**Spring Boot-Kafka Transactions**

This document will discuss transaction management in Spring boot Kafka based message processing applications.

Transactions are enabled by providing the DefaultKafkaProducerFactory with a transactionIdPrefix. Property can be set using application.yml.

spring:

kafka:

consumer:

bootstrap-servers: localhost:9092

group-id: group\_id

auto-offset-reset: earliest

key-deserializer: org.apache.kafka.common.serialization.StringDeserializer

properties:

isolation.level: read\_committed

producer:

bootstrap-servers: localhost:9092

key-serializer: org.apache.kafka.common.serialization.StringSerializer

value-serializer: org.springframework.kafka.support.serializer.JsonSerializer

transaction-id-prefix: tx.

Any KafkaTemplate operations performed in a @KafkaListener method will participate in the transaction, and the listener container will send the offsets to the transaction before committing it. We also set the isolation level for the consumers to not have visibility into uncommitted records.

**Database and Kafka Transaction Synchronization:**

The KafkaTransactionManager is an implementation of Spring Framework’s PlatformTransactionManager. It is provided with a reference to the producer factory in its constructor.

We can use the KafkaTransactionManager with normal Spring transaction support (@Transactional, TransactionTemplate, and others). If a transaction is active, any KafkaTemplate operations performed within the scope of the transaction use the transaction’s Producer. The manager commits or rolls back the transaction, depending on success or failure.

If order to synchronize a Kafka transaction with some other transaction, configure the listener container with the appropriate transaction manager (one that supports synchronization, such as the MongoTransactionManager). Any operations performed on a transactional KafkaTemplate from the listener participate in a single transaction. The Kafka transaction is committed (or rolled back) immediately after the controlling transaction.

The ChainedKafkaTransactionManager is a subclass of ChainedTransactionManager that can have exactly one KafkaTransactionManager. Since it is a KafkaAwareTransactionManager, the container can send the offsets to the transaction in the same way as when the container is configured with a simple KafkaTransactionManager. This provides a mechanism for synchronizing transactions without having to send the offsets to the transaction in the listener code. We should chain our transaction managers in the desired order and provide the ChainedTransactionManager in the ContainerProperties.(Reference: <https://docs.spring.io/spring-kafka/reference/html/#transactions>)

@Value("${app.retry.attempts}")

**private** **int** retryAttempts;

@Value("${app.retry.interval}")

**private** **int** retryInterval;

@Value("${app.retry.topic}")

**private** String retryTopic;

@Value("${app.dlt.topic}")

**private** String dltTopic;

@Bean

**public** ChainedKafkaTransactionManager<Object, Object> chainedTm(

KafkaTransactionManager<String, String> ktm,

MongoTransactionManager mdtm) {

**return** **new** ChainedKafkaTransactionManager<>(ktm, mdtm);

}

@Bean(name = "mongoTransactionManager")

@Primary

MongoTransactionManager mongoTransactionManager(MongoDatabaseFactory dbFactory) {

**return** **new** MongoTransactionManager(dbFactory);

}

@Bean

**public** ConcurrentKafkaListenerContainerFactory<Object, Object> kafkaListenerContainerFactory(

ConcurrentKafkaListenerContainerFactoryConfigurer configurer,

ConsumerFactory<Object, Object> kafkaConsumerFactory,

KafkaTemplate<Object, Object> template

, ChainedKafkaTransactionManager<Object, Object> chainedTM

, ObjectMapper objectMapper

)

{

ConcurrentKafkaListenerContainerFactory<Object, Object>

factory = **new** ConcurrentKafkaListenerContainerFactory<>();

configurer.configure(factory, kafkaConsumerFactory);

factory.getContainerProperties().setEosMode(ContainerProperties.EOSMode.*valueOf*(eosMode));

factory.getContainerProperties().setSubBatchPerPartition(subBatchPerPartition);

factory.getContainerProperties().setTransactionManager(chainedTM);

factory.setAfterRollbackProcessor(**new** DefaultAfterRollbackProcessor<Object, Object>((record, exception) -> {

template.executeInTransaction(kTemplate ->{

**try** {

kTemplate.send(retryTopic, objectMapper.readValue(record.value().toString(), Object.**class**));

