**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**

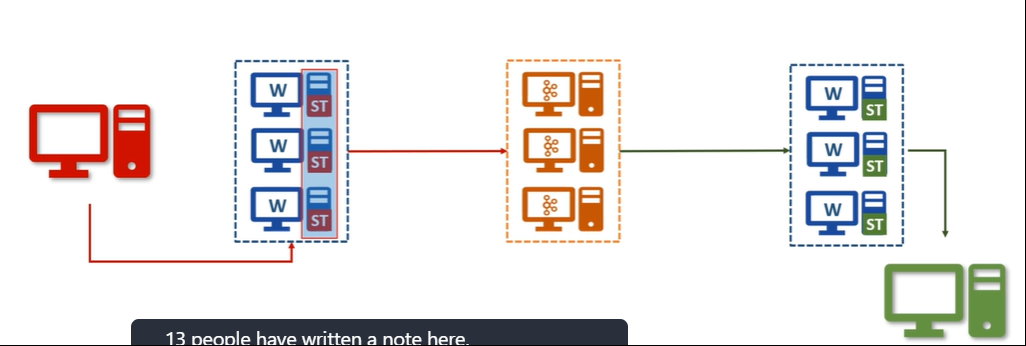
* Kafka Connect is a component of Kafka for connecting and moving data between Kafka and external systems. We have two types of Kafka Connectors. Source Connector and Sink connector. And together, they support a bunch of systems and offer you an out of the box data integration capability without writing 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.

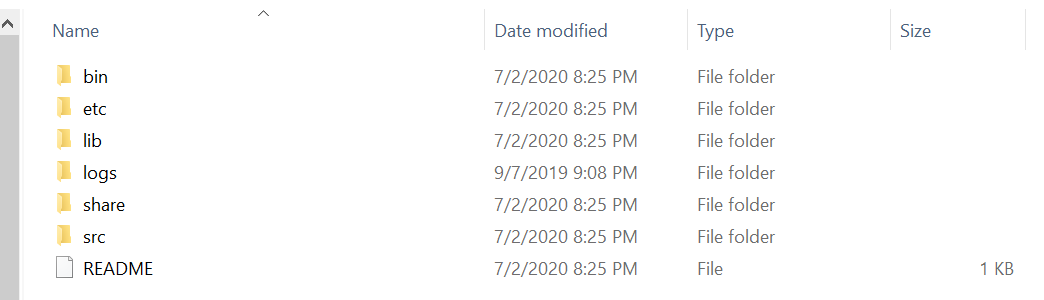
* KSQL Architecture:
* The KSQL comes with three components.
  + KSQL engine,
  + REST interface and
  + KSQL CLI/UI.
* The KSQL engine and the REST interface together form the KSQL server. The KSQL server can be deployed in one of the available modes. Interactive mode and Headless Mode. We can also 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?**

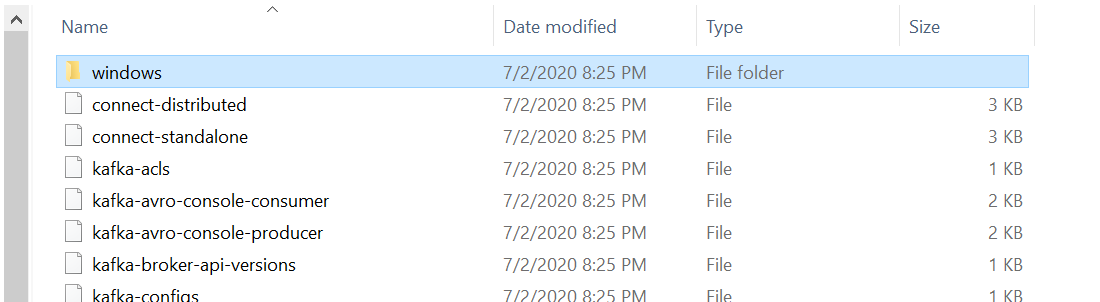
* 3 patterns in which Kafka is normally used:
* Data integration pattern - using a combination of the first three Kafka components. Kafka broker, Kafka Client API and Kafka connect.
* Real-time stream processing using micro service architecture
* Real-time data warehousing pattern.

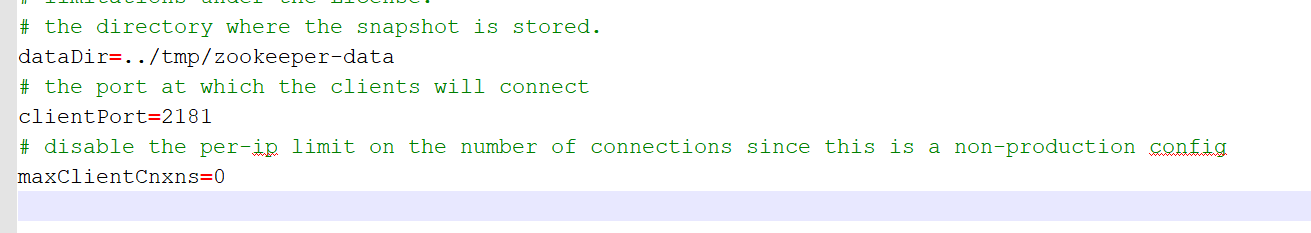
**Installing Single node Kafka**

* 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