Applied Software Project Report

By

Abhijeet Moreshwar Kale

A Master's Project Report submitted to Scaler Neovarsity - Woolf in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

June 2025

Scaler Mentee Email ID : mr.abhijeetkale@gmail.com

Thesis Supervisor : Naman Bhalla

Date of Submission : 22/06/2025

Certification

I confirm that I have overseen / reviewed this applied project and, in my judgment, it adheres to the appropriate standards of academic presentation. I believe it satisfactorily meets the criteria, in terms of both quality and breadth, to serve as an applied project report for the attainment of Master of Science in Computer Science degree. This applied project report has been submitted to Woolf and is deemed sufficient to fulfill the prerequisites for the Master of Science in Computer Science degree.

Naman Bhalla

……

Project Guide / Supervisor

DECLARATION

I confirm that this project report, submitted to fulfill the requirements for the Master of Science in Computer Science degree, completed by me from 26/07/2024 to 30/09/2024, is the result of my own individual endeavor. The Project has been made on my own under the guidance of my supervisor with proper acknowledgement and without plagiarism. Any contributions from external sources or individuals, including the use of AI tools, are appropriately acknowledged through citation. By making this declaration, I acknowledge that any violation of this statement constitutes academic misconduct. I understand that such misconduct may lead to expulsion from the program and/or disqualification from receiving the degree.

Abhijeet Moreshwar Kale

<Signature of the Candidate> **Date: 22 June 2025**

ACKNOWLEDGMENT

I am profoundly grateful to everyone who contributed to the successful completion of the backend module for the *E-commerce Website* project.

First and foremost, I extend my deepest thanks to Anurag Khanna, module instructor for their invaluable guidance and expertise throughout the development of this project. Their constructive feedback and technical insights were instrumental in designing and implementing the backend architecture using Java, Spring Boot, MySQL, Elasticsearch, Apache Kafka, Redis, MongoDB and API Gateway.

I also wish to express my gratitude to Scaler for providing the resources, tools, and collaborative environment necessary to bring this project to life. Special thanks to module colleagues for their support in debugging, optimizing queries, and ensuring seamless integration of core functionalities such as user authentication, product management, and payment processing.

I would like to acknowledge my peers and colleagues for their encouragement and assistance in tackling challenges during various stages of development, from creating RESTful APIs to optimizing database performance. Their input played a vital role in improving the robustness and scalability of the backend system.

Finally, I am immensely thankful to my family for their unwavering support, patience, and encouragement, which gave me the strength to complete this project successfully.

This achievement would not have been possible without the collective effort and contributions of everyone involved. Thank you for being a part of this journey.

Table of Contents

List of Tables **6**
List of Figures **7**
Applied Software Project **9**
Abstract 9
Project Description 10
Requirement Gathering 12
Class Diagrams 14
Database Schema Design 39
Feature Development Process 43
Deployment Flow 45
Technologies Used 47
Conclusion 50
References **51**

List of Tables

Table No.	Title	Page No.
1.1	Comparison between Synchronous and Asynchronous communication	11
5.1	Differences between Caching and Traditional DB Queries	44
7.1	Comparison between MySQL and MongoDB	47
7.2	Comparison between Apache Kafka and RabbitMQ	48
7.3	Differences between Load Balancer and API Gateway	49

List of Figures

Figure No.	Title	Page No.
1.1	Agile Software Project Development Process	11
2.2	Flow Diagram	13
3.1.1	User Management Models	14
3.1.2	User Management Repository	15
3.1.3	User Management Controller	15
3.1.4	User Management Service	16
3.1.5	User Management Configurations	17
3.1.6	User Management Exceptions	17
3.1.7	User Management Dtos	18
3.1.8	User Management Kafka Event Producer	19
3.2.1	Product Catalog Models	19
3.2.2	Product Catalog JPA Repository	20
3.2.3	Product Catalog ElasticSearch Repository	20
3.2.4	Product Catalog Controller	20
3.2.5	Product Catalog Exceptions	20
3.2.6	Product Catalog Service	21
3.2.7	Product Catalog Dtos	22
3.2.8	Product Catalog Configurations	22
3.3.1	Cart Models	23
3.3.2	Cart Repository	23
3.3.3	Cart Service	24
3.3.4	Cart Dtos	25
3.3.5	Cart Controller	26
3.3.6	Cart Configuration	27
3.3.7	Cart Exceptions	27
3.4.1	Order Management Models	28
3.4.2	Order Management Repository	28
3.4.3	Order Management Service	29
3.4.4	Order Management Controller	30
3.4.5	Order Management Dtos	30
3.4.6	Order Management Configurations	31
3.4.7	Order Management Exceptions	31
3.4.8	Order Management Kafka Producer	32

3.4.9	Order Management Kafka Consumer	32
3.5.1	Payment Models	33
3.5.2	Payment Repository	33
3.5.3	Payment Service	34
3.5.4	Payment Controller	34
3.5.5	Payment Dtos	35
3.5.6	Payment Configurations	35
3.5.7	Payment Exceptions	36
3.5.8	Payment Kafka Producer	36
3.5.9	Payment Kafka Consumer	36
3.6.1	Notification Models	37
3.6.2	Notification Service	37
3.6.3	Notification Dtos	38
3.6.4	Notification Kafka Consumer	38
4.1	User Management Schema	39
4.2	Product Catalog Schema	40
4.3	Order Management Schema	41
4.4	Payment Schema	42

Applied Software Project

Abstract

This project presents the design and implementation of a scalable, microservices-based **e-commerce application** that provides a seamless and efficient backend infrastructure for managing an online shopping platform. In response to the growing need for modular, high-performance systems, this application adopts a microservices architecture that allows for independent development, deployment, and scaling of core services.

The platform is composed of multiple dedicated microservices:

- **User Management Service** handles user registration, authentication, and session management.
- **Product Catalog Service** maintains product listings, categories, and inventory data.
- **Cart Service** enables users to manage their shopping cart contents.
- **Order Management Service** processes order placement and tracks their status.
- **Payment Service** simulates secure transaction handling.
- **Notification Service** dispatches real-time updates such as order confirmations and payment status.

These services communicate via **REST APIs** and utilize **Apache Kafka** for asynchronous, event-driven interactions. An **API Gateway** is implemented to act as a unified entry point, managing routing, authentication, and rate limiting.

The backend system is built using **Spring Boot**, with **MySQL** as the primary data store for persistence. Using **MongoDB** as a primary data store for Cart Service to handle dynamic nature of the cart. **Redis caching** is integrated to improve performance by reducing latency and database load, particularly for frequently accessed data such as product listings and user sessions.

This project demonstrates the practical application of modern backend architectural principles in building a robust e-commerce system. It lays the groundwork for further enhancements such as containerized deployment, horizontal scalability, and CI/CD automation.

Project Description

The goal of this project is to design and implement a cloud-ready, microservices-based **e-commerce backend system** that facilitates end-to-end management of online shopping operations. The project aims to address the core requirements of a modern e-commerce platform—scalability, modularity, maintainability, and performance—by decomposing the application into independently deployable services.

Key Features

The system is divided into multiple microservices, each handling a specific domain:

1. **User Management Service**
   - Handles user registration, authentication (JWT-based), and profile management.
   - Implements secure session handling with MySQL for performance and scalability.
2. **Product Catalog Service**
   - Manages product data, categories, and inventory.
   - Exposes APIs to add, update, retrieve, and delete products.
   - Uses Elasticsearch for faster product lookups with full text search.
3. **Cart Service**
   - Allows users to add, update, and remove items from their shopping cart.
   - Stores cart state independently and ensures consistency across user sessions.
   - Uses Redis for faster retrieval of Cart
4. **Order Management Service**
   - Handles order creation, order status tracking, and historical order retrieval.
   - Integrates with the Payment Service and sends events to Kafka.
5. **Payment Service**
   - Simulates payment processing using test flows.
   - Generates transaction IDs and validates payment status.
6. **Notification Service**
   - Sends order confirmation and payment status notifications via email or sms.
   - Listens to Kafka topics for event-driven messaging.
7. **API Gateway**
   - Acts as a single-entry point to the system.
   - Handles request routing, load balancing.

Technology Stack

- **Backend Framework**: Spring Boot (Java)
- **Database:** MySQL, MongoDB
- **Cache:** Redis
- **Messaging:** Apache Kafka
- **Security:** JWT-based authentication
- **API Gateway/Load Balancer:** HAProxy

Inter-Service Communication

Services communicate using **REST APIs** for synchronous operations and **Kafka** for asynchronous event-driven processing. This design ensures high availability and responsiveness even under heavy load.

Table 1.1: Comparison between Synchronous and Asynchronous communication

Feature	Synchronous	Asynchronous
Communication Pattern	Request → Response	Event → Message → Acknowledgement
Blocking	Yes	No
Coupling	Tighter	Looser
Resilience	Low (failures propagate fast)	High (messages can be retried)
Complexity	Lower	Higher (retry, ordering, delivery)
Tools	REST, gRPC, Feign	Kafka, RabbitMQ, SQS

Objectives Achieved

- Clear separation of concerns with dedicated services
- Scalable and loosely coupled architecture
- High-performance backend with Redis caching
- Resilient inter-service communication via Kafka
- Centralized access and routing through an API Gateway

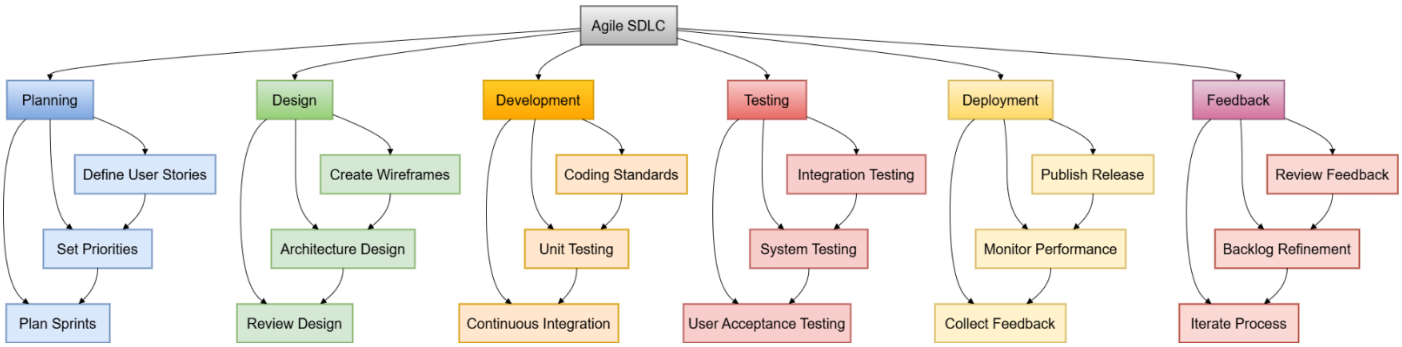

Figure 1.1: Agile Software Project Development Process

Requirement Gathering

1. **User Management**
   - Registration: Allow new users to create an account using their email or social media profiles.
   - Login: Users should be able to securely log in using their credentials.
   - Profile Management: Users should have the ability to view and modify their profile details.
   - Password Reset: Users must have the option to reset their password through a secure link.
2. **Product Catalog**
   - Browsing: Users should be able to browse products by different categories.
   - Product Details: Detailed product pages with product images, descriptions, specifications, and other relevant information.
   - Search: Users must be able to search for products using keywords.
3. **Cart & Checkout**
   - Add to Cart: Users should be able to add products to their cart.
   - Cart Review: View selected items in the cart with price, quantity, and total details.
   - Checkout: Seamless process to finalize the purchase, including specifying delivery address and payment method.
4. **Order Management**
   - Order Confirmation: After making a purchase, users should receive a confirmation with order details.
   - Order History: Users should be able to view their past orders.
   - Order Tracking: Provide users with a way to track their order's delivery status.
5. **Payment**
   - Multiple Payment Options: Support for credit/debit cards, online banking, and other popular payment methods.
   - Secure Transactions: Ensure user trust by facilitating secure payment transactions.
   - Payment Receipt: Provide users with a receipt after a successful payment.
6. **Authentication**
   - Secure Authentication: Ensure that user data remains private and secure during login and throughout their session.
   - Session Management: Users should remain logged in for a specified duration or until they decide to log out.

Use Case: Typical Flow with Kafka & Elasticsearch Integration

Part 1

- User logs in and searches for a product.
- Request reaches LB, then passed to API Gateway.
- API Gateway routes the search request to Product Catalog Service.
- Product Catalog Service queries Elasticsearch for a fast product search.

Part 2

- User adds a product to the cart.
- When user checks out; create order is invoked in Order Management Service.

Part 3

- User checks out, triggering the Order Management Service.
- After placing the order, a message is sent to Kafka.
- Payment Service consumes the Kafka message to process payment.
- Order confirmation is sent back to User after successful payment.

