



Sri Indu

College of Engineering & Technology

UGC Autonomous Institution

Recognized under 2(f) & 12(B) of UGC Act 1956,
NAAC, Approved by AICTE &
Permanently Affiliated to JNTUH



NAAC
NATIONAL ASSESSMENT AND
ACCREDITATION COUNCIL



SKILL DEVELOPMENT COURSE LAB MANUAL

CSE III (AI&DS) –Semester I

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (AI&DS)

ACADEMIC YEAR 2024-25



SRI INDU COLLEGE OF ENGINEERING & TECHNOLOGY

(An Autonomous Institution under UGC, New Delhi)

Recognized under 2(f) and 12(B) of UGC Act 1956

NBA Accredited, Approved by AICTE and Permanently affiliated to JNTUH

Sheriguda (V), Ibrahimpatnam, R.R.Dist, Hyderabad - 501 510

DEPARTMENT OF

COMPUTER SCIENCE ENGINEERING(ARTIFICIAL INTELLIGENCE AND DATA SCIENCE)

LAB MANUAL

Branch: CSE(AI&CS)

Subject: Computer Networks Lab

Academic Year: 2023-24

Core/Elective/H&S: Core

Class: B.Tech- III Year-I sem

Code:R22CSD3241

Regulation: R22

Credits: 1.5

Prepared By

Name: MISS.T.GLORY

Verified By

Head of the Department:



SRI INDU COLLEGE OF ENGINEERING & TECHNOLOGY
B. TECH –COMPUTER SCIENCE AND ENGINEERING(CS)

INSTITUTION VISION

To be a premier Institution in Engineering & Technology and Management with competency, values and social consciousness.

INSTITUTION MISSION

- IM1** Provide high quality academic programs, training activities and research facilities.
- IM2** Promote Continuous Industry-Institute interaction for employability, Entrepreneurship, leadership and research aptitude among stakeholders.
- IM3** Contribute to the economical and technological development of the region, state and nation.

DEPARTMENT VISION

To be a Technologically adaptive centre for computing by grooming the students as top notch professionals.

DEPARTMENT MISSION

The Department has following Missions:

- DM1** To offer quality education in computing.
- DM2** To provide an environment that enables overall development of all the stakeholders.
- DM3** To impart training on emerging technologies like data analytics , artificial intelligence and internet of things.
- DM4** To encourage participation of stake holders in research and development.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

- PEO1: Higher Studies:** Graduates with an ability to pursue higher studies and get employment in reputed institutions and organizations.
- PEO2: Domain knowledge:** Graduate with an ability to design and develop a product.
- PEO3: Professional Career:** Graduate with an ability to design and develop a product.
- PEO4: Life Long Learning:** Graduate with an ability to learn advanced skills to face professional competence through lifelong learning.

PROGRAM OUTCOMES(POs)

PO	Description
PO 1	Engineering Knowledge: To be able to apply knowledge of computing, mathematics, Science and Engineering appropriate to the discipline
PO 2	Problem Analysis: To be able identify, formulate & analyze a problem, and ascertain and define the computing requirements appropriate to its solution.
PO 3	Design & Development Solutions: To be able to design, implement, and evaluate a computer-based system, process, component, or program to meet desired needs.
PO 4	Investigation of complex problems: To be able to identify and analyze user needs And consider them in the selection, creation, evaluation and administration of computer-based systems for providing valid solutions to complex problems.
PO 5	Modern Tool Usage: To possess skills for creating and in using contemporary techniques, skills, and tools necessary for computing Practice.
PO 6	Engineering & Society: To apply conceptual knowledge relevant to professional engineering practices in societal, health, safety, legal and cultural issues and their consequences
PO 7	Environment & Sustainability: To be able to Analyze the local and global impact of computing on individuals, organizations, and society and work towards sustainable development.
PO 8	Ethics: To understand contemporary professional, ethical, legal, security and social issue sand responsibilities.
PO 9	Individual & Team work: To Be able to function effectively as an individual and on teams to accomplish a common goal.
PO 10	Communication: To communicate precisely and effectively both in oral and written form with a range of audiences.
PO 11	Project management & finance: To apply engineering and management principles For managing and leading economically feasible projects in multi-disciplinary environments with an effective project plan.
PO 12	Life Long Learning: To recognize the need for and an ability to engage in independent & lifelong learning for continuing professional development.
Program Specific Outcomes	
PSO 1	Develop software projects using standard practices and suitable programming environment.
PSO 2	Identify , formulate and solve the real life problems faced in the society, industry and other areas by applying the skills of the programming languages, networks and databases learned.
PSO 3	To apply computer science knowledge in exploring and adopting latest technologies in different co-curricular activities.

