Elio Al Alam - 3717403

Promise Ononokpono - 3692431

Christopher Pelletier - 3703390

Luvneet Bamrah - 3709995

SWE4403 - Software Architecture and Design Patterns

Dr. Julian Cardenas Barrera

Project 1

Microservices - WebBazaar

Quick Start Guide

Starting the Program:

- Open every microservice on its own window.
- Start all the microservices before starting the front-end
- Once all microservices have started, start the Frontend application.
- Choose an Option to Login or Signup (Admin accounts are pre-created; only customer accounts can be created.)

- Pre-set customer account:
    - Username: Elio (or Chris)
    - Password: Pass

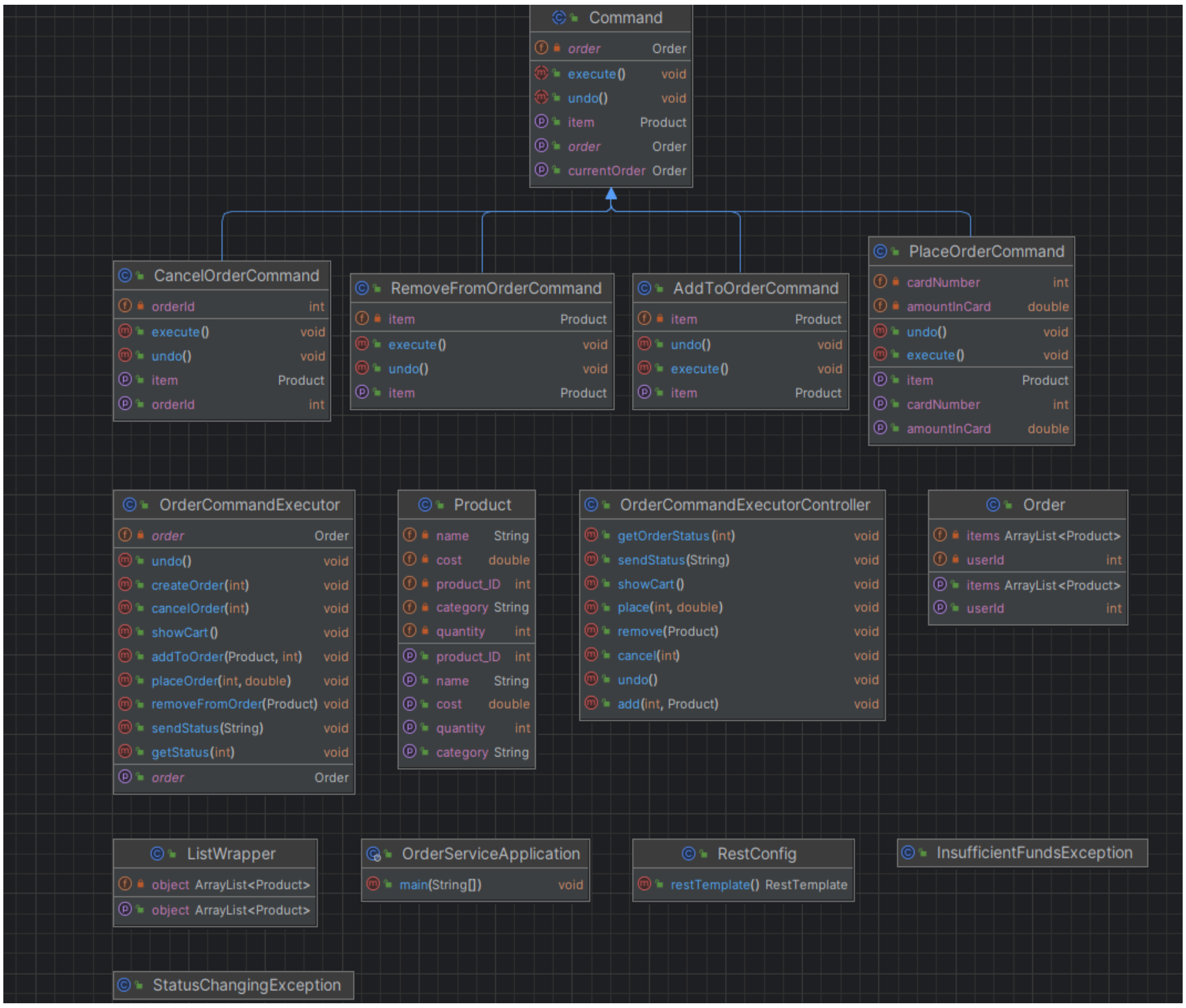
- Pre-set Admin(retailer) account:
    - Username: Promise
    - Password: Pass

