**Kafka**

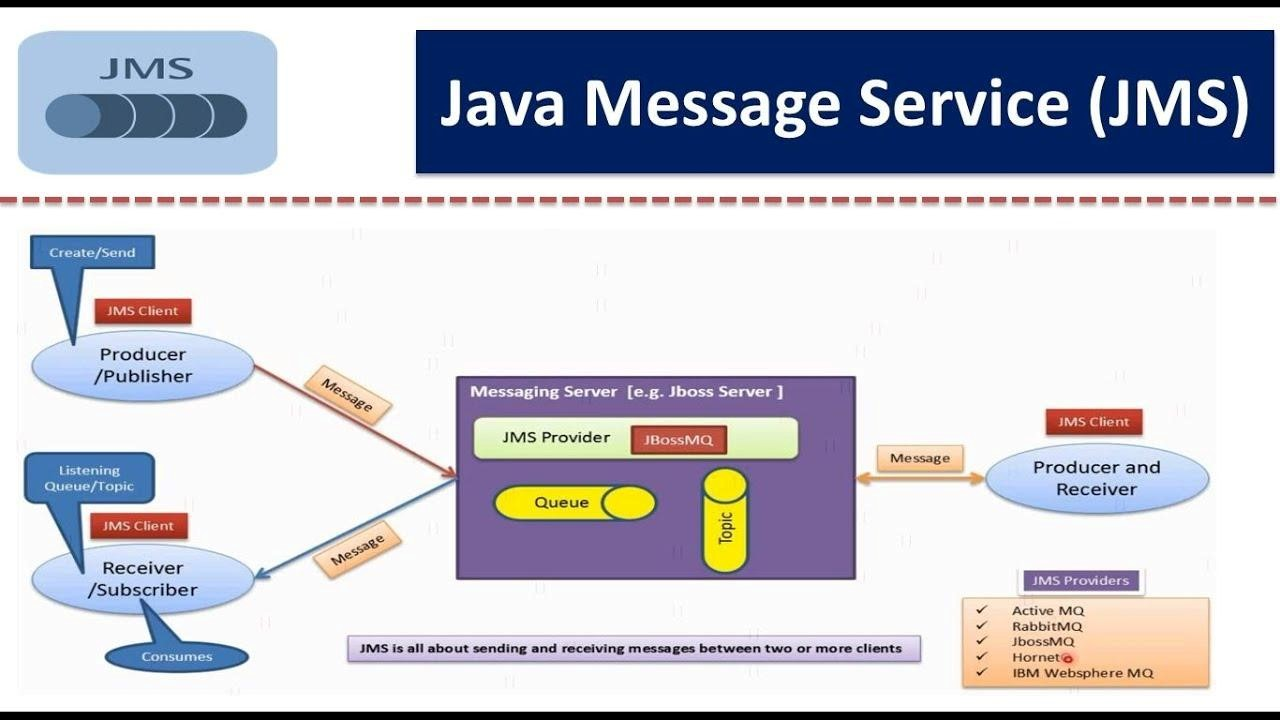
**What is JMS?**

JMS (Java Message Service) is an API that provides the facility to create, send and read messages. It provides loosely coupled, reliable and asynchronous communication.

JMS is also known as a **messaging service.** [**https://www.oracle.com/technical-resources/articles/java/**intro**-java-message-service.html**](https://www.oracle.com/technical-resources/articles/java/intro-java-message-service.html)

**Understanding Messaging**

* Messaging is a technique to communicate applications or software components.
* JMS is mainly used to send and receive message from one application to another.



**Implementation provided techniques for JMS**

1. SQS (Amazon)
2. ActiveMQ (Apache)
3. Weblogic Messaging (Oracle)
4. Websphere MQ (IBM)
5. RabbitMQ (Pivotal)

**What is Kafka?**

* Kafka is a **distributed streaming** platform. Using Kafka, applications can publish and subscribe to messages similar to JMS.
* Kafka allows for both the reliable transfer and transformation of data. Applications use Kafka to process streams of data in a scalable, fault tolerant manner.

**How does Kafka work?**

* Applications called producers **publish** messages to **topics**. Consumer applications **subscribe** to these topics. This creates a **pub/sub mode** similar to JMS.
* Kafka topics are partitioned across a cluster of servers. Consumers are divided into consumer groups. Multiple consumer groups can read from the same topic.
* Each consumer within a consumer group is responsible for reading from **specific partitions**. This guarantees the **order of messages** within a partition of a topic.
* Having groups of consumers collectively consume from topics improves **scalability** and **fault tolerance** when consuming data.
* This also decouples the consumption of messages from the production of messages. You can add as many consumers as you want to a Kafka cluster and it **won't impact performance**.

**JMS vs Kafka**

**Consuming messages**

* Unlike JMS, Kafka messages can be consumed multiple times by multiple consumers. While JMS allows for multiple consumers subscribing to the same topic, once message is delivered they are gone. With Kafka, these consumers can reread messages as topics retain data as a persisted log of messages (**for a configurable amount of time**).

**Scalability**

* A key advantage to **Kafka is scalability**, especially when applied to the pub/sub model. You can scale the processing in JMS queues by adding more consumers but that means consumers compete to read a single message. And while you can broadcast the same message to multiple subscribers it’s difficult to scale using JMS. Each message is delivered to each subscriber.
* Alternatively, Kafka uses **consumer groups** to subscribe to a given topic. This allows for easier scaling as more consumer instances can be added to a group.

**Performance**

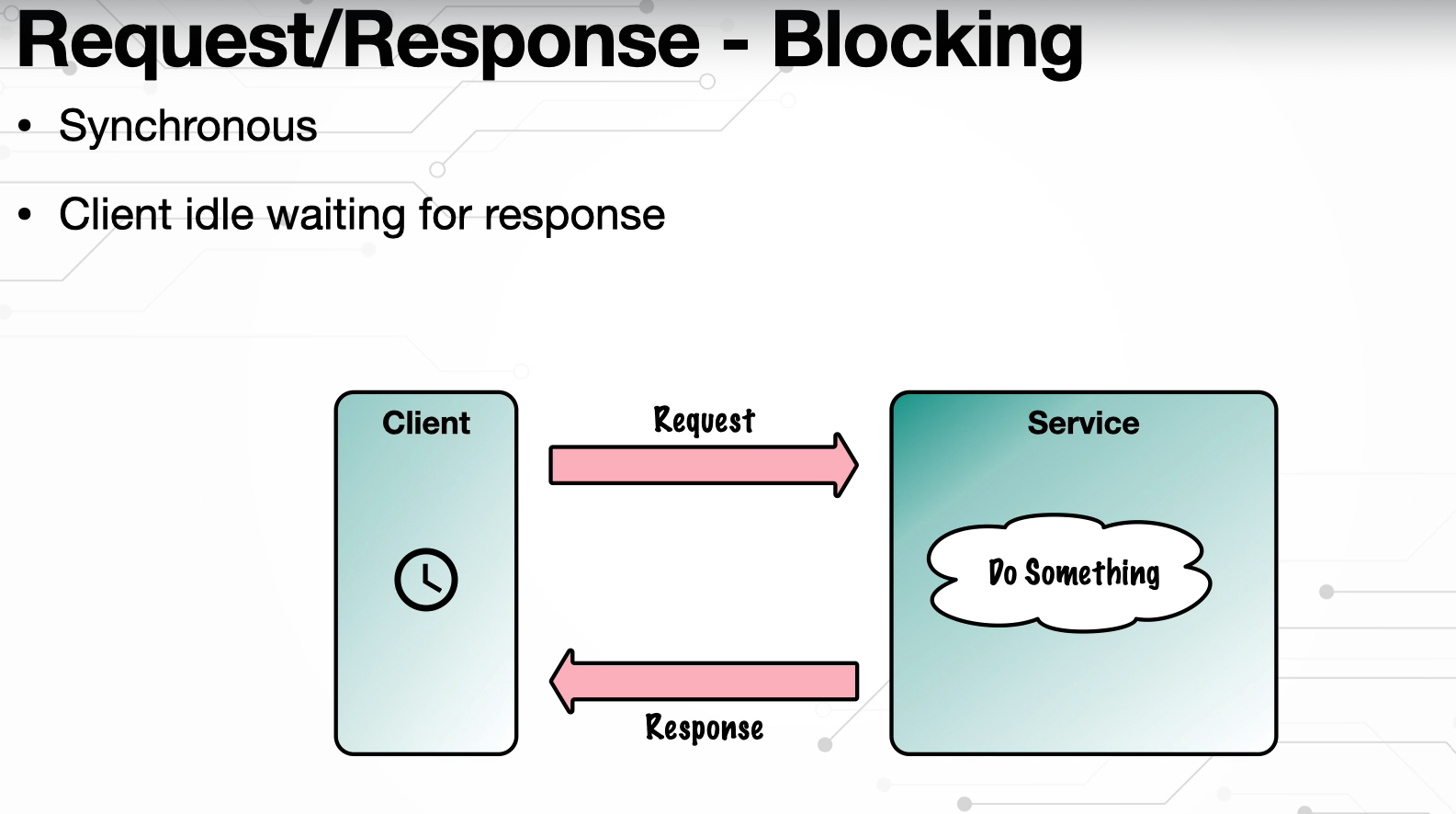
* Kafka has higher **throughput** than other messaging systems. This can be partially attributed to the techniques Kafka uses to read/write messages regarding memory and disk usage.
* But the major performance differences can be realized in how Kafka scales. Unlike JMS pub/sub model, you don’t have to add a new queue for each subscriber. **Instead the burden of reading messages is placed on consumer groups** and not the **message broker** themselves.
* This makes it possible to add an **infinite number of producers and consumers** as Kafka leverages a distributed system to handle **processes in parallel** these.

**Conclusion**

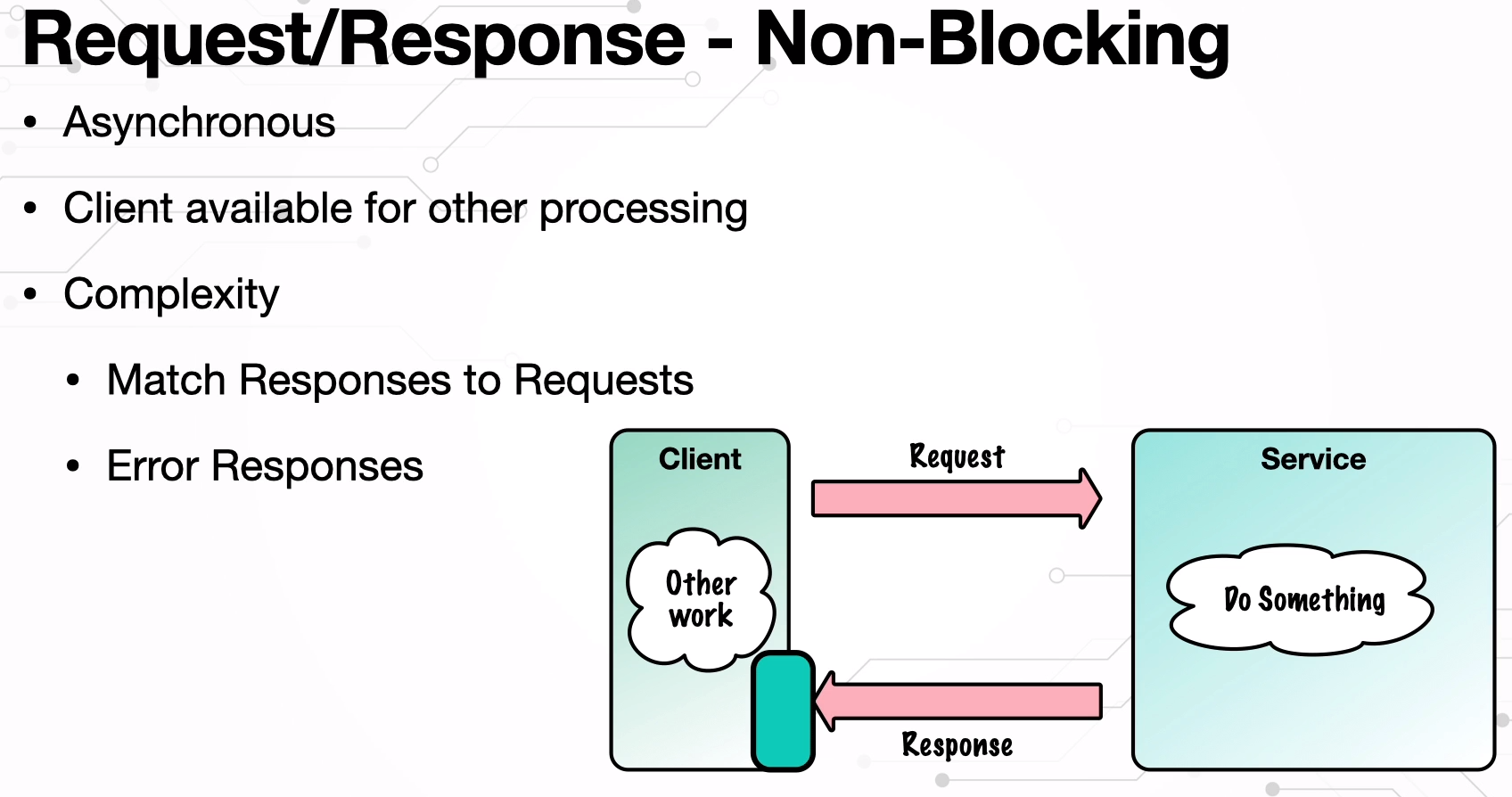
* Kafka solves several problems with traditional messaging. It scales the processing of messages in a pub/sub model where multiple consumers can read messages in parallel from a persisted log.
* **Kafka can do everything JMS** can but **JMS can’t do everything Kafka can**.
* For these reasons, **Kafka is becoming more and more popular as an enterprise** data streaming platform over more traditional messaging providers.

