Learn more here:

<https://kafka.apache.org/documentation/>

# Source Code

**Spring Boot + Kafka**

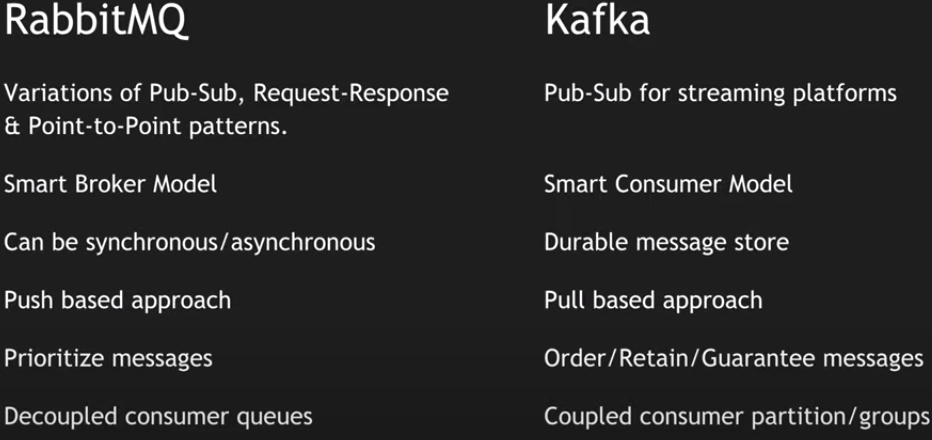
<https://github.com/anilkumarb1248/Kafka-Demos>

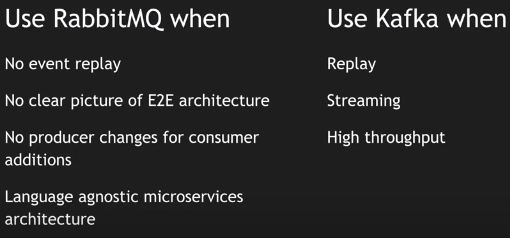
<https://github.com/anilkumarb1248/kafka-springboot-demos>

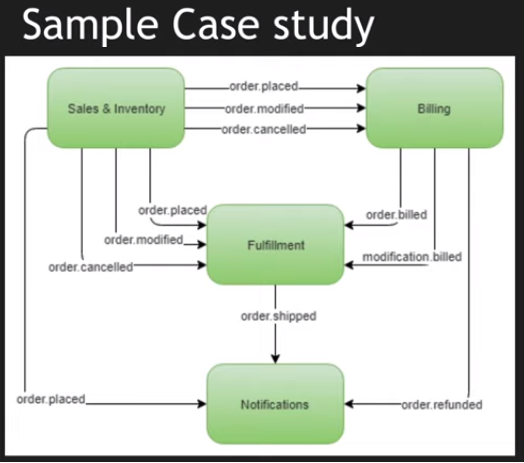
# Diff b/n Kafka & RabbitMQ

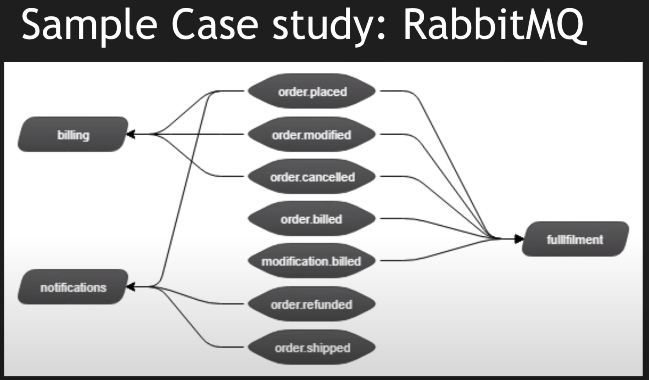
<https://www.upsolver.com/blog/kafka-versus-rabbitmq-architecture-performance-use-case>

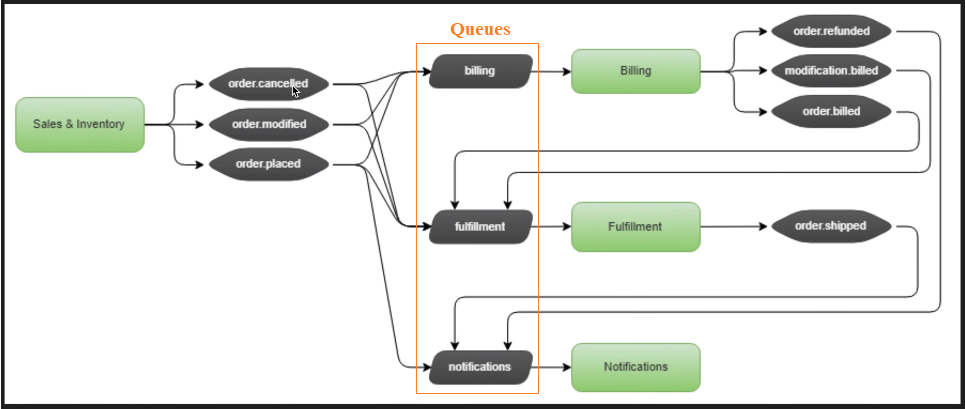
<https://www.youtube.com/watch?v=GMmRtSFQ5Z0>

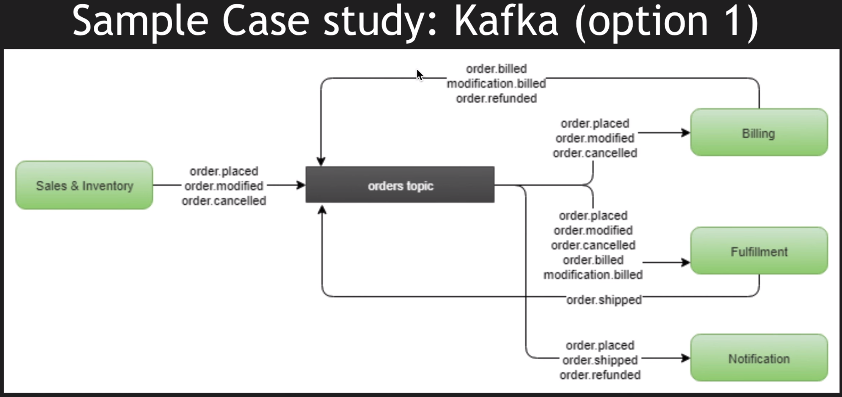


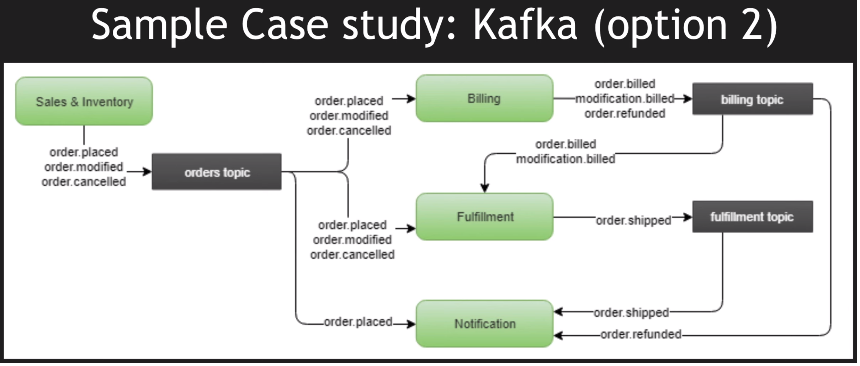












## Differences

**RabbitMQ** is an open source message broker that uses a messaging queue approach. Queues are spread across a cluster of nodes and optionally replicated, with each message only being delivered to a single consumer.

|  |  |  |
| --- | --- | --- |
| **Characteristics** | **Apache Kafka** | **RabbitMQ** |
| **Architecture** | Kafka uses a partitioned log model, which combines messaging queue and publish subscribe approaches. | RabbitMQ uses a messaging queue. |
| **Scalability** | Kafka provides scalability by allowing partitions to be distributed across different servers. | Increase the number of consumers to the queue to scale out processing across those competing consumers. |
| **Message retention** | Policy based, for example messages may be stored for one day. The user can configure this retention window. | Acknowledgement based, meaning messages are deleted as they are consumed. |
| **Multiple consumers** | Multiple consumers can subscribe to the same topic, because Kafka allows the same message to be replayed for a given window of time. | Multiple consumers cannot all receive the same message, because messages are removed as they are consumed. |
| **Replication** | Topics are automatically replicated, but the user can manually configure topics to not be replicated. | Messages are not automatically replicated, but the user can manually configure them to be replicated. |
| **Message ordering** | Each consumer receives information in order because of the partitioned log architecture. | Messages are delivered to consumers in the order of their arrival to the queue. If there are competing consumers, each consumer will process a subset of that message. |
| **Protocols** | Kafka uses a binary protocol over TCP. | Advanced messaging queue protocol (AMQP) with support via plugins: MQTT, STOMP. |

# Apache Kafka Introduction

## Overview

### What is Apache Kafka?

Apache Kafka is an open-source **distributed event streaming platform** used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications.

Apache Kafka is a distributed data store optimized for ingesting and processing streaming data in real-time. Streaming data is data that is continuously generated by thousands of data sources, which typically send the data records in simultaneously. A streaming platform needs to handle this constant influx of data, and process the data sequentially and incrementally.

Kafka provides three main functions to its users:

* Publish and subscribe to streams of records
* Effectively store streams of records in the order in which records were generated
* Process streams of records in real time

Kafka is primarily used to build real-time streaming data pipelines and applications that adapt to the data streams. It combines messaging, storage, and stream processing to allow storage and analysis of both historical and real-time data.

* Apache Kafka is a distributed event streaming platform used to **collect**, **process**, **store**, and **integrate data** at scale. It has numerous use cases including ***distributed logging***, ***stream processing***, ***data integration***, and ***pub/sub messaging***.
* Kafka is written in Scala and Java.
* Kafka is fast, scalable, durable, fault-tolerant and distributed by design.

**What Apache Kafka can do?**

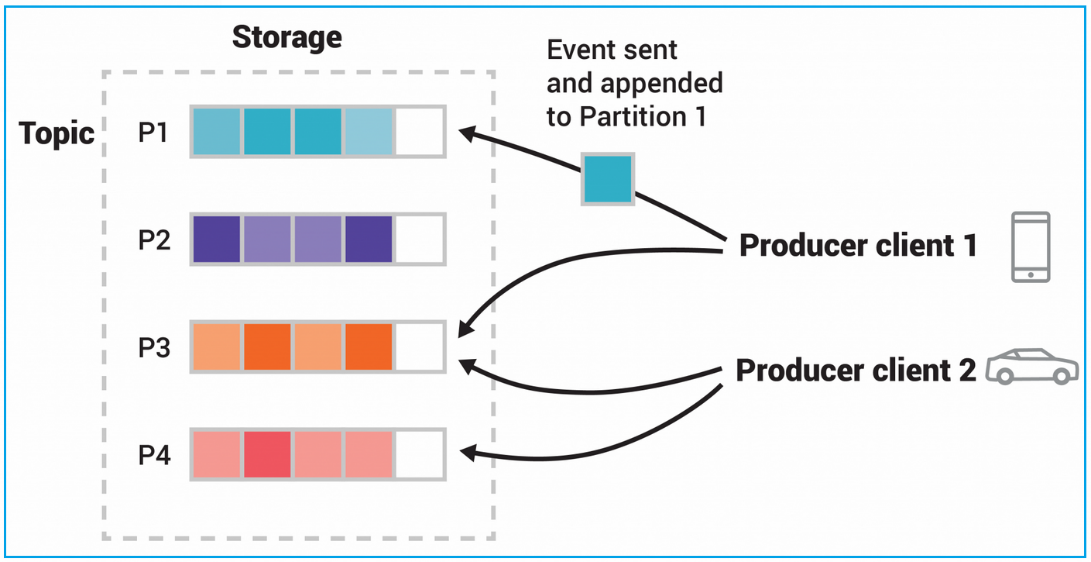
* It can be used as an enterprise messaging system.
* It can be used for stream processing
* It can also provides connectors to import and export bulk data from databases and other systems.

