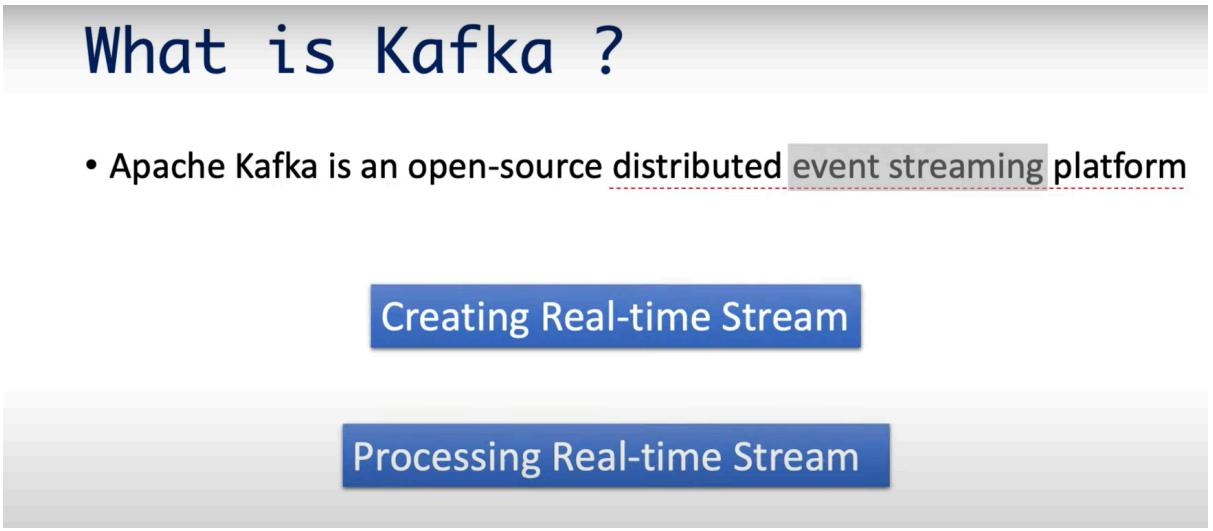
Kafka

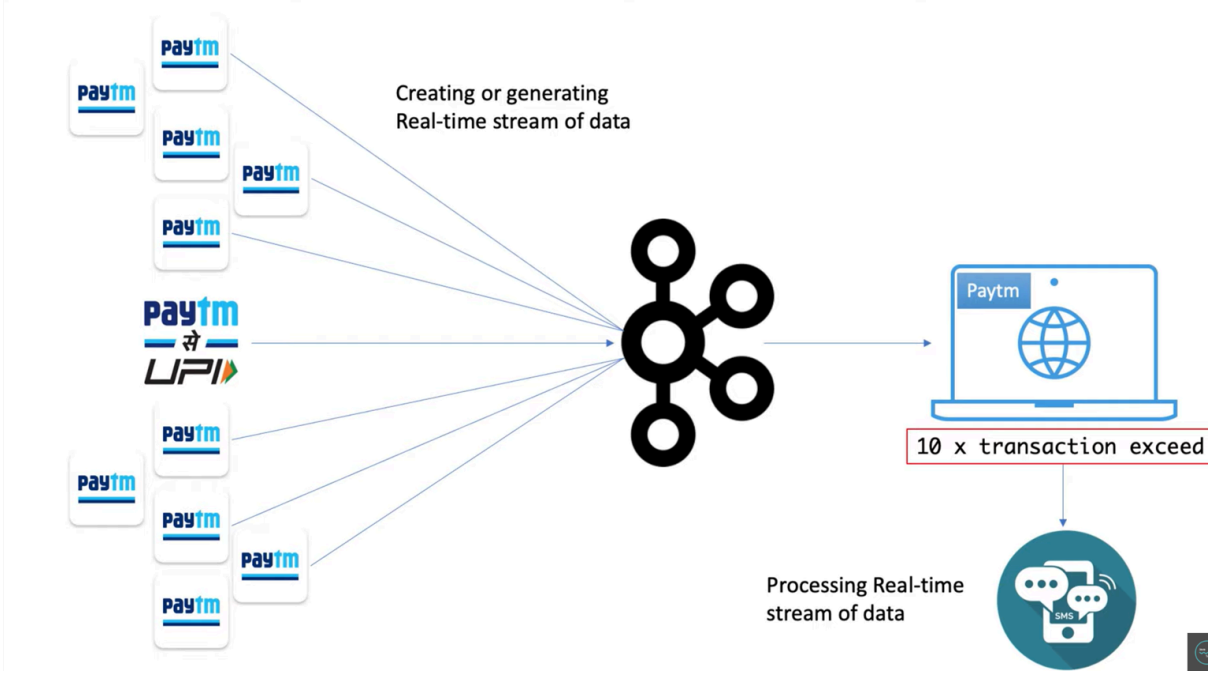
Apache Kafka is an open-source distributed event streaming platform designed for high-throughput, low-latency handling of real-time data feeds.



Event streaming means

1. Creating real time system
2. Processing real time system

Suppose i am using paytm ,when i am doing any transaction event go to the kafka server ,but there are multiple paytm users so kafka server receives millions of request at the same time so sending streams of continuous data from paytm to kafka is called creating or generating real time data of streams.



Paytm restricted 10 transactions per user per day. If the user exceeds the limit ,paytm sends notification to the clients. This continuously sending messages or events to the kafka server ,reading and processing them is called real time event streaming.

Where does kafka comes from

Apache Kafka originated at LinkedIn in 2010.In early 2011, LinkedIn open-sourced Kafka, and it was later donated to the Apache Software Foundation. Kafka graduated from the Apache Incubator on October 23, 2012, becoming a top-level project.

Why we use kafka

🔑 Key Benefits of Using Apache Kafka

1. High Throughput and Low Latency: Kafka can process millions of messages per second with minimal delay, making it suitable for real-time applications where speed is crucial. Toxigon

2. Scalability: Its distributed architecture allows Kafka to scale horizontally by adding more brokers, accommodating growing data volumes without compromising performance.

3. Fault Tolerance: Kafka replicates data across multiple brokers, ensuring data durability and system resilience even in the event of hardware failures. Codecademy

4. Durability: Messages are stored on disk and replicated within the cluster, providing reliable message storage and retrieval.

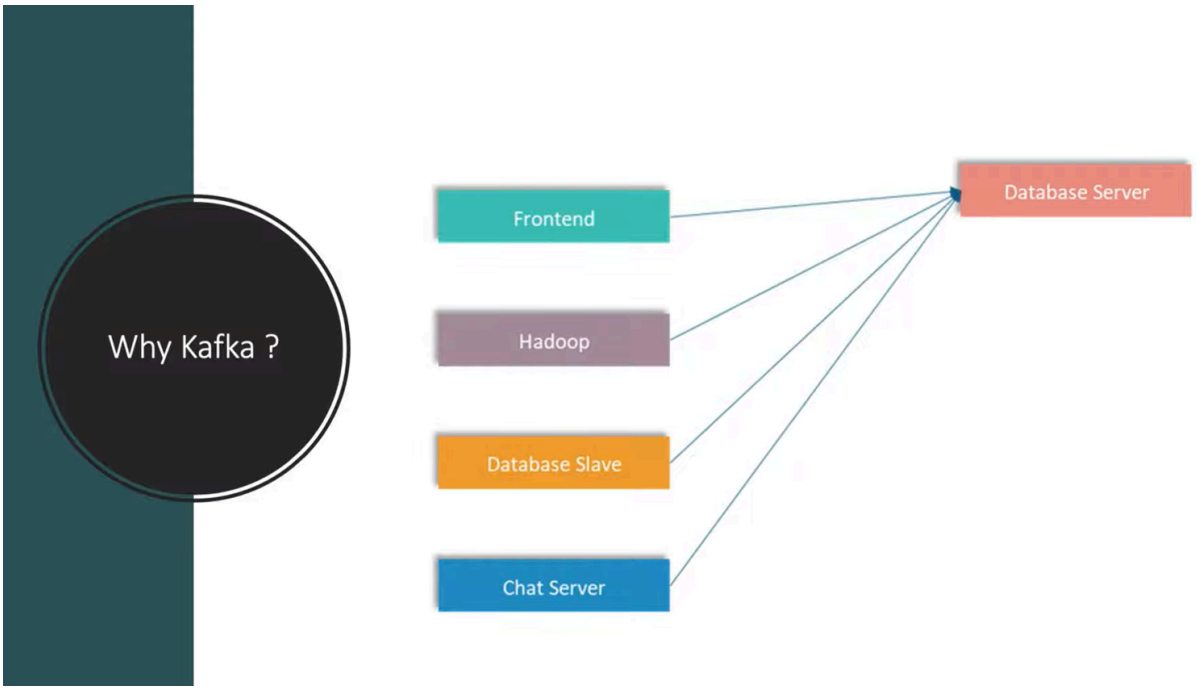
5. Flexibility: Kafka supports various data processing models, including real-time stream processing and batch processing, catering to diverse application needs.

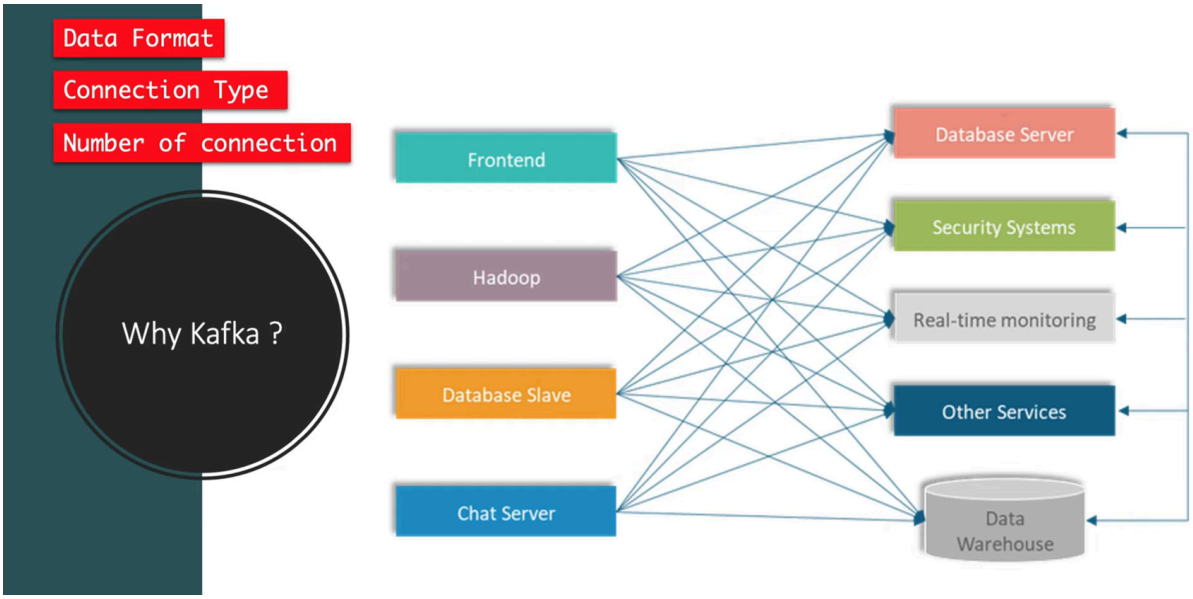
📌 Common Use Cases

- Real-Time Analytics: Processing streaming data for immediate insights, such as monitoring user activity or financial transactions.
- Log Aggregation: Collecting and centralizing logs from different services for analysis and troubleshooting.
- Event Sourcing: Capturing changes to application state as a sequence of events, useful in microservices architectures.
- Data Integration: Serving as a central hub for data exchange between different systems and applications.
- Messaging: Decoupling producers and consumers, allowing for asynchronous communication between services.

