

JavaEE2024-WMS-stage3

文档摘要（强烈建议使用飞书链接查看文档 [📖 JavaEE2024-WMS-stage3](#)）



我们完成了当前阶段（第三阶段）中的所有basic和credit要求（详见要求文档）。

我们将Assignment03中对于前面实现的逻辑结构进行了改进，使得符合本次作业的更复杂的功能需求，包括但不限于物流、仓库及其相关操作等，为了方便体现kafka的功能，我们将建模后得到的五个状态流程包装为了五个接口（对非事件接口进行了简化，非事件接口参考先前的文档stage1与stage2），并通过websocket以及postman对状态流程结果进行可视化。

代码已上传至GitHub，<https://github.com/Brony01/JavaEE03>。

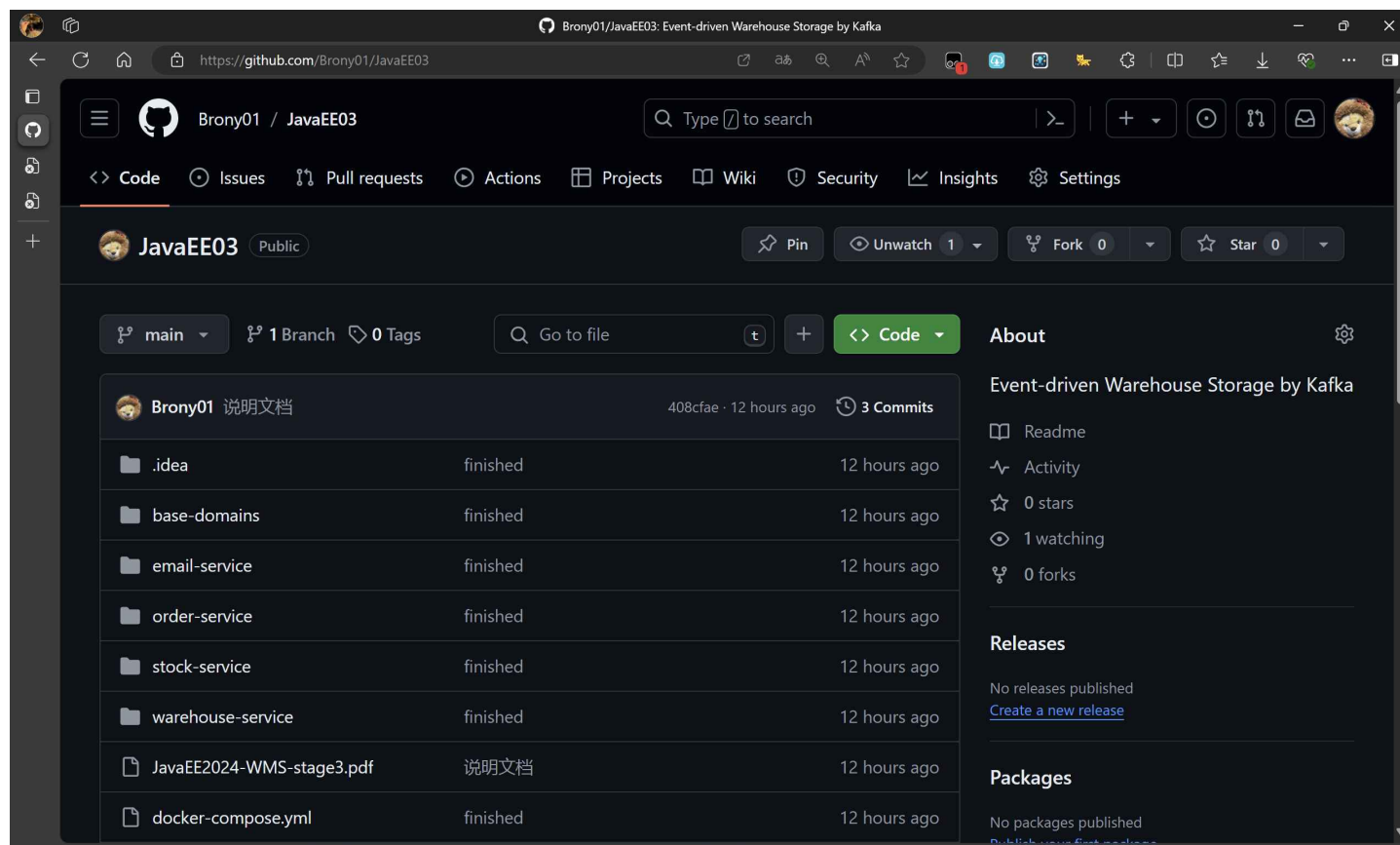
其余阶段文档的跳转链接

[📖 JavaEE2024-WMS-stage1](#)

[📖 JavaEE2024-WMS-stage2](#)

[📖 JavaEE2024-WMS-stage3](#)

[📖 JavaEE2024-WMS-stage4](#)



一、概述

本文档描述了kafka事件驱动系统的demo的关键实现过程及其结果，我们按照Assignment03的要求逐个进行了实现。

我们对文档中的要求进行了总结与建模，我们针对每个流程进行详细的分析与设计，主要形成了五个状态流程。

该项目旨在开发一个模拟从物流方接收货物到仓库存储的过程的系统。该过程包括以下状态和流程：

1. 物流卸货
2. 检查和记录
3. 仓库存储
4. 库存管理
5. 出货装载

我们采用事件驱动架构设计系统，使用Kafka作为事件通道，并通过WebSocket将模拟结果推送到前端。

这五个状态流程接口通过postman测试结果，以物流卸货为例子，其余内容详见[文末结果](#)：

POST

http://localhost:8080/api/v1/warehouses/logistics?eventId=12345&shipmentId=54321&status=UNLOADING

Send

Params

Authorization

Headers (8)

Body

Pre-request Script

Tests

Settings

Cookies

Query Params

<input checked="" type="checkbox"/>	Key	Value	Description	...	Bulk Edit
<input checked="" type="checkbox"/>	eventId	12345			
<input checked="" type="checkbox"/>	shipmentId	54321			
<input checked="" type="checkbox"/>	status	UNLOADING			
<input type="checkbox"/>	Key	Value	Description		

Body

Cookies

Headers (5)

Test Results

Status: 200 OK

Time: 16 ms

Size: 270 B

Save as example

...

Pretty

Raw

Preview

Visualize

JSON

...

1

2

3

"message": "Logistics event processed: LogisticsEvent(eventId=12345, shipmentId=54321, status=UNLOADING)"

POST

/api/v1/warehouses/logistics 处理物流事件

Try it out

Parameters

Name	Description
eventId <small>required</small> integer(\$int32) (query)	eventId
shipmentId <small>required</small> integer(\$int32) (query)	shipmentId
status <small>required</small> string (query)	status

Responses

Code	Description	Links
200	OK	No links

Media type

/

Controls Accept header.

Example Value | Schema

{
 "additionalProp1": "string",
 "additionalProp2": "string",
 "additionalProp3": "string"
}

这五个状态流程分别在控制台的输出和通过WebSocket的前端输出分别如下：

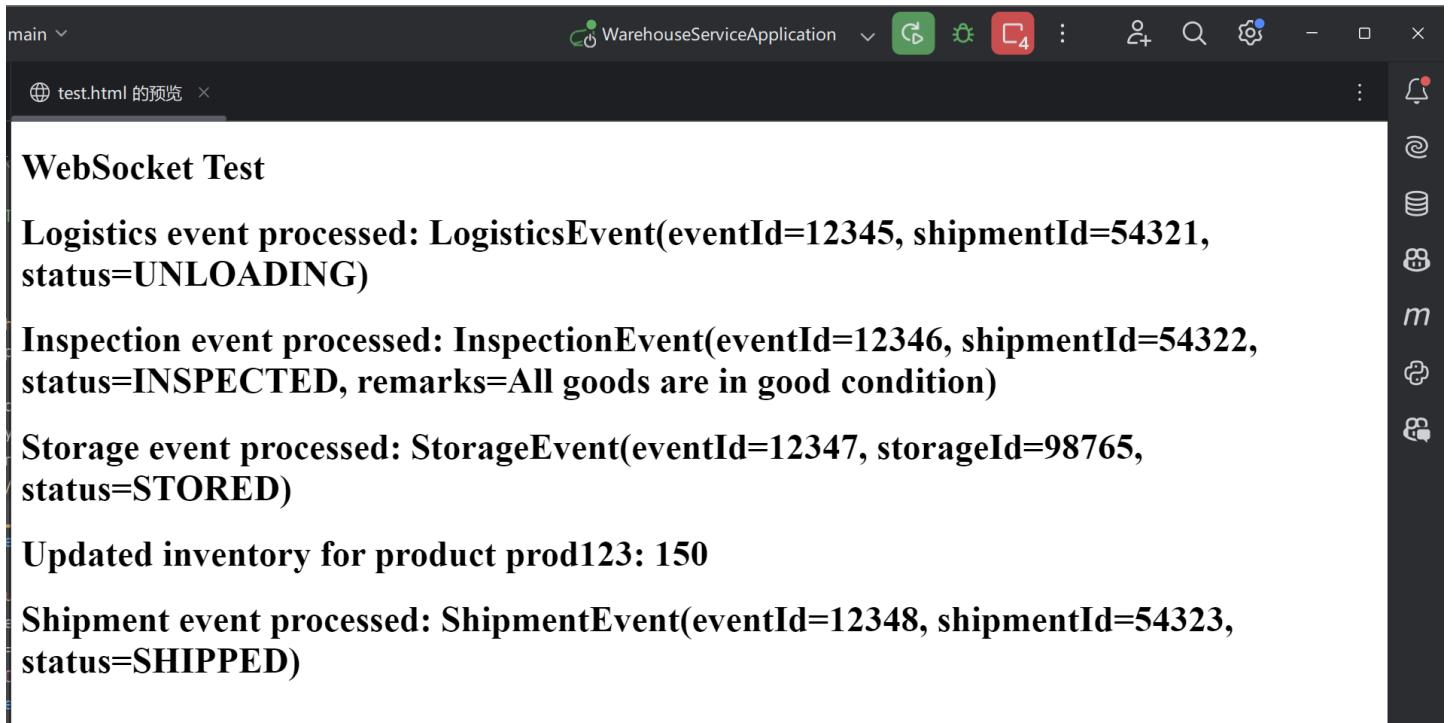
Handling logistics event: LogisticsEvent(eventId=12345, shipmentId=54321, status=UNLOADING)

