, 14]. До прикладу, якщо серверу потрібно повернути дані користувача, замість того, щоб повертати об’єкт з його паролем усередині, хоч і зашифрованим, створено додатковий клас UserDTO, як проілюстровано на рисунку 2.23, де для передачі використано лише необхідну частину його даних.

```
public record UserDTO(
        Long userId,
        String name,
        String surname,
        String fathername,
        String email,
        String phone,
        Date birthdate
) {
}
```

Рисунок 2.23 – Приклад використання частини даних

2.3.10 Аутентифікація та авторизація

Згадавши про контролери, за результатами написання коду серверна частина в результаті надає для клієнтів більше двадцяти шляхів, до яких можна звернутися, розділені по восьми контролерам. Оскільки програма має працювати не тільки з загально доступними даними, включаючи персональні дані користувачів, які реєструються у системі, впроваджено систему безпеки, яка в свою чергу надає механізми аутентифікації та подальшої авторизації користувачів, розмежовуючи всі API, які надає сервер, на ті, які доступні для всіх, і ті, які недоступні, допоки користувач не є авторизованим [15].

Основою системи на серверній частині є інструменти, надані Spring Boot Security, а саме механізми, які взаємодіють з конкретним класом, який відповідає сутності, яка має бути в процесі аутентифікована або ні, в нашому випадку це клас User, який відповідає сутності користувача. Щоб дати можливість Spring Security механізмам працювати з цим класом, він імплементує інтерфейс UserDetails і всі необхідні методи (рисунок 2.24). Додатково до цього необхідно створити Enum з списком ролей, які ми будемо надавати юзеру та додати поле до нашого класу. У нашому випадку в користувача є заздалегідь прописане поле role.

```
public class User implements UserDetails {
```

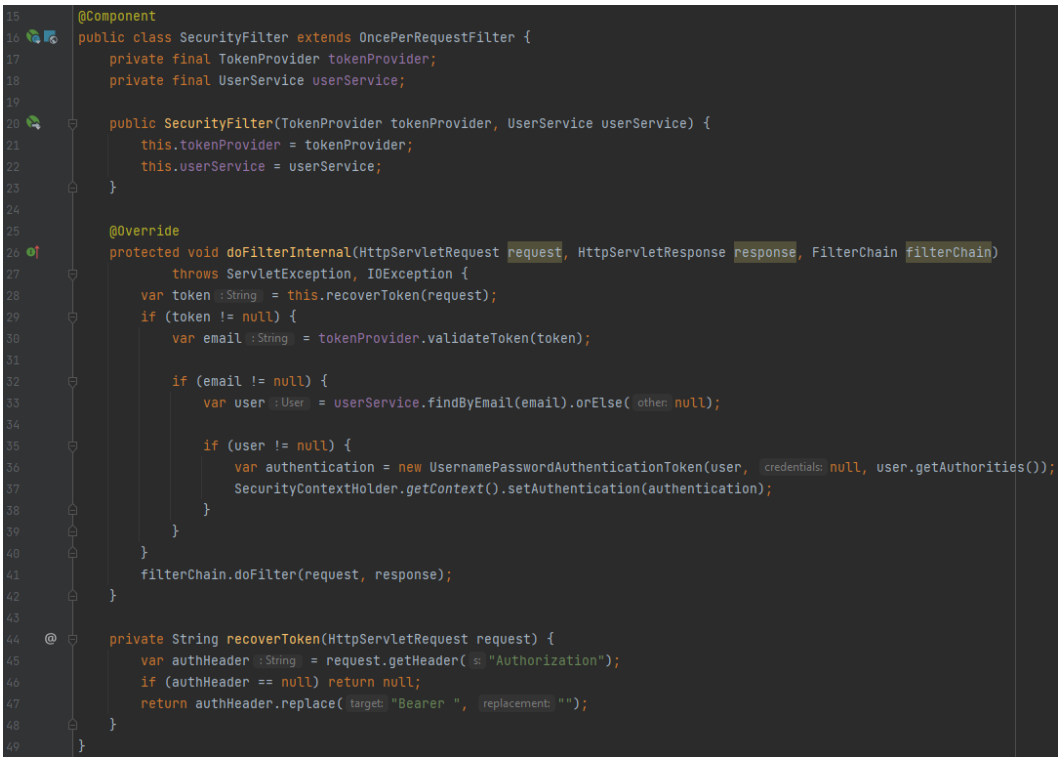
Рисунок 2.24 – Визначення класу для механізмів аутентифікації

