Component-based Software Development

Event-driven Microservices / Distributed Messaging/ Apache Kafka

Dr. Vinod Dubey
SWE 645
George Mason University

Acknowledgement / Reference

- https://kafka.apache.org/documentation/
- https://strimzi.io/documentation/
- http://www.google.com
- http://kafka.apache.org
- https://www.tutorialspoint.com/apache_kafka/apache_kafka_simple_producer_example.htm

Agenda

- **Microservices - Recap**
- **Apache Kafka**
  - Message Records
  - Topics/Partitions
  - Broker/Cluster
  - Producers/Consumers
- **Strimzi Kafka Operator**
  - Approach to deploy Kafka on Kubernetes

Microservices - RECAP

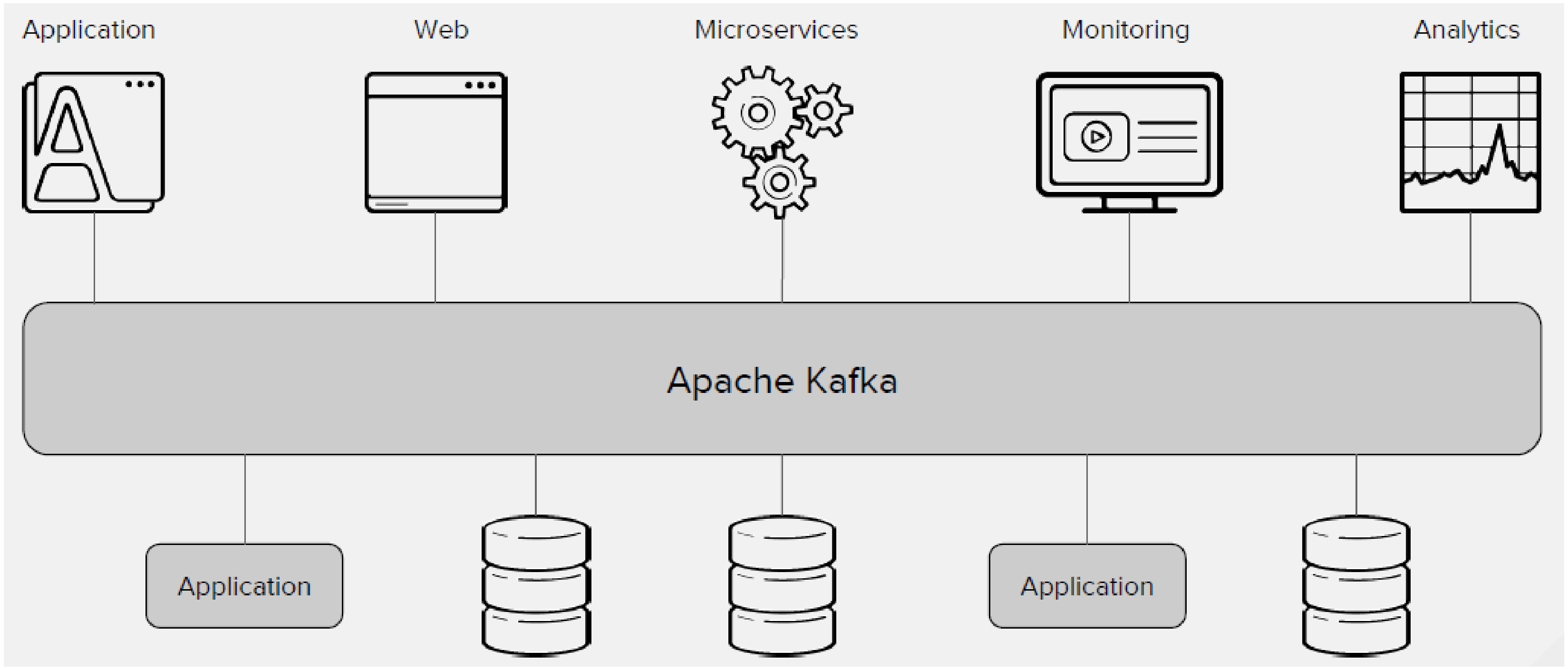
- **Microservices is an architectural style for building modular applications where a complex application can be structured as a collection of relatively smaller services**
  - Self contained based on single responsibility principle
  - Independently deployable, scalable, and maintainable, and testable.
  - Loosely coupled – communicate over a network using open-source technology and protocol
  - Often, microservices are contrasted with monolithic applications that are hard to maintain, update, and evolve; hard to scale

Microservices - RECAP

- **Generally, Microservices are wrapped into RESTful web services to expose its interface with request-response communication model**
- **Applications then chain different microservices together to implement their business processes**
  - In complex applications, this may result in many **point-to-point connections** that will be difficult to maintain
  - This has **potential for long latency** as one service makes synchronous call to other service that may call other services and waits for the response
  - It is based on the **assumption** that **all services are available** all the time, however if one service is not available for any reason, e.g., during maintenance window, this may break the chain
  - This **necessitates** to build some sort of **buffer** between the microservices calls by introducing **asynchronous interface**

Event Driven Microservices Architecture

- **In an event-driven microservice architecture, services communicate each-other via events/messages.**
  - When business events occur, producers publish them with messages.
  - Other services consume them through event listeners.
- **Apache Kafka serves as a central hub in an event driven microservices architecture**
  - Removes point-to-point connections

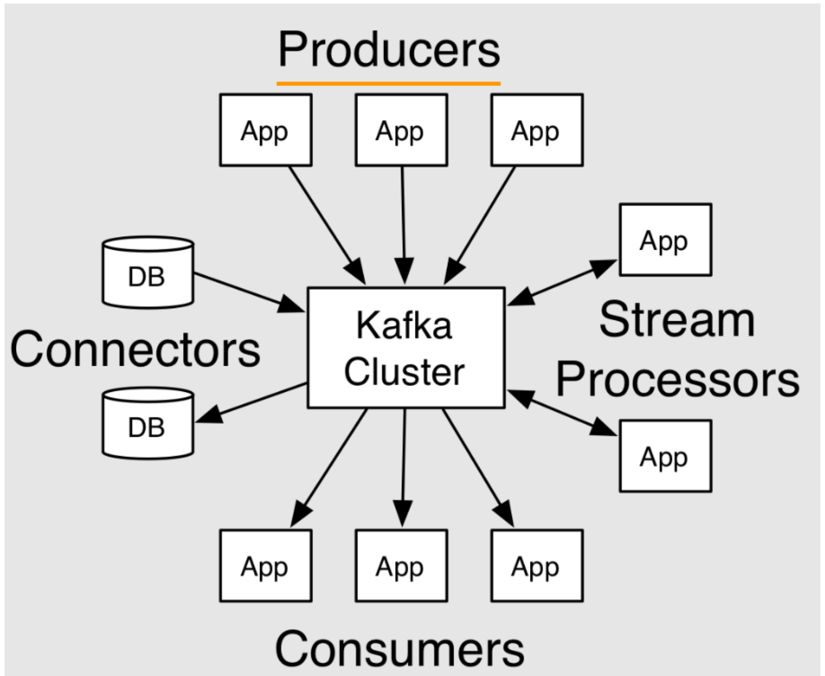


What is Apache Kafka?

- **Apache Kafka was developed as a publish/subscribe messaging system.**
- **It is often described as a**
  - "distributed commit log" or more recently as a
  - "distributed streaming platform" for handling real time data feeds at scale.
- **Data within Kafka is stored durably, *in order*, and can be read deterministically.**
- **The data can be distributed within the system to provide fault tolerance and scalability.**

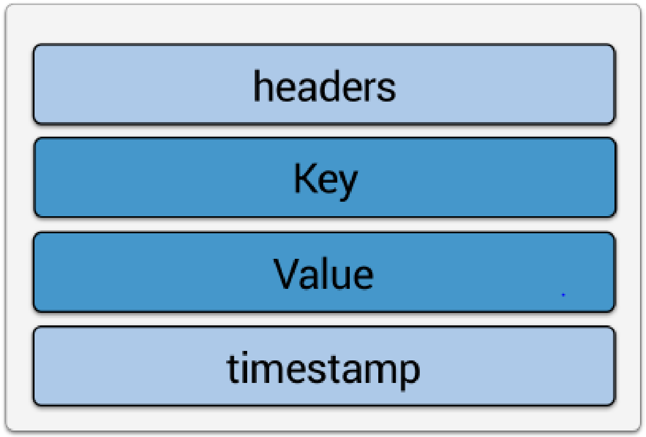
What is Apache Kafka?

- **Designed to be fast, scalable, durable, and highly available data platform:**
  - To **publish** (write) & **subscribe/consume** (read) streams of events/messages in near-real time
  - To **store** streams of events durably and reliably for as long as you want
  - To **process** streams of events as they occur, or retrospectively/retroactively
  - High throughput/low latency via partitioning (data sharding)
- **Originated at LinkedIn and later open sourced in 2011.**
  - Written in Scala, Java



Messages and Batches

- **The unit of data within Kafka is called a message.**
  - similar to a row or a record in a database table
- **This is what producers create and write to topic**
  - A message, also known as a message record or event,
- **Typically, a message record has a key and value holding the business relevant data**
  - Can have optional headers and timestamp – creation time (default) or ingestion time
  - Brokers can handle message of up to 1 MB in size, although Kafka is optimized for smaller message size (1 KB)



Messages and Batches

- **A message is simply an array of bytes as far as Kafka is concerned**
  - Kafka stores data as ByteArray or binary array (arrays of bytes).
- **Producers convert data from its native format into byte arrays before sending to Kafka.**
  - This conversion is called serialization.
- **Kafka API serializes key and value of the message**
  - need to provide Serializer for key and value

Messages and Batches

- **The data (i.e., message) consumers fetch from Kafka comes in the form of byte arrays that must be deserialized in order to be read.**
- **There are many serializers and deserializers available for Kafka,**
  - such as for Avro, JSON, String, custom, but the one for Avro is popular in the Kafka ecosystem.

Messages and Batches

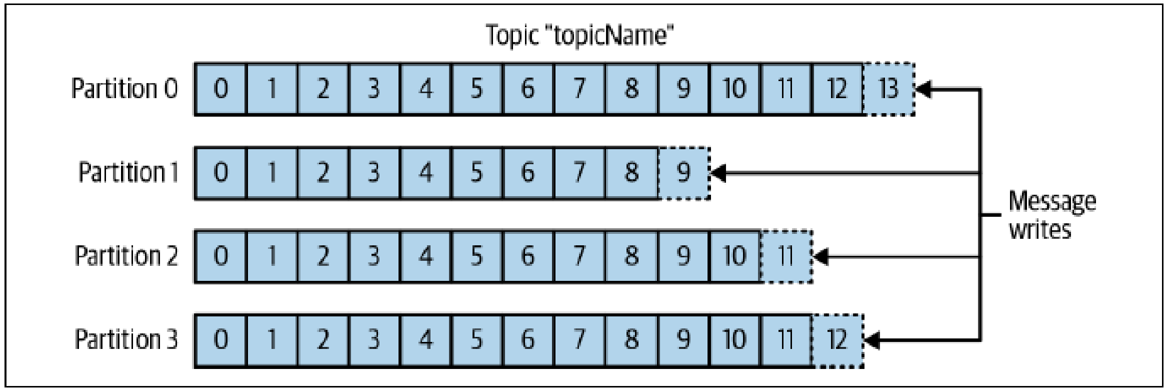
- **A message can have an optional piece of metadata, which is referred to as a key.**
  - The key is also a byte array.
- **Every record a producer produces has *an optional key* and a value. The value is the actual data that you want to store in Kafka topic**
  - If the record is sent without a key, Kafka will append the log to one of the partitions using *round robin* (by default)
  - If the record has a key, it is sent to a specific partition
  - Uses hash(key) / # of partitions to decide which partition to send the record to
  - Records with the same key go to the same partition

Messages and Batches

- **For efficiency, messages are written into Kafka in batches.**
- **A batch is just a collection of messages, all of which are being produced to the same topic and partition.**
- **Messages are delivered in batches for improved throughput**
  - Batches and records contain headers and metadata
  - An individual round trip across the network for each message would result in excessive overhead, and collecting messages together into a batch reduces this.
- **This is a trade-off between latency and throughput:**
  - the larger the batches, the more messages that can be handled per unit of time, but
  - the longer it takes an individual message to propagate.
- **Batches are also typically compressed,**
  - providing more efficient data transfer and storage at the cost of some processing power.

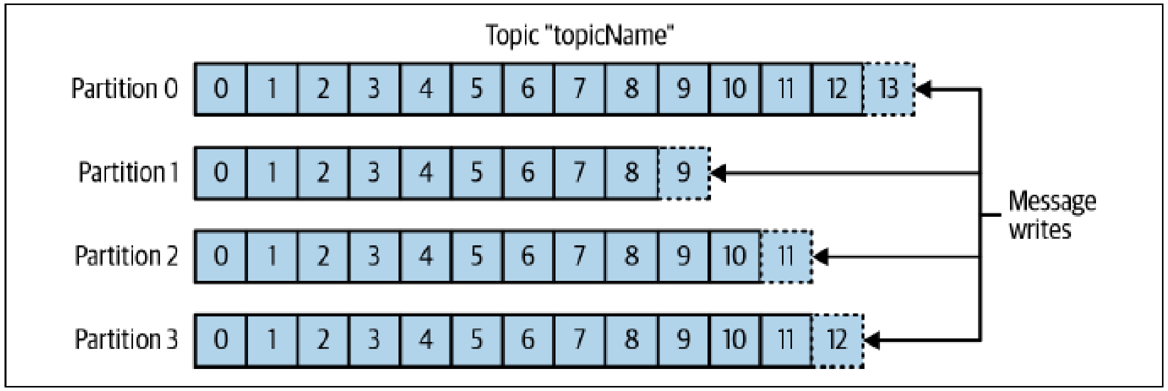
Topics and Partitions

- **Messages in Kafka are categorized into topics.**
- **A Kafka Topic provides a channel to publish and subscribe streams of events**
  - A logical representation of data or Logical grouping of like messages, e.g., weather-data, book-orders, credit-card-transactions, a twitter feed
  - Messages are sent to and received from a topic
  - The closest analogies for a **topic** are a database **table** or a **folder** in a filesystem.



Topics and Partitions

- **Topics are additionally broken down into a number of partitions.**
  - where message records are stored – usually records of same type
  - A topic partition is a committed append-only log – the terms partitions and logs are used interchangeably
  - Partitioning is usually done based on the message key
  - All actual work is done at the partition level – topic is just a virtual object. Each message is written only into one selected partition
    - Going back to the "commit log" description, a partition is a single log.
    - Messages are written to it in an append-only fashion and are read in order from beginning to end.

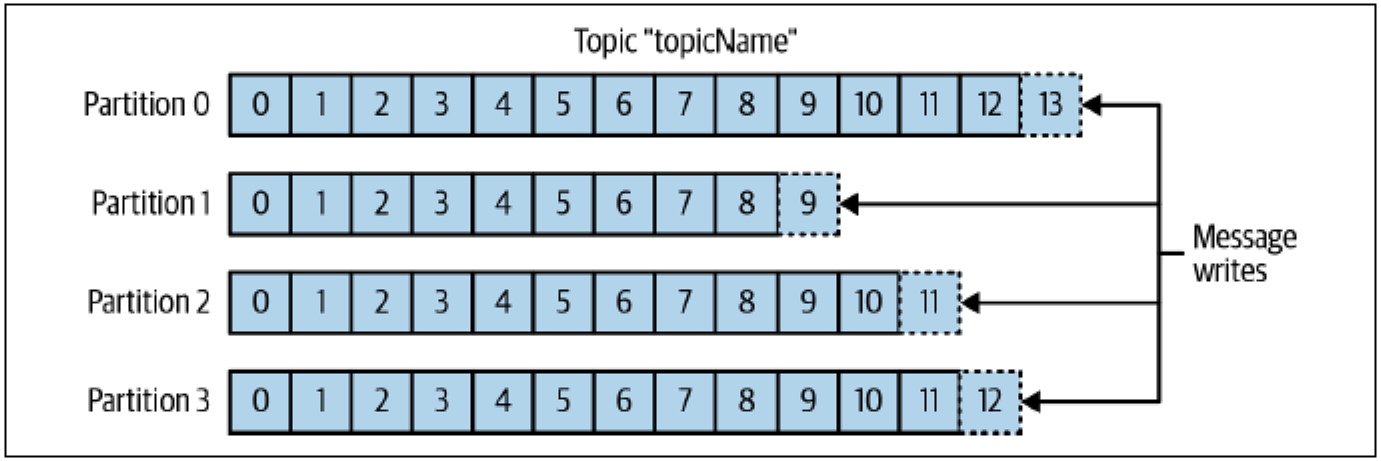


Topics and Partitions

- **Partitioning does not give high availability, they are used to parallelize the work related to a topic**
  - For high availability of data, partitions needs to be replicated across brokers
- **Topic Partitions are:**
  - distributed across brokers for distributed processing and for performance and scalability
  - the ones where you write data to and read data from – a way to distributed processing
  - replicated across brokers for durability, fault tolerance / resiliency
  - The number of Topic Partitions is defined by a topic partition count.

Topics and Partitions

- **There is no guarantee of message ordering across the entire topic, just within a single partition.**
  - Message ordering within a partition is guaranteed
  - Figure shows a topic with four partitions, with writes being appended to the end of each one.



Topics and Partitions

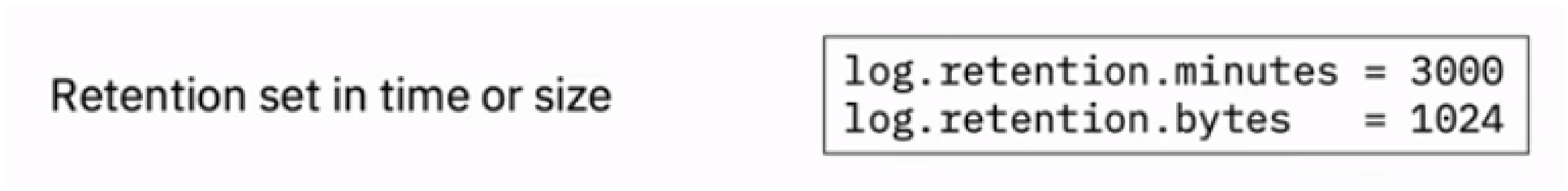
- **Partitions are the way that Kafka provides redundancy and scalability.**
- **Each partition can be hosted on a different server,**
  - which means that a single topic can be scaled horizontally across multiple servers to provide performance far beyond the ability of a single server.
- **Partitions can be replicated, such that different servers will store a copy of the same partition in case one server fails.**
  - Each partition can exist in one or more backup copies to achieve high availability in case of node failures

Topics and Partitions

- **Segment File**
  - A partition has one or more segments
  - When the size of a segment file reaches 1 GB (default), the next one gets created
  - A segment can have as many as five physical files
  - A segment file can’t have data past 7 days (default) – Kafka deletes the entire segment
  - Log.segment.bytes, and log.roll.ms determine when to close a segment file and open a new segment

Partition – Retention

- **A key feature of Apache Kafka is that of retention, which is the durable storage of messages for some period of time.**
- **Retention determines how long does a record persist in a topic**
- **Retention**
  - Based on message age or size
    - Retention can be specified using time or size
    - In other words, how long a log stays in a partition before it gets deleted


Retention set in time or size

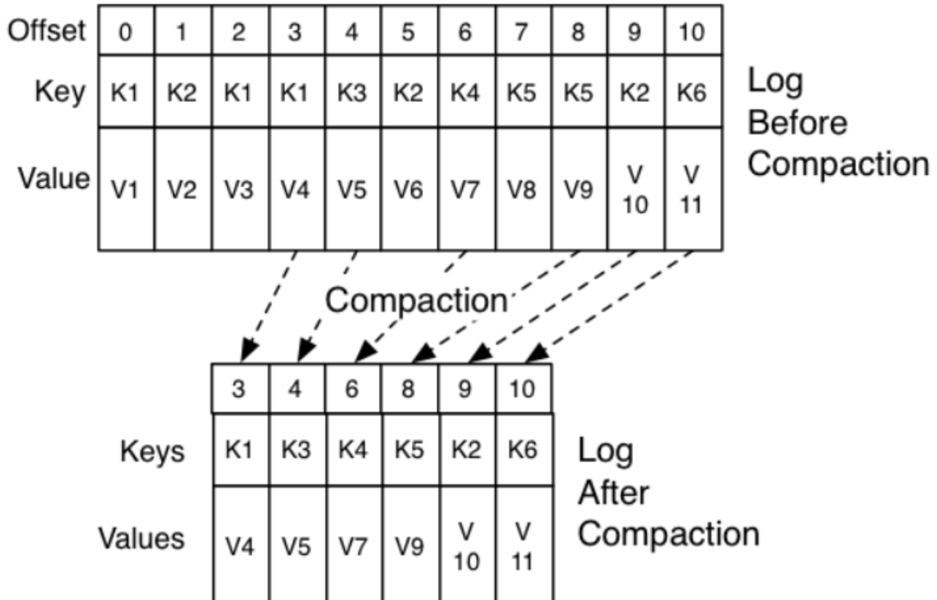
```
log.retention.minutes  = 3000
log.retention.bytes    = 1024
```

Partition – Retention

- **Kafka brokers are configured with a default retention setting for topics, either retaining messages for some period of time (e.g., 7 days) or**
  - until the partition reaches a certain size in bytes (e.g., 1 GB).
  - Once these limits are reached, messages are expired and deleted.
  - In this way, the retention configuration defines a minimum amount of data available at any time.
- **Individual topics can also be configured with their own retention settings so that messages are stored for only as long as they are useful.**

Partition – Log Compaction

- **Topics can also be configured as log compacted, which means that Kafka will retain only the last message produced with a specific key.**
  - This can be useful for changelog-type data, where only the last update is interesting.
- **Log Compaction allows to take the latest value of a given key**
  - The compact policy only works for keyed messages.
  - Configure the Topic's cleanup.policy to compact,
    - The log cleaner will only preserve the last message with that key; older messages with that key will be deleted.

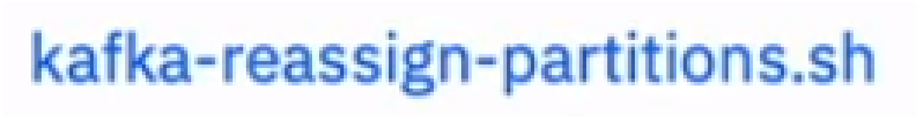


Offset	0	1	2	3	4	5	6	7	8	9	10
Key	K1	K2	K1	K1	K3	K2	K4	K5	K5	K2	K6
Value	V1	V2	V3	V4	V5	V6	V7	V8	V9	V 10	V 11

	3	4	6	8	9	10
Keys	K1	K3	K4	K5	K2	K6
Values	V4	V5	V7	V9	V 10	V 11

Partition Rebalancing

- **Generally, partitions are balanced (i.e., uniformly distributed) across the brokers**
  - This ensures that load (from producers) is spread across the brokers
  - Partition rebalancing is needed especially when we want to either decrease or increase the number of nodes in the cluster.



```
kafka-reassign-partitions.sh
```

Brokers and Clusters

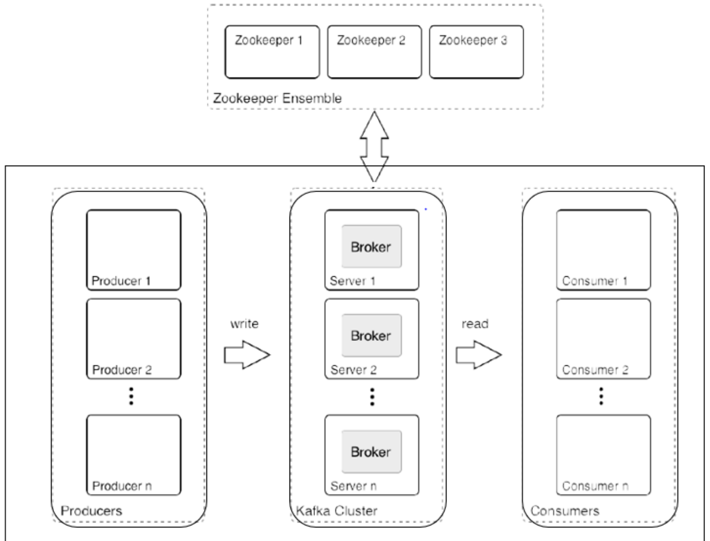
- **A single Kafka server is called a broker**
- **Kafka Broker is a server or software responsible for hosting Kafka Topics and delivering messages**
  - The broker receives messages from producers,
    - assigns offsets to them, and
    - writes the messages to storage on disk.
  - The broker also services consumers,
    - responding to fetch requests for partitions and
    - responding with the messages that have been published.

