APACHE KAFKA

INVENT THE FUTURE

ERNESTO LEE

APACHE KAFKA
INVENT THE FUTURE

ERNESTO LEE

APACHE KAFKA

Table of Contents

CHAPTER 1:
INTRODUCTION TO APACHE KAFKA

THEORY

This chapter is intended to provide a comprehensive introduction to Apache Kafka, the focus throughout this book. We shall present a brief overview of BigData before discussing Kafka.

OVERVIEW OF BIGDATA

Brief Introduction to Hadoop

Apache Hadoop is an open-source distributed framework that allows for the storage and processing of large data (BigData) sets across clusters of commodity machines. Hadoop overcomes the traditional limitations of the storing and computing of data by distributing the data over clusters of commodity machines, making it scalable and cost-effective.

The concept of Hadoop was originated when Google released a whitepaper about the **Google File System (GFS)**, a computing model built by Google designed to provide efficient, reliable access to data using large clusters of commodity hardware. Following this, the model was adopted by Doug Cutting and Mike Cafarella for their search engine called “Nutch.” Hadoop was then developed to support distribution for the Nutch search engine project by Doug Cutting and Mike Cafarella. What does the name Hadoop mean? The name has no significance, nor is it an acronym. Rather, Hadoop is the name that Doug Cutting’s son gave to his yellow stuffed elephant. The name is therefore unique, easy to remember, and sometimes funny. Not only does Hadoop have a name with no significance, but also its sub-projects tend to have such names, which are based on names of animals, such as Pig,

for the same reason. These names are unique, not used anywhere else, and easy to remember.

Why Hadoop?

Companies today have begun to realize that there is much information in unstructured documents spread across the network. Much data is available in the form of spreadsheets, text files, emails, logs, PDFs, and other data formats that contain valuable information and can help to discover new trends, design new products, improve existing products, know customers better, and so on. Data is increasing at an alarming rate beyond limits never seen before, and there are no signs of slowing, at least in the near future. To deal with such data, we need a reliable and low-cost tool to meaningfully process it; therefore, we use Hadoop. Hadoop helps us to reliably process all the BigData present in a variety of formats, in a much shorter time and in a flexible and cost-effective manner.

Let us explore why Hadoop is so popular and what it offers:

- **Scalable**: Hadoop is scalable; thus, the user can begin with a single-node server and eventually increase to more nodes as additional storage and computing power are needed.

- **Fault-Tolerant**: Hadoop helps to prevent the loss of data. All the data stored in the Hadoop Distributed File System is broken into blocks and stored with a default replication factor of 3. While processing the data, if a node goes off, the process does not stop but continues, as the data still exists in other nodes.

- **Flexible**: Hadoop does not require a schema. Hadoop can process unstructured, semi-structured, and structured data from any type of source or even from multiple sources.

- **Cost-effective**: Hadoop does not require expensive, high-end computing hardware. Hadoop works well with a cluster of commodity machines by parallel computing.

Brief Introduction to the Hadoop Distributed File System

Hadoop Distributed File System (HDFS) is a file system that extends over a cluster of commodity machines rather than a single high-end machine. The HDFS is a distributed large-scale storage component and is highly scalable. Additionally, the HDFS can accept node failures without losing data, and it is widely known for its reliability. Let us now examine why HDFS is so favorable in terms of distributed file systems.

Reliable Data Storage	The HDFS is very reliable in terms of data storage. The data stored in the HDFS is replicated by a default replication factor of 3. Therefore, even if a machine fails, the data will still be available in two other machines.
Cost-Effective	The HDFS can be deployed on clusters of commodity hardware and save on much money. High-end, expensive hardware is not required by the HDFS to function.
Big Datasets	The HDFS is capable of storing petabytes of data over a cluster of machines, in which a file can range from gigabytes to terabytes in size. The HDFS is not designed to store a large number of small-sized files, as the file system metadata is stored in the memory of NameNode.
Streaming Data Access	The HDFS provides streaming access to data. It is best suited for batch processing data and is not suitable for interactive processing. Moreover, the HDFS is not designed for applications that require low latency access to data, such as online transaction processing (OLTP).

Simple Coherency Model	The HDFS is designed to *write once and read many times* as its access model for files. Appending the content to files is supported at the end but cannot be updated at an arbitrary point, and it is not possible to have multiple writers. Files can be written by only a single writer.

Block Placement in HDFS

Hadoop is designed such that the first block replica is placed on the same node as the client, and the second replica is placed on a different rack from that of the first replica. The third replica is placed on a random node on the same rack as the second replica. If the replication factor is greater, random nodes in the cluster are selected to place the replicas. If a client running outside the cluster stores a file, a random node (that is not busy) is selected to place the first replica. This way, if a node fails, the data is still available on other nodes of the cluster, and if a rack fails, the data remains intact, as well.

HDFS Architecture

The HDFS has a master and slave architecture, in which the master node controls and assigns jobs to all its slave nodes. The following terminologies are used to describe the master and slave nodes:

The master nodes in the HDFS are as follows:

- NameNode
- Secondary NameNode

The slave nodes in the HDFS are as follows:

- Data nodes

These nodes represent the core serving roles in the HDFS architecture. Let us now explore in detail the roles of each node to better understand them.

NameNode	NameNode is an HDFS daemon that controls all the data nodes and handles

	all the file system operations, such as creating a directory, creating a file, or reading and writing a file. The NameNode is responsible for managing the file system namespace image. It holds the image in memory, representing how the file system looks. Additionally, it maintains the metadata for all the blocks of files in the file system and tracks the replication value, so it knows the locations of blocks stored on data nodes within the cluster. However, the metadata is not stored onto the disk and, each time, gets recreated when it starts. NameNode stores all this information persistently on the local disk in the form of a namespace image and edit log. The NameNode is the *single point of failure* in the Hadoop cluster. If the NameNode fails, the entire cluster fails.
Data Nodes	Data nodes are the slave machines controlled by the NameNode that perform all the block operations. Data nodes store and retrieve blocks when asked to do so by the NameNode, and they periodically inform the NameNode with the lists of blocks they store by sending heartbeats. Data nodes replicate the data physically when instructed by the NameNode regarding where and how to replicate.
Secondary NameNode	The secondary NameNode, as its name implies, is not exactly the secondary NameNode. The secondary NameNode

is not a high-availability solution and does not automatically adopt the responsibilities of NameNode on failure. Its primary role is to create a checkpoint and back up the NameNode periodically; thus, it is like a backup solution to the NameNode. The hardware specifications of the secondary NameNode should be similar to those of the NameNode. In the event of the NameNode's failure, the secondary NameNode can be manually configured to work as a primary NameNode; this is not a high-availability solution.

Now that we have been briefly introduction to Hadoop, let us shift our focus to the main topic of our discussion, Apache Spark.

BRIEF INTRODUCTION TO SPARK

What is Spark?

Apache Spark is an open-source, fast, and unified parallel large-scale data processing engine. It provides a framework for programming, allowing for the distributed processing of large data sets at high speeds. Spark supports the most popular programming languages such as Java, Python, Scala, and R. Spark is scalable, meaning that it can run on a single desktop machine or a laptop to a cluster of thousands of machines. Spark provides a set of inbuilt libraries that can be accessed to perform data analysis over a large data set. However, if the user's requirements are not satisfied by the inbuilt libraries, a library can be written, or the user can explore countless external libraries from open-source communities on the internet.

Why Spark?

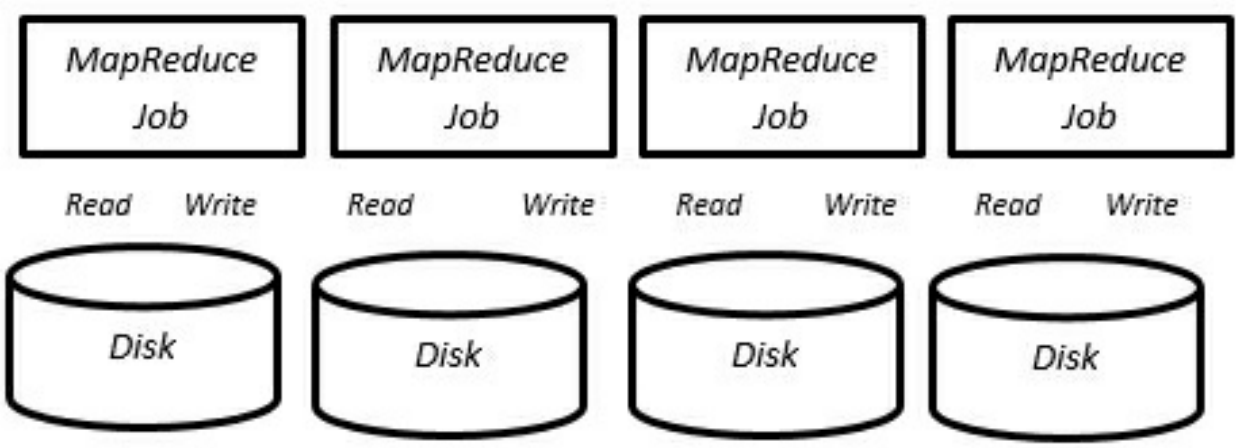
Why should one use Spark when Hadoop exists? Spark excels as a unified platform for processing large quantities of data at very high speeds for

various data processing requirements (discussed later in this chapter). Moreover, Spark is an in-memory processing framework. Spark is considered a successor of Apache Hadoop. Let us briefly discuss the advantages Spark offers over Hadoop.

Within the Hadoop ecosystem, there are various frameworks for data processing requirements. A developer would use the MapReduce framework to analyze data using any of the languages, such as Java, C++, Python, and so on, but a data warehouse engineer, who is an SQL expert, must learn any of the aforementioned programming languages. To overcome this problem, a new framework that runs on top of Hadoop, called "Hive," was introduced. This was likewise a problem for ETL processing, so "Pig" was introduced, as well. Similarly, tools such as "Giraph" and "Mahout" were developed for graphs processing and machine learning, respectively. Moreover, Hadoop is used only for batch processing, and there is no way to process data in real-time. Therefore, to achieve this, a new framework called "Storm" was integrated with Hadoop to work with streaming data. With so many frameworks, organizations had a difficult time maintaining all the frameworks and tracking issues with them. Fortunately, all this would change with the advent of Spark. As mentioned, Spark is a unified platform that provides all these frameworks as a single package containing four major components.

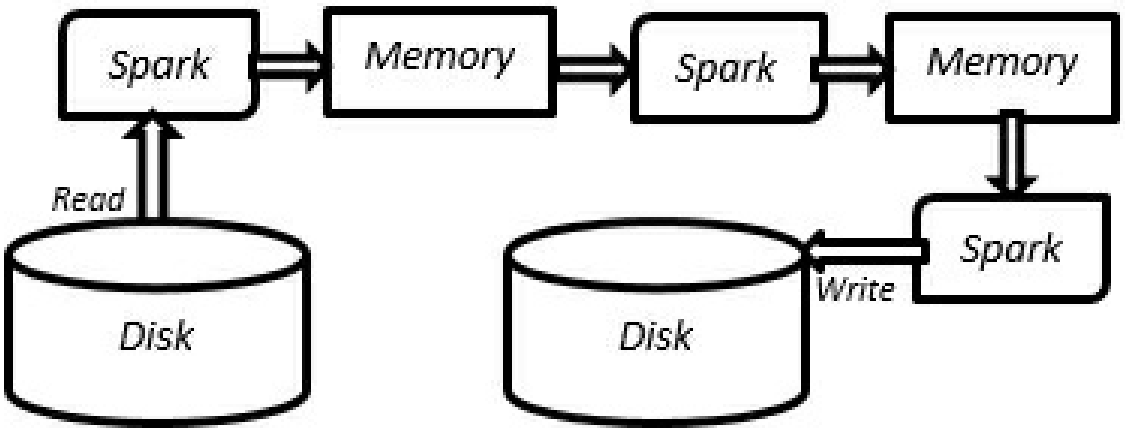
What does in-memory processing refer to? All the applications are processed in memory only, are they not? Indeed, all the applications are processed in memory and written back to the disk once the processing is completed, but Spark can process data in-memory and retain the data within the memory or write to the disk. Let us examine this with a figure that compares Spark with MapReduce.

1(a) Data Processing with MapReduce



In MapReduce, the data present in the HDFS or any other *distributed file system* is read by a MapReduce application and is processed in memory and then written back to the disk after the job is complete. If the processed data is needed again for further processing, the data is read from the disk by a MapReduce application, processed in memory, and then written back to the disk. This process continues per the requirements, as seen in Figure 1(a). The processes of reading and writing data from and to the disk increase the IO latency, so the overall job duration is increased. This is optimized in Spark, as presented in Figure 1(b).

1(b) Data Processing with Spark



In Spark, the data is read from the disk and processed in-memory, but instead of spilling it back to the disk, Spark can retain the data within the memory for further processing. Therefore, if the processed data is again

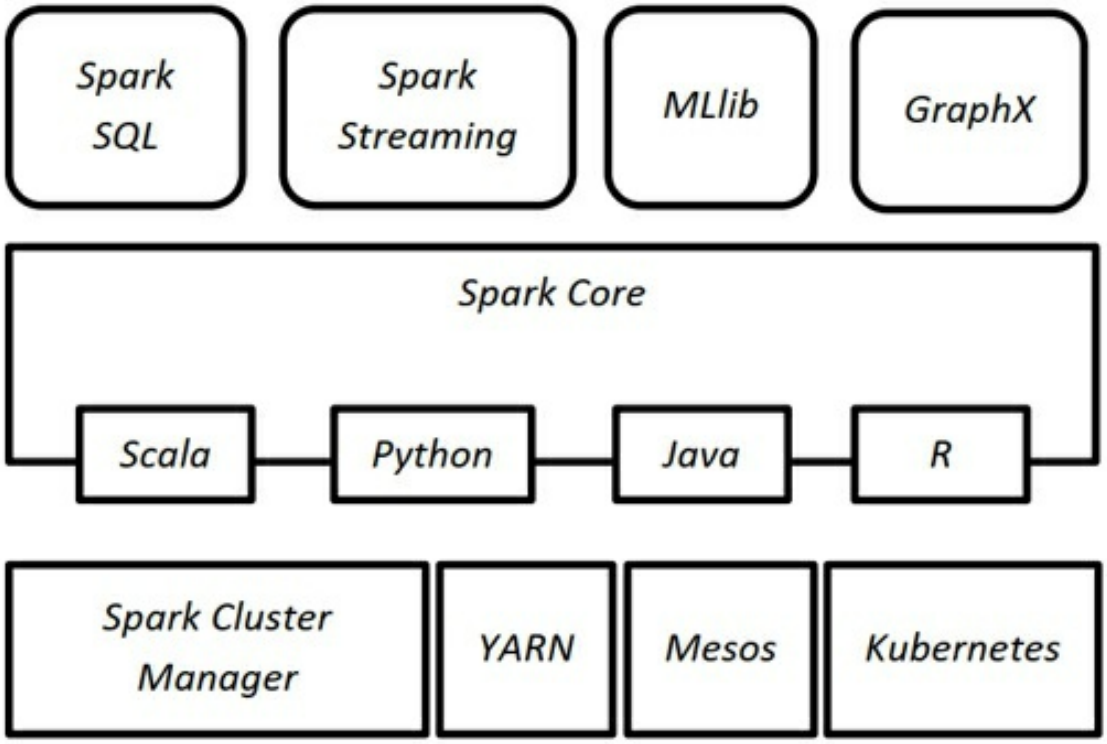
required for further processing, the data is already present in the memory, and the Spark application processes the data eliminating the IO latency, and so the overall time to process the job is significantly reduced. Through this, the processing speed compared with MapReduce is increased up to 100 times. The processed data from a Spark application can either be retained in memory or stored on the disk per the requirements, illustrated in Figure 1(b).

The reasons, such as a unified platform for various data processing requirements and high-speed in-memory processing, have gained worldwide popularity throughout the industry, with nearly all major organizations using Spark for their data processing requirements.

Components of Spark

Now that we understand why Spark is used, let us further investigate and learn the inner workings of Spark. Let us consider each of Spark's major components individually and learn about them in detail. Figure 1(c) presents the components of Spark.

1(c) Components of Spark



Let us consider a brief explanation so that we can better understand the Spark components.

Spark Core	Spark Core, as the name suggests, is the core component of Spark and contains all the basic functionalities for processing large data sets. Some of its functionalities include managing memory, scheduling jobs, providing fault tolerance, using in-memory computation, referring data sets stored in storage systems, and so on. Spark Core includes a programming abstraction (API) called resilient distributed data sets (RDDs), which is responsible for partitioning data across nodes on a cluster. With the help of these RDDs, the data can be transformed, collected, and reduced together. These RDD APIs can be referred to using any of the

programming languages, such as Scala, Python, Java, and R, as depicted in Figure 1(c).

Spark SQL The Spark SQL component provides the developer with an SQL-like interface to work with large structured data, which is distributed over a cluster of nodes. Spark SQL works well with structured and semi-structured data. Moreover, Spark SQL can work with data sources such as Apache Hive tables, Avro, JDBC, ORC, JSON, and Parquet file formats. In addition, Spark SQL allows developers to combine RDD APIs along with Spark SQL code in a single application.

Spark Streaming The Spark Streaming component of Spark concerns the processing of real-time data, known as streaming data. Streaming data can result from a fleet of web servers, sensors, IOT devices, or any other sources that generate data. This enables Spark to ingest data as it is generated in real-time and perform data manipulation on that data. There are three major phases of Spark Streaming: *gathering*, *processing,* and *data storage*. Moreover, Spark Streaming is fault-tolerant and scalable. Spark Streaming is discussed a little in this book.

Spark MLlib Spark MLlib is short for machine learning libraries, which provides machine learning for large data sets. MLlib contains various machine learning algorithms, such as *regression*, *clustering*, *classification,* and *collaborative filtering*. Furthermore, MLlib contains lower-level primitives, such as a generic gradient descent optimization

	algorithm. Additionally, MLlib uses the linear algebra package *Breeze* for numerical computing.
GraphX	GraphX concerns the efficient and distributed processing of graphs. GraphX extends the RDD APIs, which allows a developer to create a directed multigraph using properties attached to each vertex and edge.
Cluster Managers	Spark involves processing massive amounts of data sets by distributing them over a number of nodes and scaling the cluster as required. To efficiently perform this task, a cluster manager is required; Spark offers its own cluster manager, called *Standalone Scheduler*. Moreover, Spark can be deployed using *Hadoop YARN, Apache Mesos,* or *Kubernetes* as a cluster manager to schedule jobs and manage the cluster's resources.

Spark Data Storage

Spark supports major file systems such as HDFS, Amazon S3, Azure Blob, and so on. Moreover, Spark supports the local file system for storing the data, as well. However, using a distributed file system, such as the HDFS, can leverage the power of Spark by distributing the data sets throughout the cluster. In addition, Spark is capable of handling various file formats, such as text, ORC, parquet, and so on.

Hadoop and Spark are used to analyze large amounts of data; however, this solves only one of the challenges faced with BigData. The other challenge is collecting large amounts of data efficiently; Kafka helps us with this. Let us now proceed with an introduction of Kafka to obtain an understanding of how it addresses this challenge.

INTRODUCTION TO KAFKA

What is Kafka?

Kafka is an open-source, distributed, persistent, and fault-tolerant message-streaming platform or central repository that can handle a high volume (trillions) of Publish-Subscribe messages each day. The Publish-Subscribe messaging system is a system in which data is produced (publish) by producers and consumed (subscribe) by consumers. Producers and consumers are described in detail in the following chapter.

Kafka is written in Scala and built on top of the ZooKeeper coordination service. The integration of Spark and Kafka enables real-time streaming data analysis. Kafka was built at LinkedIn and was later donated to the Apache Software Foundation, making it open-source.

Kafka is popular due to the following features:

- **Scalable**: Kafka can be scaled from a single machine to a cluster of machines spanning data centers with zero downtime. The number of machines required can be scaled per the requirement.
- **Persistent**: The data or messages are stored and cached in the disk rather than memory, making them persistent and durable. Moreover, Kafka is fault-tolerant with replications and partitions.
- **Performance:** Kafka provides great performance and stability with large volumes of publishing and subscribing messages.
- **Distributed**: Kafka is distributed to a cluster of machines and hence processes streams with great speed and efficiency.
- **Real-Time Streaming:** Kafka is capable of processing streams of messages in real-time.

Kafka is used to develop real-time streaming data pipelines for the steady transfer of data between applications. The use cases include collecting logs from multiple servers, data from sensors, and so on.

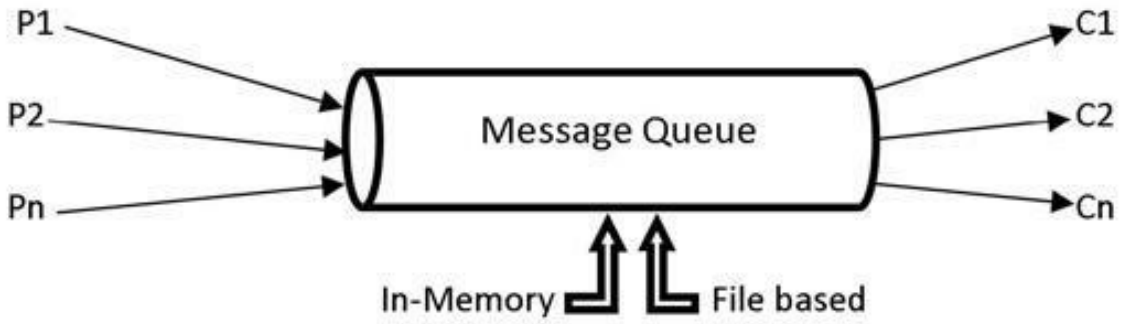
Why Kafka?

Why do we need Kafka? We already have message queue and data

streaming platforms, do we not? How is Kafka better than the traditional data streaming platforms? Let us answer these questions now. To understand why Kafka is necessary, we should first understand how traditional streaming platforms work.

Consider a traditional system with a message queue, as presented below. The message queue could be implemented in any programming language. The message queue receives messages from various processes, denoted P1, P2…Pn. This message queue may store data in-memory or on the file system based on the implementation. If the system is memory-based, the data will be lost in the event of system failure. However, if the system is file-based, the data will remain intact even if the system goes down. The data from the message queue will be consumed by various consumer processes, denoted by C1, C2…Cn.

The data is produced by the producers, and it is consumed by the consumers via the message queue. However, the problem arises when one of the consumers is connected to a distributed platform, such as the Spark application. Spark is capable of processing large amounts of data in a distributed manner; however, the message queue is not distributed, and it is implemented in a single machine. Therefore, the traditional message queue is limited by the resources of that machine, such as the CPU core, RAM, and disk size, and becomes the bottleneck, as it cannot receive large amounts of data similar to Spark.



1(d) Traditional Message Queue

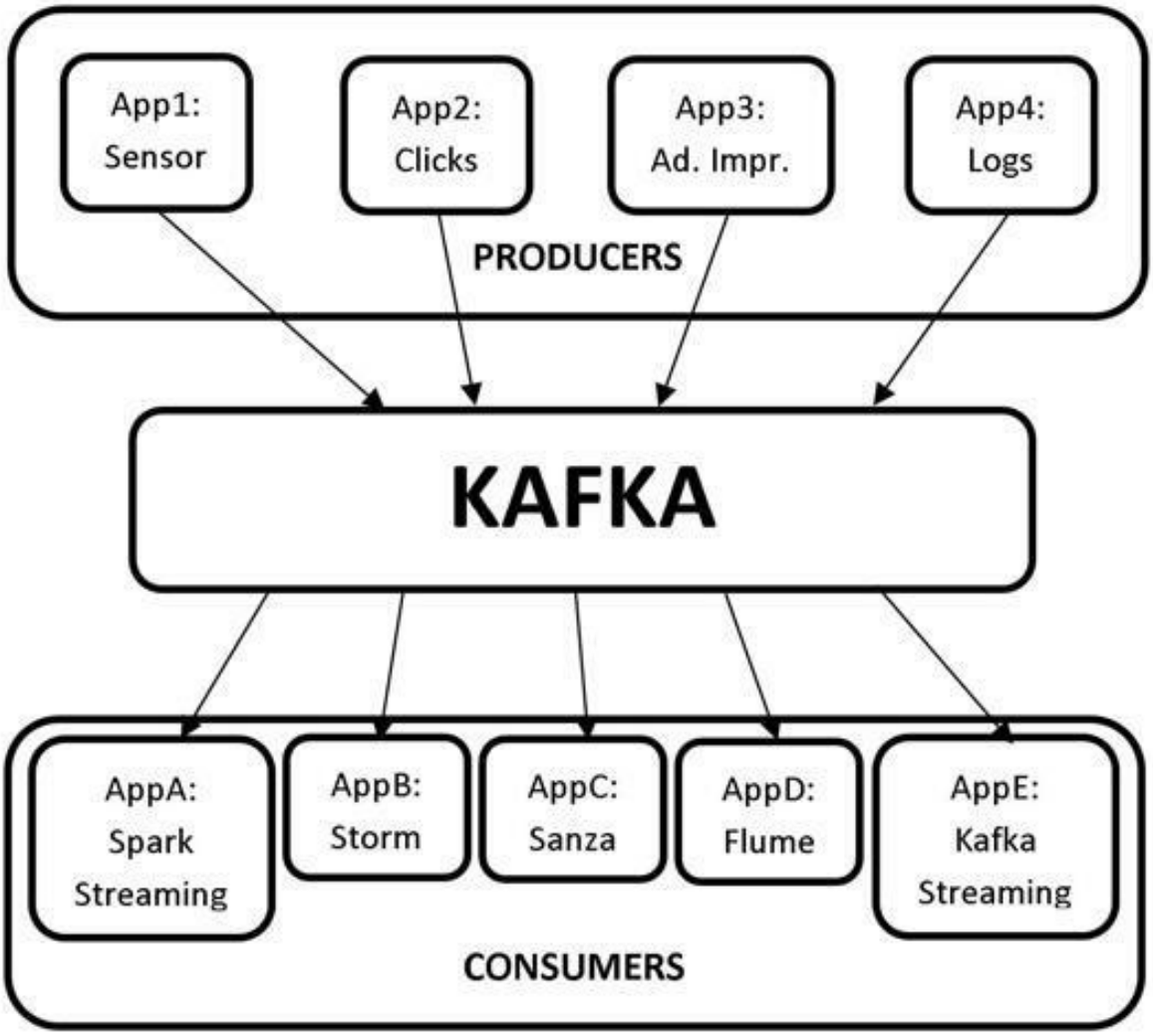
To cater to a distributed processing application such as Spark, it is efficient

only to have a message queue distributed across several machines. The limitation of machine resources, such as CPU, RAM, and disk size, was one of the reasons for developing Kafka, which is a distributed message queue.

Let us now consider another reason why Kafka was developed. Consider an application *app1* that generates data during its operations. This data is required by an analytics application, say *appA*. The data is simply transferred via an interface from *app1* to *appA*. Later, there is another application, say *app2*, that generates data. This data is likewise required by the analytics application *appA* for more analytics. Thus, an interface must be implemented to send the data from *app2* to *appA*. Once the interface for *app2* is implemented and tested, the interface for *app1* should also be tested to ensure the new implementation has not broken something with the old implementation.

Eventually, more operational (*app3, app4, app5…*) and analytics (*appB, appC, appD…*) applications were developed that require data from the operational to analytics applications. For example, data from *app4* might be needed by *appB* and *appD*, or data from *app3* might be required by *appA* and *appC.* Many such possibilities may exist for transferring data from one or more operational applications to one or more analytics applications. Each time a new operational application generates data for an analytics application, a new interface must be implemented, and all the interfaces should be tested to determine whether something is broken due to the new implementations. This becomes highly difficult to maintain when there are so many applications and interfaces. The entire system should be tested each time a new interface between applications is implemented.

All these problems have led to the development of Kafka. Instead of having different interfaces for different applications, all the operational applications send the data to Kafka. The analytics applications can then consume the data from Kafka, making it a central repository. The figure below illustrates how data is being produced and consumed with Kafka as a central repository.



1(e) Kafka Message Queue

As seen in the figure above, we do not need to build interfaces and test them to transfer data each time a new application is implemented. All the producers generate the data to Kafka, while the consumers pull the data from Kafka, making it a central repository.

CONFLUENT OVERVIEW

Confluent is a data streaming platform based on Apache Kafka and is founded by the creators of Apache Kafka. Confluent expands the capabilities of Apache Kafka for not only Publish-Subscribe messages but also a full-scale event streaming platform that allows for storing and processing real-time streams. The Confluent data streaming platform consists of Apache Kafka as

its core component.

The Confluent data streaming platform provides the following components, making it a complete distribution of Apache Kafka:

- **Apache Kafka:** Apache Kafka is the core component of the Confluent platform. Apache Kafka is an open-source, distributed, persistent, and fault-tolerant message-streaming platform or central repository that can handle high volumes (trillions) of Publish-Subscribe messages each day.

  However, Apache Kafka is not a complete data streaming platform. It provides data storage and interfaces only for reading and writing data. It does not directly integrate with other services, such as RDBMS. With Confluent's other components, the capabilities of Kafka can be extended such that it can integrate with other services.

- **Kafka Connect:** Kafka Connect is used to transfer data to and from Kafka. HDFS, JDBC, S3, Elasticsearch, and so on are some Kafka connectors that transfer data to and from Kafka.

- **Kafka REST Proxy:** The Kafka REST proxy provides a RESTful interface to a Kafka cluster. The Kafka REST proxy can be used to send and receive messages, view the state of the cluster, and perform administrative actions.

- **Kafka Streams:** Kafka Streams is a powerful yet easy-to-use client library for stream processing and analysis. With the Kafka Streams processing layer, we can perform transformations or analysis by reading the real-time data and writing the results back to Kafka.

- **Schema Registry:** Schema Registry is a serving layer for metadata. Schema Registry provides a RESTful interface for storing and retrieving AVRO schemas. It ensures that the data being sent and received is in a common format (*i.e.*, checking schema compatibility for Kafka). This is described in greater detail in the upcoming chapters.

- **KSQL:** KSQL is a streaming SQL engine for Kafka used to run queries on data stored in the Kafka cluster. KSQL is used internally on Kafka streams for processing.

We focus only on Apache Kafka throughout this book. However, let us consider a use case to better understand how all these components are integrated to form an end-to-end pipeline using the Confluent data streaming platform.

KAFKA USE CASE

Let us now consider a fraud detection use case at a high level to better understand how all the components of Confluent can be integrated to form an end-to-end pipeline. We shall observe a use case of fraud detection in credit card transactions.

The use of credit cards has been on the rise in the past few decades, in addition to instances of fraud regarding cards. People make transactions using credit cards, and this transaction data can be analyzed to prevent fraud in real-time and minimize monetary loss.

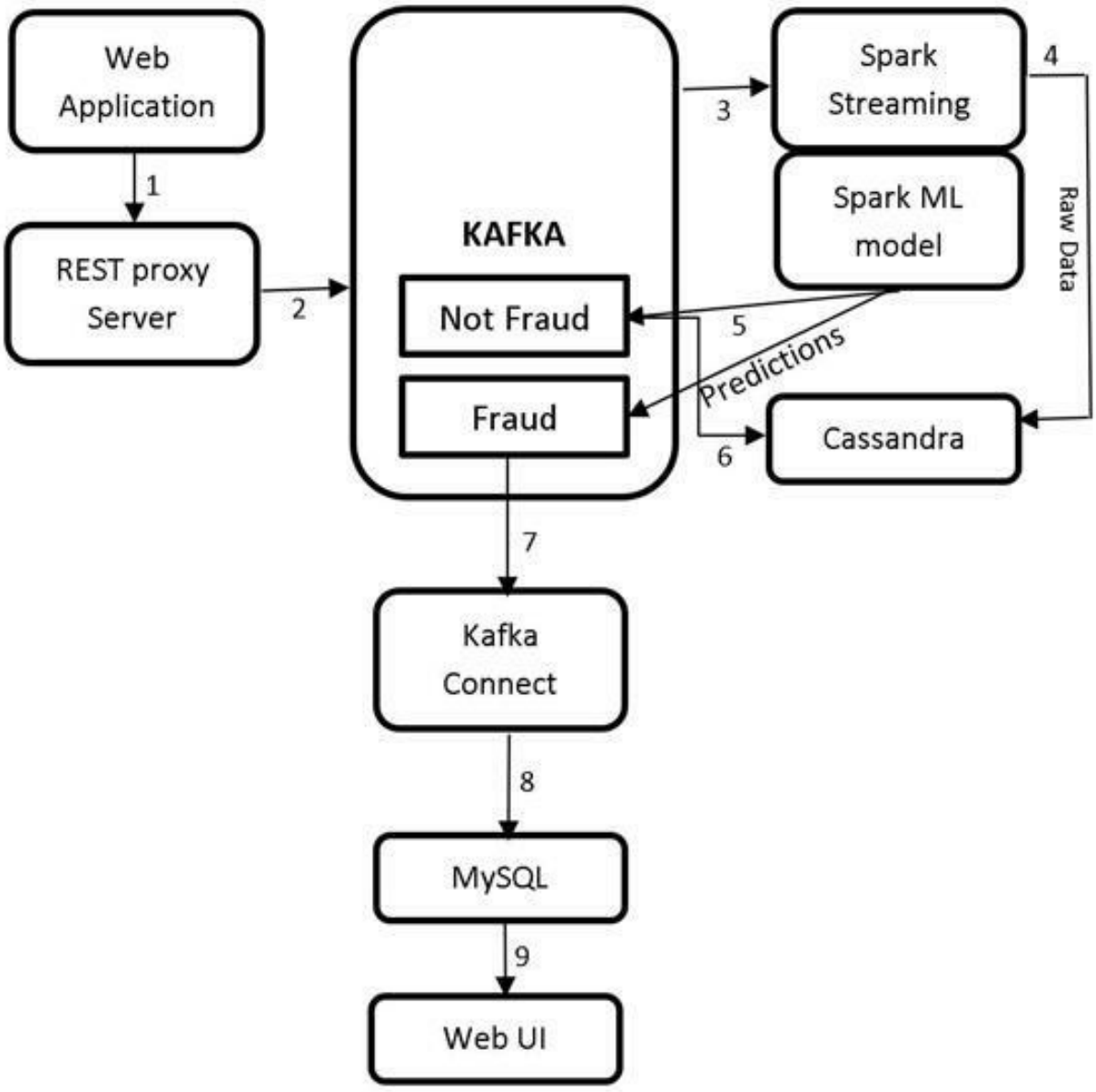
1. Let us consider that all credit card transactions are recorded in a web application.

2. These transactions are then transferred to the Kafka message queue using the REST proxy server. The REST proxy server serves as an interface to Kafka.

3. The transactions are then pulled from Kafka by Kafka Streaming or Spark Streaming to apply transformations or analysis. *Please note that Spark Streaming is generally used to process real-time messages similar to credit card transactions, since we feed the processed data to a machine learning model.*

After processing the streams, the data is internally fed to a Spark machine learning model to predict whether the transaction is fraudulent. Confluent Kafka does not provide a machine learning module, so Spark MLlib is used for predictions.

4. The unpredicted raw data from Spark Streaming is saved to Cassandra for further processing, if required.



1(f) Kafka Use Case

5. Two outcomes are possible for these predictions: *i.e.*genuine transactions and potentially fraudulent transactions. This data is sent to Kafka.

6. In general, there will be more genuine or non-fraud transactions than fraudulent transactions. Hence, all the genuine transactions

are sent to a NoSQL database, such as Cassandra.

7. The potential fraud transactions are then transferred to an RDBMS database, such as MySQL, via Kafka Connect.

8. These records can then be pulled from MySQL and displayed on webUI.

9. Domain experts can then take the necessary actions to determine whether these predicted potentially fraudulent transactions are actually fraudulent. If it is determined that a transaction is fraudulent, the card issuer can block or hold the card from making transactions to prevent further loss.

This is how the Confluent Kafka components can be used to build an end-to-end pipeline.

Before concluding this chapter, let us consider ZooKeeper and why it is needed by Kafka.

INTRODUCTION TO ZOOKEEPER

What is ZooKeeper?

ZooKeeper is an open-source, robust distributed coordination service for distributed applications. In particular, ZooKeeper is an open-source Apache Software Foundation project that is available for free and ready to use. ZooKeeper helps to overcome many of the common challenges faced by distributed applications. ZooKeeper can be used for synchronization, sequential consistency, and coordination between distributed applications. It helps to maintain the configuration information that can be shared to all the nodes in a distributed system. Moreover, ZooKeeper helps in group services, such as leader election and many more. In addition, ZooKeeper is reliable and fast, yet simple to work with. With ZooKeeper, one can build reliable, distributed data structures for group membership, leader election, coordinated workflow, and configuration services. Furthermore, one can build generalized distributed data structures, such as locks, queues, barriers, and latches.

ZooKeeper provides an eventually consistent view of its *znodes*, which are nothing but files or directories in a file system. Moreover, ZooKeeper provides basic CRUD operations, such as creating, updating, and deleting *znodes*. It provides an event-driven model in which clients can watch for changes to specific znodes, such as if a new child is added to an existing *znode*. ZooKeeper is a high-availability service, as it consists of a set of ZooKeeper servers known as the *ensemble* (cluster), with each of the servers holding an in-memory image of the distributed file system to serve client read requests; in addition, each server holds a persistent copy on disk.

One of the servers in the *ensemble* is dynamically selected by consensus to be the leader, and all other servers are the followers. The leader is responsible for all writes and updating the changes to its followers. When the majority of followers update a change successfully, the write succeeds, and the data is still available even if the leader fails. When a leader fails, a new leader is again dynamically selected by consensus within the *ensemble*. This eliminates the single point of failure scenario, and the *ensemble* continues working as it should.

When a client connects to ZooKeeper, it is provided with the list of servers in the *ensemble*. The client connects to one of the servers in the ensemble at random until a connection is established. Once connected, ZooKeeper creates a session with a timeout period pre-specified by the client. The ZooKeeper client automatically sends heartbeats periodically to keep the session alive if no operations are performed for a time, and it automatically handles failover. If the connection between ZooKeeper and the client fails, the client automatically detects this and tries again to connect to a different server in the *ensemble*. After it is reconnected, the same client session is retained while the failure has occurred.

ZooKeeper Data Consistency

ZooKeeper provides the following guaranteed consistencies:

Sequential consistency: Updates from a client to the ZooKeeper service are applied in the order they are sent. Since all writes go through the leader, the global order is simply the order in which the leader receives the write requests.

Single System Image: The Single System Image guarantees that a client will see the same view of the ZooKeeper service no matter the server in the ensemble to which it is connected.

Atomicity: There are no partial failures. The updates from a client to the ZooKeeper service either succeed or fail. For example, assume a client sends an update to a server, but before the response is received, the network connection is lost or the server goes down. Now, did the update get through to the server? If yes, did the operation complete successfully? The only way to know the answers to these questions is when the server/network is back up again. ZooKeepe, however, cannot help with network problems or partial failures; rather, it handles through atomicity. If the network/server goes down during an update operation, the operation is marked as failed; otherwise, it is marked as a success.

Reliability: If the update is successful, it is persistent and will not be rolled back. The update will be overwritten only when the client makes a new update. The updates are still available even when the server fails.

Consistent Client View: A client's view of the system is guaranteed to be up-to-date within a certain time bound, generally within tens of seconds. If a client does not observe system changes within that time bound, the client assumes a service outage has occurred and will connect to a different server in the *ensemble*.

ZooKeeper Architecture

The ZooKeeper architecture consists of a leader and follower servers. The collection of these servers is known as the *ensemble.* The number of servers in a ZooKeeper ensemble should always be an odd number. The reason for this is because we need a majority during the voting process of electing a leader. Let us now consider the responsibilities of the leader and follower servers.

- **Leader**: When an *ensemble* is first started, a voting process is held to select a leader. During the voting process, a leader is elected, and the process is completed as soon as a simple majority of followers have synchronized their state with the leader. After the leader election is complete, the leader is responsible for handling all the write requests from clients, and changes are committed to all followers. Once a majority of followers have persisted with the change, the leader commits the change and notifies the client of a successful update. There should always be a leader; if the leader is down, all the existing followers vote for and elect a new leader.

- **Followers**: The followers' function is similar to that of the leader, allowing clients to connect to them and send, read, and write requests to them; here, however, the writes are forwarded to the leader.

- **Observers:** Observers are the non-voting members of an *ensemble*, which do not participate in the voting process but only hear the voting results. When more followers participate in voting, the write performance drops significantly, so observers are added to the *ensemble* to address this issue. Observers improve ZooKeeper's scalability, and we can increase the number of observers as much as we like without harming the performance of votes. The observers' function is the same as the followers, in which clients connect to them and send, read, and write requests to them. Observers forward these requests to the leader, akin to followers, but they then simply wait to hear the result of the vote.

WHY DOES KAFKA NEED ZOOKEEPER?

Kafka cannot be started without ZooKeeper. We must first start the ZooKeeper service before starting Kafka. The ZooKeeper service will run on a separate server rather than on the servers running Kafka Brokers. Based on the explanation of Zookeeper in the sections above, Kafka needs ZooKeeper for the following purposes:

- **Electing a Controller:** Kafka consists of brokers to handle requests, such as sending and receiving messages. The broker acts as a mediator for both producers and consumers to handle the requests. The broker is a Kafka server, and multiple brokers form a Kafka cluster. Since a Kafka cluster contains multiple brokers, we must elect a leader among these brokers to maintain the cluster state. The broker that we elect as the leader is called the controller and is responsible for maintaining the leader-follower relationships. In the event of the failure of a broker, the controller's responsibility is to instruct all the replicas to act as partition leaders to fulfill the duties of the partition leaders on the broker that is about to fail. ZooKeeper is used to elect this controller, ensures there is only one leader, and elects a new leader in the event of failure.

- **High Availability:** ZooKeeper tracks the membership of all the brokers and periodically checks whether any of the brokers that are part of the cluster have failed. It maintains a high availability of the controller broker.

- **Topic Configuration:** ZooKeeper tracks existing topics, partitions for each topic, replica locations, the preferred leader, and configuration override information for each topic.

- **Access Control Lists:** ZooKeeper maintains access control lists for each topic (*i.e.*, the read and write permissions of clients).

Do not worry if the new concepts are difficult to understand at this time. We shall explore brokers, replications, partitions, topics, and so on in greater detail in the next chapter. It will become clear why we need ZooKeeper once we better understand the architecture of Kafka.
This concludes the theory for this chapter.

LAB EXERCISE 1

"There are no activities required for this lab"

SUMMARY

Kafka is an open-source, distributed, persistent, and fault-tolerant message-streaming platform or central repository that can handle high volumes (trillions) of Publish-Subscribe messages each day. A Publish-Subscribe messaging system is a system in which data is produced (publish) by producers and consumed (subscribe) by consumers.

Kafka is written in Scala and built on top of the ZooKeeper coordination service. The integration of Spark and Kafka allows for real-time streaming data analysis. Kafka was built at LinkedIn and later donated to the Apache Software Foundation, making it open-source.

ZooKeeper is an open-source, robust distributed coordination service for distributed applications. In particular, ZooKeeper is an open-source Apache Software Foundation project that is available for free and ready to use. ZooKeeper helps to overcome many of the common challenges faced by distributed applications. Moreover, ZooKeeper can be used for synchronization, sequential consistency, and coordination between distributed applications. It helps to maintain the configuration information that can be shared to all the nodes in a distributed system.

CHAPTER 2:
KAFKA FRAMEWORK

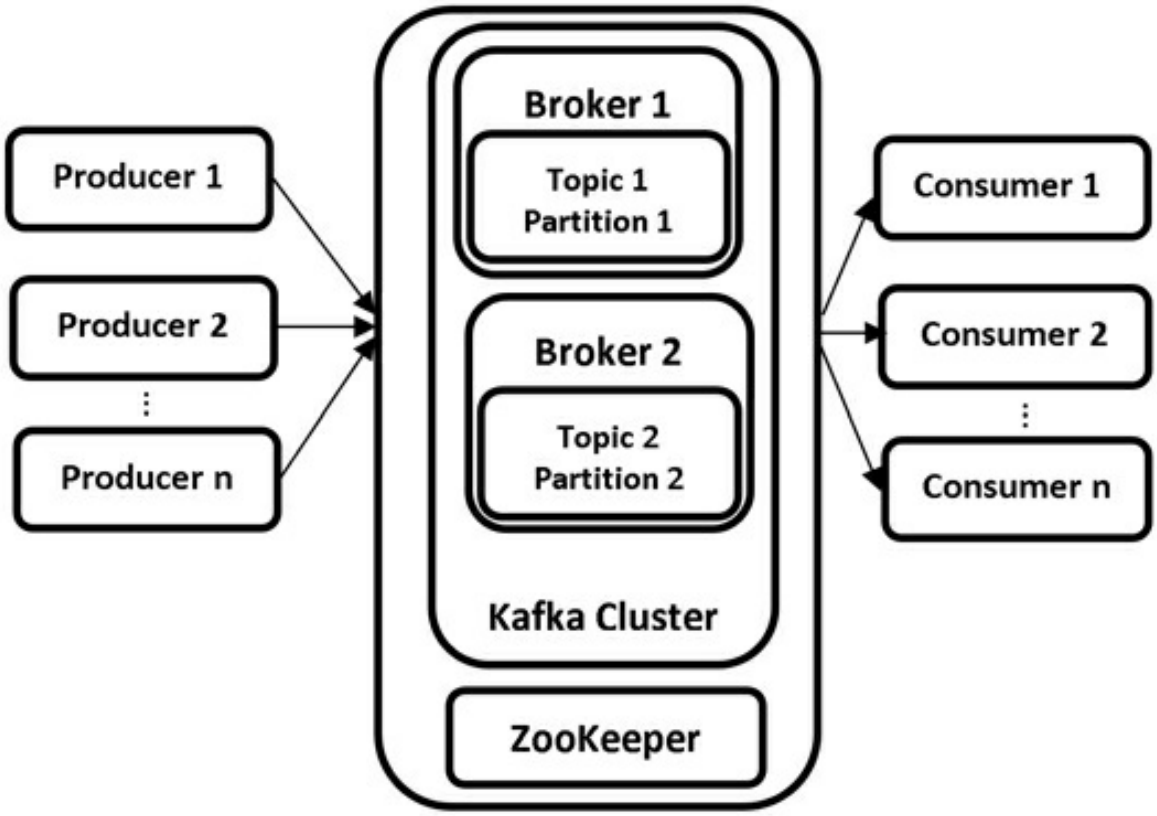
THEORY

The previous chapter provides an overview of BigData through a brief introduction to the Apache Hadoop and Apache Spark frameworks. Moreover, it presents a comprehensive introduction to Apache Kafka and Apache ZooKeeper. Let us now learn more about Apache Kafka, starting with the architecture, APIs, and CLI of Apache Kafka before proceeding to the labs to install Apache ZooKeeper and Apache Kafka on our machines.

KAFKA ARCHITECTURE

We have considered a use case of Kafka in the previous chapter, presenting a high-level representation of the Kafka architecture. Let us now explore the Hive architecture in detail.

The Kafka architecture is presented in the figure below.



2(a) Kafka Architecture

The Kafka architecture consists of the following components:

- Brokers
- Producers
- Consumers
- Topics
- Partitions
- Replications
- ZooKeeper

Let us now explore an overview of each component.

Topics

Kafka is capable of streaming messages from multiple sources. A topic is a

name provided to each source to distinguish and store the incoming messages. Topics are used to group similar messages. Producers write messages to topics, and consumers read data from topics with the help of brokers. A new directory is created with the name of the topic's partition whenever a new topic is created. A leader broker is responsible for all the read and write operations for that partition.

Partitions

Kafka topics are further divided into partitions to achieve scalability. A topic can have multiple partitions. A partition is a log to which the messages are appended as received. The messages are distributed across multiple brokers with the help of partitions, thus making Kafka a distributed message queue. This allows multiple consumers to read data from a topic in parallel.

Producers

Producers are the Kafka clients that create and send new messages to the brokers. Producers create messages for one or more Kafka topics and are the source of data for a Kafka cluster; there can be multiple producers in a Kafka cluster. Each message produced is assigned an offset by the broker. An offset is the integer metadata associated with each message, and it increases monotonically for each message.

Consumers

Consumers are the Kafka clients that read data from brokers, and there can be multiple consumers in a Kafka cluster. Consumers read the data from topics to which they are subscribed. The data is read in the order it was produced. Consumers know which messages have already been consumed with the help of offsets. Partitions have a unique offset for each message. A consumer simply stores the last offset it consumed in ZooKeeper or Kafka, which will help the consumer to continue consuming the messages from precisely where it stopped.

Brokers

A Kafka cluster consists of daemons known as brokers and consists of one or more brokers. Brokers are the most important components and workhorses of a Kafka cluster; without brokers, there is no Kafka. As the name suggests,

brokers act as middlemen between the producers and consumers. Brokers receive messages from producers, assign them with offsets, and store them on disk into Kafka topics. Similarly, they respond to consumers' data pull requests from Kafka topics.

In the production environment, there should be only one Broker per node in the Kafka cluster; however, in the testing environment, there can be multiple brokers on a single node. The brokers are further classified as the leader broker and controller broker.

Controller Broker

A controller broker is available in the pool of brokers within a Kafka cluster. A controller broker is elected with the help of ZooKeeper as soon as the brokers in a Kafka cluster are started, and it is considered the master broker. A controller broker is similar to any other broker in the Kafka cluster but has extra responsibilities. There is only one active controller broker at all times.

The controller broker is responsible for assigning partitions to brokers, monitoring brokers for failure, and rebalancing partitions to other brokers in the event of failure. Moreover, the controller broker is responsible for the duties of any other broker in the cluster (*i.e.*, leading partitions, performing reads/writes, and having partition replications).

Leader Broker

Each partition in a Kafka cluster has one broker that acts as a leader broker. If the partition is created with a replication factor greater than one, the partitions are replicated to the follower brokers. The leader broker manages the read/write requests for the partitions from producers and consumers, and the follower brokers simply replicate the partitions in the leader broker. In the event of failure of the leader broker, another broker that has the replications will adopt the leadership, making it redundant.

Replications

The partitions are replicated across the Kafka cluster to achieve high availability. This is done to ensure that the data is not lost in the event of a broker failure in the Kafka cluster. The replications are of two types: leader and follower replicas. The messages transmitted from the leader broker are

leader replicas, and the follower replicas are used for fault tolerance in the event of data loss available in the follower brokers.

ZooKeeper

As discussed in the previous chapter, ZooKeeper provides a coordination service for a Kafka cluster. Please review Why does Kafka need ZooKeeper? for more information.

We consider the internals of these concepts in detail in the upcoming chapters.

This concludes the theory for this chapter. Let us proceed to the lab exercise to install ZooKeeper and Kafka onto our machines.

AIM

The aim of the following lab exercises is to install and configure ZooKeeper and Kafka.

The labs for this chapter include the following exercises.

- Downloading and installing JDK
- Downloading and installing ZooKeeper
- Configuring ZooKeeper
- Downloading and installing Kafka
- Configuring Kafka
- Starting ZooKeeper and Kafka

We require the following packages to perform this lab exercise:

- Java Development Kit
- Apache ZooKeeper
- Apache Kafka

We will use an Ubuntu 18.04 LTS operating system with at least 4 GB of RAM throughout this book for all our exercises. Please ensure that you install this version of OS before proceeding with Kafka installation.

LAB EXERCISE 2: KAFKA INSTALLATION

1. **Download and install JDK**
2. **Download and install ZooKeeper**
3. **Configure ZooKeeper**
4. **Download and install Kafka**
5. **Configure Kafka**
6. **Start ZooKeeper and Kafka**

TASK 1: DOWNLOAD AND INSTALL JDK

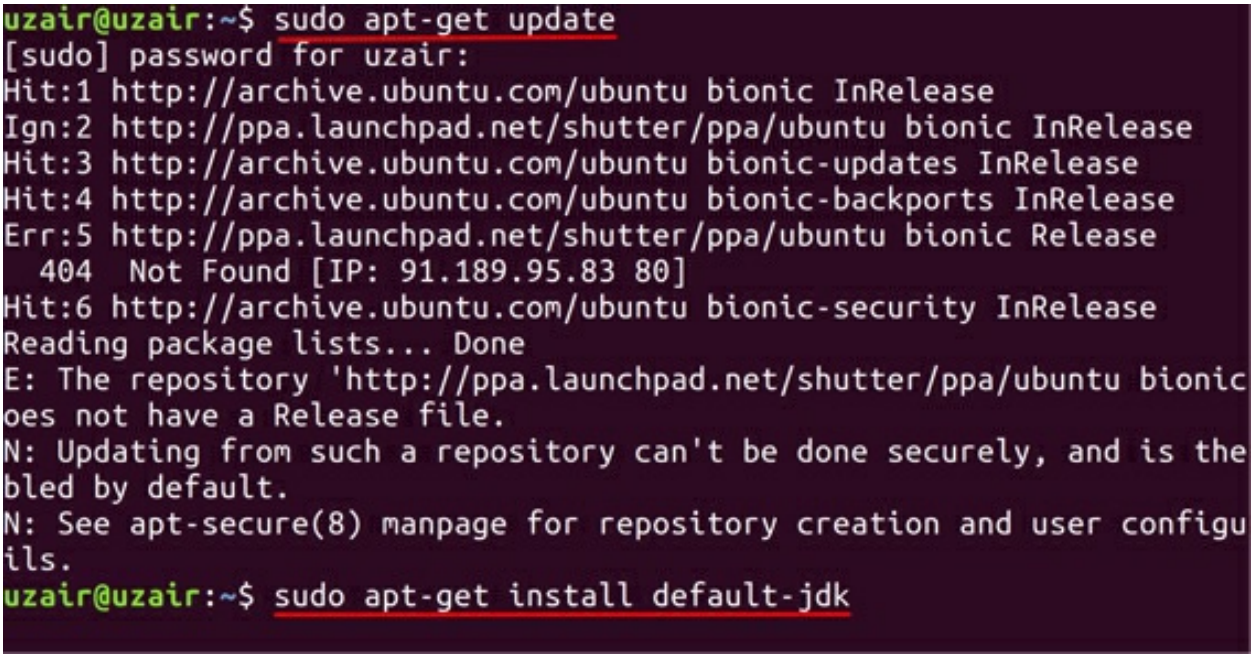
Step 1: From the terminal, run the following commands to install JDK (Java Development Kit):

```
$ sudo apt-get update
```

This will update the package index. You might be asked to enter your password after running the command above.

