VIETNAM GENERAL CONFEDERATION OF LABOR
TON DUC THANG UNIVERSITY
MAJOR INFORMATION TECHNOLOGY

VÕ TẤN PHÁT – 518H0239
NGUYỄN VIẾT MINH NHẬT – 518H0542

BUILDING E-COMMERCE WEBSITE USING JAVA SPRING BY MICRO-SERVICES ARCHITECTURE

Information Technology Project 2

SOFTWARE ENGINEERING

HO CHI MINH CITY, 2023

VIETNAM GENERAL CONFEDERATION OF LABOR
TON DUC THANG UNIVERSITY
MAJOR INFORMATION TECHNOLOGY

VÕ TẤN PHÁT – 518H0239
NGUYỄN VIẾT MINH NHẬT – 518H0542

BUILDING E-COMMERCE WEBSITE USING JAVA SPRING BY MICRO-SERVICES ARCHITECTURE

Information Technology Project 2

SOFTWARE ENGINEERING

HO CHI MINH CITY, 2023

ACKNOWLEDGMENT

We would like to express our deep gratitude to Mr. Vo Van Thanh for guiding the team in the process of learning the microservices architecture using Java Spring Boot. You helped us understand much important knowledge about microservices architecture. Besides, he also helped the team, closely following the progress throughout the completion of this project. We thank you very much for taking the time to guide and help the group in the learning process. They believe that the knowledge they have learned will help them in their future careers. Thank you once again and the team looks forward to having the opportunity to study with you in the future.

Ho Chi Minh city, 8 February 2023
Authors
(Sign and write your full name)

Võ Tấn Phát

Nguyễn Viết Minh Nhật

TEACHER'S ASSESSMENT SHEET

Name of instructor: Võ Văn Thành..

Comments: ....

....

....

....

Overall score according to Rubrik rubric:...

Ho Chi Minh city, 8 February 2023
Authors
(Sign and write your full name)

Thành

Võ Văn Thành

PROJECT COMPLETED
AT TON DUC THONG UNIVERSITY

I hereby declare that this is the product of our project and under the guidance of Mr. Vo Van Thanh. The research contents and results in this topic are honest and have not been published in any form before. The data in the tables for analysis, comments and evaluation are collected by the author himself from different sources, clearly stated in the reference section.

In addition, the project also uses a number of comments, assessments as well as data of other authors, other agencies and organizations, with citations and source annotations.

If I find any fraud, I will take full responsibility for the content of my project. Ton Duc Thang University is not related to copyright and copyright violations caused by me during the implementation process.

Ho Chi Minh city, 8 February 2023
Authors
(Sign and write your full name)

Võ Tấn Phát

Nguyễn Viết Minh Nhật

SUMMARY

My group learned about Microservices architecture and built an e-commerce website using Java Spring Boot. My team will split the back end into services including UserService, ProductService, OrderService, PaymentService, ImageService and HistoryService. The front-end part will also be divided into 2 separate projects, the sales website and the management website.

We first explored the Microservices architecture and Domain Driven Design. After learning the above topics, we continue to learn about Java Spring Boot and try to implement a service using this technology. After success, continue with the next services. After completing the services and completing the back-end, we continue to do the front-end for the project.

After completing the project, we have an e-commerce website with all basic functions. Our team also understands microservices architecture, DDD and also improves coding ability with Java Spring Boot technology.

TABLE OF CONTENTS

CHAPTER I – INTRODUCTION

1.1 Introduction to the topic

With the assigned topic, we will learn about the Microservice architecture that is being applied by many projects today. Then my team will re-implement this architecture with Java Spring Boot in the backend and use ReactJS and Angular frameworks to build an e-commerce website.

1.2 Purpose and meaning of the topic

Microservice architecture is the architecture that is trending recently and is being used by a lot of businesses and projects. Microservice helps website systems can be easily extended and maintenance, so this is a very popular architecture. My team will use this architecture to create an e-commerce website, a type of website that is also growing and it can demonstrate the advantages of microservices architecture.

E-commerce website will include the following main functions:

- Product Management.
- Customer Information Management.
- Manage shopping cart.
- Manage order list.
- Payment Management.

1.3 General description

An e-commerce website built on a microservice architecture can be easily extended later, and when one function fails, it will not affect the other functions.

CHAPTER II – ANALYSIS

2.1 Use case

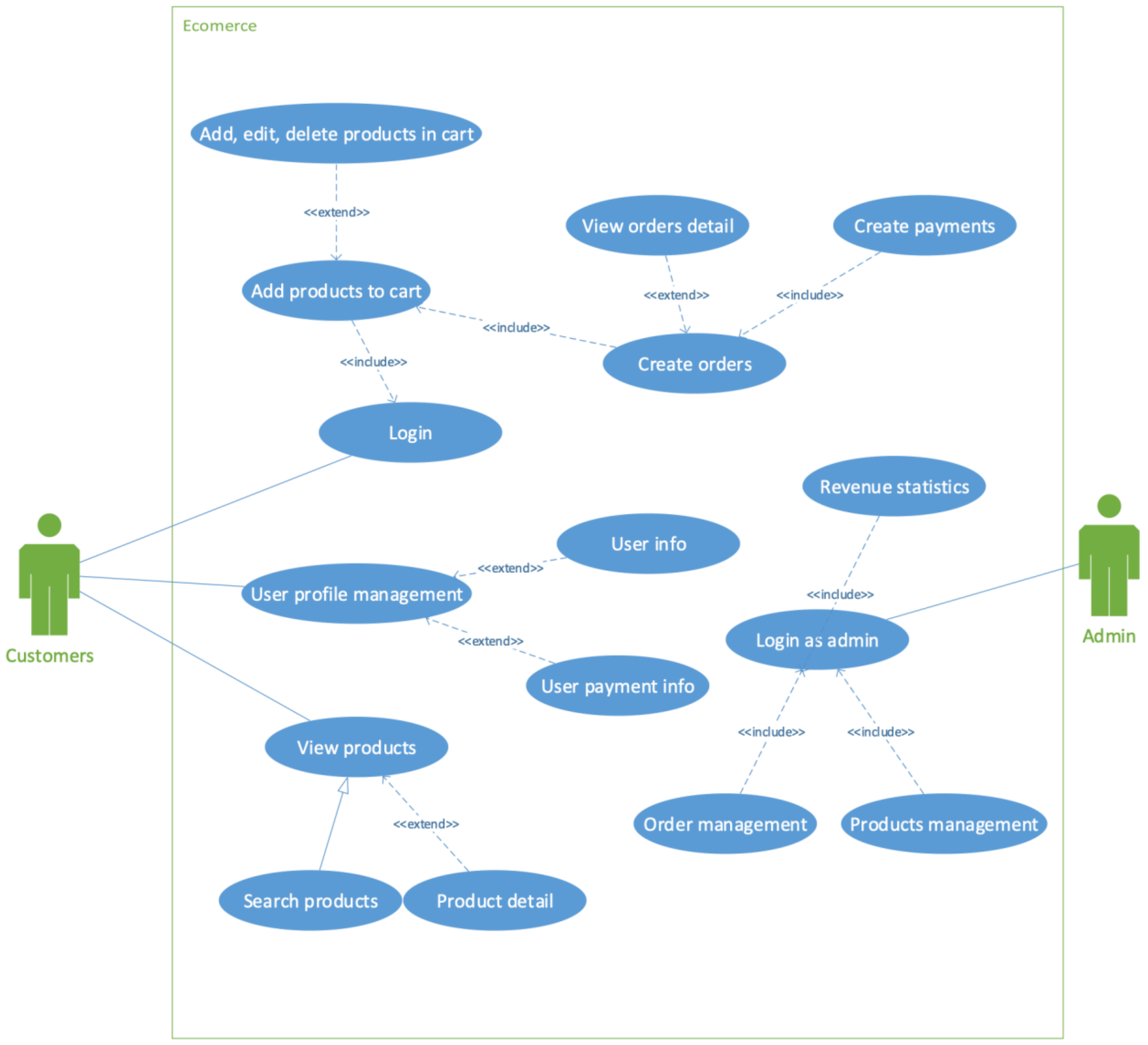


Figure 2.1.1 Use case diagram

E-commerce website will have the following main functions with the following use cases:

Table 2.1.1 Use cases

STT	Use case	Description
1	Login	Allow users to login to get user information
2	User profile management	Manage basic user information as well as accounts and payment methods.
3	View products	Displays a list of available products in the system
4	Search products	Displays a list of products according to users' conditions.
5	Product detail	The function displays product details such as price, color, quantity, description...
6	Add product to cart	The function allows users to store the products they intend to buy in the shopping cart
7	Edit cart	User can change cart during use of website. The operations can be adding, editing, deleting products in the shopping cart.
8	Create orders	User will create orders based on the number of products in the cart.
9	View order detail	The function allows users to track order details such as order status, amount to be paid, estimated delivery time, number of products in the order, etc.
10	Create payment	The function allows users to pay for the order created and the order will be posted to the store system to start shipping after making the payment.
11	Products management	Function for admin to add, edit, delete quantity and information of products
12	Order management	Functions for admin to add, edit, delete information on orders as well as track the progress of orders

2.2 System specification

- Receive orders and save them.
- Manage user's basic information and card and bank account information.
- Statistics of revenue from orders.
- Monitor the status and quantity of products in stock.

CHAPTER III – THEORETICAL BASIS

3.1 Microservices Architecture

3.1.1 Concept

Microservices is an architectural pattern for building software systems consisting of small, independently deployable services. Each service is responsible for a specific business, such as user management, payment processing, or image hosting.

Many websites and applications have adopted microservices architecture to improve scalability and reliability. Some examples include:

- Netflix: One of the largest and well-known adopters of microservices, Netflix uses microservices to deliver its streaming service to over 200 million users globally.
- Amazon: Amazon uses microservices to power its various e-commerce and retail services, including Amazon Prime, Amazon Fresh, and Amazon Web Services.
- Uber: Uber uses microservices to power its ride-hailing app, allowing different parts of the app, such as the map, driver database, and payment processing, to run as separate services.
- Airbnb: Airbnb uses microservices to manage the various parts of its platform, including property listings, user profiles, and booking management.
- SoundCloud: SoundCloud, the online audio distribution platform, uses microservices to manage its large and complex system of audio uploads, storage, and delivery.

The goal of microservices is to improve scalability, reliability, and development speed for complex software systems. By breaking down the system into smaller, independent services, teams can deliver new features faster, scale services as needed, and improve overall performance and system stability.

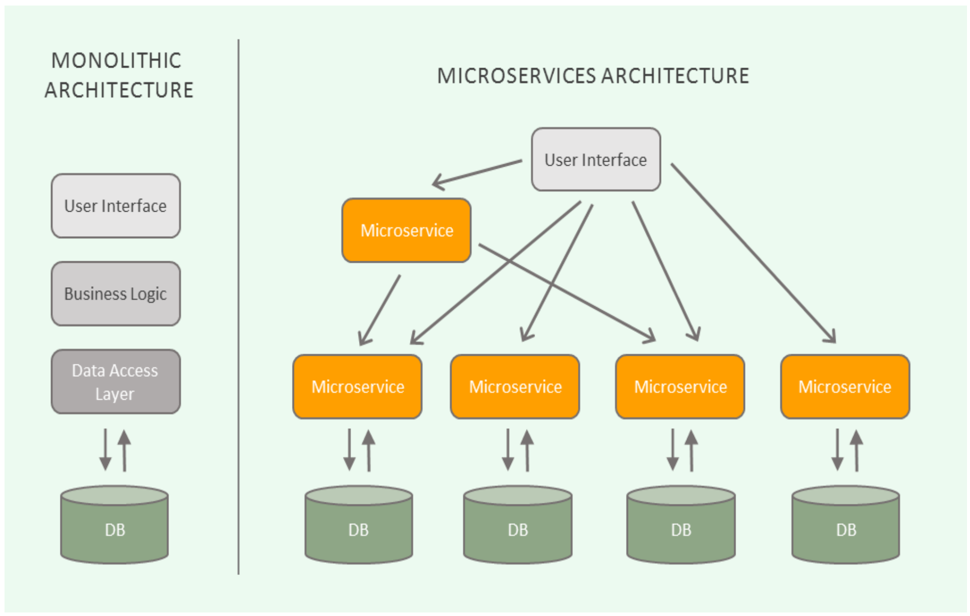


Figure 3.1.1 Illustration of microservices architecture

3.1.2 Features of microservices

- Modularity: Each service is small and has a single, well-defined responsibility, making it easier to understand, develop, and maintain.
- Independence: Services can be deployed, updated, and versioned independently, enabling faster development cycles and reduced downtime.
- Loose Coupling: Services communicate with each other through lightweight mechanisms, typically using APIs, and are loosely coupled, allowing them to evolve and scale independently.
- Polyglot: Services can be built using different programming languages, databases, and tools, allowing teams to choose the best technology for their needs.
- Decentralized: The responsibility for decision-making and management is decentralized, enabling teams to make decisions and move quickly.

3.1.3 Advantages and disadvantages of microservices

3.1.3.1 Advantage

- Scalability: Services can be scaled independently based on specific requirements.

- Resilience: If one service fails, the other services continue to function.
- Agility: Services can be updated, deployed and versioned independently.
- Technology Heterogeneity: Different services can be built using different programming languages, databases, and tools.
- Separation of Concerns: Each service has a specific, well-defined responsibility which makes the application easier to understand, develop, and maintain.

3.1.3.2 Disadvantage

- Complexity: Microservices can introduce complexity in terms of deployment, testing, and management.
- Inter-service communication: Services must communicate with each other, and there may be challenges in ensuring seamless communication between services, particularly in terms of performance, reliability, and security.
- Service discovery: Finding the right service to communicate with can be challenging, especially in a large and complex system.
- Distributed data management: Managing data consistency across multiple services can be difficult, especially in situations where data updates occur in more than one service.
- Debugging and Monitoring: Debugging and monitoring a microservices-based application can be more complex than in a monolithic architecture, as there are more moving parts to monitor and debug.
- Initial investment: Setting up a microservices architecture can require a significant initial investment, including time and resources for refactoring and deploying the application.

3.2. Domain Driven Design (DDD)

3.2.1 Concept of DDD

- Domain-Driven Design (DDD) is a software development approach that focuses on modeling and understanding the core business concepts and problems, and then defining the architecture and design of the system based on those models.
- DDD aims to improve the communication between technical and business stakeholders, resulting in a more aligned and accurate representation of the business domain within the software system. It also promotes the use of patterns and practices that support the design of maintainable, scalable, and testable systems.
- DDD is often used in conjunction with microservices to create systems that are flexible, scalable, and capable of evolving with changing business needs. The microservices can be aligned with the bounded contexts within the domain, making it easier to manage the dependencies between services and improve the overall system design.

3.2.2 Layers in DDD

Domain-Driven Design (DDD) defines a layered architecture that consists of the following layers:

- Domain layer: This is the core of the system and represents the business concepts, entities, and rules. The domain layer is responsible for defining the business logic and behavior.
- Application layer: This layer acts as the interface between the domain layer and the infrastructure layer. It implements the application-specific use cases and communicates with the domain layer to execute the desired behavior.
- Infrastructure layer: This layer provides the underlying technical infrastructure for the system, such as databases, messaging systems, and web servers. The

infrastructure layer communicates with the application layer to perform the necessary technical operations.

- Presentation layer: This layer is responsible for presenting the information to the user and receiving input from the user. It communicates with the application layer to retrieve the necessary information and present it to the user.

These layers are loosely coupled and can be developed and deployed independently, allowing for greater flexibility and scalability in the system. The domain layer is the core of the system, and the other layers are designed to support it. This layered architecture helps to ensure that the system is maintainable, scalable, and testable, and that the business logic is well-encapsulated.

CHAPTER IV – STRUCTURAL DIAGRAM

4.1 System Diagram

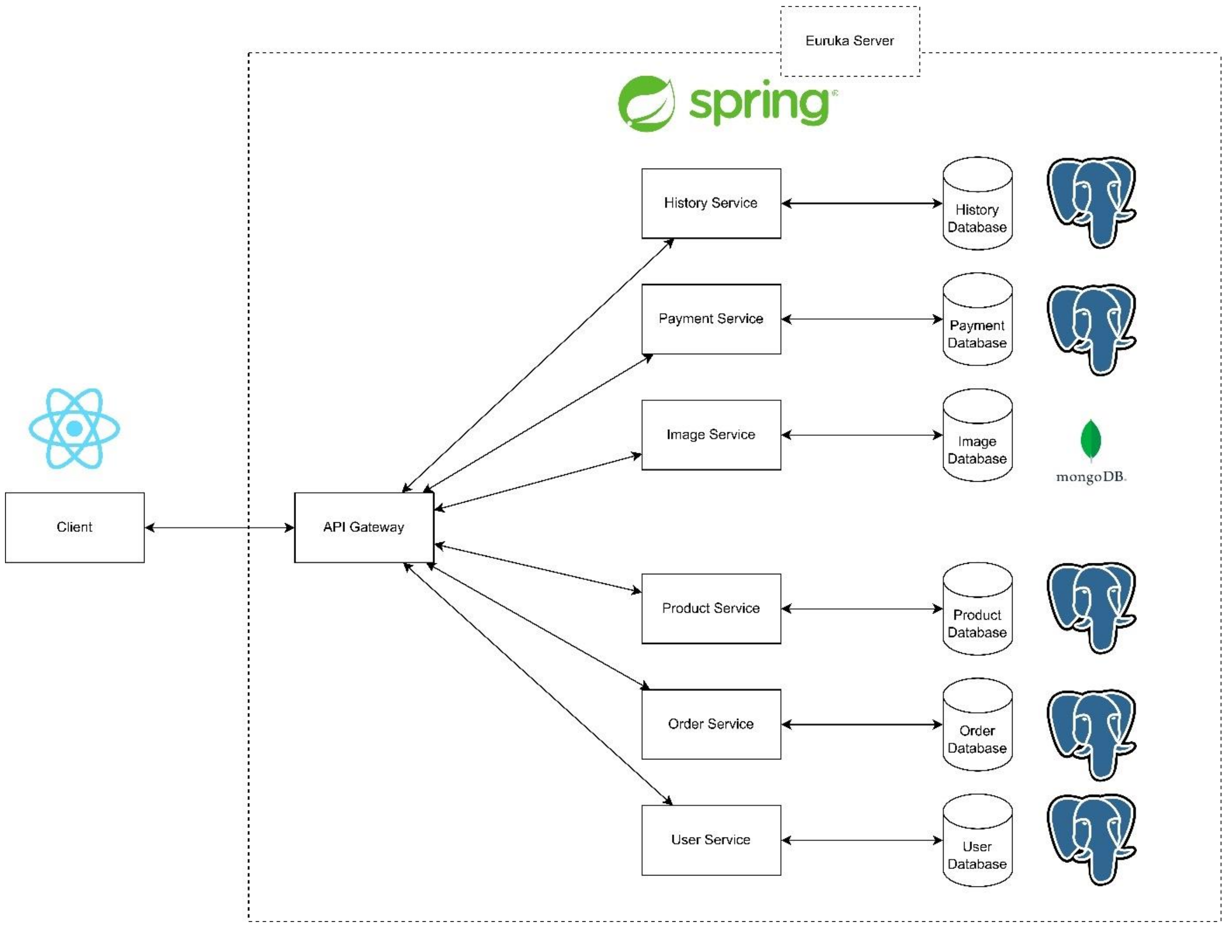


Figure 4.1.1 System diagram

E-commerce website built according to microservices architecture will be divided into many independent services, each service will also have its own database. My group's e-commerce website will divide the system into six services:

- Product Service: Manage products, quantity of products in stock.
- Order Service: Manage orders, order status.
- Image Service: Image management, used mainly with the product.
- User Service: Manage customer information and the number of orders created.
- Payment Service: Manage payment information of orders.

- History Service: Manage product history information, customer reviews.

As for the system, my team will use Java Spring Boot to implement the Services and the database will use PostgreSQL. Particularly Image Service will be using Mongo SQL. Services will communicate with each other through RestAPIs to receive information from other Services and return results. In each Service will be built according to Domain Driven Design (DDD) architecture

On the Client side, the Client will communicate with the microservices system through the API gateway, this is an intermediate gateway to enter the system, the API Gateway will receive requests from the Client to edit, authenticate and direct them to the APIs. specifically, to the Services at the back of the system. The client side will be built with ReactJS, a powerful library that is being appreciated and used by many projects currently, for sales websites. And Angular, a framework built by Google, for the sales management page. The division of the sales system into 2 different websites as well as a microservices architecture for the front-end.

