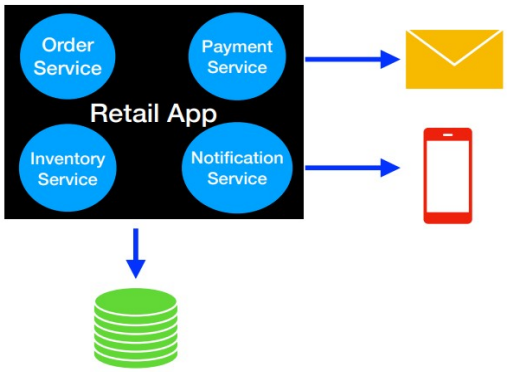
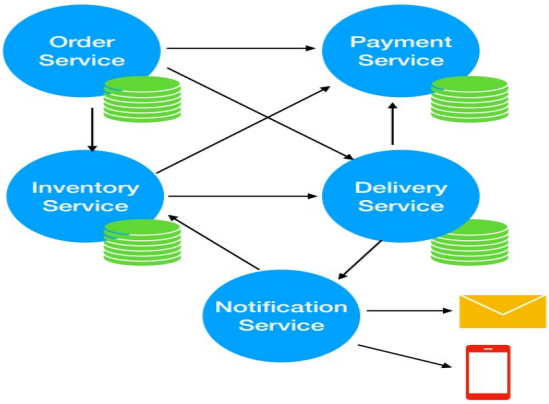
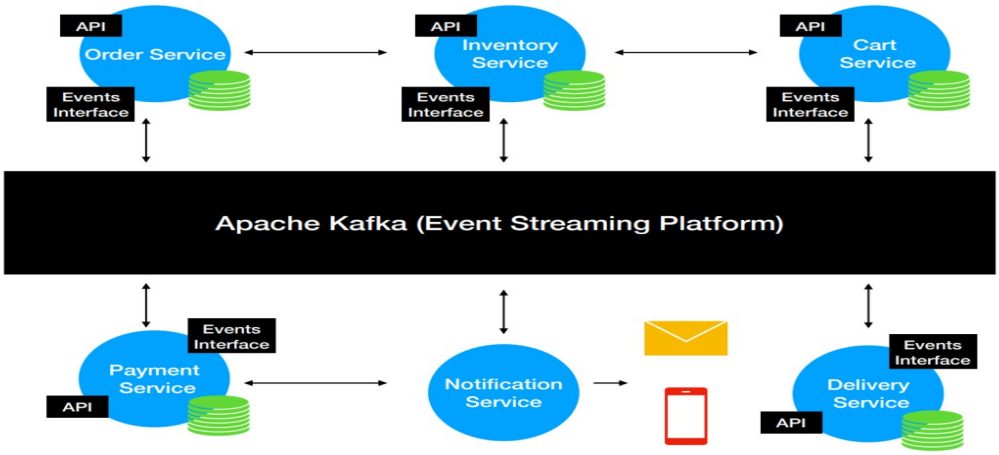
Why Event Streaming in case of Microservices



In monolith architecture, all the functionality will reside in one single application and share the same database. **This monolith architecture is proven to fail under heavy load.**



Modern architecture now-a-days is microservice architecture. The application is decomposed into multiple micro services. In order to deliver value, multiple micro services need to interact with each other. **It will look spagetti if each microservice talk to each other.**



Having a middleware (event streaming platform like kakfa) is necessary to communicate instead of each microservices directly talking to each other. In a nutshell, **every microservice will provide API, be a event producer and a event consumer. Producer and Consumer are independent of each other.** The event streaming platform store the stream of events which can replayed if needed. These events are stored in multiple servers for fault-tolerance and availability.

Traditionl Messaging System vs Kakfa

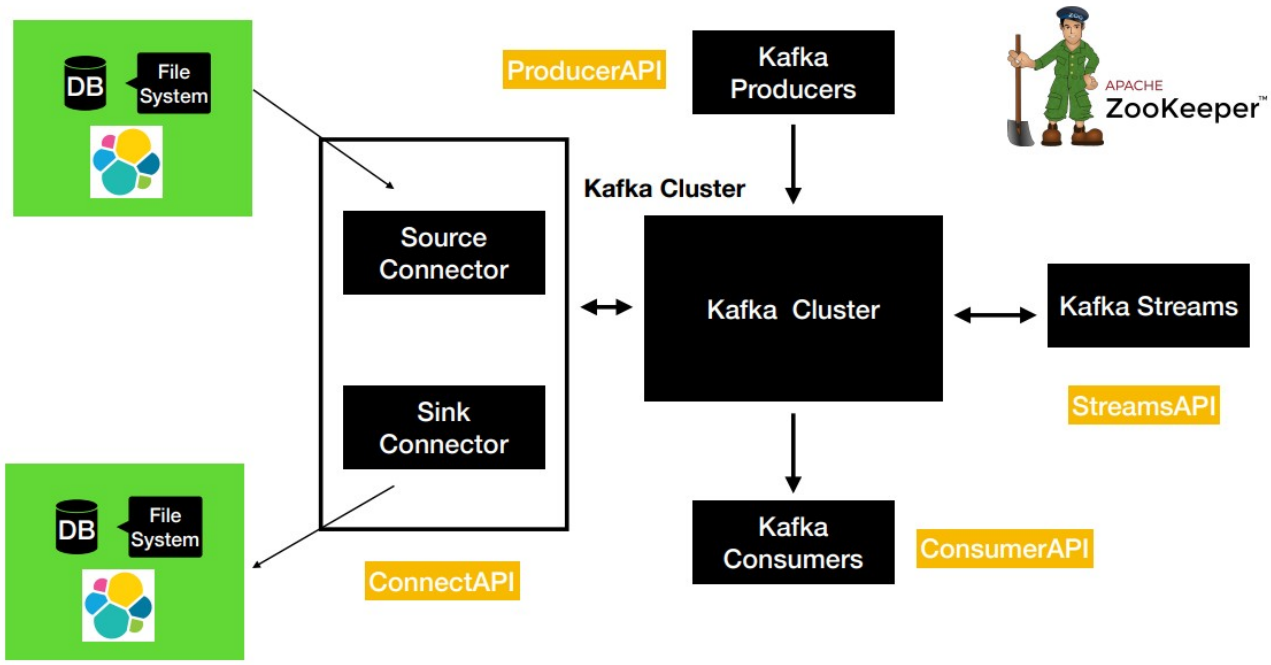
Traditional Messaging System	Kafka
Transient Message Persistence	Store events based on retention time. Events are immutable.
Brokers keep track of consumed message	Consumer's responsibility to keep tract of the consumed messages
Not a distributed system	Distributed stream system

Transient Message Persistence means once the message is consumed, the message is removed from the Message Broker

Kafka is not just a messaging system. It is also a distributed streaming system too.

Kafka Terminology

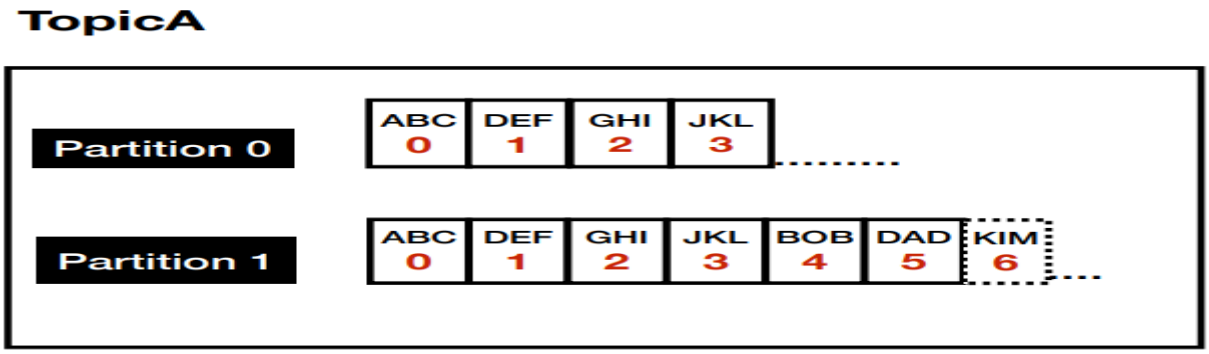
- Kafka Cluster – Multiple Brokers
- Zookeeper – To manage multiple brokers and keeps track the health of the brokers
- Kafka Producers – Uses the client producer API to write/produce new data in kafka
- Kafka Consumers – Consume the message using the Client consumer API
- Kafka Connect – Client API (Source Connector and Sink Connector). Source Connector is used to pull the data from external sources like Database, files etc., into a Kafka Topic. With Kakfa connect, we can do the data movement (in and out of Kafka) without writing a single line of code.
- Kakfa Streams API – Used to take data from Kafka and perform simple to complex transformations on it and put it back to kafka
- 4 Client API Types – Producer API, Consumer API, Kafka Connect, Kafka Streams API using which you can interact with Kakfa



Understanding Kakfa Components

Kakfa Topic: Kafka Topic live inside the broker. Kafka clients use the Topic name to produce and consume messages.

Topic Partitions: Partitions are where the message exactly lives. Each topic can have one or more partitions. Partitions has significant effect on scalable message consumption. Each partition is an ordered, immutable sequence of records. Each record is assigned a sequential number called offset. This offset is generated when the record is placed in a Topic Partition. Each Parition is independent of each other and so ordering is guaranteed only at the Parition level. Partiion continue grow as new records are added and the offsets get incremented one by one. All records in the partition is persisted in a distributed commit log in the file system where kakfa is installed.



Setup of Zookeeper and Kafka Broker

Zookeeper maintains the metadata of the Kafka Broker and also the kafka client information. Anytime the kafka broker starts it registers itself with the Zookeeper. T

he Zookeeper will keep track of the health of the kafka broker. It acts as a centrlaized service for maintaining the configuration information, health of the broker and provide synchronization when we have multiple brokers.

Reference: https://github.com/dilipsundarraj1/kafka-for-developers-using-spring-boot-v2/blob/main/docker-compose.yml

docker-compose.yml

```
version: '2.1'

services:
  zoo1:
    image: confluentinc/cp-zookeeper:7.3.2
    hostname: zoo1
    container_name: zoo1
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_SERVER_ID: 1
      ZOOKEEPER_SERVERS: zoo1:2888:3888


  kafka1:
    image: confluentinc/cp-kafka:7.3.2
    hostname: kafka1
    container_name: kafka1
```

```
  ports:
    - "9092:9092"
    - "29092:29092"
  environment:
    KAFKA_ADVERTISED_LISTENERS: INTERNAL://kafka1:19092,EXTERNAL://${DOCKER_HOST_IP:-127.0.0.1}:9092,DOCKER://host.docker.internal:29092
    KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT,DOCKER:PLAINTEXT
    KAFKA_INTER_BROKER_LISTENER_NAME: INTERNAL
    KAFKA_ZOOKEEPER_CONNECT: "zoo1:2181"
    KAFKA_BROKER_ID: 1
    KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
    KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
  depends_on:
    - zoo1
```

Explanation:

```
depends_on:
  - zoo1
```

This means the kafka broker will wait for the zookeepr to start

$ docker-compose -f docker-compose.yml up

```
PS C:\Users\rames> docker ps
CONTAINER ID   IMAGE                             COMMAND                  CREATED       STATUS       PORTS                                                  NAMES
6036c7ca1c14   confluentinc/cp-kafka:7.3.2       "/etc/confluent/dock…"   2 hours ago   Up 2 hours   0.0.0.0:9092->9092/tcp, 0.0.0.0:29092->29092/tcp   kafka1
8265981cd341   confluentinc/cp-zookeeper:7.3.2   "/etc/confluent/dock…"   2 hours ago   Up 2 hours   2888/tcp, 0.0.0.0:2181->2181/tcp, 3888/tcp         zoo1
```

<u>Produce and Consume messages using CLI</u>

<u>Create the topic</u>

First, login to the docker container that is running kafka.
$ docker exec -it kafka1 bash

kafka1:19092 refers to the KAFKA_ADVERTISED_LISTENERS in the docker-compose.yml file

$ **kafka-topics --bootstrap-server *kafka1:19092*** \
--create \
--topic *test-topic* \
--replication-factor *1* --partitions *1*

```
[appuser@kafka1 ~]$ kafka-topics --bootstrap-server kafka1:19092 \
>               --create \
>               --topic test-topic \
>               --replication-factor 1 --partitions 1
Created topic test-topic.
[appuser@kafka1 ~]$
```

<u>Produce messages to the Topic</u>
We will use the kafka console producer to produce messages to the topic.