- Customer Privileges:
    - View Profile info
    - View Categories
    - View Products in each Category
    - View Previous Orders and the status of each order for a user.
    - Cancel Processing orders.
    - Add Items to Cart
    - Remove Items from the Cart
    - View Cart
    - Place Order
    - Logout

- Admin (Retailer) Privileges:
    - View Profile info
    - View Categories
    - Create/Read/Update/Delete Products from Categories
    - Logout

- Note: The Database Server will stay running until April 30th, 2024

Order Service



Order Design Analysis:

Service Decomposition: The codebase demonstrates decomposition into multiple classes, each responsible for a specific command or functionality related to the order service. Each class represents a microservice-like entity handling a specific responsibility.

Command Pattern: The use of the Command pattern is evident, where commands (Command and its subclasses) encapsulate actions (such as adding, removing, placing, or canceling an order) and allow for decoupling of invokers (OrderCommandExecutor) from the actual operations.

Service Layer Abstraction: The service layer (OrderCommandExecutor and its dependencies) encapsulates business logic and interacts with external services via REST calls. This ensures the separation of concerns and facilitates easier testing and maintenance.

RESTful Communication: Communication between microservices is achieved through RESTful HTTP requests (RestTemplate) to endpoints another services expose. This promotes interoperability and flexibility.
`

Spring Boot: Leveraging Spring Boot annotations and features (e.g., @Service, @Autowired) simplifies configuration, dependency injection, and overall development of the microservice.

Data Transfer Objects (DTOs): The use of DTOs (e.g., Product, Order) helps in representing data transferred between microservices and clients, promoting loose coupling and preventing domain model leakage.

Resiliency: The use of exception handling (e.g., StatusChangingException, InsufficientFundsException) and the ability to undo operations (e.g., undo() method) contributes to the resilience of the system.

Applied Patterns:
Command Pattern: The Command abstract class and its subclasses (AddToOrderCommand, CancelOrderCommand, PlaceOrderCommand, RemoveFromOrderCommand) implement the Command pattern. This pattern encapsulates a request as an object, allowing for the parameterization of clients with queues, requests, and operations.

Singleton Pattern: The OrderCommandExecutor class uses a singleton pattern to ensure that only one instance of the class is created, managing the order queue and operations.

Facade Pattern: The OrderCommandExecutor class acts as a facade, providing a simplified interface to a complex subsystem of order operations.

Quality Attributes Analysis:
Maintainability: The codebase follows best practices such as separation of concerns, encapsulation, and abstraction, enhancing maintainability by making it easier to understand and modify individual components.

Scalability: While the current design allows horizontal scaling of the order service by deploying multiple instances, introducing asynchronous communication and load balancing mechanisms could further enhance scalability.

Reliability: The use of exception handling and retries in case of failures improves the reliability of the service. However, more comprehensive error handling and fault tolerance mechanisms could be implemented for robustness.