Handling inspection event: InspectionEvent(eventId=12346, shipmentId=54322, status=INSPECTED, remarks=All goods are in good condition)

Handling storage event: StorageEvent(eventId=12347, storageId=98765, status=STORED)

Updated inventory for product prod123: 150

Handling shipment event: ShipmentEvent(eventId=12348, shipmentId=54323, status=SHIPPED)



二、技术栈及实现

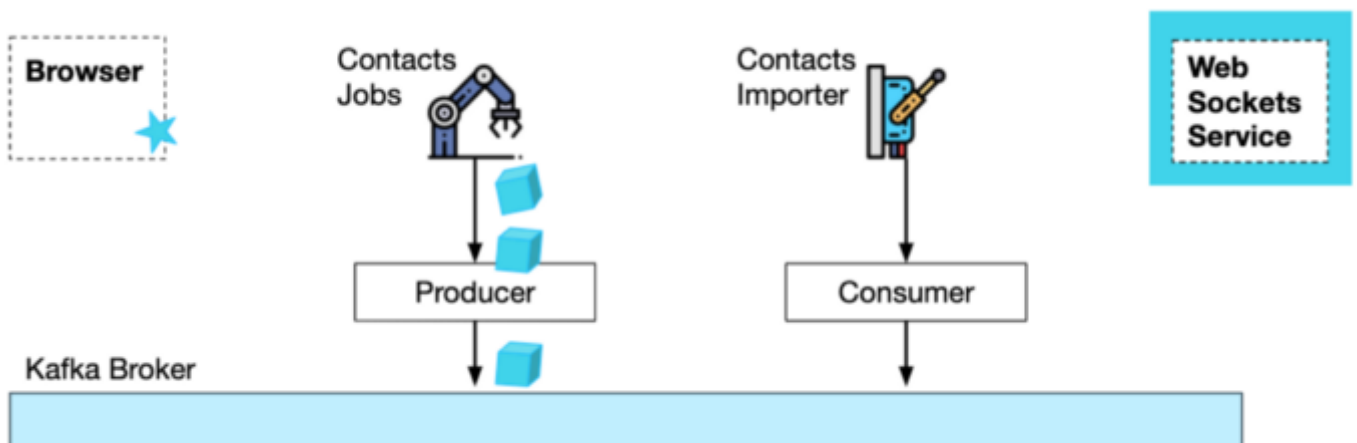
2.1 Kfaka

Kafka 在微服务架构中充当通信骨干，不同的服务通过事件进行通信。每个微服务都可以充当事件生产者或消费者，从而支持松散耦合和可扩展的架构。Kafka 确保可靠和异步的事件交付，使服务能够独立运行，并以自己的速度处理事件。这个用例有助于构建可扩展和解耦的系统，在基于微服务的应用程序中实现敏捷性和自主性。

下面我们将说明此技术栈的简介以及项目中对此的实现细节。

2.1.1 技术栈简介

Kafka 是一种分布式流处理平台，最初由LinkedIn开发，并在2011年开源，随后成为Apache项目。Kafka 主要用于实时数据流处理，数据管道和事件流处理。它的核心概念包括生产者、消费者、主题和分区。



- Topic主题

Kafka中的数据以主题为单位进行分类，每个主题可以看作是一个日志。生产者将数据发布到主题中，消费者从主题中订阅和消费数据。

- Producer生产者

生产者是产生数据的客户端，负责将数据发送到Kafka的主题中。它们可以决定将数据发送到哪个分区，以实现负载均衡和更高效的数据存储和处理。

- Customer消费者

消费者是读取和处理数据的客户端，从指定的主题中消费数据。一组消费者可以组成一个消费组(Consumer Group)，每个消费组中的消费者分工合作，消费主题中的数据。

- Partition分区

每个主题可以分为多个分区，每个分区是一个有序的、不可变的日志文件。分区允许Kafka进行水平扩展，通过分区，Kafka可以处理大规模的数据流，并实现并行处理。

- Broker节点

Kafka集群由多个代理节点组成，每个代理节点负责存储和管理一个或多个分区。代理节点共同工作，以确保数据的高可用性和故障恢复能力。

2.1.2 实现细节

在本项目中，Kafka被用作事件通道，通过它实现了事件驱动的架构。具体实现细节如下：

1. Kafka设置

Kafka配置文件

在 `docker-compose.yml` 文件中配置了Kafka和Zookeeper：

```
1 version: '3.7'
2 services:
3   zookeeper:
4     image: wurstmeister/zookeeper:3.4.6
5     ports:
6       - "2181:2181"
7   kafka:
8     image: wurstmeister/kafka:2.13-2.7.0
9     ports:
10      - "9092:9092"
11     environment:
12       KAFKA_ADVERTISED_LISTENERS: INSIDE://kafka:9092,OUTSIDE://localhost:9092
13       KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: INSIDE:PLAINTEXT,OUTSIDE:PLAINTEXT
14       KAFKA_INTER_BROKER_LISTENER_NAME: INSIDE
15       KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
16       KAFKA_AUTO_CREATE_TOPICS_ENABLE: 'true'
```

```
17     volumes:
18         - /var/run/docker.sock:/var/run/docker.sock
```

启动Kafka和Zookeeper:

```
1 docker-compose up -d
```

2. 定义事件

事件类

定义了不同类型的事件，例如LogisticsEvent、InspectionEvent、StorageEvent和ShipmentEvent:

```
1 @Data
2 @AllArgsConstructor
3 @NoArgsConstructor
4 public class LogisticsEvent {
5     private String eventId;
6     private String shipmentId;
7     private String status;
8 }
```

3. 生产者组件

生产者服务

在物流、检查、存储和出货的各个服务中，实现事件的生成和发送:

```
1 @Service
2 public class EventProducer {
3
4     @Autowired
5     private KafkaTemplate<String, LogisticsEvent> kafkaTemplate;
6
7     public void sendLogisticsEvent(LogisticsEvent event) {
8         kafkaTemplate.send("logistics_topic", event);
9     }
10 }
```

发送事件

```

1 @RestController
2 @RequestMapping("/api/v1/warehouses")
3 public class WarehouseController {
4
5     @Autowired
6     private EventProducer eventProducer;
7
8     @PostMapping("/logistics")
9     public ResponseEntity<String> handleLogisticsEvent(@RequestBody
10     LogisticsEvent event) {
11         eventProducer.sendLogisticsEvent(event);
12         return ResponseEntity.ok("Logistics event processed: " + event);
13     }
14 }

```

4. 消费者组件

消费者服务

在仓库服务中实现事件的消费和处理：

```

1 @Service
2 public class OrderConsumer {
3
4     @Autowired
5     private WarehouseService warehouseService;
6
7     @KafkaListener(topics = "logistics_topic", groupId = "warehouse-group")
8     public void consumeLogisticsEvent(LogisticsEvent event) {
9         warehouseService.handleLogisticsEvent(event);
10    }
11 }

```

处理事件

```

1 @Service
2 public class WarehouseService {
3
4     public void handleLogisticsEvent(LogisticsEvent event) {
5         System.out.println("Handling logistics event: " + event);
6     }
7 }

