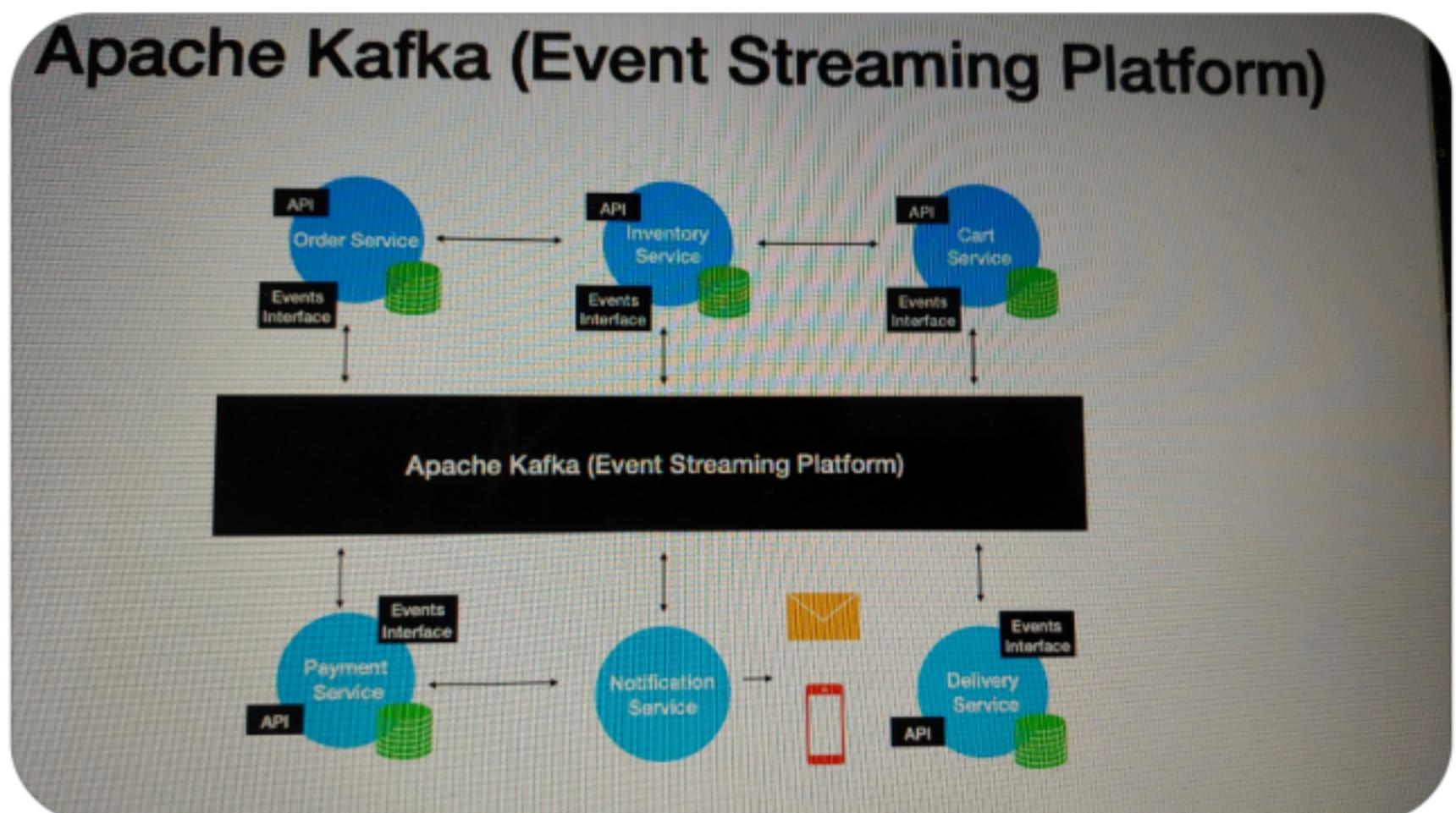


# Kafka

It is a distributed messaging or streaming system.



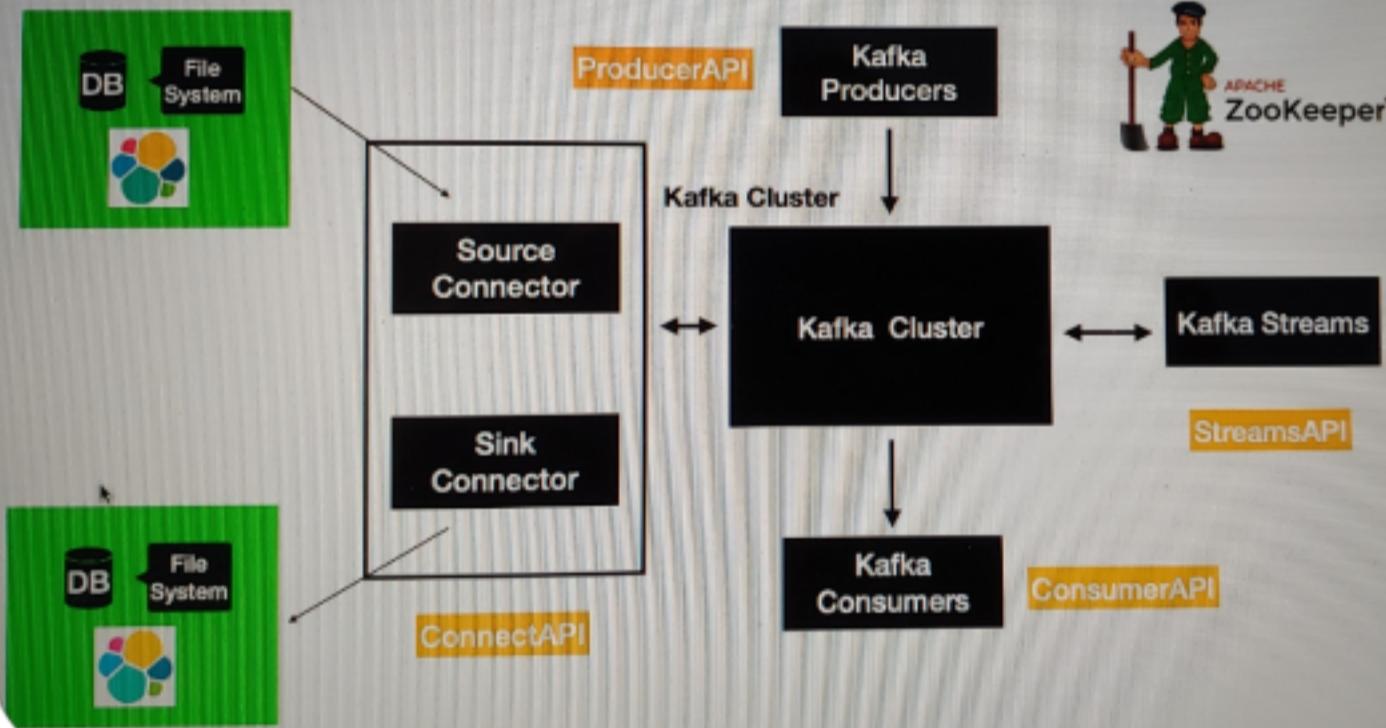
## Traditional Messaging System

- Transient Message Persistence
- Brokers responsibility to keep track of consumed messages
- Target a specific Consumer
- Not a distributed system

## Kafka Streaming Platform

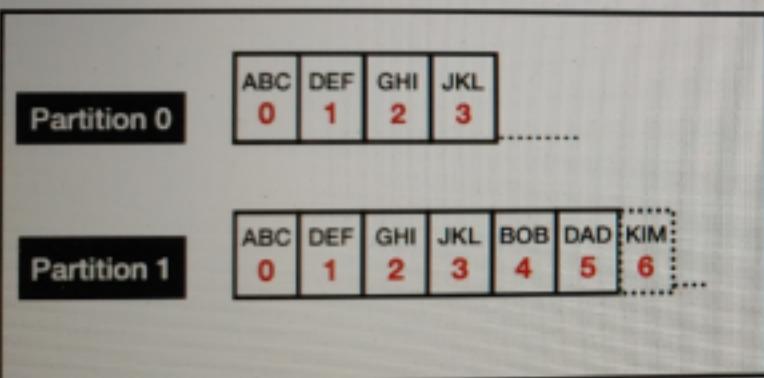
- Stores events based on a retention time. Events are Immutable
- Consumers Responsibility to keep track of consumed messages
- Any Consumer can access a message from the broker
- It's a distributed streaming system

