***Apache Kafka***

Kafka is a distributed streaming platform that is used publish and subscribe to streams of records.

Kafka is a highly scalable and fault tolerant enterprise messaging system.

**A streaming platform has three key capabilities:**

* Publish and subscribe to streams of records, similar to a message queue or enterprise messaging system.
* Store streams of records in a fault-tolerant durable way.
* Process streams of records as they occur.

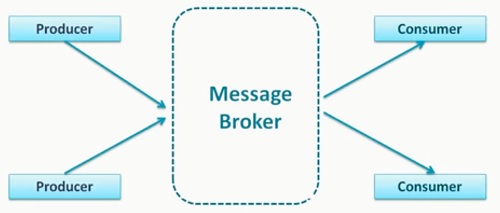
**First a few concepts:**

* Kafka is run as a cluster on one or more servers that can span multiple datacenters.
* The Kafka cluster stores streams of *records* in categories called *topics*.
* Each record consists of a key, a value, and a timestamp.

## **Messaging System:**

There are three components.

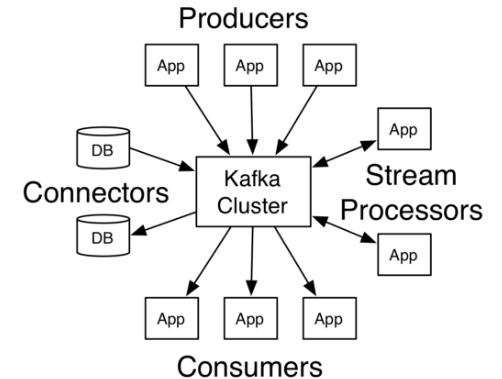
1. Producer or Publisher
2. Broker
3. Consumer.



The producers are the client applications, and they send some messages.   
The Brokers receive those messages from publishers and store them.   
The consumers read the message records from brokers.

**Kafka has four core APIs:**

* The [Producer API](http://kafka.apache.org/documentation.html#producerapi) allows an application to publish a stream of records to one or more Kafka topics.
* The [Consumer API](http://kafka.apache.org/documentation.html#consumerapi) allows an application to subscribe to one or more topics and process the stream of records produced to them.
* The [Streams API](http://kafka.apache.org/documentation/streams) allows an application to act as a *stream processor*, consuming an input stream from one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams.
* The [Connector API](http://kafka.apache.org/documentation.html#connect) allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems. For example, a connector to a relational database might capture every change to a table.



In Kafka, the communication between the clients and the servers is done with a simple, high-performance, language agnostic [TCP protocol](https://kafka.apache.org/protocol.html).

**The Producer applications** are sending messages to Kafka cluster. The Kafka cluster is nothing but a bunch of brokers running in a group of computers. They take message records from producers and store it in Kafka message log.

**There are consumer applications**. they read messages from Kafka cluster, processes it and do whatever they want to do. They may want to send them to Hadoop, Cassandra, HBase or may be pushing it back again into Kafka for someone else to read these modified or transformed records

**Kafka Streams:**

What is a stream:  
Stream is a continuous flow of data or we can define it as a constant stream of messages.   
Kafka, as a messaging system is so powerful regarding throughput and scalability that it allows you to handle a continuous stream of messages. If you can just plug in some stream processing framework to Kafka, Those are some stream processing applications. They read a continuous stream of data from Kafka, process them and then either store them back in Kafka or send them directly to other systems. Kafka provides some stream processing APIs as well. So you can do a lot of things using Kafka stream processing APIs, or you can use other stream processing frameworks like Spark streaming or Storm.

### Kafka Connect

The next thing is Kafka connector. These are the most compelling features. They are ready to use connectors to import data from databases into Kafka or export data from Kafka to databases. These are not just out of the box connectors but also a framework to build specialized connectors for any other application.

This Kafka Connector API allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems. For example, a connector to a relational database might capture every change to a table

## **Kafka Broker**

Now, let’s move on and try to understand a Broker. The broker is Kafka server. It is just a meaningful name given to Kafka server. And this title makes sense as well because all that Kafka does is act as a message broker between producer and consumer. The producer and consumer don not interact directly. They use Kafka server as an agent or a broker to exchange messages.

## **Kafka Cluster**

Let's come to the next term. The cluster. This one is simple. If you have any background in distributed systems, you already know that a Cluster is a group of computers acting together for a common purpose. Since Kafka is a distributed system, so the cluster has the same meaning for Kafka. It is merely a group of computers, each executing one instance of Kafka broker.

#### [**Topics and Logs**](http://kafka.apache.org/intro.html#intro_topics)**:**

A topic is a category or feed name to which records are published. Topics in Kafka are always multi-subscriber; that is, a topic can have zero, one, or many consumers that subscribe to the data written to it.

For each topic, the Kafka cluster maintains a partitioned log that looks like this:



Each partition is an ordered, immutable sequence of records that is continually appended to—a structured commit log. The records in the partitions are each assigned a sequential id number called the offset that uniquely identifies each record within the partition.

Kafka is a distributed system that runs on a cluster of machines. So, it is self-evident that Kafka can break a topic into partitions and store one partition on one computer. And that's what the Partition means.

Kafka knows that it should create 100 partitions or just ten partitions could be enough?   
The answer is simple. Kafka doesn't take that decision. We, as a developer make that decision. When we create a topic, we make that decision, and Kafka broker will create that many partitions for our Topic. But remember that every Partition sits on a single machine. You can't break it again. So, do some estimation and simple math to calculate the number of partitions.

## **Offsets:**

Let's talk about offset. The offset is simple. It is a sequence number of a message in a partition. This number is assigned as the messages arrive in a partition. And these numbers, once assigned, they never change. They are immutable. This sequencing means that Kafka stores messages in the order of arrival within a partition. The first message gets an offset zero. The next message receives an offset one and so on. But remember that there is no global offset across partitions. Offsets are local to the partition. So, if you want to locate a message, you should know three things.   
Topic name, Partition number, and an offset number. If you have these three things, you can directly locate a message.

## **Kafka Consumer Groups**

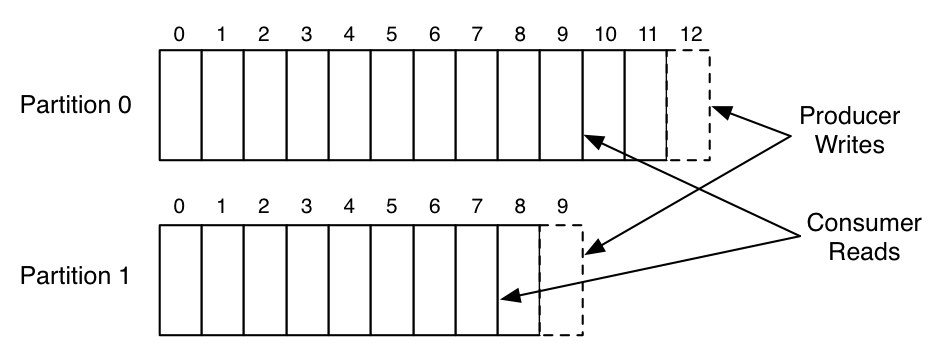
Now we are left with the last thing. The consumer groups. We already understand the Consumer. What is Consumer Group?   
It is a group of consumers. Several Consumers form a group to share the work. You can think of it like there is one large task and you want to divide it among multiple people, so you create a group, and members of the same group share the work

**The Advantages of using Apache Kafka are as follows-**

* **High Throughput:**  
  The design of Kafka enables the platform to process messages at very fast speed. The processing rates in Kafka can exceed beyond 100k/seconds. The data is processed in a partitioned and ordered fashion.
* **Scalability:**  
  The scalability can be achieved in Kafka at various levels. Multiple producers can write to the same topic. Topics can be partitioned. Consumers can be grouped to consume individual partitions.
* **Fault Tolerance:**  
  Kafka is a distributed architecture which means there are several nodes running together to serve the cluster. Topics inside Kafka are replicated. Users can choose the number of replicas for each topic to be safe in case of a node failure. Node failure in cluster wonâ€™t impact. Integration with Zookeeper provides producers and consumers accurate information about the cluster. Internally each topic has its own leader which takes care of the writes. Failure of node ensures new leader election.
* **Durability:**  
  Kafka offers data durability as well. The message written in Kafka can be persisted. The persistence can be configured. This ensures re-processing, if required, can be performed.