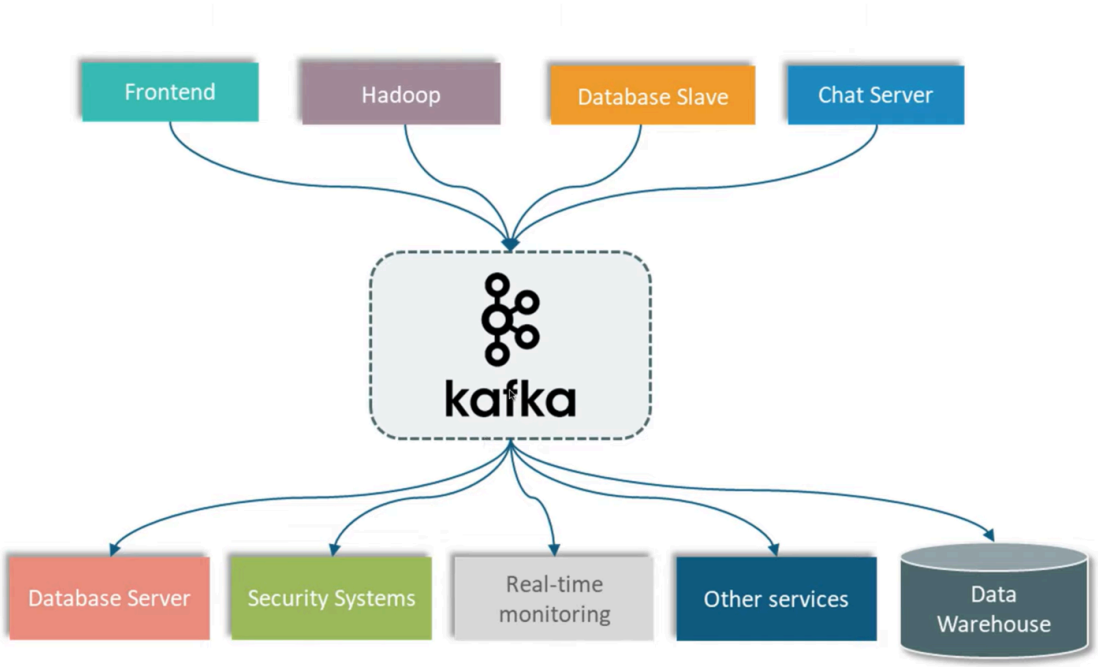
There are 2 apps ,if one of the apps is not working while sending data from app1 to app2 ,we will get data loss. If we add a messaging system between 2 apps then we can get data from the kafka message broker.

We have 4 apps to get data in DB. When apps grow then we can face some challenges like data formatting , maintaining and creating connections.





We can add a kafka server between services . We are managing a centralized system which can manage the data transform and manage the connections so we can reduce the connections also because all services are connected to kafka servers and getting data from kafka server.



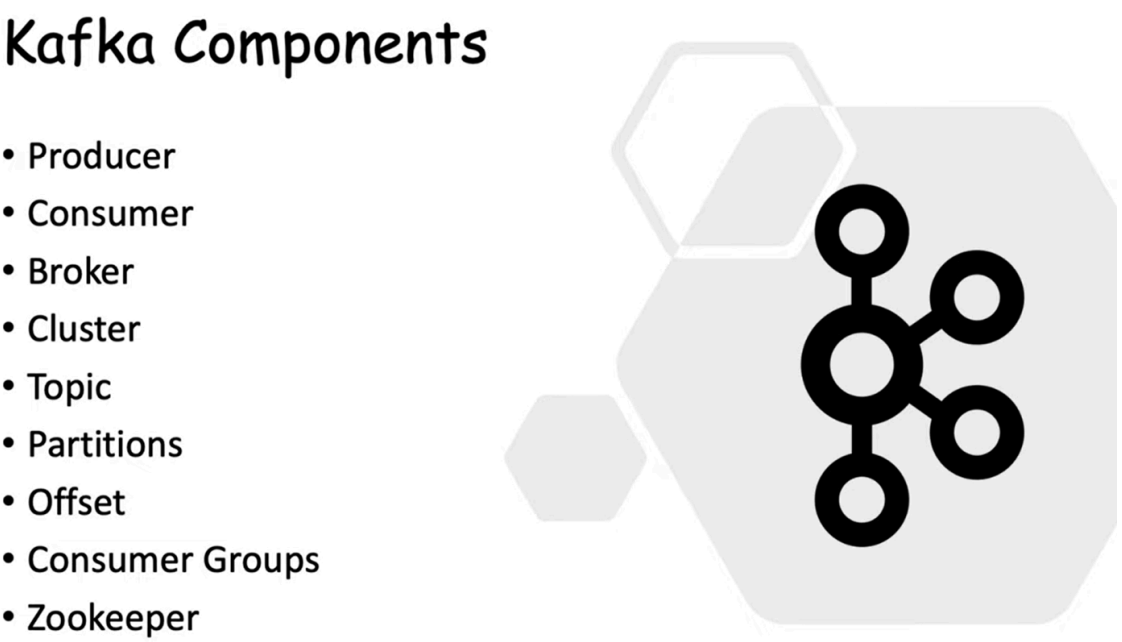
Kafka architecture

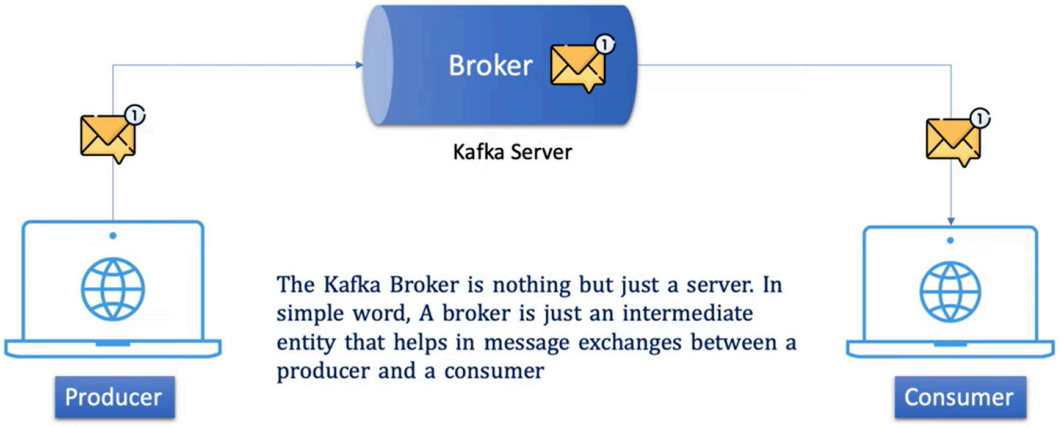
🔧 Core Components of Kafka Architecture

Producer: Applications that publish (write) data to Kafka topics. Producers send records to specific topics, and Kafka appends these records to the appropriate partitions within the topic.

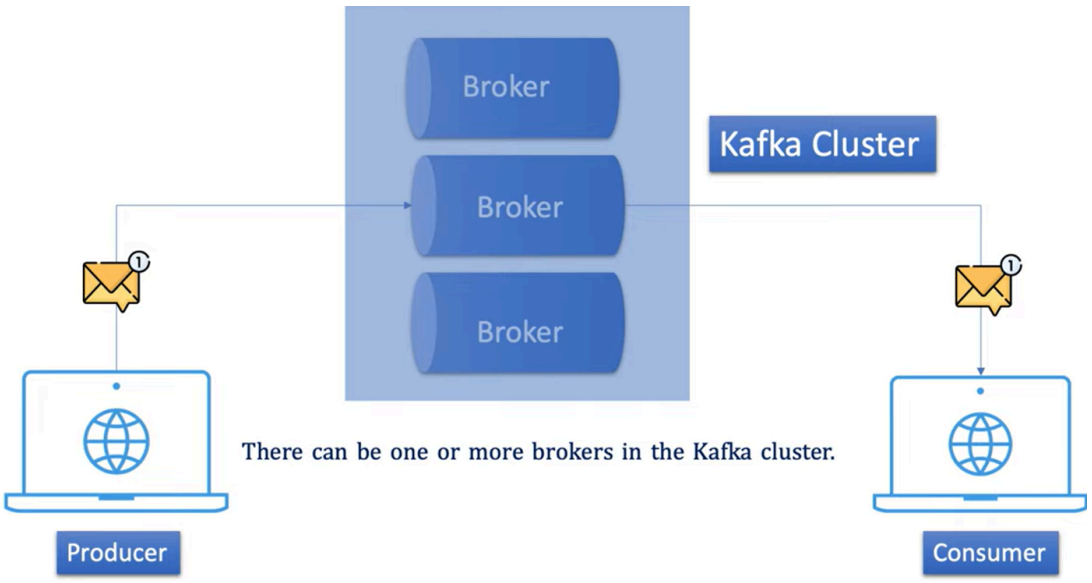
Consumer: Applications that subscribe to (read) data from Kafka topics. Consumers read records from topics by subscribing to one or more partitions and processing the incoming data.