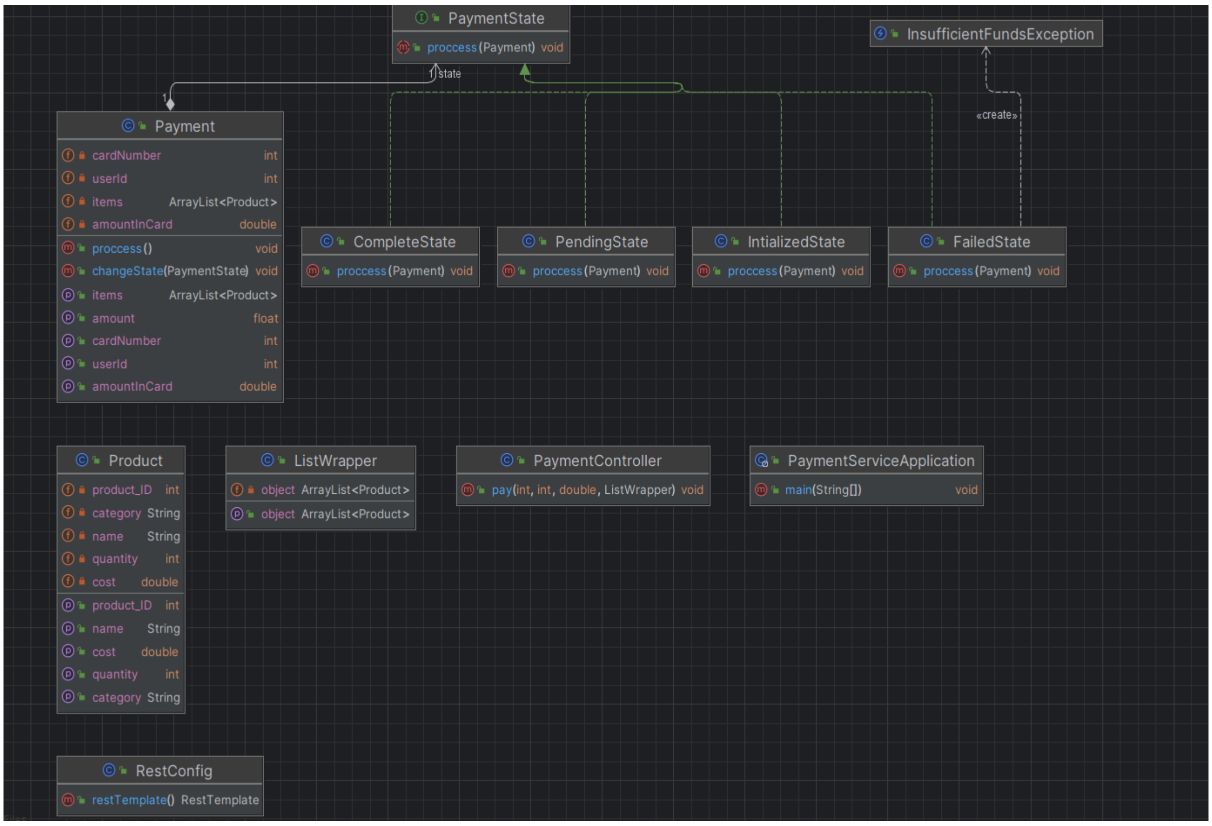
Performance: The synchronous nature of communication might introduce latency, especially during high loads. Introducing caching mechanisms and optimizing RESTful calls could improve performance.

Security: The codebase lacks explicit security measures such as authentication and authorization. Implementing security protocols (e.g., OAuth, JWT) and encryption would enhance the security posture of the microservice.

Testability: The design facilitates unit testing of individual components due to loose coupling and encapsulation. However, comprehensive integration and end-to-end testing are necessary to validate interactions between microservices.

Flexibility: The use of design patterns and abstraction layers provides flexibility to accommodate changes in business requirements or technology stacks without extensive rework. However, the service might benefit from more modularization to enhance flexibility further.

Payment Service:



Microservices Design Analysis:
Service Decomposition: The codebase follows a state pattern design where different states of payment processing are encapsulated into separate classes (PaymentState). Each state handles a specific aspect of payment processing, promoting separation of concerns.

State Pattern: The state pattern is employed to model the lifecycle of a payment. States encapsulate behaviours associated with different stages of payment processing, enabling clean and maintainable transitions between states.

Service Layer Abstraction: The Payment class acts as a service layer that interacts with different payment states (PaymentState) and external services (e.g., order placement service) via REST calls. This abstraction allows for the encapsulation of payment logic and facilitates easy integration with external systems.

RESTful Communication: Communication with external services is achieved through RESTful HTTP requests (RestTemplate). This approach promotes interoperability and enables the payment service to interact with other microservices seamlessly.

Spring Boot: The use of Spring Boot annotations (@Service, @Autowired) simplifies the configuration and development of the payment service, enhancing productivity and maintainability.

Resiliency: The use of exception handling (e.g., InsufficientFundsException) and the ability to change states (e.g., from PendingState to CompleteState or FailedState) contribute to the resilience of the system.

Applied Patterns:
State Pattern: The PaymentState interface and its implementations (IntializedState, PendingState, CompleteState, FailedState) implement the State pattern. This pattern allows an object to alter its behaviour when its internal state changes, providing a way to encapsulate state-specific behaviour.

