Alberto Escamilla Jasso

Final Project Java Course Xideral

September 2024

Entities

```java
1  package com.java.FinallProjectJava.entity;
2
3  import com.fasterxml.jackson.annotation.JsonIgnore;
14
15 @Entity // Marks this class as a JPA entity
16 @Data // Lombok: generates getters, setters, toString, equals, and hashCode
17 @NoArgsConstructor // Lombok: generates a no-arguments constructor
18 public class Product { // Represents a product entity
19
20     @Id // Marks this field as the primary key
21     @GeneratedValue(strategy = GenerationType.IDENTITY) // Auto-generates ID values
22     private Long id; // Unique identifier for the product
23
24     private String name; // Product name
25
26     private String description; // Product description
27
28     private double price; // Product price
29
30     @ManyToOne(fetch = FetchType.LAZY) // Many products can be associated with one order
31     @JoinColumn(name ="order_id", nullable =false) // Maps the foreign key to the order
32     @JsonIgnore // Prevents circular references during JSON serialization
33     private CustomerOrder customerOrder; // The associated order for this product
34 }
35
36
```

```java
1  package com.java.FinallProjectJava.entity;
2
3  import java.io.Serializable;
18
19 @Entity // Marks this class as a JPA entity
20 @Data // Lombok: generates getters, setters, toString, equals, and hashCode
21 @NoArgsConstructor // Lombok: generates a no-arguments constructor
22 public class CustomerOrder implements Serializable { // Implements Serializable for message conversion
23
24     private static final long serialVersionUID = 1L; // Ensures compatibility during serialization
25
26     @Id // Marks this field as the primary key
27     @GeneratedValue(strategy = GenerationType.IDENTITY) // Auto-generates ID values
28     private Long id; // Unique identifier for the order
29
30     @NotNull // Ensures this field cannot be null
31     private String customerName; // Customer's name
32
33     private String orderDate; // Date when the order was placed
34
35     @OneToMany(mappedBy = "customerOrder", orphanRemoval = true, cascade = CascadeType.ALL, fetch = FetchType.LAZY)
36     // OneToMany relationship: an order can have multiple products
37     @ToString.Exclude // Excludes products from toString() to avoid circular reference
38     private List<Product> products; // List of products associated with this order
39 }
```

First, we define two main entities, `CustomerOrder` and `Product`, which represent the core structure of the application's data model. The `CustomerOrder` entity encapsulates information about a customer's order, such as the customer's name and the order date. It also maintains a list of `Product` instances, representing the products included in that specific order. The `@OneToMany` relationship between `CustomerOrder` and `Product` is mapped such that one order can have multiple products. In this setup, we utilize `CascadeType.ALL`, which propagates actions such as persisting or deleting from the order entity to its associated products. Additionally, `orphanRemoval = true` ensures that

products are removed from the database if they are no longer associated with an order, maintaining data integrity.

The `CustomerOrder` entity uses `@ToString.Exclude` on the products field to avoid circular references when converting the object to a string. This is particularly important because the relationship between `CustomerOrder` and `Product` is bidirectional, meaning that `Product` also holds a reference back to `CustomerOrder`. The `FetchType.LAZY` setting delays loading the list of products until it is explicitly accessed, improving performance when an order is retrieved but its products are not immediately needed.

On the other hand, the `Product` entity models the items that are part of a customer's order. Each `Product` has basic attributes like `id`, `name`, `description`, and `price`. The relationship to `CustomerOrder` is defined using a `@ManyToOne` annotation, meaning many products can be linked to one order. The `@JoinColumn` annotation defines the foreign key in the database that links a product to a specific order, and `@JsonIgnore` prevents the serialization of the `customerOrder` field to avoid recursive references during JSON conversion.

In both entities, Lombok's `@Data` annotation is used to automatically generate boilerplate code like getters, setters, and `toString()` methods. Lombok's `@NoArgsConstructor` also ensures that each entity has a default constructor, which is required by many Java frameworks like JPA for entity instantiation. Lastly, `CustomerOrder` implements `Serializable` to ensure that it can be safely transmitted in messaging systems like RabbitMQ, especially when converting the order into bytes for network communication.

