Table of Contents

Introduction

In the modern digital world, web applications have turned out to be essential to the functioning of businesses in all spheres of economic activity. Various means of e-commerce, among which web stores are the most important, are the ones that provide the platforms for businesses to reach the wider public and, at the same time, simplify the process of looking for a specific product or service, checking the details and reviews, and making a purchase, all of it from the comfort of their home. To make such functionalities, and many more, constantly available, responsive, scalable, and maintainable, the application architecture, which is the basis of any application, plays an important role in providing the solutions and methodologies to achieve such key features and performance indicators.

This paper deals with the development of a web application for e-commerce using the microservice-based strategy and the client-server architecture to improve the efficiency and flexibility of the web store. Microservices divide the applications into small, loosely connected services, which in turn, makes the independent development, deployment, and scaling of each service possible. Such a modular approach is advantageous for web stores, which usually need the integration of various functions like inventory management, user authentication, payment processing, and order fulfilment. Each functionality can be developed and managed as a separate microservice, and therefore, the whole application will become more resilient and adaptable.

The client-server model is the companion of the microservices architecture, which defines the roles of the client and the server. Here, the client is the one who takes care of the user interface and user interactions, whereas the server oversees the data processing, storage, and business logic. The application can deal with user requests and give a responsive, scalable service to the users by including the microservices in the framework.

This thesis will investigate the design and implementation of a microservice web application for a web store, detailing the processes and technologies used.

By the time this thesis is over, readers will be able to know how microservices are used to construct web applications. The findings and methodologies introduced in this issue will be of great importance to the field of software engineering, giving a valuable reference to the developers and researchers working on the architectures of web applications.

1. Topic evaluation

Web store applications, also known as e-commerce platforms, are digital storefronts used to browse and purchase products online. They encompass features and functionalities like product listings, shopping carts, checkouts, secure payment gateways, user accounts, and many more. To develop a web shop, a combination of technical skills for development, functionalities, and understanding of marketing and customer behaviour are required.

1.1. Related work

Before diving into the functionality defining and the development of the fragrance e-commerce application, existing web applications within the fragrance industry should be examined. The analysis, focused on the strengths and weaknesses of such applications in terms of user experience and functionality, will give a good founding path for the development of this e-commerce web application.

The application examined is belodore.hr. It is a Croatian web shop specializing in niche fragrances and cosmetics. Their home page is shown on the figure (Figure 1.1).

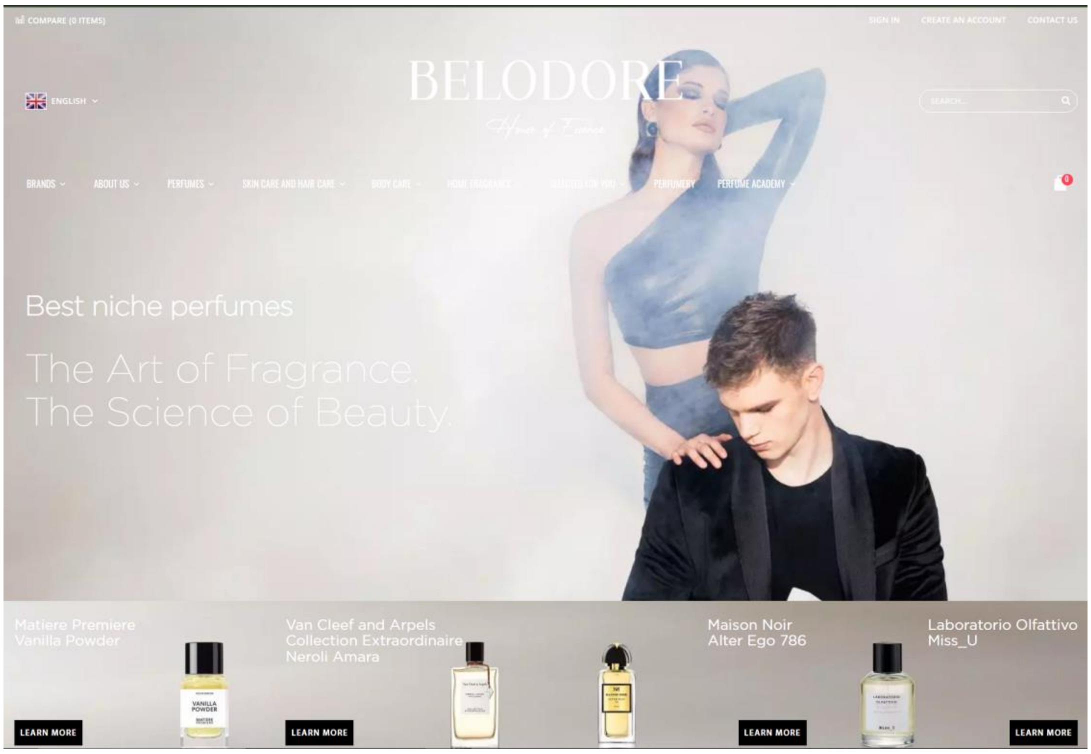


Figure 1.1 Belodore.hr home page

It is seeable from the very start, from the home page, that the web application offers a lot of options and functionalities. Firstly, it displays currently trending or just arrived products, enabling the visiting users the ability to come in touch with such products, and possibility to explore them further and purchase.

The header of this e-commerce web application provides a toolbar with many features, such as language selection, web shop name and logo, the main search bar for basic products search, cart, authentication and contact options, and most importantly, the main application navigation bar. The navigation consists of many options, from product selection options, like perfumes, skin care, body care, etc., to the brands selections, the about us section and more.

Regarding the products selection, based on the option and filter selected, a product listing page is opened with an applied filter, such as brand and category. An example of fragrance listing, with a man fragrance filter criteria, can be seen in figure (Figure 1.2).

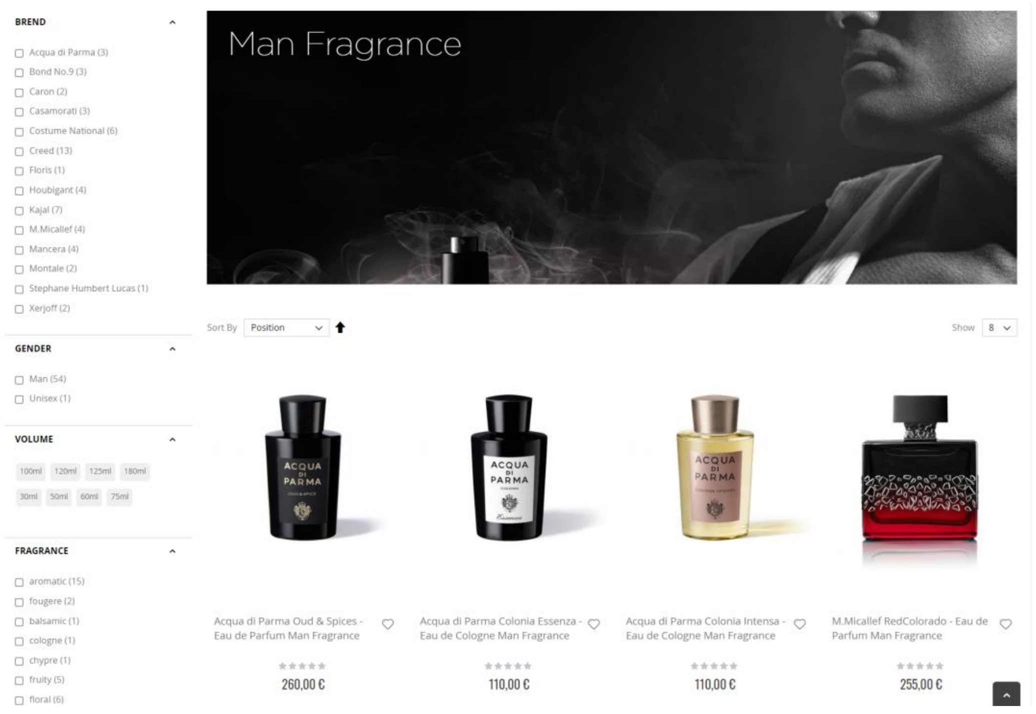
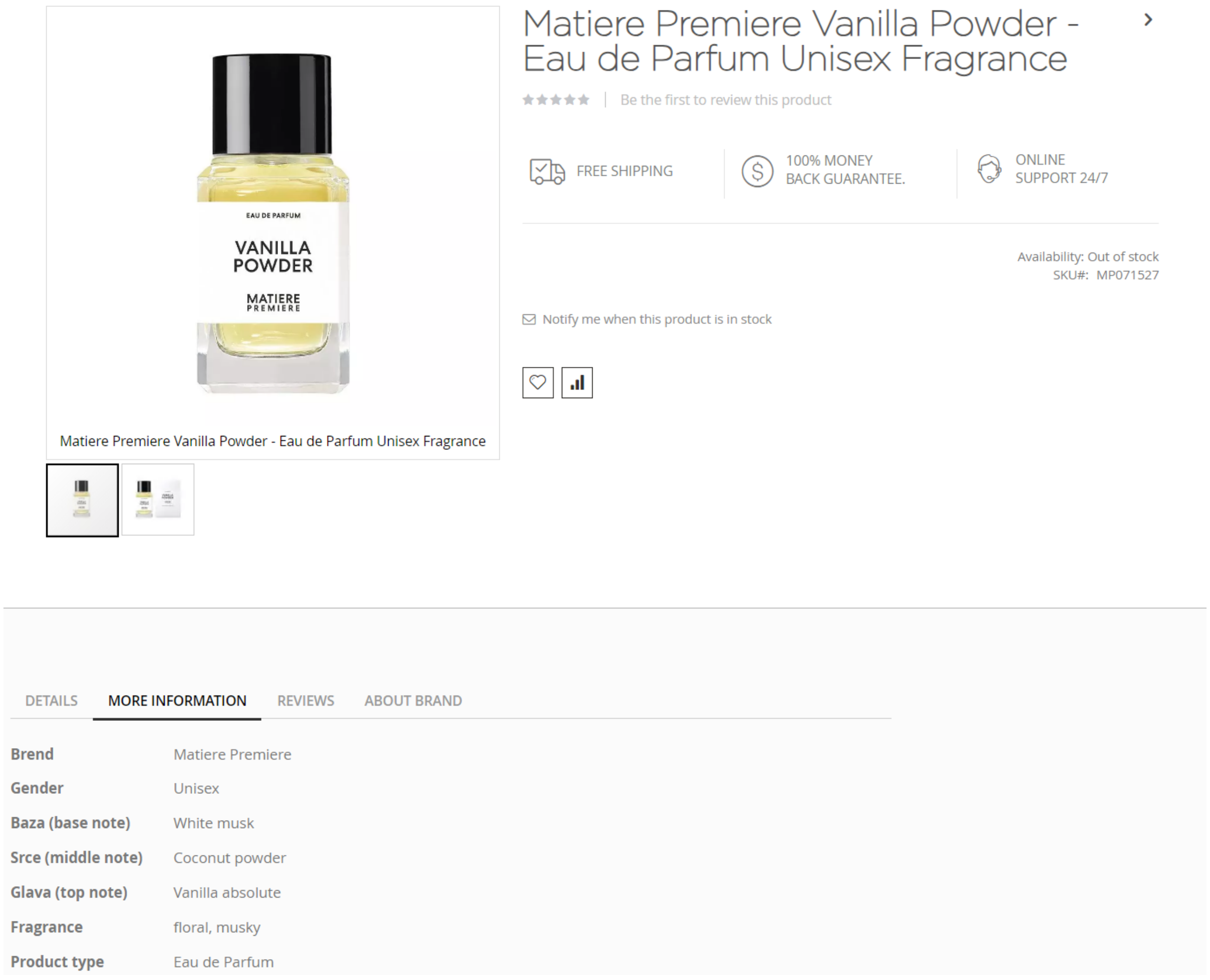


Figure 1.2 Belodore.hr products listing page

It is visible that each product listed contains some vital basic information that should be shown to the user, such as the product name, main product image, reviews rating, and the price. Additionally, the wishlist adding option is also available. Overall, the products listing should contain some form of filtering, and in this case, there are filter options like brand, gender, type, volume, fragrance notes, and price. Sorting is also available, along with pagination, in order to not load all the products at once, which puts a big strain on the performance aspect. By selecting the listed product, its belonging detail page is opened. The fragrance details consist of already known attributes from the listing page, such as title, reviews rating, main image and the price. Along with them, additionally it contains description, reviews and comments, specification including the fragrance notes, category, type and brand, additional images and the availability status. Some of that information can be seen on figure (Figure 1.3).


Matiere Premiere Vanilla Powder - Eau de Parfum Unisex Fragrance

Be the first to review this product

FREE SHIPPING | 100% MONEY BACK GUARANTEE. | ONLINE SUPPORT 24/7

Availability: Out of stock
SKU#: MP071527

Notify me when this product is in stock

Matiere Premiere Vanilla Powder - Eau de Parfum Unisex Fragrance

DETAILS | MORE INFORMATION | REVIEWS | ABOUT BRAND

Brend	Matiere Premiere
Gender	Unisex
Baza (base note)	White musk
Srce (middle note)	Coconut powder
Glava (top note)	Vanilla absolute
Fragrance	floral, musky
Product type	Eau de Parfum

Figure 1.3 Belodore.hr product details page

Alongside the listed features, this e-commerce website contains the classic cart and checkout features. The checkout is implemented as a two-step stepper, the first being the personal and shipping information form, and the second containing order summary, review and payment options.

1.2. Defining solution

Based on the previously analysed e-commerce platforms, an initial sketch of this paper's web application requirements can be made. In terms of functionality, this web shop should implement some of the key features that each e-commerce platform should contain, alongside some domain specific features, in this case, tied to the cosmetics and fragrances category.

The key functionalities of Fragrance.hr web shop:

- Products listing
- Products filtering, sorting, pagination
- Product details
- Product reviews
- Cart
- Checkout
- Payment
- Orders history
- AI chat assistant
- Authentication and authorization
- Administrator specific functionalities: creating updating, deleting products, managing users

1.2.1. Requirements specification

The requirements specification plays a founding role in the software development process, around which the entire process is built. It provides an accurate, detailed, and structured representation of all the necessary conditions applicable to the design and implementation, which need to be satisfied. [14] In most cases, the software requirements are divided into the two opposing categories, the functional and non-functional system requirements. The functional requirements are more common ones, and they describe what a software system must do ("system shall do"), outlining its features and functionalities, that are directly served to the initial user, or more precisely, to the actor that interacts with the system. On the other hand, non-functional requirements specify how the system should perform ("system shall be"), addressing aspects like security, performance, usability, and many more indicators.

The listed functionalities from the previous section belong to the functional requirements, as they describe the use cases that the application should offer to the actor. The actor represents any entity that is interacting with the system and its functionalities.

Initiating type of actor is the one that has direct interaction with the system, and in this web application, regular user and administrator are the initiating actors. Each one of them has specific or shared functional requirements that are initiated by them. The other type of actors, the participant actor, is the one that is passively affected by the system but does not directly interact with it. In this case, the application does not contain participant actors.

Functional requirements are defined and structured through the use cases. A use case is a structured description of a single functional requirement, focusing on what the system does from the user's perspective, rather than how it achieves those actions. These are the functionalities the system provides to actors to accomplish certain goals. Use cases can be documented in both text and visual formats. A use case diagram is a UML (Unified Modelling Language) diagram that visualizes the use cases. It acts as a simple blueprint for a system's functionality. [15] This application's use cases UML diagram, containing main functionalities, is shown in figure (Figure 1.4).

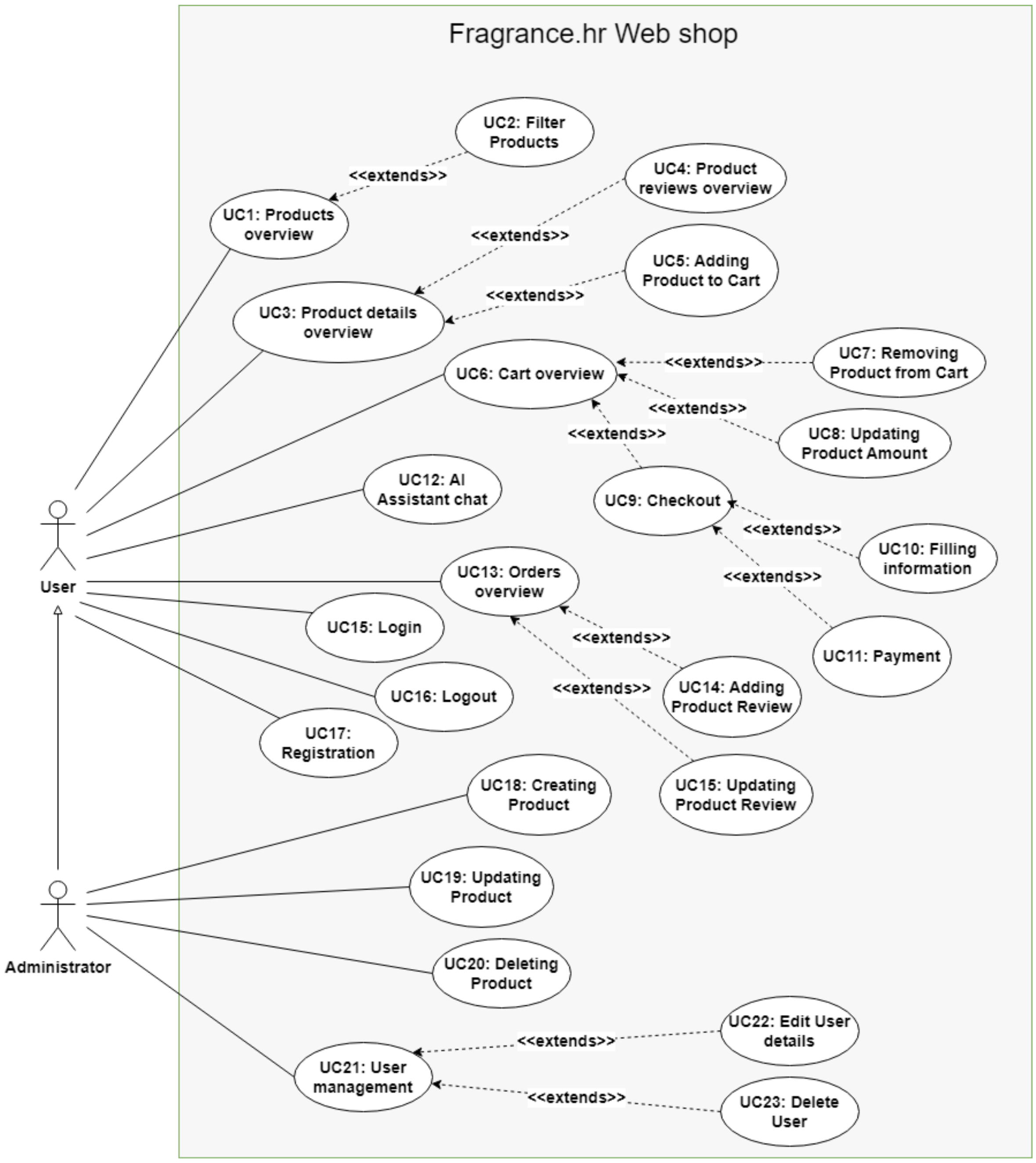


Figure 1.4 Web shop use case diagram

2. Technologies

Developing a web application requires a combination of many, powerful tools. This section explores the essential technologies, programming languages and frameworks, used on both the client-side (frontend) and server-side (backend) parts of the application to bring this web shop e-commerce application to life.

Java programming language is used for the development of the backend application. It is a general-purpose, object-oriented programming language, popular in the domain of web applications, enterprise software, and Android development.

For simplified building of Java project applications and managing its dependencies, this project uses Maven. It is a build automation tool used primarily for Java projects.

Spring Boot is a popular framework for building Java web applications. It acts as a pre-built toolbox, making it easy to create stand-alone Spring based applications by removing much of the boilerplate code and configurations associated with the web development. Spring is a blueprint for building a wide range of complex functionalities within Java applications, like database access, security, transactions, etc.

To implement microservices, Spring offers Spring Cloud, which is a suite of libraries that simplify building microservices-based applications.

Spring AI is a relatively new framework from Spring that provides tools for integrating AI capabilities into Spring applications. This framework will be used to implement the chatbot functionality with the OpenAI integration.

Docker is a platform for developing, deploying, and running applications in containers, which pack an application with all its dependencies and environment into a unit that can run independently and consistently on different environments. It will be used to containerize the PostgreSQL database and the Keycloak authorization server. Additionally, each microservice can be separated into a Docker container, making it independent and self-containing.

PostgreSQL is an open-source relation database management system (RDBMS). It is used for data management and storage in web applications, and it will be used as the database for the web shop application.

For handling and providing features to secure the application, like user authentication and authorization, single sign-on, the application will use Keycloak. It is an open-source identity and access management (IAM) solution, which provides features for handling authentication and authorization through the application.

To implement AI chatbot functionality, the application will integrate the OpenAI ChatGPT for the chatbot service. To further enhance and improve factual accuracy and contextual understanding of the ChatGPT, Retrieval-Augmented Generation will be used.

For payment integration, this web shop application will use PayPal, a popular online payment processing system that allows users to send and receive funds online.

On the client-side application, Angular framework is used for the development. It is a TypeScript (JavaScript superset) framework which provides a structured approach to building web applications with features like dependency injection, routing, and component-based development. It lies on the foundation technologies for building web pages, which are HTML, CSS, and JS. HTML defines the content and structure of a webpage, CSS styles its appearance, and JavaScript adds interactivity.

To empower the process of building and developing code for this web shop application, two suitable programming environments are used: Visual Studio Code (VS Code) and IntelliJ IDEA.

VS Code is a lightweight, open-source code editor from Microsoft. It is known for a support for a wide variety of programming languages, extensive customization, and clean interface. In this context, it is use for the development of the client-side application in Angular framework.

IntelliJ IDEA is an IDE (Integrated Development Environment) specifically designed for Java development by JetBrains. It offers robust code completion, refactoring tools, built-in debugging, and integration with various build systems and version control tools.

3. System architecture

When considering the matter of code structure, every software development project requires an architecture to serve as the backbone and provide basic guidelines and rules to enhance software quality and performance indicators. [1]

System architecture in the context of web applications means that it is the structured design and organization of the software components and their interactions to support the functionalities and performance of an application. The architecture typically follows a multi-tiered approach, ensuring separation of concerns. Its definition is complex and comprehensive, covering different layers and elements, such as the presentation layer, business logic layer, data access layer and the underlying infrastructure.

When discussing architecture for a server-side part of a web application, a fundamental consideration lies in choosing between many available options, each offering its own distinct advantages and trade-offs. This paper will cover microservices architecture as the one used, along with its opposing monolithic architecture.

3.1. Monolithic architecture

Monolithic architecture is a traditional first-choice software development approach to building an application, where an entire application is built as a single, unified unit. In this structure, all the components are tightly coupled and deployed together, where monolith holds all classes, functions, and namespaces for the entire application.

The primary benefit of monolithic architecture lies in its simplicity. Unlike distributed applications, monolithic architectures are significantly easier to test, deploy, debug, and monitor. All data remains centralized in a single database, eliminating the need for synchronization. [2] Additionally, it is characterized by ease of development, monitoring, and debugging, as developers work within a single codebase and have a clear understanding of the entire application’s functionality.

An example of a monolith server side, communicating with the client-side, is shown on diagram (Figure 3.1).

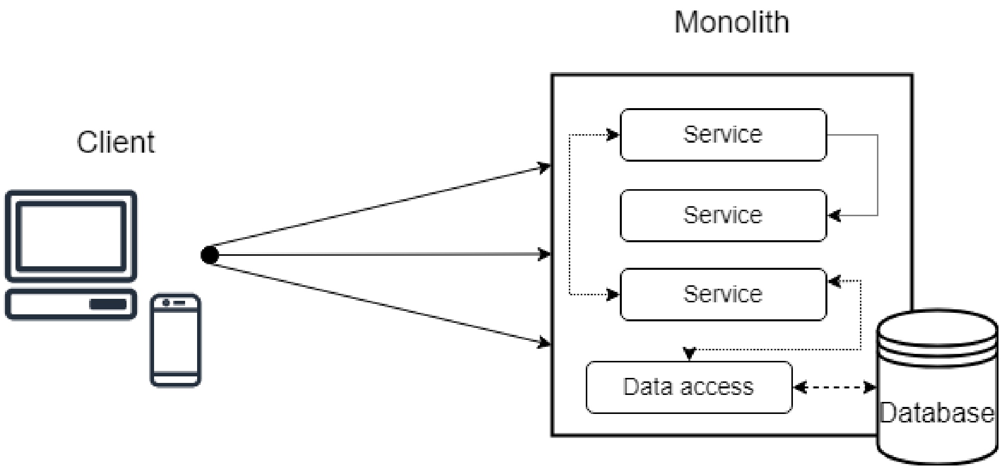


Figure 3.1 Monolithic backend example

The problem with a monolith architecture occurs when the application's complexity and size start increasing. Firstly, any changes to the application may require modifications to the entire monolith codebase. The larger size of the monolith causes longer start-up time and drop in performance, the more and more complex tightly coupled code may bring unexpected errors and the put a toll on the whole monolith system. Such scenarios can become a big obstacle for continuous development and productivity and can increase the overall project cost. [2]

Having in mind the mentioned advantages and disadvantages of a monolith architectural approach, it is safe to conclude that the monolithic architecture is a decent choice in scenarios where an application is smaller with basic functionalities and low complexity, and that would not require bigger future upgrades in the code. Also, when the product budget and deadline are very limiting, the monolithic architecture should be taken into consideration.

Since the goal of this paper is development of an e-commerce web shop application, based on the domain and functionalities estimations mentioned in the chapter above, the monolithic architecture would not serve as a good choice, since most of the functionalities are loosely coupled and do not need to be integrated in a single combined block. Also, all the user requests from the client to the server-side would be handled by the same receiving API. A diagram showing the web shop application components and client communication in a monolith context is shown in diagram (Figure 3.2).