**$ docker exec --interactive --tty *kafka1* **
**kafka-console-producer --bootstrap-server *kafka1:19092* **
--topic *test-topic*

```
PS C:\Users\rames> docker exec --interactive --tty kafka1 kafka-console-producer --bootstrap-server kafka1:19092 --topic test-topic
>First Message - Helloworld from Kafka
>Second Message - How are you?
>
```

Consume messages from the Topic

We will use the kafka console consumer to consume messages from the topic

**$ docker exec --interactive --tty *kafka1* **
**kafka-console-consumer --bootstrap-server *kafka1:19092* **
**--topic *test-topic* **
--from-beginning

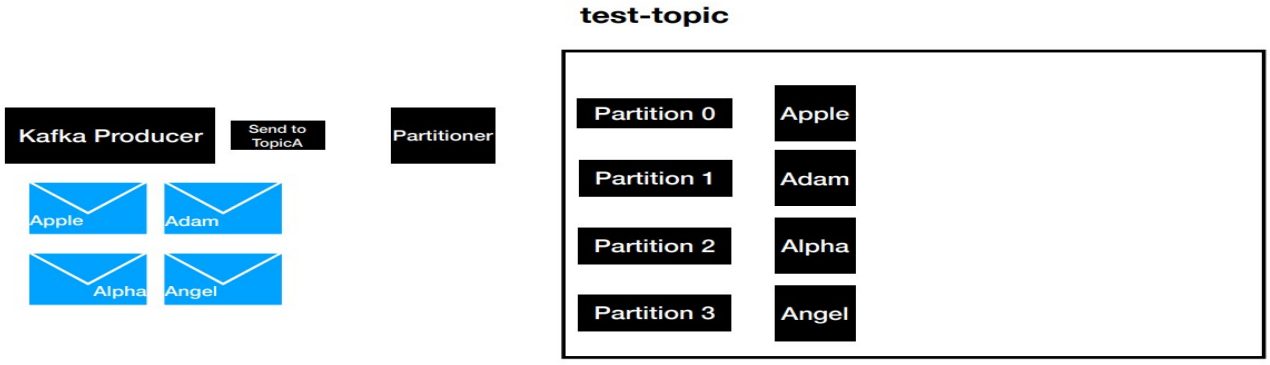
```
PS C:\Users\rames> docker exec --interactive --tty kafka1 kafka-console-consumer --bootstrap-server kafka1:19092 --topic test-topic -from-beginning
First Message - Helloworld from Kafka
Second Message - How are you?
```

Produce and Consume messages with a Key

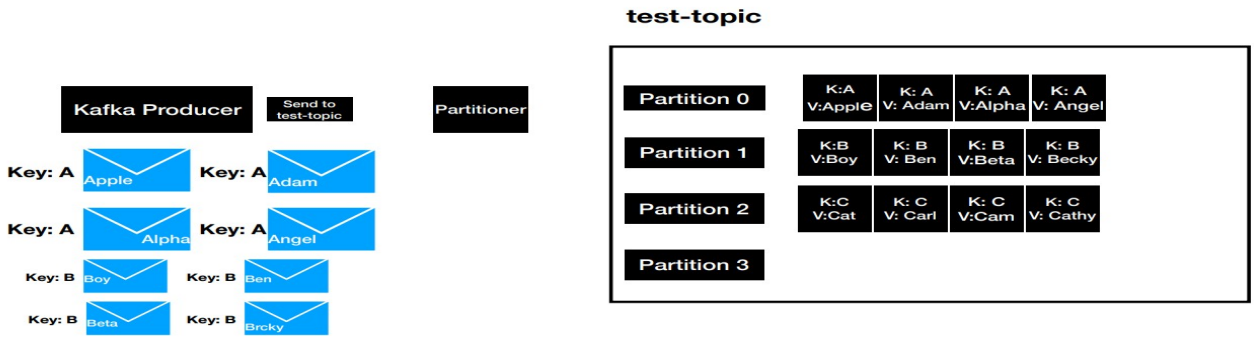
Before the message is sent to the kafka, it passes through multiple layers beneath the Kafka Producer. One such layer is called the Partitioner.

If no key is sent and only the value is sent. Then, the Partitioner will use the round-robin approach to choose the partition to place the messages.
If key is sent along with the value. Then, the Partitioner will do hashing of the key to determine the partition. The same key will always resolve to the same partition.

Message without Key



Message with Key



Console Producer to Create Message with key and value

```
$ docker exec --interactive --tty kafka1  \
kafka-console-producer --bootstrap-server kafka1:19092 \
                --topic test-topic \
                --property "key.separator=-" --property "parse.key=true"
```

Note: we are sending messages like “A-Apple”, “B-Bob” because we are using the property key.separator as “-”

```
PS C:\Users\rames> docker exec --interactive --tty kafka1 kafka-console-producer --bootstrap-server kafka1:19092 --topic test-topic --property "key.separator=-" --property "parse.key=true"
>A-Apple
>B-Bob
>C-Cat
>NoKey
org.apache.kafka.common.KafkaException: No key separator found on line number 4: 'NoKey'
        at kafka.tools.ConsoleProducer$LineMessageReader.parse(ConsoleProducer.scala:374)
        at kafka.tools.ConsoleProducer$LineMessageReader.readMessage(ConsoleProducer.scala:349)
        at kafka.tools.ConsoleProducer$.main(ConsoleProducer.scala:50)
        at kafka.tools.ConsoleProducer.main(ConsoleProducer.scala)
PS C:\Users\rames>
```

Console Consumer to Read Message with key and value

```
$ docker exec --interactive --tty kafka1  \
kafka-console-consumer --bootstrap-server kafka1:19092 \
                --topic test-topic \
                --property "key.separator=-" --property "print.key=true"
```

```
PS C:\Users\rames> docker exec --interactive --tty kafka1 kafka-console-consumer --bootstrap-server kafka1:19092 --topic test-topic --property "key.separator= - " --property "print.key=true"
A - Apple
B - Bob
C - Cat
```

Consumer Offset

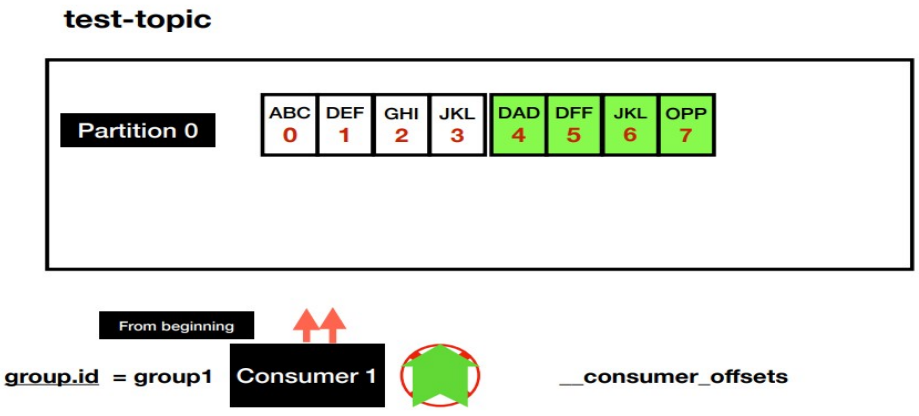
Any message that is produced into the topic will have an unique id called offset.

Consumer has three positions to read

- from-beginning
- latest – Read messages from the instance the consumer is spun-up
- from a specific offset – This can be done only programmatically

Every consumer is required to provide a group id

How does the consumer knows from which offset to start (let’s say for example the consumer has crashed and restarted. Also, the producer has created more messages meanwhile)



The consumer offsets is stored in an internal topic called __consumer_offsets. Consumer offsets

behaves like a bookmark for the consumer to start reading the messages from the point it left of.

Get the List of topics

```
docker exec --interactive --tty kafka1  \
        kafka-topics --bootstrap-server kafka1:19092 --list
```

```
PS C:\Users\rames> docker exec --interactive --tty kafka1 kafka-topics --bootstrap-server kafka1:19092 --list
__consumer_offsets
test-topic
PS C:\Users\rames>
```

__consumer_offsets topic is local and internal to Kakfa

Consumer Groups

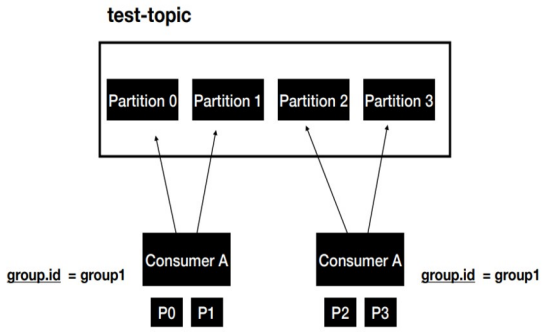
Group id plays a major role in scalabe message consumption.

If we have consumer with group id 1 and have 4 partitions. The consumer is single threaded and will pool all the four parititons. If the producer is producing message at a higher rate. Then, it will create a lag at the consumption side. So, the messages might not be processed realtime.

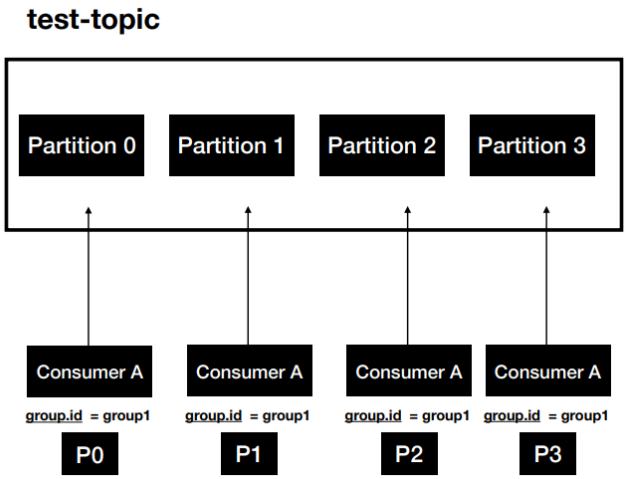
If more consumers with same group id is started. Now, the partitions are split between the two consumers. This way we can scale our message consumption.

As we add more consumers with the same group id. Then, the consumption of the messages by consumer is split between the partitions.