Repositories

```java
package com.java.FinallProjectJava.repository;

import org.springframework.data.jpa.repository.JpaRepository;

@Repository
public interface OrderRepository extends JpaRepository<CustomerOrder, Long> {
    // JpaRepository provides basic CRUD and query operations for CustomerOrder
    // Extending JpaRepository defines this as a repository for CustomerOrder
    // Long specifies the type of the entity's primary key
}
```

```java
@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {
    // JpaRepository provides basic CRUD and query operations for Product
    // Extending JpaRepository defines this as a repository for Product
    // Long specifies the type of the entity's primary key
}
```

Next, we define two repository interfaces, OrderRepository and ProductRepository, which are responsible for interacting with the database to manage the CustomerOrder and Product entities, respectively. Both repositories extend JpaRepository, a Spring Data interface that provides a wide range of pre-defined methods for performing basic CRUD (Create, Read, Update, Delete) operations without needing to write any SQL code or custom queries manually.

The OrderRepository is designed to manage the CustomerOrder entity, while the ProductRepository handles the Product entity. By extending JpaRepository<CustomerOrder, Long> and JpaRepository<Product, Long>, both repositories automatically inherit methods such as findAll(), findById(), save(), and deleteById(). The Long type in both interfaces specifies the data type of the primary key for these entities, which is the id field.

By marking these interfaces with the @Repository annotation, Spring automatically detects them during component scanning and creates the necessary implementations at runtime. This annotation also indicates that these interfaces are part of the persistence layer, helping to define a clear separation between the database logic and the rest of the application. Through these repository interfaces, the service layer can easily access and manipulate order and product data in the database without needing to know the underlying database operations.

Controller

```java
21
22 @RestController
23 @RequestMapping("/api/orders")
24 @RequiredArgsConstructor
25 @Slf4j
26 public class OrderController {
27
28     // Injects services and message sender
29     private final OrderService orderService;
30     private final OrderMessageSender messageSender;
31
32     @GetMapping
33     public List<CustomerOrder> getAllOrders() {
34         // Returns all customer orders
35         return orderService.getAllOrders();
36     }
37
38     @GetMapping("/{id}")
39     public ResponseEntity<CustomerOrder> getOrderById(@PathVariable Long id) {
40         // Retrieves order by ID, handles not found
41         Optional<CustomerOrder> order = orderService.getOrderById(id);
42         return order.map(ResponseEntity::ok)
43                 .orElse(ResponseEntity.notFound().build());
44     }
45
46     @PostMapping
47     public CustomerOrder createOrder(@RequestBody CustomerOrder customerOrder) {
48         // Saves a new customer order
49         CustomerOrder savedOrder = orderService.saveOrder(customerOrder);
50
51         try {
52             // Sends order to RabbitMQ
53             messageSender.sendOrder(savedOrder);
54         } catch (Exception e) {
55             // Logs error if RabbitMQ fails
56             Log.error("Failed to send order to RabbitMQ", e);
57         }
58
59         return savedOrder;
60     }
61
62     @DeleteMapping("/{id}")
63     public ResponseEntity<Void> deleteOrder(@PathVariable Long id) {
64         // Deletes order by ID
65         orderService.deleteOrder(id);
66         return ResponseEntity.noContent().build();
67     }
68 }
69
70
```

The OrderController class is responsible for handling HTTP requests related to customer orders. It is a REST controller, meaning it exposes endpoints that allow external clients to interact with the system. The class defines several endpoints, each performing a specific operation on CustomerOrder entities.

The @GetMapping endpoint retrieves a list of all customer orders from the database using the OrderService and returns it as a JSON response. This is a simple operation where all existing orders are fetched and returned to the client.

The second @GetMapping("/{id}") is used to fetch a specific order by its ID. If the order exists, it is returned with an HTTP status of 200 (OK); if not, a 404 (Not Found) response is returned. This ensures proper handling of cases where an order might not exist.

The @PostMapping method handles the creation of new customer orders. It accepts an order in JSON format via the @RequestBody annotation, saves it to the database through the service, and then attempts to send the saved order to RabbitMQ for further processing. If the RabbitMQ message fails to send, an error is logged, but the method still returns the saved order.

Finally, the @DeleteMapping("/{id}") endpoint allows the deletion of a specific order by its ID. It calls the service to delete the order and returns an HTTP 204 (No Content) response to indicate that the operation was successful without returning any data. This structure ensures that all CRUD operations (Create, Read, Update, Delete) are available for CustomerOrder through clean and straightforward endpoints.