Step 2: Once you run the above command, run the following command to download and install JDK.

```
$ sudo apt-get install default-jdk
```



The prompt will ask you to press “Y” after running the above command, as illustrated in the screenshot. Press “Y” on your keyboard to continue with the installation, and finally press the Enter key. This will download and install JDK onto your machine.

```
The following NEW packages will be installed:
  ca-certificates-java default-jdk default-jdk-headless default-jre
  default-jre-headless fonts-dejavu-extra java-common libatk-wrapper-java
  libatk-wrapper-java-jni libgif7 libice-dev libpthread-stubs0-dev libsm-dev
  libx11-dev libx11-doc libxau-dev libxcb1-dev libxdmcp-dev libxt-dev
  openjdk-11-jdk openjdk-11-jdk-headless openjdk-11-jre
  openjdk-11-jre-headless x11proto-core-dev x11proto-dev xorg-sgml-doctools
  xtrans-dev
0 upgraded, 27 newly installed, 0 to remove and 0 not upgraded.
Need to get 236 MB of archives.
After this operation, 398 MB of additional disk space will be used.
Do you want to continue? [Y/n] Y
```

The installation process might take some time depending on your internet connection. Please allow it to download and install completely. You will be presented with the following message once the installation has been completed.

```
Setting up default-jre (2:1.11-68ubuntu1~18.04.1) ...
Setting up openjdk-11-jdk:amd64 (11.0.2+9-3ubuntu1~18.04.3) ...
update-alternatives: using /usr/lib/jvm/java-11-openjdk-amd64/bin/
ovide /usr/bin/jconsole (jconsole) in auto mode
Setting up default-jdk (2:1.11-68ubuntu1~18.04.1) ...
Processing triggers for libc-bin (2.27-3ubuntu1) ...
Processing triggers for ca-certificates (20180409) ...
Updating certificates in /etc/ssl/certs...
0 added, 0 removed; done.
Running hooks in /etc/ca-certificates/update.d...

done.
done.
```

Step 3: Run the following command to check whether Java has been installed successfully. The terminal should display the Java version, as presented in the screenshot.

```
$ java –version
```

```
uzair@uzair:~$ java -version
openjdk version "11.0.2" 2019-01-15
OpenJDK Runtime Environment (build 11.0.2+9-Ubuntu-3ubuntu118.04.3)
OpenJDK 64-Bit Server VM (build 11.0.2+9-Ubuntu-3ubuntu118.04.3, mixed mode, sha
ring)
```

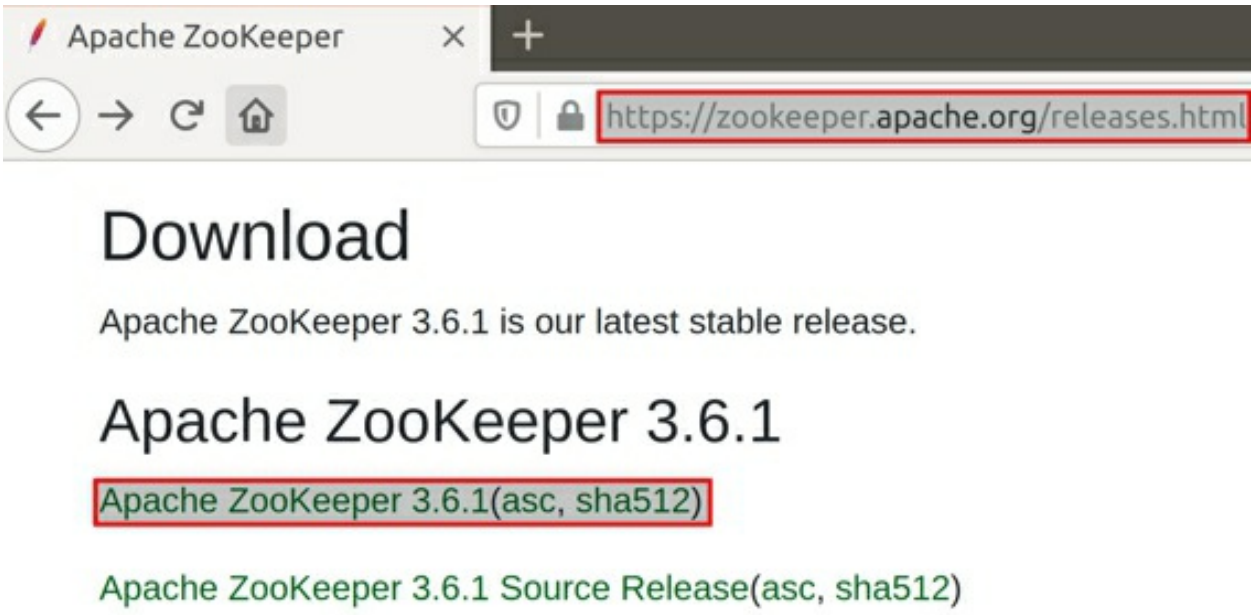
Please note that your version of JDK might be the latest version, as opposed to what is shown in the screenshot.

Task 1 is complete!

TASK 2: DOWNLOAD AND INSTALL ZOOKEEPER

Step 1: Let us ZooKeeper in standalone mode. Navigate to the download URL below and click to download the latest stable version for ZooKeeper (ZooKeeper 3.6.1 at the time of writing this book).

`Download URL:` https://zookeeper.apache.org/releases.html



After clicking the download link, you will be taken to a page with a mirror site to download ZooKeeper. Click the mirror link as illustrated below, and your download should start. The download might take some time depending upon your internet connection.

Step 2: The download will be saved to the *Downloads* directory by default. Execute the following command from your terminal to change the directory to the *Downloads* folder:

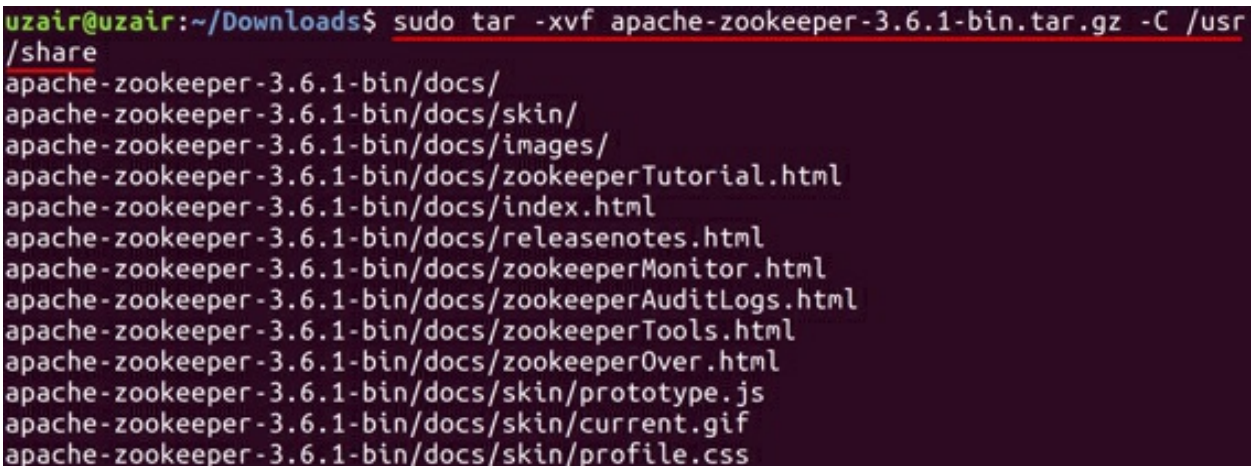
```
$ cd Downloads
```

Once you are in the *Downloads* directory, you may optionally check whether ZooKeeper has been downloaded using the *ls* command.

```
$ ls
```

Now that you are certain you have the ZooKeeper tar file, untar the ZooKeeper tar file to */usr/share* directory using the command below:

```
$ sudo tar –xvf apache-zookeeper-3.6.1-bin.tar.gz –C /usr/share
```



The file will begin to untar to the */usr/share* directory, as presented in the screenshot above. You can verify this by executing the command below:

```
$ cd /usr/share
```

```
$ ls
```

```
uzair@uzair:/usr/share$ cd /usr/share
uzair@uzair:/usr/share$ ls
accountsservice                 kde4
aclocal                         keyrings
acpi-support                    landscape
adduser                         language-selector
adium                           language-support
aisleriot                       language-tools
alsa                            libaudio2
alsa-base                       libc-bin
apache-zookeeper-3.6.1-bin      libdrm
appdata                         libexttextcat
```

As we can see from the screenshot above, the ZooKeeper directory is listed.

Step 3: Let us create a softlink to the ZooKeeper directory so that we do not need to refer to ZooKeeper with the entire name as above. This will be useful for future updates, as well. Execute the following command:

```
$ sudo ln –s apache-zookeeper-3.6.1-bin zookeeper
```

Run the following command again to check whether we could create the softlink successfully.

```
uzair@uzair:/usr/share$ sudo ln -s apache-zookeeper-3.6.1-bin zookeeper
uzair@uzair:/usr/share$ ls zookeeper
bin  conf  docs  lib  LICENSE.txt  NOTICE.txt  README.md  README_packaging.md
uzair@uzair:/usr/share$
```

Step 4: Let us conclude the installation process by creating the directory where ZooKeeper will store its data.

```
$ sudo mkdir zookeeper/data
```

```
$ ls zookeeper
```

```
uzair@uzair:/usr/share$ sudo mkdir zookeeper/data
uzair@uzair:/usr/share$ ls zookeeper
bin   data  lib          NOTICE.txt  README_packaging.md
conf  docs  LICENSE.txt  README.md
```

Step 5: Change the permissions of the ZooKeeper directory by running the

following command:

```
$ sudo chown <username> /usr/share/zookeeper
```

<username> - Insert your username.

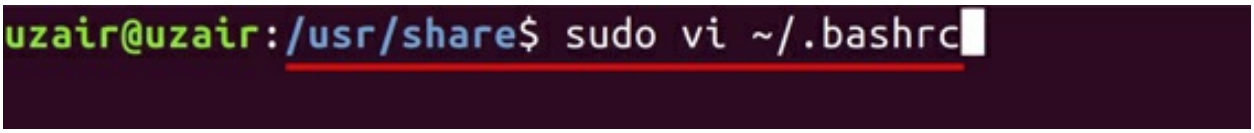
This completes the installation of ZooKeeper. Let us now proceed to the next task to configure ZooKeeper.

Task 2 is complete!

TASK 3: CONFIGURE ZOOKEEPER

Step 1: Let us now set up the environment variables for ZooKeeper. Begin by executing the following command:

```
$ sudo vi ~/.bashrc
```

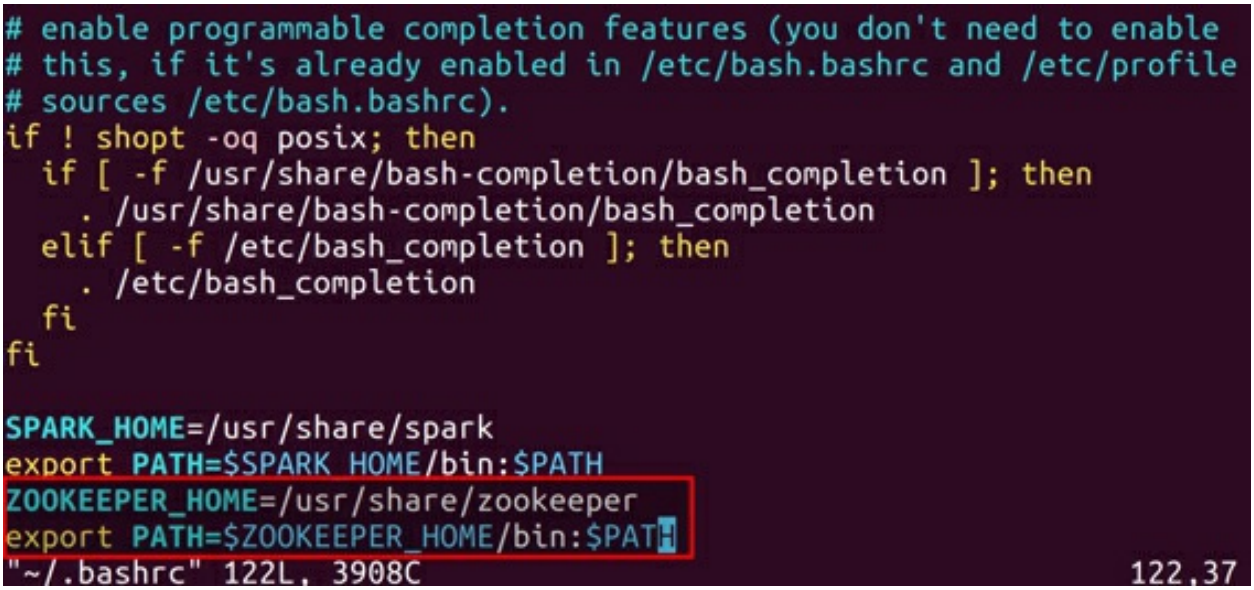


The file should open as presented below.

```
# ~/.bashrc: executed by bash(1) for non-login shells.
# see /usr/share/doc/bash/examples/startup-files (in the package bash-doc)
# for examples

# If not running interactively, don't do anything
case $- in
    *i*) ;;
      *) return;;
esac

# don't put duplicate lines or lines starting with space in the history.
# See bash(1) for more options
HISTCONTROL=ignoreboth

# append to the history file, don't overwrite it
shopt -s histappend

# for setting history length see HISTSIZE and HISTFILESIZE in bash(1)
HISTSIZE=1000
HISTFILESIZE=2000

# check the window size after each command and, if necessary,
# update the values of LINES and COLUMNS.
```

Now, press the *i* key to edit the file and append the following environment variable at the end of the file:

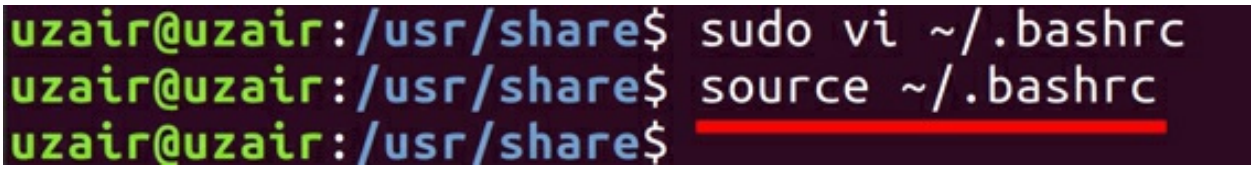
```
ZOOKEEPER_HOME=/usr/share/zookeeper
Export PATH=$ZOOKEEPER_HOME/bin:$PATH
```



After you have finished appending the text above, press the *Esc* key on your keyboard to stop editing, and then press *Shift - Z - Z* to exit the editor by saving the changes (please note that you need to press Z twice while holding the Shift key).

Now, reload the modified *.bashrc* file using the following command:

```
$ source ~/.bashrc
```

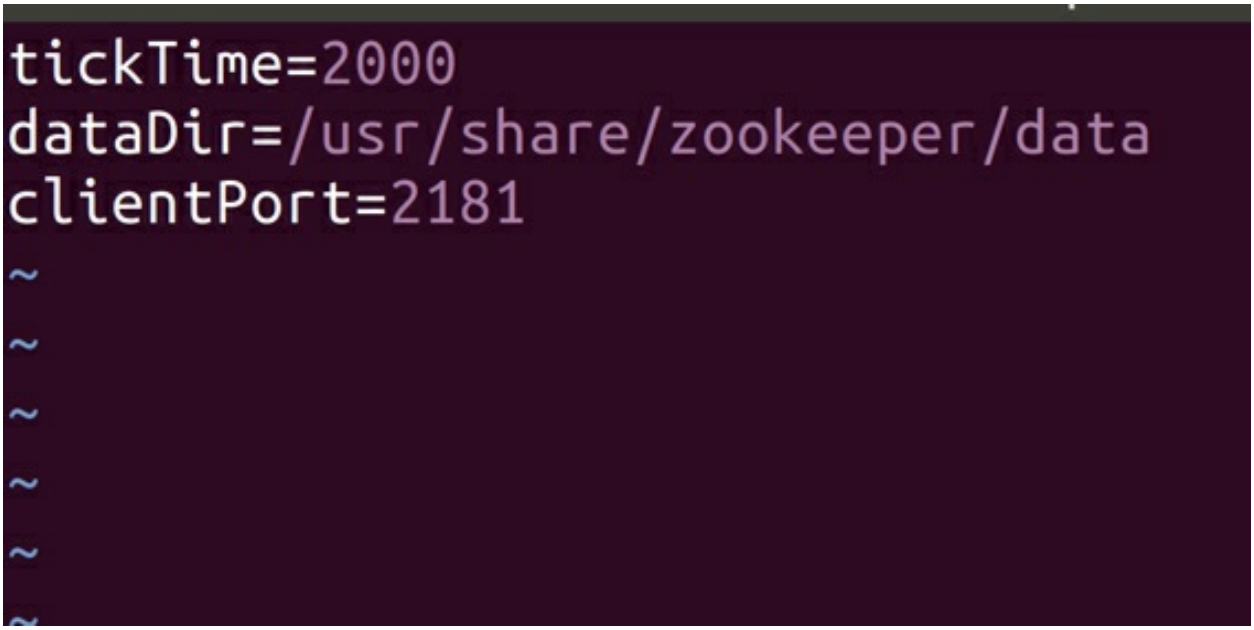


Step 5: Finally, let us create a ZooKeeper configuration file to include the data directory we created in the previous task along with other information. To create the configuration file, run the following command from the command line interface:

```
$ sudo vi zookeeper/conf/zoo.cfg
```

Enter the following settings in the file, and save it:

```
tickTime=2000
dataDir=/usr/share/zookeeper/data
clientPort=2181
```



The above settings are sufficient for configuring ZooKeeper in standalone mode.

However, If you wish to install ZooKeeper in replicated mode, please enter the following settings in the configuration file and ensure you have the same settings in all the servers in ZooKeeper Ensemble:

```
tickTime=2000
dataDir=/usr/local/zookeeper-3.4.6/data
clientPort=2181
initLimit=5
syncLimit=2
server.1=<server_name>:2888:3888
server.2=<server_name>:2888:3888
server.3=<server_name>:2888:3888
```

tickTime: The basic time unit in milliseconds used by Zookeeper. This is used to do heartbeats. The minimum session timeout is twice the *tickTime*.

dataDir: The location for storing the in-memory database snapshots and, unless specified otherwise, the transaction log of updates to the database.

clientPort: The port to listen for client connections.

initLimit: The total amount of time allowed for the quorum members (followers) to connect to and sync with the leader. If most of the quorum members fail to sync with the leader during this period, the leader powers are revoked, and a new election for leader occurs.

syncLimit: The total amount of time allowed for the quorum members to sync with the leader. If the quorum member fails to sync during this period, it will restart itself. Clients will be routed to other quorum members if they were connected to this quorum member.

Next, we specify the servers in the ensemble for each line with a server number. There are two port settings: The first port is used by followers to connect to the leader, while the second is used for leader election. There are three port numbers on which the servers listen. The description is as follows:

2181: Port for client connections.
2888: Used by followers to connect to the leader.
3888: Used for leader election.

In addition to these settings, you will need to create a file called *myid* in the data directory that contains a numeric identifier for each ZooKeeper server in the ensemble. The range for this numeric identifier is from 1 to 255. This denotes the server number set in the configuration file above. If you are creating the *myid* file in server.1, simply enter 1 in the *myid* file, and save it. Similarly, if you are creating the *myid* file in server.2, simply enter 2 in the *myid* file, and so on.

```
$ sudo vi zookeeper/data/myid
```

Enter the numeric identifier ranging from 1 to 255 in the file depending on the server configuration file.

Once we start the server, it reads the *myid* file and determines which server it is based on the numeric identifier. It then reads the configuration file for all the information regarding the ports and network addresses of other servers within the ensemble.

Step 6: Change the ownership of the files zoo.cfg and myid (replicated mode) by running the following command:

```
$ sudo chown <username> zookeeper/conf/zookeeper.cfg
```

```
$ sudo chown <username> zookeeper/data/myid
```

Run the above command only if you are installing ZooKeeper in replicated mode.

This completes the installation of ZooKeeper.

Task 3 is complete!

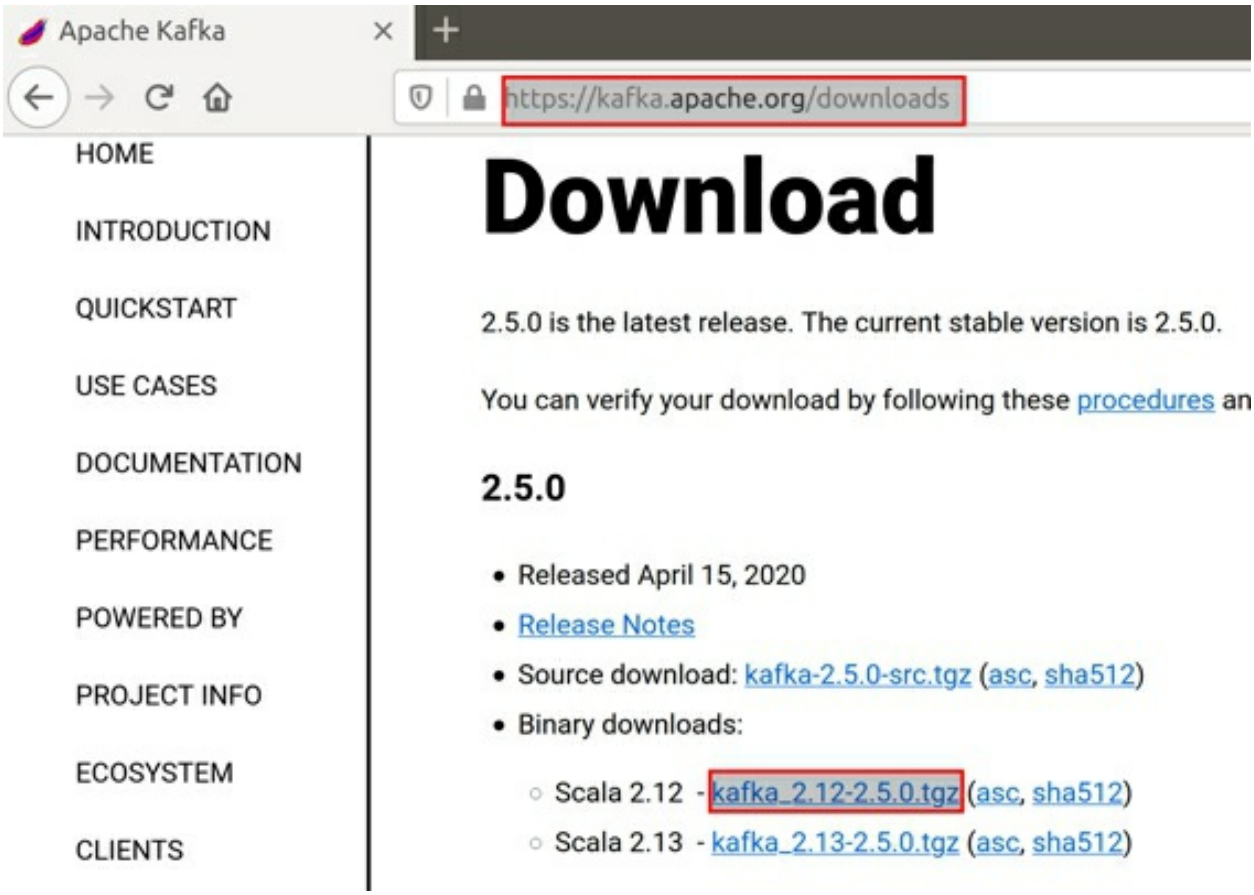
TASK 4: DOWNLOAD AND INSTALL KAFKA

Now that we have ZooKeeper installed and configured, let us proceed by installing Kafka.

Step 1: Let us install Kafka in standalone mode. Navigate to the download URL below and click to download the latest stable version of Kafka (Kafka 2.12-2.5.0 at the time of writing this book).

Note: 2.12 is the Scala version, and 2.5.0 is the Kafka version.

Download URL: https://kafka.apache.org/downloads



After clicking the download link, you will be directed to a page with a mirror site to download Kafka. Click the mirror link as presented below, and your download should begin. The download might take some time depending on your internet connection.

Step 2: The download will be saved to the *Downloads* directory by default. Execute the following command from your terminal to change the directory to the *Downloads* folder:

```
$ cd
$ cd Downloads
```

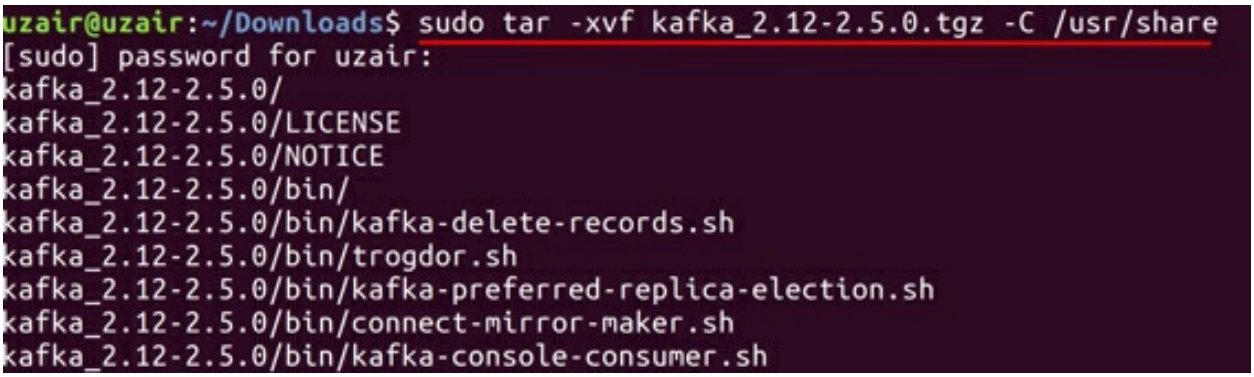
Once you are in the *Downloads* directory, you may optionally check whether Kafka has been downloaded using the *ls* command:

```
$ ls
```

Now that you are certain you have the Kafka tar file, untar the Kafka tar file to the */usr/share* directory using the command below:

```
$ sudo tar –xvf apache-zookeeper-3.6.1-bin.tar.gz –C /usr/share
```

The file will begin to untar to the */usr/share* directory, as illustrated in the screenshot below.



Step 3: Let us create a softlink to the Kafka directory so that we do not need to refer to Kafka with the entire name as above. This will be useful for future updates, as well. Execute the following command:

```
$ sudo ln –s kafka_2.12-2.50 kafka
```

Run the ls command again to check whether we could create the softlink successfully.

```
uzair@uzair:/usr/share$ cd /usr/share
uzair@uzair:/usr/share$ ls ka*
bin  config  libs  LICENSE  NOTICE  site-docs
uzair@uzair:/usr/share$ sudo ln -s kafka-2.12-2.5.0 kafka
uzair@uzair:/usr/share$ ls kafka
kafka
uzair@uzair:/usr/share$
```

Step 4: Change the permissions of the Kafka directory by running the following command:

```
$ sudo chown <username> /usr/share/kafka
```

<username> - Insert your username.

Task 4 is complete!

TASK 5: CONFIGURE KAFKA

Step 1: Let us now set up the environment variables for Kafka. Begin by executing the following command:

```
$ sudo vi ~/.bashrc
```

```
uzair@uzair:/usr/share$ sudo vi ~/.bashrc
```

The file should open as presented below.

```
# ~/.bashrc: executed by bash(1) for non-login shells.
# see /usr/share/doc/bash/examples/startup-files (in the package bash-doc)
# for examples

# If not running interactively, don't do anything
case $- in
    *i*) ;;
      *) return;;
esac

# don't put duplicate lines or lines starting with space in the history.
# See bash(1) for more options
HISTCONTROL=ignoreboth

# append to the history file, don't overwrite it
shopt -s histappend

# for setting history length see HISTSIZE and HISTFILESIZE in bash(1)
HISTSIZE=1000
HISTFILESIZE=2000

# check the window size after each command and, if necessary,
# update the values of LINES and COLUMNS.
```

Now press the *i* key to edit the file and append the following environment variable at the end of the file:

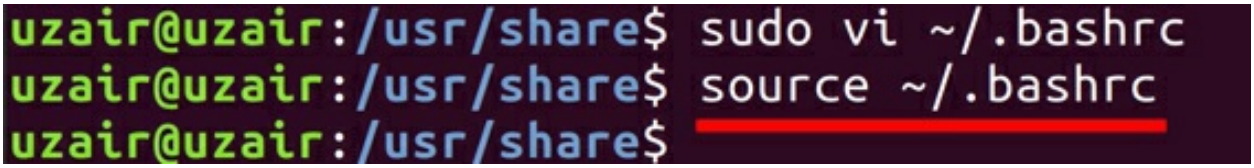
```
KAFKA_HOME=/usr/share/kafka
Export PATH=$KAFKA_HOME/bin:$PATH
```

```
# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if ! shopt -oq posix; then
  if [ -f /usr/share/bash-completion/bash_completion ]; then
    . /usr/share/bash-completion/bash_completion
  elif [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
  fi
fi

SPARK_HOME=/usr/share/spark
export PATH=$SPARK_HOME/bin:$PATH
ZOOKEEPER_HOME=/usr/share/zookeeper
export PATH=$ZOOKEEPER_HOME/bin:$PATH
KAFKA_HOME=/usr/share/kafka
export PATH=$KAFKA_HOME/bin:$PATH
"~/.bashrc" 124L, 3970C                                        123,27
```

After you have finished appending the text above, press the *Esc* key on your keyboard to stop editing, and then press *Shift - Z - Z* to exit the editor by saving the changes (please note that you need to press Z twice while holding the Shift key).

Now reload the modified *.bashrc* file using the following command:

```
$ source ~/.bashrc
```

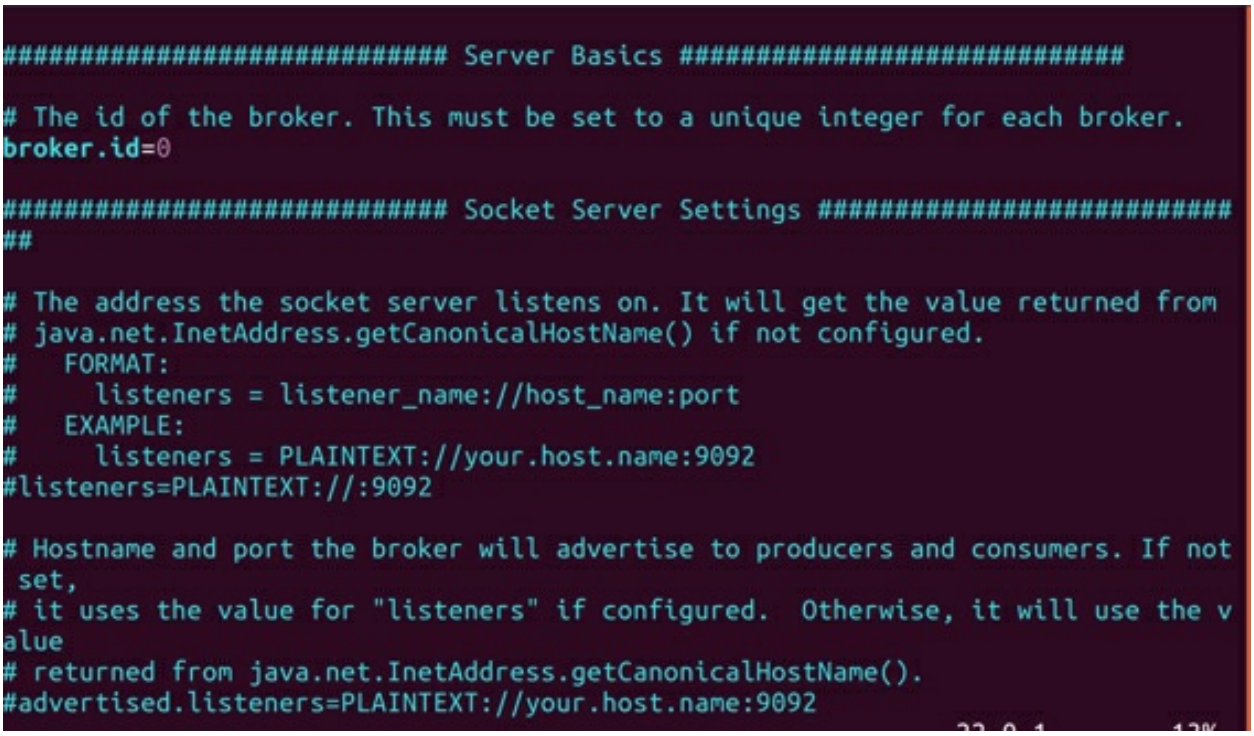


This completes the installation for Kafka in standalone mode. The configuration file provided with the Kafka installation is sufficient for running Kafka for training purposes. However, when working in a real-time environment, these configurations are not sufficient.

Let us now consider a few of the configuration options available in Kafka.

Step 2: The configuration properties for Kafka are available in the following path:

```
/usr/share/kafka/config/server.properties
```



broker.id: The broker.id is the configuration property used to identify a broker in a Kafka cluster. The broker.id must be a unique integer value for every broker in the Kafka cluster and is set to 0 by default.

listeners: The listeners property is used to set the URIs as comma-separated values, which the brokers will use to create server sockets. The value consists of the hostname and port, with the default port being 9092. The port number can be changed to any other available port.

zookeeper.connect: This configuration property is used to set the connection string of a ZooKeeper server in *hostname:port* format. Multiple hostnames can be provided, separated with a comma. The default value is *localhost:2181* for this property. The hostname could be the hostname or IP address of the ZooKeeper server. Optionally, this property can have a ZooKeeper chroot path as part of its ZooKeeper connection string, which puts its data under some path in the global ZooKeeper namespace.

zookeeper.connection.timeout.ms: The value set in this property specifies the timeout value for connecting to ZooKeeper in ms. The default value is 18,000.

log.dirs: We have learned that messages in Kafka are persistent (*i.e.*, the

data received from producers is stored on the disk). This property specifies the location/path for data to be stored. Multiple paths can be specified as a comma-separated list. Whenever multiple paths are specified, the broker stores the partitions in the path that has the fewest partitions. The default path is set to */tmp/kafka-logs*.

num.partitions: The num.partitions property is used to specify the default number of partitions per topic. The default value for this property is 1.

num.recovery.threads.per.data.dir: This property specifies the number of threads used per data directory to recover the data at startup and flushing at shutdown. The default value for this property is 1.

auto.create.topics.enable: This property, when set to true, creates topics automatically whenever applications produce, consume, or fetch metadata for a non-existent topic. The automatically created topic will have the default partitions and replications. It is always recommended to set this property to false so that the topics are not automatically created without your knowledge.

log.retention.hours: This property specifies the number of hours after which Kafka will delete the messages. Other similar properties concern the deletion of logs after a certain period, namely *log.retention.minutes* and *log.retention.ms*. All these properties perform the same operation; however, the property with the lowest time unit takes precedence (*i.e.*, *log.retention.ms* has precedence over *log.retention.minutes*, and so on). The default value for *log.retention.hours* is 168.

log.retention.bytes: This property determines the size of a message upon which it should be deleted. The size after which the log should be deleted must be specified in bytes.

These are a few of the configuration properties for configuring the Kafka broker. You can find all the configuration properties in the Kafka documentation URL available in the references link.

Task 5 is complete!

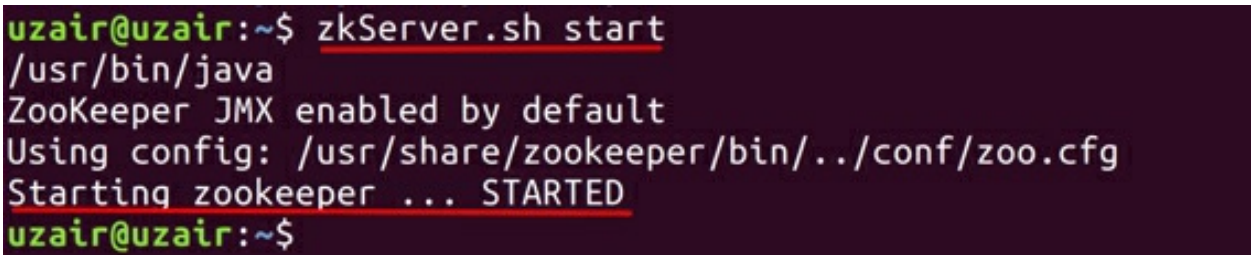
TASK 6: STARTING ZOOKEEPER AND KAFKA

Now that we have completed the installation and configuration of ZooKeeper and Kafka, let us start them.

Step 1: First, start ZooKeeper by running the following command:

```
$ zkServer.sh start
```

You should observe that the ZooKeeper server starts as presented in the screenshot below.



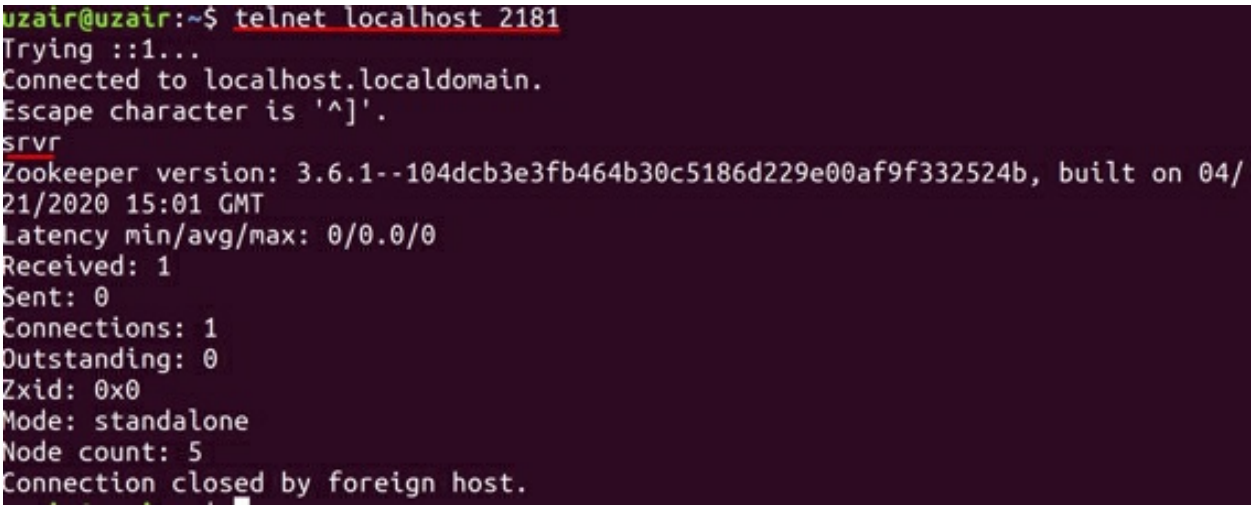
Step 2: Let us attempt to connect to the client port and run the command *srvr*. Run the following command first:

```
$ telnet localhost 2181
```

When you see you are connected to localhost, type the following command:

```
srvr
```

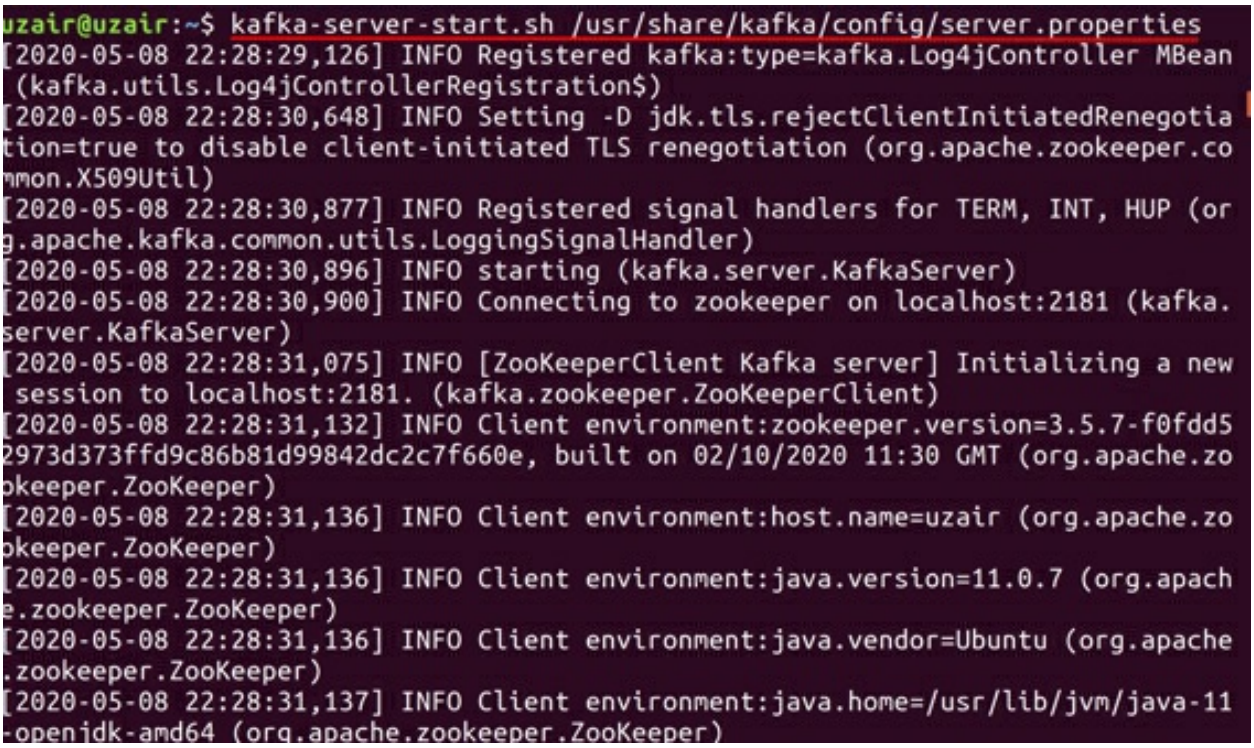
You should see the output as displayed below.



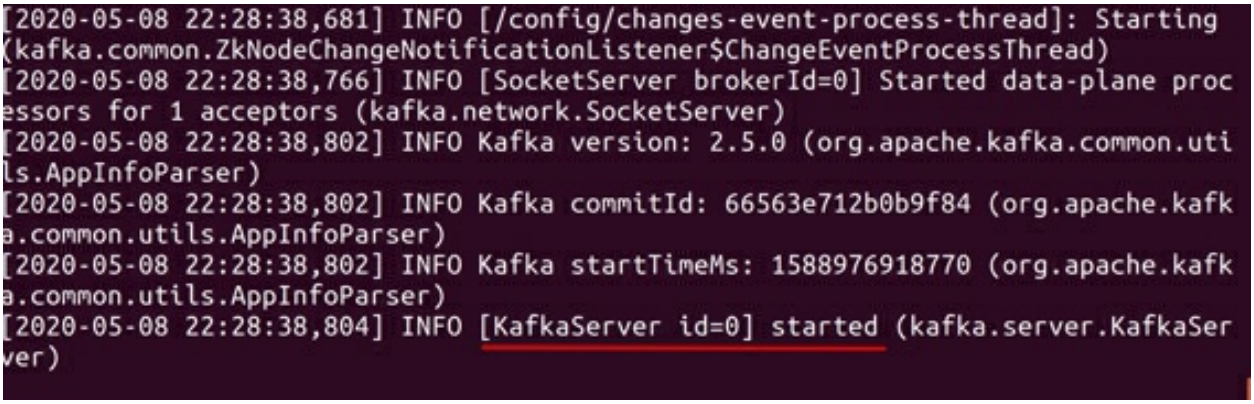
Step 3: Let us now start the Kafka server. Run the following command:

```
$                                    kafka-server-start.sh
```

/usr/share/kafka/config/server.properties



You should see a large amount of information while Kafka starts. The last line informs you that Kafka has been started, as presented below.



Task 6 is complete!

SUMMARY

Kafka is an open-source, distributed, persistent, and fault-tolerant message-streaming platform or central repository that can handle high volumes (trillions) of Publish-Subscribe messages each day. A Publish-Subscribe messaging system is a system in which data is produced (publish) by producers and consumed (subscribe) by consumers.

The Kafka architecture consists of the following components:

- Brokers
- Producers
- Consumers
- Topics
- Partitions
- Replications
- ZooKeeper

CHAPTER 3:
KAFKA IN-DEPTH
PART I

THEORY

In the previous chapter, we have looked at the architecture of Kafka with an overview of its components. We have also installed and configured Kafka in the labs. In this chapter and the next one, let us dig a little deeper and understand how Kafka works internally at core. It is paramount to understand the internal working of Kafka for efficient production and to troubleshoot issues, when necessary.

Let us look at the following internals of Kafka in this chapter.

- Topics Operations
- Topics Overview
- Data Model in ZooKeeper
- ZooKeeper Watches
- ZooKeeper's role in cluster membership
- Election of Controller Broker
- Responsibilities of Controller Broker

TOPIC OPERATIONS

In the previous chapter, we have seen what are Topics. Now, let us look at various operations of Topics and see what happens internally when Topics are created, altered, deleted, etc.

Creating a Kafka Topic

A Kafka Topic can be created with the following syntax.

```
$ kafka-topics.sh \
--zookeeper <ZOOKEEPER_URL:PORT> \
--create --topic <TOPIC_NAME> \
--replication-factor <NO_OF_REPLICATIONS> \
--partitions <NO_OF_PARTITIONS>
```

Please note that the '\' symbol is not required if you specify the syntax in single line. The '\' symbol is only used here since the syntax is specified in multiple lines.

- The script to manage (create, alter, delete, list, etc.) Kafka Topics is *kafka-topics.sh*. This script is available in *<KAFKA_HOME>/bin/* directory. Therefore, we use the *kafka-topics.sh* script file to create a Kafka Topic.

- Next, we specify the URL and Port on which ZooKeeper is running using the --*zookeeper* switch.

- The next part of the syntax uses the --*create*, followed by the --*topic* switch to specify the name of the new topic that has to be created.

- The --*replication-factor* switch is used to specify the number of replications for the Kafka Topic.

- Similarly, the --partitions switch is used to specify the number of partitions.

Since Kafka is a persistent message queue, all the messages are stored on the disk. The path where the messages will be stored is set in the *server.properties* file pointed in the *log.dirs* property, as seen in the previous chapter.
We shall be looking at the contents of logs directory in the lab exercise for this chapter.

Listing Kafka Topics

The following syntax is used to list all the available Topics on the Kafka

server.

```
$ kafka-topics.sh \
--zookeeper <ZOOKEEPER_URL:PORT> \
--list
```

We use the *kafka-topics.sh* script, followed by the ZooKeeper details and specify the --*list* switch to list all the Kafka Topics.

Modifying Kafka Topics

A Kafka Topic can be modified to change the number of partitions using the --*alter* switch in *kafka-topics.sh* script as shown below.

```
$ kafka-topics.sh \
--zookeeper <ZOOKEEPER_URL:PORT> \
--alter --topic <TOPIC_NAME> \
--partitions <NO_OF_PARTITIONS>
```

Note: Modifying the number of partitions will affect the partition logic or ordering of messages for consumers. Do not worry if you do not understand this. It will be clear once we look at Consumers in detail.

Please note that we cannot change the number of replications using the above syntax. We shall look at this in the upcoming chapters.

Deleting Kafka Topics

A Kafka Topic can be deleted using the following syntax.

```
$ kafka-topics.sh \
--zookeeper <ZOOKEEPER_URL:PORT> \
--delete --topic <TOPIC_NAME>
```

Please note that the topics cannot be deleted if *delete.topic.enable* property is not set to true.

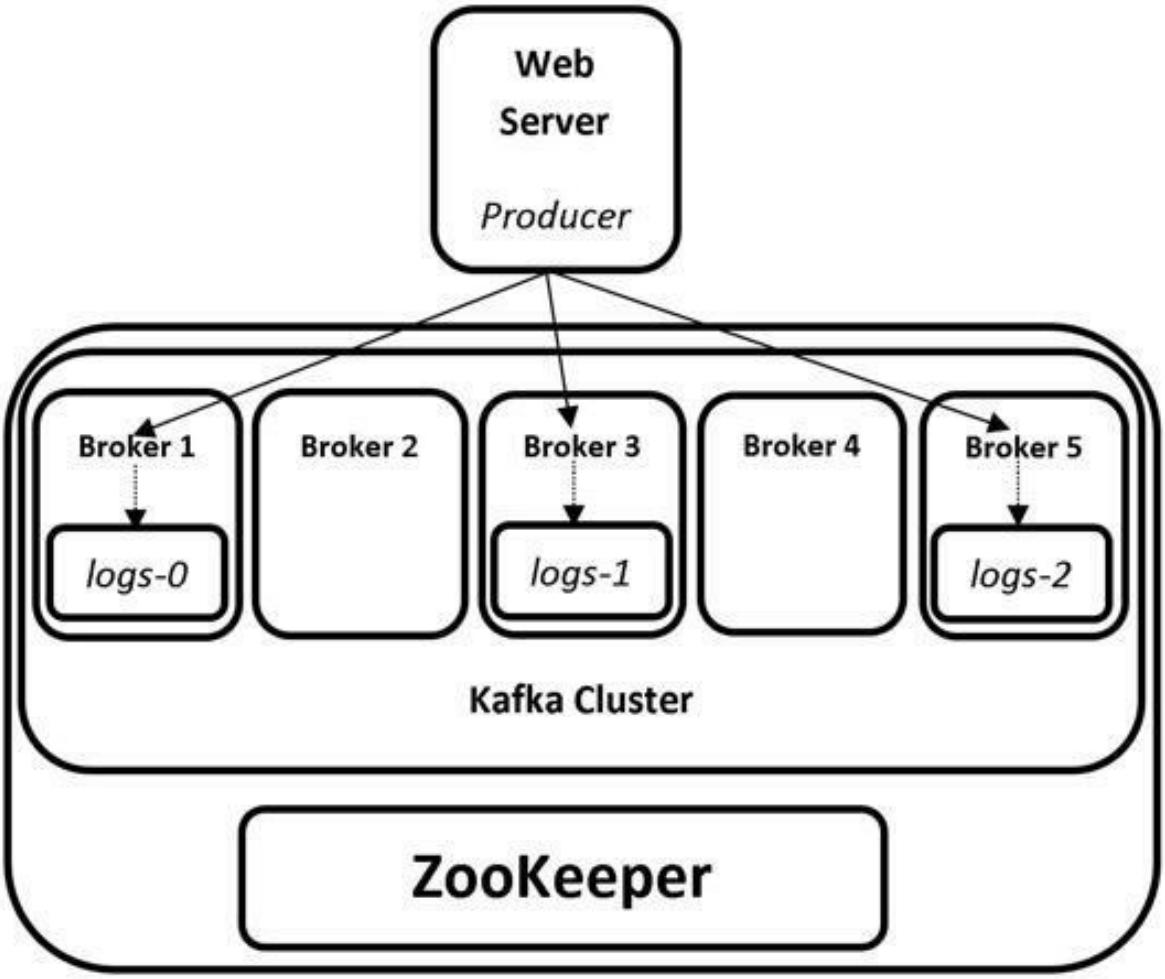
TOPICS OVERVIEW

Let us now understand how Topics receive messages from Producers. Consider a Kafka cluster with 5 Brokers and a web server that generates logs as our Kafka Producer.

- First, start the ZooKeeper server and all the Brokers one by one. Post starting all the Brokers, create a Topic as explained in the previous section. Let us name the Topic as *logs*.

- The topic is created with 3 Partitions and a replication factor of 1. Once the Topic is created, three directories will be created under the path specified in *logs.dir* property on three different Brokers, *i.e.*, one directory for each partition. The directories will have the names as logs-0, logs-1 and logs-2.

- Let us assume that the log directories are created in Brokers with ids 1, 3 and 5 as shown in the figure below.

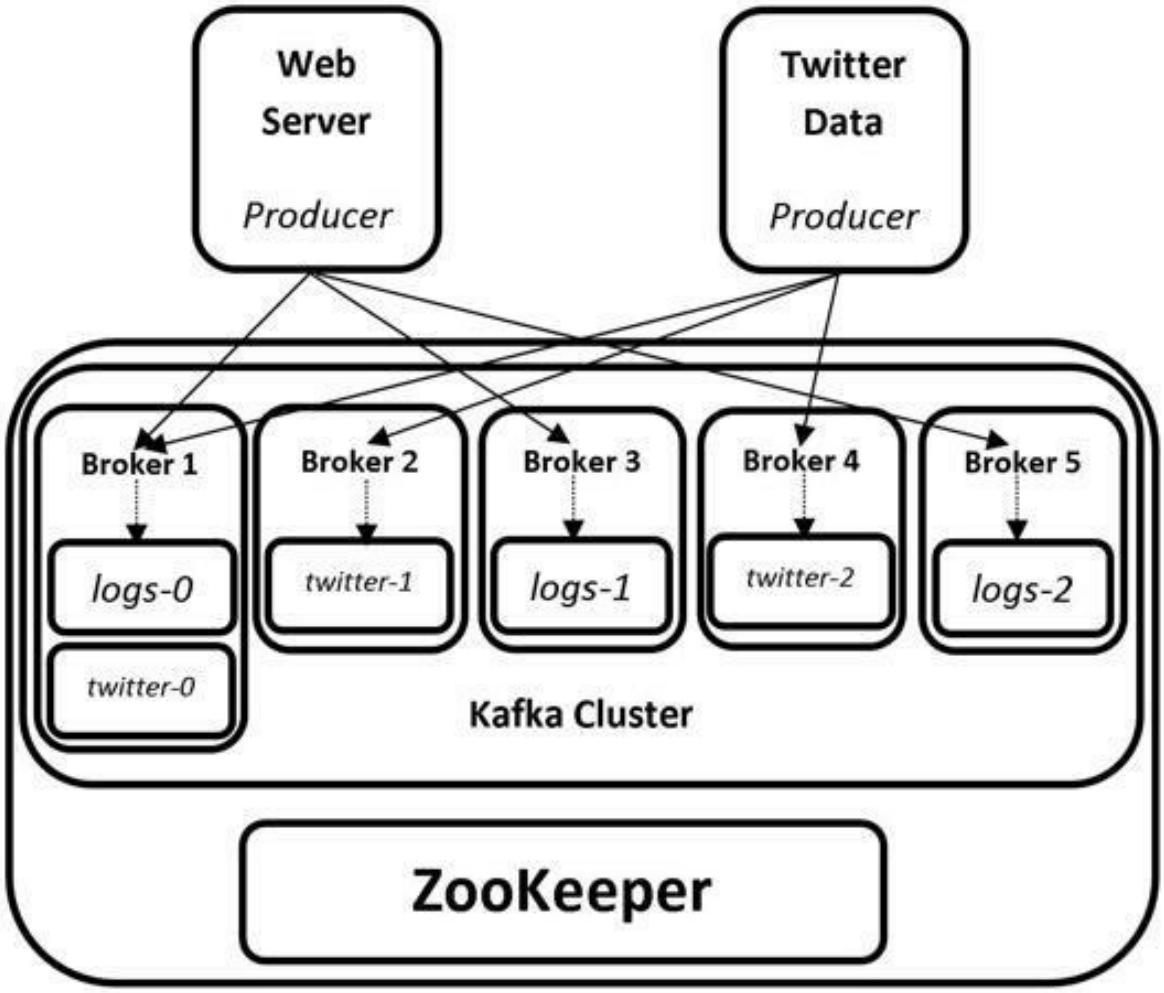


3(a) Topics Overview – Single Producer

- The Producer will now send the data to the Brokers 1, 3 and 5.
- The Broker 1 will write the data to *logs-0*, Broker 3 will write the data to *logs-1* and Broker 5 will write the data to *logs-2*.

Let us consider we have one more producer which generates Twitter data. The data is stored in the same process as above.

- Create a topic and name it *twitter*. The topic has 3 partitions with a replication factor of 1.
- Let us assume that the log directories are created in Brokers with ids 1, 2 and 4 as shown in the figure below.
- The Producer will now send the data to the Brokers 1, 2 and 4.
- The Broker 1 will write the data to *twitter-0*, Broker 2 will write the data to *twitter-1* and Broker 4 will write the data to *twitter-2*.



3(b) Topics Overview – Multiple Producers

From the above illustration, we can say that Kafka is a distributed, as the data is saved in the form of partitions across the cluster and, is persistent as the data is saved on disk.

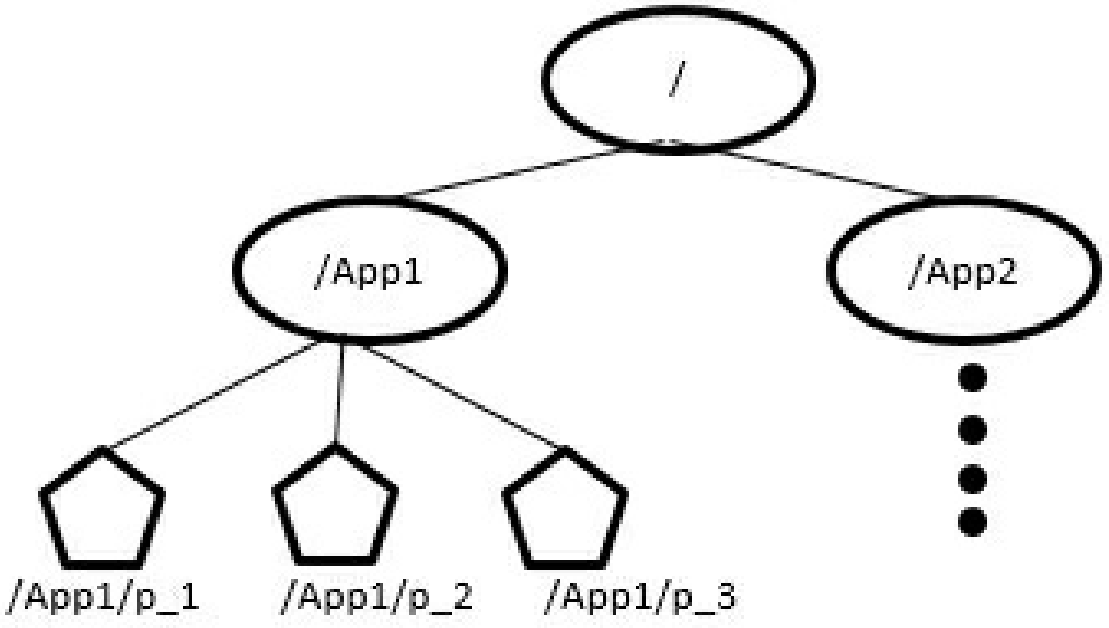
In the illustration above, we have seen that when a directory is created in a Broker, the Broker takes all the ownership (read/write operations) of that directory. From our example above, Broker 1 is responsible for all the operations for *logs-0* and *twitter-0* partitions. But, how has the Broker 1 been assigned the ownership for *logs-0* and *twitter-0* partitions? Through which process has this responsibility assigned? Controller Broker is the answer for these questions. Move on to the next sections for a detailed explanation.