COURSE OUTCOMES

C216.1	Implement data link layer framing methods and Analyze error detection and error correction codes.
C216.2	Implement and analyze routing and congestion issues in network design.
C216.3	Implement Encoding and Decoding techniques used in presentation layer.

COsMAPPINGWITHPOs&PSOs

Cours e Outco me	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO1 1	PO 12	PSO 1	PSO2	PS O3
C216.1	2	2	3	2	2	-	-	-	-	1	-	-	1	2	1
C216.2	2	1	2	1	2	-	-	-	-	-	2	-	1	1	1
C216.3	1	2	1	2	1	-	-	--	-	-	-	-	2	1	1
C216	1.6	1.6	2.0	1.6	1.6	-	-	-	-	0.3	0.6	-	1.3	1.3	1.16

List of Experiments

SNO	PROGRAMS
1	Week 1- Installing and Running Kafka.
2	Week 2 - : Configuring Multi-Broker Kafka
3	Week 3 - : Exploring Zookeeper.
4	Week 4 - : Overview of kafka consumers
5	Week 5- : Using Kafka with Docker..
6	Week 6- : Developing Kafka Producer.
7	Week 7- : Sending Messages with Callback.
8	Week 8- :Kafka monitoring and schema Registry.
9	Week 9- : Kafka streams and kafka connectors.
10	Week 10- : Integration of kafka with storm.
11	Week 11- : Kafka integration with spark and flume.
12	Week 12- : Kafka Application as Consumer and Producer.
13	Week 13- : Word Count Per Record

INTRODUCTION TO KAFKA COMPONENTS

TOPICS

- A topic is a feed name or category to which records are published.
- Topics in Kafka are always multi-subscriber that is, a topic can have zero, one, or many consumers that subscribe to the data written to it.
- For each topic, the Kafka cluster maintains a partition log.

PARTITIONS

- A topic may have many partitions so that it can handle an arbitrary amount of data.
- If you create multiple partition for a single topic then the data will be stored randomly on any partition. Which means the data will be received in unordered manner by the consumer.

PARTITION OFFSET

- Each partitioned message has a unique sequence ID called an offset.

REPLICATION FACTOR

Replicas are nothing but backups of a partition. If the replication factor of the topic is set to 4, then Kafka will create four identical replicas of each partition and place them in the cluster to make them available for all its operations. Replicas are never used to read or write data. They are used to prevent data loss.

BROKERS

Brokers are simple systems responsible for maintaining published data. Kafka brokers are stateless, so they use Zookeeper for maintaining their cluster state. Each broker may have zero or more partitions per topic.

ZOOKEEPER

Zookeeper is used for managing and coordinating Kafka brokers. It is mainly used to notify producers and consumers about the presence or about the failure of any broker in the Kafka system.

EXPERIMENT NO 1

NAME OF THE EXPERIMENT: Installing and Running Kafka.

