Apache Kafka 3.8 Installation on Windows OS

Table of Contents

This document outlines the steps needed to install **Apache Kafka 3.8** with single broker instance configuration and multi-broker instance configuration and integrate with Apache Spark on Windows Operating system.

1. Overview:

Apache Kafka is an open-source distributed streaming processing platform used to handle the real time data storage with a processing capacity of millions of data or messages per second. Apache Kafka was originally developed at **LinkedIn** and later donated to **Apache Software Foundation** and is currently maintained by **Confluent** under Apache Software Foundation.

Apache Kafka works as a **broker** between sender and receiver and provides a **messaging system** *(responsible for exchanging messages between two applications)* in publish-subscribe (pub-sub) model. In pub-sub model, a sender *(known as **Producer** or **Publisher**)* sends/writes a message (nothing but data) to a persisted location called **topic** and then receiver *(known as **Consumer** or **Subscriber**)* consumes a message by subscribing to a topic.

Apache Kafka include components such as **Topic**, **Broker**, **Producer**, **Consumer** and **Zookeeper** and provides the following features:

- Capable of handling millions of messages per second.
- Works as a mediator between source and target systems in which source system (producer) data is sent to Apache Kafka where it decouples data and target system (consumer) consumes data from Kafka.
- Extreme high performance with low latency value less than 10ms.
- Fault-tolerant, highly available and resilient to node failures.

We must also need to know the following five core APIs provided by Apache Kafka:

1. **Producer API:** This API allows an application to publish streams of records to one or more topics.
2. **Consumer API:** This API allows an application to subscribe to one or more topics and process the stream of records produced to them.
3. **Streams API:** This API allows an application to effectively transform streams of records from input topics to output topics *(i.e. an application acts as a stream processor to consume an input stream from one or more topics, and produce an output stream to one or more output topics).*
4. **Connector API:** This API allows implementing connectors to move large amounts of data from source system to Kafka or push from Kafka into some sink data system.
5. **Admin API:** This API allows us to manage and inspect topics, partitions, brokers, ACLs and other Kafka objects.

Apache Kafka is heavily used across many organizations such as Netflix, UBER, Walmart, Twitter, LinkedIn, Mozilla, Oracle etc. with several use cases such as:

- **Messaging**: Kafka works better as a replacement for traditional message broker since it provides publish-subscribe messaging system.
- **Metrics:** Kafka is used for operational monitoring data by aggregating statistics from distributed applications.
- **Even Sourcing:** Kafka is an excellent backend for applications of event sourcing since it supports very large stored data.
- **Log Aggregation:** Kafka can be used across organization to collect logs from multiple services and make them available in a standard format to multiple consumers.
- **Stream Processing:** Kafka's strong durability is useful for stream processing in case of some popular frameworks such as Storm and Spark Streaming to read data from a topic, process it and write processed data to a new topic which becomes available for users and applications.
- **Website Activity Tracking:** Kafka can be used as a user activity tracking pipeline of a website as a set of real-time publish-subscribe feeds where each site activity that includes page views, searches, etc. is published to respective central topics.

2. Prerequisites:

The following prerequisites need to be installed before running Kafka.

1. **File Archiver:** Any file archiver such as **7zip** or **WinRAR** is needed to unzip the downloaded Spark binaries. 7zip can be downloaded from the 7zip Downloads website and WinRAR can be downloaded from the RAR lab Downloads website.

2. **JRE 8:** Kafka 3.x requires Java 8 runtime environment. We can either download just JRE 8 (Java Runtime Environment) for Windows offline installation from the official Java Download for Windows Offline website or download the whole JDK 8 (Java Development Kit) directly from Oracle Java Downloads website. For the complete JDK installation steps, look at here.

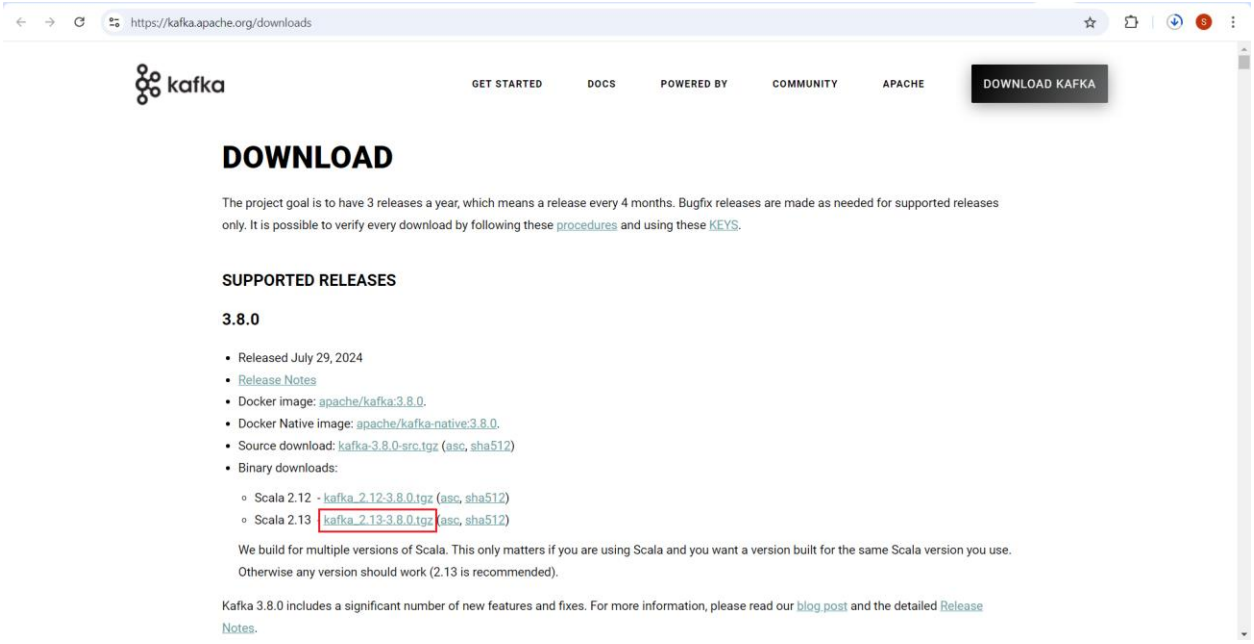
3. Install Kafka:

Let us see the step by step procedure to install Apache Kafka in Windows.

3.1. Download Kafka Binaries:

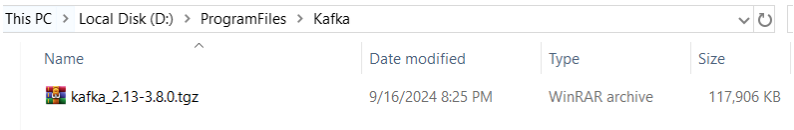
Download the stable version of Kafka from the official Apache Kafka Downloads website. At the time of this document preparation, the most recent stable release is 3.8.0.
Here, we will find multiple Kafka build versions of Scala *(you can select either Scala 2.12 or Scala 2.13 version for the Scala version that you use)*.

Let us select **Scala 2.13** version and so click on `kafka_2.13-3.8.0.tgz` link to download it into the **Downloads** folder in your machine.

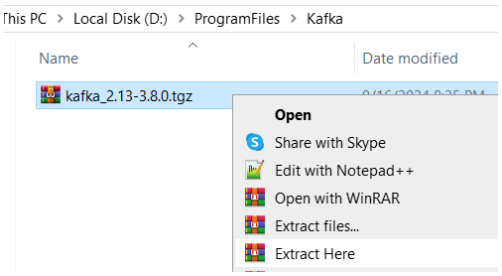


After the binary file is downloaded, unpack it using any file archiver (**7zip** or **WinRAR**) utility as below:

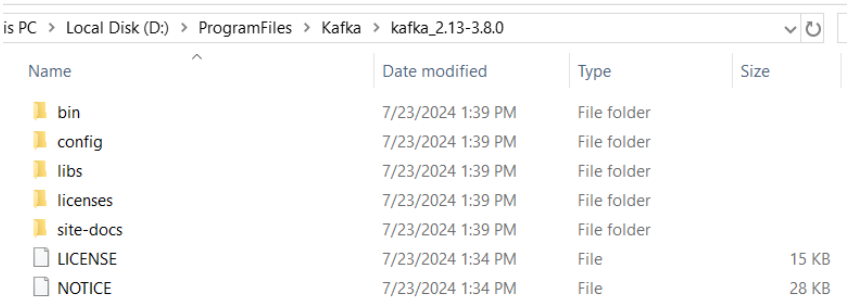
- Choose the installation directory in your machine and copy `kafka_2.13-3.8.0.tgz` file to that directory. Here, we are choosing Kafka installation directory as `D:\ProgramFiles\Kafka`.



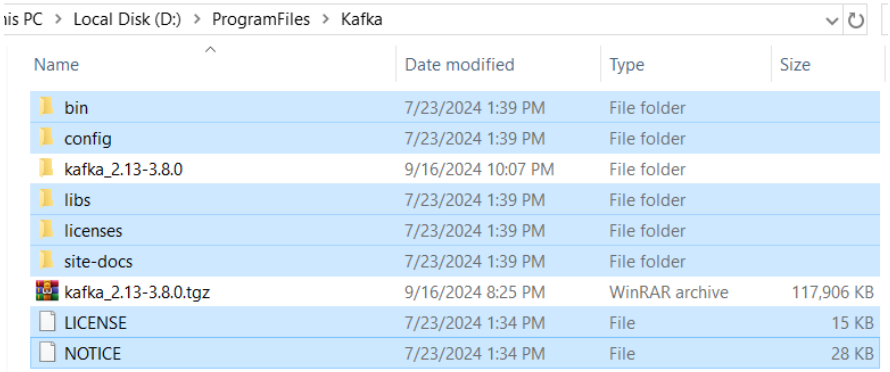
- Right click on `kafka_2.13-3.8.0.tgz` and choose **Extract Here** option.



- The tar file extraction may take few minutes to finish. After finishing, you see a folder named `kafka_2.13-3.8.0` which consists of Kafka binaries and libraries.



Note:
When your Kafka Installation path is too long, you might encounter an error "The input line is too long" while executing Kafka scrips. To avoid this error, move all folders and files from `D:\ProgramFiles\Kafka\kafka_2.13-3.8.0` location to `D:\ProgramFiles\Kafka` folder.



3.2. Set up Environment variables:

After installing pre-requisites and Kafka binaries, we should configure the below environment variables defining Java and Kafka default paths.

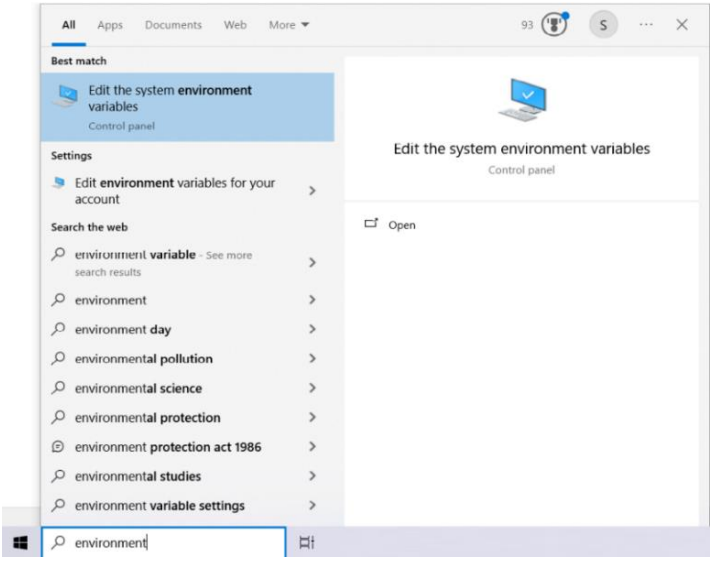
- **JAVA_HOME**: This is the JDK installation directory path in the machine *(in my machine, it is `D:\ProgramFiles\Java\jdk-1.8`)*. Ignore it if this is already done.

- **KAFKA_HOME**: This is the Kafka installation directory path in the machine *(in our case, it is `D:\ProgramFiles\Kafka`)*

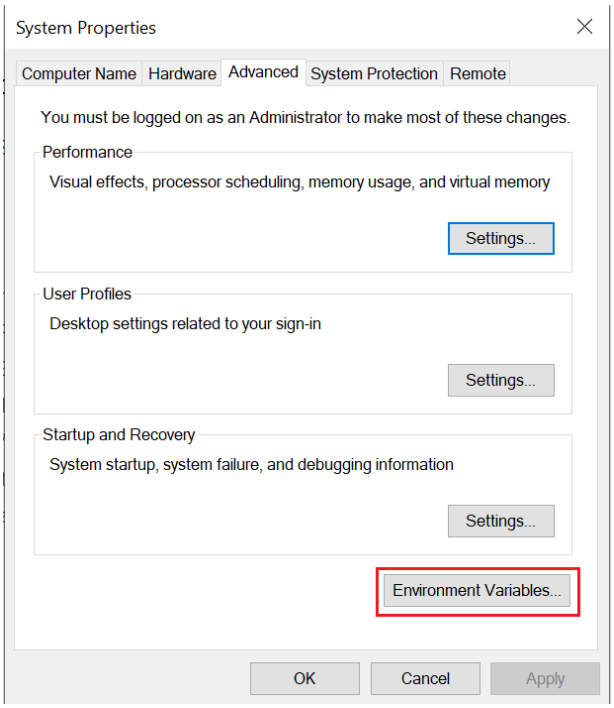
These variables need to be added to either **User environment variables** or **System environment variables** depending on Kafka configuration needed **for a single user** or **for multiple users**.

In this tutorial, we will add User environment variables since we are configuring Kafka for a single user. If you would like to configure Kafka for multiple users, then define System environment variables.

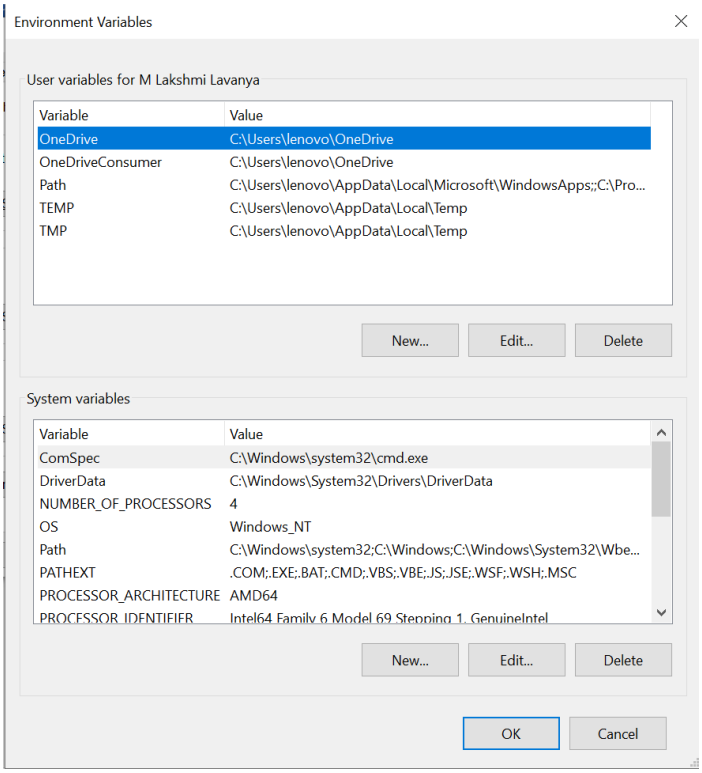
In the Windows search bar, start typing “environment variables” and select the first match which opens up **System Properties** dialog.



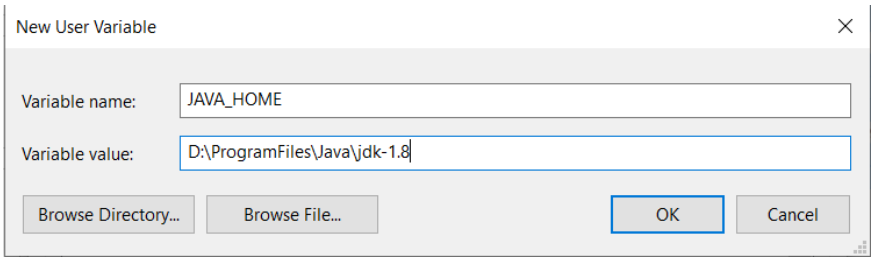
On the **System Properties** window, press **Environment Variables** button.



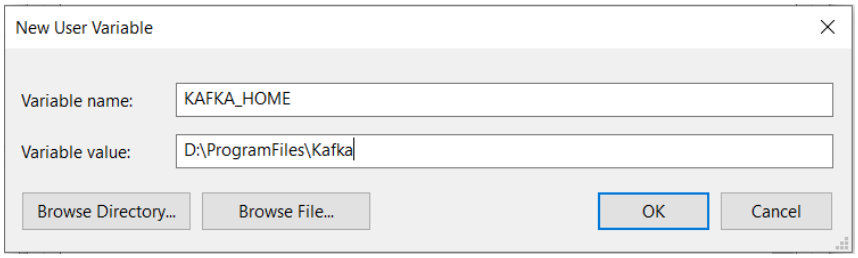
In the **Environment Variables** dialog, click on **New** under **User variables** section.



Add `JAVA_HOME` variable and press OK.

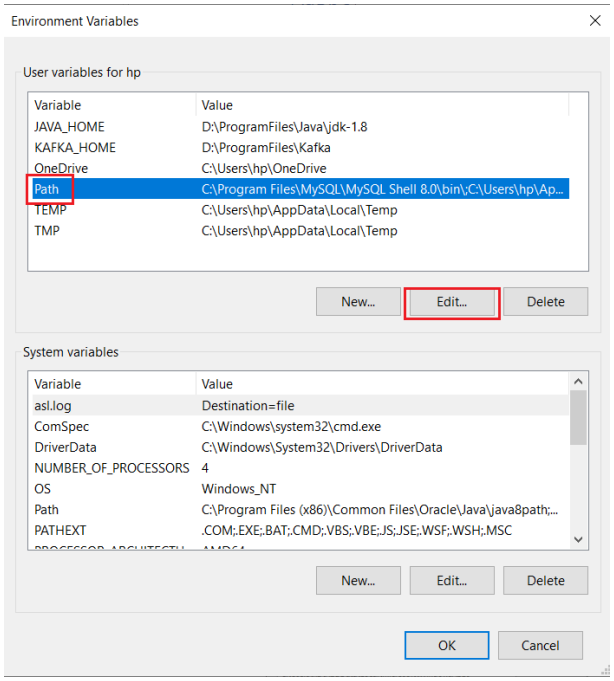


Click on **New** again and add `KAFKA_HOME` variable and press OK.



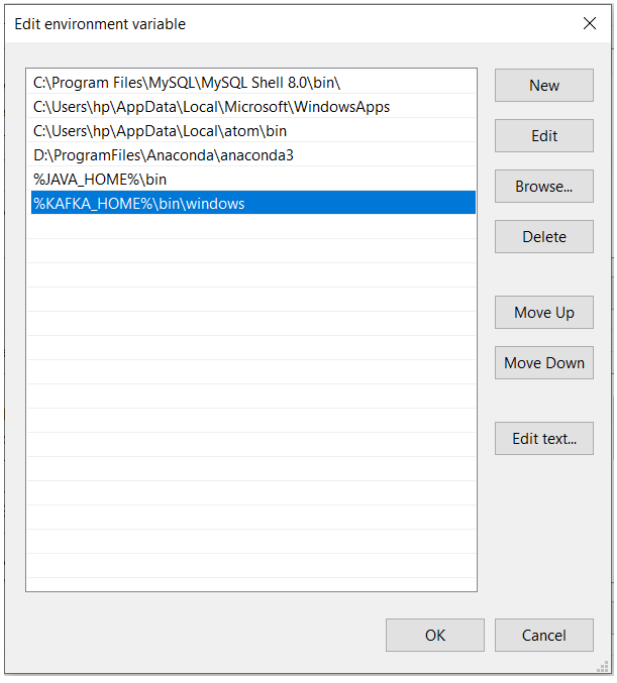
Now, we will update `PATH` variable to add Java, and Kafka binary paths.

Select `PATH` variable under **User Variables** and press **Edit** button.

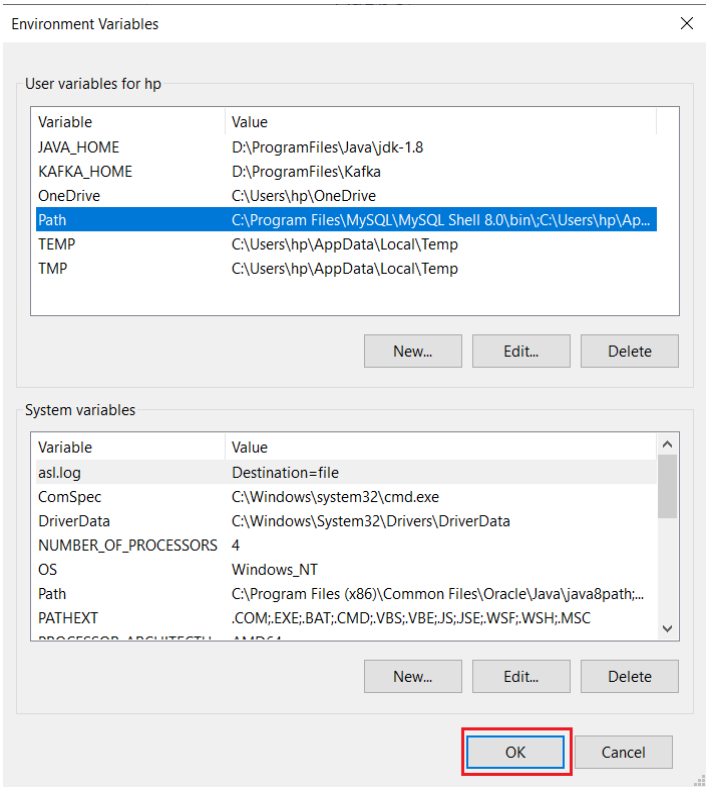


Press **New** and add the following values and press OK.

```
%JAVA_HOME%\bin
%KAFKA_HOME%\bin\windows
```



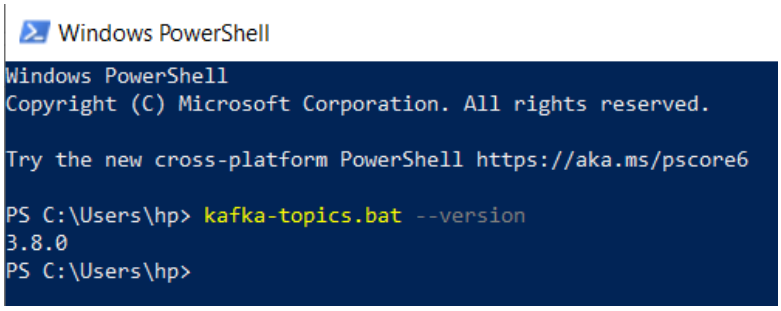
Press OK again to apply environment variable changes and close window.



3.3. Verify Kafka Installation:

Open **Windows PowerShell** and run the following command to verify if Kafka is installed properly:

```
kafka-topics.bat --version
```



Here, it shows **3.8.0** which indicates that Kafka has been installed successfully.

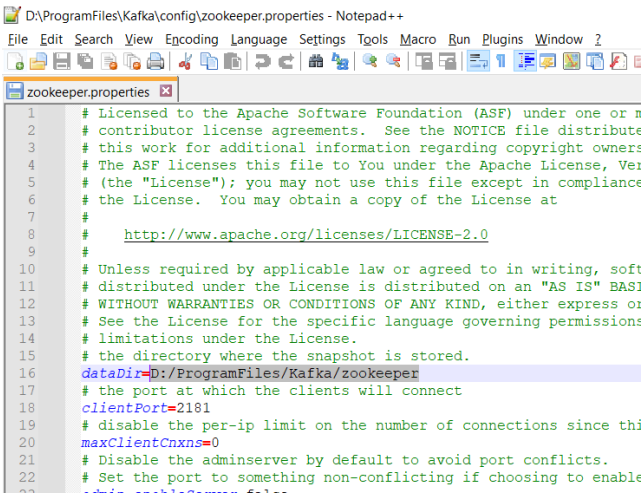
4. Configure Single Node-Single Broker:

In this configuration, we will setup a single Zookeeper with single Broker instance running on a single machine.

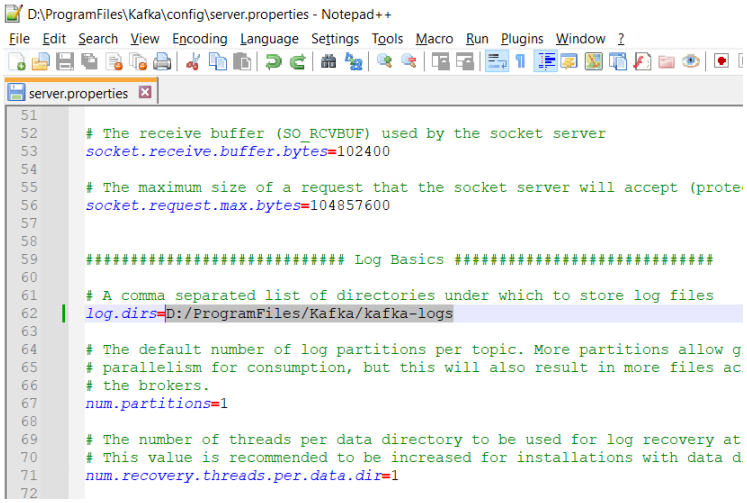
4.1. Configure Kafka Logs:

By default `zookeeper.properties` and `server.properties` files are configured with default directory locations that should be updated to the desired locations.

Open `zookeeper.properties` file in `%KAFKA_HOME%\config` folder and change the `dataDir` value from `/tmp/zookeeper` to `D:/ProgramFiles/Kafka/zookeeper` as shown below:



Next, open `server.properties` file in `%KAFKA_HOME%\config` folder and scroll down to `log.dirs` and change the value from `/tmp/kafka-logs` to `D:/ProgramFiles/Kafka/kafka-logs` as shown below:



Note that `kafka-logs` and `zookeeper` directories will be created automatically in `%KAFKA_HOME%` location when Zookeeper and Kafka broker services are started.