Другим кроком створено клас-сервіс, який не залежить від репозиторіїв, і надає функціонал, за допомогою якого потім буде генеруватися Jwt токен для користувача, який авторизується, і який в свою чергу використовує для генерації механізми Auth0 Jwt (рисунок 2.25) [16].

```
@Service
public class TokenProvider {
    @Value("jwtsecret")
    private String JWT_SECRET;

    public String generateAccessToken(User user) {
        try {
            Algorithm algorithm = Algorithm.HMAC256(JWT_SECRET);
            return JWT.create()
                    .withSubject(user.getUsername())
                    .withClaim( name: "username", user.getUsername())
                    .withExpiresAt(generateAccessExpirationDate())
                    .sign(algorithm);
        } catch (JWTCreationException e) {
            throw new JWTCreationException("Error generating token", e);
        }
    }

    public String validateToken(String token) {
        try {
            Algorithm algorithm = Algorithm.HMAC256(JWT_SECRET);
            return JWT.require(algorithm) Verification
                    .build() JWTVerifier
                    .verify(token) DecodedJWT
                    .getSubject();
        } catch (JWTVerificationException e) {
            return null;
        }
    }

    private Instant generateAccessExpirationDate() {
        return LocalDateTime.now().plusHours(1).toInstant(ZoneOffset.of( offsetId: "+03:00"));
    }
}
```

Рисунок 2.25 – Сервіс для генерації Jwt токенів

Останній клас, реалізований для роботи механізмів Spring Security для наших користувачів, є клас, який відповідає за додатковий фільтр безпеки, який обробляє абсолютно кожен запит перед надходженням його на відповідний контролер (рисунок 2.26). Його задача – перевірити, чи є у запиті Jwt токер клієнта, чи коректний цей токен, і на основі отриманих даних визначити, чи аутентифікований цей клієнт і запит, чи ні, і на основі цього визначити, чи надавати доступ до контролеру.



```
@Component
public class SecurityFilter extends OncePerRequestFilter {
    private final TokenProvider tokenProvider;
    private final UserService userService;

    public SecurityFilter(TokenProvider tokenProvider, UserService userService) {
        this.tokenProvider = tokenProvider;
        this.userService = userService;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {
        var token = this.recoverToken(request);
        if (token != null) {
            var email = tokenProvider.validateToken(token);

            if (email != null) {
                var user = userService.findByEmail(email).orElse( other: null);

                if (user != null) {
                    var authentication = new UsernamePasswordAuthenticationToken(user, credentials: null, user.getAuthorities());
                    SecurityContextHolder.getContext().setAuthentication(authentication);
                }
            }
        }
        filterChain.doFilter(request, response);
    }

    private String recoverToken(HttpServletRequest request) {
        var authHeader = request.getHeader( s: "Authorization");
        if (authHeader == null) return null;
        return authHeader.replace( target: "Bearer ", replacement: "");
    }
}
```

Рисунок 2.26 – Клас для фільтру запитів

На даному етапі в застосунку є працюючий механізм аутентифікації та авторизації, проте окрім цього необхідно налаштувати, яким ролям надавати який рівень доступу до API. Це реалізовано в класі-конфігурації, анотованим @Configuration, а також @EnableWebSecurity. Усередині основного методу знаходиться конфігурація всіх шляхів, до яких надходять запити, і окрім налаштувань доступів, ставиться вище згаданий фільтр перед кожним запитом (рисунок 2.27).

```
@Configuration
@EnableWebSecurity
public class AuthConfig {
    private final SecurityFilter securityFilter;

    public AuthConfig(SecurityFilter securityFilter) { this.securityFilter = securityFilter; }

    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws Exception {
        return httpSecurity
                .csrf(AbstractHttpConfigurer::disable)
                .authorizeHttpRequests(authorize -> authorize
                        .requestMatchers( ...patterns: "/api/auth/**").permitAll()
                        .requestMatchers(HttpMethod.GET, ...patterns: "/api/categories/**").permitAll()
                        .requestMatchers( ...patterns: "/api/categories/**").hasRole("ADMIN")
                        .requestMatchers(HttpMethod.GET, ...patterns: "/api/colors/**").permitAll()
                        .requestMatchers( ...patterns: "/api/colors/**").hasRole("ADMIN")
                        .requestMatchers(HttpMethod.GET, ...patterns: "/api/materials/**").permitAll()
                        .requestMatchers( ...patterns: "/api/materials/**").hasRole("ADMIN")
                        .requestMatchers( ...patterns: "/api/orders/create").hasAnyRole( ...roles: "USER", "ADMIN")
                        .requestMatchers(HttpMethod.GET, ...patterns: "/api/products/**").permitAll()
                        .requestMatchers( ...patterns: "/api/products/filtered-count", "/api/products/edge-prices", "/api/products/filtered").permitAll()
                        .requestMatchers( ...patterns: "/api/products/**").hasRole("ADMIN")
                        .requestMatchers( ...patterns: "/api/subcategories/**").hasRole("ADMIN")
                        .requestMatchers( ...patterns: "/api/users/{userId}",
                                "/api/users/{userId}/profile-edit",
                                "/api/users/{userId}/password-change",
                                "/api/users/{userId}/orders").hasAnyRole( ...roles: "USER", "ADMIN")
                        .requestMatchers( ...patterns: "/api/users/**").hasRole("ADMIN")
                        .anyRequest().permitAll())
                .addFilterBefore(securityFilter, UsernamePasswordAuthenticationFilter.class)
                .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                .build();
    }
```