Brokers and Clusters

- **A single broker can easily handle thousands of partitions and millions of messages per second**
  - This largely depends on the specific hardware and its performance characteristics
  - One broker instance can handle hundreds of thousands of reads and writes per second as well as can handle TB of messages without performance impact.
  - Brokers can run on virtual machines, physical machines or in containers

Brokers and Clusters

- **Kafka brokers are designed to operate as part of a cluster.**
- **Kafka cluster is comprised of a number of Kafka Brokers – networked together to form a cluster**
  - Typically, three brokers or more for load balancing
  - Each broker has its own unique ID



Brokers and Clusters

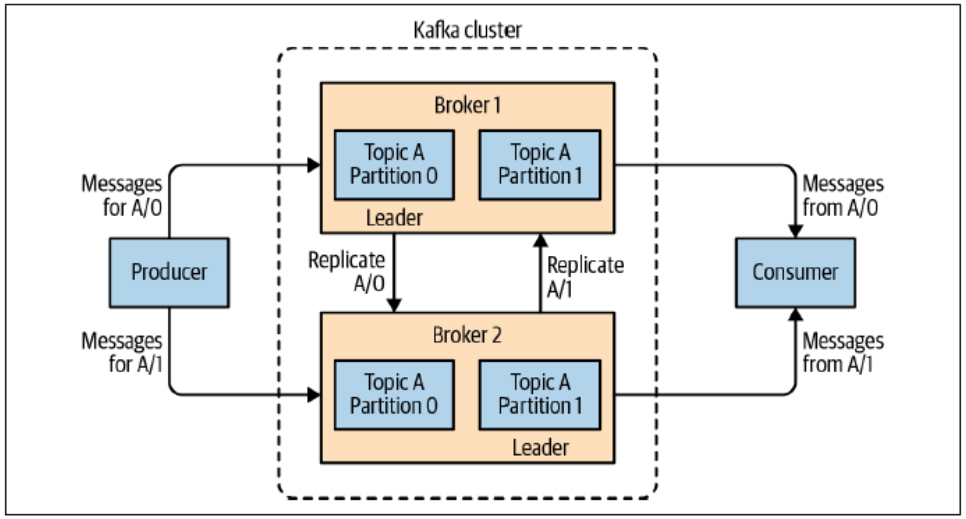
- **Within the cluster of brokers, one of the brokers in the cluster is given additional responsibility of being a controller**
  - Elected automatically from the live members of the cluster
  - The controller is responsible for administrative operations,
  - including assigning partitions to brokers and monitoring for broker failures.
  - One broker contacts ZooKeeper and declares itself the controller
  - This is in addition to being the potential leader of different partitions

Brokers and Clusters

- **In older version of the Kafka, Controller maintains brokers' liveliness via ZoopKeeper and keeps track of all cluster matadata of all other brokers**
- **The controller determines who is the leader for different partitions**
- **The controller information is stored in ZooKeeper that makes sure only one broker is assigned as a Controller**
  - ZooKeeper tree stores information about the brokers and their IDs, endpoints, controller (i.e., which broker is controller), controller epoch etc.
  - Any broker that tries to be a controller registers its intent and adds an entry into the ZooKeeper tree.
    - Only one entry for a controller from only one broker is allowed.
    - If the controller broker goes down or have connectivity issue with ZooKeeper, ZK allows next connecting Broker to become the controller by updating (or incrementing) the controller epoch

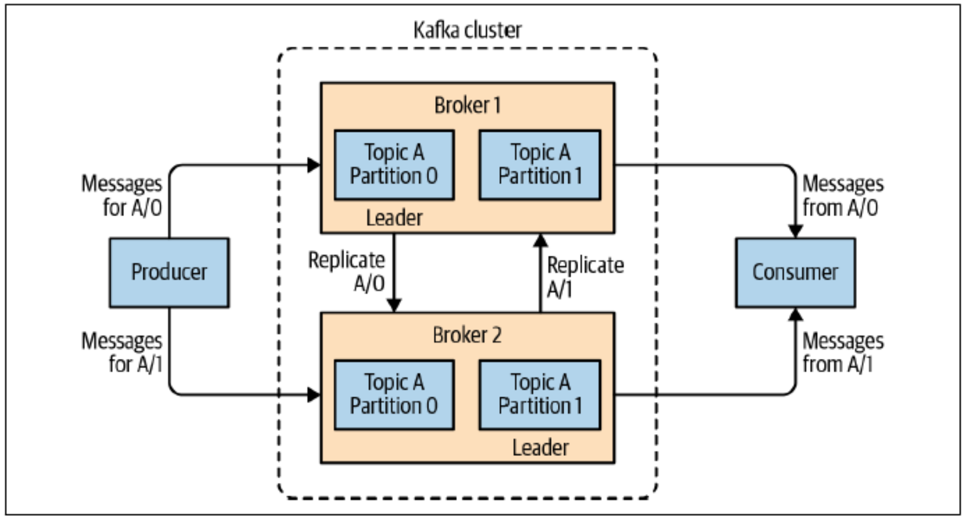
Brokers and Clusters

- **A partition is owned by a single broker in the cluster, and that broker is called the leader of the partition.**
- **Partition leader**
  - A partition leader a broker that handles all producer requests for a topic.
  - Kafka can spread leaders of different partitions across brokers (i.e., load balance leadership across brokers) and avoid hot brokers
    - By default, all consumers read from the leader of a partition. Producers write only to the leader – this may lead to hot brokers if partition leadership is not distributed
- **All producers must connect to the leader in order to publish messages,**
  - but consumers may fetch from either the leader or one of the followers.



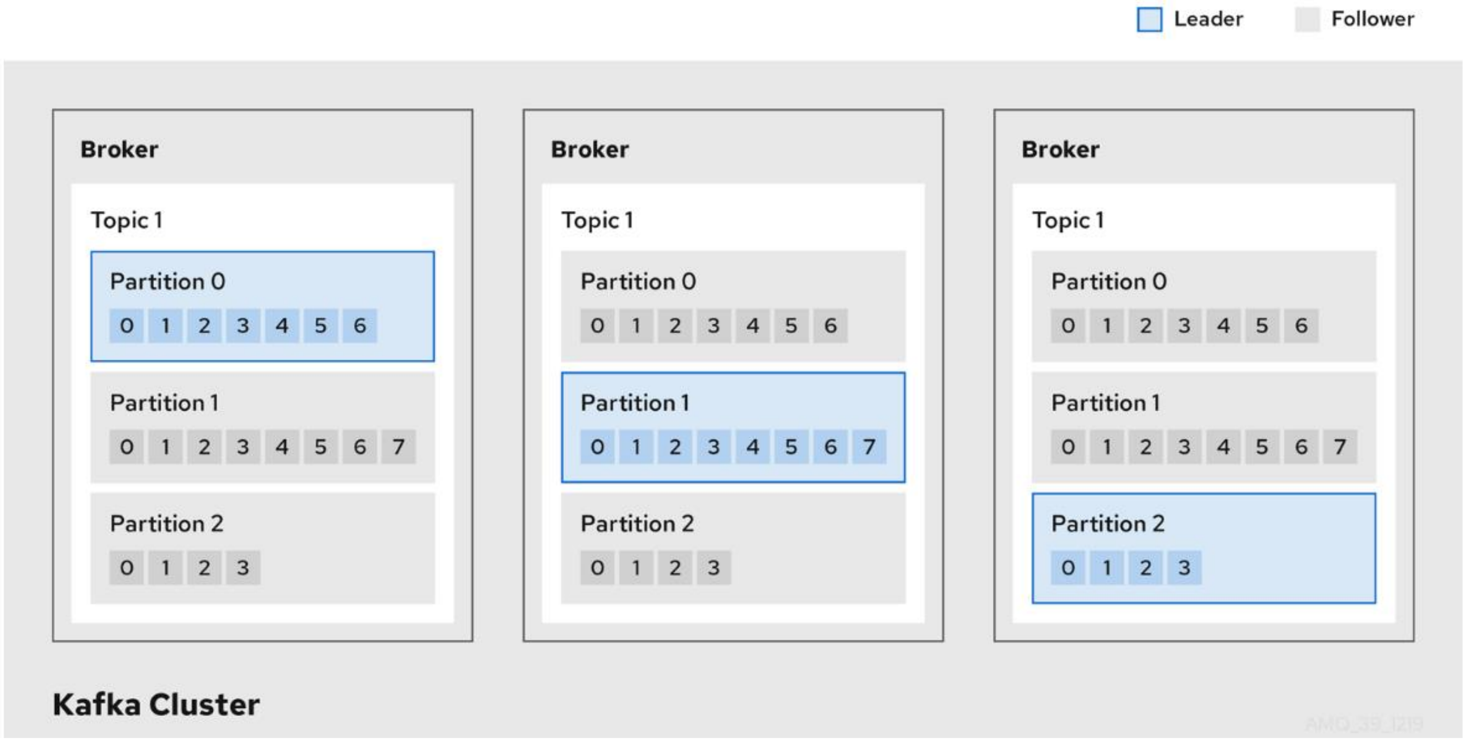
Brokers and Clusters

- **A replicated partition is assigned to additional brokers, called followers of the partition.**
- **Partition follower**
  - A partition follower is broker that replicates the partition data of a partition leader
  - Topics use a replication factor to configure the number of replicas of each partition within the cluster. – default replication factor is 1.
  - A topic comprises at least one partition.
- **Replication provides redundancy of messages in the partition,**
  - such that one of the followers can take over leadership if there is a broker failure.



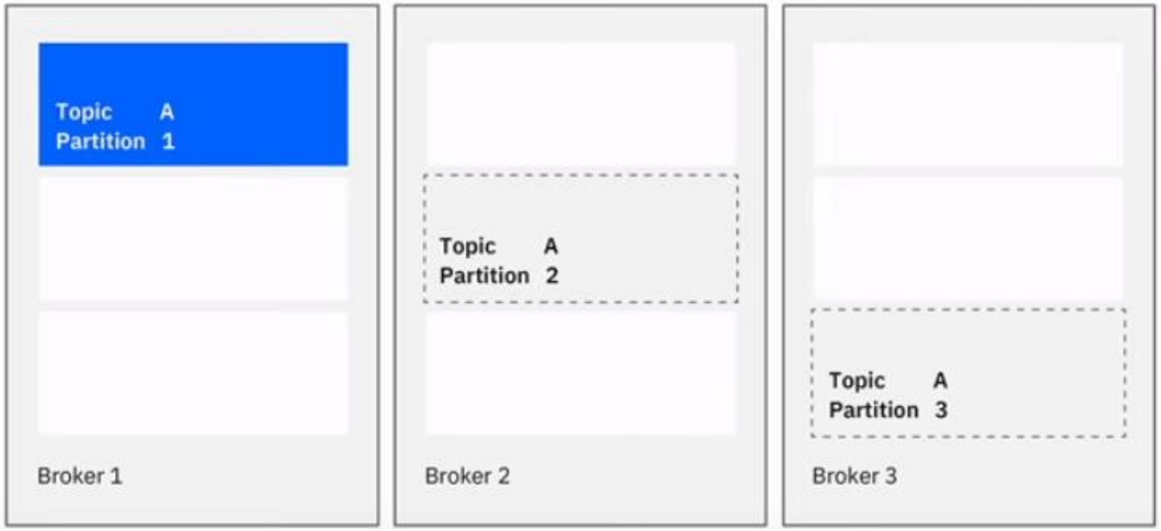
Brokers and Clusters

- In this example, each partition has a **leader** and **two followers** in replicated topics.



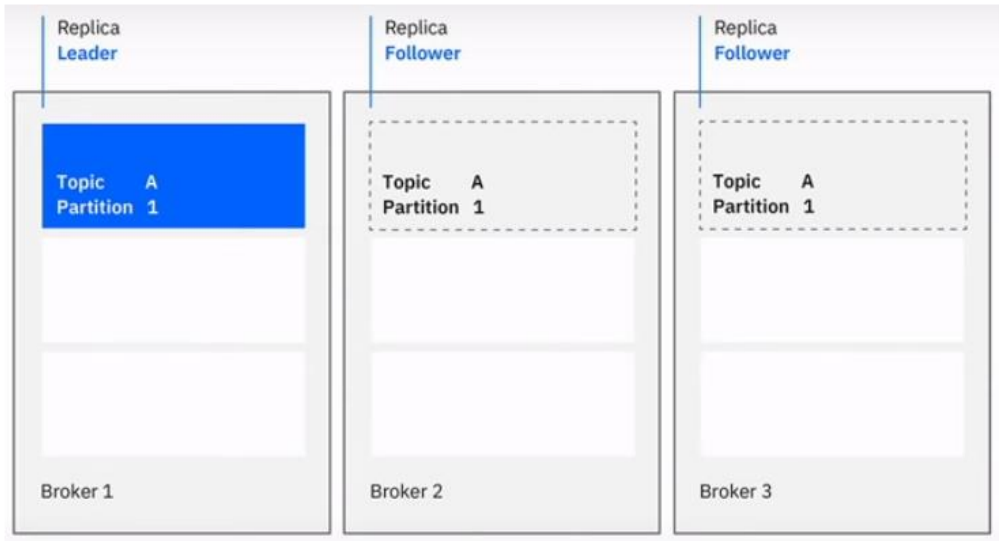
High Scalability through Partitions

- **Kafka supports scalability and high throughput by using scaled out architecture where it distributes the topic partitions across different brokers/servers in the cluster**
  - Distributed partitions allows multiple applications to send events to same topic concurrently by sending data to different partitions on different brokers, without having to overwhelm one broker



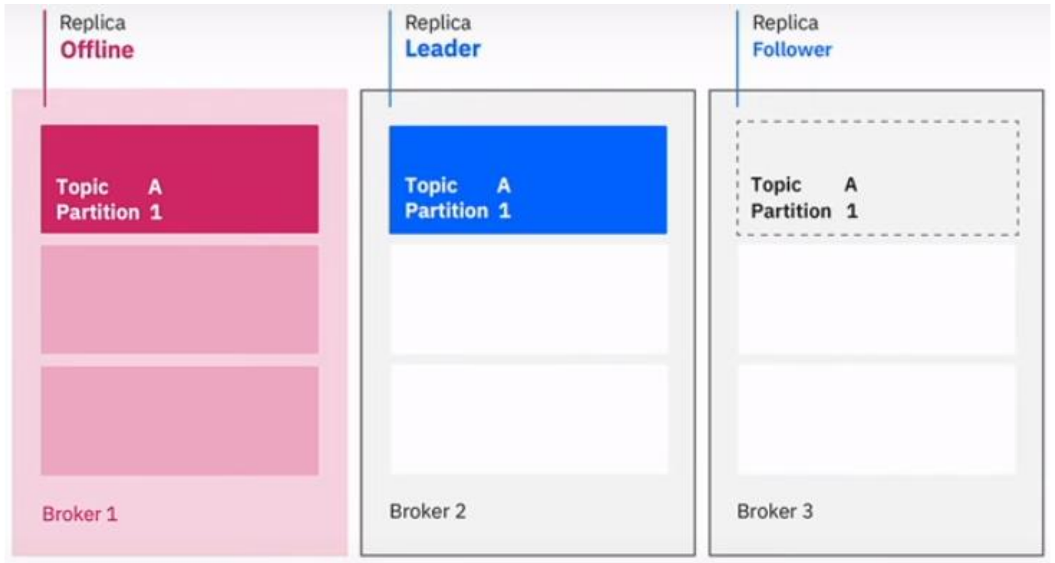
High Availability through Replication

- **Kafka can be configured to be highly available by using Replication, that allows multiple copies of a partitions across different brokers**
  - Kafka prevents a single broker from having more than one replica of the same partition
  - Kafka client applications talk to the leader (broker) to write/read data to/from a partition
  - Kafka, under the cover, replicates all the data across to the other brokers (followers)



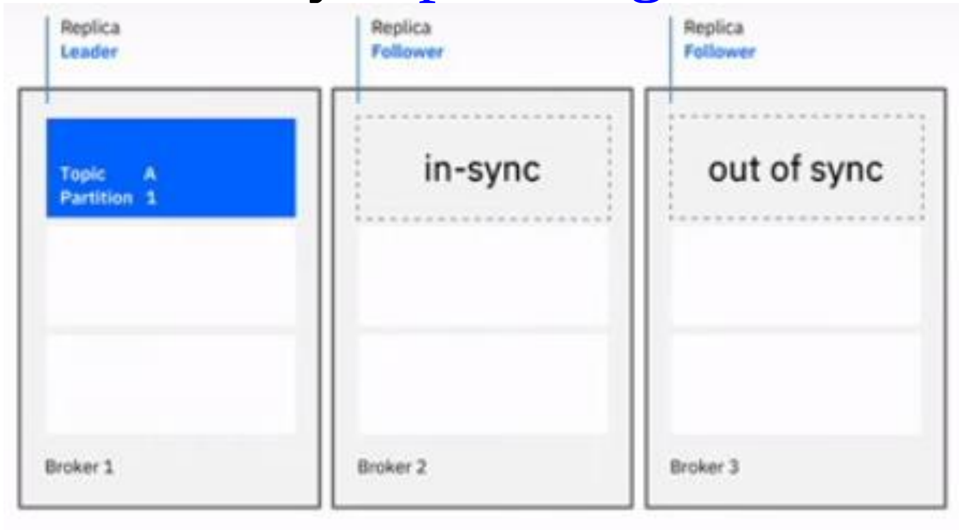
High Availability through Replication

- **If the leader was to go down, Kafka elects one of the followers (that has the same data) as the new leader for that partition, and all client applications start communicating with the new leader (of that topic partition)**
  - When clients try to connect to the Kafka, it tells which broker is the new leader and the clients continue publishing/consuming data to/from for that partition via the new leader



In-sync vs. out-of-sync Replicas

- **Situation of out-of-sync replicas arise if under-the-cover replication is not keeping up**
  - The leader tracks the last requested offset of followers
  - A replica is out of sync if it hasn't requested a message in more than 10 sec or it hasn't caught up to recent messages in 10 sec.
  - Time controlled by replica.lag.time.max.ms

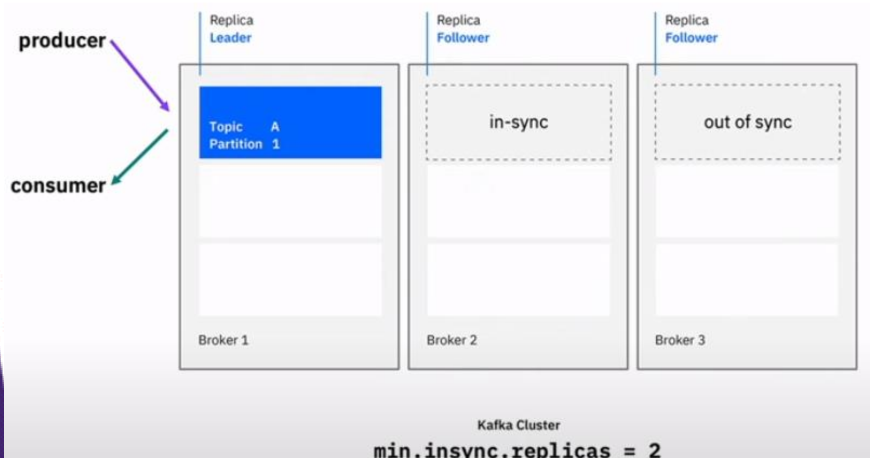


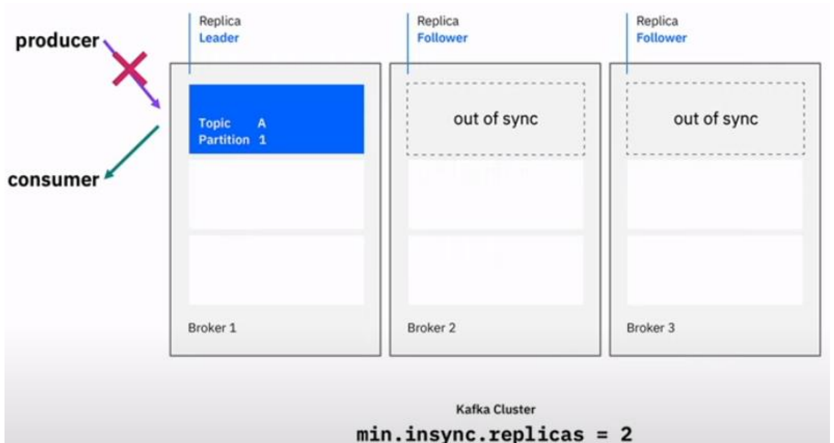
Min In-sync Replicas

- **Refers to the number of replicas that must be in-sync for a partition of the topic to be considered available**
- **min.insync.replicas – a configurable property**
  - An administrator declares the number of replicas (for a partition) as well as min-insync replicas at the time of topic creation
- **In-Sync Replica**
  - An in-sync replica has the same number of messages as the leader.
  - A message is considered committed only if it’s received by all replicas in the in-sync-replica set
    - Configuration defines how many replicas must be in-sync to be able to produce messages, ensuring that a message is committed only after it has been successfully copied to the replica partition. In this way, if the leader fails the message is not lost.