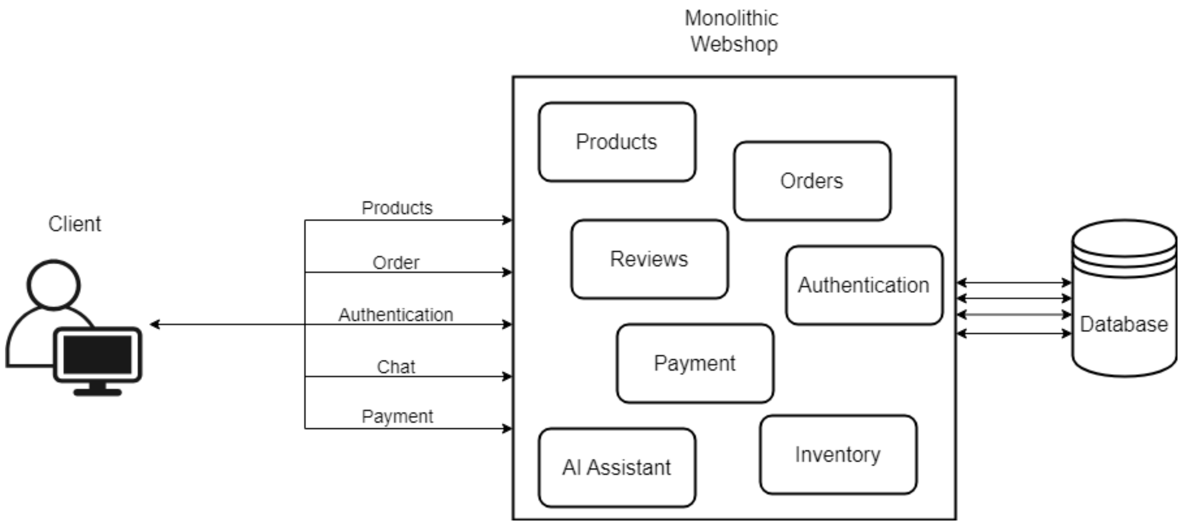


Figure 3.2 Web shop in monolithic architecture

Also, if one of the functionalities would go down, for example the payment functionality, it would cause the whole server-side application to go down, and the user would not be able to access any of the functionalities of a web store. Lastly, having in mind the future upgrades, functionality additions, and scalability, a monolithic architecture would be a bottleneck since all the mentioned aspects would ask for the entire codebase and modifications.

In the next section, the paper will cover a microservices architecture, and it will show why is it a better fit for this type of web application, having in mind previously stated issues that it would have with the monolithic approach.

3.2. Microservices architecture

Microservices are an architectural approach to building software applications as a collection of small, independent, and loosely coupled services, each responsible for a specific business function. They are gaining great traction in industry and a growing scientific interest. More and more companies are adopting this architectural style for modernizing their products and capitalizing on its benefits. [4]

3.2.1. Advantages of microservices

Microservices architecture offers several advantages:

- Greater agility in software development
- Flexible on-demand scalability
- Shorter release cycles
- Enhanced fault tolerance and resilience
- Improved maintainability

Microservices are usually loose coupled and independently functioning components, while, as already mentioned, monolithic applications are large and highly coupled systems. This modularity allows for independent development, testing, and deployment of individual services. Teams can work on features in isolation, leading to faster development cycles and improved time-to-market. [5] Individual microservices can be scaled independently based on their specific needs. This allows for efficient resource allocation, cost optimization, and overall agility and reliability. Such key indicators are facilitated and demonstrated in e-commerce domain by successful implementation at otto.de [6]. Additionally, smaller, more focused services are easier to deploy and update. Microservices architecture facilitates continuous integration and continuous delivery (CI/CD) practices, enabling frequent releases with minimal risk of introducing regression. [7]

Each individual service can be developed in a different programming language, or use different technologies, promoting flexibility and team autonomy. Since microservices are loosely coupled, an important outcome of this architecture is that the failure of one service is less likely to cascade and bring down the entire application. [3] This is improvement in overall system reliability and availability.

Microservices, while offering all the stated benefits, also presents several challenges and complexities, that require careful consideration.

3.2.2. Disadvantages and challenges of microservices

Possible disadvantages of a microservices architecture:

- Increased security concerns
- Increased potential for errors
- Scalability and management challenges

- Steeper learning curve and team “maturity”
- Misaligned expectations of benefits

In a microservice architecture, the increased application complexity introduces a larger attack surface with more entry points for potential security vulnerabilities. [8] Managing authentication, authorization, and an API security across multiple services becomes a critical necessity. Choosing the right services granularity and ensuring proper communication between them is crucial to avoid errors and project risks. [9] While microservices offer scalability for individual services, separation of management functionalities from the application can make a difficulty of achieving overall application-level scaling, requiring more intervention from the managing side. [10] Additionally, distributed system management introduces complexities in staying in track with monitoring, logging, and debugging across multiple services. The microservices approach requires a higher level of development team maturity and expertise in distributed compared to the monolithic architectures. [11] This can be a challenge for smaller teams or those new to the microservices concepts.

Simply refactoring a monolithic application to microservices will not automatically yield the same benefits seen in large-scale deployments. Microservices are well-suited for complex, high-traffic application, and may not outperform monoliths on a single machine, especially for a smaller-scale projects. [2]

Project requirements, development team capabilities, budget, deadlines, and much more factors should be carefully evaluated before considering building or transitioning to microservices architecture. While it offers many of the stated potential benefits, it also comes with many challenges, and thus, it is not a one-size-fits-all solution.

For the architecture to be called micro serviced, it needs to follow some principles and rules.

3.2.3. Main principles of a single microservice

The main principles of a single service in microservices architecture are: [2]

- Single responsibility per service
- Microservices independency
- Services act as first-class citizens

Each microservice should embody the principle of single responsibility, adhering to the SOLID principle of software development, where each unit carries out a one specific responsibility, avoiding any overlap in responsibilities with other units, or in this case, services. Microservices are operating autonomously, encapsulating self-contained and deployable services responsible for a specific business function. The boundaries of service are closely related to business requirements and the boundaries of the organizational structure. [12] They encompass all necessary dependencies, including libraries, environments, containers, etc., which they need to operate properly and independently. The principles of technological independence and heterogeneity are important, as they allow for the integration of components from different frameworks. [13] Furthermore, microservice treats its responsible services as a first-class citizen, meaning it's exposing their functionality through APIs while abstracting away implementation details. This encapsulates internal workings, including implementation logic, architecture, and the underlying databases.

3.2.4. Defining microservices

In the last section of the analysis of monolithic architecture, a diagram showing a monolith composing the needing backend functionalities was illustrated.

Based on the previously stated benefits of the microservices architecture, that conquer some of the flaws of the monolithic architecture, it is decided to build this web shop's backend architecture in the microservices style.

The first step to approach the development of microservices is to "breakdown" the monolithic definition of the web shop and separate the functionalities into the services, by following the single responsibility per service approach.

For the first microservice, it is safe to separate all the product functionalities from the rest. The product service will have a single responsibility, and it's handling all the business logic based around products. Such functionalities include products listing, product details, adding, updating, and deleting products.

Secondly, it is required to maintain the inventory status of the products, and to do it, it would be necessary to have the service with the only responsibility to handle that business logic. For that purpose, the inventory service is defined, with a single functionality of handling the product availability status and amount.

Each e-commerce platform should have some form of a cart, handling the logic of selecting products and amounts, and preparing them for checkout. It is decided to handle all the functionalities on the client side, through the browser’s local storage. Based on that, the cart service will not have its belonging microservice on the backend.

To handle all the checkout logic, including personal information, address, ordered products and details, the order service is added. It will take care of the single responsibility of handling orders, in terms of receiving order with user details, address, cart products listing and amounts, and total price.

The last piece of the checkout puzzle is the payment functionality, and since it is a very important feature on its own, it is separated from the order service into its own payment service. It will handle the responsibility of payments, communicating with the 3rd-party payment provider, and ensuring the availability of the payment service, without depending on the availability of other services.

To implement the user reviews functionality on the products, it is safe to separate all the logic around the reviews into its separate microservice. It will contain the business logic for fetching reviews for a specific product, adding a review, editing a review, etc.

Lastly, since it is stated in the system requirements that the Fragrance.hr web shop will contain an AI assistant through the chat interface, the whole business logic around the chatbot and the AI provider can be separated into the separate service, the AI assistant service.

Through the six previously defined services, an initial monolithic structure of the web shop is separated to microservice blocks, each responsible for each own function. Such decomposition can be seen on the diagram (Figure 3.3).

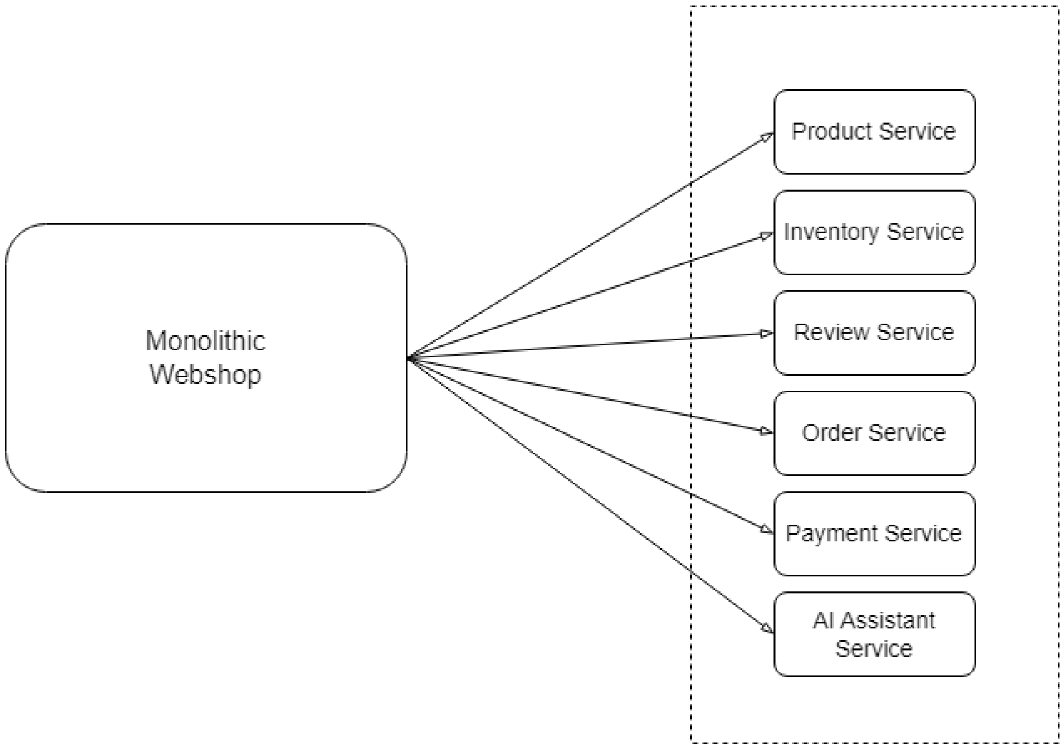


Figure 3.3 Monolith decomposition

Regarding the microservices communication, some application functionalities will require for one microservice to communicate with another, to fetch additional data, etc. Additionally, business logic for handling the authentication and authorization will be added to the API gateway, which is the main entry point for any request to the microservices. This concept, and many other, will be explained in the microservices implementation section.

4. Microservices Implementation

This chapter will delve into the implementation details of the microservices structure of Fragrance.hr web shop.

In the last section, a system architecture comparison was provided, between the microservices architecture and its opposing monolithic architecture. The microservices architecture was chosen, and the initial monolithic definition of our web shop got divided into separate independent services.

The defined microservices of Fragrance.hr web shop application are:

- Product Service
- Inventory Service
- Review Service
- Order Service
- Payment Service
- AI Assistant Service

The full implementation schema, with the client-side application and the server-side with microservices architecture can be seen in diagram (Figure 4.1).

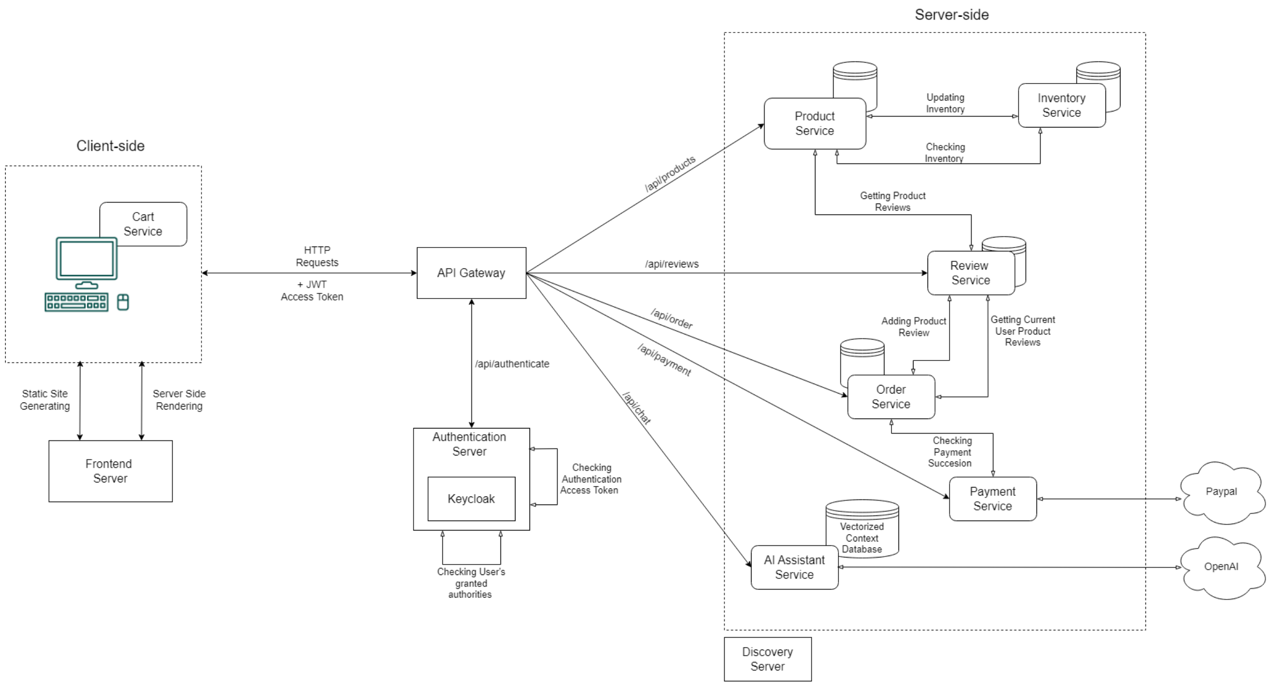


Figure 4.1 Web shop architecture

As explained in the Technologies section, the main technology used for the backend development is Java with the Spring Boot framework. Most of the services are built inside a multi-module project, where each service is an independent module. It is facilitated by Maven, which is a build automation and project management tool. It allows for defining a parent project that includes multiple child modules, where each module can be built, tested, and packaged independently, but they share a common configuration defined in the parent project. [16] With such module independency advantage and reduction of duplicate code, this approach and structure is well suited for the implementation of the web shop services.

Each project contains the *pom.xml* file, which is the fundamental unit of work in Maven. The POM stands for Project Object Model, and it is an XML file which contains project information and configuration details, such as dependencies, plugins, build and source directories, that are used by Maven to build the project. [17]

The child modules of the parent web shop module are specified in its *pom.xml* file, which can be seen in code (Code 4.1).

```
<name>webshop</name>
<modules>
        <module>product</module>
        <module>inventory</module>
        <module>clients</module>
        <module>eureka-server</module>
        <module>api-gateway</module>
        <module>payment</module>
        <module>order</module>
        <module>review</module>
  </modules>
```

Code 4.1 Web shop modules

It is visible from the *<modules>* that the main web shop module contains most of the previously defined microservice modules, like the product, inventory, payment, order, and review services. The one service missing is the AI assistant service. The reason is the version incompatibility, since the assistant service will need a higher Spring Boot version to import needed dependencies and implement needed functionality. Because of that, the AI assistant service is built outside this multi-module structure, as a separate project. The modules regarding the eureka server, API gateway and the client module will be described later.

To ensure consistency across all the modules, the dependency management in the parent module plays a crucial role in managing the dependencies for the entire project. The *dependencyManagement* section of the parent *pom.xml file* allows centralized specification of the dependency versions. Instead of defining the version of a dependency in each separate module, it is defined once in the parent POM, and then each module can refer to it without specifying the version. This allows for better control of versions through the whole project and easier maintenance. The dependency management inside the parent configuration can be seen in the code (Code 4.2). The parent module defines the versions for the Spring Boot and Spring Cloud dependencies.

```
<dependencyManagement>
   <dependencies>
     <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>${spring.boot.maven.plugin.version}</version>
      <scope>import</scope>
      <type>pom</type>
     </dependency>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring.cloud-version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Code 4.2 Parent module dependency management

Additionally, the parent web shop module defines the Docker compose file, which is used to define and manage multi-container Docker applications. It provides a convenient way to specify the configuration of all the containers needed for an application in a single file. For each container, it defines the service and specifies which image to use, the command to run, and other configuration options. Additionally, it defines the networking for the services to communicate with each other, environment variables which can be defined and passed to the containers, and volumes to persist the data. [18] An example of *docker-compose.yml* file with the basic *postgres* service configuration can be seen in code (Code 4.3)

```
services:
  postgres:
    container_name: db-postgres
    image: postgres
    environment:
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: password
      PGDATA: /data/postgres
    volumes:
      - postgres:/data/postgres
    ports:
      - "5432:5432"
    networks:
      - postgres
    restart: unless-stopped
…
```

Code 4.3 Docker compose postgres database container configuration

The main database used for the web shop project is the PostgreSQL open-source relational database [19], set up in the Docker container. Although each microservice can have a separate database defined, for the sake of simplicity, each service will connect to the same PostgreSQL container, but with separate databases for each service which needs the database. Alongside the *postgres* container, the project uses the *pgAdmin*, ran inside the Docker container. It is an open-source administration and development platform for the PostgreSQL. [20]

4.1. REST API

All the microservices are implemented through the REST architectural style. The REST stands for the representational state transfer, and it is a lightweight, flexible, and stateless architecture, which allows clients and components to communicate over HTTP (Hypertext Transfer Protocol).

An API, or Application Programming Interface, is a set of rules and protocols that allows different software applications to communicate with each other. The REST API communicates through HTTP requests to perform standard database functions, like creating, reading, updating and deleting entries, within a specified resource.

For the service to be called RESTful, it needs to adhere to the six main REST design principles [21]:

- Statelessness
- Client-Server Architecture
- Uniform Interface
- Layered System Architecture
- Cacheable
- Code on demand (optional)

The statelessness defines that each request from a client to a server must contain all the information needed to understand and process the request. The server does not store any state about the client session on the server side.

The client and server in the REST implementation are separate independent entities, allowing for separate development, maintainability, and scalability. The requests are managed through the HTTP.

The uniform interface rule between components defines that the information should be transferred in a standard form.

When possible, resources should be cacheable on the client or server side, with the goal of the performance improvement. REST APIs must indicate whether responses are cacheable or not.

In REST APIs, the requests and responses go through different layers. The API needs to be designed so that neither the server nor the client can tell whether it communicates with the end application or an intermediary.

The code-on-demand rule defines the ability to send executable code from the server to the client on request, extending client functionality.

REST API are a common method for connection of components and services in a microservices architecture, and it is used in the implementation of this web shop application.

4.2. Controller-Service-Repository pattern

The Controller-Service-Repository pattern is a widely used architectural pattern for building RESTful applications with Spring Boot. The main benefit of such approach is the separation of concerns. It separates the application into distinct layers, promoting clean code, loose coupling, and easier testing.

The Controller layer acts as the entry point for the API. It is responsible for exposing the functionality so it can be consumed by the clients requesting it. It defines the methods that are mapped to the HTTP requests like GET, POST, PUT, DELETE.

In Spring Boot, the Controller class is annotated with the *@RestController* annotation. It indicates that the response returned by each controller method is written into the response body. Each controller method is annotated with the mapping type, corresponding to the HTTP methods. Therefore, the controller can have *@PostMapping*, *@GetMapping*, *@DeleteMapping* etc. Each mapping method can have a defined *path* attribute, which serves as the navigator, telling the controller which method should be called based on the request path. An example of the Controller class methods of the Product service can be seen in code (Code 4.4).

```
// Base path for all the product endpoints
@RestController @RequestMapping("/api/products")
public class ProductController {
    // Injected service dependency
    private final ProductService productService;

    public ProductController(ProductService productService) {
       this.productService = productService;
    }
    // Handles POST requests to /api/products
    @PostMapping
    public ResponseEntity<Product> createProduct(@RequestBody
    Product product) {
      Product savedProduct = productService.createProduct(product);

      // Return created product with status 200 OK
      return ResponseEntity.ok(savedProduct);
    }
    // Handles GET requests to /api/products/{id}
    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(@PathVariable
    Long id) {
      Product product = productService.getProductById(id);
      if (product == null) {
         // Return 404 Not Found if product not found
         return ResponseEntity.notFound().build();
      }
      // Return product with status 200 OK
      return ResponseEntity.ok(product);
   }
}
```

Code 4.4 Product service controller methods example

This example product controller contains two methods for handling client requests. One for handling the GET requests for retrieving specific product by ID, and one for handling the POST request to create the product. The *createProduct* method takes the *@RequestBody* attribute, which takes the body from the HTTP request and maps it to the Product class. The *getProductById* method receives the *@PathVariable* attribute for the product ID, and the annotation defines that the *ID* value is extracted from the path and mapped to the variable. Both methods are wired to the service layer, since it is handling the business logic around the data required.

Since the controller serves only as a facade of the API, when the client request is needing a specific business functionality to be executed, the Controller passes it forward to the Service layer. The Service layer handles the actual functionality of the application, related to the data. It can also handle additional logic like validation, transformation, or interacting with other services. The Java class annotated with *@Service* indicates that the class is holding the business logic. An example of the product service layer methods is shown in code (Code 4.5)

```
@Service
public class ProductService {
      private final ProductRepository productRepository;

      public ProductService(ProductRepository productRepository) {
          this.productRepository = productRepository;
      }
      public Product createProduct(Product product) {
          return productRepository.save(product);
      }
      public Product getProductById(Long id) {
          return productRepository.findById(id);
      }
}
```

Code 4.5 Service layer methods example

This service class contains two methods, corresponding to the two endpoints being handled in the controller layer. Since both methods are tied to the data, precisely, retrieval and storing the data, the service uses the repository layer to communicate with the database and perform queries.

Lastly, to interact with the data source, the Repository layer is defined. Its primary and only responsibility built around data access method, regarding storing and retrieving the sets of data. The Java class handling this data is annotated with *@Repository*. It indicates that the class provides a mechanism for database operations, such as storage, retrieval, etc. If the business logic from the Service layer requires fetching or storing the data, it is wired to the Repository layer. An example of the repository interface, which is based on the JPA interface, is shown here:

```
public interface ProductRepository extends JpaRepository<Product, Long>
{
    Product save(Product product);
    Product findById(Long id);
}
```

For the Controller to use the Service layer, or for the Service layer to use the Repository layer, dependency injection is used. It allows for loose coupling between the layers. For example, the Controller class should not create the Service object itself. Instead, it declares the service as a dependency. This tells Spring to manage the service's lifecycle and provide an instance when needed.

Precisely, this application will use the Constructor-based dependency injection, where the container invokes a constructor with several arguments, each representing a dependency. The constructor generating, alongside other boilerplate code, is handled by the Lombok library. It is a "time saving" tool that reduces boilerplate code and improves readability and maintenance of code. Constructor-based dependency injection of Product service controller can be seen here:

```
...