Рисунок 2.27 – Конфігурація доступу для користувачів

Для безпосередньо надання клієнтам можливості авторизації та реєстрації створено один додатковий контролер і сервіс, що відповідають за ці дві опції. За допомогою раніше налаштованих інструментів дані частини програми можуть використовувати контекст програми для перевірки імені та паролю, та відповідно вирішувати, чи надавати йому Jwt токен для подальшого його використання для отримання доступу до того чи іншого місця на сервері (рисунок 2.28).

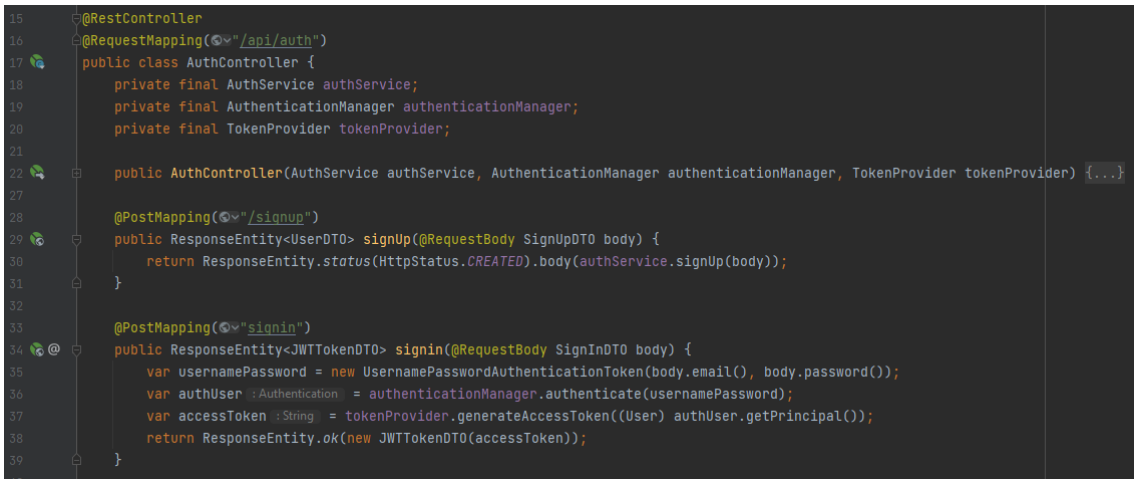
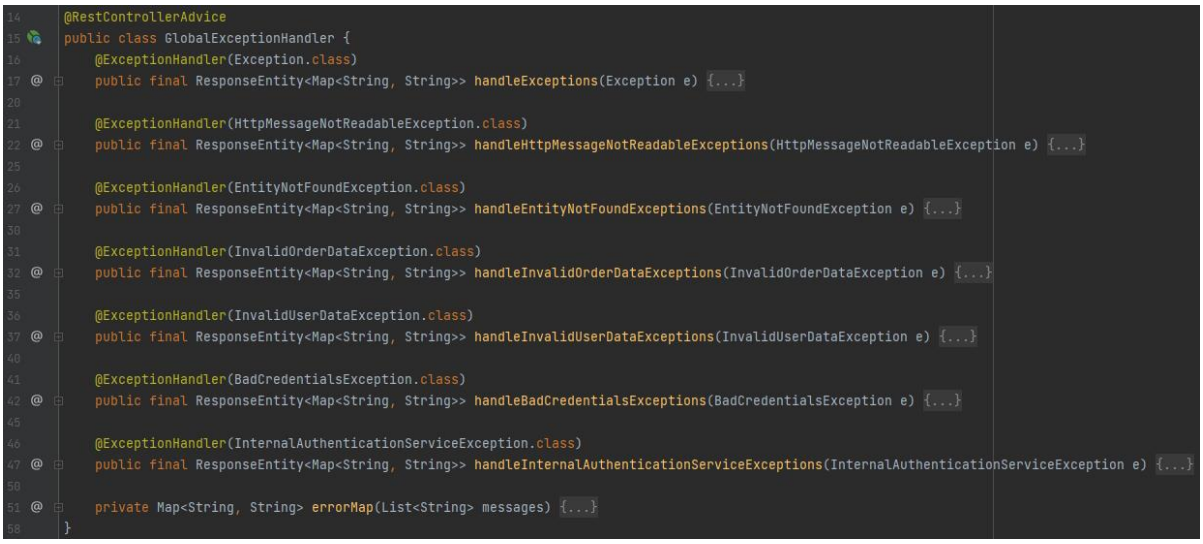