```

5. 事件驱动的流程

每个事件都触发特定的服务操作，并通过WebSocket将结果推送到前端。以下是各个流程的事件触发和处理示例：

1. 物流卸货事件

- 发送请求：

```
curl -X POST -H "Content-Type: application/json" -d '{"eventId":"12345","shipmentId":"54321","status":"UNLOADING"}' http://localhost:8080/api/v1/warehouses/logistics
```
- Kafka主题：`logistics_topic`
- 处理结果推送到前端：

```
"Logistics event processed: LogisticsEvent(eventId=12345, shipmentId=54321, status=UNLOADING)"
```

2. 检查和记录事件

- 发送请求：

```
curl -X POST -H "Content-Type: application/json" -d '{"eventId":"12346","shipmentId":"54322","status":"INSPECTED","remarks":"All goods are in good condition"}' http://localhost:8080/api/v1/warehouses/inspection
```
- Kafka主题：`inspection_topic`
- 处理结果推送到前端：

```
"Inspection event processed: InspectionEvent(eventId=12346, shipmentId=54322, status=INSPECTED, remarks=All goods are in good condition)"
```

3. 仓库存储事件

- 发送请求：

```
curl -X POST -H "Content-Type: application/json" -d '{"eventId":"12347","storageId":"98765","status":"STORED"}' http://localhost:8080/api/v1/warehouses/storage
```
- Kafka主题：`storage_topic`
- 处理结果推送到前端：

```
"Storage event processed: StorageEvent(eventId=12347, storageId=98765, status=STORED)"
```

4. 出货装载事件

- 发送请求：

```
curl -X POST -H "Content-Type: application/json" -d '{"eventId":"12348","shipmentId":"54323","status":"SHIPPED"}' http://localhost:8080/api/v1/warehouses/shipment
```
- Kafka主题：`shipment_topic`
- 处理结果推送到前端：

```
"Shipment event processed: ShipmentEvent(eventId=12348, shipmentId=54323, status=SHIPPED)"
```


这些事件通过Kafka进行通信，并通过WebSocket将结果实时推送到前端，确保整个系统的事件驱动和实时响应。

2.2 WebSocket

2.2.1 技术栈简介

WebSocket 是一种全双工通信协议，基于TCP连接，旨在实现浏览器与服务器之间的双向实时通信。它在2008年首次提出，并在2011年成为IETF RFC 6455标准。WebSocket协议由IETF标准化，API由W3C标准化。

- 全双工通信

WebSocket允许客户端和服务端之间同时发送和接收数据。这与传统的HTTP协议不同，HTTP协议是请求-响应模型，而WebSocket是持久连接，允许双方实时交换数据。

- 低延迟

WebSocket连接一旦建立，通信就没有了HTTP那样的头部开销，因而延迟较低，适用于需要低延迟的数据交换场景。

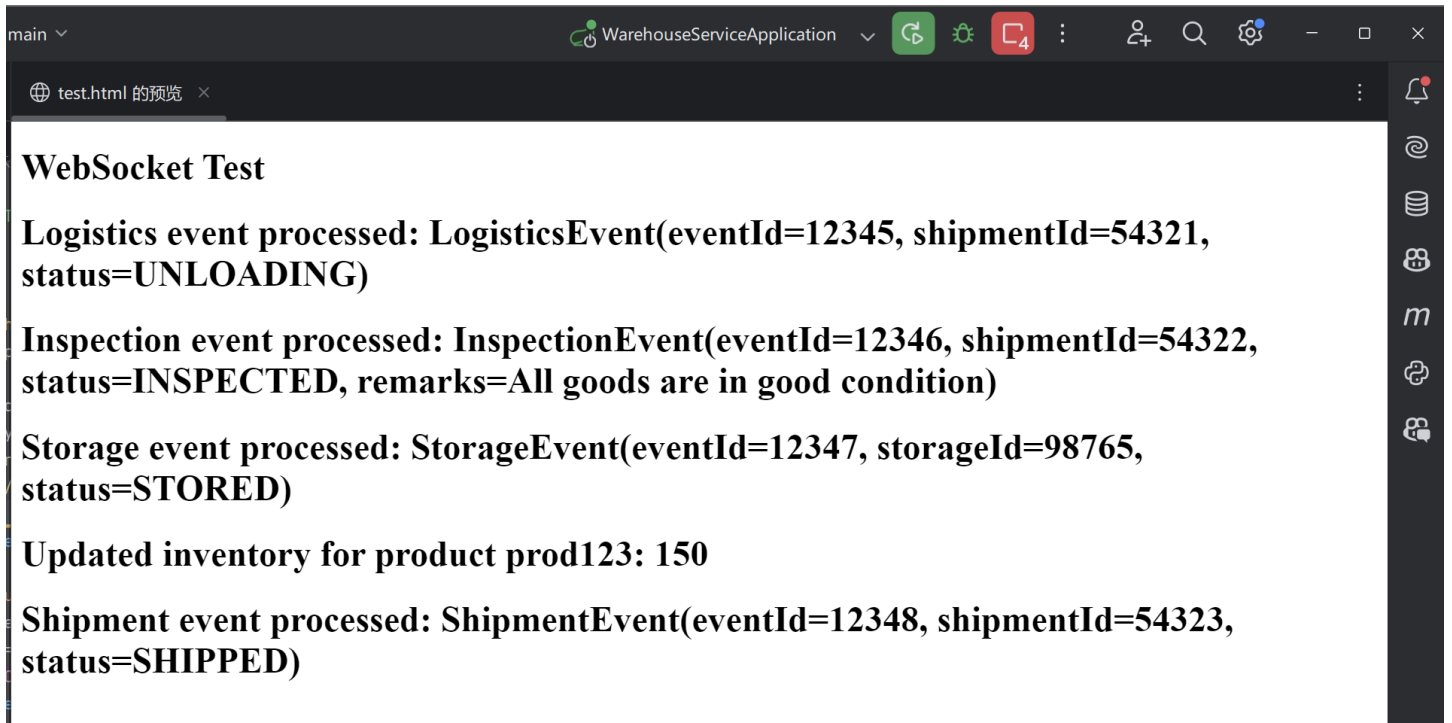
- 长连接

WebSocket连接可以长时间保持，不像HTTP连接那样在数据交换后立即关闭，这使得它特别适用于需要持续数据更新的应用场景。

2.2.2 实现细节

在本项目中，WebSocket通过以下步骤实现了前后端的实时通信：

1. 配置WebSocket，使Spring Boot应用程序能够处理WebSocket连接。
2. 定义WebSocket处理器，管理客户端连接并处理发送的消息。
3. 在事件处理服务中集成WebSocket通知，将事件处理结果推送到所有连接的客户端。
4. 前端使用WebSocket连接到后端，接收并显示推送的消息。



在本项目中，WebSocket被用作在前端和后端之间实现实时通信的技术。通过WebSocket，模拟的事件处理结果可以即时推送到前端，提供实时的反馈。

详情如下：

1. WebSocket配置

WebSocket配置类

在Spring Boot中，我们需要配置WebSocket，使其能够处理来自客户端的连接请求。

```
1 import org.springframework.context.annotation.Configuration;
2 import org.springframework.web.socket.config.annotation.EnableWebSocket;
3 import org.springframework.web.socket.config.annotation.WebSocketConfigurer;
4 import
    org.springframework.web.socket.config.annotation.WebSocketHandlerRegistry;
5
6 @Configuration
7 @EnableWebSocket
8 public class WebSocketConfig implements WebSocketConfigurer {
9
10     private final WebSocketHandler webSocketHandler;
11
12     public WebSocketConfig(WebSocketHandler webSocketHandler) {
13         this.webSocketHandler = webSocketHandler;
14     }
15
16     @Override
17     public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
18         registry.addHandler(webSocketHandler, "/ws").setAllowedOrigins("*");
19     }
20 }
```

WebSocket处理器

定义WebSocket处理器来处理消息：

```
1  import org.springframework.web.socket.WebSocketSession;
2  import org.springframework.web.socket.handler.TextWebSocketHandler;
3  import org.springframework.web.socket.TextMessage;
4  import org.springframework.stereotype.Component;
5
6  import java.io.IOException;
7  import java.util.ArrayList;
8  import java.util.List;
9
10 @Component
11 public class WebSocketHandler extends TextWebSocketHandler {
12
13     private List<WebSocketSession> sessions = new ArrayList<>();
14
15     @Override
16     public void afterConnectionEstablished(WebSocketSession session) {
17         sessions.add(session);
18     }
19
20     @Override
21     public void handleTextMessage(WebSocketSession session, TextMessage
message) {
22         // 处理接收到的消息
23     }
24
25     public void notifyClients(String message) {
26         for (WebSocketSession session : sessions) {
27             try {
28                 session.sendMessage(new TextMessage(message));
29             } catch (IOException e) {
30                 e.printStackTrace();
31             }
32         }
33     }
34 }
```

