

▶ Kafka Series (Part 1)

Before understanding Kafka, let's first understand a few terminologies.

✓ **Publish-Subscribe:**

The publish-subscribe model is a messaging pattern used in distributed systems, where:

- ✓ Senders of messages (publishers) do not send messages directly to specific receivers (subscribers).
- ✓ Publishers send messages to an intermediary called a **Message Broker** which then distributes the messages to all interested subscribers.

In this model:

- ✓ Publishers and subscribers are **decoupled from each other**, i.e., that they do not need to know about the existence of each other.
- ✓ Publishers only need to know the topic or channel to which they want to publish messages, and subscribers only need to know the topics or channels to which they want to subscribe.

✓ **Message Broker**

- ✓ It is an intermediary service that enables communication between applications or components by transmitting messages between them.
- ✓ It typically provides a set of features, including message routing, transformation, and filtering, as well as scalability, reliability, and fault-tolerance.
- ✓ It is commonly used in enterprise integration and distributed systems, including microservices architectures, event-driven systems, and service-oriented architectures.
- ✓ Some of the popular message brokers are RabbitMQ, IBM MQ, Apache Kafka, Amazon SQS, Apache Pulsar, etc.

✓ **Event Streaming**

Event streaming is the practice of:

- ✓ Capturing data in real-time from event sources like databases, sensors, mobile devices, cloud services, and software applications in the form of streams of events.
- ✓ Storing these event streams durably for later retrieval.
- ✓ Manipulating, processing, and reacting to the event streams in real-time.
- ✓ Routing the event streams to different destination technologies as needed.

✓ **What exactly is Kafka?**

- ✦ It is an open-source distributed **Event Streaming** platform.
- ✦ It is used for building real-time data pipelines and streaming applications.
- ✦ It offers a **Publish-Subscribe** based messaging system.
- ✦ It was originally developed at **LinkedIn** as a stream processing platform and was subsequently open sourced in the late 2010 and accepted as an Apache Software Foundation incubator project in July 2011.
- ✦ The development team at LinkedIn was led by **Jay Kreps (currently, Co-founder and CEO at Confluent)**.

Kafka Series (Part 2)

What is a message?

- ✓ It is the unit of data that is produced and consumed by Kafka clients.
- ✓ It is simply an array of bytes, as far as Kafka is concerned.
- ✓ Each message in Kafka has two parts: a **key** and a **value**.
- ✓ The key is used to determine the partition to which a message is assigned.
- ✓ The value contains the actual data of the message.

What is an Offset?

- ✓ It is another piece of metadata that Kafka adds to each message as(when) it is produced.
- ✓ It is an integer value that continually increases.
- ✓ Each message in a given partition has a unique offset.

What are Topics and Partitions?

- ✓ Messages in Kafka are categorized into topics.
- ✓ You may think of a topic as a database table or a folder in a filesystem.
- ✓ Kafka topics are broken down into a number of Partitions.

Important:

- ✓ There is **no guarantee** of message ordering **across the entire topic** as the topic is broken down into several partitions.
- ✓ There is a **guarantee** of message ordering within a single partition.

What are Producers?

The Producers:

- ✓ Are client applications that **publish messages** to a Kafka cluster.
- ✓ Can be written in a variety of programming languages using Kafka client libraries.
- ✓ Produce message for a specific Kafka topic (sometimes also for a specific partition).

Important:

- ✓ By default, the producer will balance messages over all partitions of a topic **evenly**.
- ✓ In some cases, the producer can send messages to a **specific partition**. This is achieved using the message "key" and a partitioner that generates a hash of the key and map it to a specific partition. This ensures that all messages produced with a given key will get written to the same partition.
- ✓ The producers could also use a custom partitioner for mapping messages to partitions.

What are Consumers?

The Consumers:

- ✓ Are client applications that **read data from Kafka topics**.

- ✓ Subscribe to one or more topics and reads the messages in the order in which they were produced to each partition.
- ✓ Keeps track of which messages it has already consumed by keeping track of the offset of messages.

