Warsaw University of Technology

FACULTY OF
MATHEMATICS AND INFORMATION SCIENCE

Bachelor's diploma thesis

in the field of study Computer Science and Information Systems

Distributed Recommendation System for Enhanced User Experience in Client Applications

Wojciech Basiński

student record book number 323475

Szymon Kupisz

student record book number 320603

Jakub Oganowski

student record book number 318334

thesis supervisor

dr inż. Grzegorz Ostrek

WARSAW 2025

Abstract

Distributed Recommendation System for Enhanced User Experience in Client Applications

In the XXIst century, our civilization has been witnessing the drastic rise of the e-commerce industry. One of the crucial reasons why this was possible is the rapid development of software development, including the programming technologies available on the market. The modern e-commerce systems should therefore take advantage of those technologies in order to create fast-working and user-friendly application.

Moreover, in the preceding years, the world has seen the tremendous evolution of the Artificial Intelligence industry that has dominated many different sectors of the IT industry. Hence, one should consider the AI methods to provide better user experience to the customers.

The primary goal of this thesis is to develop a system that reflects the functionality of the Amazon e-commerce platform, enhanced with a comprehensive recommendation engine. This system aims to deliver personalized product suggestions to users while maintaining low response latency and high reliability. To achieve this, a range of software development methods and techniques will be applied.

Keywords: e-commerce, artificial intelligence, distributed systems, microservices, microfrontends, recommendation systems

Streszczenie

Rozproszony system rekomendacji wspomagający wybory użytkownika w aplikacjach klienckich

W XXI wieku nasza cywilizacja jest świadkiem gwałtownego rozwoju branży e-commerce. Jednym z kluczowych powodów, dla których było to możliwe, jest szybki rozwój technologii programistycznych dostępnych na rynku. Nowoczesne systemy e-commerce powinny zatem wykorzystywać te technologie, aby tworzyć szybkie i przyjazne dla użytkownika aplikacje.

Co więcej, w ostatnich latach świat doświadczył ogromnej ewolucji branży sztucznej inteligencji, która zdominowała wiele różnych sektorów przemysłu IT. W związku z tym warto rozważyć zastosowanie metod AI w celu zapewnienia użytkownikom lepszych wrażeń z korzystania z aplikacji.

Głównym celem tej pracy jest opracowanie systemu odzwierciedlającego funkcjonalność platformy e-commerce Amazon, wzbogaconego o kompleksowy silnik rekomendacji. System ten ma na celu dostarczanie spersonalizowanych rekomendacji produktów użytkownikom przy jednoczesnym zachowaniu niskiego czasu odpowiedzi i wysokiej niezawodności. W tym celu zostanie zastosowany szereg metod i technik programistycznych.

Słowa kluczowe: e-commerce, sztuczna inteligencja, systemy rozproszone, mikroserwisy, mikrofrontendy, systemy rekomendacji

Declaration / Oświadczenie

We hereby declare that:
Niniejszym oświadczamy, że:

1. We [**have not**] used IT tools to generate the content of the manuscript of this thesis.
   [**Nie**] używaliśmy narzędzi informatycznych do generowania treści niniejszej pracy
2. We [**have**] used IT tools to generate the code of the software developed for this thesis.
   Używaliśmy narzędzi informatycznych do generowania kodu programów stworzonych na potrzeby niniejszej pracy.
3. We take full responsibility for all content in the thesis, including both the manuscript and the software developed for it. / Bierzemy pełną odpowiedzialność za zawartość niniejszej pracy, która składa się zarówno z jej tekstu, jak i stworzonego na jej potrzeby oprogramowania.

Category	**English description**	**Polish description**
The scope of the use of IT tools to generate software code/zakres użycia narzędzi informatycznych do generowania kodu	ChatGPT was used to give advices upon simple programming tasks like mocking the components for the testing purpose etc.	ChatGPT został użyty do udzielania rad w przypadku prostych problemów programistycznych, na przykad mockowania komponentów celem testowania itp.

Continued on next page

The scope of the use of IT tools to generate manuscript / (zakres użycia narzędzi informatycznych do generowania tekstu pracy	AI tools were used to support the redacting process	Narzędzia AI zostały użyte do pomocy w redakcji tekstu niniejszej pracy

Continued on next page

Contents

1. Introduction

1.1. Introduction of the application

E-commerce platforms like Amazon have revolutionized shopping by allowing personalized user experience which was mainly enabled by the use of recommendation systems. This thesis focuses on developing a distributed web application resembing an ecommerce platform, enhanced by content-based and user-based collaborative filtering recommendation systems. Our goal is to replicate the best practices of such pltforms and to explore how a distributed architecture can address certain limitations of building fault tolerant solutions in eccommerce and online commerce.

Traditional monolith ecommerce architectures often have issues with being fault tolerant and maintainable enough. Our system addresses these problems by using distributed methods. The backend is built using **microservices**, allowing for independent development, deployment and potential scaling of individual components, such as product management, user auth and products interactions. This modularity also increases fault tolerance because a failure in one microservice does not have impact on the entire system - only on the broken one. Similarly, the frontend utilizes a **microfrontend** architecture which in the anlogical way improves development and the deployment of each frontend modules like product browsing or shopping cart management. We successfully implemented both the microservice and microfrontend architecture strategies.

Existing platforms offer recommendations, but often rely on single approaches. Our system enhances personalization by incorporating two distinct recommendation engines: **content-based filtering**, which suggests similar products based on product attributes, and **user-based collaborative filtering**, which recommends products based on the preferences of similar users. This dual approach provides more diverse and targeted suggestions, aiming to improve user engagement and potentially increase sales. Both recommendation engines were successfully implemented.

Moreover, our system leverages async communication with **message brokers** ensuring

eventual consistency between the modules. It is also focused on **handling large scale data** by using modern **SQL** and **NoSQL** and **vector** database technologies.

This thesis provides a guide to the design and implementation of our distributed ecommerce application. It details the design and technical implementation and shows insights into modern ecommerce technologies, distributed and recommentation systems.

1.2. Goal of the project

1.2.1. Description of the goal

The goal of this project is to develop a distributed ecommerce web app including content-based and user-based collaborative filtering recommendation systems.

The app aims to use the microservices architecture and real-world data [9], to enhance online shopping experienceof the user, improve user engagement on the ecommerce platform, and ensure high performance and system reliability.

The solution attempts to resemble modern e-commerce platforms by implementing its core functionalities with advanced AI driven recommendations, providing a reliable and user friendly web application.

1.2.2. Functional requirements

The functional requirements have been gathered in form of user stories related to different features. They are displayed below in **Table 1.1**

Table 1.1: Functional Tests

Feature	Action	Reason
User Authentication	I want to register an account.	So I can create a new account and access the application.
	I want to log in.	So I can access my account and use the system's features.
	I want to log out.	So I can end my session and prevent unauthorized access.

Continued on next page

Table 1.1 (continued)

Feature	Action	Reason
	I want to change password	So I can change my access credentials.
	I want to delete my account	So I can permanently remove my access and data.
User Profile Management	I want to update my personal data.	So I can ensure that my information is accurate and up to date.
	I want to manage my addresses.	So I can easily manage my delivery/contact addresses.
	I want to retrieve my profile information.	So I can review and confirm my information at any time.
Product Management	I want to search for products using filters.	So I can find products that match my preferences efficiently.
	I want to retrieve product details by ID.	So I can make better decisions when buying a specific product.
	I want to retrieve my own product list.	So I can manage and monitor the products I have listed for sale.
	I want to create and list a new product.	So I can make it available for buyers on the platform.
	I want to add images to a product.	So I can show the product to buyers.
	I want to delete a product.	So I can remove listings that are no longer valid.
Likes and Reviews	I want to like a product.	So I can show that I enjoy it/like it.
	I want to view products I liked.	So I can go back to my favorite items.

Continued on next page

Table 1.1 (continued)

Feature	Action	Reason
	I want to see the number of likes of a product.	So I can see which products are popular and which not.
	I want to remove my like from a product.	So I can change my preference.
	I want to see the average rating for a product.	So I can assess its overall quality and other users' reviews.
	I want to add or update a product review.	So I can share feedback or my previous review.
	I want to delete my product review.	So I can remove my entries.
Basket Management	I want to add a product to my shopping cart.	So I can organize the items I want to buy.
	I want to remove a product from my shopping cart.	So I can adjust my order if I no longer want to buy the item.
	I want to retrieve a list of products in my shopping cart.	So I can review and confirm the products I want to buy.
Admin Management	I want to delete a user account by UUID.	So I can remove accounts that violate platform policies or are no longer active.
	I want to retrieve user account details by ID.	So I can verify and manage the user information.
	I want to retrieve a paginated list of users with filters.	So I can manage the users.
	I want to change a user's role.	So I can adjust user permissions and roles as needed.
Order Management	I want to create an order.	So I can buy products or services on the platform.

Continued on next page

Table 1.1 (continued)

Feature	Action	Reason
	I want to retrieve details of a specific order.	So I can check the status and details of the purchase.
	I want to retrieve a list of my orders.	So I can view and track products that I bought.

1.2.3. Non-functional requirements

Functionality

- The system should allow users to perform CRUD operations on products, reviews, likes, accounts, basket and orders while maintaining consistency across microservices.
- The system should ensure data encryption for sensitive information such as passwords.
- The recommendation system should provide products suggestions for a user based on the preferences of other users and similar products to the ones that they have interacted with.

Usability

- Each microfrontend should maintain a consistent layout and navigation using a shared design system such as Microsoft Fluent UI, ensuring a uniform user experience.
- The user interface should be available in many languages.
- App should display error messages and informations to make the navigation simpler.

Reliability

- The system should display informative error messages for failed actions such as invalid form submissions, payment errors etc.
- Each microservice should be fault-tolerant and the their messages system should be asynchronous and providing data consistency.

- The system should be designed to ensure that stored data is valid by validating it on the input

Performance

- Page load times on the frontend should typically remain below 2 seconds for users with standard internet connections.
- Each REST API call should have a response time under 300ms for 95% of requests under normal load.

Supportability

- Unit tests should cover core functionalities to avoid bugs.
- Tests should use an in built in memory database to simulate real life scenarios.
- The backend system should be easily containerized with Docker.
- The system should include documentation of the endpoints.

1.2.4. Distribution of work accross the team members

The team members were responsible for conducting the following work regarding the application:

Table 1.2: Work distribution

Name and surname	Work description
Wojciech Basiński	primarly responsible for creating the AI models for the recommendation systems
Szymon Kupisz	primarly responsible for the frontend layer of the application
Jakub Oganowski	primarly responsible for creating the backend layer of the application

The tasks conducted by the entire team were creating the application scope, including functional and non-functional requirements, and writing this thesis.

1.2.5. Boundaries of the project

In this project we develop an e-commerce application focused on: user registration, login, logout, product search, product recommendations, reviews, likes, shopping basket with checkout. The goal is to provide rather a proof of concept of the core functionalities of the online ecommerce system than an enterprise solution.

2. Main part

2.1. Problem analysis

Ecommerce platforms are large and complex systems due to the number of customers and traffic they serve on the daily basis. The initial job of this thesis is to identify them to properly adress them in our solution. Key issues include:

- **Data Management:** Platforms process structured data (e.g. user profiles) and unstructured data (e.g. product descriptions) which makes the data processing challenging.
- **Personalization:** Users expect personalized recommendations as they became industry standard - since that our platform must analyse the data and provide them.
- **Communication:** Distributed systems are based on smooth communication between the services - we must ensure it will be consistent over time.
- **User Experience:** A visually appealing user interface is necessary to compete with other platforms. It is also good to make it reliable and different parts of this user interface should be working independent from each other.
- **Fault Tolerance:** Failures can cause disruptions and break the user experience - since that the system must handle them gently and not crash fully when issue arises.

The following sections of this thesis will outline the architectural and theoretical foundations needed to address these challenges and problems.

2.2. Architecture

2.2.1. Application architecture decisions & description

Overview of architecture

The application architecture is designed to use a microfrontend approach on the frontend and a microservices architecture on the backend. Advantages from this structure are that the application is divided into services, which enables easier development, testing and provides independence of each service. Additionally, microservices architecture provides an opportunity to use different technology for each service, which was used in this application. The backend is deployed on a GCP(Google Cloud Platform) on a virtual machine and each service and database is containerized using Docker, which provides a repeatable and simple process of deployment of services and consistency of environment.

Data flow

The data flow in the system is standard for web app applications based on microservices:

1. Frontend communicates with backend through REST APIs exposed by the API Gateway using HTTP requests.
2. Gateway authenticates and authorizes requests and then routes them to the proper service.
3. Request is received on one of the backend services. To proceed, the request backend may need to communicate with other services to fetch required data.
4. Communication between microservices is asynchronous using the Apache Kafka message broker.
5. Each service has its own database from which it retrieves already stored data.
6. After processing, the response is delivered back to the gateway, which then passes it to the frontend.

This structure ensures clear division of functionality and ease of tracking and debugging flows. Diagram visualizing data flow (2.1)

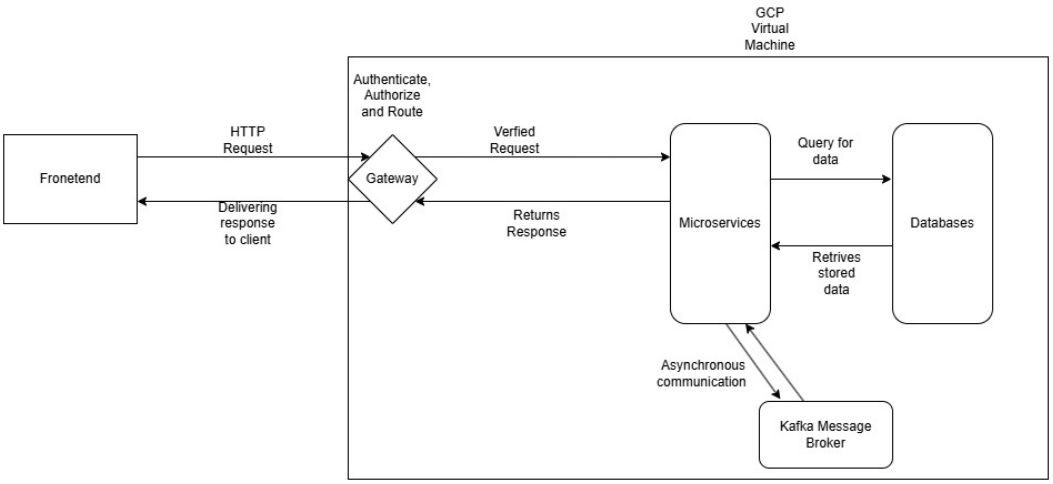


Figure 2.1: Data Flow Diagram

2.2.2. Frontend technology decisions & description

Overview of frontend

The main objective of the frontend layer of the application is to provide to the user the GUI interface, being as accessible as possible and easy to use.

Technology choice

Concerning the goals set for the frontend layer, it was decided to choose the web application as the technology with which the implementation of the frontend layer of the application will occur. Web browsers are one of the most accessible programs in the world, therefore the users will be able to access the application in many different situations.

Language For the programming language used to build the frontend layer of the application, it was decided to use the Typescript language. The choice of this technology was motivated by the fact that it is built on top of the Javascript language, a commonly used high-level scripting language for building the web applications. Moreover, Typescript, in comparison to Javascript, is optionally static typing programming language, which allows the software engineer to manage the variables' types more efficiently. Furthermore, Typescript is compiled to Javascript with the use of various code-optimization techniques, which makes it possible to run the code faster than in case of pure Javascript[1].

Build tool & environment As the build tool, it was decided that the Vite.js build tool will be used due to its abilities to compile Typescript rapidly and due to the fact that it has recently became the standard build tool for the web application programming industry.

Core Libraries In order to create the frontend application's code faster, more efficiently and in more secure way, it was decided to use various different libraries available in the NPM registry, which is the commonly used package manager and installer for the Javascript dependencies, including the following ones:

Table 2.1: Main frontend dependencies

Name	Description
React.JS	a Javascipt library created and maintained by Meta for creating the UI interfaces swiftly. Its code is stable, it has a vast community of users, is simple to use, SEO friendly, declarative and the components built with it are composable i.e. we can use one component inside another. Those were the reasons why it was decided to choose it as the library for creating the frontend application
Microsoft Fluent UI	a UI components library provided by Microsoft. It was selected due to the large number of components provided and to the fact that it is used by the Microsoft company, hence it can be considered reliable.
React Router DOM	a package which is essential for implementing the dynamic routing of the application
Axios	a library for handling HTTP/HTTPS connections in a convenient way
Styled components	a library for styling the components with the use of tagged template literals
React i18next	a library for React application for supporting the i18next framework, which is a framework for supporting the internationalization of Javascript applications

Continued on next page

Table 2.1 (continued)

Name	Description
originjs/vite-plugin-federation	a vite plugin for supporting the Module Federation i.e. the Webpack plugin that will make it possible to implement the architecture of microfrontends

Testing To test the application, Jest, a JavaScript testing framework focused on simplicity, was chosen. This decision was motivated by several factors: it is widely known in the web programming community, well-maintained and up-to-date, some team members have commercial experience with Jest, and it allows mocking parts of the software that are not the focus of a particular test.

Architecture description

It was decided that it would the be the microfrontends architecture that would be chosen as the frontend layer architecture. Microfrontends are a way of designing and implementing the web application by building separate subapplications, known as microfrontends, that will be providing the components for the application itself, known as container. This architecture has the following advantages:

- The components provided by microfrontends are loaded depending on the current needs of the container app, therefore the overall loading time of the application is faster
- The development of the particular application's parts can be done in parallel by different teams
- Different Javascript libraries and frameworks can be used to build the same application i.e. one microfrontend can be build with React.JS, the second one with Angular, the third one with Vue.JS etc.

Below, a diagram showing how the components providing between the container application i.e. the application ingesting the microfrontend-provided components, and its microfrontends looks like is provided.

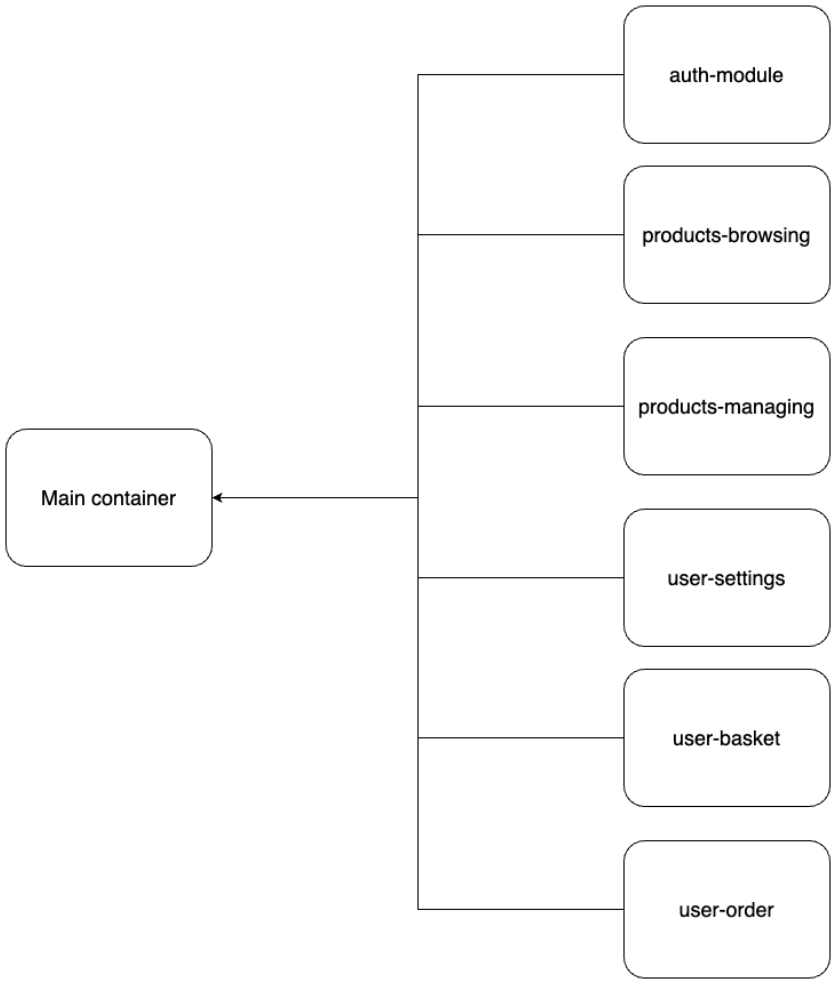


Figure 2.2: Microfrontends Architecture Diagram

However, this architecture has certain disadvantages. For example, if the dependencies management is done poorly, meaning the core libraries for the container application and the microfrontend ones are not updated simultaneously, it can lead to creating the technical debt.

2.2.3. Backend technology decisions & description

Overview of backend

The main objective of the backend layer of the application is to provide a well organized and effective infrastructure to process business logic of the application. Backend layer should manage data storage and provide secure and reliable communication between the frontend and other services. Each mircoservices is responsible for a specific domain of business functionality.

Technology Choice

Microservices Architecture: To achieve the goals that were stated for the backend layer, a microservices architecture has been chosen. In comparison to a monolithic architecture, microservices can be developed, tested and deployed independently. In contrast to a monolith application, every change requires work on the entire application. Due to this fact microservice is more cost-effective, simpler to develop and more reliable to failures. Moreover, microservices allow the use of different technologies for various modules. In terms of this application for example Python is used for recommendation services and Java for rest of services, whereas Monolithic architectures are limited to a single, shared technology stack for the entire system.

Diagram visualizing comparison between microservice and monolith architecture.(Figure 2.3)

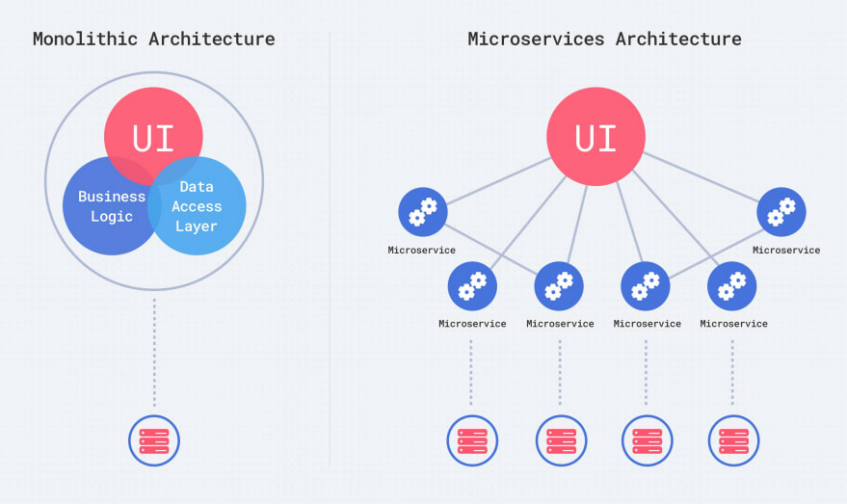


Figure 2.3: Comparison Monolith vs Microservices: [22]

Service Registration A service registry is a centralized system that keeps track of all the services in a distributed architecture, allowing them to dynamically register and discover each other. In this application, the Eureka service discovery system is used for this purpose. Developed by Netflix, Eureka allows microservices to register themselves and find other services without hardcoding endpoints. It provides load balancing, fault tolerance, and easier management of service-to-service communication.

