**Bookstore Web Application**

Parisa Khosravi

Student No: GH1027620

**Abstract**

The Project discussed in this report is a Java Spring Boot application for an E-commerce platform at the management level which belongs to a book store. The project has a layered architecture following an MVC pattern, comprising of Entity, Repository, Service and Controller layers. The application connects to a SQL database with multiple tables, supports CURD operations and maps different tables together to handle orders, products and customers.

Link to the Github Repository

<https://github.com/parisakh4/E-Commerce-Java/tree/master/E-commerce-project/src/main/java/com/gismateaching/E_commerce_project>

Link to the Video Demonstration in YouTube

**Introduction**

E-commerce platforms are one of the most popular use cases for web applications, requiring a well-developed back-end system for both online shopping and managing an online business. This project builds a back-end system for a bookstore online platform with the use of Java Spring Boot at the application layer and MySQL for the database. The goal of the project at this phase to handling CURD operations related to Customers, Orders, Products, Suppliers and Order Details, as well as relational mapping for handling relationships between these entities. To interact with the backend system a RESTful API is also developed.

One of the main focuses of this project was to try and go deep into different CURD operations and services, which is why at this phase the system has integrated five out of nine tables of the database, to try and go further into different functions and methods, specifically for operations related to order handling and processing, which is one of the main parts of every E-commerce platform. In future developments all the tables will be included and interacted with. Consequently, the code is developed to be maintainable and scalable for further development using a layered architecture.

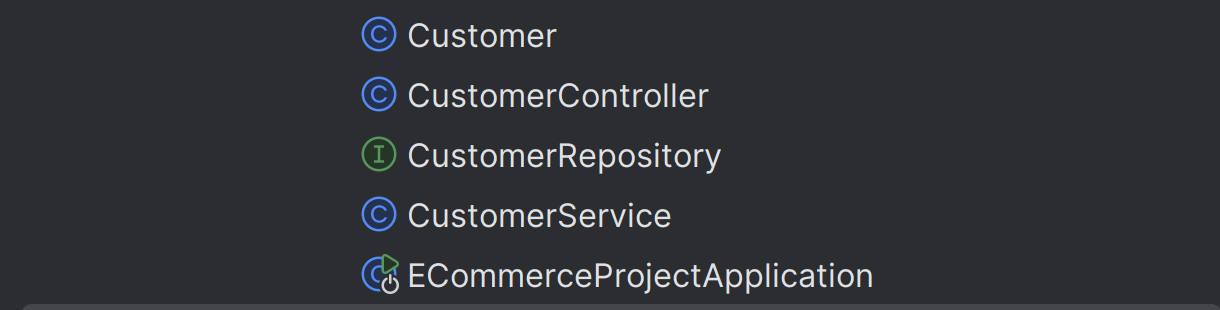
This report covers all the stages of production, including system design, database architecture, API handling, error handling, challenges and solutions and in the end, conclusion and future developments.

**System Design**

The system design follows an MVC pattern, meaning there are four different layers of code.

* Entity layer, that defines the database schema.
* Repository layer, which uses the JPA for database operations.
* Service layer, in which the business logic and operations are defined.
* Controller layer, that maps the HTTP requests to methods in service layer.

Ideally, these layers can be organized in separate packages for clarity and maintainability. In this phase of project however, the files are all located in the main package, “com.gismateaching.E\_commerce\_project”.



**Database Architecture**

The database for this system has nine tables; customers, addresses, orders, order details, transactions, order logs, products, product rating and suppliers. However, the current version of this application works with five tables including customers, orders, order details, products and suppliers. The customers table is in a one to many relationship with the orders table, the orders table and product table are in a one to many relationship with the order details table. The suppliers table is also in a one to many relationship with the products table. The relation between the tables is shown in the ER diagram in figure one. The included table are highlighted.

