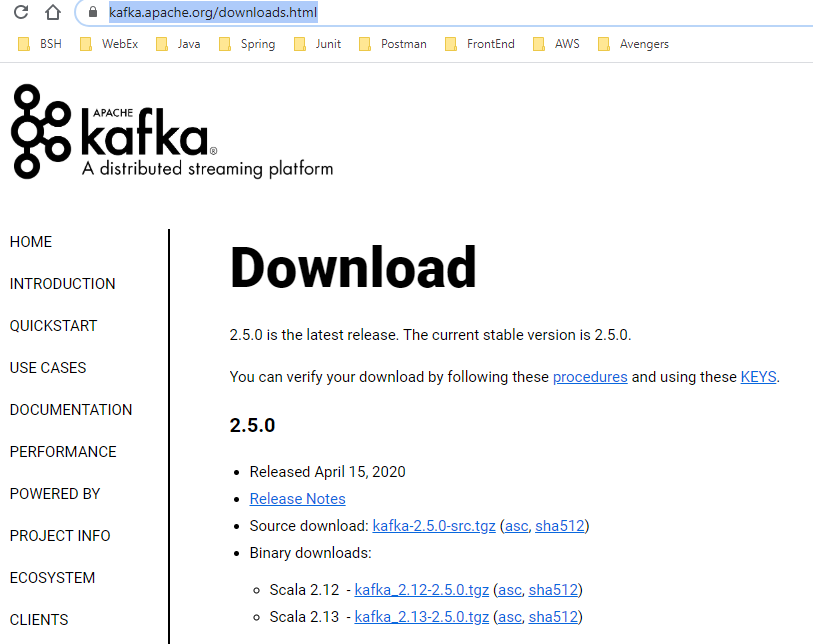
How to download Kafka?



Current version while downloading Kafka is 2.5.0. You can download any latest version.

You can download kafka from apache official website <https://kafka.apache.org/downloads.html>

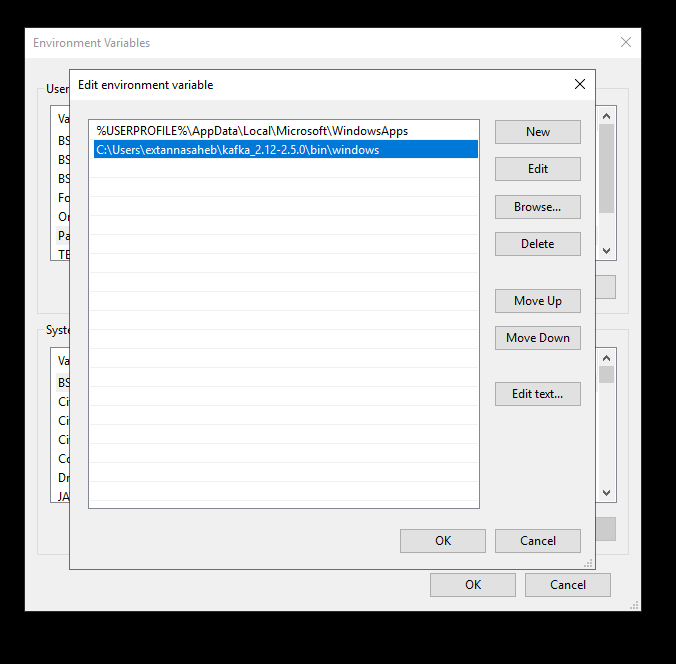
Select .tgz Scala binary download files

Download *Scala 2.12  -*[*kafka\_2.12-2.5.0.tgz*](https://www.apache.org/dyn/closer.cgi?path=/kafka/2.5.0/kafka_2.12-2.5.0.tgz)*(*[*asc*](https://www.apache.org/dist/kafka/2.5.0/kafka_2.12-2.5.0.tgz.asc)*,*[*sha512*](https://www.apache.org/dist/kafka/2.5.0/kafka_2.12-2.5.0.tgz.sha512)*)*



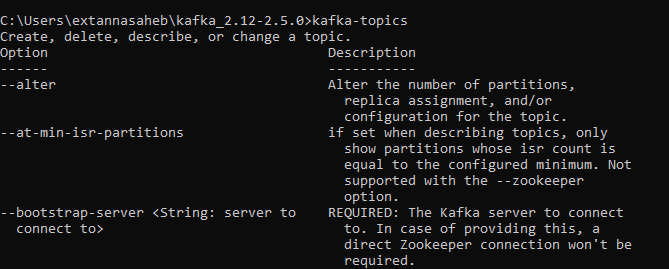
Extract downloaded tar file and extract it any folder

Set the Kafka path in environment path of windows



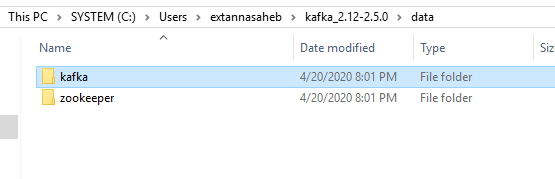
Kafka path has to be up to /bin/windows

You can test Kafka installation using command prompt.



If you see list of command options then Kafka is installed correctly in your system.

Create 2 folders in Kafka data folder 1) zookeeper 2) kafka



Edit zookeeper.properties & kafka.properties present in config folder of Kafka.

In zookeeper.properties: change this line **dataDir** with below folder directory

Dir=C:/kafka\_2.12-2.0.0/data/zookeeper (yes the slashes are inversed)

In server.properties: change **log.dirs** with

C:/kafka\_2.12-2.0.0/data/kafka (yes the slashes are inversed)

After setting these properties start zookeeper and kafka server in sequence.

**Note:** First Zookeeper has to be started first.

Start Zookeeper in one command line:

zookeeper-server-start.bat config\zookeeper.properties

Note: If you get this error it means zookeeper.properties file is missing or path where the file is present couldn’t be found

*Caused by: java.lang.IllegalArgumentException: config\zookeeper.properties file is missing*

Then you can use this command as well if first command doesn’t work

zookeeper-server-start.bat ../../config/zookeeper.properties

Start Kafka in another command line:

kafka-server-start.bat config\server.properties

Or use the below command

kafka-server-start.bat ../../config/server.properties

Important: Don't forget to add the extension .bat to commands being run

Hint: Open command prompt in Run As Administrator

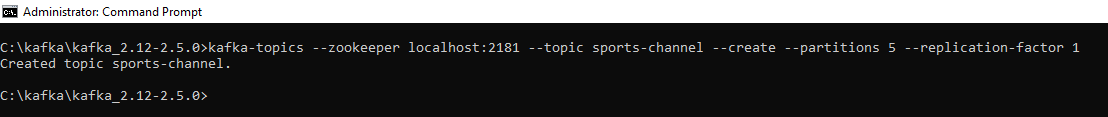
How to create topic in Kafka?

