**What is Kafka:**

* Apache Kafka is a highly scalable and distributed platform for creating and processing extremes in real time.

We can use Kafka to

* create one or more real time streams of data.
* process these streams and produce results in real time.
* Real Time Stream processing: Processing mechanism where application does not accumulate data and then begin processing. It continuously listens to the data and processes it as soon as the data arrives.
* Kafka has adopted Pub/Sub messaging system architecture, and it works as an enterprise messaging system.

A typical messaging system has got three components. Producer, Message broker And Consumer.

* + Producer: Client application that sends data records (Messages)
  + Broker: responsible for receiving messages from the producers and storing them into local storage. Broker is in the center and acts as a middleman between producers and consumers.
  + Consumer: Client applications that read messages from the broker and process them.

Kafka works as a Pub/Sub messaging system, where we create producer applications to send data as a stream. We install and configure Kafka server to act as a message broker. And finally, we create consumer applications to process the data stream in real-time.

* Kafka was initially conceptualized and developed by LinkedIn and later open sourced in 2011. In LinkedIn Kafka Started as a Data integration solution. It stood between multiple applications and facilitated data integration between various services.
* From Integration Solution to a Streaming Platform:

Kafka initially started with two things.

* Server software that you can install and configure to work as a message broker.
* A Java-based client API library to help with the following.
  + - Create Kafka producer applications and
    - Create Kafka consumer applications.

Later it aspired to become a full-fledged real time streaming platform. With this in view 3 more components were added:

* Kafka Connect (Open Source and available with Apache 2.0 license)
* Kafka Streams (Open Source and available with Apache 2.0 license)
* KSQL (licensing restrictions and offered by Confluent Inc as a commercial tool)
* From 2011 to 2019 Kafka evolved as a set of five components.
* Kafka broker - Central server system
* Kafka Client API - Producer and Consumer APIs.
* Kafka Connect - Addresses the initial data integration problem for which Kafka was initially designed.
* Kafka streams: Library for creating real time Stream processing applications.
* KSQL - Aim to become a real time database and capture some market sharing Databases and DW/BI space.
* Kafka in Enterprise application ecosystem:

By adopting Pub/Sub semantics, Apache Kafka becomes the circulatory system of your data ecosystem. Kafka brings data to various members of the infrastructure. Kafka occupies a central place in your real time data integration infrastructure.

The producers and consumers are completely decoupled, and they do not need tight coupling or direct connections.

They always interact with the Kafka broker using a consistent interface.

Producers do not need to be concerned about who is using the data, and they just send the data once without worrying about how many consumers would be reading it. Producers and consumers can be added, removed, and modified as the business case evolves.

**Apache Kafka Core Components**

* **Producer**: An Application that sends Data/Message/Message record.

A Message is nothing but a small to medium sized piece of data.   
The message record may have a different meaning and schema or record structure for us. But for Kafka, it is a simple array of bytes.

We can create a new producer or use out of the box ready to use producer which fits our purpose.

* **Consumer**: An application that receives data. Producers don’t send data directly to the recipients. They just send it to the Kafka server. Anyone interested in that data should come forward and consume the data from the server. An application that requests data from the Kafka server is a consumer.
* **Broker:** The broker is the Kafka server. Kafka act as a message broker between producer and consumer. The producer and consumers do not interact directly. They use the Kafka server as an agent or a broker to exchange messages.
* **Cluster**: As a general definition a cluster is a group of computers acting together for a common purpose. Kafka cluster is a group of computers, each running one instance of the Kafka broker.
* **Topic:** A topic is an arbitrary name given to a data set. We can say it's a unique name for a data stream. (We can also think of it as a database table). Creating a topic is a design time decision.

When designing our application, an architect is responsible for creating one or more topics. Producers and the consumers send and receive data by the topic.

* **Topic Partition:** A Partition is a small and independent portion of the topic. Kafka is a distributed system, and it runs on a cluster of computers. Thus, Kafka breaks the topic into smaller partitions and stores those partitions on different machines.

The number of partitions in our topic is a design decision. An architect decides the number of partitions for each topic. When we create a topic, we need to specify the number of partitions that we need, and the Kafka will produce it. The partition is the smallest unit and it is going to be sitting on a single machine. We cannot break it further.  
Topic partitions are not only a solution to increase their storage capacity but also a method to distribute the workload. Kafka topic partitions are the core idea of making Kafka are distributed and a scalable system.

* **Partition Offset:** A unique immutable sequence ID of a message in the partition. The sequence ID is automatically assigned by the broker to every message record as it arrives in the partition. Remember these offsets are local within the partitions. There is no global ordering in the topic across partitions. To locate a specific message, you must need 3 things: Topic name, Partition number and then the offset number.
* **Consumer group:** Group of consumers to share the workload.

Normally we start multiple copies of the consumer application in the same group and let them divide the workload.

Topic partition is a tool for scalability. The maximum possible parallel consumers are limited by the number of partitions in that topic.

Kafka doesn't allow more than one consumer to read and process data from these same partitions simultaneously. This restriction is necessary to avoid the double reading of records.

**Apache Connect Core Components**

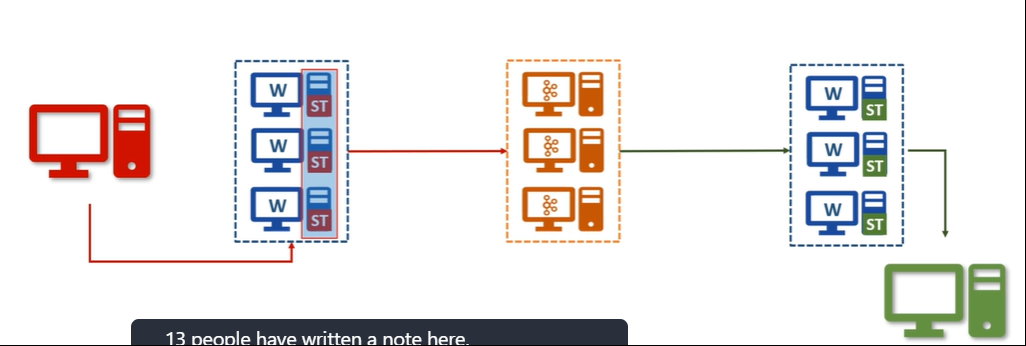
* Kafka Connect is a component of Kafka for connecting and moving data between Kafka and external systems. We have two types of Kafka Connectors. Source Connector and Sink connector. And together, they support a bunch of systems and offer you an out of the box data integration capability without writing a single line of code.
* There are two ways to create a producer depending upon the source code availability of the source system.
  + If we have the source code of the source application and it is practically feasible to modify our source application, we can create an embedded Kafka producer using Kafka producer APIs. The embedded Kafka producer becomes part of our source application, it runs inside the application and sends invoices to the Kafka Cluster.
  + Independent Kafka producer for reading and writing. On one side, it connects to the source application database, reads the data, and sends it to Kafka Cluster on the other side. However, this solution is already available to us in form of Kafka Connect.
* **Kafka Connect** is a system which you can place in between our data source and Kafka Cluster.

All we do is to configure it to consume data from a source system and send it to the Kafka Cluster. We do not need to write a single line of code. Everything is already done and made available to us. We can also place Kafka connect on the target side. It can read data from the cluster and store in our target system.

* **Kafka Source Connector:** Pull data from a source system and send it to the Kafka Cluster. The Source Connector will internally use the Kafka producer API.
* **Kafka Sink connector** to consume the data from Kafka topic and Sink it to an external system.

These Sink connectors will internally use the Kafka Consumer API

* **Kafka Connector Framework:**
  + It’s an open source framework for implementing Kafka connectors. The Kafka connect framework allows us to write connectors.
  + These connectors are implemented in two flavors - Source connector and Sink connector.
  + The Kafka connect framework takes care of all the heavy lifting, scalability, fault tolerance, error handling, and bunch of other things.
  + As a connector developer, all we need to do is to implement two Java classes.   
    The first one is SourceConnector or SinkConnector class. And the second one is the SourceTask or the SinkTask.
  + Once our Connector is developed and tested, we can package it as an Uber Jar or as a Zip archive. and share it with others
  + Ex: Say we want to bring some data from an RDBMS to a Kafka Cluster. All we need to do is to take an appropriate source connector, for example a JDBC source connector. Then we install it in your Kafka connect, configure it and run it. That's all. The JDBC Connector will take care of the rest.
* **Kafka Connect Scalability:**
  + We know how to scale core Kafka components:
    - We can scale producers by adding more producers to send data in parallel.
    - We scale the Cluster by simply adding more brokers. We also partition the Kafka topic.
    - We scale the consumers by adding more consumers in the consumer group.
  + Scaling Kafka Connect is easy as The Kafka Connect itself is a Cluster. Each individual unit in the Connect Cluster is called a Connect Worker. We can think of it as a group of computers, each running one Kafka Connect Worker. We can play with the number of tasks and scale the Cluster capacity by adding more workers.



We can have one Kafka connect Cluster and run as many connectors as you want.

* + If our Cluster is fully utilized, we can scale it by adding more workers to the same Cluster. And we can do it dynamically without stopping any existing connectors.
* **Single Message transformations (SMTs):** Kafka connect also allowed some fundamental Single Message Transformations (SMTs). We can apply some transformations or changes to each message on the fly. And this is allowed with both source and Sink connectors.

Following is list of some SMTs:  
Sin e Messa Transformations — SMTS 
Add a new field in your record using static data 
Filter or Rename Fields 
Mask some fields with a Null Value. 
Change the Record Key 
Route the record to a different Kafka 

We can chain multiple SMTs and play with it to restructure your records and route them to a different topic. However, these SMTs are not good enough to perform some real-life data validations and transformations.

* **Kafka Connect architecture:**
* To understand Kafka Connect architecture we need to understand **3 things – Worker, Connector and Task**
* A Kafka Connect is a Cluster and it runs one or more workers. These workers are fault tolerant, and they use the Group ID to form a Cluster. This Group ID mechanism is the same as Kafka Consumer Groups. So, all you need to do is to start workers with the same group id, and they will join hands to form a Kafka Connect Cluster.  
  These workers are the main workhorse of the Kafka Connect.   
  These workers are fault-tolerant and self-managed.
  + If a worker processes stops or crashes other workers in the Connect Cluster will recognize that and reassign the connectors and tasks that ran on that worker to the remaining workers.
  + If a new worker joins a connect Cluster, other workers will notice that and assign connectors or tasks to it and make sure the load is balanced. So, in a nutshell, these workers will give you reliability, higher availability, scalability and load balancing.
* Say we want to copy data from relational database. We shall **download the JDBC Source Connector, install it within the Cluster**. (making sure the JAR files and all its dependencies are made available to these workers). Next, we **configure the connector**. Configuration means providing some necessary information.

For example, database connection details, a list of tables to copy, frequency to poll the source for the new data, the maximum number of tasks, and many other things depending upon your Connector.

* All this configuration goes into a file, and we will start the connector using some command line tool.

Kafka Connect also offers you REST APIs, so you can even begin the connector using the REST API instead of the command line tool. At this stage, one of the workers will start your Connector process. Connectors determine the degree of parallelism. That means how many parallel tasks can it start to copy the data.

* Connector does not copy the data. It is only responsible for defining and creating a task list. The Connector will also include some additional configurations such as database connection details and other things to make sure that the task can operate as an independent process. Finally, the list of tasks will be given to these workers, and they will start the task. **So your task is distributed across the workers for balancing the Cluster load.**
* Now the task is responsible for connecting to the source system, polling the data at a regular interval, collecting the records, and handing over it to the worker.

That task is only responsible for interacting with the external system. This source task will hand over the data to the worker, and the worker is responsible for sending it to the Kafka. In the case of the Sink task, they get the Kafka record from the worker, and the task is only responsible for inserting the record into the target system.

