Kafka Python Tutorial

Documentation

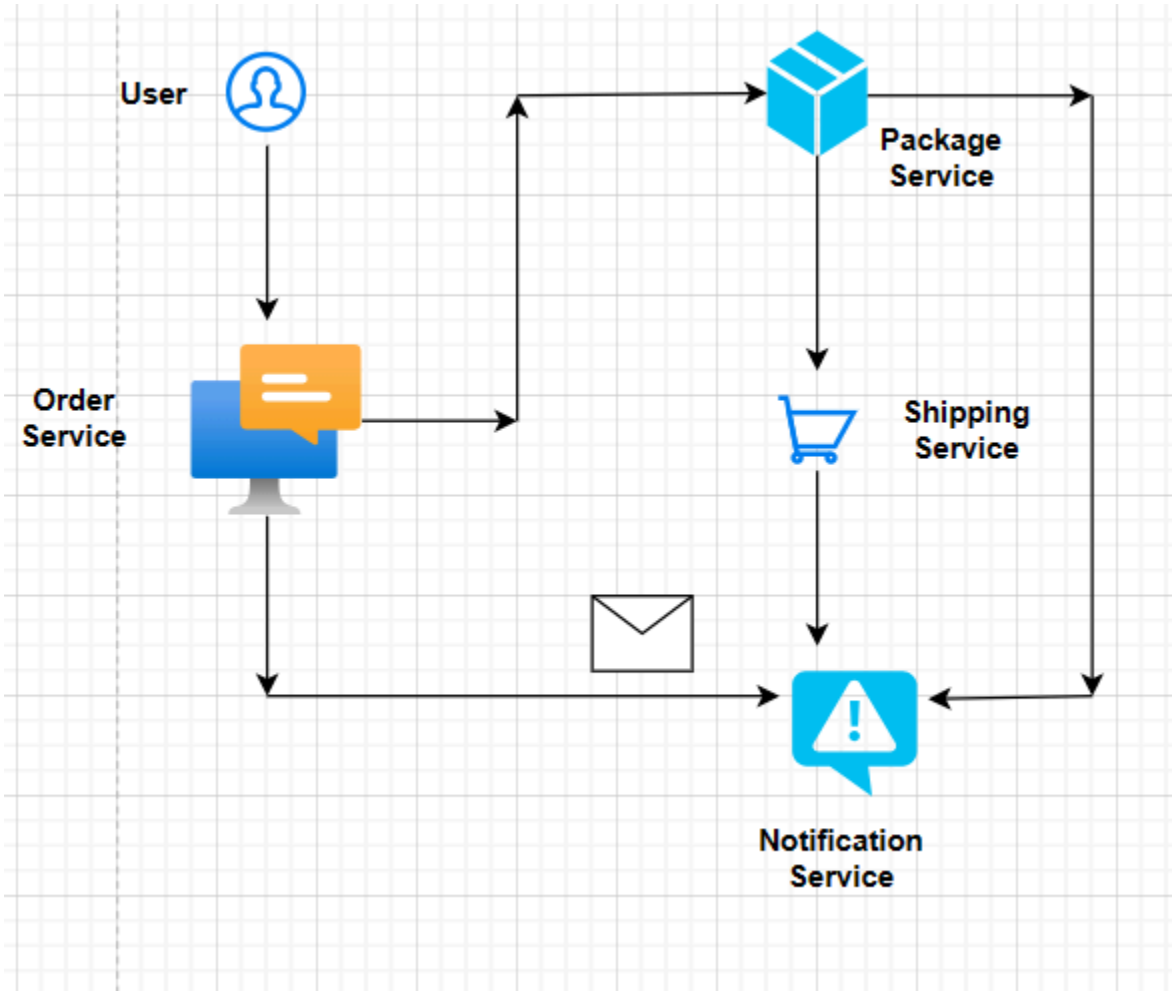
https://developer.confluent.io/get-started/python/#introduction
https://docs.confluent.io/platform/current/clients/confluent-kafka-python/html/index.html#
https://github.com/confluentinc/librdkafka/blob/master/CONFIGURATION.md (Librdkafka Configuration)
https://docs.confluent.io/platform/current/installation/configuration/producer-configs.html (Producer Configuration)

Tutorial-1

Why do we need Kafka, What Is the Problem statement which Kafka Solved and What is Kafka (Brief Overview) ?

Use Cases - https://www.confluent.io/en-gb/learn/apache-kafka-benefits-and-use-cases/#kafka-use-cases

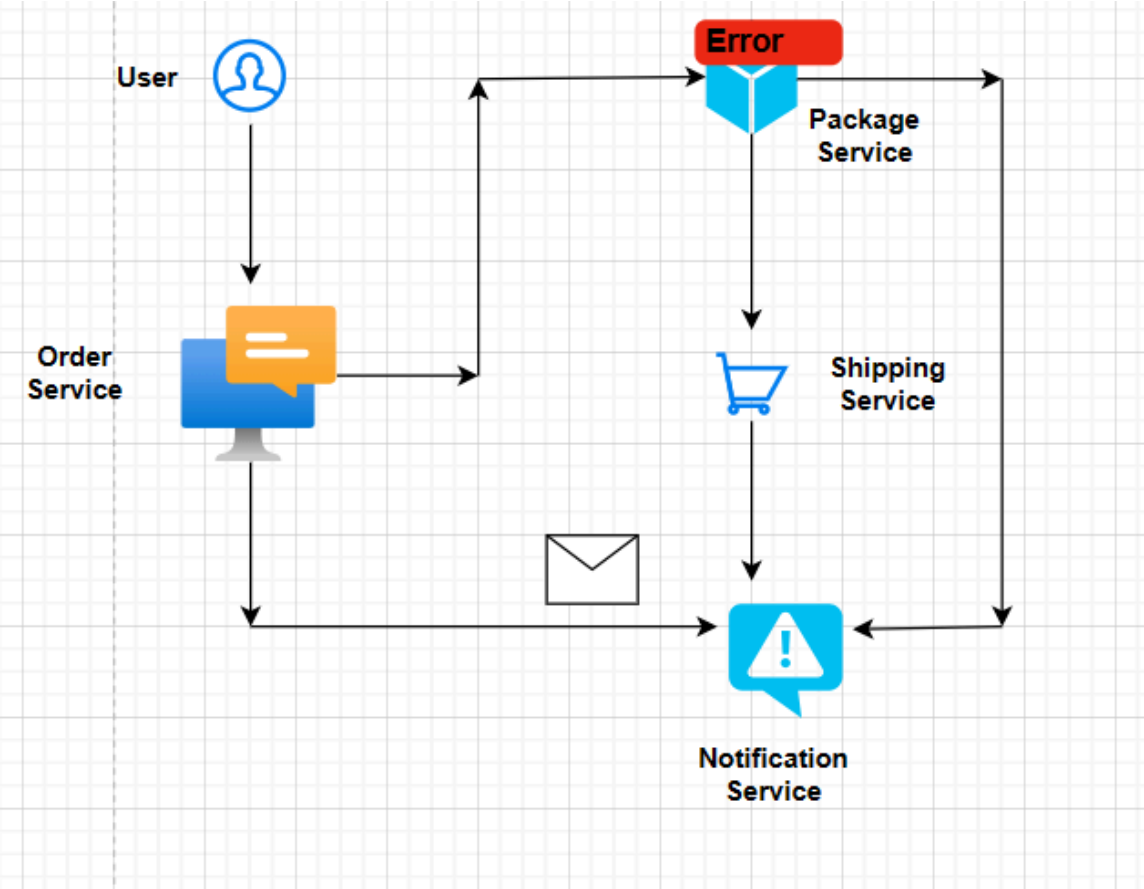
- **Problem-1**: Tightly Coupled Microservices (Example E-commerce) - **Retail Sector**



- User will place the Order.
- Order Service will process the Order
- and notify to the Package Service
- Package Service will package the order and notify to the shipping Service
- Shipping Service will add the Label and Shipped and notify to the notification Service
- Notification Service will send the message to the user.
- Along with that, when the Order get Placed, Order Service also send the message to the notification service to send the updated information to the customer like Order is placed, Order has been packed.

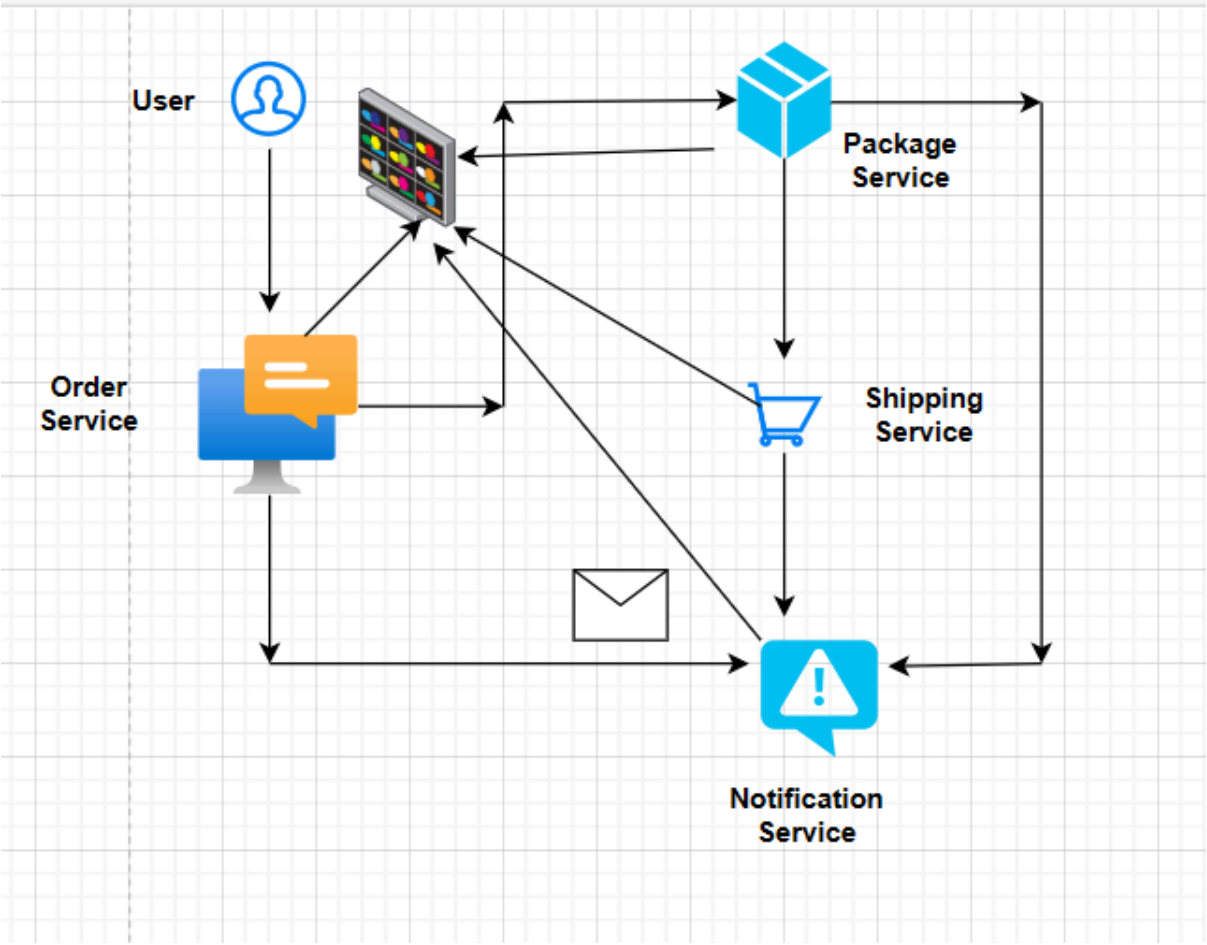
Problems with above System

What is Package Service Failed ?



So we need some kind of retry mechanism , this can be done, but needs a lot of work.

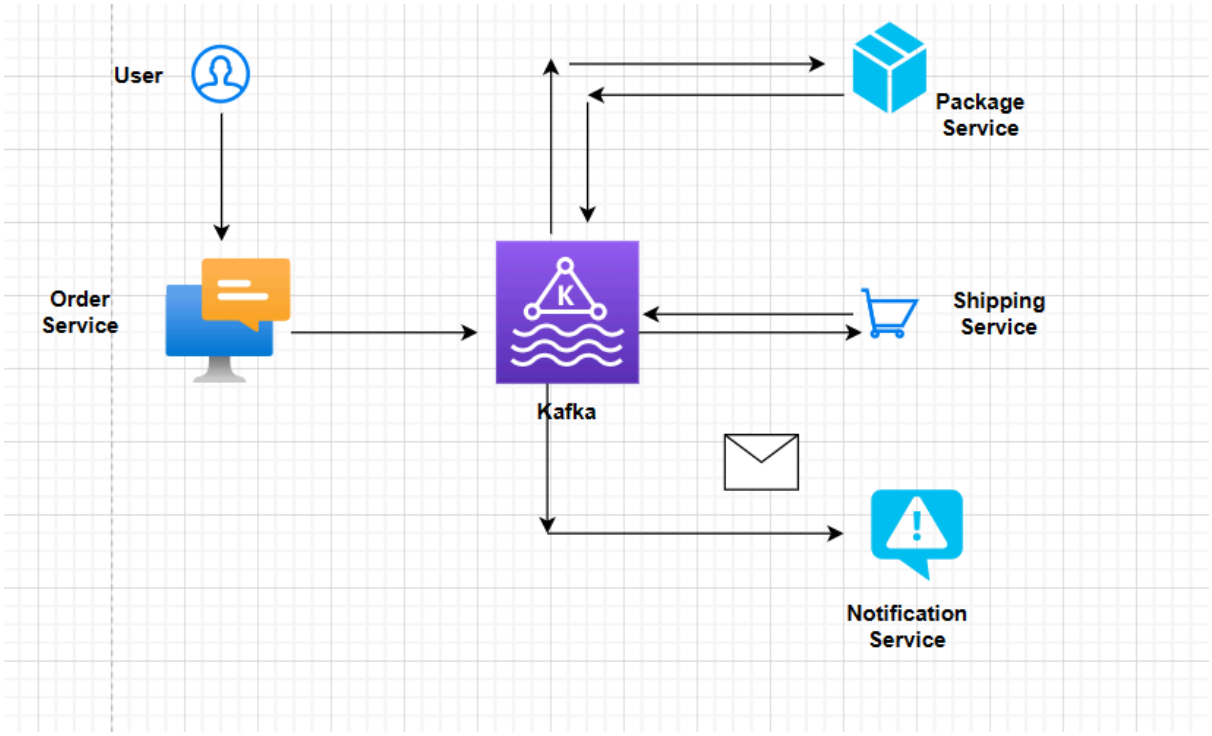
What happens if a new service is introduced that logs all these transactions to a central store.