4.2. Start Kafka Servers:

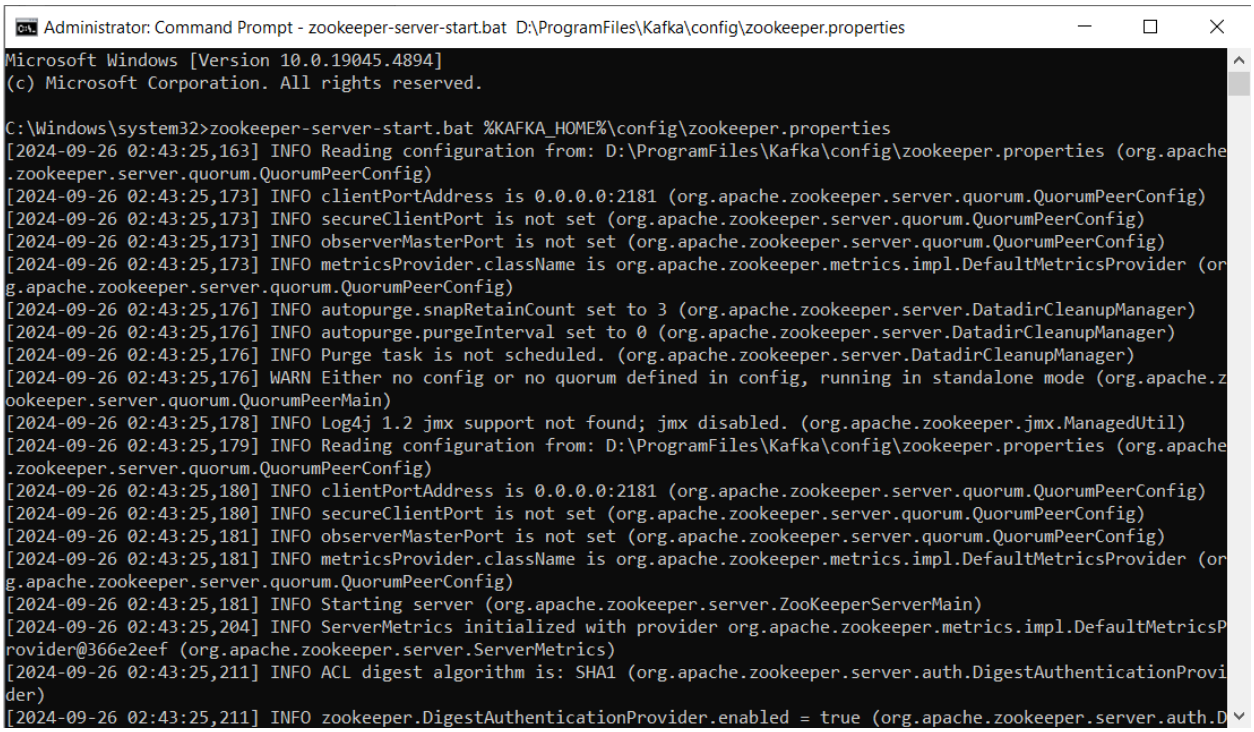
Now, we will start the Zookeeper and Kafka Broker services in the local system.

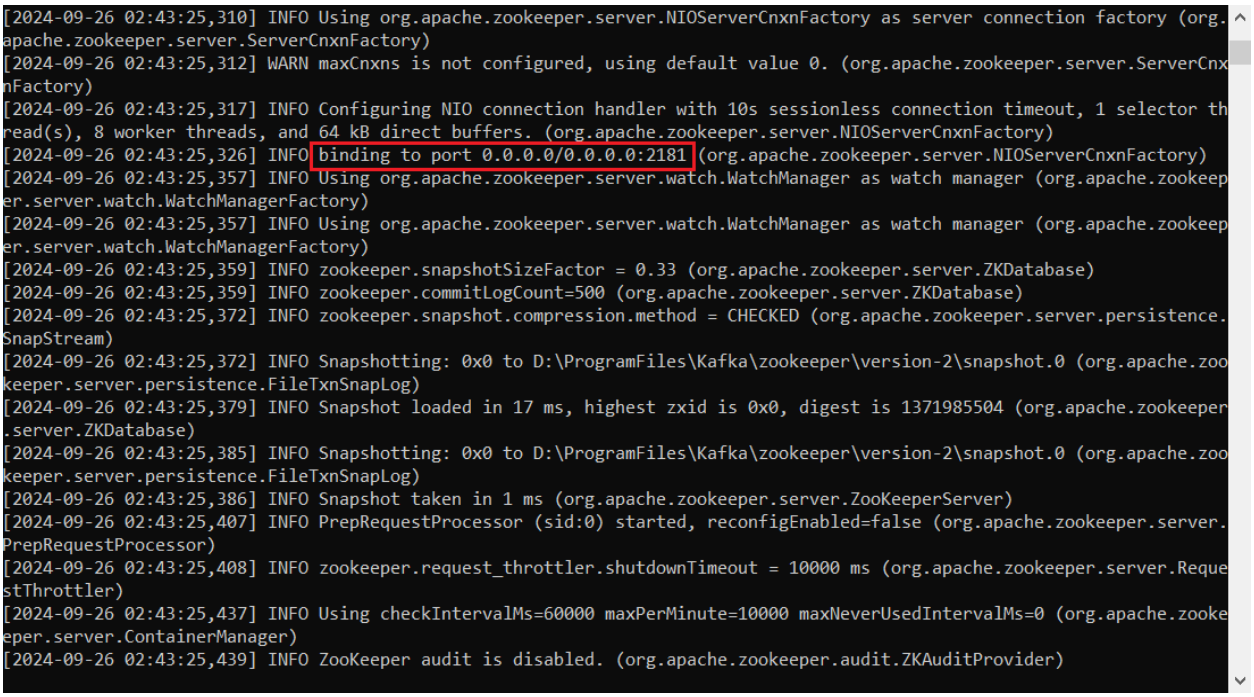
4.2.1. Start Zookeeper:

The Zookeeper is a centralized service in Apache Kafka that manages configuration changes and provides distributed configuration. It is responsible for managing the Kafka cluster by performing broker leader election and provides metadata to brokers about the processes running in the system and facilitate health checking of each broker in the cluster.

Open **Command Prompt** as **Administrator** and start Zookeeper using the below command:

```
zookeeper-server-start.bat %KAFKA_HOME%\config\zookeeper.properties
```





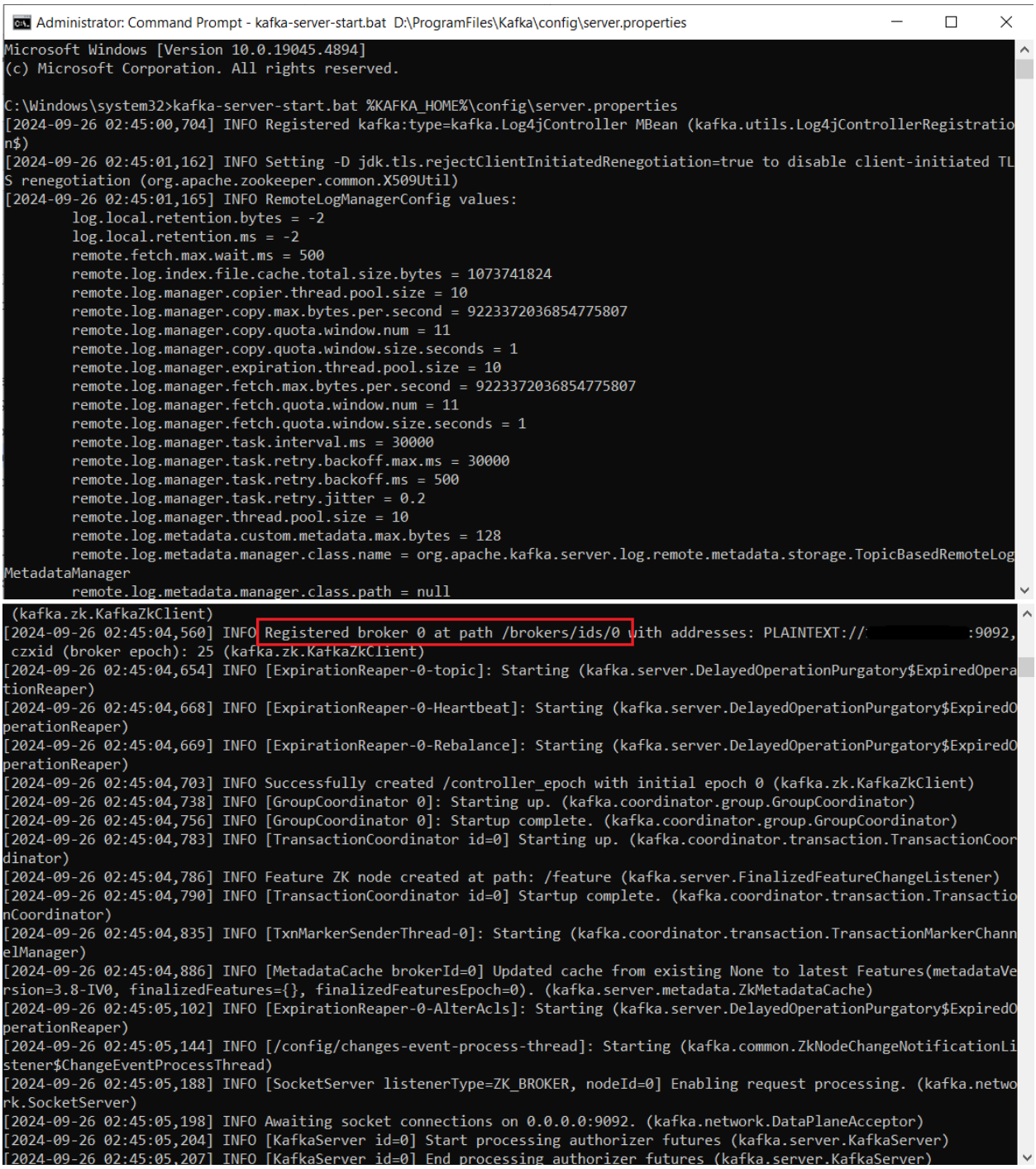
On the console, you will see a message "**binding to port 0.0.0.0/0.0.0.0:2181**" which indicates that the Zookeeper is up and running. Do not close this command prompt window.

4.2.2. Start Kafka Broker:

The Kafka broker manages the storage of messages in topics. Kafka cluster typically consists of multiple brokers to maintain the load balance and Kafka broker leader election is done by Zookeeper.

Open another **Command Prompt** as **Administrator** and start Kafka broker using the below command:

```
kafka-server-start.bat %KAFKA_HOME%\config\server.properties
```

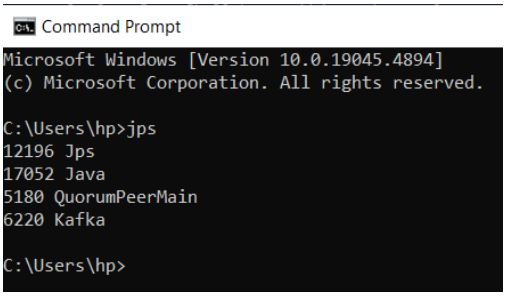


On the console, you will see a message "**Registered broker 0 at path /brokers/ids/0**" which indicates that the Kafka Broker is up and running. Do not close this command prompt window.

To verify the Kafka has been started successfully, use the `jps` command as below:

```
jps
```

It should display two demons **QuroumPeerMain** and **Kafka** process which denotes **Zookeeper**, and single **Kafka Broker** services are running.



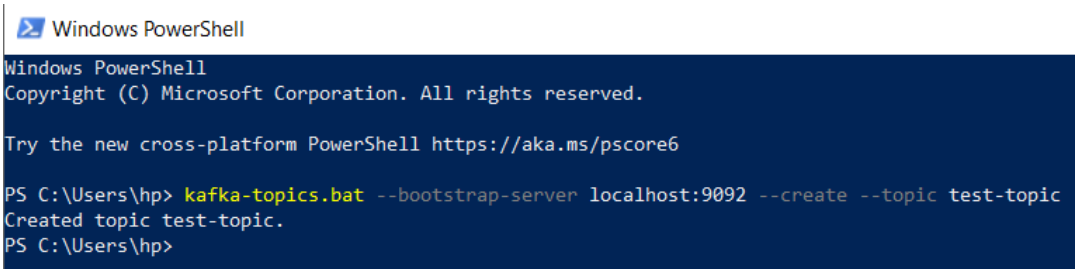
4.3. Create Topic:

Kafka provides a command line utility `kafa-topics.bat` to manage topics *(create, list, describe, delete, etc.)* on the cluster. This utility requires `-bootstrap-server` argument which takes Kafka broker instance details in `hostname:port` format.

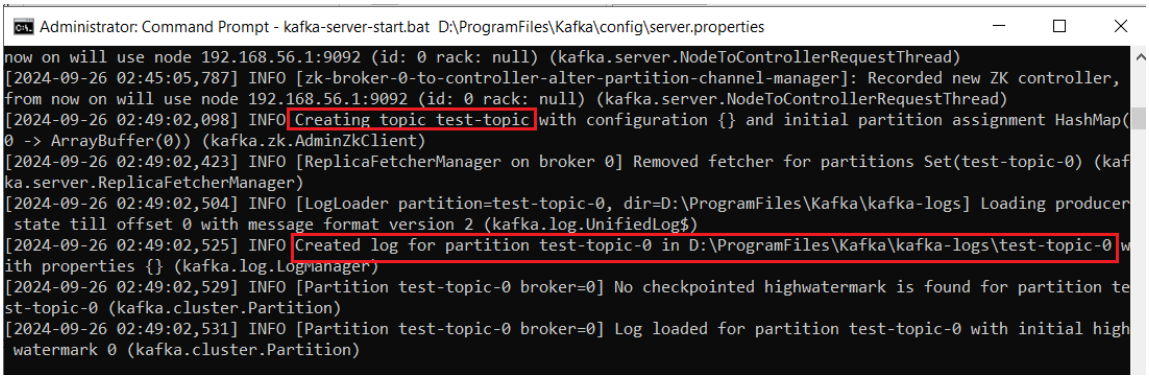
By default, a topic is created with 1 replication factor and 1 partition unless explicitly specified. Note that you cannot specify more than 1 replication factor when a single broker is running in the Kafka cluster *(i.e. maximum replication factor value = number of Kafka brokers)*.

Open **Windows PowerShell** or **Command Prompt** and run the below command to create a Kafka topic named `test-topic` :

```
kafka-topics.bat --bootstrap-server localhost:9092 --create --topic test-topic
```

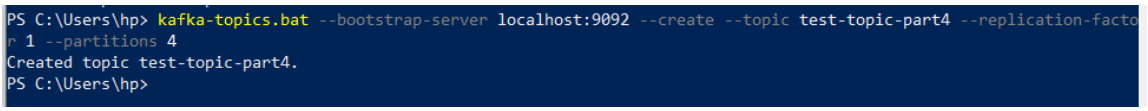


Once the topic has been created, you will see a message on the Kafka broker terminal window and the log location for the created topic `test-topic`.

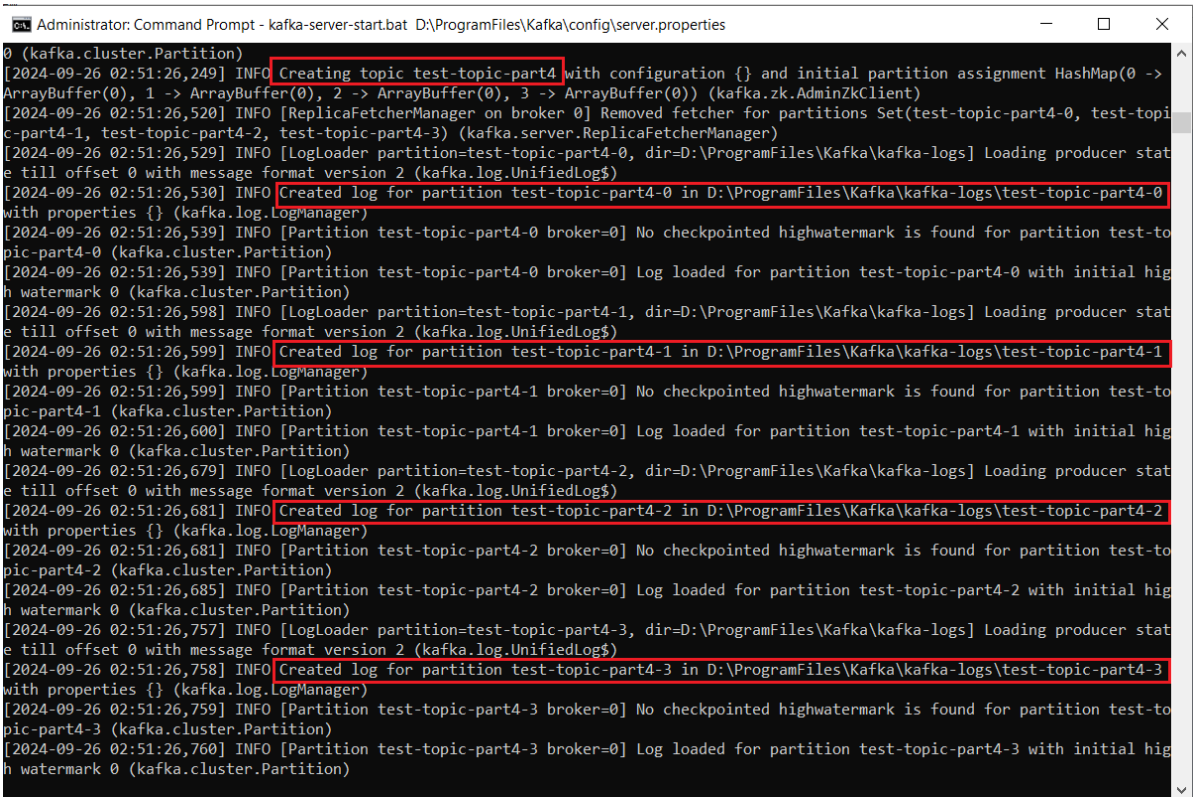


Let us run the below command to create another Kafka topic named `test-topic-part4` with 4 partitions:

```
kafka-topics.bat --bootstrap-server localhost:9092 --create --topic test-topic-part4 --replication-factor 1 --partitions 4
```



Once the topic has been created, you will see a message on the Kafka broker terminal window and the log location for each partition of the created topic `test-topic-part4`.



4.4. List Topics:

Use the below command to get the list of topics in Kafka server:

```
kafka-topics.bat --bootstrap-server localhost:9092 --list
```

```
PS C:\Users\hp> kafka-topics.bat --bootstrap-server localhost:9092 --list
test-topic
test-topic-part4
PS C:\Users\hp>
```

It has listed out two topics – `test-topic` and `test-topic-part4` – that we created.

4.5. Describe Topics:

Use the below command to get more details including topic partitions and replication information of topics in Kafka server:

```
kafka-topics.bat --bootstrap-server localhost:9092 --describe
```

```
PS C:\Users\hp> kafka-topics.bat --bootstrap-server localhost:9092 --describe
[2024-09-26 02:56:02,694] WARN [AdminClient clientId=adminclient-1] The DescribeTopicPartitions API is not supported, us
ing Metadata API to describe topics. (org.apache.kafka.clients.admin.KafkaAdminClient)
Topic: test-topic-part4 TopicId: 97bahCGXTJ-KjBzTfAxWbA PartitionCount: 4       ReplicationFactor: 1    Configs:
        Topic: test-topic-part4 Partition: 0    Leader: 0       Replicas: 0     Isr: 0  Elr: N/A        LastKnownElr: N/
A
        Topic: test-topic-part4 Partition: 1    Leader: 0       Replicas: 0     Isr: 0  Elr: N/A        LastKnownElr: N/
A
        Topic: test-topic-part4 Partition: 2    Leader: 0       Replicas: 0     Isr: 0  Elr: N/A        LastKnownElr: N/
A
        Topic: test-topic-part4 Partition: 3    Leader: 0       Replicas: 0     Isr: 0  Elr: N/A        LastKnownElr: N/
A
Topic: test-topic       TopicId: 7EcgcoNHRkC8IWynt9PKBg PartitionCount: 1       ReplicationFactor: 1    Configs:
        Topic: test-topic       Partition: 0    Leader: 0       Replicas: 0     Isr: 0  Elr: N/A        LastKnownElr: N/
A
PS C:\Users\hp>
```

It provided us the complete details of two topics – `test-topic` and `test-topic-part4` – that we created.

4.6. Run Kafka Producer:

Kafka distribution provides a command line utility named `kafa-console-producer.bat` to publish messages into a Kafka topic.

Open **Command Prompt** or **Windows PowerShell** and run the following command to start producing messages:

```
kafka-console-producer.bat --bootstrap-server localhost:9092 --topic
Hello-Kafka
```

```
PS C:\Users\hp> kafka-console-producer.bat --bootstrap-server localhost:9092 --topic Hello-Kafka
>
```

The producer will wait on input from user and publishes to `Hello-Kafka` topic in the Kafka cluster. Note that this topic is not available in Kafka cluster but it will be created when we start producing messages.

Now you can type a few lines of messages in the terminal:

```
PS C:\Users\hp> kafka-console-producer.bat --bootstrap-server localhost:9092 --topic Hello-Kafka
>My first message
[2024-09-26 02:57:47,478] WARN [Producer clientId=console-producer] Error while fetching metadata with correlation id 6
: {Hello-Kafka=LEADER_NOT_AVAILABLE} (org.apache.kafka.clients.NetworkClient)
[2024-09-26 02:57:47,602] WARN [Producer clientId=console-producer] Error while fetching metadata with correlation id 7
: {Hello-Kafka=LEADER_NOT_AVAILABLE} (org.apache.kafka.clients.NetworkClient)
>My second message
>This is third message
>
```

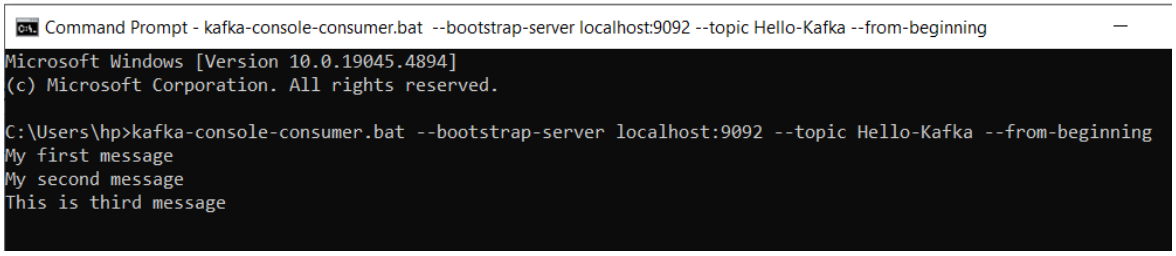
By default, every new line entered on the producer console is published as a new message to Kafka and this configuration can be changed in `%KAFKA_HOME%\config\producer.properties` file.

4.7. Run Kafka Consumer:

Kafka distribution provides a command line utility named `kafa-console-consumer.bat` to subscribe messages from a Kafka topic.

Open a new **Command Prompt** or **Windows PowerShell** and run the following command to start consuming messages:

```
kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic Hello-Kafka --from-beginning
```

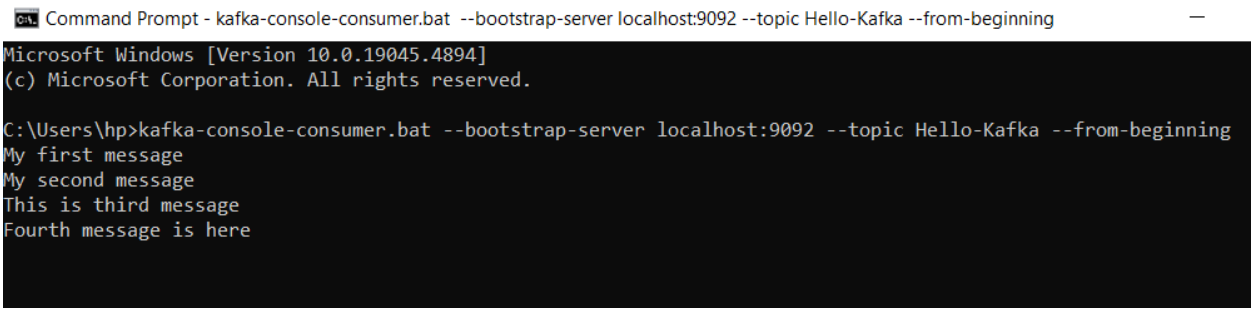


Here, you will see all messages that are produced to `Hello-Kafka` topic until now and will keep getting the new messages that will be produced to this topic.