## Kafka Terminology & Client APIs



# Topic and Partitions

## TopicA



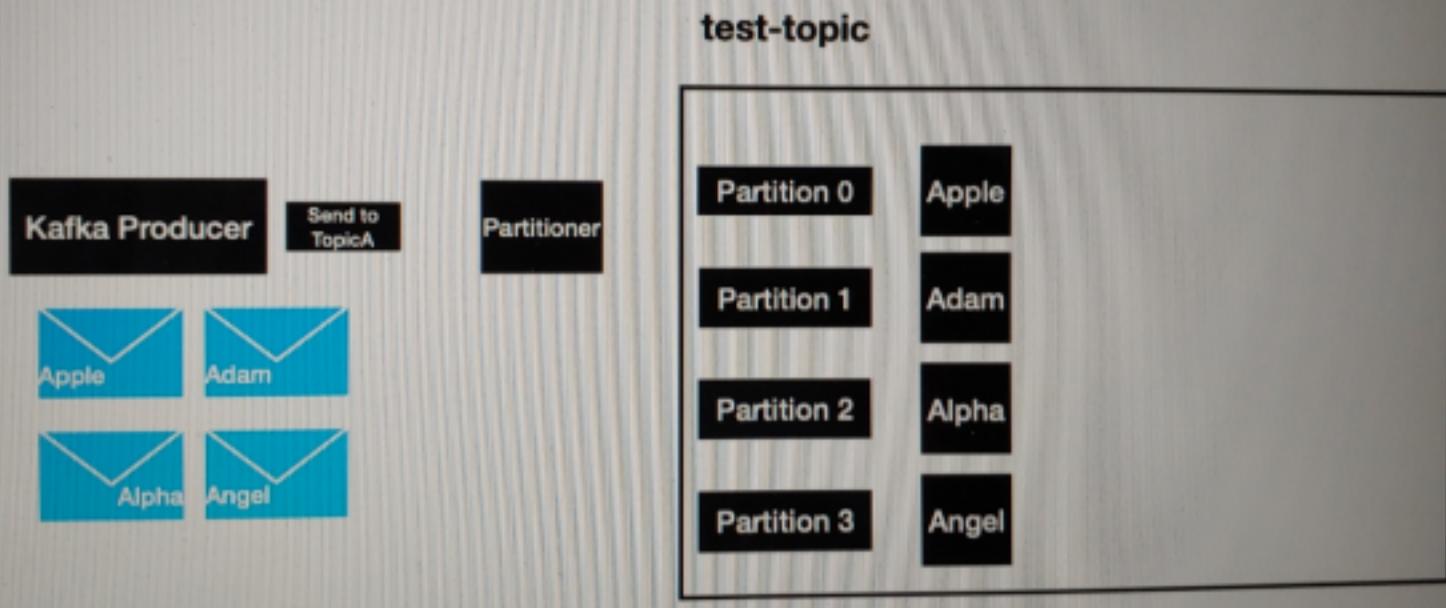
- Each Partition is an ordered , immutable sequence of records
- Each record is assigned a sequential number called **offset**
- Each partition is independent of each other
- Ordering is guaranteed only at the partition level
- Partition continuously grows as new records are produced
- All the records are persisted in a commit log in the file system where Kafka is installed

= ○ We can create topic and partition through script provided by kafka cli

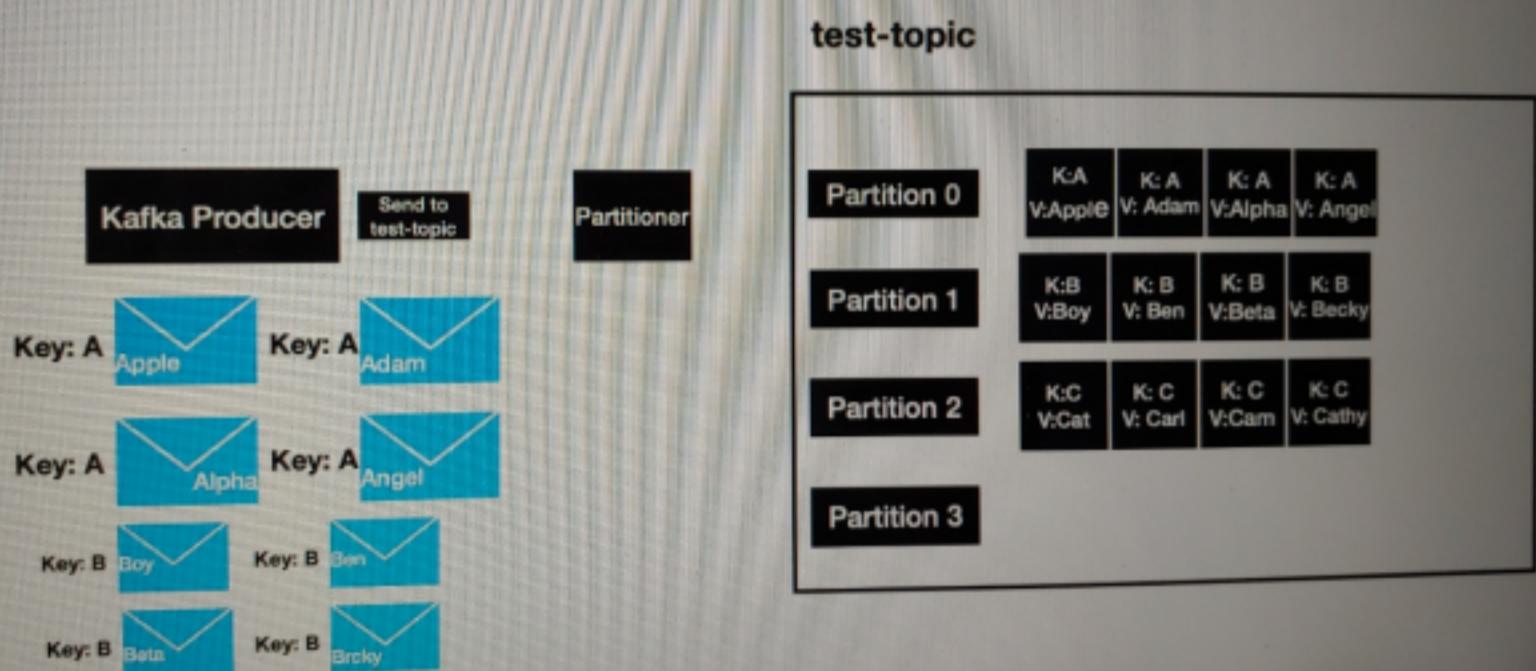
## Kafka Message

- Kafka Message these sent from producer has two properties
  - Key (optional)
  - Value

# Sending Message Without Key



# Sending Message With Key



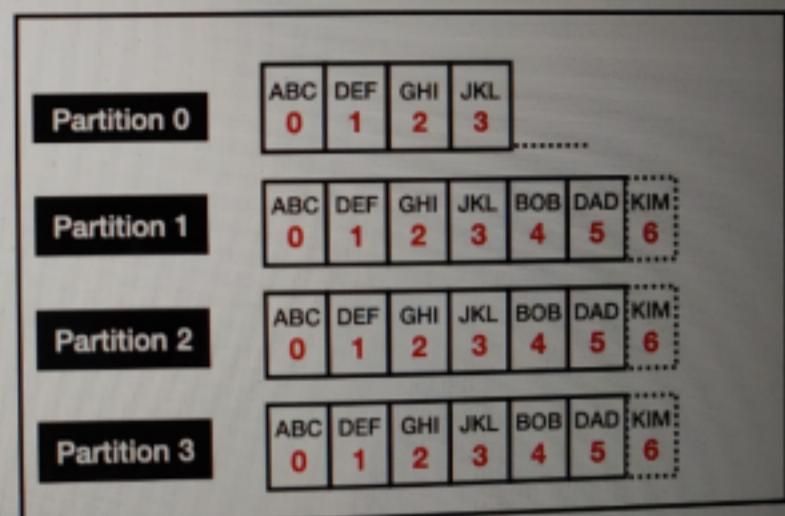
- ≡ ○ In order to maintain the order of the message which is sent to kafka topic use a specific key along with value. This will make sure the message will always

sent to the same partition with the same key

## Consumer Offsets

- Consumer have three options to read
  - from-beginning
  - latest
  - specific offset

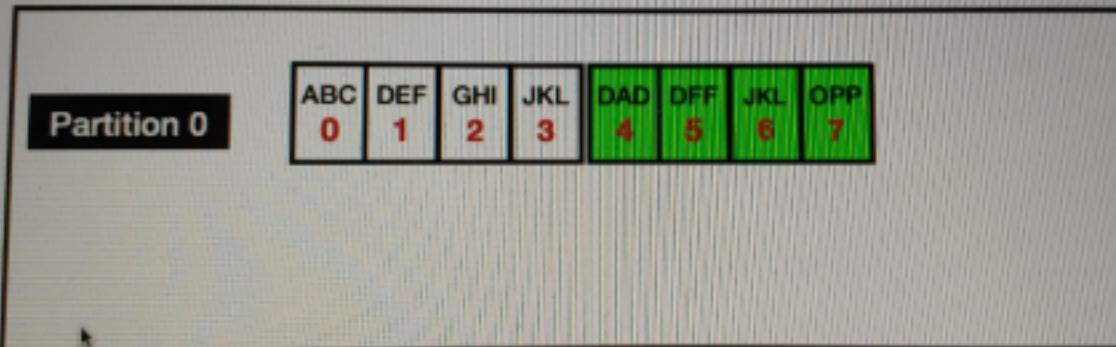
test-topic



- ≡ ○ Any message produced to topic has unique id called offset.
- ≡ ○ Consumer will use this offset information while reading any message
- ≡ ○ Consumer offset behaves like a bookmark for the consumer to start reading the message from the point it left off

# Consumer Offsets

test-topic

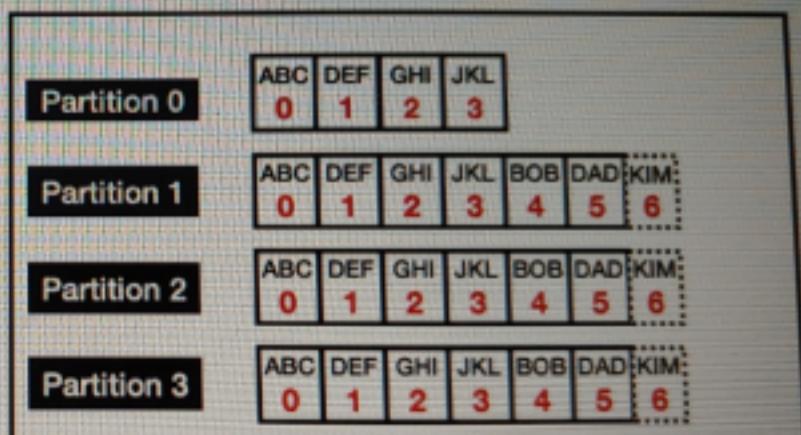


- ≡ ○ For each consumer, the offset information will be stored in an internal kafka topic called `_consumer_offsets`