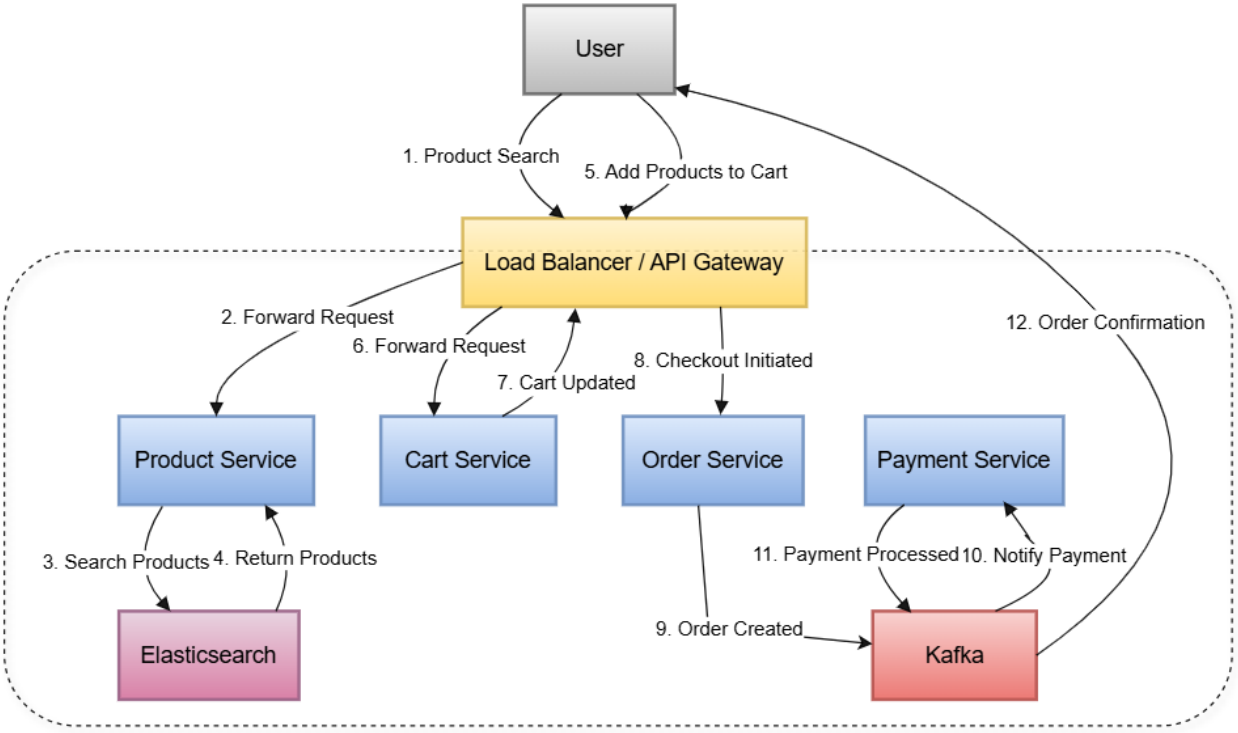

Figure 2.1: Flow Diagram

Class Diagrams

1. **User Management Service**
   - **Models**

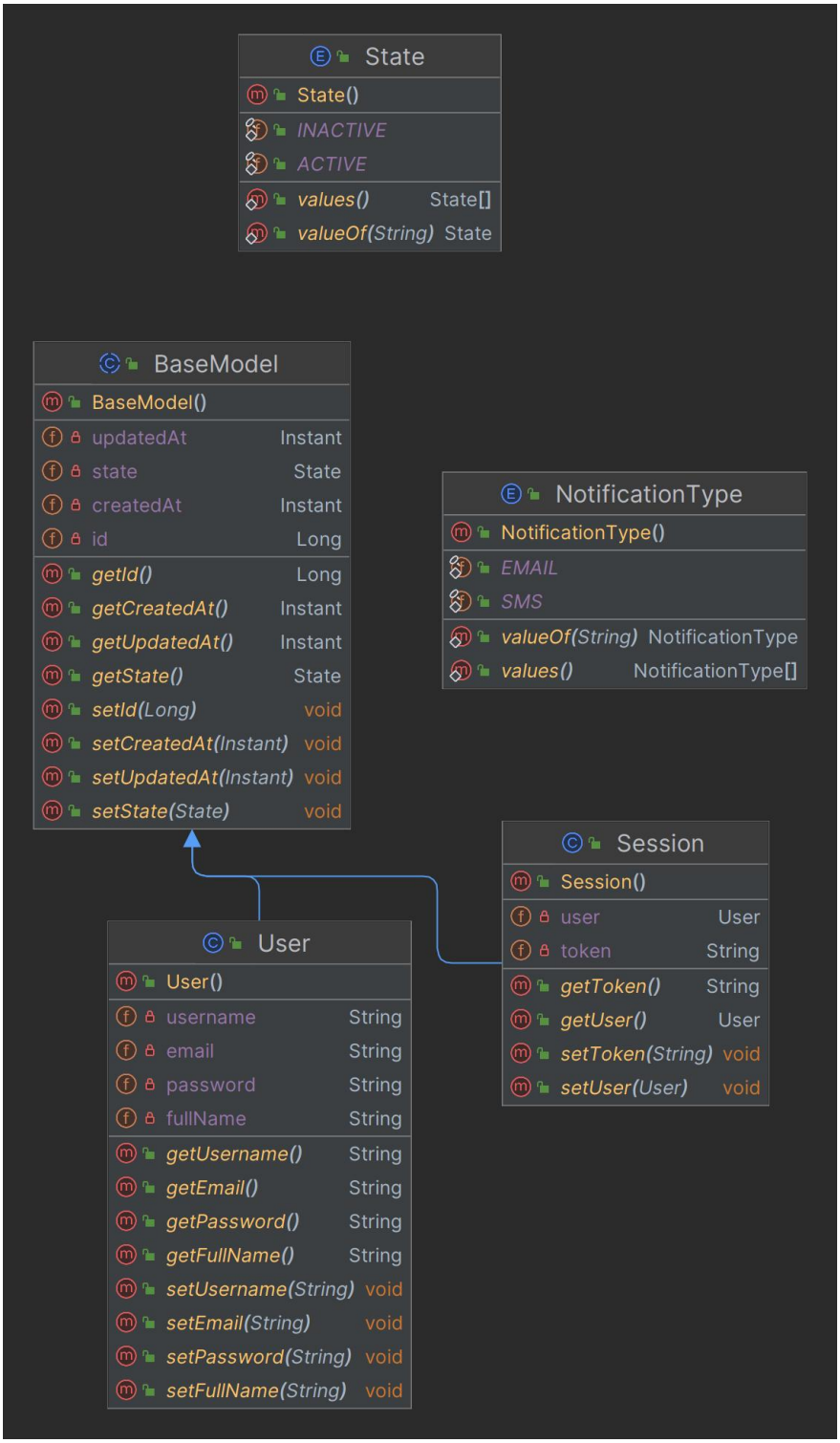

Figure 3.1.1: User Management Models

- **Repositories**

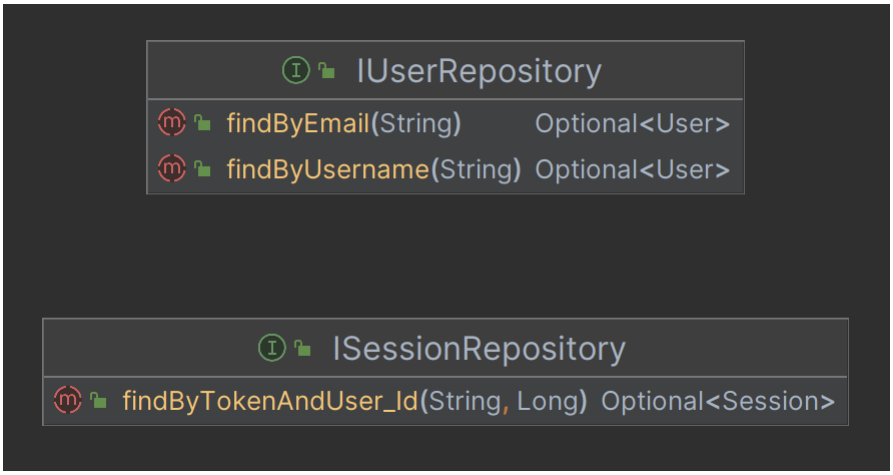

Figure 3.1.2: User Management Repository

- **Controllers**

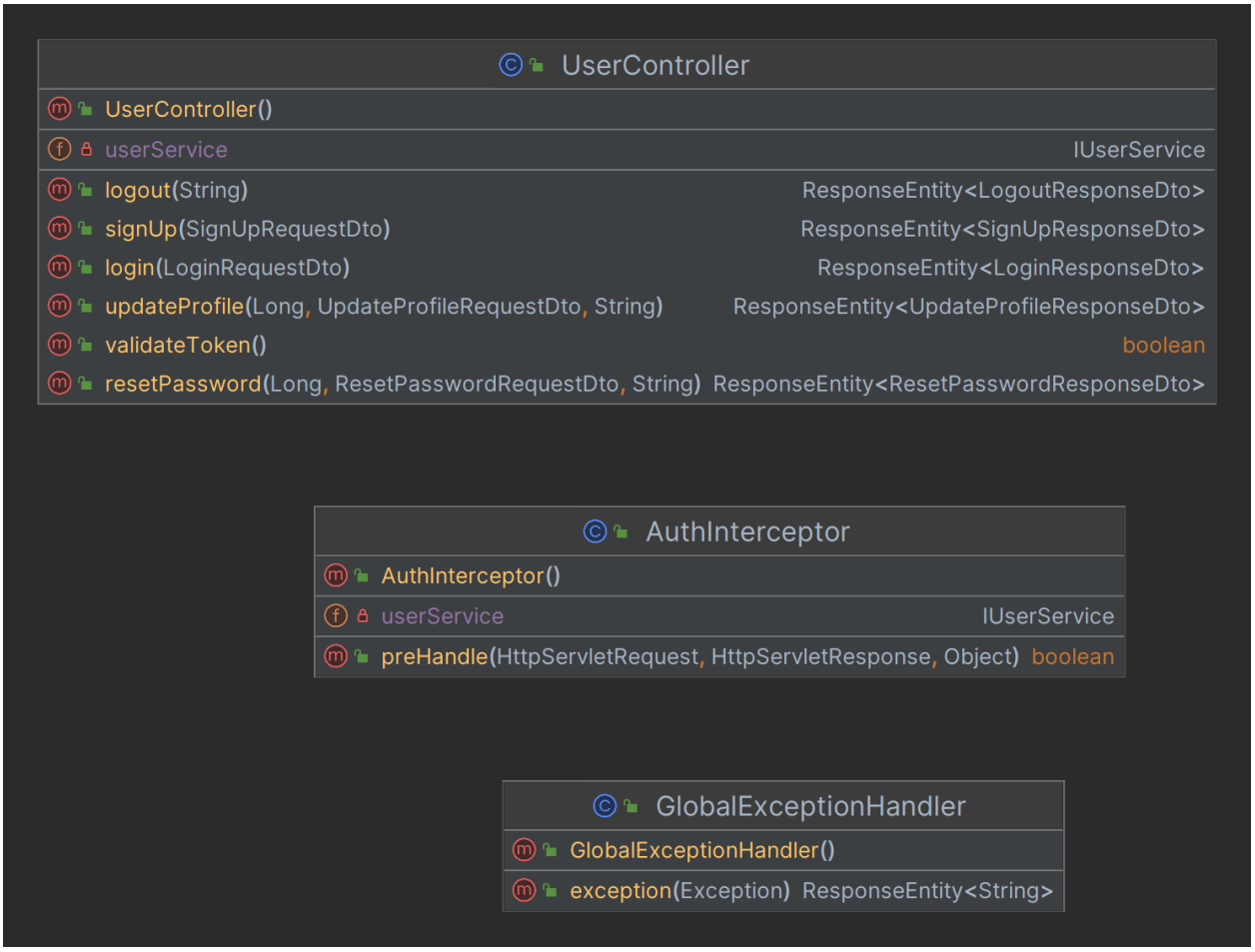

Figure 3.1.3: User Management Controller

- **Services**

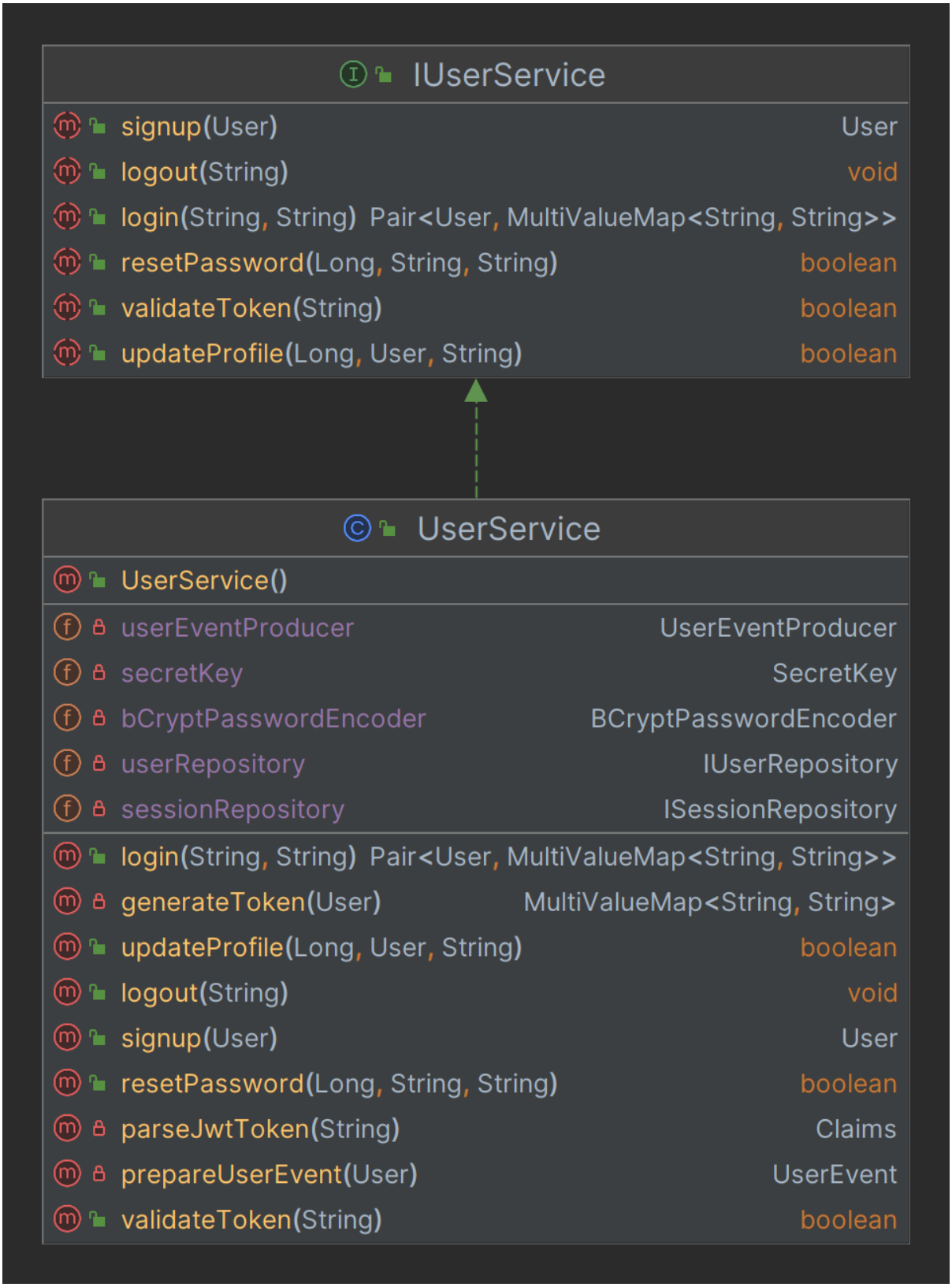

Figure 3.1.4: User Management Service

- **Configs**

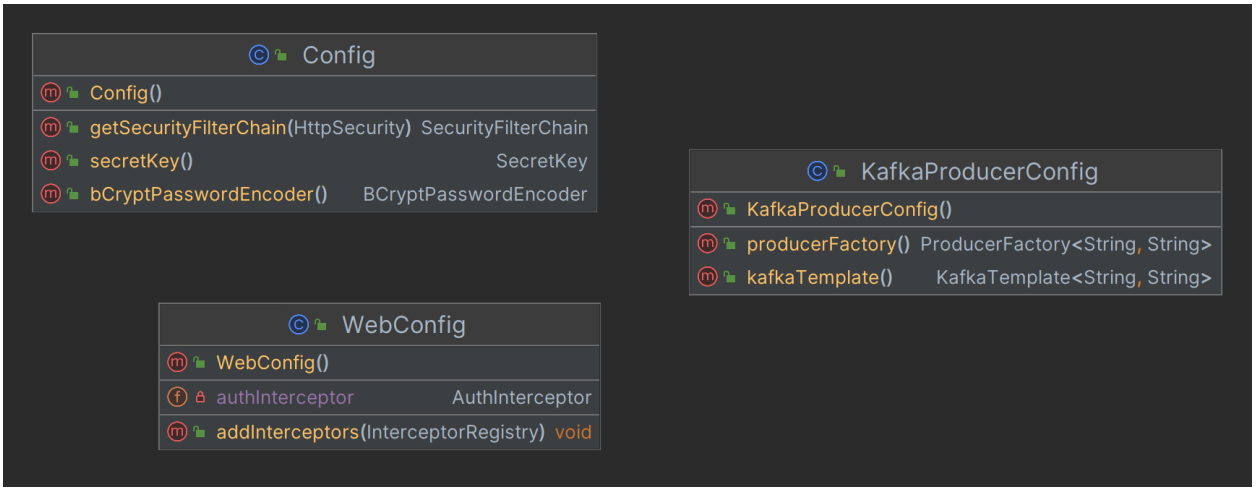

Figure 3.1.5: User Management Configurations

- **Exceptions**

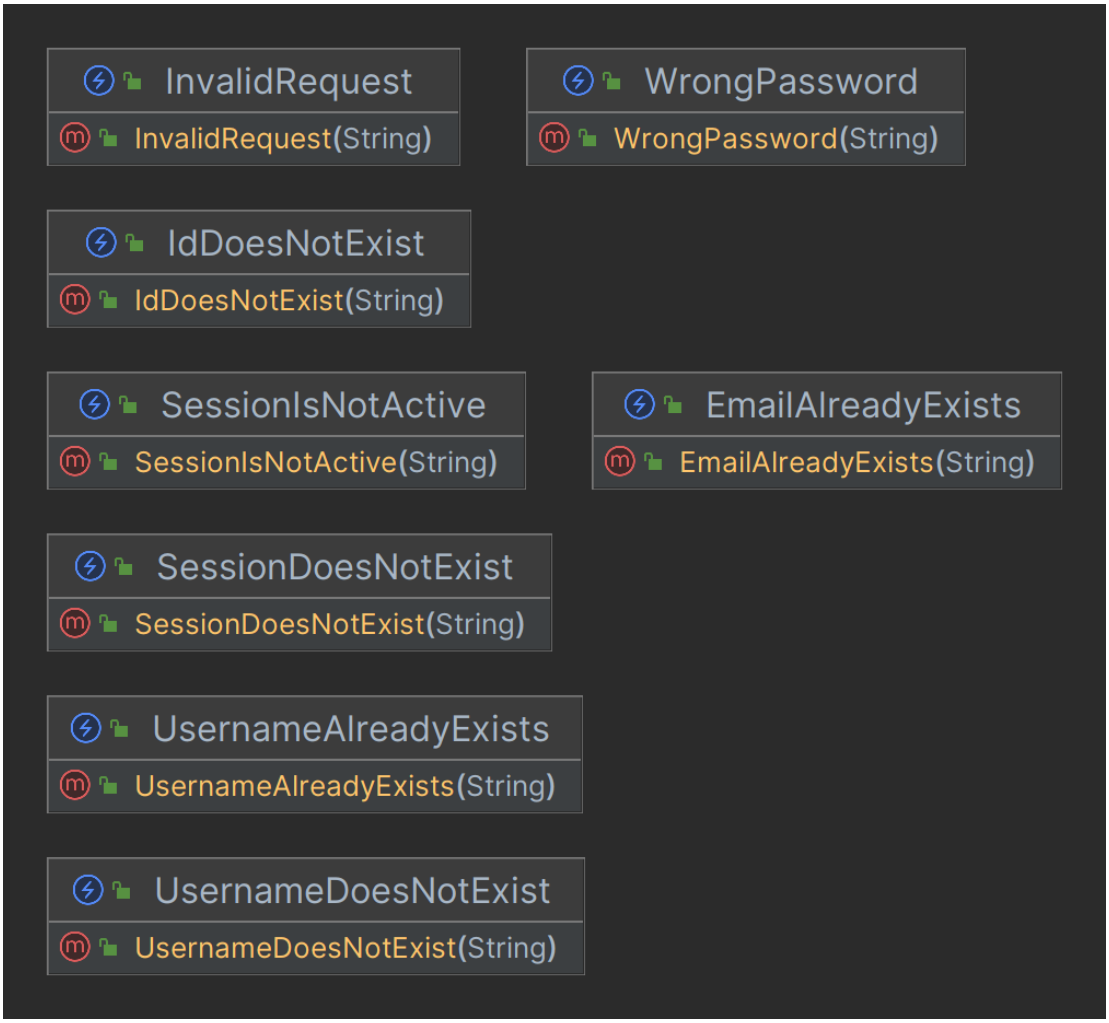

Figure 3.1.6: User Management Exceptions

- **Dtos**

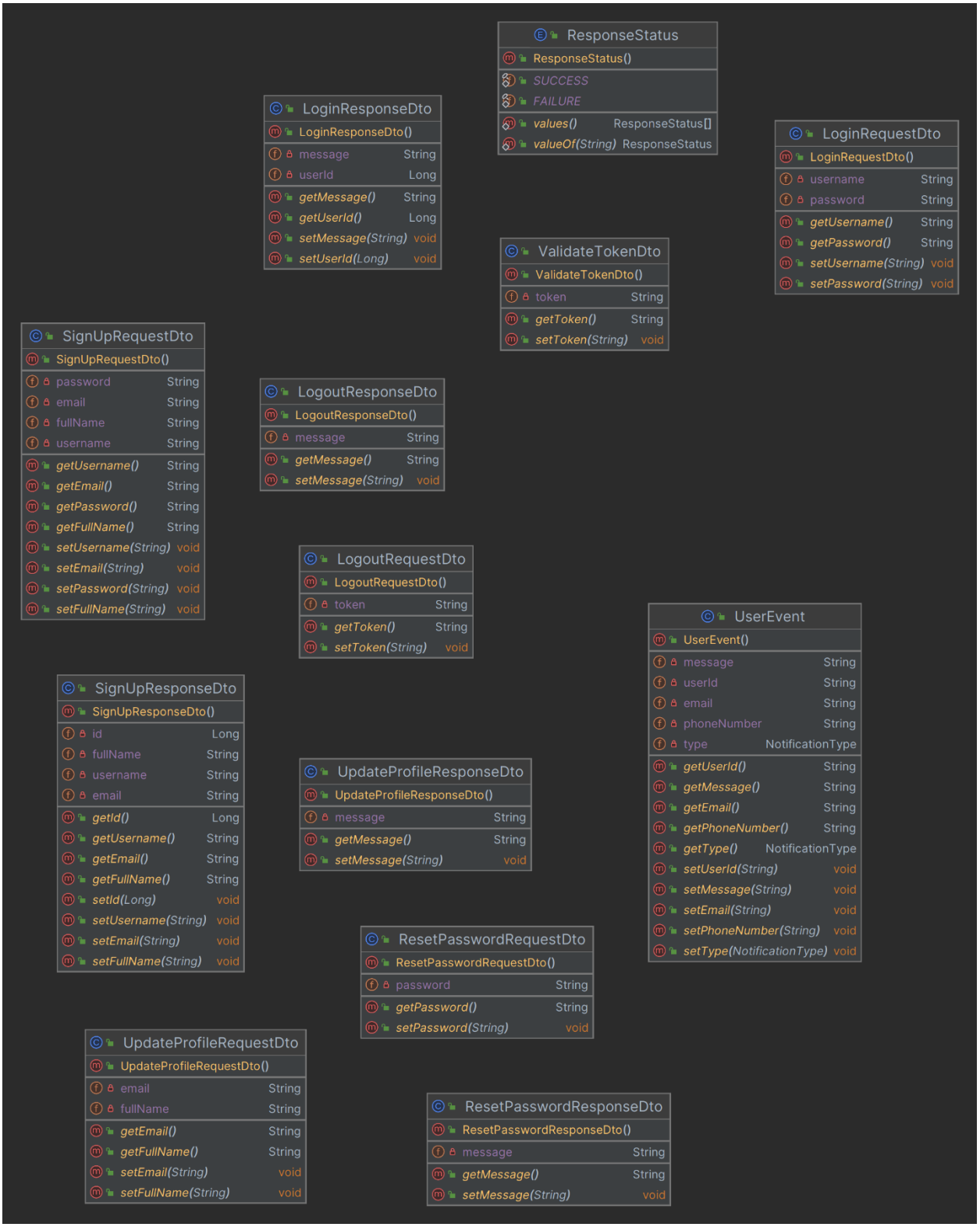

Figure 3.1.7: User Management Dtos

- **Producers**

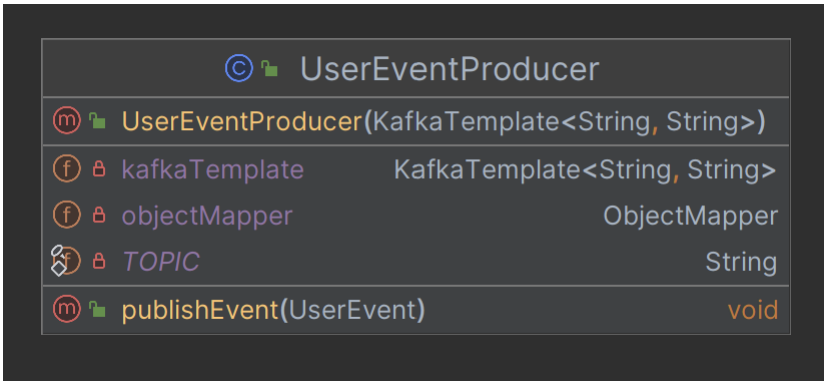

Figure 3.1.8: User Management Kafka Event Producer