▶ Kafka Series (Part 3)

✓ **What is a Consumer Group?**

- ✦ A consumer group in Kafka is a group of one or more consumers that work together to consume and process the messages from Kafka topics.
- ✦ The group ensures that each partition is only consumed by one member.

Important:

- ✓ When a topic receives new messages, Kafka distributes those messages across the partitions of the topic.
- ✓ Each consumer group is assigned a set of partitions to consume from.
- ✓ Each consumer in the group reads from a unique subset of those partitions.
- ✓ If a single consumer fails, the remaining members of the group will reassign the partitions being consumed to take over for the missing member.
- ✓ Consumer groups in Kafka are useful for scaling the processing of messages across multiple consumers.

✓ **What is a Broker in Kafka?**

- ✦ A single Kafka server is called a broker.
- ✦ It receives messages from producers, assigns offsets to them, and writes the messages to storage on disk.
- ✦ It also services consumers, responding to fetch requests for partitions and responding with the messages that have been published.
- ✦ It store data in the form of topics, which are partitioned and distributed across multiple brokers in a cluster.

✓ **What is Kafka cluster?**

- ✦ A cluster is a group of broker servers that work together to provide a distributed messaging system.
- ✦ Kafka clusters are designed to provide high availability, fault tolerance, and scalability by distributing data across multiple nodes.
- ✦ Within a cluster of brokers, one broker will also function as the cluster controller (elected automatically from the live members of the cluster).

Important:

- ✓ A partition is owned by a single broker in the cluster, and that broker is called the **leader** of the partition.
- ✓ A replicated partition is assigned to additional brokers, called **followers** of the partition.
- ✓ All producers must connect to the leader in order to publish messages.

✓ Consumers may fetch from either the leader or one of the followers.

✓ **What is retention policy?**

- ✦ It is defined as the durable storage of messages for some period of time.
- ✦ Kafka brokers are configured with a default retention setting for topics, either retaining messages for some period of time (e.g., 7 days) or until the partition reaches a certain size in bytes (e.g., 1 GB).
- ✦ Once these limits are reached, the messages are expired and deleted.

▶ **Kafka Series (Part 4)**

✦ **What is a MirrorMaker?**

- ✓ It is a tool included in the Kafka project used for **replicating data to other clusters**.
- ✓ It is simply a Kafka consumer and producer, linked together with a queue.
- ✓ Messages are consumed from one Kafka cluster and produced to another.
- ✓ It is quite useful when there is the need for multiple Kafka clusters in multiple datacenters (for segregation of types of data, isolation for security requirements, disaster recovery, etc.).

✦ **What is Zookeeper?**

- ✓ It is like a centralized service that manages cluster memberships, relevant configurations, and cluster registry services.
- ✓ It acts as a Kafka **cluster coordinator** that manages cluster membership of brokers, producers, and consumers participating in message transfers via Kafka.
- ✓ It also helps in **leader election** for a Kafka topic.

✓ **Summary of Kafka features:**

✦ **Multiple Producers**

Kafka is able to seamlessly handle multiple producers, whether those clients are using many topics or the same topic.

✦ **Multiple Consumers**

Kafka is designed for multiple consumers to read any single stream of messages without interfering with other clients.

✦ **Disk-Based Retention**

Durable message retention means that consumers do not always need to work in real time. Messages are written to disk and will be stored with configurable retention rules. Durable retention means that if a consumer falls behind, either due to slow processing or a burst in traffic, there is no danger of losing data.

✦ **Scalable**

Kafka's flexible scalability makes it easy to handle any amount of data. We can start with a single broker for POC and move to production with a large cluster easily.

🚩 High Performance

All these features come together to make Apache Kafka a publish/subscribe messaging system with excellent performance under high load. Producers, consumers, and brokers can all be scaled out to handle very large message streams with ease.

🚩 Platform Features

These features are in the form of APIs and libraries:

✓ *Kafka Connect*:

It assists with the task of pulling data from a source data system and pushing it into Kafka or pulling data from Kafka and pushing it into a sink data system.