**API Handling**

The API handling uses the RESTful system with the help of @ResrController annotations from Spring Boot. The system exposes endpoints for CURD operations on the entities including Customers, Orders, Order Details, Products and Suppliers. Each of these operations maps to one of the HTTP methods including GET, POST, PUT and delete. The endpoints consist of name of the entity and mostly the HTTP method that they are using, with a few exceptions for more complex operations. In cases of an input request, the input is added at the end. The HTTP methods in this project are:

**GET**

* /Customer/get
* /Customer/withOrders/{customerId}
* /Order/get
* /OrderDetail/get
* /OrderDetail/pending
* /OrderDetail /order/{orderId}
* /Product/get
* /Supplier/get
* /Supplier/products/{supplierId}

**POST**

* /Customer/add
* /Order/add
* /OrderDetail/add
* /Product/add
* /Supplier/add

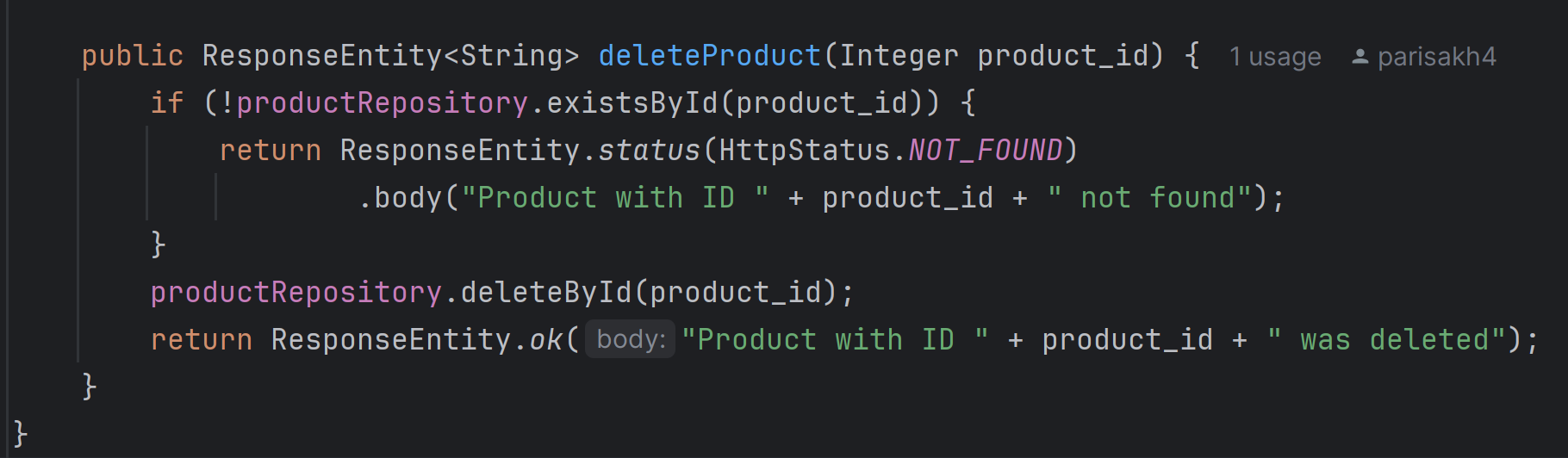
**PUT**

* /Customer/update/{customerId}
* /Order/update/{orderId}
* /Orders/{orderId}/status
* /OrderDetail/update/{orderDetailId}
* /Product/update/{productId}
* /Supplier/update/{supplierId}