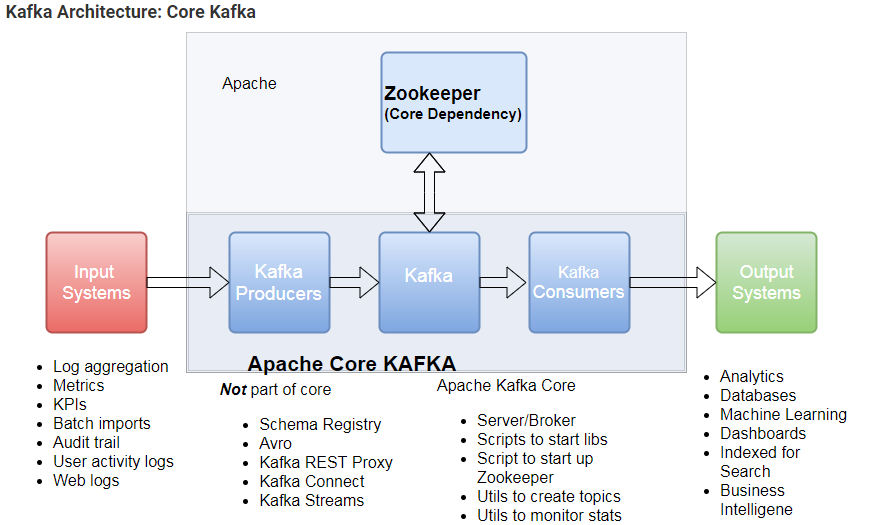
**The key features of Kafka are:**

* Kafka allows producers to publish messages while allowing consumers to read them. Messages are arrays of bytes, meaning you can send arbitrary data into Kafka.
* Messages are organized by topics. Messages sent into a topic can only be consumed by consumers subscribing to that topic.
* Messages are persisted on disk. The duration of retention is configured per topic and is 7 days by default.
* Messages within a topic are divided into partitions and are distributed. The idea is that a partition should fit on a single server. This allows to scale horizontally, i.e. by adding servers as you need more throughput.
* Not only messages are distributed in the cluster, they are replicated so that you never lose messages.
* Messages are assigned an offset. Offsets are unique and linear within a partition. It is the consumer’s responsibility to know which message it has read and which are to be consumed. In other terms, Kafka does not know about the status of each consumer. Instead, consumers have to maintain an offset per partition. That may sound strange but this is very powerful, e.g. it’s easy to process messages again by simply asking for messages from a given offset.
* The picture below shows a topic with 2 partitions. The producer writes messages to both partitions (with some partitioning logic or by round-robin) and they are simply added to the end of each partition while being assigned an offset by Kafka. At the same time, a consumer reads messages from each partition.



### Kafka Architecture

[Kafka](http://cloudurable.com/kafka-training/index.html) consists of Records, Topics, Consumers, Producers, Brokers, Logs, Partitions, and Clusters. Records can have key (optional), value and timestamp. Kafka Records are immutable. A Kafka Topic is a stream of records ("/orders", "/user-signups"). You can think of a Topic as a feed name. A topic has a Log which is the topic’s storage on disk. A Topic Log is broken up into partitions and segments. The Kafka Producer API is used to produce streams of data records. The Kafka Consumer API is used to consume a stream of records from Kafka. A Broker is a Kafka server that runs in a Kafka Cluster. Kafka Brokers form a cluster. The Kafka Cluster consists of many Kafka Brokers on many servers. Broker sometimes refer to more of a logical system or as Kafka as a whole.



### Kafka Topic Architecture in Review

#### **What is an ISR?**

An ISR is an in-sync replica. If a leader fails, an ISR is picked to be a new leader.

#### **How does Kafka scale consumers?**

Kafka scales consumers by partition such that each consumer gets its share of partitions. A consumer can have more than one partition, but a partition can only be used by one consumer in a consumer group at a time. If you only have one partition, then you can only have one consumer.

#### **What are leaders? Followers?**

Leaders perform all reads and writes to a particular topic partition. Followers replicate leaders.

#### **How does Kafka perform failover for consumers?**

If a consumer in a consumer group dies, the partitions assigned to that consumer is divided up amongst the remaining consumers in that group.

#### **How does Kafka perform failover for Brokers?**

If a broker dies, then Kafka divides up leadership of its topic partitions to the remaining brokers in the cluster.

**Zookeeper:**

Zookeeper is mainly used to track status of nodes present in Kafka cluster and also to keep track of Kafka topics

Sample Kafka message communication with producer and consumer

1. Extract the binaries into a software/kafka folder. For the current version it's software/kafka\_2.11-0.9.0.0.
2. Change your current directory to point to the new folder.
3. Start the Zookeeper server by executing the command: bin/zookeeper-server-start.sh config/zookeeper.properties.
4. Start the Kafka server by executing: bin/kafka-server-start.sh config/server.properties.
5. Create a test topic that you can use for testing: bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic javaworld.
6. Start a simple console consumer that can consume messages published to a given topic, such as javaworld: bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic javaworld --from-beginning.
7. Start up a simple producer console that can publish messages to the test topic: bin/kafka-console-producer.sh --broker-list localhost:9092 --topic javaworld.
8. Try typing one or two messages into the producer console. Your messages should show in the consumer console.

If the producer publishes three messages the first one might get an offset of 1, the second an offset of 2, and the third an offset of 3.

When the Kafka consumer first starts, it will send a pull request to the server, asking to retrieve any messages for a particular topic with an offset value higher than 0. The server will check the log file for that topic and return the three new messages. The consumer will process the messages, then send a request for messages with an offset *higher* than 3, and so on.

In Kafka, the client is responsible for remembering the offset count and retrieving messages.The Kafka server doesn't track or manage message consumption. By default, a Kafka server will keep a message for seven days. A background thread in the server checks and deletes messages that are seven days or older. A consumer can access messages as long as they are on the server. It can read a message multiple times, and even read messages in reverse order of receipt. But if the consumer fails to retrieve the message before the seven days are up, it will miss that message.

## **Basic Topic Operations**

### Modifying a Topic

As you have already understood how to create a topic in Kafka Cluster. Now let us modify a created topic using the following command

Syntax

bin/kafka-topics.sh —zookeeper localhost:2181 --alter --topic topic\_name

--parti-tions count

Example

We have already created a topic “Hello-Kafka” with single partition count and one replica factor.

Now using “alter” command we have changed the partition count.

bin/kafka-topics.sh --zookeeper localhost:2181

--alter --topic Hello-kafka --parti-tions 2

Output

WARNING: If partitions are increased for a topic that has a key,

the partition logic or ordering of the messages will be affected

Adding partitions succeeded!

### Deleting a Topic

To delete a topic, you can use the following syntax.

Syntax

bin/kafka-topics.sh --zookeeper localhost:2181 --delete --topic topic\_name

Example

bin/kafka-topics.sh --zookeeper localhost:2181 --delete --topic Hello-kafka

Output

> Topic Hello-kafka marked for deletion

Note −This will have no impact if delete.topic.enable is not set to true

**Fault tolerance**

## **What is fault tolerance?**

The term fault tolerance is very common in distributed systems. It means, making your data available even in the case of some failures.

**How to do it?**   
One simple solution is to make multiple copies of the data and keep it on separate systems. So if you have three copies of a partition, and Kafka stores them on three different machines, you should be able to avoid two failures. Since you have three copies on three different systems, even if two of them fails, you can still read your data from the third system.

## **Replication Factor:**

There is a particular term used for making multiple copies. We call it replication factor. So, if I say, replication factor is three, that means, I am maintaining three copies of my partition. If I say replication factor is two, that means we are keeping two copies of a partition. The replication factor of three is a reasonable number. You can even set it to higher if your data is supercritical or you are using cheap machines.

## **Understanding Kafka Replication:**

You may want to understand how it works in Kafka. I mean, How Kafka make these copies? Let me explain that as well.   
Kafka implements a leader and follower model.   
So, for every partition, One Broker is elected as a leader. And the Leader takes care of all client interactions. What does that mean? That means when a producer is willing to send some data. It connects to the Leader and starts sending data. It is Leader's responsibility to receive the message, store it in local disk and send back an acknowledgment to the producer.   
Similarly, when a consumer is willing to read data, it sends a request to the leader. It is leader's responsibility to send requested data back to the consumer.   
For every partition, we have a leader, and the leader takes care of all requests and responses. I hope that part is clear.   
You may be wondering, that in all the above explanation, we haven't made any copy. That's where the followers come into play. So, if we create a topic with the replication factor set to three, A leader of the Topic is already maintaining the first copy. We need two more copies. So, Kafka will identify two more brokers as followers to make two copies. These Followers will copy the data from a leader. They don't talk to producer or consumer. They just copy data from a Leader. Simple, isn't it?

## **Multi-node Kafka Cluster**

So, let's make two more copies of the original properties file and modify them. Then we will use these modified files to start two more Brokers.

cp config/server.properties config/server-1.properties

cp config/server.properties config/server-2.properties

After executing the above commands, you should have two more property files. Now, I want to change three properties in these files. Let me explain those properties, and then, we will go ahead and modify them.

1. Broker id - The first property is Broker id. It's a unique identifier for the Broker. The default values for the first broker is zero, so we will change it to 1 for the second broker, and 2 for the third Broker. This change is to provide a unique identification to each broker.
2. Broker port - The next property is the Broker port. It's a network port number to which Broker will bind itself. The Broker will use this port number to communicate with producers and consumers. We will just increment it to whatever the default value is there. In fact, when you start brokers on separate systems, you don't need to change this port number, but since we are starting them on a single machine, we need to change it. Otherwise, all brokers will start reading and writing on the same port number.
3. Broker log directory - The third property is the Broker log directory. The Broker log directory is the main data directory of a Broker. We don't want all of the brokers to write into the same directory, so we need to change this value as well.

Great, go ahead, modify these properties in the files, and prepare a new file for other two brokers.   
You can start two more Brokers using those two new property files.

bin/kafka-server-start.sh config/server-1.properties

bin/kafka-server-start.sh config/server-2.properties

Great, you should have a three node Kafka cluster running.   
We did all of this because I wanted to create a topic with a replication factor three, and show you the leader and the follower for each partition. So, let's do that.

## **Kafka Topic Leader and Follower:**

You already know how to create a topic. Create a new topic with replication factor 3. We also make sure that we have atleast two partitions.

bin/kafka-topics.sh --zookeeper localhost:2181 --create --topic TestTopicXYZ --partitions 2 --replication-factor 2

## The *kafka-topics.sh* is a great tool to manage a Kafka Topic. We call it Topic management tool. This tool also provides a describe command.

bin/kafka-topics.sh --zookeeper localhost:2181 --describe --topic TestTopicXYZ

## **What is the ISR?**

The ISR is a list of In Sync Replicas. You might have three copies, but one of them may not be in sync with the leader. So, The ISR shows the list of replicas that are in sync with the Leader. In our case, all three are in sync.