2. **Product Catalog Service**
   - **Models**

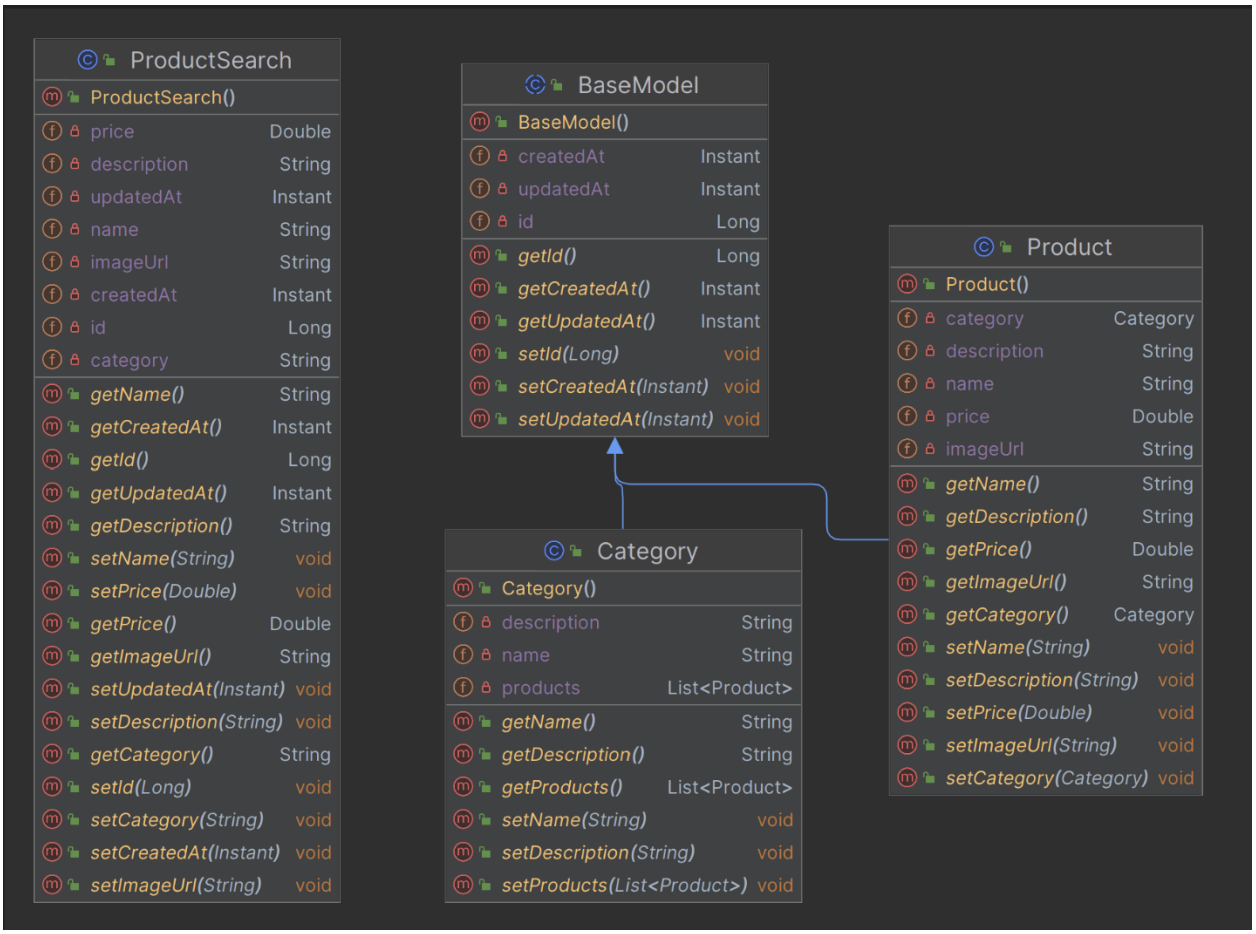

Figure 3.2.1: Product Catalog Models

- **Repositories**

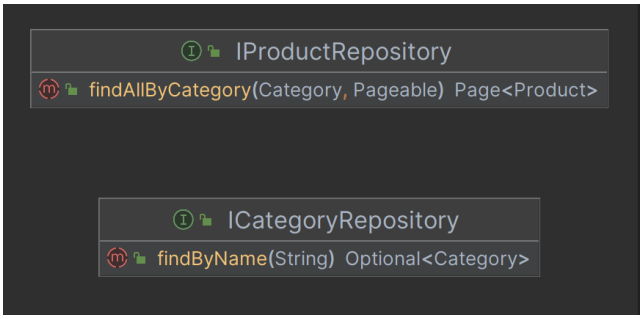

Figure 3.2.2: Product Catalog JPA Repository

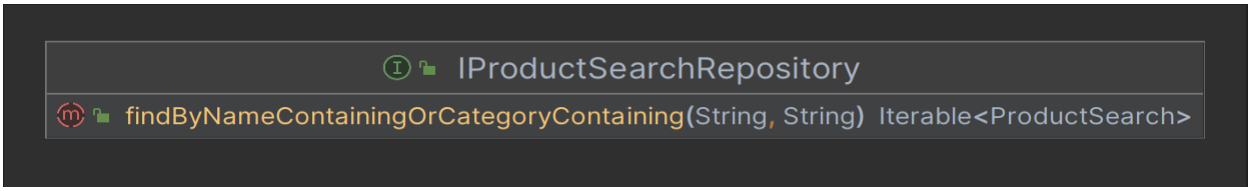

Figure 3.2.3: Product Catalog ElasticSearch Repository

- **Controllers**

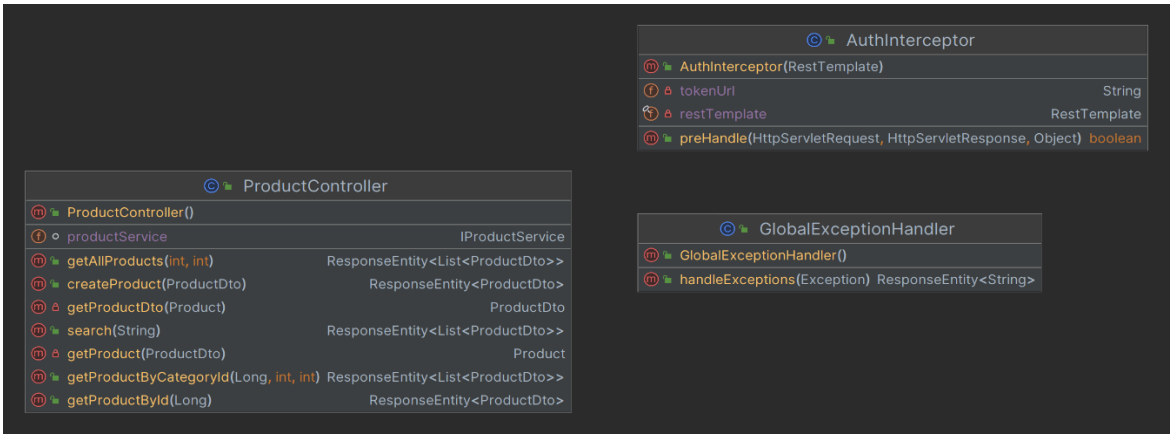

Figure 3.2.4: Product Catalog Controller

- **Exceptions**

Figure 3.2.5: Product Catalog Exceptions

- **Services**

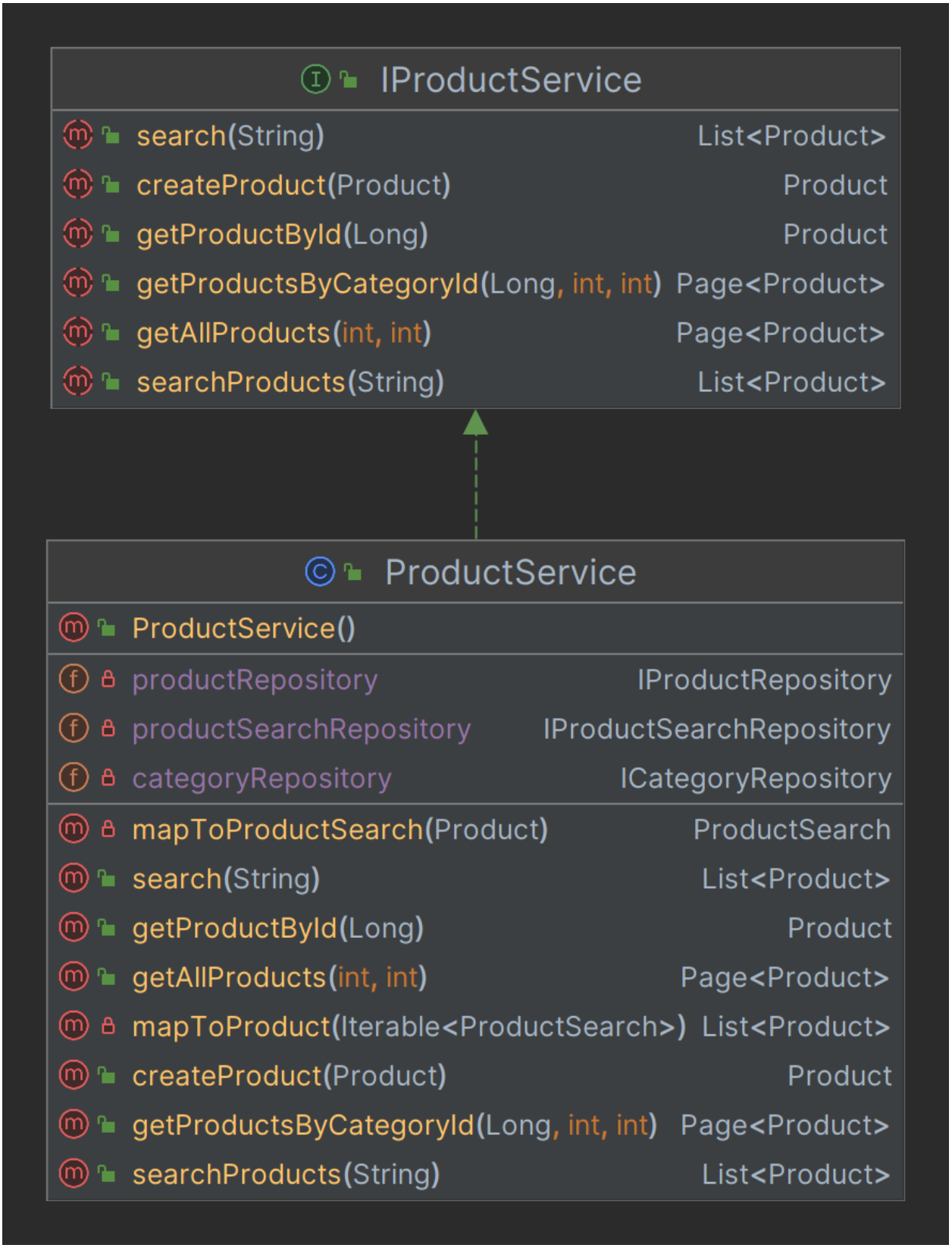

Figure 3.2.6: Product Catalog Service

- **Dtos**

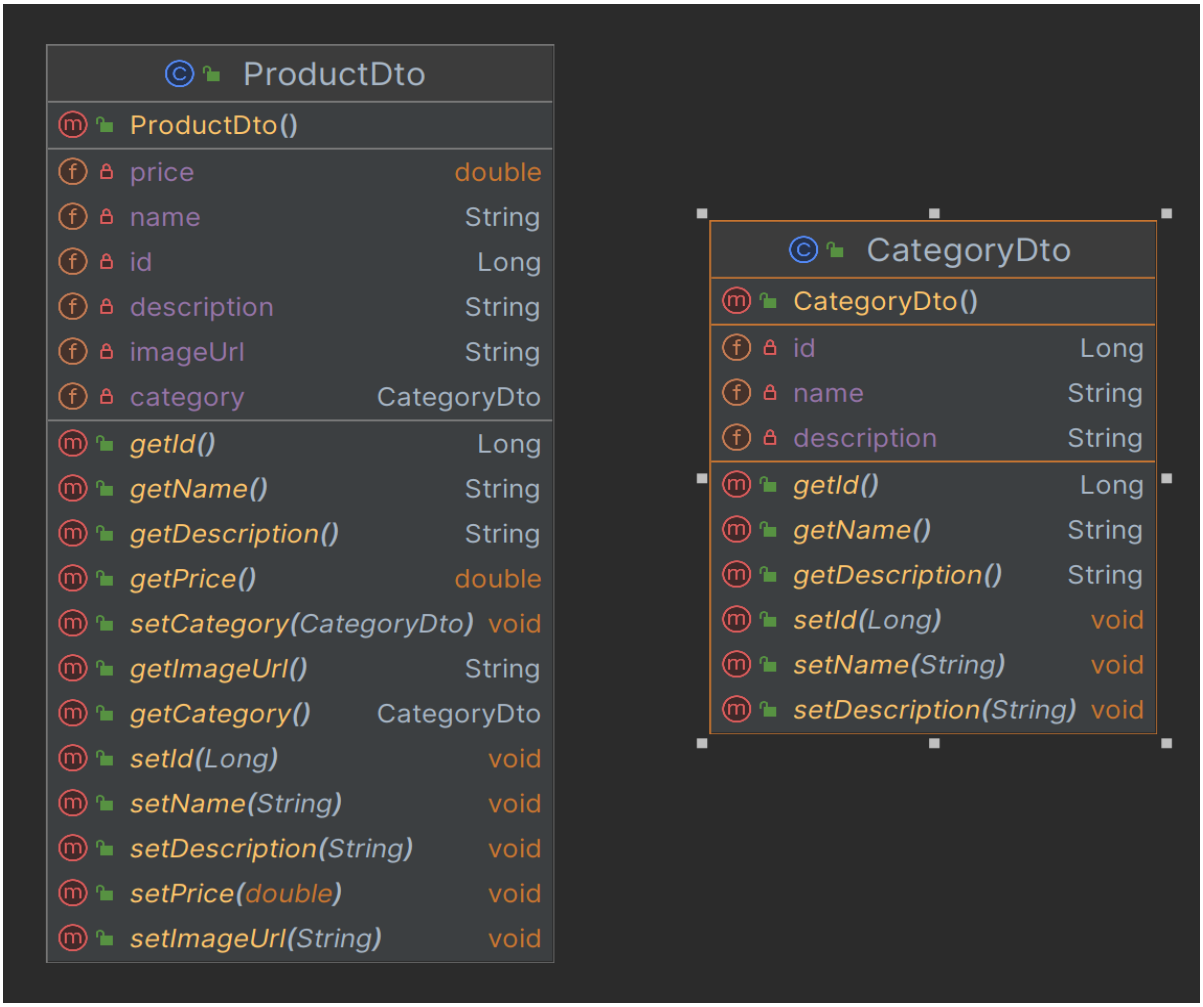

Figure 3.2.7: Product Catalog Dtos

- **Configs**

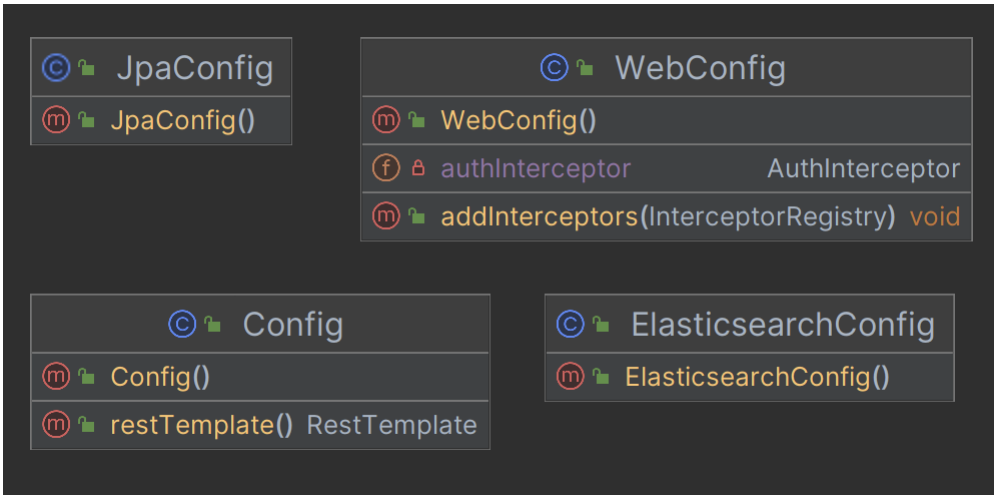

Figure 3.2.8: Product Catalog Configurations

3. **Cart Service**
   - **Models**

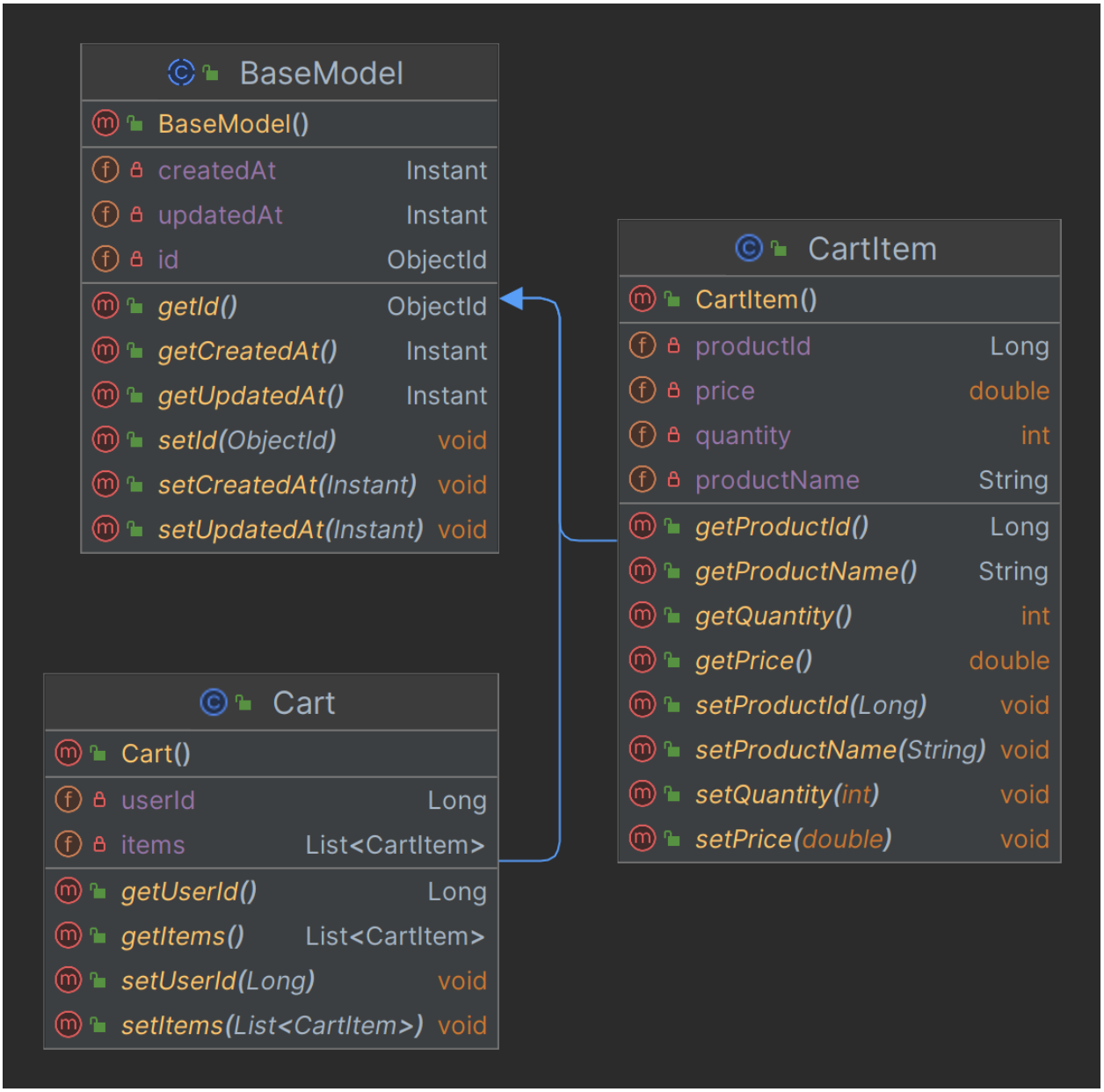

Figure 3.3.1: Cart Models

   - **Repositories**