To create topic use below command

Syntax:

***kafka-topics*** --***zookeeper localhost:2181*** ***--topic*** <topic\_name> ***--create*** ***--partitions*** <partition\_number> ***--replication-factor*** <replication\_number>

kafka-topics --zookeeper localhost:2181 --topic topic1 --create --partitions 5 --replication-factor 1



Error scenarios while creating topic

kafka-topics --zookeeper localhost:2181 --topic first-topic --create

Missing required argument "[partitions]"

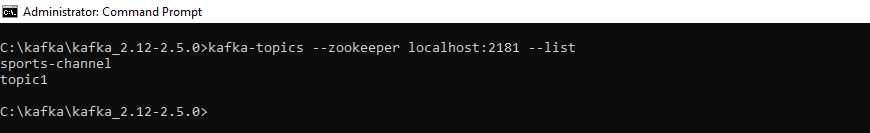
kafka-topics --zookeeper localhost:2181 --topic first-topic --create --partitions 3 replication-factor 2

Missing required argument "[replication-factor]"

How to list topics?

Syntax:

***kafka-topics*** --***zookeeper*** ***localhost:2181*** --***list***

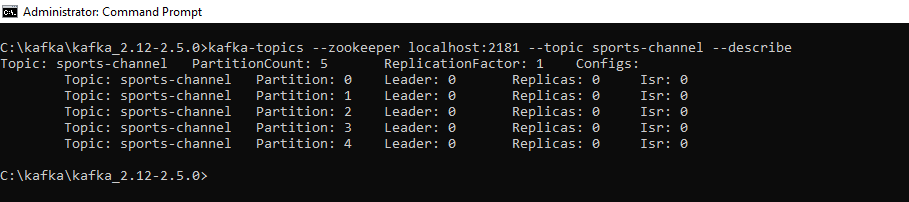


How to describe topic?

Syntax:

***kafka-topics*** --***zookeeper localhost:2181*** --***topic*** ***<topic\_name>*** --**d*escribe***

***kafka-topics*** --***zookeeper localhost:2181*** --***topic*** ***sports-channel*** --**d*escribe***



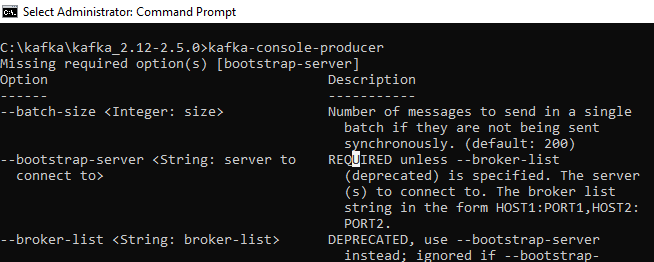
This command will give complete description of a topic.

How to produce message in Kafka?

Use ***kafka-console-producer*** to produce message in kafka.

Run kafka-console-producer to identify the required parameters for producing message in kafka.

Kafka producer run on port 9092

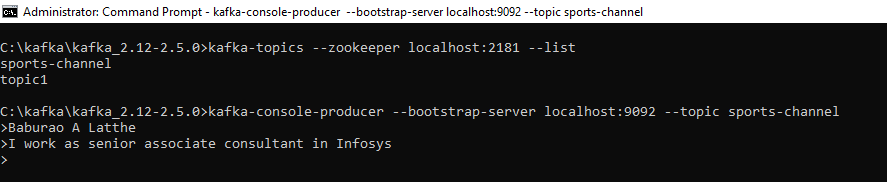


Syntax:

***kafka-console-producer*** --***bootstrap-server localhost:***<port-no> --***topic*** <topic-name>

***kafka-console-producer*** --***bootstrap-server localhost:***9092 --***topic*** sports-channel

After entering this command post the messages and later end producing message using Ctlr+C command.



The > caret symbol indicates to produce message.

End producing message using Ctrl+C.

How kafka produce message to topics which doesn’t exist?

Kafka before producing messages to topic which doesn't exist's it first creates the topic and then produce the message to that topic.

The topic created in this manner will set the default replication-factor value present in zookeeper.properties.

Description of non-existing topic replication-factor is by defaults set as 1 you can change the value by setting the value in zookeeper.properties.

How to consume Kafka messages?

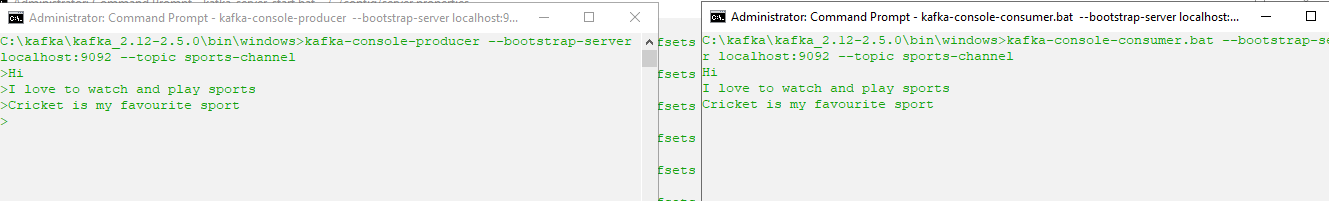
Kafka messages can consumed when producer produces messages. Kafka consumer will be ready to listen the messages what kafka producer will produce. Kafka consumer will run on 9092 port locally and it has to specify the topic name which it want to consume

Syntax:

***kafka-console-consumer.bat*** --***bootstrap-server localhost:***<port-no> --topic <topic-name>

***kafka-console-consumer.bat*** --***bootstrap-server localhost***:9092 --topic sports-channel

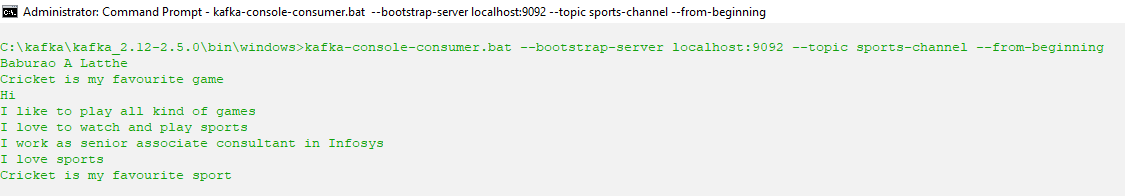
This is how messages are consumed in kafka which is shown below.



Producer Consumer

Kafka consumer will consume messages from the time where consumer started to consume messages produced by producer, the messages prior to it are not consumed in order to shift message offset to beginning of partition we have option --from-beginning which reads messages from offset zero.

kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic sports-channel --from-beginning



What is Kafka consumer group?

