What is Apache Kafka

Apache Kafka is a software platform which is based on a distributed streaming process. It is a publish-subscribe messaging system which let exchanging of data between applications, servers, and processors as well. Apache Kafka was originally developed by **LinkedIn**, and later it was donated to the Apache Software Foundation. Currently, it is maintained by **Confluent** under Apache Software Foundation. Apache Kafka has resolved the lethargic trouble of data communication between a sender and a receiver.

What is a messaging system

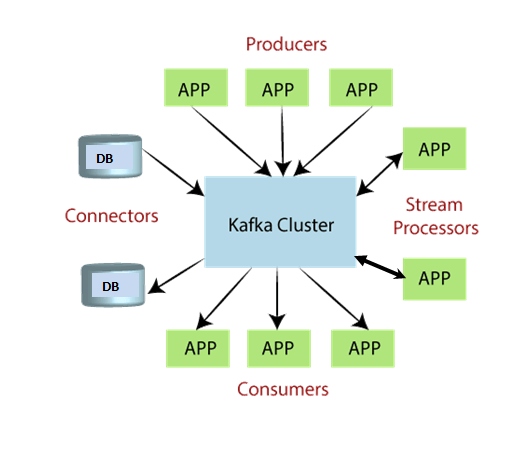
A messaging system is a simple exchange of messages between two or more persons, devices, etc. A publish-subscribe messaging system allows a sender to send/write the message and a receiver to read that message. In Apache Kafka, a sender is known as a **producer** who publishes messages, and a receiver is known as a **consumer** who consumes that message by subscribing it.

AD

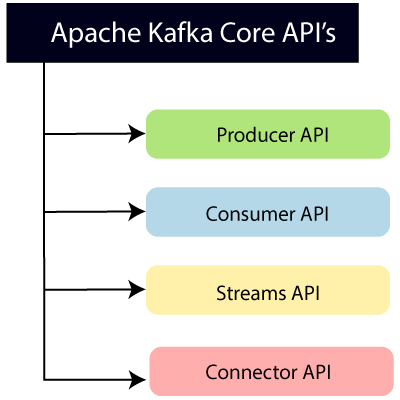
What is Streaming process

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.



To learn and understand Apache Kafka, the aspirants should know the following four core APIs :



**Producer API:** This API allows/permits an application to publish streams of records to one or more topics. (discussed in later section)

**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.

Why Apache Kafka



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

1. Apache Kafka is capable of handling millions of data or messages per second.
2. 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.
3. 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.
4. Apache Kafka has a resilient architecture which has resolved unusual complications in data sharing.
5. Organizations such as NETFLIX, UBER, Walmart, etc. and over thousands of such firms make use of Apache Kafka.
6. 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.
7. Learning Kafka is a good source of income. So, those who wish to raise their income in future in IT sector can learn.

**Prerequisite**

The aspirants should have basic knowledge of Java programming, and some knowledge of Linux commands.

**Audience**

This Apache Kafka journey is designed for beginners, developers, and people who wish to learn new things.

**Problems**

We assure that you will not find any problem in this Apache KafkaTutorial. But if there is any mistake, please post the problem in a contact form.

At a high level, Kafka possesses three important capabilities:

* It enables applications to publish or subscribe to data or event streams.
* Ensures accurate storage of records, in a fault-tolerant and durable manner.
* Real-time processing of records.

We will deep dive into more concepts in coming sections.

**2. Kafka vs. Traditional Messaging Systems**

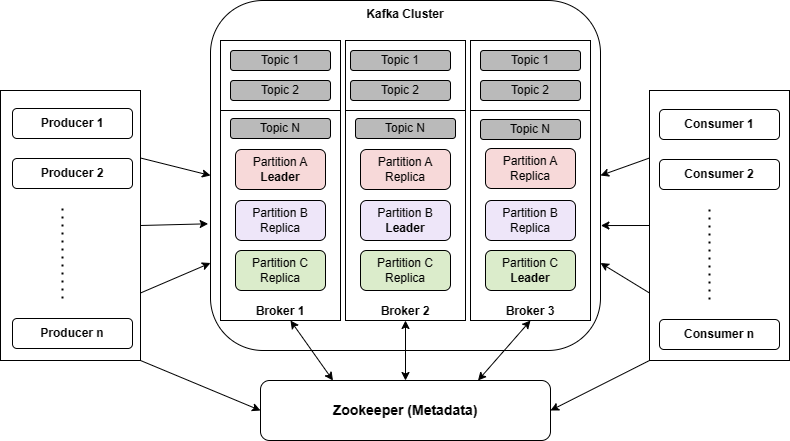
When compared with counterparts, Apache Kafka provides major upgrades from the traditional messaging system. Below are the **differences between a traditional messaging system (like *RabbitMQ*, *ActiveMQ*, etc.) and Kafka**.