Min In-sync Replicas

- **Let min.insync.replicas =2 on a cluster with 3 brokers and with the replication factor 3,**
  - In this example, one out-of-sync replica is tolerable and producers and consumers will be allowed to write and read data from the leader
  - However, if another broker goes down, the min.insync.replicas is less than 2, in that case Kafka will prevent any producers to send events into the Kafka
    - Producers will get an error message from Kafka that min-insync-replicas is not high enough
  - This avoids a situation where the producer sends some more events to the leader and the data is not being replicated and if the leader goes down, then data may get lost
  - Consumers can still continue consuming what's there already



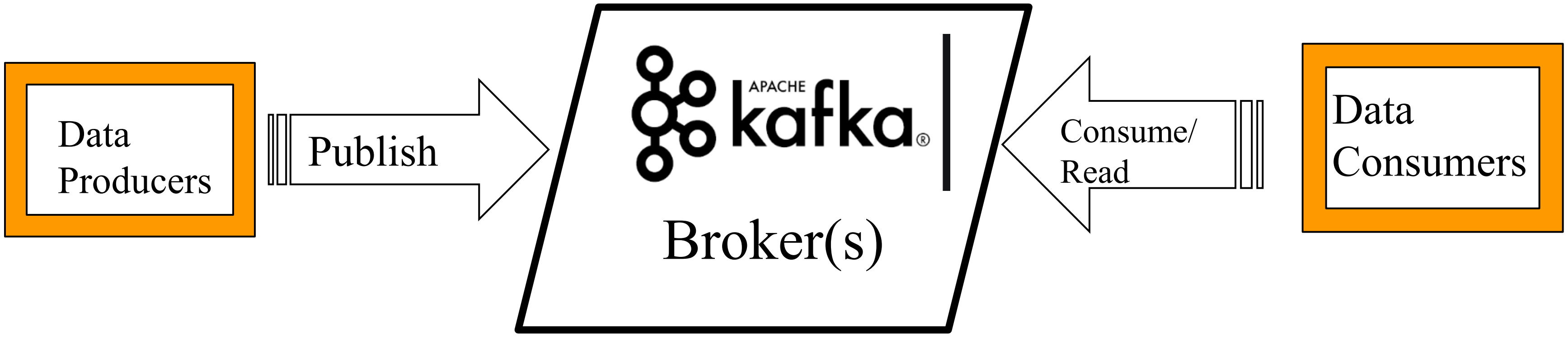


Best practice configuration on Server (Brokers)

- **Assuming 3 broker processes running on three different machines in three different availability zones (in AWS) –**
  - In this case, need to be careful of latency if putting across AZs – need to make sure the EC2 instance used have high network throughput (10 GB)
- **The following will be a typical production configuration for durability**
  - On server:
    - Replication factor = 3
    - Min-insync-replica = 2
  - On Producers:
    - Acks = all or -1 (to minimize data loss), Or acks = 0 if you care for end-to-end latency
    - Enable idempotence to be true
    - Max.inflight-requests-per-connection = 5
  - On consumers
    - Isolation level: committed
    - Commit offset of messages manually

Producers and Consumers

- **Producers and Consumers are two basic types of Kafka clients, the users of the system.**
  - Data producers produce (send/publish) messages to Kafka topics and
  - Consumers consume (subscribe/read) messages from those topics and process the messages
  - The data producer and consumer don't interact directly but use the Kafka server as an agent or broker to exchange message.
- **Kafka decouples data producers and consumers**
  - Enabling apps to produce/consume data at their own pace

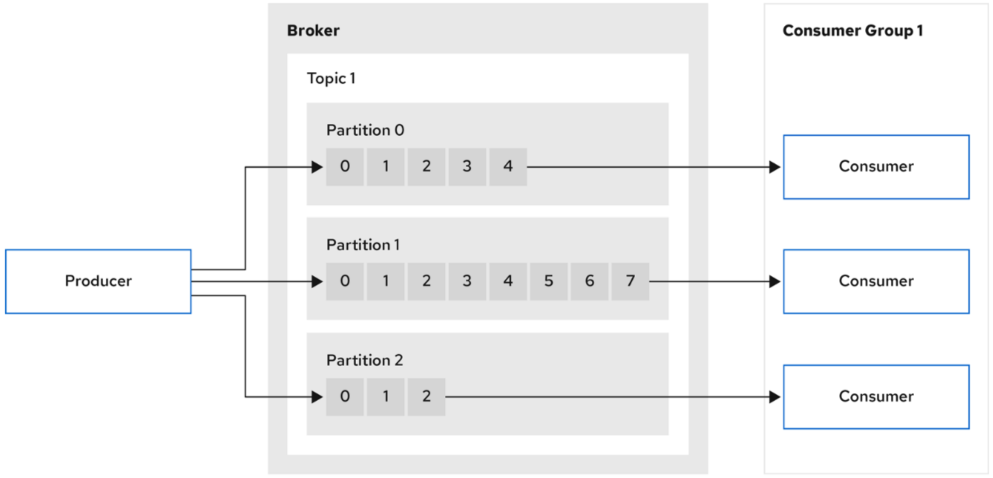


Kafka Producers

- **Producers create new messages.**
  - Producers are applications that get the data into the Kafka
- **Kafka producers** are the publishers responsible for writing records to topics.
  - Typically, this means writing a program using the KafkaProducer API.
  - Producers typically send events or records to a given topic in Kafka to a certain partition
  - Producers publish log to topics in monotonically increasing order

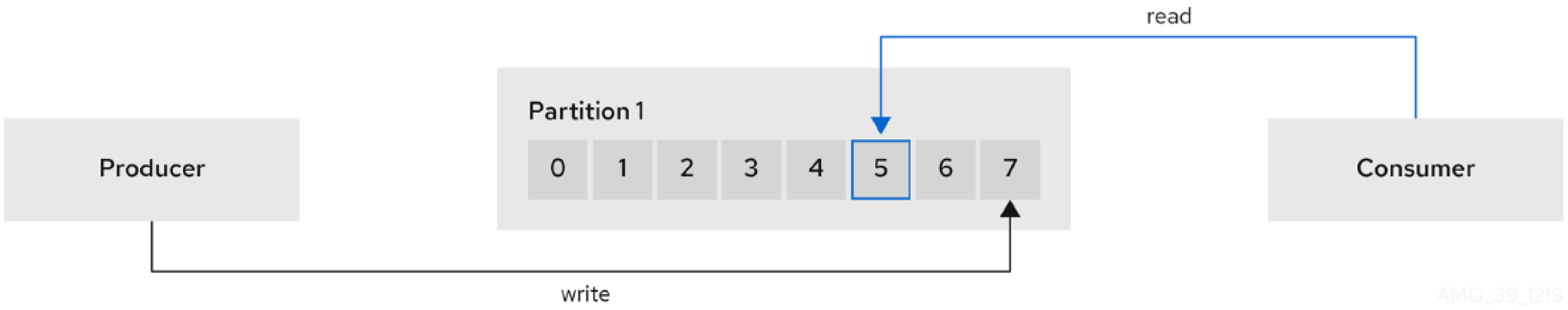
Producers

- **A message will be produced to a specific topic.**
- **By default, the producer will balance messages over all partitions of a topic evenly.**
- **In some cases, the producer will direct messages to specific partitions.**
  - This is typically done using the message key and a partitioner that will generate a hash of the key and map it to a specific partition.
  - This ensures that all messages produced with a given key will get written to the same partition.
  - The producer could also use a custom partitioner that follows other business rules for mapping messages to partitions.



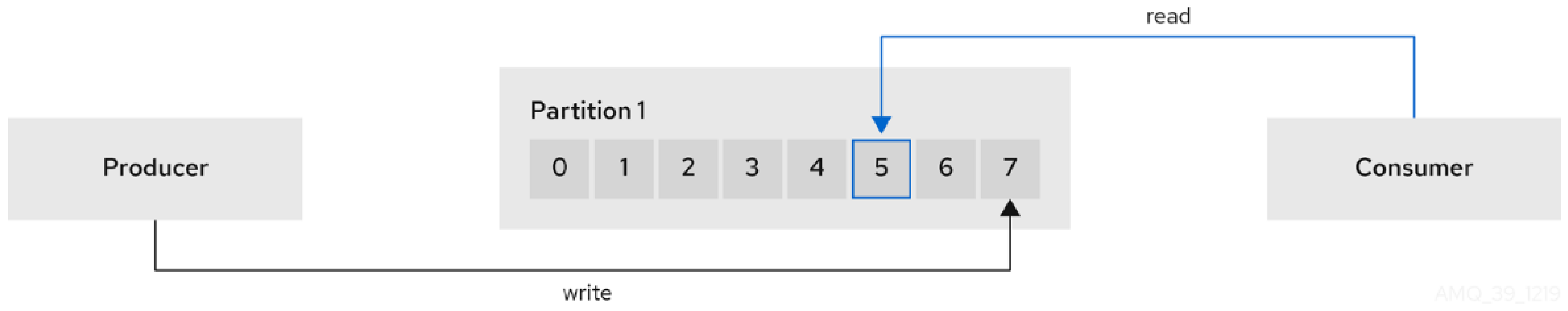
Producers

- A producer **publishes the data to a Kafka topic** to be **written to the end of a partition** using monotonically increasing sequence called Kafka **message offsets**
  - (i.e., position), starting with 0 and then incrementing with each new record log
- Kafka message offset numbers are unique within a partition



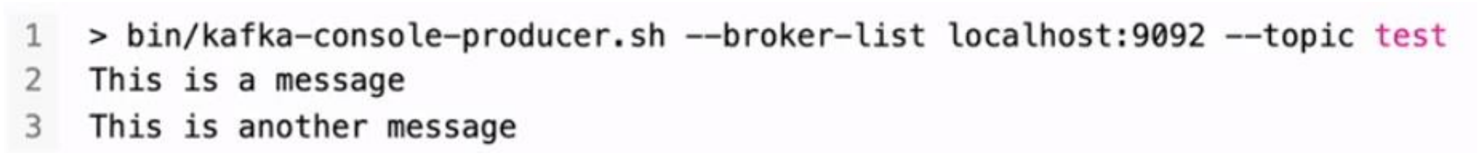
Producers

- **Message Key determines which partition your message should be written to.**
  - The key identifies the subject of the message, or a property of the message.
- **All message with same key go to the same partition.**
- **Messages without the (optional) key are sent to partitions in round-robin fashion**
- **Partitioner and Serializer are part of Producer**



Creating Kafka Producers

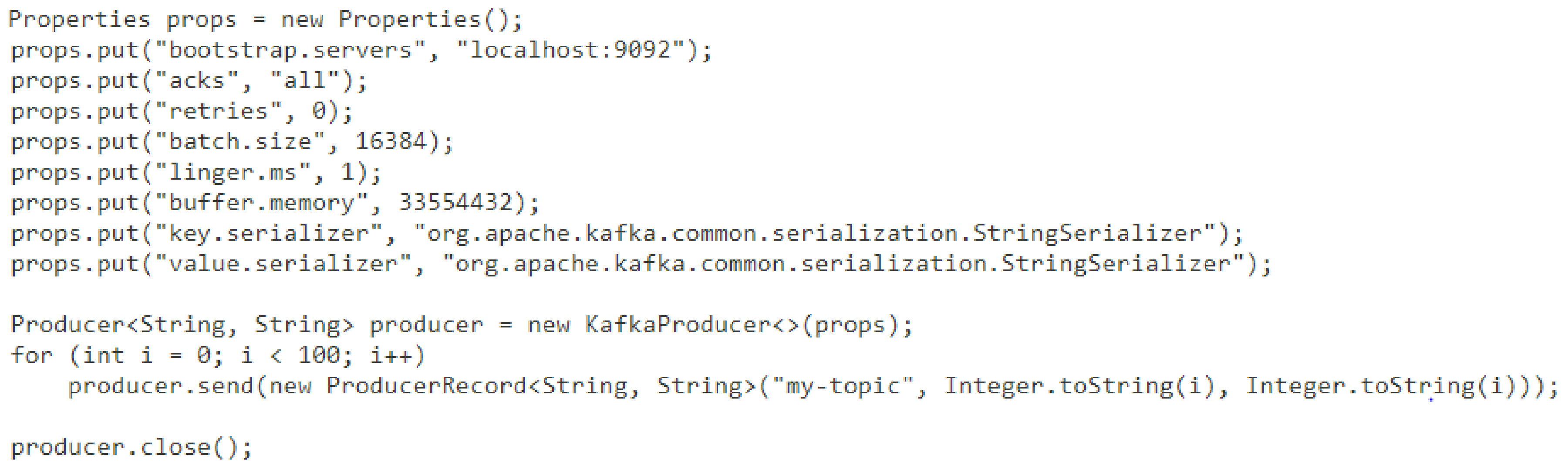
- **Kafka download comes with Kafka console producer script that can be used to start a producer and send messages to a Kafka topic**



```
> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
This is a message
This is another message
```

Creating Kafka Producers

- **Kafka also provide Java-based Producer API that can be used to write more in-depth Producer**



```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("acks", "all");
props.put("retries", 0);
props.put("batch.size", 16384);
props.put("linger.ms", 1);
props.put("buffer.memory", 33554432);
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

Producer<String, String> producer = new KafkaProducer<>(props);
for (int i = 0; i < 100; i++)
    producer.send(new ProducerRecord<String, String>("my-topic", Integer.toString(i), Integer.toString(i)));

producer.close();
```

Refer to this page for a sample producer/consumer example:
https://www.tutorialspoint.com/apache_kafka/apache_kafka_simple_producer_example.htm

Producer Configurations

- **Acknowledgement Level**
  - Producers can choose acknowledgement level from Kafka before considering the record to have been committed into the Kafka topic
- **Options Include: 0, 1, all**
  - **0** – this means producer won’t wait for any acknowledgement back from Kafka to consider the record have been committed
    - You send record to Kafka and Kafka could/could not have committed it, but producer moves on the next record and keeps producing and sending it
    - There’s a risk that the record may not have made into Kafka
    - However, if you care for end-to-end latency, then acks = 0 is way to go – it’s the fastest option to send data as you don’t have to wait for any response from Kafka

Producer Configurations

- **Options Include: 0, 1, all**
  - **1** – waits for one broker (the leader) to acknowledge that it has received the record
    - this is "at most once delivery" – may lose messages if the leader crashes and data ha not been written to replicas
  - **All** – wait for leader and all replica brokers (i.e., all in-sync replicas) to acknowledge that they have received the message. This is referred to at least once delivery
    - Although it takes longer to get the acknowledgement, it ensures the record has been saved at all in-sync replicas.
    - The broker sends acknowledgment only after replication based on the min.insync.replica property.

Producer Configurations

- **Retries: producers can choose whether to retry**
- **Retries options include**
  - 0 (do not retry) – can lose data on error
    - useful in use cases such as IOT sensor where data is being produced every second or so and you don’t mind if a second worth data is lost
  - Greater than 0 (# of retries)
    - Useful when you really care your data getting into Kafka
    - The Kafka producer sends the record to the broker and waits for a response from the broker. If no acknowledgment is received for the message sent, then the producer will retry sending the messages based on the retry target#
    - Retries might result in duplicates in error

Producer Configurations

- **Producers can choose idempotence**
  - This works by producer sending the process ID and monotonically increasing number (starting with 0) with each record to broker
  - When the broker acknowledges, it will send the combination of process ID and number
  - Broker knows what acknowledgement is going for which message, so it won’t send the same message twice
  - Enabler of exactly-once semantics!

Producer Configurations

- **Exactly Once Semantics**
  - Ensures only once delivery of data from producers to Kafka
  - Prevents duplicate messages from being produced by client applications (idempotent producers)
  - Very useful to write real-time, mission-critical streaming applications that require guarantees that data is processed "exactly once"
  - Brings strong transactional guarantees to Kafka
    - Ensures messages in a transaction are all consumed or none are consumed (atomic messages)
  - Sample use cases:
    - Tracking orders for billing
    - Processing financial transactions, such as Credit Card
    - Tracking inventory in the supply chain

Producer Configurations

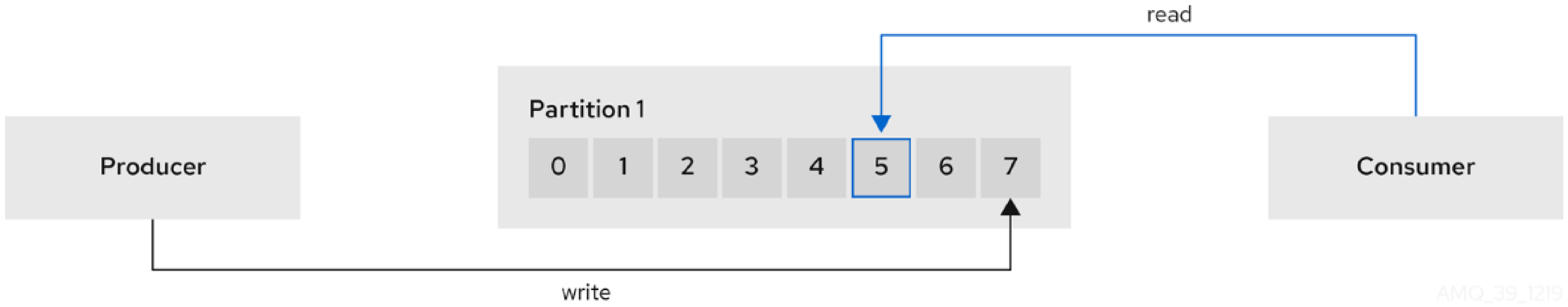
- **Exactly Once Semantics configuration requires the following:**
  - On producer,
    - Enable idempotence on producer – that is as part of the producer property, enable idempotent to be true
    - Start transaction in producer code
    - Every message will have 3 things: ProducerID, SequenceID, and TransactionID
  - On consumer
    - Set isolation level to committed, that is read only committed messages (i.e., when leader and all followers are in in-sync-replica list)
    - Commit offset of messages manually
  - Kafka will avoid taking duplicate messages

Producer Configurations

- **Producers can batch records**
  - Producer can send records as they occur or can batch them to a configurable size (using batch.size property and setting it a value, e.g., 5000) and then send them together
  - Grouping records together reduces network traffic
- **Producers can compress records**
  - Compression reduces size of records to be sent across the network
  - Can use compression.type property
  - A number of compression techniques available, such gzip

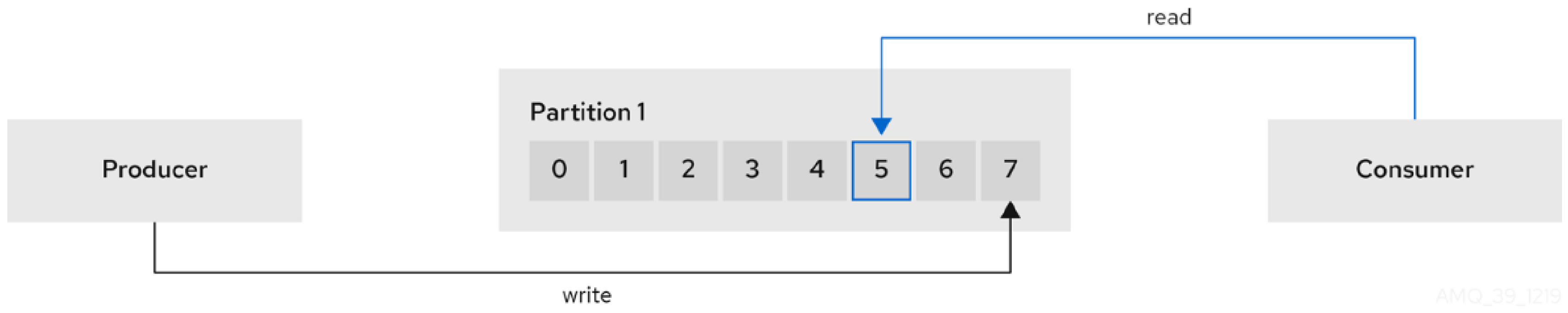
Consumers

- **Consumers read messages.**
- **The consumer subscribes to one or more topics**
  - and reads the messages in the order in which they were produced to each partition.
- **A consumer subscribes to a topic and reads messages from topic partition using offset.**
  - The consumer keeps track of which messages it has already consumed by keeping track of the offset of messages.
  - The offset—an integer value that continually increases—is another piece of metadata that Kafka adds to each message as it is produced.
  - Each message in a given partition has a unique offset, and the following message has a greater offset.
  - By storing the next possible offset for each partition, typically in Kafka itself, a consumer can stop and restart without losing its place.



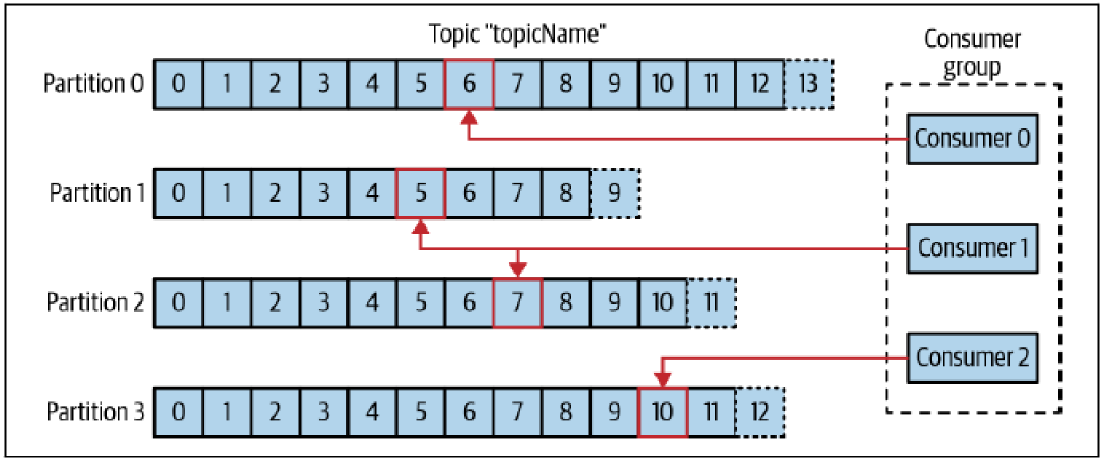
Consumers

- Data is not pushed to consumers, they poll Kafka topics and read
- When a consumer finishes reading the data, the data still stays in topics/partitions



Consumers

- **Consumers work as part of a consumer group,**
  - which is one or more consumers that work together to consume a topic.
  - Multiple instances of consumers to parallelize the read
  - The group ensures that each partition is only consumed by one member.
  - Consumers within a consumer group do not read data from the same partition but can receive data from one or more partitions.
    - In Figure, there are three consumers in a single group consuming a topic.
    - Two of the consumers are working from one partition each, while the third consumer is working from two partitions.

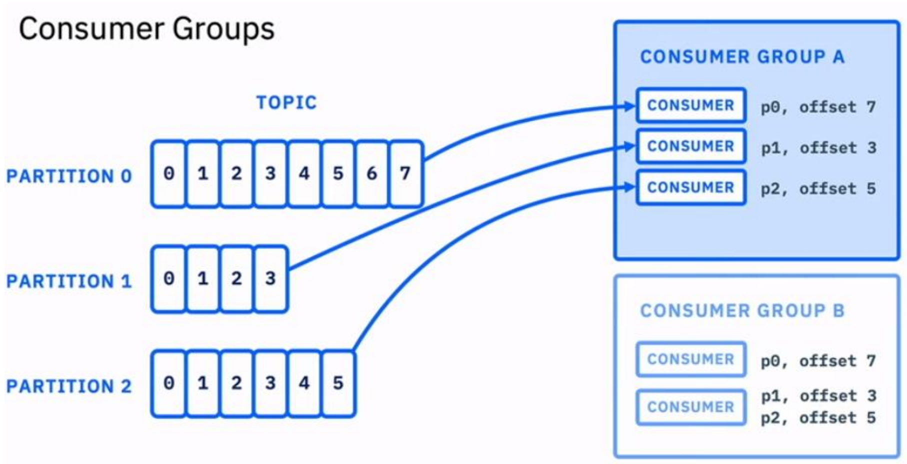


Consumer Group

- **Consumer groups are way for consumers to work together to consume in parallel from a topic**
  - A consumer group has multiple consumers
  - Each consumer of the same consumer group consumes from a specific partition of a topic

Consumer Group

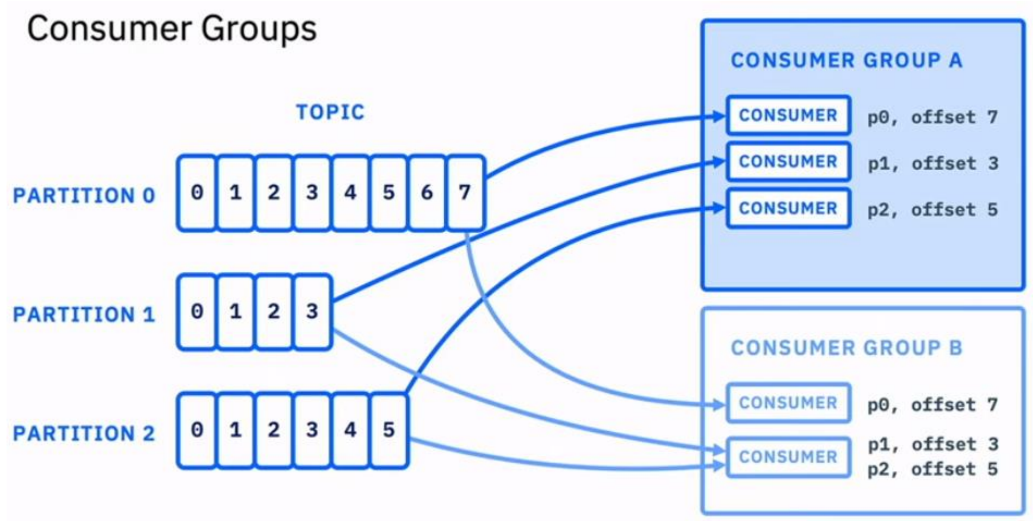
- **In the example below, Consumer Group A has three consumers and they are reading from different partitions from different offsets**
  - This allows three consumers to read in parallel
  - If there was only one consumer in the group, then it will read from the first partition, then from the second partition, and then from the third partition
  - Consumer Group should contain as many consumers as there are partitions in a topic you are consuming from



Source: Google

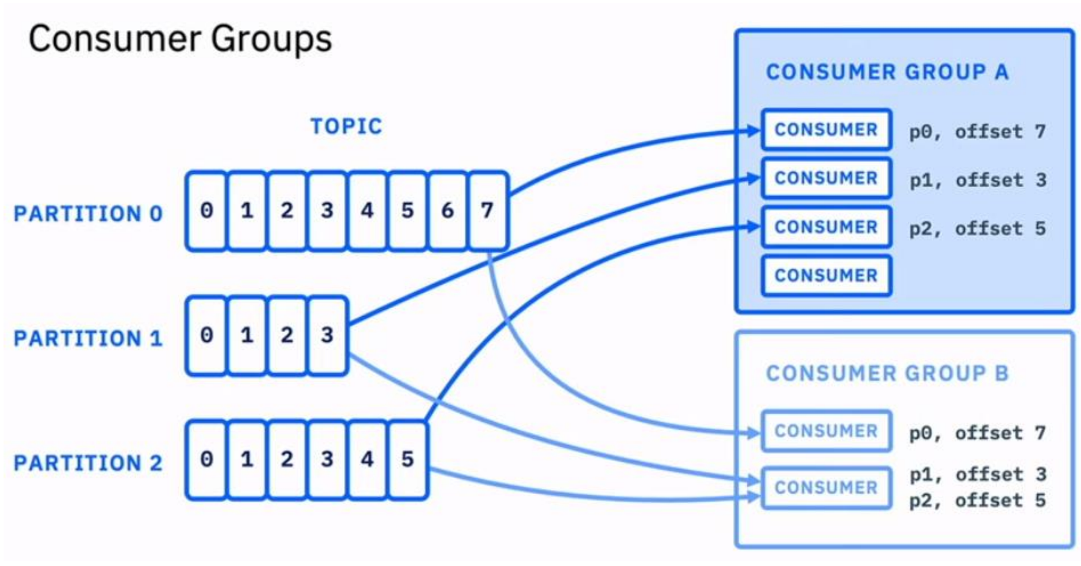
Consumer Group

- **Consumer Group B has only two consumers**
- **When trying to split the load between consumers, one consumer end up having two partitions – this is not the ideal and it will impact consumer's performance**
  - Ideally, consumer group should contain as many consumers as there are partitions in a topic you are consuming from



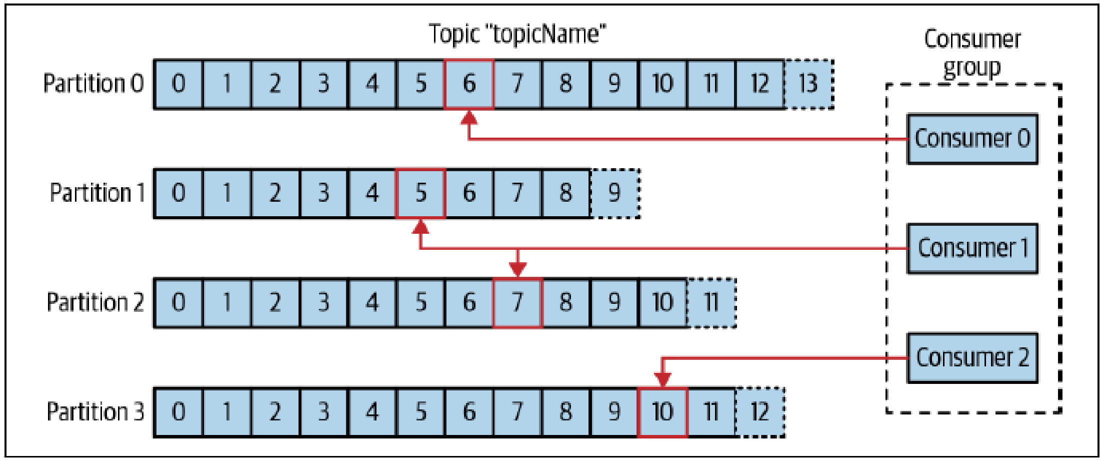
Consumer Group

- You can *also* add another consumer in Consumer Group A that can be used as a standby consumer
- If one of the consumers goes down, the standby consumer could pick up where the other consumer left off



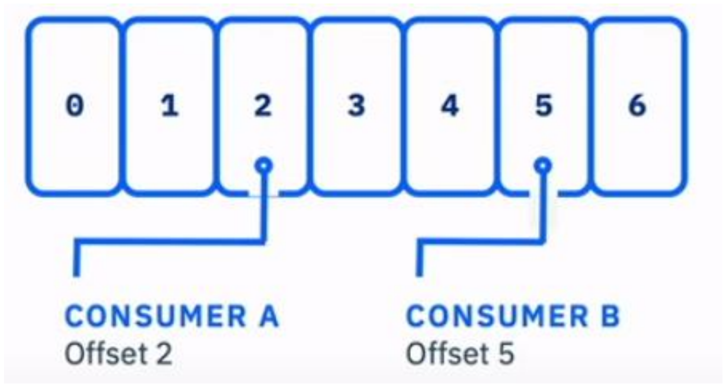
Consumers

- **Consumers in a consumer group are instances of the same application with the same application ID.**
  - Consumers are grouped using a group.id, allowing messages to be spread across the members.
  - Consumers can horizontally scale to consume topics with a large number of messages.
  - Additionally, if a single consumer fails, the remaining members of the group will reassign the partitions being consumed to take over for the missing member.



