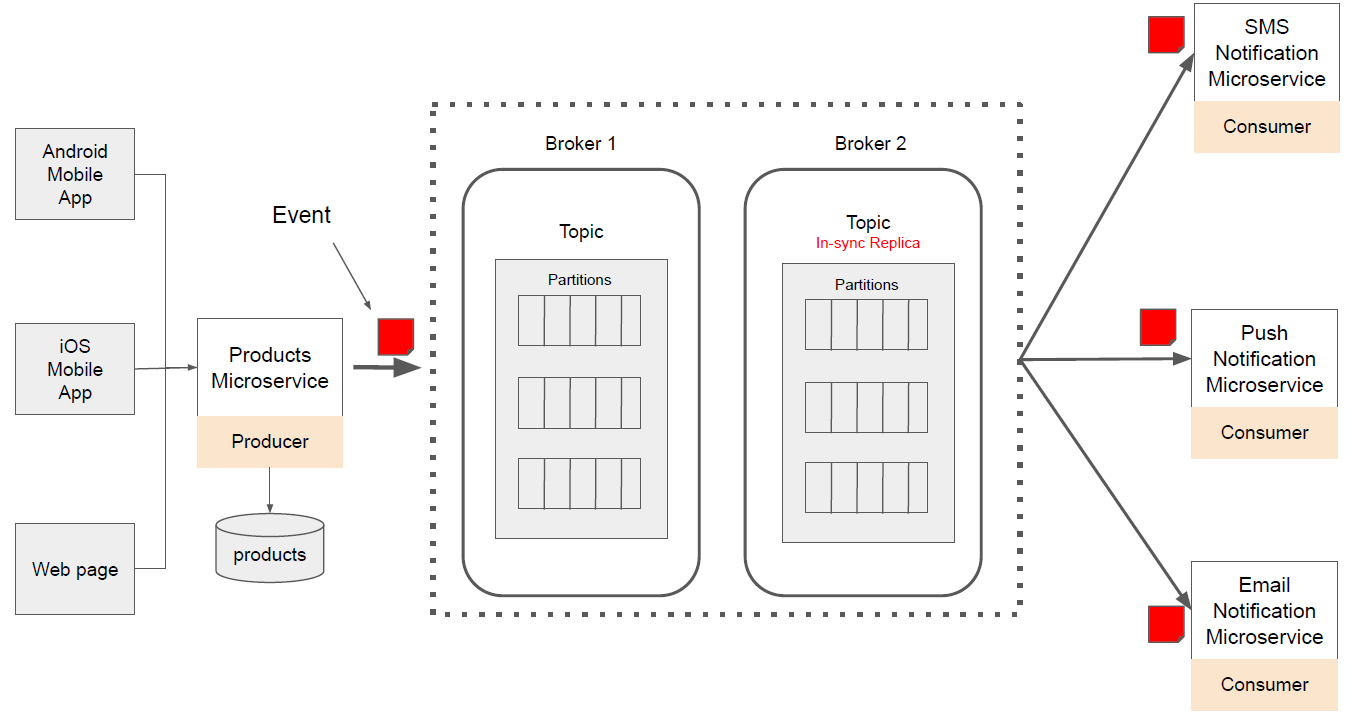
Kafka

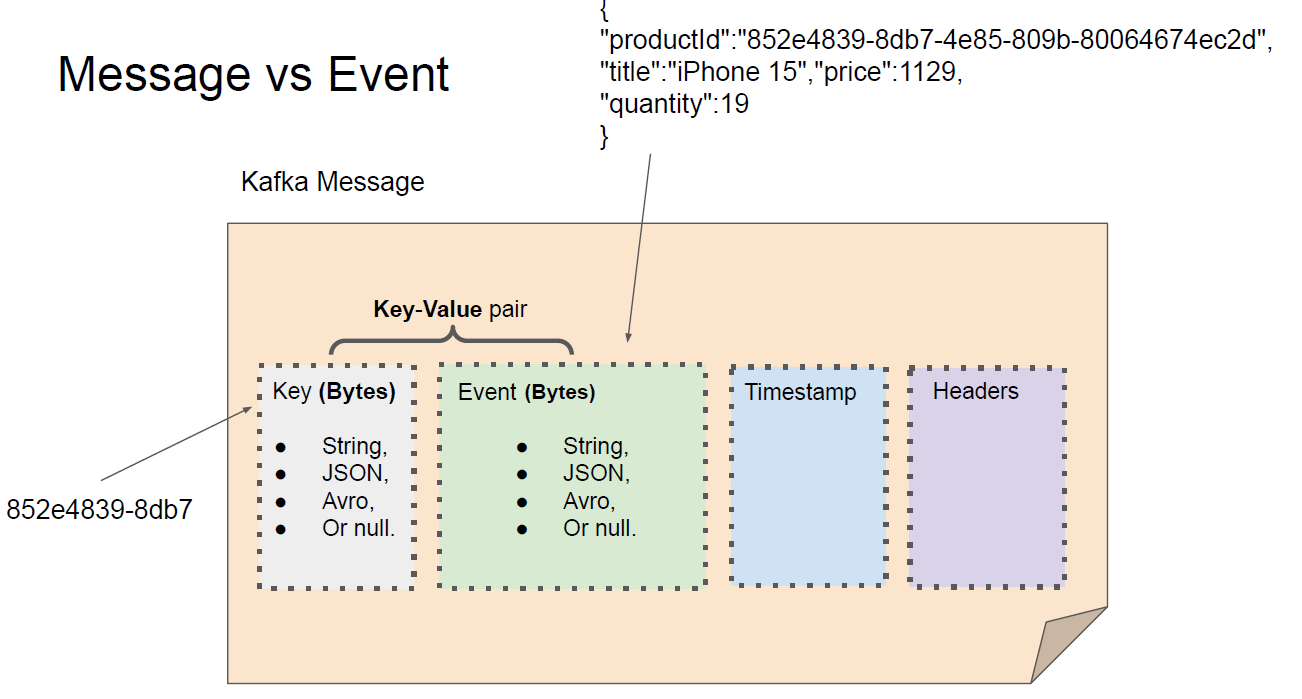
Kafka, also known as **Apache Kafka**, is an open-source, distributed event streaming platform designed to handle real-time data feeds. It is widely used for building real-time data pipelines and streaming applications.

**Section :1**

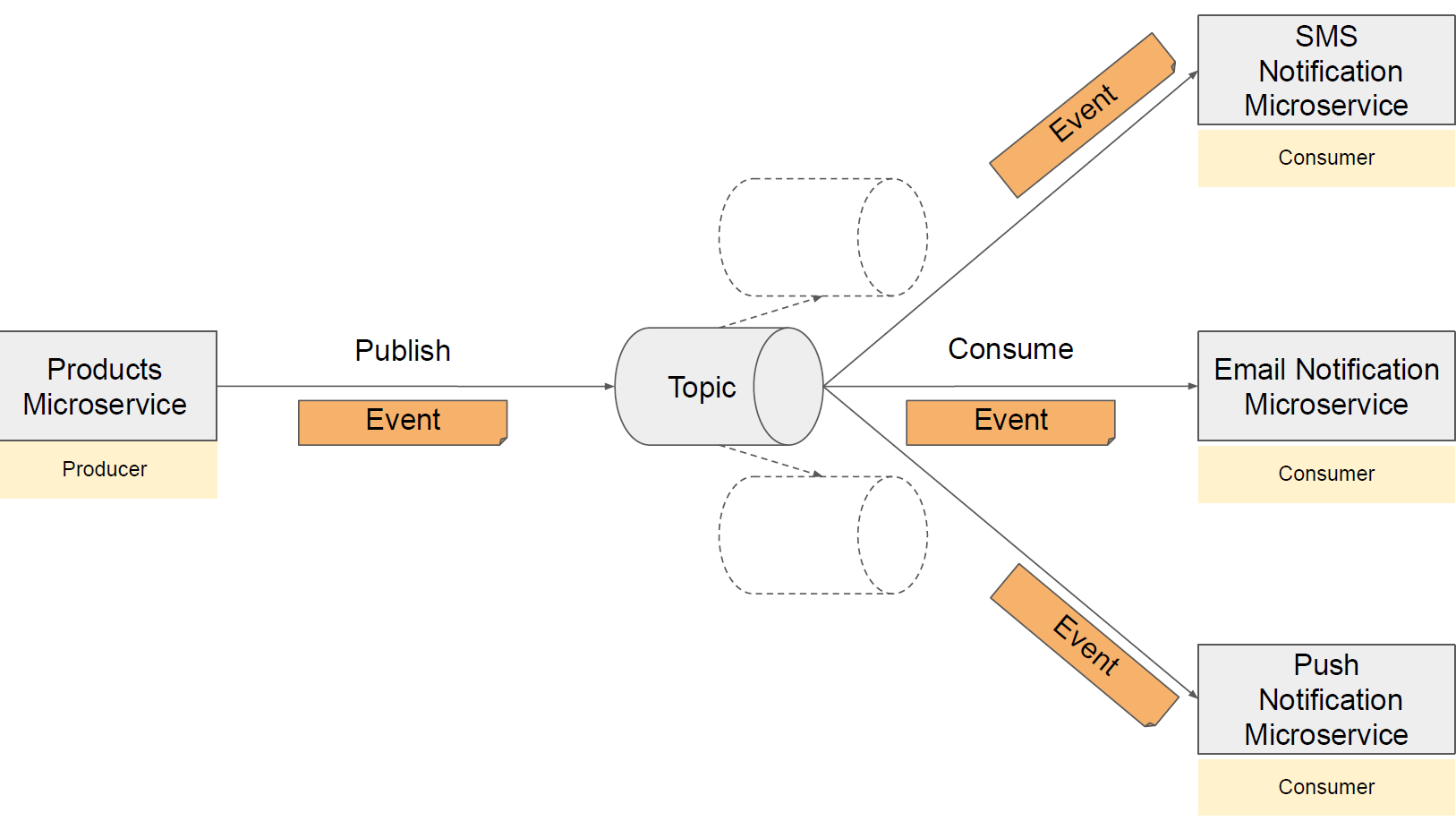
1. **Messages and Events**

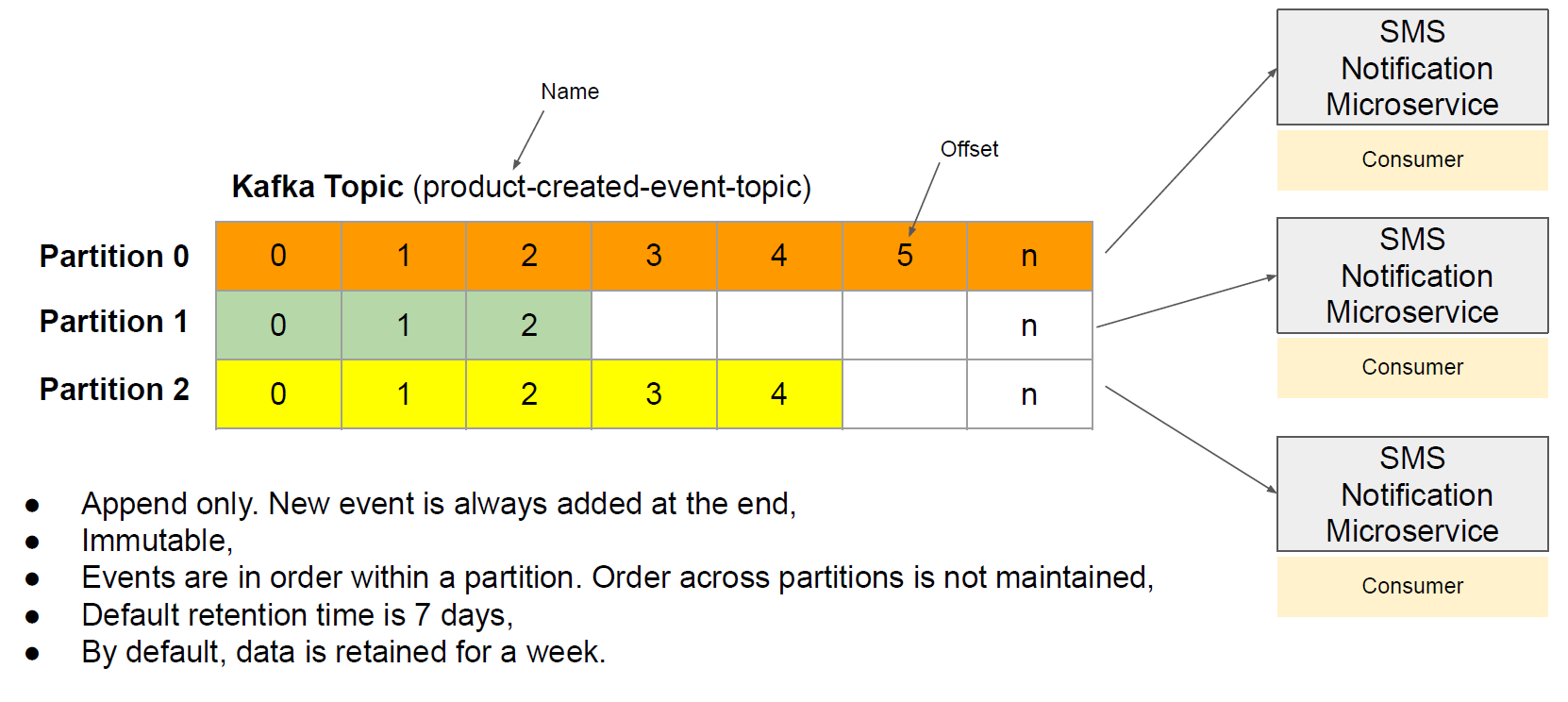
****

What is an Event? In Apache Kafka, an event is an indication that something has happened. ● UserLoggedInEvent ● ProductCreatedEvent, ● OrderCreatedEvent.

****

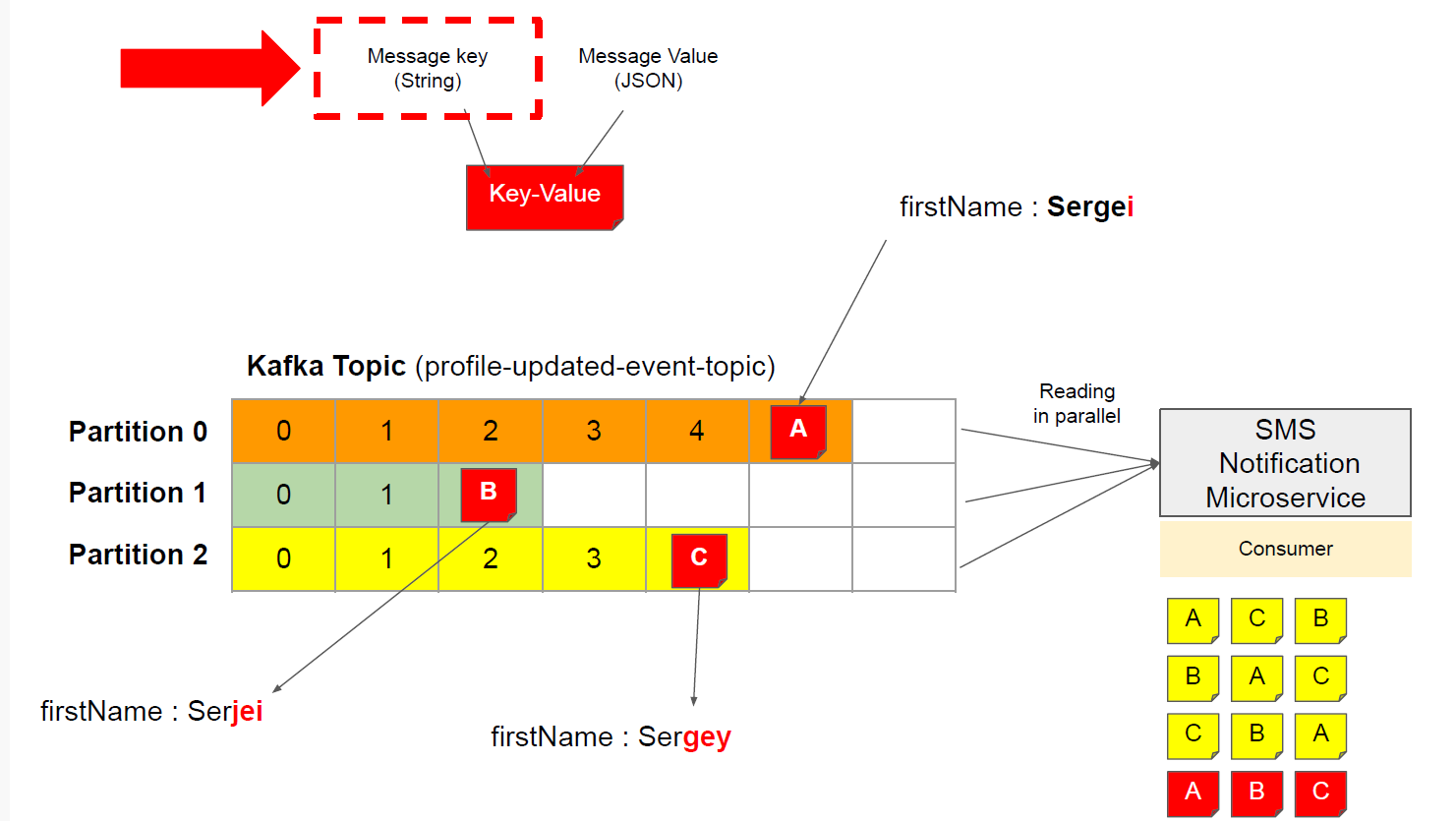
1. **Kafka Topic**

****

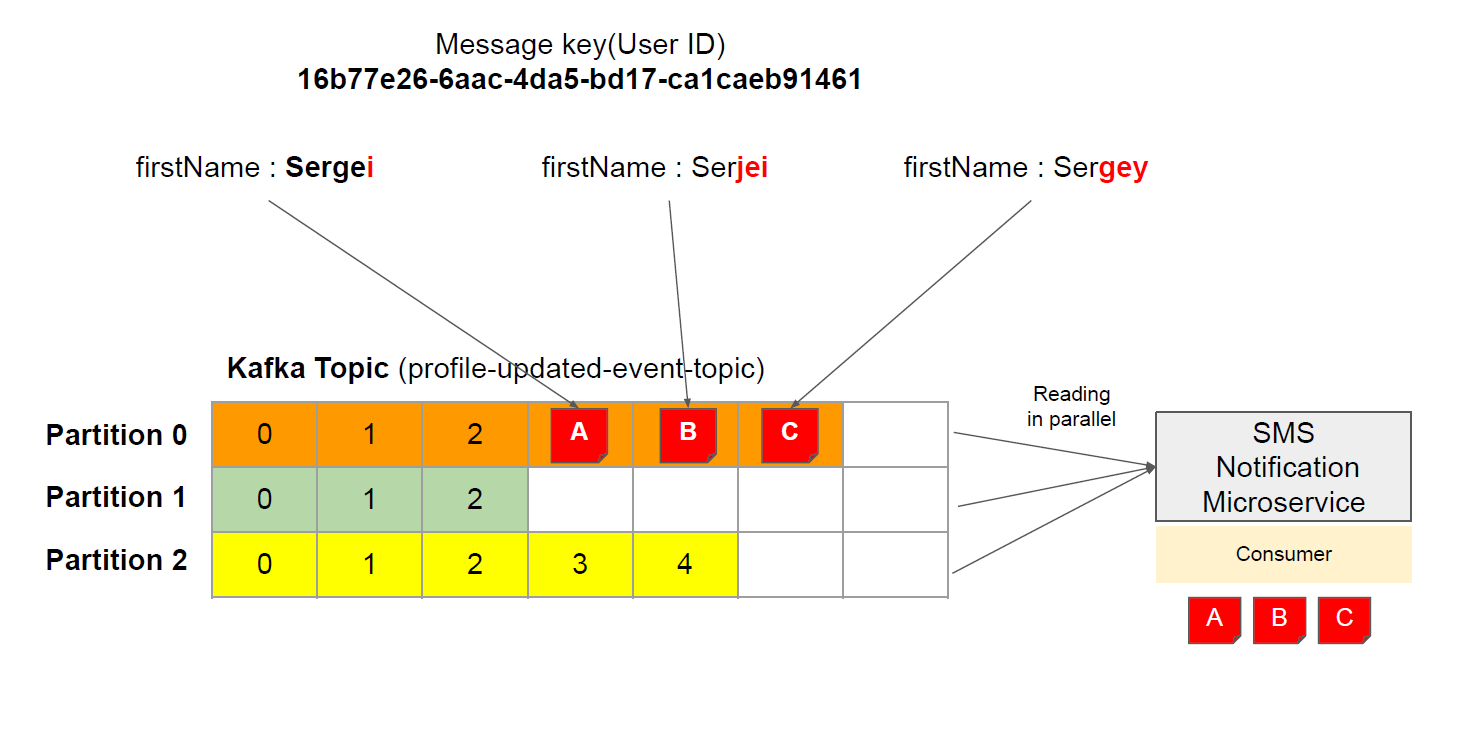


**Note** : topic is logical entity where as partion actually exist.

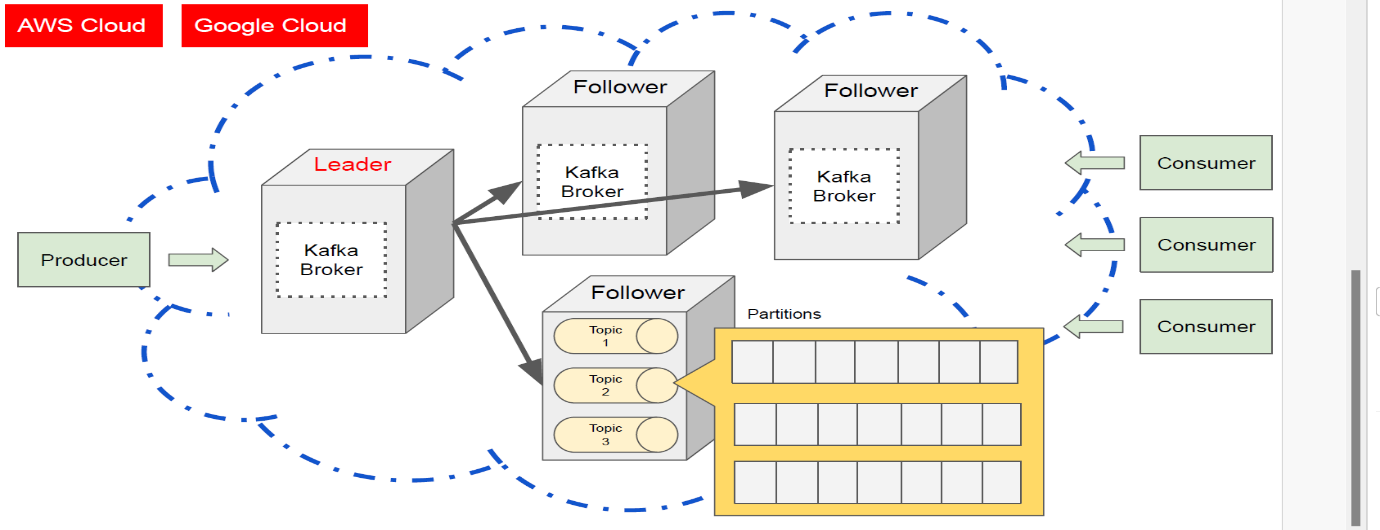
1. **Events Ordering**
2. If key is not present in kafka msg -> order is consumption in not gurrantte



