Event Driven Microservice **Kafka, RabbitMQ**

short line

# Why event-based design

Temporal coupling - When a caller service depends on response from another service **it is called temporal coupling**. So, the problem happens due to synchronous communication. So, we need to make it asynchronous by making it event driven.

There are 2 types of events-driven model

1. **Pub /Sub model**-
   1. This model evolves around subscriptions.
   2. Producers will generate the events and that are distributed to all the interested subscribers for consumption.
   3. Inside this model, once an event is received and consumed by the consumers, it cannot be replayed again.
   4. Any new subscribers joining later will not have access to the past events.
   5. E.g. Rabbit MQ
2. **Event streaming model-**
   1. Inside event streaming model events will be written into a log in a sequential manner.
   2. Producers publish the events as they occur, and these events are stored in a well-ordered fashion.
   3. Consumers, instead of subscribing to the events, will have the ability to read from any part of the event stream,
   4. which means the events can be replayed, allowing the clients to join at any time and receive all the past events as well.
   5. E.g. Apache Kafka

# Rabbit MQ

## Example Flow

**🔹 Normal Flow (Happy Path)**

1. Client/Actor → Order Service API
2. Order Service → Generate Order (UUID) + Persisit to DB.
3. Order Sevice 🡪 order.exchange → order-service-queue 🡪Publish **order.created**.
   * Exchange: **order.exchange**
   * Routing Key: **order.created**
4. Order Sevice → Payment Exchange → Payment Service consumes.
   * Queue: payment-service-queue
   * Binding: order.exchange → payment-service-queue (order.created)
5. Payment Service → Payment DB/Outbox → Persist Payment (status=INITIATED).
6. Payment Outbox → Payment Exchange → Publish payment.created.
   * Exchange: payment.exchange
   * Routing Key: payment.created
7. Payment Service → Payment DB/Outbox → Update Payment (status=COMPLETED).
8. Payment Outbox → Payment Exchange → Publish payment.completed.
   * Exchange: payment.exchange
   * Routing Key: payment.completed
9. Payment Exchange → Delivery Service → Delivery Service consumes.
   * Queue: delivery-service-queue
   * Binding: payment.exchange → delivery-service-queue (payment.completed)
10. Delivery Service → Delivery DB/Outbox → Persist Delivery (status=CREATED).
11. Delivery Outbox → Delivery Exchange → Publish delivery.created.

* Exchange: delivery.exchange
* Routing Key: delivery.created

1. Delivery Service → Delivery DB/Outbox → Update Delivery (status=COMPLETED).
2. Delivery Outbox → Delivery Exchange → Publish delivery.completed.

* Exchange: delivery.exchange
* Routing Key: delivery.completed

1. Delivery Exchange → Order Service → Order Service consumes.

* Queue: order-service-queue
* Binding: delivery.exchange → order-service-queue (delivery.completed)

1. Order Service → Order DB/Outbox → Update Order (status=CLOSED).
2. Order Outbox → Order Exchange → Publish order.closed.

* Exchange: order.exchange
* Routing Key: order.closed

**❌ Payment Failure Flow (Compensation)**

1. **Payment Service → Payment DB/Outbox** → Update Payment (status=FAILED).
2. **Payment Outbox → Payment Exchange** → Publish **payment.failed**.

* **Exchange**: payment.exchange
* **Routing Key**: payment.failed

1. **Payment Exchange → Order Service** → Order Service consumes.

* **Queue**: order-service-queue
* **Binding**: payment.exchange → order-service-queue (payment.failed)

1. **Order Service → Order DB/Outbox** → Update Order (status=CANCELLED).
2. **Order Outbox → Order Exchange** → Publish **order.cancelled**.

* **Exchange**: order.exchange
* **Routing Key**: order.cancelled

**⚠️ Delivery Failure Flow (Retry + Final Failure)**

1. **Delivery Service → Delivery DB/Outbox** → Update Delivery (status=FAILED).
2. **Delivery Outbox → Delivery Exchange** → Publish **delivery.failed**.

* **Exchange**: delivery.exchange
* **Routing Key**: delivery.failed

1. **Delivery Exchange → Delivery Retry Queue** → Message held with TTL.

* **Queue**: delivery-retry-queue
* **Binding**: delivery.exchange → delivery-retry-queue (delivery.failed)

1. **Retry Queue → Delivery Service** → After TTL, redelivers to main queue.

* **Queue**: delivery-service-queue
* **Re-routing Key**: delivery.retry

1. **Delivery Service → Delivery DB/Outbox** → Update Delivery (FAILED again).
2. **Delivery Outbox → Delivery Exchange** → Publish **delivery.failed (final)**.

* **Exchange**: delivery.exchange
* **Routing Key**: delivery.failed

1. **Delivery Exchange → Delivery DLQ** → Final failed message routed.

* **Queue**: delivery-service-dlq
* **Binding**: via delivery.dlx

1. **Delivery Exchange → Order Service** → Order Service consumes.

* **Queue**: order-service-queue
* **Binding**: delivery.exchange → order-service-queue (delivery.failed)

1. **Order Service → Order DB/Outbox** → Update Order (status=FAILED).
2. **Order Outbox → Order Exchange** → Publish **order.failed**.

* **Exchange**: order.exchange
* **Routing Key**: order.failed

**🔹 End-to-End Flow Mapping (Normal + Failure + Retry)**

| **Step** | **Action** | **Exchange** | **Queue** | **Routing Key** | **Purpose** |
| --- | --- | --- | --- | --- | --- |
| 1 | Client/Actor → Order Service (Place Order) | — | — | — | generate UUID , Persist to DB |
| 2 | Order Service → Publish order.created | order.exchange | order-service-queue | order.created | Generatet order event |
| 4 | Payment Service consumes order.created | order.exchange | payment-service-queue | order.created | Start payment process |
| 5 | Payment Service saves Payment (INITIATED) | — | — | — | Persist payment |
| 6 | Outbox → Publish payment.created | payment.exchange | — | payment.created | Notify that payment is initiated |
| 7 | Payment Service updates Payment (COMPLETED) | — | — | — | Persist success |
| 8 | Outbox → Publish payment.completed | payment.exchange | — | payment.completed | Notify that payment succeeded |
| 9 | Delivery Service consumes payment.completed | payment.exchange | delivery-service-queue | payment.completed | Start delivery process |
| 10 | Delivery Service saves Delivery (CREATED) | — | — | — | Persist delivery start |
| 11 | Outbox → Publish delivery.created | delivery.exchange | — | delivery.created | Notify delivery is created |
| 12 | Delivery Service updates Delivery (COMPLETED) | — | — | — | Persist completion |
| 13 | Outbox → Publish delivery.completed | delivery.exchange | — | delivery.completed | Notify delivery is done |
| 14 | Order Service consumes delivery.completed | delivery.exchange | order-service-queue | delivery.completed | Trigger order close |
| 15 | Order Service updates Order (CLOSED) | — | — | — | Persist closure |
| 16 | Outbox → Publish order.closed | order.exchange | — | order.closed | Notify that order is closed |

**❌ Payment Failure (Compensation)**

| **Step** | **Action** | **Exchange** | **Queue** | **Routing Key** | **Purpose** |
| --- | --- | --- | --- | --- | --- |
| 17 | Payment Service updates Payment (FAILED) | — | — | — | Persist failure |
| 18 | Outbox → Publish payment.failed | payment.exchange | — | payment.failed | Notify that payment failed |
| 19 | Order Service consumes payment.failed | payment.exchange | order-service-queue | payment.failed | Cancel order |
| 20 | Order Service updates Order (CANCELLED) | — | — | — | Persist cancel |
| 21 | Outbox → Publish order.cancelled | order.exchange | — | order.cancelled | Notify that order is cancelled |

**⚠️ Delivery Failure (Retry + Final Failure)**

| **Step** | **Action** | **Exchange** | **Queue** | **Routing Key** | **Purpose** |
| --- | --- | --- | --- | --- | --- |
| 22 | Delivery Service updates Delivery (FAILED) | — | — | — | Persist failure |
| 23 | Outbox → Publish delivery.failed | delivery.exchange | — | delivery.failed | Notify failure |
| 24 | delivery.failed → Retry Queue (TTL) | delivery.exchange | delivery-retry-queue | delivery.failed | Retry after delay |
| 25 | Retry Queue → Delivery Service | — | delivery-service-queue | delivery.retry | Re-deliver for retry |
| 26 | Delivery Service updates Delivery (FAILED again) | — | — | — | Persist retry failure |
| 27 | Outbox → Publish delivery.failed (final) | delivery.exchange | — | delivery.failed | Notify final failure |
| 28 | Delivery Exchange → Delivery DLQ | delivery.dlx | delivery-service-dlq | — | Store failed messages |
| 29 | Order Service consumes delivery.failed (final) | delivery.exchange | order-service-queue | delivery.failed | Mark order failed |
| 30 | Order Service updates Order (FAILED) | — | — | — | Persist order failure |
| 31 | Outbox → Publish order.failed | order.exchange | — | order.failed | Notify that order failed |

**Sequence diagram**

A black lines on a white background

AI-generated content may be incorrect.

**RabbitMQ topology diagram**

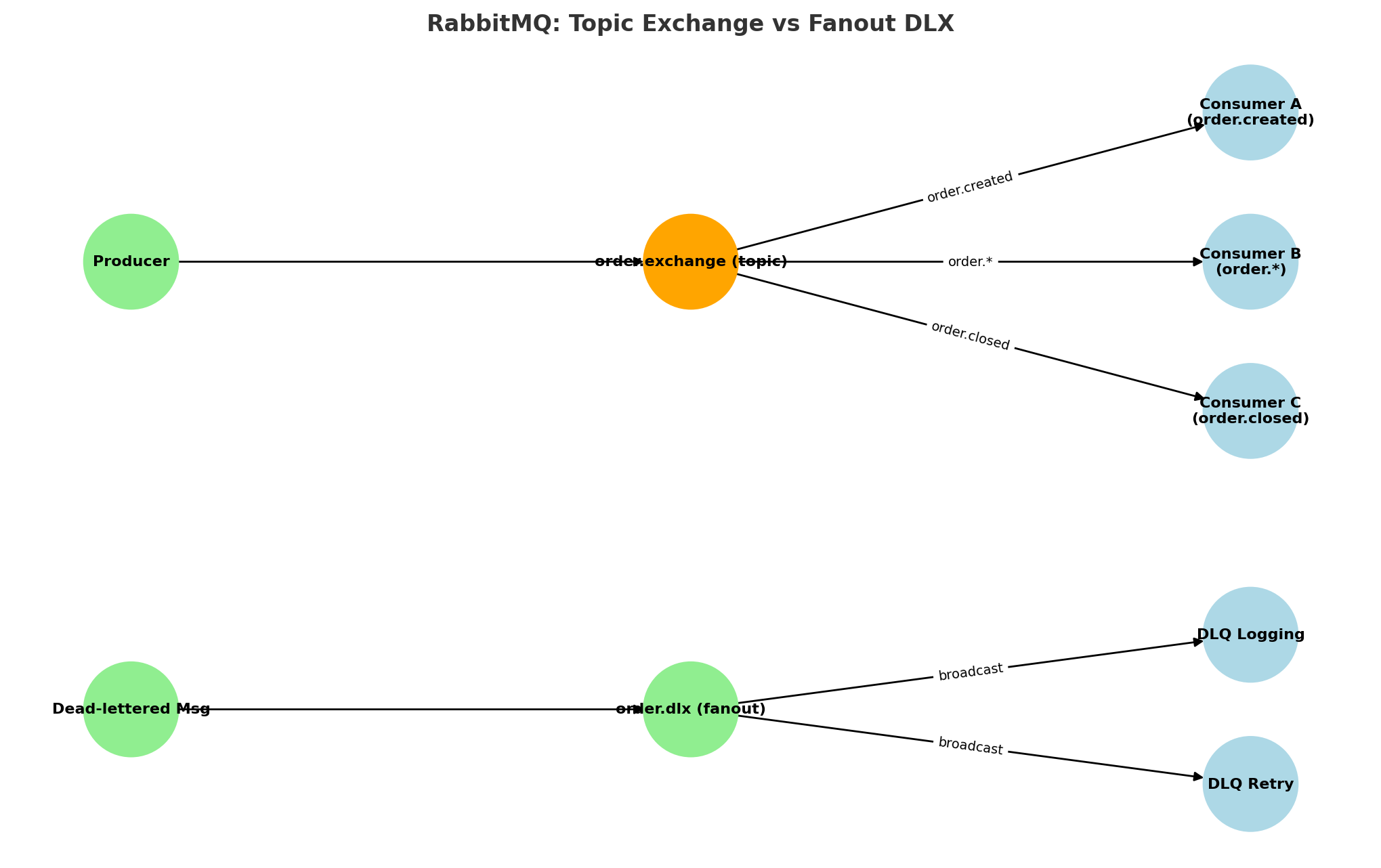
A diagram of a network

AI-generated content may be incorrect.

Rabbit MQ config

**🔹 Exchanges**

| Exchange | Type | Purpose |
| --- | --- | --- |
| order.exchange | topic | Publishes all order lifecycle events (order.created, order.closed, order.cancelled). |
| payment.exchange | topic | Publishes all payment lifecycle events (payment.created, payment.completed, payment.failed). |
| delivery.exchange | topic | Publishes all delivery lifecycle events (delivery.created, delivery.updated, delivery.completed, delivery.failed). |
| order.dlx | fanout | DLX for Order Service queue (failed messages). |
| payment.dlx | fanout | DLX for Payment Service queue (failed messages). |
| delivery.dlx | fanout | DLX for Delivery Service queue (failed messages). |



**🔹 Exchange Basics**

* An **exchange** in RabbitMQ is responsible for receiving messages from producers and routing them to queues based on routing rules.
* Different **types of exchanges** exist depending on how messages should be routed.

**🔹 Topic Exchange** Routes messages based on a **routing key pattern** to allow fine-grained and flexible routing by matching routing keys against **binding patterns** (like order.\*, payment.#).

* **When to use:**
  + When you have events with multiple types/states (created, completed, failed, closed, updated, etc.)
  + Consumers can subscribe to **specific events** or **wildcards**.

**🔹 Fanout Exchange** Broadcasts the message to **all bound queues**, ignoring routing keys.

* **Usage:**
  + Good for broadcasting the same event to multiple services simultaneously.
  + Commonly used for **Dead Letter Exchanges (DLX)**, where you don’t care about routing key but just need all dead messages captured.

**🔹 Why Enterprise Design Uses Them Differently**

* **Business event exchanges (order, payment, delivery)** → topic because they need selective routing (payment.completed vs payment.failed).
* **Dead-letter exchanges (order.dlx, payment.dlx, delivery.dlx)** → fanout because all dead messages should be broadcast to DLQs, monitoring, or retry queues without caring about routing keys.

**🔹 Queues**

| Queue | Purpose |
| --- | --- |
| order-service-queue | Order Service consumes delivery.completed, payment.failed, delivery.failed (final). |
| payment-service-queue | Payment Service consumes order.created. |
| delivery-service-queue | Delivery Service consumes payment.completed and retried messages. |
| delivery-retry-queue | Holds delivery.failed messages for retry (TTL). Routes back to delivery-service-queue. |
| analytics-service-queue | Consumes all lifecycle events via wildcards (order.\*, payment.\*, delivery.\*). |
| notification-service-queue | Consumes customer-facing events (payment.failed, delivery.updated). |
| order-service-dlq | Dead letters from order-service-queue. |
| payment-service-dlq | Dead letters from payment-service-queue. |
| delivery-service-dlq | Dead letters from delivery-service-queue. |

**🔹 Bindings and Purpose**

| **Binding (Exchange → Queue)** | **Routing Key** | **Purpose** |
| --- | --- | --- |
| order.exchange → payment-service-queue | order.created | Trigger Payment Service when order is created. |
| payment.exchange → delivery-service-queue | payment.completed | Trigger Delivery Service when payment is successful. |
| delivery.exchange → order-service-queue | delivery.completed | Trigger Order Service to close order when delivery is complete. |
| order.exchange → analytics-service-queue | order.\* | Feed all order events to analytics. |
| payment.exchange → analytics-service-queue | payment.\* | Feed all payment events to analytics. |
| delivery.exchange → analytics-service-queue | delivery.\* | Feed all delivery events to analytics. |
| payment.exchange → notification-service-queue | payment.failed | Notify customer if payment fails. |
| delivery.exchange → notification-service-queue | delivery.updated | Notify customer about delivery progress. |
| order.dlx → order-service-dlq | N/A | Capture failed messages from Order Service queue. |
| payment.dlx → payment-service-dlq | N/A | Capture failed messages from Payment Service queue. |
| delivery.dlx → delivery-service-dlq | N/A | Capture failed messages from Delivery Service queue. |

**🔹 Topics (Routing Keys / Events)**

**Order Events**

* order.created → Order is persisted with UUID.
* order.closed → Order is marked closed after delivery.
* order.\* → Wildcard for analytics/monitoring.

