TRƯỜNG ĐẠI HỌC FPT

CAPSTONE PROJECT REPORT

Report 4 – Software Design Document

– Da Nang, April 2023 –

Table of Contents

I. Software Design Description

1. System Design

1.1 System Architecture

Description: The overall picture of system architecture

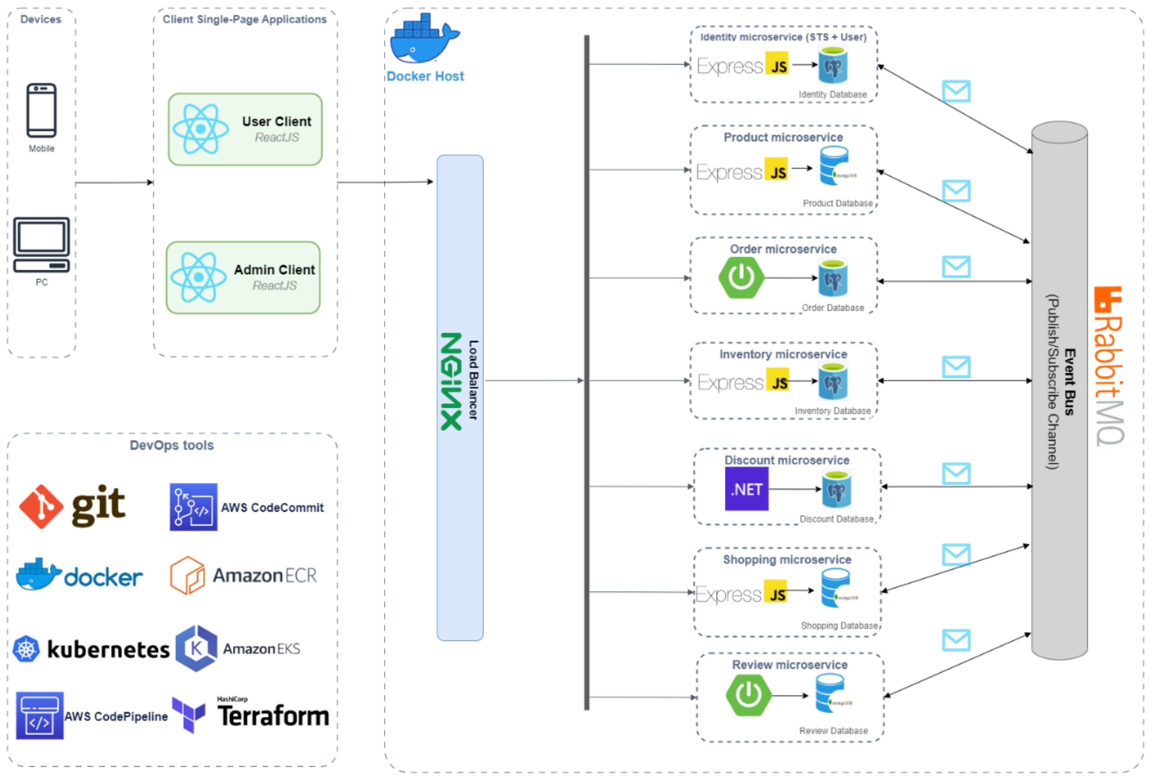


Figure 82: System architecture overview

Details Information: To meet the hypothetical situation of this project, we have proposed to apply the microservice architecture. In the top-down view, the system includes two individual websites for admin and general users overlay the microservice backend.

No	Components	Description
01	Devices	The devices are supported for the end-user to access the system.
02	User Client	The web app for general users
03	Admin Client	The web app for system admin

04	Docker host	Virtual environment wrapping backend microservices with other auxiliary services and let them run on the physical servers
05	Load blancer	Reversed proxy forwards request from clients to corresponding microservices depend on request prefix path
06	Identity microservice	Microservice for authentication, authorization and handles user profile service
07	Product microservice	Microservice handles facilities operations of product
08	Order microservice	Microservice handles bussiness logic of order product flow and get orders history operation
09	Inventory microservice	Microservies handles prices, product variants and quantity of each variant
10	Discount microservice	Microservice manange discount of products
11	Shopping microservice	Microservices handles facilities operation of genertal user's cart and wishlist
12	Review microservice	Microservice manage reivews of users for products
13	RabbitMQ	The event bus transmit event across microservices
14	Devops tools	Tools using in application CI/CD and deployment

1.1.1 Clean architect

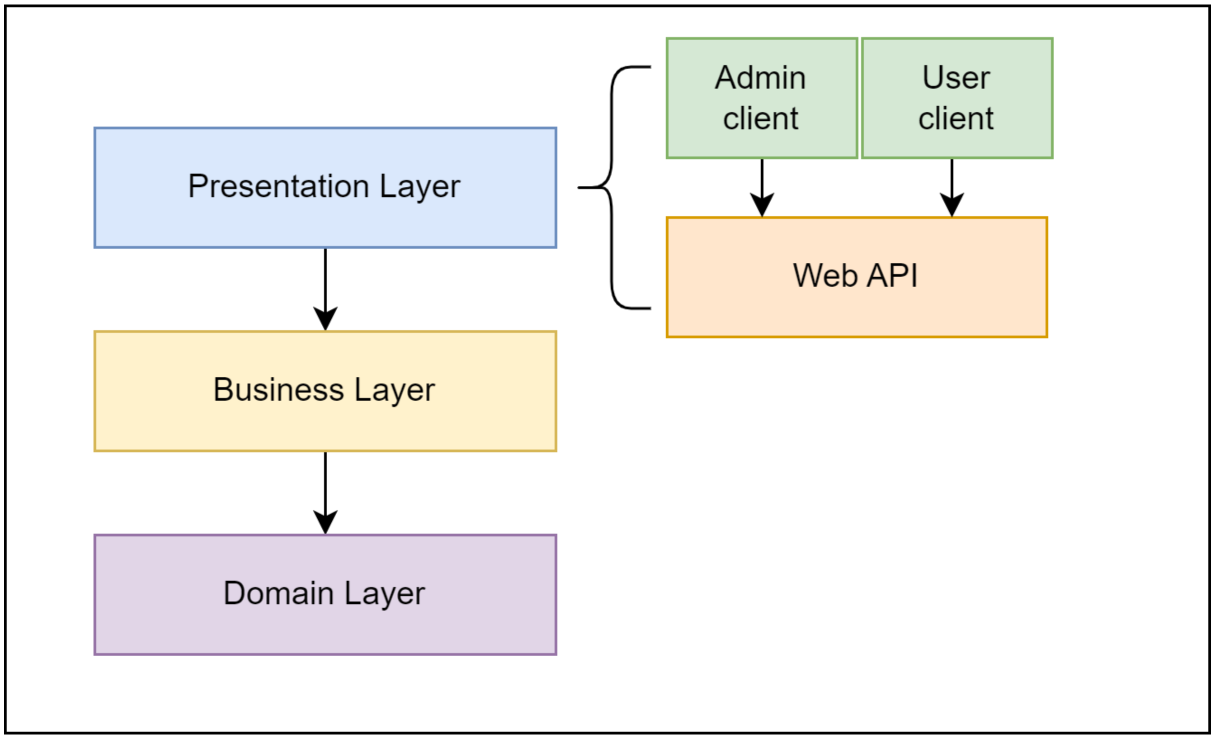


Figure 83: Clean architecture

Description: The Clean Architecture applied in each microservice.

Details Information: Based on the assumption that Potoro is a start-up project, which business and technical requirements change frequently. So, the system must be designed to be quickly adaptable, scalable, support testing, and easy to maintain. The Clean Architecture is an approach in system design, which satisfies the following requirements: Framework Independent, Database Independent, UI Independent, Highly Testable. All the code implementations are based on the business logic (Domain layer and Application layer). These points are completely consistent with the requirements of the project.

1.1.2 Event-driven architecture

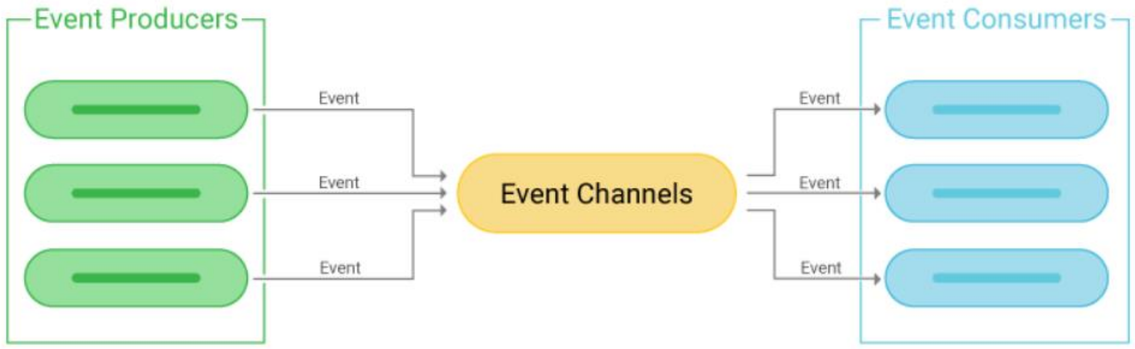


Figure 84: event-driven architecture

Details Information: An event-driven architecture uses events to trigger and communicate between decoupled services and is common in modern applications built with microservices. An event is a change in state, or an update, like an item being placed in a shopping cart on an e-commerce website. Events can either carry the state (the item purchased, its price, and a delivery address) or events can be identifiers (a notification that an order was shipped).

Event-driven architectures have three key components: event producers, event routers, and event consumers. A producer publishes an event to the router, which filters and pushes the events to consumers. Producer services and consumer services are decoupled, which allows them to be scaled, updated, and deployed independently.

1.1.3 Message queue pattern

1.1.3.1 Publish/Subscribe pattern

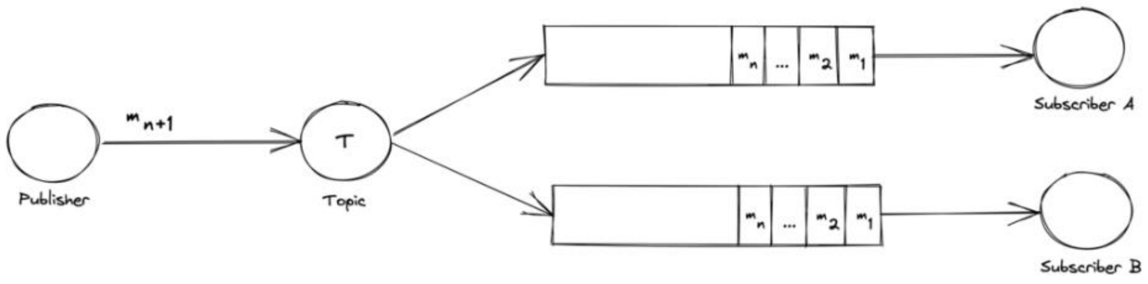


Figure 85: Message queue publish/subscribe pattern

Description: Pub/sub pattern in messae queue

Detail description: Pub/Sub is a message pattern in a message queue, where the **publisher** supplies information in the form of messages and the **subscriber** consumes messages. When the producer publishes this message, it specifies a topic that identifies the subject of data inside the message. And the subscriber is the consumer of this message based on the interest in the message.

1.1.3.2 RPC pattern

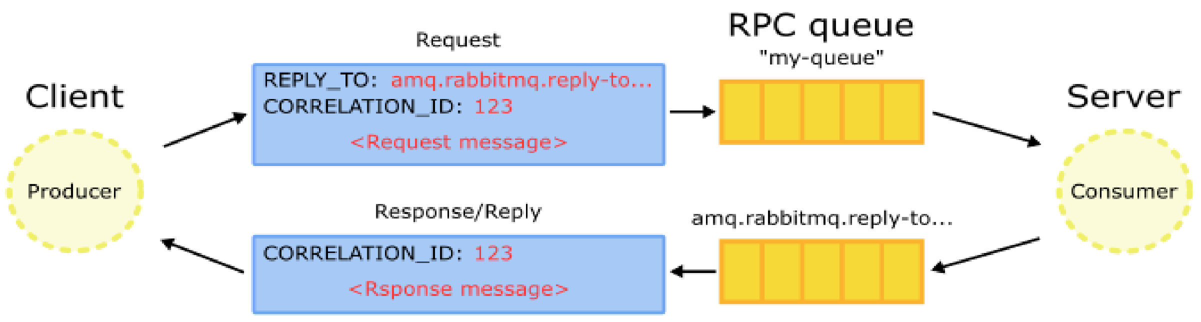


Figure 86: message queue RPC pattern

Detail description: A client sends a request message and a server replies with a response message. In order to receive a response we need to send a 'callback' queue address with the request. The client distingunish reply messages by correlation id.

