Table of contents:-

Sync and Async communication

When two services communicate via APIs (REST or GraphQL) then they wait for the request to complete or throw any error. This wait is to make sure things goes in sequence which is called sync communication.

Async communication is where a service publishes an event that can be consumed by anyone and the publishing service donot care about whether any one consumes or not. It will go on processing its further instructions. It is called async communication.

Why async communication

When we use microservices architecture it has some purposes because of which microservices architecture is recommended.

- One is making our code decoupled and distributing the responsibilities to the different services
- Scaling individually is possible in case of microservices. If the whole service is monolythic then single point of failure is

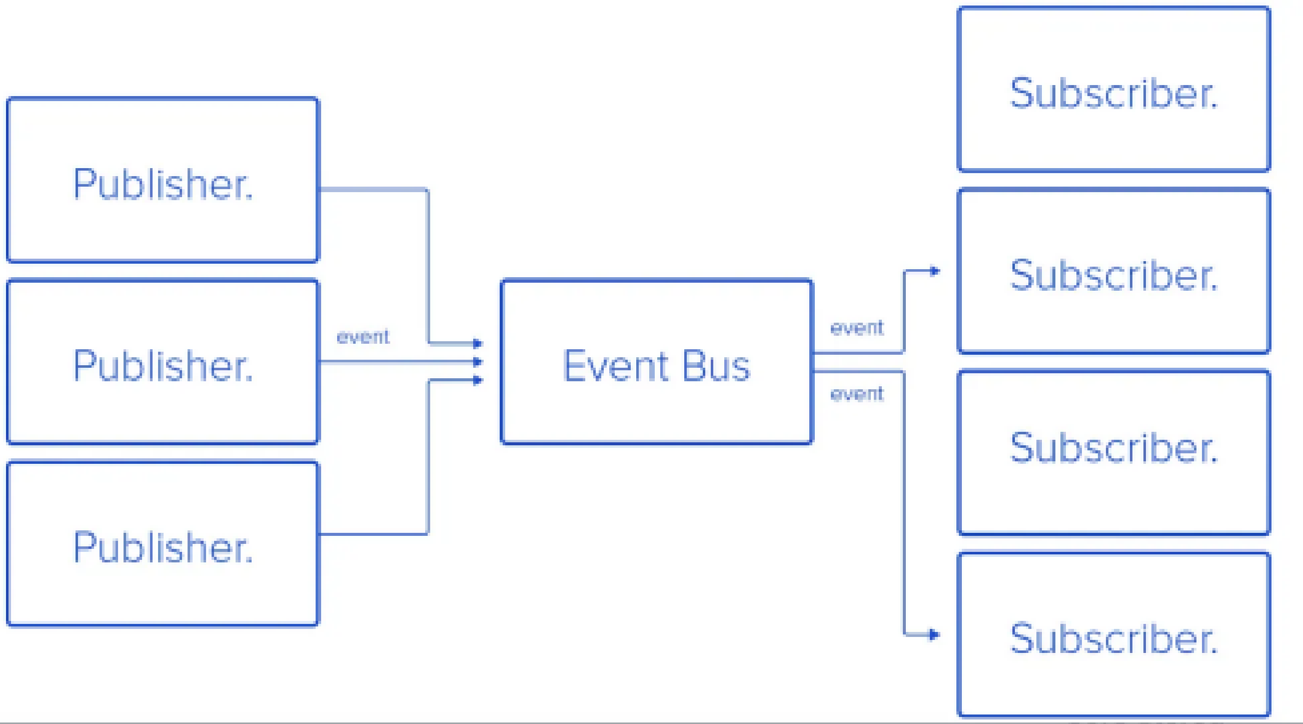
introduced.

Now, if one service gets high rps then it can scale its tasks more independent of the other services.

Also, if any service goes down the entire app/product will not get impacted

Different Patterns

Pub-Sub model (Publisher Subscriber Model)



It is a kind of middleware or message broker where publisher (service) publishes an event to a shared resource (having a topic or channel) which is scalable and fault tolerant and a subscriber (service) subscribes to a topic and consumes the message from the queue or any shared Data structure or implementation.