4.2 Database in Services

4.2.1 Product Service

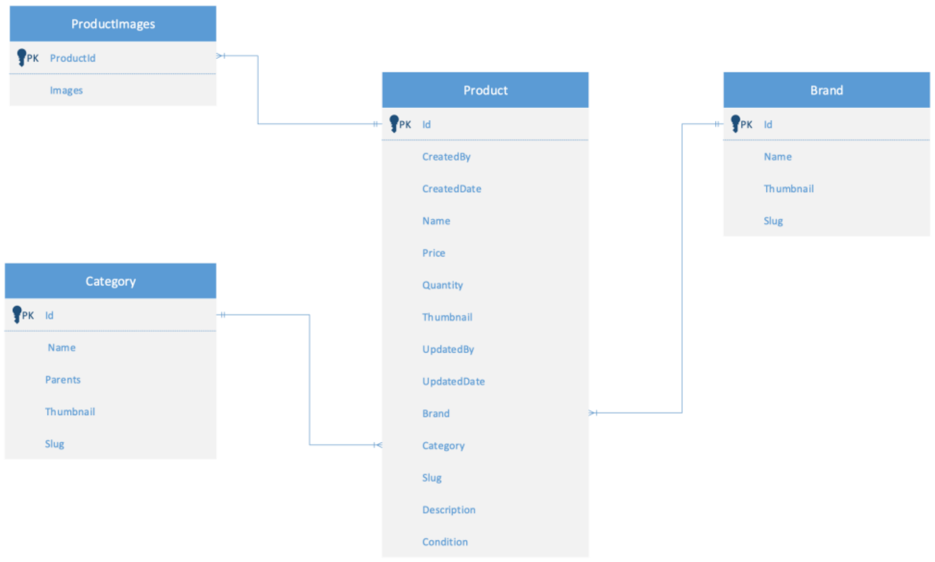


Figure 4.2.1 Product Service database diagram

4.2.2. User Service

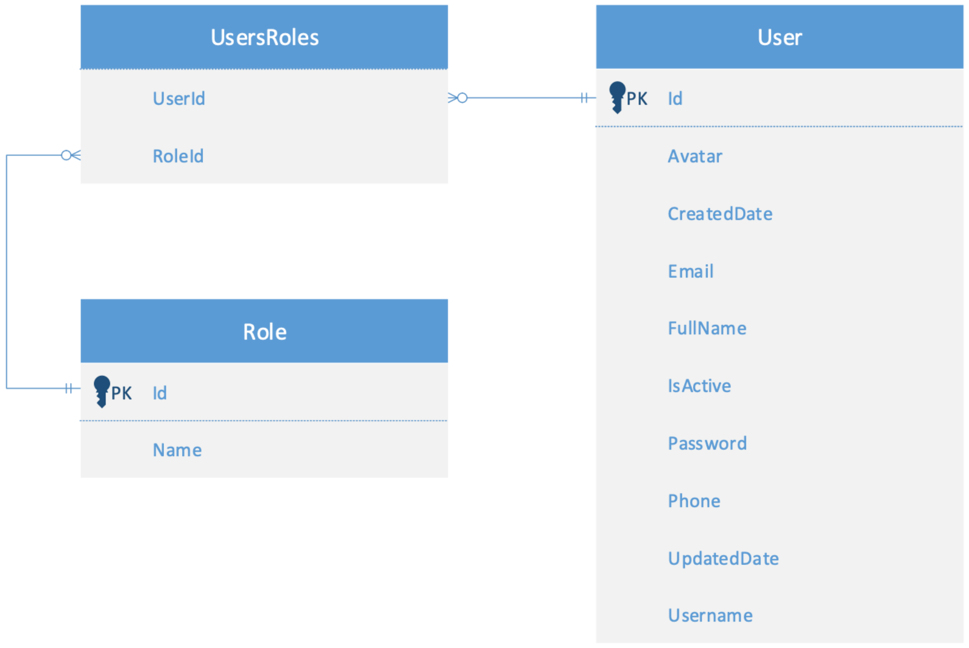


Figure 4.2.2 User Service database diagram

4.2.3 Payment Service

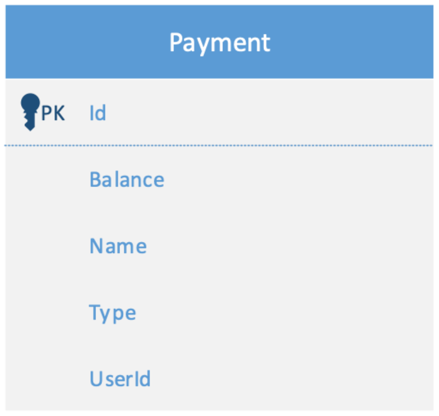


Figure 4.2.3 Payment Service database diagram

4.2.4 Order Service

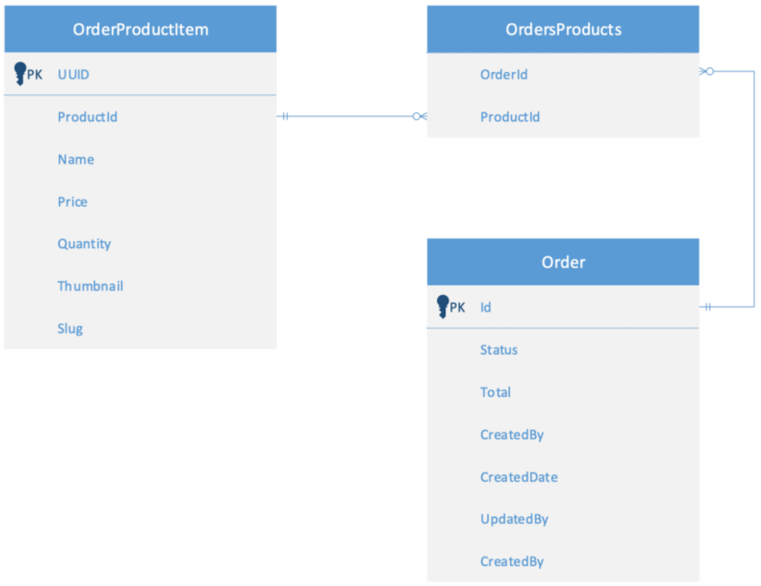


Figure 4.2.4 Order Service database diagram

4.2.5 History Service

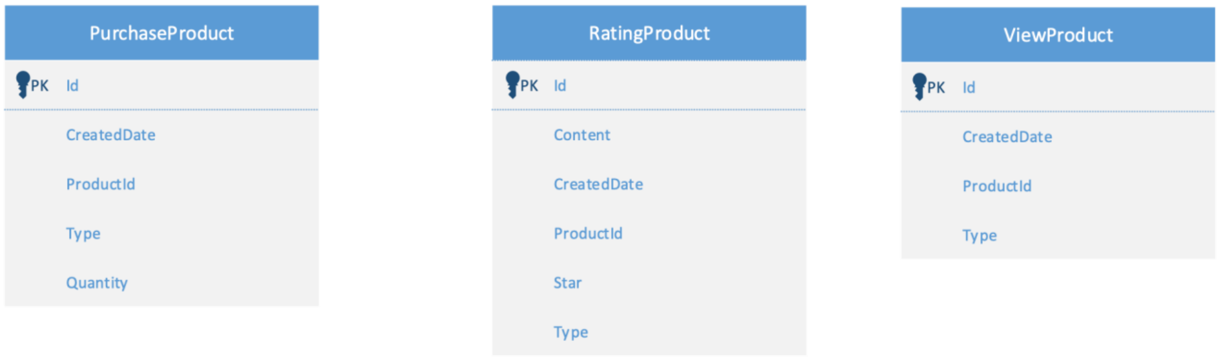


Figure 4.2.5 History Service database diagram

4.2.6 Image Service

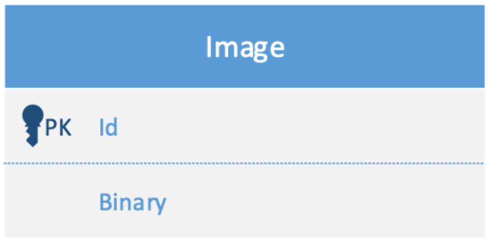


Figure 4.2.6 Image Service database diagram

4.3 Flowchart of the system

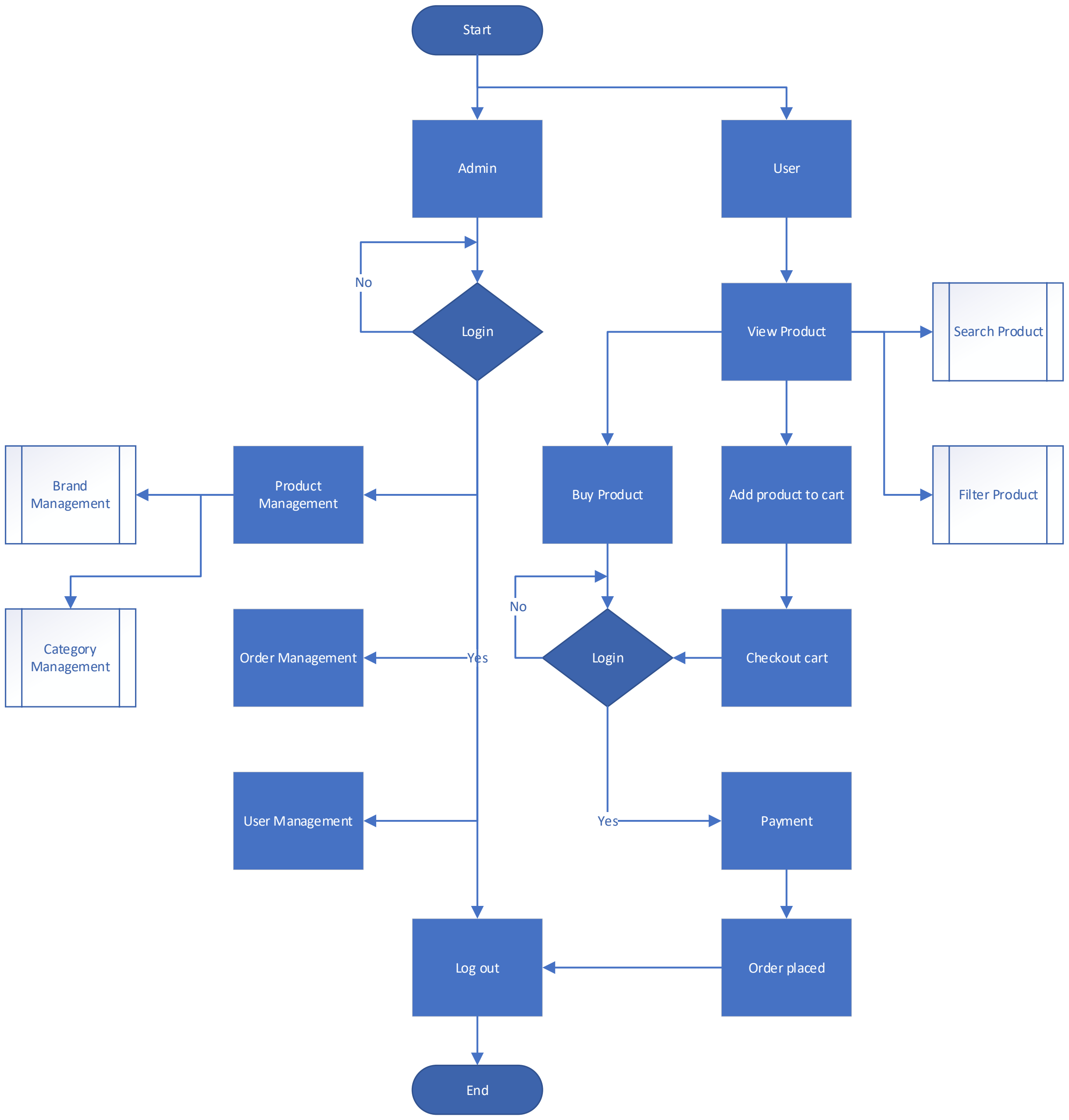


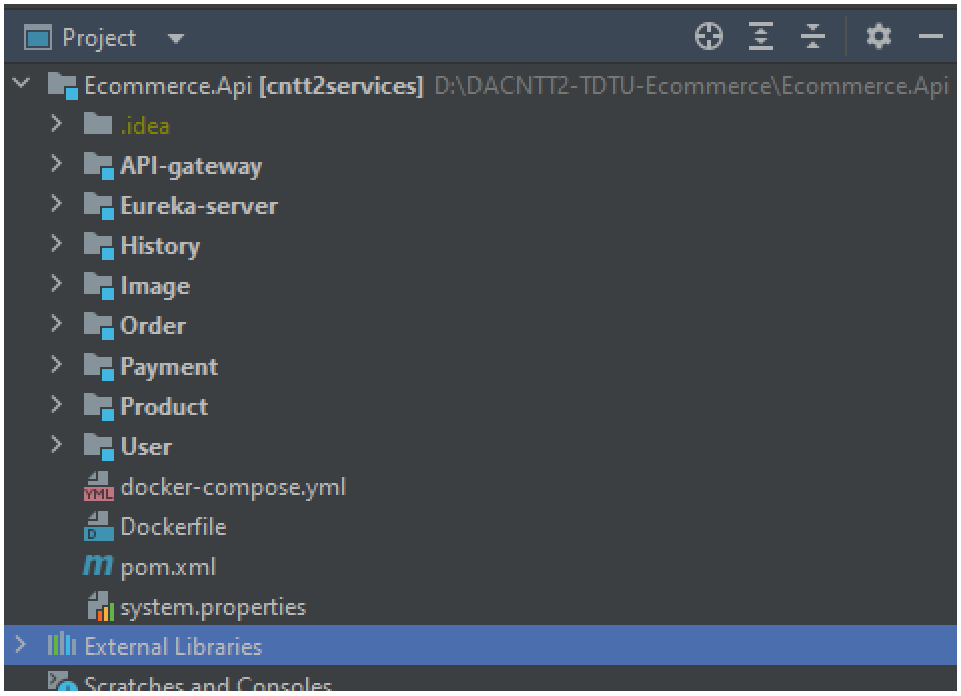
Figure 4.3.1 Flowchart

CHAPTER V - SYSTEM IMPLEMENTATION

5.1 Implementation of microservices-based back-end using Java Spring Boot:

5.1.1 Introduction:

As briefly summarized in the structural diagram section, our architecture is comprised of five services, namely the Product Service, User Service, Order Service, Payment Service, History Service, and Image Service. The team utilizes an API Gateway to facilitate the routing of requests from clients to the respective services. Furthermore, Eureka is employed to ensure seamless communication and coordination between the services.

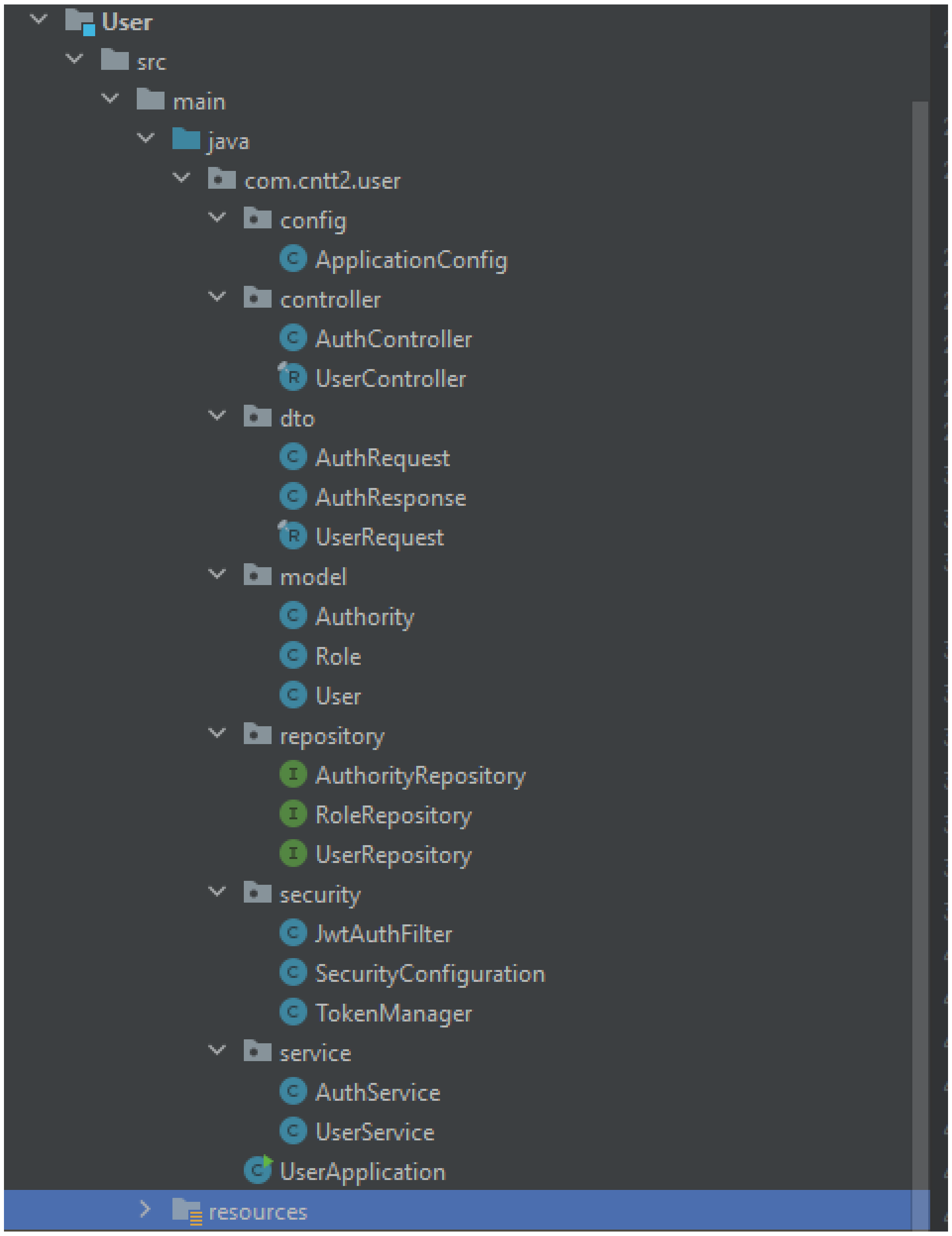


```
Project
Ecommerce.Api [cntt2services] D:\DACNTT2-TDTU-Ecommerce\Ecommerce.Api
  .idea
  API-gateway
  Eureka-server
  History
  Image
  Order
  Payment
  Product
  User
  docker-compose.yml
  Dockerfile
  pom.xml
  system.properties
External Libraries
Scratches and Consoles
```

Figure 5.1.1.1: Architecture of the entire project.

5.1.2 User Service

The User Service is the most critical component of this system, and it is divided into two parts, namely User and Auth. The organizational structure of the User Service is as follows:



```
User
  src
    main
      java
        com.cntt2.user
          config
            ApplicationConfig
          controller
            AuthController
            UserController
          dto
            AuthRequest
            AuthResponse
            UserRequest
          model
            Authority
            Role
            User
          repository
            AuthorityRepository
            RoleRepository
            UserRepository
          security
            JwtAuthFilter
            SecurityConfiguration
            TokenManager
          service
            AuthService
            UserService
          UserApplication
      resources
```

Figure 5.1.2.1: Organizational structure of the User Service.

This service is organized according to the DDD structure with the following layers:

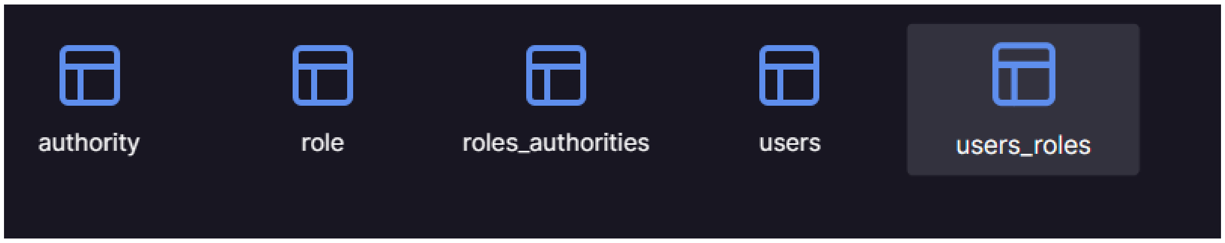
- Application layer: Controllers.
- Domain layer: Models, DTOs, Repositories, Services
- Infrastructure layer: Config

After dividing the structure of UserService as above, our team began implementing the functionalities of this service.

5.1.2.1 Model

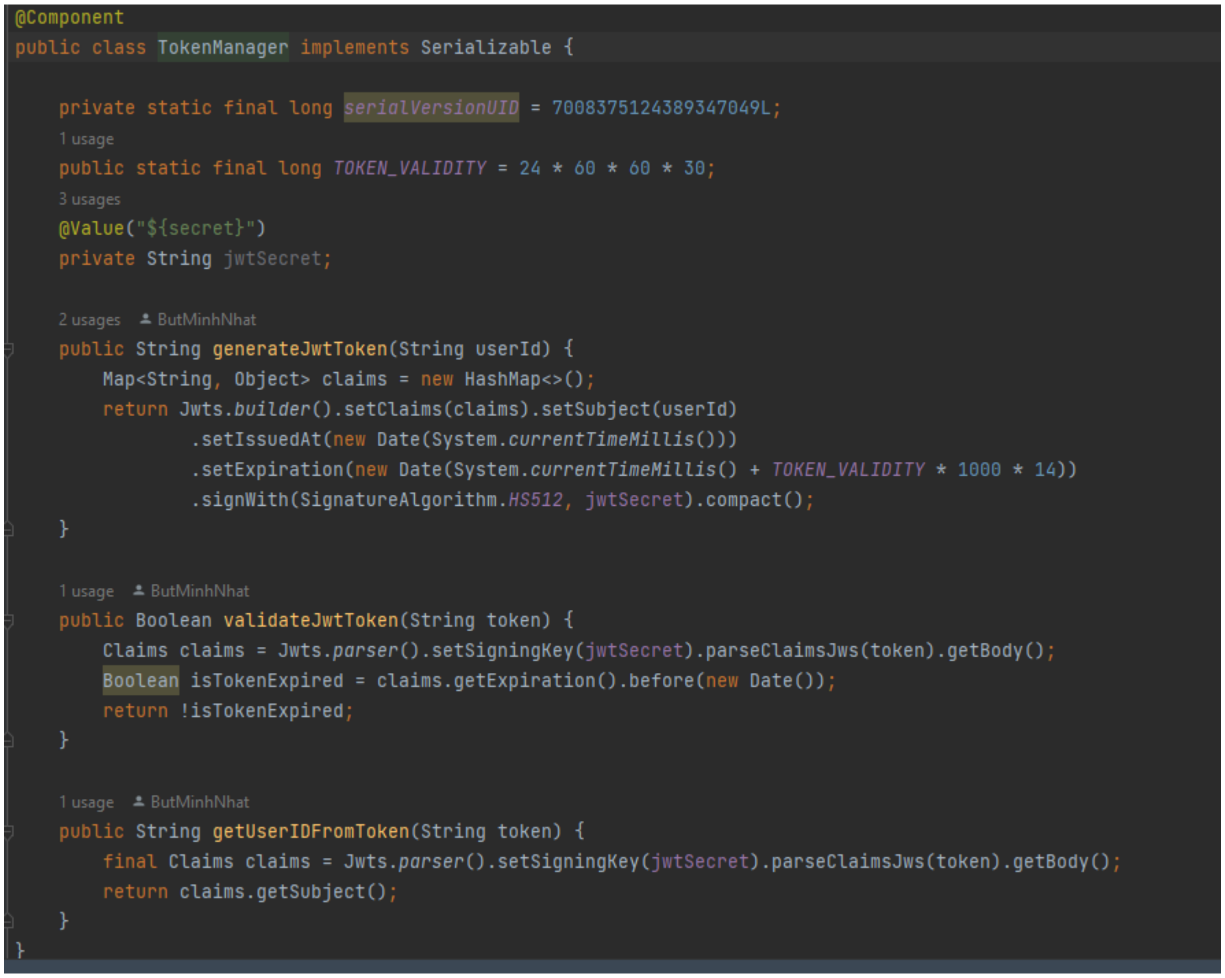
In UserService, there are five different models, each with the following details:

- Authority: a cache model used for authentication purposes.
- Role: contains the roles of users in the system.
- Roles_authorities: a cache model used for authenticating roles.
- Users: contains information about users.
- User_roles: contains the roles of users.



5.1.2.2 Security

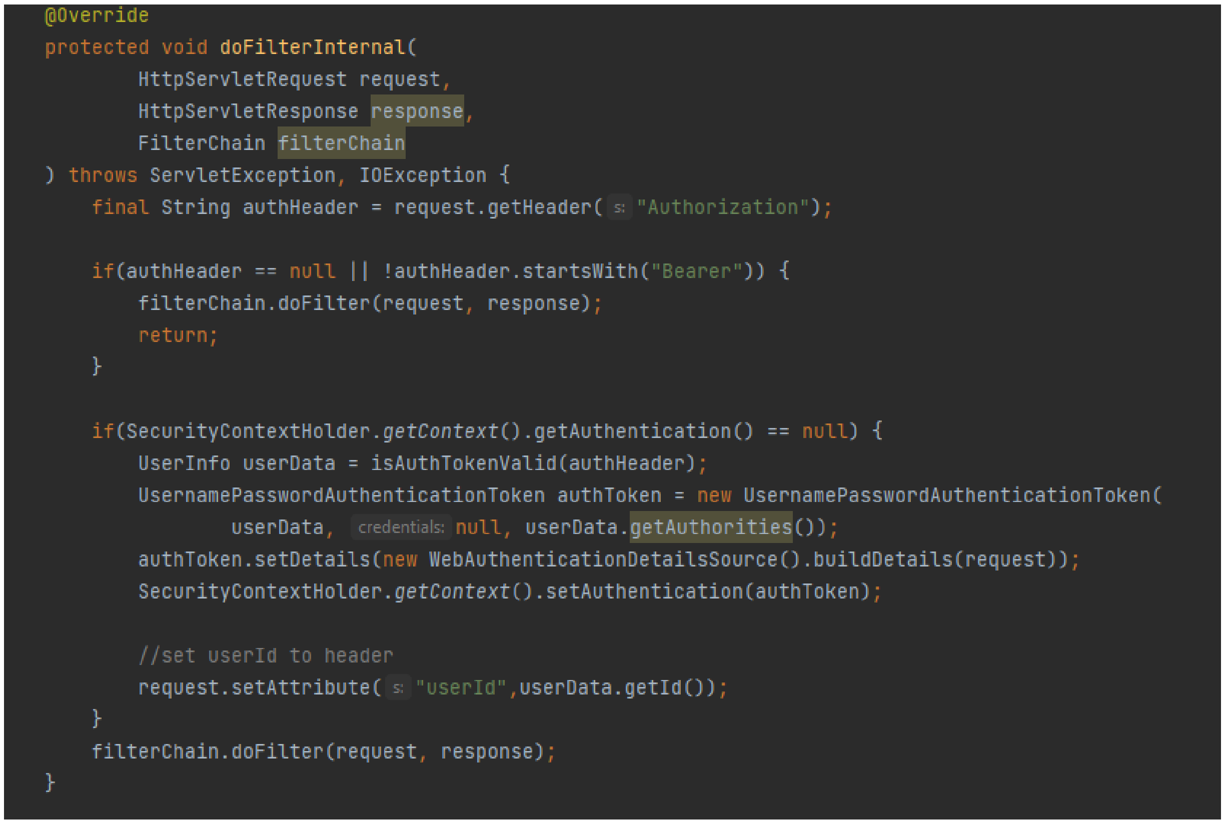
In this service, there is a security layer that serves as middleware to receive requests from clients and other services. It retrieves the token attached in the request header and uses it to authenticate via the API authentication, which will be discussed below. This layer also includes a TokenManager file used to configure the token generation method as well as the token decryption method (as shown in Figure 5.1.2.2).



```
@Component
public class TokenManager implements Serializable {

    private static final long serialVersionUID = 7008375124389347049L;
    1 usage
    public static final long TOKEN_VALIDITY = 24 * 60 * 60 * 30;
    3 usages
    @Value("${secret}")
    private String jwtSecret;

    2 usages  ButMinhNhat
    public String generateJwtToken(String userId) {
        Map<String, Object> claims = new HashMap<>();
        return Jwts.builder().setClaims(claims).setSubject(userId)
                .setIssuedAt(new Date(System.currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis() + TOKEN_VALIDITY * 1000 * 14))
                .signWith(SignatureAlgorithm.HS512, jwtSecret).compact();
    }

    1 usage  ButMinhNhat
    public Boolean validateJwtToken(String token) {
        Claims claims = Jwts.parser().setSigningKey(jwtSecret).parseClaimsJws(token).getBody();
        Boolean isTokenExpired = claims.getExpiration().before(new Date());
        return !isTokenExpired;
    }

    1 usage  ButMinhNhat
    public String getUserIDFromToken(String token) {
        final Claims claims = Jwts.parser().setSigningKey(jwtSecret).parseClaimsJws(token).getBody();
        return claims.getSubject();
    }
}
```

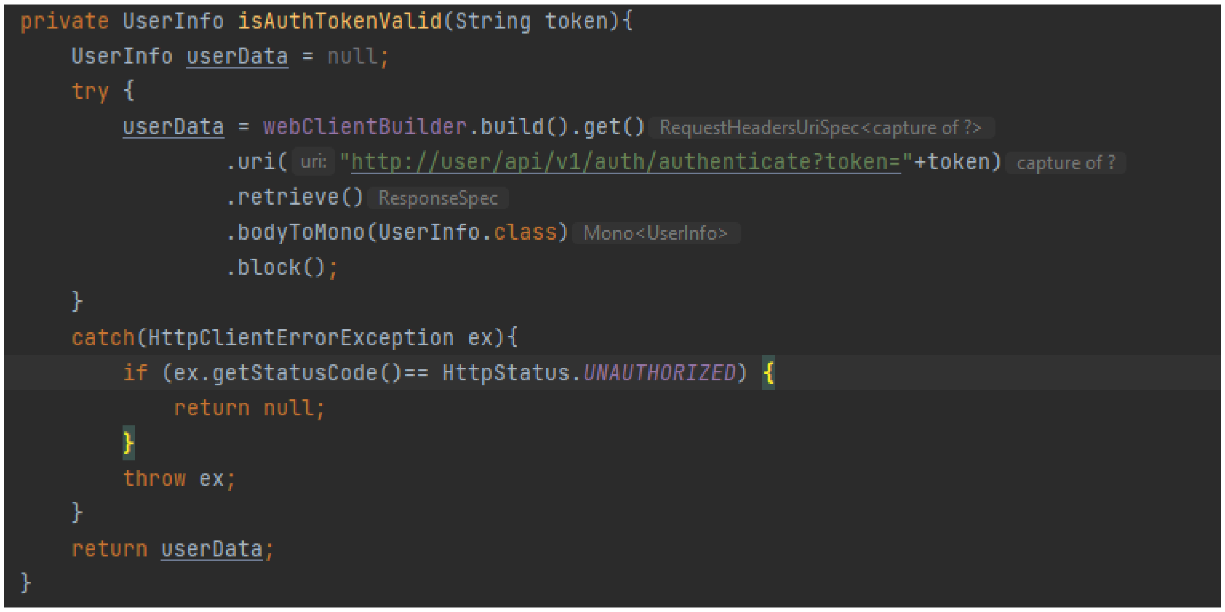
Figure 5.1.2.2 TokenManager

In the JWTAuthFilter file, the implementation allows this layer to retrieve the token from the request. Here, we can see that the token will be retrieved from the request header with the header "Authorization" and the token starts with the keyword "Bearer". Then, it will proceed to the API authentication to check the token, which will be explained below. If the API returns information, then the request will be passed down to the controller. However, if the API returns a failed result or no data, then the request will be blocked and not proceed any further (as shown in Figures 5.1.2.3 and 5.1.2.4).



```java
@Override
protected void doFilterInternal(
        HttpServletRequest request,
        HttpServletResponse response,
        FilterChain filterChain
) throws ServletException, IOException {
    final String authHeader = request.getHeader( s: "Authorization");

    if(authHeader == null || !authHeader.startsWith("Bearer")) {
        filterChain.doFilter(request, response);
        return;
    }

    if(SecurityContextHolder.getContext().getAuthentication() == null) {
        UserInfo userData = isAuthTokenValid(authHeader);
        UsernamePasswordAuthenticationToken authToken = new UsernamePasswordAuthenticationToken(
                userData,  credentials: null, userData.getAuthorities());
        authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
        SecurityContextHolder.getContext().setAuthentication(authToken);

        //set userId to header
        request.setAttribute( s: "userId",userData.getId());
    }
    filterChain.doFilter(request, response);
}
```