1. If key is present in kafka msg -> order of consumption is gurrantted

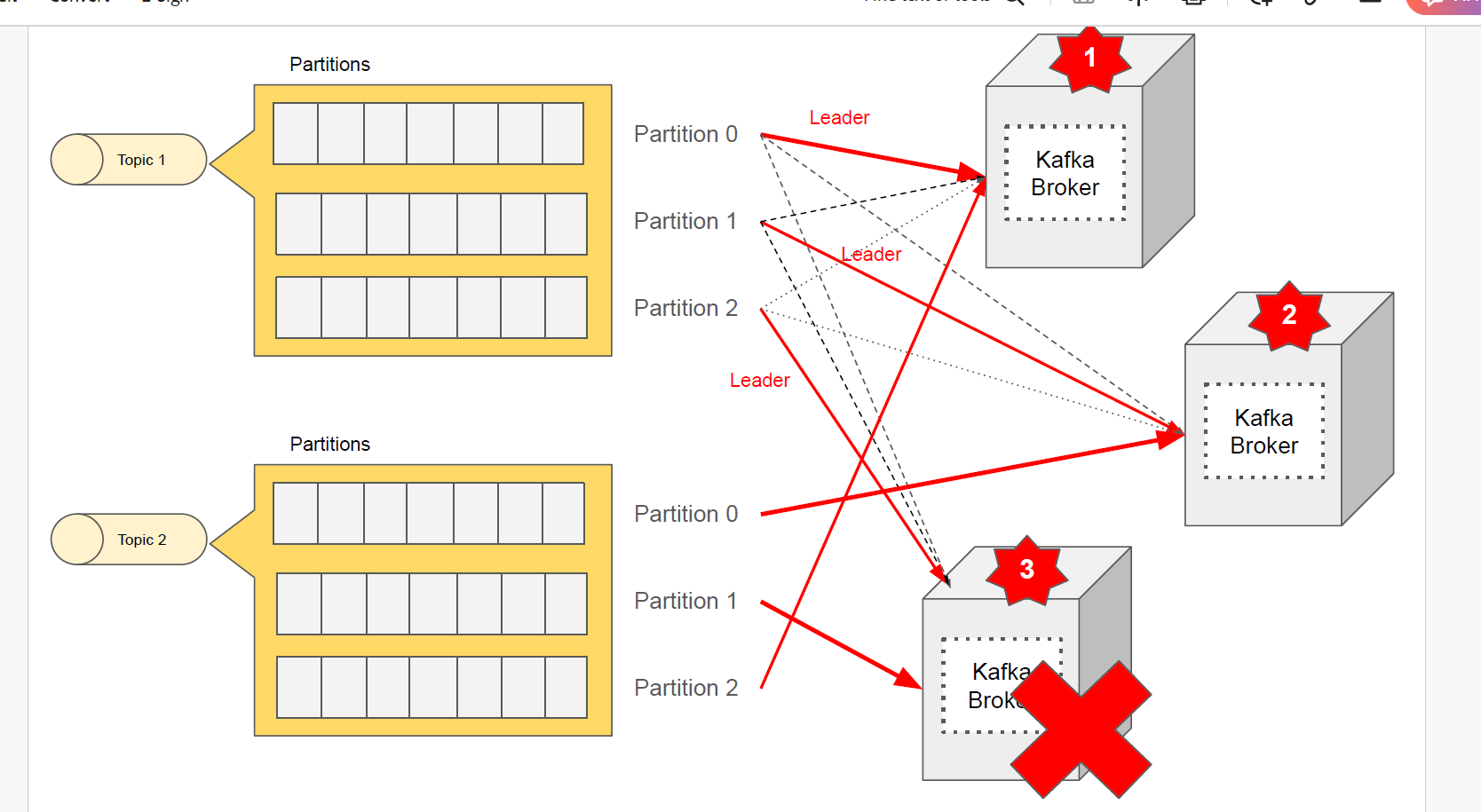
****

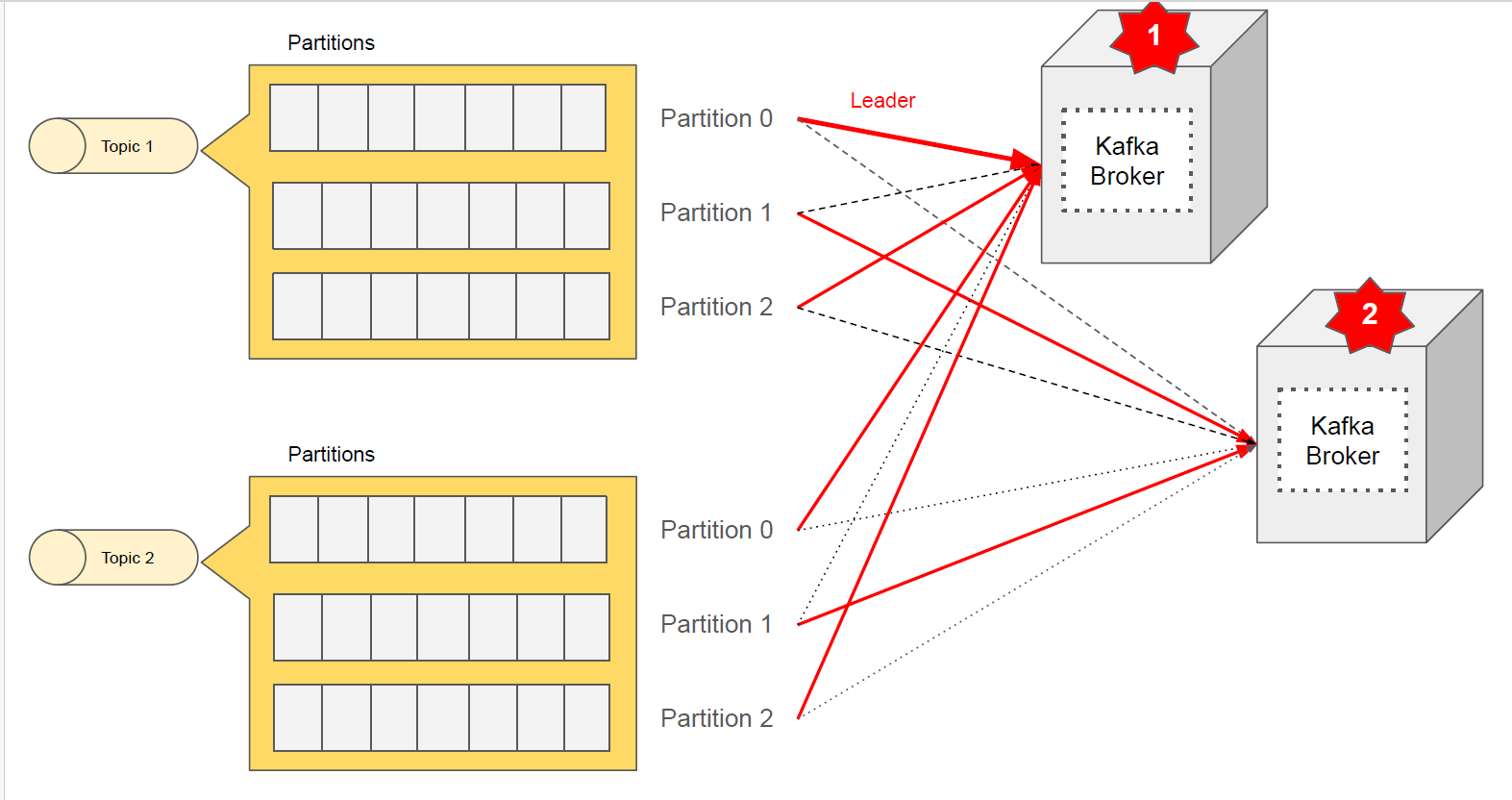
**Kafka broker**

**Section :2**

****

**In kafka leadership getting shifted or getting when required and hance it avoids bottleneck for the same**

**Image shows each partion working with other broker as leader**

**it gives durability due to replicating msg across broker**

**when broker is lost leadership managed by kafka**

**2 ways to start apache kafka**

1. **Zookeeper**
2. **Kraft -> kafka raft -> it s protocol**

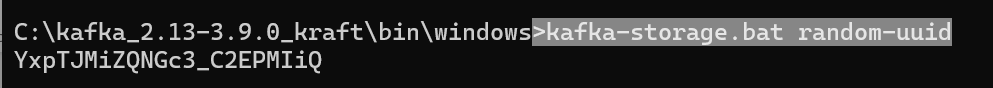
**Kraft**

**C:\kafka\_2.13-3.9.0\_kraft\config\kraft**

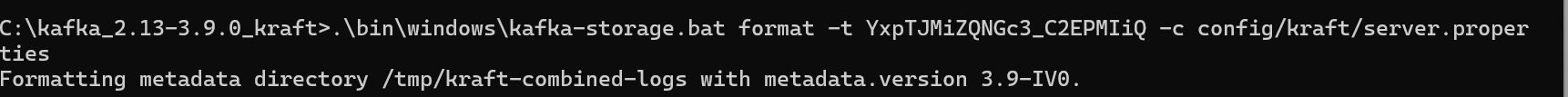
**3 properties file:**

1. **Broker.properties ->contains configuration for server which acts as broker which is responsible storing and serving data from partitions.**
2. **Controller.properties-. contains data for server that acts as server which is responsible managing cluster metadata and co-ordinating leader election for partitions**
3. **Server.properties -> stores data for both controller and broker**

**Step 1: Generate a Cluster UUID**

****

**Step 2: Format Log Directories**

****

**Step 3: Start the Kafka Server**

****

**What are log directories?**

**Log directories are the folders where Kafka stores its data as ordered sequences of messages called logs. Each log is divided into smaller files called segments, which contain the actual messages and some index files to help locate them. Log directories are organized by topics and partitions, which are logical groups of logs that belong to a specific use case and provide parallelism and fault tolerance.**

**Why do we need to format log directories?**

**We need to format log directories before we can start Kafka in KRaft mode. Formatting log directories means assigning a unique identifier to the cluster and initializing the metadata for the partitions. This prevents accidental configuration changes and ensures compatibility with KRaft mode.**

**What is server.properties file?**

**The server.properties is a configuration file that specifies various settings for the Kafka server, such as the broker id, the listeners, the log directories, the number of controllers, the ports, and the log retention policies. You can modify this file to change the default values according to your needs and preferences. You can also override some of the properties by passing them as command-line arguments when starting the Kafka server.**

**Step 4: Test the Kafka Server**

**# Create a topic named product-created-events-topic with one partition and one replica**

bin/kafka-topics.sh --create --topic product-created-events-topic --partitions 3 --replication-factor 1 --bootstrap-server localhost:9092

**# Produce some messages to the topic**

bin/kafka-console-producer.sh --topic product-created-events-topic --bootstrap-server localhost:9092

**# Consume the messages from the topic**

bin/kafka-console-consumer.sh --topic product-created-events-topic --from-beginning --bootstrap-server localhost:9092

Starting more than 1 kafka server:

1. Need to have 3 server.properties

Need to change below property to unique data

**a)node.id**

**b)** **listeners :pair of ports**

**b) advertised.listener :**

**c) controller.quorum.voters**

**e)** **log.dirs -> to maintain logs for each broker**

**Kafka Cli Topics**

**Section 3:**

**bin/kafka-topics.sh --create --topic topic1 --partitions 3 --replication-factor 3 --bootstrap-server localhost:9092**

**What is an Apache Kafka Topic?**

**A Kafka Topic is a logical collection of messages that are produced and**

**consumed by Kafka applications.**

**A Kafka Topic has a unique name that identifies it in a Kafka cluster. A Kafka cluster is a group of servers, called brokers, that store and manage the topics and messages. A Kafka application can connect to a Kafka cluster and interact with the topics and messages.**

**What is Topic Partition?**

**A Kafka Topic is divided into one or more partitions, each of which stores a subset of messages in an ordered sequence. A partition is a physical unit of storage and processing in a Kafka broker. Each partition has a unique identifier, called a partition ID, that is assigned by the broker.**

**The number of partitions affects the scalability, parallelism, fault-tolerance, and ordering guarantees of a topic.**

**● Scalability: The more partitions a topic has, the more messages it can store and handle. A topic with many partitions can be distributed across multiple brokers, which increases the storage capacity and performance of the topic.**

**● Parallelism: The more partitions a topic has, the more producers and consumers can interact with it concurrently. A topic with many partitions can have multiple producers sending messages to ifferent partitions, and multiple consumers receiving messages**

**from different partitions. This increases the throughput and efficiency of the topic.**

**● Fault-tolerance: The more partitions a topic has, the more resilient it is to failures. A topic with many partitions can have replicas, which are copies of the partitions stored on different brokers. If a broker fails, the replicas can take over and continue serving the messages. This increases the availability and reliability of the topic.**

**● Ordering guarantees: The more partitions a topic has, the less strict the ordering guarantees are. A topic with many partitions can only guarantee the order of messages within each partition, but not across partitions. This means that messages from different partitions may be delivered out of order to the consumers. This may or may not be acceptable, depending on the use case of the topic.**

**How to create topic ?**

**To create a topic, we need to specify the following parameters:**

**● --bootstrap-server: The address of one or more brokers in the**

**Kafka cluster that we want to connect to. For example,**

**localhost:9092.**

**● --create: The flag that indicates that we want to create a topic.**

**● --topic: The name of the topic that we want to create. For**

**example, product-created-events-topic.**

**● --partitions: The number of partitions that we want to create**

**for the topic. For example, 3.**

**● --replication-factor: The number of replicas that we want to**

**create for each partition of the topic. For example, 2.**

**For example, the following command creates a topic named**

**product-created-events-topic with 3 partitions and 2 replicas in a**

**Kafka cluster with a broker at localhost:9092:**

kafka-topics.sh --bootstrap-server localhost:9092 --create --topic

product-created-events-topic --partitions 3 --replication-factor 2

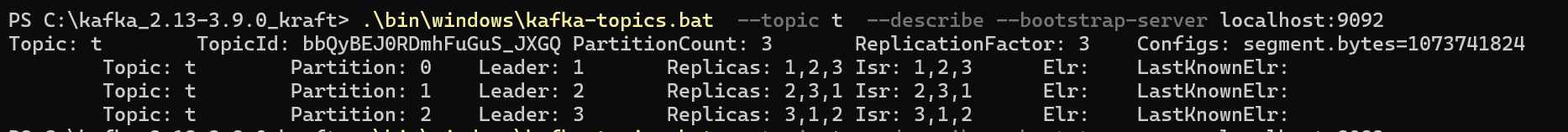
If the topic is created successfully, the tool will print a confirmation message, such as:

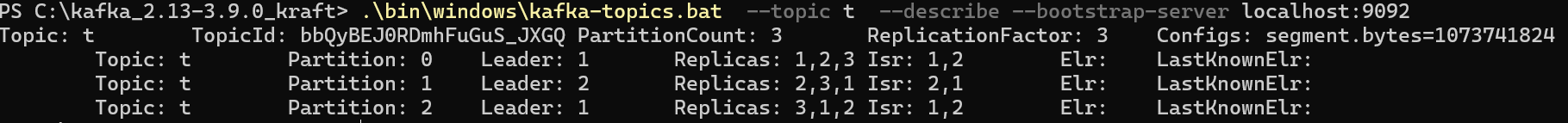
Created topic product-created-events-topic.

If the topic already exists, or if there is an error, the tool will print an error message, such as:

Topic 'product-created-events-topic' already exists.

Case : 3 brokers up : 3 partition: 3 replication



I shutdown my 1 broker that is server 3 after leader will be changes  


…………………………………………………..XXXXXXXXXXXXXXXXXXXX……………………………………..

**How to create topic with partitions?**

kafka-topics.sh --bootstrap-server localhost:9092 --create –topic topic1 --**partitions** 3

………………………………………………..XXXXXXXXXXXXXXXXXXXXXXX…………………………………

**How to update topic?**

kafka-topics.sh --bootstrap-server localhost:9092 --**alter** –topic topic1 --config retention.ms=43200000

**updating topic partition:**

kafka-topics.sh --bootstrap-server localhost:9092 --**alter** –topic topic1 --partitions 6

**updating replication factor:**

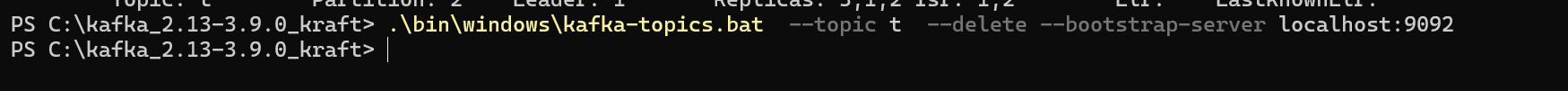
kafka-topics.sh --bootstrap-server localhost:9092 --**alter** –topic topic1 --replication-factor 3

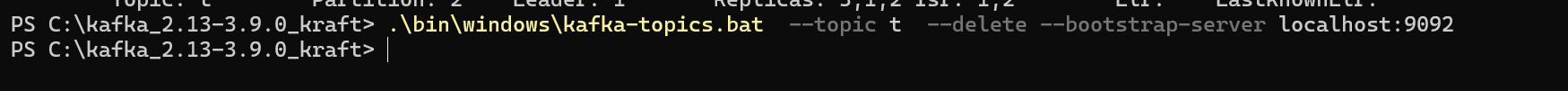
**Note**: Note that you can only increase the replication factor for a topic, but not decrease it. This is because reducing the replication factor would cause data loss and inconsistency. If you want to reduce the replication factor for a topic, you have to delete the topic and recreate it with the desiredreplication factor.

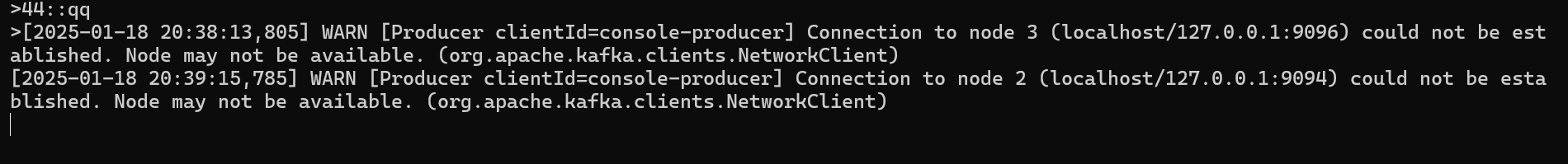
…………………………………..XXXXXXXXXXXXXXXXXXXXXXX…………………………………

**How to delete topic?**

kafka-topics.sh --bootstrap-server localhost:9092 --delete –topic topic1

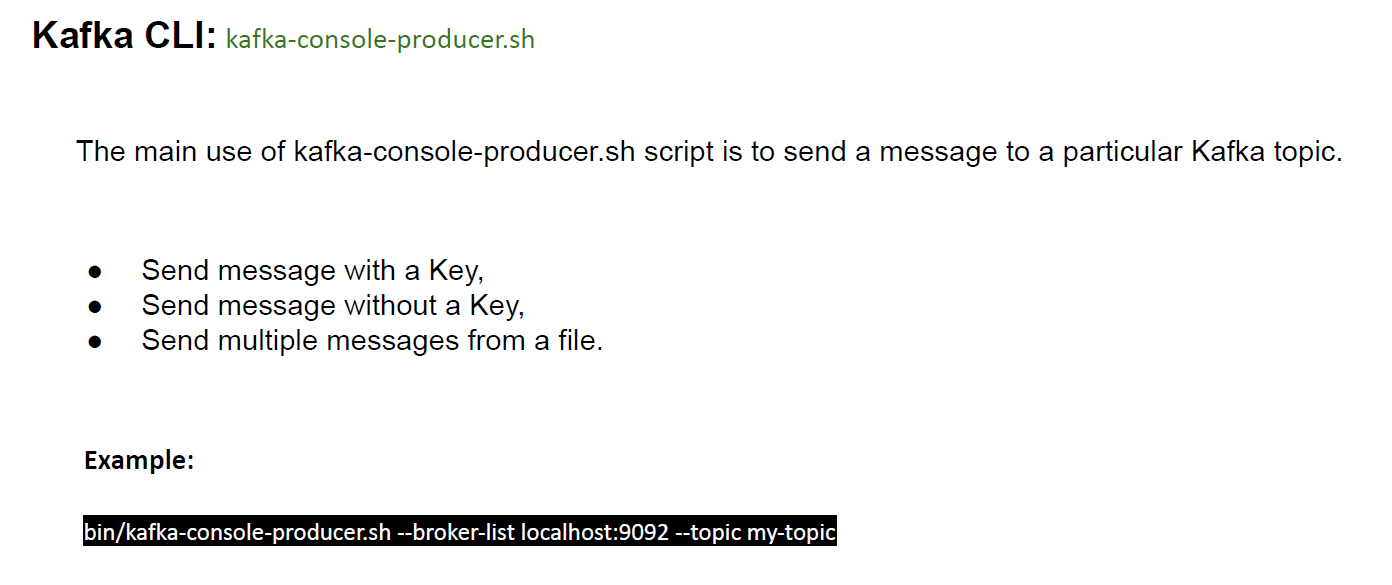
  
after deleting producer and consumer leasning to this topics will start throwing exception:





**Kafka Producer**

Section 4



**How to produce messages to Kafka topics without a key?**

Key insures sequence gurantte of msg consumption

./kafka-console-producer.sh --bootstrap-server localhost:9092 --topic topic1

**To consume msg from consumer:**

./kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic topic --from-beginning

………………………………………….XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX…………………………………………….

**How to produce messages to Kafka topics with a key**

./kafka-console-producer.sh --bootstrap-server localhost:9092 --topic topic1 **--property "parse.key=true"**

**--property "key.separator=:"**

**To consume msg from consumer:**

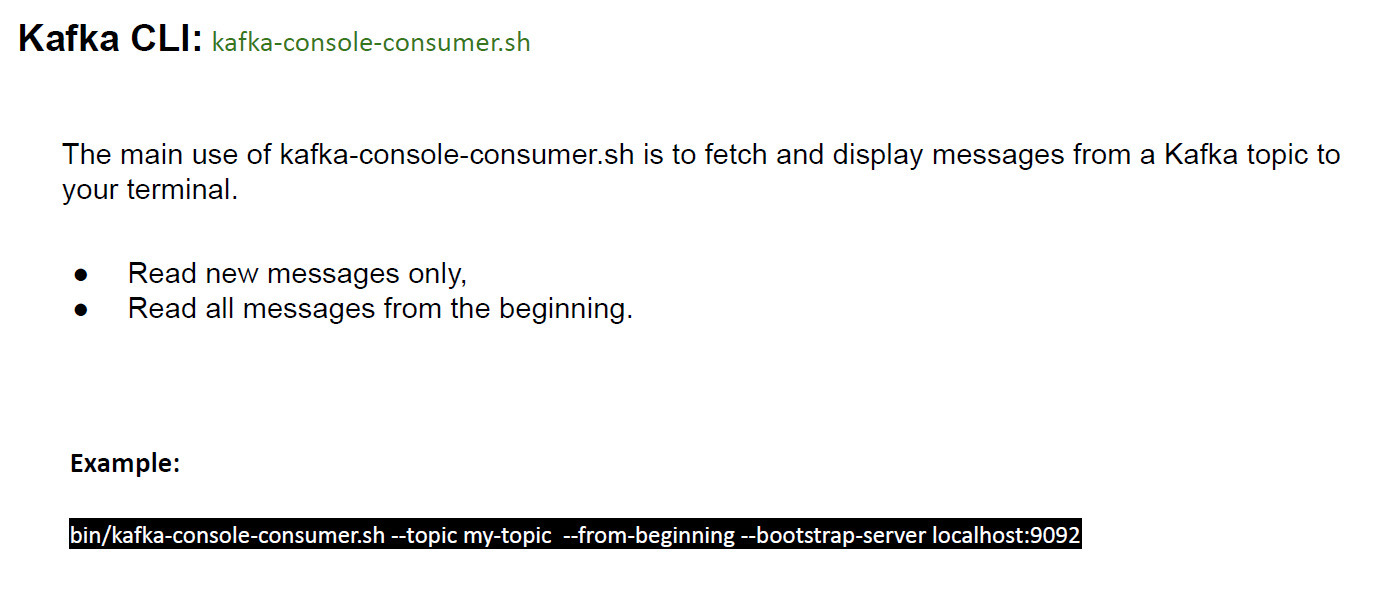
./kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic topic1 --from-beginning **--property**

**"print.key=true" --property "key.separator=:"**

**Note :**The print.key=true property tells the consumer to print both the key and the value of each message, separated by the key.separator character.