It helps us to make our architecture scalable, decoupled and flexible

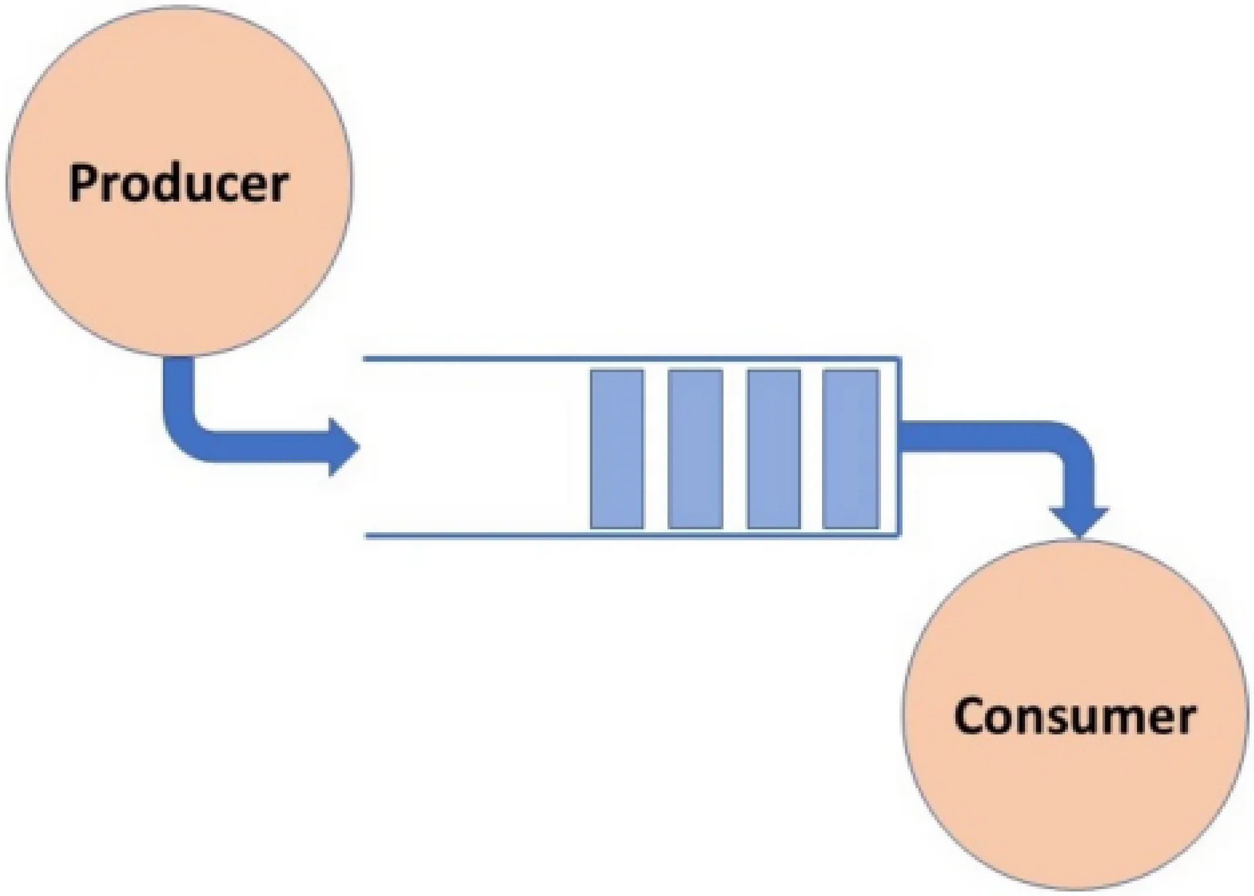
It is scalable and fault tolerant
In Pub-Sub model message broker maintains a registry of who produced the event to which topic and to which subscriber subscribed to a channel or topic, or in other words which are the subs that subscribed to a particular topic and which all are the producers that are producing the events/messages to a particular topic

Publisher publishes without caring who has subscribed for its message

Subscriber consumes without caring who published it to the topic

This pattern is well-suited for scenarios where multiple subscribers need to receive the same message, such as broadcasting events to multiple listeners or implementing fan out messaging.
ex: Kafka

Producer Consumer Model



Here, Publisher Subscriber model is also similar to PubSub but

In the producer-consumer pattern, each message is typically consumed by only one consumer.

Producers enqueue messages into a shared queue, and multiple consumers compete to dequeue messages from the queue.