Service

```java
15  @Service
16  @RequiredArgsConstructor
17  @Slf4j
18  public class OrderService {
19
20      // Injects the OrderRepository for database access
21      private final OrderRepository orderRepository;
22
23      public List<CustomerOrder> getAllOrders() {
24          // Fetches all customer orders from the database
25          return orderRepository.findAll();
26      }
27
28      public Optional<CustomerOrder> getOrderById(Long id) {
29          // Fetches a specific customer order by its ID
30          return orderRepository.findById(id);
31      }
32
33      public CustomerOrder saveOrder(CustomerOrder customerOrder) {
34          // Logs the process of saving the order
35          Log.info("Saving order for customer: {}", customerOrder.getCustomerName());
36
37          // Associates products with the order if they exist
38          List<Product> products = customerOrder.getProducts();
39          if (products != null) {
40              Log.info("Products before association: {}", products);
41              for (Product product : products) {
42                  // Associates each product with the order
43                  Log.info("Associating product {} with order {}", product.getName(), customerOrder.getCustomerName());
44                  product.setCustomerOrder(customerOrder);
45              }
46          } else {
47              // Logs a warning if no products are found
48              Log.warn("No products found in the order");
49          }
50
51          // Saves the order and returns the saved instance
52          CustomerOrder savedOrder = orderRepository.save(customerOrder);
53          return savedOrder;
54      }
55
56      public void deleteOrder(Long id) {
57          // Deletes a specific order by its ID
58          orderRepository.deleteById(id);
59      }
60  }
```

The OrderService class is responsible for handling the business logic related to customer orders. It interacts with the OrderRepository to access the database and performs operations like fetching, saving, and deleting customer orders.

The method getAllOrders() retrieves all orders stored in the database by calling the findAll() method of the repository. This method returns a list of CustomerOrder entities, representing all the customer orders in the system.

The method getOrderById(Long id) retrieves a specific order by its unique ID. It returns an Optional<CustomerOrder> to safely handle cases where the order might not exist.

The saveOrder(CustomerOrder customerOrder) method handles the logic for saving a new or updated order. It first logs the customer name for whom the order is being saved. If the order contains products, it iterates through the list of products and associates each product with the CustomerOrder by setting the customerOrder field in the Product entity. This ensures that the relationship between orders and products is properly established before the order is saved to the database. If no products are found in the order, a warning is logged. Finally, the method

saves the order using the repository's save() method and returns the saved instance of CustomerOrder.

The deleteOrder(Long id) method deletes an order from the database by its ID. It simply calls the deleteById() method from the repository to perform the operation. This completes the basic CRUD functionality within the service layer, abstracting the database interaction logic from the controller.

Rabbitmq Configuration

In the RabbitMQ configuration, the RabbitMQConfig class sets up the core messaging infrastructure for sending and receiving messages between different services using RabbitMQ. The configuration defines key components such as the queue, exchange, and routing key, all of which are loaded dynamically from the application's properties. The method queue() creates a durable queue, ensuring that messages are not lost if RabbitMQ restarts. The exchange() method creates a TopicExchange, which is responsible for routing messages to queues based on the routing key. The binding() method links the queue to the exchange using the specified routing key.