**Kafka Consumer**

**Section 5:**

****

We have 2 different properties for each consumer and producer as well  
For Producer:

**--property "parse.key=true"**

**--property "key.separator=:"**

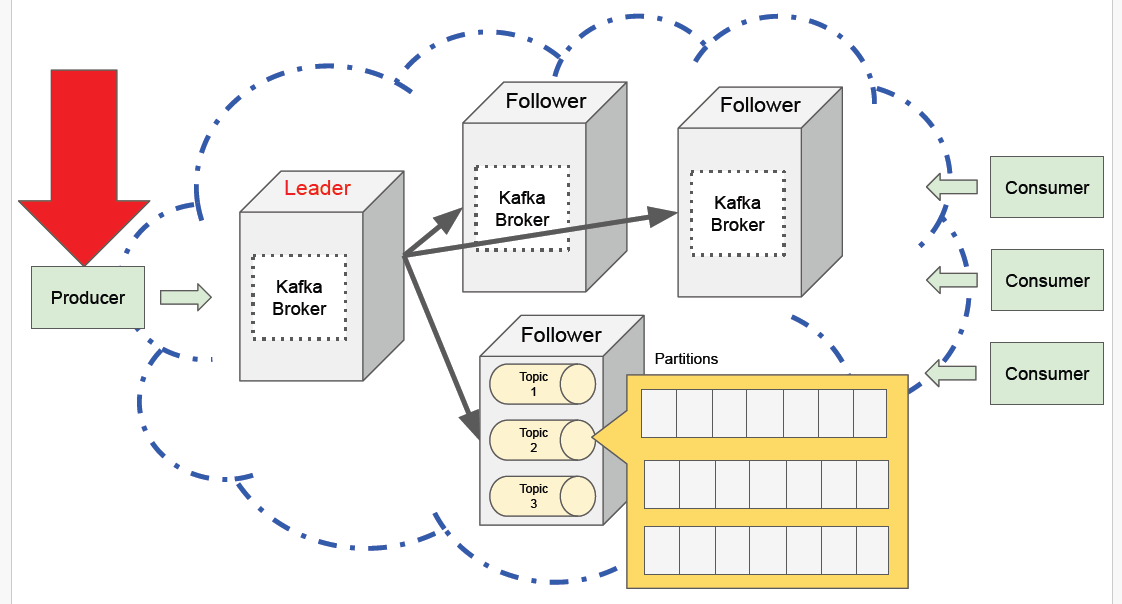
For consumer:

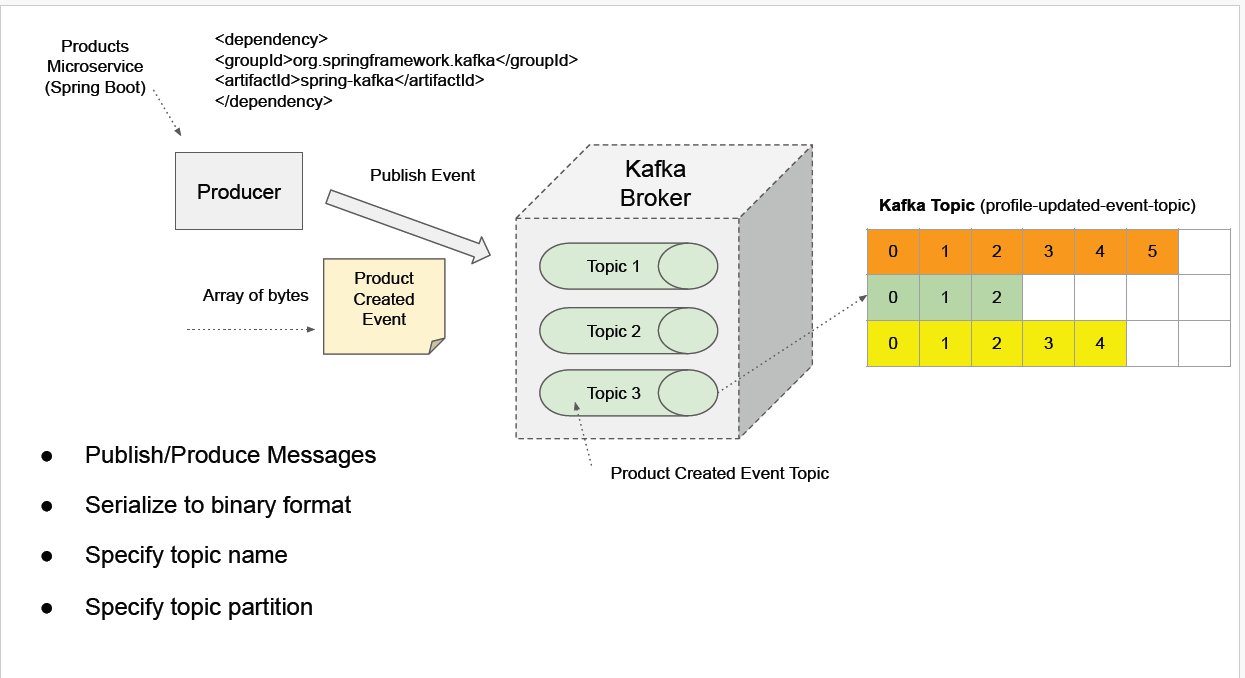
**--property print.key=true**

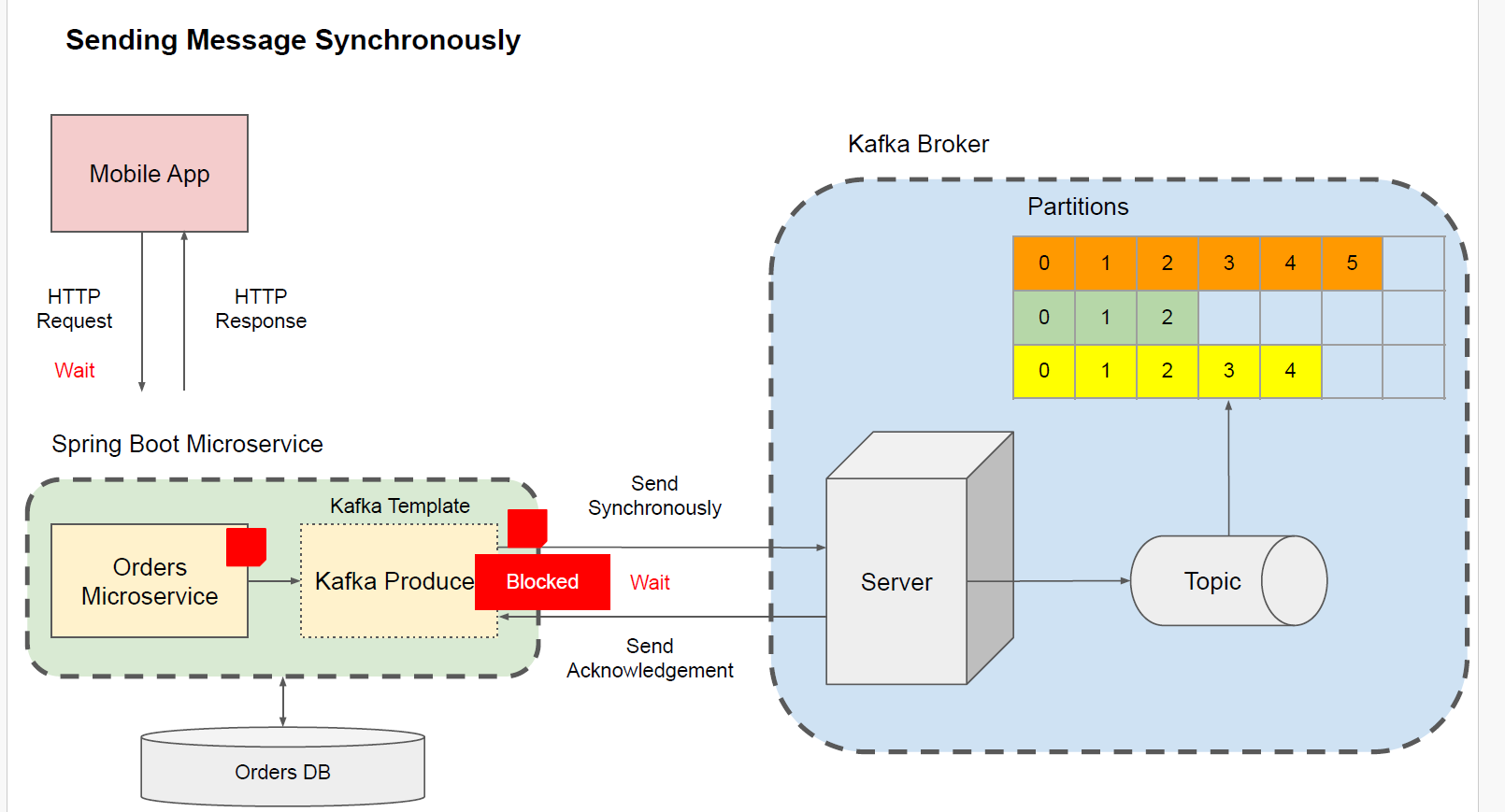
**--property print.value=true //default is true**

**--property key.separator=::**

**--from-beginning**

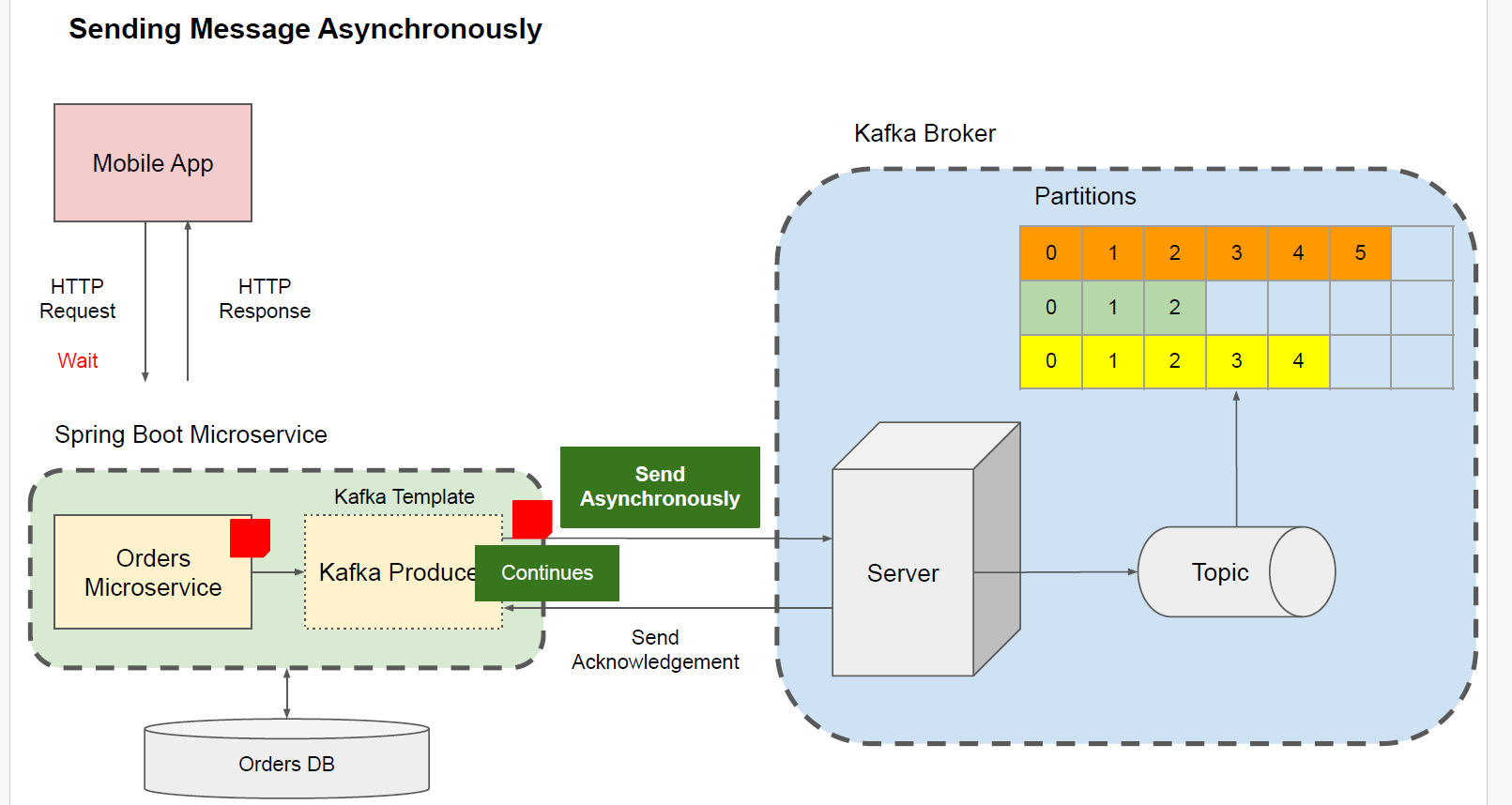
Spring Boot project : Doc **Kafka Producer**



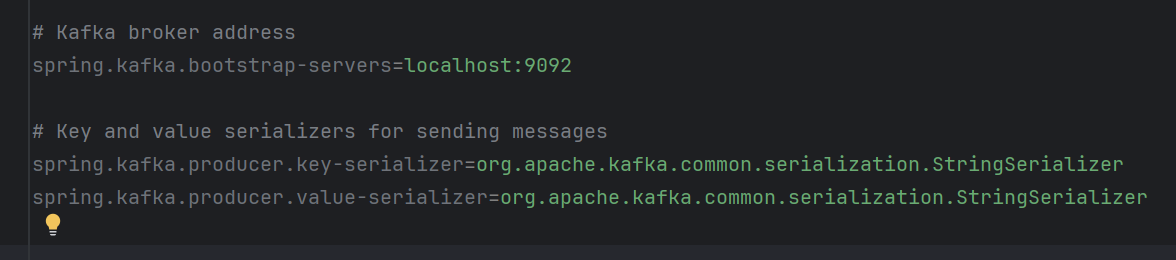
**Synchronous Communication Style**

\

**ASynchronous Communication Style**

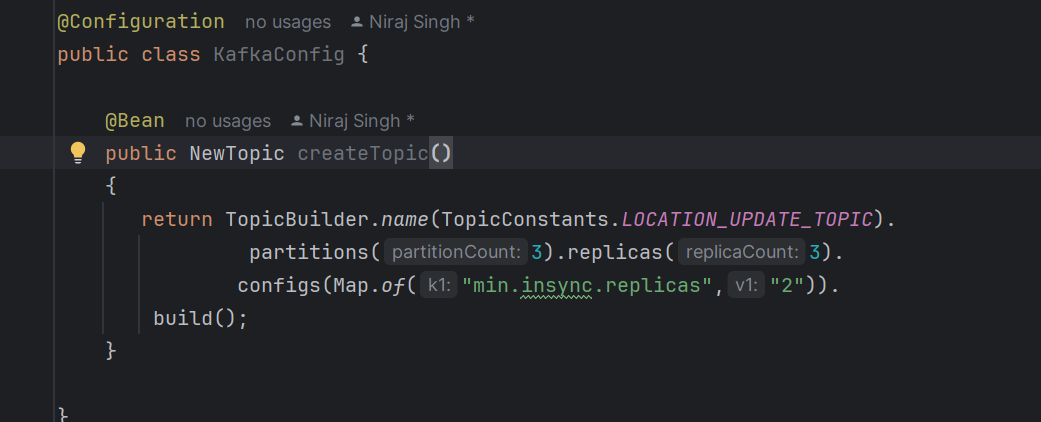


Producer Spring boot application

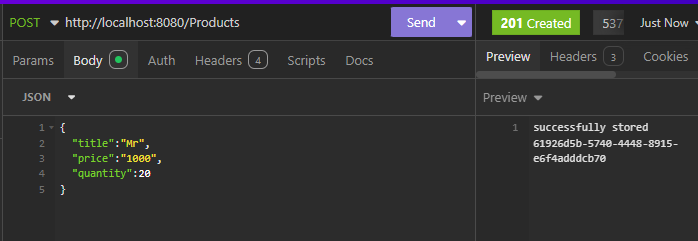
Application properties:  


1. KafkaConfig.java -> to create topic when spring boot application get up  
   @Configuratoin 🡪 class level

@Bean 🡪 Method level



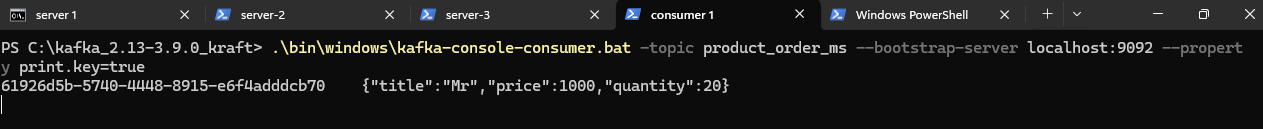
Producer:



Producer produced msg on partion 0



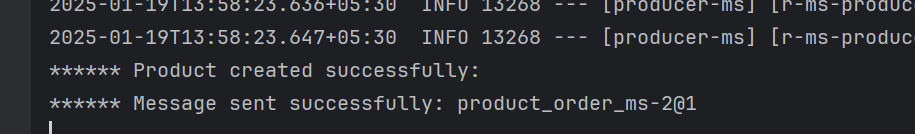
Consumer received msg which was produced by spring (here we have 3 brokers running)



**Asynchronous: As per code**



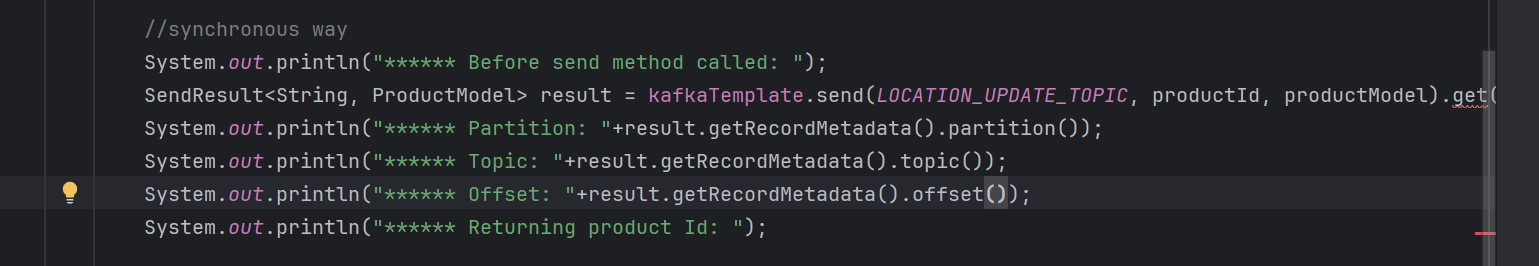
After executing order of print statement proves that it is working asynchronously.(order of printing varied from actually code written as kafka response delayed)