* The Connector class determines how to split the input for parallel processing. Interaction wit External system is taken care by Task Class. These are the only things developer needs to take care.  Most of the other stuff like interacting with Kafka, handling configurations, errors, monitoring connectors, and tasks, scaling up and down, and handling failures are standard things and are taken care of by the Kafka Connect Framework.

**Apache Streams Core Components**

* Kafka Broker, Kafka Client API, Kafka Connect help us create a simplified and manageable data integration solution.

Kafka Streams and KSQL will take our Kafka implementation beyond the data integration and allow us to create a scalable and fault tolerant real time stream processing application.

* Data Streams and Stream processing: Data streams are an unbounded, infinite and ever-growing sequence of data that is continuously generated and sent in small sizes in order of KBs. In Stream processing, we ask the question once and the system should give us the most recent version of the answer all the time. So, stream processing is a continuous process and the business reports are updated continuously based on the data available till time. Kafka Streams is a tool specifically designed for stream processing.

Kafka producer, consumer and Kafka connect are good tools for data integration. However they are not enough for Stream processing. For Stream processing, we need specialized tool like Kafka Streams.

* At the most basic level, Kafka Streams is a library for building applications and microservices where the input data are streamed in a Kafka topic. So we cannot use Kafka streams if data is not coming to a Kafka topic. The starting point for the Kafka Stream is one or more Kafka topics.
* The most powerful feature of Kafka streams is being a simple library. So, you can create a standard Java and Scala applications to perform real time stream processing. And you can deploy your applications to any machine, virtual machine, container, or on a Kubernetes cluster. (No cluster needed)

Your application is just another typical application with inherent parallel processing capability, fault tolerance and scalability which is given to you by the Kafka streams library as an out of the box capability.

* Kafka streams libraries is specifically designed for the sole purpose of stream processing. So, it allows us to handle unique streaming challenges as shown below:

What Kafka Stream Offers? 
Working with streams/tables and interoperating 
Grouping and Continuously updating Aggregates 
2. 
Join streams, tables, and a combination of both 
3. 
Create and manage fault-tolerant, efficient local state stores 
4. 
Flexible Windowing capability 
5. 
Flexible Time Schematics - Event time, Processing time, 
6. 
Latecomers, High watermark, Exactly-once processing, etc 
7. 
Interactive Query - Serving other microservices 
Unit testing tools 
8. 
Easy to use DLS and extensibility to create custom processors 
9. 
10. Inherent fault tolerance and dynamic scalability 
11. Deploy in containers and manage using Kubernetes 

* Kafka streams is all about continuously reading a stream of data from one or more Kafka topics and then, we develop your application logic to process those streams in real-time and take necessary actions.
* Say we deployed our Kafka Streams application on a single machine, and input topic has 3 partitions. Kafka streams will internally create three logical tasks because the maximum number of partitions across the input topic is 3. So the Kafka stream's framework knows that we can create three consumers where each could be consuming from one partition in parallel. We don’t have to code this thing. The framework is smartly detecting it and creates three logical tasks and assigns partitions to these tasks.

**Kafka SQL Core Concepts**

* KSQL is an SQL interface to the Kafka Streams. most of the things which we can do using Kafka Streams are available to you in KSQL.

It means we can create scalable and fault-tolerant stream processing workloads without the need to write code in a programming language such as Java or Scala.

* KSQL has got two operating modes:
  + Interactive Mode and
  + Headless Mode.

Interactive mode is using a command-line interface (CLI) or a web-based UI to submit KSQL and get an immediate response.

The CLI works like any database SQL interface would work. The Headless mode is a non-interactive mode that allows you to submit your KSQL files, which are executed by the KSQL server. The headless mode is ideal for the production environment, whereas the CLI mode is ideal for the development environment.

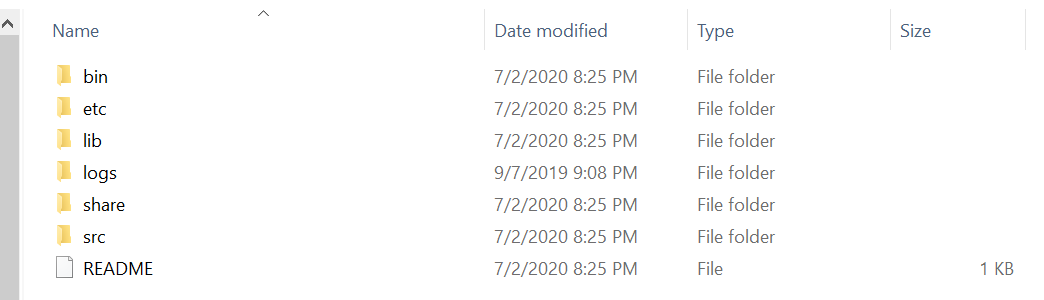
* KSQL Architecture:
* The KSQL comes with three components.
  + KSQL engine,
  + REST interface and
  + KSQL CLI/UI.
* The KSQL engine and the REST interface together form the KSQL server. The KSQL server can be deployed in one of the available modes. Interactive mode and Headless Mode. We can also deploy multiple KSQL servers to form a scalable KSQL cluster. However, all servers that run in a cluster must use the same deployment mode.
* The KSQL engine is the core component which is responsible for KSQL statement and queries. Under the hood, the engine is going to parse your KSQL statements, build corresponding Kafka streams topology, and run them as streams tasks. And these are streams tasks are executed on the available KSQL servers in the cluster.
* We can dynamically add more servers in the cluster to scale out the resources, and fault Tolerance is an inherent feature of the Kafka streams.
* KSQL cluster is separate from your Kafka cluster, and your KSQL Server will internally communicate to the Kafka cluster for reading inputs and writing outputs.
* The REST interface is to power the KSQL clients. So, the KSQL CLient will send the commands to the REST Interface, which will internally communicate with the KSQL Engine to execute your KSQL Commands.
* KSQL allows you to use your Kafka topic as a table and fire SQL like queries over those topics. We can use group by and aggregates on your Kafka topics. We can group an aggregate over time window. We can apply filters. We can join two topics. We can sink the result of your query into another topic.
* KSQL for Kafka is one big step forward for Kafka to become a real time data warehouse.

**When to use what?**

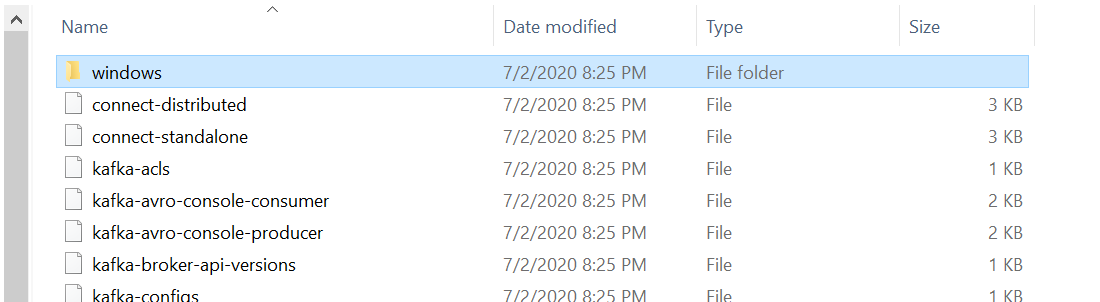
* 3 patterns in which Kafka is normally used:
* Data integration pattern - using a combination of the first three Kafka components. Kafka broker, Kafka Client API and Kafka connect.
* Real-time stream processing using micro service architecture
* Real-time data warehousing pattern.

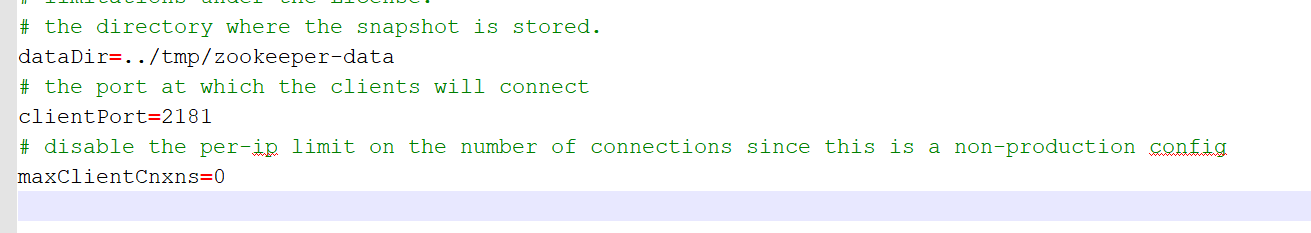
**Installing Single node Kafka**

* Kafka comes in 3 flavors and we can classify into 3 categories:  
  + **Open Source Version of Kafka** – Download from Apache website. We need to install and manage it ourselves. Not used normally in production applications.
  + **Commercial Distributions** – these comes with a bunch of tools and utilities to manage our day to day operations and monitor our cluster. This option comes with a cost to your organization. Ex: Confluent. Confluent also offers a community edition without any cost.
  + **Fully managed Kafka service in Cloud** – We do not need to download, install, run operate or maintain anything related to the Kafka cluster. Just use the cluster for producing and consuming data. All the infrastructure headache is taken care of by the managed service provider. This option is the simplest way of using Kafka. Ex: Confluent, Amazon, Aiven.io
* Install Confluent edition of Kafka:
  + Download Community Edition of Confluent Kafka.
  + Un-compress the downloaded file. Our download is a pre-configured single load Kafka cluster.
  + Since Kafka is a JVM based application, we need to make sure that we have got Java installation on your machine. (java -version to check for java installation)



* Kafka comes with a bunch of command-line tools, and we can find them in our bin directory. These contain shell scripts. However we can also find a compatible windows batch files in the windows directory.



* Starting a Kafka cluster is a 2-step process:
  + **Start Zookeeper** – using **zookeeper-server-start.bat**. This script also takes a mandatory argument ( Zookeeper Config file). File Name is zookeeper.properties and is found in etc\kafka\ directory.  
      
    zookeeper.properties:  
      
    





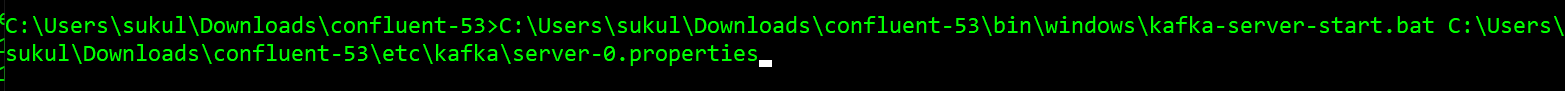
Zookeeper is a kind of database where Kafka brokers would store shared information.

It is used as a shared system among multiple Kafka brokers to coordinate among themselves for various things.

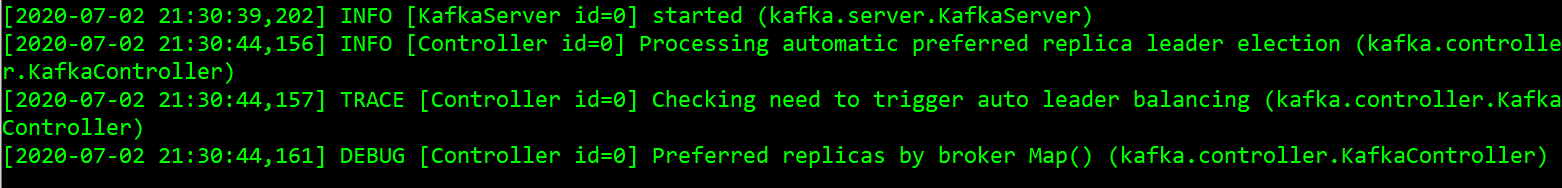
Kafka community has already announced that they are going to retire the zookeeper in the coming days.

