Palash Bajpai

**Kafka**

Table of Contents

[**1.** **What is Spring framework?** 2](#_Toc189042511)

[1.1 Inversion of Control(IoC) 3](#_Toc189042512)

[1.2 Dependency Injection 4](#_Toc189042513)

[1.3 Types of Dependency Injection 7](#_Toc189042514)

[1.4 Types of Annotations 11](#_Toc189042515)

[**2.** **What is Spring Boot?** 13](#_Toc189042516)

[2.1. Key Features of Spring boot 13](#_Toc189042517)

[How is Spring Boot Different from Spring? 14](#_Toc189042518)

[2.2. Configuration Properties: 16](#_Toc189042519)

[2.2. Embedded Servers 17](#_Toc189042520)

[2.3 Actuators 18](#_Toc189042521)

[**3. Web Service** 19](#_Toc189042522)

[3.1 Types of web service 20](#_Toc189042523)

[**4. REST API with Spring Boot** 23](#_Toc189042524)

[4.1. Security 23](#_Toc189042525)

[**5. Intro to Microservice** 24](#_Toc189042526)

[5.1. Monolithic Architecture 25](#_Toc189042527)

[5.2. How do Microservices communicate? 26](#_Toc189042528)

[**6. Interview Questions** 29](#_Toc189042529)

[Why Use Spring Boot? 29](#_Toc189042530)

[Difference Between @Controller and @RestController 29](#_Toc189042531)

[Examples 30](#_Toc189042532)

# **1. Intro to Kafka**

**Apache Kafka** is a distributed, high-throughput, fault-tolerant messaging system used for building real-time data pipelines and streaming applications. It enables services to communicate asynchronously by publishing and subscribing to streams of records (events) in a decoupled, scalable manner.

**Key points:**

1. **Distributed Messaging System**: Apache Kafka is a **distributed publish-subscribe system** used for building real-time data pipelines and event-driven architectures.

2. **High Throughput & Low Latency**: Kafka handles **millions of messages per second** with **low latency**, making it ideal for large-scale systems.

3. **Topics and Partitions**: Data is organized into **topics**, which are split into **partitions** for scalability and parallel processing.

4. **Producers and Consumers**: **Producers** send data to topics, **Consumers** read data from topics, often as part of a **consumer group**.

5. **Fault Tolerant & Durable**: Kafka stores messages on disk and replicates them across brokers, ensuring **no data loss** even if servers crash.

6. **Replayable Events**: Messages are retained for a configured time, allowing **replay** and **reprocessing** by consumers.

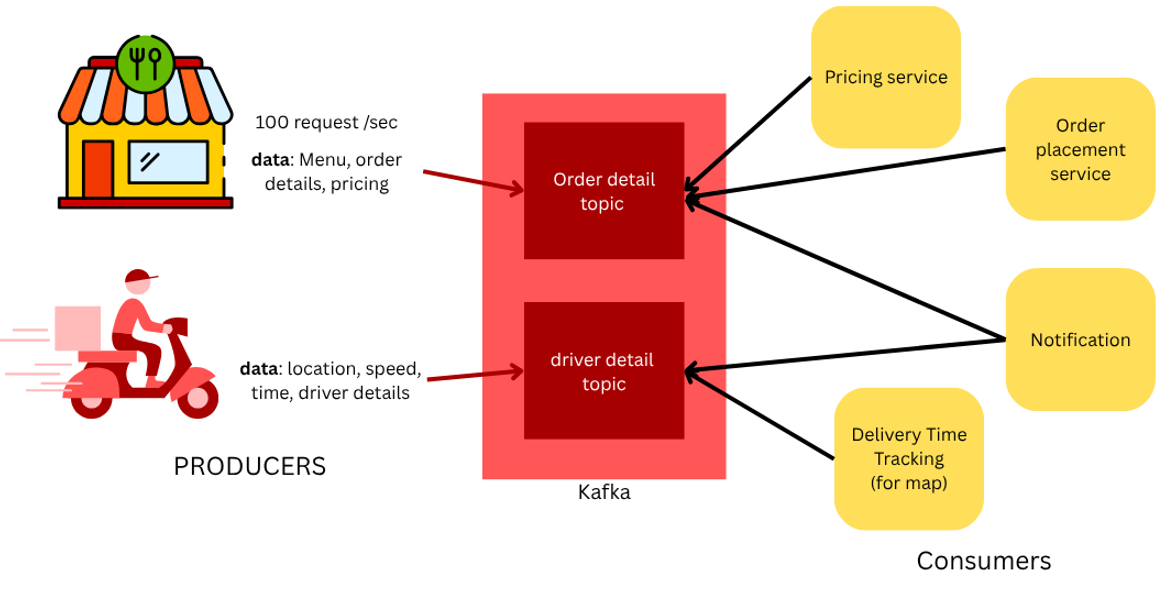
7. **Decouples Services**: Kafka **decouples microservices**, enabling asynchronous communication and reducing system dependency.