# Consumer Groups

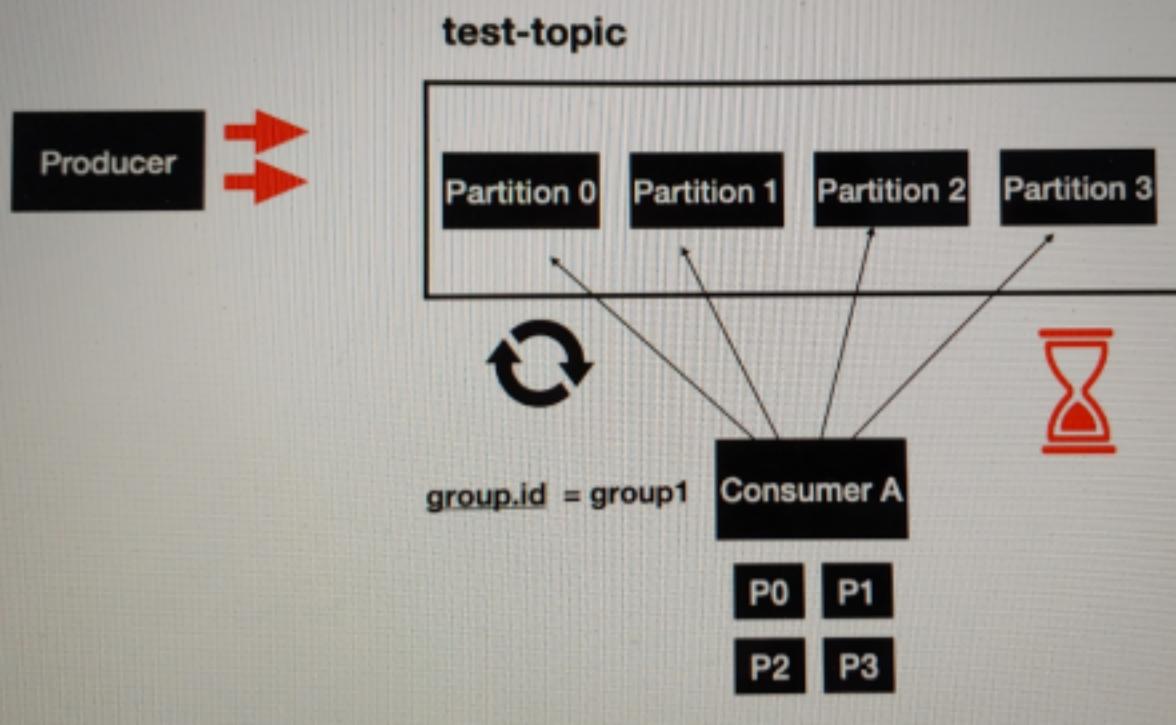
- `group.id` is mandatory
- `group.id` plays a major role when it comes to scalable message consumption.

test-topic



group.id = group1 Consumer 1

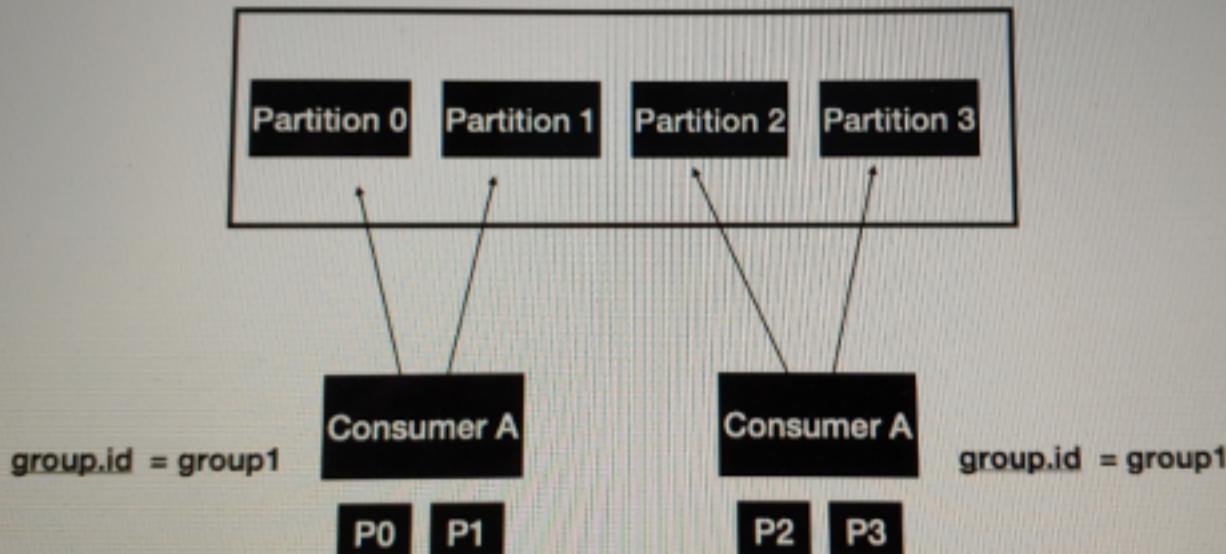
# Consumer Groups



- ≡ ○ If producer is producing records at higher rate and we have only one consumer per consumer group which is consuming records from all existing 4 partitions, then there will be lag at the consumer end in processing these records as each consumer will be running in a single thread

# Consumer Groups

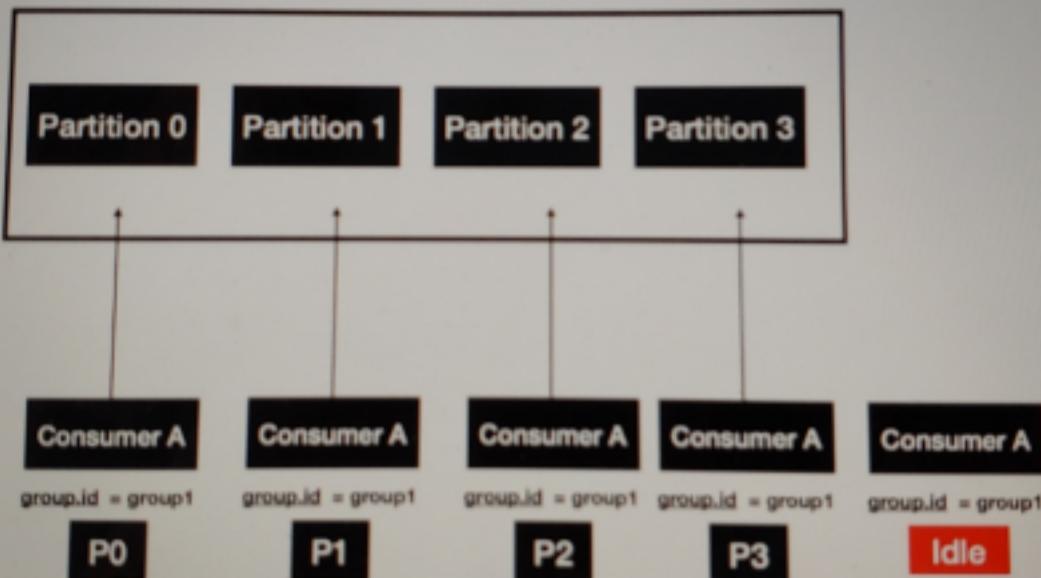
test-topic



- ≡ ○ Now if we increase number of consumer to 2 per same consumer group, each consumer will be consume and process the records from 2 partitions each which will increase the processing rate as both consumer will be executing parallelly