Consumers

- **Consume messages from a topic – consumers read records from a given certain offset**
  - Consumers reads and makes copy of the data for itself and the records still persist at the topic as the immutable logs are preserved for a configurable period of time.
  - This enables multiple consumers to read from the same topic at the same time

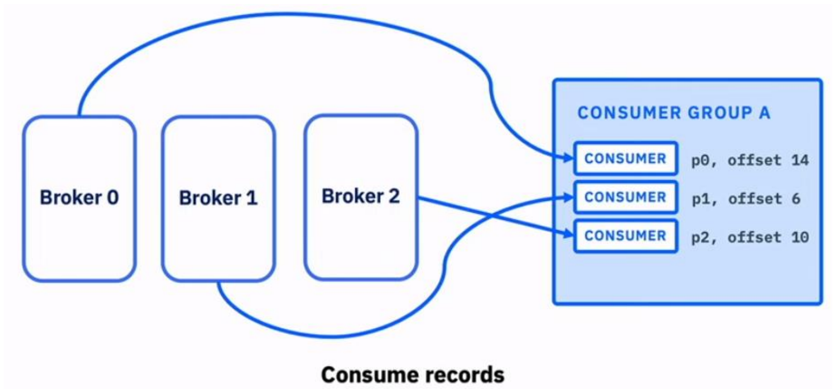


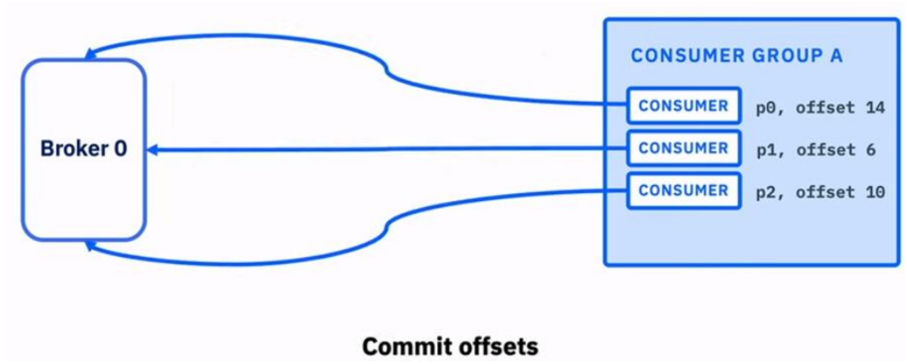
Consumer offsets

- **Offsets describe the position of messages within a partition.**
- **Each message in a given partition has a unique offset,**
  - helps identify the position of a consumer within the partition to track the number of records that have been consumed.
- **Committed offsets are written to an offset commit log.**
  - __consumer_offsets topic stores information on committed offsets, the position of last and next offset, according to consumer group.
- **Each topic has a leader (e.g., broker 0 in the example), which gets set to be the group coordinator**
- **Consumers commit their offsets (the last position read) to this group coordinator broker**

Consumer offsets

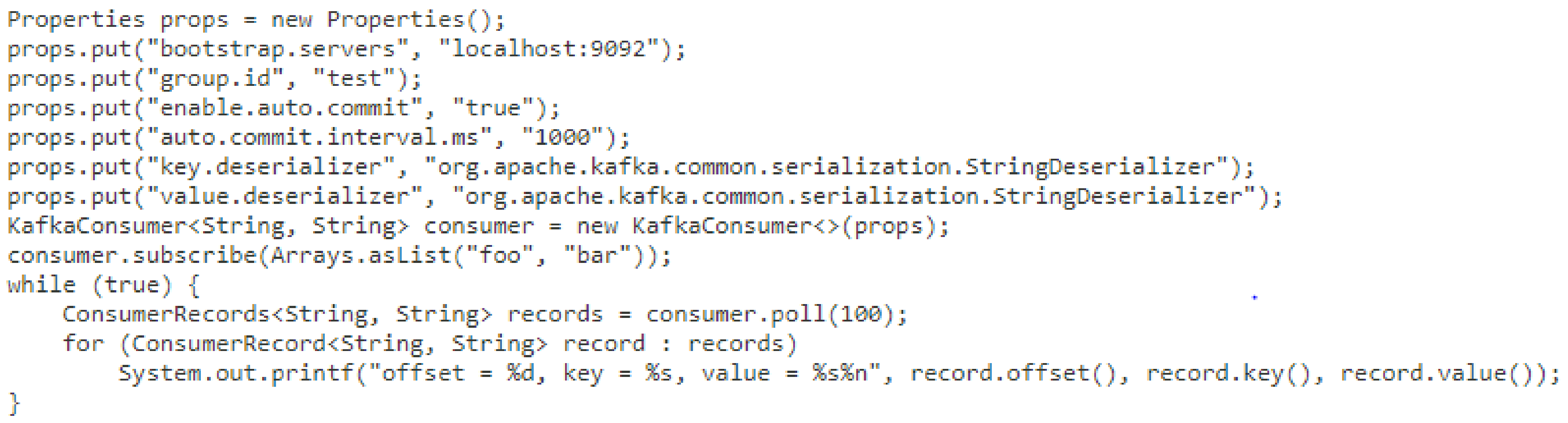
- **Consumers consume from different partitions of the topic from different brokers and then commit their offsets back to the group coordinator (broker 0)**
  - This allows to add new consumers if any one of the existing consumers fail





Kafka Consumers

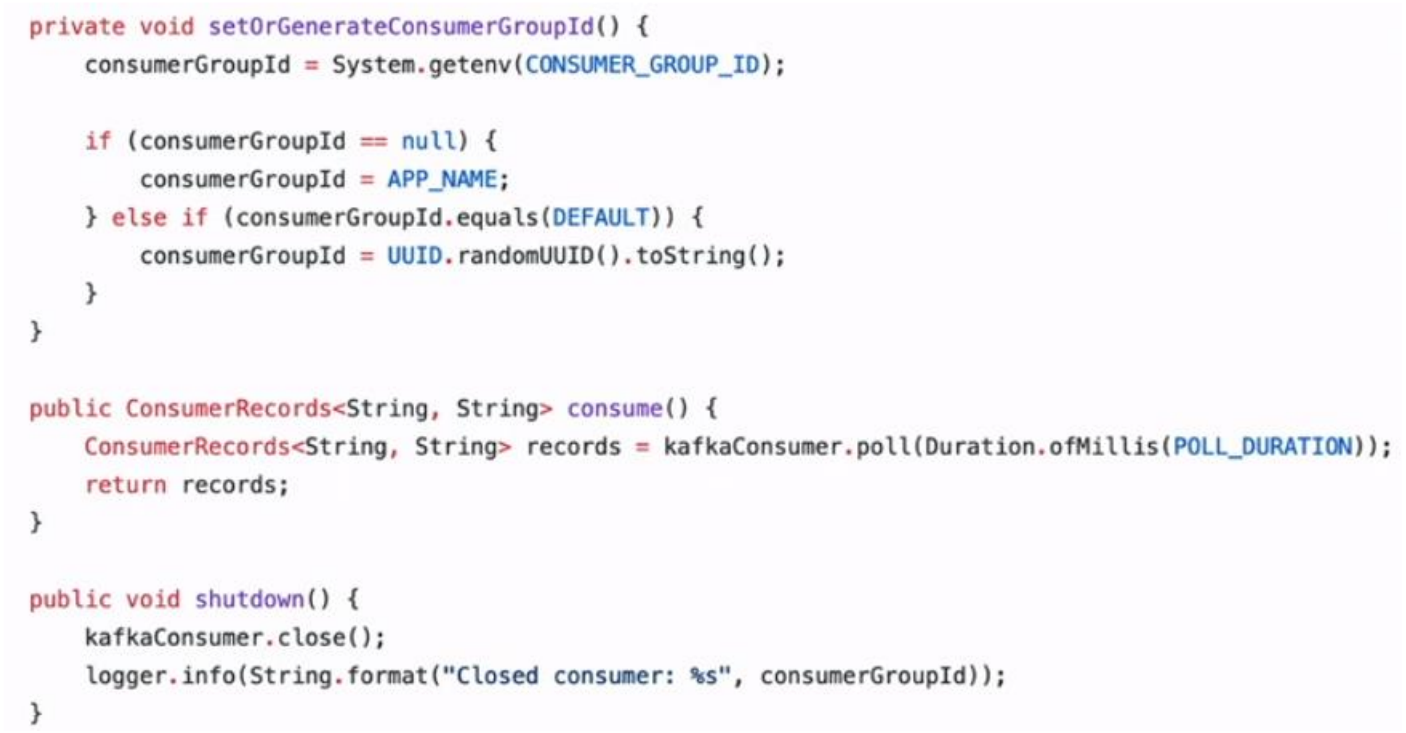
- **Can write Consumers using Java APIs**
  - Create and configure Properties object and then use that to instantiate Consumer object



```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "test");
props.put("enable.auto.commit", "true");
props.put("auto.commit.interval.ms", "1000");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("foo", "bar"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
        System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(), record.key(), record.value());
}
```

Kafka Consumers

- **Can write Consumers using Java APIs**
  - setOrGenerateConsumerGroupId() allows to be part of same group using same consumer group ID and share the load of consuming of the topic



```
private void setOrGenerateConsumerGroupId() {
    consumerGroupId = System.getenv(CONSUMER_GROUP_ID);

    if (consumerGroupId == null) {
        consumerGroupId = APP_NAME;
    } else if (consumerGroupId.equals(DEFAULT)) {
        consumerGroupId = UUID.randomUUID().toString();
    }
}

public ConsumerRecords<String, String> consume() {
    ConsumerRecords<String, String> records = kafkaConsumer.poll(Duration.ofMillis(POLL_DURATION));
    return records;
}

public void shutdown() {
    kafkaConsumer.close();
    logger.info(String.format("Closed consumer: %s", consumerGroupId));
}
```

Producers and Consumers

- **There are also advanced client APIs—**
  - Kafka Connect API for data integration and
  - Kafka Streams for stream processing.
    - A stream is considered to be a single topic of data, regardless of the number of partitions.
    - This represents a single stream of data moving from the producers to the consumers.
  - The advanced clients use producers and consumers as building blocks and provide higher-level functionality on top.

Strimzi

- **An open source project that focuses on running Apache Kafka on Kubernetes. It provides:**
  - Container images for Apache Kafka, Apache ZooKeeper
  - Operators for deploying, managing and configuring Kafka clusters on Kubernetes
- **Provides a Kubernetes-native experience**
  - Not only Kafka clusters, but also KafakUsers, KafkaTopics and the rest of the Kafka ecosystem deployed can be deployed on Kubernetes
- **Strimzi supports Kafka using *Operators* to deploy and manage the components and dependencies of Kafka to Kubernetes.**

Strimzi Kafka

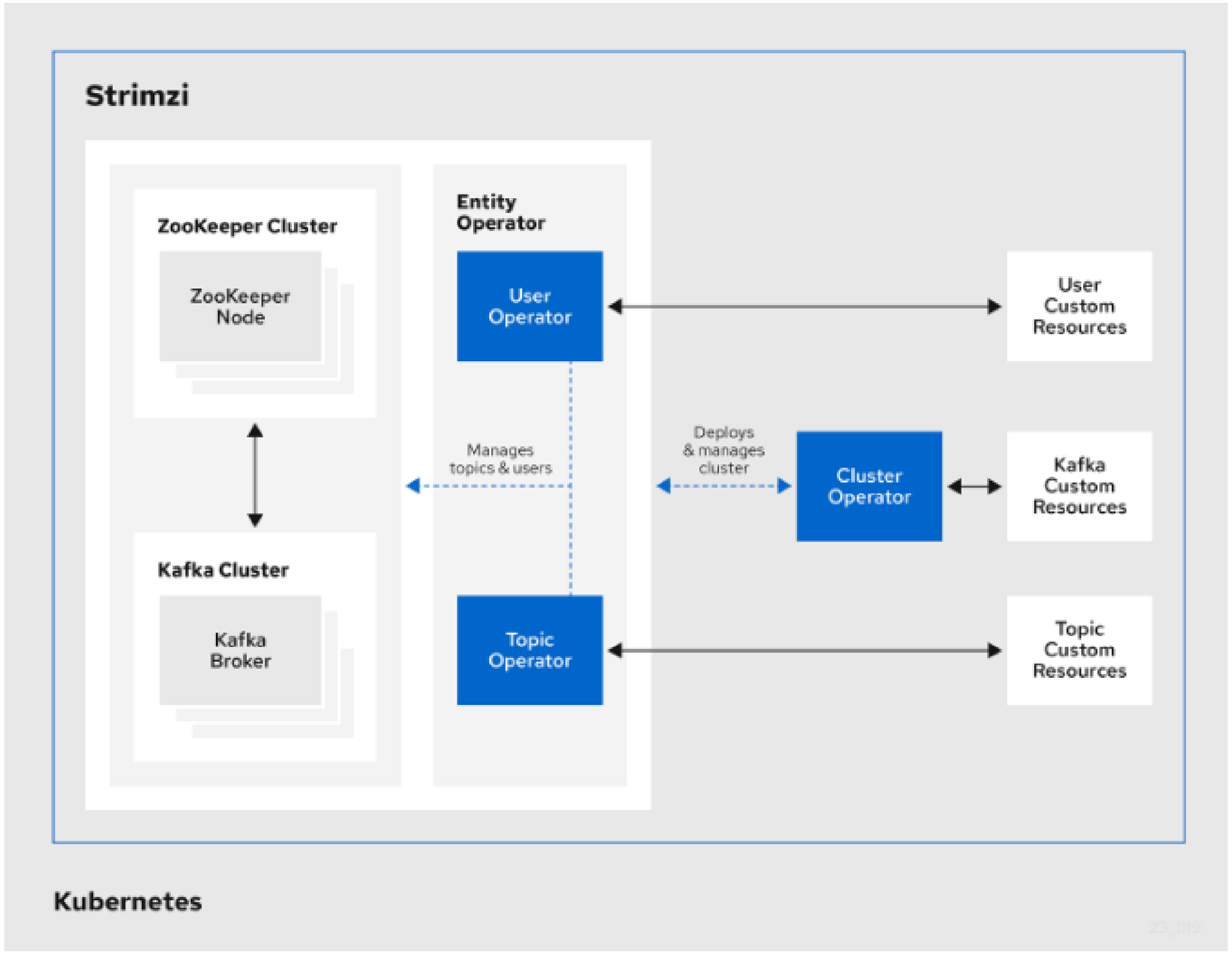
- **Strimzi Operators are designed to effectively manage Kafka on Kubernetes and simplify the process of:**
  - Deploying and running Kafka clusters/components
  - Configuring access to Kafka
  - Securing access to Kafka
  - Upgrading Kafka
  - Managing brokers
  - Creating and managing topics
  - Creating and managing users – but allows to use your own user management separately using PKI etc.

Strimzi Operators for managing Kafka cluster on Kubernetes cluster

- **Cluster Operator**
  - Deploys and manages Apache Kafka clusters, Kafka Connect, Kafka MirrorMaker, Kafka Bridge, Kafka Exporter, and the Entity Operator
  - The Cluster Operator deploys the Entity Operator configuration at the same time as a Kafka cluster.
- **Entity Operator**
  - Comprises the Topic Operator and User Operator
  - **Topic Operator**
    - Manages Kafka topics
  - **User Operator**
    - Manages Kafka users

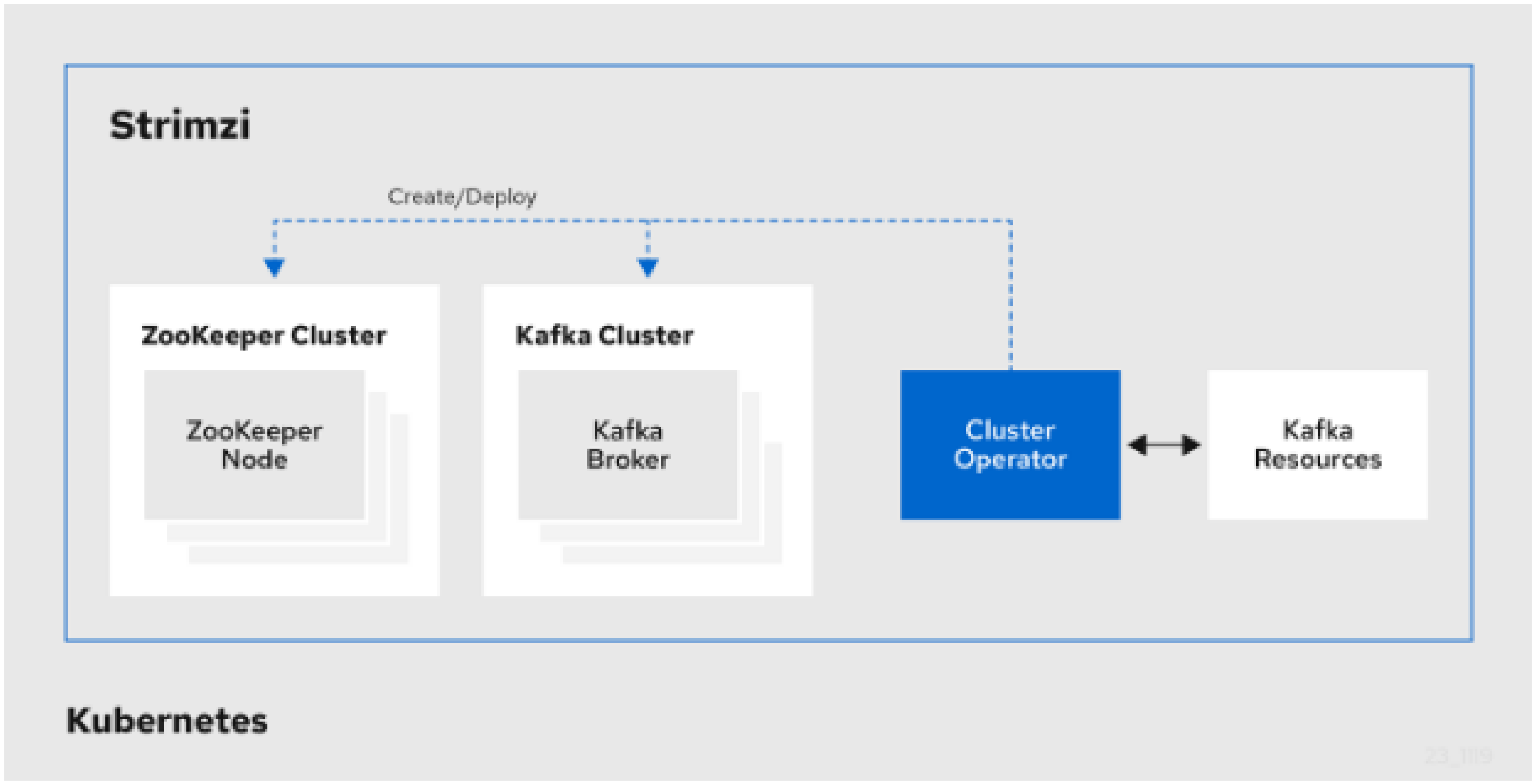
Strimzi Operators for managing Kafka cluster on Kubernetes cluster

- Operators within the Strimzi architecture



Strimzi Operators for managing Kafka cluster on Kubernetes cluster

- Example architecture for the Cluster Operator

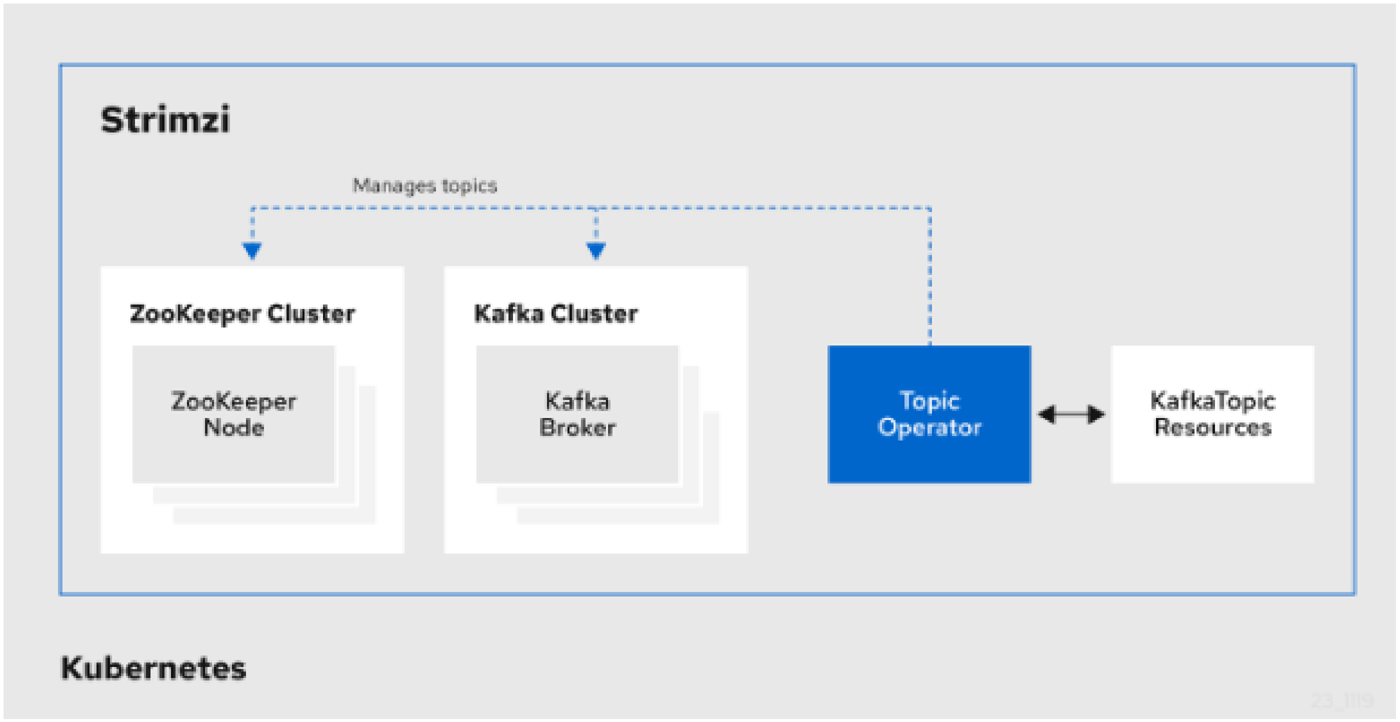


Strimzi Operators for managing Kafka cluster on Kubernetes cluster

- **Strimzi uses the Cluster Operator to deploy and manage clusters for:**
  - Kafka (including ZooKeeper, Entity Operator, Kafka Exporter, and Cruise Control)
  - Kafka Connect
  - Kafka MirrorMaker
  - Kafka Bridge
- **Custom resources are used to deploy the clusters. For example, to deploy a Kafka cluster:**
  - A Kafka resource with the cluster configuration is created within the Kubernetes cluster.
  - The Cluster Operator deploys a corresponding Kafka cluster, based on what is declared in the Kafka resource.
- **The Cluster Operator can also deploy (through configuration of the Kafka resource):**
  - A Topic Operator to provide operator-style topic management through KafkaTopic custom resources
  - A User Operator to provide operator-style user management through KafkaUser custom resources

Strimzi Operators for managing Kafka cluster on Kubernetes cluster

- Example architecture for the Topic Operator



Strimzi Custom Resources

- **Strimzi uses Custom Resource Definitions (CRDs) to deploy Kafka components on a Kubernetes cluster**
  - Strimzi makes the deployment of Kafka components to a K8s cluster highly configurable
  - CRDs have been created by extending Kubernetes resources using Kubernetes extensible APIs
  - CRDs act as configuration instructions to describe the custom resources in a Kubernetes cluster, and are provided with Strimzi for each Kafka component used in a deployment, as well as users and topics.
  - CRDs and custom resources are defined as YAML files. Example YAML files are provided with the Strimzi distribution.
  - CRDs also allow Strimzi resources to benefit from native Kubernetes features like CLI accessibility and configuration validation.

