Apache Kafka: A Beginner's Guide

Welcome, FIL Freshers!

What is Kafka? (The Big Picture)

📮 Imagine a Super-Efficient Postal Service

Instead of sending mail directly from person to person (which can be slow and unreliable), you send all mail to a central, organised post office.

> **This post office sorts mail into different mailboxes (Topics).**
> Anyone interested in certain types of mail can subscribe to a specific mailbox and get the mail whenever they want.

In Tech Terms:

- Kafka is a **distributed event streaming platform**
- Used for building **real-time data pipelines** and streaming applications
- **Key Idea:** Decouple systems that produce data from systems that consume it

Why use it?
To handle massive amounts of data in real-time, reliably, and at high speed!

Kafka Architecture - Core Components

Let's meet the key players in our "Postal Service":

Producer

The sender. An application that writes data to Kafka.

Like the person dropping a letter into a mailbox

Consumer

The receiver. An application that reads data from Kafka.

Like someone with a key to a specific PO Box

Topic

A category or feed name to which messages are published.

Like a specific PO Box (e.g., "user_signups")

Broker

A Kafka server that stores the data.

Like the physical post office building

Cluster: A group of brokers working together for fault tolerance and scalability

A Deeper Look into Kafka's Architecture

Core Design Capabilities

High Throughput
Moving lots of data quickly

Scalability
Handling more data by adding machines

Fault Tolerance
Surviving machine failures

1. Storage Layer

Partition as a Log
Sequential, immutable write-ahead log enables high-speed disk I/O

Segments
Partitions split into manageable files for efficient retention management

Zero-Copy
Direct data transfer from disk to network without memory overhead

2. Replication

Leader & Followers
Leader handles writes while followers replicate data continuously

In-Sync Replicas (ISR)
Only fully caught-up followers qualify for leader election

Automatic Failover
Immediate leader election from ISR pool ensures minimal disruption

A Deeper Look into Kafka's Architecture

3. Producers

Smart Partitioning

Key-based hashing ensures ordered delivery; round-robin for load balancing

Acknowledgements

acks=0 (fast), acks=1 (balanced), acks=all (durable) - choose your guarantee

4. Consumer Groups

Parallel Processing

One partition per consumer within a group enables true scalability

Offset Management

Committed offsets in __consumer_offsets ensure at-least-once delivery

Deep Dive - Topics, Partitions & Offsets

How Kafka Organizes Data for Speed and Scale

A Topic is split into Partitions:

- **Partitions:** Each is an ordered, immutable sequence of messages (like a single log file)
- **Why split?** For parallelism! Multiple consumers can read from different partitions simultaneously

> **Offset:** Each message within a partition has a unique ID called an offset (0, 1, 2, 3...)

Why offsets matter: Consumers track which offset they've read.
If a consumer crashes and restarts, it knows exactly where to pick up from—ensuring no data is lost!

Understanding the Kafka Cluster

💪 Strength in Numbers

A Kafka cluster is a group of one or more brokers providing:

- **Scalability:** Distribute topics and partitions across many servers
- **Fault Tolerance:** Kafka replicates each partition across multiple brokers
- **High Availability:** If the leader broker fails, a follower is automatically elected as the new leader

> **Replication Factor:** Number of copies of a partition. Common is 3 (1 leader, 2 followers)
>
> - The **leader** handles all reads and writes
> - The **followers** copy the leader's data

Zookeeper's Role: Manages the cluster, tracks which brokers are alive, and stores configuration.

Practical Hands-On - Let's Build!

Goal: Set up a single-node Kafka cluster on your local machine using Docker

Prerequisites:

Docker Desktop is installed and running on your machine

Step 1: Create 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
```

docker-compose.yml

Create a new file named `docker-compose.yml` and paste the following content into it. This file defines the Zookeeper and Kafka services.

Hands-On - Starting the Cluster

Step 2: Start the Cluster

Open a terminal in the same directory as your file and run:

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

This will download the images and start Kafka and Zookeeper containers in the background. Your local Kafka cluster is now running!

Step 3: Access Kafka Container

```
docker exec -it kafka /bin/bash
```

You are now inside the Kafka container's command line.

Hands-On - Producer & Consumer

Create a Topic

Let's create a topic named `fil-freshers` with one partition and one replica.

```
kafka-topics --create \
--topic fil-freshers \
--bootstrap-server localhost:9092 \
--replication-factor 1 \
 --partitions 1
```

Start a Producer

Open a **new** local terminal and run this command to get a shell inside the Kafka container again. We will use this one for the producer.

```
docker exec -it kafka /bin/bash
kafka-console-producer \
--topic fil-freshers \
--bootstrap-server localhost:9092
```

Hands-On - Producer & Consumer

Type some messages and press Enter after each one:

>Hello Kafka!
>This is my first message.
>FIL Training Rocks!

Start a Consumer

Go back to your **first** terminal (the one from Step 1). Start the consumer to read messages from the beginning of the topic.

```
kafka-console-consumer \
--topic fil-freshers \
--bootstrap-server localhost:9092 \
--from-beginning
```

You will instantly see the messages you typed in the producer window appear here! You have successfully sent and received messages with Kafka.

Hands-On - Producer & Consumer

To shut down your cluster when you're done:

docker-compose down

Key Takeaways & Q&A

What We Learned Today:

- **What Kafka Is:** A distributed streaming platform acting as a central "post office" to decouple senders and receivers
- **Core Components:** Producers (send), Consumers (receive), Brokers (store), and Topics (categorize)
- **Architecture:** Topics split into Partitions for parallelism. Messages have Offsets for tracking
- **Clusters:** Groups of brokers provide fault tolerance and scalability through replication
- **Hands-On:** You can quickly set up a local Kafka cluster with Docker and use command-line tools

1) What we are building

A minimal, **in-memory** message broker (like a tiny Kafka). Three apps:

- **Broker (FastAPI server)** – stores messages per topic and tracks each consumer's offset.
- **Producer (Python script)** – registers a topic, then publishes messages via HTTP.
- **Consumer (Python script)** – registers to a topic, then polls the broker for the next message.

Key constraints:

- Data exists only in RAM (lost if the broker restarts).
- One message list per topic (no partitions or replication).
- Each consumer has its own offset.