2 consumers in a group and 4 partitions

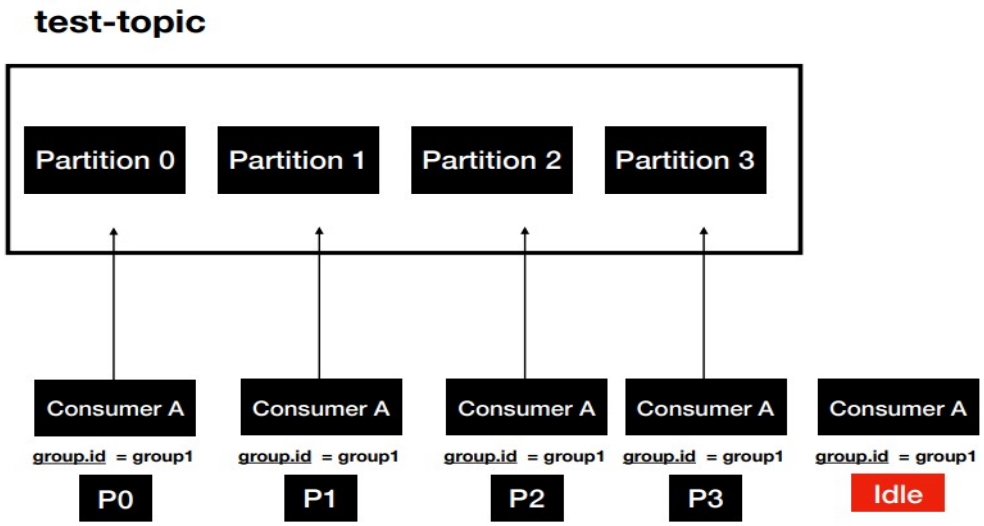


4 consumers in a group and 4 partitions



5 consumers in a group and 4 partitions

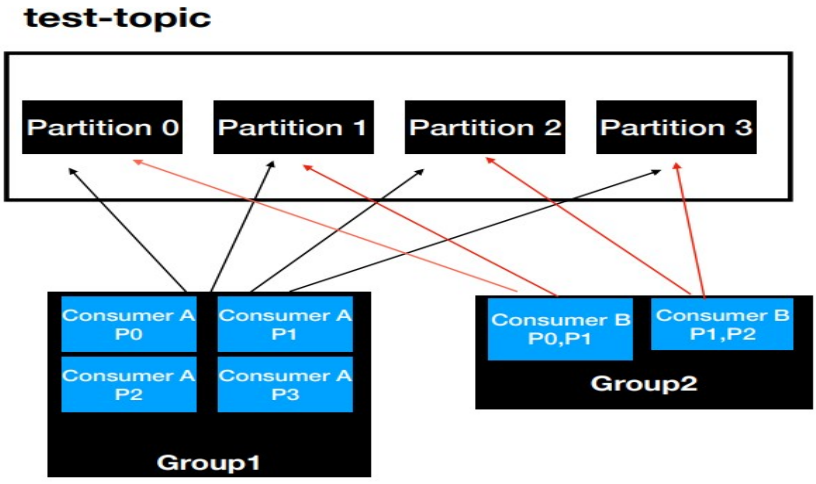
Note: One of the Consumer in the group will remain idle



Each Consumer Group is like an application Concept
Two different application for the same topic.
Here, every/same message will be read by two different consumers of different consumer groups.

Each application will have their own processing logic and hence each application that consume will have their own group id. Also, each application has different number of instances based on the requirement.
Here, the application (consuming group id of group1) has four consumers to read the messages from the topic whereas a different applicaiton (consuming group id of group2) has two consumers to read the messages from the same topic. It is upto the individual application team to decide how many consumers they want.

You have make sure each applicaiton uses a unique group id for themselves



Summary

- **Consumer Groups are used for scalable message consumption**
- **Each different application will have a unique consumer group**

Hands-on Consumer Groups

View Consumer Groups
docker exec --interactive --tty kafka1 \
kafka-consumer-groups --bootstrap-server kafka1:19092 –list

```
PS C:\Users\rames> docker exec --interactive --tty kafka1 kafka-consumer-groups --bootstrap-server kafka1:19092 --list
console-consumer-96243
PS C:\Users\rames>
```

Describe All topics
docker exec --interactive --tty kafka1 \
kafka-topics --bootstrap-server kafka1:19092 –describe

```
PS C:\Users\rames> docker exec --interactive --tty kafka1 kafka-topics --bootstrap-server kafka1:19092 --describe
Topic: test-topic       TopicId: ed0IUqzaRQOn-mSJPjUK7Q PartitionCount: 1       ReplicationFactor: 1    Configs:
        Topic: test-topic       Partition: 0    Leader: 1       Replicas: 1     Isr: 1
PS C:\Users\rames>
```

Describe a specific Topic (test-topic here)
docker exec --interactive --tty kafka1 \
kafka-topics --bootstrap-server kafka1:19092 --describe \
--topic test-topic

```
PS C:\Users\rames> docker exec --interactive --tty kafka1 kafka-topics --bootstrap-server kafka1:19092 --describe --topic test-topic
Topic: test-topic       TopicId: ed0IUqzaRQOn-mSJPjUK7Q PartitionCount: 1       ReplicationFactor: 1    Configs:
        Topic: test-topic       Partition: 0    Leader: 1       Replicas: 1     Isr: 1
PS C:\Users\rames>
```

Alter the number of partitions to 4 in order to test Consumer Topics
docker exec --interactive --tty kafka1 \
kafka-topics --bootstrap-server kafka1:19092 \
--alter --topic test-topic --partitions 4

```
PS C:\Users\rames> docker exec --interactive --tty kafka1 kafka-topics --bootstrap-server kafka1:19092 --alter --topic test-topic --partitions 4
PS C:\Users\rames> docker exec --interactive --tty kafka1 kafka-topics --bootstrap-server kafka1:19092 --describe --topic test-topic
Topic: test-topic       TopicId: ed0IUqzaRQOn-mSJPjUK7Q PartitionCount: 4       ReplicationFactor: 1    Configs:
        Topic: test-topic       Partition: 0    Leader: 1       Replicas: 1     Isr: 1
        Topic: test-topic       Partition: 1    Leader: 1       Replicas: 1     Isr: 1
        Topic: test-topic       Partition: 2    Leader: 1       Replicas: 1     Isr: 1
        Topic: test-topic       Partition: 3    Leader: 1       Replicas: 1     Isr: 1
PS C:\Users\rames>
```

Start the Producer

docker exec --interactive --tty kafka1 \
kafka-console-producer --bootstrap-server kafka1:19092 \
--topic test-topic \
--property "key.separator=-" --property "parse.key=true"

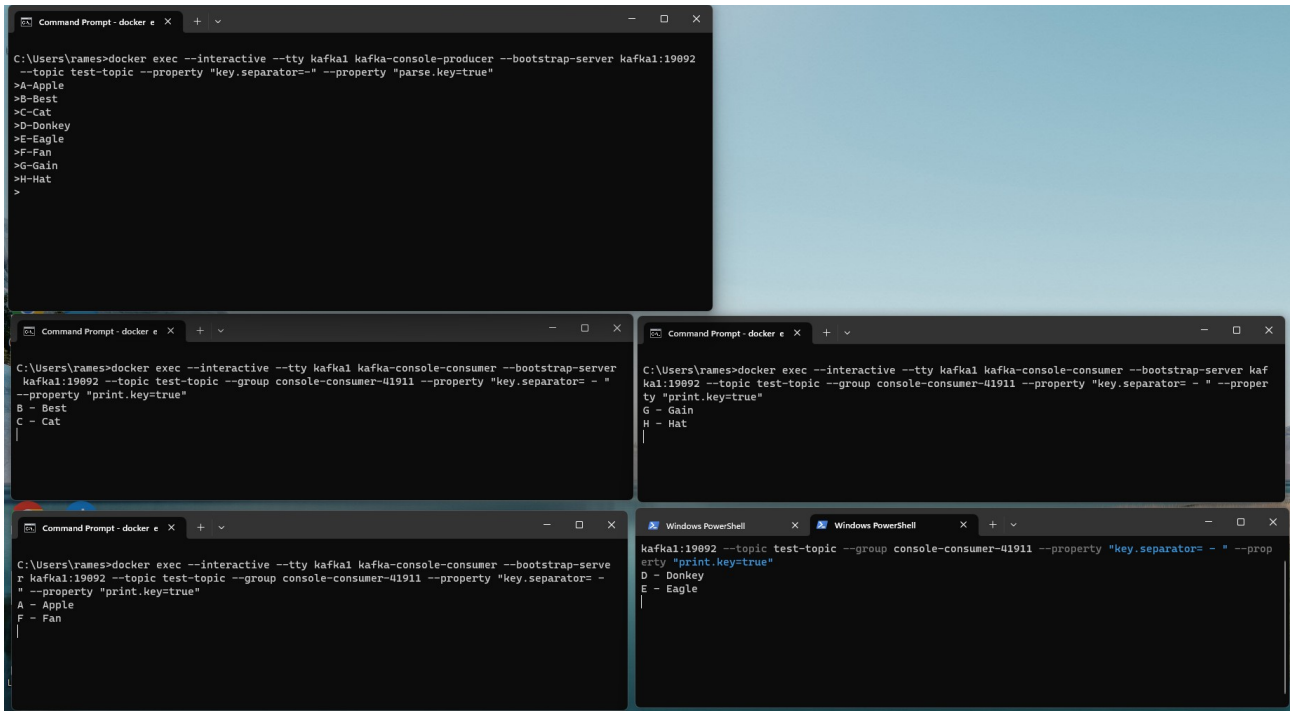
Start one consumer
docker exec --interactive --tty kafka1 \
kafka-console-consumer --bootstrap-server kafka1:19092 \
--topic test-topic --group console-consumer-41911\
--property "key.separator= - " --property "print.key=true"

Get the consumer group of the started consumer

docker exec --interactive --tty kafka1 kafka-consumer-groups --bootstrap-server kafka1:19092 –list

```
C:\Users\rames>docker exec --interactive --tty kafka1 kafka-consumer-groups --bootstrap-server kafka1:19092 --list
console-consumer-41911
```

Start the Other three consumers using the same consumer group
docker exec --interactive --tty kafka1 \
kafka-console-consumer --bootstrap-server kafka1:19092 \
--topic test-topic --group console-consumer-41911\
--property "key.separator= - " --property "print.key=true"



Each Consumer receiving the messages from its corresponding partition

Let’s start the 5th Consumer in the same consumer group (**Number of Consumers > Number of Paritions**)
You will notice one of the consumer will remain idle

Commit Log and Retention Policy

Commit Log

Whenever a message is produced to a topic, It is written to a file with extension of .log in the filesystem called commit log. Each partition will have its own log file. The filesystem location for the commit log is configured in server.properties in a property called **log.dirs**.

The message is written to the filesystem as bytes. It is when the message is written to the file, the message produced by the producer is considered committed. Consumers can see only the records that are committed.

Retention Policy

Kakfa retains the messages for a prefined period of time called retention policy. It determines how long the message is going to be retained. Configured using the property **log.retention.hours** in server.properties
The default retention period is 168 hours (7 days)

