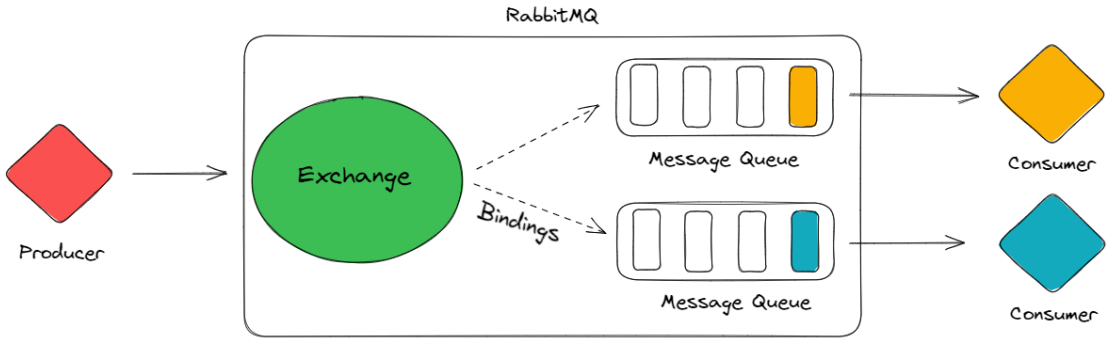
Microservices & RabbitMQ lab Report

Amo Samuel

Brief Overview of RabbitMQ

RabbitMQ is a widely-used messaging broker that allows applications to communicate asynchronously and decouples processes through message queues. Its architecture is built around the following core concepts.



Key Components:

1. **Producer**: The application that sends the message. It does not communicate directly with the consumer but instead with an **Exchange**.
2. **Exchange**: This component receives the message from the producer and determines which queue(s) to route the message to, based on routing rules and bindings.
3. **Bindings**: These are the rules that tell the exchange which queue(s) to send the message to.
4. **Message Queue**: A place where messages are stored until they are consumed by the consumer. The queue is independent of both the producer and the consumer, providing reliability and asynchronous messaging.
5. **Consumer**: The application that listens for messages from the queue, processes them, and can save or perform any logic with the message.

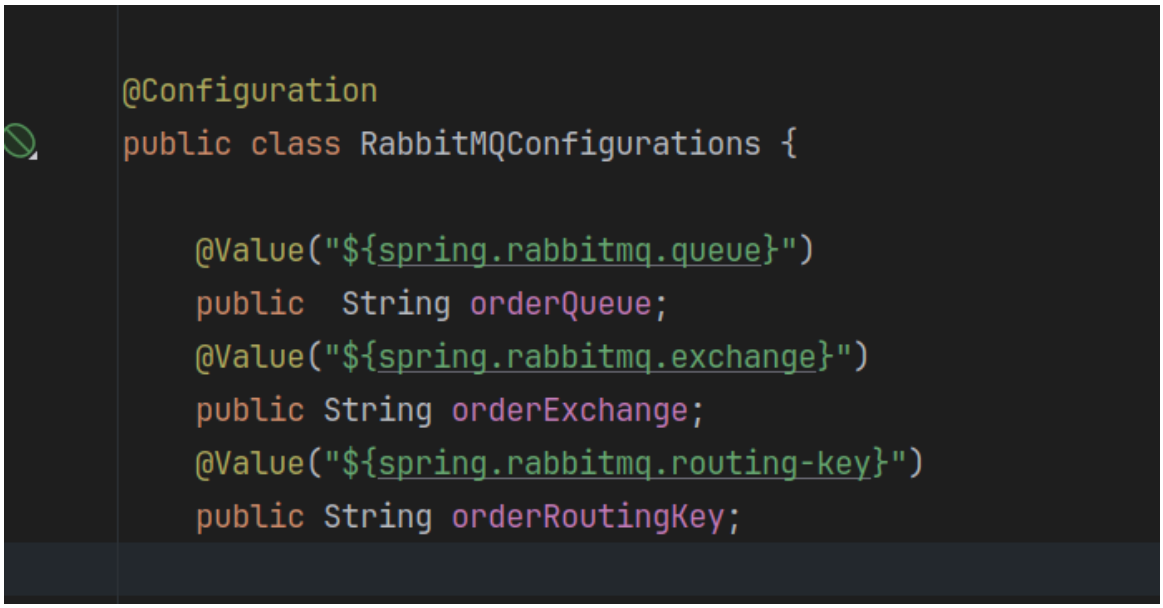
RabbitMQ acts as the intermediary, allowing these components to interact while remaining decoupled, which increases the scalability and flexibility of the system.

My RabbitMQ Configuration in Code

In my project, I configured a Spring Boot application to use RabbitMQ for order processing. Below is how I went about implementing RabbitMQ's messaging architecture using Spring AMQP.

1. Defining the Configuration:

In the RabbitMQConfigurations class, I used Spring's @Configuration to define essential RabbitMQ components such as the queue, exchange, and bindings. The configuration details were fetched from application properties using @Value.



```
@Configuration
public class RabbitMQConfigurations {

    @Value("${spring.rabbitmq.queue}")
    public  String orderQueue;
    @Value("${spring.rabbitmq.exchange}")
    public String orderExchange;
    @Value("${spring.rabbitmq.routing-key}")
    public String orderRoutingKey;
```

- **Queue**: I declared the order queue using the Queue class.
- **Exchange**: The TopicExchange was set up to route the messages based on a routing key.
- **Binding**: This connects the queue to the exchange using the routing key provided.