| **Feature** | **Traditional Messaging System** | **Kafka Streaming Platform** |
| --- | --- | --- |
| Message Persistence | The broker is responsible for keeping track of consumed messages and removing them when messages are read. | Messages are typically retained in Kafka topics for a configurable period of time, even after they have been consumed. Kafka offers message persistence, ensuring data durability. |
| Scalability | Not a distributed system, so it is not possible to scale horizontally. | It is a distributed streaming system, so by adding more partitions, we can scale horizontally. |
| Data Model | Primarily point-to-point (queues/topics) messaging model. | Built around a publish-subscribe (logs) model, which enables multiple consumers to subscribe to topics and process data concurrently. |
| Ordering of Messages | Message ordering can be guaranteed within a single queue or topic but may not be guaranteed across different queues or topics. | Kafka maintains message order within a partition, ensuring that messages within a partition are processed in the order they were received. |
| Message Replay | Limited or no built-in support for message replay. Once consumed, messages may be lost unless custom solutions are implemented. | Supports message replay from a specified offset, allowing consumers to reprocess past data, which is valuable for debugging, analytics, and data reprocessing. |
| Use Cases | Typically used for traditional enterprise messaging, remote procedure calls (RPC), and task queues. | Well-suited for real-time analytics, log aggregation, event sourcing, and building data-intensive, real-time applications. |

**3. Apache Kafka Architecture**

When it comes to designing the architecture of Apache Kafka, various applications employ different approaches. However, there are several essential components that are necessary to create a robust Kafka architecture like **Kafka Cluster**, **Producers**, **Consumers**, **Brokers**, **Topics**, **Partitions**, and **Zookeeper**.

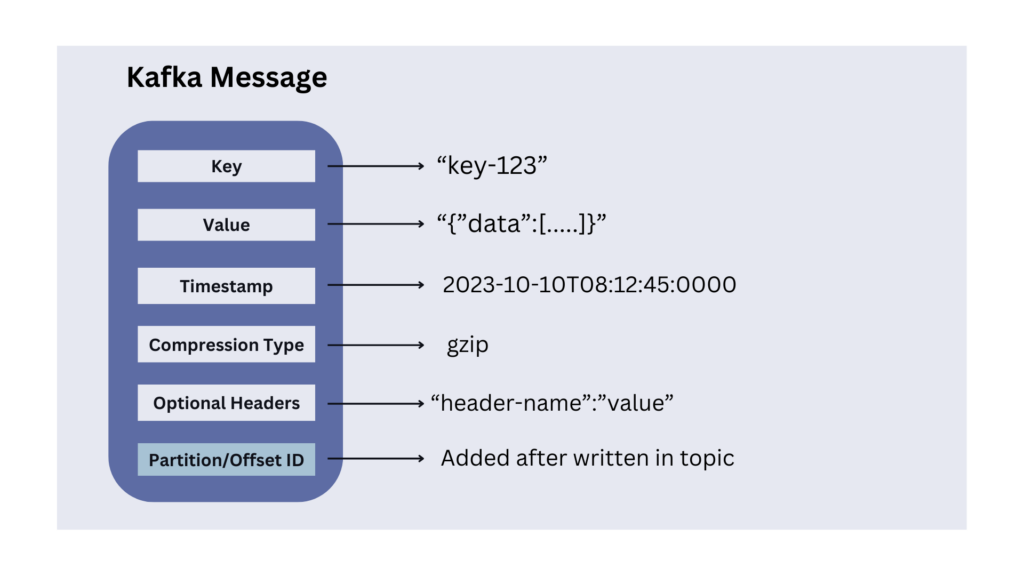
Here’s an overview of the main components and their roles in Kafka’s architecture:



The above diagram represents Kafka’s architecture. Let’s discuss each component in detail.