Each message is processed by only one consumer, ensuring that work is divided among consumers without duplication. This pattern is useful for scenarios where multiple consumers need to process different messages independently, such as task distribution or load balancing.

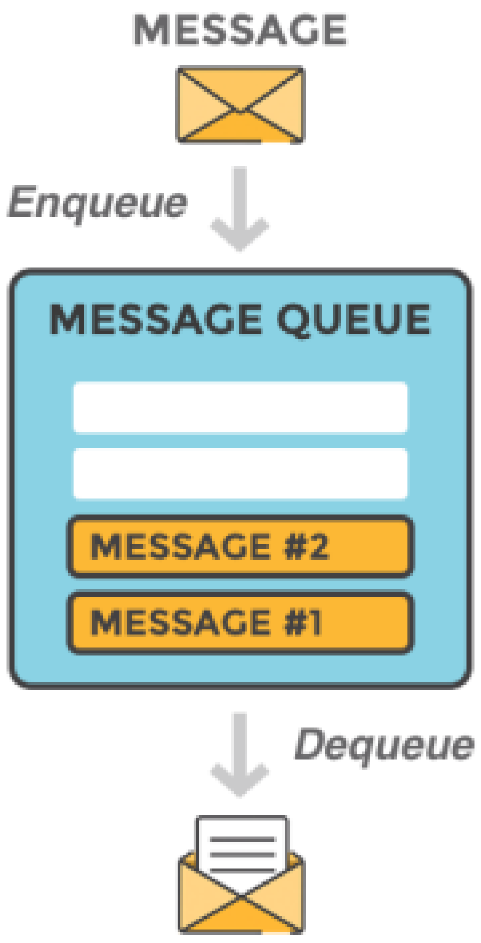
Conclusion:-

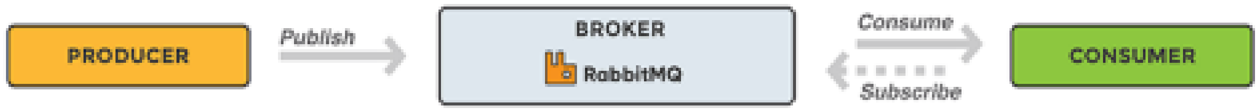
Use publisher-subscriber when you want to **broadcast** data to many consumers without the publisher knowing the identity or number of consumers, and when the data can be processed synchronously.

Use producer-consumer when you want to **distribute** data to one or a few consumers.