8. **Horizontally Scalable**: Kafka scales easily by adding more brokers and partitions to handle growing loads.

9. **Real-Time Data Streaming**: Kafka supports **real-time analytics and processing**, used by companies like LinkedIn, Uber, Netflix.

10. **Open Source and Extensible**: Kafka is an open-source Apache project with support for **Kafka Streams**, **Kafka Connect**, and integration with Spark, Flink, and more

## 1.1. Use Case (Zomato)

****

* **Kafka:** Has high throughput(amount of data that can pass through it), but has temporary storage.
* **DB:** Has low throughput but has permanent storage.

Data first pass through kafka as volume of data is large, and if such large data is directly handled in database, it can crash due to low throughput. So here data is passed through kafka first, later it is bulk inserted in database.

**Flow**

1. **Order Placement**
   * User places an order → OrderPlaced event is **published to Kafka**.
   * Topic: order-events
2. **Inventory & Kitchen Update**
   * Inventory Service **consumes** OrderPlaced to update stock.
   * Kitchen Dashboard Service **consumes** to start preparation.
3. **Delivery Assignment**
   * Kafka topic delivery-events used for **assigning delivery partner**.
   * When food is ready → OrderReady event is published → triggers driver assignment.
4. **Real-Time Tracking**
   * Delivery partner’s location is sent to Kafka topic location-updates every few seconds.
   * Map service **consumes** this to show real-time tracking to the user.
5. **Notification System**
   * OrderConfirmed, OrderOutForDelivery, OrderDelivered events are consumed by Notification Service.
   * Sends push/email/SMS updates via Kafka.
6. **Billing and Analytics**
   * All events (order, payment, delivery) are sent to Kafka topic analytics-events.
   * Analytics Service **aggregates data** for business insights.

## 1.2. Key Properties of Kafka

* **High Throughput**  
  Kafka can handle **millions of messages per second**, making it suitable for large-scale data pipelines.
* **Durability & Persistence**  
  Messages are stored on disk and **replicated** across brokers, ensuring data is not lost.
* **Scalability**  
  Kafka is horizontally scalable via **topics and partitions** across multiple brokers.
* **Fault Tolerance**  
  Kafka replicates data across multiple brokers, so even if a node fails, data remains available.
* **Decoupled Communication**  
  Producers and consumers are **independent**, enabling asynchronous and loosely coupled systems.
* **Replay ability**  
  Consumers can **re-read past messages** by resetting their offsets, useful for debugging or reprocessing.
* **Real-Time Processing**  
  Kafka supports **low-latency** stream processing with integration to tools like Kafka Streams, Flink, and Spark.
* **Exactly Once Semantics (EOS)**  
  Kafka supports **exactly-once delivery** for critical use cases (with proper configuration).
* **Message Retention**  
  Kafka retains messages for a **configurable duration** (e.g., hours, days, or indefinitely).
* **Multiple Subscribers**  
  A Kafka topic can have **multiple consumers/groups**, each processing messages independently.

## 1.3. Why Kafka

| **Problem** | **Kafka’s Solution** | **Example** |
| --- | --- | --- |
| Tight coupling | Event-based, decoupled | Order → Inventory + Email |
| Delays in processing | Real-time streaming | Fraud detection |
| Scalability limits | Horizontal scalability | Uber location data |
| Data loss on failure | Persistent, replayable events | Reprocess failed orders |
| System integration | Central event bus | CRM + Billing + Analytics |
| Event-driven needs | Pub/Sub with multiple consumers | E-commerce workflows |

**1. Decoupling Services (Loose Coupling)**

**Problem:** In a monolithic system, service A calls service B directly. If B is down, A fails. In microservices, this is risky.