**3.1. Message**

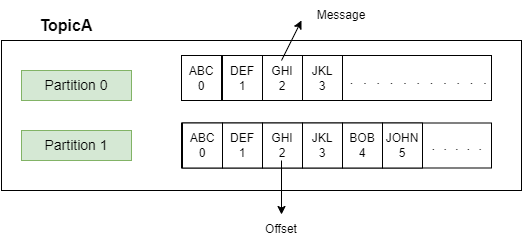
A message is a **primary unit of data** within Kafka. Messages sent from a producer consist of the following parts:



* **Message Key (*optional*)**: Keys are commonly used when ordering or grouping related data. For example, in a log processing system, we can use the user ID as the key to ensure that all log messages for a specific user are processed in the order they were generated.
* **Message Value**: It contains the actual data we want to transmit. Kafka does not interpret the content of the value. It is received and sent as it is. It can be XML, JSON, String, or anything. Kafka does not care and stores everything. Many Kafka developers favor using [Apache Avro](https://howtodoinjava.com/kafka/kafka-with-avro-and-schema-registry/), a serialization framework initially developed for Hadoop.
* **Timestamp (Optional)**: We can include an optional timestamp in a message indicating its creation timestamp. It is useful for tracking when events occurred, especially in scenarios where event time is important.
* **Compression Type (Optional)**: Kafka messages are generally small in size, and sent in a standard data format, such as JSON, Avro, or Protobuf. Additionally, we can further compress them into gzip, lz4, snappy or zstd formats.
* **Headers (Optional)**: Kafka allows adding headers that may contain additional meta-information related to the message.
* **Partition and Offset Id**: Once a message is sent into a Kafka topic, it also receives a partition number and offset id that is stored within the message.

**3.2. Topic and Partition**

The topics and partitions play a crucial role in organizing and distributing messages across the cluster.



**A Kafka topic is a logical name or category to which records (messages) are published by producers and from which records are consumed by consumers.** We can think of them as event streams where data is written and read.

Topics are created on the Kafka broker and can have one or more partitions.

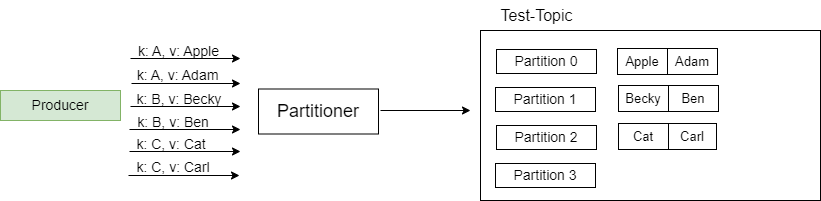
**A partition is the smallest storage unit where the messages live inside the topic**. The partitions have a significant effect on scalable message consumption. Each partition is an ordered and immutable sequence of records; meaning once a message is stored, it cannot be changed.

A random number is assigned to each record in a partition called *offset*. The **offset represents the position of the last consumed message in each partition**. Each partition works independently of each other.

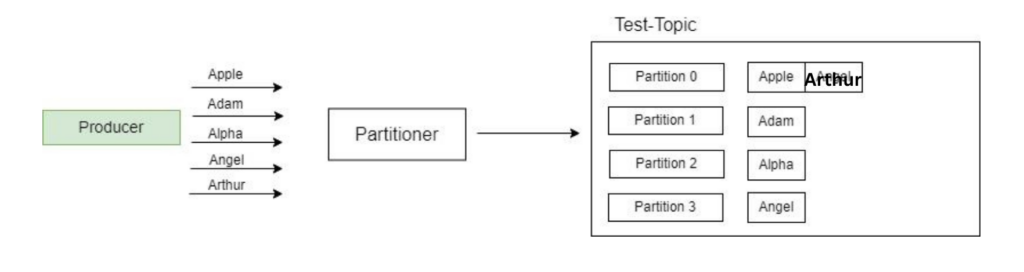
Message ordering is guaranteed only at the partition level. If message ordering is required then we must make sure to publish the records to the same partition.