@AllArgsConstructor
public class ProductController {

    private final ProductService productService;
    private final ImageService imageService;
     …
```

Each microservice will be based and implemented through the controller-service-repository pattern, separating the logic and responsibilities to three different layers.

4.3. Product service

The microservices list for the web shop implementation starts with the main service needed, the product service. It is a core microservice with a single responsibility of managing the lifecycle of products within the web shop application. The products, in the Fragrance.hr domain, represent the fragrances. The product's lifecycle includes creating, updating, retrieving, and deleting the products and belonging information.

It is obvious that the product service will need the belonging database to store the product information and details. The database entities defined for the product service are shown in figure (Figure 4.2).

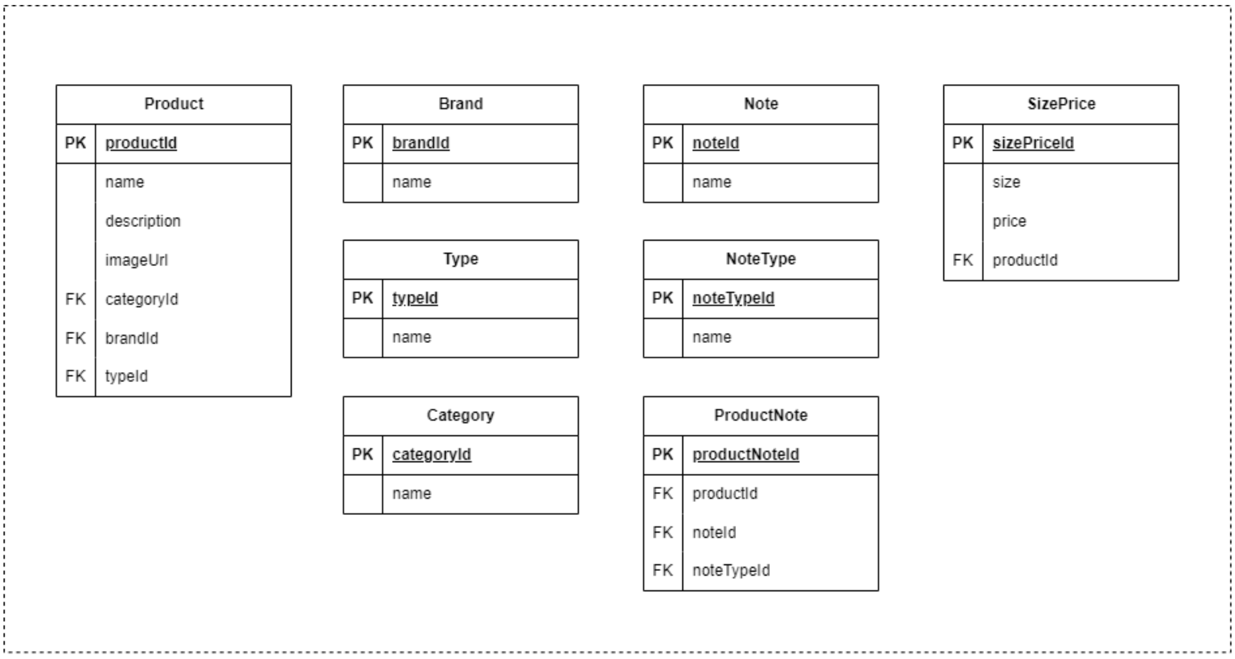


Figure 4.2 Product service database tables

Firstly, the main entity is obviously the *Product* itself. It has basic attributes for the product name, product description, and the main product image URL, which will be used to fetch the belonging cover image from the location. The primary key is the product ID, which is used to identify the product and perform operations like updating, finding, and deleting by product ID. The last three attributes, category, brand, and type IDs are the foreign keys to the belonging tables. The *Product* entity is tied to the *Brand*, *Category*, and *Type* entities through the many-to-one relationship. It means that each product is associated, for example, with one specific brand, while a single brand can be linked to multiple products. This means that multiple products (many) can belong to a single brand (one). This many-to-one rule applies in the same way to the Product-Category and Product-Type relations as well.

The *Brand* entity represents the fragrance (product) brand, and it is represented with the unique brand ID and name. Some examples are Giorgio Armani, Channel, Givenchy, etc. The *Category* entity represents the fragrance categories, which are defined as Male, Female, and Unisex categories. Product *Type* entity refers to the fragrance types, or precisely – concentrations. There are six fragrance types defined: Parfum, Eau de Parfum, Eau de Toilette, Eau de Cologne, Eau Fraiche, Parfum Oil. The *Type* and *Category* tables also consist of the identifier and name attributes.

There are additional entities used to further enhance and expand the fragrance details and information. Since each perfume comes in different size options, and that each size options have a different price set to it, the *SizePrice* entity is defined. It has the ID attribute used as the unique identifier, alongside the size and price attributes. The product ID foreign key ties this table to the product table. The Product entity is tied to the *SizePrice* entity through the one-to-many relationship. It means that a single product can have multiple sizes (with belonging prices) available, whereas each size is associated with only one specific product. The size will be expressed in the millilitre unit, and the price will be expressed in the EUR currency.

The last bit of the fragrance entity definitions are the notes. In the perfume and scent world, each scent is created from many elements, called notes. Some examples are citrus, leather, wood, beans, etc. Each fragrance is built of the combination of many notes, organized through the three layers – top, middle, and base note. The top notes are the one immediately noticed on the first spray of a fragrance. They evaporate quickly, but they leave the first impression on customers. The next note layer is the middle or the heart note. They come through after the top note fades. They form the core of the fragrance. Although they are longer-lasting than the top notes, they fade quicker than the last layer, the base notes. They last the longest, and they are what remains of a fragrance once it has fully settled.

The three layers explained are defined through the *NoteType* entity, with belonging identifier and name attributes. The *Note* entity is used for the notes, also with the identifier and name attributes. As already stated, some examples of notes are citrus, vanilla, wood, leather, etc.

Lastly, the *ProductNote* table ties the fragrance, the note, and the note type entities. From the Figure 10, it is seeable that it contains the ID attribute, along with three foreign keys, tying it to the *Product*, *Note*, and *NoteType* tables. All three of these relations are defined as the many-to-one relationships. It means that each product note is associated with one specific product, one specific note, and one specific note type. Globally looking, if we filter the ProductNote table by some product identifier, one product can have a list of product notes, and that list can be categorized based on the note types, which was the main goal of this structure.

The product service will need a couple of dependencies in order to implement needed functionalities, and those are: *spring-boot-starter-web*, *spring-boot-starter-data-jpa*, and *postgresql* dependencies.

The starter web dependency is essentially a toolbox containing essential components to build a web application. Spring Boot provides a web server and tools to manage HTTP requests and responses, allowing the application to interact with clients through the web.

The PostgreSQL dependency introduces the PostgreSQL JDBC driver, which acts like a bridge between the application and a PostgreSQL database. It allows the application to connect and interact with the database.

The JPA (Java Persistence API) dependency acts like a translator between the Java application and a relation database. It simplifies data access by letting you define data models as Java classes, and database queries through JPA repositories, and letting the Spring wire it up automatically.

For the product service to connect to the database, the database configuration and connection settings need to be setup. Such configuration is set through the *application.yml* file, which is a configuration file used in Spring Boot applications. It stores settings and properties that control how the application behaves. [22] Additionally, a configuration for the JPA is also used. The settings are shown in code (Code 4.6).

```
server:
  port: 8080
spring:
  application:
    name: product
  datasource:
    password: password
    url: jdbc:postgresql://localhost:5432/product
    username: admin
  jpa:
    hibernate:
      ddl-auto: update
    properties:
      hibernate:
        dialect: org.hibernate.dialect.PostgreSQLDialect
        format_sql: true
    show-sql: true
```

Code 4.6 Product service database configuration

As already mentioned, the product service uses Spring Data JPA for the data access functionalities. [23] Each database entity can be reflected on the belonging Java class with the use of JPA annotations. The structure of the Product JPA entity class can be seen in code (Code 4.7).

```
@Entity
public class Product {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long productId;
  private String name;
  private String description;
  private String imageUrl;
  @ManyToOne
  @JoinColumn(name = "category_id", nullable = false)
  private Category category;
  @ManyToOne
  @JoinColumn(name = "type_id", nullable = false)
  private Type type;
  @ManyToOne
  @JoinColumn(name = "brand_id", nullable = false)
  private Brand brand;
  @OneToMany(mappedBy = "product", cascade = CascadeType.ALL)
  private List<SizePrice> sizePrices;
  @OneToMany(mappedBy = "product", cascade = CascadeType.ALL)
  private List<ProductNote> productNotes;
}
```

Code 4.7 Product JPA entity

It is seeable from the code that the product class attributes correspond to the database table attributes, and that product has relations to other classes and that they are reflecting the database table relations with the use of *@ManyToOne* and *@OneToMany* annotations. Remaining entities tied to the product service are structured through the same principles, reflecting their database relations and attributes.

Now that the actual classes and database entities are defined, it is safe to show an example of an actual data object for the product, that is sent as a response to the client HTTP request. Generally, every valid HTTP response will be in JSON (JavaScript Object Notation) format. It is lightweight, human-readable, language-independent, and flexible data structure which is well suited for the HTTP communication. It is built on two structures: a collection of name/value pairs and an ordered list of values, like arrays. [24]

The JSON representation of a product object, which corresponds to the product details HTTP GET request, is shown in code (Code 4.8).

```
{
      "productId": 33,
      "name": "Sauvage",
      "description": "Drawing inspiration from the magical hour of
      twilight in the desert, Sauvage Eau de Parfum exudes a scent
      painted in dark black. His scent trail is a combination of
      the cold of the night and the hot desert air.",
      "imageUrl": "acd8650c-39fe-4725-b7cb-6f08cfad154f.webp",
      "category": {
        "categoryId": 1,
        "name": "Male"
      },
      "type": {
         "typeId": 2,
         "name": "Eau de Parfum"
       },
       "brand": {
         "brandId": 2,
         "name": "Dior"
       },
       "sizePrices": [
          {
            "sizePriceId": 42,
            "size": 100,
            "price": 125.0
          },
            {
              "sizePriceId": 43,
              "size": 200,
              "price": 195.0
             },
             {
               "sizePriceId": 44,
               "size": 60,
               "price": 95.0
              }
          ],
          "productNotes": [
              {
                "productNoteId": 41,
                "note": {
                   "noteId": 38,
                   "name": "Pepper"
                },
                "noteType": {
                   "noteTypeId": 1,
                   "name": "Top"
                  }
              },
              {
                "productNoteId": 43,
                "note": {
                    "noteId": 42,
                    "name": "Nutmeg"
                },
                "noteType": {
                    "noteTypeId": 2,
                    "name": "Middle"
                }
              },
              {
                "productNoteId": 44,
                "note": {
                    "noteId": 49,
                    "name": "Vanilla"
                },
                "noteType": {
                    "noteTypeId": 3,
                    "name": "Base"
                }
              }
          ],
}
```

Code 4.8 Product details JSON response

The next step is to define the functions needed to be handled and performed by the product service. The functionalities will be exposed to the client-side through the Spring

Boot Controller, which acts as the central gateway for handling incoming HTTP requests from the clients. The list of needed endpoints to serve all the needed functionalities is shown in a table (Table 4.1). Additionally, there are additional calls for fetching lists of product brands, types, and categories, but they are used only to fill the selection options on the product add and update forms on the client-side. Therefore, they are not stated in the table (Table 4.1).

Table 4.1 Product service endpoints

Endpoint	**HTTP method**	**Description**
api/products	GET	Fetching all the products
api/products/{productId}	GET	Fetching product details by product ID
api/products/add	POST	Adding new product
api/products/update/{productId}	POST	Updating existing product by product ID
api/products/delete/{productId}	DELETE	Deleting existing product by product ID

The next step in the layers is the service layer, which is called from the controller layer based on the endpoints stated above. Let’s look in few methods of Product Service class, for example the *getProductById* and *addProduct* methods. The method for fetching the product based on the ID is shown in code (Code 4.9).

```
…
private final ProductRepository productRepository;
private final ReviewClient reviewClient;

@Override
public ProductResponse getProductById(Long productId) {
    Optional<Product> product =  productRepository.findById(productId);

    if(product.isEmpty()) {
        return null;
    }
    List<Review> productReviews =
    reviewClient.getReviewsByProductId(product.get().getProductId())
                .getBody();
    AverageRatingAndCount averageRatingAndCount =
    reviewClient.getAverageReviewsRatingAndCountByProductId(product.get()
                .getProductId())
                .getBody();
    ProductResponse productResponse = ProductResponse.builder()
            .productId(product.get().getProductId())
            .name(product.get().getName())
            .description(product.get().getDescription())
            .imageUrl(product.get().getImageUrl())
            .brand(product.get().getBrand())
            .category(product.get().getCategory())
            .type(product.get().getType())
            .productNotes(product.get().getProductNotes())
            .sizePrices(product.get().getSizePrices())
            .reviews(productReviews)
            .averageRatingAndCount(averageRatingAndCount)
            .build();
    return productResponse;
}
```

Code 4.9 Service layer method for fetching product by id

The code shows standard flow of fetching specific product from database. Firstly, the service calls the product repository with the built-in JPA function *findById*, which returns an *Optional* type. If the product is present, the method creates the *ProductResponse* object and builds it by setting its attributes to the ones received from the database.

Alongside fetching product specific information, which is defined in the product entities section above, one more thing is seeable from the method, and it is the variables *productReviews* and *averageRatingAndCount.* On the first clue, it is assumed that this information is related to the other microservice, the Review service, since it is declared to have the responsibility of handling the user reviews of the products. Since it is beneficial to have the actual user reviews and ratings tied to the Product details, through this method, the product microservice communicates with the review microservice. The goal is to fetch the user reviews based on the product ID. It is seeable that it is done through the *ReviewClient* instance and the belonging methods *getReviewsByProductId* and *getAverageReviewsRatingAndCountByProductId*. The whole communication logic and implementation will be explained through the Review service analysis.

The service layer method for adding new product is shown in code (Code 4.10).

```
private final ProductRepository productRepository;
private final SizePriceRepository sizePriceRepository;
private final ProductNoteRepository productNoteRepository;
private final InventoryClient inventoryClient;

@Override
public ProductResponse addProduct(ProductRequest productRequest) {

    Product product = Product.builder()
            .name(productRequest.getName())
            .description(productRequest.getDescription())
            .imageUrl(productRequest.getImageUrl())
            .brand(productRequest.getBrand())
            .category(productRequest.getCategory())
            .type(productRequest.getType())
            .build();

    productRepository.saveAndFlush(product);

    List<SizePrice> sizePrices = new ArrayList<>();
    for (SizePrice sizePrice : productRequest.getSizePrices()) {
        sizePrice.setProduct(product);

        sizePrices.add(sizePrice);
    }
    sizePriceRepository.saveAll(sizePrices);

    List<ProductNote> productNotes = new ArrayList<>();
    for (ProductNote productNote : productRequest.getProductNotes()) {
        productNote.setProduct(product);

        productNotes.add(productNote);
    }
    productNoteRepository.saveAll(productNotes);

    ProductResponse productResponse = ProductResponse.builder()
            .productId(product.getProductId())
            .name(product.getName())
            .description(product.getDescription())
            .imageUrl(product.getImageUrl())
            .brand(product.getBrand())
            .category(product.getCategory())
            .type(product.getType())
            .productNotes(productNotes)
            .sizePrices(sizePrices)
            .amount(productRequest.getAmount())
            .build();

    //Adding the product to inventory
    InventoryItemRequest inventoryItemRequest = InventoryItemRequest
            .builder()
            .productId(product.getProductId())
            .amount(productRequest.getAmount())
            .build();
    inventoryClient.updateInventoryProductAmount(inventoryItemRequest);

    return productResponse;
}
```

Code 4.10 Service layer method for adding new product

On product adding and updating, a specific flow must be fulfilled regarding the multiple entities tied to the product. Firstly, a *Product* entity class is filled with the new data and saved through the JPA repository method. To tie the Product to the *SizePrice* and *ProductNote* entities, it is necessary to set the product object to each element of those lists, and lastly, save them through their separate JPA repositories.

In the last few lines, another communication with other microservice is seeable, and that is the communication with the Inventory service. Since the Inventory service handles the business logic around the updating and maintaining the products inventory amounts, on every product add or update, the ProductRequest object sent from the client contains the amount attribute. This defines the number of products to update the inventory with, based on the product ID. That is why the *InventoryItemRequest* object is built with those attributes and sent to the inventory service through the *InventoryClient* instance and the method *updateInventoryProductAmount*. Additionally, in the method handling product deletion, the inventory service is called to delete the product inventory entry from the inventory database. The logic behind the inventory-product communication will be explained in the Inventory service section.

4.4. Inventory service

The Inventory service is a simple microservice responsible for handling the product inventory status and amount. To retrieve and store information regarding the product availability and amount, it needs to be connected to the belonging inventory database.

The Inventory service, due to its simplicity, only needs one database entity, which is shown in figure (**Pogreška! Izvor reference nije pronađen.**). The *InventoryItem* table consists of the ID, product ID, and amount attributes.

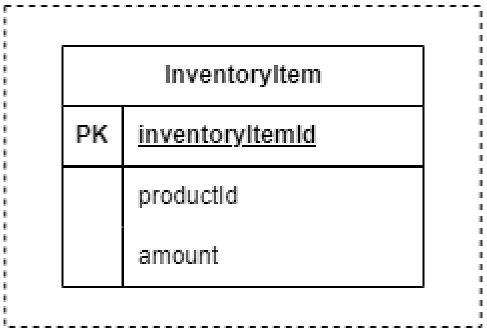


Figure 4.3 Inventory service database table

Regarding the functionalities, the inventory service needs to handle the product inventory amount setup, update, deletion, and additionally, it needs to provide the functionality to check if the specific product with the specific amount is available in the inventory. Such functionality is needed, for example on the products cart checkout in the web shop, where the product needs to be checked for availability before proceeding to the order and purchase steps.

In terms of the dependencies needed to implement the needed functionalities, the Inventory service uses the same dependencies as the Product service, which are *spring-boot-starter-web*, *spring-boot-starter-data-jpa*, and *postgresql* dependencies. The same applies to the configuration properties, with the addition of defining a separate database for the inventory microservice only.

Let's define the Inventory service endpoints that will be exposed to the client to make requests to. They will correspond to the functionalities defined above. Such controller endpoints definition can be seen in table (Table 4.2).

Table 4.2 Inventory service endpoints

Endpoint	**HTTP method**	**Description**
api/inventory/check	POST	Checking is the product with specified amount in stock
api/inventory/product-update	POST	Updating product inventory with new amount by product ID
api/inventory/products-order	POST	Updating product inventory amounts on order
api/inventory/remove/{productid}	DELETE	Deleting existing product inventory by product ID

The Inventory service will not be directly accessible from the client-side of the web shop application. It will only be called by other microservices, in this case, by the already mentioned Product service and by the Order service. The Product service communicates with the Inventory service through the product adding, updating, and deleting functionalities, where it needs to call the specific Inventory service functionality to handle the business logic around the product inventory handling. Since that is not the concern of the Product service.. Regarding the Order service, it will communicate with the Inventory service on the product availability check and after the order completion, to update the ordered product inventory amounts.

There are two ways how two microservices can communicate with each other, synchronous and asynchronous communication. [25]

In a synchronous communication, one microservice sends a request to another and waits for a response before proceeding. It is a standard approach to microservices communication. It is simple, with clear flow of control, and suitable for situations where the immediate result is needed. Furhermore, it is often achieved through HTTP-based protocols. The main disadvantage is that the synchronous communication can lead to potential blocking, if the called service is slow or unresponsive.

In asynchronous communication, a microservice sends a request to another and continues its execution without waiting for a response. The called service processes the request independently and responds later. Regarding the advantages, it provides decoupled interaction and services independency, and it enables better responsiveness. Compared to the synchronous communication implementation, it is more complex to manage, and it typically requires message brokers, like *RabbitMQ*.

Regarding the Product and Inventory services communication, all the functionalities provided from the Inventory service to the Product service need to be executed synchronously, upon the request from the Product service. The reasoning is that, for example, product amount update needs to be executed as soon as possible, because the product inventory availability and amount needs to be up to date. It cannot rely on asynchronous way of handling the communication, where the product inventory update could be postponed. The same rule applies to the Order and Inventory service communication as well.

To achieve the synchronous communication, the Inventory service uses the Open Feign. [26] It is an open-source tool that simplifies calling other microservices in the application. Feign is a REST client that creates a dynamic implementation of the interface that is declared as the Feign Client, which is used to consume REST API endpoints which are exposed by a microservice. It lets it seem like local methods are called, but behind the scenes it makes HTTP requests to other services. This makes the code cleaner and easier to manage. [27]

To implement Feign, firstly the dependency needs to be added to the main *pom.xml* file, which is the web shop parent module. The dependency is shown here:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

Secondly, an interface needs to be declared, which will act as the Feign Client for the Inventory service. The Inventory client methods will serve as the mirror to the API endpoints that are needed to be called on the Inventory microservice. Feign takes care of the details, like creating the HTTP requests and handling response, and the only implementation that needs to be done is to call the methods from the Feign Client interface. The Inventory Client interface is shown in code (Code 4.11).

```
@FeignClient("inventory")
public interface InventoryClient {
    @GetMapping(path = "api/inventory/check")
    ResponseEntity<Boolean> isProductInStock(@RequestBody
InventoryItemRequest inventoryItemRequest);

    @PostMapping(path = "api/inventory/product-update")
    void updateInventoryProductAmount(@RequestBody
InventoryItemRequest inventoryItemRequest);

    @PostMapping(path = "api/inventory/products-order")
    void updateProductsAmountOnOrder(@RequestBody
List<InventoryItemRequest> inventoryItemRequestList);

    @DeleteMapping(path = "/remove/{productId}")
    void removeProductFromInventory(@PathVariable("productId") Long
productId);

}
```

Code 4.11 Inventory Feign Client interface

The code shows all the methods from the Inventory service that are “exposed” through the Feign Client and can be reached by other microservices. The mappings and path correspond to the actual endpoints of the Inventory microservice. The *@FeignClient* annotation defines that the *InventoryClient* interface is a Feign Client, with the mandatory value argument.

To make the code cleaner, all the Feign clients, starting with the Inventory, will be placed in a separate module called *clients.*

When some microservice needs to communicate with another microservice which uses the Feign Client, the microservice needs to add annotation *@EnableFeignClients*, enabling the component scanning for Feign Client interfaces. Since the Product service will need to communicate to the Inventory service through the Inventory’s interface, it defines the annotation in its *ProductApplication* main class, which can be seen here:

```
@EnableFeignClients(
        basePackages = "com.fer.hr.clients"
)
public class ProductApplication {
    public static void main(String[] args) {
        SpringApplication.run(ProductApplication.class, args);
    }
}
```

The *basePackages* attribute defines in which packages are the Feign Client interfaces located.

Additionally, the Product service needs to declare dependency for the *clients* module, importing all the client interfaces and methods that can be called. The dependency for the import of *clients* module can be seen here:

```
<dependency>
    <groupId>com.fer.hr</groupId>
    <artifactId>clients</artifactId>
    <version>1.0-SNAPSHOT</version>
    <scope>compile</scope>
</dependency>
```

As already seen in the Product service methods for adding, updating, or deleting the product, through the injected *InventoryClient*, the Product service calls for the methods defined in the interface, wiring it to the Inventory microservice and belonging endpoints, handling the business function required. An example of such call can be seen here:

```
InventoryItemRequest inventoryItemRequest = InventoryItemRequest
        .builder()
        .productId(product.getProductId())
        .amount(productRequest.getAmount())
        .build();
inventoryClient.updateInventoryProductAmount(inventoryItemRequest);
```

4.5. Review service

The Review microservice, as the name suggests, handles the responsibility and business logic around user reviews on the products. The user review can be added, through the orders list page, on the purchased products. The reviews can be seen on two places. Through the completed order details, as a singular user review fetched by the user ID and product IDs, and on the product details page, fetched by the product ID. Since the product can have multiple reviews from multiple users, the product details also fetch the calculated average rating and ratings count of all the user reviews.

Since the reviews need to be stored and fetched, the Review service will have its own database. The Review service database entity logic is simple, containing one Review entity, as it can be seen on figure (Figure 4.4).

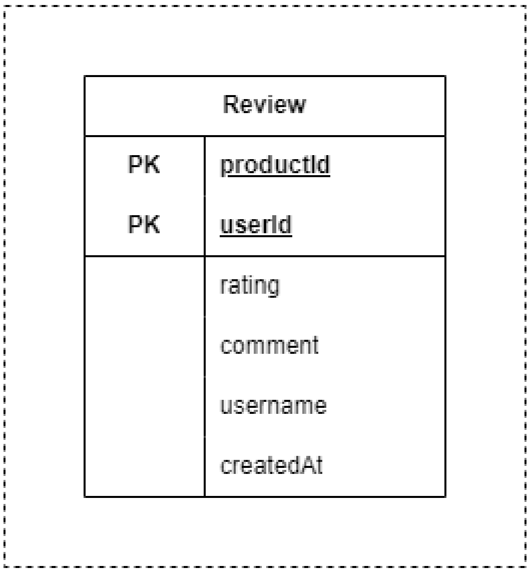


Figure 4.4 Review service database table

Since each review is identified based on the ID of the product rated, and the ID of the user making the review, its primary key is a composite key created from the product ID and user id. Additionally, the Review contains attributes such as rating, comment, the username, and the date of the review creation.

An example of Review in JSON format, fetched as the response to the get request, is shown here:

```
{
    "reviewId": {
    "productId": 52,
            "userId": "93516232-2fae-4169-8425-2ef9fc3ce333"
},
    "userName": "Dalijo",
        "rating": 5,
        "comment": "it’s awesome. I recommend it",
        "createdAt": "2024-05-14T17:30:30.436+00:00"
}
```

The Review microservice uses the same dependencies and configurations as the Inventory service. It is also based on the controller-service-repository pattern.

Functionality wise, as already mentioned, the Review service needs to expose the endpoints for adding a review, fetching reviews by product ID, fetching one review by product and user ID. It also needs the endpoint for fetching average rating and count by the product ID. Those endpoints are listed in table (Table 4.3).

Table 4.3 Review service endpoints

Endpoint	**HTTP method**	**Description**

api/reviews/{productId}	GET	Fetching reviews by product ID
api/reviews/{productId}/user/{userId}	GET	Fetching specific review by product ID and user ID
api/reviews/add	POST	Adding new review
api/reviews/average/{productId}	GET	Fetching average rating and reviews count by product ID

Since the Review service will communicate with other microservices, the Product and Order microservices, an additional Feign Client will be created, this time for wiring the requests to the Review service. The implementation approach is the same as in the Inventory Service, and the Review Client will also be defined in the separate *clients* module. The Feign Client interface for the Review service methods is shown in code (Code 4.12). The interface does not define the review add POST method, since that method is handled by the Review service itself.

```
@FeignClient("review")
public interface ReviewClient {

    @GetMapping("/api/reviews/average/{productId}")
    public ResponseEntity<AverageRatingAndCount>
getAverageReviewsRatingAndCountByProductId(@PathVariable("product
Id") Long productId);

    @GetMapping("/api/reviews/{productId}")
    public ResponseEntity<List<Review>>
getReviewsByProductId(@PathVariable("productId") Long productId);

    @GetMapping("/api/reviews/{productId}/user/{userId}")
    public ResponseEntity<Review>
getReviewByProductIdAndUserId(@PathVariable("productId") Long
productId, @PathVariable("userId") String userId);
}
```

Code 4.12 Review Feign Client interface

The usage of the Review Client was already shown in the product details fetch method of the Product service, where the review list and average rating and count are appended to the product response object. The call from Product to Review microservice can be seen here:

```
List<Review> productReviews = reviewClient
        .getReviewsByProductId(product.get().getProductId())
```

```
          .getBody();
AverageRatingAndCount averageRatingAndCount =
  reviewClient
          .getAverageReviewsRatingAndCountByProductId(product.get()
          .getProductId())
          .getBody();
```

4.6. Order service

The Order microservice is responsible for handling the business logic around checkout and managing user, order, and payment information. It consists of two functionalities: receiving and storing order details and retrieving orders history by the user ID.

The Order service connects to the belonging database, which contains two tables, the *Order* and *CartItem* table. The *Order* entity contains attributes regarding the order details, such as order ID, time of order creation, pay ID, payer ID, total sum for payment, and user details like name, address, email, phone number, etc. The *CartItem* entity stores the actual products purchased, with the order ID foreign key, wiring it to the *Order* entity. Each item contains information like product ID, name, amount purchased, size, price, image URL, brand and type. The tables can be seen in figure (Figure 4.5).

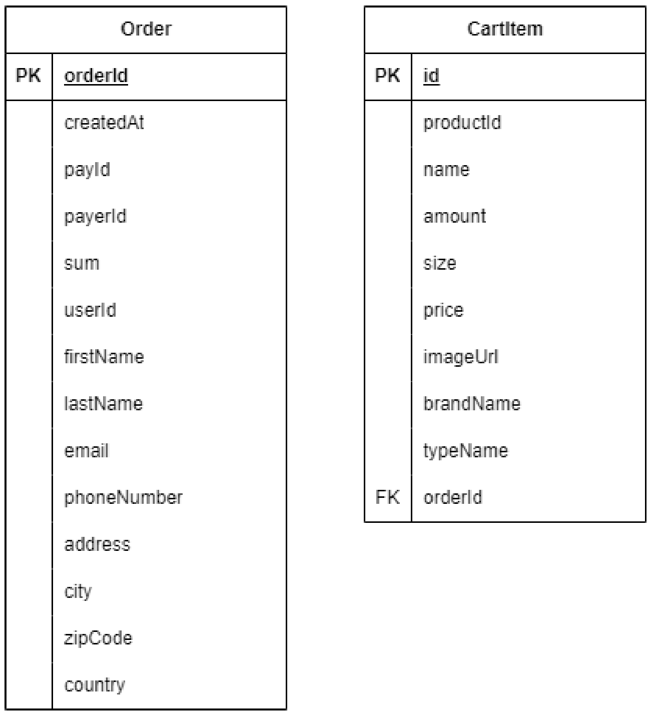


Figure 4.5 Order service database tables

Regarding the dependencies, Order microservice uses the same dependencies as the other microservices, with one difference. To change things up regarding the database interaction, the Order microservice uses JDBC. It stands for Java Database Connectivity, and as well as the JPA, it is a translator between the Java program and the relation database which allows connecting and running SQL queries on various databases. [28]

In comparison to the JPA, JDBC is a low-level, where the SQL queries and mapping to Java objects is done manually. JPA is a higher-level approach that automatically translates operations on the Java class objects to the SQL queries. It is simpler but offers less control over the database operations.

The Order service repository JDBC method for fetching the order list from the database based on the user ID, with the belonging mapper for mapping it to the Java class object is shown in code (Code 4.13).