**Kafka Solution:** Kafka acts as a **buffer**. Service A just publishes to a topic, and B consumes from it later.

**Example:**

* **Order Service** publishes "Order Placed" event.
* **Inventory Service** and **Email Service** both consume this event independently.
  + No tight coupling between services.
  + If Email service is down, the event remains in Kafka and is picked up when it comes back up.

**2. Real-Time Processing:**  
**Problem:** Traditional databases or batch jobs are slow and not real-time.

**Kafka Solution:** Kafka streams data in real-time, allowing you to process it instantly.

**Example:**

* Uber uses Kafka to process location data from millions of drivers in real time.
* Fraud detection systems consume transaction events instantly and trigger alerts.

**3. Scalability**  
**Problem:** High-traffic systems can’t scale linearly using traditional message queues or REST APIs.

**Kafka Solution:** Kafka is **horizontally scalable**, handling **millions of messages per second** using partitions and brokers.

**Example:**

* LinkedIn processes **trillions** of events per day using Kafka for activity tracking.

**4. Use in Event-Driven Architecture**  
Kafka enables **event-driven** systems where business logic is based on events instead of function calls.

**Example:**

* In an ecommerce site:
* User adds item → "ItemAddedToCart" event
* Inventory updates → "StockUpdated" event
* Checkout → "PaymentProcessed" → "OrderConfirmed"

**5. Data Integration Across Systems**  
**Problem:** Different systems (e.g., CRM, billing, analytics) need to share data.

**Kafka Solution:**Kafka acts as a **central data pipeline**, broadcasting to multiple systems.

**Example:**

* A "User Registered" event is consumed by:
  + **CRM system** to add user info
  + **Billing system** to create account
  + **Analytics system** to track signup

**6. Replayable Event Store**  
**Problem:** If a service goes down or misses data, how do we reprocess events?

**Kafka Solution:** Kafka **persists messages for a configurable time (or forever)**. Consumers can rewind and replay.

**Example:**

* Data pipeline in a bank can reprocess 3 days of failed transactions just by resetting the Kafka offset.

**7. Reliability and Fault Tolerance**  
**Problem:** What happens if a consumer fails or data is lost?

**Kafka Solution:** Kafka replicates data across multiple brokers. Consumers commit offsets, ensuring **no data loss**.

**Example:**

* Payment processor service restarts after a crash and resumes from last committed offset.

## 1.4. Can Kafka replace Databases

**No, Kafka is not a replacement for a database**, but in **some specific scenarios**, Kafka can **temporarily act like one** — with trade-offs. As Kafka stores data temporary only(default 7 days but configurable)

**❌ Kafka Is Not Designed For:**

1. **Querying data** (like SELECT queries in SQL)
2. **Data relationships** (joins, foreign keys, etc.)
3. **Long-term storage** (Kafka retains data for limited time)
4. **Transactional integrity** like ACID in RDBMS

**✅ When Kafka Can Act *Like* a Database**

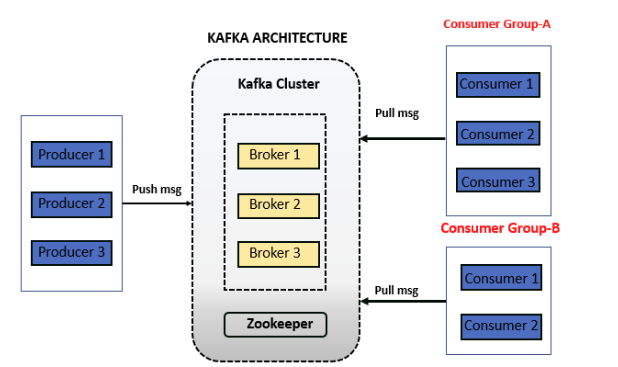
Kafka can **temporarily serve as a source of truth or event log** in some use cases:

**Examples:**

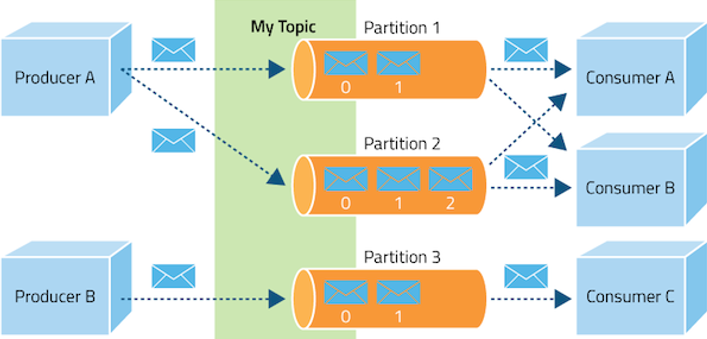
* **Event Sourcing** systems: State is rebuilt by replaying events from Kafka.
* **Log-based architectures**: Store change data (CDC) events from a database.