We must have it running even if we have got a single broker.

* + **Start Broker** – Minimize the Zookeeper command prompt, start a new window and start the Kafka server using the script kafka-server-start.bat . This also needs a configuration file server.properties which is found under etc\kafka directory.



This should start a Kafka broker and we will get an Id for the broker. Ex: 0



**Using Command Line Producer and Consumer**

* Kafka Comes with out of the both command line producer and consumer

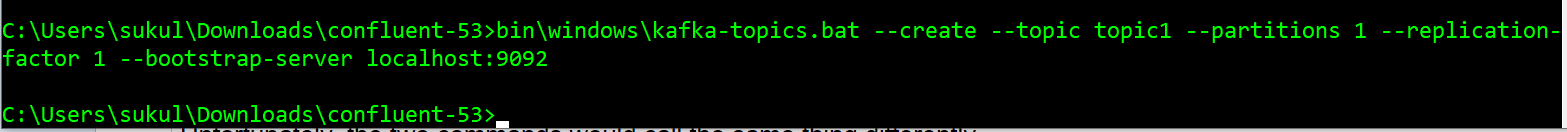
We can use the command line Kafka producer to send files to Kafka broker.

* We first start with creating a topic. While creating a topic we need to specify following things:
* **No of partitions:** We have to define number of partitions in each topic. Two considerations that impact the number of partitions include – Storage requirement and Parallel processing requirement.   
  In this example we don’t have any requirement to read in parallel. So we use only one partition.
* **Replication Factor:** The replication-factor is the number of copies of each partition. Higher replication factor provides fault tolerance. The copies are stored at different brokers. And if one broker is down, then we will have another broker with a copy of the partition.

So, your consumers can still read the data from the cluster even if one or two brokers are down.

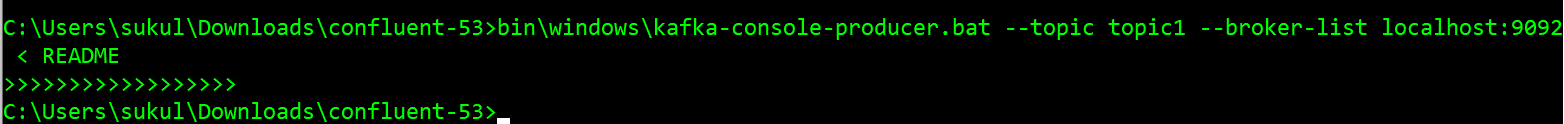
In this example we have a single broker. So we keep the replication factor as 1.

* **Cluster Coordinates:** We need to provide the cluster coordinates when creating a topic. The property takes cluster coordinates as bootstrap server – Ip/hostname and listener port of the broker (Default port no is 9092)

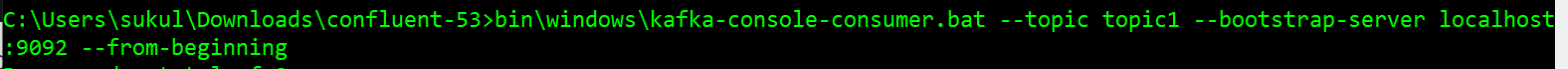


* Next we create a console producer and send a file. Here also we need to provide the name of the topic we send data to, and provide cluster coordinates using the –broker-list parameter.

Note that Kafka console producer can take input from a redirected file.



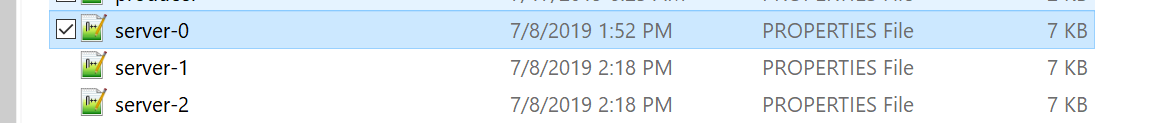
* Next we make use of console producer to read the file from Kafka broker. Here as well we specific the topic name and the broker-list.  
  Also note that we specify the option –from-beginning to indicate we need to read all data the broker has from the beginning.



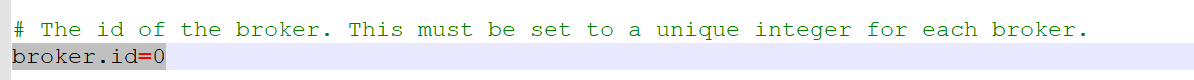
* Here we only had a single node cluster, but it could have been a 50 node cluster. Data files came from the same computer, but that also could have been a remove server connected over tcpip. Since we only had to send files we could use console producer. It’s good at sending data files. Here consumer was also on the same machine, but it could very well be on a different computer reading data over tcp/ip network.

**Installing a Multi-Node Kafka Cluster**

* As we already know the server.properties file is used to start the kafka cluster. This file is found under etc/kafka if we are using kafka confluent or under config directory if we are using open source kafka. The Kafka-server-start command is going to read broker configurations from this file.
* To start 3 brokers on the same machine we do following preparations:
  + Create 3 copies of the server.properties files as below. Each of these files shall be used for 3 different brokers.

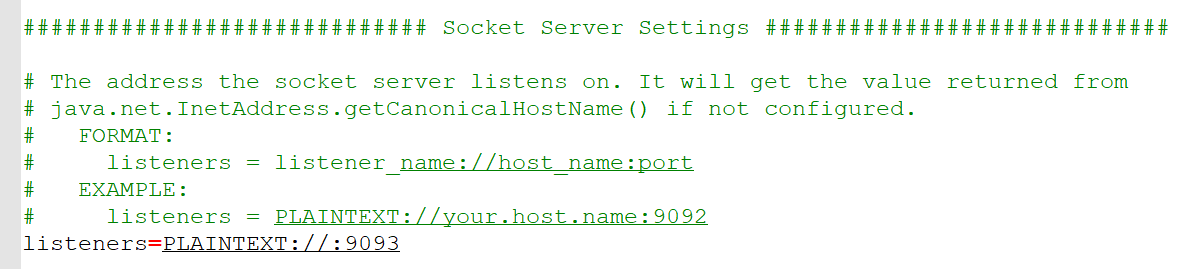


* + Each Broker should have a separate broker id. So we update the broker.id properties in each of these server properties files and assign values 0,1,2 respectively.



* + Next property that we need to change is the broker port number. By default it gets the port number 9092. This is the port number at which the producers are going to send data to the Kafka broker. Similarly, consumers are also going to request the data using this port. Kafka broker listens to this port for produce and consumes requests coming from the producer and the consumer. Since we are going to run three brokers on this same machine, we should assign 3 different ports for all three brokers.

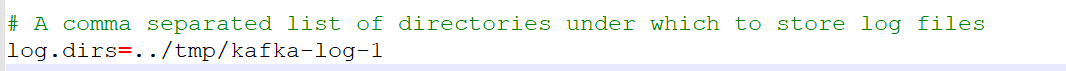
For the first broker, we will leave the default value 9092. But for the second broker, we replace it with 9093, similarly for the third broker we change it to 9094.



Making sure each broker is listening on a different port shall avoid the “Port already in use error”.

Note: In prod situations we shall launch one broker on one machine. In that case we don’t need to change the port number. But running them on a single device would need to assign different ports

* + The 3rd Configuration we need to modify is the Kafka Log directory location. This is the directory location where Kafka is going to store the partition data. When running multiple brokers on the same machine, we should also assign a different directory to each broker.



For other brokers the value shall be ../tmp/kafka-log-0 and ../tmp/kafka-log-2

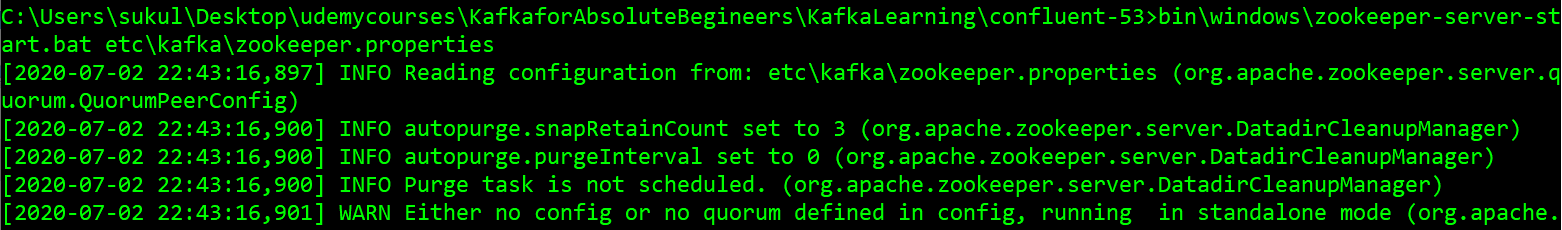
Imp: When running one broker per machine, we don’t need to change anything except the broker id. However we can avoid that as well if we configure kafka to auto assign an identifier to each broker.

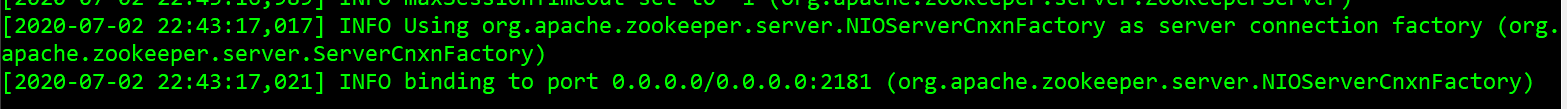
* Next we start the zookeeper and 3 brokers as follows:

**Zookeeper:**

cd C:\Users\sukul\Desktop\udemycourses\KafkaforAbsoluteBegineers\KafkaLearning\confluent-53

bin\windows\kafka-server-start.bat etc\kafka\server-0.properties

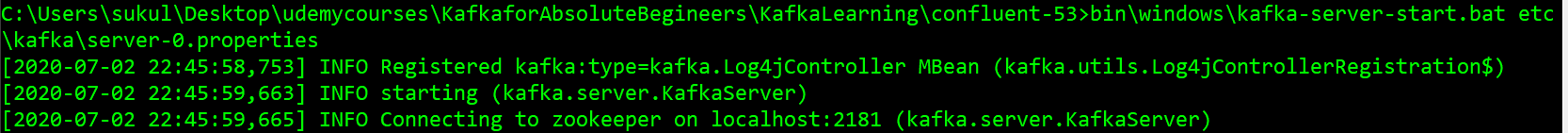




**Server 0:**

cd C:\Users\sukul\Desktop\udemycourses\KafkaforAbsoluteBegineers\KafkaLearning\confluent-53

bin\windows\kafka-server-start.bat etc\kafka\server-0.properties



**Server 1:**

cd C:\Users\sukul\Desktop\udemycourses\KafkaforAbsoluteBegineers\KafkaLearning\confluent-53

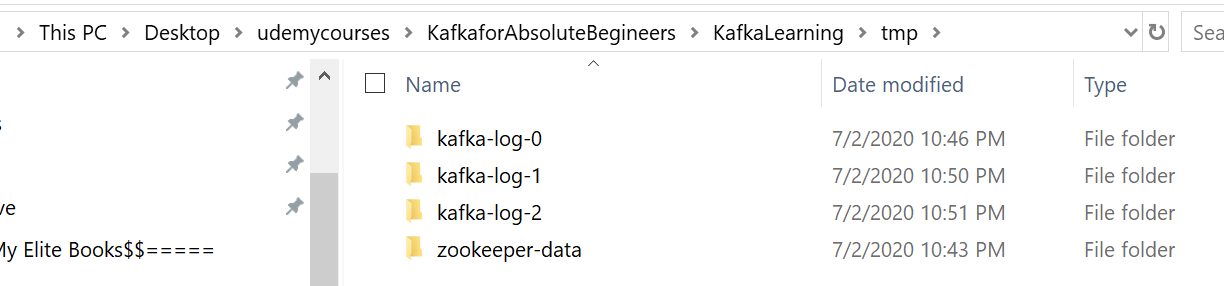
bin\windows\kafka-server-start.bat etc\kafka\server-1.properties