Рисунок 2.28 – Контролер для авторизації

2.3.11 Оброблення помилок

У будь-якому випадку не буде такого, що завжди вся логіка спрацює як заплановано, через те, що запити до серверу можуть нести неправильну, чи неповну інформацію, тому існують сценарії, які обробляються в коді і в окремих випадках викидаються помилки, які перекидуються на контролери із сервісів.

У таких випадках такі помилки можуть бути критичними і зупинити сервер, а отже було реалізовано окрему частину програми для оброблення будь-яких помилок, які можуть викликатись. Для цього створено клас, анотований @RestControllerAdvice (рисунок 2.29). Ця анотація, надана SpringBoot, дозволяє в одному місці програми оброблювати будь-яку помилку, викликану в будь-якому місці в контролерах. А оскільки архітектура застосунку така, що контролер ініціює будь-які операції, то всі помилки так чи інакше вийдуть туди, що значить, що абсолютна більшість помилок буде оброблена і ситуацій з зупинкою сервера буде уникнуто [17].



```
@RestControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(Exception.class)
    public final ResponseEntity<Map<String, String>> handleExceptions(Exception e) {...}

    @ExceptionHandler(HttpMessageNotReadableException.class)
    public final ResponseEntity<Map<String, String>> handleHttpMessageNotReadableExceptions(HttpMessageNotReadableException e) {...}

    @ExceptionHandler(EntityNotFoundException.class)
    public final ResponseEntity<Map<String, String>> handleEntityNotFoundExceptions(EntityNotFoundException e) {...}

    @ExceptionHandler(InvalidOrderDataException.class)
    public final ResponseEntity<Map<String, String>> handleInvalidOrderDataExceptions(InvalidOrderDataException e) {...}

    @ExceptionHandler(InvalidUserDataException.class)
    public final ResponseEntity<Map<String, String>> handleInvalidUserDataExceptions(InvalidUserDataException e) {...}

    @ExceptionHandler(BadCredentialsException.class)
    public final ResponseEntity<Map<String, String>> handleBadCredentialsExceptions(BadCredentialsException e) {...}

    @ExceptionHandler(InternalAuthenticationServiceException.class)
    public final ResponseEntity<Map<String, String>> handleInternalAuthenticationServiceExceptions(InternalAuthenticationServiceException e) {...}

    private Map<String, String> errorMap(List<String> messages) {...}
}
```

Рисунок 2.29 – Обробник помилок

2.4 Front-end

Для розробки клієнтської частини було обрано JavaScript бібліотеку React – найпопулярніший і один з найпотужніших інструментів для написання веб-інтерфейсу, що і було головною ціллю. На відміну він звичайного написання розмітки та коду, React надає можливість створювати інтерфейси складного характеру з використанням компонентів, окремо і незалежно написаних, таким чином сприяючи повторному використанню коду, і також розділення його на різні частини застосунку, уникаючи явища нагромадження великої кількості коду в одному місці, що в свою чергу дозволяє в майбутньому легше тестувати та масштабувати програму [18].

Крім того, що це потужний і зручний інструмент для реалізації поставлених задач, React має відкритий код та широку й активну спільноту розробників, що дозволяє йому розвиватися з плином часу без витрачення великої кількості ресурсів, що в свою чергу каже про те, що в перспективі застосунки, написані на React матимуть довготривалу підтримку, що є окремою перевагою такого вибору.

2.4.1 Архітектура

У клієнтській частині на React, окрім визначення елементів для написання компонентів, які будуть відображені на сторінці, було додатково вирішено використати архітектуру Page-Element. Ідея такої архітектури полягає в тому, що кожен елемент, який безпосередньо несе основну частину інтерфейсу і реалізованої логіки, обгортається додатковим компонентом, який зветься Page. Це зроблено для того, щоб забезпечити додаткову організацію коду та підвищення його читабельності та зручності написання у випадку розширення проєкту.