Before we learn about Controller Broker in detail, let us first understand the ZooKeeper data model and watches in ZooKeeper.

DATA MODEL IN ZOOKEEPER

Data model in ZooKeeper is a distributed, hierarchical file system comprised of znodes (directories). The znodes are containers for data and other nodes. The znodes have associated data such as statistic information and versioning information. Znodes also store user data which is limited to 1 MB (per znode).

3(c) ZooKeeper Data Model



The znodes can be of Persistent or Ephemeral znodes and optionally sequential znodes.

Persistent znodes	Persistent znodes persist even after the client session. expires or disconnected. Persistent znodes can have child. Persistent znodes are always present and is only deleted when clients explicitly deletes it.
Ephemeral	Unlike persistent znodes, ephemeral znodes

znodes	are automatically deleted by zookeeper as soon as the session in which they are created is closed or expired. Ephemeral znodes cannot have children, not even the ephemeral children.
Sequential znode	A sequential znode is given a sequence number while znode is being created. All znodes can optionally be sequential, for which ZooKeeper maintains a monotonically increasing counter which is automatically appended to the znode name upon creation. Each sequence number is guaranteed to be unique.

ZOOKEEPER WATCHES

Watches notify clients when a znode has been changed. We can set watches on read or get notified when there is a write operation on a znode. For example, a client, say *x*, sets a watch on a znode and another client *y* performs a write request on that znode, a watch is triggered and client *x* is notified about the write operation. Client *x* and *y* are completely independent and do not know anything about each other's existence, so long as they each know their own responsibilities in relation to specific znodes.

Also the watches are triggered only once and notify the client about the change. Once triggered, the client has to register again to get notified for future notifications. It is also possible that during the period between receipt of the notification and re-registration, other clients could perform write operations on the znode before the new Watch is registered by the client and the client may miss that update/notification.

ZOOKEEPER'S ROLE IN CLUSTER MEMBERSHIP

As we have learned in previous chapters, Kafka uses ZooKeeper to manage the Brokers which are the members of Kafka cluster. Each Broker has a

unique id that is either set via *broker.id* property in the Broker's *server.properties* configuration file or is automatically generated. The following actions happen when Brokers are started.

- Whenever a Broker is started, the Broker registers itself with its ID in ZooKeeper with information such as its hostname, port and other metadata. The Broker creates a directory in ZooKeeper known as Ephemeral node, to store this information. The list of Brokers will be available in the following ZooKeeper path: */brokers/ids*. For example, if the Broker id is 1, an Ephemeral node is created as */brokers/ids/1*. The Ephemeral node 1 consists of all the metadata information regarding Broker 1.

- All the different Kafka components such as other brokers, producers, consumers etc., subscribe (set ZooKeeper watch) to the Brokers' path above. This was they get notified whenever a change such as addition or removal of Brokers takes place.

- A Broker ID is always unique and a new Broker cannot be started with the same ID. If a Broker ID is started with same ID, an error is thrown since there is already a Broker running with the same ID.

- In case of Broker failure, the Ephemeral node created by that particular Broker which failed, will be removed from ZooKeeper. This will trigger a notification to all the Kafka components, which subscribed to Brokers' path regarding the Broker failure.

- A new Broker with the same ID can then be commissioned and have the data restored with the help of replicas available in other Brokers. The metadata information of partitions stored in the failed Broker will be available with the Controller Broker (Details in the next section). Thus making Kafka fault tolerant.

This is how ZooKeeper helps in maintaining Brokers' membership with Kafka cluster.

ELECTION OF CONTROLLER BROKER

The Controller Broker can be thought as a master which controls the membership of all other Brokers in Kafka cluster. The Controller Broker is responsible to allocate partitions to the Brokers, also known as Leader Brokers. Controller Broker is not a special server, but one among the list of Brokers available in the Kafka cluster. The Controller Broker is elected with the help of ZooKeeper. Let us see how the Controller Broker is elected with the help of ZooKeeper and learn its responsibilities.

- Consider a Kafka cluster with few Brokers and a ZooKeeper server.
- When the Brokers are started in Kafka cluster, every Broker tries to create an ephemeral node (directory) called */controller* in ZooKeeper.
- The Broker that succeeds in creating this ephemeral node becomes the Controller Broker in Kafka cluster. Generally, the first Broker that starts in Kafka cluster succeeds in creating the ephemeral node and thus becomes Controller Broker.
- The other Brokers also try to create this node but fail to do so, as the Controller Broker is already elected. This ensures that there is only one Controller Broker at any given point of time.
- Since all the other Brokers now know there is a Controller Broker, they subscribe (set ZooKeeper watch) to the */controller* node to get notified of any changes that happen to this node.
- If the Controller Broker shuts down due to hardware failure or experiences connectivity issues with ZooKeeper, the */controller* ephemeral node will be deleted By ZooKeeper (See data model in ZooKeeper section for more information).
- Since the removal of */controller* node is a change, all the other Brokers that have set a watch to this node will be notified that the cluster no longer has a Controller Broker.

- All the Brokers will once again try to create an ephemeral node called */controller* in ZooKeeper. Whichever Broker succeeds in creating this node will become the Controller Broker and the rest of the Brokers will again set a ZooKeeper watch to this node.

- Every Controller Broker that gets elected is assigned with a controller epoch number, which is higher than the epoch that is assigned to previous Controller. So, if the Controller Broker that failed due to connectivity issues or any other reason, gets connected again, the rest of the Brokers will simply ignore its messages based on the Controller's epoch number.

This is how a Controller Broker is elected and its high availability is ensured using ZooKeeper. We shall look this in real time in our lab exercises.

RESPONSIBILITIES OF CONTROLLER BROKER

Controller Broker has a few more responsibilities apart from the responsibilities of a regular Broker in Kafka. The Controller Broker acts as a regular Broker for communication between Producer and Consumer, as well as the following responsibilities.

- The Controller Broker is responsible to assign partitions and replications to Brokers. The Controller Broker maintains metadata of Leader Brokers and Follower Brokers. Controller Brokers sets a ZooKeeper watch on the */brokers/ids* node and hence it gets notified when a new Broker has joined or left the cluster.

- The Controller Broker shares all the metadata information such as the Brokers leading partitions, the Brokers having replications with all the other Brokers in the Cluster. This way if the Controller Broker is down, the newly elected Controller Broker has all the metadata information to start its duties as a Controller. In simple words, every Broker on the cluster knows the metadata information of every other Broker. Thus any Broker has the potential to become a Controller Broker.

- The Controller Broker is also responsible for the duties like any other Broker in the cluster, *i.e.*, leading partitions, performing reads/writes and having partition replications. In other words, the Controller Broker assigns partitions and replications to itself and also to other Brokers.

- The Controller Broker monitors Brokers for failures and rebalancing partitions to other Brokers in the event of failure. When a Broker goes down, the Controller Broker elects a new Partition leader which has the replications of failed Broker. The new Partition leader will now start serving the requests from Producers and Consumers while the Follower Brokers start replicating the messages from Leader Broker.

That's all for the theory for this chapter. Let us move to the lab exercise to check all this theory in action.

AIM

The aim of the following lab exercises is to perform Kafka topic operations, work with ZooKeeper shell and demonstrate the election and responsibilities of a Controller Broker.

The labs for this chapter include the following exercises.

- Kafka Topic operations
- Hands-on ZooKeeper Shell
- Controller Broker Election

We need the following packages to perform the lab exercise:

- Java Development Kit
- Apache ZooKeeper
- Apache Kafka

LAB EXERCISE 3: KAFKA IN-DEPTH – PART I

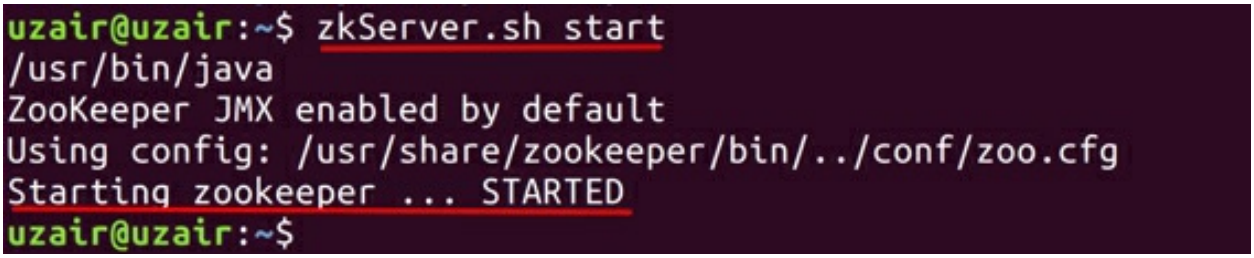
1. **Kafka Topic Operations**
2. **Hands-on ZooKeeper Shell**
3. **Controller Broker election**

TASK 1: KAFKA TOPIC OPERATIONS

Step 1: Let us first start the ZooKeeper server, if not started already.

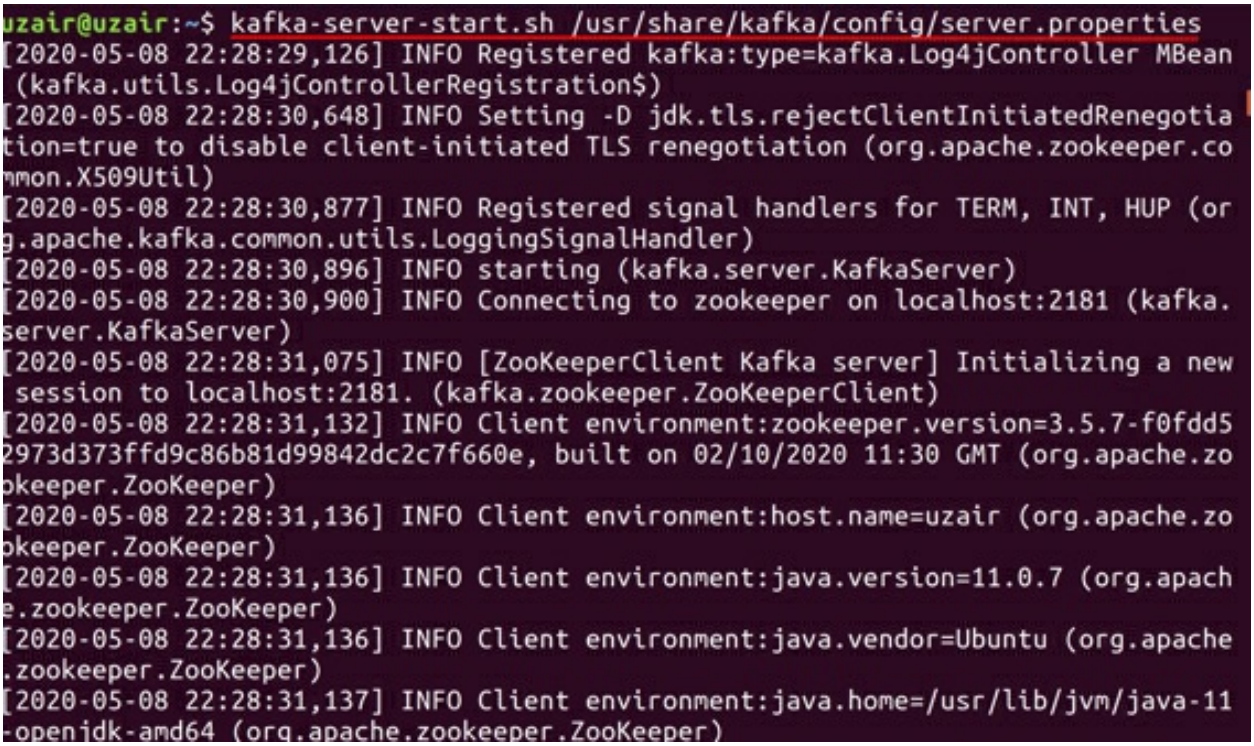
```
$ zkServer.sh start
```

You should see that the ZooKeeper server starts as shown in the screenshot below.



Step 2: Let us now start the Kafka Broker, if not started already

```
$ kafka-server-start.sh /usr/share/kafka/config/server.properties
```



You should see a lot of information while Kafka starts. The last line would inform you that Kafka has been started as shown below.

```
[2020-05-08 22:28:38,681] INFO [/config/changes-event-process-thread]: Starting
(kafka.common.ZkNodeChangeNotificationListener$ChangeEventProcessThread)
[2020-05-08 22:28:38,766] INFO [SocketServer brokerId=0] Started data-plane proc
essors for 1 acceptors (kafka.network.SocketServer)
[2020-05-08 22:28:38,802] INFO Kafka version: 2.5.0 (org.apache.kafka.common.uti
ls.AppInfoParser)
[2020-05-08 22:28:38,802] INFO Kafka commitId: 66563e712b0b9f84 (org.apache.kafk
a.common.utils.AppInfoParser)
[2020-05-08 22:28:38,802] INFO Kafka startTimeMs: 1588976918770 (org.apache.kafk
a.common.utils.AppInfoParser)
[2020-05-08 22:28:38,804] INFO [KafkaServer id=0] started (kafka.server.KafkaSer
ver)
```

Step 3: Now that we have ZooKeeper and Kafka server up and running, let us perform few Topic operations. Open a new terminal and run the following command to create a new Topic.

```
$ kafka-topics.sh \
--zookeeper localhost:2181 \
--create --topic http-logs \
--replication-factor 1 \
--partitions 3
```

You should see the confirmation message that the Topic is created as shown in the screenshot below.

```
uzair@uzair:~$ kafka-topics.sh \
> --zookeeper localhost:2181 \
> --create --topic http-logs \
> --replication-factor 1 \
> --partitions 3
Created topic http-logs.
uzair@uzair:~$
```

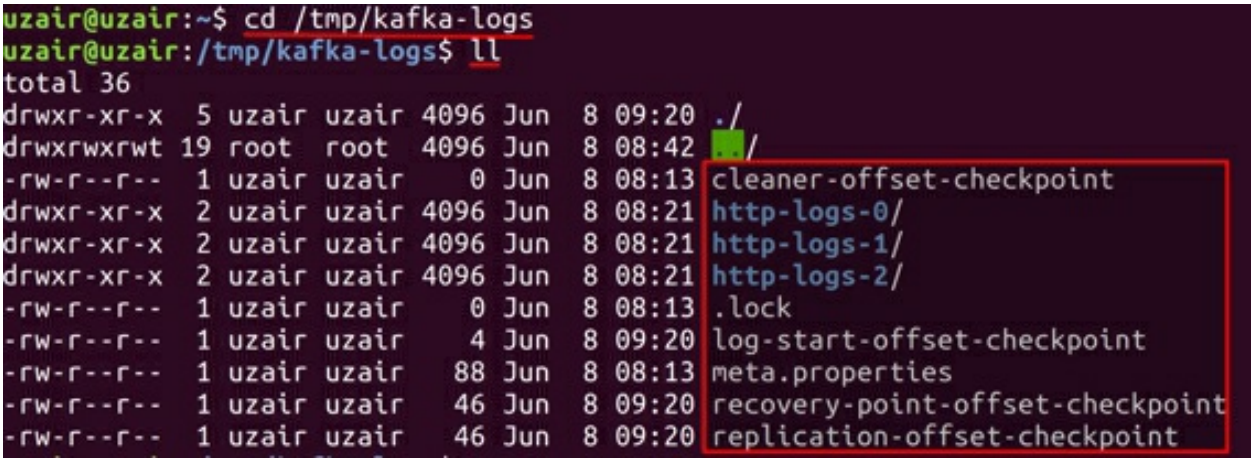
Since Kafka is a persistent message queue, the messages will be stored on disk. Let us now look the path where this new topic is created. This is the same path that is specified in the *logs.dir* property in *server.properties* configuration file. The path that is specified in *server.properties* configuration file is */tmp/kafka-logs*. Please check the previous lab exercise for more information on Kafka configurations.

Note: The default path specified to store Kafka logs is the *tmp* folder. This path should always be changed to a new path, as the *tmp* folder will be deleted automatically when you restart the server.

Enter the following commands to check the contents of the topic we created.

```
$ cd /tmp/kafka-logs
$ ll
```

You should see the following contents as shown in the screenshot below.



As seen in the screenshot above, three directories have been created for the topic as we have specified 3 partitions. The number of directories created will be directly proportional to the number of partitions specified while creating a topic. Since we are only working on single Kafka server, all the three partitions have been created in the same server. In multi-node Kafka cluster, these partitions will be created on any of the Kafka servers. The rest of the files in the kafka-logs directory contain the metadata information.

Now, let us look the contents of a directory of the partition.

```
$ cd http-logs-0
$ ll
```

The directory contains the following files.

```
uzair@uzair:/tmp/kafka-logs$ cd http-logs-1
uzair@uzair:/tmp/kafka-logs/http-logs-1$ ll
total 12
drwxr-xr-x 2 uzair uzair     4096 Jun  8 08:21 ./
drwxr-xr-x 5 uzair uzair     4096 Jun  8 10:38 ../
-rw-r--r-- 1 uzair uzair 10485760 Jun  8 08:21 00000000000000000000.index
-rw-r--r-- 1 uzair uzair        0 Jun  8 08:21 00000000000000000000.log
-rw-r--r-- 1 uzair uzair 10485756 Jun  8 08:21 00000000000000000000.timeindex
-rw-r--r-- 1 uzair uzair        8 Jun  8 08:21 leader-epoch-checkpoint
uzair@uzair:/tmp/kafka-logs/http-logs-1$
```

The file with *.log* extension is where all the messages from producers get appended. Any new messages generated are appended here by the Broker, thus making Kafka a persistent storage message queue.

Step 4: Let us now list all the Topics that are available in Kafka. Since we have only created one Topic, it is the only one that should return once we run the command below.

```
$ kafka-topics.sh \
--zookeeper localhost:2181 \
--list
```

```
uzair@uzair:~$ kafka-topics.sh \
> --zookeeper localhost:2181 \
> --list
http-logs
uzair@uzair:~$
```

Step 5: Let us now modify the number of partitions for the topic we created in previous step to 4.

```
$ kafka-topics.sh \
--zookeeper localhost:2181 \
--alter --topic http-logs \
--partitions 4
```

```
uzair@uzair:~$ kafka-topics.sh \
> --zookeeper localhost:2181 \
> --alter --topic http-logs \
> --partitions 4
WARNING: If partitions are increased for a topic that has a key, the partition l
ogic or ordering of the messages will be affected
Adding partitions succeeded!
```

Please note that you can only increase the number of partitions and not decrease them.

You can check if this action has been successful by navigating to the *kafka-logs* directory.

```
uzair@uzair:~$ ll /tmp/kafka-logs
total 40
drwxr-xr-x  6 uzair uzair 4096 Jun  8 10:58 ./
drwxrwxrwt 18 root  root  4096 Jun  8 10:55 ../
-rw-r--r--  1 uzair uzair    0 Jun  8 08:13 cleaner-offset-checkpoint
drwxr-xr-x  2 uzair uzair 4096 Jun  8 08:21 http-logs-0/
drwxr-xr-x  2 uzair uzair 4096 Jun  8 08:21 http-logs-1/
drwxr-xr-x  2 uzair uzair 4096 Jun  8 08:21 http-logs-2/
drwxr-xr-x  2 uzair uzair 4096 Jun  8 10:54 http-logs-3/
-rw-r--r--  1 uzair uzair    0 Jun  8 08:13 .lock
-rw-r--r--  1 uzair uzair    4 Jun  8 10:58 log-start-offset-checkpoint
-rw-r--r--  1 uzair uzair   88 Jun  8 08:13 meta.properties
-rw-r--r--  1 uzair uzair   60 Jun  8 10:58 recovery-point-offset-checkpoint
-rw-r--r--  1 uzair uzair   60 Jun  8 10:58 replication-offset-checkpoint
```

As you can see from the screenshot above, a new directory is created indicating the modification of partition was successful.

Step 6: Let us finally delete the Topic we created.

```
$ kafka-topics.sh \
--zookeeper localhost:2181 \
--delete --topic http-logs
```

You will see a message that the topic is marked for deletion as shown in the screenshot below.

```
uzair@uzair:~$ kafka-topics.sh \
> --zookeeper localhost:2181 \
> --delete --topic http-logs
Topic http-logs is marked for deletion.
Note: This will have no impact if delete.topic.enable is not set to true.
```

This is because Kafka will never delete a Topic right away. The Topic doesn't get deleted if Producers are still producing messages for the Topic, or if the

Consumers are still consuming the messages from the Topic. The Topic cannot be deleted if the *delete.topic.enable* is not set to true. This is to ensure no data loss happens when a Topic is accidentally deleted.

Task 1 is complete!

TASK 2: HANDS-ON ZOOKEEPER SHELL

To better understand how ZooKeeper works as a coordination service, let us look at the file system in ZooKeeper.

Step 1: Before we look at the ZooKeeper shell, let us first stop the Kafka and ZooKeeper servers. Let us also remove the directories that got created in *tmp* folder while we created new topic in the previous task.

Let us first stop Kafka and ZooKeeper.

```
$ kafka-server-stop.sh
$ zookeeper-server-stop.sh
```

Make sure you stop the ZooKeeper server after you stop the Kafka server.

Let us now remove all the *kafka-logs* and *zookeeper* directory inside the *tmp* folder.

```
$ rm –rf /tmp/kafka-logs
$ rm –rf /tmp/zookeeper
$ rm –rf /tmp/hsperfdata_{insert_your_username_here}
```

```
uzair@uzair:~$ kafka-server-stop.sh
uzair@uzair:~$ zookeeper-server-stop.sh
```

```
uzair@uzair:~$ rm -rf /tmp/kafka-logs
uzair@uzair:~$ rm -rf /tmp/zookeeper
uzair@uzair:~$ rm -rf /tmp/hsperfdata_uzair
```

Let us now only start the ZooKeeper server.

```
$ zkServer.sh start
```

```
uzair@uzair:~$ zkServer.sh start
/usr/bin/java
ZooKeeper JMX enabled by default
Using config: /usr/share/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
uzair@uzair:~$
```

Step 2: Now that the ZooKeeper server has been started, let us connect to the ZooKeeper server using the ZooKeeper Shell. Run the following command to connect to the ZooKeeper server.

```
$ zookeeper-shell.sh localhost:2181
```

You should see the prompt as shown in the screenshot below.

```
uzair@uzair:~$ zookeeper-shell.sh localhost:2181
Connecting to localhost:2181
Welcome to ZooKeeper!
JLine support is disabled

WATCHER::

WatchedEvent state:SyncConnected type:None path:null
```

Please note that the ZooKeeper Shell is not very user friendly. The ZooKeeper shell lacks many features you would expect.

Step 3: Let us now check what all directories or files exist in the ZooKeeper file system. We shall be using the list command on the root path of ZooKeeper.

```
ls /
```

```
uzair@uzair:~$ zookeeper-shell.sh localhost:2181
Connecting to localhost:2181
Welcome to ZooKeeper!
JLine support is disabled

WATCHER::

WatchedEvent state:SyncConnected type:None path:null
ls /
[zookeeper]
```

As you can see from the screenshot above there is only one item called *zookeeper*. This item can either be a file or a folder.

Step 4: Let us check if this item is a folder by running the list command.

```
ls /zookeeper
```

As you can see from the screenshot below, *zookeeper* is a directory and it contains two more items inside it. These items can again be either files or directories.

```
WatchedEvent state:SyncConnected type:None path:null
ls /
[zookeeper]
ls /zookeeper
[config, quota]
```

Step 5: Let us now try to read these files using the *get* command as shown below. You may also check if they are directories using the *ls* command. You should only get the square brackets if the item is not a directory as shown below.

```
get /zookeeper/quota
```

```
WatchedEvent state:SyncConnected type:None path:null
ls /
[zookeeper]
ls /zookeeper
[config, quota]
get /zookeeper/config

get /zookeeper/quota

ls /zookeeper/quota
[]
```

There nothing exists in those files as of now.

Step 6: Let is now start the Kafka server in a new terminal and check what all the files get generated within the ZooKeeper file system.

```
$ kafka-server-start.sh /usr/share/kafka/config/server.properties
```

Once the Kafka server has been started, switch to ZooKeeper shell and run the *ls* command on the root directory of ZooKeeper.

```
ls /
```

You should see the following nodes getting created in the Zookeeper as shown in the screenshot below.

```
ls /
[admin, brokers, cluster, config, consumers, controller, controller_epoch, isr_c
hange_notification, latest_producer_id_block, log_dir_event_notification, zookee
per]
```

Earlier we only had the *zookeeper* node. After we have started the Kafka server all the above directories have been created.

Step 7: Let us now check what information the *controller* file consists of.

```
get /controller
```

```
ls /
[admin, brokers, cluster, config, consumers, controller, controller_epoch, isr_c
hange_notification, latest_producer_id_block, log_dir_event_notification, zookee
per]
get /controller
{"version":1,"brokerid":0,"timestamp":"1591706876643"}
```

As you can see from the screenshot above, the *controller* file shows the *brokerid* as 0. The Broker with id 0 created this file on ZooKeeper and hence is the Controller Broker. It also shows the version and the timestamp at which it was created.

Step 8: Let us now check the contents of *brokers* directory.

```
ls /zookeeper/brokers
```

The screenshot below shows the contents of *brokers* directory.

```
ls /
[admin, brokers, cluster, config, consumers, controller, controller_epoch, isr_c
hange_notification, latest_producer_id_block, log_dir_event_notification, zookee
per]
get /controller
{"version":1,"brokerid":0,"timestamp":"1591706876643"}
ls /brokers
[ids, seqid, topics]
```

The *ids* folder contains the list of Broker ids that are the members of the Kafka cluster. We can check that using the *ls* command as shown below.

```
ls /zookeeper/ids
```

```
ls /
[admin, brokers, cluster, config, consumers, controller, controller_epoch, isr_c
hange_notification, latest_producer_id_block, log_dir_event_notification, zookee
per]
ls /brokers
[ids, seqid, topics]
ls /brokers/ids
[0]
```

As we can see from the screenshot above, there is a folder named *0* inside */brokers/ids* folder. This indicates that there is one broker as part of the Kafka cluster with id 0.

Task 2 is complete!

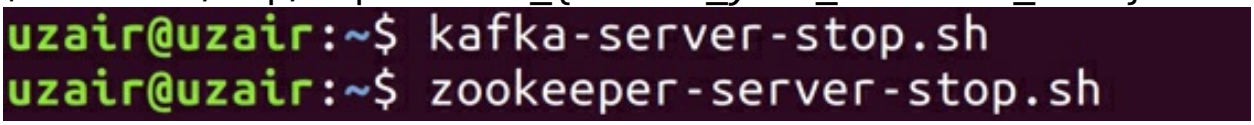
TASK 3: CONTROLLER BROKER ELECTION

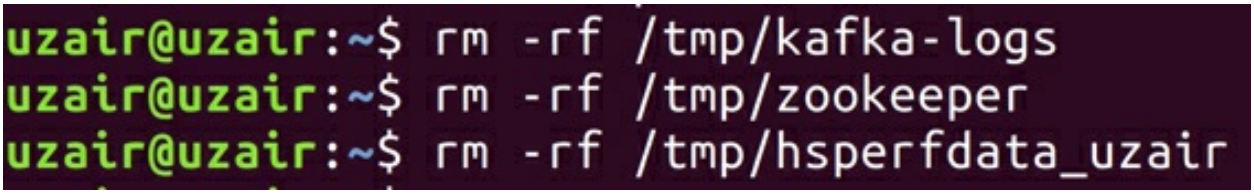
We have learned how a Controller Broker is elected using ZooKeeper in the theory. Let us now test that theory and see the election in real time.

Step 1: Let us create multiple Brokers in a single server so that we have multiple Brokers to demonstrate the election of Controller Broker. Please note that we are only creating multiple brokers in single server for training and demonstration purposes. Never create multiple Brokers in single server in production environment.

Stop the Kafka and ZooKeeper servers in order and also remove the data from *tmp* folder as shown below.

```
$ kafka-server-stop.sh
$ zookeeper-server-stop.sh
$ rm –rf /tmp/kafka-logs
$ rm –rf /tmp/zookeeper
$ rm –rf /tmp/hsperfdata_{insert_your_username_here}
```





Step 2: In a multi-node Kafka cluster, every server has its own *server.properties* configuration file. We have to start each Kafka server individually by referring to its *server.properties* configuration file. In standalone single Kafka server, we can create multiple *server.properties* configuration files to simulate a multi-node Kafka cluster.

We should first create three *server.properties* configuration files (*server1.properties, server1.properties and server2.properties)* and make sure the Broker id, port number and the *logs.dir* path is different in all the three configuration files.

In the terminal navigate to the following path and copy the *server.properties*

file 3 times as shown below.

```
$ cd /usr/share/kafka/config
```

```
$ sudo cp server.properties server1.properties
$ sudo cp server.properties server2.properties
$ sudo cp server.properties server3.properties
```

```
uzair@uzair:/usr/share/kafka/config$ ls
connect-console-sink.properties    log4j.properties
connect-console-source.properties  producer.properties
connect-distributed.properties     server1.properties
connect-file-sink.properties       server2.properties
connect-file-source.properties     server3.properties
connect-log4j.properties           server.properties
connect-mirror-maker.properties    tools-log4j.properties
connect-standalone.properties      trogdor.conf
consumer.properties                zookeeper.properties
uzair@uzair:/usr/share/kafka/config$
```

Step 3: Let's make sure the Broker id, port number and path to Kafka logs is different in all three *server.properties* configuration files.

Edit the *server1.properties* file to have the Broker id as 1, port number as 9092 and Kafka logs as */tmp/kafka-logs-1*.

```
$ sudo vi server1.properties
```

Press 'i' key to edit.

You should see the values as per the screenshot below.

```
############################# Server Basics #############################

# The id of the broker. This must be set to a unique integer for each broker.
broker.id=1

############################# Socket Server Settings #############################
##

# The address the socket server listens on. It will get the value returned from
# java.net.InetAddress.getCanonicalHostName() if not configured.
#   FORMAT:
#     listeners = listener_name://host_name:port
#   EXAMPLE:
#     listeners = PLAINTEXT://your.host.name:9092
listeners=PLAINTEXT://:9091
```

```
############################# Log Basics #############################

# A comma separated list of directories under which to store log files
log.dirs=/tmp/kafka-logs-1

# The default number of log partitions per topic. More partitions allow greater
# parallelism for consumption, but this will also result in more files across
# the brokers.
num.partitions=1
```

Make sure you remove '#" symbol before any of the properties highlighted above.

Press "esc" to come out of edit mode and save the file by pressing ':' and 'x' key.

Repeat the same process and edit the *server2.properties* file. Edit the Broker id to be 2, the port to be 9092 and Kafka logs path to */tmp/kafka-logs-2*. The file should have the values as highlighted in the screenshot below.

```
############################# Server Basics #############################

# The id of the broker. This must be set to a unique integer for each broker.
broker.id=2

############################# Socket Server Settings #############################
##

# The address the socket server listens on. It will get the value returned from
# java.net.InetAddress.getCanonicalHostName() if not configured.
#   FORMAT:
#     listeners = listener_name://host_name:port
#   EXAMPLE:
#     listeners = PLAINTEXT://your.host.name:9092
listeners=PLAINTEXT://:9092
```

```
############################# Log Basics #############################

# A comma separated list of directories under which to store log files
log.dirs=/tmp/kafka-logs-2
```

Finally, repeat the same process and edit the *server3.properties* file. Edit the Broker id to be 3, the port to be 9093 and Kafka logs path to */tmp/kafka-logs-3*. The file should have the values as highlighted in the screenshot below.

```
############################# Server Basics #############################

# The id of the broker. This must be set to a unique integer for each broker.
broker.id=3

############################# Socket Server Settings #############################
##

# The address the socket server listens on. It will get the value returned from
# java.net.InetAddress.getCanonicalHostName() if not configured.
#   FORMAT:
#     listeners = listener_name://host_name:port
#   EXAMPLE:
#     listeners = PLAINTEXT://your.host.name:9092
listeners=PLAINTEXT://:9093
```

```
############################# Log Basics #############################

# A comma separated list of directories under which to store log files
log.dirs=/tmp/kafka-logs-3
```

Step 4: Let us now start the ZooKeeper server and connect to the ZooKeeper server using ZooKeeper shell.

```
$ zkServer.sh start
```

```
uzair@uzair:/usr/share/kafka/config$ zkServer.sh start
/usr/bin/java
ZooKeeper JMX enabled by default
Using config: /usr/share/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
```

```
$zookeeper-shell localhost:2181
```

```
uzair@uzair:/usr/share/kafka/config$ zookeeper-shell.sh localhost:2181
Connecting to localhost:2181
Welcome to ZooKeeper!
JLine support is disabled

WATCHER::

WatchedEvent state:SyncConnected type:None path:null
```

Let us check the content that exists in ZooKeeper by running the *ls* command on root directory.

```
$ ls /
```

```
Welcome to ZooKeeper!
JLine support is disabled

WATCHER::

WatchedEvent state:SyncConnected type:None path:null
ls /
[admin, brokers, cluster, config, consumers, controller_epoch, isr_change_notifi
cation, latest_producer_id_block, log_dir_event_notification, zookeeper]
```

You might see the items in the root path as shown above, but there will not be any Brokers. We can *ls* on the */brokers/ids* path to check that there are no Brokers in Kafka cluster.

```
$ ls /brokers/ids
```

```
ls /brokers/ids
[]
```

Step 5: Let us now start the first broker. Run the following command from a new terminal. Do not close the ZooKeeper shell.

```
$ kafka-server-start.sh /usr/share/kafka/config/server1.properties
```

Since this is the first Broker that has been started in the cluster, it will try and succeed in creating the *controller* node in ZooKeeper. Since the *controller* node does not exist in ZooKeeper already, it becomes the Controller Broker.

You should see the *controller* file in ZooKeeper file system as shown in the screenshot below. All the Brokers when joined to this cluster, will put a ZooKeeper watch on this file so that ZooKeeper notifies them whenever a change happens to the file.

```
ls /
```

You Should also see the Broker with id 1 having membership in Kafka cluster.

```
ls /brokers/ids
```

```
ls /
[admin, brokers, cluster, config, consumers, controller, controller_epoch, isr_c
hange_notification, latest_producer_id_block, log_dir_event_notification, zookee
per]
ls /brokers/ids
[1]
```

Let us see if the Broker 1 is the controller broker.

```
get /controller
```

```
get /controller
{"version":1,"brokerid":1,"timestamp":"1591725341510"}
```

Let us also check the content in the */brokers/ids/1* file.

```
get /brokers/ids/1
```

```
get /brokers/ids/1
{"listener_security_protocol_map":{"PLAINTEXT":"PLAINTEXT"},"endpoints":["PLAINT
EXT://uzair:9091"],"jmx_port":-1,"host":"uzair","timestamp":"1591725340568","por
t":9091,"version":4}
```

This file contains the endpoints (ip address), port number, host and other information about the Broker.

Let us finally check if the path to store Kafka logs has been created for Broker 1. Open a new terminal and run the command as shown below.

```
$ ll /tmp
```

As you can see from the screenshot, the path for Kafka logs has been successfully created.

```
uzair@uzair:~$ ll /tmp
total 188
drwxrwxrwt 17 root  root   4096 Jun  9 18:07 ./
drwxr-xr-x 23 root  root   4096 May 20 03:09 ../
-rw-------  1 uzair uzair     0 Jun  9 09:47 config-err-K2mN33
drwxrwxrwt  2 root  root   4096 Jun  9 09:44 .font-unix/
-rw-------  1 uzair uzair 60186 Jun  9 18:07 GchQtVdSv0
drwxr-xr-x  2 uzair uzair  4096 Jun  9 17:55 hsperfdata_uzair/
drwxrwxrwt  2 root  root   4096 Jun  9 09:47 .ICE-unix/
drwxr-xr-x  2 uzair uzair  4096 Jun  9 17:55 kafka-logs-1/
drwx------  2 uzair uzair  4096 Jun  9 09:47 ssh-y9z2sN6usM6H/
```

Step 6: Let us now start the second Broker.

```
$ kafka-server-start.sh /usr/share/kafka/config/server2.properties
```

Go back to the ZooKeeper shell and check if the new Broker has joined the Kafka cluster.

```
ls /brokers/ids
```

```
ls /brokers/ids
[1, 2]
```

A path for Broker 2 should also be created in the *tmp* directory as shown below.

```
$ ll /tmp
```

```
uzair@uzair:~$ ll /tmp
total 96
drwxrwxrwt 18 root  root  4096 Jun  9 18:21 ./
drwxr-xr-x 23 root  root  4096 May 20 03:09 ../
-rw-------  1 uzair uzair    0 Jun  9 09:47 config-err-K2mN33
drwxrwxrwt  2 root  root  4096 Jun  9 09:44 .font-unix/
drwxr-xr-x  2 uzair uzair 4096 Jun  9 18:16 hsperfdata_uzair/
-rw-------  1 uzair uzair 9502 Jun  9 18:21 hUdqMgwKck
drwxrwxrwt  2 root  root  4096 Jun  9 09:47 .ICE-unix/
drwxr-xr-x  2 uzair uzair 4096 Jun  9 17:55 kafka-logs-1/
drwxr-xr-x  2 uzair uzair 4096 Jun  9 18:16 kafka-logs-2/
drwx------  2 uzair uzair 4096 Jun  9 09:47 ssh-y9z2sN6usM6H/
```

Step 7: Let us finally start Broker 3.

```
$ kafka-server-start.sh /usr/share/kafka/config/server3.properties
```

Go back to the ZooKeeper shell and check if the new Broker has joined the Kafka cluster.

```
ls /brokers/ids
```

```
ls /brokers/ids
[1, 2]
ls /brokers/ids
[1, 2, 3]
```

Every time a new Broker is started, a new file is created in *ids* directory and the file name is same as the Broker id.

A path for Broker 3 should also be created in the *tmp* directory as shown below.

```
$ ll /tmp
```

```
uzair@uzair:/usr/share/kafka_2.12-2.5.0/config$ ll /tmp
total 116
drwxrwxrwt 19 root  root   4096 Jun  9 18:29 ./
drwxr-xr-x 23 root  root   4096 May 20 03:09 ../
-rw-------  1 uzair uzair     0 Jun  9 09:47 config-err-K2mN33
drwxrwxrwt  2 root  root   4096 Jun  9 09:44 .font-unix/
drwxr-xr-x  2 uzair uzair  4096 Jun  9 18:28 hsperfdata_uzair/
drwxrwxrwt  2 root  root   4096 Jun  9 09:47 .ICE-unix/
drwxr-xr-x  2 uzair uzair  4096 Jun  9 17:55 kafka-logs-1/
drwxr-xr-x  2 uzair uzair  4096 Jun  9 18:16 kafka-logs-2/
drwxr-xr-x  2 uzair uzair  4096 Jun  9 18:28 kafka-logs-3/
-rw-------  1 uzair uzair 19212 Jun  9 18:29 LPb3pjHWQp
```

Step 8: Let us now create a Topic and see how the partitions are distributed within the 3 brokers. Let us create a Topic called *twitter* with 3 partitions and replication factor as 1.

```
$ kafka-topics.sh --zookeeper localhost:2181 --create --topic twitter --replication-factor 1 --partitions 3
```

```
uzair@uzair:~$ kafka-topics.sh --zookeeper localhost:2181 --create --topic twitt
er --replication-factor 1 --partitions 3
Created topic twitter.
```

Let us check the Kafka logs directory for each Broker.

```
$ ll /tmp/kafka-logs-1
```

```
uzair@uzair:~$ ll /tmp/kafka-logs-1
total 20
drwxr-xr-x  3 uzair uzair 4096 Jun  9 18:36 ./
drwxrwxrwt 19 root  root  4096 Jun  9 18:31 ../
-rw-r--r--  1 uzair uzair    0 Jun  9 17:55 cleaner-offset-checkpoint
-rw-r--r--  1 uzair uzair    0 Jun  9 17:55 .lock
-rw-r--r--  1 uzair uzair    0 Jun  9 17:55 log-start-offset-checkpoint
-rw-r--r--  1 uzair uzair   88 Jun  9 17:55 meta.properties
-rw-r--r--  1 uzair uzair    0 Jun  9 17:55 recovery-point-offset-checkpoint
-rw-r--r--  1 uzair uzair   16 Jun  9 18:36 replication-offset-checkpoint
drwxr-xr-x  2 uzair uzair 4096 Jun  9 18:36 twitter-1/
```

```
$ ll /tmp/kafka-logs-2
$ ll /tmp/kafka-logs-3
```

```
uzair@uzair:~$ ll /tmp/kafka-logs-2
total 20
drwxr-xr-x  3 uzair uzair 4096 Jun  9 18:36 ./
drwxrwxrwt 19 root  root  4096 Jun  9 18:31 ../
-rw-r--r--  1 uzair uzair    0 Jun  9 18:16 cleaner-offset-checkpoint
-rw-r--r--  1 uzair uzair    0 Jun  9 18:16 .lock
-rw-r--r--  1 uzair uzair    0 Jun  9 18:16 log-start-offset-checkpoint
-rw-r--r--  1 uzair uzair   88 Jun  9 18:16 meta.properties
-rw-r--r--  1 uzair uzair    0 Jun  9 18:16 recovery-point-offset-checkpoint
-rw-r--r--  1 uzair uzair   16 Jun  9 18:36 replication-offset-checkpoint
drwxr-xr-x  2 uzair uzair 4096 Jun  9 18:36 twitter-2/
uzair@uzair:~$ ll /tmp/kafka-logs-3
total 20
drwxr-xr-x  3 uzair uzair 4096 Jun  9 18:36 ./
drwxrwxrwt 19 root  root  4096 Jun  9 18:31 ../
-rw-r--r--  1 uzair uzair    0 Jun  9 18:28 cleaner-offset-checkpoint
-rw-r--r--  1 uzair uzair    0 Jun  9 18:28 .lock
-rw-r--r--  1 uzair uzair    0 Jun  9 18:28 log-start-offset-checkpoint
-rw-r--r--  1 uzair uzair   88 Jun  9 18:28 meta.properties
-rw-r--r--  1 uzair uzair    0 Jun  9 18:28 recovery-point-offset-checkpoint
-rw-r--r--  1 uzair uzair   16 Jun  9 18:36 replication-offset-checkpoint
drwxr-xr-x  2 uzair uzair 4096 Jun  9 18:36 twitter-0/
uzair@uzair:~$
```

This is how the data is distributed across the cluster in a Kafka cluster.

Step 9: Let us stop the Controller Broker *i.e.*, Broker 1 to simulate failure and see which Broker gets successful in creating the *controller* file in ZooKeeper to become the Controller Broker.

You can either press Ctrl + C keys or run the stop command to shut down the Kafka Broker 1.

After the Broker 1 is shut down, go back to the ZooKeeper shell and check the list of available Brokers.

```
ls /brokers/ids
```

You should see that Broker 1 is no longer the member of Kafka cluster.

Now, check the contents of *controller* file to see which Broker was successful in becoming the Controller Broker.

As you can see from the screenshot above, Broker 2 is now the Controller Broker.

When Broker 1, *i.e.*, the previous Controller Broker was down, the *controller* file gets deleted in ZooKeeper. Since all the other Brokers subscribe to this file, they get notified about the deletion of file, *i.e.*, the Controller Broker failure. All the Brokers will once again try to create the *controller* file. Whichever Broker succeeds in creating the *controller* file becomes the Controller Broker. This entire process happens in milliseconds ensuring the high availability of Controller Broker.

You may try shutting down the Broker 2 to check if the Broker 3 will become the Controller Broker.

Task 3 is complete!

SUMMARY

In this chapter, we have looked at Topic operations, Topic Overview, Data model in ZooKeeper, ZooKeeper watches, Election and responsibilities of Controller Broker. We have seen how the Topics receive messages from Producers. We have also seen how ZooKeeper helps Kafka in Controller Broker election ensuring high availability and coordination.

In the labs, we have performed, Topic operations, had hands-on the ZooKeeper shell and demonstrated the Controller Broker election.

CHAPTER 4:
KAFKA IN-DEPTH
PART II

In the previous chapter, we had discussed the internals of the following Topics: ZooKeeper, Election, and responsibilities of the Controller Broker. In this chapter, we describe the remaining Kafka components comprehensively.

Let us learn the following internals of Kafka in this chapter.

- Replications
- Partitions
- Bootstrap Server

REPLICATIONS

Replication in Kafka plays an important role in the durability of data. It guarantees that the data and/or messages are not lost in case of a Broker failure. As discussed in the previous chapters, messages are classified in the form of Topics and stored in the Brokers. A Topic is further divided into multiple Partitions, and each Partition can contain any number of replications. The durability of data is ensured by storing the replicas of a Partition across multiple Brokers; hence, an individual Broker failure does not lead to the data loss. Let us understand the concept of replication with the following example:

- Consider a Kafka cluster containing six Brokers and a RFID sensor application as Producer that sends messages to the Brokers in the cluster.

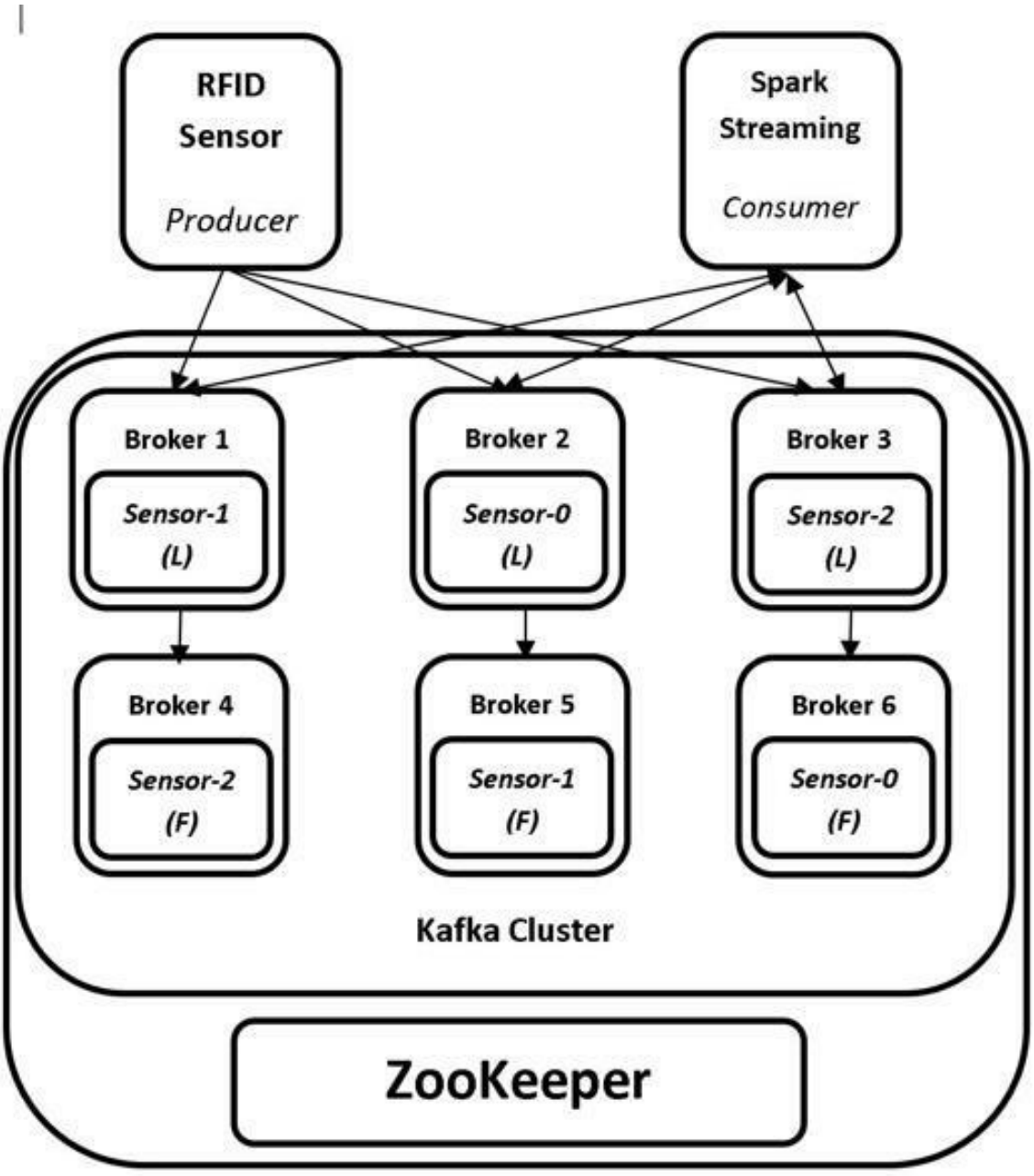
- The producer sends sensor messages to a Topic called *sensor*, which contains three partitions with a replication factor of two.

For creating this particular *sensor* Topic, the following details are specified:

Topic – sensor
Partitions – 3
Replication factor – 2

After the creation of the Topic *sensor*, the partitions and replications are automatically stored in the Brokers, as depicted in Figure 4(a). It can be observed that there are three partitions with a replication factor of two, and they are stored in six different Brokers.

- There are two types of replicas in Kafka: they are termed as, the *Leader replica* and the *Follower replica*. When a new Topic (like *sensor*) is created, Kafka automatically elects a partition as *Leader* replica using its leader election algorithm. The first replica is elected as the *Leader* from a list of replicas.



4(a) Kafka Cluster with Replications

- Let us consider that the replicas in Brokers 1, 2, and 3 are elected as the *Leader* replicas, whereas, the replicas in Brokers 4, 5, and 6 are the *Follower* replicas as depicted in Figure 4(a). The *Leader* replicas are denoted with 'L', whereas the *Follower* replicas are denoted with 'F'.

- All the communication between the Producer and Consumer occurs through those Brokers that host the *Leader* replicas. In our example, the Producer sends data to Brokers 1, 2, and 3 *i.e.,* the Brokers hosting the *Leader* replicas. The Producer does not send the data to the Brokers 4, 5, and 6 hosting the *Follower* replicas. Similarly, the Consumer requests the data from the Brokers 1, 2, and 3 and not from the Brokers 4, 5, and 6.

- The Brokers containing replicas of Partitions that are not the *Leader* replicas hold the *Follower* replicas. The *Follower* replicas act similar to the Consumers, but internally within the Kafka cluster. The *Follower* replicas request data from the Brokers hosting the *Leader* replicas and replicate all the messages from the Brokers holding the *Leader* replicas. Hence, the Brokers containing *Follower* replicas do not communicate with the Producers or Consumers. Their job is to replicate the data received by the Brokers that contain *Leader* replicas and update the newly received messages.

In our example above, Broker 6 (hosting a *Follower* replica) sends a request to Broker 2 (hosting a *Leader* replica) to replicate the data in Broker 2. Similarly, Brokers 4 and 5 request Brokers 1 and 3 for replicating the data. This helps the *Follower* Brokers to maintain the current state of data to that of *Leader* Brokers for an overall data durability.

- The Brokers can also host both a *Leader* replica and a *Follower* replica. There is no such constraint that a Broker hosting a *Leader* replica cannot host a *Follower* replica. In the aforementioned example, the Brokers 1, 2, and 3 can also host the *Follower* replica partitions of other Topics, and similarly, the Brokers 4, 5, and 6 can host *Leader* replica partitions of other Topics. A Broker can also be termed as a *Leader* Broker or a *Follower* Broker with respect to the Topic's Partition it is currently hosting.

- The *Leader* Broker is also responsible to keep track of the current state of data being replicated by its *Followers*. In this manner, the *Followers* have an up-to-date data replicated with the help of the *Leader*. However, there could be multiple scenarios, where the *Followers* are not in sync with *Leader* due to network and hardware issues. In order to tackle this issue, the following mechanism is invoked:

    - If a *Follower* does not send a fetch request to the *Leader* for more than 10 seconds or the message sent by the *Leader* has not been acknowledged by the *Follower*, it is tagged as an *out of sync Follower* with the *Leader*. Therefore, this *out of sync Follower* replica cannot be upgraded to become a new *Leader* replica in case of the failure of the current *Leader*.

    - On the other hand, the *Follower* replica that maintains the updated state of data as that of the *Leader* replica is termed as an *In-Sync Replica* (ISR). The ISR is eligible to become a new *Leader* in the event of current *Leader* failure. The maximum amount of time a *Follower* replica can lag before it is considered *out of sync* can be controlled by the *replica.lag.time.max.ms* configuration property. By default, this time is set as 10 seconds.

- In the event of Broker failure that hosts the *Leader* replica, the Controller Broker gets notified about the failure using ZooKeeper (Please refer to Chapter 3 for more information about ZooKeeper and Controller Broker). The Controller Broker contains all the metadata information regarding the *Leader* and *Follower* replicas. Subsequently, the Controller Broker looks for a *Follower* replica that was *in-sync* with *Leader* prior to its failure. Once it verifies the *in-sync* replica, it is made the new *Leader* replica. This simple process ensures a high availability of the Kafka cluster.

- In our example, let us assume that Broker 2 that hosts the *sensor-0* Partition replica is down due to a hardware failure. The Controller Broker that keeps a watch (*subscribe*) on Brokers' IDs directory will be notified about the failure as the Broker ID 2 node will be deleted. Subsequently, the Controller Broker with the help of metadata information will search for a *Follower* copy that is *in-sync* with the *Leader*. In this case, it is the Broker 6 that hosts the *Follower* replica of *sensor-0,* and hence it will be promoted as the *Leader* replica.

- This scenario entails a logical query: With Broker 6 now hosting the *Leader* replica, how does the Producer know that the Broker 2 had failed earlier, and Broker 6 now hosts the *Leader* replica? The Producer will generate the data and send it to Broker 2 as usual. But this time the Producer will receive an exception called *LeaderNotAvailable*. The Producer then contacts the Controller Broker requesting the metadata, which responds to the metadata request of the Producer by notifying about the newly-elected Broker hosting the *Leader* replica. Therefore, the Producer starts sending data to Broker 6. This ensures that the communication between the components of Kafka cluster is always integrated and available.

- In our example, we have used a replication factor of 2 for simplicity. However, the recommended replication factor at which the partitions should be replicated is 3.

To summarize, replication helps in achieving the high availability of communication and the durability of data between Kafka components.

PARTITIONS

Partitions serve as the unit of storage for messages in Kafka. As discussed earlier, a Topic can be divided into any number of Partitions. These Partitions cannot be split across Brokers in a Kafka cluster or within the disks on the same Broker. Hence the maximum size of a Partition cannot exceed the size of the disk. Therefore, the number of Partitions is specified while

creating a Topic. The path at which the Partitions will be stored is specified in the configurations file, as we have discussed in the previous chapters.

But how does Kafka decide which Partition should be stored in which Broker? To answer this question, the following section thoroughly describes how partitions are assigned in Kafka.

Assigning Partitions

After a Topic is created, Kafka allocates partitions to the available Brokers in the cluster. Consider the example from the previous section. We created a Topic containing 3 Partitions with a replication factor of 2. This translates to a total of six partition replicas on six Brokers. Kafka makes sure the following conditions are met while assigning these Partitions.

1. All the Partition replicas for a given Topic are distributed evenly among the list of Brokers. Each Partition is assigned per Broker in case of our example.