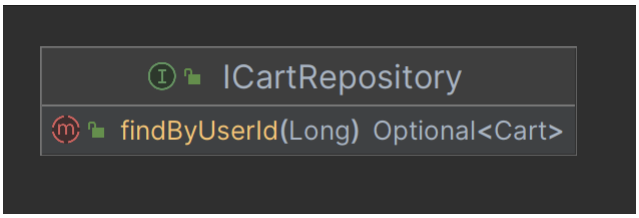

Figure 3.3.2: Cart Repository

- **Services**

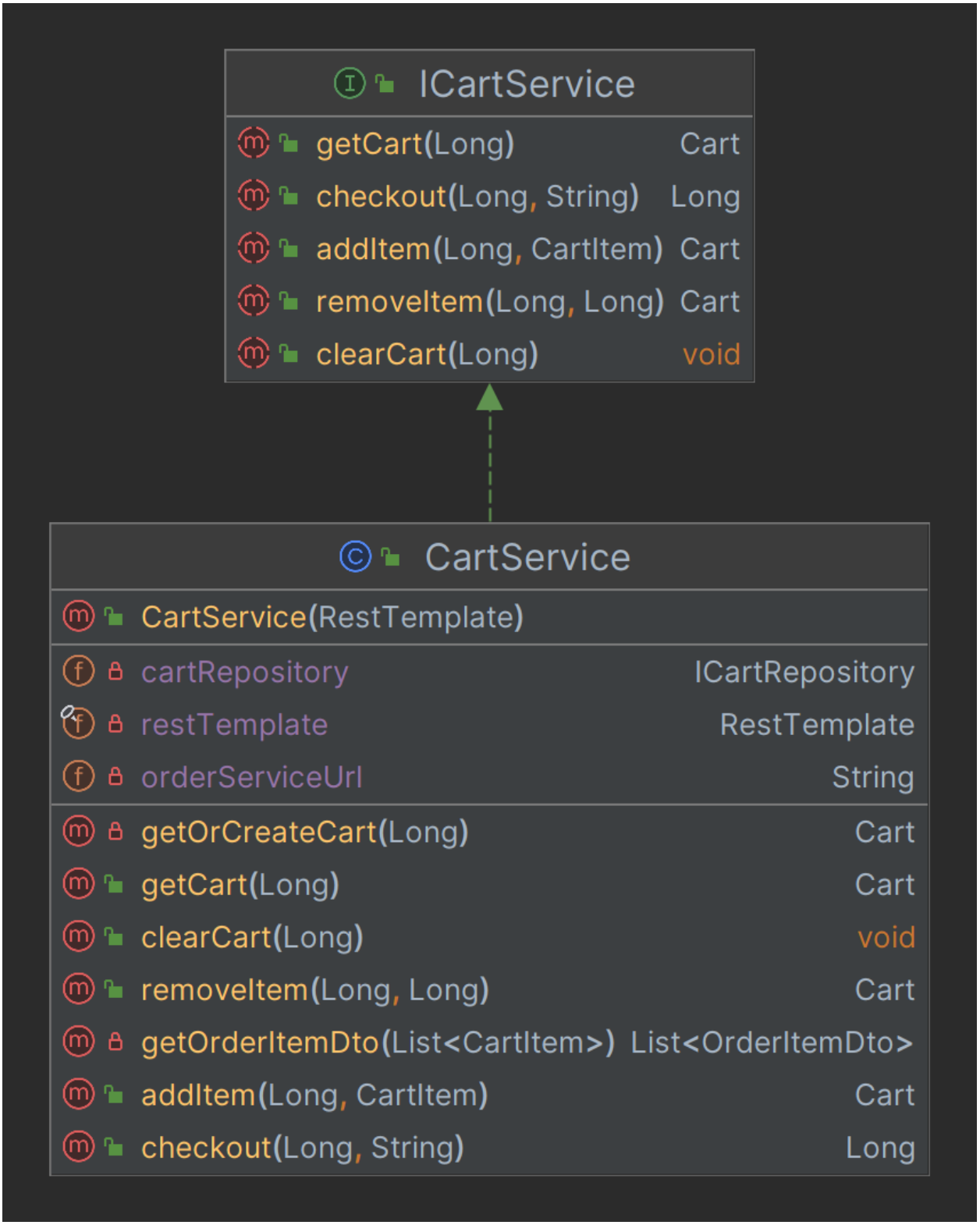

Figure 3.3.3: Cart Service

- **Dtos**

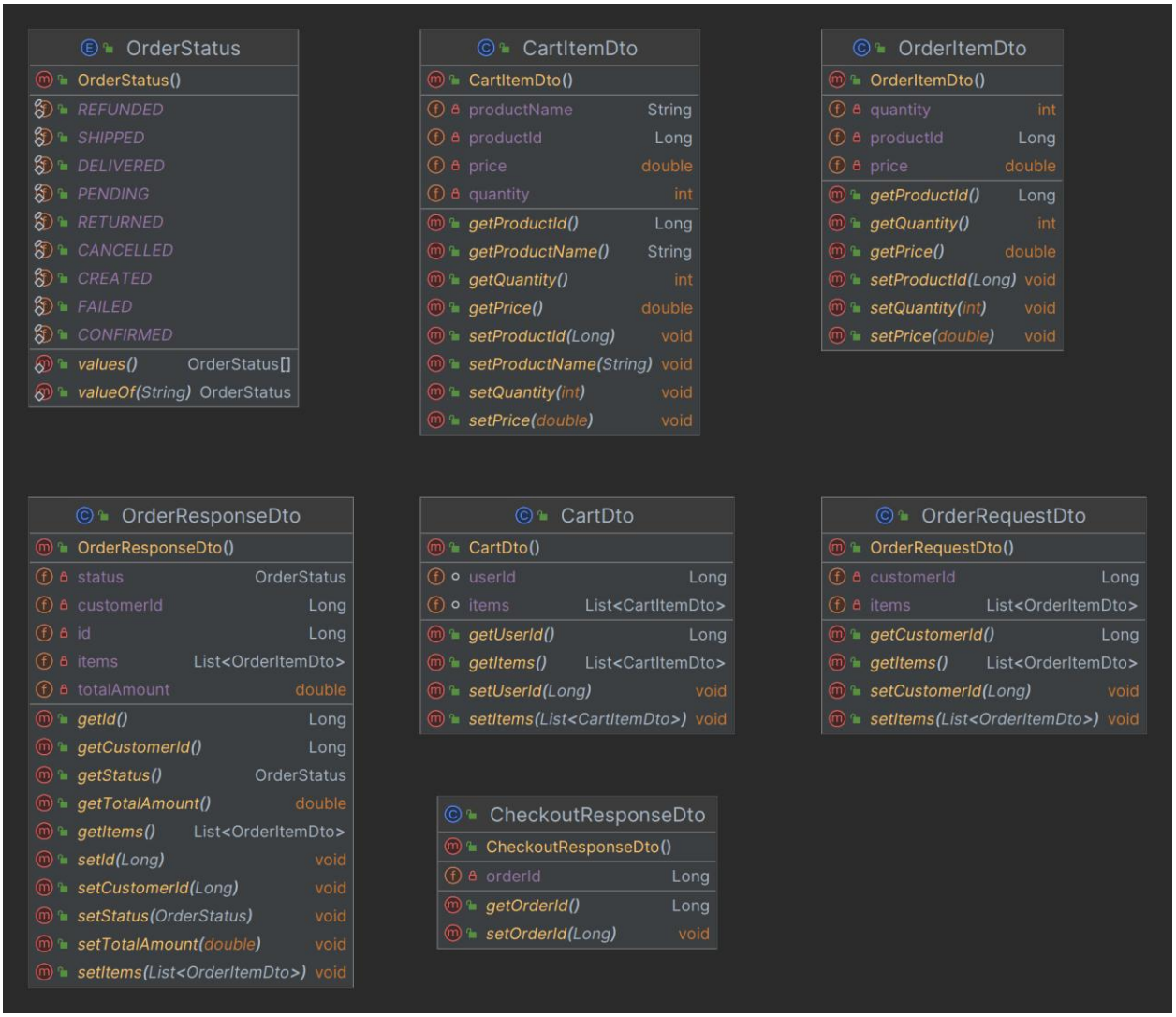

Figure 3.3.4: Cart Dtos

- **Controllers**

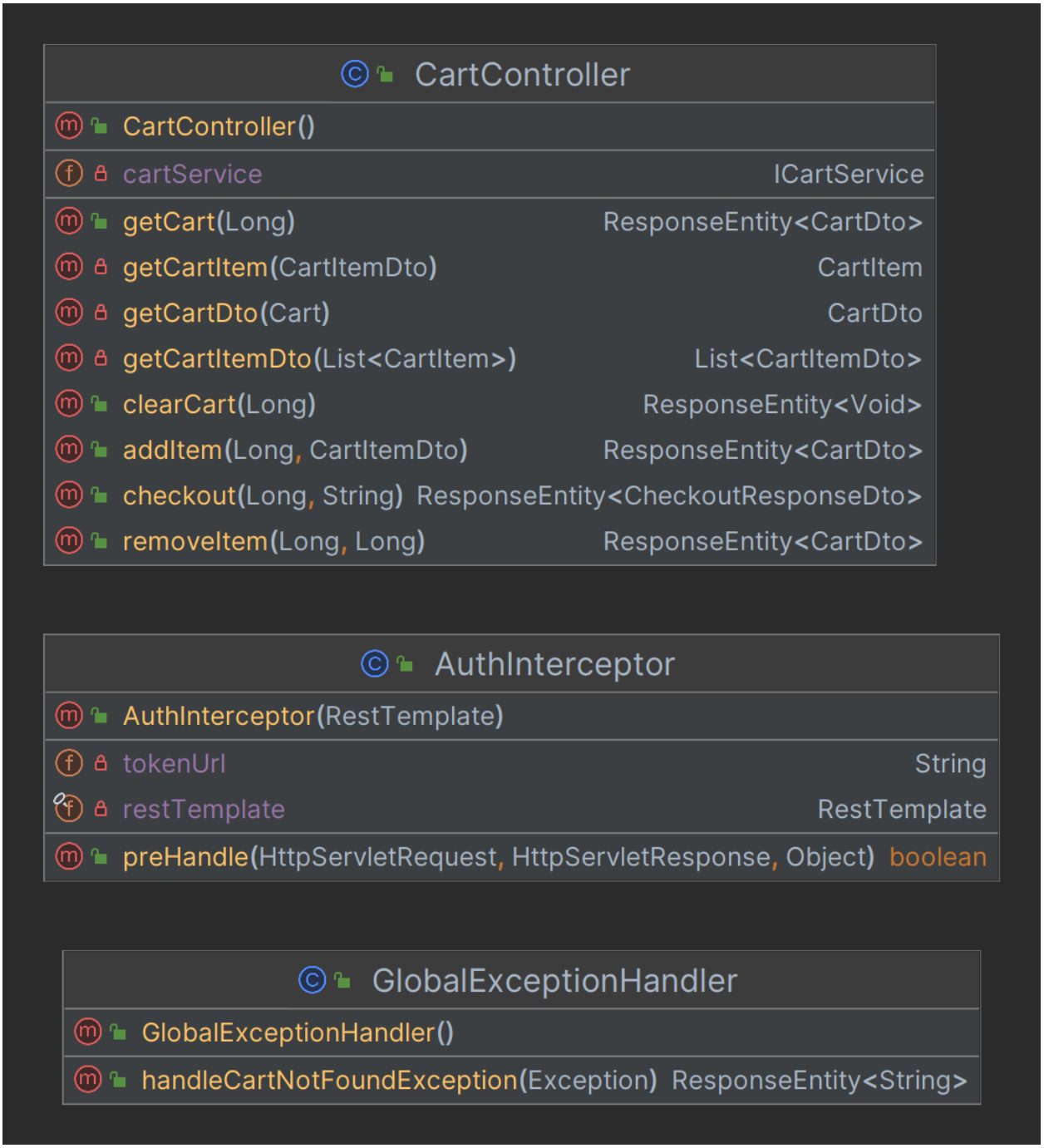

Figure 3.3.5: Cart Controller

- **Configs**

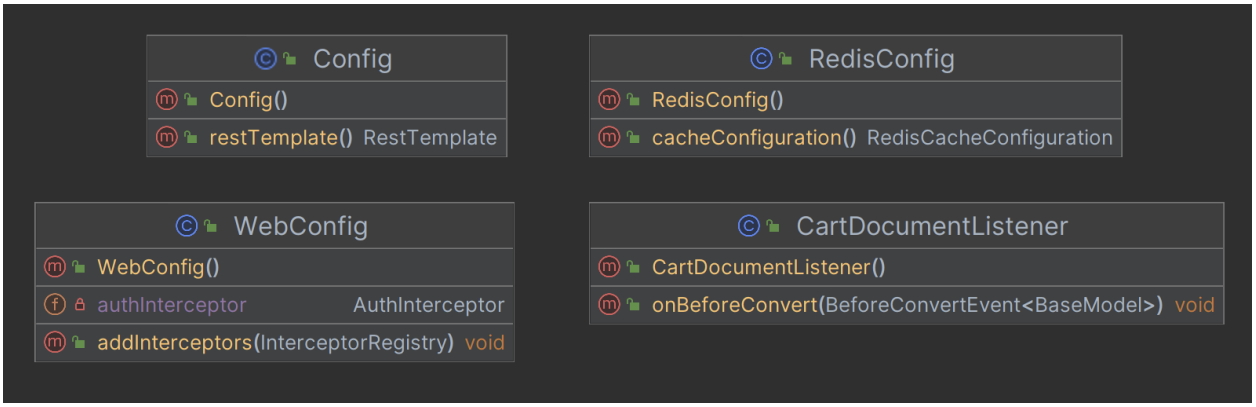

Figure 3.3.6: Cart Configuration

- **Exceptions**

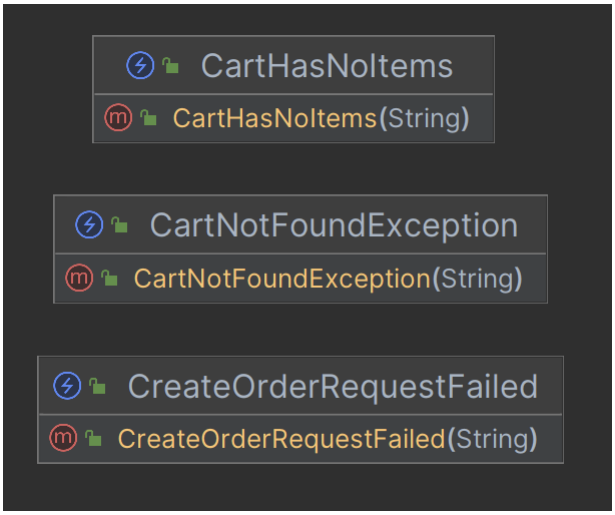

Figure 3.3.7: Cart Exceptions

4. **Order Management Service**
   - **Models**

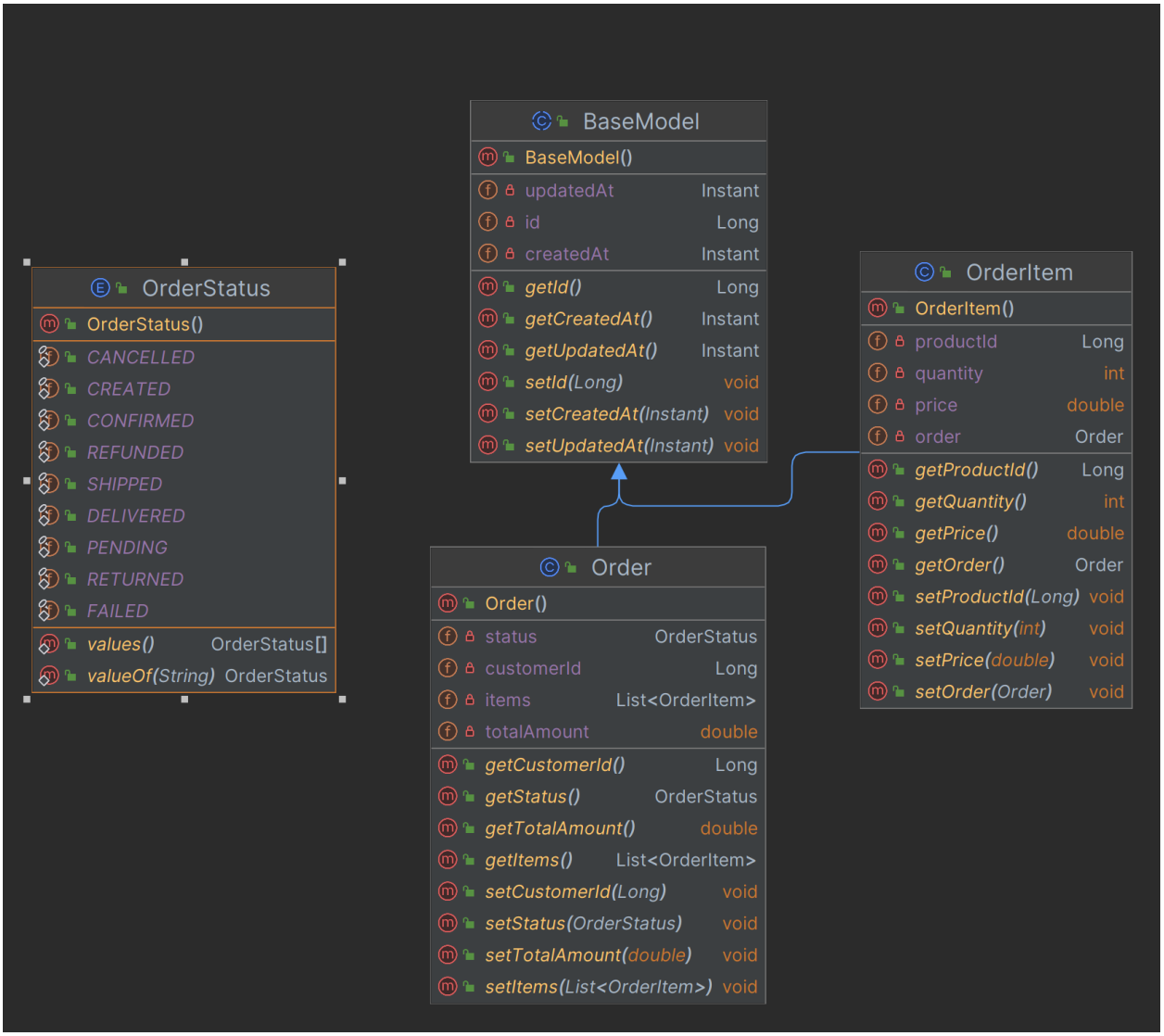

Figure 3.4.1: Order Management Models

   - **Repositories**

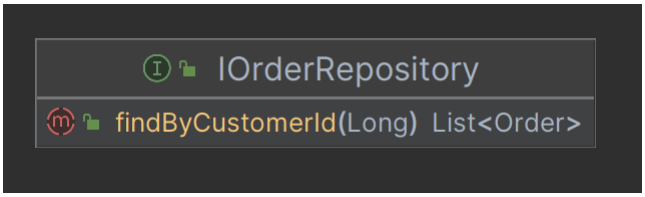

Figure 3.4.2: Order Management Repository

- **Services**

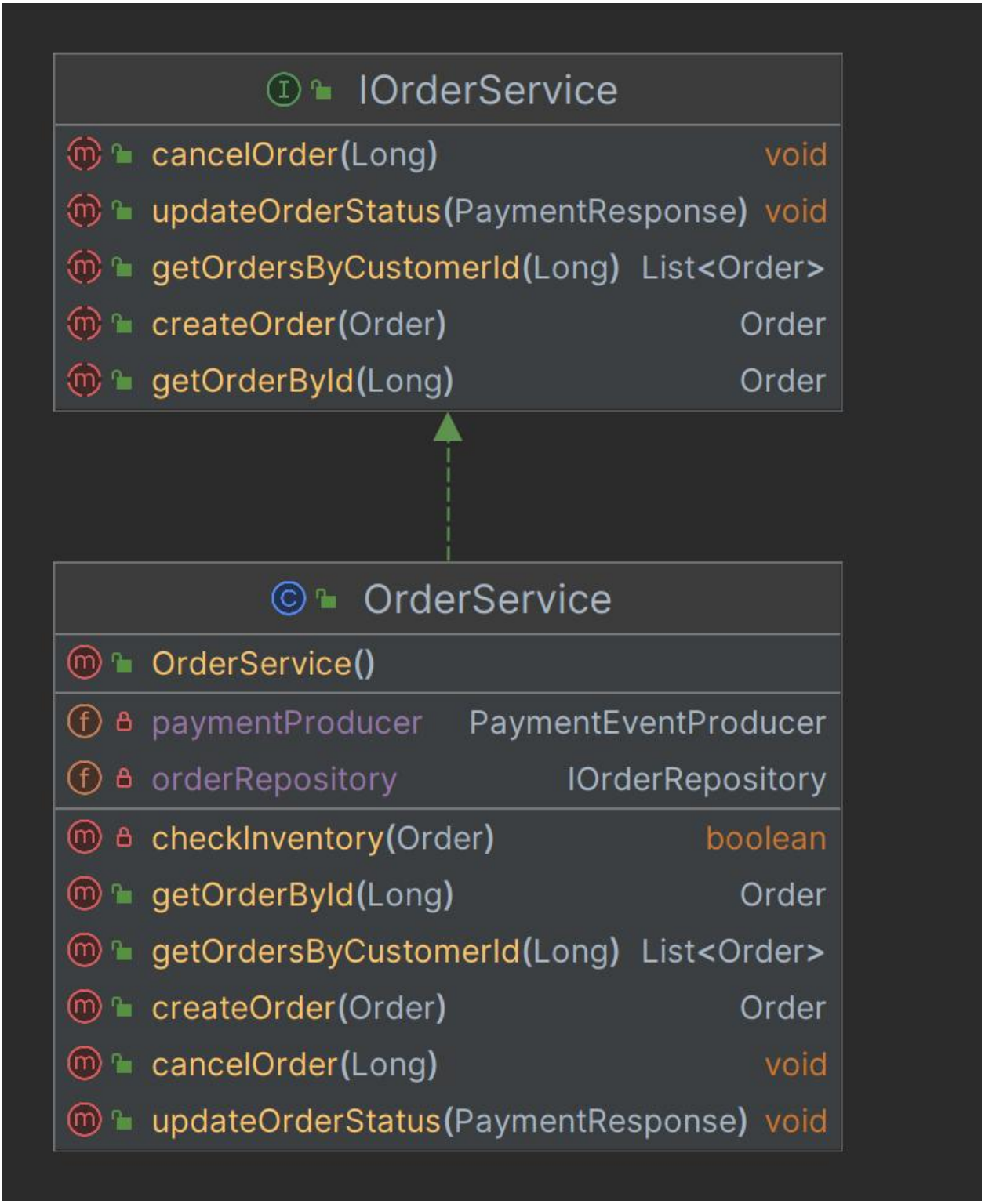

Figure 3.4.3: Order Management Service

- **Controllers**

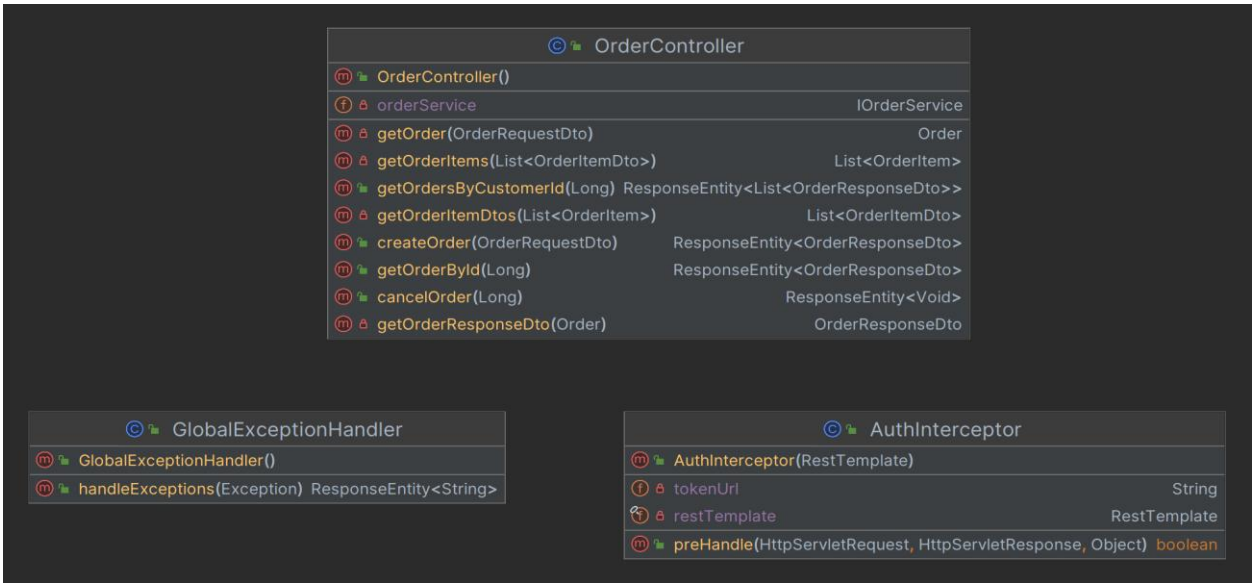