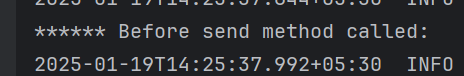


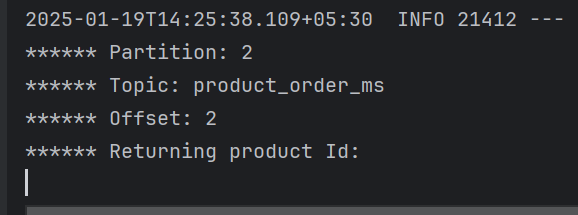
**Synchronous->**

**1)second approach**

**.get()**

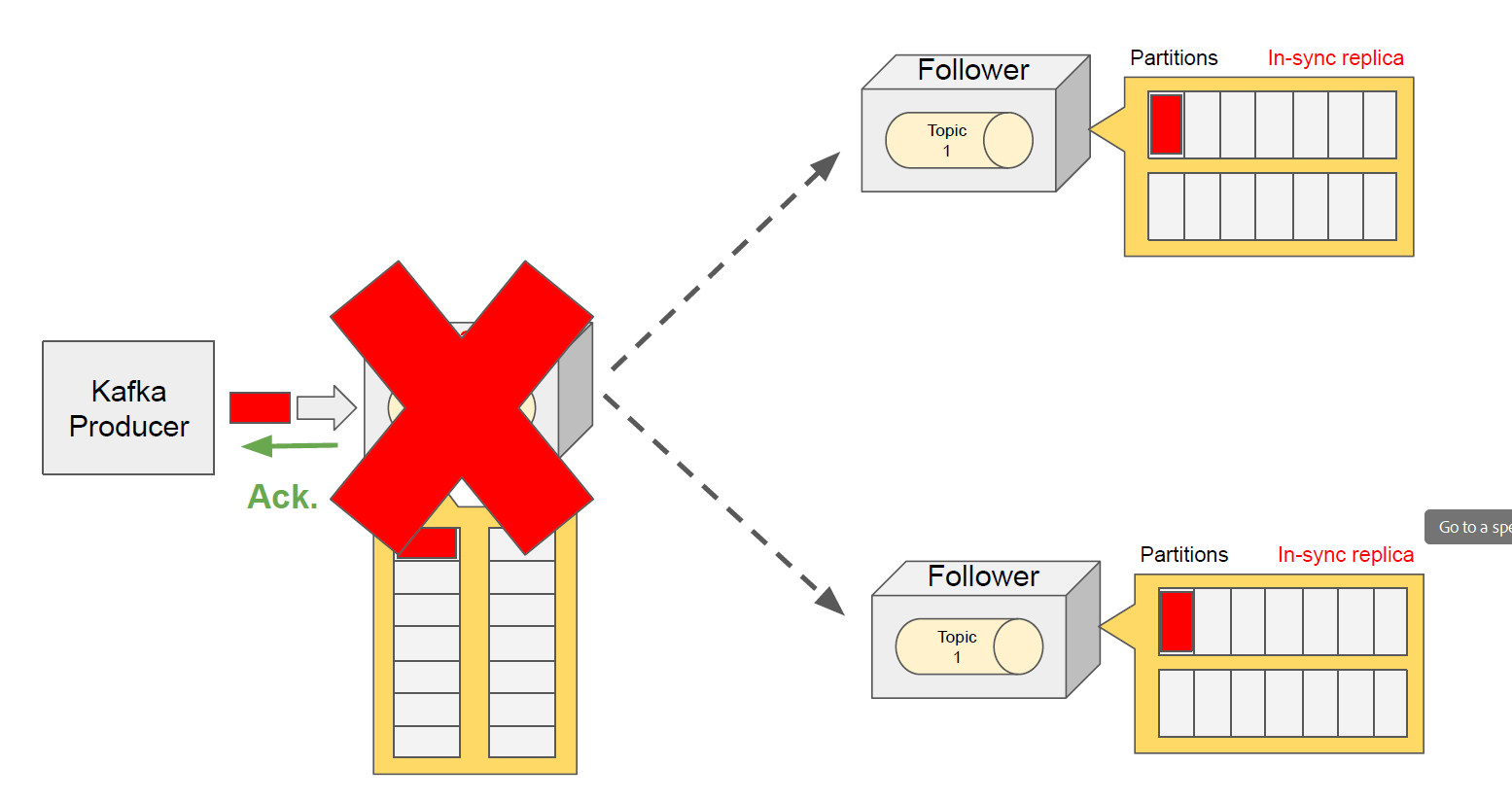
****

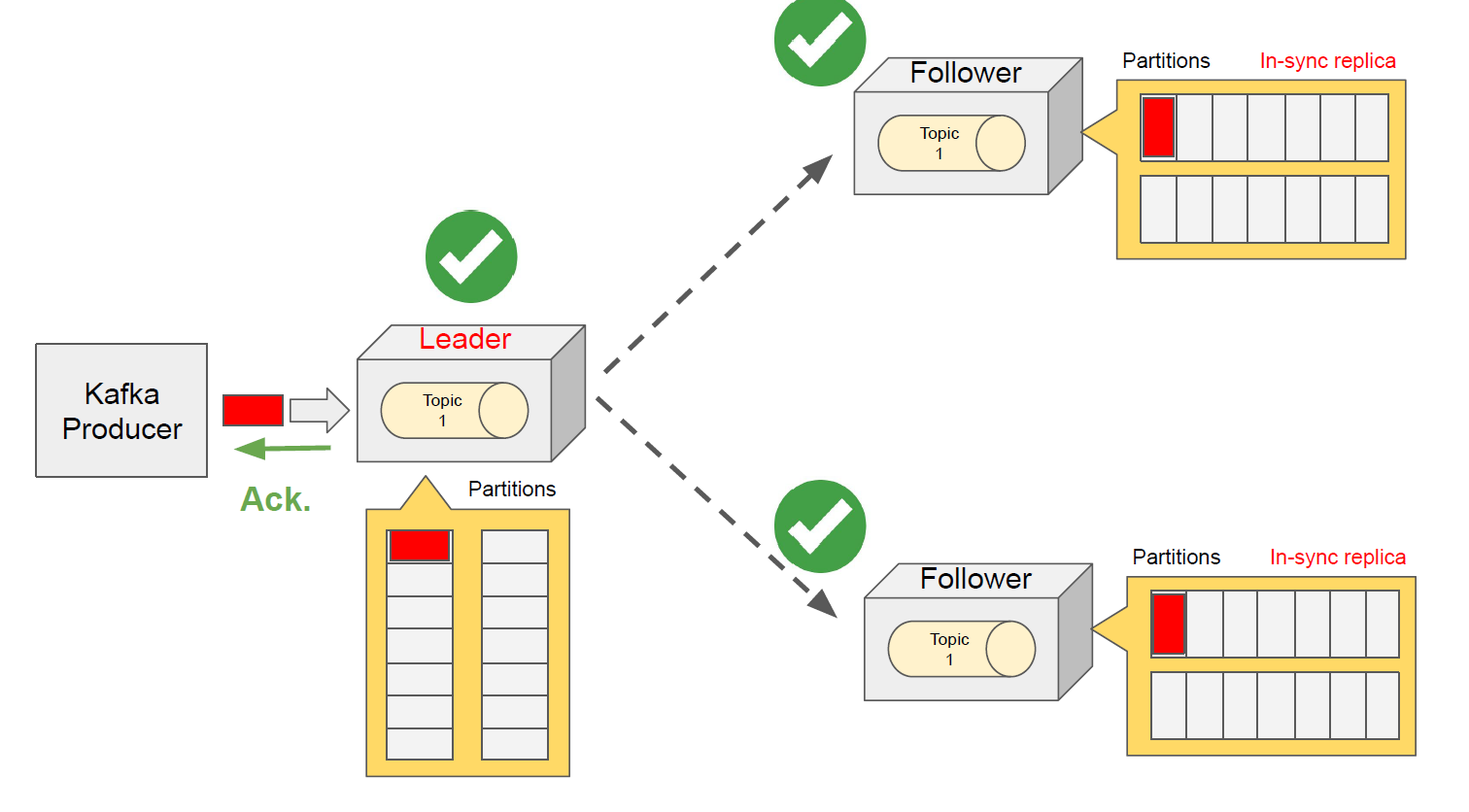
****

****

**Kafka Producer Acknowledgement and retries**

**Section 7:**

**Bydefault kafka produce 1 broker msg received ack, however lets say it got down after ack without being replicating it to other broker, in this case msg will be lost.**

We can have acknowledgement after lets say min 2in sync replica, process will increase time.  


● **spring.kafka.producer.acks=all // more broker will make producer slower**

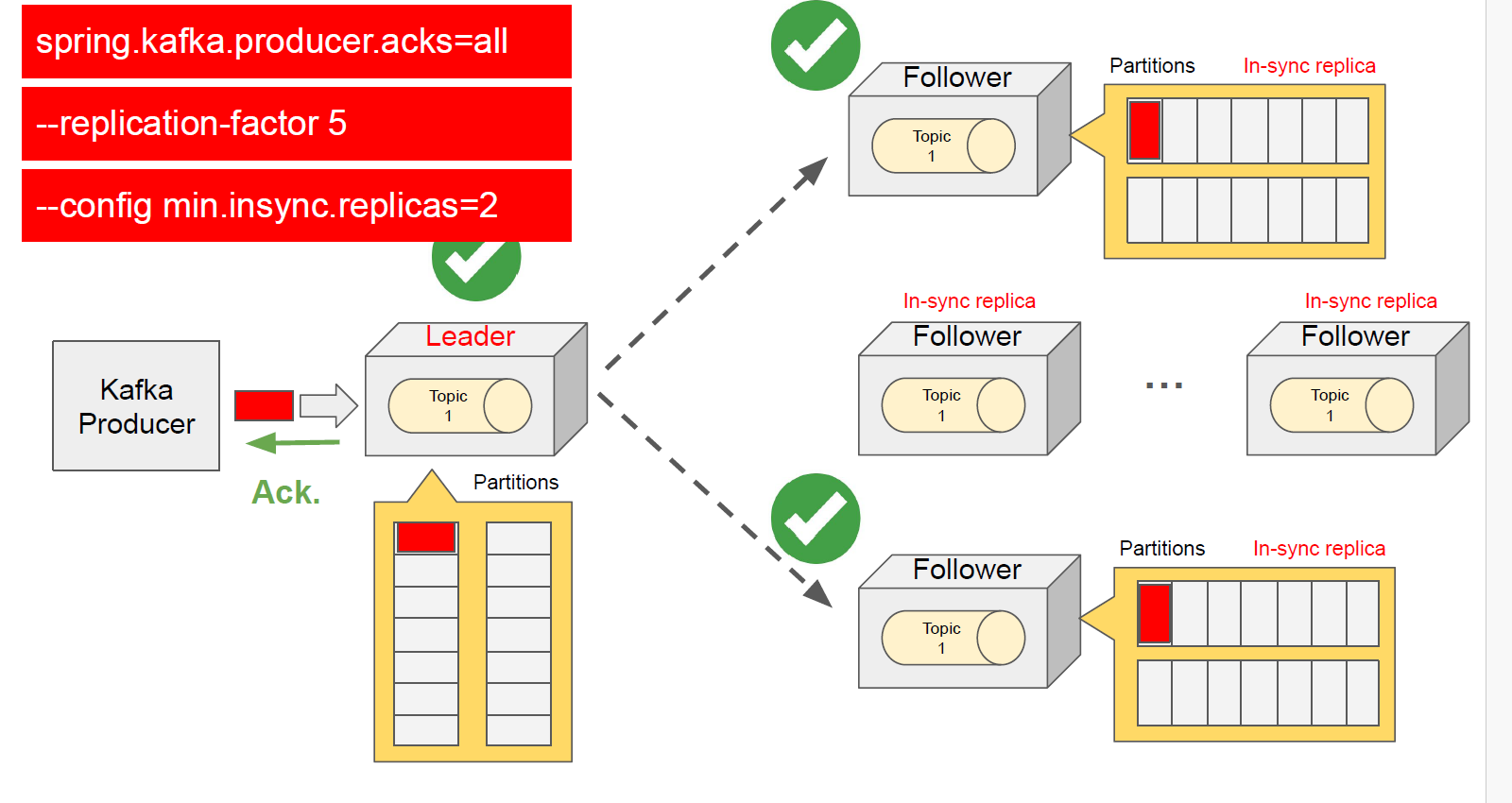
Waits for an acknowledgement from all brokers.

● **spring.kafka.producer.acks=1**

Waits for an acknowledgement from a leader broker only.

● **spring.kafka.producer.acks=0**

Does not wait for an acknowledgement.

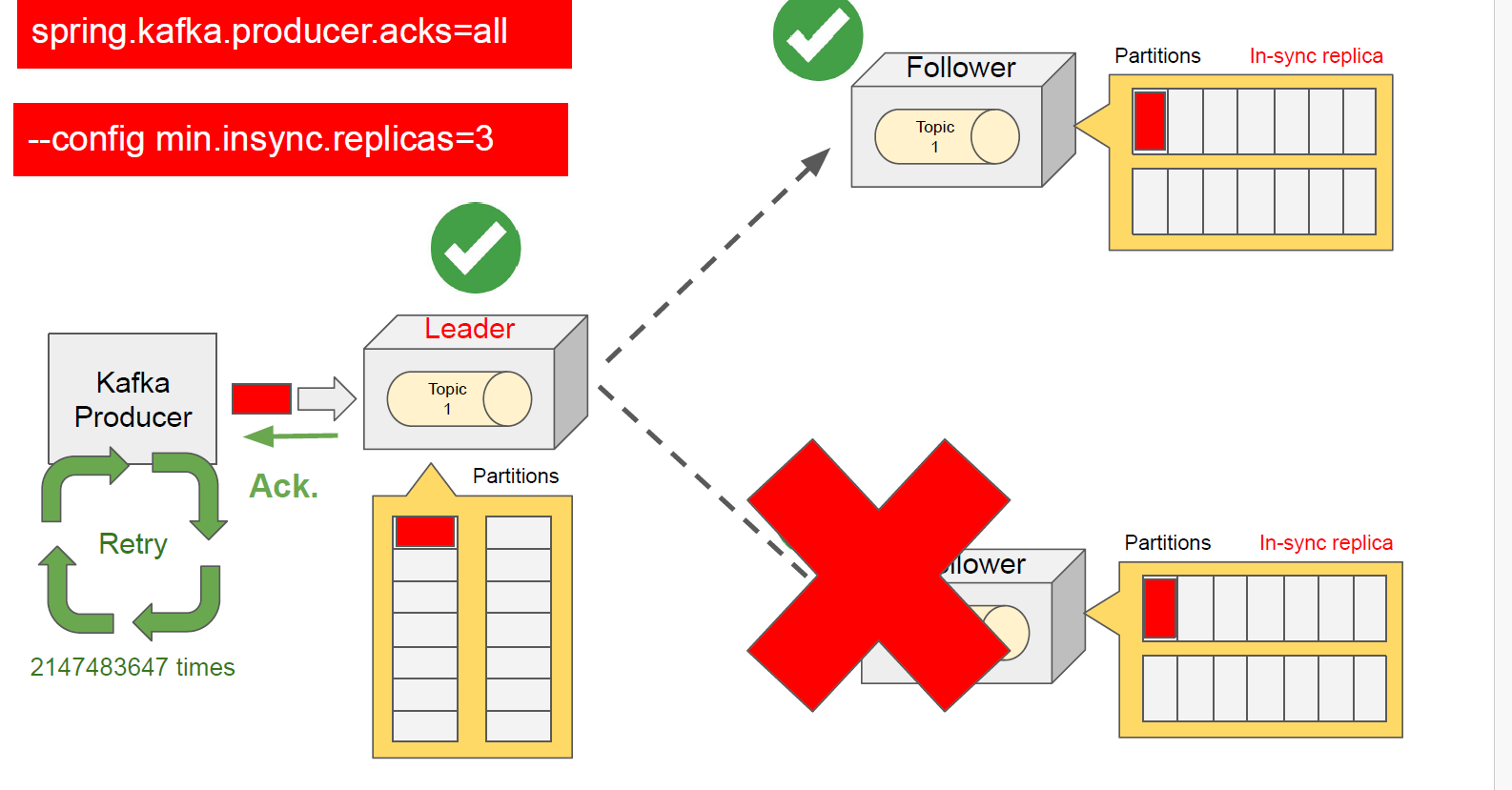


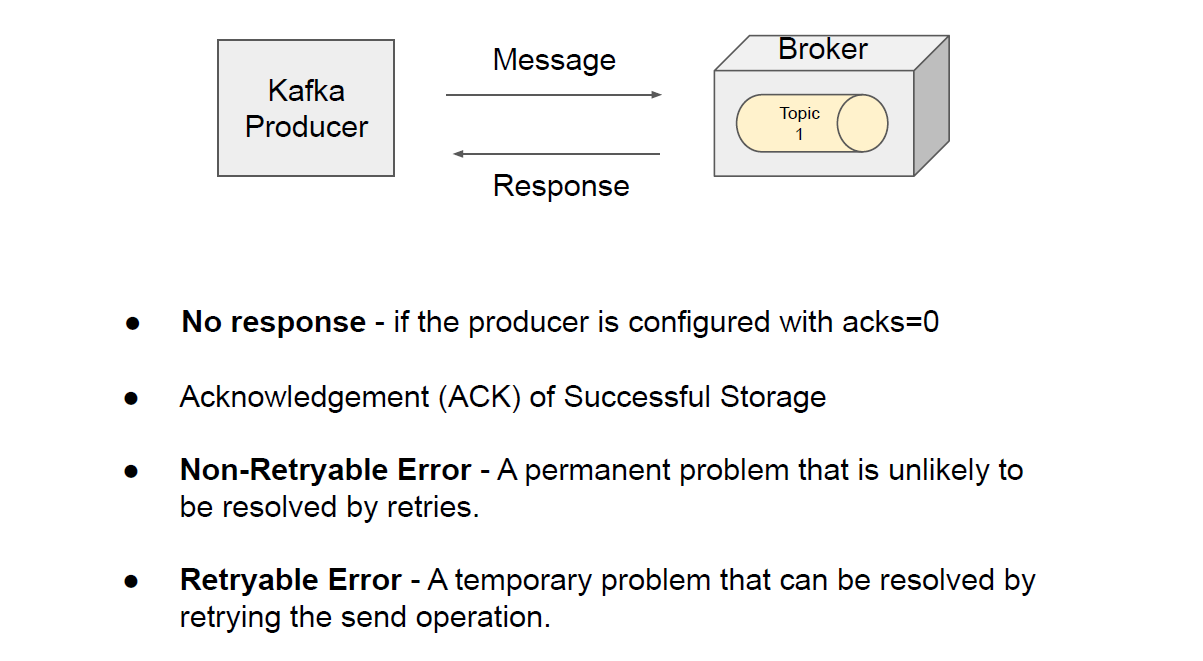
If set min.insyn replica it will or override the acks all to insync replica value. Producer will only wait for insyc replica to ack.  
below combination



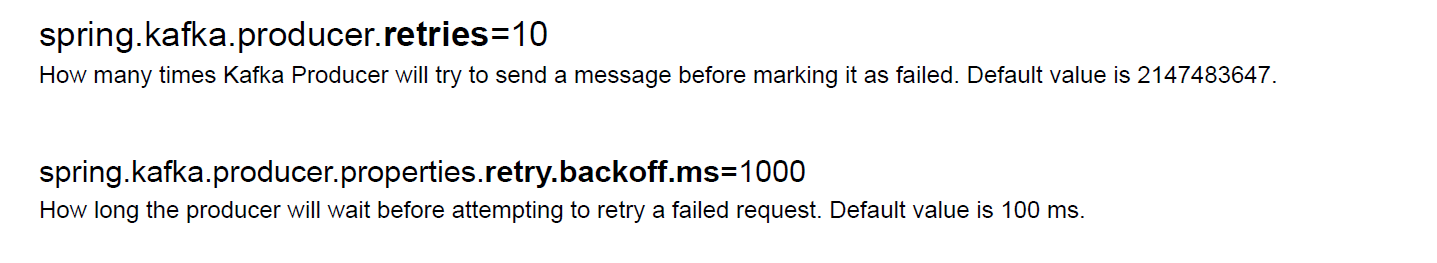
Kafka retries:

Problem:

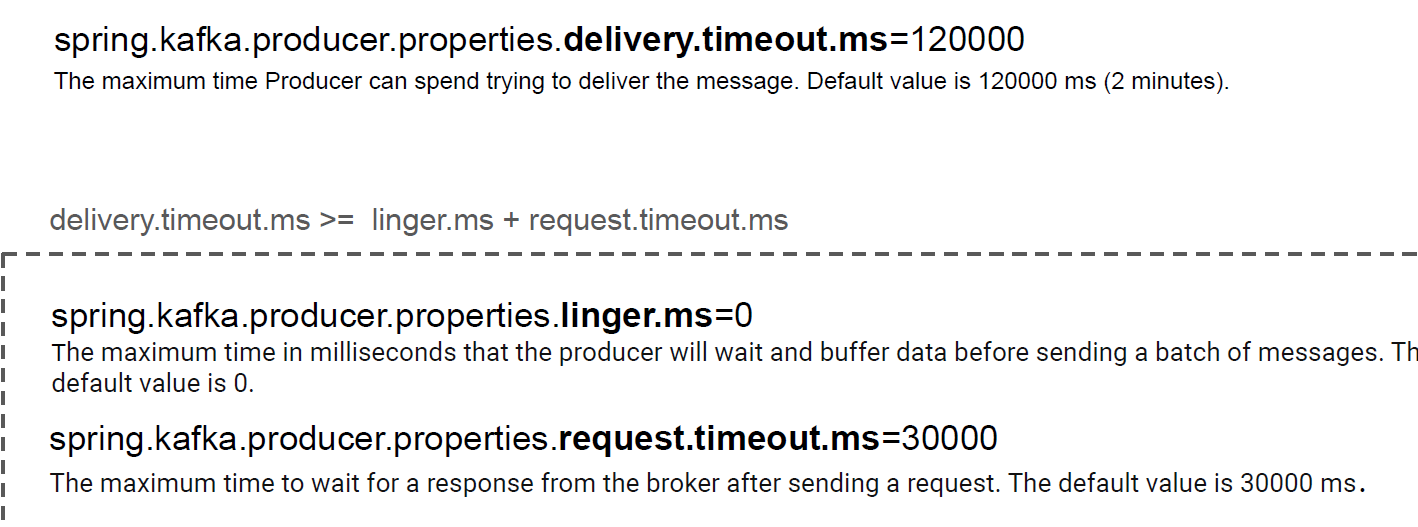


Insync replica is 3 however our 3rd replica gone down then, kafka will retry again and again.  
Below are the scenarios between broker and producer.  


**Retryable and Non-retryable will be decided by kafka.**

**Properties to set for retries.**

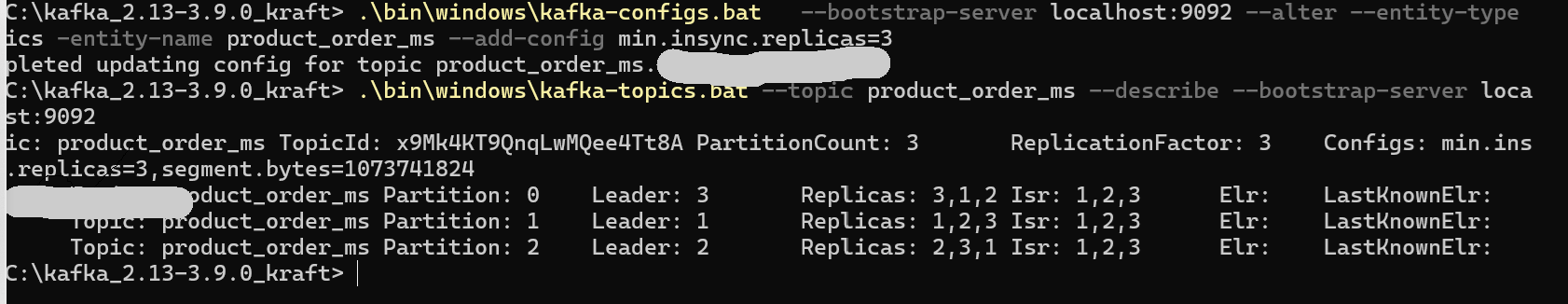
……………………………………………………………………………………………………………………………………………………



**Delivery.timeout.ms -> it is the that whole send process is concerned**

**Request.timeout.ms -> this time is considered for singe request.**

**Updating insync replicas count :**  
.\bin\windows\kafka-configs.bat --bootstrap-server localhost:9092 --alter --entity-type topics -entity-name product\_order\_ms --add-config min.insync.replicas=3



**Trying how it works in code Now:  
in our code we have ack as all, and topic insync replicas is 2 -> so any point of time producer requires 2 insync replicas data patch ack from broker. When 1 broker is running we will get error and multiple retries will happen from producer till timeout.**

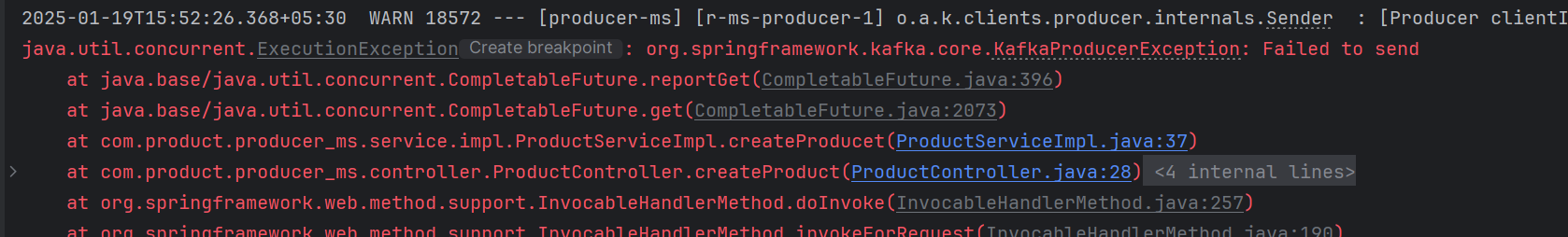
**Peoperties:**

**Ack=all**

**Insync replica=2**

**Broker up count 1:**

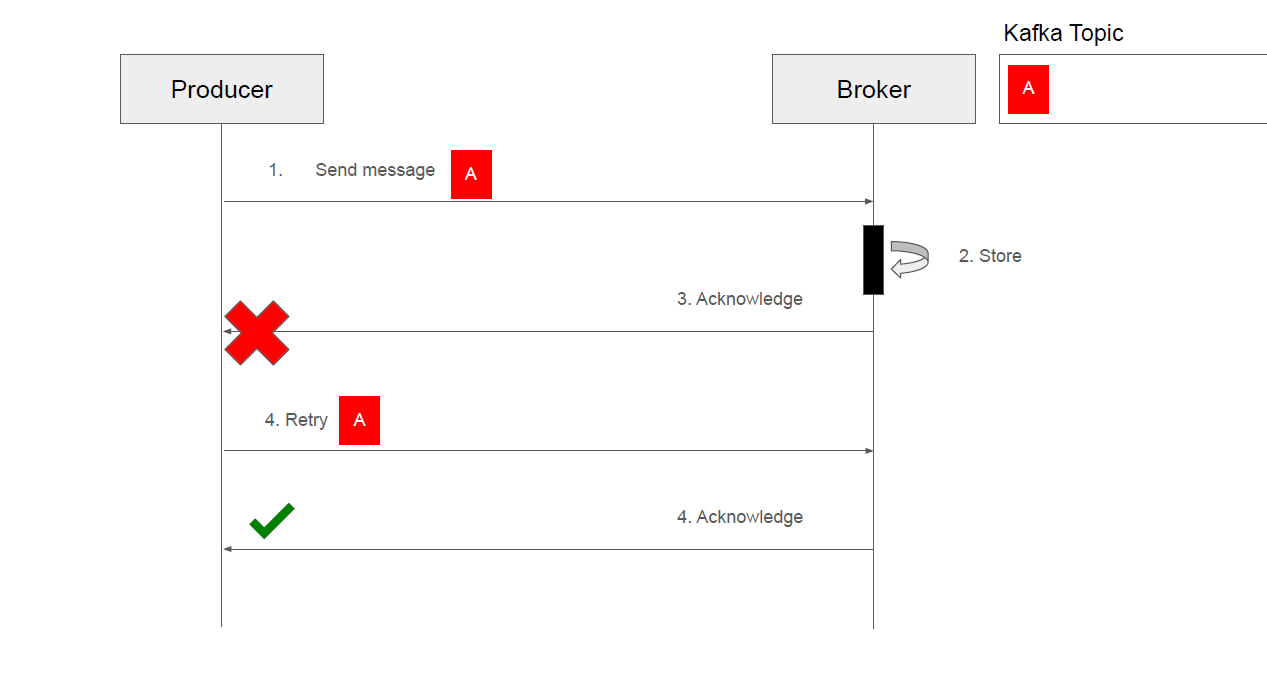
**Retries =10**

**Producer trying to publish message:  
**

****

**Section 8: Idempotent Producer in Kafka**

**Problem in graphical representation:**

****

**In Apache Kafka, idempotent means that a producer can send the same record multiple times, but it will be stored exactly once in the Kafka log. This ensures that even if the producer retries sending a record due to failures (like network issues or broker downtime), the record will not be duplicated in the Kafka topic.**

**The idempotent producer is a feature introduced in Kafka to ensure exactly-once semantics (EOS) when producing records**

**Benefits of Idempotence**

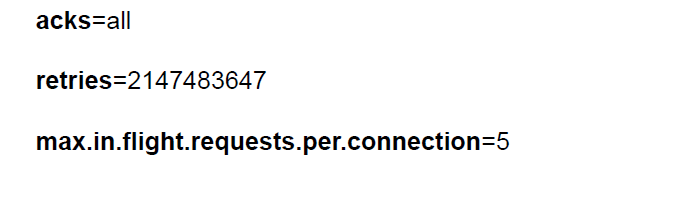
1. **Exactly-once Semantics**: Idempotence is a key building block for achieving exactly-once semantics in Kafka.
2. **Resilience**: Producers can safely retry sending messages without worrying about duplicates in case of transient failures.
3. **Data Integrity**: It prevents data duplication in Kafka topics, which is crucial for applications that need to maintain data accuracy.