**3.3. Producer**

**A producer publishes the messages using the topic name**. The user is not required to specify the broker and the partition. By default, Kafka uses the message *key* to select the topic partition by *DefaultPartitioner* which uses a [32-bit murmur2 hash](https://github.com/aappleby/smhasher).

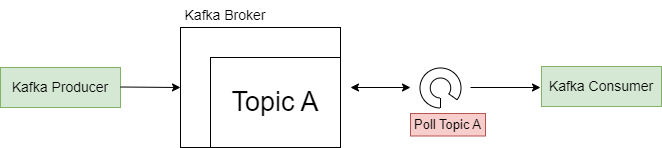
Messages with the same key go into the same partition

Remember that the message key is optional, so **if there is no key provided, then Kafka will partition the data in a round-robin fashion.**



**3.4. Consumer, Consumer Group and Consumer Offset**

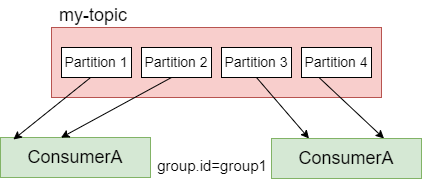
**A consumer reads messages from the Kafka cluster using a topic name**. It continuously polls the Kafka broker using the topic name for new messages. Once the polling loop notices a new message, the message is consumed by the consumer and processing is done on the retrieved message.

Consumer polls the broker at intervals

**Consumer Group**

Every consumer is always part of a consumer group. **A consumer group represents a group of consumer instances that work together to consume messages from the specified topic(s).**

When a topic is created in Kafka, it can have one or more consumer groups associated with it. The consumer groups maintain the offset information for the partitions they consume.



When messages are published to a topic, they are distributed across the partitions in a configurable manner. Each consumer within a consumer group is assigned one or more partitions to read from. **Each partition is consumed by only one consumer within a consumer group at a time. This ensures that all messages within a partition are processed in the order they were received.**

To decide which consumer should read data first and from which partition, consumers within a group use ***GroupCoordinator*** and ***ConsumerCoordinator***, which assign a consumer to a partition, managed by Kafka Broker.

**Consumer Offset**

Typically, Kafka consumers have 3 options to read the message from the partition:

* **from-beginning** – Always reads data from the start of the partition.
* **latest**– reads the messages produced after the consumer started.
* **specific offset** – reads the messages using a specific offset value.

Consumer offset represents the position or *offset* within a partition from which a consumer group has consumed messages. In other words, each consumer group maintains its offset for each partition it consumes. The offset helps in determining the next message to read from a specified partition inside the topic.

As soon as a consumer reads the message, Kafka automatically increments the offset value and commits the offsets, in a topic known as *\_\_consumer\_offsets*. This helps in case of machine failures. Consumer offsets behave like a bookmark for the consumer to start reading the messages from the point it left off.

**3.5. Broker**

**A Kafka broker is a single server instance that stores and manages the partitions**. Brokers act as a bridge between consumers and producers.

Kafka brokers store data in a directory on the server disk they run on. Each topic partition receives its own sub-directory with the associated name of the topic.

**3.6. Cluster and Bootstrap Server**

**A Kafka cluster consists of multiple Kafka brokers working together**. Some clusters may contain just one broker or others may include three or potentially hundreds of brokers. Companies like *Netflix* and *Uber* run hundreds and thousands of Kafka brokers to handle their data.

A client that wants to send or receive messages through the Kafka cluster may connect to any broker in the cluster. Each broker in the cluster has metadata about all the other brokers, and therefore **any broker in the cluster can act as a bootstrap server (the initial connection point used by Kafka clients to connect to the cluster)**.

The client connects to the provided broker (*bootstrap server*) and requests metadata about the Kafka cluster, such as the addresses of all the other brokers, the available topics, and the partition information. Once the client has obtained the metadata from the bootstrap server, it can establish connections to other brokers in the cluster as needed for producing or consuming messages.

**3.7. Zookeeper**

Zookeeper handles metadata management in the Kafka world. It

* keeps track of which brokers are part of the Kafka cluster.
* determines which broker is the leader of a given partition and topic and performs leader elections.
* stores topics’ configurations and permissions.
* sends notifications to Kafka in case of changes (e.g., new topic, broker dies, broker comes up, delete a topic, etc….).