The monitoring service needs to interface with all the other services, So this requires a code change in all the other services. This makes the developer work very hard.
Testing will also be hard, we need more mock services to test each individual service.

Example, if I need to test my package service, because it is dependent on Order, Shipping and Notification service , I need to mock up all these just to test my packaging service.

Solution: Solved by Kafka Using Decoupling.



Now the order service instead of directly sending the message to the packaging service, will send these messages to the kafka where they will be stored.

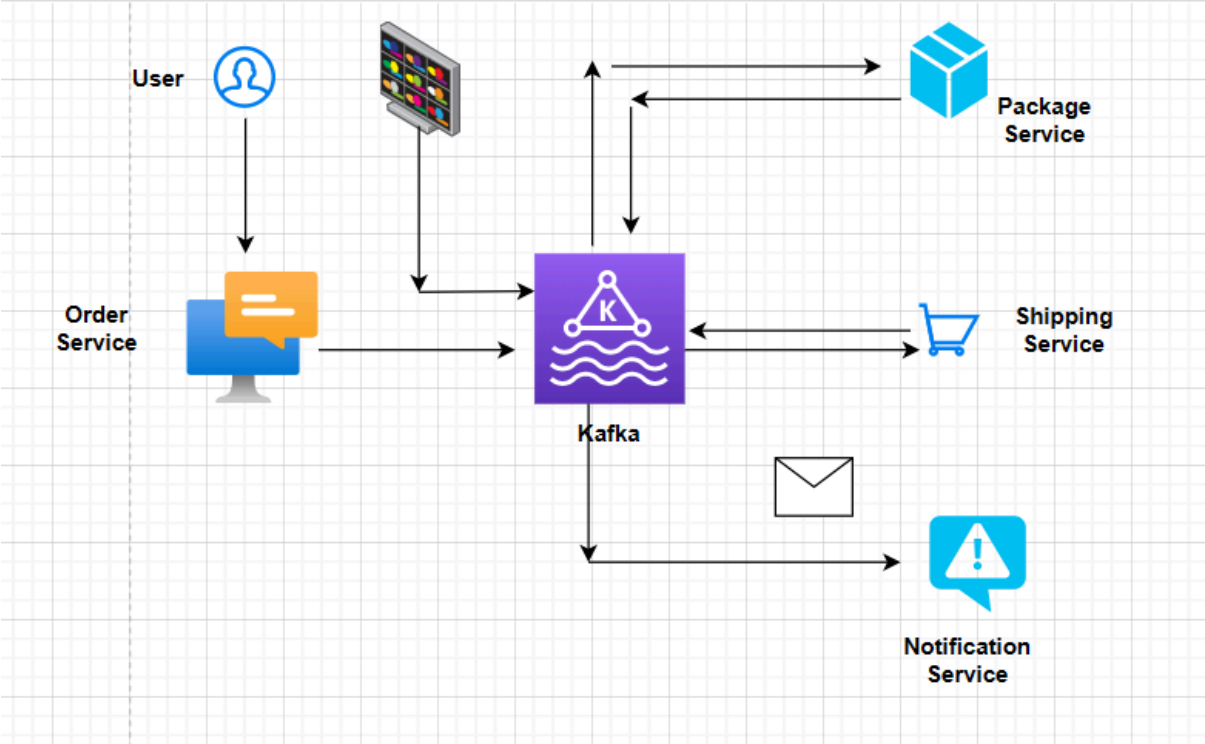
The packaging service and the notification service will receive these messages either by a push or pull.

Package Service Instead of directly sending the message to the Shipping Service, it will send to Kafka and the event will be received by the Shipping and the Notification Service.

Package Services need not need to know which services are receiving the events.

Similarly Shipping service will send the message to Kafka.

Even if any new service will come, all it need to do just subscribe the messages from the existing services



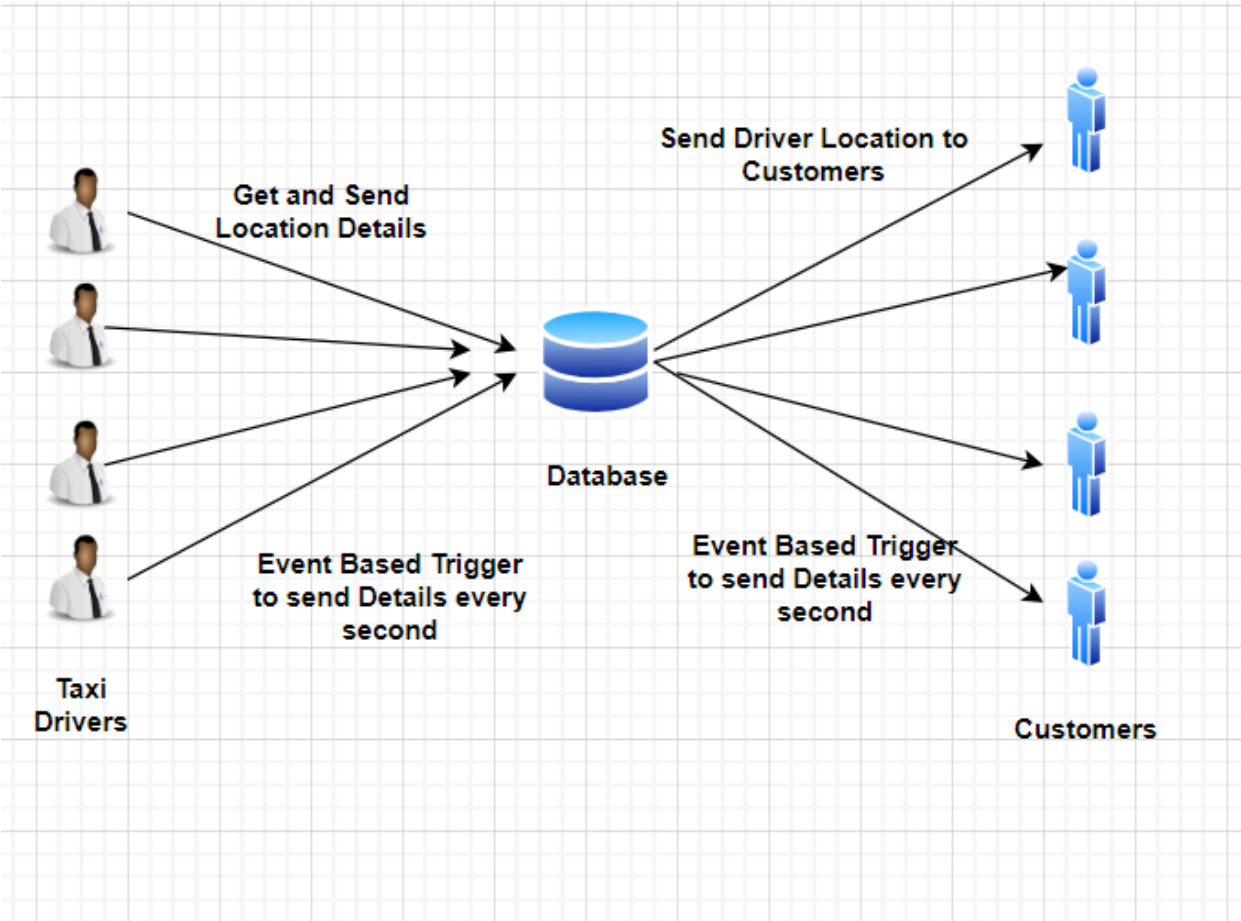
So, now if any service fail, messages get store in Kafka (for particular number of days) or until the service will come back

You may ask, instead of storing messages into Kafka or messaging systems, we can store them in a Database.

And all other microservices will get the data from that Database.

There is a difference, In messaging system, as soon as the data comes into the messaging system , other microservices get notified.

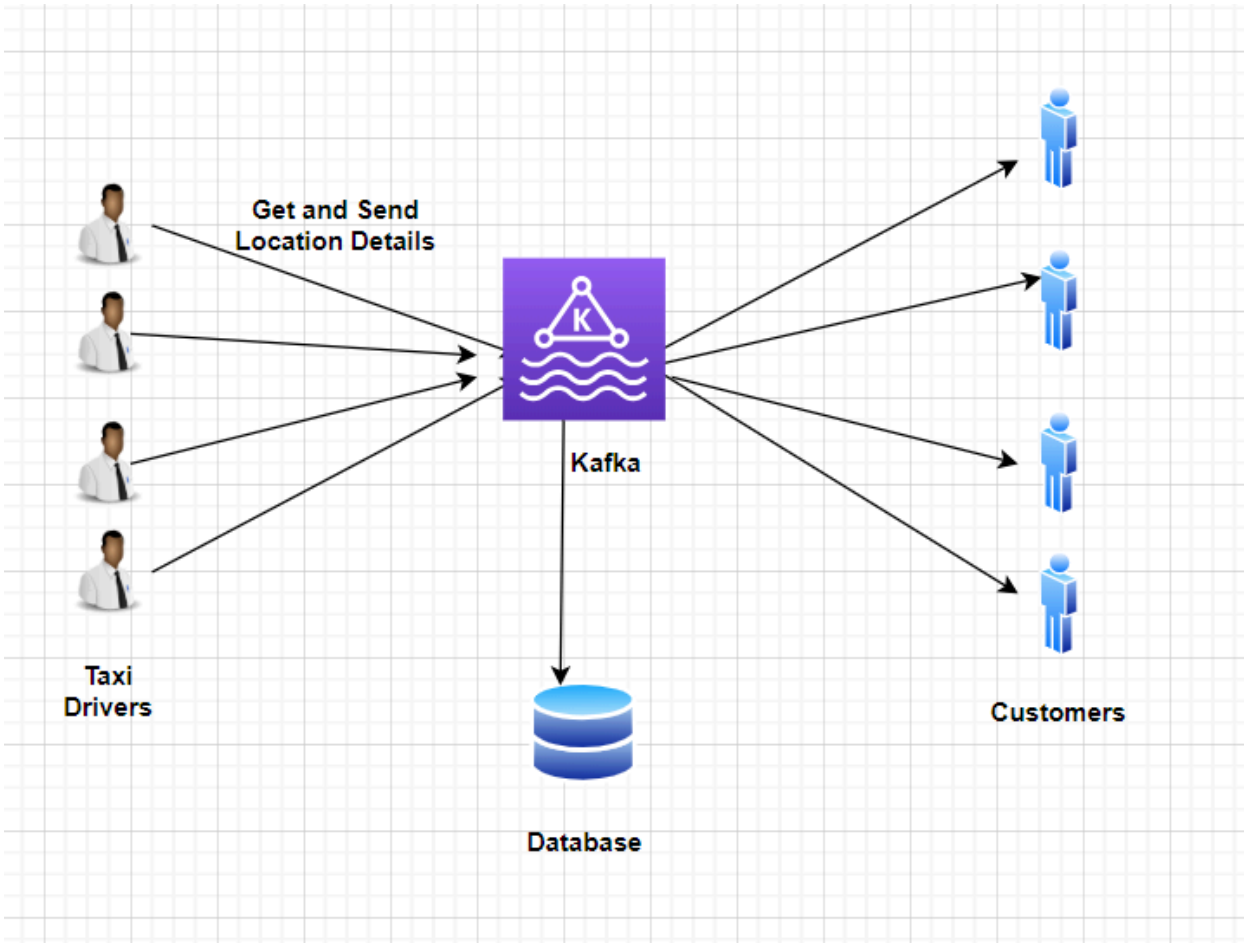
Problem-2: Low Throughput (Transactions per seconds) on Database (Example -scenarios where frequent updates, like driver location updates in a taxi service app, are required). if the database is overloaded with write requests, it can slow down the read requests and affect the user experience.if the database is not designed to handle such high throughput, it can become a bottleneck and degrade the performance of the application (**Transportation Sector**)



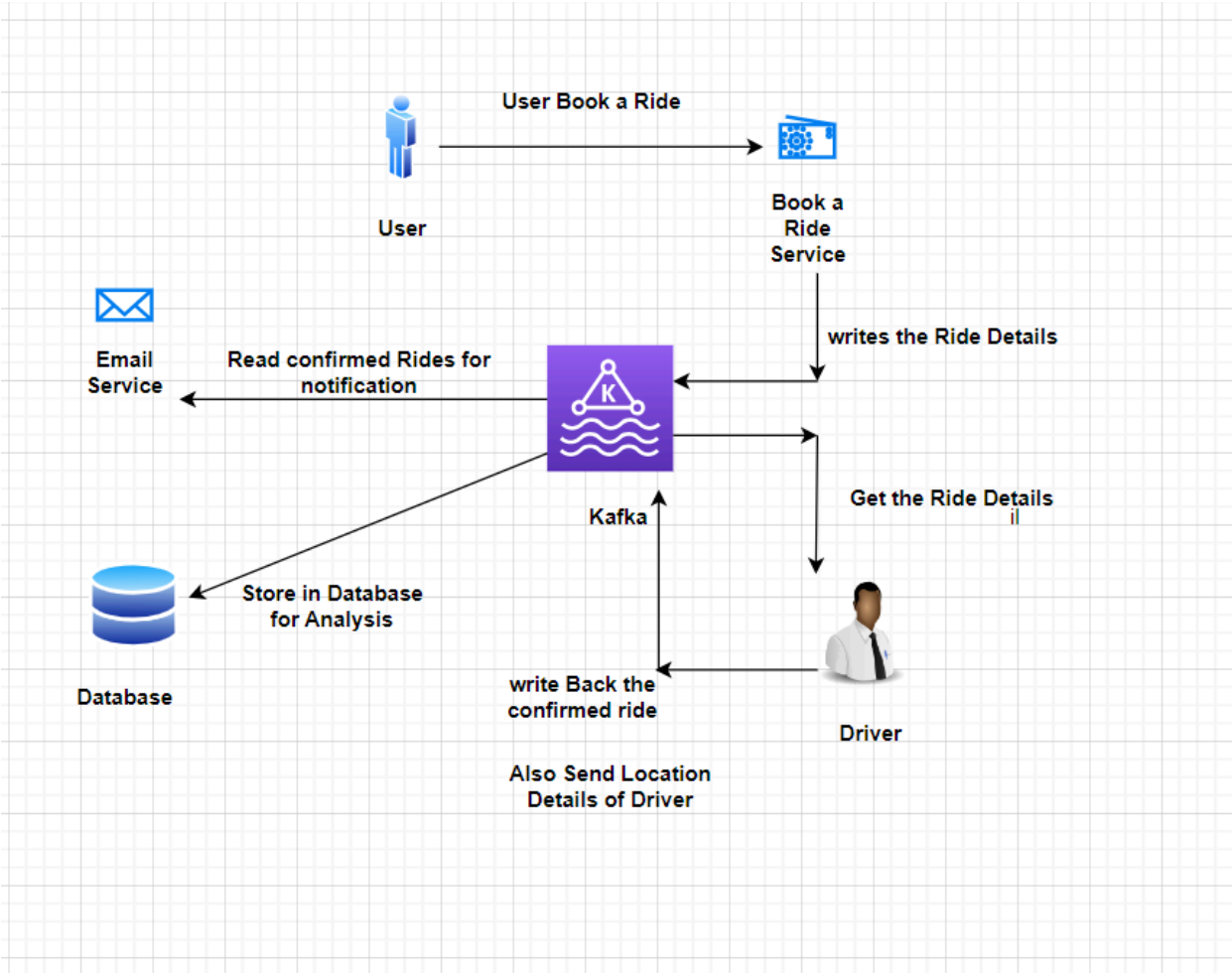
- Real-time updates: Customers may not see the location of their driver in real time, as there is a one-second delay between updates.