2. 事件处理中的WebSocket通知

在服务中集成WebSocket通知

在每个事件处理过程中，通过WebSocket将处理结果推送到前端。

```
1 import org.springframework.beans.factory.annotation.Autowired;
2 import org.springframework.stereotype.Service;
3
4 @Service
5 public class WarehouseService {
6
7     @Autowired
8     private WebSocketHandler webSocketHandler;
9
10    public void handleLogisticsEvent(LogisticsEvent event) {
11        System.out.println("Handling logistics event: " + event);
12        webSocketHandler.notifyClients("Logistics event processed: " + event);
13    }
14
15    public void handleInspectionEvent(InspectionEvent event) {
16        System.out.println("Handling inspection event: " + event);
17        webSocketHandler.notifyClients("Inspection event processed: " + event);
18    }
19
20    public void handleStorageEvent(StorageEvent event) {
21        System.out.println("Handling storage event: " + event);
22        webSocketHandler.notifyClients("Storage event processed: " + event);
23    }
24
25    public void handleShipmentEvent(ShipmentEvent event) {
26        System.out.println("Handling shipment event: " + event);
27        webSocketHandler.notifyClients("Shipment event processed: " + event);
28    }
29 }
```

3. 前端接收WebSocket推送

前端HTML和JavaScript示例

前端页面中，通过WebSocket连接到后端，并接收推送的消息。

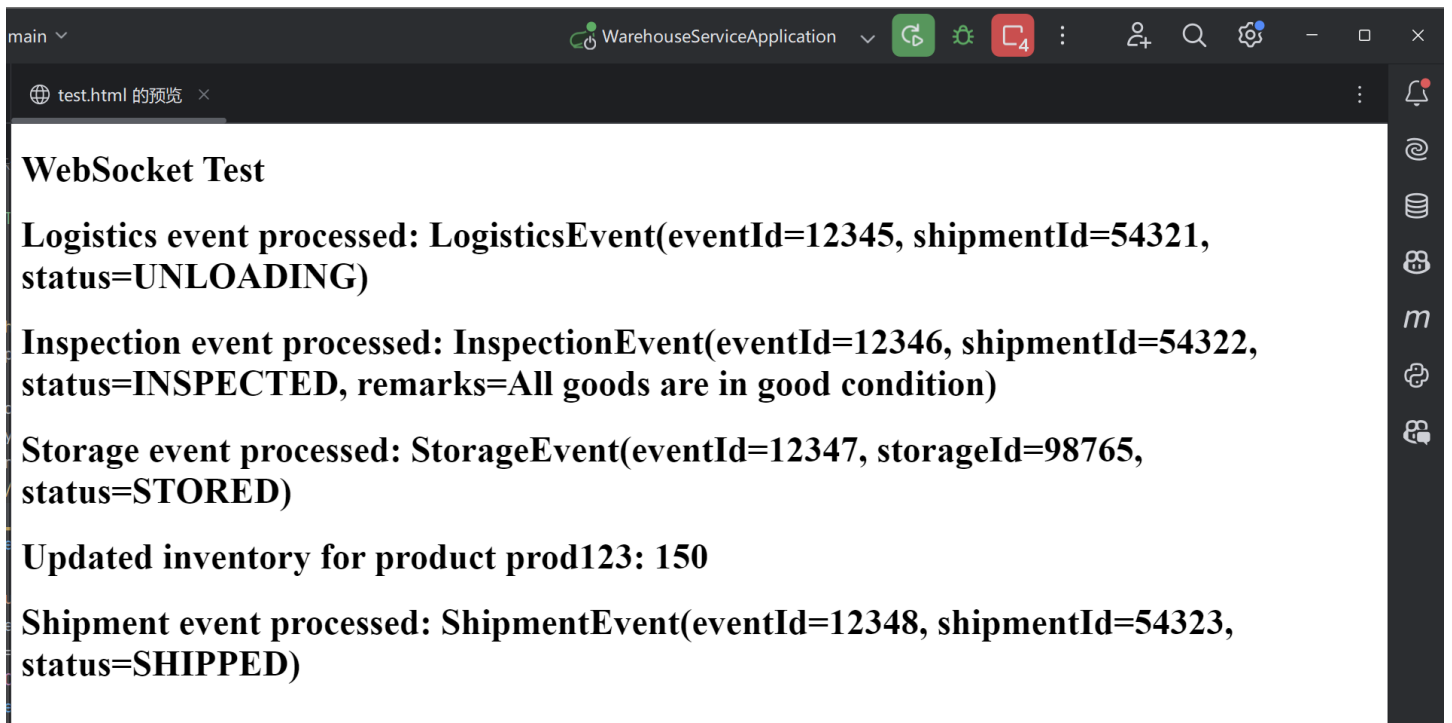
```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>WebSocket Test</title>
6 </head>
7 <body>
```

```

8      <h1>WebSocket Test</h1>
9      <div id="messages"></div>
10     <script>
11         const ws = new WebSocket("ws://localhost:8080/ws");
12
13         ws.onmessage = function(event) {
14             const messages = document.getElementById('messages');
15             const message = document.createElement('p');
16             message.textContent = event.data;
17             messages.appendChild(message);
18         };
19
20         ws.onerror = function(error) {
21             console.log("WebSocket error: " + error);
22         };
23     </script>
24 </body>
25 </html>

```

将各事件运行后前端显示如下：



2.3 ZooKeeper

2.3.1 技术栈简介

ZooKeeper 是一个分布式协调服务，旨在为分布式应用提供一致性、高可用性和高性能的协调服务。

- 分布式协调

ZooKeeper 提供了一组基本服务，使得分布式应用程序可以在不需要复杂代码的情况下实现协调功能，例如配置管理、同步、分布式锁和领导选举等。

- 数据存储

ZooKeeper 将数据存储在内​​存中，提供高吞吐量和低延迟的数据访问。数据以层次结构存储，类似于文件系统，称为znode树。

- 一致性保证

ZooKeeper 提供严格的一致性保证，即任何更新操作在传播到所有节点之前，不会向客户端确认。这确保了所有客户端看到的数据是一致的。

2.3.2 实现细节

在本项目中，ZooKeeper被用作Kafka的分布式协调服务，管理和维护Kafka集群的状态。具体实现步骤如下：

1. 在 `docker-compose.yml` 文件中配置ZooKeeper和Kafka。
2. 在Kafka的配置文件中设置ZooKeeper连接。
3. 在Spring Boot应用程序中配置Kafka与ZooKeeper的连接。
4. 启动ZooKeeper和Kafka，确保Kafka集群能够正确协调和管理。

通过这些步骤，确保了Kafka集群的高可用性和可扩展性，使系统能够高效地处理和管理事件驱动的仓库管理流程。以下是项目中与ZooKeeper相关的具体实现细节：

1. ZooKeeper配置

`docker-compose.yml`文件中的ZooKeeper配置

在 `docker-compose.yml` 文件中配置了ZooKeeper和Kafka，使得Kafka能够使用ZooKeeper进行分布式协调。

```
1 version: '3.7'
2 services:
3   zookeeper:
4     image: wurstmeister/zookeeper:3.4.6
5     ports:
6       - "2181:2181"
7   kafka:
8     image: wurstmeister/kafka:2.13-2.7.0
9     ports:
10      - "9092:9092"
11    environment:
12      KAFKA_ADVERTISED_LISTENERS: INSIDE://kafka:9092,OUTSIDE://localhost:9092
13      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: INSIDE:PLAINTEXT,OUTSIDE:PLAINTEXT
14      KAFKA_INTER_BROKER_LISTENER_NAME: INSIDE
15      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
```

```
16     KAFKA_AUTO_CREATE_TOPICS_ENABLE: 'true'
17     volumes:
18     - /var/run/docker.sock:/var/run/docker.sock
19
```

启动ZooKeeper和Kafka：

```
1 docker-compose up -d
```

2. ZooKeeper在Kafka中的作用

管理Kafka集群

ZooKeeper在Kafka中主要用于以下任务：

- 管理和维护Kafka集群的元数据。
- 追踪Kafka broker的状态（例如，哪个broker是leader）。
- 处理Kafka的分区和副本分配。
- 维护Kafka的消费者组偏移。

Kafka配置文件中的ZooKeeper设置

在Kafka的配置文件 `server.properties` 中，设置ZooKeeper连接：

```
1 zookeeper.connect=localhost:2181
```

3. 事件驱动架构中的ZooKeeper

使用ZooKeeper协调Kafka

Kafka使用ZooKeeper来确保消息队列的分区和副本管理的正确性。每当有新的broker加入Kafka集群或者现有broker失效时，ZooKeeper都会帮助协调这些变化。

4. 实现代码示例

Kafka配置示例

在 `application.yml` 或 `application.properties` 中，可以配置Kafka连接ZooKeeper的相关参数：

```
1 spring:
2   kafka:
3     bootstrap-servers: localhost:9092
```

```
4     consumer:
5         group-id: warehouse-group
6         auto-offset-reset: earliest
7         properties:
8             spring.json.value.default.type:
com.example.warehouseservice.model.Event
9     producer:
10         key-serializer: org.apache.kafka.common.serialization.StringSerializer
11         value-serializer:
org.springframework.kafka.support.serializer.JsonSerializer
12     zookeeper:
13         connect: localhost:2181
```

