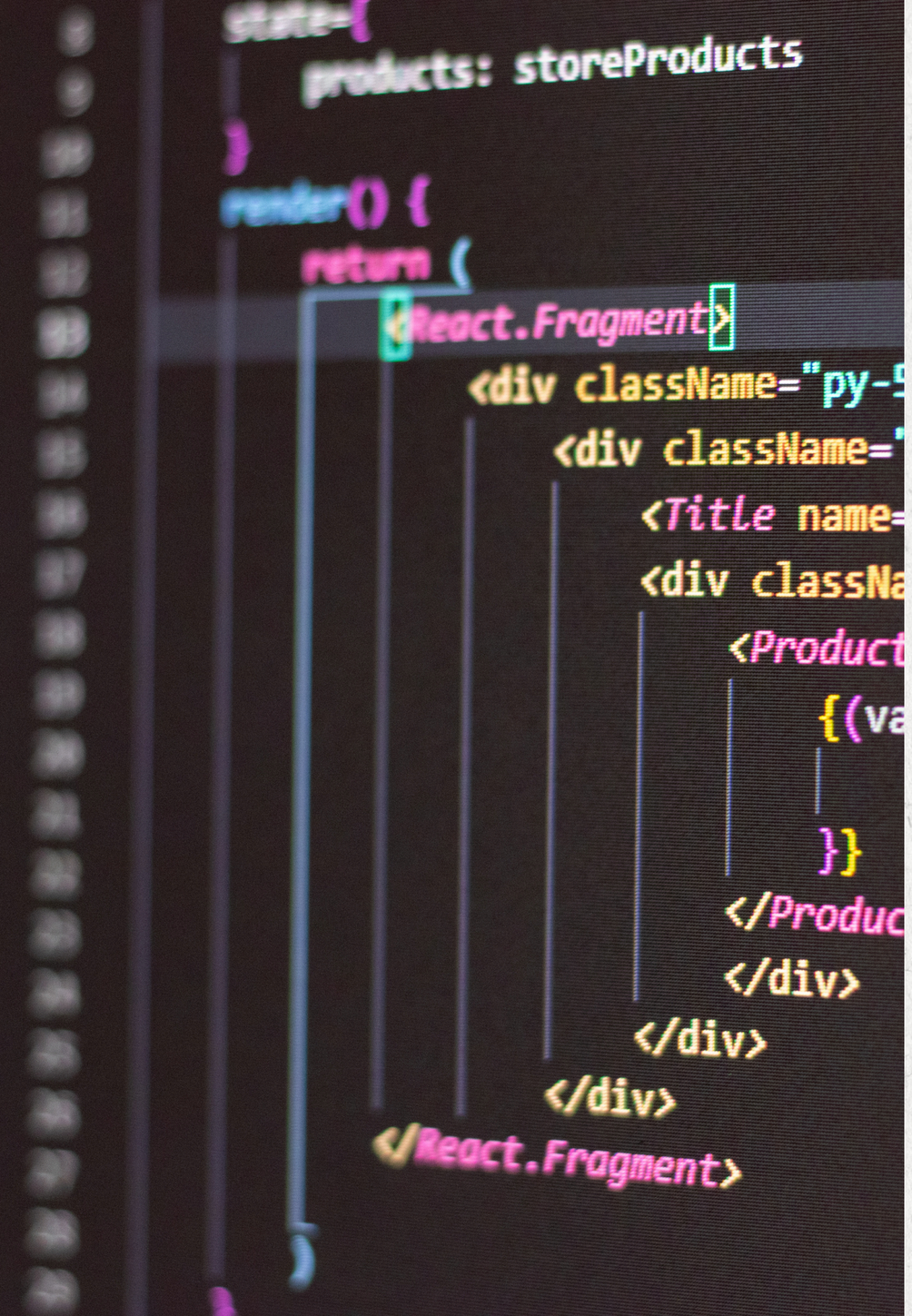
FINAL PROJECT JAVA

BY: ALBERTO ESCAMILLA

JASSO



INTRO

This project is a customer order management application that allows the creation, viewing, and deletion of orders with associated products. Developed with Spring Boot, it exposes a REST API to perform CRUD operations on orders and their products. Additionally, it uses RabbitMQ to manage asynchronous messaging, enabling efficient order processing.

The technologies used include Java, Spring Boot for the API, Spring Data JPA for data persistence, RabbitMQ for messaging, and Docker to manage containers. Postman is employed to test the API. The project demonstrates an effective integration of these technologies to build a scalable and efficient application.