Generally, a Kafka consumer belongs to a particular consumer group. A consumer group basically represents the name of an application. In order to consume messages in a consumer group, '**-group**' command is used.

Let' see how consumers will consume messages from Kafka topics:

**Step1:** Open the Windows command prompt.

Generally, a Kafka consumer belongs to a particular consumer group. A consumer group basically represents the name of an application. In order to consume messages in a consumer group, **-group** command is used.

Let' see how consumers will consume messages from Kafka topics:

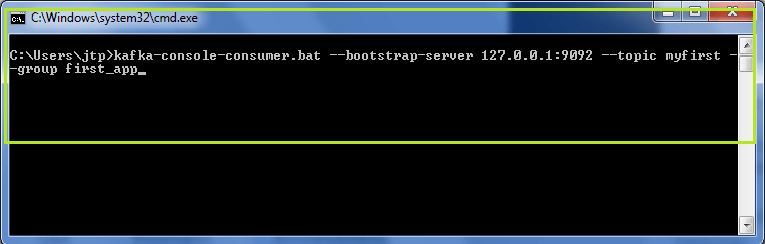
**Step1:** Open the Windows command prompt.

**Step2:** Use the **-group** command as:

Syntax:

**kafka-console-consumer -bootstrap-server localhost:9092 -topic <topic-name> -group <group\_name>**

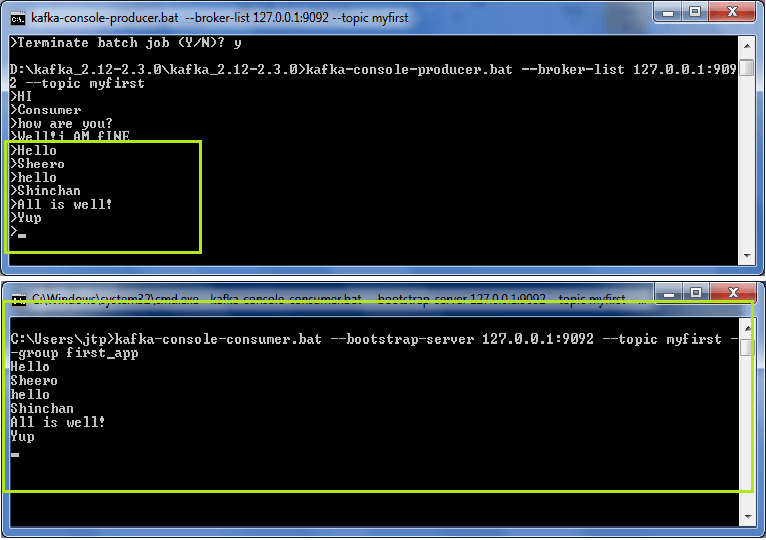
Give some name to the group. Press enter.



In the above snapshot, the name of the group is '**first\_app**'. It is seen that no messages are displayed because no new messages were produced to this topic.

If **-from-beginning** command will be used, all the previous messages will be displayed.

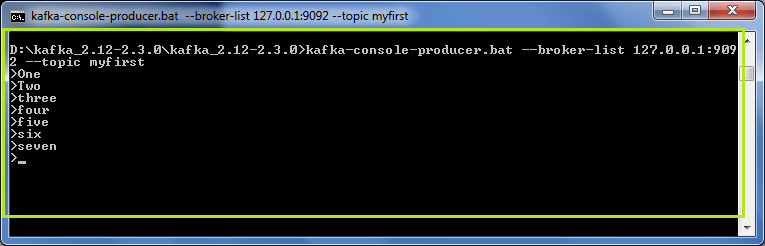
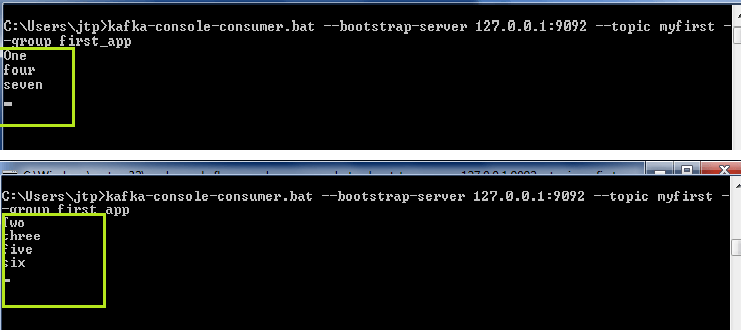
**Step3:** To view some new messages, produce some instant messages from the producer console.



So, the new messages produced by the producer can be seen in the consumer's console.

**Step4:** But, it was a single consumer reading data in the group. Let's create more consumers to understand the power of a consumer group. For that, open a new terminal and type the exact same consumer command as:

**kafka-console-consumer.bat --bootstrap-server 127.0.0.1:9092 --topic <topic\_name> --group <group\_name>**

In the above snapshot, it is clear that the producer is sending data to the Kafka topics. The two consumers are consuming the messages. Look at the sequence of the messages. As there were three partitions created for 'myfirst' topic, so messages are split in that sequence only.

We can further create more consumers under the same group, and each consumer will consume the messages according to the number of partitions. Try yourself to understand better.

#### Note: The group id should be the same, then only the messages will be split between the consumers.

However, if any of the consumers is terminated, the partitions will be reassigned to the active consumers, and these active consumers will receive the messages.

So, in this way, various consumers in a consumer group consume the messages from the Kafka topics.

zookeeper-server-start.bat ../../config/zookeeper.properties

kafka-server-start.bat ../../config/server.properties

kafka-console-producer --bootstrap-server localhost:9092 --topic sports-channel

kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic sports-channel

Kafka Programming

Note: <https://kafka.apache.org/documentation/>

# Creating Kafka Producer in Java

Create a basic Kafka Project. Now, before creating a Kafka producer in java, we need to define the essential Project dependencies. In our project, there will be two dependencies required:

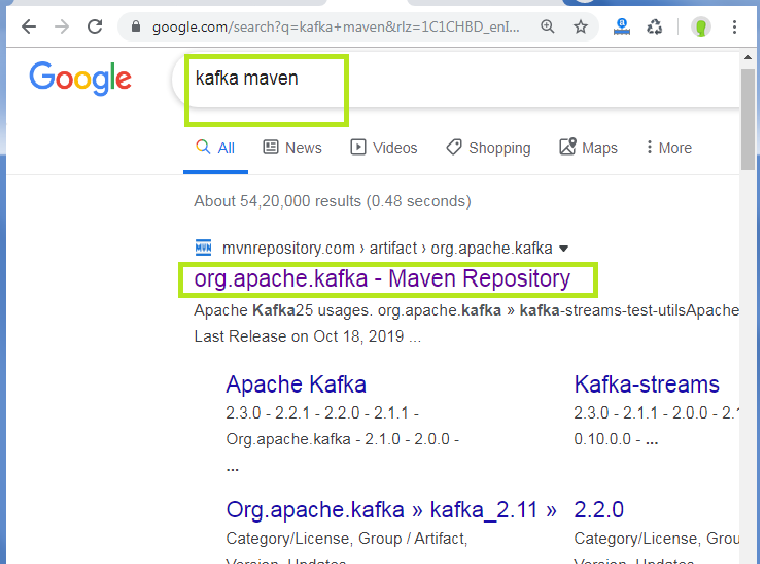
1. Kafka Dependencies
2. Logging Dependencies, i.e., SLF4J Logger.