#### What are Events?

An event is any type of action, incident, or change that's identified or recorded by software or applications.

For example, a payment, a website click, or a temperature reading, along with a description of what happened.

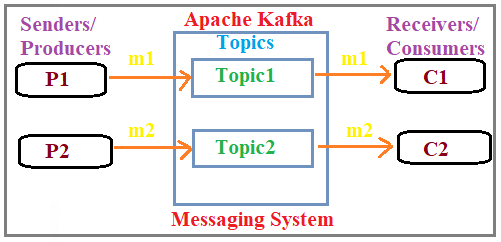
An event **records the fact that "something happened" in the world or in your business**. It is also called record or message in the documentation. When you read or write data to Kafka, you do this in the form of events. Conceptually, an ***event has a key, value, timestamp, and optional metadata headers***.



This example topic has four partitions P1–P4. Two different producer clients are publishing, independently from each other, new events to the topic by writing events over the network to the topic's partitions. Events with the same key (denoted by their color in the figure) are written to the same partition. Note that both producers can write to the same partition if appropriate.

#### What is a messaging system?

A messaging system is a **simple exchange of messages** between two or more persons, devices, etc. A “publish-subscribe” messaging system allows a sender to send/write the message and a receiver to read that message.



In Apache Kafka, a sender is known as a **producer** who publishes messages, and a receiver is known as a **consumer** who consumes that message by subscribing it.

### Why Apache Kafka?

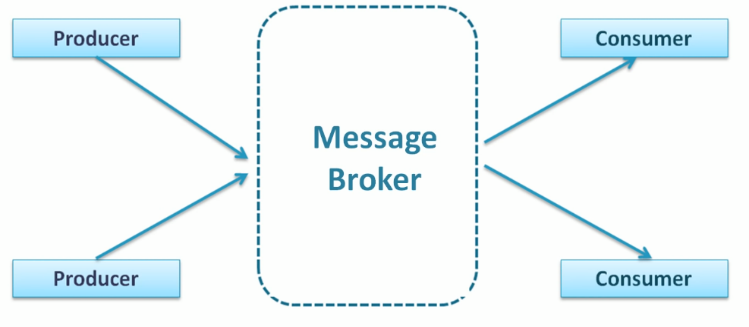
**Why would we use Kafka?**

Kafka is used to build real-time streaming data pipelines and real-time streaming applications. A data pipeline reliably processes and moves data from one system to another, and a streaming application is an application that consumes streams of data. For example, if you want to create a data pipeline that takes in user activity data to track how people use your website in real-time, Kafka would be used to ingest and store streaming data while serving reads for the applications powering the data pipeline. Kafka is also often used as a message broker solution, which is a platform that processes and mediates communication between two applications.

#### Why do we need Apache Kafka?

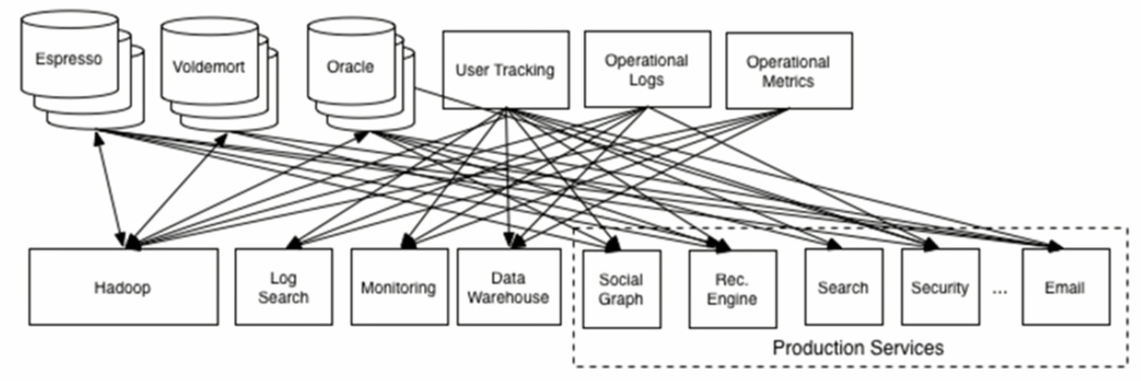
##### Kafka as a Message Broker

Let’s say Apache Kafka works as a Message Broker.

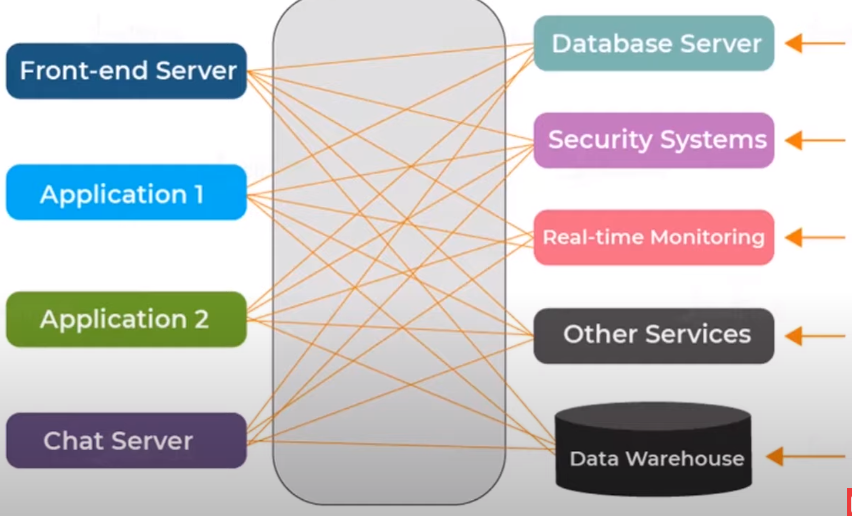


###### How systems will interact without Kafka

Assume that, without having any Message Broker like Kafka, how the systems will connect each other.

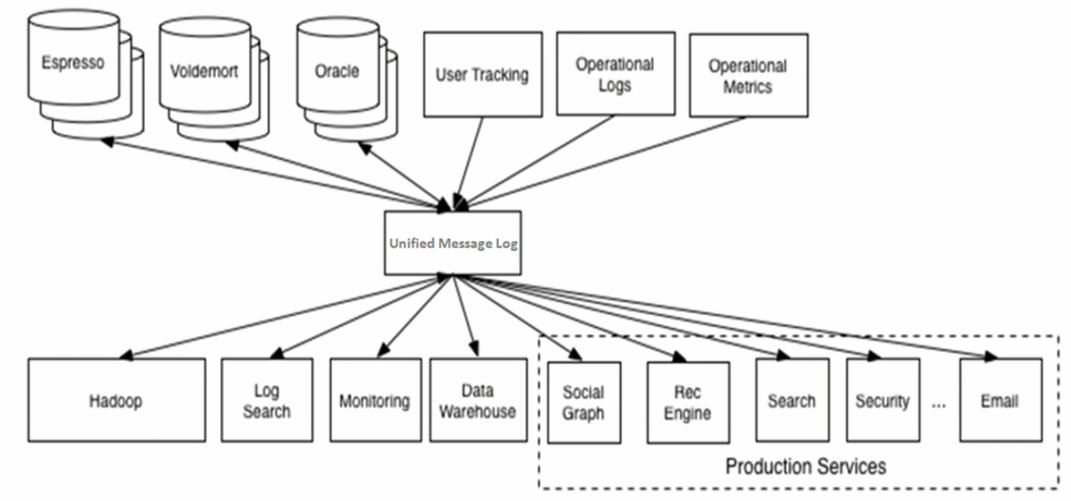


* You can see that the data pipelines are getting complex with the increase in the number of systems.
* Adding a new system or a server requires more data pipelines which will make the data flow more complicated.
* Managing these data pipelines becomes very difficult as each data pipeline has its own set of requirements.
* Adding or removing some pipelines is difficult in such cases.

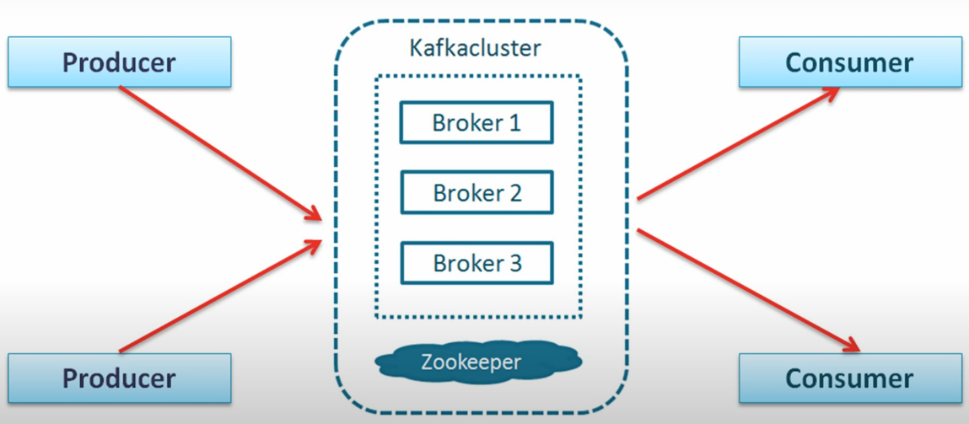


###### With Kafa as Message Broker

If we have a Messag Broker like Kafka in between the Source systems and Target Systems.

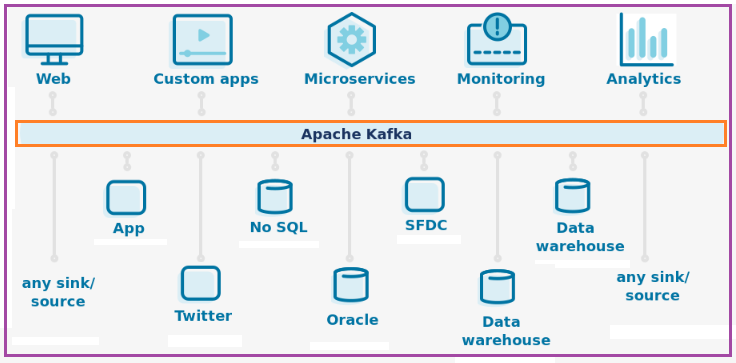


We can use the Kafka as a Message Broker between the Source systems and Target Systems.



* Apache Kafka reduces the complexity of data pipelines.
* It makes communication between systems simpler and manageable.
* With Kafka, it is easy to establish remote communication and send data across a network.
* We can establish asynchronous communication and send messages with the help of Kafka.
* It ensures reliable communication.

### Where Apache Kafka fits in?



### Kafka Usecases

Apache Kafka is a software platform that has the following reasons which best describes the need of Apache Kafka.

#### Basic

* **Publish-subscribe messaging**
* **Stream processing**
* **Log aggregation**
* **Data Integration**
* **Metrics & monitoring**
* Apache Kafka is capable of **handling millions of data** or messages per second.
* Apache Kafka works as a **mediator** between the source system and the target system. Thus, the source system (producer) data is sent to the Apache Kafka, where it decouples the data, and the target system (consumer) consumes the data from Kafka.
* Apache Kafka is having extremely **high performance**, i.e., it has really **low latency** value less than 10ms which proves it as a well-versed software.
* Apache Kafka has a **resilient architecture** which has resolved unusual complications in data sharing.
* Apache Kafka is able to maintain the **fault-tolerance**. Fault-tolerance means that sometimes a consumer successfully consumes the message that was delivered by the producer. But, the consumer fails to process the message back due to backend database failure, or due to presence of a bug in the consumer code. In such a situation, the consumer is unable to consume the message again. Consequently, Apache Kafka has resolved the problem by reprocessing the data.
* Organizations such as NETFLIX, UBER, Walmart, etc. and over thousands of such firms make use of Apache Kafka.

#### Messaging:

Kafka works best on messaging. Since it is the primary work of Kafka. The servers of Kafka known as brokers are best in buffering the messages which are yet to be published and also to start the processing at the consumers by ending it in the producers. It is proficient in transferring the message of huge sets of of data due to its high resistance capabilities.