2. Replicas of the same Partitions are not assigned to the same Broker. In case of our example, sensor-*0* Partition is assigned to Broker 2, whereas the other replica Partition is allocated to Broker 5, as depicted in Figure 4(a). Kafka does not assign both the *Leader* replica and *Follower* replica to the same Broker. In this way, it assures that the *Follower* replica in Broker 5 is still available if Broker 2 is down due to network issues or hardware failure, *etc*.

3. Kafka can also be configured for the rack awareness, if available. If the Kafka cluster spans multiple racks, the Partitions can be assigned in Brokers available on different racks. Assigning Partitions with rack awareness ensures that in the case of a rack failure, the data is still available in the Brokers on other racks.

The *Leader* replica of a Partition is first assigned by selecting a random Broker in the Kafka cluster. Subsequently, the rest of the *Leader* Partitions are assigned in a round-robin fashion. The *Follower* replicas are allotted with the increased offsets from their *Leader*. For example, if a *Leader* replica is

assigned to Broker 3, its *Follower* replicas will be assigned to Broker 4, Broker 5, and so on.

The above approach does not apply, when rack-awareness is available for the Kafka cluster. In case of rack-awareness, a Broker is selected from each rack alternatively. If a cluster contains of a total of 10 Brokers, with an arrangement of 5 Brokers on each rack, a Broker list for the rack awareness is prepared as 0, 5, 1, 6, 2, 7, 3, 8, 4, and 9. This list contains alternating Brokers from rack 1 and rack 2. Hence if a *Leader* replica of a Partition is present on Broker 6, its replica will be on Broker 2. This also means that the *Leader* replica is on a Broker in rack 1 and its *Follower* replica is on a Broker in rack 2. This approach ensures that in the event of rack 1 failure, the *Follower* replica is still available in the alternate rack 2.

BOOTSTRAP SERVER

Bootstrap servers are a list of Brokers utilized by the clients (Producers and Consumers) to establish an initial connection with the Kafka cluster. Once the connection is established, the clients acquire the full list of Brokers available in that cluster using the metadata information, which is shared with every Broker in the cluster by the Controller Broker. The client then looks for the Brokers hosting the Partitions for their specific Topic using the metadata information, and starts communicating with the corresponding Brokers.

Let us understand this better with the following example:

- Consider a Kafka cluster with six Brokers and one Producer that generates the sensor data. Moreover, consider a Topic with three Partitions and a replication factor of 1.

- The Producer now starts generating the messages. But how does the Producer know about the Brokers which host the corresponding Partitions it has to send the messages to? This is where the Bootstrap servers play their role.

- In the Producer code, we need to specify a list of host:port pairs of Brokers in the configuration property called *bootstrap.servers*.

- The Producer utilizes the first host:port pair of the Broker to establish an initial connection with the Kafka cluster.

- The Producer then requests the Broker to send it the corresponding metadata information of the Kafka cluster. As a result, the Broker responds with the required metadata information.

- The client then looks up for the Brokers containing the Partitions related to the relevant Topic and starts generating messages to the Partitions.

- But how many Broker host:port pairs should be provided as the value for *bootstrap.servers* configuration property? It is recommended to provide at least two Broker host:port pairs. This is to ensure that the client is always able to connect with the Kafka cluster, even if one of the Brokers is down for some reason.

In the labs, we shall be looking at an example Producer code to practically understand how the *bootstrap.servers* configuration property is specified.

The theory of this chapter concludes here. Let us now proceed to the lab exercise to install IntelliJ IDEA and Scala language to our virtual machines. This exercise will serve as a platform when we start writing Producer and Consumer code in the succeeding chapters.

AIM OF LAB EXERCISE 4

The primary aim of the following lab exercises is to download and install IntelliJ IDEA and Scala. Subsequently, we will then learn how to specify Bootstrap servers in the Producer code.

The labs for this chapter include the following exercises:

- Download and Install Scala
- Download and Install IntelliJ IDEA
- Configuring IntelliJ IDEA
- Specifying Bootstrap servers

We need the following packages to perform the lab exercise:

- Java Development Kit (JDK)
- Apache ZooKeeper
- Apache Kafka
- Scala
- IntelliJ IDEA

LAB EXERCISE 4: KAFKA IN-DEPTH – PART II

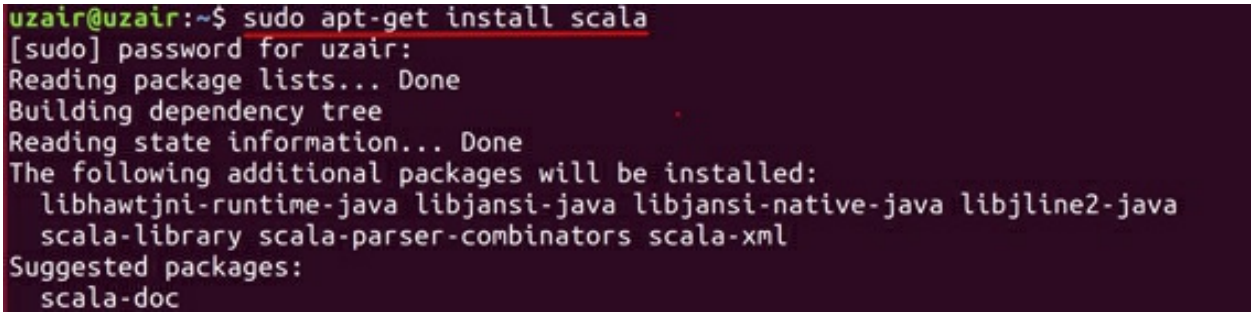
1. **Download & Install Scala**
2. **Download & Install IntelliJ IDEA**
3. **Configuring IntelliJ IDEA**
4. **Specifying Bootstrap Servers**

TASK 1: DOWNLOAD AND INSTALL SCALA

Let us first install Scala, as we will be writing the Producer and Consumer code in Scala in the upcoming lab exercises.

Step 1: Run the following command from the terminal to install Scala.

```
$ sudo apt-get install scala
```



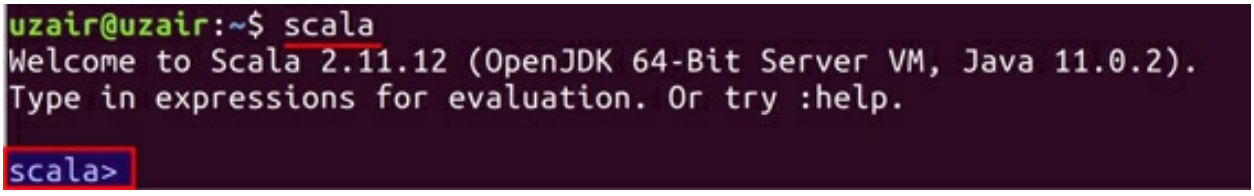
The command prompt will ask you to hit 'Y' after running the above command as shown in the screenshot above. Therefore, please hit 'Y' from your keyboard to continue with the installation, and finally hit the Enter key to proceed further.

Step 2: Verify the Scala installation version by running the following command:

```
$ scala -version
```

Step 3: After the installation is completed successfully, type *scala* in your terminal and a *Scala* prompt will appear as shown below.

```
$ scala
```

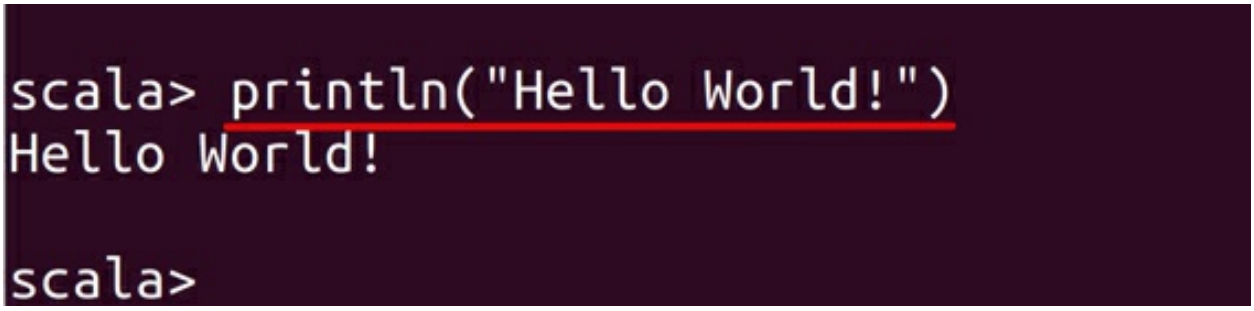


These simple steps complete the *Scala* installation. The *Scala* prompt is an interactive shell, where you can write and run the *Scala* code. This

interactive shell is also known as REPL.

Step 4: Using the REPL, you can now start writing the *Scala* code! Let's start by printing the classic "Hello world!" from the shell. To do this, simply type the following code and hit Enter on your keyboard.

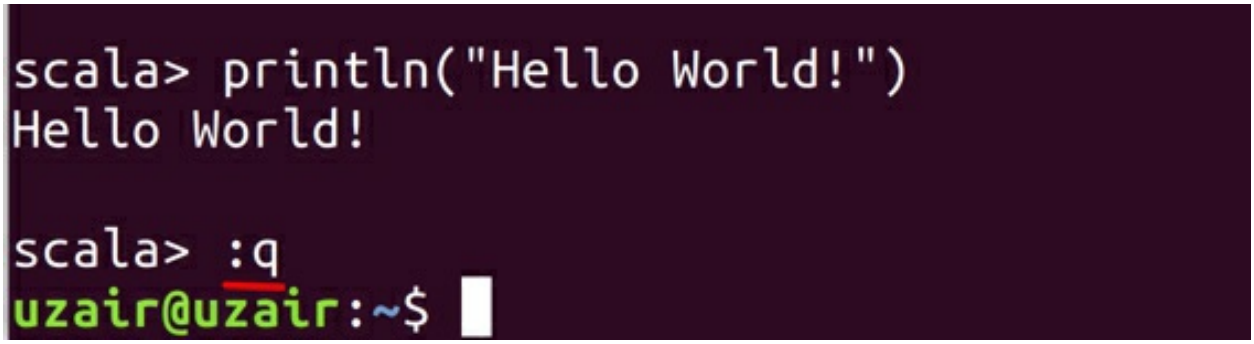
```
scala> println("Hello World!")
```



As you can observe from the prompt screenshot, the output is displayed below immediately, as soon as you hit the Enter button.

Step 5: To quit the Scala REPL, you can use the following command:

```
scala> :q
```



This will take you back to the terminal prompt.

Task 1 is complete!

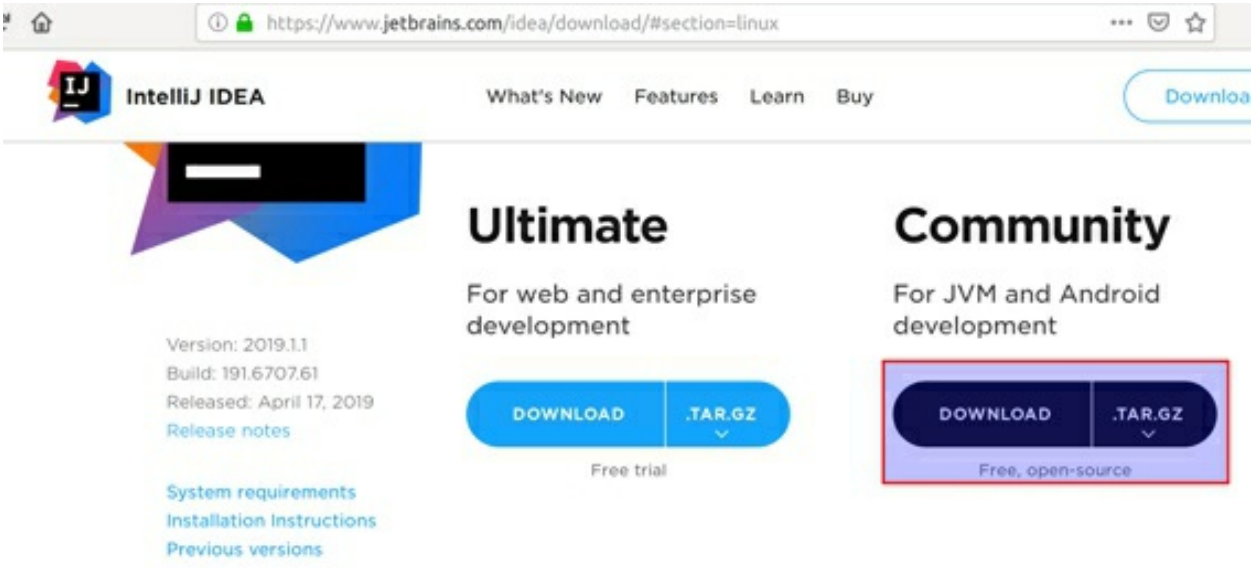
TASK 2: DOWNLOAD AND INSTALL INTELLIJ IDEA

We shall be using IntelliJ IDEA to write the client code throughout the lab exercises. In Task 2, we shall download and install the IntelliJ IDEA and then install the *Scala* plugin, so that we can write the Scala code.

Step 1: Navigate to the following URL from your web browser and click on

the "*Download*" button for the *Community* edition as depicted in the screenshot below.

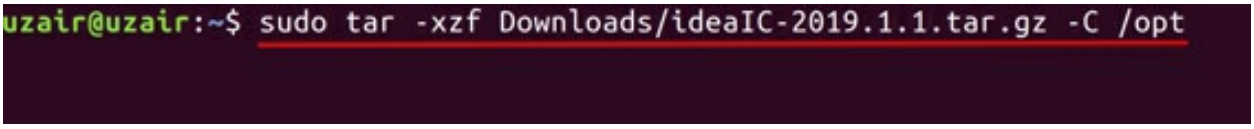
http://bit.ly/2V1HFYO



The file shall download automatically to the *Downloads* folder. This download might take time, depending upon the speed of your internet connection.

Step 2: Once the download is complete, open the terminal and run the following commands to untar the package. We shall be extracting the tar ball to the /opt directory.

```
$ sudo tar –xzf Downloads/ideaIC-2019.1.1.tar.gz –C /opt
```



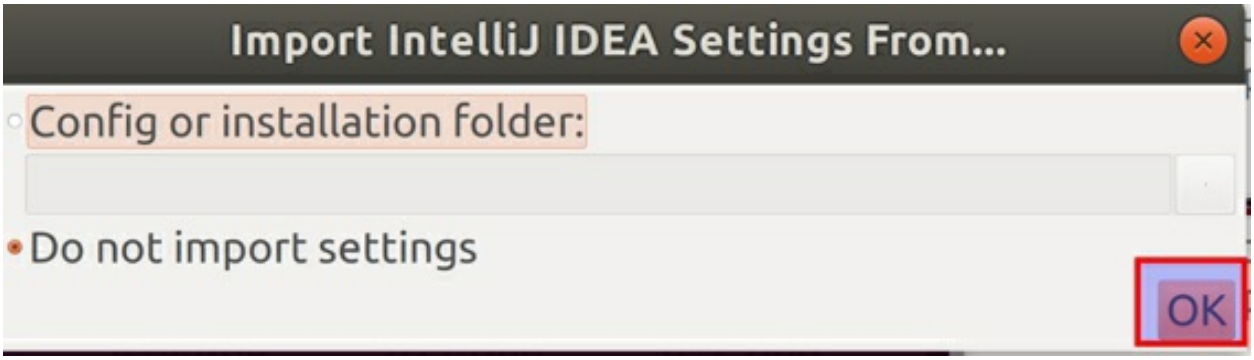
Please note that your downloaded version of IntelliJ might be different. Therefore, please type the version correctly in the above command.

Step 3: Now run the following command to install IntelliJ.

```
$/opt/idea-IC-191.6707.61/bin/idea.sh
```

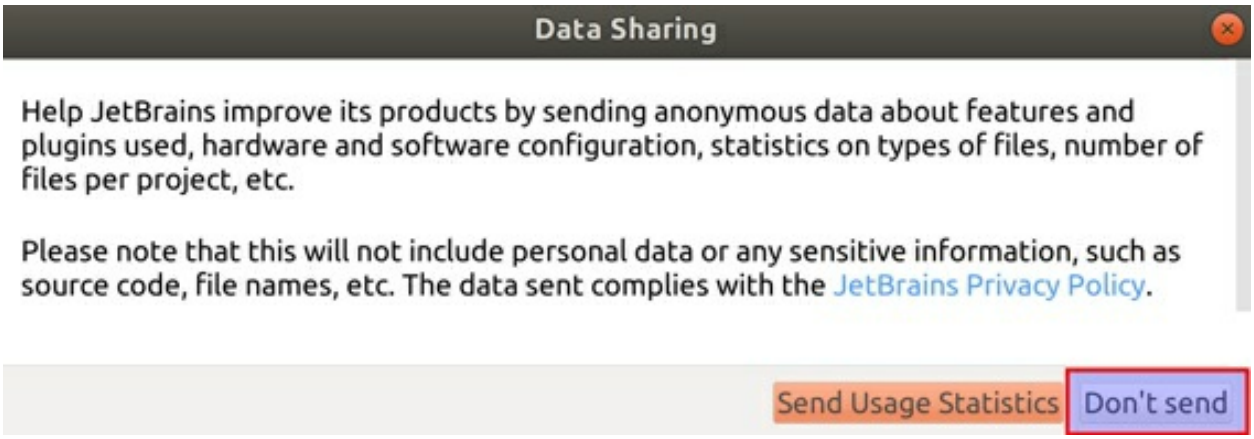
Please note that your path or the version might be different from the ones given in the command.

You should now see a prompt asking to import the settings. Simply click “OK”.

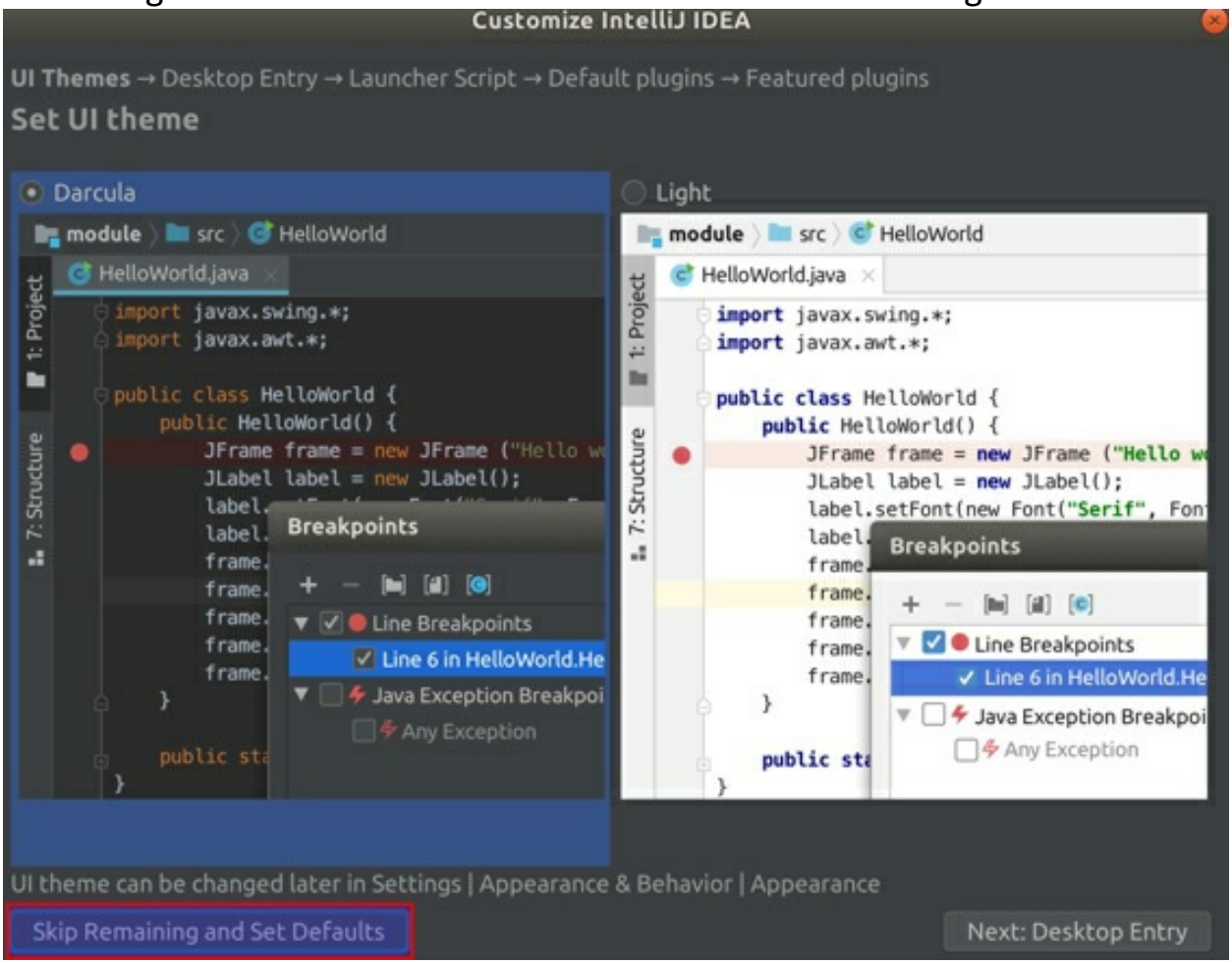


Step 4: You should now be prompted with a *Privacy Policy* window. Click on the check box to accept the policy and click on the “*Continue*” button as depicted in the screenshot.

In the next prompted window, please click on the “*Don’t Send*” button.



Step 5: You will now be prompted to select a theme. Therefore, please select a theme according to your own comfort and click on the "Skip remaining and Set Defaults" button as shown in the following screenshot.

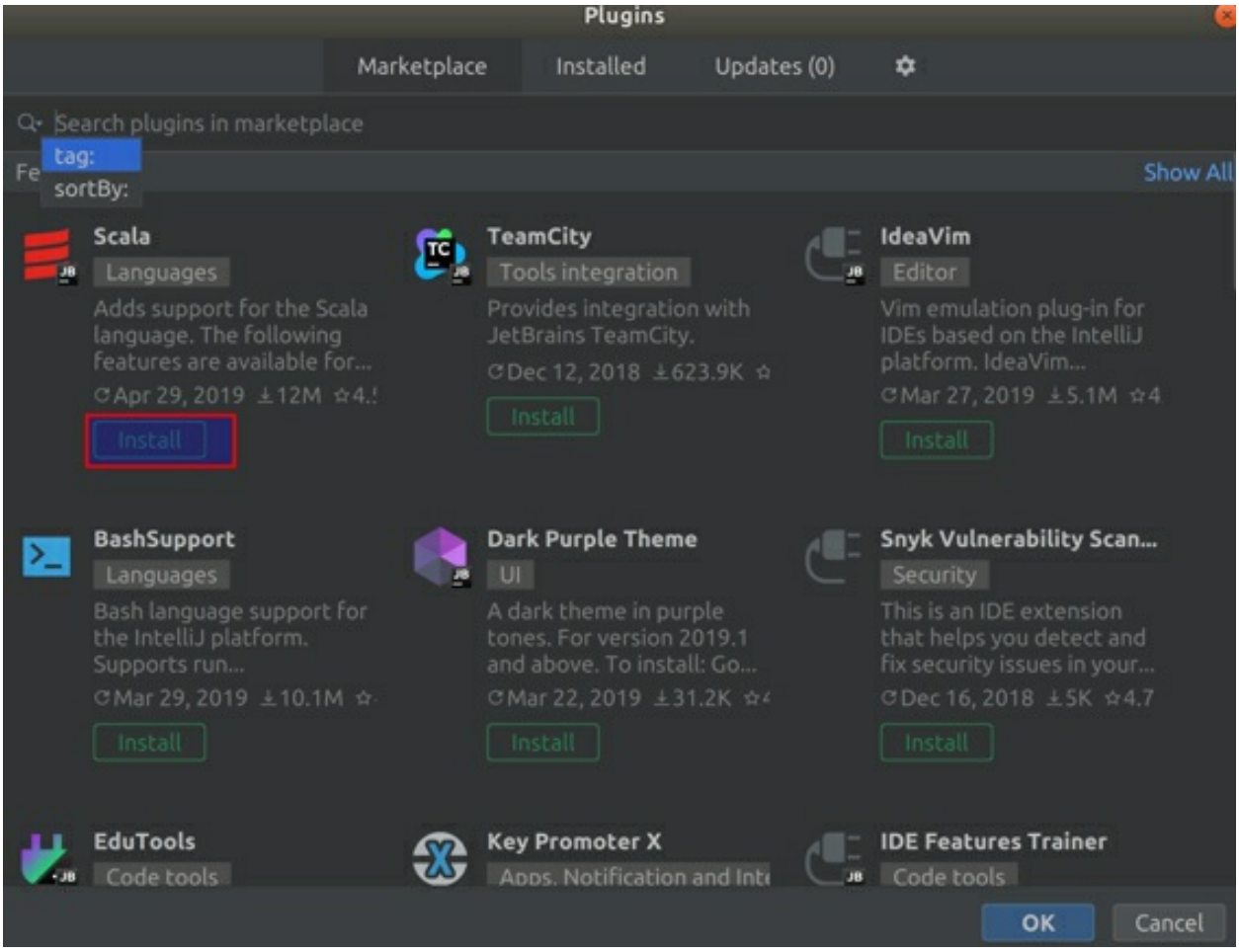


You shall then be able to see the Welcome screen as shown below.

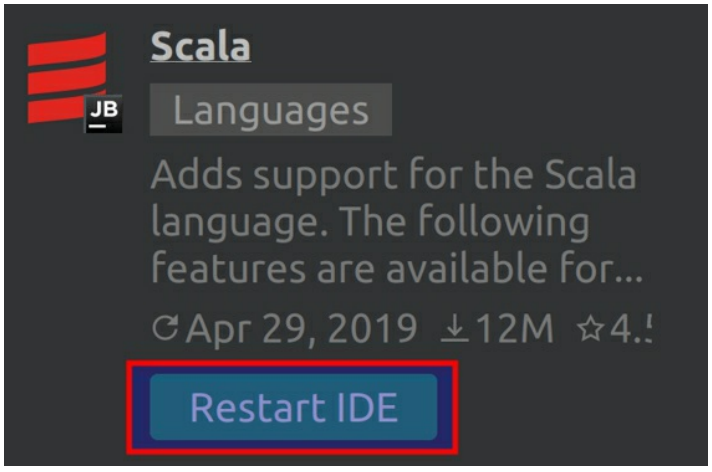
This concludes the installation of IntelliJ IDEA. But to run the Spark *Scala* code, we need to install the *Scala* plugin.

Step 6: Click on the “Configure” button as shown in the screenshot below and click on the “Plugins” in the dropdown menu.

Step 7: The marketplace for the plugins will then be opened. Click on the "Install" button for "Scala" plugin as shown in the screenshot. If you do not see "Scala" right away, search for "Scala" in the search bar above and then click the "Install" button.



This action should begin the download. Once it is downloaded, you will be requested to Restart IDE. Please click on the “Restart IDE” button. Click “Restart” in the confirmation pop-up.

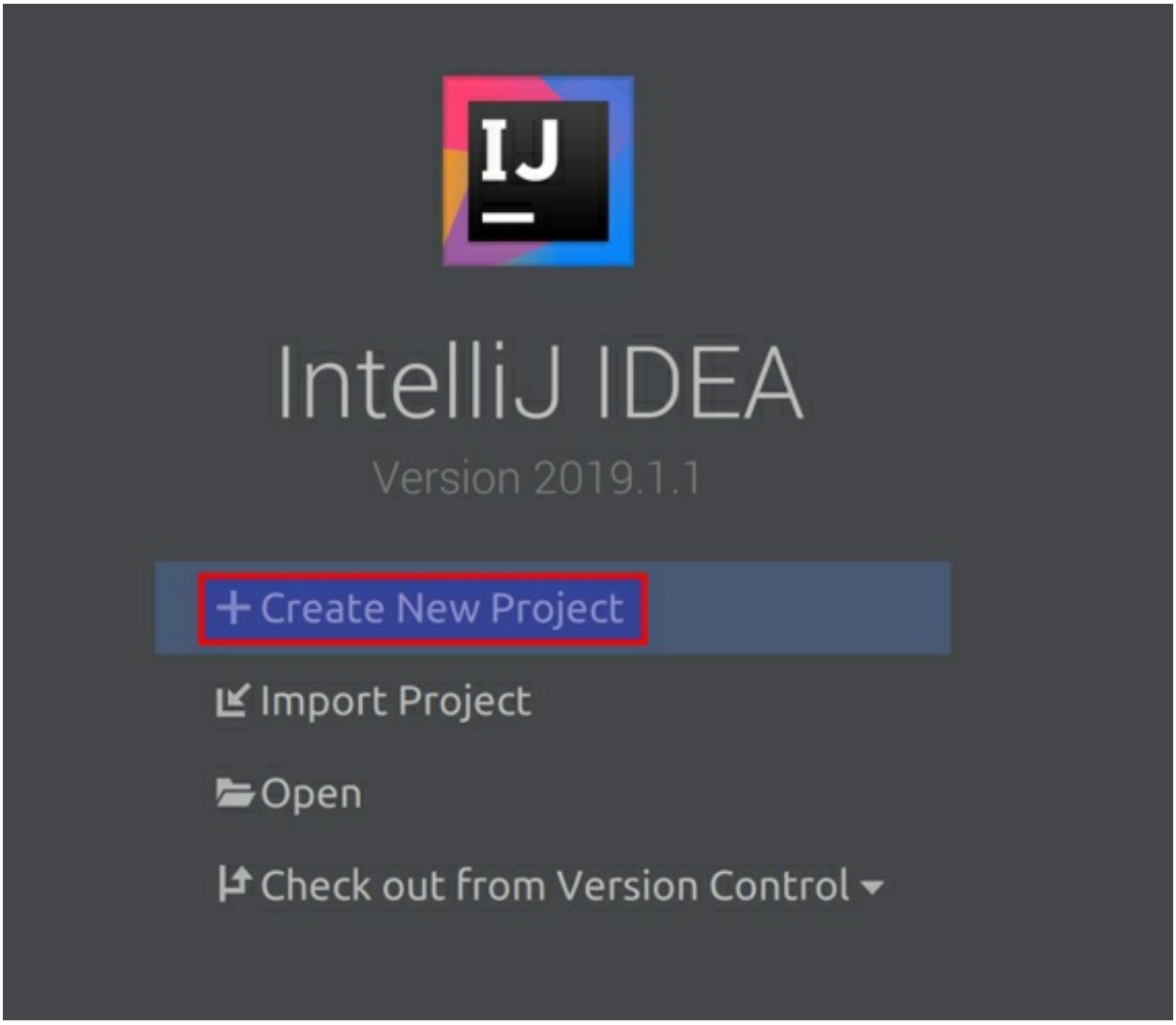


The IDE will now restart and show the welcome screen once again. With this, you have successfully installed the IntelliJ IDEA with the Scala plugin.

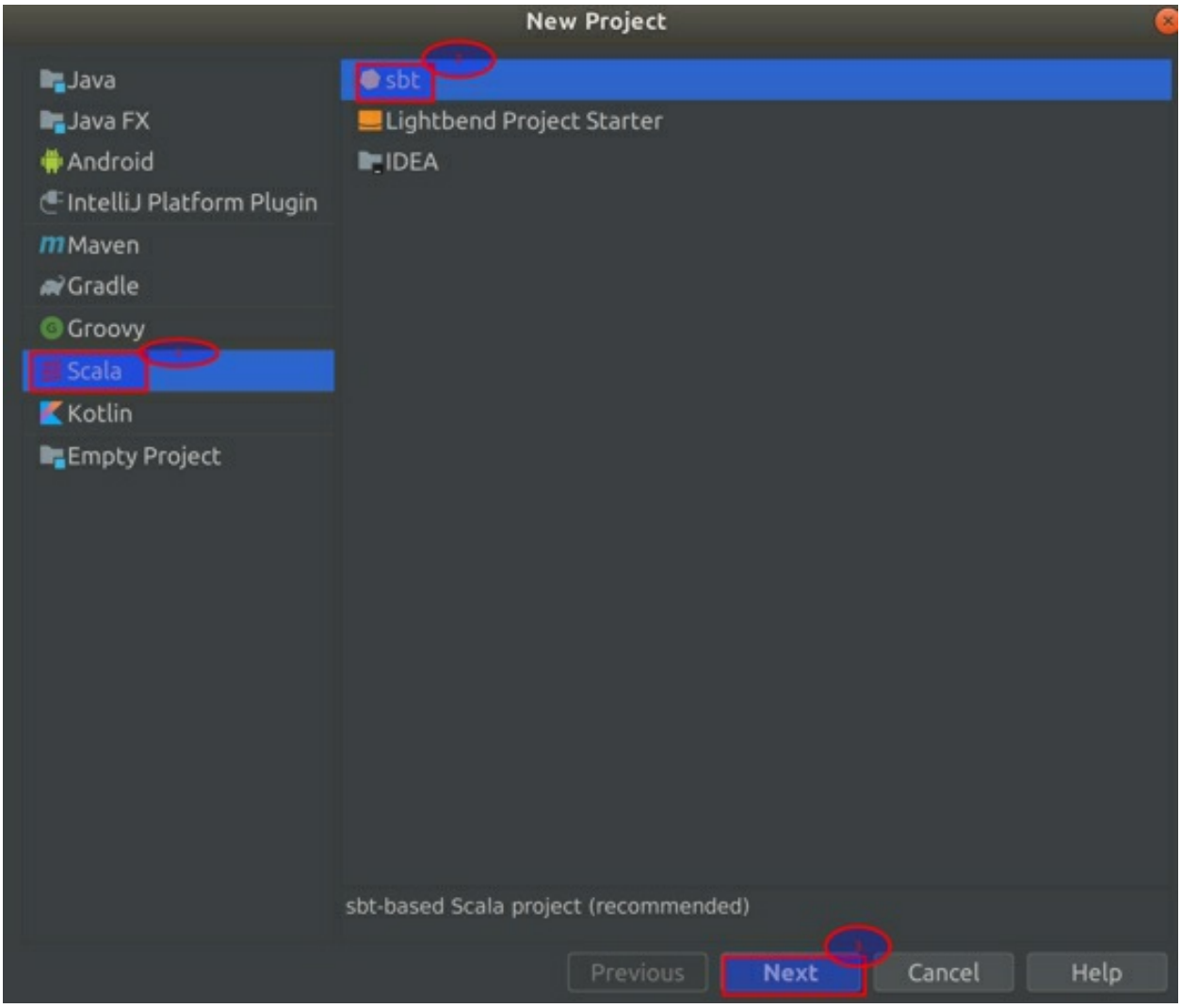
Task 2 is complete!

TASK 3: CONFIGURING INTELLIJ IDEA

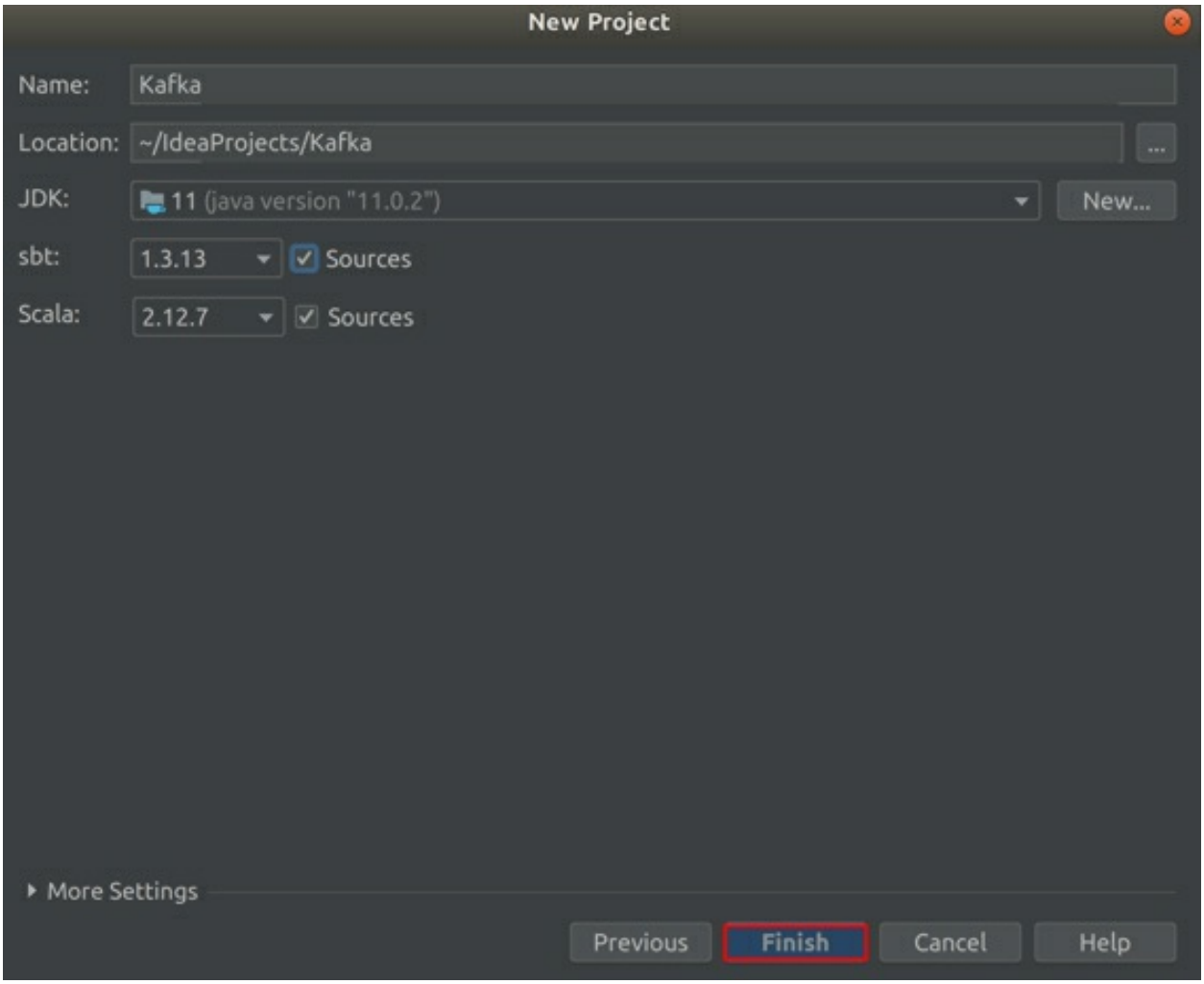
Step 1: Click on the “Create New Project” button on the welcome screen, as shown in the screenshot below.



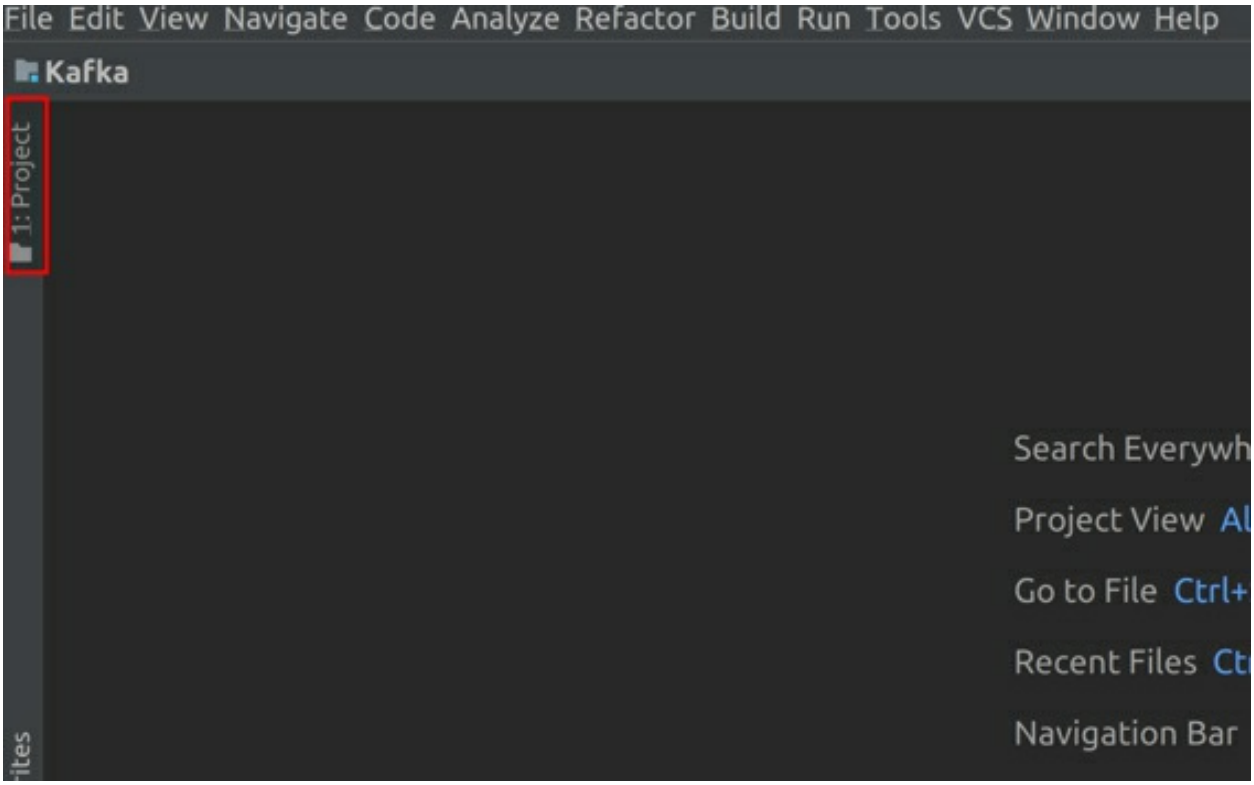
Step 2: You will then be taken to the “New Project” screen. Click on Scala in the left panel, select “SBT” and then click on the “Next” button as shown in the screenshot below.



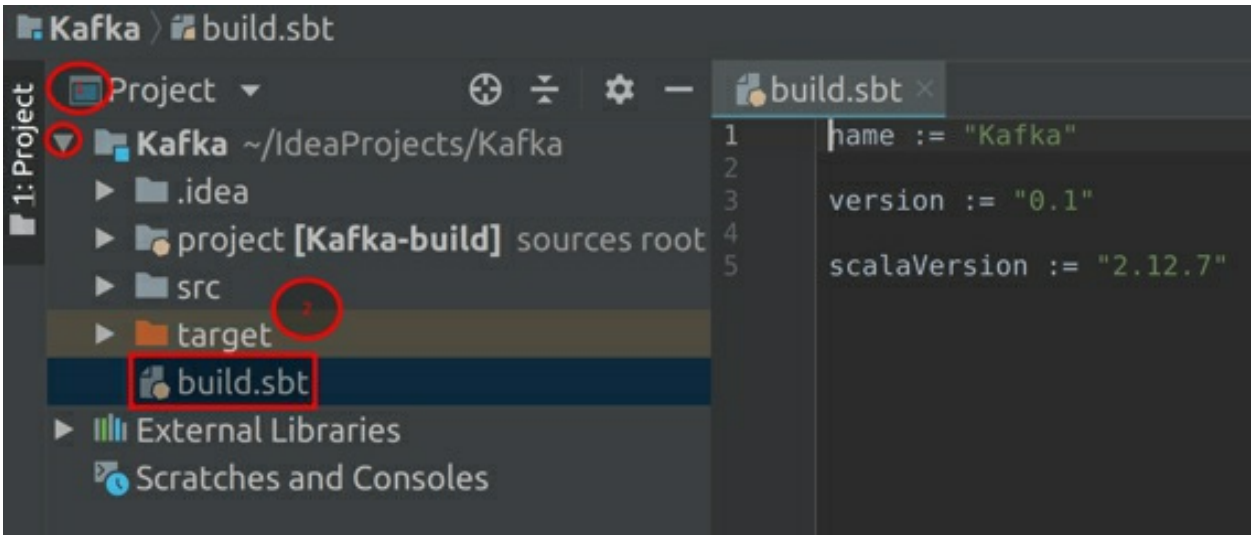
Step 3: After clicking the "Next" button in the previous step, you will be taken to a prompt screen to enter your project's name. Enter the project name as "Kafka". Please make sure that the JDK, SBT, and Scala versions are selected automatically as shown in the screenshot below. Furthermore, check the "Sources" checkbox for both the SBT and Scala, if not *checked* already. Finally, click the *Finish* button.



Step 4: You will now be taken to the IDE interface. Click on the “Project” as shown in the screenshot, if it is not open already.



Expand the *Kafka* project by clicking on the small triangle to the left of Kafka's title, if not already expanded. Thereafter, please double click on the *build.sbt* file as shown in the screenshot.



Step 5: Now go to the Maven Repository for Kafka using the following URL:

`https://bit.ly/3iJGkOw`

Click on the *Kafka Clients* link as shown in the screenshot below.

Next, please select the version of Kafka that you have already installed. For this book, we have installed Kafka 2.5.0 and hence we will be clicking on *2.5.0* link for *Kafka*. Please select the correct version according to your specific installation package.

Now, select SBT tab and copy all the lines of code for SBT and paste it in the *build.sbt* file.

Go back to the Maven Repository page and copy paste the *Kafka* libraries as well in the *build.sbt* file.

Furthermore, please add the following *slf4j* dependency. You may visit the website below to add this dependency. Please ensure that you have added the latest stable version, and not the alpha or beta versions.

```
// https://mvnrepository.com/artifact/org.slf4j/slf4j-simple
libraryDependencies += "org.slf4j" % "slf4j-simple" % "1.7.30" % Test
```

Finally, click on the “Import Changes” button to finish the configuration. The import may take some time depending upon the speed of your Internet connection.

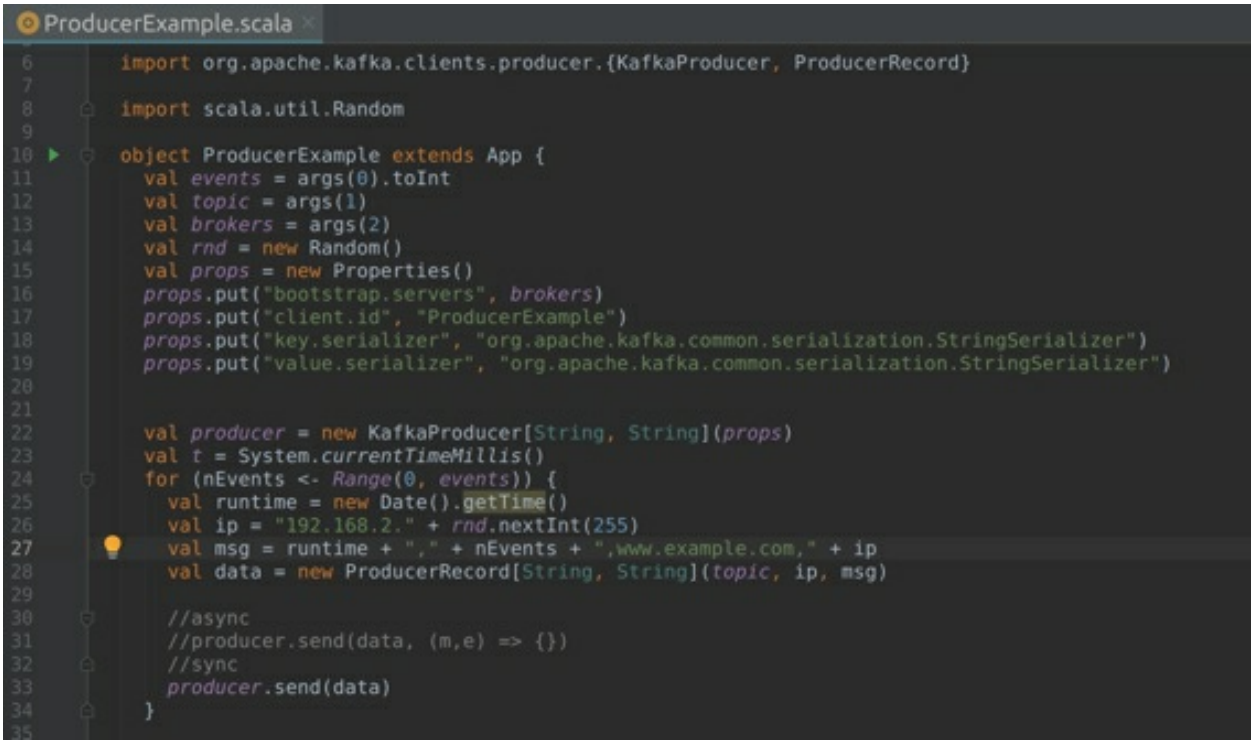
You can add the new libraries (if required) by following the same procedure. You are now ready to write your first Client program in Kafka!

Task 3 is complete!

TASK 4: SPECIFYING BOOTSTRAP SERVERS

The Bootstrap servers are specified as a configuration property within the Producer or Consumer code. Let us look at the Bootstrap server in the following Producer example:

Step 1: The following screenshot provides a Producer code.



Step 2: The Bootstrap servers are set by a configuration property called *bootstrap.servers* as shown in the screenshot below.

The host:port pairs of Brokers are then mentioned in terms of the comma-separated values.

For example, the *bootstrap.servers* will look like

`props.put(“bootstrap.servers”, “localhost:9092, localhost:9093”)`

Please note that the example Producer code above takes the list of Brokers as arguments, and hence, the variable name is specified in the value field of

the property instead of the list of Broker addresses.

We shall be working with the Producers in the next chapter and execute the Producer code.

Task 4 is complete!

SUMMARY

In this chapter, we discussed the Replications, Partitions, and the Bootstrap server configuration property. We have learned that how the replication process may help in the high availability of data in the Kafka cluster, in the event of Broker failure due to the hardware or network issues.

Subsequently, in the labs, we learned to download and install the *Scala* and IntelliJ. Furthermore, we have also configured IntelliJ to execute the Kafka client code. Finally, we discovered how to specify the Bootstrap servers in the Producer example.

CHAPTER 5:
THE PRODUCER

THEORY

We have so far elaborated the basics as well as the in-depth architecture of Kafka. Let us now dive deeper and explain the most important component of Kafka called the Producer, which has the ability to generate messages and allow them to write into Kafka using the Kafka Producer APIs.

What are the Producer APIs? According to the Kafka documentation, "*The Producer API allows applications to send streams of data to Topics in the Kafka cluster.*" This simply means that the Producer APIs allow the users to implement a Producer application that can send data to one or more Topics present in the cluster of Kafka Brokers. For example, consider a few web servers generating HTTP logs for all the user visits to those servers. A security team might want to analyze these logs generated by the web servers for the security issues in real time. In this scenario, they could implement a Producer application using the Producer APIs to get the log data from the servers into Kafka and then consume the data using the Consumer APIs and perform the required analysis.

In this chapter, let us first look at all the internal components and workflow of a Producer. Subsequently, we will learn how to implement a Producer application using the Producer APIs in Scala. Furthermore, we shall implement and run a Producer to send messages to the Brokers of a Kafka cluster in the labs.

PRODUCER WORKFLOW

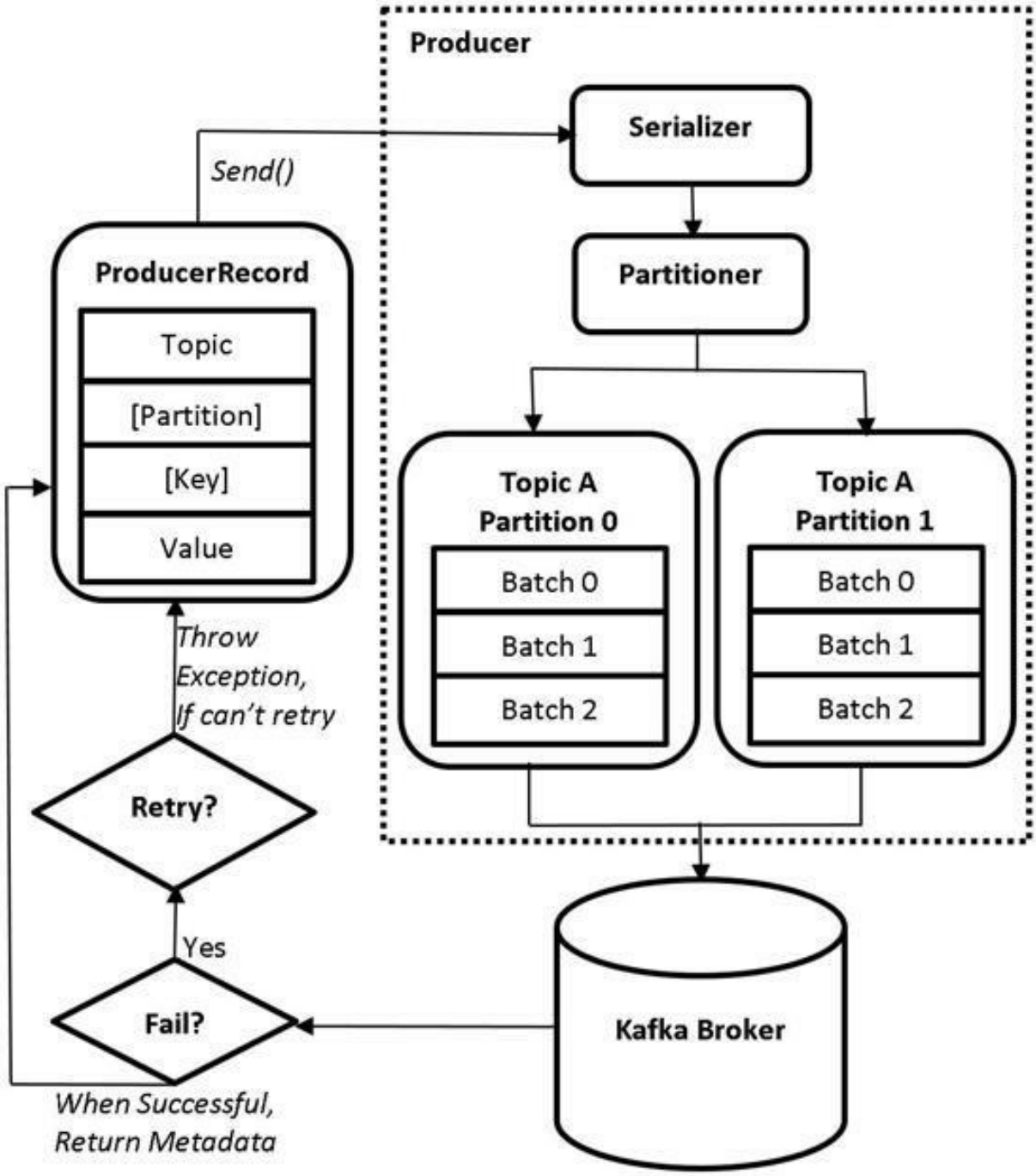
The basic aim of a Producer is to be able to send messages to the Topics in a Kafka cluster. This aim may seem simple at first, but a lot of process takes place behind the scenes of a Producer.

The following are the internal components of a Producer:

- ProducerRecord
- Serializer
- Partitioner
- Producer Partition buffers

Let us look at these components in detail and understand the workflow of a Kafka Producer.

Figure 5(a) depicts the workflow of sending messages from a Producer to the Brokers in Kafka cluster.



5(a) Producer Workflow

ProducerRecord

The first step in the workflow is to create a *ProducerRecord* object, which contains the name of the Topic to which the messages (or events) may be sent by the Producer. The *ProducerRecord* object also contains another field

called ‘value’. Essentially, a *value* is the actual message that has to be sent by the Producer to Brokers. Moreover, we can optionally specify the key and Partition in the *ProducerRecord* object. Since they are optional, Figure 5(a) above shows them enclosed inside ‘[]’. A typical example of the *ProducerRecord* object is shown below.

```
val record = new ProducerRecord[String, String](topic, value)
```

This line of code simply creates a *ProducerRecord* object which takes two parameters of type String *i.e.*, the Topic name and value (actual message). This *ProducerRecord* object has to be programmed by the developer, while the rest of the process is internally managed by the inherent Producer mechanism in Kafka. This is why the internals of Producer are placed inside a dotted box in Figure 5(a). Subsequently, the *ProducerRecord* is processed to the next step (*Serializer*) using the *send()* method.