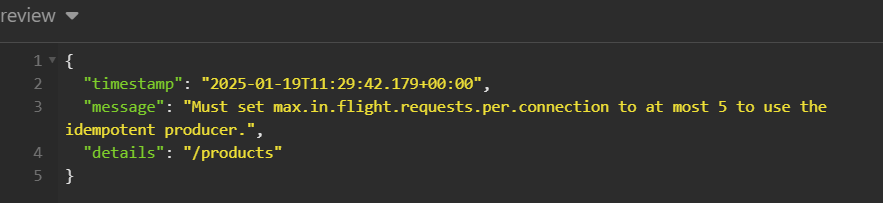
**spring.kafka.producer.properties.enable.idempotence=true**

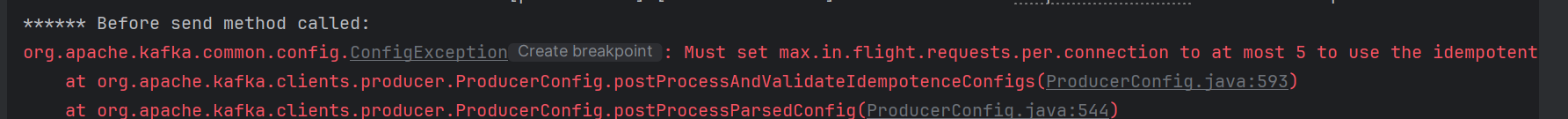


**Note :**

**By default idepotent is true. However it is recommended to set it true explicitly as it can indirectly disabled by below confliting values .**

****

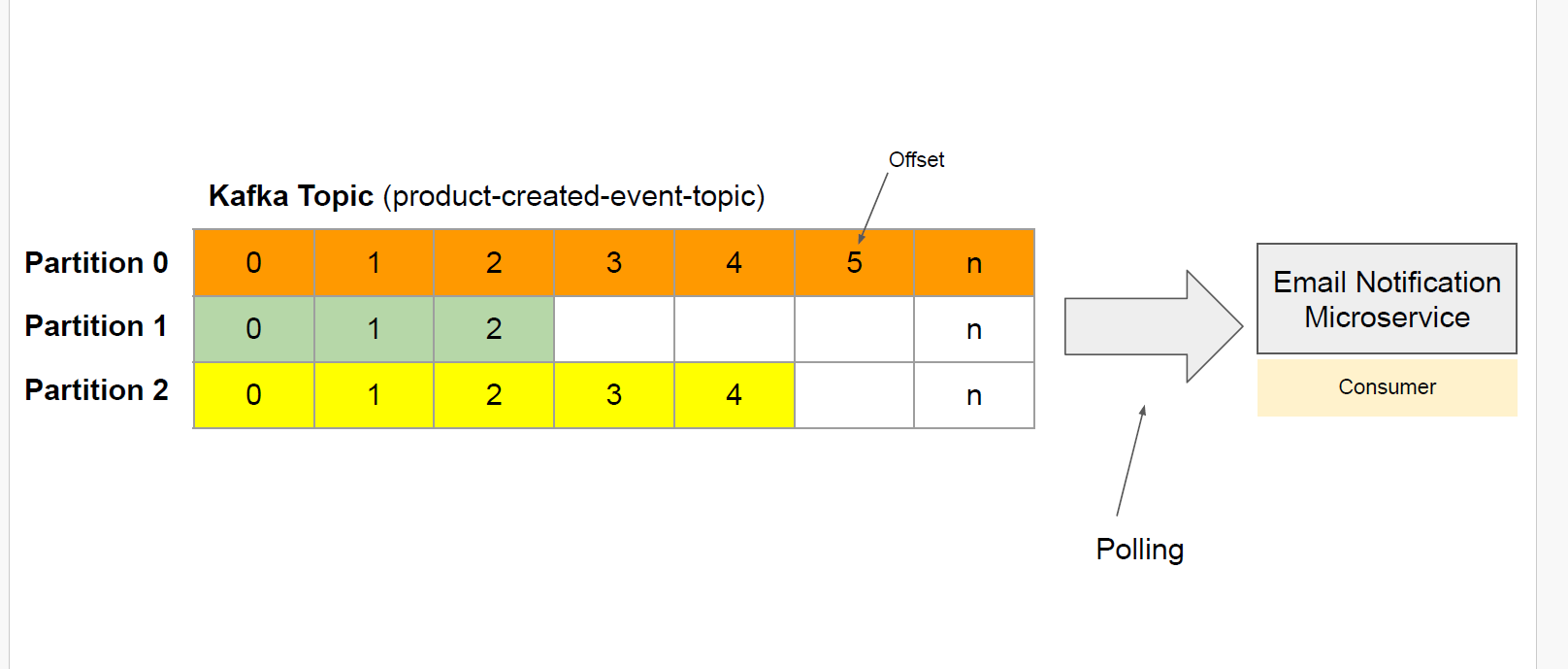
**In order to use idempotence, as per documetation , request inflight value should not be greater than 5. Id you make > 5 it will below error   
  
**

****

**Section 9: Kafka Consumer - Spring boot Ms**

****

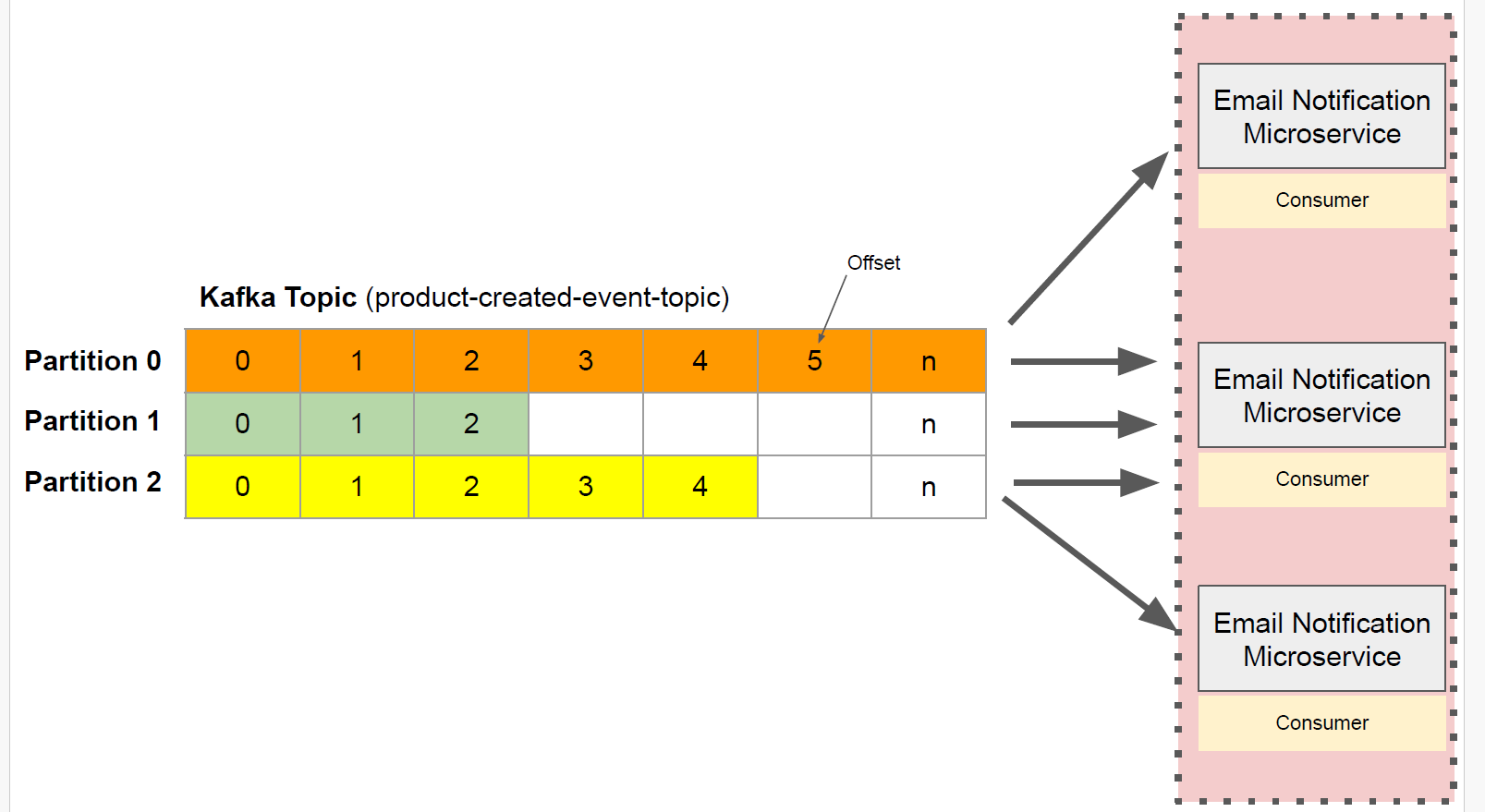
**……**

**  
……………………………………………………………………………………………………………………………………………………………..**

**Points to Note:**

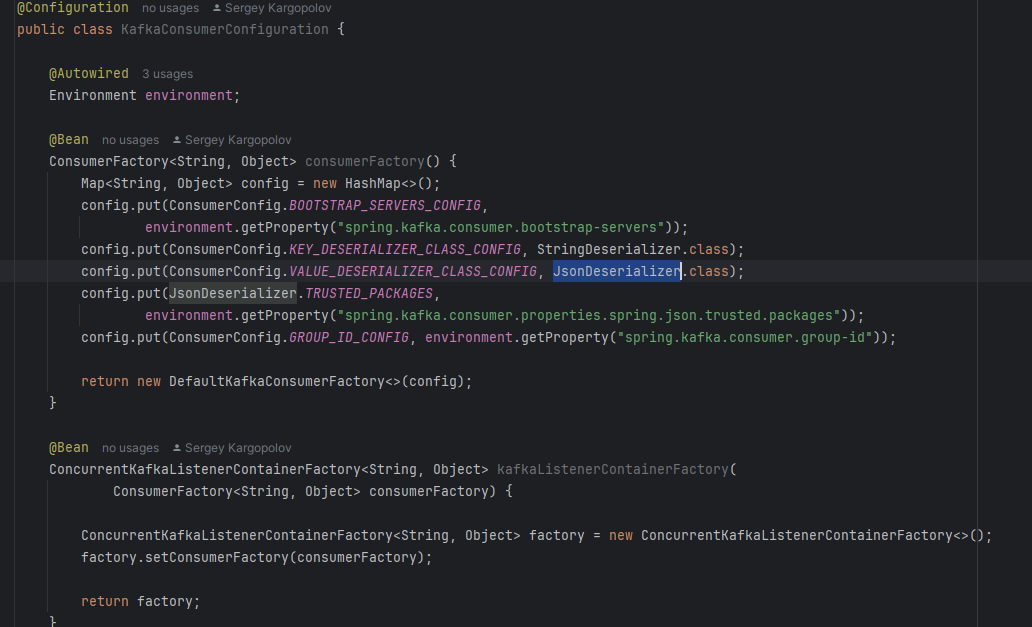
**1)order of consuming msg in partition is guarantee**

**2) order of consuming msg in across partition is not guarantee**

**That is why we can have consumer group: each will pick partion to process  
**

**How Consumer is set in Springboot?**

**Configuration class**

****

**Listener class:**

**@KafkaListener -> class level or method level**

**@KafkaHandler -> method level**

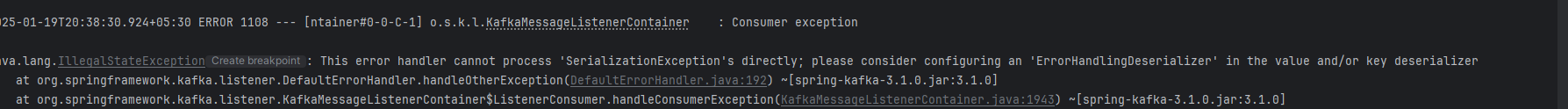
****

**Section 10 kafka consumer -handler Desrialize Errors**

Problem when deSrialization not handled properly:



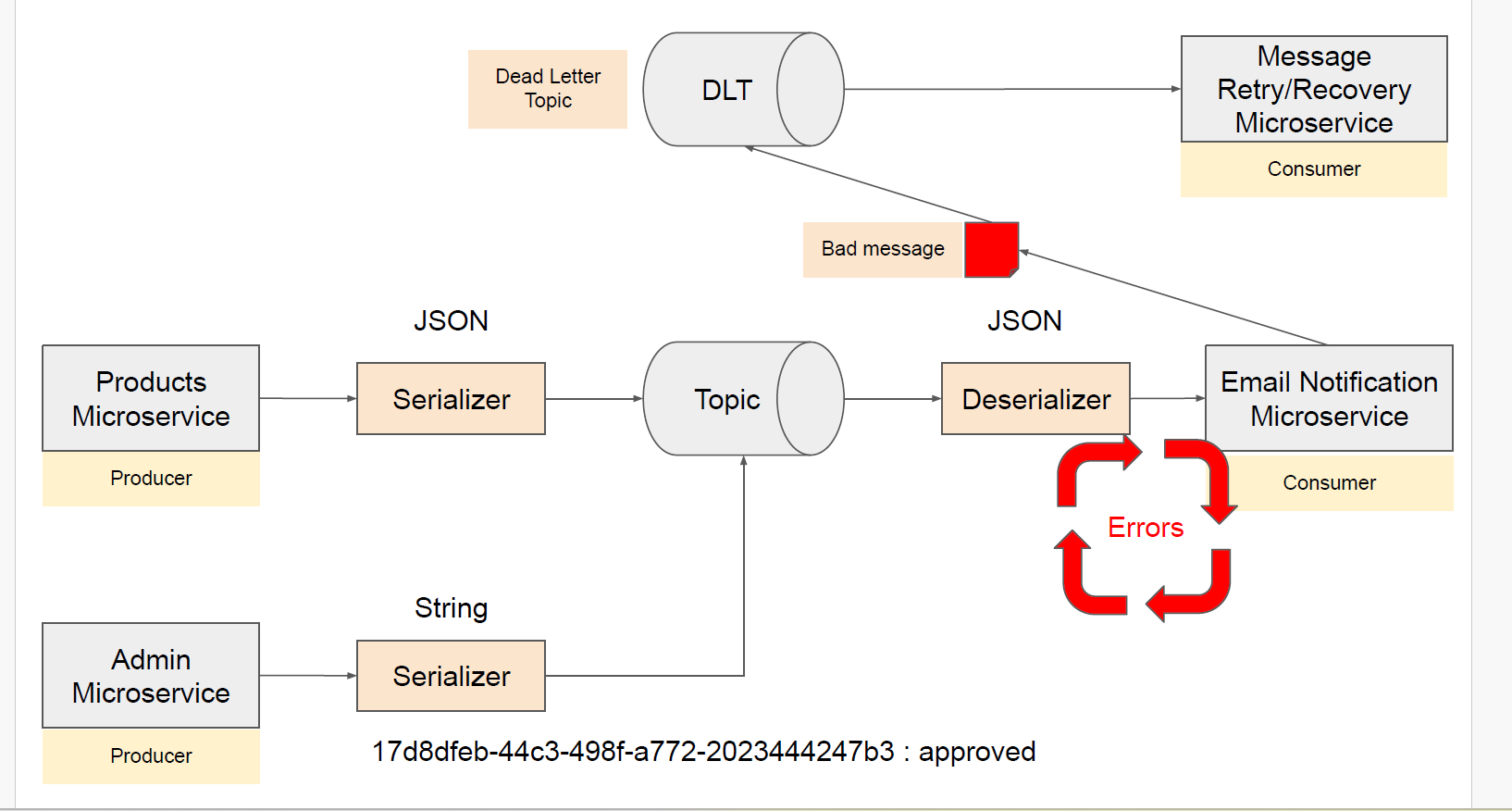
1. Scenarios where is will be there : endless loop where consumer will try to consumer msg  
   



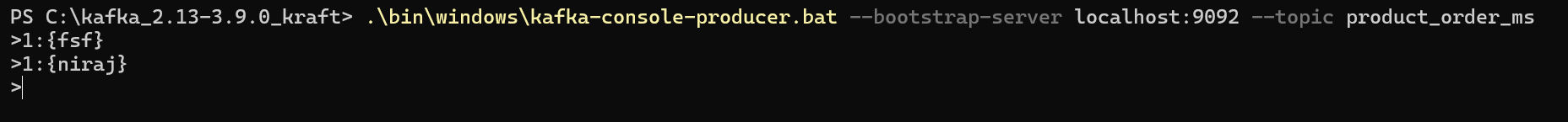
1. Solution:



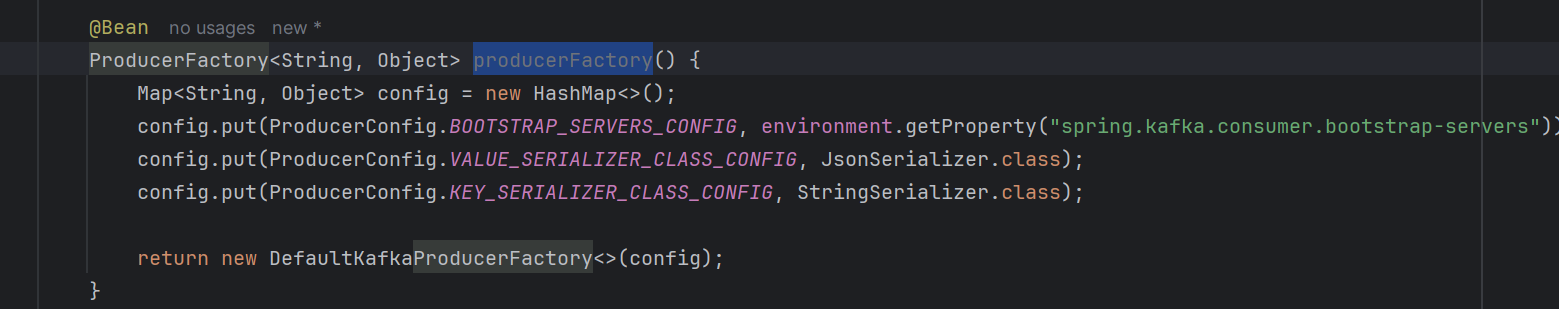
Use ErrorHandlingDeserializer class

**Section 11 Kafka Consumer – Dead Leter Topics(DLT)**

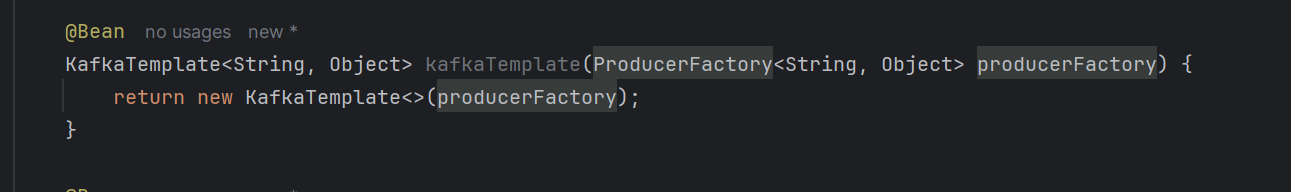
In Apache Kafka, **DLT** stands for **Dead Letter Topic**. A **Dead Letter Topic (DLT)** is used to handle messages that cannot be processed successfully by consumers, even after multiple retries.

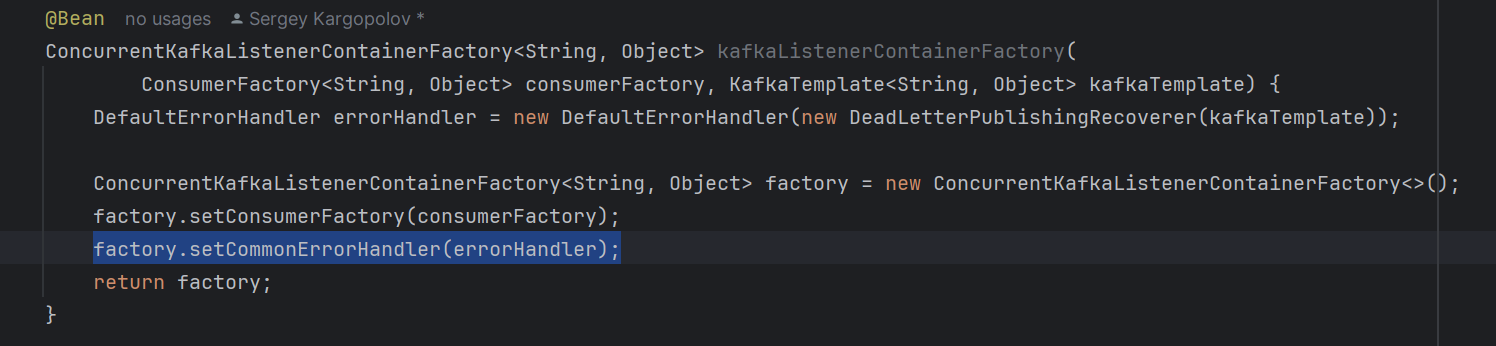


Steps to handle:

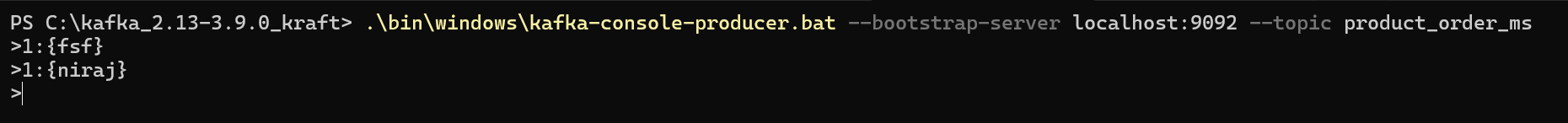
1)need to create producerFactory Object as consumer need to produce error msg to dlt topic.  


