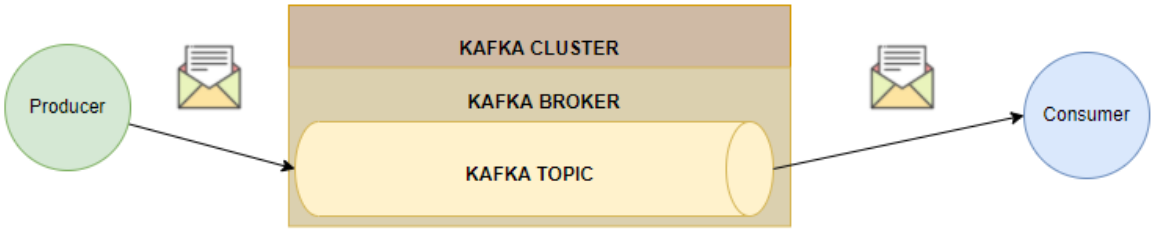
1. Quick Intro To Apache Kafka

We can start by considering Kafka as a message router system.

> Kafka = A High throughput, scalable, durable and fault-tolerant Messaging System



A simple view of the Kafka messaging platform

In the image above, a producer (a process) is sending messages to a Kafka topic that are then consumed by a consumer (a process).

> Kafka can be deployed on bare-metal hardware, virtual machines, and containers, and on-premises as well as in the cloud.

To install Kafka:

1. Download the latest Kafka release and extract it
2. Start Zookeeper service

bin/zookeeper-server-start.sh config/zookeeper.properties

3. Start Kafka broker service

bin/kafka-server-start.sh config/server.properties

We can also download a Kafka docker image, such as *confluentinc/cp-kafka/*

2. Kafka Within Microservices Architecture

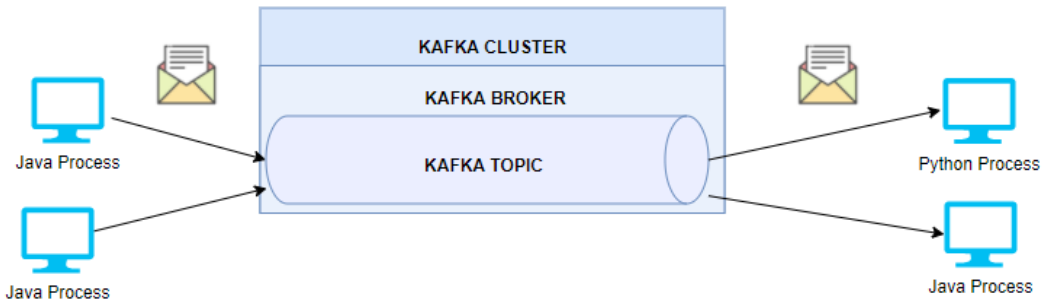
Before we attempt to use Kafka, let's spend some time understanding where Kafka can fit into the architecture and how Kafka helps one decouple the components of a system.

> The most important lesson I've learnt while designing systems is to clearly define the roles and responsibilities of each component within the system.

There are numerous benefits to designing a decoupled system. In a microservices architecture, each service will likely evolve independently. Plus, it is recommended for each service to have its own database. This strategy increases the flexibility of the system as it makes it easier to change the databases as they are specific to the microservices rather than every service being tied to a constraint of a shared database.

One of the ways to decouple microservices is to introduce a messaging and streaming platform in-between the services so that they are not dependent or even aware of each other. To elaborate, two services can communicate with each other via HTTP protocol. And for the services to communicate over the HTTP protocol, they would need to be aware of each other's existence. Additionally, the services are unable to communicate if one of the services is down.

Apache Kafka is an Open source of distributed streaming technology that allows us to build event-based decoupled microservices architecture. We can build real-time event-based microservices with Kafka streaming technology. It is fault-tolerant and enables services written in different languages to communicate with each other without being aware of each other's existence as shown in the image below.



java processes are publishing messages. python + java processes are consuming them.