```
@Override
public List<Order> getOrdersByUserId(String userId) {
    final String sql = "SELECT id, sum, payid, payerid, createdat
FROM public.order WHERE userid = :userId";

    Map<String, Object> params = new HashMap<>();
    params.put("userId", userId);
    try {
        return namedParameterJdbcTemplate.query(sql, params,
orderRowMapper());

    } catch (DataAccessException e) {
        log.error(e.getMessage());

    }
    return null;
}

public RowMapper<Order> orderRowMapper() {
    return ((rs, rowNum) -> {
       Order order = Order.builder()
               .id(rs.getLong("id"))
               .sum(rs.getFloat("sum"))
               .payId(rs.getString("payid"))
               .payerId(rs.getString("payerid"))
               .createdAt(rs.getDate("createdat"))
               .build();
       return order;
    });
}
```

Code 4.13 JDBC method for fetching orders

Firstly, the actual SQL query string is defined with one parameter for the user ID. That parameter is passed to the query through the parameters map. The Order service for the JDBC queries uses the named parameter JDBC template. Instead of the regular JDBC approach with using “?” as the placeholders for values, the named parameter uses placeholders like ID, name, etc. Then, a separate map of objects associating the placeholders with the actual values can be made and applied to the SQL statement. This makes the code cleaner and improves readability. Any code that communicates with the database can throw an error regarding the data access, and that is why the query method of the JDBC is wrapped with the try-catch block, for handling such exception. Along with the SQL string and the parameters map, the query method takes the row mapper function argument as well. The *orderRowMapper* function, acts as a translator, taking each row of data retrieved from the database and converting it into a specific object, through specifying the column name and column type of the value mapped.

As already mentioned, the Order microservice contains two endpoints, which are shown in the table (Table 4.4).

Table 4.4 Order service endpoints

Endpoint	**HTTP method**	**Description**

api/order	POST	Saving order details
api/order/{userId}	GET	Fetching orders by user ID

As it is already stated, each order made from a specific user needs to have information if the products purchased through the order are reviewed by the user who purchased them. For that reason, the Order microservice needs to communicate with the Review microservice, and fetch the review based on the product and user ID. There can only be one review left from a specific user on the specific product, so the response of such request is a singular review object. As already defined in the Review service section, the Review Feign Client interface is used, with the method that maps the request to the Review service *api/reviews/{productId}/user/{userId}* path. The getOrderByUserId method of the Order service, implementing the communication with the Review Client, can be seen in code (Code 4.14).

```
@Override
public List<Order> getOrdersByUserId(String userId) {
    List<Order> orders = this.orderDao.getOrdersByUserId(userId);

    for (Order order : orders) {
        List<CartItem> cartItems =
this.orderDao.getItemsByOrderId(order.getId());
        for (CartItem cartItem : cartItems) {
            Review review =
reviewClient.getReviewByProductIdAndUserId(cartItem.getProductId(
), userId).getBody();
            cartItem.setReview(review);
        }

        order.setItems(cartItems);
    }
    Collections.reverse(orders);
    return orders;
}
```

Code 4.14 Order and Review service communication

4.7. Payment service

The Payment microservice is responsible for handling the payment process of the web store. It is implemented with the PayPal integration, which is one of the most common payment service providers used in the web applications. The payment process is triggered when the personal details information is filled and the PayPal payment method is selected.

To integrate the PayPal payment service provider into the web shop application, the PayPal SDK (Software Development Kit) needs to be used. It is a set of tools that allow for integration of PayPal's secure checkout and payment methods into the website or application. [29] Since the integration is done through the server-side Payment microservice in Java and Spring Boot, the *checkout-sdk* dependency is used, which allows the REST API, such as the Payment service, to integrate the PayPal checkout.

Firstly, to implement the PayPal functionality, the configuration properties for the PayPal environment need to be set. The properties include setting the client ID and client secret, which are used to authenticate API calls. The client ID identifies the application, where the client secret authenticates the client ID. [30] The configuration for the PayPal HTTP client is shown here:

```
@Configuration
public class PaypalConfiguration {
    @Bean
    public PayPalHttpClient getPaypalClient(
            @Value("${paypal.clientId}") String clientId,
            @Value("${paypal.clientSecret}") String clientSecret) {
        return new PayPalHttpClient(new
PayPalEnvironment.Sandbox(clientId, clientSecret));
    }
}
```

The complete payment flow and communication between the Payment microservice, client side, and the PayPal is shown in diagram (Figure 4.6).

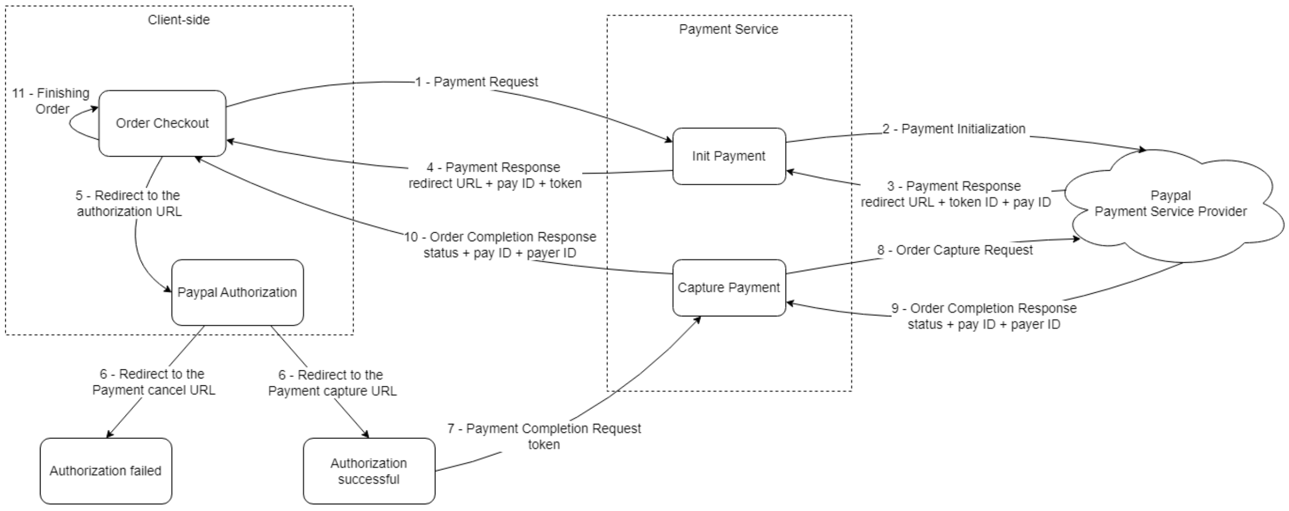

Client-side
Payment Service
11 - Finishing Order
Order Checkout
1 - Payment Request
Init Payment
2 - Payment Initialization
Paypal Payment Service Provider
3 - Payment Response redirect URL + token ID + pay ID
4 - Payment Response redirect URL + pay ID + token
5 - Redirect to the authorization URL
10 - Order Completion Response status + pay ID + payer ID
8 - Order Capture Request
Paypal Authorization
Capture Payment
9 - Order Completion Response status + pay ID + payer ID
6 - Redirect to the Payment cancel URL
6 - Redirect to the Payment capture URL
7 - Payment Completion Request token
Authorization failed
Authorization successful

Figure 4.6 Payment service flow

Firstly, the client-side initiates a checkout, by sending a payment request to the Payment microservice. The payment request contains the parameter *sum*, which defines the total purchase price. The request is handled through the *api/payment/init* endpoint of the Payment microservice and the method *createPayment*, visible in code (Code 4.15).

```
public PaymentOrder createPayment(BigDecimal fee) {
    //Create a new OrderRequest object to define the payment details
    OrderRequest orderRequest = new OrderRequest();

    //Set the payment intent to "CAPTURE" for immediate payment
    orderRequest.checkoutPaymentIntent("CAPTURE");

    //Set the amount details with currency code and the fee to pay
    AmountWithBreakdown amountWithBreakdown = new AmountWithBreakdown()
            .currencyCode("EUR").value(fee.toString());

    //Create a PurchaseUnitRequest object which contains the amount
breakdown
    PurchaseUnitRequest purchaseUnitRequest = new PurchaseUnitRequest()
            .amountWithBreakdown(amountWithBreakdown);

    //Add the purchase unit request to the order request
    orderRequest.purchaseUnits(List.of(purchaseUnitRequest));

    //Define the return and cancel URL for the user navigation after
payment
    ApplicationContext applicationContext = new ApplicationContext()
            .returnUrl("http://localhost:4200/purchase/capture")
            .cancelUrl("http://localhost:4200/purchase/cancel");
    orderRequest.applicationContext(applicationContext);

    //Create the OrdersCreateRequest object to send to Paypal HTTP client
    OrdersCreateRequest ordersCreateRequest = new OrdersCreateRequest()
            .requestBody(orderRequest);

    try {
        //Try to execute the request to create order on Paypal
        HttpResponse<Order> orderHttpResponse =
payPalHttpClient.execute(ordersCreateRequest);
        Order order = orderHttpResponse.result();

        //Extracting the redirect URL from response to navigate to the
Paypal view
        String redirectUrl = order.links().stream()
                .filter(link -> "approve".equals(link.rel()))
                .findFirst()
                .orElseThrow(NoSuchElementException::new)
                .href();
        return new PaymentOrder("success", order.id(), redirectUrl);

    } catch (IOException e) {
        log.error(e.getMessage());
        return new PaymentOrder("error");
    }
}
```

Code 4.15 Service method for creating a payment

The method prepares the order details, such as the fee, currency code, payment intent, return and cancel URLs. Then, it executes the request with the order create object and sends it to the PayPal Client. If the request is successful, the Payment service will send the status, order (pay) ID and redirect URL with the token to the client-side.

Based on the redirect URL received, if the initialization was successful, the client-side will redirect to the PayPal authorization window. If the user authorization fails, the payment process is cancelled. If the authorization succeeds, a request for payment completion, containing the token, is sent to the Payment microservice, to the *api/payment/capture* endpoint. The endpoint triggers the *completePayment* method of the Payment service, visible in code (Code 4.16).

```
public CompletedOrder completePayment(String token) {
    //Create a new OrderCaptureRequest object to capture the
authorized payment using the provided token
    OrdersCaptureRequest ordersCaptureRequest = new
OrdersCaptureRequest(token);
    try {
        //Execute the capture request using the Paypal HTTP client
        HttpResponse<Order> httpResponse =
payPalHttpClient.execute(ordersCaptureRequest);
        if (httpResponse.result().status() != null) {
            //Payment captured successful, return success message
and token
            return new CompletedOrder("success", token);
        }
    } catch (IOException e) {
        log.error(e.getMessage());
    }
    //Payment capture failed, return error message
    return new CompletedOrder("error");
}
```

Code 4.16 Method for payment completion

The method creates the *OrderCaptureRequest* object, with the token attribute set. Then, it executes the order capture request on the PayPal HTTP client. The payment results are returned to the client-side. If the payment capture is successful, the payment process is completed, otherwise the payment is cancelled.

4.8. AI Assistant service

To help the customers when visiting the web shop application, just like in the regular psychical store, an assistant is needed. In terms of the web applications, customer assistance is mostly implemented in a form of the chat. Through the chat, customers can ask various questions regarding the issue, or the information required, in the domain of the web application visited. The chat agent is usually implemented as an AI assistant, the chatbot.

The Fragrance.hr web shop will use the AI assistant to help customers by giving useful information regarding the store policies, returns, payment, and more, based on the question asked.

The AI assistant is built on the RAG (Retrieval-Augmented Generation) technique. It is a technique that combines LLMs (Large Language Models) with external knowledge sources to improve the accuracy and factual grounding of the LLM’s outputs. LLM is a type of AI that can understand and generate human language text. These models are trained on massive amounts of data, which allows them to recognize and understand language patterns, and generate response. RAG essentially addresses the limitations of LLMs. While LLMs are excellent at processing language and generating response, their factual knowledge comes solely from their training data. RAG bridges this gap by retrieving information from external source and allowing LLMs to access and leverage external knowledge bases, making it generate responses that are more accurate and relevant. RAG is useful for extending the LLM capabilities to specific domains and knowledge bases. That is why this web shop application assistant is built on RAG, so that the chatbot can be focused on the provided web shop context only, removing unpredictability from the LLM responses.

The key elements of the Retrieval Augmented Generation technique are the Embedding model, Vector database, and the already mentioned LLM. [39]

The Embedding, or a vector, is a numerical representation of the word. It is a way to capture the meaning and relationships of words in numerical format. Embedding enables AI models to efficiently find similar and relevant objects and information. For RAG to use the embedded context for comparison, it needs to store it in the database.

The Vector database is a specialized type of database that stores embeddings (vectors). The queries performed on the vector databases are based around similarity searches. When the user query is given, the embedding model converts it to embeddings and sends them to the vector database. The vector database then performs the similarity search between the query embeddings and the stored embeddings and returns the embeddings that are like the query embedding.

RAG will be implemented in the AI Assistant microservice through the *Spring AI*, which serves as the foundation for developing AI applications in Java.

The whole process of the Retrieval Augmented Generation for this web application's AI assistant is shown on diagram (Figure 4.7).

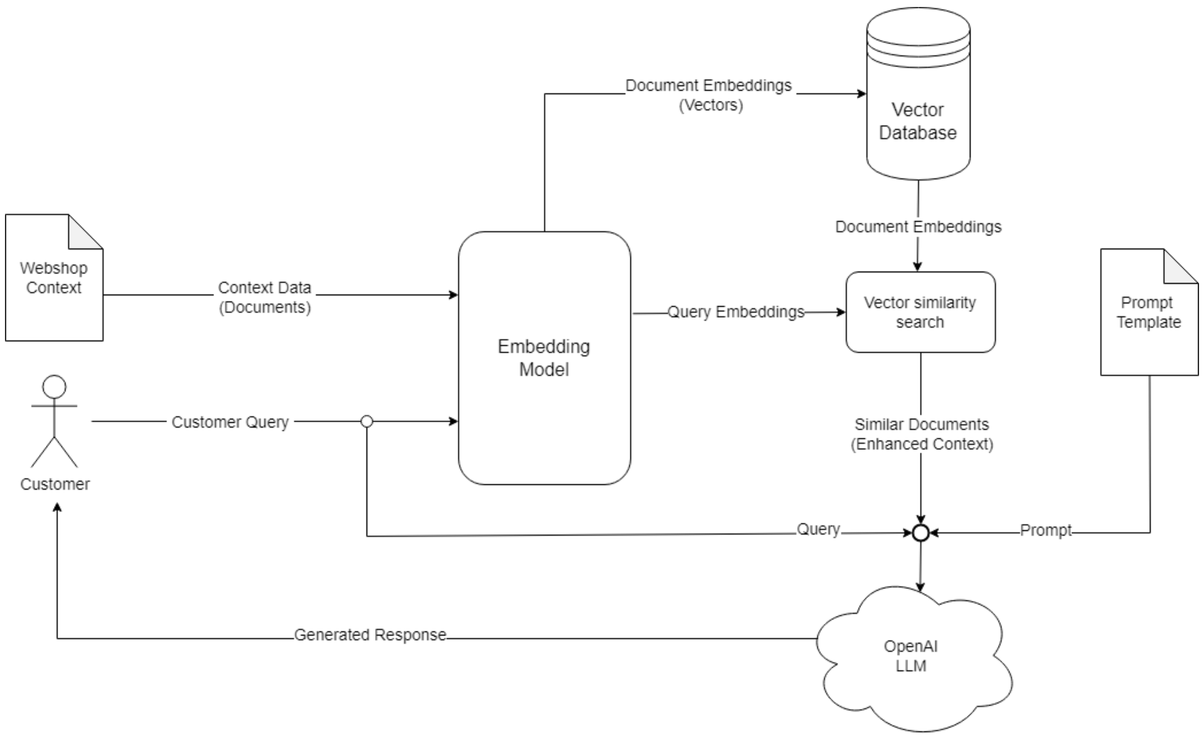


Figure 4.7 Retrieval Augmented Generation process

Firstly, the AI Assistant service contains two predefined templates – the web shop context and the prompt template. The web shop context is a document containing information about Fragrance.hr web application. That information includes application specific description, functionalities offered, products overview, etc. Additionally, it contains the web shop policies, such as the return policy, checkout, payment, and delivery information. Based around this context, the LLM will generate the response as the answer to the customer question. Secondly, the prompt template is used to guide the LLM in the response generation, to improve accuracy and relevance. Additionally, through the prompt template a "personality" is given to the LLM, making it seem like a friendly, helpful and vibrant chat support agent of Fragrance.hr. The prompt template also holds the parameters for the documents and user input insert, that will be defined before the chat client call.

Initially, the context needs to be vectorized (embedded) and stored in the vector database. For the vector database implementation, Spring AI offers the *VectorStore* interface, which contains methods for similarity searches and context documents adding to the database. On each application start, if it doesn't exist, the vector store file will be created, which is a JSON file containing the application context in a form of embeddings list formed from the content. The embedding client acts as the transformer upon the insertion of document tokens into the vector database, where it transforms them into the vector embeddings. The whole logic can be seen in code (Code 4.17).

```
@Value("classpath:/docs/webshop-context.txt")
private Resource context;

@Value("vectorstore.json")
private String vectorStoreName;

@Bean
SimpleVectorStore simpleVectorStore(EmbeddingClient embeddingClient) {
    //Define SimpleVectorStore object with the client for embedding the
context documents
    SimpleVectorStore simpleVectorStore = new
SimpleVectorStore(embeddingClient);
    File vectorStoreFile = getVectorStoreFile();

    //Check if the vector store file already exists
    if(vectorStoreFile.exists()) {
        //Load pre-processed vectors
        simpleVectorStore.load(vectorStoreFile);
    }
    else {
        //Create TextReader object for web shop context path
        TextReader textReader = new TextReader(context);
        textReader.getCustomMetadata().put("filename", "context.txt");

        //Read documents from the context
        List<Document> documents = textReader.get();

        TokenTextSplitter tokenTextSplitter = new TokenTextSplitter();
        //Split documents into tokens (words, phrases, etc.)
        List<Document> splitDocuments =
tokenTextSplitter.apply(documents);

        //Add tokenized documents to the vector store
        simpleVectorStore.add(splitDocuments);
        //Save the processed vector store to a file
        simpleVectorStore.save(vectorStoreFile);
    }
    return simpleVectorStore;
}
```

Code 4.17 Saving context embeddings to vector store

The AI Assistant microservice contains one endpoint, and it is the chat endpoint on path /api/assistant/chat, which expects the question message parameter from the user. The controller method for that endpoint can be seen in code below. If the user sends the blank message, the default prompt value is set.

```
@GetMapping("/chat")
public ChatResponse chat(@RequestParam(value="message",
defaultValue = "Hi! Who are you") String message) {
    return assistantService.chat(message);
}
```

The AI Assistant service uses the OpenAI gpt-3.5-turbo model for the generative based LLM. The OpenAI chat client contains the *call* function that takes the prompt template argument. The prompt template contains initial prompt setup information, and additionally, the user input and enhanced context fetched from the vector database based on the similarity search. The search is initially run at the start of the function, based on the user message and the stored context embeddings in the vector store. The whole service handling the chat call can be seen in Code 24.

```
@Service
public class AssistantService {

    // ChatClient bean for interacting with the chat engine
    private final ChatClient chatClient;

    // VectorStore bean for interacting with the vector store
    private final VectorStore vectorStore;

    // Path to the assistant prompt template
    @Value("classpath:/prompts/assistant-prompt-template.st")
    private Resource assistantPromptTemplate;

    public AssistantService(ChatClient chatClient, VectorStore
vectorStore) {
        this.chatClient = chatClient;
        this.vectorStore = vectorStore;
    }

    public ChatResponse chat(String message) {
        // Run the vector store similarity search for similar context
documents based on the user message
        List<Document> similarDocuments = vectorStore.similaritySearch(
                SearchRequest.query(message) // Build a search request
with the user message
                        .withTopK(2)); // Retrieve the top 2 most similar
documents

        // Extract the content (text) of the retrieved documents
        List<String> contentList = similarDocuments
                .stream()
                .map(Document::getContent).toList();

        // Load the assistant prompt template
        PromptTemplate promptTemplate = new
PromptTemplate(assistantPromptTemplate);

        // Create a map to store prompt parameters
        Map<String, Object> promptParameters = new HashMap<>();
        promptParameters.put("input", message);
        promptParameters.put("documents", String.join("\n",
contentList));

        // Create a prompt object using the prompt template and
parameters
        Prompt prompt = promptTemplate.create(promptParameters);

        // Create a ChatResponse object to hold the response message
        ChatResponse chatResponse = new ChatResponse();

        // Use the OpenAI ChatClient to call the chat engine with the
generated prompt template
        String answer =
chatClient.call(prompt).getResult().getOutput().getContent();
        chatResponse.setMessage(answer);

        // Return the ChatResponse object containing the generated answer
        return chatResponse;

    }
}
```

4.9. Service discovery

Service discovery plays a crucial role in enabling microservices to locate and communicate with each other effectively. It eliminates the need for hardcoding the microservices locations and ports.

Each microservice registers itself with the service registry upon startup, providing its name and network details. When a microservice needs to communicate with another microservice, it consults the service registry to find the location of the target microservice. The service registry searches its database based on the requested microservice name and retrieves the corresponding network details. With the network location, the initiating microservice can initiate direct communication with the target microservice. Such flow can be seen on the diagram (Figure 4.8).

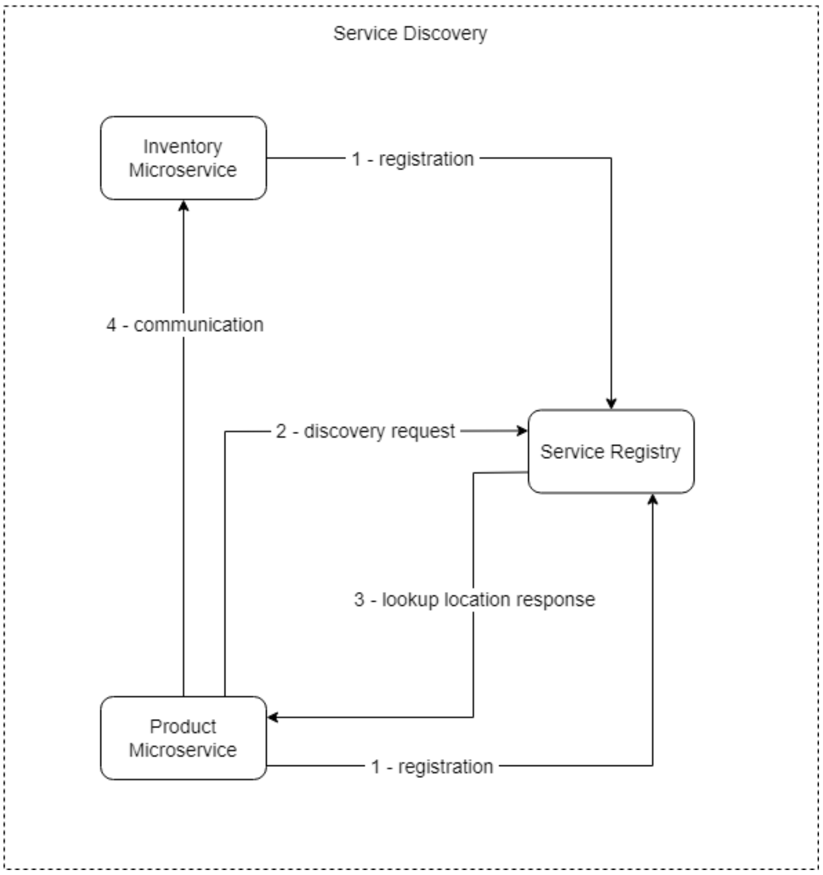


Figure 4.8 Service discovery process

To implement service discovery, the web shop server-side uses the *Spring Cloud Netflix*, which offers all the necessary tools to build the discovery server. [31]

The Eureka Server is defined in the separate module in the multi-module project structure. To create and run the Netflix Eureka discovery server, the module needs to import the discovery server dependency, which can be seen here:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

Additionally, the main application needs to be annotated with *@EnableEurekaServer*, as seen here:

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {

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

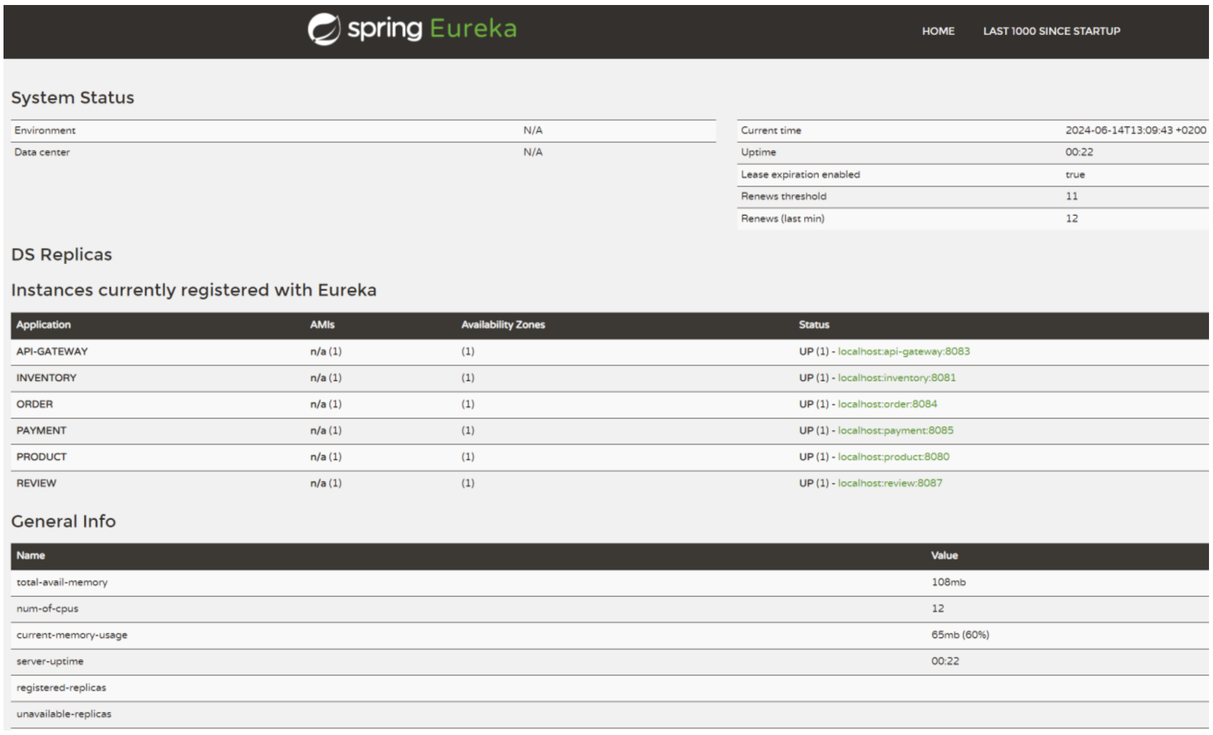
Any microservice that wants to register inside the discovery server, and that wants to communicate with other registered microservices, needs to register as the *Eureka Client*. To do so, all the web shop microservices need to import the client dependency that can be seen in this code:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

Additionally, microservice needs to configure eureka properties in order to register and communicate with the discovery server. The *application properties* of such service discovery client are shown here:

```
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka
    fetch-registry: true
    register-with-eureka: true
    enabled: true
```

Lastly, the microservice application needs to annotate with @EnableEurekaClient to mark itself as the Eureka client. Figure (Figure 4.9) shows the Eureka interface, with all the registered microservices in the web shop application. It checks their address, port, health status and availability.



System Status

Environment	N/A
Data center	N/A

Current time	2024-06-14T13:09:43 +0200
Uptime	00:22
Lease expiration enabled	true
Renews threshold	11
Renews (last min)	12

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
API-GATEWAY	n/a (1)	(1)	UP (1) - localhost:api-gateway:8083
INVENTORY	n/a (1)	(1)	UP (1) - localhost:inventory:8081
ORDER	n/a (1)	(1)	UP (1) - localhost:order:8084
PAYMENT	n/a (1)	(1)	UP (1) - localhost:payment:8085
PRODUCT	n/a (1)	(1)	UP (1) - localhost:product:8080
REVIEW	n/a (1)	(1)	UP (1) - localhost:review:8087

General Info

Name	Value
total-avail-memory	108mb
num-of-cpus	12
current-memory-usage	65mb (60%)
server-uptime	00:22
registered-replicas	
unavailable-replicas	

Figure 4.9 Netflix Eureka interface

4.10. API gateway

API gateway is one of the essential parts in the microservices architecture, acting as a centralized entry point. It simplifies access for client applications by providing a single-entry point. This way, the clients do not need to be aware of the individual microservices or their locations, since they only interact with the API gateway. [1]

When the client-side makes the request to the server-side, the request is sent to the main entry point of the application, the API gateway. Based on the route predicates, the gateway directs the request to the needed microservice. This behaviour can be seen on Figure 16.

API gateway is implemented as a separate module, with the usage of *Spring Cloud Gateway*. It provides everything necessary for building an API Gateway in Spring Boot application. The dependency for it can be seen here:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

The gateway routes configuration lies in the application properties of the API gateway module. The API gateway configuration for web shop routes can be seen in code (Code 4.18).

```
cloud:
  gateway:
    discovery:
      locator:
        enabled: true
        lower-case-service-id: true
    routes:
      - id: product
        uri: lb://product
        predicates:
          - Path=/api/products/**, /api/selects/**,
/images/**
      - id: payment
        uri: lb://payment
        predicates:
          - Path=/api/payment/**
      - id: inventory
        uri: lb://inventory
        predicates:
          - Path=/api/inventory/**
      - id: order
        uri: lb://order
        predicates:
          - Path=/api/order/**
      - id: review
        uri: lb://review
        predicates:
          - Path=/api/reviews/**
      - id: assistant
        uri: http://localhost:8088
        predicates:
          - Path=/api/assistant/**
      - id: eureka
        uri: http://localhost:8761
        predicates:
          - Path=/eureka/web
        filters:
          - SetPath=/
```

Code 4.18 Web shop routes gateway configuration

Spring Cloud Gateway routes define how incoming requests are handled and forwarded within the microservices architecture. Each route contains key components like ID, URI, predicates, and filters.

URI specifies the destination addresses where the request is forwarded after it matches a route. When the URI contains for example “lb://product” it instructs the gateway to use load balancing for a service named “product”. When the request matches the route predicate, the gateway employs the configured service discovery server to locate the instances of the product service. To pick one of the available instances, the gateway load balancer uses one of the algorithms to distribute incoming requests across the discovered product service instances. This ensures that the traffic is dispersed evenly along available instances of the service, promoting availability and resilience of the services. Such flow can be seen on the diagram (Figure 4.10).

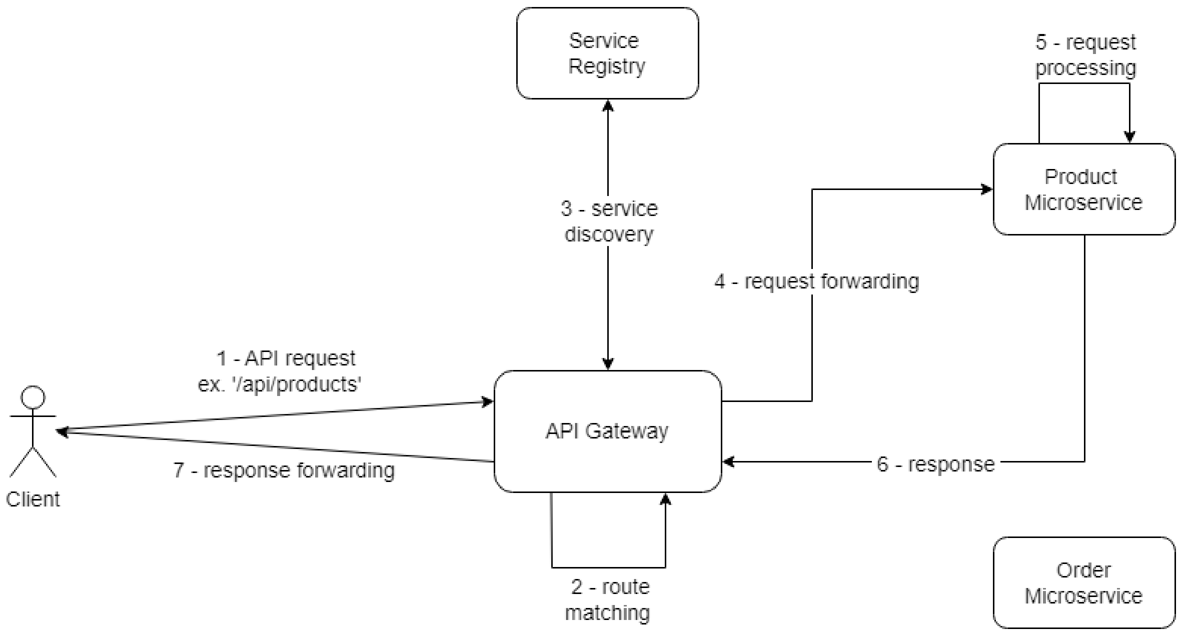


Figure 4.10 API gateway flow

To enable service discovery for the API gateway, a property *discovery-locator* of the gateway needs to be set to *true*, which can be seen in configuration above.

4.11.Security

Since the API gateway is a main entry point for the server-side application from the client-side, it can act as a centralized security point of the application.

To implement the authentication and authorization of the web shop application, API gateway leverages *Keycloak*, an open-source identity and access management (IAM) solution. It provides user federation and management, strong authentication, authorization, and more. Keycloak leverages industry-standard protocols like OAuth 2.0 and OpenID Connect, ensuring compatibility with various applications and frameworks. [32]

OAuth 2.0 is an authorization protocol that enables applications (clients) to obtain limited access to user accounts on a resource server, allowing users to grant access to their data without sharing their credentials directly with the requesting application. The authorization server (Keycloak) verifies user authorization and issues access tokens. [33]

OpenID Connect (OIDC) is an authentication protocol that builds on top of the OAuth 2.0. It extends it by defining a standard to exchange an access token for information about the user, delivered in a form of JSON Web Token (JWT). With OIDC, an application can retrieve user information without needing separate API calls. [34]

To manage a set of users, Keycloak uses realms. It manages a set of users, credentials, and roles. Each realm acts independently and manages its own set, and users can only log in into realms they are associated with. Client is any application or service that wants to leverage Keycloak for user authentication. They rely on OAuth 2.0 flows to request access tokens from Keycloak, verifying the user. [35] The web shop application defines one realm in Keycloak, responsible for managing its set of users. Regarding the client, it requires one client-side Keycloak client for handling user authentication.

To leverage Keycloak, the frontend Angular application uses the `angular-oauth2-oidc` library. [36] It simplifies implementing secure authentication in Angular applications, by handling OAuth 2.0 and OIDC flows with Keycloak or other providers.

```
export const authCodeFlowConfig: AuthConfig = {
        issuer: 'http://localhost:8181/realms/webshop',
        tokenEndpoint:
'http://localhost:8181/realms/webshop/protocol/openid-connect/token',
        redirectUri: 'http://localhost:4200',
        clientId: 'webshop-frontend',
        responseType: 'code',
        scope: 'openid profile',
        showDebugInformation: true,
};

...

function initializeOAuth(oauthService: OAuthService): Promise<void> {
    return new Promise((resolve) => {
            oauthService.configure(authCodeFlowConfig);
            oauthService.setupAutomaticSilentRefresh();
            oauthService.loadDiscoveryDocumentAndTryLogin()
                .then(() => resolve());
    });
}
```

Code 4.19 Angular OAuth initialization

The initialization of OAuth in Angular can be seen in code (Code 4.19). It sets the configuration for the OAuth service, specifying things like Keycloak token endpoint, client ID, realm, redirect URI for redirecting the user back to the application, etc. The *setupAutomaticSilentRefresh()* enables automatic background refresh of access tokens, ensuring the application has valid tokens to access user data. The *loadDiscoveryDocumentAndTryLogin()* checks the provided configuration and attempts to renew any existing tokens without prompting the user.

To attach the access token to every HTTP request from the client-side, Angular application defines an HTTP interceptor for custom handling every request going from the client, which can be seen in code (Code 4.20).

```
export const authInterceptor: HttpInterceptorFn = (req, next) => {
    const oAuthService = inject(OAuthService);
    const authToken = oAuthService.getAccessToken();

    // Clone the request and add the authorization header
    if (oAuthService.hasValidAccessToken()) {
        const authReq = req.clone({
                setHeaders: {
                    Authorization: `Bearer ${authToken}`
                }
        });

        // Pass the cloned request with the updated header to the
next handler
        return next(authReq);
    }

    return next(req);

};
```

Code 4.20 Angular HTTP interceptor

The whole authentication and authorization flow of the web shop application can be seen in diagram (Figure 4.11). Components involved are the user, client-side application, server-side with the microservices and API gateway, and Keycloak server.

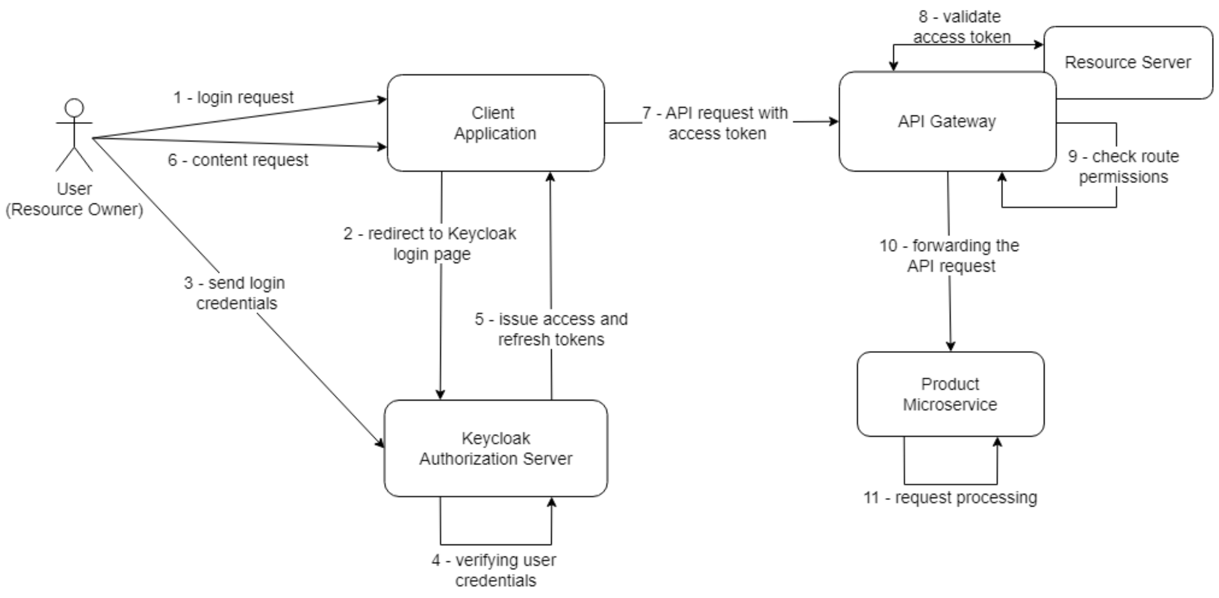

8 - validate access token
Resource Server
1 - login request
Client Application
7 - API request with access token
API Gateway
6 - content request
User (Resource Owner)
9 - check route permissions
2 - redirect to Keycloak login page
10 - forwarding the API request
3 - send login credentials
5 - issue access and refresh tokens
Product Microservice
Keycloak Authorization Server
11 - request processing
4 - verifying user credentials

Figure 4.11 Authorization and authentication flow

Firstly, to access a specific resource, the user initiates authentication through the login request to the Client application. The client redirects the user to the Keycloak authentication page, where the user inserts the login credentials. Keycloak server verifies the credentials, and if correct, issues access and refresh JWT tokens (OAuth 2.0 flow). JSON Web Token (JWT) is an open standard that defines a self-contained way to securely transmit information between parties. Since it is digitally signed, it ensures the integrity of the token. Each JWT contains a payload: claims set, which is data than can include user information, token expiration, etc. [37] An access token represents a short-lived credential issued by an authorization server, in this case Keycloak, after successful user authentication. It allows the client to access protected resources for a limited time. Along with the access token, also provides claims set, containing user information like email, name, etc. (OIDC) When the access token expires, the client application sends the refresh token to the authorization server, to obtain a new access token. The client sends the access token with each request, through the `Authorization: Bearer <access_token>` header attribute.

An example of the web shop application decoded JWT access token, with belonging user information and roles claims set, can be seen in code (Code 4.21).

```
{
    "exp": 1718393619,
    "iat": 1718393319,
    "jti": "d15d611d-c3d9-4513-b550-a7affbf99392",
    "iss": "http://localhost:8181/realms/webshop",
    "aud": "account",
    "sub": "93516232-2fae-4169-8425-2ef9fc3ce333",
    "typ": "Bearer",
    "azp": "webshop-frontend",
    "session_state": "9742c072-e9de-4c2e-b480-ec9fa0515086",
    "acr": "1",
    "allowed-origins": [
        "http://localhost:4200"
    ],
    "realm_access": {
        "roles": [
            "default-roles-webshop",
            "offline_access",
            "uma_authorization"
        ]
    },
    "resource_access": {
        "account": {
            "roles": [
                "manage-account",
                "manage-account-links",
                "view-profile"
            ]
        }
    },
    "scope": "profile email",
    "sid": "9742c072-e9de-4c2e-b480-ec9fa0515086",
    "email_verified": true,
    "name": "Dalijo Vorkapić",
    "preferred_username": "dadodv",
    "given_name": "Dalijo",
    "family_name": "Vorkapić",
    "email": "vorkapicdalijo@gmail.com"
}
```

Code 4.21 JWT access token example

The API gateway intercepts the request from the client application. Firstly, it validates the access token, ensuring it is properly formatted and that it did not expire. Then, it extracts the data from the JWT token and reads the user roles for authorization. If the requested API route require authentication and specific role authorization, like administrator, API gateway checks the user roles from the token and grants access to the requested microservice if the user is authorized for it. Otherwise, if the token is invalid or the user is unauthorized, the gateway returns `HTTP 401 Unauthorized` code. Since API gateway acts as the main gate to all the resources (microservices) in application, to implement access token validation, and route and roles permission control, the gateway needs to be set as the oauth2 resource server, setting this dependency:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

The API gateway configuration properties for the oauth2 resource server can be seen in this code:

```
spring:
  application:
    name: api-gateway
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8181/realms/webshop
          jwk-set-uri:
http://localhost:8181/realms/webshop/protocol/openid-connect/certs
```

The *oauth2* section specifies the oauth2 settings. *resourceserver* configures Spring Security to a resource server. The *jwt* section sets how the resource server will validate JWT tokens. The *issuer-uri* property is set to the Keycloak on realm webshop, meaning that the gateway will verify that a received JWT comes from the correct issuer. Through the *jwk-set-uri* API gateway will verify the digital signatures of the access tokens. The gateway filter chain configuration for permission check can be seen in code (Code 4.22).

```
@Bean
public SecurityWebFilterChain filterChain(ServerHttpSecurity http) throws
Exception {
    http    .csrf(ServerHttpSecurity.CsrfSpec::disable)
            .cors().configurationSource(corsConfigurationSource())
            .and()
            .authorizeExchange()
                    .pathMatchers("/api/products/**").permitAll()

.pathMatchers("/api/products/add").hasAuthority("ROLE_ADMIN")

.pathMatchers("/api/products/update").hasAuthority("ROLE_ADMIN")

.pathMatchers("/api/products/delete").hasAuthority("ROLE_ADMIN")
                    .pathMatchers("/api/inventory/**").permitAll()
                    .pathMatchers("/images/**").permitAll()
                    .pathMatchers("/api/order/**").authenticated()
                    .pathMatchers("api/payment/**").authenticated()
                    .anyExchange().authenticated()
            .and()
            .oauth2ResourceServer((oauth2) -> oauth2.jwt(
                    jwt ->
jwt.jwtAuthenticationConverter(customJwtConverter())
                    )
            );
    return http.build();
}
```

Code 4.22 API gateway routes permission check

After the access token validation if the token is valid and the user has the necessary permissions for the resource microservice, the API gateway forwards the request to the appropriate microservice. This service processes the request and returns response back to the client.

5. Client-side

The client-side of the Fragrance.hr web shop application is built using Angular. It provides a powerful framework for building dynamic and interactive Single Page Applications (SPAs), which creates a smooth and responsive user experience by removing full page reloads.

Each user interface or view can be defined by using Angular component. Its template consists of an HTML template, styles, and TypeScript class. TypeScript class defines the component's behaviour and logic. It is written in TypeScript, a superset of JavaScript, a high-level programming language for forming interactivity and dynamic behaviour on web applications. An HTML template defines the structure and appearance of the component's elements. It uses HTML syntax with special Angular directives and bindings to integrate with the TypeScript code. Lastly, CSS styles are defined to set the styling and look of the HTML components. [38] An example of products component TypeScript class within the web shop application can be seen here:

```
@Component({
        selector: 'app-products',
        standalone: true,
        imports: [
        ...
        ],
        templateUrl: './products.component.html',
        styleUrl: './products.component.scss',
        })
export class ProductsComponent implements OnInit {
    ...
}
```

Through *@Component* decorator, *ProductsComponent* class is defined as an Angular component. By providing *templateUrl* and *styleUrl*, the product component specifies where HTML template and styling files are located.

Other crucial part of an Angular applications are services. Angular service is a reusable chunk of code, a backbone, designed to handle specific functionality that is separated from the components. It usually handles the data access and business logic of the application. When multiple components are requiring the same data or functionality, one service can be created to provide the needing functionalities to the components. Any of the API HTTP calls made from the application are handled through the services, making a request, fetching and providing response data. [38] Service responsible for handling the product logic and data access is shown in code (Code 5.1).

```
@Injectable({
        providedIn: 'root'
        })
export class ProductService {
    constructor(private http: HttpClient, private oauthService:
OAuthService) {}

    public getProducts(brandId?: number, categoryId?: number, typeId?:
number): Observable<Product[]> {
        let params = new HttpParams();
        if(brandId)
            params = params.set('brandId', brandId);
        if(categoryId)
            params = params.set('categoryId', categoryId);
        if(typeId)
            params = params.set('typeId', typeId);
        return
this.http.get<Product[]>(environment.baseUrl+'api/products', { params });
    }
    public getProductById(id: number): Observable<Product> {
        return
this.http.get<Product>(environment.baseUrl+`api/products/${id}`);
    }
    public addProduct(formData: FormData): Observable<any> {
        return this.http.post(environment.baseUrl+'api/products/add',
                formData
        );
    }
    public updateProduct(formData: FormData, productId: number):
Observable<any> {
        return
this.http.post(environment.baseUrl+`api/products/update/${productId}`,
        formData
        );
    }
    public deleteProduct(productId: number) {
        return
this.http.delete(environment.baseUrl+`api/products/delete/${productId}`);
    }
}
```

Code 5.1 Angular Product service

Through @injectable decorator, a class can be provided as a dependency and injected into other components and services. It essentially creates a service that can be instantiated and used throughout the application.

The Fragrance.hr web shop uses the next Angular services, each handling a separate responsibility, functionality, and making API calls to the appropriate microservice on the server-side:

- Auth Service

- Product Service
- Cart Service
- Review Service
- Payment Service
- Order Service
- Chat Service
- Select Service (for forms)

It is already stated that a user interface, or a page view, can be defined through the Angular component. Since Angular is a single page application, when a user navigates to the specific page route, Angular needs to intercept the route and use the route configuration to determine the appropriate component to display. Essentially, routes act as instructions for Angular on which component to render based on the current route. A list of routes used within this application can be seen in code (Code 5.2).

```
export const routes: Routes = [
    { path: 'products', component: ProductsComponent },
    { path: 'products/:categoryId', component: ProductsComponent },
    { path: 'product-details/:id', component: ProductDetailsComponent },
    { path: '', component: HomeComponent },
    { path: 'login', component: LoginComponent },
    { path: 'about-us', component: AboutUsComponent },
    { path: 'cart', component: CartComponent },
    { path: 'orders', component: OrdersComponent, canActivate:[AuthGuard]
},
    { path: 'purchase', component: PurchaseComponent,
canActivate:[AuthGuard] },
    { path: 'purchase/capture', component: PurchaseComponent,
canActivate:[AuthGuard] },
    { path: 'purchase/cancel', component: PurchaseComponent,
canActivate:[AuthGuard] },
    { path: '', redirectTo: '/home', pathMatch: 'full' },
    { path: '**', component: HomeComponent },
];
```

Code 5.2 Angular routes definition

The *canActivate* tag represents an interface for a route guard. Route guards are functions that control whether a user can access a specific route within the application. The configuration can be seen in code (Code 5.3).