**4. Message Replication in Kafka**

When we send a message to a Kafka topic, it is stored in a partition. By default, each topic has only a single partition. However, each topic can have multiple partitions if configured.

Suppose, if we store the data in only one partition, and if the broker goes down then there might be a data loss problem. **To avoid data loss issues, Kafka uses replication**.

Let us discuss this in more detail.

**4.1. Partition Leader and Followers**

In a distributed Kafka setup, a partition can be replicated across multiple brokers in the cluster to provide fault tolerance and high availability.

One broker is marked leader and other brokers are called followers for a specific partition. This designated broker assumes the role of the leader for the topic partition. On the other hand, any additional broker that keeps track of the leader partition is called a follower and it stores replicated data for that partition.

**Note that the leader receives and serves all incoming messages from producers and serves them to consumers. Followers do not serve read or write requests directly from producers or consumers.** Followers just act as backups and can take over as the leader in case the current leader fails.

Therefore, each partition has one leader and multiple followers.

**4.2. Replication-Factor**

While creating a topic, we provide a replication-factor value. A r**eplication factor of 1 means no replication**, mostly used for development purposes and should be avoided in test and production Kafka clusters. A **replication factor 3 is commonly used** as it provides the right balance between broker loss and replication overhead.

In the cluster below consisting of three brokers, the replication factor is 2. Let’s say a producer produces a message to *Partition 0,* it goes to the leader partition. Upon receiving the message, *Broker1*proceeds to store it persistently within the file system. Since we have replicator factor = 2, we need one more copy of the message. Now the follower replica in another broker receives a copy of the same message and stores it in the filesystem.



**4.3. In-Sync Replicas (ISR)**

When a partition is replicated across multiple brokers, not all replicas are necessarily in sync with the leader at all times. The in-sync replicas represent the number of **replicas that are always up-to-date and synchronized with the partition’s leader**. The leader continuously sends messages to the in-sync replicas, and they acknowledge the receipt of those messages.

The recommended value for ISR is always greater than 1.

The ideal value of ISR is equal to the replication factor.

**5. Message Acknowledgments in Kafka**

When utilizing Kafka, producers are restricted to writing data exclusively to the leader broker for a given partition. Furthermore, **producers must specify the level of acknowledgment, known as acks, to determine the minimum number of replicas that need to receive the message before considering the write operation successful.**

Let us consider a few scenarios of how this value affects the message producers.

**acks = 0**

Producers consider messages as “successfully written” as soon as they are sent, without waiting for the broker to confirm their acceptance.



However, this approach comes with a risk. If the broker goes offline or an exception occurs, the producer won’t receive any notification and data loss may occur. This method is typically suitable for scenarios where it is acceptable to lose messages, such as in metrics collection. It offers the advantage of achieving the highest throughput setting since the network overhead is minimized.

**acks = 1**

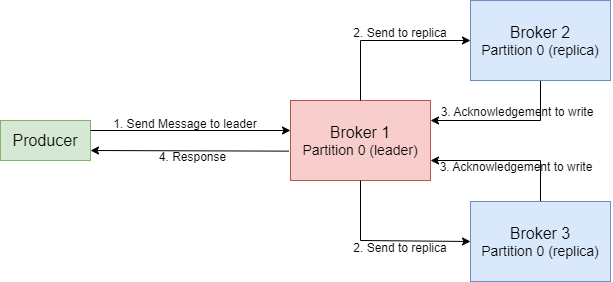
When the acks value is set to 1, producers consider messages as “written successfully” only when the message receives an acknowledgment from the leader.



While requesting a response from the leader, the replication process occurs in the background, but **it doesn’t guarantee replication**. In the event of not receiving an acknowledgment, the producer can retry the request. However, if the leader broker goes offline unexpectedly and the replicas haven’t replicated the data yet, data loss may occur.

**acks = all**

Producers consider messages as “written successfully” only when the message is accepted by all in-sync replicas (ISR).



To ensure the safety of writing the message, the leader for a partition checks if there are enough in-sync replicas, which is determined by the broker setting*min.insync.replicas*. The request remains stored in a buffer until the leader confirms that the follower replicas have replicated the message. At this point, a successful acknowledgment is sent back to the client.