Kafka provides an asynchronous protocol to help services communicate with each other. Asynchronous is essentially a protocol that is often referred to as fire-and-forget. To further explain, one service can publish data on a Kafka topic. It can then move on to work on the next steps. The messages are picked up from the Kafka topic by a consumer service that then performs the heavy-duty operations and publishes back the response to a Kafka topic upon its completion. The sender service then receives the event from the Kafka topic that the response is ready.

3. Apache-Kafka Key Concepts

Let’s take a deep dive into Kafka.

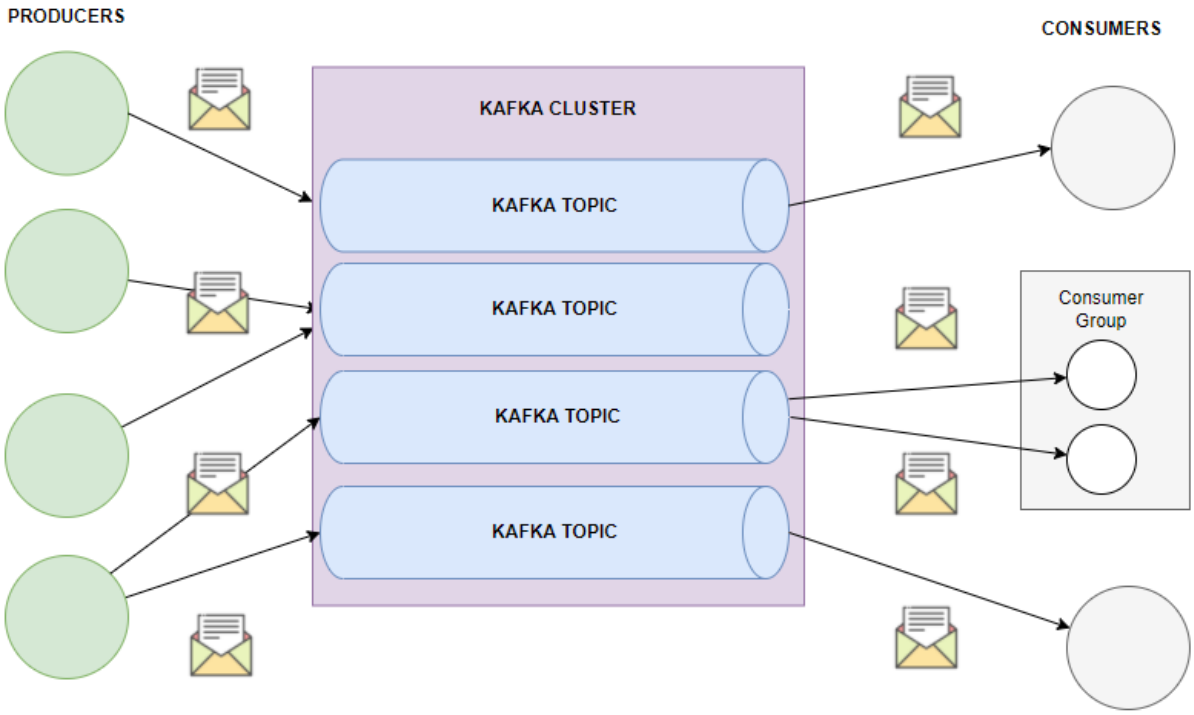
At a high level, the client applications, e.g. a web server, an internal service, and/or a mobile device can be the producers that publish messages to a Kafka topic.

The Kafka ecosystem is designed around clusters, brokers, topics, partitions, producers and consumer groups. I will explain these key concepts in this section.

> Aim is to explain the Kafka ecosystem in a simplistic manner. It will help us design the systems better

1. Producers push messages to a Kafka cluster.
2. A Kafka cluster can have multiple Kafka brokers
3. Each broker can have multiple topics.
4. Brokers can be scaled without any downtime. The brokers can be deployed across multiple machines. This provides high scalability.
5. Brokers can help us connect applications running on different technologies.