Figure 3.4.4: Order Management Controller

- **Dtos**

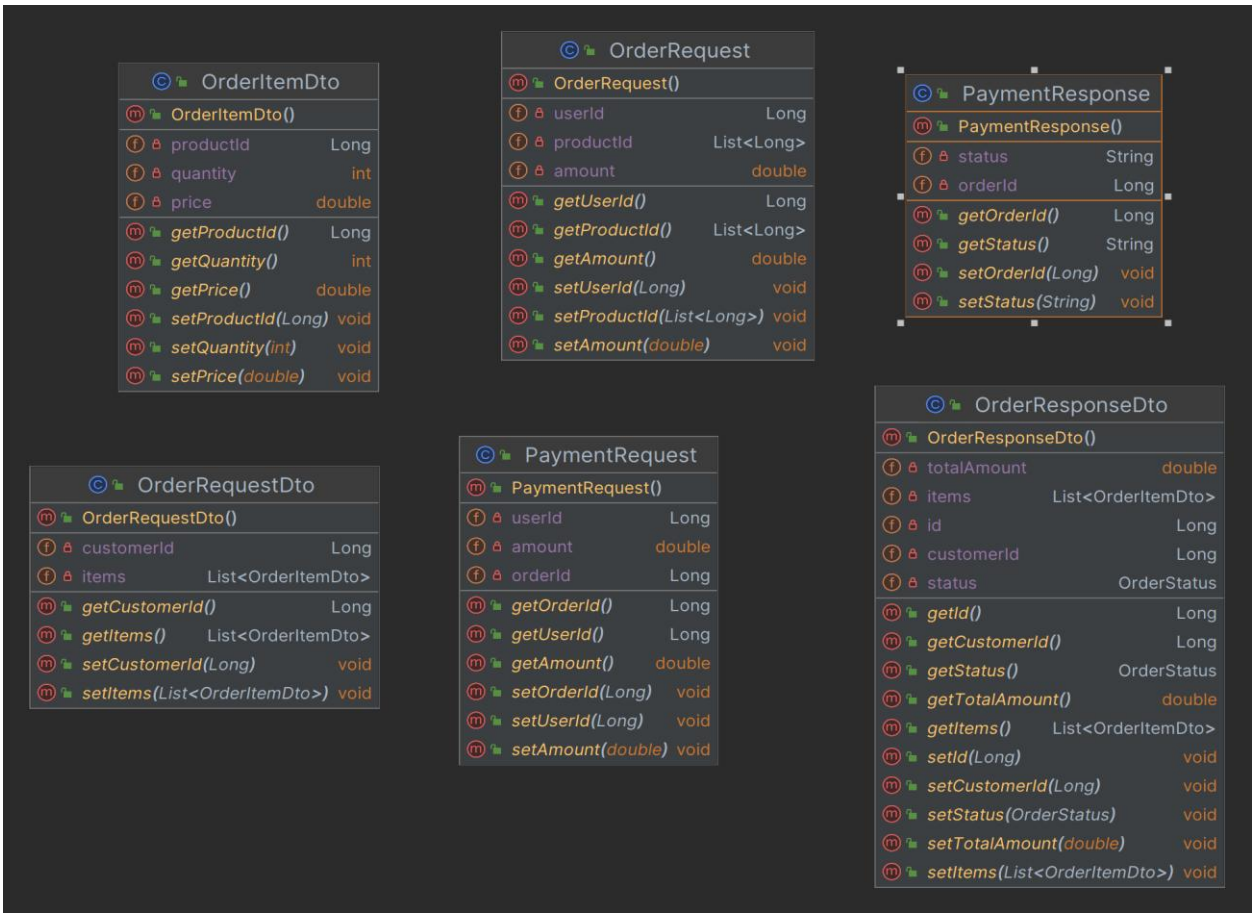

Figure 3.4.5: Order Management Dtos

- **Configs**

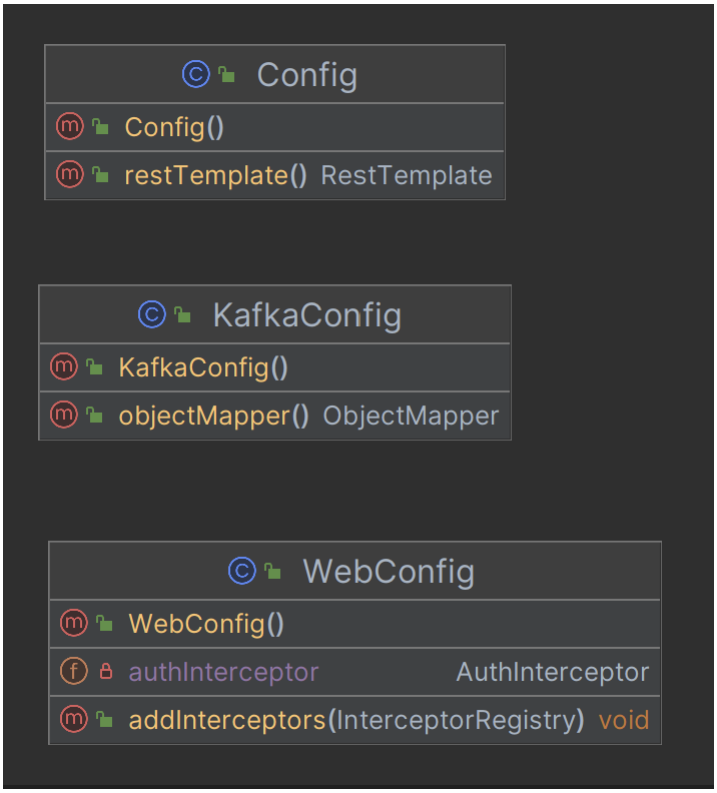

Figure 3.4.6: Order Management Configurations

- **Exceptions**

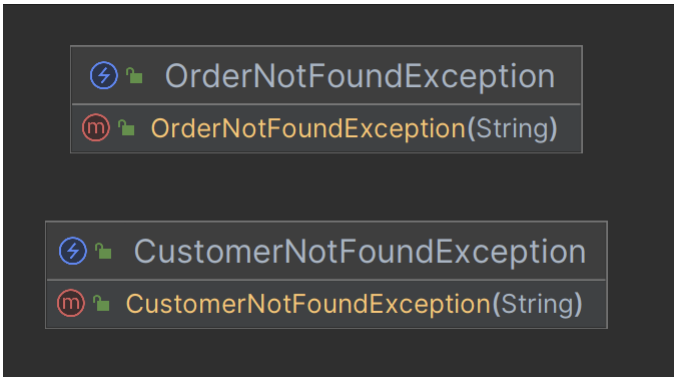

Figure 3.4.7: Order Management Exceptions

- **Producers**

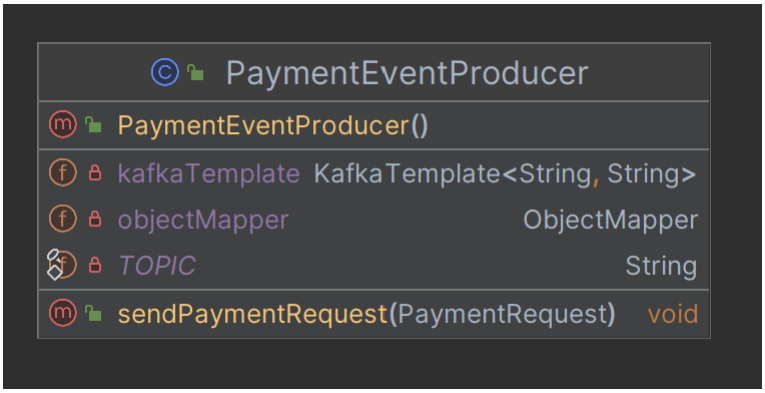

Figure 3.4.8: Order Management Kafka Producer

- **Consumers**

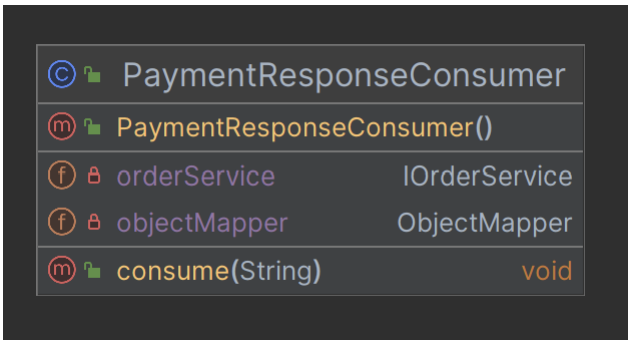

Figure 3.4.9: Order Management Kafka Consumer

5. **Payment Service**
   - **Models**

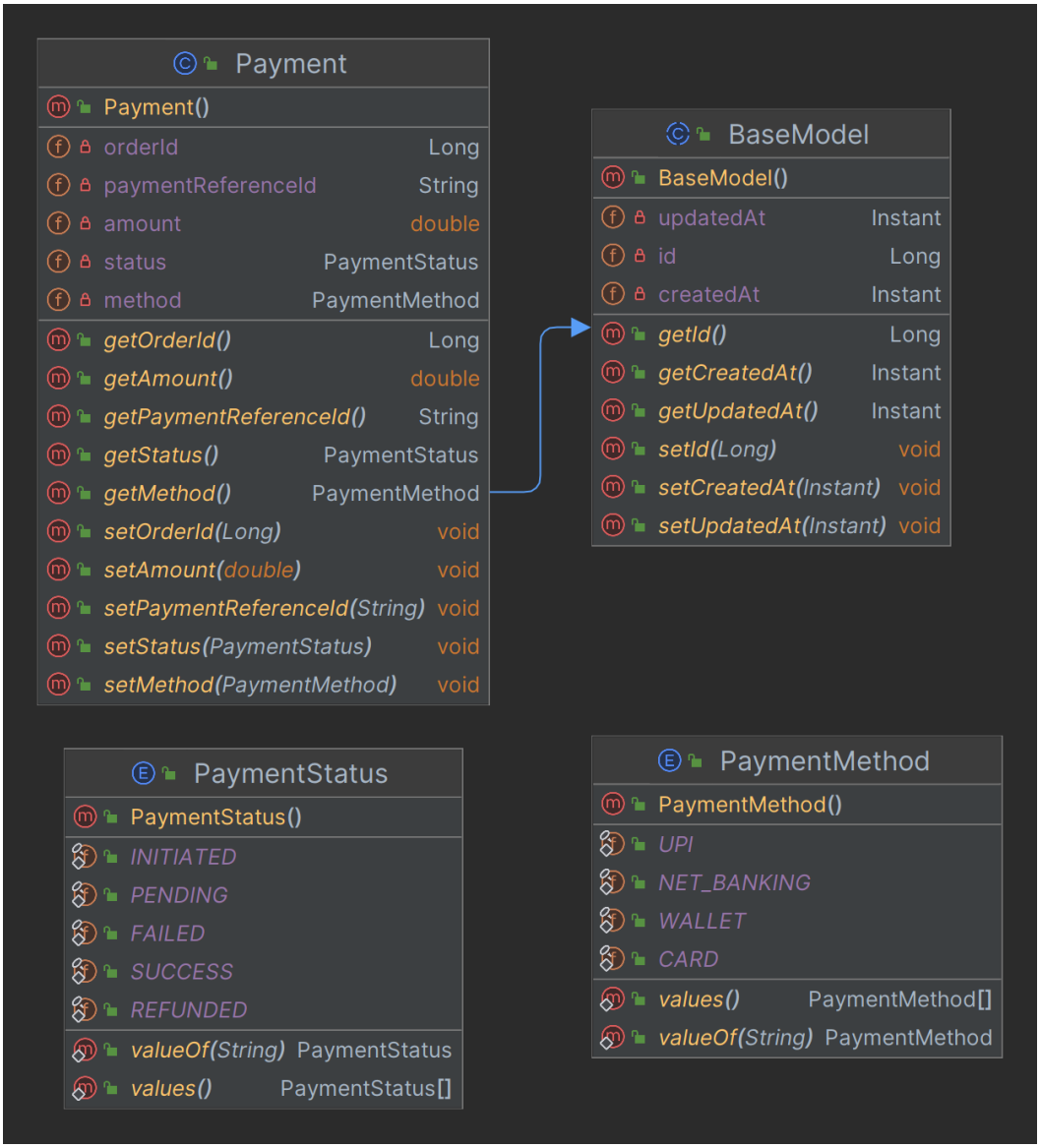

Figure 3.5.1: Payment Models

   - **Repositories**

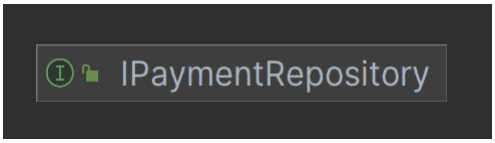

Figure 3.5.2: Payment Repository

- **Services**

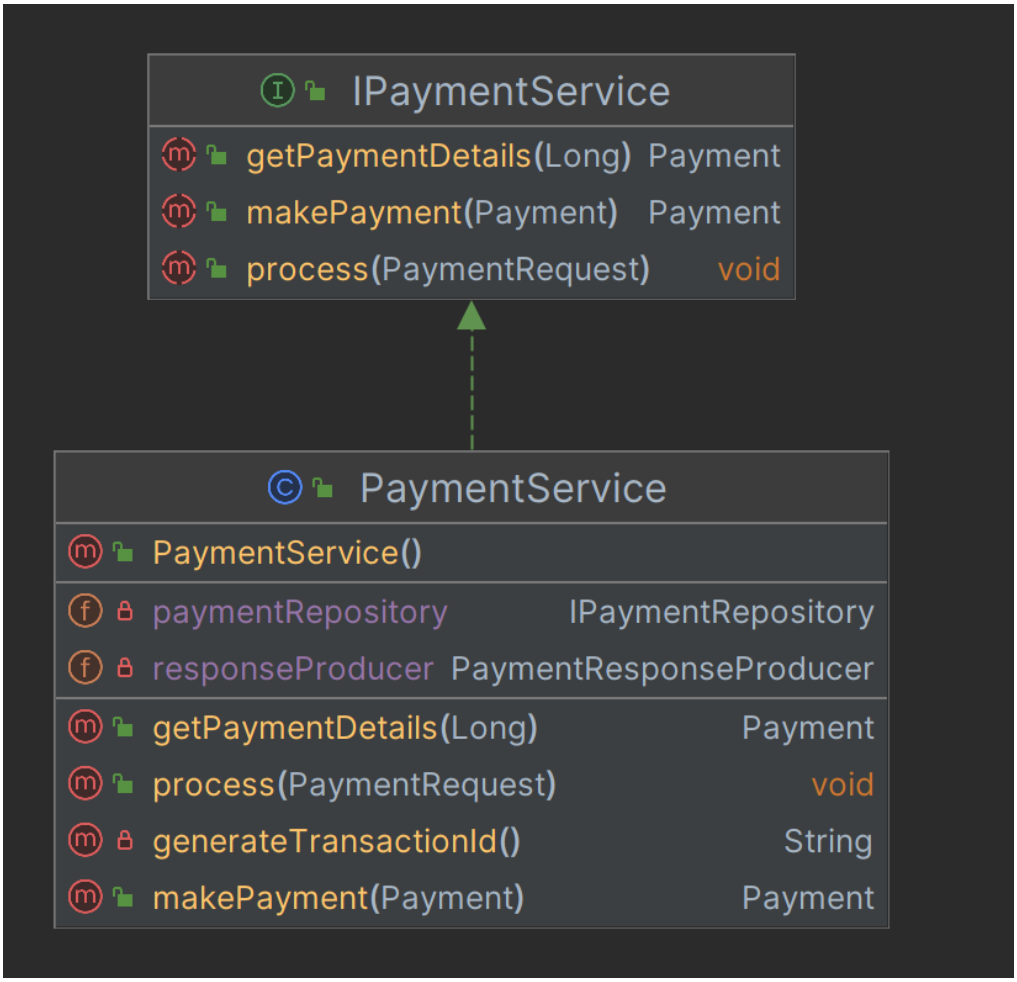

Figure 3.5.3: Payment Service

- **Controllers**

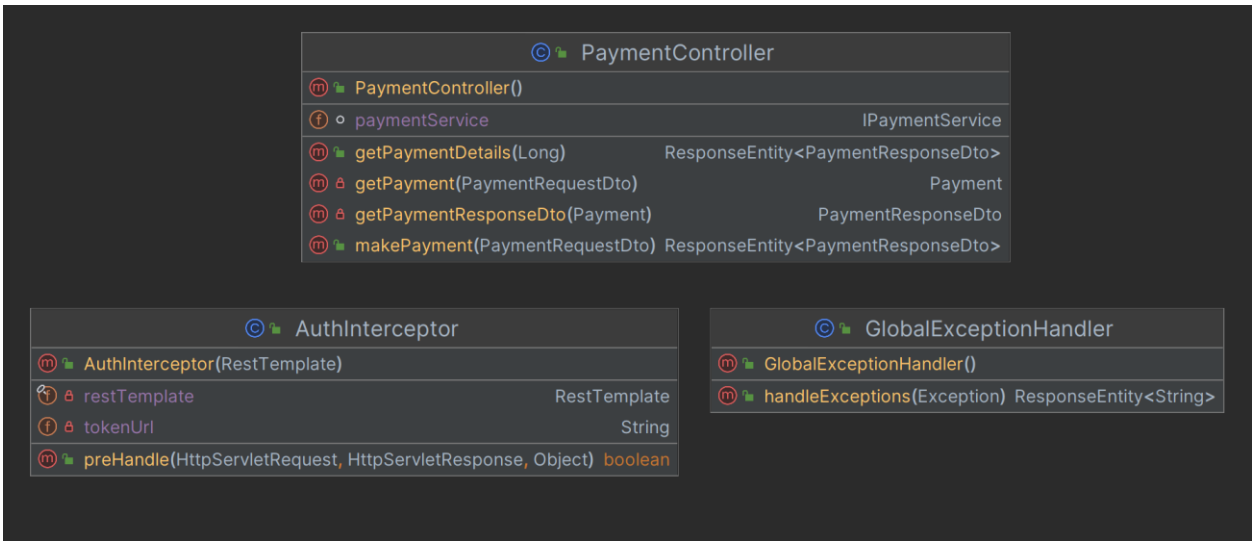

Figure 3.5.4: Payment Controller