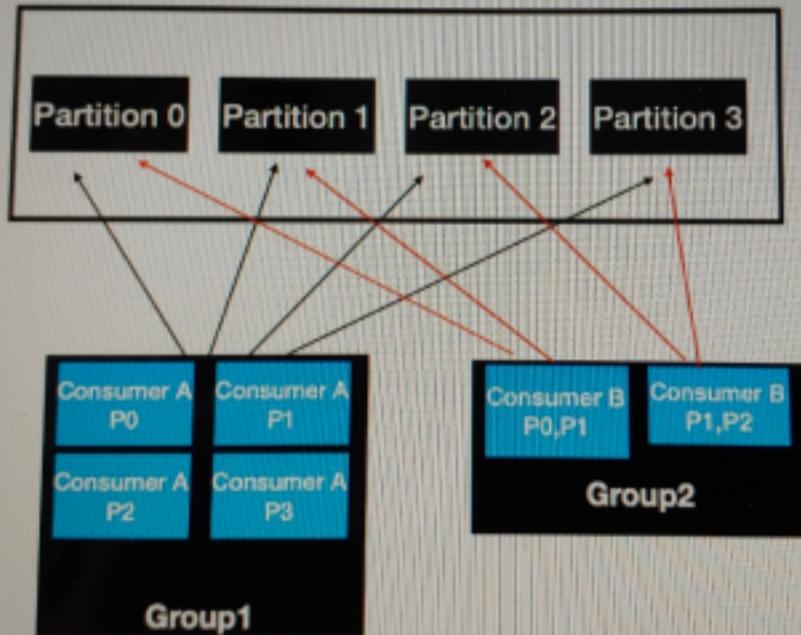
# Consumer Groups

test-topic



# Consumer Groups

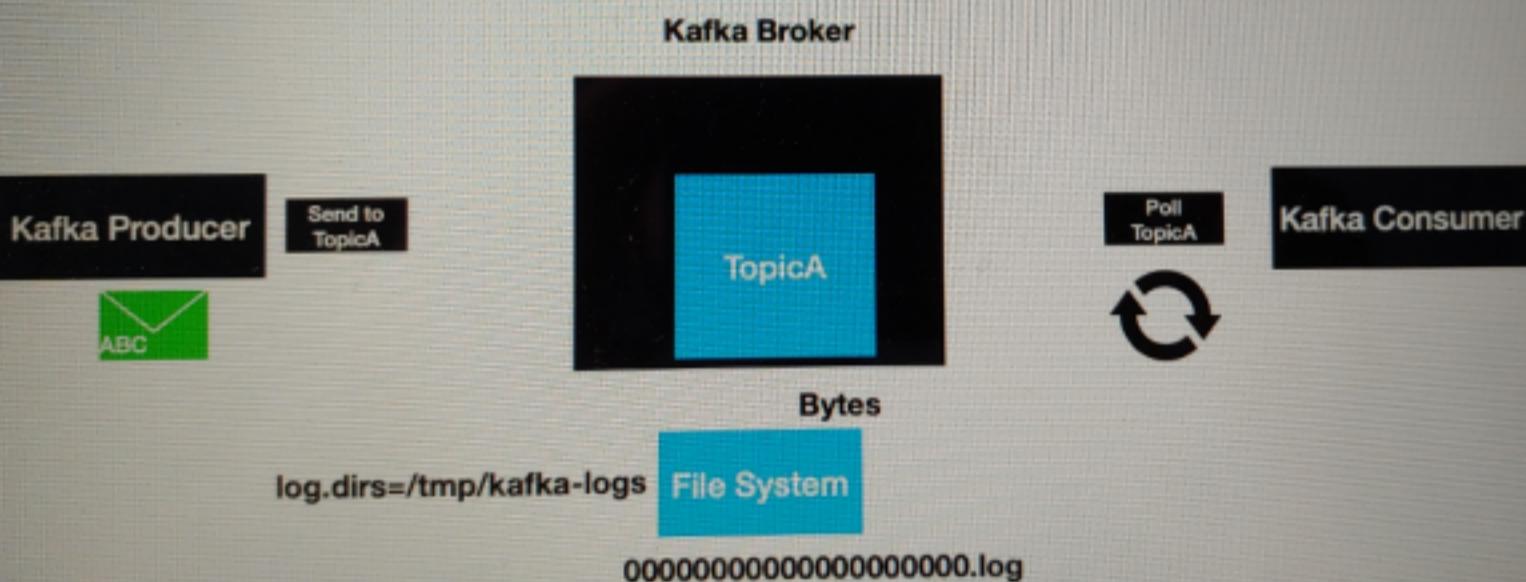
test-topic



# Consumer Groups : Summary

- Consumer Groups are used for scalable message consumption
- Each different application will have a unique consumer group
- Who manages the consumer group?
  - Kafka Broker manages the consumer-groups
  - Kafka Broker acts as a Group Co-ordinator

## Commit Log



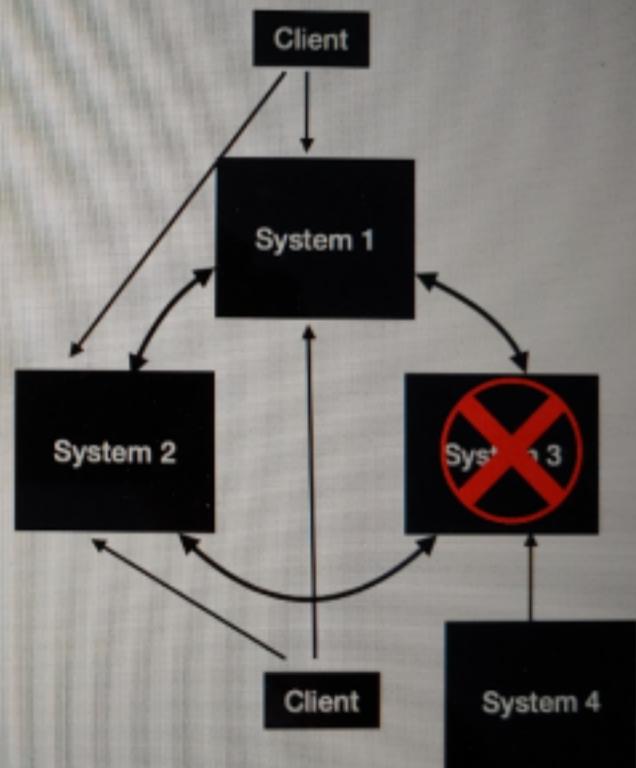
- = ○ Each partition has its own commit log in the local file system. Consumers can see these records only after those are committed into commit log.

# Retention Policy

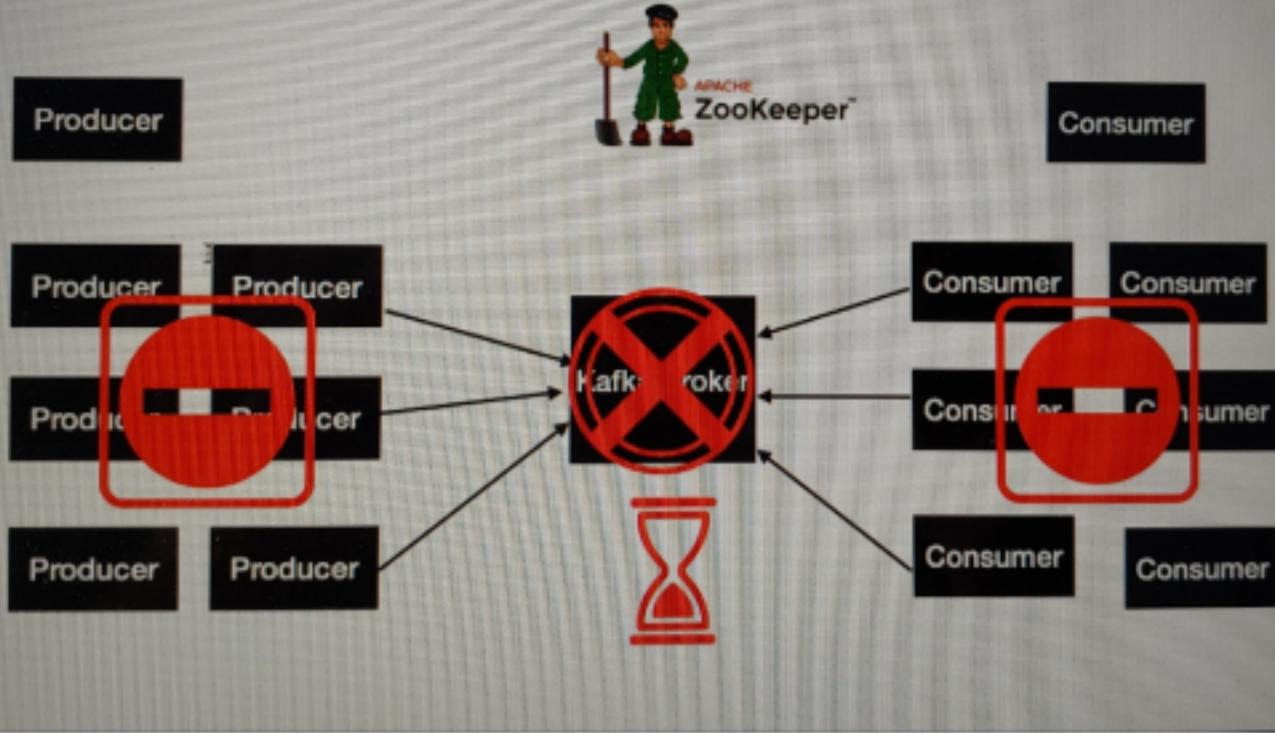
- Determines how long the message is retained ?
- Configured using the property **log.retention.hours** in **server.properties** file
- Default retention period is **168 hours** (7 days)

## Characteristics of Distributed System

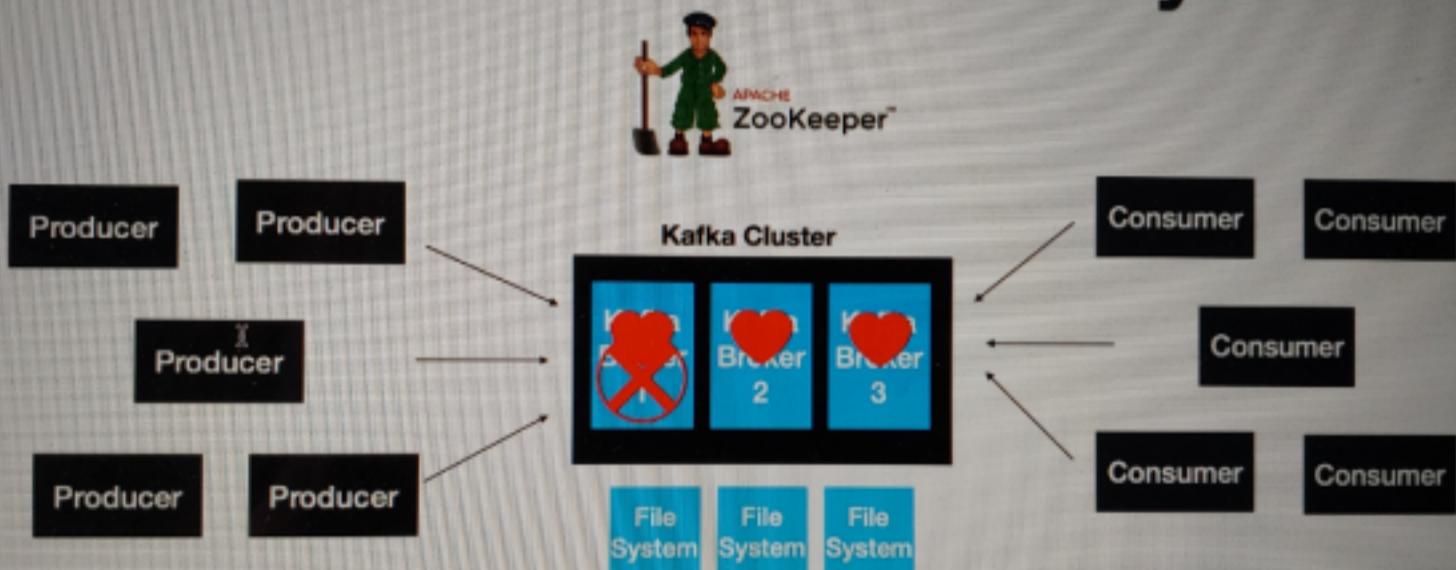
- Availability and Fault Tolerance
- Reliable Work Distribution
- Easily Scalable
- Handling Concurrency is fairly easy