1.1.4 Microservice system flow

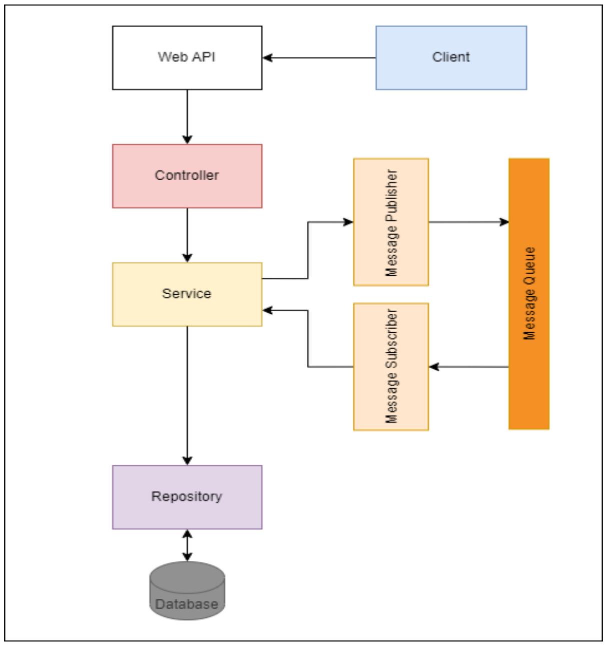


Figure 87: system main flow

Description: Main flow describes communications and data flow of microservice

Detail Descriptions:

Client requests will forward to the related service to handle business logic depending on the HTTP request method and path. Service components perform operations and store data in the database through the repository component. They can call the message publishers to publish messages to the event bus for other services that subscribe to it and trigger action for that message event.

1.1.5 Saga Pattern for Distributed transactions

Problem: The microservices system is designed that allows services could run concurrently to reach high performance requirement. However, there are many problems that require services to process requests synchronously. This problem is called Distributed transactions.

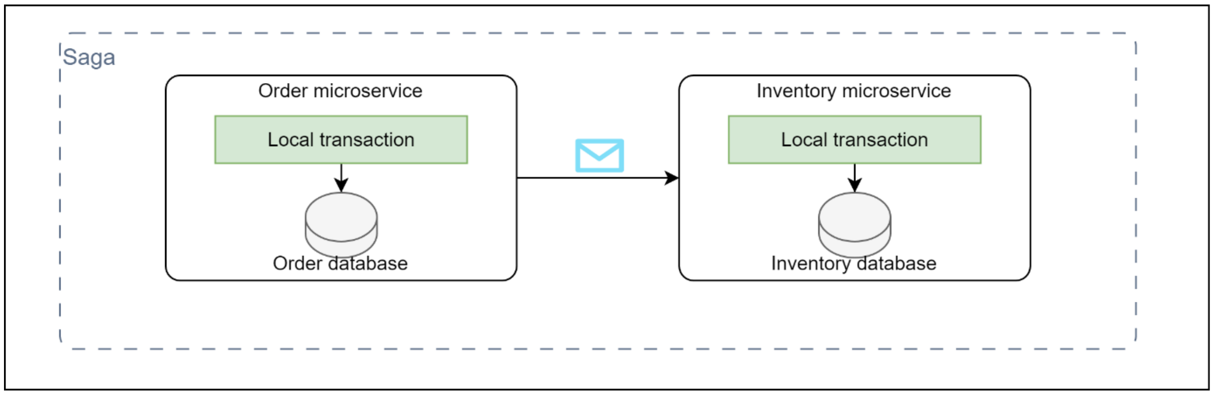


Figure 88: distribute transaction

In our system, there is a problem: When the customer order product, the product variant must be in stock, and have enough quantity for the order, then the quantity of the ordered product variant will be reduced amount equal to the quantity in the order. When local transactions in inventory microservice are completed, the transaction at order microservice could be completed.

Solution:

Choreography with dual write pattern: Choreography is a style of service coordination where participants exchange events without a centralized point of control. With this pattern, each service performs a local transaction and publishes events that trigger local transactions in other services. Each component of the system participates in decision-making about a business transaction's workflow, instead of relying on a central point of control. Historically, the most common implementation for the choreography approach was using an asynchronous messaging layer for the service interactions.

Local-commit-then-publish: commit the local transaction first and then publish the message. This has a small probability of failure occurring after a local transaction has been committed and before publishing the message. But even in that case, you could design your services to be idempotent and retry the operation. That would mean committing the local transaction again and then publishing the message. This approach can work if you control the downstream consumers and can make them idempotent, too. It's also a pretty good implementation option overall.

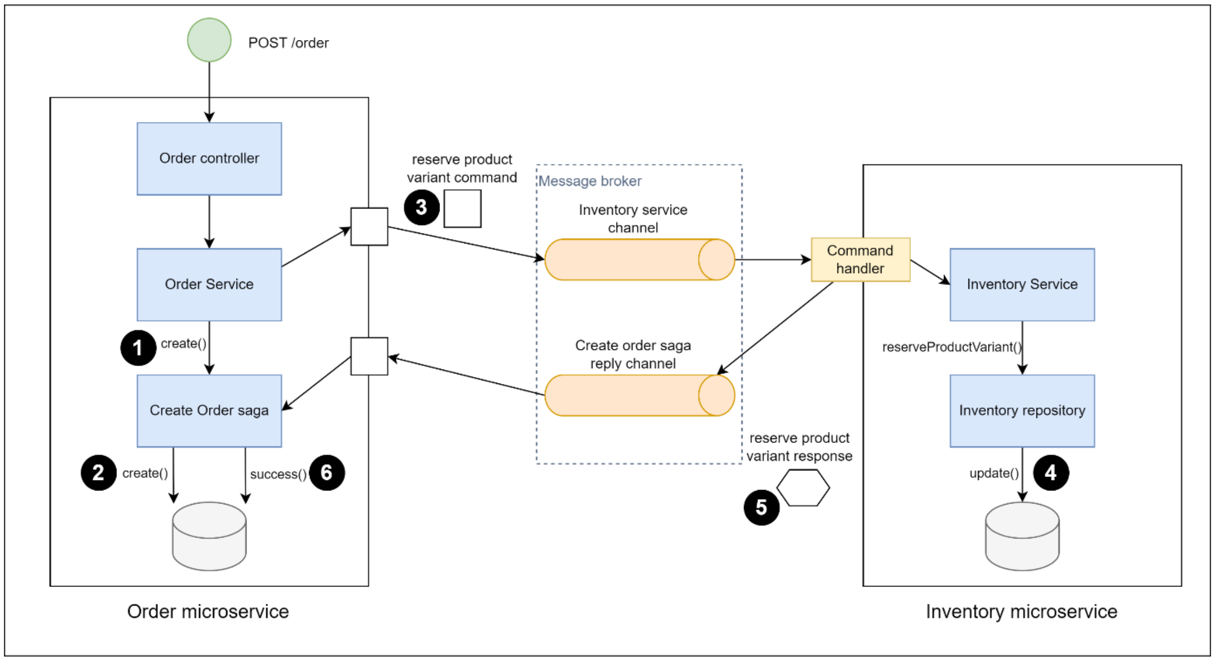
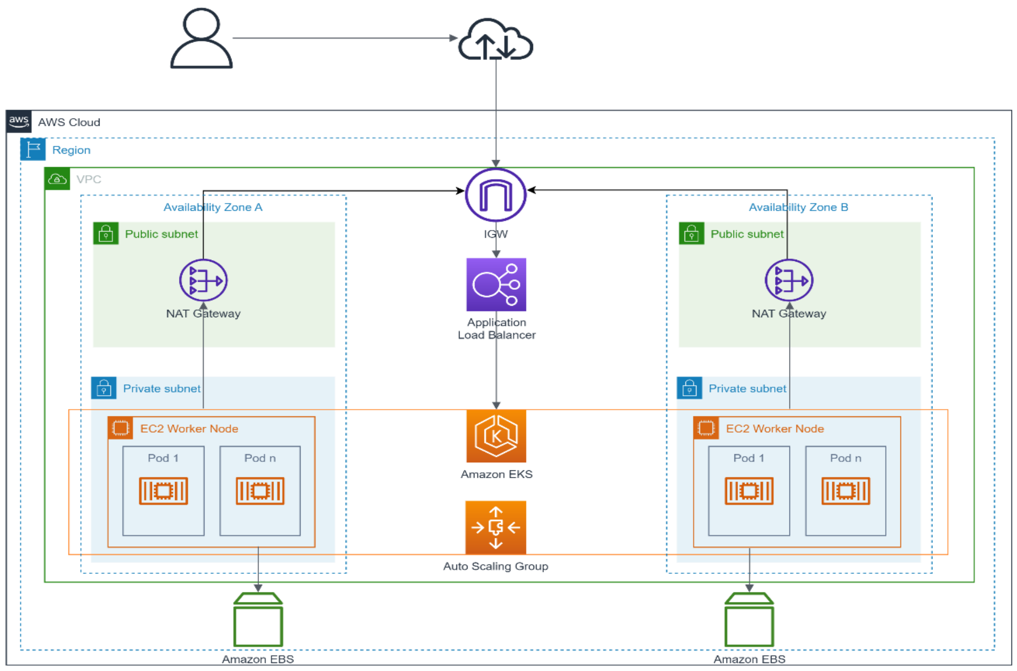


Figure 89: Saga pattern in order product flow

Each Distributed transaction is called a Saga. After the Order microservice handles the business logic, it saves the Saga state as PENDING and publishes a message to the message broker (RabbitMQ). The Inventory service will receive and process that message according to the business logic. It then sends a message to the Saga Reply channel to reply to the status of the order. Finally, when Order microservice receives the message, it will commit the transaction, Saga's status is now SUCCESS.



1.1.6 Deployment acrchitect

Figure 90: deployment diagram

Detail description: One of this project goal is to allow our website can be accessed from internet and system must be deploy in cloud service. We use AWS for deployment our system. The microservices will be containerized to image run in containers. Kubernetes manage container running, deployment and scalling. To launch kubernetes in cloud environment we use Amazon EKS service to manange Kubernetes cluster. Other AWS services are coordination used and will be describe in below table.

Table: Deployment diagram component description

No	Components	Description
01	AWS Region	AWS Regions are physical locations around the world where Amazon clusters data centers for application and service delivery in AWS Availability Zones.
02	AWS Availability Zone	Availability Zones are distinct locations within an AWS Region that are engineered to be isolated from failures in other Availability Zones. They provide inexpensive, low-latency network connectivity to other Availability Zones in the same AWS Region. Important. Each region is completely independent.
03	VPC	Enables you to launch AWS resources into a virtual network that you've defined. This virtual network closely resembles a traditional

		network that you'd operate in your own data center, with the benefits of using the scalable infrastructure of AWS.
04	Internet gateway	An internet gateway is a horizontally scaled, redundant, and highly available VPC component that allows communication between your VPC and the internet
05	Public subnet	Is a subnet that's associated with a route table that has a route to an internet gateway
06	Private subnet	A subnet within a Virtual Private Cloud (VPC) that does not have direct access to the internet. This means that instances running in a private subnet cannot communicate with the internet or be accessed directly from the internet.
07	NAT gateway	A highly available AWS managed service that makes it easy to connect to the Internet from instances within a private subnet in an Amazon Virtual Private Cloud (Amazon VPC)
08	Amazon EKS	Automatically manages the availability and scalability of the Kubernetes control plane nodes on AWS cloud
09	EC2 worker node	The virtual machines that run containerized applications in a Kubernetes cluster managed by AWS EKS.
10	Auto scalling group	Auto Scaling Group is a feature provided by AWS that enables automatic scaling of worker nodes in an EKS cluster based on the demand of the containerized applications running on the cluster
11	Pod	The and most basic unit in a Kubernetes cluster, representing a single instance of a running containerized application.
12	AWS EBS	A block-level storage service that provides durable, low-latency, and scalable storage volumes for use with EC2 instances in AWS
13	AWS LoadBalancer	A managed service that distributes incoming network traffic across multiple Amazon EC2 instances, containers, or IP addresses in multiple Availability Zones (AZs) to ensure high availability and fault tolerance for applications running on AWS. Load balancers help distribute traffic evenly across multiple instances or containers to optimize performance, improve availability, and provide fault tolerance for applications.