ENTITIES

The main entities of the application are CustomerOrder and Product. CustomerOrder represents an order that includes customer information and a list of products, while Product represents the products associated with that order.

The two entities are related through a OneToMany relationship between the order and the products, and a ManyToOne from the products to the order. This allows for efficient management of order and product data.

```java
package com.java.FinallProjectJava.entity;

import java.io.Serializable;

@Entity // Marks this class as a JPA entity
@Data // Lombok: generates getters, setters, toString, equals, and hashCode
@NoArgsConstructor // Lombok: generates a no-arguments constructor
public class CustomerOrder implements Serializable { // Implements Serializable for message conversion

    private static final long serialVersionUID = 1L; // Ensures compatibility during serialization

    @Id // Marks this field as the primary key
    @GeneratedValue(strategy = GenerationType.IDENTITY) // Auto-generates ID values
    private Long id; // Unique identifier for the order

    @NotNull // Ensures this field cannot be null
    private String customerName; // Customer's name

    private String orderDate; // Date when the order was placed

    @OneToMany(mappedBy = "customerOrder", orphanRemoval = true, cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    // OneToMany relationship: an order can have multiple products
    @ToString.Exclude // Excludes products from toString() to avoid circular reference
    private List<Product> products; // List of products associated with this order
}
```

```java
package com.java.FinallProjectJava.entity;

import com.fasterxml.jackson.annotation.JsonIgnore;

@Entity // Marks this class as a JPA entity
@Data // Lombok: generates getters, setters, toString, equals, and hashCode
@NoArgsConstructor // Lombok: generates a no-arguments constructor
public class Product { // Represents a product entity

    @Id // Marks this field as the primary key
    @GeneratedValue(strategy = GenerationType.IDENTITY) // Auto-generates ID values
    private Long id; // Unique identifier for the product

    private String name; // Product name

    private String description; // Product description

    private double price; // Product price

    @ManyToOne(fetch = FetchType.LAZY) // Many products can be associated with one ord
    @JoinColumn(name ="order_id", nullable =false) // Maps the foreign key to the orde
    @JsonIgnore // Prevents circular references during JSON serialization
    private CustomerOrder customerOrder; // The associated order for this product
}
```

CONTROLLERS

The OrderController exposes REST endpoints to manage orders. It allows retrieving all orders, searching for a specific one by ID, creating new orders, and deleting them. Through these endpoints, users interact with the system to manage order and product data, facilitating communication between the client and the business service.

```
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 }
```

REPOSITORIES

The OrderRepository and ProductRepository are responsible for interacting with the database. Both extend JpaRepository, which provides CRUD operations without the need to implement additional logic.

Spring detects these interfaces with the @Repository annotation, automatically generating the necessary implementations to manage the entities.

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

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

@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
}
```

SERVICE

```java
package com.java.FinallProjectJava.controllers;

import java.util.List;

@RestController
@RequestMapping("/api/orders")
@RequiredArgsConstructor
@Slf4j
public class OrderController {

    // Injects services and message sender
    private final OrderService orderService;
    private final OrderMessageSender messageSender;

    @GetMapping
    public List<CustomerOrder> getAllOrders() {
        // Returns all customer orders
        return orderService.getAllOrders();
    }

    @GetMapping("/{id}")
    public ResponseEntity<CustomerOrder> getOrderById(@PathVariable Long id) {
        // Retrieves order by ID, handles not found
        Optional<CustomerOrder> order = orderService.getOrderById(id);
        return order.map(ResponseEntity::ok)
                    .orElse(ResponseEntity.notFound().build());
    }

    @PostMapping
    public CustomerOrder createOrder(@RequestBody CustomerOrder customerOrder) {
        // Saves a new customer order
        CustomerOrder savedOrder = orderService.saveOrder(customerOrder);

        try {
            // Sends order to RabbitMQ
            messageSender.sendOrder(savedOrder);
        } catch (Exception e) {
            // Logs error if RabbitMQ fails
            log.error("Failed to send order to RabbitMQ", e);
        }

        return savedOrder;
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteOrder(@PathVariable Long id) {
        // Deletes order by ID
        orderService.deleteOrder(id);
        return ResponseEntity.noContent().build();
    }
}
```

The OrderService handles the business logic for orders. It provides methods to save, search, and delete orders, as well as associate products with orders.

It also sends the orders to RabbitMQ for processing. In this way, the service centralizes the main operations of the application, separating the logic from data access.

RABBIT MQ CONFIG

The RabbitMQ configuration defines how the messaging system is managed, creating the queue, exchange, and routing key.

It also configures a JSON message converter to facilitate communication between RabbitMQ and the application. Messages are correctly routed and processed thanks to this configuration.