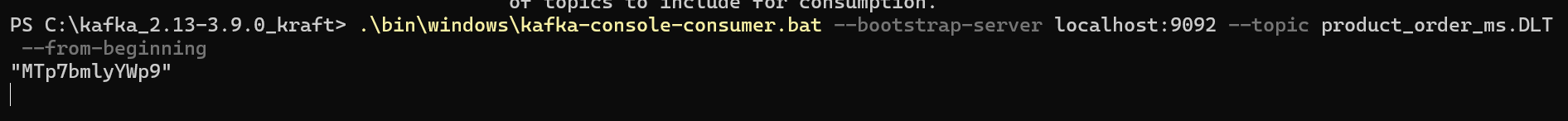
2)Need to create kafkaTemplate which take producer from producer factory

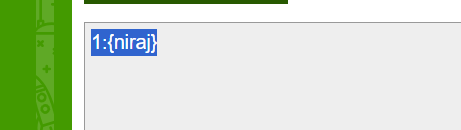


3)above kafka template need to set in kafkaConsumerContianer factory for producing msg to dlt.  


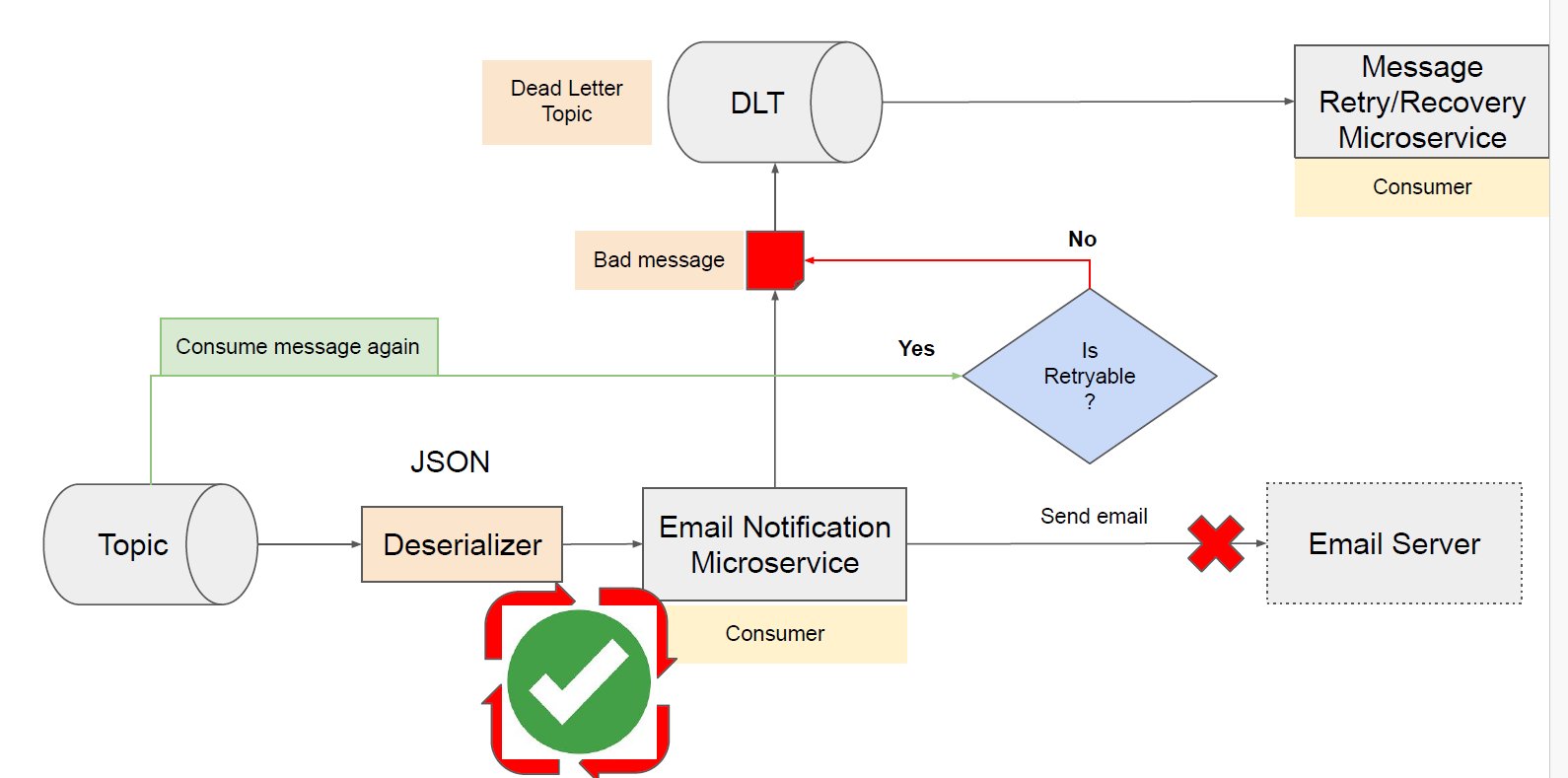
Example:

1)Published wrong format to kafka topic.  


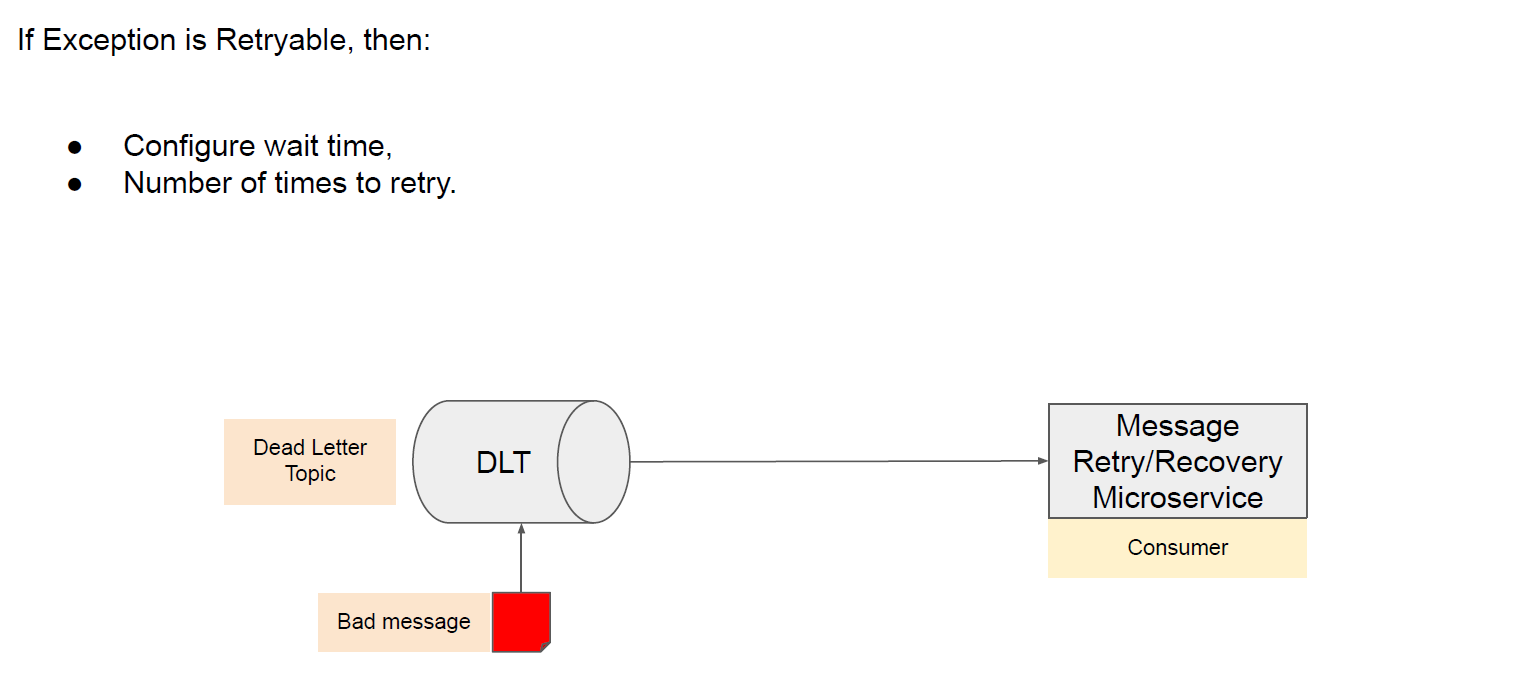
2) we had this msg on our dlt topic as msg not proper format as per consumer requirement.  


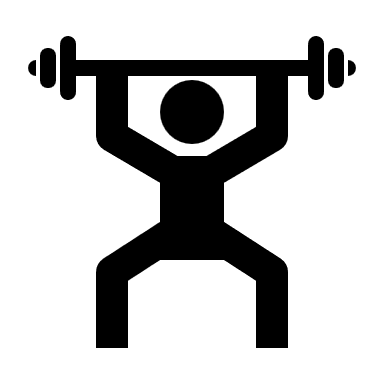
It was in Base64 format.  
After conversion  


Section 12: **Kafka Consumer – Exception and retries**

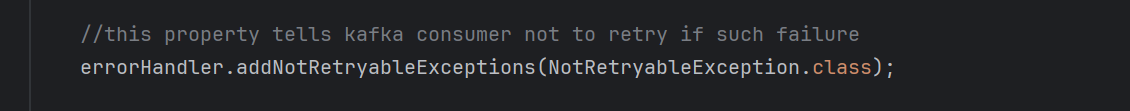
****

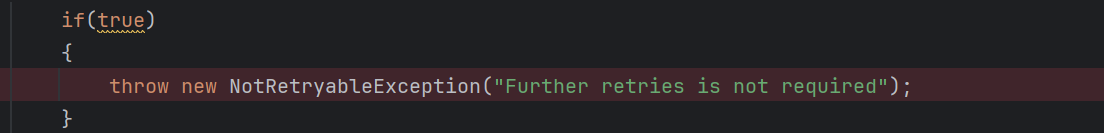






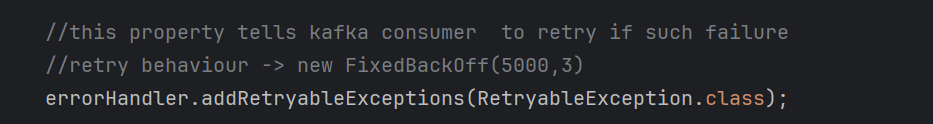
**A)Spring boot NotRetryable setup:**

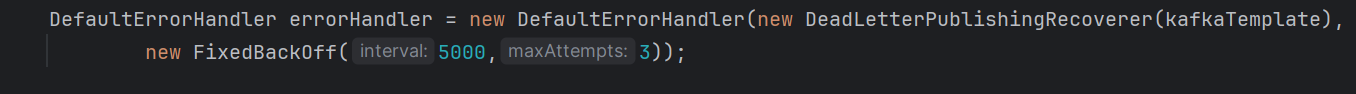
1. Set configuration for errorHandler
2. Throw NotRetryable Exception from consumer method.



**B)Spring boot Retryable setup:**

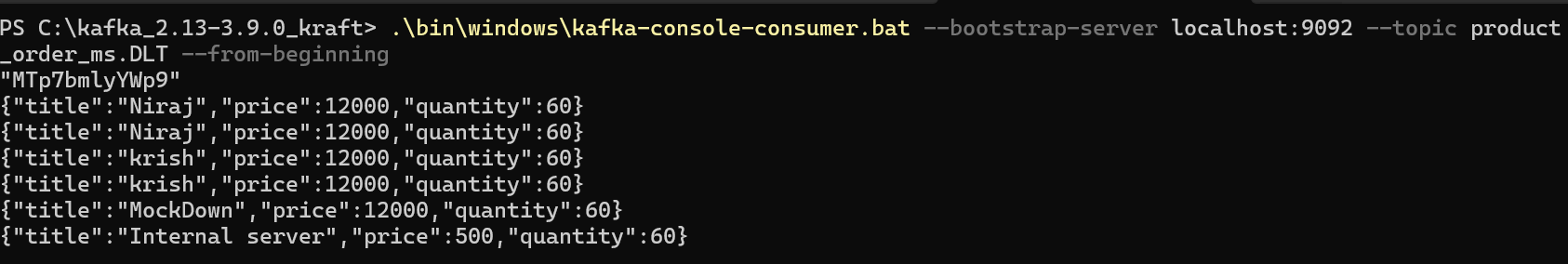
1)Set configuration for errorHandler

****

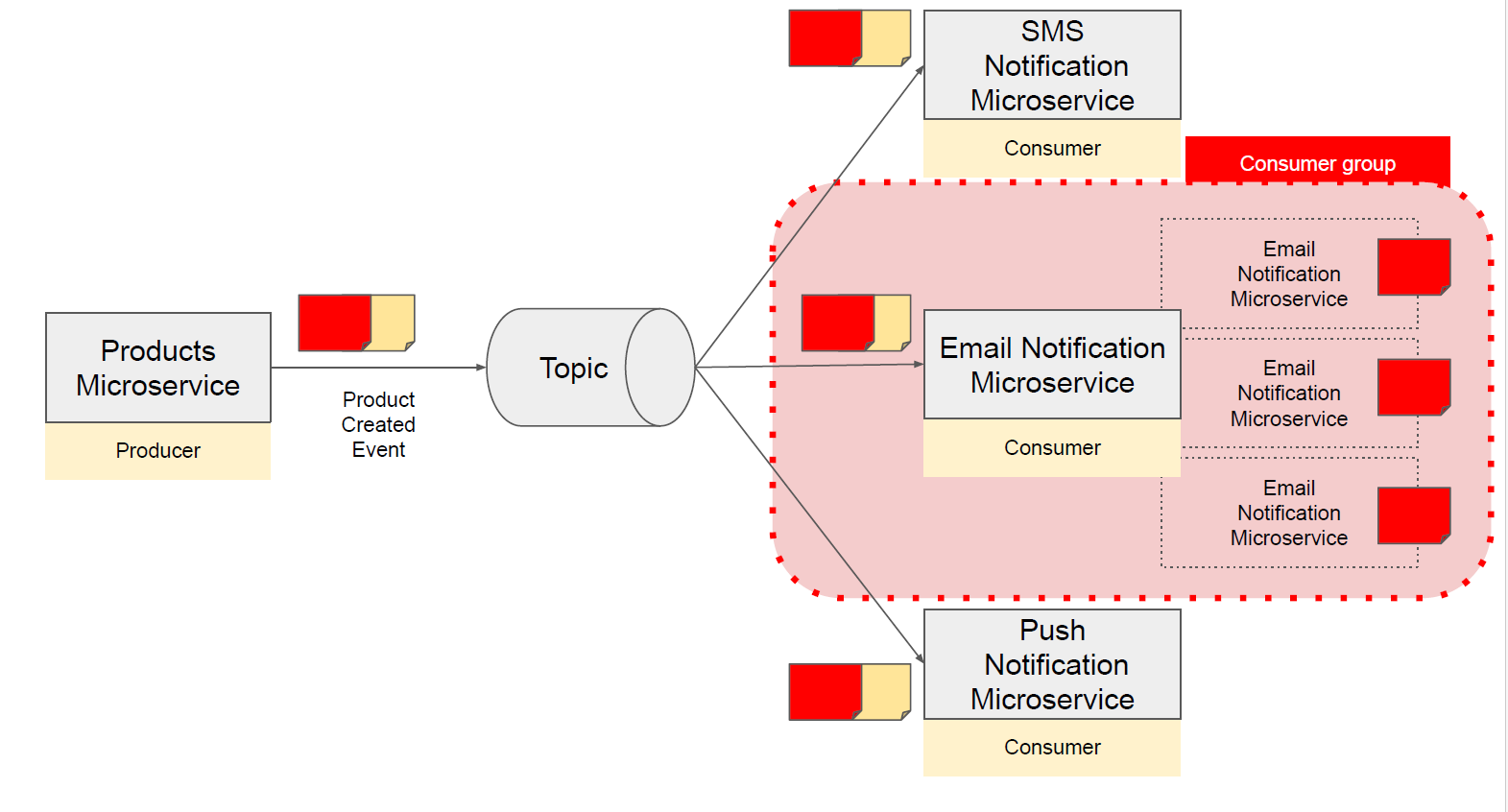


Above lines set cofiguration for Retrying when such custom class thrown from listener method.  
below screen shot show how scenarios handled for retry and notRetry.  


In case of failure at the end -> msg posted to dead layer topic.



**Section 13**  **Kafka consumer – Multiple Consumers in a Consumer group**

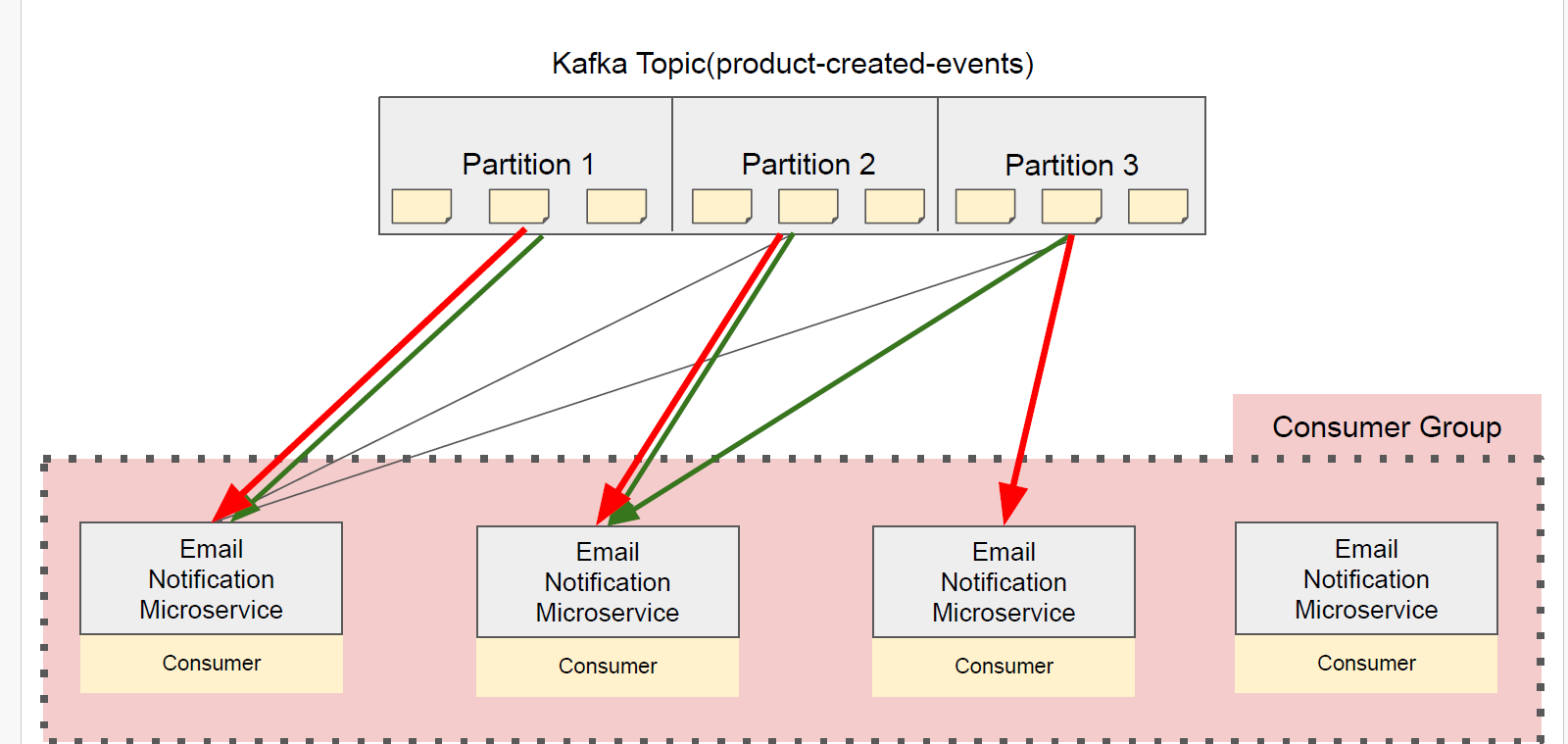
****In consumer group-> each msg in topic is consumed by one member of consumer group.  
Above scenario is valid for grouping, it makes processing faster.

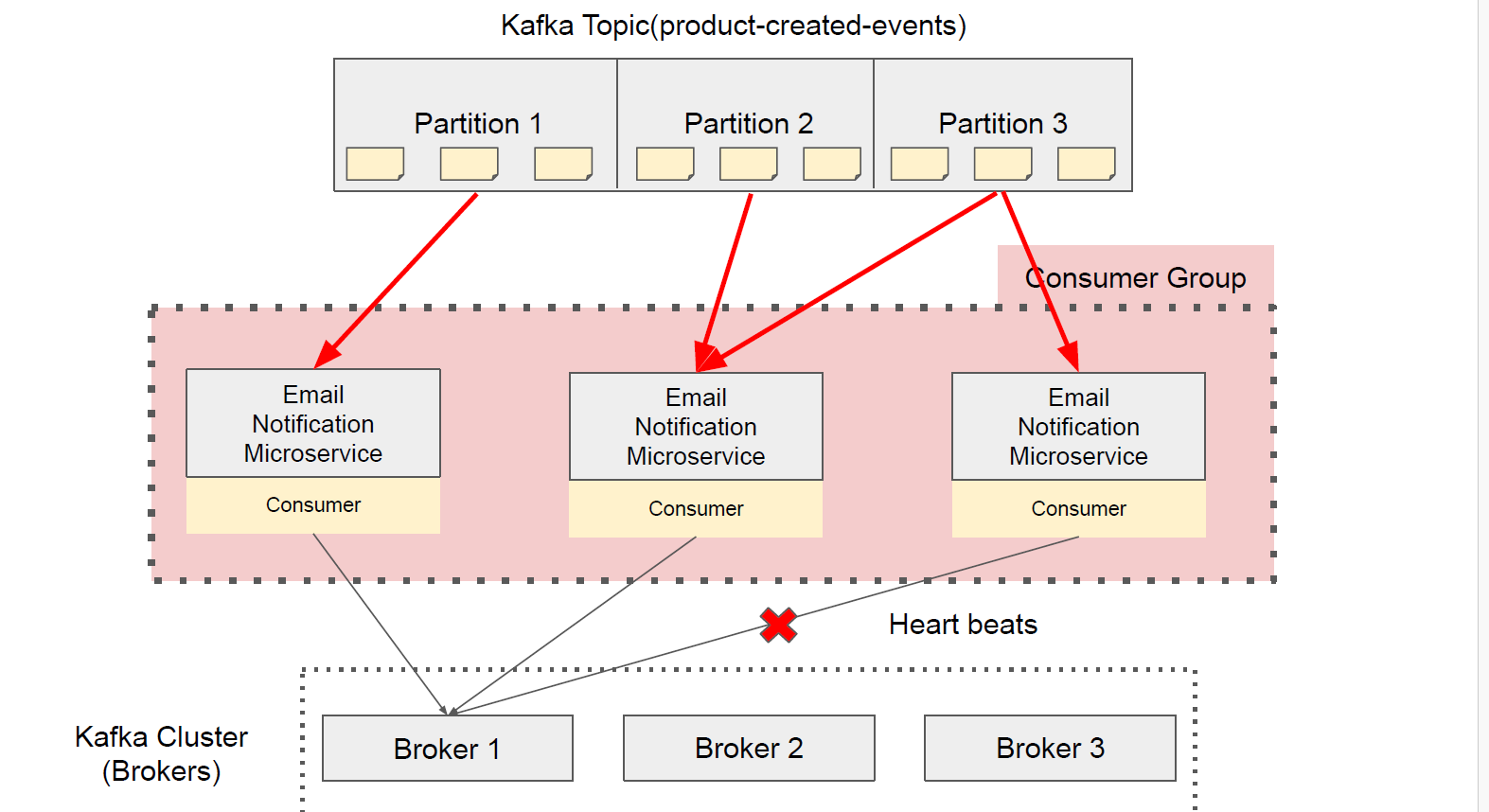
**Rebalancing and Partition Assignment**

This process ensures that each partition is assigned to exactly one consumer in the group, and that consumers share the work of processing the partitions as evenly as possible.