Facade Pattern: The Payment class acts as a facade, providing a simplified interface to a complex subsystem of payment operations.

DTO Pattern: The Product class serves as a Data Transfer Object (DTO) to represent product information transferred between services, promoting loose coupling and encapsulation.

Quality Attribute Analysis:
Maintainability: The use of design patterns such as the state pattern and dependency injection enhance maintainability by promoting modularization, encapsulation, and loose coupling.

Scalability: The design allows for horizontal scaling of the payment service by deploying multiple instances. However, scaling might be limited by external dependencies and resource constraints.

Reliability: The codebase includes exception-handling mechanisms to deal with errors and failures gracefully, improving the reliability of the payment service. However, more comprehensive error handling and fault tolerance mechanisms could be implemented.

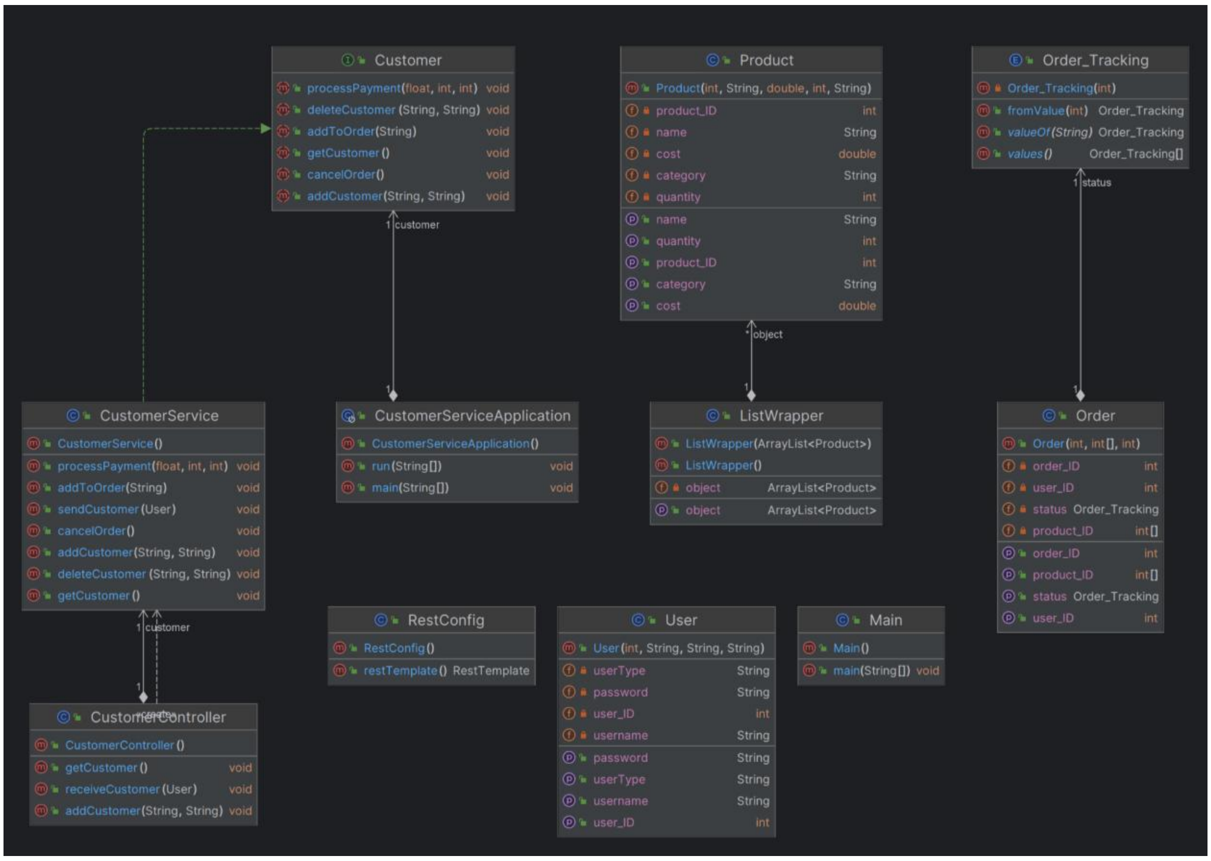
Performance: The synchronous nature of communication with external services and the use of blocking operations (e.g., Thread.sleep) might introduce latency, especially during high loads. Asynchronous communication and optimization of RESTful calls could improve performance.