ex: RabbitMQ

RabbitMQ

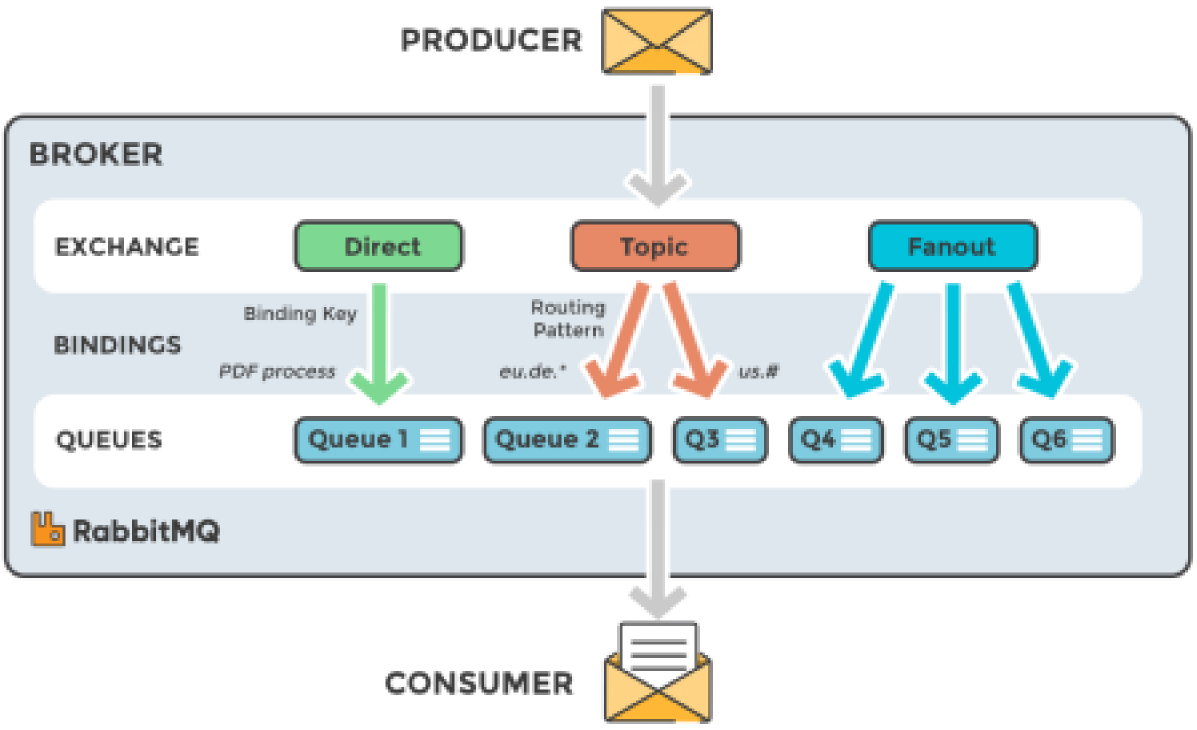




RabbitMQ Exchanges

Direct Exchange:- Messages will be published with a specific routing key to an exchange and it will get routed to the queue for which that routing key will match. And the queue's keys are called binding keys. If any message is being sent with the routing key "abc" then it would be sent to the queue having binding key as "abc". Here messages are routed according to the routing keys.

Fanout:- Here in this case message is broadcasted to every queue associated with that message broker. It doesn't use binding and routing keys to match and send the messages.
Topic exchange:- It is the case where routing key is matched with the pattern instead of direct matching. It is like regex matching only where wild characters are used for ex:- (*) and (#)
Header exchange:- Here basically headers of the messages is used to route the message to a queue.



RabbitMQ components:-

Producer:- It is a service or your code which publishes a message

Consumer:- It is a service again that consumes and responds to the message

Message:- A piece of information being sent from producer to consumer

TCP Connection:- This is the connection that is established between the services and broker to communicate with each other

Exchanges:- It is responsible of taking message from the producer and then publishing them to a specific queue based on different rules/protocols defined.

Binding:- Pathway that is established between a Exchange and queue.

Routing key:- Some key that is used to determine the destination to which a message to be delivered by the exchange.

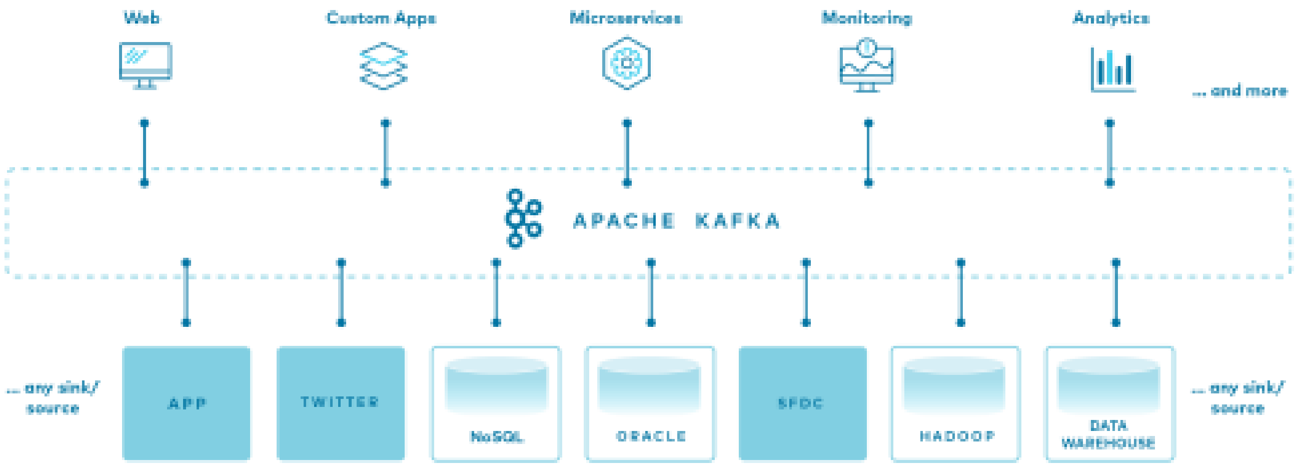
AMQP:- It is an advanced messaging queue protocol that is being used by RabbitMQ.