- **Dtos**

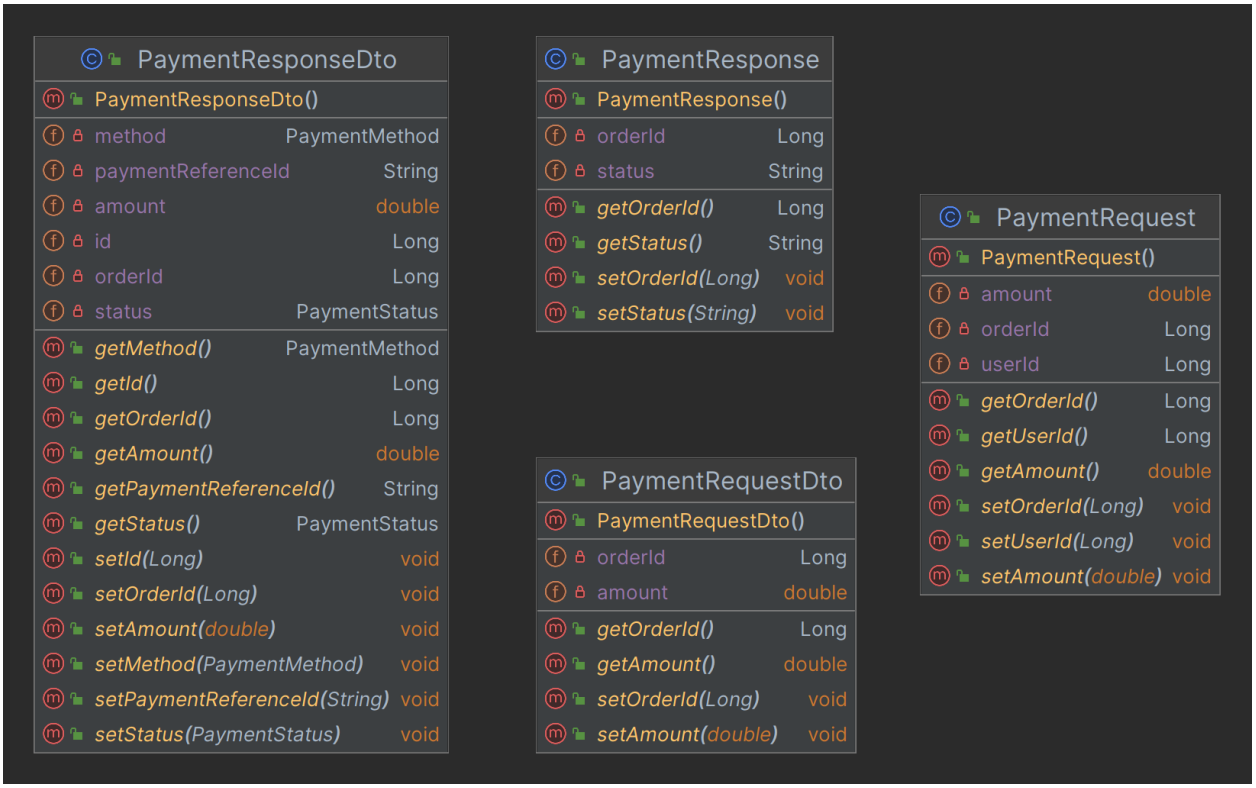

Figure 3.5.5: Payment Dtos

- **Configs**

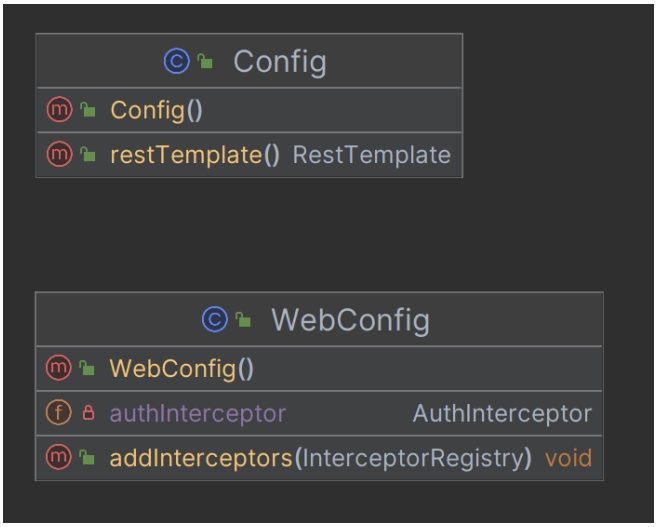

Figure 3.5.6: Payment Configurations

- **Exceptions**

Figure 3.5.7: Payment Exceptions

- **Producers**

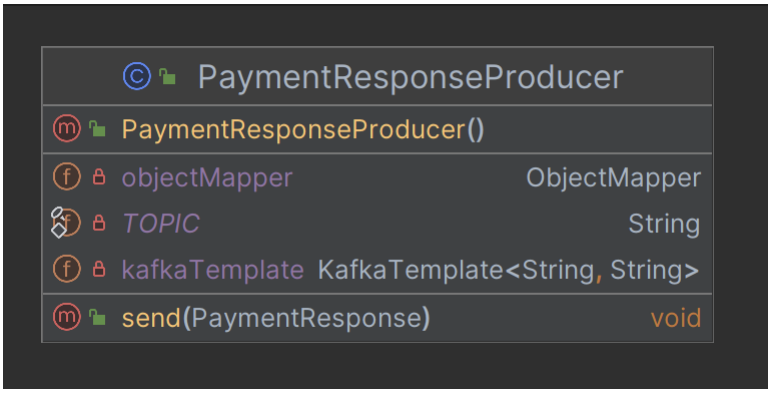

Figure 3.5.8: Payment Kafka Producer

- **Consumers**

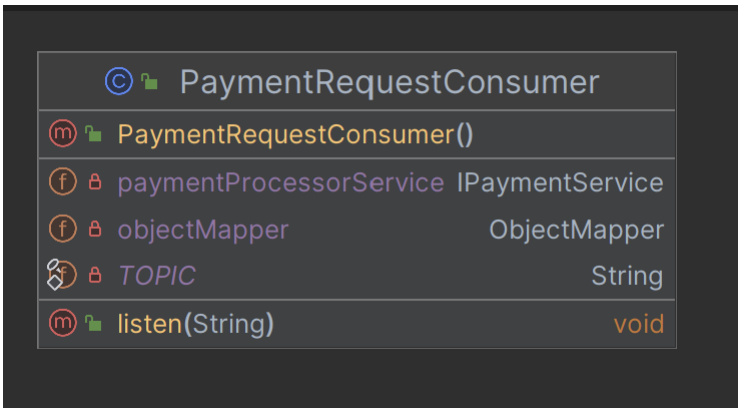

Figure 3.5.9: Payment Kafka Consumer

6. **Notification Service**
   - **Models**

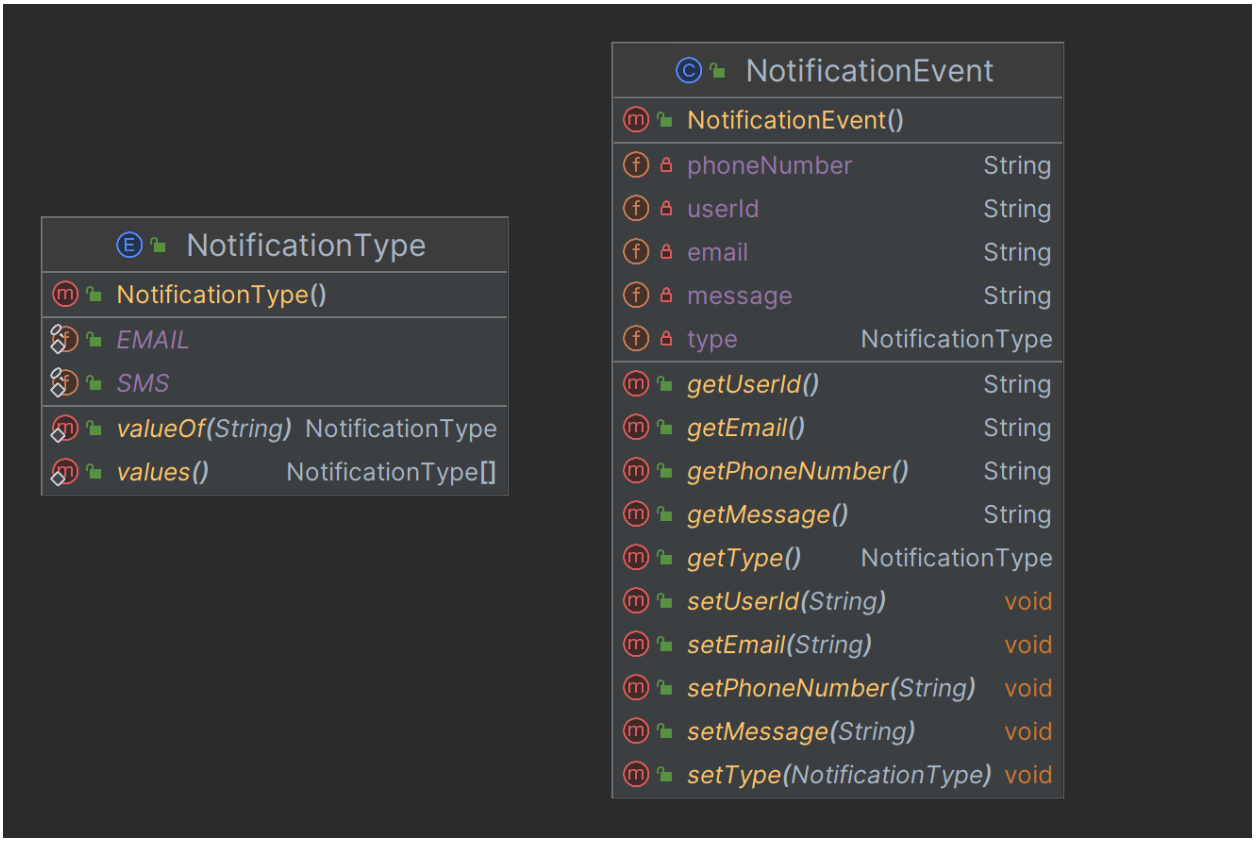

Figure 3.6.1: Notification Models

   - **Services**

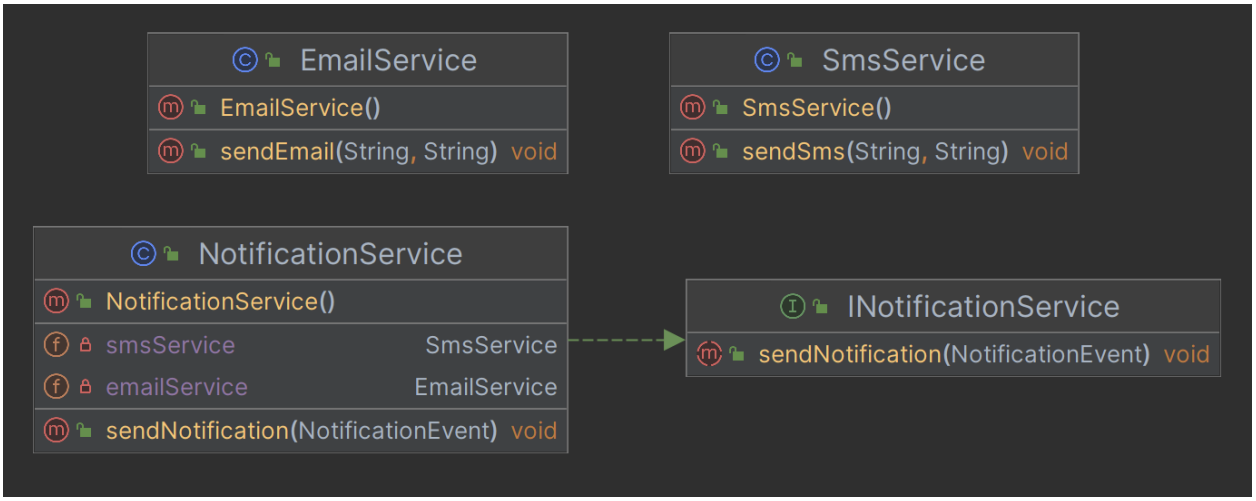

Figure 3.6.2: Notification Service

- **Dtos**

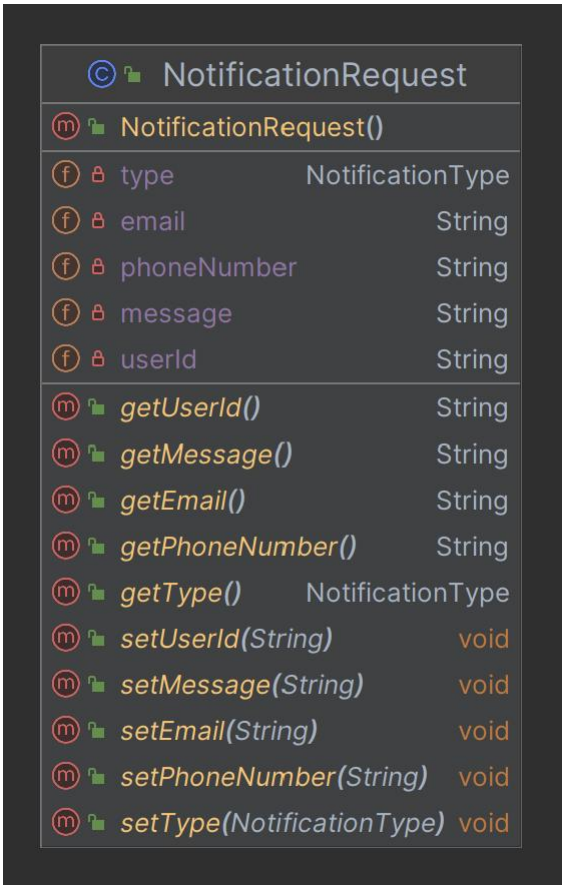

Figure 3.6.3: Notification Dtos

- **Consumers**

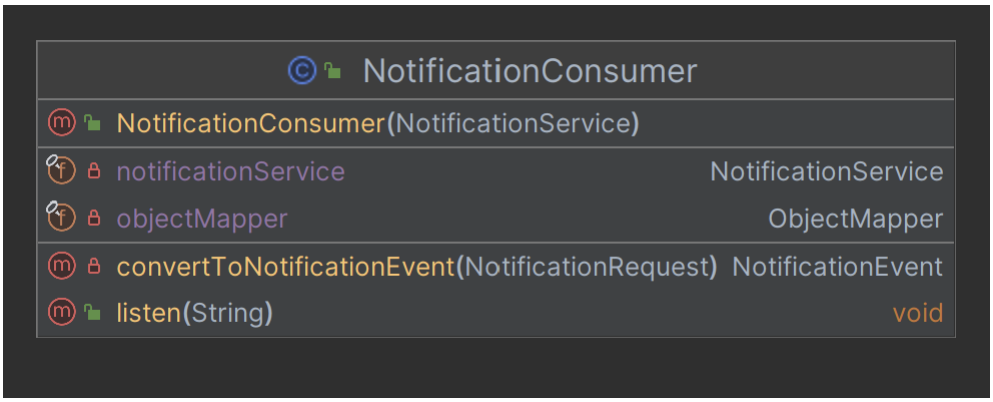

Figure 3.6.4: Notification Kafka Consumer

Database Schema Design

1. **User Management Service**
   - Tables

     users