```
@Injectable({
        providedIn: 'root'
        })
export class AuthGuard implements CanActivate {
    constructor(private oAuthService: OAuthService,) {}
    canActivate(
            route: ActivatedRouteSnapshot,
            state: RouterStateSnapshot): Observable<boolean | UrlTree> |
Promise<boolean | UrlTree> | boolean | UrlTree {
        var isAuthenticated = this.oAuthService.hasValidAccessToken();
        if(!isAuthenticated) {
            this.oAuthService.initLoginFlow();
        }
        return isAuthenticated;
    }
}
```

Code 5.3 Angular auth guard

Since Angular integrates and implements the OAuth 2.0 service, the auth guard is based around the validity of the access token received upon the user authentication.

5.1. User interface

This section covers the final look of the Fragrance.hr web shop application. The Angular application serving the user interface, and the microservices on the server-side being a foundation and providing and managing all the data and business logic for the application.

To enhance user experience and overall look of the user interface, this Angular application uses the Angular Material and bootstrap libraries. Angular Material provides a components library, offering a choice between many prebuilt components and functionalities, like buttons, tables, navigations, etc. Bootstrap server as the utility library, providing tools for building and designing responsive websites.

On initial web-shop page load, the home screen is loaded, visible on figure (Figure 5.1).

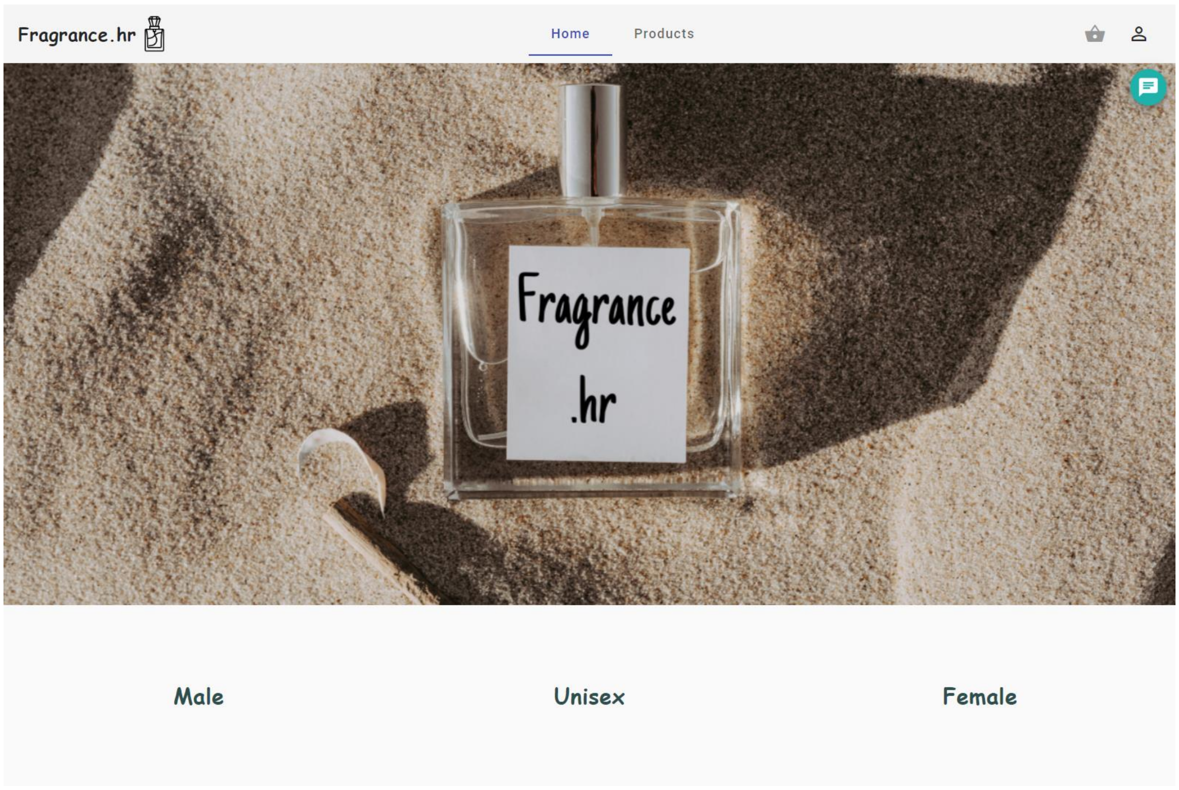
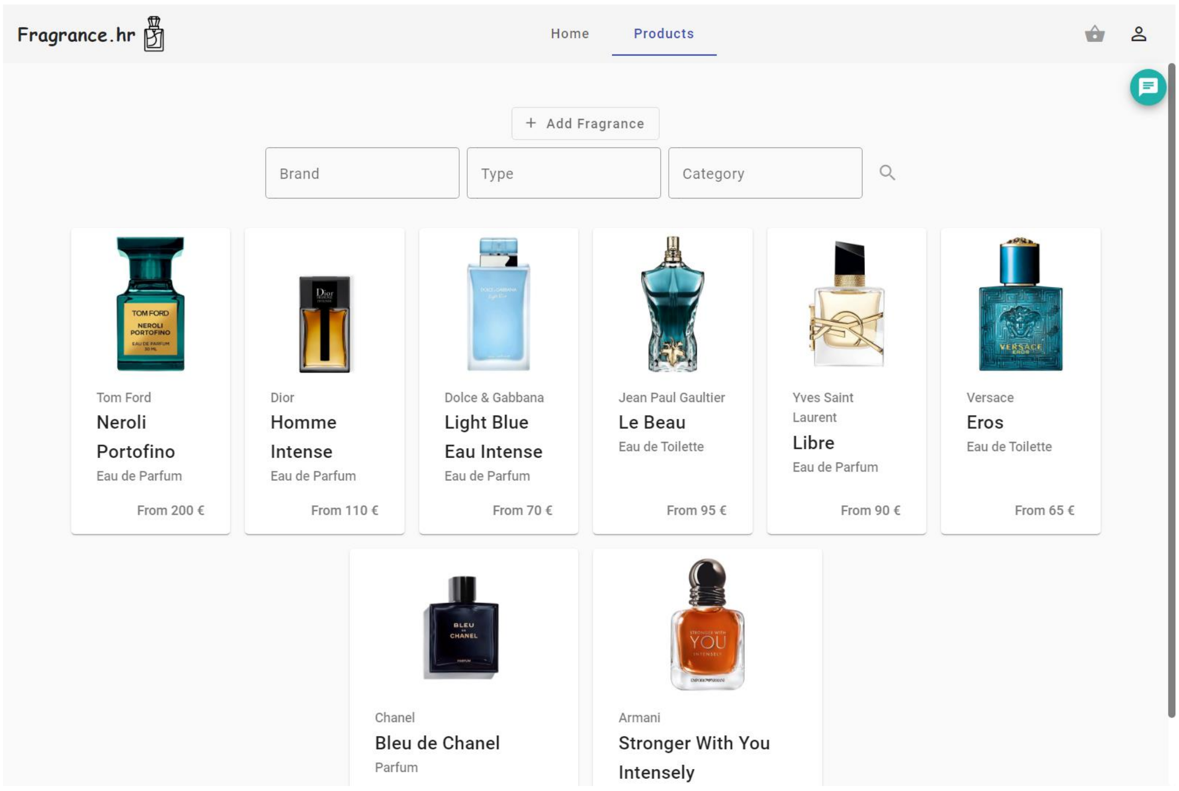


Figure 5.1 Web shop home page

Additionally, a header can be seen, containing web shop name and logo, page navigation, cart button, and account button. Header component will be always present on each page screen, making its functionalities available through the whole application. The home component contains the main menu section, where the user can select which products (fragrances) it wants to be listed, based on the category.

By selecting a category, or through the header products navigation, the user is routed to the products listing view. Such view can be seen on figure (Figure 5.2).



Tom Ford
Neroli Portofino
Eau de Parfum
From 200 €

Dior
Homme Intense
Eau de Parfum
From 110 €

Dolce & Gabbana
Light Blue Eau Intense
Eau de Parfum
From 70 €

Jean Paul Gaultier
Le Beau
Eau de Toilette
From 95 €

Yves Saint Laurent
Libre
Eau de Parfum
From 90 €

Versace
Eros
Eau de Toilette
From 65 €

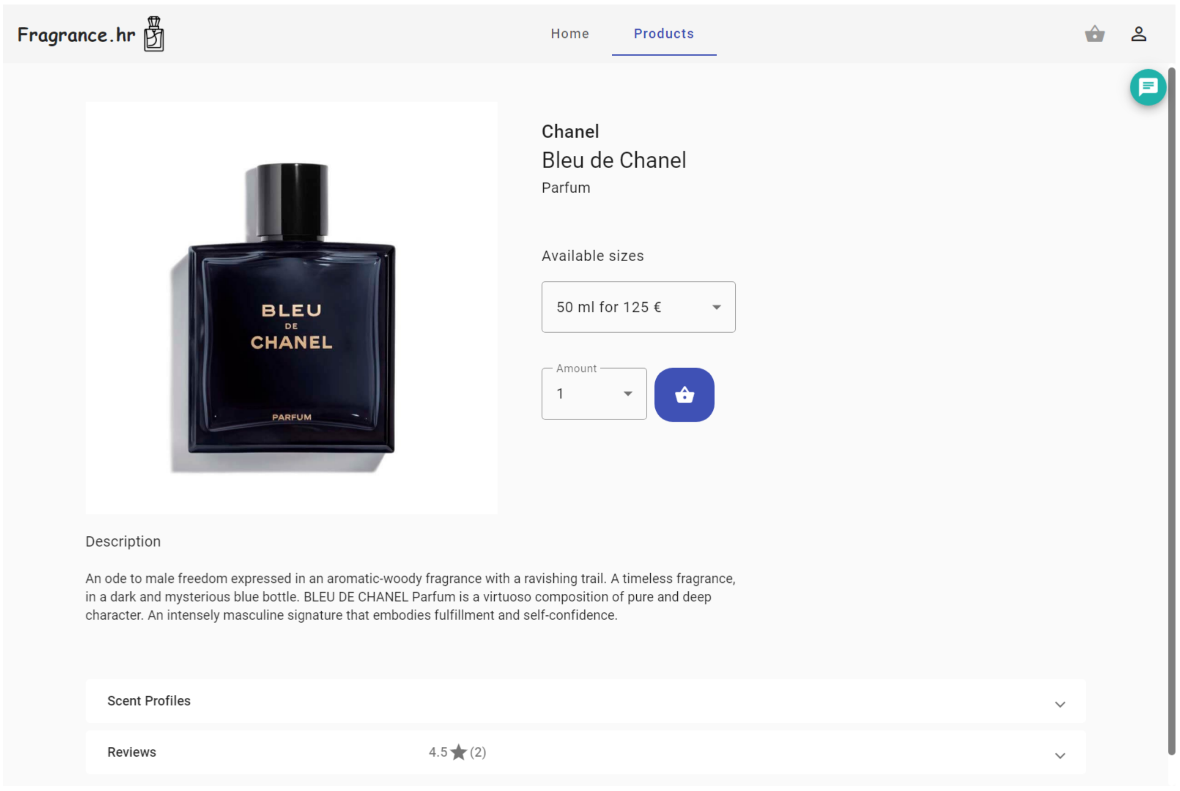
Chanel
Bleu de Chanel
Parfum

Armani
Stronger With You Intensely

Figure 5.2 Products listing page

The main purpose of this view is to show a list of available products (fragrances) to the user (customer). Each product listed contains name, brand, type, and lowest available price, based on the sizes available. Additionally, each listed product contains one cover image representing it. To filter the listed products, user has three filter options available. It can filter products based on the brand, type, or category of the product. Each filter comes in a form of an autocomplete dropdown menu, where on text insert, the user gets offered the filtered menu of options to select. The page visitor does not need to be authenticated to visit this view. If the authenticated user has the administrator role, an option to add a new fragrance (product) will be available to it.

When a product from the listing is selected, the user is routed to the details page of the selected product, which can be seen on figure (Figure 5.3).



Chanel

Bleu de Chanel

Parfum

Available sizes

50 ml for 125 €

Amount

1

Description

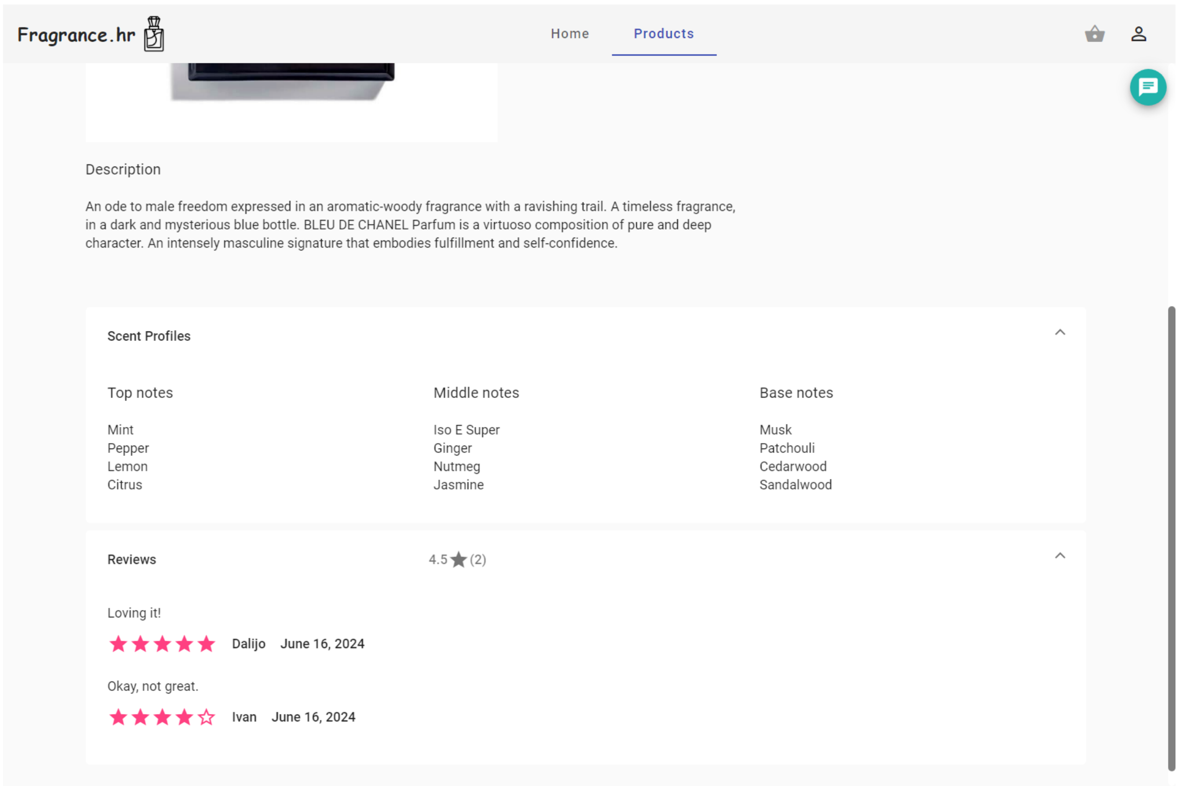
An ode to male freedom expressed in an aromatic-woody fragrance with a ravishing trail. A timeless fragrance, in a dark and mysterious blue bottle. BLEU DE CHANEL Parfum is a virtuoso composition of pure and deep character. An intensely masculine signature that embodies fulfillment and self-confidence.

Scent Profiles

Reviews 4.5 ★ (2)

Figure 5.3 Product details page

As the name suggests, the details page serves as the main place to find all the information regarding the product. It contains the fields already seen in the listing page, and additionally, it contains the fragrance description, available sizes selection, and the functionality to select the product amount and add it to the cart. Additionally, two dropdown drawers can be seen, one for displaying scent profiles of the fragrance, and the other for displaying user reviews. That view can be seen on figure (Figure 5.4).



Description

An ode to male freedom expressed in an aromatic-woody fragrance with a ravishing trail. A timeless fragrance, in a dark and mysterious blue bottle. BLEU DE CHANEL Parfum is a virtuoso composition of pure and deep character. An intensely masculine signature that embodies fulfillment and self-confidence.

Scent Profiles

Top notes	Middle notes	Base notes
Mint	Iso E Super	Musk
Pepper	Ginger	Patchouli
Lemon	Nutmeg	Cedarwood
Citrus	Jasmine	Sandalwood

Reviews 4.5 ★ (2)

Loving it!
★★★★★ Dalijo June 16, 2024

Okay, not great.
★★★★☆ Ivan June 16, 2024

Figure 5.4 Product reviews and scent profiles

The scent profiles drawer shows the notes associated with the product, spread on three note layers. Such information is one of the crucial aspects of each fragrance in general. Reviews drawer initially contains the average rating and number of comments, and by opening it, the actual reviews are shown. Each review contains the comment, star rating, name of the customer, and the date of review creation.

Once the products cart is filled with one or more products, the current user can see the quantity of items in the cart through the cart icon in the header. On click, the cart icon expands, showing the dropdown with the added fragrances. Such view can be seen on figure (Figure 5.5).

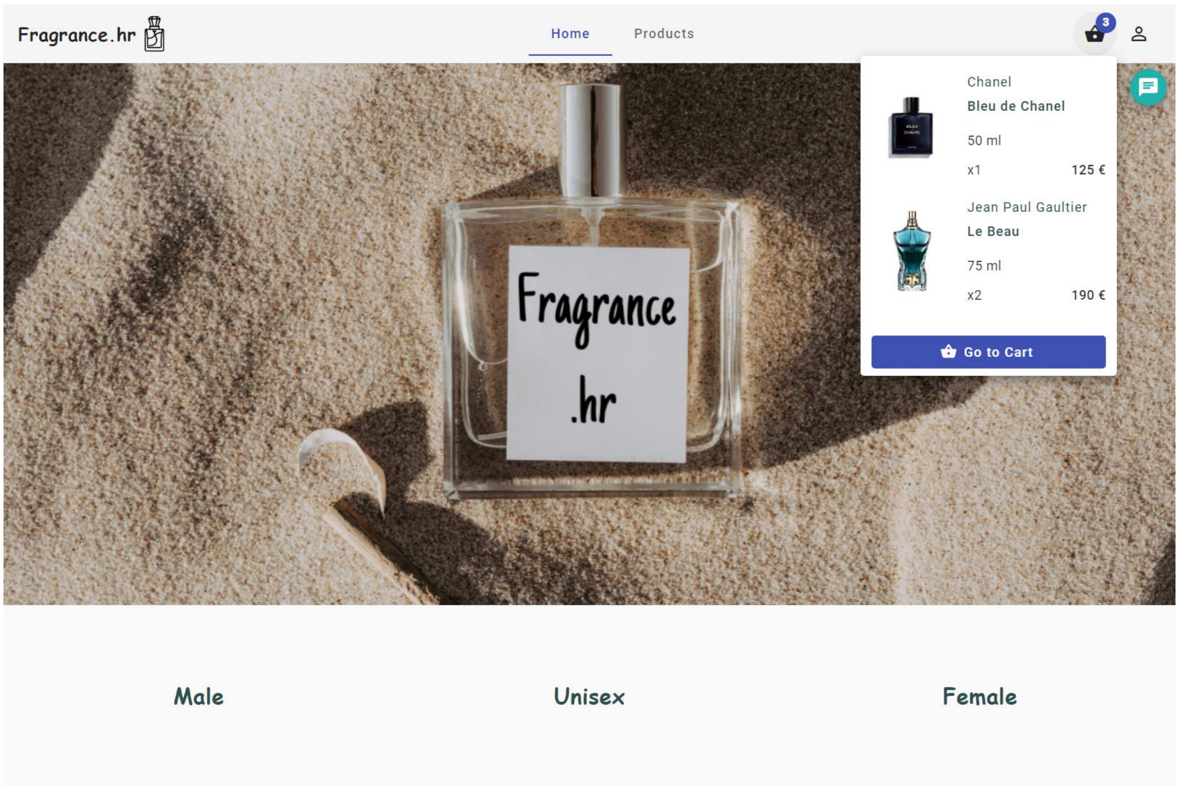
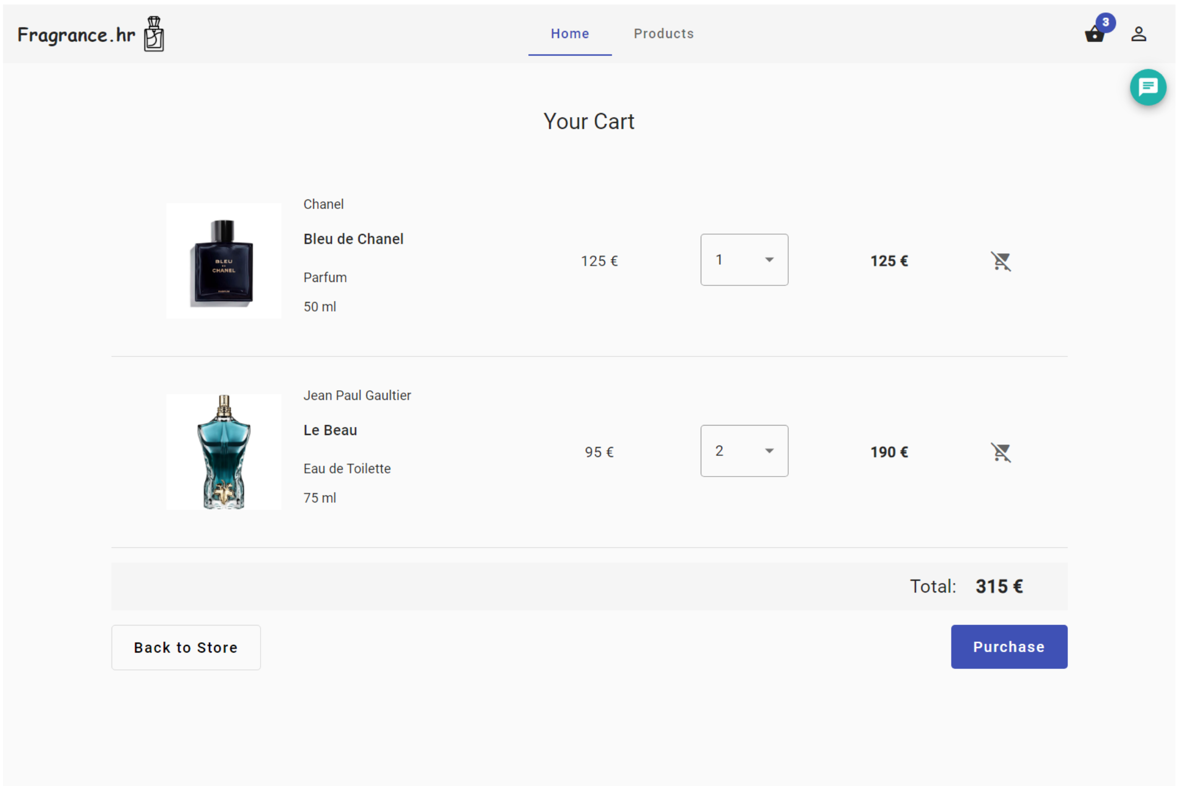


Figure 5.5 Cart dropdown list

Each product listed in cart dropdown menu contains information like brand, name, sizing, quantity selected, and price. On click of a product, the user is routed to that product's details page. By selecting "Go to Cart" option, the user is routed to the cart page view, seen on figure (Figure 5.6).


Fragrance.hr

Home Products

Your Cart

Chanel **Bleu de Chanel** Parfum 50 ml	125 €	1	**125 €**		
Jean Paul Gaultier **Le Beau** Eau de Toilette 75 ml	95 €	2	**190 €**		

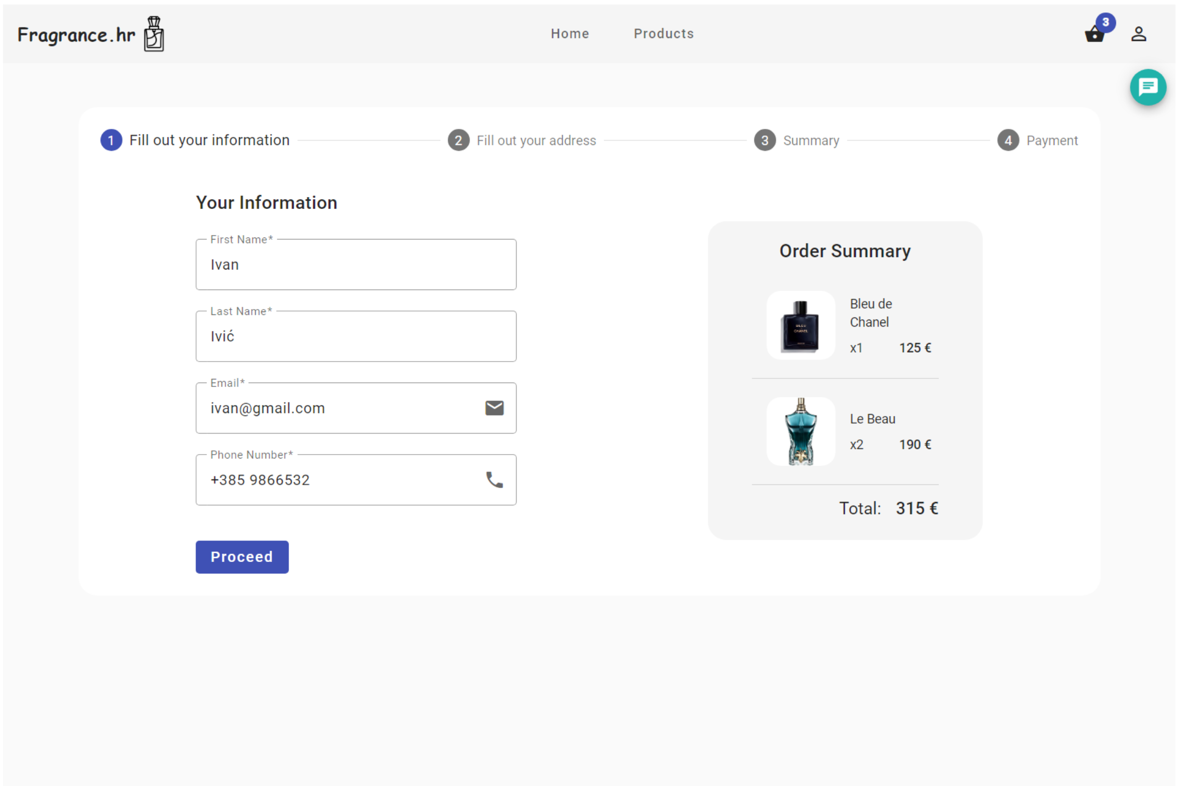
Total: **315 €**

Back to Store

Purchase

Figure 5.6 Cart page

On the cart page, the user can see the overview of fragrances added to the cart, and it can change the quantity of the products, or remove the product from the cart. Additionally, each product is checked for availability through the inventory microservice, and if it is not available, a message under the product will show, and the user will not be able to proceed to the purchase (checkout) view. If all the products with their quantities are available, on selecting the purchase option, customer is routed to the purchase view, seeable on figure (Figure 5.7).



Your Information

First Name*: Ivan

Last Name*: Ivić

Email*: ivan@gmail.com

Phone Number*: +385 9866532

Order Summary

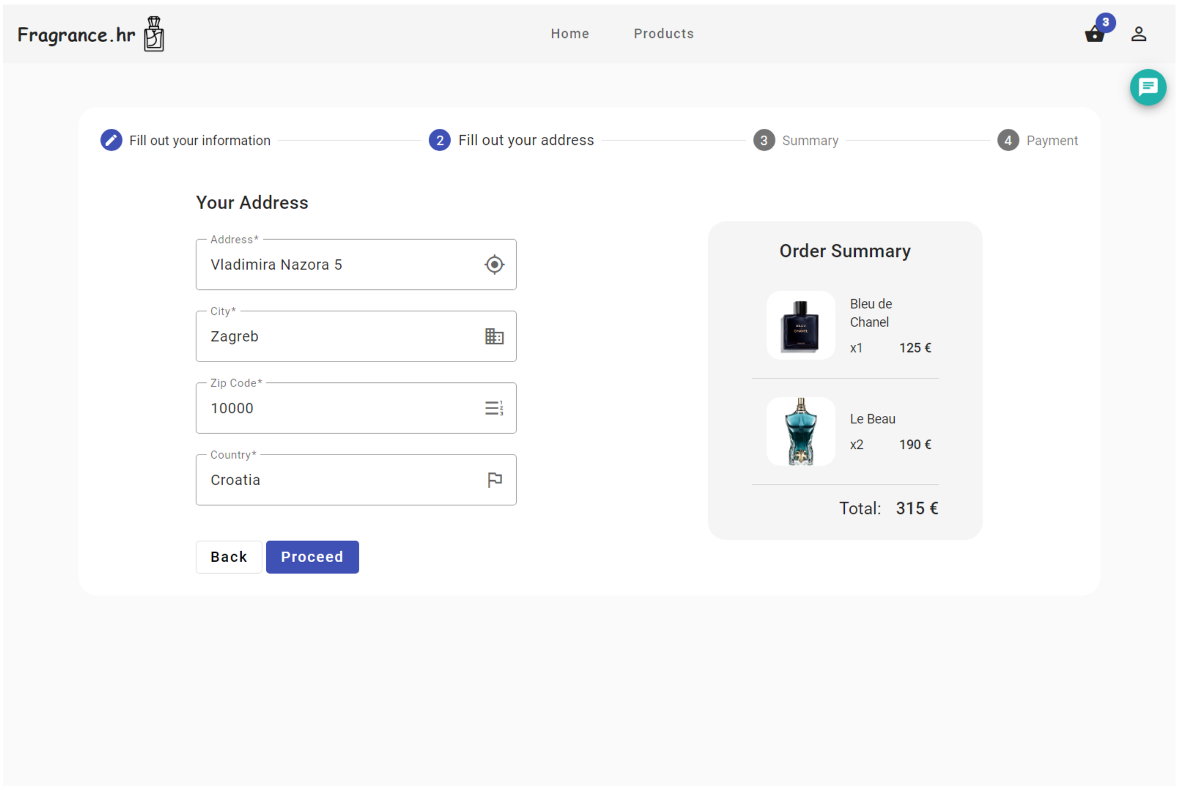
Bleu de Chanel	x1	125 €
Le Beau	x2	190 €

Total: 315 €

Figure 5.7 User information form for order

The checkout process is handled through the stepper flow. The stepper contains four steps, each handling a specific task. The first stepper contains a form for storing user information, such as first name, last name, email, and phone number. Since this view is accessible only for the authenticated users, some information gets filled automatically based on the account information stored during user registration. The second step contains a form for storing user address, with fields for address, city, postal code, and country. That step can be seen on figure (Figure 5.8). The third step offers a summary of information filled by the user from previous two steps, and it can be seen on figure (Figure 5.9). Since each step is returnable, the user can return to the previous steps and change up the data if needed. Additionally, the first, second, and last step offer the order summary view, for the customer to check up on the products selected for purchase.

The last step of the customer checkout process is the payment step, shown on figure (Figure 5.10).



Your Address

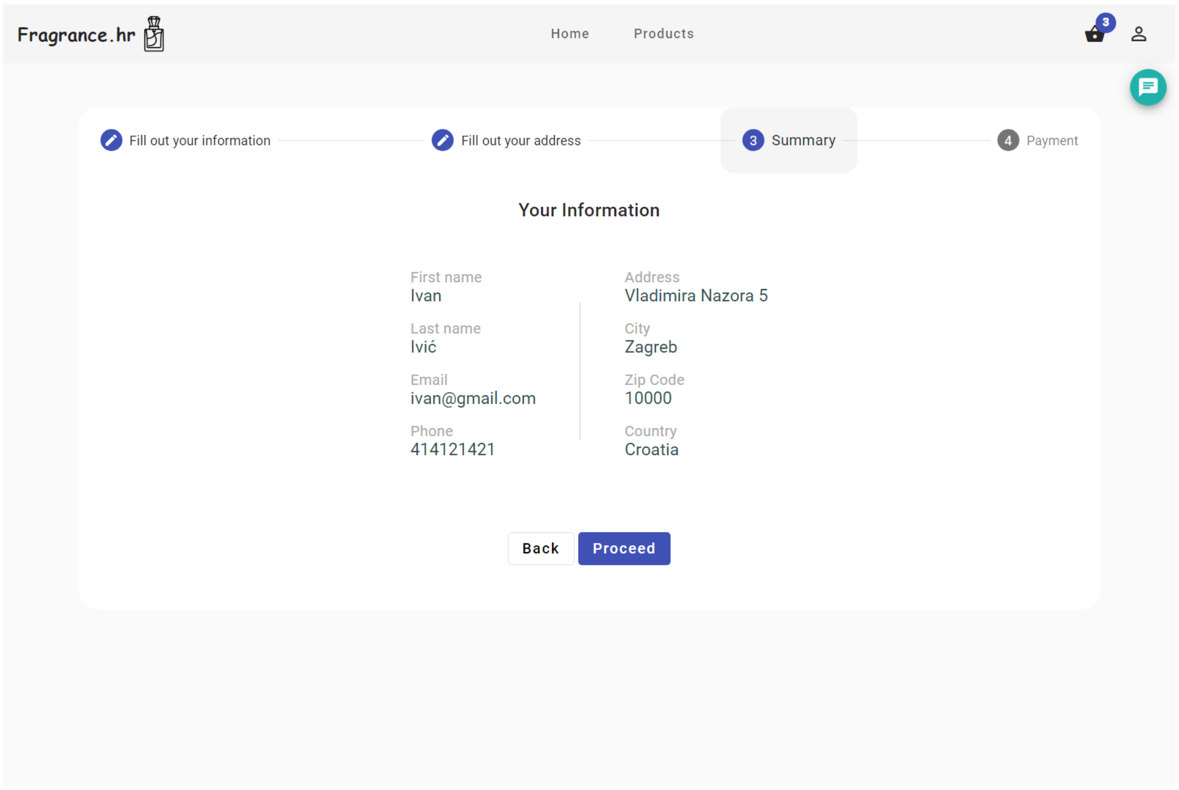
Address*	Vladimira Nazora 5
City*	Zagreb
Zip Code*	10000
Country*	Croatia

Order Summary

Bleu de Chanel	x1	125 €
Le Beau	x2	190 €