#### Website Activity Tracking:

The Kafka is a good website activity tracker. Every page visit, including each and every URL can be found by Kafka. Each search activity, including page views is done perfectly by this web tracker.

#### Log Aggregation:

Kafka also works best in aggregating log files from different servers and adding them up in the main server system.

#### Commit log:

In other distributed systems, it acts as a log replicating the information between nodes and acts as substitutes for the broken nodes.

#### Event sourcing:

All the changes in the stages in the stream changes are arranged in a series relating to time.

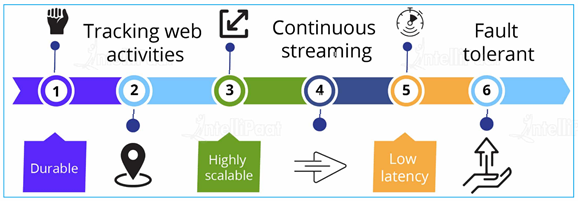
#### Metrics:

Kafka aggregates data from several numbers of applications to obtain a one main data. So it monitors a large number of data pipelines to collect the required sets of data from them.

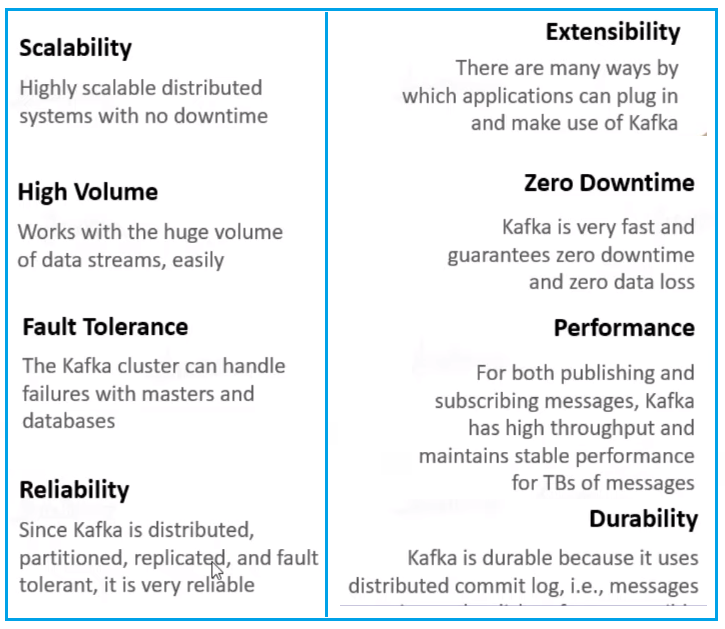
#### Stream processing:

The processing of the data from the Topics of fresh data and then accumulating them, enriching them and then sending to newer Topics are all stream processing done by the Kafka.

### Kafka Features / Advantages



1. **Durable**: Apache Kafka lets us replicate messages, and the messages persist as fast as possible on the disk which makes Kafka durable as well.
2. **Tracking web activities**: Apache Kafka keeps a track of web activities by storing or sending events for real-time processes.
3. **Highly scalable**: Apache Kafka shows high scalability, without any downtime.
4. **Continuous streaming**: Apache Kafka keeps up the continuous processing of streaming data.
5. **Low latency**: Apache Kafka decreases the latency, making it possible for you to deliver data in mere milliseconds in real time.
6. **Fault-tolerant**: Kafka is highly available and resilient to node failures and supports automatic recovery.
7. **High throughput**: Kafka is able to handle a high volume of data even at a high velocity. It also supports high message throughput, i.e., it can handle thousands of messages every second.
8. **Standard format**: It transforms data of different formats into a standard format so that there is no ambiguity.



### What is Streaming process?

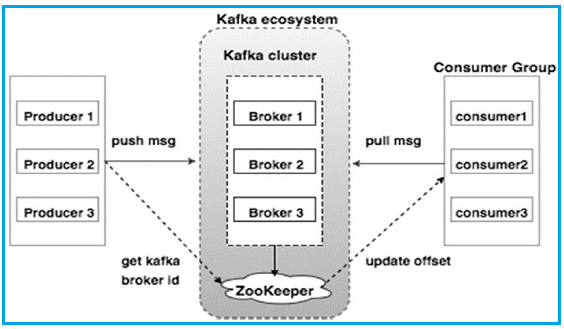
Stream is nothing but **“*Continous flow of data or Constant stream of messages*”**

A streaming process is the processing of data in parallelly connected systems. This process allows different applications to limit the parallel execution of the data, where one record executes without waiting for the output of the previous record. Therefore, a distributed streaming platform enables the user to simplify the task of the streaming process and parallel execution. Therefore, a streaming platform in Kafka has the following key capabilities:

* As soon as the streams of records occur, it processes it.
* It works similar to an enterprise messaging system where it publishes and subscribes streams of records.
* It stores the streams of records in a fault-tolerant durable way.

## Apache Kafka Architecture

Take a look at the following illustration. It shows the cluster diagram of Kafka.



### Broker

Kafka cluster typically consists of multiple brokers to maintain load balance. Kafka brokers are stateless, so they use ZooKeeper for maintaining their cluster state. One Kafka broker instance can handle hundreds of thousands of reads and writes per second and each broker can handle TB of messages without performance impact. Kafka broker leader election can be done by ZooKeeper.

### ZooKeeper

ZooKeeper is used for managing and coordinating Kafka broker. ZooKeeper service is mainly used to notify producer and consumer about the presence of any new broker in the Kafka system or failure of the broker in the Kafka system. As per the notification received by the Zookeeper regarding presence or failure of the broker then producer and consumer takes decision and starts coordinating their task with some other broker.

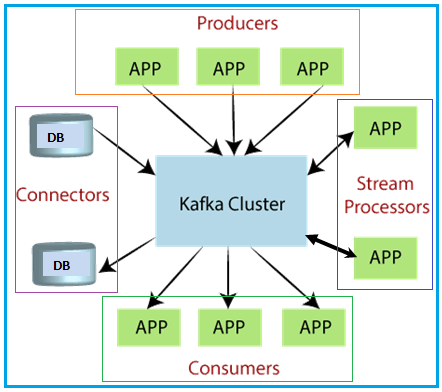
### Producers

Producers push data to brokers. When the new broker is started, all the producers search it and automatically sends a message to that new broker. Kafka producer doesn’t wait for acknowledgements from the broker and sends messages as fast as the broker can handle.

### Consumers

Since Kafka brokers are stateless, which means that the consumer has to maintain how many messages have been consumed by using partition offset. If the consumer acknowledges a particular message offset, it implies that the consumer has consumed all prior messages. The consumer issues an asynchronous pull request to the broker to have a buffer of bytes ready to consume. The consumers can rewind or skip to any point in a partition simply by supplying an offset value. Consumer offset value is notified by ZooKeeper.

## Apache Kafka Core APIs



### Producer API

This API allows/permits an application to publish streams of records to one or more topics.

### Consumer API

This API allows an application to subscribe one or more topics and process the stream of records produced to them.

### Streams API

This API allows an application to effectively transform the input streams to the output streams. It permits an application to act as a stream processor which consumes an input stream from one or more topics, and produce an output stream to one or more output topics.

### Connector API

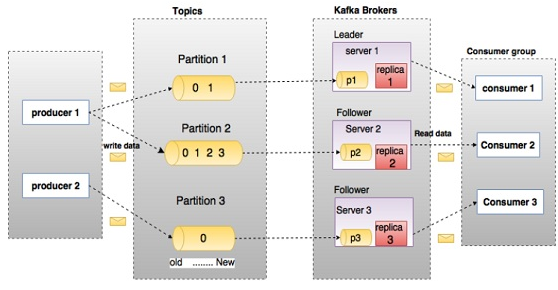
This API executes the reusable producer and consumer APIs with the existing data systems or applications.

## Kafka Terminology

<https://www.youtube.com/watch?v=udnX21__SuU&list=PLkz1SCf5iB4enAR00Z46JwY9GGkaS2NON&index=4>

|  |  |
| --- | --- |
| 1. Message 2. Producer 3. Consumer 4. Broker 5. Cluster 6. Topic | 1. Partitions 2. Offset 3. Consumer Groups 4. Replication 5. Leader 6. Follower |

Before moving deep into the Kafka, you must aware of the main terminologies such as topics, brokers, producers and consumers…etc. The following diagram illustrates the main terminologies and the table describes the diagram components in detail.



In the above diagram, a topic is configured into three partitions. Partition 1 has two offset factors 0 and 1. Partition 2 has four offset factors 0, 1, 2, and 3. Partition 3 has one offset factor 0. The id of the replica is same as the id of the server that hosts it.

Assume, if the replication factor of the topic is set to 3, then Kafka will create 3 identical replicas of each partition and place them in the cluster to make available for all its operations. To balance a load in cluster, each broker stores one or more of those partitions. Multiple producers and consumers can publish and retrieve messages at the same time.

|  |  |
| --- | --- |
| **Terminology** | **Description** |
| Producer | * An application that sends the data to Kafka * Producers are the processes that publish data or message to one or more topics * They are basically the source of data stream in Kafka |
| Consumer | * The application that receives the data from the Kafka * Consumers are the processes that read and process the data from topics by subscribing to one or more topics in the Kafka cluster |
| Message | * Small piece of data in the form of byte arrays * Messages are simply byte arrays, and any object can be stored in any format by developers. |
| Broker | * Kafka brokers are the servers that manage and mediate the conversation between two different systems. |
| Cluster | * In Kafka, more that one broker, i.e., a set of servers is collectively known as a Kafka cluster * It is a group of computers, each having one instance of Kafka broker. |
| Topic | * In Apache Kafka, all messages are maintained in what we call topics * Messages are stored, published and organized in Kafka topics |
| Partition | * Every broker holds few partitions and each partition can be either a leader or a replica for a topic * All 'writes' and 'reads' to a topic go via the leader, which is responsible for updating replicas with new data. * If the leader fails, the replica takes over as the new leader |
| Offset | * **Unique id** for a message within a partition |
| Consumer Groups | * A group of consumers acting as a signle logical unit |
| Replication | * Replicas are nothing but backups of a partition * Replicas are never read or write data, just update the changes with leader. * They are used to prevent data loss |
| Leader | * Leader is the node responsible for all reads and writes for the given partition |
| Follower | * Node which follows leader instructions are called as follower * If the leader fails, one of the follower will automatically become the new leader |

### Message

Message is nothing but a small to medium sized piece of data. For us a Message is a data, but for Kafka it is simple array of bytes.

***The unit of data within Kafka is called a message***. Think of this as a row in database table.

The massage has two parts – **key** and **body**. Both are simply an array of bytes and Kafka does not do anything magical to read and make sense of these bytes. It can be XML, JSON, String or anything. Many Kafka developers favor the use of **Apache Avro**, which is a serialization framework originally developed for Hadoop.

### Producer

***An application that sends messages/data to Kafka.***

Producers are the publisher of messages to one or more Kafka topics. Producers send data to Kafka brokers. Every time a producer publishes a message to a broker, the broker simply appends the message to the last segment file. Actually, the message will be appended to a partition. Producer can also send messages to a partition of their choice.

### Consumer

***An application that reads the message/data from Kafka.***

Consumers read data from brokers. Consumers subscribes to one or more topics and consume published messages by pulling data from the brokers.

### Broker

***Broker is nothing but a Kafka server (system) who acts as a Message Broker.***

Broker is an instance of Kafka that communicate with ZooKeeper. Each broker holds partition(s) of topic(s). Some of those partitions are leaders and others are replicas of leader partitions from other brokers.

A single Kafka server is called a broker. The broker receives messages from producer clients, assigns and maintain their offsets, and stores the messages in storage system.

It also services consumers, responding to fetch requests for partitions and responding with the messages that have been committed to disk.

If hardware support is good then a single broker can easily handle thousands of partitions and millions of messages per second.

Brokers are simple system responsible for maintaining the published data. Each broker may have zero or more partitions per topic.