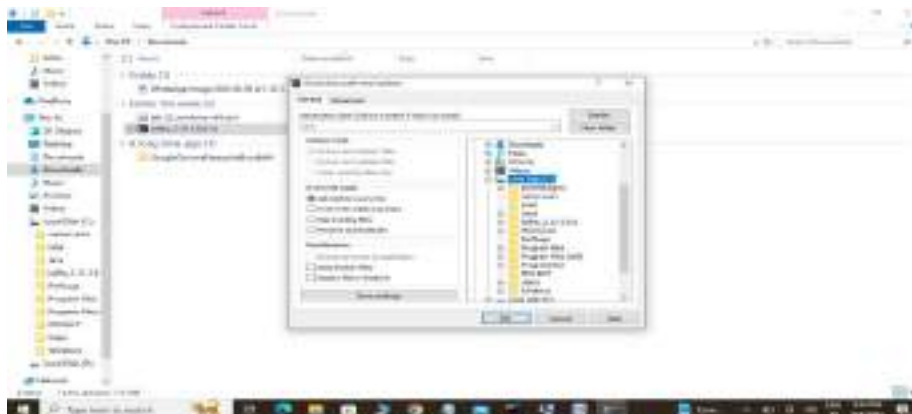
STEP 1: Check java is installed or not, if not install jdk latest version



STEP 2: Install Apache Kafka

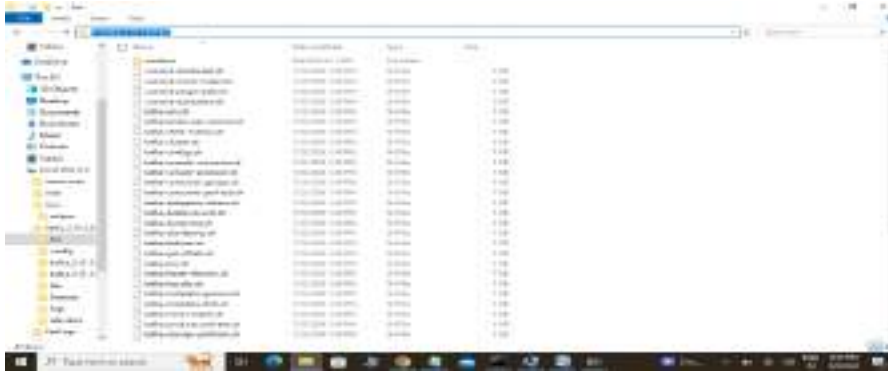


STEP 3: Extract Apache Kafka into C drive

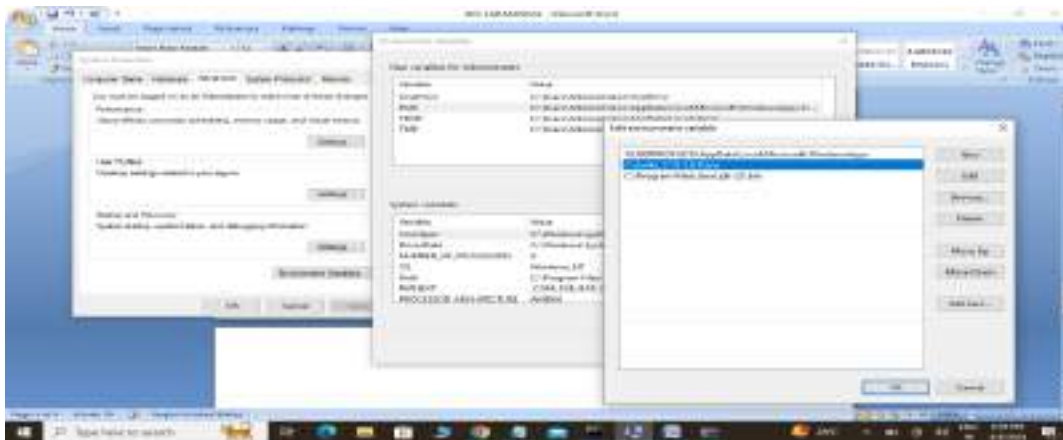


STEP 4: PATH SETUP

Copy Path



- Search Edit The System Environment Variable
- Environment Variable->Path-->Edit-->New-->Paste Path-Ok
- Kafka(copy path-->C:\kafka_2.13-3.8.0\bin)