Reference Link: <https://www.stackchief.com/blog/JMS%20vs%20Kafka>

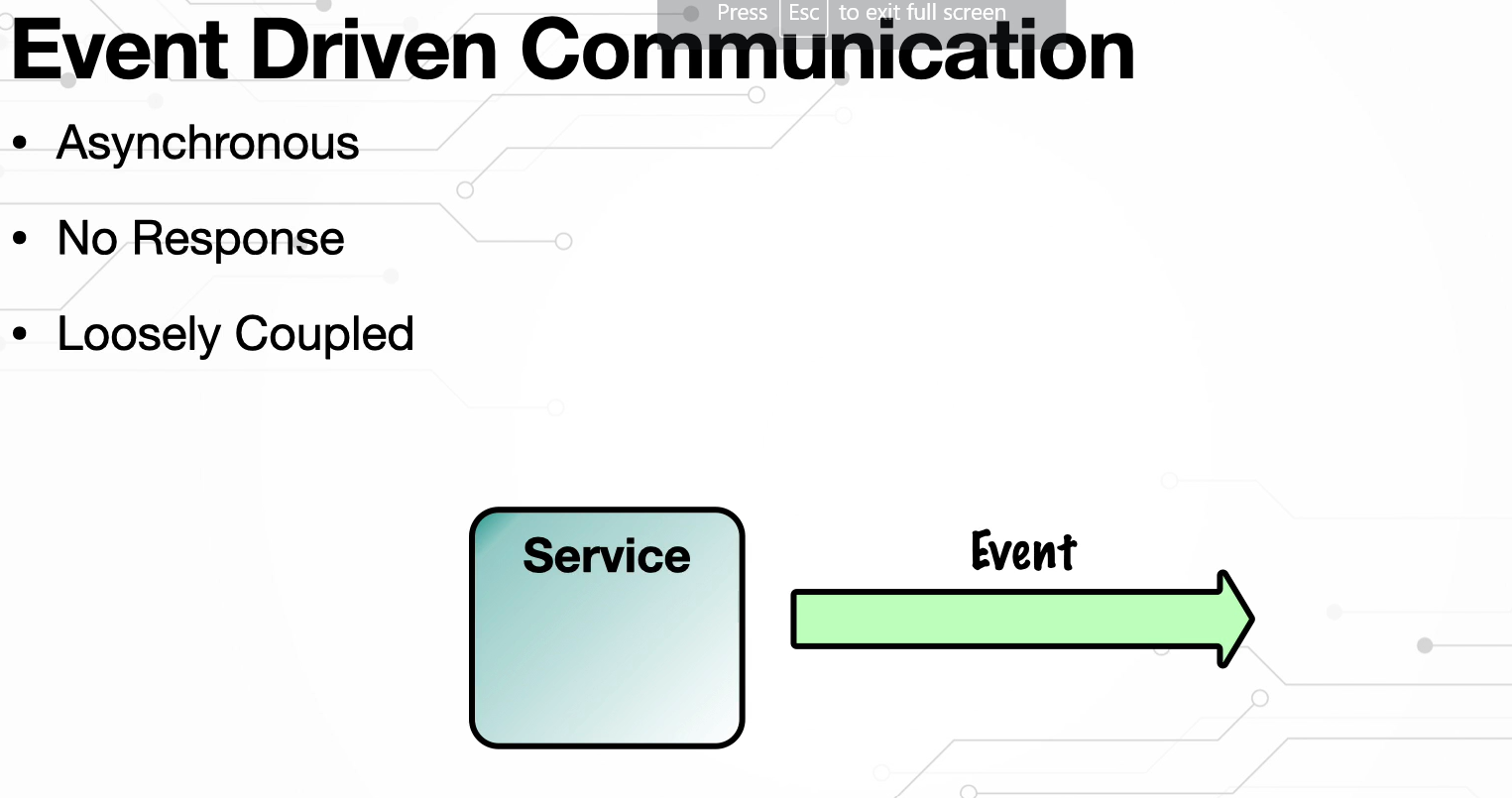
**Synchronous vs Asynchronous vs Event Driven Model:**

**Synchronous**: 

**Asynchronous:**



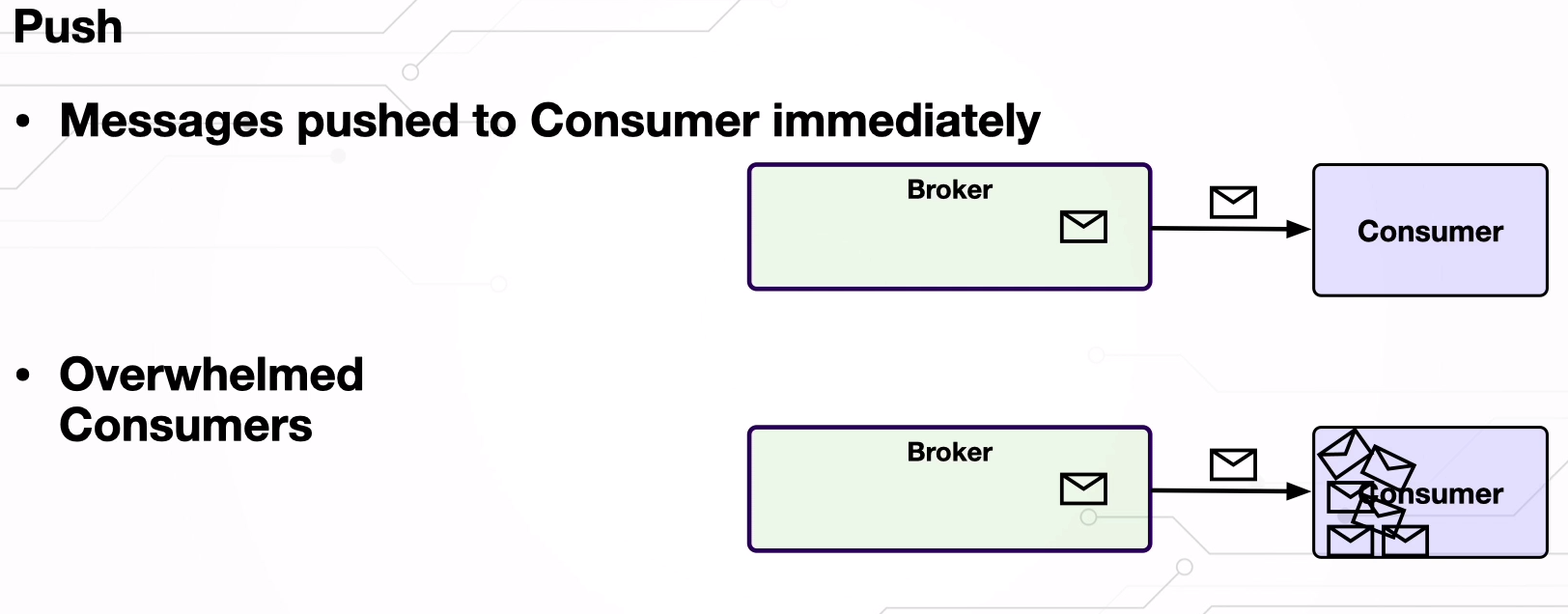
**Event Driven Model:**

****

**Push vs Pull Based Mechanism:**

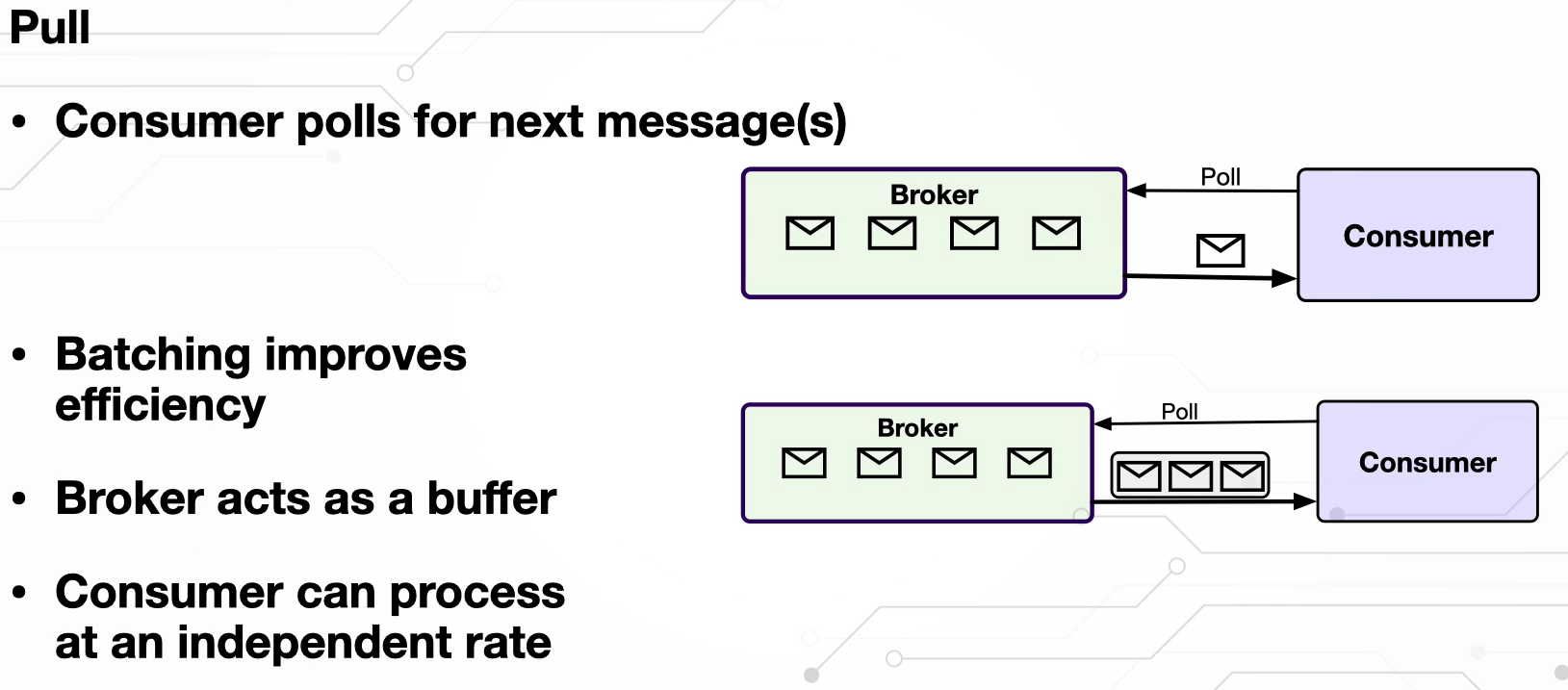
**Push model:**

* when a message sent by the producer arrives at the server, the server will immediately deliver it to the consumer
* Here JMS will push the event to target consumers, it doesn’t care about retains the messages.



**Pull model:**

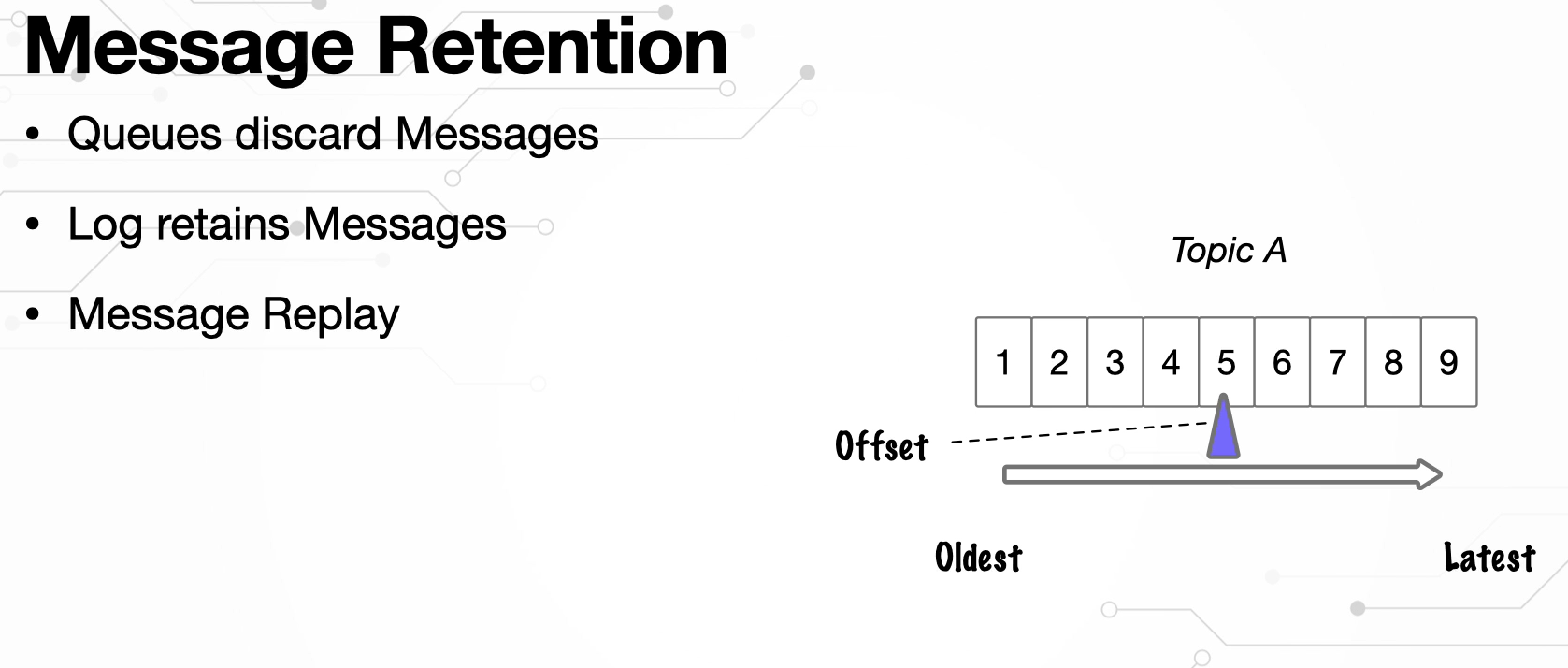
* The server does not process the received messages; instead, it only waits for the consumer to actively read them from it, that is, the consumer needs to "pull" messages.
* Ultimately, RabbitMQ is a message broker, while Kafka is a distributed streaming platform. One of the primary differences between the two is that Kafka is pull-based, while RabbitMQ is push-based. **A pull-based system waits for consumers to ask for data**.



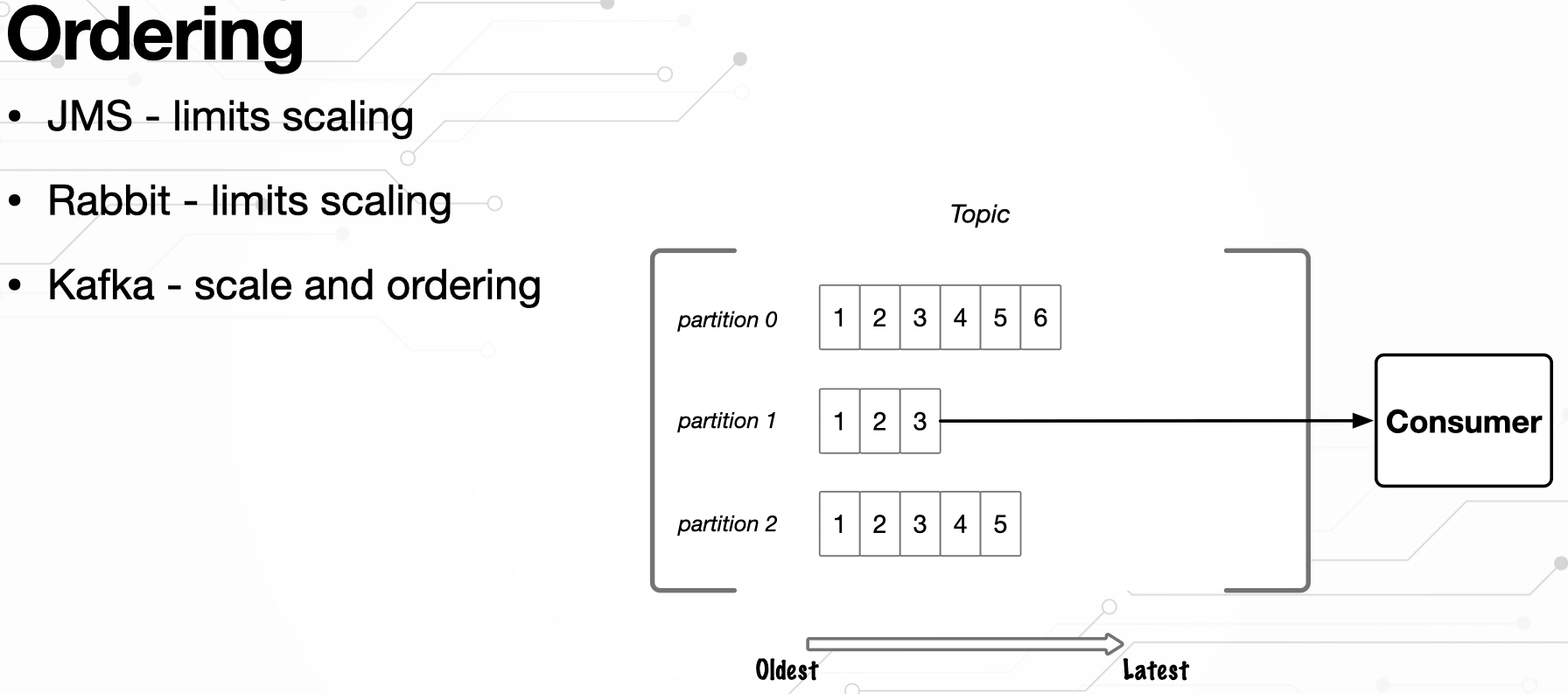
Brokers, Producers,Consumers

* Topics, Partitions, Logs, Offsets
* Keys
* Consumer Groups
* Clusters, Replication
* Leader Partitions,
* Follower Partitions

**Message Retention:**

****

**Ordering:**

* Kafka supports both scalability (consumers up and down) and ordering of the messages
* 

**Note:** JMS topics are different than the KAFKA TOPICS.

JMS -> Active MQ

Active MQ -> Rabbit MQ

Three components involved in communication in kafka:

1. **Brokers, Producers,Consumers:**

* **Producer** is source of message
* **Consumer** is responsible to pull the messages from broker store
* **Broker** is destination for messages and persist the data and point of contact for both producer and consumer.