} **catch** (JsonMappingException e) {

logger.error(e.getMessage());

} **catch** (JsonProcessingException e) {

logger.error(e.getMessage());

} **catch** (Exception e) {

logger.error(e.getMessage());

}

**return** **true**;

});

}, **new** FixedBackOff(0L, 0L)));//A simple BackOff implementation that provides a configured interval between two attempts and a configured number of retries.

logger.info(String.*format*("KafkaTemplate.transactionIdPrefix: %s - producerPerConsumerPartition: %s" +

" - ConcurrentKafkaListenerContainerFactory EOS Mode: %s - subBatchPerPartition: %s ",

**this**.kafkaTemplate.getTransactionIdPrefix()

, **this**.kafkaTemplate.getProducerFactory().isProducerPerConsumerPartition(),

factory.getContainerProperties().getEosMode()

, factory.getContainerProperties().getSubBatchPerPartition()

));

**return** factory;

}

@KafkaListener(topics = "#{'${app.consumer.subscribed-to.topic}'.split(',')}", containerFactory="kafkaListenerContainerFactory", groupId = "${spring.kafka.consumer.group-id}", properties="${app.consumer.props}")

**public** **void** consume(EventData eventData) **throws** Exception {

logger.info(String.*format*("Consuming message: %s - KafkaTemplate.transactionIdPrefix: %s ", eventData, **this**.kafkaTemplate.getTransactionIdPrefix()));

eventDataService.createEvent(eventData);

**this**.kafkaTemplate.send(topicToPublish,eventData);

}

We use executeInTransaction method of KafkaTemplate so that the operations are invoked within a local transaction.

**public** **void** sendEventMessage(String topic, List<EventData> events) {

standaloneTransactionKafkaTemplate.executeInTransaction(kTemplate -> {

**for** (EventData eventData : events)

{

logger.info(String.*format*("Producing message: %s - TransactionIdPrefix: %s",

eventData.getDesc(), kafkaTemplate.getTransactionIdPrefix()));

kTemplate.send(topic, eventData);

}

**return** **null**;

});

}

Rest Controller: -

@RestController

@RequestMapping(value = "/kafka")

**public** **class** Controller {

@Value("${app.consumer.subscribed-to.topic}")

**private** String topicToPublish;

**private** **final** EventProcessor eventProcessor;

@Autowired

Controller(EventProcessor eventProcessor) {

**this**.eventProcessor = eventProcessor;

}

@PostMapping("/send/message")