2.4.2 React + Redux

Для вирішення окремих задач у написанні логіки клієнтської частини, до прикладу реалізація функціональної корзини, куди користувач зберігатиме товари, які бажав би в майбутньому оформити як замовлення, потрібні механізми зберігання стану відповідних елементів. У цьому проєкті було вирішено використати Redux у зв’язці з React для управління глобальним станом застосунку. Redux – бібліотека для управління станом, доступним з будь-якої частини коду або будь-якого елементу та шляху, що забезпечує керування такими даними в масштабних додатках [19].

На прикладі, проілюстрованому на рисунку 2.30, продемонстровано створення відповідного функціонального об’єкту для управління станом корзини користувача за допомогою createSlice функції, наданої інструментами Redux. За допомогою таких механізмів, надалі з’являється можливість зручно керувати даними з будь-якої частини застосунка в потрібні моменти.

```
import { createSlice } from "@reduxjs/toolkit";

const cartSlice = createSlice( options: {
    name: 'cart',
    initialState: {
        items: []
    },
    reducers: {
        setCartItems(state, action) {
            state.items = action.payload;
        },
        addToCart(state, action) {
            state.items.push(action.payload);
            localStorage.setItem('cartItems', JSON.stringify(state.items));
        },
        removeFromCart(state, action) {
            state.items = state.items.filter(item => item.productId !== action.payload);
            localStorage.setItem('cartItems', JSON.stringify(state.items));
        },
        loadCartItems(state) {
            const savedCartItems = localStorage.getItem( key: 'cartItems');
            if (savedCartItems) {
                state.items = JSON.parse(savedCartItems);
            }
        }
    }
});
```

Рисунок 2.30 – приклад використання Redux

2.4.3 Маршрутизація

У Front-end частину застосунку було використано сучасний підхід до конфігурації маршрутизації, а саме з використанням компонентів під назвами MainLayout та Outlet. Цей механізм дозволяє створювати однаковий макет для абсолютно всіх сторінок сайту, при цьому змінюючи лише наповнення певної частини. У нашому випадку це виявляється корисним, адже такі дві частини як навігаційний блок та “Footer” сайту матиме відображатися на всіх сторінках. Тому в даному випадку MainLayout компонент складається з трьох частин – компоненти Navbar, Footer і безпосередньо Outlet, наповнення якого вже буде динамічно змінюватися в залежності від шляху (рисунок 2.31). Сама зміна наповнення компоненту Outlet відбувається за допомогою конфігурації шляхів і відповідних їм елементів за допомогою компонентів Route і RouteProvider, наданих React (рисунок 2.32) [20].

```
const MainLayout = () =>{
    return (
        <>
            <Navbar />
            <Outlet />
            <Footer />
        </>
    )
}
```

Рисунок 2.31 – Компонент MainLayout

```
const router = createBrowserRouter(
    createRoutesFromElements(
        <Route path='/' element={<MainLayout />}>
            <Route index element={<HomePage popularProducts={popularProducts}/>}/>
            <Route path='/products/:productId/' element={<ProductPage/>}/>
            <Route path='/:categoryName/:subcategoryName' element={<SelectedCategoryPage />}/>
            <Route path='/:categoryName' element={<SelectedCategoryPage/>}/>
            <Route path='/search?s=:search' element={<SelectedCategoryPage/>}/>
            <Route path='/cart' element={<CartPage popularProducts={popularProducts}/>}/>
            <Route path='/order' element={<OrderPage/>}/>
            <Route path='/profile' element={<ProfilePage/>}/>
        </Route>
    )
);
return <RouterProvider router={router}/>;
```

Рисунок 2.32 – Конфігурація маршрутизації

2.4.4 Компонент

У готовій клієнтській частині було розроблено досить велика кількість компонентів для відображення абсолютно різних сценаріїв для користувача залежно від шляхів та його дій (див. рисунок 2.32), отже розглянемо компонент на прикладі найбільш функціональної сторінки – відображення товарів з можливостями фільтрації, пошуку та переходу на ці товари.

Мова про компонент під назвою SelectedCategory, який відповідає одразу за три різних сценарії – відображення товарів конкретної категорії, товарів конкретної підкатегорії, а також товарів за введеним пошуковим запитом.