**Key Points about Consumer Rebalancing:**

1. **Triggers for Rebalancing:**
   * **Consumer Joins:** When a new consumer joins the group.
   * **Consumer Leaves:** When an existing consumer leaves the group (voluntarily or due to failure).
   * **Partition Count Changes:** When the number of partitions in a topic is increased.
2. **Rebalance Process:**
   * When any of the above events occur, Kafka's **group coordinator** initiates a rebalance.
   * All consumers stop processing messages temporarily and synchronize their state with the coordinator.
   * The coordinator redistributes the partitions among the consumers.
   * Consumers resume processing messages from their newly assigned partitions.
3. **Implications of Rebalancing:**
   * **Temporary Downtime:** During rebalancing, there can be a short period when no consumer processes messages.
   * **Data Reprocessing:** If a consumer was processing a message when the rebalance occurred, the message might be reprocessed by the new consumer assigned to that partition.
   * **Performance Impact:** Frequent rebalancing can affect the throughput and stability of the consumer group

****

****

**Assigning ms to a consumer group**

1. **Using @KafkaListener Annotation:**

@KafkaListener(topics = "your-topic-name", groupId = "your-group-id")

1. **Configuring in application.properties or application.yml:**

spring.kafka.consumer.group-id=your-group-id

1. **Using KafkaListenerContainerFactory:**

factory.getContainerProperties().setGroupId("your-group-id");

1. **Using Multiple Consumer Groups:**

**@KafkaListener(topics = "topic1", groupId = "group1")**

**public void consumeFromGroup1(String message) {**

**System.out.println("Group1: " + message);**

**}**

**@KafkaListener(topics = "topic2", groupId = "group2")**

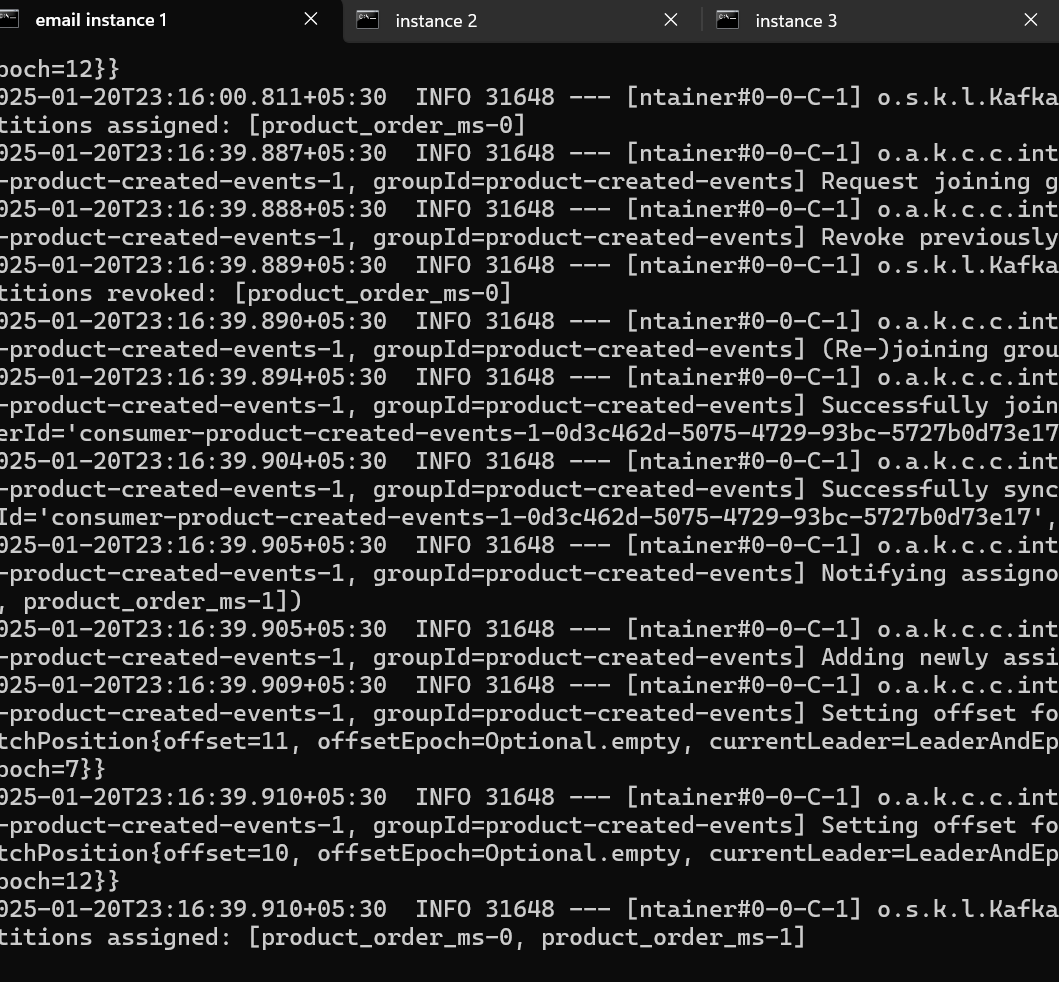
**public void consumeFromGroup2(String message) {**

**System.out.println("Group2: " + message);**

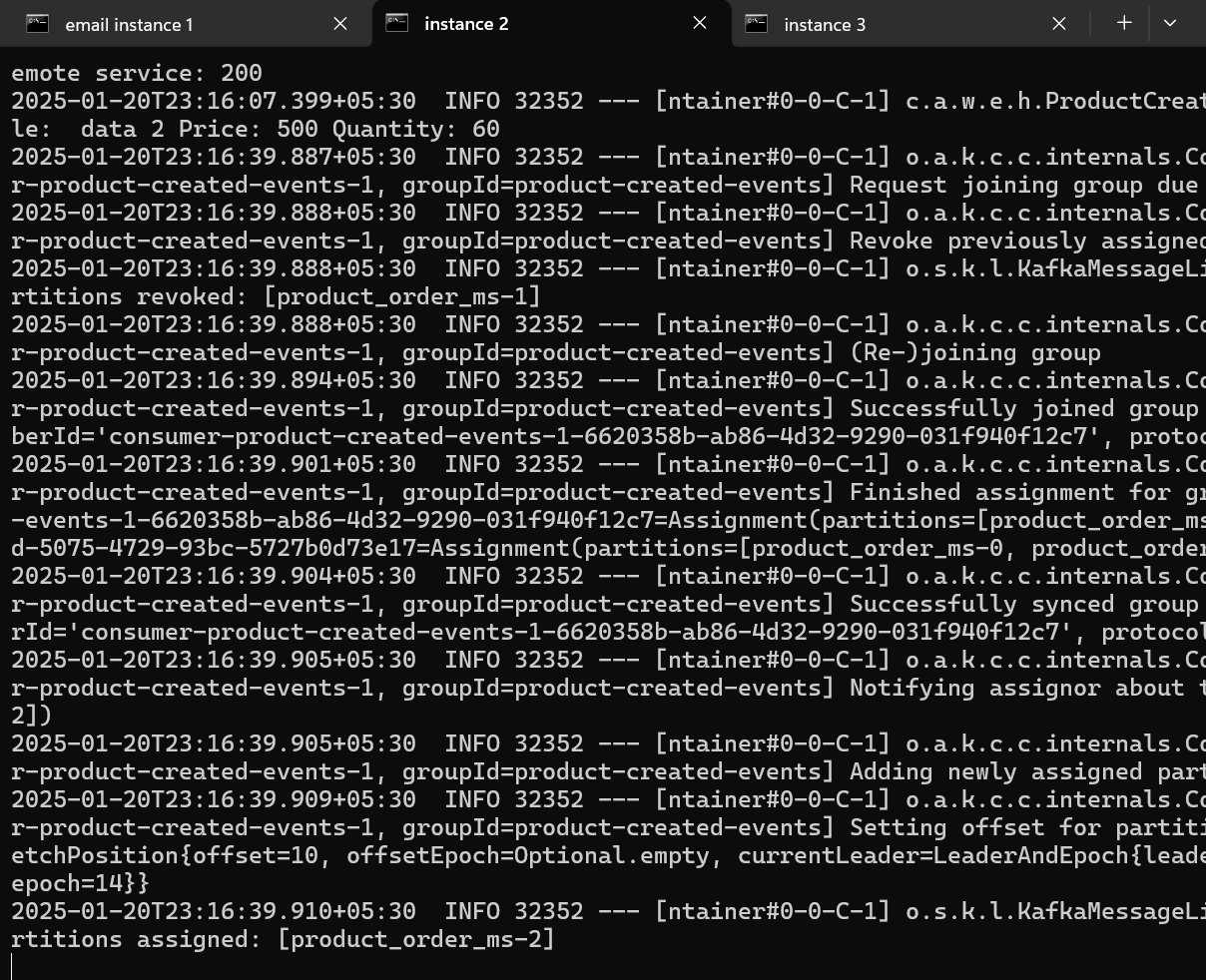
**}**

**All above theory I validated from EmailNotificationService, having 3 instance in same group.**

**3-broker, 3 partition, 3 replica**  
 Instance 1 assigned -> partitition 0 and 1

****

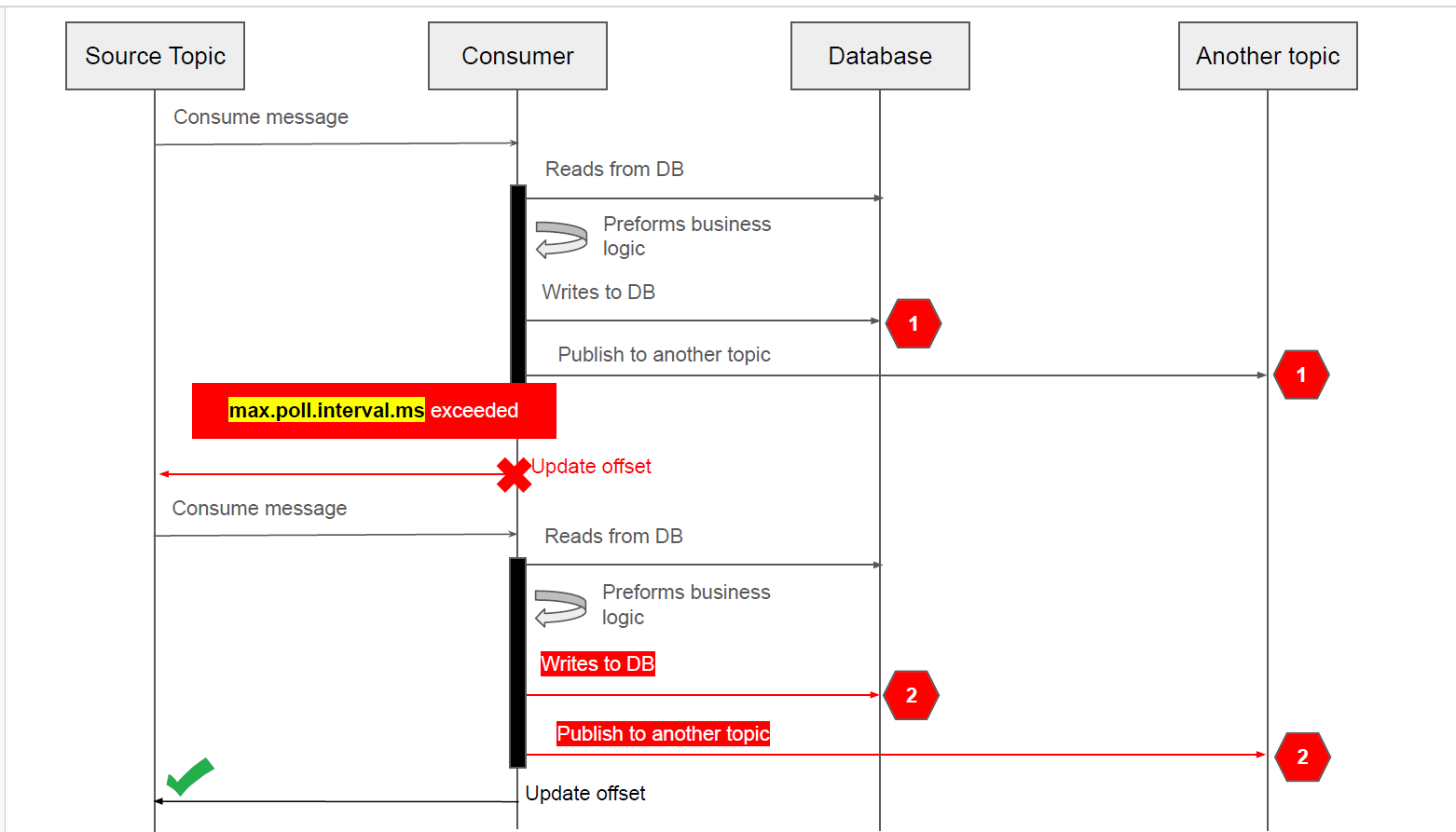
**Instance 2 assigned -> partition 2**



**Instance 3 is down**

Section 14 **Idempotent Consumer**

**An idempotent Apache Kafka consumer is a consumer that can process the same message multiple times without causing any side effects or data inconsistencies.**

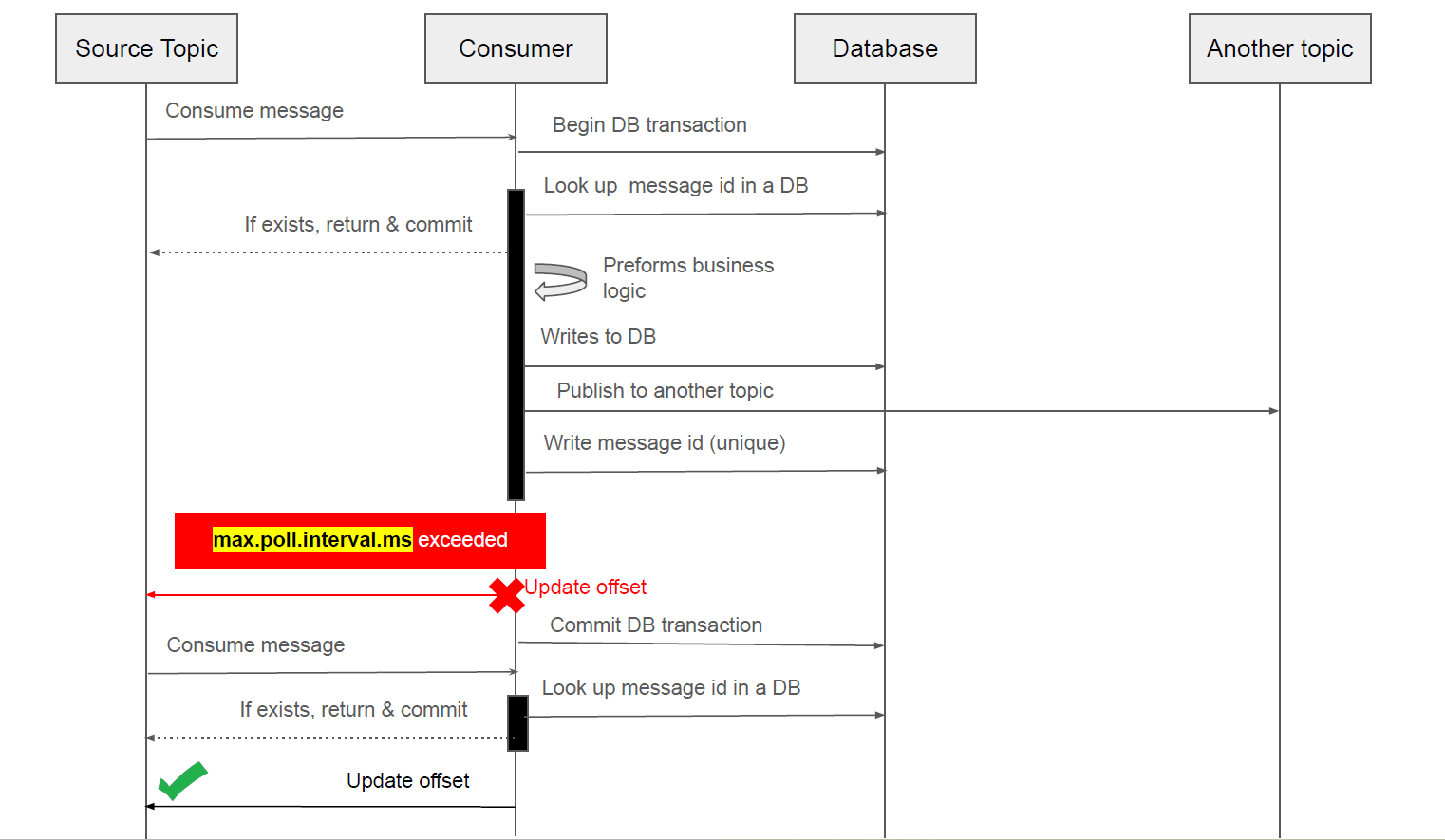
Problem when Consumer is Idempotent: ****

**In Order to avoid processing duplicate message:**

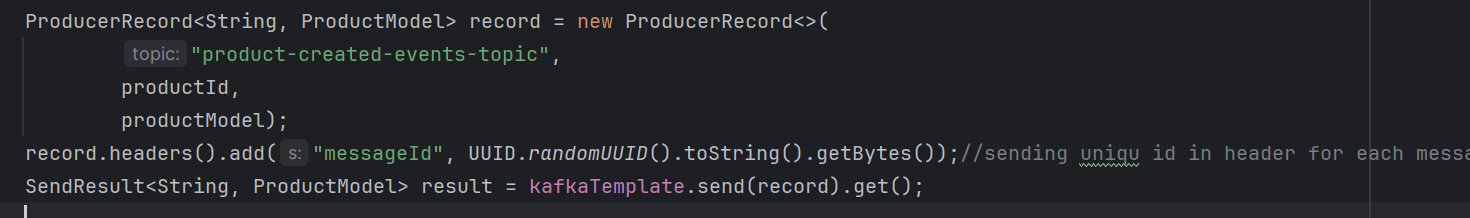
**● Idempotent Consumer**

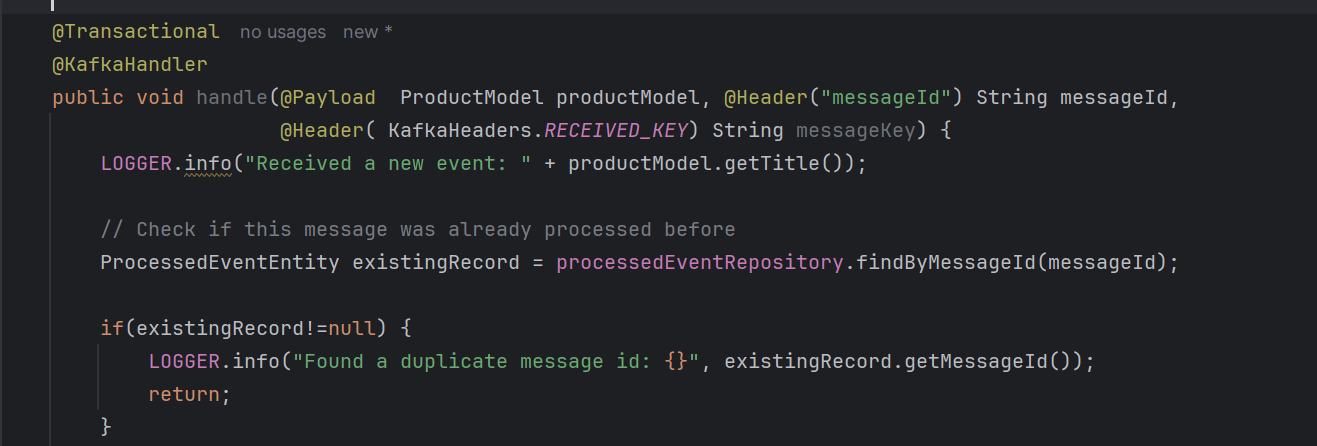
**● Idempotent Producer**

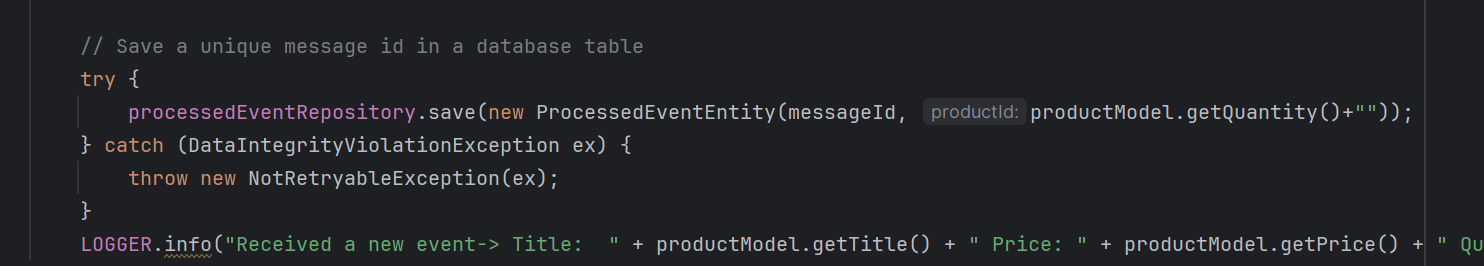
**● Transactions**

**Solution as architecture: Idempotent Consumer**  


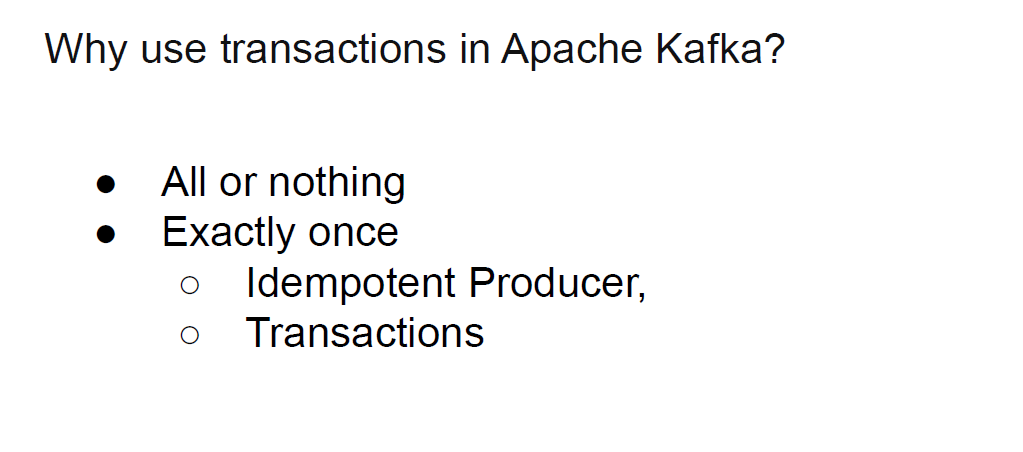
SpringBoot Chnages:

1)Producer Side changes: passing unique id for each msg :  


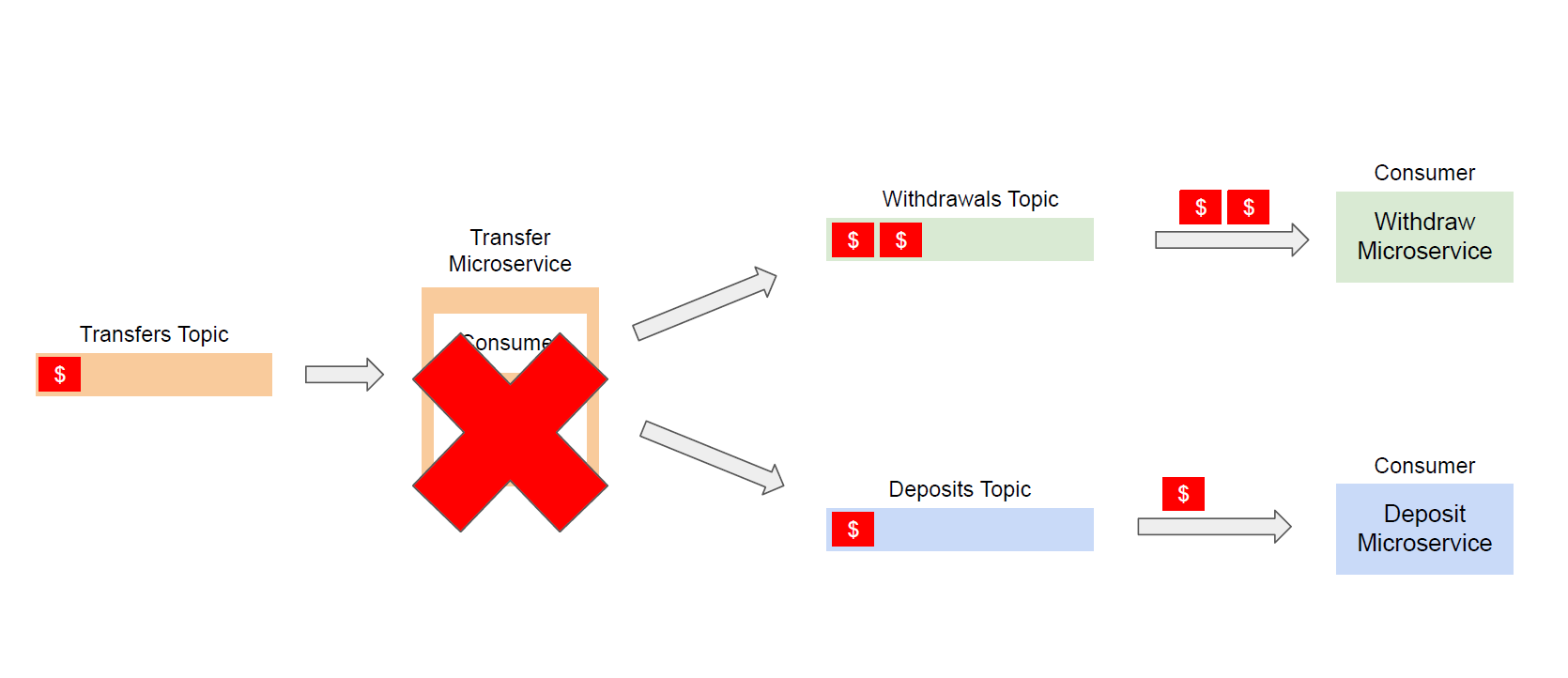
2) consumer side persiting it in db and every for new msg from consumer is validated from db where msg already consumed or not. As per above architecture.  