**public** ResponseEntity<Object> sendEventMessage(

@RequestBody List<EventData> events) {

**this**.eventProcessor.sendEventMessage(topicToPublish, events);

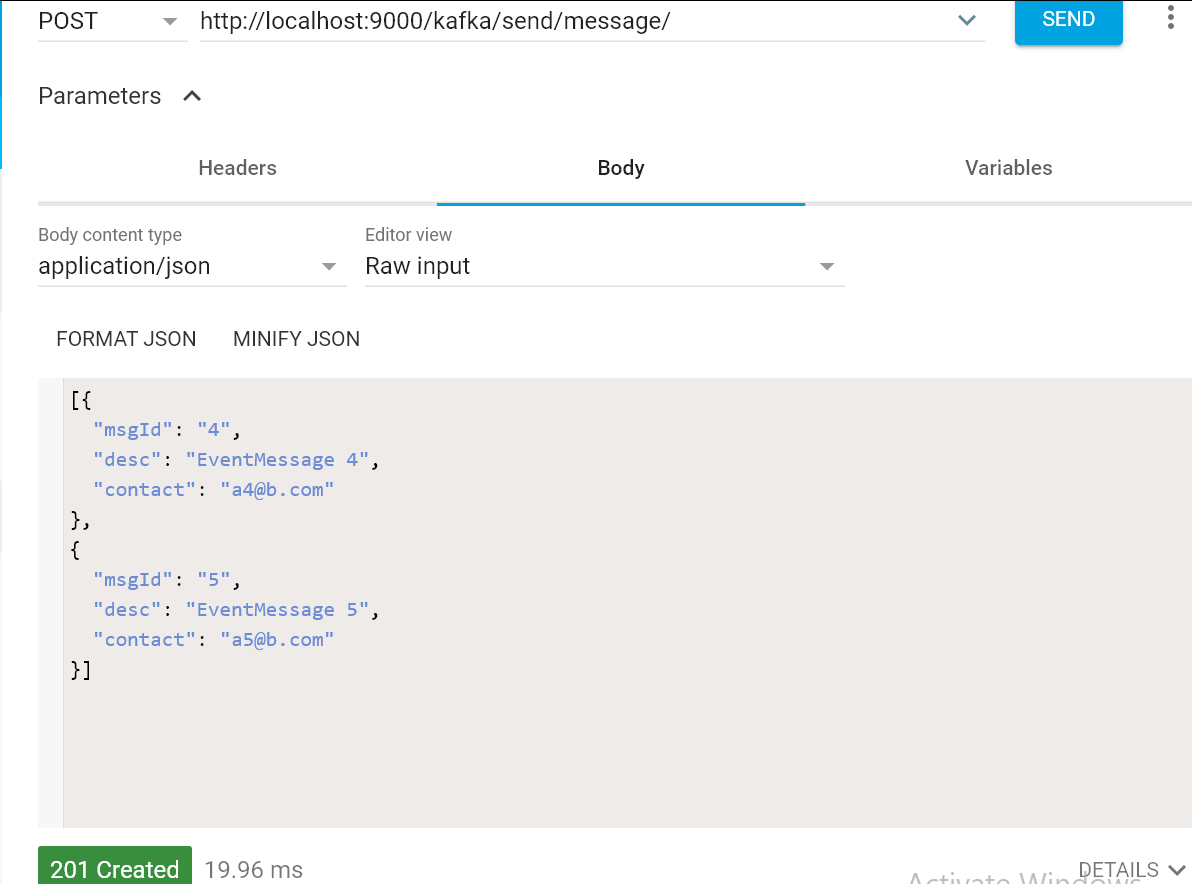
**return** **new** ResponseEntity<>("Event is stored successfully", HttpStatus.***CREATED***);

}

}

Our producer sends multiple records in a single transaction.

If we run our Spring Boot application, we can invoke rest resource http://localhost:9000/ kafka/send/message/ using the Post method.



If the message has invalid email address, processing service in our consumer will throw DataIntegrityViolationException, and the transaction will be rolled back.

**AfterRollbackProcessor**

By default, any unprocessed records (including the failed record) are re-fetched on the next poll. This is achieved by performing seek operations in the DefaultAfterRollbackProcessor. To modify this behavior, we can configure the listener container with a custom AfterRollbackProcessor. For example, we can configure our processor with a custom recoverer (BiConsumer) that sends failed messages to a retry topic.

By default, after ten failures, the failed record is logged (at the ERROR level).

We can also configure a separate listener container for the retry topic consumer. Here, we can configure a custom recoverer that sends failed messages to a dead letter topic after maximum attempts, as configured in the property file.

@Bean

public ConcurrentKafkaListenerContainerFactory<Object, Object> kafkaRetryListenerContainerFactory (

ConcurrentKafkaListenerContainerFactoryConfigurer configurer,

ConsumerFactory<Object, Object> kafkaConsumerFactory,

KafkaTemplate<Object, Object> template

, ChainedKafkaTransactionManager<Object, Object> chainedTM

,ObjectMapper objectMapper

)

{

ConcurrentKafkaListenerContainerFactory<Object, Object>

factory = new ConcurrentKafkaListenerContainerFactory<>();

configurer.configure(factory, kafkaConsumerFactory);

factory.getContainerProperties().setTransactionManager(chainedTM);

factory.setAfterRollbackProcessor(new DefaultAfterRollbackProcessor<Object, Object>((record, exception) -> {

template.executeInTransaction(kTemplate ->{

try {

kTemplate.send(dltTopic, objectMapper.readValue(record.value().toString(), Object.class));