Strimzi Custom Resources

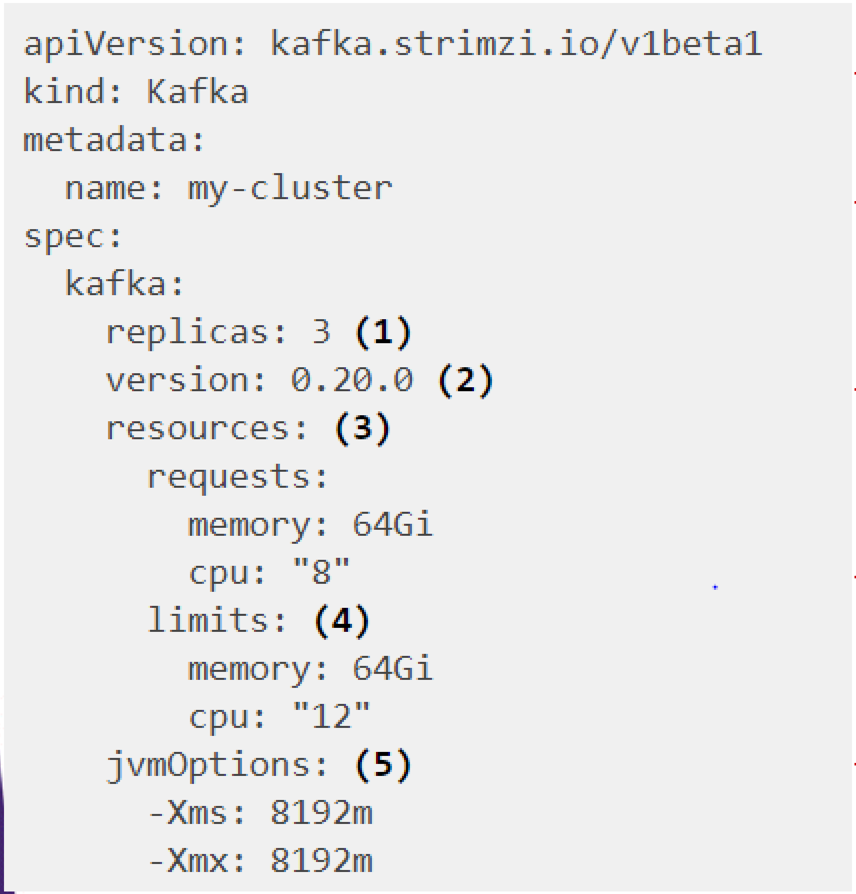
- **Examples of CRDs that serve as a Kubernetes native resource:**
  - **Kafka** for creating Kafka cluster
  - **KafkaUser** and **KafkaTopic** for handing users and topics
  - **KafkaConnect** and **KafkaConnector** for handing a Kafka Connect deployment
  - **KafkaBridge** for enabling HTTP access to the cluster
  - **KafkaMirrorMaker** and **KafkaMirrorMaker2** for mirroring data across clusters
  - **KafkaRebalance** for rebalancing the cluster through Cruise Control

Sample Kafka YAML Configuration

- **The sample shows only some of the important configuration options**
  - **Resource requests** (CPU/Memory)
  - **JVM options** for maximum and minimum memory allocation
  - **Listeners** (and authentication)
  - **Authentication**
  - **Storage**
  - **Rack awareness**
  - **Metrics**

Sample Kafka YAML Configuration

- Custom Resource: Kafka cluster

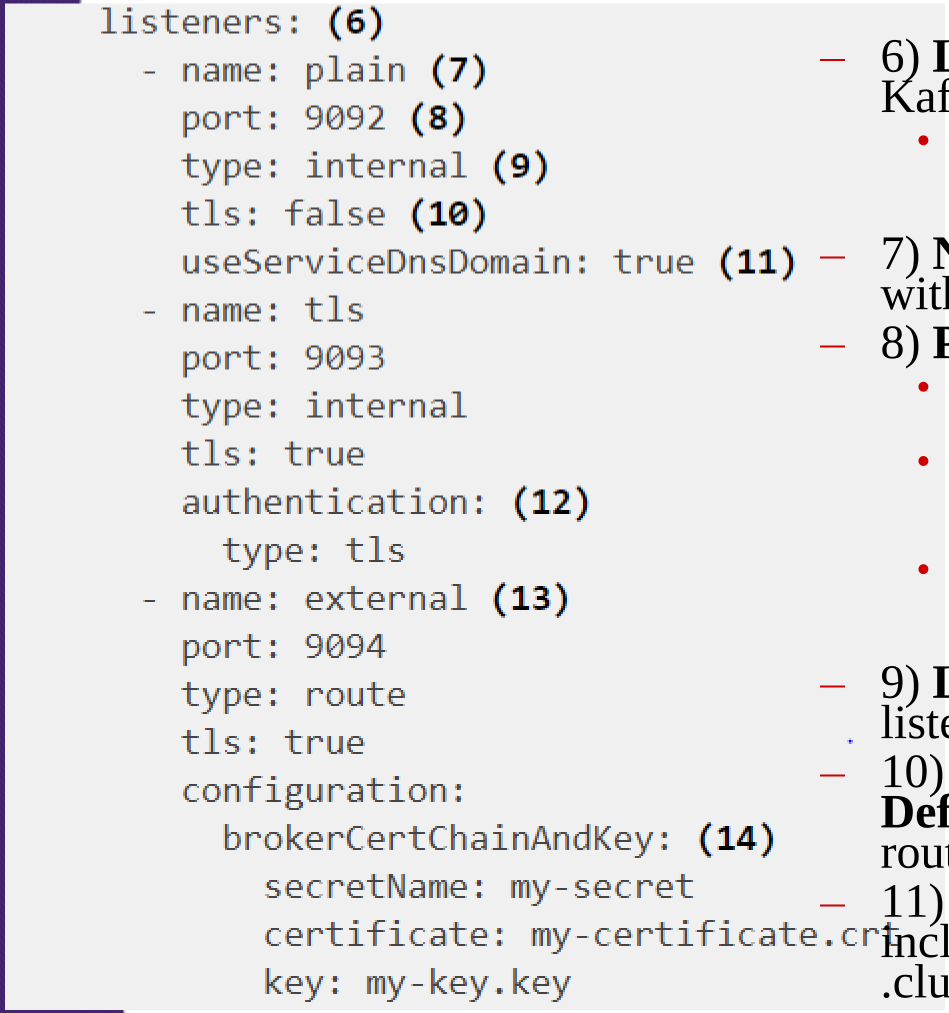


```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    replicas: 3 (1)
    version: 0.20.0 (2)
    resources: (3)
      requests:
        memory: 64Gi
        cpu: "8"
      limits: (4)
        memory: 64Gi
        cpu: "12"
    jvmOptions: (5)
      -Xms: 8192m
      -Xmx: 8192m
```

- 1) **Replicas** specifies the number of broker nodes.
- 2) Kafka **version**, which can be changed by following the upgrade procedure.
- 3) **Resource requests** specify the resources to reserve for a given container.
- 4) **Resource limits** specify the maximum resources that can be consumed by a container.
- 5) **JVM options** can specify the minimum (-Xms) and maximum (-Xmx) memory allocation for JVM.

Sample Kafka YAML Configuration

- Custom Resource: Kafka cluster

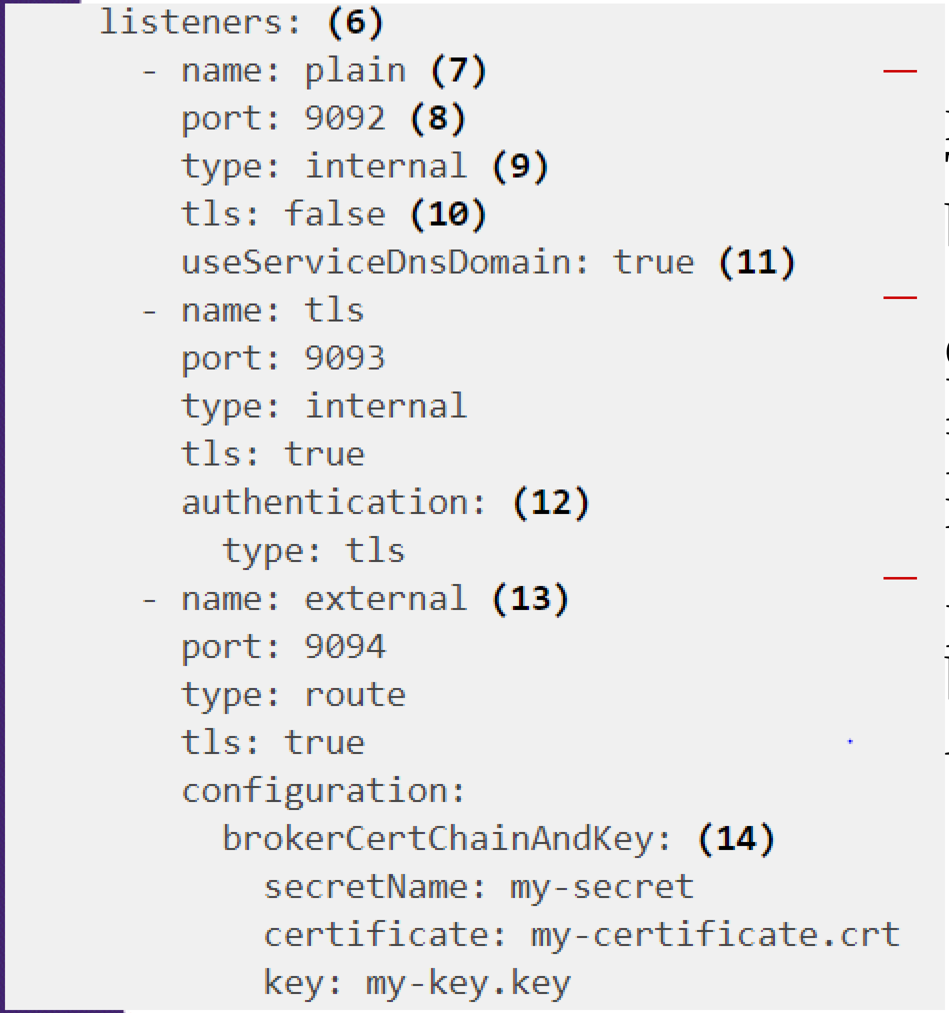


```
listeners: (6)
  - name: plain (7)
    port: 9092 (8)
    type: internal (9)
    tls: false (10)
    useServiceDnsDomain: true (11)
  - name: tls
    port: 9093
    type: internal
    tls: true
    authentication: (12)
      type: tls
  - name: external (13)
    port: 9094
    type: route
    tls: true
    configuration:
      brokerCertChainAndKey: (14)
        secretName: my-secret
        certificate: my-certificate.cr
        key: my-key.key
```

– 6) **Listeners** configure how clients connect to the Kafka cluster via bootstrap addresses.
  - Listeners are configured as **internal** or **external** listeners for connection inside or outside the Kubernetes cluster.
– 7) **Name** to identify the listener. Must be unique within the Kafka cluster.
– 8) **Port number** used by the listener inside Kafka.
  - The port number has to be unique within a given Kafka cluster.
  - Allowed port numbers are 9092 and higher with the exception of ports 9404 and 9999, which are already used for Prometheus and JMX.
  - Depending on the listener type, the port number might not be the same as the port number that connects Kafka clients.
– 9) **Listener type** specified as internal, or for external listeners, as route, loadbalancer, nodeport or ingress.
– 10) **Enables TLS** encryption for each listener. **Default** is **false**. TLS encryption is not required for route listeners.
– 11) Defines whether the fully-qualified DNS names including the cluster service suffix (usually .cluster.local) are assigned.

Sample Kafka YAML Configuration

- **Custom Resource: Kafka cluster**



```
listeners: (6)
  - name: plain (7)
    port: 9092 (8)
    type: internal (9)
    tls: false (10)
    useServiceDnsDomain: true (11)
  - name: tls
    port: 9093
    type: internal
    tls: true
    authentication: (12)
      type: tls
  - name: external (13)
    port: 9094
    type: route
    tls: true
    configuration:
      brokerCertChainAndKey: (14)
        secretName: my-secret
        certificate: my-certificate.crt
        key: my-key.key
```

- 12) **Listener authentication** mechanism specified as mutual TLS, SCRAM-SHA-512 or token-based OAuth 2.0.
- 13) **External listener configuration** specifies how the Kafka cluster is exposed outside Kubernetes, such as through a route, **loadbalancer** or **nodeport**.
- **14) Optional configuration** for a Kafka listener certificate managed by an external Certificate Authority.
  - The **brokerCertChainAndKey** property specifies a Secret that holds a server certificate and a private key. Kafka listener certificates can also be configured for TLS listeners.

Sample Kafka YAML Configuration

- **Custom Resource: Kafka cluster**

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    replicas: 3 (1)
    version: 0.20.0 (2)
    resources: (3)
      requests:
        memory: 64Gi
        cpu: "8"
      limits: (4)
        memory: 64Gi
        cpu: "12"
    jvmOptions: (5)
      -Xms: 8192m
      -Xmx: 8192m
```

```
    listeners: (6)
      - name: plain (7)
        port: 9092 (8)
        type: internal (9)
        tls: false (10)
        useServiceDnsDomain: true (11)
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication: (12)
          type: tls
      - name: external (13)
        port: 9094
        type: route
        tls: true
        configuration:
          brokerCertChainAndKey: (14)
            secretName: my-secret
            certificate: my-certificate.crt
            key: my-key.key
```

Sample Kafka YAML Configuration

- **Custom Resource: Kafka cluster**

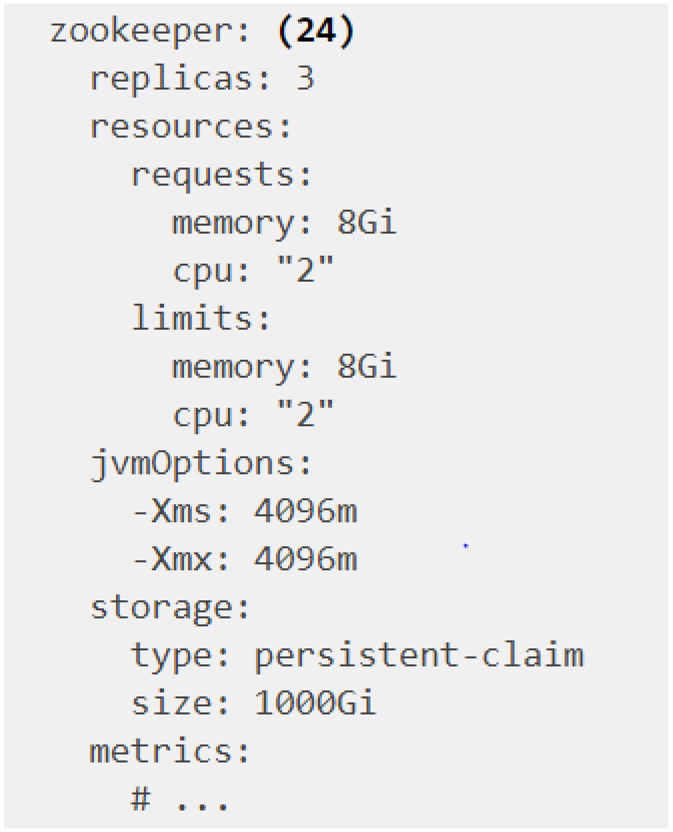
```
  authorization: (15)
    type: simple
  config: (16)
    auto.create.topics.enable: "false"
    offsets.topic.replication.factor: 3
    transaction.state.log.replication.factor: 3
    transaction.state.log.min.isr: 2
    ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" (17)
    ssl.enabled.protocols: "TLSv1.2"
    ssl.protocol: "TLSv1.2"
  storage: (18)
    type: persistent-claim (19)
    size: 10000Gi (20)
  rack: (21)
    topologyKey: topology.kubernetes.io/zone
  metrics: (22)
    lowercaseOutputName: true
    rules: (23)
    # Special cases and very specific rules
    - pattern : kafka.server<type=(.+), name=(.+), clientId=(.+), topic=(.+), partition=(.*
      name: kafka_server_$1_$2
      type: GAUGE
      labels:
        clientId: "$3"
        topic: "$4"
        partition: "$5"
```

Sample Kafka YAML Configuration

- **Custom Resource: Kafka cluster**
  - 15) **Authorization** enables simple, OAUTH 2.0 or OPA authorization on the Kafka broker.
    - **Simple** authorization uses the AclAuthorizer Kafka plugin.
  - 16) **Config** specifies the broker configuration.
    - Standard Apache Kafka configuration may be provided, restricted to those properties not managed directly by Strimzi.
  - 17) SSL properties for external listeners to run with a specific cipher suite for a TLS version.
  - 18) **Storage** is configured as **ephemeral**, **persistent-claim** or **jbod**.
  - 19) **Storage size** for persistent volumes may be increased and additional volumes may be added to JBOD storage.
  - 20) **Persistent storage** has **additional configuration** options, such as a storage id and class for dynamic volume provisioning.
  - 21) **Rack awareness** is configured to spread replicas across different racks. A topology key must match the label of a cluster node.
  - 22) **Kafka metrics configuration** for use with Prometheus.
  - 23) **Kafka rules** for exporting metrics to a Grafana dashboard through the JMX Exporter.
    - A set of rules provided with Strimzi may be copied to your Kafka resource configuration.

Sample Kafka YAML Configuration

- **Custom Resource: Kafka cluster**

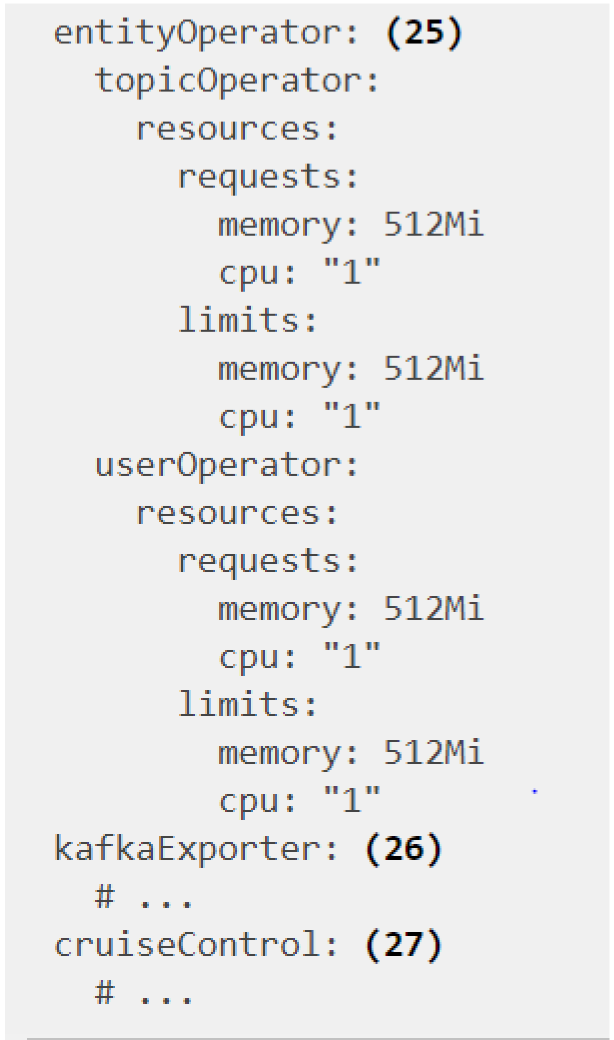


```
zookeeper: (24)
  replicas: 3
  resources:
    requests:
      memory: 8Gi
      cpu: "2"
    limits:
      memory: 8Gi
      cpu: "2"
  jvmOptions:
    -Xms: 4096m
    -Xmx: 4096m
  storage:
    type: persistent-claim
    size: 1000Gi
  metrics:
    # ...
```

  – 24) **ZooKeeper-specific** configuration, which contains properties similar to the Kafka configuration.

Sample Kafka YAML Configuration

- **Custom Resource: Kafka cluster**



```
entityOperator: (25)
  topicOperator:
    resources:
      requests:
        memory: 512Mi
        cpu: "1"
      limits:
        memory: 512Mi
        cpu: "1"
  userOperator:
    resources:
      requests:
        memory: 512Mi
        cpu: "1"
      limits:
        memory: 512Mi
        cpu: "1"
kafkaExporter: (26)
  # ...
cruiseControl: (27)
  # ...
```

  – 25) **Entity Operator** configuration, which specifies the configuration for the **Topic Operator** and **User Operator.**
  – 26) **Kafka Exporter** configuration, which is used to expose data as Prometheus metrics.
  – 27) **Cruise Control** configuration, which is used to rebalance the Kafka cluster.

Sample Kafka YAML Configuration

- **Kafka Topic Custom resource**

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaTopic (1)
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster (2)
spec: (3)
  partitions: 1
  replicas: 1
  config:
    retention.ms: 7200000
    segment.bytes: 1073741824
```

1) The **kind** and **apiVersion** identify the CRD of which the custom resource is an instance.

2) A **label**, applicable only to KafkaTopic and KafkaUser resources, that **defines** the **name** o the **Kafka cluster** to which a topic or user belongs.

3) The spec shows the **number** of **partitions** and **replicas** for the topic as well as the configuration parameters for the topic itself.