Broker: A Kafka server that stores data and serves client requests. Each broker manages a set of partitions and handles read and write operations for them.[Instaclustr]

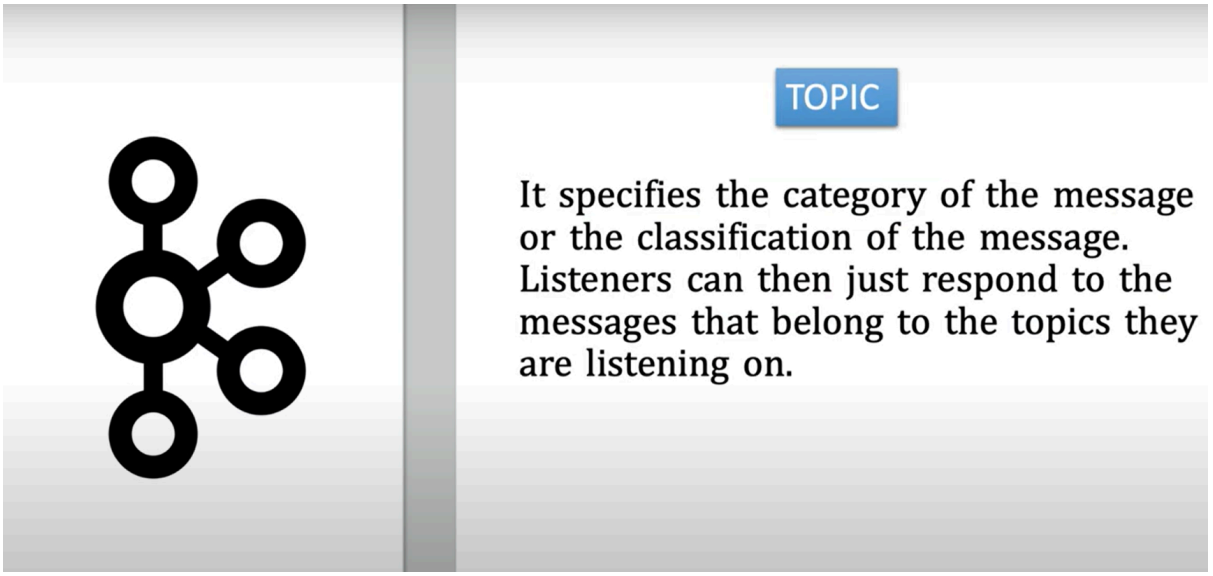




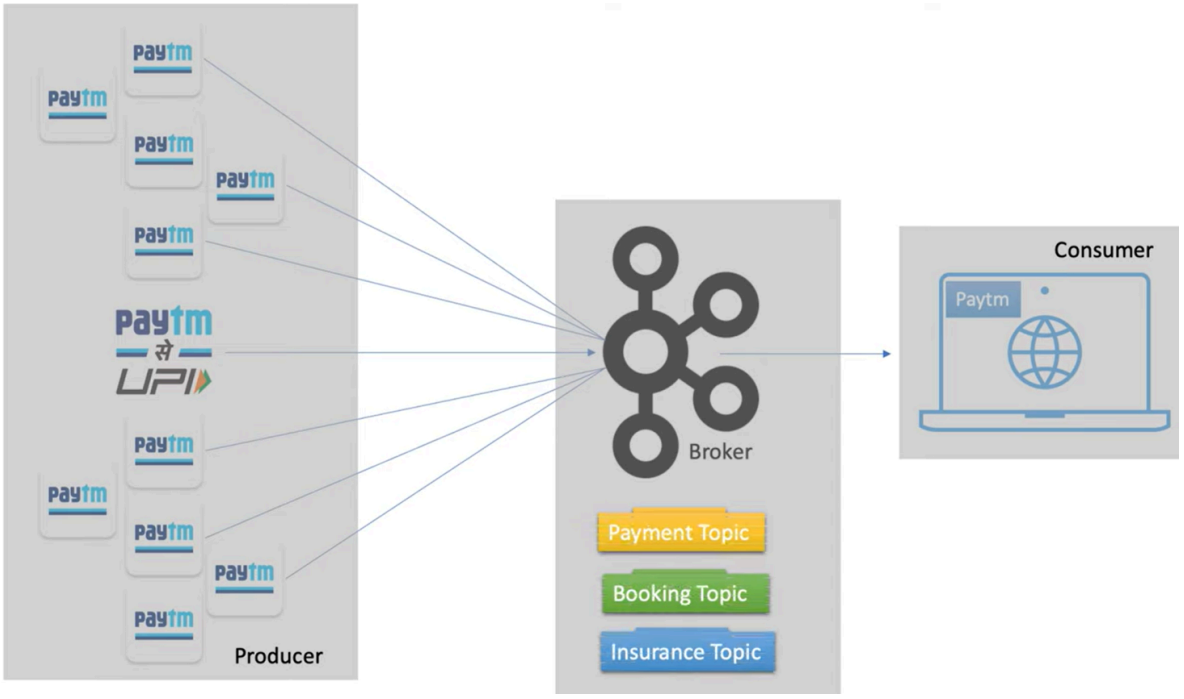
Cluster: A group of Kafka brokers working together. A Kafka cluster handles the storage and processing of data, providing fault tolerance and scalability. If Producer sends a huge volume of data then a single kafka broker may not handle the load ,we can group multiple brokers in a cluster.



Topic: Brokers receive different types of messages . Consumers can segregate the message by topic .It specifies the category of messages or classifications of messages then listeners can respond to the messages that belong to what topic they are listening on.

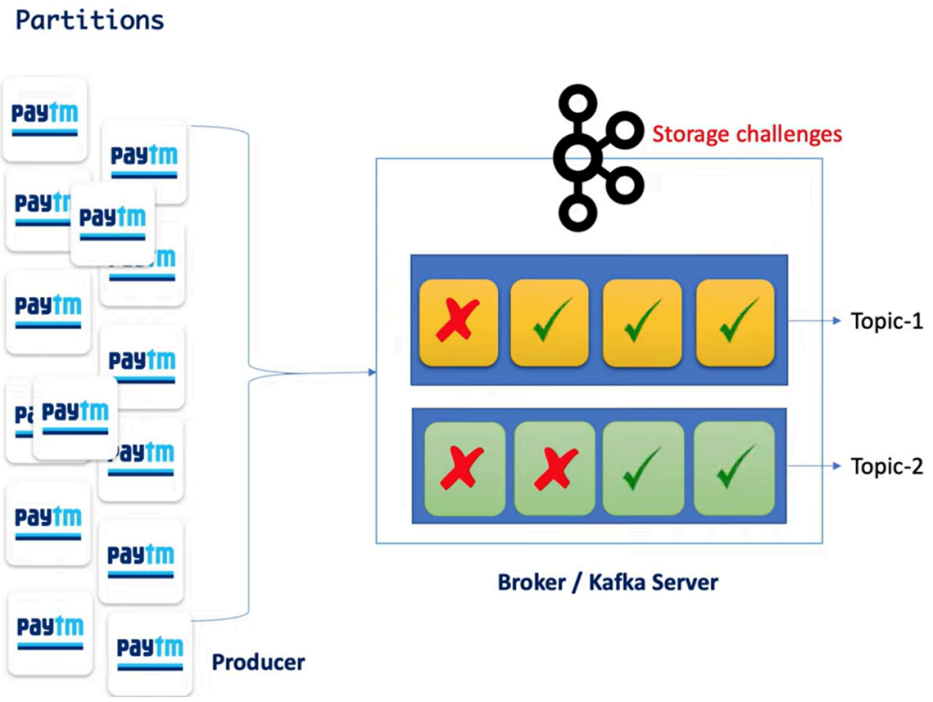


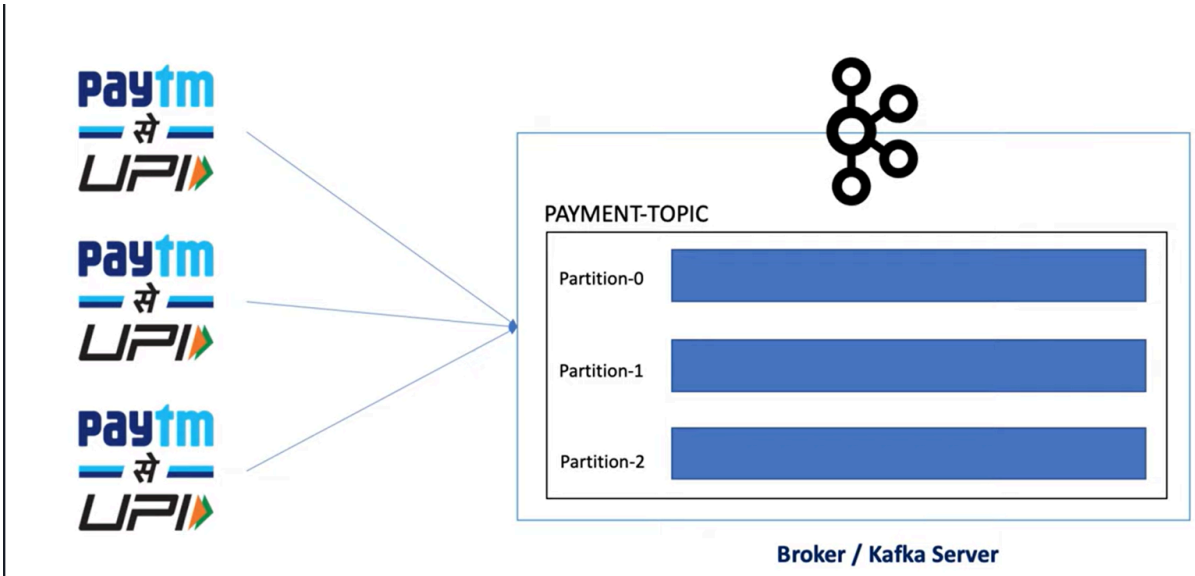
Like payment related messages binds on payment topic ,booking related messages binds on booking topic and insurances related topic binds on insurance topics.