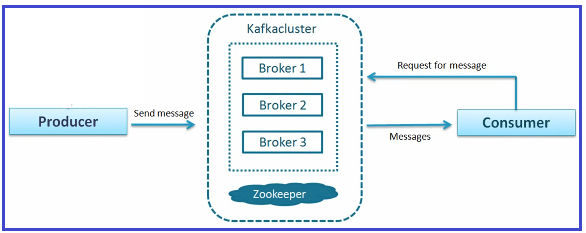
* Assume, if there are N partitions in a topic and N number of brokers, each broker will have one partition.
* Assume if there are N partitions in a topic and more than N brokers (n + m), the first N broker will have one partition and the next M broker will not have any partition for that particular topic.

### Cluster

***A group of computers sharing workload for a common purpose.***

***Kafka Cluster consists of multiple brokers and a Zookeeper.***

Kafka’s having more than one broker are called as Kafka cluster. A Kafka cluster can be expanded without downtime. These clusters are used to manage the persistence and replication of message data.



### Topic

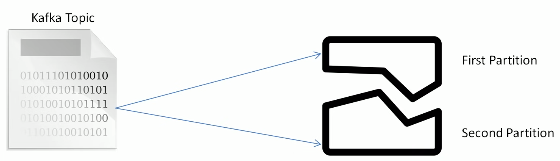
***A topic is a unique name for Kafka stream.***

A stream of messages belonging to a particular category is called a topic. Data is stored in topics.

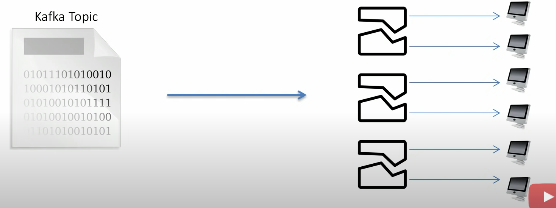
Topics are split into partitions. For each topic, Kafka keeps a minimum of one partition. Each such partition contains messages in an immutable ordered sequence. A partition is implemented as a set of segment files of equal sizes.

### Partitions

Topics may have many partitions, so it can handle an arbitrary amount of data.



Kafka doesn’t decides how many partitions should be there. We have to provide that partition information (number of partitions). So that the Kafka Broker creates that many partitions for our topic.

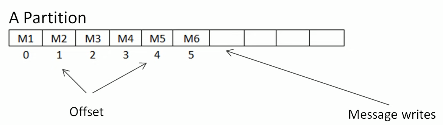


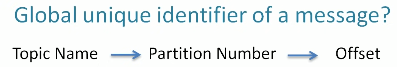
**Note**: Every partition should sit on a single machine. So do some estimation of partitions based on available systems.

### Offset

***A sequence id given to messages as they arrive in a partition.***

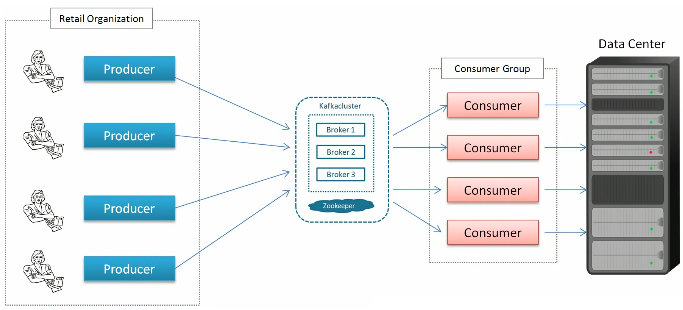
Each partitioned message has a unique sequence id called as offset





### Consumer Groups

***A group of consumers acting as single logical unit.***



### Replication

Replicas are nothing but **backups** of a partition. Replicas are never read or write data. They are used to prevent data loss.

### Leader

Leader is the node responsible for all reads and writes for the given partition. Every partition has one server acting as a leader.

### Follower

Node which follows leader instructions are called as follower. If the leader fails, one of the follower will automatically become the new leader. A follower acts as normal consumer, pulls messages and up-dates its own data store.

### Batch

A batch is just a collection of messages, all of which are being produced to the same topic and partition.

Messages move within network in form of batches. This is done for efficiency is network utilization.

Batches are also typically compressed, providing more efficient data transfer and storage at the cost of some processing power.

## Kafka Components

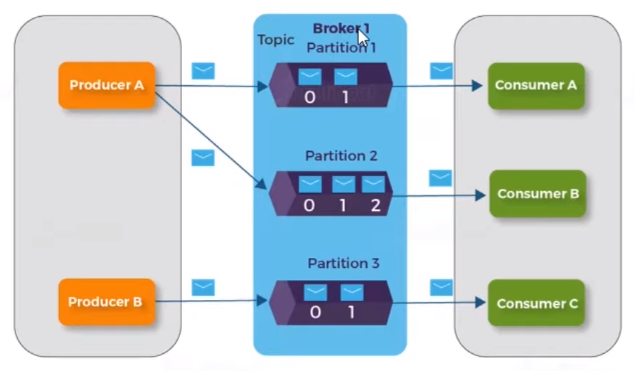
### Producer

* Producers send records (messages) to topics.
* Producers select the partition to send the message per topic.
* Producers can implement priority systems, which are based on sending records to certain partitions depending on the priority of the record.
* Producers send records to a partition based on the record's key.
* Producers don’t wait for acknowledgements from a broker and send messages as fast as the broker can handle.

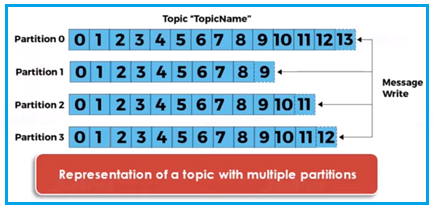
### Broker

* A cluster typically consists of multiple brokers to maintain the load balance.
* A broker on receiving messages from the producers assigns offsets to them and commits the messages to the storage on the disk.
* It serves consumers by responding to fetch requests for partitions.
* One broker instance can handle thousands of reads-writes per second and TBs of messages.
* Backups of topic partitions are present in multiple brokers.
* If a broker goes down, one of the brokers containing the backup partitions would be elected as the leader for the respective partitions.

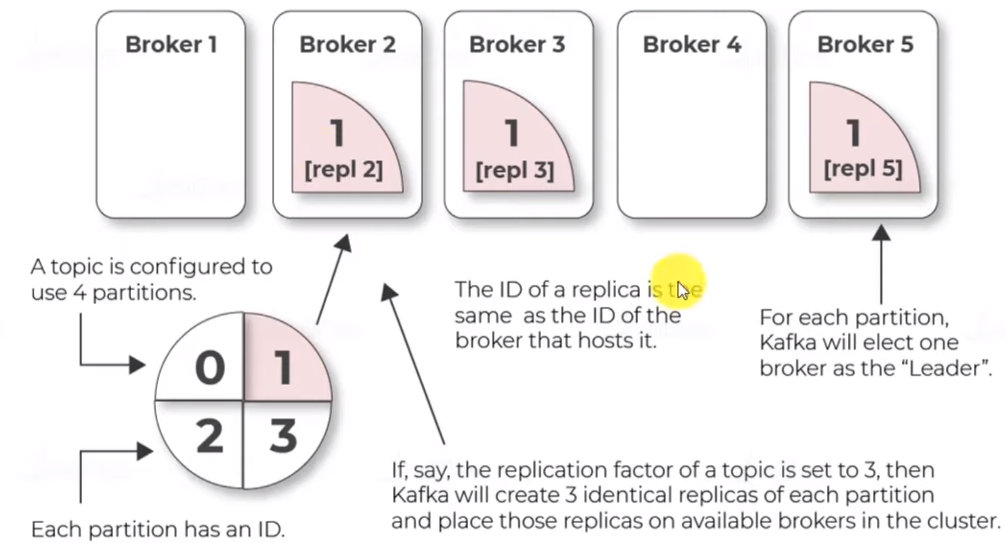
### Topic & Partitions



* Messages in Kafka are categorized into topics
* Topics are broken down into a number of partitions
* Messages are written to it in an appended-only fashion
* Reading messages can either be done in the order from begining to end or skip or rewind to any point in the partition by providing an offset value
* An offset value is the sequntial ID provided to messages
* Partitions provide redundancy and scalability
* Partitions can be hosted on a different server, i.e. a single topic can be scaled horizontally across multiple servers, thus enhancing the performance

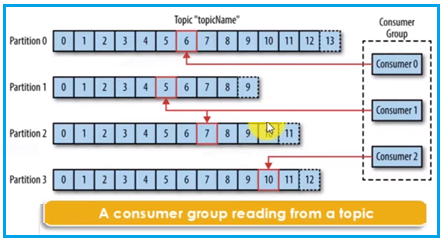


* The figure shows a topic with four partitions, with writes being appended to the end of each
* A record is stored on a partition either by the record key if the key is present or by round-robin if the key is missing (the default behaviour)



### Consumer

* The consumer can subscribe to one or more topics and read messages in the order they were produced
* The consumer keeps track of the messages it has already consumed by keeping the track of the offset of messages.
* Consumers work as part of a consumer group, i.e. One or more consumers that work together to consume a topics
* Messages with the same key arrive at the same consumer
* The group assures that each partition is consumed by only one memeber.

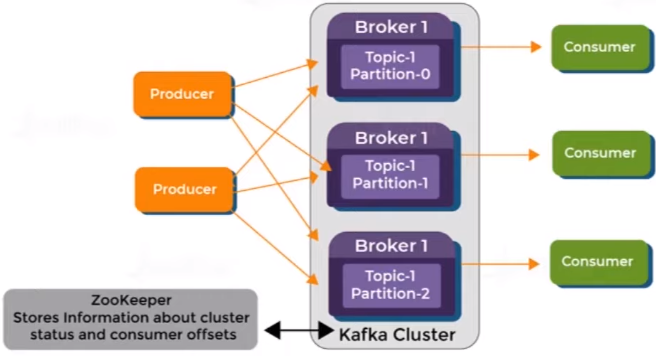


* Three consumers in a single group, consuming a topic
* Two consumers are working on one partition each, while the third consumer is working on two partitions.

### ZooKeeper

|  |
| --- |
| ZooKeeper is an open-source Apache project that provides centralized infrastructure and services that enable synchronization accross an Apache Hadoop cluster |

* Developed originally at Yahoo, ZooKeeper faciliatates synchronization in the process by maintaining a status on ZooKeeper servers, which store information in local log files.
* ZooKeeper servers are capable of supporting a large Hadoop cluster.
* Kafka brokers coordinate with each other using ZooKeeper.
* Producers and consumers are notified by the ZooKeeper service about the presence of a new broker in the system or about the failure of a broker in the system
* If the leader node fails, then on the basis of the currently live nodes, Apache ZooKeeper will elect the new leader.
* ZooKeeper in Kafka keeps a set of in-sync replicas.

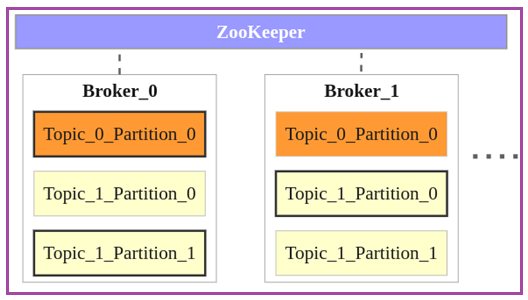


#### Role of ZooKeeper in Kafka

ZooKeeper is used to automatically select a leader for a partition. In case of any broker shutdown, an election is held, by ZooKeeper, for leader position of partitions (that went down with the broker). Also metadata like, in which broker a leader partition is living, etc., are held by ZooKeeper. Producers that stream data to topics, or Consumers that read stream data from topics, contact ZooKeeper for the nearest or less occupied broker.

A critical dependency of Apache Kafka is Apache Zookeeper, which is a distributed configuration and synchronization service. Zookeeper serves as the coordination interface between the Kafka brokers and consumers. The Kafka servers share information via a Zookeeper cluster. Kafka stores basic metadata in Zookeeper such as information about topics, brokers, consumer offsets (queue readers) and so on.

Since all the critical information is stored in the Zookeeper and it normally replicates this data across its ensemble, failure of Kafka broker / Zookeeper does not affect the state of the Kafka cluster. Kafka will restore the state, once the Zookeeper restarts. This gives zero downtime for Kafka. The leader election between the Kafka broker is also done by using Zookeeper in the event of leader failure.