# **2. Kafka Working**

## 2.1. Architecture

****

1. **Producers** send messages to **Kafka topics**, which are split into **partitions** for scalability.
2. **Brokers** (Kafka servers) store these partitions and handle incoming/outgoing data.
3. **Consumers** read messages from partitions, tracking their progress using **offsets**.
4. **Consumer groups** allow multiple consumers to share the load for parallel processing.
5. Kafka ensures **high availability** through **replication** and supports **replayable, decoupled event streams**.



While Apache Kafka is often referred to and used as a message queue, it's more accurately described as a distributed streaming platform that incorporates aspects of both message queues and publish-subscribe systems.

**Message queue:** Traditional message queues typically follow a First-In, First-Out (FIFO) pattern, where messages are processed in the order they are received. Kafka can be configured to mimic this behaviour within a single partition, but it doesn't guarantee global FIFO order across multiple partitions

Kafka provides ordering guarantees within a single partition, but achieving global ordering across multiple partitions can be more complex.

**Publish-Subscribe:** In publish-subscribe model, messages are broadcast to multiple subscribers (consumers) interested in a specific topic. Kafka supports this through its topic-based architecture, allowing multiple consumer groups to subscribe to the same topic and receive messages.

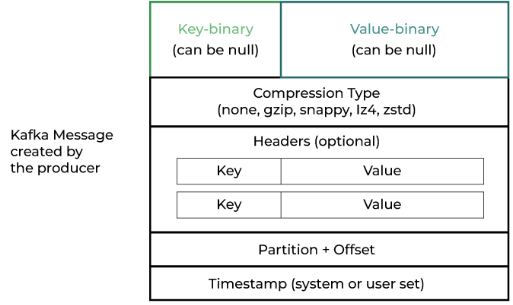
## 2.2. Key Components

**1. Producers**

* Clients that publish (send) data to Kafka topics.
* Can push data continuously (e.g., Order events, logs, metrics).

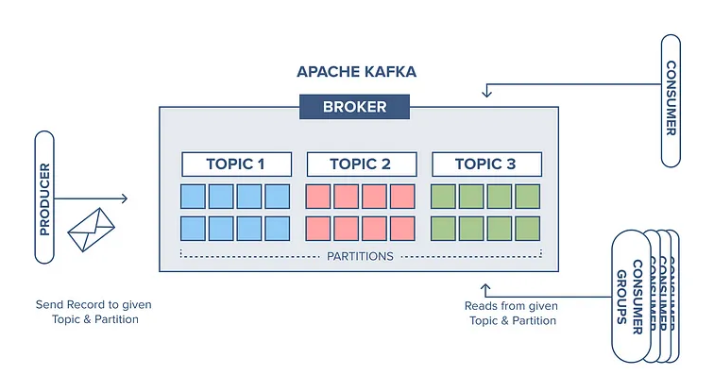
**2. Topics**

* Named channels where messages are published.
* Each topic is split into partitions for scalability.
* Logical grouping of message



**3. Partitions**

* Ordered, immutable logs of messages within a topic.
* Each message has an offset (like a unique ID).
* Enables parallel processing**.**
* Can partition data based on region, category etc.

****

**4. Offsets**

* Unique identifiers for each message **within a partition**.
* Consumers track offsets to know where they left off.

**5. Consumers**

* Clients that **subscribe and read** data from Kafka topics.
* Can belong to a **consumer group** for parallel consumption.

**6. Consumer Groups**

* A group of consumers that **share the work** of reading from a topic.
* Each partition is read by **only one consumer** within a group.

**7. Brokers**

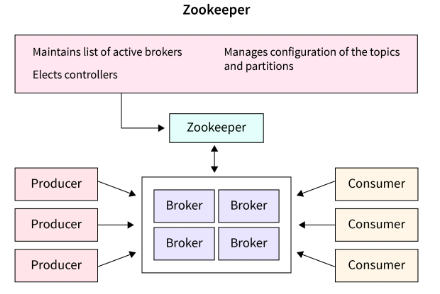
* Kafka servers that **store and serve** messages.
* Each broker handles partitions of multiple topics.
* A Kafka cluster = Multiple broker

**8. Replication**

* Kafka **replicates partitions** across multiple brokers.
* Ensures **high availability** and **fault tolerance**.