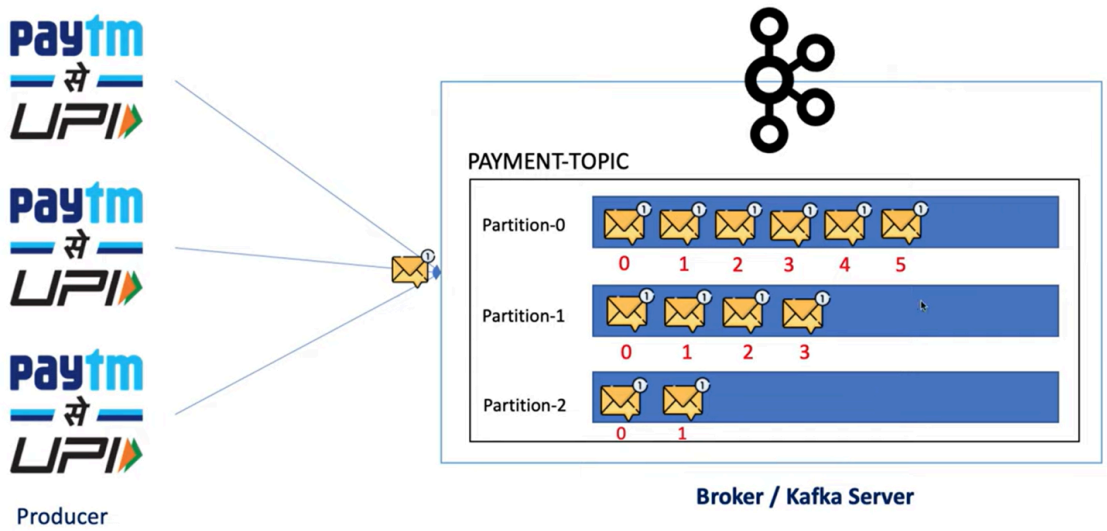
Partition: A topic is divided into partitions, which are ordered, immutable sequences of records. Each record within a partition has a unique offset. Partitions allow Kafka to parallelize processing and provide scalability.

When we have a huge volume of data then we need to store messages into multiple partitions. We can do topic partition ,we can split the topic into multiple partitions . If any of the topics goes down another topic in the partition will work.

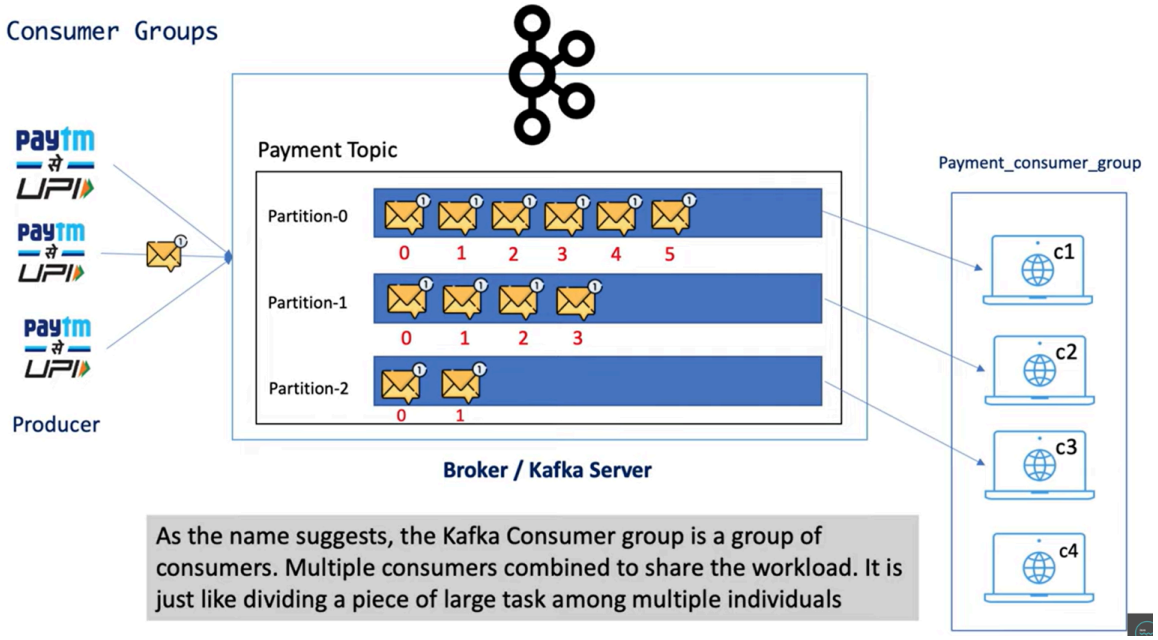




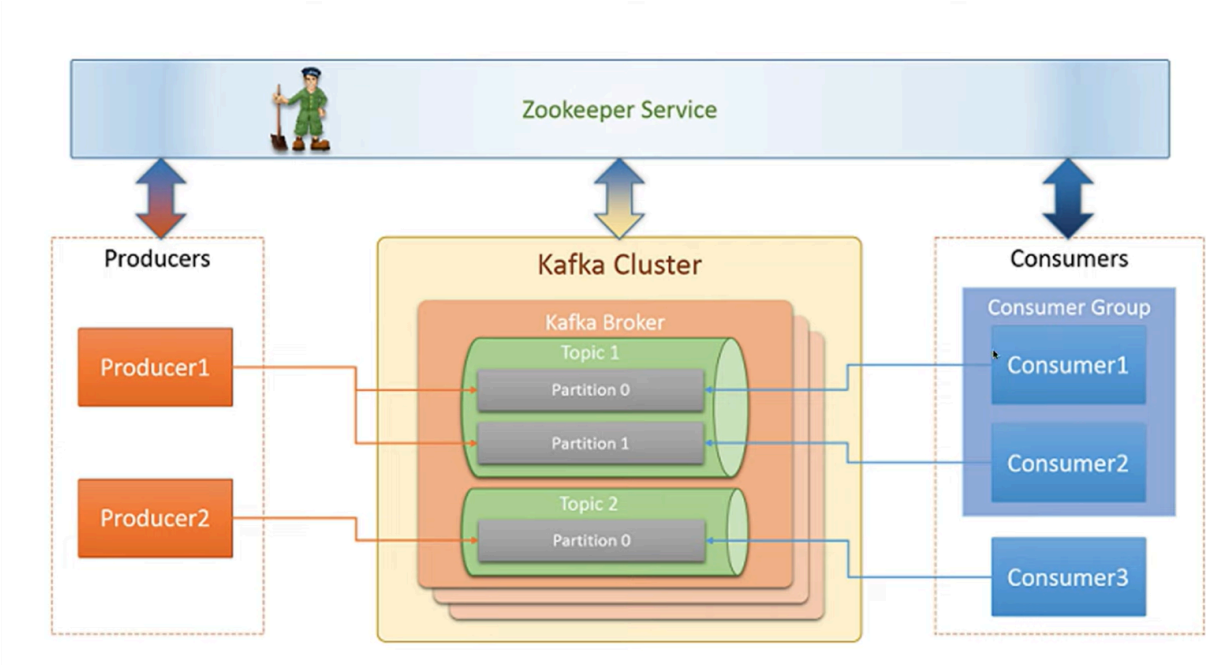
Offset : Offset is a unique, sequential identifier assigned to each message within a partition. It represents the position of a message in the partition's log and is crucial for tracking and managing message consumption.



Consumer Group: when we have multiple partitions then a single consumer may not work in an optimised way so we can group consumers according to the topic partitions.All the consumer can read the data from topic parallelly. If we add an extra one more consumer then that consumer will sit idle if any of the consumer goes down that idle consumer will work ,that's called consumer rebalancing.



ZooKeeper: A centralized service used by Kafka to manage and coordinate the cluster. ZooKeeper keeps track of broker metadata, leader election for partitions, and configuration management.



Kafka CLI

- Open your terminal and navigate to the directory where you extracted the Kafka zip or tar file:

```
$ cd kafka_2.13-3.1.0
```

- Run the following command to launch the ZooKeeper service:

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

- In another terminal, run the following command to start the Kafka broker service:

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

Create a Topic

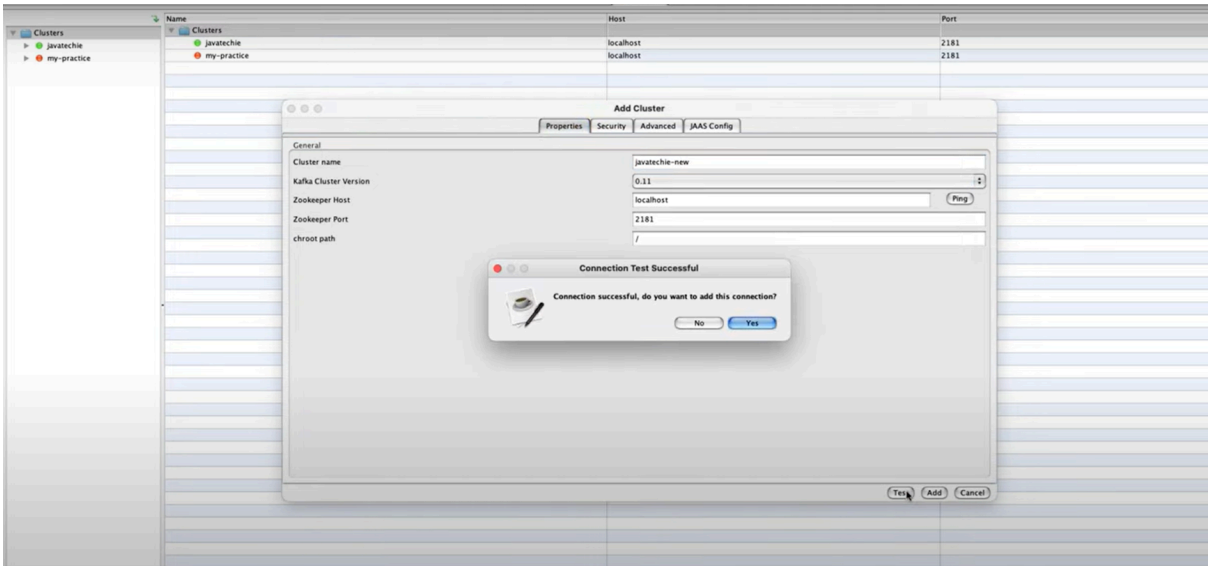
A topic in Kafka is akin to a directory or folder on your computer. The only difference is that a topic stores events (records/messages) rather than files and these events are normally distributed across multiple nodes/brokers.

Events can be application logs, web clickstreams, data emitted by IoT sensors, and much more. So before creating events, the first step is to create a topic or topics that will store and organize these records.