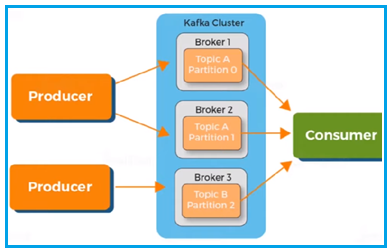


**Following points could be made from the above image:**

* There are two brokers, and many more can be added to the Kafka cluster.
* There are two topics Topic\_0 and Topic\_1.
* Topic\_0 has only one partition with replication factor of 2. The partition in Broker\_0 is the leader and the partition in Broker\_1 is the replica.
* Topic\_1 has two partitions with replication factor of 2. Partition\_0 in Broker\_1 and Partion\_1 in Broker\_0 are the leaders. Others are replicas.
* When a producer or consumers tries to connect to a topic, they connect to leaders with help from ZooKeeper.

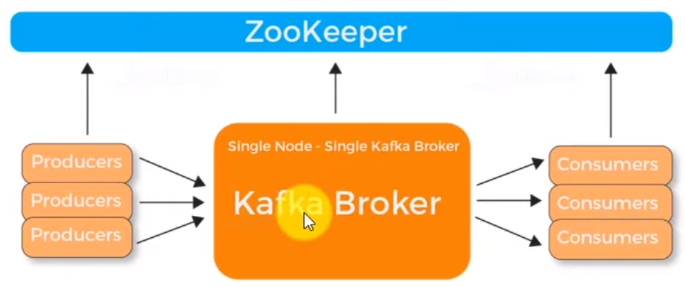
### Cluster

* A single Kafka server works well for the local development work.
* There are significant benefits of having multiple brokers configured as a cluster.
* The biggest benefit is the ability to scale the load across multiple servers.
* Relations help in performance maintenance of the Kafka cluster or the underlying systems.
* A Kafka cluster is effective for applications that involve large scale messsage processing.

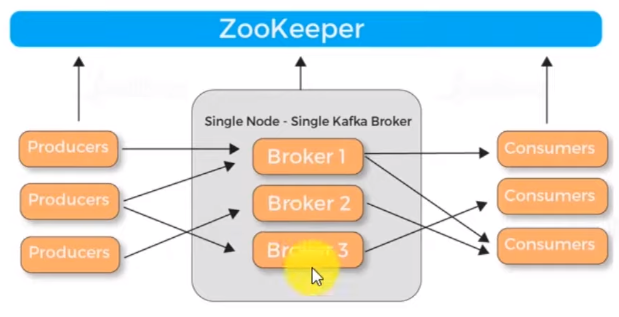


#### Cluster Types

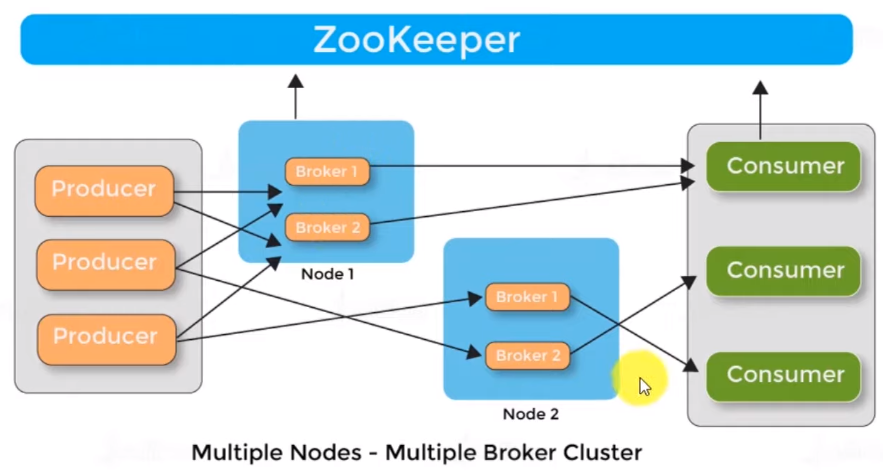
##### Single Node - Single Broker Cluster



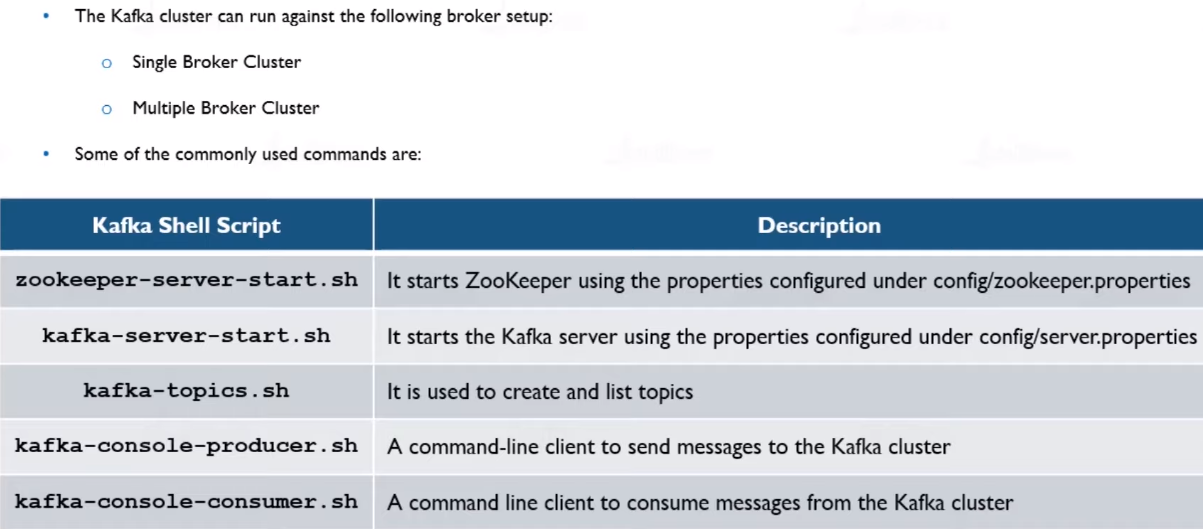
##### Single Node - Multiple Broker Cluster



##### Multiple Node - Multiple Broker Cluster



### Kafka Command Line Tools



Note: We need to use the .bat files in window instead of .sh files

## Work Flows

### Work Flow of Apache Kafka

Kafka is simply a collection of topics split into one or more partitions. A Kafka partition is a linearly ordered sequence of messages, where each message is identified by their index (called as offset). All the data in a Kafka cluster is the disjointed union of partitions. Incoming messages are written at the end of a partition and messages are sequentially read by consumers. Durability is provided by replicating messages to different brokers.

Kafka provides both **pub-sub** and **queue** based messaging system in a fast, reliable, persisted, fault-tolerance and zero downtime manner. In both cases, producers simply send the message to a topic and consumer can choose any one type of messaging system depending on their need.

Let us follow the steps in the next section to understand how the consumer can choose the messaging system of their choice.

#### Workflow of Pub/Sub Messaging

Following is the step wise workflow of the Pub-Sub Messaging

* Producers will send messages to a topic at regular intervals.
* Kafka broker stores all messages in the partitions configured for that particular topic. It ensures the messages are equally shared between partitions. If the producer sends two messages and there are two partitions, Kafka will store one message in the first partition and the second message in the second partition.
* A consumer always subscribe to a specific topic.
* When the consumer subscribes to a topic, Kafka provides the current offset of the topic to the consumer and also saves the offset in the Zookeeper ensemble.
* Consumer will request the Kafka in a regular interval (like 100 Ms) for new messages.
* Once Kafka receives the messages from producers, it forwards these messages to the consumers.
* On receiving the message, consumers will process it.
* Consumer will receive the message and process it.
* Once the messages are processed, consumer will send an acknowledgement to the Kafka broker.
* Once Kafka receives an acknowledgement, it changes the offset to the new value and updates it in the Zookeeper. Since offsets are maintained in the Zookeeper, the consumer can read next message correctly even during server outrages.
* This above flow will repeat until the consumer stops the request.
* Consumer has the option to rewind/skip to the desired offset of a topic at any time and read all the subsequent messages.

#### Workflow of Queue Messaging / Consumer Group

In a queue messaging system instead of a single consumer, a group of consumers having the same Group ID will subscribe to a topic. In simple terms, consumers subscribing to a topic with same Group ID are considered as a single group and the messages are shared among them. Let us check the actual workflow of this system.

* Producers send message to a topic in a regular interval.
* Kafka stores all messages in the partitions configured for that particular topic similar to the earlier scenario.
* A single consumer subscribes to a specific topic, assume Topic-01 with Group ID as Group-1.
* Kafka interacts with the consumer in the same way as Pub-Sub Messaging until new consumer subscribes the same topic, Topic-01 with the same Group ID as Group-1.
* Once the new consumer arrives, Kafka switches its operation to share mode and shares the data between the two consumers. This sharing will go on until the number of consumers reach the number of partition configured for that particular topic.
* Once the number of consumer exceeds the number of partitions, the new consumer will not receive any further message until any one of the existing consumer unsubscribes. This scenario arises because each consumer in Kafka will be assigned a minimum of one partition and once all the partitions are assigned to the existing consumers, the new consumers will have to wait.
* This feature is also called as Consumer Group. In the same way, Kafka will provide the best of both the systems in a very simple and efficient manner.

### Data flow in Kafka

Before getting in to the complete data flow, let us understand few key concepts in Kafka architecture.

* **Kafka Cluster** stores stream of messages where every message is identified by a unique id called "**Offset**"
* Set of messages are grouped by category. Each category is known as a “**Topic**” to which producers publish the messages
* Every **Kafka Topic** is ***partitioned and distributed*** across the brokers/servers in the cluster each of which is known as a “**Partition**”\*
* Each partition is a sequence of immutable (not editable) messages wherein new messages are appended to it as and when they arrive
* For reliability, Kafka replicates partitions across brokers in the cluster
* Kafka broker holding the main copy of a particular partition is considered as **Leader**, and zero or more brokers holding the replicas are considered as **Followers**
* The number of partitions per topic and the replication factor are configurable at the time of topic creation

\*Partitions are log files which store messages that are being published

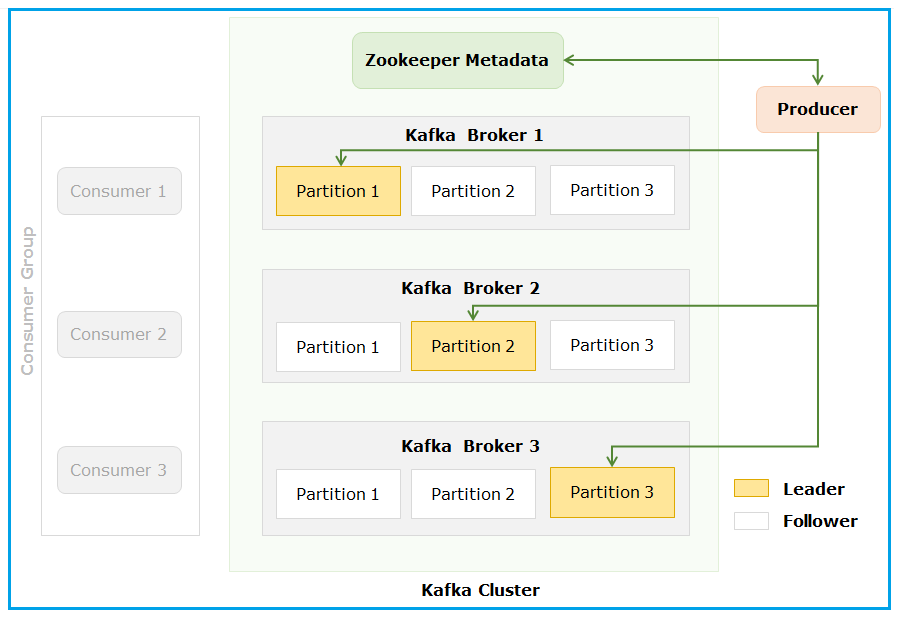
\*\*Replication factor - Number of back up copies for each partition

Below is the complete data flow diagram to understand how messages flow from producers to consumers through message brokers in a Kafka cluster.