To ensure that messages can be serialized and deserialized correctly, the Jackson2JsonMessageConverter bean is defined. This enables RabbitMQ to automatically convert Java objects, like CustomerOrder, to JSON when sending messages and back to Java objects when receiving them. The rabbitTemplate() method returns a RabbitTemplate instance, which is the primary interface for sending messages to RabbitMQ. It is configured to use the JSON converter, ensuring seamless integration between RabbitMQ and the application's domain objects.
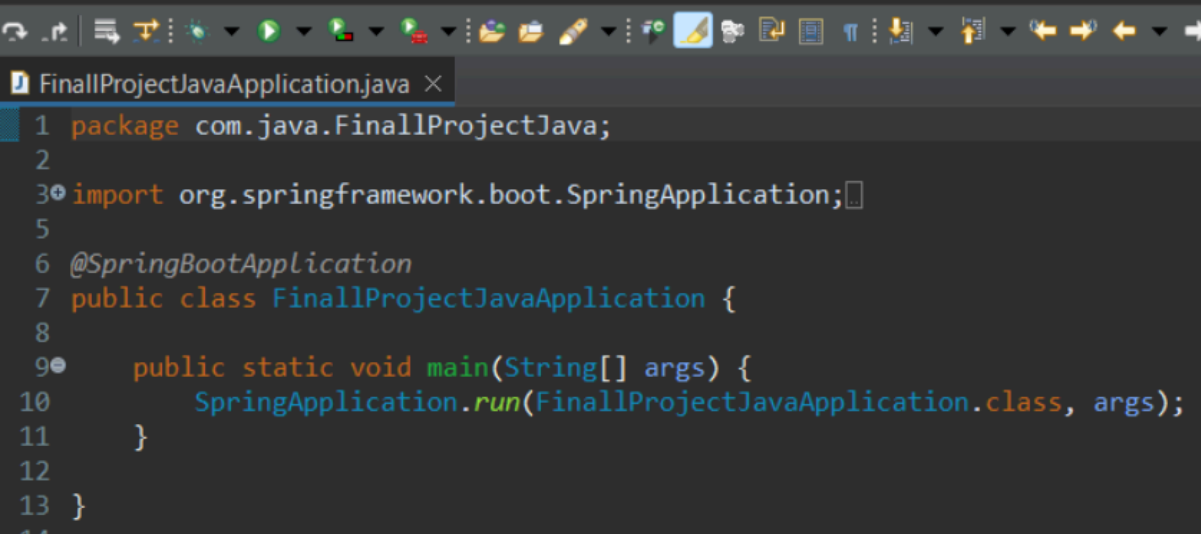
The OrderMessageSender class is responsible for sending messages related to customer orders to RabbitMQ. It uses the RabbitTemplate to convert the CustomerOrder object into a message and then sends it to the RabbitMQ exchange with a specific routing key. The sendOrder() method logs the order being sent and handles the actual communication with RabbitMQ, ensuring that the order is successfully placed in the message queue for processing.

Finally, the OrderMessageListener class listens for incoming messages on the RabbitMQ queue. It is annotated with @RabbitListener, which tells Spring to invoke the receiveOrder() method whenever a message arrives on the specified queue. When a message is received, the listener converts it back into a CustomerOrder object and logs the information, ensuring that the message has been processed correctly. This class is the receiving side of the messaging system, responsible for handling incoming order messages.
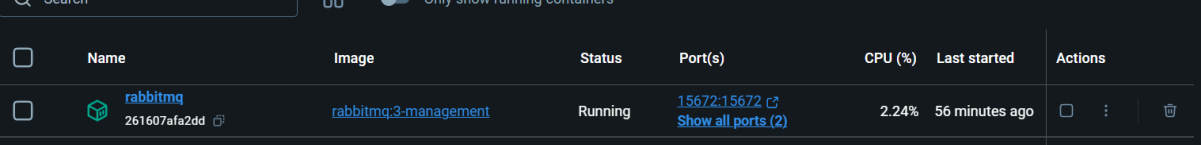
App functionality