Access:- It is used to give access to different users for read write etc. It might use username and password for that

Ref:- *[https://www.linkedin.com/pulse/rabbitmq-features-architecture huzaifa-asif/](https://www.linkedin.com/pulse/rabbitmq-features-architecture-huzaifa-asif/)*

Kafka

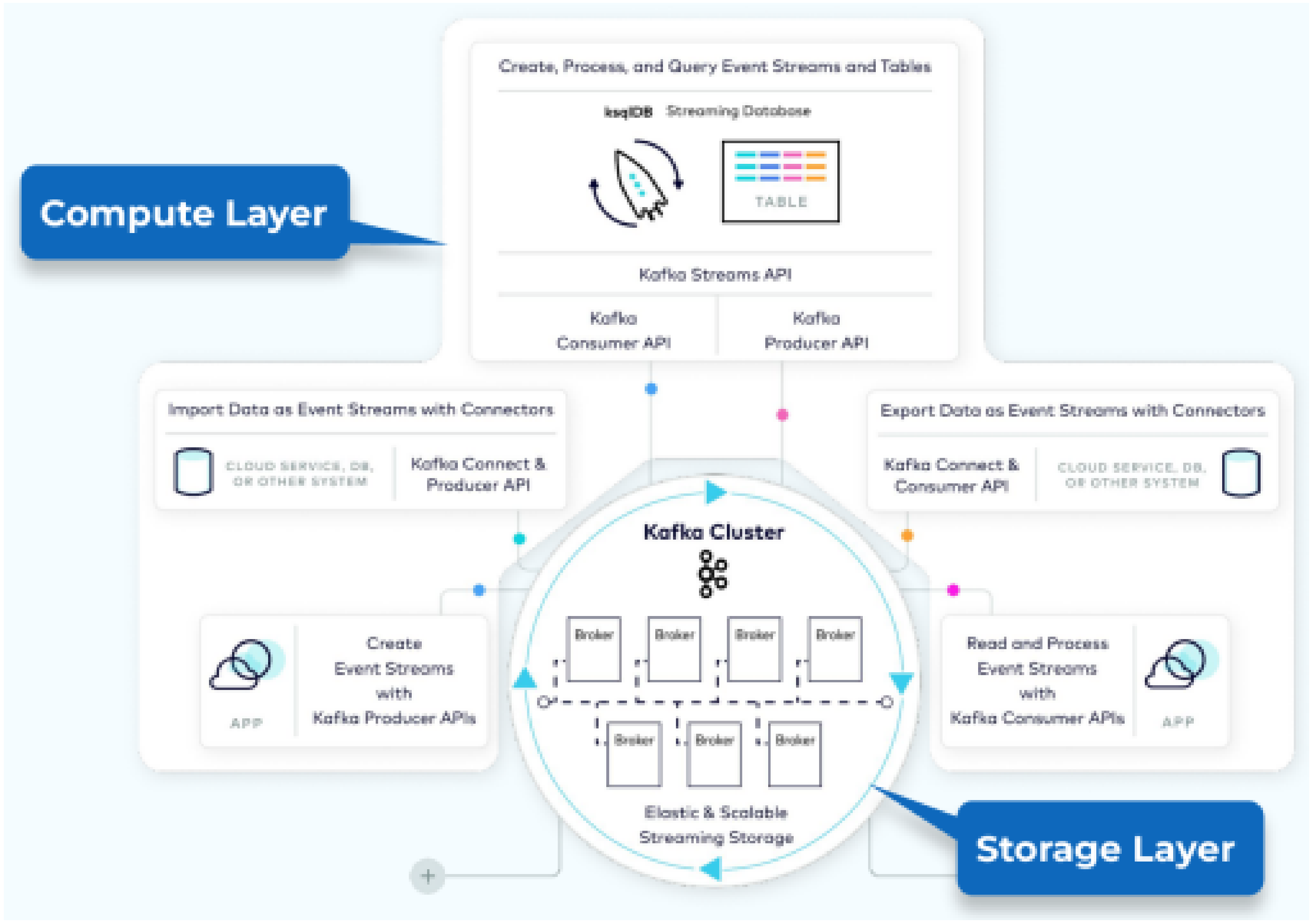
Ref:- https://developer.confluent.io/what-is-apache-kafka/



It is an event/data streaming platform and can be used for multiple purposes.