Communication Between Services: In this application Apache Kafka, which is an open-source distributed event streaming platform. It takes responsibility for communi-

cation between microservices, which ensures reliable and asynchronous communication. For inter service communication Kafka is more suited option than HTTP, due its asynchronous message delivery system, while HTTP is synchronous. Asynchronous type of communication is more efficient and fault tolerance. It also reduce dependencies between services. The ability to queue and replay messages ensures no losing data during service downtime.

Frontend Communication via REST API: The REST API architecture, using the HTTP protocol, was selected for communication with the frontend layer. The decision was motivated by popularity and ease of implementation. HTTP protocol is compatible with almost all front-end technologies. It provides standard methods like GET, POST, PUT, PATCH, DELETE. As a result it integrates easily with different systems and is less resource-intensive compared to SOAP, which is more complex and requires a larger data payload. As well, in comparison to for example GraphQL, REST API is much easier to maintain and works better in applications with simpler data structure requirements.

Language As a programming language for backend, it was decided to use Java 21, which is the latest long-term support (LTS). This choice was supported by Java's respectable reputation, extensive documentation, and its rich ecosystem of tools and libraries. Java is widely used for building big enterprise applications, for example: Netflix, Linkedin and Amazon. Additionally, Java's static typing, compilation to byte code, multithreading and strong support for object oriented programming makes it well-suited to project requirements, unlike dynamically-typed languages like Python or JavaScript.

Along with Java, Spring Boot was chosen as the primary framework for building the backend services. It provides autoconfiguration and starter dependencies, which enable rapid development. Moreover, it provides a rich ecosystem of prebuilt libraries, such as Spring Web for building RESTful APIs, Spring Data for database interactions, and Spring Security for authentication and authorization.

Build tool & environment It was decided to use **Maven** as the build tool. Maven is a popular build automation and dependency management tool in the Java ecosystem. It provides a simple process of compiling, testing, and packaging the application. Maven also has a special mechanism for managing Java libraries and ensuring compatibility with the latest versions.

Core Libraries In order to create the backend applications code faster, more efficiently, and in a more secure way, it was decided to use various libraries and frameworks from the rich Java ecosystem, primarily managed through Maven. The following libraries were selected to build the backend application:

Table 2.2: Core Libraries

Library	Description
Spring Security	A framework within the Spring ecosystem that provides security mechanisms, including user authentication, authorization, and protection against common vulnerabilities like CSRF and XSS.
Spring Cloud	A framework within the Spring ecosystem that provides tools for building distributed systems, including service discovery, configuration management, circuit breakers, and API gateways.
Kafka Client	A library for integration with Apache Kafka.
Cloudinary SDK	A library for integrating with the Cloudinary platform, allowing easy in use uploading, deleting and retriving images on cloud .
Stripe-java	A library for integrating with the Stripe payment platform, allowing to mock payments .
Eureka(Netflix OSS)	A service discovery library within the Spring Cloud ecosystem. It allows microservices to dynamically register and discover each other.
Jackson	A library for serializing and deserializing JSON data, which enable omitting manual parsing.
Lombok	A Java library that reduces repetitive code by generating getters, setters, constructors, and other common methods at compile time using annotations, improving code readability and development efficiency.

Continued on next page

Table 2.2 (continued)

Library	Description
MapStruct	A Java library that simplifies object mapping by automatically generating code for mapping between DTOs and entities at compile time using annotations, improving development efficiency and ensuring consistency in data transformations.
OpenAPI (Swagger)	A library that simplifies the creation of API documentation. It automatically generates documentation for endpoints based on the code, ensuring consistency and reducing manual effort.
spring-boot-starter-data-mongodb	A dependency that integrates MongoDB, a NoSQL database, into Spring Boot applications, simplifying CRUD operations and query creation. It provides support for document-oriented data models.
JPA (Java Persistence API)	A Java specification for managing relational data with object-relational mapping (ORM), allowing developers to work with Java objects instead of raw SQL by providing annotations. Integrated with frameworks like Hibernate, JPA simplifies database interactions and supports features like lazy loading, caching, and query generation.

Testing The backend application was tested using **H2 Database** and **Mockito**:

H2 Database is an in-memory database, allows for quick integration and unit testing without relying on a fully configured database, minimizing setup time and ensuring tests remain lightweight. It also enables testing database queries, which are a critical part of the application.

Mockito, a mocking framework for Java, allows the creation of mock objects to isolate components during testing, simplifying dependency simulation and enabling a focus on specific business logic. Together, these tools make testing easier by mocking parts of proccess that are not intended to be tested.

2.2.4. AI technology decisions & description

Overview of AI

The project will implement both **content-based** and **user-based collaborative filtering** recommendation systems. They are both deployed as independent microservices exposed to an api gateway with their individual databases. In this way, we divide the risk of failure between the services. This architecture is depicted on the following **Figure 2.4**.

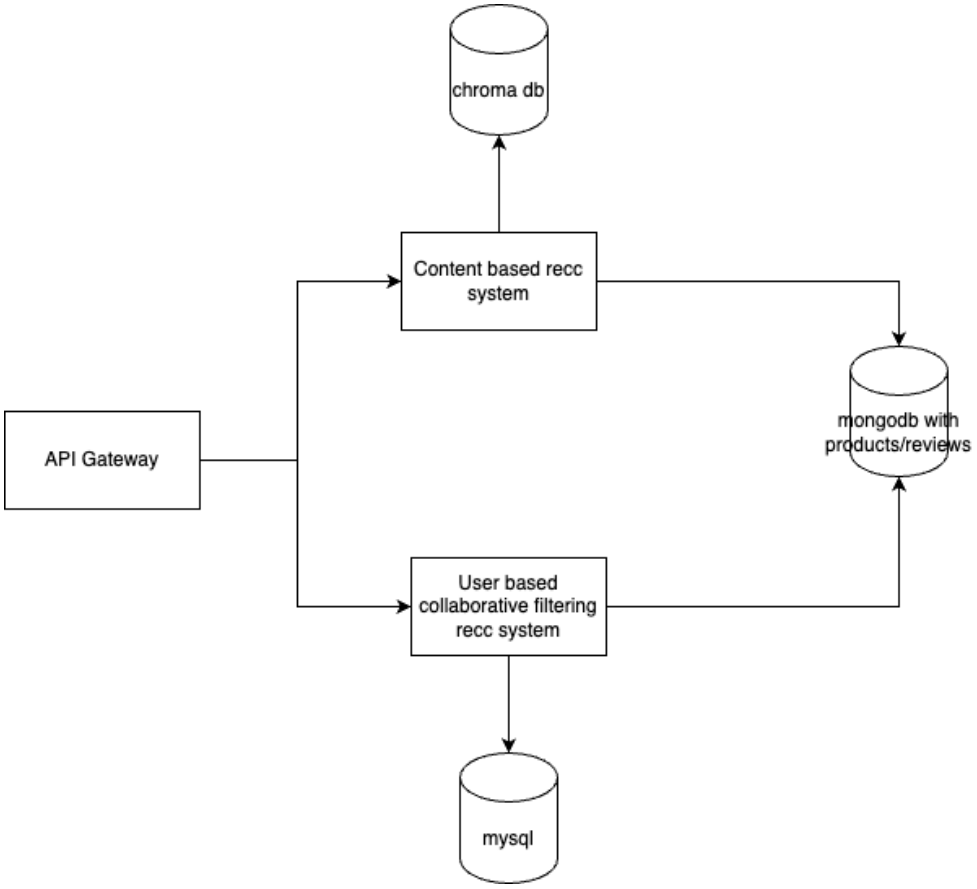


Figure 2.4: Recommendation System Architecture Diagram

Technology choice

Python was chosen as the main programming language because its considered an industry standard in data-related applications. There are many libraries and frameworks available to use for this purpose. On top of that, in order to serve content we'll utilise **Flask** library as it is a very lightweight server lib. In order to manage the products data, embeddings data and reccomendation matrices data, **MongoDB**, **ChromaDB**

and **mysql** python integrations were used respectively.

Language Python is an ideal choice for this project due to its wide adoption in the data science and machine learning communities. Its simplicity and readability accelerate development, while its extensive standard library and third-party ecosystem support a wide range of data-related tasks. Python's interoperability with various data formats and tools makes it particularly suited for building recommendation systems.

Data storage choice

The content-based recommendation system stores embeddings derived from product descriptions in **Chromadb**. The **user-based collaborative filtering** system stores user-item interaction matrices to provide recommendations in **mysql** db and last but not least products are stored in **mongodb**. All that is shown in **Figure 2.4**.

Build tool & environment The build environment leverages virtual environments to isolate project dependencies so it standardises the local work and the deployment process. We are using **pip** for package management, **venv** for virtual environments and **Docker** for components orchiestration. This setup is a great foundation for potential CI/CD flow in the future.

Core Libraries The libraries described in **Table 2.3** can be considered **the most important** for the recommendation system part.

Table 2.3: Libraries and Tools Overview

Category	Library/Tool	Description
Web Frameworks	Flask	A lightweight framework for web applications.
Data Processing and Analysis	pandas	Essential for data manipulation and analysis.
	numpy	Core library for numerical computations.
	scikit-learn	For machine learning tasks.

Continued on next page

Table 2.3 (continued)

Category	Library/Tool	Description
	scipy	For scientific computing and advanced mathematics.
Machine Learning and NLP	transformers	Hugging Face library for NLP models.
	sentence-transformers	For creating and using sentence embeddings.
Vector Databases and Embeddings	chromadb	For managing and querying embeddings efficiently.
Database and ORM	pymongo	For MongoDB interactions.
	SQLAlchemy	For relational database management.
Utility Libraries	python-dotenv	For managing environment variables.
	requests	For synchronous HTTP requests.

Testing This project uses Pytest which is a simple testing framework for Python. It supports unit, integration, and end-to-end tests. It's ease of setup and use was the main argument to introduce it to our project.

2.2.5. Recommendation system theoretical part

Overview of the recommendation systems

The project implements two independent recommendation systems: **content-based filtering** and **user-based collaborative filtering**. The following section with describe the theoretical part behind them

Content-based filtering

Content-based filtering analyzes the features of products from the database to recommend products that are similar to the one that the user is currently viewing. In this project, the method is implemented by generating embeddings from product details, storing these embeddings in a vector database and comparing them then looking for similarities. The process goes as follows:

1. **Feature Representation** Each product is represented using key attributes:

   - **ID**: A unique identifier.
   - **Title**: The product's name, preprocessed, cleaned up
   - **Description**: A textual description of the product, preprocessed, cleaned up.
   - **Main Category**: The primary classification of the product - cleaned for consistency
   - **Price**: A numeric field normalized to a predefined range

   These features are transformed into embeddings using a pretrained model: **all-MiniLM-L6-v2** [20]. The embeddings capture the semantic relationships between text-based features across different products.

2. **Weighted Concatenation of Embeddings** The **content based system** join the embeddings of different features into a single weighted concatenated vector. This is achieved using the following process:

   - Each feature f_i is assigned a weight w_i, that shows its importance in the comparison process. Example weights shown in **Table 2.4**
   - The embedding e_i for each feature f_i is multiplied by a corresponding weight w_i:

   $$v_i = w_i \cdot e_i \tag{2.1}$$

   - The weighted embeddings are concatenated to form a vector showing all necessary features together. It allows to emphasise the more important features during the search process.:

   $$V = [v_1, v_2, \ldots, v_n] \tag{2.2}$$

3. **Similarity Measures**

The system identifies similar products by comparing their embeddings using **cosine similarity** [12]. This metric evaluates the cosine of the angle between two vectors A and B shown in the **equation** (2.3):

$$\text{Cosine Similarity} = \frac{A \cdot B}{\|A\|\|B\|} \tag{2.3}$$

Where:

- $A \cdot B$ is the dot product of the vectors.
- $\|A\|$ and $\|B\|$ are the magnitudes of the vectors.

Cosine similarity ranges from -1 to 1, with 1 indicating identical vectors and 0 indicating no similarity. This allows us to catch the magnitude of the similarity aas well as its direction.

Table 2.4: Example Weights

Feature	**Symbol**	**Weight**
Title	w_1	0.4
Description	w_2	0.3
Main Category	w_3	0.2
Price	w_4	0.1

4. **Recommendation Generation**

The recommendation process involves:

(a) **Embedding Calculation**: Generating the embedding for the target product using the described preprocessing and embedding model.

(b) **Similarity Search**: Querying the vector database for the most similar embeddings using cosine similarity.

(c) **Result Retrieval**: Fetching the top n most similar products, ranked by their similarity scores.

This approach allows us to provide most similar products based on their features.

User-based Collaborative Filtering

User-based **collaborative filtering** [21] focuses on identifying users with similar preferences based on the reviews of products they've made.

1. **User-Item Interaction Matrix** The **user based collaborative filtering** is based on the **user-item interaction matrix** R:
   - $R \in \mathbb{R}^{|U| \times |I|}$, where:
     - $|U|$ is the number of users.
     - $|I|$ is the number of items.
     - $R_{u,i}$ represents the rating or interaction of user u with item i. If no interaction exists, $R_{u,i} = 0$.
2. **Similarity Computation** To identify similar users, a **cosine similary** is computed between the rows of R. This refers to the **equation** (2.3). Below we have it described how this particular reccomendation system: **equation** (2.4)

$$\text{sim}(u, v) = \frac{R_u \cdot R_v}{\|R_u\| \|R_v\|} \tag{2.4}$$

   where:
   - R_u: Row vector representing user u's ratings.
   - $\cdot$: Dot product.
   - $\|R_u\|$: Magnitude of R_u.
3. **Dimensionality Reduction with SVD** To handle sparse data and reduce noise, **Singular Value Decomposition (SVD)** [11] is applied:
   - Decomposition of R:

$$R \approx U \Sigma V^T \tag{2.5}$$

     where:
     - $U \in \mathbb{R}^{|U| \times k}$: User latent factors.
     - $\Sigma \in \mathbb{R}^{k \times k}$: Diagonal matrix of singular values.
     - $V \in \mathbb{R}^{|I| \times k}$: Item latent factors.
     - k: Number of latent dimensions.

- Reconstructed matrix:

$$\hat{R} = U\Sigma V^T \tag{2.6}$$

  where $\hat{R}$ represents the approximated user-item interaction matrix.

4. **Neighborhood Formation** Once user similarities are calculated, a **neighborhood** - a set of most similar entries - is formed by selecting the top N most similar users for the target user:

$$\mathcal{N}_u = \{v \mid \text{sim}(u, v) \text{ is among the top } N \text{ values}\} \tag{2.7}$$

5. **Prediction and Recommendation** The system predicts ratings for items that the target user u has not yet rated by considering the ratings from their neighborhood - the similar users:

   - Predicted rating for item i:

$$\hat{R}_{u,i} = \bar{R}_u + \frac{\sum_{v \in \mathcal{N}_u} \text{sim}(u, v) \cdot (R_{v,i} - \bar{R}_v)}{\sum_{v \in \mathcal{N}_u} |\text{sim}(u, v)|} \tag{2.8}$$

     where:

     – $\bar{R}_u$: Mean rating of user u.

     – $R_{v,i}$: Rating of item i by user v.

     – $\text{sim}(u, v)$: Similarity between users u and v.

   - Recommended items: The top n items with the highest predicted ratings $\hat{R}_{u,i}$ are selected as recommendations.

   This approach ensures personalized and relevant product recommendations based on user similarities and historical interactions.

2.3. Implementation

The implementation part of this thesis focuses on translating the theoretical foundations into a practical, working system. It details the design choices, technologies, and methods used to address the challenges outlined earlier. This section provides an in-depth look at the architecture, component interactions, and integration of key features such as data management, user experience, and fault tolerance. By bridging the gap between theory and practice, the implementation demonstrates how the proposed solutions are applied in a real-world context to create a functional and reliable e-commerce platform with reccomendastion system.

2.3.1. Architecture implementation

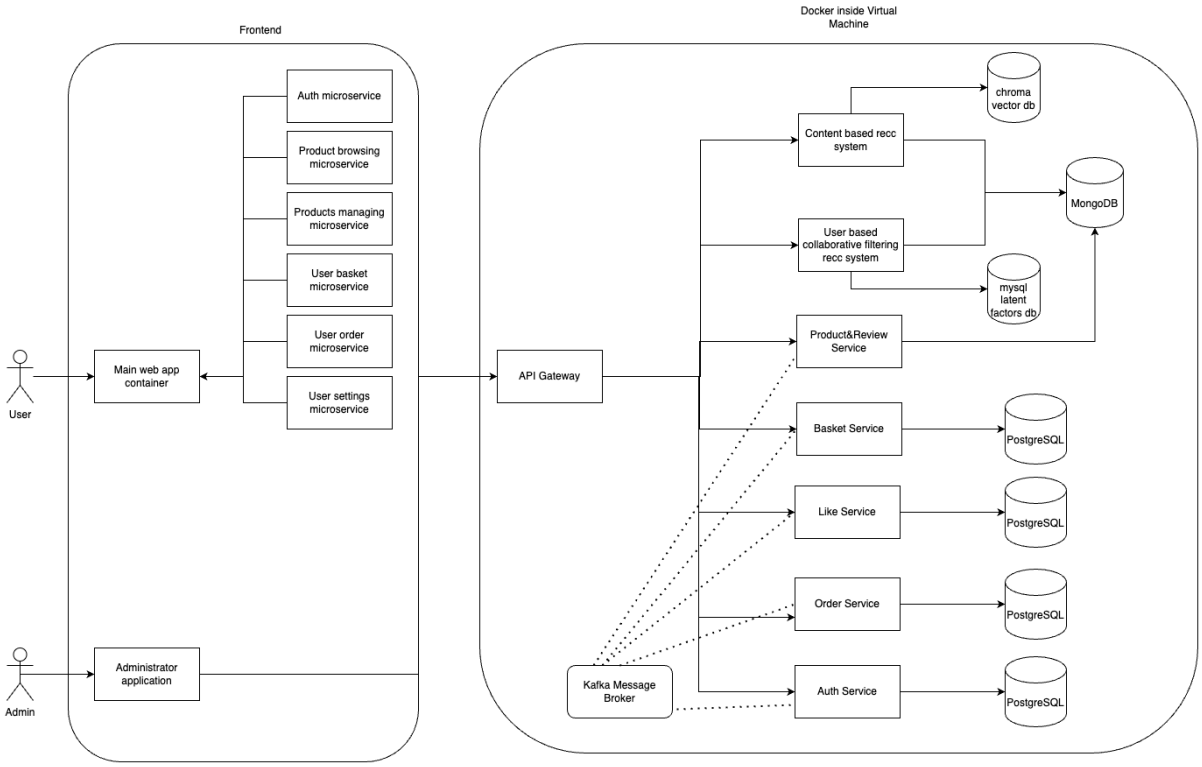


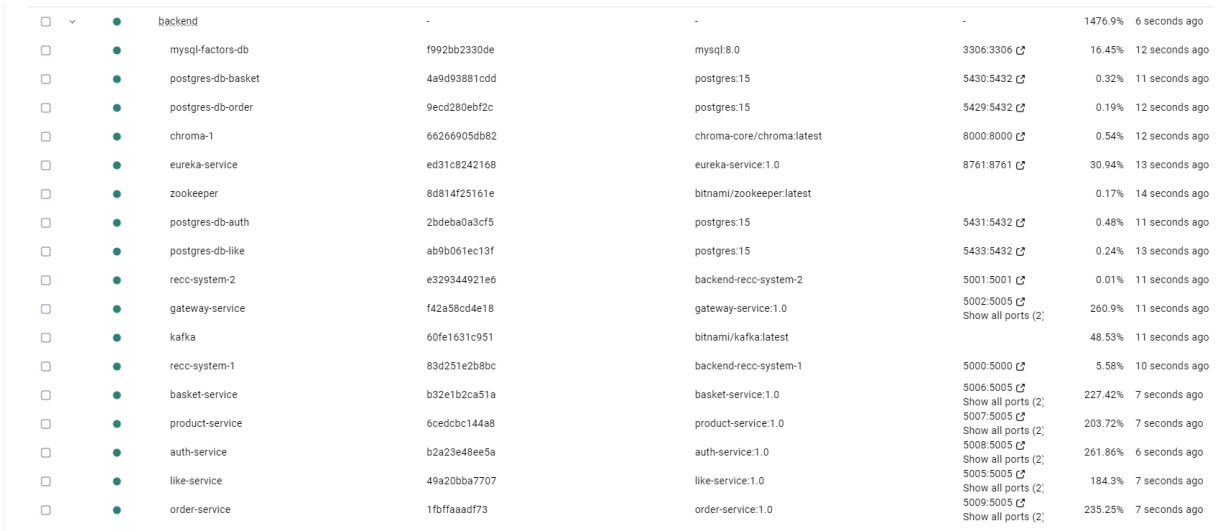
Figure 2.5: General Application Architecture Diagram

Diagram Description

Frontend Layer The frontend layer consists of a main web application container that aggregates multiple microfrontends. This main web application provides a unified entry URL prefix with common port.

Backend Layer As shown in the diagram (Figure 2.5), the backend part of the application is hosted on the Google Cloud Platform in a virtual machine. Each service and database are run in their own Docker container, defined in a Docker-Compose file. The file specifies the ports for each container and enables communication via a shared network. Containers for services are based on precompiled images (e.g., JAR files for Java), whereas databases are defined using database images (e.g., Postgres). Similarly, Kafka is based on an accessible image. To enable Kafka's proper configuration inside Docker, Zookeeper was also containerized, serving as a distributed coordination service to manage distributed systems.