Total: 315 €

Figure 5.8 User address form for order



Your Information

First name	Ivan	Address	Vladimira Nazora 5
Last name	Ivić	City	Zagreb
Email	ivan@gmail.com	Zip Code	10000
Phone	414121421	Country	Croatia

Figure 5.9 User information and address summary

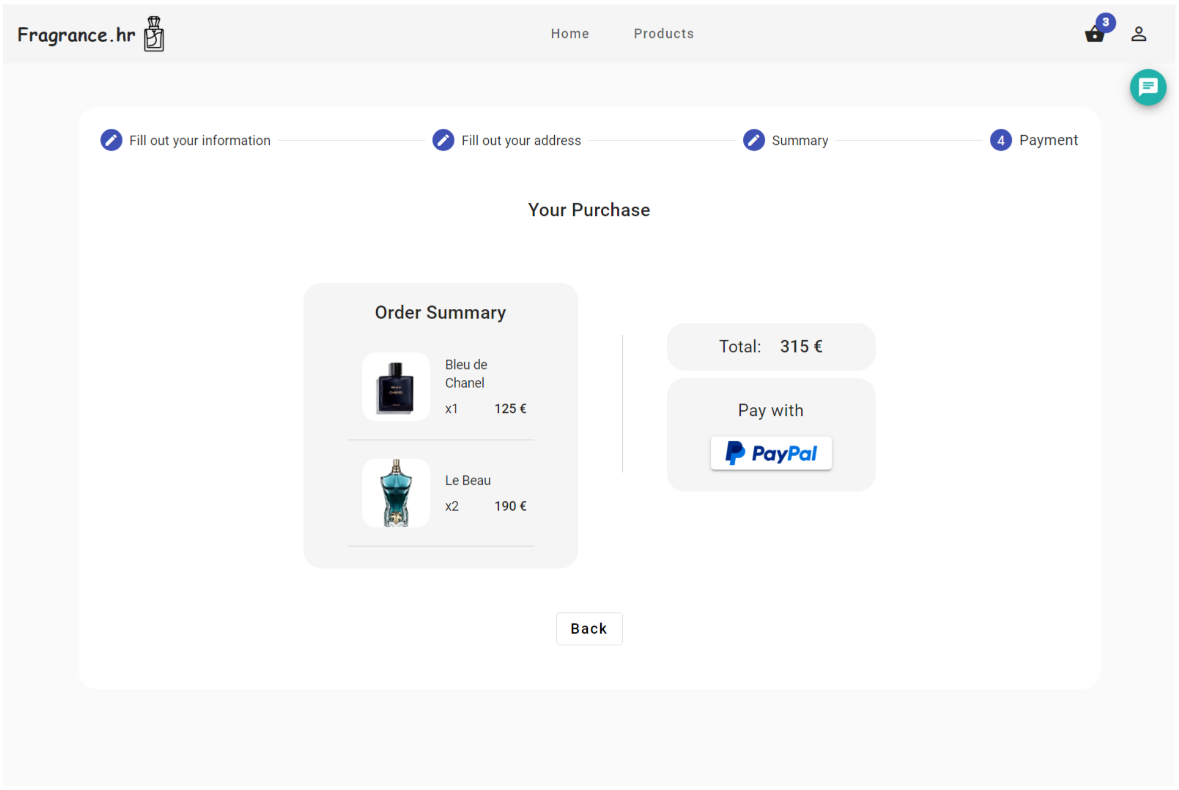
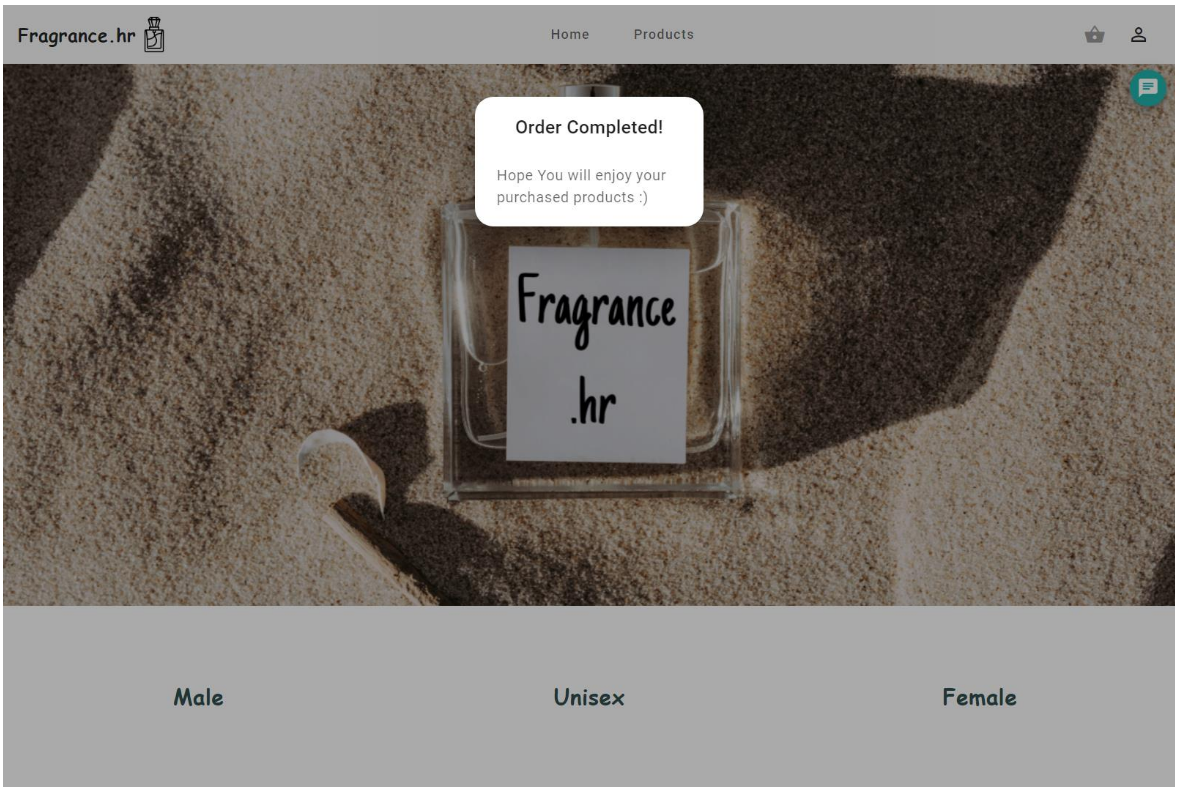


Figure 5.10 Order payment step

On the payment step, the user can select the option to purchase the fragrances and initiate a payment through the PayPal payment provider. On click, the user is redirected to the PayPal authentication screen. On successful authentication, the user can create a payment request through the PayPal interface, by selecting payment option. If the payment succeeds, the products purchase process is finished, resulting in an appropriate pop-up message seeable in figure (Figure 5.11). If the payment gets cancelled, the purchase process cancels.

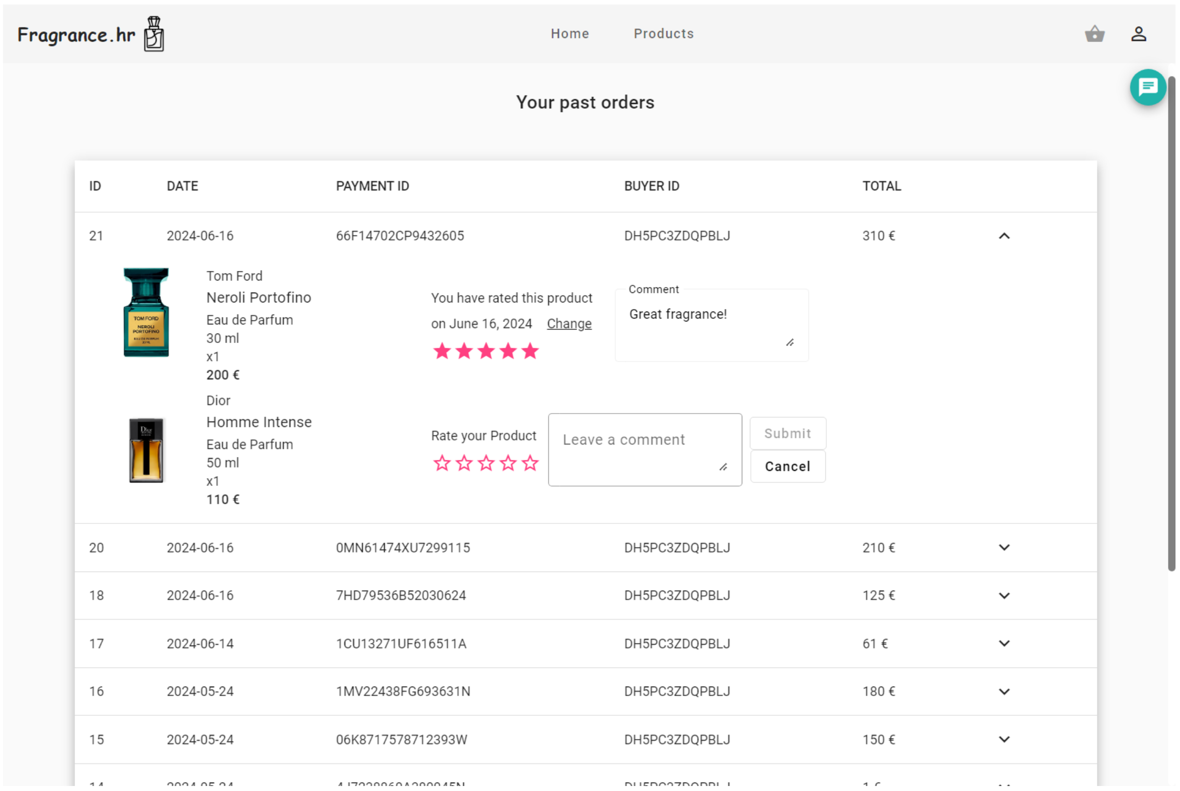


Order Completed!

Hope You will enjoy your purchased products :)

Figure 5.11 Order completed alert

By selecting the orders option from the account dropdown menu from the header, the authenticated user is directed to the past orders overview page, which can be seen on figure (Figure 5.12).



Your past orders

ID	DATE	PAYMENT ID	BUYER ID	TOTAL
21	2024-06-16	66F14702CP9432605	DH5PC3ZDQPBLJ	310 €
20	2024-06-16	0MN61474XU7299115	DH5PC3ZDQPBLJ	210 €
18	2024-06-16	7HD79536B52030624	DH5PC3ZDQPBLJ	125 €
17	2024-06-14	1CU13271UF616511A	DH5PC3ZDQPBLJ	61 €
16	2024-05-24	1MV22438FG693631N	DH5PC3ZDQPBLJ	180 €
15	2024-05-24	06K8717578712393W	DH5PC3ZDQPBLJ	150 €

Tom Ford
Neroli Portofino
Eau de Parfum
30 ml
x1
200 €

You have rated this product on June 16, 2024 Change

Comment
Great fragrance!

Dior
Homme Intense
Eau de Parfum
50 ml
x1
110 €

Rate your Product

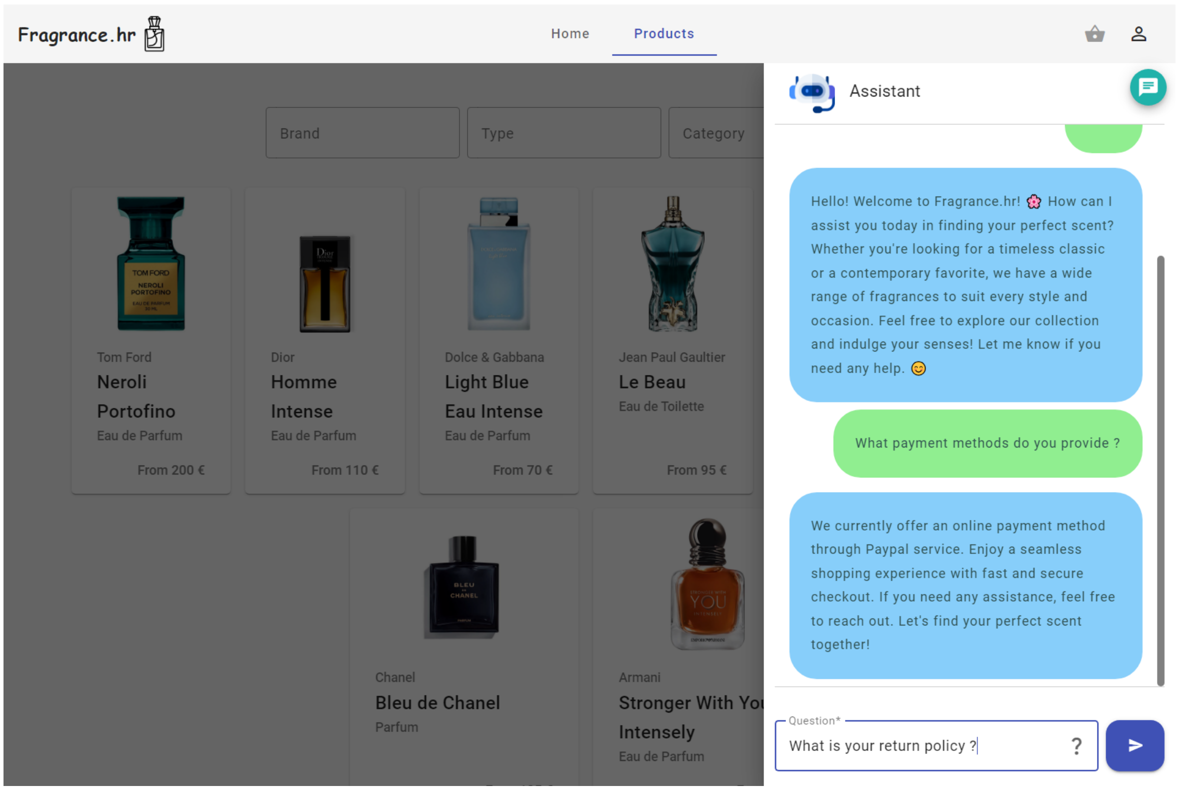
Leave a comment

Submit
Cancel

Figure 5.12 Orders listing and adding reviews

Each order contains information like date of order, payment, and buyer ID from PayPal, and total price paid. On order click, a dropdown drawer opens, displaying products purchased within that order. Additionally, user can leave a review on the purchased product, by leaving a comment and selecting number of stars (rating).

At any point throughout the application, a floating chat button is visible in the top right corner. On click, it displays the chat window, allowing the visiting users to chat to the AI chatbot of a Fragrance.hr web shop. Such window and chat can be seen on figure (Figure 5.13).



Hello! Welcome to Fragrance.hr! 🌸 How can I assist you today in finding your perfect scent? Whether you're looking for a timeless classic or a contemporary favorite, we have a wide range of fragrances to suit every style and occasion. Feel free to explore our collection and indulge your senses! Let me know if you need any help. 😊

What payment methods do you provide ?

We currently offer an online payment method through Paypal service. Enjoy a seamless shopping experience with fast and secure checkout. If you need any assistance, feel free to reach out. Let's find your perfect scent together!

What is your return policy ?

Figure 5.13 AI chatbot

Figure (Figure 5.14) shows the account dropdown menu for the user with administrator role.

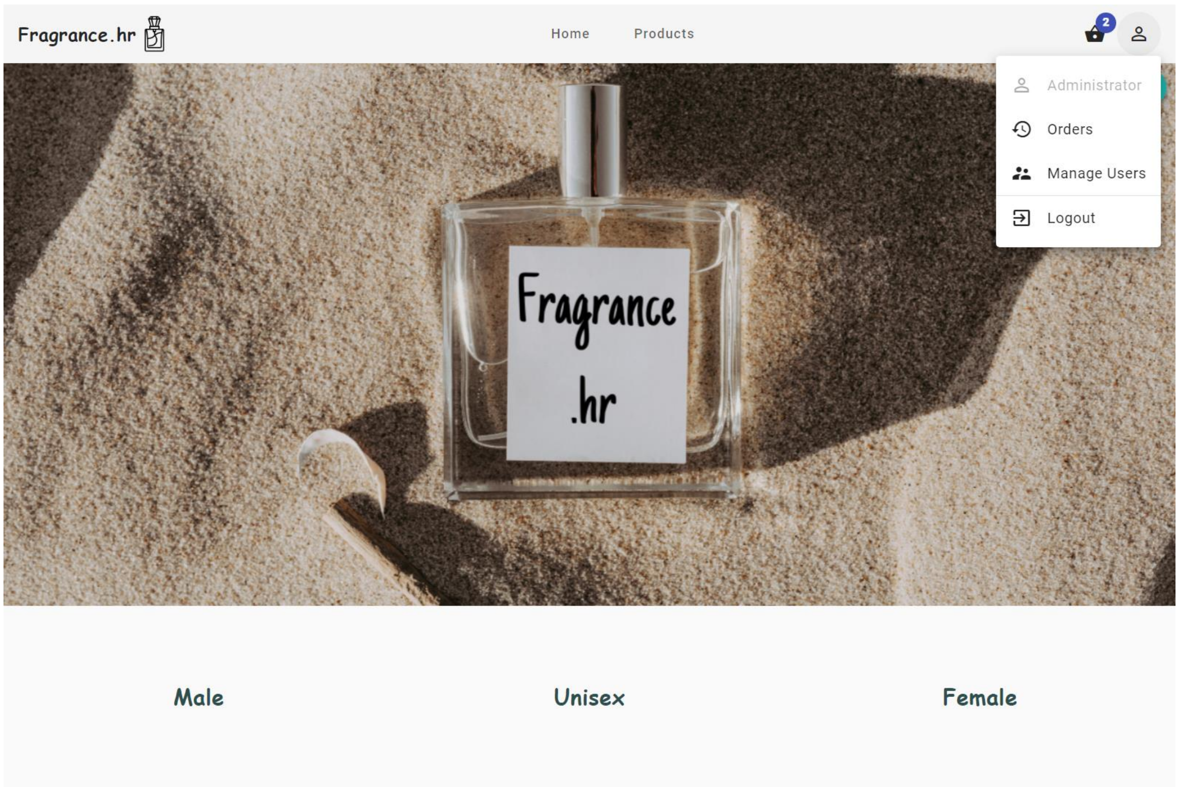
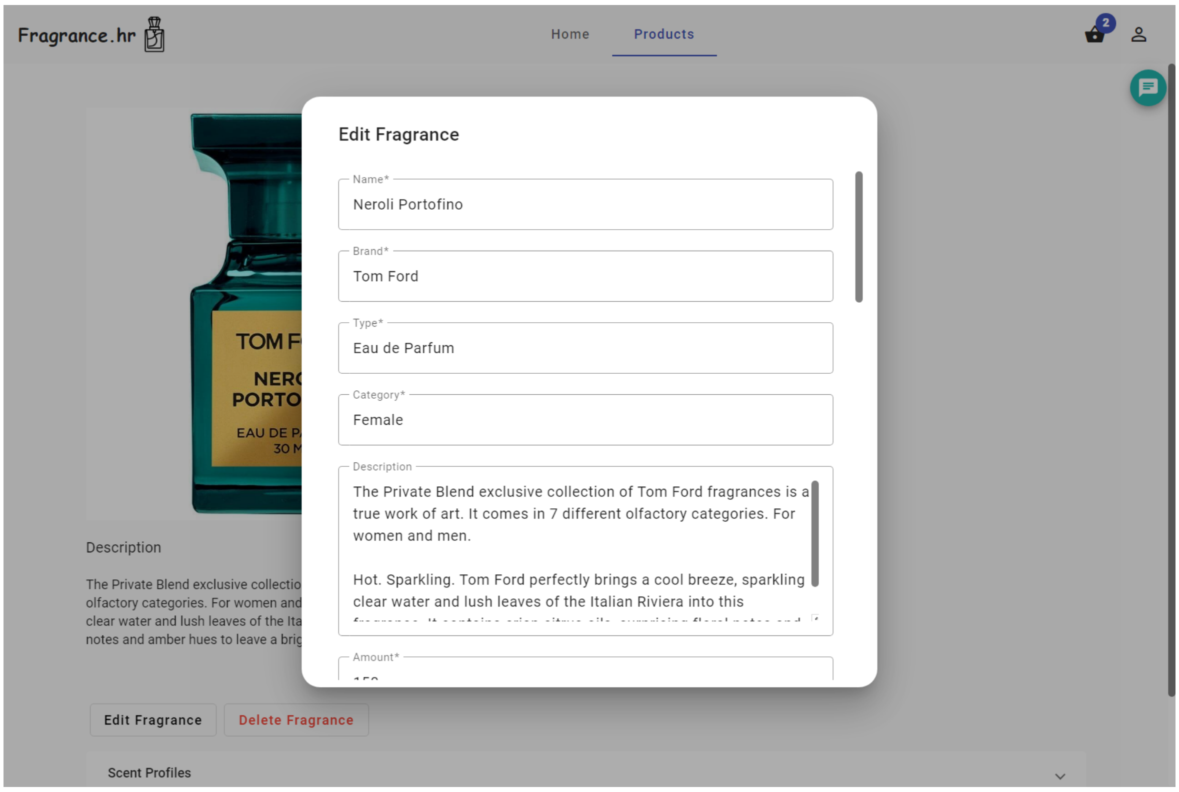


Figure 5.14 User (administrator) dropdown menu

As already mentioned, the administrator has the available option to add a new fragrance. Along with that, on the product details page, administrator gets offered two more options. The option to edit current fragrance, and to delete current fragrance.

On fragrance deletion, the administrator gets redirected to the fragrances listing, with a pop-up message informing about deletion.

By selecting the option to edit a fragrance, a dialog window is opened, with the form containing the input fields available for the edit. The inputs are filled with the current fragrance information, and on submit, the information is updated. The view of the edit form can be seen in figures (Figure 5.15, Figure 5.16, Figure 5.17). Each input needs to be filled to submit the form.



Edit Fragrance

Name*: Neroli Portofino

Brand*: Tom Ford

Type*: Eau de Parfum

Category*: Female

Description: The Private Blend exclusive collection of Tom Ford fragrances is a true work of art. It comes in 7 different olfactory categories. For women and men.

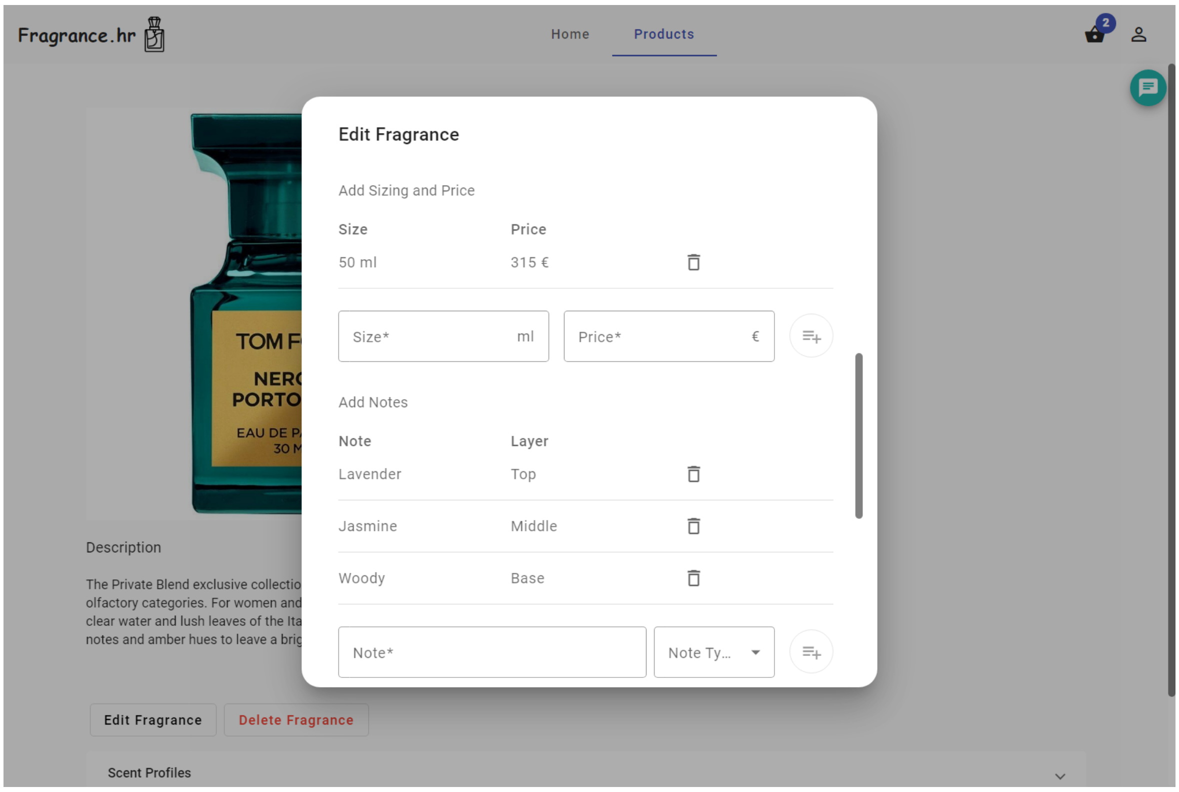
Hot. Sparkling. Tom Ford perfectly brings a cool breeze, sparkling clear water and lush leaves of the Italian Riviera into this

Amount*

Description

The Private Blend exclusive collectio
olfactory categories. For women and
clear water and lush leaves of the Ita
notes and amber hues to leave a brig

Figure 5.15 Product information edit/add


Fragrance.hr

Home

Products

Edit Fragrance

Add Sizing and Price

Size	Price
50 ml	315 €

Size* ml

Price* €

Add Notes

Note	Layer
Lavender	Top
Jasmine	Middle
Woody	Base

Note*

Note Ty...

Description

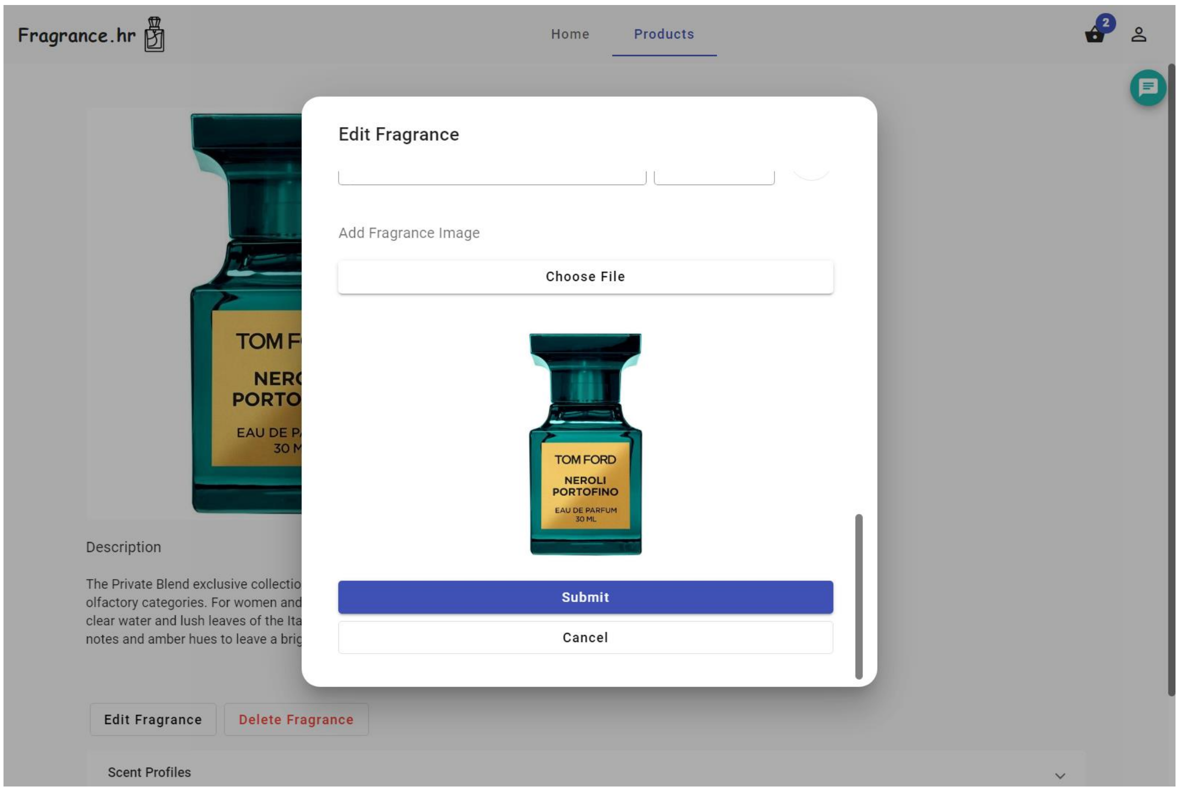
The Private Blend exclusive collectio
olfactory categories. For women and
clear water and lush leaves of the Ita
notes and amber hues to leave a brig

Edit Fragrance

Delete Fragrance

Scent Profiles

Figure 5.16 Product sizes, prices, and notes edit/add


Fragrance.hr

Home

Products

Edit Fragrance

Add Fragrance Image

Choose File

Submit

Cancel

Description

The Private Blend exclusive collectio
olfactory categories. For women and
clear water and lush leaves of the Ita
notes and amber hues to leave a brig

Edit Fragrance

Delete Fragrance

Scent Profiles

Figure 5.17 Product image add

The same window and form are opened when the administrator selects the option to add a new fragrance on the fragrances listing page, except for the fields not being filled, since a new product is being added.

Lastly, unauthenticated visitor, on header account icon click, is offered with an option so sign in to the account. By selecting it, the user is redirected to the Keycloak authentication view, where it is offered between sign in and register options, both visible in figures (Figure 5.18, Figure 5.19).

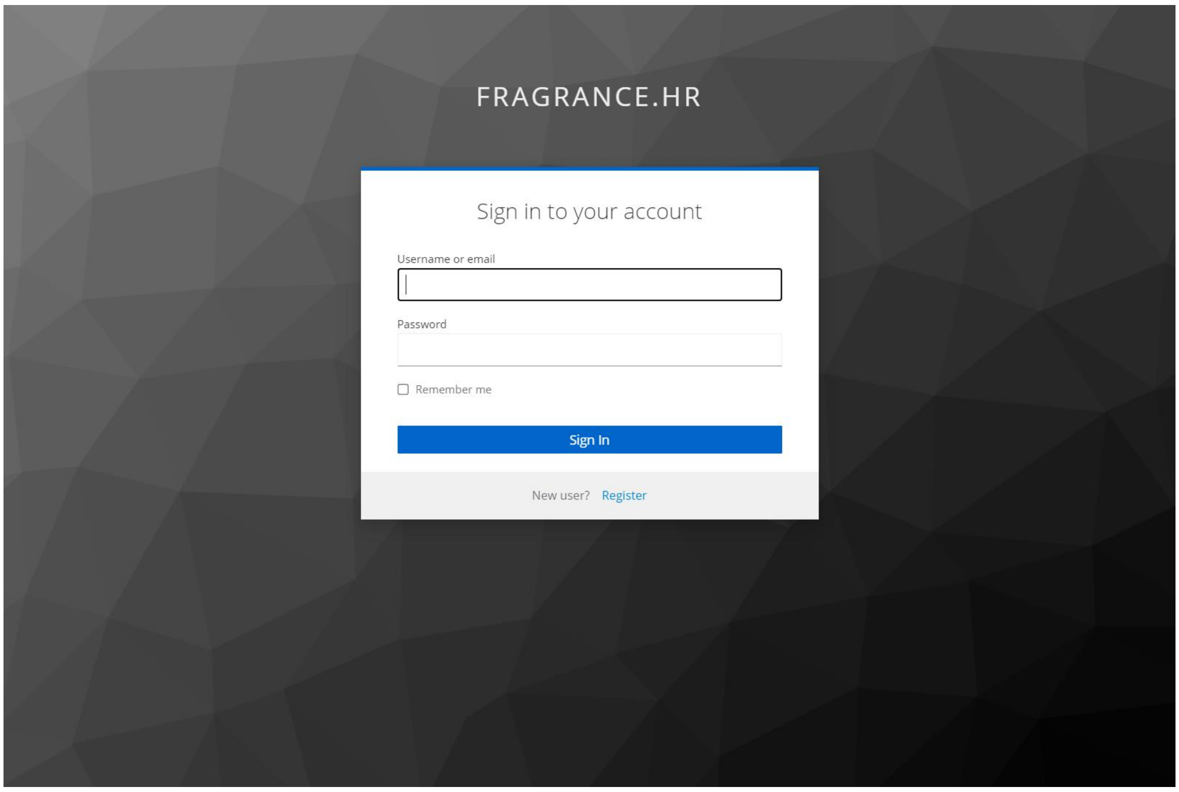


Figure 5.18 User sign in form

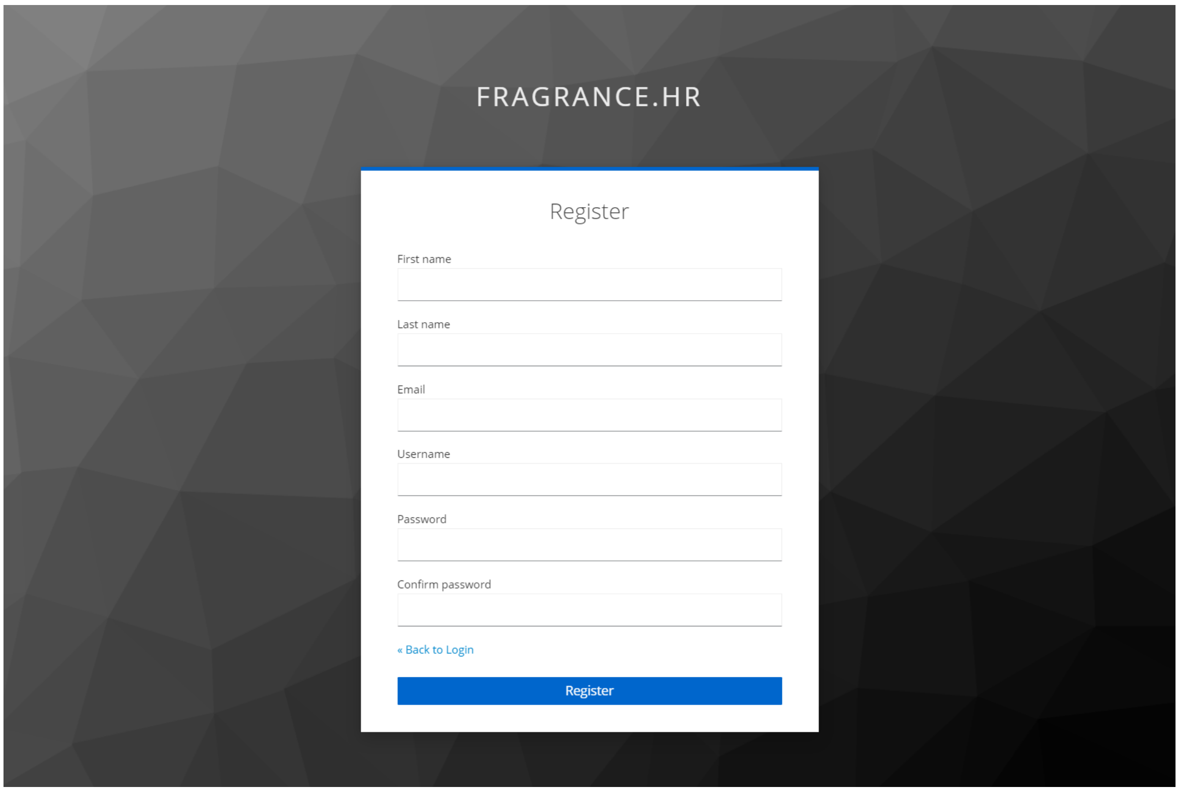


Figure 5.19 User registration form

6. Conclusion

In conclusion, microservices architecture presents a compelling approach for building web shop-based applications, since such applications can contain a lot of functionalities. Each functionality can be independently separated into the services, where each service handles a specific business logic.

By decomposing the application into independent services, the system gains numerous advantages, like scalability, resilience, and flexibility. The ability for each service to be distributed and independent makes them well suited for cloud-native and distributed systems. Additionally, microservices promote technological freedom, allowing for diverse tech stack across services.

Through the usage of patterns and components like API gateway, service discovery, service communication, security, etc., microservices can be a well-rounded and stable solution. Pairing it together with the client-side application, microservices can offer a great full stack solution for development of web applications.

However, implementation of microservices necessitates careful planning and consideration of the increased complexity in areas like service coordination, communication, and data consistency.

I think that microservice architecture offers a lot of benefits, with my favourite being the ability to separate concerns across the services and making them an independent and self-preserving unit. But this architecture needs to be carefully thought out, taking into account that the increased complexity and size can lead to bigger risks and costs, and sometimes, less is more.

Bibliography

[1] Salaheddin, Nada & Ahmed, Nuredin, *Microservices vs. Monolithic architectures [the differential structure between two architectures]*, MINAR International Journal of Applied Sciences and Technology.

[2] G. Blinowski, A. Ojdowska and A. Przybyłek, *Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation*, Institute of Computer Science, Warsaw University of Technology, Warsaw, Poland, 2022, vol. 10, pp. 20357-20374

[3] Florian Auer, Valentina Lenarduzzi, Michael Felderer, Davide Taibi,Ivić, *From monolithic systems to Microservices: An assessment framework*, Information and Software Technology journal, 2021, vol. 137