Першим кроком створено сам компонент у вигляді функції та розмітки JSX, наданої React. На рисунку 2.33 проілюстровано частину розмітки, яку написано з використанням згаданого синтаксису. Окрім самої розмітки, яка відповідає за розміщення елементів на сторінці, додатково до кожного елементу прописані стилі для застосування до елементів, частина яких для компоненту SelectedCategory проілюстровано на рисунку 2.34.

```
return (
    <div className='selected-category-container'>
        <div className="category-path">
            {subcategory ?
                <h1>Home &#8594; <span>{category.name} &#8594; {subcategory.name}</span></h1>
                :
                (category ? <h1>Home &#8594; <span>{category.name}</span></h1> : <h1>Home &#8594; <span>{search}</span></h1>)
            }
        </div>

        <div className="category-top-menu">
            {subcategory ?
                <h2 className='category-title'>{subcategory.name}</h2>
                :
                <h2 className='category-title'>{category && category.name}</h2>
            }

            <div className="category-overall-filter">
                <span>Sort By: </span>
                <select name="" id="price-sort" className='category-select' onChange={togglePriceSort}>
                    <option value="Lowest Price">Lowest Price</option>
                    <option value="Highest Price">Highest Price</option>
                </select>
            </div>
        </div>
```

Рисунок 2.33 – Приклад розмітки компоненту

```
.selected-category-container{
    margin: 0 80px;
}
.category-path{
    margin-top: 20px;
}
.category-top-menu{
    display: flex;
    justify-content: space-between;
    margin-top: 35px;
}

.category-title{
    font-size: 30px;
    font-weight: 700;
    color: #03132c;
}
.category-select{
    width: 300px;
    padding: 8px 2px;
    font-size: 14px;
    font-weight: 400;
}
```

Рисунок 2.34 – Приклад стилів для компоненту

Ці два етапи дозволили створити структуру елементів для відображення даних, проте основна частина полягає в тому, щоб ці дані стягнути з сервера та коректно розташувати у створених місцях. Для цього необхідно звернутися до серверної частини з запитом (рисунок 2.35).

```
const getEdgePrices = async () => {
    const body = {...};
    const response= await fetch( input: 'http://localhost:8080/api/products/edge-prices', {method: 'POST'...});
    const data = await response.json();
    return data;
}

const getMaterials = async () =>{
    const response = await fetch( input: 'http://localhost:8080/api/materials');
    const data = await response.json();
    setMaterials(data);
}

const getColors = async () =>{
    const response = await fetch( input: 'http://localhost:8080/api/colors');
    const data = await response.json();
    setColors(data);
}
```

Рисунок 2.35 – Приклад запитів до серверу

Оскільки на сервері реалізовані механізми безпеки, які залежать на аутентифікації та авторизації користувачів, до кожного запиту з клієнтської сторони необхідно додавати в тіло Jwt токен, який повертається від самого серверу в момент, коли користувач коректно авторизується на сайті через відповідну форму та сторінку.

Результат відображення даного компоненту разом з усіма даними, стягнутими з сервера, можна побачити на рисунку 2.36.