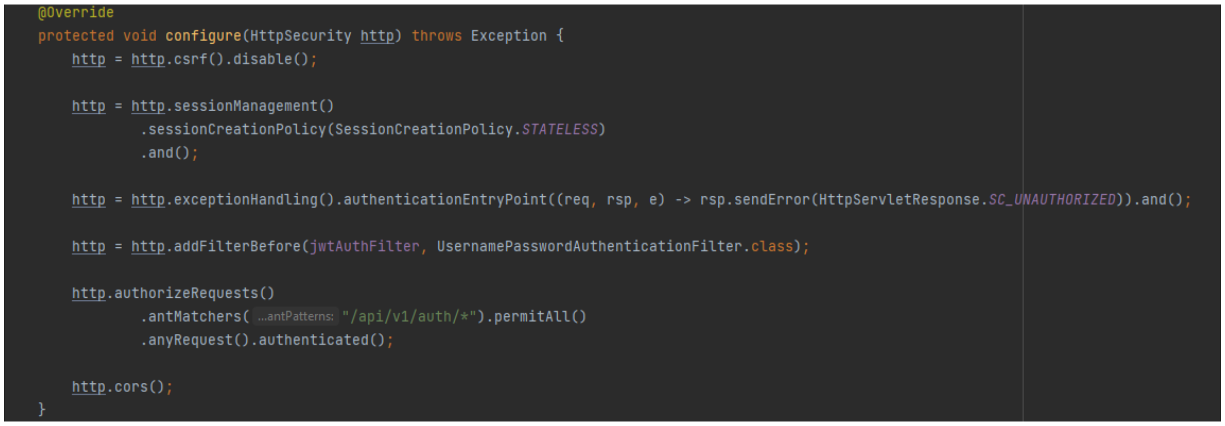
Figure 5.1.2.3: Method for retrieving the token from the request.



```java
private UserInfo isAuthTokenValid(String token){
    UserInfo userData = null;
    try {
        userData = webClientBuilder.build().get() RequestHeadersUriSpec<capture of ?>
                .uri( uri: "http://user/api/v1/auth/authenticate?token="+token) capture of ?
                .retrieve() ResponseSpec
                .bodyToMono(UserInfo.class) Mono<UserInfo>
                .block();
    }
    catch(HttpClientErrorException ex){
        if (ex.getStatusCode()== HttpStatus.UNAUTHORIZED) {
            return null;
        }
        throw ex;
    }
    return userData;
}
```

Figure 5.1.2.4: Method for checking the token via API authentication

In the SecurityConfiguration file, the settings are configured to run the security middleware, especially defining which APIs in the service will be exempted from the token authentication check. In here, the auth-related APIs are exempted from this authentication check, as these APIs are the methods that create and authenticate tokens. Details will be presented below (as shown in Figure 5.1.2.5).



```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http = http.csrf().disable();

    http = http.sessionManagement()
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            .and();

    http = http.exceptionHandling().authenticationEntryPoint((req, rsp, e) -> rsp.sendError(HttpServletResponse.SC_UNAUTHORIZED)).and();

    http = http.addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class);

    http.authorizeRequests()
            .antMatchers( ...antPatterns: "/api/v1/auth/*").permitAll()
            .anyRequest().authenticated();

    http.cors();
}
```

Figure 5.1.2.5 SecurityConfiguration

5.1.2.3 Controller (Routes)

In UserService, there are 2 controllers (routes): AuthController and UserController. These two controllers are routed as follows:

- AuthController: http://localhost:8081/api/v1/auth
- UserController: http://localhost:8081/api/v1/user

a) <u>AuthController</u>

APIs:

- POST: SignIn
- POST: SignUp
- GET: Authenticate

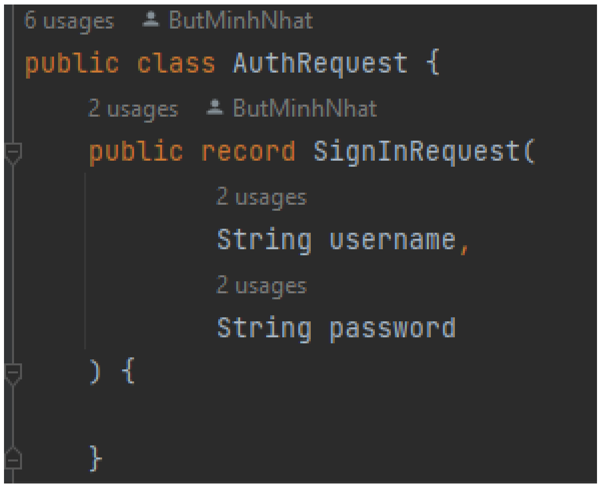
Explanation of the APIs' operations:

- SignIn: Starting with a request from the client, the client will request this API, and the API will receive the request according to the SignInRequest model (figure

5.1.2.6). Then, the request will be received by AuthController and pushed to AuthService (figure 5.1.2.7). In AuthService, the information including username and password will be used for AuthRepository to go down under the data system to check if there is data or not, if there is, AuthService will generate a token and return the information to the request, including the information of that user and the generated token. In the case of no data in the database, AuthService will return a status code of 401: Unauthorized (figure 5.1.2.8).

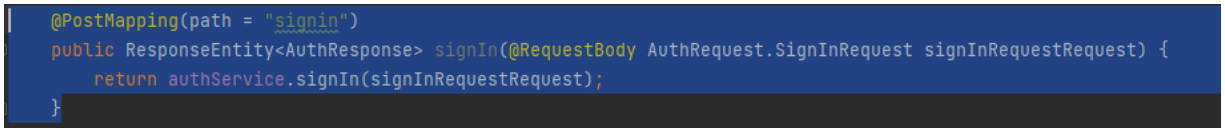
-

-



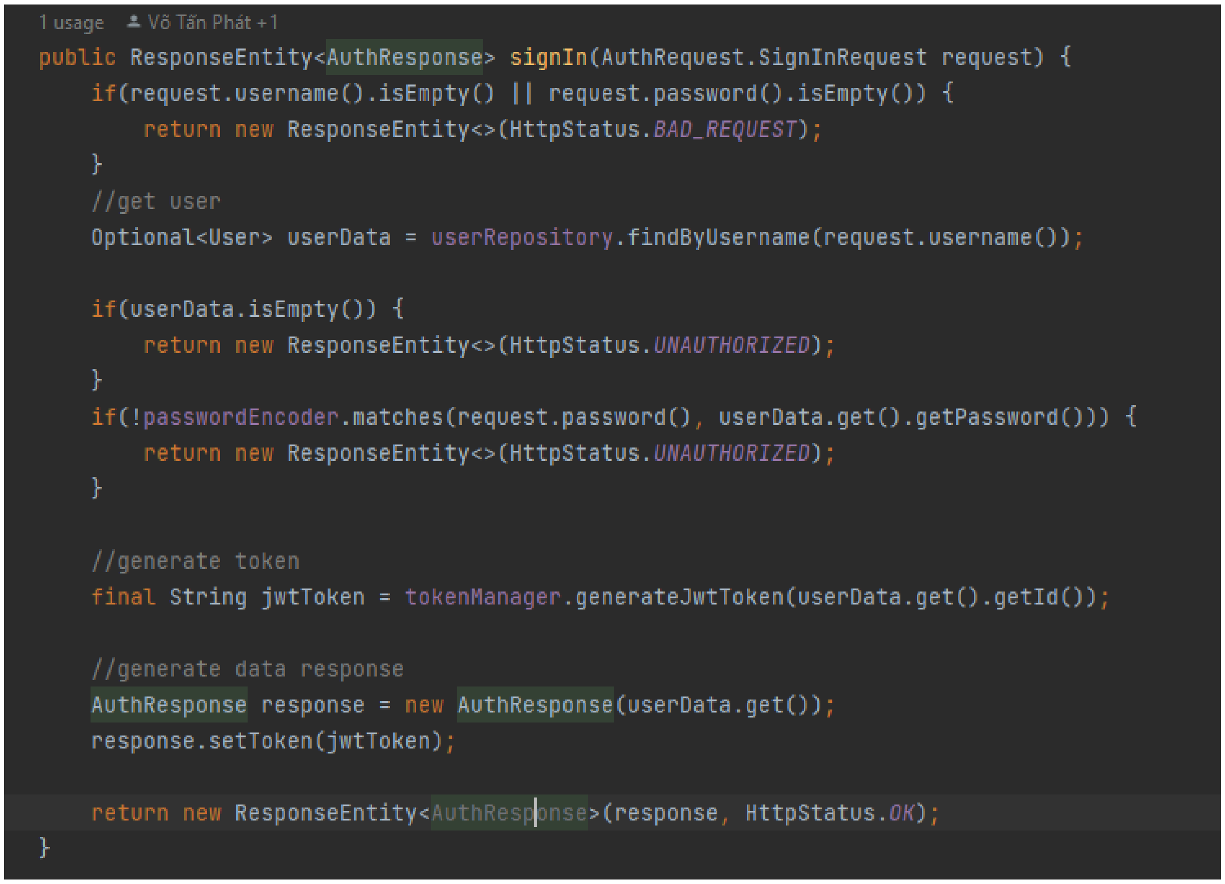
```
6 usages  ButMinhNhat
public class AuthRequest {
    2 usages  ButMinhNhat
    public record SignInRequest(
            2 usages
            String username,
            2 usages
            String password
    ) {

    }
```

Figure 5.1.2.6 Model of SignInRequest



```
@PostMapping(path = "signin")
public ResponseEntity<AuthResponse> signIn(@RequestBody AuthRequest.SignInRequest signInRequestRequest) {
    return authService.signIn(signInRequestRequest);
}
```

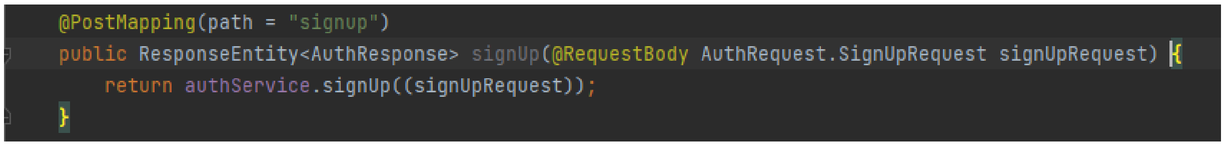
Figure 5.1.2.7 SignIn of AuthController



```
1 usage   Võ Tấn Phát +1
public ResponseEntity<AuthResponse> signIn(AuthRequest.SignInRequest request) {
    if(request.username().isEmpty() || request.password().isEmpty()) {
        return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
    }
    //get user
    Optional<User> userData = userRepository.findByUsername(request.username());

    if(userData.isEmpty()) {
        return new ResponseEntity<>(HttpStatus.UNAUTHORIZED);
    }
    if(!passwordEncoder.matches(request.password(), userData.get().getPassword())) {
        return new ResponseEntity<>(HttpStatus.UNAUTHORIZED);
    }

    //generate token
    final String jwtToken = tokenManager.generateJwtToken(userData.get().getId());

    //generate data response
    AuthResponse response = new AuthResponse(userData.get());
    response.setToken(jwtToken);

    return new ResponseEntity<AuthResponse>(response, HttpStatus.OK);
}
```

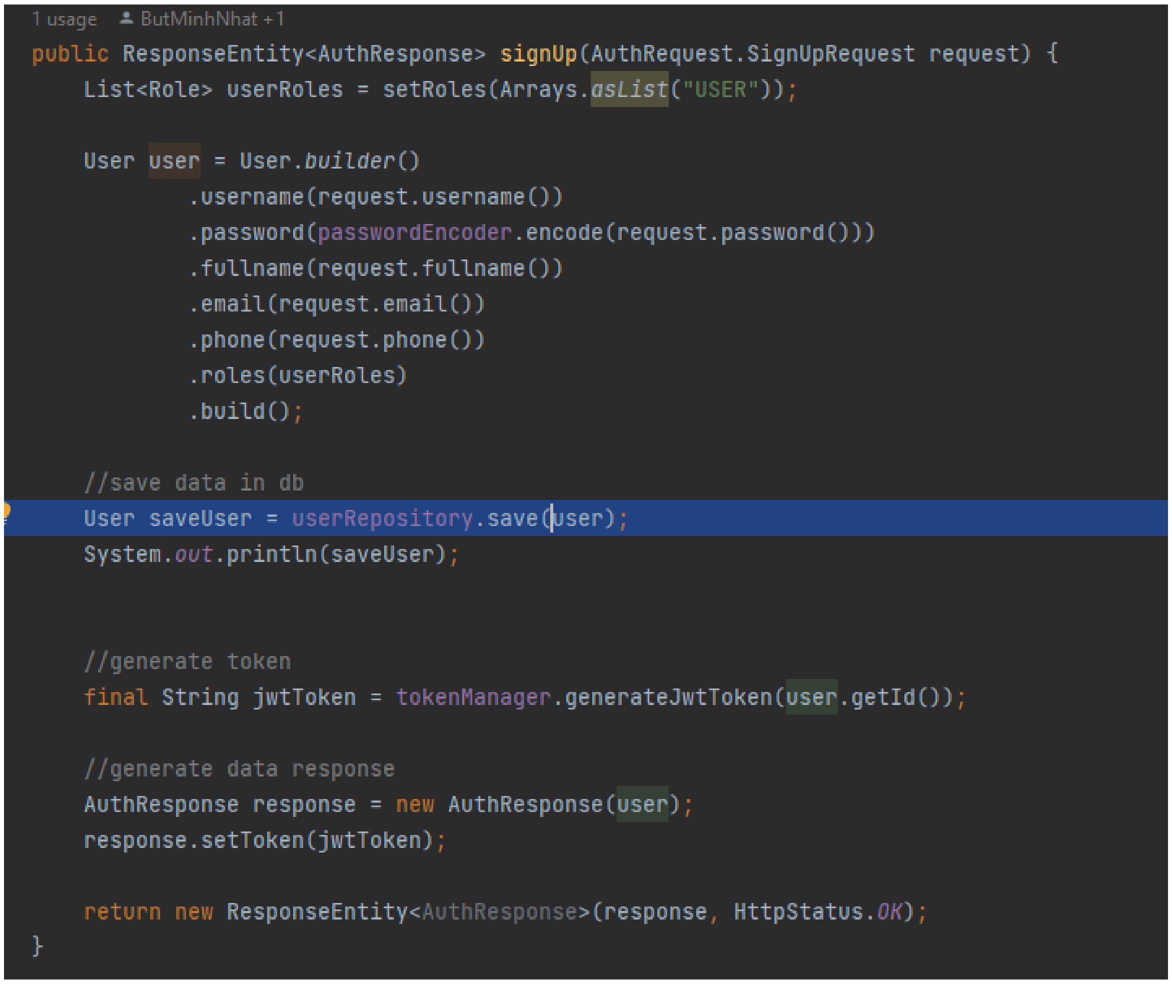
Figure 5.1.2.8 AuthService

- SignUp: Similar to the flow above, in the signUp API, the API also follows the same path as the signin API but with the difference that this API will create a new user based on the information passed in from the request. The controller receives the data and forwards it to the authService. The process of creating a new user takes place in the AuthService and is added through the UserRepository. Then this API will return the created user information and token. (Figure 5.1.2.10, 5.1.2.11)



```
@PostMapping(path = "signup")
public ResponseEntity<AuthResponse> signUp(@RequestBody AuthRequest.SignUpRequest signUpRequest) {
    return authService.signUp((signUpRequest));
}
```

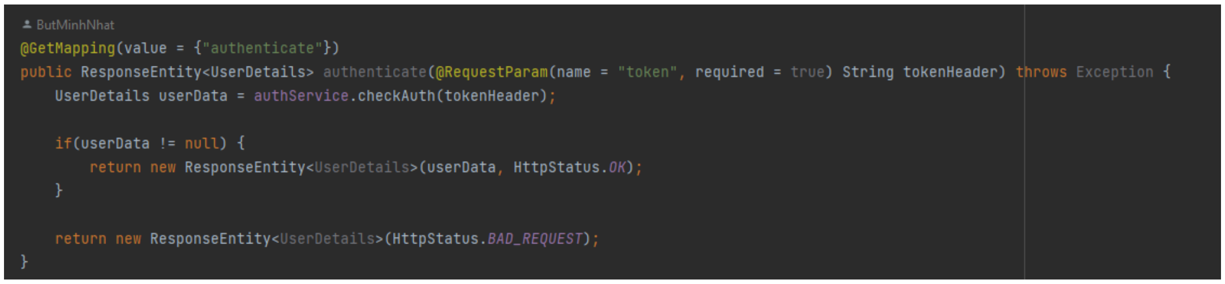
Figure 5.1.2.10 Controller passes params down to AuthService and receives result from it, then returns to client.



```
1 usage  ButMinhNhat +1
public ResponseEntity<AuthResponse> signUp(AuthRequest.SignUpRequest request) {
    List<Role> userRoles = setRoles(Arrays.asList("USER"));

    User user = User.builder()
            .username(request.username())
            .password(passwordEncoder.encode(request.password()))
            .fullname(request.fullname())
            .email(request.email())
            .phone(request.phone())
            .roles(userRoles)
            .build();

    //save data in db
    User saveUser = userRepository.save(user);
    System.out.println(saveUser);


    //generate token
    final String jwtToken = tokenManager.generateJwtToken(user.getId());

    //generate data response
    AuthResponse response = new AuthResponse(user);
    response.setToken(jwtToken);

    return new ResponseEntity<AuthResponse>(response, HttpStatus.OK);
}
```

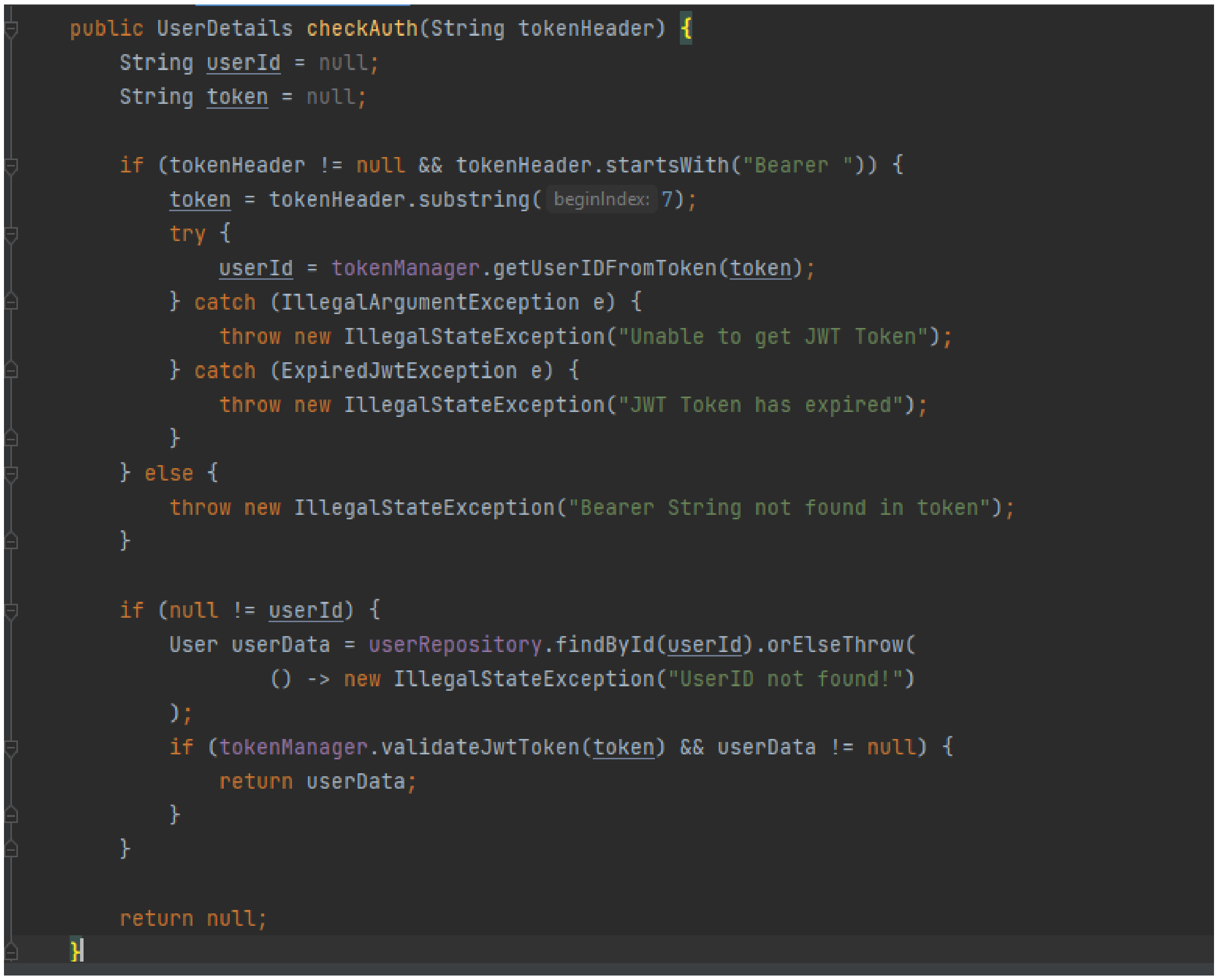
Figure 5.1.2.11 AuthService verifies and runs UserRepository to create a new user and generate a token.