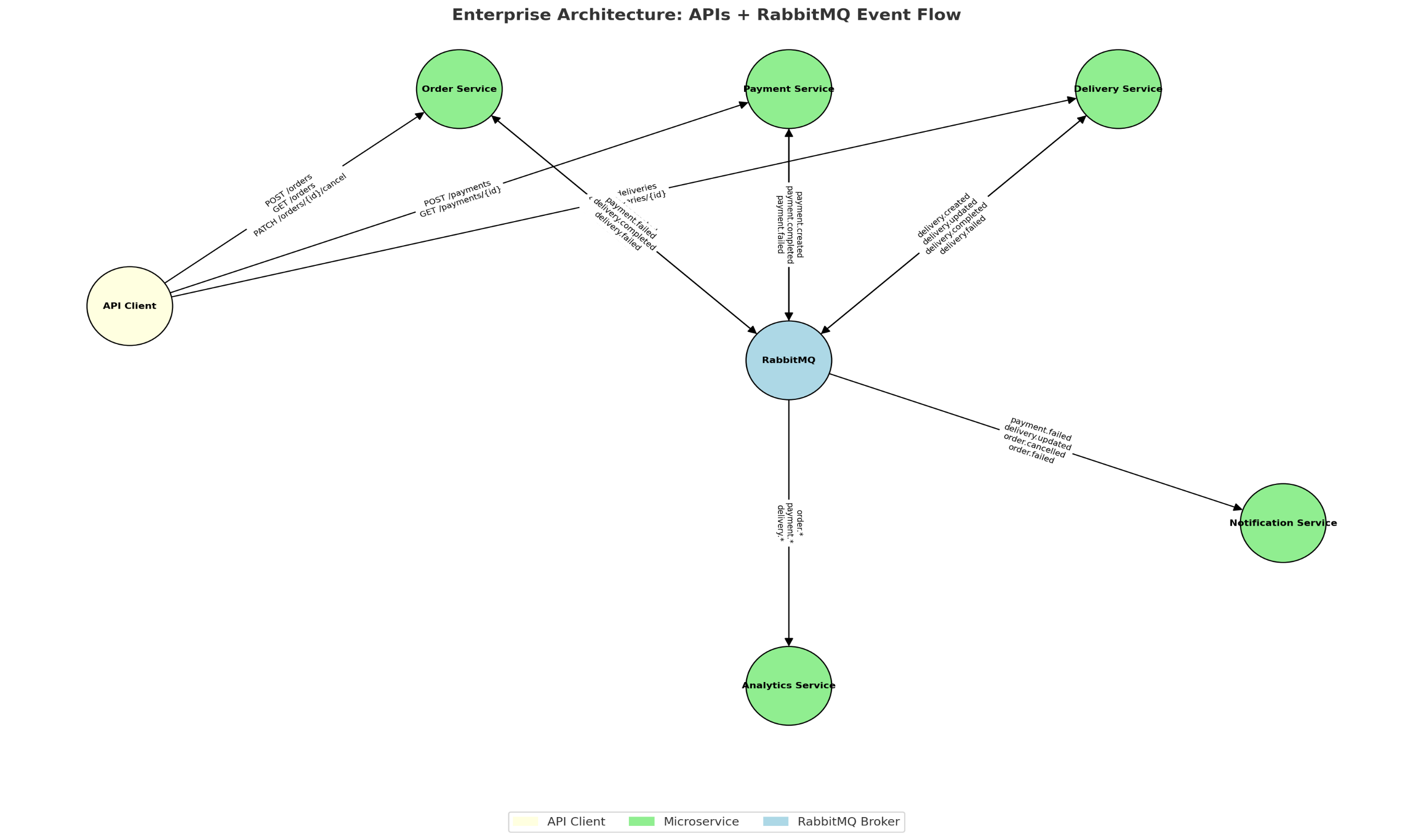
**Payment Events**

* payment.created → Payment record created.
* payment.completed → Payment success.
* payment.failed → Payment failed.
* payment.\* → Wildcard for analytics/monitoring.

**Delivery Events**

* delivery.created → Delivery started.
* delivery.updated → Delivery progress/ETA updates.
* delivery.completed → Delivery completed successfully.
* delivery.\* → Wildcard for analytics/monitoring.

## API design



**1. Order Service**

* **POST /orders** → Place new order (generates OrderUUID, status=CREATED).
* **GET /orders/{id}** → Fetch order details.
* **PATCH /orders/{id}/cancel** → Cancel order manually (rare, but useful).
* Publishes:
  + order.created, order.closed, order.cancelled, order.failed.
* Consumes:
  + payment.failed, delivery.completed, delivery.failed.

**2. Payment Service**

* **POST /payments** → (Optional) Start a payment manually.
* **GET /payments/{id}** → Fetch payment details.
* Publishes:
  + payment.created, payment.completed, payment.failed.
* Consumes:
  + order.created.

**3. Delivery Service**

* **POST /deliveries** → (Optional) Schedule a delivery manually.
* **GET /deliveries/{id}** → Fetch delivery details.
* Publishes:
  + delivery.created, delivery.updated, delivery.completed, delivery.failed.
* Consumes:
  + payment.completed.
  + Retry messages from delivery-retry-queue.

**4. Cross-Cutting Services**

* **Analytics Service**
  + Consumes order.\*, payment.\*, delivery.\*.
  + No APIs exposed (internal reporting).
* **Notification Service**
  + Consumes payment.failed, delivery.updated, order.cancelled, order.failed.
  + Exposes APIs to send email/SMS/Push (to external providers).

## How to Connect to RabbitMQ

**Spring Boot** you can connect to RabbitMQ in three common ways. Here’s a compact summary with when to use each:

**1️⃣ Reactive way (****Reactor RabbitMQ)**

* **Library:** io.projectreactor.rabbitmq
* **Style:** Reactive, non-blocking (Mono, Flux).
* **How:**
  + Create a Sender / Receiver via RabbitFlux.
  + Publish with sender.send(Mono.just(new OutboundMessage(...))).
  + Consume with receiver.consumeAutoAck(queue).subscribe(msg -> …).
* **Pros:**
  + Fully non-blocking → scales well with high concurrency (many publishers/consumers).
  + Works naturally in WebFlux apps.
  + Backpressure-aware (Flux).
* **Cons:**
  + Lower-level API — you manage exchanges/queues, message serialization, retries, DLQs yourself.
  + Less Spring integration (you handle plumbing manually).
* **Best for:** modern **reactive microservices** where throughput and scalability matter.

**2️⃣** **Non-reactive way (Spring AMQP / RabbitTemplate)**

* **Library:** [spring-boot-starter-amqp] → Spring AMQP abstraction over RabbitMQ client.
* **Style:** Blocking, imperative.
* **How:**
  + Publish with rabbitTemplate.convertAndSend(exchange, routingKey, payload).
  + Consume with @RabbitListener(queues = "queueName").
* **Pros:**
  + Very **Spring-friendly** — simple annotations, auto-config, retries, DLQs, message converters built-in.
  + Minimal code for basic pub/sub.
* **Cons:**
  + Blocking → not great for reactive stacks (can cause thread starvation in WebFlux).
  + Less control over backpressure and advanced reactive patterns.
* **Best for:** **classic Spring MVC apps** or when you want productivity over scalability.

**3️⃣** **Spring Cloud Function + Rabbit (function-based)**

* **Library:** [spring-cloud-starter-function-webflux + spring-cloud-stream-binder-rabbit]
* **Style:** Declarative, function-as-a-service style.
* **How:**
  + Define functions (Supplier, Function, Consumer) as beans.
  + Bind them to queues/exchanges via properties.
  + Example:
  + @Bean
  + public Consumer<Order> processOrder() {
  + return order -> { … };
  + }

Config:

spring.cloud.function.definition=processOrder

spring.cloud.stream.bindings.processOrder-in-0.destination=order.created

* **Pros:**
  + Very high-level → no boilerplate for connection, exchange, queue setup.
  + Portable across messaging systems (Kafka, RabbitMQ, etc.) just by changing binder.
  + Great for event-driven “pipeline” apps.
* **Cons:**
  + Less fine-grained control (bindings abstract away details).
  + More Spring Cloud “magic” → harder to debug.
* **Best for:** **event-driven microservices** in Spring Cloud ecosystem, especially when you want to swap RabbitMQ/Kafka without rewriting logic.

## DLQ

**What is a DLQ (Dead-Letter Queue) — short version**

A **Dead-Letter Queue (DLQ)** is a queue where messages go when they *cannot be processed normally*.  
Common reasons a message is dead-lettered:

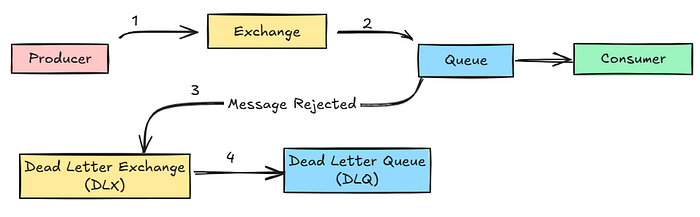
* Consumer **rejected** it (nack) and asked *not* to requeue.
* Message **expired** (TTL — time-to-live) on the original queue.
* Message **length/size** or other broker policy rejected it.
* Message repeatedly failed and was routed by a retry/parking mechanism.

**What it achieves**

* Prevents poison messages from blocking normal processing (avoids infinite retry loops).
* Preserves failed messages for later inspection, reprocessing, or alerting.
* Let’s you implement separate handling strategies (inspect, alert, repair, replay).

**How it works (RabbitMQ basics)**

* You configure the original queue with arguments that point to a *dead-letter exchange* (DLX) and optional routing key(s):
  + x-dead-letter-exchange → name of DLX to publish dead messages to.
  + x-dead-letter-routing-key (optional) → routing key to use when the broker forwards the dead message to the DLX.
* When a message is rejected (consumer nack(false,false)), or expires due to x-message-ttl, or is negatively requeued beyond x-death policy, RabbitMQ republishes the message to the specified DLX.
* The DLX routes it (via bindings) into one or more DLQ queues (e.g., order-error-queue) where you can inspect/process later.



# Spring Cloud Functions

For systems with messaging, we don’t use Rest controller instead we use spring cloud functions

Spring Cloud Function Facilitates the development of business logic by utilizing functions that adhere to the standard interface introduced in Java 8, namely Supplier, Function, and Consumer.

* Supplier: A supplier is a function that produces an output without requiring any input. It can also be referred to as a producer, publisher, or source.
* Function: A function accepts input and generates an output. It is commonly referred to as a processor.
* Consumer: A consumer is a function that consumes input but does not produce any output. It can also be called a subscriber or sink.

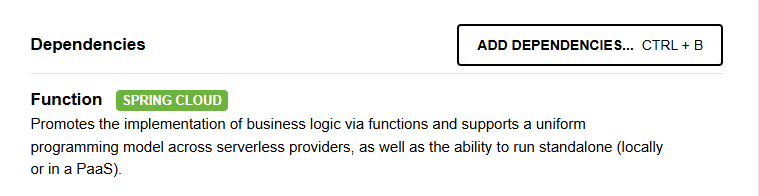
Developing business logic with the help of spring cloud function is going to be simple and at the same time it is going to provide you a lot of flexibility to expose your business logic using various patterns. By default, all your logic that you are going to write inside your functions will be exposed as an REST API automatically by your spring cloud function and if needed, you can also integrate these spring cloud function with the event brokers like Rabbitmq, Apache Kafka by adding one more project which is Spring Cloud Stream.

Spring Cloud Function features:

* Choice of programming styles - reactive, imperative or hybrid.
* POJO Functions (i.e., if something fits the @FunctionalInterface semantics we'll treat it as function)
* Function composition which includes composing imperative functions with reactive.
* REST support to expose functions as HTTP endpoints etc.
* Streaming data (via Apache Kafka, Solace, RabbitMQ and more) to/from Functions via Spring Cloud Stream framework.
* Packaging functions for deployments, specific to the target platform (e.g., AWS Lambda and possibly other "serverless" service providers)

With the spring cloud function, the developer will always focus on building the business logic with the help of functions and exposing that business logic with the various approach is going to be easily achieved by mentioning few properties inside application.yml file and this spring cloud function is best suitable for event driven architecture or event driven model because these functions give flexibility to us to take our business logic wherever we want. Today we have some technology in the future, if some other technology comes, we can easily migrate from one technology to other technology with a minimum configuration.

If we add below dependency



Along with spring cloud function web our web application can accept requests

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-function-web</artifactId>

</dependency>

Then below methods will start accepting web requests with url <http://localhost:8080/email> and [http://localhost:8080](http://localhost:8080/email)/sms

So by adding dependency my functions are exposed as APIs. Similarly we can expose our functions with message brokers like rabbit MQ or kafka

@Bean

public Function<AccountsMsgDto,AccountsMsgDto> email() {

return accountsMsgDto -> {

log.info("Sending email with the details : " + accountsMsgDto.toString());

return accountsMsgDto;

};

}

@Bean

public Function<AccountsMsgDto,Long> sms() {

return accountsMsgDto -> {

log.info("Sending sms with the details : " + accountsMsgDto.toString());

return accountsMsgDto.accountNumber();

};

}

We can also expose our functions as combined API

spring:

application:

name: "message"

cloud:

function:

definition: email|sms

Now we can call both using <http://localhost:8080/email>sms. First email will be called then SMS

# Spring Cloud Stream

Why use Spring Cloud Stream?

Spring Cloud Stream is a framework designed for creating scalable, event-driven, and streaming applications. Its core principle is to allow developers to focus on the business logic while the framework takes care of infrastructure-related tasks, such as integrating with a message broker.

Spring Cloud Stream leverages the native capabilities of each message broker, while also providing an abstraction layer to ensure a consistent experience regardless of the underlying middleware. By just adding a dependency to your project, you can have functions automatically connected to an external message broker. The beauty of this approach is that you don't need to modify any application code; you simply adjust the configuration in the application.yml file.

The framework supports integrations with RabbitMQ, Apache Kafka, Kafka Streams, and Amazon Kinesis. There are also integrations maintained by partners for Google PubSub, Solace PubSub+, Azure Event Hubs, and Apache RocketMQ.

The core building blocks of Spring Cloud Stream are:

* **Destination Binders:** Components responsible to provide integration with the external messaging systems.
* **Destination Bindings:** Bridge between the external messaging systems and application code (producer/consumer) provided by the end user.
* **Message:** The canonical data structure used by producers and consumers to communicate with Destination Binders (and thus other applications via external messaging systems).

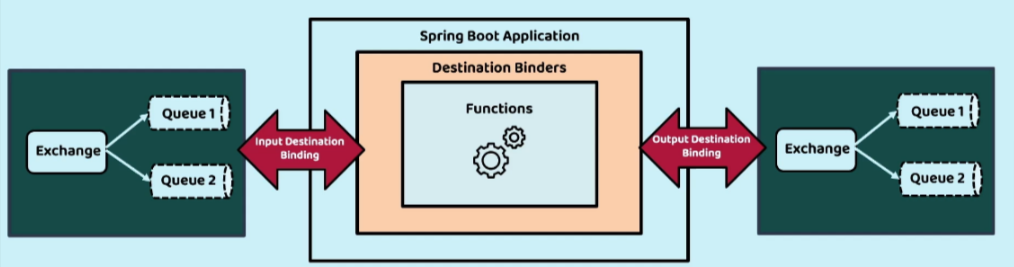
**Why use Spring cloud stream**

Spring Cloud Stream equips a Spring Boot application with a destination binder that seamlessly integrates with an external messaging system. This binder takes on the responsibility of establishing communication channels between the application's producers and consumers and the entities within the messaging system (such as exchanges and queues in the case of RabbitMQ). These communication channels, known as destination bindings, serve as connections between applications and brokers.