> Essentially, there are one or more Kafka brokers in-between the consumer and the publisher.



> The image is highlighting that the messages are published by the Producers and they are consumed by the Consumers. The producers and consumers are

processes that can be implemented in any technology that supports Kafka, such as Python, C#, Java, Scala and so on.

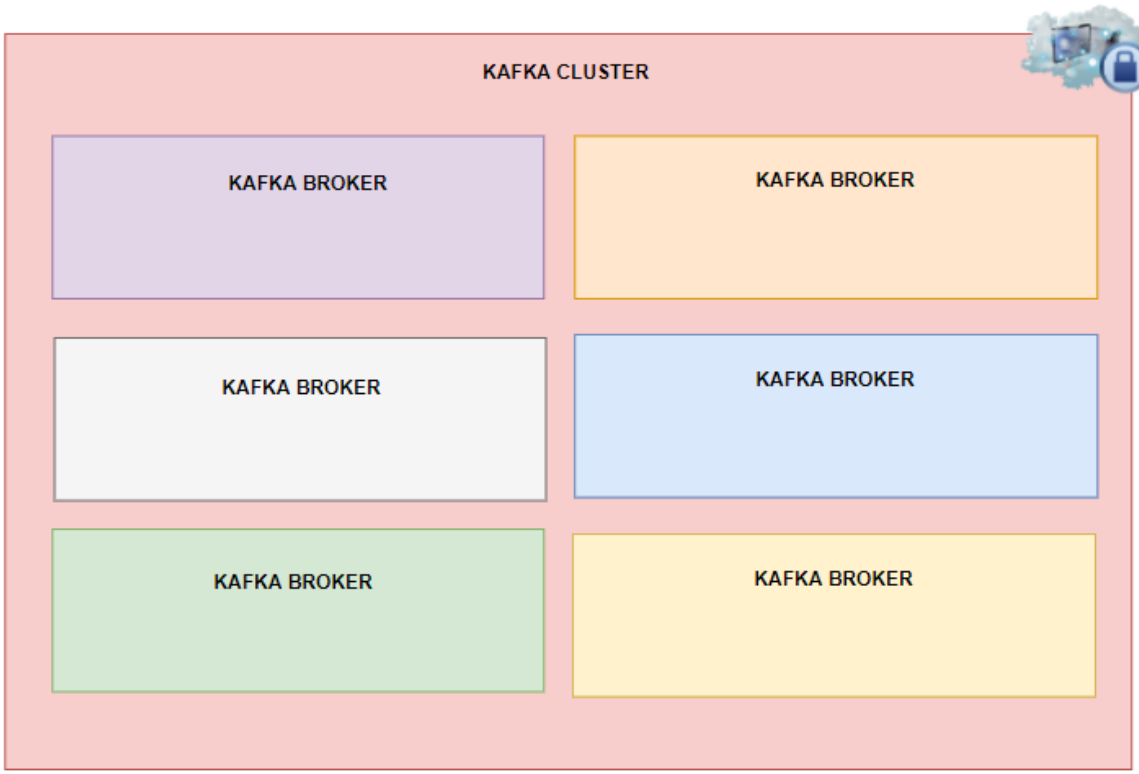
- Each topic can have multiple partitions. Each partition stores the messages in log files. Each message has an offset Id
- Producers publish the messages on a Kafka topic. A producer can be an independent process running on a server. Kafka buffers the data. It then persists the data and replicates it on the disk across nodes in the cluster. This ensures that the system is fault-tolerant.
- Messages are consumed by consumers. Consumers can be independent processes running on different machines than the producers. Consumers can be grouped together in a consumer group
- We can interact with Kafka broker API using a number of technologies from Python, Scala, Java and so on.

4. Kafka Building Blocks

Let's understand the key components of Apache Kafka. I will focus on the Kafka cluster, broker, zookeeper, topic, partition, offset, consumer and consumer group components of the platform that I have highlighted in the previous section.

