University of St.Gallen - Exercise Submission

Course Information

- **Course:** Event-driven and Process-oriented Architectures FS2024
- **Instructors:** B. Weber, R. Seiger, A. Abbad-Andaloussi

Deadline

- **Submission Date:** 05.03.2024; 23:59 CET
- **Updated Document based on Feedback:** 10.03.2024; 20:00 CET
- **Work distribution**

Exercise 02: Kafka with Spring

Code

Release 1.0

The mailing service is located in kafka/java/mailing directory.

The README.md file provides detailed description of implementation.

Implementation of an Event Notification Service in an Event-Driven Architecture

Decision

We have decided to introduce a notification service within our event-driven architecture for the 'flowing-retail' application. This service will be responsible for sending email notifications to customers, informing them about key milestones in their order processing, such as when their order is completed.

Rationale

The decision to implement this feature as a listener for the 'OrderCompletedEvent' on the 'flowing-retail' topic is twofold:

- It aligns with the event-driven nature of our system, ensuring that notifications are triggered by actual business events.
- It decouples the notification concern from the core order processing logic, which adheres to the principle of single responsibility and facilitates independent scaling and maintenance.

Design

The mailing service is implemented as a Spring Boot application, leveraging the Spring Kafka library to interact with the Kafka broker. It is subscribed to the 'flowing-retail' topic and listens for 'OrderCompletedEvent' messages. The mailing service is implemented according to the EDA pattern Event Notification.

EventListener in the MessageListener "handleEvents":

- OrderPlacedEvent
- PaymentReceivedEvent
- GoodsShippedEvent
- OrderedCompletedEvent
- VGRFinishedEvent

Example for: "OrderPlacedEvent"

```
// look for the event type and call the corresponding method
try {
    switch (messageType) {
        case "OrderPlacedEvent": //step 1
            sendMailForOrderPlacedEvent(messageJson, messageType);
            break;
        ...
        default:
            System.out.println("Received unsupported event type: " + messageType);
    }
} catch (Exception e) {
    System.out.println("Error processing message" + e);
}

...

// send mail for OrderPlacedEvent
private void sendMailForOrderPlacedEvent(String messageJson, String messageType)
throws Exception {
    try {
        Message<OrderPlacedEventPayload> message =
objectMapper.readValue(messageJson, new
TypeReference<Message<OrderPlacedEventPayload>>() {});
        OrderPlacedEventPayload eventPayload = message.getData();

        String emailSubject = "Order has been placed and is ready for payment";
        String emailContent = "The order with ID " + eventPayload.getOrder() + "
has been shipped.";
        String recipient = eventPayload.getOrder().getCustomer(); // This should
be replaced with the actual recipient from the payload

        mailingService.sendMail(emailSubject, emailContent);
        System.out.println("Email sent for " + messageType + ": " + emailContent);
    } catch (Exception e) {
        System.out.println("Error " + Thread.currentThread().getStackTrace() + e);
    }
}
```

Upon detecting an event, it will extract the necessary customer information (which is limited to the name at this point) from the event payload and send an email notification using a predefined template that includes relevant order details. The use of a topic allows for a decoupling of the event producer and consumer, which

means that any part of the system interested in this event can listen to this topic without direct interaction with the order management component.

Additional Considerations

The service is designed to be extensible to accommodate additional event types for notification in the future, further leveraging the benefits of our event-driven architecture.