**9. Controller (via Zookeeper or KRaft)**

* **Zookeeper manages Kafka cluster metadata**, like broker info, topics, and partitions.
* It **elects the controller broker** and **monitors broker health** for fault tolerance.
* It also helps coordinate **leader election and rebalancing** across the cluster.



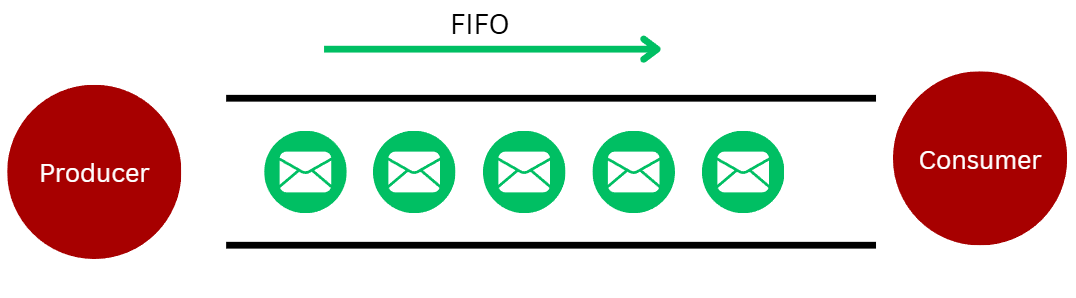
## 2.3. Key Working Principles

1. **Producer Chooses Partition**  
   A producer decides **which partition** of a topic to write to — either by:
   * Using a **key** (deterministic hash → partition), or
   * Kafka **randomly selects** one if no key is provided.
2. **Each Topic Has Multiple Partitions**  
   Partitions allow **parallelism and scalability**. More partitions = more parallel writes and reads.
3. **Messages Are Written with Offsets**  
   Every message in a partition gets a unique **offset**, used by consumers to track reading progress.
4. **Producer Doesn’t Know Consumers**  
   Producers just send data to Kafka topics — they are **completely decoupled** from consumers.
5. **Consumer Groups Control Read Logic**  
   A **consumer group** shares the workload:
   * Each partition is read by **only one consumer** in the group
   * But a consumer can read **from multiple partitions**
6. **One Partition = One Consumer (per group)**  
   Within a group, **a single partition is never read by multiple consumers** at the same time.
7. **One Partition Can Have Many Consumers (in different groups)**  
   Multiple consumer groups can read the **same partition independently**, enabling **pub/sub behavior**.
8. **Brokers Store Data, Not Route It**  
   Kafka brokers **store partitions** and respond to pull requests from consumers; they don’t push data.
9. **Kafka Stores Data Regardless of Consumption**  
   Kafka retains messages for a set time (e.g., 7 days), whether they are consumed or not.
10. **Consumers Track Their Own Offsets**  
    Consumers are responsible for **managing and committing offsets** (automatically or manually), to resume reading where they left off.

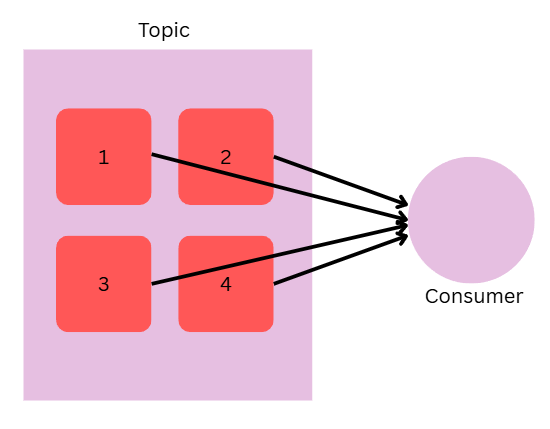
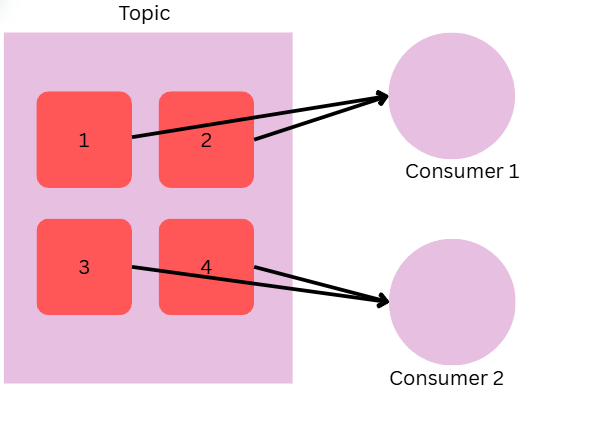
## 2.4. Auto Balancing