Event:-

An event is a trigger which can be anything either a button click, completion of some code etc and it can carry a small info most of the time alongwith it like in the form of JSON most of the time.


Compute Layer
Create, Process, and Query Event Streams and Tables
ksqlDB Streaming Database
TABLE
Kafka Streams API
Kafka Consumer API
Kafka Producer API
Import Data as Event Streams with Connectors
CLOUD SERVICE, DB, OR OTHER SYSTEM
Kafka Connect & Producer API
Export Data as Event Streams with Connectors
Kafka Connect & Consumer API
CLOUD SERVICE, DB, OR OTHER SYSTEM
Kafka Cluster
Broker
APP
Create Event Streams with Kafka Producer APIs
Read and Process Event Streams with Kafka Consumer APIs
APP
Elastic & Scalable Streaming Storage
Storage Layer

It has two layers:- One is compute layer and other one is storage layer.

Compute layer consists of four components which are producer, consumer, streams and connectorAPIs.

Storage Layer is responsible of storing data effectively so that it can scale according to the needs.

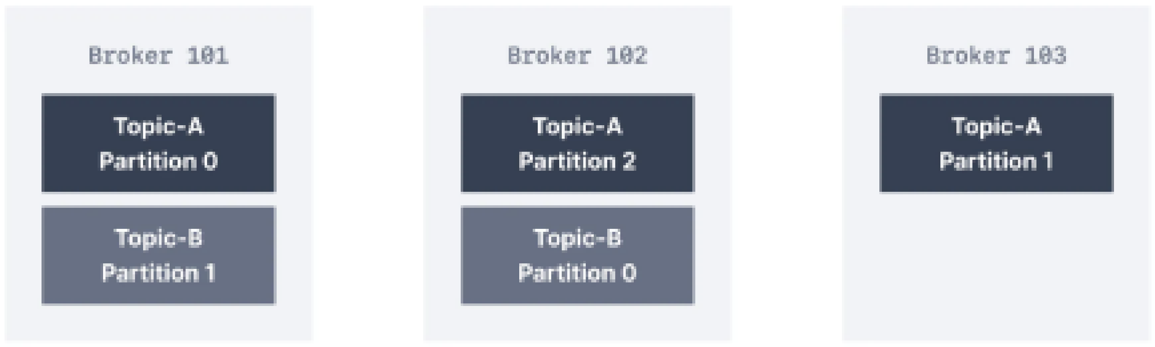
Kafka components:-

Kafka Topics:- They are abstracted containers having logs data structure which is append only. whenever we write some message to any topic that is written/appended at the end of the log and while reading an offset needs to be maintained. These logs enable us to deliver throughput in and out as high as possible. Topics are stored on the disc as files only. There is nothing temporary in kafka as it doesn't initializes a temp buffer like most of the message brokers do between src and dst. Kafka Partitioning:- As topics have logs which are stored on the disk itself and they couldn't be stored in a single node because it would limit the scalability. Hence topic can be partitioned into multiple logs which can live on multiple nodes and hence scalable.