1. **Topics:**

Message exchanges with the broker is done visa **topics**.

Producer send the data to the particular topic and it persist the data in store.

Consumer read the data from the same topic

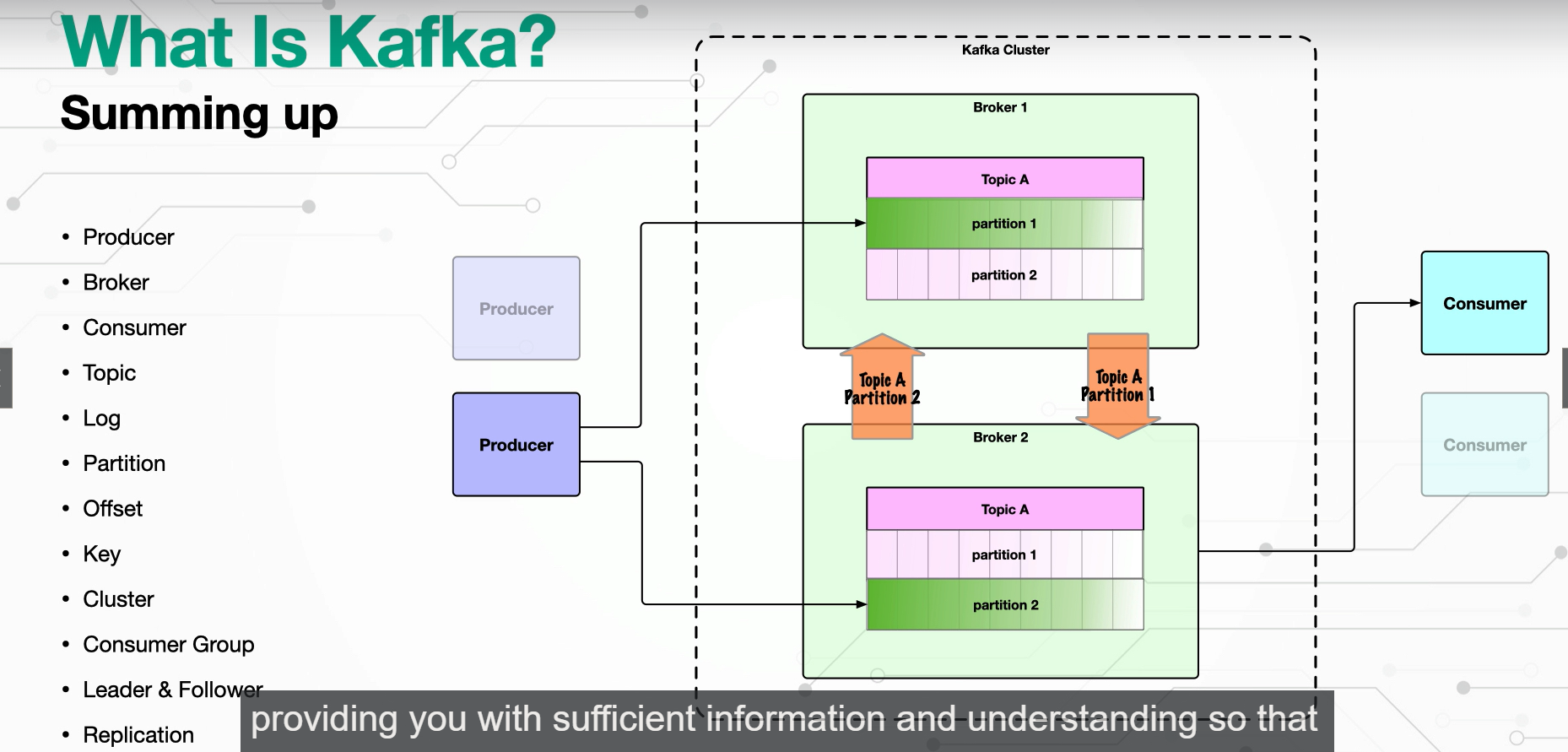
**So what is topic here?**

* Topic is not like a port or channel for which the message is sent, but it is actually a destination on broker.
* So physically topic is ordered sequence of events or messages which stores in durable way.
* It is s persistent record of events known in kafka parlance as log and log is essentially files

1. Log:

All events recorder in log files on disk

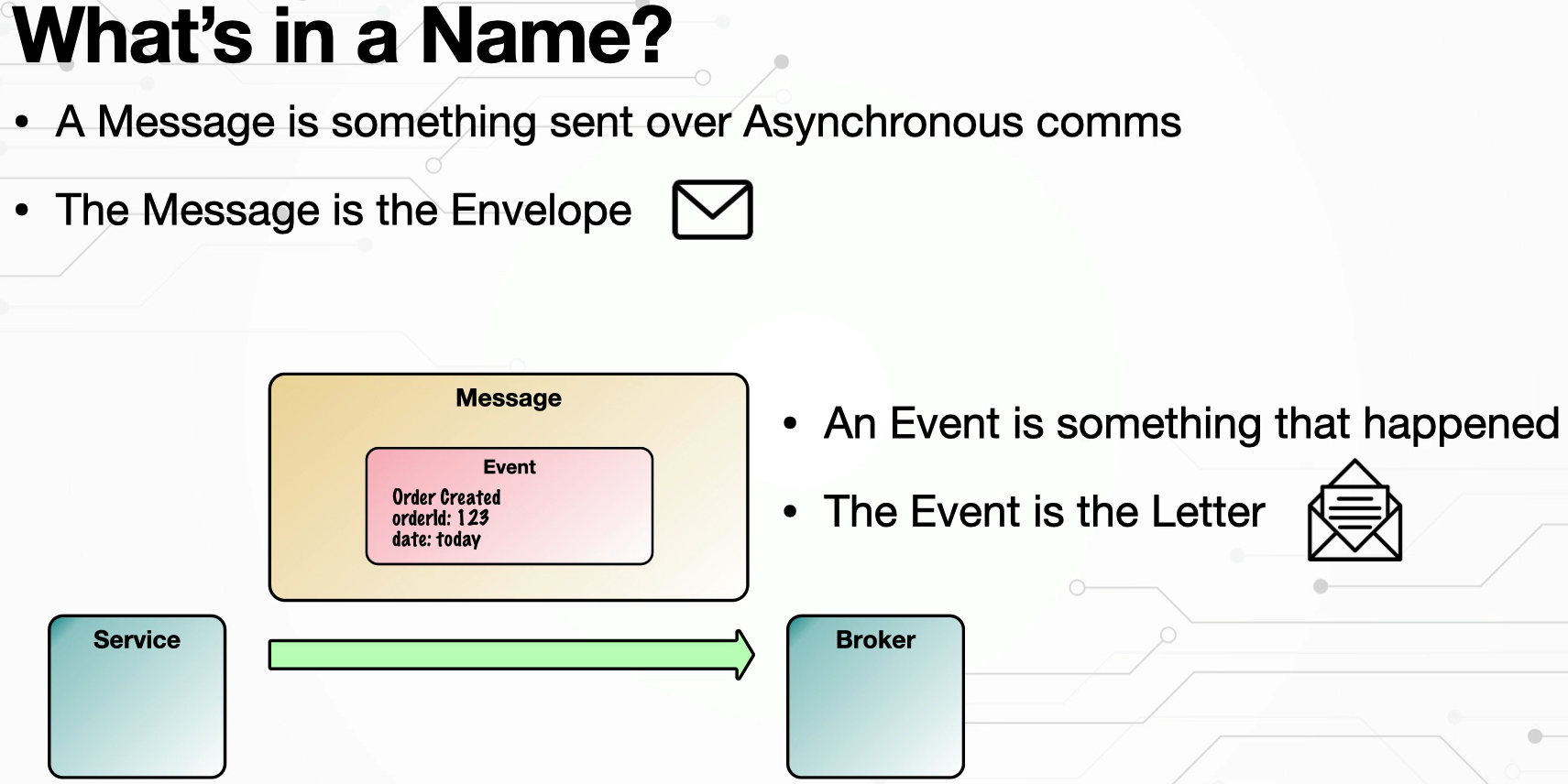
Log is also immutable which means once events are stored in topics then content will not/ cannot change.



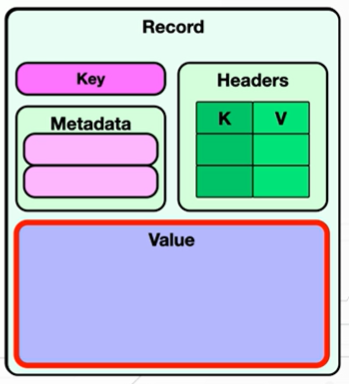
**Message and Events**

Message contains the meta data information about the event. It is like envelop

Event is the data, that contains message.



**Kafka Message/Record Structure:**

****

* **Value-Payload/content/data:**

It is event or content or data which as information

Max default size is 1 Mega Bite

* **Headers:**

It defines the meta data about events or pay load. It consists of key and value pair. It is similar to our HTTP header like we need to pass some algorithm to entry and decrypt, authentication mechanism while consuming the data in consumer side.

* **Key:**

Key is an optional value, by default you can send as null also. It will use round Robbin algorithm to choose topic when we pass null as key.

Key will be helpful to decide which partition to should go this message.

**Zookeeper and KRraft: Cluster Meta Data Management:**

* Zookeeper manages the number availability brokers, topics(add/remove) choose leadership topics
* To manage the Kafka cluster (cluster is combination of Kafka brokers)
* To work with Kafka, we need at least 3 zookeeper nodes to manage the Kafka. cluster and make them communicate with brokers.

**Limitations:**

* While growing your applications dynamically we might require more number rof instances to make more consistency. That will take huge processing effort form java side and networking side to maintain that many number zookeeper instances.

**To overcome above problem with zookeep , Kafka introduced a concept called Apache Kafaka Raft (KRaft)**

* Kraft will remove dependency(zookeeper) with external system to manage the metadata of the cluster.
* Apache Kafka Raft itself contains cluster to manage the clusters which contains brokers and topics.
* Kraft improves the scalability.

**Quiz Test(pending)**

**What is topic Offset:**

* The last read position in a topic partition for consumer.

**Installing and Running the Kafka:**

Download Apache Kafka - <https://kafka.apache.org/downloads>

Step1: wsl –install

Step2: set user name and password

Step3: install java on top of that and Exort JAVA\_HOME: path

Step4: create the cluster id

**Useful commands:**

* sudo wget https://archive.apache.org/dist/kafka/3.6.1/kafka\_2.13-3.6.1.tgz
* tar xvf kafka\_2.13-3.6.1.tgz
* mv kafka\_2.13-3.6.1 ~
* KAFKA\_CLUSTER\_ID="$(bin/kafka-storage.sh random-uuid)"
* export JAVA\_HOME="/usr/lib/jvm/java-11-openjdk-amd64"
* sudo(super user- admin persm)
* bin/kafka-storage.sh format -t $KAFKA\_CLUSTER\_ID -c config/kraft/server.properties

Formatting /tmp/kraft-combined-logs with metadata.version 3.6-IV2.

|  |
| --- |
| * **KAFKA\_CLUSTER\_ID="$(bin/kafka-storage.sh random-uuid)"** * **bin/kafka-storage.sh format -t $KAFKA\_CLUSTER\_ID -c config/kraft/server.properties** * **sudo bin/kafka-server-start.sh config/kraft/server.properties** |

Reference link: <https://www.conduktor.io/kafka/how-to-install-apache-kafka-on-windows/>

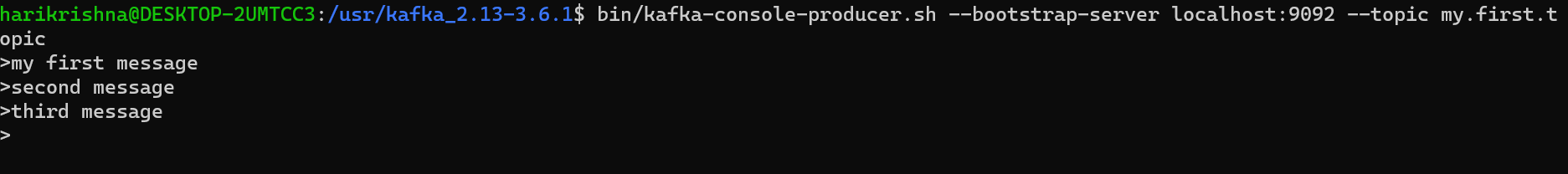
**Important Points:**

* We cannot run Kafka without zookeepr in wondows directly because all kafka feature don’t work properlyin wdindows operating system.
* We have use WSL(Windows Subsystem for Linux) to work with Kafka in windows.
* WSL will install ubutuntu linux environment on top our kernel and we have VM ready to run our kafka on top of it.

**Sending and Receiving:**

|  |
| --- |
| * sudo bin/kafka-server-start.sh config/kraft/server.properties * bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic my.first.topic * bin/kafka-console-producer.sh --bootstrap-server localhost:9092 --topic my.first.topic |

**Producer:**

****

**Consumer:**

****

**What is the need of bootstrat.server.sh:**

* Bootstrap means something it loads/start by itself.
* In kafka cluster , pod is replacing kafka broker(node).
* Bootstratp server in kafka acting like service to provide numbe rof available /alive nodes inside kafka cluster to the registered client. We can say bootstarp server is acting like kubernetes service for all kafka nodes.
* Bootstrap server is replacement of zookeeper for new versionfo Kraft kafa to monitor and manage the nodes(could be add or remove nodes, manage topics, elect leadership topics).
* If broker should work as hots/node then we should give boot strap server should give.
* Bootstratp server is one of the host machine in Kafka cluster.
* Broker/node is going to refer to bootstartp server.

Bootstratp server is mandatory for each broker and using this boostartp server we can call topics and partitions inside the broker.