Each container has defined dependencies to enable functionality and communication. For Java-based services, dependencies include Kafka for messaging and Eureka for service discovery and registration. Additionally, all functional services (both Java-based and recommendation systems) depend on their respective databases. Each container also has its own environment variables, which are read and utilized within the code for configuration and functionality. There is how this set up looks on docker desktop(Figure 2.6)



backend	-	-	-	1476.9%	6 seconds ago
mysql-factors-db	f992bb2330de	mysql:8.0	3306:3306	16.45%	12 seconds ago
postgres-db-basket	4a9d93881cdd	postgres:15	5430:5432	0.32%	11 seconds ago
postgres-db-order	9ecd280ebf2c	postgres:15	5429:5432	0.19%	12 seconds ago
chroma-1	66266905db82	chroma-core/chroma:latest	8000:8000	0.54%	12 seconds ago
eureka-service	ed31c8242168	eureka-service:1.0	8761:8761	30.94%	13 seconds ago
zookeeper	8d814f25161e	bitnami/zookeeper:latest		0.17%	14 seconds ago
postgres-db-auth	2bdeba0a3cf5	postgres:15	5431:5432	0.48%	11 seconds ago
postgres-db-like	ab9b061ec13f	postgres:15	5433:5432	0.24%	13 seconds ago
recc-system-2	e329344921e6	backend-recc-system-2	5001:5001	0.01%	11 seconds ago
gateway-service	f42a58cd4e18	gateway-service:1.0	5002:5005 Show all ports (2)	260.9%	11 seconds ago
kafka	60fe1631c951	bitnami/kafka:latest		48.53%	11 seconds ago
recc-system-1	83d251e2b8bc	backend-recc-system-1	5000:5000	5.58%	10 seconds ago
basket-service	b32e1b2ca51a	basket-service:1.0	5006:5005 Show all ports (2)	227.42%	7 seconds ago
product-service	6cedcbc144a8	product-service:1.0	5007:5005 Show all ports (2)	203.72%	7 seconds ago
auth-service	b2a23e48ee5a	auth-service:1.0	5008:5005 Show all ports (2)	261.86%	6 seconds ago
like-service	49a20bba7707	like-service:1.0	5005:5005 Show all ports (2)	184.3%	7 seconds ago
order-service	1fbffaaadf73	order-service:1.0	5009:5005 Show all ports (2)	235.25%	7 seconds ago

Figure 2.6: Docker Visualization

2.3.2. Frontend implementation

Overview

As described in the theoretical part, the frontend application architecture relies on the concept of microfrontends. To implement them, the @originjs/vite-plugin-federation vite package was used. This package implements the concept of Module Federation, which is

crucial for sharing the components between the projects.
The client frontend application consists of the following parts and their pruposes:

Component name	Purpose
Main	providing the user with the actual app and microfrontend components
auth-module	providing authentication panels and mechanics
products-browsing	handling all user activities related to products browsing and product display
products-managing	providing panels for adding, modifying and deleting products
user-settings	handling user account's settings
user-basket	handling the user basket UI
user-order	providing finalization of the user's order

Table 2.5: Application components and their purpose

Apart from the main part of the application, the additional Admin application, having no connection with the main part, was implement. Its purpose is to hand the system administrator in the ability to manage the user accounts, products etc.

Container application setup

The container application, named Main, is the gathering point of all of the microfrontends' content and some common parts of the application, including the main navbar, routing etc.
It was created by running the *npm create vite@latest* command, which made an empty project, and then with installing all of the necessary dependencies. In order to ingest the components from different microfrontends, the Main application uses @originjs/vite-plugin-federation vite plugin for Module Federation. The exact configuration is pasted in appendix 6

Below, a screenshot presenting how the main application search bar looks like.

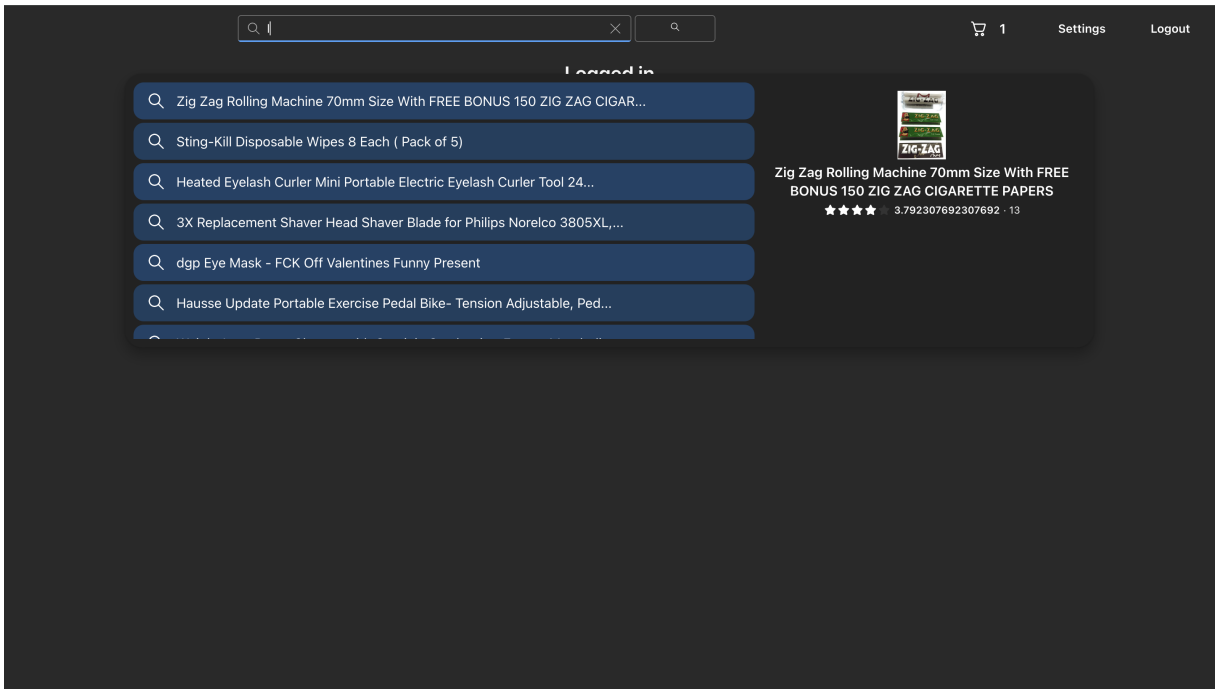


Figure 2.7: Search bar results

The container application uses the packages described in the appendix 5.4 together with the fuse.js package, which is used performing the fuzzy-search in the main search bar.

The backend connection relies on the HTTP methods described in the appendix 7. The Main application also takes care of the proper routing. The table with complete routing can be found in the 5.2 appendix.

For the implementation of the unit tests, the Jest testing framework was used for testing all of the .ts and .tsx files. The Main application uses the standard Environmental Variables Mocks (5.1) for the testing purposes. The test results look as follows:

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	99.26	92	96.55	100	
components/navbar	100	100	100	100	
Navbar.styled.tsx	100	100	100	100	
Navbar.tsx	100	100	100	100	
components/navbar/search	98.07	90	90.9	100	
Search.styled.tsx	100	100	100	100	
Search.tsx	97.77	90	90.9	100	92
components/navbar/search/lastItems	100	92.3	100	100	
LastItems.styled.tsx	100	100	100	100	
LastItems.tsx	100	92.3	100	100	71
components/preloader	100	100	100	100	
Preloader.tsx	100	100	100	100	
pages/page404	100	100	100	100	
Page404.styled.tsx	100	100	100	100	
Page404.tsx	100	100	100	100	

Test Suites: 5 passed, 5 total
Tests: 18 passed, 18 total
Snapshots: 0 total
Time: 8.598 s, estimated 17 s

Figure 2.8: Main Testing Results

Microfrontend setup and connection to the container application

Each of the microfrontends have been created with the *npm create vite@latest*, which initialized the empty React-Typescript app. Then, in the new empty app, being the particular microfrontend, all of the necessary libraries have been installed with the use of the *npm install* command and its variations.

To connect the microfrontends with the container application, the @originjs/vite-plugin-federation vite plugin for Module Federation was used. In order to implement it, the *vite.config.js* file is modified by adding the $federation$ function imported from @originjs/vite-plugin-federation, and by setting up the proper preview port, so the container app can handle the components ingestion. The skeleton of the configuration of the *vite.config.js* file is attached in the appendix 8.

The microfrontend preview port is a crucial thing in order to make sure the microfrontend can be ran independently of each other, and therefore to make the components management in the container application easier. The preview ports for different microfrontends can be found in the 5.5 appendix.

In terms of testing, each microfrontend uses the same approach i.e. the Jest testing framework to test all of the .ts and .tsx files.

Microfrontends implementations

auth-module The auth-module microfrontend provides the following components to the Container application:

Component	Description
SignIn	a sign in panel
SignUp	a sign up panel
AuthProvider	a component providing the authentication wrapper for the whole application

Table 2.6: Auth-module components

Below, a screen of the Sign In Panel is available to give an idea regarding how the authentication panels look like.

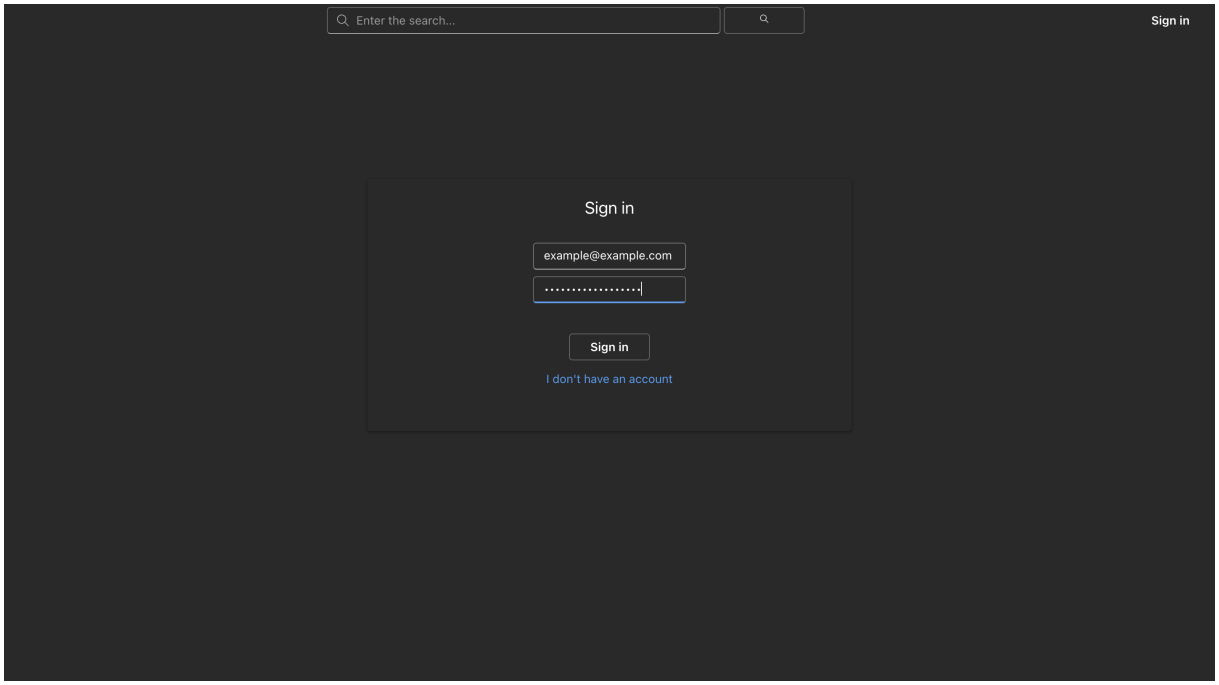


Figure 2.9: Sign in panel

The auth-module also uses the Event bus implemented for the communication with the container application. In certain places in its code, it emits the Javascript event named 'redirect', which purpose is to trigger the container's redirecting mechanics. As the container app uses the SignUp and SignIn panels as components, the event is received by the container app and is therefore processed.

The auth-module microfrontend was implemented with the use of the libraries listed in the appendix (5.4).
The backend connection is describe in the appendix 9
The auth-module microfrontend the standard Environmental Variables Mocks (5.1) for the testing purposes. The tests results are available at 5.1 appendix.

products-browsing The products-browsing microfrontend provides all of the parts needed to both browse through the available products and the display of any particular product.
Below, screenshots providing the parts of product displaying are provided.

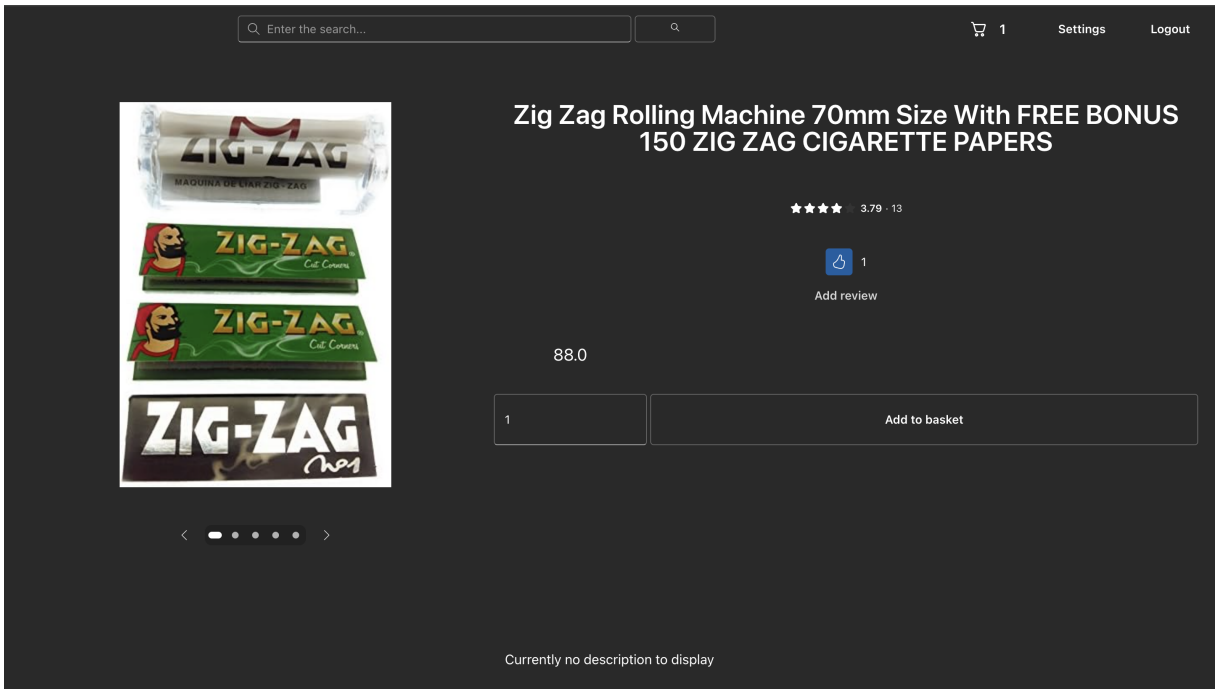


Figure 2.10: Product display

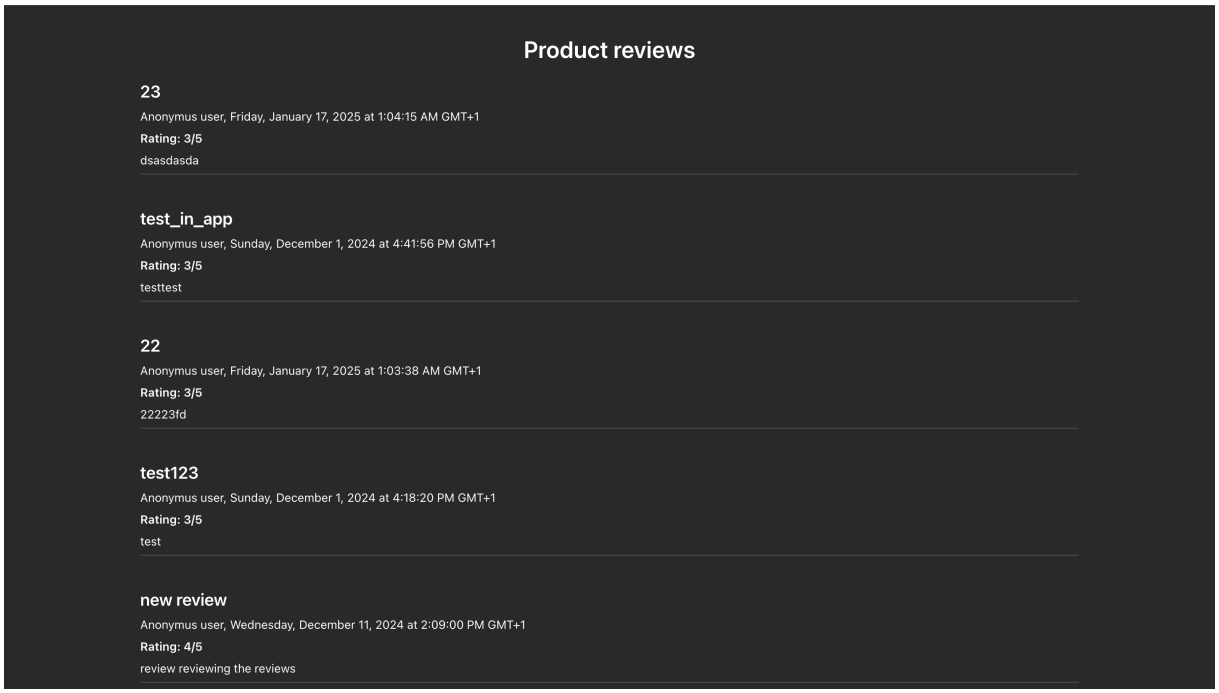


Figure 2.11: Product reviews

The products-browsing microfrontend can emit the 'reloadBasketNumber' Javascript event in case any product gets added to the basket. This event will trigger the container app to make a query to the backend API in order to reevaluate the number of items stored in the basket.

The products-browsing microfrontend provides the following components:

Component	Description
Product	a component displaying a selected product
Tiles	a panel providing the product browsing UI

Table 2.7: Products-browsing components

The products-browsing microfrontend was implemented with the use of the libraries listed in the appendix (5.4). The backend connection is describe in the appendix 11. The products-browsing microfrontend uses the dedicated Environmental Variables Mocks shown in the 5.3 appendix. The tests results are available in the 5.2 appendix.

products-managing The products-managing microfrontend provides the components which aim is to make the users capable of managing their products i.e. adding them,

deleting, etc.

Below, a screenshot presenting one of the parts of product adding process is shown.

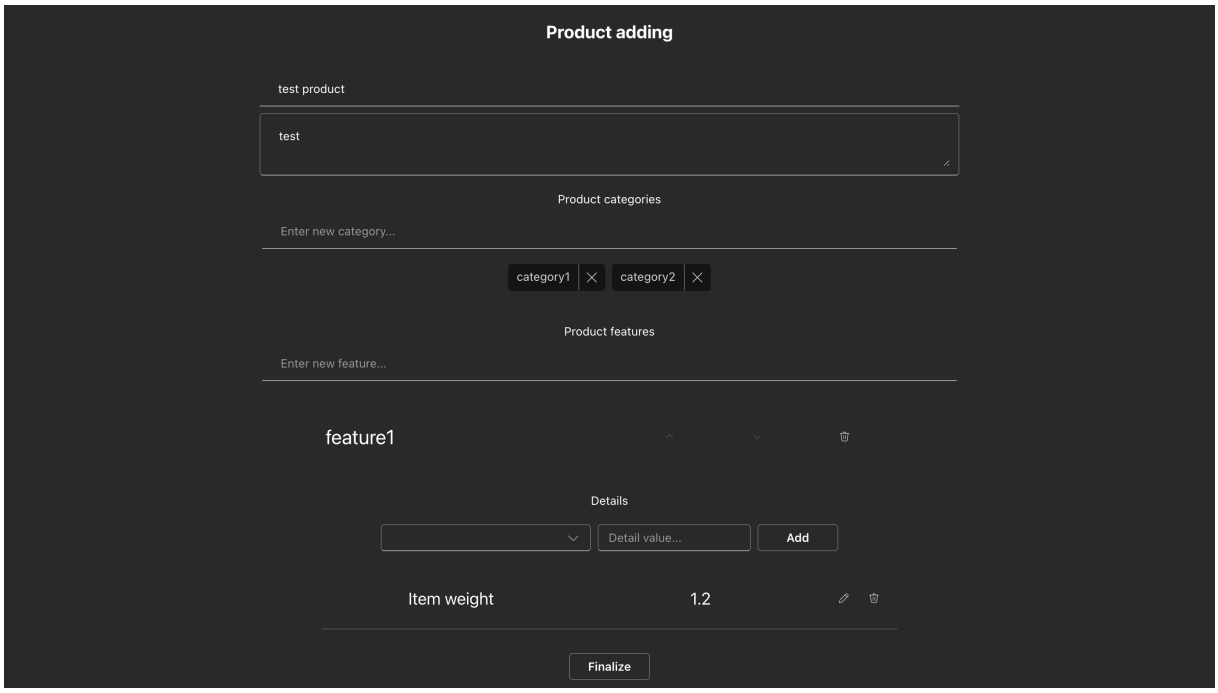


Figure 2.12: Products Managing product adding

The products-managing microfrontend is the source of the following components:

Component	Description
AddProduct	component for adding a new product
ProductsList	a component providing the functionalities of displaying and removing the products
ProductsLikes	a component for displaying the products the user likes

Table 2.8: Products-managing components

The products-managing microfrontend was implemented with the use of the libraries listed in the appendix (5.4). The backend connection is describe in the appendix 13.The products-managing microfrontend uses the standard Environmental Variables Mocks (5.1) for the testing purposes. The tests results are available in the 5.3 appendix.

user-settings The user-settings microfrontend provides the settings panel component. Below, a screenshot of it running in the application is provided.

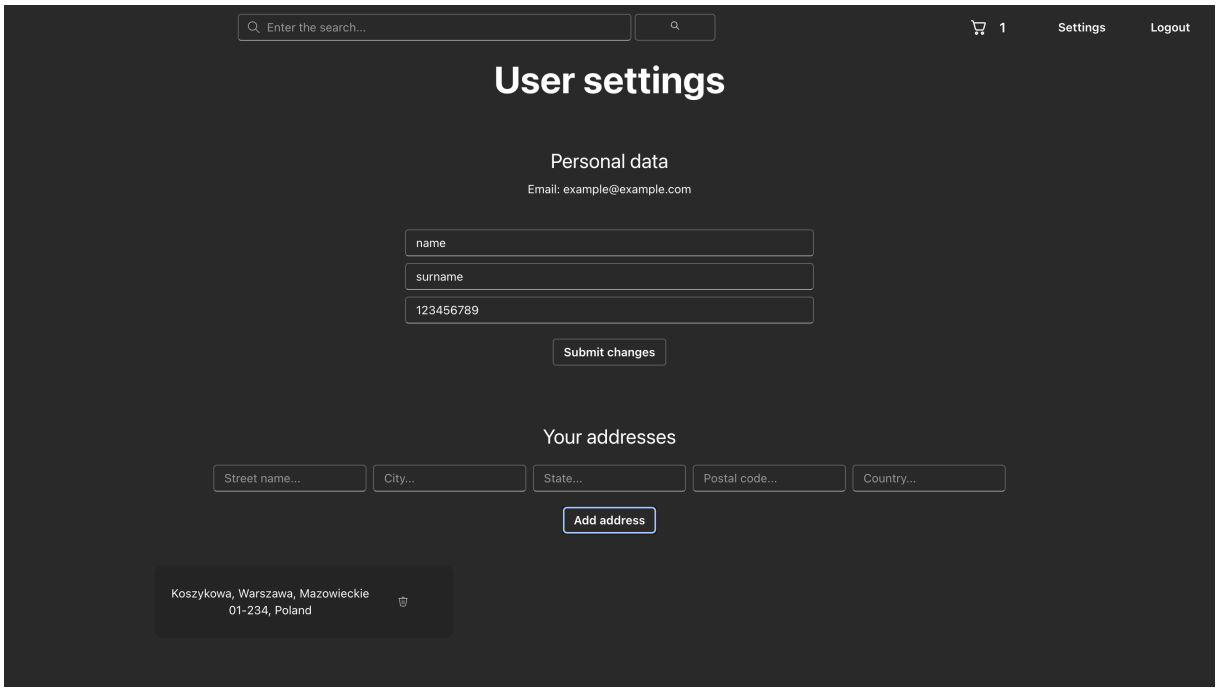


Figure 2.13: Settings panel

The user-settings microfrontend was implemented with the use of the libraries listed in the appendix (5.4). The backend connection is described in the appendix 15. The user-settings microfrontend the standard Environmental Variables Mocks (5.1) for the testing purposes. The tests results are available at 5.4 appendix.

user-basket The user-basket microfrontend provides one component, being the basket panel that displays all of the user's basket products and provides the ability to remove them. Below, a screenshot of the user basket panel is provided.