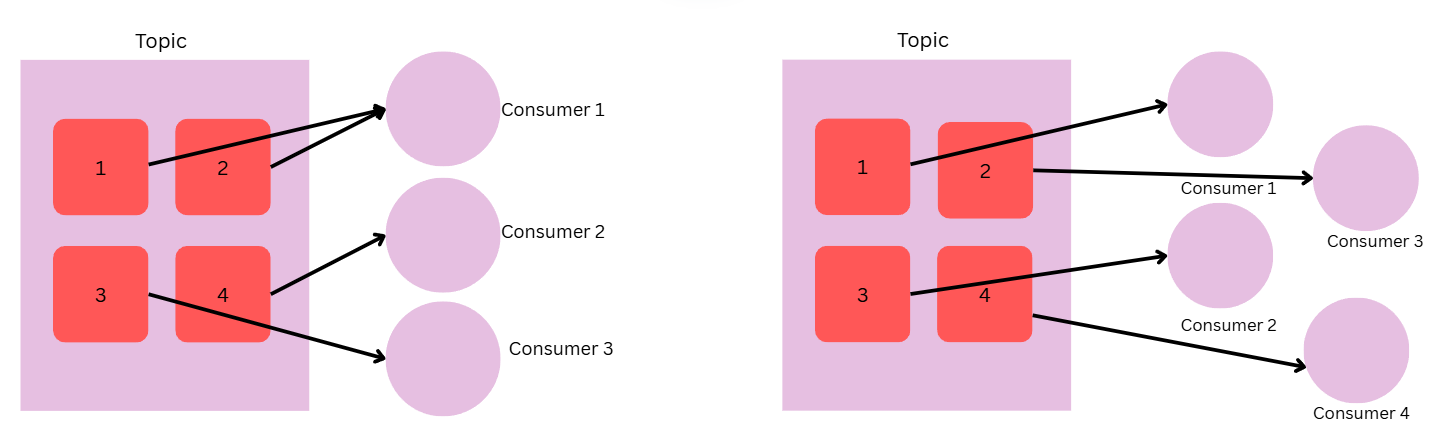
**Auto-balancing in Kafka** refers to the automatic **reassignment of partitions** among consumers in a **consumer group** when changes occur, like:

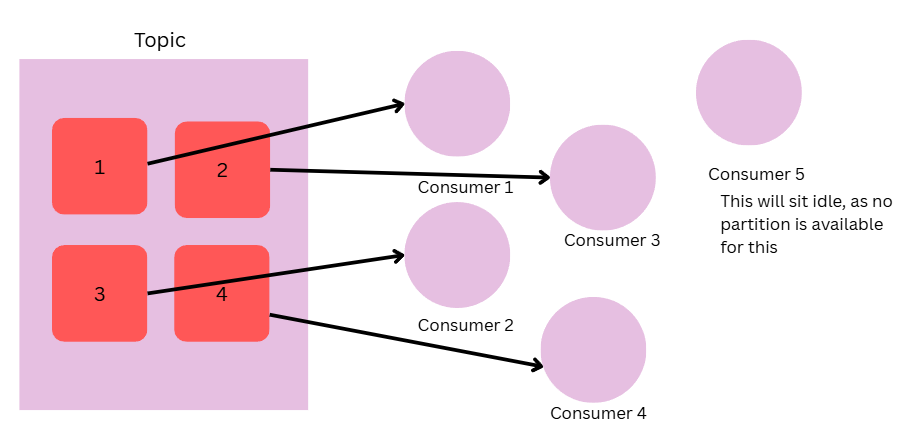
* A new consumer **joins**
* A consumer **leaves or crashes**
* A consumer **restarts**
* 1 consumer: - Can consume multiple partition
* 1 partition: - Can be consumed by only one consumer (in single consumer group), this helps to act it like queue, helps it to act as queue



**Cases of Auto balancing**





This is all happening for consumers in same consumer group, if we have consumers in different consumer group, they can share same partition.