5. 启动和测试

启动ZooKeeper和Kafka

确保 `docker-compose.yml` 配置正确后，运行以下命令启动ZooKeeper和Kafka：

```
1 docker-compose up -d
```

启动服务后，Kafka将使用ZooKeeper来管理和协调集群。

三、事件驱动流程实现

3.1 物流卸货

流程描述：

- 货物到达时开始卸货过程。
- 物流团队确保货物安全高效地卸载。
- 使用叉车或传送带将货物从运输车辆移动到仓库地板。

关键代码：

```
1 // LogisticsEvent.java
2 package com.example.warehouseservice.model;
3
4 import lombok.AllArgsConstructor;
5 import lombok.Data;
6 import lombok.NoArgsConstructor;
7
8 @Data
```

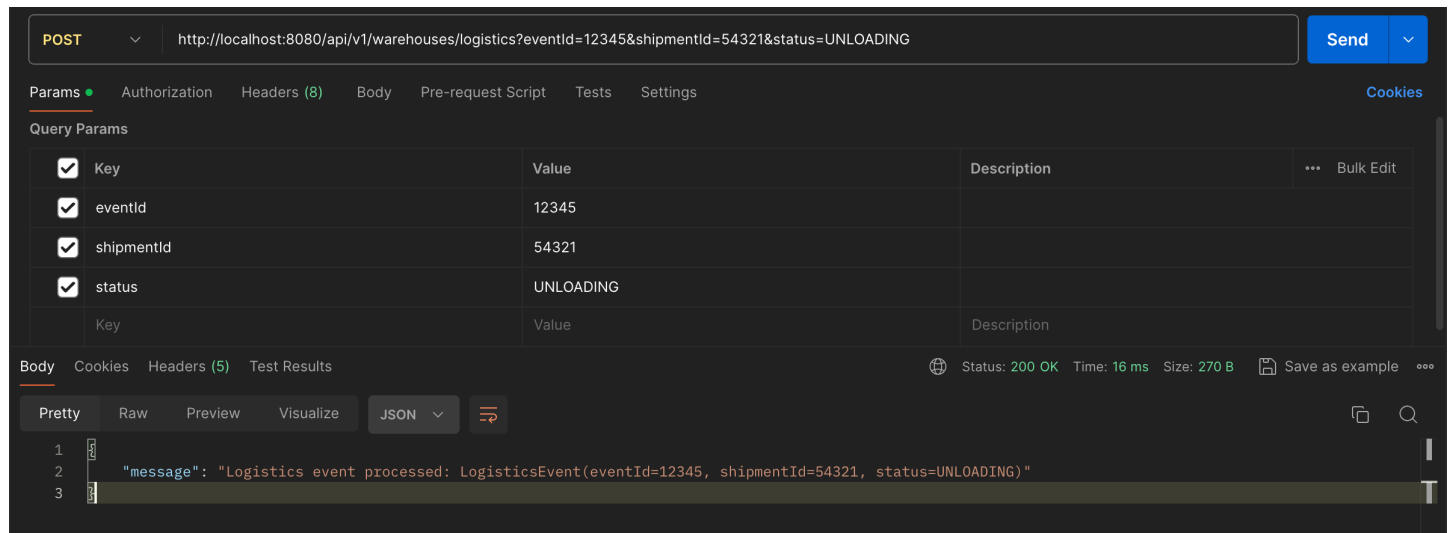


```
9  @AllArgsConstructor
10 @NoArgsConstructor
11 public class LogisticsEvent {
12     private String eventId;
13     private String shipmentId;
14     private String status;
15 }
16
17 // WarehouseService.java
18 package com.example.warehouseservice.service;
19
20 import com.example.warehouseservice.model.LogisticsEvent;
21 import com.example.warehouseservice.websocket.WebSocketService;
22 import org.springframework.beans.factory.annotation.Autowired;
23 import org.springframework.stereotype.Service;
24
25 @Service
26 public class WarehouseService {
27
28     @Autowired
29     private WebSocketService webSocketService;
30
31     public void handleLogisticsEvent(LogisticsEvent event) {
32         System.out.println("Handling logistics event: " + event);
33         webSocketService.notifyClients("Logistics event processed: " + event);
34     }
35 }
36
37 // OrderConsumer.java
38 package com.example.warehouseservice.kafka;
39
40 import com.example.warehouseservice.model.LogisticsEvent;
41 import com.example.warehouseservice.service.WarehouseService;
42 import org.springframework.beans.factory.annotation.Autowired;
43 import org.springframework.kafka.annotation.KafkaListener;
44 import org.springframework.stereotype.Service;
45
46 @Service
47 public class OrderConsumer {
48
49     @Autowired
50     private WarehouseService warehouseService;
51
52     @KafkaListener(topics = "logistics_topic", groupId = "warehouse-group")
53     public void consumeLogisticsEvent(LogisticsEvent event) {
54         warehouseService.handleLogisticsEvent(event);
55     }
56 }
```

运行结果：

```
1 Handling logistics event: LogisticsEvent(eventId=12345, shipmentId=54321, status=UNLOADING)
```

运行结果（postman）：



3.2 检查和记录

流程描述：

- 每件货物都需要检查以确保其状况良好并与发货详情一致。
- 记录任何损坏或差异并报告给物流团队。
- 验证并记录必要的文件，如发票、提单和保修证书。
- 将信息输入仓库管理系统以进行跟踪和库存管理。
- 使用RFID标签对货物进行分类和排序。

关键代码：

```
1 // InspectionEvent.java
2 package com.example.warehouseservice.model;
3
4 import lombok.AllArgsConstructor;
5 import lombok.Data;
6 import lombok.NoArgsConstructor;
7
8 @Data
```

```

9  @AllArgsConstructor
10 @NoArgsConstructor
11 public class InspectionEvent {
12     private String eventId;
13     private String shipmentId;
14     private String status;
15     private String remarks;
16 }
17
18 // WarehouseService.java
19 package com.example.warehouseservice.service;
20
21 import com.example.warehouseservice.model.InspectionEvent;
22 import com.example.warehouseservice.websocket.WebSocketService;
23 import org.springframework.beans.factory.annotation.Autowired;
24 import org.springframework.stereotype.Service;
25
26 @Service
27 public class WarehouseService {
28
29     @Autowired
30     private WebSocketService websocketService;
31
32     public void handleInspectionEvent(InspectionEvent event) {
33         System.out.println("Handling inspection event: " + event);
34         websocketService.notifyClients("Inspection event processed: " + event);
35     }
36 }
37
38 // OrderConsumer.java
39 package com.example.warehouseservice.kafka;
40
41 import com.example.warehouseservice.model.InspectionEvent;
42 import com.example.warehouseservice.service.WarehouseService;
43 import org.springframework.beans.factory.annotation.Autowired;
44 import org.springframework.kafka.annotation.KafkaListener;
45 import org.springframework.stereotype.Service;
46
47 @Service
48 public class OrderConsumer {
49
50     @Autowired
51     private WarehouseService warehouseService;
52
53     @KafkaListener(topics = "inspection_topic", groupId = "warehouse-group")
54     public void consumeInspectionEvent(InspectionEvent event) {
55         warehouseService.handleInspectionEvent(event);

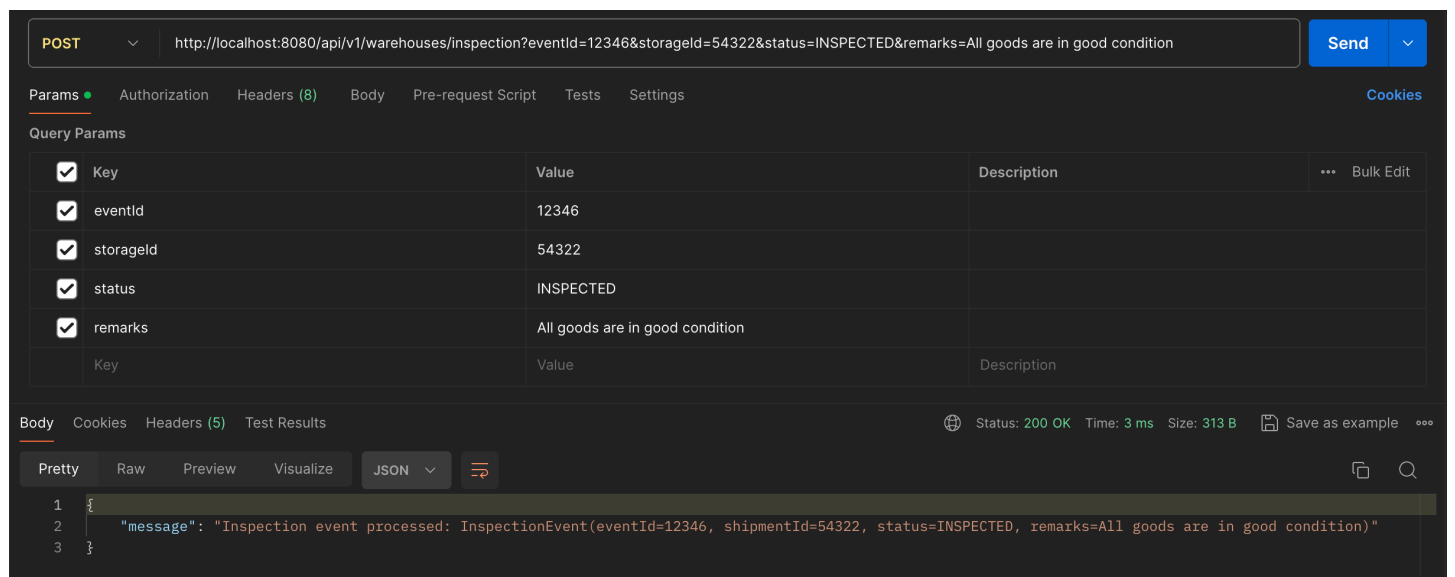
```

```
56     }  
57 }  
58
```