Authenticate: Unlike the two APIs above used to generate tokens, this API is used to authenticate that token with requests. The API will retrieve the token in the requests from the client as well as the services. This API also has a similar flow to other APIs when it will receive requests from the Controller. The Controller will pass the params down to AuthService to perform the logic. Here, AuthService will decode the token to userId, then use this param to pass into UserRepository to find data in the database. If there is data, it will return information of that User, if not, AuthService will return null to the Controller. If the value is null, the Controller will return a 400: Bad Request status code to the client. If there is a value, the Controller will return the user information received from AuthService to the client. (Figure 5.1.2.12, Figure 5.1.2.13).



```java
ButMinhNhat
@GetMapping(value = {"authenticate"})
public ResponseEntity<UserDetails> authenticate(@RequestParam(name = "token", required = true) String tokenHeader) throws Exception {
    UserDetails userData = authService.checkAuth(tokenHeader);

    if(userData != null) {
        return new ResponseEntity<UserDetails>(userData, HttpStatus.OK);
    }

    return new ResponseEntity<UserDetails>(HttpStatus.BAD_REQUEST);
}
```

Figure 5.1.2.12 The controller receives the request and passes the parameters down to the AuthService, then returns the result to the client if the AuthService returns data, otherwise returns Bad Request to the client.



```java
public UserDetails checkAuth(String tokenHeader) {
    String userId = null;
    String token = null;

    if (tokenHeader != null && tokenHeader.startsWith("Bearer ")) {
        token = tokenHeader.substring( beginIndex: 7);
        try {
            userId = tokenManager.getUserIDFromToken(token);
        } catch (IllegalArgumentException e) {
            throw new IllegalStateException("Unable to get JWT Token");
        } catch (ExpiredJwtException e) {
            throw new IllegalStateException("JWT Token has expired");
        }
    } else {
        throw new IllegalStateException("Bearer String not found in token");
    }

    if (null != userId) {
        User userData = userRepository.findById(userId).orElseThrow(
                () -> new IllegalStateException("UserID not found!")
        );
        if (tokenManager.validateJwtToken(token) && userData != null) {
            return userData;
        }
    }

    return null;
}
```

Figure 5.1.2.13 AuthService decodes token to userId and check user exists or not via userRepository

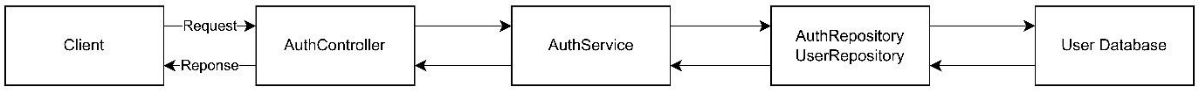


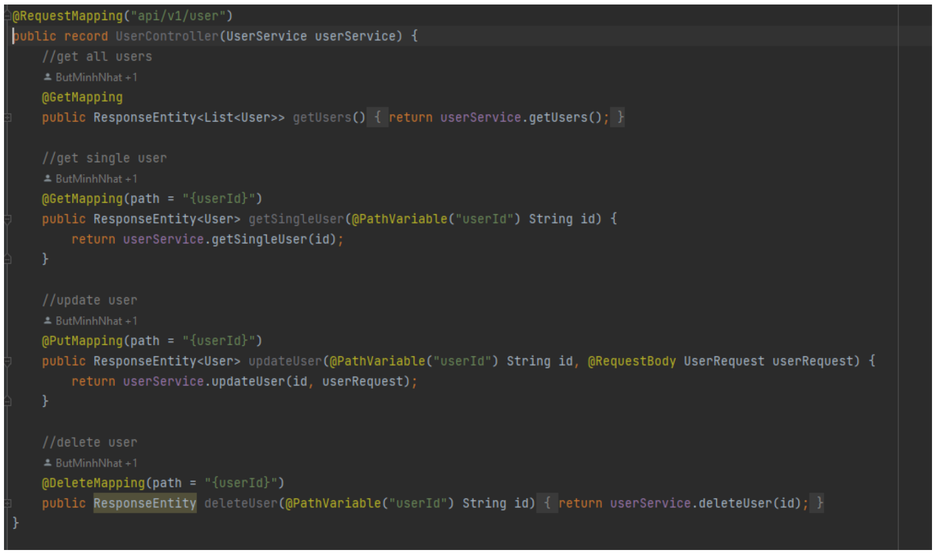
Figure 5.1.2.14 Flow Auth APIs

b) *<u>UserController</u>*

APIs:

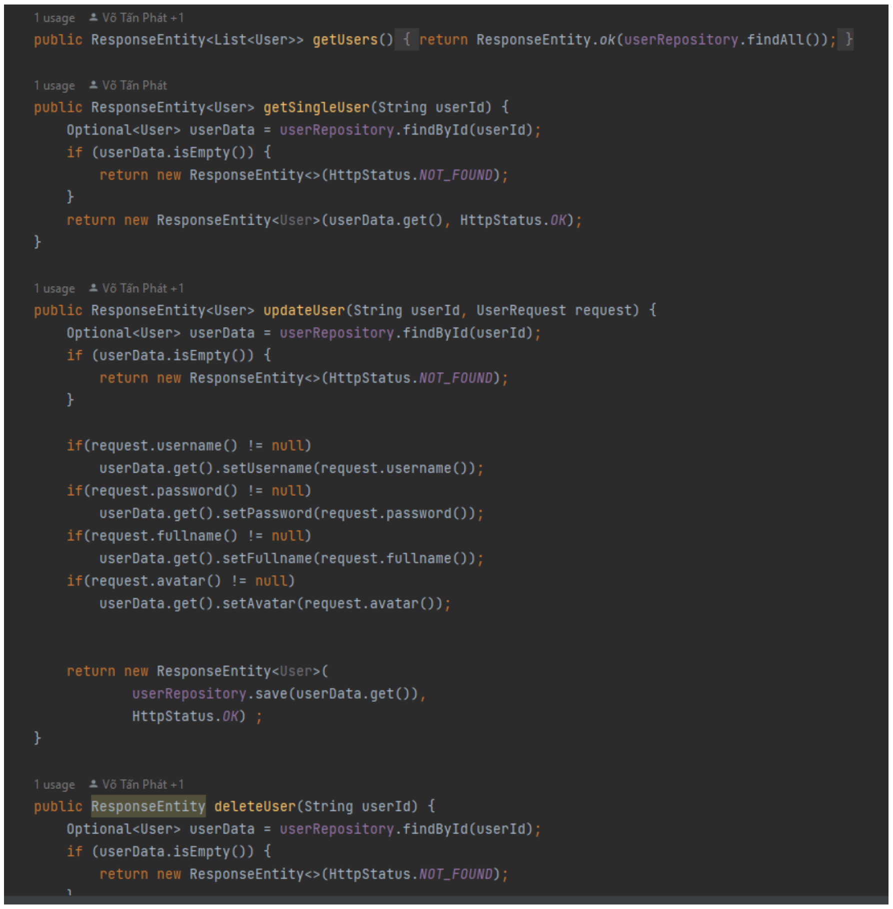
- GET: GetUsers
- GET: GetUser
- PUT: UpdateUser
- DELETE: DeleteUser

Flow from the time the API is executed until the result is received as a User is similar to Auth, but the difference here is that the APIs belonging to the User must go through the security middleware. As explained above, this middleware will look in the request to these APIs to get the "Authorization" header to check the token through the checkAuth method in AuthService. If the token is not available or cannot be authenticated. The request will not be continued and will return Bad Request. Otherwise, the request will go to the Controller. From here, the request of the request will be continued by the UserController to navigate to the UserService layer, the UserService will interact with the data in the database through the UserRepository, to be able to make the request and return the desired result (Figure 5.1.2.15, 5.1.2.16, 5.1.2.17)



```java
@RequestMapping("api/v1/user")
public record UserController(UserService userService) {
    //get all users
    ButMinhNhat +1
    @GetMapping
    public ResponseEntity<List<User>> getUsers() { return userService.getUsers(); }

    //get single user
    ButMinhNhat +1
    @GetMapping(path = "{userId}")
    public ResponseEntity<User> getSingleUser(@PathVariable("userId") String id) {
        return userService.getSingleUser(id);
    }

    //update user
    ButMinhNhat +1
    @PutMapping(path = "{userId}")
    public ResponseEntity<User> updateUser(@PathVariable("userId") String id, @RequestBody UserRequest userRequest) {
        return userService.updateUser(id, userRequest);
    }

    //delete user
    ButMinhNhat +1
    @DeleteMapping(path = "{userId}")
    public ResponseEntity deleteUser(@PathVariable("userId") String id) { return userService.deleteUser(id); }
}
```

Figure 5.1.2.15 The APIs in User are in the controller



```java
1 usage   Võ Tấn Phát +1
public ResponseEntity<List<User>> getUsers() { return ResponseEntity.ok(userRepository.findAll()); }

1 usage   Võ Tấn Phát
public ResponseEntity<User> getSingleUser(String userId) {
    Optional<User> userData = userRepository.findById(userId);
    if (userData.isEmpty()) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    return new ResponseEntity<User>(userData.get(), HttpStatus.OK);
}

1 usage   Võ Tấn Phát +1
public ResponseEntity<User> updateUser(String userId, UserRequest request) {
    Optional<User> userData = userRepository.findById(userId);
    if (userData.isEmpty()) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }

    if(request.username() != null)
        userData.get().setUsername(request.username());
    if(request.password() != null)
        userData.get().setPassword(request.password());
    if(request.fullname() != null)
        userData.get().setFullname(request.fullname());
    if(request.avatar() != null)
        userData.get().setAvatar(request.avatar());


    return new ResponseEntity<User>(
            userRepository.save(userData.get()),
            HttpStatus.OK) ;
}

1 usage   Võ Tấn Phát +1
public ResponseEntity deleteUser(String userId) {
    Optional<User> userData = userRepository.findById(userId);
    if (userData.isEmpty()) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
```

Figure 5.1.2.16 Methods included in UserService to perform logic to return results via UserRepository

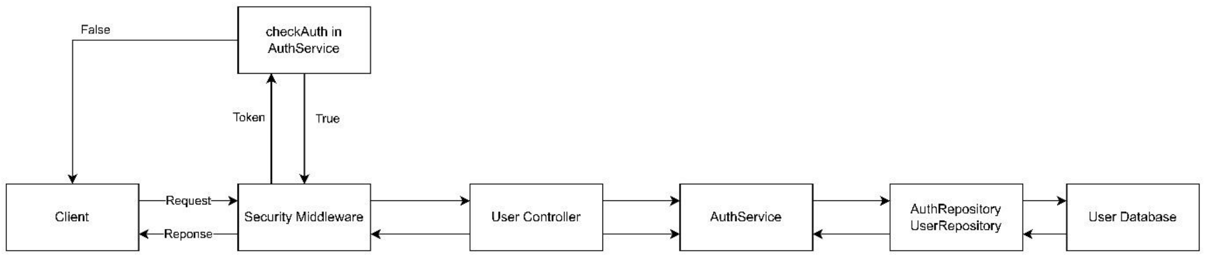
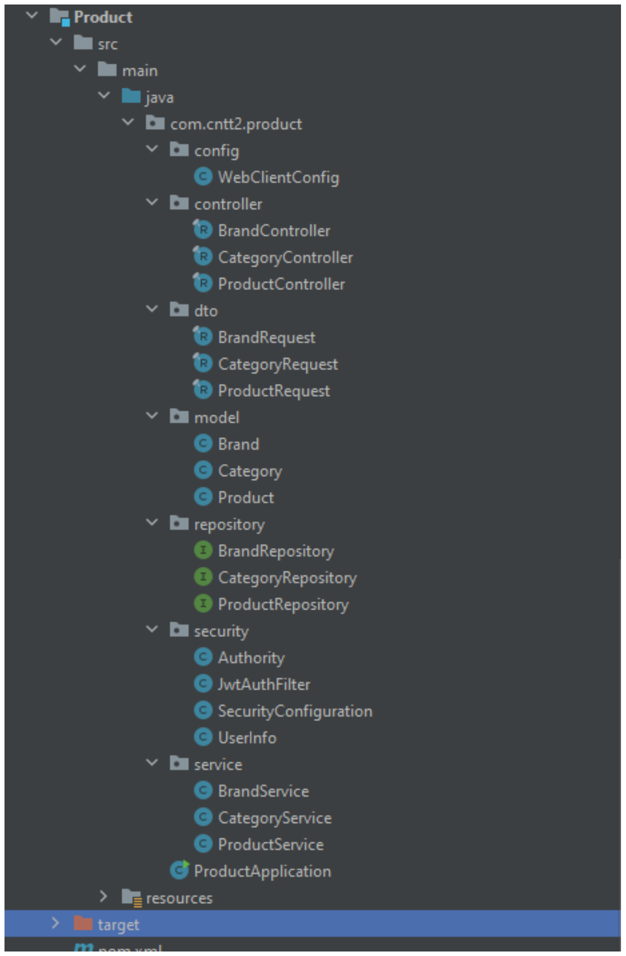


Figure 5.1.2.17 Flow of requests to User

5.1.3 Product Service

Similar to UserService, Product Service is organized according to DDD and is divided into 3 parts: Product, Category and Brand (Figure 5.1.3.1).



```
Product
  src
    main
      java
        com.cntt2.product
          config
            WebClientConfig
          controller
            BrandController
            CategoryController
            ProductController
          dto
            BrandRequest
            CategoryRequest
            ProductRequest
          model
            Brand
            Category
            Product
          repository
            BrandRepository
            CategoryRepository
            ProductRepository
          security
            Authority
            JwtAuthFilter
            SecurityConfiguration
            UserInfo
          service
            BrandService
            CategoryService
            ProductService
          ProductApplication
      resources
  target
  pom.xml
```

Figure 5.1.3.1 Organizational structure of Product Service

5.1.3.1 Model

In ProductService there are four models, detailing the models:

- Product: contains information of products
- Category: contains information of categories
- Brand: contains information of brands
- Product_images: contains information about which images the product has

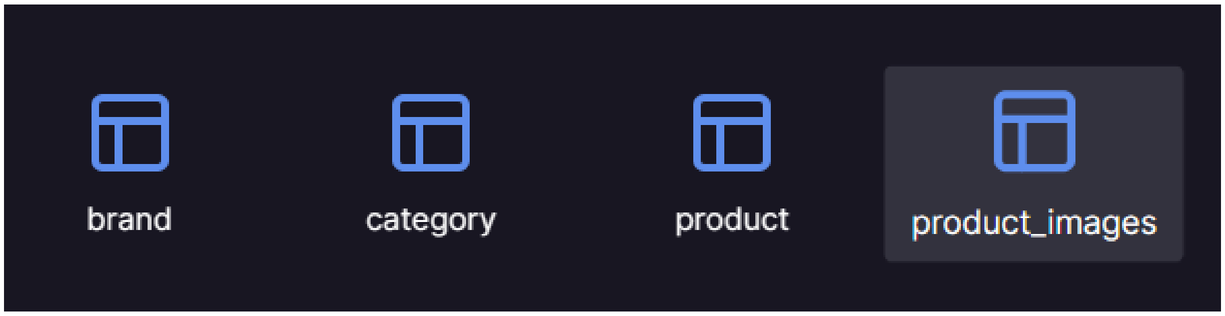
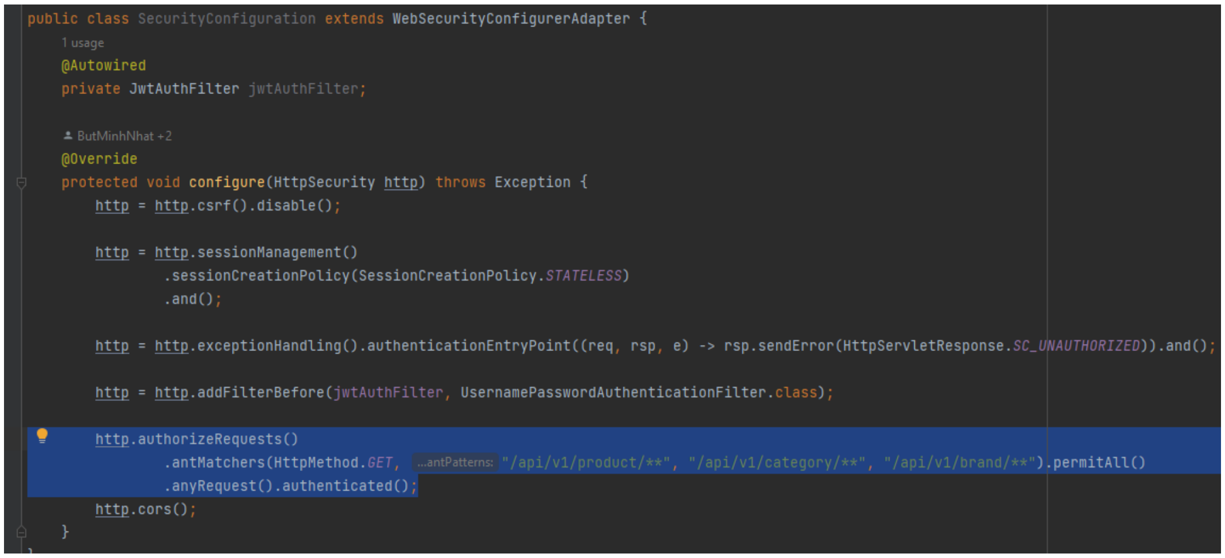


Figure 5.1.3.2 Models in ProductService

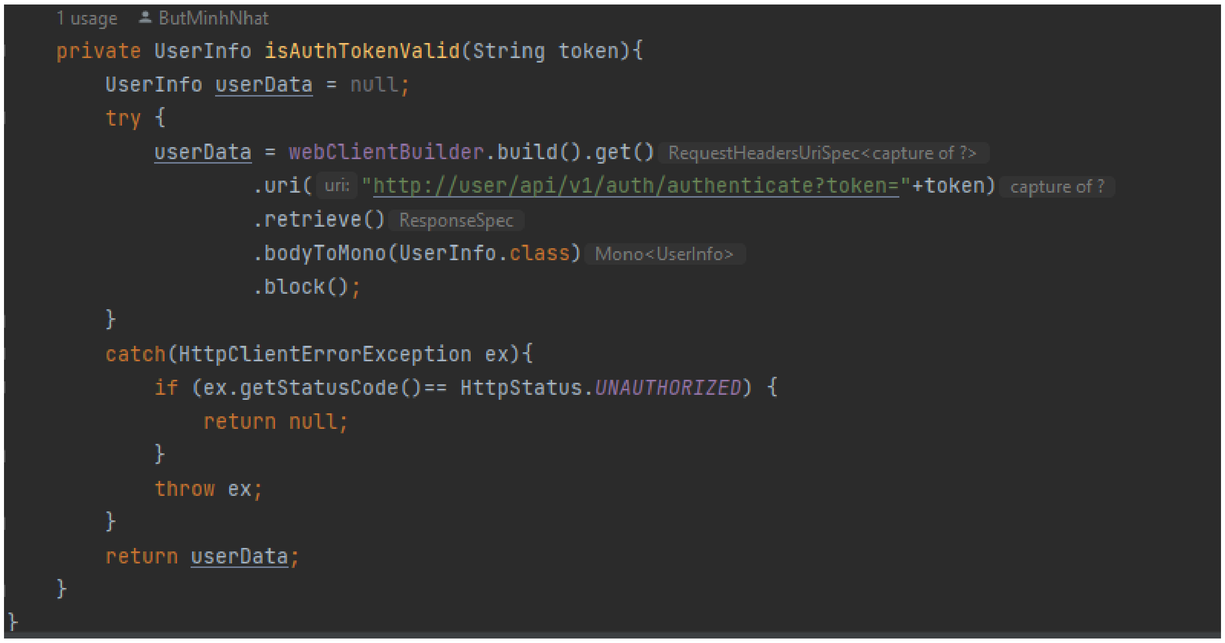
5.1.3.2 Security

Similar to UserService, ProductService also has security middleware. The ProductService also uses JWTAuthFilter to be able to get the token from the request header in the token through its own AuthService class, in the ProductService, this security middleware must pass the request through the API Gateway to go to the authenticate API in the UserSevice (Figure 5.1.3.3.). The request will be authenticated at this API and return the result. Security middleware in ProductService will receive response from UserService's authenticate API. From here, like the old logic in UserService, if the token is successfully authenticated, the request will be sent to the Controllers, otherwise the request will be blocked (Figure 5.1.3.4).

Also, APIs get in ProductService do not need token authentication, this rule is configured in SecurityConfiguration (Figure 5.1.3.5).

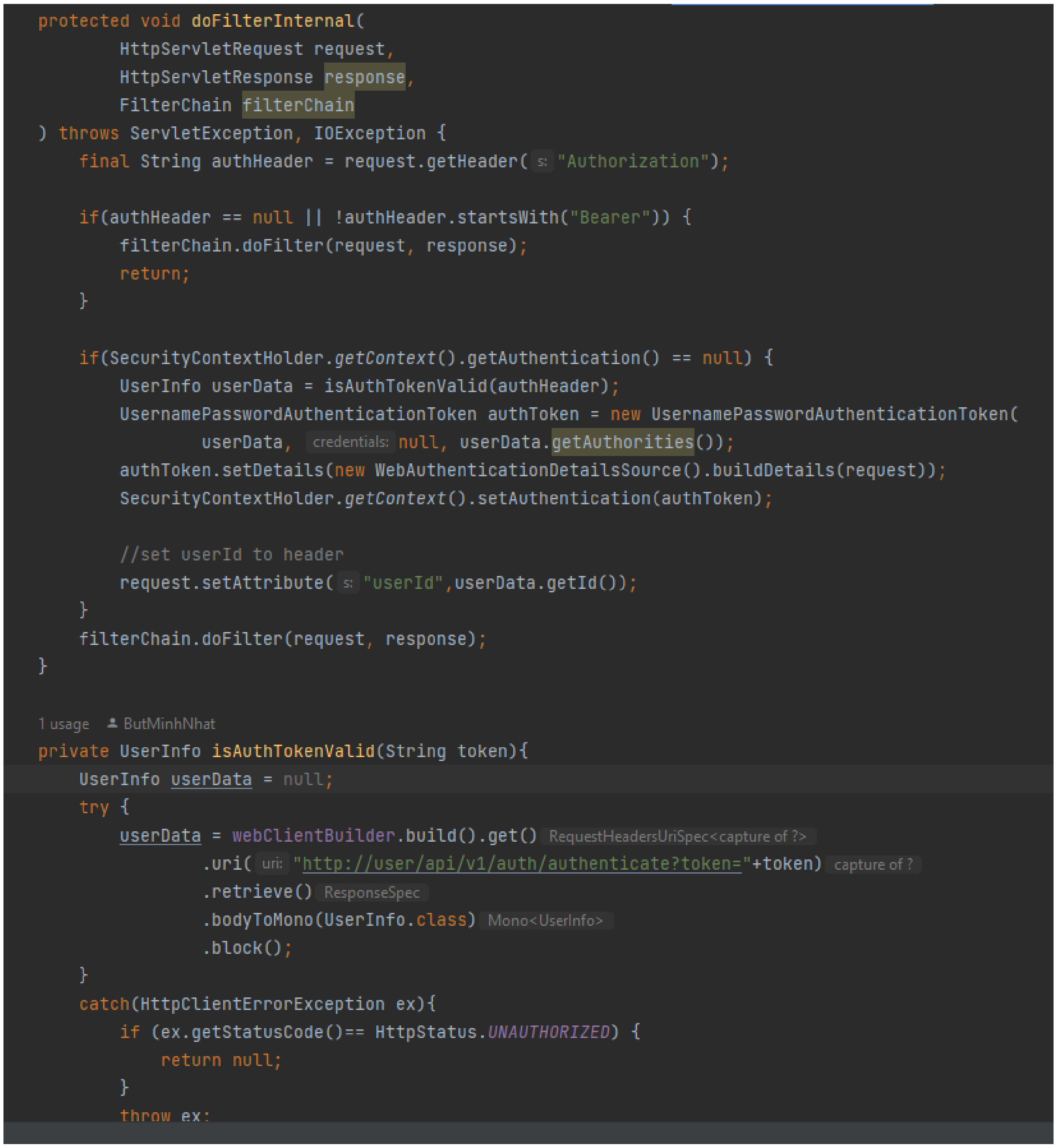


```
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
    1 usage
    @Autowired
    private JwtAuthFilter jwtAuthFilter;

    ButMinhNhat +2
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http = http.csrf().disable();

        http = http.sessionManagement()
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                .and();

        http = http.exceptionHandling().authenticationEntryPoint((req, rsp, e) -> rsp.sendError(HttpServletResponse.SC_UNAUTHORIZED)).and();

        http = http.addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class);

        http.authorizeRequests()
                .antMatchers(HttpMethod.GET,  ...antPatterns: "/api/v1/product/**", "/api/v1/category/**", "/api/v1/brand/**").permitAll()
                .anyRequest().authenticated();
        http.cors();
    }
```



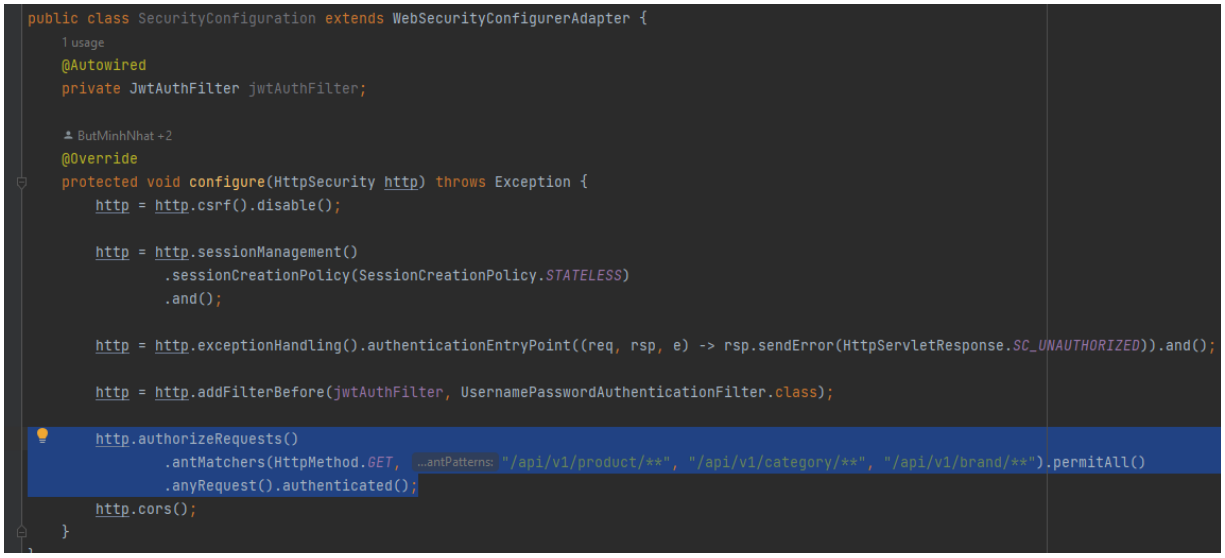
```
    1 usage   ButMinhNhat
    private UserInfo isAuthTokenValid(String token){
        UserInfo userData = null;
        try {
            userData = webClientBuilder.build().get() RequestHeadersUriSpec<capture of ?>
                    .uri( uri: "http://user/api/v1/auth/authenticate?token="+token) capture of ?
                    .retrieve() ResponseSpec
                    .bodyToMono(UserInfo.class) Mono<UserInfo>
                    .block();
        }
        catch(HttpClientErrorException ex){
            if (ex.getStatusCode()== HttpStatus.UNAUTHORIZED) {
                return null;
            }
            throw ex;
        }
        return userData;
    }
}
```

Figure 5.1.3.3 Middleware navigates to UserService to validate token



```
protected void doFilterInternal(
        HttpServletRequest request,
        HttpServletResponse response,
        FilterChain filterChain
) throws ServletException, IOException {
    final String authHeader = request.getHeader( s: "Authorization");

    if(authHeader == null || !authHeader.startsWith("Bearer")) {
        filterChain.doFilter(request, response);
        return;
    }

    if(SecurityContextHolder.getContext().getAuthentication() == null) {
        UserInfo userData = isAuthTokenValid(authHeader);
        UsernamePasswordAuthenticationToken authToken = new UsernamePasswordAuthenticationToken(
                userData,  credentials: null, userData.getAuthorities());
        authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
        SecurityContextHolder.getContext().setAuthentication(authToken);

        //set userId to header
        request.setAttribute( s: "userId",userData.getId());
    }
    filterChain.doFilter(request, response);
}

1 usage   ButMinhNhat
private UserInfo isAuthTokenValid(String token){
    UserInfo userData = null;
    try {
        userData = webClientBuilder.build().get() RequestHeadersUriSpec<capture of ?>
                .uri( uri: "http://user/api/v1/auth/authenticate?token="+token) capture of ?
                .retrieve() ResponseSpec
                .bodyToMono(UserInfo.class) Mono<UserInfo>
                .block();
    }
    catch(HttpClientErrorException ex){
        if (ex.getStatusCode()== HttpStatus.UNAUTHORIZED) {
            return null;
        }
        throw ex;
```