- Database bottleneck: The database could become a bottleneck, as it is responsible for storing and retrieving all of the information about the taxi service.

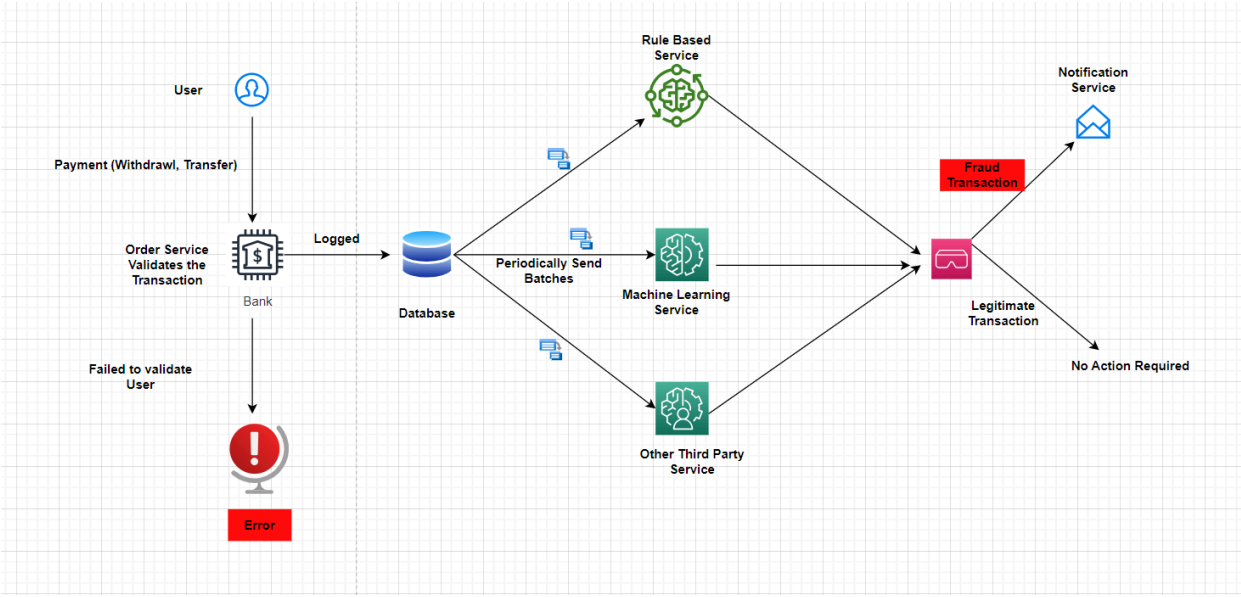
Solution: Solved by Kafka by Providing High Throughput



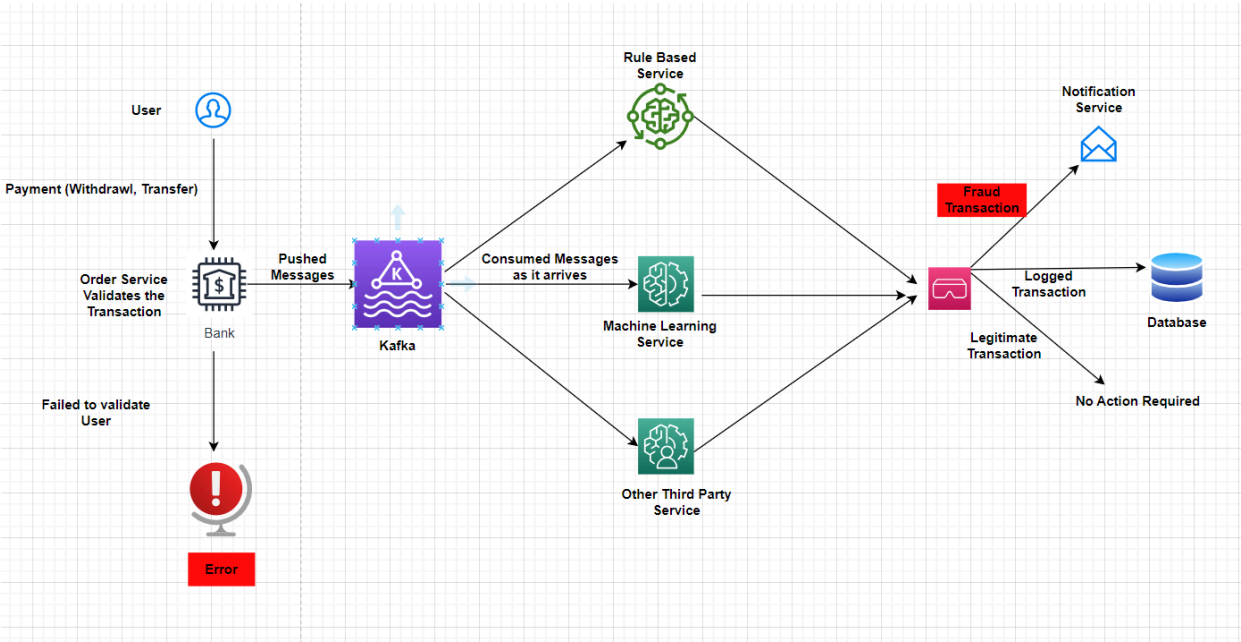
More Specific Diagram - with other Services



Problem-3: Banking (Fraud Detection) (**Banking Sector**)



Solution: Solved by Kafka (As it's Real Time Event Processing) - Real Time Fraud Detection



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

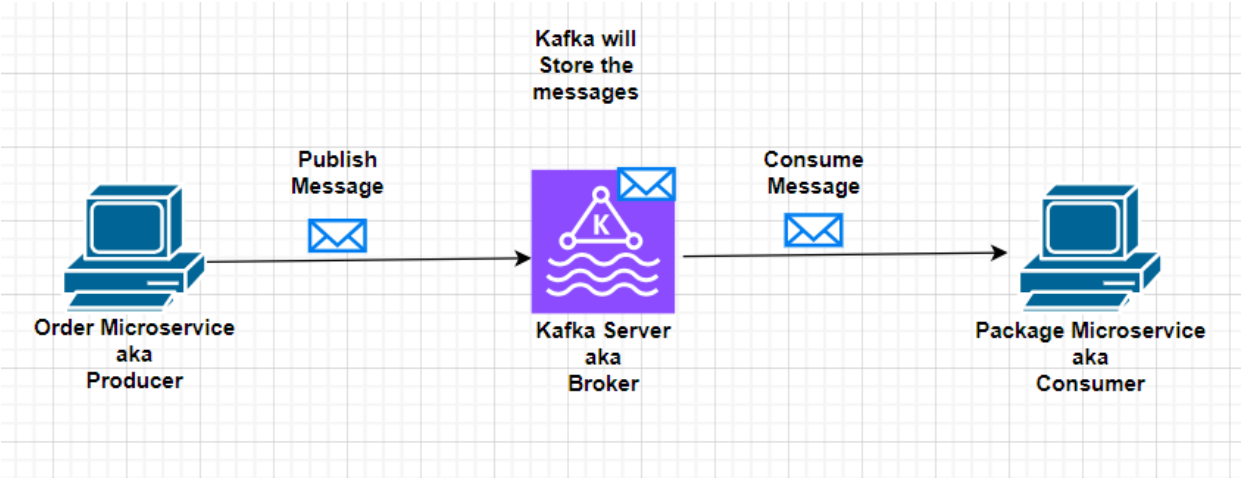
- Publish-Subscribe messaging system to handle mass amount of data.

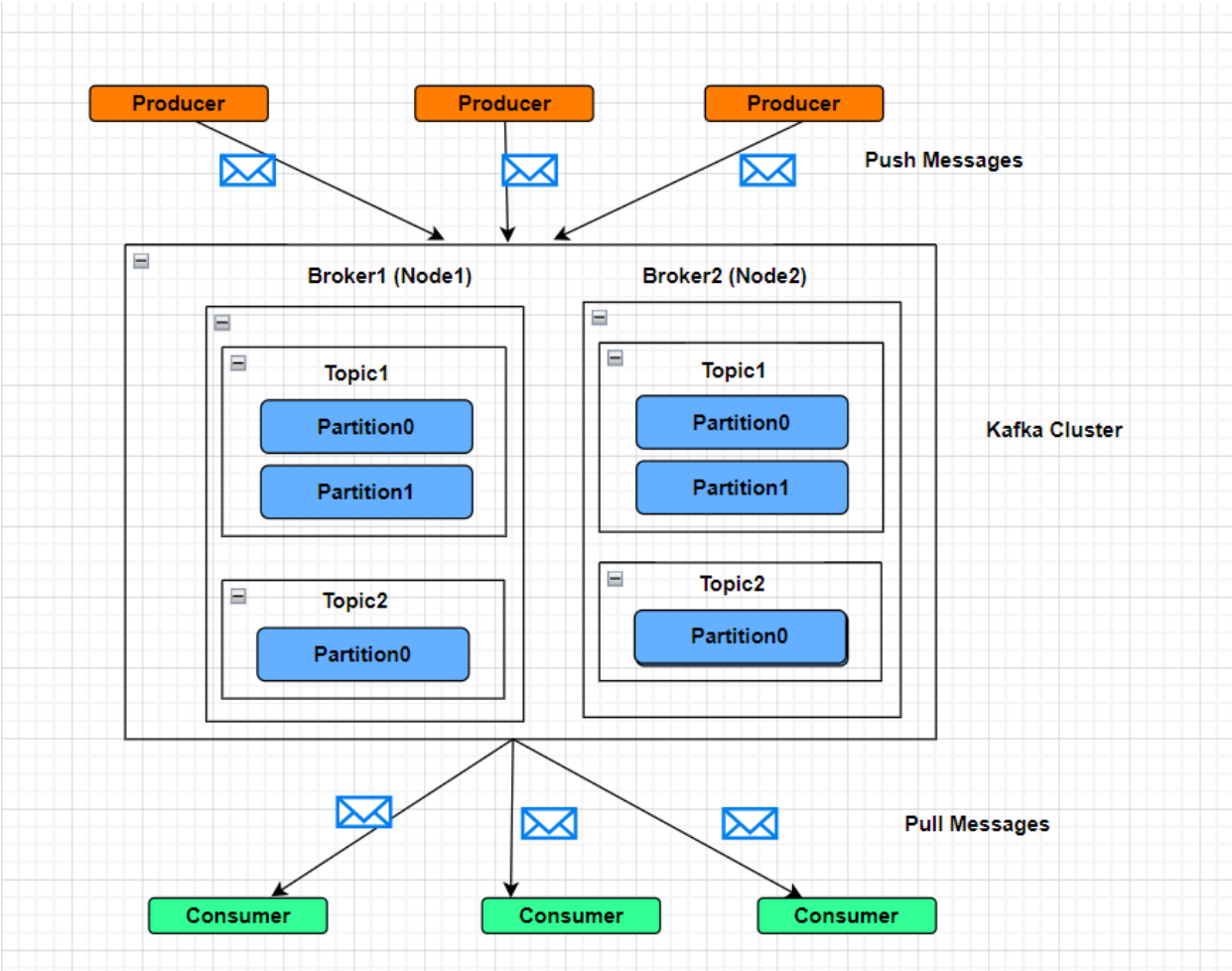
- Event Streaming Platform (i.e. continuous and real-time flow of events or messages between different systems and applications)

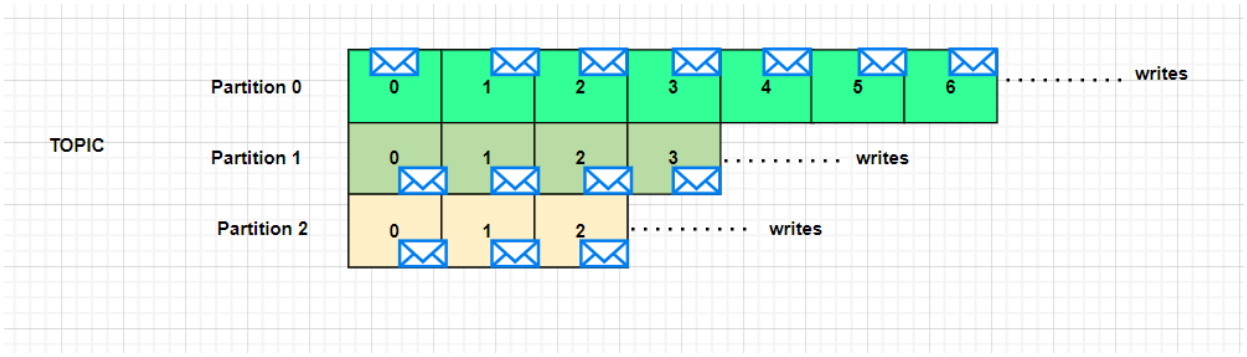
Tutorial-2

Kafka Components & Architecture (Basic Overview)

- **Producer -** is an application that send messages to kafka topic
- **Consumer -** is an application that reads messages from kafka topic
- **Broker -** is a server instance that stores messages published to kafka topics.
- **Topic -** A named category of messages. Producers publish messages to topics and consumers subscribe to topics to receive messages. Topic represents a stream of data. (Same as table in a Database)
- **Cluster -** is a group of kafka brokers (Multiple broker will distribute the load during peak demands)
- **Partitions -** Sub-division of a topic within a cluster, each message in a topic belongs to a specific partition.
- **Offset -** Every message published will get a unique ID called offset. A numerical value representing the position of a consumer within a partition.
- **Consumer Groups -** A set of consumers working together to collaboratively consume messages from topics (For parallel processing)
- **Zookeeper (Not Required in latest version) -** Required in older version of Kafka - used for cluster coordination and storing offset for consumer groups. Modern versions eliminate the need of zookeeper.