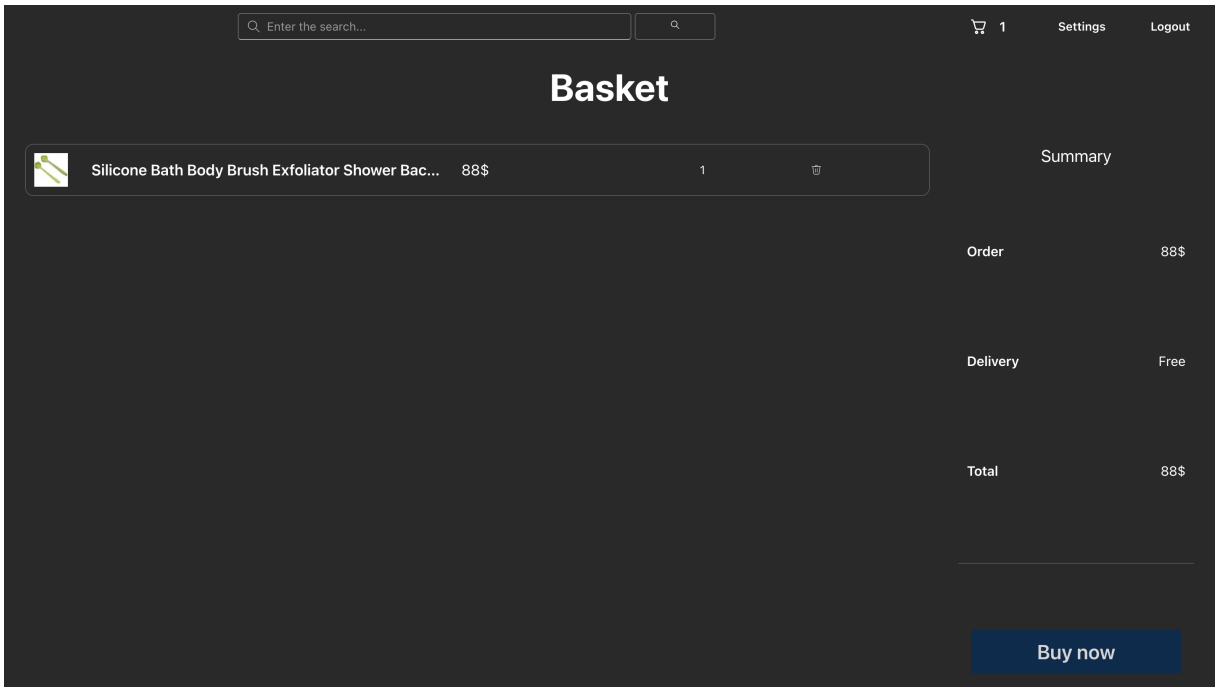


Figure 2.14: Basket panel

The user-basket microfrontend was implemented with the use of the libraries listed in the appendix (5.4). The backend connection is described in the appendix 17. For the implementation of the unit tests, the Jest testing framework was used for testing all of the .ts and .tsx files. The user-basket microfrontend uses the standard Environmental Variables Mocks (5.1) for the testing purposes. The tests results are available at the 5.5 appendix.

user-order The user-order microfrontend provides the following components:

Component	Description
Order	a component for displaying the current order and finalizing it
OrderHistory	a component for handling the user orders history

Table 2.9: User-order components

Below, a screenshot of the user order Order panel is provided.

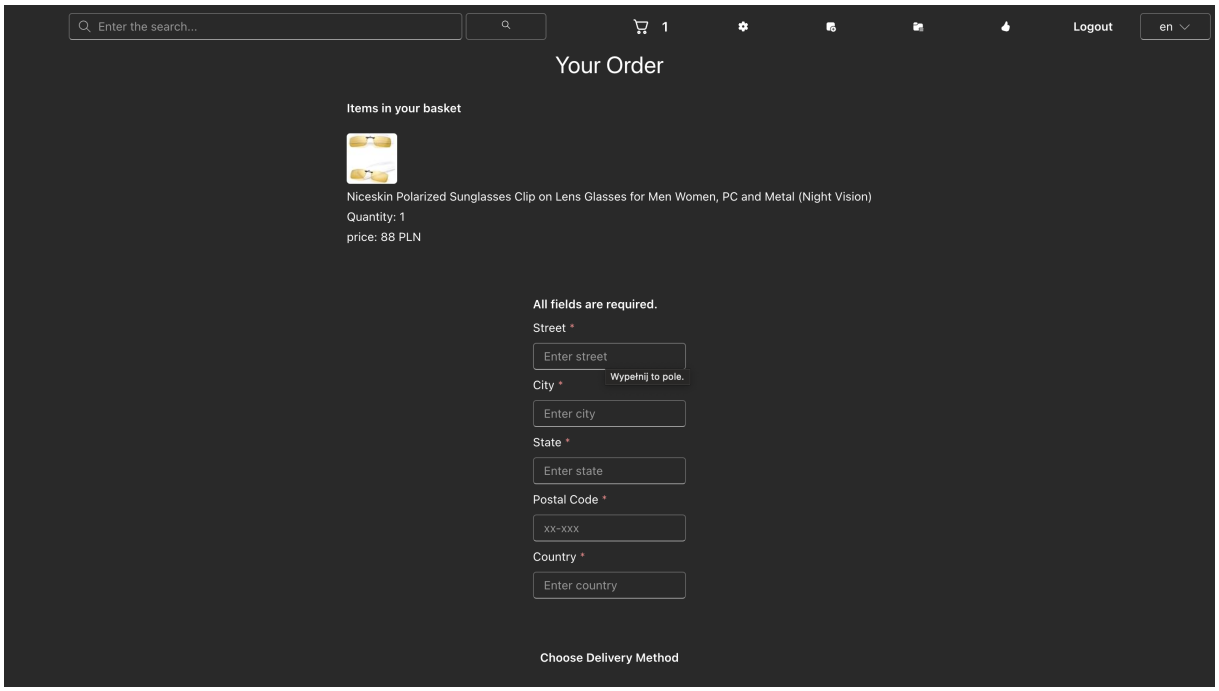


Figure 2.15: User order Order panel

The user-order microfrontend was implemented with the use of the libraries listed in the appendix (5.4). Additionally, the @stripe/react-stripe-js was used to handle the Stripe library payments. The backend connection is described in the appendix 19. The user-order microfrontend uses the standard Environmental Variables Mocks (5.1) for the testing purposes. The tests results can be found at the 5.6 appendix.

Administrator application implementation

The administrator application was implemented in separation from the main application and its microfrontends, as it was decided that making it a separate application will increase the security. The application consists of two parts:

Component	Description
Login panel	the panel for signing the admin in
Main panel	a panel for handling the application administrative tasks

Table 2.10: Administrator application components

Below, a screenshot of the Admin Main panel is provided.

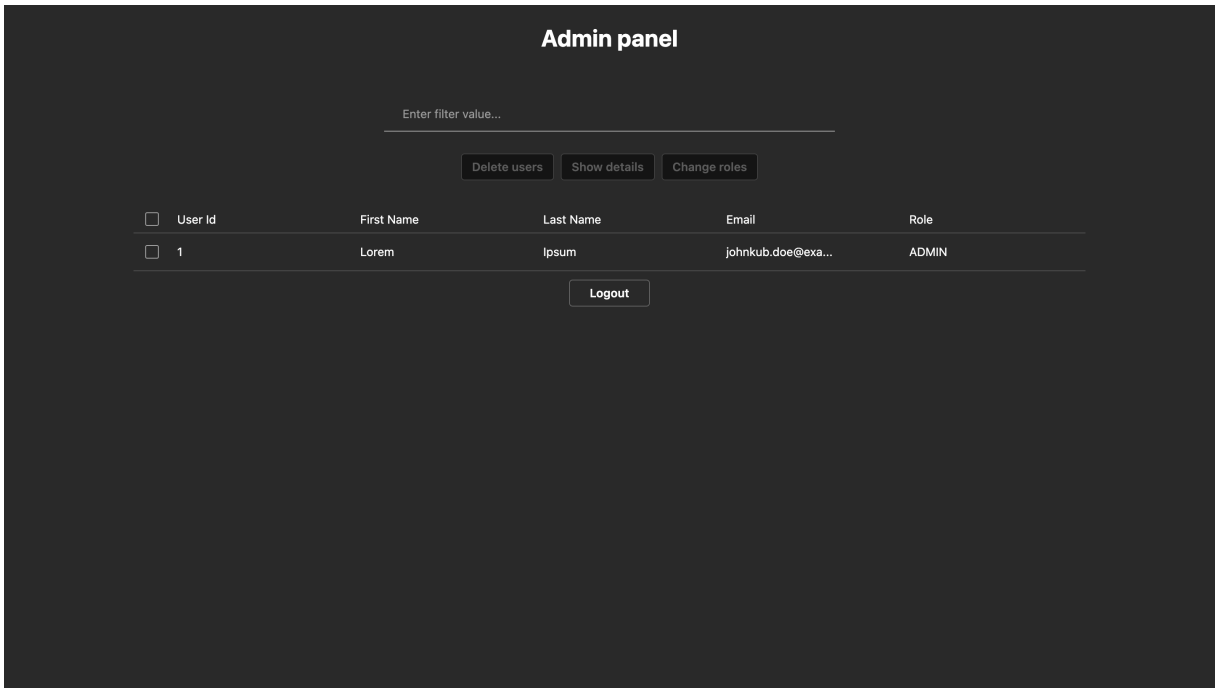


Figure 2.16: Admin Main panel

In comparison to the main application, it was decided that the implementation of the i18n internazionalization standard was unnecessary.

The administrator application was implemented with the use of the libraries described in the appendix 5.6. In terms of the backend connection, it relies on the HTTP methods described in the appendix 22. For the implementation of the unit tests, the Jest testing framework was used for testing all of the .ts and .tsx files. The Admin application uses the standard Environmental Variables Mocks (5.1) for the testing purposes. The tests results are available at the 5.7 appendix.

2.3.3. Backend Implementation

Architecture Description

As it was said, the backend system is based on a microservices architecture. Eureka is used for service registration and discovery. A central entry point(Gateway Service) handles routing, authentication, and authorization using JWT tokens. Before request will be delivered to a microservice it also is processed by load balancer. Each microservice has its own database, ensuring data isolation and service independence. Two types of communication can be found in the implementation:

- REST APIs are used for synchronous communication, as exemplified by the Gateway

Service communicating with the Auth Service to authorize user requests.

- Apache Kafka enables asynchronous, event-driven interactions, used for communication between all other microservices.

This architecture can be approximately represented by the following diagram Figure 2.17.

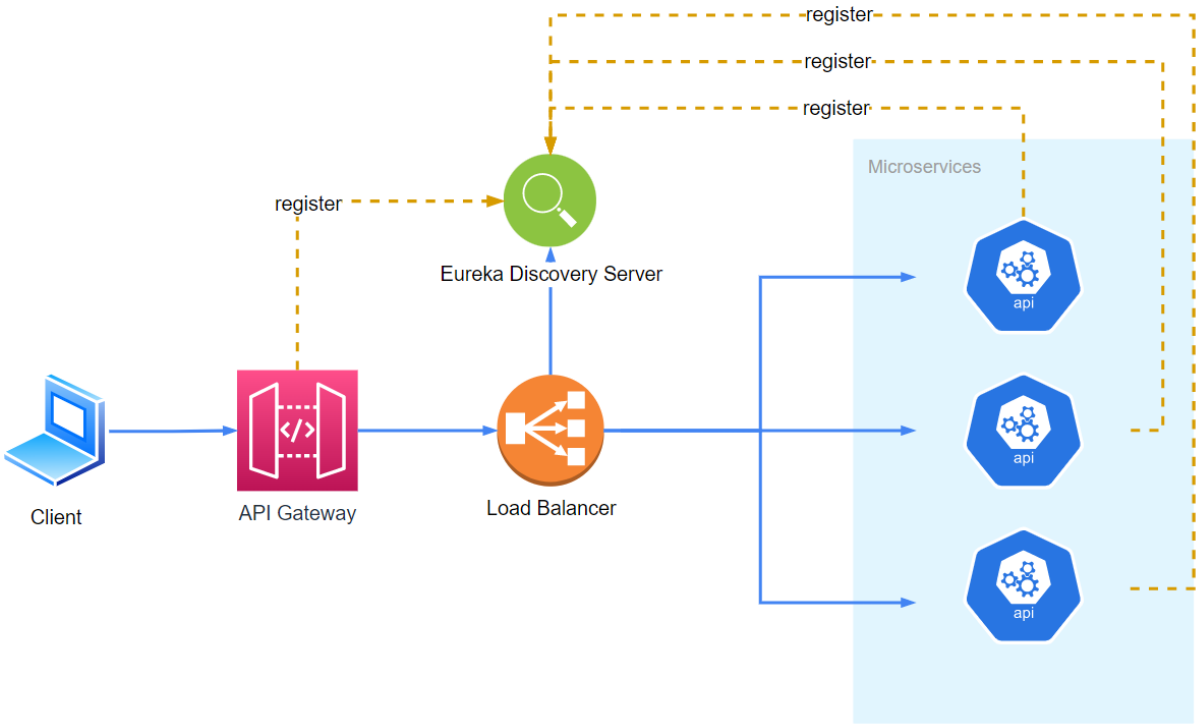


Figure 2.17: Architecture Backend Source: [18]

Communication between microservices

There are two types of communication patterns that can be observed in the implementation context of microservices:

- **Request-Response Pattern:** One microservice sends a request to a second microservice, waits asynchronously for a response, and processes the received data. Specifically, the first microservice sends the request to a defined request-topic using a Kafka producer class. Concurrently, it stores a CompletableFuture in a ConcurrentHashMap with a key (e.g., userId for which the request was sent) to track the pending response. The second microservice consumes the request, processes it by retrieving required information from the database and then publishes the response to a response-topic. The first microservice listens for responses on this topic, retrieves the corresponding CompletableFuture using the defined key , and completes

the future with the received data, enabling non-blocking and concurrent request-response handling. Timeout handling is also implemented. If no response arrives within the specified timeout (30 seconds), the CompletableFuture timeout. The exception method handles the timeout scenario by logging the error and throwing an exception. Figure representing this - (2.19)

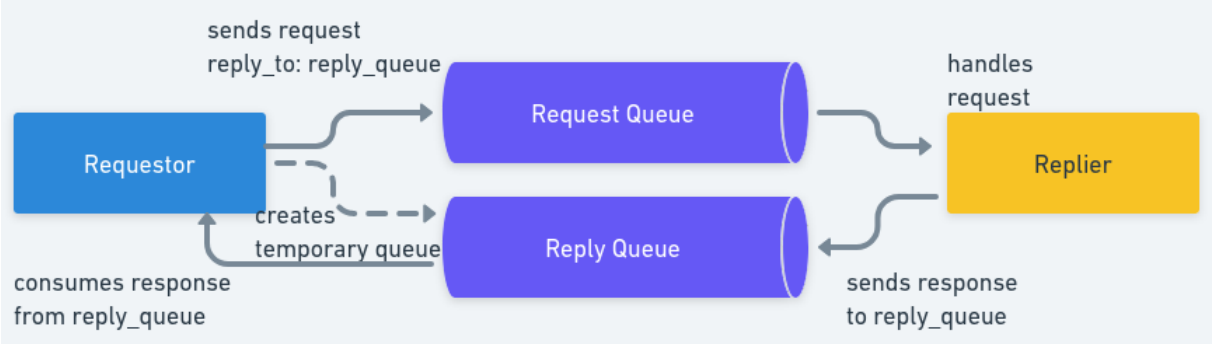


Figure 2.18: Request-Response Pattern Source:[19]

- **Event-Driven Notification:** One microservice broadcasts an event without expecting a direct response. Other microservices consume the event and take action based on its content. Figure representing this - (2.19)

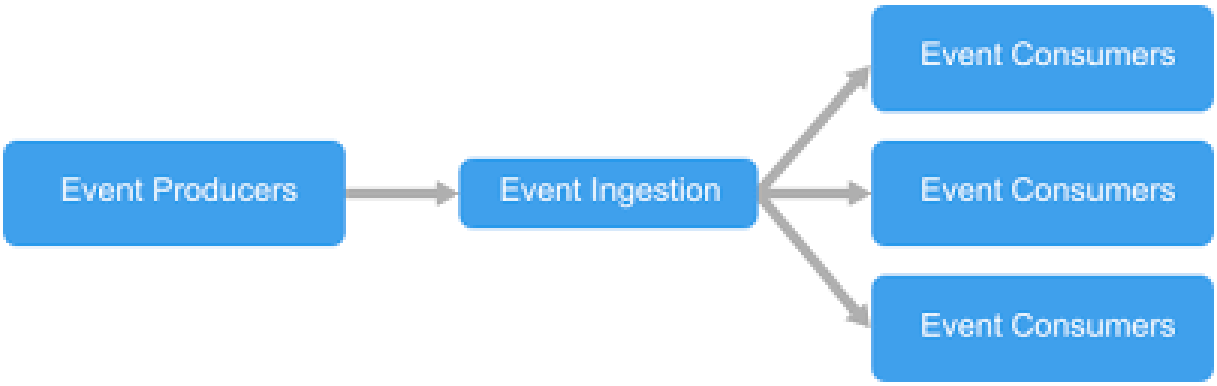


Figure 2.19: Event Driven Source:[17]

For both types of communciation listeners were configured to acknowledge message receiving manual by setting AckMode.MANUAL flag. The reason for that setting is a control over when a message is considered processed.

Microservices Communication Overview

1. The **Gateway Service** communicates with the **Auth Service** using REST APIs over the HTTP protocol. The endpoints `/authorize` and `/validate` are used for user

authentication and authorization, enabling the validation of user requests.

2. The **Auth Service** utilizes the *Request-Response Pattern* to communicate with the **Product Service** and the **Order Service**. It provides user details. The interaction is facilitated through the Kafka topics `user-request-topic` and `user-response-topic`.

3. The **Auth Service** also employs the *Event-Driven Notification Pattern* to notify the **Product Service** when a user is deactivated. These notifications are sent through the Kafka topic `user-deactivate-request-topic`, enabling the Product Service to identify products that are no longer available for purchase. This process is triggered when a user account is deleted or deactivated.

4. The **Basket Service** interacts with the **Product Service** using the *Request-Response Pattern*, requesting product details. This communication is conducted via the Kafka topics `basket-product-request-topic` and `basket-product-response-topic`. It enables the Basket Service to calculate basket prices and display essential product details, such as images, prices, and availability. The process is initiated when products are added to the basket.

5. The **Basket Service** also listens for *Event-Driven Notifications* from the **Order Service**, which are sent after a basket is successfully purchased. The notification is published to the `basket-remove-request-topic`, triggering the deletion of the corresponding basket from the Basket Service's database, as the basket's information is already stored in the Order Service. This process is initiated in the Order Service upon successful purchase completion.

6. The **Order Service** communicates with the **Basket Service** using the *Request-Response Pattern*, retrieving essential basket data. The communication is carried out via the Kafka topics `basket-items-request-topic` and `basket-items-response-topic`. This enables the Order Service to process the order with detailed product information and store the order data in its database. The mechanism is triggered when an order is created in the Order Service.

7. The **Like Service** communicates with the **Product Service** using the *Request-Response Pattern*, retrieving essential product data. The communication is carried out via the Kafka topics `product-details-request-topic` and `product-details-response-topic`. This enables the Like Service to have information about products that were liked by users. In consequence, like service can provide to client products which were liked by specific user.

Here is a diagram that summarizes and provides an overview of communication between microservices in the system(2.20).

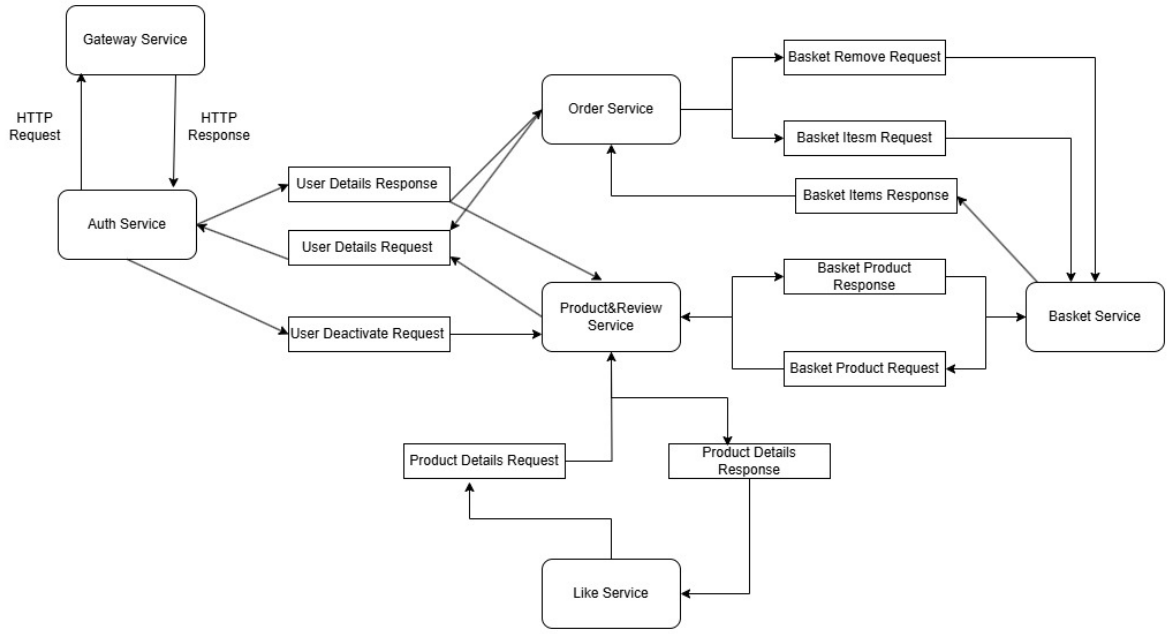


Figure 2.20: Communication between Microservices

Common Modules

Maintaining consistency in libraries, modules, design patterns, and structural approaches across microservices is essential for reducing complexity, It simplifies debugging. Reusable shared modules and patterns improves code quality, reduces duplication, and ensures uniform practices for security and testing. Due to those advantages there were declared shared modules which provide reusable functionalities. Dependency to those modules was set in every service pom.xml configuration.

Table 2.11: Main backend shared modules

Name	Description
Common-dto	Contains shared Data Transfer Object (DTO) definitions used across multiple services. These objects facilitate data exchange between services, primarily during communication via Kafka.