1.2 Package Diagram

1.2.1 Back-end package diagram

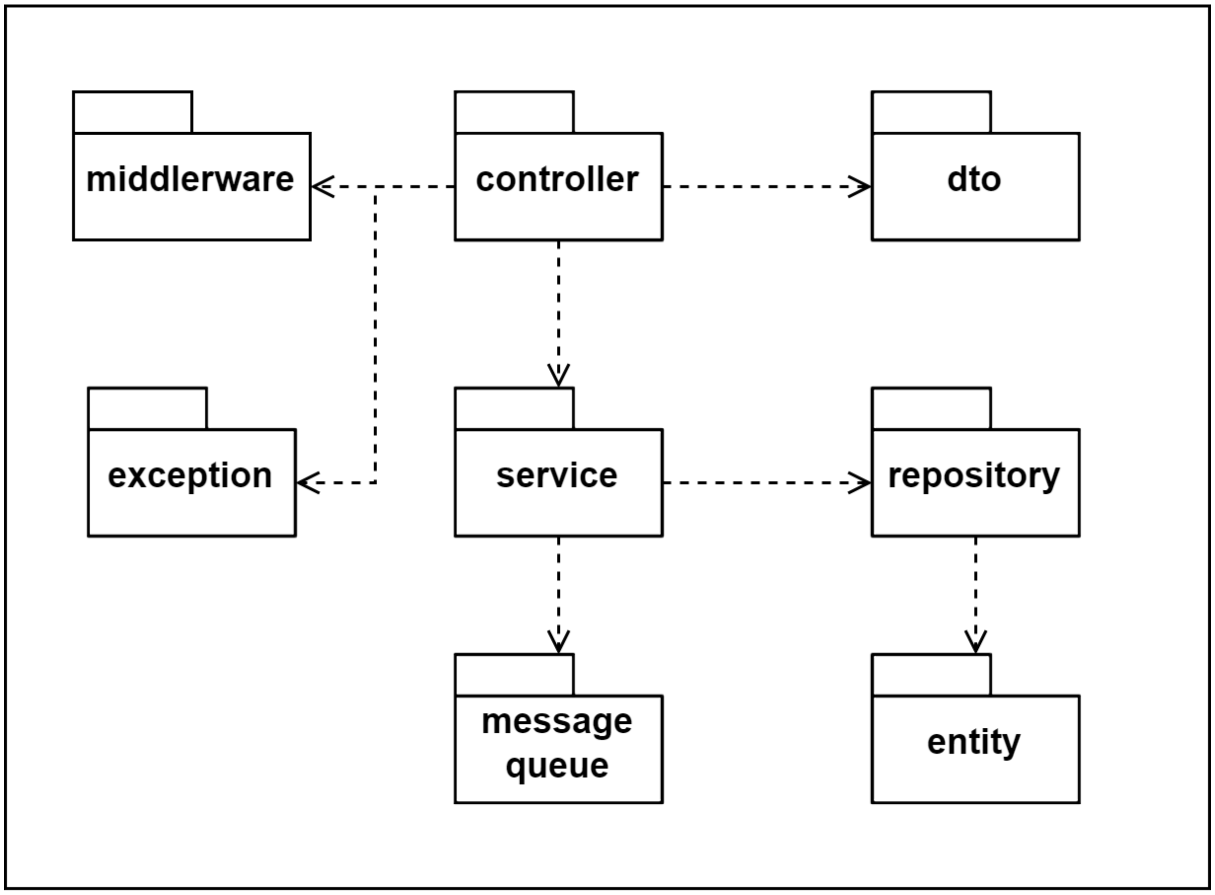


Figure 87:Package diagram

No	Package	Description
01	middlerware	Interceptor of each requests and perform authorization on requests to controller.
02	controller	Contains class file for the controllers. These contain RestAPI to execute requests and return responses for client.
03	dto	Contains Data Transfer Object class file that mapping for resquest and response object model
04	service	Contains class files that execute flexible logic business of the application.
05	repository	Contains repository interface that directly execute query and receive data from database.
06	entity	Contains database entity model class file
07	message queue	Publish and subsribe message of event bus
08	exception	Contains exception response model calss file

2. Database Design

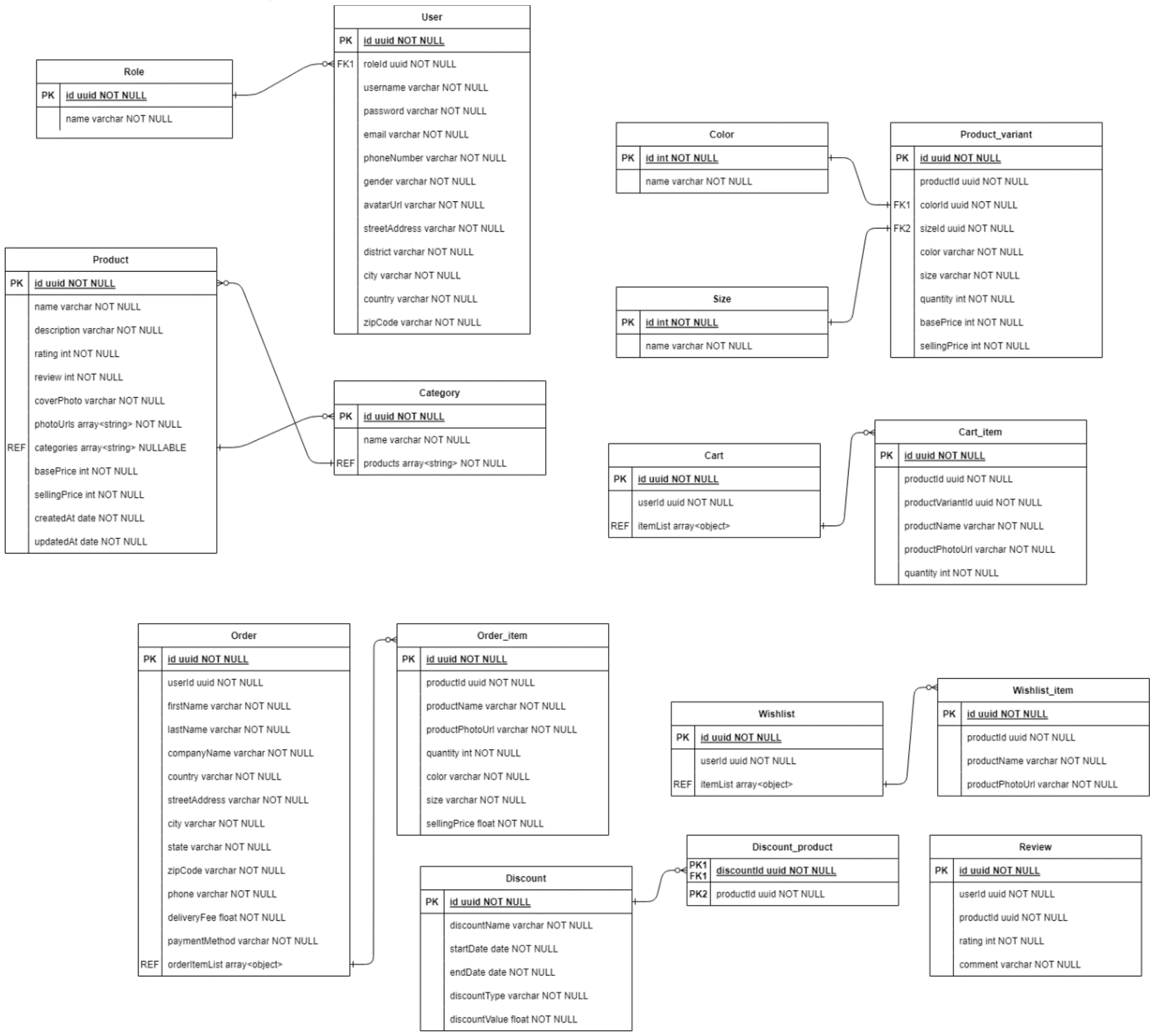


Table Descriptions

No	Entity / Table	Description
01	**User**	1. **Primary keys**: id (id of the user) 2. **Foreign keys**: roleId (id of the role) 3. **Other fields**: • username: the username is used to log in to the system. • password: the password is used to log in to the system. • email: email address of the user. • phoneNumber: the phone number of the user. • gender: the gender of the user. • avatarUrl: the url to the user photo. • streetAddress: the street address of the user. • district: the district of the user.

		• city: the city of the user. • country: the country of the user. • zipCode: the zip code of the user living area.
02	**Role**	1. **Primary keys**: id (id of the role) 2. **Foreign keys**: none. 3. **Other fields**: • name: the name of the role.
03	**Product**	1. **Primary keys**: id (id of the product) 2. **Foreign keys**: none. 3. **Other fields**: • name: the name of the product. • description: the description of the product. • rating: the rating of the product. • review: the reviews of the product. • coverPhoto: the url to the product photo. • photoUrls: the list urls of product sub photos. • categories: the list category ids (reference from the category table). • basePrice: the base price of the product. • sellingPrice: the price uses for sale of the product. • createdAt: the date time of the product creation. • updatedAt: the date time of the product edition.
04	**Category**	1. **Primary keys**: id (id of the category) 2. **Foreign keys**: none. 3. **Other fields**: • name: the name of the category. • products: the list product ids (reference from the product table).
05	**Color**	1. **Primary keys**: id (id of the color) 2. **Foreign keys**: none. 3. **Other fields**: • name: the name of the color.
06	**Size**	1. **Primary keys**: id (id of the size) 2. **Foreign keys**: none. 3. **Other fields**: • name: the name of the size.

07	**Product_variant**	1. **Primary keys**: id (id of the product variant) 2. **Foreign keys**: • colorId (id of the color) • sizeId (id of the size) 3. **Other fields**: • productId: the id of the product. • color: the color name of the product variant. • size: the size name of the product variant. • quantity: the quantity of the product variant. • basePrice: the base price of the product variant. • sellingPrice: the price uses for sale of the product variant.
08	**Cart**	1. **Primary keys**: id (id of the cart) 2. **Foreign keys**: none. 3. **Other fields**: • userId: the id of the user. • itemList: the list of cart items (reference from the cart_item table).
09	**Cart_item**	1. **Primary keys**: id (id of the cart item) 2. **Foreign keys**: none. 3. **Other fields**: • productId: the id of the product. • productVariantId: the id of the product variant. • productName: the name of the product. • productPhotoUrl: the url to the product photo. • quantity: the quantity of the product.
10	**Wishlist**	1. **Primary keys**: id (id of the wishlist) 2. **Foreign keys**: none. 3. **Other fields**: • userId: the id of the user. • itemList: the list of wishlist items (reference from the wishlist_item table).
11	**Wishlist_item**	1. **Primary keys**: id (id of the wishlist item) 2. **Foreign keys**: none. 3. **Other fields**: • productId: the id of the product. • productName: the name of the product. • productPhotoUrl: the url to the product photo.
12	**Order**	1. **Primary keys**: id (id of the order) 2. **Foreign keys**: none. 3. **Other fields**: • userId: the id of the user. • firstName: the first name of the user.

		• lastName: the last name of the user. • companyName: the company name of the user. • country: the country of the user. • streetAddress: the street address of the user. • city: the city of the user. • state: the state of the user. • zipCode: the zip code of the user living area. • phone: the phone number of the user. • deliveryFee: the delivery fee of the order. • paymentMethod: the payment method of the user. • orderItemList: the list of order items(reference from the order_item table).
13	**Order_item**	1. **Primary keys**: id (id of the order item) 2. **Foreign keys**: none. 3. **Other fields**: • productId: the id of the product. • productName: the name of the product. • productPhotoUrl: the url to the product photo. • quantity: the quantity of the product. • color: the color name of the order item. • size: the size name of the order item. • sellingPrice: the price uses for sale of the product.
14	**Discount**	1. **Primary keys**: id (id of the discount) 2. **Foreign keys**: none. 3. **Other fields**: • discountName: the name of the discount. • startDate: the start date of the discount. • endDate: the end date of the discount. • discountType: the type of the discount. • discountValue: the sale value of the discount.
15	**Discount_product**	1. **Primary keys**: • discountId (reference from the discount table). • productId: the id of the product. 2. **Foreign keys**: discountId (id of the discount) 3. **Other fields**: none.
16	**Review**	1. **Primary keys**: id (id of the review) 2. **Foreign keys**: none. 3. **Other fields**: • userId: the id of the user. • productId: the id of the product. • rating: the rating of the review. • comment: the comment of the review.