There are following steps required to set the dependencies:

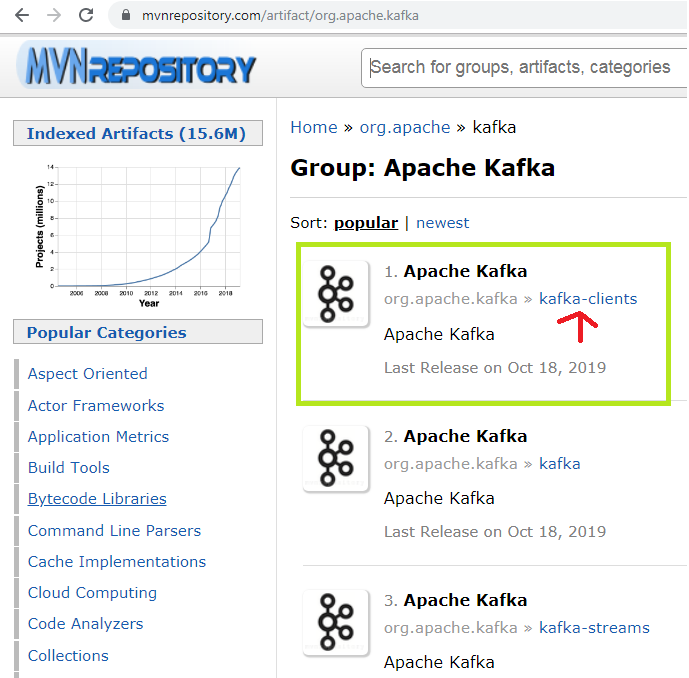
**Step1:** The build tool Maven contains a '**pom.xml**' file. The 'pom.xml' is a default XML file that carries all the information regarding the GroupID, ArtifactID, as well as the Version value. The user needs to define all the necessary project dependencies in the 'pom.xml' file. Go to the 'pom.xml' file.

**Step2:** Firstly, we need to define the Kafka Dependencies. Create a '**<dependencies>...</dependencies>**' block within which we will define the required dependencies.

**Step3:** Now, open a web browser and search for 'Kafka Maven' as shown below:



Click on the highlighted link and select the '**Apache Kafka, Kafka-Clients**' repository. A sample is shown in the below snapshot:



**Step4:** Select the repository version according to the downloaded Kafka version on the system. For example, in this tutorial, we are using 'Apache Kafka 2.3.0'. Thus, we require the repository version 2.3.0 (the highlighted one).



**Step5:** After clicking on the repository version, a new window will open. Copy the dependency code from there.



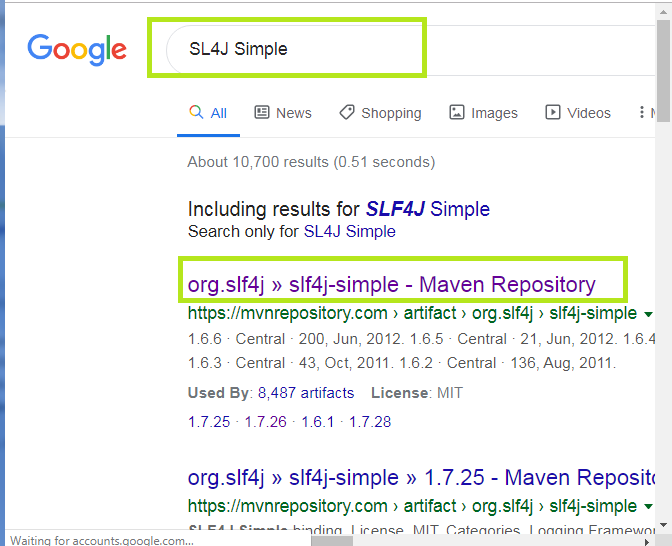
Since, we are using Maven, copy the Maven code. If the user is using Gradle, copy the Gradle written code.

**Step6:** Paste the copied code to the '**<dependencies>...</dependencies>**' block, as shown below:

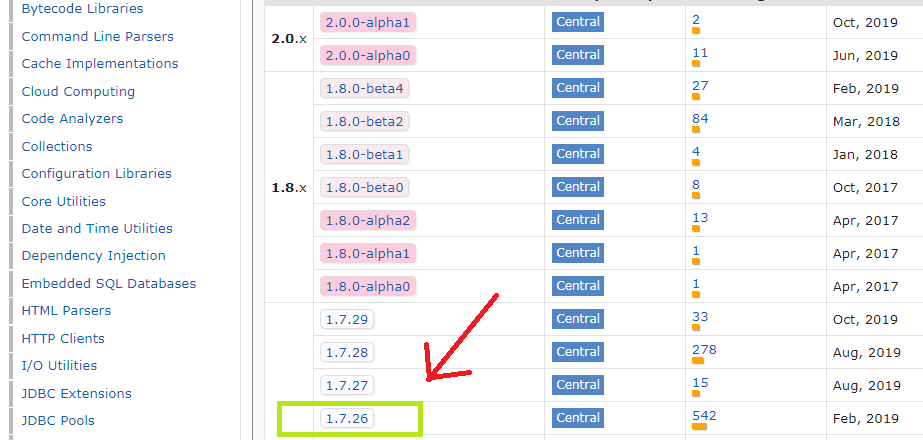


If the version number appears red in color, it means the user missed to enable the '**Auto-Import**' option. If so, go to **View>Tool Windows>Maven**. A Maven Projects Window will appear on the right side of the screen. Click on the 'Refresh' button appearing right there. This will enable the missed Auto-Import Maven Projects. If the color changes to black, it means the missed dependency is downloaded. The user can proceed to the next step.