Serializer

The *ProducerRecord* is now sent to a *Serializer*. *Serialization* is the process of transforming the structured objects to a Byte stream in order to send them over the network or save it to a persistent storage. The reverse process, *i.e.*, the transformation of Byte stream to the structured objects is termed as the *Deserialization*. But why do we have to serialize? This is because Kafka only deals with the data that is formally arranged in Bytes. But this does not imply that the users should only send data in the form of Bytes. The data can be sent in many forms with various types such as, *String*, *Int*, *JSON*, class objects and many more. Any type of data will be serialized by the *serializer* and transmitted over the network for storing in the Brokers. The consumers can then deserialize these messages and receive them in the type they were originally produced and sent by the Producer. The programmer has to specify the type of serializer in the Producer application to tailor the type of messages being sent. For example, if the original message is of the type *String*, the programmer must specify the String Serializer.

Partitioner

After the serialization process, the data is then sent to the *Partitioner*. Let us understand this with the following example. Let us consider that we have a

Topic called *transactions* with three different partitions, *i.e.*, *transactions-0, transactions-1* and *transactions-2* stored in three Brokers. Now the Producer starts transmitting the messages to this Topic. How does the Producer determine which message should specifically be sent to which Partition? This process is performed with the help of Partitioner. A Partitioner determines the Partition to which a record/message should go. There are 4 types of Partitioners that determine how the messages are delivered to the Partitions. These include:

- Round-Robbin Partitioner
- Hash Partitioner
- Specify Partition number explicitly
- Custom Partitioner

  - The Round-Robbin Partitioner is the default Partitioner in Kafka. When no key or partition number is specified in the *ProducerRecord*, the messages from the Producer are automatically sent to the partitions in a Round-Robbin fashion. In case of our example, the first message, say M1 will be sent to *transactions-0*, second message, M2, will go to *transactions-1* and the third message, M3, will go to *transactions-2*. Similarly, the messages M4, M5, and M6 will be transmitted to *transactions-0*, *transactions-1* and *transactions-2* respectively, and so on. In this manner, the messages are continuously assigned to the partitions in the Round-Robbin manner.

  - The Hash Partitioner is used whenever a key is specified in the *ProducerRecord*. The hash Partitioner computes the hash of the key and performs the modulo of hash result with the number of partitions. The computed result is the number of partitions, to which the message for that key should go to. This computation is given below:

```
hash(key) % Number of partitions
```

In case of our example, there are a total of three partitions. Hence,

when the hash of key is performed, the modulo operation with this number of partitions will always be 1, 2, or 3, based on the key.

- We can also specify the number of partitions explicitly in the *ProducerRecord*. This will enable all the messages to explicitly go to the specified partition number.

- Finally, we can also create a custom Partitioner. We can create our own Partitioner and specify how the messages should be delivered to the specified number of partitions. Kafka uses the custom Partitioner when specified within the *partitioner.class* property in the Producer application.

Producer Partition Buffers

The Producer sends the messages to partitions in batches. Depending upon the Partitioner used, the Producer maintains a memory buffer for each partition. These messages are flushed to the partitions upon reaching a certain threshold. Let us understand this with the following example:

- In Figure 5(a) depicting the Producer Workflow, there are two partitions for *Topic A*. For these two Partitions (Partition 0 and Partition 1), Kafka creates two memory buffers (RAM allocated) within the Producer. The size of each buffer is allocated using the property called *buffer.memory*.

- Let us consider that the Round-Robbin Partitioner (the default one) is used to determine the partitions for messages to be delivered to.

- Once the Producer has started producing the messages, it does not immediately send these messages to the partitions. Instead, the Producer stores these messages in the buffer. The first message, say M1, is buffered in the allocated buffer for Partition 0. Similarly, the second message, M2, is buffered in buffer allocated for Partition 1.

- In the same way, the messages M3, M5, and M7 will be buffered in Partition 0 and the messages M4, M6, and M8 will be buffered in Partition 1.

- At this point of time, the buffer for Partition 0 contains messages M1, M3, M5, and M7, whereas the buffer for Partition 1 contains messages M2, M4, M6, and M8. The messages within the Partition 0 buffer are called Batch 0, whereas the messages in Partition 1 buffer are also called Batch 0, as presented in Figure 5(a).

- The Producer always sends the messages to the Partitions in the form of batches. Once the preset threshold for a batch is crossed, the batches are flushed to the appropriate Brokers containing those Partitions. In our example, Batch 0 will be transmitted to the partitions once the threshold is reached. A new batch, let's say Batch 1, is allocated for new messages and the same cycle is repeated.

- The preset threshold at which the batches have to be flushed are controlled by using the following configuration properties:

```
batch.size
linger.ms
```

- The *batch.size* property is used to specify the size of batch in bytes, while the *linger.ms* property specifies the minimum amount of time (in milliseconds) to wait for the additional messages before sending the batch to the specified partitions. The messages are flushed in case if either the size of batch crosses the size preset in *batch.size* property*,* or if the time set in the *linger*.ms property has reached.

- A batch may contain any number of messages based on the properties discussed above. We shall learn how to determine the size for *batch.size* and time for *linger.ms* properties in the proceeding chapters.

Once the Broker starts receiving messages, it sends a response back to the Producer. If the messages were successfully received by the Broker, the Broker will send metadata information *i.e.*, the *RecordMetadata* object. This object contains Topic, partition, and the offset of the record within the partition. However, if the Broker fails to receive the message, it responds with an error. The Producer then retries to send the messages until the Broker receives them. If the Broker continues to respond with error, the Producer eventually quits and returns an error itself.

TYPES OF PRODUCERS

There are three different types of approaches which are used by the Producers to send data to the Brokers. The major difference between these approaches is based on the way in which the data is being sent to the Brokers and how the errors are being handled. Let us look at these approaches in detail.

Fire and Forget	The Fire and Forget Producer simply sends a message to a Broker, and does not worry if the message was successfully received by that Broker. Even though Kafka is fault tolerant (*i.e.,* Producer automatically resends messages in case of a failure), there may be some scenarios where the messages sent by the Producer are not received by the Brokers. Therefore, there is a chance of messages being lost with this type of Producer. It is recommended that this type of Producer should be utilized when we are dealing with the huge volumes of data, and losing a few messages is not a problem. One example of such a Producer is ingesting tweets from Twitter for the sentiment analysis. It is evident that losing a small percentage of tweets would not really be a big problem in this case.

Basically, using this type of approach may result in some data loss. Therefore, it is not recommended to use the Fire and Forget approach, if it is mandatory to receive all the messages transmitted by the Producer. The Fire and Forget Producer has a very low latency, but may not be fault-tolerant to the data loss.

Synchronous Producer

The Synchronous Producer sends a message to a Broker and waits until it receives a response from the Broker. The Broker responds with the *RecordMetadata* object, if it had successfully received the message. In case of a failure, an exception is thrown. There is no data loss when this type of Producer is utilized. The Producer waits for a response before sending out another message. This type of Producer should be used when it is mandatory to receive every generated message by the Producer. One example of this sort of Producer is the credit card transaction processing, because it is mandatory to receive all the messages regarding every transaction that has been processed and we cannot afford to lose even one of them.

Synchronous Producer has a very high latency but is fault tolerant to the data loss. It guarantees that every message that is sent to the Producer is either received by the Broker or otherwise, an exception is thrown in case of a failure. Consequently, the developer has to implement a solution on how to deal with the exception.

Asynchronous Producer	While Fire & Forget and Synchronous Producers are two extreme approaches of sending data, the Asynchronous Producer approach is somewhere in middle of the above two approaches. The Asynchronous Producer sends a message to a Broker and registers a callback method to handle the responses from the Brokers. The Producer keeps on sending the messages and does not wait for the response. Instead, it registers a callback method that gets triggered when there is a response from the Brokers. The callback method returns either the *RecordMetadata* object when the message is successfully received or an exception in case of a failure. We do nothing if the message was successfully received; however, we need to implement a solution in case of failure. The difference here when compared to synchronous producer is, in this approach there is no waiting after sending every message for response from the Broker. The callback method fetches the response while still being able to send the messages. There is low latency when compared to synchronous producer but there may be data loss.

Let us understand these concepts better with an example. Consider a fast food joint that sells burgers. A customer (Broker) places an order for a burger (Messages). This message (burger request) can be handed over to the customer by chef (Producer) in three different ways.

- The chef makes the burger and places it over the counter. If the chef doesn't care whether the customer received the burger or not, this situation is similar to the Fire and Forget approach. The customer might or might not receive the burger. If the customer was not available at the counter for some reason while the burger was placed over the counter, he might have never received it.

- Let us look at the second approach. The customer (after placing the order for a burger) is handed over a receipt. The customer always stays close to the counter and constantly inquires if the burger is ready and available on the counter. Once the chef has prepared the burger, the customer is asked for the receipt. The customer hands over the receipt (refer to the return *RecordMetadata* object) and takes his burger. In this manner, the customer always receives his burger, whereas the chef knows that the customer had received his burger. Subsequently, the chef takes the next order. However, if the customer is not available (refer to the error exception) at the counter after the burger is ready, the chef might make an announcement (refer to exception handling) with the order number for a few times before canceling the order. In this type of approach, the chef does not take new orders until the current order is properly handed over to the customer, or in case of his absence, the situation is handled accordingly. This kind of approach is similar to the Synchronous Producer.

- Let us now look at the final approach, *i.e.*, Asynchronous Producer. The customer places an order online. The chef receives the order, prepares the burger and places it in the pickup tray. The chef does not stop here, and continues to take customer orders and prepares stuff as ordered and places them in the pickup tray. The delivery boy (callback method) then picks up the orders and delivers them to the respective customers. Then the delivery boy informs the chef that all the orders have been successfully delivered (Return *RecordMetadata* object) to customers. However, if the customer is unavailable (error exception) to receive the order, the delivery boy also relays this information back to chef. In this case, the chef may ask the delivery boy to reattempt the delivery at a later time or cancel the order. In this approach, the chef does not stop working on the upcoming orders and also ensures that the customers received their orders. However, the customers might not receive their orders if they are not available at the time of delivery, and the chef has to handle the situation accordingly. This kind of approach is similar to the Asynchronous Producer.

PRODUCER CONFIGURATIONS

Let us look at a few of the Producer configurations, so that we can change them according to our requirement. There are a lot of Producer Configurations available. These configurations can be found in the Kafka documentation page, using the link available in the references section. Most of the configurations are preset with the default values that need not be modified. We shall now look at these configurations which can have a major impact on the performance when modified.

acks: The acks property is a Producer configuration that defines the acknowledgments sent to the Producers by the Brokers. This property has the acknowledgment values of *0*, *1,* and *all*. It defines the number of replicas that can receive the messages, before the Producer can consider that the message was successfully received by all the Brokers.

***acks = 0*:** If the acks property is set to 0, the Producer will not wait to receive

the acknowledgment from the Brokers. Therefore, the Producer just keeps sending data to the Brokers without caring about the brokers' acknowledgments. The Brokers may or may not receive any data. There could be data loss, if the Broker is unavailable when the Producer sends data. However, there is no latency, as the Producer does not wait for the acknowledgments and keeps sending data. It can be noted that the configuration leads to an approach which is equivalent to the Fire and Forget approach.

***acks = 1*:** When the acks property is set to 1, the Producer waits for the acknowledgment from the *Leader* Broker hosting the Partition. The Producer sends the data to the *Leader* Broker and waits for its acknowledgment. The *Leader* Broker after receiving the data sends an acknowledgment to the Producer, which then starts sending other messages. However, the Producer does not care if the *Follower* Brokers containing the replicas have replicated the data from *Leader* Broker.

If the *Leader* is down before the *Followers* were able to replicate the data from the *Leader*, there could be data loss. It can be noticed that there could be a limited data loss in this case within the domain of a single *Leader* as well as a little latency as the Producer has to wait for the acknowledgment from the *Leader*. This setting guarantees that the *Leader* received the data but there is no guarantee that the replicas within the domain of the *Leader* have also received the data.

***acks = all*:** When the acks property is set to '*all*', the Producer waits for the acknowledgments from both the *Leader* and *Follower* Brokers hosting the Partition replicas. The Producer sends the data to *Leader* Broker and waits for its acknowledgment. The *Leader* Broker then commits the data to its log file. The replicas in *Follower* Brokers then request the data from the *Leader* and replicate the data. Once the *Follower* Brokers have replicated the data, they immediately send an acknowledgment to their specific *Leader* Broker. The *Leader* then sends the acknowledgment to Producer. The Producer then starts sending other messages.

There is no data loss in this case. However, there is high latency as the Producer waits for the acknowledgment from all the replicas of a Partition. This setting guarantees that all the replicas of a partition have received the

data properly.

batch.size: Please check *Producer Partition Buffers* section.

buffer.memory: The *buffer.memory* property specifies the size of memory in Producer to buffer the messages before sending them to the Brokers. If the records are sent faster than they can be transmitted to the server, then the buffer space will be exhausted. When the buffer space is exhausted, the additional send calls will block. The threshold for time to block is determined by the *max.block.ms* property after which it throws an exception called *TimeoutException*.

compression.type: The *compression.type* property is used to specify the compression algorithm to be used while sending data. The messages are sent uncompressed, by default. We can specify various compression types such as, *snappy, gzip, or lz4* as values. When a compression is specified, the data is compressed before sending it over the network. This enables less network utilization and storage.

linger.ms: Please check *Producer Partition Buffers* section.

max.block.ms: The *max.block.ms* property is used to control the duration for which the producer blocks while calling the methods *KafkaProducer.send()* and *KafkaProducer.partitionsFor()* methods. These methods can be blocked either because the buffer is full or the metadata is unavailable.

max.request.size: The *max.request.size* property is used to specify the maximum request size a Producer can send. This limits the maximum size of a single message, as well as, the total number of messages that can be sent in a batch by the Producer. Similarly, a Broker also has the limit for the maximum request size it can accept using the *message.max.bytes* property. It is recommended to have the same value for both the properties, so that the Broker does not reject a message sent by the Producer, which is more than the configured maximum size a Broker can accept.

These are few Producer configurations required to configure the Producer as per requirement. Please note that the configuration properties mentioned above are only a few out of a long list of properties. Please check the link in

References section for the complete list of all configurations.

That’s all for the theory part of this chapter. Let us move to the lab exercise to check all this theory in action.

AIM

The aim of the following lab exercises is to implement a Kafka Producer. We shall also run the Producer to send data to the Brokers.

The labs related to this chapter include the following exercises:

- Import Kafka Packages & Declare variables
- Create a Kafka Producer and *ProducerRecord* Object
- Running the Producer
- Sending the messages Synchronously
- Sending the messages Asynchronously

For this purpose, we need the following packages to perform these lab exercises:

- Java Development Kit (JDK)
- Apache ZooKeeper
- Apache Kafka
- Scala
- IntelliJ IDEA

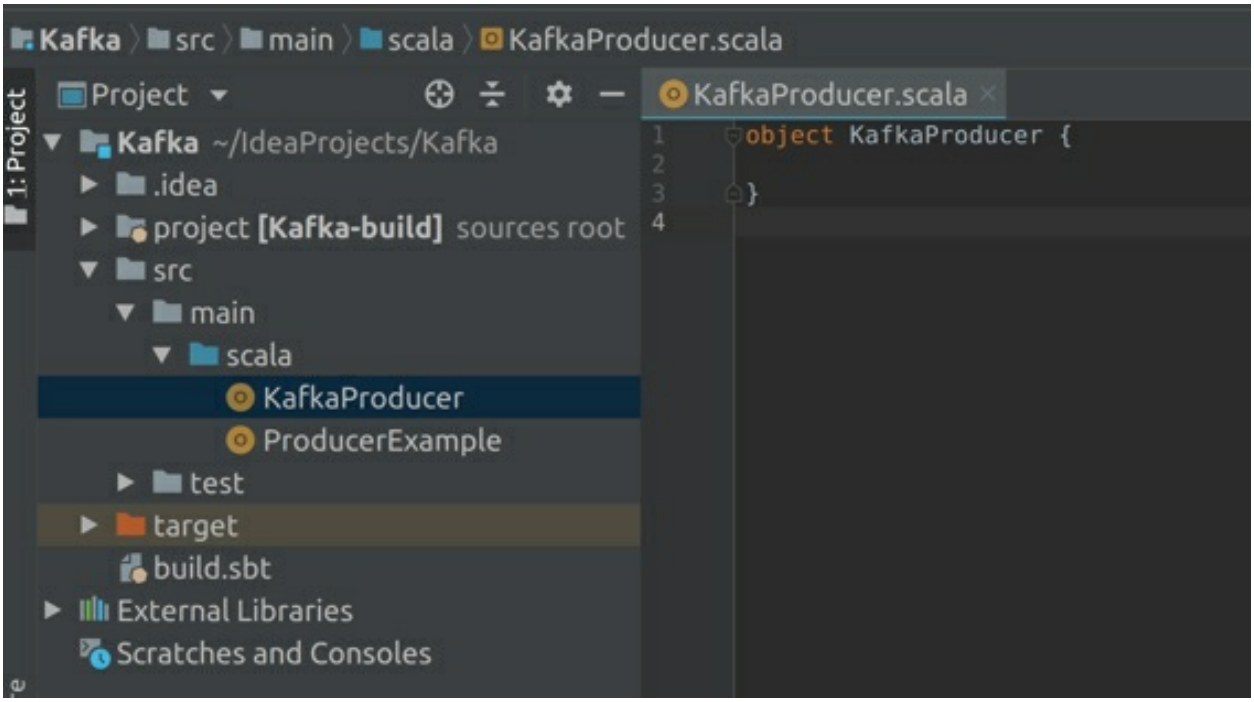
LAB EXERCISE 5: THE PRODUCER

1. **Import the Kafka Packages & Declare Variables**
2. **Create a Kafka Producer and *ProducerRecord* Object**
3. **Running the Producer**
4. **Sending messages Synchronously**
5. **Sending messages Asynchronously**

Let us start by implementing a very basic Kafka Producer. For the simplicity and better understanding, this lab exercise is divided into various tasks for the major steps that are involved in implementing a Kafka Producer.

TASK 1: IMPORT KAFKA PACKAGES AND DECLARE VARIABLES

Step 1: Let us create a new *Scala* Object. Right click on the *Scala* folder in IDE, hover on *New* and click on *Scala Class*. You should see a prompt to enter the name. Enter *KafkaProducer* in the name field. Click on *kind* dropdown and select *Object*. Click *OK* and you should see the *KafkaProducer* object, as depicted in the screenshot below:



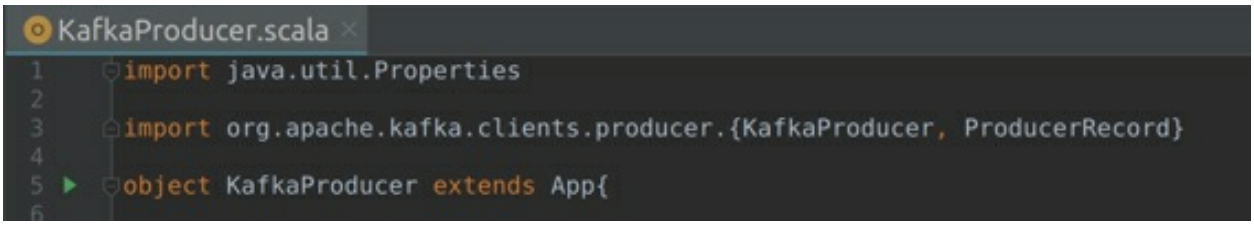
Let's *extend* this object to *App* trait to convert it into an executable program.

```
object KafkaProducer extends App { }
```

Step 2: Now that we have created an object, let us import the packages given below. These imports are required to specify the properties and create *Producer* & *ProducerRecord* objects.

```
import java.util.properties
import org.apache.kafka.client.producer.KafkaProducer
```

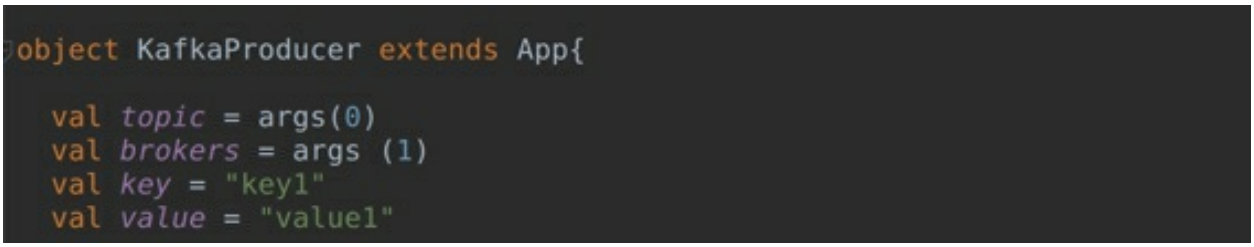
```
import org.apache.kafka.client.producer.ProducerRecord
```



Step 3: Let us now specify a few variables. First, we specify the name of the *Topic* and the bootstrap address (Brokers) as arguments. Next, we specify the message, *i.e.,* the key and its value. This is the message that will be sent to the brokers. These messages are sent in the form of *key* and *value* pairs. However, a message can also be sent without a *key* as well.

```
val topic = args(0)
val brokers = args(1)
val key = “key1”
val value = “value1”
```

Since we are specifying a *key*, the hash Partitioner is used to determine which message goes to which Partition.



Step 4: The next step is to specify the properties. We have already learned the *bootstrap.servers* property in the previous lab exercises.

```
val props = new Properties()
props.put(“bootstrap.servers”, brokers)
props.put(“client.id”, “Kafka Producer”)
props.put(“key.serializer”,
“org.apache.kafka.common.serialization.StringSerializer”)

props.put(“value.serializer”,
“org.apache.kafka.common.serialization.StringSerializer”)
```

- The *put* method is used to specify the configuration properties to the *props* object.
- The *client.id* is an optional property that is used to set a unique identifier to the Producer.
- The *key.serializer* is used to specify the serialization for the *key* while the *value.serializer* is used to specify the serialization for the *value*. Since our message is of type '*String'* in both *key* and *value*, a *StringSerializer* is used for both the *key* and *value*. However, you can also use an *IntSerializer, DoubleSerializer, LongSerializer, JSONSerializer etc,* if the keys or values are of that type.

The various types of *serializers* in Kafka include *ByteArraySerializer, ByteBufferSerializer, BytesSerializer, DoubleSerializer, ExtendedSerializer.Wrapper, FloatSerializer, IntegerSerializer, LongSerializer, SessionWindowedSerializer, ShortSerializer, StringSerializer, TimeWindowedSerializer, UUIDSerializer*

Please check *Serializer* section under *Producer Workflow* for more information on serialization.

```
object KafkaProducer extends App{

  val topic = args(0)
  val brokers = args (1)
  val key = "key1"
  val value = "value1"

  val props = new Properties()
    props.put("bootstrap.servers", brokers)
    props.put("client.id", "Kafka Producer")
    props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer")
    props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer")
```

Task 1 is complete!

TASK 2: CREATE A KAFKA PRODUCER PRODUCERRECORD OBJECT

Step 1: Now that we have configured the Producer, the next step is to create a Producer object. The Kafka Producer object is instantiated by passing the *props* object as an argument.

```
val producer = new KafkaProducer[String, String] (props)
```

The above line of code instantiates a Producer object that is of type *String*. The *String* type also specifies the type for *key* and *value*.

```
val props = new Properties()
  props.put("bootstrap.servers", brokers)
  props.put("client.id", "Kafka Producer")
  props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer")
  props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer")

val producer = new KafkaProducer[String, String] (props)
```

Step 2: Now that we have a Producer object created, the next step is to enable this Producer to send messages. To be able to send messages, we must instantiate a *ProducerRecord* object.

```
val message = new ProducerRecord[String, String] (topic, key, value)
```

The *ProducerRecord* object constructor is instantiated by passing the arguments as the name of the Topic, *key,* and *value*. The *ProducerRecord* object contains the actual message that is being sent to the Brokers.

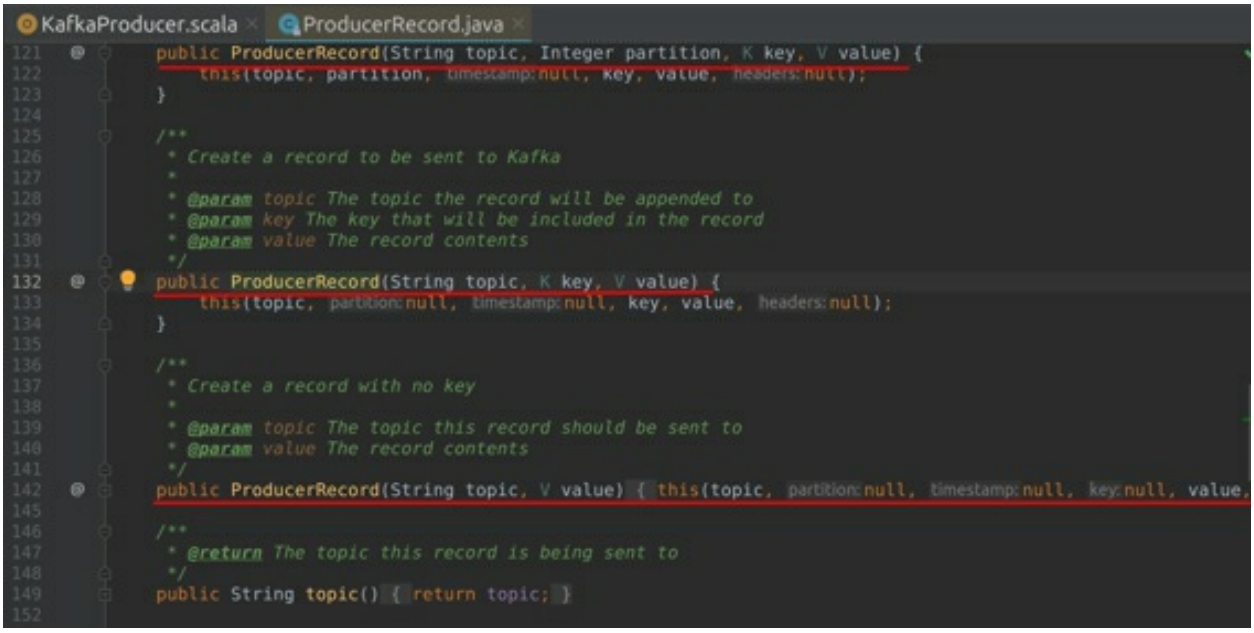
```
val props = new Properties()
  props.put("bootstrap.servers", brokers)
  props.put("client.id", "Kafka Producer")
  props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer")
  props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer")

val producer = new KafkaProducer[String, String] (props)

val message =  new ProducerRecord[String, String] (topic, key, value)
```

The *ProducerRecord* object has various constructors. It is mandatory to pass Topic and value to *ProducerRecord* constructor; you can also specify the key, the partition number and a timestamp to include within your message.

To learn more about the *ProducerRecord* object constructors, you may hold the *ctrl* key and click on the *ProducerRecord* as shown in the screenshot below:

```
val props = new Properties()
  props.put("bootstrap.servers", brokers)
  props.put("client.id", "Kafka Producer")
  props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer")
  props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer")

val producer = new KafkaProducer[String, String] (props)

val message =  new ProducerRecord[String, String] (topic, key, value)
```

You should then find a new tab *ProducerRecord.java* displaying all the constructors as depicted in the screenshot below:



Step 3: Finally, we need to *trigger* the message to be sent. This can be performed by using the *send* method. We simply call the *send* method on our producer object by passing the *ProducerRecord* as an argument. The Producer will then start sending the messages to the Brokers.

```
producer.send(message)
```

Finally, after sending the messages, we must close the Producer object using the close message method as shown below. Closing the Producer object will free up the resources being used by the Producer.

```
producer.close()
```

```
object KafkaProducer extends App{

  val topic = args(0)
  val brokers = args (1)
  val key = "key1"
  val value = "value1"

  val props = new Properties()
    props.put("bootstrap.servers", brokers)
    props.put("client.id", "Kafka Producer")
    props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer")
    props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer")

  val producer = new KafkaProducer[String, String] (props)

  val message =  new ProducerRecord[String, String] (topic, key, value)
  producer.send(message)
  producer.close()

}
```

In this manner, we have completed implementing a very basic Producer. Let us run this and check if the message is successfully sent to the Topic.

Task 2 is complete!

TASK 3: RUNNING THE PRODUCER

Step 1: Now that we have finished implementing the Kafka Producer, let us start ZooKeeper and Kafka server in the terminal.

```
$ zkServer.sh start
```

```
$ kafka-server-start.sh /usr/share/kafka/config/server.properties
```

```
uzair@uzair:~$ zkServer.sh start
/usr/bin/java
ZooKeeper JMX enabled by default
Using config: /usr/share/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
uzair@uzair:~$ kafka-server-start.sh /usr/share/kafka/config/server.properties
[2020-08-08 13:10:59,238] INFO Registered kafka:type=kafka.Log4jController MBean
 (kafka.utils.Log4jControllerRegistration$)
[2020-08-08 13:11:01,010] INFO Setting -D jdk.tls.rejectClientInitiatedRenegotia
tion=true to disable client-initiated TLS renegotiation (org.apache.zookeeper.co
mmon.X509Util)
[2020-08-08 13:11:01,207] INFO Registered signal handlers for TERM, INT, HUP (or
g.apache.kafka.common.utils.LoggingSignalHandler)
[2020-08-08 13:11:01,224] INFO starting (kafka.server.KafkaServer)
[2020-08-08 13:11:01,228] INFO Connecting to zookeeper on localhost:2181 (kafka.
server.KafkaServer)
[2020-08-08 13:11:01,323] INFO [ZooKeeperClient Kafka server] Initializing a new
 session to localhost:2181. (kafka.zookeeper.ZooKeeperClient)
[2020-08-08 13:11:02,138] INFO Client environment:zookeeper.version=3.5.7-f0fdd5
2973d373ffd9c86b81d99842dc2c7f660e, built on 02/10/2020 11:30 GMT (org.apache.zo
okeeper.ZooKeeper)
```

Step 2: Let us now create a new Topic and name it *logs*. The Topic has three partitions and the replication factor of 3. Please open a new terminal to perform this.

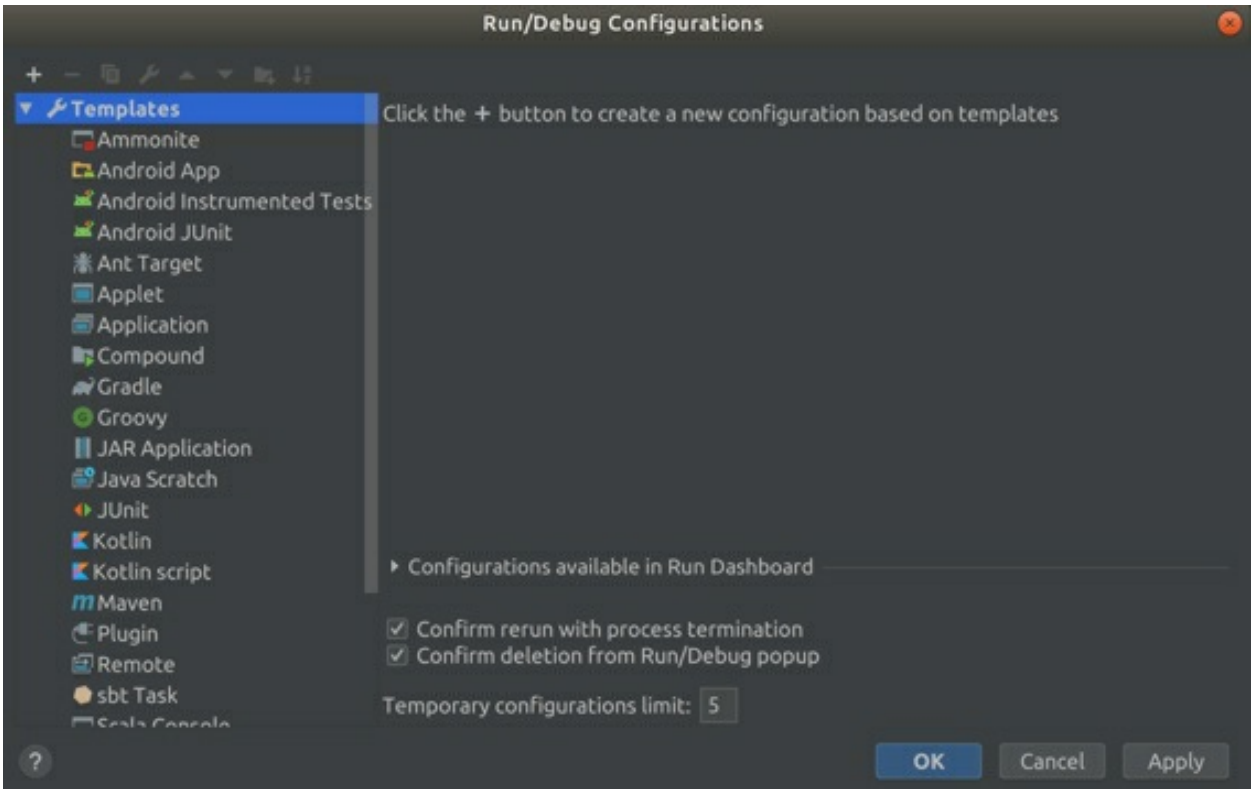
```
$ kafka-topics.sh \
--zookeeper localhost:2181 \
--create --topic logs \
--replication-factor 1 \
--partitions 3
```

```
uzair@uzair:~$ kafka-topics.sh \
> --zookeeper localhost:2181 \
> --create --topic logs \
> --replication-factor 1 \
> --partitions 3
Created topic logs.
```

We may also list all the available Topics in our Kafka server.

Step 3: Now that we have the Topic created, let us finally run the Producer code that we had implemented in the previous steps. Switch back to the IDE, click on *Run* and select the *Edit Configurations…* option. We should see the

configurations window as shown in the screenshot below:

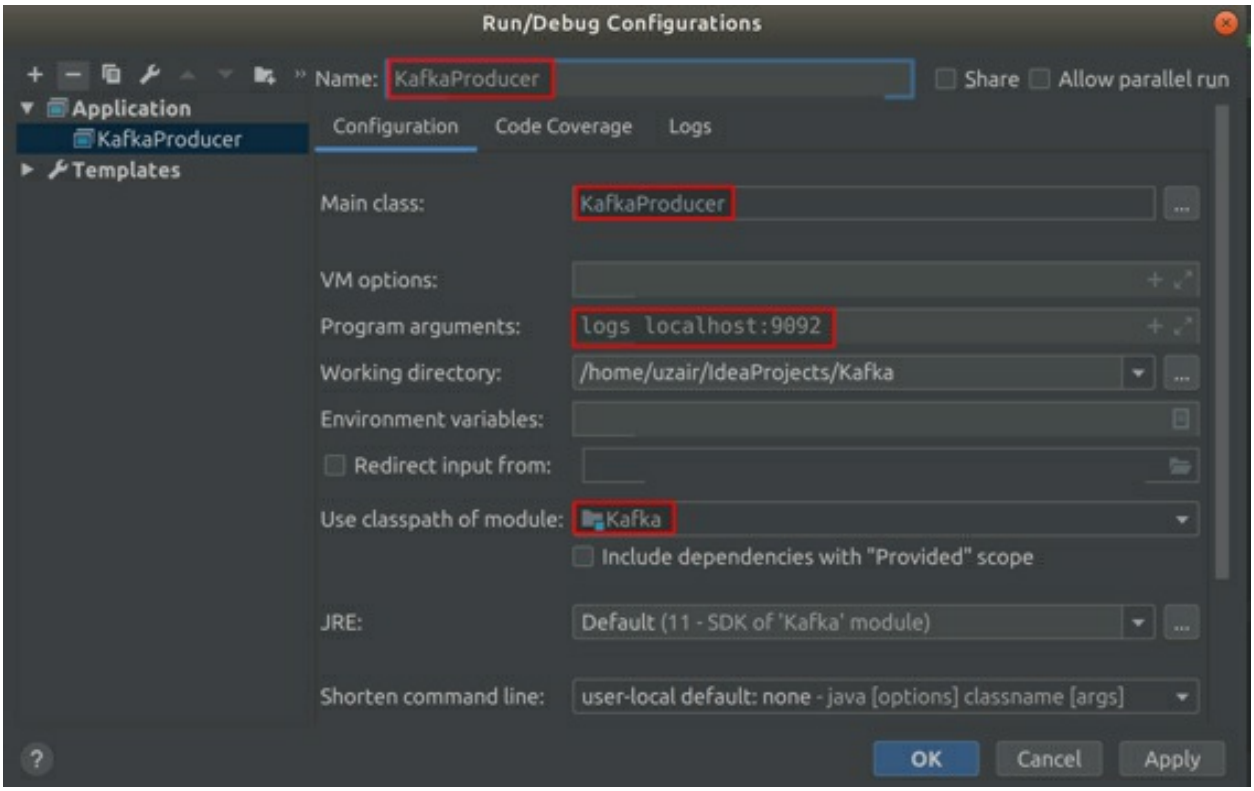


Click on the + icon on the top left of the window and select *Application* from the drop-down list. Enter any name as you like in the name field. We have named it *KafkaProducer*. Click on the … button for the *Main Class* field and select the class.

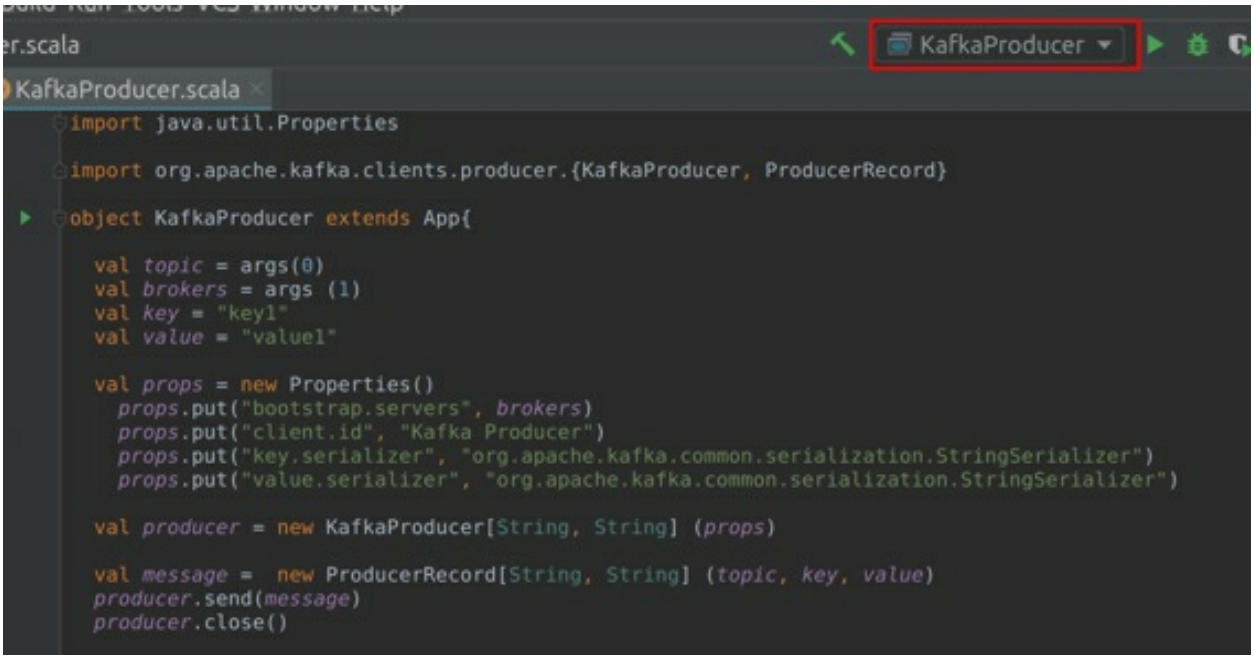
Next, input the program arguments as shown below:

```
logs localhost:9092
```

Finally, if the '*use classpath of module'* is empty, select the value *Kafka* from the drop-down menu. You should see all the values as shown in the screenshot below.

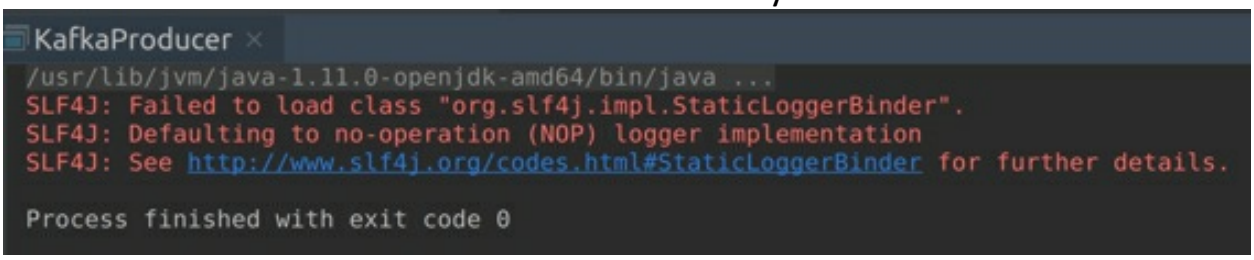


Step 4: Once you have ensured that all the entered values are correct, click on the *OK* button. You should see the configuration on the top right of the IDE as shown in the screenshot below.



Step 5: Let us now run the Kafka Producer. Click on the green play icon to

the right of the configuration drop-down menu. After some time, you should find that the Producer has executed successfully with the exit code 0.



Step 6: Now check the *kafka-logs* directory; and its default location is */tmp/kafka-logs*. You should be able to see three partitions for *logs* Topic. Go through each partition and check the *.log* file for the message that has been received.

This is a very basic implementation of the Kafka Producer. In the real-world applications, Producers keep sending messages for a long duration. We have only implemented this very basic type of Producer for the sake of simplicity and better understanding of the Producer.

Task 3 is complete!

TASK 4: SENDING MESSAGE SYNCHRONOUSLY

In the previous task, we have sent a message and did not really bother if the Broker had received that message or not. In this task, let us learn how to send a message synchronously to the Kafka Broker.

Step 1: Create a new *Scala* object and name it *SyncProducer*. Next, copy the entire code from the *kafkaProducer*. Make sure you change the object name to *SyncProducer.* You should see this code as shown in the screenshot below.

KafkaProducer.scala SyncProducer.scala

```scala
import java.util.Properties

import org.apache.kafka.clients.producer.{KafkaProducer, ProducerRecord}

object SyncProducer extends App{

  val topic = args(0)
  val brokers = args (1)
  val key = "key1"
  val value = "value1"

  val props = new Properties()
  props.put("bootstrap.servers", brokers)
  props.put("client.id", "Kafka Producer")
  props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer")
  props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer")

  val producer = new KafkaProducer[String, String] (props)

  val message =  new ProducerRecord[String, String] (topic, key, value)
  producer.send(message)
  producer.close()
}
```

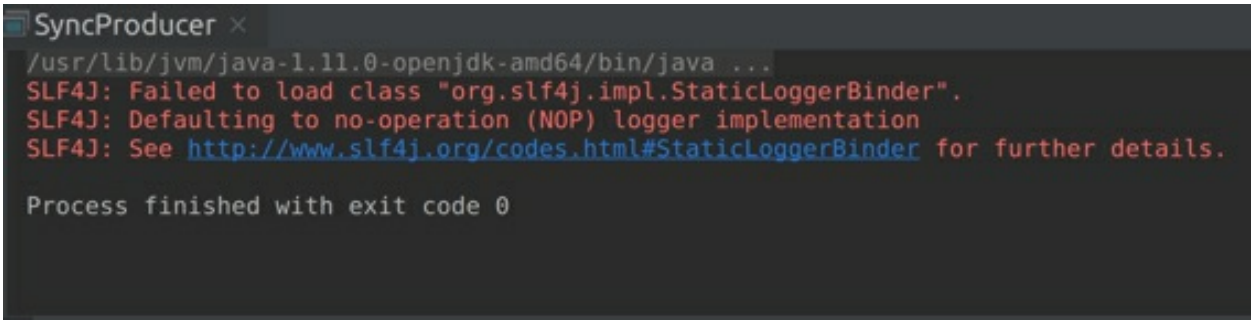
Step 2: In order to send a message synchronously, we use the *Future.get()* method and wait for a response from Kafka server. The *get()* method throws an exception, if the Broker fails to receive the message successfully. However, if the message was successfully delivered, a *RecordMetadata* object will be sent.

The *send()* method we had used in the previous task is enclosed within a try-catch block as shown below:

```
try{
      producer.send(message).get()
} catch {
      case x => {
      x.printStackTrace()
}
}
```

```
val props = new Properties()
props.put("bootstrap.servers", brokers)
props.put("client.id", "Kafka Producer")
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer")
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer")

val producer = new KafkaProducer[String, String](props)

val message = new ProducerRecord[String, String](topic, key, value)

try {
  producer.send(message).get()
} catch {
  case x: Exception => {
    x.printStackTrace()
  }
}
    producer.close()
}
```

Step 3: Before running the Producer, let us create a new Topic and name it *sync-log*. After creating the new Topic, also provide the new configurations as we had done in the previous task, and save these configurations. Make sure you enter the new Topic name in the arguments. Now, let us run the program. You should see that the program has successfully executed with the exit code 0.



You may check the received message in the *kafka-logs* directory under the *sync-logs* topic.

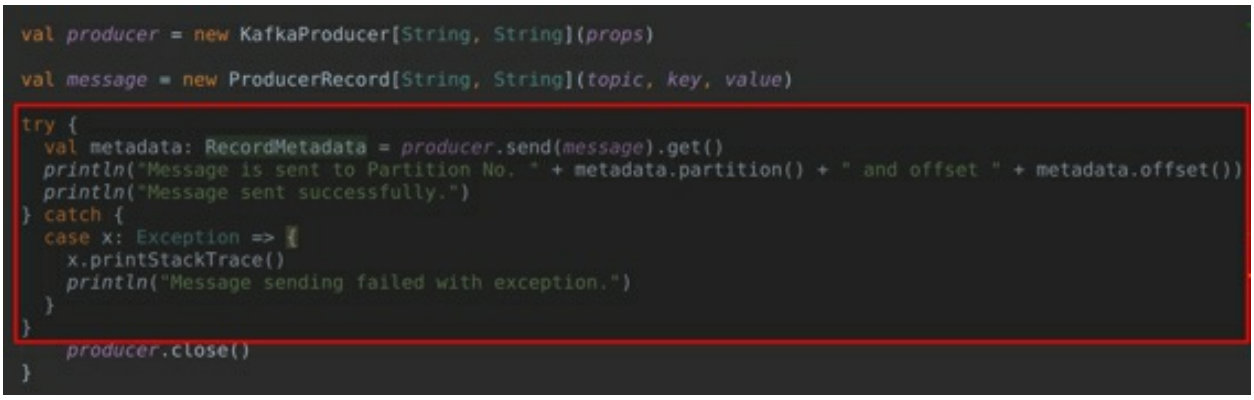
There isn't much difference in this task, compared to what you had performed in the previous task. Let us make a few changes and create a *RecordMetadata* object to learn more about the messages being sent.

Step 4: Let us create a *RecordMetadata* object as shown below. Make sure to add the new imports as shown below:

```
import org.apache.kafka.clients.producer.RecordMetadata
```

```
try{
val metadata: RecordMetadata = (producer.send(message).get())
println(“Message sent to Partition No. ” + metadata.partition() + “ and offset ” + metadata.offset())
println(“Message sent successfully.”)
} catch {
      case x => {
      x.printStackTrace()
println(“Message sending failed with exception.”)
}
}
```

When the Broker receives the message, it responds back with the *RecordMetadata* object. We can then get the partition and offset from the *RecordMetadata* object and display it on the console.



Please note that we have not covered the offset section yet. We will be looking at the offsets in the upcoming chapters.

Step 5: Let us now run the Producer again. You should see the output as shown in the screenshot below:

Step 6: As a lab challenge, stop the Kafka Broker and execute the Producer. You can observe what happens when the message is not delivered successfully.

Task 4 is complete!

TASK 5: SENDING MESSAGE ASYNCHRONOUSLY

Let us finally send the messages asynchronously.

Step 1: Create a new *Scala* object and name it as *AsyncProducer*. Next, copy the entire code from the *kafkaProducer*. Make sure you change the object name to *AsyncProducer.* You should see the same code as shown in the screenshot below.

Step 2: To send a message synchronously, we use *callbacks*. Instead of waiting for a response from the Broker the Producer continues to send messages but instead registers a *callback* method to handle responses from the Brokers.

A *callback* class is passed within the *send* method along with the *ProducerRecord.* Let us now look at the Asynchronous implementation.

Make sure to add the new imports as shown below:

```
import org.apache.kafka.clients.producer.RecordMetadata
import org.apache.kafka.clients.producer.Callback

producer.send(message, new ProducerCallback)
```

We must now implement the callback class (*ProducerCallback*) which extends an interface called *callback*. This class contains a single function called *onCompletion()*, which should be overridden in order to specify the error handling technique.

The *onCompletion()* function will take *RecordMetadata* and *exception* as arguments. If the record was successfully received, the Broker will send a *RecordMetadata* object and the *exception* will be null. However, if it was not successful, the Broker will send an *exception*. Therefore, we should check if the *exception* is not null. If the *exception* is not null, we have handled the error appropriately. Here, we are simply printing the error stack trace to the console. If the *exception* is null, we have to do nothing, as the message was sent successfully.

From this exercise, it is evident that the Asynchronous approach eliminates the requirement to wait for a response to send a new message.

```
class ProducerCallback extends callback {

override        def        onCompletion(recordMetadata:
RecordMetadata, e: Exception): Unit = {

if(e != null) {

e.printStackTrace()

println("Sending Messages Asynchronously failed.")

} else

println("Messages send Asynchronously.")

     }
}
```

```
val producer = new KafkaProducer[String, String](props)

val message = new ProducerRecord[String, String](topic, key, value)
producer.send(message, new ProducerCallback())
producer.close()

lass ProducerCallback extends Callback {

override def onCompletion(recordMetadata: RecordMetadata, e: Exception): Unit = {

  if (e != null) {
    e.printStackTrace()
    println("Sending messages Asynchronously failed")
  }
  else println("Messages sent Asynchronously.")

}
```

Step 3: Before running this Producer, let us create a new Topic and name it a*sync-log*. After creating the new Topic, create the new configurations as we had done in the previous task, and save them. Make sure that you have entered the new Topic name in *arguments*. Now, run the program. You should see that the program has successfully executed with the exit code 0.

```
AsyncProducer
/usr/lib/jvm/java-1.11.0-openjdk-amd64/bin/java ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Messages sent Asynchronously.

Process finished with exit code 0
```

Step 4: As a lab challenge, stop the *Kafka Broker* and run the Producer. You can now observe what happens when the message is not delivered successfully.

Task 5 is complete!

SUMMARY

In this chapter, we have learned that a Producer is an integral Kafka component that generates messages and allows writing them into Kafka using the Kafka Producer APIs.

What are the Producer APIs? According to the Kafka documentation, "*The Producer API allows applications to send streams of data to Topics in the Kafka cluster.*" This simply means that the Producer APIs allow the users to implement a Producer application that sends data to one or more *Topics* present in the cluster of Kafka Brokers.

In the labs, we learned to implement various Kafka Producer approaches.

CHAPTER 6:
THE CONSUMER

THEORY

The previous chapter was all about producing data for a Kafka cluster. In this chapter, let us learn how to *consume* data from the Kafka cluster. The *Consumer* is an important Kafka component which allows consuming or reading data from the Kafka cluster using the Consumer APIs.

Let us first briefly review the concepts we have learned. Kafka receives messages and stores them in their particular *Topics*. These *Topics* are further divided into one or more Partitions to achieve scalability. Producers generate the messages and send them to the Kafka Brokers. The *Leader* Brokers append the received messages to Partitions. The *Follower* Brokers replicate themselves with the messages (data) received by the *Leader* Broker. Once the messages are available in Kafka, Consumers can start reading or consuming these messages. The messages read by Consumers are tracked with the help of *offsets*.

We had mentioned the *offsets* in the previous chapters. Let us now look at the *offsets* in detail.

OFFSET

What is an *Offset*?

Offset is the integer metadata associated with each message, and it increases monotonically for every message. Each produced message contains a *key* and a *value*. Once the Producer transmits the message to the Broker(s), the Broker automatically assigns an *offset*, *i.e.*, an integer starting from zero.

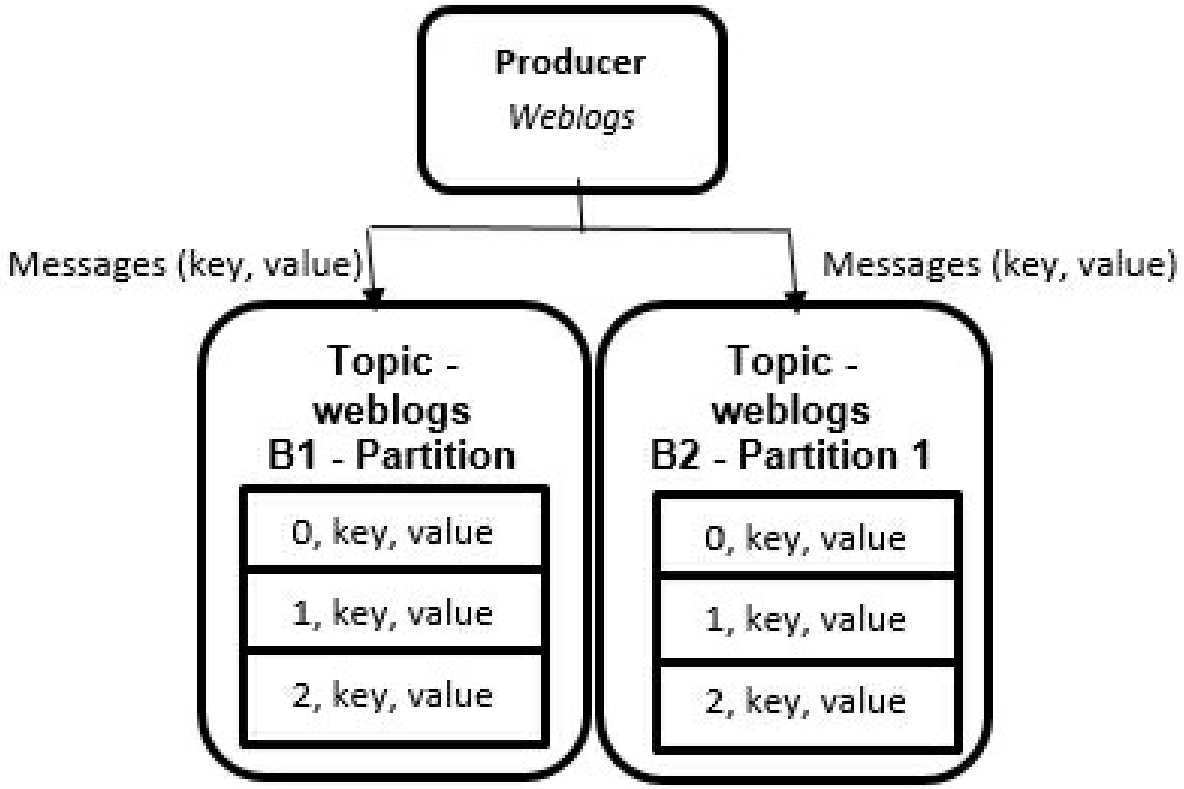
Let us understand this concept with Figure 6(a), as shown below:

- Consider a Kafka cluster with the web server as a Producer. The Producer generates web-logs. The logs are being sent to a Topic with two partitions.

- The Producer transmits a message in the form of *key* and *value*. Let us denote this with *Messages(key, value).*

- The Broker B1, which hosts the Partition *weblogs-0,* receives the message and then assigns an integer *offset* starting from *zero*. The *offset* increases monotonically for each message. For instance, the first message will be denoted as m1(0, key, value), the second message will be m2(1, key, value), the third will be m3(2, key, value) and so on.

- Similarly, the Broker B2 which hosts the Partition *weblogs-1* also receives the message and assigns an integer *offset* starting from zero. This offset also increases monotonically for each message.