Go to the producer window and enter a new message:

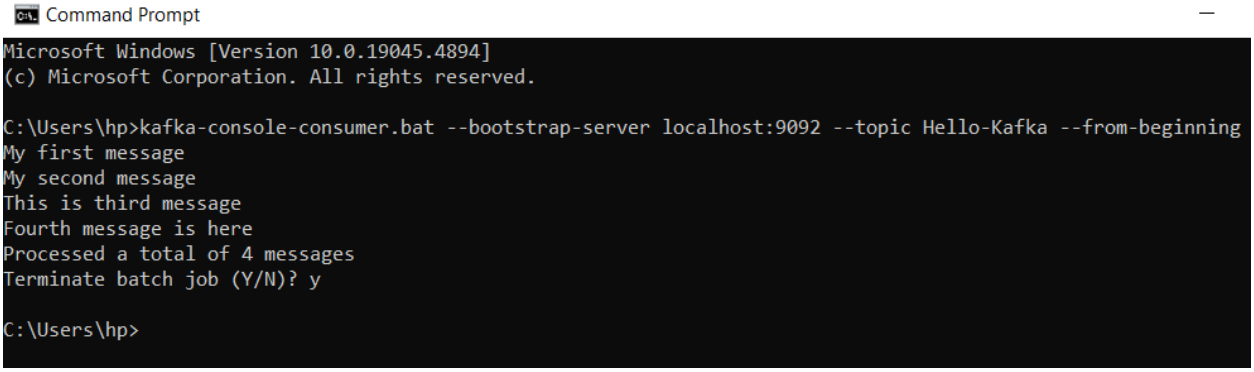
```
PS C:\Users\hp> kafka-console-producer.bat --bootstrap-server localhost:9092 --topic Hello-Kafka
>My first message
[2024-09-26 02:57:47,478] WARN [Producer clientId=console-producer] Error while fetching metadata with correlation id 6
: {Hello-Kafka=LEADER_NOT_AVAILABLE} (org.apache.kafka.clients.NetworkClient)
[2024-09-26 02:57:47,602] WARN [Producer clientId=console-producer] Error while fetching metadata with correlation id 7
: {Hello-Kafka=LEADER_NOT_AVAILABLE} (org.apache.kafka.clients.NetworkClient)
>My second message
>This is third message
>Fourth message is here
>
```

Now, go to the consumer window where you can see the fourth message that was produced:



Use **Ctrl + C** to exit out of Producer and Consumer windows:

```
PS C:\Users\hp> kafka-console-producer.bat --bootstrap-server localhost:9092 --topic Hello-Kafka
>My first message
[2024-09-26 02:57:47,478] WARN [Producer clientId=console-producer] Error while fetching metadata with correlation id 6
: {Hello-Kafka=LEADER_NOT_AVAILABLE} (org.apache.kafka.clients.NetworkClient)
[2024-09-26 02:57:47,602] WARN [Producer clientId=console-producer] Error while fetching metadata with correlation id 7
: {Hello-Kafka=LEADER_NOT_AVAILABLE} (org.apache.kafka.clients.NetworkClient)
>My second message
>This is third message
>Fourth message is here
>Terminate batch job (Y/N)? y
PS C:\Users\hp>
```



4.8. Delete Topic:

Use the following command to delete the topic named `test-topic` in Kafka cluster.

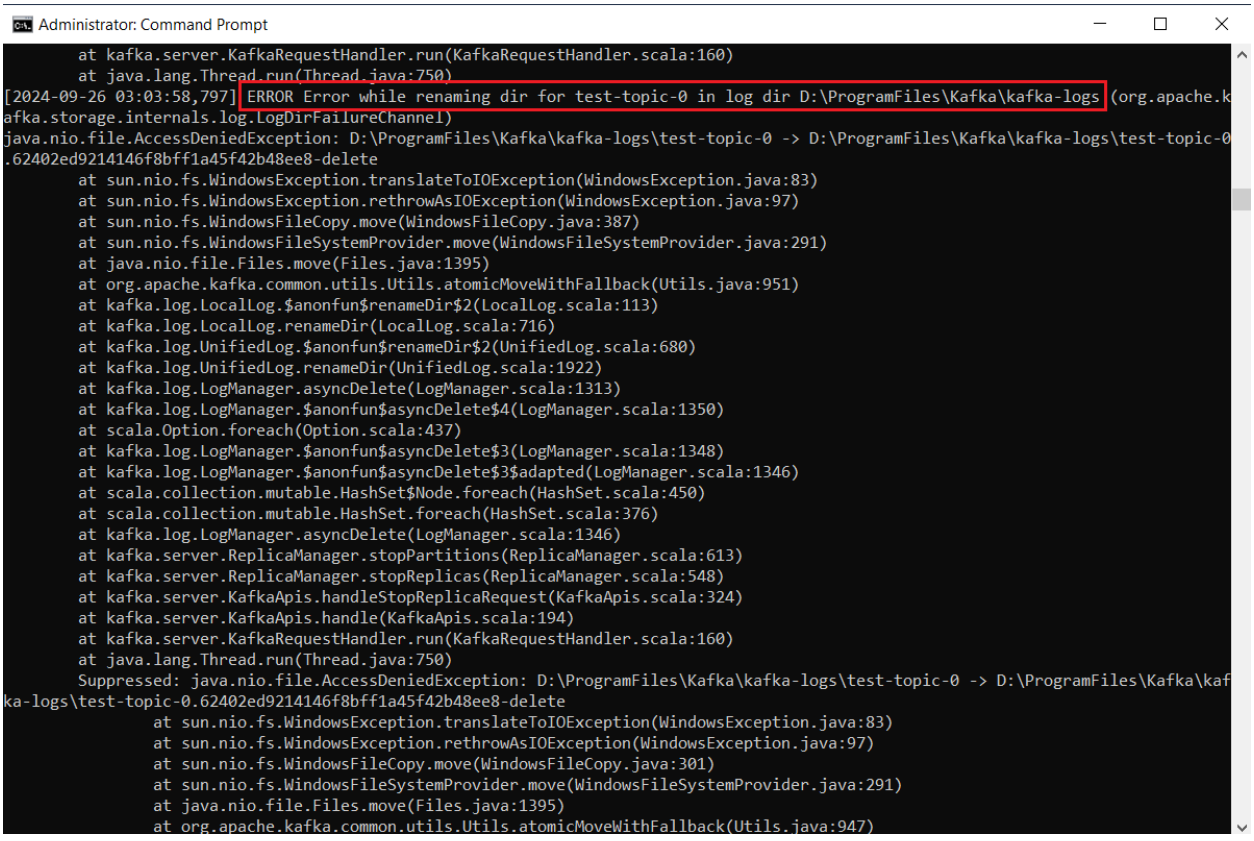
```
kafka-topics.bat --bootstrap-server localhost:9092 --delete --topic test-topic
```

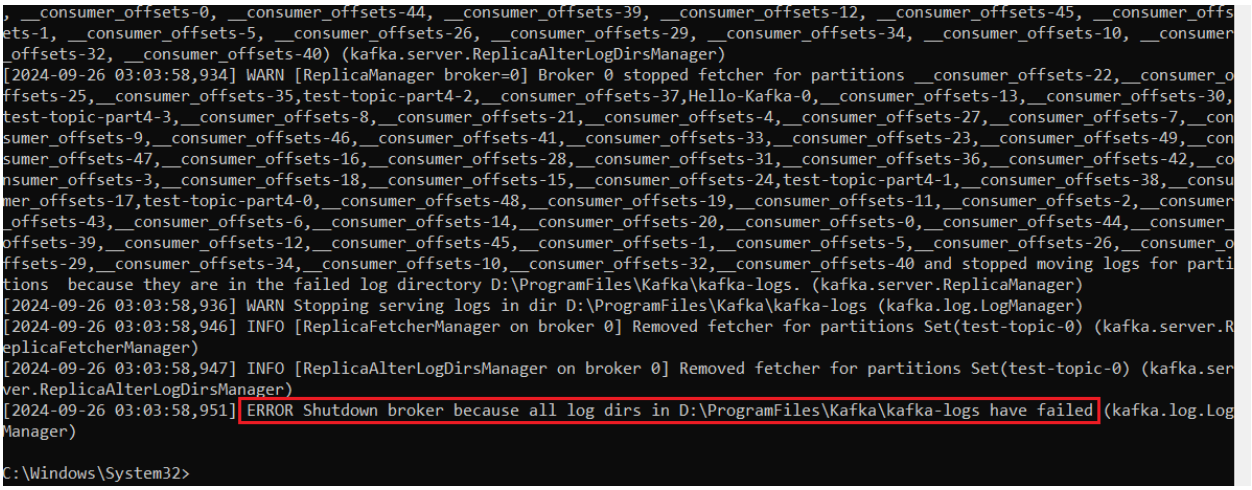
```
PS C:\Users\hp> kafka-topics.bat --bootstrap-server localhost:9092 --delete --topic test-topic
PS C:\Users\hp>
```

Note:

After issuing `--delete --topic` command for Kafka running in Windows, it will only mark the respective topic name for deletion but will not physically delete the topic folder present in `%KAFKA_HOME%\kafka-logs` folder.

On the Kafka broker terminal window, you will see an *Error while renaming dir for test-topic-0 in log dir D:\ProgramFiles\Kafka\kafka-logs* and then Kafka server was shut down.





Note: The above issue occurs in Windows operating system since Kafka is intended to run on Linux operating system. If we try to run Kafka in Windows natively, it may arise several issues over a period of time.

Therefore, it is recommended to run Apache Kafka on Windows through:

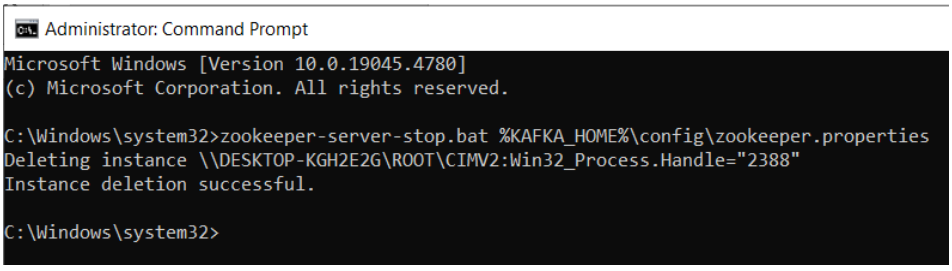
- **WSL2** or **Docker** if using **Windows 10** or above.
- **Docker** if using **Windows 8** or below.

It is not recommended to run Kafka on the JVM on Windows, because it lacks some of the Linux specific features (for ex: POSIX). We will also run into issues at some point if we try to run Kafka on Windows without WSL2.

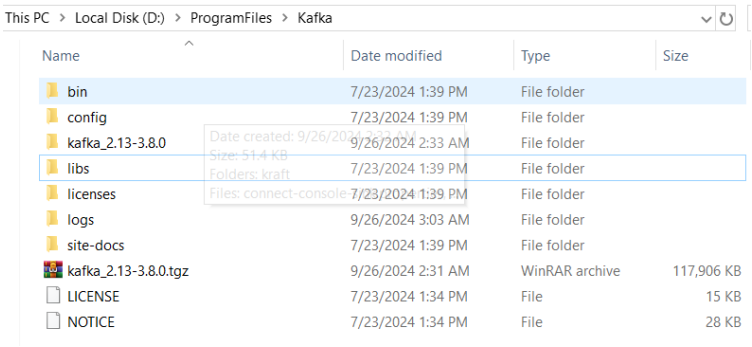
To resolve the above error, follow the below steps to manually delete the respective topic folder

- Open new **Command Prompt** as **Administrator** and stop the zookeeper with this command:

```
zookeeper-server-stop.bat
%KAFKA_HOME%\config\zookeeper.properties
```

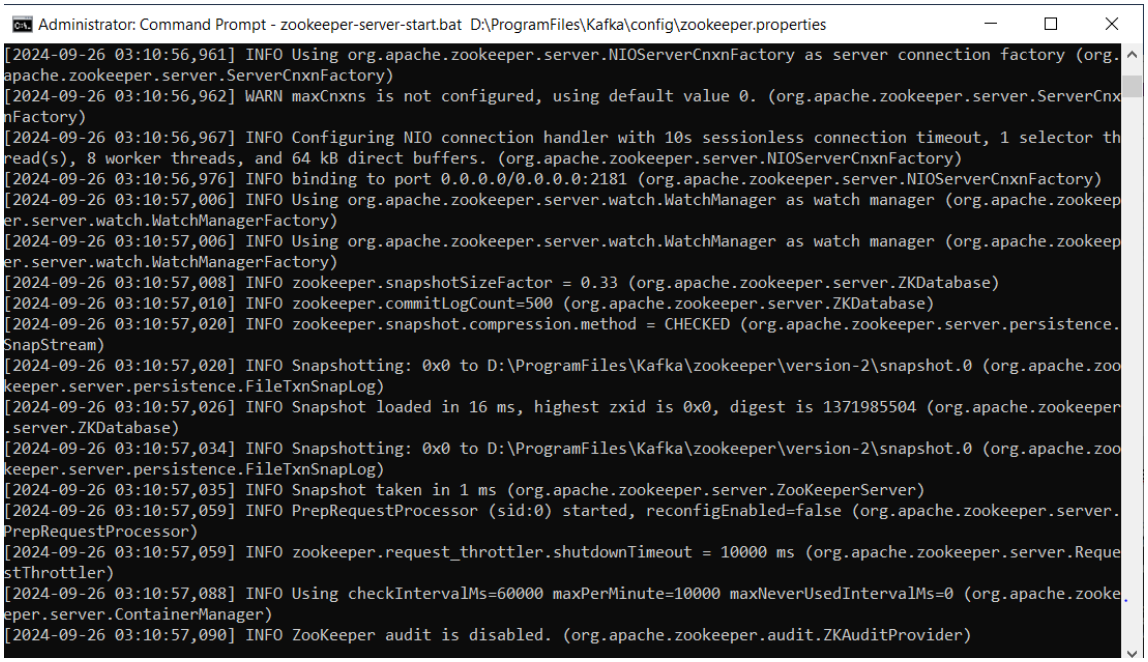


- Go to `%KAFKA_HOME%` location and delete `kafka-logs` and `zookeeper` folders



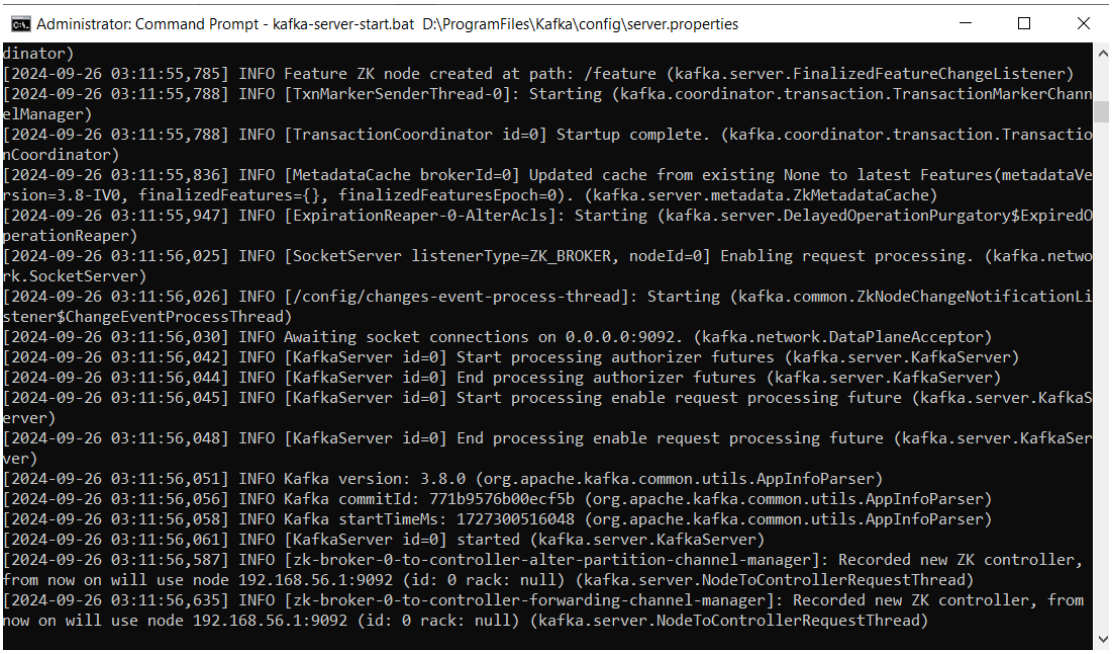
- Open new **Command Prompt** as **Administrator** and start zookeeper with this command:

```
zookeeper-server-start.bat
%KAFKA_HOME%\config\zookeeper.properties
```

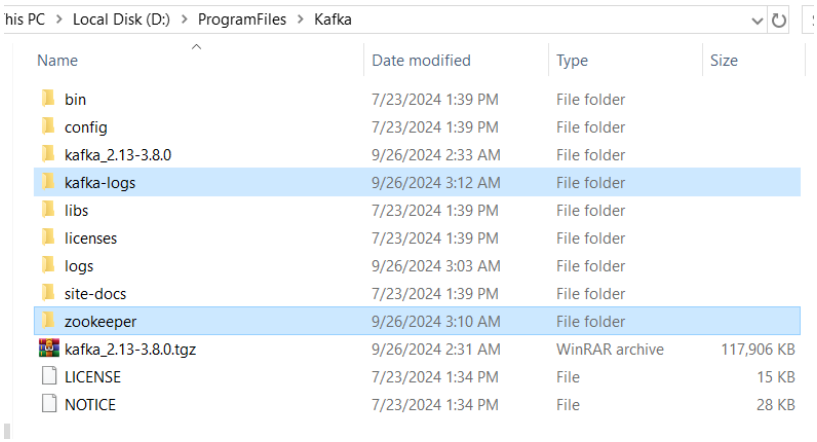


- Open new **Command Prompt** as **Administrator** and start Kafka broker with this command:

```
kafka-server-start.bat %KAFKA_HOME%\config\server.properties
```



- Go to `%KAFKA_HOME%` location and you can see `kafka-logs` and `zookeeper` folders were created freshly.



Note that we have deleted all topics and will have to start creating topic freshly.

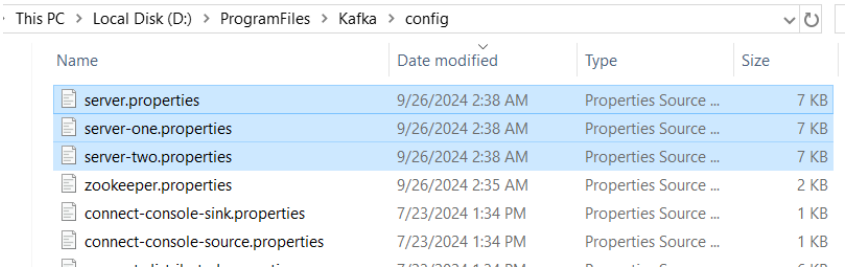
5. Configure Single Node-Multiple Brokers:

In this configuration, we will have single Zookeeper and multiple broker instances running on a single machine.

5.1. Setup Multiple Brokers:

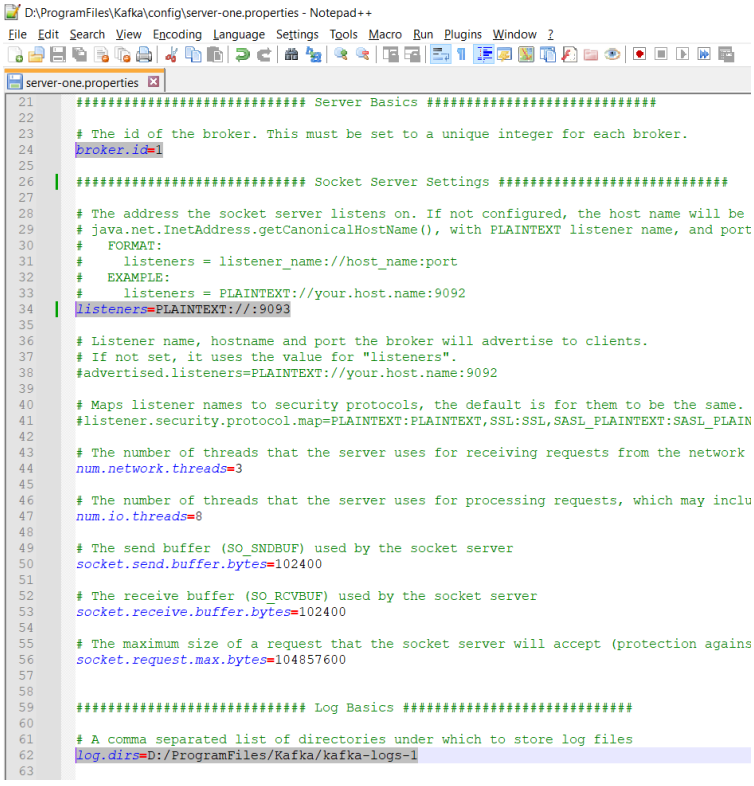
In the previous configuration, one Kafka broker instance was already configured in `server.properties` file.

To setup multiple broker instances, take two copies of `server.properties` file and rename as `server-one.properties` and `server-two.properties` in `%KAFKA_HOME%\config` location.



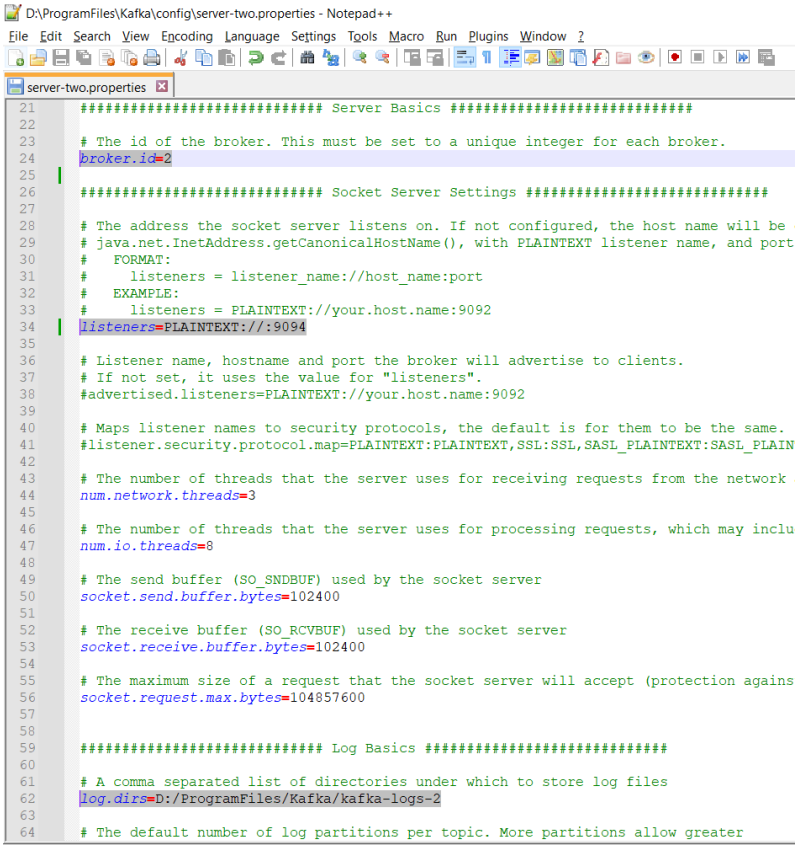
Open `server-one.properties` and change the following settings:

```
broker.id=1
listeners=PLAINTEXT://:9093
log.dirs=D:/ProgramFiles/Kafka/kafka-logs-1
```



Open `server-two.properties` and change the following settings:

```
broker.id=2
listeners=PLAINTEXT://:9094
log.dirs=D:/ProgramFiles/Kafka/kafka-logs-2
```



Note that `kafka-logs-1` and `kafka-logs-2` folders will be created automatically in `%KAFKA_HOME%` when Kafka brokers are started.

5.2. Start Kafka Servers:

Now, we will start the Zookeeper and multiple Kafka Broker instances in the local system.

5.2.1. Start Zookeeper:

Open **Command Prompt** as **Administrator** and start Zookeeper using the below command *(stop it if it is already running)*

```
zookeeper-server-stop.bat %KAFKA_HOME%\config\zookeeper.properties
```

```
zookeeper-server-start.bat %KAFKA_HOME%\config\zookeeper.properties
```

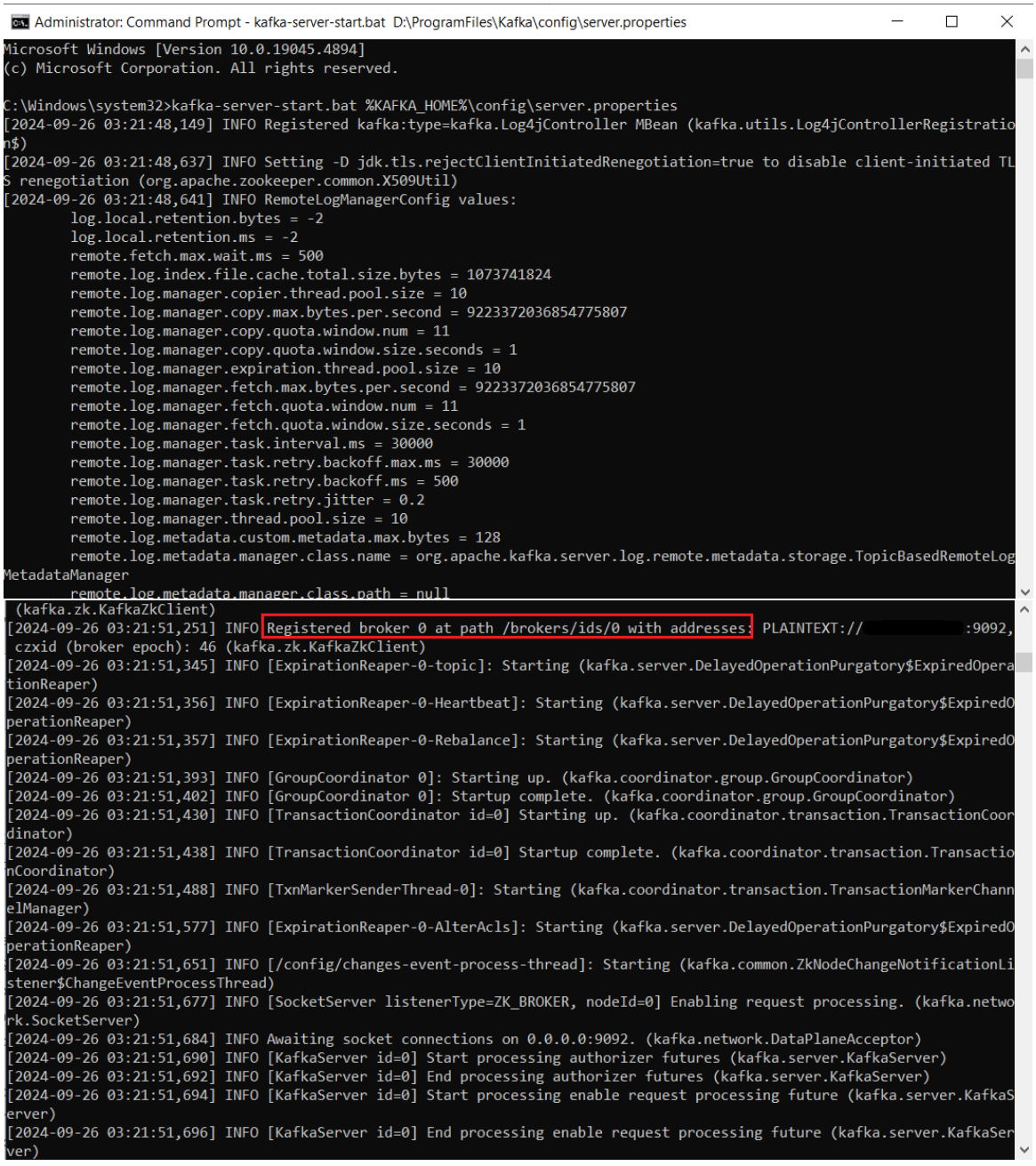
On the console, you will see a message "**binding to port 0.0.0.0/0.0.0.0:2181**" which indicates that the Zookeeper is up and running. Do not close this command prompt window.