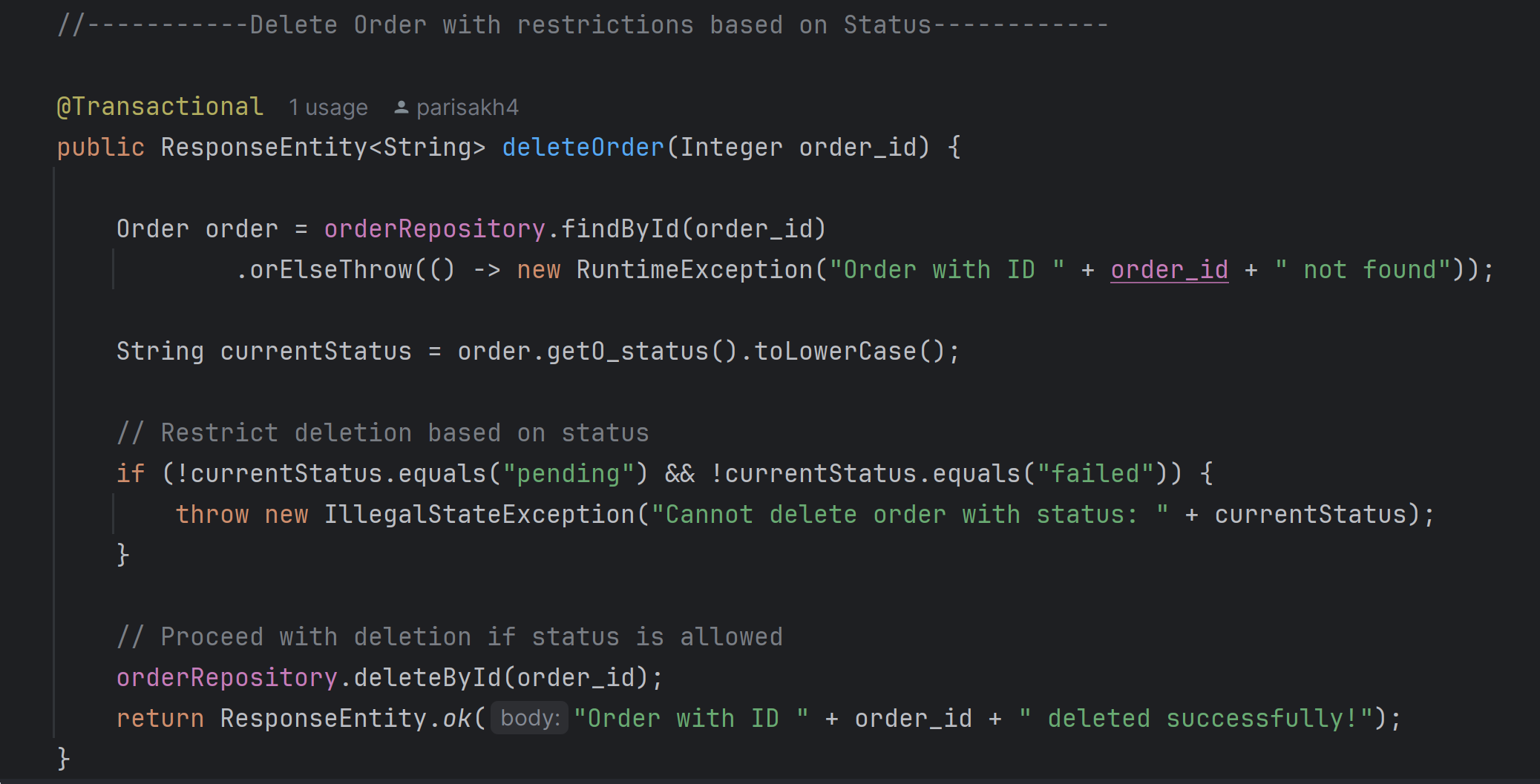
**DELETE**

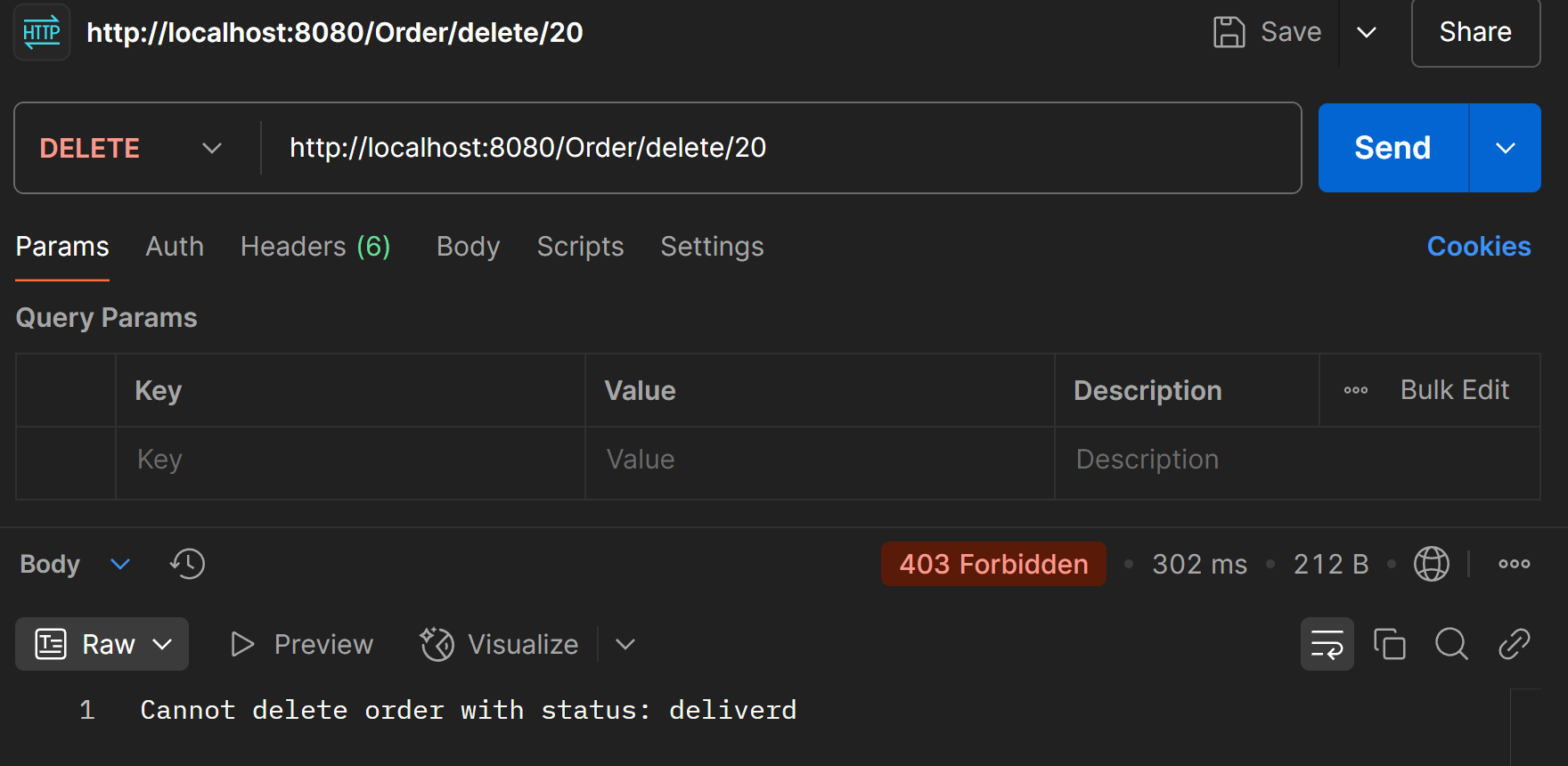
* /Customer/delete/ customerId}
* /Order/delete/{orderId}
* /OrderDetail/Delete/{orderDetailId}
* /Product/delete/{productId}
* /Supplier/delete/{supplierId}

**Development**

Regarding the development process, at the first step, the basic methods were created for each entity, i.e. simply adding customers or updating orders or deleting products without any logic or restrictions, along with manual error handling. 

In the next phase, some of these methods were refined. For instance, for order update and deletion, restrictions were made based on their status. If and order was completed, the status could only change to shipped and the order cannot be deleted. Additionally, the error handling was refined with throwing exceptions.