# **Broker Configurations**

1. broker.id
2. port
3. log.dirs
4. zookeeper.connect
5. delete.topic.enable
6. auto.create.topics.enable
7. default.replication.factor
8. num.partitions
9. log.retention.ms
10. log.retention.bytes

I am skipping first three because we have already covered them in previous session.

## **Zookeeper.connect**

This parameter takes zookeeper connection string. The connection string is simply a hostname with a port number. We already know that Kafka uses zookeeper for various coordination purposes, so it is critical that every broker knows the zookeeper address. This parameter is also necessary to form a cluster.  
What do I mean by forming a Cluster?   
Well, all brokers are running on different systems, how do they know about each other. If they don't know each other, they are not part of the cluster. So, the zookeeper is the connecting link among all brokers to form a cluster.

## **delete.topic.enable**

If you want to delete a topic, you can use topic management tool. But by default, deleting a topic is not allowed. You can't remove a topic because the default value for this parameter is false. That is reasonable protection for production environments. But in development or testing environment, you may want to delete topics. So, if you want Kafka to allow deleting a topic, you need to set this parameter to true.

## **auto.create.topics.enable**

We have already discussed auto-create topic feature. If a producer starts sending messages to a non-existent topic, Kafka will create the topic automatically and accept the data. This behaviour is suitable for dev environments. But in a production environment, you may want to implement a more controlled approach. You can set this parameter to false, and Kafka will stop creating topics automatically. You can create topics manually using the topic management tool, and no one will be able to send data to a non-existent topic.

## **default.replication.factor and num.partitions**

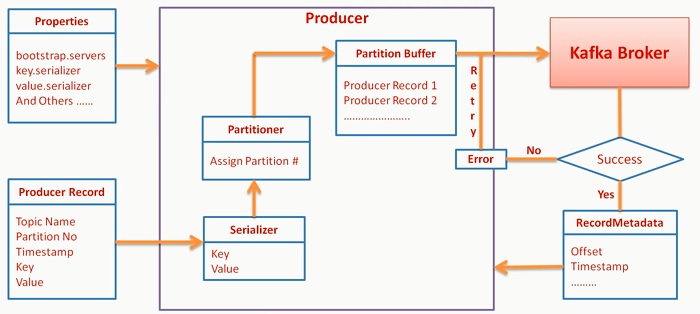
These two parameters are quite straightforward. The default values for both of them is one, and they are effective when you have auto create topics enabled. So, if Kafka is creating your topic automatically, the new topic will have only one partition and a single copy. If you want some other values, you can change the default settings accordingly.

## **log.retention.ms and log.retention.bytes**

These two are critical and not obvious. So, whatever data you send to Kafka, it is not retained by Kafka forever. Kafka is not a database. You don't send data to Kafka for storage so that you can query it later. It is a message broker. It should deliver the data to the consumer and then clean it up. There is no reason to retain messages for longer than needed.   
Kafka gives you two options to configure the retention period.

The default option is retention by time, and the default retention period is seven days. So, in this case, Kafka will clean up all the messages older than seven days. If you want to change the duration, you can specify your value for log.retention.ms configuration.   
Kafka gives you another option to define this retention period. You can specify it by size. That's where the second parameter log retention bytes is applicable. But this size applies to partition. So, if you set *log.retention.bytes = 1 GB*, Kafka will trigger a clean-up activity when the partition size reaches to 1 GB. Remember that it is not a topic size. It is partition size.   
If you specify both configurations, the clean-up will start on meeting either of the criteria.   
That's it for this session. I believe you are ready to start writing some code, so in next session, we will create a custom producer and send some data to Kafka.

**Producer Workflow**



## **Producer Configurations**

The first step is to create a Java properties object and package all the producer configurations that we want to set. These settings include three mandatory configurations that we learned in the previous session.

1. bootstrap.servers
2. key.serializer
3. value.serializer

You can also set some additional properties or even custom configs. In the example that we created earlier, we used only three basic configs, but I will create some more examples in next session with other properties including custom configs.

## **Producer Record**

On the other side, we create a producer record and package five things in a *ProducerRecord* object. Those five things are listed below.

1. Topic Name
2. Partition Number
3. Timestamp
4. Key
5. Value

The partition number, timestamp, and key are optional depending upon your use case. The *ProducerRecord* object is, in fact, the message that we want to send to Kafka Broker.   
Once we have the *Properties* and the *ProducerRecord* definition, we instantiate a *Producer* object using the *Properties* object. Then we send the*ProducerRecord* to the producer object. That’s it. The message is handed over to the producer. When the message is handed over to the producer, following things happen.

The Kafka server expects messages in byte[] key, byte[] value format. Rather than converting every key and value

## **Serialization**

The Producer will apply the serializer to serialize your Key and Value. That's the first thing. You already know that serialization is converting your Key and Value objects into a byte array, and the producer will use the serializer class that we specified to accomplish this.

## **Partitioning**

Then, it will send the record to the partitioner. The partitioner will choose a partition for the message. We already discussed earlier that the default partitioner would use your message key to determine an appropriate partition. If a message key is specified, Kafka will hash the key for getting a partition number. If you specify the same key to multiple messages, all of them will go to the same partition.   
If message key is not specified, the default partitioner will try to evenly distribute the messages to all available partitions for the topic. It uses a round robin algorithm, so few messages go to the first partition, then some of them goes to second and so on.

## **Partition Buffer**

Once we have a partition number, the partitioner is ready to send it to Broker. But instead of sending the message immediately, the partitioner will keep the message into a partition buffer. The producer maintains an in-memory buffers for each partition and sends the records in batches. You might be wondering that what is the size of the batch? How much time the producer will linger waiting for more messages to arrive? We can configure all those things by adding appropriate configuration parameters to the properties object. I will cover them in the upcoming sessions.

## **Record Metadata and Retires**

Finally, the producer will send a batch of records to the broker. If the broker can receive and save the message, it will send an acknowledgment in the form of*RecordMetadata* object. If anything goes wrong, the producer receives an error. Some errors may be recoverable with a retry, for example, suppose the leader of the partition was down, if we retry sending the batch in few milliseconds, we may have a new leader elected by that time. So, In the case of a recoverable error, the producer will retry sending the batch before it throws and exception.   
We can configure the number of retries and time between two retires. The producer will not attempt for a retry if the error is not a recoverable error.   
Great, the workflow of a producer is quite simple, and we can configure almost everything using the producer configuration parameters. 

# **Callback and Acknowledgment**

There are three approaches to send a message to Kafka.

* **Fire-and-forget Producer**

Send and forget is the simplest approach. In this method, we send a message to the broker and don’t care if it was successfully received or no.

|  |
| --- |
| import java.util.\*;  import org.apache.kafka.clients.producer.\*;  public class SimpleProducer {    public static void main(String[] args) throws Exception{    String topicName = "SimpleProducerTopic";  String key = "Key1";  String value = "Value-1";    Properties props = new Properties();  props.put("bootstrap.servers", "localhost:9092,localhost:9093");  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);    ProducerRecord<String, String> record = new ProducerRecord<>(topicName,key,value);  producer.send(record);  producer.close();    System.out.println("SimpleProducer Completed.");  }  } |

* **Synchronous Producer**

In this approach, we send a message and wait until we get a response. In the case of a success, we get a RecordMetadata object, and in the event of failure, we get an exception. Most of the time, we don't care about the success and the RecordMetadata that we receive. We only care about exception because we want to log errors for later analysis and appropriate action. You can adopt this method if your messages are critical and you can't afford to lose anything. But it is important to notice that synchronous approach will slow you down. It will limit your throughput because you are waiting for every message to get acknowledged. You are sending one message and waiting for success, then you send your next message, and again wait for success. Each message will take some time to be delivered over the network. So, after every message, you will be waiting for a network delay, and the most interesting thing is that, you may not be doing anything in case of success. You care only failures, if it fails, you may want to take some action.   
We have already seen an example of fire and forget approach. Let's look at the example of synchronous send.

|  |
| --- |
| import java.util.\*;  import org.apache.kafka.clients.producer.\*;  public class SynchronousProducer {    public static void main(String[] args) throws Exception{    String topicName = "SynchronousProducerTopic";  String key = "Key-1";  String value = "Value-1";    Properties props = new Properties();  props.put("bootstrap.servers", "localhost:9092,localhost:9093");  props.put("key.serializer","org.apache.kafka.common.serialization.StringSerializer");  props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");  Producer producer = new KafkaProducer<>(props);  ProducerRecord record = new ProducerRecord<>(topicName,key,value);    try{  RecordMetadata metadata = producer.send(record).get();  System.out.println("Message is sent to Partition no " + metadata.partition() + " and offset " + metadata.offset());  System.out.println("SynchronousProducer Completed with success.");  }catch (Exception e) {  e.printStackTrace();  System.out.println("SynchronousProducer failed with an exception");  }finally{  producer.close();  }  }  } |

* **Asynchronous Producer**

In this method, we send a message and provide a call back function to receive acknowledgment. We don't wait for success and failure. The producer will callback our function with RecordMetadata and an exception object. So, if you just care about exceptions, you simply look at the exception, if it is null, don't do anything. If the exception is not null, it then you know failed, so you can record the message details for later analysis.   
Let's look at an example of the asynchronous approach

|  |
| --- |
| import java.util.\*;  import org.apache.kafka.clients.producer.\*;    public class AsynchronousProducer {    public static void main(String[] args) throws Exception{  String topicName = "AsynchronousProducerTopic";  String key = "Key1";  String value = "Value-1";    Properties props = new Properties();  props.put("bootstrap.servers", "localhost:9092,localhost:9093");  props.put("key.serializer","org.apache.kafka.common.serialization.StringSerializer");  props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");    Producer producer = new KafkaProducer<>(props);  ProducerRecord record = new ProducerRecord<>(topicName,key,value);    producer.send(record, new MyProducerCallback());  System.out.println("AsynchronousProducer call completed");  producer.close();    }  }    class MyProducerCallback implements Callback{    @Override  public voidonCompletion(RecordMetadatarecordMetadata, Exception e) {  if (e != null)  System.out.println("AsynchronousProducer failed with an exception");  else  System.out.println("AsynchronousProducer call Success:");  }  } |

* **Async Interface Callback and Async Send Method:**

Kafka defines a [Callback](https://kafka.apache.org/0100/javadoc/org/apache/kafka/clients/producer/Callback.html) interface that you use for asynchronous operations. The callback interface allows code to execute when the request is complete. The callback executes in a background I/O thread so it should be fast (don’t block it). The onCompletion(RecordMetadata metadata, Exception exception) gets called when the asynchronous operation completes. The [metadata](https://kafka.apache.org/0100/javadoc/org/apache/kafka/clients/producer/RecordMetadata.html) gets set (not null) if the operation was a success, and the exception gets set (not null) if the operation had an error.