} catch (JsonMappingException e) {

logger.error(e.getMessage());

} catch (JsonProcessingException e) {

logger.error(e.getMessage());

} catch (Exception e) {

logger.error(e.getMessage());

}

return true;

});

}, new FixedBackOff(retryInterval, retryAttempts)));//A simple BackOff implementation that provides a configured interval between two attempts and a configured number of retries.

return factory;

}

@KafkaListener(topics = "${app.retry.topic}", containerFactory="kafkaRetryListenerContainerFactory", groupId = "${app.consumer.group-id}")

**public** **void** retry(EventMessage eventMessage) **throws** Exception {

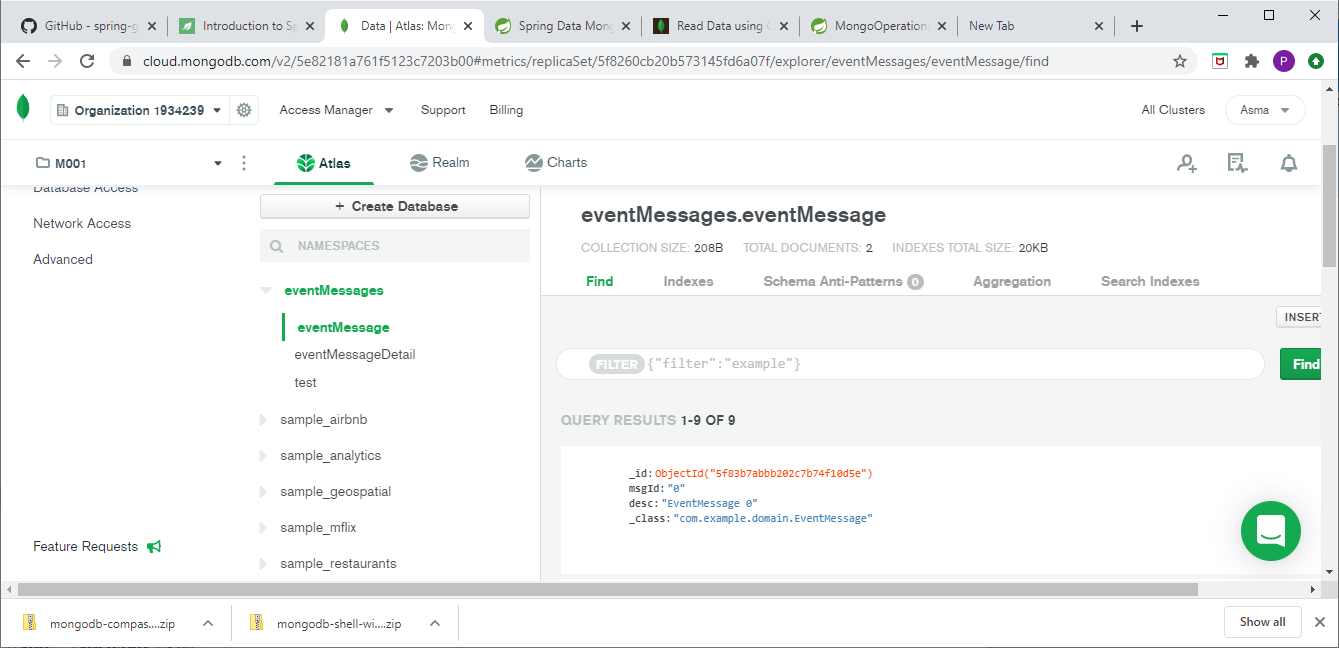
logger.info(String.*format*("Recieved Message in Retry: %s", eventMessage));

EventMessage msg = **new** EventMessage(eventMessage.getDescription()+"");

eventMessageService.insert(msg);

**this**.kafkaTemplate.send(topicToPublish,eventMessage);

}



Source code of this demo.