# Kafka as a Distributed System



## Kafka as a Distributed System

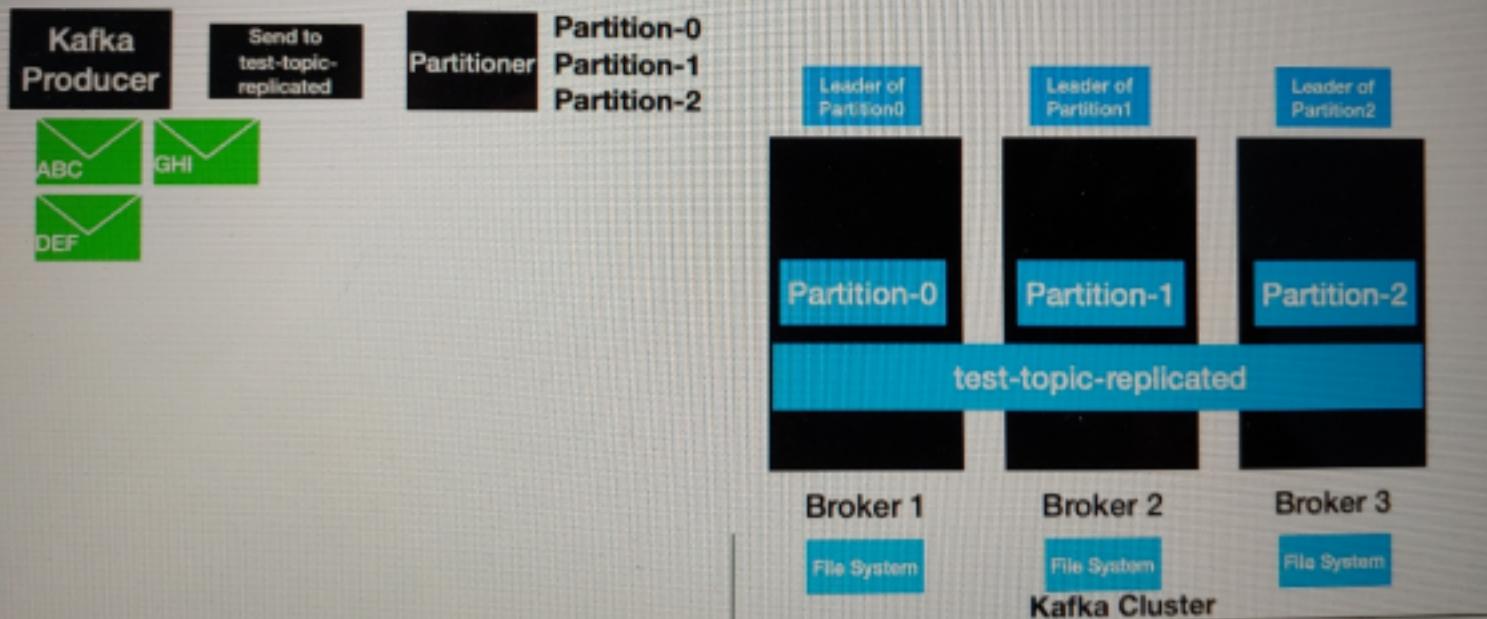


- Client requests are distributed between brokers
- Easy to scale by adding more brokers based on the need
- Handles data loss using Replication

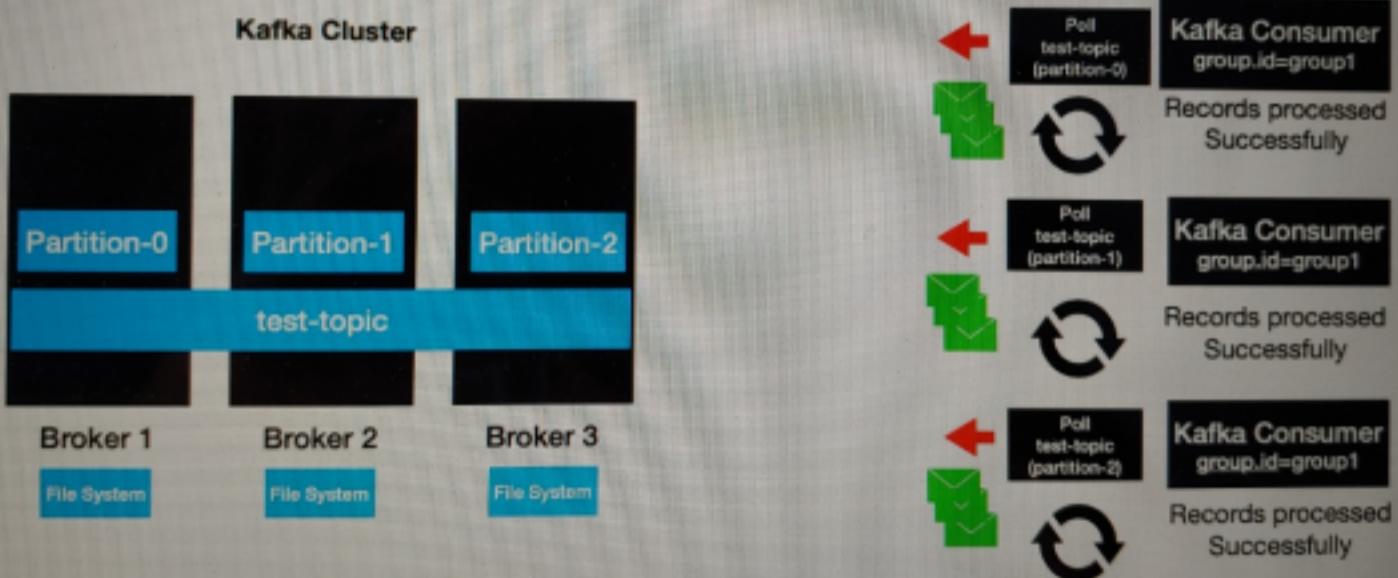
= ○ Distributing the available partition of a topic into available brokers of a kafka cluster is called leader assignments.

# How Kafka Distributes Client Requests?

## Kafka Producer



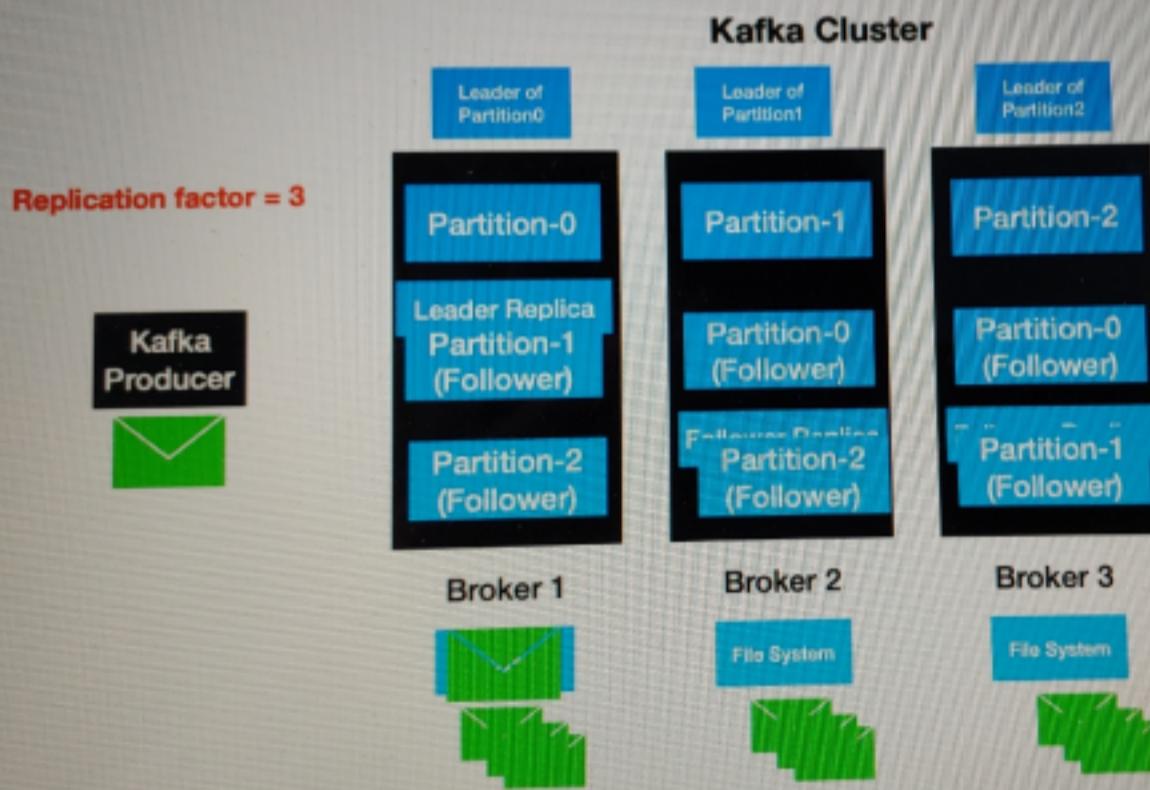
## How Kafka Distributes Client Requests? Kafka Consumer Groups