Kafka Cluster

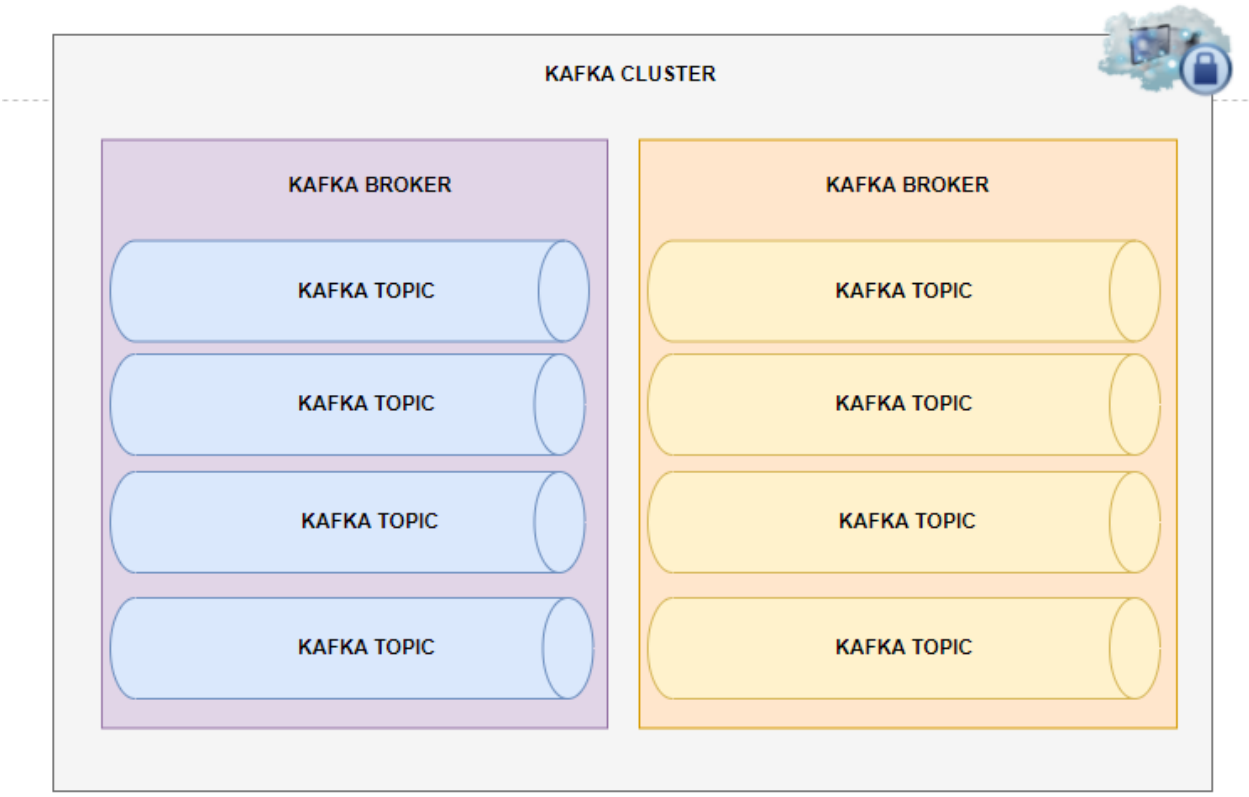
- At the heart of the Kafka platform is a Kafka cluster.
- The Kafka cluster is essentially a distributed system that runs the Kafka services that process the messages. Kafka brokers are installed within Kafka clusters



- Kafka is suitable for both offline and online message consumption. Kafka messages are persisted on the disk and replicated within the cluster to prevent data loss.

Kafka Broker

- Kafka broker is a component that runs on a server within a Kafka cluster. Therefore, a Kafka broker is part of a Kafka cluster.
- There can be many Kafka brokers running within a Kafka cluster. This encourages reliability and provides load-balancing.
- One broker serves as the leader of each partition, while the others act as followers.
- The brokers can run in a distributed manner and are synchronised by Zookeeper.