3. Detailed Design

3.1 Class Diagram

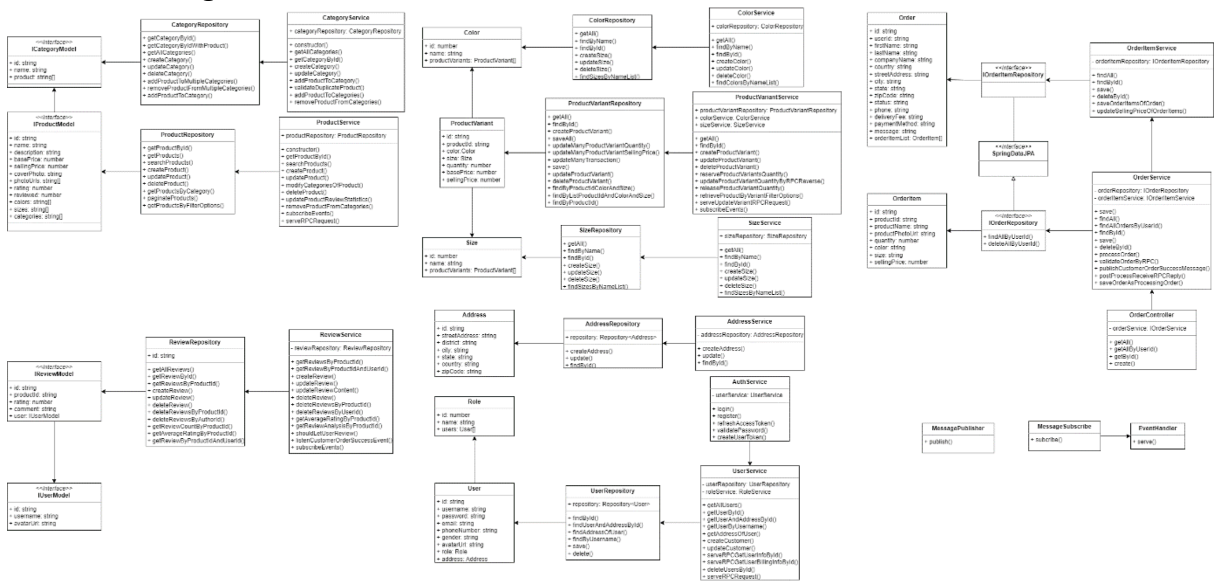


Figure 91: Class diagram

3.2 Main flow sequence diagram

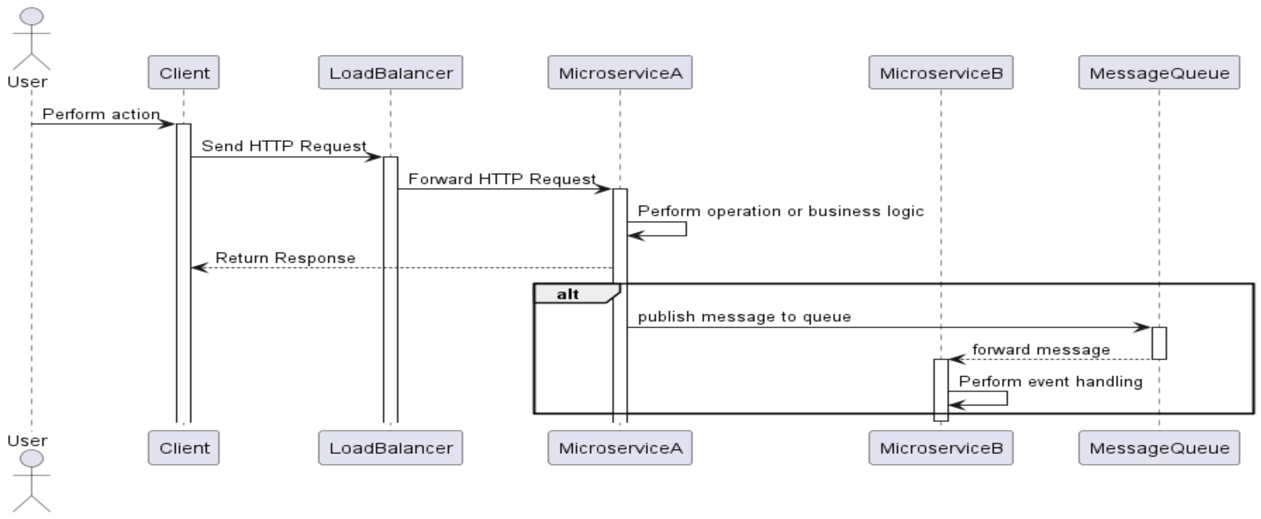


Figure 92: main flow sequence

3.3 Authentication

3.3.1 Common auth

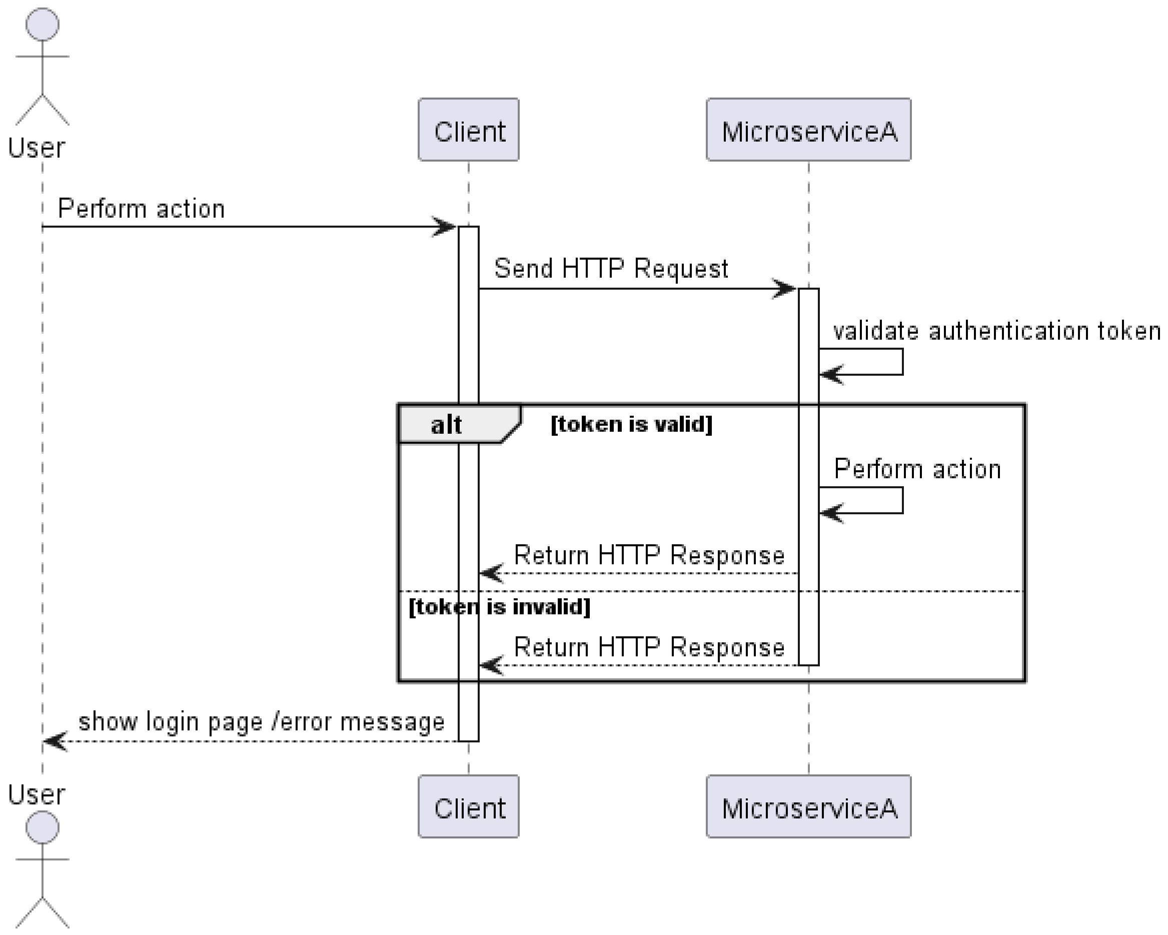


Figure 93 : Common sequence

3.3.2 Register

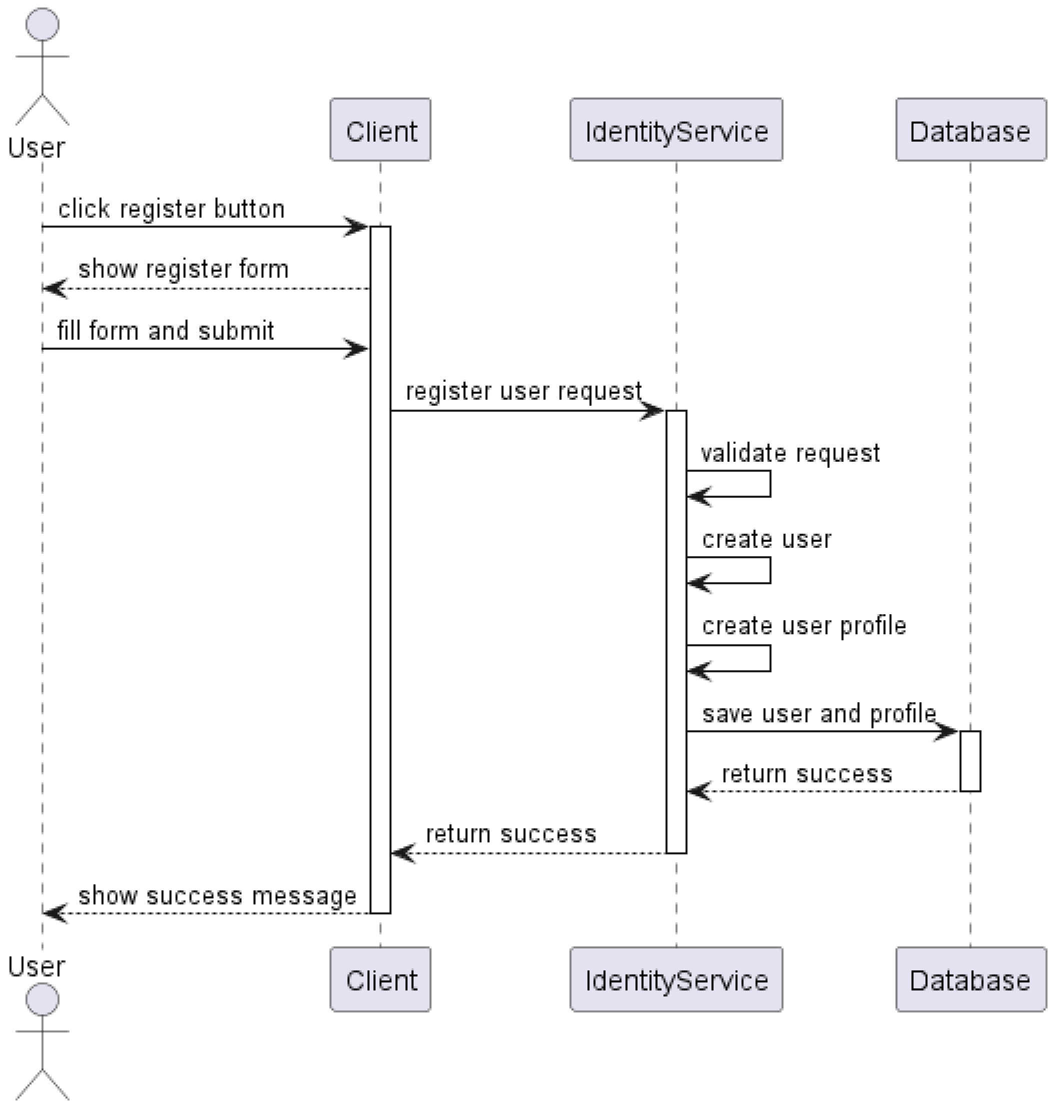


Figure 94: Register sequence diagram

3.3.3 Login

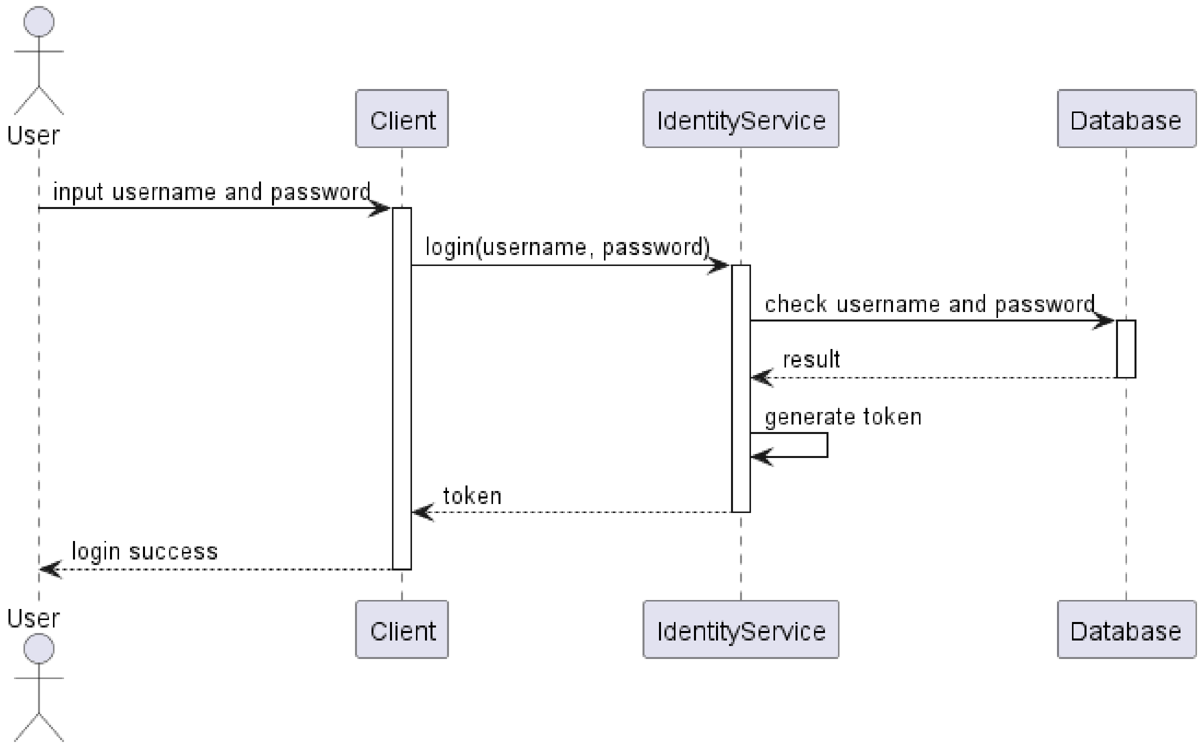


Figure 95: Login sequence diagram

3.2 Product

3.2.1 Paginated products

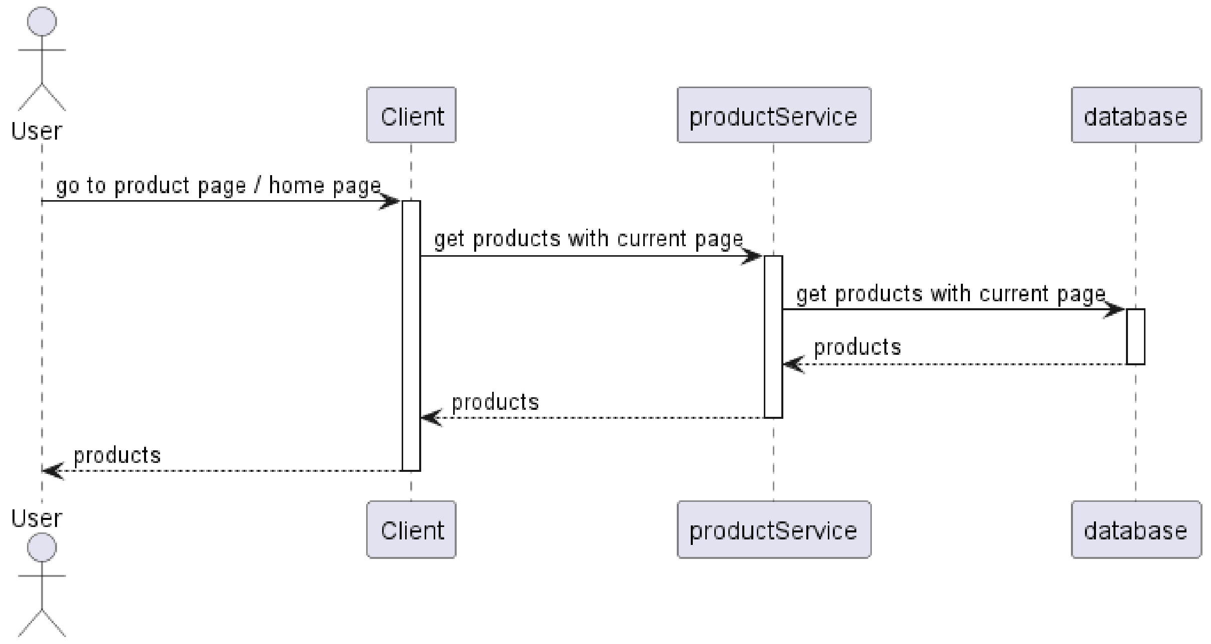

User
Client
productService
database