Continued on next page

Table 2.11 (continued)

Name	Description
Common-Utils	Provides utility functions for extracting crucial information, such as the user's UUID and email, from authentication tokens.
Exception	Implements a centralized exception-handling mechanism using the 'CustomGlobalExceptionHandler' class. This class is annotated with '@RestControllerAdvice' and standardizes exception handling across all controllers, making debugging and error management more efficient.
Common-kafka-config	Includes the 'KafkaConfigUtils' class, which provides utility methods for configuring and managing Apache Kafka producers and consumers in a Spring application. It enables the creation of customized Kafka clients by specifying properties such as bootstrap servers, serializers, deserializers, and trusted packages.
RegisterEndpointInformation	Implements the 'ApiGatewayEndpointConfiguration' interface, which is responsible for registering and managing API endpoints. Each endpoint is represented by the 'Endpoint' class, defining its URL, HTTP method (e.g., GET, POST), and the required user role (GUEST, USER, or ADMIN) for access control. Communication occurs via the HTTP protocol with endpoint accessibility definitions.

Common Implementation Patterns

Configuration Layer: Each service includes a configuration folder containing classes responsible for configuring properties of service. In this place classes for configuration:

1. Third party APIs (for example Cloudinary)
2. Spring Security
3. Kafka settings for Consumers and Listeners using Common-Kafka-Config.

4. RegisterEndpointInformationImpl configuration which implements interface define in RegisterEndpointInformation common module.

Controllers: Each microservice defines controller classes to handle specific functionalities. Endpoints are grouped with a common URL prefix to align with business logic and ensure modularity. These classes are annotated with @RestController and @RequestMapping from Spring Web, enabling them to expose RESTful APIs and handle HTTP requests. Example Controller (2.21)

```
@RestController
@RequiredArgsConstructor
@RequestMapping("/api/v1/like")
public class LikeController {
    private final LikeService likeService;

    OganKuba
    @RequestMapping(path = "/{productId}", method = RequestMethod.POST)
    public ResponseEntity<LikeResponse> addLike(@PathVariable String productId, HttpServletRequest request) {
        return likeService.addLike(productId, request);
    }

    OganKuba
    @RequestMapping(path = "/my", method = RequestMethod.GET)
    public ResponseEntity<List<ProductResponse>> getMyLike(HttpServletRequest request) {
        return likeService.getMyLikedProducts(request);
    }
```

Figure 2.21: Controller example

DTO: Each microservice defines its own set of Data Transfer Objects (DTOs), categorized into:

- **Response DTOs:** Represent data sent back to clients.
- **Request DTOs:** Represents data received from clients. These DTOs are defined with jakarta.validation annotations to ensure the incoming data is validated, preventing invalid or incomplete data from being processed. Example of DTO using validation annotations(2.22).

```java
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class LoginRequest {
    @Email
    @NotBlank(message = "Email is mandatory")
    private String email;

    @NotBlank(message = "Password is mandatory")
    private String password;
}
```

Figure 2.22: DTO example

Entity: Microservices define entities to represent database structures, mapping directly to database tables. Relationships between database tables were implemented using JPA in services using SQL databases. To optimize database operations and relationship joins, each entity uses a primary key based on a sequence, which serves as the unique identifier within the database. Entities are interconnected using this primary key, ensuring efficient joins and relationships between tables. Additionally, a UUID(UUId type random string) key is provided for external client access, while the primary key remains internal and hidden to enhance security. The following JPA annotations were used:

- **@Entity:** To define a class as a JPA entity.
- **@Table:** To specify the corresponding database table name.
- **@Id** and **@GeneratedValue:** To mark the primary key and configure its auto-generation.
- **@OneToOne, @OneToMany, @ManyToOne,** and **@ManyToMany:** To define relationships between entities.
- **@JoinColumn:** To specify foreign keys in relationships.

- **@Column:** To customize column properties such as name, length, and constraints.

Example databse definition code (2.23)

```java
@Table(name = "likes")
@Entity
@Setter
@Getter
public class Like {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "likes_id_seq")
    @SequenceGenerator(name = "likes_id_seq", sequenceName = "likes_id_seq", allocationSize = 1)
    private Long id;

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

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

    @ManyToOne
    @JoinColumn(name = "product_id", nullable = false)
    private Product product;

    @Column(name = "date_added", nullable = false)
    @Temporal(TemporalType.TIMESTAMP)
    private Date dateAdded;
```

Figure 2.23: Entity declaration example

Kafka: Each microservice implements Kafka Producers and Listeners, where:

- **Consumers:** Responsible for listening to specific topics, extracting relevant data from the messages, and performing necessary business logic. For example, this may involve querying a database, validating input, or other domain-specific operations. Consumers manually ensure reliable acknowledgment of the processed messages.
- **Producers:** Focus on preparing structured events based on processed data and sending these events to designated topics. It enables further consumption by other services.

Mapper Layer: The Mapper Layer in the system is implemented using MapStruct. Each entity and its associated Data Transfer Objects (DTOs) have a dedicated mapper interface to handle object transformations. This ensures that each mapper is focused on transforming data related to its specific domain object (e.g., User, Product, Order). Each mapper provides a static `INSTANCE` field created using MapStruct's factory method

(`Mappers.getMapper(Class)`), which acts as a single access point for mapping logic. Example of mapper(2.24)

```java
@Mapper
public interface ProductMapper {
    2 usages
    ProductMapper INSTANCE = Mappers.getMapper(ProductMapper.class);

    2 usages  1 implementation  OganKuba
    ProductResponse toProductResponse(Product product);

    3 usages  1 implementation  OganKuba
    @Mapping(target = "owner", source = "user")
    ProductDetailResponse toProductDetailResponse(Product product, User user);

    1 usage  1 implementation  OganKuba
    @Mapping(target = "productId", source = "parentAsin")
    ProductEvent toProductEvent(Product product);

    1 usage  1 implementation  OganKuba
    List<ProductResponse> toProductResponseList(List<Product> products);


    1 usage  1 implementation  OganKuba
    @Mapping(target = "videos", ignore = true)
    @Mapping(target = "userId", source = "userId")
    @Mapping(target = "ratingNumber", constant = "0")
    @Mapping(target = "parentAsin", source = "uuid")
    @Mapping(target = "isActive", constant = "true")
    @Mapping(target = "images", ignore = true)
    @Mapping(target = "id", ignore = true)
    @Mapping(target = "boughtTogether", ignore = true)
    @Mapping(target = "averageRating", constant = "0.0")
    Product toProduct(AddProductRequest addProductRequest, String uuid, String userId);

}
```

Figure 2.24: Mapper Example

Repository Layer

SQL SQL-based repositories extending `JpaRepository`. Custom queries are implemented using JPQL or native SQL with the `@Query` annotation. To gain more control over each method, the @Query annotation was frequently used to tailor queries to specific requirements and better code clarity. Example(2.25).

```java
@Repository
public interface LikeRepository extends JpaRepository<Like, Long> {

    2 usages  OganKuba
    @Query("SELECT CASE WHEN COUNT(l) > 0 THEN TRUE ELSE FALSE END FROM Like l WHERE l.userId = :userId AND l.product.uuid = :productId")
    boolean existsByUserIdAndProductId(String userId, String productId);

    1 usage  OganKuba
    @Query("SELECT COUNT(l) FROM Like l WHERE l.product.id = :productId")
    Long countByProductId(Long productId);

    1 usage  OganKuba
    @Modifying
    @Transactional
    @Query("DELETE FROM Like l WHERE l.uuid = :uuid")
    int deleteByUuid(String uuid);

    1 usage  OganKuba
    @Query("SELECT l FROM Like l WHERE l.uuid = :uuid")
    Optional<Like> findByUuid(String uuid);

    1 usage  OganKuba
    @Query("SELECT CASE WHEN COUNT(l) > 0 THEN TRUE ELSE FALSE END FROM Like l WHERE l.product.id = :id")
    boolean existsByProductId(long id);
}
```

Figure 2.25: Jpa Repository

NoSQL NoSQL-based services use MongoDB, with repositories extending MongoRepository for handling unstructured data. Custom repository methods allow dynamic querying, pagination, and sorting. This approach ensures that specific query requirements are integrated into the repository layer, maintaining clean and modular code.

Service Layer The Service Layer follows a service-interface pattern, where interfaces define the business logic contract(e.g ProductService) and their implementations(e.g ProductServiceImpl). (Visualization 2.26)

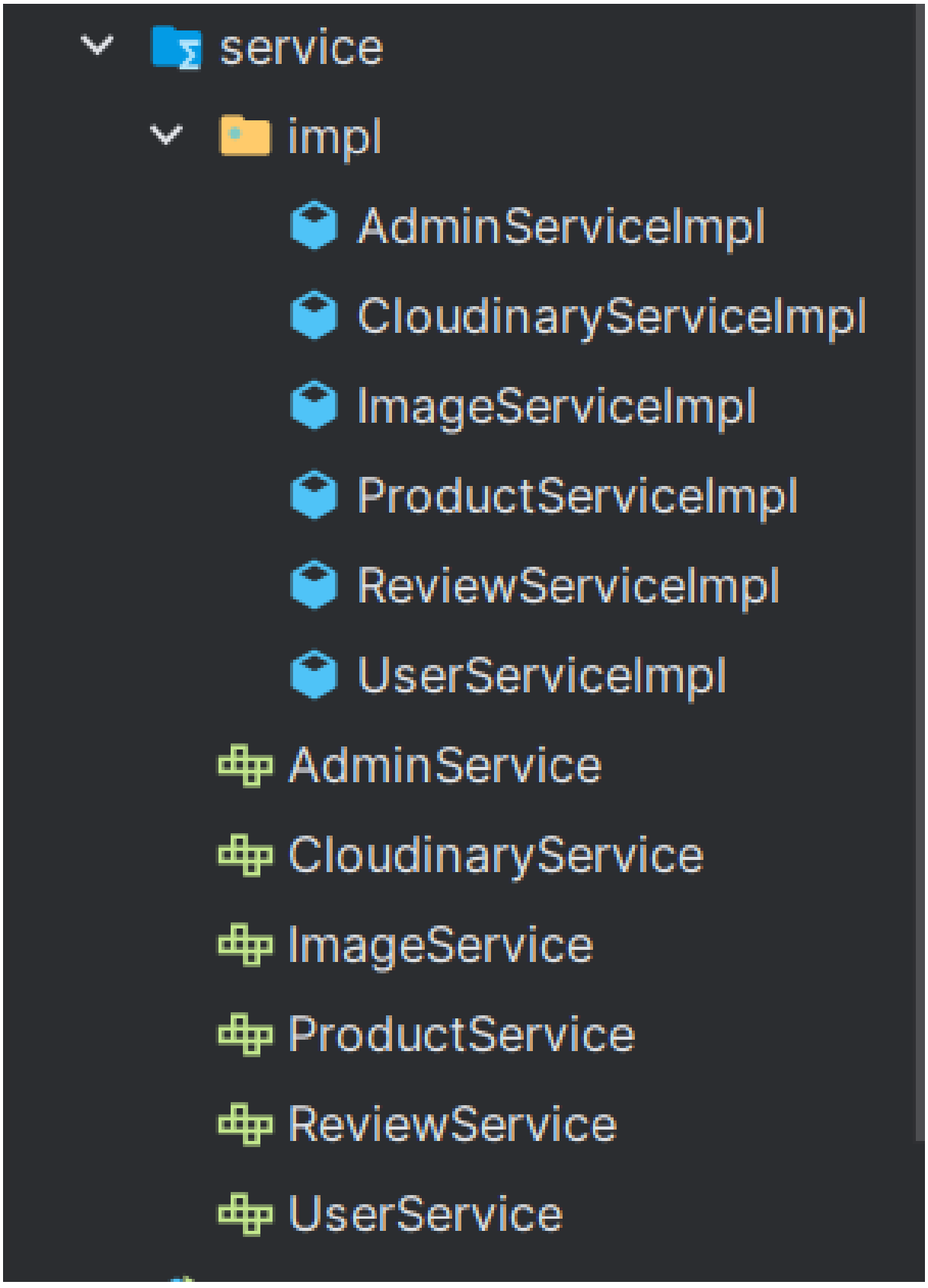


Figure 2.26: Services Structure

Microservices:

1.Gateway Service

Functionalities: Gateway service acts as the single entry point for client requests, handling authentication, authorization, and routing.

Routing Configuration: The Gateway Service leverages Spring Cloud Gateway to manage routing and request redirection to various microservices. Routes are configured in the application.properties file, where each route specifies the path and target microservice. This configuration ensures seamless and efficient traffic management by directing incoming requests to the appropriate microservices based on defined rules. The configuration for route includes:

Table 2.12: Gateway Route Configuration

Parameter	Description
id	A unique identifier assigned to each route.
uri	Specifies the target destination for the route in the format ‘uri=lb://service-name‘. The ‘lb://‘ prefix indicates that the gateway should use a load balancer to direct requests to one of the available instances of the specified service. The ‘service-name‘ must match the name of a service registered in the Eureka Discovery Server.
Predicates	Define the URL patterns that determine which requests are routed to specific microservices. Each microservice has a unique URL prefix used for routing. For example, all requests starting with ‘/api/v1/auth/**‘ are routed to the ‘AUTH-SERVICE‘.
Filters	Specify actions to be performed on requests before they are forwarded to the service. In this case, an Authentication Filter is implemented to process requests and verify their authenticity before routing them.

Example(2.27).

```
spring.cloud.gateway.routes[1].id=PRODUCT-SERVICE
spring.cloud.gateway.routes[1].uri=lb://PRODUCT-SERVICE
spring.cloud.gateway.routes[1].predicates[0]=Path=/api/v1/product/**
spring.cloud.gateway.routes[1].filters[0]=AuthenticationFilter
```

Figure 2.27: Gateway Config

Authentication and Authorization: The AuthenticationFilter responsible for ensuring secure access to services. Its implementation manages the following:

- **Token Validation** Extracts the Bearer token from the Authorization header and validate the token's signature and extract user information (e.g., UUID).
- **Integration with AUTH-SERVICE** Based on the requested endpoint, the filter determines which AUTH-SERVICE endpoint to call. For admin routes, it calls using HTTP protocol /api/v1/auth/authorize and for endpoints, which require only token it calls /api/v1/auth/validate.
- **Dynamic Token Handling** Upon successful validation, the filter updates the Authorization and Refresh-Token headers with newly generated tokens from the AUTH-SERVICE. If validation fails, it returns a 401 Unauthorized error response to the client.

Instance Management and Load Balancing: In the Gateway implementation, a custom Carousel was created specifically for AUTH-SERVICE. It manually cycles through available authentication service instances using a round-robin approach and updates instantly when instances register or unregister in Eureka. For all other services, the Spring Cloud LoadBalancer is used, which automatically distributes traffic without custom logic.

Route Validation: The **RouteValidator** class ensures proper validation of requests to determine whether they require authentication or admin-level permissions. Endpoints are categorized into two groups: **Public Endpoints (openApiEndpoints)**, such as `/auth/register` and `/auth/login`, which are accessible to guests without authentication, and **Admin Endpoints (adminEndpoints)**, like `/api/v1/auth/admin/**`, which require admin permissions. Validation is performed using two methods: isSecure - Determines if a request is targeting a public endpoint, isAdmin if a request matches an admin-protected endpoint.

Dynamic Route Updates: The RegistrationController provides an API to dynamically add new endpoints to RouteValidator at runtime. It works that receives a list of Endpoint objects via a POST HTTP request, that is send from each microservice's RegisterEndpointInformation. Then adds the endpoints to the appropriate sets (adminEndpoints or openApiEndpoints) based on their asigned roles.

Integration with Eureka Discovery: The Gateway is registered as a client with Eureka, allowing it to dynamically retrieve service information and discover available service instances.

2.Authentication service

Functionalities The Authentication Service provides user authentication, authorization, and account management functionalities. It facilitates authorization and validation endpoints. Which are used in Gateway service. It also supports user registration and login with secure token-based validation. Additionally, Admin functionalities include retrieving user details, managing user roles, and deleting user accounts via /api/v1/auth/admin. The service also allows users to update personal data, manage addresses, and handle profile images. Due to this service, users are also able to change passwords and deactivate their accounts.

Database: The database schema(2.28) consists of two main entities: User and Address, representing the core structure of the authentication and user management system. These entities are connected via a One-to-Many relationship, where a single user can have multiple associated addresses. They are linked using foreign key constraint on user_id in the addresses table.

User Table: The User entity is mapped to the users table and contains key attributes such as id (primary key), uuid (globally unique identifier), email (primary identifier), password (hashed for authentication), role (user role), and boolean flags (isEnabled, isLock, isActive) to track account status.

Address Table: The Address entity is mapped to the addresses table and includes key attributes such as id (primary key), uuid (globally unique identifier), userid (foreign key linking to the User table in a One-to-Many relationship), and address-related fields like street, city, state, postalCode, and country.

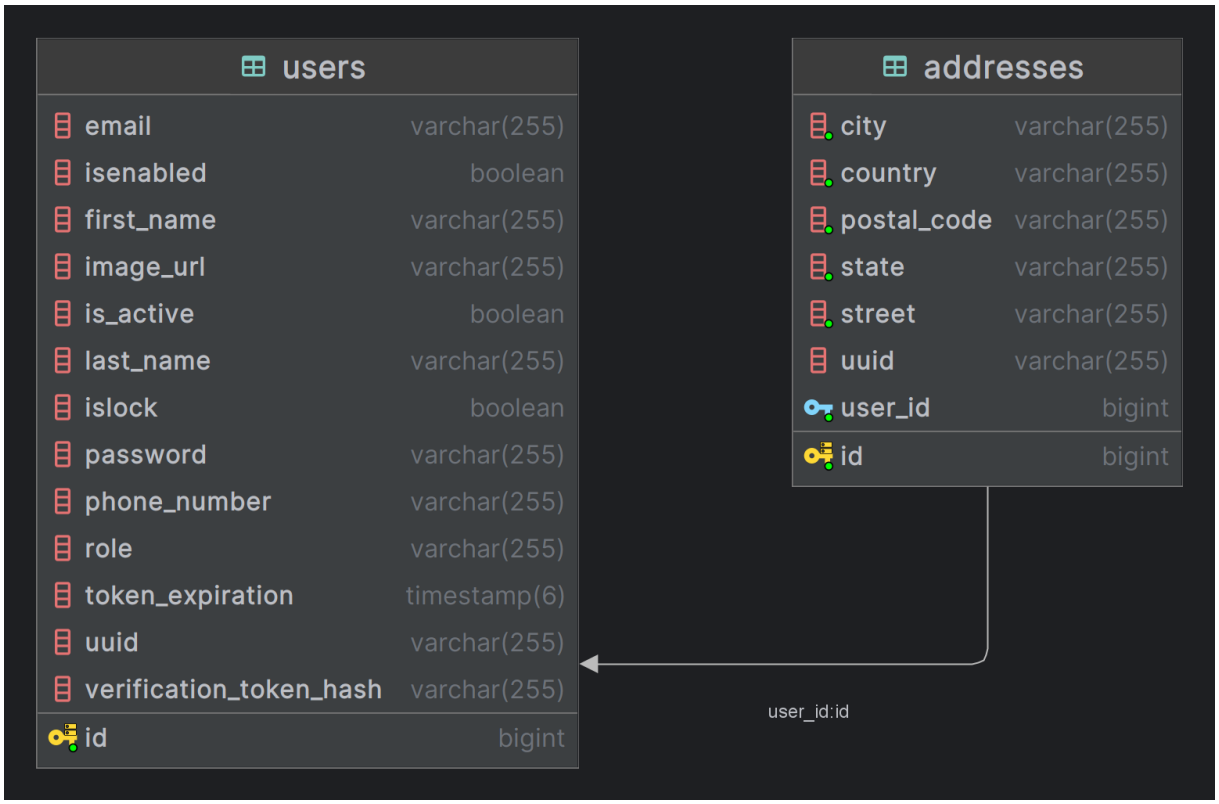


Figure 2.28: Auth Database Scheme

Supporting Services

- **Image and Cloudinary Service**: The Cloudinary service is responsible for handling interactions with the Cloudinary API for image management, including uploading and deleting images. It is implemented using Cloudinary's Java SDK, which provides methods for uploading images (uploadImage) and deleting images (deleteImage). Additionally, implementation facilitate handling multipart file uploads, converting them to temporary files before passing them to the CloudinaryService.
- **Jwt Service**: Provides JSON Web Token management which is a more extensive version of the common-utils module. It generates tokens using the HS256 algorithm, embedding claims like email and UUID, and sets expiration. The signing key is derived from a Base64-encoded secret. The service also extracts tokens from HTTP request headers and retrieves specific claims using JWT parsing utilities.

Rest API Endpoints Appendix REST API endpoints: 25

Product & Review Service

Functionalities This service is responsible for management of products and reviews.

Database The NoSQL database is structured(2.29) around two primary collections: metaHealthandPersonalCare(Products) for storing product information and reviewsHealthandPersonalCare(Reviews) for storing product reviews. These collections are connected through the parent_asin field, which acts as a reference identifier, linking reviews to their corresponding products.

Product: The Product entity is stored in the metaHealthandPersonalCare collection and includes key fields such as id (unique identifier), userid (owner of the product), parentAsin (linking to reviews), and category-related attributes for organization. It also contains descriptive fields (description, details, features), media lists (images, videos), pricing and rating data, and an isActive flag to indicate product availability in the system.

Review The Review entity is stored in the reviewsHealthandPersonalCare collection and includes key fields such as id (unique identifier), asin (review-specific ID), and parentAsin (linking to a product in the Product collection). It captures user feedback through rating, text, userid, helpfulvote, and verifiedPurchase, while also storing associated media (images) and a timestamp marking the review's submission time.