A destination binding can function as either an input channel or an output channel. By default, Spring Cloud Stream maps each binding, both input and output, to an exchange within RabbitMQ (specifically, a topic exchange). Additionally, for each input binding, it binds a queue to the associated exchange. This queue serves as the source from which consumers receive and process events. This configuration provides the necessary infrastructure for implementing event-driven architectures based on the pub/sub model.

So the destination binding can be of two types:

1. input destination binding: whereas input destination binding is responsible to read the messages from the queue.
2. output destination binding.Output destination binding, we need to use whenever we are trying to trigger an event from our microservice.



Spring cloud stream will internally have the dependency of spring cloud functions. So we can remove those

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-stream</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-stream-binder-rabbit</artifactId>

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-stream-test-binder</artifactId>

<scope>test</scope>

</dependency>

spring:

application:

name: "message"

cloud:

function:

definition: email|sms \\function details

stream:

bindings:

emailsms-in-0: \\ function name -in property , 0 index

destination: send-communication \\ queuename

group: ${spring.application.name} \\ application name

emailsms-out-0:

destination: communication-sent

rabbitmq:

host: localhost

port: 5672

username: guest

password: guest

connection-timeout: 10s

Sender side

function:

definition: updateCommunication

stream:

bindings:

updateCommunication-in-0:

destination: communication-sent

group: ${spring.application.name}

sendCommunication-out-0:

destination: send-communication

private final StreamBridge *streamBridge*;

private void sendCommunication(Accounts account, Customer customer) {

var accountsMsgDto = new AccountsMsgDto(account.getAccountNumber(), customer.getName(),

customer.getEmail(), customer.getMobileNumber());

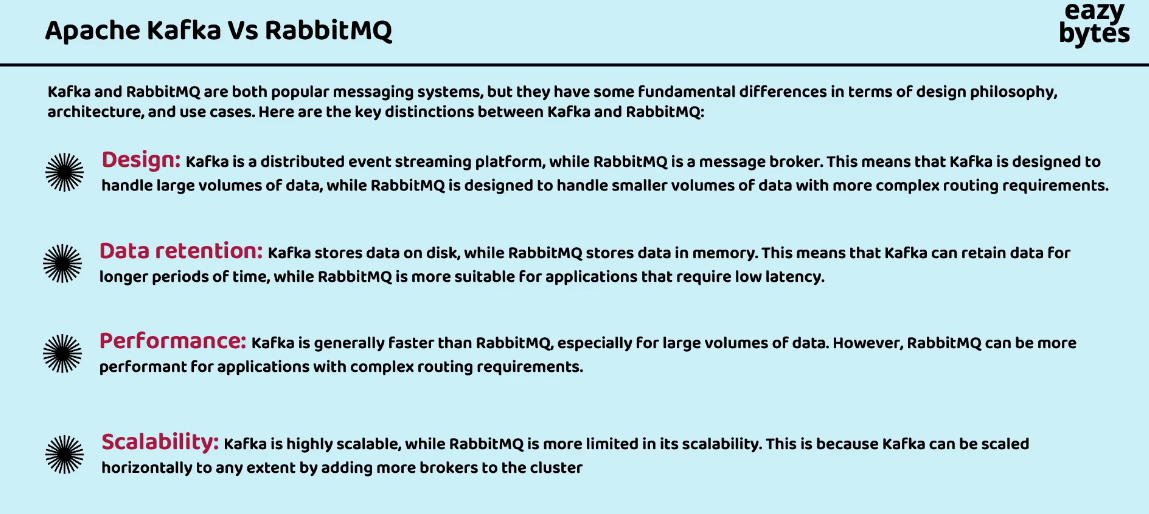
***log***.info("Sending Communication request for the details: {}", accountsMsgDto);

var result = *streamBridge*.send("sendCommunication-out-0", accountsMsgDto);

***log***.info("Is the Communication request successfully triggered ? : {}", result);

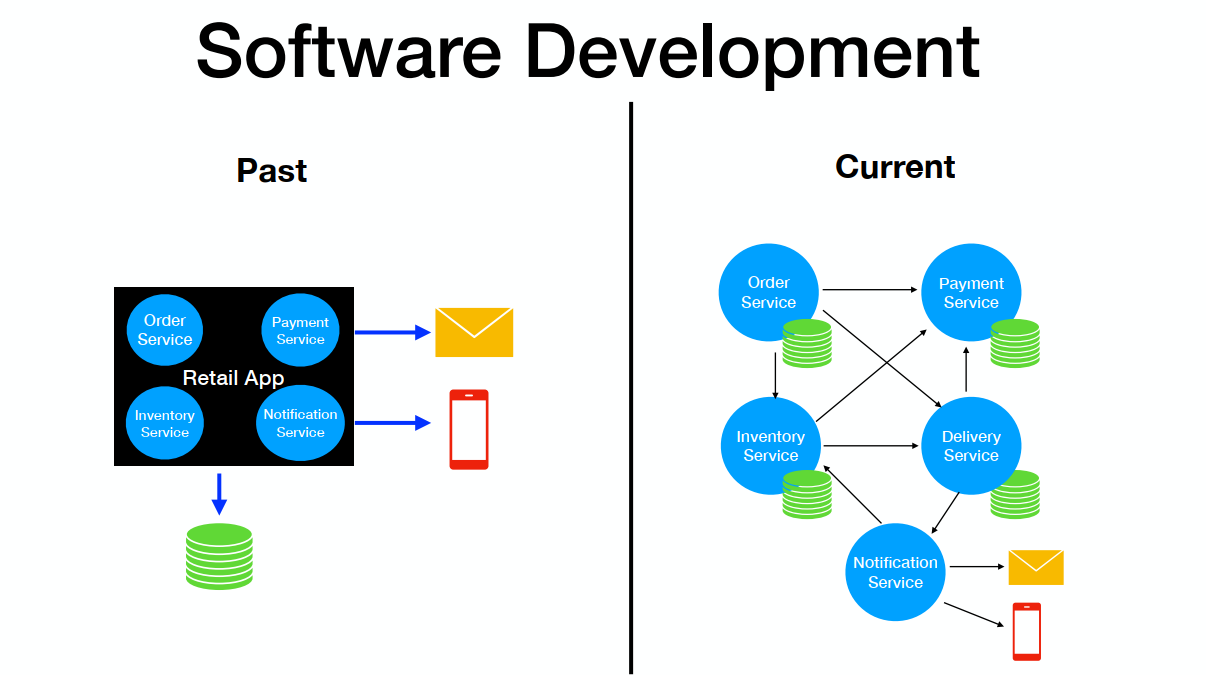
}

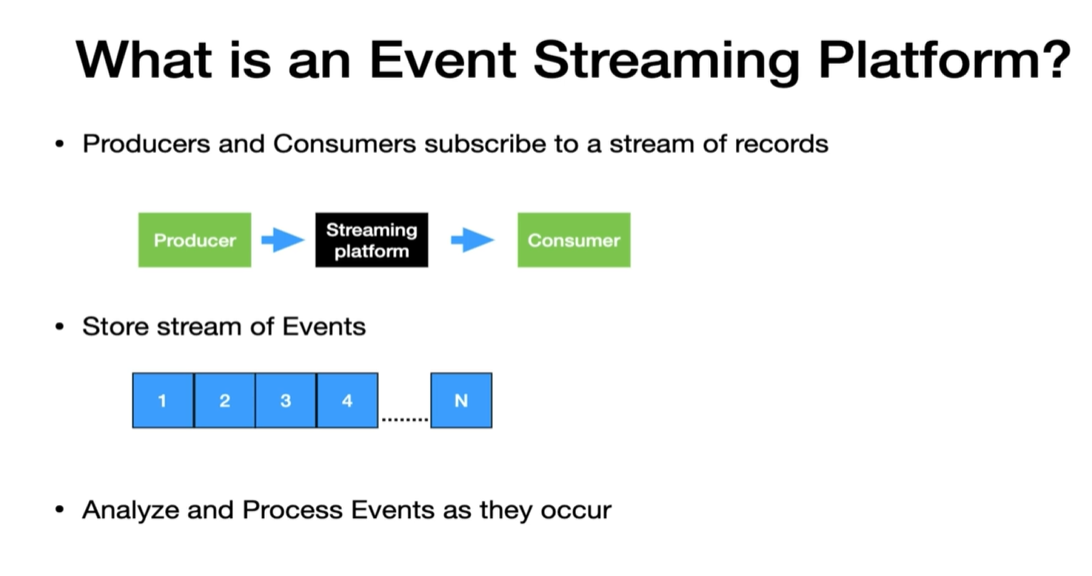
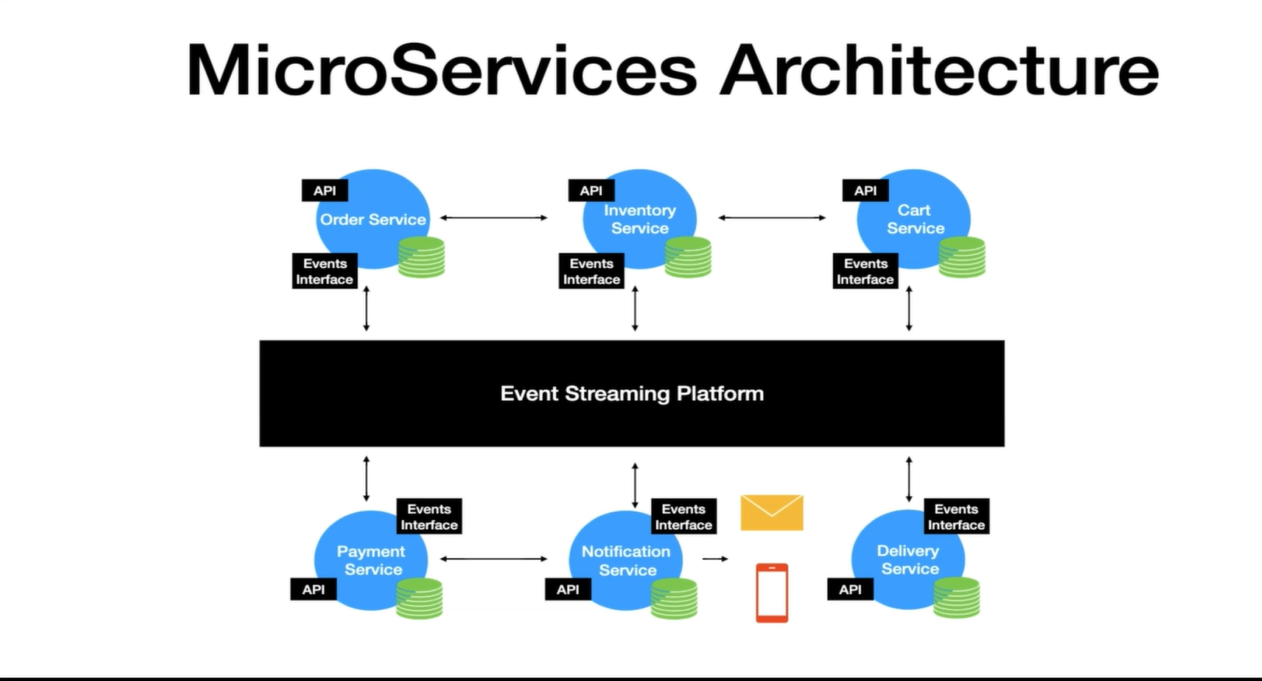
Apache kafka vs rabbit MQ



# Kafka

In microservices due to multiple systems interacting it can be messy to maintain and will cause too many calls between services. So the solution is an event Streaming platform where producers will send and consumers will consume.



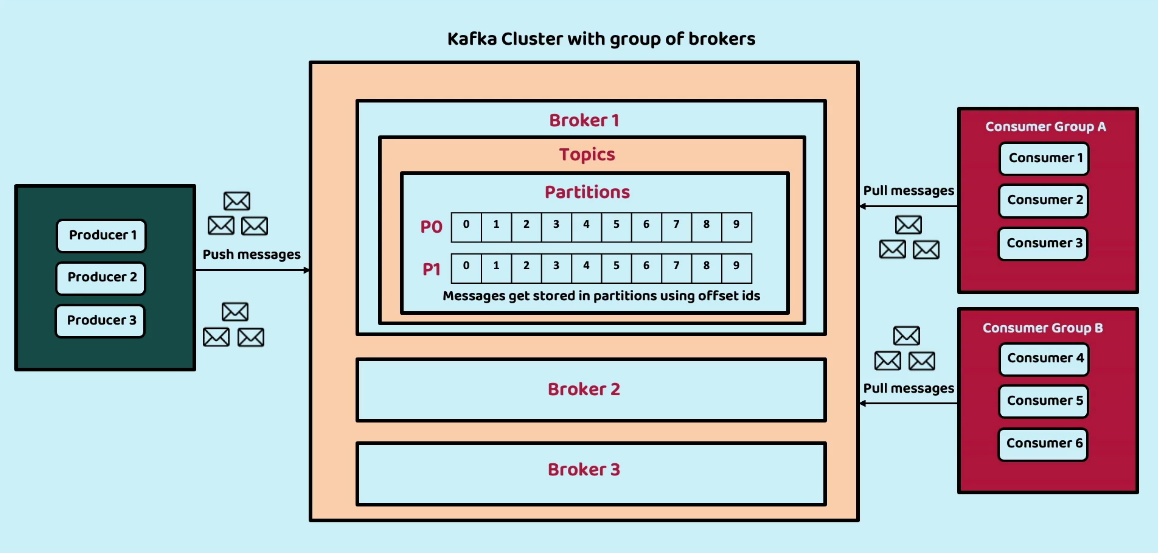


Apache Kafka is an open-source distributed event streaming platform. It is designed to handle large-scale, real-time data streams and enables high-throughput, fault-tolerance, and scalable data processing. It is used to build real-time data streaming pipelines and applications that adapt to the data streams. Key points of Event streaming platform which Kafka implements

* Producer and consumer are independent of each other
* It stores a stream of events so that it can replay a stream of events. Events are generally retained in multiple servers for providing fault tolerance and availability.

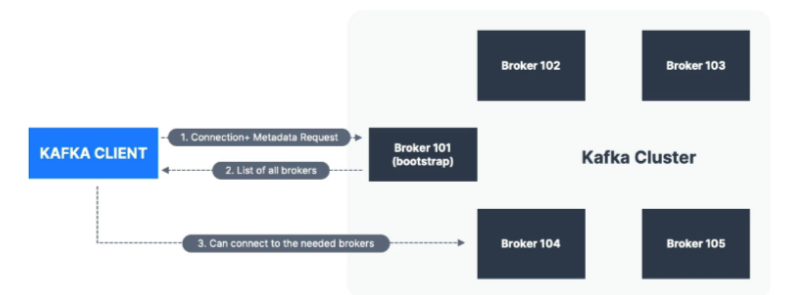
So you can have a producer application continuously sending the data in the format of logs or in the format like Json, XML. So regardless of whatever format that you are trying to use the producers, they can continuously send that data. And on the other side, the consumers, they are going to process the data by accepting the same from Kafka.

## Terminology



1. **Cluster**: It is a **set of servers** (VM or physical host machine) which are going to work together to produce the desired output.
2. **Broker:** Inside the Kafka cluster, we can have any number of brokers. **Broker represents a Kafka server, i**nside the Kafka cluster that can accept the data from the producers and send the same to the consumers.

* Kafka cluster is composed of multiple brokers
* Each broker is identified with its Ids.
* The recommendation is to have **at least three brokers** so that your messages can be replicated with at least two different brokers. Even if we lost one of the brokers, data can be backed up inside broker2.
* Brokers can be in different geographical locations.
* Every Kafka broker is also called a "bootstrap server". That means that you only need to connect to one broker, and the Kafka clients will know how to be connected to the entire cluster (smart clients)
* Each broker knows about all brokers, topics and partitions (metadata)



1. **Topic:** Inside a broker, we have topics which are like the exchange inside Rabbitmq. Producers send messages to Topic. Based upon the scenarios, you can have multiple topics and accordingly you can configure a producer to a specific topic for e.g. we can have separate topics for profile update, address update, payment, delivery status update. And whenever that event happens inside a producer, it is going to push a message or event into a topic.