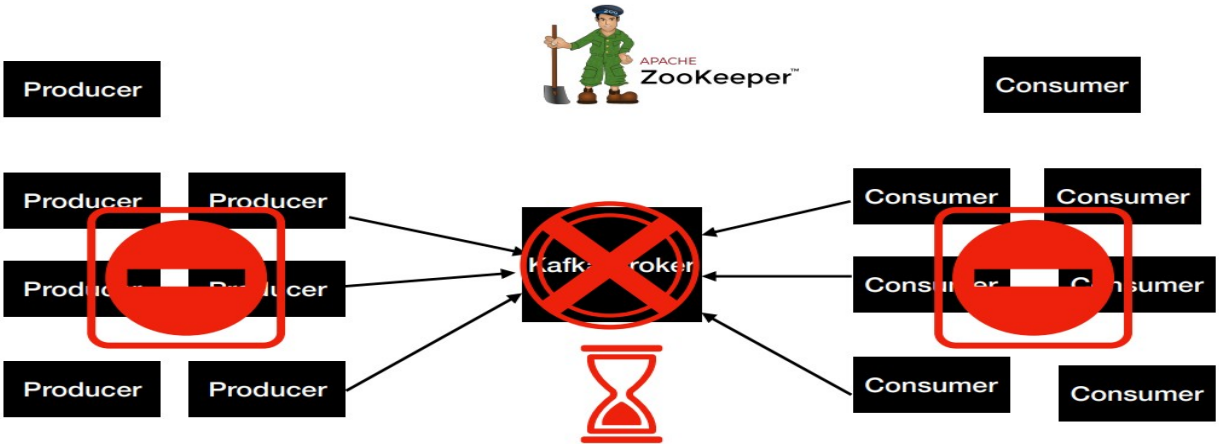
Kakfa – As a Distribute System

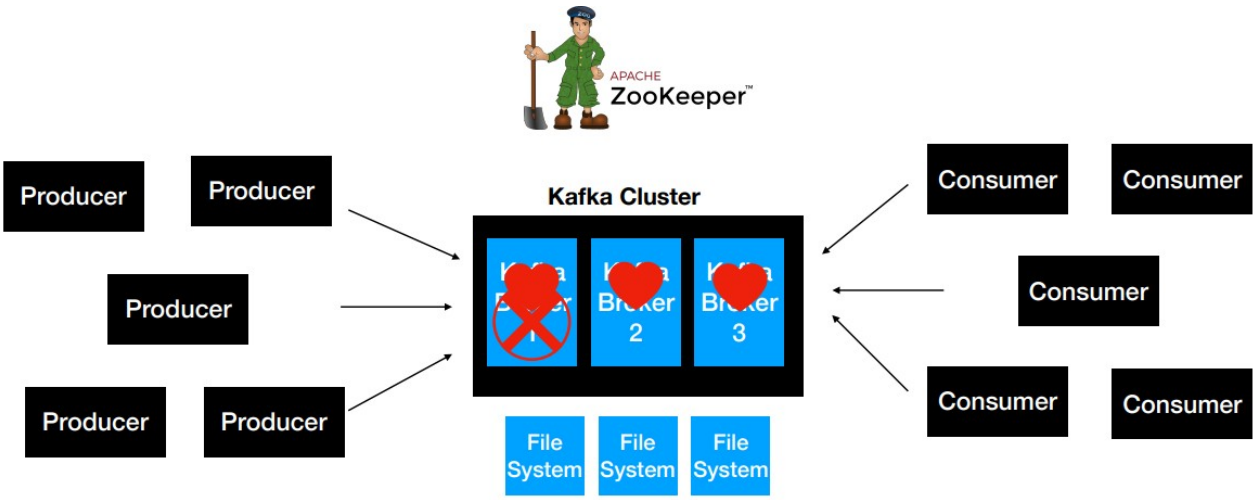
Kafka is a distributed streaming platform. Distributed systems are a collection of systems working together to deliver a value

Characteristics of Distributed System

- Availability and Fault Tolerance – Even if one of the system is down, Still it won't impact the overall availability of the system.
- Reliable Work distribution – The work load is distributed and shared by the different systems.
- Easily scalable – Adding another system to the existing setup can be done easily

Single point of failure





With a Kafka Cluster. Kafka Cluster is nothing but 1 or more brokers. It is common to have more than 1 broker.

- Client requests are distributed between brokers
- Easy to scale by adding more brokers based on the need
- Handles data loss using Replication
- If one of the broker goes down. Then, cluster manager (namely the Zookeeper) is notified and all the client requests are routed to the available brokers.

Kafka Cluster Setup

Reference: https://github.com/dilipsundarraj1/kafka-for-developers-using-spring-boot-v2/blob/main/docker-compose-multi-broker.yml

Cluster with three brokers

```
version: '2.1'

services:
  zoo1:
    image: confluentinc/cp-zookeeper:7.3.2
    platform: linux/amd64
    hostname: zoo1
    container_name: zoo1
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_SERVER_ID: 1
      ZOOKEEPER_SERVERS: zoo1:2888:3888


  kafka1:
    image: confluentinc/cp-kafka:7.3.2
    platform: linux/amd64
    hostname: kafka1
    container_name: kafka1
    ports:
      - "9092:9092"
      - "29092:29092"
    environment:
      KAFKA_ADVERTISED_LISTENERS: INTERNAL://kafka1:19092,EXTERNAL://${DOCKER_HOST_IP:-127.0.0.1}:9092,DOCKER://host.docker.internal:29092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT,DOCKER:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: INTERNAL
      KAFKA_ZOOKEEPER_CONNECT: "zoo1:2181"
      KAFKA_BROKER_ID: 1
      KAFKA_LOG4J_LOGGERS: "kafka.controller=INFO,kafka.producer.async.DefaultEventHandler=INFO,state.change.logger=INFO"
    depends_on:
      - zoo1

  kafka2:
    image: confluentinc/cp-kafka:7.3.2
    platform: linux/amd64
    hostname: kafka2
    container_name: kafka2
    ports:
      - "9093:9093"
      - "29093:29093"
    environment:
      KAFKA_ADVERTISED_LISTENERS: INTERNAL://kafka2:19093,EXTERNAL://$
```

```
{DOCKER_HOST_IP:-127.0.0.1}:9093,DOCKER://host.docker.internal:29093
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT,DOCKER:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: INTERNAL
      KAFKA_ZOOKEEPER_CONNECT: "zoo1:2181"
      KAFKA_BROKER_ID: 2
      KAFKA_LOG4J_LOGGERS:
"kafka.controller=INFO,kafka.producer.async.DefaultEventHandler=INFO,state.change.logger=I
NFO"
    depends_on:
      - zoo1


  kafka3:
    image: confluentinc/cp-kafka:7.3.2
    platform: linux/amd64
    hostname: kafka3
    container_name: kafka3
    ports:
      - "9094:9094"
      - "29094:29094"
    environment:
      KAFKA_ADVERTISED_LISTENERS: INTERNAL://kafka3:19094,EXTERNAL://$
{DOCKER_HOST_IP:-127.0.0.1}:9094,DOCKER://host.docker.internal:29094
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT,DOCKER:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: INTERNAL
      KAFKA_ZOOKEEPER_CONNECT: "zoo1:2181"
      KAFKA_BROKER_ID: 3
      KAFKA_LOG4J_LOGGERS:
"kafka.controller=INFO,kafka.producer.async.DefaultEventHandler=INFO,state.change.logger=I
NFO"
    depends_on:
- zoo1
```

$ docker-compose -f docker-compose-multi-broker.yml up

```
PS C:\Users\rames> docker ps
CONTAINER ID   IMAGE                             COMMAND                  CREATED          STATUS          PORTS                                                  NAMES
f79eb0a59a6c   confluentinc/cp-kafka:7.3.2       "/etc/confluent/dock…"   20 seconds ago   Up 18 seconds   0.0.0.0:9092->9092/tcp, 0.0.0.0:29092->29092/tcp       kafka1
e81e667fa083   confluentinc/cp-kafka:7.3.2       "/etc/confluent/dock…"   20 seconds ago   Up 18 seconds   0.0.0.0:9093->9093/tcp, 9092/tcp, 0.0.0.0:29093->29093/tcp   kafka2
4a3ac6c94d6e   confluentinc/cp-kafka:7.3.2       "/etc/confluent/dock…"   20 seconds ago   Up 18 seconds   0.0.0.0:9094->9094/tcp, 9092/tcp, 0.0.0.0:29094->29094/tcp   kafka3
33e26070c3a0   confluentinc/cp-zookeeper:7.3.2   "/etc/confluent/dock…"   20 seconds ago   Up 19 seconds   2888/tcp, 0.0.0.0:2181->2181/tcp, 3888/tcp             zoo1
```

Create a topic with replication factor of 3

```
docker exec --interactive --tty kafka1  \
kafka-topics --bootstrap-server kafka1:19092 \
        --create \
        --topic test-topic \
        --replication-factor 3 --partitions 3
```

Note: Though, we connect to one broker to create the topic. Still , the topic gets created in all the brokers.

```
PS C:\Users\rames> docker exec --interactive --tty kafka1 kafka-topics --bootstrap-server kafka1:19092 --create --topic test-topic --replication-factor 3 --partitions 3
Created topic test-topic.
PS C:\Users\rames>
```

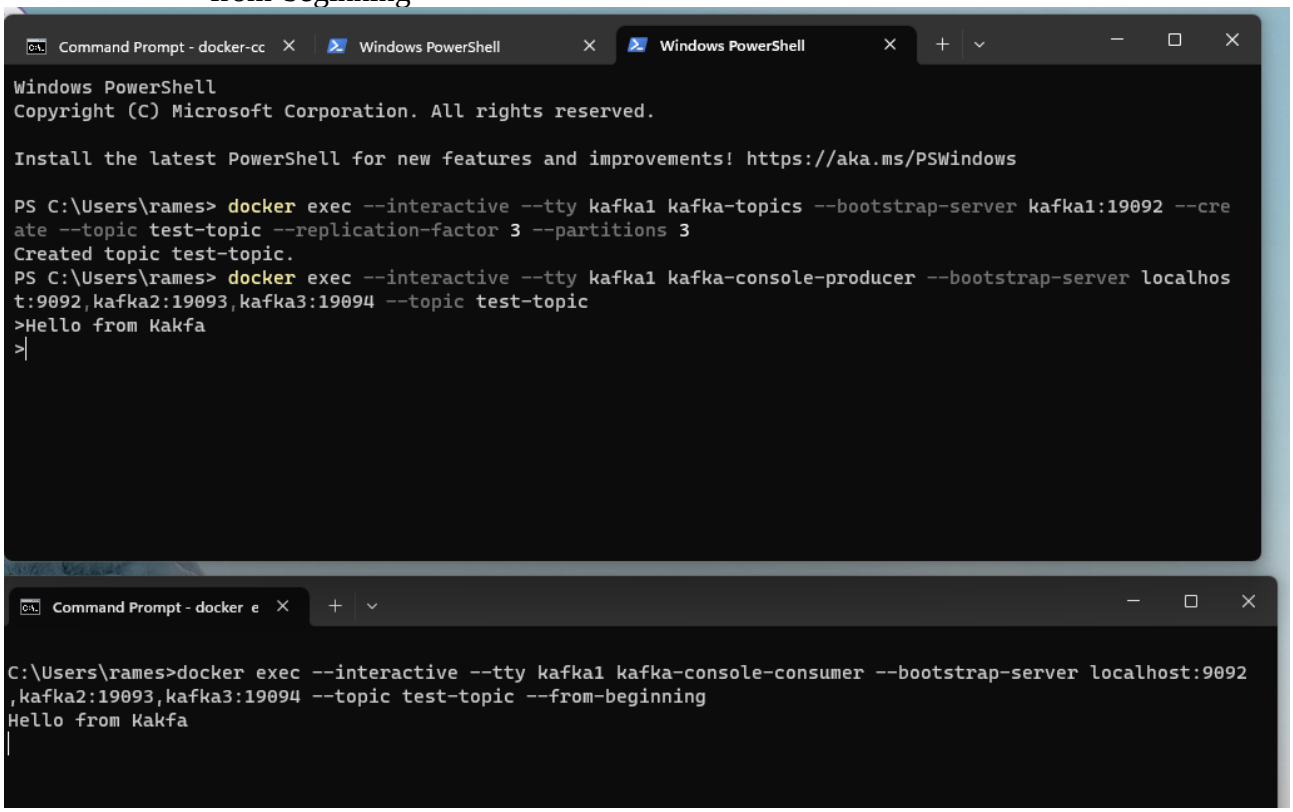
Produce Messages to the Topic

docker exec --interactive --tty **kafka1** \
kafka-console-producer **--bootstrap-server localhost**:9092,kafka2:19093,kafka3:19094 \
--topic test-topic

Note: We give all the broker address in the bootstrap-server when producting the topic.
Since, we login to kafka1 to issue the command, we use localhost for kafka1