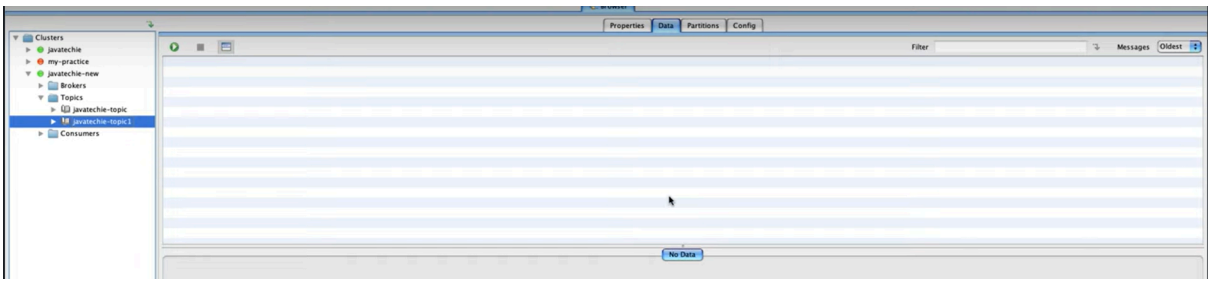
To create a Kafka topic using the Kafka CLI, you will use the **bin/kafka-topics.sh** shell script that's bundled in the downloaded Kafka distribution. Launch another terminal session and execute the following command:

```
$ bin/kafka-topics.sh --create --topic my-topic --bootstrap-server
localhost:9092 --partitions 3 --replication-factor 1
```

Create a cluster



Check data in topics



Start the producer

bin/kafka-console-producer.sh --bootstrap-server localhost:9092 --topic my-topic

🚀 Starting a Kafka Console Producer

Open your terminal and execute the following command:

```
bin/kafka-console-producer.sh --bootstrap-server localhost:9092
--topic my-topic
```

This command connects to the Kafka broker running on `localhost:9092` and targets the topic named `my-topic`.GeeksforGeeks

Once the producer starts, you'll see a prompt (`>`) indicating it's ready to accept input. You can type your messages directly, and each line you enter will be sent as a separate message to the specified Kafka topic.GeeksforGeeks

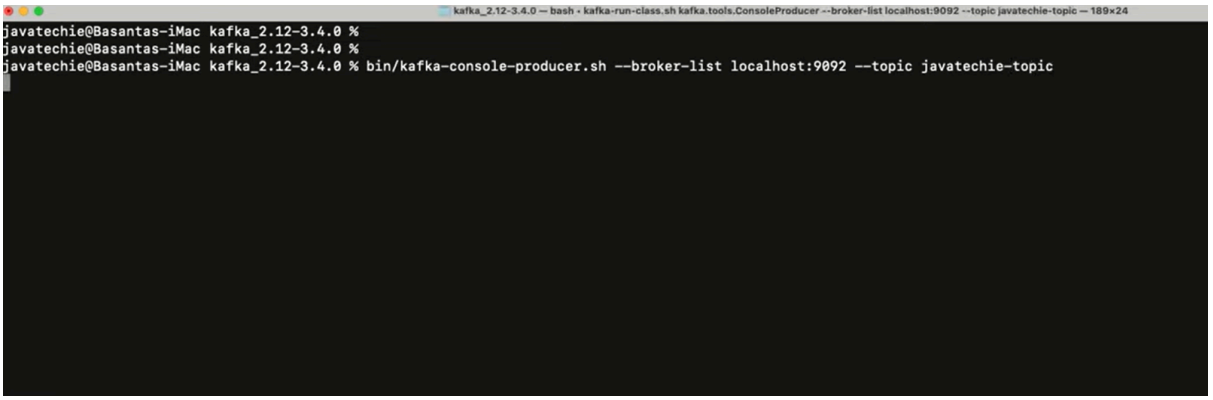
---

🔑 Sending Key-Value Messages

If you want to send messages with specific keys (useful for partitioning), you can use the `--property` option to define a key-value separator. For example:

```
bin/kafka-console-producer.sh \
  --bootstrap-server localhost:9092 \
  --topic my-topic \
  --property "parse.key=true" \
  --property "key.separator=:"
```

With this setup, you can input messages in the format `key:value`. Kafka will parse the key and value accordingly, which can be beneficial for message routing and partitioning.



Start the Consumer

To start a Kafka consumer using the command line, you can utilize the kafka-console-consumer.sh

📥 Starting a Kafka Console Consumer

To consume messages from a Kafka topic named my-topic, run the following command:Apps Developer Blog

```
bin/kafka-console-consumer.sh --bootstrap-server localhost:9092
--topic my-topic
```

This command connects to the Kafka broker at localhost:9092 and starts consuming messages from the my-topic topic. By default, it will only display new messages that are produced after the consumer starts.GeeksforGeeks

---

🔁 Consuming Messages from the Beginning

If you want to read all existing messages from the beginning of the topic, include the --from-beginning flag:

```
bin/kafka-console-consumer.sh --bootstrap-server localhost:9092
--topic my-topic --from-beginning
```

This will display all historical messages in the topic, followed by any new messages as they arrive.Datacadamia - Data and Co+5GeeksforGeeks+5Stack Overflow+5

🔑 Displaying Message Keys

To display both the key and value of each message, use the following command:

```
bin/kafka-console-consumer.sh --bootstrap-server localhost:9092
--topic my-topic --from-beginning --property print.key=true
--property key.separator=":"
```

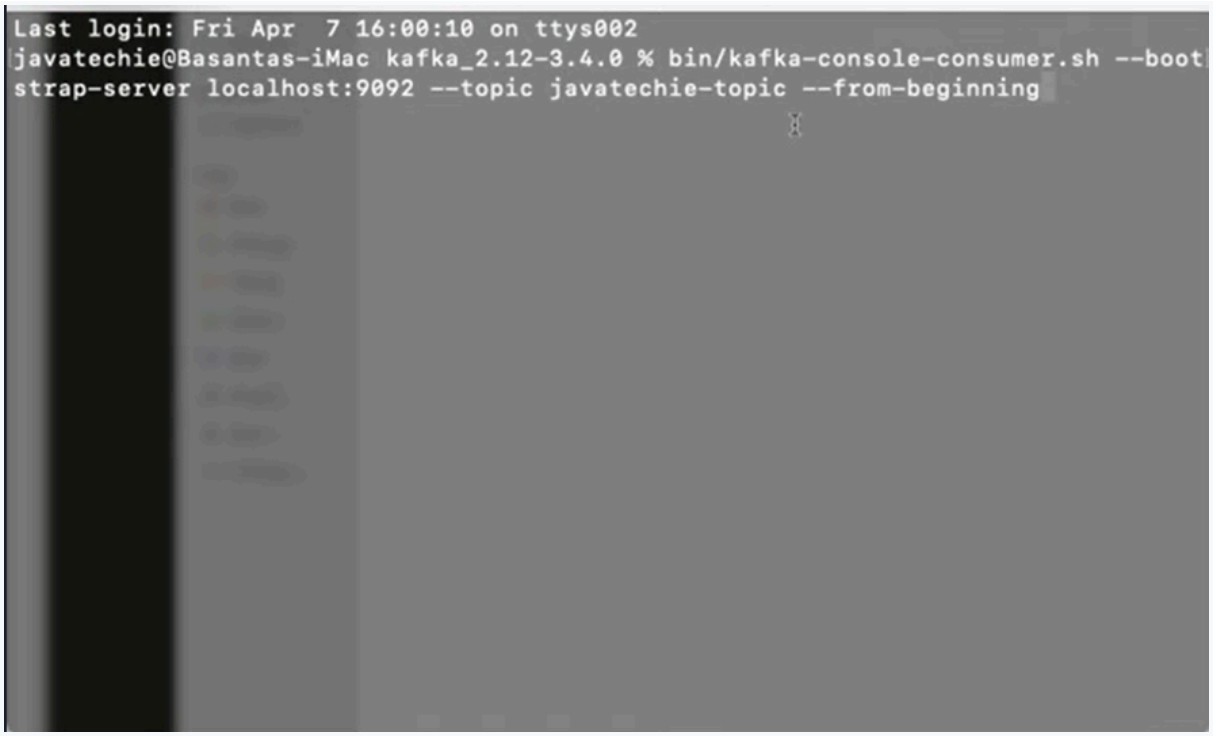
This will output messages in the format key:value, allowing you to see the associated keys for each message.

👥 Using Consumer Groups

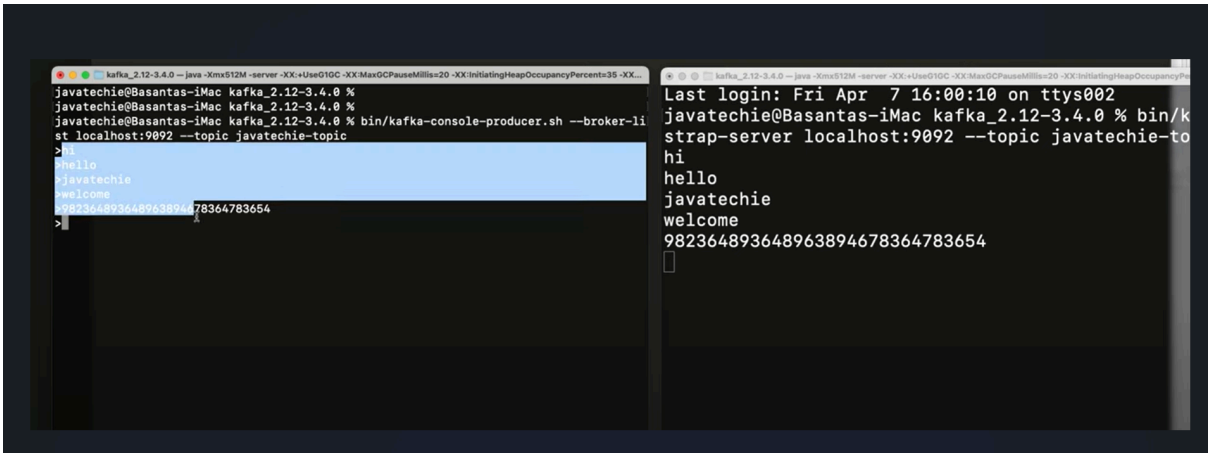
To associate the consumer with a specific consumer group, which enables load balancing and fault tolerance, use the --group option:

```
bin/kafka-console-consumer.sh --bootstrap-server localhost:9092
--topic my-topic --group my-consumer-group
```

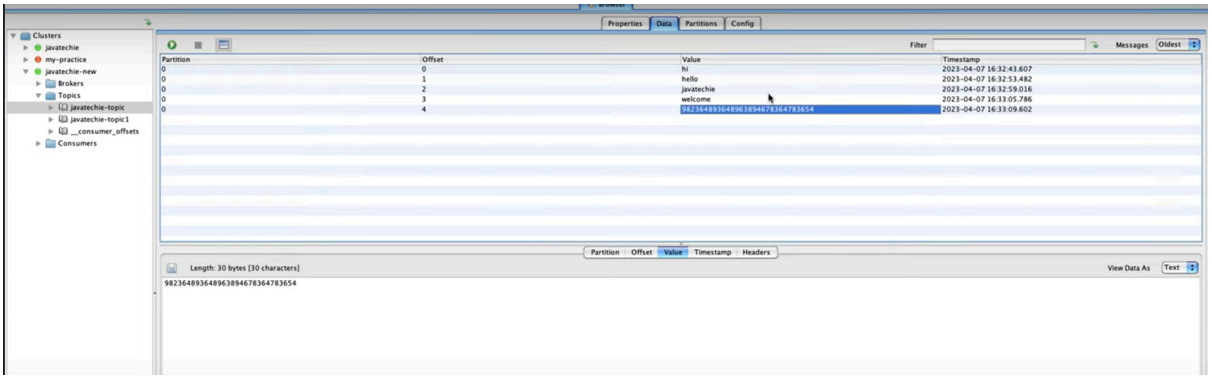
This allows multiple consumers in the same group to share the consumption of messages from the topic.