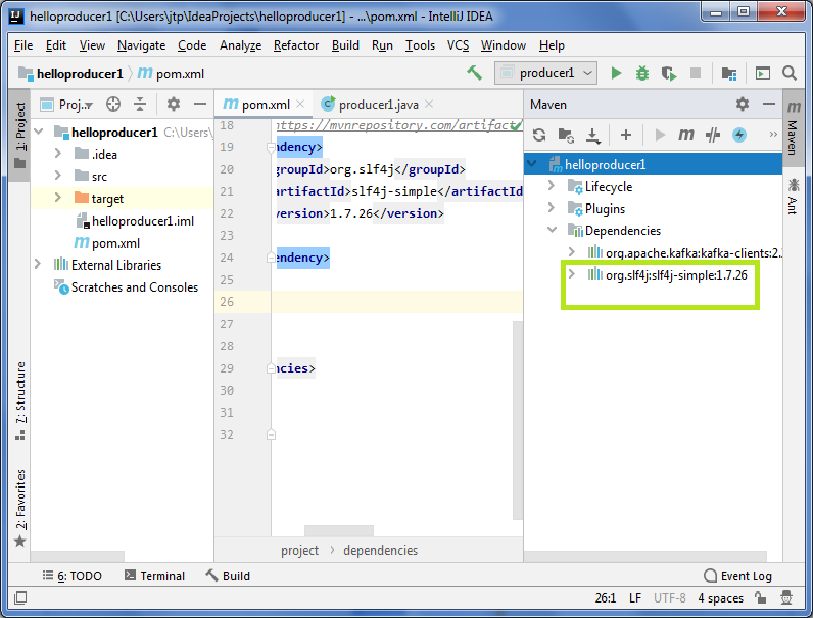
**Step7:** Now, open the web browser and search for 'SL4J Simple' and open the highlighted link shown in the below snapshot:



A bunch of repositories will appear. Click on the appropriate repository.



To know the appropriate repository, look at the Maven projects window, and see the slf4j version under 'Dependencies'.



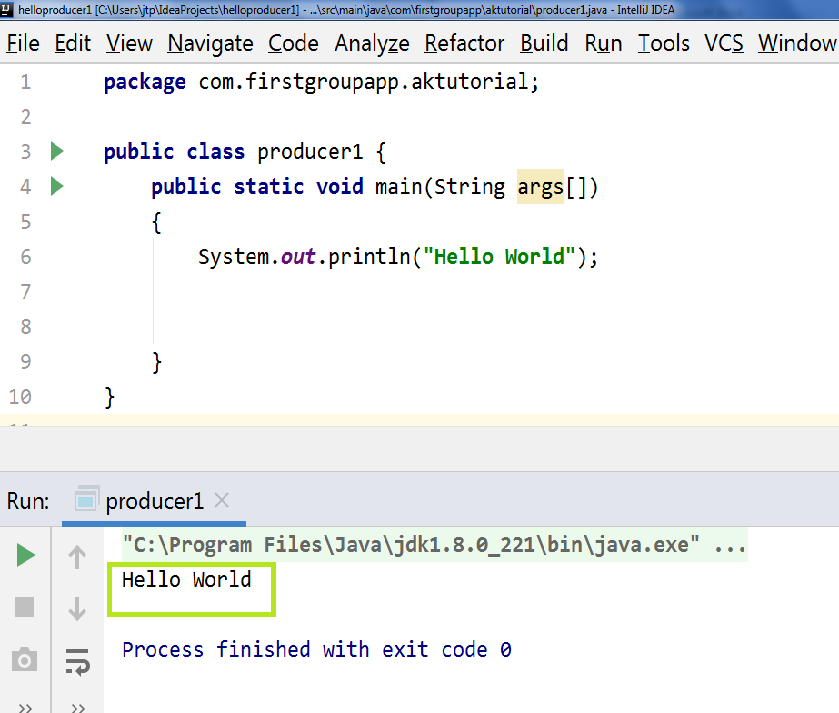
Click on the appropriate version and copy the code, and paste below the Kafka dependency in the 'pom.xml' file, as shown below:



#### Note: Either put a comment or remove the <scope> test</scope> tag line from the code. Because this scope tag defines a limited scope for the dependency, and we need this dependency for all code, the scope should not be limited.

Now, we have set all the required dependencies. Let's try the 'Simple Hello World' example.

Firstly, create a java package say, ‘com.kafka’ and a java class beneath it. While creating the java package, follow the package naming conventions. Finally, create the 'hello world' program.



After executing the 'producer1.java' file, the output is successfully displayed as 'Hello World'. This tells the successful working of the IntelliJ IDEA.

## Creating Java Producer

Basically, there are four steps to create a java producer, as discussed earlier:

1. Create producer properties
2. Create the producer
3. Create a producer record
4. Send the data.

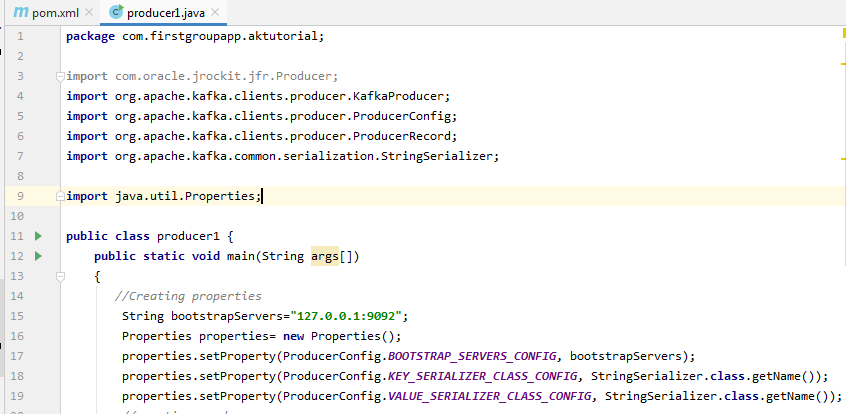
## Creating Producer Properties

Apache Kafka offers various Kafka Properties which are used for creating a producer. To know about each property, visit the official site of Apache, i.e., '**https://kafka.apache.org**'. Move to Kafka>Documentations>Configurations>Producer Configs.

There the users can know about all the producer properties offered by Apache Kafka. Here, we will discuss the required properties, such as:

1. **bootstrap.servers:** It is a list of the port pairs which are used for establishing an initial connection to the Kafka cluster. The users can use the bootstrap servers only for making an initial connection only. This server is present in the host:port, host:port,... form.
2. **key.serializer:** It is a type of Serializer class of the key which is used to implement the 'org.apache.kafka.common.serialization.Serializer' interface.
3. **value.serializer:** It is a type of Serializer class which implements the 'org.apache.kafka.common.serialization.Serializer' interface.

Now, let's see the implementation of the producer properties in the IntelliJ IDEA.



When we create the properties, it imports the '**java.util.Properties**' to the code.

So, in this way, the first step to create producer properties is completed.