Kafka will
Store the
messages
Publish
Message
Consume
Message
Order Microservice
aka
Producer
Kafka Server
aka
Broker
Package Microservice
aka
Consumer


Producer
Producer
Producer
Push Messages
Broker1 (Node1)
Broker2 (Node2)
Topic1
Partition0
Partition1
Topic2
Partition0
Topic1
Partition0
Partition1
Topic2
Partition0
Kafka Cluster
Pull Messages
Consumer
Consumer
Consumer



Tutorial-3

Install Redpanda (With Docker Compose).

https://redpanda.com/

Redpanda: Redpanda is a simple , powerful and cost-efficient streaming data platform that is compatible with Kafka API
https://hub.docker.com/r/redpandadata/redpanda
https://docs.redpanda.com/current/get-started/quick-start/ (Docker-Compose file Syntax)
https://docs.redpanda.com/current/reference/docker-compose/ (Sample Application Docker-Compose)

Understand All parameters

Smp : simultaneous multiprocessing, specify the number of worker threads used by the kafka broker for processing requests.

Overprovisioned : When enabled, Redpanda can allocate more memory than physically available on the system using swap space. This can improve performance by allowing Redpanda to keep more data in memory for faster access.

kafka-addr : This address is used for internal broker communication i.e. broker to broker communication and itself communication (in single broker) i.e. internal communication

with itself like metadata exchange etc. So even if you setup single node cluster, this address is mandatory

Advertise-kafka-addr : This address is used by external clients to connect like Producers and Consumers.
https://redpanda.com/blog/advertised-kafka-address-explanation

Pandaproxy - Pandaproxy in Redpanda provides a RESTful API that helps you manage your Kafka topics and data in a more user-friendly way compared to the traditional Kafka protocol.

Pandaproxy-addr: Internal Redpanda REST API

https://redpanda.com/blog/pandaproxy

https://docs.redpanda.com/api/pandaproxy-rest/

Advertise-pandaproxy-addr: Used for clients connecting to the Pandaproxy API, an optional RESTful interface provided by Redpanda. External Redpanda REST API

localhost:8082/v1 - Rest API

https://redpanda.com/blog/pandaproxy

Schema-registry-addr : Schema can be considered as human-readable documentation for data. Schema-registry-addr in Redpanda defines the addresses where the Schema Registry service listens for connections. The Schema Registry plays a crucial role in managing and validating schemas for topics in your Redpanda cluster.

https://redpanda.com/blog/schema-registry-kafka-streaming

–rpc-addr : Redpanda brokers use the RPC API to communicate with each other internally. This address used for RPC (Remote Procedure Call)

Though both **–kafka-addr** and **—rpc-addr are used** for broker internal communication. But which one to use depends on specific communication needs.

Kafka-addr:

- When brokers need to exchange data related to topics, partitions, replication, and leader election.
- When a new broker joins the cluster, it communicates with existing brokers using their Kafka addresses

Rpc-addr:

- When brokers need to coordinate beyond Kafka-specific tasks (e.g., configuration updates, health checks, or custom management commands).
- Brokers might use RPC to exchange metadata, check the status of other brokers, or perform administrative actions.

rpk (redpanda keeper) - command line tool designed to manage your redpanda cluster

- **Get information about cluster**
  docker exec -it redpanda rpk cluster info
- **Create Topic**
  docker exec -it redpanda rpk topic create chat-room
- **Describe Topic**
  docker exec -it redpanda rpk topic describe chat-room
- **Produce Some message**
  docker exec -it redpanda rpk topic produce chat-room
- **Consume Message**
  docker exec -it redpanda rpk topic consume chat-room --num 1

Tutorial-4

Install Redpanda Console (with Docker-compose)

Redpanda Console: is a developer friendly UI for managing your Kafka/Redpanda workloads

https://github.com/redpanda-data/console

https://hub.docker.com/r/redpandadata/console
https://docs.redpanda.com/current/reference/docker-compose/ (Example Docker-compose file)

Console Config
https://docs.redpanda.com/current/reference/console/config/

Tutorial-5

Install Redpanda (With Three Brokers)

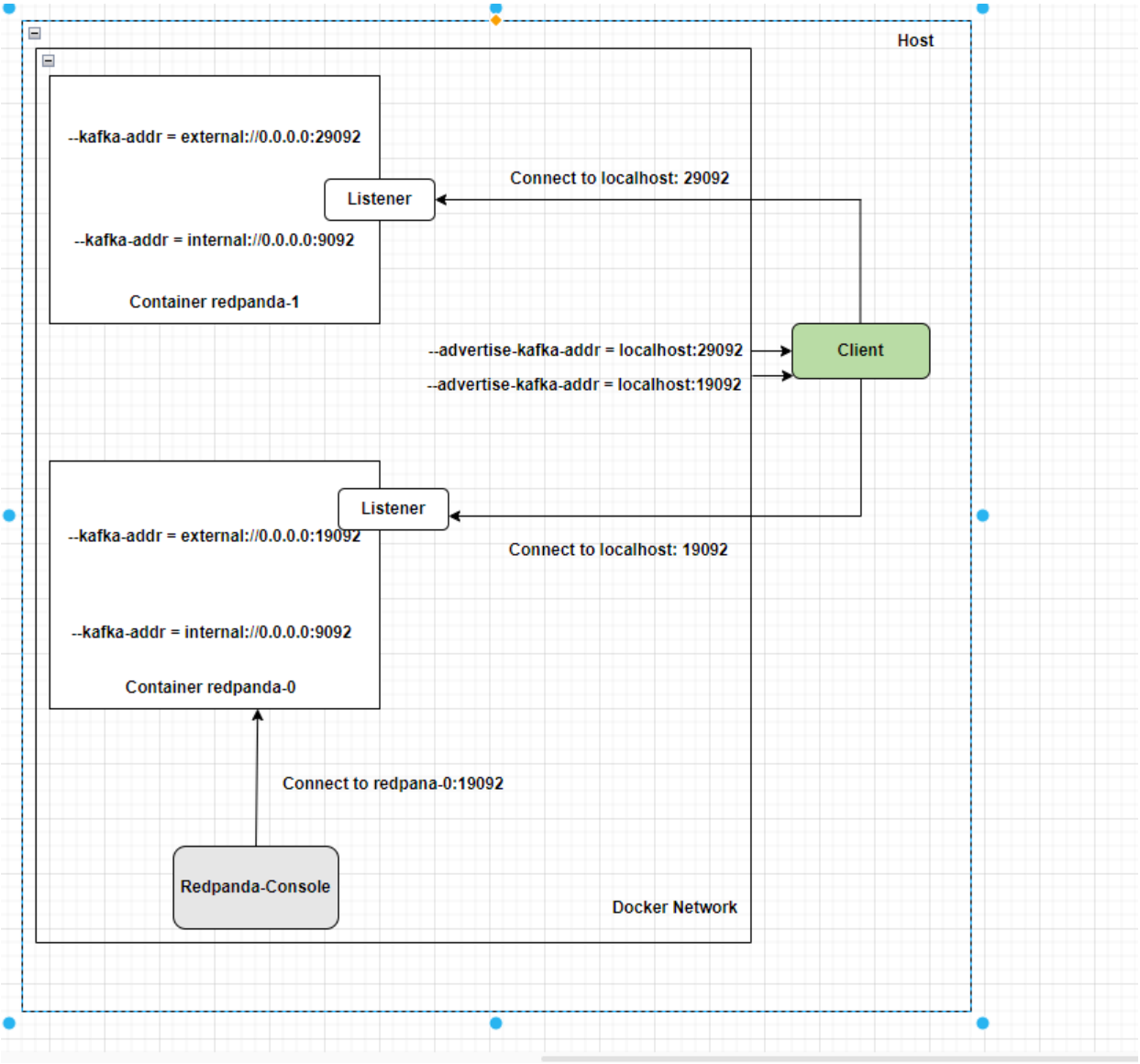
https://docs.redpanda.com/current/get-started/quick-start/?tab=tabs-1-three-brokers

https://github.com/Sumanshu-Nankana/kafka-python/blob/main/docker-compose-3-nodes.yaml

Redpanda Console - By default 1 topic (**_schemas**) with replication as 3

- **Command to Check list of topics**
  docker exec -it redpanda-2 rpk topic list

- **Create topic with replication factor**
  docker exec -it redpanda-0 rpk topic create chat-room -r 3



Tutorial-6

Create Producer to Produce Message

Mainly Two Python Libraries available to work with kafka

- Confluent-kafka
- kafka-python

Confluent-kafka

https://pypi.org/project/confluent-kafka/

https://docs.confluent.io/platform/current/clients/confluent-kafka-python/html/index.html

https://github.com/confluentinc/confluent-kafka-python

kafka-python

https://pypi.org/project/kafka-python/

https://github.com/dpkp/kafka-python

Producer-Code -
https://github.com/Sumanshu-Nankana/kafka-python/blob/main/src/producer.py

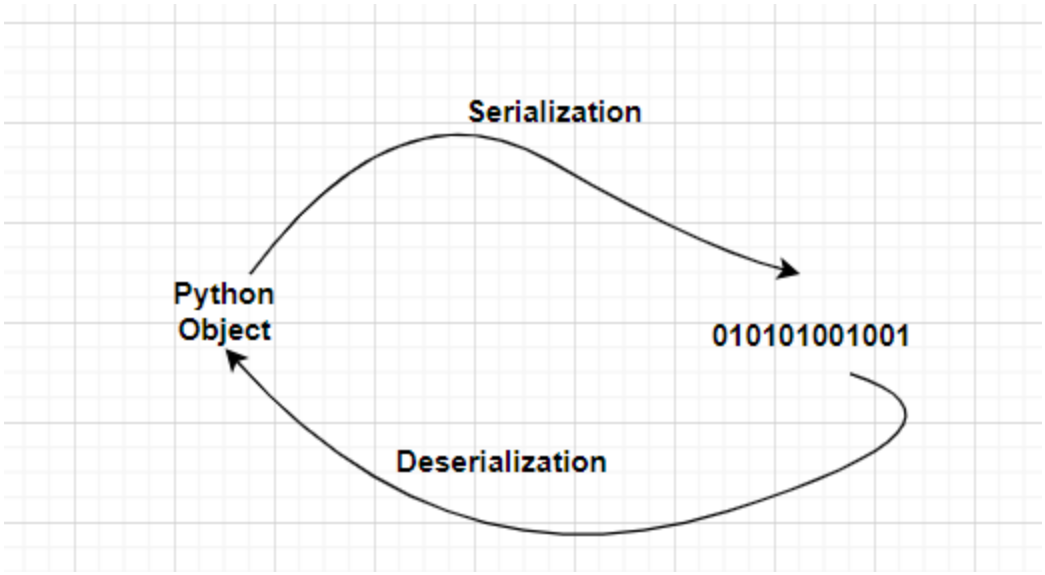
Tutorial-7

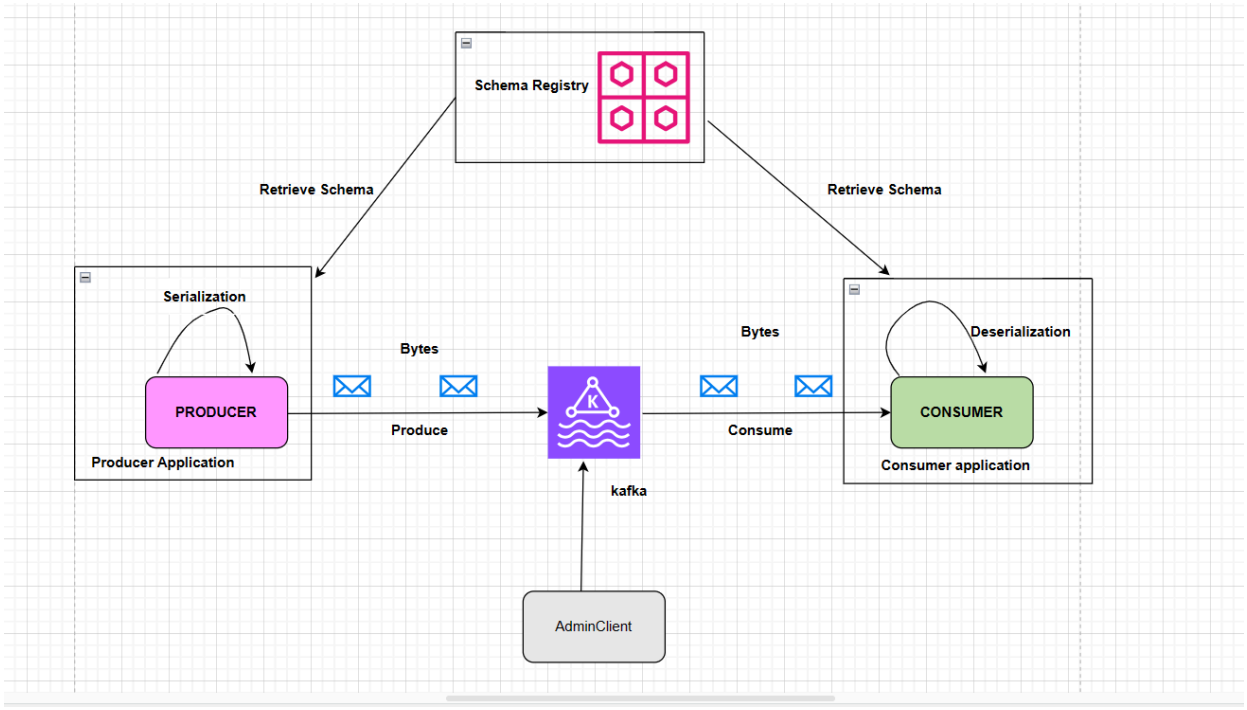
Create Consumer to Consume Messages

https://github.com/Sumanshu-Nankana/kafka-python/blob/main/src/consumer.py

Tutorial-8

Serialization and Deserialization





General-Steps (*if using the Schema based format AVRO or Protobuf***)**

- The producer application typically defines a schema for the messages it wants to send. This schema may be registered with a schema registry (If format is AVRO or Protobuf)

- When the producer application wants to send a message, it serializes the message data according to the defined schema. This serialization process typically involves converting

the message data into a byte array.