EXPERIMENT NO 2

NAME OF THE EXPERIMENT: Configuring Multi-Broker Kafka

STEP 1: Open cmd and enter into the path C:\kafka_2.13-3.8.0

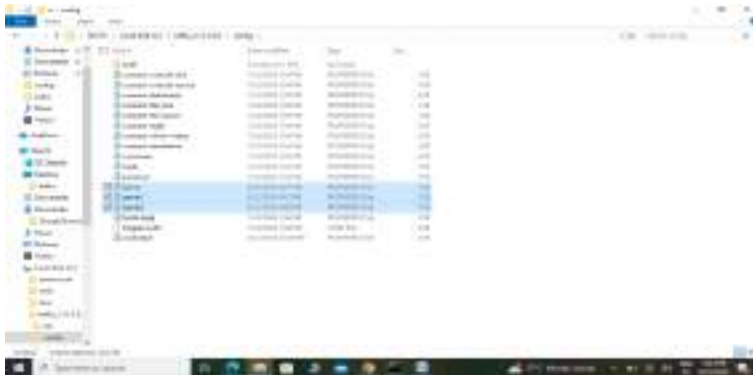
PATH: C:\kafka_2.13-3.8.0

TYPE: `.\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties`



STEP 2: Open kafka folder and click on config

STEP 3: Copy the server and paste 2 times and save it with server1 and server2



STEP 4 : Now OPEN server1 and change the borker.id , listeners and log.dirs

Examples: - These three steps are there in below images

1. In Server, it contains `borker.id=0` then convert 0 into 1 in server1 and 2 in server2

2. In Server contains this : `#listeners=PLAINTEXT://:9092` then

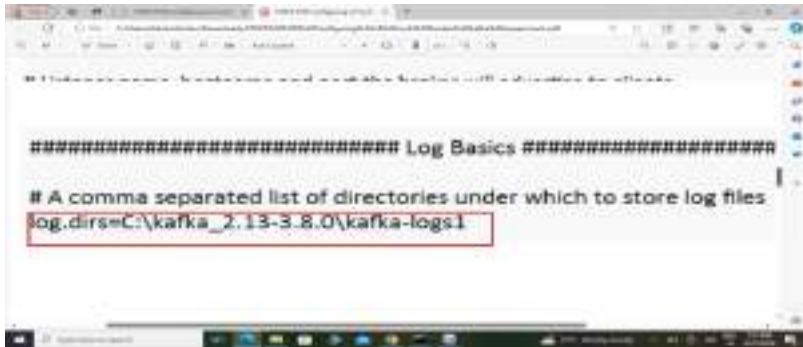
We have to copy it and paste same and change host number

→ `listeners=PLAINTEXT://:9093` for server1 and

→ `listeners=PLAINTEXT://:9094` for server2

3. In server, it contains log.dirs

We have change, log into log1 and log2 in server1 and server2



STEP 5: Open A NEW CMD and enter into the path C:\kafka_2.13-3.8.0

PATH: C:\kafka_2.13-3.8.0

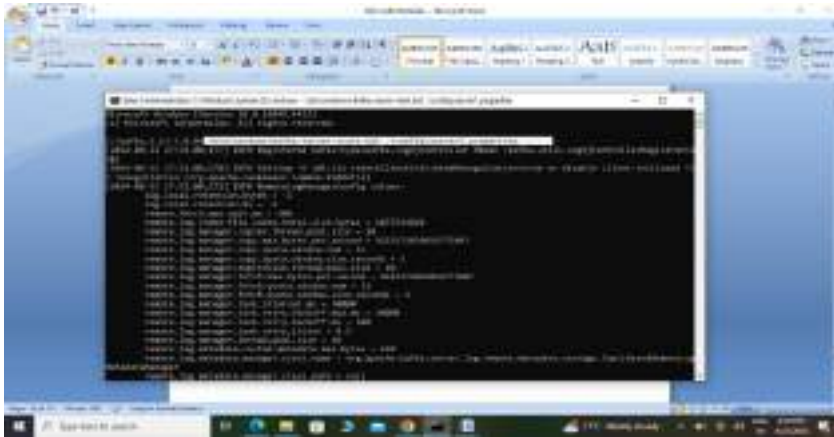
TYPE: `.\bin\windows\kafka-server-start.bat .\config\server.properties`