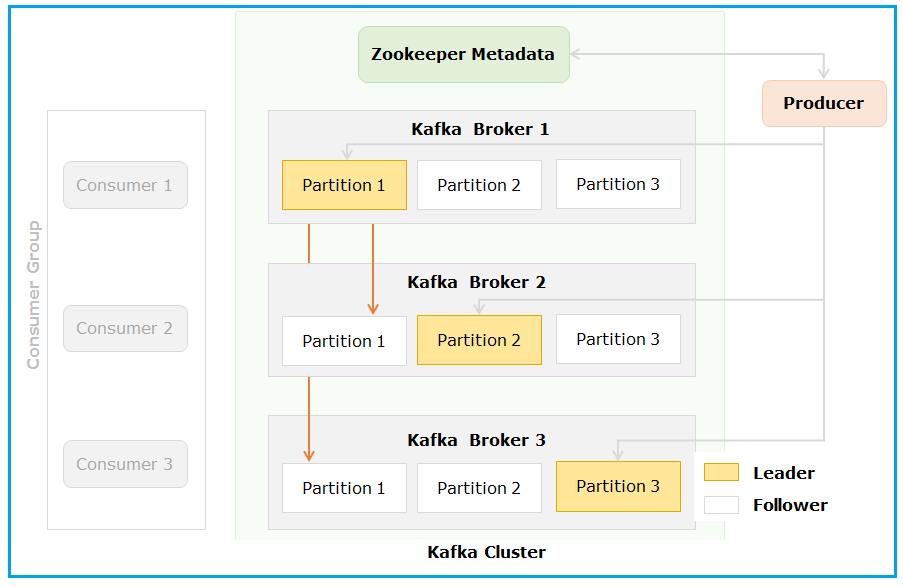
#### Step 1: At Producer side

* Producers get the required metadata (Broker, Topic and Partition) from Zookeeper that tells to which Leader partition the messages need to be published
* Producers then publish messages to the leader partition

Note: Messages can be read/written only from/to the leader partition

****

#### Step 2: At Kafka Broker side

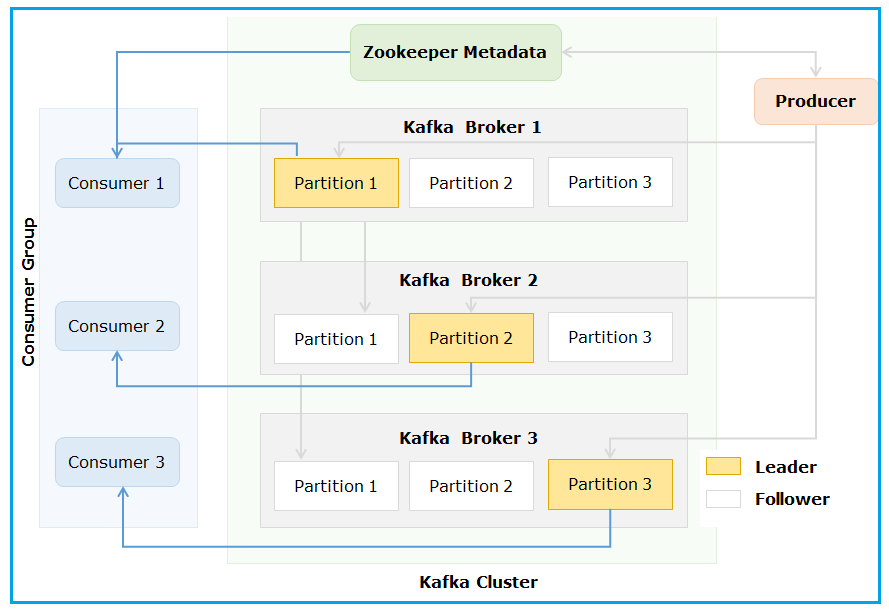


* Messages written to the leader (for e.g., Partition 1 on Broker 1) are passively replicated to zero or more followers (Partition 1 on Broker 2 and 3)
* In case the leader goes down, Zookeeper elects any one of the followers holding an In Sync Replica\* as the leader
* Producers continue to publish messages to the new leader ensuring zero downtime

\* In Sync Replica (ISR) is the replica that is continuously in sync with the leader

Note: Broker 2 contains the leader copy of Partition 2 while Broker 1 and 3 contain follower copies for this partition. Similarly, Broker 3 contains the leader copy of Partition 3 while Broker 1 and 2 contain follower copies.

#### Step 3: At Consumer side



* Consumers subscribe to specific topic as per the requirement and consume the messages. A topic may have zero or more subscribers

## Apache Avro

# Setup & Basic Commands

## Setup & Starting the Servers

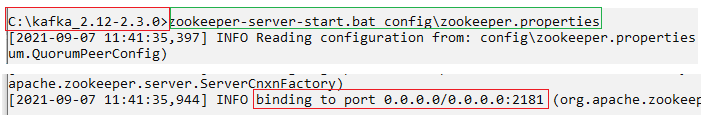
<https://www.javatpoint.com/installation-of-apache-kafka>

Note: After completed the setup from above portal we can start the servers.

### Starting ZooKeeper Server

* To start the ZooKeeper server use the following command (from the Kafka root folder).

zookeeper-server-start.bat config\zookeeper.properties

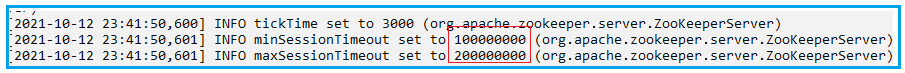


Note: The ZooKeeper Server is throwing timeout error. So for that add below properties to **zookeeper.properites** file inside config folder.

|  |
| --- |
| minSessionTimeout=100000000  maxSessionTimeout=200000000 |



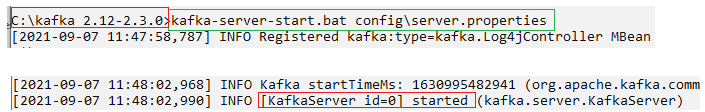
These properties are used by the ZooKeeper server when we start it. Observe the below logs.



### Starting the Kafka Server

To start the Kafka server use the following command (from the Kafka root folder).

kafka-server-start.bat config\server.properties



Note: It is mandatory to start the zookeeper server first to run the Kafka commands.

## Basic Commands in Kafka

Note: For windows we have to use .bat files instead .sh files.

### To create a topic using the command line tool kafka-topics.sh /.bat

* To create ‘DemoTopic’ with number of partitions as 1 and replication factor 1

|  |
| --- |
| kafka-topics.sh --create --topic DemoTopic --partitions 1 --replication-factor 1 --zookeeper localhost:2181 |

* To create ‘DemoTopic1’ with number of partitions as 2 and replication factor 2

|  |
| --- |
| kafka-topics.sh --create --topic DemoTopic1 --partitions 2 --replication-factor 2 --zookeeper localhost:2181 |

### To display all topics available in Kafka Cluster

|  |
| --- |
| kafka-topics.sh --list --zookeeper localhost:2181 |

### To display the details of a topic using the command line tool kafka-topics.sh

* Command to display the details of all the topics in Kafka Cluster

|  |
| --- |
| kafka-topics.sh --describe --zookeeper localhost:2181 |

* Command to display the details of a particular topic in Kafka Cluster

|  |
| --- |
| kafka-topics.sh --describe --zookeeper localhost:2181 --topic DemoTopic1 |

### To delete a topic from the Kafka Cluster

|  |
| --- |
| kafka-topics.sh --zookeeper localhost:2181 --delete --topic DemoTopic1 |

### To write messages into the topics using kafka-console-producer.sh

|  |
| --- |
| kafka-console-producer.sh --broker-list localhost:9092 --topic DemoTopic |

You may use CTRL+C to stop the console producer.

### To consume the messages you just produced, use kafka-console-consumer.sh

* To consume all the messages from the topic, open another terminal and execute

|  |
| --- |
| kafka-console-consumer.sh --zookeeper localhost:2181 --topic DemoTopic --from-beginning |

* To consume the latest messages from the topic, open a new terminal and execute

|  |
| --- |
| kafka-console-consumer.sh --zookeeper localhost:2181 --topic DemoTopic |

# Kafka Demos

## Quick Start Demo with CLI

<https://www.javatpoint.com/creating-kafka-topics>

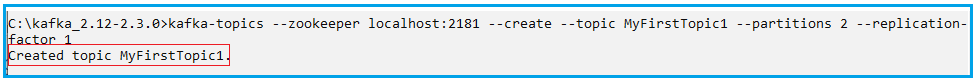
### Setup and starting the servers

Check the Setup section above.

### Create a topic

Create a topic using the below command

|  |
| --- |
| kafka-topics --zookeeper localhost:2181 --create --topic MyFirstTopic1 --partitions 2 --replication-factor 1 |



### Start a console Producer

Create a Producer with above created topic using the following command in separate terminal.

|  |
| --- |
| kafka-console-producer.bat --broker-list localhost:9092 --topic MyFirstTopic1 |



### Start a console consumer

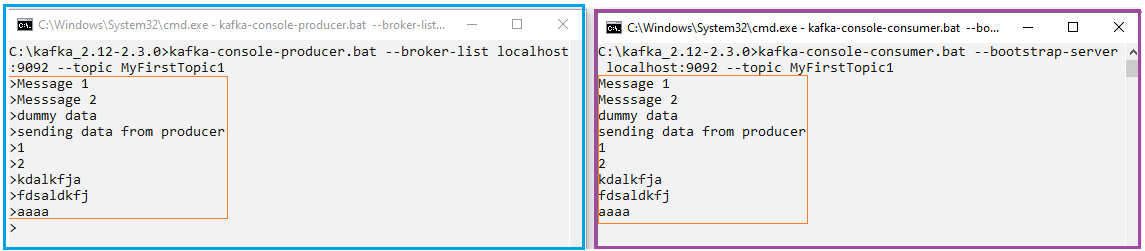
Create a Consumer with above created topic using the following command in separate terminal.

|  |
| --- |
| kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic MyFirstTopic1 |



### Send and reciev messages

Send some messages from Producer console and you can see those messages in Consumer console.



## Demo with Java Programming

### Simple Demo

#### Creating Maven project and adding the dependencies

Create a simple maven project.

To work with Kafka, we need to add the Kafka-client dependency

|  |
| --- |
| <dependency>  <groupId>org.apache.kafka</groupId>  <artifactId>kafka-clients</artifactId>  <version>2.3.0</version>  </dependency> |

Note: The dependency version should be same as the kafka server.