- The producer then sends the serialized message (byte array) along with metadata (e.g., topic name) to a Kafka broker.

- The Kafka broker receives the serialized message from the producer and appends it to the appropriate partition of the specified topic.

- The consumer application subscribes to one or more Kafka topics and periodically polls the Kafka broker for new messages.

- When a message is received, the consumer deserializes the message from its byte array form back into a structured format (e.g., object, record).

- If the message was serialized using a schema-based format, the consumer may retrieve the schema from the schema registry to assist in deserialization. The schema helps the consumer understand the structure of the serialized data and deserialize it correctly.

Tutorial-9

Produce & Consume Message in JSON format

- **Without defining schema**

1) Prepare Data in form of python dictionary
2) Convert Python Dictionary to JSON
3) Serialize and send it to kafka

On Producer Side : Python Dictionary —> JSON String → Byte Object
On Consumer Side : Byte Object —> JSON String —> Python Dictionary

- **With schema** (Schema can be defined Locally or in Schema Registry)

1) Prepare Data in form of dictionary
2) Convert Python Dictionary to JSON
3) Validate that JSON message whether it matches the schema
4) Then Serialize and send it to Kafka

Tutorial-10

Register the Schema in the Schema Registry

https://github.com/Sumanshu-Nankana/kafka-python/blob/main/src/schema_registry_client.py
https://github.com/redpanda-data/redpanda/issues/14462

Tutorial-11

Produce Message in AVRO format

https://github.com/Sumanshu-Nankana/kafka-python/blob/main/src/avro_producer.py

- Create Topic
- Register Schema
- Produce Message

https://stackoverflow.com/questions/49168794/what-is-key-schema-in-schema-registry

We will notice, after sending the message, there are two schema in schema registry
One with <topic-name> and other with <topic-name>**-value**

Tutorial-12

Consume Message in AVRO format

https://github.com/Sumanshu-Nankana/kafka-python/blob/main/src/avro_consumer.py

Tutorial-13

Send Key and Headers (example- Coorelation_ID) with every message

https://github.com/Sumanshu-Nankana/kafka-python/blob/main/src/avro_producer.py

Tutorial-14

On_delivery callback

on_delivery - eliminates the need of expliciting flushing

The flush() method forces the producer to send all pending messages in its buffer immediately. This can be useful when you want to ensure messages are sent before exiting the program or in situations where immediate delivery confirmation is critical.

on_delivery **Callback:** The on_delivery callback is triggered **after** each message is delivered or fails. This provides information about the delivery status in real-time, eliminating the need for explicit flushing.

enabling on_delivery ensures that the producer waits for acknowledgement for each message before proceeding, but it doesn't block the producer from sending subsequent messages in the meantime.

on_delivery:

- **Advantage**: You get precise feedback for each message. If something goes wrong, you know exactly which one caused the issue.
- **Disadvantage**: It can slow things down because you're waiting for confirmation after every message.

flush():

- **Advantage**: It's faster because it batches up messages and sends them together. You don't wait for individual confirmations.
- **Disadvantage**: If there's an error, you won't know which specific message caused it

- **If you need precision**: Use on_delivery to track each message.
- **If speed matters more**: Go with flush() to send batches efficiently.

Tutorial-15

Producer Configs (compression.type & message.max.bytes)

https://github.com/confluentinc/librdkafka/blob/master/CONFIGURATION.md

https://www.confluent.io/blog/apache-kafka-message-compression/

https://docs.confluent.io/platform/current/installation/configuration/topic-configs.html

- compression.type
- message.max.bytes

compression.type - is a configuration parameter that specifies the compression algorithm to use when producing messages

There are two places to set compression.type

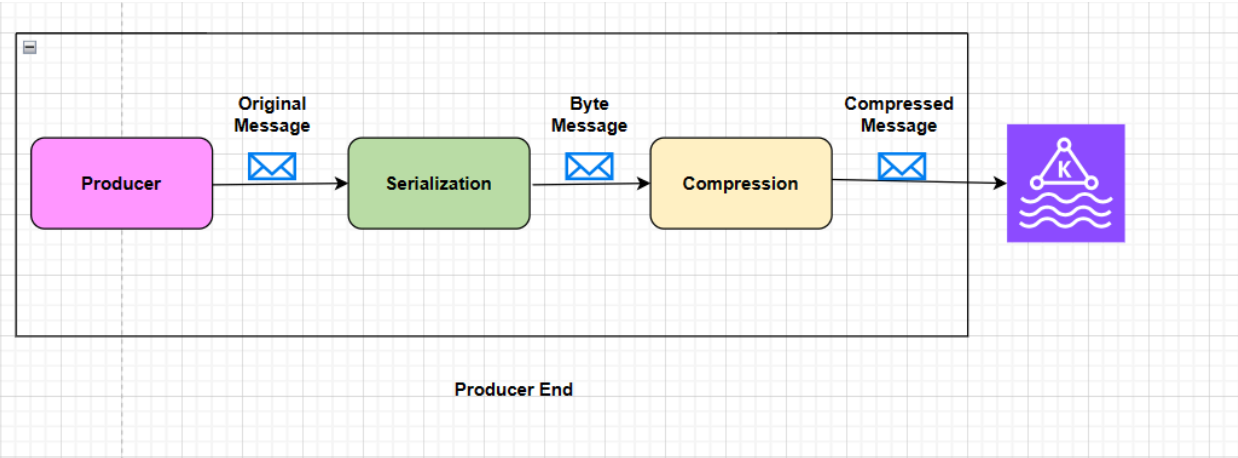
Producer Level - is used to compress the messages/data generated by producer
Compression.type = None (By default on Producer)

https://docs.confluent.io/platform/current/installation/configuration/producer-configs.html#compression-type
https://www.confluent.io/blog/apache-kafka-message-compression/#configuring-compression-type

```
Valid values are:

- none
- Gzip
- Snappy
```

- Lz4
- Zstd



Topic Level - Compression will be happen while storing the message on broker and Broker to consumer communication

compression.type = producer (By default on Topic Level)

A	B	C	D	E
Topic (compression.type)	**Producer (compression.type)**	**Compression type for producer to broker communication**	**Compression type for storage on broker, and broker to consumer communication**	Comments
producer	none	producer's compression.type No compression will be applied	producer's compression.type. No compression will be applied	Not recommended for production
producer	gzip, snappy, lz4, zstd	Compression applied whatever mentioned on producer	Compression applied whatever mentioned on producer	This is a common combination giving responsibility to the producer.
gzip, snappy, lz4, zstd	(something different than Topic, but not "none")	producer's compression.type	topic's compression.type	In this situation, the broker will have to recompress (using the topic's compression.type).
gzip, snappy, lz4, zstd	(same as Topic)	producer's compression.type	topic's compression.type	Not recommended because if producer's compression.type eventually changes, the broker will override it, which typically isn't desired. Suggest using topic compression.type = producer instead.
gzip, snappy, lz4, zstd	none	producer's compression.type	topic's compression.type	These combinations are uncommon,

message.max.bytes

https://github.com/confluentinc/librdkafka/blob/master/CONFIGURATION.md

This is the maximum size of uncompressed message in bytes sent by the Producer

Default is: 1MB

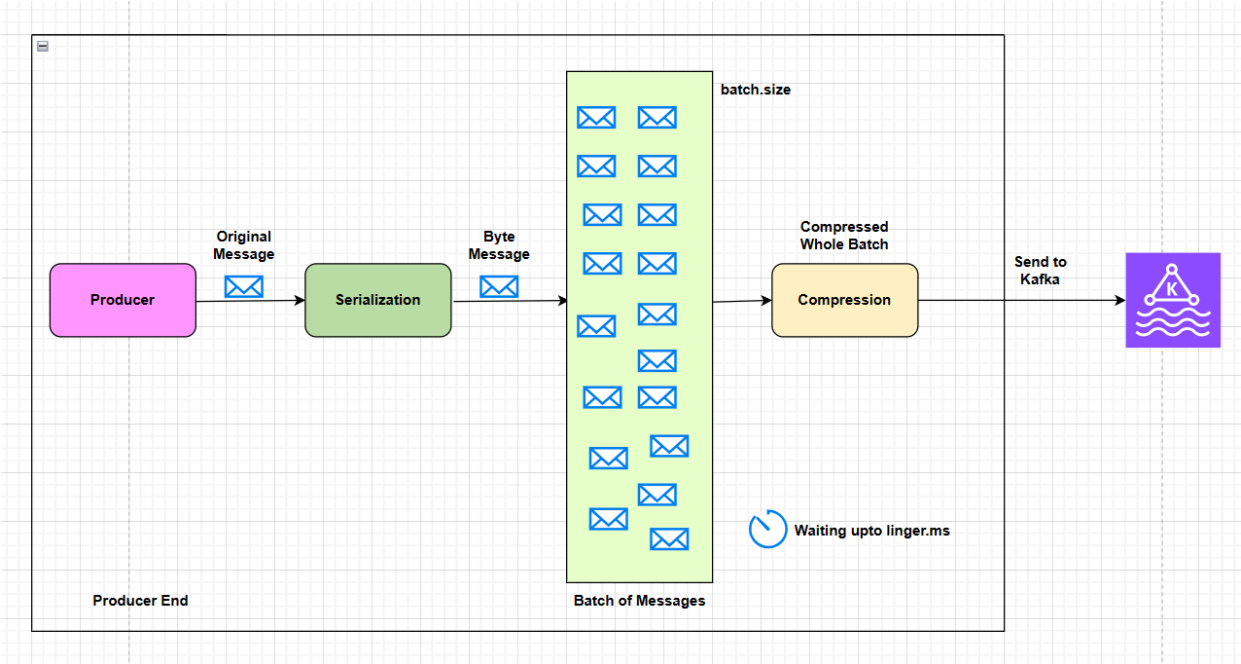
KafkaError{code=MSG_SIZE_TOO_LARGE,val=10,str="Unable to produce message: Broker: Message size too large"}

https://docs.confluent.io/platform/current/clients/confluent-kafka-python/html/index.html#kafkaerror

Tutorial-16

Producer Configs (batch.size & linger.ms)

- batch.size
- linger.ms



Batch.size

This parameter determines the maximum amount of data (in bytes) that the producer will attempt to batch together before sending it to Kafka.

When a producer sends messages to Kafka, it's often more efficient to send multiple messages together in a batch rather than individually.

batch.size - specifies the maximum size of such buffers

Default = 16KB (16384 bytes)

https://docs.confluent.io/platform/current/installation/configuration/producer-configs.html#batch-size

linger.ms

This parameter specifies the amount of time (in milliseconds) that the producer will wait before sending a batch of messages to Kafka.

Default = 0, which means we'll immediately send out a record.

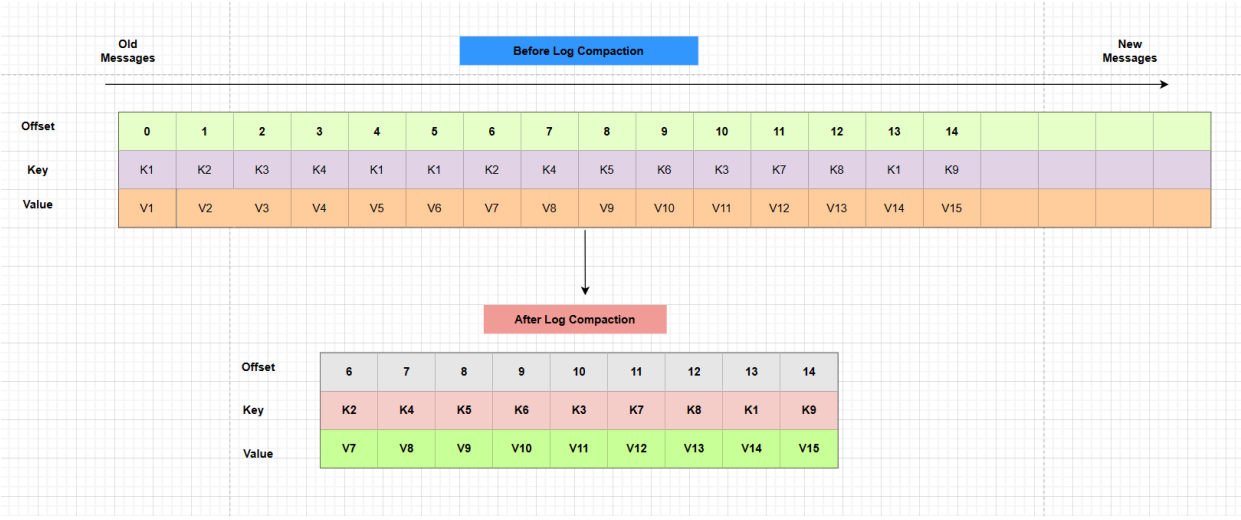
If the accumulated messages don't reach `batch.size` within `linger.ms`, the messages will be sent regardless of the batch size

https://docs.confluent.io/platform/current/installation/configuration/producer-configs.html#linger-ms

- When you call producer.produce(topic, key=None, value=msg), the message is added to the buffer.
- If the buffer size exceeds the batch.size, the entire batch is sent to Kafka.
- If the buffer size is below the batch.size, the producer waits for the specified linger.ms.
- After the linger.ms time elapses, the producer sends whatever messages are in the buffer (even if the batch size is not reached).

Tutorial-17

Log Compaction aka cleanup.policy = compact



Cleanup.policy - On topic Level

- **Delete** (Default) - Messages are deleted from the topic once the specified retention period (time and/or size allocations) is exceeded. This is the default mechanism and is analogous to disabling compaction.

- **Compact** - These settings enable log compaction. This triggers only cleanup of messages with multiple versions. A message that represents the only version for a given key is not deleted.

- **Compact, delete** - This combines both policies, deleting messages exceeding the retention period while compacting multiple versions of messages.

https://docs.confluent.io/kafka/design/log_compaction.html

Redpanda's compaction (just like Kafka's) does not guarantee perfect de-duplication of a topic. It represents a best effort mechanism to reduce storage needs but duplicates of a key may still exist within a topic. Compaction is not a complete topic operation, either, since it operates on subsets of each partition within the topic.