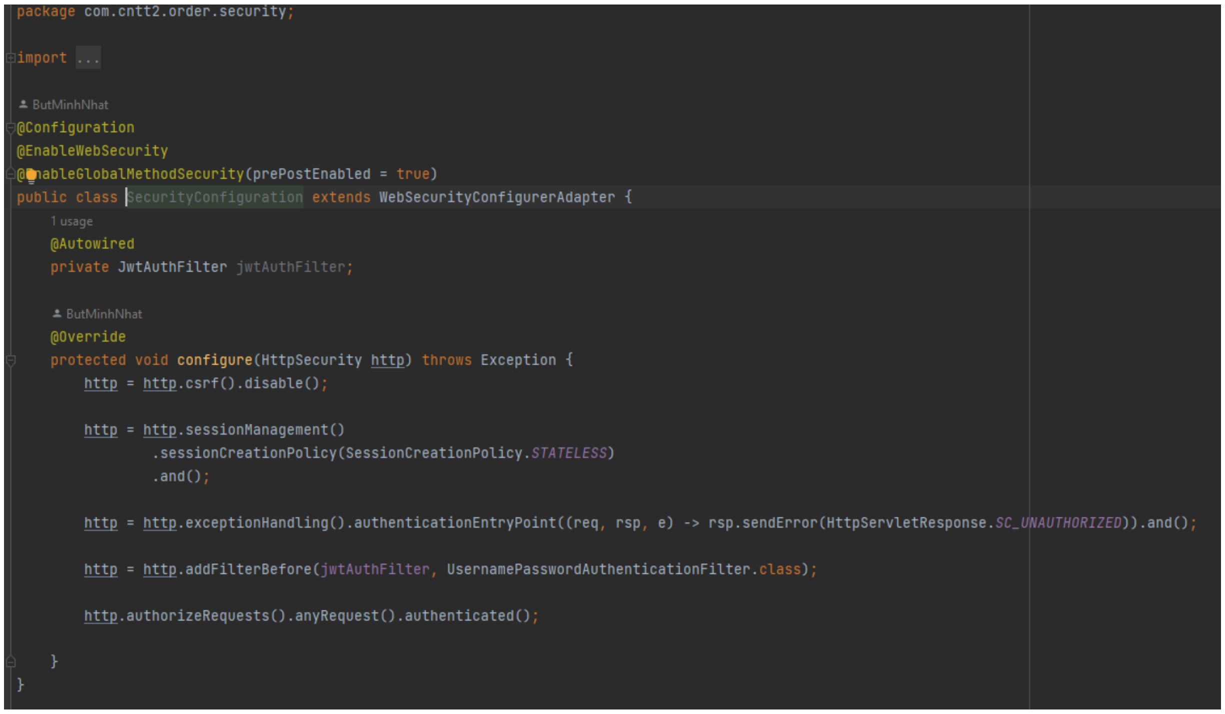
Figure 5.1.3.4 Security middleware



```
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
    1 usage
    @Autowired
    private JwtAuthFilter jwtAuthFilter;

    ButMinhNhat +2
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http = http.csrf().disable();

        http = http.sessionManagement()
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                .and();

        http = http.exceptionHandling().authenticationEntryPoint((req, rsp, e) -> rsp.sendError(HttpServletResponse.SC_UNAUTHORIZED)).and();

        http = http.addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class);

        http.authorizeRequests()
                .antMatchers(HttpMethod.GET, ...antPatterns: "/api/v1/product/**", "/api/v1/category/**", "/api/v1/brand/**").permitAll()
                .anyRequest().authenticated();
        http.cors();
    }
```

Figure 5.1.3.5 Security Configuration

).



```
package com.cntt2.order.security;

import ...

ButMinhNhat
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
    1 usage
    @Autowired
    private JwtAuthFilter jwtAuthFilter;

    ButMinhNhat
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http = http.csrf().disable();

        http = http.sessionManagement()
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                .and();

        http = http.exceptionHandling().authenticationEntryPoint((req, rsp, e) -> rsp.sendError(HttpServletResponse.SC_UNAUTHORIZED)).and();

        http = http.addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class);

        http.authorizeRequests().anyRequest().authenticated();

    }
}
```

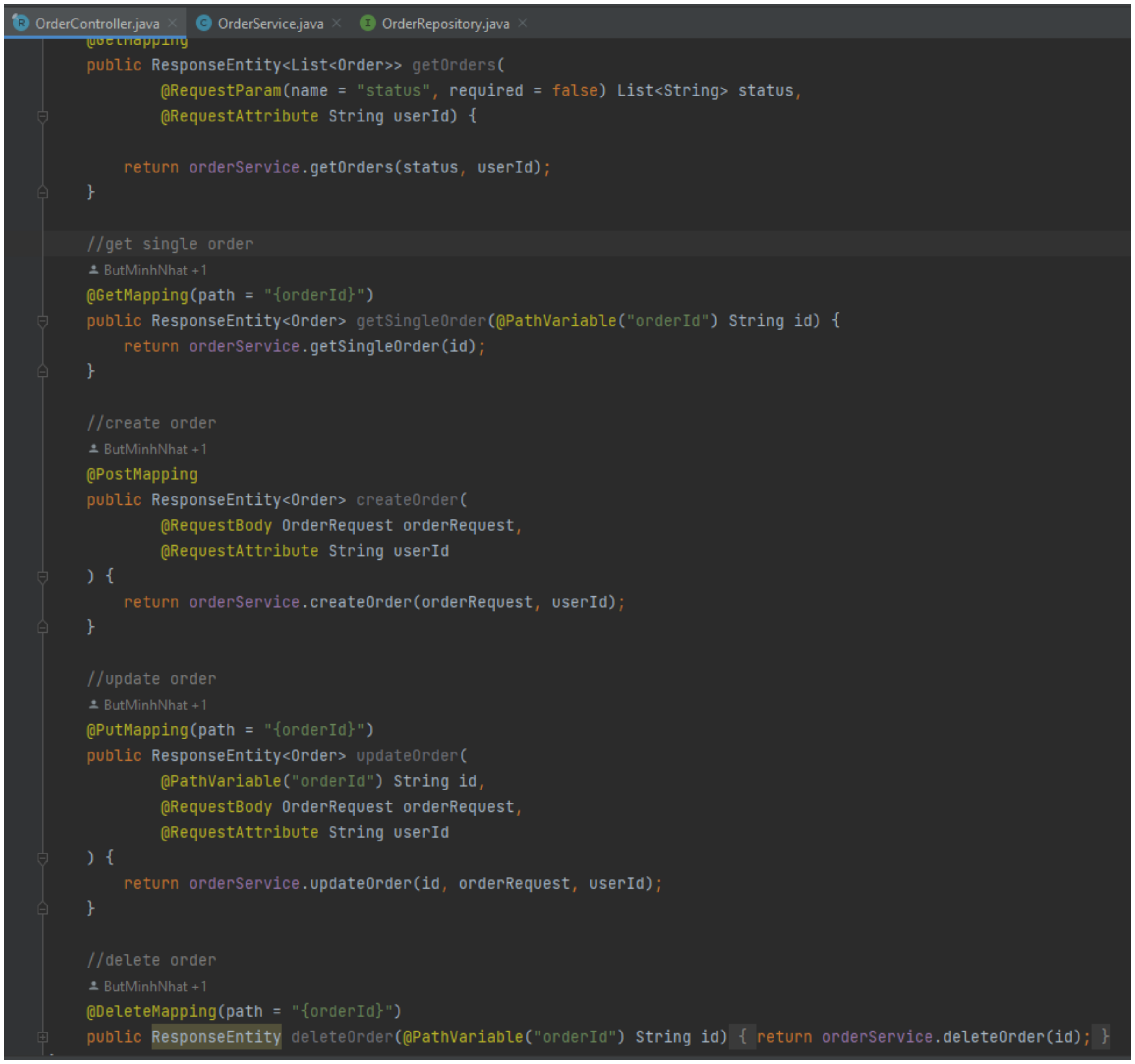
Figure 5.1.4.3 SecurityConfiguration

5.1.4 Order Service

5.1.4.3 Controller (Routes)

APIs:

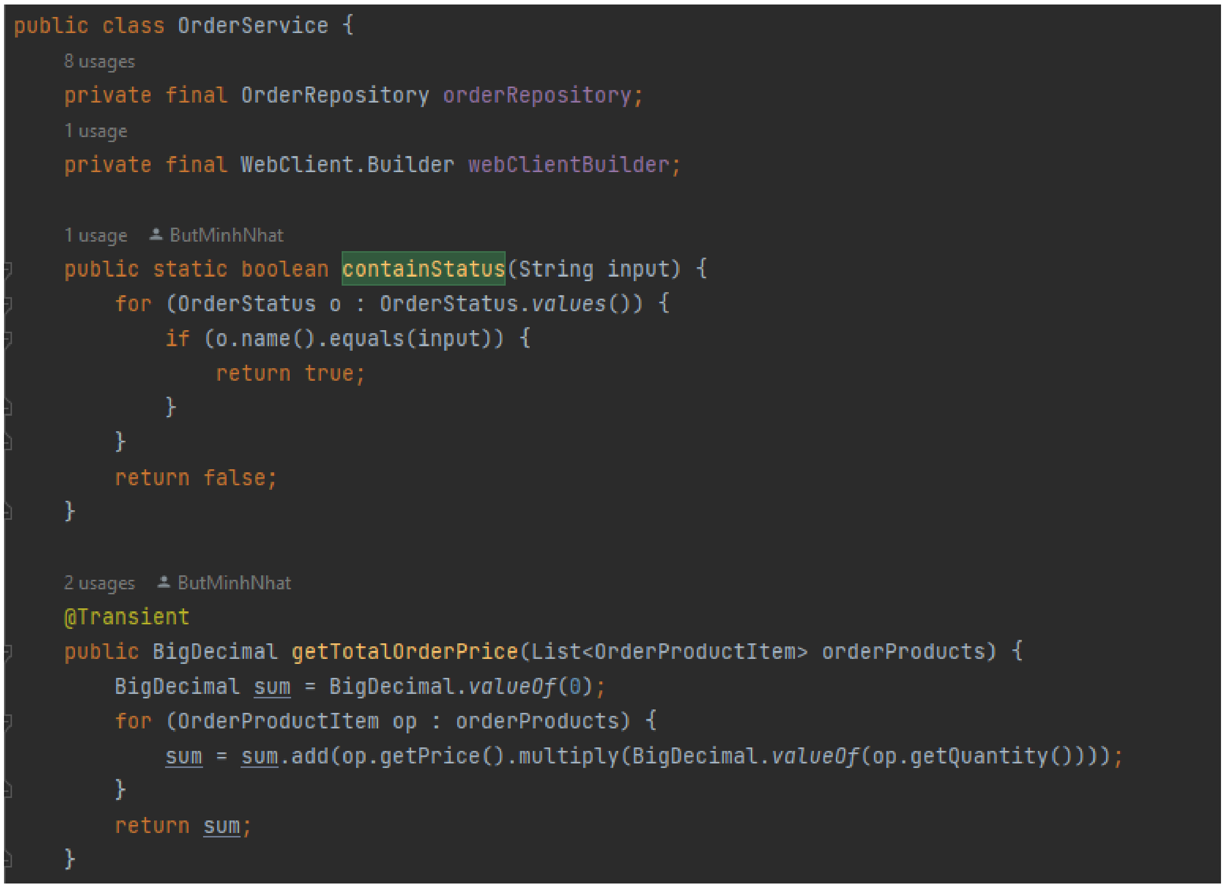
- GET: GetOrders
- GET:GetOrder
- POST: CreateOrder
- PUT: UpdateOrder
- DELETE: DeleteOrder



```java
    @GetMapping
    public ResponseEntity<List<Order>> getOrders(
            @RequestParam(name = "status", required = false) List<String> status,
            @RequestAttribute String userId) {

        return orderService.getOrders(status, userId);
    }

    //get single order
    @GetMapping(path = "{orderId}")
    public ResponseEntity<Order> getSingleOrder(@PathVariable("orderId") String id) {
        return orderService.getSingleOrder(id);
    }

    //create order
    @PostMapping
    public ResponseEntity<Order> createOrder(
            @RequestBody OrderRequest orderRequest,
            @RequestAttribute String userId
    ) {
        return orderService.createOrder(orderRequest, userId);
    }

    //update order
    @PutMapping(path = "{orderId}")
    public ResponseEntity<Order> updateOrder(
            @PathVariable("orderId") String id,
            @RequestBody OrderRequest orderRequest,
            @RequestAttribute String userId
    ) {
        return orderService.updateOrder(id, orderRequest, userId);
    }

    //delete order
    @DeleteMapping(path = "{orderId}")
    public ResponseEntity deleteOrder(@PathVariable("orderId") String id) { return orderService.deleteOrder(id); }
```

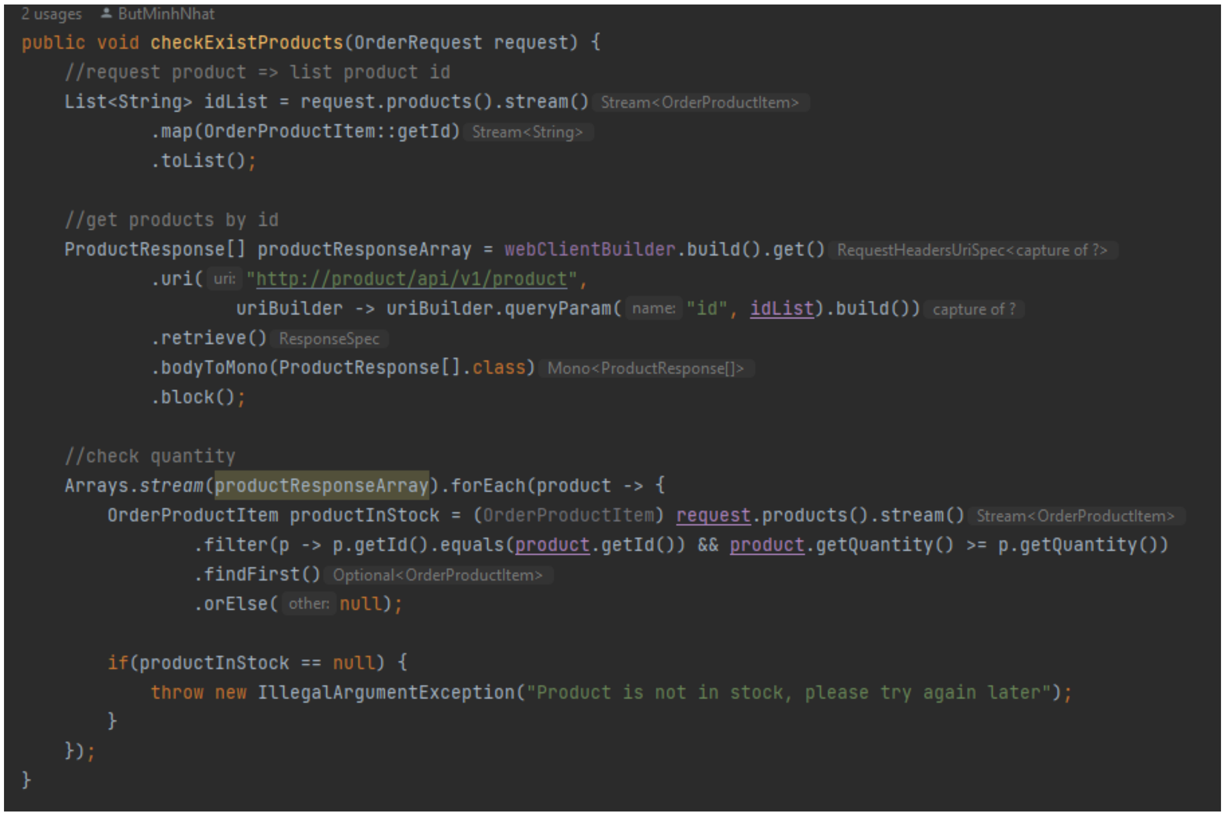
Figure 5.1.4.4 OrderController

The methods in this OrderController just call its next method in the OrderService (Figure 5.1.4.4). However, in OrderService, there are three other methods: containsStatus, getTotalPrice (Figure 5.1.4.5) and checkExistProducts (Figure 5.1.4.6).



```
public class OrderService {
    8 usages
    private final OrderRepository orderRepository;
    1 usage
    private final WebClient.Builder webClientBuilder;

    1 usage   ButMinhNhat
    public static boolean containStatus(String input) {
        for (OrderStatus o : OrderStatus.values()) {
            if (o.name().equals(input)) {
                return true;
            }
        }
        return false;
    }

    2 usages   ButMinhNhat
    @Transient
    public BigDecimal getTotalOrderPrice(List<OrderProductItem> orderProducts) {
        BigDecimal sum = BigDecimal.valueOf(0);
        for (OrderProductItem op : orderProducts) {
            sum = sum.add(op.getPrice().multiply(BigDecimal.valueOf(op.getQuantity())));
        }
        return sum;
    }
```

Figure 5.1.4.5 containStatus and getTotalPrice

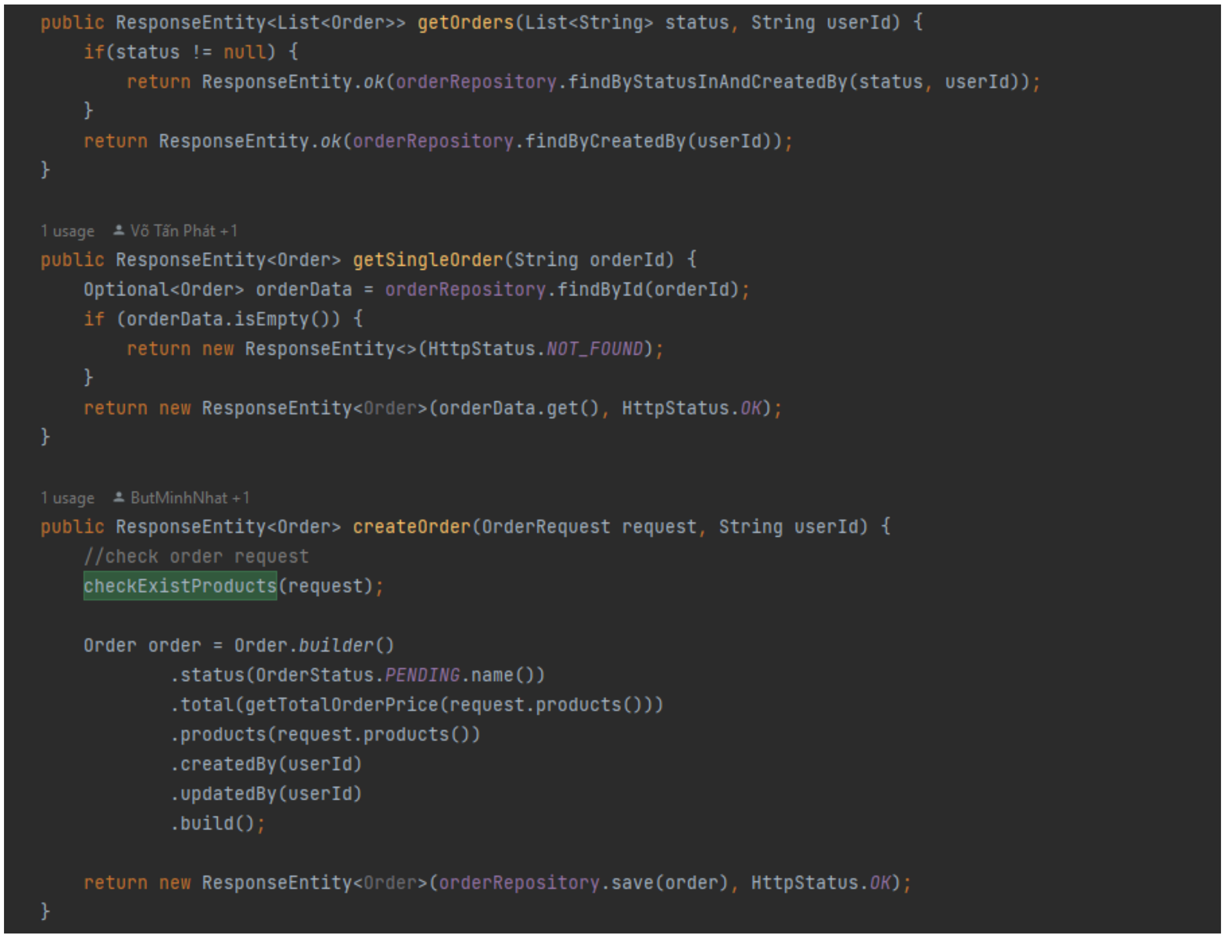


```java
2 usages  ButMinhNhat
public void checkExistProducts(OrderRequest request) {
    //request product => list product id
    List<String> idList = request.products().stream() Stream<OrderProductItem>
            .map(OrderProductItem::getId) Stream<String>
            .toList();

    //get products by id
    ProductResponse[] productResponseArray = webClientBuilder.build().get() RequestHeadersUriSpec<capture of ?>
            .uri( uri: "http://product/api/v1/product",
                    uriBuilder -> uriBuilder.queryParam( name: "id", idList).build()) capture of ?
            .retrieve() ResponseSpec
            .bodyToMono(ProductResponse[].class) Mono<ProductResponse[]>
            .block();

    //check quantity
    Arrays.stream(productResponseArray).forEach(product -> {
        OrderProductItem productInStock = (OrderProductItem) request.products().stream() Stream<OrderProductItem>
                .filter(p -> p.getId().equals(product.getId()) && product.getQuantity() >= p.getQuantity())
                .findFirst() Optional<OrderProductItem>
                .orElse( other: null);

        if(productInStock == null) {
            throw new IllegalArgumentException("Product is not in stock, please try again later");
        }
    });
}
```

Figure 5.1.4.6 checkExistProducts

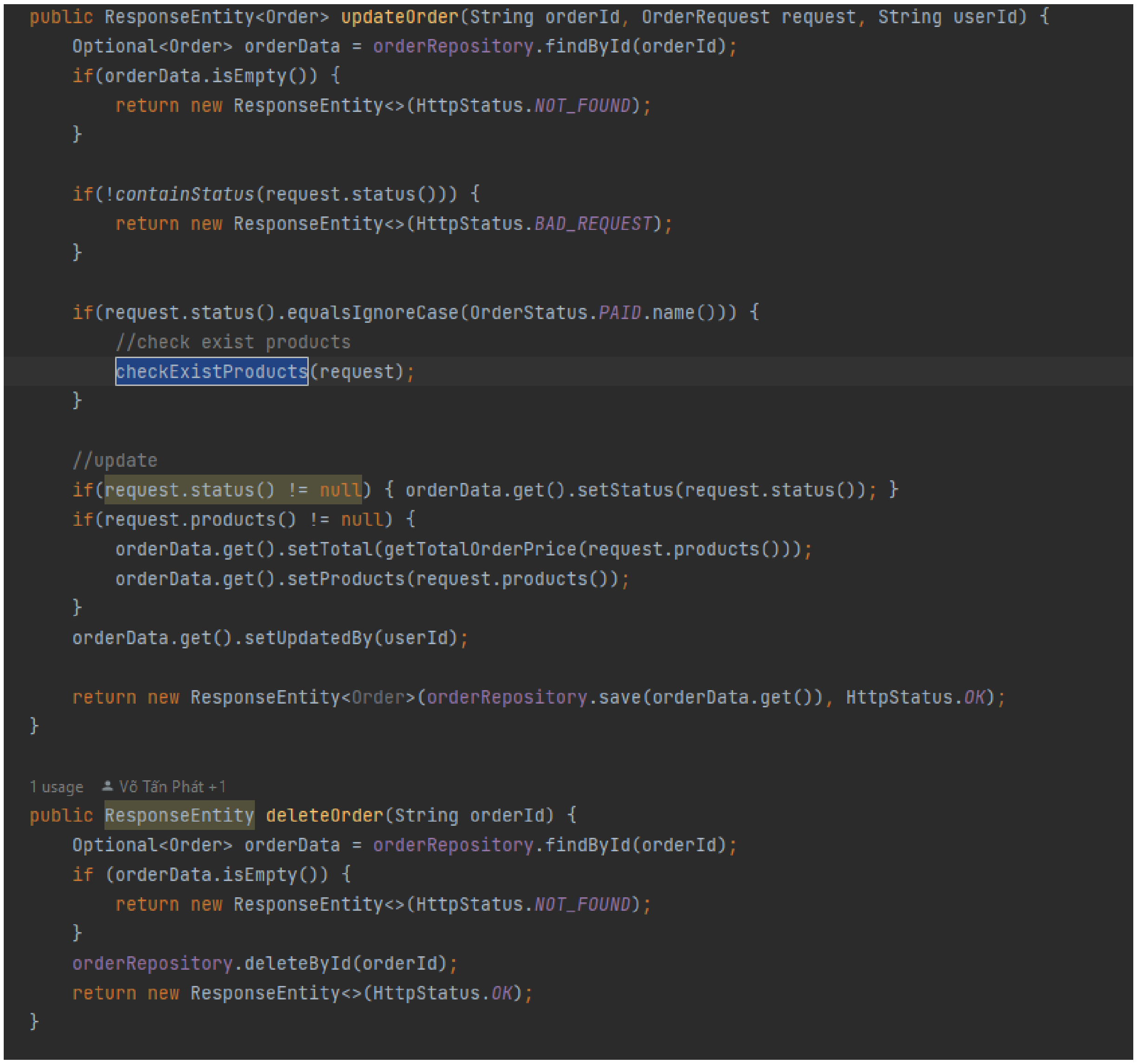
The two methods containsStatus and getTotalPrice are simply small functions implemented for main methods. But the checkExistProducts method is a special function. It is used to check whether the Product exists or not and whether the quantity of that product is still available. This method calls the ProductService's getsingleproduct API to get information about the product to be searched based on the product-slug this method is passed in. If the product is found, the information is received from the ProductService, it will further check whether the product needs the quantity or not? If there is a product and quantity is still available, it will continue, otherwise, it will immediately stop the request and return an error with the message "Product is not in stock, please try again later." This method is used in two requests, createOrder and updateOrder. When creating and modifying an Order, the OrderService will need to interact with the ProductService to get product information and based on that, the

OrderService will continue to process logic and interact with the database through the OrderRepository. (Figure 5.1.4.7, 5.1.4.8, 5.1.4.9)