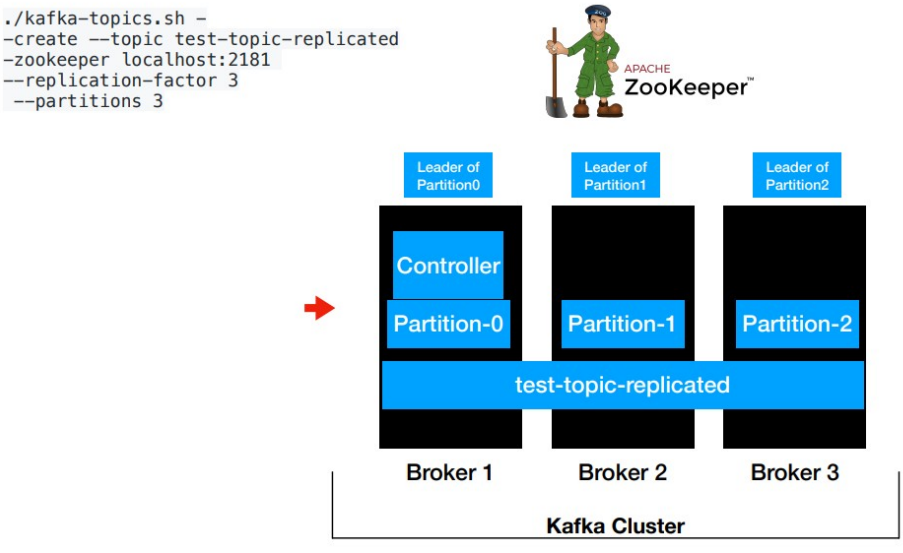
Consume Messages from the Topic

docker exec --interactive --tty kafka1 \
kafka-console-consumer --bootstrap-server localhost:9092,kafka2:19093,kafka3:19094 \
--topic test-topic \
--from-beginning



Distribution of Client Requests – Leader and Follower

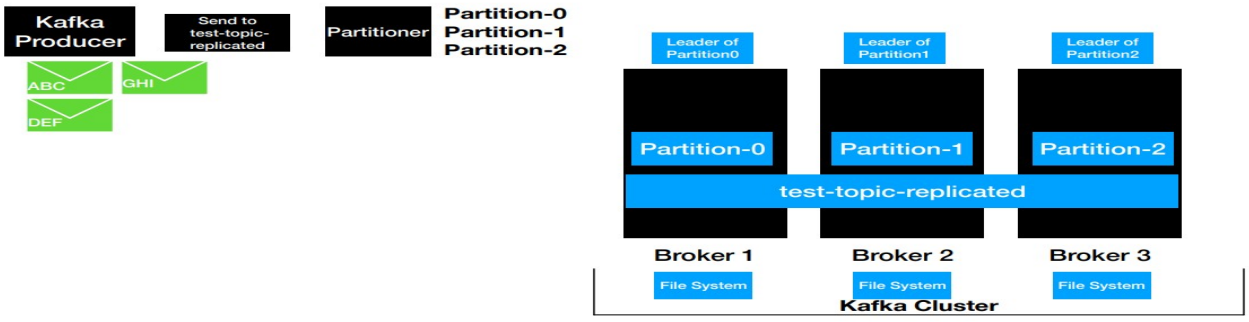
How Kafka cluster distributes the client requests between the brokers.
When the broker starts up and connect to the zookeeper, one of the broker acts as a controller.



When the create topic command is issued to the cluster, the zookeeper takes care of re-directing the request to the controller. The role of the controller is to distribute the partitions to the available broker. This process is called leader assignment.

How Producer distributes the client requests?

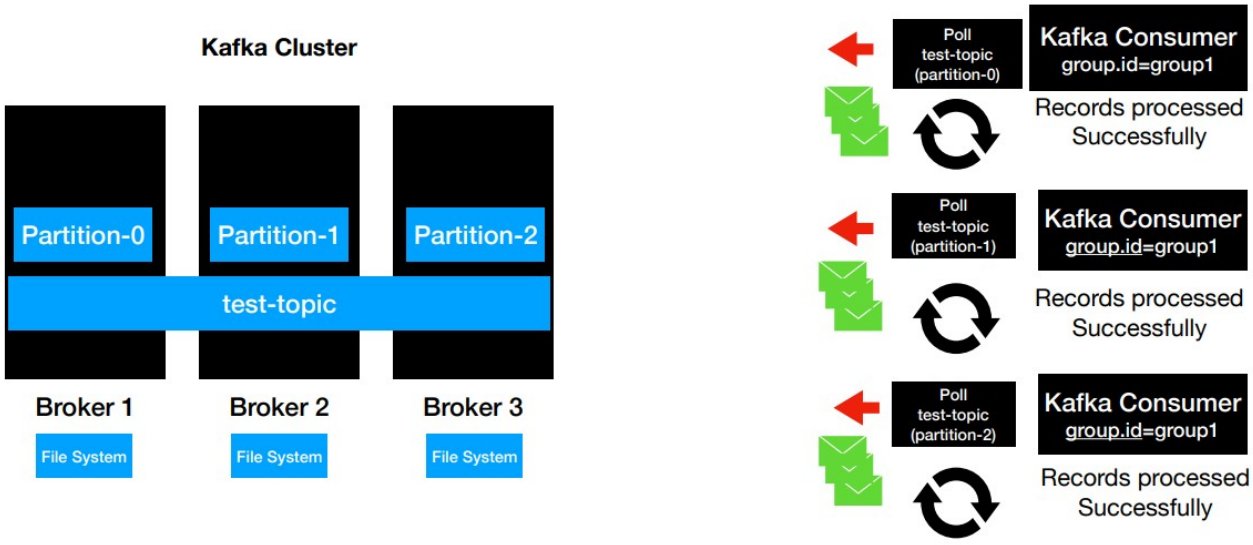
The partitioner finds out which partition the message should be delivered. For example, the partitioner decides to send the message to Partiton-0. In this case, the leader of Parition-0 is Broker1. Therefore, the message is sent to Broker1. **The client will always invoke the leader of the partition.** The client requests from the producer end is distributed between the brokers based on the partition which means indirectly it is distributed among the brokers.



How Kakfa distributes the client request – Kakfa Consumer

Requests go to all the paritions/brokers to retrieve the records to process.

With consumer groups, it is a common practise to run multiple instances of a consumer in a consumer group and process the records in parallel. If one ore more consumers are started with the same groupid. Then, the partitions are distributed for scalable consumption. Each consumer gets one partition assigned. Also, the call goes to the broker which leader of the respective partition.

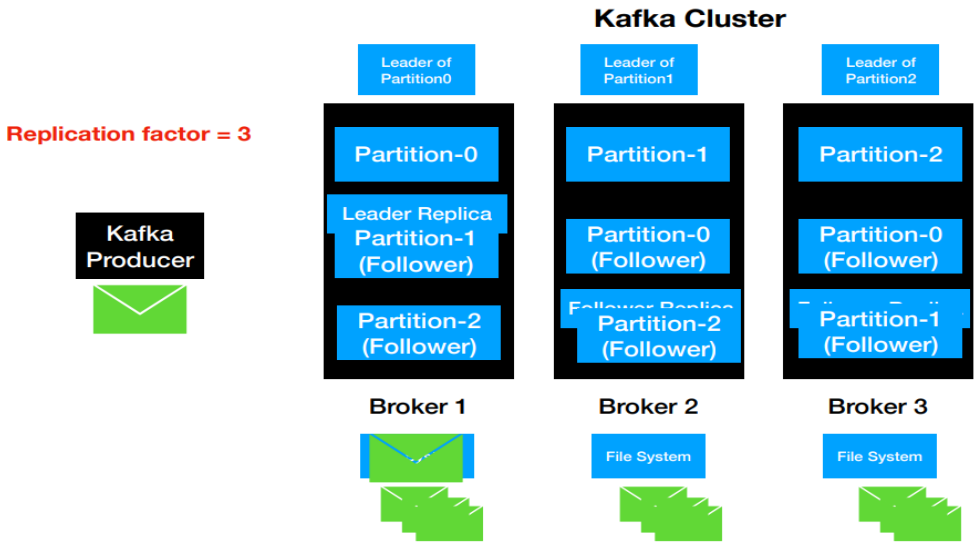


How Kafka handles data loss?

Kafka handles data loss via replication. The producer and consumer will always talk to the leader o the partition.

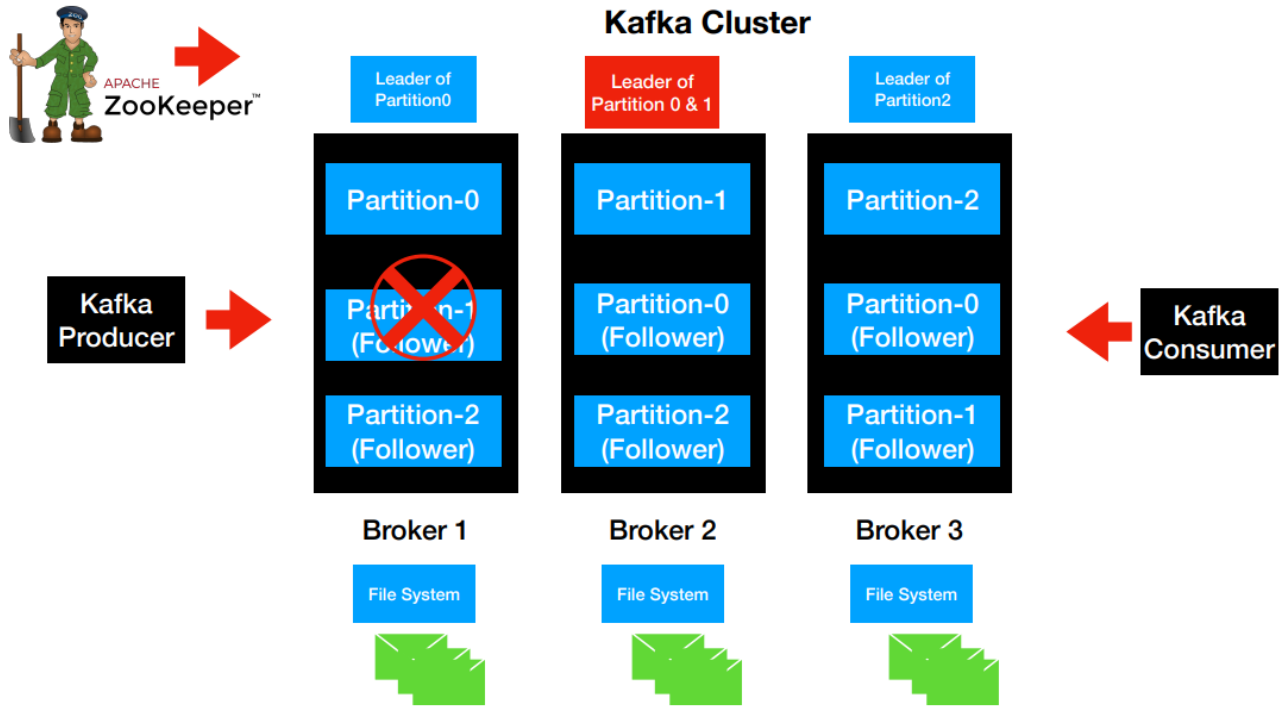
When the data is producer it will written to the leader of the partition and ultimately to the file system of the leader of the partition.

Replication factor represents the number of copies of the message. With replication-factor of 3, two more brokers will be identifier as follower replicas. The message is written to the leader replica and the same message is written to the follower replicas. **In total for replicaiton-factor of 3, 3 copies of the message will be written.** Note: The replication factor should be greater than 1.



What happens when one of the broker fails.

We have leader replica for each partition and follower replicas for them. Let's assume the Broker 1 is down. Still, the data of the partition-0 is available in its follower replicas namely Broker 2 and Broker 3. Zookeeper is informed about the failure of the broker and Zookeeper assigns a new leader. Now, Broker 2 becomes the leader of both partition 0 and partition 1.



In-Sync Replicas (ISR)

Represents the number of replicas which is in sync with each other in the cluster. The number includes both the leader and the follower replica.

Ideal value is **ISR == replication factor**

The value for ISR can be configured using the property **min.insync.replicas.** Can be set at the broker level or at the topic level.