```java
package com.java.FinallProjectJava.Rabbit;

import org.springframework.amqp.core.Binding;

@Configuration
public class RabbitMQConfig {

    // Queue name, loaded from properties
    @Value("${order.queue}")
    private String queueName;

    // Exchange name, loaded from properties
    @Value("${order.exchange}")
    private String exchangeName;

    // Routing key, loaded from properties
    @Value("${order.routingkey}")
    private String routingKey;

    @Bean
    public Queue queue() {
        // Defines a durable RabbitMQ queue
        return new Queue(queueName, true);
    }

    @Bean
    public TopicExchange exchange() {
        // Defines a RabbitMQ topic exchange
        return new TopicExchange(exchangeName);
    }

    @Bean
    public Binding binding(Queue queue, TopicExchange exchange) {
        // Binds the queue to the exchange using the routing key
        return BindingBuilder.bind(queue).to(exchange).with(routingKey);
    }

    @Bean
    public Jackson2JsonMessageConverter jsonMessageConverter() {
        // Configures a JSON message converter for RabbitMQ
        return new Jackson2JsonMessageConverter();
    }

    @Bean
    public RabbitTemplate rabbitTemplate(ConnectionFactory connectionFactory) {
        // Configures RabbitTemplate with JSON message converter
        RabbitTemplate template = new RabbitTemplate(connectionFactory);
        template.setMessageConverter(jsonMessageConverter());
        return template;
    }
}
```

RABBITMQ SENDER Y LISTENER

```java
package com.java.FinallProjectJava.Rabbit;

import org.springframework.amqp.rabbit.core.RabbitTemplate;

@Component
@RequiredArgsConstructor
@Slf4j
public class OrderMessageSender {

    // Injects RabbitTemplate for message sending
    private final RabbitTemplate rabbitTemplate;

    // Routing key, loaded from properties
    @Value("${order.routingkey}")
    private String routingKey;

    // Exchange name, loaded from properties
    @Value("${order.exchange}")
    private String exchange;

    public void sendOrder(CustomerOrder order) {
        // Logs and sends order to RabbitMQ
        log.info("Sending order message: {}", order);
        rabbitTemplate.convertAndSend(exchange, routingKey, order);
        log.info("Order message sent: {}", order);
    }
}
```

```java
package com.java.FinallProjectJava.Rabbit;

import org.springframework.amqp.rabbit.annotation.RabbitListener;

@Component
@Slf4j
public class OrderMessageListener {

    @RabbitListener(queues = "${order.queue}")
    public void receiveOrder(CustomerOrder order) {
        // Logs the received order from RabbitMQ
        log.info("Received order message: {}", order);
    }
}
```

The OrderMessageSender sends orders to RabbitMQ to be processed by other components, while the OrderMessageListener listens to and processes the messages arriving in the queue.

Together, they enable asynchronous communication, ensuring that orders are handled efficiently within the system

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.

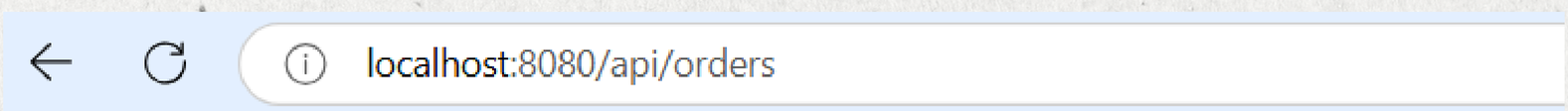
```java
package com.java.FinallProjectJava;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class FinallProjectJavaApplication {

