Microservices architecture is a software development approach that breaks down an application into smaller, independent services that communicate with each other through APIs.

Benefits

- **Continuous delivery and deployment :** new features and updates can be released more frequently and with less risk.
- **Better scalability :** can be scaled up or down as needed without affecting other services
- **Improved fault isolation :** Larger applications can remain mostly unaffected by the failure of a single module, as each service is independent.
- **Greater flexibility and Smaller development teams :** smaller teams can work with their preferred programming language for faster development.
- **Higher software testability :** Can be tested independently, which makes it easier to identify and fix bugs.
- **Improved maintainability :** easier to make changes to the application without affecting other services.

Drawbacks

However, microservices architecture also has some drawbacks.

- It can be more complex to develop and maintain than traditional monolithic architectures, as it requires more coordination between services.
- If database per service is used, it can be more difficult to ensure data consistency across services.

Communication between services

- **Synchronous Communication :** Services communicate with each other using direct requests and require immediate feedback.
- **Asynchronous Communication :** This is great when you don't need immediate results to complete an operation. The client does not wait for a response and just sends the request to a message broker.

Microservices Architecture Patterns

- API Composition
- Circuit Breaker
- Command query responsibility segregation (CQRS)
- Event sourcing
- Saga

Microservices Architecture Patterns

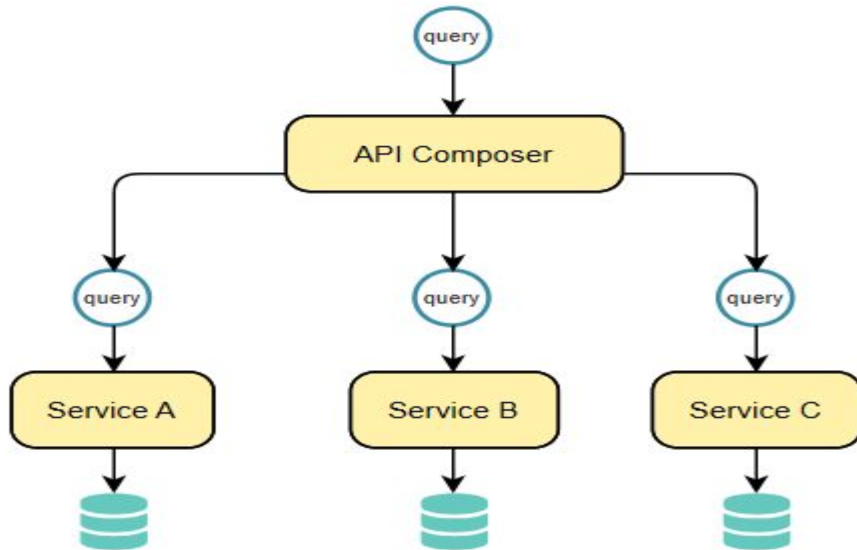
API composition

Pattern uses an API composer, or aggregator, to implement a query by invoking individual microservices that own the data. It then combines the results by performing an in-memory join.

But not suitable

- for complex queries and large datasets that require in-memory joins.
- number of microservices connected increase overall system becomes less available

API composition pattern



Circuit Breaker

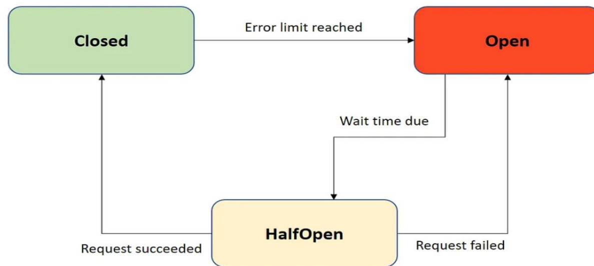
Circuit Breaker pattern is used to handle faults and failures. It prevents system overload by automatically turning off service interactions when the system is identified as under stress or a service is unresponsive.

This pattern is crucial for developing resilient microservices architectures, as it limits the impact of service failures and latencies.