* If we don’t create the topic or broker by default kafka create bootstrap server with broker(localhost:9092)

**CLI Tools Introductions:**

* **List topics**

bin/kafka-topics.sh --bootstartp-server localhost:9092 –list

* **Create topic**

bin/kafka-topics.sh –bootstrap-servre localhost:9092 –create –topic name –partitions 5

* **Alter Topic**

bin/kafka-topics.sh –bootstartp-server localhost –alter –topic my.new –partitions 7

**What is difference between Offset and Index:**

* **Offset** acts as unique identifier/integer member of a record with in that partition and also denote the reading position of the consumer in that partitioned.
* **Internally Offset refer the Index and consumers refer the offset**
* The Kafka offset is majorly deal with in two different types, like the current **offset and the committed offset.** It will also be further divided into different parts also. Kafka is using the **current offset to know the position of the Kafka consumer.** While doing the partition **rebalancing, the committed offset** plays an important role.
* The consumer offset in general are stored in the internal topic **\_\_consumer\_offsets.**
* **Consumer offset behaves like a bookmark for the consumer to start reading the messages from the point it left off.**
* They often work out to be the same, but conceptually they are different.
* An index is picking a particular numbered thing out of an array or other structure.
* An offset is starting at one point, moving by the offset, and seeing where you now are.
* Index: A data structure for efficiently mapping keys to record locations.
* Offset: A relative location measured in bytes, e.g. offsetof(type, member)
* Offset means start here from now on.
* Index means here's that thing you wanted.

**Explain Consumer Offsets in Kafka**

The consumer has three options to read messages from the Topic:

* **--from-beginnin**g — reading all the messages from the beginning.
* **--latest** — will read only those messages that will come after the consumer spun up.
* **--offset** — will read from a particular position and particular offset. This option can be done only programmatically.

e.g --offset 6 reads the message in a topic by passing a specific value from the consumer.

* The first two options --from-beginning and --latest can be explored using the consumer itself.
* Reference link: <https://sagarkudu.medium.com/explain-consumer-offsets-in-kafka-e2bce1b6c746>

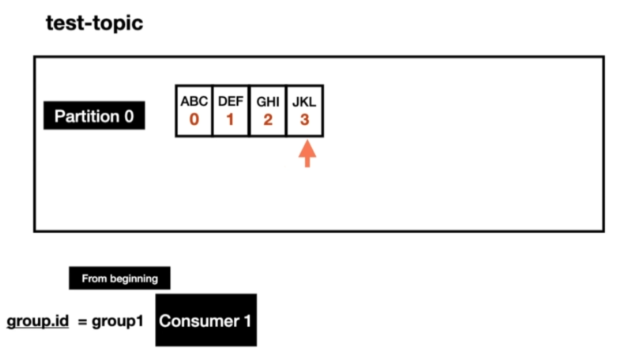
**Let’s Deep dive into Consumer Offset.**

* We have only 1 partition having some records in it. Let’s say we have a Consumer which is going to read the message from the beginning.
* For any Kafka consumer, it is required the consumer provide the group id. Now the consumer in general pulls and retrieves multiple records at the same time.

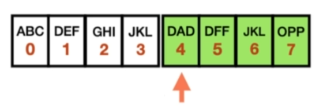
As it processes each message it moves the consumer read offset one by one starting from 0,1,2,3…etc.

Let say for some reason the consumer is crashed or shut down. While the consumer was down, the producer of the Topic produces some messages.

**Question:-** Again now the consumer is up after some time. So now how it knows that it needs to read from offset **4**?



**Answer:**



The consumer offset in general are stored in the internal topic \_\_**consumer\_offsets.**

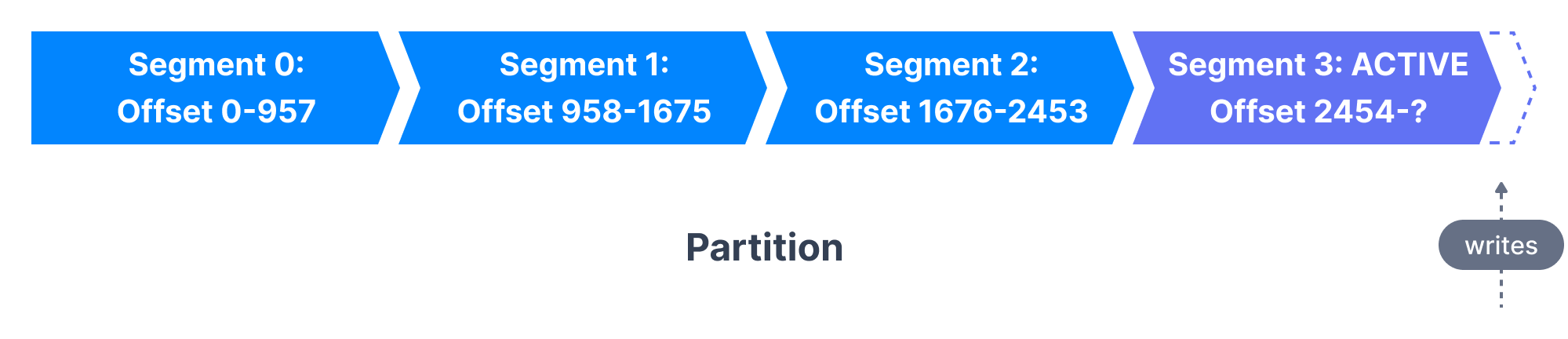
Consumer offset behaves like a bookmark for the consumer to start reading the messages from the point it left off.

**Practical**- Do we really have internal topics?

Please make sure that Zookeeper and Broker instances are running.

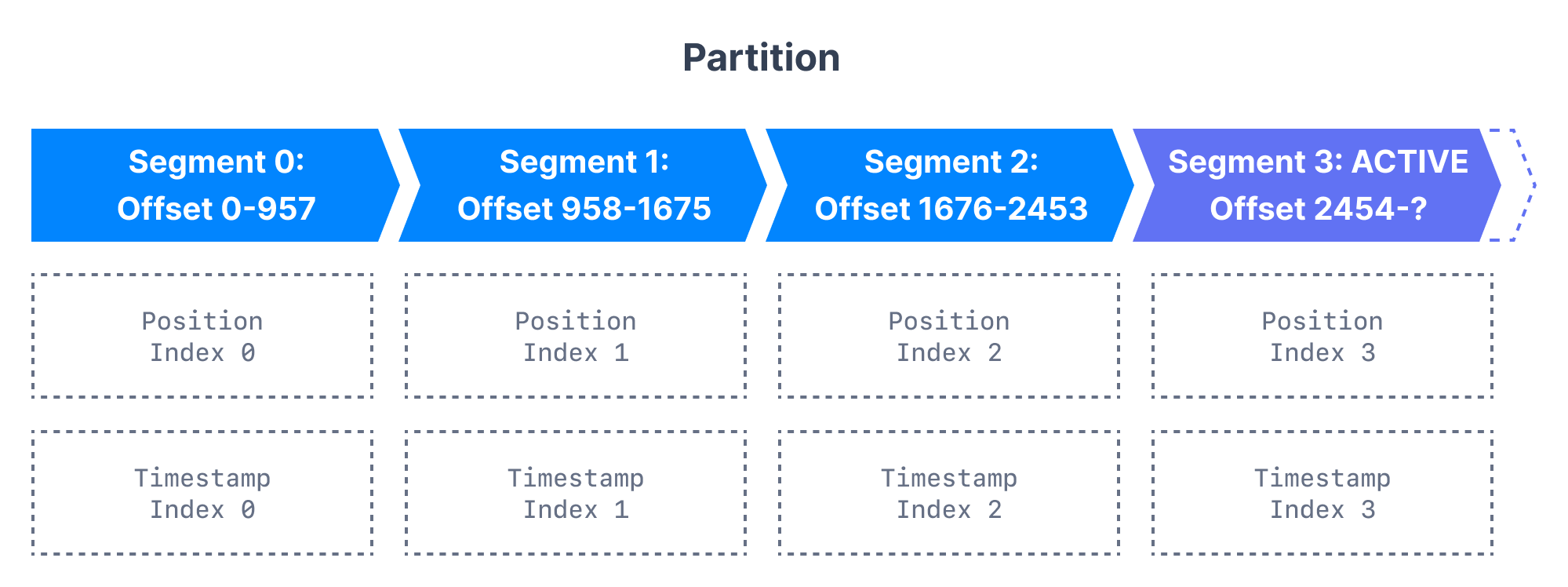
→ To check internal topic or consumer offsets we have a command, which is going to list all brokers or list of clusters available in your machine.

**Kafka Topic Internals:Kafka Topic Partitions and Segments:**

* The basic storage unit of Kafka is a partition replica. When you create a topic, Kafka first decides how to allocate the partitions between brokers. It spreads replicas evenly among brokers.
* Kafka brokers splits each partition into segments. Each segment is stored in a single data file on the disk attached to the broker. By default, each segment contains either 1 GB of data or a week of data, whichever limit is attained first. When the Kafka broker receives data for a partition, as the segment limit is reached, it will close the file and start a new one:
* 
* Only one segment is ACTIVE at any point in time - the one data is being written to. A segment can only be deleted if it has been closed beforehand. The size of a segment is controlled by two Broker configurations ([which can be modified at the topic level too](https://www.conduktor.io/kafka/kafka-topic-configuration-log-retention/))
* log.segment.bytes: the max size of a single segment in bytes (default 1 GB)
* log.segment.ms: the time Kafka will wait before committing the segment if not full (default 1 week)
* A Kafka broker keeps an open file handle to every segment in every partition - even inactive segments. This leads to a usually high number of open file handles, and the OS must be tuned accordingly.

**Kafka Topic Segments and Indexes:**

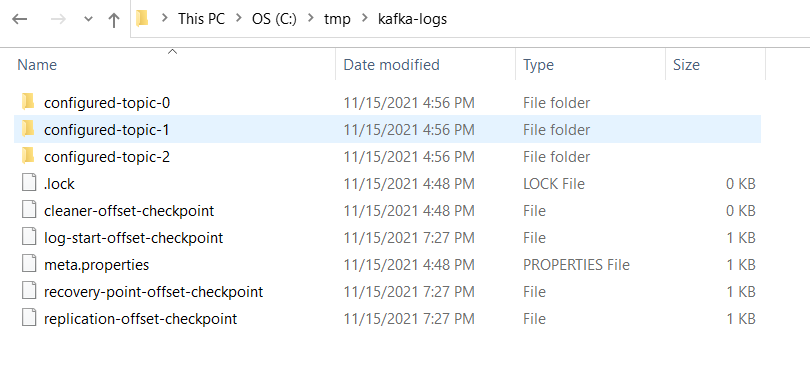
* Kafka allows consumers to start fetching messages from any available offset. In order to help brokers quickly locate the message for a given offset, Kafka maintains two indexes for each segment:
* An offset to position index - It helps Kafka know what part of a segment to read to find a message
* A timestamp to offset index - It allows Kafka to find messages with a specific timestamp



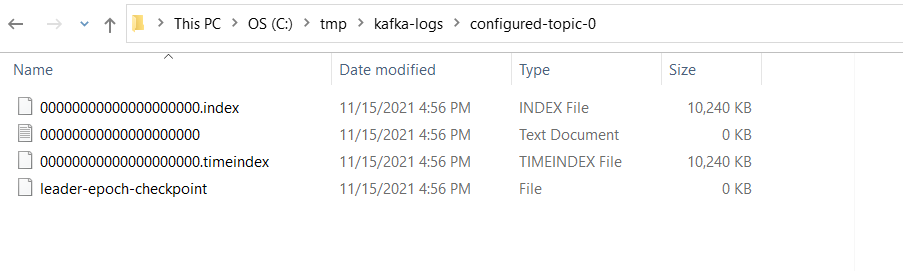
**Inspecting the Kafka Directory Structure:**

Kafka stores all of its data in a directory on the broker disk. This directory is specified using the property log.dirs in the broker's configuration file. For example,

|  |
| --- |
| # A comma separated list of directories under which to store log files  **log.dirs=/tmp/kafka-logs** |

Explore the directory and notice that there is a folder for each topic partition. All the segments of the partition are located inside the partition directory. Here, the topic named configured-topic has three partitions, each having one directory - configured-topic-0, configured-topic-1 and configured-topic-2.

Descend into a directory for a topic partition. Notice the indexes - time and offset for the segment and the segment file itself where the messages are stored.

**Considerations for Segment Configurations****:**

* Let us review the configurations for segments and learn their importance.
* log.segment.bytes As messages are produced to the Kafka broker, they are appended to the current segment for the partition. Once the segment has reached the size specified by the log.segment.bytes parameter, which defaults to 1 GB, the segment is closed and a new one is opened.
* A smaller segment size means that files must be closed and allocated more often, which reduces the overall efficiency of disk writes.
* Once a segment has been closed, it can be considered for expiration. Adjusting the size of the segments can be important if topics have a low produce rate. Having a small segment size would mean Kafka has to keep a lot of files open which may lead to Too many open files error.
* log.segment.ms Another way to control when segments are closed is by using the log.segment.ms parameter, which specifies the amount of time after which a segment should be closed. The default is 1 week. Kafka will close a segment either when the size limit is reached or when the time limit is reached, whichever comes first.
* When using a time-based segment limit, it is important to consider the impact on disk performance when multiple segments are closed simultaneously.

FYI Article link:

* <https://www.conduktor.io/kafka/kafka-topics-internals-segments-and-indexes/>
* <https://codingharbour.com/apache-kafka/the-introduction-to-kafka-topics-and-partitions/#:~:text=Messages%20sent%20by%20a%20producer,this%20in%20the%20next%20section>)

**What are indexes in Kafka?**

Kafka allows consumers to start fetching messages from any available offset. This means that if a consumer asks for 1 MB messages starting at offset 100, the broker must be able to quickly locate the message for offset 100 (which can be in any of the

segments for the partition) and start reading the messages from that offset on. In order to help brokers quickly locate the message for a given offset, Kafka maintains an index for each partition. The index maps offsets to segment files and positions within the file.

**Where does consumer offsets stores?**

The consumer offsets are recorded in Kafka itself by the consumers in an internal topic called **\_\_consumer\_offsets**. The way the consumer offsets are recorded is an incremental number starting from **0**.

**what is a Consumer Group?**

* When sending messages in a distributed setup using a messaging system, you typically have two scenarios you want to achieve.

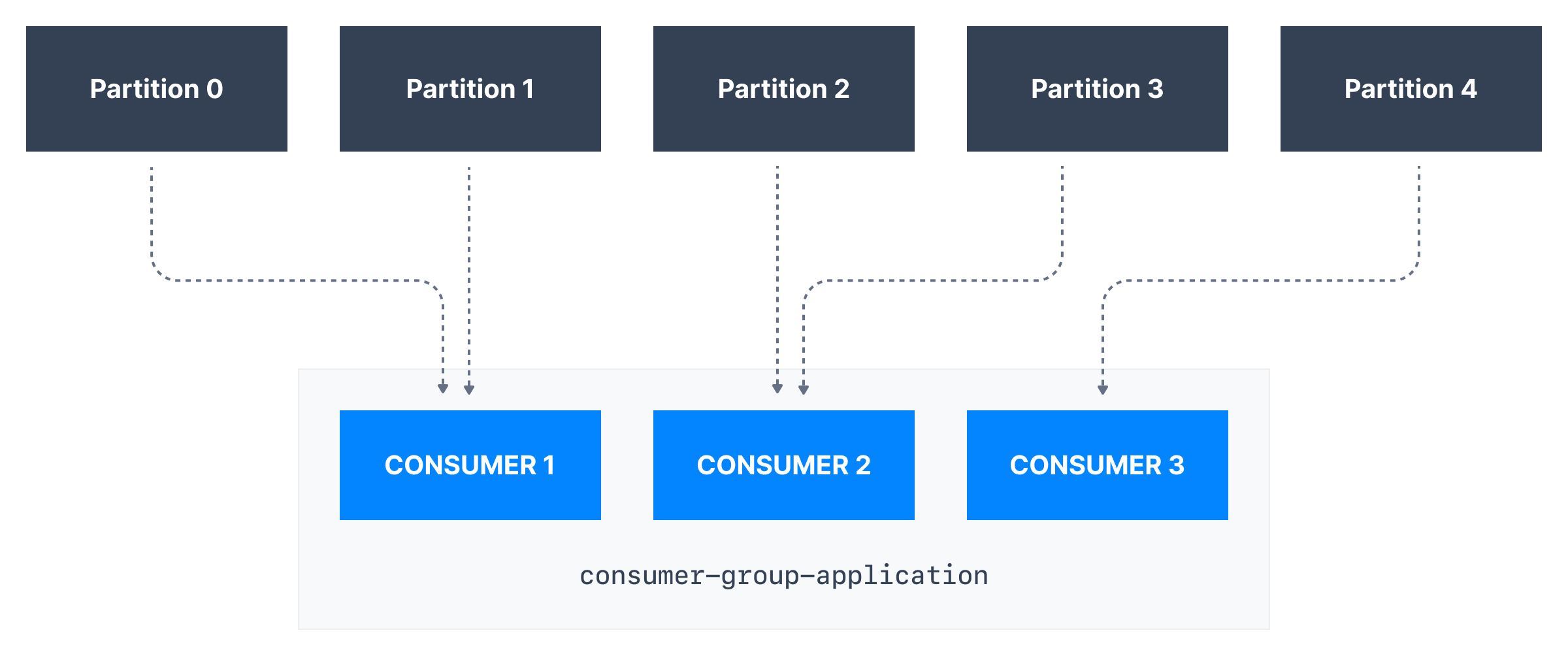
**Either you want to:**

* send a message to a targeted group of consumers (which might be just one consumer) or broadcast the message to all the consumers
* Kafka allows you to achieve both of these scenarios by using consumer groups.