5.2.2. Start Kafka Brokers:

Open another **Command Prompt** as **Administrator** and start the first Kafka broker instance (stop it if it is already running) using the below command:

```
kafka-server-stop.bat %KAFKA_HOME%\config\server.properties
```

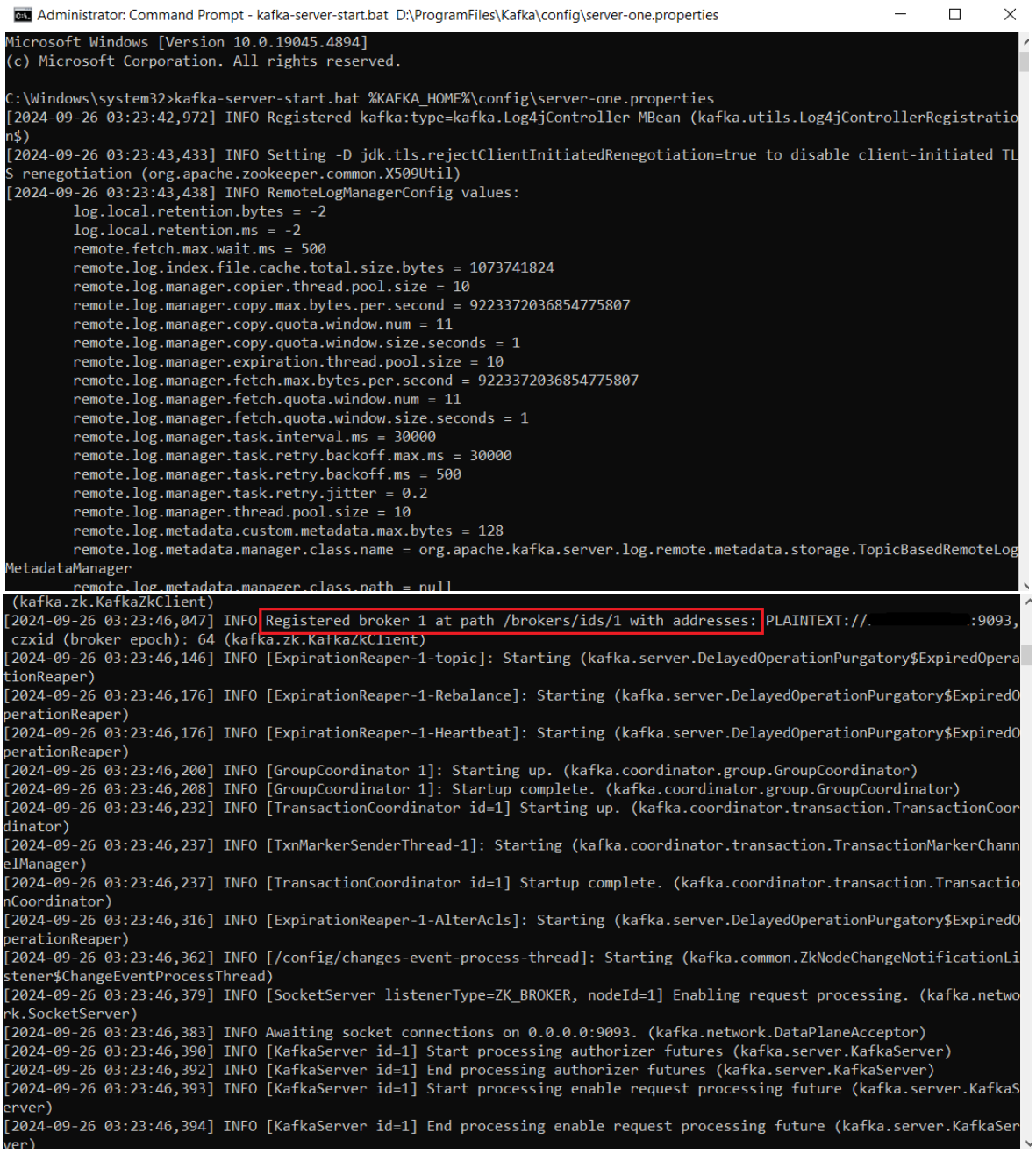
```
kafka-server-start.bat %KAFKA_HOME%\config\server.properties
```



On the console, you will see a message "**Registered broker 0 at path /brokers/ids/0**" which indicates that the first Kafka Broker is up and running. Do not close this command prompt window.

Open another **Command Prompt** as **Administrator** and start the second Kafka broker instance using the below command:

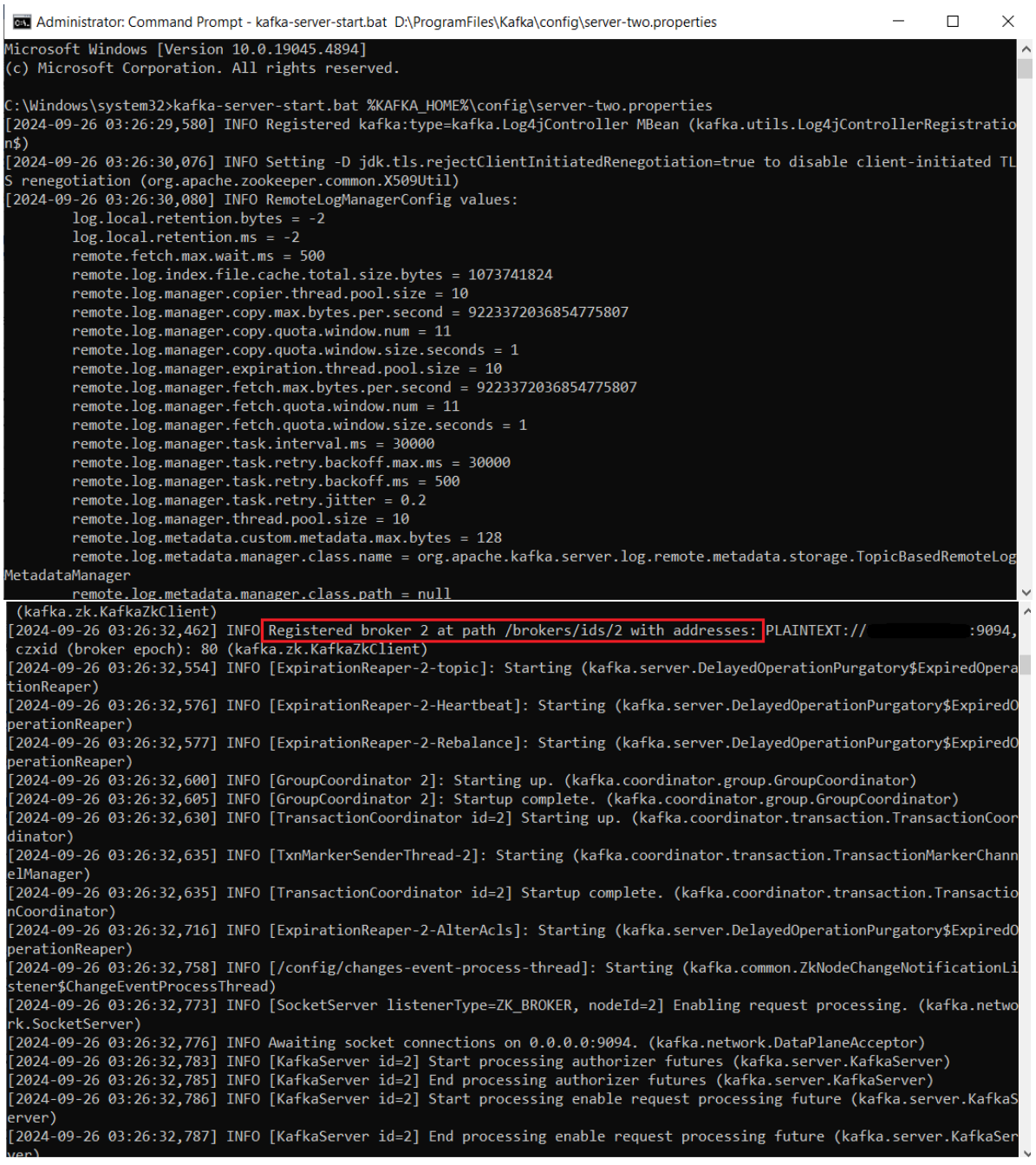
```
kafka-server-start.bat %KAFKA_HOME%\config\server-one.properties
```



On the console, you will see a message "**Registered broker 1 at path /brokers/ids/1**" which indicates that the second Kafka Broker is up and running. Do not close this command prompt window.

Open another **Command Prompt** as **Administrator** and start the third Kafka broker instance using the below command:

```
kafka-server-start.bat %KAFKA_HOME%\config\server-two.properties
```

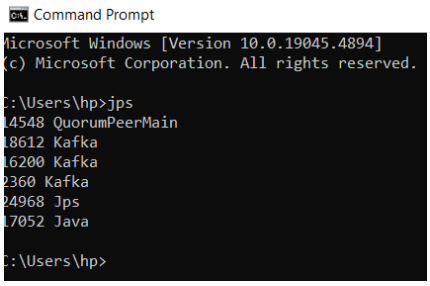


On the console, you will see a message "**Registered broker 2 at path /brokers/ids/2**" which indicates that the third Kafka Broker is up and running. Do not close this command prompt window.

Run `jps` command to verify the Kafka has been started successfully with multiple brokers

```
jps
```

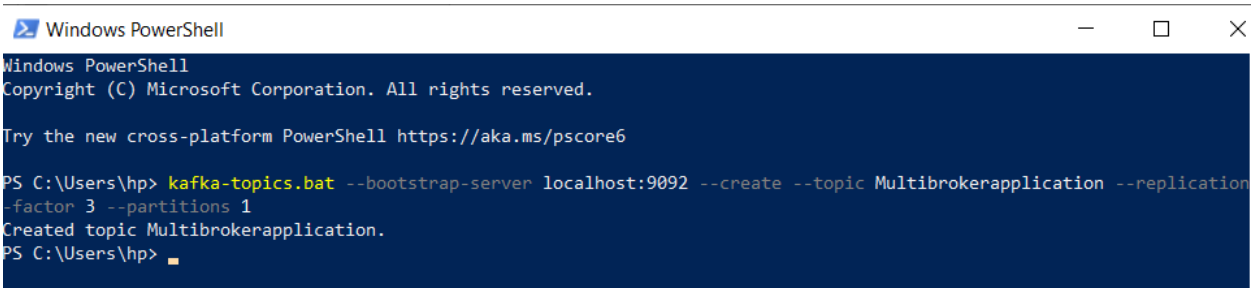
It should display one **QuroumPeerMain** daemon and three **Kafka** daemons which denotes one **Zookeeper**, and three **Kafka Broker** services are running.



5.3. Create Topic:

Open **Windows PowerShell** and run the below command to create a new Kafka topic with replication factor 3 (since we have 3 broker instances running)

```
kafka-topics.bat --bootstrap-server localhost:9092 --create --topic
Multibrokerapplication --replication-factor 3 --partitions 1
```



5.4. Describe Topic:

Use the below command to check which broker is running on the current created topic:

```
kafka-topics.bat --bootstrap-server localhost:9092 –describe --topic
Multibrokerapplication
```

```
PS C:\Users\hp> kafka-topics.bat --bootstrap-server localhost:9092 --describe --topic Multibrokerapplication
[2024-09-26 03:30:08,687] WARN [AdminClient clientId=adminclient-1] The DescribeTopicPartitions API is not supported, us
ing Metadata API to describe topics. (org.apache.kafka.clients.admin.KafkaAdminClient)
Topic: Multibrokerapplication   TopicId: qAHJI760QAS8UNLFEMVy8Q PartitionCount: 1       ReplicationFactor: 3    Configs:

        Topic: Multibrokerapplication   Partition: 0    Leader: 0       Replicas: 0,2,1 Isr: 0,2,1      Elr: N/A
LastKnownElr: N/A
PS C:\Users\hp>
```

Here, we see that our first broker (**Leader 0** with `broker.id` 0) is the leader. Then **Replicas:0,2,1** means that all the brokers replicate the topic and **Isr** is the set of in-sync replicas *(a subset of replicas that are currently alive and caught up by the leader).*

5.5. Run Kafka Producer:

Open **Command Prompt** or **Windows PowerShell** and run the following command to start producing messages to second broker running on port 9093:

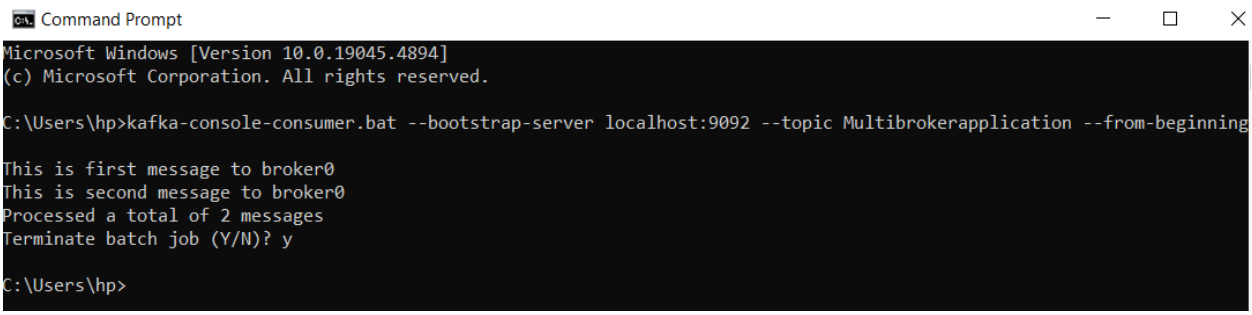
```
kafka-console-producer.bat --bootstrap-server localhost:9093 --topic
Multibrokerapplication
```

```
PS C:\Users\hp> kafka-console-producer.bat --bootstrap-server localhost:9093 --topic Multibrokerapplication
>This is first message to broker0
>This is second message to broker0
>
```

5.6. Run Kafka Consumer:

Open a new **Command Prompt** or **Windows PowerShell** and run the following command to start consuming messages:
Note that we can connect to any broker to see messages since we have given the replication factor as 3 while creating `Multibrokerapplication` topic.

```
kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic
Multibrokerapplication --from-beginning
```




```
kafka-console-consumer.bat --bootstrap-server localhost:9093 --topic
Multibrokerapplication --from-beginning
```

```
C:\Users\hp>kafka-console-consumer.bat --bootstrap-server localhost:9093 --topic Multibrokerapplication --from-beginning

This is first message to broker0
This is second message to broker0
Processed a total of 2 messages
Terminate batch job (Y/N)? y

C:\Users\hp>
```

```
kafka-console-consumer.bat --bootstrap-server localhost:9094 --topic Multibrokerapplication --from-beginning
```

```
C:\Users\hp>kafka-console-consumer.bat --bootstrap-server localhost:9094 --topic Multibrokerapplication --from-beginning

This is first message to broker0
This is second message to broker0
Processed a total of 2 messages
Terminate batch job (Y/N)? y

C:\Users\hp>
```

Here, you will see all messages that are produced to `Multibrokerapplication` topic until now and will keep getting the new messages that will be produced to this topic.

Use **Ctrl + C** to exit out of Producer and Consumer windows.

5.7. Delete Topic:

Use the following command to delete the topic named `test-topic` in Kafka cluster.

```
kafka-topics.bat --bootstrap-server localhost:9094 --delete --topic Multibrokerapplication
```

```
PS C:\Users\hp> kafka-topics.bat --bootstrap-server localhost:9094 --delete --topic Multibrokerapplication
PS C:\Users\hp>
```

Note:

After issuing `--delete --topic` command for Kafka running in Windows, it will only mark the respective topic name for deletion but will not physically delete the topic folder present in `kafka-logs`, `kafka-logs-1` and `kafka-logs-2` folders in `%KAFKA_HOME%` location.

On the three Kafka broker terminal windows, you will see an *Error while renaming dir for test-topic-0 in log dir D:\ProgramFiles\Kafka\kafka-logs* and then Kafka server was shut down.

Note: The above issue occurs in Windows operating system since Kafka is intended to run on Linux operating system. If we try to run Kafka in Windows natively, it may arise several issues over a period of time.

Therefore, it is recommended to run Apache Kafka on Windows through:

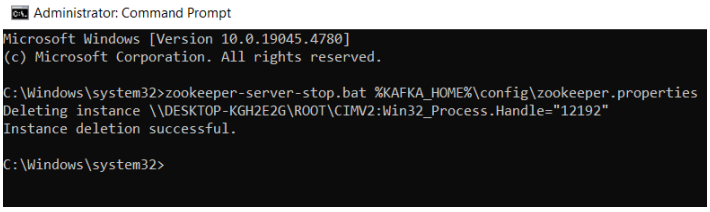
- **WSL2** or **Docker** if using **Windows 10** or above.
- **Docker** if using **Windows 8** or below.

It is not recommended to run Kafka on the JVM on Windows, because it lacks some of the Linux specific features (for ex: POSIX). We will also run into issues at some point if we try to run Kafka on Windows without WSL2.

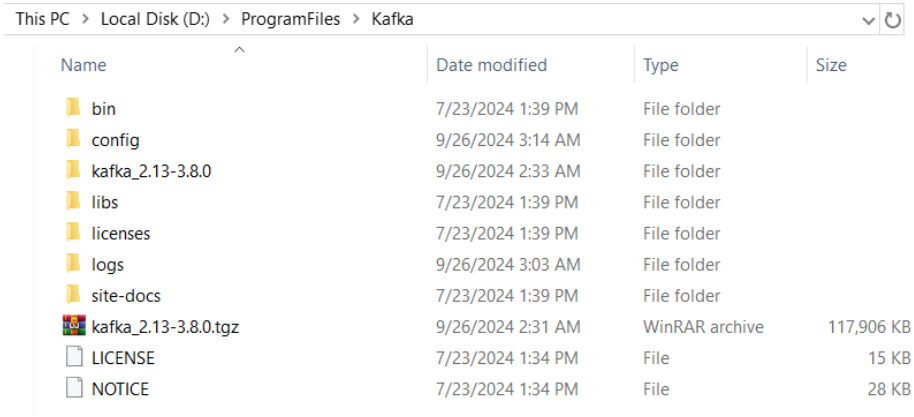
To resolve the above error, follow the below steps to manually delete the respective topic folder

- Open new **Command Prompt** as **Administrator** and stop the zookeeper with this command:

```
zookeeper-server-stop.bat
%KAFKA_HOME%\config\zookeeper.properties
```

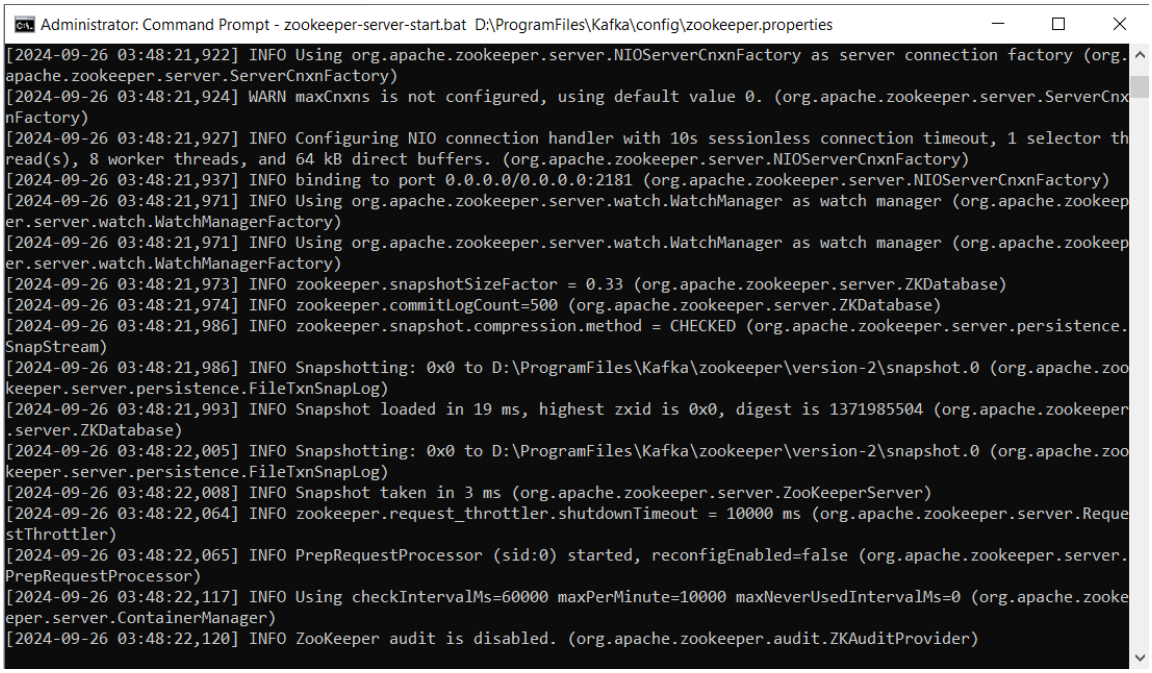


- Go to `%KAFKA_HOME%` location and delete `zookeeper`, `kafka-logs`, `kafka-logs-1` and `kafka-logs-2` folders.



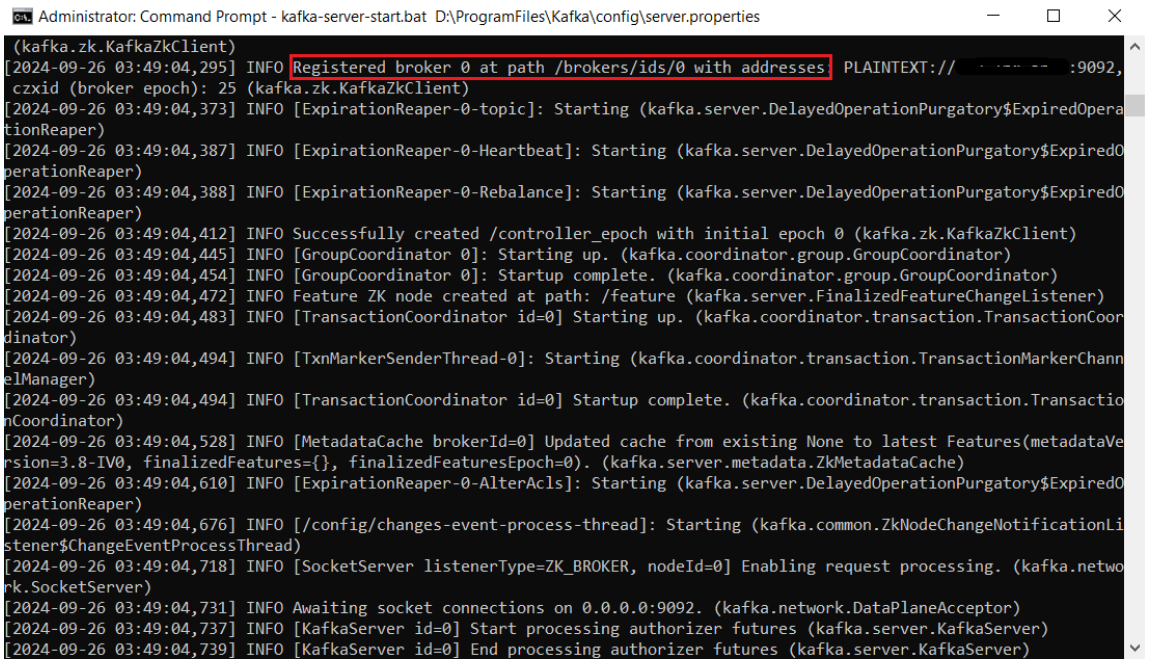
- Open new **Command Prompt** as **Administrator** and start zookeeper with this command:

```
zookeeper-server-start.bat
%KAFKA_HOME%\config\zookeeper.properties
```

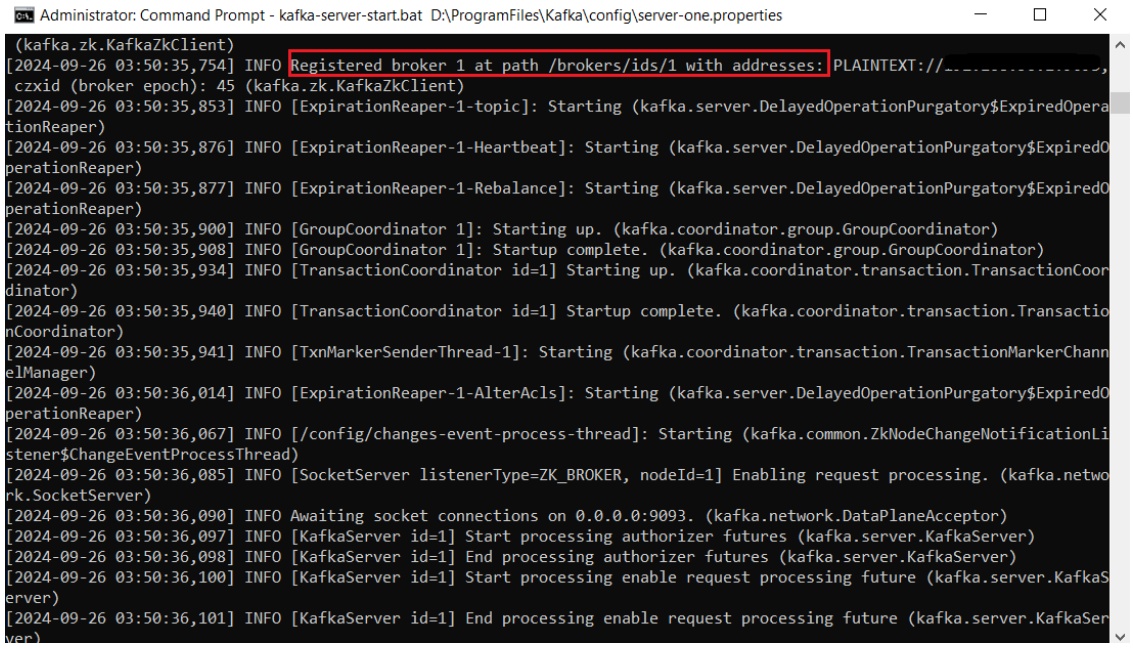


- Open three **Command Prompts** as **Administrator** and start three Kafka brokers with these commands:

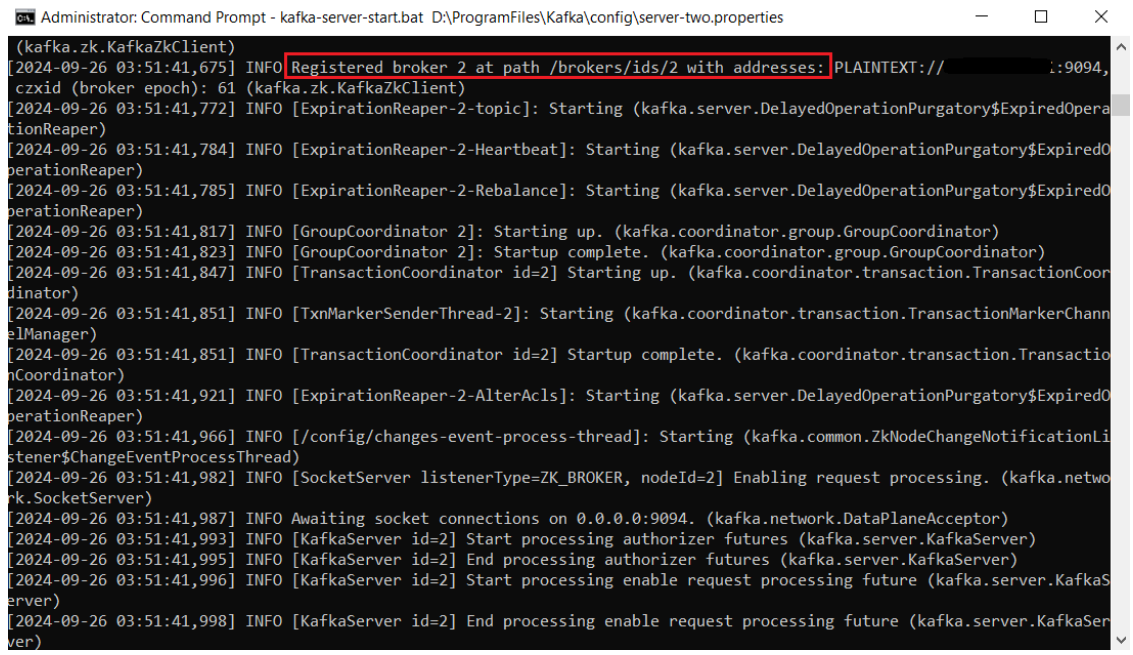
```
kafka-server-start.bat %KAFKA_HOME%\config\server.properties
```



```
kafka-server-start.bat %KAFKA_HOME%\config\server-one.properties
```




```
kafka-server-start.bat %KAFKA_HOME%\config\server-two.properties
```



Note that we have deleted all topics and will have to start creating topic freshly.