Security: The codebase lacks explicit security measures such as authentication and authorization. Implementing security protocols (e.g., OAuth, JWT) and encryption would enhance the security posture of the payment service.

Testability: The design facilitates unit testing of individual components (e.g., payment status, REST endpoints) due to loose coupling and dependency injection. However, comprehensive integration testing is necessary to validate interactions with external services.

Flexibility: The use of design patterns and abstraction layers provides flexibility to accommodate changes in business requirements or technology stacks without extensive rework. However, the service might benefit from more modularization to enhance flexibility further.

Customer Service:



Customer Service Design Analysis:

Service Decomposition: The CustomerService codebase is structured in such a way, that each method represents a specific functionality related to customer management. This decomposition facilitates easier maintenance and testing of individual features.

Service Layer Abstraction: The CustomerService class encapsulates the business logic related to customer management and interacts with external services via REST calls. This abstraction ensures separation of concerns and promotes easier integration with other microservices.

RESTful Communication: The communication between the CustomerService and other services occurs through RESTful HTTP requests using Spring's RestTemplate. This approach promotes interoperability and flexibility in the system architecture.

Spring Boot: The codebase leverages Spring Boot annotations and features such as @Service and @Autowired, simplifying configuration, dependency injection, and overall development of the microservice.

Data Transfer Objects (DTOs): The User object serves as a Data Transfer Object (DTO) for transferring user-related data between microservices and clients. This helps in maintaining loose coupling and prevents domain model leakage.

Applied Patterns:

Service Layer Pattern: The CustomerService class encapsulates the business logic related to customer management, adhering to the service layer pattern. This pattern promotes separation of concerns by separating business logic from presentation and data access layers.

RESTful Communication: The codebase utilizes RESTful communication for interaction with other microservices, following the principles of Representational State Transfer (REST). This architectural style enables stateless communication and promotes scalability and flexibility.

Singleton Pattern (RestTemplate): Although not explicitly implemented in the provided code, the RestTemplate instance in the CustomerService class effectively follows the principles of the Singleton pattern. By default, Spring beans (such as RestTemplate) are singletons, ensuring that only one instance is created and shared throughout the application context.

Quality Attributes Analysis:

Maintainability: The codebase maintains good maintainability by adhering to best practices such as separation of concerns and encapsulation. This makes it easier to understand and modify individual components.

Scalability: The current design allows for horizontal scaling of the customer service by deploying multiple instances. However, introducing asynchronous communication and load-balancing mechanisms could further enhance scalability.

Reliability: Exception handling in the codebase improves reliability by handling potential errors gracefully. More comprehensive error handling and fault tolerance mechanisms could be implemented to enhance reliability further.