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

}
```

FUNCTIONALITY

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.

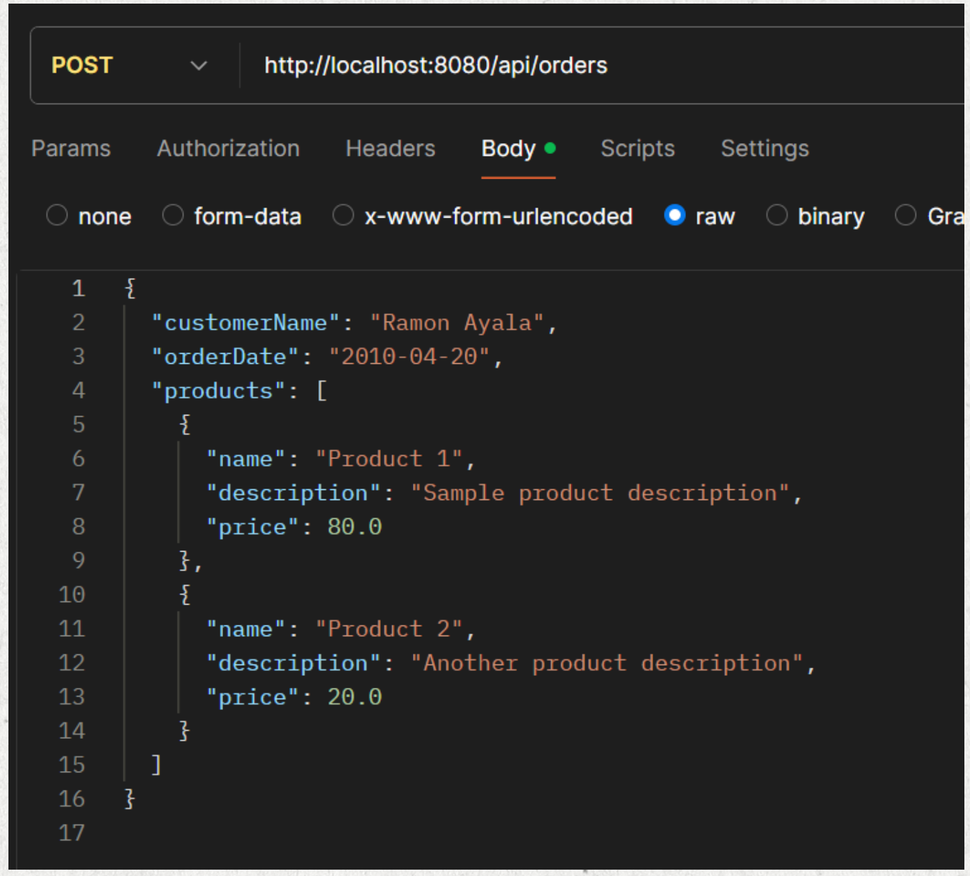
Name	Image	Status	Port(s)	CPU (%)	Last started	Actions
rabbitmq 261607afa2	rabbitmq:3-management	Running	15672:15672 Show all ports (2)	2.55%	12 minutes ago	

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



FUNCTIONALITY

Now, we switch to Postman, where we use the POST method to insert data into the application. In this step, we send a customer order along with its associated products, which is processed by the app and passed to RabbitMQ.



FUNCTIONALITY

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

```
2024-09-17T13:09:32.866-06:00  INFO 8512 --- [FinallProjectJava] [nio-8080-exec-5] c.j.F.Rabbit.OrderMessageSender          : Sending order message: CustomerOrder(id=33, customerName=Ramon Ayala, orderDate=2020-04-20)
2024-09-17T13:09:32.925-06:00  INFO 8512 --- [FinallProjectJava] [nio-8080-exec-5] c.j.F.Rabbit.OrderMessageSender          : Order message sent: CustomerOrder(id=33, customerName=Ramon Ayala, orderDate=2020-04-20)
2024-09-17T13:09:33.012-06:00  INFO 8512 --- [FinallProjectJava] [ntContainer#0-1] c.j.F.Rabbit.OrderMessageListener       : Received order message: CustomerOrder(id=33, customerName=Ramon Ayala, orderDate=2020-04-20)
```

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


```
    {
        "id": 32,
        "customerName": "Ramon Ayala",
        "orderDate": "2010-04-20",
        "products": [
            {
                "id": 17,
                "name": "Product 1",
                "description": "Sample product description",
                "price": 80
            },
            {
                "id": 18,
                "name": "Product 2",
                "description": "Another product description",
                "price": 20
            }
        ]
    }
]
```

CONCLUSIONS

The application demonstrates a successful integration of Spring Boot with RabbitMQ to handle orders asynchronously.

Spring Data JPA simplifies data persistence management and CRUD operations.

The use of Docker facilitates the deployment and management of services such as RabbitMQ.

The architecture is scalable and allows for easy adaptation to new functionalities.