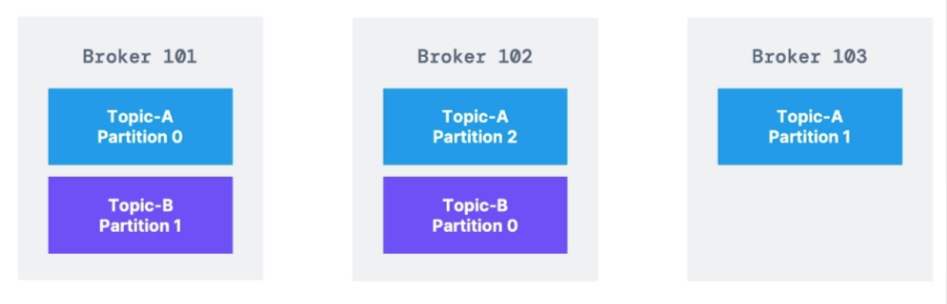
* Messages in topic are a particular stream of data like , address update, payment, delivery status etc.
* Like DB tables but without any constraint
* Identified by name
* Any message format
* Sequence of message called stream
* Can’t query topics. Producers will send data and consumers will read

1. **Partition**: Inside the topics we are going to have multiple partitions. Partition is where the message is stored in Kafka. Kafka can store large volumes of data in files. And it's not advisable to store data in the same place. So, we will store the data on the same topic but in different partitions like P0, P1, P2, P3.

For example, take a scenario where I'm going to trigger an event whenever a new account is created inside a Bank Application. This event is responsible for sending communication to my end user. So, my Bank Application may have millions of customers sending data to a particular topic or into a particular broker, which is not a recommended approach. To handle these kinds of scenarios, Kafka has partitions. I can build logic in such a way that all the communication that I want to send to the New York customer needs to go to the partition zero. And similarly, all the customer communication details related to Washington will must go to the other partition like P1. So, this way, based upon my business use case, I can separate my messages or events to various partitions in a single topic, and this gives me flexibility to store any amount of data because I can add any number of brokers inside my cluster.

* Partitions continue to grow as new records are produced, and offsets get incremented one by one. All records persisted in a physical log file where the Kafka is installed. It is like a commit to database transactions, but this one is distributed
* Ordering is guaranteed only at partition level. So, if you have a use case where you would like to publish and read the records in a certain order, then you must make sure to publish the records to the same partition
* Data is kept only for a specific duration of time default 1 week which is configurable

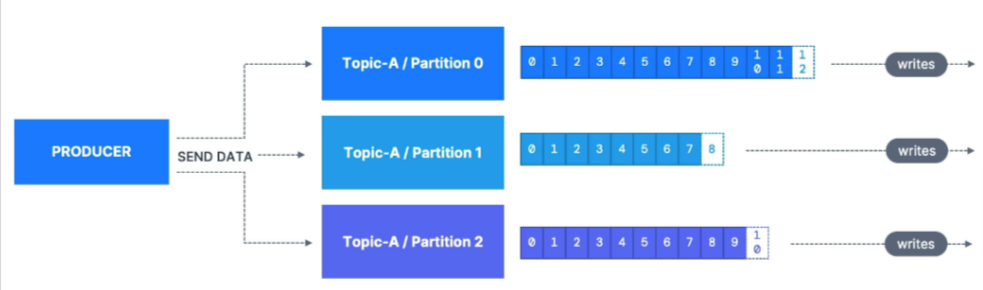
Topic A has 3 partitions each divided into Broker 101, 102 and 103. Topic B has 2 partitions on 101 and 102 and nothing on 103



1. **Producer:** Inside your application you can have any number of producers. Producers will connect to a Kafka cluster, and they are going to be continuously pushing the messages into the Kafka cluster.

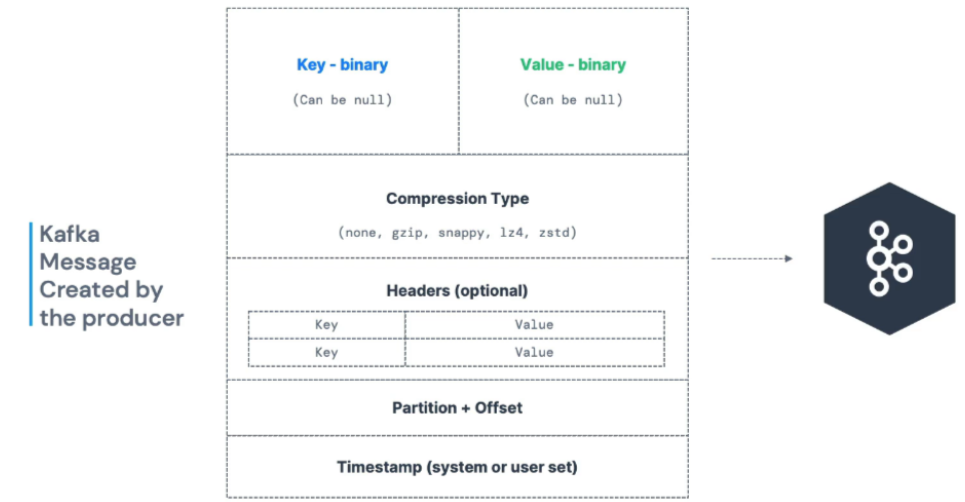
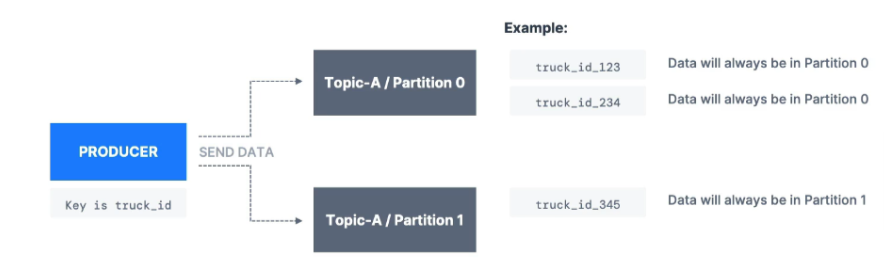
* Producers write data to topics (which are made of partitions)
* Producers know which partition to write to (and which Kafka broker has it)
* In case of Kafka broker failures, Producers will automatically recover

When the producer is invoked for sending the messages. It goes through a lot of layers behind the scenes before the message is sent to Kafka. One of the layers is Partitioner. When no key is sent, the Partitioner will use the round robin approach to send the message across all the partitions. In this approach, there is no guarantee the consumer will be able to read the messages in the same order because the consumer pulls the messages from all the partitions at the same time.



**Message Structure**

* Producers can choose to send a key with the message (string, number, binary, etc.)
* If key=null, data is sent round robin (partition 0, then 1, then 2...)
* If key! =null, then all messages for that key will always go to the same partition (hashing)
* A key is typically sent if you need message ordering for a specific field (ex: zipcode)



|  |  |
| --- | --- |
| * Kafka only accepts bytes as an input from producers and sends bytes out as an output to consumers * Message Serialization means transforming objects / data into bytes * They are used on the value and the key * Common Serializers   + String (incl. JSON)   + Int, Float   + Avro   + Protobuf |  |

1. **Offset:** whenever a message we are trying to store inside a partition, we are going to provide an offset ID to the message right from 0 to 1, two, three, four, and so on. It will keep on increasing the offset number. This will give flexibility to the Apache Kafka and consumers to uniquely identify a particular message. The consumer can keep a track of the messages with the offset id. These offset ID's are similar to sequence ID's that we have inside the database rows.

There can be the same offsetId in multiple partitions. So **the combination of topic partition and the offset ID is going to be unique** always inside your Apache Kafka. So once my message is stored inside a topic with the help of partition and after assigning the offset ID, these messages are ready to be consumed by my consumers.

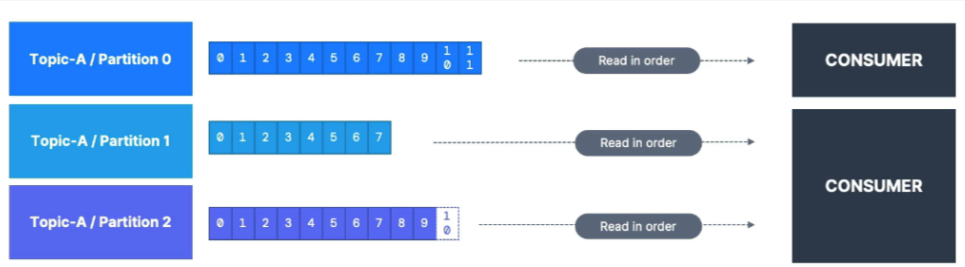
When a consumer sends a request to the broker to connect it sends the topic, partition and offset from where it wants to connect.

1. **Consumers** read data from a topic (identified by name) - pull model. Consumers automatically know which broker to read from. In case of broker failures, consumers know how to recover. Data is read in order from low to high offset within each partitions

Let's say you have a consumer which is going to read the message from the beginning. For any Kafka consumer, it is required for the consumer to provide the group ID otherwise it is created randomly. Now the consumer in general polls and retrieves multiple records at the same time as it processes each message, it moves the consumer read offset one by one. Let's say for some reason the consumer crashed. While the consumer was down the producer of the topic produced some more messages. Now the consumer is brought up after some time. How does it know that it needs to read from offset. It is done as **the consumer offset is stored in an internal topic called \_consumer\_offsets.**

**Consumer Deserializer**

* Deserialize indicates how to transform bytes into objects/data
* They are used on the value and the key of the message
* Common Deserializers.
  + String (incl. JSON)
  + Int, Float
  + Avro
  + Protobuf
* The serialization/ deserialization type must not change during a topic lifecycle (create a new topic instead)

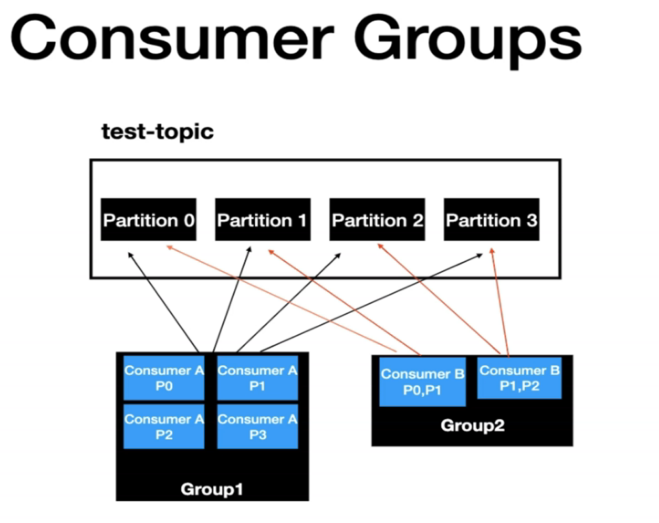
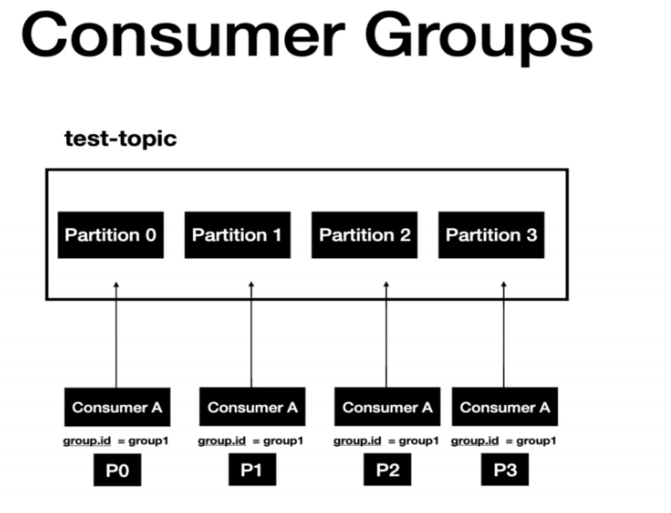
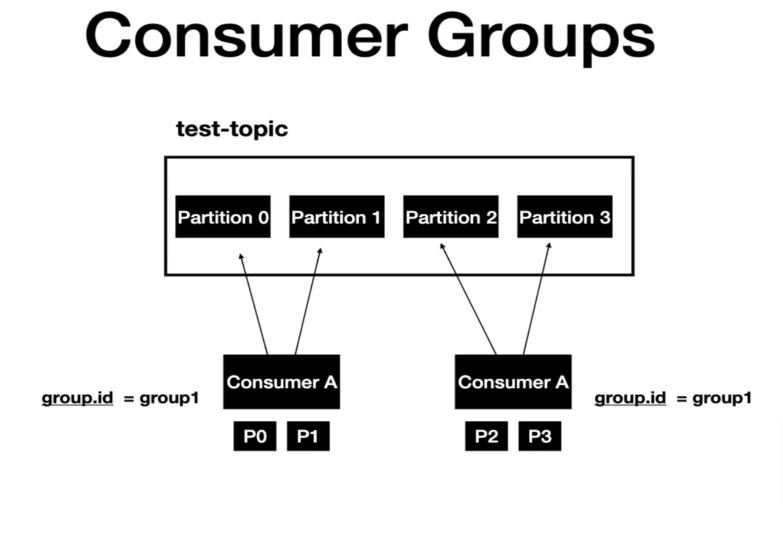


1. **Consumer Group** On the consumer side we can have multiple consumers who are continuously trying to pull the messages from a particular topic and partitions . The consumers will get the partition details where topic data is present and will pull data, and they can be grouped based on consumer group id

Consumer Group ID plays a major role when it comes to scalable message consumption. Let's say we have a topic here named Test Topic and it has four partitions. Now we have a consumer with a group ID, group1, we have one single consumer polling, all the four partitions in the topic and processing them. The poll loop is always single threaded. So in this case, a single thread is going to pull from all the partitions. Let's say the producer of the topic is going to produce messages at a faster rate than the consumer processing rate. Then in that case it will introduce a lag in the consumer and you might end up not processing the events real time.

This is where consumer groups come in handy.Now let's say we spin up another instance of consumer A, but make sure you are using the same group ID. Now the partitions are split between the two instances of the consumer partition zero and partition one is taken care of by the first instance and partition two and partition three is taken care of by the second instance. Basically what this means is that we have scaled our message consumption.

Another use case: Say we have another application reading from Kafka, we can configure it with different consumer group idSo we can spin up more consumer to pull from multiple partitions



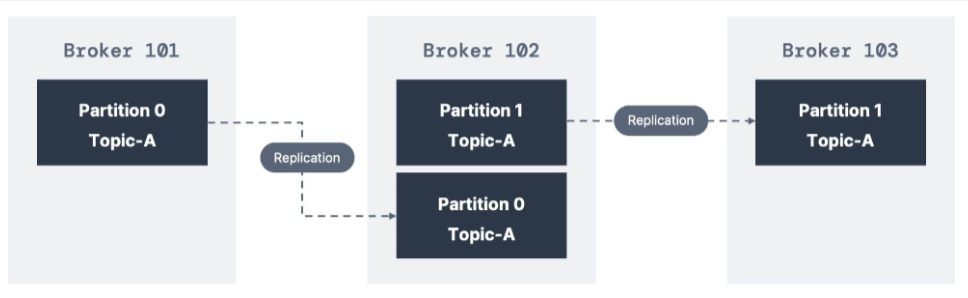
* Kafka stores the offsets at which a consumer group has been reading
* The offsets committed are in Kafka topic named \_\_ consumer\_offsets
* When a consumer in a group has processed data received from Kafka, it should be periodically committing the offsets (the Kafka broker will write to \_\_ consumer\_offsets, not the group itself)
* If a consumer dies, it will be able to read back from where it left off thanks to the committed consumer offsets!