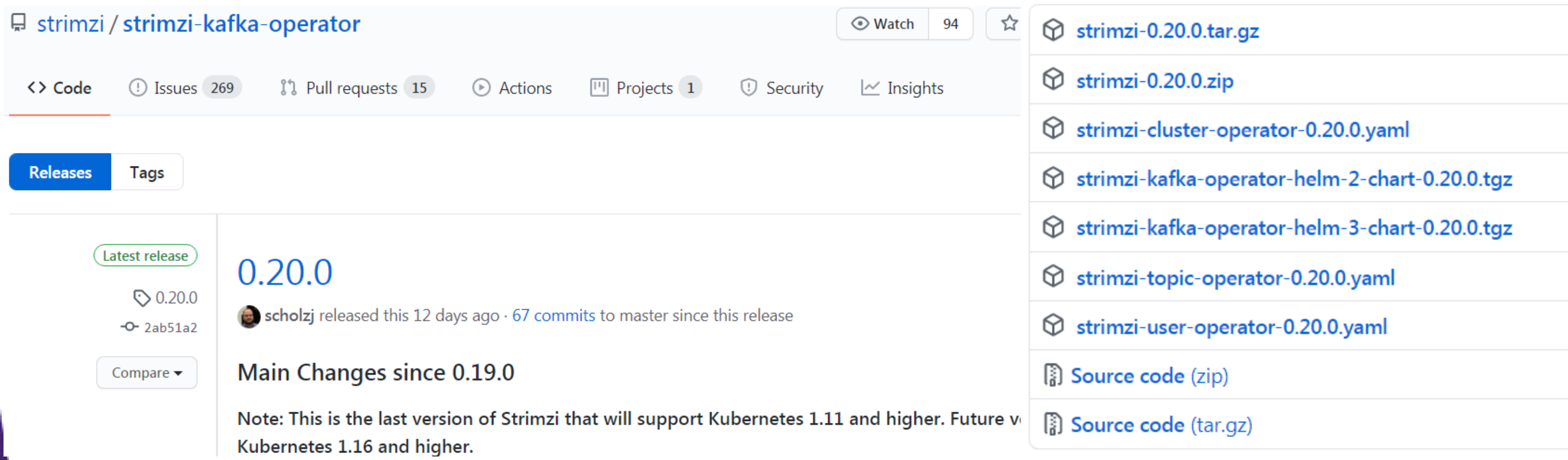
- In this example, the **retention period** for a message to remain in the topic and the **segment file size** for the log are specified.

Strimzi Kafka

- **Strimzi provides a Kubernetes native experience for Kafka**
  - In Kubernetes, there are a number of Kubernetes native resources/objects, such as Pods, Deployment, StatefulSet, ConfigMap, Secret etc. that users can use today
  - Strimzi provides custom resource definitions, by extending Kubernetes APIs, to provide Kubernetes native experience for Kafka

How to install Strimzi Kafka

- **Download the Strimzi Kafka operator from this site**
  - https://strimzi.io/downloads/
  - Click on the latest, 0.20
  - Download strimzi-0.20.0.zip or strimzi-0.20.0.tar.gz for example and documentation
  - Download strimzi-kafka-operator-helm-3-chart-0.20.0.tgz



0.20.0

scholzj released this 12 days ago · 67 commits to master since this release

Main Changes since 0.19.0

Note: This is the last version of Strimzi that will support Kubernetes 1.11 and higher. Future v
Kubernetes 1.16 and higher.

- strimzi-0.20.0.tar.gz
- strimzi-0.20.0.zip
- strimzi-cluster-operator-0.20.0.yaml
- strimzi-kafka-operator-helm-2-chart-0.20.0.tgz
- strimzi-kafka-operator-helm-3-chart-0.20.0.tgz
- strimzi-topic-operator-0.20.0.yaml
- strimzi-user-operator-0.20.0.yaml
- Source code (zip)
- Source code (tar.gz)

How to install Strimzi Kafka

- **Installation and running a Strimzi Kafka cluster is a two-step process.**
  - Install the Strimzi Helm Chart
  - Create a Kafka Kubernetes custom resource
- **To install Strimzi Helm Chart**

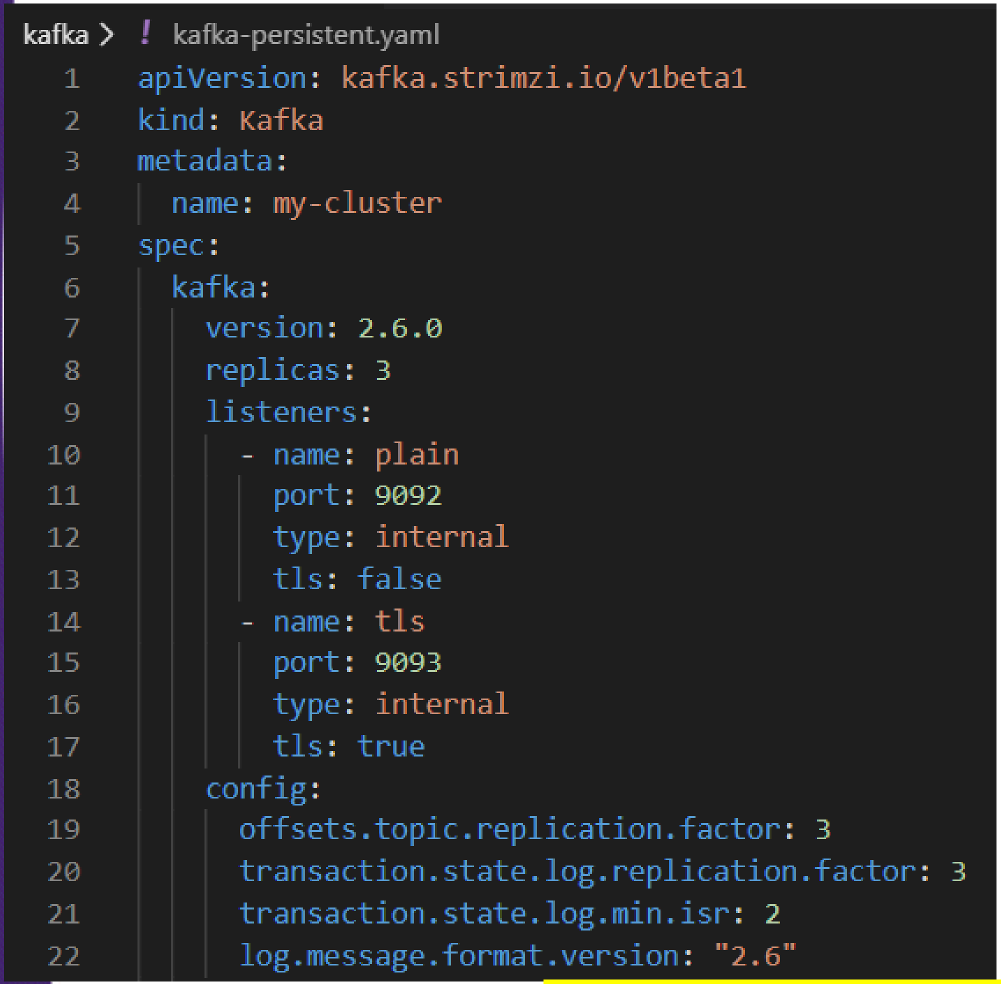
```
%helm repo add strimzi http://strimzi.io/charts/
%helm install strimzi/strimzi-kafka-operator
```

  - Or, run the following command

```
%helm install strimzi-kafka ./strimzi-kafka-operator-helm-3-
chart-0.20.0.tgz
```

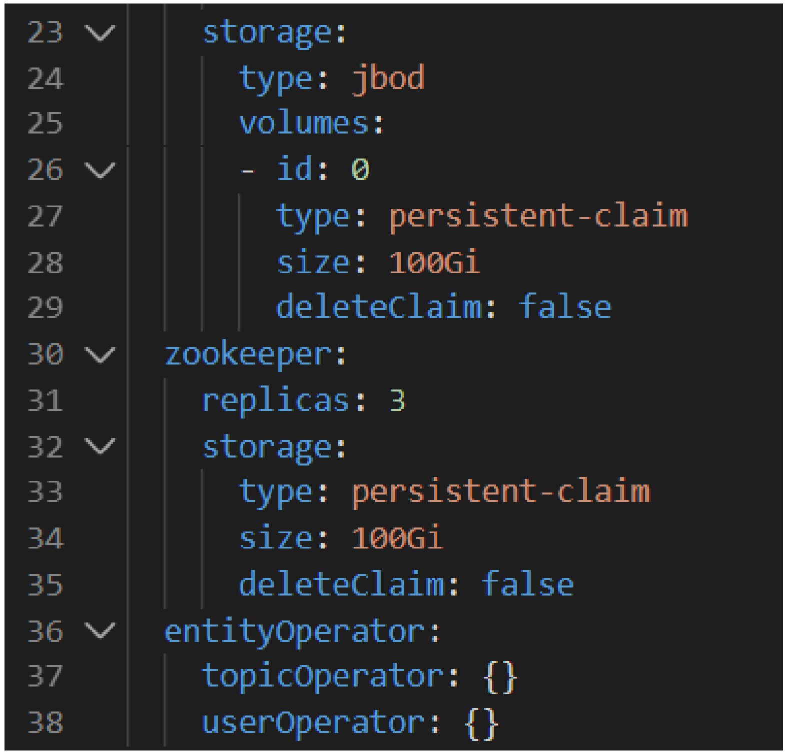
Example Kafka cluster configuration

- Example kafka-persistent.yaml



kafka > ! kafka-persistent.yaml

```yaml
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    version: 2.6.0
    replicas: 3
    listeners:
      - name: plain
        port: 9092
        type: internal
        tls: false
      - name: tls
        port: 9093
        type: internal
        tls: true
    config:
      offsets.topic.replication.factor: 3
      transaction.state.log.replication.factor: 3
      transaction.state.log.min.isr: 2
      log.message.format.version: "2.6"
```

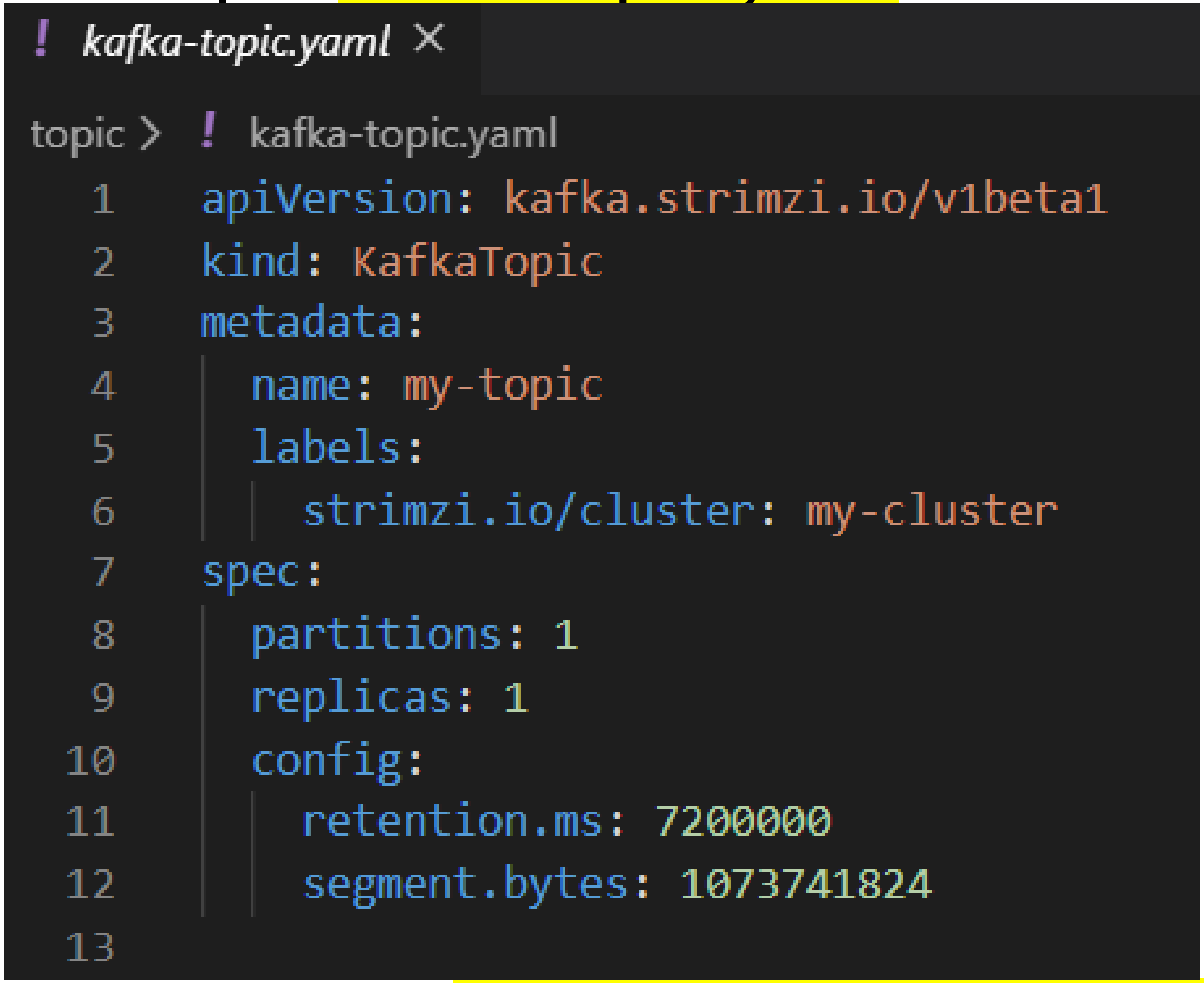


```yaml
    storage:
      type: jbod
      volumes:
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
  zookeeper:
    replicas: 3
    storage:
      type: persistent-claim
      size: 100Gi
      deleteClaim: false
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

- To deploy, %kubectl apply -f kafka-persistent.yaml

Example Kafka topic configuration

- Example kafka-topic.yaml



kafka-topic.yaml ×

topic > ! kafka-topic.yaml

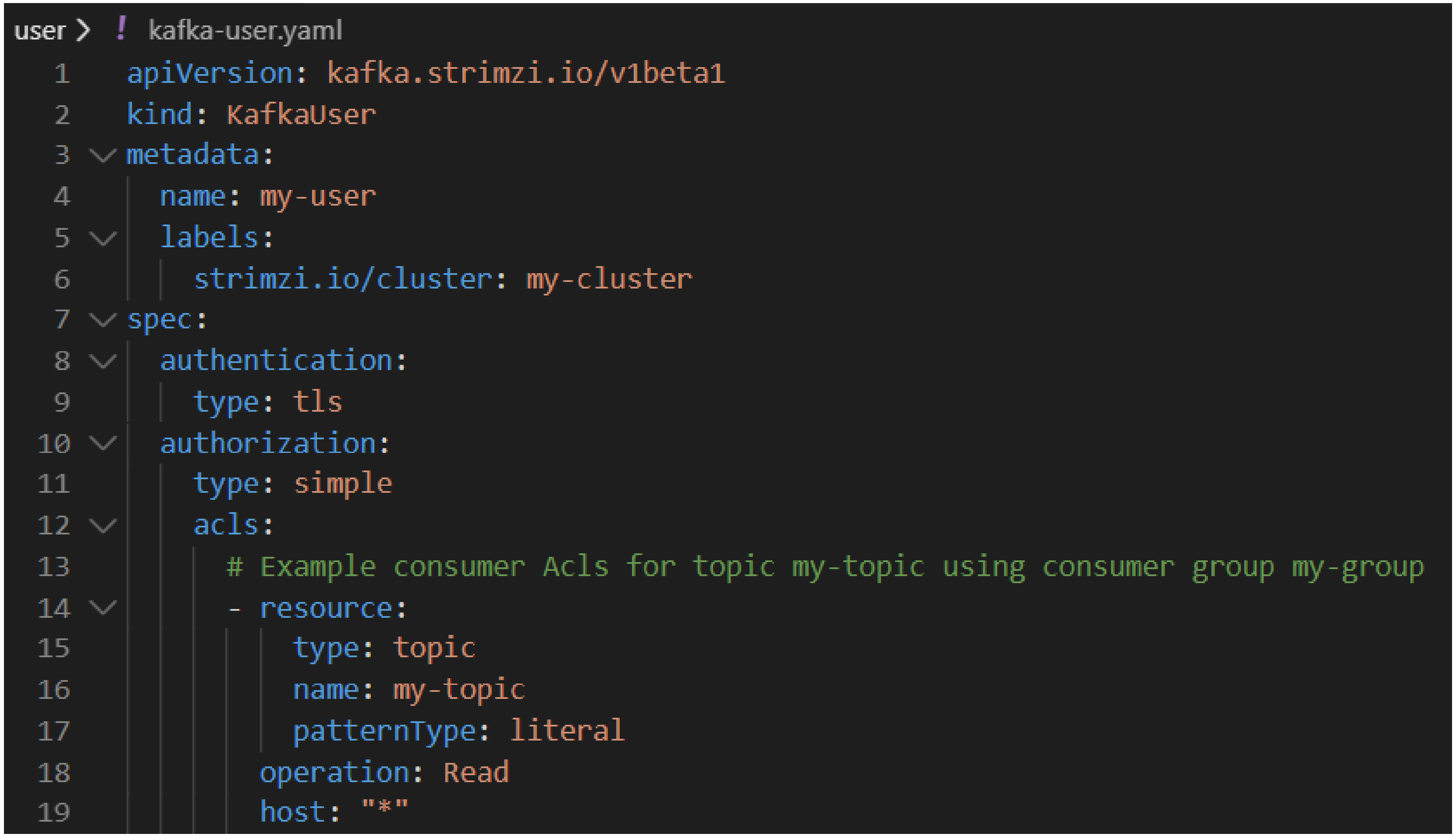
```yaml
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 1
  replicas: 1
  config:
    retention.ms: 7200000
    segment.bytes: 1073741824

```

- To deploy, %kubectl apply -f kafka-topic.yaml

Example Kafka user configuration

- Example kafka-user.yaml



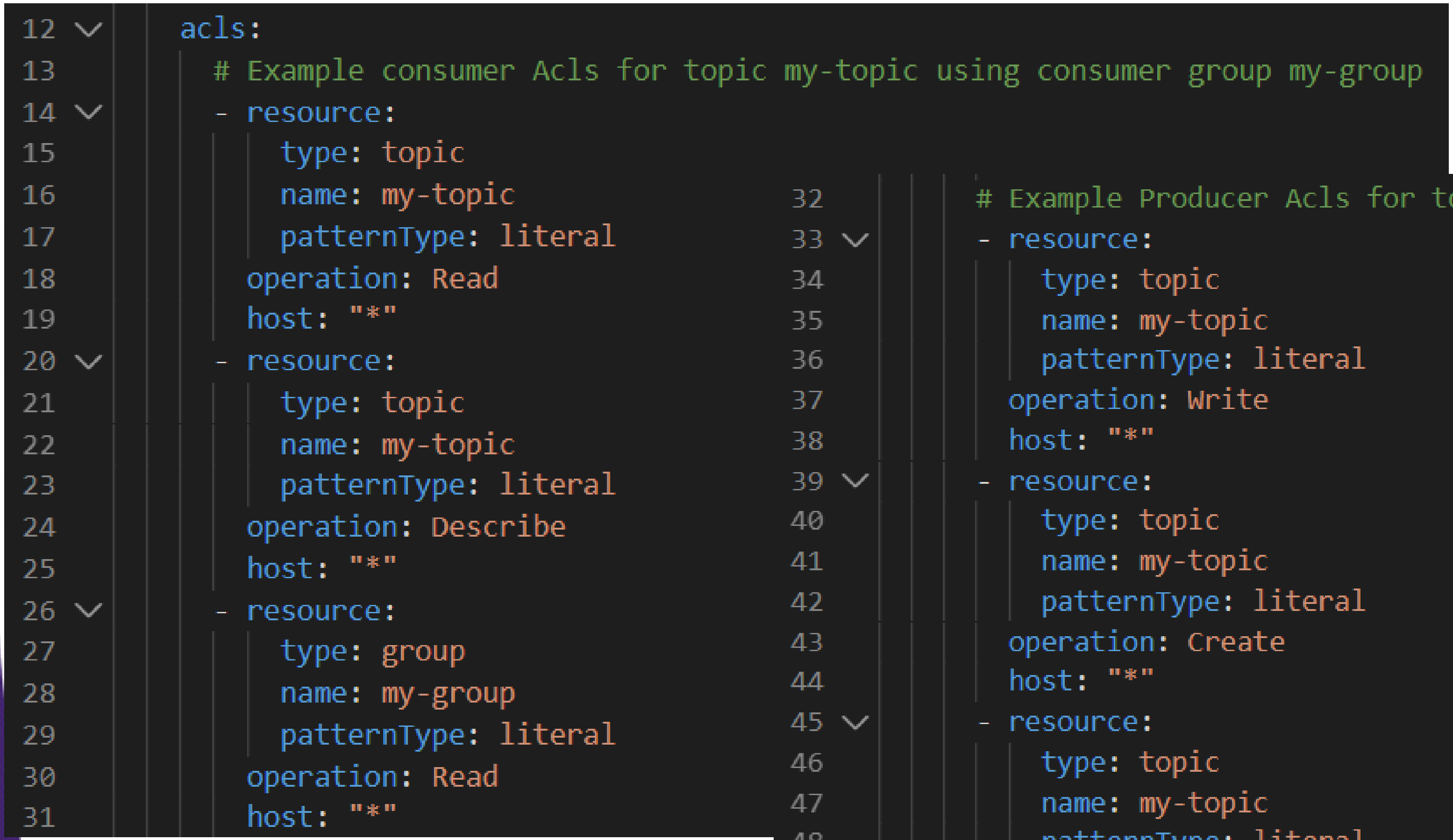
user > ! kafka-user.yaml

```yaml
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: tls
  authorization:
    type: simple
    acls:
      # Example consumer Acls for topic my-topic using consumer group my-group
      - resource:
          type: topic
          name: my-topic
          patternType: literal
        operation: Read
        host: "*"
```

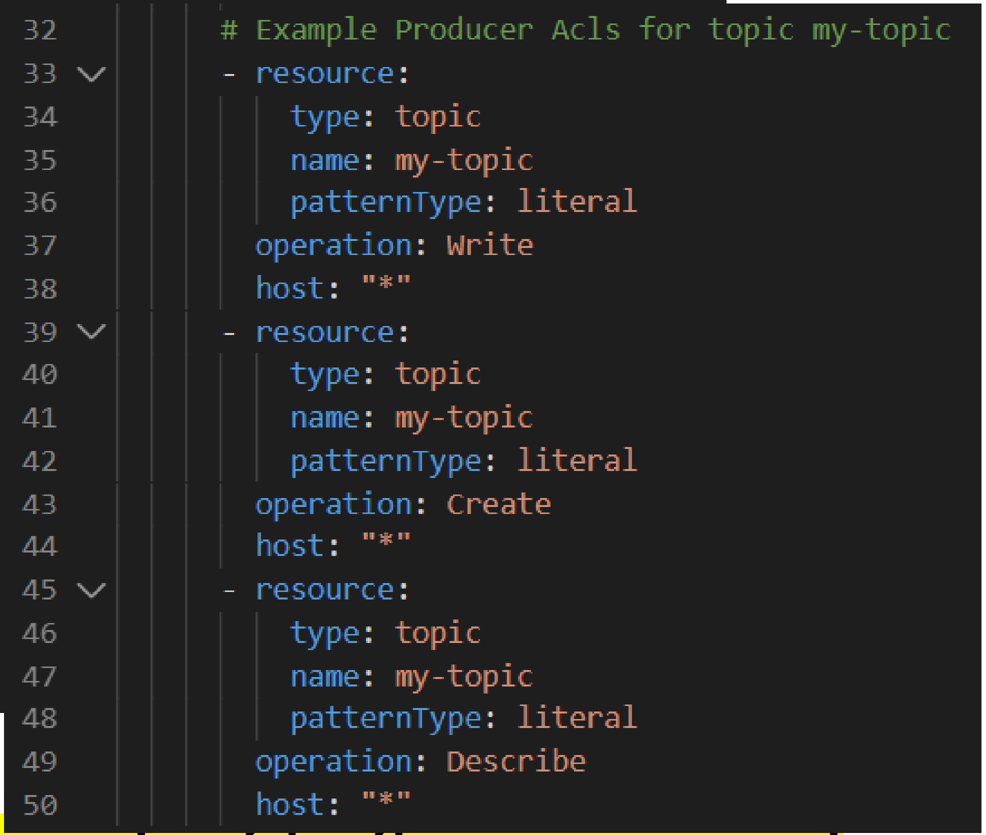
- To deploy, %kubectl apply -f kafka-user.yaml

Example Kafka user configuration

- Example kafka-user.yaml



```yaml
acls:
  # Example consumer Acls for topic my-topic using consumer group my-group
  - resource:
      type: topic
      name: my-topic
      patternType: literal
    operation: Read
    host: "*"
  - resource:
      type: topic
      name: my-topic
      patternType: literal
    operation: Describe
    host: "*"
  - resource:
      type: group
      name: my-group
      patternType: literal
    operation: Read
    host: "*"
```