Periodically, Kafka runs a background process called compaction

Tutorial-18

Deleting Records in Kafka aka Kafka Tombstone

https://github.com/Sumanshu-Nankana/kafka-python/blob/main/src/avro_producer.py

Tombstone: is a special message used to indicate the deletion of a record associated with a specific key. It is essentially a flag that tells consumers this key no longer has valid data.

- A tombstone record is a regular kafka message with the same key as the record you want to delete, but with a null value for the payload

- Tombstones are only relevant for the compacted topics. In Kafka, compaction is an optimization technique that removes older versions of a record with the same key, keeping only the latest one. Tombstones help with this process by explicitly marking deletions.

Example:

Initial Insert: Key:123, Value: user123@gmail.com
Update: Key:123, Value: new_email123@gmail.com
Tombstone: Key: 123, Value: None

When compaction runs in background, it go through the topic partitions and identifies the messages with the same key, In this example, It see Original Record, Updated Record & Tombstone

Since, the Compaction Process recognizes the Tombstone, It understands that the user with ID: 123 is deleted. IT removes the older records (Initial and Updated one) and keeping only the Tombstone

A null payload is a payload with 0 bytes. A possible mistake here would be to send a "null" string encoded to UTF-8. It will not be interpreted as a tombstone.

Tutorial-19

auto_offset_reset (Consumer Configuration)

https://docs.confluent.io/platform/current/installation/configuration/consumer-configs.html#auto-offset-reset

The auto.offset.reset property specifies what offset the consumer should start reading from in the event.

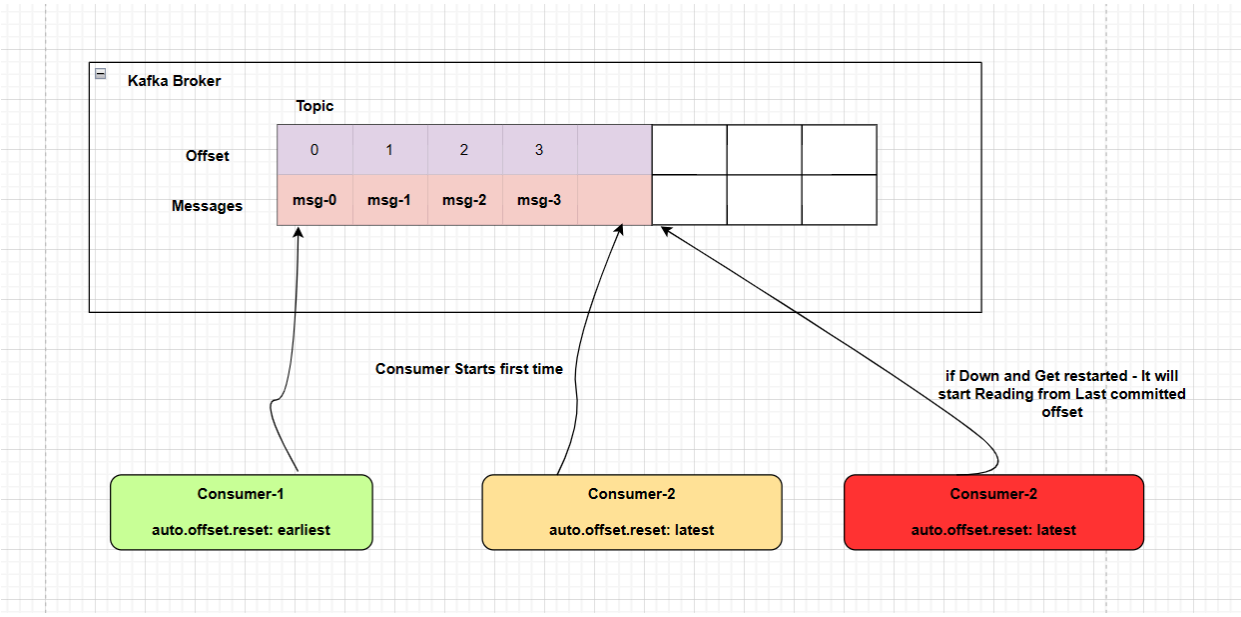
- **earliest:** automatically reset the offset to the earlier offset.
- **latest**: automatically reset the offset to the latest committed offset (**NOT** the latest offset)
- **none**: Throw exception to the consumer, if no previous offset is found for the consumer's group.
- **anything else**: Throw exceptions to the consumer.

Default : latest

When to Use -

Latest: use; if you're only interested in real-time data and not historical data.

Earliest: Use this if you want to consume all messages from the beginning of the partition, including historical data.



Scenarios:

- Fresh Consumer Started and Subscribe to the existing Topic (which already have lot of messages)

  If latest: - It will not read the existing messages. In other words, it will only consume messages published to the topic after the consumer has started.

  If earliest - It will read all the existing messages.This means it will read all messages from the beginning of the partition, including those published before the consumer started.

- Consumers stop (Because of some issue) and Get **restarted**. (auto.offset.reset - Does not come into picture, if earlier committed offset is available)

  If latest: Upon restart, It will start reading from the latest committed offset
  If earliest: Upon restart, It will start reading from the latest committed offset

Consumer Group

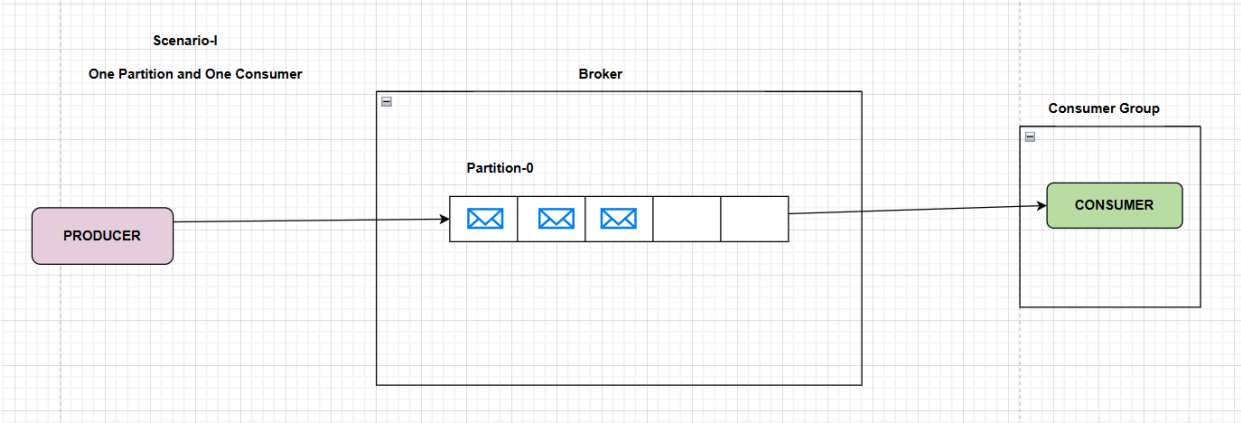
https://github.com/Sumanshu-Nankana/kafka-python/commit/ffda4ff2cc76ab3e9b78a6dc5862a94065506166

Consumer Group - Consumer group consists of one or more consumers sharing the same group-id.

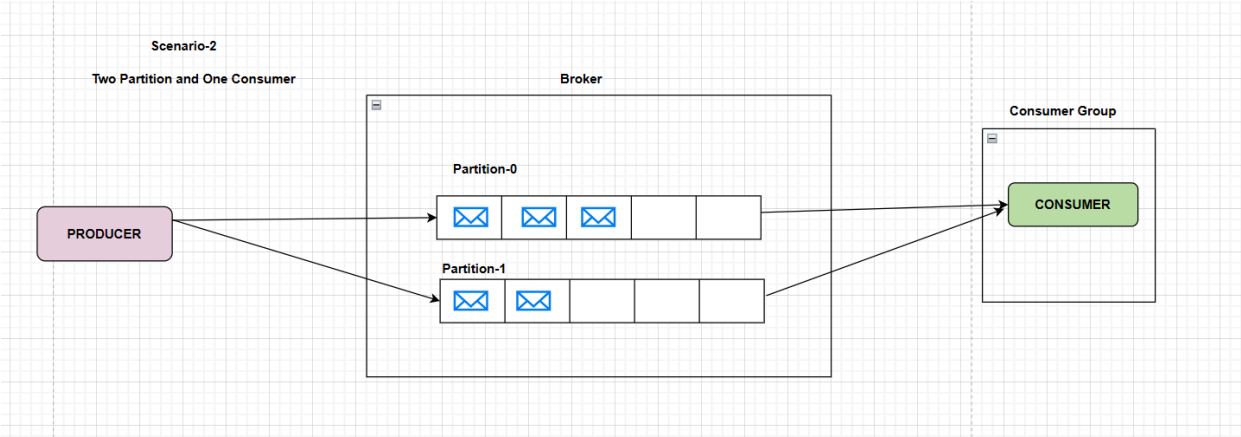
Topics - Each topic is divided into one or more partitions.

Scenarios:-

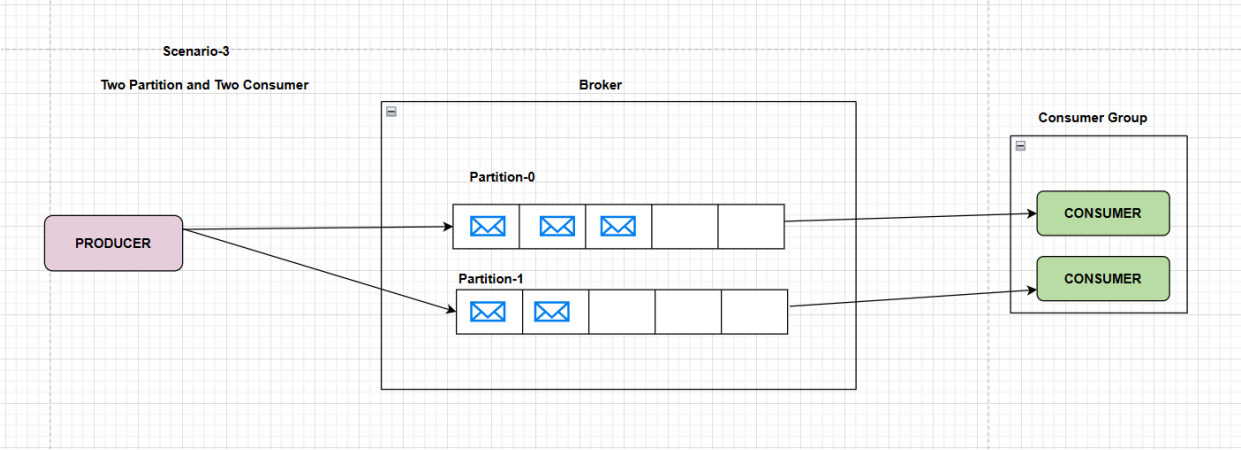
One Partition & One Consumer in Consumer Group: The consumer will read from one partition.



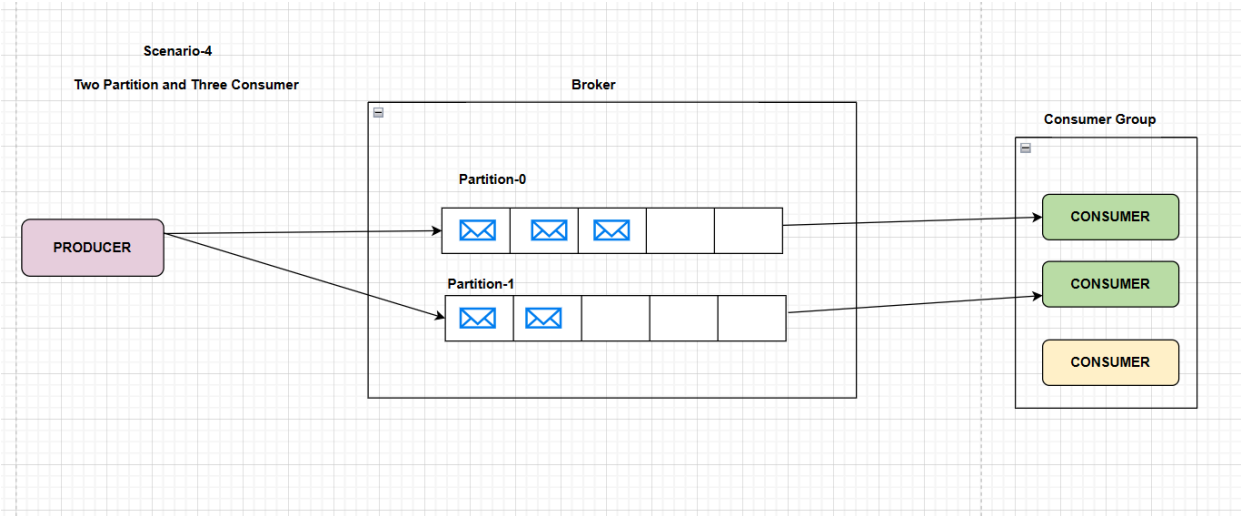
Two Partition & One Consumer in Consumer Group : If we have two partitions and only one consumer, the consumer reads from both



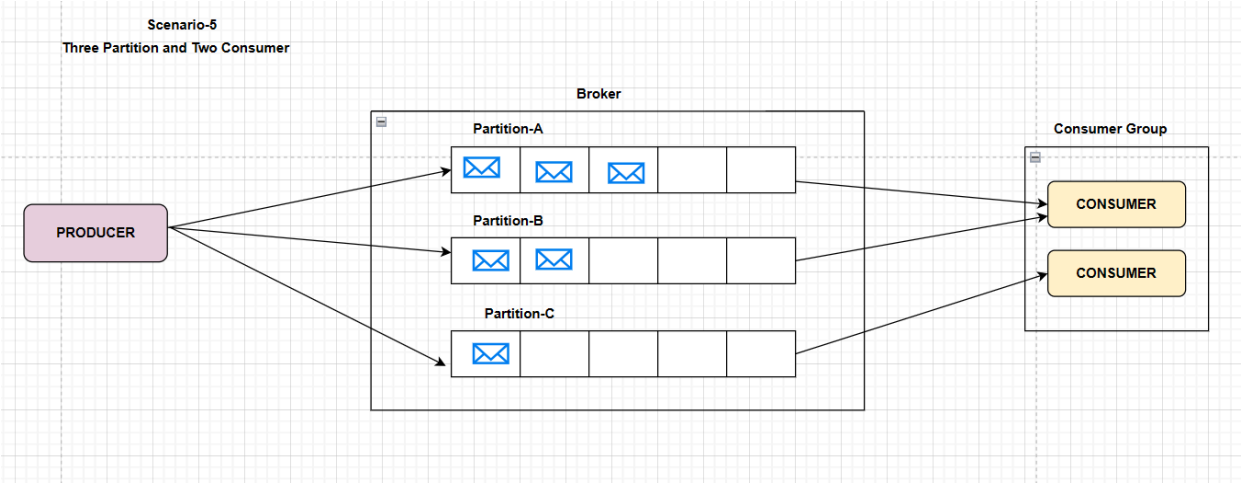
Two Partition & Two Consumer in Consumer Group: Each Consumer is assigned one partition, and this allows parallel consumption.