运行结果：

```
1 Handling inspection event: InspectionEvent(eventId=12346, shipmentId=54322,  
    status=INSPECTED, remarks=All goods are in good condition)
```

运行结果（postman）：



3.3 仓库存储

流程描述：

- 卸货和检查事件启动分配仓库位置的过程，通过事件来占用某些位置。
- 通过事件模拟自动车辆运输货物到仓库位置，使用提升机装卸货物。
- 通过事件模拟提升机从车辆接收货物并提升到指定位置。
- 仓库管理系统根据事件更新到货和存储信息。

关键代码：

```
1 // StorageEvent.java  
2 package com.example.warehouseservice.model;  
3  
4 import lombok.AllArgsConstructor;  
5 import lombok.Data;  
6 import lombok.NoArgsConstructor;  
7
```

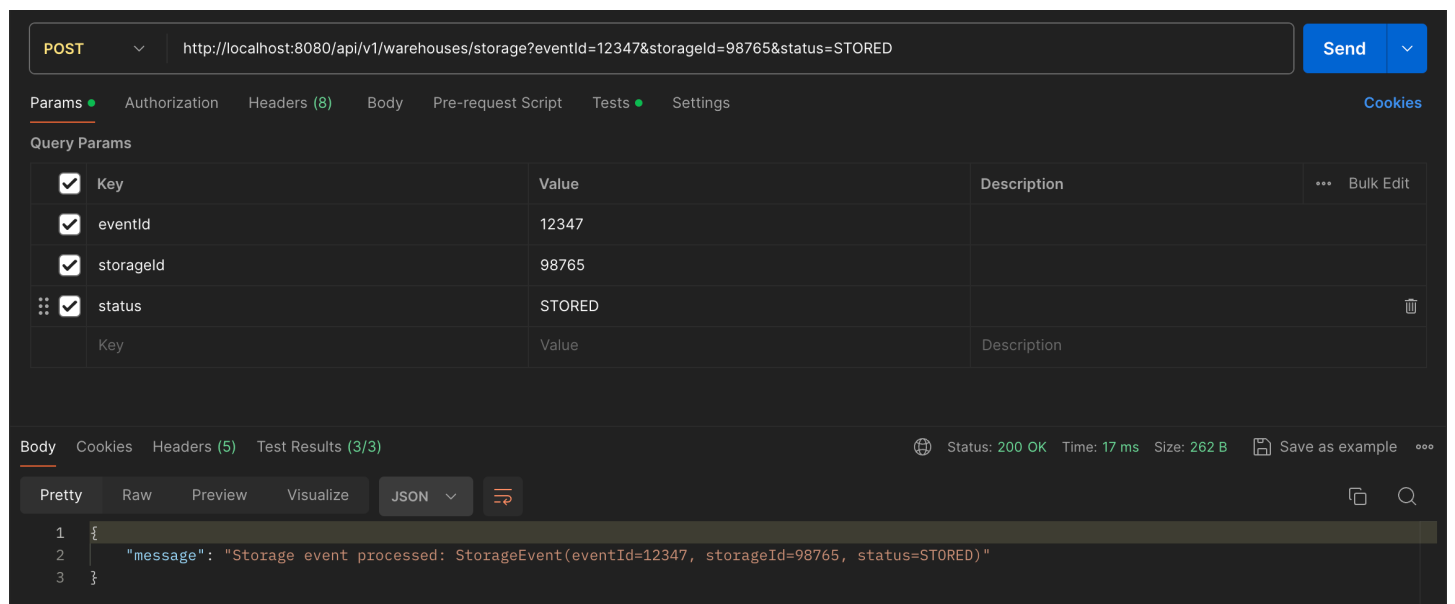
```
8  @Data
9  @AllArgsConstructor
10 @NoArgsConstructor
11 public class StorageEvent {
12     private String eventId;
13     private String storageId;
14     private String status;
15 }
16
17 // WarehouseService.java
18 package com.example.warehouseservice.service;
19
20 import com.example.warehouseservice.model.StorageEvent;
21 import com.example.warehouseservice.websocket.WebSocketService;
22 import org.springframework.beans.factory.annotation.Autowired;
23 import org.springframework.stereotype.Service;
24
25 @Service
26 public class WarehouseService {
27
28     @Autowired
29     private WebSocketService websocketService;
30
31     public void handleStorageEvent(StorageEvent event) {
32         System.out.println("Handling storage event: " + event);
33         websocketService.notifyClients("Storage event processed: " + event);
34     }
35 }
36
37 // OrderConsumer.java
38 package com.example.warehouseservice.kafka;
39
40 import com.example.warehouseservice.model.StorageEvent;
41 import com.example.warehouseservice.service.WarehouseService;
42 import org.springframework.beans.factory.annotation.Autowired;
43 import org.springframework.kafka.annotation.KafkaListener;
44 import org.springframework.stereotype.Service;
45
46 @Service
47 public class OrderConsumer {
48
49     @Autowired
50     private WarehouseService warehouseService;
51
52     @KafkaListener(topics = "storage_topic", groupId = "warehouse-group")
53     public void consumeStorageEvent(StorageEvent event) {
54         warehouseService.handleStorageEvent(event);
55     }
56 }
```

```
55     }  
56 }  
57
```

运行结果：

```
1 Handling storage event: StorageEvent(eventId=12347, storageId=98765,  
  status=STORED)
```

运行结果（postman）：



3.4 库存管理

流程描述：

- 仓库管理系统根据到达事件更新库存信息。
- 监控和跟踪库存水平以确保准确性，避免库存过多或不足。

关键代码：

```
1 // InventoryService.java  
2 package com.example.warehouseservice.service;  
3  
4 import org.springframework.stereotype.Service;  
5  
6 import java.util.HashMap;  
7 import java.util.Map;  
8  
9 @Service
```

```

10 public class InventoryService {
11
12     private Map<String, Integer> inventory = new HashMap<>();
13
14     public void updateInventory(String productId, int quantity) {
15         inventory.put(productId, inventory.getOrDefault(productId, 0) +
quantity);
16         System.out.println("Updated inventory for product " + productId + ": "
+ inventory.get(productId));
17     }
18
19     public int getInventory(String productId) {
20         return inventory.getOrDefault(productId, 0);
21     }
22 }
23

```

运行结果：

```

1 Updated inventory for product prod123: 150

```

运行结果（postman）：

The screenshot shows a Postman interface for a POST request to `http://localhost:8080/api/v1/warehouses/inventory?productId=prod123&quantity=150`. The request is successful with a status of 200 OK. The response body is a JSON object: `{ "message": "Updated inventory for product prod123: 150" }`.

