# ****E-Commerce Order Management System****

## ****Project Overview****

The **E-Commerce Order Management System** allows users to place, update, and cancel orders in an online e-commerce store. The system is built using **Java**, **Spring Boot**, Micro-services **Architecture**, and **Messaging Queues** (Kafka or RabbitMQ). It includes features like managing orders, products, and customers, with efficient communication between different services via REST APIs and asynchronous messaging.

## ****Technologies Used****

* **Java 11**
* **Spring Boot** for developing RESTful APIs
* **Micro-services Architecture** for decoupling services
* **Kafka/RabbitMQ** for messaging queues
* **Redis** for caching frequently accessed data
* **MySQL** for database storage
* **JUnit & Mockito** for unit and integration testing
* **Maven** for project management and dependency management

## ****Project Structure****

### ****Microservices****

The system is divided into multiple micro-services for different domains:

1. **Order Service**: Manages all order-related functionalities (creating, updating, and canceling orders).
2. **Product Service**: Manages product details and availability.
3. **Customer Service**: Manages customer information and interactions.

### ****Communication Between Services****

* **REST APIs**: Each microservice exposes REST APIs to communicate with other services.
* **Messaging Queues (Kafka/RabbitMQ)**: Asynchronous communication is handled using Kafka or RabbitMQ for tasks like order status updates and notifications.

## ****System Design****

### ****1. Order Module****

The **Order** module is responsible for managing the orders placed by customers. An order contains the following fields:

* **orderId**: Unique identifier for the order.
* **productId**: Product associated with the order.
* **customerId**: Customer who placed the order.
* **orderDate**: Date when the order was placed.
* **status**: Current status of the order (e.g., NEW, PROCESSING, SHIPPED, DELIVERED).
* **totalAmount**: Total cost of the order.

#### ****APIs for Order Management****

**Create Order**

* + **Method**: POST /orders/place(http://localhost:8081/api/orders/place)
  + **Description**: Create a new order with a list of products, customer details, and payment method.
  + **Request Body**:

{

  "productId": 1,

  "quantity": 500,

  "customerId": 2,

  "orderDate": "2024-12-16T12:00:00",

  "status": "PENDING",

  "totalAmount": 250.00

}

* + **Response**:

{

    "orderId": 99,

    "productId": 1,

    "quantity": 500,

    "customerId": 2,

    "orderDate": "2024-12-19T01:29:45.6459872",

    "status": "PENDING",

    "price": 70000.5,

    "totalAmount": 3.500025E7

}

**Update Order Status**

* + **Method**: PUT /orders/{orderId}(http://localhost:8081/api/orders/88)
  + **Description**: Update the status of an existing order (e.g., from NEW to PROCESSING).
  + **Request Body**: {

  "productId": 3,

  "customerId": 2,

  "orderDate": "2024-12-16T12:00:00",

  "status": "PROCESSING",

  "totalAmount": 250.00

}

* + **Response**:

{

    "orderId": 88,

    "productId": 1,

    "quantity": 500,

    "customerId": 2,

    "orderDate": "2024-12-18T21:15:53.130461",

    "status": "PROCESSING",

    "price": 70000.5,

    "totalAmount": 3.500025E7

}

**Cancel Order**

* + **Method**: DELETE /orders/{orderId}(http://localhost:8081/api/orders/89)
  + **Description**: Cancel an order.
  + **Response**:

Order cancelled successfully.

### ****For customer,product and Order APIs, refer Postman logs file.****

### ****2. Microservices Architecture****

Each service is developed independently with its own database and exposed RESTful APIs. These services interact with each other using **HTTP** and **Kafka** (or **RabbitMQ**) for asynchronous operations.

* **Order Service** communicates with the **Product Service** to check the availability of products and update the inventory.
* **Order Service** also communicates with the **Customer Service** to fetch customer details.
* **Kafka/RabbitMQ** is used for sending asynchronous messages for order status updates.

### ****3. Messaging Queue (Kafka/RabbitMQ)****

* **Kafka** or **RabbitMQ** is integrated to handle order notifications and status updates asynchronously.
* **Order Status Update**: When an order's status changes (e.g., from NEW to SHIPPED), a message is sent to a messaging queue. Other services or external systems can subscribe to the queue to perform further actions (e.g., email notifications or inventory updates).

### ****4. Database & Caching****

* **Database (MySQL)**: Each microservice manages its own database. The **Order Service** uses MySQL to store order-related information.
* **Redis**: Redis is used for caching frequently accessed data (e.g., product details) to enhance performance and reduce database load.

### ****5. Unit & Integration Testing****

* **JUnit** is used for writing unit tests for each service.
* **Mockito** is used for mocking dependencies in tests to isolate services.
* **Integration Testing** ensures that the services communicate correctly and that APIs are functioning as expected.

Starting zookeeper-server:

C:\kafka\_2.13-3.9.0\bin\windows>zookeeper-server-start.bat C:\kafka\_2.13-3.9.0\config\zookeeper.properties

Starting kafka-server:

C:\kafka\_2.13-3.9.0\bin\windows>kafka-server-start.bat C:\kafka\_2.13-3.9.0\config\server.properties

Checking Updates on Kafka:

C:\kafka\_2.13-3.9.0\bin\windows>kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic order-status --from-beginning:

Starting redis-server:

C:\Redis-x64-3.0.504\redis-server.exe /path/to/redis.conf