go to product page / home page
get products with current page
get products with current page
products
products
products

Figure 96: paginated sequence diagram

3.2.2 Paginated and filter products

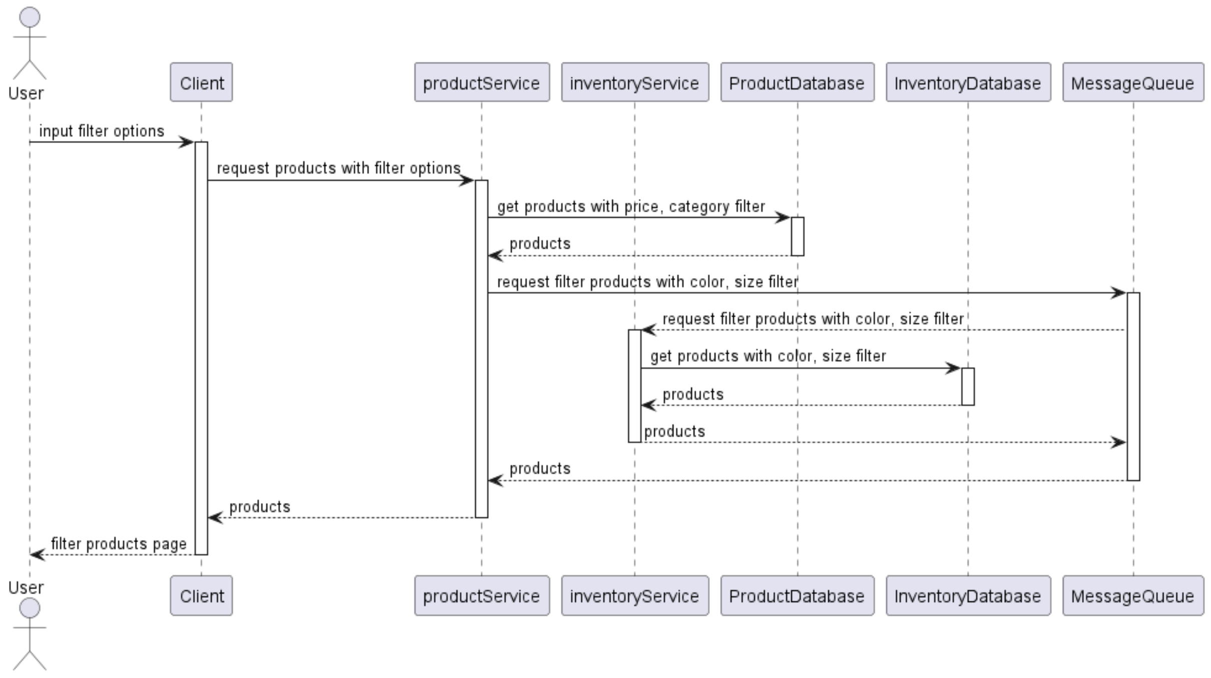

User
Client
productService
inventoryService
ProductDatabase
InventoryDatabase
MessageQueue
input filter options
request products with filter options
get products with price, category filter
products
request filter products with color, size filter
request filter products with color, size filter
get products with color, size filter
products
products
products
products
filter products page