Send the data from one producer terminal to consumer terminal



Check messages in kafka dashboard



Send messages by .csv files

We will get all the messages in consumer terminal

Confluent Kafka Community Edition in local

1. **Start Zookeeper Server**

   ```
   bin/zookeeper-server-start etc/kafka/zookeeper.properties
   ```

2. **Start Kafka Server / Broker**

```
bin/kafka-server-start etc/kafka/server.properties
```

3. **Create topic**

```
bin/kafka-topics --bootstrap-server localhost:9092 --create --topic
NewTopic1 --partitions 3 --replication-factor 1
```

4. **List out all topic names**

```
bin/kafka-topics --bootstrap-server localhost:9092 --list
```

5. **Describe topics**

6. `bin/kafka-topics --bootstrap-server localhost:9092 --describe --topic NewTopic1`

6. **Produce message**

```
bin/kafka-console-producer --broker-list localhost:9092 --topic NewTopic1
```

7. **Consume message**

```
bin/kafka-console-consumer --bootstrap-server localhost:9092 --topic NewTopic1 --from-beginning
```

8. **Send CSV File data to kafka**

```
bin/kafka-console-producer --broker-list localhost:9092 --topic NewTopic1 <bin/customers.csv
```

Install kafka using docket

✅ Step 1: Create a Docker Compose file

Create a file named `docker-compose.yml`:

```
version: '3'
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    ports:
      - "2181:2181"

  kafka:
    image: confluentinc/cp-kafka:latest
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
    environment:
```

```
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
```

✅ Step 2: Start Kafka and Zookeeper

Run this command in the directory where your `docker-compose.yml` file is located:

```
docker-compose up -d
```

- This will pull the required images and start the services in detached mode.

✅ Step 3: Verify Installation

Check if Kafka and Zookeeper containers are running:

```
docker ps
```

✅ Step 4: Access Kafka CLI (Optional)

You can access the Kafka container's shell using:

```
docker exec -it <kafka_container_id> bash
```

From inside the container, you can run Kafka commands like creating topics, producing, or consuming messages.

✅ Stop Services

To stop Kafka and Zookeeper:

```
docker-compose down
```

Reference : https://github.com/Java-Techie-jt/kafka-installation/blob/main/README.md

✅ Step-by-Step: Kafka Producer with Spring Boot

1️⃣ Add Maven Dependencies (`pom.xml`)

```
<dependencies>
    <!-- Spring Boot Starter Web -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- Spring for Apache Kafka -->
    <dependency>
        <groupId>org.springframework.kafka</groupId>
        <artifactId>spring-kafka</artifactId>
    </dependency>
</dependencies>
```

2️⃣ Add Kafka Config in `application.yml` or `application.properties`

`application.yml`:

```
spring:
  kafka:
    bootstrap-servers: localhost:9092
    producer:
      key-serializer:
org.apache.kafka.common.serialization.StringSerializer
```

```
      value-serializer:
org.apache.kafka.common.serialization.StringSerializer
```

3️⃣ Create Kafka Producer Configuration Class

```
import org.apache.kafka.clients.producer.ProducerConfig;

import org.apache.kafka.common.serialization.StringSerializer;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.kafka.core.DefaultKafkaProducerFactory;

import org.springframework.kafka.core.KafkaTemplate;

import org.springframework.kafka.core.ProducerFactory;


import java.util.HashMap;

import java.util.Map;


@Configuration

public class KafkaProducerConfig {


    @Bean

    public ProducerFactory<String, String> producerFactory() {

        Map<String, Object> config = new HashMap<>();

        config.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");

        config.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
```

```
        config.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        return new DefaultKafkaProducerFactory<>(config);
    }

    @Bean
    public KafkaTemplate<String, String> kafkaTemplate() {
        return new KafkaTemplate<>(producerFactory());
    }
}
```

4️⃣ Create a Kafka Producer Service

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Service;

@Service
public class KafkaProducerService {

    private static final String TOPIC = "NewTopic1";

    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;
```

```
    public void sendMessage(String message) {
        kafkaTemplate.send(TOPIC, message);
    }
}
```

5️⃣ Create a REST Controller to Trigger Message Sending

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/kafka")
public class KafkaProducerController {

    @Autowired
    private KafkaProducerService producerService;

    @PostMapping("/publish")
    public String publishMessage(@RequestParam("message") String message) {
        producerService.sendMessage(message);
        return "Message sent to Kafka topic successfully!";
    }
}
```

✅ Test with Postman or CURL

```
curl -X POST
"http://localhost:8080/api/kafka/publish?message=HelloKafka"
```

You should see:
"Message sent to Kafka topic successfully!"

reference: https://github.com/Java-Techie-jt/kafka-producer-example/blob/main/README.md

✅ Kafka Consumer in Spring Boot

1️⃣ Add Maven Dependencies (if not already added in `pom.xml`):

```
<dependencies>

    <!-- Spring Boot Web -->

    <dependency>

        <groupId>org.springframework.boot</groupId>

        <artifactId>spring-boot-starter-web</artifactId>

    </dependency>


    <!-- Spring Kafka -->

    <dependency>

        <groupId>org.springframework.kafka</groupId>

        <artifactId>spring-kafka</artifactId>

    </dependency>

</dependencies>
```

2️⃣ Add Kafka Configuration in `application.yml` or `application.properties`

`application.yml`:

```
spring:

  kafka:

    bootstrap-servers: localhost:9092

    consumer:

      group-id: my-consumer-group

      key-deserializer:
org.apache.kafka.common.serialization.StringDeserializer

      value-deserializer:
org.apache.kafka.common.serialization.StringDeserializer

      auto-offset-reset: earliest
```

3️⃣ Create Kafka Consumer Service

```
import org.springframework.kafka.annotation.KafkaListener;

import org.springframework.stereotype.Service;


@Service

public class KafkaConsumerService {


    @KafkaListener(topics = "NewTopic1", groupId =
"my-consumer-group")
```

```
    public void consume(String message) {
        System.out.println("Received message: " + message);
    }
}
```

✅ What This Does:

- Listen to the topic **`NewTopic1`**.
- Automatically receives messages produced by any Kafka producer to that topic.
- Belongs to the consumer **group** `my-consumer-group`.

🔁 Testing

1. Start your Kafka broker and Zookeeper.
2. Run your Spring Boot application.

Publish a message using:
`curl -X POST "http://localhost:8080/api/kafka/publish?message=HelloFromProducer"`

3.
4. Check the logs — the message should appear in the console.

Reference : https://github.com/Java-Techie-jt/kafka-consumer-example

Apache Kafka JSON Serialization & Deserialization

GitHub:
https://github.com/Java-Techie-jt/kafka-producer-example
https://github.com/Java-Techie-jt/kafka-consumer-example

To **send a Kafka message to a specific partition** in **Spring Boot**, you can use the `KafkaTemplate` and explicitly provide the partition number.

```
    private KafkaTemplate<String, String> kafkaTemplate;

    public void sendToPartition(String topic, String message, int partition) {
        kafkaTemplate.send(topic, partition, null, message);
        System.out.println("Sent to partition " + partition + ": " + message);
    }
```

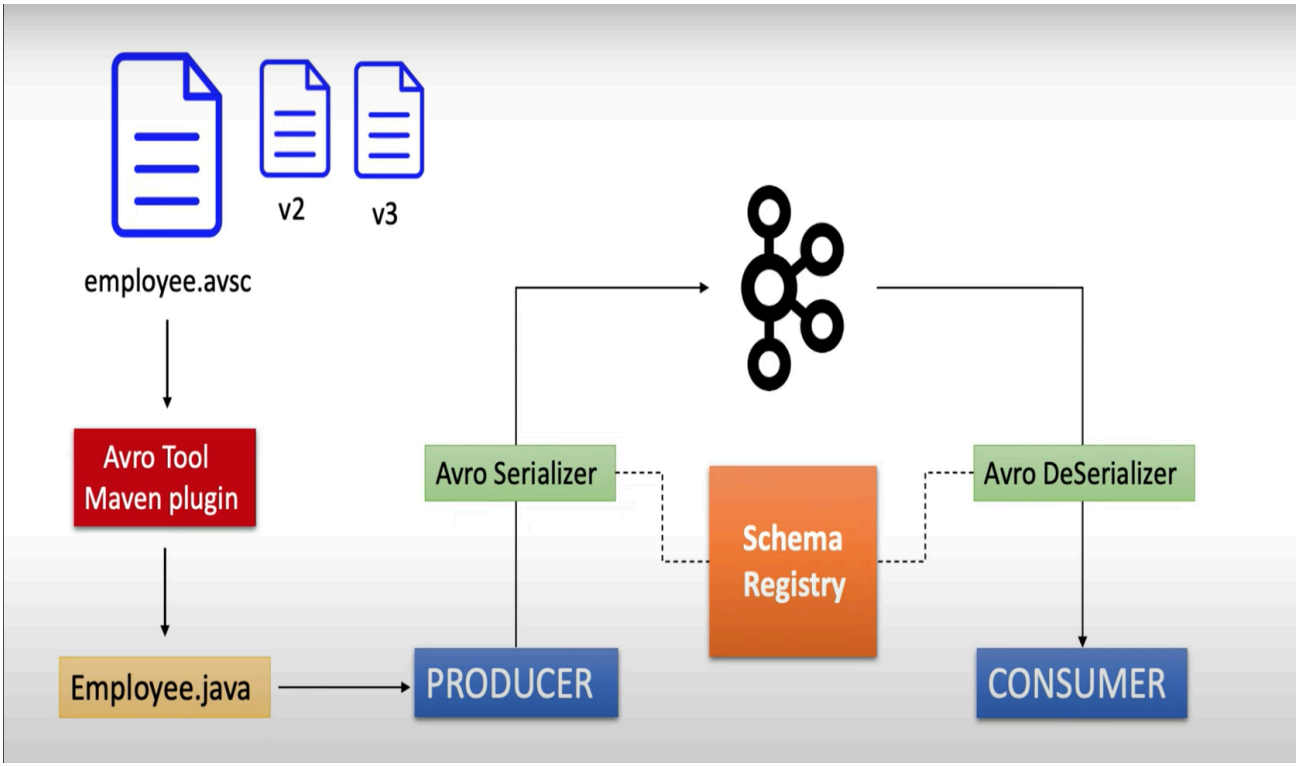
Consume from a Specific Partition

```
  // Listens to only partition 0 of topic 'NewTopic1'
    @KafkaListener(
        topicPartitions = @TopicPartition(topic = "NewTopic1", partitions = { "0" }),
        groupId = "my-partitioned-group"
    )
    public void listenPartition0(ConsumerRecord<String, String> record) {
        System.out.println("Received message from partition 0: " + record.value());
    }
```

📘 What is Kafka Schema Registry?