**Server 2:**

cd C:\Users\sukul\Desktop\udemycourses\KafkaforAbsoluteBegineers\KafkaLearning\confluent-53

bin\windows\kafka-server-start.bat etc\kafka\server-2.properties

* Note that Zookeeper-data and 3 log directories were created

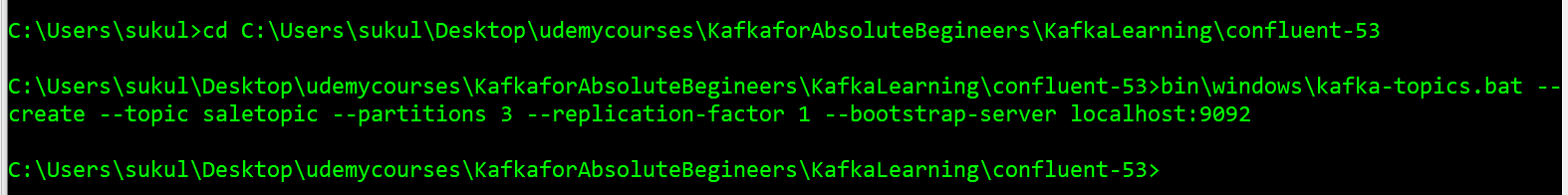


**Kafka Consumer Groups**

* Following are the steps followed in this Demo:
  + Create a Topic and make sure it has 3 partitions.
  + Start 2 consumers in a group. They shall read data from same topic. Since they are running in same group, they will share the workload automatically.
  + Start Kafka producer and send the data file to cluster.
  + Observe the outcome.
* Creating Topic with 3 partitions

cd C:\Users\sukul\Desktop\udemycourses\KafkaforAbsoluteBegineers\KafkaLearning\confluent-53

bin\windows\kafka-topics.bat --create --topic saletopic --partitions 3 --replication-factor 1 --bootstrap-server localhost:9092

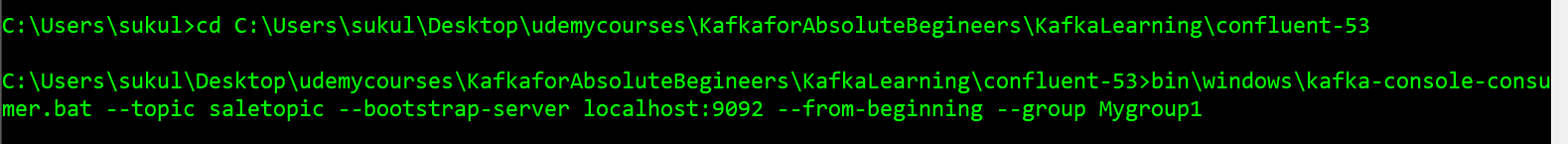


* Create 2 console consumer in a same group.

Note that in below command, we don’t need to provide all broker coordinates. Just one broker is enough.

From 1 cmd window: bin\windows\kafka-console-consumer.bat --topic saletopic --bootstrap-server localhost:9092 --from-beginning --group Mygroup1

From 2nd cmd window: bin\windows\kafka-console-consumer.bat --topic saletopic --bootstrap-server localhost:9092 --from-beginning --group Mygroup1



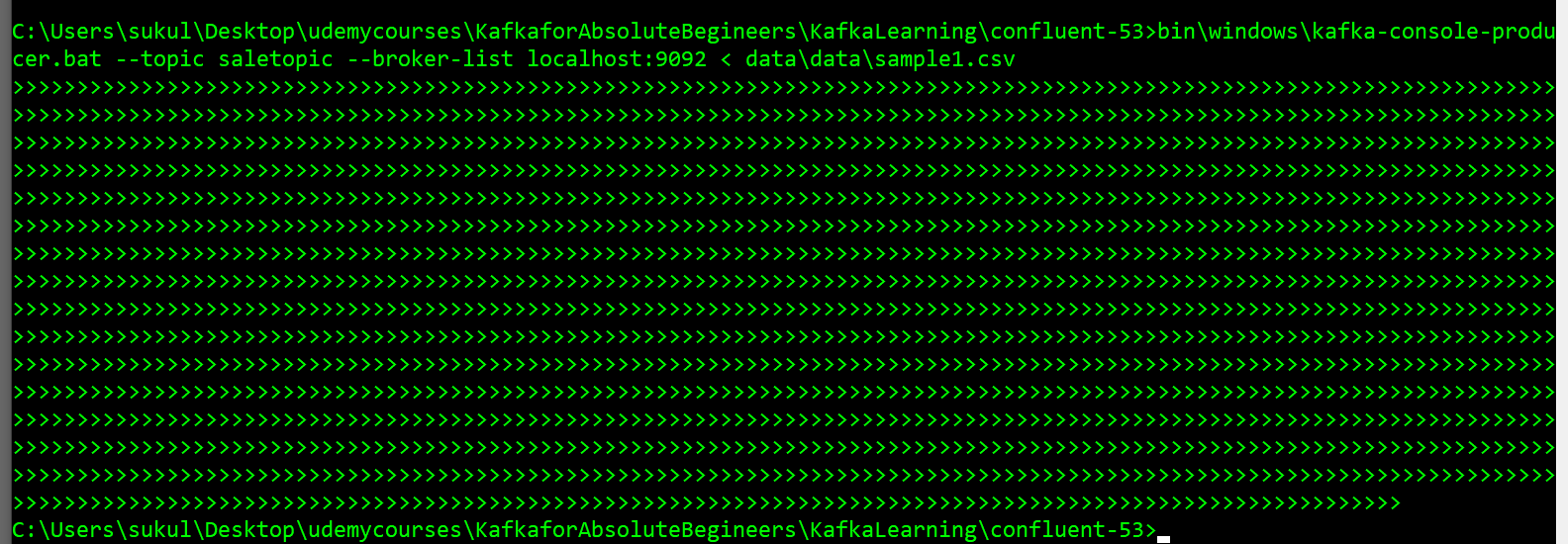
* Start the producer to send data.

Since the topic is partitioned all the data will be distributed among the three partitions.

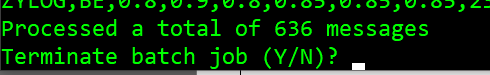
Some records will come to the first broker in the first partition.

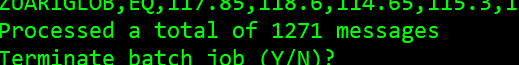
Some of them will go to the other two brokers, and hence they will land in the other two partitions. Since we have three partitions but two consumers, one of these consumers is going to read data from two partitions. And the other one is going to read the data from the other remaining partition. If we had a single consumer, it would read data from all the partitions.

bin\windows\kafka-console-producer.bat --topic saletopic --broker-list localhost:9092 < data\data\sample1.csv



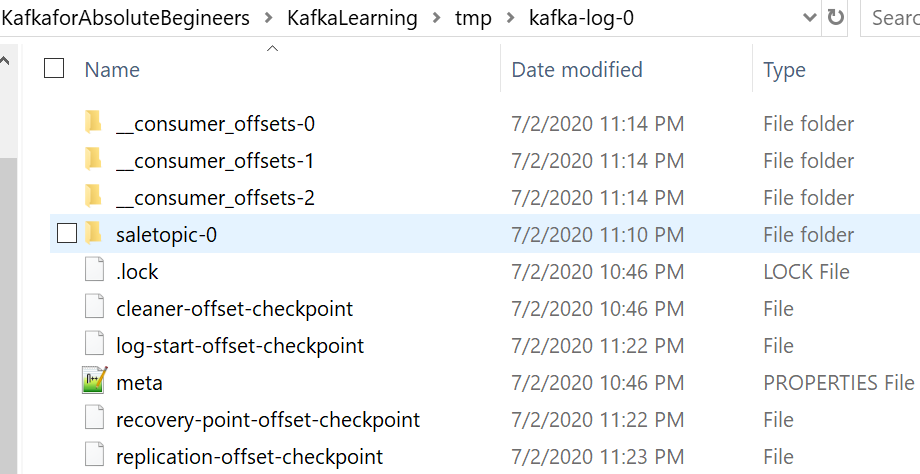
* Press Cntl+C on the consumer windows to see the number of records processed by each consumer.





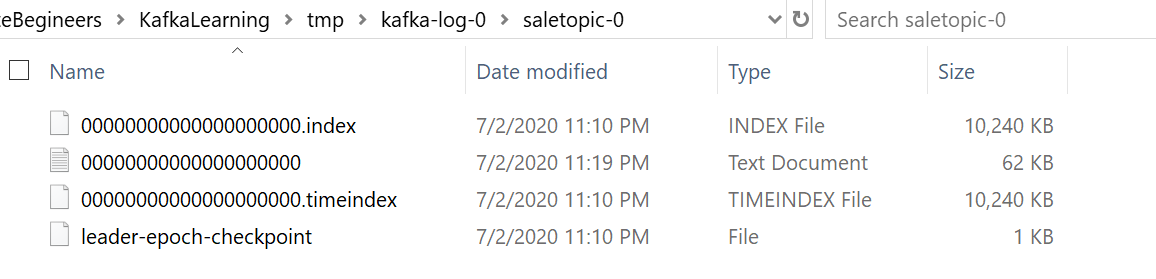
Note that one of the consumer in the group processes 1271 records, while other processes 636 records.

* Let’s looks inside the directories tmp\kafka-log-0, tmp\kafka-log-1, tmp\kafka-log-2



Note that we have a directory saletopic-0, where saletopic is the topic name and 0 is the partition number assigned to this broker.

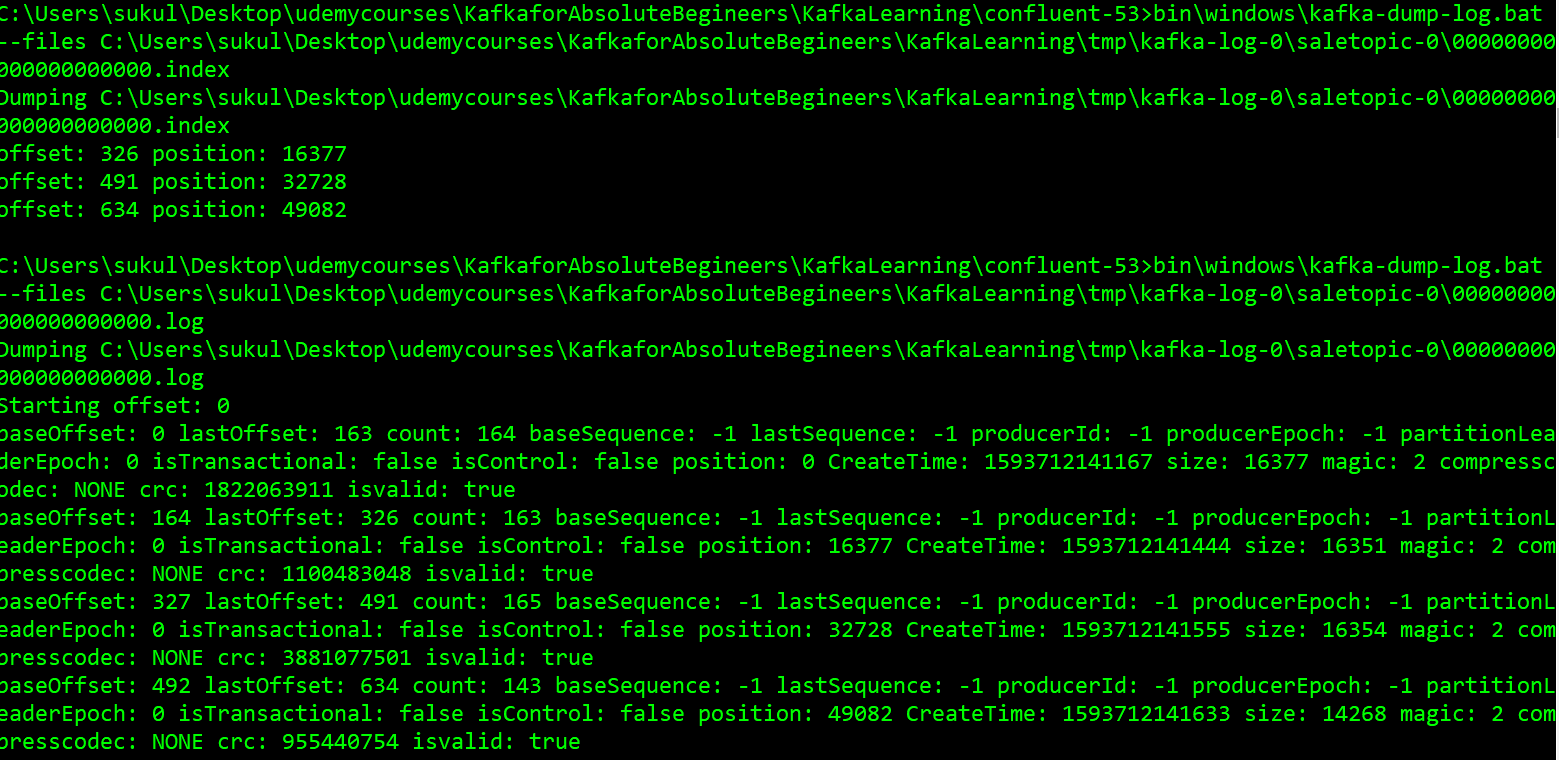
Inside this folder we can see the actual kafka log file created for partition 0. This is where the partition data is stored.