* By default, Java Consumers will automatically commit offsets (at least once)
* There are 3 delivery semantics if you choose to commit manually
  + At least once (usually preferred)
    - Offsets are committed after the message is processed
    - If the processing goes wrong, the message will be read again
    - This can result in duplicate processing of messages. Make sure your processing is idempotent (i.e. processing again the messages won't impact your systems)
  + At most once
    - Offsets are committed as soon as messages are received
    - If the processing goes wrong, some messages will be lost (they won't be read again)
  + Exactly once
    - For Kafka => Kafka workflows: use the Transactional API (easy with Kafka Streams API)
    - For Kafka => External System workflows: use an idempotent consumer

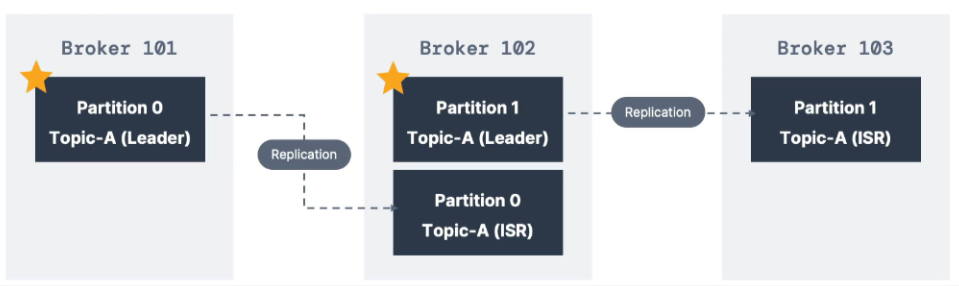
1. **Topic replication factor**

* Topics should have a replication factor > 1 (usually between 2 and 3)
* This way if a broker is down, another broker can serve the data
* Example: Topic-A with 2 partitions and replication factor of 2
* If we lose broker 102 101 and 103 can still server data

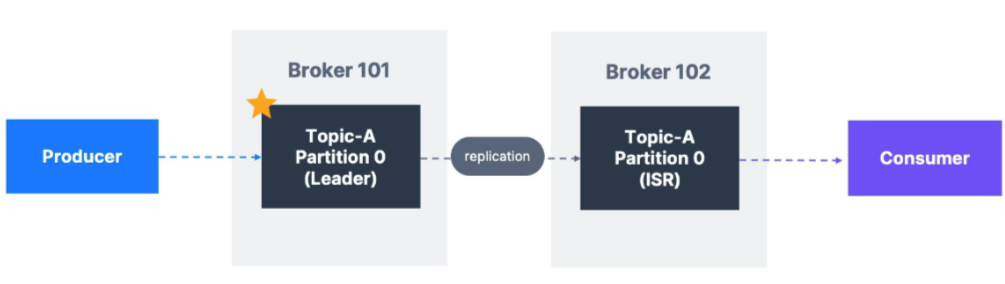


1. **Leader for a Partition**

* At any time only ONE broker can be a leader for a given partition
* Producers can only send data to the broker that is leader of a partition
* The other brokers will replicate the data
* Therefore, each partition has one leader and multiple ISR (in-sync replica)
* Kafka Producers can only write to the leader broker for a partition
* Kafka Consumers by default will read from the leader broker for a partition



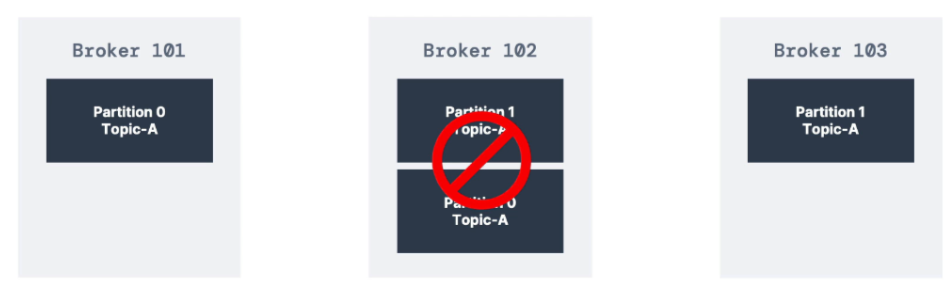
* Since Kafka 2.4, it is possible to configure consumers to read from the closest replica
* This may help improve latency, and also decrease network costs if using the cloud



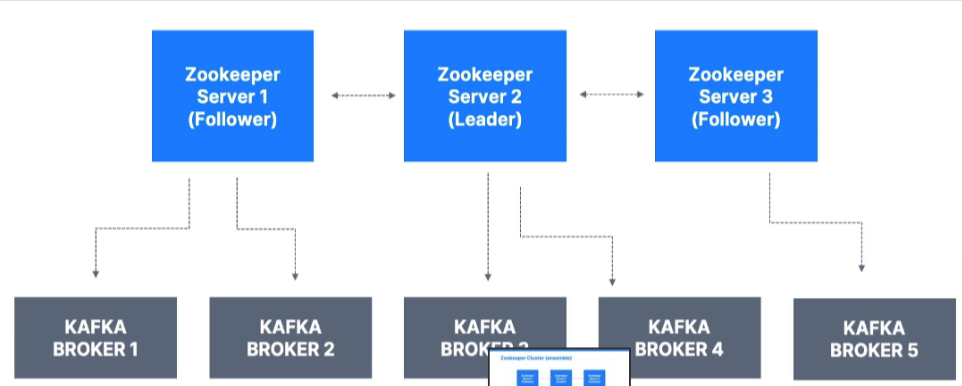
1. **Acknowledgements** Producers can choose to receive acknowledgment of data writes:
   * acks=0: Producer won't wait for acknowledgment (possible data loss)
   * acks=1: Producer will wait for leader acknowledgment (limited data loss)
   * acks=all: Leader + replicas acknowledgment (no data loss)



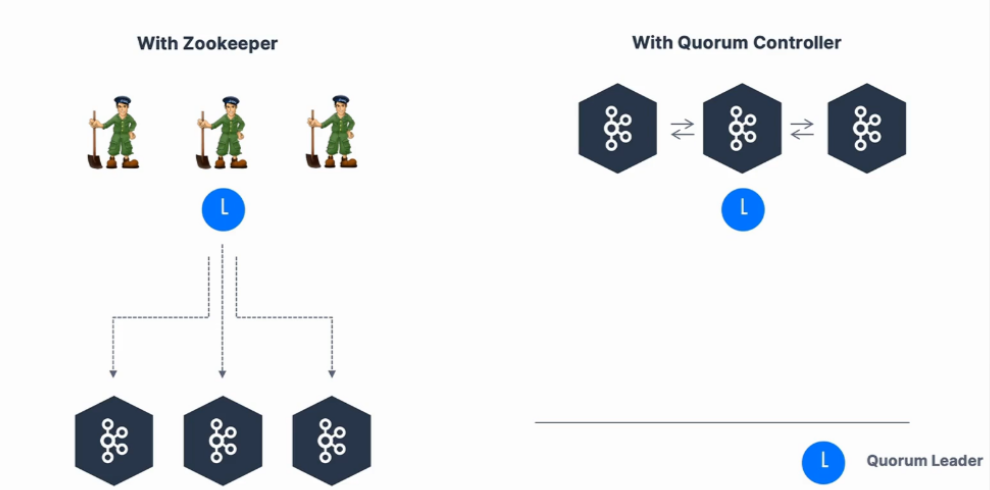
1. **Kafka Topic Durability**
   * For a topic replication factor of 3, topic data durability can withstand 2 brokers loss.
   * As a rule, for a replication factor of N, you can permanently lose up to N-1 brokers and still recover your data**.**

****

1. **Zookeeper**
   * Zookeeper manages brokers (keeps a list of them)
   * Zookeeper helps in performing leader election for partitions
   * Zookeeper sends notifications to Kafka in case of changes (e.g. new topic, broker dies, broker comes up, delete topics, etc....)
   * Kafka 2.x can't work without Zookeeper
   * Kafka 3.x can work without Zookeeper (KIP-500) - using Kafka Raft instead
   * Kafka 4.x will not have Zookeeper
   * Zookeeper by design operates with an odd number of servers (1, 3, 5, 7)
   * Zookeeper has a leader (writes) the rest of the servers are followers (reads)
   * (Zookeeper does NOT store consumer offsets with Kafka > v0.10)



1. **Should you use Zookeeper?**
   * With Kafka Brokers?
     1. Yes, until Kafka 4.0 is out while waiting for Kafka without Zookeeper to be production-ready
   * With Kafka Clients?
     1. Over time, the Kafka clients and CLI have been migrated to leverage the brokers as a connection endpoint instead of Zookeeper
     2. Since Kafka 0.10, consumers store offset in Kafka and Zookeeper and must not connect to
     3. Zookeeper as it is deprecated
     4. Since Kafka 2.2, the kafka-topics.sh CLI command references Kafka brokers and not
     5. Zookeeper for topic management (creation, deletion, etc...) and the Zookeeper CLI argument is deprecated.
     6. All the APIs and commands that were previously leveraging Zookeeper are migrated to use
     7. Kafka instead, so that when clusters are migrated to be without Zookeeper, the change is invisible to clients.
     8. Zookeeper is also less secure than Kafka, and therefore Zookeeper ports should only be opened to allow traffic from Kafka brokers, and not Kafka clients
     9. Therefore, to be a great modern-day Kafka developer, never ever use Zookeeper as a configuration in your Kafka clients, and other programs that connect to Kafka.
2. **About Kafka KRaft**
   * In 2020, the Apache Kafka project started to work to remove the Zookeeper dependency from it (KIP-500)
   * Zookeeper shows scaling issues when Kafka clusters have > 100,000 partitions
   * By removing Zookeeper, Apache Kafka can
     1. Scale to millions of partitions, and becomes easier to maintain and set-up.
     2. Improve stability, makes it easier to monitor, support and administer
     3. Single security model for the whole system
     4. Single process to start with Kafka
     5. Faster controller shutdown and recovery time
   * Kafka 3.X now implements the Raft protocol (KRaft) in order to replace Zookeeper
     1. Production ready since Kafka 3.3.1 (KIP-833)
     2. Kafka 4.0 will be released only with KRaft (no Zookeeper)

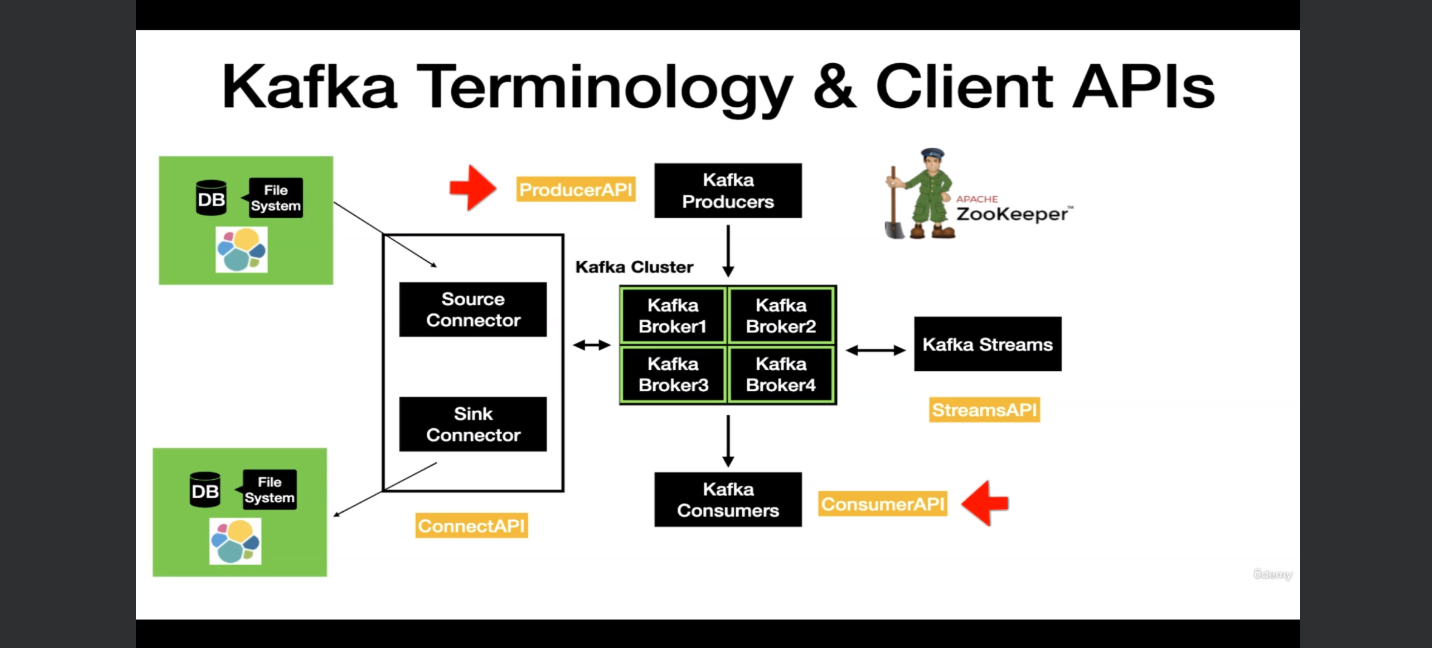


1. **Kafka VS traditional systems**

|  |  |
| --- | --- |
| **Kafka Streaming platform** | **Traditional messaging system** |
| Stores events based on a retention time. Events are immutable | Transient message persistance |
| Consumer is responsible to keep track of consumed messages | Broker is responsible for keeping track of consumed messages |
| Any consumer can access message from broker | Target specific consumer |
| Distributed system | Not usually distributed system |

Traditional messaging systems have the transient message persistence, meaning once the records are read by the consumers, then the messages will be removed from the message broker.

In the case of Kafka, it is going to save the event in the file system where Kafka is installed and the events are retained for a certain time. All the events in Kafka are immutable, meaning once the records are sent to Kafka, then it cannot be altered.



There are two advanced client APIs that come with Kafka.

1. **Kafka Connect.** It has two different types of connectors named

* **Source- Connector** The source Connector is used to pull the data from an external data source, such as database or file system or an Elasticsearch into the Kafka topic.
* **Sink Connector** And the opposite of the same is done using the Sink Connector. With Kafka Connect, you can perform the data movement in and out of Kafka without writing a single line of code.

1. **Streams API** The next advanced API is the, which is basically used to take the data from Kafka and perform simple to complex transformations on it and put it back to Kafka.

To summarize this lecture, you have four client APIs such as the

1. **Producer API**
2. **Consumer API**
3. **Connect API**
4. **Streams API**

using which you can interact with Apache Kafka.

## Set up in GCP

**✅ Option 1: Run Kafka with Docker Compose on a GCE VM (simple, fast)**

1. **Create a VM**

gcloud compute instances create kafka-vm \

--zone=us-central1-a \

--machine-type=e2-medium \

--image-project=debian-cloud \

--image-family=debian-11

1. **SSH into VM**

gcloud compute ssh kafka-vm --zone=us-central1-a

1. **Install Docker & Docker Compose**

sudo apt update

sudo apt install -y docker.io docker-compose

sudo usermod -aG docker $USER

newgrp docker

1. **Copy your docker-compose.yml to the VM**  
   If you’re on your local machine:

gcloud compute scp docker-compose.yml kafka-vm:~/ --zone=us-central1-a

1. **Run Kafka**

docker-compose up -d

1. **Open firewall for Kafka**
   * By default, GCE blocks external traffic. Run:

gcloud compute firewall-rules create kafka-ports \

--allow=tcp:9092,tcp:9093,tcp:9094,tcp:2181 \

--target-tags=kafka-vm

* + Or update the VM’s **VPC firewall rules** in the GCP Console.

1. **Change KAFKA\_ADVERTISED\_LISTENERS in your YAML**  
   Replace localhost with the **external IP** of your VM (check with gcloud compute instances list).

## Kafka CLI

### Basic set up

Create Topic

* Go inside the container by running the command below.

docker exec -it kafka1 bash

* Create a Kafka topic using the kafka-topics command.

kafka-topics --bootstrap-server kafka1:19092 --create --topic test-topic --replication-factor 1 --partitions 1

if you want to do the same from outside the container run below

docker exec --interactive --tty kafka1 kafka-topics --bootstrap-server kafka1:19092 --create --topic test-topic --replication-factor 1 --partitions 1

Check topic

docker exec --interactive --tty kafka1 kafka-topics --bootstrap-server kafka1:19092 --describe

Check specific topic

docker exec --interactive --tty kafka1 kafka-topics --bootstrap-server kafka1:19092 --describe --topic test-topic

### Setup producer consumer

Instantiate **producer** in one terminal

docker exec --interactive --tty kafka1 kafka-console-producer --bootstrap-server kafka1:19092 --topic test-topic

Now Instantiate **consumer** in another terminal

docker exec --interactive --tty kafka1 kafka-console-consumer --bootstrap-server kafka1:19092 --topic test-topic --from-beginning

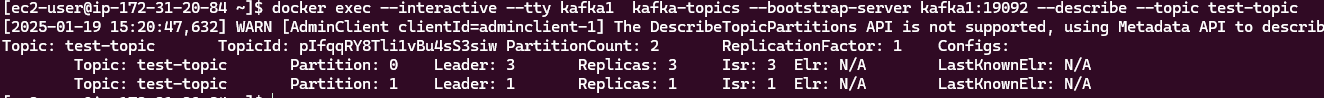
1. Now messages published in the producer terminal will show up in the consumer terminal.
2. There is no consumer group mentioned in the command so when we run the command the Kafka console consumer will act as part of an **anonymous, auto-generated consumer group** with a random group ID.
3. If we run the command multiple times in different windows different consumers will be consuming same messages end by producers

**Describe topic**

docker exec --interactive --tty kafka1 kafka-topics --bootstrap-server kafka1:19092 --describe --topic test-topic

**Change partition** number of partitions to 2

docker exec --interactive --tty kafka1 kafka-topics --bootstrap-server kafka1:19092 --alter --topic test-topic --partitions 2



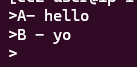
### Messages with keys

Kafka messages have 2 parts, Key(optional) Message. Previous e.g. did not have a key.