Schema Registry is a **centralized service** for storing and retrieving **schemas** (data structure definitions) used by Kafka messages—especially when using **Avro**, **Protobuf**, or **JSON Schema**.

> It is a part of the Confluent platform (but you can use open-source alternatives too).



🧠 Why Do We Need Schema Registry?

Without Schema Registry:

- You serialize data, but **no one knows its structure** unless they have prior knowledge.
- If a consumer expects `field A`, but the producer sends `field B`, it **breaks the pipeline**.

With Schema Registry:

- Both **producers and consumers** know the **structure** of the data via the **schema ID** embedded in the message.
- Supports **versioning and compatibility checks**, making microservices more **robust and scalable**.

💡 Core Features of Schema Registry

Feature	Description

Centralized Schema Storage	Schemas are registered and stored in a centralized registry.
Version Control	Each schema gets a version and can evolve over time.
Schema Evolution	Supports compatibility settings like Backward, Forward, Full, and None.
Integration with Kafka	Works smoothly with Kafka producers/consumers using Avro/Protobuf/JSON.
REST APIs	Exposes REST APIs to register, fetch, or validate schemas.

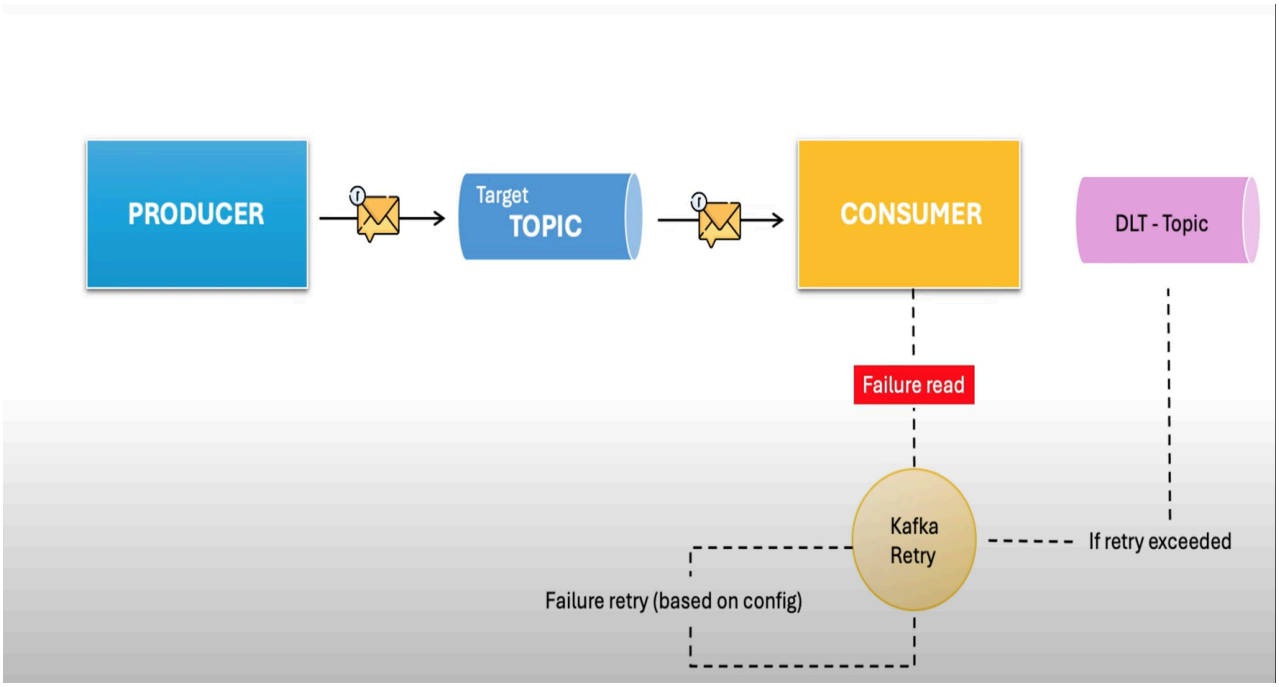
🧩 How Schema Registry Works with Avro in Kafka

🔁 Message Flow

1. **Producer side**:
    - Serializes the data using **KafkaAvroSerializer**
    - Registers the schema with Schema Registry
    - Embeds a **schema ID** in the message
2. **Consumer side**:
    - Deserializes the message using **KafkaAvroDeserializer**
    - Uses schema ID to fetch the **latest or matching schema**
    - Converts bytes back to structured object

Reference : https://github.com/Java-Techie-jt/spring-kafka-avro

Kafka Error Handling with Spring Boot — In Detail



1. Background: Why Error Handling Matters

When consuming messages from Kafka, you often encounter errors like:

- Temporary failures: downstream service down, network glitch, DB locked
- Permanent failures: malformed messages, schema mismatch, validation errors

If you don't handle errors, your consumer might crash, or the problematic message could be skipped or lost silently.

Goal:

- Automatically retry transient errors a few times.
- If retries fail, send messages to a **Dead Letter Topic** (DLT) for later inspection.
- Have a dedicated handler for the DLT topic to manually fix or alert.

---

2. Spring Boot + Kafka Basics

Add these dependencies in your `pom.xml` or `build.gradle`:

```
<dependency>
```

```
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.retry</groupId>
    <artifactId>spring-retry</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId> <!--
Required for @Retryable -->
</dependency>
```

---

3. Retry Strategy with `@Retryable`

3.1 Enable Retry in your Spring Boot app

Add `@EnableRetry` to your main app or config:

```
@SpringBootApplication
@EnableRetry
public class KafkaErrorHandlingApplication {
    public static void main(String[] args) {

SpringApplication.run(KafkaErrorHandlingApplication.class,
args);
    }
}
```

3.2 Consumer with Retryable method

```
@Component
public class MyKafkaConsumer {

    @KafkaListener(topics = "my-topic", groupId = "group1")
    @Retryable(
        value = { TransientException.class },
        maxAttempts = 4,
        backoff = @Backoff(delay = 3000, multiplier = 2))
    public void listen(String message) {
```

```
        System.out.println("Processing message: " + message);

        if (message.contains("transient-error")) {
            throw new TransientException("Temporary failure,
retrying...");
        }

        if (message.contains("permanent-error")) {
            throw new IllegalArgumentException("Bad message,
no retry");
        }

        // normal processing here
    }

    @Recover
    public void recover(TransientException e, String message)
{
        System.err.println("Retries exhausted for message: " +
message);
        // Here you can manually send to DLT, log, or alert
    }
}
```

Explanation:

- `@Retryable` retries method on exceptions you specify (`TransientException` here).
- `maxAttempts` = total attempts (initial + retries).
- `@Backoff` defines delay and multiplier between retries.
- `@Recover` is called when retries are exhausted.

---

4. Dead Letter Topic (DLT) Setup

When retries fail, you want to **automatically send** the problematic message to a DLT for later manual handling or alerting.

4.1 How Spring Kafka supports DLTs

Spring Kafka's **ErrorHandler** interfaces let you customize what happens on failure.

Use the `DefaultErrorHandler` (introduced in Spring Kafka 2.8+), which can automatically send failed messages to DLT topics.

4.2 Configure DLT with `DefaultErrorHandler`

```
@Bean
public DefaultErrorHandler errorHandler(KafkaTemplate<String,
String> kafkaTemplate) {
    // Create a DeadLetterPublishingRecoverer to send messages
to DLT
    DeadLetterPublishingRecoverer recoverer = new
DeadLetterPublishingRecoverer(kafkaTemplate,
        (record, ex) -> new TopicPartition(record.topic() +
".DLT", record.partition()));

    // Backoff policy: initial delay 1s, multiplier 2, max
delay 10s
    FixedBackOff fixedBackOff = new FixedBackOff(1000L, 3L);
// 3 retries, 1s apart

    // The error handler with recoverer and backoff
    DefaultErrorHandler errorHandler = new
DefaultErrorHandler(recoverer, fixedBackOff);

    // You can add exceptions to not retry, e.g.:

errorHandler.addNotRetryableExceptions(IllegalArgumentExceptio
n.class);

    return errorHandler;
}
```

4.3 Attach the error handler to your KafkaListener container factory

```
@Bean
public ConcurrentKafkaListenerContainerFactory<String, String>
kafkaListenerContainerFactory(
        ConsumerFactory<String, String> consumerFactory,
        DefaultErrorHandler errorHandler) {

    ConcurrentKafkaListenerContainerFactory<String, String>
factory =
            new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory);
    factory.setCommonErrorHandler(errorHandler);  // Attach
the error handler

    return factory;
}
```

Explanation:

- `DeadLetterPublishingRecoverer` routes failed messages to `original-topic.DLT` (default naming).
- `FixedBackOff` sets retry attempts before sending to DLT.
- `DefaultErrorHandler` manages retries and sends to DLT after retries fail.
- Exceptions like `IllegalArgumentException` can be marked as non-retryable and send to DLT immediately.

---

5. Handling messages from the DLT with `@Dlthandler`

You want to consume and process DLT messages for monitoring, alerting, or manual fixing.