Figure 97: paginated and filter product sequence diagram

3.2.3 Create products

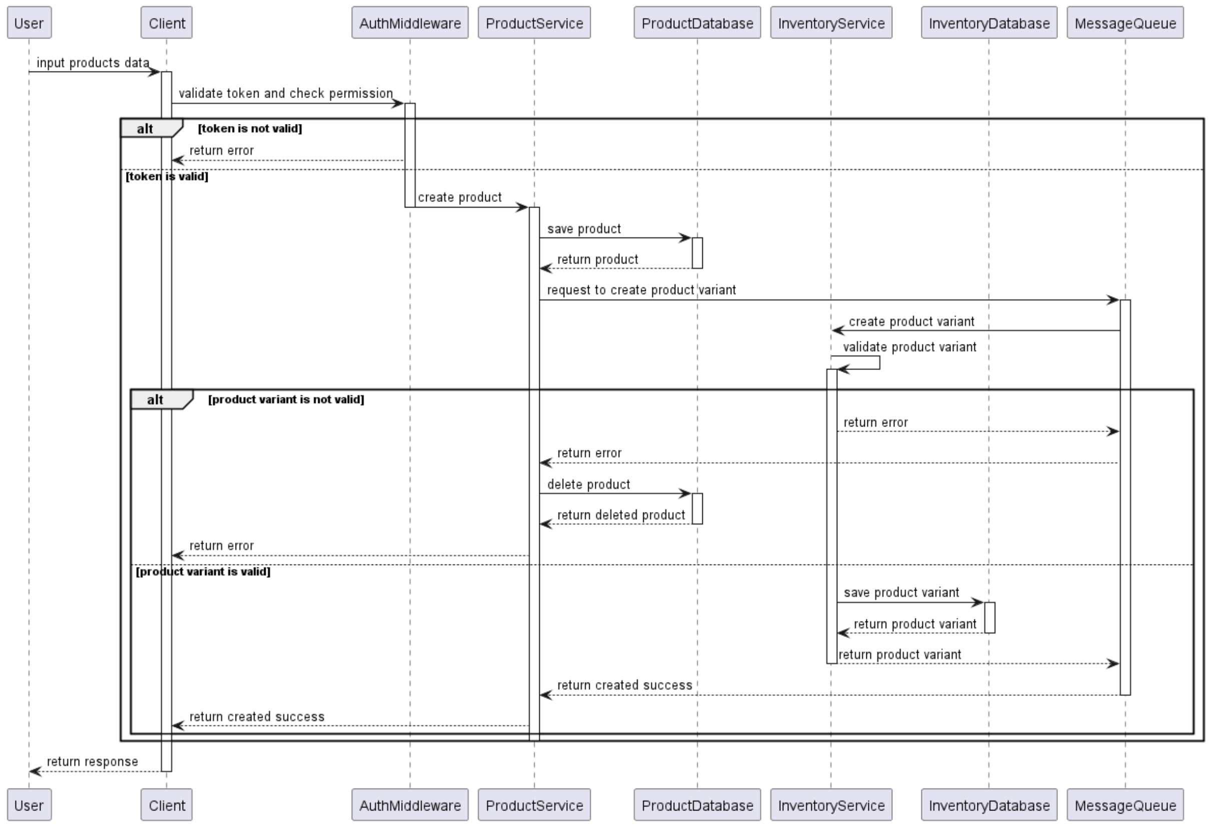


Figure 98: create product sequence diagram

3.2.4 Delete product

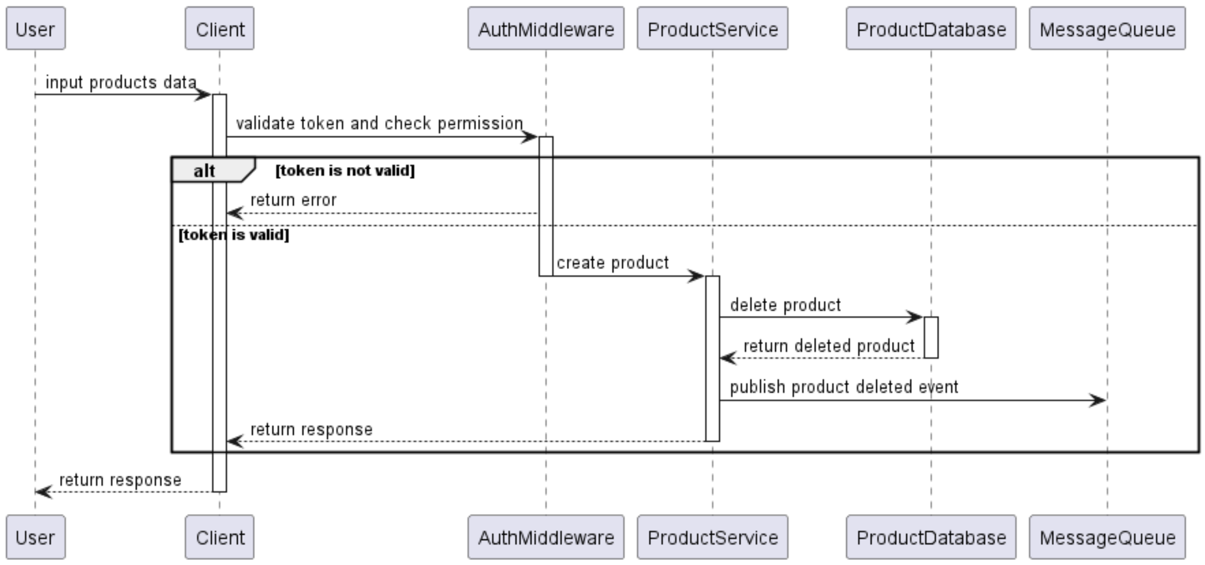


Figure 99: delete product sequence diagram

3.3 Order

3.3.1 Create order

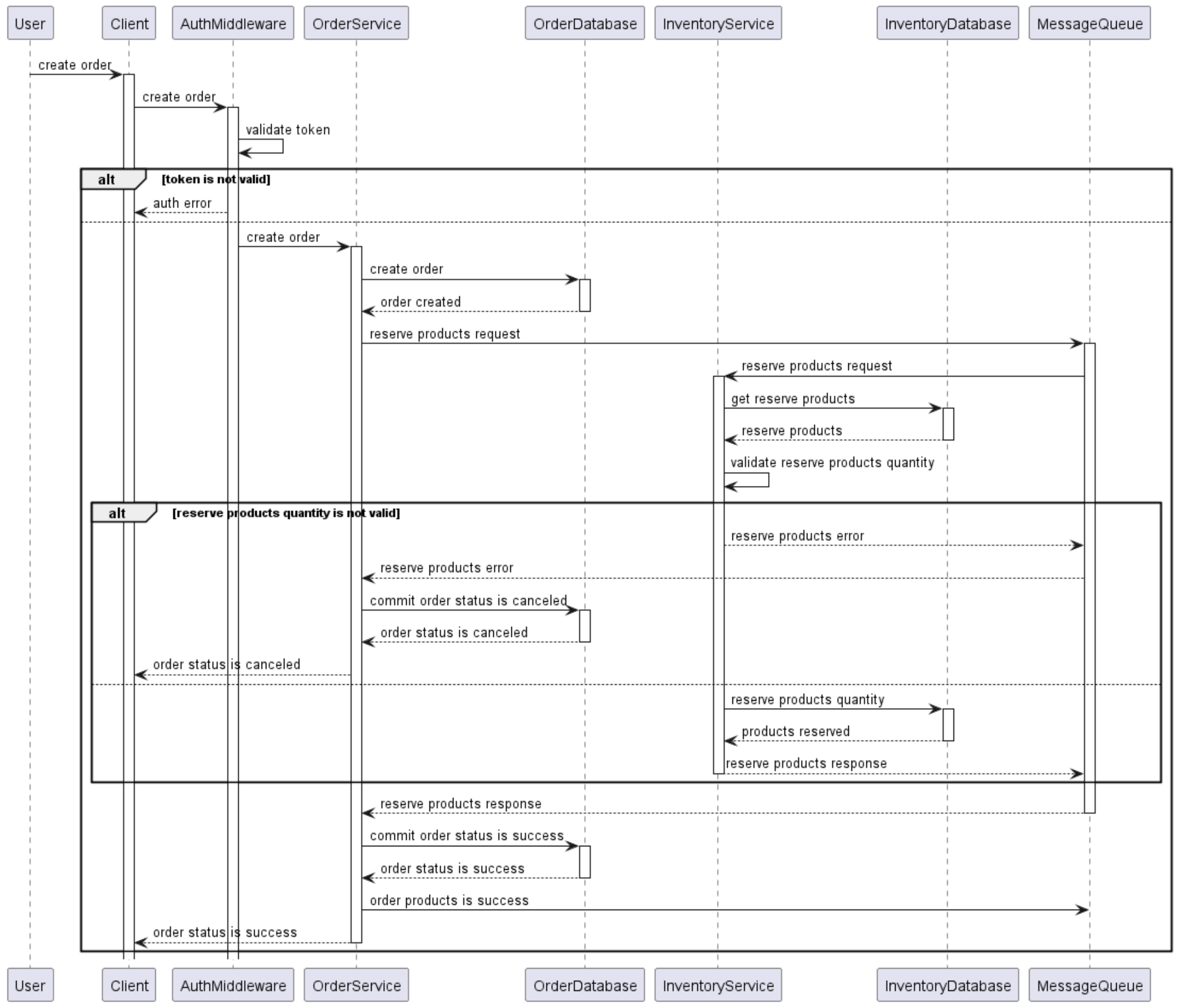


Figure 100: create order sequence diagram

3.3.2 Get all orders of user

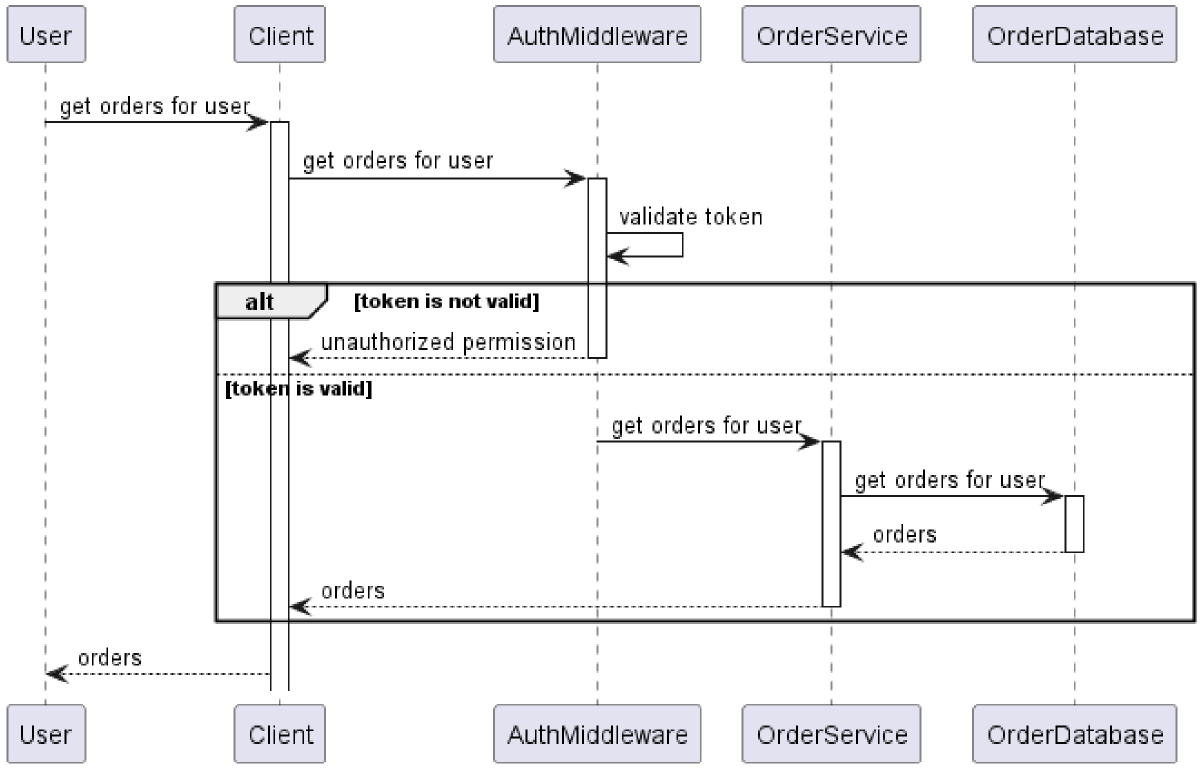


Figure 101: get all order of user sequence diagram

3.4 Review

3.4.1 Allow user review product

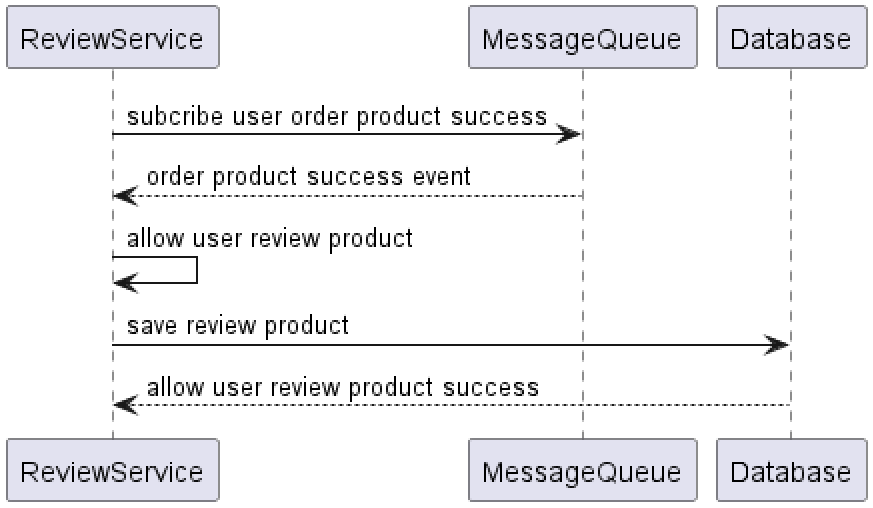

ReviewService
MessageQueue
Database
subcribe user order product success
order product success event
allow user review product
save review product
allow user review product success
ReviewService
MessageQueue
Database