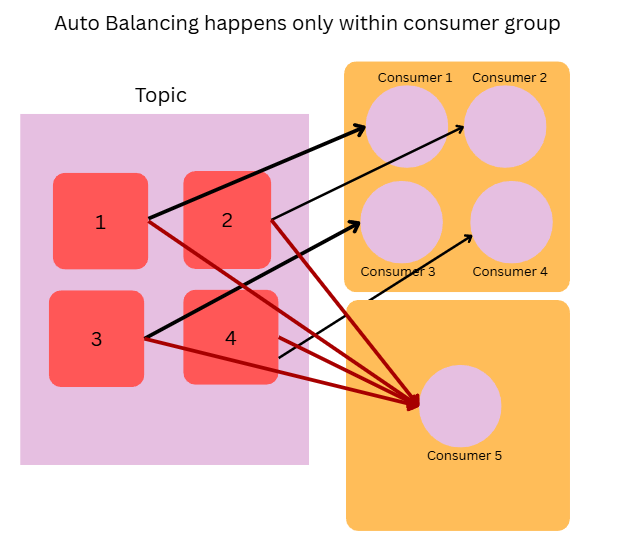
**How it works**

1. **Initial Assignment**  
   When a consumer group starts, Kafka **evenly assigns partitions** of a topic across available consumers.
2. **Dynamic Rebalancing**  
   If a consumer joins or leaves the group:
   * Kafka **triggers a rebalance**
   * Partitions are **re-distributed automatically** among the remaining consumers
3. **Coordinated by Group Coordinator**
   * One broker acts as the **Group Coordinator**
   * It manages the rebalance process using a **leader consumer** in the group

**Impact of Rebalancing**

* During rebalance, consumers **temporarily stop consuming**
* This causes a **brief pause** in data processing
* Too frequent rebalancing can lead to instability

**Behaviour with multiple consumer groups**



# **3. Kafka with Spring**

**1. Prerequisites:**

* Java 17+
* Spring Boot 3.x
* Apache Kafka running locally (or use Docker)

**2. Install Kafka(Docker way):**

docker run -d --name zookeeper -p 2181:2181 zookeeper

docker run -d --name kafka -p 9092:9092 --env KAFKA\_ZOOKEEPER\_CONNECT=host.docker.internal:2181 --env KAFKA\_ADVERTISED\_LISTENERS=PLAINTEXT://localhost:9092 --env KAFKA\_LISTENERS=PLAINTEXT://0.0.0.0:9092 kafka

**3. Add Dependencies (Maven)**

<dependency>  
 <groupId>org.springframework.kafka</groupId>  
 <artifactId>spring-kafka</artifactId>  
</dependency>

**4. application.yml**

spring:

  kafka:

    bootstrap-servers: localhost:9092

    consumer:

      group-id: my-group

      auto-offset-reset: earliest

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

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

    producer:

      key-serializer: org.apache.kafka.common.serialization.StringSerializer

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

**5. Kafka Producer**

@Service  
public class KafkaProducer {  
 private final KafkaTemplate<String, String> kafkaTemplate;  
  
 public KafkaProducer(KafkaTemplate<String, String> kafkaTemplate) {  
 this.kafkaTemplate = kafkaTemplate;  
 }  
  
 public void sendMessage(String message) {  
 kafkaTemplate.send("test-topic", message);  
 }  
}

**6. Kafka Consumer**

@Service  
public class KafkaConsumer {  
 @KafkaListener(topics = "test-topic", groupId = "my-group")  
 public void listen(String message) {  
 System.*out*.println("Received: " + message);  
 }  
}

**7. REST Endpoint to Send Message**

@RestController  
@RequestMapping("/kafka")  
public class KafkaController {  
 private final KafkaProducer producer;  
  
 public KafkaController(KafkaProducer producer) {  
 this.producer = producer;  
 }  
  
 @PostMapping("/publish")  
 public ResponseEntity<String> send(@RequestParam String message) {  
 producer.sendMessage(message);  
 return ResponseEntity.ok("Sent: " + message);  
 }  
}

**8. Test it**

* Run Spring Boot app
* Hit: POST http://localhost:8080/kafka/publish?message=hello-kafka
* See output in console from the consumer.