For instance, let’s consider a topic with three replicas and *min.insync.replicas* set to 2. In this case, writing to a partition in the topic is possible only when at least two out of the three replicas are in sync. When all three replicas are in-sync, the process proceeds as usual. This remains true even if one of the replicas becomes unavailable. However, if two out of three replicas are unavailable, the brokers will no longer accept produce requests. Instead, producers attempting to send data will receive a *NotEnoughReplicasException*.

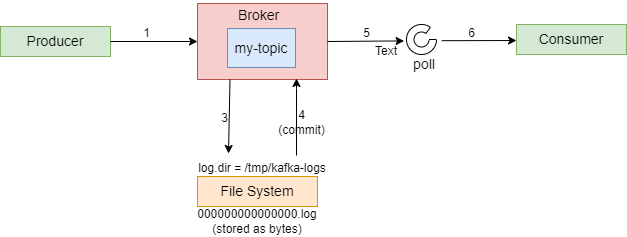
The most widely adopted choice for ensuring data durability and availability, capable of tolerating the loss of a single Kafka broker, is setting “**acks=all**” and “**min.insync.replicas=2**“.

**6. Kafka Commit Log and Retention Policy**

**6.1. Commit Log**

In Apache Kafka, the commit log is an append-only data structure that records all published messages in the order they were received. Each record in the log each record represents a single message, in the order they are produced, maintaining the message ordering within a partition.

Let’s understand the commit log in Kafka using the diagram below.



When the message is produced, the record or log is saved as a file with the “*.log*” extension. Each partition within the Kafka topic has its own dedicated log file. Therefore, if there are six partitions for a topic, there will be six log files in the file system. These files are commonly referred to as ***Partition Commit Logs***.

After the messages are written to the log file, the produced records are then committed. Consequently, only the records that have been committed to the file system are visible to consumers actively polling for new records.

Subsequently, as new records are published to the Kafka Topic, they are appended to the respective log file, and the process continues seamlessly, ensuring that messages are not lost in the event of failures.

Note that although the commit log is an append-only structure, Kafka provides efficient random access to specific offsets within a partition. Consumers can read messages from any offset in a partition, allowing them to replay or skip messages as needed.

**6.2. Retention Policy**

The retention policy serves as the primary determinant of how long messages will be stored, making it a crucial policy to establish. By default, Kafka retains messages for a period of *168 hours*, equivalent to *7 days*. This default retention period can be adjusted as needed.

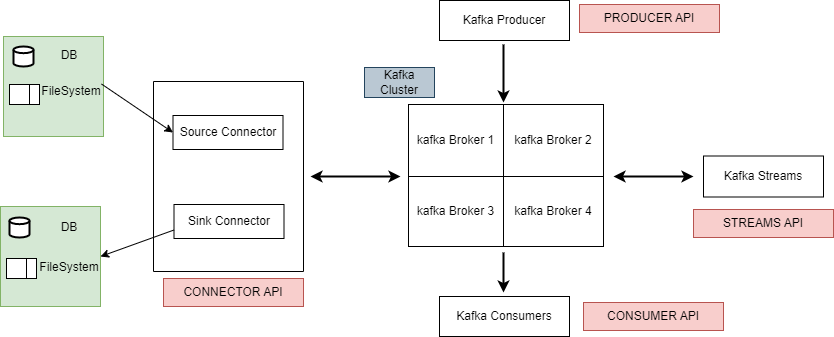
If the log retention period is exceeded, Kafka will automatically delete the corresponding data from the log. This process is controlled by the *log.retention.check.interval.ms* property, which specifies the interval at which retention checks occur (e.g., 300000 milliseconds).

Also, when the log size reaches a specified threshold, a new log segment is created. The *log.segment.bytes* property determines the size of each log segment, with a default value of 1073741824 bytes (1 gigabyte).

**7. Apache Kafka APIs**

Kafka APIs play a crucial role in enabling the implementation of various data pipelines and real-time data streams. They serve as a bridge between Kafka clients and Kafka servers, facilitating seamless communication and data transfer.

There are 4 APIs available that developers can use to leverage Kafka capabilities:



**7.1. Producer API**

It allows applications to effortlessly publish a continuous stream of data to a designated Kafka topic.