We also need to add the logging dependency for logging purpose- SLF4J, it is optional.

|  |
| --- |
| <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-simple</artifactId>  <version>1.7.30</version>  <!-- <scope>test</scope> -->  </dependency> |

#### Creating a Producer

Learn the concepts form here: <https://www.javatpoint.com/creating-kafka-producer-in-java>

|  |
| --- |
| **public** **class** MyKafkaProducer {  **private** **final** **static** String ***BOOTSTRAP\_SERVERS*** = "localhost:9092";  **private** **final** **static** String ***TOPIC\_NAME*** = "MyFirstTopic1";  // Method to create producer object  **public** **static** KafkaProducer<String, String> createProducer() {  // Create Configuration properties object  Properties properties = **new** Properties();  // Kafka broker topic details.  properties.setProperty(ProducerConfig.***BOOTSTRAP\_SERVERS\_CONFIG***, ***BOOTSTRAP\_SERVERS***);  properties.put(ProducerConfig.***CLIENT\_ID\_CONFIG***, "my-kafka-producer");  // Serialization class details  properties.setProperty(ProducerConfig.***KEY\_SERIALIZER\_CLASS\_CONFIG***, StringSerializer.**class**.getName());  properties.setProperty(ProducerConfig.***VALUE\_SERIALIZER\_CLASS\_CONFIG***, StringSerializer.**class**.getName());  // Creating a producer  KafkaProducer<String, String> kafkaProducer =  **new** KafkaProducer<String, String>(properties);  **return** kafkaProducer;  }  **public** **static** **void** main(String[] args) **throws** InterruptedException {  **final** KafkaProducer<String, String> kafkaProducer = *createProducer*();  // Sending single Message  // ProducerRecord<String, String> producerRecord =  new ProducerRecord<>(topic, "First Message from Java 123");  // kafkaProducer.send(producerRecord);  // kafkaProducer.flush();  // Sending Multiple Messages  **for** (**int** i = 1; i <= 10; i++) {  String key = "ID\_" + i;  String message = "Dummy Message\_" + i;  ProducerRecord<String, String> producerRecord =  **new** ProducerRecord<>(***TOPIC\_NAME***, key, message);  kafkaProducer.send(producerRecord);  kafkaProducer.flush();  Thread.*sleep*(5000);  }  kafkaProducer.close();  }  } |

#### Creating a Consumer

Learn the concepts form here: <https://www.javatpoint.com/creating-kafka-consumer-in-java>

|  |
| --- |
| **public** **class** MyKafkaConsumer {  **private** **final** **static** String ***BOOTSTRAP\_SERVERS*** = "localhost:9092";  **private** **final** **static** String ***TOPIC\_NAME*** = "MyFirstTopic1";  **private** **final** **static** String ***GROUP\_ID*** = "consumer-app";  **private** **final** **static** String ***OFFSET\_RESET\_CONFIG*** = "earliest";  **private** **static** KafkaConsumer<String, String> *kafkaConsumer* = **null**;  // Method to create consumer object  **public** **static** KafkaConsumer<String, String> createConsumer() {  // Create Configuration properties object  Properties properties = **new** Properties();  // Kafka broker topic details.  properties.setProperty(ConsumerConfig.***BOOTSTRAP\_SERVERS\_CONFIG***, ***BOOTSTRAP\_SERVERS***);  properties.setProperty(ConsumerConfig.***GROUP\_ID\_CONFIG***, ***GROUP\_ID***);  properties.setProperty(ConsumerConfig.***AUTO\_OFFSET\_RESET\_CONFIG***, ***OFFSET\_RESET\_CONFIG***);  // DeSerialization class details  properties.setProperty(ConsumerConfig.***KEY\_DESERIALIZER\_CLASS\_CONFIG***, StringDeserializer.**class**.getName());  properties.setProperty(ConsumerConfig.***VALUE\_DESERIALIZER\_CLASS\_CONFIG***, StringDeserializer.**class**.getName());  **return** **new** KafkaConsumer<>(properties);  }  **public** **static** **void** main(String[] args) {  *kafkaConsumer* = *createConsumer*();  *kafkaConsumer*.subscribe(Arrays.*asList*(***TOPIC\_NAME***));  System.***out***.println("-- Consumer App is started and waiting for the messages: ");  // polling  **while** (**true**) {  ConsumerRecords<String, String> records = *kafkaConsumer*.poll(Duration.*ofMillis*(100));  **for** (ConsumerRecord<String, String> record : records) {  System.***out***.println("\*\*\*\*\* New Message Arrived at: " + LocalTime.*now*());  System.***out***.println(" Key: " + record.key() + ", Value:" + record.value());  System.***out***.println(" Partition:" + record.partition() + ",Offset:" + record.offset());  System.***out***.println(" Total record: " + record);  }  }  }  @Override  **protected** **void** finalize() **throws** Throwable {  **if** (*kafkaConsumer* != **null**) {  *kafkaConsumer*.close();  }  **super**.finalize();  }  } |

#### Testing

Start the Consumer and then producer then check the console:

|  |
| --- |
| -- Consumer App is started and waiting for the messages:  \*\*\*\*\* New Message Arrived at: 17:51:47.050  Key: ID\_1, Value:Dummy Message\_1  Partition:1, Offset:70  Total record: ConsumerRecord(topic = MyFirstTopic1, partition = 1, leaderEpoch = 0, offset = 70, CreateTime = 1631017306959, serialized key size = 4, serialized value size = 15, headers = RecordHeaders(headers = [], isReadOnly = false), key = ID\_1, value = Dummy Message\_1)  \*\*\*\*\* New Message Arrived at: 17:51:52.028  Key: ID\_2, Value:Dummy Message\_2  Partition:1, Offset:71  \*\*\*\*\* New Message Arrived at: 17:51:57.044  Key: ID\_3, Value:Dummy Message\_3  Partition:0, Offset:64 |

### Kafka Producer Callbacks

<https://www.javatpoint.com/kafka-producer-callbacks>

If we want to know whether the producer sends data to the Kafka successfully or not, we can use this callback implementation.

Simply adding the Callbak() implementation in the send() method.

|  |
| --- |
| kafkaProducer.send(producerRecord, **new** Callback() {  @Override  **public** **void** onCompletion(RecordMetadata metadata, Exception e) {  **if** (e == **null**) {  System.***out***.println("\*\*\* Successfully received the details");  System.***out***.println(" Topic: " + metadata.topic() + "\n"  + " Partition: " + metadata.partition() + "\n"  + " Offset: " + metadata.offset());  } **else** {  System.***err***.println("Cannot produce, getting exception: " + e.getMessage());  }  }  });  O/P:  \*\*\* Successfully received the details  Topic: MyFirstTopic1  Partition: 1  Offset: 78  \*\*\* Successfully received the details  Topic: MyFirstTopic1  Partition: 1  Offset: 79 |

## Internal Working of Apache Kafka

<https://www.javainuse.com/messaging/kafka/internal>

# Fundamental Concepts

## Kafka Topics

## Kafka Partitioning

## Kafka Brokers

## Replication

## Client Applications

## Kafka Producers

## Kafka Consumers

## Kafka Streams

<https://www.javatpoint.com/kafka-stream-processing>

## Kafka Connectors

<https://www.javatpoint.com/kafka-connect>

## Multiple Clusters

### Single Node Clusters

### Multiple Node Clusters

## Kafka Monitoring

<https://www.javatpoint.com/kafka-monitoring>

## Kafka Security

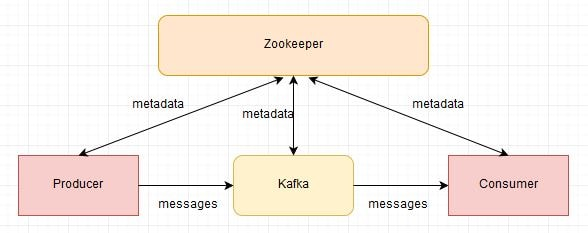
<https://www.javatpoint.com/kafka-security>

## Schema Registry

# Kafka Integrations

## Integration with Spring/Spring Boot

### Introduction



Kafka provides 5 core API’s to work with it: find [here](https://kafka.apache.org/documentation/#api).

1. Producer API
2. Consumer API
3. Streams API
4. Connect API
5. Admin API

* The **Producer API** allows applications to send streams of data to topics in the Kafka cluster.
* The **Consumer API** allows applications to read streams of data from topics in the Kafka cluster.
* The **Streams API** allows transforming streams of data from input topics to output topics.
* The **Connect API** allows implementing connectors that continually pull from some source system or application into Kafka or push from Kafka into some sink system or application.
* The **Admin API** allows managing and inspecting topics, brokers, and other Kafka objects.

**Spring-Kafka**

Spring provides good support for Kafka and provides the abstraction layers to work with over the native Kafka Java clients.

Spring provides a **KafkaTemplate** as a high-level abstraction for sending messages and **@KafkaListener** annotation & **KafkaMessageListenerContainer** to receive the messages at Consumer side.

* @EnableKafka
* KafkaTemplate
* KafkaMessageListenerContainer
* @KafkaListener
* KafkaTransactionManager
* spring-kafka-test

#### @EnableKafka

#### KafkaTemplate

#### KafkaMessageListenerContainer

#### @KafkaListener

#### KafkaTransactionManager

#### spring-kafka-test

### Working with Kafka in Spring

#### Dependency

To work with kafka, we need to add the spring-kafka dependency.

|  |
| --- |
| <dependency>  <groupId>org.springframework.kafka</groupId>  <artifactId>spring-kafka</artifactId>  </dependency> |

#### Key-Value Serializer & Deserializer

### Demos

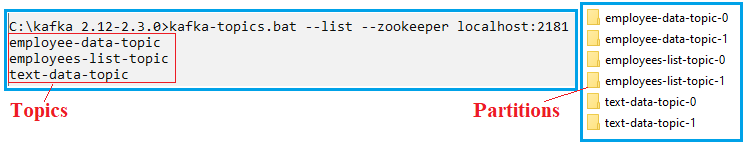
#### Kafka-Spring Producers/Consumers Application

<https://www.baeldung.com/spring-kafka>

<https://docs.spring.io/spring-kafka/docs/current/reference/html/#configuring-topics>

<https://www.confluent.io/blog/apache-kafka-spring-boot-application/>

Whenever the run the producer application, all the configured topics are created with partitions.



<https://github.com/anilkumarb1248/Kafka-Demos>

#### Sample Demo Application (Producer & Consumer)

##### **Prerequisites**

Before starting, install and start the Kafka and Zookeeper servers.

Create the topic with name: **employee-data**

|  |
| --- |
| kafka-topics --zookeeper localhost:2181 --create --topic **employee-data** --partitions 2 --replication-factor 1 |

##### Producer

<https://www.youtube.com/watch?v=NjHYWEV_E_o>

###### Create a spring boot starter project by adding web and Kafka modules.

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.kafka</groupId>  <artifactId>spring-kafka</artifactId>  </dependency> |

###### Create Employee model class

|  |
| --- |
| **public** **class** Employee {  **private** Long id;  **private** String name;  **private** Double salary;  } |

###### Create a simple controller

|  |
| --- |
| @RestController  @RequestMapping("/producer")  **public** **class** KafkaController {  @Autowired  KafkaSender kafkaSender;  @GetMapping("/send/{name}")  **public** String createEmployee(@PathVariable("name") String name) {  Employee employee = **new** Employee(**new** Random().nextLong(), name,  **new** Random().nextDouble());  **return** kafkaSender.sendToKafka(employee);  }  } |

###### Create a Kafka configuration file

|  |
| --- |
| @Configuration  **public** **class** KafkaConfiguration {  **private** **final** **static** String ***BOOTSTRAP\_SERVERS*** = "localhost:9092";  @Bean  **public** KafkaTemplate<String, Employee> kafkaTemplate() {  **return** **new** KafkaTemplate<>(producerFactory());  }  @Bean  **public** ProducerFactory<String, Employee> producerFactory() {  Map<String, Object> config = **new** HashMap<>();  config.put(ProducerConfig.***BOOTSTRAP\_SERVERS\_CONFIG***, ***BOOTSTRAP\_SERVERS***);  config.put(ProducerConfig.***KEY\_SERIALIZER\_CLASS\_CONFIG***, StringSerializer.**class**);  config.put(ProducerConfig.***VALUE\_SERIALIZER\_CLASS\_CONFIG***, JsonSerializer.**class**);  **return** **new** DefaultKafkaProducerFactory<>(config);  }  } |

Note: By default Kafka serializes the String values, but if we want serialize the custom objects like Employee, we need to use “JsonSerializer”.