Circuit Breaker

It has three states:

- **Closed** : services are allowed to communicate
- **Open : will completely block the communication between services**
- **Half Open**: will allow limited number of requests to reach the services if requests are successful the breaker will be closed



CQRS

Command query responsibility segregation

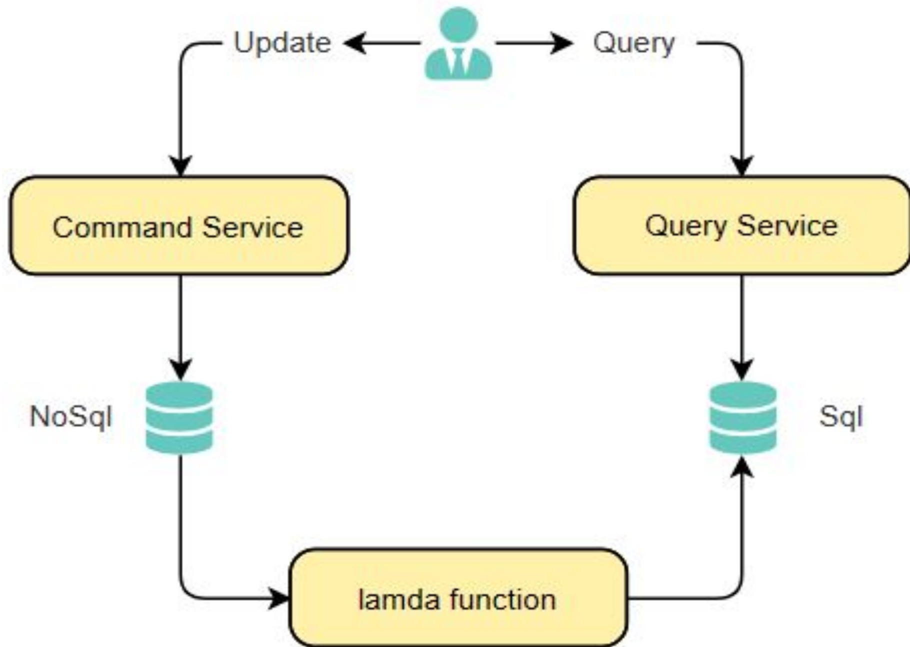
The CQRS pattern splits the application into two parts:

- **command side** handles create, update, and delete requests
- **query side** runs the query part by using the read replicas.

Used when

- implemented the database-per-service pattern and want to join data from multiple microservices
- Your read and write workloads have separate requirements for scaling, latency, and consistency.

CQRS pattern



Event sourcing

The event sourcing pattern is typically used with the CQRS pattern. Data is stored as a series of events, instead of direct updates to data stores. Events are used to completely rebuild the application's state.