Figure 102: allow user reivew product sequence diagram

3.4.2 Create review for product

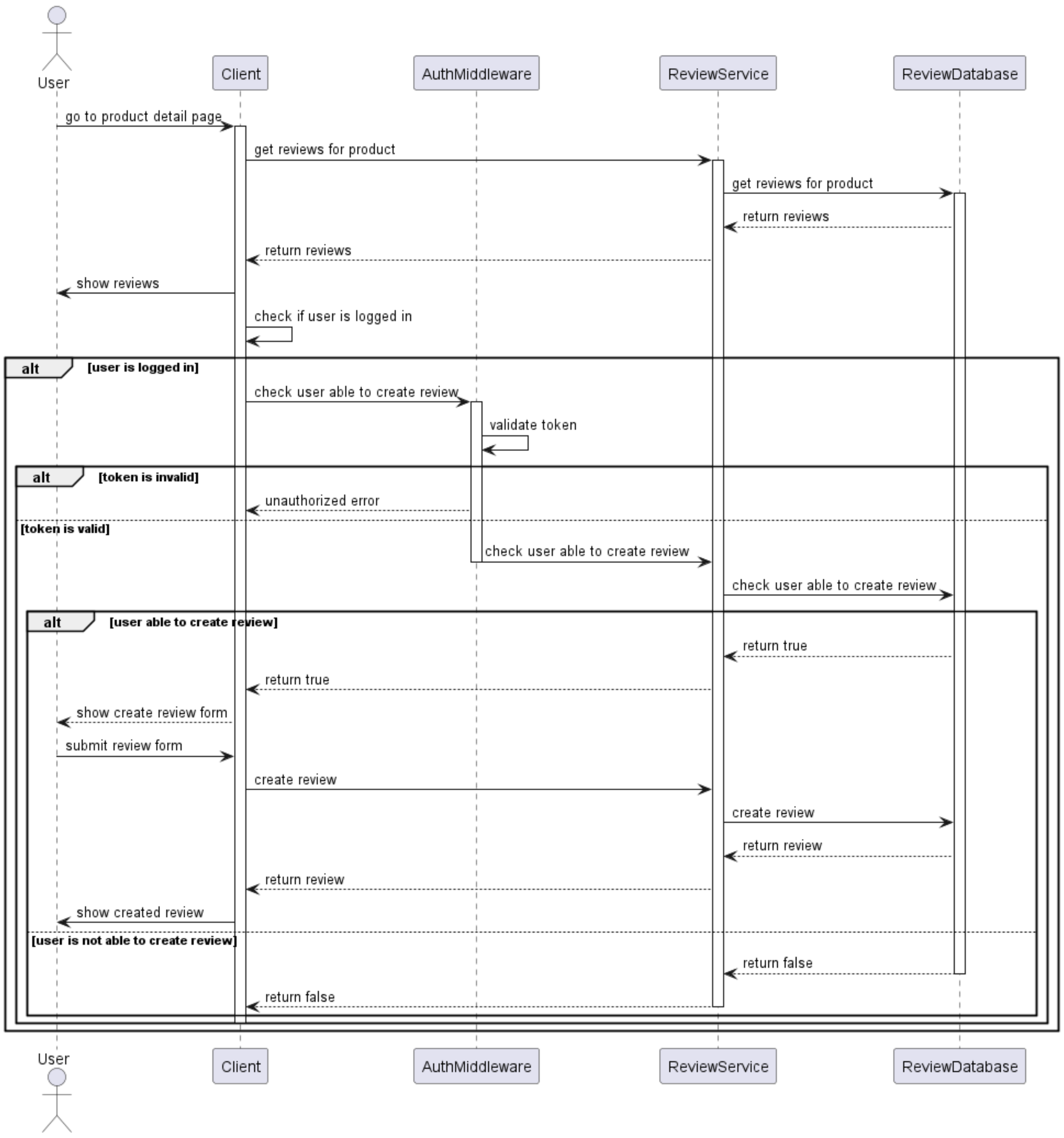


Figure 103: paginated and filter product sequence diagram

3.4.3 Get reviews of product

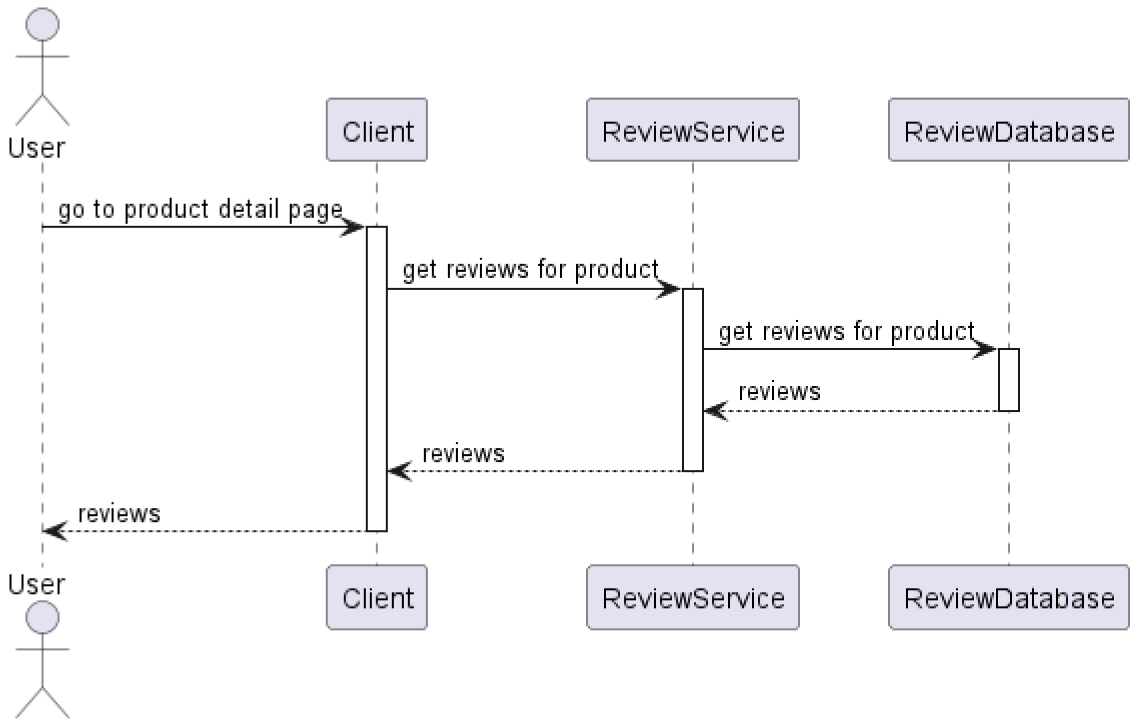


Figure 104: get reviews of product sequence diagram

3.4.4 Delete review of product

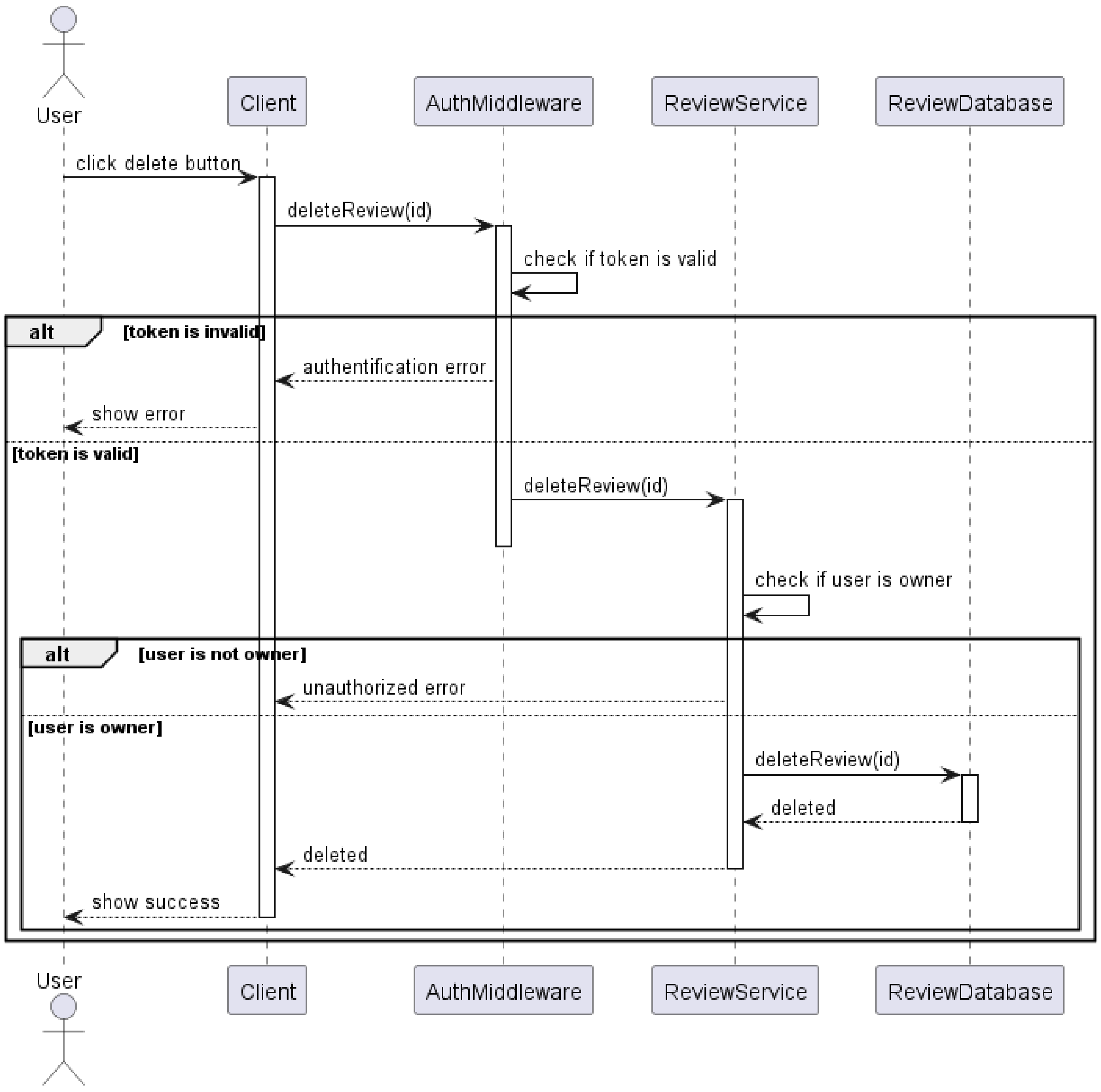


Figure 105: delete review of product sequence diagram

3.4.5 Update product rating analysis

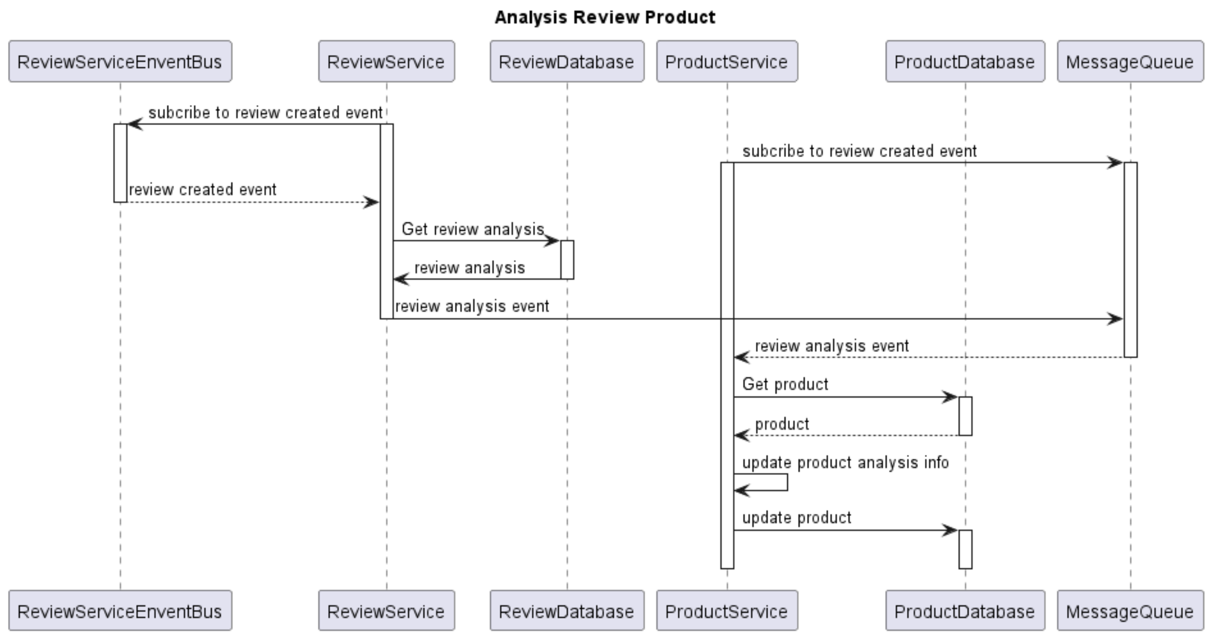


Figure 106: update product rating analysis sequence diagram

3.5 Shopping Cart