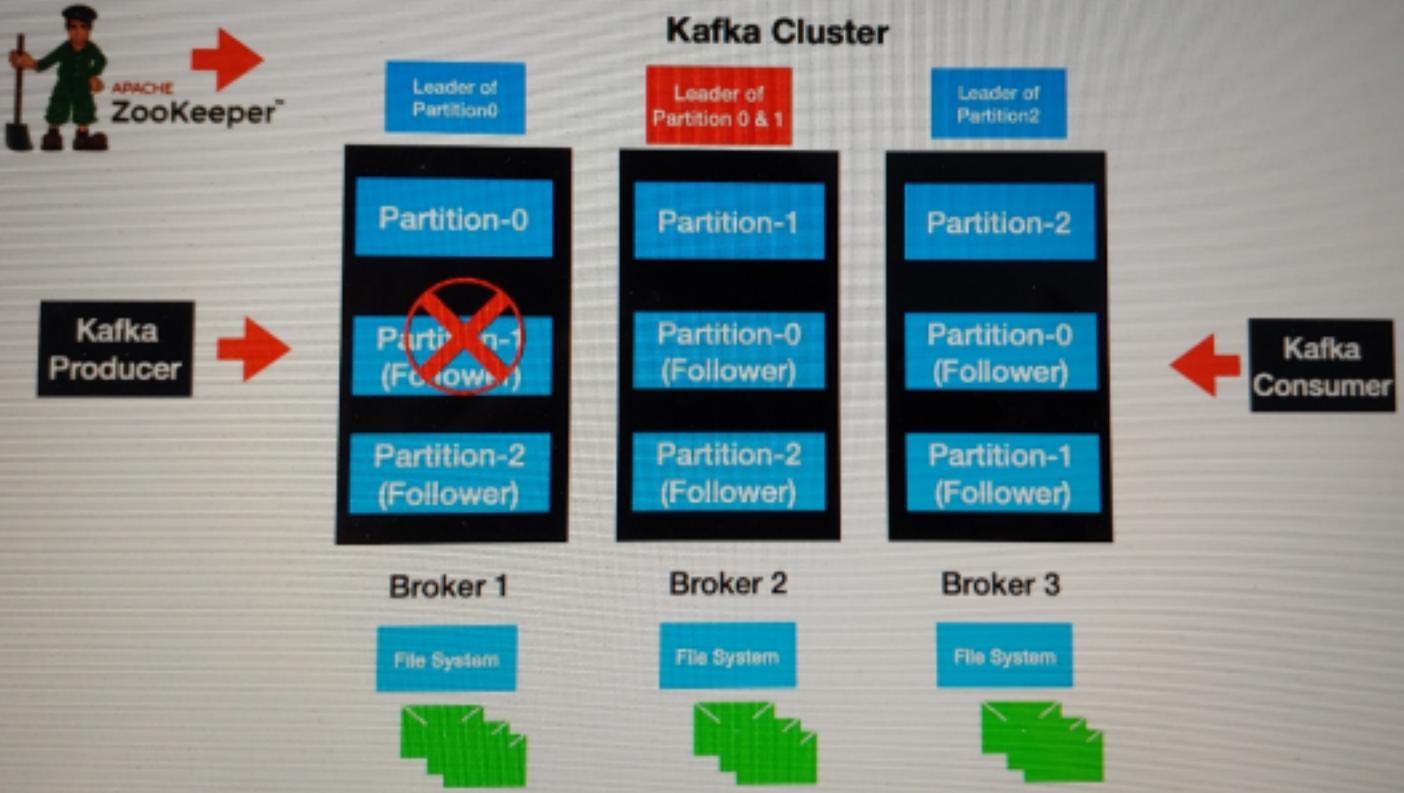
## Summary : How Kafka Distributes the Client Requests?

- Partition leaders are assigned during topic Creation
- Clients will only invoke leader of the partition to produce and consume data
  - Load is evenly distributed between the brokers

## Replication



# Replication

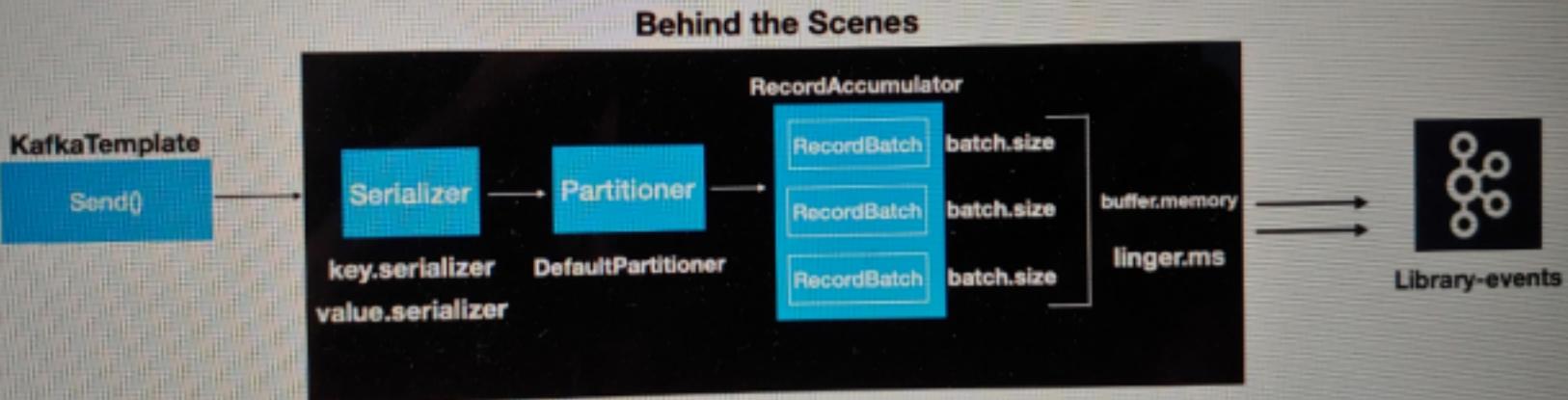


## In-Sync Replica(ISR)

- Represents the number of replica in sync with each other in the cluster
  - Includes both **leader** and **follower** replica
- Recommended value is always greater than 1
- Ideal value is **ISR == Replication Factor**
- This can be controlled by **min.insync.replicas** property
  - It can be set at the **broker** or **topic** level

= ○ Replication factor is equal to number of copies of same message

# KafkaTemplate.send()



- ≡ ○ linger.ms to send batch of records to consumer before batch.size limit reaches

## KafkaAdmin

- Create topics Programmatically
- Part of the **SpringKafka**
- How to Create a topic from Code?
  - Create a Bean of type **KafkaAdmin** in SpringConfiguration
  - Create a Bean of type **NewTopic** in SpringConfiguration

# @KafkaListener

- This is the easiest way to build Kafka Consumer
- KafkaListener Sample Code

```
@KafkaListener(topics = {"${spring.kafka.topic}"})
public void onMessage(ConsumerRecord<Integer, String> consumerRecord) {
    log.info("OnMessage Record : {}", consumerRecord);
}
```

- Configuration Sample Code

```
@Configuration
@EnableKafka
@Slf4j
public class LibraryEventsConsumerConfig {
```

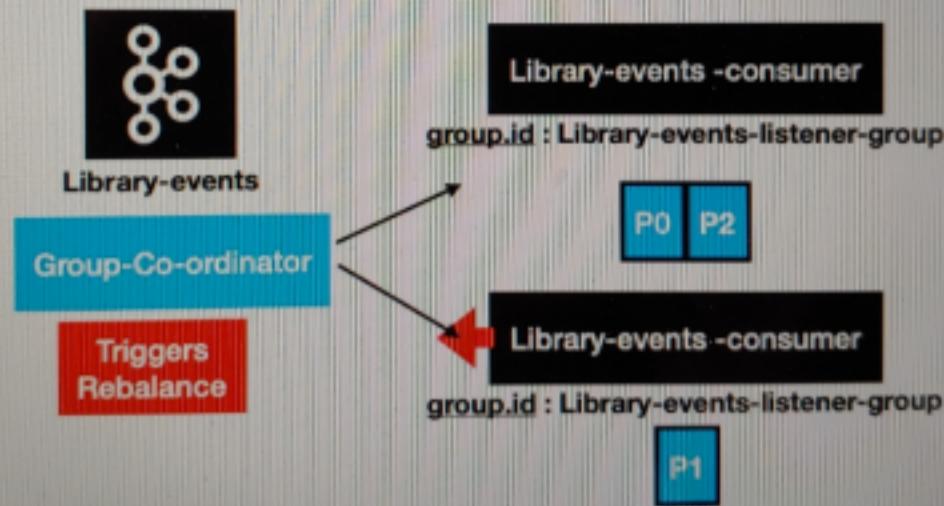
## KafkaConsumer Config

```
key-deserializer: org.apache.kafka.common.serialization.IntegerDeserializer
value-deserializer: org.apache.kafka.common.serialization.StringDeserializer
group-id: library-events-listener-group
```