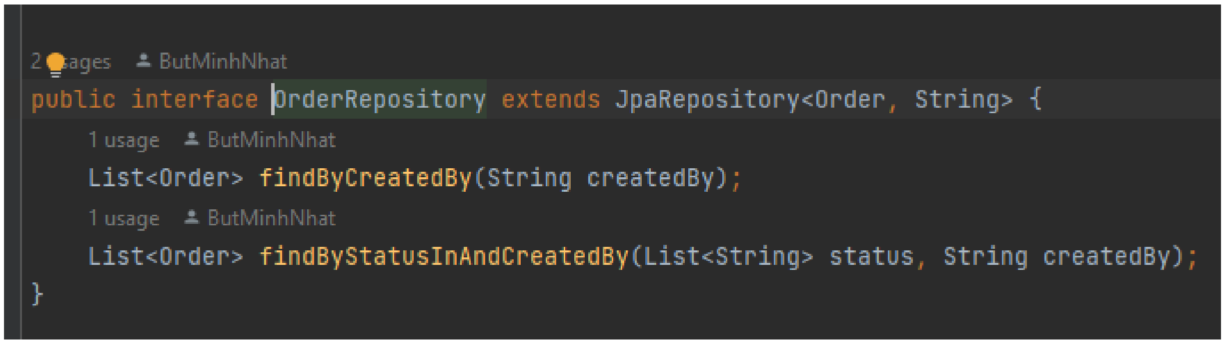
```
public ResponseEntity<List<Order>> getOrders(List<String> status, String userId) {
    if(status != null) {
        return ResponseEntity.ok(orderRepository.findByStatusInAndCreatedBy(status, userId));
    }
    return ResponseEntity.ok(orderRepository.findByCreatedBy(userId));
}

1 usage   Võ Tấn Phát +1
public ResponseEntity<Order> getSingleOrder(String orderId) {
    Optional<Order> orderData = orderRepository.findById(orderId);
    if (orderData.isEmpty()) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    return new ResponseEntity<Order>(orderData.get(), HttpStatus.OK);
}

1 usage   ButMinhNhat +1
public ResponseEntity<Order> createOrder(OrderRequest request, String userId) {
    //check order request
    checkExistProducts(request);

    Order order = Order.builder()
            .status(OrderStatus.PENDING.name())
            .total(getTotalOrderPrice(request.products()))
            .products(request.products())
            .createdBy(userId)
            .updatedBy(userId)
            .build();

    return new ResponseEntity<Order>(orderRepository.save(order), HttpStatus.OK);
}
```

Figure 5.1.4.7 OrderService [1]



```
public ResponseEntity<Order> updateOrder(String orderId, OrderRequest request, String userId) {
    Optional<Order> orderData = orderRepository.findById(orderId);
    if(orderData.isEmpty()) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }

    if(!containStatus(request.status())) {
        return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
    }

    if(request.status().equalsIgnoreCase(OrderStatus.PAID.name())) {
        //check exist products
        checkExistProducts(request);
    }

    //update
    if(request.status() != null) { orderData.get().setStatus(request.status()); }
    if(request.products() != null) {
        orderData.get().setTotal(getTotalOrderPrice(request.products()));
        orderData.get().setProducts(request.products());
    }
    orderData.get().setUpdatedBy(userId);

    return new ResponseEntity<Order>(orderRepository.save(orderData.get()), HttpStatus.OK);
}

1 usage  Võ Tấn Phát +1
public ResponseEntity deleteOrder(String orderId) {
    Optional<Order> orderData = orderRepository.findById(orderId);
    if (orderData.isEmpty()) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    orderRepository.deleteById(orderId);
    return new ResponseEntity<>(HttpStatus.OK);
}
```

Figure 5.1.4.8 OrderService [2]



```
2 usages  ButMinhNhat
public interface OrderRepository extends JpaRepository<Order, String> {
    1 usage  ButMinhNhat
    List<Order> findByCreatedBy(String createdBy);
    1 usage  ButMinhNhat
    List<Order> findByStatusInAndCreatedBy(List<String> status, String createdBy);
}
```

Figure 5.1.4.9 OrderRepository

5.1.4.4 Conclusion

The requests of the OrderService are represented as shown in the figure below:

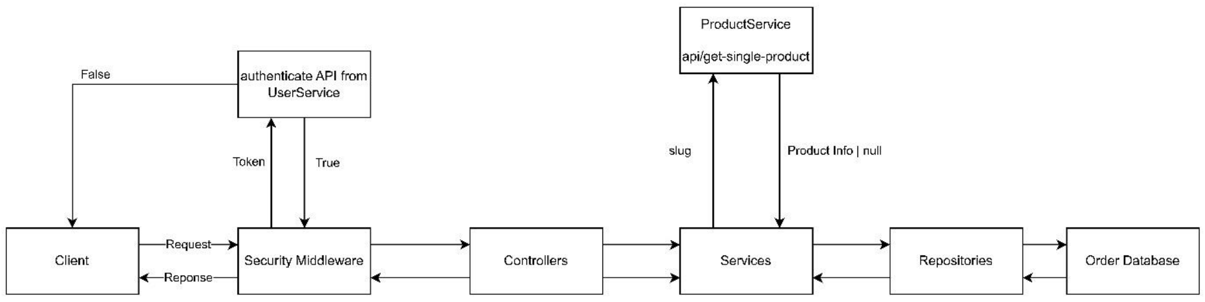
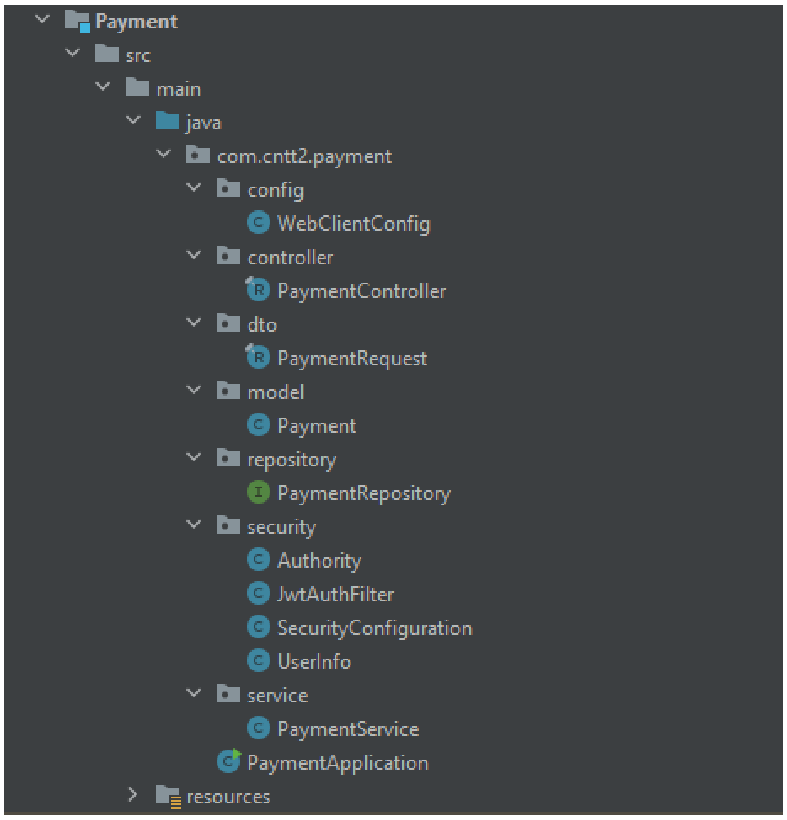


Figure 5.1.4.10 Flow request of OrderService

5.1.5 Payment Service

Payment Service is also organized according to the DDD architecture and has Security Middleware similar to ProductService, UserService, and OrderService:

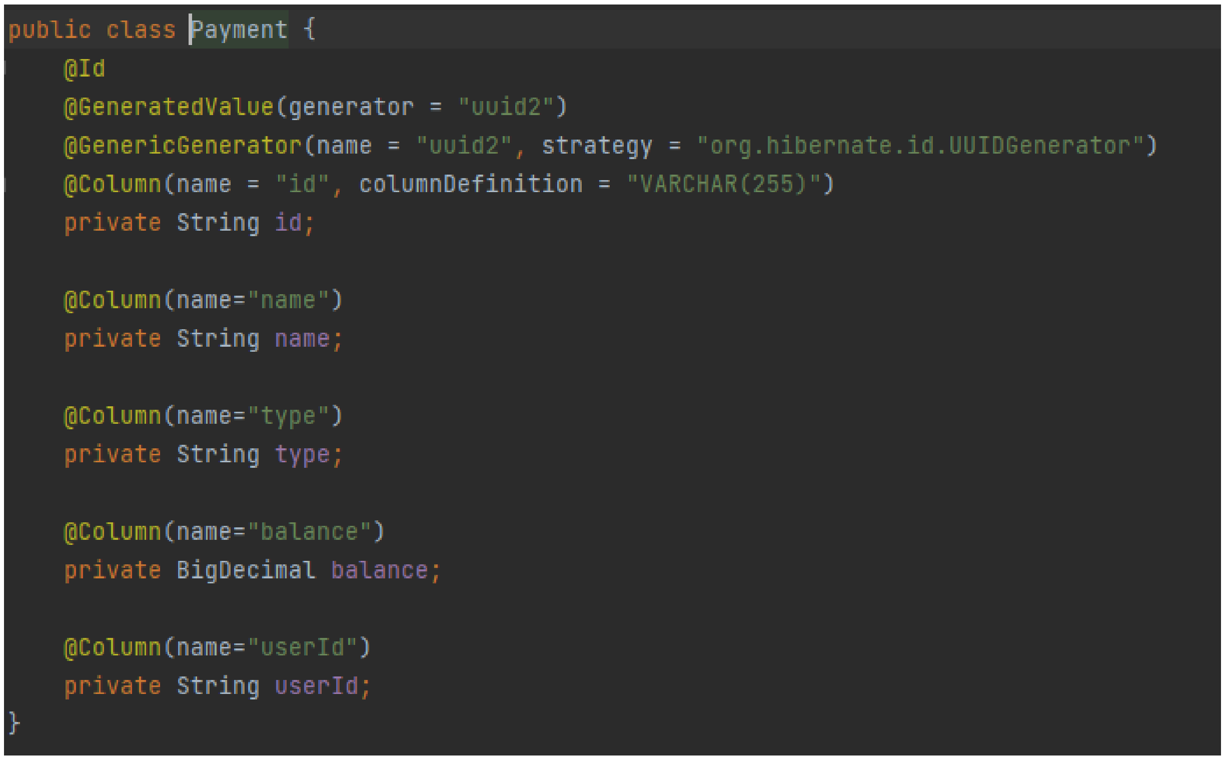


```
Payment
  src
    main
      java
        com.cntt2.payment
          config
            WebClientConfig
          controller
            PaymentController
          dto
            PaymentRequest
          model
            Payment
          repository
            PaymentRepository
          security
            Authority
            JwtAuthFilter
            SecurityConfiguration
            UserInfo
          service
            PaymentService
          PaymentApplication
      resources
```

Figure 5.1.5.1 Payment Service's organization

5.1.5.1 Model

PaymentService has only one model, which is described as follows:



```
public class Payment {
    @Id
    @GeneratedValue(generator = "uuid2")
    @GenericGenerator(name = "uuid2", strategy = "org.hibernate.id.UUIDGenerator")
    @Column(name = "id", columnDefinition = "VARCHAR(255)")
    private String id;

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

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

    @Column(name="balance")
    private BigDecimal balance;

    @Column(name="userId")
    private String userId;
}
```

Figure 5.1.5.2 PaymentModel

5.1.5.2 Security

As with the OrderService, the Payment Service also interacts with the UserService to authenticate tokens, and any requests within this service also require authentication.

5.1.5.3 Controller (Routes)

APIs:

- GET: GetPayments
- GET: GetPayment
- POST: CreatePayment
- PUT: UpdatePayment

- DELETE: DeletePayment

These APIs will receive requests and pass them to the Controller after going through the token authentication step. The Controller will continue to route the request to the Service layer to process the logic, and the service layer will interact with the database through PaymentRepository similar to other services. In this case, the Payment Service does not interact with other services to process logic. Refer to the source code for more details.

5.1.4.3. Conclusion

Requests to the PaymentService will be represented as follows:

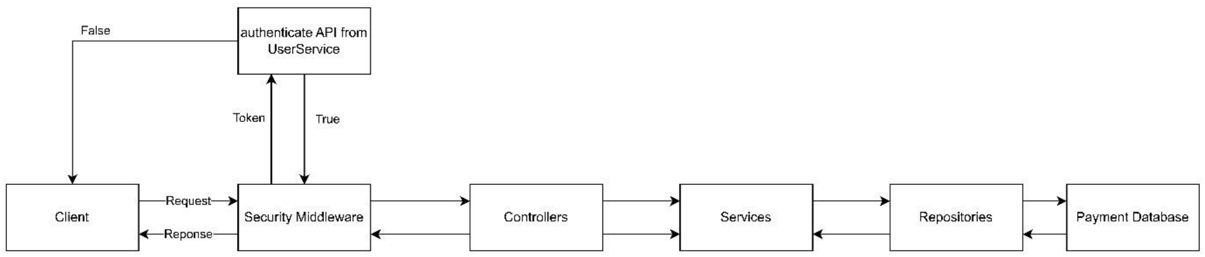
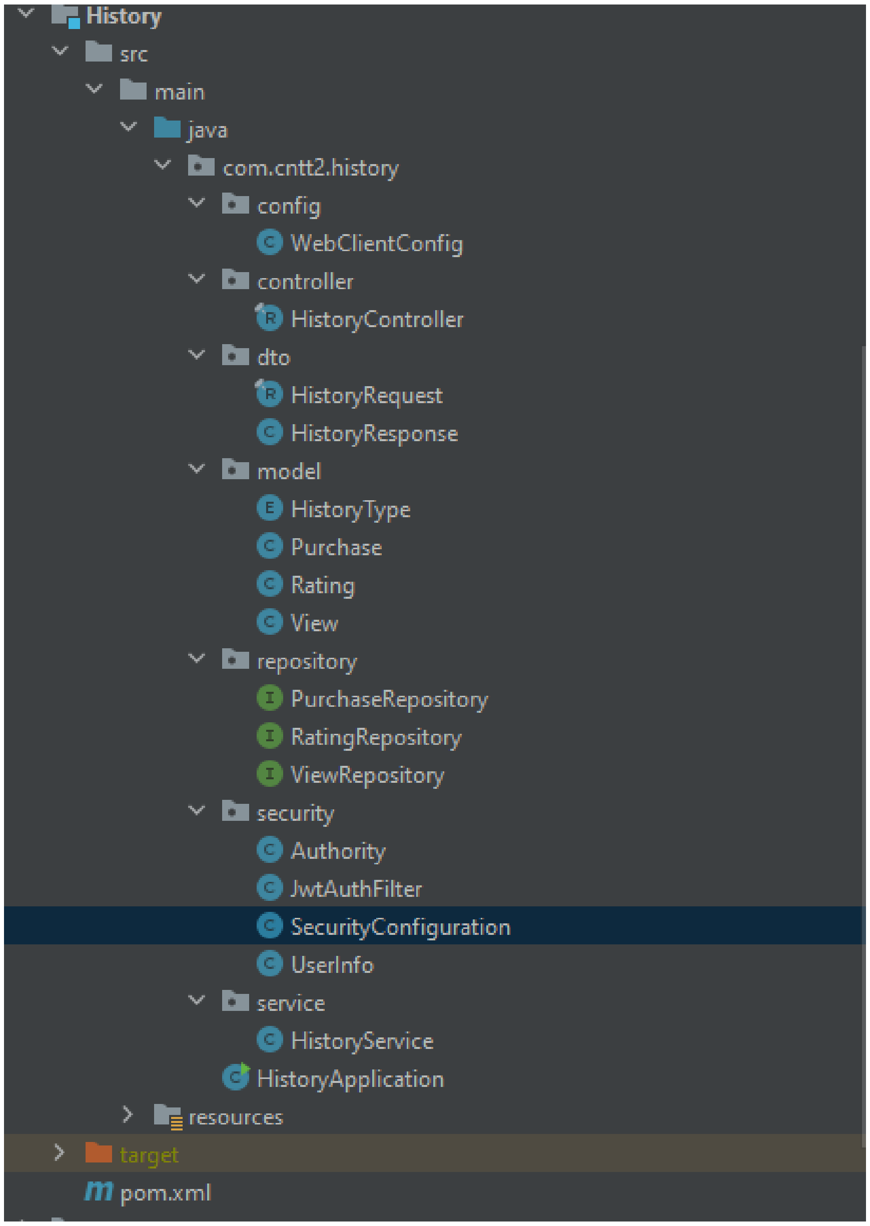


Figure 5.1.5.3 Flow request of Payment Service

5.1.6 History Service

The History Service is also organized according to the DDD structure, and it has security check middleware similar to other services.



```
History
  src
    main
      java
        com.cntt2.history
          config
            WebClientConfig
          controller
            HistoryController
          dto
            HistoryRequest
            HistoryResponse
          model
            HistoryType
            Purchase
            Rating
            View
          repository
            PurchaseRepository
            RatingRepository
            ViewRepository
          security
            Authority
            JwtAuthFilter
            SecurityConfiguration
            UserInfo
          service
            HistoryService
          HistoryApplication
      resources
  target
  pom.xml
```

Figure 5.1.6.1 History Service Organization

5.1.6.1 Model

The History Service has 3 models:

- Purchase_product: purchased products
- Rating_product: ratings of products
- View_product: number of views for each product

5.1.6.2 Security

It is recommended to have authentication for all APIs in the History Service, and to use the UserService for authentication.

5.1.6.3 Controller (Routes)

APIs:

- GET: GetHistories
- POST: PostHistory

The History Service follows the flow of other services, and it only interacts with its own database through the Repository, without interacting with other services. For more details, please refer to the source code.

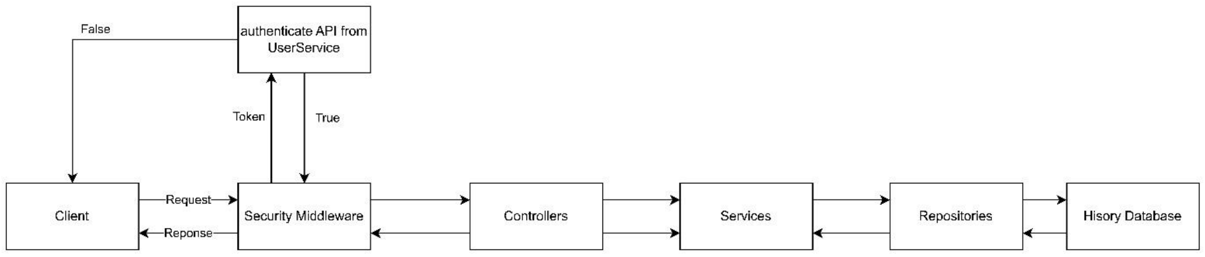


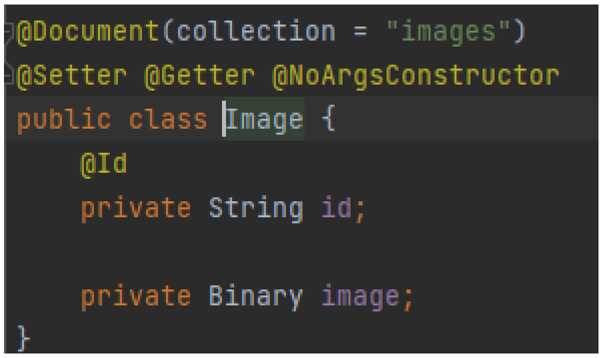
Figure 5.1.6.2 History Service's Flow Request

5.1.7 Image Service

The Image Service is organized like the other services, but it does not have any Security Middleware.

5.1.7.1 Model

The model for this service only includes the Image Model.



```java
@Document(collection = "images")
@Setter @Getter @NoArgsConstructor
public class Image {
    @Id
    private String id;

    private Binary image;
}
```

Figure 5.1.7.1 Image Model

5.1.7.2 Security

Because the Image Service does not have any Security Middleware, there is no need for any authentication.

5.1.7.3 Controller (Routes)

APIs:

- GET: GetPhoto
- POST: AddPhoto

The Image Service is not different from other services in the system. It also receives requests in the controller and passes them down to the service layer to handle logic and interact with the database using the repository. For more details, please refer to the source code.

5.1.7.4 Conclusion

The Image Service can be represented as follows:

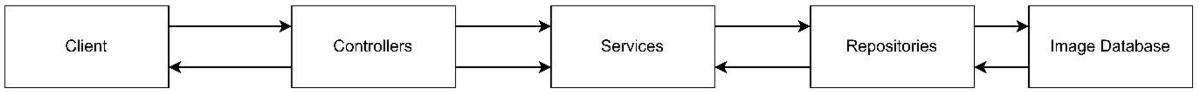


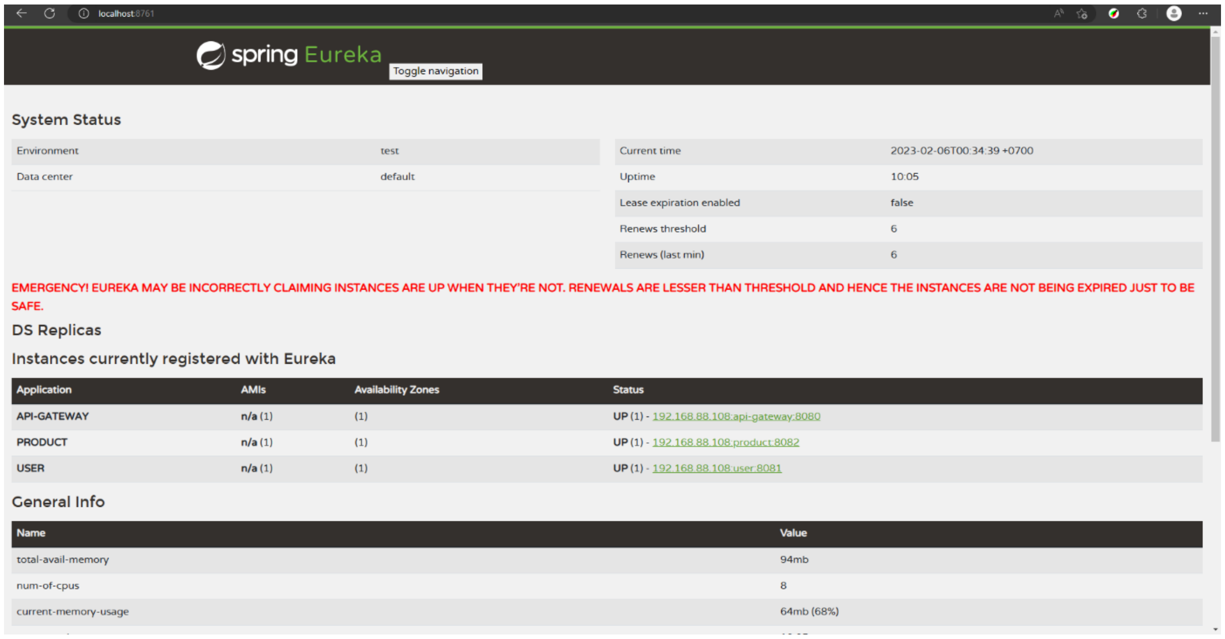
Figure 5.1.7.2 Flow of request in Image Service

5.1.8 API Gateway

API Gateway is used to declare ports for services. For more details, please refer to the source code.