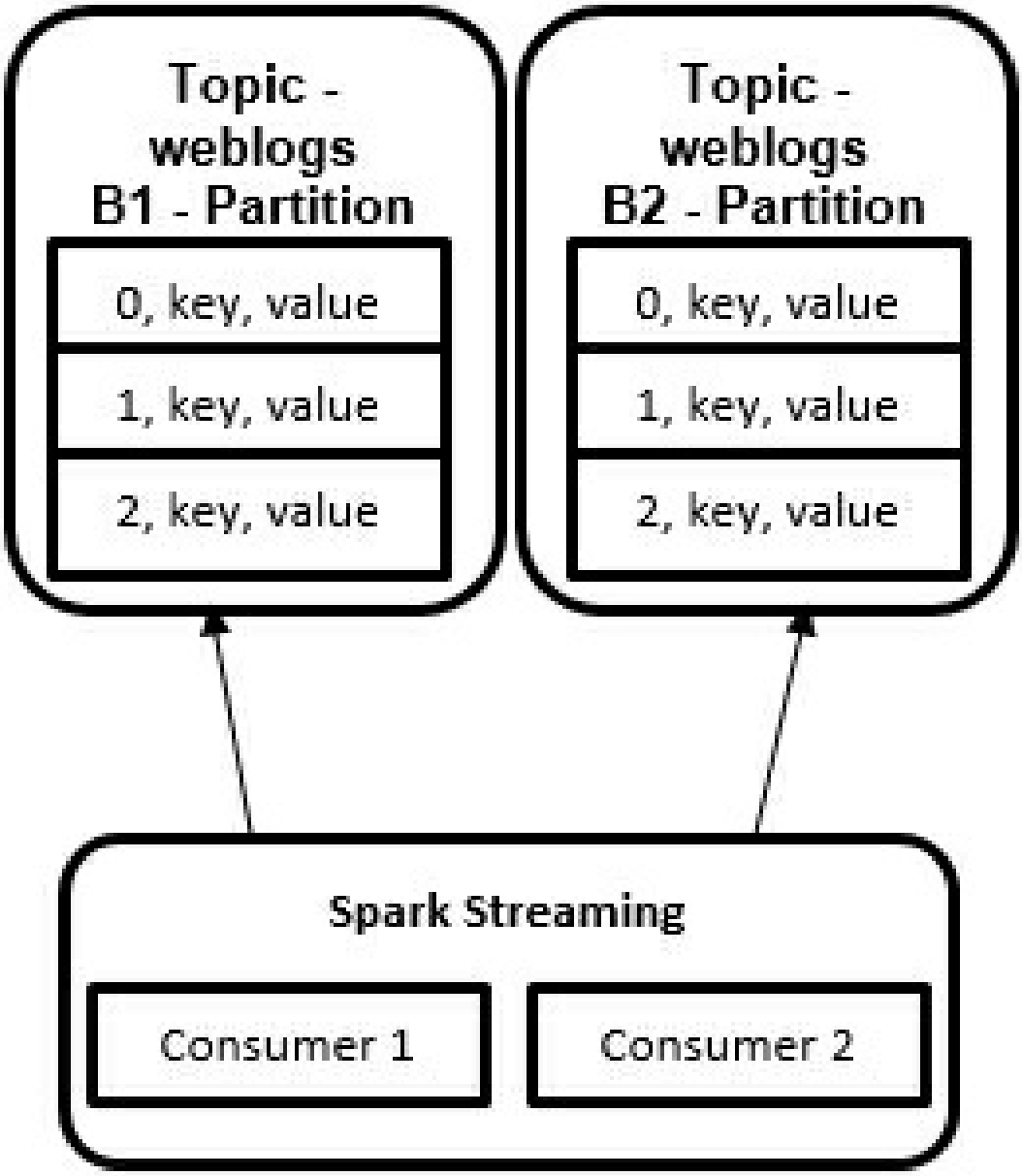
6(a) Offset

What is the purpose of *Offsets*?

Offsets are used by the Consumers to fetch messages from the Kafka cluster. The Consumers periodically request the Brokers hosting the Leader replica of the Partition with the offset number. Subsequently, the Broker transmits the messages from that *offset* number to the Consumer. The Broker will start sending messages to the Consumer, when the minimum available size of new messages in the Broker is equal to the size set in the property called *fetch.min.bytes*. Let us understand this better with the following example:

Let us go back to the example in Figure. 6(a), but this time let us also add two Consumers C1 and C2. Let us consider that the Consumer in our example is a *Spark Streaming application*.

- The Consumer C1 sends a fetch request to the Broker B1 which contains the *weblogs-0* Partition. The request that is sent to the Broker contains the Topic name, partition number and the *offset* number. In our example, the first fetch request will be (weblogs, 0, 0), *i.e.,* the Topic name, partition number, and the *offset* number, respectively.



6(b) Using Offsets to Consume Messages

- The Consumer only requests the messages to fetch from an *offset* number. The Consumer does not specify to which offset the messages should be sent. So, how many messages the Broker sends to the Consumer? This depends on the property called *fetch.min.bytes*. This property specifies the minimum size of messages the Broker should accumulate before it sends them over to the Consumer.

Let's assume that the *fetch.min.bytes* property is set to 10 Kb. This implies that 10 Kb is the minimum size of the messages that must be accumulated by the Broker before it can send it over to the Consumer. The Broker may send more than 10 Kb of messages according to the fetch request.

- Coming back to our example, when the Consumer sends the following fetch request: (weblogs, 0, 0), the Broker might have accumulated 50 Kb of messages. Let's assume that the size of 50 Kb messages constitutes 5 messages. Please note that the message size may vary.

- Now that the Consumer has received 5 messages in the first fetch request. The next fetch request from the Consumer will be (weblogs, 0, 5), since the messages with *offsets* 0, 1, 2, 3, and 4 are already consumed (or read) by the Consumer. The Broker then starts sending messages from the 5^{th} *offset*. This request might have fetched 10 messages of size 100 Kb. This means that the Consumer has now consumed messages from the *offset* 5 to *offset* 14.

- In this manner, the next fetch request from the Consumer will be (weblogs, 0, 15) and so on. The Consumer 2 also fetches the data by following the same process as described for the Consumer 1.

- The replications in *Follower* Brokers also follow the same process. The *Follower* Brokers send fetch requests to *Leader* Broker to replicate themselves with the *Leader* messages. The *Follower* Brokers act as internal Consumers within Kafka.

It is clear from this example that the primary use of *Offsets* is that they are utilized to fetch the next message from the Kafka Brokers.

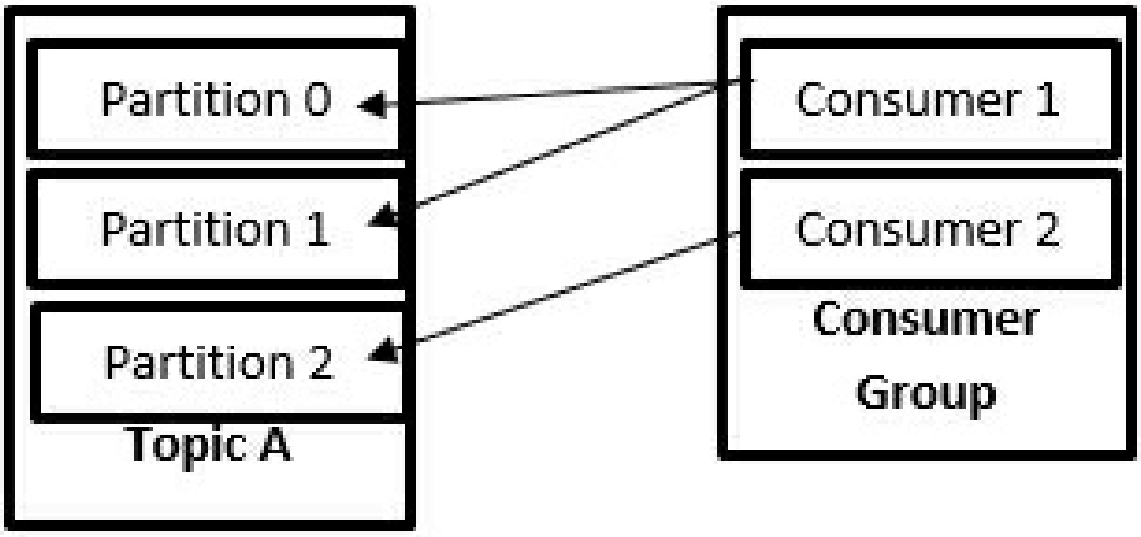
Now that we know how Consumers use the *offsets* to fetch data, let us now understand how Consumers read data from Kafka in an efficient manner.

CONSUMER GROUPS

Kafka uses the Consumer Groups to effectively read data from multiple Partitions of a *Topic* in a distributed fashion. The Consumer Groups consist of multiple Consumers that share a common group identifier. In simple words, multiple Consumers in a Consumer Group actually consume or read data from a single *Topic*. But why do we need multiple consumers to consume from a single *Topic*? The simple answer is: to achieve scalability.

Consider a Consumer application that reads (consumes) data from Kafka with a single Consumer. This application then runs some validations against the data that is received from Kafka and stores the validated data in some other data store. Now, the Producer keeps on generating the data but the Consumer application which reads and processes the data, cannot keep up with the Producer's pace of generating messages. As a result, the Consumer will start trailing and will no longer be able to keep up. The simple solution to this problem is to increase the total number of Consumers. That is where the scalability comes to the rescue, because with multiple Consumers reading and processing the incoming messages, the data consumption is distributed and efficient.

Figure 6(c) below shows two Consumers of a Consumer Group reading messages from a *Topic* that contains three different Partitions.



6(c) One Consumer Group of two Consumers reading from three Partitions

In the this scenario, Consumer 1 is reading messages from Partitions 0 & 1, while Consumer 2 is reading messages from Partition 2. Furthermore, it is also possible to scale the Consumer Group by adding one more Consumer; say Consumer 3, in order to ease the burden on Consumer 1.

Adding Consumers to a Consumer Group helps to achieve better scalability of reading and processing the messages more effectively. But how are the new Consumers assigned Partitions? Who decides which Partition is read by which Consumer? What happens when a Consumer goes down? Let us answer all these questions in the next section.

Group Coordinator and Group Leader

Group Coordinator is the answer for all the questions above. Let's explore what is a Group Coordinator? A Group Coordinator is elected from a pool of Brokers in Kafka. The Group Coordinator receives a 'join request' from the consumers, when they want to join a group. The first Consumer sending the join request becomes the *Group Leader*. The Consumers which join the group later become the normal members of the group. Subsequently, the Group Coordinator sends the list of Consumers available in that particular group to the Group Leader. Then, the Group Coordinator monitors if the Consumers are up by the means of heartbeat. Consumers send heartbeat to the Group Coordinator in regular intervals. In the event of a Consumer going

down or new Consumer joining the group, the Group Coordinator triggers a *Rebalance* instruction to the Group Leader by providing the updated list of available Consumers in that group.

The following are the responsibilities of Group Leader:

- **Assign Partitions to Consumers:** The Group Leader, after receiving the list of Consumers from Group Coordinator, assigns Partitions to all the Consumers of that group. Then, the Consumers start reading messages from their assigned Partitions. Moreover, the Group Leader also assigns Partitions to itself.

- **Executing Rebalance:** When the Group Coordinator triggers a rebalancing activity, the Group Leader revokes the Partitions that are assigned to all the Consumers. This implies that no Consumer in that group is allowed to read messages from their Partition(s). The Group Leader then reassigns the Partitions to the available Consumers. The Consumers are now rebalanced.

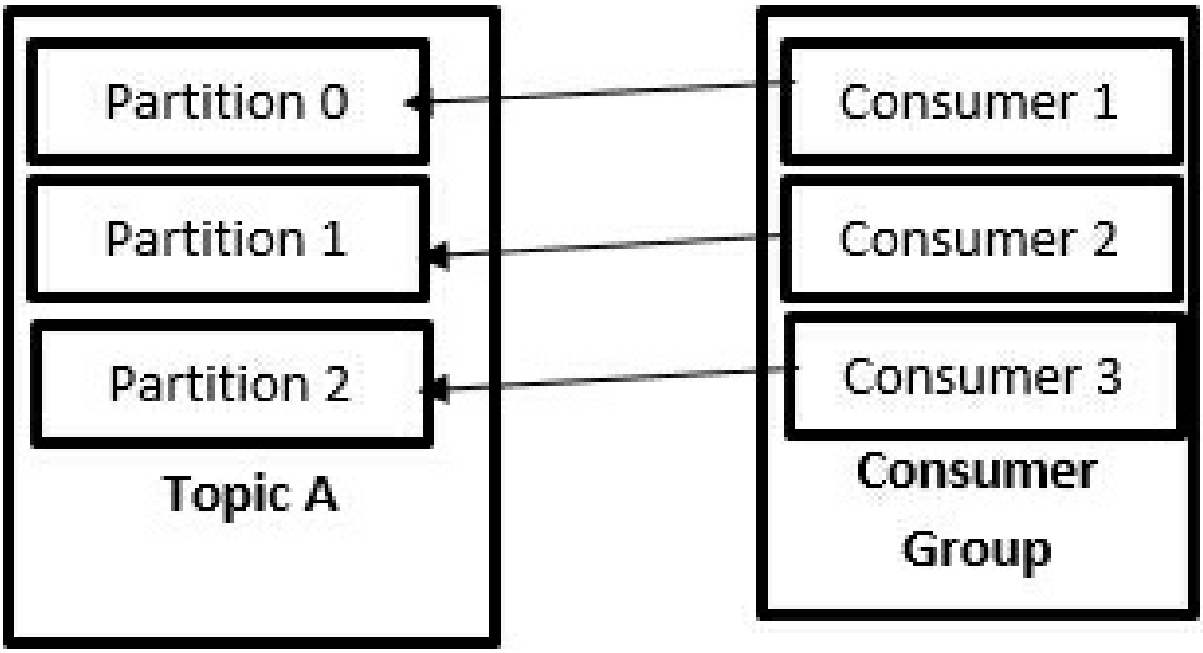
To summarize, the Group Coordinator is responsible to manage the list of available Consumers in a Consumer Group. It triggers the rebalancing activity whenever the list of active Consumers is modified. The Group Leader assigns Partitions to the available Consumers of the group. Moreover, the Group Leader performs the rebalancing activity when triggered by the Group Coordinator.

OFFSET MANAGEMENT

Now that we know how the Consumers read messages from Kafka, let us see how the messages which are read, are committed. Let us determine the concept of offset management by starting from where we had left in the previous section*, i.e.,* the aftermath of rebalancing.

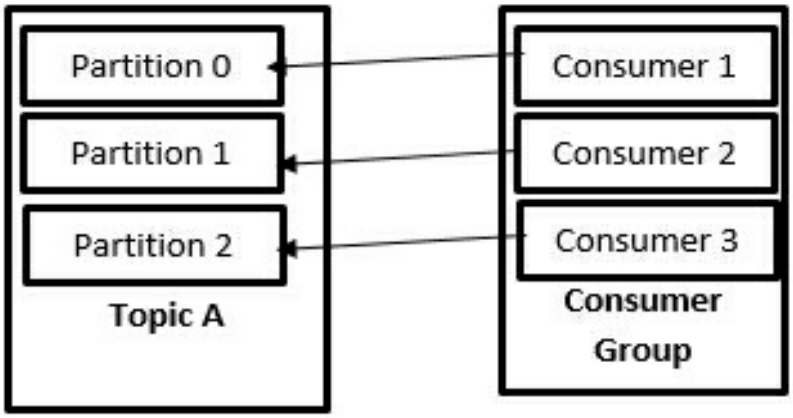
Let us consider a Consumer group with 3 Consumers reading data from three Partitions of a Topic as shown in the Figure 6(d) given below. Consumer 1 reads data from Partition 0, Consumer 2 reads data from Partition 1, whereas Consumer 3 reads data from Partition 2. The Consumers fetch data from the Partitions using Offsets as explained in a

previous section titled as, '*what is the use of Offset?*'



6(d) One Consumer Group of three Consumers reading from three Partitions

Consumers keep fetching data but, out of the blue, the Consumer 2 goes down due to a hardware failure. The Group Coordinator observes that the Consumer 2 has not sent its heartbeat for a while, and marks it as unavailable. Subsequently, the Group Coordinator triggers the rebalance. The Group Leader then executes the rebalancing by revoking the Partitions and, let's say, reassigning the Partition 1 to Consumer 1. But how does the Group Leader know till which offset the Consumer 2 had read data?



6(e) Reassigning Partitions when a Consumer is down

Let's say the Consumers have read data till the following offsets before a rebalance had been triggered.

- Consumer 1 has fetched 200 messages, *i.e.,* till 199th offset from Partition 0.
- Consumer 2 has fetched 202 messages, *i.e.,* till 201th offset from Partition 1 and then it goes down.
- Consumer 3 has fetched 201 messages, *i.e.,* till 200th offset from Partition 0.

We know that the Group leader has reassigned Partition 1 to Consumer 1 (see Figure 6(e)). Thus, from which offset will the Consumer 1 start consuming the messages? It is quite logical that the Consumer 1 should pick up from where Consumer 2 had left, *i.e.,* Consumer 1 should fetch messages from 202th offset. But how does the Consumer 1 know 202 is the offset from which it should start?

The Consumers periodically commit (save) the last offset they had processed to an external storage. This is called *Commit Offset*. The committed offset is the offset till which a certain Consumer had successfully fetched and processed the messages. The external storage to commit offset can be HDFS, MySQL, HBase, Cassandra, *etc*. However, this offset can also be committed within *Kafka* itself.

Auto Commit

The offset committing is controlled using the configuration property called *enable.auto.commit*, and is set to *true* by default. This means that the offsets will be committed automatically after a preset period of interval. The configuration property to specify the interval of auto-commit is *auto.commit.interval.ms*. For instance, if the *auto.commit.interval.ms* property is set to 5000ms, the offsets will be automatically committed after every 5 seconds. Furthermore, the offsets are committed to Kafka, by default. Kafka internally maintains a Topic called *__consumer_offsets* topic for offset commits of each partition.

Let us now go back to our example above, where Consumer 2 is down after processing 202 messages. For simplicity, let us assume the commit interval of 5 seconds has passed since the previous commit, as soon as Consumer 2 had processed 202 messages, and so it has committed the offset as 201. The Consumer 2 has then gone down due to a hardware failure, as soon as it committed the offset. Afterwards, this event will trigger a rebalance and Consumer 1 has been reassigned the Partition 1, from which Consumer 2 had been reading from. Then, Consumer 1 requests the commit offset from Kafka (considering Kafka is used for the offset commit). It will then know that the messages till 201 offset have been successfully processed and it has to start consuming messages from offset of 202.

The *auto commit* option might be an easy and convenient way to commit the offsets automatically. However, it may also lead to duplicate consumption of records. Let us understand this better with an example. Considering the previous example, let us say that the commit interval of 5 seconds was triggered after processing 200 messages, *i.e.,* at 199^{th} offset, and the offset was committed. Now the Consumer 2 has started fetching messages, but after processing 204^{th} offset (Say, two seconds since the offset commit), the Consumer 2 goes down and the rebalance is triggered. The commit offset is 199, but Consumer 2 has processed messages till 204^{th} offset. However, Consumer 2 had already gone down before it could commit the next offset. Consumer 1 gets the Partition reassigned and it observes that the last committed offset is 199. Therefore, Consumer 1 now starts consuming the messages from 200^{th} offset. In this way, the messages from 200 to 205 are duplicated as the messages have been processed again by the Consumer 1.

Manual Commit

One way of overcoming the consumption of duplicate messages is to commit these messages manually. Manual committing of offsets gives additional control to the developers as compared to committing the offsets automatically. The developers can choose to commit the offsets manually at the point of their choice. To commit manually, the *enable.auto.commit* property must be set to *false* and offset should be manually committed after processing the records.

Manual commits can be achieved in the following two ways:

- **Synchronous Commit:** Synchronous Commit is a straightforward method to commit the offsets manually. However, committing synchronously blocks the application until the Broker responds to a commit request; and it also retries the commits in case of recoverable errors. This impacts the overall throughput of the application. The *commitSync* method is used to commit the offsets synchronously.

- **Asynchronous Commit:** Asynchronous Commit, on the other hand, does not block the application. Moreover, it sends the request and continues consumption of messages. However, Asynchronous Commit does not retry recoverable errors. While this may sound as if it is a drawback, there is a valid reason why it does not retry. To understand this process better, imagine a commit request sent to the Broker for the 100^{th} offset. However, due to a temporary network problem, the Broker never receives the request and obviously does not respond. We know that Asynchronous Commit does not wait for the response and keeps on consuming the messages. The consumer has now consumed the next batch of messages and sent another commit request to the broker for the 150^{th} offset. The Broker now receives this request and responds with the committed offset. Since a higher commit is already successful, retrying the previously failed commit does not make sense and also leads to duplicates if a rebalance has been triggered. The *commitAsync* method is used to commit offsets asynchronously.

- **Using Both Sync & Async Commit:** While not committing offsets at times in case of recoverable failures is not a problem with Asynchronous Commits, it poses a potential problem if it is the last offset that must be committed before closing the Consumer. To overcome this problem, we use a combination of both Asynchronous and Synchronous commits. The Asynchronous commit, as usual, keeps committing the offsets without blocking the application. But while closing the Consumer, there is only one final commit. Therefore, to be able to make sure that the offset is committed before we close the consumer, we use the Synchronous Commit. The Synchronous Commit retries to commit the final offset in case of recoverable errors before closing the Consumer.

The combination of Sync and Asynchronous commit methods can be utilized when we want to commit the final offset before closing the Consumer; or to be sure to commit just before the rebalancing is triggered. In the next section, we shall learn how to commit an offset just before a rebalance is triggered.

- **Specific Offset Commit:** We have so far learned how to commit the latest offset after processing a batch of messages. However, what if we want to commit a specific offset in the middle of processing a batch of messages? But again, why do we wish to commit a specific offset in the middle of processing a batch of messages? To answer these questions, let us consider a scenario where a Consumer has received a huge batch of messages to process. The Consumer will take a reasonable amount of time to process this huge batch of messages. Consequently, this will delay the next batch, and therefore, the Group Coordinator may trigger a rebalance, assuming the Consumer is down.

Using the *commitSync* and *commitAsync* methods only commits the latest offsets. Since we want to commit a specific offset and not the latest offset, we can use the *commitSync* and *commitAsync* methods and pass partitions and offsets we wish to commit to a hashmap. Subsequently, we can specify the offset commit based on the

number of records processed.

Let's say the consumer application received a batch of 10,000 messages to process. Since it may take a reasonable amount of time to process these records, we want to commit the offset for every 2000 messages that have been processed. We can achieve this by updating the map object with the processed partitions and offsets. We can then specify a condition that commits offsets after processing 2000 records. This helps the commit-specific offsets, while the application is still consuming a huge batch of messages.

We shall further explore the offset management techniques learned above in the lab exercises.

REBALANCE LISTENERS

The Rebalance Listeners help the developer to commit the latest offset just before a rebalance activity is triggered; and also start consuming messages from where it had left after the rebalance activity. The Kafka Consumer API provides a class called *ConsumerRebalanceListener*. This class has the following two methods that can be implemented:

- **onPartitionsRevoked:** The *onPartitionsRevoked* method is called before the Partitions are revoked from the consumers, *i.e.,* before a rebalance activity is triggered. We have learned from the previous sections that whenever a rebalance activity is triggered, the partitions are revoked from the consumers; and no consumers are allowed to read these messages. This is where we commit the latest offset, so that those consumers which get the partition reassigned, can start consuming messages from this point onwards.

- **onPartitionsAssigned:** The *onPartitionsAssigned* method is invoked after the rebalance activity is complete and before the consumers start consuming the messages again.

The theory part for this chapter is over at this point. Let us now get hands on these topics in the labs that we have learned so far.

AIM

The aim of the following lab exercises is to implement a Kafka Consumer. In addition, we shall also look at the offset management techniques in this lab exercise.

The labs for this chapter include the following exercises:

- Constructing the Kafka Consumer
- Running the Consumer
- The Synchronous & Asynchronous offset commit
- Using both Synchronous & Asynchronous offset commit
- Commit Specified Offset

We require the following packages to perform the lab exercises related to this chapter:

- Java Development Kit (JDK)
- Apache ZooKeeper
- Apache Kafka
- Scala
- IntelliJ IDEA

LAB EXERCISE 6: THE CONSUMER

1. **Constructing a Kafka Consumer**
2. **Running the Consumer**
3. **Synchronous & Asynchronous Offset Commit**
4. **Using both the Synchronous & Asynchronous Offset Commit**
5. **Commit specified offset**

TASK 1: CONSTRUCTING A KAFKA CONSUMER

Construction of a Kafka Consumer has a similar procedure to constructing a Kafka Producer. Let us start creating a Kafka Consumer.

Step 1: Open IntelliJ and create a new Scala object. Name the Scala object as *KafkaConsumer*. Moreover, extend the object to the *App* trait to make it an executable program. Once you have created the Scala object, enter the following required imports:

```
import java.util.*

import org.apache.kafka.clients.consumer.KafkaConsumer
import org.apache.kafka.clients.consumer.ConsumerRecords

import scala.Collection.JavaConverters._

object KafkaConsumer extends App { }
```

Step 2: Now, let us create a properties object and specify the properties in the same way as we did while creating a Kafka Producer.

```
val topics = args(0)
val brokers = args(1)

val props = new Properties()
props.put("bootstrap.servers", brokers)
props.put("group.id", "kafkaGroup1")
props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer

props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer
```

We have first specified the variables for a list of topics and brokers. We will be passing these values as arguments. Next, we create a properties object to specify the properties for our Consumer. Here is the process:

- The *put* method is utilized to specify the configuration properties to the *props* object.

- The *bootstrap.servers* property is used to specify the Bootstrap servers. Bootstrap servers are a list of servers (Brokers) utilized by the clients (Producers and Consumers) to establish an initial connection with the Kafka cluster.

- The *group.id* is an optional property that is used to set a unique identifier to the Consumer Group name. While it is not mandatory to specify the name of a Consumer Group, it is recommended, because without a Consumer Group, all the messages are received and processed by only one of the independent Consumers. Having a Group name will have multiple Consumers reading the messages from a Topic's Partitions and thus distributing the workload amongst all the Consumers.

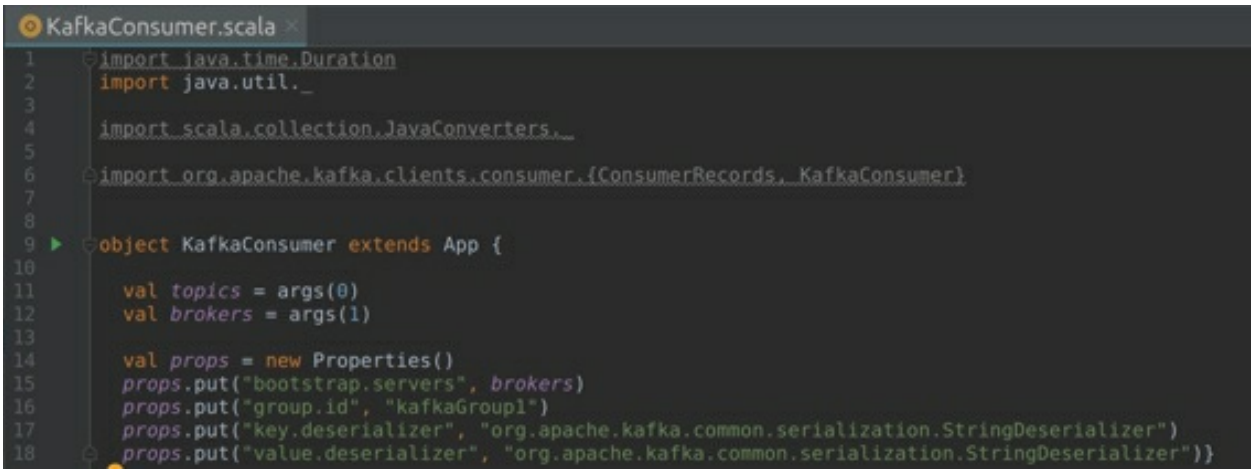
The Consumer API takes care of electing a Group Coordinator and Group Leader. The Consumer Group is created when we specify the group name. All we require to specify a String for a group name.

- The *key.deserializer* is used to specify the deserialization for key while the *value.deserializer* is used to specify the deserialization for the value. Since our message is of type *String* in both key and value, a *StringDeserializer* is utilized for both the key and value. However, you can also use an *IntDeserializer, DoubleDeserializer, LongDeserializer, JSONDeserializer etc,* if the keys or values are of that type.

The deserializers in Kafka include the following:
ByteArrayDeserializer, ByteBufferDeserializer, ExtendedDeserializer. Wrapper, BytesDeserializer, DoubleDeserializer, IntegerDeserializer, FloatDeserializer, LongDeserializer, SessionWindowedDeserializer, ShortDeserializer, StringDeserializer, TimeWindowedDeserializer, and

UUIDDeserializer.

The following screenshot shows the resultant code being generated.



Step 3: Now that we have specified the required properties, let us now instantiate a new Kafka Consumer object, and pass the properties as the arguments.

```
val consumer = new KafkaConsumer[String, String](props)
```

Once we have the Kafka Consumer object instantiated, we have to subscribe to the Topic, so that the Consumer can start reading data from that Topic. This can be performed by using the *subscribe* method within the Kafka Consumer API.

```
consumer.subscribe(Collections.SingletonList(topics))
```

Additionally, we can subscribe to multiple topics using a regular expression or an Arrays List. Using regular expression ensures to subscribe to all the topics matching the regex. Whenever a new Topic is created, that matches

the regex, the rebalance triggers and the Consumers start consuming from the new Topic.

Step 4: The next step is to initiate the process of reading data from the Topics to which the Consumer is subscribed. To do this, the Kafka Consumer provides a *poll* method. The *poll* method internally takes care of the coordination, rebalance, and heartbeats; and then fetches data. A developer has to simply subscribe to the Topic from which the messages must be read, and the messages are fetched from the Topic's partitions.

```
try{
     while(true){ //1
          val records: ConsumerRecord[String, String] =
          consumer.poll(Duration.ofMillis(100)) //2

for(record <- records.asScala){ //3

println("Topic: " + record.topic()
          + " ,Key: " + record.key()
          + ", Value: " + record.value()
          + ", Partition: " + record.partition()
          + ", Offset: " + record.offset()) //4

     }
   }
}catch{
     case e:Exception => e.printStackTrace()
}finally {
     consumer.close() //5
   }
}
```

1. This is the infinite *while* loop. The infinite *while* loop makes sense because, the Consumers are usually long-running applications and keep fetching data from the Kafka Brokers. Furthermore, the Consumers can be scheduled to run for a few hours and then sleep for a few hours using a scheduler. Moreover, it is possible to exit the loop using the *consumer.wakeup* method by executing it from a shutdown hook in a different thread.

2. We utilize the *poll* method on our consumer object to fetch data. The *poll* method takes an argument to specify the timeout interval in milliseconds. This is the time for which the Consumer has to wait for the messages from the Broker. The Kafka Consumers must keep polling continuously for the messages. If they do not poll, they are considered dead and the rebalance activity starts to trigger.

3. The *poll* method once executed fetches the records from the Kafka Brokers. The records consist of the name of the Topic, partitions, offsets, keys, and values. We use the '*for*' loop to iterate over the records.

4. Subsequently, we print the name of the Topic, key, value, partition, and offset values of each record to the console. This is the point where these messages are usually stored to a storage or a database. This is printed out on the console for the purpose of simplicity.

5. Finally, we take care of the error handling before closing the consumer. You should always close the consumer so that it will trigger a rebalance activity by closing the network connections and sockets.

```
try {
  while (true) {
    val records: ConsumerRecords[String, String] = consumer.poll(Duration.ofMillis( millis = 100))

    for (record <- records.asScala) {
      println("Topic: " + record.topic()
        + ", Key: " + record.key()
        + ", Value: " + record.value()
        + ", Partition: " + record.partition()
        + ", Offset: " + record.offset())

    }
  }
} catch {
  case e: Exception => e.printStackTrace()
} finally {
  consumer.close()
}

}
```

This completes the implementation of a Kafka Consumer. A Kafka Consumer can be extremely complex depending upon the requirement, but in this lab, we have only learned a simple Kafka Consumer for better understanding and simplicity.

Let us run this Kafka Consumer in the next task.

Task 1 is complete!

TASK 2: RUNNING THE CONSUMER

Step 1: Now that we have finished implementing the Kafka Consumer, let us start the ZooKeeper and Kafka server in the terminal.

```
$ zkServer.sh start
$ kafka-server-start.sh /usr/share/kafka/config/server.properties
```

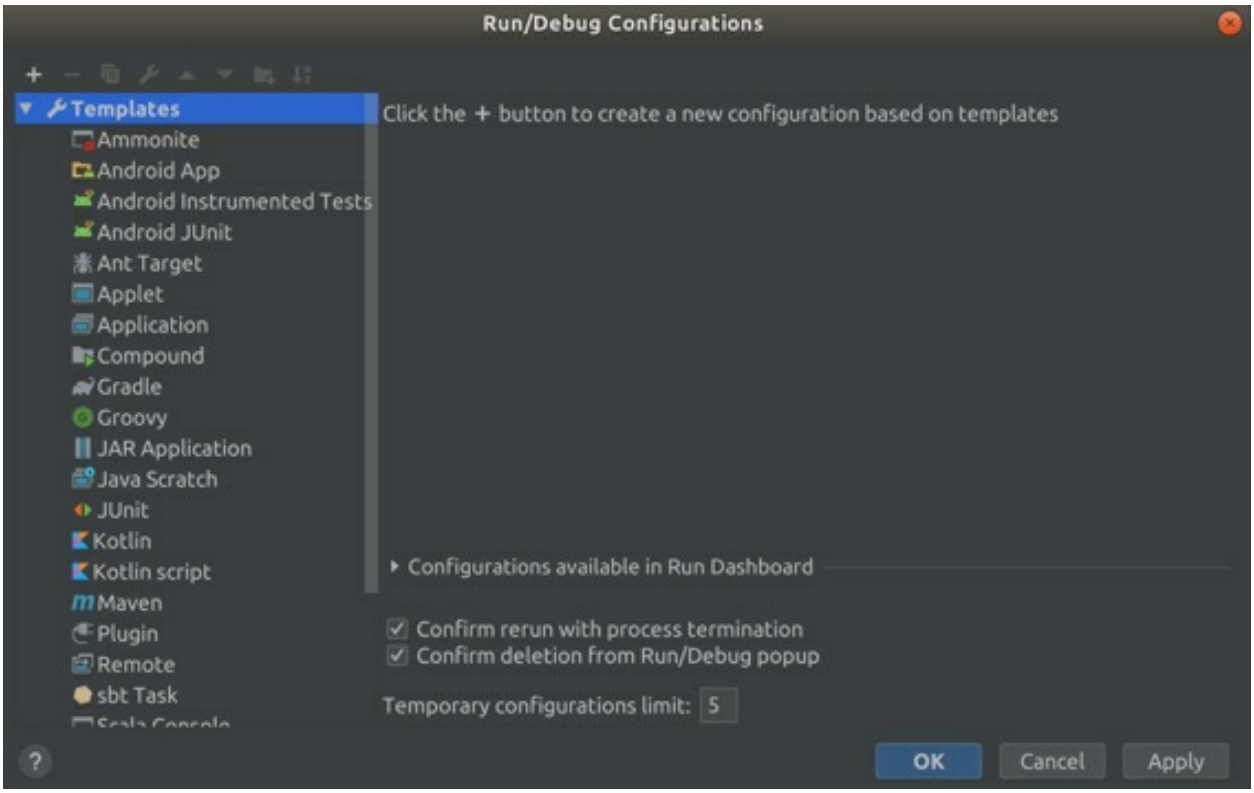
```
uzair@uzair:~$ zkServer.sh start
/usr/bin/java
ZooKeeper JMX enabled by default
Using config: /usr/share/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
uzair@uzair:~$ kafka-server-start.sh /usr/share/kafka/config/server.properties
[2020-09-29 11:25:09,169] INFO Registered kafka:type=kafka.Log4jController MBean
 (kafka.utils.Log4jControllerRegistration$)
[2020-09-29 11:25:09,936] INFO Setting -D jdk.tls.rejectClientInitiatedRenegotia
tion=true to disable client-initiated TLS renegotiation (org.apache.zookeeper.co
mmon.X509Util)
[2020-09-29 11:25:10,025] INFO Registered signal handlers for TERM, INT, HUP (or
g.apache.kafka.common.utils.LoggingSignalHandler)
[2020-09-29 11:25:10,037] INFO starting (kafka.server.KafkaServer)
[2020-09-29 11:25:10,040] INFO Connecting to zookeeper on localhost:2181 (kafka.
server.KafkaServer)
```

Step 2: We have already created Topics and have run our Producer application to produce the messages. Before proceeding to the next step, make sure that you have the *logs* Topic containing the messages produced by the Producer that we had implemented in the previous chapter's lab exercise.

If the Topic (or the messages within the Topic) does not exist, please go through the tasks 1 through 3 before proceeding to the next step.

Step 3: Switch back to the IDE, click on *Run* and select *Edit Configurations…* option. You should see the configurations window as depicted in the

screenshot below.

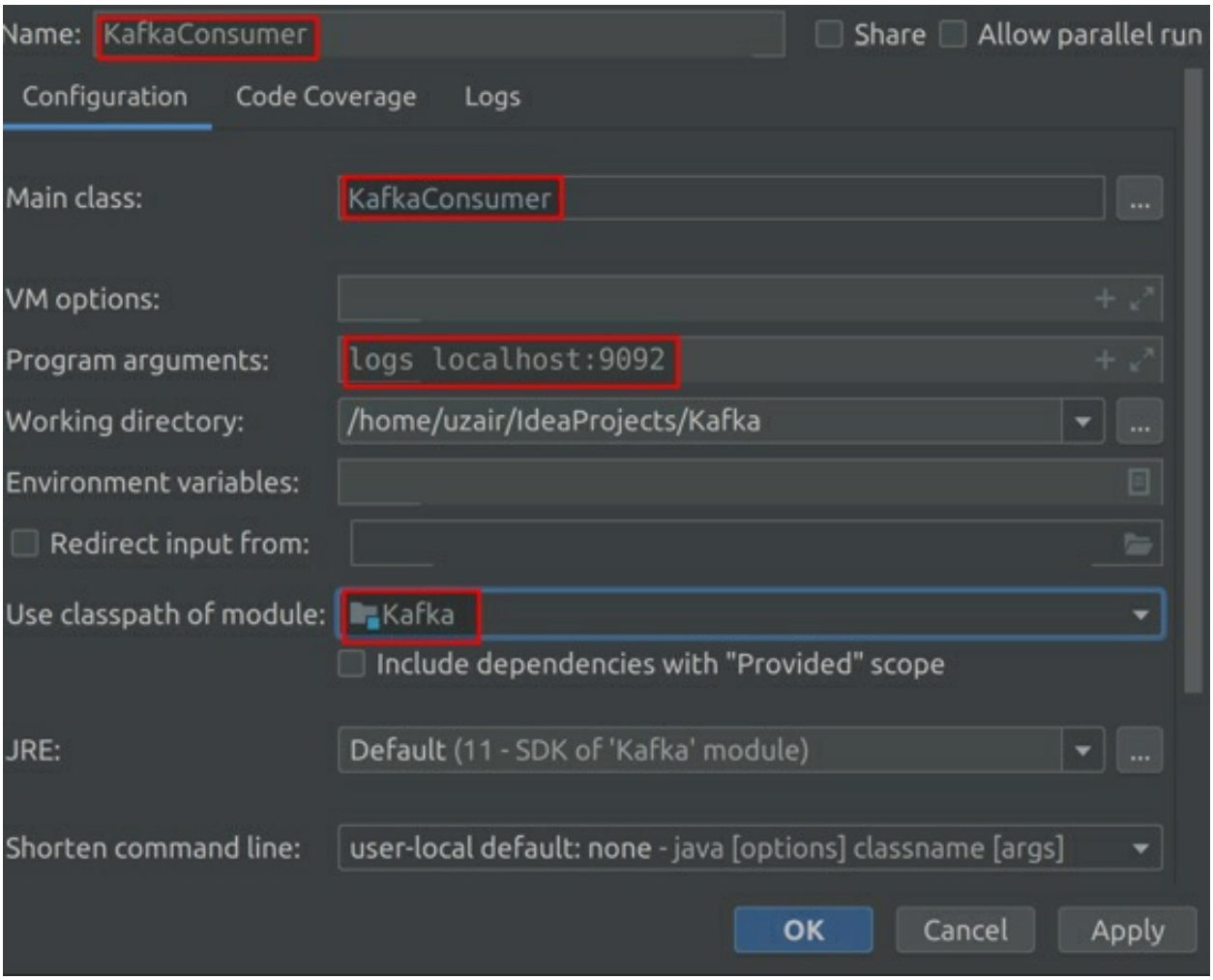


Click on the + icon on the top left of the window; and select Application from the drop-down list. Enter any name (of your choice) in the name field. Here, we have named it as *KafkaConsumer*. Click on the … button for *Main Class* field and select the class.

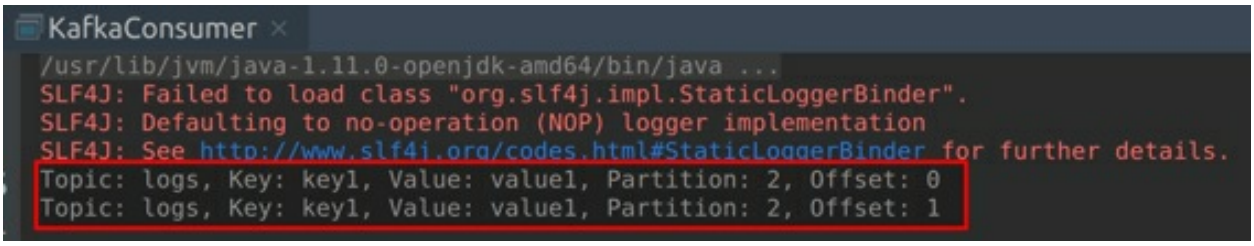
Next, input the program arguments as shown below:

```
logs localhost:9092
```

Finally, if *Use classpath of module* is empty, select the value *Kafka* from the drop-down menu. You should find all the values as depicted in the screenshot below:



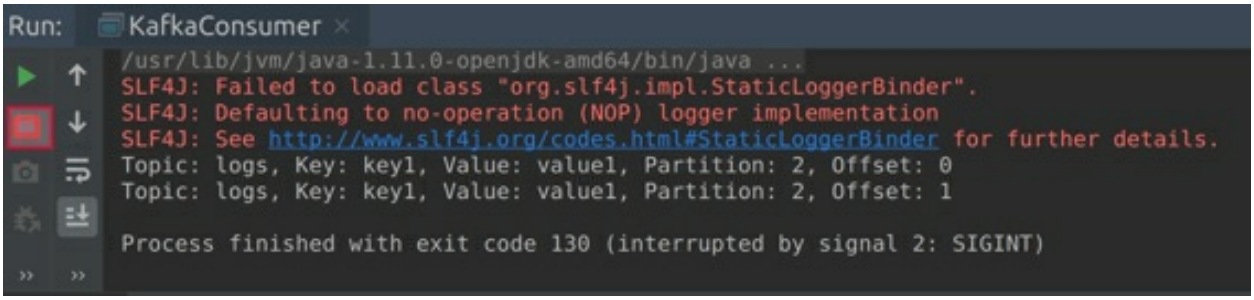
Step 4: Let us run the Kafka Consumer. Click on the green play icon to the right of the configuration drop-down menu. After a while, you should see that the Consumer has run successfully by displaying the records as depicted in the screenshot below.



Step 5: Since the Consumer is polling on an infinite loop, it would not exit and will keep on polling for data from the Kafka Brokers. Click on the red stop button as depicted in the screenshot to stop the Consumer. While, this is not the best exit method, we have only used this, in order to stop the

Consumer from going into an infinite loop.

Subsequently, you should implement a shutdown hook or a scheduler to safely exit the Consumer in the production environment.



We have simply printed the records on the console for this lab exercise. However, as a lab challenge, try to process all these records and store them in an external storage or a database.

Task 2 is complete!

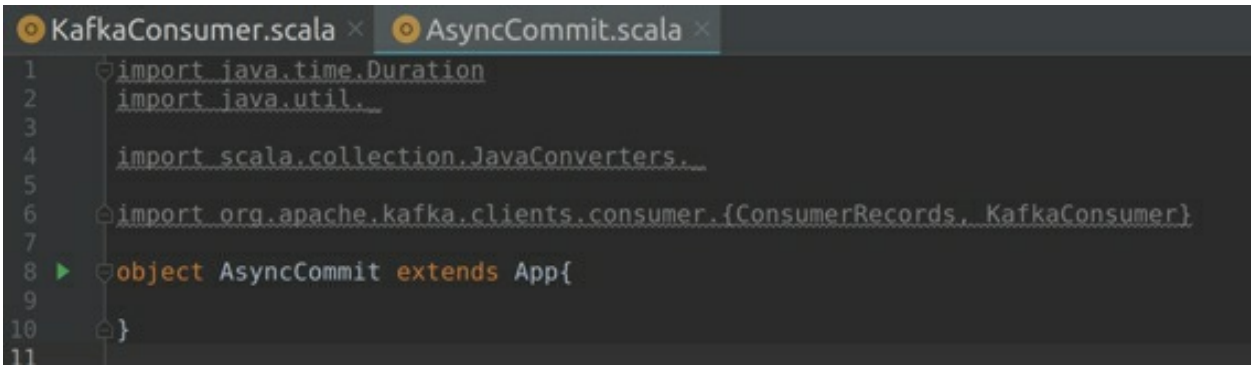
TASK 3: SYNCHRONOUS & ASYNCHRONOUS OFFSET COMMIT

Let us now implement a Kafka Consumer that commits offsets synchronously as well as asynchronously. Please check the Manual Commit section in Offset Management for more information on the manual committing offsets.

Let us first look at Asynchronous Commit:

Step 1: Create a new Scala object and name it *AsyncCommit*. Moreover, extend the object to *App* trait to make it an executable program. Once you have created the Scala object, add the following required imports:

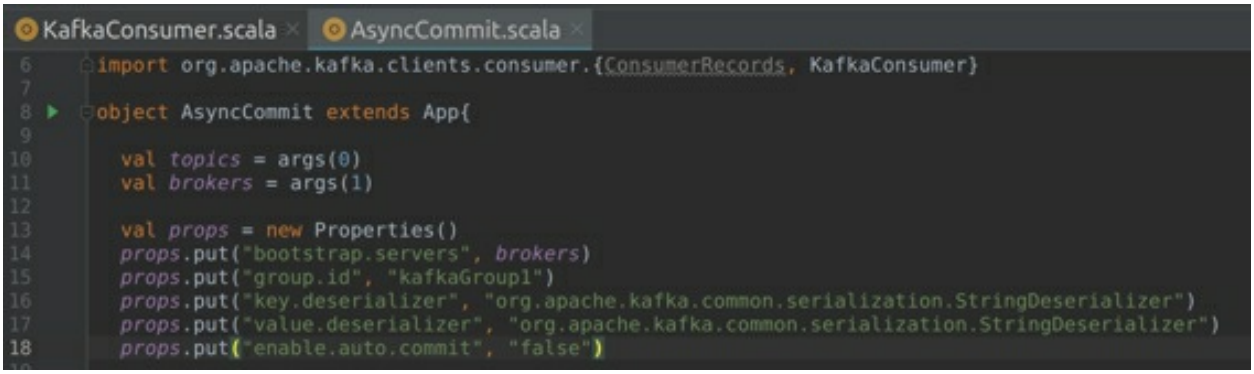
```
import java.util.*

import org.apache.kafka.clients.consumer.KafkaConsumer
import org.apache.kafka.clients.consumer.ConsumerRecords

import scala.Collection.JavaConverters._

object AsyncCommit extends App { }
```



Step 2: Create a new *properties* object and specify the required properties as in step 2 of Task 1. However, ensure to add a new *property* that disables the auto commit.

```
props.put("enable.auto.commit", "false")
```

So far, you should have the code as depicted in the screenshot below:



Step 3: Let us now instantiate a new Kafka Consumer object and pass the *properties* as arguments.

```
val consumer = new KafkaConsumer[String, String](props)
```

Once we have the Kafka Consumer object instantiated, we have to subscribe to the Topic, so that the Consumer can start reading data from that particular Topic. We can do this by utilizing the *subscribe* method within the Kafka Consumer API.

```
consumer.subscribe(Collections.SingletonList(topics))
```

Step 4: Now that we have a Kafka Consumer object instantiated, let us use the *poll* method to fetch the records. We then use the *fetch* records to iterate by using the '*for'* loop and display the records on the console.

Till now, the code is almost the same as we described while creating a Kafka Consumer we had seen in Task 1.

So far, you should have the code as depicted in the screenshot below:

```
val consumer = new KafkaConsumer[String, String](props)
consumer.subscribe(Collections.singletonList(topics))

try {
  while (true) {
    val records: ConsumerRecords[String, String] = consumer.poll(Duration.ofMillis( millis = 100))

    for (record <- records.asScala) {
      println("Topic: " + record.topic()
        + ", Key: " + record.key()
        + ", Value: " + record.value()
        + ", Partition: " + record.partition()
        + ", Offset: " + record.offset())

    }
```

Step 5: Now, it is the right time to commit the *offset* as we have fetched the first batch of records and processed them accordingly. We can commit the offsets asynchronously using the c*ommitAsync* method as shown below:

```
consumer.commitAsync()
```

Next, we perform the error handling as seen in Task 1. The final code should now match with the code in the screenshot given below:

KafkaConsumer.scala | AsyncCommit.scala

```
    try {
      while (true) {
        val records: ConsumerRecords[String, String] = consumer.poll(Duration.ofMillis( millis = 100))

        for (record <- records.asScala) {
          println("Topic: " + record.topic()
            + ", Key: " + record.key()
            + ", Value: " + record.value()
            + ", Partition: " + record.partition()
            + ", Offset: " + record.offset())
        }
        consumer.commitAsync()
      }
    } catch {
        case e: Exception => e.printStackTrace()
      } finally {
      consumer.close()
    }
  }
```

The offsets are now committed *Asynchronously* every time when we fetch and process a batch of records.

To implement the Synchronous offset commit, simply replace the *consumer.commitAsync* method with the *consumer.commitSync* method.

As a lab challenge, please implement a Kafka Consumer to commit *offsets* synchronously.

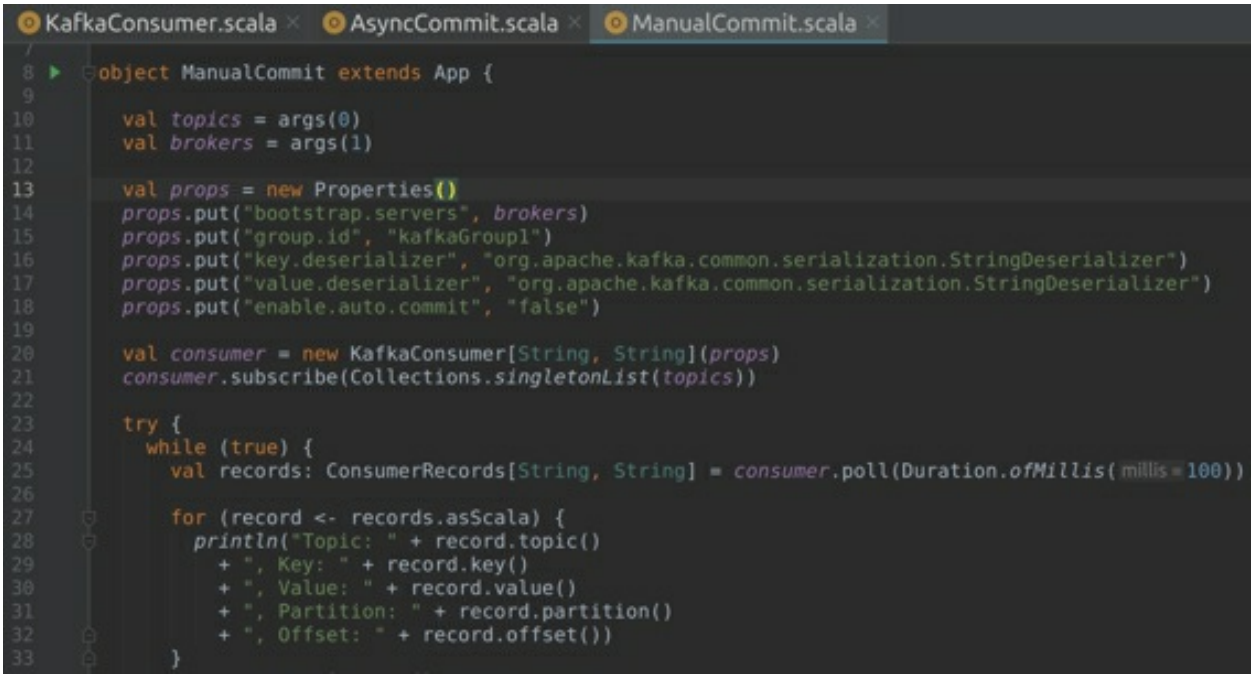
Task 3 is complete!

TASK 4: USING BOTH SYNCHRONOUS & ASYNCHRONOUS OFFSET COMMIT

In the previous task, we have learned how to commit offsets using the *Synchronous* and *Asynchronous* methods individually. Let us now see how we can commit *offsets* using both sync and async methods in the same application.

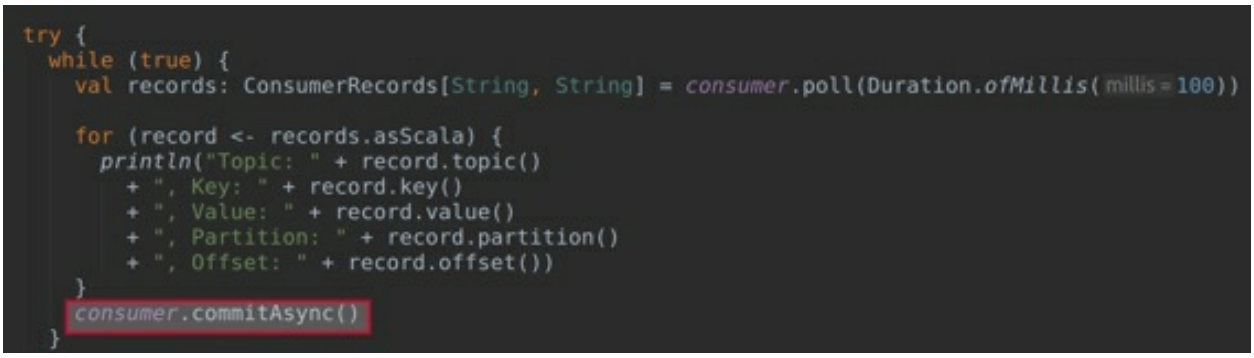
Step 1: Please follow the Steps 1 through 4 in the previous task. However, do change the name of Scala object to *ManualCommit* or any other suitable name.

Your code should now match the code depicted in the screenshot below:



Step 2: We know that *AsyncCommit* method does retry for recoverable errors, if the *offset* commit was not successful. But successfully committing a higher *offset* for next batch compensates for this failure. Therefore, we would first commit *offsets* asynchronously after processing every batch of records.

```
consumer.commitAsync()
```



Step 3: Asynchronous commit works well, when there are multiple batches to be consumed after the current batch. However, if it is the last batch before closing the Consumer, we must ensure that the last *offset* is successfully committed. If the last *offset* was not successfully committed, the next Consumer again fetches the records from the previous *offset* leading to duplicate messages.

To overcome this scenario and to ensure that the last *offset* just before closing of Consumer is committed, we utilize the *consumer.commitSync* method. The *consumer.commitSync* method retires for recoverable errors by blocking the application until it successfully commits the *offset*.