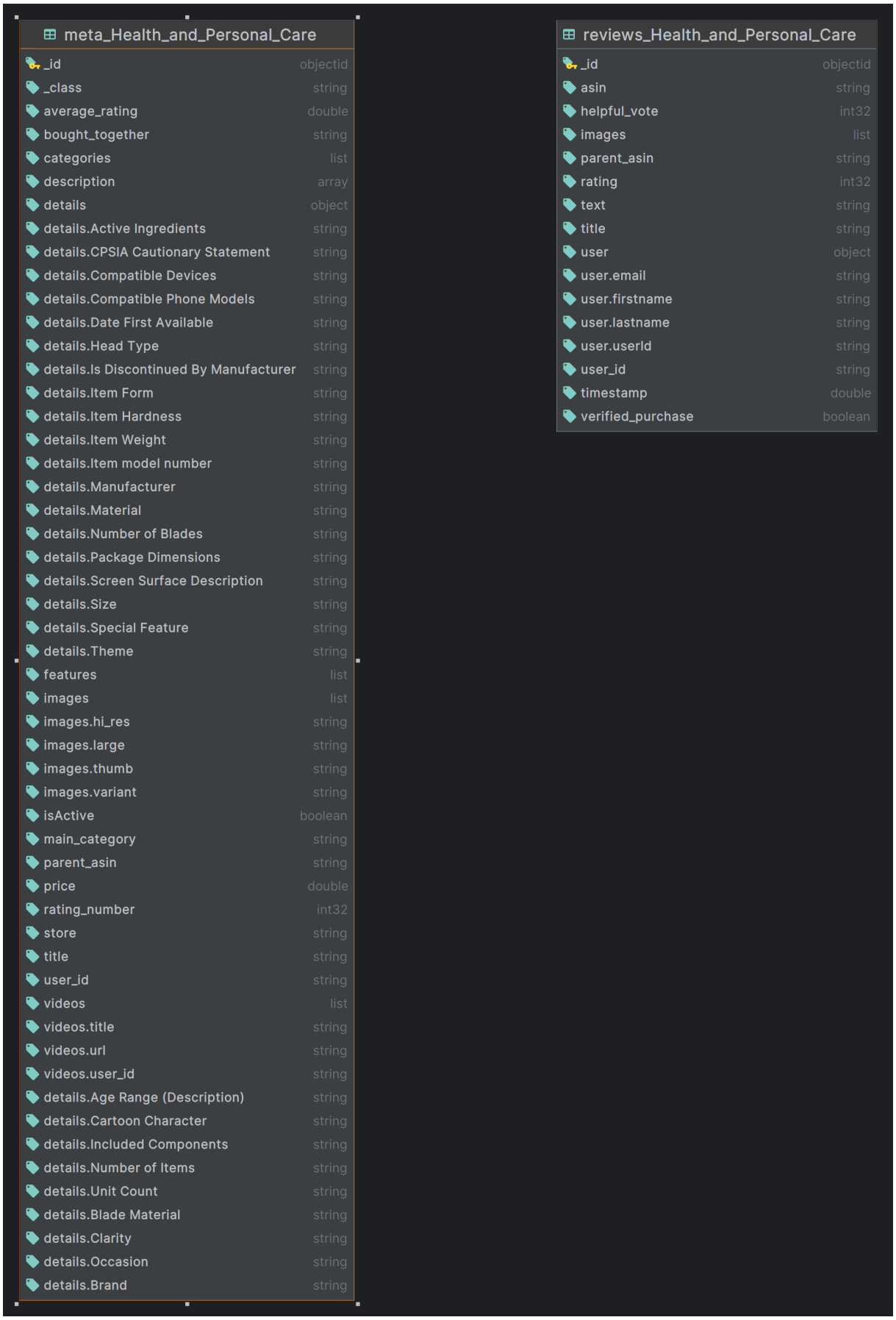


Figure 2.29: Product and Review Database Scheme

Supporting Services The BasketService provides functionality to manage a user's shopping basket. It allows adding, deleting and retrieving products in a basket.

Rest API Endpoints Endpoints in appendix 25a

Like Service

Functionalities The LikeService in this application provides functionality for managing "likes" on products. Users can add a like to a product using its product ID and retrieve a list of products they have liked. The service also allows querying the total number of likes for a specific product and checking if a particular product is liked by the user. Additionally, users can remove their like from a product using the unique like ID.

Database The database schema(2.30) is designed to manage information about products, likes, and their associated images. It consists of three primary tables: products, likes, and images. Based on schema, we can observe relationships where a One-to-Many link between products and images allows multiple images per product, while a Many-to-One relationship between likes and products associates user likes with specific products. All relationships are established using the primary keys of entities as foreign keys in related tables.

Products Table: The products table stores product details, including id (primary key), uuid, title, price, and rating-related fields (ratingNumber, averageRating).

Images Table: The images table holds product image data with fields like thumb, large, variant, and hiRes for different resolutions.

Likes Table: The likes table tracks user interactions with products, storing id, uuid. It also includes a date_added timestamp to record when the like was made.

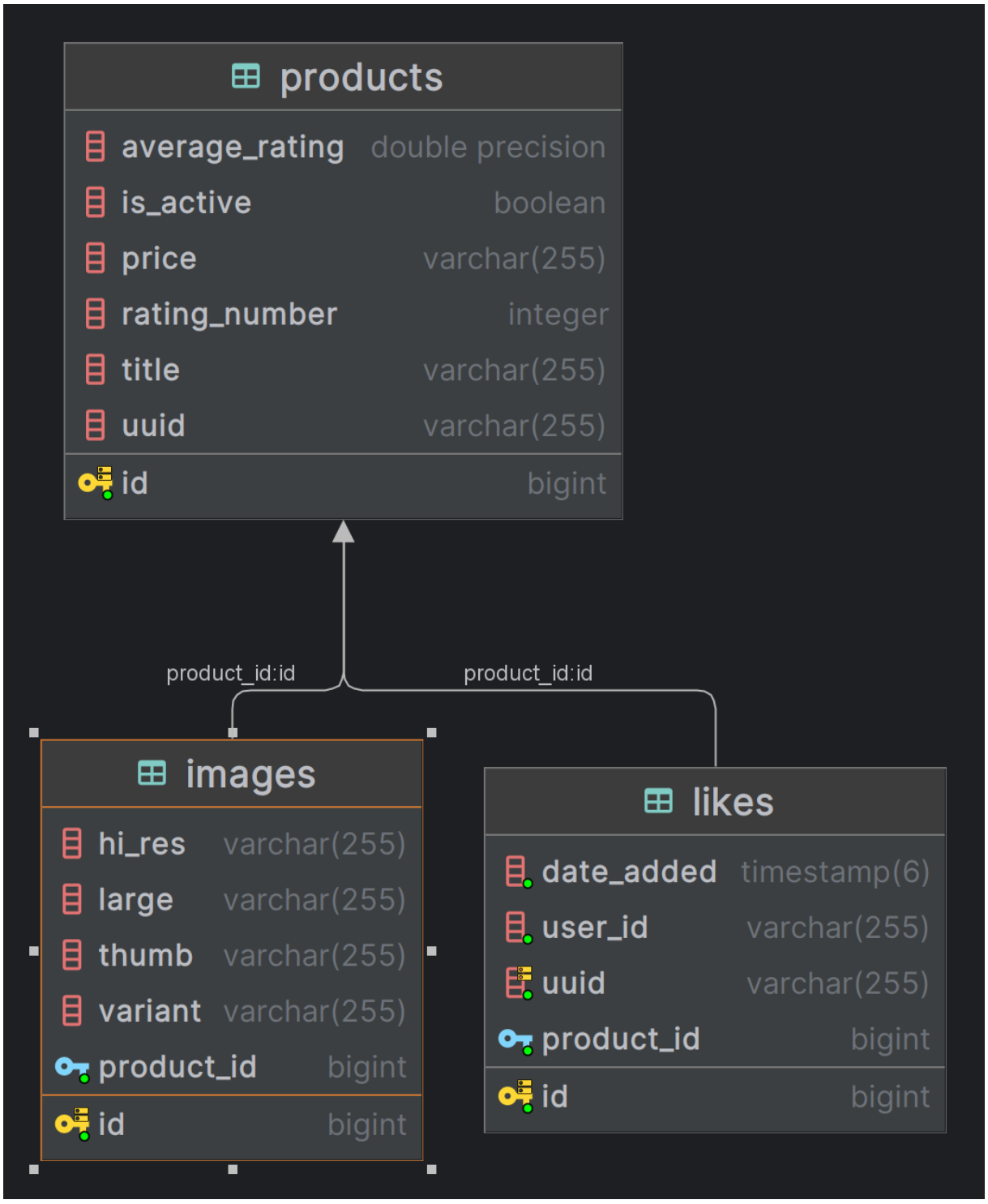


Figure 2.30: Like Database Scheme

Supporting Services There is one supporting service, which is the Product Service, responsible for separation of logic for retrieving product details from the Product Service.

Rest API Endpoints Endpoints in appendix 25b

Basket Service

Functionalities The BasketService provides functionality to manage a user's shopping basket. It allows adding, deleting and retrieving products in a basket.

Database The Basket service database schema(2.31) consists of two main tables: Basket and BasketItems. There is defined One-to-Many relationship between the Basket and BasketItems tables, which means that each basket can contain multiple items, while each item belongs to a single basket. This relationship is enforced through a foreign key constraint on the basket field in the BasketItems table.

Basket Table The Basket table represents a user's shopping basket, containing key fields such as id (primary key), uuid (globally unique identifier), and ownerId (user who owns the basket).

BasketItems Table The BasketItems table stores individual products added to a basket, including id, uuid, product (reference to the product), quantity, and basket_id, which serves as a foreign key establishing a Many-to-One relationship with the Basket table.

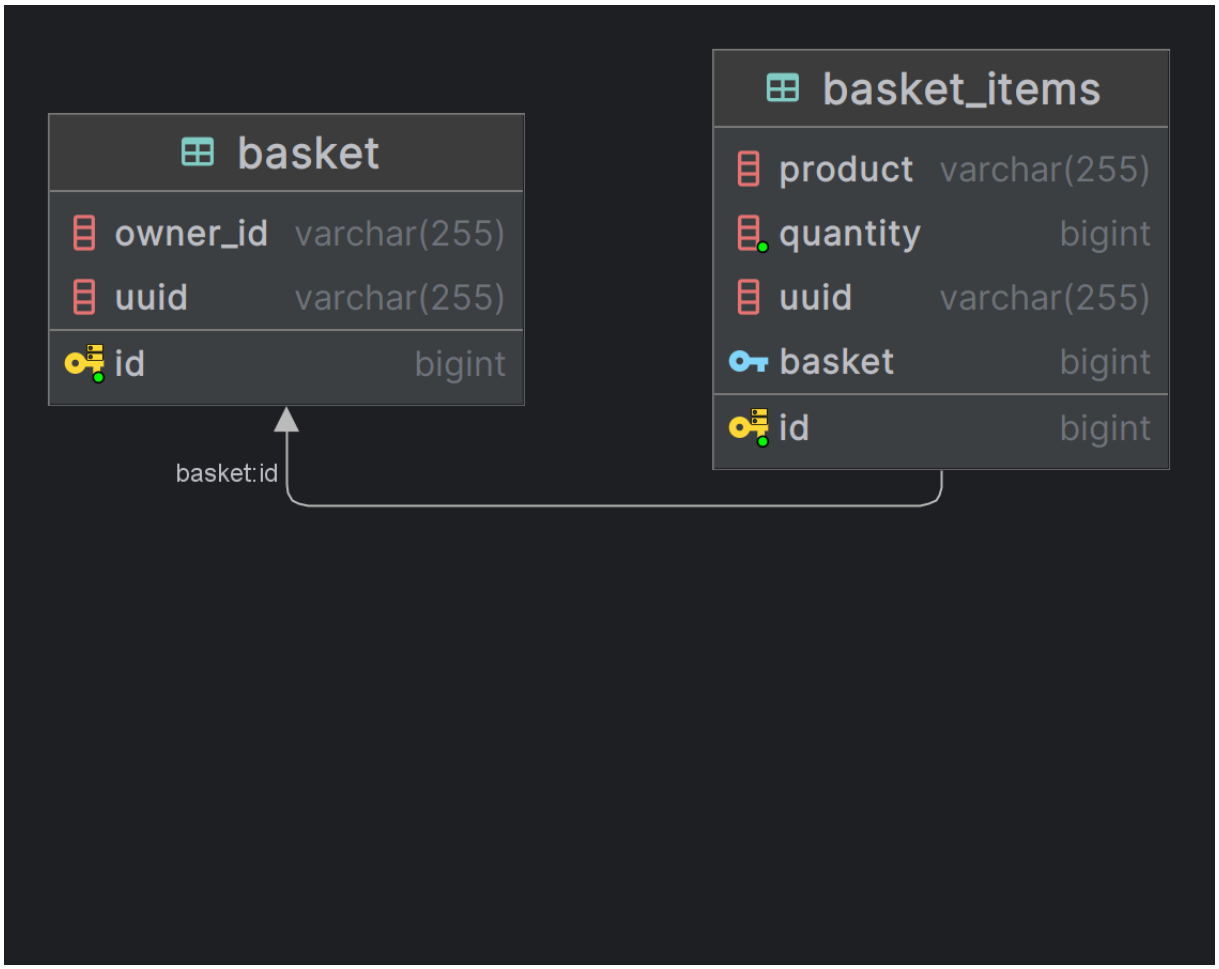


Figure 2.31: Basket Database Scheme

Supporting Services There is one supporting service, which is the Product Service, responsible for the service and separation of logic for retrieving product details from the Product Service.

Rest API Endpoints Endpoints in appendix 25c

Order Service

Functionalities The OrderService provides functionalities to manage orders and their delivery processes. It allows the creation of new orders using Stripe API. The service supports retrieving specific orders by their unique ID and fetching all orders associated with a particular client. It also includes delivery management by allowing the creation of delivery methods.

Payments The payment process is based on the Stripe API. After successfully fetching basket information, a Stripe PaymentIntent is created for the order. This initializes a payment request, attaches the order ID as metadata, and returns the client secret, which is used on the frontend to complete the payment. The order status is updated after a successful payment on the frontend and this information is delivered via the /notify endpoint. This endpoint is also responsible for notifying the basket service to delete the successfully processed basket.

Database The Order service database schema(2.32) consists of tables: orders, order_items, and deliver, with One-to-Many relationship between orders and order_items, allowing each order to contain multiple items. Additionally, a Many-to-One relationship links orders to deliver, associating each order with a specific delivery method.

Orders Table The orders table stores customer order details, including id (primary key), uuid, status, and summaryPrice (total price).

OrderItems Table The order_items table records individual products within an order, storing id, uuid, product (product identifier), quantity, pricing details (priceUnit, priceSummary), and an imageUrl. I

Deliver Table The deliver table manages available delivery options, containing id, uuid, name (delivery method name), and price (cost of the delivery).

Figure 2.32: Order Database Scheme

Supporting services There are two helping services which are the Basket Service, responsible for the service and separation of logic for retrieving product details which are in the current user basket from the Basket Service. And similar User service responsible for the service and separation of logic for retrieving user details from the User Service.

Rest API Endpoints 25d

Testing

Unit tests were implemented using a combination of Mockito for mocking service dependencies and in-memory databases (H2 for PostgreSQL services) to simulate database operations without the need for external configurations. This allowed real database interactions, ensuring the applications behavior closely mirrors production conditions. For NoSQL services using MongoDB, the use of mocks replaced direct database calls to simplify testing while maintaining logical verification of CRUD operations.

The test structure followed clear principles, including mocking repositories, utility classes, and external services to isolate the tested service. Assertions were used to verify expected outcomes, while realistic scenarios like pagination, sorting, and filtering were tested with mock data. Additionally, external interactions such as Kafka events and HTTP requests were simulated using mock frameworks.

2.3.4. AI Recommendation implementation

Overview

The recommendation system is designed to enhance user experience by providing personalized suggestions for products based on their attributes and user interactions. It leverages the large-scale **Amazon Reviews dataset (2023, McAuley Lab)**[9], which includes features like:

- **User Reviews:** Ratings, text reviews
- **Item Metadata:** Descriptions, price, and list of image urls

The project implements two independent recommendation systems as mentioned in the architecture part:

1. **Content-Based Filtering**
2. **User-Based Collaborative Filtering**

The following parts will show the practical implementation of the aforementioned systems in the key areas

Data Storage & Management

ChromaDB Integration ChromaDB is used to store vectorized embeddings of product features for content-based filtering. The system combines various product features

into vectors and an embedding model is used to generate numerical vectors, which are then stored in ChromaDB The **Figure 2.33** shows this process:

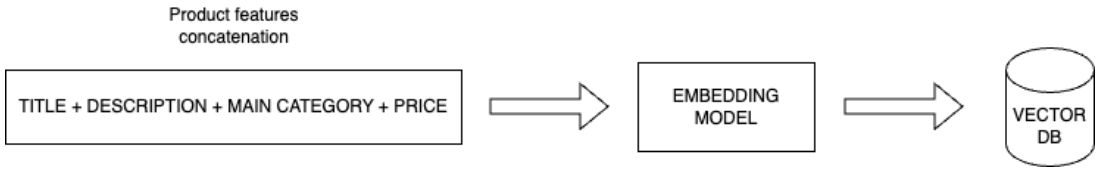


Figure 2.33: Data Flow into ChromaDB for Storing Product Embeddings

MongoDB Integration MongoDB serves as the primary database for storing product metadata such as titles, categories, prices etc.

MySQL Integration MySQL is used to store latent factors obtained from collaborative filtering models. These factors represent user and item preferences, enabling personalized recommendations. The database schema includes tables for users, items, and their latent factors. It is stored this way to easily restore the original matrix. The **Figure 2.34** shows the storage of latent factors for the user and **Figure 2.35** show it for the items.

id	user_id	factor_index	value
1	AE25NQAZI3725GZIL5FS52ZIKWKQ	0	-0.00000000335398
2	AE25NQAZI3725GZIL5FS52ZIKWKQ	1	0.000000000817997
3	AE25NQAZI3725GZIL5FS52ZIKWKQ	2	-0.0000000455404
4	AE25NQAZI3725GZIL5FS52ZIKWKQ	3	0.0000000985964
5	AE25NQAZI3725GZIL5FS52ZIKWKQ	4	0.0000000762811
6	AE25NQAZI3725GZIL5FS52ZIKWKQ	5	0.00000021867
7	AE25NQAZI3725GZIL5FS52ZIKWKQ	6	-0.000000507463
8	AE25NQAZI3725GZIL5FS52ZIKWKQ	7	0.00000180784
9	AE25NQAZI3725GZIL5FS52ZIKWKQ	8	-0.00000574937
10	AE25NQAZI3725GZIL5FS52ZIKWKQ	9	-0.00000194195
11	AE25NQAZI3725GZIL5FS52ZIKWKQ	10	-0.00000388341
12	AE25NQAZI3725GZIL5FS52ZIKWKQ	11	0.00000143139
13	AE25NQAZI3725GZIL5FS52ZIKWKQ	12	0.0000146698
14	AE25NQAZI3725GZIL5FS52ZIKWKQ	13	-0.0000147022

Figure 2.34: User latent factors in database

id	item_id	factor_ind...	value
1	B00005Q50D	0	-0.0000000000004428
2	B00005Q50D	1	0.0000000000097877
3	B00005Q50D	2	-0.0000000000342255
4	B00005Q50D	3	0.0000000000412566
5	B00005Q50D	4	0.0000000000763545
6	B00005Q50D	5	0.0000000000466315
7	B00005Q50D	6	0.000000000396988
8	B00005Q50D	7	0.000000000617465

Figure 2.35: Item latent factors in database

Preprocessing & Data Cleaning

To create high quality recommendations the product data used was preprocessed and cleaned up.

Techniques for Data Cleaning

- **Handling Missing Values**: Entries with missing fields such as ‘ID‘ or ‘Title‘ are removed using the `NullValueCleaner`.
- **Normalizing Text**: Text fields are cleaned using the `TextContentCleaner`, which removes unwanted characters and stop words.
- **Short Text Removal**: Entries where ields with too short content like descriptions with less than four words are filtered out using the `ShortTextCleaner`.

Examples of Cleaners

ShortTextCleaner The `ShortTextCleaner` returns None for entries with fields with insufficient content (e.g., fewer than 4 words). Then such an entry will not be taken into account. Its pseudocode representation in **Algorithm 1**:

Content-based filtering

Item feature representation In this implementation, item features are represented as embeddings by weighting and concatenating key attributes to form a single vector. The following weights are assigned to features based on their importance:

Algorithm 1 Short Text Cleaner

Input: `value` (string or list of strings), `min_words` (minimum word count, default = 4)
Output: Cleaned text or list of cleaned text
Define `clean_text(text)`: Return `text` if it has at least `min_words`, otherwise return `None`
if `value` is a string **then**
    return `clean_text(value)`
else if `value` is a list of strings **then**
    return List of cleaned strings from `value` where `clean_text` is not `None`
else
    return `None`
end if

Product Attribute	Weight
Title	0.4
Description	0.3
Main Category	0.2
Price	0.1

Table 2.13: Feature Weights

Features like `TITLE` and `DESCRIPTION` dominate due to their higher semantic relevance, while `MAIN_CATEGORY` and `PRICE` provide supplementary context.

The embeddings are generated using the following method, which processes each feature based on its type (text or numeric), applies the corresponding weight, and concatenates the results into a single vector as shown on the **Algorithm 2**:

Explanation:

- For each feature in the `data` dictionary, the corresponding weight is retrieved from `my_weights` different weights assigned are displayed on the **Table 2.13** .
- Text features like `TITLE` or `DESCRIPTION` are embedded using the **embedding model** [20], weighted, and added to the vector.
- Numeric features like `PRICE` are scaled using the weight and appended as single values.
- The final embedding is a concatenated vector combining all weighted components

Algorithm 2 Weighted Embedding Generation

Input: `data` (a dictionary with fields and values), `weights` (field weights), `embedding_model`

Output: `concatenated_vector` (a weighted concatenated embedding)

Initialize an empty list `vector`
Initialize an empty list `used_fields`
Initialize an empty dictionary `field_shapes`
for each `column, weight` in `weights` **do**
  if `column` exists in `data` **then**
    `value` ← `data[column]`
    Append `column` to `used_fields`
    if `value` is a string **then**
      `embedding` ← `embedding_model(value)`
      `weighted_embedding` ← `weight` × `embedding`
      Append `weighted_embedding` to `vector`
      `field_shapes[column]` ← `shape of weighted_embedding`
    else if `value` is a number (int or float) **then**
      `weighted_value` ← `weight` × `value`
      Append `[weighted_value]` to `vector`
      `field_shapes[column]` ← `(1,)` (numeric value shape)
    end if
  end if
end for
`concatenated_vector` ← Concatenate all components in `vector`
return `concatenated_vector`

making sure each one contributes proportionatelly.

This approach creates a robust item representation that captures the importance of both textual and numeric features, enabling effective content-based filtering.

Similarity Computation

To compute similar products for a given product, the input is first processed to generate a query embedding that can be used to search for similar items in the database. The input data represents the product for which similar products need to be retrieved. The process goes the following way:

1. **Data Cleaning:** The input product data is cleaned using a **dictionary cleaner**. Invalid or incomplete data entries are skipped during this step.
2. **Data Normalization:** The cleaned product data is normalized using a **dictionary normalizer**. Products with missing critical fields, such as `price`, are removed.
3. **Embedding Generation:** A single vector representation (embedding) of the normalized product data is generated using an **embedding generator**. This embedding represents the input product in the similarity search space.
4. **Similarity Search:** The generated embedding is passed to the vector database (ChromaDB) where its drivers allow to retrieve the top n similar products. ChromaDB ranks the results based on cosine similarity between the query embedding and stored embeddings.

The following graphic illustrates the similarity computation process with cosine similarity as the way to determine the proximity between embeddings. **Figure 2.36**

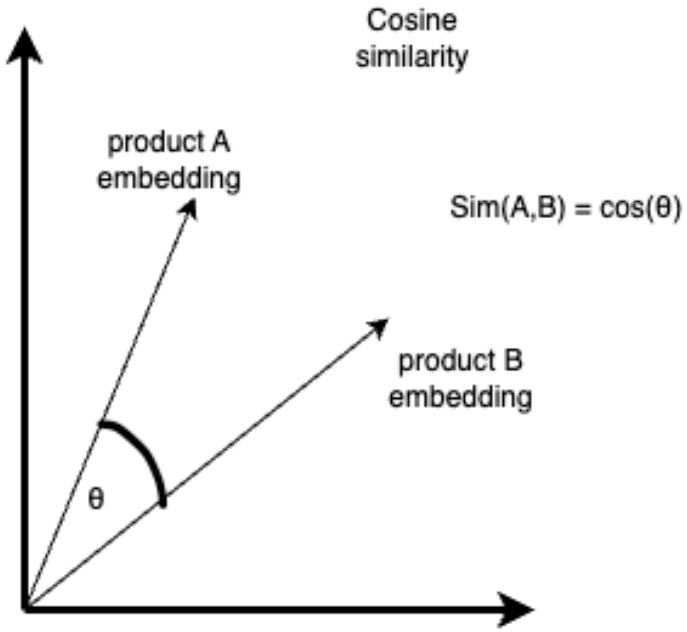


Figure 2.36: Similarity Computation Process Using Cosine Similarity

System Integration & API Contracts

The system provides a `GET` API endpoint, `/recc-system-1`, to retrieve products similar to a specified product. Clients provides a `product_id` and the desired number of similar products (`number_of_products`) as query parameters.