* We can look into these log files using the kafka dump tool as shown below:

bin\windows\kafka-dump-log.bat --files C:\Users\sukul\Desktop\udemycourses\KafkaforAbsoluteBegineers\KafkaLearning\tmp\kafka-log-0\saletopic-0\00000000000000000000.index

bin\windows\kafka-dump-log.bat --files C:\Users\sukul\Desktop\udemycourses\KafkaforAbsoluteBegineers\KafkaLearning\tmp\kafka-log-0\saletopic-0\00000000000000000000.log



* **Conclusion:**

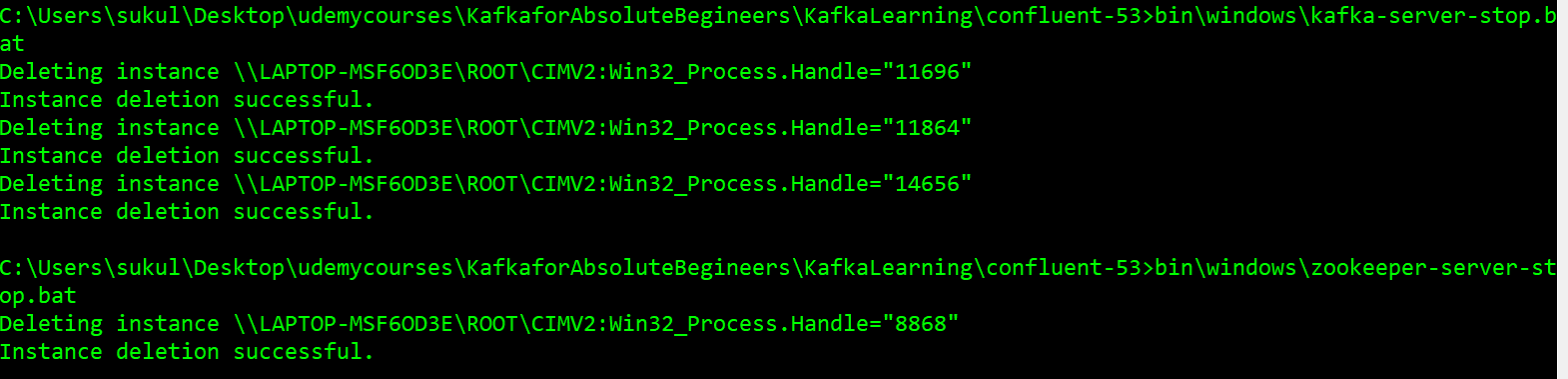
Kafka cluster will store data in the topic partitions.

Each partition is managed by a separate broker as a storage directory, and the actual data sets inside the log files. You can use the log dump tool to investigate these log files.

Consumers can work in consumer group to share the work load and try to achieve workload balance to the extent it is possible.

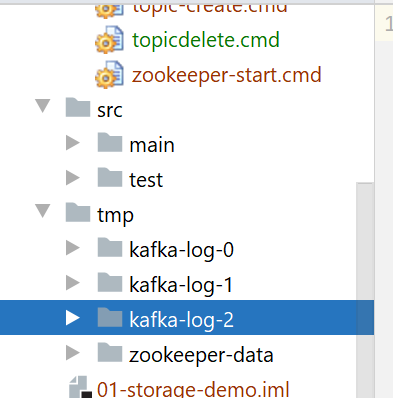
(Note that this sharing load does not mean a partition will be read by 2 consumers. One consumer in a group will only read one partition. Thus no of Partition becomes the degree of parallelism)

* **Stopping Servers and zookeeper:**



**Kafka Storage Architecture**

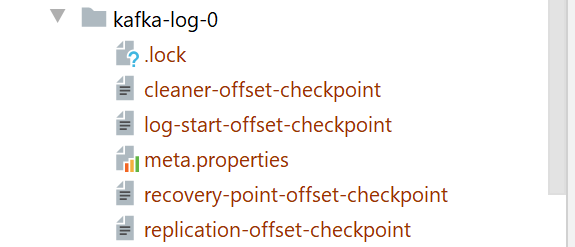
* Apache Kafka organizes the messages in topics, and the broker creates a log file for each topic to store these messages. However, these log files are partitioned, replicated and segmented.
* A topic is a logical name to group your messages. Like in a database, we must create a table to store our data records, in Kafka, you must create a topic to store messages.
* The Kafka broker/server properties and zookeeper properties are setup in such a way that zookeeper and Kafka data directories are created inside the following tmp directory in the project.



Inside each of the Kafka log directory (for each broker) we see following files. When a broker starts, it creates some initial files.

Most of these files would be empty at this point. We haven't created any topic yet, so we do not have any file for a topic.

This is an initial state of the Kafka broker.



In Kafka, a single topic may store millions of messages, and hence, it is not practical to keep all those messages in a single file.

* The **topic partitions are a mechanism to break the topic further into smaller parts.**

For Apache Kafka, **a partition is nothing but a physical directory**. Apache Kafka **creates a separate directory for each topic partition**.

* Below command creates a topic with 5 partitions but no replication.

C:\Users\sukul\Desktop\udemycourses\KafkaforAbsoluteBegineers\01-storage-demo\scripts>C:\Users\sukul\Desktop\udemycourses\Kafkafo

rAbsoluteBegineers\KafkaLearning\confluent-53\bin\windows\kafka-topics.bat --create --zookeeper localhost:2181 --topic Netflix --

partitions 5 --replication-factor 1 --config segment.bytes=1000000

Created topic Netflix.

The first argument is to tell Kafka that we want to create a topic. The second argument is the zookeeper coordinates. The third argument is the name of the topic. These three arguments are mandatory and fundamental arguments for the command to work. When you create a topic in Apache Kafka, you must specify two more configurations. Number of Partitions and Replication Factor.

If we do not provide these parameters, Kafka assumes a default value, but every topic must have some values for these two parameters.

Now we can see how the directories are created under Kafka Data directories. Since we only have 3 brokers, 5 partitions are divided amongst 3 brokers.

Two of the Brokers get 2 partitions and one gets 1 partition.

Note that each Partition directory name starts with Topic Name followed by the partition Number.

C:\Users\sukul\Desktop\udemycourses\KafkaforAbsoluteBegineers\01-storage-demo\tmp>dir /s Netflix\*

Volume in drive C has no label.

Volume Serial Number is 1EE8-7BCA

Directory of C:\Users\sukul\Desktop\udemycourses\KafkaforAbsoluteBegineers\01-storage-demo\tmp\kafka-log-0

07/03/2020 02:56 AM <DIR> Netflix-0

07/03/2020 02:56 AM <DIR> Netflix-3

0 File(s) 0 bytes

Directory of C:\Users\sukul\Desktop\udemycourses\KafkaforAbsoluteBegineers\01-storage-demo\tmp\kafka-log-1

07/03/2020 02:56 AM <DIR> Netflix-1

07/03/2020 02:56 AM <DIR> Netflix-4

0 File(s) 0 bytes

Directory of C:\Users\sukul\Desktop\udemycourses\KafkaforAbsoluteBegineers\01-storage-demo\tmp\kafka-log-2

07/03/2020 02:56 AM <DIR> Netflix-2

0 File(s) 0 bytes

* The **replication factor** specifies how many copies you want to maintain for each partition.  
  That simply means the replication factor multiplies to the number of partitions.

Following creates a new Topic with 5 Partitions, but replication factor of 2:

C:\Users\sukul\Desktop\udemycourses\KafkaforAbsoluteBegineers\01-storage-demo\scripts>C:\Users\sukul\Desktop\udemycourses\Kafkafo

rAbsoluteBegineers\KafkaLearning\confluent-53\bin\windows\kafka-topics.bat --create --zookeeper localhost:2181 --topic Amazon --p

artitions 5 --replication-factor 2 --config segment.bytes=1000000

Created topic Amazon.

C:\Users\sukul\Desktop\udemycourses\KafkaforAbsoluteBegineers\01-storage-demo\tmp>dir /s Amazon\*

Volume in drive C has no label.

Volume Serial Number is 1EE8-7BCA

Directory of C:\Users\sukul\Desktop\udemycourses\KafkaforAbsoluteBegineers\01-storage-demo\tmp\kafka-log-0

07/03/2020 03:11 AM <DIR> Amazon-1

07/03/2020 03:11 AM <DIR> Amazon-2

07/03/2020 03:11 AM <DIR> Amazon-3

0 File(s) 0 bytes

Directory of C:\Users\sukul\Desktop\udemycourses\KafkaforAbsoluteBegineers\01-storage-demo\tmp\kafka-log-1

07/03/2020 03:11 AM <DIR> Amazon-0

07/03/2020 03:11 AM <DIR> Amazon-2

07/03/2020 03:11 AM <DIR> Amazon-3

07/03/2020 03:11 AM <DIR> Amazon-4

0 File(s) 0 bytes

Directory of C:\Users\sukul\Desktop\udemycourses\KafkaforAbsoluteBegineers\01-storage-demo\tmp\kafka-log-2

07/03/2020 03:11 AM <DIR> Amazon-0

07/03/2020 03:11 AM <DIR> Amazon-1

07/03/2020 03:11 AM <DIR> Amazon-4

0 File(s) 0 bytes

Notice that a total of 10 directories are created and they are spread across 3 brokers. 1st replica of partition-0 is on Broker1, but the 2nd replica is on Broker 2. We term these directories as a partition replica.

Thus we have 10 directories part of the same Topic ‘Amazon’, but they are distributed among available brokers.

* We can classify topic partition replicas into two categories. **Leader partitions and Follower partitions.**While creating the Topic, we specified the number of partitions as five, and Kafka created five directories. These five directories are called the Leader Partitions. **So, the leaders are created first.**

Then we also specified the replication factor as two. That means Kafka should ensure 2 copies for each of these five partitions.

One is already there, the leader, hence Kafka creates 1 more directory for each leader, and we call these **copies as Followers.**

The number of follower partition depends on the replication factor.