6. Simple Kafka Project using Java:

In the previous section, we used Kafka command line tools such as `kafka-topics.bat`, `kafka-console-producer.bat` and `kafka-console-consumer.bat` to create a Kafka topic and start producing and subscribing data. In this section, we will create a simple Kafka project to do the same activities using Java programming language.

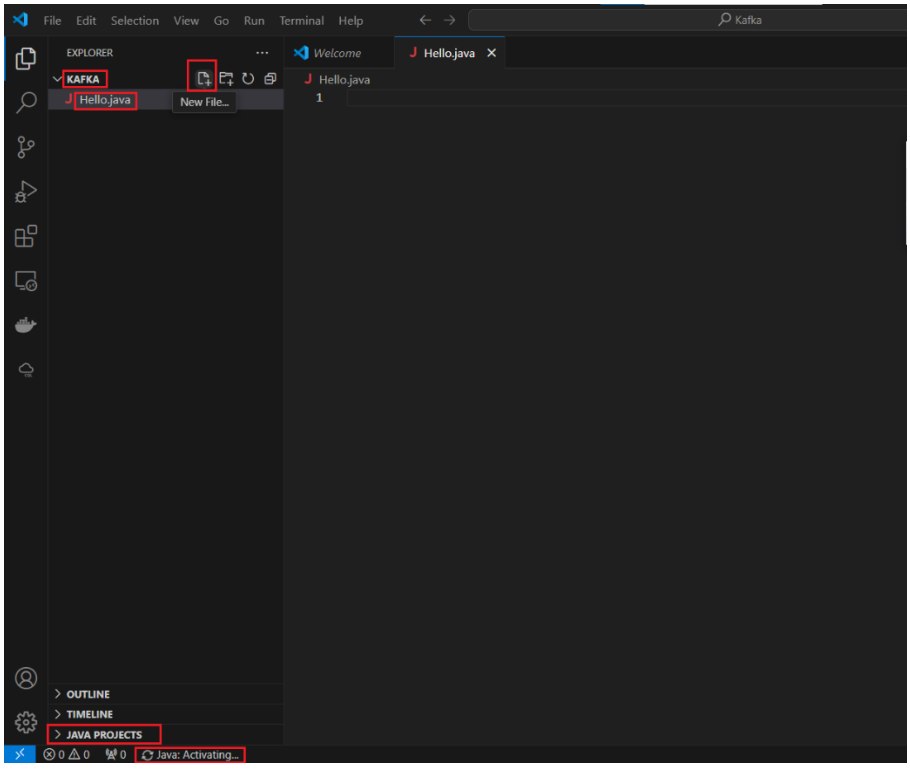
6.1. Launch Java IDE:

To create a Kafka project using Java, we need an **IDE tool** such as Eclipse, VS Code, Intellij IDEA, Notepad, etc. and **JDK 1.8** or higher.

In this tutorial, we will use **VS Code** to create a java project. If you do not have it installed already, you can download from official Microsoft site and install Extension Pack for Java extension.

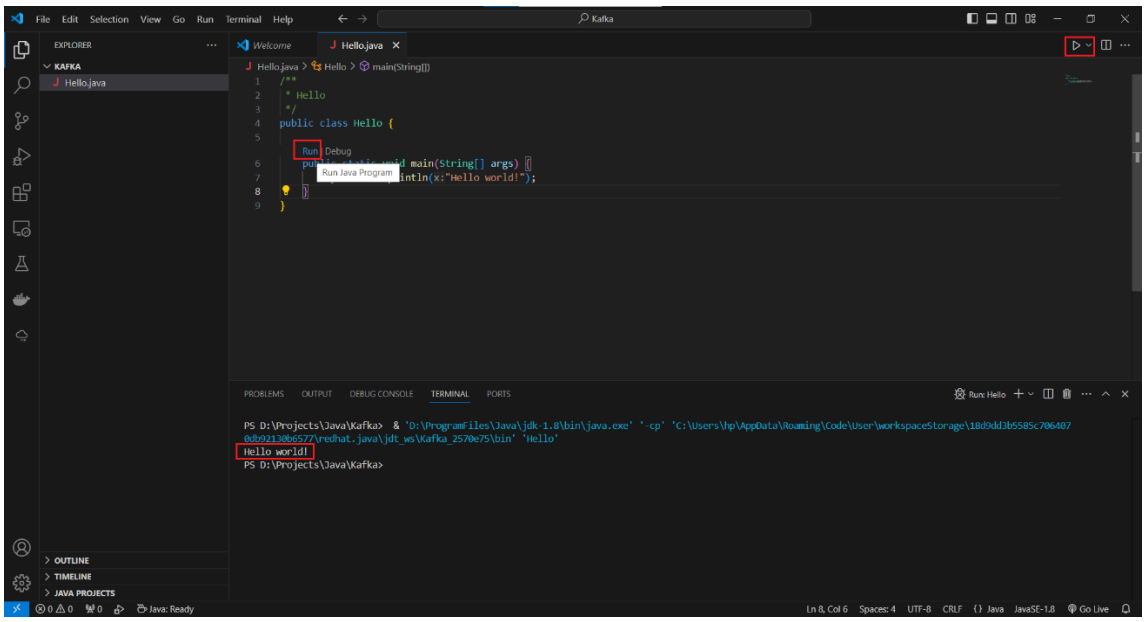
In **VS Code** application, open a new folder and navigate to D drive to create a folder named **`Kafka`** and select it in your local system.

On the **Explorer** window, click on **New File** button next to **KAFKA** folder and name it with .java extension (for example, `Hello.java`) after which it starts activating Java extension and you can see **JAVA PROJECTS** section on the bottom left of the application.

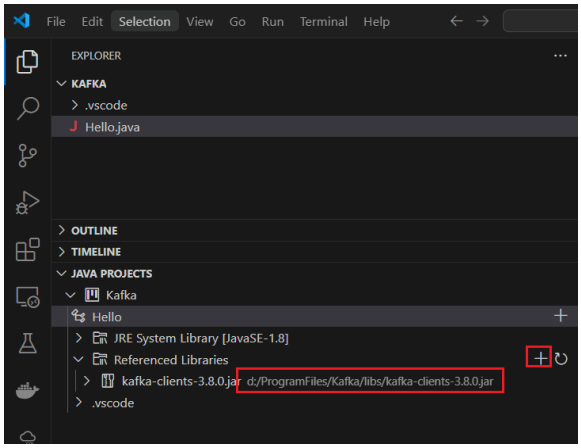


Once the Java extension is ready, enter the below code in `Hello.java` file and click on **Run** link or button to see the output of java program on the **Terminal.**

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```



Now that our sample Java code is working, we will write java programs to create a Kafka topic, producer and consumer application. To do this, expand **JAVA PROJECTS** window on the left bottom window of the VS Code, then expand **KAFKA** project and click on **+** icon next to **Referenced Libraries** and select `D:\ProgramFiles\Kafka\libs\kafka-clients-3.8.0.jar` file.



Note that Apache Kafka provides `kafka-clients-xxx.jar` library *(located at `%KAFKA_HOME%\libs` folder)* which contains `admin` API, `producer` API and `consumer` APIs using which we will create a topic, send data and read data programmatically.

Before proceeding further, make sure Kafka services are up and running. If not already running, open 4 **Command Prompts** in **Administrator** mode and run the following commands to start Zookeeper first and then 3 Kafka brokers:

```
zookeeper-server-start.bat %KAFKA_HOME%\config\zookeeper.properties
```

```
kafka-server-start.bat %KAFKA_HOME%\config\server.properties
```

```
kafka-server-start.bat %KAFKA_HOME%\config\server-one.properties
```

```
kafka-server-start.bat %KAFKA_HOME%\config\server-two.properties
```

6.2. Create Topic:

Apache Kafka provides the `admin` API in which we need to start with using `create()` method of `Admin` interface to establish a connection to Kafka cluster and create a topic.

- First, we should create an `Admin` object by providing a set of key-value pairs as configuration properties to `create()` method. It is required to specify at least `bootstrap.servers` property which takes a list of host/port pairs used to establish an initial connection to the Kafka cluster. For the complete list of `admin` properties, refer to Apache Kafka Admin Client Configs Documentation.

  Note: In the `Properties` object, we can set the property `bootstrap.servers` directly or by calling `BOOTSTRAP_SERVERS_CONFIG` variable in `AdminClientConfig` class.

- Once the `Admin` object is created, one or more topics can be created using `createTopics()` method which takes a collection of `NewTopic` objects where we can specify topic name, number of partitions and replication factor for a specific topic. If you omit `partitions` and (or) `replicationFactor` arguments in `NewTopic` object then `topicName` is created with default partitions and replication factor values that are mentioned in `%KAFKA_HOME%\config\server.properties` file.

- `createTopics()` method returns `CreateTopicsResult` object of which `values()` method of which `get()` method is needed to get the result of a specific topic created. This `get()` method returns `KafkaFuture<Void>` object.

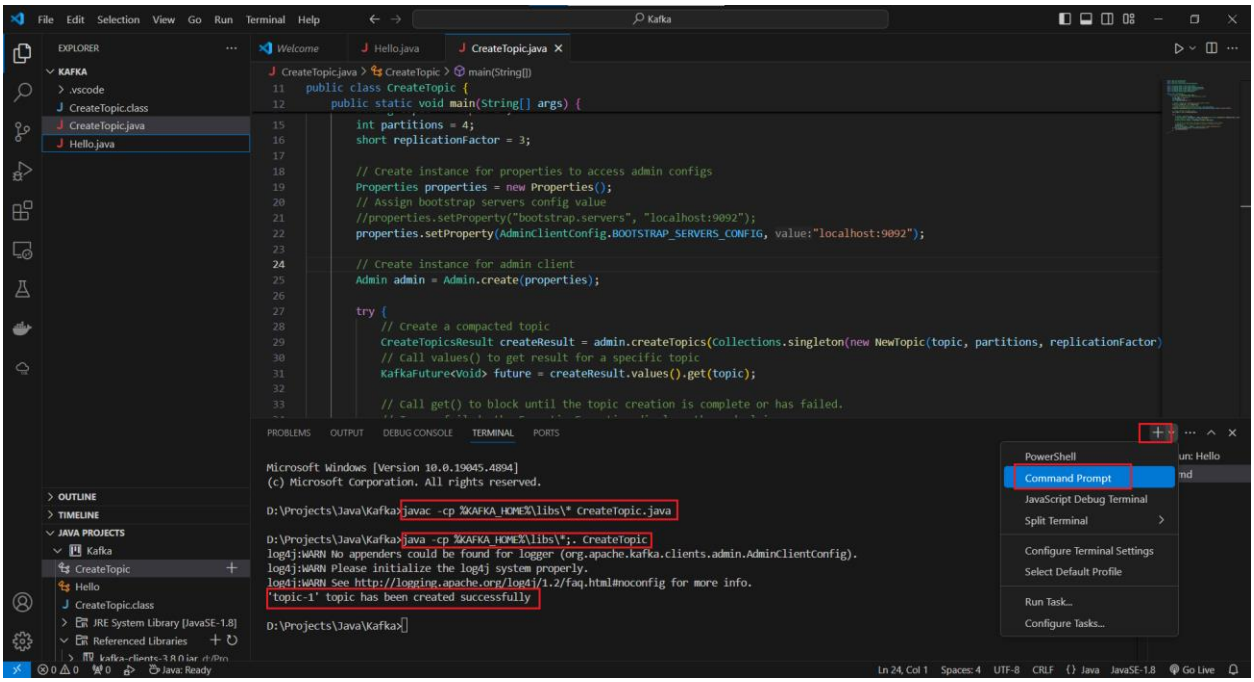
- `KafkaFuture<Void>` object has `get()` method which helps to wait until the topic creation is either completed or failed. This method can throw 2 exceptions - `InterruptedException` and `ExecutionException` - in case of any failure.

In VS Code, create a new filed named `CreateTopic.java` and enter the following code:

```
import java.util.Collections;
import java.util.Properties;
import java.util.concurrent.ExecutionException;

import org.apache.kafka.clients.admin.Admin;
import org.apache.kafka.clients.admin.AdminClientConfig;
import org.apache.kafka.clients.admin.CreateTopicsResult;
import org.apache.kafka.clients.admin.NewTopic;
import org.apache.kafka.common.KafkaFuture;

public class CreateTopic {
    public static void main(String[] args) {
        // Set topic name, partitions and replication factor
        String topic = "topic-1";
        int partitions = 4;
        short replicationFactor = 3;

        // Create instance for properties to access admin configs
        Properties properties = new Properties();
        // Assign bootstrap servers config value
        //properties.setProperty("bootstrap.servers", "localhost:9092");
        properties.setProperty(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");

        // Create instance for admin client
        Admin admin = Admin.create(properties);

        try {
            // Create a compacted topic
            CreateTopicsResult createResult =
admin.createTopics(Collections.singleton(new NewTopic(topic, partitions,
replicationFactor)));
            // Call values() to get result for a specific topic
            KafkaFuture<Void> future = createResult.values().get(topic);

            // Call get() to block until the topic creation is complete or
has failed.
            // Incase failed, the ExecutionException displays the underlying
cause.
            future.get();
            System.out.println("'"+topic+"' topic has been created
successfully");
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

After the above code is entered, click on **+** button on **Terminal** window and select **Command Prompt** where you need to run the following commands to compile the `CreateTopic.java` file and run it.

```
javac -cp %KAFKA_HOME%\libs\* CreateTopic.java
java -cp %KAFKA_HOME%\libs\*;. CreateTopic
```



You can see that a topic name `topic-1` has been created.

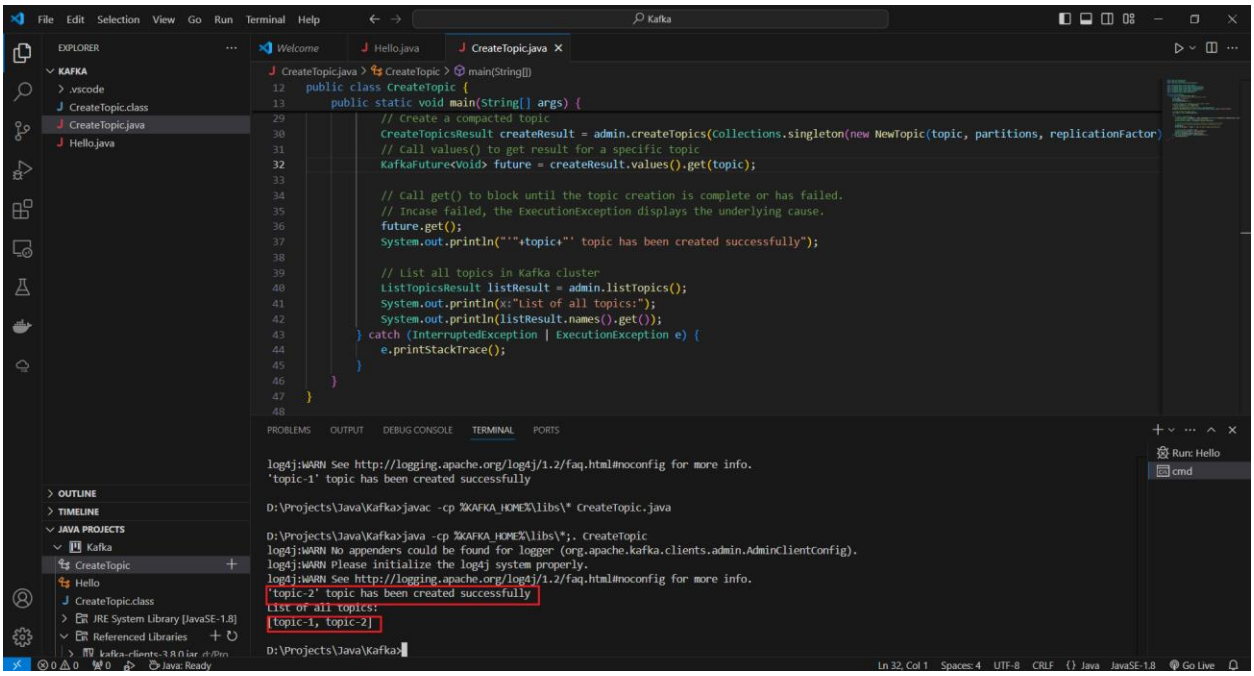
6.2.1. List Topics:

We can also list out the available topics in the cluster using `listTopics()` method of `Admin` object which returns `ListTopicsResult` object of which `names()` method of which `get()` method can be used to retrieve the list of topic names available in Kafka cluster.

In VS Code, enter the following code before `catch()` block in `CreateTopic.java` file. Since `topic-1` is already created, change the topic variable value to `topic-2` to create a new topic and then list out both topics:

```
            // List all topics in Kafka cluster
            ListTopicsResult listResult = admin.listTopics();
            System.out.println("List of all topics:");
            System.out.println(listResult.names().get());
```

Once the above code is entered, compile `CreateTopic.java` file and run it



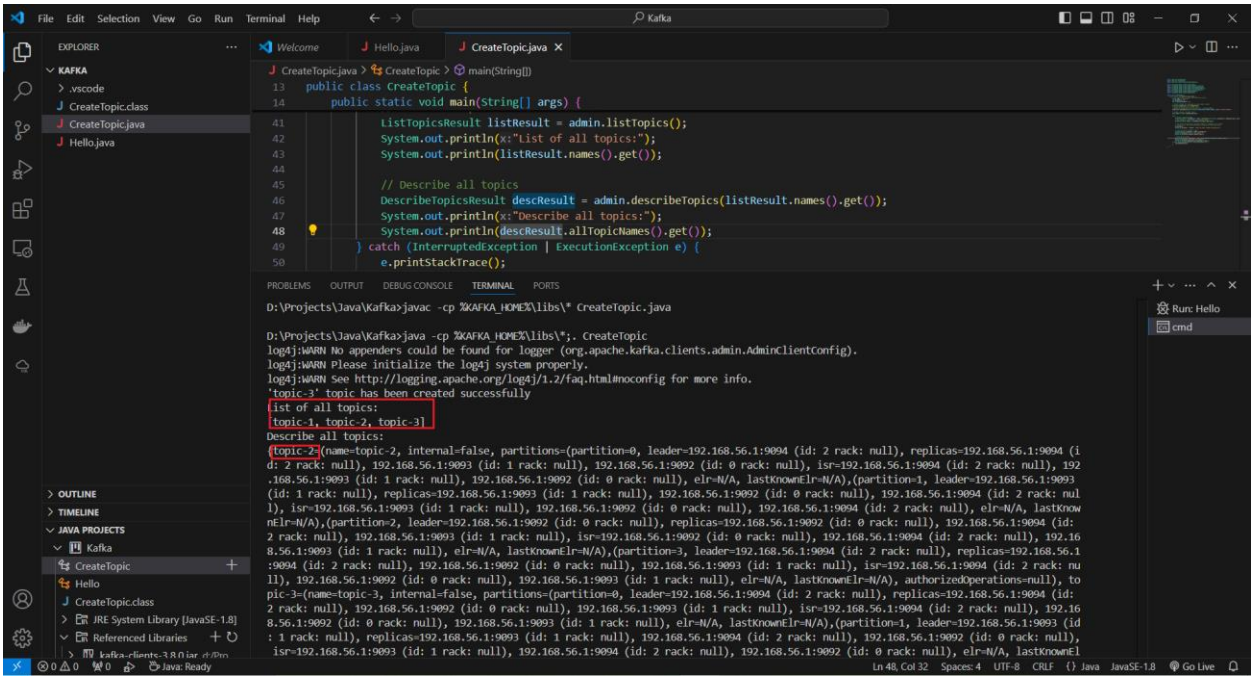
You can see that it created a new topic `topic-2` and listed out both topics in the output window.

6.2.2. Describe Topics:

Similar to `listTopics()`, we can get the topic details using `describeTopics()` method of `Admin` object which returns `DescribeTopicsResult` object of which `values()` method of which `get()` method is needed to get the result of a specific topic created. This `get()` method returns `KafkaFuture<Void>` object of which `allTopicnames()` method of which `get()` method can be used to retrieve the details such of topic name, partitions, leader, replication factor etc.

In VS Code, enter the following code before `catch()` block in `CreateTopic.java` file. Since `topic-2` is already created, change the topic variable value to `topic-3` to create a new topic and provide details of each topic:

```
            // Describe all topics
            DescribeTopicsResult descResult =
admin.describeTopics(listResult.names().get());
            System.out.println("Describe all topics:");
            System.out.println(descResult.allTopicNames().get());
```



You can see that it created a new topic `topic-2` and described all topics in the output window.

6.2.3. Delete Topics:

Similar to `createTopics()`, we can delete one or more topics using `deleteTopics()` method of `Admin` object which returns `DeleteTopicsResult` object which has `topicNameValues()` method of which `get()` method returns `KafkaFuture<Void>` object. The `get()` method of `KafkaFuture<Void>` objects ensures to wait until the topic deletion is either completed or failed and throws 2 exceptions - `InterruptedException` and `ExecutionException` - in case of any failure.

The following code deletes 3 topics in the Kafka cluster:

```
            // Delete multiple topics
            ArrayList<String> topics= new ArrayList<String>();
            topics.add("topic-1");
            topics.add("topic-2");
            topics.add("topic-3");
            DeleteTopicsResult delResult = admin.deleteTopics(topics);
            for (String topicName : topics) {
                KafkaFuture<Void> delFuture =
delResult.topicNameValues().get(topicName);
                // Call get() to block until the topic deletion is complete
or has failed
                delFuture.get();
                System.out.println(topicName + " has been deleted
successfully");
            }
```

Note:
After executing the above code, it appears that topics were deleted, but it marks for deletion without deleting the underlying files in `%KAFA_HOME%\kafka-logs` folder and shuts down all Kafka brokers abruptly. This is a known issue in Windows operating system. To resolve this, we should manually delete all files under `kafka-logs`, `kafka-logs-1`, `kafka-logs-2`, and `zookeeper` folders in `%KAFA_HOME%` location then restart Zookeeper and Kafka brokers services.

6.3. Create Producer:

Now that topics are created in Kafka, let us create a Producer application.

Apache Kafka provides the `producer` API in which we need to start with using `KafkaProducer` class which implements `Producer` class to establish a connection to Kafka cluster and produce data.

- First, we should instantiate the `Producer` class by providing a set of key-value pairs as configuration properties. Some of the required properties are highlighted below:
    - `bootstrap.servers`: A list of host/port pairs (in the form of host1:port1, host2:port2,..) used for establishing initial connection to Kafka cluster.
    - `key.serializer`: A serializer class for key that implements the `org.apache.kafka.common.serialization.StringSerializer` interface.
    - `value.serializer`: A serializer class for value that implements the `org.apache.kafka.common.serialization.StringSerializer` interface.

  Other commonly used properties are `client.id` (unique id for the producer application), `producer.type` (sync or async), `retries` (auto-retry on failure), `batch.size` (batch records), `buffer.memory` (memory to buffer records to be sent to server).
  For the complete list of `producer` properties, refer to Apache Kafka Producer Configs Documentation.

  Note: In the `Properties` object, we can set properties `bootstrap.servers`, `key.serializer` and `value.serializer` directly or by calling `BOOTSTRAP_SERVERS_CONFIG`, `KEY_SERIALIZER_CLASS_CONFIG` and `VALUE_SERIALIZER_CLASS_CONFIG` variables in `ProducerConfig` class.

- The `send()` method of `KafkaProducer` class is used to send a producer record to Kafka. This method takes `ProducerRecord` object and `callback` object (this is optional) as arguments.

- The `ProducerRecord` class manages records to be sent to Kafka for which we need to pass `topic` (topic name) and `value` (data) as mandatory arguments to `ProducerRecord` constructor. Additionally, we can provide `key`, `partition` and `timestamp` values as well to `ProducerRecord`.

- It is important to flush the data sent and close the producer using `flush()` and `close()` methods provided by `producer` object to avoid any resource leaks.

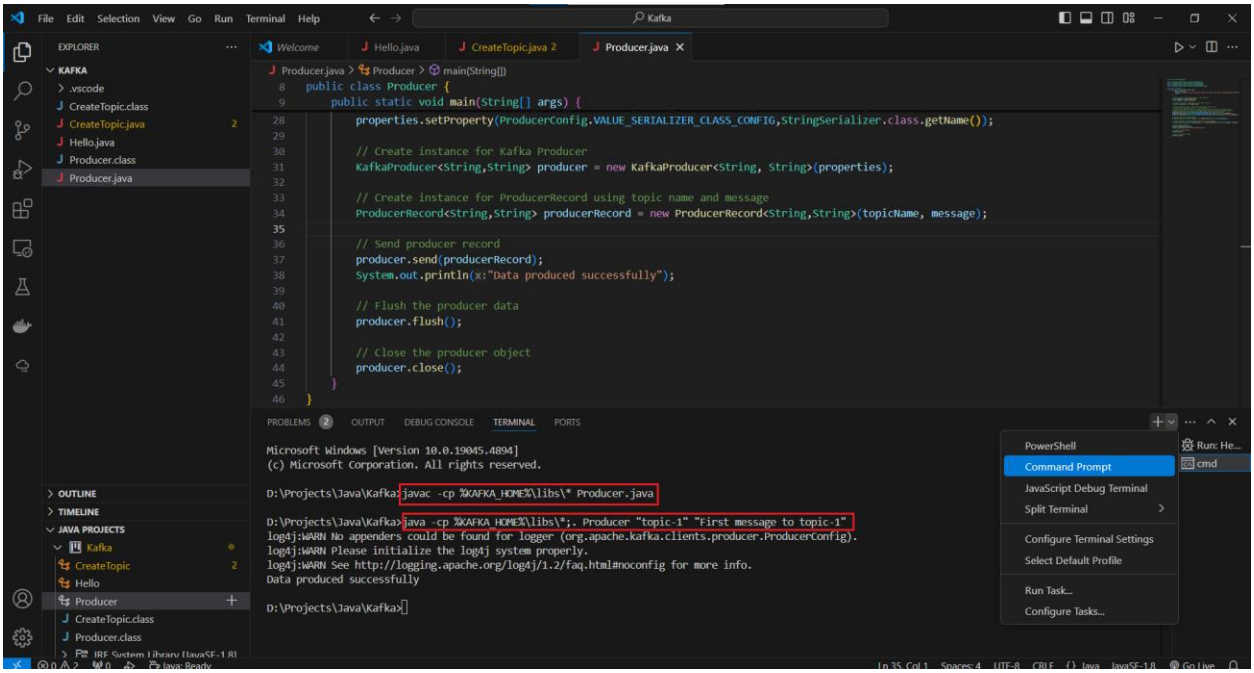
In VS Code, create a new file named `Producer.java` and enter the following code:

```
import java.util.Properties;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;

public class Producer {
    public static void main(String[] args) {
        if (args.length < 2){
            System.err.println("Invalid arguments passed, you need to pass
topic name and message parameters");
            System.exit(1);
        }

        // Read topic name and message passed as input arguments
        String topicName = args[0];
        String message = args[1];

        // Create instance for properties to access admin configs
        Properties properties = new Properties();

        // Assign bootstrap servers, key and value serializer config values
        //properties.put("bootstrap.servers", "localhost:9092");

//properties.put("key.serializer","org.apache.kafka.common.serialization.Ser
ializer");

//properties.put("value.serializer","org.apache.kafka.common.serialization.S
erializer");
        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");

properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,"org.apache.kafka.
common.serialization.Serializer");

properties.setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,"org.apa
che.kafka.common.serialization.Serializer");
```

```
        // Create instance for Kafka Producer
        KafkaProducer<String,String> producer = new KafkaProducer<String,
String>(properties);

        // Create instance for ProducerRecord using topic name and message
        ProducerRecord<String,String> producerRecord = new
ProducerRecord<String,String>(topicName, message);

        // Send producer record
        producer.send(producerRecord);
        System.out.println("Data produced successfully");

        // Flush the producer data
        producer.flush();

        // Close the producer object
        producer.close();
    }
}
```

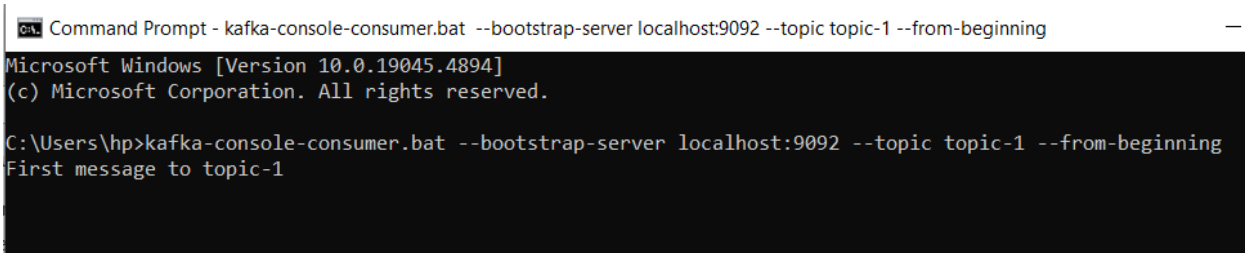
After the above code is entered, click on **+** button on **Terminal** window and select **Command Prompt** where you need to run the following commands to compile the `Producer.java` file and run it.

```
javac -cp %KAFKA_HOME%\libs\* Producer.java
java -cp %KAFKA_HOME%\libs\*;. Producer "topic-1" "First message to
topic-1"
```



You can see that the data has been produced successfully into the topic. To verify the same, open **Command Prompt** and run the following consumer command to see data in `topic-1`.

```
kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic topic-1 --from-beginning
```



In **VS Code**, run the following command on **Terminal** window to produce a second message to `topic-1`.

```
java -cp %KAFKA_HOME%\libs\*;. Producer "topic-1" "Second message to topic-1"
```

You can see this second message in consumer window

6.3.1. Producer Callbacks:

We can do Producer callbacks to understand how a producer sends data to Kafka i.e. if the data was correctly produced, where it was produced, its offset value, partition value, etc. The callback function used by producer is `onCompletion()` which is implemented for asynchronously handling the request completion and is implemented in the block where producer sends data to Kafka.

The `onCompletion()` method expects two arguments – `RecordMetadata` *(metadata of record regarding partition and offset)* and `Exception` *(two possible exceptions – Retirable exception which indicates the message may be sent and Non-retirable exception which indicates message will never be sent)*.

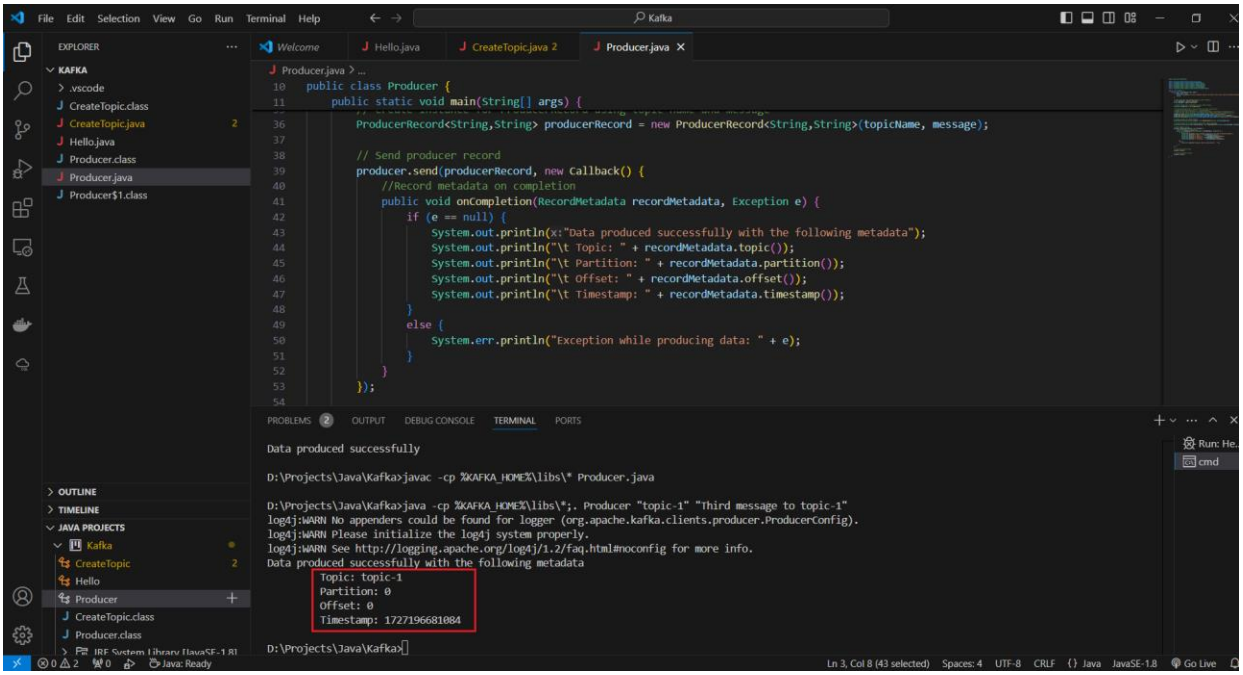
In `Producer.java` file, modify the `producer.send()` line with the below code to display `recordMetadata` output once the record is published into Kafka.

```
        // Send producer record
        producer.send(producerRecord, new Callback() {
            //Record metadata on completion
            public void onCompletion(RecordMetadata recordMetadata,
Exception e) {
                if (e == null) {
                    System.out.println("Data produced successfully with the
following metadata");
                    System.out.println("\t Topic: " +
recordMetadata.topic());
                    System.out.println("\t Partition: " +
recordMetadata.partition());
                    System.out.println("\t Offset: " +
recordMetadata.offset());
                    System.out.println("\t Timestamp: " +
recordMetadata.timestamp());
                }
                else {
                    System.err.println("Exception while producing data: " +
e);
                }
            }
        });
```

Note that the above code will automatically import `org.apache.kafka.clients.producer.Callback` class.

Then compile and run `Producer.java` program with these commands:

```
javac -cp %KAFKA_HOME%\libs\* Producer.java
java -cp %KAFKA_HOME%\libs\*;. Producer "topic-1" "Third message to
topic-1"
```



6.3.2. Produce Data with Same Key:

When a same key is used to produce messages, those will go into same partition.

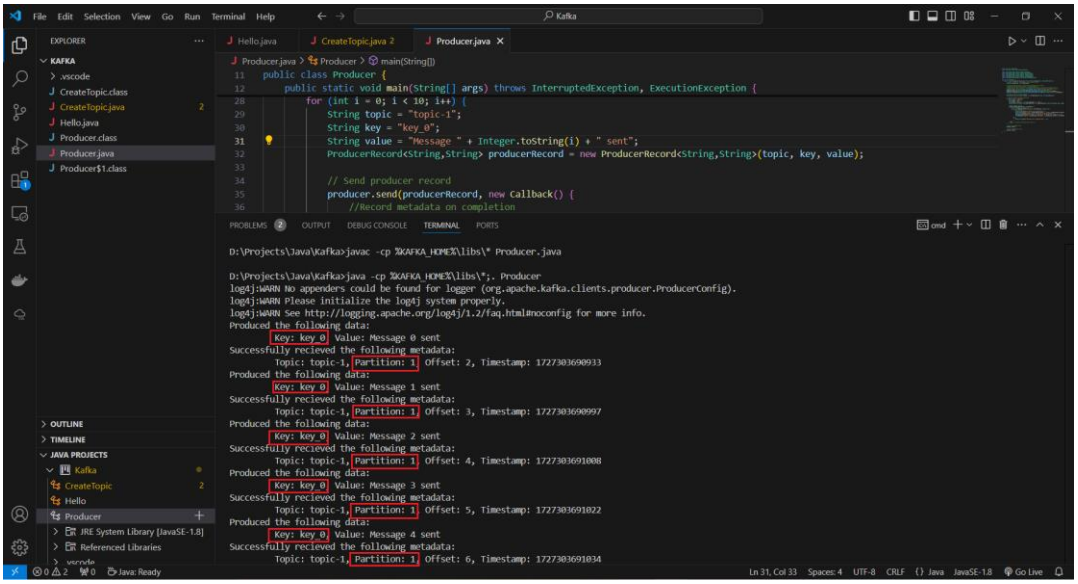
In VS Code, modify `Producer.java` code with the following lines:

```
import java.util.Properties;
import java.util.concurrent.ExecutionException;

import org.apache.kafka.clients.producer.Callback;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.apache.kafka.common.serialization.StringSerializer;

public class Producer {
    public static void main(String[] args) throws InterruptedException,
ExecutionException {
        // Create instance for properties to access admin configs
        Properties properties = new Properties();

        // Assign bootstrap servers, key and value serializer config values
        //properties.put("bootstrap.servers", "localhost:9092");

//properties.put("key.serializer","org.apache.kafka.common.serialization.Str
ingSerializer");

//properties.put("value.serializer","org.apache.kafka.common.serialization.S
tringSerializer");
        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");
```

```
properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,StringSerializer.c
lass.getName());

properties.setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,StringSe
rializer.class.getName());

        // Create instance for Kafka Producer
        KafkaProducer<String,String> producer = new KafkaProducer<String,
String>(properties);

        // Generate producer records iteratively
        for (int i = 0; i < 10; i++) {
            String topic = "topic-1";
            String key = "id_"+ Integer.toString(i);
            String value = "Message " + Integer.toString(i) + " sent";
            ProducerRecord<String,String> producerRecord = new
ProducerRecord<String,String>(topic, key, value);

            // Send producer record
            producer.send(producerRecord, new Callback() {
                //Record metadata on completion
                public void onCompletion(RecordMetadata recordMetadata,
Exception e) {
                    if (e == null) {
                        System.out.println("Produced the following data: ");
                        System.out.println("\t Key: "+producerRecord.key() +
", Value: "+producerRecord.value());
                        System.out.println("Successfully recieved the
following metadata:");
                        System.out.printf("\t Topic: %s, Partition: %d,
Offset: %d, Timestamp: %d \n",
                        recordMetadata.topic(), recordMetadata.partition(),
recordMetadata.offset(), recordMetadata.timestamp());
                    }
                    else {
                        System.err.println("Exception while producing data:
" + e);
                    }
                }
            }).get(); //sending synchronous data forcefully
        }

        // Flush the producer data
        producer.flush();

        // Close the producer object
        producer.close();
    }
}
```

Then compile and run `Producer.java` program with these commands:

```
javac -cp %KAFKA_HOME%\libs\* Producer.java
java -cp %KAFKA_HOME%\libs\*;. Producer
```



Here, you can see that all messages are going into partition 1 since we used the same key named `key_0`.

6.3.3. Produce Data with Different Keys:

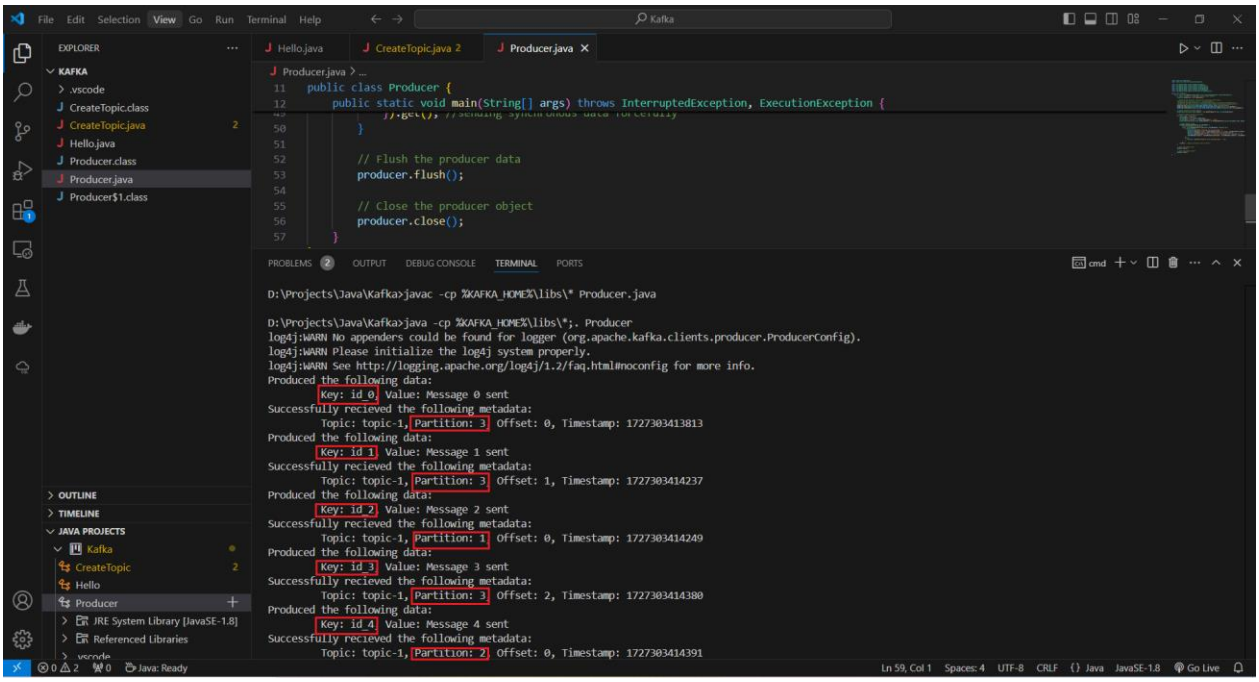
The key in Kafka help to uniquely identify a partition from other partitions. We can send synchronous messages to Kafka forcefully using `get()` method of `ProducerRecord.send()` method.

In VS Code, modify `Producer.java` code with the highlighted line:

```
        // Generate producer records iteratively
        for (int i = 0; i < 10; i++) {
            String topic = "topic-1";
            String key = "id_"+ Integer.toString(i);
            String value = "Message " + Integer.toString(i) + " sent";
            ProducerRecord<String,String> producerRecord = new
ProducerRecord<String,String>(topic, key, value);
```

Then compile and run `Producer.java` program with these commands:

```
javac -cp %KAFKA_HOME%\libs\* Producer.java
java -cp %KAFKA_HOME%\libs\*;. Producer
```



Here, you can see that records with keys id_0 and id_1 are sent to Partition 3, id_2 sent to partition 1, id_3 sent to partition3 and id_4 sent to partition 2.

6.4. Create Consumer:

Until now, we create producer to send messages to Kafka cluster. Now, let us create a consumer to consume messages from Kafka cluster.

Apache Kafka provides the `consumer` API in which we need to start with using `KafkaConsumer` class which implements `Consumer` class to establish a connection to Kafka cluster and consume data.

- First, we should instantiate the `Consumer` class by providing a set of key-value pairs as configuration properties. Some of the required properties are highlighted below:
    - `bootstrap.servers`: A list of host/port pairs (in the form of host1:port1, host2:port2,..) used for establishing initial connection to Kafka cluster.
    - `key.deserializer`: A serializer class for key that implements the `org.apache.kafka.common.serialization.StringDeserializer` interface.
    - `value.deserializer`: A serializer class for value that implements the `org.apache.kafka.common.serialization.StringDeserializer` interface.

    Other commonly used properties are:

- `group.id`: A unique identifier for a consumer group. This property is needed when the consumer uses group management functionality which is used to achieve the load balancing of consuming records from a Kafka topic *(i.e balancing topic partitions between all consumers in Consumer group so that one partition is assigned to exactly one consumer in a group)*
- `enable.auto.commit`: If set to true, then it commits offset value automatically to Zookepeer, otherwise not committed. By default, this value is set to `true`.
- `auto.commit.interval.ms`: The frequency of auto committing consumed offsets to Zookeeper. The default value is 5000 (i.e. 5 seconds).
- `auto.offset.reset`: This property is needed when no initial offset is present of current offset does not exist on server. It can have values – `earliest` *(set to earliest offset to read data from the beginning in a partition)*, `latest` *(set to latest offset to read data from the last committed offset value)*, `none` *(throws error when no previous offset found).* By default, it is set to `latest` value.

For the complete list of `consumer` properties, refer to Apache Kafka Consumer Configs Documentation.

Note: In the `Properties` object, we can set properties `bootstrap.servers`, `key.deserializer` and `value.deserializer` directly or by calling the respective `BOOTSTRAP_SERVERS_CONFIG`, `KEY_DESERIALIZER_CLASS_CONFIG` and `VALUE_DESERIALIZER_CLASS_CONFIG` variables in `ConsumerConfig` class.

- The `KafkaConsumer` class provides `subscribe()` method to subscribe to a given topic in Kafka cluster.

- The `poll()` method of `KafkaConsumer` class allows to fetch data from topic partitions. This returns error if topics are not subscribed before polling.

- It is important to close the consumer using `close()` methods provided by `consumer` object to avoid any resource leaks.

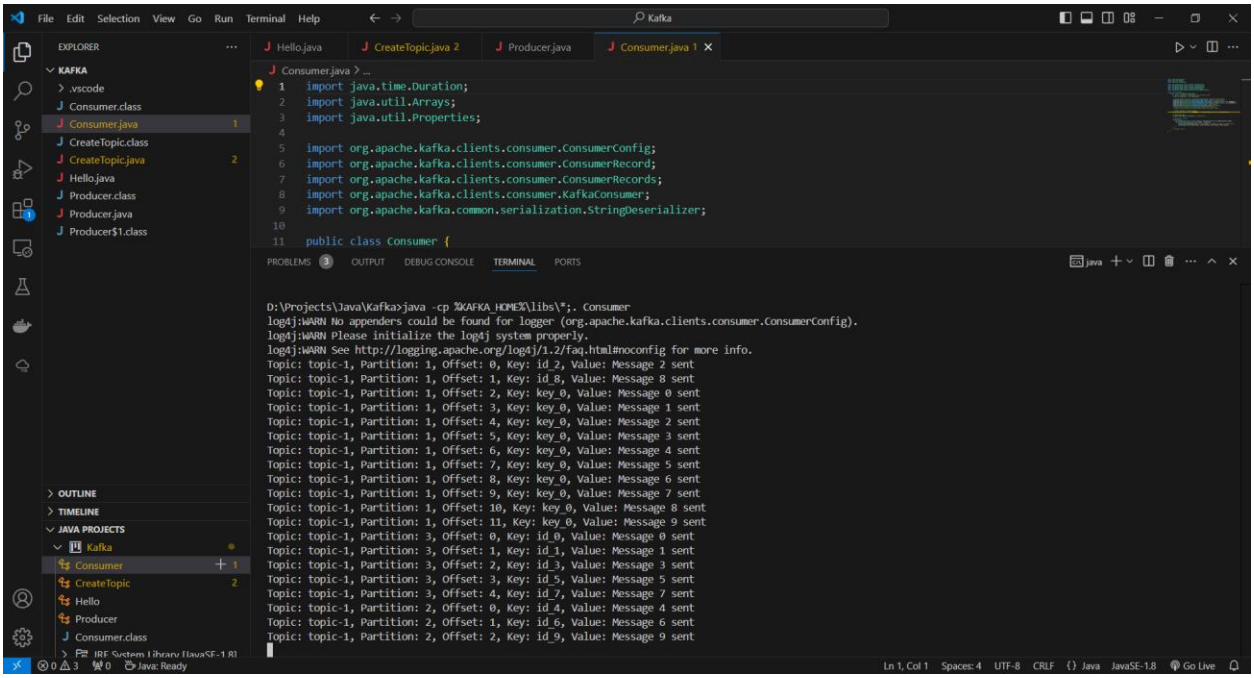
In VS Code, create a new filed named `Consumer.java` and enter the following code:

```
import java.time.Duration;
import java.util.Arrays;
import java.util.Properties;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
```

```
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;

public class Consumer {
    public static void main(String[] args) {
        // Create instance for properties to access admin configs
        Properties properties = new Properties();

        // Assign bootstrap servers, key and value deserializer and other
config values
        properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");
        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());

properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,StringDeserial
izer.class.getName());
        properties.put(ConsumerConfig.GROUP_ID_CONFIG,"consumer-grp1");
        properties.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,"earliest");
        //properties.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,"true");

//properties.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG,"1000");

        // Create instance for Kafka Consumer
        KafkaConsumer<String, String> consumer = new KafkaConsumer<String,
String>(properties);

        // Subscribe topic
        consumer.subscribe(Arrays.asList("topic-1"));

        // Polling for records
        while(true) {
            ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(100));
            for (ConsumerRecord<String,String> record : records) {
                System.out.printf("Topic: %s, Partition: %d, Offset: %d,
Key: %s, Value: %s\n",

record.topic(),record.partition(),record.offset(),record.key(),record.value(
));
            }
        }
        //consumer.close();
    }
}
```

After the above code is entered, click on **+** button on **Terminal** window and select **Command Prompt** where you need to run the following commands to compile the `Consumer.java` file and run it.