- A broker can handle thousands of large messages per second.
- A broker can manage one or more topics where publishers publish messages on a continuous basis and from where the consumers consume messages.

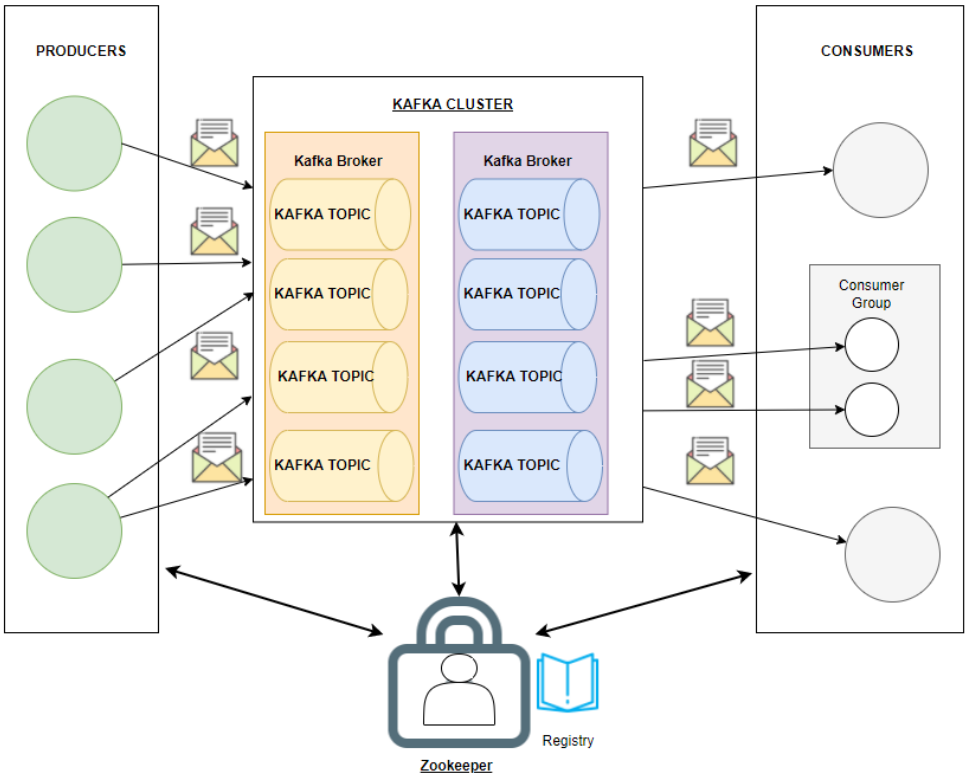
Apache ZooKeeper

- Apache Zookeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. It is the registry-keeper.

  Zookeeper is a synchronization service that can be distributed

- Kafka brokers are statless. The Zookeeper is used to maintain the state of the clusters.
- Kafka is built on top of the ZooKeeper synchronization service.
- In short, Zookeeper facilitates configuration, coordination between servers, leader election and related management tasks.