5.1 Example DLT handler

```
@Component
public class MyDltHandler {
```

```
    @KafkaListener(topics = "my-topic.DLT", groupId =
"dlt-group")
    @Dlthandler
    public void handleDltMessages(String message) {
        System.err.println("Received from DLT: " + message);
        // Process the failed message: alerting, logging, or
manual fix
    }
}
```

---

6. Full flow summary

What happens
Kafka consumer receives message
`@Retryable` or `DefaultErrorHandler` retries transient failures
After max retries, if still failing, message sent to DLT topic
Dedicated `@Dlthandler` listens on DLT and handles failed messages

---

7. Optional: Custom Exception Classes

Example for your transient exceptions:

```
public class TransientException extends RuntimeException {
    public TransientException(String message) {
        super(message);
    }
}
```

---

8. Example Application Properties for Kafka

```
spring:
  kafka:
    bootstrap-servers: localhost:9092
    consumer:
      group-id: group1
      auto-offset-reset: earliest
    producer:
      key-serializer:
org.apache.kafka.common.serialization.StringSerializer
      value-serializer:
org.apache.kafka.common.serialization.StringSerializer
```

Bonus: Additional tips

- Use `RetryTemplate` if you want programmatic retries instead of `@Retryable`.
- DLT topic naming convention: `<original-topic>.DLT`
- You can customize the `DeadLetterPublishingRecoverer` to change topic or partition.
- Monitor your DLT topics regularly.
- Consider alerting on certain types of errors in DLT.

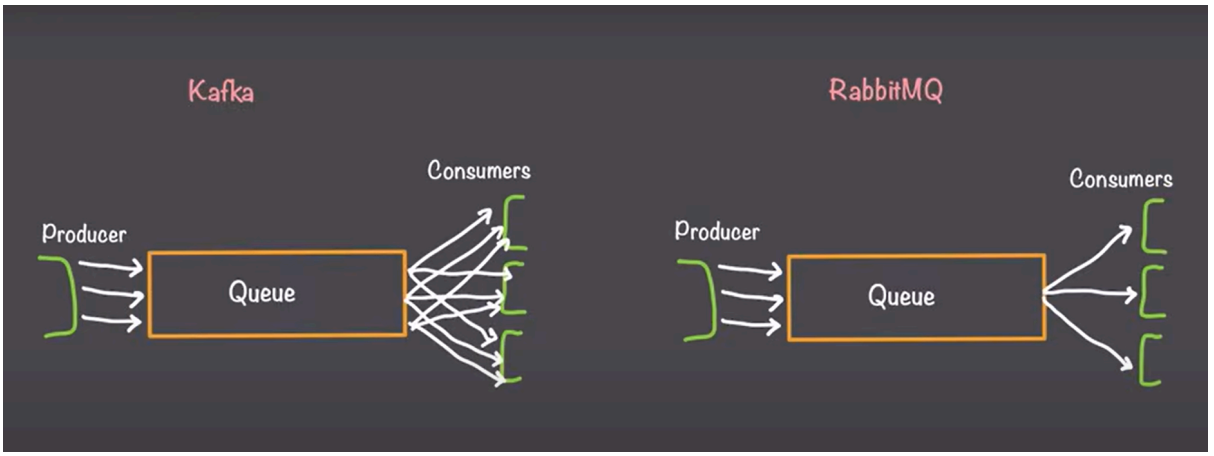
Reference : https://github.com/Java-Techie-jt/kafka-error-handling

When to use RabbitMq and kafka

With distributed systems, a common mistake is thinking that these two systems are interchangeable, but they actually solve very different purposes, and using one of them when you should be using the other can cause a lot of problems down the road.

Kafka is inherently a stream processing system, so it's designed for taking in a stream of events and sending them off to a bunch of different consumers of those events. Kafka has very high throughput, and it keeps all the messages around until their time to live expires, so even messages that have already been consumed can still be replayed later. Kafka is also fan-out by default.

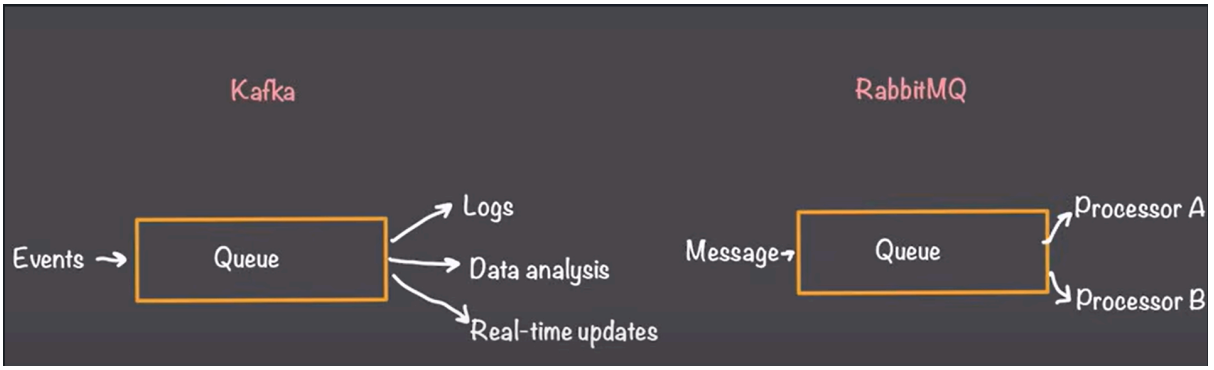
Three consumers connected to one queue, and we send three messages into that queue, each consumer will receive all three messages. With RabbitMQ, on the other hand, each consumer will receive one of the three messages.So Kafka and RabbitMQ both support doing this the other way around as well, but it requires a little bit more setup and isn't quite as scalable.



For Kafka we could have a stream of events coming in and we want to send those out to logging, we want to do some data analytics on those events, and we want to send real-time updates to our users. Each event that's coming in needs to be processed by each one of these services.

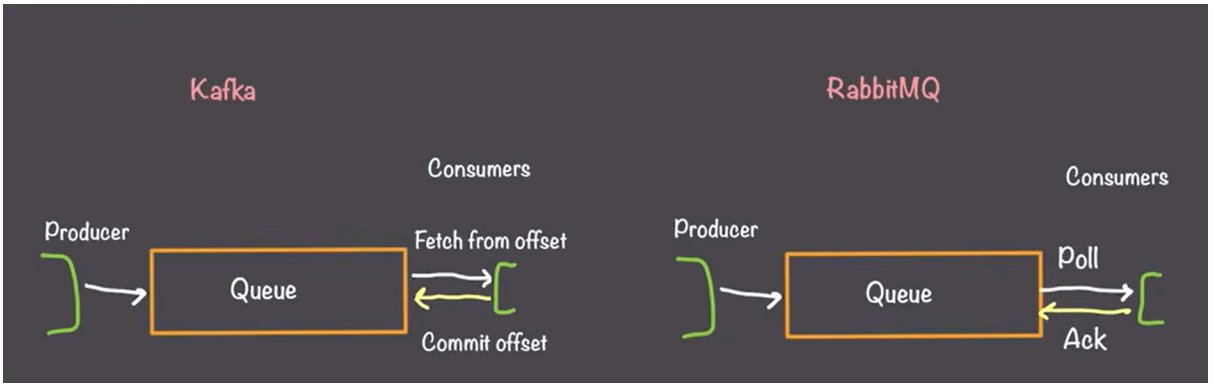
However, the individual job of these services is relatively small, and likely doesn't need to be scaled horizontally too much. This means that most of our messages are simply fanning out to each of these three consumers. RabbitMQ, on the other hand,

is great for situations where we have messages coming in, and we need to process those messages. So in this example, we have two processors, and any message that comes in needs to be read by one of these two processors.



Now the final piece that we're going to talk about here is acknowledgment. So if something goes wrong and a consumer fails to process a message, we need some way to be able to retry and send that message off to a different consumer. So that's where acknowledgments come into play. So with the Kafka model, we don't actually have acknowledgments. We instead have offsets.

So Kafka logs an offset of how many messages each consumer has received so far. And whenever a consumer needs new data, it fetches data from the topic from that offset. So it's getting all new data from the last position it read from. Once it's done processing that batch of data, it then commits its offsets to tell Kafka that it actually successfully processed that data. If a consumer disconnects before it commits its offsets, Kafka will be able to automatically send that data to another consumer, because that consumer will just pick up from the last committed offset.



RabbitMQ model tends to work better when we have long-running tasks, and we need to acknowledge those tasks as completed or failed.The model of committing offsets with Kafka is great when we have to process batches of data when we have a large quantity of small events coming in.

🆚 RabbitMQ vs Kafka — When to Use What?

Use Case	Use RabbitMQ	Use Kafka
🔁 **Short-lived messages**	Yes — Good for quick, transient messages	No — Not ideal for short-term messaging
🕒 **Event replay**	❌ Not supported easily	✅ Built-in (can replay from log anytime)
🔄 **Traditional queueing**	✅ Excellent — FIFO, routing, retries	Not the main use case
🧵 **Request/Response**	✅ Built for this pattern	❌ Not ideal
📊 **Stream processing**	❌ Not designed for high-throughput analytics	✅ Best choice for real-time stream processing
📦 **Message durability**	Good (ack, persistence)	Excellent — immutable logs and high fault tolerance
🧠 **Ordering guarantees**	In a queue, yes — but per consumer only	✅ Ordering by partition
🚀 **High throughput**	Good (medium throughput)	✅ Very high throughput (millions of messages/sec)
🌐 **Microservices**	✅ Good for async communication	✅ Great for event-driven architectures
📂 **Audit/logging data**	❌ Not meant for large retention	✅ Perfect for log/event storage over time
🕒 **Message Retention**	Short (usually deleted after being read)	Long (you decide how long to keep it)

🧪 Real-World Examples

✅ Use RabbitMQ When:

- You need **classic messaging patterns** like pub-sub, topics, routing, fan-out.
- You require **low-latency messaging** with simple retry and dead-letter queues.
- You're building **backend services**, APIs, or **real-time task queues**.
- You want **acknowledgements, TTL, and retry features** out of the box.

Example Use Cases:

- Background job queues (e.g., sending emails, image processing)
- Short-term event communication between microservices
- RPC-like messaging patterns

✅ Use Kafka When:

- You need **event sourcing**, **audit logging**, or **stream processing**.
- You expect a huge **volume** of data (metrics, clickstreams, IoT).
- You want **event replay**, **message retention**, and high scalability.
- You're building a **data pipeline** or **real-time analytics system**.

Example Use Cases:

- Logging platform (like ELK, Splunk)
- Financial transactions or audit logs
- Real-time analytics (clicks, sensors, events)
- Streaming ETL pipelines (Kafka → Spark/Flink → DB)

🧠 Simple Analogy

Analogy	RabbitMQ	Kafka
Like a 📬 mailbox	Delivers a message once	Stores a message log forever
Memory scope	Short-term, "send and forget"	Long-term, "store and replay"
Ideal for	Jobs and tasks	Events and history

🚨 Summary

Feature / Need	Use RabbitMQ	Use Kafka
Real-time task queues	✅	❌
Durable event logs	❌	✅
Microservice comms	✅	✅
Stream analytics (Flink)	❌	✅
High-throughput ingestion	❌	✅