Thus we invoke the *consumer.commitSync* method just before closing the consumer as shown below:

```
…
}
}catch{
      case e:Exception => e.printStackTrace()
}finally {
      consumer.commitSync()
      consumer.close()
   }
}
```

The *consumer.commitSync* method is invoked just before closing the Consumer as shown in the screenshot below:

```
try {
  while (true) {
    val records: ConsumerRecords[String, String] = consumer.poll(Duration.ofMillis( millis = 100))

    for (record <- records.asScala) {
      println("Topic: " + record.topic()
        + ", Key: " + record.key()
        + ", Value: " + record.value()
        + ", Partition: " + record.partition()
        + ", Offset: " + record.offset())
    }
    consumer.commitAsync()
  }
} catch {
  case e: Exception => e.printStackTrace()
} finally {
  consumer.commitSync()
  consumer.close()
}
}
```

From Task 4, we have learned that we can use both *commitSync* and *commitAsync* methods in the same application.

Task 4 is complete!

TASK 5: COMMIT SPECIFIED OFFSET

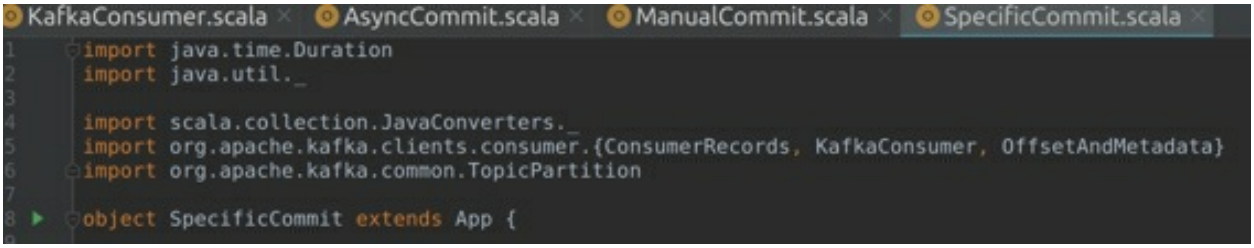
Till now, we have learned to commit the latest offset. Let us now learn about committing a specific offset.

Step 1: Create a new *Scala* object and name it *SpecificCommit.* Moreover, extend the object to *App* trait to make it an executable program. Once you have created the *Scala* object, enter the following required imports:

```
import java.time.Duration
import java.util._

importscala.collection.JavaConverters._

import                    org.apache.kafka.clients.consumer.
{ConsumerRecords, KafkaConsumer, OffsetAndMetadata}
import org.apache.kafka.common.TopicPartition

object SpecificCommit extends App{}
```

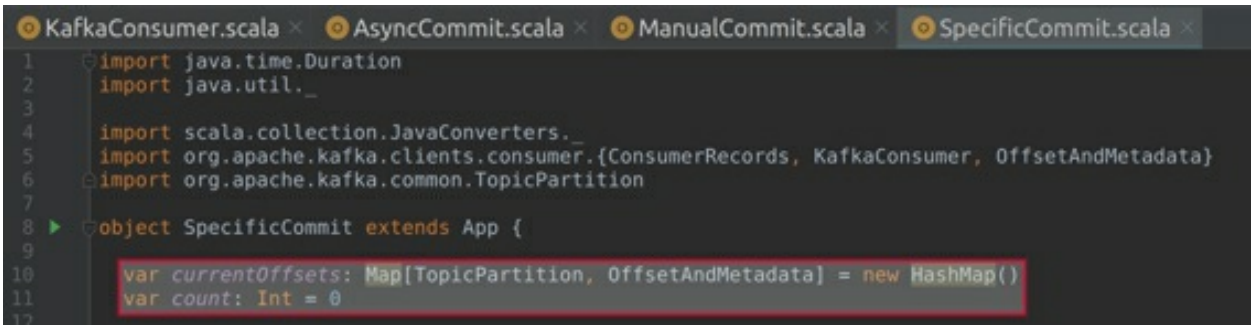
Here, the new imports are *OffserAndMetadata* and *TopicPartition*. The *OffserAndMetadata* is a Kafka offset commit API, which allows the users to provide the additional metadata (in the form of a string) when an *offset* is committed. This can be useful to store information about which node made

the *commit*, what time the commit was made, *etc*. The *TopicPartition* class provides methods to extract the Topic name and the Partition number.

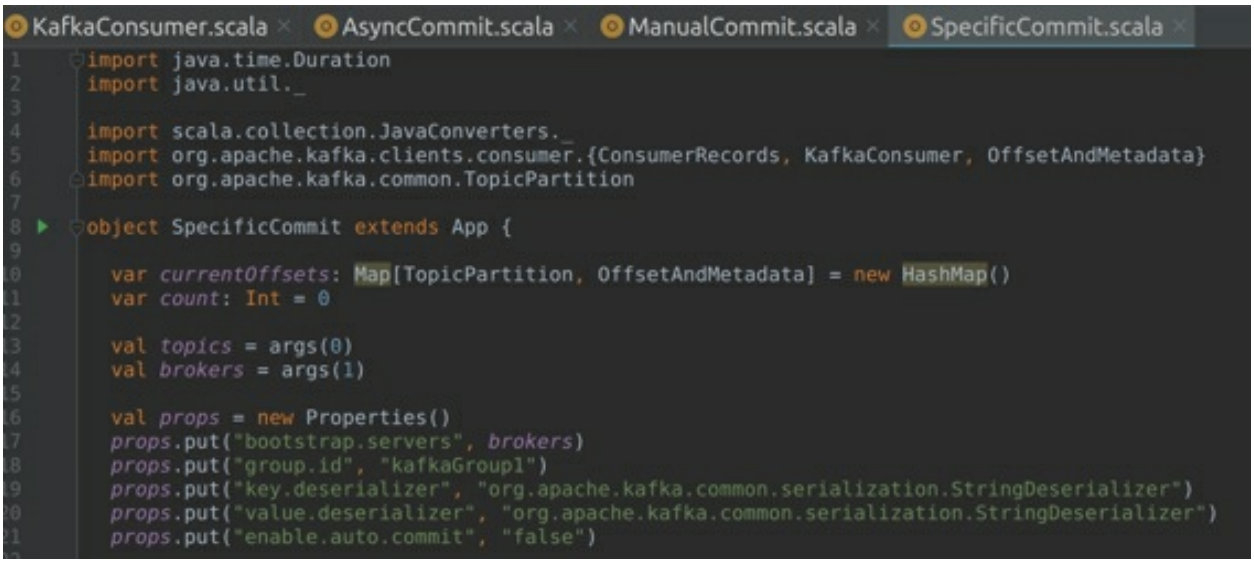


Step 2: Let us now create the map object to manually track the *offsets*. Let us also declare a count variable, so that we can increment it for every processed record.

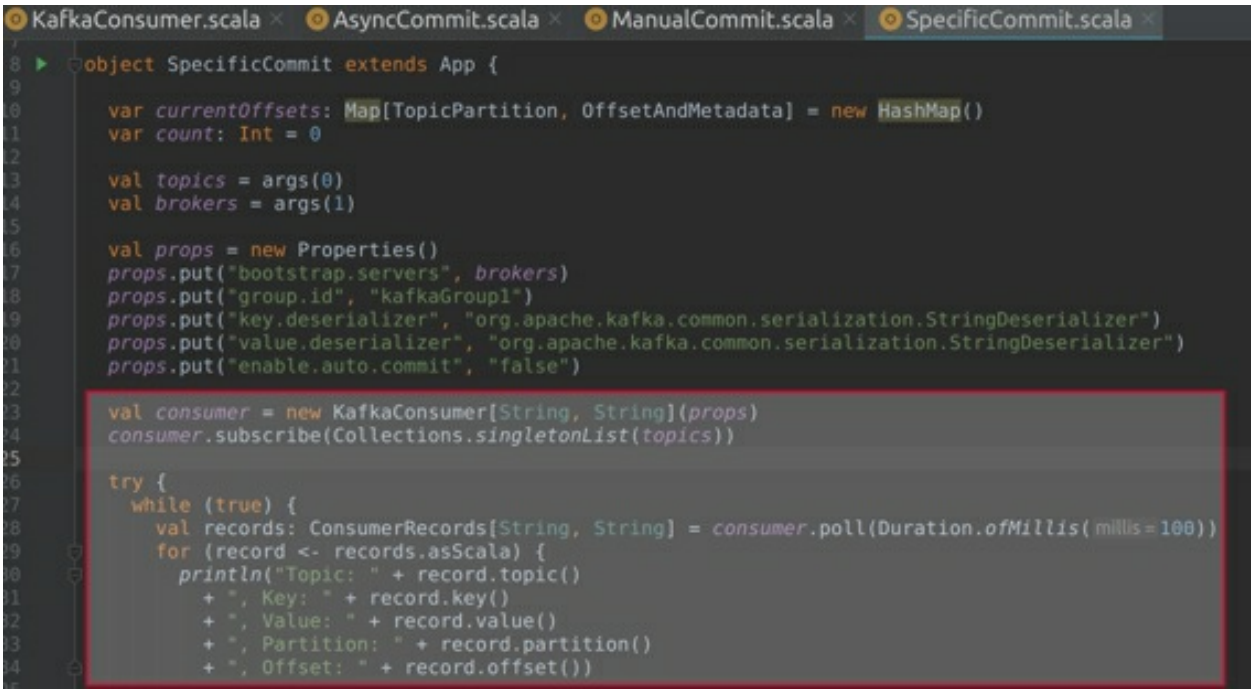
```
var currentOffsets: Map[TopicPartition, OffsetAndMetadata] = new HashMap()
var count = 0
```



Step 3: The next step is to create the *properties* object and specify the *properties*. Your code should now match with the code given in the screenshot below:



Step 4: Now create a Kafka Consumer object and subscribe to the Topic. Then start the poll loop to fetch records and process by displaying them on to the console.



Step 5: After processing the records by displaying them on the console, we update the *offset* map with the *offset* number of the next message to process. This way, we can manually track every offset from the Map object.

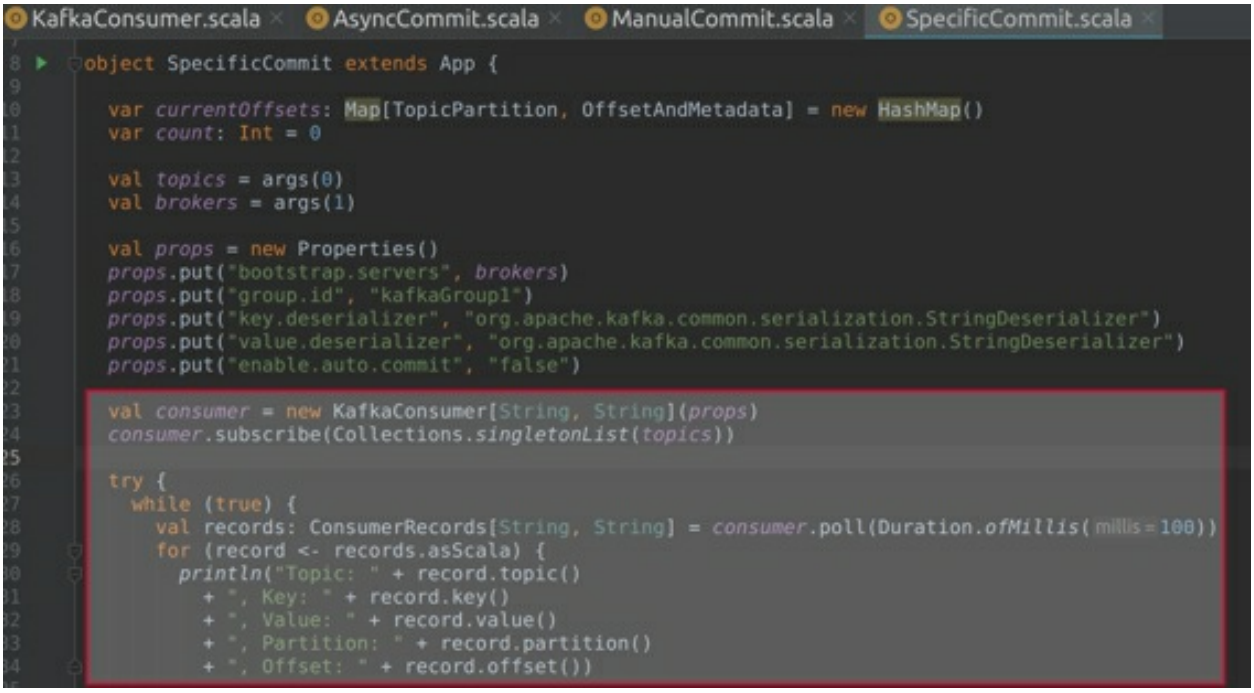
```
currentOffsets.put(new TopicPartition(record.topic(),
record.partition()), new
```

```
OffsetAndMetadata(record.offset() + 1, "no metadata"))
```

Next, we commit the *offset* for every 5000 records that have been processed using the '*if*' condition. As you can observe, we have control on committing the specified *offset* instead of only committing the latest *offset*. You may choose to commit after any number of records according to your requirement. The 5000 records in this example have been shown as an example:

```
if (count % 1000 == 0) {
consumer.commitAsync(currentOffsets, null)
count += 1
}
```

We finally use the *commitAsync* method to commit the *offsets*. Moreover, you can also choose to utilize the *commitSync* method.



Task 5 is complete!

SUMMARY

The Consumer is the Kafka component that consumes messages from Kafka by making use of the Kafka Consumer APIs.

Kafka utilizes the Consumer Groups to effectively read data from multiple Partitions of a Topic in a distributed fashion. The Consumer Groups consist of multiple Consumers that share a common group identifier. In other words, multiple Consumers related to a Consumer Group can read data from a single Topic.

Offset is the integer metadata associated with each message. The *offset* increases monotonically for every message. The produced message consists of a key and a value. Once the Producer transmits the message to the Broker, the Broker assigns an offset to it, *i.e.*, an integer starting from zero.

The Consumers periodically commit (*i.e.,* save) the last *offset* they have processed to an external storage. This is called the Commit Offset. The committed offset is an *offset* point till which the Consumer has successfully fetched and processed all the messages. The external storage (or databse) to commit *offset* can be HDFS, MySQL, HBase, and Cassandra, *etc*. The *offset* can be committed within Kafka itself. Finally, the *offsets* can be committed automatically as well as manually.

In the labs, we implemented the Kafka Consumer and the various other approaches to commit the *offsets* manually.

CHAPTER 7:
KAFKA DATA DELIVERY

Theory

In the previous chapter, we learned about the *offsets* and how they are managed. Furthermore, we learned that Kafka itself does the *offset* management by committing the *offsets* to Kafka. In this chapter, let us learn how to manage the committing of *offsets* to the external storage by understanding the delivery semantics.

DELIVERY SEMANTICS

The messages in Kafka can be delivered in the following three (possible) ways:

- **At least once**: No messages had been lost, but there may have been duplicate messages.
- **At most once**: Messages had been lost, but there are no duplicates.
- **Exactly once**: Neither messages were lost nor there were duplicate messages. Each message is delivered exactly once.

Let us understand these delivery semantics in detail below.

At least Once Semantics

By default, Kafka is configured for "*at least once semantics*". The scenario which we have observed in the offset management section in the previous chapter is "*at least once semantics"*. As a developer, there is nothing else to do if the business requirement is "*at least once semantics*", *i.e.,* if you are OK to receive the duplicate messages without any message loss. However, if you would like to do the *offset* management yourself, the following steps

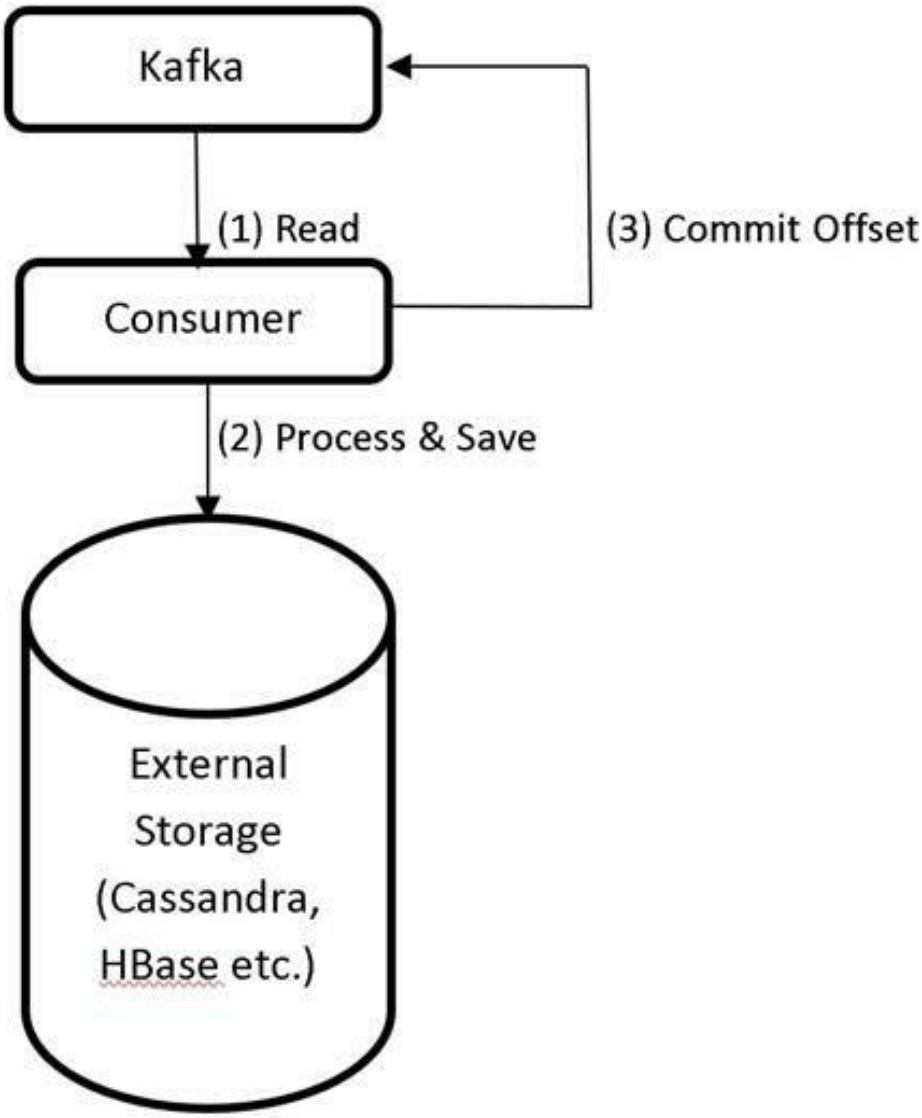
should be followed:

1. Read the messages from Kafka.

2. Process the messages and save them in an external storage, such as Cassandra, HBase, and Elastic search *etc*. However, you may also choose to save it within Kafka itself.

3. Commit the *offset*.

The following figure explains how the "*at least once semantics"* process works:



7(a) At least once Semantics

Let's understand this better with the following example:

- Consider a *logs* Topic to which the Consumer has read 100 messages with few poll requests earlier.

- The next poll request will start from 100th offset. We receive, say, 30 messages for this poll request. The messages are then processed and saved in an external storage. Afterwards, we commit the *offset* as 130. Let's assume that we are committing the offsets in Kafka itself for this example.

- Similarly, 40 more messages are processed and saved in the next poll request. The *offset* is committed as 170.

- Another poll request is made to fetch more messages. We receive, say, 50 more messages in this batch. Subsequently, these messages are processed and saved. However, before committing the *offset* as 220, the application crashes down due to a hardware or network failure.

- At this point, the number of processed messages is 220, but the last committed *offset* is 170.

- After the rebalance, a new Consumer is assigned to this Partition. The Consumer fetches the latest committed *offset* as 170, and so it sends a poll request for the messages from 170th *offset*. This results in the duplication of last 50 messages, as these messages are being read again.

This is how the "*at least once semantics"* work. This delivery semantics is developed in such a way that the messages are processed first and then the *offset* is committed. With "*at least once semantics"*, there is no loss of messages, but there can be duplicate messages in case of failure.

At most Once Semantics

The following steps have to be performed to achieve the "*at most once semantics"*.

1. Read the messages from Kafka.

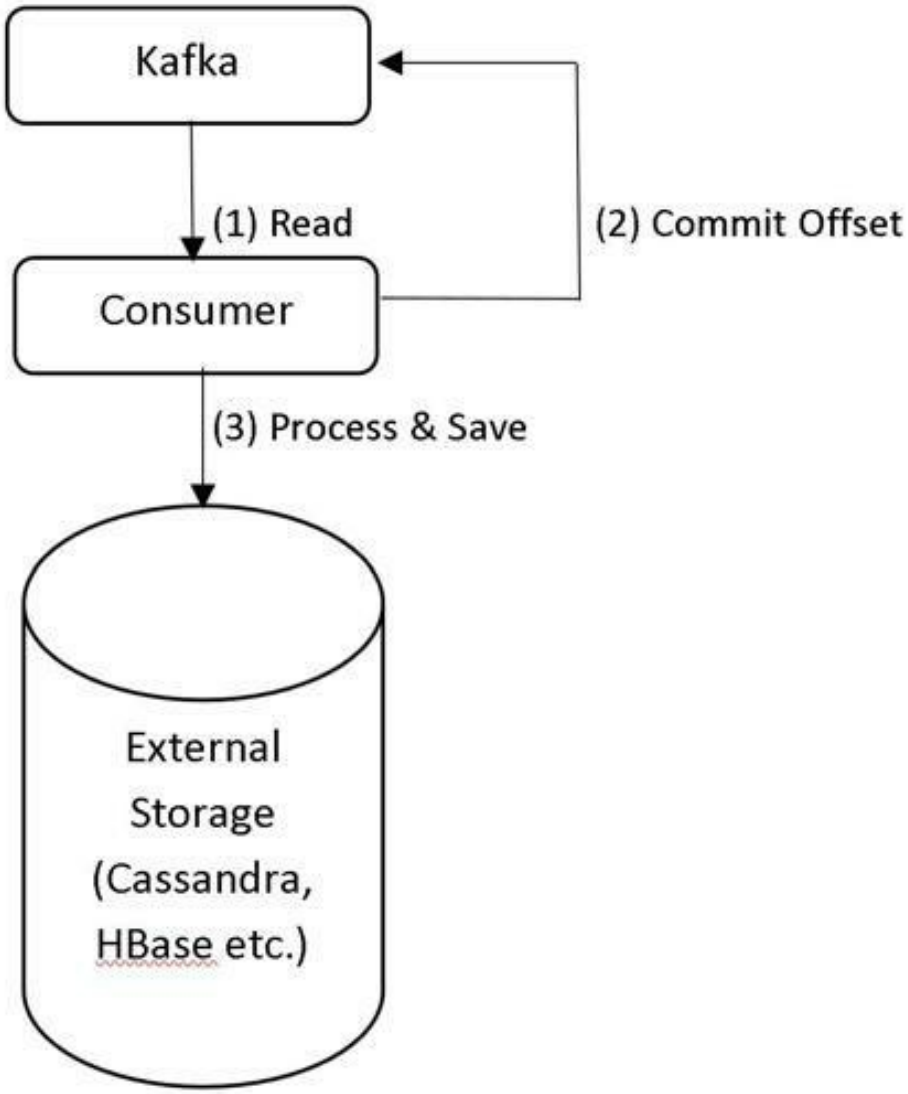
2. Commit the *offset*.

3. Process the messages and save them in an external storage such as Cassandra, HBase, and Elastic search *etc*. Moreover, you may also choose to save it within Kafka itself.

Here, we first commit the offset immediately after reading the messages and then process them accordingly. With "*At most once semantics"*, we may lose our data, if a failure occurs as soon as we commit the *offset*.

Let's understand this process better with the following example:

- Consider a *logs* topic to which the Consumer has read 100 messages with a few poll requests earlier.



7(b) At most once Semantics

- The next poll request starts from the 100th *offset*. We receive, say, 30 messages for this poll request. Instead of processing the records and saving them to an external storage, we commit the *offset* first. The *offset* is committed as 130.
- After committing the *offset*, we process and save the messages.

- Similarly, the next poll request starts from the 130th *offset*. We receive, say, 40 messages for this particular poll request. The *offset* is committed as 170. Subsequently, the messages are processed and saved in an external storage.

- Again the poll request is made to fetch more messages. We receive, say, 50 more messages in this batch. The *offset* is committed as 220. However, after committing the *offset* as 220, the application crashes down due to a hardware or network failure. The messages received in this particular batch could never be processed.

- At this point, the last committed *offset* is 220 but the messages fetched from Kafka have only been processed till the *offset* of 170.

- After the rebalance, a new Consumer is assigned to this Partition. The Consumer fetches the latest committed *offset* as 220 and so it sends a poll request for messages from 220th offset. This results in the loss of last 50 messages, as these messages could never be processed.

This is how the "*at most once semantics"* works. This delivery semantics is developed in such a way that the *offset* is committed first and then the messages are processed and saved. With "*at most once semantics"*, there is a loss of messages but there will be no duplicate messages in case of a failure. The use-case for this semantics would be a scenario where you can afford a loss of data, but you cannot have duplicates.

Exactly Once Semantics

The "*Exactly once*" semantics in Kafka ensures that each message is processed exactly once, with neither duplicates nor data loss. This is the most sought-out delivery semantics which guarantees that the messages are processed without any duplicates or data loss.

In order to achieve the "*exactly once*" semantics, we must make sure that if the processing and saving of messages fails, the *offset* committing will also

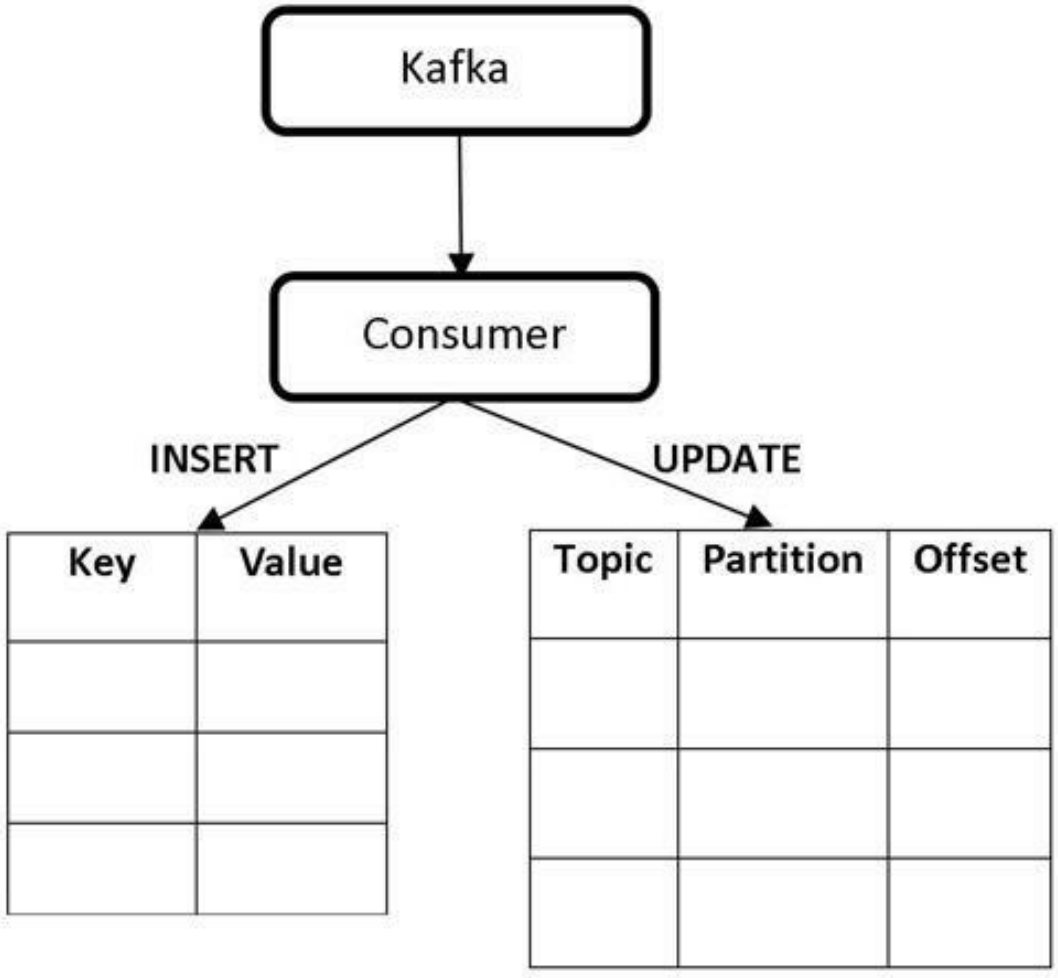
fail. Similarly, if you are committing first and then processing the messages, the processing should also fail if the committing fails. In case of failure, this will ensure that the new Consumer assigned to a partition will only fetch the *offsets* that have not been processed before failure or only process the messages from the offset that have not been processed by the previously-failing Consumer.

The ”*exactly once*” semantics is atomic, *i.e.*, unlike having two different storages for saving the processed data and *offsets*, as witnessed in the “*at least once”* and *“at most once*” semantics, we have a single storage (NoSQL, MySQL, *etc*.) in the “*exactly once*” semantics. The processed messages as well as the committed *offsets* are stored in a single storage. This ensures that if at all, the offset commit fails, the processing of messages also fails and *vice versa*.

Let us understand this better with the following example:

- Consider a *sensor* Topic from which a Consumer is reading the data.
- The data that is being read from Kafka is being processed and inserted in an external data storage, such as the MySQL database table.

The following Figure 7(c) shows the process of “*exactly once”* semantics.



7(c) Exactly once Semantics

- Moreover, instead of committing the latest *offset* to Kafka, we will be updating another table with the current offset that is processed.

- The UPDATE and INSERT statements are always within a single transaction. This ensures that either both the statements are completed or both will fail. In this way, we can achieve the "*exactly once"* semantics.

In the lab exercises, we shall be implementing the "*exactly once*" semantics in the lab exercise.

SERVICE GOALS

There are a few service goals to be aware of, before developing a Kafka application. These goals are described as follows:

i. Throughput
ii. Latency
iii. Durability
iv. High Availability

It is important to realize that we cannot achieve all these service goals in a single Kafka application. Therefore, there should always be a compromise between one or more service goals. The decision regarding which service goals we have to keep and which we have to trade off; and it purely depends on the business requirement(s).

Let us now understand these service goals in detail.

Throughput

Throughput is the speed at which the data is transmitted between various Kafka components, *i.e.,* between Producers to Brokers and Brokers to Producers. More formally, the total number of messages transmitted per second between the Kafka components is called *throughput*. Here are some of the features of *throughput*:

- Throughput can be achieved with higher number of partitions: the larger the number of partitions, higher the throughput. However, it does not mean that we should create Topics with high number of Partitions. Let us now see how to decide the number of Partitions for a specific Topic.

As a starting point, consider that the size of a message is approximately 2 KB. Each partition should be able to handle 10,000 messages per second. This means that one partition should handle 20 MB (10,000 messages * 2KB) of messages per second. If the application requires 1 GB throughput, *i.e.*, 1 GB data per second, we should divide 1 GB with 20 MB, resulting in 51.2. This implies that we need to create a Topic with 50 – 75 partitions to achieve the required

throughput of 1 GB. Afterward, we can keep experimenting with other message sizes to even further optimize the throughput.

- The *batch.size* and *linger.ms* configuration properties can also affect the throughput. Thus, increasing the batch sizes and lingering the time period can increase the throughput. We have already learned about these configuration properties in the previous chapters.

- We can also specify the compression codec type by using the *compression.type* property. There is no compression by default, but a batch of data will be compressed and sent in case if the compression codec is specified. Basically, compression of data will generate more space for even more messages to accommodate in the buffer of a batch.

- Setting *acks=0* or *acks=1* and *retries=0* configuration values can also increase the throughput. Please check *Producer Configurations* section in Chapter 5 for more information.

- Setting a higher value for *buffer.memory* when there are more partitions can also increase the throughput. The *buffer.memory* should not be confused with *batch.size*. The *batch.size* determines the size at which the messages are flushed, whereas *buffer.memory* is the size of memory allocated to buffer the messages. The *batch.size* is usually lower than the *buffer.memory*.

While we are trying to increase the throughput, we are also trading off the latency. It means that when the throughput increases, the latency also increases. Confusing? Jump to the next section and there will be no confusion anymore.

Latency

Latency is the total time taken for a single message to transmit from the Producer to the Consumer. By default in Kafka, the system configuration is set to ensure low latency. Therefore, there are not many changes required to configure for latency. Although it is highly subjective, but it is considered

as the low latency, if the single message reaches in milliseconds, while it is high latency is the message takes more than a second to complete its journey. Here it might seem that latency and throughput are similar, but throughput is the time taken for a number of messages per second while latency is the time taken by a single message.

- More partitions will increase latency. It is because, having more partitions takes more time to replicate the partitions, thus increasing latency.

- The settings for other configuration properties, such as *linger.ms*, *compression.type and acks etc*. are set for the low latency by default. Hence, you need not worry about these configuration settings if your application requires low latency.

Having low latency setup automatically reduces the throughput. Therefore, if you want to achieve low latency, you cannot achieve high throughput.

Durability

Durability can be achieved when we reduce the chances of a message being lost during the transit. We can achieve durability by utilizing the replication factor feature in Kafka. Setting the replication factor to 3 ensures that data is always available in two other Brokers, when a Broker goes down due to a hardware or network failure.

- As stated earlier, we can achieve durability by setting a replication factor of 3.

- Setting *acks=all* and *retries=1 or more* will help you achieve high durability, as there is no data lost during the transit.

- We should also disable auto Topic creation by setting *auto.create.topics.enable=false* so that we are always in control of the replication factor and partition settings for every Topic.

- Setting a value of 2 for *min.insync.replicas* will ensure that, there are at least two follower replicas that are in sync with the leader replicas. This ensures that the follower replicas are up-to-date with the leader replica.

- Durability can be increased by setting the property *unclean.leader.election.enable* to *false*. A leader can still be elected even if the follower replicas are not in sync with the leader. Electing a follower that is not in sync with the leader will lead to the data loss because, the leader may have already committed messages that have not been replicated by the followers.

- Disabling the *auto commit* feature of Consumers also enhances the durability. For this purpose, the property *auto.commit.enable* should be set to *false*. We have already learned about the committing of auto and manual *offset* in the previous chapter.

- Durability is the strongest when your Kafka cluster is rack-aware. There can be durability guarantee when an entire rack fails as well. Therefore, when we specify the rack associated with a particular Broker, we can still have data available even when there is a rack failure. We can simply specify the rack a broker belongs to by setting the configuration parameter *broker.rack*, and then Kafka will automatically ensure that the replicas span as many racks as they can.

Having high durability will tradeoff for high availability, which is explained as follows:

High Availability

High availability is achieved when we configure Kafka to have the lowest possible downtime in case of unexpected failures. High availability and Durability might seem similar but they should not be confused with each other. Durability is to prevent the data loss, whereas high availability is related to the insurance to have back-up as quickly as possible during a failure.

- We have high availability when the property *unclean.leader.election.enable* is set to *true*. This will ensure that the leader election happens quickly and there is no downtime. However, this is a tradeoff for durability.

- Setting *min.insync.replicas* property to 1 ensures that the Producer can keep sending data until there is one follower replica in sync with the leader. This increases the availability for the partition.

As mentioned earlier, we cannot achieve all these service goals. We must compromise on one or the other goal(s) based on the business requirement.

The theory related to this chapter ends here. Let us now move to the lab exercises and implement "*exactly once"* semantics.

AIM

The aim of the following lab exercises is to implement the "*Exactly once*" semantics. To implement the semantics, we need an external storage to insert the processed data as well as update the current *offset*. We shall be utilizing MySQL as an external storage for this lab exercise.

The labs for this chapter include the following exercises:

- Download & Install MySQL
- Create Database & Tables
- Constructing a Producer
- Constructing a Consumer
- Running Producer & Consumer

We need the following packages to perform the lab exercise:

- Java Development Kit (JDK)
- Apache ZooKeeper
- Apache Kafka
- Scala
- IntelliJ IDEA
- MySQL Server

LAB EXERCISE 7: KAFKA DATA DELIVERY

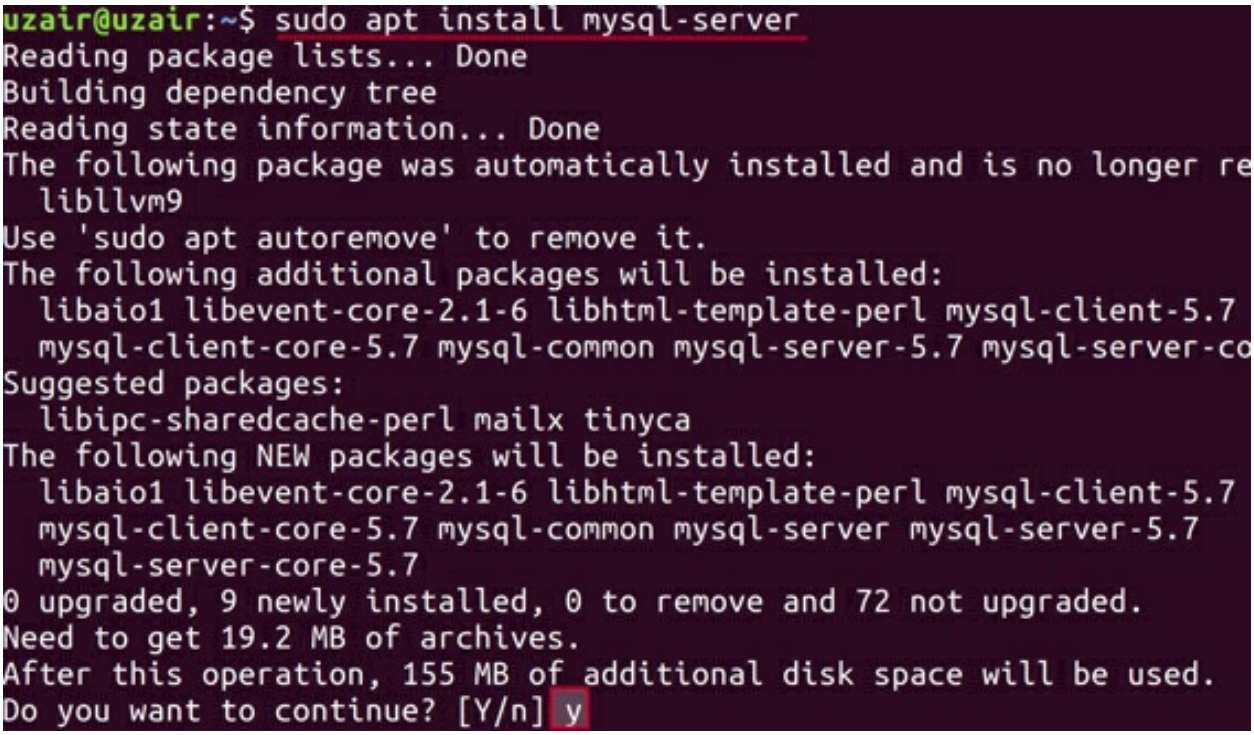
1. **Download & Install MySQL**
2. **Create Database and Tables**
3. **Constructing a Producer**
4. **Constructing a Consumer**
5. **Running Producer & Consumer**

TASK 1: DOWNLOAD & INSTALL MYSQL

Step 1: To download the MySQL server to your machines, open the terminal and execute the following commands:

```
$ sudo apt update
$ sudo apt install mysql-server
```

You will have to accept the prompts by pressing 'Y' on your keyboard as shown in the screenshot below:

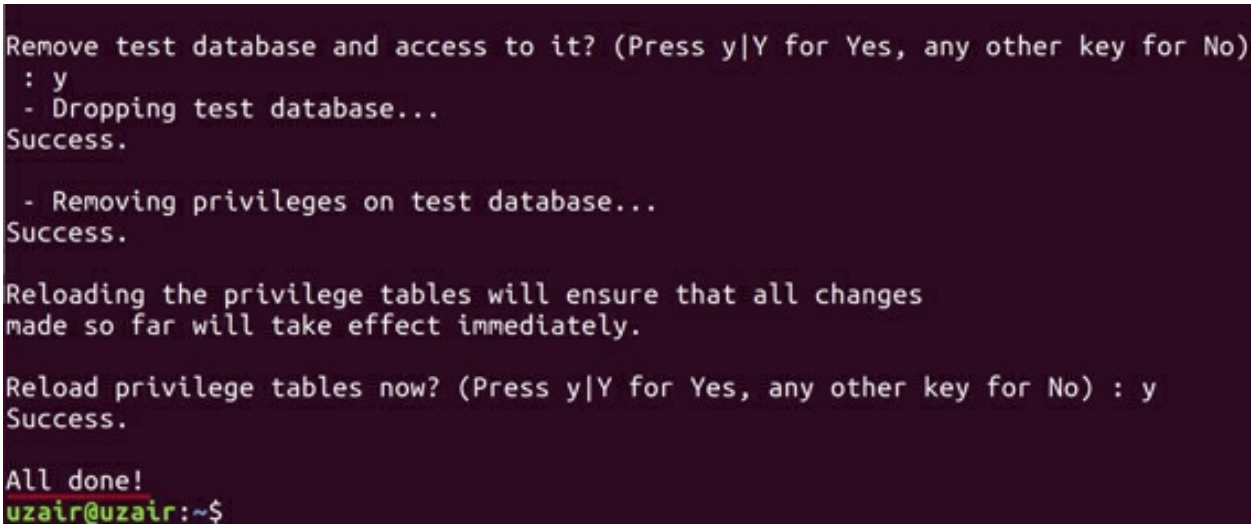


Step 2: Once the installation is complete, we should set a password for the *root* user. We can do this by executing the following command:

```
$ sudo mysql_secure_installation
```

Subsequently, you will be prompted to install the '*validate password*' plugin. But, you may or may not choose to install. In this case, we did not choose to install it, and therefore, we continued by pressing any key. Next, you will be asked to set a password and confirm it. Please set a password of your choice. In a production environment, please make sure that you set a secure password.

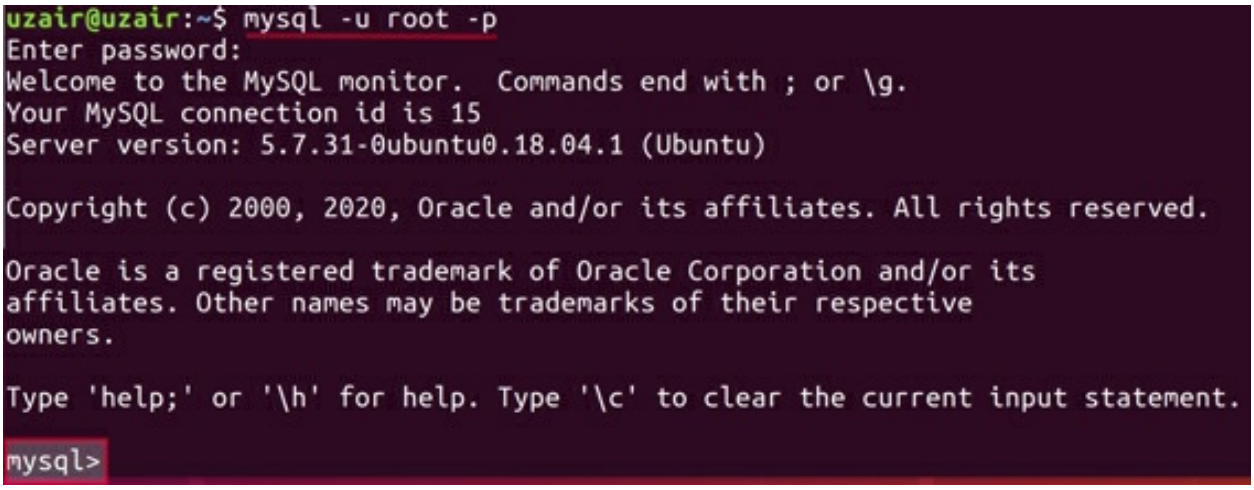
Once you set a password, keep pressing ‘Y’ for all the prompts. You should see the *‘All done!’* message as depicted in the screenshot below:



Step 3: Now that we have installed and set the *root* password for our MySQL server, let us access the MySQL shell as *root* user.

```
$ mysql –u root -p
```

After executing the command above, enter the password. You should see the *mysql* prompt as shown in the screenshot below.



If you get an *‘access denied’* error, enter the following commands. Make sure to replace the *password* with your desired password. Repeat the following step:

```
$ sudo mysql
mysql> ALTER USER 'root'@'localhost' IDENTIFIED WITH
```

```
mysql_native_password BY 'password';
mysql> exit;
```

Task 1 is complete!

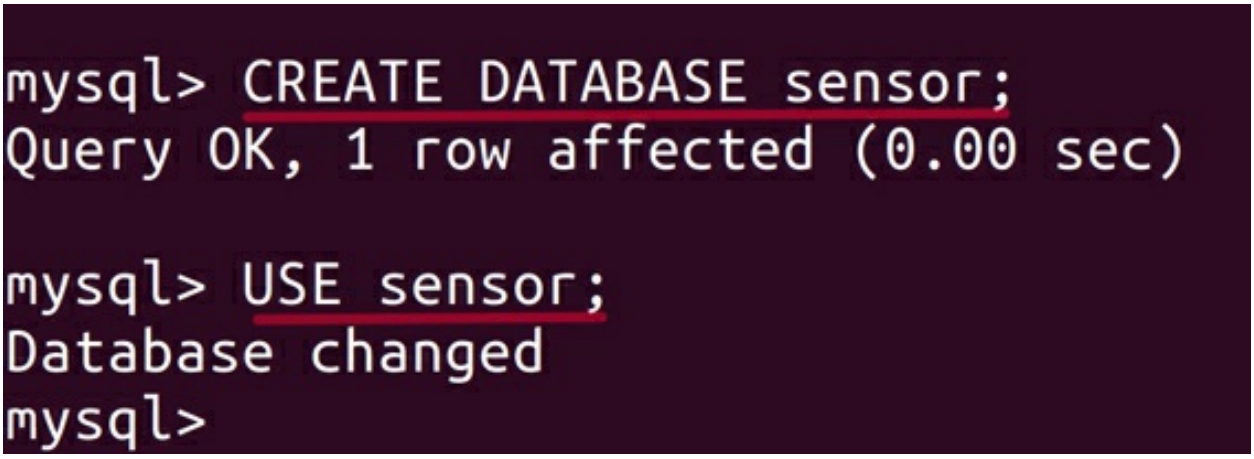
TASK 2: CREATE DATABASE & TABLES

Now that we have MySQL serve ready, let us first create a database and two tables for inserting the messages and updating the current *offset*.

Step 1: Let us first create a database as follows:

```
mysql> CREATE DATABASE sensor;
mysql> USE sensor;
```

You should see the confirmation as shown in the screenshot below:



Step 2: Let us now create a new table, where we can insert the sensor data received from Kafka.

```
mysql> CREATE TABLE sensor_data(skey VARCHAR(50), svalue VARCHAR(50));
```

Step 3: Let us also create a new table to update the current *offset* with Topic name, partition, and the current *offset* columns.

```
mysql> CREATE TABLE sensor_offsets(topic_name VARCHAR(50), partitions INT, current_offset INT);
```

You should now have two tables created. You can now check if these tables have been properly created or not, by executing the following command:

```
mysql> SHOW TABLES;
```

```
mysql> CREATE TABLE sensor_offsets(topic_name VARCHA
nt_offser INT);
Query OK, 0 rows affected (0.06 sec)

mysql> SHOW TABLES;
+------------------+
| Tables_in_sensor |
+------------------+
| sensor_data      |
| sensor_offsets   |
+------------------+
2 rows in set (0.01 sec)
```

Step 4: Now that we have the required tables created and confirmed, let us initialize the *offset* values as 0 in the *sensor_offsets* table, so that the consumer updates the current *offset* every time it reads the messages. But, we need not touch the *sensor_data* table, as the Consumer automatically inserts the data into it as soon as it receives it.

The Topic name is *sensor* with 3 partitions, which we shall create later in the lab exercise. Now, we shall simply insert three rows for each partition as shown below.

```
mysql> INSERT INTO sensor_offsets values('sensor', 0, 0); mysql> INSERT INTO sensor_offsets values('sensor', 1, 0); mysql> INSERT INTO sensor_offsets values('sensor', 2, 0);
```

You may check if the *insert* was successful using the following query.

```
mysql> SELECT * FROM sensor_offsets;
```

As you can see in the screenshot below, we have initialized the current *offset* as 0 for each partition. Once we execute our Consumer application, the current *offset* value automatically starts to update and hence Consumer will know from which *offset* it should start polling.

```
mysql> INSERT INTO sensor_offsets values('sensor', 0, 0);
Query OK, 1 row affected (0.01 sec)

mysql> INSERT INTO sensor_offsets values('sensor', 1, 0);
Query OK, 1 row affected (0.02 sec)

mysql> INSERT INTO sensor_offsets values('sensor', 2, 0);
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM sensor_offsets;
+------------+------------+----------------+
| topic_name | partitions | current_offser |
+------------+------------+----------------+
| sensor     |          0 |              0 |
| sensor     |          1 |              0 |
| sensor     |          2 |              0 |
+------------+------------+----------------+
3 rows in set (0.00 sec)

mysql> SELECT * FROM sensor_data;
Empty set (0.00 sec)

mysql>
```

As we can see from the screenshot above, the *sensor_data* table is empty for now. But it will be updated once the consumer starts fetching data from the brokers.

Task 2 is complete!

TASK 3: CONSTRUCTING A PRODUCER

In the previous two tasks, we have installed MySQL server, and created the required database and tables. Now, let us start constructing a Producer that will generate the sensor data.

Step 1: Create a new *Scala* object and name it *SProducer*. Make sure you extend the object to *App* trait to make it an executable program. Next insert the following required imports. These imports are required to specify properties and create *Producer* & *ProducerRecord* objects. Moreover, include the variables for Topic name and bootstrap servers as shown below.

```
import java.util.properties

import org.apache.kafka.client.producer.KafkaProducer
import org.apache.kafka.client.producer.ProducerRecord

object SProducer extends App {

      val topic = args(0)
      val brokers = args(1)}
```

Step 2: The next step is to specify the properties.

```
val props = new Properties()
props.put(“bootstrap.servers”, brokers)
props.put(“client.id”, “Kafka Sensor Producer”)
props.put(“key.serializer”,
“org.apache.kafka.common.serialization.StringSerializer”)
props.put(“value.serializer”,
“org.apache.kafka.common.serialization.StringSerializer”)
```

Step 3: Let us now create a Producer object. We shall instantiate the Producer object by passing *props* as the argument, and specify the String types for key and value.

```
val producer = new KafkaProducer[String, String] (props)
```

The above line of code instantiates a Producer object that is of type String and String. The String type specifies the type for *key* and *value*.

Step 4: Finally, we can now generate the sensor data by using the *ProducerRecord* object. We use the '*for*' loop to generate any random data.

```
for (i <- 0.until(10))

producer.send(new ProducerRecord(topic, "RFID", "100" + i))
```

The *ProducerRecord* object constructor is instantiated by passing arguments as name of the Topic, *key,* and *value*. The *ProducerRecord* object contains the actual message that is being transmitted to the Brokers. Subsequently, we simply invoke the *send* method on our producer object by passing the *ProducerRecord* as an argument. The Producer will then start sending the messages to the Brokers.

Finally, after sending the messages, we must close the Producer object using the *close message* as shown below. Closing the Producer object basically frees up the resources being utilized by the Producer.

```
producer.close()
```

```
import java.util.Properties

import org.apache.kafka.clients.producer.{KafkaProducer, ProducerRecord}

object SProducer extends App {

  val topic = args(0)
  val brokers = args (1)

  val props = new Properties()
  props.put("bootstrap.servers", brokers)
  props.put("client.id", "Kafka Sensor Producer")
  props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer")
  props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer")

  val producer = new KafkaProducer[String, String] (props)

  for (i <-0.until(10))
    producer.send(new ProducerRecord(topic, key="RFID", value="100" + i))

  producer.close()
}
```

Step 5: Before we execute our Producer, let us first create a Topic called *sensor-data*. Start the ZooKeeper and Kafka servers.

```
uzair@uzair:~$ zkServer.sh start                                                ns
/usr/bin/java
ZooKeeper JMX enabled by default
Using config: /usr/share/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
uzair@uzair:~$ kafka-server-start.sh /usr/share/kafka/config/server.properties
[2020-10-26 21:08:47,867] INFO Registered kafka:type=kafka.Log4jController MBean
 (kafka.utils.Log4jControllerRegistration$)
[2020-10-26 21:08:49,941] INFO Setting -D jdk.tls.rejectClientInitiatedRenegotia
tion=true to disable client-initiated TLS renegotiation (org.apache.zookeeper.co
mmon.X509Util)
[2020-10-26 21:08:50,142] INFO Registered signal handlers for TERM, INT, HUP (or
g.apache.kafka.common.utils.LoggingSignalHandler)
[2020-10-26 21:08:50,162] INFO starting (kafka.server.KafkaServer)
[2020-10-26 21:08:50,165] INFO Connecting to zookeeper on localhost:2181 (kafka.
server.KafkaServer)
```

Once the servers are up and running, create a new Topic with the name *sensor-data*.

```
uzair@uzair:~$ kafka-topics.sh --zookeeper localhost:2181 --create --topic senso
r-data --replication-factor 1 --partitions 3
Created topic sensor-data.
uzair@uzair:~$
```

Step 6: Let us now run the Producer. Please check Lab Exercise 5 (Task 3) on how to run the Producer. After a while, you should see that the Producer has run successfully with the *exit code* 0.

Step 7: Now let us check the *kafka-logs* directory. The default location of this directory is */tmp/kafka-logs*. You should see three partitions for the *sensor-data* Topic. Go through each partition, and check the *.log* file for the message that has been received.

Task 3 is complete!

TASK 4: CONSTRUCTING A CONSUMER

Let us finally construct a Consumer that implements the '*exactly once*' semantics.

Step 1: Create a new *Scala* object and name it *SConsumer*. Next, insert the following required imports.

```
import java.util._
import java.sql._
import java.time.Duration

import scala.collections.JavaConversions._

import org.apache.kafka.clients.consumer._
import org.apache.kafka.common._

object SensorConsumer extends App{
```

Step 2: Let us now declare and initialize the required variables.

```
  val topics: String = args(0)
  val brokers: String = args(1)

  var consumer: KafkaConsumer[String, String] = null
  var rCount: Int = 0
```

Here we are simply creating a Kafka Consumer object as a mutable variable and assigning its value to *null*. Moreover, we also declare a mutable variable

to keep track of the record count and initialize its value as 0.

Step 3: The next step is to create a *properties* object and specify the required properties for this Consumer application.

```
val props: Properties = new Properties()

    props.put("bootstrap.servers", brokers)
    props.put("key.deserializer","org.apache.kafka.commo
    props.put("value.deserializer","org.apache.kafka.cor
    props.put("enable.auto.commit", "false")
```

We have learned about all these properties, during the process of creating our first Consumer in the previous chapter.