```yaml
  # Example Producer Acls for topic my-topic
  - resource:
      type: topic
      name: my-topic
      patternType: literal
    operation: Write
    host: "*"
  - resource:
      type: topic
      name: my-topic
      patternType: literal
    operation: Create
    host: "*"
  - resource:
      type: topic
      name: my-topic
      patternType: literal
    operation: Describe
    host: "*"
```

- To deploy %kubectl apply -f kafka-user.yaml

URL to connect to Kafka broker

- Fully qualified service hostname

$$\underbrace{\texttt{simple-strimzi}}_{\text{Kafka resource metadata.name}}\texttt{-}\underbrace{\texttt{kafka-bootstrap}}_{\text{Broker load balancer name}}\texttt{.}\underbrace{\texttt{strimzi}}_{\text{Namespace}}\texttt{.}\underbrace{\texttt{svc}}_{\text{K8s Service}}\texttt{.cluster.local:}\underbrace{\texttt{9092}}$$

“Plain”	9092
TLS	9093
Interbroker	9094
Prometheus	9404

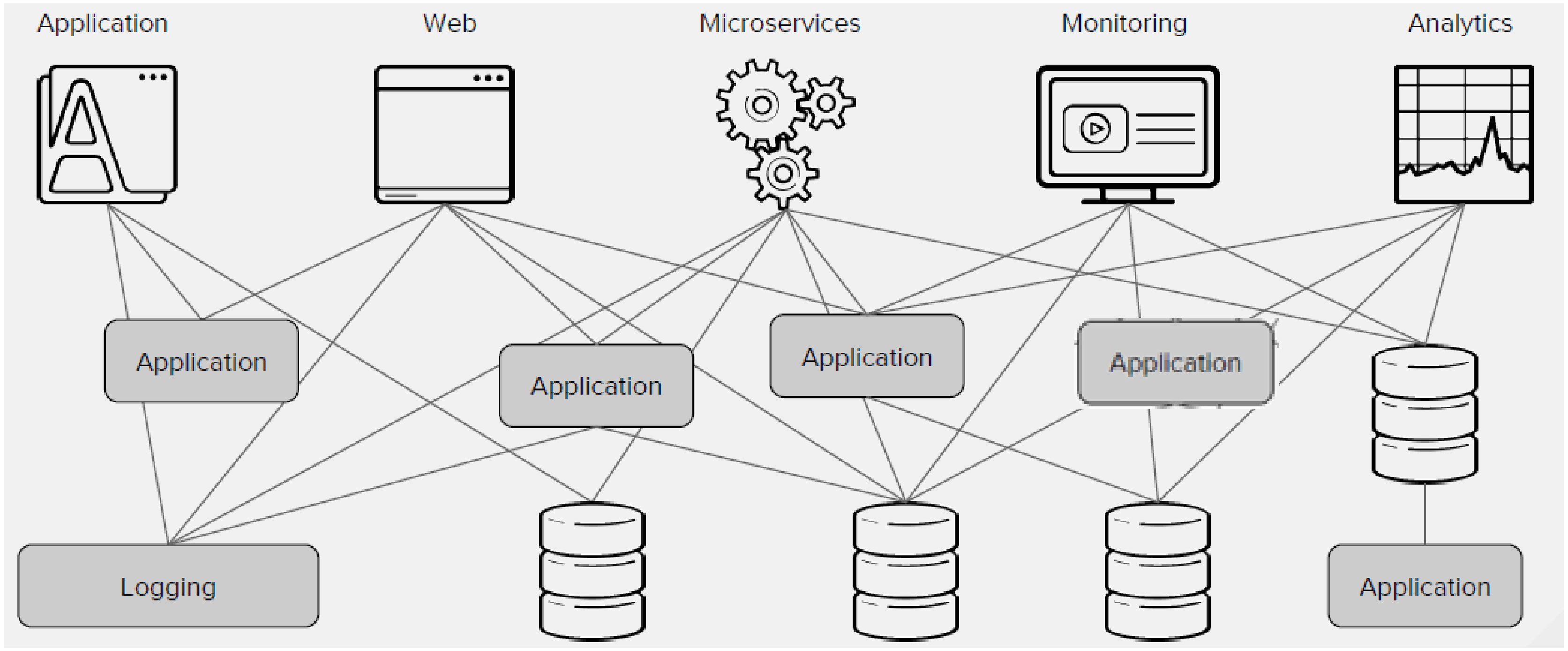
Optional Backups

Kafka Built-in Monitoring

- **Kafka code produces yammer metrics exposed via JMX (Java Management Extension),**
  - which allows to hook in external applications, such as Prometheus/Grafana to access metrics that Kafka is producing
- **Metrics produced by Kafka**
  - Messages/bytes in per second – coming from producers
  - Bytes out per second – read by consumers
  - Under replicated partitions
    - If a given partition is under replicated, that means replicas of the leader are not keeping up with the load on that partition
    - This could be due to too much load coming to the cluster and Kafka can't handle it
  - Network metrics – to measure how much traffic going through network

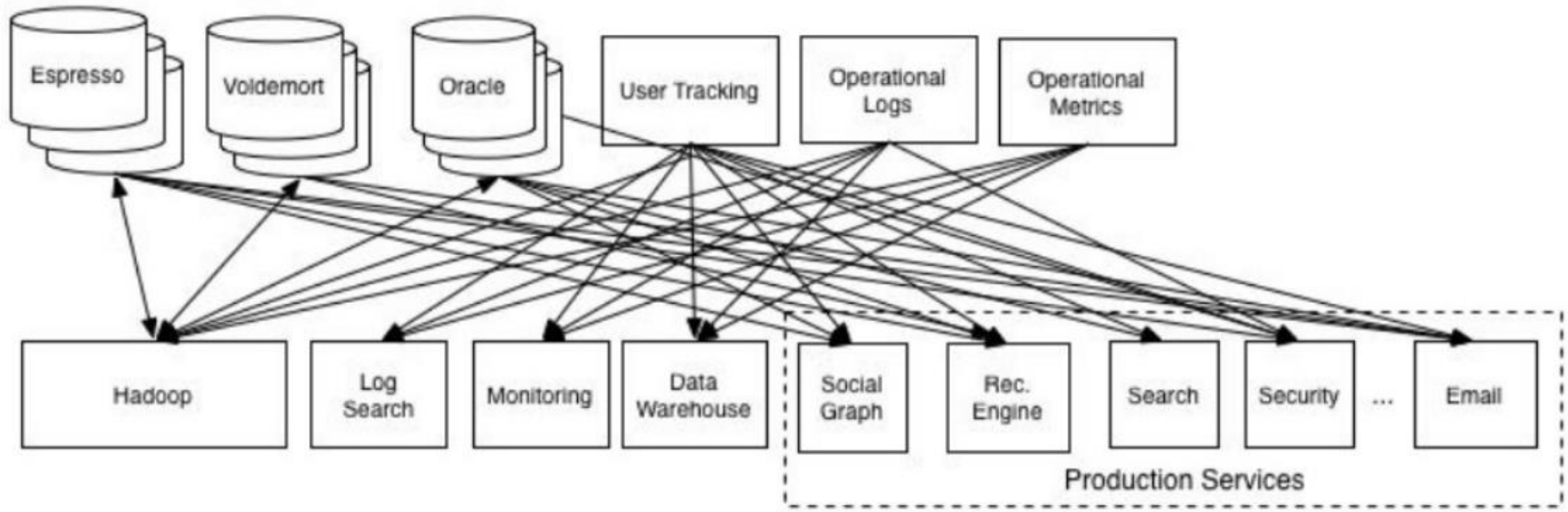
Potential for Spaghetti Architecture!

- **With the advent of microservices, there is a risk of creating too many point-to-point connections that will be difficult to maintain,**
  - Especially for business processes that may be composed of several micro services



Potential for Spaghetti Architecture!

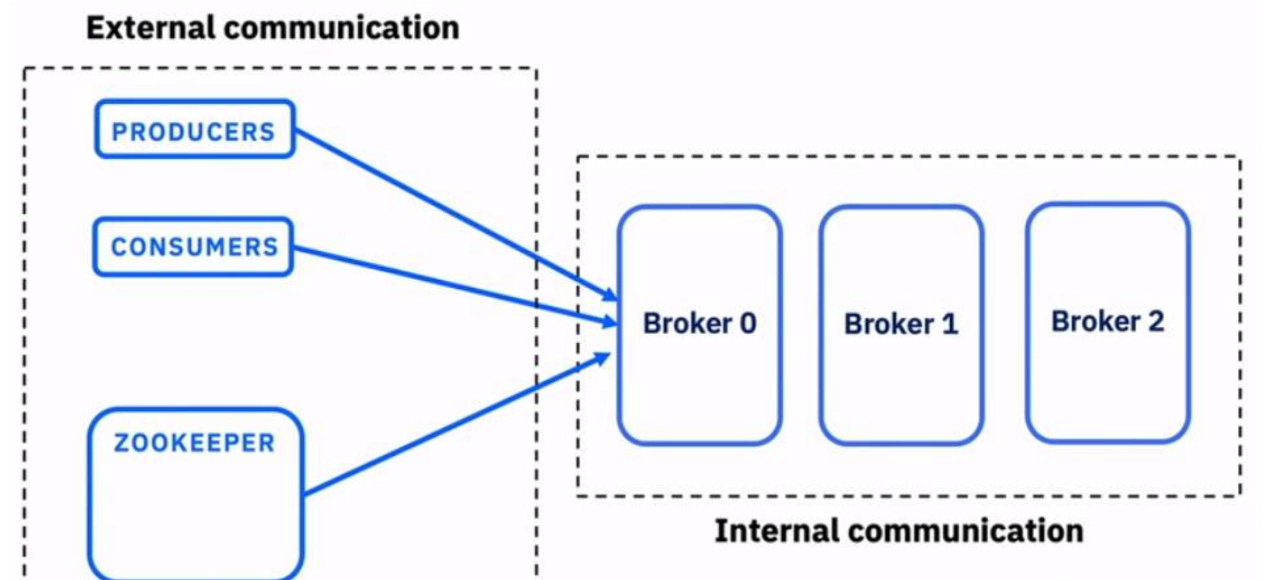
- **Another example**



Source: LinkedIn

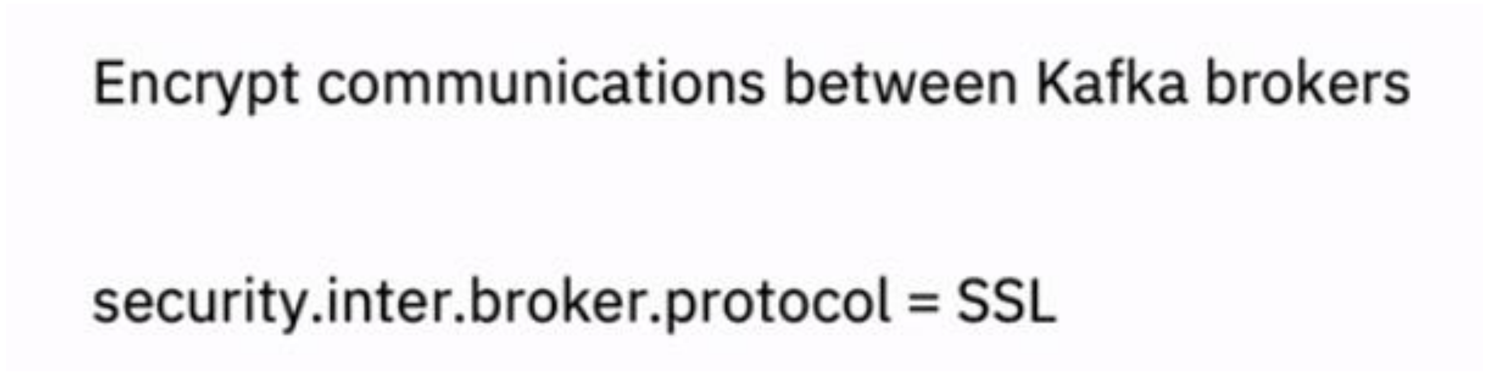
Kafka Built-in Security

- **Kafka supports two types of Security**
  - **Internal Communication** – for communication between Kafka brokers, e.g., for replication – this traffic, by default, not encrypted but can be configured to be encrypted
  - **External Communication** – for traffic going into Kafka from Producers, Consumers, and ZooKeepers



Kafka Built-in Security

- **Internal SSL**

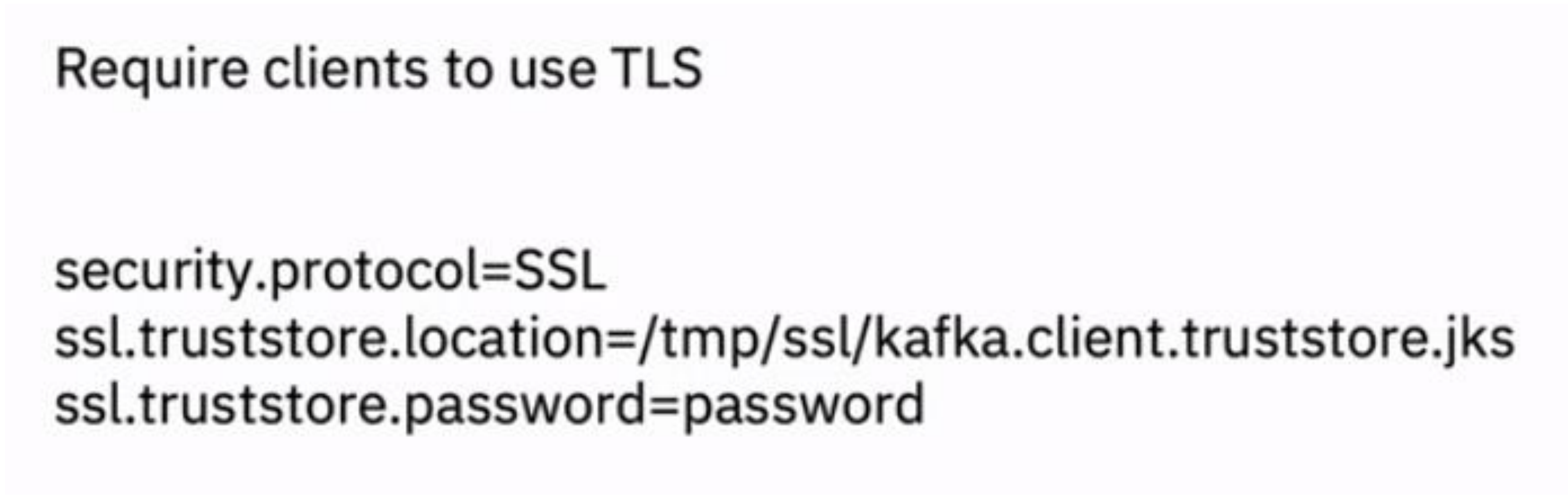


Encrypt communications between Kafka brokers

```
security.inter.broker.protocol = SSL
```

Kafka Built-in Security

- **External SSL**

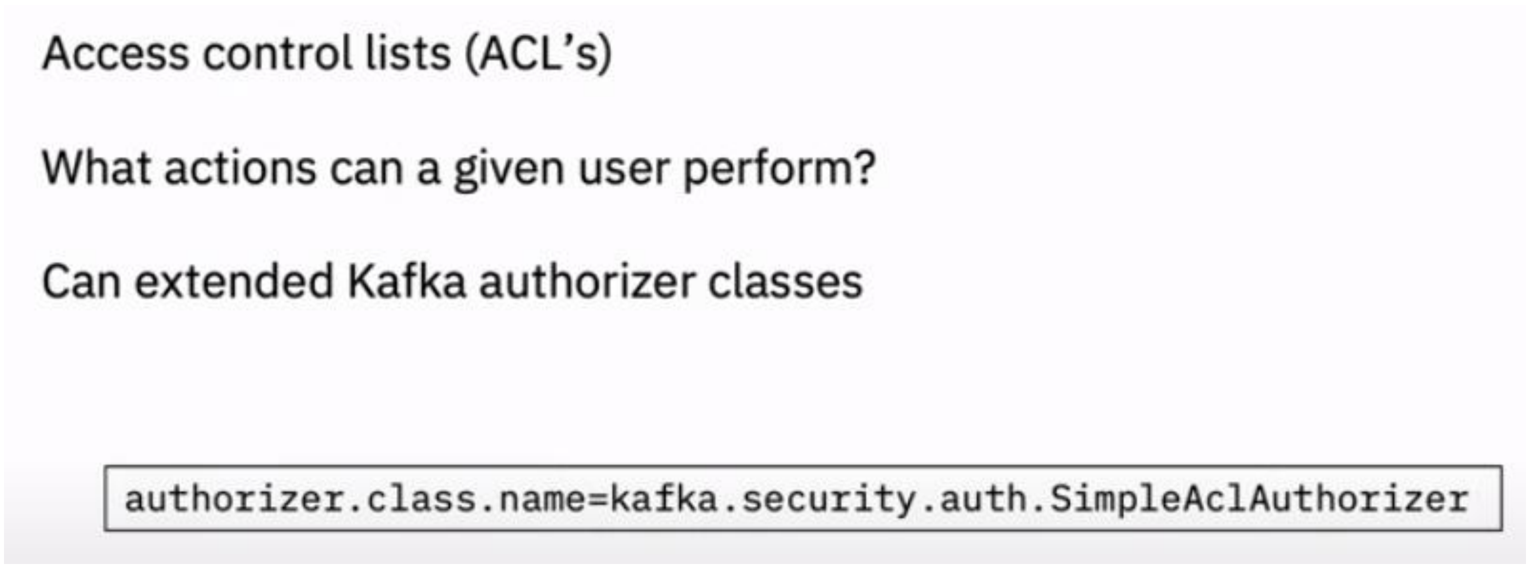


Require clients to use TLS

```
security.protocol=SSL
ssl.truststore.location=/tmp/ssl/kafka.client.truststore.jks
ssl.truststore.password=password
```

Kafka Built-in Security

- **Authorization**



Access control lists (ACL's)

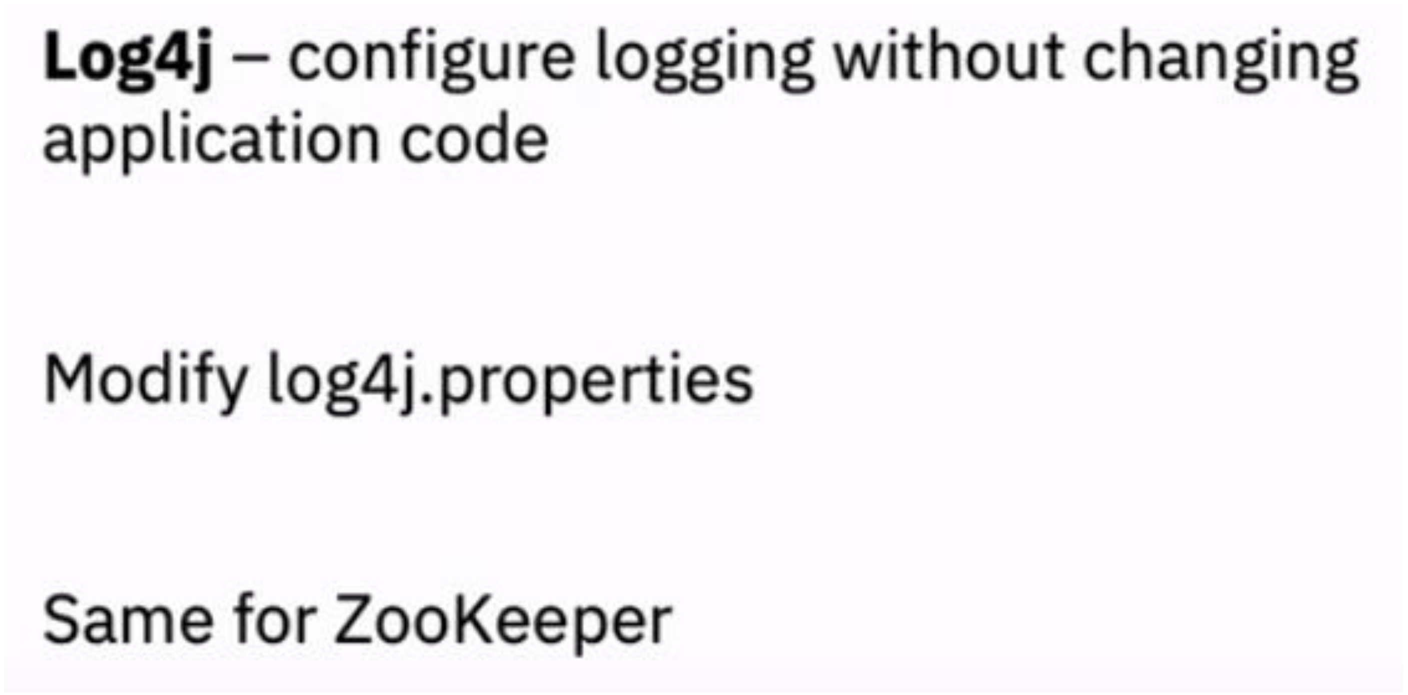
What actions can a given user perform?

Can extended Kafka authorizer classes

```
authorizer.class.name=kafka.security.auth.SimpleAclAuthorizer
```

Kafka Debugging

- **Logging**



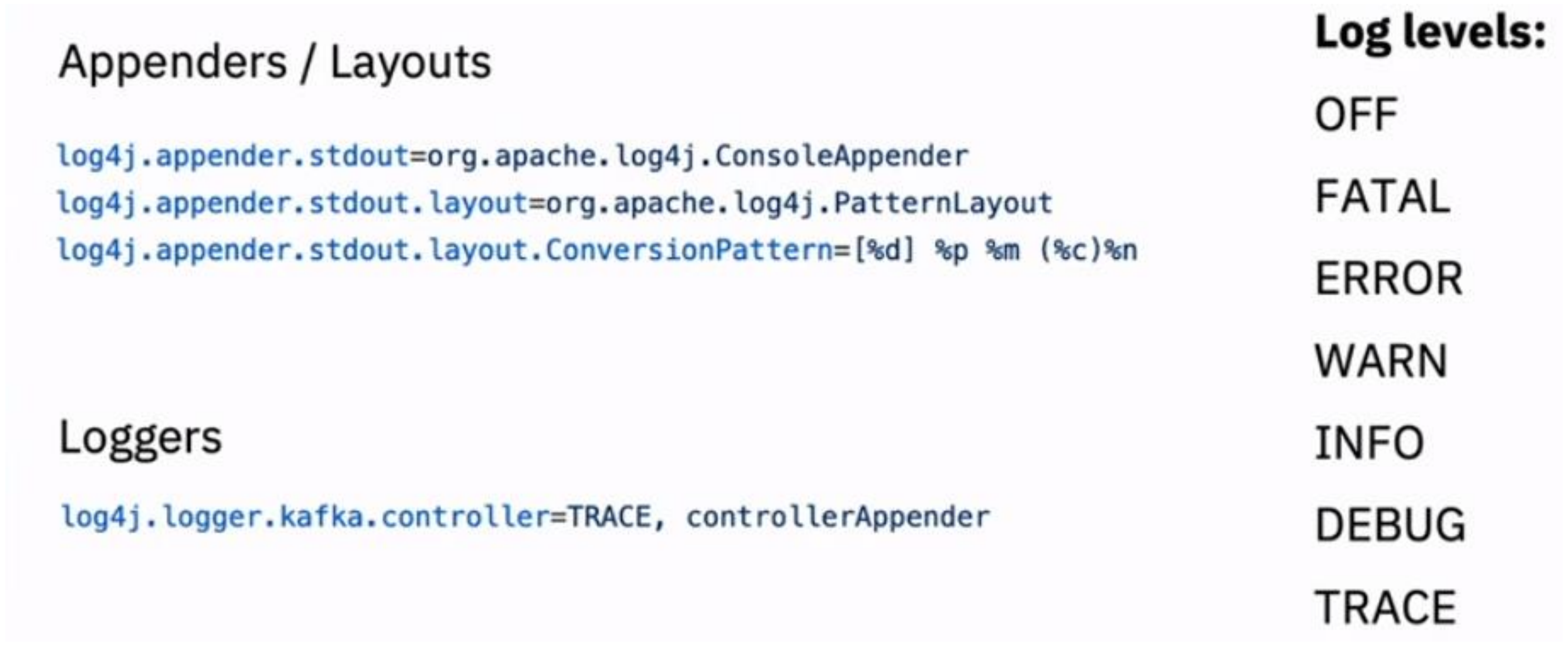
Log4j – configure logging without changing application code

Modify log4j.properties

Same for ZooKeeper

Kafka Debugging

- **Logging**



Appenders / Layouts

```
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=[%d] %p %m (%c)%n
```

Loggers