The async send method is used to send a record to a topic, and the provided callback gets called when the send is acknowledged. The send method is asynchronous, and when called returns immediately once the record gets stored in the buffer of records waiting to post to the Kafka broker. The send method allows sending many records in parallel without blocking to wait for the response after each one.

* **In Flight Messages**

The asynchronous method appears to provide you a throughput that is as good as fire and forget approach. But there is a catch. You have a limit of in-flight messages. This limit is controlled by a configuration parameter max.in.flight.requests.per.connection. This parameter controls that how many messages you can send to the server without receiving a response. The default value is 5. You can increase this value, but there are other considerations. We will cover producer configurations in a separate session. Till then, just understand that asynchronous send gives you a better throughput compared to synchronous, but the max.in.flight.requests.per.connection limits it.

### Review Kafka Producer

#### **What does the Callback lambda do?**

The callback gets notified when the request is complete.

#### **What will happen if the first server is down in the bootstrap list? Can the producer still connect to the other Kafka brokers in the cluster?**

The producer will try to contact the next broker in the list. Any of the brokers once contacted, will let the producer know about the entire Kafka cluster. The Producer will connect as long as at least one of the brokers in the list is running. If you have 100 brokers and two of the brokers in a list of three servers in the bootstrap list are down, the producer can still use the 98 remaining brokers.

#### **When would you use Kafka async send vs. sync send?**

If you were already using an async code (Akka, QBit, Reakt, Vert.x) base, and you wanted to send records quickly.

#### **Why do you need two serializers for a Kafka record?**

One of the serializers is for the Kafka record key, and the other is for the Kafka record value.

# **Partitions in Kafka**

Topics in Kafka can be subdivided into partitions. For example, while creating a topic named Demo, you might configure it to have three partitions. The server would create three log files, one for each of the demo partitions. When a producer published a message to the topic, it would assign a partition ID for that message. The server would then append the message to the log file for that partition only.

If you then started two consumers, the server might assign partitions 1 and 2 to the first consumer, and partition 3 to the second consumer. Each consumer would read only from its assigned partitions. You can see the Demo topic configured for three partitions in Figure 1.

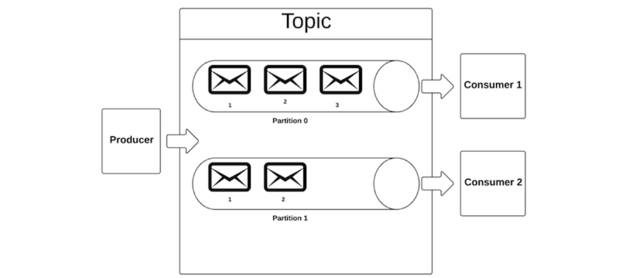


Figure 1. A partitioned topic in Apache Kafka

To expand the scenario, imagine a Kafka cluster with two brokers, housed in two machines. When you partitioned the demo topic, you would configure it to have two partitions and two replicas. For this type of configuration, the Kafka server would assign the two partitions to the two brokers in your cluster. Each broker would be the leader for one of the partitions.

When a producer published a message, it would go to the partition leader. The leader would take the message and append it to the log file on the local machine. The second broker would passively replicate that commit log to its own machine. If the partition leader went down, the second broker would become the new leader and start serving client requests. In the same way, when a consumer sent a request to a partition, that request would go first to the partition leader, which would return the requested messages.

# **Custom Partitioning**

**Two ways to partition**

The producer is responsible for deciding what partition a message will go to. The producer has two options for controlling this assignment:

* **Custom partitioner**: You can create a class implementing the org.apache.kafka.clients.producer.Partitioner interface. This custom Partitioner will implement the business logic to decide where messages are sent. See: Partitioning in Kafka Example
* **DefaultPartitioner**: If you don’t create a custom partitioner class, then by default the org.apache.kafka.clients.producer.internals.DefaultPartitionerclass will be used. The default partitioner is good enough for most cases, providing three options:
  1. **Manual**: When you create a ProducerRecord, use the overloaded constructor new ProducerRecord(topicName, partitionId,messageKey,message) to specify a partition ID.
  2. **Hashing(Locality sensitive)**: When you create a ProducerRecord, specify a messageKey, by calling new ProducerRecord(topicName,messageKey,message). DefaultPartitioner will use the hash of the key to ensure that all messages for the same key go to same producer. This is the easiest and most common approach.
  3. **Spraying(Random Load Balancing)**: If you don’t want to control which partition messages go to, simply call new ProducerRecord(topicName, message) to create your ProducerRecord. In this case the partitioner will send messages to all the partitions in round-robin fashion, ensuring a balanced server load.

|  |
| --- |
| **public** **class** CountryPartitioner **implements** Partitioner {  **public** **void** configure(Map<String, ?> configs) {  // **TODO** Auto-generated method stub  }  **public** **int** partition(String topic, Object key, **byte**[] keyBytes, Object value, **byte**[] valueBytes, Cluster cluster) {  // **TODO** Auto-generated method stub  **return** 0;  }  **public** **void** close() {  // **TODO** Auto-generated method stub  }  } |

|  |
| --- |
| **CustomPartioner:**  **import** org.apache.kafka.clients.producer.Partitioner;  **import** org.apache.kafka.common.Cluster;  **import** java.util.HashMap;  **import** java.util.List;  **import** java.util.Map;  **public** **class** CountryPartitioner **implements** Partitioner {  **private** **static** Map<String,Integer> *countryToPartitionMap*;  // This method will gets called at the start, you should use it to do one time startup activity  **public** **void** configure(Map<String, ?> configs) {  System.***out***.println("Inside CountryPartitioner.configure " + configs);  *countryToPartitionMap* = **new** HashMap<String, Integer>();  **for**(Map.Entry<String,?> entry: configs.entrySet()){  **if**(entry.getKey().startsWith("partitions.")){  String keyName = entry.getKey();  String value = (String)entry.getValue();  System.***out***.println( keyName.substring(11));  **int** paritionId = Integer.*parseInt*(keyName.substring(11));  *countryToPartitionMap*.put(value,paritionId);  }  }  }  //This method will get called once for each message  **public** **int** partition(String topic, Object key, **byte**[] keyBytes, Object value, **byte**[] valueBytes,  Cluster cluster) {  List partitions = cluster.availablePartitionsForTopic(topic);  String valueStr = (String)value;  String countryName = ((String) value).split(":")[0];  **if**(*countryToPartitionMap*.containsKey(countryName)){  //If the country is mapped to particular partition return it  **return** *countryToPartitionMap*.get(countryName);  }**else** {  //If no country is mapped to particular partition distribute between remaining partitions  **int** noOfPartitions = cluster.topics().size();  **return** value.hashCode()%noOfPartitions + *countryToPartitionMap*.size() ;  }  }  // This method will get called at the end and gives your partitioner class chance to cleanup  **public** **void** close() {}   * }   Producer:  **import** org.apache.kafka.clients.producer.\*;  **import** java.util.Properties;  **import** java.util.Scanner;  **public** **class** Producer {  **private** **static** Scanner *in*;  **public** **static** **void** main(String[] argv)**throws** Exception {  **if** (argv.length != 1) {  System.***err***.println("Please specify 1 parameters ");  System.*exit*(-1);  }  String topicName = argv[0];  *in* = **new** Scanner(System.***in***);  System.***out***.println("Enter message(type exit to quit)");  //Configure the Producer  Properties configProperties = **new** Properties();  configProperties.put(ProducerConfig.***BOOTSTRAP\_SERVERS\_CONFIG***,"localhost:9092");  configProperties.put(ProducerConfig.***KEY\_SERIALIZER\_CLASS\_CONFIG***,"org.apache.kafka.common.serialization.ByteArraySerializer");  configProperties.put(ProducerConfig.***VALUE\_SERIALIZER\_CLASS\_CONFIG***,"org.apache.kafka.common.serialization.StringSerializer");  configProperties.put(ProducerConfig.***PARTITIONER\_CLASS\_CONFIG***,CountryPartitioner.**class**.getCanonicalName());  configProperties.put("partitions.0","USA");  configProperties.put("partitions.1","India");  org.apache.kafka.clients.producer.Producer producer = **new** KafkaProducer(configProperties);  String line = *in*.nextLine();  **while**(!line.equals("exit")) {  ProducerRecord<String, String> rec = **new** ProducerRecord<String, String>(topicName, line);  producer.send(rec, **new** Callback() {  **public** **void** onCompletion(RecordMetadata metadata, Exception exception) {  System.***out***.println("Message sent to topic ->" + metadata.topic()+ " ,parition->" + metadata.partition() +" stored at offset->" + metadata.offset());  }  });  line = *in*.nextLine();  }  *in*.close();  producer.close();  }  } |

### Assigning partitions to consumers

The Kafka server guarantees that a partition is assigned to only one consumer, thereby guaranteeing the order of message consumption. You can manually assign a partition or have it assigned automatically.