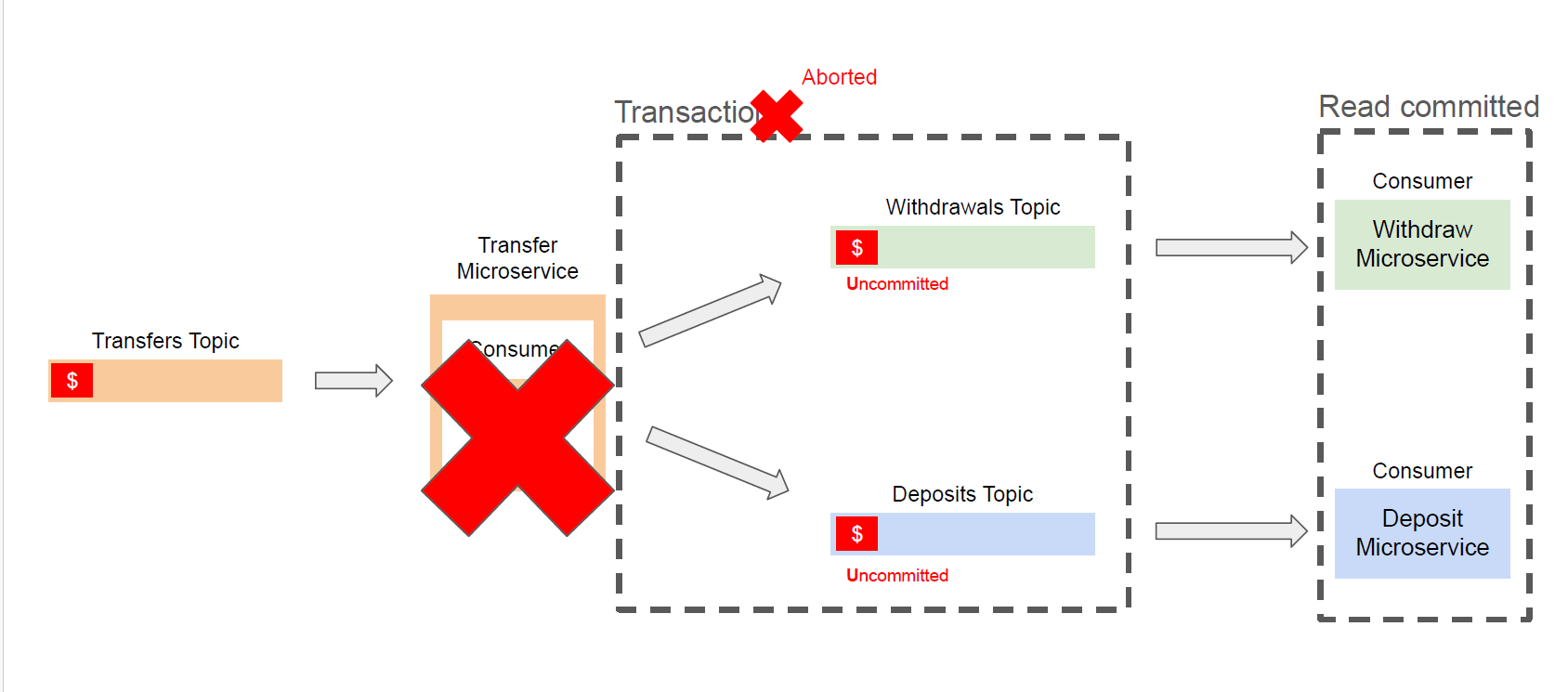
3) if it’s a new transaction save it.: in case bof failure it will start from beginning @transaction roll backs  


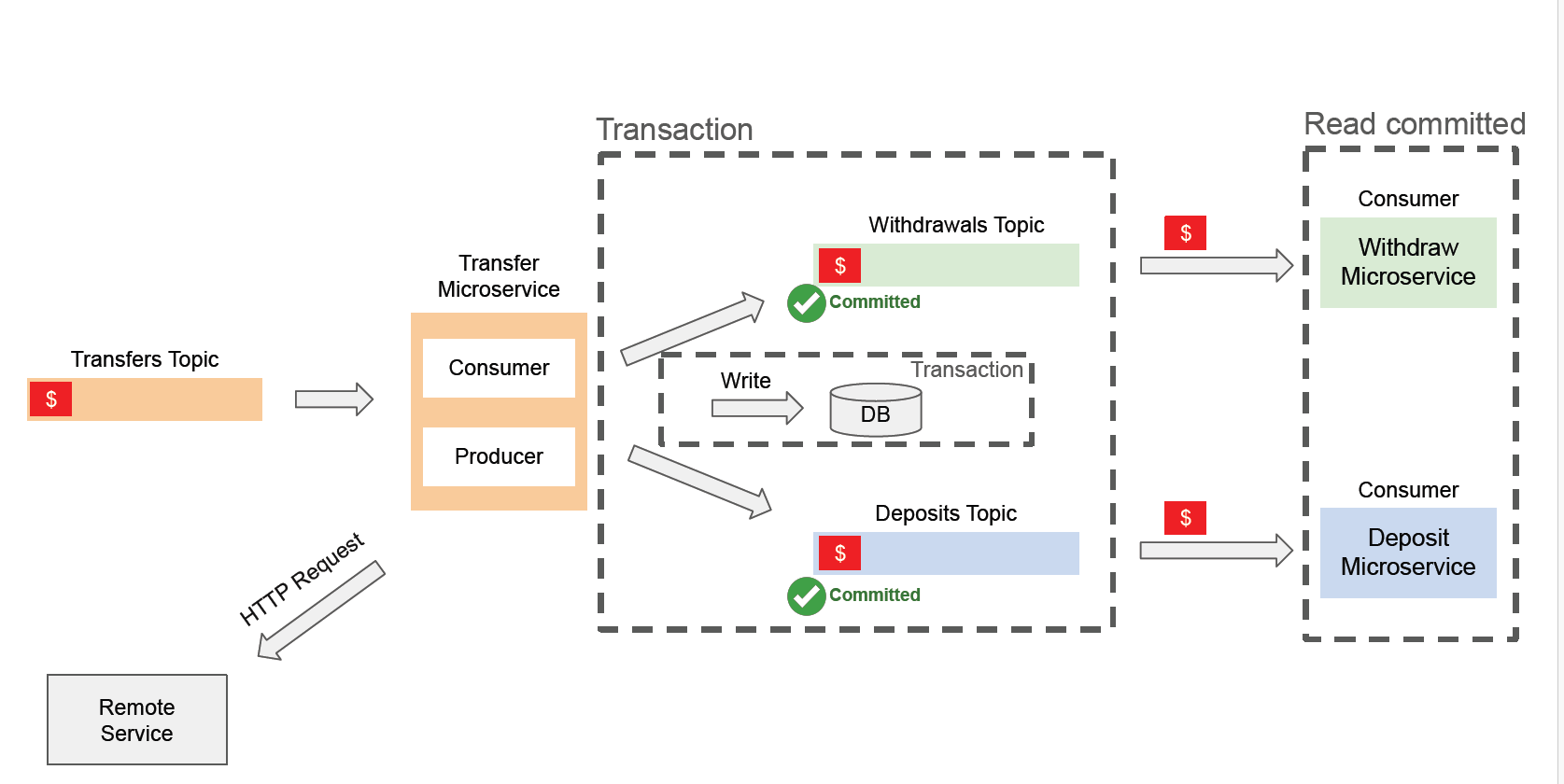
Section 15 **Apache kafka tramsaction**





Problem :

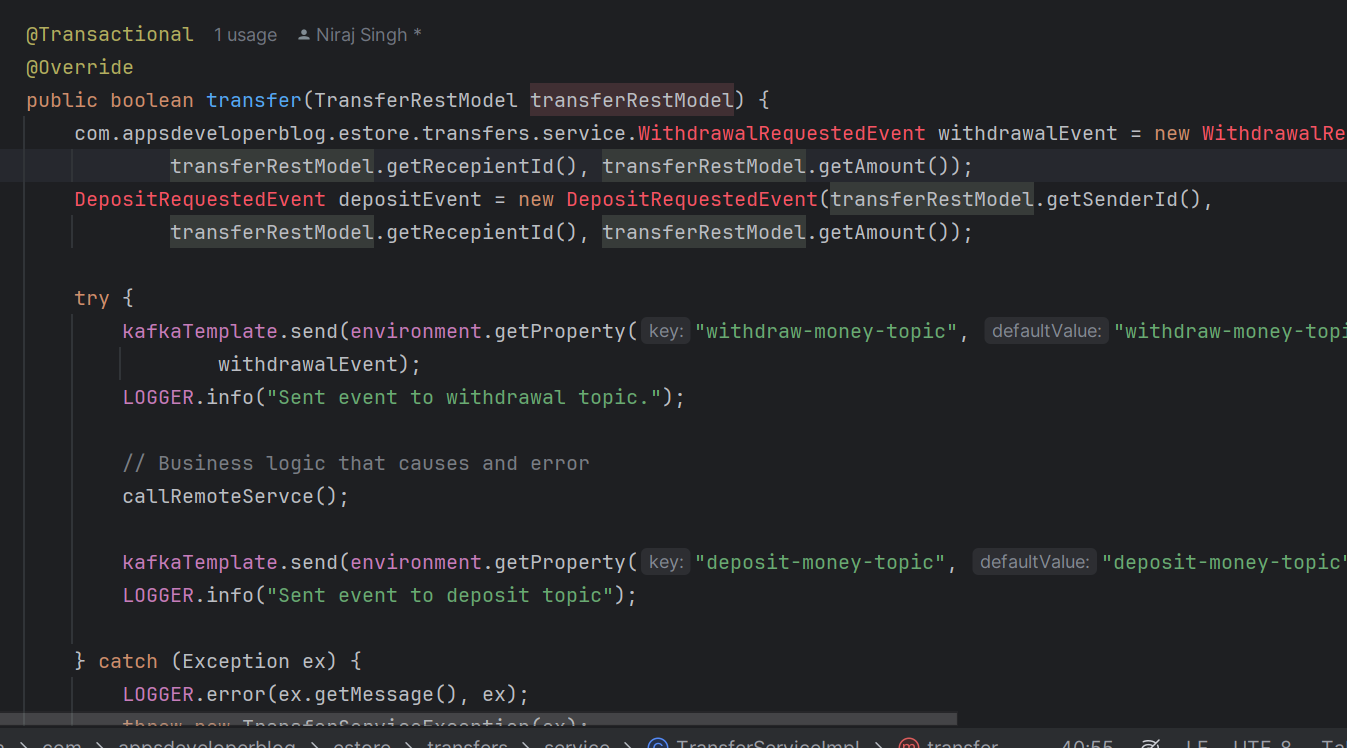
Solution:



Steps in Springboot :

1) spring.kafka.producer.transaction-id-prefix=transfer-service-${random.value}-

Above property tells kafka for to maintain transation

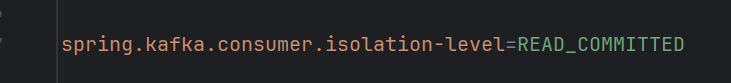
2)@Transactional  


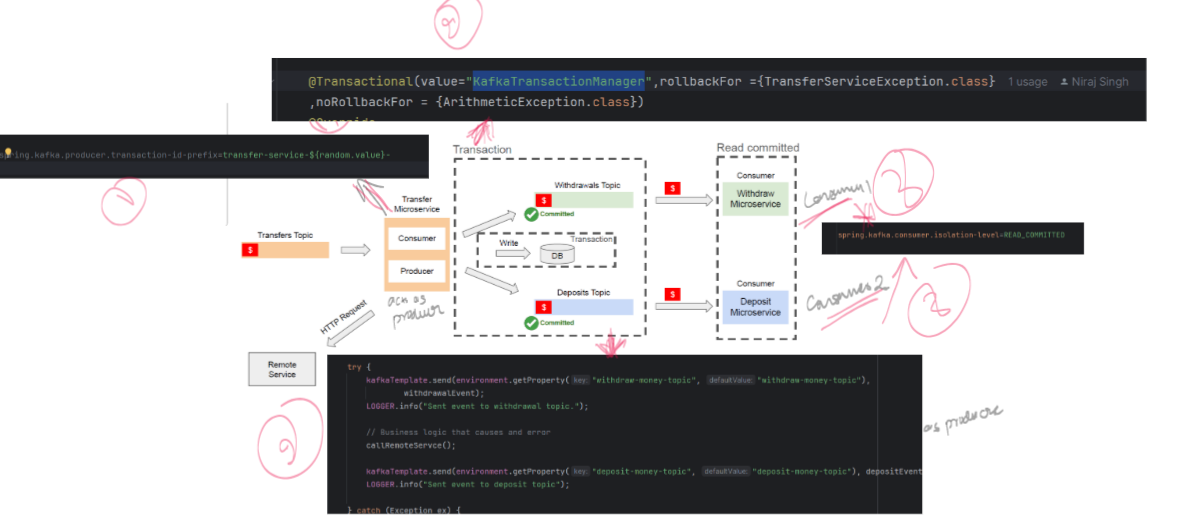
@Transactional(value="KafkaTransactionManager",rollbackFor ={TransferServiceException.class}  
,noRollbackFor = {ArithmeticException.class})

**Value** -> specifies to pic transaction manager which is defined in config class

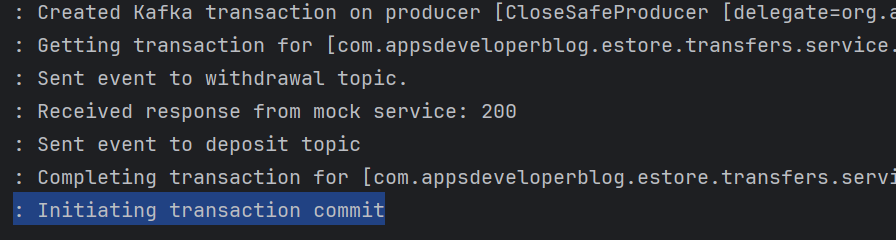
**rollbakFor** => we can specify which cchecked exception class for which current transaction shoul rolled back

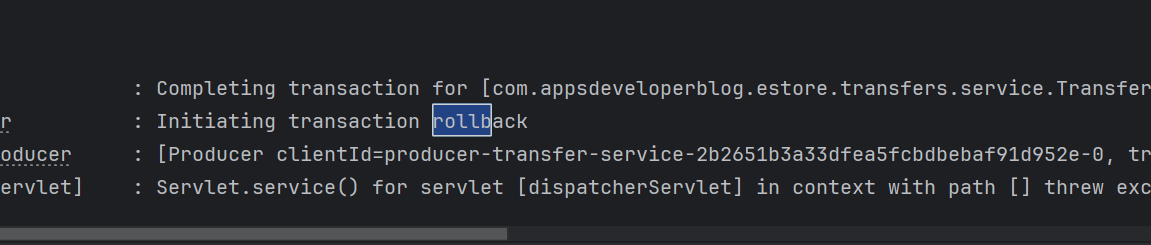
**norollbakFor** => we can specify which checked exception class for which current transaction should not rolled back

1. Consumer should use below propertyso that it reads only committed msg from topics

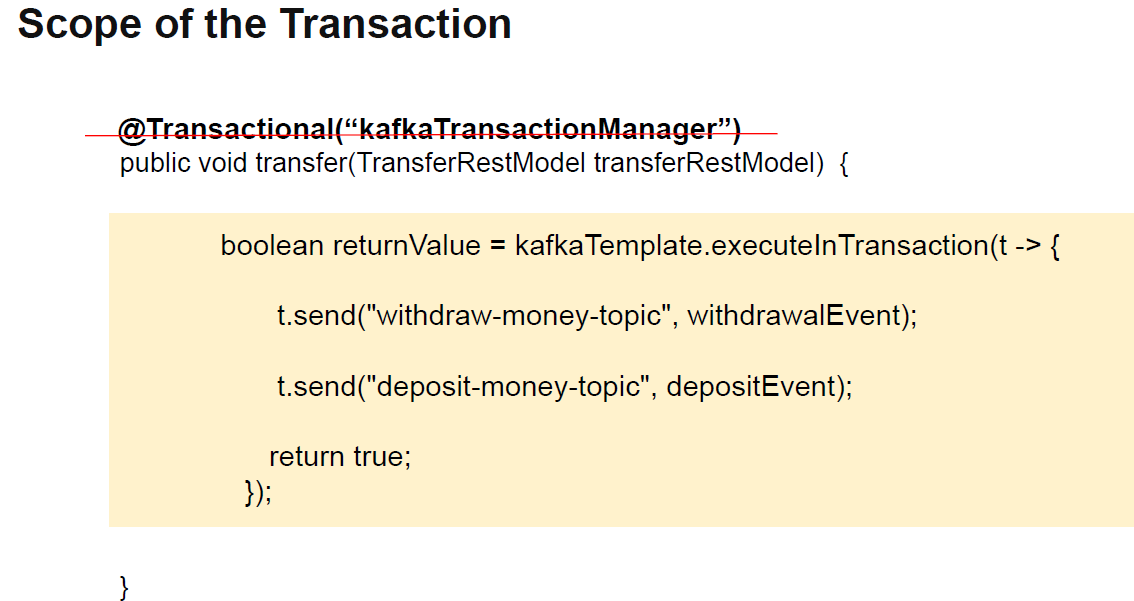


Final graphi:

In console it will look like :  


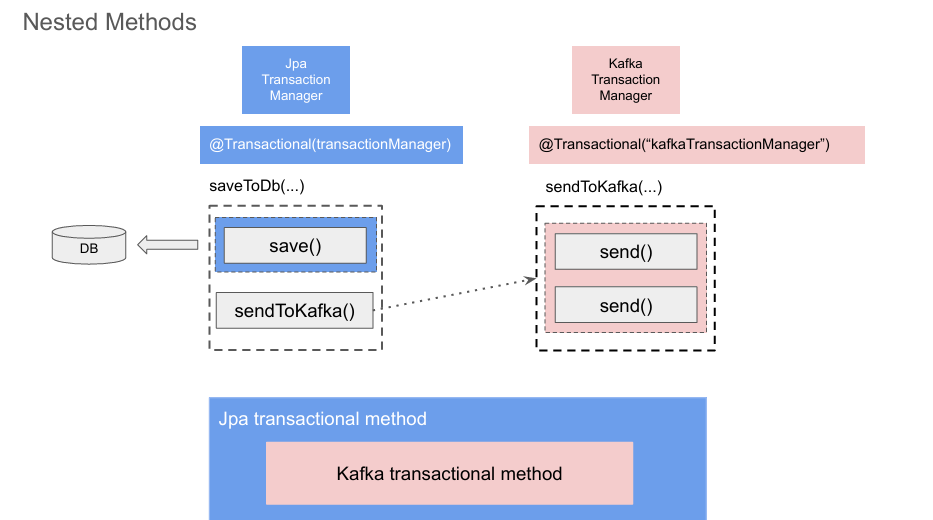
Incase of failure :  


In case you don’t want whole method to be in one transaction we can go through local tracation as well:



**Section 16 Apache Kafka and database Transaction**

**So kafka transaction don’t treat db transaction separately. We need to configure our structure way that both acts a single transaction.**

****

When method with **transactional annotation** is called **INTERCEPTOR** is invoked first, it will start transaction before executing the method. Now , when this **INTERCEPTOR** starts transaction kafka template object that we use to send kafka messages, it will synchronize its transaction with transaction that is started by this transaction manager(jpa transaction manager here).  
  
So Each kafka send method and db save save method, they will all participate in the same transaction.

And when our method exists, **db transaction will commit first**, followed by kafka transaction.

**In Spring boot:**

1. Created bean for jpa transaction manager and kafka transaction manager as well.

2)JpatrsactionManger provided to @Transaction method. So jpa transaction manager starts thetrsaction and kafka template will sync with jpa transaction and that is how we can achive synchronized trsanction between db and kafka

Important points:

1. **listeners** is how Kafka brokers know where to wait for connections, and **advertised.listeners** is how clients know where to find the broker.
2. **No of partition should be >= no of consumer (other wise no sense)**
3. **Replication factor should not be greater than no of brokers.**
4. **Message key ensures sequence of deliver of msg as it get stores in same partion.**
5. **Kafka manager or kafka cruise ui based project control n monitor kafka brokers and all**cloned the gradle project :

**Started it : .\gradlew.bat build**