Describe the topic

docker exec --interactive --tty kafka1 \

kafka-topics --bootstrap-server kafka1:19092 --describe \

--topic test-topic

```
PS C:\Users\rames> docker exec --interactive --tty kafka1 kafka-topics --bootstrap-server kafka1:19092 --describe --topic test-topic
Topic: test-topic       TopicId: nS4bpnfxR-KRWFE73gm02A PartitionCount: 3       ReplicationFactor: 3    Configs:
        Topic: test-topic       Partition: 0    Leader: 2       Replicas: 2,3,1 Isr: 2,3,1
        Topic: test-topic       Partition: 1    Leader: 3       Replicas: 3,1,2 Isr: 3,1,2
        Topic: test-topic       Partition: 2    Leader: 1       Replicas: 1,2,3 Isr: 1,2,3
PS C:\Users\rames>
```

Here, you can see each parition which is the leader/follower and the Replica details.

Topic: test-topic TopicId: nS4bpnfxR-KRWFE73gm02A PartitionCount: 3 **ReplicationFactor: 3** Configs:

Topic: test-topic **Partition: 0 Leader: 2 Replicas: 2,3,1 Isr: 2,3,1**

Topic: test-topic **Partition: 1 Leader: 3 Replicas: 3,1,2 Isr: 3,1,2**

Topic: test-topic **Partition: 2 Leader: 1 Replicas: 1,2,3 Isr: 1,2,3**

If there are say 40 partitions and 3 brokers. Then, each broker will be leader replica for many partitions.

Isr: 2,3,1 – Means three replicas are in sync.

Let’s remove one of the broker from the cluster
$ docker stop <containerid_of_kafka3>

```
PS C:\Users\rames> docker ps
CONTAINER ID   IMAGE                             COMMAND                  CREATED       STATUS       PORTS                                                  NAMES
f79eb0a59a6c   confluentinc/cp-kafka:7.3.2       "/etc/confluent/dock…"   3 hours ago   Up 3 hours   0.0.0.0:9092->9092/tcp, 0.0.0.0:29092->29092/tcp       kafka1
e81e667fa083   confluentinc/cp-kafka:7.3.2       "/etc/confluent/dock…"   3 hours ago   Up 3 hours   0.0.0.0:9093->9093/tcp, 9092/tcp, 0.0.0.0:29093->29093/tcp   kafka2
4a3ac6c94d6e   confluentinc/cp-kafka:7.3.2       "/etc/confluent/dock…"   3 hours ago   Up 3 hours   0.0.0.0:9094->9094/tcp, 9092/tcp, 0.0.0.0:29094->29094/tcp   kafka3
33e26070c3a0   confluentinc/cp-zookeeper:7.3.2   "/etc/confluent/dock…"   3 hours ago   Up 3 hours   2888/tcp, 0.0.0.0:2181->2181/tcp, 3888/tcp             zoo1
```

```
PS C:\Users\rames> docker stop 33e26070c3a0
33e26070c3a0
PS C:\Users\rames>
```

Now, let’s describe of the topic again now

```
PS C:\Users\rames> docker exec --interactive --tty kafka1 kafka-topics --bootstrap-server kafka1:19092 --describe --topic test-topic
Topic: test-topic       TopicId: D05Am97mRO2QGBub-WZQRw PartitionCount: 3       ReplicationFactor: 3    Configs:
        Topic: test-topic       Partition: 0    Leader: 2       Replicas: 3,2,1 Isr: 2,1
        Topic: test-topic       Partition: 1    Leader: 1       Replicas: 1,3,2 Isr: 1,2
        Topic: test-topic       Partition: 2    Leader: 2       Replicas: 2,1,3 Isr: 2,1
PS C:\Users\rames>
```

PS C:\Users\rames> docker exec --interactive --tty kafka1 kafka-topics --bootstrap-server kafka1:19092 --describe --topic test-topic
Topic: test-topic TopicId: D05Am97mRO2QGBub-WZQRw PartitionCount: 3 ReplicationFactor: 3 Configs:
Topic: test-topic Partition: 0 Leader: 2 Replicas: 3,2,1 Isr: 2,1
Topic: test-topic Partition: 1 Leader: 1 Replicas: 1,3,2 Isr: 1,2
Topic: test-topic Partition: 2 Leader: 2 Replicas: 2,1,3 Isr: 2,1

You will ntoice the Isr: 2,1 for partition-0 and all partitions has only 2 in-sync replicas

Note: Like the replication-factor, it is advisable to **keep the min.insync.replicas value to be greater than 1** so that there is atleast 1 backup of the messages.

Configuring min.insync.replicas

Setting min.insync.replicas at Topic Level
docker exec --interactive --tty kafka1 \
kafka-configs --bootstrap-server localhost:9092 --entity-type topics --entity-name test-topic \
--alter --add-config min.insync.replicas=2

```
PS C:\Users\rames> docker exec --interactive --tty kafka1 kafka-configs  --bootstrap-server localhost:9092 --entity-type topics --entity-name test-topic --alter --add-config min.insync.replicas=2
[2023-09-09 10:51:47,885] WARN [AdminClient clientId=adminclient-1] Connection to node 3 (/127.0.0.1:9094) could not be established. Broker may not be available. (org.apache.kafka.clients.NetworkClient)
[2023-09-09 10:51:47,889] WARN [AdminClient clientId=adminclient-1] Connection to node 2 (/127.0.0.1:9093) could not be established. Broker may not be available. (org.apache.kafka.clients.NetworkClient)
Completed updating config for topic test-topic.
PS C:\Users\rames>
```

Let’s instatiate the producer
docker exec --interactive --tty kafka1 \
kafka-console-producer --bootstrap-server localhost:9092,kafka2:19093,kafka3:19094 \
--topic test-topic

Let’s instatiate a consumer
docker exec --interactive --tty kafka1 \
kafka-console-consumer --bootstrap-server localhost:9092,kafka2:19093,kafka3:19094 \
--topic test-topic \
--from-beginning

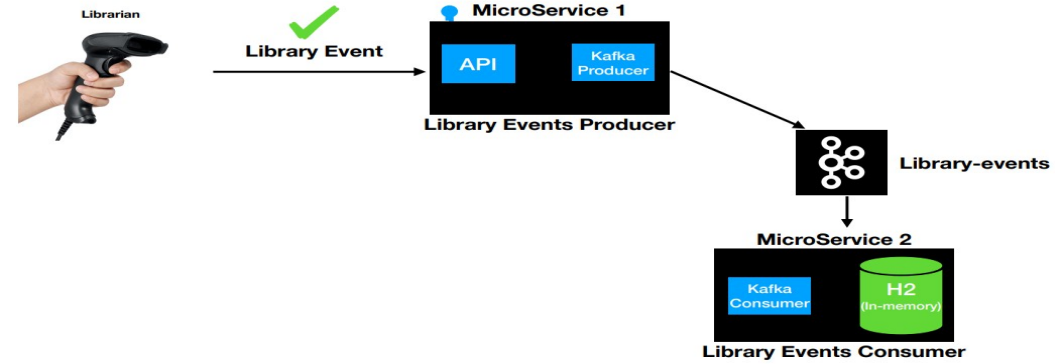
Message is flowing as expected.

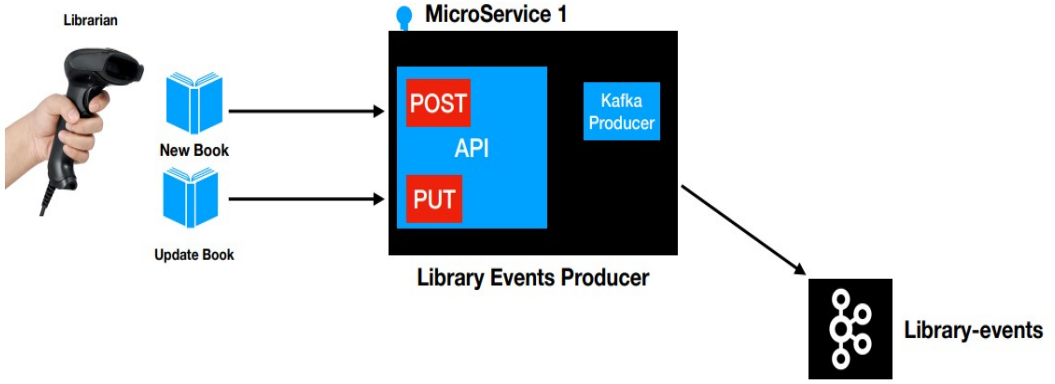
Now, Let's bring two out of the threee brokers down. The setting for min.insync.replicas = 2 and the number of brokers = 1
Now if you publish the message, you will get error saying NOTENOUGH_REPLICAS

If two out of the three brokers are up. Then, the requirements will be met and will be able publish successfully.

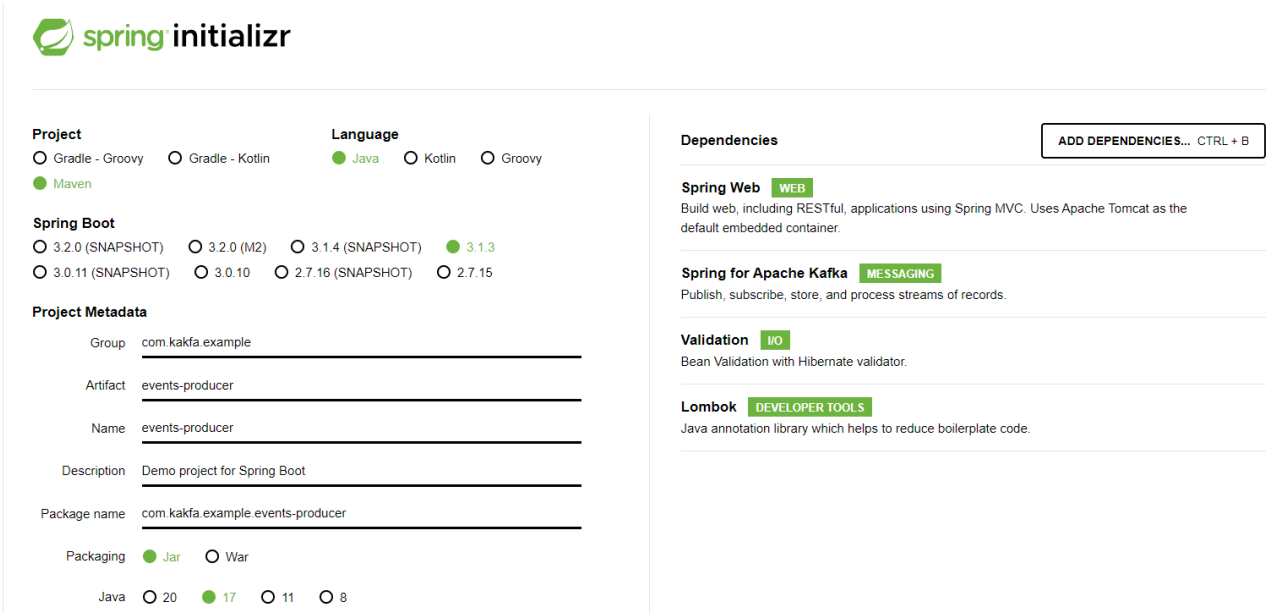
The min.insync.replicas will ensure that the required/stated number of copies of the data is available in more than 1 kafka broker.