If your business logic demands more control, then you’ll need to manually assign partitions. In this case you would use KafkaConsumer.assign(<listOfPartitions>) to pass a list of partitions that each consumer was interested in to the Kakfa server.

Having partitions assigned automatically is the default and most common choice. In this case, the Kafka server will assign a partition to each consumer, and will reassign partitions to scale for new consumers.

Say you’re creating a new topic with three partitions. When you start the first consumer for the new topic, Kafka will assign all three partitions to the same consumer. If you then start a second consumer, Kafka will reassign all the partitions, assigning one partition to the first consumer and the remaining two partitions to the second consumer. If you add a third consumer, Kafka will reassign the partitions again, so that each consumer is assigned a single partition. Finally, if you start fourth and fifth consumers, then three of the consumers will have an assigned partition, but the others won’t receive any messages. If one of the initial three partitions goes down, Kafka will use the same partitioning logic to reassign that consumer’s partition to one of the additional consumers.

Being able to partition a single topic into multiple parts is one essential to Kafka’s scalability. Partitioning lets you scale your messaging infrastructure horizontally while also maintaining order within each partition.

## **Managing message offsets**

Whenever a producer publishes a message, the Kafka server assigns an offset to that message. A consumer is able to control which messages it wants to consume by setting or resetting the message offset. When developing a consumer you have two options for managing the offset: automatic and manual.

The offset is a simple integer number that is used by Kafka to maintain the current position of a consumer

The offset is a position within a partition for the next message to be sent to a consumer. Kafka maintains two types of offsets.

1. Current offset
2. Committed offset

**Current offset** -> Sent Records -> This is used to avoid resending same records again to the same consumer.

**Committed offset** -> Processed Records -> It is used to avoid resending same records to a new consumer in the event of partition rebalance.

The committed offset is critical in the case of partition rebalance.   
In the event of rebalancing. When a new consumer is assigned a new partition, it should ask a question. Where to start? What is already processed by the previous owner? The answer to the question is the committed offset.

## **How to commit an offset?**

Now, since we understand both the offsets maintained by Kafka, the next question is, How to commit an offset?   
**There are two ways to do it.**

1. Auto commit
2. Manual-commit

The commit has a significant impact on the client application, so we need to choose an appropriate method based on our use case. Let us look at the auto-commit approach.

## Auto Commit

Auto-commit is the easiest method. You can control this feature by setting two properties.

1. enable.auto.commit
2. auto.commit.interval.ms

The first property is by default *true.* So auto-commit is enabled by default. You can turn it off by setting *enable.auto.commit* to false. The second property defines the interval of auto-commit. The default value for this property is five seconds. So, in a default configuration, when you make a call to the poll method, it will check if it is time to commit. If you have passed five seconds since the previous call, the consumer will commit the last offset. So, Kafka will commit your current offset every five seconds.

## **Manual Commit**

The solution to this particular problem is a manual commit. So, we can configure the auto-commit off and manually commit after processing the records. There are two approaches to manual commit.

1. Commit Sync
2. Commit async

I hope you already understand the difference between synchronous and asynchronous. Synchronous commit is a straightforward and reliable method, but it is a blocking method. It will block your call for completing a commit operation, and it will also retry if there are recoverable errors.   
Asynchronous commit will send the request and continue. The drawback is that *commitAsync* will not retry. But there is a valid reason for such behaviour. Let's understand it with an example.

|  |
| --- |
| import java.util.\*;  import java.io.\*;  import org.apache.kafka.clients.consumer.KafkaConsumer;  import org.apache.kafka.clients.consumer.ConsumerRecords;  import org.apache.kafka.clients.consumer.ConsumerRecord;    public class ManualConsumer{    public static void main(String[] args) throws Exception{    String topicName = "SupplierTopic";  String groupName = "SupplierTopicGroup";    Properties props = new Properties();  props.put("bootstrap.servers", "localhost:9092,localhost:9093");  props.put("group.id", groupName);  props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");  props.put("value.deserializer", "SupplierDeserializer");  props.put("enable.auto.commit", "false");    KafkaConsumer consumer = null;    try {  consumer = new KafkaConsumer<>(props);  consumer.subscribe(Arrays.asList(topicName));    while (true){  ConsumerRecords records = consumer.poll(100);  for (ConsumerRecordrecord : records){  System.out.println("Supplier id= " + String.valueOf(record.value().getID()) +  " Supplier Name = " + record.value().getName() +  " Supplier Start Date = " + record.value().getStartDate().toString());  }  consumer.commitAsync();  }  }catch(Exception ex){  ex.printStackTrace();  }finally{  consumer.commitSync();  consumer.close();  }  }  } |

The code is straightforward, and we have already seen it earlier. There is nothing new except two new lines. The first one is asynchronous commit and the second one is synchronous commit. In this example, I am manually committing my current offset before pulling the next set of records.   
You may be wondering that does it solve my problem completely.   
I mean, I got 100 records in the first poll. After processing all 100 records, I am committing my current offset.

1. What if a rebalance occurs after processing 50 records?
2. What if an exception occurs after processing 50 records?

I leave these two questions for you to think and post me an answer as a comment or start a discussion on these two issues.

Let me give you a hint.   
You can fix both above problems if you know how to commit a particular offset instead of committing current offset. Kafka offset management and handling rebalance gracefully is the most critical part of implementing appropriate Kafka consumers.

***ConsumerRebalanceListener:***

The synchronous and asynchronous commit that we learned earlier is committing the latest offset given by the last poll. We don't want that. I will show you an example that will maintain a current offset of processed records and commit the current offset when a rebalancing is triggered.   
The answer to the second question is simple. Kafka API allows us to specify a ConsumerRebalanceListener class. This class offers two methods.

1. onPartitionsRevoked
2. onPartitionsAssigned

The Kafka will call the onPartitionsRevoked method just before it takes away your partitions. So, this is where you can commit your current offset.   
The Kafka will call the onPartitionsAssigned method right after the rebalancing is complete and before you start consuming records from the new partitions.

https://www.learningjournal.guru/courses/kafka/kafka-foundation-training/offset-management/

### Two types of offset

When you start a consumer in the Kafka client, it will read the value of your ConsumerConfig.AUTO\_OFFSET\_RESET\_CONFIG(auto.offset.reset)configuration. If that config is set to earliest then the consumer will start with the smallest offset available for the topic. In its first request to Kafka, the consumer will say: give me all the messages in this partition with an offset greater than the smallest one available. It will also specify a batch size. The Kafka server will return it all the matching messages in batches of the specified size.

The consumer keeps track of the offset of the last message it has processed, so it will always request messages with an offset higher than the last offset. This setup works when a consumer is functioning normally, but what happens if the consumer crashes, or you want to stop it for maintenance? In this case you would want the consumer to remember the offset of last message processed, so that it can start with the first unprocessed message.

In order to ensure message persistence, Kafka uses two types of offset: The current offset is used to track messages consumed when the consumer is working normally. The committed offset also tracks the last message offset, but it sends that information to the Kafka server for persistent storage.

If the consumer goes down or is taken down for some reason, it can query the Kafka server for the last committed offset and resume message consumption as if no time has been lost. For its part, the Kafka broker stores this information in a topic called **3** This data is replicated to multiple brokers so that the broker won’t ever lose the offsets.

### Committing offset data

You have a choice about how often to commit offset data. If you commit frequently, you’ll take a performance penalty. On the other hand, if the consumer does go down you will have fewer messages to reprocess and consume. Your other option is to commit less frequently (for better performance), but reprocess more messages in case of failure. In either case the consumer has two options for committing the offset:

1. **Auto commits**: You can set auto.commit to true and set the auto.commit.interval.ms property with a value in milliseconds. Once you’ve enabled this, the Kafka consumer will commit the offset of the last message received in response to its poll() call. The poll() call is issued in the background at the set auto.commit.interval.ms.
2. **Manual commits**: You can call a commitSync() or commitAsync() method anytime on the KafkaConsumer. When you issue the call, the consumer will take the offset of the last message received during a poll() and commit that to the Kafka server.

## Three use cases for manual offsets

Let’s consider three use cases where you wouldn’t want to use Kafka’s default offset management infrastructure. Instead, you’ll manually decide what message to to start from.

1. **Start from the beginning**: In this use case, you are capturing database changes in Kafka. The first record was the full record; thereafter you only get columns whose value has changed (delta of changes). In this case you always need to read all the messages in a topic from the beginning, in order to construct the full state of the record. To solve a scenario like this, you can configure the consumer to read from the beginning by calling **the kafkaConsumer.seekToBeginning(topicPartition) method. Remember that by default Kafka will delete messages more than seven days old, so you need to configure log.retention.hours to a higher value for this use case.**

**In service.properties:**

# The minimum age of a log file to be eligible for deletion due to age

log.retention.hours=168

1. **Go to the end**: Now let’s say you’re building a stock recommendation application by analyzing trades in realtime. The worst case happens and your consumer application goes down. In this case, you’ve **used kafkaConsumer.seekToEnd(topicPartition) to configure the offset to ignore** messages that are posted during downtime. Instead, the consumer will begin processing trades that are happening from the instant that it restarts.
2. **Start at a given offset**: Finally, say that you just released a new version of the producer in your production environment. After watching it produce a few messages, you realize that it is generating bad messages. You fix the producer and start it again. **You don’t want your consumer to consume those bad messages, so you manually set the offset to the first good message produced, by calling kafkaConsumer.seek(topicPartition, startingOffset).**

## **Consumer groups in Kafka**

Traditional messaging use cases can be divided into two main types: point to point and publish-subscribe. In a point-to-point scenario, one consumer consumes one message. When a message relays a bank transaction, only one consumer should respond by updating the bank account. In a publish-subscribescenario, multiple consumers will consume a single message but respond differently to it. When a web server goes down, you want the alert to go to consumers programmed to respond in different ways.