**Consumer group**

* A consumer group is a group of consumers (I guess you didn’t see this coming?) that share the same group id. When a topic is consumed by consumers in the same group, every record will be delivered to only one consumer. As the official documentation states: “If all the consumer instances have the same consumer group, then the records will effectively be load-balanced over the consumer instances.”

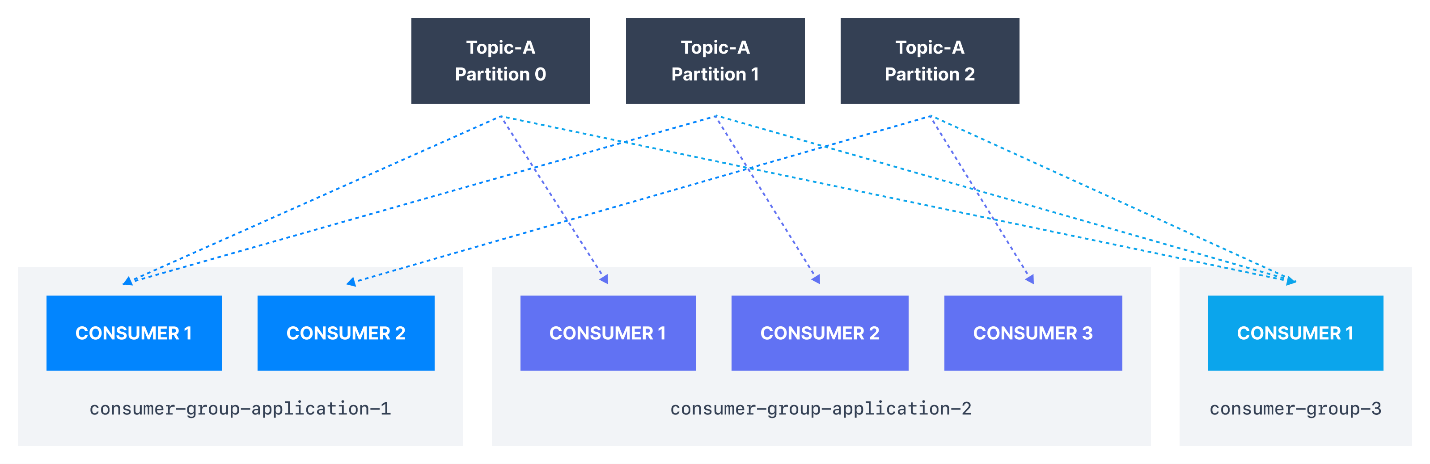
* This way you can ensure parallel processing of records from a topic and be sure that your consumers won’t be stepping on each other toes.



**Example:**

In the example above, Consumer 1 of consumer group consumer-group-application-1 has been assigned Partition 0 and Partition 1, whereas Consumer 2 is assigned Partition 2 and Partition 3, and finally Consumer 3 is assigned Partition 4. Only Consumer 1 receives messages from Partition 0 and Partition 1, while only consumer Consumer 2 receives messages from Partition 2 and 3, and only Consumer 3 receives messages from Partition 4.

Each of your applications (that may be composed of many consumers) reading from Kafka topics must specify a different **group.id.** That means that multiple applications (consumer groups) can consume from the same topic at the same time:



If there are more consumers than the number of partitions of a topic, then some consumers will remain inactive as shown below. Usually, we have as many consumers in a consumer group as the number of partitions. If we want more consumers for higher throughput, we should create more partitions while creating the topic. Otherwise, some of the consumers may remain inactive.



FYI: <https://www.conduktor.io/kafka/kafka-consumer-groups-and-consumer-offsets/>

* Each consumer is task which will make our application as parallisam.

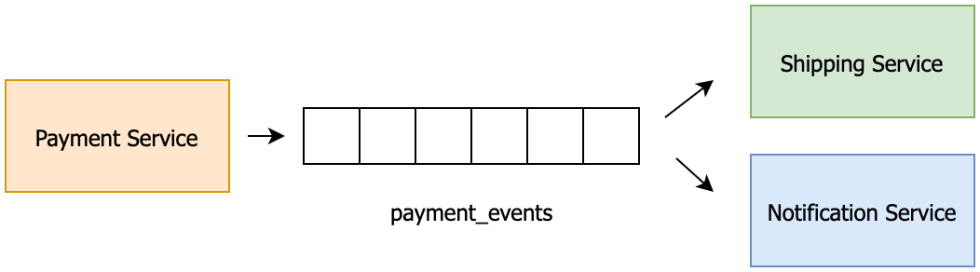
**Why do we need consumer groups?**

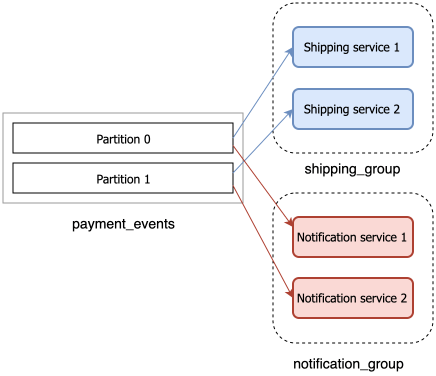
* Consumers that are part of the same application and therefore performing the same "logical job" can be grouped together as a Kafka consumer group.
* A topic usually consists of many partitions. These partitions are a unit of parallelism for Kafka consumers.
* The benefit of leveraging a Kafka consumer group is that the consumers within the group will coordinate to split the work of reading from different partitions.

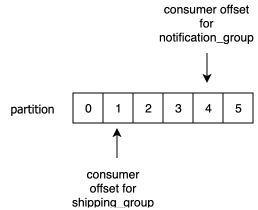
FYI: <https://medium.com/javarevisited/kafka-partitions-and-consumer-groups-in-6-mins-9e0e336c6c00>

**Microservice communication using kafka:**

Reference link : <https://codingharbour.com/apache-kafka/what-is-a-consumer-group-in-kafka/>



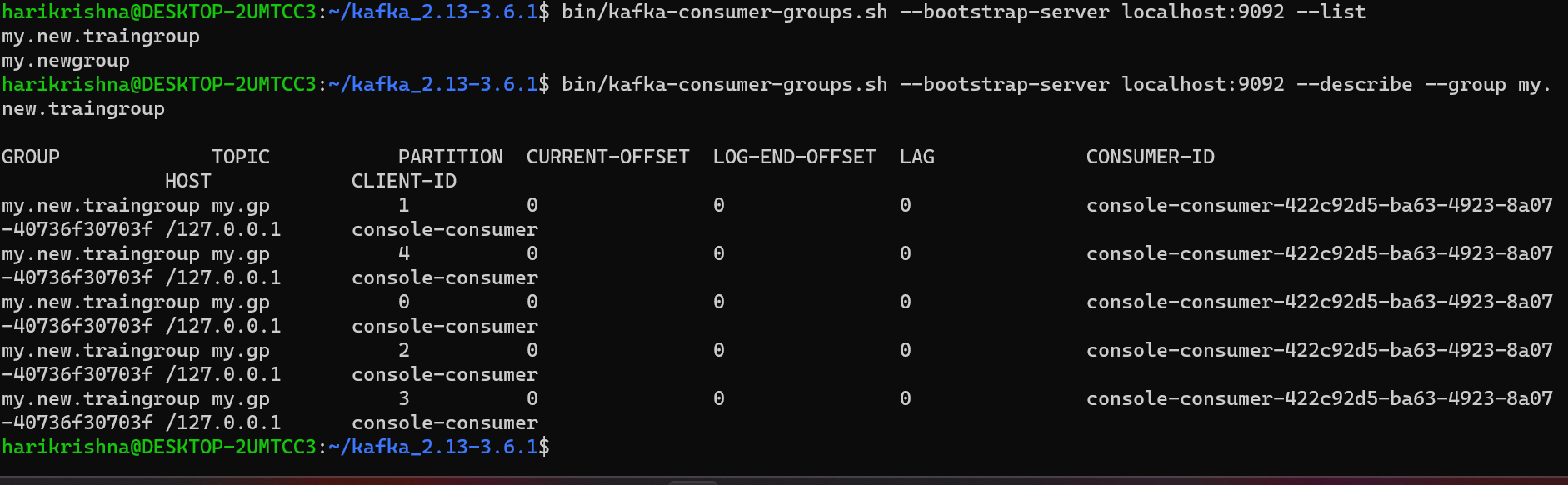


* A consumer group and record offset
* If you remember when we talked about [topics](https://codingharbour.com/apache-kafka/the-introduction-to-kafka-topics-and-partitions/), we said that each record is uniquely identified by an offset in the partition. These offsets are used to track which record has been consumed by which consumer group.
* Kafka employs an approach of ‘a dumb pipeline, smart clients’ meaning that Kafka brokers don’t know anything about consumer offsets. The consumers themselves are in charge of tracking which records have been consumed. Once the consumer reads the record it will store this offset in a special Kafka topic called ***\_\_consumer\_offsets*** (yes, those are two underscores at the beginning). When a consumer stores the offset in this topic we’re saying that it’s **committing the offset**.
* This enables consumers to always know which record should be consumed next from a given partition. Since the consumer offset is stored in Kafka, it means that the position of the consumer group is maintained even after restarts.
* An example of consumer offsets

You do not explicitly create consumer groups but rather build consumers which always belong to a consumer group. No matter which technology (Spark, Spring, Flink, ...) you are using, each Kafka Consumer will have a Consumer Group. The consumer group is configurable for each individual consumer.

**Command to create consumer group:**

* bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic my.gp --group my.new.traingroup



**Running Kraft kafka in docker:Using Image1 : bitnami – docker-compose.yaml** – docker compose up

|  |
| --- |
| version: "3"  services:    kafka:      image: bitnami/kafka      container\_name: kafka      ports:        - 9092:9092      environment:        - KAFKA\_ENABLE\_KRAFT=yes        - KAFKA\_CFG\_PROCESS\_ROLES=broker,controller        - KAFKA\_CFG\_CONTROLLER\_LISTENER\_NAMES=CONTROLLER        - KAFKA\_CFG\_LISTENERS=PLAINTEXT://:9092,CONTROLLER://:2181        - KAFKA\_CFG\_LISTENER\_SECURITY\_PROTOCOL\_MAP=CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT        - KAFKA\_CFG\_ADVERTISED\_LISTENERS=PLAINTEXT://127.0.0.1:9092        - KAFKA\_BROKER\_ID=1        - KAFKA\_CFG\_CONTROLLER\_QUORUM\_VOTERS=1@127.0.0.1:2181        - ALLOW\_PLAINTEXT\_LISTENER=yes        - KAFKA\_CFG\_NODE\_ID=1        - KAFKA\_KRAFT\_CLUSTER\_ID=MkU3OEVBNTcwNTJENDM2Qk      volumes:        - ./kafka:/bitnami/kafka |

**Uisng Image2: confluent**

|  |
| --- |
| version: '3'  services:    kafka1:      image: confluentinc/cp-kafka      container\_name: kafka1      hostname: kafka1      ports:        - "9092:9092"      environment:        KAFKA\_NODE\_ID: 1        KAFKA\_CONTROLLER\_LISTENER\_NAMES: 'CONTROLLER'        KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: 'CONTROLLER:PLAINTEXT,INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT'        KAFKA\_LISTENERS: 'INTERNAL://kafka1:29092,CONTROLLER://kafka1:29093,EXTERNAL://0.0.0.0:9092'        KAFKA\_ADVERTISED\_LISTENERS: 'INTERNAL://kafka1:29092,EXTERNAL://localhost:9092'        KAFKA\_INTER\_BROKER\_LISTENER\_NAME: 'INTERNAL'        KAFKA\_CONTROLLER\_QUORUM\_VOTERS: '1@kafka1:29093'        KAFKA\_PROCESS\_ROLES: 'broker,controller'        KAFKA\_GROUP\_INITIAL\_REBALANCE\_DELAY\_MS: 0        KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1        KAFKA\_TRANSACTION\_STATE\_LOG\_REPLICATION\_FACTOR: 1        CLUSTER\_ID: 'ciWo7IWazngRchmPES6q5A=='        KAFKA\_LOG\_DIRS: '/tmp/kraft-combined-logs' |

**Connecting docker kraft from spring boot application :**

* **kafka listener properties :**

|  |
| --- |
| * *@Component* * public class TrainTrackingHandler { * *@KafkaListener*(id = "paytmConsumerClient", topics = "17235", groupId = "paytm.consumer.group") * public void listner(String payload) { * System.***out***.println("Message Received " + payload); * } * } |

* **applciations yaml properties**

|  |
| --- |
| * spring: * kafka: * consumer: * enable-auto-commit: true * bootstrap-servers: localhost:9092 * group-id: paytm.consumer.group * auto-offset-reset: earliest * key-deserializer: org.apache.kafka.common.serialization.StringDeserializer * value-deserializer: org.apache.kafka.common.serialization.StringDeserializer |

* **Git hub repo link:**<https://github.com/HarikrishnaThopugunta/paytm-microservice>

**For StringSerializer we need to use below property in consumer end:**

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

**For JSONDeserilizer w eneed to use below property in consumer end:**

|  |
| --- |
| spring:    kafka:      consumer:        enable-auto-commit: true        bootstrap-servers: localhost:9092        group-id: makemytrip.consumer.group        auto-offset-reset: earliest        key-deserializer: org.springframework.kafka.support.serializer.JsonDeserializer        value-deserializer: org.springframework.kafka.support.serializer.JsonDeserializer        properties.spring.json.value.default.type: com.example.demo.entity.TrakingEntity        properties.spring.json.trusted.packages: com.example.demo |

**We need add JACKSON API in pom.xml file:**

|  |
| --- |
| <dependency>        <groupId>com.fasterxml.jackson.core</groupId>        <artifactId>jackson-core</artifactId>        <version>2.15.2</version>      </dependency> |

**Deserializer Error handling:**

**Kafka lister will continuously listen to its own kafka consumer group until it found matched payload object ias per it configurations.**

**For example we have configured for kafka lister like below**

*@KafkaListener*(id = "paytmConsumerClient", topics = "17235", groupId = "paytm.consumer.group")

public void listner(TranckingEntity payload) {

System.***out***.println("Message Received " + payload);

}

Our payload object form kafka topic shoulde match with TrakcingEntity class fileds anems and it’s data types to deserialize them.

|  |
| --- |
| {  "trainName":"Narsapur Express",  "trainNumber":17235,  "locationName":"GNT",  "latitude":"23238239",  "langitude":"538379583"  } |

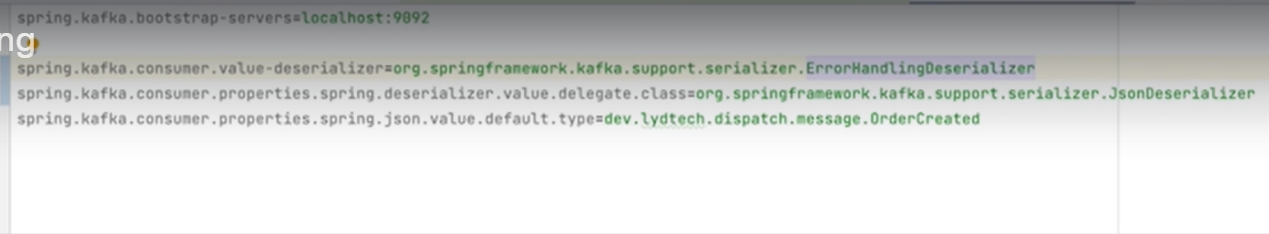
* If it doesn’t match the payload from kafka topic , then listener will throw the exception continuously to the spring poll(), this is called poison pills.
* Spring has provided concept of Error HandlerDesrializer to handle these kind of issues, to handle the exception only once and it can set to wait for next event.