Two Partition & Three or More Consumers in Consumer Group: Two Consumers assigned two partitions (One Each) and The extra consumer will sit idle since there are no partitions to assign.



Three Partitions & Two Consumers in Consumer Group: Two partition will be assigned to first consumer and remaining partition to 2nd consumer



Tutorial-21

Schema Evolution & Schema Compatibility

https://docs.confluent.io/platform/current/schema-registry/fundamentals/schema-evolution.html

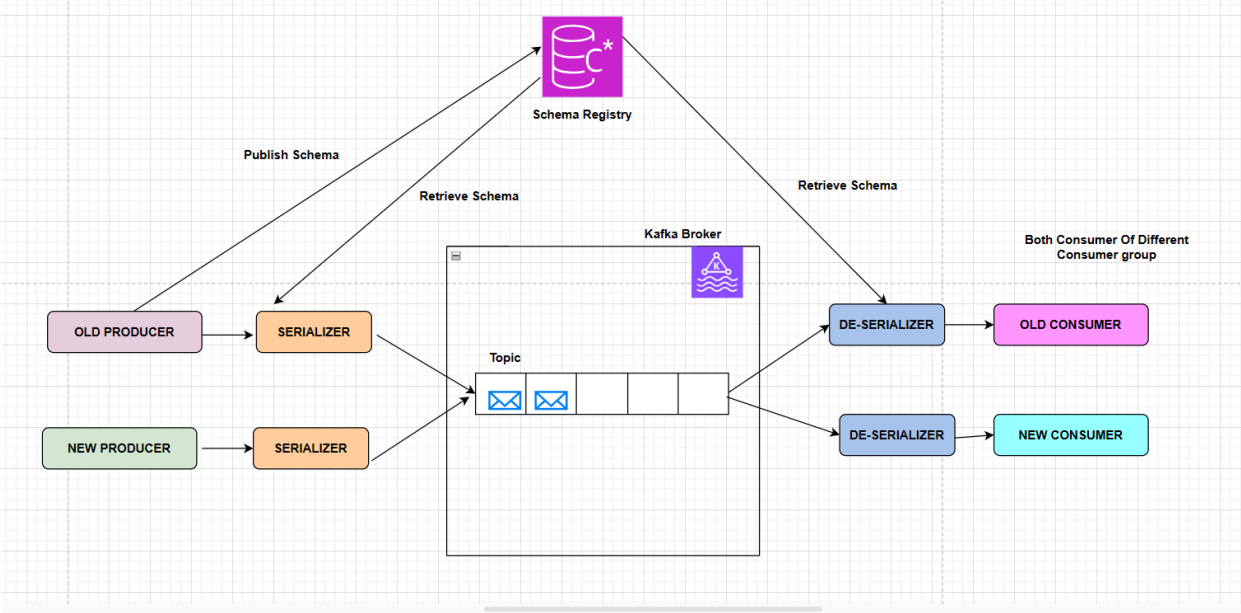
https://developer.confluent.io/courses/schema-registry/schema-compatibility/

Schema Evolution

Schema evolution involves making changes to the structure or definition of the data schema over time. This could include adding new fields, removing existing fields, or modifying the data types of existing fields, typically in response to evolving business requirements or application needs

Schema Compatibility

This ensures that changes made to the schema maintain compatibility with both old and new data, allowing data produced with different versions of the schema to be processed seamlessly. Compatibility ensures that consumers can handle data produced with different schema versions without errors or data loss.



Compatibility Types:

Backward-Data

- **Backward** 🙂 This is the default compatibility type for confluent schema registry.

  Which means while schema evolution or schema change or registering the new schema, we can

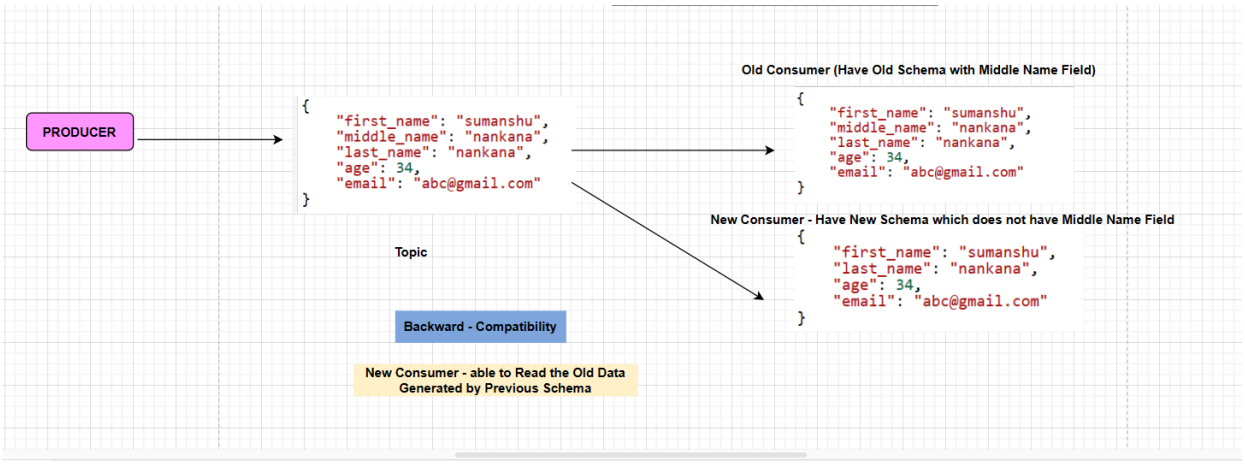
  - Delete the existing fields from schema
  - Add Optional Fields

- We CAN NOT change data types of field

  Comparison of newly registered schema is going to be happen with the Last Version i.e. Schema Compatibility will be checked with the last version.

BACKWARD compatibility means that consumers using the new schema can read data produced with the last schema.

Backward in simple terms —> Go Back (can new consumers which has latest schema – read old (previous) data having old schema)



Testing Steps (Backward Compatibility) :

- Start the Redpanda Kafka Cluster
- Start the Producer (called Old Producer) which is going to Register the Schema on Starting (Schema with Middle-Name)
- Start the Consumer (called Old consumer) which is going to Deserialize the Record with the Schema (Having Middle-Name)
- Publish/Produce some messages with middle-name
- Old Consumer will Consume the data and show Middle Name

  Till this point (It is normal Kafka Behaviour) - Irrespective of Schema Compatibility

- Evolve the Schema - Delete Middle Name Field (i.e. Backward Compatible) - which supports Deletion and Register into Schema Registry
- Start one More consumer (called New consumer) - should be part of different group-id

- And Check whether This New Consumer is able to read the existing Records (which was published with Middle-Name).

- **Backward_Transitive** 😄

  Everything remains the same as Backward, except Schema Compatibility will be checked with all the previous versions.

- **Forward** 👏

  Which means while schema evolution or schema change or registering the new schema, we can

  - Add New Fields
  - Delete Optional Fields

  Consumers using the previous schema can read data produced with the new version.

- FORWARD_TRANSITIVE

  Everything remains the same as Forward, except Schema Compatibility will be checked with all the previous versions.

- **Full** 👏

  Full compatibility means schemas are both backward **and** forward compatible. Schemas evolve in a fully compatible way: old data can be read with the new schema, and new data can also be read with the last schema

  - Add Optional Fields
  - Delete Optional Fields

- **Full Transitive** 👏

  Everything remains the same as Full, except Schema Compatibility will be checked with all the previous versions.

- **NONE** 😣 Compatibility Checking Disabled

  - All changes are accepted

Tutorial-22

Special topic called *_consumer_offsets & _schemas topic*

https://www.conduktor.io/kafka/kafka-consumer-groups-and-consumer-offsets/

https://docs.confluent.io/platform/current/schema-registry/installation/deployment.html

https://docs.redpanda.com/current/reference/rpk/rpk-topic/rpk-topic-list/

`__consumer_offsets`

https://docs.redpanda.com/current/develop/consume-data/consumer-offsets/

Kafka brokers use an internal topic named `__consumer_offsets` that keeps track of what messages a given consumer group last successfully processed.

The process of committing offsets is not done for every message consumed (because this would be inefficient), and instead is a periodic process

This also means that when a specific offset is committed, all previous messages that have a lower offset are also considered to be committed.

docker exec -it redpanda rpk topic list -i (Pass -i to see the internal topics)

```
NAME              PARTITIONS  REPLICAS
__consumer_offsets  3         1
_schemas          1         1
my-topic          2         1
```

To View the information, what information it store, you can consume the message

docker exec -it redpanda rpk topic consume __consumer_offsets --num 1


```
{
  "topic": "__consumer_offsets",
  "key": "\u0000\u0002\u0000\u0013consumer-group-id-1",
```

```
  "value":
"\u0000\u0003\u0000\u0008consumer\u0000\u0000\u0000\u0001\u0000\u0005rang
e\u0000,rdkafka-4c41c9ca-93aa-486e-bc53-e4ed7af54a9e\u0000\u0000\u0001\ufffd\
ufffd?\ufffd\ufffd\u0000\u
0000\u0000\u0001\u0000\u0003\u0000,rdkafka-4c41c9ca-93aa-486e-bc53-e4ed7af54
a9e\ufffd\ufffd\u0000\u0007rdkafka\u0000\n172.20.0.1\u0000\u0004\ufffd\ufffd\u0
000\u0000\ufffd\ufffd\u0000\u000
0\u0000\u001e\u0000\u0003\u0000\u0000\u0000\u0001\u0000\u0008my-topic\u00
00\u0000\u0000\u0000\u0000\u0000\u0000\u0000\ufffd\ufffd\ufffd\ufffd\u0000\u0
000\u0000\u0000\u0000
\u0000\u0000\u0000\u0000\u0000\u0001\u0000\u0008my-topic\u0000\u0000\u000
0\u0002\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0001\u0000\u0000\u000
0\u0000",
  "timestamp": 1716648908495,
  "partition": 1,
  "offset": 0
}
```


`_schemas :`

This is the internal topic used by the Schema Registry. Any information relevant to schema like (Topic Name, Version Number, metadata, compatibility) stored in this topic.

There is one default topic named _schemas available

docker exec -it redpanda rpk topic list

```
NAME      PARTITIONS  REPLICAS
_schemas  1           1
```

To view the information, from _schemas topic

docker exec -it redpanda rpk topic consume _schemas --num 1


```
{
  "topic": "_schemas",
```

```
  "key":
"{\"keytype\":\"CONFIG\",\"subject\":\"my-topic\",\"magic\":0,\"
seq\":0,\"node\":0}",
  "value": "{\"compatibilityLevel\":\"BACKWARD\"}",
  "timestamp": 1716648622525,
  "partition": 0,
  "offset": 0
}
```


Tutorial-23

Kcat (Kafka cat commands) | kcat (formerly kafkacat)

KafkaCat is a powerful command-line utility designed to help you interact with Kafka topics, produce and consume messages, and navigate Kafka clusters seamlessly.

https://github.com/edenhill/kcat

Kafkacat or kcat (Both are same)

kafkacat is the original name, while **kcat** is the renamed, shorter version.

https://hub.docker.com/r/edenhill/kcat

Docs: https://docs.confluent.io/platform/current/tools/kafkacat-usage.html

Commads

- **List Metadata**

  docker run -it --rm --network host edenhill/kcat:1.7.1 -L -b localhost:19092
  OR
  (Open ubuntu and install kafkacat)
  - sudo apt update
  - Sudo apt install kafkacat

kafkacat -b localhost:19092 -L

- **Produce Messages**

  -P (Producer Mode), default is Consumer Mode

  docker run -it --rm --network host edenhill/kcat:1.7.1 -t topic -P -b localhost:19092

  Then enter your message, and then press **CTRL + D**

  OR

  kafkacat -b localhost:19092 -t my-topic -P

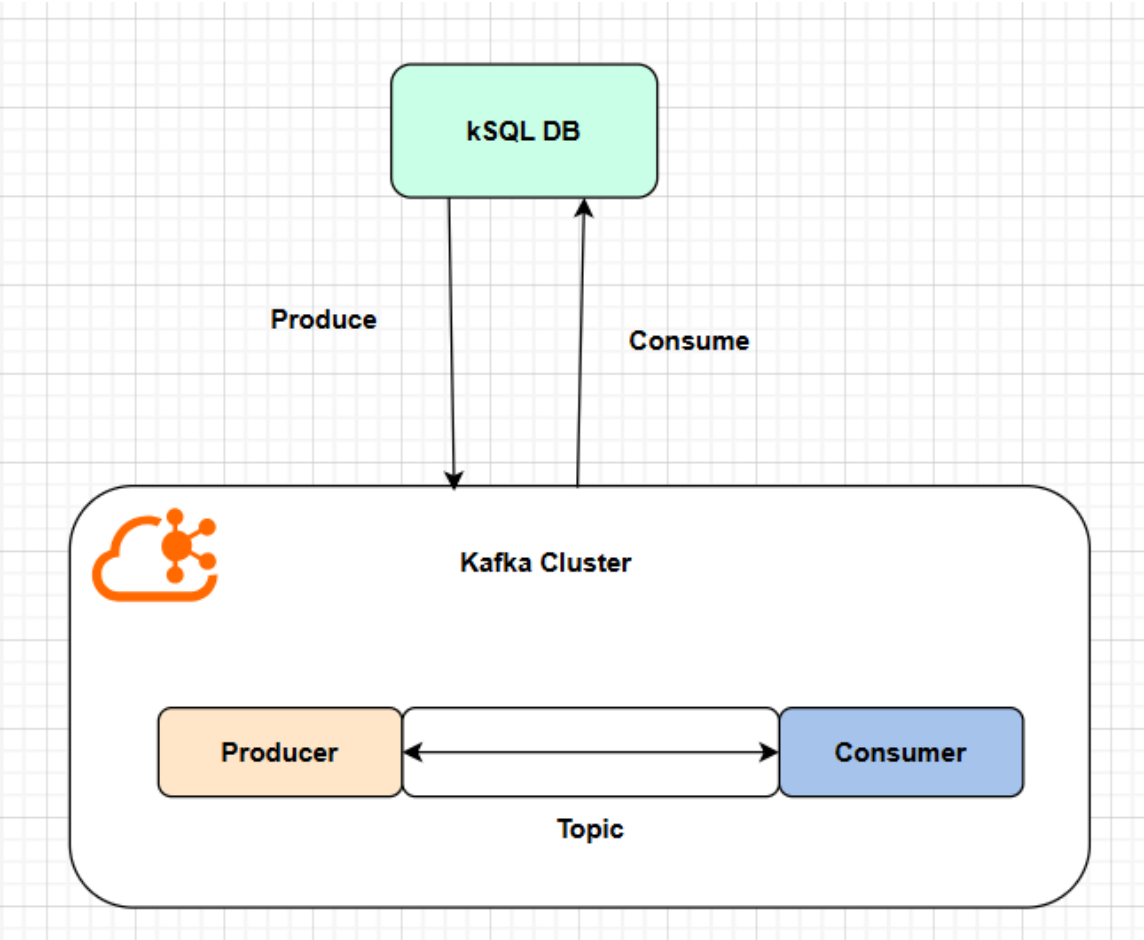
- **Consume Message**
  -C (For Consume)
  docker run -it --rm --network host edenhill/kcat:1.7.1 -t topic -P -b localhost:19092
  kafkacat -b localhost:19092 -t my-topic -C