First, we run the application from the `main` method. This starts the Spring Boot application and initializes all components, including the configuration for RabbitMQ.
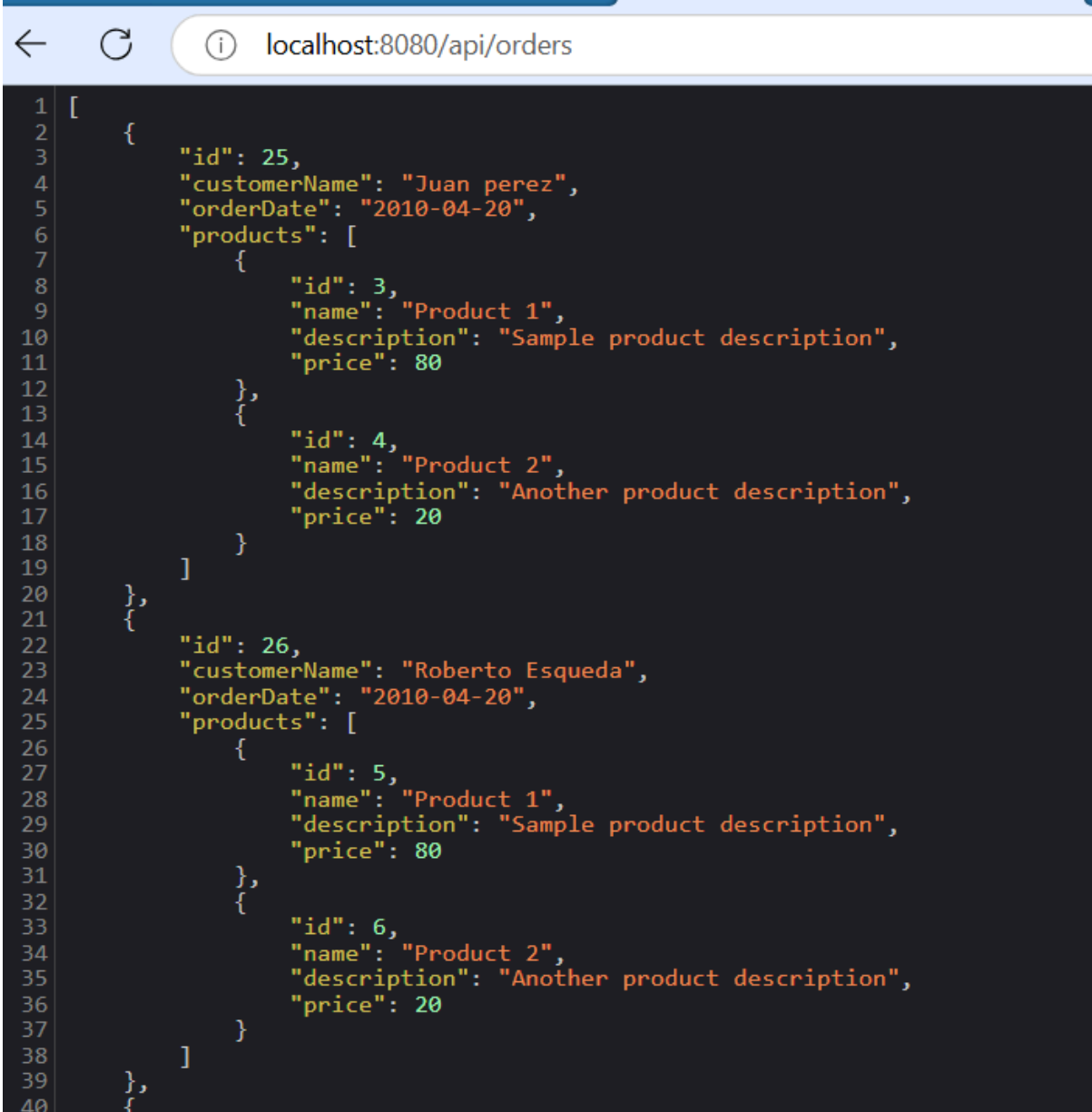


```java
package com.java.FinallProjectJava;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class FinallProjectJavaApplication {

    public static void main(String[] args) {
        SpringApplication.run(FinallProjectJavaApplication.class, args);
    }

}
```

Next, we power up the RabbitMQ container using Docker. This is done through Docker Desktop, where we ensure the RabbitMQ container is up and running.

	Name	Image	Status	Port(s)	CPU (%)	Last started	Actions
☐	rabbitmq 261607afa2dd	rabbitmq:3-management	Running	15672:15672 ↗ Show all ports (2)	2.24%	56 minutes ago	☐ ⋮ 🗑

Once the application and RabbitMQ are both running, we open a browser and navigate to localhost:8080/api/orders to verify that the app is reachable

localhost:8080/api/orders

```
 1  [
 2      {
 3          "id": 25,
 4          "customerName": "Juan perez",
 5          "orderDate": "2010-04-20",
 6          "products": [
 7              {
 8                  "id": 3,
 9                  "name": "Product 1",
10                  "description": "Sample product description",
11                  "price": 80
12              },
13              {
14                  "id": 4,
15                  "name": "Product 2",
16                  "description": "Another product description",
17                  "price": 20
18              }
19          ]
20      },
21      {
22          "id": 26,
23          "customerName": "Roberto Esqueda",
24          "orderDate": "2010-04-20",
25          "products": [
26              {
27                  "id": 5,
28                  "name": "Product 1",
29                  "description": "Sample product description",
30                  "price": 80
31              },
32              {
33                  "id": 6,
34                  "name": "Product 2",
35                  "description": "Another product description",
36                  "price": 20
37              }
38          ]
39      },
40      {
```