Key	Value	Description
productId	prod123	
quantity	150	

3.5 出货装载

流程描述：

- 收到订单时，仓库管理系统识别所需物品的位置。
- 从存储中取出货物并准备装载。

- 使用适当的设备将货物安全地移动到出库区。
- 准备并附上必要的出货文件。
- 货物装载后准备运输到目的地。

关键代码：

```
1 // ShipmentEvent.java
2 package com.example.warehouseservice.model;
3
4 import lombok.AllArgsConstructor;
5 import lombok.Data;
6 import lombok.NoArgsConstructor;
7
8 @Data
9 @AllArgsConstructor
10 @NoArgsConstructor
11 public class ShipmentEvent {
12     private String eventId;
13     private String shipmentId;
14     private String status;
15 }
16
17 // WarehouseService.java
18 package com.example.warehouseservice.service;
19
20 import com.example.warehouseservice.model.ShipmentEvent;
21 import com.example.warehouseservice.websocket.WebSocketService;
22 import org.springframework.beans.factory.annotation.Autowired;
23 import org.springframework.stereotype.Service;
24
25 @Service
26 public class WarehouseService {
27
28     @Autowired
29     private WebSocketService websocketService;
30
31     public void handleShipmentEvent(ShipmentEvent event) {
32         System.out.println("Handling shipment event: " + event);
33         websocketService.notifyClients("Shipment event processed: " + event);
34     }
35 }
36
37 // OrderConsumer.java
38 package com.example.warehouseservice.kafka;
39
```



```

40 import com.example.warehouseservice.model.ShipmentEvent;
41 import com.example.warehouseservice.service.WarehouseService;
42 import org.springframework.beans.factory.annotation.Autowired;
43 import org.springframework.kafka.annotation.KafkaListener;
44 import org.springframework.stereotype.Service;
45
46 @Service
47 public class OrderConsumer {
48
49     @Autowired
50     private WarehouseService warehouseService;
51
52     @KafkaListener(topics = "shipment_topic", groupId = "warehouse-group")
53     public void consumeShipmentEvent(ShipmentEvent event) {
54         warehouseService.handleShipmentEvent(event);
55     }
56 }
57

```

运行结果：

```

1 Handling shipment event: ShipmentEvent(eventId=12348, shipmentId=54323,
status=SHIPPED)

```

运行结果（postman）：

POST http://localhost:8080/api/v1/warehouses/shipment?eventId=12348&shipmentId=54323&status=SHIPPED

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

Query Params

Key	Value	Description
eventId	12348	
shipmentId	54323	
status	SHIPPED	

Body Cookies Headers (5) Test Results (3/3) Status: 200 OK Time: 58 ms Size: 266 B Save as example

Pretty Raw Preview Visualize JSON

```

1 {
2   "message": "Shipment event processed: ShipmentEvent(eventId=12348, shipmentId=54323, status=SHIPPED)"
3 }

```

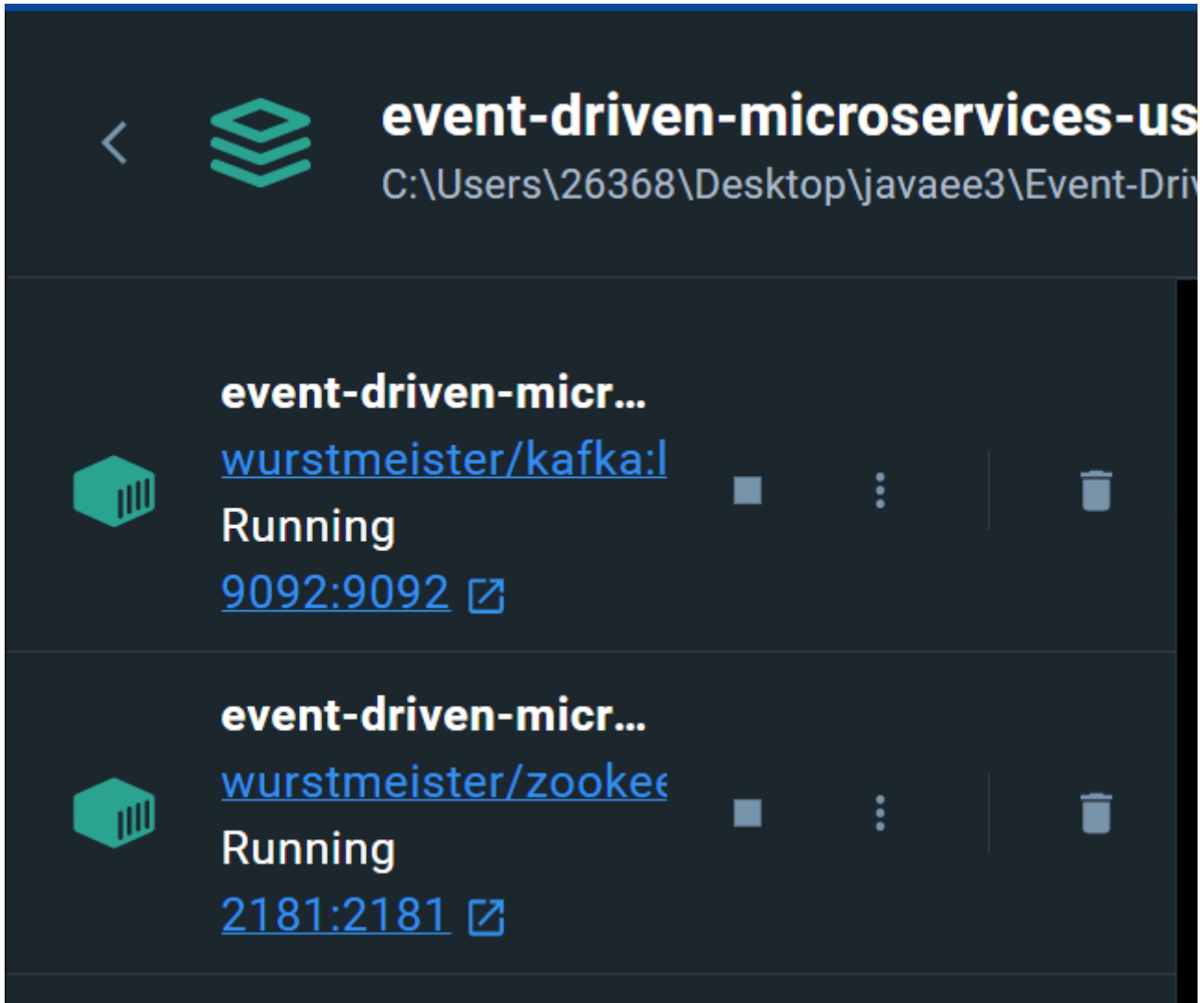
四、系统测试与运行截图

4.1 系统测试

1. 启动Kafka和Zookeeper:

```
1 docker-compose up -d
```

可以看到kafka和zookeeper已启动，端口分别为9092和2181



此步骤需要确保yaml文件存在:

```
1 version: '3'
2 services:
3   zookeeper:
4     image: wurstmeister/zookeeper:3.4.6
5     ports:
6       - "2181:2181"
7
8   kafka:
9     image: wurstmeister/kafka:latest
```

```

10     ports:
11         - "9092:9092"
12     environment:
13         KAFKA_ADVERTISED_LISTENERS: INSIDE://kafka:29092,OUTSIDE://localhost:9092
14         KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: INSIDE:PLAINTEXT,OUTSIDE:PLAINTEXT
15         KAFKA_LISTENERS: INSIDE://0.0.0.0:29092,OUTSIDE://0.0.0.0:9092
16         KAFKA_INTER_BROKER_LISTENER_NAME: INSIDE
17         KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
18     volumes:
19         - /var/run/docker.sock:/var/run/docker.sock
20

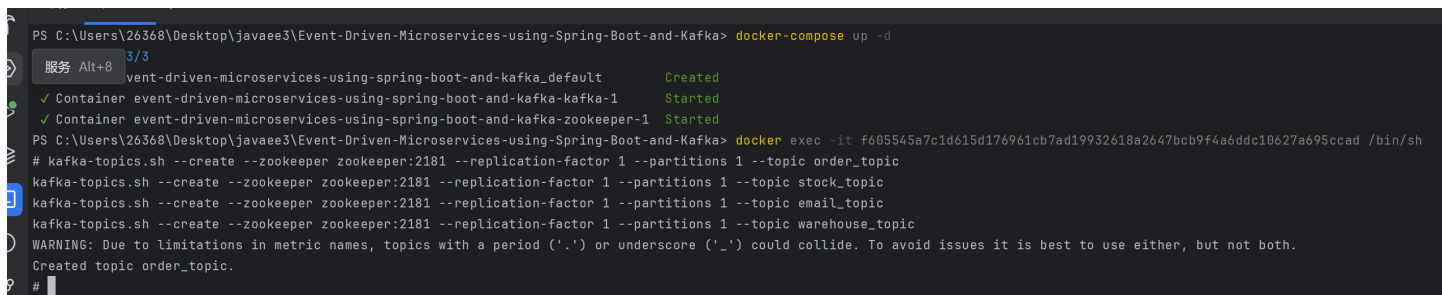
```

确认Kafka和Zookeeper启动后，使用以下命令创建所需的Kafka主题：

```

1 docker exec -it
  f605545a7c1d615d176961cb7ad19932618a2647bcb9f4a6ddc10627a695ccad /bin/sh
2 kafka-topics.sh --create --zookeeper zookeeper:2181 --replication-factor 1 --
  partitions 1 --topic order_topic kafka-topics.sh --create --zookeeper
  zookeeper:2181 --replication-factor 1 --partitions 1 --topic stock_topic kafka-
  topics.sh --create --zookeeper zookeeper:2181 --replication-factor 1 --
  partitions 1 --topic email_topic kafka-topics.sh --create --zookeeper
  zookeeper:2181 --replication-factor 1 --partitions 1 --topic warehouse_topic