✓ *Kafka Streams*:

It provides a library for easily developing stream processing applications that are scalable and fault tolerant.

With the latest version of Kafka, By default the producer will batch the records (bunch of records) and then send to Kafka broker. Single partition even though Round Robin partition present and no key. This is to avoid making more connection requests to Broker. When we want to achieve higher Throughput, we can use three Kafka topic properties like `rolling.interval.ms`, `batch size` and `compression` considering very minimal latency delay.

Each message can be sent with additional optional parts like, header, timestamp along with key, value. There are Producer Record constructors present in the Kafka client source code. Batching messages enables a Kafka producer to increase its throughput.

Reducing the number of network requests the producer makes in order to send data will improve the performance of the system.

▶ Kafka Series (Part 5) - Kafka Installation on the local system

▶ Kafka Series (Part 6)

Kafka Producer.

👉 Please don't worry if you don't understand a few terms like '*Future*', '*Asynchronous*', '*Synchronous*', '*Serializer*', etc, which are used here.

👉 A synchronous request blocks the client until operation completes.

👉 An asynchronous request doesn't block the client i.e. browser is responsive.

👉 Serialization in Java is a mechanism of writing the state of an object into a byte-stream.

The reverse operation of serialization is called deserialization where byte-stream is converted into an object.

👉 In Java, Future is an interface that belongs to `java.util.concurrent` package. It is used to represent the result of an asynchronous computation.

👉 Applications need to write messages to Kafka for a variety of use cases, e.g.,:

- ✓ Recording user activities from a website for auditing or analysis.
- ✓ Recording metrics.
- ✓ Storing log messages.
- ✓ Recording information from smart devices.
- ✓ Buffering information before writing to a database.

Constructing a Kafka Producer

👉 Create a Producer object with the required properties. The mandatory properties are:

✓ ***bootstrap.servers***: List of host:port pairs of brokers that the producer will use to establish initial connection to the Kafka cluster.

✓ ***key.serializer***: Name of a class that will be used to serialize the keys of the records being produced to Kafka (e.g., `org.apache.kafka.common.serialization.StringSerializer`).

✓ ***value.serializer***: Name of a class that will be used to serialize the values of the records being produced to Kafka.

👉 Once we instantiate a Producer object, the below three primary methods can be used for sending messages:

✦ **Fire-and-Forget:**

- ✓ We send a message to the server and don't really care if it arrives successfully or not.
- ✓ In case of nonretriable errors or timeout, messages will get lost, and the application will not get any information or exceptions about this.

✦ **Synchronous:**

- ✓ When we send a message, the `send()` method returns a Future object. We use `get()` to wait on the Future and see if the `send()` was successful or not before sending the next record.
- ✓ It can lead to poor performance as the thread will spend time waiting and doing nothing else.
- ✓ It is usually not used in production applications.

✦ **Asynchronous:**

- ✓ To send messages asynchronously and still handle error scenarios, the producer supports adding a callback when sending a record.
- ✓ We call the `send()` method with a callback function, which gets triggered when it receives a response from the Kafka broker.

▶▶ **Kafka Series (Part 7)**

✅ **Producer Configuration**

The producer has a large number of configuration parameters. However, some of the parameters have a significant impact on **memory use**, **performance**, and **reliability** of the producers.

▶▶ Below are some of the important ones.

📌 ***client.id:***

- ✓ It's a logical identifier for the client and the application it's used in.
- ✓ It can be any string and will be used by the brokers to identify messages sent from the client.
- ✓ It is used in logging and metrics.
- ✓ Choosing a good client name will make troubleshooting much easier.

📌 ***acks:***

- ✓ It controls how many partition replicas must receive the record before the producer can consider the write successful.
- ✓ It has a significant impact on the durability of the written messages.

The 'acks' parameter has the below three values.