Queue refers to a point-to-point scenario, where a message is consumed by only one consumer. Topic refers to a publish-subscribe scenario, where a message is consumed by every consumer. Kafka doesn’t define a separate API for the queue and topic use cases; instead, when you start your consumer you need to specify the ConsumerConfig.GROUP\_ID\_CONFIG property.

If you use the same GROUP\_ID\_CONFIG for more than one consumer, Kafka will assume that both of them are part of a single group, and it will deliver messages to only one of the consumers. If you start the two consumers in separate group.ids, Kafka will assume that they are not related, so each consumer will get its own copy of the message.

**Link for website:**

<http://blog.empeccableweb.com/wp/bigdata/kafka/>

# **Consumer Groups**

**When a consumer joins a group, how is a partition assigned to it? Moreover, what happens to the partition when a consumer leaves the group? Who manages all of this**

## **Kafka Group Coordinator:**

The answer is simple. A group coordinator oversees all of this. So, one of the Kafka broker gets elected as a Group Coordinator. When a consumer wants to join a group, it sends a request to the coordinator. The first consumer to participate in a group becomes a leader. All other consumers joining later becomes the members of the group.   
So, we have two actors, A coordinator, and a group leader. The coordinator is responsible for managing a list of group members. So, every time a new member joins the group, or an existing member leaves the group, the coordinator modifies the list.   
On an event of membership change, the coordinator realizes that it is time to rebalance the partition assignment. Because you may have a new member, and you need to assign it some partitions, or a member left, and you need to reassign those partitions to someone else, So, every time the list is modified, the coordinator initiates a rebalance activity.

## **Kafka Group Leader:**

The group leader is responsible for executing rebalance activity. The group leader will take a list of current members, assign partitions to them and send it back to the coordinator. The Coordinator then communicates back to the members about their new partitions. The important thing to note here is, during the rebalance activity, none of the consumers are allowed to read any message.

## **Summary**

Let us summarize it quickly.

1. Consumer Groups –They are used to read and process data in parallel.
2. Partitions are not shared - To protect duplicate reads in a group, Kafka does not allow more than one Consumers to read data from a single partition at the same time.
3. A Group Coordinator - A broker is designated as a group coordinator and it maintains a list of active consumers.
4. Rebalance - Every time the list of active consumers is modified, the coordinator orders a rebalance activity to the group leader.
5. The Group leader - executes a rebalance activity.

Rebalance activity is nothing but assigning partitions to individual consumers.

 if somehow any consumer or broker fails to send heartbeat to ZooKeeper, then it can be re-configured via the Kafka cluster. Also, Kafka will assign available partitions to the available threads, possibly moving a partition to another process, during this re-balance.

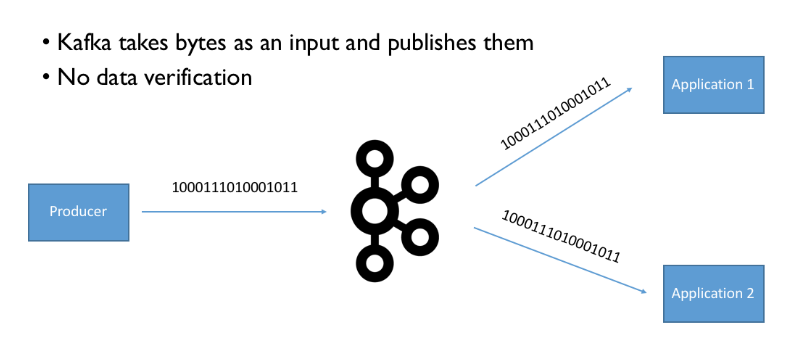
**AVRO**

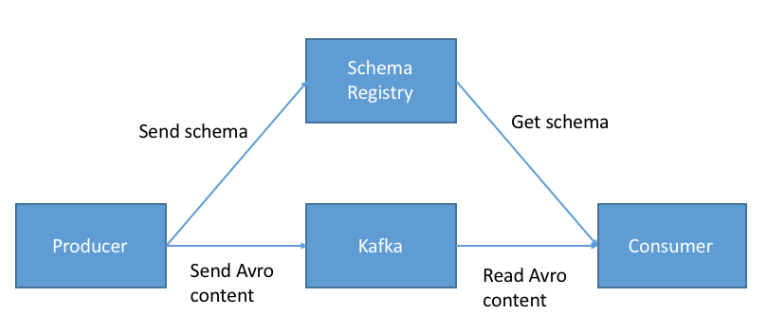
Avro has grown in popularity in the Big Data community. It also has become the favorite Fast-Data serialization format thanks to a big push by Confluent (due to the Confluent Schema Registry).

#### **Schema as a first-class citizen**

Similarly to how in a SQL database you can’t add data without creating a table first, one can’t create an Avro object without first providing a schema.

* **Avro has support for primitive types**( int, string, long, bytes, etc…), complex types (enum, arrays, unions, optionals), logical types (dates, timestamp-millis, decimal), and data record (name and namespace). All the types you’ll ever need.
* **Avro schemas are defined using JSON**. Because every developer knows or can easily learn JSON, there’s a very low barrier to entry.
* **An Avro object contains the schema and the data**. The data without the schema is an invalid Avro object. That’s a big difference with say, CSV, or JSON.
* Avro data serialization is efficient in space, can be read by any language





## **Creating Avro Schemas**

The Avro schema is created in JavaScript Object Notation (JSON) document format, which is a lightweight text-based data interchange format. It is created in one of the following ways

* A JSON string
* A JSON object
* A JSON array

**Example** − The given schema defines a (record type) document within "Tutorialspoint" namespace. The name of document is "Employee" which contains two "Fields" → Name and Age.

{

"type" : "record",

"namespace" : "Tutorialspoint",

"name" : "Employee",

"fields" : [

{ "name" : "Name" , "type" : "string" },

{ "name" : "Age" , "type" : "int" }

]

}

We observed that schema contains four attributes, they are briefly described below −

* **type** − Describes document type, in this case a "record".
* **namespace** − Describes the name of the namespace in which the object resides.
* **name** − Describes the schema name.
* **fields** − This is an attribute array which contains the following −
  + **name** − Describes the name of field
  + **type** − Describes data type of field

## **Primitive Data Types of Avro**

Avro schema is having primitive data types as well as complex data types. The following table describes the **primitive data types** of Avro −

|  |  |
| --- | --- |
| **Data type** | **Description** |
| null | Null is a type having no value. |
| int | 32-bit signed integer. |
| long | 64-bit signed integer. |
| float | single precision (32-bit) IEEE 754 floating-point number. |
| double | double precision (64-bit) IEEE 754 floating-point number. |
| bytes | sequence of 8-bit unsigned bytes. |
| string | Unicode character sequence. |

## **Complex Data Types of Avro**

Along with primitive data types, Avro provides six complex data types namely Records, Enums, Arrays, Maps, Unions, and Fixed.

### Record

As we know already by now, a record data type in Avro is a collection of multiple attributes. It supports the following attributes −

* **name**
* **namespace**
* **type**
* **fields**

### Enum

An enumeration is a list of items in a collection, Avro enumeration supports the following attributes −

* **name** − The value of this field holds the name of the enumeration.
* **namespace** − The value of this field contains the string that qualifies the name of the Enumeration.
* **symbols** − The value of this field holds the enum's symbols as an array of names.

**Example**

Given below is the example of an enumeration.

{

"type" : "enum",

"name" : "Numbers", "namespace": "data", "symbols" : [ "ONE", "TWO", "THREE", "FOUR" ]

}

### Arrays

This data type defines an array field having a single attribute items. This items attribute specifies the type of items in the array.

**Example**

{ " type " : " array ", " items " : " int " }

### Maps

The map data type is an array of key-value pairs. The **values** attribute holds the data type of the content of map. Avro map values are implicitly taken as strings. The below example shows map from string to int.

**Example**

{"type" : "map", "values" : "int"}

### Unions

A union datatype is used whenever the field has one or more datatypes. They are represented as JSON arrays. For example, if a field that could be either an int or null, then the union is represented as ["int", "null"].

**Example**

Given below is an example document using unions −

{

"type" : "record",

"namespace" : "tutorialspoint",

"name" : "empdetails ",

"fields" :

[

{ "name" : "experience", "type": ["int", "null"] }, { "name" : "age", "type": "int" }

]

}

### Fixed

This data type is used to declare a fixed-sized field that can be used for storing binary data. It has field name and data as attributes. Name holds the name of the field, and size holds the size of the field.

**Example**

{ "type" : "fixed" , "name" : "bdata", "size" : 1048576}

**Schema definition:**

{"namespace": "example.avro",

"type": "record",

"name": "User",

"fields": [

{"name": "name", "type": "string"},

{"name": "favorite\_number", "type": ["int", "null"]},

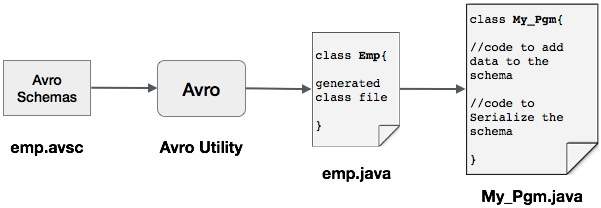
{"name": "favorite\_color", "type": ["string", "null"]}

]

}

**Process Serializing and DeSerializing**

The following is a depiction of serializing the data with Avro by generating a class. Here, emp.avsc is the schema file which we pass as input to Avro utility.