Properties properties = **new** Properties();

properties.put("bootstrap.servers", "localhost:9092");

properties.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");

properties.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

Producer<String, String> producer = **new** KafkaProducer<>(properties);

String topic = "my\_topic";

String key = "key1";

String value = "Hello, Kafka!";

ProducerRecord<String, String> record = **new** ProducerRecord<>(topic, key, value);

producer.send(record, **new** Callback() {

@Override

**public** **void** onCompletion(RecordMetadata metadata, Exception exception) {

**if** (exception == **null**) {

System.out.println("Message sent successfully. Offset: " + metadata.offset());

} **else** {

System.err.println("Error sending message: " + exception.getMessage());

}

}

});

producer.close();

**7.2. Consumer API**

It allows applications to subscribe to one or multiple topics, enabling them to consume and process the stored data stream. It can handle real-time records or process historical data.

Properties properties = **new** Properties();

properties.put("bootstrap.servers", "localhost:9092");

properties.put("group.id", "my-group");

properties.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

properties.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = **new** KafkaConsumer<>(properties);

String topic = "my\_topic";

consumer.subscribe(Collections.singletonList(topic));

**while** (true) {

ConsumerRecords<String, String> records = consumer.poll(100);

**for** (ConsumerRecord<String, String> record : records) {

System.out.printf("Received message: key=%s, value=%s%n", record.key(), record.value());

}

}

**7.3. Connector API**

It enables developers to create reusable connectors, serving as either producers or consumers, simplifying and automating the integration of external data sources into Kafka. It comprises two types of connectors:

* **Source Connector** – is used to pull data from an external data source such as DB, File System or Elasticsearch and store them in Kafka topics, making the data available for stream processing.
* **Sink Connector** – is used to push data from Kafka Topics to Elasticsearch, or batch systems such as Hadoop for offline analysis.

Source Connector Example

{

"name": "mysql-source-connector",

"config": {

"connector.class": "io.confluent.connect.jdbc.JdbcSourceConnector",

"tasks.max": "1",

"connection.url": "jdbc:mysql://localhost:3306/mydb",

"connection.user": "user",

"connection.password": "password",

"mode": "timestamp",

"timestamp.column.name": "updated\_at",

"table.whitelist": "my\_table",

"topic.prefix": "mysql-",

"validate.non.null": "false"

}

}

**7.4. Stream API**

It is built upon the foundations of the Producer and Consumer APIs. It offers advanced processing capabilities and empowers applications to engage in continuous, end-to-end stream processing. This involves consuming records from one or multiple topics, performing analysis, aggregation, or transformation operations as required, and subsequently publishing the resulting streams back to the original topics or other designated topics.

Properties properties = **new** Properties();

properties.put(StreamsConfig.APPLICATION\_ID\_CONFIG, "my-streams-app");

properties.put(StreamsConfig.BOOTSTRAP\_SERVERS\_CONFIG, "localhost:9092");

StreamsBuilder builder = **new** StreamsBuilder();

KStream<String, String> sourceStream = builder.stream("my\_input\_topic");

KStream<String, String> transformedStream = sourceStream

.filter((key, value) -> value.contains("important"))

.mapValues(value -> value.toUpperCase());

transformedStream.**to**("my\_output\_topic");

KafkaStreams streams = **new** KafkaStreams(builder.build(), properties);

streams.start();

*// Shutdown hook to gracefully stop the stream processing application*

Runtime.getRuntime().addShutdownHook(**new** Thread(streams::close));

**7.5. Admin API**

Kafka provides APIs for administrative operations and metadata management for Kafka clusters. Using admin APIs, developers can create and delete topics, manage consumer groups, modify configurations, and retrieve cluster metadata.

*// Set up Kafka admin properties*

Properties adminProps = **new** Properties();

adminProps.put(AdminClientConfig.BOOTSTRAP\_SERVERS\_CONFIG, "localhost:9092");

AdminClient adminClient = AdminClient.create(adminProps);

*// Define the new topic*

String topicName = "myNewTopic";

**int** numPartitions = 3;

**short** replicationFactor = 1;

*// Create a NewTopic object*

NewTopic newTopic = **new** NewTopic(topicName, numPartitions, replicationFactor);

*// Create the topic*

adminClient.createTopics(Collections.singletonList(newTopic));

*// Close the admin client*

adminClient.close();

System.out.println("Topic " + topicName + " created successfully.");

Other than the above-mentioned APIs, developers can use [KSQL](https://ksqldb.io/) which is an open-source streaming SQL engine for Apache Kafka. It is an SQL engine that allows us to process (transformations and aggregations) and analyze the real-time streaming data present in the Apache Kafka platform. Developers can use standard SQL constructs like *SELECT*, *JOIN*, *GROUP BY*, and *WHERE* clauses to query and manipulate data.