📌 ***acks=0:***

- ▶ The producer will not wait for a reply from the broker before assuming the message was sent successfully.
- ▶ If something goes wrong and the broker does not receive the message, the producer will not know about it, and the message will be lost.
- ▶ We can send messages as fast as the network will support (very high throughput).

📌 ***acks=1:***

- ▶ The producer will receive a success response from the broker the moment the leader replica receives the message.
- ▶ If the message can't be written to the leader, the producer will receive an error response and can retry sending the message, avoiding potential loss of data.
- ▶ The message can still get lost if the leader crashes and the latest messages were not yet replicated to the new leader.

📌 ***acks=all:***

- ▶ The producer will receive a success response from the broker once all in sync replicas receive the message.
- ▶ This is the safest mode since we can make sure more than one broker has the message and that the message will survive even in case of a crash.
- ▶ The latency will be higher, since we will be waiting for more than just one broker to receive the message.

▶▶ **Kafka Series (Part 8)**

Producer Configuration [Continued...]

👉 Below are some of the very important configurations related to Kafka Producers:

✦ **buffer.memory**

It sets the amount of memory the producer will use to buffer messages waiting to be sent to brokers.

✦ **compression.type**

By default, messages are sent uncompressed. This parameter can be set to snappy, gzip, lz4, or zstd, in which case the corresponding compression algorithms will be used to compress the data before sending it to the brokers.

✦ **batch.size**

When multiple records are sent to the same partition, the producer will batch them together. This parameter controls the amount of memory in bytes that will be used for each batch.

✦ **max.request.size**

It controls the size of a produce request sent by the producer. It caps both the size of the largest message that can be sent and the number of messages that the producer can send in one request.

✦ **receive.buffer.bytes & send.buffer.bytes**

These are the sizes of the TCP send and receive buffers used by the sockets when writing and reading data. If these are set to -1, the OS defaults will be used.

✦ **enable.idempotence**

- ✓ A service is called idempotent if performing the same operation multiple times has the same result as performing it a single time.
- ✓ Starting in version 0.11, Kafka supports exactly once semantics. Exactly-once messaging semantics with Kafka means the combined outcome of multiple steps will happen exactly-once.
- ✓ A message will be consumed, processed, and resulting messages produced, exactly-once.

✦ **max.block.ms**

It controls how long the producer may block when calling send() and when explicitly requesting metadata via partitionsFor().

✦ **delivery.timeout.ms**

It will limit the amount of time spent from the point a record is ready for sending (send() returned successfully and the record is placed in a batch) until either the broker responds or the client gives up, including time spent on retries.

✦ **request.timeout.ms**

It controls how long the producer will wait for a reply from the server when sending data.

📌 **linger.ms**

It controls the amount of time to wait for additional messages before sending the current batch. KafkaProducer sends a batch of messages either when the current batch is full or when the linger limit is reached.

▶ **Kafka Series (Part 9)**

📌 **Serialization.**

👉 Serialization plays an important role in the performance of any distributed application. Formats that are slow to serialize objects into, or consume a large number of bytes, will greatly slow down the computation.

👉 It is the process of translating a data structure or object state into a format that can be stored or transmitted and reconstructed later.

👉 In Part 6, we saw how to use String serializer in our code. Kafka also includes serializers for Integers, ByteArrays, and many more.

👉 Let's talk about **Apache Avro**, which is a data serialization system:

- ✓ It provides rich data structures and simple integration with dynamic languages.
- ✓ It uses JSON for defining data types/protocols and serializes data in a compact binary format.
- ✓ It provides a container file, to store persistent data.
- ✓ An Avro container file consists of a file header, followed by one or more file data blocks.

👉 A file header consists of:

- ✓ Four bytes, ASCII 'O', 'b', 'j', followed by the Avro version number which is 1 (0x01) (Binary values 0x4F 0x62 0x6A 0x01).
- ✓ File metadata, including the schema definition.
- ✓ The 16-byte, randomly-generated sync marker for this file.