**The output of Avro is a java file.**

## Serialization by Generating a Class

To serialize the data using Avro, follow the steps as given below −

* Define an Avro schema.
* Compile the schema using Avro utility. You get the Java code corresponding to that schema.
* Populate the schema with the data.
* Serialize it using Avro library.

## **Defining a Schema**

Suppose you want a schema with the following details −

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Field | Name | id | age | salary | address |
| type | String | int | int | int | string |

Create an Avro schema as shown below and save it as emp.avsc.

{

"namespace": "com.mng.avro.sample",

"type": "record",

"name": "Employee",

"fields": [

{"name": "name", "type": "string"},

{"name": "id", "type": "int"},

{"name": "salary", "type": "int"},

{"name": "age", "type": "int"},

{"name": "address", "type": "string"}

]

}

## **Compiling the Schema**

### java -

### Serializing

Now that we've created our user objects, serializing and deserializing them is almost identical to the example above which uses code generation. The main difference is that we use generic instead of specific readers and writers.

First we'll serialize our users to a data file on disk.

// Serialize

|  |
| --- |
| **package** com.mng.avro.AvroDemo;  **import** java.io.File;  **import** org.apache.avro.file.DataFileWriter;  **import** org.apache.avro.io.DatumWriter;  **import** org.apache.avro.specific.SpecificDatumWriter;  **import** com.mng.avro.sample.Employee;  /\*\*  \* Hello world!  \*  \*/  **public** **class** AvroSerializeRunner {  **public** **static** **void** main(String[] args) {  Employee e1 = **new** Employee();  e1.setName("Nagendra");  e1.setAddress("Banglore city");  e1.setAge(30);  e1.setId(101);  e1.setSalary(4000);  Employee e2 = **new** Employee();  e2.setName("XyZ");  e2.setAddress("Hyderbad city");  e2.setAge(30);  e2.setId(102);  e2.setSalary(5000);  Employee e3 = **new** Employee();  e3.setName("atish");  e3.setAddress("Ananthpur city");  e3.setAge(30);  e3.setId(103);  e3.setSalary(6000);  File outPutfile = **new** File("emp\_record.avsc");  DataFileWriter<Employee> dataFileWriter = **null**;  **try** {  DatumWriter<Employee> datumWriter = **new** SpecificDatumWriter<Employee>(Employee.**class**);  dataFileWriter = **new** DataFileWriter<Employee>(datumWriter);  dataFileWriter.create(e1.getSchema(), outPutfile);  dataFileWriter.append(e1);  dataFileWriter.append(e2);  dataFileWriter.append(e3);  dataFileWriter.close();  System.***out***.println("data successfully serialized");  } **catch** (Exception e) {  } **finally** {  }  }  } |

We create a DatumWriter, which converts Java objects into an in-memory serialized format. Since we are not using code generation, we create a GenericDatumWriter. It requires the schema both to determine how to write the GenericRecords and to verify that all non-nullable fields are present.

As in the code generation example, we also create a DataFileWriter, which writes the serialized records, as well as the schema, to the file specified in the dataFileWriter.createcall. We write our users to the file via calls to the dataFileWriter.append method. When we are done writing, we close the data file.

### Deserializing

Finally, we'll deserialize the data file we just created.

// Deserialize users from disk

|  |
| --- |
| **package** com.mng.avro.AvroDemo;  **import** java.io.File;  **import** org.apache.avro.file.DataFileReader;  **import** org.apache.avro.io.DatumReader;  **import** org.apache.avro.specific.SpecificDatumReader;  **import** com.mng.avro.sample.Employee;  **public** **class** AvroDeSerializeRunner {  **private** **static** DataFileReader<Employee> *dataFileReader*;  **public** **static** **void** main(String[] args) {  File outPutfile = **new** File("emp\_record.avsc");  **try** {    DatumReader<Employee> datumReader = **new** SpecificDatumReader<Employee>(Employee.**class**);  *dataFileReader* = **new** DataFileReader<Employee>(outPutfile, datumReader);  Employee employee = **null**;    **while** (*dataFileReader*.hasNext()) {  // Reuse user object by passing it to next(). This saves us from  // allocating and garbage collecting many objects for files with  // many items.  employee = *dataFileReader*.next(employee);  System.***out***.println("data successfully Deserialized");  }  } **catch** (Exception e) {  System.***out***.println("Error writing Avro");  } **finally** {  }  }  } |

**This outputs:**

{"name": "Nagendra", "id": 101, "salary": 4000, "age": 30, "address": "Banglore city"}

{"name": "XyZ", "id": 102, "salary": 5000, "age": 30, "address": "Hyderbad city"}

{"name": "atish", "id": 103, "salary": 6000, "age": 30, "address": "Ananthpur city"}

data successfully Deserialized

Deserializing is very similar to serializing. We create a GenericDatumReader, analogous to the GenericDatumWriter we used in serialization, which converts in-memory serialized items into GenericRecords. We pass the DatumReader and the previously created File to a DataFileReader, analogous to the DataFileWriter, which reads the data file on disk.

Next, we use the DataFileReader to iterate through the serialized users and print the deserialized object to stdout. Note how we perform the iteration: we create a singleGenericRecord object which we store the current deserialized user in, and pass this record object to every call of dataFileReader.next. This is a performance optimization that allows the DataFileReader to reuse the same record object rather than allocating a new GenericRecord for every iteration, which can be very expensive in terms of object allocation and garbage collection if we deserialize a large data file. While this technique is the standard way to iterate through a data file, it's also possible to use for (GenericRecord user : dataFileReader) if performance is not a concern.

**Generate java files from avro.jar file**

java -jar /path/to/avro-tools-1.8.2.jar compile schema <schema file> <destination>

**Spring boot with Kafka**

[Apache Kafka](https://kafka.apache.org/) is distributed and fault-tolerant stream processing system.

Spring support for Kafka and the level of abstractions it provides over native Kafka Java client APIs.

Spring Kafka brings the simple and typical Spring template programming model with a *KafkaTemplate*and Message-driven POJOs via *@KafkaListener* annotation

## **Producing Messages:**

To create messages, first, we need to configure a [*ProducerFactory*](http://docs.spring.io/spring-kafka/api/org/springframework/kafka/core/ProducerFactory.html) which sets the strategy for creating Kafka [*Producer*](https://kafka.apache.org/0100/javadoc/org/apache/kafka/clients/producer/Producer.html) instances.

Then we need a [*KafkaTemplate*](http://docs.spring.io/spring-kafka/api/org/springframework/kafka/core/KafkaTemplate.html) which wraps a *Producer* instance and provides convenience methods for sending messages to Kafka topics.

*Producer* instances are thread-safe and hence using a single instance throughout an application context will give higher performance. Consequently, *KakfaTemplate* instances are also thread-safe and use of one instance is recommended.

**Spring Kafka Producer:**

|  |
| --- |
| Kafka Producer Configuration:  import java.util.HashMap;  import java.util.Map;  import org.apache.kafka.clients.producer.ProducerConfig;  import org.apache.kafka.common.serialization.StringSerializer;  import org.springframework.context.annotation.Bean;  import org.springframework.context.annotation.Configuration;  import org.springframework.kafka.core.DefaultKafkaProducerFactory;  import org.springframework.kafka.core.KafkaTemplate;  import org.springframework.kafka.core.ProducerFactory;  import org.springframework.kafka.support.serializer.JsonSerializer;  @Configuration  public class KafkaConfig {  @Bean  public ProducerFactory<String, User> producerFactory() {  Map<String, Object> configs = new HashMap<>();  configs.put(ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG, "localhost:9092");  configs.put(ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG, StringSerializer.class);  configs.put(ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG, JsonSerializer.class);  return new DefaultKafkaProducerFactory<String, User>(configs);  }    @Bean  public KafkaTemplate<String, User> kafkaTemplate(){  return new KafkaTemplate<String, User>(producerFactory());  }  }  Producer sending the message using KafkaTemplate:  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.kafka.core.KafkaTemplate;  **import** org.springframework.web.bind.annotation.GetMapping;  **import** org.springframework.web.bind.annotation.PathVariable;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RestController;  @RestController  @RequestMapping("kafka")  **public** **class** UserController {    @Autowired  **private** KafkaTemplate<String, User> kafkaTemplate;    **private** **static** **final** String ***TOPIC*** = "spring-json-topic";    @GetMapping("/send/{name}")  **public** String post(@PathVariable("name") **final** String name) {  kafkaTemplate.send(***TOPIC***, **new** User(name, "It", 60000));  **return** "message successfully sent";  }    @GetMapping("/hello")  **public** String hello() {  **return** "Hello message";  }  }  User model Class:  **public** **class** User {  **private** String name;  **private** String dept;  **private** **int** salary;    **public** User(String name, String dept, **int** salary) {  **super**();  **this**.name = name;  **this**.dept = dept;  **this**.salary = salary;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** String getDept() {  **return** dept;  }  **public** **void** setDept(String dept) {  **this**.dept = dept;  }  **public** **int** getSalary() {  **return** salary;  }  **public** **void** setSalary(**int** salary) {  **this**.salary = salary;  }  }  Running Spring boot Application:  @SpringBootApplication  **public** **class** SpringBootKafakaProducerApplication {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(SpringBootKafakaProducerApplication.**class**, args);  }  }  Dependency:  <dependency>  <groupId>org.springframework.kafka</groupId>  <artifactId>spring-kafka</artifactId>  </dependency>  Url:  http://localhost:8585/kafka/send/nagendra |

Spring Kafka Consumer:

## **Consuming Messages:**

For consuming messages, we need to configure a [ConsumerFactory](http://docs.spring.io/autorepo/docs/spring-kafka-dist/1.1.3.RELEASE/api/org/springframework/kafka/core/ConsumerFactory.html) and a [*KafkaListenerContainerFactory*](http://docs.spring.io/autorepo/docs/spring-kafka-dist/1.1.3.RELEASE/api/org/springframework/kafka/config/KafkaListenerContainerFactory.html). Once these beans are available in spring bean factory, POJO based consumers can be configured using [*@KafkaListener*](http://docs.spring.io/autorepo/docs/spring-kafka-dist/1.1.3.RELEASE/api/org/springframework/kafka/annotation/KafkaListener.html) annotation.