Overview of the Application – Library inventory





Build SpringBoot Kafka Producer – Hands On

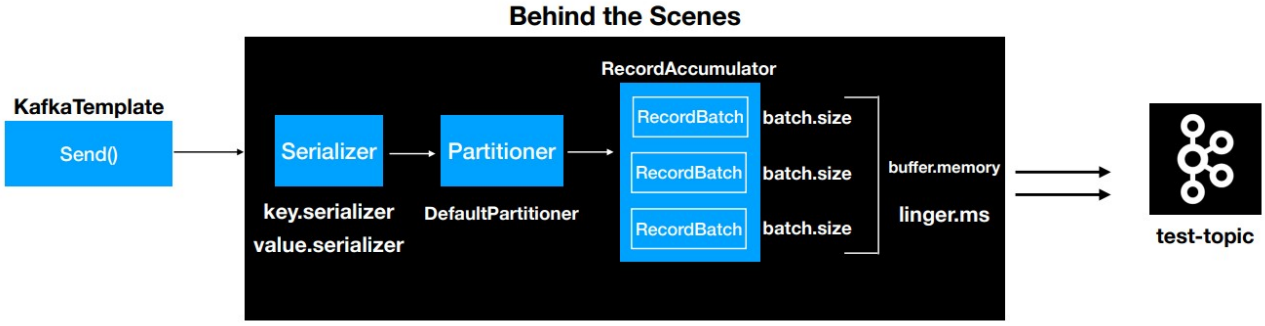


Kafka Producer – KafkaTemplate (Spring)

Kafka Template is similar to the JdbcTemplate for DB interactions. It is used to produce records into Kafka Topic.

The KafkaTemplate wraps a producer and provides convenience methods to send data to Kafka topics. You will notice many variations of send and sendDefault methods.

The message goes through different layers before it reaches.



Serializer

The record needs to be serialized to bytes. There are two types of Serializer and is mandatory for the Producer and the client need to configure them by providing configuration values for the Serializer.

1. Key Serializer (**key.serializer**)
2. Value Serializer (**value.serializer**)

Partitioner

It determines which partition the message is going to go in the Topic. Configuring the Partitioner is not mandatory as the DefaultPartitioner comes into action if no Partitioner is configured. The DefaultPartitioner is more than enough most of the times

RecordAccumulator

Any record sent by the Producer/KafkaTemplate won't get sent immediately to the topic. This helps to increase the performance by limiting the number of trips to the Kafka Cluster. If the topic has three partitions. Then, there will be three RecordBatches (one for each partition) to accumulate the records.

Every RecordBatch has a batch size (specified by batch.size)
Also, the RecordAccumulator has a overall buffer memory (specified by buffer.memory)
The record accumulator buffers the record. The records are sent to the topic once the buffer (**buffer.memory**) is full.

Once the batch is full, the records are sent to the topic. If the batch size doesn't fill up for a long time, the producer will not wait for long. Instead it will send after the threshold time configured (**linger.ms**) even if the batch does not fill.

Simple Configuration of KakfaTemplate

Mandatory Values:

```
bootstrap-servers: localhost:9092,localhost:9093,localhost:9094
key-serializer: org.apache.kafka.common.serialization.IntegerSerializer
value-serializer: org.apache.kafka.common.serialization.StringSerializer
```

Autocreate Topic using KakfaAdmin
This is **not a recommended approach to create topic programmatically.** KakfaAdmin is part of SpringKakfa dependency.

To create a topic from code
Create a Bean of type **KafkaAdmin** in SpringConfiguration
Create a Bean of type **NewTopic** in SpringConfiguration

application-local.yml (KafkaAdmin Bean):

```
spring:
  kafka:
    topic:  library-events
    admin:
      properties:
        bootstrap.servers: localhost:9092,localhost:9093,localhost:9094
```

Bean of Type NewTopic:

```
@Configuration
@Profile("local")
public class AutoCreateTopicConfig {
    @Value("${spring.kafka.topic}")
    public String topic;

    @Bean
    public NewTopic libraryEvents(){
        return TopicBuilder.name(topic)
                .partitions(3)
                .replicas(3)
                .build();
    }
```

}

Kafka Template - Producer API

send(org.apache.kafka.clients.producer.ProducerRecord<K,V> record)
The send method which accepts ProducerRecord is helpful when you want to send headers for the message.

Reference:
https://docs.spring.io/spring-kafka/reference/html/#sending-messages

Integration Testing using JUnit5

Why we need Automated Tests? - Manual Testing is error-prone, time consuming and slows down delivery. Automated Tests can run as part of your build process which is a requirement for today's software development. Also, it is easy to catch bugs as it will fail the build.
Types of Automated Tests – Unit Test, Integration Test and End to End Tests.

Integration Test – It tests all the layers of the code and verify their behaviour is working as expected.
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
We start the whole spring application using the @SpringBootTest

EmbeddedKafka – Configure and Integrate in Junit
EmbeddedKafka is needed if the integration test is running in a CI/CD pipeline as we cannot start a kakfa instance locally and run the test.

@EmbeddedKafka(**topics** = {"***library-events***"}, **partitions** = ***3***)
@TestPropertySource(properties =
{"spring.kafka.producer.bootstrap-servers=${***spring.embedded.kafka.brokers***}",
"spring.kafka.admin.properties.bootstrap.servers=${***spring.embedded.kafka.brokers***}"})

The above starts an EmbeddedKafka instance and we the provide the values for the properties spring.kafka.producer.bootstrap-servers and spring.kafka.admin.properties.bootstrap.servers with the values from the EmbeddedKafka instance namely the values of spring.embedded.kafka.brokers and spring.embedded.kafka.brokers

@Autowired
private EmbeddedKafkaBroker embeddedKafkaBroker;
We can access the EmbeddedKafka broker instance that was started as part of the test by Autowiring the EmbeddedKafkaBroker

Sample Integration Test Case
Produce Message via the Application Rest Endpoint.
Validate by consuming the published message.

```
import org.springframework.kafka.test.utils.KafkaTestUtils;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.kafka.core.DefaultKafkaConsumerFactory;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
```

```
@EmbeddedKafka(topics = {"library-events"}, partitions = 3)
@TestPropertySource(properties =
    {"spring.kafka.producer.bootstrap-servers=${spring.embedded.kafka.brokers}",
     "spring.kafka.admin.properties.bootstrap.servers=${spring.embedded.kafka.brokers}"})
public class LibraryEventsControllerIntegrationTest {

   // Will have the base path configured with the Random Port the application starts
   @Autowired
   TestRestTemplate testRestTemplate;

   @Autowired
   private EmbeddedKafkaBroker embeddedKafkaBroker;

   private Consumer<Integer,String> consumer;

   // We are setting up a consumer to start consumimg/subscribing for the messges
   @BeforeEach
   void setUp() {
      Map<String, Object> configs = new HashMap<>(KafkaTestUtils.consumerProps
                                          ("group1", "true", embeddedKafkaBroker));
      configs.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "latest");
      consumer = new DefaultKafkaConsumerFactory<>(configs,
                                                   new IntegerDeserializer(),
                                                   new StringDeserializer())
                                                    .createConsumer();
      embeddedKafkaBroker.consumeFromAllEmbeddedTopics(consumer);
   }

   // Close the Consumer
   @AfterEach
   void tearDown() {
      consumer.close();
   }

   // Publish the message using the Application Rest End Point
   // Consume the published message to validate.
   @Test
   void postLibraryEvent() {
       ....
       ....
      ResponseEntity<LibraryEvent> responseEntity =
               testRestTemplate.exchange("/v1/libraryevent", HttpMethod.POST, request,
                                   LibraryEvent.class);
      assertEquals(HttpStatus.CREATED, responseEntity.getStatusCode());

      ConsumerRecords<Integer, String> consumerRecords =
                                          KafkaTestUtils.getRecords(consumer);
      consumerRecords.forEach(record -> {
         var libraryEventActual = parseLibraryEventRecord(objectMapper, record.value());
         assertEquals(libraryEvent, libraryEventActual);
      });
   }
```

}

Unit Testing using JUnit5

Unit Tests are faster compared to Integration Test because it doesn't need the whole environment to run the test.

All external dependencies need to be mocked out. For Example., If you are going to test the controller. Then, you mock the service layer, mock the writing of messages(Producer) and all external dependencies.

We are going to just test the web layer without starting the complete spring application and not the whole application context.

We just need a slice of the spring application context and test/load only the web layer using the @WebMvcTest

@SpringBootTest – Use it only for Integration Test

Whereas we use @WebMvcTest

1. **@WebMvcTest** – Unit Test the Web Layer
2. Autowire **MockMvc** bean to invoke the endpoints
3. **@MockBean** to mock the dependencies

Validation API

When we add the @Valid annotation to the controller method
public ResponseEntity<LibraryEvent> postLibraryEvent (@RequestBody **@Valid** LibraryEvent libraryEvent) { ... }
If the validation fails. Then, **we get an exception (MethodArgumentNotValidException) and HttpStatusCode of 400 (Client Error – Bad Request)**

We can customize this by using an Exception Handler for the Controller using ControllerAdvice.
Note: **If you don't write an Exception Handler, Spring Framework will use the DefaultHandlerExceptionResolver** (org.springframework.web.servlet.mvc.support) as part of the framework. **DefaultHandlerExceptionResolver is the default implementation of the HandlerExceptionResolver interface that resolves standard Spring exceptions and translates them to corresponding HTTP status codes**.

Let's write an custom exception handler for the MethodArgumentNotValidException

```
@ControllerAdvice
public class LibraryEventControllerAdvice {
   @ExceptionHandler(MethodArgumentNotValidException.class)
   public ResponseEntity<?> handleRequestBody(MethodArgumentNotValidException ex) {

      List<FieldError> errorList = ex.getBindingResult().getFieldErrors();
      String errorMessage = errorList.stream()
            .map(fieldError -> fieldError.getField() + " - " + fieldError.getDefaultMessage())
            .sorted()
```

```
            .collect(Collectors.joining(", "));
        return new ResponseEntity<>(errorMessage, HttpStatus.BAD_REQUEST);
    }
}
```

Kakfa Producer Configurations

acks – Possible values (0 , 1 and -1)
retries – Number of retries when there is any failure in producing the messages to kafka.
retries.backoff.ms – Integer value in milliseconds (default value is 100)

acks

When the producer.send call is considered successful.

- acks = 1 (Guarantees the message is written to the Leader)
- acks = -1 (Guarantees the message is written to the Leader and all the replicas, This is the Default)
- acks = 0 (No Guarantee, This is not recommended. Doesn't considered whether message is written to Leader / Replicas. Considered successful as soon as the send call is invoked)

retries

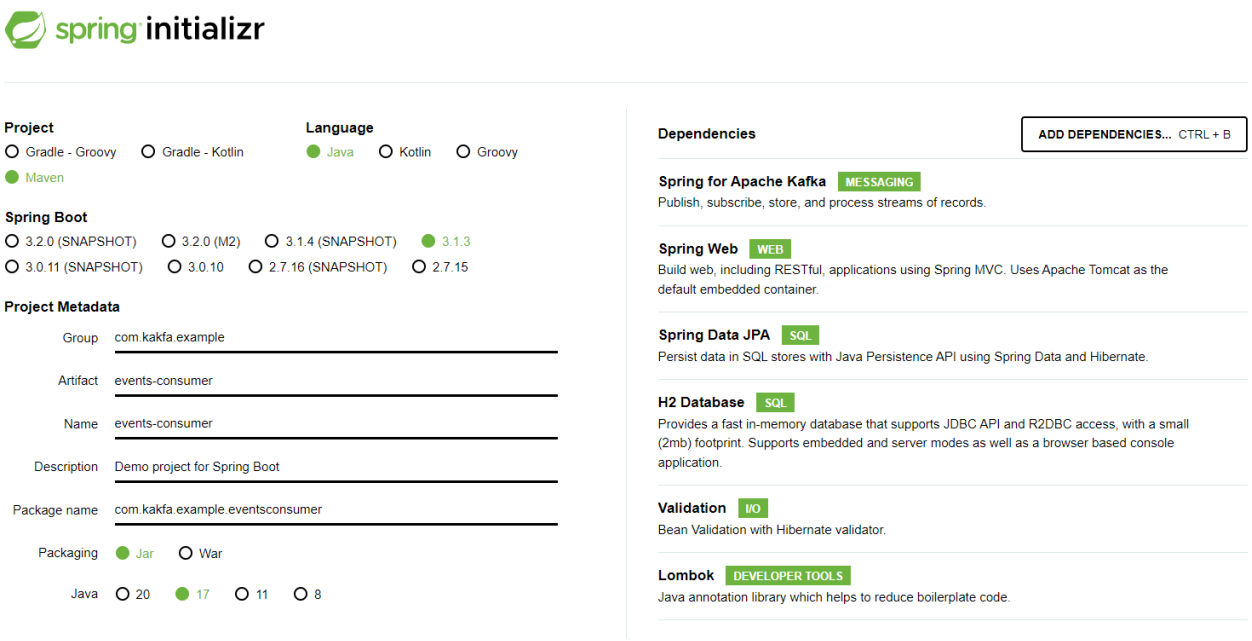
Integer value – 0 to 21474483647
Default value for retries is 21474483647