👉 We will see the code on how to write Avro records using Kafka producer and how to read those records using consumer. Before that, you need to understand **Schema Registry**.

👉 What is Schema Registry?

- ✓ It provides a centralized repository for managing and validating schemas for topic message data, and for serialization and deserialization of the data over the network.
- ✓ It's not part of Apache Kafka, but there are several open source options to choose from.

👉 We will install Schema registry in our local machine using Docker. Before that, please use the link in the comments to install Docker software on your system as we will be using Docker in the later part of the series to install any kind of software.

👉 Make sure you have local instance of Kafka running. Follow **Part#5** of the series to install and configure Kafka on your local machine.

The docker command to run Schema Registry on your local machine has been provided in comments.

▶ Kafka Series (Part 10):

🔥 *Read and write Avro data using Kafka producers and consumers.* 🔥

👉 Before that, let's understand a bit more about Avro. Below are the benefits of serializing the data in Avro format:

- ▶ Avro relies on a schema. This means every field is properly described and documented.
- ▶ Avro data format is a compact binary format, so it takes less space both on a wire and on a disk.
- ▶ It has support for a variety of programming languages.
- ▶ In Avro, every message contains the schema used to serialize it. That means that when you're reading messages, you always know how to deserialize them, even if the schema has changed.

👉 **Drawback:**

Every Avro message contains the schema used to serialize the message. So, if there is a need to send millions of messages per day to Kafka, it's a waste of bandwidth and storage space to send the same schema information repeatedly.

👉 **Schema Registry to the rescue:**

- ▶ It provides a RESTful interface for storing and receiving Avro schemas.
- ▶ It supports schema evolution and allows us to enforce the rules for validating a schema compatibility when the schema is modified.

👉 **How does it work?**

- ▶ The schema is not sent inside the Kafka record.
- ▶ The producer checks whether schema already exists in the Schema Registry.
- ▶ If the schema is not present, it will write the schema in the Schema Registry.
- ▶ The producer will obtain the id of the schema and will send that id inside the record, saving a lot of space.
- ▶ The consumer will read the message and then contact the Schema Registry with the schema id from the record to get the full schema and cache it locally.

👉 Now that you are aware about Avro and Schema Registry, it's time to see them in action.

✳️ **Important Note:**

The Avro serializer can only serialize Avro objects, not POJO (Plain Old Java Object). Generating Avro classes can be done either using the avro-tools.jar or the Avro Maven plugin. I have used *Avro Maven plugin* and you can see the dependencies in pom.xml file.

▶ Kafka Series (Part 11)

🔥 What are *Interceptors* in Kafka? 🔥

✦ Interceptors are a pluggable mechanism that allows us to intercept and process messages as they are being produced or consumed by Kafka clients.

✦ We can modify the behavior of our Kafka client application without modifying its code.

→ There are various use cases of Interceptors, such as:

✦ **Logging:** Interceptors can be used to log messages that are produced or consumed by Kafka clients. This can be useful for debugging purposes.

✦ **Metrics collection:** Interceptors can be used to collect metrics on the messages that are produced or consumed by Kafka clients. This can be useful for monitoring and performance tuning.

✦ **Security:** Interceptors can be used to add or remove security-related metadata to messages as they are produced or consumed. This can be useful for enforcing security policies and preventing unauthorized access to Kafka topics.

👉 To use an interceptor, we can simply **configure the interceptor class in the Kafka producer or consumer** configuration.

👉 The interceptor will then be **invoked for every message that is sent or received** by the client.

👉 Interceptors should be used with caution, as **they can impact the performance of Kafka clients**.

💡 Additionally, interceptors should only be used for tasks that cannot be accomplished through other means, such as by modifying the producer or consumer code directly. **100**

→ All you need to do is:

▶ Make sure your Kafka local instance is up and running.

▶ Build the project to create JAR file (mvn clean install).

▶ Add the JAR file to the CLASSPATH.