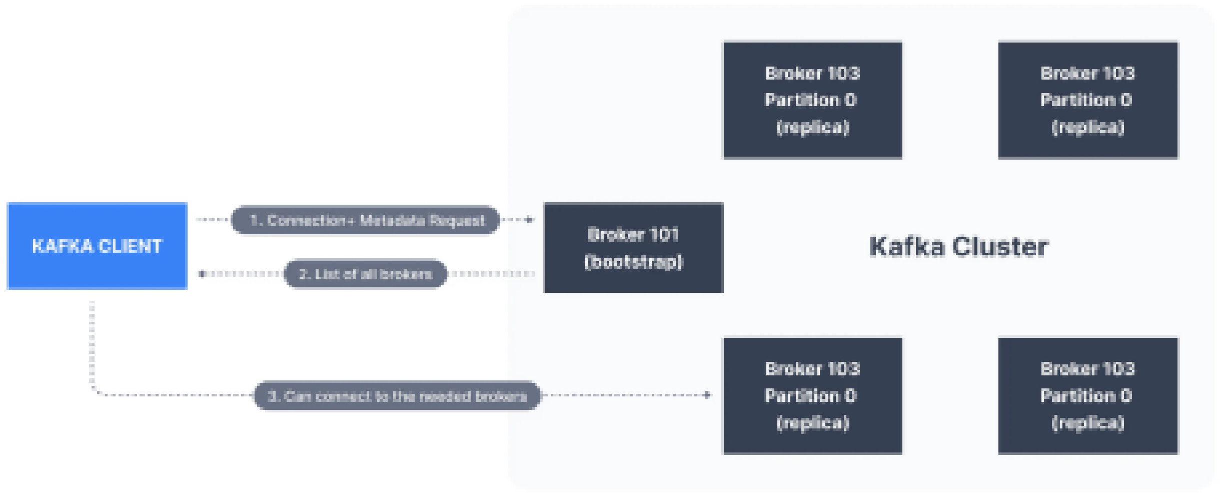
How Kafka Partitioning works:-

when a topic is partitioned into many partitions then how kafka will decide that messages will go to which partition? It depends if message has any ID. If it has any id then it will go with according to that key and otherwise it will go in round robin fashion and each partition will get even distribution of the Data.

Kafka Brokers:-



Kafka Topic Partitions



Connecting to a Kafka Cluster

Each broker is a mini server that includes some partitions and topics. These brokers can be discovered from their IDs and once a broker is discovered all of the kafka cluster will be accessible.

What happens when we scale kafka running in docker containers:-

> When we scale any kafka service it will add new broker instances and load will be distributed among the existing ones and the new brokers. This leads to high throughput and fault tolerance.

Kafka Replication:- Whether kafka instances are running in multiple containers but they are prone to failure. So, to prevent that kafka has leader and follower mechanism. when one instance is about to die then the other one will become leader. Writes and reads happens on the leader but leader always take follower together.

Kafka producers:- There is a class `KafkaProducer` which accepts the configs and all to connect to any kafka cluster and there is an another class `ProducerRecord` that you use to hold the key value pair that you want to send.

Kafka consumer:- Similarly we have `KafkaConsumer` which passes config map to connect to the cluster and all. When messages are present then they come from the topic in `ConsumerRecords` which includes individual instances of messages in `ConsumerRecord` objects.

Consumer Groups:- Kafka can have multiple consumers and it works on consumer group level as:-

1. **Multiple Partitions Consumed by Multiple Consumer Groups**:

   Yes, multiple partitions can be consumed by multiple consumer groups simultaneously.

   Each consumer group can have multiple consumers, and each consumer within a group can be assigned to one or more partitions.

   Kafka ensures that each partition is consumed by exactly one consumer within each consumer group.

   Different consumer groups can consume the same set of partitions concurrently, allowing for parallel processing and scalability.

2. **Single Partition Consumed by Multiple Consumer Groups**:

No, a single partition cannot be consumed by multiple consumer groups simultaneously in Kafka.

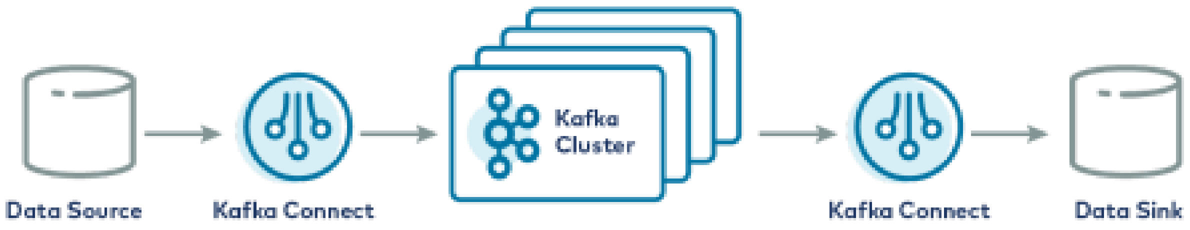
   Kafka follows the "one-consumer-per-partition" model, where each partition is exclusively assigned to one consumer within a consumer group.

If multiple consumer groups attempt to consume the same partition, Kafka ensures that only one consumer group is actively consuming from that partition while the others remain idle.

This design ensures that each message within a partition is processed by only one consumer group to maintain message ordering and prevent duplication.