User-based Collaborative Filtering

User-based collaborative filtering recommends products to a user based on what other users with similar preferences have liked. The system uses the reviews collection from Amazon dataset [9], applies SVD (Singular Value Decomposition)[11] for dimensionality reduction, and stores latent factors in a MySQL database to provide recommendations for the user.

User-item matrix creation The foundation of user-based collaborative filtering is a user-item interaction matrix. This matrix captures the interactions between users and products, where each cell represents a users rating or interaction with a product. Sparse entries (missing ratings) are filled with 0. Example sparse matrix:**Figure 2.37**

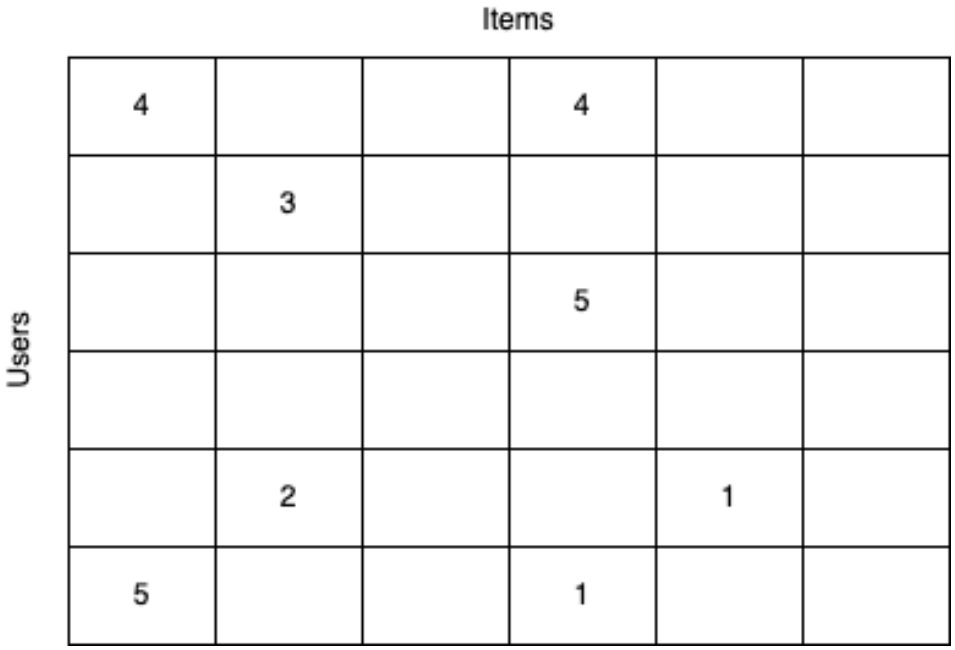


Figure 2.37: User-Item Interaction Matrix for Collaborative Filtering

The user-item matrix is constructed using Python's **scipy.sparse** library. The pseudocode implementation of this algorithm is shown in **Algorithm 3**

Algorithm 3 Create Sparse Matrix from Reviews

1: **Input:** `reviews` (a dataset with `user_id`, `asin`, and `rating`)
2: **Output:** `sparse_matrix` (a CSR matrix)
3: `user_codes` ← `reviews['user_id'].astype('category').cat.codes`
4: `item_codes` ← `reviews['asin'].astype('category').cat.codes`
5: `sparse_matrix` ← `csr_matrix((reviews['rating'], (user_codes, item_codes)))`

Dimensionality reduction To handle the high dimensionality and sparsity of the user-item matrix, SVD [11] is applied. This technique decomposes the matrix into three smaller matrices: user latent factors, item latent factors, and singular values. These latent factors represent abstract features of users and items (**n of them**), enabling similarity calculations. The concept of the reduction: **Figure 2.38**

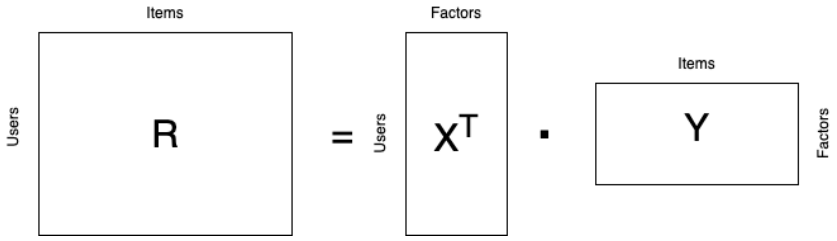


Figure 2.38: SVD Factorization of the User-Item Matrix

The SVD computation is implemented as follows, prepresented by a pseudocode in: **Algorithm 4**

Algorithm 4 SVD Factorization of User-Item Matrix

1: **Input:** `sparse_matrix` (a CSR matrix)
2: **Output:** `user_factors`, `item_factors`
3: Initialize SVD model: `svd` ← `TruncatedSVD(n_components=30, random_state=42)`
4: Compute user factors: `user_factors` ← `svd.fit_transform(sparse_matrix)`
5: Compute item factors: `item_factors` ← `svd.components_.T`

These latent factors are stored in the MySQL database for efficient retrieval during recommendation computation.

User similarity calculation Using the user latent factors derived from SVD, pairwise similarity between users is computed. Cosine similarity is used to measure the closeness of user preferences in the latent factor space. This similarity metric identifies users with shared tastes. The stored user factors can be queried to dynamically compute similarities when needed.

Neighborhood formation Based on the computed user similarities, a neighborhood of similar users is formed for each target user. The neighborhood consists of the top k most similar users. These users' preferences are then aggregated to recommend products that the target user has not interacted with.

Implementation Integration The recommendation system initialization process ties these components together:

- Reviews data is loaded from MongoDB.
- The user-item matrix is constructed, and SVD is applied to derive latent factors.

- The latent factors are stored in MySQL for efficient computation.

The initialization code ensures this workflow is triggered only when required, displayed as pseudocode **Algorithm 5** :

Algorithm 5 Initializing the Recommendation System

1: **Input:** MongoDB connection details, review collection name, number of products (`n`), SVD components (`n_components`)
2: **Output:** Saved latent factors in MySQL database
3: Connect to MongoDB and access the `config` collection
4: Retrieve configuration: `config_doc` ← `config_collection.find_one({"collection_name": "recc_system_2"})`
5: **if** `config_doc.should_run = True` **then**
6: Load reviews data: `reviews` ← `load_n_products(main_collection, n)`
7: Train SVD: `user_factors, item_factors` ← `train_svd(reviews, n_components)`
8: Save latent factors to MySQL: `save_latent_factors_to_db(engine, reviews, user_factors, item_factors)`
9: **end if**

System Integration & API Contracts

The system provides a `GET` API endpoint, `/recc-system-2`, to retrieve product recommendations based on user preferences using a collaborative filtering approach. Clients can provide a `user_id` as a query parameter or as a header (`userId`) and specify the desired number of product recommendations (`number_of_products`, defaulting to 10).

3. Conclusions

3.1. Achieved goals

In this project we achieved our goals of developing a distributed e-commerce web app enhanced with content-based and user-based collaborative filtering recommendation systems. The application provides an ecommerce platform features like user auth, product management, shopping cart and products recommendations.
The application is making the user capable of performing the most crucial task for which it was created, which is browsing through the products recommended by the AI model and passed to the frontend layer by the API provided by the backend part.
Moreover, the user can add and remove their products, which is also a great milestone achieved during building this project.
However, there were also things that were finally abandoned and not implemented in the application, including the user-chat microfrontend and user-chat services. Those two components were supposed to give the users the ability to chat with each other. Unfortunately, due to the technical complications in frontend-backend integration, we decided to cross this out of the project boundaries. Another application feature that was not implemented due to the lack of time was transforming the frontend layer into a Progressive Web Application, which would enable the application to work just like the mobile or desktop one.

3.2. Challenges and solutions

While implementing the project, we faced various interesting challenges. One of them was making the frontend container application, named Main, work with the Module Federation components. Several problems emerged in terms of testing, where the components provided by microfrontend could not be found as a result of the typescript compila-

tion failure. We managed to solve this issue by modifying Typescript compiler settings, *vite.config.js* file of the Main application, and by creating the *declarations.d.ts* file, which content is provided in the 24 appendix. Another problem, which emerged at the very beginning of the project, was to make the Main application capable of ingesting particular components from the same URL addresses of considered microfrontends. This challenge was solved with the use of fixing the ports of each microfrontends' previews. The list of those ports is available in the 5.5 appendix.

We also had to face concerning tasks while creating the backend layer. The microservices architecture challenged us with many issues, including the communication problems between the particular microservices. Fortunately, we found the solutions for each of the most crucial tasks. To provide the microservices with communication between each other, we used Apache Kafka with timeout mechanism to terminate the connection if no response came back in 30 seconds. For solving the issue of code duplication, we introduced shared DTO classes to standarize data exchange. In order to ensure the data consistency, we implemented the Request-Response and Event-Driven Notification Patterns. Finally, to standarize Apache Kafka configuration, we used the Common-kafka-config module.

As we progressed with the project implementation, we also had to find solutions for the recommendation system problems. To start with we had to solve the issue of data sparsity in collaborative filtering. Another challenge was to handle the recommendation data updates complexity, which we did by initializing the recommendation matrices once and preparing the baseline to update them in a scheduled way. Another concerning challenge was to find the optimal weights for different product features. Through experimentation, we decided to give the title and description more weight than the main category or price.

3.3. Potential improvements

There certainly is a place for some improvements of this project. One of the things that could be improved is to actually introduce the PWA standard in the frontend layer, which would improve the application accessibility. Another major improvement could be using some kind of a monorepo tool to manage the microfrontends, which would make the management process easier.

In terms of the backend layer, there could be a major improvement of the microservices communication. Instead of Apache Kafka, one could use Saga Pattern to make the imple-

mentation of more complex scenarios and handling the success/failure operations easier. Another potential improvement could be introduced relating to the deployment process. Currently, it relies on Docker, so the next logical step would be to implement Auto Scaling with the use of a container orchestration system, like Kubernetes.
When it comes to the recommendation system itself, we could potentially explore strategies to mitigate the cold start problem for new users and products, as this could leverage the user demographic information. We could have also leveraged Apache Kafka to achieve real-time updates of products embedding, which would increase the recommendations accuracy.

3.4. Final thoughts

Both recommendation systems successfully achieved its goals - providing users with AI driven similar products and by suggesting products of other user with similar taste.

The use of microfrontend and microservices architectures for the frontend and backend significantly improved the user experience making it more reliable and fault tolerant.

There is still a room for improvement but the core idea of the distributed ecommerce platform has been implemented and that is the most important for us from the learning standpoint. The project allowed us to understand the core foundations of building ecommerce apps in depth.

4. Bibliography

[1] React.JS documentation - https://react.dev/

[2] S.Bajaj, Mastering Jest Configuration for React TypeScript Projects with Vite: A Step-by-Step Guide, https://dev.to/bajajcodes/mastering-jest-configuration-for-react-typescript-projects-with-vite-a-step-by-step 2023

[3] Herr_Hansen, Answer to the question: "Set size of window in Jest and jest-dom and jsdom", https://stackoverflow.com/questions/60396600/set-size-of-window-in-jest-and-jest-dom-and-jsdom/60817030, 2020

[4] Testing React apps with Jest - https://jestjs.io/docs/tutorial-react

[5] A.V, React Micro-Frontends using Vite, https://dev.to/abhi0498/react-micro-frontends-using-vite-30ah, 2023 .

[6] Fluent UI Library documentation - https://react.fluentui.dev/?path=/docs/concepts-introduction--docs

[7] Fuse.js documentation - https://www.fusejs.io/

[8] Y. Hou, J. Li, Z. He, A. Yan, X. Chen, J. McAuley, *Bridging Language and Items for Retrieval and Recommendation*, arXiv preprint arXiv:2403.03952, 2024. Available at: https://arxiv.org/abs/2403.03952.

[9] Amazon scraped dataset https://amazon-reviews-2023.github.io/

[10] Kumar, A. (2021, December 17). Various implementations of collaborative filtering. Towards Data Science. https://towardsdatascience.com/various-implementations-of-collaborative-filtering-100385c6dfe0

[11] Wikipedia contributors. (2025, January 28). Singular value decomposition. Wikipedia. https://en.wikipedia.org/wiki/Singular_value_decomposition

[12] Wikipedia contributors. (2025, January 28). Cosine similarity. Wikipedia. https://en.wikipedia.org/wiki/Cosine_similarity

[13] Wikipedia contributors. (2025, January 28). Cosine similarity. Wikipedia. https://en.wikipedia.org/wiki/Cosine_similarity

[14] Saurav9786. (2023, April 12). Recommender system using Amazon reviews. Kaggle. https://www.kaggle.com/code/saurav9786/recommender-system-using-amazon-reviews/notebook#Popularity-Based-Recommendation

[15] Docs - Java Documentation - https://docs.oracle.com/en/java/

[16] Docs - Spring Boot Documentation - https://docs.spring.io/spring-boot/index.html

[17] Figure Event Driven - https://learn.microsoft.com/pl-pl/azure/architecture/guide/architecture-styles/event-driven

[18] Gateway diagram - https://medium.com/@okan.ardic/spring-cloud-microservices-part-2-integrating-api-gateway-80f95e2c25d9

[19] Request-Response diagram - https://blog.devgenius.io/event-patterns-request-reply-is-it-a-pattern-or-anti-pattern-641a257192d4

[20] Reimers, N., Gurevych, I. (2020). 'all-MiniLM-L6-v2' [Computer software]. Hugging Face. Available at: https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

[21] Collaborative filtering https://en.wikipedia.org/wiki/Collaborative_filtering

[22] Architecture Comparison https://dashbird.io/knowledge-base/well-architected/monolith-vs-microservices/

List of Figures

List of tables

5. List of appendices

1. Frontend Standard Environmental Variables Mocks

Table 5.1: Environmental Variables Mocks

Name	Value
VITE_PREVIEW_MODE	true
VITE_API_URL	http://localhost:3001
VITE_BLOCK_I18NEXT	true

2. Routing table

Table 5.2: Routing configuration

Route	Component name	Microfrontend
/ (default route)	SignInPanel	auth-module
/signin	SignInPanel	auth-module
/signup	SignUpPanel	auth-module
/products/:productId	Product	products-browsing
/products/search/:query	Tiles	products-browsing
/settings	UserSettings	user-settings
/basket	UserBasket	user-basket
/products/add	AddProduct	products-managing
/products/mine	ProductsList	products-managing
/order	Order	user-order
/order-history	OrderHistory	user-order
* (all of the other routings)	Page404	none (Main's component)

3. Products Browsing Microfrontend Environmental Variables Mocks

Table 5.3: Environmental Variables Mocks

Name	Value
VITE_PREVIEW_MODE	true
VITE_API_URL	http://localhost:3001
VITE_BLOCK_I18NEXT	true

4. Frontend Standard Microfrontends libraries

Table 5.4: Microfrontend Packages

Name	Description
React.JS	a Javascript library for building the application's UI
Axios	a library for handling the HTTP calls
Microsoft Fluent UI	a library providing UI components
Styled components	a library for styling the components with the use of tagged template literals
React router dom	a package which is essential for implementing the dynamic routing of the application
react-i18next	a package handling the usage of the i18next internationalization standard, including providing the translation function
i18next-http-backend	a package needed to make the user-settings microfrontend capable of accessing the translation files

5. Microfrontends preview ports

Table 5.5: Environmental Variables Mocks

Microfrontend	Port
auth-module	4173
products-browsing	4176
products-managing	4178
user-settings	4175
user-basket	4174
user-order	4177

6. Main application configuration

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'
import federation from "@originjs/vite-plugin-federation"

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [
    react(),
    federation({
      name: 'engineering-project',
      remotes: {
        authComponents:
        'http://localhost:4173/assets/remoteAuthEntry.js',
        userSettings:
```

```
            'http://localhost:4174/assets/remoteUserSettingsEntry.js',
            userBasket:
            'http://localhost:4175/assets/remoteUserBasketEntry.js',
            productsBrowsing:
            'http://localhost:4176/assets/remoteProductsBrowsingEntry.js',
            userOrder:
            'http://localhost:4177/assets/remoteUserOrderEntry.js',
            productsManaging:
            'http://localhost:4178/assets/assets/remoteProductsManagingEntry.
          },
          shared: ['react', 'react-dom', 'react-router-dom',
          '@fluentui/react-components', 'react-i18next']
        })
      ],
    })
```

7. Main application API used

   (a) **basket** - a GET method for extracting the information regarding the currently logged in user's basket

   (b) **product/search** - a GET method used to fetch all of the products matching the search criteria

8. Microfrontend configuration

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'
import federation from '@originjs/vite-plugin-federation';

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [
    react(),
    federation({
      name: '<microfrontend_name>',
```

```
            filename: '<exposed_file_name>.js',
            exposes: {
              './<exposed_component_name>':
              '<exposed_component_microfrontend_path>.tsx',
            },
            shared: <array_of_shared_dependencies>
          })
        ],
        build: {
          modulePreload: false,
          target: "esnext",
          minify: false,
          cssCodeSplit: false,
        },
        preview: {
          port: <microfrontend_port_name>
        }
      })
```

, where:

- $<$ *microfrontend_name* $>$ - the name of the particular microfrontend
- $<$ *exposed_file_name* $>$ - the name of the file which will be visible for the container application
- $<$ *exposed_component_name* $>$, $<$ *exposed_component_microfrontend_path* $>$ - the name of the component that will be used in the container path and the local path to the file containing the exposed component
- $<$ *array_of_shared_dependencies* $>$ - an array of a dependencies that the microfrontend and the container app share, given in order to improve the dependencies loading efficiency
- $<$ *microfrontend_port_name* $>$ - a port that will be used by the *npm run preview* command, that will run up a server providing the components exposed for the container application

9. Auth module API used

(a) **auth/login** - a POST method for logging the user in

(b) **auth/register** - a POST method for registering the user

10. Auth module test results

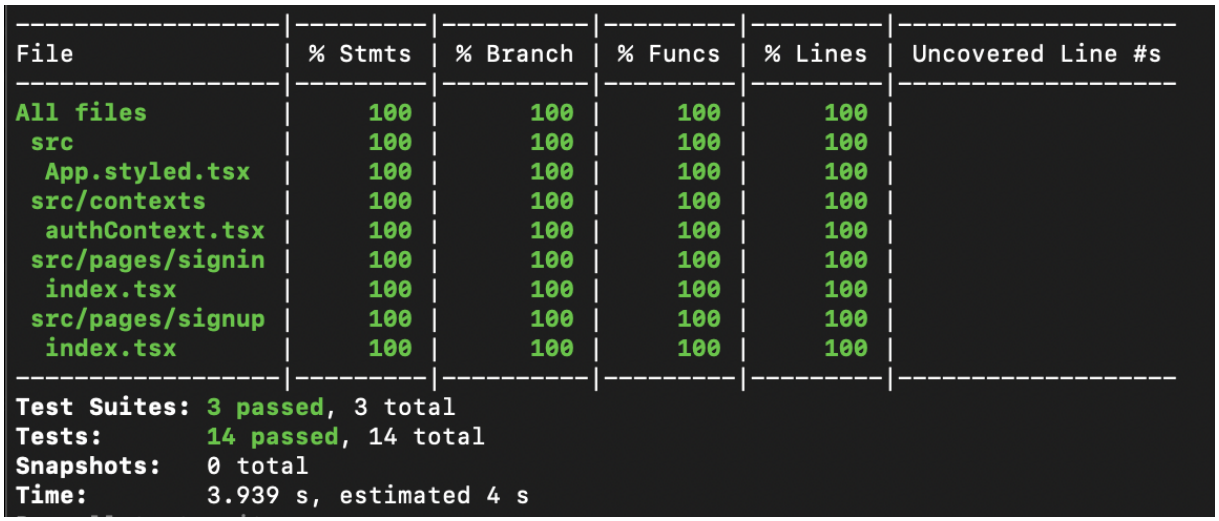


Figure 5.1: Auth Module testing results

11. Products browsing API used

(a) The main API:

i. **product/review/:productId/reviews** - a GET method for retrieving all of the product identified by the passed product ID reviews

ii. **product/review/:productId** - a POST method for posting a new review of the product identified with the passed product ID

iii. **basket** - a POST method for adding the selected product to the user's basket

iv. **like/:productId** - a POST method for adding a like to the displayed product

v. **like/remove/:productId** - a POST method for removing the like given by the user to the displayed product

vi. **product/:productId** - a GET method for retrieving all of the information regarding the product identified by the product ID

vii. **product/search** - a GET method to find all of the products' matching the criteria

(b) **"Similar products" API:**

i. **recc-system-1?product_id=**: *productId*&**number_of_products=4** - a GET method for retrieving the products similar to the currently browsed one

(c) **"Other users browsed" API:**

i. **recc-system-2?user_id=**: *productId*&**number_of_products=4** - a GET method for retrieving the products the other users also browsed

12. Products Browsing testing results

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	97.81	87.01	93.67	98.2	
components/product/DetailsAndFeatures	100	100	100	100	
DetailsAndFeatures.styled.tsx	100	100	100	100	
DetailsAndFeatures.tsx	100	100	100	100	
components/product/ImagesCarousel	69.23	0	0	66.66	
ImagesCarousel.styled.tsx	100	100	100	100	
ImagesCarousel.tsx	55.55	0	0	50	16,35-55
components/product/ProductPresentation	100	100	100	100	
ProductPresentation.styled.tsx	100	100	100	100	
ProductPresentation.tsx	100	100	100	100	
components/product/Recommendations	100	58.33	100	100	
Recommendations.styled.tsx	100	100	100	100	
Recommendations.tsx	100	58.33	100	100	41-75,80
components/product/ReviewDisplay	95.77	92.3	100	97.01	
ReviewDisplay.styled.tsx	100	100	100	100	
ReviewDisplay.tsx	95.65	92.3	100	96.92	50,140
components/product/ReviewDisplay/components/ParticularReviewDisplay	100	100	100	100	
ParticularReviewDisplay.styled.tsx	100	100	100	100	
ParticularReviewDisplay.tsx	100	100	100	100	
components/product/ReviewDisplay/components/ReviewEditDialog	92.3	100	80	100	
ReviewEditDialog.styled.tsx	100	100	100	100	
ReviewEditDialog.tsx	90.9	100	80	100	
components/product/ReviewDisplay/components/ReviewPagination	100	100	100	100	
ReviewPagination.styled.tsx	100	100	100	100	
ReviewPagination.tsx	100	100	100	100	
components/product/ReviewForm	97.43	100	100	97.36	
ReviewForm.styled.tsx	100	100	100	100	
ReviewForm.tsx	96.42	100	100	96.29	67
components/tiles/sidebar	100	84.61	100	100	
Sidebar.styled.tsx	100	50	100	100	13-20
Sidebar.tsx	100	100	100	100	
components/tiles/tile	100	100	100	100	
Tile.styled.tsx	100	100	100	100	
Tile.tsx	100	100	100	100	
pages/product	100	100	100	100	
Product.styled.tsx	100	100	100	100	
Product.tsx	100	100	100	100	
pages/tiles	100	100	100	100	
Tiles.styled.tsx	100	100	100	100	
Tiles.tsx	100	100	100	100	

```
Test Suites: 12 passed, 12 total
Tests:       43 passed, 43 total
Snapshots:   0 total
Time:        14.372 s
```

Figure 5.2: Products Browsing testing results

The *ImagesCarousel.tsx* file was not tested due to the testing problems with the Microsoft Fluent UI Carousel component.