▶ Create a config file (let's say producer.config) with the below content:

```
interceptor.classes=com.ashu.tutorial.interceptors.CountingProducerInterceptor  
counting(dot)interceptor(dot>window(dot)size(dot)ms=10000
```

▶ Create a Kafka topic:

```
$kafka_home/bin/kafka-topics[dot]sh --create --bootstrap-server localhost[colon] --topic  
interceptor-test
```

▶ For testing, run the Kafka console producer:

```
$kafka_home/bin/kafka-console-producer[dot]sh --broker-list localhost[colon]9092 --topic  
interceptor-test --producer.config producer.config
```


→ You will immediately see the Interceptors in action. Try to produce some messages on the console, it will print like this (we have used `System.out.println` in our Interceptor code):

Total sent: 4

Total acknowledged: 0

Kafka Series (Part 12)

Have you heard about Conduktor?

 Conduktor is a software platform designed to make working with Apache Kafka easier for developers, DevOps engineers, and data scientists. Some of the common use cases of Conduktor are:

Monitoring Kafka clusters:

Conduktor provides real-time monitoring of Kafka clusters and helps users keep track of the performance and health of their Kafka clusters. Users can view important metrics such as broker and topic-level throughput, lag, and CPU usage.

Debugging Kafka applications:

Conduktor provides a comprehensive set of debugging tools to help users debug Kafka applications. For example, users can view the details of individual messages, view the message history of a topic, and analyze consumer lag.

Managing Kafka topics and data:


Conduktor provides an intuitive and user-friendly interface to manage Kafka topics and data. Users can create, delete, and modify topics, as well as view and edit messages in real-time.

Securing Kafka clusters:

Conduktor provides features to help users secure their Kafka clusters, such as SSL/TLS encryption and authentication, SASL authentication, and ACL management.

Developing Kafka applications:

Conduktor provides features to help users develop Kafka applications, such as the ability to create and test Kafka producers and consumers within the platform. Additionally, Conduktor provides support for various programming languages and frameworks, including Java, Python, and Spring.

 If you have Docker software installed, you can get Conduktor up and running in no time, for testing and learning purposes.

▶ Kafka Series (Part 13):

🔥 Now that we know about Kafka Producers and how to write a sample Kafka producer application, it's time to understand about *Consumers and Consumer Groups*.

✦ **Consumers and Consumer Groups:**

✓ If a single consumer is reading and processing the data, the application may fall behind, unable to keep up with the rate of incoming messages. So, there is a need to scale consumption from topics.

✓ Just like multiple producers can write to the same topic, we need to allow multiple consumers to read from the same topic, splitting the data among them.

✓ *Kafka consumers are generally part of consumer group*. When multiple consumers are subscribed to the same topic and belong to the same consumer group, *each consumer in the group will receive messages from a different subset of the partitions in the topic*.

👉 So, if we want to scale data consumption from a Kafka topic, we can add more consumers to a consumer group.

✦ **Consumer Groups and Partition Rebalance:**

✓ Let's say we add a new consumer to the group, it will start consuming messages from partitions previously consumed by another consumer.

✓ Let's say a consumer shuts down or crashes, it will leave the group and the partitions it used to consume will be consumed by one of the remaining consumers.

✓ Moving partition ownership from one consumer to another is called a **rebalance**.

✧ **Why Rebalances are important ?**

Because they provide the consumer group with high availability and scalability (allowing us to add and remove consumers).

✧ However, rebalances have downside too. *I'll let you explore about problems that can happen during rebalance.*

✦ **Types of Rebalances:**

▶ **Eager rebalances:**

During an eager rebalance, all consumers stop consuming, give up their ownership of all partitions, rejoin the consumer group, and get a brand-new partition assignment. This is essentially a short window of unavailability of the entire consumer group.

▶ **Cooperative rebalances:**

Cooperative rebalances (also called incremental rebalances) typically involve reassigning only a small subset of the partitions from one consumer to another, and allowing consumers to continue processing records from all the partitions that are not reassigned.

👉 If you want to read more about Kafka rebalances, you may read the article published by Rob Golder.