When we send messages with keys like above, Key A goes to partition zero and B goes to partition one and so on. But in reality, Key A can go to one of the available partitions of the topic based on the hashing technique. But one thing to keep in mind is that the same key always resolves to the same partition.

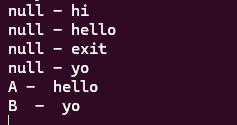
Produce Messages with Key and Value to the topic. Here the key separator is -. So, we must send messages like A- hello, where A is key and hello message

docker exec --interactive --tty kafka1 kafka-console-producer --bootstrap-server kafka1:19092 --topic test-topic --property "key.separator=-" --property "parse.key=true"



Consuming messages with Key and Value from a topic.

docker exec --interactive --tty kafka1 kafka-console-consumer --bootstrap-server kafka1:19092 --topic test-topic --from-beginning --property "key.separator= - " --property "print.key=true"

null for the older message send without keys

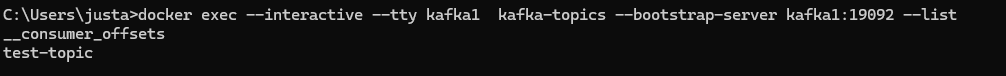
Since we didn’t mention any consumer group, all consumer we set up will pick all the messages as Kafka will set **anonymous, auto-generated consumer group to all consumers**

### Consumer Group and Offset

**Consumer offsets behave like a bookmark for the consumers to go and check from which point in the topic it needs to read the messages from.**

docker exec --interactive --tty kafka1 kafka-topics --bootstrap-server kafka1:19092 --list \_ \_consumer\_offset

A topic named \_\_consumer\_offset is created internally to maintain the offer



Consumer group ID being a mandatory attribute to start up a consumer, if we don’t provide it will generate automatically

### Consumer Groups:

* Consumer Groups are used for scalable message consumption
* Each different application will have a unique consumer group
* Who manages the consumer group?
  + Kafka Broker manages the consumer-groups
  + Kafka Broker acts as a Group Coordinator

Change partition number of **partitions to 2**

docker exec --interactive --tty kafka1 kafka-topics --bootstrap-server kafka1:19092 --alter --topic test-topic --partitions 2

docker exec --interactive --tty kafka1 kafka-topics --bootstrap-server kafka1:19092 --describe --topic test-topic

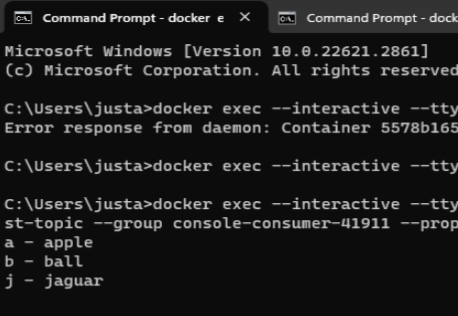
**Create multiple consumer** with same consumer group

docker exec --interactive --tty kafka1 kafka-console-consumer --bootstrap-server kafka1:19092 --topic test-topic --group console-consumer-group-1 --property "key.separator= - " --property "print.key=true"

docker exec --interactive --tty kafka1 kafka-console-consumer --bootstrap-server kafka1:19092 --topic test-topic --group console-consumer-group-1 --property "key.separator= - " --property "print.key=true"

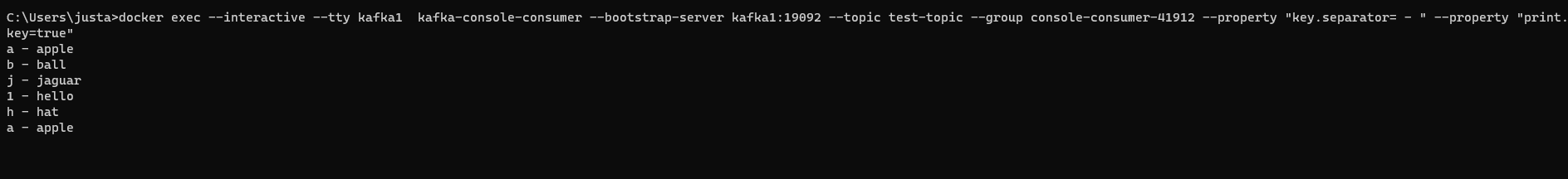
Create a producer with key and value

docker exec --interactive --tty kafka1 kafka-console-producer --bootstrap-server kafka1:19092 --topic test-topic --property "key.separator=-" --property "parse.key=true"



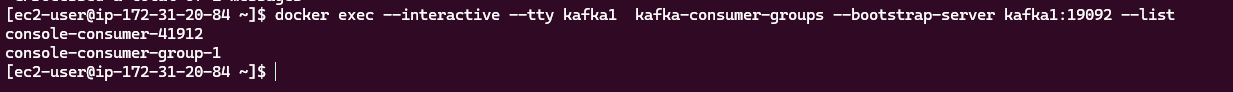
docker exec --interactive --tty kafka1 kafka-console-consumer --bootstrap-server kafka1:19092 --topic test-topic --group console-consumer-2 --property "key.separator= - " --property "print.key=true"

It will consume all the messages that were consumed by the 2 consumer with same consumer group

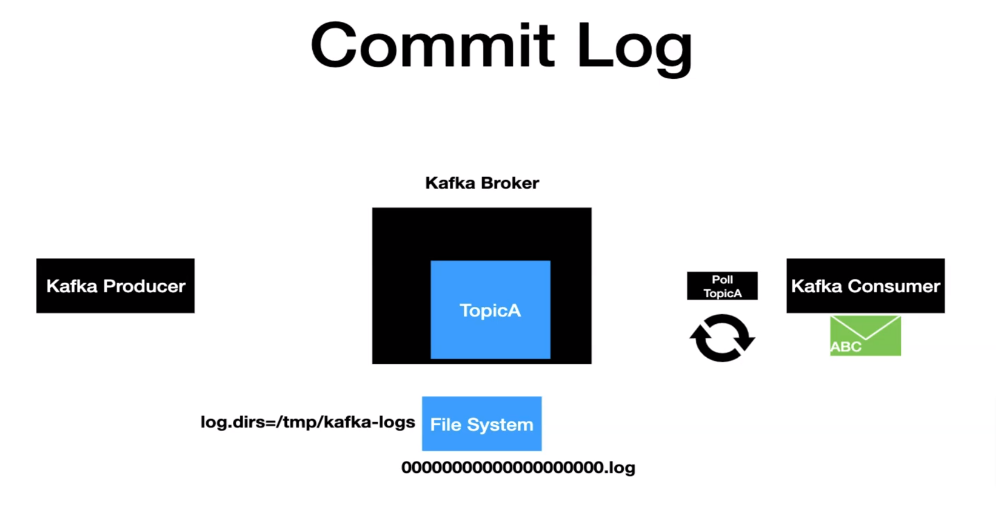


How to view consumer groups

docker exec --interactive --tty kafka1 kafka-consumer-groups --bootstrap-server kafka1:19092 --list



### Commit-log



One of the key qualities of Kafka is the concept of retaining the record for a certain period of time.So here we have the Kafka broker, topic producer and consumer. When the producer sends a message.

* it first reaches the topic, and then the very next thing that happens is that the record gets written to a file system in the machine.
* The record is always written into the file system as bytes.
* The file system where that file needs to be written is configured using the log .dyrs property. The property is available in the server dot properties file. It creates a file with the extension of .log.
* Each partition will have its own log actually. Meaning if we have four partitions, then you will have four log files in the file system. After the messages are written into the log file, that's when the records that got produced are committed.
* So when the consumer who is continuously pulling for new records can only see the records that are committed to the file system as new records are produced to the topic, then the records get appended to the log file and the process continues.
* Retention policy is one of the key properties that's going to determine how long the message is going to be retained. Retention policy is configured using the log.retention.property, which resides inside the server.properties file. The default retention period is 168 hours, or seven days.

docker exec -it kafka1 bash

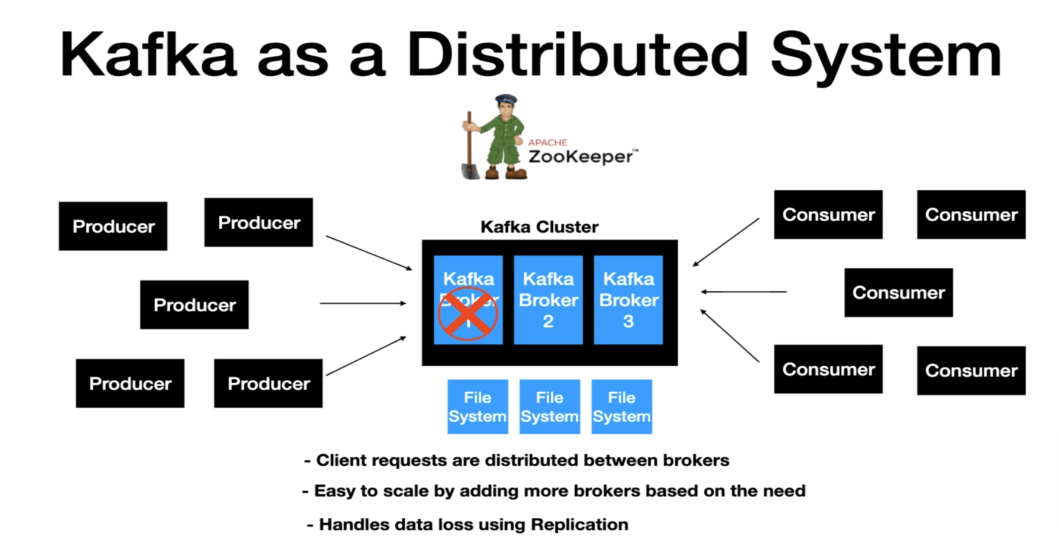
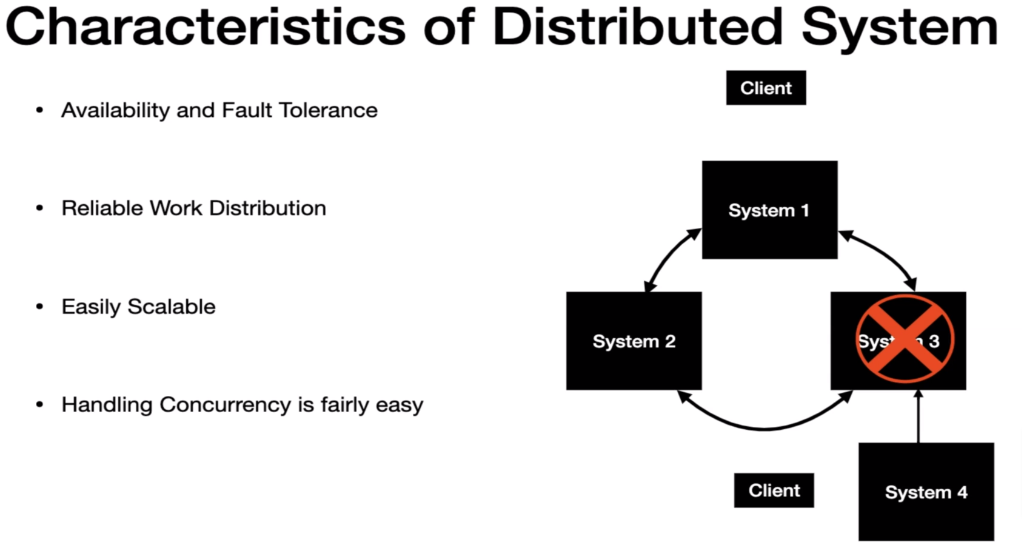
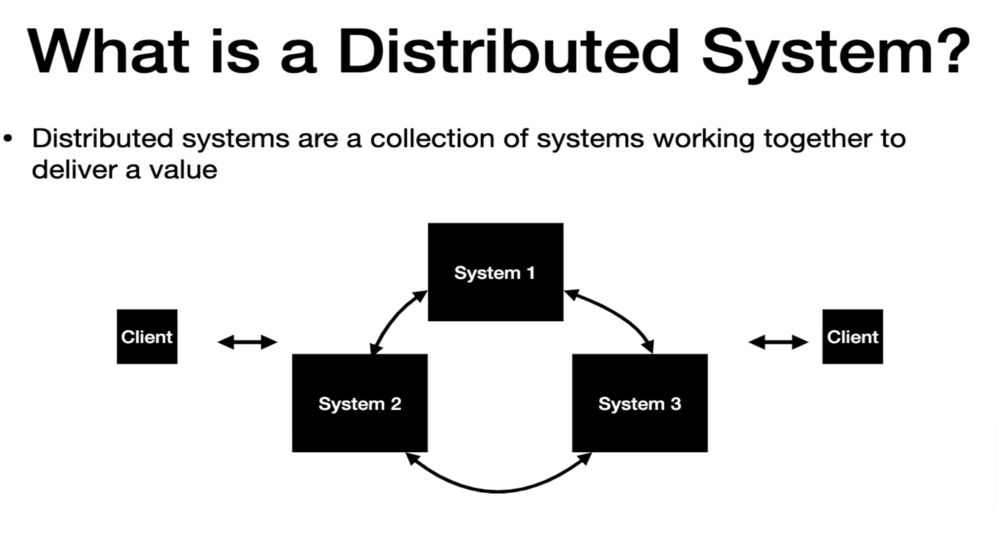
/etc/kafka/server.properties The config file is present in the below path.

/var/lib/kafka/data/ The log file is present in the below path.

### Replication factor

**Kafka as distributed System and Replication factor**

Kafka is a distributed streaming platform.A distributed system in general are a collection of systems, work and interact together in order to deliver the functionality.



Run this command and this will spin up a kafka cluster with 3 brokers.

docker-compose -f docker-compose-multi-broker.yml up

Create topic with the replication factor as 3

docker exec --interactive --tty kafka1 kafka-topics --bootstrap-server kafka1:19092 --create --topic test-topic --replication-factor 3 --partitions 3

Produce Messages to the topic.

docker exec --interactive --tty kafka1 kafka-console-producer --bootstrap-server localhost:9092,kafka2:19093,kafka3:19094 --topic test-topic

Consume Messages from the topic.

docker exec --interactive --tty kafka1 kafka-console-consumer --bootstrap-server localhost:9092,kafka2:19093,kafka3:19094 --topic test-topic --from-beginning

**Kafka handling of data loss**

Let's say the broker one goes down for some reason right now this is the broker, which is the leader of Partition Zero.