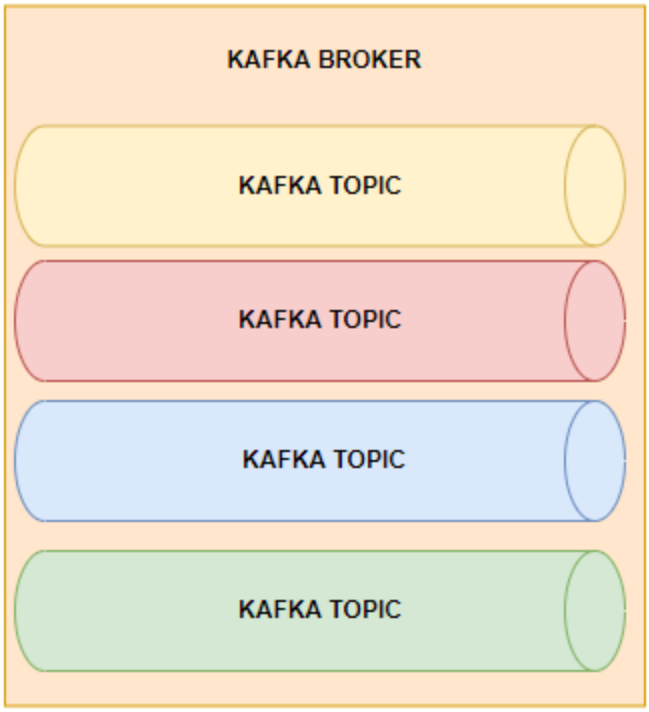
  Zookeeper keeps a naming registery



- Zookeeper checks if a broker is running by sending heart-beat requests. Therefore, we can query zoo-keeper to find all available brokers and topics within each broker.
- Zookeeper elects partition leaders.
- ZooKeeper is also aware of the publishers, the consumers and the messages count of each consumed message from a topic.

Topic

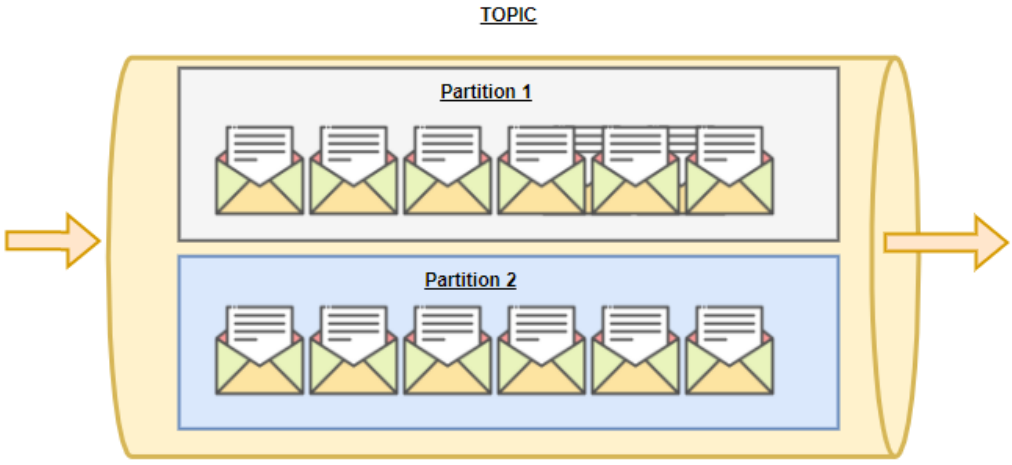
- Let’s understand what a topic is. As illustrated in the image above, a topic is a bucket where messages are published and from where the messages are consumed.
- Topics are split into partitions. Messages are stored in log files. The messages are organised and retained within the partitions.



- Kafka gets a message from a producer, saves it in a log file and sends the message to a consumer.
- The producer and consumer can be two different applications running in completely different environments using completely different technologies.
- With the message router and log saver features, Kafka can replicate the messages. This makes Kafka fault-tolerant and it can handle failures gracefully.

Partitions and Log Files

- A topic can have multiple partitions. We can partition a topic across multiple servers.
- Think of a partition as a placeholder that is used to hold the logs/messages. The log messages can be replayed. This feature allows us to build replayable systems where a production instance can be copied over to a development environment to replay historic events.
- Each partition can be replicated across different machines to provide fault tolerance and high availability.
- Having multiple partitions in a topic enables parallel consumption of messages.