The follower is a duplicate copy of the Leader. And all of them are nothing but directories.

To understand which partition is the leader and which is follower we can use the describe command as shown below:

C:\Users\sukul\Desktop\udemycourses\KafkaforAbsoluteBegineers\01-storage-demo\scripts>C:\Users\sukul\Desktop\udemycourses\Kafkafo

rAbsoluteBegineers\KafkaLearning\confluent-53\bin\windows\kafka-topics.bat --describe --zookeeper localhost:2181 --topic Amazon

Topic:Amazon PartitionCount:5 ReplicationFactor:2 Configs:segment.bytes=1000000

Topic: Amazon Partition: 0 Leader: 1 Replicas: 1,2 Isr: 1,2

Topic: Amazon Partition: 1 Leader: 2 Replicas: 2,0 Isr: 2,0

Topic: Amazon Partition: 2 Leader: 0 Replicas: 0,1 Isr: 0,1

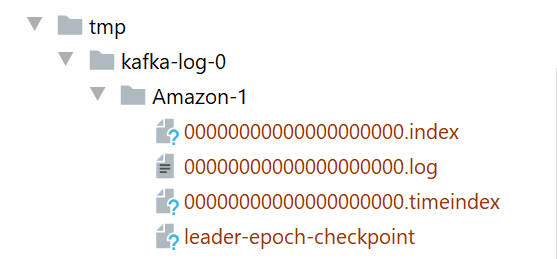
Topic: Amazon Partition: 3 Leader: 1 Replicas: 1,0 Isr: 1,0

Topic: Amazon Partition: 4 Leader: 2 Replicas: 2,1 Isr: 2,1

The output clearly tells, where the leader resides for the given partition. This number is the broker id

* As we know messages are stored within the directories in the log files. However, instead of creating one large file in the partition directory, Kafka creates several smaller files. That means the **Kafka log files is split into smaller files known as segments**.

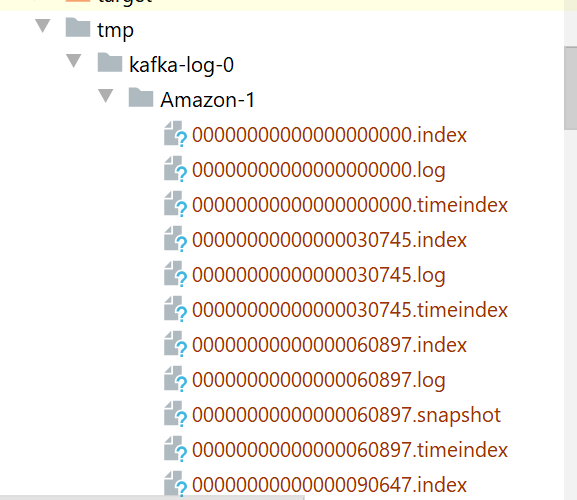
Following shows the 1st segment file created in one of the partitions. As more data gets added new segment files will be created.



When the partition receives its first message, it stores the message in the first segment. The next message also goes in the same segment and the segment file continues to grow until the maximum segment limit is reached. As the Kafka broker is writing to the partition, If the segment limit is reached, they close the file and start the new segment.

The default maximum segment size is either 1 GB. of data or a week of data, whichever is smaller.

Below shows how the segments look after we get some data in our partition. For our example, we have reduced the default maximum segment to 1 MB:



* **Kafka Message Offsets:** Each Kafka message in the partition is uniquely identified by a 64 bit integer offset.

For example, the offset for the first message in the partition would be 0000, the offset for the second message would be 0001, and so on. This numbering also continues across the segments to keep the offset unique within the partition. The offset for the first message in the new segment continues from the earlier segment. For easy identification, the segment file name is also suffixed by the first offset in that segment.

The offset is unique within the partition. If you look across the partitions, the offset is starts from zero in each partition.

Since the offsets are not unique across the topic, if you want to locate a specific message, you must know at least three things: Topic name, Partition number and then offset number.

* Consumer application requests messages based on the offset. Kafka allows consumers to start fetching messages from a given offset number. To help brokers rapidly find the message for a given offset, Kafka maintains an index of offsets. The index files are also segmented for easy management, and they are also is stored in the partition directory along with the log file segments.

These are the files with .index extension.

Consumer application can also request messages after a specific timestamp. To support such needs, Kafka also maintains the timestamp for each message builds a time index to quickly seek the first message that arrived after the given timestamp.

The time index is like the offset index, and it is also segmented and stored in the partition directory along with the offset index and log final segment. These are the files with .timeindex extension

Apart from these files there are also other files in the Kafka log directory that have nothing to do with data. Kafka creates those files to keep some control information and clean them from time to time.

**Apache Kafka – Cluster Architecture**

* Kafka Cluster: Kafka Brokers are often configured to form a cluster. A cluster is nothing but a group of brokers that work together to share the workload, and that's how Apache Kafka becomes are distributed and scalable system. We can start with a single broker and as the workload grows we can increase the number of brokers in the cluster.
* Notion of cluster brings 2 imp questions –
  + Who manages Cluster Membership: Normally in a typical distributed system, we have a master node that maintains a list of active cluster members. Master normally knows the state of other members. So the question in Kafka is who manages list of active brokers and knows which brokers and joined/left the cluster.
  + Who performs the routine administrative tasks in cluster: Who in Kafka perform admin tasks? Who is responsible for reassigning tasks when broker joins and leaves a cluster to ensure that cluster continues to function?
* Zookeeper:
  + Kafka Cluster is a Master-less Cluster. It does not follow the master-slave architecture. Apache Zookeeper is used to maintain the list of active brokers. Every Kafka broker has a unique id that you define in the broker configuration file. We also specify the zookeeper connection details in the broker configuration file.
  + When the broker starts, it connects to the zookeeper and creates an ephemeral node using broker-id to represent an active broker session. The ephemeral node remains intact as long as the broker session with the zookeeper is active.
  + When the broker loses connectivity to the zookeeper for some reason, the zookeeper automatically removes the ephemeral node. So the list of active brokers in the cluster is maintained as the list of ephemeral nodes under the /brokers/ids path in the zookeeper.
  + We can use the zookeeper prompt to check what is maintained on it:

Note that we have a hierarchical structure here. Zookeeper maintains the brokers, Topic on our brokers.

The list of active brokers in the cluster is maintained as the list of ephemeral nodes under the /brokers/ids path in the zookeeper.

C:\Users\sukul\Desktop\udemycourses\KafkaforAbsoluteBegineers\KafkaLearning\confluent-53\bin\windows>zookeeper-shell.bat localhos

t:2181

Connecting to localhost:2181

Welcome to ZooKeeper!

JLine support is disabled

WATCHER::

WatchedEvent state:SyncConnected type:None path:null

ls /

[cluster, controller\_epoch, controller, brokers, zookeeper, admin, isr\_change\_notification, consumers, log\_dir\_event\_notification

, latest\_producer\_id\_block, config]

ls /brokers

[ids, topics, seqid]

ls /brokers/ids

[0, 1, 2]

ls brokers/topics

Command failed: java.lang.IllegalArgumentException: Path must start with / character

ls /brokers/topics

[Netflix, Amazon]

As and when we add or remove brokers from the cluster, we can see that out of ‘ls /brokers\ids’ changes.

* Controller:
  + The controller in Kafka perform the routine administrative activities such as monitoring the list of active brokers and reassigning the work when an active broker leaves the cluster. The controller is not a master. Kafka is a master less cluster, and the list of active brokers is maintained in the zookeeper. It is simply a broker that is elected as a controller to pick up some extra responsibilities. SO a Controller also does the regular broker tasks.
  + There is only one controller in the Kafka cluster at any point in time. The controller is responsible for monitoring the list of active brokers in the zookeeper. When the controller notices that a broker left the cluster, it knows that it is time to reassign some work to the other brokers.
  + Controller Election: The first broker that starts in the cluster becomes the controller by creating an ephemeral (controller) node in the zookeeper. When other brokers start, they also try to create this node, but they receive an exception as 'node already exists,’ which means that the controller is already elected. In that case, they start watching the controller more than the zookeeper to disappear. When the controller dies, the ephemeral node disappears. Now, every broker again tries to create the controller node in the zookeeper, but only one succeeds, and others get an exception once again. This process ensures that there is always a controller in that cluster, and there exists only one controller.
  + We can invoke the following command on the zookeeper prompt to see which broker is a controller:

get /controller

{"version":1,"brokerid":0,"timestamp":"1593774318409"}

cZxid = 0x71

ctime = Fri Jul 03 16:35:18 IST 2020

mZxid = 0x71

mtime = Fri Jul 03 16:35:18 IST 2020

pZxid = 0x71

cversion = 0

dataVersion = 0

aclVersion = 0

ephemeralOwner = 0x100033717230000

dataLength = 54

numChildren = 0

Now if the broker-0 fails, other broker will be elected as the controller. We can see that broker 2 was elected after we terminated the broker 0

get /controller

{"version":1,"brokerid":2,"timestamp":"1593775732026"}

cZxid = 0xb3

ctime = Fri Jul 03 16:58:52 IST 2020

mZxid = 0xb3

mtime = Fri Jul 03 16:58:52 IST 2020

pZxid = 0xb3

cversion = 0

dataVersion = 0

aclVersion = 0

ephemeralOwner = 0x100033717230002

dataLength = 54

numChildren = 0

Now even if the broker-0 comes back, then it doesn't become a controller. Someone else is already elected as a controller.

* A Kafka Topic is broken into two independent partitions. Each partition is self-contained .i.e All the information about the partition, such as segment files and indexes are stored inside the same directory. This allows us to distribute the work among Kafka brokers in a cluster efficiently. When you create a topic, the responsibility to create, store, and manage partitions is distributed among the available brokers in the cluster. Every Kafka broker in the cluster is responsible for managing one or more partitions that are assigned to that broker. This is how work is shared between Brokers.
* When allocating partitions on brokers, Kafka tries to achieve two goals:
  + Evenly distribute the partitions as much as possible to achieve load balance.
  + Follower partitions/Duplicate copies are placed on different machines to achieve tolerance.

These are not rules. Kafka tries to achieve them as much as possible.

* To distribute partitions, Kafka creates a list of available brokers and assigns headers and followers in order.

It randomly selects a broker and places it into a list. Next broker on the list must come from a different rack. This goes on as an alternating process for selecting another broker in a different rack. Once this alternating list of brokers is ready, Kafka starts assigning partitions to the list. Once we have the ordered list of available brokers, assigning partitions is as simple as assign one to each broker using a round robin method. Kafka starts with the leader partitions and finishes creating all leaders first. Once the leader partitions are created, it starts creating the first follower. The first follower allocation simply starts from the second broker in the list and follows a round robin approach. The 2nd follower allocation starts from the 3rd broker in the list and follows a round robin approach. Thus Leaders and followers of the topic are created across the cluster.

This Kafka achieves fault tolerance by merely placing partition copies on different machines across the rack. This fault tolerance makes sure that is a broker fails, we still have some copy on some other broker. Also the placement of the partition followers is such that if the entire rack fails, we still have a copy on a different rack.

* **Leader and Follower activities:** We know that broker manages two types of Partitions: Leader Partition and Follower Partitions. Depending upon the partition type, a typical broker performs two kinds of activities. Leader activities and follower activities.

The leader is responsible for all the requests from the producers and consumers. When Producer wants to send messages to a Kafka topic, it will connect to one of the brokers in the cluster and query for the topic metadata. All Kafka brokers can answer the metadata request, and hence the producer can connect to any of the broker and query for that metadata.

The metadata contains a list of all the leader partitions and their respective host and port information. Now the producer has a list of all leaders.

It is the producer that decides on which partition does it want to send the data, and accordingly send the message to the respective broker.