Next, we pass all these properties as an argument to our Consumer object.

```
consumer = new KafkaConsumer(props)
```

```
object SConsumer {

  def main(args: Array[String]): Unit = {

    val topics: String = args(0)
    val brokers: String = args(1)

    var consumer: KafkaConsumer[String, String] = null
    var rCount: Int = 0

    val props: Properties = new Properties()
    props.put("bootstrap.servers", brokers)
    props.put("key.deserializer",
      "org.apache.kafka.common.serialization.StringDeserializer")
    props.put("value.deserializer",
      "org.apache.kafka.common.serialization.StringDeserializer")
    props.put("enable.auto.commit", "false")

    consumer = new KafkaConsumer(props)
  }
}
```

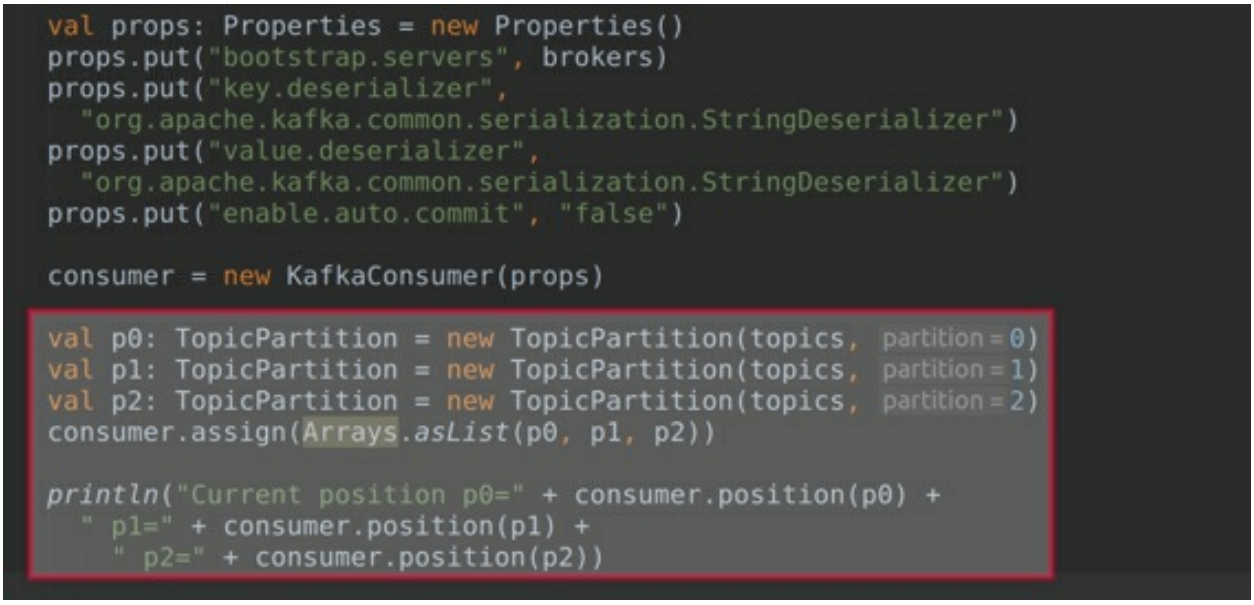
Step 4: Since we do not want Kafka to commit the *offsets* automatically, and assign the partitions (automatically) to the Consumers in a Consumer Group, we manually assign the partitions to the Consumer object we had created earlier. Please note that we have not specified the group *id* property and hence we must manually assign the partitions to the Consumer.

Now, we create three *TopicPartition* objects for three partitions of our Topic, and then assign these three partitions to the *Consumer* object.

```
val p0: TopicPartition = new TopicPartition(topics, 0)
val p1: TopicPartition = new TopicPartition(topics, 1)
val p2: TopicPartition = new TopicPartition(topics, 2)

consumer.assign(Arrays.asList(p0, p1, p2))
```

Let us also print the current position of the partitions to the console.

```
println("Current position p0=" + consumer.position(p0) +
"  p1="  +  consumer.position(p1)  +  "  p2="  +
consumer.position(p2))
```



We have now all three partitions assigned to our Consumer using the *assign* method. This method takes a list of partitions as arguments. The Consumer can now read data from these three partitions.

Step 5: The next step is to specify the *offset* position for each partition, so that the Consumer knows where to start reading from. In Task 2, we had set the current *offset* value as 0 for all the partitions. Now, we simply fetch the *offset* from the *sensor_offsets* table using *getOffsetFromDB* method for each partition. The *seek* method is utilized to specify the position of the current *offset*. It takes the Topic partition and the *offset* number as arguments.

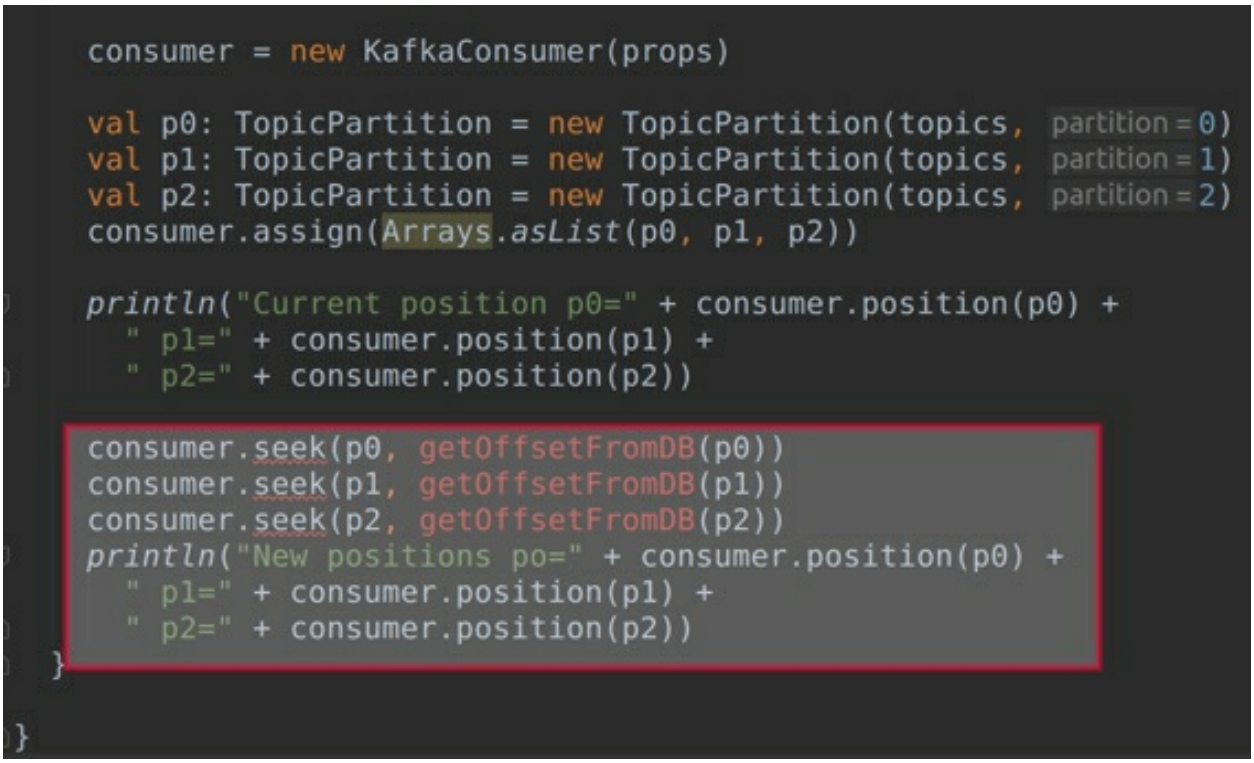
```
consumer.seek(p0, getOffsetFromDB(p0))
```

```
consumer.seek(p1, getOffsetFromDB(p1))
consumer.seek(p2, getOffsetFromDB(p2))
```

Let us also print the latest positions to the console.

```
println("New positions p0=" + consumer.position(p0) + "
p1="    +    consumer.position(p1)    +    "    p2="    +
consumer.position(p2))
```

Please note that you will see a few errors in case of using the *seek* and *getOffsetFromDB* methods. These errors will vanish, when we define the *getOffsetFromDB* method later in this task.



Step 6: Now that we have already assigned the partitions for the Topic, and specified the *offset* positions for the partitions, we can start reading the data from Kafka. Let us start fetching this data using the *poll* method within the '*do while'* loop with error handling.

When there are no more records to fetch, we finally close the Consumer as follows:

```
println("Starting to Fetch Records")
```

```
try{
do {
   val   records:   ConsumerRecords[String,   String]   =
consumer.poll(Duration.ofMillis(1000))

println("Record polled " + records.count())
rCount = records.count()

for (record <- records) {
    saveAndCommit(consumer, record)
    }
}

while (rCount > 0);

} catch {
 case e:Exception => e.printStackTrace()
}

finally{
   consumer.close()
  }
}
```

The *saveAndCommit* function inserts the records we fetch from Kafka and it also updates the current *offset* in the database. This is how we achieve the '*exactly once'* semantics.

Please ignore errors for the saveAndCommit method, as they will

automatically vanish once we implement the saveAndCommit method.

At this point, we have created a Consumer object, assigned the partitions and specified the *offset* position. Moreover, we have also implemented the *poll* method to fetch the records. Subsequently, all we have to do now is to implement the *getOffsetFromDB* and *saveAndCommit* methods.

Step 7: Let us implement the *getOffsetFromDB* method in this step. The *getOffsetFromDB* method takes the partition as an argument, and returns the *offset* which is of type *Long*. In addition, we also declare a mutable variable for the *offset* of type *Long* and initialize it as 0.

```
private def getOffsetFromDB(p: TopicPartition): Long = {
   var offset: Long = 0
```

Next, we connect to the MySQL server by using the JDBC driver by specifying the database and login credentials.

```
try {
Class.forName("com.mysql.cj.jdbc.Driver")
val con: Connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/
"root", "password") //Please enter the password of your
MySQL root user here.
```

Once we have a connection with MySQL, we can write a SQL query to get the current *offset*. The method finally returns an *offset*.

```
val sql: String = "select current_offser from
sensor_data where topic_name='" + p.topic() + "' and
partitions=" + p.partition()

val stmt: Statement = con.createStatement()
val rs: ResultSet = stmt.executeQuery(sql)
if (rs.next())
offset = rs.getInt("current_offser")
     stmt.close()
    con.close()
   } catch {
    case e: Exception => e.printStackTrace
```

```
  }
  offset
 }
```

```
private def getOffsetFromDB(p: TopicPartition): Long = {
  var offset: Long = 0
  try {
    Class.forName(className = "com.mysql.jdbc.Driver")
    val con: Connection = DriverManager.getConnection(url = "jdbc:mysql://localhost:3306/sensor",
      user = "root", password = "[redacted]")
    val sql: String = "SELECT current_offser from sensor_offsets where topic_name='" +
      p.topic() + "' and partition=" + p.partition()

    val stmt: Statement = con.createStatement()
    val rs: ResultSet = stmt.executeQuery(sql)
    if (rs.next())
      offset = rs.getInt("current_offser")
    stmt.close()
    con.close()
  } catch {
    case e: Exception => println("Exception in getOffsetFromDB")

  }
  offset
}
```

Step 8: Finally, let us implement the *saveAndCommit* method. The *saveAndCommit* method takes Kafka Consumer and Consumer record as the arguments.

```
private def saveAndCommit(c: KafkaConsumer[String, String], r: ConsumerRecord[String, String]): Unit = {

println("Topic=" + r.topic() + " Partition=" + r.partition() + " Offset=" + r.offset() + " Key=" + r.key() + " Value=" + r.value())
```

Next, we connect to MySQL, as we had done in the previous step and insert the records to the *sensor_data* table as well as update the *sensor_offsets* table with the current *offset* for each partition.

```
try {
Class.forName("com.mysql.cj.jdbc.Driver")
val con: Connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/ "root", "password")//Please enter the password of your MySQL root user here.
```

After connecting to MySQL, we set the auto commit to *false*.

```
con.setAutoCommit(false)
```

Then we simply run the SQL query to insert the record to a sensor_data table.

```
val insertSQL: String = "insert into sensor_data values(?,?)"

val psInsert: PreparedStatement = con.prepareStatement(insertSQL)
psInsert.setString(1, r.key())
psInsert.setString(2, r.value())
```

Now that the *insert* process is complete, we specify an SQL query to update the current *offset* in *sensor_offsets* table.

```
val updateSQL: String = "update sensor_offsets set current_offset=? where topic_name=? and partition=?"

val psUpdate: PreparedStatement = con.prepareStatement(updateSQL)
psUpdate.setLong(1, r.offset() + 1)
psUpdate.setString(2, r.topic())
psUpdate.setInt(3, r.partition())
```

We then execute the insert and update operations.

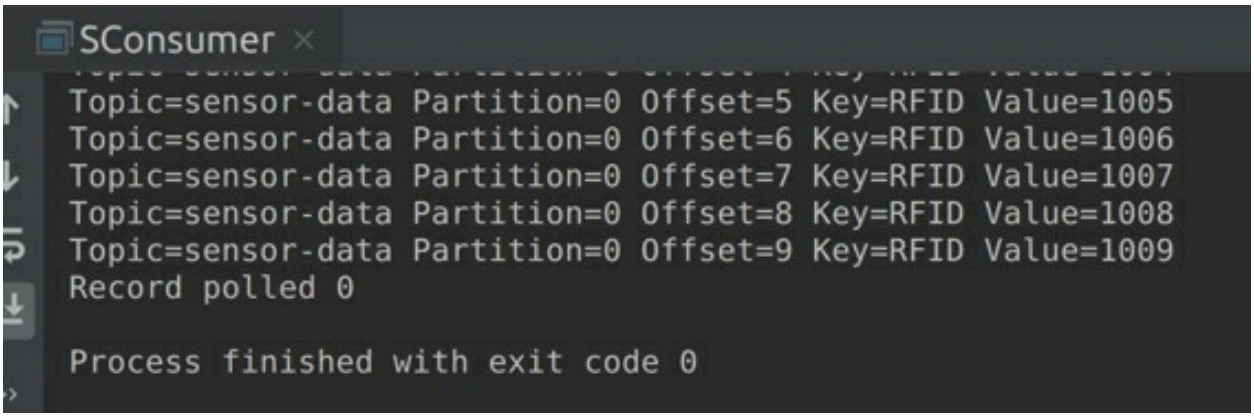
```
psInsert.executeUpdate()
psUpdate.executeUpdate()
```

We finally commit the *offset*. This is an atomic transaction. In this way, any failure results in both *insert* and *update* to fail. Using this process, we can achieve the '*exactly once*' semantics.

```
con.commit()
con.close()
   } catch {
case e: Exception => e.printStackTrace

   }
 }
}
```

```
private def getOffsetFromDB(p: TopicPartition): Long = {
  var offset: Long = 0
  try {
    Class.forName(className = "com.mysql.jdbc.Driver")
    val con: Connection = DriverManager.getConnection(url = "jdbc:mysql://localhost:3306/sensor",
      user = "root",  password = "[redacted]")
    val sql: String = "SELECT current_offser from sensor_offsets where topic_name='" +
      p.topic() + "' and partition=" + p.partition()

    val stmt: Statement = con.createStatement()
    val rs: ResultSet = stmt.executeQuery(sql)
    if (rs.next())
      offset = rs.getInt("current_offser")
    stmt.close()
    con.close()
  } catch {
    case e: Exception => println("Exception in getOffsetFromDB")

  }
  offset
}
```

Step 9: Let us finally run the Consumer. Please check the Lab Exercise 6 (Task 2) on how to run the Consumer. After a while, you can notice that the Producer has executed successfully with exit code 0.



Moreover, you can also execute a query in MySQL to check the records in both tables.

Task 4 is complete!

SUMMARY

The messages in Kafka can be delivered in the following three possible ways. They include:

- **At least once**: No messages are lost, but there may be duplicate messages.
- **At most once**: Messages may be lost, but there are no duplicates.
- **Exactly once**: Neither message loss occurs, nor duplicate messages are created. Each message is delivered *exactly once*.

Before developing a Kafka application, we should ascertain the following service goals:

i. Throughput
ii. Latency
iii. Durability
iv. High Availability

One important point to consider is that it is impossible to achieve all these service goals in a single Kafka application. Therefore, there is always some compromise among one or the other goals. Those service goals which we have to keep and/or trade off depend on the business requirement.

In the corresponding labs, we have implemented the '*Exactly Once*' Semantics.

CHAPTER 8:
KAFKA ADMINISTRATION

Theory

In this chapter, we will learn about the administration of Kafka. We have already covered how to configure Kafka using configuration properties in the Lab exercise 2. Furthermore, we have also worked on Kafka Topic Operations in Lab exercise 3. Let us now look at more administration concepts in Kafka.

BASIC KAFKA OPERATIONS

Let us start Kafka administration with some basic Kafka operations.

Topic Operations

Please check chapter 3, Topic Operations section.

Graceful Shutdown

A new Partition Leader is elected whenever a Broker hosting leader the copy of a partition goes down due to the hardware failure or manual shut down for maintenance. In an alternate scenario, Kafka offers a graceful solution to take the server down instead of hard killing it. The advantages of shutting down gracefully are given as follows:

- The Broker syncs the messages to a disk before it shuts down. This process saves the time taken for recovering the messages once the Broker restarts. The recovery of messages is a time-consuming process, and therefore, graceful shutdown helps to come back up quickly during the intentional restarts.

- The Broker transfers all the Partitions that it is currently leading to other replicas before the shutdown. This will ensure a quick new leader election and also minimize the time for which the partition is unavailable during this process.

The sync happens automatically whenever the Broker is stopped except during a hard kill. However, for the controlled leadership transfer, it requires the following property to be set to *true*.

```
controlled.shutdown.enable=true
```

The controlled shutdown is only useful when all the partitions of that broker have a replication factor of more than 1. If there is only one replica for a partition it will anyway be unavailable during the shutdown.

Rack Awareness

The placement of replicas plays a vital role when it is intended to achieve high durability. The replications in Kafka are saved to the separate Brokers. However, if the Kafka cluster spans multiple racks of a data center and if all the replications are saved in the Brokers belonging to the same rack, there is a risk of data loss in case of a rack failure.

With Kafka's rack awareness feature, the replications are spread across various racks. This ensures that even during a rack-level failure, the data is still available in the Brokers of other racks. To achieve rack awareness, we must ensure that the Brokers are spread across multiple racks in a Kafka cluster and utilize the following configuration property to specify the rack name for each Broker. The configuration property must be specified in *server.properties* file.

```
broker.rack = my-rack-name
```

When a rack name is specified for each Broker, Kafka will ensure to have replications across multiple racks providing a higher durability.

Scaling Kafka Cluster

We can scale the Kafka cluster by adding new Brokers as required. To add new Brokers to our Kafka clusters, we simply need to assign a unique Broker id and fire up Kafka on that Broker. To utilize these newly-commissioned

Kafka Brokers, we must either create a new Topic or move the Partitions to them. The Partitions would not be allotted automatically as soon as they are commissioned.

We shall learn more about migrating Partitions in the Handling Partitions section of this chapter.

KAFKA CONSUMER GROUPS TOOL

The Consumer Groups tool provides you all the required information related to the Consumers. We can fetch the status of all the Consumer Groups in the Kafka cluster using the Consumer Groups tool. The Consumer Groups tool can also be used to list, describe and delete the Consumers.

The Kafka Consumer Group tool is specified with the shell script *kafka-consumer-groups.sh*. The *kafka-consumer-groups.sh* tool needs to be specified with the *--bootstrap-server* parameter followed by the host and port pair of the Kafka Broker.

For example, we can list Consumer Groups in Kafka using the following command:

```
$ kafka-consumer-groups.sh \
--bootstrap-server localhost:9092 \
--list
```

We can also fetch more details at group level by using the following command. This command returns all the Topics that are being consumed by the group along with the *offsets* for each partition.

```
$ kafka-consumer-groups.sh \
--bootstrap-server localhost:9092 \
--describe \
--group myGroup
```

We shall be looking at all the possible functions of Kafka Consumer Groups tool in the lab exercises.

Reset Offsets

It is also possible to reset *offsets* of the Consumer Group using the *--reset-offsets* option. Resetting the *offsets* can be useful in various failure scenarios

that require messages to be read again. This *--reset-offsets* option supports one consumer group at the time followed by *--all-topics* or *–topic* parameters. One of the parameters must be selected, except when *--from-file* parameter is used.

It has the following three execution options:

- *--dry-run* : to display which *offsets* to reset.
- *--execute* : to execute --reset-*offsets* process.
- *--export* : to export the results to the CSV format.

Moreover*, --reset-offsets* also has the following scenarios to choose from (at least one scenario must be selected):

- *--to-datetime <String: datetime>* : Reset offsets to offsets from datetime. Format: 'YYYY-MM-DDTHH:mm:SS.sss'.
- *--to-earliest* : Reset offsets to the earliest offset.
- *--to-latest* : Reset offsets to the latest offset.
- *--shift-by <Long: number-of-offsets>* : Reset offsets shifting the current offset by 'n', where 'n' can be positive or negative.
- *--from-file* : Reset *offsets* to values defined in the CSV file.
- *--to-current* : Resets *offsets* to the current offset.
- *--by-duration <String: duration>*: Reset offsets to *offset* by duration from current timestamp. Format: 'PnDTnHnMnS'
- *--to-offset* : Reset *offsets* to a specific *offset*

For example, the following command resets the *offsets* of a consumer group to the earliest *offset*.

```
$ kafka-consumer-groups.sh \
--bootstrap-server localhost:9092 \
--reset-offsets --group myGroup \
--topic logs \
--to-earliest
```

DYNAMIC CONFIGURATIONS

Configurations can be overridden dynamically on the fly using the *kafka-configs.sh* shell script tool. The *kafka-configs.sh* tool can be utilized to modify the Broker, Topic level configurations and override quotas for the Producer and Consumer clients.

Broker Configs

We can execute the following command to override a number of network threads for the Broker 1 as 2 using the *kafka-configs.sh* tool.

```
$ kafka-configs.sh \
--bootstrap-server localhost:9092 \
--entity-type brokers \
--entity-name 1 \
--alter \
--add-config num.network.threads=2
```

Moreover, the configuration overrides can be fetched by using the *--describe* command as shown below.

```
$ kafka-configs.sh \
--bootstrap-server localhost:9092 \
--entity-type brokers \
--entity-name 1 \
--describe
```

We can also delete the configuration changes by using the --delete-config command followed by the configuration property as shown below:

```
$ kafka-configs.sh \
--bootstrap-server localhost:9092 \
--entity-type brokers \
--entity-name 1 \
--alter \
--delete-config num.network.threads
```

There could be scenarios where we might want to change or override the configurations for an entire cluster. This helps to have all the configurations consistent across the cluster. We can set the cluster-wide configurations as shown below:

```
$ kafka-configs.sh \
--bootstrap-server localhost:9092 \
--entity-type brokers \
--entity-default \
--alter \
--add-config num.network.threads=2
```

We can also list the cluster-wide configurations using the following command:

```
$ kafka-configs.sh \
--bootstrap-server localhost:9092 \
--entity-type brokers \
--entity-default \
--describe
```

Please note that you cannot configure the read-only configuration properties dynamically. Please check the link in references that specifies if a broker configuration is read-only or not.

Topic Configs

We can also override the Topic-level configurations, describe and delete as seen for the Broker configurations. The configurations for Topics can have server defaults as well as the optional per-Topic override. If there are no per-Topic configurations specified, the default configurations of the server are used.

The per-Topic configurations can be specified in two different ways. First, while creating the Topic using one or more *--config* options with *kafka-topics.sh* tool, and the second, by using the *kafka-configs.sh* tool. The *kafka-configs.sh* tool for Topic-level configurations can be used with *--entity-type* option with *value* as Topics and *--entity-name* option with value as the name of the Topic.

For example, properties can be dynamically configured while creating a Topic as shown below.

```
$ kafka-topics.sh \
--bootstrap-server localhost:9092 \
```

```
--create \
--topic my-topic \
--partitions 1 \
--replication-factor 1 \
--config max.message.bytes=64000 \
--config flush.messages=1
```

The above command will create a Topic and override the two configurations *max.message.bytes* and *flush.messages*.

As mentioned earlier, we can also override the configurations by dynamically using the *kafka-configs.sh* tool. For example, we can override the compression codec as snappy for the Topic by using the following command.

```
$ kafka-configs.sh \
--bootstrap-server localhost:9092 \
--entity-type topics \
--entity-name logs \
--alter \
--add-config compression.type = “snappy”
```

The overrides can be checked using the *--describe* option as shown below. This will list all the dynamic overrides for the Topics entity.

```
$ kafka-configs.sh \
--bootstrap-server localhost:9092 \
--entity-type topics \
--entity-name my-topic \
--describe
```

The deletion and listing of overrides is similar to what we have witnessed for the Broker configurations. We simply have to replace the entity type as Topics followed by the name of the Topic and delete the configuration.

For example, the following command is used to delete the overridden configuration.

```
$ kafka-configs.sh \
--bootstrap-server localhost:9092  \
--entity-type topics \
```

```
--entity-name logs \
--alter \
--delete-config compression.type
```

Setting Quotas

We can also override the default quotas for the Producer and Consumer clients using the *kafka-configs.sh* tool. The quotas can be set for a client id or at a user level. The configurations that can be overridden for the Producer and Consumer clients are given as follows:

- **producer_bytes_rate**: The number of messages in bytes produced by a single client *id* to a single Broker per second.
- **consumer_bytes_rate**: The number of messages in bytes consumed by a single client id to a single Broker per second.

Let us configure the custom quota for user 1 and client A as follows:

```
$ bin/kafka-configs.sh  \
--bootstrap-server localhost:9092 \
--alter \
--add-config                    'producer_byte_rate=1024,
consumer_byte_rate=2048' \
--entity-type users \
--entity-name user1 \
--entity-type clients \
--entity-name clientA
```

We can also configure the custom quota just for the user or a client id. The configured quotas can be listed by using the *--describe* command as observed in the previous sections.

Please check the link to the documentation in the references section to learn all the valid configurations that can be overridden by using the *kafka-configs.sh* tool.

HANDLING PARTITIONS

Let us now look at the handling partitions. Kafka provides two different tools for handling partitions. These tools help us in electing the preferred replica

leaders, reassigning the partitions to Brokers and also modifying the replication factor of these partitions. Let us look at them in detail.

Electing Preferred Replications

In the event of Broker failure, the leader partitions hosted in that Broker will be reassigned to the other Brokers. When a Broker comes back from the failure, it will only be assigned its previous follower replicas of the partitions, at the time when its failure had happened.
However, to avoid this situation, Kafka has a concept of preferred replicas. The list of replicas for a partition is called the *assigned replicas*. The first replica in this list is called, the *preferred replica*. When enabled, Kafka tries to restore the leadership of the Partitions to the Broker that is set as *preferred* for its leadership. For example, if the list of assigned replicas for a partition are 1, 3 and 5, the partition 1 is preferred as the leader for 3 or 5, as it is the first replica in the Broker's replica list.

This following configuration property must be set to *true* to enable this particular behavior:

```
auto.leader.rebalance.enable=true
```

If this configuration property is not set to *true*, we can manually restore the leadership by using the `kafka-preferred-replica-election.sh` tool.

```
$ kafka-preferred-replica-election.sh \
--bootstrap-server broker_host:port
```

Reassigning Partitions

The Partitions can be reassigned to other Brokers using the partition reassignment tool. Partitions are reassigned to balance a Kafka cluster after commissioning (or decommissioning) of Brokers. Partitions are also reassigned in the event of the Broker failures, so that the load is evenly balanced across the cluster. Having Partitions distributed across the cluster ensures an evenly-distributed data-load on the Brokers. However, the Partition reassignment tool cannot automatically evaluate the data distribution and reassign the partitions. The user has to manually evaluate and come up with a plan to move the Topics or its Partitions.

The Partition reassignment tool has the following three steps to reassign the partitions:

- **--generate:** The first step is to generate a plan for the reassignment of partitions. The tool is provided with a list of Topics and Brokers to generate a plan.
- **--execute:** The second step is the execution step where the actual reassignment is performed.
- **--verify:** Finally, the tool verifies the status of the reassigned partitions by using the generated list. The status can be '*successful'*, '*failed'* or '*in progress*'.

Moving Topics to new machines

Let us first look at migrating the entire Topics. This tool comes in handy when the new Brokers are added to the Kafka cluster. The partitions can be reassigned by moving entire Topics to the newly-added Brokers, instead of moving these partitions individually. In order to reassign partitions, all we have to do is to specify the list of Topics to move, along with the list of new Brokers. Afterwards, the tool does the job of reassigning the partitions evenly across the newly-added Brokers.

Let us understand this with the following example. Assume a Kafka cluster with 3 Brokers (with *ids* 0, 1, and 2). These three Brokers contain multiple Topics and are serving the client requests. However, due to huge production and consumption on two of the Topics (let's assume, *server-logs* and *sensor logs*), we have then decided to scale our Kafka cluster by adding two new Brokers (ids 3 and 4). We now have to move these Topics (*server-logs and sensor logs*) to the newly added Brokers (3 and 4). To migrate these Topics to the newly-added Brokers, we have to assign them as a list to the Partition reassignment tool. After the execution of partition reassignment tool, the Topics (*server-logs and sensor logs*) will be moved to the newly-added Brokers (3 and 4). Now, the Topics will only exist on the newly-added Brokers (3 and 4) and will be removed from those Brokers that were hosting these Topics previously.

The tool accepts the list of Topics in the *json* file. The first step is to list the Topics that should be moved in the *json* file as shown below:

```
$ cat topics-to-move.json

{"topics": [{"topic": "server-logs"},
        {"topic": "sensor-logs"}],
"version":1
}
```

Next, we have to utilize the partition reassignment tool to generate the execution plan by specifying the path to *json* file containing the list of Topics to move and to the newly-added Broker list.

```
$ kafka-reassign-partitions.sh \
--bootstrap-server localhost:9092 \
--topics-to-move-json-file topics-to-move.json \
--broker-list "3,4" \
--generate
```

The path to *json* file is specified as a parameter with the *--topics-to-move-json-file* option and the Broker *ids* are specified with *--broker-list* option. We then utilize the *--generate* option to generate the execution plan. This will generate the current partition replica assignment and the proposed partition reassignment configuration as shown below. At this point of time, the reassignment has not yet started. It only provides with the current assignment and proposed assignment, *i.e.*, the partitions of Topics (*server-logs and sensor logs*) are proposed to move to the newly-added Brokers (3 and 4) from the Brokers (0, 1 and 2).

```
Current partition replica assignment

{"version":1,
 "partitions":[
{"topic":"sensor-logs","partition":2,"replicas":[0,1]},
{"topic":"sensor-logs","partition":0,"replicas":[1,2]},
{"topic":"server-logs","partition":2,"replicas":[1,2]},
{"topic":"server-logs","partition":0,"replicas":[0,1]},
{"topic":"sensor-logs","partition":1,"replicas":[0,2]},
{"topic":"server-logs","partition":1,"replicas":[0,2]}]
```

```
 }
 Proposed partition reassignment configuration
 {"version":1,
 "partitions":[
{"topic":"sensor-logs","partition":2,"replicas":[3,4]},
{"topic":"sensor-logs","partition":0,"replicas":[3,4]},
{"topic":"server-logs","partition":2,"replicas":[3,4]},
{"topic":"server-logs","partition":0,"replicas":[3,4]},
{"topic":"sensor-logs","partition":1,"replicas":[3,4]},
{"topic":"server-logs","partition":1,"replicas":[3,4]}]
 }
```

The current assignment should be saved as a backup for he rollback purposes. The new assignment should be saved in a *json* file to feed the *--execute* parameter, which is the next step.

Now that we have successfully generated the set of partition moves, it is time to execute this process.

```
$ kafka-reassign-partitions.sh \
--bootstrap-server localhost:9092 \
--reassignment-json-file partition-reassignment.json \
--execute
```

The path to *json* file (*partition-reassignment.json*) containing the proposed reassignment configuration is specified as a parameter to *--reassignment-json-file* option. Subsequently, we use the *--execute* option to start the execution of the partition reassignment.

```
Current partition replica assignment
{"version":1,
 "partitions":[
{"topic":"sensor-logs","partition":2,"replicas":[0,1]},
{"topic":"sensor-logs","partition":0,"replicas":[1,2]},
{"topic":"server-logs","partition":2,"replicas":[1,2]},
{"topic":"server-logs","partition":0,"replicas":[0,1]},
{"topic":"sensor-logs","partition":1,"replicas":[0,2]},
{"topic":"server-logs","partition":1,"replicas":[0,2]}]
```

```
 }
 Save this to use as the --reassignment-json-file option
during rollback
 Successfully started reassignment of partitions
 {"version":1,
 "partitions":[
{"topic":"sensor-logs","partition":2,"replicas":[3,4]},
{"topic":"sensor-logs","partition":0,"replicas":[3,4]},
{"topic":"server-logs","partition":2,"replicas":[3,4]},
{"topic":"server-logs","partition":0,"replicas":[3,4]},
{"topic":"sensor-logs","partition":1,"replicas":[3,4]},
{"topic":"server-logs","partition":1,"replicas":[3,4]}]
 }
```

The final step is to verify the reassignment using *--verify* option. The *json* file (*partition-reassignment.json*) used in the *--execute* step has to be specified along with the *--reassignment-json-file* option in the verification step. This step provides the information of reassignments that are completed successfully, failed or are in progress.

```
$ kafka-reassign-partitions.sh \
--bootstrap-server localhost:909 \
--reassignment-json-file partition-reassignment.json \
--verify

Status of partition reassignment:
 Reassignment of partition [sensor-logs,0] completed
successfully
 Reassignment of partition [sensor-logs,1] is in
progress
 Reassignment of partition [sensor-logs,2] is in
progress
 Reassignment of partition [server-logs,0] completed
successfully
 Reassignment of partition [server-logs,1] completed
successfully
 Reassignment of partition [server-logs,2] completed
```

```
successfully
```

We have now successfully moved Topics to the new machines using the partition reassignment tool.

Custom Partition Reassignment

Instead of having the Partition reassignment tool to generate the partition reassignment plan, users can also create their own custom reassignment plan by selecting the partitions that are required to be moved to other Brokers. This way the user can skip the *--generate* step and directly start from the *--execute* step.

For example, let us move the partition 0 of Topic *sensor-logs* to brokers 3, 4 and partition 1 of Topic *server-logs* to Brokers 1, 3. The first step is to manually create the reassignment plan in a *json* file (*custom-reassignment.json*) as given below:

```
$ cat custom-reassignment.json

 {"version":1,
"partitions":[
{"topic":"sensor-logs","partition":0,"replicas":[3,4]},
{"topic":"server-logs","partition":1,"replicas":[1,3]}]
}
```

We then use the *--execute* option to trigger the partition reassignment as observed in the previous section.

```
$ kafka-reassign-partitions.sh \
--bootstrap-server localhost:9092 \
--reassignment-json-file custom-reassignment.json \
--execute

 Current partition replica assignment

 {"version":1,
 "partitions":[
{"topic":"sensor-logs","partition":0,"replicas":[1,2]},
{"topic":"server-logs","partition":1,"replicas":[0,2]}]
 }

 Save this to use as the --reassignment-json-file option
```

```
during rollback
 Successfully started reassignment of partitions

{"version":1,
 "partitions":[
{"topic":"sensor-logs","partition":0,"replicas":[3,4]},
{"topic":"server-logs","partition":1,"replicas":[1,3]}]
 }
```

Finally, we can use the *--verify* option to check the status of reassignment as we had performed in the previous section.

Increasing Replication factor

We can also increase the replication factor by using the partition reassignment tool. All we need to do is to specify the extra replicas (Broker ids) in the custom reassignment json file for the partition that needs to have the replication factor increased, and use the *–execute* option.

For example, let us increase the replication factor of partition 1 of Topic *server_logs* from 2 to 3. The partition 1 of the *server_logs* has only two replicas in Brokers 1 and 3. We shall now have this partition replicated in Broker 2, so that it will exist in three Brokers 1, 2, and 3.

```
> cat increase-replication-factor.json
 {"version":1,
 "partitions":[
{"topic":"server-logs","partition":1,"replicas":[1,2,3]}
]}
```

We can now use this *json* file with the *--execute* option and trigger the reassignment process.

```
$ kafka-reassign-partitions.sh \
--bootstrap-server localhost:9092 \
--reassignment-json-file           increase-replication-
factor.json \
--execute

Current partition replica assignment
```

```
 {"version":1,
 "partitions":[
{"topic":"server-logs","partition":1,"replicas":[0,2]}
]}
 Save this to use as the --reassignment-json-file option
during rollback
 Successfully started reassignment of partitions
 {"version":1,
 "partitions":[
{"topic":"server-logs","partition":1,"replicas":[1,2,3]}
]}
```

The theory for this chapter finishes at this point. Let's jump ahead to the lab exercises and have our hands on the administration of Kafka.

AIM

The aim of the following lab exercises is to learn about the administration process of Kafka

The labs for this chapter include the following exercises:

- Executing Graceful Shutdown
- Working with the Consumer Groups Tool
- Dynamically overriding Configurations

We need the following packages to perform the lab exercises:

- Java Development Kit (JDK)
- Apache ZooKeeper
- Apache Kafka
- Scala
- IntelliJ IDEA

LAB EXERCISE 8: KAFKA ADMINISTRATION

1. **Executing Graceful Shutdown**
2. **Working with Consumer Groups Tool**
3. **Dynamically overriding Configurations**

TASK 1: EXECUTING GRACEFUL SHUTDOWN

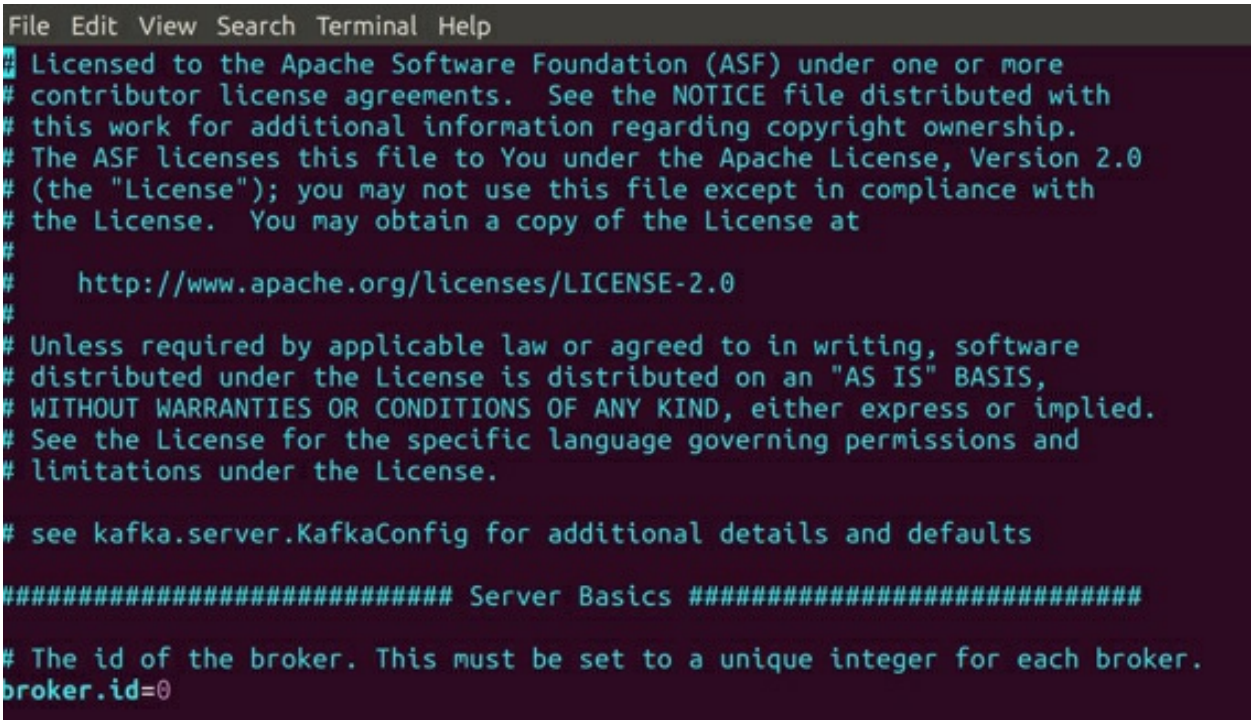
We have learned shutting down Kafka Brokers in a controlled environment in theory for this chapter. Let us now learn how we can practically execute it. Before starting this task, make sure the Kafka server is stopped.

Step 1: Open the *server.properties* configuration file. The configuration properties for Kafka are available in the following path:

```
/usr/share/kafka/config/server.properties
```

```
$ sudo vi /user/share/kafka/config/server.properties
```

You should observe the following screen:



Step 2: Enter the following configuration property and set it to *true*, so as to enable the controlled shutdown of the Kafka Broker. Save the *server.properties* file.

```
controlled.shutdown.enable=true
```

```
############################# Server Basics #############################

# The id of the broker. This must be set to a unique integer for each broker.
broker.id=0

controlled.shutdown.enable=true

############################# Socket Server Settings #####################
```

Step 3: Now start the ZooKeeper and then Kafka. You must start Kafka on all the nodes if you have multiple nodes in your Kafka cluster.

```
$ zkServer.sh start
```

You should observe that the ZooKeeper server starts as shown in the screenshot below.

```
uzair@uzair:~$ zkServer.sh start
/usr/bin/java
ZooKeeper JMX enabled by default
Using config: /usr/share/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
uzair@uzair:~$
```

```
$ kafka-server-start.sh /usr/share/kafka/config/server.properties
```

```
uzair@uzair:~$ kafka-server-start.sh /usr/share/kafka/config/server.properties
[2020-05-08 22:28:29,126] INFO Registered kafka:type=kafka.Log4jController MBean
 (kafka.utils.Log4jControllerRegistration$)
[2020-05-08 22:28:30,648] INFO Setting -D jdk.tls.rejectClientInitiatedRenegotia
tion=true to disable client-initiated TLS renegotiation (org.apache.zookeeper.co
mmon.X509Util)
[2020-05-08 22:28:30,877] INFO Registered signal handlers for TERM, INT, HUP (or
g.apache.kafka.common.utils.LoggingSignalHandler)
[2020-05-08 22:28:30,896] INFO starting (kafka.server.KafkaServer)
[2020-05-08 22:28:30,900] INFO Connecting to zookeeper on localhost:2181 (kafka.
server.KafkaServer)
[2020-05-08 22:28:31,075] INFO [ZooKeeperClient Kafka server] Initializing a new
 session to localhost:2181. (kafka.zookeeper.ZooKeeperClient)
[2020-05-08 22:28:31,132] INFO Client environment:zookeeper.version=3.5.7-f0fdd5
2973d373ffd9c86b81d99842dc2c7f660e, built on 02/10/2020 11:30 GMT (org.apache.zo
okeeper.ZooKeeper)
[2020-05-08 22:28:31,136] INFO Client environment:host.name=uzair (org.apache.zo
okeeper.ZooKeeper)
[2020-05-08 22:28:31,136] INFO Client environment:java.version=11.0.7 (org.apach
e.zookeeper.ZooKeeper)
[2020-05-08 22:28:31,136] INFO Client environment:java.vendor=Ubuntu (org.apache
.zookeeper.ZooKeeper)
[2020-05-08 22:28:31,137] INFO Client environment:java.home=/usr/lib/jvm/java-11
-openjdk-amd64 (org.apache.zookeeper.ZooKeeper)
```

Step 4: Now run the following command from the Broker you want to shut down from another terminal.

```
$ kafka-server-stop.sh
```

You should see the following information notifying that the Broker is being shut down in the controlled environment.

```
[2020-11-21 16:30:03,754] INFO Terminating process due to signal SIGTERM (org.ap
ache.kafka.common.utils.LoggingSignalHandler)
[2020-11-21 16:30:03,778] INFO [KafkaServer id=0] shutting down (kafka.server.Ka
fkaServer)
[2020-11-21 16:30:03,794] INFO [KafkaServer id=0] Starting controlled shutdown (
kafka.server.KafkaServer)
[2020-11-21 16:30:04,137] INFO [KafkaServer id=0] Controlled shutdown succeeded
(kafka.server.KafkaServer)
[2020-11-21 16:30:04,172] INFO [/config/changes-event-process-thread]: Shutting
down (kafka.common.ZkNodeChangeNotificationListener$ChangeEventProcessThread)
[2020-11-21 16:30:04,173] INFO [/config/changes-event-process-thread]: Stopped (
kafka.common.ZkNodeChangeNotificationListener$ChangeEventProcessThread)
```

As mentioned in the theory of this chapter, the controlled shutdown is only useful if all the partitions of that Broker have a replication factor of more than 1. If there is only one replica for a partition, the partition will be unavailable during the shutdown anyway.

Task 1 is complete!

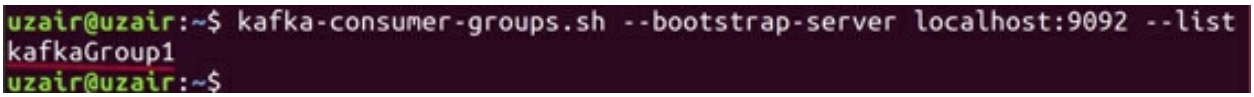
TASK 2: WORKING WITH CONSUMER GROUPS TOOL

Step 1: Before we start working with the Kafka Consumer Groups tool, we must ensure that there are some Consumers which are consuming from the Topic. Please run *kafkaProducer.scala* and *kafkaConsumer.scala* before going to the next step. The tool is not useful if there are no consumer groups running at all.

Step 2: Let us first list all the Consumer Groups using the Consumer Groups tool.

```
$ kafka-consumer-groups.sh \
--bootstrap-server localhost:9092 \
--list
```

You should see the *kafkaGroup1* as the Consumer Group as depicted in the screenshot below.



Step 3: We can also fetch more details at group level by using the following command. This command returns all the Topics that are being consumed by the group along with the *offsets* for each partition.

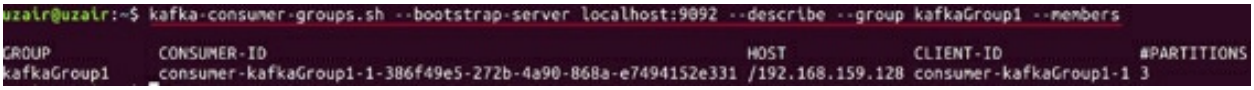
```
$ kafka-consumer-groups.sh \
--bootstrap-server localhost:9092 \
--describe \
--group kafkaGroup1
```

```
uzair@uzair:~$ kafka-consumer-groups.sh --bootstrap-server localhost:9092 --desc
ribe --group kafkaGroup1

GROUP           TOPIC           PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG
          CONSUMER-ID                                                HOST
        CLIENT-ID
kafkaGroup1     logs            0          0               0               0
          consumer-kafkaGroup1-1-27c6eb16-9e4a-43af-b293-7ab7a8ca2b25 /192.168.
159.128 consumer-kafkaGroup1-1
kafkaGroup1     logs            1          0               0               0
          consumer-kafkaGroup1-1-27c6eb16-9e4a-43af-b293-7ab7a8ca2b25 /192.168.
159.128 consumer-kafkaGroup1-1
kafkaGroup1     logs            2          1               1               0
          consumer-kafkaGroup1-1-27c6eb16-9e4a-43af-b293-7ab7a8ca2b25 /192.168.
159.128 consumer-kafkaGroup1-1
uzair@uzair:~$
```

Step 4: The Consumer Group tool can also be used to list all active members in a consumer group.

```
$ kafka-consumer-groups.sh \
--bootstrap-server localhost:9092 \
–describe \
--group kafkaGroup1 \
--members
```

Since we have one Consumer which is consuming from this consumer group, we can only see one Consumer as a member of this group as depicted in the screenshot below.



Apart from the information provided by the *--members* option, we can also use *--members --verbose* option to list the partitions assigned to each member.

```
$ kafka-consumer-groups.sh \
--bootstrap-server localhost:9092 \
–describe \
--group kafkaGroup1 \
--members \
--verbose
```

```
uzair@uzair:~$ kafka-consumer-groups.sh --bootstrap-server localhost:9092 --desc
ribe --group kafkaGroup1 --members --verbose

GROUP           CONSUMER-ID                                               HOST
            CLIENT-ID              #PARTITIONS     ASSIGNMENT
kafkaGroup1     consumer-kafkaGroup1-1-386f49e5-272b-4a90-868a-e7494152e331 /192
.168.159.128 consumer-kafkaGroup1-1 3               logs(0,1,2)
```

Step 5: Let us now use the *--state* option to list the useful group-level information.

```
$ kafka-consumer-groups.sh \
--bootstrap-server localhost:9092 \
--describe \
--group kafkaGroup1 \
--state
```

```
uzair@uzair:~$ kafka-consumer-groups.sh --bootstrap-server localhost:9092 --describe --group kafkaGroup1 --state

GROUP                COORDINATOR (ID)          ASSIGNMENT-STRATEGY  STATE          #MEMBERS
kafkaGroup1          uzair:9092 (0)            range                Stable         1
```

Step 6: Let us use the Consumer Groups tool to reset the *offsets* to the latest offset.

```
$ kafka-consumer-groups.sh \
--bootstrap-server localhost:9092 \
--reset-offsets \
--group kafkaGroup1 \
--topic logs \
--to-latest \
–execute
```

```
uzair@uzair:~$ kafka-consumer-groups.sh --bootstrap-server localhost:9092 --reset-o
ffsets --group kafkaGroup1 --topic logs --to-latest --execute

GROUP                          TOPIC                          PARTITION  NEW-OFFSET

kafkaGroup1                    logs                           0          0

kafkaGroup1                    logs                           1          0

kafkaGroup1                    logs                           2          2

uzair@uzair:~$
```

You may also run the command without *--execute* option to simply display the offsets to reset.

```
$ kafka-consumer-groups.sh \
--bootstrap-server localhost:9092 \
--reset-offsets \
--group kafkaGroup1 \
--topic logs \
--to-latest \
--dry-run
```

```
uzair@uzair:~$ kafka-consumer-groups.sh --bootstrap-server localhost:9092 --reset-o
ffsets --group kafkaGroup1 --topic logs --to-latest --dry-run

GROUP                          TOPIC                          PARTITION  NEW-OFFSET
kafkaGroup1                    logs                           0          0
kafkaGroup1                    logs                           1          0
kafkaGroup1                    logs                           2          2
```

Please check the '*Reset Offset*' section in the theory of this chapter for all the possible ways to reset the *offsets*.

Step 7: The Consumer Groups tool can also be used to manually delete a single or multiple consumer groups.

```
$ kafka-consumer-groups.sh \
--bootstrap-server localhost:9092 \
--delete \
--group kafkaGroup1
```

```
uzair@uzair:~$ kafka-consumer-groups.sh --bootstrap-server localhost:9092 --delete
--group kafkaGroup1
Deletion of requested consumer groups ('kafkaGroup1') was successful.
uzair@uzair:~$
```

We can then check if the consumer group was deleted successfully by running the list command as shown in the screenshot below. Since there are no more consumer groups, nothing has returned when we execute the *list* command.

```
uzair@uzair:~$ kafka-consumer-groups.sh --bootstrap-server localhost:9092 --delete
--group kafkaGroup1
Deletion of requested consumer groups ('kafkaGroup1') was successful.
uzair@uzair:~$ kafka-consumer-groups.sh --bootstrap-server localhost:9092 --list
uzair@uzair:~$
```

We have just deleted a single consumer group. You can also delete the

multiple consumer groups as shown below.

```
$ kafka-consumer-groups.sh \
--bootstrap-server localhost:9092 \
--delete \
--group kafkaGroup1 \
--group kafkaGroup2 \
--group kafkaGroup3
```

Task 2 is complete!

TASK 3: DYNAMICALLY OVERRIDING ONFIGURATIONS

We have looked at the dynamic configurations in the theory section of this chapter. Let us now practically apply a couple of configuration properties dynamically.

Step 1: Let us first dynamically configure a Broker configuration property using the *kafka-configs.sh* tool. We shall be enabling an unclean leader election configuration property. Setting this property to *true* will choose a follower replica as leader in the event of the failure of leader replica, even though the follower replica is not In-Sync Replica (ISR).

```
$ kafka-configs.sh \
--bootstrap-server localhost:9092 \
--entity-type brokers \
--entity-name 0 \
--alter \
--add-config unclean.leader.election.enable=true
```

We can now use the *--describe* option to list the dynamic Broker configurations for Broker id 0.

```
$ kafka-configs.sh \
--bootstrap-server localhost:9092 \
--entity-type brokers \
--entity-name 0 \
```

```
--describe
```

```
uzair@uzair:~$ kafka-configs.sh --bootstrap-server localhost:9092 --entity-type bro
kers --entity-name 0 --describe
Dynamic configs for broker 0 are:
  unclean.leader.election.enable=true sensitive=false synonyms={DYNAMIC_BROKER_CONF
IG:unclean.leader.election.enable=true, DEFAULT_CONFIG:unclean.leader.election.enab
le=false}
uzair@uzair:~$
```

Step 2: The dynamically-configured property can be deleted and returned to its default value by using the *--delete* option.

```
$ kafka-configs.sh \
--bootstrap-server localhost:9092 \
--entity-type brokers \
--entity-name 0 \
--alter \
--delete-config unclean.leader.election.enable
```

```
uzair@uzair:~$ kafka-configs.sh --bootstrap-server localhost:9092 --entity-type bro
kers --entity-name 0 --alter --delete-config unclean.leader.election.enable
Completed updating config for broker 0.
uzair@uzair:~$ kafka-configs.sh --bootstrap-server localhost:9092 --entity-type bro
kers --entity-name 0 --describe
Dynamic configs for broker 0 are:
uzair@uzair:~$
```

The *--describe* option can be utilized to verify the deletion. As we can see from the screenshot above, there are no dynamic *configs* returned indicating the successful deletion.

Step 3: Not only can the configurations be overridden dynamically per Broker, but some of them can be modified cluster-wide. Using the *--entity-default* option with *--entity-type* of Brokers will apply the modified value cluster-wide.

Let us set the *log.cleaner.threads* property to 2 for all the Brokers in the cluster.

```
$ kafka-configs.sh \
--bootstrap-server localhost:9092 \
--entity-type brokers \
--entity-default \
--alter \
--add-config log.cleaner.threads=2
```

```
uzair@uzair:~$ kafka-configs.sh --bootstrap-server localhost:9092 --entity-type bro
kers --entity-default --alter --add-config log.cleaner.threads=2
Completed updating default config for brokers in the cluster.
uzair@uzair:~$
```

You may check if the configuration has been successfully updated using the *--describe* as observed in the previous step. Moreover, you may also delete this updated cluster-wide configuration using the same process. However, be sure to utilize the *--entity-default* option instead of *--entity-name* option as depicted in the screenshot below:

```
uzair@uzair:~$ kafka-configs.sh --bootstrap-server localhost:9092 --entity-type bro
kers --entity-default --describe
Default configs for brokers in the cluster are:
  log.cleaner.threads=2 sensitive=false synonyms={DYNAMIC_DEFAULT_BROKER_CONFIG:log
.cleaner.threads=2}
uzair@uzair:~$ kafka-configs.sh --bootstrap-server localhost:9092 --entity-type bro
kers --entity-default --alter --delete-config log.cleaner.threads
Completed updating default config for brokers in the cluster.
uzair@uzair:~$ kafka-configs.sh --bootstrap-server localhost:9092 --entity-type bro
kers --entity-default --describe
Default configs for brokers in the cluster are:
uzair@uzair:~$
```

The following order of precedence is used for considering the configurations, as these configurations can be set in different ways:

1. Dynamic per-Broker configurations stored in the ZooKeeper.
2. Dynamic cluster-wide default configurations stored in the ZooKeeper.
3. Static Broker configurations from the *server.properties* file.
4. The Kafka default values. Please check the link in the references section to the Kafka documentation for all the default values of the configuration properties.

Step 4: As a lab challenge, override the Topic-level configurations. You may refer to the *Dynamic Configurations* section in the theory part of this chapter. You should be able to successfully override per-Topic configurations while creating a Topic using the *kafka-topics.sh* tool, as well as the *kafka-configs.sh* tool. In this way, you should be able to describe as well as delete the overridden configurations.

Task 3 is complete!

SUMMARY

In theory, we have learned the basic Kafka operations which include the Graceful Shutdown, Rack Awareness, and Scaling Kafka cluster. Subsequently, we learned the Kafka Consumer Groups tool. This tool provides all the relevant information regarding the Consumers. Thus, we can fetch the status of all the Consumer Groups in the Kafka cluster using the Consumer Groups tool. In addition, the Consumer Groups tool can also be utilized to list, describe and delete the Consumers.

Then, we have also learned the dynamically-overriding configurations for the Brokers and Topics. Moreover, we have also learned how to set quotas for the Producer and Consumer clients. Then, we determined how to handle Partitions. Kafka provides two tools for handling Partitions. These tools help us in electing the preferred replica leaders, reassigning the partitions to Brokers and also modifying the replication factor of the partitions.

Finally, in the labs of this chapter, we had our hands-on experience about executing a graceful shutdown of the Brokers, and the Consumer Groups tool. Finally, we learned how to dynamically override the configurations.

REFERENCES

- http://kafka.apache.org/
- https://zookeeper.apache.org/
- http://spark.apache.org/
- http://hadoop.apache.org/
- https://www.confluent.io/
- https://kafka.apache.org/downloads
- https://kafka.apache.org/documentation/#design
- https://kafka.apache.org/documentation/#configuration
- https://kafka.apache.org/documentation/#operations