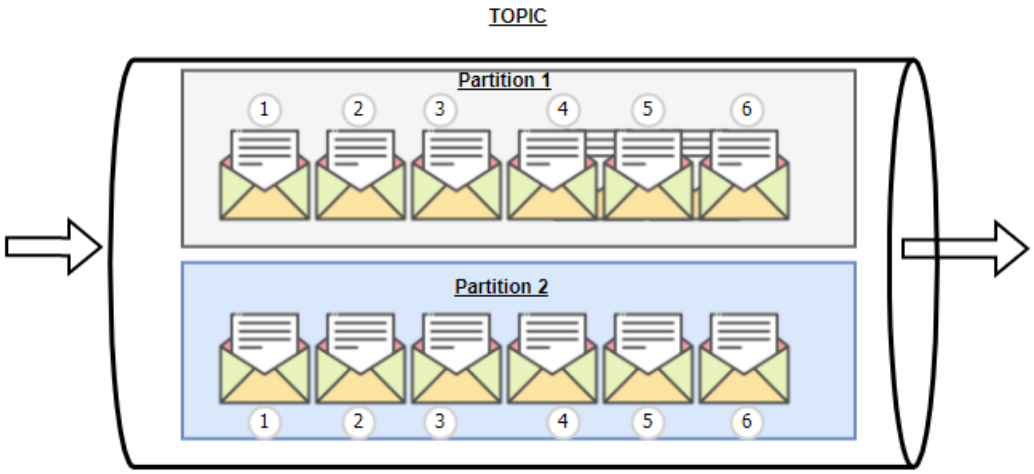
- Each partition keeps records in a sequence.
- The logs are append-only in nature and the log files where the logs are appended can be distributed across multiple machines.
- Each partition can be uniquely recognised by a key that can be a property of the log message. Each log message has a position. And all messages are stored sequentially in a log file. This is known as the offset. This allows Kafka to save the same message in the same partition. Plus, this guarantees strong message ordering and it enables us to consume the messages in the right sequence.

  The main point to remember is to use the same partition Id when publishing the messages to ensure the messages are received in the right order.

Consumer Offsets

This section will explain what consumer offset is. It can help us understand how Kafka ensures that the messages are not lost.

- Offset is the position of the message, which is needed for scenarios when we are required to keep track of the order of the messages. It is a sequential ID. The offset ID increases as the messages are pushed into a topic.
- Each message can be uniquely identified by its topic, partition and offset ID.



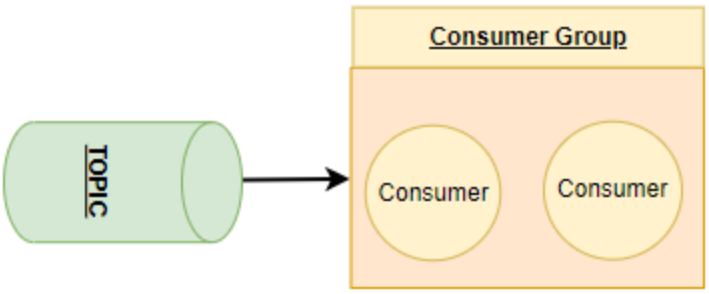
- Let's consider that a consumer faces an unexpected failure and stops working. When it is restarted, it needs to start consuming the messages from where it left off. When a consumer starts, it can either start reading the messages from the beginning or from the end (latest) of the topic. Whenever a consumer consumes a message and commits, the offset is logged and is maintained on the server.

  The concept of offset allows us to replay the messages too. For instance, if we want to simulate the behaviour of a Production environment in a Development environment then we can start consuming the messages from the beginning and consume them in our Development environment.

- A consumer service would have to seek the position of the message to read it. Both reads and writes of log messages are sequential operations and the data resides on the disk. This design of reading and writing messages is O(1). Additionally, if a service is down for some time, it can read the messages where it left off by reading.

Consumer Groups

- A consumer consumes a message from a Kafka topic. Multiple consumers can be grouped into a consumer group.
- Consumers within a consumer group can consume messages across multiple topics. This offers scalability.



- Each consumer remembers the offset of the last message it successfully consumed. Therefore, each consumer group has its own offset per partition. To enable parallel processing of messages, create multiple partitions in a topic. This will enable multiple consumers to process the messages in parallel.
- Kafka can automatically route the messages to the consumers if one of the consumer fails.