Components of Kafka:-

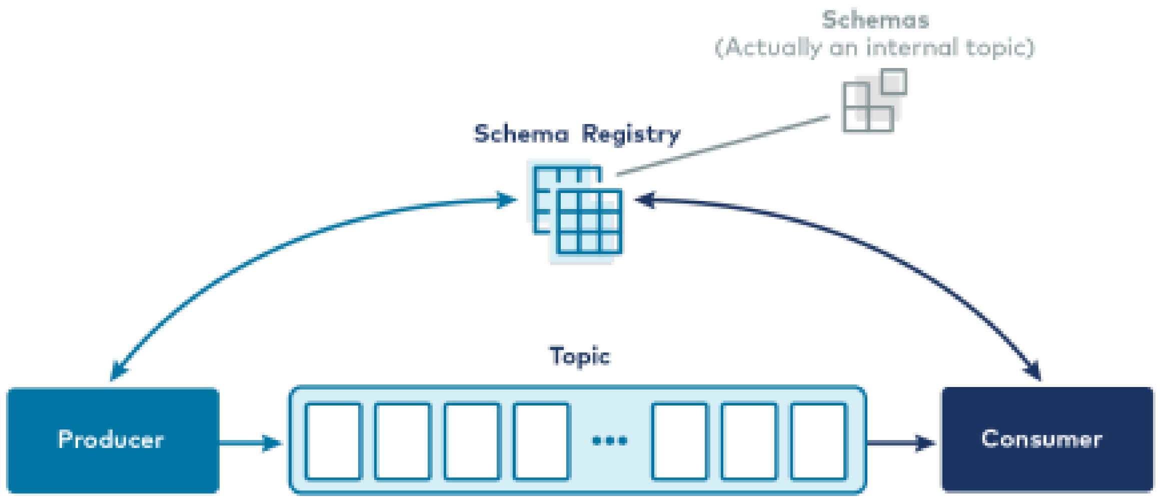
Kafka connect:-



Kafka can be used with any kind of data source or sink. For ex: Your application wants to send data from kafka to elasticSearch. It can easily do that using a simple JSON config file:-

```
{
 "connector.class":
"io.confluent.connect.elasticsearch.ElasticsearchSinkCo nnector",
 "topics" : "my_topic",
 "connection.url" : "http://elasticsearch:9200",  "type.name" :
"_doc",
 "key.ignore" : "true",
 "schema.ignore" : "true"
}
```

Schema Registry:-



It is a kind of Database which stores the schema and all like what is allowed to be stored inside topic. Schema registry is exposed as API to the services to which producer/consumer can call and validate if they can produce or consume certain type of messages.

Kafka Streams:-

Kafka Streams is a client library for building applications and microservices, where the input and output data are stored in Kafka clusters. It provides an abstraction for representing data streams as immutable, ordered sequences of events called "KStream" and "KTable". Operations like aggregations and Joins are supported out of the box for streams.

How to setup AWS VPS (Lightsail):-
https://www.youtube.com/watch?v=ENR-J6c9Xoo
make sure you setup the ubuntu user's password using

```
sudo passwd ubuntu
```

Installing Kafka in ubuntu 22LTS:-

https://vegastack.com/tutorials/how-to-install-apache-kafka-on
ubuntu-22-04/

```
sudo apt install openjdk-17-jre-headless

// give permission to kafka or any other user `<username>
ALL=(ALL) ALL`
// add this in (`sudo visudo`) (Download kafka not src tgz)
```

first start zookeeper using

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

Installing and configuring Prometheus and Grafana on Ubuntu

https://www.fosstechnix.com/install-prometheus-and-grafana-on
ubuntu-22-04/

After installing, change below:-

allow every connection to connect to prometheus

`sudo nano /etc/prometheus/prometheus.yml`

`sudo iptables -A INPUT -p tcp --dport 9090 -j ACCEPT` Add prometheus in build gradle

Add /metric endpoint in application properties

Add

```
scrape_configs:
 - job_name: 'spring-boot'
 metrics_path: '/actuator/prometheus'
 static_configs:
 - targets:
 ['your_spring_boot_host:your_spring_boot_port']
```

allow every connection to connect to prometheus

`/etc/grafana/grafana.ini`

`sudo iptables -A INPUT -p tcp --dport 3000 -j ACCEPT` disable the firewall

add your public IP in IPv4 of AWS lightsail to allow you to connect it to, Talk about TCP connection

/path/to/kafka/config/server.properties

```
# Set the host name or IP address for the broker to listen on
listeners=PLAINTEXT://0.0.0.0:9092 # Set the advertised host name or IP
address to be published to clients
advertised.listeners=PLAINTEXT://your_public_ip:9092
```

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