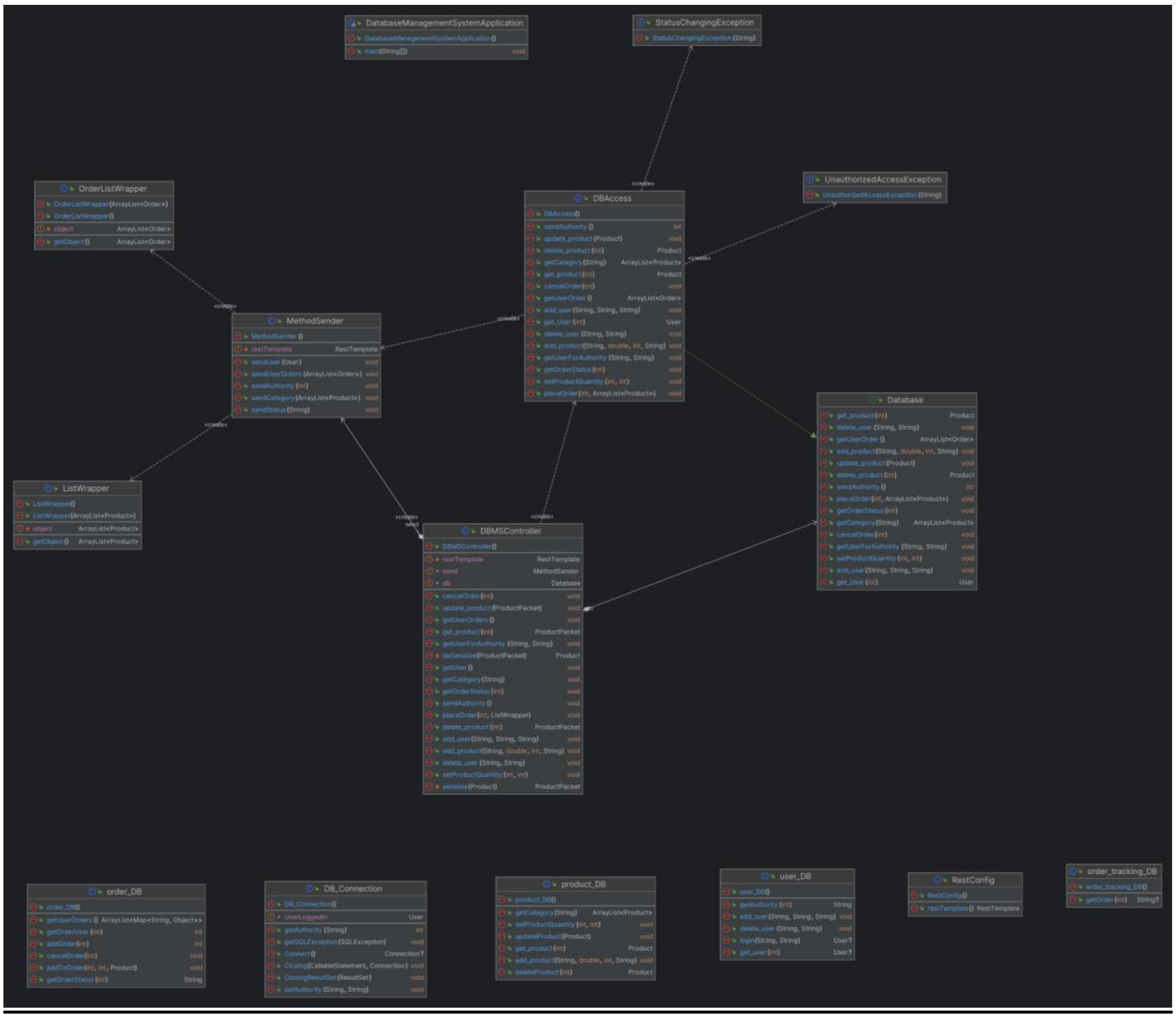
Performance: Asynchronous communication and optimization of RESTful calls could improve performance, especially during high loads. Introducing caching mechanisms may also enhance performance further.

Security: The codebase lacks explicit security measures such as authentication and authorization. Implementing security protocols and encryption would enhance the security posture of the microservice.

Testability: The design facilitates unit testing of individual components due to loose coupling and encapsulation. Comprehensive integration and end-to-end testing are necessary to validate interactions between microservices effectively.

Flexibility: The use of design patterns and abstraction layers provides flexibility to accommodate changes in business requirements or technology stacks.

Database Management System



Service Decomposition: This service follows a Proxy Pattern demonstrated in some of its method that require extra security. This service provides a secure connection to a MySQL database used to store information about users, Products, and Orders. Each Class in this microservice represents a specific Responsibility.

Proxy Pattern: The Proxy Pattern was used to authorize some methods that require further security. For example, methods like creating a new product and updating the information for it are only authorized for “retailer” users.

RESTful Communication: The Database Management System is communicated with through RESTful HTTP commands received from different microservices.

Spring Boot: The Use of Spring Boot annotations (@Service, @RestController, …) simplifies the configuration and development of the Database Management System, enhancing productivity.