Tutorial-24

K-SQL DB

KSQL DB is a tool that lets you use SQL to process real-time data from kafka topics without writing code

- Works directly with kafka topics
- Use simple SQL commands to filter, join the data
- We don’t need Java/Python consumers code for basic data processing
- KSQLDB runs separately from the kafka brokers
- KSQLDB has no build-in web UI




```
# https://hub.docker.com/r/confluentinc/ksqldb-server
ksqldb-server:
 image: confluentinc/ksqldb-server
 container_name: ksqldb-server
 depends_on:
   - redpanda
 ports:
   - 8088:8088
 environment:
   KSQL_BOOTSTRAP_SERVERS: "redpanda:9092"  # tells ksql-db which kafka broker to
connect
   KSQL_KSQL_SCHEMA_REGISTRY_URL: "http://redpanda:8081"  # Tells ksqlDB where the
Schema Registry is  (useful in case if AVRO/JSON/Protobuf)
   KSQL_LISTENERS: "http://0.0.0.0:8088" # Specify the address where the ksqlDB
should listen
   KSQL_CACHE_MAX_BYTES_BUFFERING: 0 # executes queries with no buffering
```

KSQLDB - REST-API Endpoints:
https://docs.confluent.io/platform/current/ksqldb/developer-guide/ksqldb-rest-api/overview.html

ksqlDB-CLI

- Ksqldb-cli is a CLI tool for working with ksqlDB which lets you execute SQL queries interactively
- It is a lightweight tool that connected with ksqldb-server
- We can launch it via docker OR installed locally

https://docs.confluent.io/platform/current/ksqldb/quickstart.html#ksqldb-quick-start-create-docker-compose-file

```
# https://hub.docker.com/r/confluentinc/ksqldb-cli
ksqldb-cli:
 image: confluentinc/ksqldb-cli
 container_name: ksqldb-cli
 depends_on:
   - ksqldb-server
 entrypoint: /bin/sh
 tty: true # without this, container will started and exited (This is to simulate a terminal)
```

```
docker exec -it ksqldb-cli /bin/sh
ksql http://ksqldb-server:8088
Exit;

docker exec -it ksqldb-cli ksql http://ksqldb-server:8088

SHOW TABLES;
SHOW STREAMS;
```

CREATE STREAM

https://docs.confluent.io/platform/current/ksqldb/developer-guide/ksqldb-reference/create-stream.html

```
ksql> CREATE STREAM person_stream (
>  first_name VARCHAR,
>  middle_name VARCHAR,
>  last_name VARCHAR,
>  age INT
>) WITH (
>  KAFKA_TOPIC = 'my-topic',
>  VALUE_FORMAT = 'JSON'
>)

ksql> show streams;

ksql> select * from person_stream;

# if we want to see the result in REAL-TIME
# EMIT CHANGES tells ksqlDB that your query is continuous and you want it
# to stream results in real-time as new data arrives.
ksql> select * from person_stream emit changes;
```

Commands:

- SHOW TOPICS;
- SHOW STREAMS;
- SHOW TABLES;

  https://docs.confluent.io/platform/current/ksqldb/developer-guide/ksqldb-reference/describe.html
-
- DESCRIBE <STREAM_NAME>;
- DESCRIBE <STREAM_NAME> EXTENDED;

  https://docs.confluent.io/platform/current/ksqldb/developer-guide/ksqldb-reference/drop-stream.html

- DROP STREAM <STREAM_NAME>; -

Create the Stream (With columns loaded from Schema registry)

```
# From JSON Schema

ksql> CREATE STREAM users WITH (
```

```
>KAFKA_TOPIC = 'my-topic',
>VALUE_FORMAT = 'JSON_SR');

# From AVRO SCHEMA

ksql> CREATE STREAM users WITH (
>KAFKA_TOPIC = 'my-topic',
>VALUE_FORMAT = AVRO);
```

KSQLDB - Query Structure Data

https://docs.confluent.io/platform/current/ksqldb/how-to-guides/query-structured-data.html

In KSQL → Index Starts from 1

```
ksql> select userid, createdat, actiontype, basicinfo -> full_name,
basicinfo -> age, basicinfo -> email, interests[1] -> category,
interests[1] -> description, preferences -> newslettersubscribed,
preferences -> preferredlanguage, preferences -> darkmode from users;

+------------+------------+------------+------------+------------+---------
---+------------+------------+------------+------------+------------+
|USERID      |CREATEDAT   |ACTIONTYPE  |FULL_NAME   |AGE         |EMAIL
|CATEGORY    |DESCRIPTION |NEWSLETTERSU|PREFERREDLAN|DARKMODE    |
|            |            |            |            |            |
|            |            |BSCRIBED    |GUAGE       |            |
+------------+------------+------------+------------+------------+---------
---+------------+------------+------------+------------+------------+
|1           |2025-07-20T1|CREATE      |Sumanshu Nan|34
|sumanshu@gma|SPORTS      |Boxing      |true        |en          |false
|
|            |8:59:28.842 |            |kana        |            |il.com
|            |            |            |            |            |
```

EXPLODE function & CREATE STREAM AS SELECT

Creating a New Materialization Stream View

https://docs.confluent.io/platform/current/ksqldb/developer-guide/ksqldb-reference/table-functions.html#explode

https://docs.confluent.io/platform/current/ksqldb/developer-guide/ksqldb-reference/create-stream-as-select.html

```
ksql> CREATE STREAM users_exploded AS
>SELECT
>  userid,
>  createdat,
>  actiontype,
>  basicinfo->full_name AS full_name,
>  basicinfo->age AS age,
>  basicinfo->email AS email,
>  EXPLODE(interests) AS interest,
>  preferences->newslettersubscribed AS newslettersubscribed,
>  preferences->preferredlanguage AS preferredlanguage,
>  preferences->darkmode AS darkmode
>FROM users;

ksql> SELECT
>  userid,
>  full_name,
>  interest->category AS category,
>  interest->description AS description
>FROM users_exploded
>EMIT CHANGES;

+-----------------------------------+-----------------------------------
-+-----------------------------------+-----------------------------------
--+
|USERID                             |FULL_NAME
|CATEGORY                           |DESCRIPTION
|
+-----------------------------------+-----------------------------------
-+-----------------------------------+-----------------------------------
--+
|2                                  |rajesh gite
|SPORTS                             |Cricket
|
```

```
|2                                        |rajesh gite
|SPORTS                                   |Gym
|
|2                                        |rajesh gite
|SPORTS                                   |Hockey
|
```

Note: At backend , it will create a topic user_exploded into and write the data into it

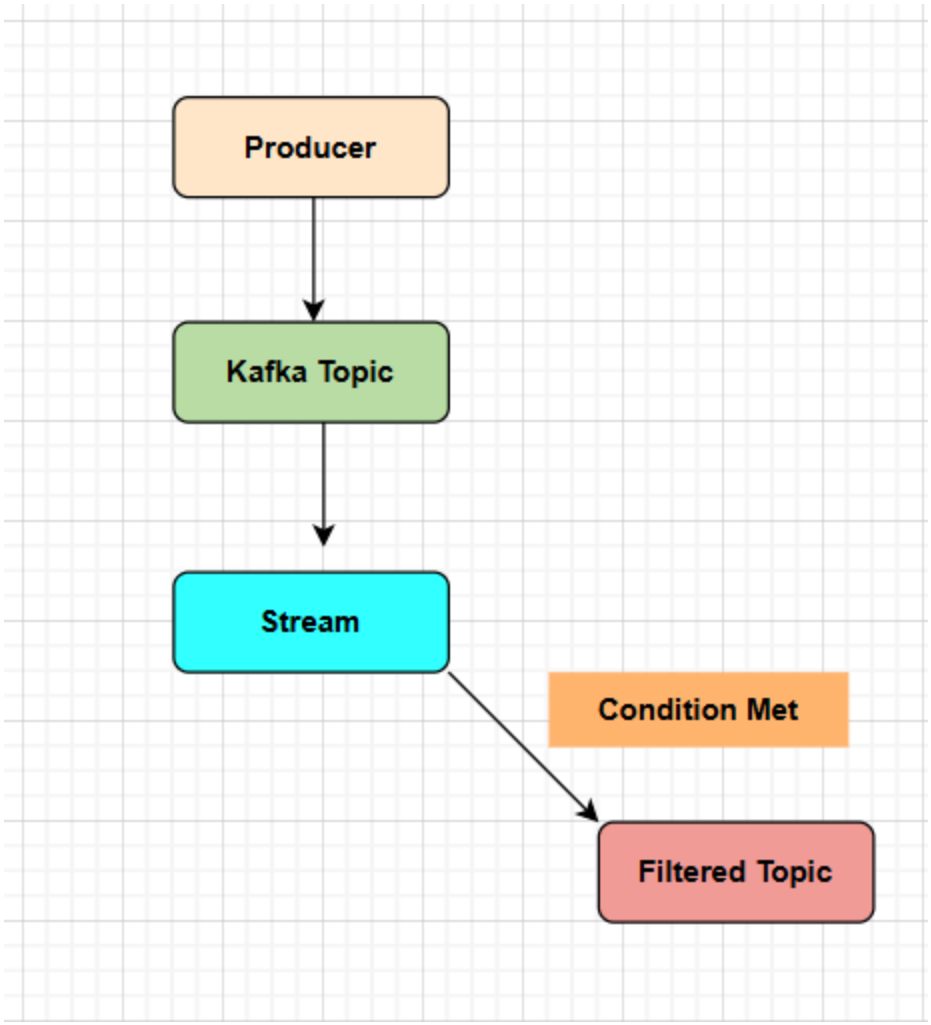
ksqlDB:

- Starts a persistent query
- This will creates a new Kafka topic
- Writes all transformed records into that topic

This topic becomes the backing store for the new stream.

See CREATE STREAM AS SELECT in action:

- Detect Unusual Credit Card Activity
- Notify Passengers of Flight Updates
- Detect and analyze SSH attacks



CREATE TABLE and CREATE TABLE AS SELECT

https://docs.confluent.io/platform/current/ksqldb/developer-guide/ksqldb-reference/create-table.html
https://docs.confluent.io/platform/current/ksqldb/developer-guide/ksqldb-reference/create-table-as-select.html

```
ksql> create table users (id varchar primary key) with (
>kafka_topic = 'test-topic-avro',
>value_format = 'avro');

 Message
---------------
```

```
 Table created
---------------
ksql> show tables;

 Table Name | Kafka Topic     | Key Format | Value Format | Windowed
-------------------------------------------------------------------
 USERS      | test-topic-avro | KAFKA      | AVRO         | false
-------------------------------------------------------------------
```

SELECT * FROM USERS → will NOT WORK

```
ksql> select * from users;
The `USERS` table isn't queryable. To derive a queryable table, you can do
'CREATE TABLE QUERYABLE_USERS AS SELECT * FROM USERS'. See
https://cnfl.io/queries for more info.
Add EMIT CHANGES if you intended to issue a push query.
```

SELECT * FROM USERS EMIT CHANGES; → This will WORK

But this will only show the updates if there is a change.

```
ksql> create table user_query as select * from users;

 Message
----------------------------------------
 Created query with ID CTAS_USER_QUERY_50
----------------------------------------
```

Table will only store latest state per key

Three Type of Queries in KSQLDB

https://docs.confluent.io/platform/current/ksqldb/concepts/queries.html

- Perisistent
- Push Queries
- Pull Queries

Persistent Queries

```
CREATE STREAM new_stream AS SELECT * FROM old_stream WHERE age > 40;

CREATE TABLE active_users AS SELECT user_id, MAX(last_login) FROM logins
GROUP BY user_id;
```

- Write the result into a new Kafka topic. (i.e. Persisted State)
- SHOW QUERIES;
- These are Long Running
- Can be stopped with `TERMINATE <query_id>;`

Push Queries:

```
SELECT * FROM some_stream EMIT CHANGES;

SELECT * FROM active_users EMIT CHANGES;
```

- Client gets real time updates
- The result of a push query isn't persisted to a backing Kafka topic.

Pull Queries:

- Get the current value (point-in-time) from a Table.
- Only Work on Tables (not Streams)
- Fast point lookups — good for APIs or read services.
- Returns only the current/latest data per key.
- A pull query is a form of query issued by a client that retrieves a result as of "now", like a query against a traditional RDBMS
- Pull queries enable you to fetch the current state of a materialized view

```
SELECT * FROM active_users WHERE user_id = 'U123';
```

JOINS

https://docs.confluent.io/platform/current/ksqldb/developer-guide/joins/overview.html

- STREAM - STREAM JOIN
- STREAM - TABLE JOIN
- TABLE - TABLE JOIN