Uses

- Event Replay
- Event Reversal
- Complete Rebuild

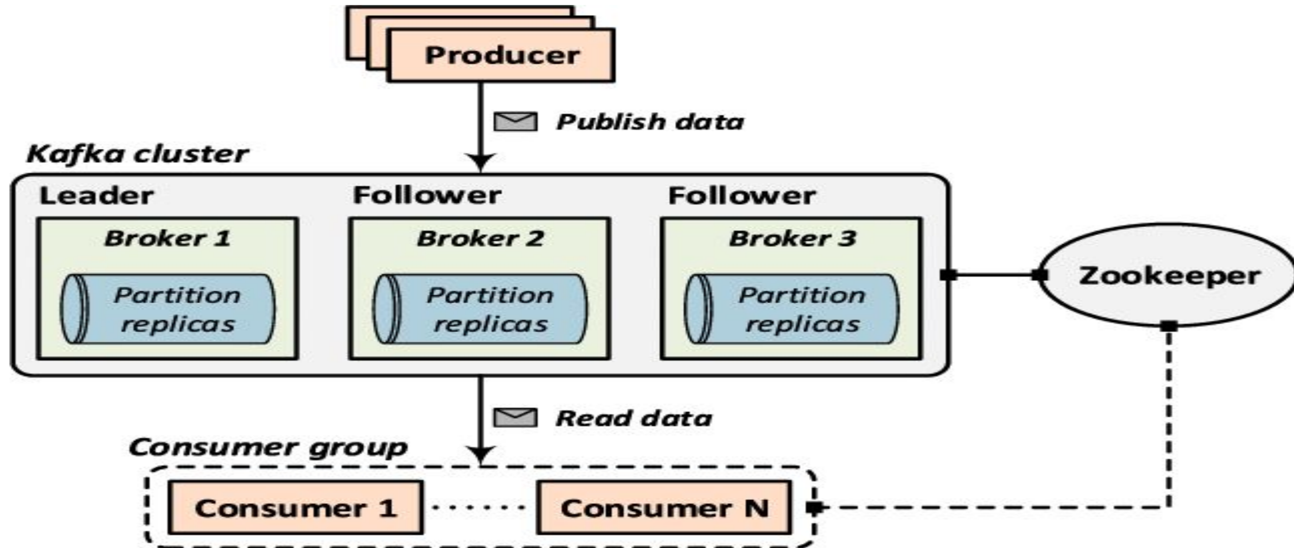
Saga

Saga is a failure management pattern that helps establish consistency in distributed applications, and coordinates transactions between multiple microservices to maintain data consistency.

The Saga Pattern provides two key methods to handle these scenarios

- **Orchestration-based** : centralizing the decision-making process and dictating the sequence of steps and is aware of the overall business process and knows what local transaction should be executed next based on the result of the previous transactions.
- **Choreography-Based Saga :** There is no central orchestrator. Each local transaction publishes domain events that trigger local transactions in other services.

Introduction to kafka



Introduction to kafka

Apache kafka is an open source distributed streaming platform that allows for the development of real-time event-driven applications.

- Kafka allows for a persistent ordered data structure.
    - The order and integrity of the records are maintained
    - A record cannot be deleted, or modified, only appended to the log.
- Every record has an assigned unique sequential ID(**offset**) used to retrieve data

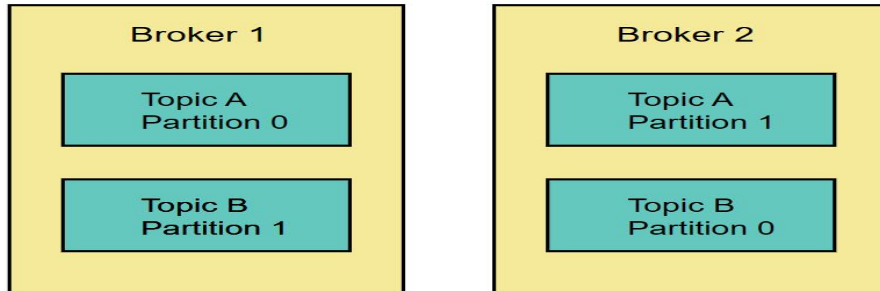
Introduction to kafka

- Brokers
- Topics
- APIs
    - Producer
    - Consumer
    - Connect
    - Stream
    - Admin

Kafka Brokers

A Kafka broker is a physical instance, virtual machine, or container that runs the Kafka process. A typical Kafka deployment contains more than one broker arranged as a cluster to ensure high availability and robustness.

- After connecting to any broker, you will be connected to the entire cluster
- Each broker contains certain topic partitions



Kafka Topics

Topics are a storage mechanism for a sequence of events. Essentially, topics are durable log files that keep events in the same order as they occur in time.

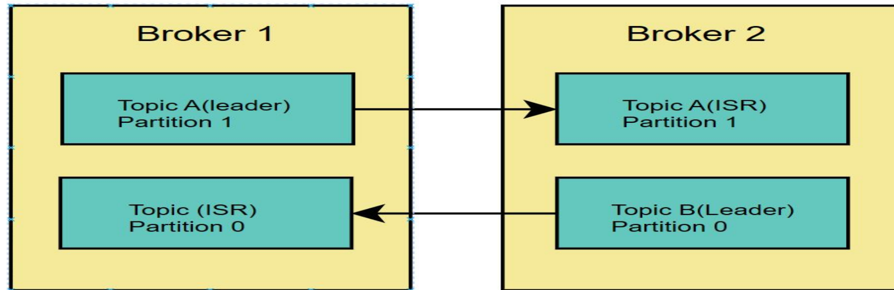
Each topic are splitted in to partitions

- Partition is ordered.
- Message within a partition gets an incremental id called **offset.**
- The order of the events within the same topic partition guaranteed. but, not across all partitions.