```
log4j.logger.kafka.controller=TRACE, controllerAppender
```

Log levels:

OFF

FATAL

ERROR

WARN

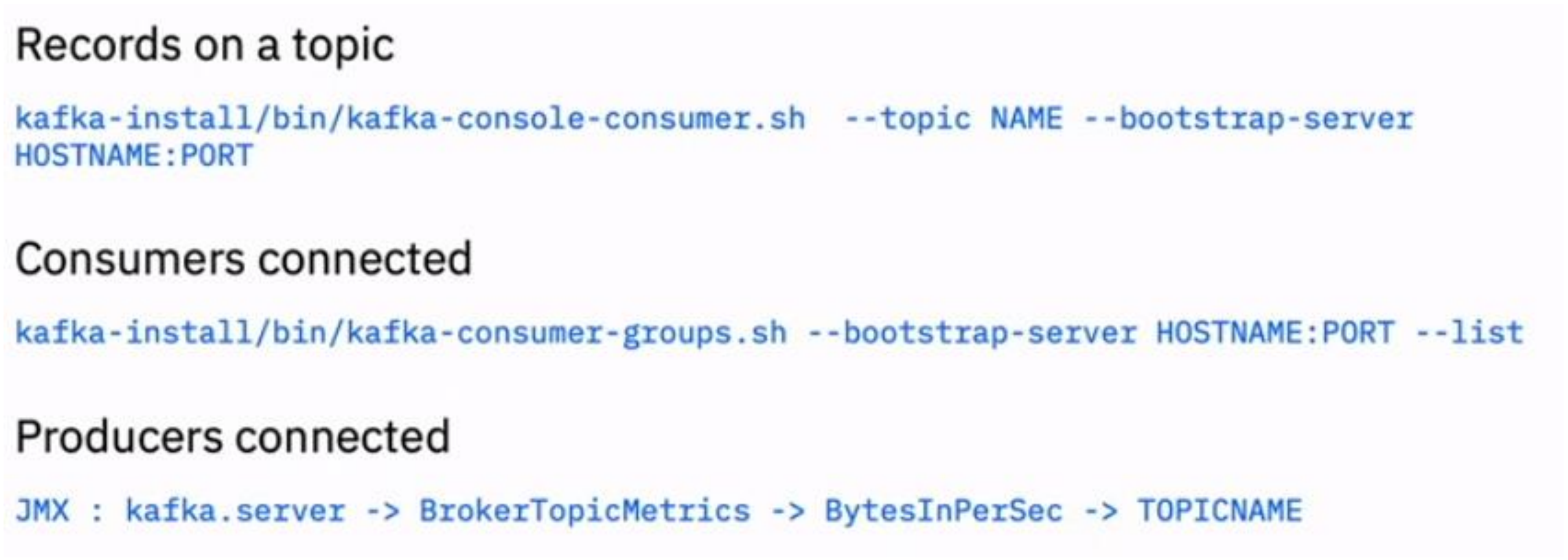
INFO

DEBUG

TRACE

Kafka Debugging

- **Debugging commands**



Records on a topic

```
kafka-install/bin/kafka-console-consumer.sh  --topic NAME --bootstrap-server
HOSTNAME:PORT
```

Consumers connected

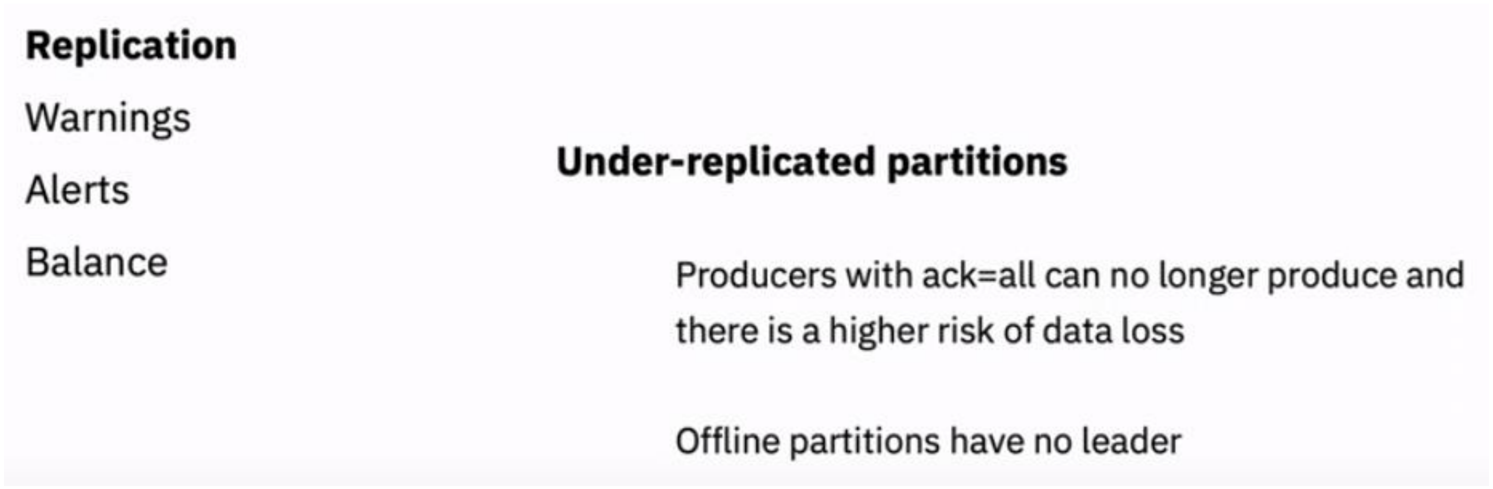
```
kafka-install/bin/kafka-consumer-groups.sh --bootstrap-server HOSTNAME:PORT --list
```

Producers connected

```
JMX : kafka.server -> BrokerTopicMetrics -> BytesInPerSec -> TOPICNAME
```

Kafka Debugging

- **What to look for in Production**



Replication

Warnings

Alerts

Balance

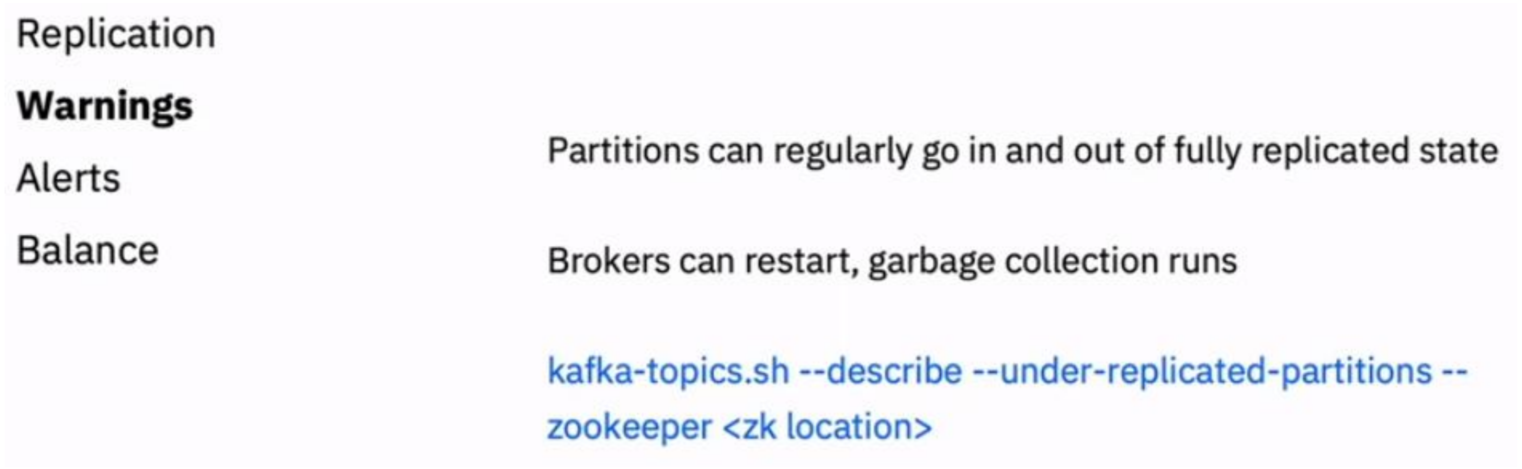
Under-replicated partitions

Producers with ack=all can no longer produce and there is a higher risk of data loss

Offline partitions have no leader

Kafka Debugging

- **What to look for in Production**


Replication

Warnings

Alerts

Balance

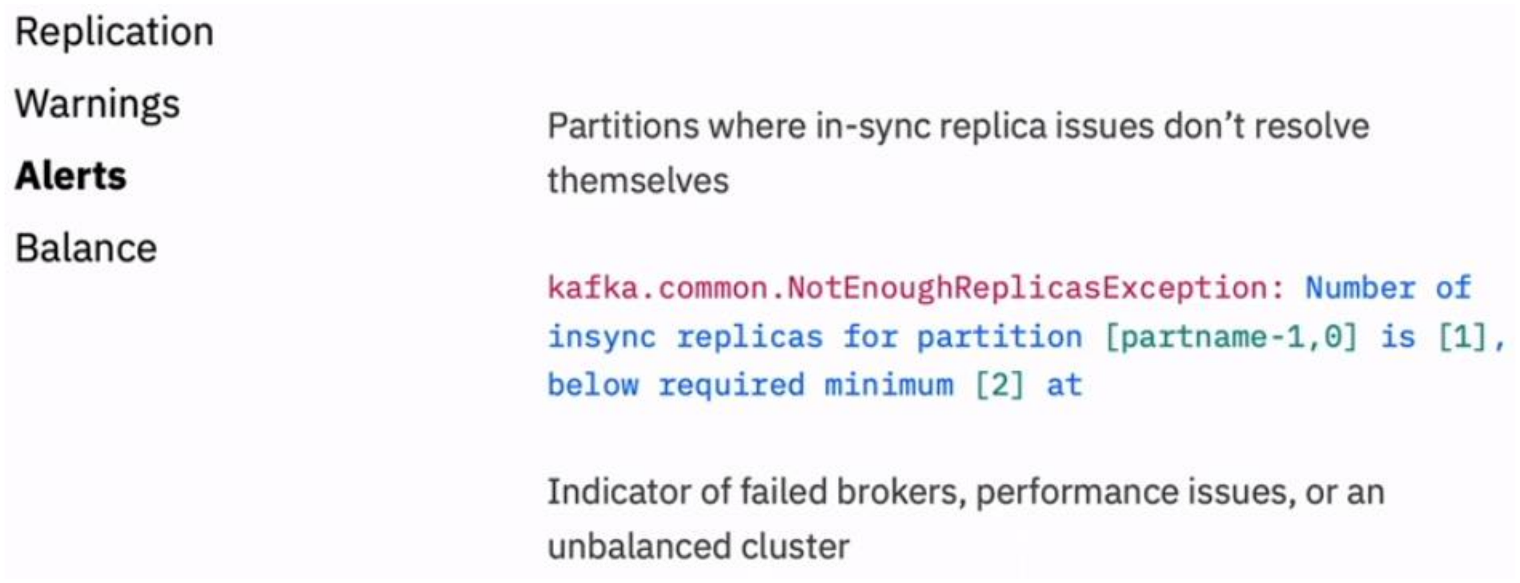
Partitions can regularly go in and out of fully replicated state

Brokers can restart, garbage collection runs

```
kafka-topics.sh --describe --under-replicated-partitions --
zookeeper <zk location>
```

Kafka Debugging

- **What to look for in Production**



Replication

Warnings

Alerts

Balance

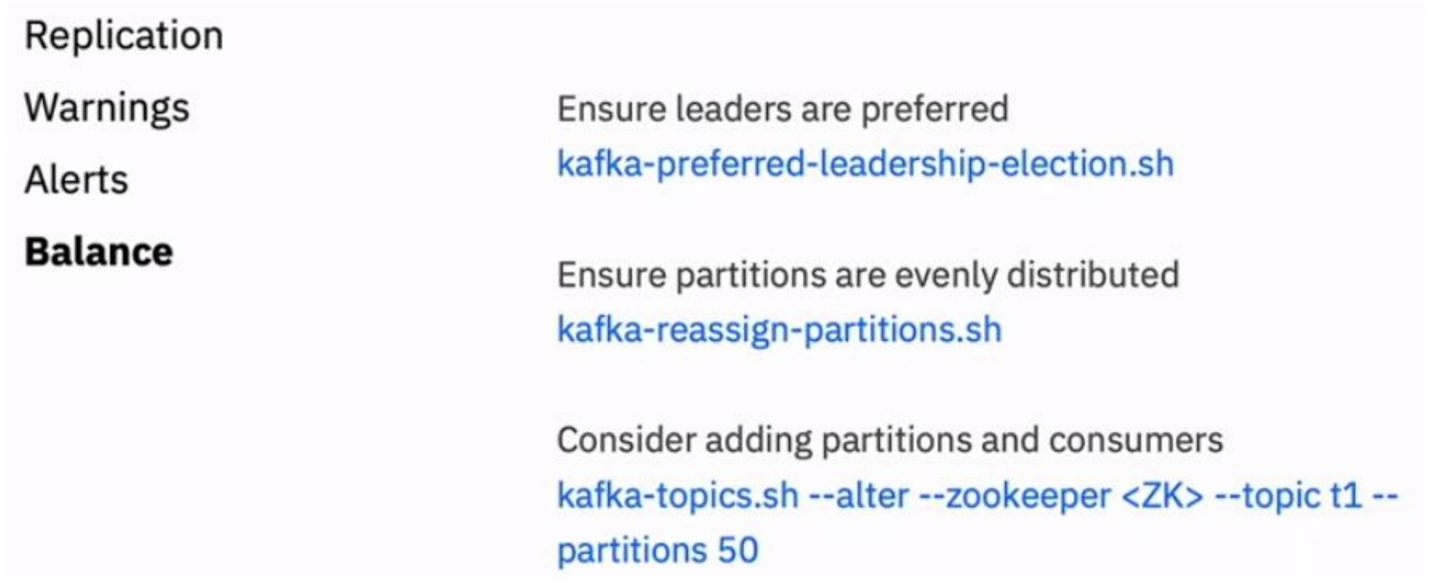
Partitions where in-sync replica issues don't resolve themselves

```
kafka.common.NotEnoughReplicasException: Number of
insync replicas for partition [partname-1,0] is [1],
below required minimum [2] at
```

Indicator of failed brokers, performance issues, or an unbalanced cluster

Kafka Debugging

- **What to look for in Production**



Replication

Warnings

Alerts

Balance

Ensure leaders are preferred

```
kafka-preferred-leadership-election.sh
```

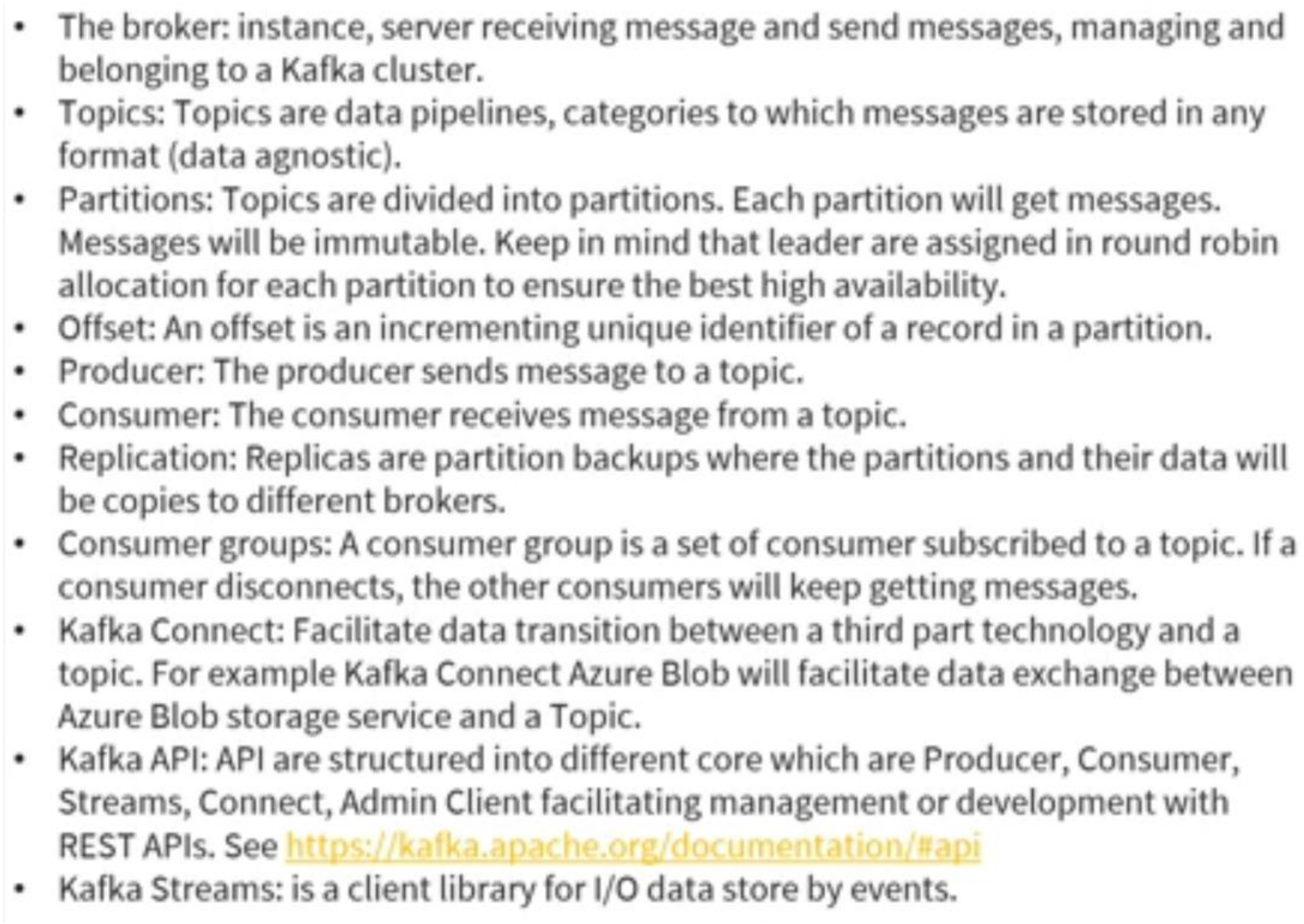
Ensure partitions are evenly distributed

```
kafka-reassign-partitions.sh
```

Consider adding partitions and consumers

```
kafka-topics.sh --alter --zookeeper <ZK> --topic t1 --partitions 50
```

Kafka in a nutshell



- The broker: instance, server receiving message and send messages, managing and belonging to a Kafka cluster.
- Topics: Topics are data pipelines, categories to which messages are stored in any format (data agnostic).
- Partitions: Topics are divided into partitions. Each partition will get messages. Messages will be immutable. Keep in mind that leader are assigned in round robin allocation for each partition to ensure the best high availability.
- Offset: An offset is an incrementing unique identifier of a record in a partition.
- Producer: The producer sends message to a topic.
- Consumer: The consumer receives message from a topic.
- Replication: Replicas are partition backups where the partitions and their data will be copies to different brokers.
- Consumer groups: A consumer group is a set of consumer subscribed to a topic. If a consumer disconnects, the other consumers will keep getting messages.
- Kafka Connect: Facilitate data transition between a third part technology and a topic. For example Kafka Connect Azure Blob will facilitate data exchange between Azure Blob storage service and a Topic.
- Kafka API: API are structured into different core which are Producer, Consumer, Streams, Connect, Admin Client facilitating management or development with REST APIs. See https://kafka.apache.org/documentation/#api
- Kafka Streams: is a client library for I/O data store by events.

Schemas

- While messages are opaque byte arrays to Kafka itself, it is recommended that additional structure, or schema, be imposed on the message content so that it can be easily understood.
- There are many options available for message schema, such as JavaScript Object Notation (JSON) and Extensible Markup Language (XML).
- Many Kafka developers favor the use of Apache Avro, which is a serialization framework originally developed for Hadoop.
- Avro provides a compact serialization format, schemas that are separate from the message pay-loads and that do not require code to be generated when they change, and strong data typing and schema evolution, with both backward and forward compatibility.
- A consistent data format is important in Kafka, as it allows writing and reading messages to be decoupled.
- By using well-defined schemas and storing them in a common repository, the messages in Kafka can be understood without coordination.

Why Kafka

- **There are many choices for publish/subscribe messaging systems, here are some of the factors that make Apache Kafka a better choice:**
- **Multiple Producers**
  - Kafka is able to seamlessly handle multiple producers, whether those clients are using many topics or the same topic.
  - This makes the system ideal for aggregating data from many frontend systems and making it consistent.
- **Multiple Consumers**
  - In addition to multiple producers, Kafka is designed for multiple consumers to read any single stream of messages without interfering with each other client.
  - This is in contrast to many queuing systems where once a message is consumed by one client, it is not available to any other.
  - Multiple Kafka consumers can choose to operate as part of a group and share a stream, assuring that the entire group processes a given message only once.

Why Kafka

- **Disk-Based Retention**
  - Not only can Kafka handle multiple consumers, but durable message retention means that consumers do not always need to work in real time.
  - Messages are written to disk and will be stored with configurable retention rules.
  - These options can be selected on a per-topic basis, allowing for different streams of messages to have different amounts of retention depending on the consumer needs.
  - Durable retention means that if a consumer falls behind, either due to slow processing or a burst in traffic, there is no danger of losing data.
  - It also means that maintenance can be performed on consumers, taking applications offline for a short period of time, with no concern about messages backing up on the producer or getting lost.
  - Consumers can be stopped, and the messages will be retained in Kafka.
  - This allows them to restart and pick up processing messages where they left off with no data loss.

Why Kafka

- **Scalable**
  - Kafka’s flexible scalability makes it easy to handle any amount of data.
  - Users can start with a single broker as a proof of concept, expand to a small development cluster of three brokers, and move into production with a larger cluster of tens or even hundreds of brokers that grows over time as the data scales up.
  - Expansions can be performed while the cluster is online, with no impact on the availability of the system as a whole.
  - This also means that a cluster of multiple brokers can handle the failure of an individual broker and continue servicing clients.
  - Clusters that need to tolerate more simultaneous failures can be configured with higher replication factors.

Why Kafka

- **High Performance**
  - Apache Kafka is a publish/subscribe messaging system with excellent performance under high load.
  - Producers, consumers, and brokers can all be scaled out to handle very large message streams with ease.
  - This can be done while still providing sub second message latency from producing a message to availability to consumers.
- **Platform Feature**
  - The core Apache Kafka project has also added some streaming platform features that can make it much easier for developers to perform common types of work.
  - Kafka Connect assists with the task of pulling data from a source data system and pushing it into Kafka, or pulling data from Kafka and pushing it into a sink data system.
  - Kafka Streams provides a library for easily developing stream processing applications that are scalable and fault tolerant.

Common Use Cases

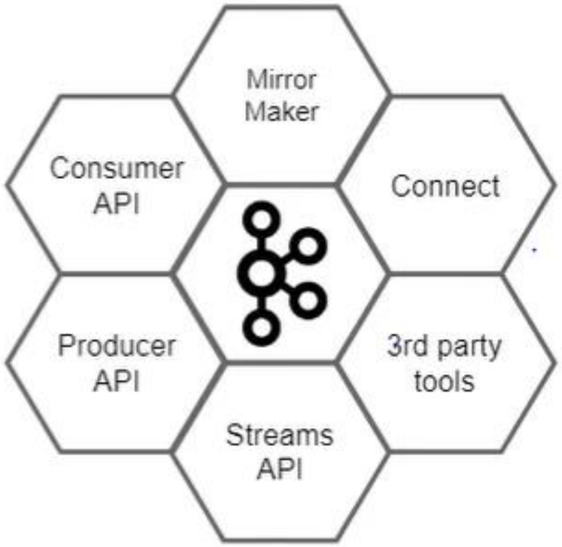
- **Commonly used for**
  - high-performance data pipelines
  - streaming analytics
  - data integration and
  - mission-critical applications
- **Also, allows continuous import/export of your data from other systems.**

Sample sources of events

- **Candidate sources of events for Kafka**
  - Twitter feeds
  - IOT Devices
  - Weblogs
  - Banking transactions
  - Database updates

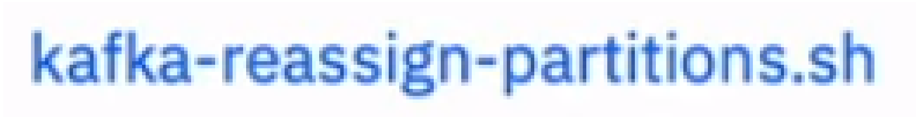
Kafka Ecosystem

- Apache Kafka has an ecosystem consisting of many components
  - Kafka Core
    - Broker(s)
    - Client library (producer, consumer)
  - Kafka Connect
  - Kafka Streams
  - Mirror Maker



Partition Rebalancing

- **Sometimes partitions become unbalanced after the assignment of partitions between brokers**
  - Imagine have 3 brokers and with one partition on each
  - If one broker goes down, and you create a new topic with three partitions, this will result in two partitions on one broker and one partition on another broker
  - When the down broker comes up, that broker will not have any partition for the new topic
  - So, rebalancing the partition using the built-in script from Kafka will evenly spread the partitions across the brokers



```
kafka-reassign-partitions.sh
```

External Kafka Clients

- **External clients are those that get data into and out of Kafka**
  - Producers
  - Consumers
  - Kafka Connect
  - Kafka Streams