Just extending **ErrorHandlingDeserializer** will solve this infinite deserialization exception problem. Note that, it will not solve the deserialization problem, but it will solve the infinite error logs shown in the application console/log file. For my case, to fix the deserialization problem I have to change code of `deserialize` method in `KafkaMessageDeserializer`.

|  |
| --- |
| properties:         key.deserializer: org.springframework.kafka.support.serializer.ErrorHandlingDeserializer         value.deserializer: org.springframework.kafka.support.serializer.ErrorHandlingDeserializer         spring.deserializer.key.delegate.class: org.apache.kafka.common.serialization.StringDeserializer |

**Reference Link:**

<https://www.linkedin.com/pulse/handle-infinite-deserialization-exception-kafka-spring-rana>



**Difference between @Resource and @Inject and @Autowired:**

**@Resource**

The @Resource annotation is part of the JSR-250 annotation collection, and is packaged with Jakarta EE. This annotation has the following execution paths, listed by precedence:

* Match by Name
* Match by Type
* Match by Qualifier

**@Inject**

The @Inject annotation belongs to the JSR-330 annotations collection. This annotation has the following execution paths, listed by precedence:

* Match by Type
* Match by Qualifier
* Match by Name

These execution paths are applicable to both setter and field injection. In order for us to access the @Inject annotation, we have to declare the javax.inject library as a Gradle or Maven dependency.

|  |
| --- |
| testCompile group: 'javax.inject', name: 'javax.inject', version: '1' |

|  |
| --- |
| <dependency>  <groupId>javax.inject</groupId>  <artifactId>javax.inject</artifactId>  <version>1</version>  </dependency> |

**@Autowired**

The behaviour of the @Autowired annotation is similar to the @Inject annotation. The only difference is that the @Autowired annotation is part of the Spring framework. This annotation has the same execution paths as the @Inject annotation, listed in order of precedence:

* Match by Type
* Match by Qualifier
* Match by Name

These execution paths are applicable to both setter and field injection.

There are four modes of autowiring a bean using an XML configuration:

**no**: the default value – this means no autowiring is used for the bean and we have to explicitly name the dependencies.

**byName**: autowiring is done based on the name of the property, therefore Spring will look for a bean with the same name as the property that needs to be set.

**byType**: similar to the byName autowiring, only based on the type of the property. This means Spring will look for a bean with the same type of the property to set. If there’s more than one bean of that type, the framework throws an exception.

**constructor**: autowiring is done based on constructor arguments, meaning Spring will look for beans with the same type as the constructor arguments.

Reference Links:

* <https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring>
* <https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring>

**@Configuration**: (class level annotation)

The @Configuration annotation indicates that the class is a source of bean definitions. We can also add it to multiple configuration classes.

**@Bean:** (method level annotation)

@Bean methods can have arguments, which can be used to inject dependencies into the bean instance. This is useful when you need to customize the behavior of a bean based on its dependencies.

We use the @Bean annotation on a method to define a bean. **If we don’t specify a custom name, then the bean name will default to the method name.**

For a bean with the default **singleton scope**, Spring first checks if a **cached instance** of the bean already exists, and only creates a new one if it doesn’t. If we’re using the prototype scope, the container returns a new bean instance for each method call. Another way to create the configuration of the beans is through XML configuration:

**@Component:**

The @Component annotation is a stereotype annotation that is used to mark a class as a component. A component is a general-purpose object that is managed by the Spring IoC container. Other stereotype annotations, such as @Service, @Repository, and @Controller, are also available and provide additional semantic information.

**Question 1:**

What if we don’t declare @Configuration and declared only @Bean annotations in a Java class and run the spring applications?

Is bean objects will get created and given to IOC?

Is Spring will visibility to invoke these @Bean annotated method with @Configuration?

**Question 2:**

What if we don’t declare @Bean and declared only @Configuration annotations in a Java class and run the spring applications?

Did Spring IOC will give any warning to me? saying like there is no @Bean annotated methods inside class, and no use of declaring @Configuration at class level?

**Quetsion3:**

Can we declare @Bean annotation for method, where the class is declared as @Component:

Yes, but

**Explanation: Full @Configuration vs “lite” @Bean mode:**

**“lite” @Bean mode**

* When @Bean methods are declared within classes that are not annotated with @Configuration, they are referred to as being processed in a **“lite” mode**.
* Bean methods declared on a bean that is not annotated with @Configuration are considered to be “lite”, with a different primary purpose of the containing class and a **@Bean** method being a sort of bonus there.
* For example, service components may expose management views to the container through an additional @Bean method on each applicable component class. In such scenarios, @Bean methods are a general-purpose factory method mechanism.

**Full @Configuration**

* Unlike **full @Configuration, lite @Bean** methods cannot declare **inter-bean dependencies**. Instead, they operate on their containing component’s internal state and, optionally, on arguments that they may declare.
* Such a @Bean method should therefore not invoke other @Bean methods.
* Each such method is literally only a factory method for a particular bean reference, without any special runtime semantics.
* The positive side-effect here is that no CGLIB subclassing has to be applied at runtime, so there are no limitations in terms of class design (that is, the containing class may be final and so forth).
* In common scenarios, @Bean methods are to be declared within @Configuration classes, ensuring that **“full”** mode is always used and that cross-method references therefore get redirected to the container’s lifecycle management.
* This prevents the same @Bean method from accidentally being invoked through a regular Java call, which helps to reduce subtle bugs that can be hard to track down when operating **in “lite”** mode.

Reference link:

* <https://docs.spring.io/spring-framework/reference/core/beans/java/basic-concepts.html>

**How many ways we can give spring bean information to IOC** :

* **XML Based Configuration** - By creating Spring Configuration XML file to configure the beans. If you are using Spring MVC framework, the xml based configuration can be loaded automatically by writing some boiler plate code in web.xml file.
* **Annotation Based Configuration** - By using @Service or @Component annotations. Scope details can be provided with @Scope annotation.
* **Java Based Configuration** - Starting from Spring 3.0, we can configure Spring beans using java programs. Some important annotations used for java based configuration are @Configuration, @ComponentScan and @Bean.
* <https://www.digitalocean.com/community/tutorials/spring-ioc-bean-example-tutorial#spring-bean-java-config>

**Enable Spring Bean configuration for ConsumerFactory and Error Handlling:**

* Add TrainTrackingConsumerConfig class

|  |
| --- |
| package com.example.demo.config;  import java.util.HashMap;  import java.util.Map;  import org.apache.kafka.clients.consumer.ConsumerConfig;  import org.springframework.beans.factory.annotation.Value;  import org.springframework.context.annotation.Bean;  import org.springframework.context.annotation.Configuration;  import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;  import org.springframework.kafka.core.ConsumerFactory;  import org.springframework.kafka.core.DefaultKafkaConsumerFactory;  import org.springframework.kafka.support.serializer.ErrorHandlingDeserializer;  import org.springframework.kafka.support.serializer.JsonDeserializer;  import com.example.demo.model.TrackingEntity;  import com.fasterxml.jackson.databind.deser.std.StringDeserializer;  *@Configuration*  public class TrainTrackingConsumerConfig {  *@Bean*  ConcurrentKafkaListenerContainerFactory<String, Object> kafkaListenerContainerFactory(  ConsumerFactory<String, Object> consumerFactory) {  ConcurrentKafkaListenerContainerFactory<String, Object> factory = new ConcurrentKafkaListenerContainerFactory<String, Object>();  factory.setConsumerFactory(consumerFactory);  return factory;  }  ConsumerFactory<String, Object> consumerFactory(  *@Value*("${kafka.consumer.bootstrap-servers}") String bootstrapServer) {  Map<String, Object> config = new HashMap<>();  config.put(ConsumerConfig.***BOOTSTRAP\_SERVERS\_CONFIG***, bootstrapServer);  config.put(ConsumerConfig.***VALUE\_DESERIALIZER\_CLASS\_CONFIG***, ErrorHandlingDeserializer.class);  config.put(ErrorHandlingDeserializer.***VALUE\_DESERIALIZER\_CLASS***, JsonDeserializer.class);  config.put(JsonDeserializer.***VALUE\_DEFAULT\_TYPE***, TrackingEntity.class);  config.put(ConsumerConfig.***KEY\_DESERIALIZER\_CLASS\_CONFIG***, StringDeserializer.class);  return new DefaultKafkaConsumerFactory<>(config);  }  } |

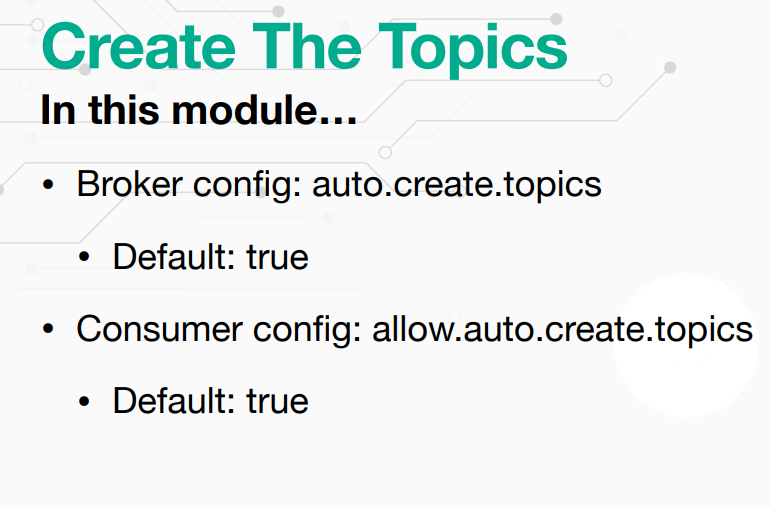
* Add containerFactory property in Kafka listner class :

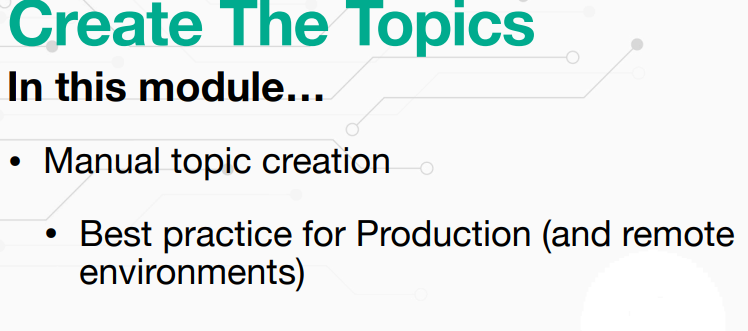
|  |
| --- |
| package com.example.demo.kafkalistner;  import org.springframework.kafka.annotation.KafkaListener;  import org.springframework.stereotype.Component;  import com.example.demo.model.TrackingEntity;  import jakarta.annotation.Resource;  import lombok.RequiredArgsConstructor;  import lombok.extern.slf4j.Slf4j;  @Component  public class TrainTrackingHandler {    @KafkaListener(  id = "paytmConsumerClient",  topics = "12797",  groupId = "kafka-paytm-group",  containerFactory = "kafkaListenerContainerFactory"  )  public void listner(TrackingEntity payload) {  System.out.println("Message Received " + payload);  }  } |

* No need to add write properties in application.properties fiel again for consumer and error handling for payload deserialization process.

**Creating Topics:**

* By default, either consumer or producer can create topics and it leads to create some unwanted topics and typo mistakes while communicating between producer and consumer end.
* By default – allow.create.topics: true for both consumer and producer.
* But it is not good practice to make it bydefault true property.

****

****

**Integration Testing:**

**Embedded Kafka using spring:**

We can test our kafka application before moving to production with help of emded kafka integration testing.

We have to add some dependencies which required to tets the created topics are produced and consumed properly.

We need to add following annotations:

* @SpringBootTest(classe={})
* @DirstisContext
* @ActiveProfile(“tets”) – passing our tets configuraions to emded kafak(bootstarp-servre)
* @EmdededKafka – to get run time apache kafka server
* @Autowired – to get it wired between all kafka beans like Kafka Templates and required objects to IOC container.

**Reference link:**

[**https://www.baeldung.com/spring-boot-kafka-testing**](https://www.baeldung.com/spring-boot-kafka-testing)

**Author Repo link:**

[**https://github.com/lydtechconsulting/introduction-to-kafka-with-spring-boot/tree/07-assignment-consume-and-produce**](https://github.com/lydtechconsulting/introduction-to-kafka-with-spring-boot/tree/07-assignment-consume-and-produce)

**Consumer Groups Parallel Processing:**

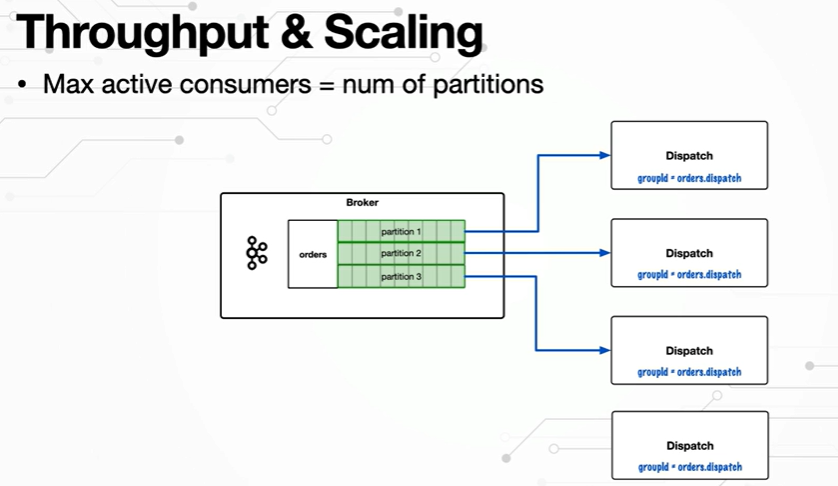
**Features of consumer groups:**

* Throughput
* Fault Tolerance
* Scaling
* Heartbeating
* Rebalancing

1. **Throughput :**

Kafka **throughput** is the measurement of the number of messages that an Apache Kafka® cluster processes in a specific period. It is one of the critical performance metrics in Kafka and is often a key determinant of the efficiency of data streaming operations.