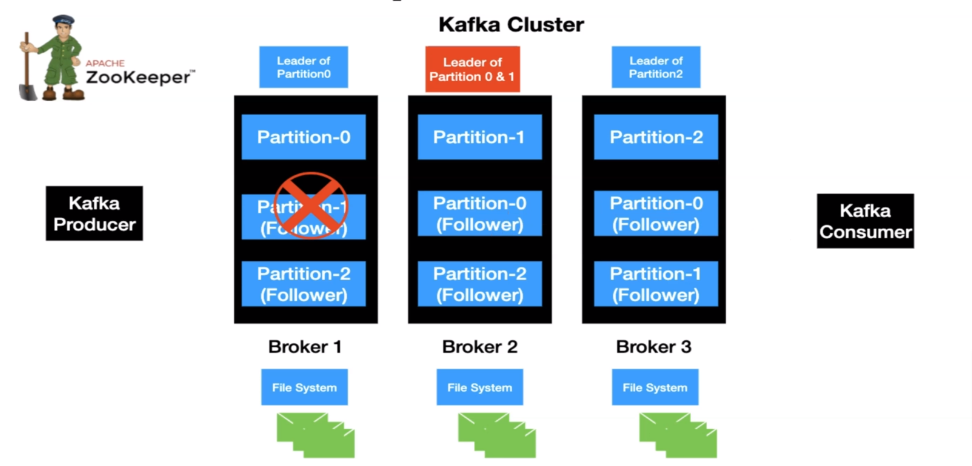
All the data which is written to partition zero is residing in the file system of this broker. Once it goes down, there is no way for the clients to access this data. Kafka handles this issue using the replication factor mentioned during topic creation.

docker exec --interactive --tty kafka1 kafka-topics --bootstrap-server kafka1:19092 --create --topic test-topic --replication-factor 3 --partitions 3

Replication factor is equal to the number of copies of the same message. So when a message is sent to broker 1 the same message is copied to broker two and it gets written into the file system. So Broker two is the follower of Partition Zero, which is also known as the follower replica, and the same step is repeated for Broker three.

Let's say the broker one is down, but still the data of the partition is available in broker two and broker three. Zookeeper gets notified about the failure and it assigns the new leader to the controller. Now the broker two is the leader of Partition zero and Partition one. This leader assignment is taken care of by the controller node, which is the part of the cluster actually. So now the client's request for producing and consuming the data for Partition zero will go to broker two hereafter.

So this is how Kafka handles data loss.



### In-Sync Replica

This represents the number of replicas in sync with each other in the Kafka cluster. This includes both the leader and Follower replica.

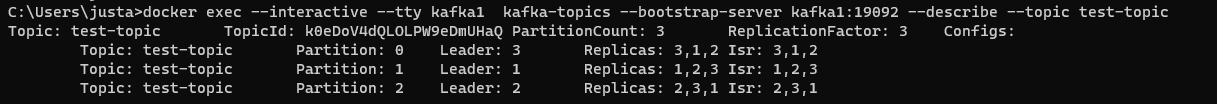
* The in sync replica is always recommended to be greater than one.
* The ideal state of replication is ISR equal to replication factor.

List topics

docker exec --interactive --tty kafka1 kafka-topics --bootstrap-server kafka1:19092 --describe

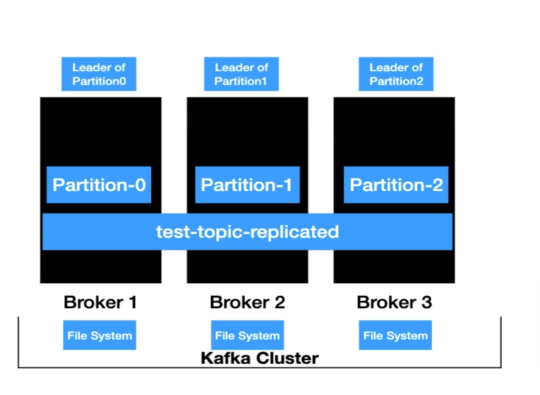
Describe a topic to see

docker exec --interactive --tty kafka1 kafka-topics --bootstrap-server kafka1:19092 --describe --topic test-topic



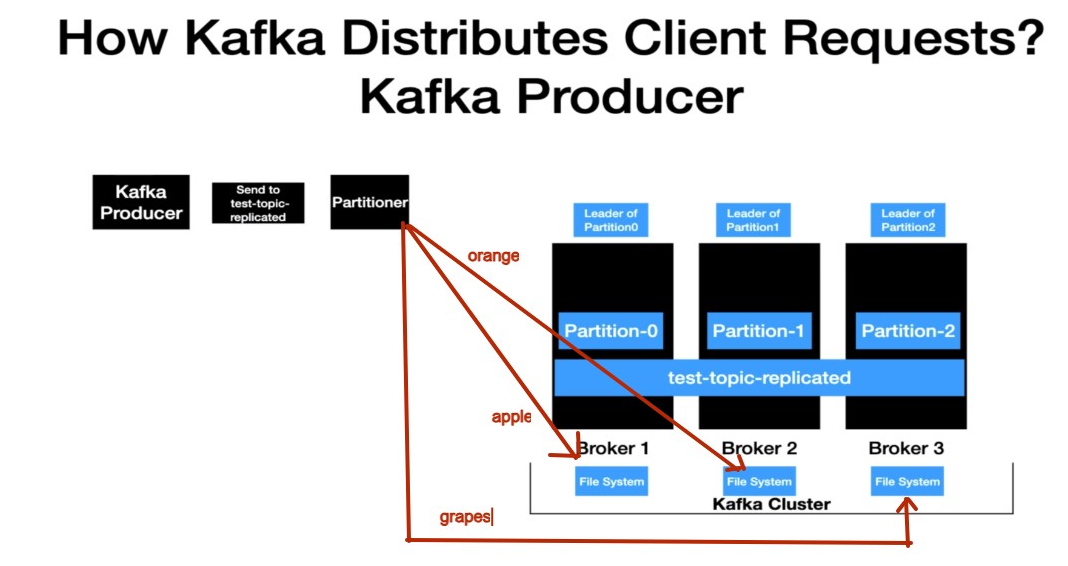
**Kafka distributed architecture**

**Broker** In this example, we have a cluster with three brokers. Out of the three brokers, one of the available brokers will behave as a controller. Normally, this is the first broker to join the cluster. When the Create Topic command is issued to the zookeeper, the zookeeper takes care of redirecting this request to the controller. The role of this controller is to distribute the ownership of the partitions to the available broker. When the Create Topic command is issued to the zookeeper, the zookeeper takes care of redirecting this request to the controller. The role of this controller is to distribute the ownership of the partitions to the available broker. So in this example we have partition zero sitting in broker one, partition, one sitting in broker two and partition two sitting in broker three. In distributed systems. This concept of distributing partitions to the brokers is called leader assignment. So in a nutshell, the test topic is distributed across the Kafka cluster.



**Producer**

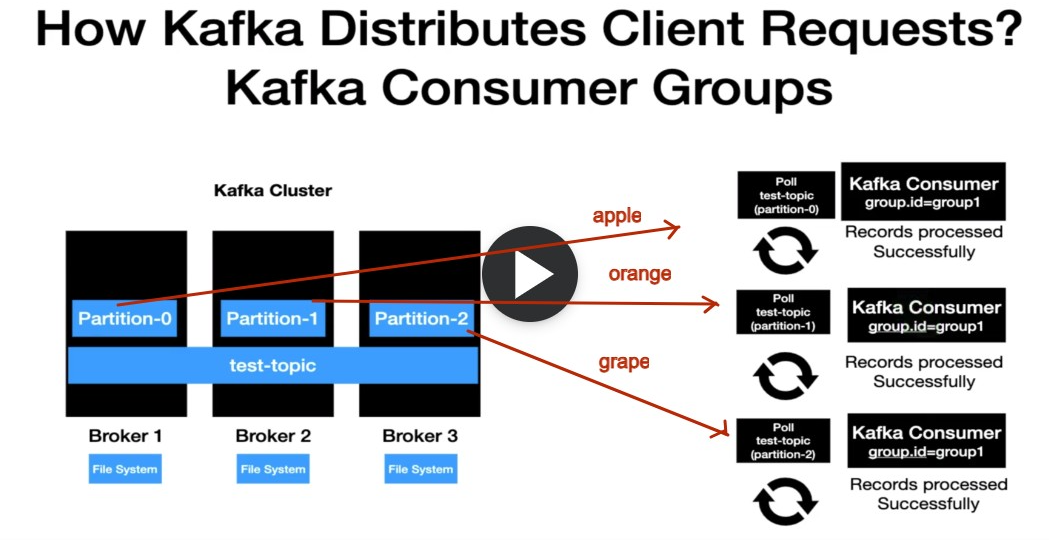
Producer has a layer called Partitioner which takes care of determining which partition the message is going to go. So the producer sends the first message. It's good it goes to the partitioner before the message is sent to the Kafka topic. In this case, the leader of Partition zero is broker one, so the message will be sent to broker one. The client will always invoke the leader of the partition. After that, the message is persisted into the file system of broker one. The process is repeated for the following messages. Now we have another message that goes to the partitioner. The Partitioner resolves it to partition one and the leader of the partition one is broker two. So the request is directed to the broker two and it gets persisted into the file system of broker two. Next, the messages. It follows the same step. The message resolves it to partition two and the leader of the partition two is broker three. So the message is directed to broker three. So as you can see, the client requests from the producer end are distributed between the brokers based on the partition, which indirectly means that the load is distributed between the brokers.



**Consumer**

We have the consumer ready to poll the test topic replicated topic. When the poll loop is executed, the request goes to all the partitions and retrieves the records from them. Here, each broker owns a partition. In this case, the poll loop request goes to all the brokers and retrieves the record from them. And the retrieved records are handed over to the Kafka consumer and the consumer process, the records successfully, and the same flow repeats. So in a nutshell, even from the consumer end, the requests to retrieve the data are distributed between the brokers. Basically, the client call will only go to the partition leader of the topic and retrieve the data.

Now let's take a look at the Kafka consumer flow with consumer groups, So in here we have three instances of the consumer with the same group ID, if you can recall from the concepts of consumer groups, if there are one or more instances of the consumer with the same group ID, then the partitions are distributed for scalable message consumption. So here, each consumer instance has one partition assigned. When the poll loop gets executed, each instance is going to poll the data from the partition that they are interested in, and the poll call goes to the leader of the partition of the topic and retrieves the data. So here, partition zero call goes to broker one and partition one call goes to broker two and so on and the process repeats.

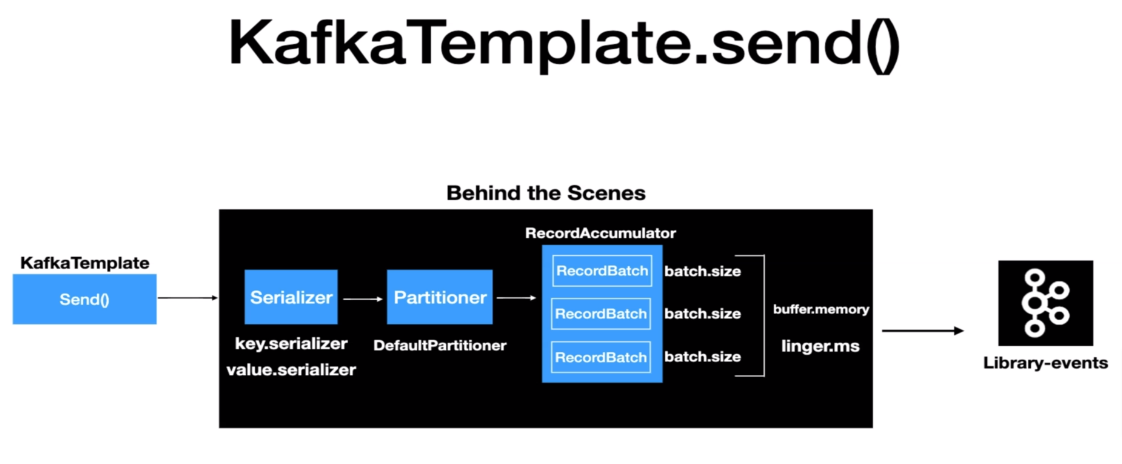


Quickly summarize

* Partition leaders are assigned during topic creation.
* Any clients, irrespective of the producer or consumer, will invoke the leader of the partition to produce or consume data.
* With this, the load is evenly distributed by invoking the partition leaders.

# Kafka Producer in Spring

Kafka template is a class that's part of the spring to produce messages into the Kafka topic. A quick analogy is to think of this as a Jdbc template for DB interactions.



KafkaTemplate.send() is going to send the message to Kafka, but in reality it goes through different layers before the message is sent to the Kafka.

The very first layer is the serializer. Any record that's sent to the Kafka needs to be serialized, parsed to bytes.There are two different types of serialization techniques that's applied to any record.

* key serializer,
* Value Serializer.

This configuration is mandatory for any producer. The client needs to provide the key serializer value and value serializer value. The Kafka client Java Libraries comes with some predefined serializers.

The second layer is a partitioner. This layer determines which partition the message is going to go into the topic. The Kafka producer API comes up with the default partitioner logic, and in most cases that's enough to handle the partitioning logic and there are options to override this default Partitioner logic too.

The next layer is the record accumulator. Any record that's sent from the Kafka template won't get sent to the topic immediately. The record accumulator buffers, the records and the records are sent to the Kafka topic once the buffer is full.The reason for this approach is to limit the number of trips from the application to the Kafka cluster, and this eventually avoids the overhead of bombarding the Kafka cluster with numerous requests, which also helps in improving the overall performance of the system. The record batch that you see here is a representation of the topic partition combination. If we have a topic with three partitions, then you will have three record batches as it's shown in the slide. Each and every record batch has a batch size which is represented by the batch.size property and the value is represented as the number of bytes. It also has an overall buffer memory which is represented by the property buffer.memory, and this value is also represented as the number of bytes.

So under what scenarios? The messages are sent to the Kafka topic. Once the batch is full, then the message will be sent to the Kafka topic. There may be many scenarios where the record batch won't fill up. The producer API is not going to wait for so long to send the message to the Kafka topic. There is also another handy property called linger.ms , which will be used in this case to publish the records into the Kafka topic. As the name suggests, this value is represented in milliseconds. If the batch is not full and the records accumulated meet the linger dot Ms. value, then the records will be sent to Kafka.

We need to provide these mandatory values to configure a simple version of Kafka template.

**spring.kafka.producer.bootstrap-servers**=localhost:9095,localhost:9096,localhost:9097

**spring.kafka.producer.key-serializer**=org.apache.kafka.common.serialization.IntegerSerializer

**spring.kafka.producer.value-serializer**=org.apache.kafka.common.serialization.StringSerializer

## How Kafka Auto configuration works

In the dependencies in classpath this file is there KafkaAutoConfiguration.class

It has this annotation @ConditionalOnClass(KafkaTemplate.class) which checks if Kafka is present in classpath or not and below code

@ConditionalOnMissingBean(KafkaTemplate.class)

public KafkaTemplate<?, ?> kafkaTemplate(ProducerFactory<Object, Object> kafkaProducerFactory,

ProducerListener<Object, Object> kafkaProducerListener,

ObjectProvider<RecordMessageConverter> messageConverter)

Which loads ProducerFactory which loads KafkaProperties which has properties which are overridden by application.properties/yaml

private Class<?> keyDeserializer = StringDeserializer.class;

private Class<?> valueDeserializer = StringDeserializer.class;

private List<String> bootstrapServers = new ArrayList<>(Collections.singletonList("localhost:9092"));

So basic flow for auto configuration is

Look for Kafka dependencies in classpath → If already there then look for any Kafkatemplate bean → If not there load KafkaProducer which will load KafkaProperties

## Create Topic Programmatically



Not recommended

spring.kafka.topic=library-events

spring.kafka.admin.properties.bootstrap.servers=localhost:9092,localhost:9093,localhost:9094

@Configuration

@Profile("local")

public class AutoCreateConfig {

@Value("${spring.kafka.topic}")

public String topic;

@Bean

public NewTopic libraryEvents(){

return TopicBuilder.*name*(topic)

.partitions(3)