```
javac -cp %KAFKA_HOME%\libs\* Consumer.java
java -cp %KAFKA_HOME%\libs\*;. Consumer
```



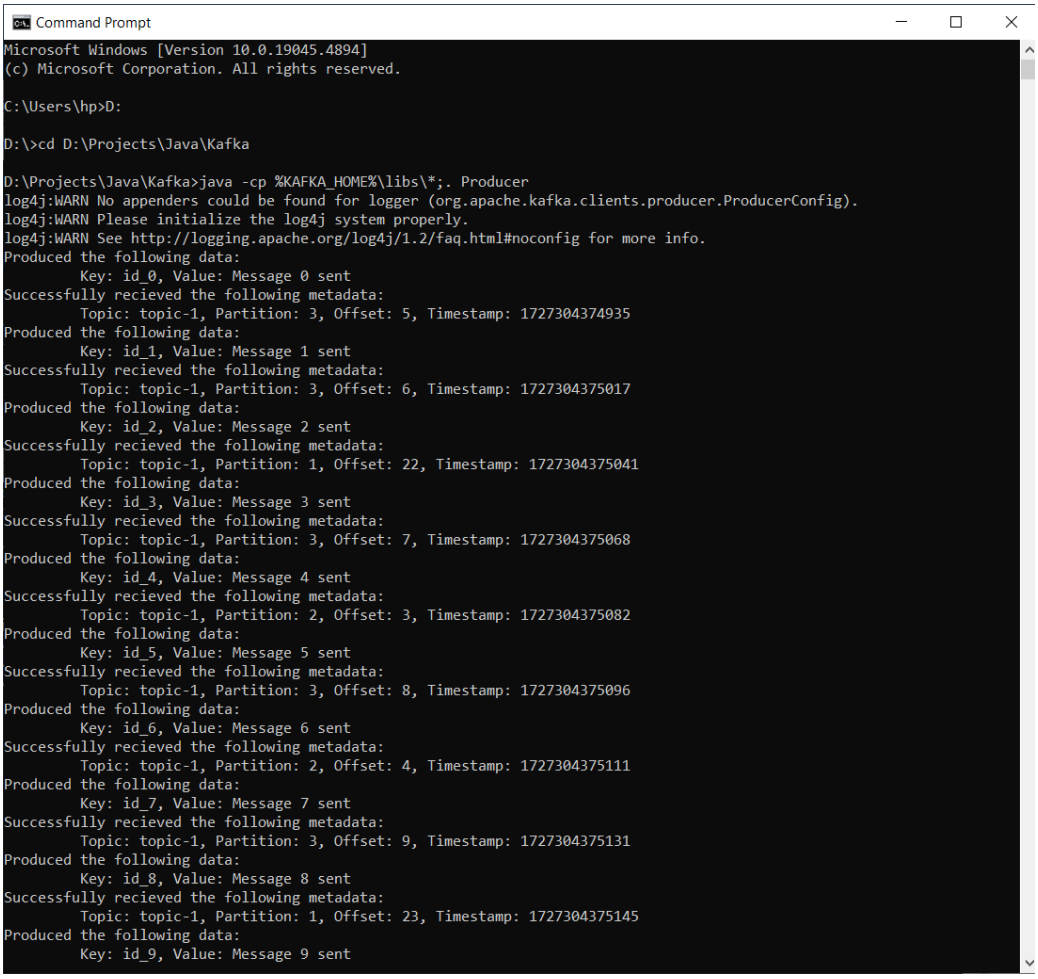
You can see that it keeps polling the topic `topic-1` every 100 milliseconds (0.1 second) and displays data fetched from Kafka. Note that although we set `auto.offset.reset` configuration to `earliest`, it works as `latest` and consumes the latest data produced to topic after the consumer was run (somehow, `earliest` offset is not working here).
Enter **Ctrl + c** to exit out of consumer application.

To take the advantage of Consumer group functionality, open two command prompts and navigate to the location where `Consumer.java` file exists and run it:

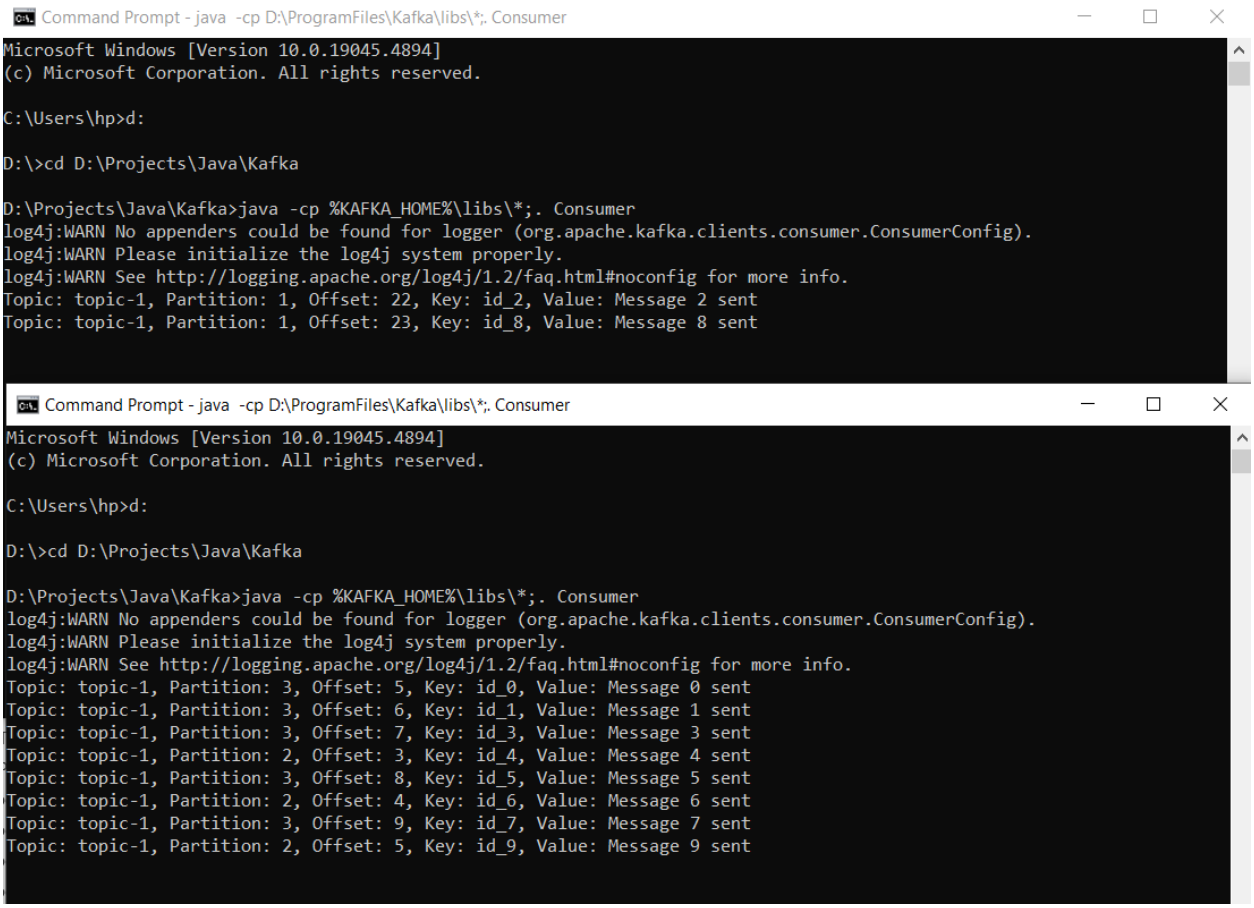
```
D:
cd D:\Projects\Java\Kafka
java -cp %KAFKA_HOME%\libs\*;. Consumer
```

Open another command prompt and navigate to the location where `Producer.java` file exists and run it to produce some data into Kafka topic:

```
D:
cd D:\Projects\Java\Kafka
java -cp %KAFKA_HOME%\libs\*;. Producer
```



On the producer terminal, you can see that it has produced 2 messages into partition 1, 3 messages into partition 2 and 5 messages into partition 3.



On the first consumer terminal, you can see that it read 2 messages from partition 1 and on the second consumer terminal, you can see that it read 8 messages from partitions 2 and 3.

7. Spark Streaming with Kafka:

Now, we will create a simple application in Java using Spark and integrate with Kafka to consume messages from a topic and display the frequency of words in messages.

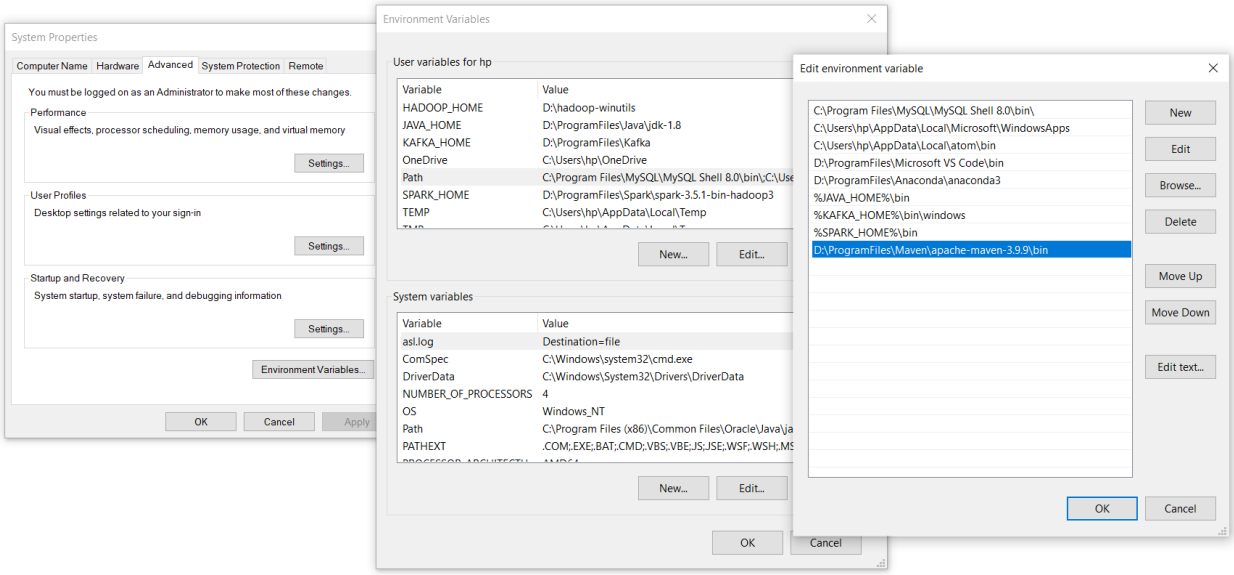
If you do not have Spark installed already, I would suggest you to install it by referring these steps.

To create this project, we need an **IDE tool** such as VS Code, a **build tool** such as Apache Maven and **JDK 1.8** or higher.

Apache Maven is a project management and build tool which is designed based on the concept of project object model (POM). Maven provides developers to manage builds, documentation, reporting, dependencies, SCMs, releases, repositories, distribution, mailing list etc.

Three important terminologies that we should be aware of while using Maven is **groupId** *(uniquely identify the project or application group)*, **artifactId** *(name of the project file to define the artifact's location within the repository)* and **Version** *(version number of the project or application created by user)*.

If you do not have Maven installed already, then install the latest `apache-maven-x.x.x-bin.zip` file from the official Apache Maven website and set the maven install `bin` location in the `PATH` environment variable as shown below:

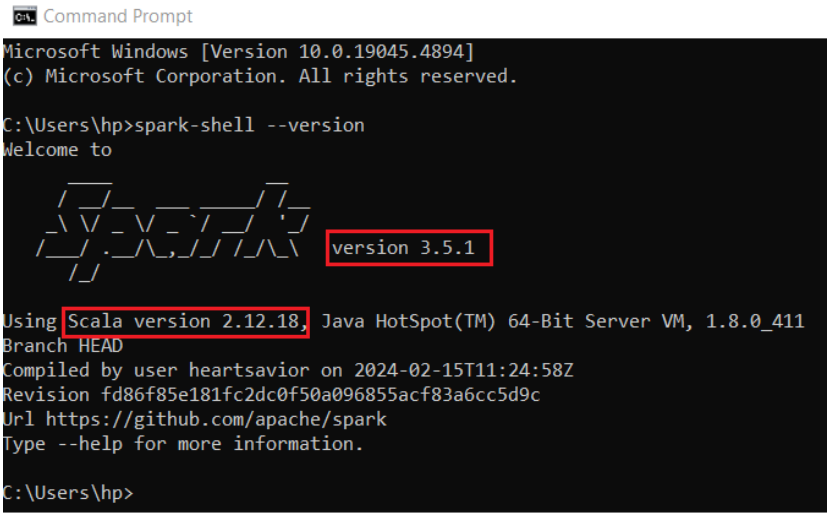


7.1. Verify Spark Version:

It is important to know the installed spark version so that the respective version of Spark Streaming API can be used to create our application.

Open **Command Prompt** and run the following command:

```
spark-shell --version
```



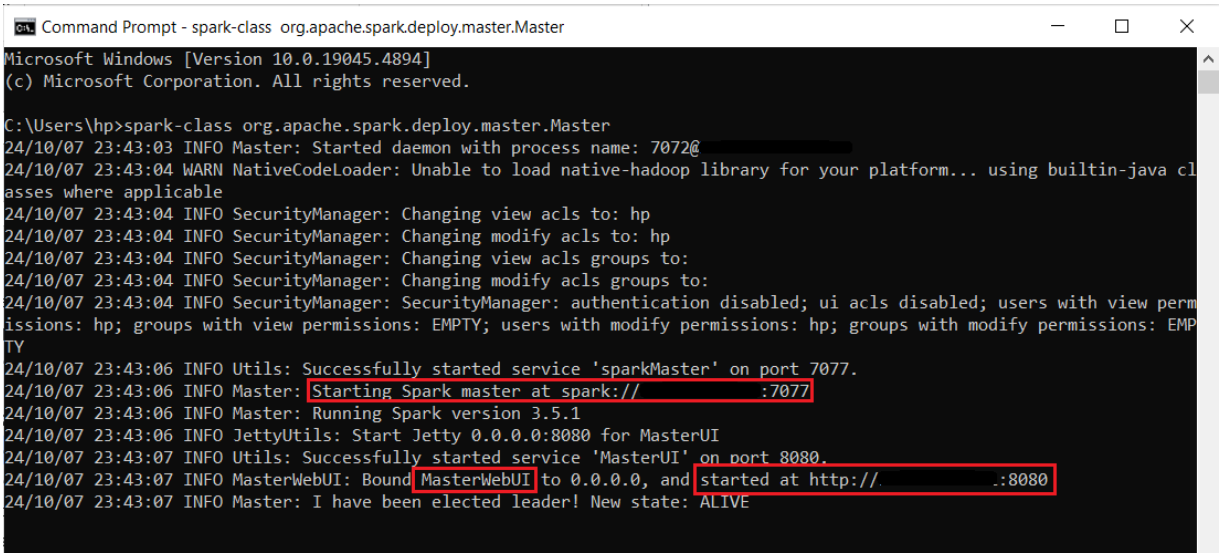
You can see that **Spark version 3.5.1** with **Scala version 2.12.18** was installed.

7.2. Start Spark Services:

Next, start the Spark Master and Worker services to access the Spark cluster.

Open a new **Command Prompt** and run the following command to start the Spark Master service:

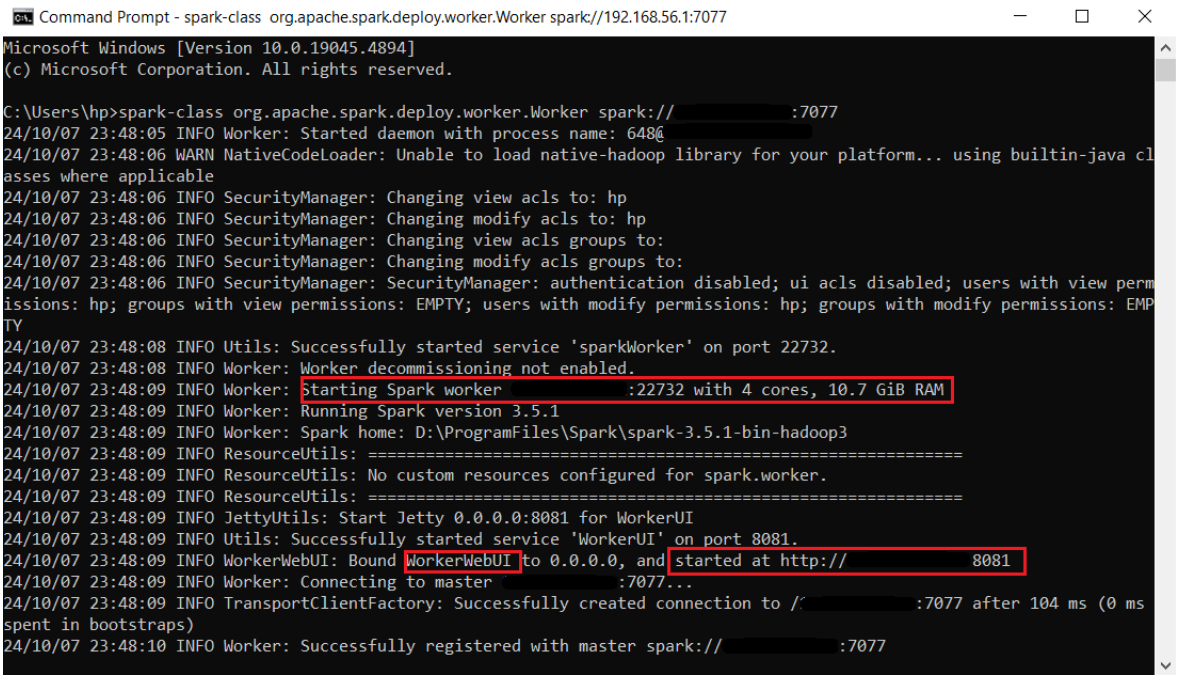
```
spark-class org.apache.spark.deploy.master.Master
```



Here, we can see that Spark master is started at spark://<ipaddress>:7077 i.e listening on default port 7077 and **MasterWebUI** is started at http://localhost:8080/ by default.

Now, start the Spark Worker service using the below command in a new **Command Prompt** *(Make sure that you replace* `<ipaddress>` *with IP address mentioned in your Spark Master URL)*

```
spark-class org.apache.spark.deploy.worker.Worker
spark://<ipaddress>:7077
```

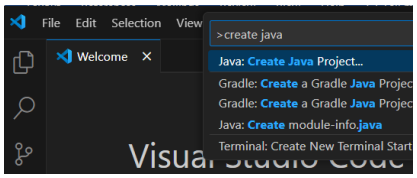


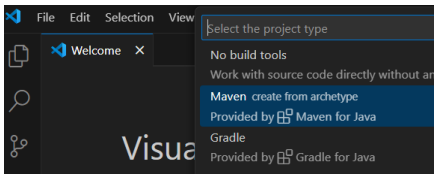
Here, we can see that Spark worker is started at `<ipaddress>:22732` (on a random port) with available number of cores and RAM running on the machine. **WorkerWebUI** is started at http://localhost:8081/ by default.

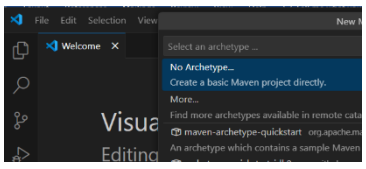
7.3. Create Maven Java Project:

Now, let us create a Maven java project in VS Code application.

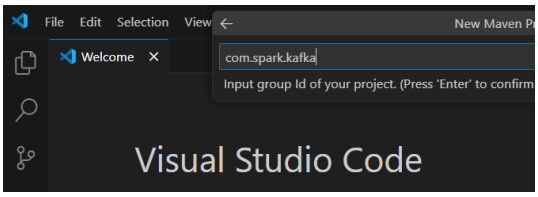
In **VS Code**, open **Command Palette** (in **View** menu or press **Ctrl + Shift + P**) where you can type **create java** and choose for **Java: Create Java Project** option. Then choose **Maven** build tool and select **No Archetype** to create a basic maven project.

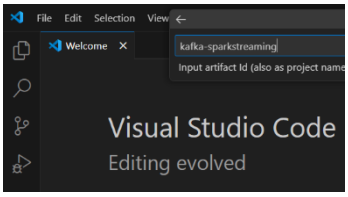






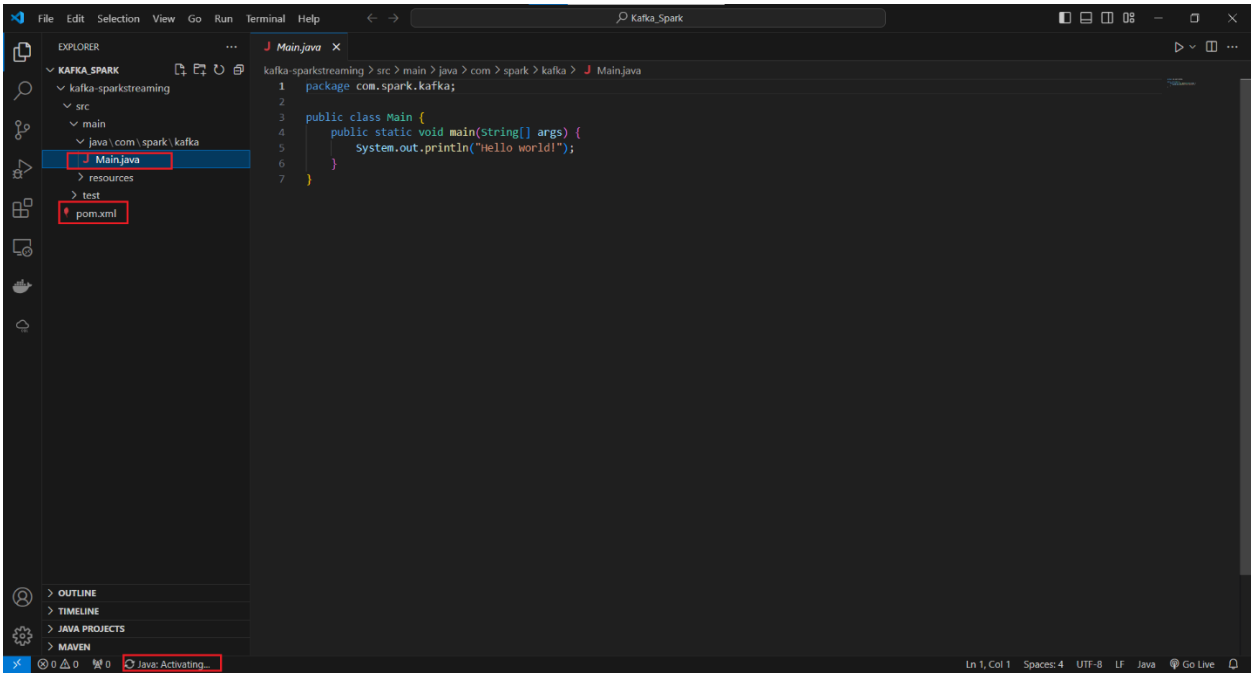
Then it asks you to enter group id where you can enter `com.spark.kafka` following that it asks you to enter the artifact id where you can enter `kafka-sparkstreaming`.



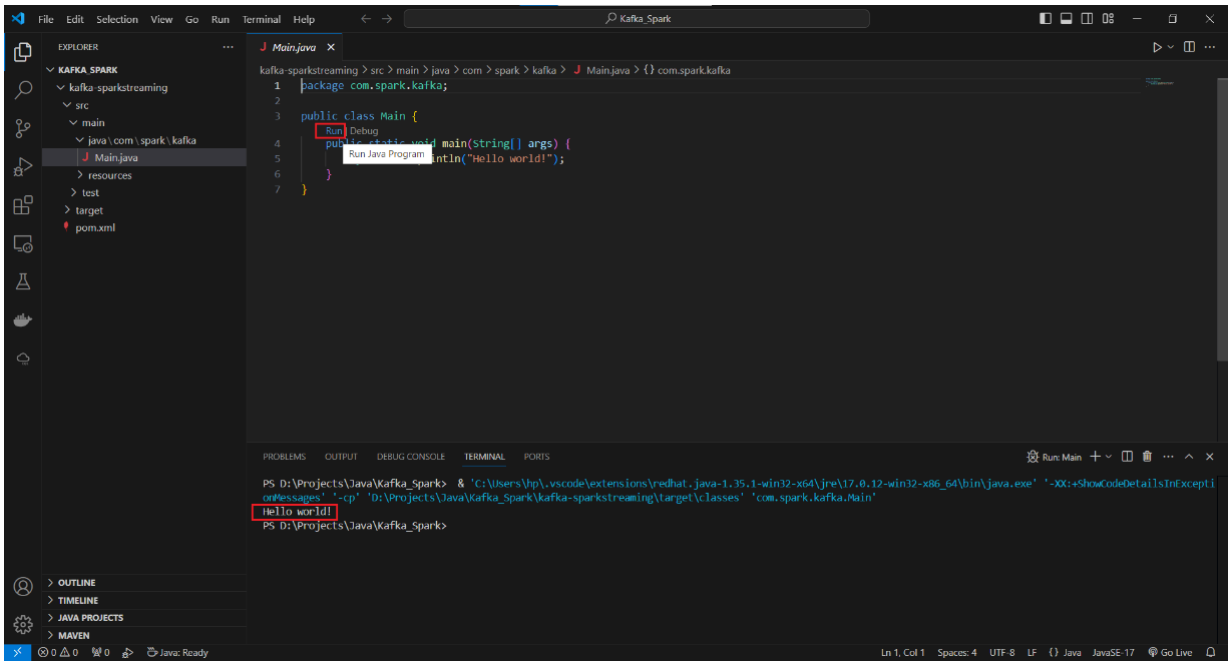


Then, it asks you to choose the destination location where you want this project to be created *(I am choosing `D:\Projects\Java\Kafka_Spark` as my location)* and open that destination folder.

When the destination folder is opened, you can see that java extension got activated and a `Main.java` file is available by default under **kafka-sparkstreaming -> src -> main -> java\com\spark\kafka** folder and a `pom.xml` file is created under `kafka-streaming` folder.



Once the Java extension is ready, click **Run** link in `Main.java` file to see the output of java program on the **Terminal.**



Now that our sample Java code is working, we will write java program to read data from Kafka topic and process through Spark. The integration between Kafka and Spark can be established using the most popular spark libraries `spark-streaming` and `spark-streaming-kafka-0-10`.

Go to `pom.xml` under `kafka-streaming` folder in VS Code and add dependencies to pull `spark-streaming` and `spark-streaming-kafka-0-10` from Maven repository. We should also add `maven-jar-plugin` to create our application jar file. Also, make sure `maven.compiler.source` and `maven.compiler.target` properties are set to **1.8** version in pom file since we are using JDK 1.8 compiler.

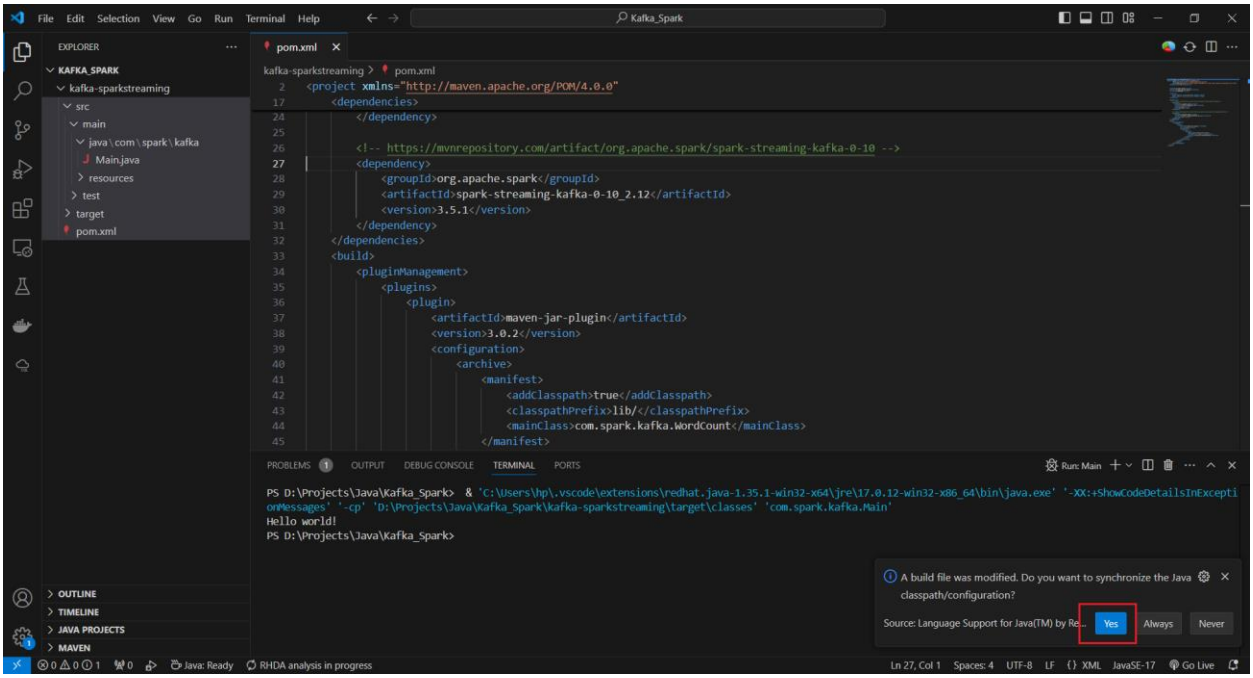
The overall `pom.xml` should look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>


    <groupId>com.spark.kafka</groupId>
    <artifactId>kafka-sparkstreaming</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
```

```
    </properties>

    <dependencies>
        <!-- https://mvnrepository.com/artifact/org.apache.spark/spark-
streaming -->
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-streaming_2.12</artifactId>
            <version>3.5.1</version>
            <scope>provided</scope>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.apache.spark/spark-
streaming-kafka-0-10 -->
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-streaming-kafka-0-10_2.12</artifactId>
            <version>3.5.1</version>
        </dependency>
    </dependencies>
    <build>
        <pluginManagement>
            <plugins>
                <plugin>
                    <artifactId>maven-jar-plugin</artifactId>
                    <version>3.0.2</version>
                    <configuration>
                        <archive>
                            <manifest>
                                <addClasspath>true</addClasspath>
                                <classpathPrefix>lib/</classpathPrefix>

<mainClass>com.spark.kafka.WordCount</mainClass>
                            </manifest>
                        </archive>
                    </configuration>
                </plugin>
            </plugins>
        </pluginManagement>
    </build>
</project>
```