Each Topic should have a replication factor > 1. This way if broker is down, another broker can server the data

Kafka Topics

At any time only 1 broker can be a **leader** for a given partition and only that leader can receive and serve data for a topic-partition and the other brokers will sync the data to the rest of the partition which are called in-sync-replica(**ISR**)



Kafka Producers

Producer is a client application that publishes (writes) events to a Kafka cluster to specific topic. If the key is provided sending events will sent to same partition.

Producer can choose to receive acknowledgment for data writes

- Acks = 0 > no acknowledgment
- Acks = 1 > acknowledgment from leader only
- Acks = all > acknowledgment from leader and replica(ISR)

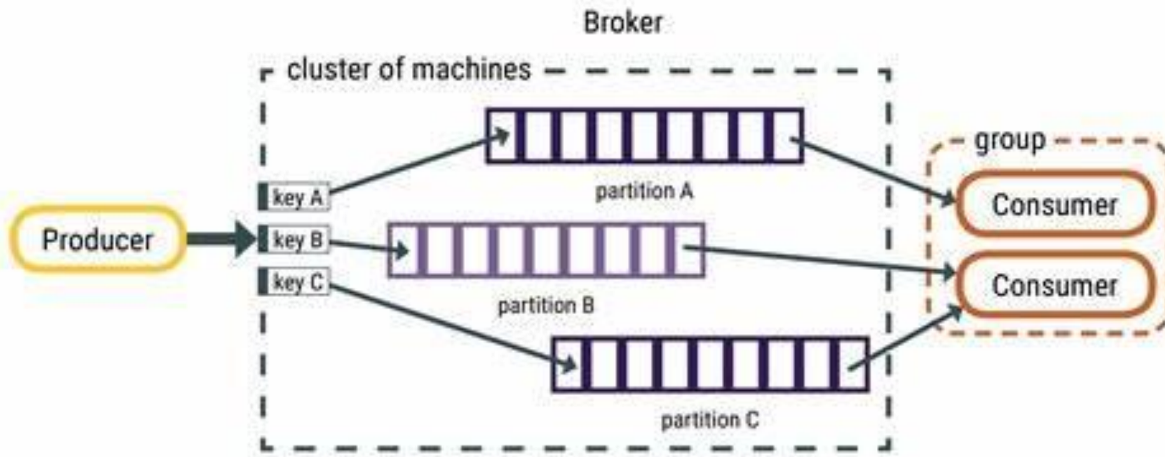
Kafka Consumer

A Kafka Consumer

- client application that subscribes to one or more topics in a Kafka cluster and consumes the records (messages) from these topics.
- Consumer only read the data from the topic data is available until its retention period.

If you decided to scale your application to multiple instance you can use consumer group to handle it.

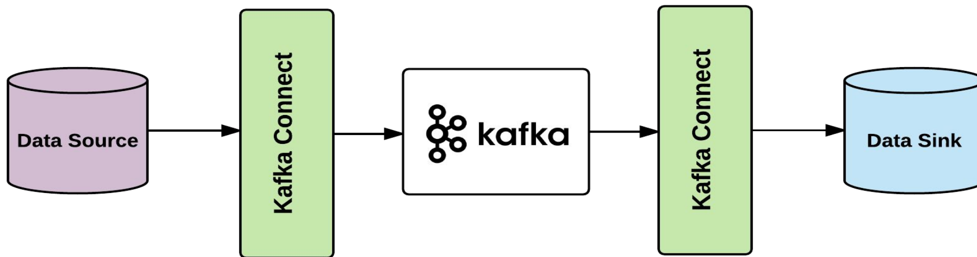
Producer and Consumer



Connect

Kafka Connect is used to perform streaming integration between Kafka and other systems such as databases, cloud services,

- **Sink Connector** push from Kafka into some sink data system.
- **Source Connector** some source data system into Kafka



Admin and Stream

Admin API supports managing and inspecting topics, brokers, acls, and other Kafka objects.

Kafka Stream API provides a higher-level abstraction for stream processing to perform complex operations

- windowing, joining, aggregating, and stateful processing.
- It supports data parallelism, fault tolerance, scalability.