```



```

PS C:\Users\26368\Desktop\javaee3\Event-Driven-Microservices-using-Spring-Boot-and-Kafka> docker-compose up -d
3/3
vent-driven-microservices-using-spring-boot-and-kafka_default Created
Container event-driven-microservices-using-spring-boot-and-kafka-kafka-1 Started
Container event-driven-microservices-using-spring-boot-and-kafka-zookeeper-1 Started
PS C:\Users\26368\Desktop\javaee3\Event-Driven-Microservices-using-Spring-Boot-and-Kafka> docker exec -it f605545a7c1d615d176961cb7ad19932618a2647bcb9f4a6ddc10627a695ccad /bin/sh
# kafka-topics.sh --create --zookeeper zookeeper:2181 --replication-factor 1 --partitions 1 --topic order_topic
kafka-topics.sh --create --zookeeper zookeeper:2181 --replication-factor 1 --partitions 1 --topic stock_topic
kafka-topics.sh --create --zookeeper zookeeper:2181 --replication-factor 1 --partitions 1 --topic email_topic
kafka-topics.sh --create --zookeeper zookeeper:2181 --replication-factor 1 --partitions 1 --topic warehouse_topic
WARNING: Due to limitations in metric names, topics with a period ('.') or underscore ('_') could collide. To avoid issues it is best to use either, but not both.
Created topic order_topic.
#

```

2. 逐个启动所有服务：

```

1 mvn spring-boot:run -pl order-service
2 mvn spring-boot:run -pl stock-service
3 mvn spring-boot:run -pl email-service
4 mvn spring-boot:run -pl warehouse-service

```

3. 发送模拟事件：

4.2 测试结果

使用Postman发送HTTP请求到相应的API端点，生成各类事件：

• 物流卸货

POST

http://localhost:8080/api/v1/warehouses/logistics?eventId=12345&shipmentId=54321&status=UNLOADING

Send

Params

Authorization

Headers (8)

Body

Pre-request Script

Tests

Settings

Cookies

Query Params

<input checked="" type="checkbox"/>	Key	Value	Description	...	Bulk Edit
<input checked="" type="checkbox"/>	eventId	12345			
<input checked="" type="checkbox"/>	shipmentId	54321			
<input checked="" type="checkbox"/>	status	UNLOADING			
<input type="checkbox"/>	Key	Value	Description		

Body

Cookies

Headers (5)

Test Results

Status: 200 OK

Time: 16 ms

Size: 270 B

Save as example

Pretty

Raw

Preview

Visualize

JSON

1

2

3

"message": "Logistics event processed: LogisticsEvent(eventId=12345, shipmentId=54321, status=UNLOADING)"

POST

/api/v1/warehouses/logistics 处理物流事件

Try it out

Parameters

Name	Description
eventId ★ required integer (\$int32) (query)	<input type="text" value="eventId"/>
shipmentId ★ required integer (\$int32) (query)	<input type="text" value="shipmentId"/>
status ★ required string (query)	<input type="text" value="status"/>

Responses

Code	Description	Links
200	OK	No links

Media type

Controls Accept header.

Example Value | Schema

```
{
  "additionalProp1": "string",
  "additionalProp2": "string",
  "additionalProp3": "string"
}
```

• 检查和记录

POST

http://localhost:8080/api/v1/warehouses/inspection?eventId=12346&storageld=54322&status=INSPECTED&remarks=All goods are in good condition

Send

Params

Authorization

Headers (8)

Body

Pre-request Script

Tests

Settings

Cookies

Query Params

<input checked="" type="checkbox"/>	Key	Value	Description	...	Bulk Edit
<input checked="" type="checkbox"/>	eventId	12346			
<input checked="" type="checkbox"/>	storageld	54322			
<input checked="" type="checkbox"/>	status	INSPECTED			
<input checked="" type="checkbox"/>	remarks	All goods are in good condition			
<input type="checkbox"/>	Key	Value	Description		

Body

Cookies

Headers (5)

Test Results

Status: 200 OK

Time: 3 ms

Size: 313 B

Save as example

...

Pretty

Raw

Preview

Visualize

JSON

...

...

```
1 {
2   "message": "Inspection event processed: InspectionEvent(eventId=12346, shipmentId=54322, status=INSPECTED, remarks=All goods are in good condition)"
3 }
```

POST

/api/v1/warehouses/inspection 处理检查事件

Try it out

Parameters

Responses

Name	Description
eventId <small>required</small> integer(int32) (query)	eventId
storageld <small>required</small> integer(int32) (query)	storageld
status <small>required</small> string (query)	status
remarks <small>required</small> string (query)	remarks

Code	Description	Links
200	OK	No links

Media type

/

Controls Accept header.

Example Value | Schema

```
{
  "additionalProp1": "string",
  "additionalProp2": "string",
  "additionalProp3": "string"
}
```

- 仓库存储

POST

http://localhost:8080/api/v1/warehouses/storage?eventId=12347&storageId=98765&status=STORED

Send

Params

Authorization

Headers (8)

Body

Pre-request Script

Tests

Settings

Cookies

Query Params

<input checked="" type="checkbox"/>	Key	Value	Description	...	Bulk Edit
<input checked="" type="checkbox"/>	eventId	12347			
<input checked="" type="checkbox"/>	storageId	98765			
<input checked="" type="checkbox"/>	status	STORED			
	Key	Value	Description		

Body

Cookies

Headers (5)

Test Results (3/3)

Status: 200 OK

Time: 17 ms

Size: 262 B

Save as example

Pretty

Raw

Preview

Visualize

JSON

```
1 {
2   "message": "Storage event processed: StorageEvent(eventId=12347, storageId=98765, status=STORED)"
3 }
```

POST

/api/v1/warehouses/storage 处理存储事件

Parameters

Try it out

Name	Description
eventId <small>required</small> integer(\$int32) (query)	<input type="text" value="eventId"/>
storageId <small>required</small> integer(\$int32) (query)	<input type="text" value="storageId"/>
status <small>required</small> string (query)	<input type="text" value="status"/>

Responses

Code	Description	Links
200	OK	No links

Media type

Controls Accept header.

Example Value | Schema

```
{
  "additionalProp1": "string",
  "additionalProp2": "string",
  "additionalProp3": "string"
}
```

- 库存管理

POST

http://localhost:8080/api/v1/warehouses/inventory?productId=prod123&quantity=150

Send

Params

Authorization

Headers (8)

Body

Pre-request Script

Tests

Settings

Cookies

Query Params

<input checked="" type="checkbox"/>	Key	Value	Description	...	Bulk Edit
<input checked="" type="checkbox"/>	productId	prod123			
<input checked="" type="checkbox"/>	quantity	150			
	Key	Value	Description		

Body

Cookies

Headers (5)

Test Results

Status: 200 OK

Time: 3 ms

Size: 220 B

Save as example

Pretty

Raw

Preview

Visualize

JSON

```
1 {
2   "message": "Updated inventory for product prod123: 150"
3 }
```

POST

/api/v1/warehouses/inventory 更新库存信息

Parameters

Try it out

Name	Description
productId <small>★ required</small> string (query)	<input type="text" value="productId"/>
quantity <small>★ required</small> integer(\$int32) (query)	<input type="text" value="quantity"/>

Responses

Code	Description	Links
200	OK	No links

Media type

Controls Accept header.

Example Value | Schema

```
{
  "additionalProp1": "string",
  "additionalProp2": "string",
  "additionalProp3": "string"
}
```

- 出货装载

POST

http://localhost:8080/api/v1/warehouses/shipment?eventId=12348&shipmentId=54323&status=SHIPPED

Send

Params

Authorization

Headers (8)

Body

Pre-request Script

Tests

Settings

Cookies

Query Params

<input checked="" type="checkbox"/>	Key	Value	Description	...	Bulk Edit
<input checked="" type="checkbox"/>	eventId	12348			
<input checked="" type="checkbox"/>	shipmentId	54323			
<input checked="" type="checkbox"/>	status	SHIPPED			
	Key	Value	Description		

Body

Cookies

Headers (5)

Test Results (3/3)

Status: 200 OK

Time: 58 ms

Size: 266 B

Save as example

...

Pretty

Raw

Preview

Visualize

JSON

...

...

1

2

3

{

"message": "Shipment event processed: ShipmentEvent(eventId=12348, shipmentId=54323, status=SHIPPED)"

}

POST

/api/v1/warehouses/shipment

处理发货事件

Try it out

Parameters

Name	Description
eventId ★ required integer(\$int32) (query)	eventId
shipmentId ★ required integer(\$int32) (query)	shipmentId
status ★ required string (query)	status

Responses

Code	Description	Links
200	OK	No links

Media type

/

Controls Accept header.

Example Value | Schema

{
 "additionalProp1": "string",
 "additionalProp2": "string",
 "additionalProp3": "string"
}