### Creating the Producer

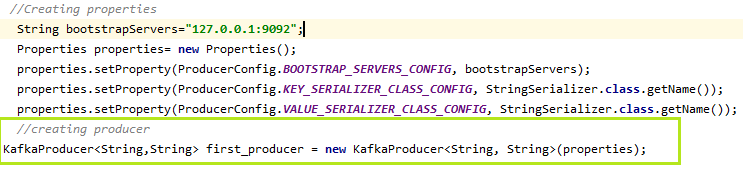
To create a Kafka producer, we just need to create an object of KafkaProducer.

The object of KafkaProducer can be created as:

1. KafkaProducer<String,sString> first\_producer = new KafkaProducer<String, String>(properties);

Here, '**first\_producer**' is the name of the producer we have chosen. The user can choose accordingly.

Let's see in the below snapshot:



## Creating the Producer Record

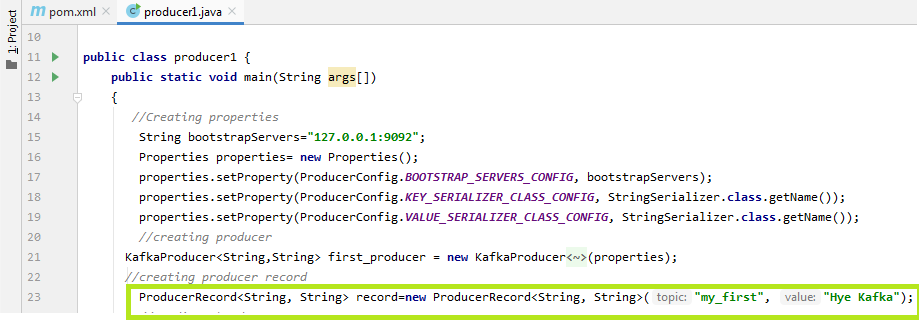
In order to send the data to Kafka, the user need to create a ProducerRecord. It is because all the producers lie inside a producer record. Here, the producer specifies the topic name as well as the message which is to be delivered to Kafka.

A ProducerRecord can be created as:

1. ProducerRecord<String, String> record=new ProducerRecord<String, String>("my\_first", "Hye Kafka");

Here, 'record' is the name chosen for creating the producer record, 'my\_first' is the topic name, and 'Hye Kafka' is the message. The user can choose accordingly.

Let's see in the below snapshot:

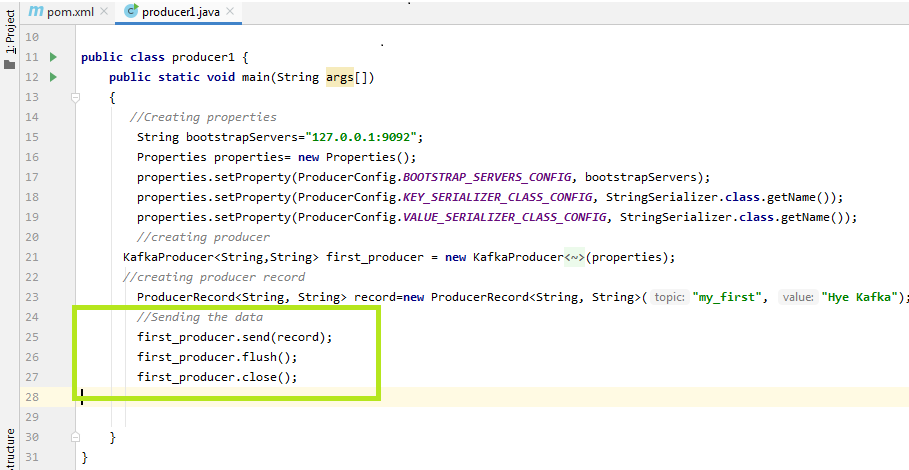


## Sending the data

Now, the user is ready to send the data to Kafka. The producer just needs to invoke the object of the ProducerRecord as:

1. first\_producer.send(record);

Let's see in the below snapshot:

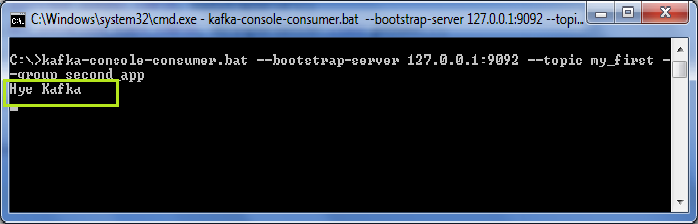
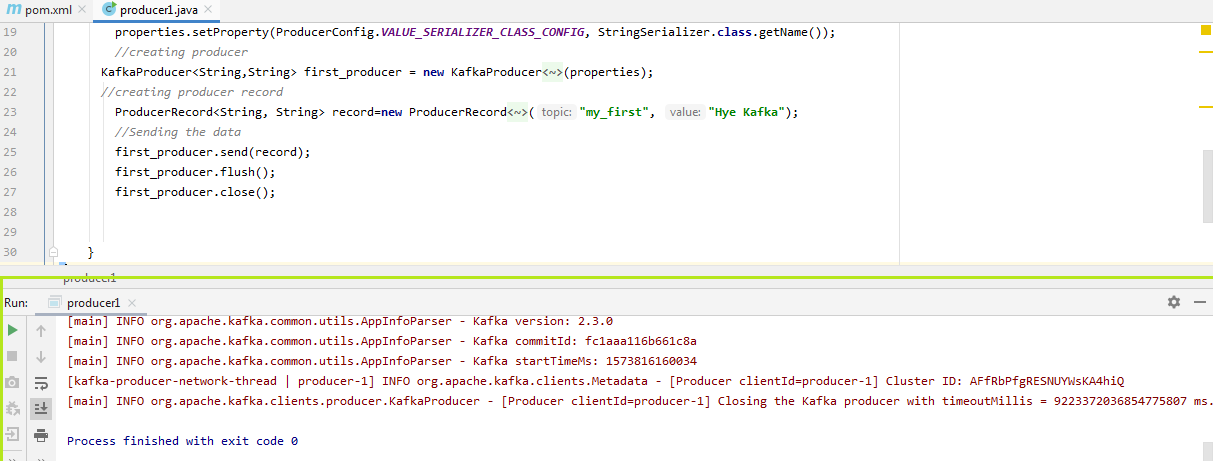


To know the output of the above codes, open the '**kafka-console-consumer**' on the CLI using the command:

'**kafka-console-consumer -bootstrap-server 127.0.0.1:9092 -topic my\_first -group first\_app**'

The data produced by a producer is asynchronous. Therefore, two additional functions, i.e., **flush()** and **close()** are required (as seen in the above snapshot). The flush() will force all the data to get produced and close() stops the producer. If these functions are not executed, data will never be sent to the Kafka, and the consumer will not be able to read it.