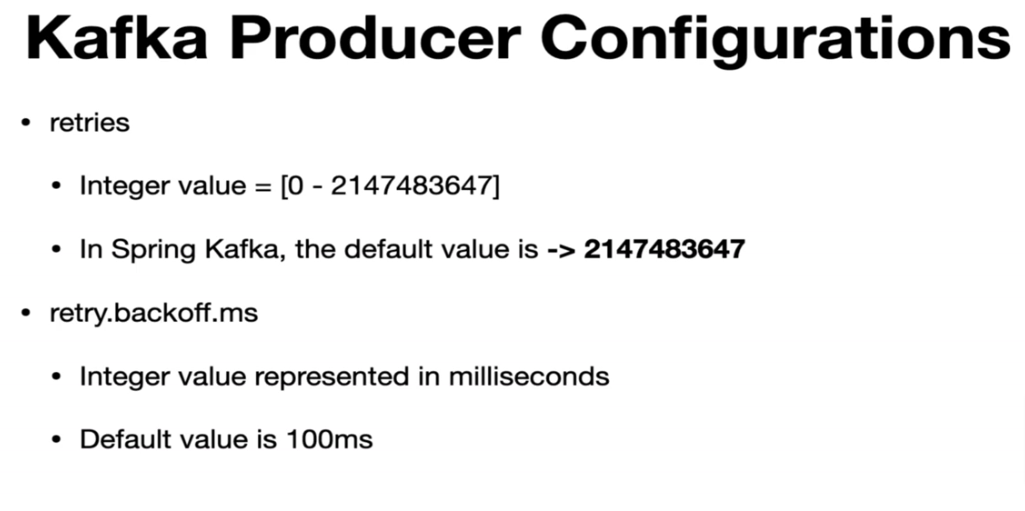
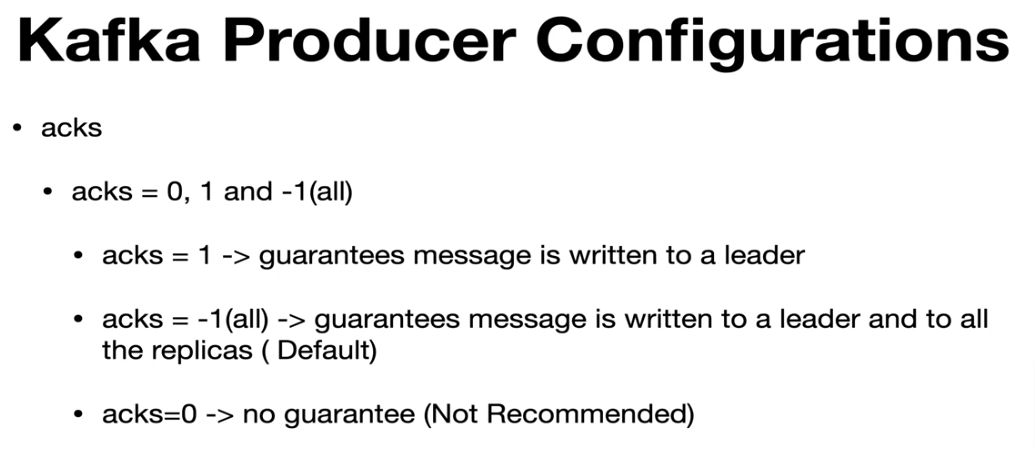
.replicas(3)

.build();

}

}

# Kafka Producer



Sending message

spring.kafka.producer.bootstrap-servers=localhost:9092,localhost:9093,localhost:9094

spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer

spring.kafka.producer.value-serializer=org.apache.kafka.common.serialization.StringSerializer

spring.kafka.producer.properties.acks=all

spring.kafka.producer.properties.retries=10

spring.kafka.producer.properties.retry.backoff.ms=1000

Approach 1 which returns completable future, non blocking call

public CompletableFuture<SendResult<String, String>> sendKafkaEvent(ProductDTO productDto) throws JsonProcessingException {

String key = productDto.getCategory();

String value = objectMapper.writeValueAsString(productDto);

var completableFuture = kafkaTemplate.send("product", key, value);

return completableFuture.whenComplete((stringStringSendResult, throwable) ->

{

if (null != throwable)

log.error("Error in sending to kafka");

else

log.info("Message successfully send to kafka" + stringStringSendResult);

});

}

Approach2 Blocking call

public SendResult<String, String> sendKafkaEvent2(ProductDTO productDto) throws JsonProcessingException, ExecutionException, InterruptedException, TimeoutException {

String key = productDto.getCategory();

String value = objectMapper.writeValueAsString(productDto);

//Blocking call

return kafkaTemplate.send("product", key, value).get(3, TimeUnit.SECONDS);

}

Approach3 bundling everything in Producer record

public CompletableFuture<SendResult<String, String>> sendKafkaEvent3(ProductDTO productDto) throws JsonProcessingException {

String key = productDto.getCategory();

String value = objectMapper.writeValueAsString(productDto);

ProducerRecord<String, String> producerRecord = buildProducerRecord(key, value, "product");

var completableFuture = kafkaTemplate.send("product", key, value);

return completableFuture.whenComplete((stringStringSendResult, throwable) ->

{

if (null != throwable)

log.error("Error in sending to kafka");

else

log.info("Message successfully send to kafka" + stringStringSendResult);

});

}

private ProducerRecord<String, String> buildProducerRecord(String key, String value, String topic) {

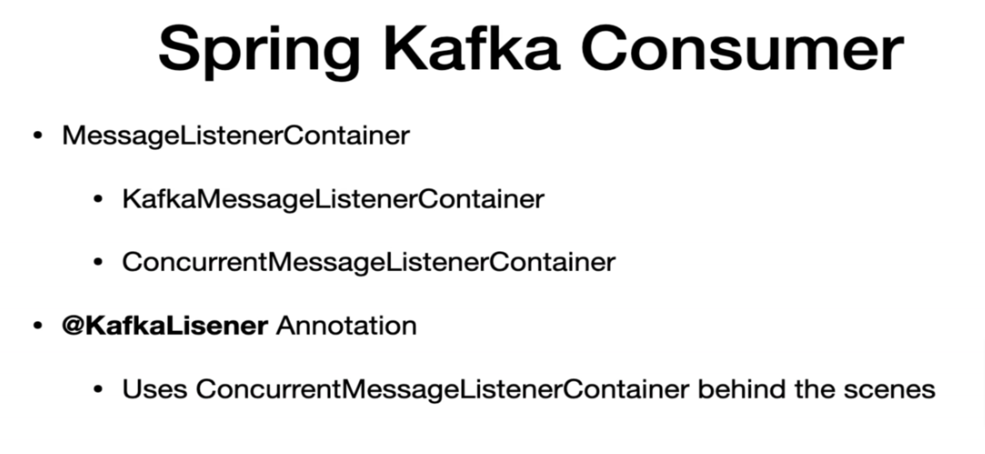
List<Header> recordHeaders = List.of(new RecordHeader("event-source", "scanner".getBytes()));

return new ProducerRecord<>(topic, null, key, value, recordHeaders);

}

# Consumer Section

## Read from kafka Topic



**KafkaMessageListenerContainer**



ConcurrentMessageListenerContainer → can use multiple instance of it which can read message concurrently

**@KafkaListener implementation**

*@Component*

*@Slf4j*

*public class* LibraryEventsConsumer {

*@KafkaListener*(topics = {"library-events"})

*public void* onMessage(ConsumerRecord<Integer, String> consumerRecord) {

log.info("ConsumerRecord : {} ", consumerRecord);

}

}

spring.kafka.consumer.bootstrap-servers=localhost:9092,localhost:9093,localhost:9094

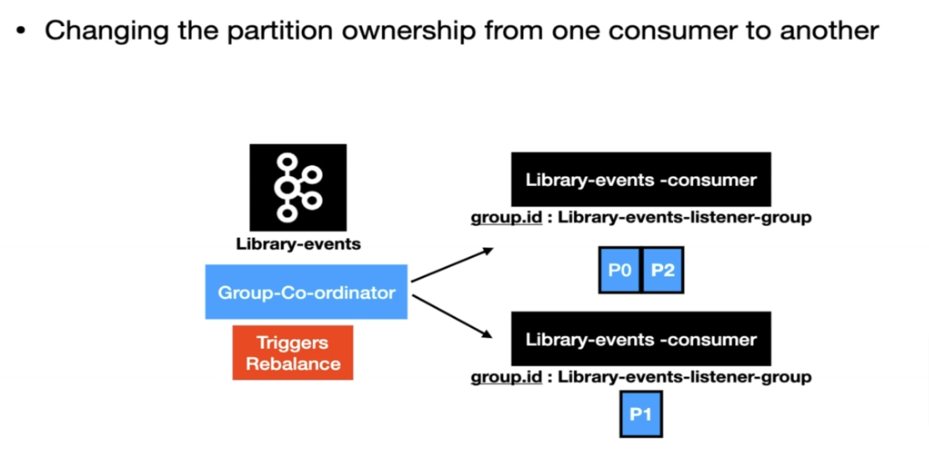
spring.kafka.consumer.key-serializer=org.apache.kafka.common.serialization.IntegerSerializer

spring.kafka.consumer.value-serializer=org.apache.kafka.common.serialization.StringSerializer

spring.kafka.consumer.group-id=library-events-listener-group

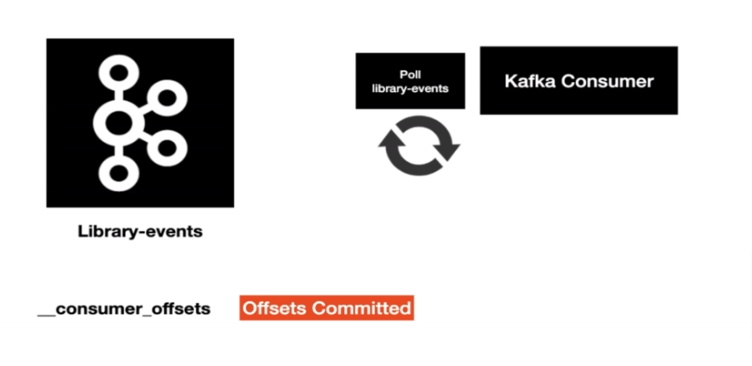
## Rebalance Consumer Group

Rebalance is triggered when more consumers are added., like below where initially there was one consumer reading from 3 partitions then one more was added. Now Consumnr 1 reads from partition 0 and 2 and 2nd consumer reads from partition 1.



## Offset management

By default once the message is read after that if we restart the consumer it will not read the message again. This is because after the message is read an offset is committed to kafka which tells that that the message is read already.



Options for committing offset

Several options are provided for committing offsets. If the enable.auto.commit consumer property is true, Kafka auto-commits the offsets according to its configuration. If it is false, the containers support several AckMode settings (described in the next list). The default AckMode is BATCH. Starting with version 2.3, the framework sets enable.auto.commit to false unless explicitly set in the configuration. Previously, the Kafka default (true) was used if the property was not set.

The consumer poll() method returns one or more ConsumerRecords. The MessageListener is called for each record. The following lists describes the action taken by the container for each AckMode (when transactions are not being used):

* RECORD: Commit the offset when the listener returns after processing the record.
* BATCH: Commit the offset when all the records returned by the poll() have been processed.
* TIME: Commit the offset when all the records returned by the poll() have been processed, as long as the ackTime since the last commit has been exceeded.
* COUNT: Commit the offset when all the records returned by the poll() have been processed, as long as ackCount records have been received since the last commit.
* COUNT\_TIME: Similar to TIME and COUNT, but the commit is performed if either condition is true.
* MANUAL: The message listener is responsible to acknowledge() the Acknowledgment. After that, the same semantics as BATCH are applied.
* MANUAL\_IMMEDIATE: Commit the offset immediately when the Acknowledgment.acknowledge() method is called by the listener.

Commonly used options are Batch(default) and manual(Application takes control of when to commit)

## Custom Offset management

We can manage our own offset In order to override default offset define a bean of ConcurrentKafkaListenerContainerFactory

*@Configuration*

*@EnableKafka*

*@Slf4j*

*public class* LibraryEventsConsumerConfig {

*@Bean*

ConcurrentKafkaListenerContainerFactory<?, ?> kafkaListenerContainerFactory(

ConcurrentKafkaListenerContainerFactoryConfigurer configurer,

ObjectProvider<ConsumerFactory<Object, Object>> kafkaConsumerFactory) {

ConcurrentKafkaListenerContainerFactory<Object, Object> factory = *new* ConcurrentKafkaListenerContainerFactory<>();

configurer.configure(factory, (ConsumerFactory<Object, Object>) kafkaConsumerFactory);

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL); **// set as manual**

*return* factory;

}

}

Then in kafka listener implement and can implement AcknowledgingMessageListenerthe listener where we can implement our acknowledgement logic with method acknowledgment.acknowledge();

*@Component*

*@Slf4j*

*public class* LibraryEventsConsumerManualOffset *implements* AcknowledgingMessageListener<Integer,String> {

*@Override*

*@KafkaListener*(topics = {"library-events"})

*public void* onMessage(ConsumerRecord<Integer, String> consumerRecord, Acknowledgment acknowledgment) {

log.info("ConsumerRecord in Manual Offset Consumer: {} ", consumerRecord );

acknowledgment.acknowledge();

}

}

## Spin concurrent consumer

If your application is not running in a cloud like environment, In those kinds of scenarios, this option is really handy to spin up multiple instances of the message listener. If you're using Kubernetes, then this option is not necessary.

factory.setConcurrency(3);

*@Bean*

ConcurrentKafkaListenerContainerFactory<?, ?> kafkaListenerContainerFactory(

ConcurrentKafkaListenerContainerFactoryConfigurer configurer,

ObjectProvider<ConsumerFactory<Object, Object>> kafkaConsumerFactory) {

ConcurrentKafkaListenerContainerFactory<Object, Object> factory = *new* ConcurrentKafkaListenerContainerFactory<>();

configurer.configure(factory, (ConsumerFactory<Object, Object>) kafkaConsumerFactory);

factory.setConcurrency(3);

*return* factory;

}

}

## Retries

We can set error handling and pass in Kafka Listener factory for re tries

public DefaultErrorHandler errorHandler() {

//retries with Fixed backoff

var fixedBackOff = new FixedBackOff(1000L, 2);

var errorHandler = new DefaultErrorHandler(fixedBackOff);

return errorHandler;

}

@Bean

ConcurrentKafkaListenerContainerFactory<?, ?> kafkaListenerContainerFactory(

ConcurrentKafkaListenerContainerFactoryConfigurer configure,

ConsumerFactory<Object, Object> kafkaConsumerFactory) {

ConcurrentKafkaListenerContainerFactory<Object, Object> factory = new ConcurrentKafkaListenerContainerFactory<>();

configure.configure(factory, kafkaConsumerFactory);

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL);

factory.setCommonErrorHandler(errorHandler());

factory.setConcurrency(3);

return factory;

}

## Exponential backoff retries

public DefaultErrorHandler errorHandler() {

ExponentialBackOffWithMaxRetries expBackOff = new ExponentialBackOffWithMaxRetries(2);

expBackOff.setInitialInterval(1\_000L);

expBackOff.setMultiplier(2.0);

expBackOff.setMaxInterval(2\_000L);

errorHandler = new DefaultErrorHandler(expBackOff);

return errorHandler;

}

## Additional logging

public DefaultErrorHandler errorHandler() {

//retries with Fixed backoff

var fixedBackOff = new FixedBackOff(1000L, 2);

var errorHandler = new DefaultErrorHandler(fixedBackOff);

//For additional logging

errorHandler.setRetryListeners(((record, ex, deliveryAttempt) -> {

log.error("Error in consuming product for "+ record +" error " +ex+ " deliveryAttempt "+deliveryAttempt );

}));

return errorHandler;

}

## Specific retry scenarios

Sometimes we don’t want retry in specific scenarios, for e.g. when data format is is incorrect

# Integration test for Kafka Consumer

Consumer is just receiving messages. There is not much logic so we will just write an integration test.

We are going to instantiate the producer, which is a Kafka template, so that we can publish

the message into the Kafka topic, basically the embedded Kafka topic so that our consumer can read.

So we will initiate producer in the properties file and overwrite with embedded Kafka properties in Integration test class

Start with

*@SpringBootTest*

*@EmbeddedKafka(topics = {"library-events"}, partitions = 3)*

*@TestPropertySource(properties = {"spring.kafka.producer.bootstrap-servers=${spring.embedded.kafka.brokers}"*

*, "spring.kafka.consumer.bootstrap-servers=${spring.embedded.kafka.brokers}"})*

*public class LibraryEventsConsumerIntegrationTest {*

*SpringBootTest* For marking it as spring boot test

*EmbeddedKafka* For providing topics

*TestPropertySource* Overwrite the kafka producer and consumer config read from properties file with embedded kafka

Then for sending message embedded kafka broker and template is set up.

*@Autowired*

EmbeddedKafkaBroker embeddedKafkaBroker;

*@Autowired*

KafkaTemplate<Integer, String> kafkaTemplate;

The next thing is we are going to make sure the Kafka consumer is completely up and running before we go ahead and write the integration test.

We add KafkaListenerEndpointRegistry So this endpoint registry has a hold of all the listener containers.

*@Autowired*

KafkaListenerEndpointRegistry endpointRegistry;

And in the below method we check if the listener is already up or not.

*@BeforeEach*

*void* setUp() {