The binding ensures that any messages sent to the exchange with the correct routing key are delivered to the intended queue

2. Message Conversion and Sending:

To send messages in JSON format, I used the Jackson2JsonMessageConverter to convert the message before sending it to the RabbitMQ exchange.

The RabbitTemplate was configured to use this message converter to simplify message processing between the producer and consumer.

```
@Bean
public RabbitTemplate rabbitTemplate(ConnectionFactory connectionFactory) {
    RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);
    rabbitTemplate.setMessageConverter(jsonMessageConverter());
    return rabbitTemplate;
}

@Bean
public Jackson2JsonMessageConverter jsonMessageConverter() {
    return new Jackson2JsonMessageConverter();
}
```

3. OrderProducer - Sending Messages:

The OrderProducer is responsible for sending messages (orders) to RabbitMQ. I used the RabbitTemplate to send the message to the exchange with the appropriate routing key.

```
    @PostMapping
    public String sendOrder(@RequestBody Order order) {
        if (order == null || order.getOrderId() == null) {
            throw new OrderProcessingException("Invalid Order Data");
        }
        rabbitTemplate.convertAndSend(rabbitMQConfigurations.orderExchange, rabbitMQConfigurations.orderRoutingKey, order);
        return "Order sent successfully";
    }
}
```

The order is validated before sending, and any issues (such as missing order IDs) trigger a custom exception (OrderProcessingException), ensuring the system handles invalid data.

OrderConsumer - Receiving Messages:

The OrderConsumer class listens to the queue for incoming messages using the @RabbitListener annotation. Once a message is received, it is logged and saved to the database using the OrderRepository.

```
    }

    @RabbitListener(queues = "${spring.rabbitmq.queue}")
@   public void processOrder(Order order) {
        logger.info("Order received and saving to database {}", order.toString());

        orderRepository.save(order);
    }
}
```

The consumer logs the received order and processes it by saving the data into the database. This method demonstrates the asynchronous nature of the message broker, as it allows orders to be processed independently of the producer.

5. Error Handling:

To ensure that the application remains stable in case of errors, I implemented a global exception handler. The GlobalExceptionHandler captures any exceptions that occur while processing an order and provides a detailed error response to the client.

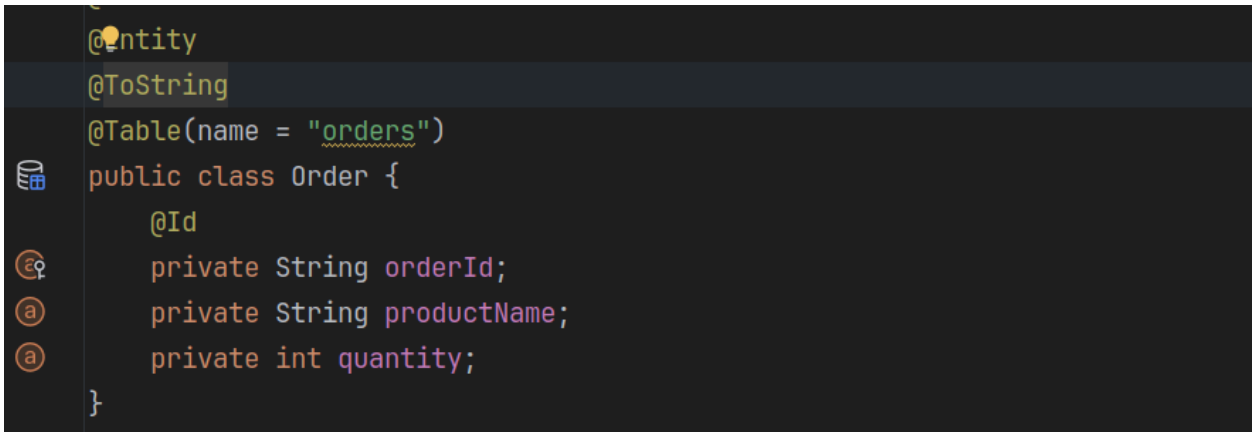
```
private final Logger logger = LoggerFactory.getLogger(GlobalExceptionHandler.class);  2 usages

@ExceptionHandler(OrderProcessingException.class)
public ResponseEntity<String> handleOrderProcessingException(OrderProcessingException processingException) {
    logger.error("Error Processing Order {}", processingException.getMessage());
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(processingException.getMessage());
}
```

With this setup, the application gracefully handles order processing exceptions, logging errors, and returning informative error messages without crashing.

6. Data Persistence:

The Order entity is annotated with JPA annotations, and the OrderRepository is responsible for saving orders to the database. This separation of concerns ensures that the producer and consumer deal only with messaging, while the repository handles data persistence.



Conclusion:

Using RabbitMQ, I was able to decouple the producer and consumer of orders, allowing for scalable and fault-tolerant order processing. The architecture ensures that messages can be processed asynchronously, with a strong focus on reliable error handling and logging. The system is now capable of efficiently processing orders, regardless of the speed at which they are produced or consumed.