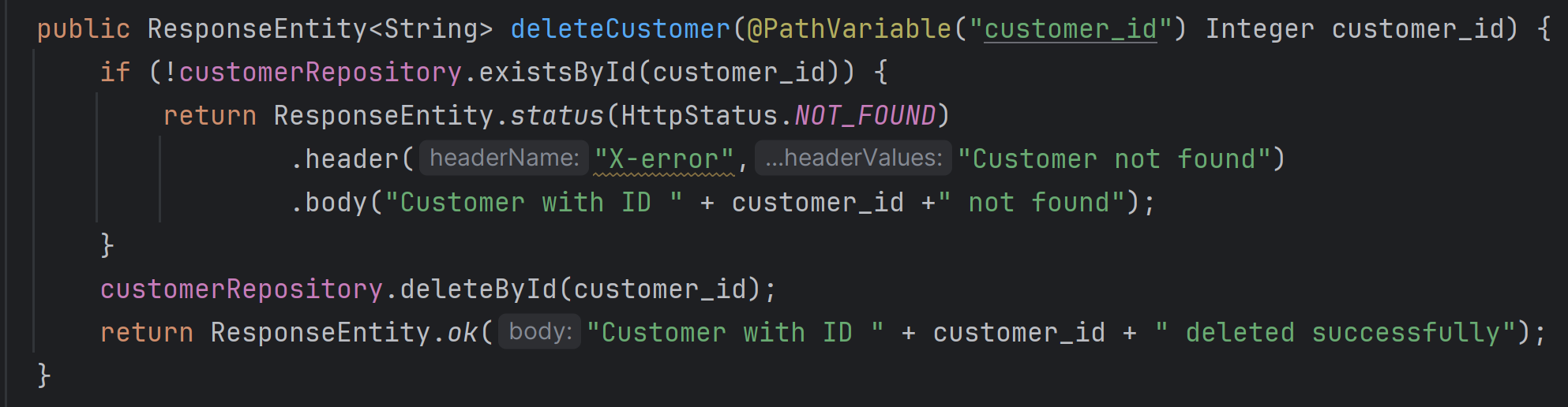




Next, the entities relationships were defined and mapping was done for further operations using annotations like @OneToMany and @ManyToOne. These operations range from simple, such as retrieving all orders for one customer and all products from one supplier, to the more complex ones, the most important being operation that forms the order flow, including three crucial operations for creating new order, handling the order to check the stock and adding order details and finally completing the order.

**Error Handling**

At the early phases of development, there error handling was being done manually, returning a ResponseEntity with status code, i.e. NOT\_FOUND and a customize response, as shown in the code below.



As the project developed further, a Global Exception Handler was created using three Exceptions:

* Runtime Exception when such exceptions are thrown, i.e. when the customer is not found.
* Illegal State Exception which prevents unauthorized action according to business logic. An example is changing the status of an order from “shipped” to “pending”.
* General Exception which is used for unexpected errors in the code.

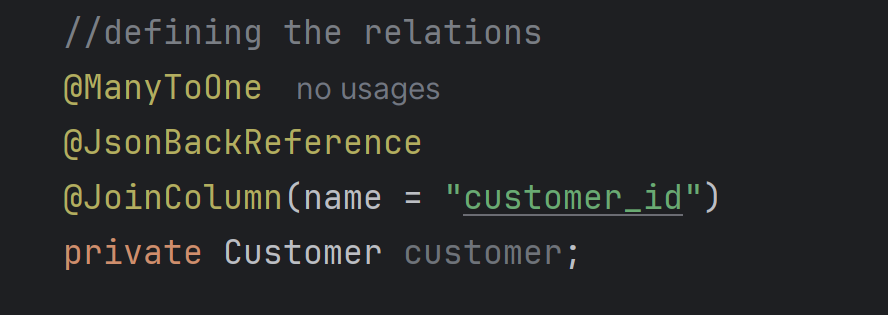
and used in the service layer. This method allows for a consistent and centralized error handling and keeps the code simpler and cleaner.