STEP 6: Open A NEW CMD and enter into the path C:\kafka_2.13-3.8.0

PATH: C:\kafka_2.13-3.8.0

TYPE: .\bin\windows\kafka-server-start.bat .\config\server1.properties



STEP 7: Open A NEW CMD and enter into the path C:\kafka_2.13-3.8.0
 PATH: C:\kafka_2.13-3.8.0

TYPE: .\bin\windows\kafka-server-start.bat .\config\server2.properties



STEP 8 : Check the multi brokers ids with given link which can connecting to localhost:2181
 PATH: C:\kafka_2.13-3.8.0\bin\windows

TYPE: .\bin\windows\zookeeper-shell.bat localhost:2181 ls /brokers/ids



➔ It Contains 3 Brokers [0,1,2]

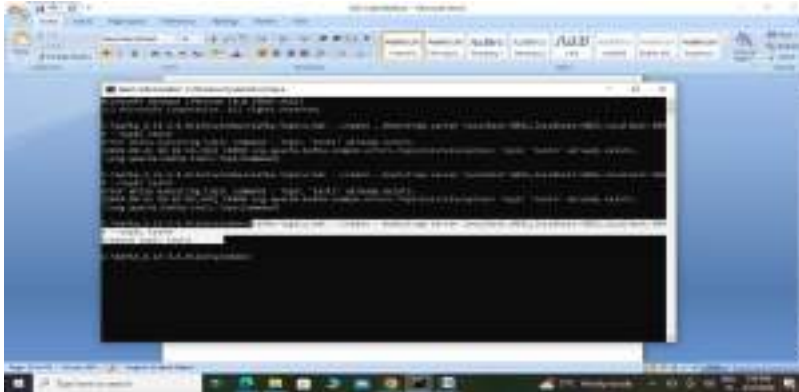
STEP 9 : Checking of multi broker server

STEP 10: open cmd create topic test2

PATH: C:\kafka_2.13-3.8.0\bin\windows

TYPE:

kafka-topics.bat --create --bootstrap-server localhost:9092,localhost:9093,localhost:9094 --topic test2



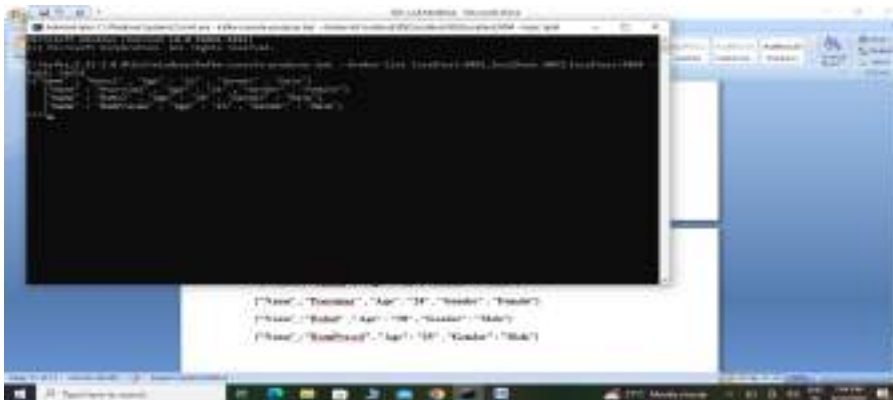
STEP 11 : Open producer on another cmd

PATH: C:\kafka_2.13-3.8.0\bin\windows

TYPE:

kafka-console-producer.bat --broker-list localhost:9092,localhost:9093,localhost:9094 --topic test2

```
>> {"Name" : "Manoj" , "Age" : "22" , "Gender" : "Male"}  
{"Name" : "Poornima" , "Age" : "24" , "Gender" : "Female"}  
{"Name" : "Rahul" , "Age" : "28" , "Gender" : "Male"}  
{"Name" : "RamPrasad" , "Age" : "25" , "Gender" : "Male"}
```



STEP 12: Open consumer on another cmd

PATH: C:\kafka_2.13-3.8.0\bin\windows

TYPE: kafka-console-consumer.bat --topic test2 --bootstrap-server localhost:9092,localhost:9093,localhost:9094 --from-beginning



EXPERIMENT NO: 3

NAME OF THE EXPERIMENT: Exploring Zookeeper.