###### Create a Kafka sender

|  |
| --- |
| @Component  **public** **class** KafkaSender {  @Autowired  KafkaTemplate<String, Employee> kafkaTemplate;  **private** **static** **final** String ***TOPIC\_NAME*** = "employee-data";  **public** String sendToKafka(Employee employee) {  String response = "Employee details send to Kafka successfully";  kafkaTemplate.send(***TOPIC\_NAME***, employee);  **return** response;  }  } |

###### Add the properties to application.properties file

|  |
| --- |
| server.port=2021  server.servlet.context-path=/KafkaProducer |

###### Testing

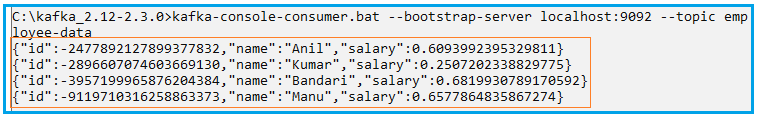
Start the consumer console by running the below command.

|  |
| --- |
| kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic employee-data |

Hit the below URL with multiple times with different names:

|  |
| --- |
| http://localhost:2021/KafkaProducer/producer/send/Anil |

Check all the messages are receiving at Consumer console:



##### Consumer

<https://www.youtube.com/watch?v=IncG0_XSSBg>

###### Create topics

|  |
| --- |
| kafka-console-producer.bat --broker-list localhost:9092 --topic employee-data  kafka-console-producer.bat --broker-list localhost:9092 --topic string-messages |

###### Create a spring boot starter project by adding web and Kafka modules.

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.kafka</groupId>  <artifactId>spring-kafka</artifactId>  </dependency> |

###### Create Employee model class

|  |
| --- |
| **public** **class** Employee {  **private** Long id;  **private** String name;  **private** Double salary;  } |

###### Create a configuration file

|  |
| --- |
| @Configuration  **@EnableKafka**  **public** **class** KafkaConsumerConfiguration {  **private** **final** **static** String BOOTSTRAP\_SERVERS = "localhost:9092";  @Bean  **public** ConsumerFactory<String, String> **consumerFactory**() {  Map<String, Object> config = **new** HashMap<>();  config.put(ConsumerConfig.BOOTSTRAP\_SERVERS\_CONFIG, BOOTSTRAP\_SERVERS);  config.put(ConsumerConfig.GROUP\_ID\_CONFIG, "string-values");  config.put(ConsumerConfig.KEY\_DESERIALIZER\_CLASS\_CONFIG, StringDeserializer.**class**);  config.put(ConsumerConfig.VALUE\_DESERIALIZER\_CLASS\_CONFIG, StringDeserializer.**class**);  **return** **new** DefaultKafkaConsumerFactory<>(config);  }  **//String Deserializer container**  @Bean  **public** ConcurrentKafkaListenerContainerFactory<String, String> kafkaListenerContainerFactory() {  ConcurrentKafkaListenerContainerFactory<String, String> factory = **new** ConcurrentKafkaListenerContainerFactory<>();  factory.setConsumerFactory(**consumerFactory**());  **return** factory;  }  @Bean  **public** ConsumerFactory<String, Employee> employeeConsumerFactory() {  Map<String, Object> config = **new** HashMap<>();  config.put(ConsumerConfig.BOOTSTRAP\_SERVERS\_CONFIG, BOOTSTRAP\_SERVERS);  config.put(ConsumerConfig.GROUP\_ID\_CONFIG, "json-values");  config.put(ConsumerConfig.KEY\_DESERIALIZER\_CLASS\_CONFIG, StringDeserializer.**class**);  config.put(ConsumerConfig.VALUE\_DESERIALIZER\_CLASS\_CONFIG, JsonDeserializer.**class**);  **return** **new** DefaultKafkaConsumerFactory<String, Employee>(config, **new** StringDeserializer(),  **new** JsonDeserializer<Employee>(Employee.**class**));  }  **//Employee Deserializer container**  @Bean  **public** ConcurrentKafkaListenerContainerFactory<String, Employee> employeeKafkaListenerContainerFactory() {  ConcurrentKafkaListenerContainerFactory<String, Employee> factory =  **new** ConcurrentKafkaListenerContainerFactory<>();  factory.setConsumerFactory(employeeConsumerFactory());  **return** factory;  }  } |

|  |
| --- |
| **return** **new** DefaultKafkaConsumerFactory<String, Employee>(config, **new** StringDeserializer(),  **new** JsonDeserializer<Employee>(Employee.**class**)); |

If we don’t specify this we will get exception at consumer side

|  |
| --- |
| 2021-10-13 23:01:58.240 ERROR 8364 --- [ntainer#2-0-C-1] MessageListenerContainer$ListenerConsumer : Consumer exception  java.lang.IllegalStateException: This error handler cannot process 'SerializationException's directly; please consider configuring an 'ErrorHandlingDeserializer' in the value and/or key deserializer |

###### Create a Listener class

|  |
| --- |
| @Service  **public** **class** KafkaConsumer {  @KafkaListener(topics = "string-messages", groupId = "string-values")  **public** **void** consumeStringMessages(String message) {  System.***out***.println("Consumed String Message: " + message);  }  @KafkaListener(topics = "employee-data", groupId="json-values",  containerFactory = "employeeKafkaListenerContainerFactory")  **public** **void** consumeEmployeesData(Employee employee) {  System.***out***.println("Consumed Employee details: "+ employee);  }  } |

Here we have created two listeners, one is listen to “string-messages” topic and another one listen “employee-data” topic.

Note: We need to add the **@EnableKafka** annotation in configuration class to enable Spring to scan the **@KafkaListener** annotation methods.

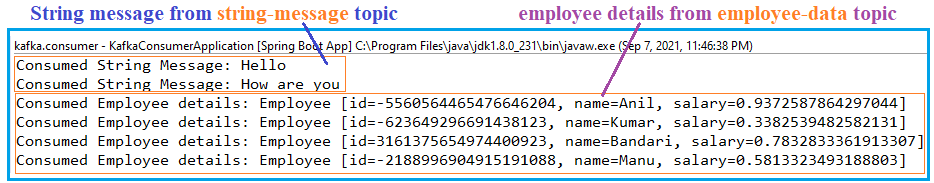
###### Testing

Open a producer console to send the string type messages

|  |
| --- |
| kafka-console-producer.bat --broker-list localhost:9092 --topic string-messages |

Send the employee details to Kafka from the producer application.

Observe the Consumer application console, all the messages and details are received.



## Send Large Messages with Kafka

<https://www.baeldung.com/java-kafka-send-large-message>

Kafka configuration limits the size of messages that it's allowed to send. By default, this limit is 1MB. However, if there's a requirement to send large messages, we need to tweak these configurations as per our requirements.

If we try to send more than 1 MB message, we will get “**MessageSizeTooLargeException**”

The idea is to have equal size of message being sent from Kafka Producer to Kafka Broker and then received by Kafka Consumer i.e.

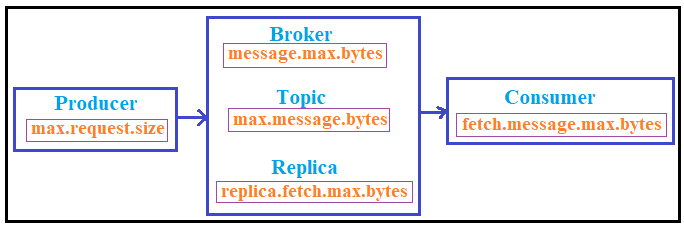
Kafka producer --> Kafka Broker & Topic --> Kafka Consumer

Suppose if the requirement is to send 20MB of message, then the Producer, the Broker, Topic (Leader & Follower) and the Consumer, all needs to be in sync.

Kafka Producer sends 20 MB --> Kafka Broker Allows/Stores 20 MB --> Kafka Consumer receives 20 MB and the Topic, Leader and Followers alos in sync.

* 20 MB = 20971520 Bytes

**The setting therefore should be:**



* **On Producer**
* max.request.size = 20971520

*max.request.size* – this is largest size of a message that can send by a producer.

* **On Topic**
* max.message.bytes=20971520

*max.message.bytes* - this is the largest size of the message the broker will allow to be appended to the topic. This size is validated pre-compression. (Defaults to broker's message.max.bytes.)

* **On Broker & Follower**
* message.max.bytes=20971520
* replica.fetch.max.bytes=20971520

*message.max.bytes* - this is the largest size of the message that can be received by the broker from a producer.

*replica.fetch.max.bytes* - this will allow for the replicas in the brokers to send messages within the cluster and make sure the messages are replicated correctly. If this is too small, then the message will never be replicated, and therefore, the consumer will never see the message because the message will never be committed (fully replicated).

* **On Consumer**
* fetch.message.max.bytes=20971520
* max.partition.fetch.bytes=20971520
* fetch.max.bytes=20971520

*fetch.message.max.bytes* - this will determine the largest size of a message that can be fetched by the consumer.

Increase *max.partition.fetch.bytes* to receive larger messages

Note: If you are increasing the properties ***message.max.bytes*** and ***replica.fetch.max.bytes***, then the ***message.max.bytes*** has to be equal or smaller than ***replica.fetch.max.bytes.***

"***replica.fetch.max.bytes*** (default: 1MB) – Maximum size of data that a broker can replicate. This has to be larger than ***message.max.bytes***, or a broker will accept messages and fail to replicate them. Leading to potential data loss.

## Integration with Spring Cloud Stream

### Introduction

### Demos

#### Domains Filter Example

<https://www.youtube.com/watch?v=rqjdSbIOrJ4>

#### Employee Filter Example

Need to work on it

|  |
| --- |
| Employee list Producer -> Employees Processor based on some condtion -> Emplioyee list Consumer |

# Kafka-Spring Integration Errors

## My Practice Errors

### Error 1:

|  |
| --- |
| 2021-10-13 23:01:58.240 ERROR 8364 --- [ntainer#2-0-C-1] MessageListenerContainer$ListenerConsumer : Consumer exception  java.lang.IllegalStateException: This error handler cannot process 'SerializationException's directly; please consider configuring an 'ErrorHandlingDeserializer' in the value and/or key deserializer |

While deserializing the string key-values we can use like this.

|  |
| --- |
| return new DefaultKafkaConsumerFactory<>(config); |

But if we are trying to deserialize any Json objects, if we specify like above we will get the above error.

By default Kafka serializes the String values, but if we want serialize the custom objects like Employee, we need to use “JsonSerializer”.

And we have to create the “DefaultKafkaConsumerFactory” as below:

|  |
| --- |
| return new DefaultKafkaConsumerFactory<String, Employee>(config, new StringDeserializer(),  new JsonDeserializer<Employee>(Employee.class)); |

### Error 2:

|  |
| --- |
| Caused by: org.apache.kafka.common.errors.SerializationException: Error deserializing key/value for partition employee-data-topic-7 at offset 0. If needed, please seek past the record to continue consumption.  Caused by: java.lang.IllegalArgumentException: The class 'app.kafka.producer.model.Employee' is not in the trusted packages: [java.util, java.lang, app.kafka.consumer.model, app.kafka.consumer.model.\*]. If you believe this class is safe to deserialize, please provide its name. If the serialization is only done by a trusted source, you can also enable trust all (\*).  at org |

<https://stackoverflow.com/questions/51688924/spring-kafka-the-class-is-not-in-the-trusted-packages>

### Error 3

|  |
| --- |
| Caused by: org.apache.kafka.common.errors.SerializationException: Error deserializing key/value for partition employee-data-topic-5 at offset 0. If needed, please seek past the record to continue consumption.  Caused by: org.springframework.messaging.converter.MessageConversionException: failed to resolve class name. Class not found [app.kafka.producer.model.Employee]; nested exception is java.lang.ClassNotFoundException: **app.kafka.producer.model.Employee**  at org.springframework.kafka.support.converter.DefaultJackson2JavaTypeMapper.getClassIdType(DefaultJackson2JavaTypeMapper.java:142) ~[spring-kafka-2.7.7.jar:2.7.7] |

Consumer side also it is expecting same class with in the same package.

So I have Placed the Employee class in same package name in producer and consumer.

### Error 4

|  |
| --- |
| [Producer clientId=producer-1] Got error produce response with correlation id 6 on topic-partition text-data-topic-1, retrying (2147483646 attempts left). Error: CORRUPT\_MESSAGE |

|  |
| --- |
| [Producer clientId=producer-1] Got error produce response with correlation id 6 on topic-partition employees-list-topic-0, retrying (2147483646 attempts left). Error: CORRUPT\_MESSAGE |

I have changed the “replicas = 3”, then the error gone. Previously it was 2

|  |
| --- |
| @Bean  **public** NewTopic textDataTopic() {  **return** TopicBuilder  .*name*("text-data-topic")  .partitions(10)  .replicas(3)  .compact()  .build();  } |

### Error 5

|  |
| --- |
| [Producer clientId=producer-1] Error while fetching metadata with correlation id 9 : {text-data-topic=LEADER\_NOT\_AVAILABLE} |