The below shows the output of the code on the consumer console as:

On the terminal, users can see various log files. The last line on the terminal says the Kafka producer is closed. Thus, the message gets displayed on the consumer console asynchronously.

# Kafka Producer Callbacks

## **Producer without Keys**

In the previous section, we saw how a producer sends data to Kafka. In order to understand more deeply, i.e., whether the data was correctly produced, where it was produced, about its offset and partition value, etc. Let's learn more.

For performing the callbacks, the user needs to implement a callback function. This function is implemented for asynchronously handling the request completion. That's why it's return type will be void. This function will be implemented in the block where the producer sends data to the Kafka. There is no requirement to make changes in other blocks of codes.

The callback function used by the producer is the onCompletion(). Basically, this method requires two arguments:

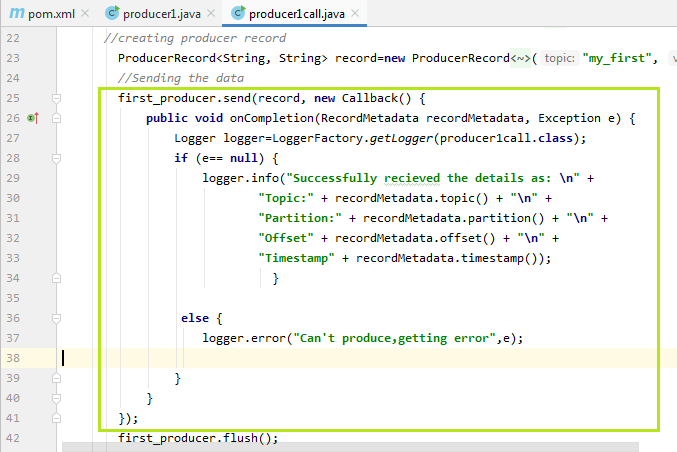
**Metadata of the Record:** Metadata of the record means fetching the information regarding the partition and its offsets. If it is not null, an error will be thrown.

**Exception:** There are following exceptions which can be thrown while processing:

**1) Retriable exception:** This exception says that the message may be sent.

**2) Non-retriable exception:** This exception throws the error that the message will never be sent.

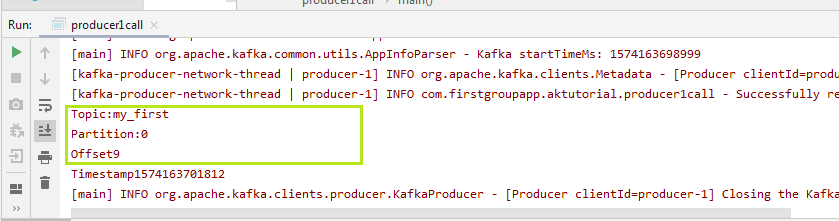
Let's see the implementation of the Producer callback in the below snapshot:



1. first\_producer.send(record, **new** Callback() {
2. **public** **void** onCompletion(RecordMetadata recordMetadata, Exception e) {
3. Logger logger=LoggerFactory.getLogger(producer1call.**class**);
4. **if** (e== **null**) {
5. logger.info("Successfully received the details as: \n" +
6. "Topic:" + recordMetadata.topic() + "\n" +
7. "Partition:" + recordMetadata.partition() + "\n" +
8. "Offset" + recordMetadata.offset() + "\n" +
9. "Timestamp" + recordMetadata.timestamp());
10. }
12. **else** {
13. logger.error("Can't produce,getting error",e);
15. }
16. }
17. });

An object of '**Logger**' has been created, which allows to import '**slf4j.Logger**' and '**slf4j.LoggerFactory**'. This logger object will log the information regarding the partition, offsets, as well as the timestamp. If the exception value is equal to null, the logger will display the information, else an error will be displayed. When the above code is executed, the user will come to know the topic name, partition number, timestamp, offset value where the message is sent.

A snapshot of the output is shown below:



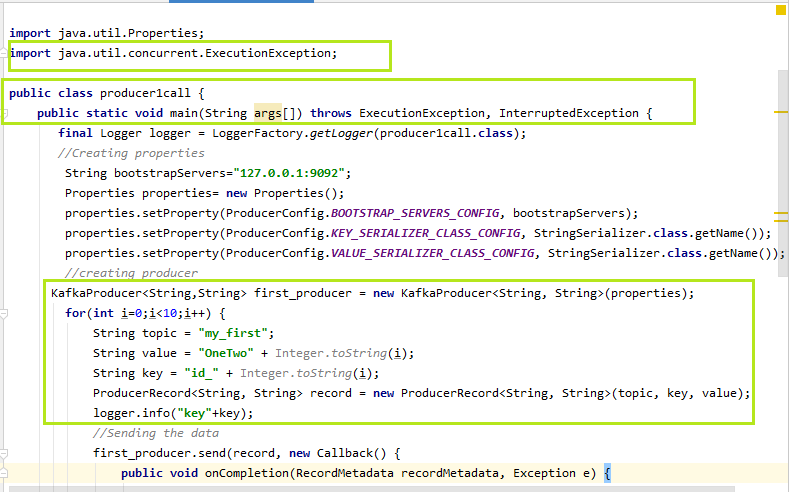
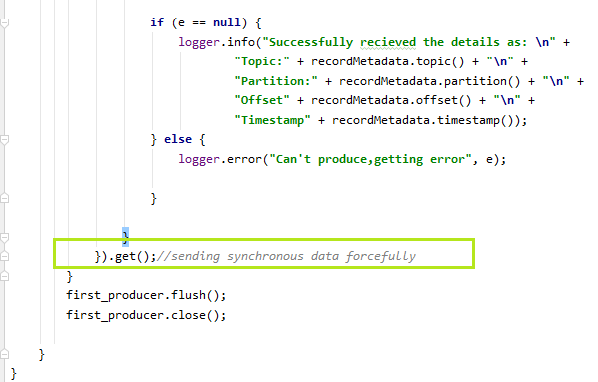
In the above output, it is seen that the message is produced to 'my\_first', stored at 'partition 0' having the 'offset value 9'.

#### Note: The messages we sent till now are without keys, therefore messages without keys get stored in the random partitions and behave asynchronously.

## **Producer with Keys**

Keys become useful when a user wants to send the message to the same partition. In order to send the data, the user need to specify a key. The key will uniquely identify the partition from the other partitions. The user needs to send synchronous messages to the Kafka.