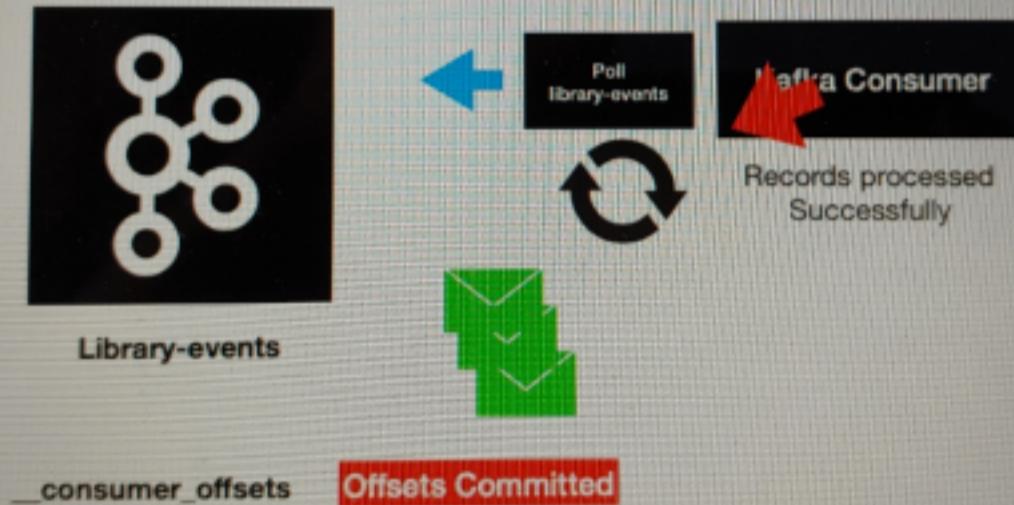
- = ○ Consumer groups: multiple instances of same application with the same group id

# Rebalance

- Changing the partition ownership from one consumer to another



## Committing Offsets



## Summary

- ≡ ○ Use KafkaTemplate<Long, String> to send messages.  
KafkaTemplate.send(topicname,

key, value)

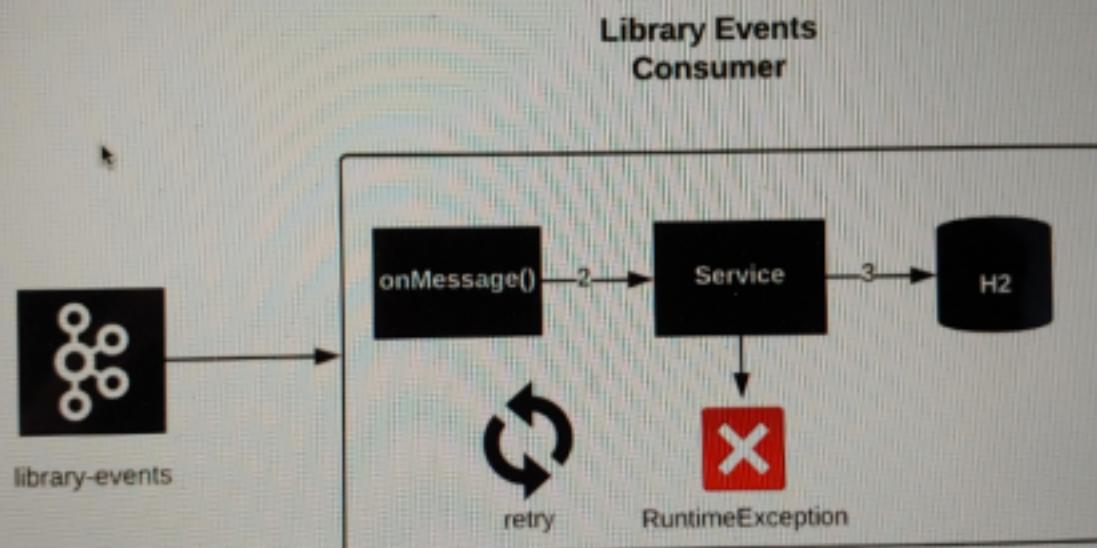
- ≡ ○ Wrap KafkaTemplate with configuration of producer
- ≡ ○ Use ProducerRecord api provided by plugin to wrap topic, key, value and any RecordHeader details
- ≡ ○ Add callback handler for both success and failure in KafkaTemplate.send()
- ≡ ○ Use @KafkaListener with a single or multiple topic names which can consume the ConsumerRecord<Long, String>
- ≡ ○ By default consumer will use auto commit with batch option. So as soon as consumer reads the data offset will be committed in kafka and moves to next offset.
- ≡ ○ To make manual commit use the below approach

```
import ...  
@Component  
@Slf4j  
public class LibraryEventsConsumerManualOffset implements AcknowledgingMessageListener<Integer, String> {  
  
    @Override  
    @KafkaListener(topics = {"library-events"})  
    public void onMessage(ConsumerRecord<Integer, String> consumerRecord, Acknowledgment acknowledgment) {  
        log.info("ConsumerRecord : {}", consumerRecord);  
        acknowledgment.acknowledge();  
    }  
}
```

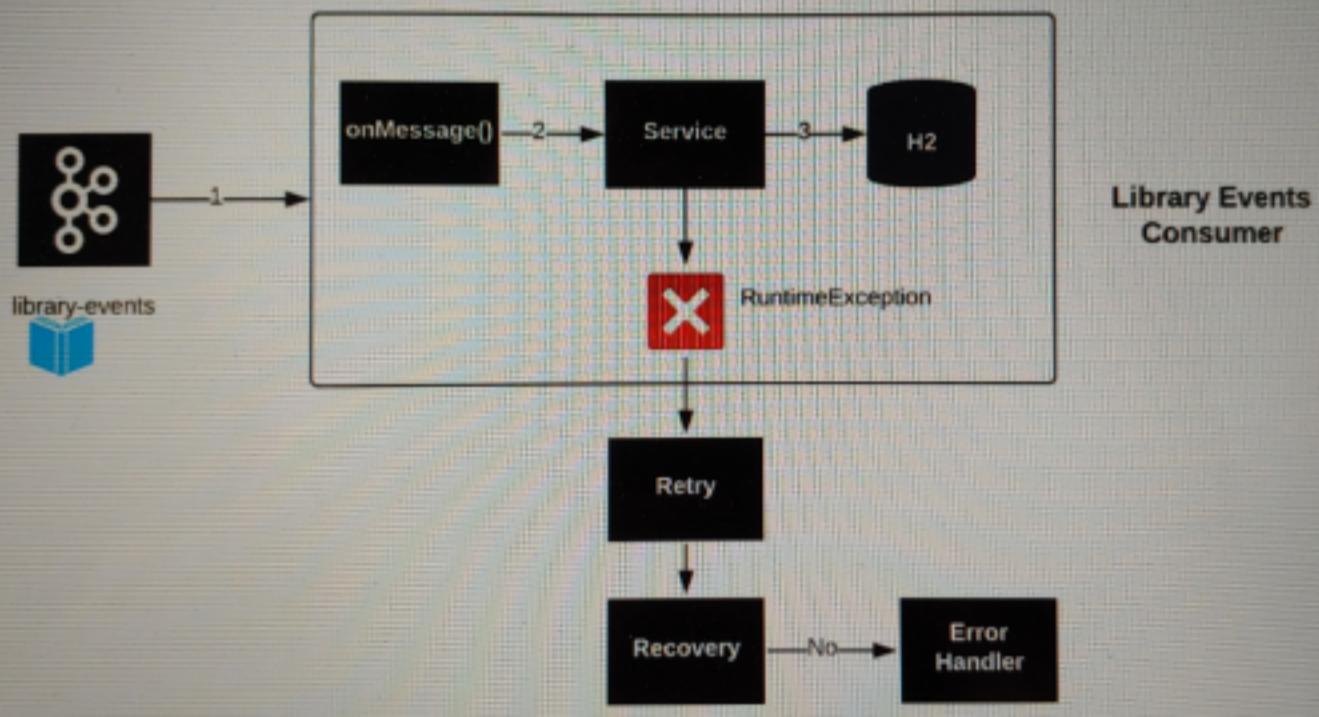
I

- ≡ ○ We can also configure error handler, retry all or specific exception (through spring retry RetryTemplate) and recovery (in case all retry are done but still exception throws) in kafka consumer and producer
- ≡ ○ kafka consumer