**Challenges and Solutions**

While trying to define the relationships between the entities and do the mapping, an error was thrown by Hibernate, stating that the foreign key column was mapped twice. To solve this issue, the foreign key field was removed from the entity and it was only accessed through the mapping. This is explained further in the example below.

When trying to join the customer and order tables to write a method for getting a customers’ orders, there was an error saying the “customer\_id” column in the order class was being mapped twice. This was caused because I had the code below to define the many to one relationship, pointing at “customer\_id” column for joining the customer table, and I had the “customer\_id” as a field in my order entity.



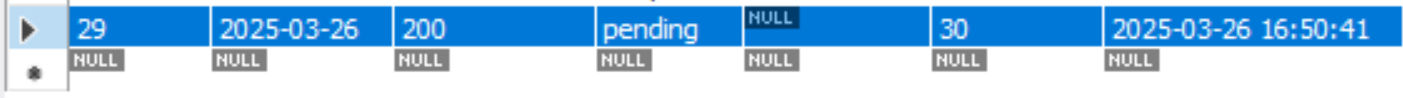
The solution implemented at first was adding the following to the “customer\_id” field.

@Column(name = "customer\_id", insertable = false, updatable = false)

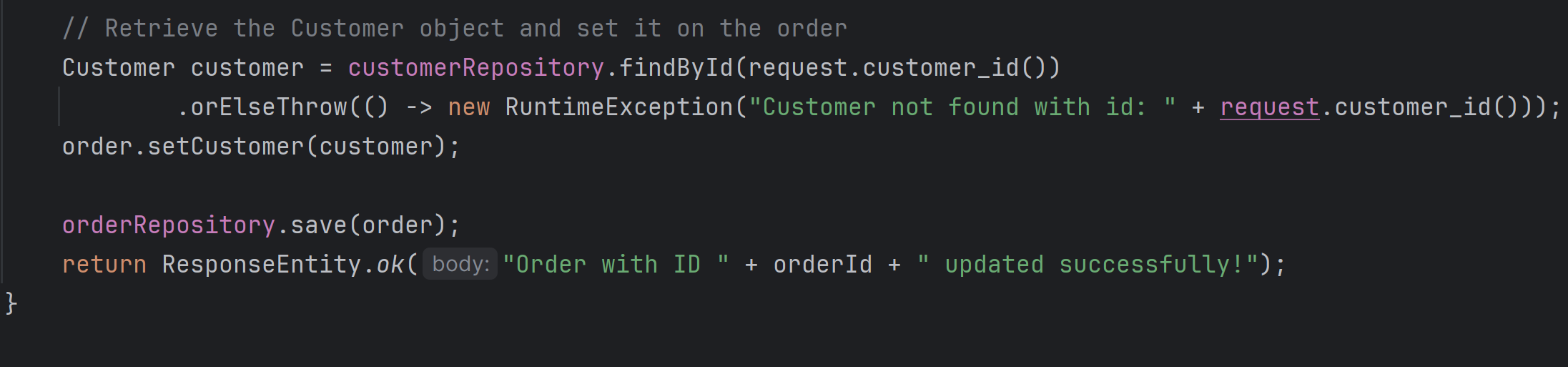
However, when creting method to add an order, this was problematic because the customer id could not be added, due to this part of th code:

inserable = false, updatable = false

As a result, the order was added to the orders table with null value for customer id



So the “customer\_id” field was deleted from the Order entity and mapping was used to retrieve the customer id from the customer entity and use it in the entity and the service layer. Below is a snapshot of order update retrieving the customer object.



**Conclusion and Future Developments**

As explained in this report, with the use of Spring Boot, an E-commerce application layer was created to interact with a database to perform a range of CURD operation from simple to complex. At this phase, the first action to be taken is to edit the manual error handling to all use the global error handling, so that the code is consistent. Secondly, not all the tables in the database were included. For the future development, the remaining tables can be added and mapped. Consequently, in the next phases the payment can be integrated, status change for order and payment can be automated, different endpoints based on user roles could be implemented with the option to login and register and finally, a front end could be developed to have a complete E-commerce application.