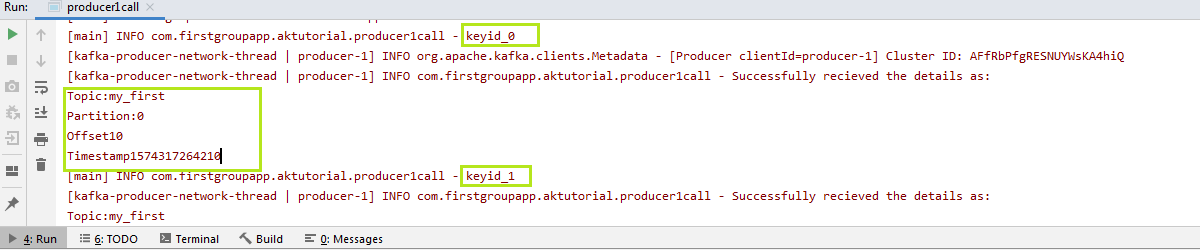
One way to implement a key is shown below:

In the above snapshot, we have specified the topic name, its value, and the key. While creating the ProducerRecord, three of them are passed as the parameters. If the exception 'e' will be equal to null, the logger will fetch the information about the key. At the end, a **get()** function is used when the data is sent to the Kafka. This method sends the data synchronously and forcefully. The users can try their own ways to implement the keys.

#### Note: Using the get(), a red underline will appear. Press Alt+Enter, it will say to 'Add exceptions to method signature', select it. This will add two exceptions to the main(), as shown above. Also, it will import 'java.util.concurrent.ExecutionException' to the code.

When the above code is executed, the output is displayed as:



The highlighted parts in the output tell the key values, topic name, partition number, offset value, as well as the timestamps. The message 'OneTwo' will always go to the specified partitions now.

So, in this way, a producer can send data to the Kafka with and without keys.

**How to create safe producer?**

Add the following configuration to Kafka Producer to make producer safe

properties.setProperty(ProducerConfig.ACKS\_CONFIG, Constants.ACKS\_CONFIG);

properties.setProperty(ProducerConfig.ENABLE\_IDEMPOTENCE\_CONFIG, "true");

properties.setProperty(ProducerConfig.RETRIES\_CONFIG, String.valueOf(Integer.MAX\_VALUE));

properties.setProperty(ProducerConfig.MAX\_IN\_FLIGHT\_REQUESTS\_PER\_CONNECTION, "5");

# Message Compression in Kafka

As we have seen that the producer sends data to the Kafka in the text format, commonly called the **JSON** format. JSON has a demerit, i.e., data is stored in the string form. This creates several duplicated records to get stored in the Kafka topic. Thus, it occupies much disk space. Consequently, it is required to reduce disk space. This can be done by compressing or lingering the data before sending it to the Kafka.

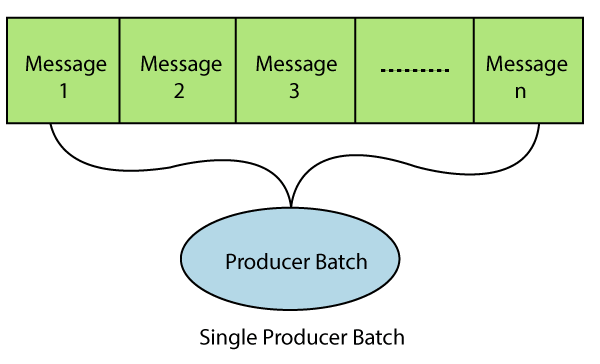
## **Need for Message Compression**

There can be the following reasons which better describes the need to reduce the message size:

1. It will reduce the latency and size required to send data to Kafka.
2. It will reduce the bandwidth that will make users increase the net messages which are sent to the broker.
3. It can lead to low cost when the data is stored in the Kafka via cloud platforms. It is because cloud services are paid. Therefore, it calculates the amount of data stored in Kafka.
4. Message compression does not need any change in the configuration of the broker and consumer.
5. Message compression does not need any change in the configuration of the broker and consumer.
6. The reduced disk load will lead to fast read and write operations.

## **Producer Batch/Record Batch**

A producer writes messages to the Kafka, one by one. Therefore, Kafka plays smartly. It waits for the messages that are being produced to Kafka. Then, it creates a batch and put the messages into it, until it becomes full. Then, send the batch to the Kafka. Such type of batch is known as a **Producer Batch**. The default batch size is 16KB, and the maximum can be anything. Large is the batch size, more is the compression, throughput, and efficiency of producer requests.

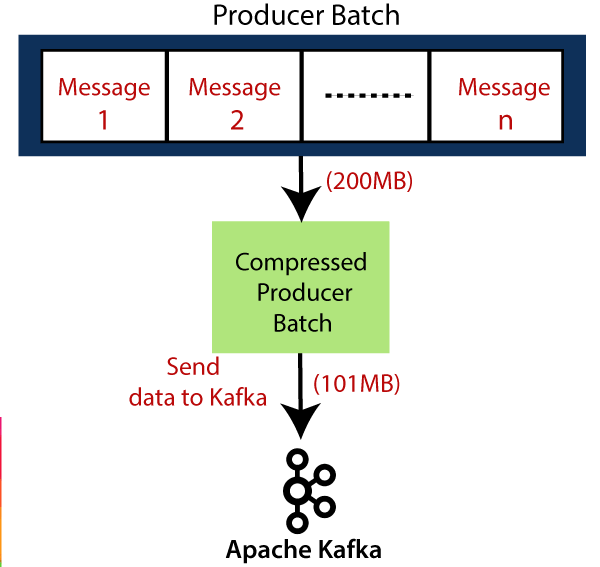


#### Note: The message size should not exceed the batch size. Otherwise, the message will not be batched. Also, the batch is allocated per partitions, so do not set it to a very high number.

Bigger is the producer batch, effective to use the message compression technique.

## **Message Compression Format**

Message Compression is always done at the producer side, so there is no requirement to change the configurations at the consumer or broker side.



In the figure, a producer batch of 200 MB is created. After compression, it is reduced to 101 MB.

To compress the data, a 'compression.type' is used. This lets users decide the type of compression. The type can be 'gzip', 'snappy', 'lz4', or 'none'(default). The 'gzip' has the maximum compression ratio.

## **Disadvantages of Message Compression**

There are following disadvantages of the message compression:

1. The producers commit some CPU cycles for compression.
2. The consumers commit some CPU cycles for decompression.
3. These disadvantages lead to increased CPU usage.

Thus, message compression is a better option to reduce the disk load.

**Linger.ms:** Linger.ms this has to be set on the Kafka Producer side. The linger.ms take milliseconds as the value. The value give will instruct Kafka Producer to wait for certain milliseconds before sending the message to Kafka.

**Batch Size:** The message size producer has to wait until producer sends message to kafka. Producer will store the messages into batches until it reaches batch size and then send to kafka as the batch message