That means the producer directly transmits the message to a leader. On receiving the message, the leader broker persists the message in the leader partition and sends back an acknowledgement. Similarly, when a consumer wants to read message. It always reads from the leader of the partition.

Thus producers and consumers always interact with the Leader. Thus the responsibility of the leader is to interact with the consumer and producers.

Followers do not serve producer and consumer requests. Their only job is to copy messages from the leader and stay up to date with all the messages. The aim of the follower is to get elected as a leader when the current leader fails or dies. So, they have a single point agenda. Stay in sync with the leader. To stay in sync with the leader, the follower connects to the leader and requests for the data. The leader send some messages, and the followed persists them in the replica and requests for more.

This goes on forever as an infinite loop to ensure that the followers are in sync with the leader.

* **ISR List :**

We know that followers would continuously copy the messages from the leader and stay in -sync with the leader. However, some followers can still fail to stay in sync for various reasons like network congestion and broker failures. Network congestion can slow down replication, and the followers may start falling behind. When a follower broker crashes, all replicas on that broker will begin falling behind until we restart the follower broker and they can start replicating again.

Since replicas can start falling behind, leader maintains a list of In-Sync-Replicas (ISR). This is the list of the partition and persisted in the zookeeper, and this list is maintained by the leader broker. ISR list is critical because all the followers in that list are known to be in sync with the leader, and they are an excellent candidate to be elected as a new leader when something wrong happens to the current leader.

The ISR list is easy to maintain. If the replica is 'not too far,’ the leader will add the follower to the ISR list, or else the followed is removed from the ISR list.

That means the ISR list is dynamic and the followers keep getting added and removed from the ISR list depending on how far they maintain their in-sync status.

Remember that followers will always be a little behind and the leader, and that's obvious because follower needs to ask for the message from the leader, received the message over the network, store them into the replica and then ask for more. So there needs to be a definition of what “not too far” means. The default value of 'not too far' is 10 seconds.

However, you can increase or decrease it using Kafka configurations. So the replica is kept in the ISR list if they are not more than 10 seconds behind the leader.

That means, if the replica has requested the most recent message in the last 10 seconds, they deserve to be in the ISR. If not, the leader removes the replica from the ISR list.

* **Committed vs Uncommitted records:**

To prevent data loss in case of leader failure before the replicas catchup, Kafka introduces the concept of committed and uncommitted messages. We can configure the leader to not to consider a message committed until the message is copied at all the followers in the ISR list. If the message is committed, we cannot lose it until we lose all the replicas.

However when the leader is lost, we may still miss the uncommitted messages. But the uncommitted messages shouldn't be a worry, because those can be read sent by the producer. Producers can choose to receive acknowledgement of sent messages only after the message is fully committed.

In that case, the producer waits for the acknowledgement for a timeout period and the reason the messages in the absence of commit acknowledgement. So, the uncommitted messages are lost at the failing leader, but the newly elected leader will receive those messages again from the producer.

That's how all the messages can be protected from loss.

* **Minimum ISR**

The data is considered committed when it is written to all the in-sync replicas. Now let's assume that we start with three replicas and all of them are healthy enough to be in the ISR.

However, after some time, two of them failed, and as a result of that, the leader will remove them from the ISR. In this case, even though we configured the topic to have three replicas, we are left with the single in-sync replica that is the leader itself.

Now the data is considered committed when it is written to all in-sync replicas, even when all means just one replica (the leader itself), right? It is a risky scenario for data consistency because data could be lost if we lose the leader. Kafka protects this scenario by setting the minimum number of in-sync replicas for a topic. If you would like to be sure that committed data is written to at least two replicas, you need to set the minimum number of in-sync replicas as two.

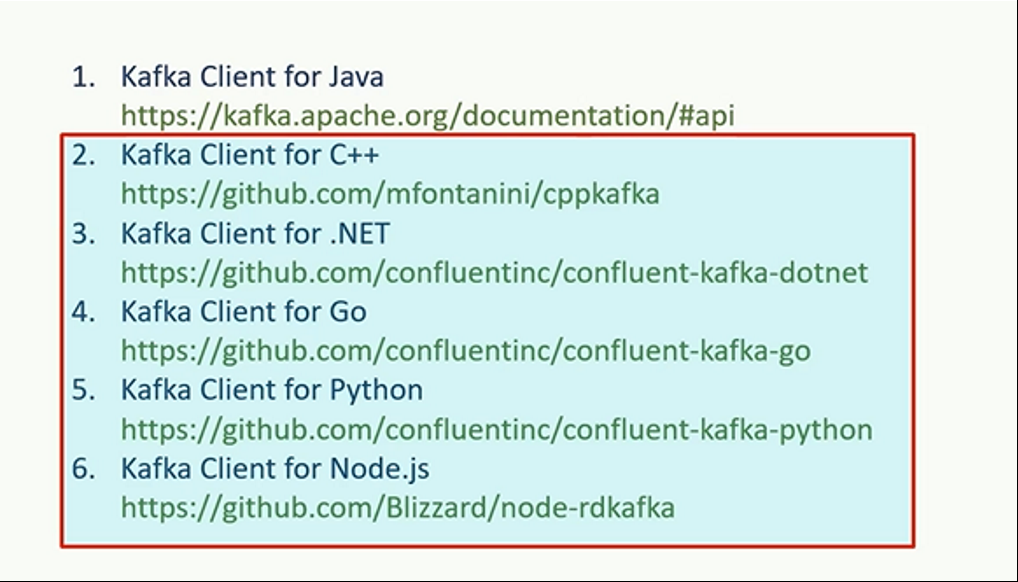
**Side Effect:** If a topic has three replicas and you set minimum in-sync replicas to two, then you can only write to a partition in the topic if at least two out of the three replicas are in-sync. When at least two replicas are not in sync, the broker will not accept any messages and respond with 'not enough replicas’ exception.

In other words, the leader becomes a read only partition.

**Kafka Producer Internals**

* Kafka producer API is primarily made available in Java.

However the API is are also available in other languages and all those language bindings follow the same structure as Java APIs. There are many other language bindings that are implemented by open source community. But except for Java API all the above language APIs are based on the Kafka client library named librdkafka.



* Sending Data to Kafka is a multistep process.

**First step** is to create a Java properties object and put some necessary configurations in it.

Kafka producer API is highly configurable, and we customize the behavior by setting different producer configurations.

4 basic configurations include:

* CLIENT\_ID\_CONFIG, which is a simple string that is passed to the Kafka server.The purpose of the client id is to track the source of the message.
* BOOTSTRAP\_SERVER\_CONFIG is a comma-separated list of host/port pairs. The producer will use this information for establishing the initial connection to the Kafka cluster. The bootstrap configuration is used only for the initial connection. Once connected, the Kafka producer will automatically query for the metadata and discover the full list of Kafka brokers in the cluster. That means you do not need to supply a complete list of Kafka brokers as a bootstrap configuration. However, it is recommended to provide two to three broker addresses of a multimode cluster. Doing so will help the producer to check for the second or third broker in case the first brokers in the list is down.
* KEY\_SERIALIZER\_CLASS\_CONFIG : Key Serializer class
* VALUE\_SERIALIZER\_CLASS\_CONFIG : Value Seriaizer class

|  |
| --- |
| *logger*.info("Creating Kafka Producer..."); Properties props = new Properties(); props.put(ProducerConfig.*CLIENT\_ID\_CONFIG*, AppConfigs.*applicationID*); props.put(ProducerConfig.*BOOTSTRAP\_SERVERS\_CONFIG*, AppConfigs.*bootstrapServers*); props.put(ProducerConfig.*KEY\_SERIALIZER\_CLASS\_CONFIG*, IntegerSerializer.class.getName()); props.put(ProducerConfig.*VALUE\_SERIALIZER\_CLASS\_CONFIG*, StringSerializer.class.getName()); |

A Kafka message must have a key/value structure. That means each message that we want to send to the Kafka server should have a key and a value. We can have a null key, but the message is still structured as a key/value pair.

Kafka messages are sent over the network. So, the key and the value must be serialized into bytes before they are streamed over the network.

Kafka producer API comes with a bunch of ready to use serializer classes.

In this example, we are setting an IntegerSerializer for the key and a StringSerializer for the message value.

**Second Step** is to create the Kafka Producer and pass these properties to the constructor.

**Third Step** is to start sending messages to Kafka. Producer object has a send method and it takes a ProducerRecord object and sends it to the Kafka cluster.

ProducerRecord constructor takes three arguments. – Topic Name, Message Key, Message itself.

|  |
| --- |
| KafkaProducer<Integer, String> producer = new KafkaProducer<>(props);  *logger*.info("Start sending messages..."); for (int i = 1; i <= AppConfigs.*numEvents*; i++) {  producer.send(new ProducerRecord<>(AppConfigs.*topicName*, i, "Simple Message-" + i)); }  *logger*.info("Finished - Closing Kafka Producer."); producer.close(); |

**The last and final step** is to close the producer instance. The producer functionality is involved, and it does a lot of things internally. Producer consists of some buffer space and background I/O thread.

If you do not close the producer after sending all the required messages, we will leak the resources created by the producer.

To Summarize Sending messages to Kafka using producer API is a four-step process.

* In the first step we will set some configurations that will control the behavior of your producer API.
* The second step is to create a producer object.
* The third step is to send all the messages.
* Finally, if you have no further messages to send, close the producer.
* **Producer Record:** Kafka producer APIs are straightforward.

You can create a producer by setting some essential configurations and start sending messages using the send() method. However we must package your message in a ProducerRecord object.

When creating a producer record we have 2 mandatory arguments - Kafka topic name, and message value

Other than these two mandatory arguments, we can also specify the following optional items: Message key, target partition, message timestamp.

The message key is one of the most critical argument, and it is used for many purposes, such as partitioning grouping, and joins.

We should consider it as another mandatory argument even if the API doesn't mandate it. Target partition and the timestamp are purely optional.

* **Producer Serialization:**

The Kafka producer is supposed to transmit the ProducerRecord to the Kafka Broker over the network. Every record goes through serialization, partitioning, and then it is kept in the buffer. The serialization is necessary to send the data over the network. Without serializing data, we can't transmit it to a remote location. We are incharge of specifying key and value serializer classes as part of producer configuration. We have seen IntegerSerializer for the Key and a StringSerializer for the value.

However, these serializers are the most elementary serializers, and they do not cover most of the use cases. In a real life scenario, your events are represented by complex Java objects. Kafka gives you an option to use a generic serialization library like Avro or Thrift.

Alternatively, you have a choice to create custom serializer.

* **Partitioner:** Every ProducerRecord includes a mandatory topic name as the destination address of the data. However, the Kafka topics are partitioned, and hence, the producer should also decide on which partition the message should be sent.

There are two approaches to specify the target partition number for the message.

* Set partition number argument in the ProducerRecord.
* Supply a partitioner class to determine the partition number at runtime. The Partitioner class implements desired partitioning strategy and assigns a partition number to each message at runtime.

We can specify a custom partitioner using the properties object. Kafka Producer comes with a default partitioner which is the most commonly used partitioner.

The default partitioner takes one of the two approaches to determine the destination topic partition:

* Hash key Partitioner: This approach is based on them message key. When the message key exists, the default partitioner will use the hashing algorithm on the key to determining the partition number for the message. The hashing ensures that all the messages with the same key go to the same partition. However, this algorithm also takes the total number of partitions as one of the inputs. So, if you increase the number of patients in the topic, the default partitioner starts giving a different output. That means, if the partition is based on the key, then you should create a topic with enough partitions and never increase it at the later stage.
* Round Robin Partitioning: When the message key is null, the default partitioner will use a round robin method to achieve an equal distribution among the available partitions.

The default partitioner is the most commonly used partitioner in most of the use cases.