[4] P. Di Francesco, P. Lago and I. Malavolta, *Migrating Towards Microservice Architectures: An Industrial Survey*, 2018 IEEE International Conference on Software Architecture (ICSA), Seattle, WA, USA, 2018, pp. 29-2909

[5] Przemysław Jatkiewicz, Szymon Okrój, *Differences in performance, scalability, and cost of using microservice and monolithic architecture*, 38th ACM/SIGAPP Symposium on Applied Computing (SAC '23). Association for Computing Machinery, New York, USA, 2023, pp. 1038-1041

[6] W. Hasselbring and G. Steinacker, *Microservice Architectures for Scalability, Agility and Reliability in E-Commerce*, 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, Sweden, 2017, pp 243-246

[7] A. R. Sampaio et al., *Supporting Microservice Evolution*, 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), Shanghai, China, 2017, pp. 539-543

[8] C. Esposito, A. Castiglione and K. -K. R. Choo, *Challenges in Delivering Software in the Cloud as Microservices*, IEEE Cloud Computing, vol. 3, no. 5, pp. 10-14, 2016

[9] Rossi, Davide, Consistency and Availability in Microservice Architectures, Web Information Systems and Technologies, 2019, pp 39-55

[10] Toffetti, Giovanni & Brunner, Sandro & Blöchlinger, Martin & Dudouet, Florian & Edmonds, Andy, *An architecture for self-managing microservices*, The 1st International Workshop conference, 2015, pp. 19-24

[11] Jaramillo, David & Nguyen, Duy & Smart, Robert, Leveraging microservices architecture by using Docker technology, SoutheastCon 2016 Conference, 2016, pp. 1-5

[12] Yura Abharian, J*ustification of microservice approach to web development*, Innovative Solutions In Modern Science, Zaporizhia National Technical University, Software Engineer SoftServe, Zaporozhye, Ukraine, 2021, vol. 8, no. 52

[13] R. Perlin, D. Ebling, V. Maran, G. Descovi and A. Machado, *An Approach to Follow Microservices Principles in Frontend*, 2023 IEEE 17th International Conference on

Application of Information and Communication Technologies (AICT), Baku, Azerbaijan, 2023, pp. 1-6

[14] Zhou, Jun Ying, *Functional requirements and non-functional requirements : a survey*, Masters thesis, Concordia University, 2004

[15] *What is Use Case Diagram?*, Visual Pardigm, link: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/, accessed: May 2024.

[16] Denis Szczukocki, *Multi-Module Project with Maven*, Baeldung, (2024, May), link: https://www.baeldung.com/maven-multi-module, accessed: May 2024.

[17] *Introduction to the POM*, Apache Maven Project, (2024, June), link: https://maven.apache.org/guides/introduction/introduction-to-the-pom, accessed: June 2024.

[18] *Docker Compose overview*, Docker documentation, link: https://docs.docker.com/compose/, accessed: June 2024.

[19] *PostgreSQL: The World's Most Advanced Open Source Relational Database*, PostgreSQL, link: https://www.postgresql.org/, accessed: June 2024.

[20] *pgAdmin PostgreSQL Tools*, pgAdmin, link: https://www.pgadmin.org/, accessed: June 2024.

[21] *What is a REST API?*, IBM, link: https://www.ibm.com/topics/rest-apis, accessed: June 2024.

[22] *Common Application Properties*, Spring documentation, link: https://docs.spring.io/spring-boot/appendix/application-properties, accessed: June 2024.

[23] *Spring Data JPA*, Spring documentation, link: https://spring.io/projects/spring-data-jpa, accessed: June 2024.

[24] *Introducing JSON*, JSON, link: https://www.json.org/json-en.html, accessed: June 2024.

[25] System Design By CHK, *System Design —A Comprehensive Guide on Synchronous & Asynchronous Microservice Communication*, Medium, (2023, August), link: https://medium.com/@systemdesignbychk/system-design-a-comprehensive-guide-on-synchronous-asynchronous-microservice-communication-8bda324943b8, accessed: June 2024.

[26] *Spring Cloud OpenFeign*, Spring documentation, link: https://spring.io/projects/spring-cloud-openfeign, accessed: June 2024.

[27] *Spring Cloud – OpenFeign with Example Project*, GeeksForGeeks, (2023, December), link: https://www.geeksforgeeks.org/spring-cloud-openfeign-with-example-project/, accessed: June 2024.

[28] *Java JDBC API*, Oracle Java documentation, link: https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/, accessed: June 2024.

[29] *Integrate PayPal Checkout*, Paypal Developer, link: https://developer.paypal.com/studio/checkout/standard/integrate, accessed: June 2024.

[30] *Get started with PayPal REST APIs*, PayPal Developer, link: https://developer.paypal.com/api/rest/, accessed: June 2024.

[31] *Spring Cloud Netflix*, Spring Documentation, link: https://spring.io/projects/spring-cloud-netflix, accessed: June 2024.

[32] *Keycloak*, Keycloak, link: https://www.keycloak.org/, accessed: June 2024.

[33] *OAuth 2.0*, oauth, link: https://oauth.net/2/, accessed: June 2024.

[34] *How OpenID Connect Works*, openid, link: https://openid.net/developers/how-connect-works/, accessed: June 2024.

[35] *Server Administration Guide*, Keycloak documentation, link: https://www.keycloak.org/docs/latest/server_admin/, accessed: June 2024.

[36] *Angular OAUTH2 OIDC*, npmjs, link: https://www.npmjs.com/package/angular-oauth2-oidc, accessed: June 2024.

[37] *Introduction to JSON Web Tokens*, jwt, link: https://jwt.io/introduction, accessed: June 2024.

[38] José Matos, *Angular Services vs. Components: Understanding When to Use Each*, Angular Dive, (2023, March), link: https://angulardive.com/blog/angular-services-vs-components-understanding-when-to-use-each/, accessed: June 2024.

[39] Rick Merritt, *What Is Retrieval-Augmented Generation, aka RAG?* Nvidia blog, (2023, November), link: https://blogs.nvidia.com/blog/what-is-retrieval-augmented-generation/, accessed: June 2024.

Sažetak

Naslov: Razvoj mikroservisne aplikacije za web trgovinu

Cilj ovog rada je prikazati proces implementacije i primjene arhitekture mikroservisa u domeni aplikacija za web trgovine. Potrebne funkcionalnosti web trgovine izdvajaju se kroz specifikaciju zahtjeva, inicijalno definirajući monolitni blok koji upravlja svim funkcionalnostima aplikacije.

Razdvajanjem i dekompozicijom odgovornosti iz monolita u servise, definira se arhitektura mikroservisa za web trgovinu. Mikroservisi definirani za web trgovinu su servisi za: proizvode, recenzije, narudžbe, plaćanja, asistenciju, i inventar.

Nakon definiranja, implementacija servisa se provodi korištenjem tehnologija Java, Spring Boot i Spring Cloud. Svaki servis se razvija odvojeno kao neovisna jedinica, s odvojenom bazom podataka, ako je potrebna. Dodatno, implementirao je aplikacijsko programsko sučelje (eng. Application Programming Interface, API) u obliku pristupnika koji služi kao glavna ulazna i sigurnosna točka za poslužiteljsku stranu, te servisni registar za upravljanje registracijom servisa. Za implementaciju plaćanja, servis za plaćanje integrira PayPal servis za plaćanje, a za chatbot, asistent servis integrira OpenAI ChatGPT u kombinaciji sa RAG-om (eng. Retrieval-Augmented Generation). Na kraju, Angular okvir se koristi za implementaciju aplikacije na klijentskoj strani, upotpunjući full-stack web aplikacijsko rješenje za platformu web trgovine.

Ključne riječi: mikroservisi, arhitektura, web trgovina, aplikacija, Java, Spring, Angular

Summary

Title: Development of a microservice application for a web store

The goal of this paper is to show the process of implementing and applying microservices architecture to the domain of web store applications. The required web shop functionalities are extracted through the requirements specification, initially defining a monolithic block handling all the business logic for the application.

By separating and decomposing the responsibilities from the monolith into the services, a microservices architecture is defined for the web store. The microservices defined for the web store are product, review, order, payment, AI assistant, and inventory services.

Once defined, the services implementation is carried out, through the usage of Java, Spring Boot, and Spring Cloud technologies. Each service is developed separately as an independent unit, with a separate database, if needed. Additionally, an API (Application Programming Interface) gateway is implemented to serve as a main entry and security point for the server-side, and discovery server to handle service registry. For the implementation of payment, the payment service integrates PayPal payment provider, and for the chatbot, the chat service integrates OpenAI ChatGPT paired with RAG (Retrieval-Augmented Generation). To round everything up, Angular framework is used for client-side application implementation, fulfilling a full-stack web application solution for e-commerce web store platform.

Keywords: microservices, architecture, web store, application, Java, Spring, Angular