Event Driven Services using Cloud Stream

Apache Kafka is a distributed event streaming platform designed for high-performance data pipelines, streaming analytics, and integration. Originally developed at LinkedIn, Kafka is an open-source system that handles large-scale message streams in real-time. Kafka is widely used for building real-time data pipelines and streaming applications, where it acts as a central hub for event streaming between services.

Key Concepts

- **Producer**: A client that sends data to Kafka topics.
- **Consumer**: A client that reads data from Kafka topics.
- **Broker**: Kafka servers that store and distribute messages.
- **Topic**: A category or feed to which records are sent. Topics are partitioned for parallelism.
- **Partition**: Topics are split into partitions, allowing data to be parallelized across different servers.
- **Consumer Group**: A group of consumers that coordinate to read records in parallel from partitions.

Example Scenario

Consider a real-time data pipeline in an e-commerce application that tracks orders. Here's how Kafka can be used in this scenario:

1. **Order Service** (Producer): When a user places an order, the service produces messages to a Kafka topic called `orders`.
2. **Inventory Service** (Consumer): Consumes the `orders` topic to update inventory levels.
3. **Shipping Service** (Consumer): Consumes the same `orders` topic to prepare the shipping process.

4. **Notification Service** (Consumer): Sends confirmation notifications to users by consuming the `orders` topic.

Each service can work independently, reading from the same source of truth (the `orders` topic), ensuring reliable and scalable message distribution.

Kafka Architecture

Kafka's architecture revolves around a distributed cluster of servers working together to handle message streams.

1. **Cluster of Brokers**: Kafka runs as a cluster on one or more servers, known as brokers. Each broker stores messages in topic partitions and serves client requests for data.
2. **Topic and Partitions**:
   - Each topic is divided into partitions to enable scalability.
   - Partitions allow data to be distributed across multiple brokers and facilitate parallel processing.
3. **Producers**:
   - Producers send messages to topics, and Kafka distributes these messages across partitions.
   - Each message within a partition has an offset, acting as a unique identifier for that message in the partition.
4. **Consumers and Consumer Groups**:
   - Consumers subscribe to topics and read data. They can be organized into consumer groups, where each consumer in a group is assigned to a specific partition.
   - Multiple consumer groups can read from the same topic independently, allowing different applications to process the data in unique ways.
5. **Zookeeper (Legacy)**:
   - Originally, Kafka used Zookeeper for maintaining cluster state, tracking broker metadata, and managing service coordination. However, in more recent versions, Kafka has been transitioning to remove this dependency.

Kafka Workflow Example in Architecture

Let’s break down how data flows in a Kafka setup with an e-commerce order pipeline.

1. **Producing Orders**:
    - When a user places an order, the Order Service sends a message (order details) to the `orders` topic. Kafka distributes this message across partitions within the topic for load balancing.
2. **Partition Assignment**:
    - Suppose the `orders` topic has three partitions. Kafka might distribute each incoming order message to one of these partitions (either round-robin or based on a key like the order ID).
3. **Consuming Services**:
    - Each downstream service (Inventory, Shipping, Notification) belongs to a separate consumer group and reads the `orders` topic. Kafka ensures that each partition is read by only one consumer in each group, enabling parallel processing without overlap within a group.
4. **Offset Management**:
    - As each service consumes messages, Kafka tracks which messages have been read (by offset). Consumers can then resume from their last read position, ensuring they don’t miss or reprocess data if they temporarily disconnect.

Kafka’s Role in Microservices

Kafka excels in microservices for decoupling services. Each service can act independently by producing or consuming data to/from Kafka, ensuring loose coupling and scalability.

Kafka enables **fault tolerance** by replicating data across multiple brokers. If one broker fails, another can take over its partitions, ensuring no data loss or service interruption. Additionally, Kafka’s high throughput makes it suitable for systems needing to handle millions of events per second, with low latency.

Download and install:

Go to apache kafka website downlaod and run

Define new properites for kafka in properties file:

```
spring.cloud.kafa.binder.brokers:localhost:9092
```

Traditional Approch: EDA