Note that I am using `spark-streaming_2.12` and `spark-streaming-kafka-0-10_2.12` dependencies of Spark 3.5.1 in the above POM since I have **Spark 3.5.1** with **Scala 2.12** version installed in my machine. If you have different Spark version installed, then update the version accordingly in `pom.xml` file.

Once you save the POM file, it asks you to sync the Java class path and click **Yes** to confirm.



Before proceeding further, make sure Kafka services are up and running. If not running already, open 4 **Command Prompts** in **Administrator** mode and run the following commands to start Zookeeper first and then 3 Kafka brokers:

```
zookeeper-server-start.bat %KAFKA_HOME%\config\zookeeper.properties
```

```
kafka-server-start.bat %KAFKA_HOME%\config\server.properties
```

```
kafka-server-start.bat %KAFKA_HOME%\config\server-one.properties
```

```
kafka-server-start.bat %KAFKA_HOME%\config\server-two.properties
```

7.4. Create Spark Application:

Now, we create a simple Spark application that consumes live streaming data from a Kafka topic and displays count of each word in the consumed messages.

Apache Spark Streaming API enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Using this API, we can ingest live data from many sources such as Kafka, Twitter, Flume etc., process it using complex algorithms and sync the processed data into many systems such as file systems, databases, live dashboards etc.

We begin with **SparkConf** API that allows us to set configuration properties as key-value pairs for a Spark application.

- First, we should create a `SparkConf` object and set our application name using `setAppName()` method. This helps us to track our application status on the Spark Master. SparkConf API provides other methods such as `setMaster()` to set Spark master URL, `set()`, `get()`, etc. To know more details of Spark properties, read Apache Spark Properties Documentation.

- Next, create a `JavaStreamingContext` object which is the main entry point for Spark Streaming functionality. While creating this object, we need to specify `SparkConf` and `Duration` parameters. This duration tells Spark for how much time it has to fetch the incoming data so that Spark divides the incoming live data into batches of given duration, processes it and generates the final result in batches. For example, if the given `Duration` is 30 seconds, then it would fetch data from source for every 30 seconds.

  Note: `JavaStreamingContext` is a Java friendly version of `StreamingContext`.

- Next, create `kafkaParams` map object and set configuration properties for Kafka as key-value pairs. We need to specify properties such as `bootstrap.servers`, `key.deserializer`, `value.deserializer`, `group.id`, `auto.offset.reset`, `enable.auto.commit` for our Kafka cluster.

- Now, we will use `KafkaUtils` API to connect to Kafka cluster through Spark Streaming. You can find this API in `spark-streaming-kafka-0-10` library. `KafkaUtils` API has `CreateDirectStream()` method which is used to create an input stream that pulls messages from Kafka topics. This method has different signatures with different types of arguments but we use a signature of passing `JavaStreamingContext`, `LocationStrategy` and `ConsumerStrategy` as parameters to the method and it returns `ConsumerRecord` as `JavaInputDStream`.
  - `LocationStrategy` is a way of distributing processing of Kafka input stream of messages across Spark executors to achieve optimum performance. The most preferred `LocationStrategy` is `PreferConsistent` which distributes Kafka partitions evenly across available Spark executions. When your Spark executors are located on the same hosts as your Kafka brokers, then choose `PreferBrokers` strategy. When you notice a significant skew in load or load is uneven across partitions, then use `PreferFixed` strategy which allows to map specific topic partition to specific host. Note that `PreferConsistent`,

`PreferBrokers, PreferFixed` are available as methods in `LocationStrategies` factory object.

- `ConsumerStrategy` allows users how to configure Kafka consumers in Spark Streaming. Spark provides three consumer strategies - `Subscribe` which allows to subscribe to a fixed collection of topics, `SubscribePattern` which allows to use regex to specify topics of interest and `Assign` which allows to specify a fixed collection of topic partitions. Note that `ConsumerStrategies` factory object provides `Subscribe`, `SubscribePattern` and `Assign` as methods to use. We will use `ConsumerStrategies.Subscribe()` method which requires us to pass `topics` collection and `kafkaParams` arguments.

- Next, we perform a series of operations on `JavaInputDStream` to obtain word frequencies in the messages.
  - Using `map()` function, call `value()` method of `JavaInputDStream ConsumerRecord` to retrieve the actual message in a Kafka record *(since Kafka topic stores messages in key-value pairs)* and return `JavaDStream` messages.
  - Using `flatmap()` function, split each message line in the above `JavaDStream` by a space which returns an array of `JavaDStream` words.
  - Using `mapToPair()` function, map the each word in the above `JavaDStream` to a tuple containing a word and 1 which returns `JavaPairDStream` object.
  - Finally, apply the `reduceByKey()` function on the above `JavaPairDStream` using lambda expression that counts by each word and returns `JavaPairDStream` object.

- The processed `JavaPairDStream` contains collection of elements as `javaPairRDDs` in batches. We can iterate over this `JavaPairDstream` to display each word and its number of occurrences.

- Since this is a streaming application, we need to keep this running by calling `start()` and `awaitTermination()` methods of `JavaStreamingContext`.

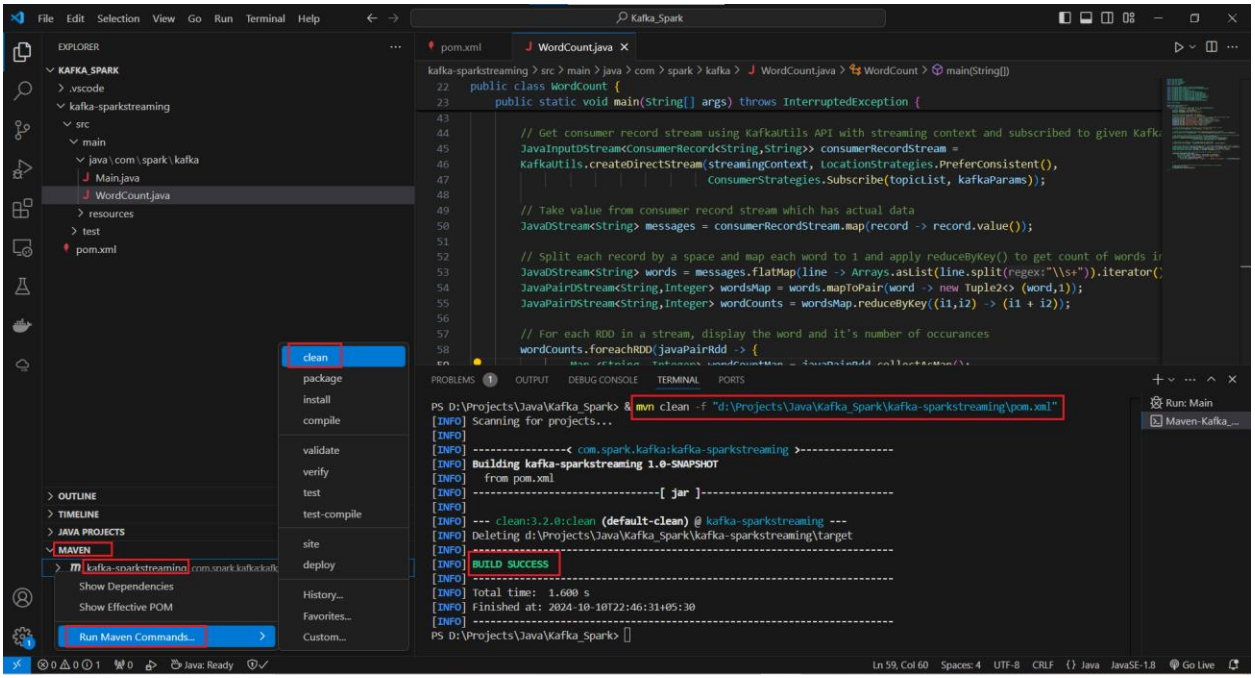
In VS Code, create a new file named `WordCount.java` and enter the following code:

```
package com.spark.kafka;
import java.util.Arrays;
import java.util.Collection;
import java.util.HashMap;
import java.util.Map;

import org.apache.kafka.clients.consumer.ConsumerRecord;
```

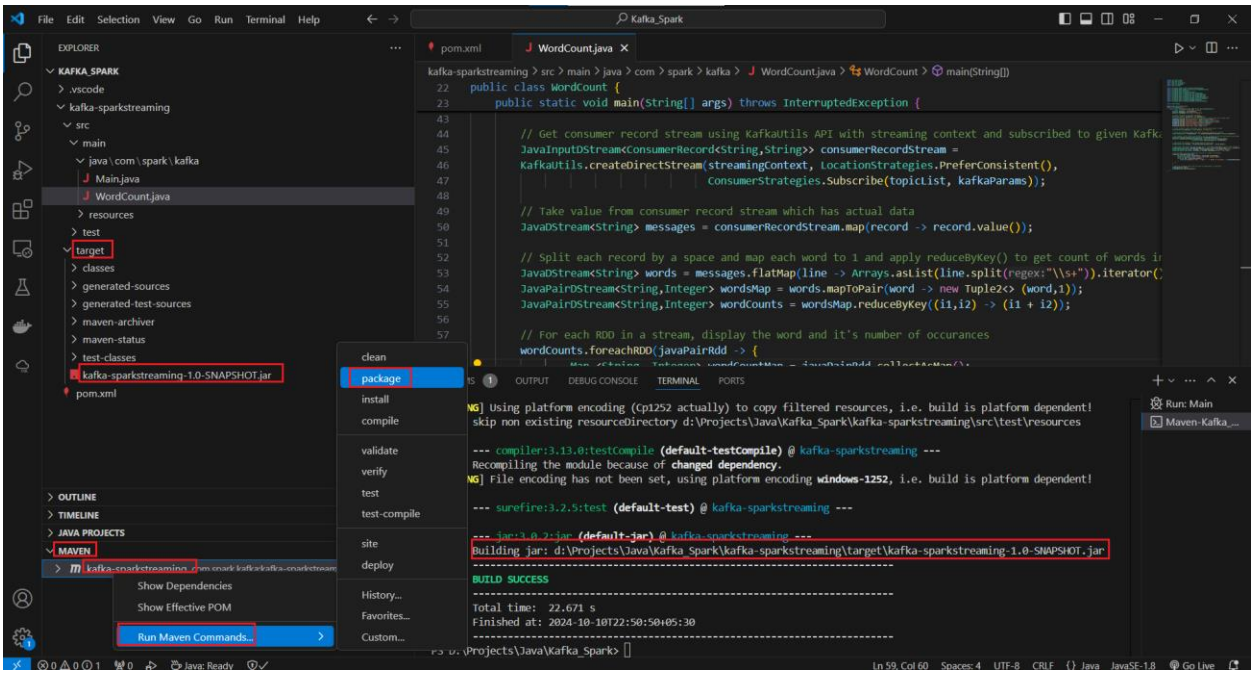
```
import org.apache.kafka.common.serialization.StringDeserializer;
import org.apache.spark.SparkConf;
import org.apache.spark.streaming.Durations;
import org.apache.spark.streaming.api.java.JavaDStream;
import org.apache.spark.streaming.api.java.JavaInputDStream;
import org.apache.spark.streaming.api.java.JavaPairDStream;
import org.apache.spark.streaming.api.java.JavaStreamingContext;
import org.apache.spark.streaming.kafka010.ConsumerStrategies;
import org.apache.spark.streaming.kafka010.KafkaUtils;
import org.apache.spark.streaming.kafka010.LocationStrategies;

import scala.Tuple2;

@SuppressWarnings("deprecation")
public class WordCount {
    public static void main(String[] args) throws InterruptedException {
        // Create instance to access Spark streaming
        SparkConf sparkConf = new SparkConf();
        sparkConf.setAppName("Kafka Spark Streaming");

        // Create map object to set kafka parameters
        Map<String, Object> kafkaParams = new HashMap<>();
        // Assign bootstrap servers, key and value serializers, group id and
other config values
        kafkaParams.put("bootstrap.servers", "localhost:9092");
        kafkaParams.put("key.deserializer", StringDeserializer.class);
        kafkaParams.put("value.deserializer",StringDeserializer.class);
        kafkaParams.put("group.id","word-count-streaming");
        kafkaParams.put("auto.offset.reset","earliest");
        kafkaParams.put("enable.auto.commit",false);

        // Create collection object and assign list of existing Kafka topics
to read
        Collection<String> topicList = Arrays.asList("topic-1");

        // Create java streaming context using spark config with duration of
2 seconds
        JavaStreamingContext streamingContext = new
JavaStreamingContext(sparkConf, Durations.seconds(30));

        // Get consumer record stream using KafkaUtils API with streaming
context and subscribed to given Kafka topics
        JavaInputDStream<ConsumerRecord<String,String>> consumerRecordStream
=
        KafkaUtils.createDirectStream(streamingContext,
LocationStrategies.PreferConsistent(),

ConsumerStrategies.Subscribe(topicList, kafkaParams));

        // Take value from consumer record stream which has actual data
        JavaDStream<String> messages = consumerRecordStream.map(record ->
record.value());

        // Split each record by a space and map each word to 1 and apply
reduceByKey() to get count of words in a stream
        JavaDStream<String> words = messages.flatMap(line ->
Arrays.asList(line.split("\\s+")).iterator());
```

```
        JavaPairDStream<String,Integer> wordsMap = words.mapToPair(word ->
new Tuple2<> (word,1));
        JavaPairDStream<String,Integer> wordCounts =
wordsMap.reduceByKey((i1,i2) -> (i1 + i2));

        // For each RDD in a stream, display the word and it's number of
occurances
        wordCounts.foreachRDD(javaPairRdd -> {
                Map <String, Integer> wordCountMap =
javaPairRdd.collectAsMap();
                System.out.println("Count of words from latest consumer
read:");
                for(String key: wordCountMap.keySet()) {
                    System.out.println("\t Word: " + key + ", " + "Number of
occurances: " + wordCountMap.get(key));
                }
            }
        );

        // Start the streaming process and wait until the application is
terminated
        streamingContext.start();
        streamingContext.awaitTermination();
    }
}
```

After the above code is entered, expand **Maven**, right-click on `kafka-sparkstreaming` artifact and select **Run Maven Commands** and choose **clean**. This executes `mvn clean` command on the terminal and displays **BUILD SUCCESS** message.



Next, right-click on `kafka-sparkstreaming` artifact under **Maven** section and select **Run Maven Commands** and choose **package**. This executes `mvn package` command on the terminal and builds a jar file called `kafka-sparkstreaming-1.0-SNAPSHOT.jar` which you can see under target folder in `kafka-sparkstreaming` folder.



7.5. Execute Spark Application:

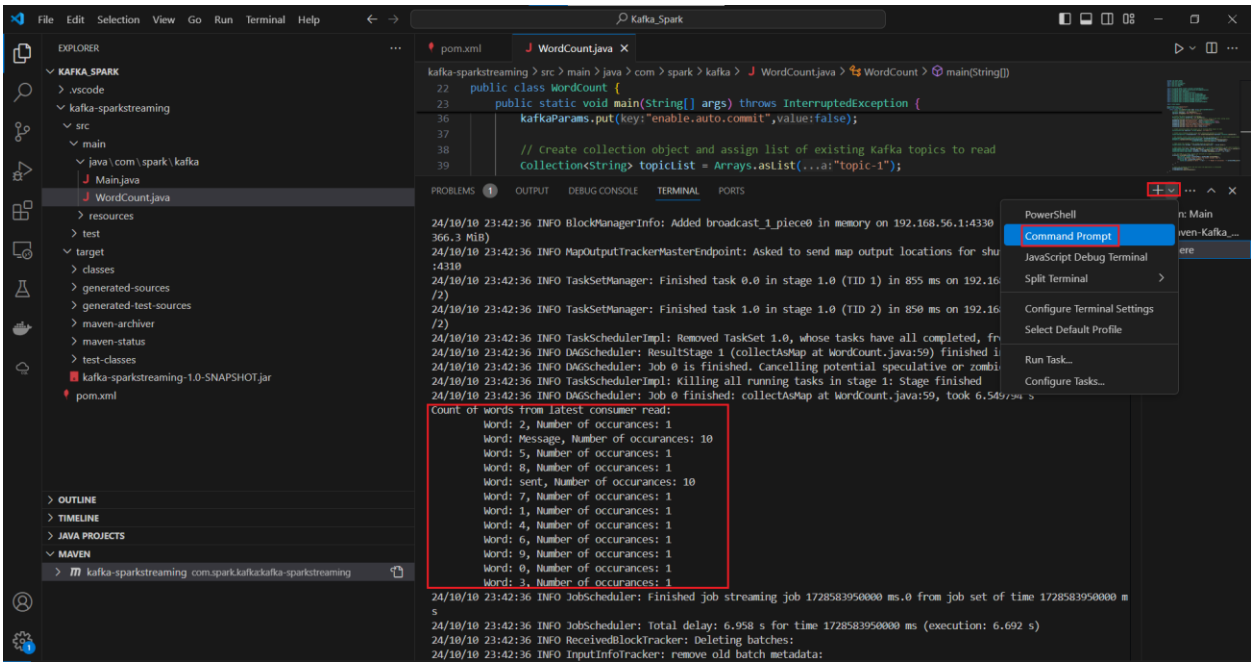
Now that our application jar file is ready, we need to deploy our application using `spark-submit` script that comes with Spark installation.

In VS Code, click on **+** button on **Terminal** window and select **Command Prompt** where you need to run the following command to deploy the application in client mode on Spark cluster. In the below command, make sure to replace `spark://<ip_address>:7077` with your Spark Master URL that is displayed on Spark Master UI at http://localhost:8080/

```
%SPARK_HOME%\bin\spark-submit.cmd --master spark://<ip_address>:7077
--packages org.apache.kafka:kafka-
clients:3.8.0,org.apache.spark:spark-streaming-kafka-0-10_2.12:3.5.1
--class com.spark.kafka.WordCount kafka-sparkstreaming\target\kafka-
sparkstreaming-1.0-SNAPSHOT.jar
```

After executing the above command, you can see INFO logging on the console which says it is downloading necessary packages, connecting to spark cluster, integrating with Kafka cluster to

read from Kafka topic `topic-1`. Then it displays the frequency of words in the messages as shown below:



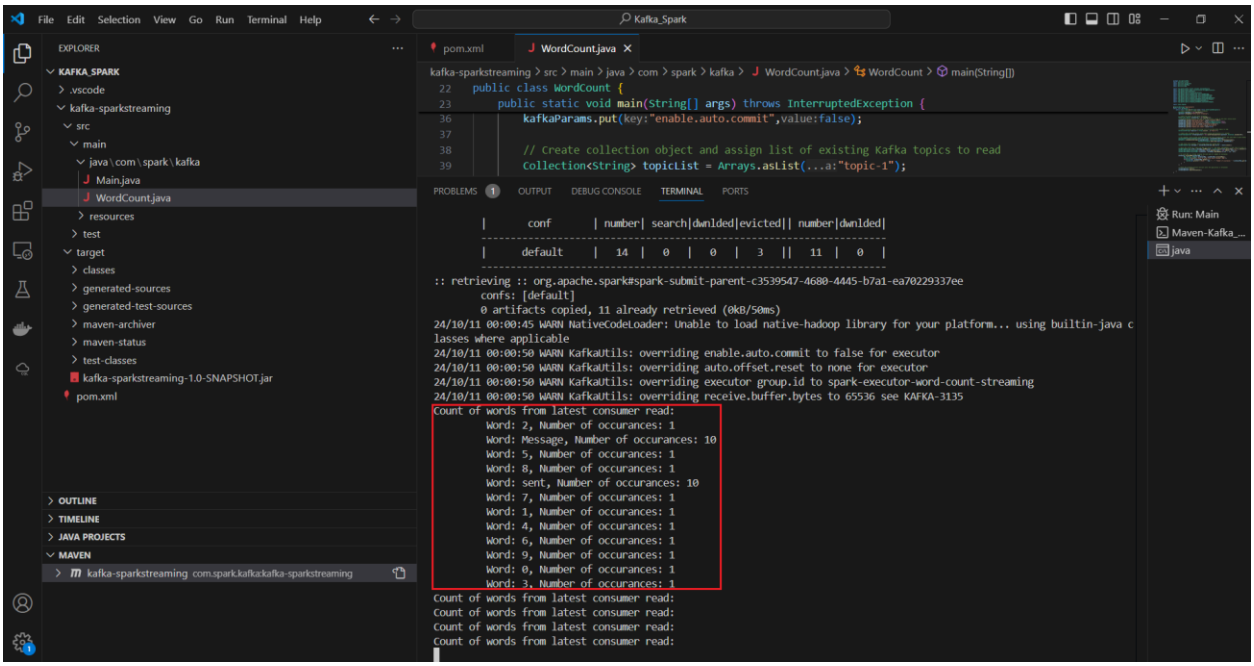
As this is a streaming application, it keeps reading data from Kafka topic at 30 seconds batch interval (as specified in our application) and displays frequency of words for every batch until the application is terminated *(You can enter **Ctrl + C** to terminate this application manually).*

Note: By default, Spark displays **INFO** logging on the console.
Follow these steps to turn off the unnecessary INFO logging:

- Go to `%SPARK_HOME%\conf` folder, take the copy of `log4j2.properties.template` file and rename as `log4j2-spark.properties`.
- Open `log4j2-spark.properties` and change the `rootLogger.level` value from `info` to `warn`.
- On the VS Code terminal, execute the below `spark-submit` command. Make sure to replace `spark://<ip_address>:7077` with your Spark Master URL displayed on Spark Master UI and also put the absolute path of your SPARK_HOME in `configurationFile` path

```
%SPARK_HOME%\bin\spark-submit.cmd --master spark://<ip_address>:7077
--driver-java-options "-
Dlog4j.configurationFile=file:///D:/ProgramFiles/Spark/spark-3.5.1-
bin-hadoop3/conf/log4j2-spark.properties" --packages
org.apache.kafka:kafka-clients:3.8.0,org.apache.spark:spark-
```

```
streaming-kafka-0-10_2.12:3.5.1 --class com.spark.kafka.WordCount
kafka-sparkstreaming\target\kafka-sparkstreaming-1.0-SNAPSHOT.jar
```



Here, you can see that the final output on the console without any other logging.

Congratulations!! You have successfully installed Apache Kafka with single broker and multi-broker configuration and created a streaming application by integrating Kafka with Spark in Windows operating system.