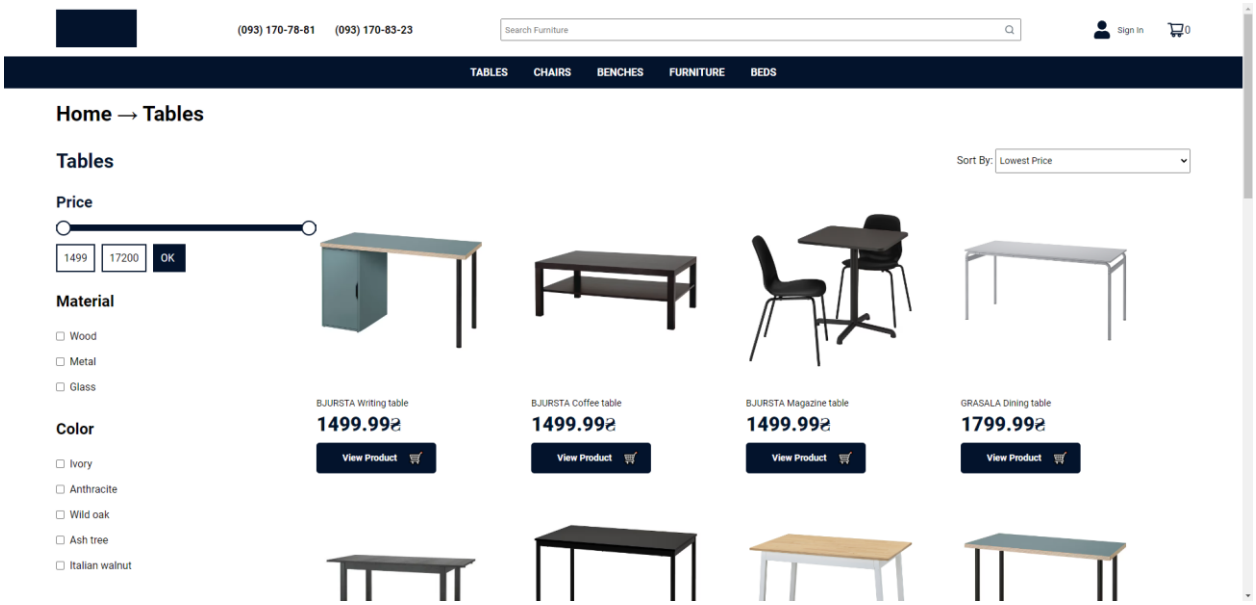


Рисунок 2.36 – Зовнішній вигляд компоненту SelectedCategory

Висновки

У цій курсовій роботі було досліджено процес розробки повноцінного веб-застосунку для онлайн-магазину, який включає в себе як клієнтську частину, так і частину серверу з відокремленою базою даних. Основною задачею було створити зручний та привабливий для користувача додаток з використанням корисних практик написання коду та з надійними механізмами аутентифікації та авторизації користувачів для убезпечення даних і персональної інформації.

Серверна частина була розроблена на фреймворку Spring Boot, з використанням низки наданих інструментів і архітектури MVC. Даний фреймворк було обрано через свою гнучкість, надійність та написання коду зі зручною подальшою можливістю масштабування проєкту. Основну увагу у відповідному розділі було присвячено коректній конфігурації, опису інструментів та вибору й імплементації архітектури, що дало результат у вигляді стабільного й ефективно працюючого серверу для подальшого використання.

Для клієнтської частини застосунку було обрано використати JavaScript бібліотеку Reactб яка дозволила зручно й ефективно написати динамічний інтерфейс для відображення сторінок користувачу. Використання архітектури Page-Element розшарувало розташування коду в застосунку, додатково розвантаживши низку місць у програмі від скупчення логіки, а інструменти, надані Redux, у поєднанні з React дали можливість управляти певною частиною логіки програми кратно зручніше й компактніше в питанні кількості коду.

Результатом виконаної роботи став веб-сервіс, який повністю відповідає поставленій меті й цілям, а також із якісно написаним кодом з можливістю масштабування застосунку без проблем, пов’язаних з великою кількості коду чи логіки, зосереджених у одному місці. Реалізований проєкт демонструє ефективність та надійність використання Spring Boot разом з вбудованими можливостями та інструментами для реалізації якісного та швидкого серверу з механізмами захисту від несанкціонованого доступу, а також React для зручного створення якісної клієнтської частини веб-застосунків. Тим самим результатом роботи є не лише створення готового продукту, а також поглиблення та закріплення моїх знань з використання патернів проєктування та розробки клієнт-серверних застосунків.

Список використаних джерел

1. tkrotoff. Front-end frameworks popularity (React, Vue, Angular and Svelte) [Електронний ресурс] / tkrotoff. – 2019. – Режим доступу до ресурсу: https://gist.github.com/tkrotoff/b1caa4c3a185629299ec234d2314e190.
2. Most Popular Backend Frameworks – 2012/2023 [Електронний ресурс] // Statistics & Data. – 2023. – Режим доступу до ресурсу: https://statisticsanddata.org/data/most-popular-backend-frameworks-2012-2023/.
3. Bahmani A. Automatic database normalization and primary key generation [Електронний ресурс] / A. Bahmani, M. Naghibzadeh // Canadian Conference on Electrical and Computer Engineering // ResearchGate. – 2008. – Режим доступу до ресурсу: https://www.researchgate.net/publication/4349094_Automatic_database_normalization_and_primary_key_generation.
4. Spring Boot Servlet Web Applications [Електронний ресурс] – Режим доступу до ресурсу: https://docs.spring.io/spring-boot/docs/current/reference/html/web.html.
5. Mancas C. On the Paramount Importance of Database Constraints [Електронний ресурс] / Christian Mancas // ResearchGate. – 2015. – Режим доступу до ресурсу: https://www.researchgate.net/publication/292672966_On_the_Paramount_Importance_of_Database_Constraints.
6. Advanced Security Mechanisms in the Spring Framework: JWT, OAuth, LDAP and Keycloak [Електронний ресурс] / N.Dimitrijević, N. Zdravković, A. Mesterovic, M. Bogdanović // The 14th International Conference on Business Information Security // ResearchGate. – 2024. – Режим доступу до ресурсу: https://www.researchgate.net/publication/379891547_Advanced_Security_Mechanisms_in_the_Spring_Framework_JWT_OAuth_LDAP_and_Keycloak.
7. Choma D. THE EFFICIENCY AND RELIABILITY OF BACKEND TECHNOLOGIES: EXPRESS, DJANGO, AND SPRING BOOT [Електронний ресурс] / D. Choma, K. Chwaleba // ResearchGate. – 2023. – Режим доступу до ресурсу: https://www.researchgate.net/publication/376709512_THE_EFFICIENCY_AND_RELIABILITY_OF_BACKEND_TECHNOLOGIES_EXPRESS_DJANGO_AND_SPRING_BOOT.
8. Hall S. How do architectures actually translate to code? [Електронний ресурс] / Sam Hall // Medium. – 2022. – Режим доступу до ресурсу: https://betterprogramming.pub/backend-software-architecture-demystified-c545e0b1460b.
9. Majeed A. MVC Architecture: A Detailed Insight to the Modern Web Applications Development [Електронний ресурс] / Abdul Majeed // Crimson Publishers. – 2018. – Режим доступу до ресурсу: https://crimsonpublishers.com/prsp/pdf/PRSP.000505.pdf.