****

****

1. Fault Tolerance:

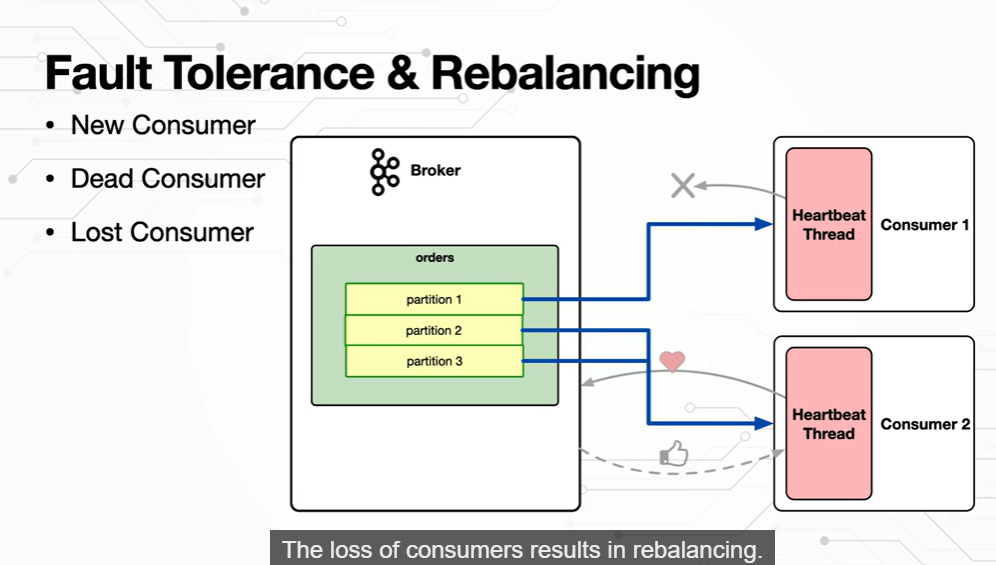
**What is fault tolerance in spring applications?**

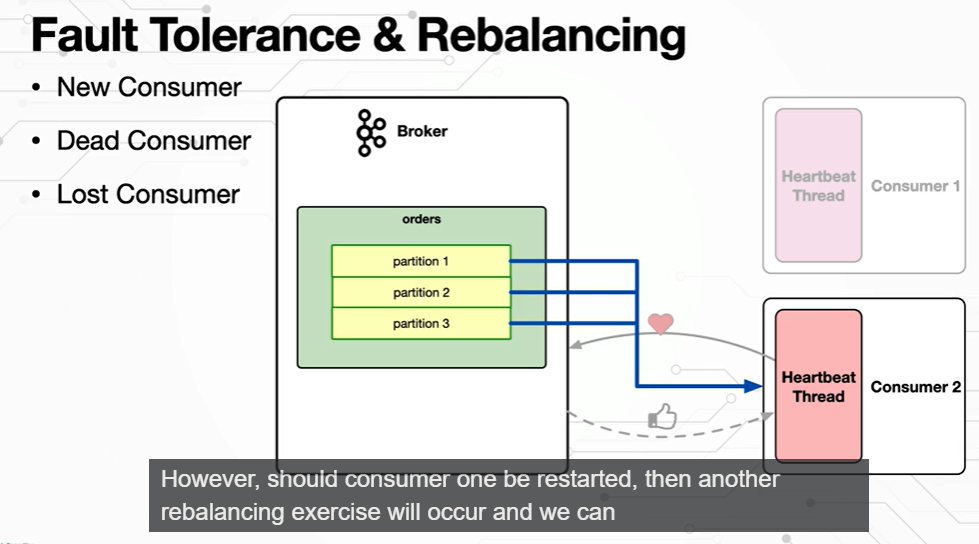
At any point of time, your application should be capable enough to give response for incoming traffic calls either downstream api’s/dependent micro-services apps available or inactive state, your application should be able to retry /fallback give proper error message which consumers/clients able to read and understand the problem.

- Heartbeat - consumer will keep check about kafka broker alive or not

- Poll Interval - some poll interval of time if consumer is not responding , then consumer group will rebalance with other consumers in consumer group.



****

****

**Consumer Gorups Based on it’s consumer’s behavior:**

* Shared Consumer Group:
* Consumer Failover
* Duplicate Consumption:

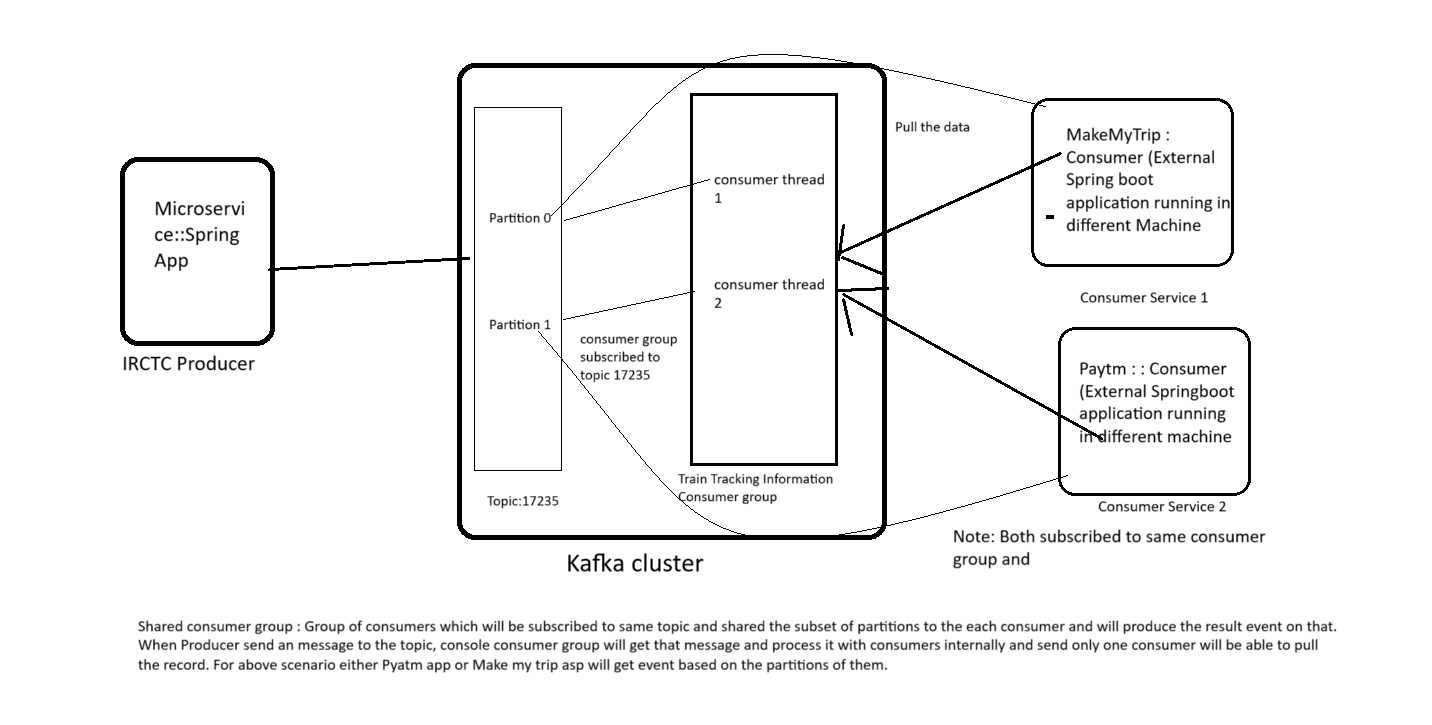
**Note:**

* Consumer in consumer group can have **one to many relationships** but not vice versa, means partition 1 can have only **one to one relationship** with one consumer only inside consumer group. <https://stackoverflow.com/questions/55291786/how-does-the-kafka-consumer-of-the-same-group-share-messages-between-them>

1. **Shared Consumer Group:**

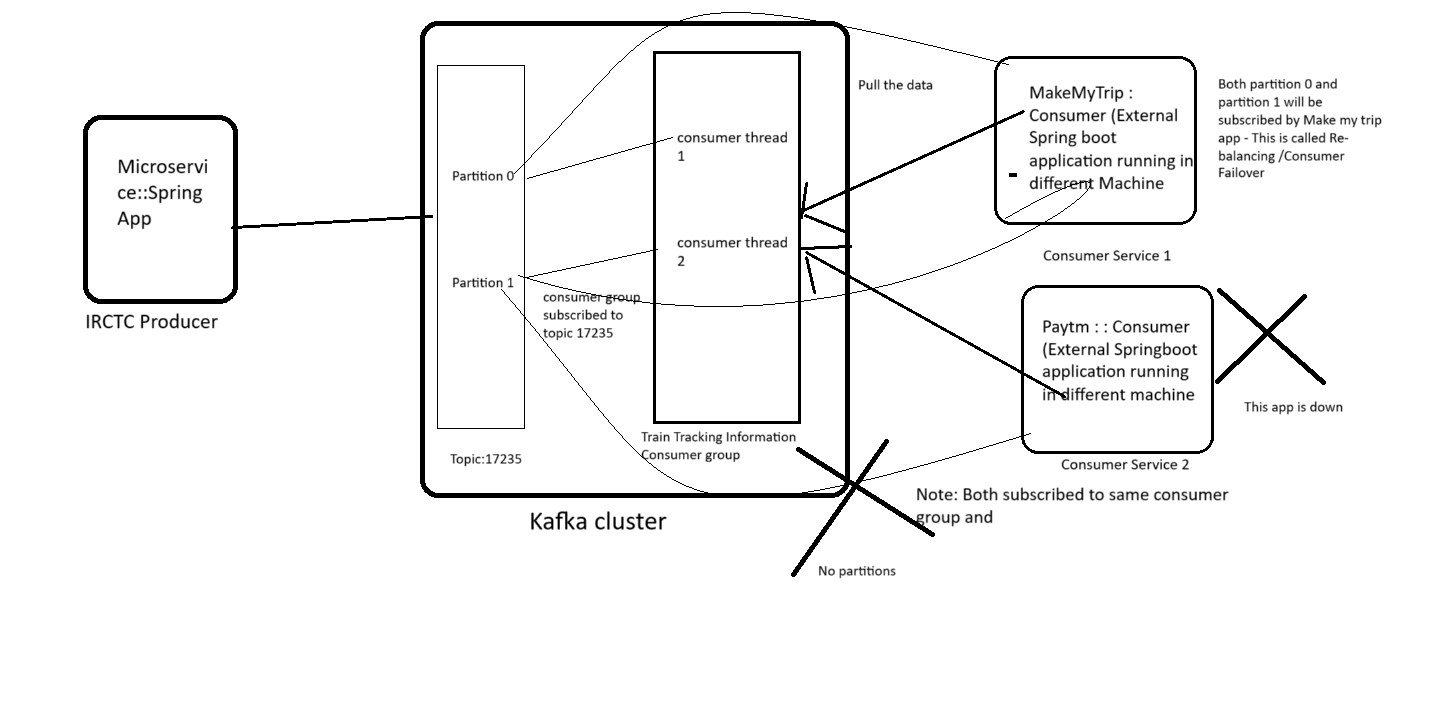
Group of consumers which will be subscribed to same topic and shared the subset of partitions to the each consumer and will produce the result event on that.

1. When Producer send an message to the topic, console consumer group will get that message and process it with consumers internally and send only one consumer will be able to pull the record. For above scenario either Pyatm app or Make my trip asp will get event based on the partitions of them.



**Consumer Failover - ReBalancing:**

* Let’s imagine above scenario after sometime if Paytm application went off/down, then what will happen from consumer group end?
* This is where our kafka consumer group acts smartly, it will re-balancing all the partitions to available consumers.
* For above case Make my trip will get Partition 0 and Partitions 1 subsets from topic(17235).



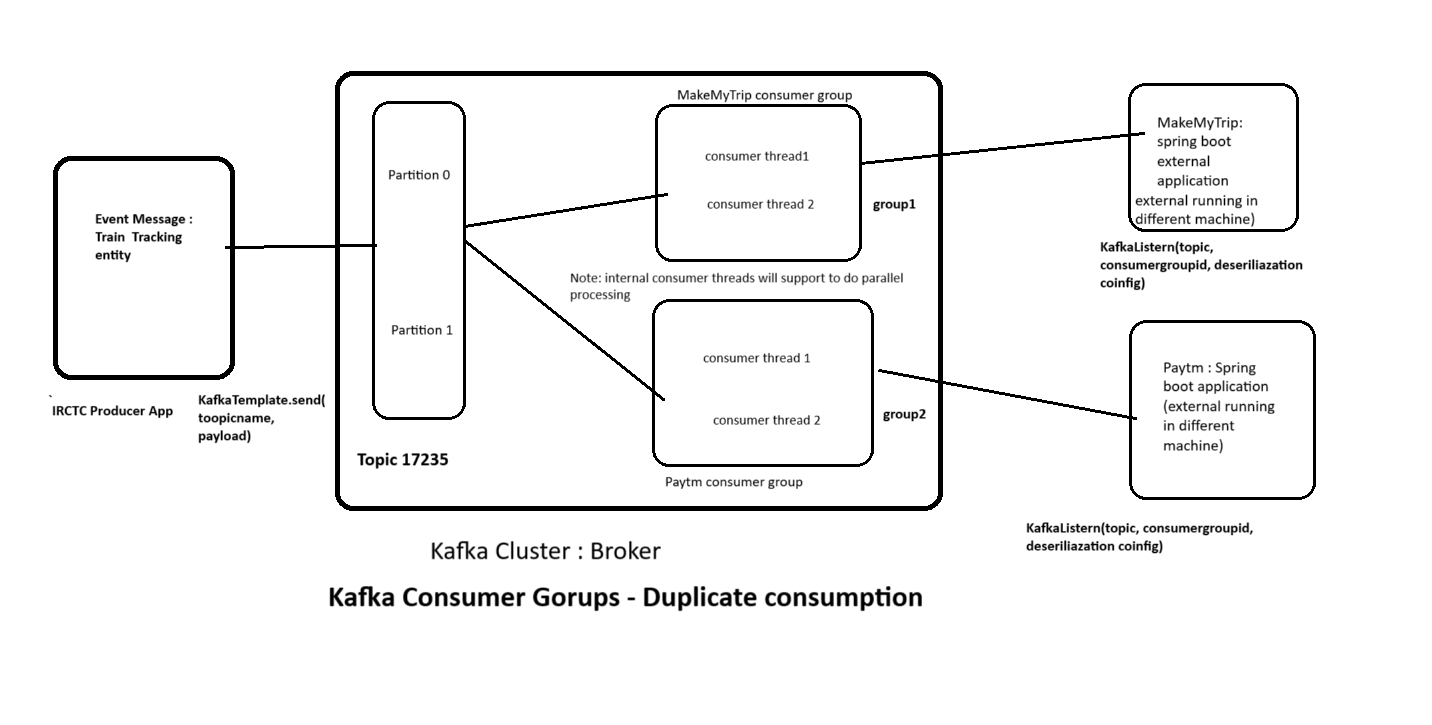
Both partition 0 and partition 1 will be subscribed by Make my trip app - This is called **Re-balancing /Consumer Failover**

**Duplicate Consumption:**

* The drawback of above both scenarios will be, only one consumer will get an dispatched event from consumer group even through they had multiple consumers subscribed to that consumer group:

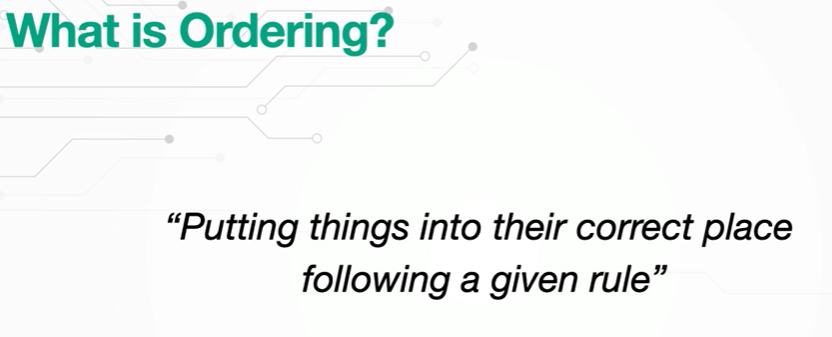
**How to overcome above issue?**

* We need to have separate consumer group for each consumer which need the messages to be consumed. Let’s say for above example we need separate consumer group for both **PayTm** and **MakeMytrip** applications separately.
* But both consumer groups will be subscribed to **same topic (17235)**.