[*@EnableKafka*](http://docs.spring.io/autorepo/docs/spring-kafka-dist/1.1.3.RELEASE/api/org/springframework/kafka/annotation/EnableKafka.html) annotation is required on the configuration class to enable detection of *@KafkaListener* annotation on spring managed beans.

|  |
| --- |
| Kafka Consumer Configuraton:  **import** org.apache.kafka.clients.consumer.ConsumerConfig;  **import** org.apache.kafka.common.serialization.StringDeserializer;  **import** org.springframework.context.annotation.Bean;  **import** org.springframework.context.annotation.Configuration;  **import** org.springframework.kafka.annotation.EnableKafka;  **import** org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;  **import** org.springframework.kafka.core.ConsumerFactory;  **import** org.springframework.kafka.core.DefaultKafkaConsumerFactory;  **import** org.springframework.kafka.support.serializer.JsonDeserializer;  **import** com.fasterxml.jackson.databind.annotation.JsonDeserialize;  @Configuration  @EnableKafka  **public** **class** KafkaConfig {    // string object consumer configuration  @Bean  **public** ConsumerFactory<String, String> consumerFactory() {  Map<String, Object> configs = **new** HashMap<>();  configs.put(ConsumerConfig.***BOOTSTRAP\_SERVERS\_CONFIG***, "localhost:9092");  configs.put(ConsumerConfig.***GROUP\_ID\_CONFIG***, "group\_id");  configs.put(ConsumerConfig.***KEY\_DESERIALIZER\_CLASS\_CONFIG***, StringDeserializer.**class**);  configs.put(ConsumerConfig.***VALUE\_DESERIALIZER\_CLASS\_CONFIG***, StringDeserializer.**class**);  **return** **new** DefaultKafkaConsumerFactory<String, String>(configs);  }  @Bean  **public** ConcurrentKafkaListenerContainerFactory<String, String> kafkaListenerContainerFactory() {  ConcurrentKafkaListenerContainerFactory<String, String> factory = **new** ConcurrentKafkaListenerContainerFactory<String, String>();  factory.setConsumerFactory(consumerFactory());  **return** factory;  }    // user object consumer configuration  @Bean  **public** ConsumerFactory<String, User> userConsumerFactory() {  Map<String, Object> configs = **new** HashMap<>();  configs.put(ConsumerConfig.***BOOTSTRAP\_SERVERS\_CONFIG***, "localhost:9092");  configs.put(ConsumerConfig.***GROUP\_ID\_CONFIG***, "group\_json");  configs.put(ConsumerConfig.***KEY\_DESERIALIZER\_CLASS\_CONFIG***, StringDeserializer.**class**);  configs.put(ConsumerConfig.***VALUE\_DESERIALIZER\_CLASS\_CONFIG***, JsonDeserialize.**class**);  **return** **new** DefaultKafkaConsumerFactory<>(configs, **new** StringDeserializer(), **new** JsonDeserializer<>(User.**class**));  }    @Bean  **public** ConcurrentKafkaListenerContainerFactory<String, User> userkafkaListenerContainerFactory() {  ConcurrentKafkaListenerContainerFactory<String, User> factory = **new** ConcurrentKafkaListenerContainerFactory<String, User>();  factory.setConsumerFactory(userConsumerFactory());  **return** factory;  }  }  Kafka Consumer listener using KafkaListener:  **import** org.springframework.kafka.annotation.KafkaListener;  **import** org.springframework.stereotype.Service;  @Service  **public** **class** KafkaConsumer {    @KafkaListener(topics="spring-topic", group="group\_json")  **public** **void** consume(String msg) {  System.***out***.println("Consumed message: "+msg);  }  @KafkaListener(topics="spring-json-topic", group="group\_id", containerFactory="userkafkaListenerContainerFactory")  **public** **void** consumeJsonMsg(User user) {  System.***out***.println("Consum User json Object: "+user);  }  }  User model Object with Default Constructor:  Dependency:  <dependency>  <groupId>org.springframework.kafka</groupId>  <artifactId>spring-kafka</artifactId>  </dependency>      <dependency>  <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-core</artifactId>  </dependency>  Running spring boot application:  Output Console:  Consum User json Object: User [name=nagendra, dept=It, salary=60000] |

**Spring Kafka Async and Sync:**

Send methods return a ListenableFuture<SendResult>. You can register a callback with the listener to receive the result of the send asynchronously.

**Non Blocking(Async):**

|  |
| --- |
| ListenableFuture<SendResult<Integer, String>> future = template.send("foo");  future.addCallback(new ListenableFutureCallback<SendResult<Integer, String>>() {  @Override  public void onSuccess(SendResult<Integer, String> result) {  ...  }  @Override  public void onFailure(Throwable ex) {  ...  }  }); |

Examples:

|  |
| --- |
| @RestController  @RequestMapping("kafka")  **public** **class** UserController {    @Autowired  **private** KafkaTemplate<String, User> kafkaTemplate;    **private** **static** **final** String ***TOPIC*** = "spring-json-topic";    @GetMapping("/send/{name}")  **public** String post(@PathVariable("name") **final** String name) {    **for** (**int** i = 0; i < 2000; i++) {  ListenableFuture<SendResult<String, User>> ac = kafkaTemplate.send(***TOPIC***, **new** User(name+i, "It", 60000));  ac.addCallback(**new** AsyncProducerHandler());  }    **return** "message successfully sent";  }    @GetMapping("/hello")  **public** String hello() {  **return** "Hello message";  }  }  CallBack Method:  **public** **class** AsyncProducerHandler **implements** ListenableFutureCallback<SendResult<String, User>>{  @Override  **public** **void** onSuccess(SendResult<String, User> result) {    // ProducerRecord details:  System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Producer Record Deatials\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  ProducerRecord<String, User> record = result.getProducerRecord();  System.***out***.println("topic: "+ record.topic());  System.***out***.println("partion: "+ record.partition());  System.***out***.println("TimeStamp: "+ record.timestamp());  System.***out***.println("Key: "+ record.key());  System.***out***.println("Value: "+ record.value());    System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RecordMetadata Deatials\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  //RecordMetadata deails:  RecordMetadata metaData = result.getRecordMetadata();  System.***out***.println("metaData partion: "+ metaData.partition());  System.***out***.println("metaData topic: "+ metaData.topic());  System.***out***.println("metaData timestamp: "+ metaData.timestamp());  System.***out***.println("metaData offset: "+ metaData.offset());  System.***out***.println("metaData checksum: "+ metaData.checksum());  System.***out***.println("metaData toString: "+ metaData.toString());    }  @Override  **public** **void** onFailure(Throwable ex) {  System.***out***.println("Exception onFailue Method: "+ex);  }  } |

**Blocking (Sync):**

|  |
| --- |
| public void sendToKafka(final MyOutputData data) {  final ProducerRecord<String, String> record = createRecord(data);  try {  template.send(record).get(10, TimeUnit.SECONDS);  handleSuccess(data);  }  catch (ExecutionException e) {  handleFailure(data, record, e.getCause());  }  catch (TimeoutException | InterruptedException e) {  handleFailure(data, record, e);  }  }  Get Method Details:  SendResult<String, User> res1 = kafkaTemplate.send(***TOPIC***, **new** User(name, "It", 60000)).get();  SendResult<String, User> res2 = kafkaTemplate.send(***TOPIC***, **new** User(name, "It", 60000)).get(102L,TimeUnit.***SECONDS***); |

## **Forwarding Listener Results using @SendTo :**

Starting from version 2.0, if you also annotate a @KafkaListener with a @SendTo annotation and the method invocation returns a result, the result will be forwarded to the topic specified by the @SendTo annotation.

The @SendTo value argument can accept several forms:

* @SendTo("someTopic") specifies a static topic to rout to.
* @SendTo("#{someExpression}") specifies an application initialization expression. This expression is evaluated once during application context initialization and’ll be forwarded to the result.
* @SendTo("!{someExpression}") specifies a runtime expression. This expression is evaluated at runtime. The #root object for the evaluation has 3 properties:
* request – the inbound ConsumerRecord (or ConsumerRecords object for a batch listener)
* source – the Message<?> converted from the request.
* result – the method return result.
* @SendTo() (no properties) is treaded as !{source.headers["kafka\_replyTopic']} (since version 2.1.3).

https://blog.imaginea.com/how-to-rebalance-topics-in-kafka-cluster/

https://www.codenotfound.com/spring-kafka-consumer-producer-example.html

http://blog.empeccableweb.com/wp/2016/11/30/kafka-producer-and-consumer-example/

http://bigdatums.net/2016/10/16/writing-to-a-database-from-spark/

https://jack-vanlightly.com/blog/2017/12/4/rabbitmq-vs-kafka-part-1-messaging-topologies