13. Products managing API used

(a) **product** - a POST method for creating a new product

(b) **product/:productId** - a GET method for extracting the data of the product of the specific product ID

(c) **product/:productId** - a DELETE method for removing, alias deactivating, the particular product

(d) **product/my-products** - a GET method for extracting all of the products created by the currently logged in user

(e) **product/:productId/image** - a POST method for uploading an image for the product of the specific product ID

(f) **product/:productId/image** - a DELETE method for removing a specified image from the product identified by the productId product ID

14. Products Managing testing results

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	97.15	81.66	95.32	97	
addProduct	98.36	71.42	92.85	98.36	
AddProduct.styled.tsx	100	100	100	100	
AddProduct.tsx	98.03	71.42	92.85	98.03	94
addProduct/categories	100	100	100	100	
Categories.styled.tsx	100	100	100	100	
Categories.tsx	100	100	100	100	
addProduct/details	100	100	100	100	
Details.styled.tsx	100	100	100	100	
Details.tsx	100	100	100	100	
addProduct/features	100	100	100	100	
Features.styled.tsx	100	100	100	100	
Features.tsx	100	100	100	100	
addProduct/finalization	100	100	100	100	
Finalization.styled.tsx	100	100	100	100	
Finalization.tsx	100	100	100	100	
productImageManagement	90.47	66.66	60	90.24	
ProductImageManagement.styled.tsx	100	100	100	100	
ProductImageManagement.tsx	89.18	66.66	60	88.88	63-65,104
productImageManagement/AddProductImage	100	100	100	100	
AddProductImage.styled.tsx	100	100	100	100	
AddProductImage.tsx	100	100	100	100	
productImageManagement/ImagesManagement	74.07	100	50	73.07	
ImagesManagement.styled.tsx	100	100	100	100	
ImagesManagement.tsx	66.66	100	50	65	41-60,83
productsList	100	75	100	100	
ProductsList.styled.tsx	100	100	100	100	
ProductsList.tsx	100	75	100	100	21-37
productsList/productsDisplay	100	78.57	100	100	
ProductsDisplay.styled.tsx	100	100	100	100	
ProductsDisplay.tsx	100	78.57	100	100	70-110
productsList/productsPager	100	100	100	100	
ProductsPager.styled.tsx	100	100	100	100	
ProductsPager.tsx	100	100	100	100	

Test Suites: 10 passed, 10 total
Tests: 36 passed, 36 total
Snapshots: 0 total
Time: 7.887 s

Figure 5.3: Products Managing testing results

15. User Settings API used

    (a) **auth/user/details/address** - a PATCH method for changing the user's addresses

    (b) **auth/user/details/personal-data** - a PATCH method for changing the user's personal data, including name, surname and phone number

    (c) **auth/user/details** - a GET method to extract the user's data

16. User Settings testing results

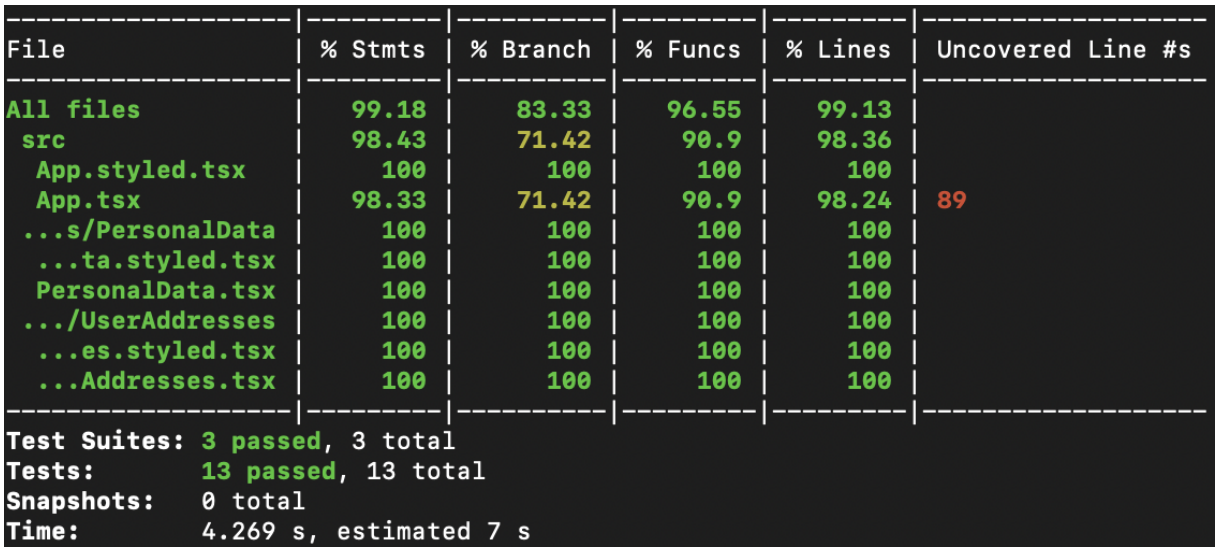


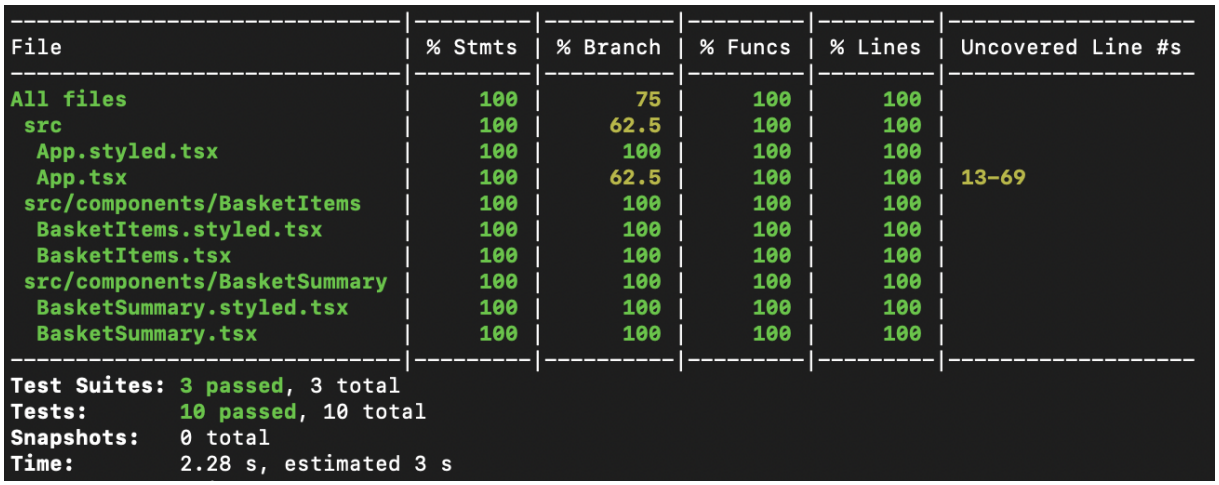
Figure 5.4: User Settings testing results

17. User Basket API used

    (a) **basket** - a GET method for providing all of the items added to the user's basket

    (b) **basket** - a DELETE method for removing an item from the user's basket

18. User Basket testing results



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	75	100	100	
src	100	62.5	100	100	
App.styled.tsx	100	100	100	100	
App.tsx	100	62.5	100	100	13-69
src/components/BasketItems	100	100	100	100	
BasketItems.styled.tsx	100	100	100	100	
BasketItems.tsx	100	100	100	100	
src/components/BasketSummary	100	100	100	100	
BasketSummary.styled.tsx	100	100	100	100	
BasketSummary.tsx	100	100	100	100	

Figure 5.5: User Basket testing results

19. User Order API used

    (a) **basket** - a GET method for providing all of the data of the user's basket

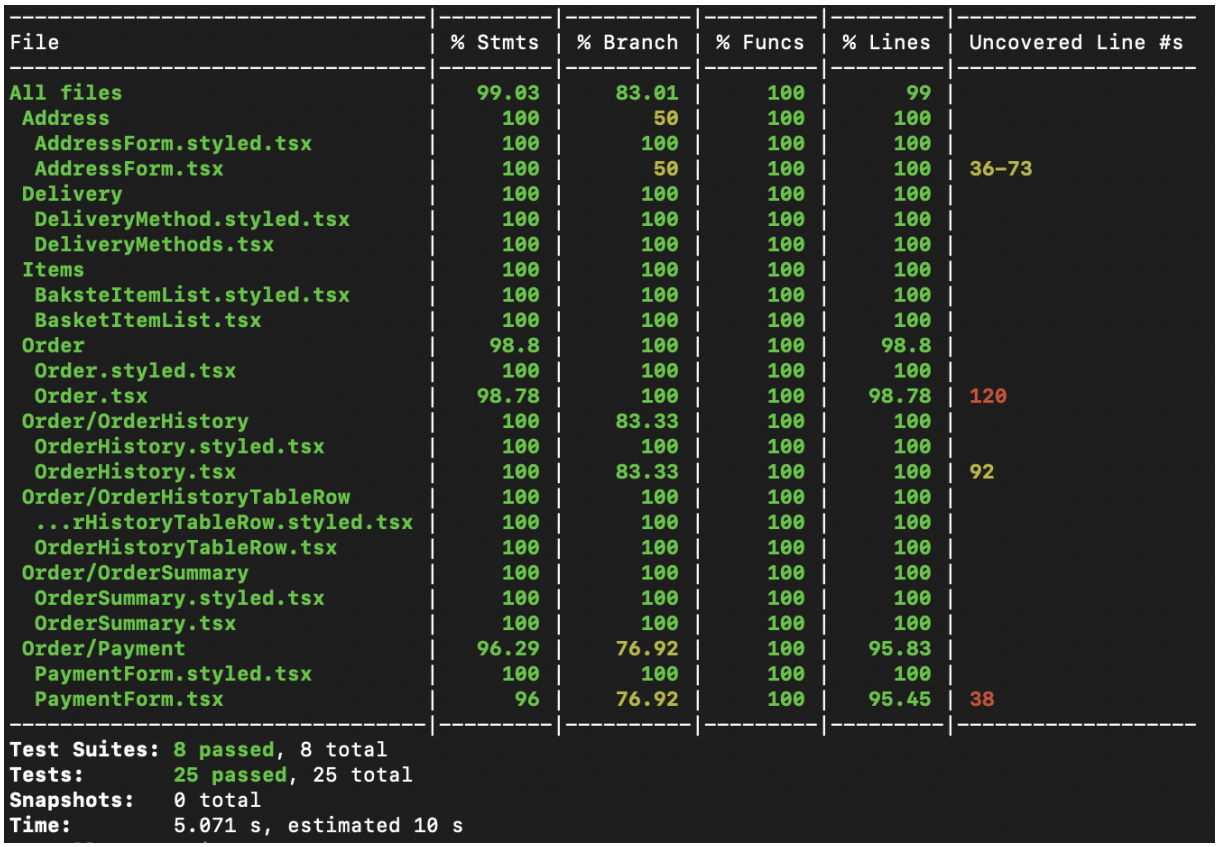
    (b) **order** - a POST method for submitting a new order

    (c) **order/notify** - a POST method for notifying the backend about the user's order

    (d) **order/deliver** - a GET method for extracting the delivery methods available

    (e) **order** - a GET method for loading the history of orders

20. User Order testing results



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	99.03	83.01	100	99	
Address	100	50	100	100	
AddressForm.styled.tsx	100	100	100	100	
AddressForm.tsx	100	50	100	100	36-73
Delivery	100	100	100	100	
DeliveryMethod.styled.tsx	100	100	100	100	
DeliveryMethods.tsx	100	100	100	100	
Items	100	100	100	100	
BaksteItemList.styled.tsx	100	100	100	100	
BasketItemList.tsx	100	100	100	100	
Order	98.8	100	100	98.8	
Order.styled.tsx	100	100	100	100	
Order.tsx	98.78	100	100	98.78	120
Order/OrderHistory	100	83.33	100	100	
OrderHistory.styled.tsx	100	100	100	100	
OrderHistory.tsx	100	83.33	100	100	92
Order/OrderHistoryTableRow	100	100	100	100	
...rHistoryTableRow.styled.tsx	100	100	100	100	
OrderHistoryTableRow.tsx	100	100	100	100	
Order/OrderSummary	100	100	100	100	
OrderSummary.styled.tsx	100	100	100	100	
OrderSummary.tsx	100	100	100	100	
Order/Payment	96.29	76.92	100	95.83	
PaymentForm.styled.tsx	100	100	100	100	
PaymentForm.tsx	96	76.92	100	95.45	38

Figure 5.6: User Order testing results

21. Administrator app packages

Table 5.6: Administrator packages

Name	Description
React.JS	a Javascript library for building the application's UI
Axios	a library for handling the HTTP calls
Microsoft Fluent UI	a library providing UI components
Styled components	a library for styling the components with the use of tagged template literals
React router dom	a package which is essential for implementing the dynamic routing of the application

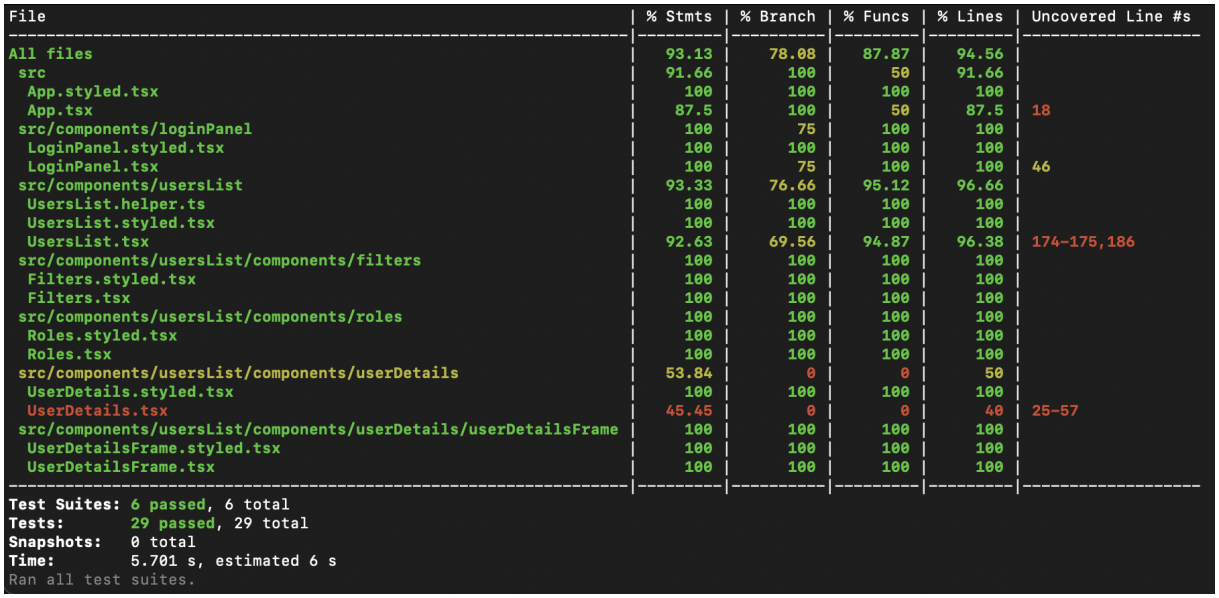
22. Administrator application API used

(a) **auth/admin/all-users** - a GET method to get the list of all users registered in the application

(b) **auth/admin/delete-user/:userId** - a DELETE method for deleting the account of the user identified with the userId ID

(c) **auth/login** - a POST method for logging the admin into the system

23. Administrator testing results



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	93.13	78.08	87.87	94.56	
src	91.66	100	50	91.66	
App.styled.tsx	100	100	100	100	
App.tsx	87.5	100	50	87.5	18
src/components/loginPanel	100	75	100	100	
LoginPanel.styled.tsx	100	100	100	100	
LoginPanel.tsx	100	75	100	100	46
src/components/usersList	93.33	76.66	95.12	96.66	
UsersList.helper.ts	100	100	100	100	
UsersList.styled.tsx	100	100	100	100	
UsersList.tsx	92.63	69.56	94.87	96.38	174-175,186
src/components/usersList/components/filters	100	100	100	100	
Filters.styled.tsx	100	100	100	100	
Filters.tsx	100	100	100	100	
src/components/usersList/components/roles	100	100	100	100	
Roles.styled.tsx	100	100	100	100	
Roles.tsx	100	100	100	100	
src/components/usersList/components/userDetails	53.84	0	0	50	
UserDetails.styled.tsx	100	100	100	100	
UserDetails.tsx	45.45	0	0	40	25-57
src/components/usersList/components/userDetails/userDetailsFrame	100	100	100	100	
UserDetailsFrame.styled.tsx	100	100	100	100	
UserDetailsFrame.tsx	100	100	100	100	

Test Suites: 6 passed, 6 total
Tests: 29 passed, 29 total
Snapshots: 0 total
Time: 5.701 s, estimated 6 s
Ran all test suites.

Figure 5.7: Administrator testing results

The *UserDetails.tsx* file was not tested due to the testing problems with the Microsoft Fluent UI Carousel component.

24. declarations.d.ts file content

```
declare module 'authComponents/AuthProvider' {
    export const useAuth: () => {
        token: string,
        logout: () => void
    };
    export const AuthProvider;
}
declare module 'authComponents/SignIn'
declare module 'authComponents/SignUp'
declare module 'userSettings/UserSettings'
declare module 'userBasket/UserBasket'
declare module 'productsBrowsing/Tiles'
declare module 'productsBrowsing/Product'
declare module 'productsManaging/AddProduct'
declare module 'productsManaging/ProductsList'
```

```
declare module 'userOrder/UserOrder'
declare module 'userOrder/OrderHistory'
```

25. Backend Endpoints

    (a) Auth Service **Implementation of AuthController Methods**

    The `AuthController` and its corresponding service implementation (`UserServiceImpl`) manage authentication, authorization, and user account operations. Below is a list of all endpoints.

    - **register user**: `/api/v1/auth/register` - POST
    - **login**: `/api/v1/auth/login` - POST
    - **validate token**: `/api/v1/auth/validate` - GET
    - **authorize**: `/api/v1/auth/authorize` - GET
    - **change password**: `/api/v1/auth/change-password` - POST
    - **delete my account**: `/api/v1/auth/delete-my-account` - DELETE
    - **verify**: `/api/v1/auth/verify` - DELETE

    Implementation of UserController Methods

    The `UserController` and `UserDetailsServiceImpl` provide functionalities related to user personal data, addresses, and image management. Below is a list of all endpoints.

    - **fill user personal data**: `/api/v1/auth/user/details/personal-data`-PATCH
    - **update addresses**: `/api/v1/auth/user/details/address`-PATCH
    - **get User Details**: `/api/v1/auth/user/details`-GET
    - **get User Details By Uuid**: `/api/v1/auth/user/details/{userId}`-GET
    - **upload Image**: `/api/v1/auth/user/details/image`-POST

    Implementation of AdminController Methods

    The `AdminController` and its corresponding service implementation (`AdminServiceImpl`) manage administrative operations, such as retrieving user details, managing roles, and deleting user accounts. Below is a list of all endpoints.

- **get all user Details**: `/api/v1/auth/admin/user-details/{userId}`
- **get all Users**: `/api/v1/auth/admin/all-users`-GET
- **change Role**: `/api/v1/auth/admin/change-role/{userId}`-PATCH
- **deleteUser**: `/api/v1/auth/admin/delete-user/{userId}`-DELETE

(b) Product and Review Service

Implementation of ProductController Methods

The `ProductController` and its corresponding service implementation (`ProductServiceImpl`) manage product-related operations such as creating, updating, deleting, and retrieving products. Below is a list of all endpoints.

- **get products**: `/api/v1/product/search`-GET
- **get product**: `/api/v1/product/{productId}`-GET
- **get my products**: `/api/v1/product/my-products`-GET
- **create product**: `/api/v1/product`-CREATE
- **add image to product**: `/api/v1/product/{productId}/image`-POST
- **update image**: `/api/v1/product/{productId}/image`-PATCH
- **delete image**: `/api/v1/product/{productId}/image`-DELETE

Implementation of ReviewController Methods

The `ReviewController` and its corresponding service implementation (`ReviewServiceImpl`) manage review-related operations such as creating, updating, deleting, and retrieving reviews. Below is a list of all endpoints.

- **get reviews**: `/api/v1/product/review/{productId}/reviews`-GET
- **get review**: `/api/v1/product/review/{reviewId}`-GET
- **create review**: `/api/v1/product/review/{productId}`-POST
- **update review**: `/api/v1/product/review/{reviewId}`-PATCH
- **delete review**: `/api/v1/product/review/{reviewId}`-DELETE

(c) Like Service

Implementation of LikeController Methods

The `LikeController` and its corresponding service implementation (`LikeServiceImpl`) handle operations related to product likes, such as adding, removing, and querying likes for products. Below is a list of all endpoints.

- **add like**: `/api/v1/like/{productId}`-POST
- **get my like**: `/api/v1/like/my`-GET
- **get number of likes**: `/api/v1/like/number/{productId}`-GET
- **remove like**: `/api/v1/like/remove/{likeId}`-REMOVE
- **is liked**: `/api/v1/like/isLiked/{productId}`-GET

(d) Basket Serivice

Implementation of BasketController Methods

The `BasketController` and its corresponding service implementation (`BasketServiceImpl`) handle operations related to user baskets, such as adding products, removing products, and retrieving basket items. Below is a list of all endpoints.

- **add product to basket**: `/api/v1/basket`-POST
- **delete product from basket**: `/api/v1/basket`-DELETE
- **get items**: `/api/v1/basket`-GET

(e) Order Serivice

Implementation of OrderController Methods

The `OrderController` and its corresponding service implementation manage operations related to orders, including creation, retrieval, and status updates. Below is a list of all endpoints.

- **create order**: `/api/v1/order`-POST
- **get order by id**: `/api/v1/order/{orderId}`-GET
- **get orders by client**: `/api/v1/order`-GET
- **notify**: `/api/v1/order/notify`-POST

Implementation of DeliverController Methods The `DeliverController` and its corresponding service implementation manage delivery options for orders. Below is a list of all endpoints.

  - **get deliver**: `/api/v1/order/deliver` - GET
  - **create deliver order**: `/api/v1/order/deliver` - POST