5.1.9 Eureka Server

Eureka Server is used to manage services. For more details, please refer to the source code.


localhost:8761

spring Eureka

Toggle navigation

System Status

Environment	test
Data center	default

Current time	2023-02-06T00:34:39 +0700
Uptime	10:05
Lease expiration enabled	false
Renews threshold	6
Renews (last min)	6

EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
API-GATEWAY	n/a (1)	(1)	UP (1) - 192.168.88.108:api-gateway:8080
PRODUCT	n/a (1)	(1)	UP (1) - 192.168.88.108:product:8082
USER	n/a (1)	(1)	UP (1) - 192.168.88.108:user:8081

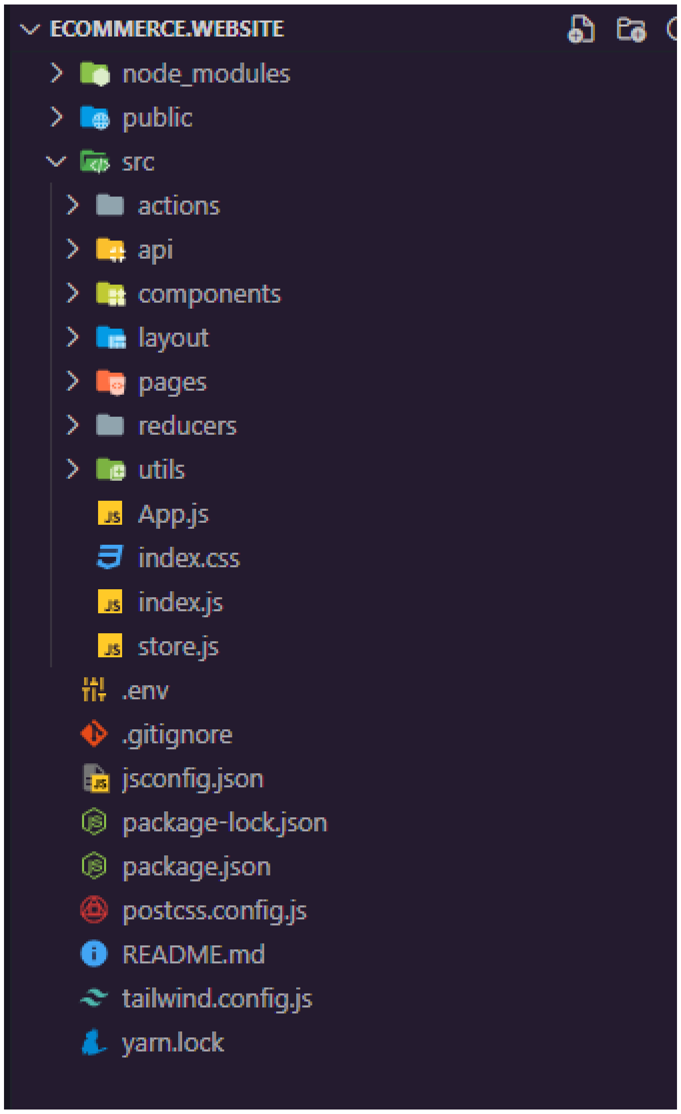
General Info

Name	Value
total-avail-memory	94mb
num-of-cpus	8
current-memory-usage	64mb (68%)

Figure 5.1.9.1 Eureka Server

5.2 Implementation of E-commerce Website

The e-commerce website is implemented using the ReactJS JavaScript library. The structure of a React project is organized as follows:



```
ECOMMERCE.WEBSITE
  node_modules
  public
  src
    actions
    api
    components
    layout
    pages
    reducers
    utils
    App.js
    index.css
    index.js
    store.js
  .env
  .gitignore
  jsconfig.json
  package-lock.json
  package.json
  postcss.config.js
  README.md
  tailwind.config.js
  yarn.lock
```

Figure 5.2.1 Project organization in ReactJS

Pages in the Website The website includes the following pages:

- Home: "/"
- Product: "/product"
- ProductDetail: "/product/slug"
- Category: "/category/slug"
- Cart: "/cart"
- Checkout: "/checkout/orderId"
- SignIn: "/auth/sign-in"
- SignUp: "/auth/sign-up"
- NotFound: "/**"

For details on interface implementation, please refer to the source code. Here are some interfaces of the e-commerce website:

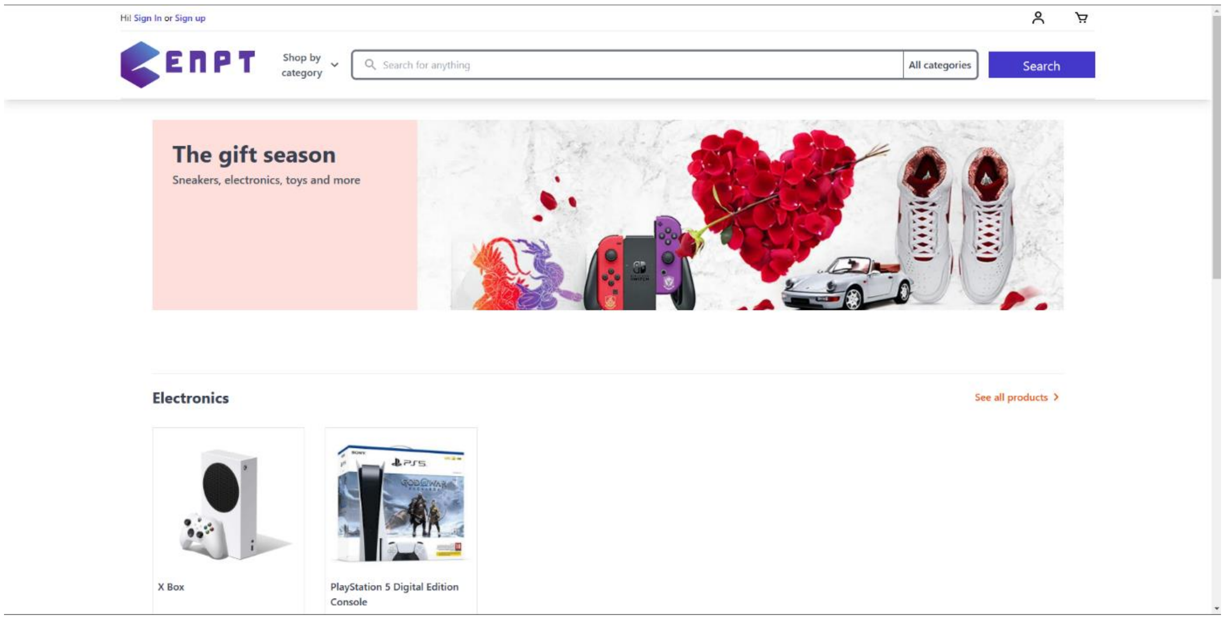
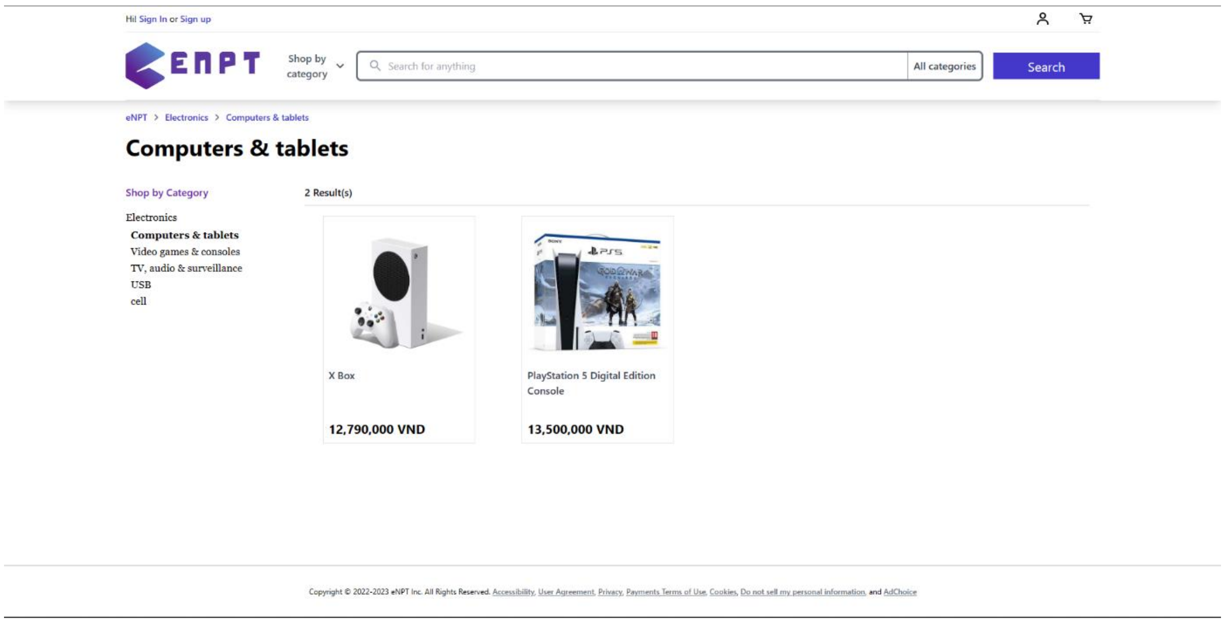


Figure 5.2.2 Home Page



eNPT > Electronics > Computers & tablets

Computers & tablets

Shop by Category

2 Result(s)

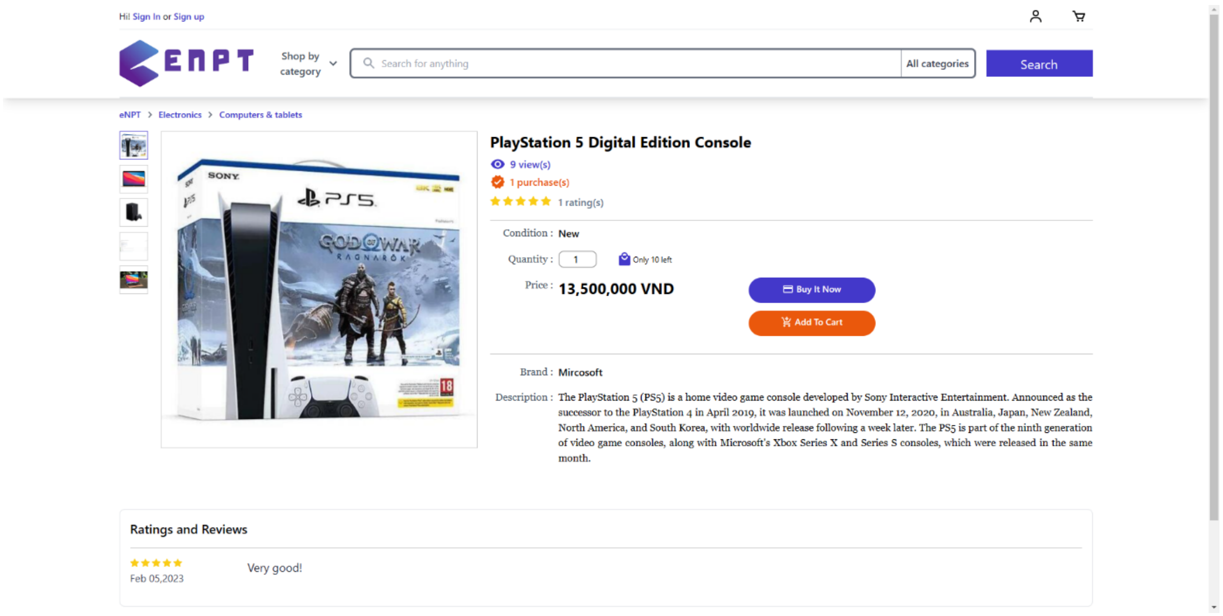
Electronics
Computers & tablets
Video games & consoles
TV, audio & surveillance
USB
cell

X Box
12,790,000 VND

PlayStation 5 Digital Edition Console
13,500,000 VND

Copyright © 2022-2023 eNPT Inc. All Rights Reserved. Accessibility, User Agreement, Privacy, Payments Terms of Use, Cookies, Do not sell my personal information, and AdChoice

Figure 5.2.3 Search filter



eNPT > Electronics > Computers & tablets

PlayStation 5 Digital Edition Console

9 view(s)

1 purchase(s)

1 rating(s)

Condition : New

Quantity : 1 Only 10 left

Price : 13,500,000 VND

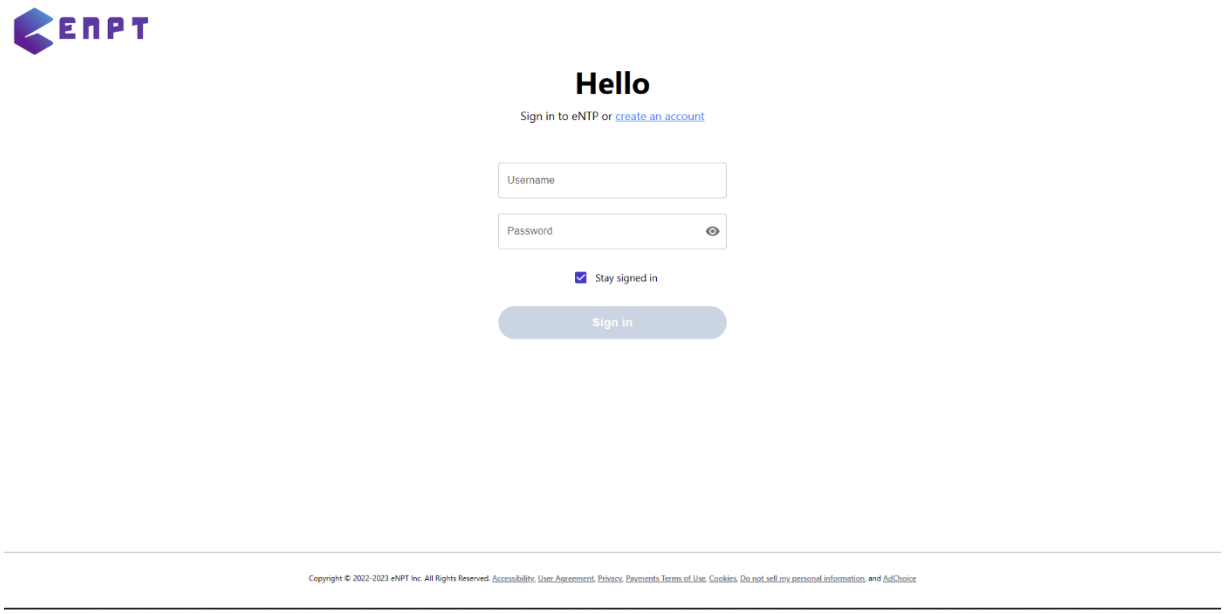
Brand : Mircosoft

Description : The PlayStation 5 (PS5) is a home video game console developed by Sony Interactive Entertainment. Announced as the successor to the PlayStation 4 in April 2019, it was launched on November 12, 2020, in Australia, Japan, New Zealand, North America, and South Korea, with worldwide release following a week later. The PS5 is part of the ninth generation of video game consoles, along with Microsoft's Xbox Series X and Series S consoles, which were released in the same month.

Ratings and Reviews

★★★★★ Feb 05,2023 Very good!

Figure 5.2.4 Product detail



Hello

Sign in to eNTP or create an account

Username

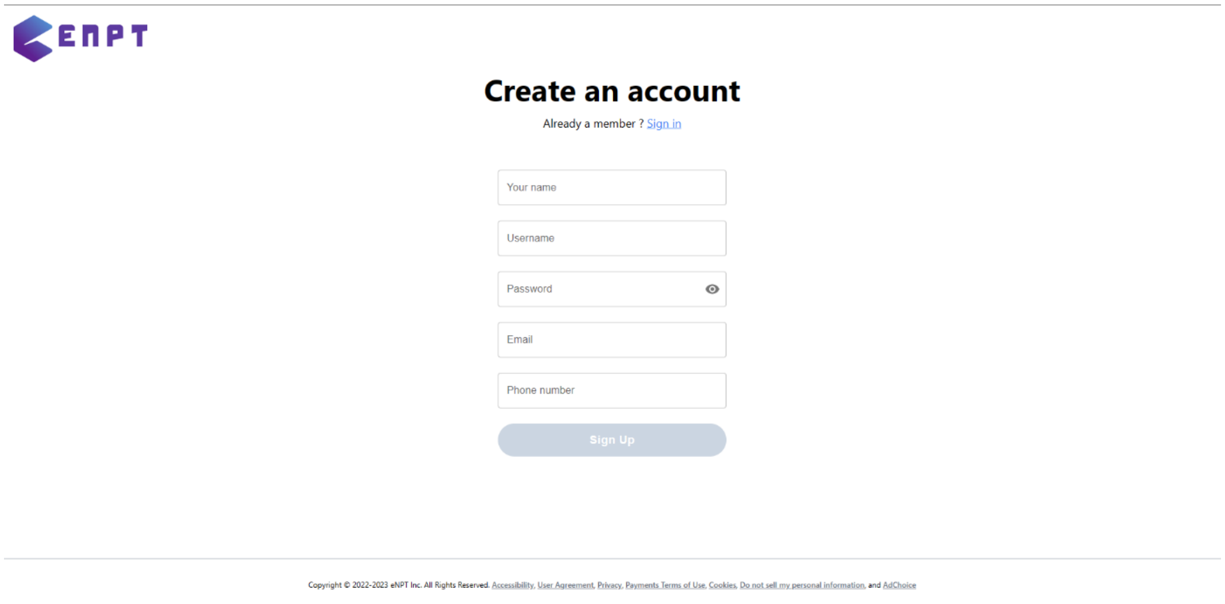
Password

Stay signed in

Sign in

Copyright © 2022-2023 eNPT Inc. All Rights Reserved. Accessibility, User Agreement, Privacy, Payments Terms of Use, Cookies, Do not sell my personal information, and AdChoice

Figure 5.2.5 Sign In Page



Create an account

Already a member ? Sign in

Your name

Username

Password

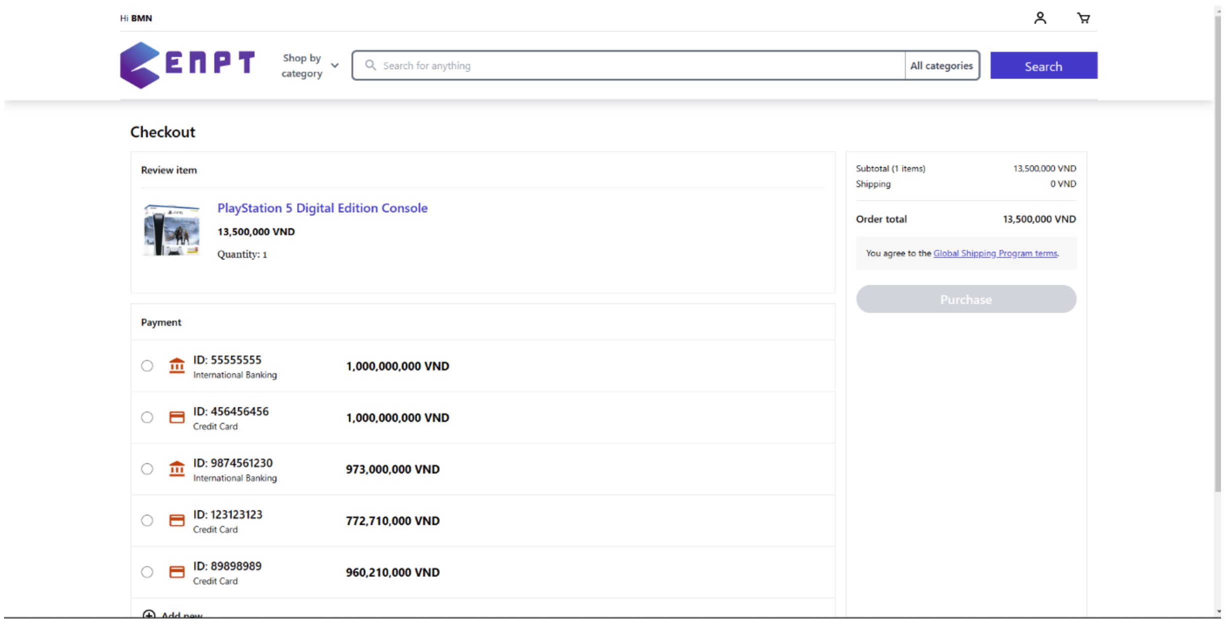
Email

Phone number

Sign Up

Copyright © 2022-2023 eNPT Inc. All Rights Reserved. Accessibility, User Agreement, Privacy, Payments Terms of Use, Cookies, Do not sell my personal information, and AdChoice

Figure 5.2.6 SignUp Page



Checkout

Review item

PlayStation 5 Digital Edition Console
13,500,000 VND
Quantity: 1

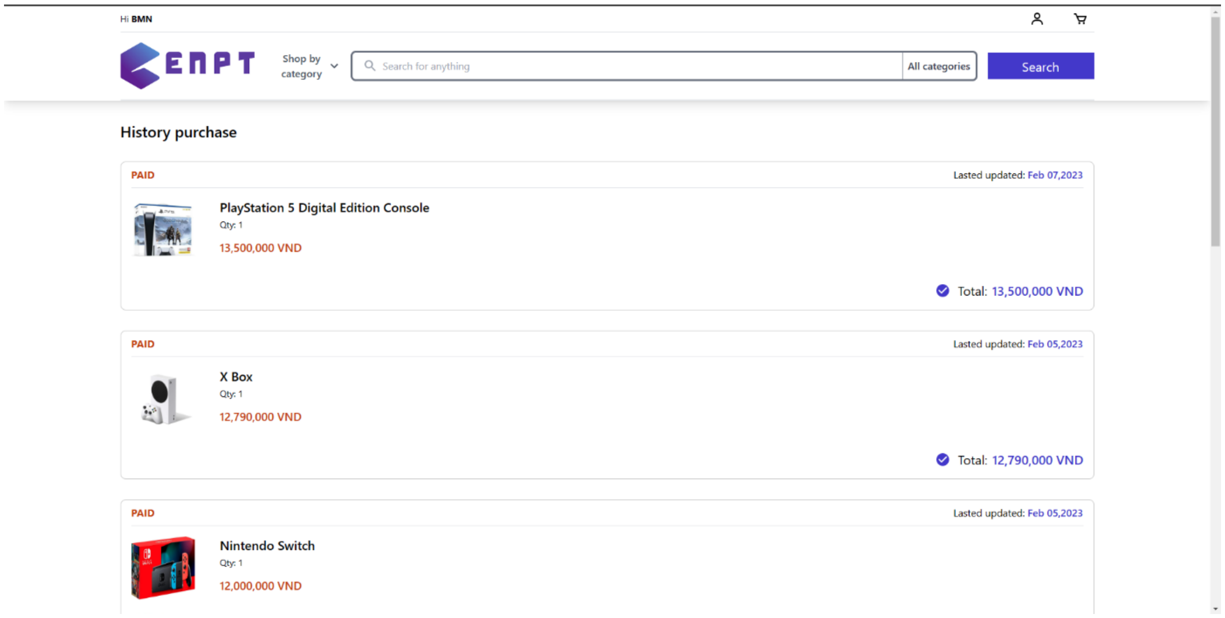
Subtotal (1 items)	13,500,000 VND
Shipping	0 VND
Order total	13,500,000 VND

You agree to the Global Shipping Program terms.

Payment

ID: 55555555 International Banking	1,000,000,000 VND
ID: 456456456 Credit Card	1,000,000,000 VND
ID: 9874561230 International Banking	973,000,000 VND
ID: 123123123 Credit Card	772,710,000 VND
ID: 89898989 Credit Card	960,210,000 VND

Figure 5.2.7 Checkout Page



History purchase

PAID — Lasted updated: Feb 07,2023
PlayStation 5 Digital Edition Console
Qty: 1
13,500,000 VND
Total: 13,500,000 VND

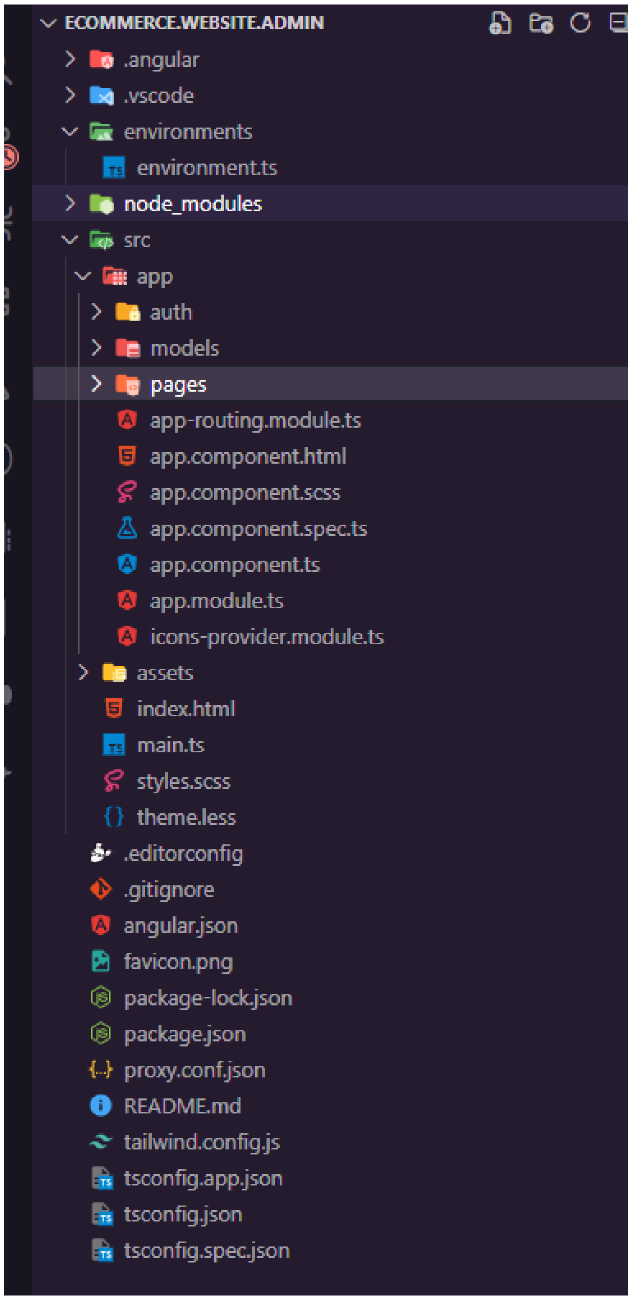
PAID — Lasted updated: Feb 05,2023
X Box
Qty: 1
12,790,000 VND
Total: 12,790,000 VND

PAID — Lasted updated: Feb 05,2023
Nintendo Switch
Qty: 1
12,000,000 VND

Figure 5.2.8 History Page

5.3 Website Management System Implementation

The website management system is implemented using the Angular framework, which uses typescript as its programming language. The structure of an Angular project is organized as follows:



```
ECOMMERCE.WEBSITE.ADMIN
  > .angular
  > .vscode
  v environments
      environment.ts
  > node_modules
  v src
    v app
      > auth
      > models
      > pages
        app-routing.module.ts
        app.component.html
        app.component.scss
        app.component.spec.ts
        app.component.ts
        app.module.ts
        icons-provider.module.ts
    > assets
      index.html
      main.ts
      styles.scss
      theme.less
    .editorconfig
    .gitignore
    angular.json
    favicon.png
    package-lock.json
    package.json
    proxy.conf.json
    README.md
    tailwind.config.js
    tsconfig.app.json
    tsconfig.json
    tsconfig.spec.json
```

Figure 5.3.1 Organize in an Angular project

The website consists of the following pages:

- Home: "dashboard/welcome"
- Product: "/product-manage/product"
- ProductDetail: "/product-manage/product-detail/slug"
- CreateProduct: "/product-manage/create-product"
- Category: "/product-manage/category"
- CategoryDetail: "/product-manage/category-detail/categoryId"
- CreateCategory: "/product-manage/create-category"
- Brand: "/product-manage/brand"
- BrandDetail: "/product-manage/brand-detail/brandId"
- CreateBrand: "/product-manage/create-brand"
- Order: "/order-manage/order"
- OrderDetail: "/order-manage/order-detail/orderId"
- User: "/user-manage/user"
- CreateUser: "/user-manage/create-user"
- SignIn: "/auth"
- NotFound: "/**"

For details on the implementation of the interfaces, please refer to the source code. Here are some interfaces of the management website: [No text was provided for this part, as it is describing visual elements.]