Now, we switch to Postman, where we use the POST method to insert data into the application (photo of the POST method in Postman). In this step, we send a customer order along with its associated products, which is processed by the app and passed to RabbitMQ.
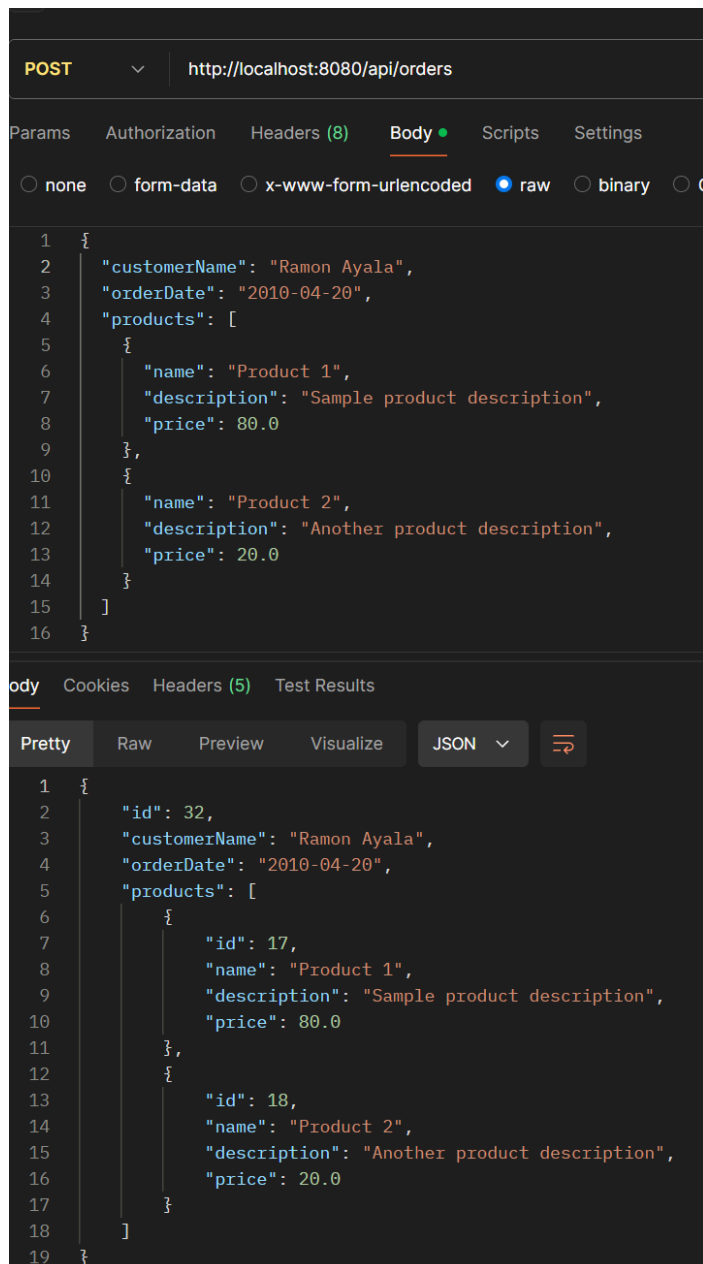
After submitting the data, we can observe several messages in the console, including those from RabbitMQ. These logs indicate that the order has been processed and successfully sent to the message queue

Finally, we can use the GET method in a browser to retrieve and verify the data that was inserted. By visiting the same URL localhost:8080/api/orders, we can see the orders that have been added to the system

```
135         {
136             "id": 32,
137             "customerName": "Ramon Ayala",
138             "orderDate": "2010-04-20",
139             "products": [
140                 {
141                     "id": 17,
142                     "name": "Product 1",
143                     "description": "Sample product description",
144                     "price": 80
145                 },
146                 {
147                     "id": 18,
148                     "name": "Product 2",
149                     "description": "Another product description",
150                     "price": 20
151                 }
152             ]
153         }
154 ]
```