However, Kafka allows you to implement your partitioning strategy by implementing a custom partitioner class.

* **Message Timestamp:** The ProducerRecord takes an optional timestamp field. For a real-time streaming application, the timestamp is the most critical value. For that reason, every message in Kafka is automatically time stamped, even if you do not explicitly specify it.

Kafka allows you to implement one of the two types of message time a stamping mechanism.

* The CreateTime is the time when the message was produced.
* The LogAppendTime is the time when the message was received at the Kafka broker.

We cannot use both. The application must decide between these two timestamping methods while creating the topic. We can set the default timestamp method by setting the topic configuration message.timestamp.type. Set to 0 for CreateTime or 1 for using LogAppendTime. Default value is 0.

When this config is set to 0, the producer API automatically sets the current producer time to the ProducerRecord#timestamp field.

However, you can override the auto time stamping by explicitly specifying this argument.

So, the message is transmitted with the producer time, either automatically set by the producer, or explicitly set by the developer.

When using LogAppendTime configuration, the broker will override the producer timestamp with its current local time before appending the message to the log. In this case, the producer time is overwritten by the broker time.

However, the message will always have a timestamp, either a producer time or the broker time.

* **Producer Message Buffer:** Once serialized and assigned a target partition number, the message goes to sit in the buffer waiting to be transmitted.

The producer object consists of a partition-wise buffer space that holds the records that haven’t yet been sent to the server.

The Producer also runs a background I/O thread that is responsible for turning these records into requests and transferring them to the cluster.

The buffering of the message is designed to offer two advantages:

* + Asynchronous send API: sender method will add the messages to the buffer and return without blocking. Those records are then transmitted by the background thread. The send() method is not delayed for the network operation.
  + Network Roundtrip Optimization: Buffering also allows the background I/O thread to combine multiple messages from the same buffer and transmit them together as a single packet to achieve better throughput.

If the records are posted faster than they can be transmitted to the server, then this buffer space will be exhausted, and your next send method will block for few milliseconds until the buffer is freed by the I/O. If the I/O thread takes too long to release the buffer, then your send method throws a TimeoutException.

When you are seeing such timeout exceptions, you may want to increase the producer memory. The default producer memory size is 32MB. You can expand the total memory allocated for the buffer by setting, buffer.memory Producer configuration.

* **Producer IO Thread and retries:**

The producer background I/O thread is responsible for transmitting the serialized messages that, are waiting in the topic partition buffer. When the broker receives the message, it sends back an acknowledgment.

If the message is successfully written to Kafka, the broker will return a success acknowledgement. If the broker failed to write the message, it would return an error. When the background I/O thread receives an error or does not receive an acknowledgment, it may retry sending the message a few more times before giving up and throwing back an error. You can control the number of read drives by setting the retries.producer configuration. When all the retries are failed, the I/O thread will return the error to the send() method.

* Summary of Producer Internals:
* We use the producer.send() method to handover the ProducerRecord to the KafkaProducer object.
* The KafkaProducer object will internally serialize the message key and the message value. And we provide the serializers using the properties object.
* Then the producer will determine the target partition number for the message to be delivered.
* You can provide a custom partitioner class using the properties object or provide a key and let the producer use the default partitioner.
* The serialized message goes and sits into a buffer depending upon the destination address. For each destination, we have a separate buffer.
* Finally, an I/O thread that runs in background will pick up some messages from the buffer, combine them to make one single data packet and send it to the broker.
* The broker would save the data in the log file and send back an acknowledgement to I/O thread.
* If the I/O thread does not receive an acknowledgement, it will try sending the packet and again wait for an acknowledgement.
* If we do not get an acknowledgement at all even after some retries, or get an error message, the I/O thread will give the error back to the send method.

**Advanced Kafka Producers**

* **Producer Scalability:**

Apache Kafka was designed with the scalability in mind. From the broker side, we can increase the number of Kafka brokers in your cluster and support hundreds of thousands of messages to be received and acknowledged.

Two ways of providing Producer side Scalability:

* + On the producer side, you can keep adding the new producers to send the messages to the Kafka server in parallel. This arrangement provides linear scalability by merely adding more producers and brokers. This approach works perfectly for scaling up your overall streaming bandwidth.
  + Also have an opportunity to scale an individual producer using multithreading technique .A single producer thread is good enough to support the use cases where the data is being produced at a reasonable pace. However, some scenarios may require parallelism at the individual producer level as well. You can handle such requirements using multithreaded Kafka producer. The multithreading scenario may not apply to applications that do not frequently generate new messages. However, if you have an application that generates or receives data at high speed and wants to send it as quickly as possible, you might want to implement a multithreaded application.
* **At least once and At most once semantics:**

At this point we know that:

* + Apache Kafka provides message durability guaranties by committing the message at the partition log. Once the data is persisted by the leader broker in the leader partition, we can't lose that message till the leader is alive. However when a leader is down we may lose messages.
  + To protect the loss of records due to leader failure, Kafka implements replication. Kafka implements replication using followers. The followers will copy messages from the leader and provide fault tolerance in case of leader failure. When the data is persisted to the leader as well as the followers in the ISR list, we consider the message to be fully committed. Once the message is fully committed, we can't lose the record until the leader, and all the replicas are lost, which is an unlikely case.

Kafka provides at least once semantics:

However we still have a possibility of committing duplicate messages, due to the producer retry mechanism. We know that if the producer I/O thread fails to get a success acknowledgment from the broker, it will retry to send the same message. So under a scenario where acknowledgement fails to each back to the I/O thread due to a network error. In that case, the producer I/O thread will wait for the acknowledgment and ultimately send the record again assuming a failure.

The broker again receives the data, but it doesn't have a mechanism to identify that the message is a duplicate of an earlier message. Hence, the broker saves the duplicate record causing a duplication problem.

This implementation is known as at-least-once semantics, where we cannot lose messages because we are retrying until we get a success acknowledgment.

However, we may have duplicates because we do not have a method to identify a duplicate message.

For that reason, Kafka is said to provide at-least-once semantics.

Kafka also allows us to implement at-most once semantics:

We can achieve at most once semantics by configuring the retries to zero.

* **Exactly once delivery semantics in Kafka:**

At this point we know that Kafka is at-least once system by default, and you can configure it to get at-most once. To meet exactly once requirement, Kafka offers an idempotent producer configuration. All you need to do is to enable idempotence, and Kafka takes care of implementing exactly-once. To enable idempotence, you should set the enable.idempotence producer configuration to true.

Once we configure idempotence, behavior of the producer API changes. It does two things:

* It will perform an initial handshake with the leader broker and ask for a unique producer id. At the broker side, the broker dynamically assigns a unique ID to each producer.
* Next things it does is message sequencing. The producer API will start assigning a sequence number to each message. This sequence number starts from zero and monotonically increments per partition. Now, when the I/O thread sends a message to a leader, the message is uniquely identified by the producer id and a sequence number. Now, the broker knows that the last committed message sequence number is X, and the next expected message sequence number is X+1. This allows the broker to identify duplicates as well as missing sequence numbers.

What does enabling idempotence does not guarantee:

If you are sending duplicate messages at your application level, this configuration cannot protect you from duplicates. That should be considered as a bug in your application.

Even if two different threads or two producer instances are sending duplicates that too is an application design problem. The idempotence is only guaranteed for the producer retires. Idempotence is not guaranteed for the application level message resends.

Or duplicates send by the application itself.

**Types and Serialization**

In real life examples records are not plain text or string messages. They are complex Java Objects.

Creating POJO(Plain Old Java Objects) for the message type is a tedious mechanical activity. However we can automate this activity. We have the ability to define a message schema using some simple Schema Definition language and then IDE or a build tool that can generate java class definition from the schema definition automatically.

Two options used normally are:

* JSON Schema to POJO
* Avro Schema to POJO

Generating POJO are just the 1st half of the problem. Next requirement is to create serializer and deserializer for those java types/classes. We Can we develop reusable serializers and deserializers and apply the same to all of the Java types.

For generating POJO from JSON we can use the open source project named jsonschema2pojo. There might be other options too. However, JsonSchema2Pojo is a reasonable choice for being able to produce Java as well as Scala code. This has its own datatypes that we specify in the Json. Those Datatypes are converted to corresponding datatypes in Java. This project also allows us to specify inheritance using our JSON schema. This will make sure the Java classes created have the proper hierarchy.

The jsonschema2pojo comes as a maven plugin. We need to configure it in our pom.xml file. In this configuration we also specify the source directory for the schema definition files, we also specify the output directory where all the generated packages and classes would land.

When we build/compile the project the POJO from schema will be automatically created.

Similarly there is also an Avro Maven plugin that allows generating Java classes from the avro schema (.avsc). Avro generated classes do not support inheritance (jsonschema2pojo supports inheritance).

**Kafka Consumers**

Kafka offers us 3 tools for consuming data: Consumer APIs, Kafka streams library, and KSQL. You can use Kafka consumer APIs to consume data from your Kafka brokers and create your stream processing applications.

Creating a Kafka Consumer is also a 4 step process (Similar to Kafka producer)

**First step** is to create a Java properties object and set required Kafka consumer configurations.

Same as producer API, Kafka consumer API is also highly configurable, and we customize the behavior by setting the config parameters. Important properties include client-Id, Bootstrap Server, Key Deserializer and value deserializer (At the source end if we used JSON serializer, we shall use JSON deserializer here. Deserializers are needed because kafka transmits a message after serializing into raw bytes). One important property is the VALUE\_CLASS\_NAME\_CONFIG – this is the target deserialized Java class name. We can also specify the consumer groups and kafka offsets and consumer positions.

**Second step** is to create the instance of KafkaConsumer class.

**Third step** is to subscribe to a topic using the subscribe method. A Kafka consumer can subscribe to a list of topics.

**Fourth step** is to read the messages. This normally would be an infinite loop as consumers are expected to keep reading the records and process them in real-time for the life of the application. The consumer can start requesting the message records by making a call to the poll() method. The poll() method will immediately return an Iterable ConsumerRecords.

If there are no records at the broker, it will wait for the timeout. When the timeout expires, an empty ConsumerRecords will be returned.

**Consumer Group and Scalability:**

If the rate at which producers write messages to the Topic, exceeds the rate at which you can transform and write them to external storage, the consumer application will start falling behind. If you are limited to a single consumer reading and processing the data, your application may fall farther and farther behind and would no longer remain a real-time application.

We can scale a consumer application by dividing the work among multiple consumers.

Just like numerous producers can write to the same Topic, we need to allow various consumers to read from the same Topic.

When we have multiple consumers working in a group and reading data from the same Topic, we can easily split the data among consumers by assigning them one or more partitions.

In this arrangement, each consumer is attached with a set of Partitions, and they read data only from the assigned partitions.

This arrangement clearly divides the data among the consumers to ensure that they do not read the same message. In this arrangement, every record is delivered to one and only one consumer in the group, and there is no duplicate processing.

This arrangement also adds a restriction on the scalability that would be equal to the number of partitions in a given topic.

Kafka offers automatic group management and rebalancing of the workload in a consumer group. All we need to do is to set the group id configuration.

Kafka automatically forms a consumer group, and it would also add the consumer to the same group if they have the same group id.

Kafka will also take care of assigning partitions to the consumer in the same group.

Membership in a consumer group is maintained dynamically.

If a consumer fails, the partitions assigned to it will be reassigned to other consumers in the same group.

Similarly, if a new consumer joins the group, partitions will be moved from existing consumers to the new one to maintain the workload balance.

Conceptually, you can think of a consumer group as being a single logical consumer that happens to be made up of multiple processes sharing the workload.

Kafka automatically manages all of this, and the whole process is transparent to users.

You, as a programmer is responsible for setting the group id configuration and starting a new consumer process either on the same machine or on a separate computer.