     - id (PK)
     - state
     - created_at
     - updated_at
     - username
     - email
     - password
     - full_name

     sessions
     - id (PK)
     - state
     - created_at
     - updated_at
     - token
     - user_id (FK to users.id)
   - Foreign Keys
     - sessions(user_id) refers users(id)
   - Cardinality of Relations
     - Between users and sessions -> 1:m

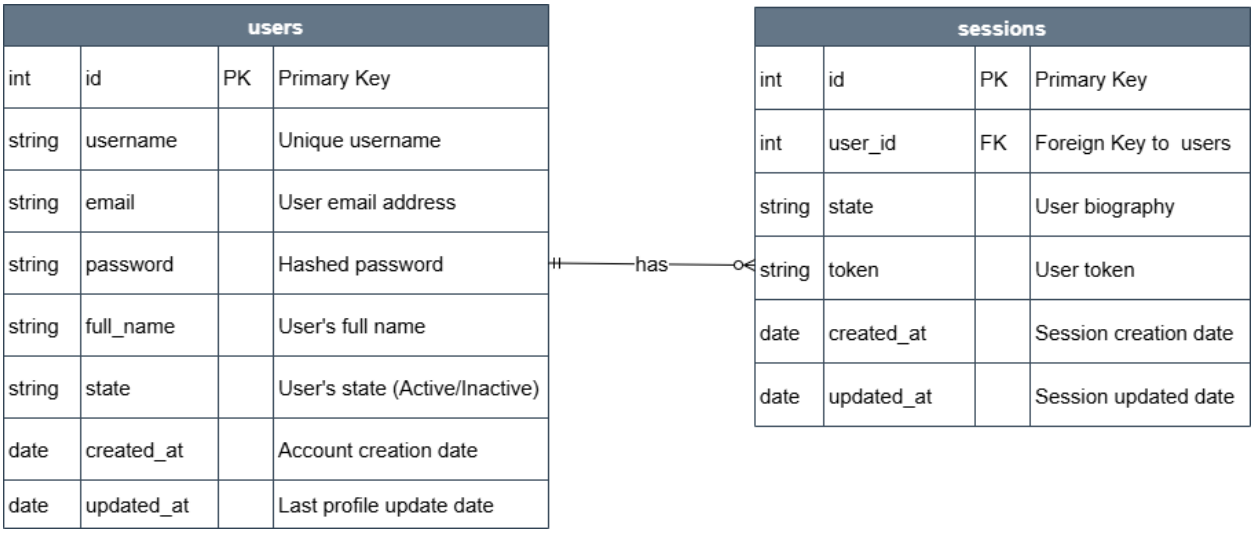

Figure 4.1: User Management Schema

2. **Product Catalog Service**
   - Tables

     products
     - id (PK)
     - created_at
     - updated_at
     - name
     - description
     - price
     - image_url
     - category_id (FK to categories.id)

     categories
     - id (PK)
     - created_at
     - updated_at
     - name
     - description
   - Foreign Keys
     - products(cateogory_id) refers categories(id)
   - Cardinality of Relations
     - Between products and categories -> m:1