3.5.1 Toggle Item in cart

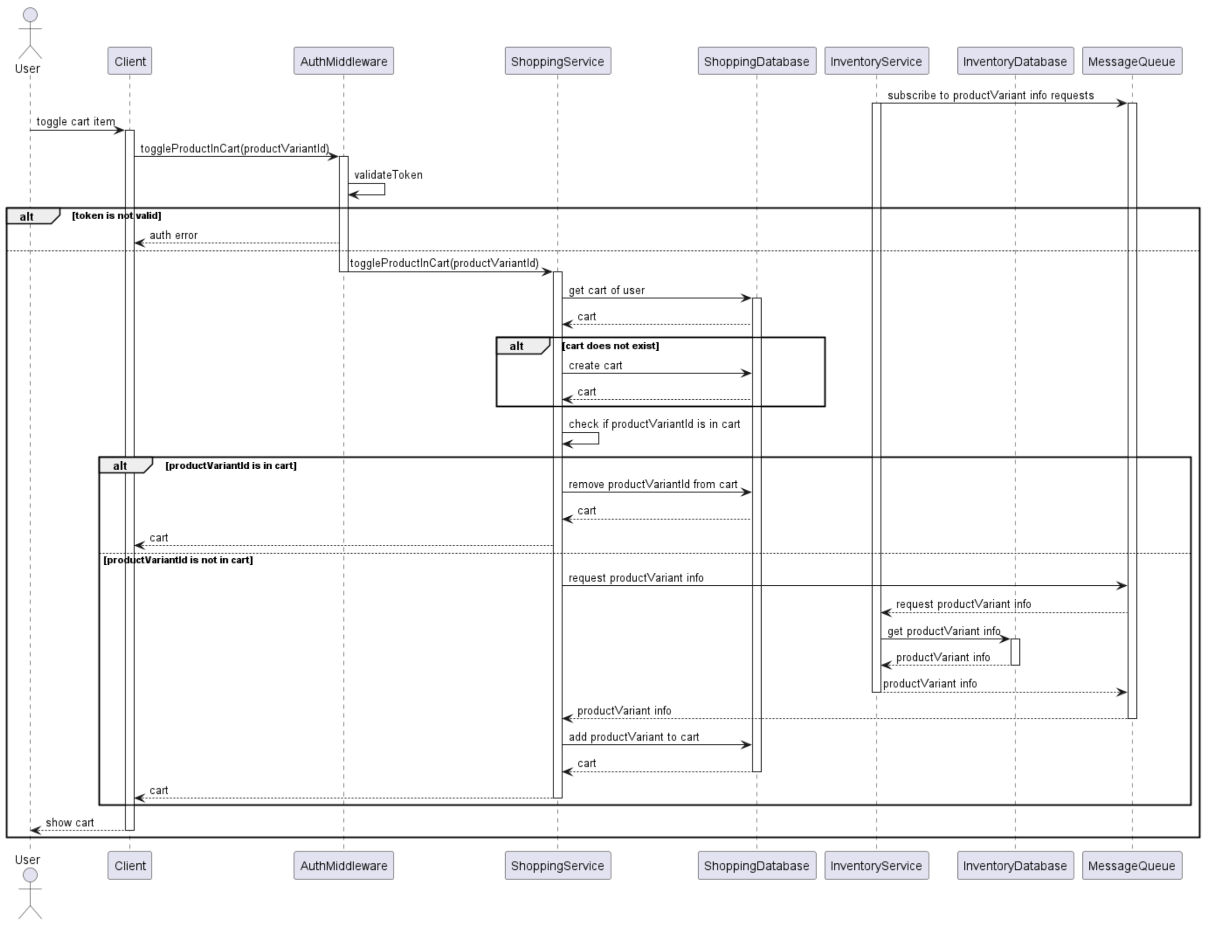


Figure 107: toggle item in cart sequence diagram

3.6 User wishlist

3.6.1 Toggle Item in wishlist

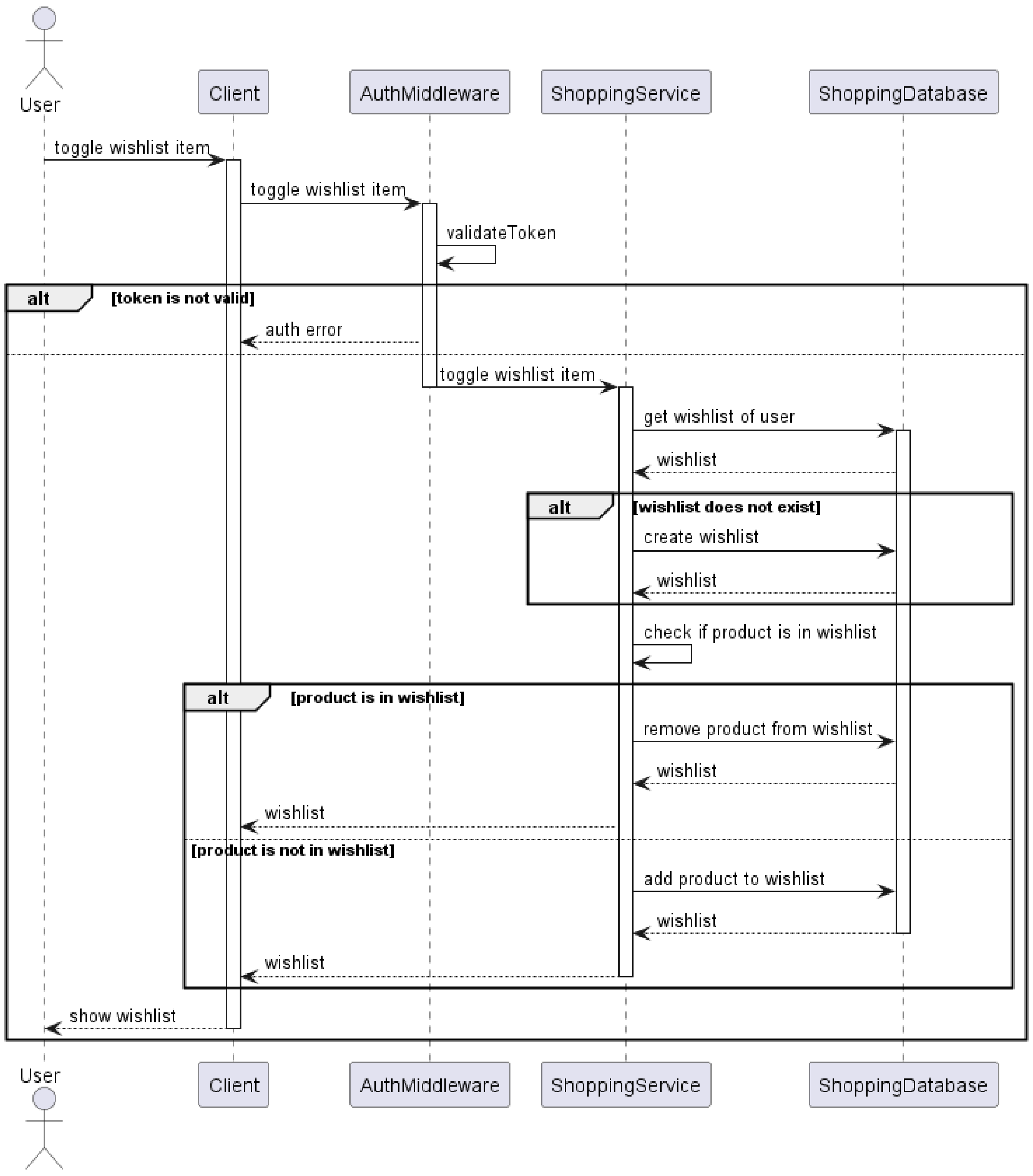


Figure 108: toggle item in wishlist sequence diagram

3.7 Discount

3.7.1 Create discount

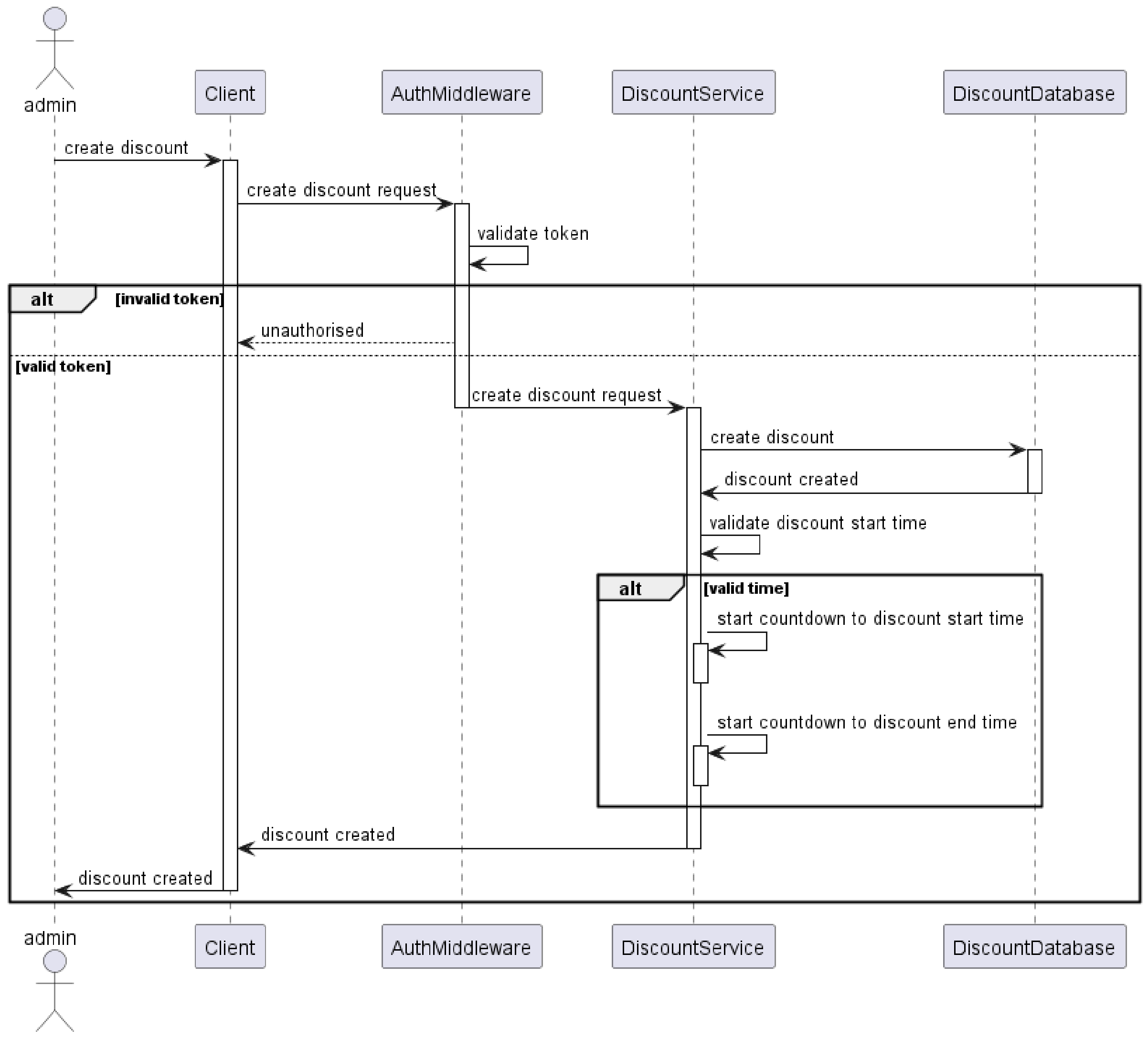


Figure 109: create discount sequence diagram

3.7.2 End discount

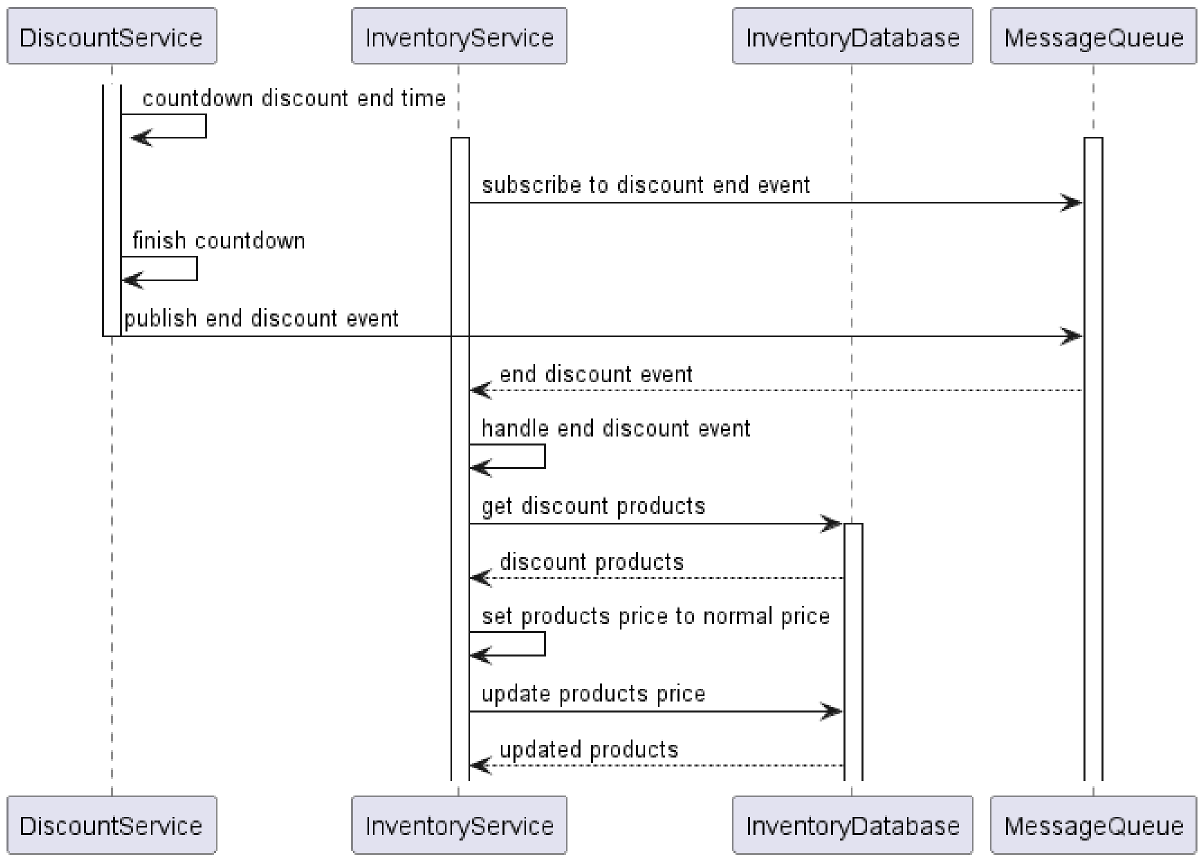


Figure 110: end discount sequence diagram

3.7.2 Launch discount

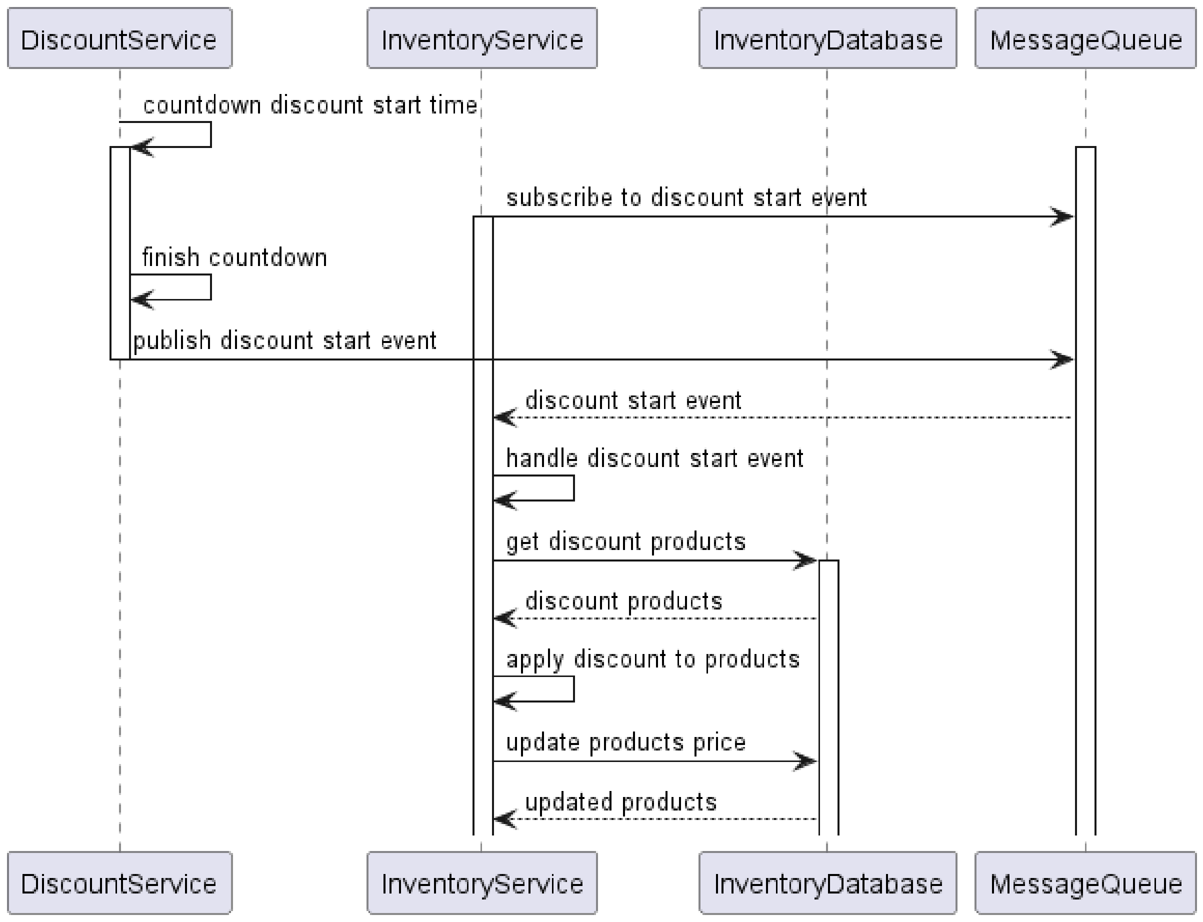


Figure 111: launch discount sequence diagram