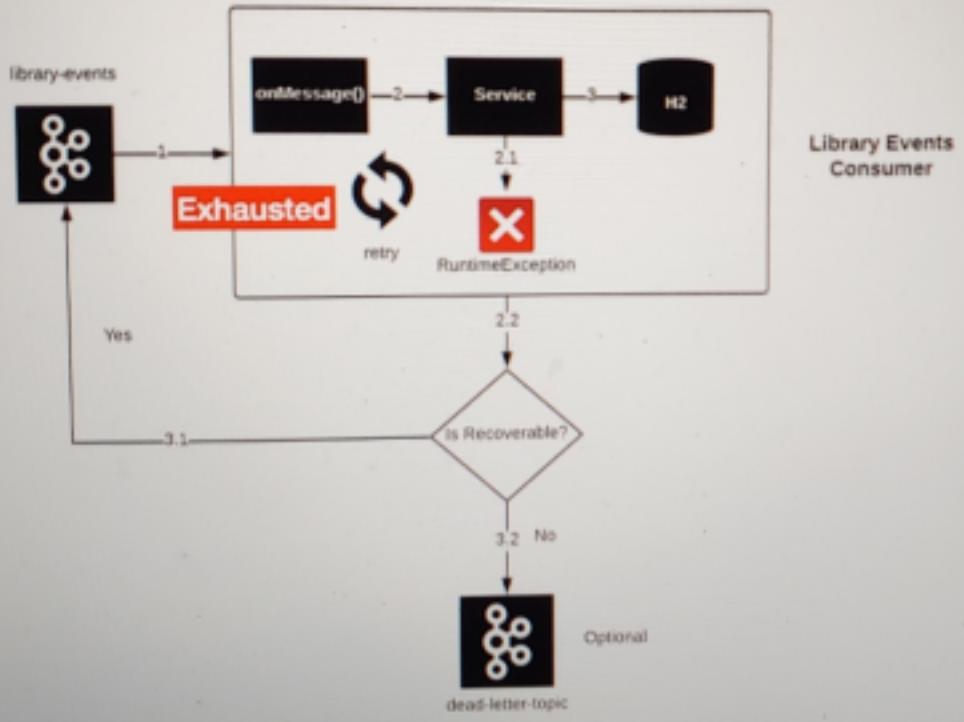
# Retry in Kafka Consumer



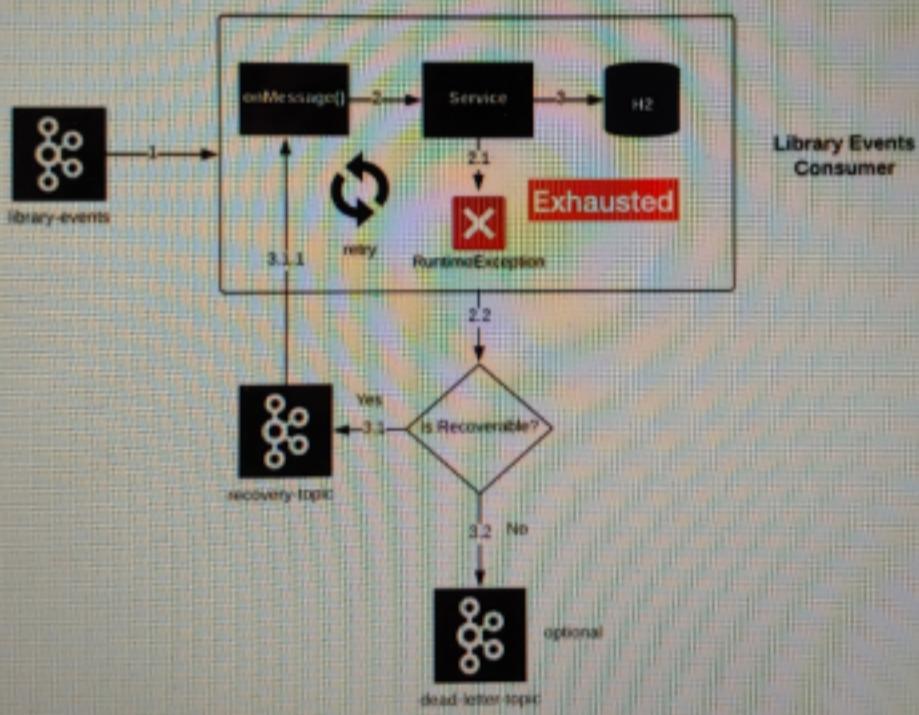
## Retry and Recovery



# Recovery - Type 1



# Recovery - Type 2



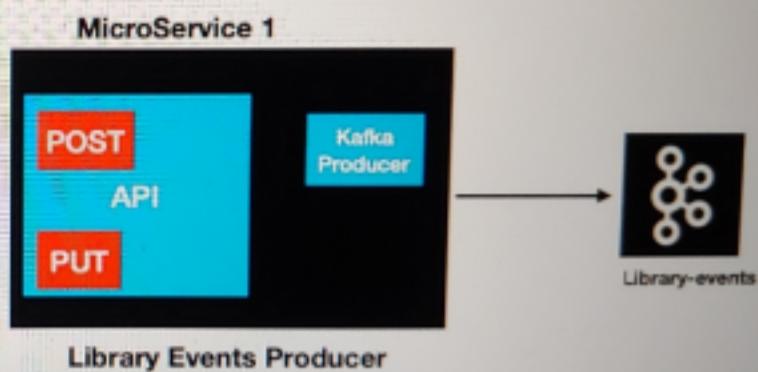
```
9     @ConditionalOnMissingBean(name = "kafkaListenerContainerFactory")
10    ConcurrentKafkaListenerContainerFactory<?, ?> kafkaListenerContainerFactory(
11        ConcurrentKafkaListenerContainerFactoryConfigurer configurer,
12        ObjectProvider<ConsumerFactory<Object, Object>> kafkaConsumerFactory) {
13        ConcurrentKafkaListenerContainerFactory<Object, Object> factory = new ConcurrentKafkaListenerContainerFactory<Object, Object>();
14        configurer.configure(factory, kafkaConsumerFactory
15            .getIfAvailable(() -> new DefaultKafkaConsumerFactory<Object, Object>(this.kafkaProperties.buildConsumerProperties())));
16        factory.setConcurrency(3);
17        // factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL);
18        factory.setErrorHandler(((thrownException, data) -> {
19            log.info("Exception in consumerConfig is {} and the record is {}", thrownException.getMessage(), data);
20            //persist
21        }));
22        factory.setRetryTemplate(retryTemplate());
23        factory.setRecoveryCallback((context -> {
24            if(context.getLastThrowable().getCause() instanceof RecoverableDataAccessException){
25                //invoke recovery logic
26                log.info("Inside the recoverable logic");
27                Arrays.asList(context.getAttributeNames())
28                    .forEach(attributeName -> {
29                        log.info("Attribute name is : {}", attributeName);
30                        log.info("Attribute Value is : {}", context.getAttribute(attributeName));
31                    });
32            }
33
34            ConsumerRecord<Integer, String> consumerRecord = (ConsumerRecord<Integer, String>) context.getAttribute("record");
35            libraryEventsService.handleRecovery(consumerRecord);
36        }) );
37        log.info("Inside the non recoverable logic");
38        throw new RuntimeException(context.getLastThrowable().getMessage());
39    }
40
41    return null;
42});
43
44 return factory;
```

## ≡ ○ Kafka producer

Kafka producer can wait continuously until any one of the broker is available and then send the message to it without stopping the server

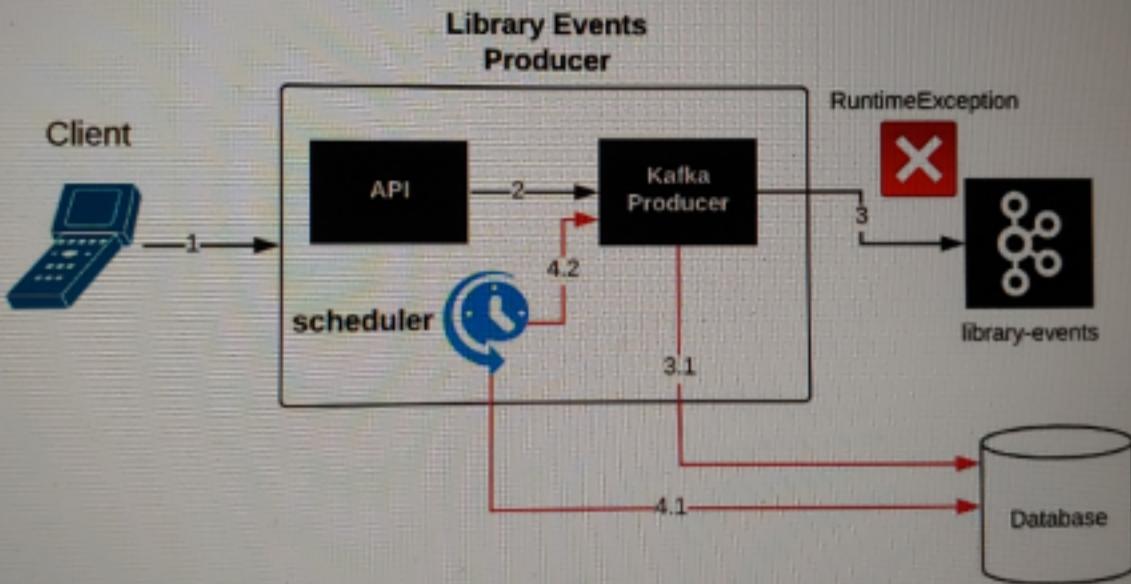
# Kafka Producer Errors

- Kafka Cluster is not available
- If `acks= all` , some brokers are not available
- `min.insync.replicas` config
  - Example : `min.insync.replicas = 2`, But only one broker is available



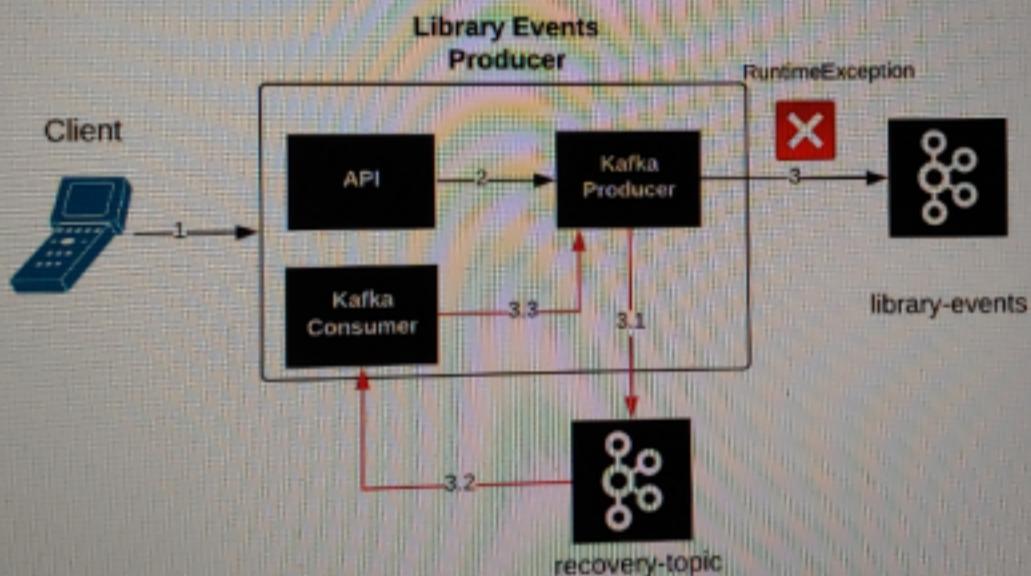
≡ ○ To retain or recover the records if it failed to send to kafka topic we can save it any database and then through scheduler we can resend it to kafka topic or else we can publish those records into other kafka topics

# Retain/Recover Failed Records



# Retain/Recover Failed Records

## Producer Misconfiguration - Option 2



MongoDB