****

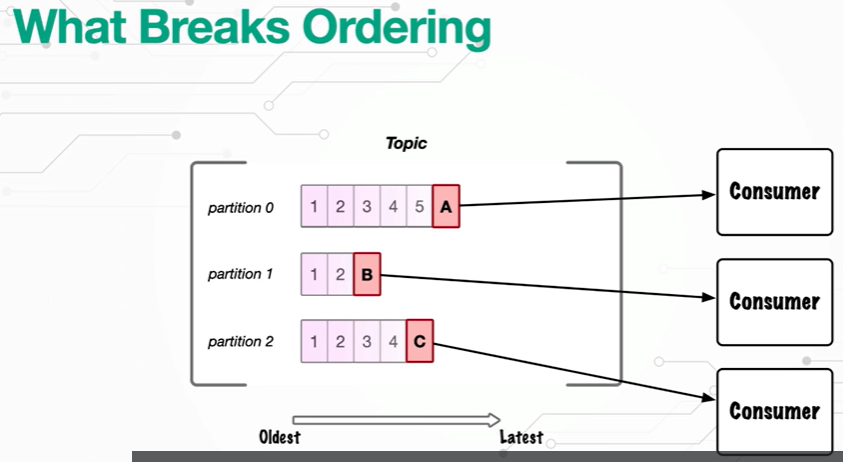
* **Consumer groups are required** for applications that need **all** of the messages from topic

**Keys, Partitions & Ordering:**



**Rule**: Passing Key inside message

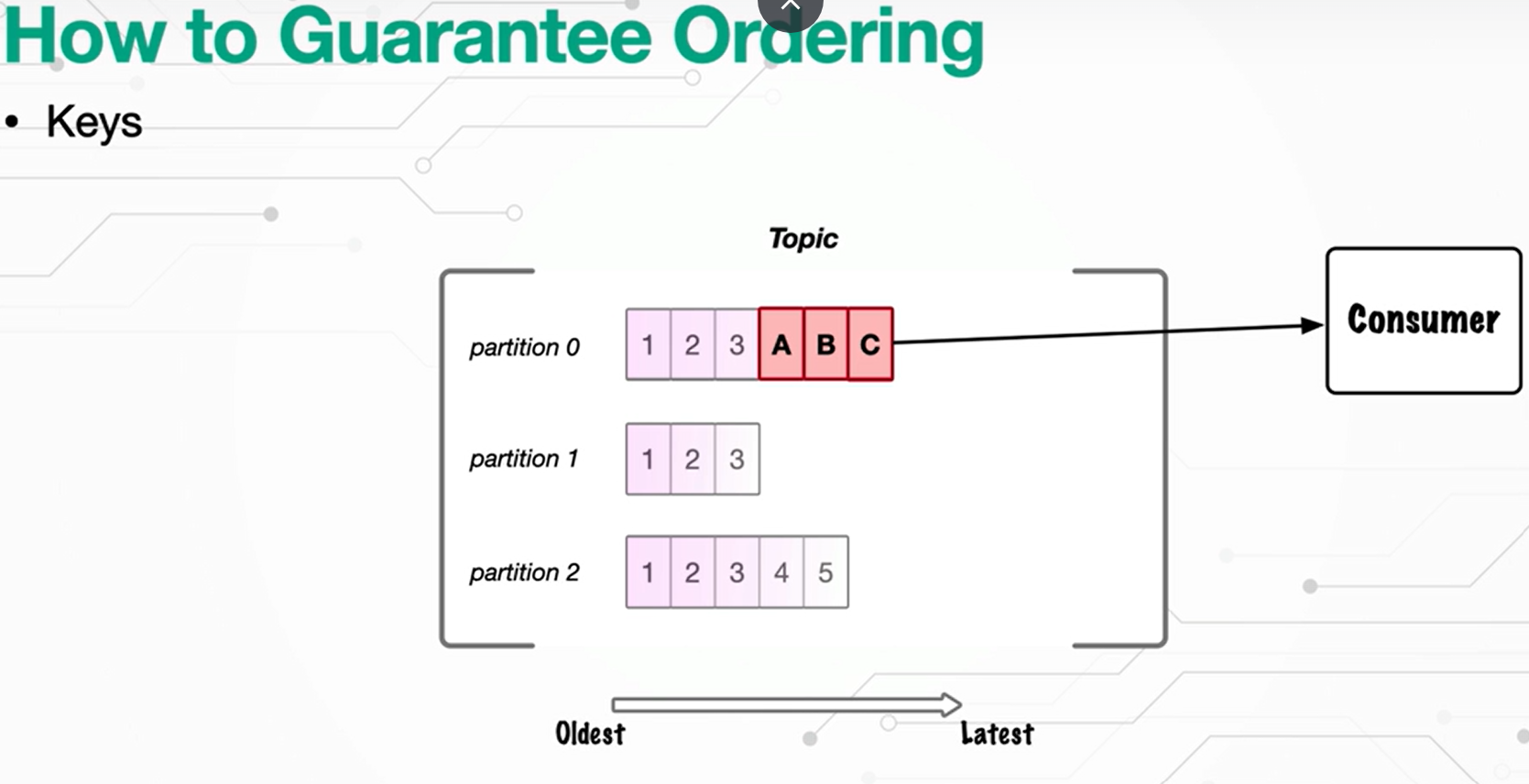
* If we insert in same partitions we will get Ordering of the messages, if we insert in different partitions then it won’t maintain the ordering. Ordering will be decided based on Message Key.
* **Producer** will send event with messages key and Kafka Producer will create hash code for that and based on partition algorithm inside Kafka it will decide which partitions should insert.
* If Key is null, then Kafka Producer will decide based on its own default algorithm **(Round Robbin)** it will decide which partition should insert.



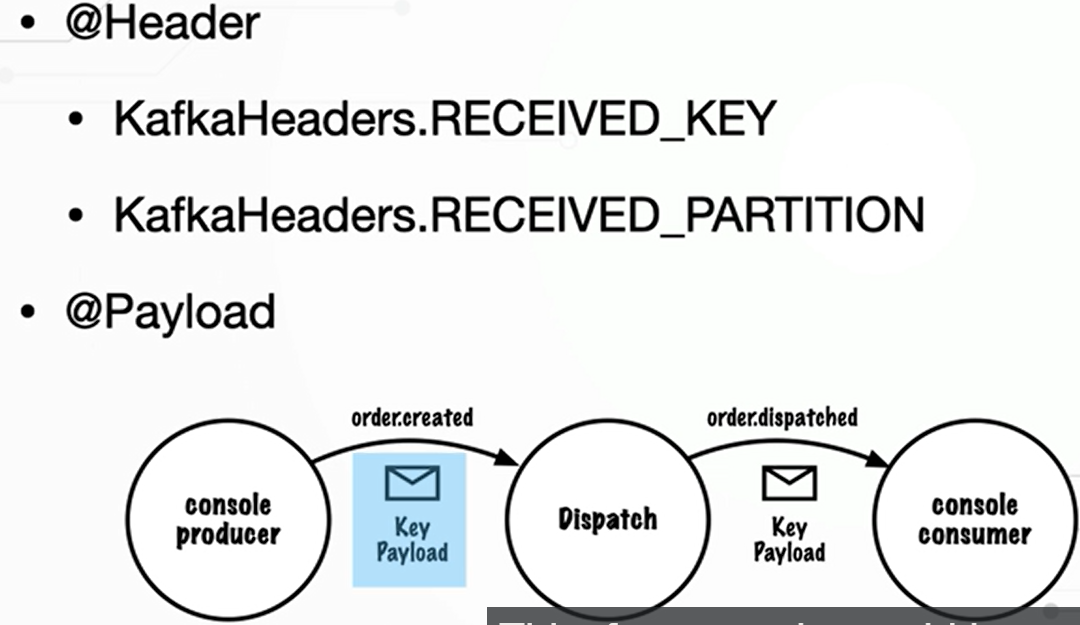
To main the ordering we need Message Key. One partition in the topic should be configured one consumer at a time but not vice versa.

Message Key is optional.

Message Key is any type (usrerid, order id, etc)



**Message Keys:**

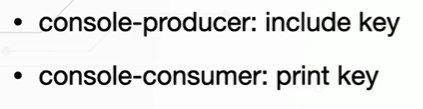
****

-

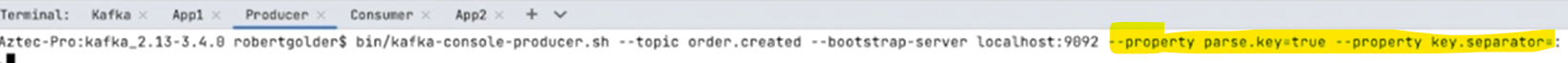
* If Consumer want to know about Message key and which partition it recived the data form kafka topic, then it has to annotated with @HEADER with KafkaHeader.RECEIVED\_KEY and KafkaHeader.RECEIVED\_PARTITION.
* Sample Kafka consumer app litsner handler class code :



**Consuming Keyed Messages:**

****

**Step1:** We have to include key in console producer



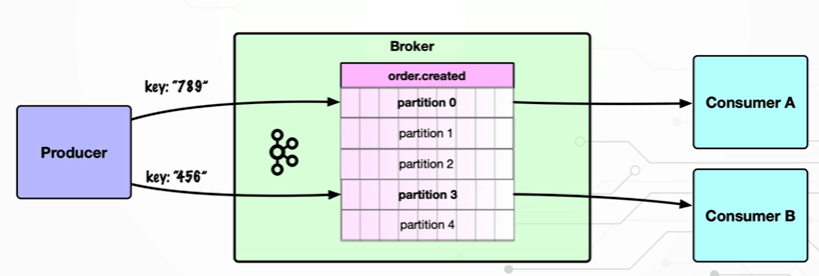
* **It is enabling with above two flag**
* **Kafka Internal Producer should know that we are using KEY along with message. So that kafka producer can decide based key which partition should send this message.**

**Step2:** We have to print key in console consumer

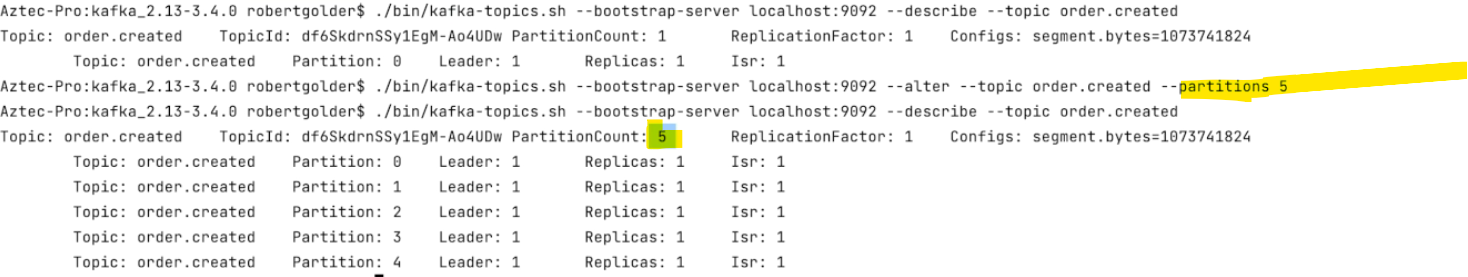


**Step3:** Passing Key along with messages, then it will go to same partition and maintain ordering.

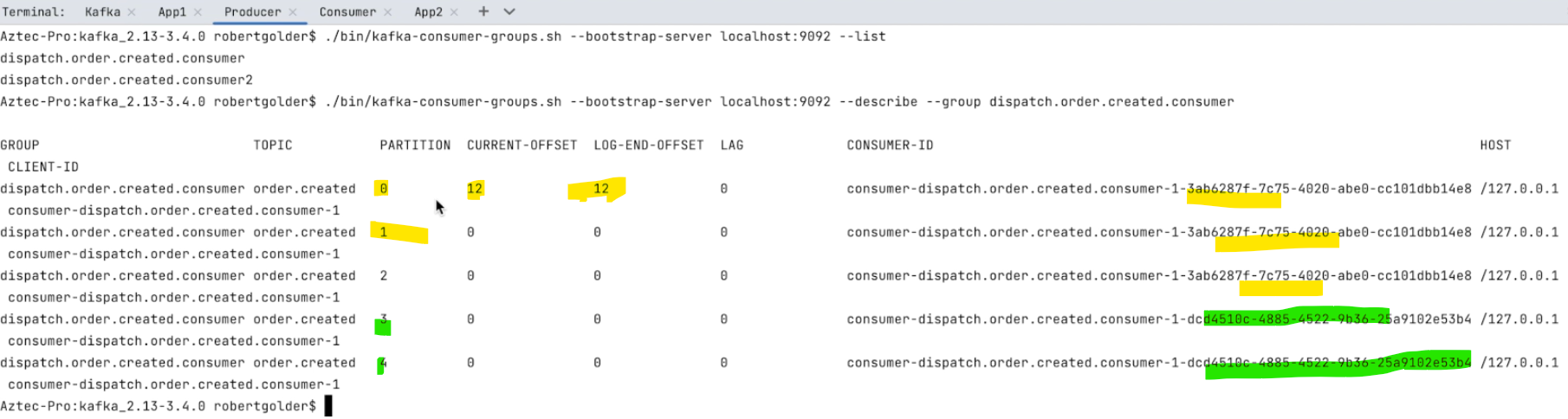




**Step 4:** Alter the topic with required number of partitions.

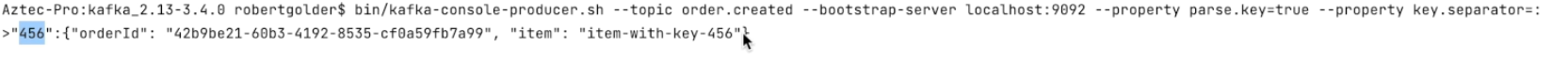


**Strep5**: Describe the consumer group how it got partitioned between the multiple consumers based on key.



**Important Note- (Need to discuss about this)**

Even though same application or different application same in consumer group if has its own key it will send the message to same partition.



* We have strong relation between partition and message
* Per consumer /per topic/ per partition