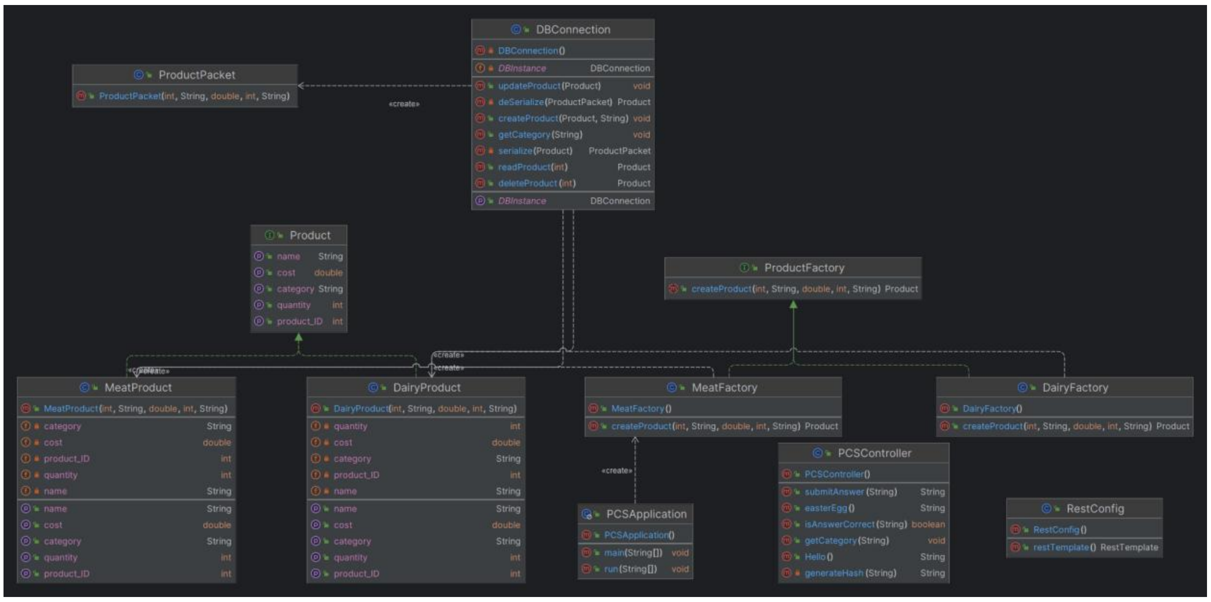
Quality Attribute Analysis:

Maintainability: The Microservice follows the maintainability attribute by receiving the information requested and delivering them to SQL Stored Procedures without altering them
Scalability: This microservice is easily scalable with the use of interfaces. The microservice allows new features to be easily integrated into the software.

Reliability: The microservice handles Exception errors, which does not crash the application but stops the feature from stepping over the error and terminates that method from the stack

Availability: Microservices are separate modules, where if one microservices crashes, only the features that use that microservice are down, but any other feature that is independent from that microservice is still available`

Product Catalog Service (PCS) System



Microservices Design:

The product catalogue service was hosted on its own microservice in order to have this business capability separated from the rest of the system. The goal of this system is to keep track of available product listings by having connections to both the front end and the database thus acting like a proxy in between the two.

Due to this design choice, other microservices can be developed, modified and tested without affecting this service.

Applied Patterns:

In this service, the following design patterns have been applied:

The factory pattern was used to create objects without specifying the exact class of objects that could be created. In my service, ProductFactory.java abstracts the creation of Product instances with its implementations: MeatFactory and DairyFactory.
A singleton pattern was used to manage the database instance class to ensure that the class is only instantiated once and provides global access to the instance. This manages the instance efficiently to reduce the load on system resources.

A service layer pattern was also used by having services like DBConnection and RestConfig to abstract logic related to external interfaces and databases. This promotes a separation of concerns and improves modularity.

Quality Attributes Analysis:

Scalability:

Due to the nature of Microservice applications, this project is very scalable because we can independently scale functionalities without affecting other services or having to scale the entire application.

Maintainability:
Due to the factory pattern and the service layer pattern, the application promotes maintainability due to the code being modular and the fact that there is a separation of concerns. This makes it easy to modify part of the service (adding a new product for example) much easier and thus making the system easier to maintain.

Availability:
Even though microservices help with availability due to independent deployment of the services we still have to take into consideration that the singleton pattern that has been implemented can cause bottlenecks with the database connection.

Performance:
Because we use RESTful services and a microservice architecture, it allows our system to have distributed processing and data management. Although, we still have to take into consideration that network latency can be a challenge with performance.

Security:
The usage of microservices for security can be a double-edged sword. On one hand, you can tailor specific security features for each of the microservices but on the other hand, this makes the application as a whole more complex due to the coordination required.