10. Collings T. Controller-Service-Repository [Електронний ресурс] / Tom Collings // Medium. – 2021. – Режим доступу до ресурсу: https://tom-collings.medium.com/controller-service-repository-16e29a4684e5.
11. Madasu V. SOLID Principles in Software Architecture and Introduction to RESM Concept in OOP [Електронний ресурс] / V. Madasu, T. Venna, T. Eltaeib // Journal of Engineering Science and Technology // ResearchGate. – 2015. – Режим доступу до ресурсу: https://www.researchgate.net/publication/273451390_SOLID_Principles_in_Software_Architecture_and_Introduction_to_RESM_Concept_in_OOP.
12. C. Hubli S. Efficient Backend Development with Spring Boot: A Comprehensive Overview [Електронний ресурс] / S. C. Hubli, D. Jaiswal // IJRASET. – 2023. – Режим доступу до ресурсу: https://www.ijraset.com/research-paper/efficient-backend-development-with-spring-boot#conclusion.
13. Abdalrhmanalkraien. Data Transfer Object pattern and Mapper [Електронний ресурс] / Abdalrhmanalkraien // Medium. – 2021. – Режим доступу до ресурсу: https://medium.com/@abdalrhmanalkraien/data-transfer-object-pattern-and-mapper-116508bc9df0.
14. Yılmaz O. The DTO Pattern (Data Transfer Objects) [Електронний ресурс] / Orçun Yılmaz // Medium. – 2023. – Режим доступу до ресурсу: https://medium.com/@orcunyilmazoy/the-dto-pattern-data-transfer-objects-8146b262636e.
15. Tumin S. A Closer Look at Authentication and Authorization Mechanisms for Web-based Applications [Електронний ресурс] / S. Tumin, S. Encheva // ResearchGate. – 2012. – Режим доступу до ресурсу: https://www.researchgate.net/publication/250310860_A_Closer_Look_at_Authentication_and_Authorization_Mechanisms_for_Web-based_Applications.
16. Shingala K. JSON Web Token (JWT) based client authentication in Message Queuing Telemetry Transport (MQTT) [Електронний ресурс] / Krishna Shingala // ResearchGate. – 2019. – Режим доступу до ресурсу: https://www.researchgate.net/publication/331587509_JSON_Web_Token_JWT_based_client_authentication_in_Message_Queuing_Telemetry_Transport_MQTT.
17. DEMİRŞEN A. Spring Boot Global Exception Handler [Електронний ресурс] / Ahmet Emre DEMİRŞEN // Medium. – 2023. – Режим доступу до ресурсу: https://medium.com/@aedemirsen/spring-boot-global-exception-handler-842d7143cf2a.
18. Chen S. Front-End Development in React: An Overview [Електронний ресурс] / S. Chen, U. Thaduri, V. Koteswara Rao Ballamudi // ResearchGate. – 2019. – Режим доступу до ресурсу: https://www.researchgate.net/publication/374154236_Front-End_Development_in_React_An_Overview.

19. Shravan G. Comprehensive Analysis of React-Redux Development Framework [Електронний ресурс] / G. Shravan, A. Sandeep // IJCRT. – 2020. – Режим доступу до ресурсу: https://www.ijcrt.org/papers/IJCRT2004607.pdf.
20. aysuncodes. Mastering React Routing: A Comprehensive Guide [Електронний ресурс] / aysuncodes // Medium. – 2023. – Режим доступу до ресурсу: https://medium.com/@aysunitai/mastering-react-routing-a-comprehensive-guide-86d5ad0d5df.