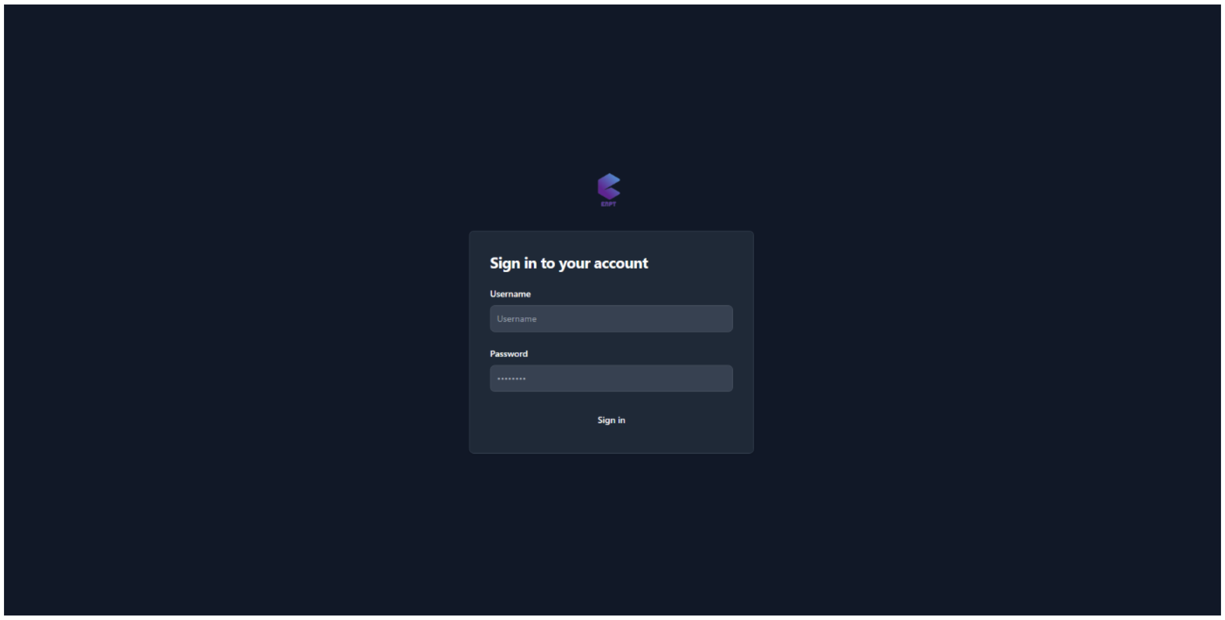
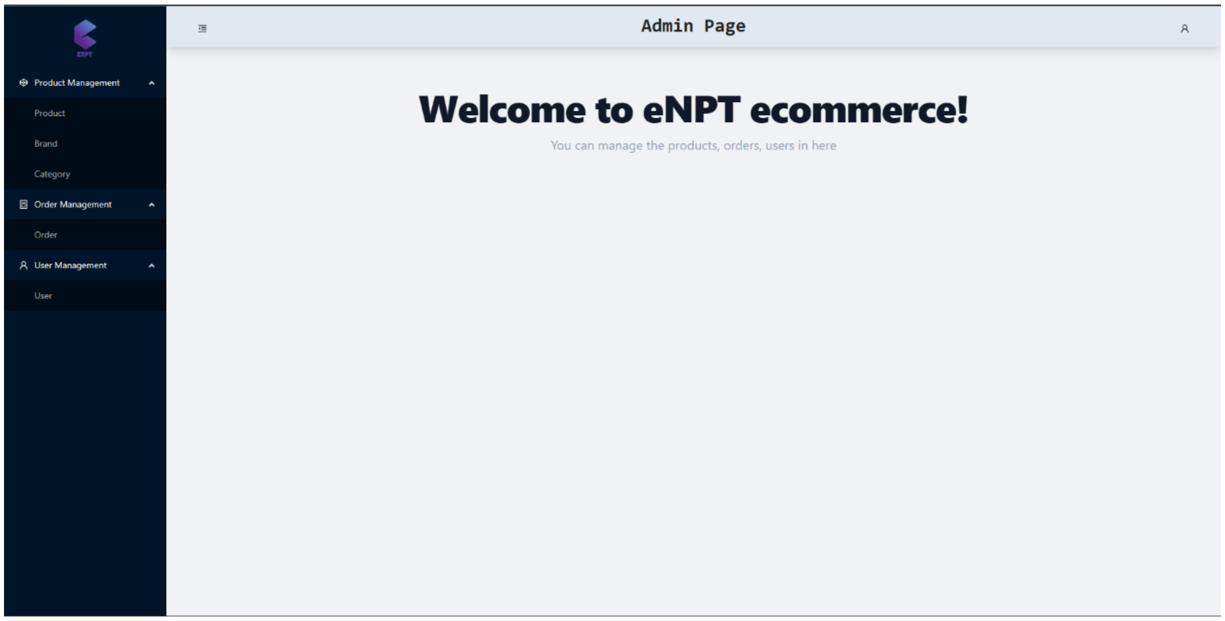


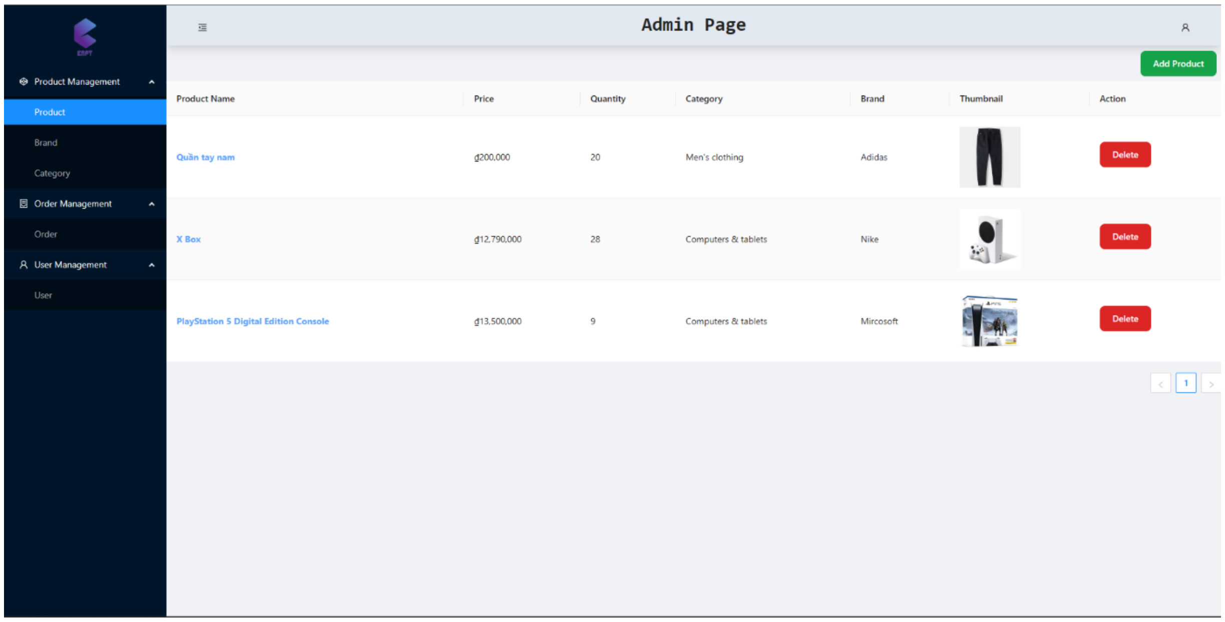
Figure 5.3.2 SignIn Page



Welcome to eNPT ecommerce!

You can manage the products, orders, users in here

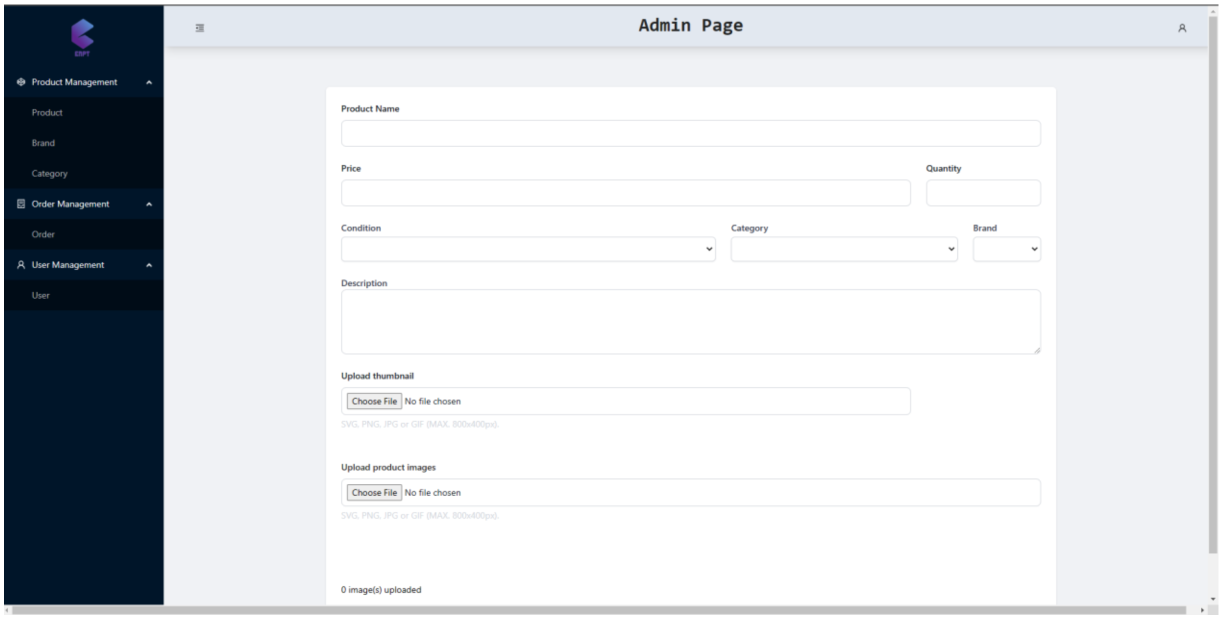
Figure 5.3.3 Homepage



Admin Page

Product Name	Price	Quantity	Category	Brand	Thumbnail	Action
Quần tay nam	₫200,000	20	Men's clothing	Adidas		Delete
X Box	₫12,790,000	28	Computers & tablets	Nike		Delete
PlayStation 5 Digital Edition Console	₫13,500,000	9	Computers & tablets	Mircosoft		Delete

Figure 5.3.4 Product Page



Admin Page

Product Name

Price

Quantity

Condition

Category

Brand

Description

Upload thumbnail

No file chosen

SVG, PNG, JPG or GIF (MAX. 800x400px).

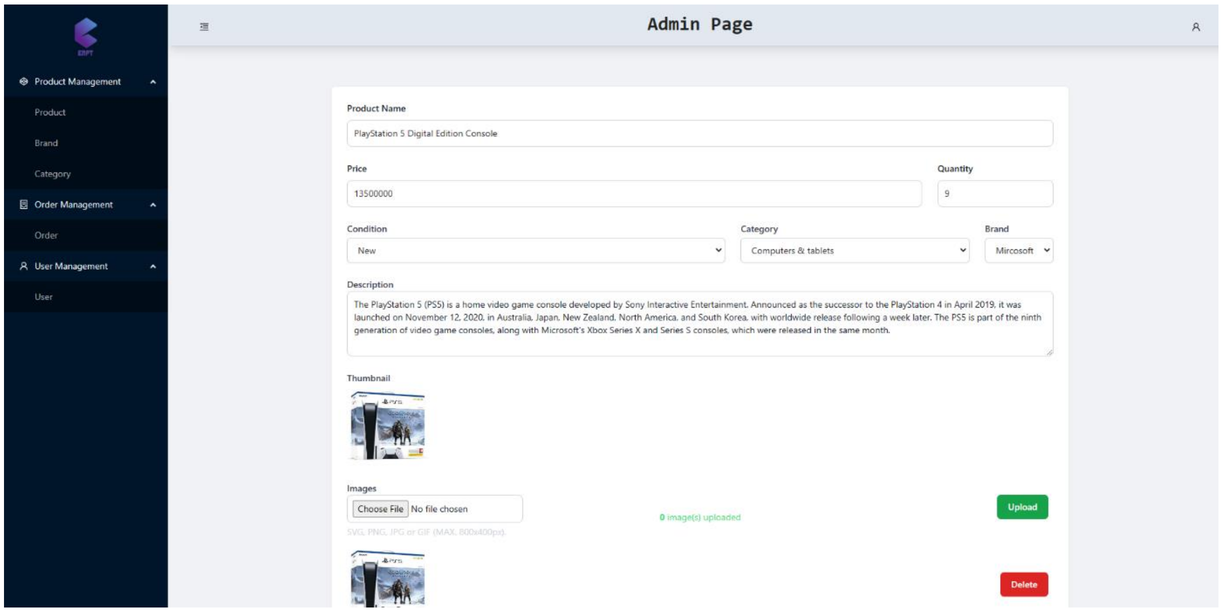
Upload product images

No file chosen

SVG, PNG, JPG or GIF (MAX. 800x400px).

0 image(s) uploaded

Figure 5.3.5 Create Product Page



Product Name: PlayStation 5 Digital Edition Console

Price: 13500000

Quantity: 9

Condition: New

Category: Computers & tablets

Brand: Mircosoft

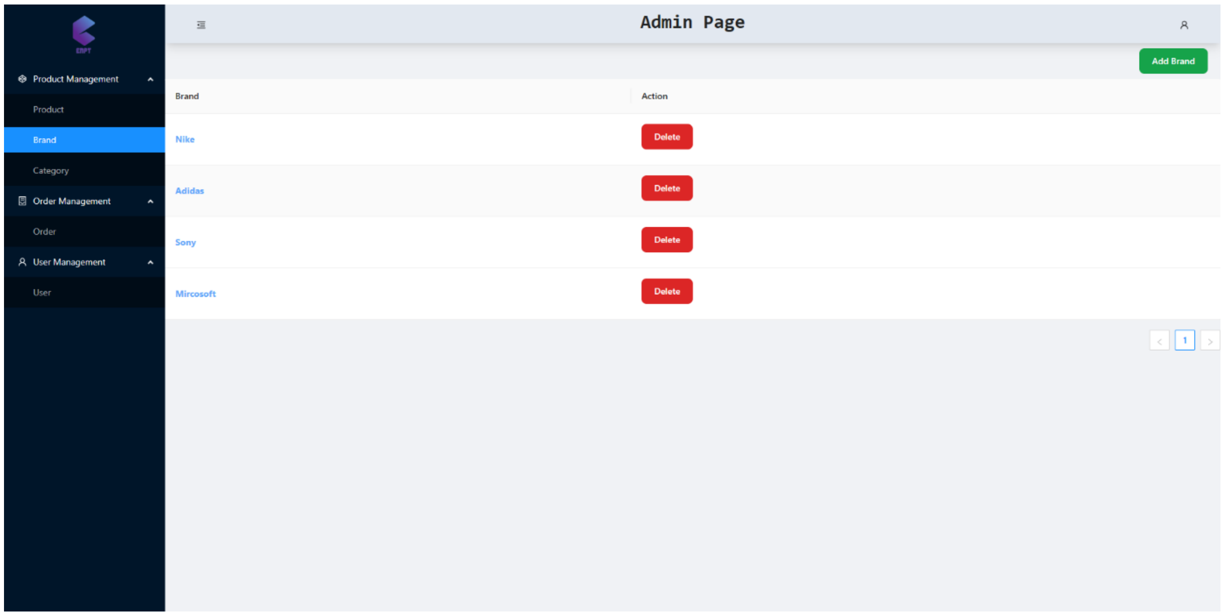
Description: The PlayStation 5 (PS5) is a home video game console developed by Sony Interactive Entertainment. Announced as the successor to the PlayStation 4 in April 2019, it was launched on November 12, 2020, in Australia, Japan, New Zealand, North America, and South Korea, with worldwide release following a week later. The PS5 is part of the ninth generation of video game consoles, along with Microsoft's Xbox Series X and Series S consoles, which were released in the same month.

Images: No file chosen

SVG, PNG, JPG or GIF (MAX. 800x400px).

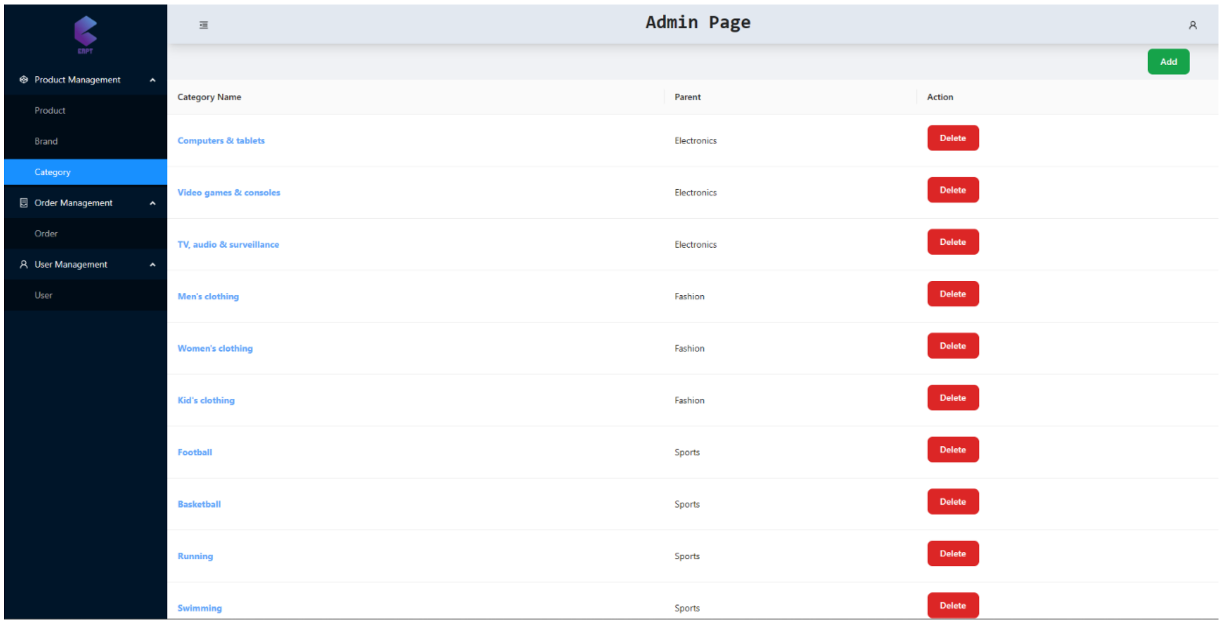
0 image(s) uploaded

Figure 5.3.6 Product Detail



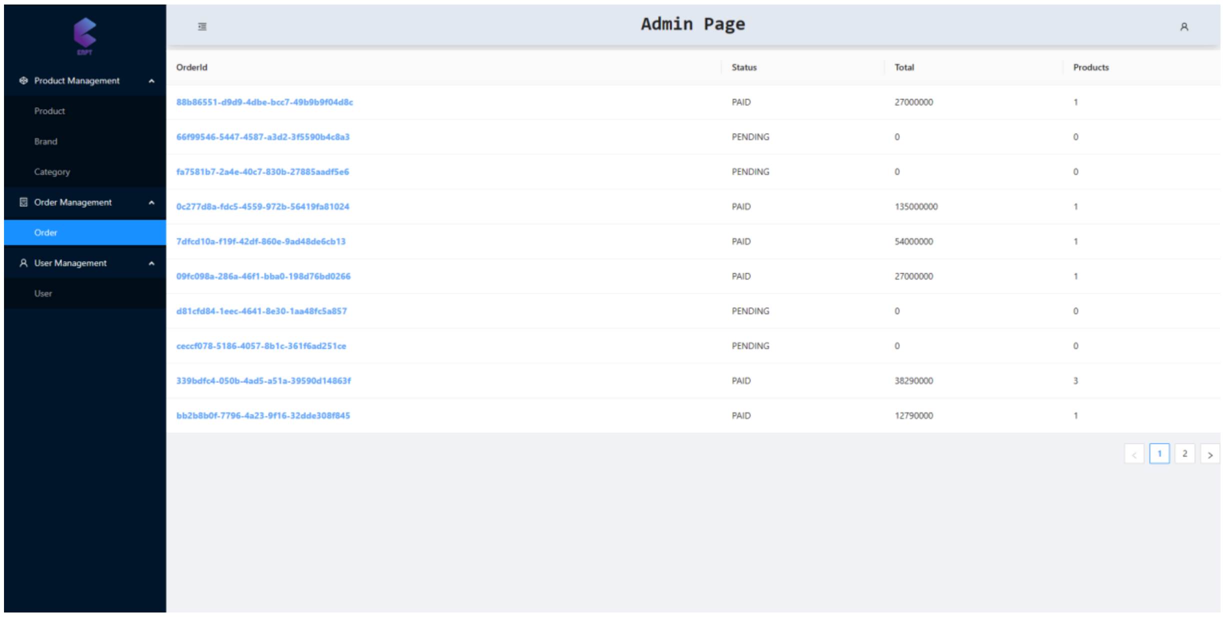
Brand	Action
Nike	Delete
Adidas	Delete
Sony	Delete
Mircosoft	Delete

Figure 5.3.7 Brand Page



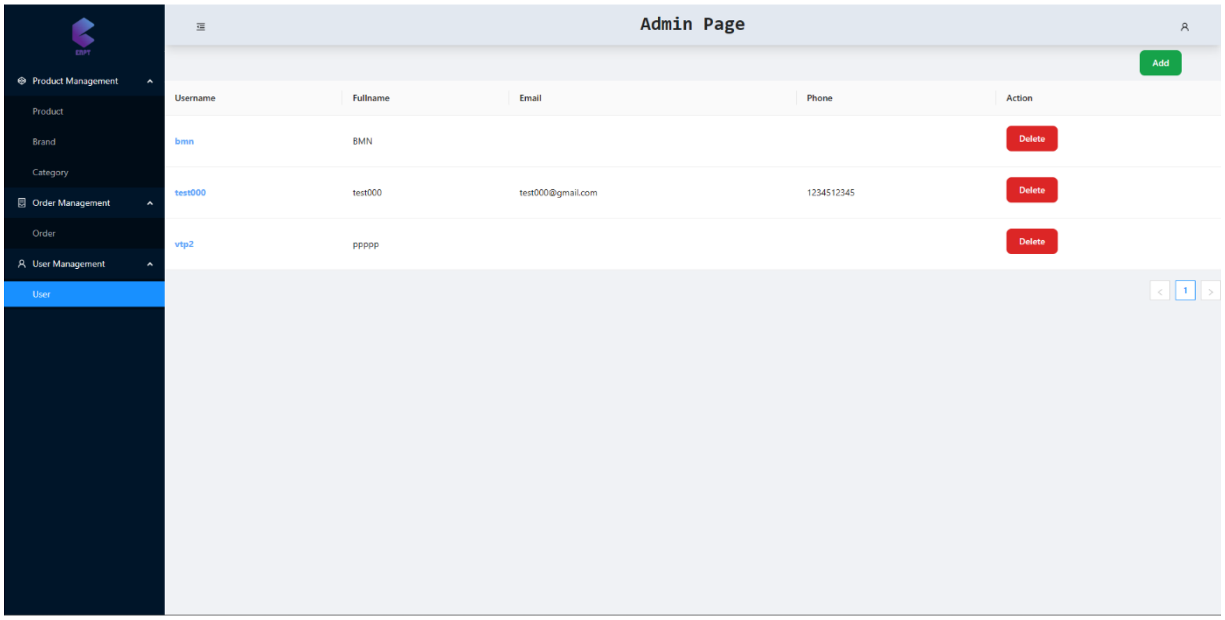
Category Name	Parent	Action
Computers & tablets	Electronics	Delete
Video games & consoles	Electronics	Delete
TV, audio & surveillance	Electronics	Delete
Men's clothing	Fashion	Delete
Women's clothing	Fashion	Delete
Kid's clothing	Fashion	Delete
Football	Sports	Delete
Basketball	Sports	Delete
Running	Sports	Delete
Swimming	Sports	Delete

Figure 5.3.8 Category Page



OrderId	Status	Total	Products
88b86551-d9d9-4dbe-bcc7-49b9b9f04d8c	PAID	27000000	1
66f99546-5447-4587-a3d2-3f5590b4c8a3	PENDING	0	0
fa7581b7-2a4e-40c7-830b-27885aadf5e6	PENDING	0	0
0c277d8a-fdc5-4559-972b-56419fa81024	PAID	135000000	1
7dfcd10a-f19f-42df-860e-9ad48de6cb13	PAID	54000000	1
09fc098a-286a-46f1-bba0-198d76bd0266	PAID	27000000	1
d81cfd84-1eec-4641-8e30-1aa48fc5a857	PENDING	0	0
ceccf078-5186-4057-8b1c-361f6ad251ce	PENDING	0	0
339bdfc4-050b-4ad5-a51a-39590d14863f	PAID	38290000	3
bb2b8b0f-7796-4a23-9f16-32dde308f845	PAID	12790000	1

Figure 5.3.9 Order Page



Username	Fullname	Email	Phone	Action
bmn	BMN			Delete
test000	test000	test000@gmail.com	1234512345	Delete
vtp2	ppppp			Delete

Figure 5.3.10 User Page

CHAPTER VI - CONCLUSION

After completing the project, our team has successfully implemented an e-commerce system using a microservices architecture. The system has achieved basic functions for selling products, and consists of two separate websites, a sales website and a management website. This is the advantage of microservices architecture as it separates services for easy maintenance and upgrade, and when one service encounters a problem, other services can still operate.

In addition to the achievements that our team has made, the system still lacks some functions that have not been completed. Our team has learned and gained experience for future projects.

REFERENCES

1. Dat Nguyen (2020). “Tìm hiểu về microservice”, Viblo Blog, https://viblo.asia/p/tim-hieu-ve-microservice-Do754PD45M6
2. Kong. “What are microservices?”, Mircoservice s IO, https://microservices.io/
3. “Domain-Driven-Design(DDD)”,GeeksForGeeks, https://www.geeksforgeeks.org/domain-driven-design-ddd/
4. Tu Bean (2018), “[Microservice] Dựng Microservice web bằng Spring Boot và Eureka”, Github, https://tubean.github.io/2018/12/microservice-springboot-eureka
5. Omar Elgabry, “Microservices with Spring Boot”, OmarElgabry's Blog, https://medium.com/omarelgabrys-blog/microservices-with-spring-boot-intro-to-microservices-part-1-c0d24cd422c3
6. Binildas Christudas (2019), Practical Mircoservices Architectureal Patterns