To override the values (Set the value in application.yml for the producer)

```
spring:
  kafka:
    topic:  library-events
    template:
      default-topic: library-events
    producer:
      bootstrap-servers: localhost:9092,localhost:9093,localhost:9094
      key-serializer: org.apache.kafka.common.serialization.IntegerSerializer
      value-serializer: org.apache.kafka.common.serialization.StringSerializer
      properties:
        acks: all
        retries: 10
        retry.backoff.ms: 1000
```

Ref: https://docs.confluent.io/platform/current/installation/configuration/producer-configs.html

Build SpringBoot Kafka Consumer– Hands On



Options for Kakfa Consumer to consume messages in Spring

1. MessageListenerContainer
    1. KakfaMessageListenerContainer
    2. ConcurrentMessageListener
2. @KakfaListener Annotation (Uses ConcurrentMessageListener behind the scenes)

KakfaMessageListenerContainer
Implementation of MessageListener
Polls the records and commits the offsets after the records are processed
Single Threaded

ConcurrentMessageListener
Represents multiple instances of KafkaMessageListenerContainer
Can poll the Kakfa topic using multiple threads

@KafkaListener and Configuration
Easiest way to build Kakfa Consumer

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

N.B: The method need not be onMessage, it can be any name

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

Configure the consumer in yml

```
spring:
  kafka:
```

```
  template:
    default-topic: library-events
  consumer:
    bootstrap-servers: localhost:9092,localhost:9093,localhost:9094
    key-deserializer: org.apache.kafka.common.serialization.IntegerDeserializer
    value-deserializer: org.apache.kafka.common.serialization.StringDeserializer
    group-id: library-events-listener-group
    auto-offset-reset: latest
```

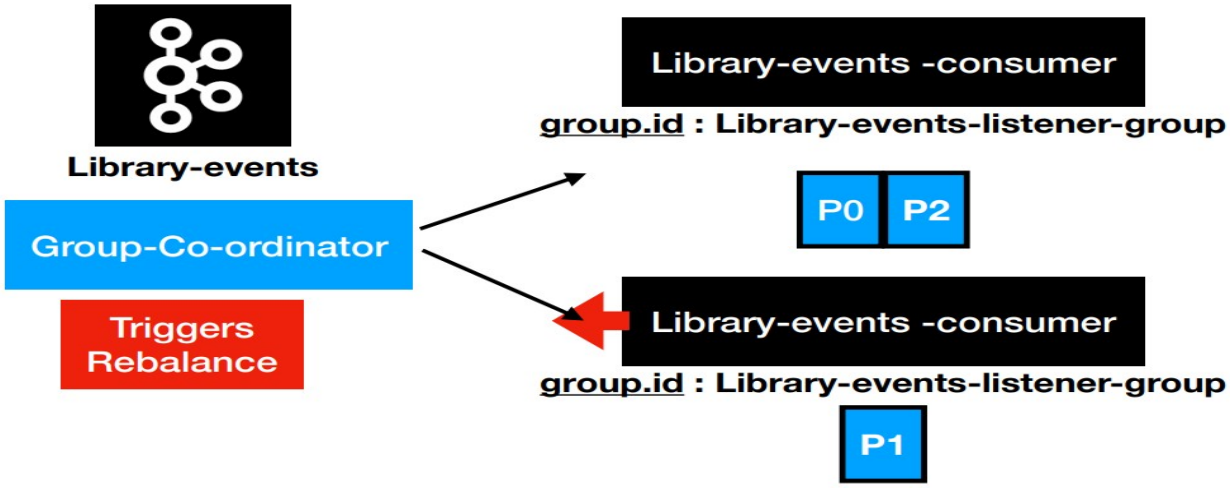
Rebalance

Rebalance is changing the partition ownership from one consumer to another.

Topic: library-events
Number of Paritions: 3
1 Consumer of group id (library-events-listener-group) is live and is consuming all partitions

Now, New Consumer comes in with the same group id (library-events-listener-group) and starts to listen to the topic(library-events)

After another consumer with the same group id joins as a consumer. Then, the Group Co-ordinator takes care of doing the rebalance. It triggers a rebalance. One of the **partition P1 is revoked** from the existing consumer and is allocated to the new consumer.



Consumer Offset Management

The offset is a position within a partition for the next message to be sent to a consumer. Kafka maintains two types of offsets.

1. Current offset
2. Committed offset

Current Offset

When we call a poll method, Kafka sends some messages to us. Let us assume we have 100 records in the partition. The initial position of the current offset is 0. We made our first call and received 20 messages. Now Kafka will move the current offset to 20. When we make our next request, it will send some more messages starting from 20 and again move the current offset forward. The offset is a simple integer number that is used by Kafka to maintain the current position of a consumer. That's

it. **The current offset is a pointer to the last record that Kafka has already sent to a consumer in the most recent poll.** So, the consumer doesn't get the same record twice because of the current offset

Committed Offset

This offset is the position that a consumer has confirmed about processing. Once we are sure that we have successfully processed the record, we may want to commit the offset. So, **the committed offset is a pointer to the last record that a consumer has successfully processed**. The committed offset is critical in the case of partition rebalance. In the event of rebalancing. When a new consumer is assigned a new partition, it should ask a question. Where to start? What is already processed by the previous owner? The answer to the question is the committed offset.

As a consumer in the group reads messages from the partitions assigned by the coordinator, it must commit the offsets corresponding to the messages it has read. If the consumer crashes or is shut down, its partitions will be re-assigned to another member, which will begin consumption from the last committed offset of each partition.

Default Consumer Offset Management

Consumer is constantly polling the records. Once consumer reads the message. Behind the scenes, the offset information is written to the __consumer_offsets topic. This is called committing the offsets. There are many different options available for committing the offset (RECORD, BATCH, TIME, MANUAL etc.,)

Now during the next poll loop, it knows from where (ie., the offset position) the records are to be read. This way, the same set of consumer records are not read again by the consumer

Manual Consumer Offset Management

1. Change the default Acknowledgement Mode
2. Manually Commit by invoking the Acknowledgement

Configure the Acknowledgement Mode:

```
@Bean
 ConcurrentKafkaListenerContainerFactory<?, ?> kafkaListenerContainerFactory(
   ConcurrentKafkaListenerContainerFactoryConfigurer configurer,
   ObjectProvider<ConsumerFactory<Object, Object>> kafkaConsumerFactory,
   ObjectProvider<ContainerCustomizer<Object, Object,
   ConcurrentMessageListenerContainer<Object, Object>>> kafkaContainerCustomizer
 )
 {
   ConcurrentKafkaListenerContainerFactory<Object, Object> factory =
            new ConcurrentKafkaListenerContainerFactory();
   configurer.configure(factory, (ConsumerFactory)kafkaConsumerFactory.getIfAvailable(() -> {
   return new DefaultKafkaConsumerFactory(this.properties.buildConsumerProperties());
   }));
   Objects.requireNonNull(factory);
   kafkaContainerCustomizer.ifAvailable(factory::setContainerCustomizer);
   factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL);;
   return factory;
 }
```

Manually Commit by invoking the Acknowledgement:

```
@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 in Manual Offset Consumer: {} ", consumerRecord );
    // Do the Processing Logic and invoke the acknowledge if processing is successful
    ...
    ...
    acknowledgment.acknowledge();
  }
}
```

Reference:
https://docs.confluent.io/platform/current/clients/consumer.html#offset-management

Concurrent Consumers

The @KakfaListener uses ConcurrentMessageListener behind the scenes. But, by default it runs in a single-thread. Hence, the Consumers consume from different partitions in a single-threaded fashion in a sequential manner.

This is fine in a cloud-like environment where we run multiple instances of the consumer application wherein each instance of consumer (belonging to the same consumer group) can connect to different partitions and read the messages at the same time

But in an on-prem environment, where we cannot run multiple instances of the consumer application, **we need to make the single instance of the application to run concurrent consumers so that we can concurrently read from the different partitions of the topic at the same time. Multiple consumer instances and each instance of the consumer need to run in a different thread)**

This is the default configuration of kafkaListenerContainerFactory that configures the @KafkaListener Annoation (Check the class: KafkaAnnotationDrivenConfiguration.class in package org.springframework.boot.autoconfigure.kafka)

```
@Bean
@ConditionalOnMissingBean(
                name = {"kafkaListenerContainerFactory"}
)
ConcurrentKafkaListenerContainerFactory<?, ?> kafkaListenerContainerFactory(
                ConcurrentKafkaListenerContainerFactoryConfigurer configurer,
                ObjectProvider<ConsumerFactory<Object, Object>> kafkaConsumerFactory,
                ObjectProvider<ContainerCustomizer<Object, Object,
                ConcurrentMessageListenerContainer<Object, Object>>> kafkaContainerCustomizer
) {
                ConcurrentKafkaListenerContainerFactory<Object, Object> factory = new
                                ConcurrentKafkaListenerContainerFactory();
                configurer.configure(factory,
                                (ConsumerFactory)kafkaConsumerFactory.getIfAvailable(() -> {
```

```
        return new DefaultKafkaConsumerFactory(this.properties.buildConsumerProperties());
        }));
        Objects.requireNonNull(factory);

        kafkaContainerCustomizer.ifAvailable(factory::setContainerCustomizer);

        return factory;
}
```

We will override the configuration in our consumer application and supply the customerized configuration bean for the Consumer that enables concurrency

```
@Component
@Slf4j
public class ConcurrentEventsConsumer {
  private final KafkaProperties properties;

  public ConcurrentEventsConsumer(KafkaProperties properties) {
   this.properties = properties;
  }

  @Bean
  ConcurrentKafkaListenerContainerFactory<?, ?> kafkaListenerContainerFactory(
    ConcurrentKafkaListenerContainerFactoryConfigurer configurer,
    ObjectProvider<ConsumerFactory<Object, Object>> kafkaConsumerFactory,
    ObjectProvider<ContainerCustomizer<Object, Object,
    ConcurrentMessageListenerContainer<Object, Object>>> kafkaContainerCustomizer
  )
  {
    ConcurrentKafkaListenerContainerFactory<Object, Object> factory =
            new ConcurrentKafkaListenerContainerFactory();
    configurer.configure(factory, (ConsumerFactory)kafkaConsumerFactory.getIfAvailable(() -> {
    return new DefaultKafkaConsumerFactory(this.properties.buildConsumerProperties());
    }));
    Objects.requireNonNull(factory);
    kafkaContainerCustomizer.ifAvailable(factory::setContainerCustomizer);
    factory.setConcurrency(3);
    return factory;
  }
}
```

Alternate Approach:

```
@KafkaListener( id = "transactions",  topics = "transactions",   groupId = "a",
                concurrency = "3")
public void listen(Order order) {
   LOG.info("Received: {}", order);
   service.process(order);
}
```

References
https://piotrminkowski.com/2023/04/30/concurrency-with-kafka-and-spring-boot/

How SpringBoot AutoConfiguration works

SpringBootAutoConfiguration class for Kafka:
org.springframework.boot.autoconfigure.kafka.KafkaAutoConfiguration

Further Reading
https://kafka.apache.org/0100/protocol.html