[Check this article](https://howtodoinjava.com/kafka/kafka-cluster-setup-using-docker-compose/) to learn how to start a Kafka Server in our local systems and execute different Kafka commands.

**8. Advantages**

Below are some of the advantages of using Apache Kafka:

* **Scalability** – Apache Kafka offers seamless scalability without any downtime by adding additional nodes on the fly. Message handling within the Kafka cluster is fully transparent, allowing for smooth scalability.
* **Durability** – Apache Kafka ensures a durable messaging service by storing data quickly. Messages are persisted on disk, minimizing the risk of data loss.
* **Fault-Tolerant** – Apache Kafka has built-in capabilities to withstand node or machine failures within a cluster. This fault-tolerant design ensures the system remains operational even in the face of failures.
* **High Concurrency** – Apache Kafka excels at handling a high volume of messages per second, even in low-latency conditions, with exceptional throughput. It enables concurrent reading and writing of messages, facilitating efficient data processing.

**9. Limitations**

Below are some of the limitations of using Apache Kafka:

* **Tool Support** – Apache Kafka is known to have limited management and monitoring tools, which can make enterprise support staff apprehensive about choosing Kafka and providing long-term support.
* **Issues with Message Tweaking** – Since Kafka relies on specific system calls to deliver messages to consumers, any modifications made to the messages can negatively impact performance. Tweaking messages significantly reduces Kafka’s efficiency, except when the message remains unchanged.
* **No support for wildcard topic selection** – Kafka only matches exact topic names and does not offer support for wildcard topic selection. This limitation prevents Kafka from addressing certain use cases that require matching patterns using wildcards due to its algorithmic constraints.
* **Performance** – While individual message size typically does not pose issues, as the size of messages increases, brokers and consumers begin compressing them. This compression process gradually consumes node memory when the messages are decompressed. Additionally, compression during data pipeline flow affects throughput and overall performance.

**10. Conclusion**

This Apache Kafka tutorial provided a comprehensive overview of Kafka and its key features. We have explored the various components of the Kafka cluster, including brokers, producers, and consumers and delved into the core concepts such as topics, partitions, consumer groups, commit logs and retention policy.

**1. Open Command Prompt**

Before running commands, navigate to the Kafka installation directory in Command Prompt or PowerShell.

cd C:\kafka\_2.12-3.1.0

**2. Produce Messages to a Kafka Topic**

**Command:**

bin\windows\kafka-console-producer.bat --bootstrap-server localhost:9092 --topic <topic\_name>

**Example:**

bin\windows\kafka-console-producer.bat --bootstrap-server localhost:9092 --topic topic\_1

1. After running the command, you'll see a blinking cursor.
2. Type your messages and press Enter to send them to the Kafka topic.

**3. Consume Messages from a Kafka Topic**

**Command:**

bin\windows\kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic <topic\_name> --from-beginning

**Example:**

bin\windows\kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic topic\_1 --from-beginning

* **--from-beginning**: Fetches all messages from the topic since its creation. Omit this if you want to consume only new messages.

**4. Start Kafka Broker**

Ensure the Kafka server is running before producing or consuming messages:

**Start ZooKeeper (if required):**

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

**Start Kafka Broker:**

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

**5. List Topics (Optional)**

You can list all topics to confirm your topic exists:

bin\windows\kafka-topics.bat --bootstrap-server localhost:9092 --list

**6. Customizing Commands**

**Produce Messages with Keys:**

bin\windows\kafka-console-producer.bat --bootstrap-server localhost:9092 --topic <topic\_name> --property "parse.key=true" --property "key.separator=:"

Example input:

key1:message1

key2:message2

**Consume Messages from a Specific Partition:**

bin\windows\kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic <topic\_name> --partition <partition\_number>

Let me know if you encounter any issues!