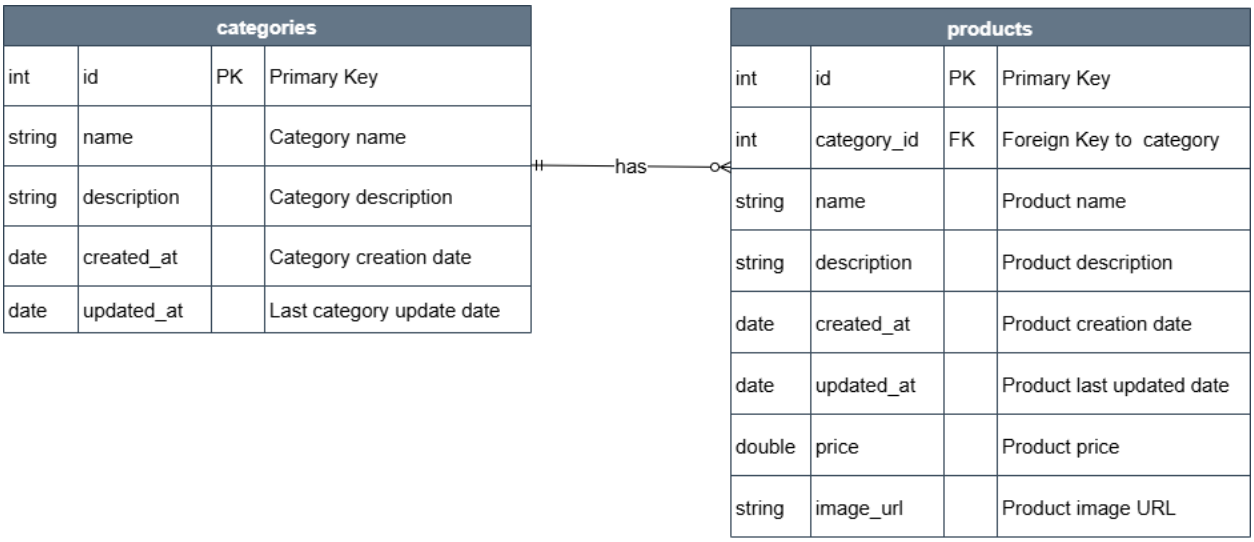

Figure 4.2: Product Catalog Schema

3. **Order Management Service**
   - Tables

     orders
     - id (PK)
     - created_at
     - updated_at
     - status
     - total_amount
     - customer_id

     order_items
     - id (PK)
     - created_at
     - updated_at
     - price
     - product_id
     - quantity
     - order_id (FK to orders.id)
   - Foreign Keys
     - order_items(order_id) refers orders(id)
   - Cardinality of Relations
     - Between order_items and orders -> m:1

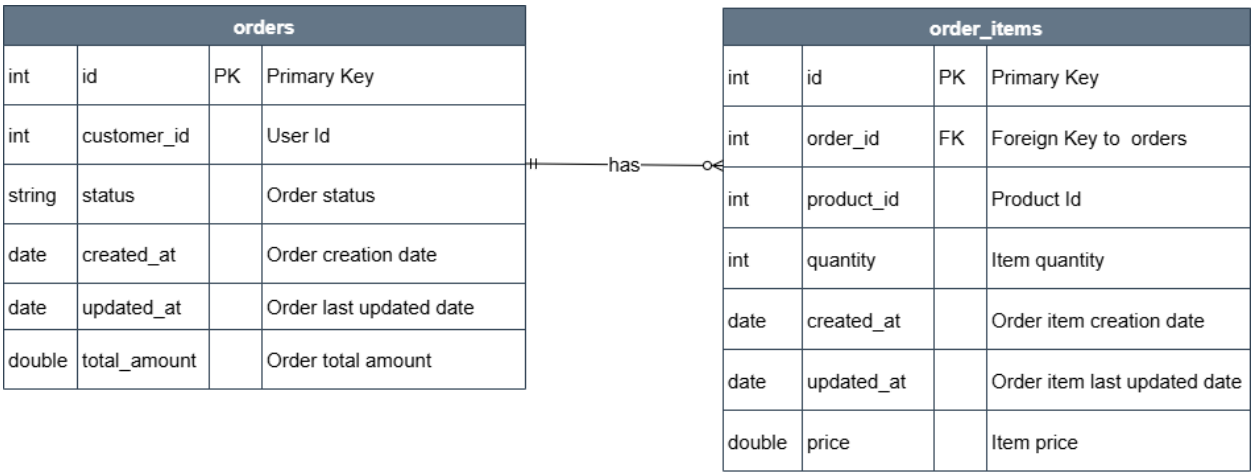

Figure 4.3: Order Management Schema

4. **Payment Service**
   - Tables

     payments
     - id (PK)
     - created_at
     - updated_at
     - amount
     - method
     - order_id
     - payment_reference_id
     - status

payments			
int	id	PK	Primary Key
double	amount		Payment amount
string	method		Payment method
int	order_id		Order Id
date	created_at		Payment creation date
date	updated_at		Payment last updated date
string	status		Payment status
int	payment_reference_id		Payment reference Id

Figure 4.4: Payment Schema

Feature Development Process

Feature: User's Cart Management

1. **Requirement Analysis**
   Goals: User should be able to
   - Add item to the cart
   - Remove items from the cart
   - View the cart
   - Empty the cart

2. **Design Phase**
   - Data model design
     Entities:
     - Cart
     - CartItem
   - API design
     RESTFul endpoints:
     - Add items: POST /api/cart/user/{userId}/items
     - Remove items: DELETE /api/cart/user/{userId}/items/{productId}
     - View cart: GET /api/cart/user/{userId}
     - Empty cart: DELETE /api/cart/user/{userId}
   - Component design
     Services:
     - CartService
     - ICartRepository

     DTOs:
     - CartDto
     - CartItemDtoq

3. **Development Phase**
   - Set up project structure (Spring Boot module)
   - Implement models, DTOs, mappers
   - Develop REST controllers
   - Implement business logic in services
   - Add JPA repositories or MongoDB access

4. **Testing Phase**
   - Unit Tests for service layer
   - Integration Tests for controller endpoints
   - Test scenarios:
     - Add item with valid/invalid product
     - Add duplicate item
     - Remove non-existent item

5. **Security and Validation**
   - Add authentication and authorization using AuthInterceptor which validates the token using **validateToken** api of User Management Service.

6. **Integration with Other Services**
   - User **Service:** Ensure cart is linked to authenticated user

7. **Monitoring & Logging**
   - Monitoring using **actuator:** /api/cart/actuator/health

8. **Load Testing**
   - During load testing it was observed that fetching the cart not meeting the expected Response Times as per NFR. To optimize this flow introduced **Redis caching** in cart service.

Table 5.1: Differences between Caching and Traditional DB Queries

Feature / Aspect	☑ Caching	🗄 Traditional DB Queries
Latency	Ultra-fast (microseconds to <1ms)	Slower (10–100ms or more)
Data Access Location	In-memory (e.g., Redis, Memcached)	Disk-based (e.g., MySQL, PostgreSQL)
Read Scalability	Very high (handles millions of QPS)	Limited by DB instance specs
Cost per Query (at scale)	Low (in-memory, fewer CPU/IO cycles)	High (more CPU, memory, disk I/O)
Load on Backend DB	Greatly reduced	High under read-heavy load
Consistency	May be eventual (stale data possible)	Strong (ACID-compliant reads)
Write Handling	Not designed for writes (usually read-only)	Fully supports writes, updates, transactions
Use Case Examples	Product catalog, user sessions, configs	Orders, payments, inventory updates
Failure Tolerance	Can serve cached data during DB outage	No data if DB is down
Deployment Complexity	Requires cache invalidation strategy	Simpler to implement without cache
Example Technologies	Redis, Memcached, CDN Cache, Guava	MySQL, PostgreSQL, MongoDB, Cassandra
Best When	Data is read-heavy & changes infrequently	Data is write-heavy or needs strong consistency

Deployment Flow

Deployment steps using Amazon EKS (Elastic Kubernetes Service). Amazon EKS is a fully managed Kubernetes service on AWS. It handles:

- Control plane provisioning and management
- High availability & scalability
- Integration with AWS services (e.g., IAM, VPC, Load Balancer)

1. **Prerequisites**
   - AWS CLI
   - kubectl (Kubernetes CLI)
   - eksctl (EKS provisioning CLI)
   - Docker
   - AWS IAM permissions (to create EKS cluster, VPC, etc.)

2. **Create an EKS Cluster**

   We can use `eksctl` for quick setup:

```
eksctl create cluster \
 --name ecommerceapp-cluster \
 --version 1.29 \
 --region us-west-1 \
 --nodegroup-name standard-workers \
 --node-type t3.medium \
 --nodes 5 \
 --nodes-min 1 \
 --nodes-max 5 \
 --managed
```

   This creates:

   - EKS cluster
   - Node group (EC2 instances running worker nodes)
   - VPC with networking

   This takes ~15 minutes.

3. **Build & Push Docker Image to Amazon ECR**
   - Build Docker image
     `docker build -t ecommerceapp .`
   - Tag for ECR
     `docker tag ecommerceapp:latest <aws_account_id>.dkr.ecr.us-east-1.amazonaws.com/ecommerceapp:latest`
   - Authenticate & push
     `aws ecr get-login-password | docker login --username AWS --password-stdin <ecr-url>`
     `docker push <aws_account_id>.dkr.ecr.us-west-1.amazonaws.com/ecommerceapp:latest`

4. **Deploy Your App to EKS**
   - Create a Kubernetes Deployment (deployment.yaml)
   - Expose the App (Service + Load Balancer) (service.yaml)
   - AWS will provision an **Elastic Load Balancer**.

5. **Monitor & Manage**
   - Logs
     `kubectl logs deployment/ecommerceapp`
   - Scale
     `kubectl scale deployment ecommerceapp --replicas=5`
   - Update image
     `kubectl set image deployment/ ecommerceapp ecommerceapp =<new-ecr-url>`

6. **Security**
   Use IAM roles for service accounts (IRSA), avoid hardcoded creds

7. **Secrets**
   Use **Kubernetes Secrets** or **AWS Secrets Manager**

8. **Scaling**
   Use Cluster Autoscaler & HPA

9. **CI/CD**
   Use GitHub Actions, ArgoCD, or CodePipeline

10. **Observability**
    Integrate **CloudWatch**, **Prometheus**, **Grafana**, or **AWS Distro for OpenTelemetry (ADOT)**

Technologies Used

1. **Java, Spring Boot**
   For application development
2. **Databases**
   - **MySQL**: For structured data.
   - **MongoDB**: For flexible, unstructured data.

Table 7.1: Comparison between MySQL and MongoDB

Feature / Aspect	MySQL	MongoDB
Database Type	Relational (SQL)	NoSQL Document-oriented
Data Model	Tables with rows and columns (schema-based)	Collections with flexible JSON-like documents
Schema	Fixed (predefined columns/types)	Dynamic (schema-less, per-document flexibility)
Query Language	SQL (SELECT, JOIN, etc.)	MongoDB Query Language (MQL – JSON-style)
Joins & Relationships	Supports complex joins	Limited join support ($lookup in aggregation)
Transactions	ACID-compliant (full support)	ACID (since 4.0 for multi-doc, native per-doc)
Scalability	Vertical (scale up) preferred	Horizontal (scale out) with sharding
Performance (Read-heavy)	Slower on complex queries with joins	Faster on reads with embedded documents
Performance (Write-heavy)	Good with indexes	Excellent for high-volume, flexible writes
Data Integrity	Strong with constraints (FK, unique, etc.)	Weak — must enforce at app level
Storage Format	Row-based (binary)	BSON (Binary JSON)
Indexing	B-tree, full-text, composite	B-tree, compound, text, geospatial
Use Cases	Banking, ERP, Inventory, traditional web apps	Real-time analytics, IoT, CMS, product catalogs
Examples of Use	Order tracking, HRMS, Financial transactions	Product catalogs, social apps, content storage
Ease of Setup	Easy, well-documented	Easy, document-based makes quick prototyping
Community & Maturity	Very mature, vast community	Growing fast, strong community
Tooling	phpMyAdmin, Workbench, JDBC, ORM tools	Compass, Mongo Shell, Mongoose, Atlas UI

3. **Kafka**

Central message broker allowing asynchronous communication between microservices, ensuring data consistency, and acting as an event store for critical actions.

Table 7.2: Comparison between Apache Kafka and RabbitMQ

Feature / Aspect	Apache Kafka	RabbitMQ
Type	Distributed event streaming platform	Traditional message broker (message queue)
Message Model	Publish–Subscribe with logs	Message Queueing (Queue–Consumer)
Persistence	Durable log-based storage (disk-based by default)	Durable queues (optional persistence)
Message Ordering	Guaranteed within partition	FIFO per queue (but not guaranteed across consumers)
Throughput	Very high (millions of msgs/sec)	Moderate to high (tens to hundreds of thousands)
Latency	Low latency but tuned for throughput	Lower latency (ms-level delivery)
Consumer Model	Pull-based	Push-based
Replayability	Yes (consume from offset anytime)	No (once acknowledged, message is gone)
Delivery Guarantees	At least once (exactly-once possible)	At least once, at most once
Routing	Topic-based routing only (basic filtering)	Advanced (topic, fanout, headers, direct exchanges)
Backpressure Handling	Retention-based, doesn't reject messages	Queue backpressure and flow control mechanisms
Use Case Fit	Event streaming, analytics, big data pipelines	Task queueing, real-time processing, RPC patterns
Built-in Retry/Dead Letter	No (must implement manually or via Kafka Streams)	Yes (DLQ and retry natively supported)
Ease of Setup	Complex (needs Zookeeper or KRaft, partitions)	Simpler (single node setup works well)
Message TTL	Retention policy based on time/size	Built-in per-message TTL
Message Size Support	Large (default ~1MB, can be tuned)	Better for smaller messages (~128KB or less)
Horizontal Scaling	Excellent (partition-based parallelism)	Harder (queues don't auto-scale)
Protocol Support	Kafka protocol (custom), limited client variety	AMQP, MQTT, STOMP, HTTP — more flexible
Tooling / UI	Kafka UI (Conduktor, Redpanda, AKHQ)	RabbitMQ Management Console

4. **Caching with Redis**
   Primarily by Cart Service for faster response times.
5. **Elasticsearch**
   Used by Product Catalog for fast and relevant product searches.
6. **HAProxy** for Load balancer
   Used **HAProxy** as a Load Balancer/API Gateway; a single-entry point to the application.

Table 7.3: Differences between Load Balancer and API Gateway

Feature / Aspect	Load Balancer	API Gateway
Primary Role	Distributes traffic across multiple instances/services	Entry point to manage, route, and secure APIs
Layer	OSI Layer 4 (TCP) or Layer 7 (HTTP)	Layer 7 (HTTP/HTTPS, API protocols)
Traffic Type	Generic (HTTP, HTTPS, TCP, UDP)	Application/API-specific traffic only
Routing Logic	IP/Port-based, round-robin, least connections, etc.	URL path, method, headers, auth tokens
Authentication / AuthZ	Not supported (needs external system)	Yes. Supports OAuth, JWT, API Keys, etc.
Request Transformation	Not supported	Yes. Can modify headers, body, paths
Rate Limiting / Throttling	Not available	Yes. Built-in support (e.g., per user/IP)
Caching / Compression	Yes. (some advanced LB like NGINX support it)	Yes. Often built-in for performance
Monitoring / Logging	Basic traffic stats	Advanced metrics, logging, tracing
Service Discovery	Yes. Often integrated with DNS or cloud-native SD	Yes. Dynamic routing based on service registry
SSL Termination	Yes. Commonly used	Yes. Also handles cert management
Security Controls	Basic (IP whitelisting, SSL termination)	Fine-grained (auth, scopes, firewall rules)
Example Tools	AWS ELB/ALB, HAProxy, NGINX	Spring Cloud Gateway, Kong, Istio Gateway, Amazon API Gateway
Best Used For	Distributing load to instances, high availability	Managing and securing microservice API calls

Conclusion

The development of this e-commerce project has provided valuable insights into designing scalable, maintainable, and modular backend systems. By adopting a microservices-based architecture and implementing services like user, cart, order, payment, and notification independently, the system ensures flexibility, reusability, and ease of future expansion.
This project followed modern development practices—such as secure session management, asynchronous messaging with Kafka, and RESTful API design—along with cloud deployment strategies to simulate a real-world e-commerce platform.

- **Key Takeaways**
  1. Microservices architecture enables better scalability and team autonomy, especially when each service is loosely coupled and independently deployable.
  2. Event-driven communication using Kafka enhances responsiveness and reliability, especially for order, payment, and notification flows.
  3. Service decoupling with proper API contracts and data modelling improves maintainability and simplifies future enhancements.

- **Practical Applications**
  1. Can be used as a reference or base framework for building real-world e-commerce platforms.
  2. Components like the Cart, Order, and Payment services are modular enough to be integrated into other domain-driven systems.
  3. Demonstrates production-grade backend practices that are directly applicable in industry settings.

- **Limitations**
  1. The current version lacks a UI or frontend layer to complete the full user experience.
  2. Certain features like fraud detection, search optimization, or third-party payment gateways are either mocked or simplified.
  3. Rate limiting, and detailed audit logging are minimal and can be improved for production readiness.
  4. Containerization and orchestration (e.g., Docker, Kubernetes) are not yet fully integrated, limiting environment portability and automated scaling.

References

- PRD Document: https://docs.google.com/document/d/1Gn2ib5YhhpcFUiWGAUbCpg0ZPh3m_wSA-9IolGMjkIE/edit?tab=t.0#heading=h.hteovoit9b96
- https://chatgpt.com/
- https://app.diagrams.net/