STEP 1: enter into c drive → kafka_2.13-3.8.0 → C:\kafka_2.13-3.8.0\config → open server

log.dirs=C:\kafka_2.13-3.8.0\kafka-logs (replace temp with C:\kafka_2.13-3.8.0) and SAVE the file

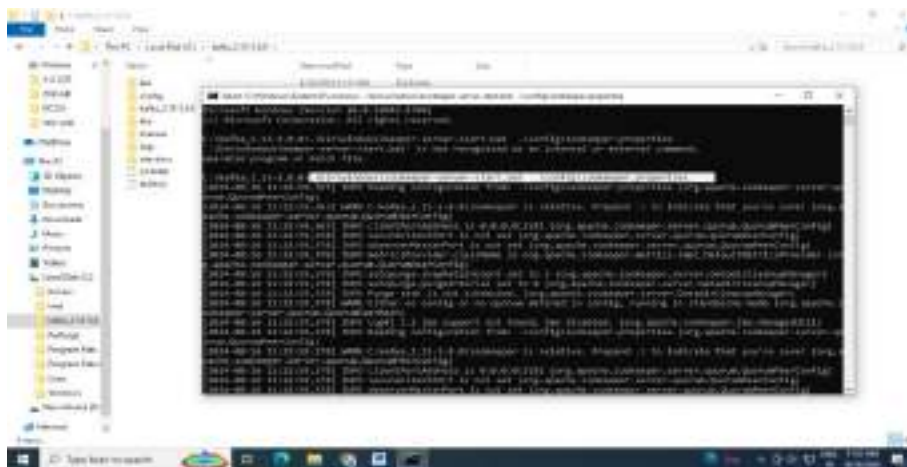
STEP 2: Enter into c drive → kafka_2.13-3.8.0 → C:\kafka_2.13-3.8.0\config → open zookeeper

dataDir=C:\kafka_2.13-3.8.0\zookeeper (replace temp with C:\kafka_2.13-3.8.0) and SAVE the file

STEP 3: Open cmd and enter into the path C:\kafka_2.13-3.8.0

PATH: C:\kafka_2.13-3.8.0

TYPE: .\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties

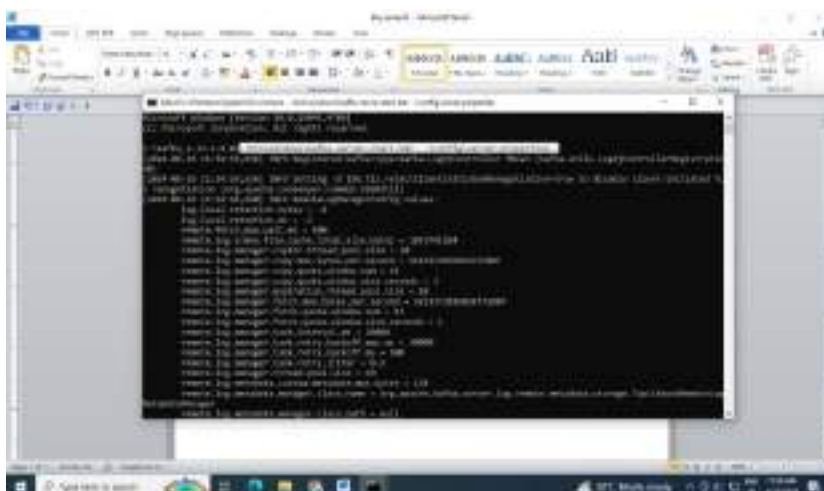


NOTE: DON'T CLOSE CMD

STEP 4: Open A NEW CMD and enter into the path C:\kafka_2.13-3.8.0

PATH: C:\kafka_2.13-3.8.0

TYPE: .\bin\windows\kafka-server-start.bat .\config\server.properties



EXPERIMENT NO 4

NAME OF THE EXPERIMENT: Overview of kafka consumers

Create a Kafka Topic:

1. Open a new command prompt in the location C:\kafka\bin\windows.
2. Run the following command:

Output: The topic test created successfully.

Creating Kafka Producer:

1. Open a new command prompt in the location C:\kafka\bin\windows
2. Run the following command:

```
C:\kafka\bin\windows>kafka-console-producer.bat --broker-list  
localhost:9092 --topic test
```

The Kafka Producer is executing sending the messages to consumer in a separate window.

Creating Kafka Consumer:

1. Open a new command prompt in the location C:\kafka\bin\windows.
2. Run the following command:

```
kafka-console-producer.bat --broker-list localhost:9092 --topic test  
C:\kafka\bin\windows>kafka-topics.bat --create --bootstrap-server localhost:9092 --topic test
```

EXPERIMENT NO: 5

NAME OF THE EXPERIMENT: Using Kafka with Docker.

STEP 1: INSTALL DOCKER

- Download latest docker from <https://docs.docker.com/desktop/install/windows-install/>
- Follow normal installation steps to install docker desktop





STEP 2: Create docker-compose.yml file

First Create a folder called kafka-windows-docker-main

Inside this folder create a .yml file called docker-compose.yml

docker-compose.yml

```
version: '2.1'
```

```
services:
```

```
zookeep1:
```

```
image: zookeeper:3.4.14
```

```
hostname: zookeep1
```

```
ports:
```

```
- "2181:2181"
```

```
environment:
```

```
ZOO_MY_ID: 1
```

```
ZOO_PORT: 2181
```

```
ZOO_SERVERS: server.1=zookeep1:2888:3888 server.2=zookeep2:2888:3888 server.3=zookeep3:2888:3888
```

```
volumes:
```

```
- ./kafka-windows-docker-data/zookeep1/data:/data
```

```
- ./kafka-windows-docker-data/zookeep1/datalog:/datalog
```

```
zookeep2:
```

```
image: zookeeper:3.4.14
```

```
hostname: zookeep2
```

```
ports:
```

```
- "2182:2182"
```

```
environment:
```

```
ZOO_MY_ID: 2
```

```
ZOO_PORT: 2182
```

```
ZOO_SERVERS: server.1=zookeep1:2888:3888 server.2=zookeep2:2888:3888 server.3=zookeep3:2888:3888
```

```
volumes:
```

```
- ./kafka-windows-docker-data/zookeep2/data:/data
```

```
- ./kafka-windows-docker-data/zookeep2/datalog:/datalog
```

zookeeper3:

image: zookeeper:3.4.14

hostname: zookeeper3

ports:

- "2183:2183"

environment:

ZOO_MY_ID: 3

ZOO_PORT: 2183

ZOO_SERVERS: server.1=zookeeper1:2888:3888 server.2=zookeeper2:2888:3888

server.3=zookeeper3:2888:3888

volumes:

- ./kafka-windows-docker-data/zookeeper3/data:/data
- ./kafka-windows-docker-data/zookeeper3/datalog:/datalog

kafka1:

image: confluentinc/cp-kafka:7.5.0

hostname: kafka1

ports:

- "9092:9092"

environment:

KAFKA_ADVERTISED_LISTENERS:

LISTENER_DOCKER_INTERNAL://kafka1:19092,LISTENER_DOCKER_EXTERNAL://{DOCKER_HOST_IP:-127.0.0.1}:9092

KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:

LISTENER_DOCKER_INTERNAL:PLAINTEXT,LISTENER_DOCKER_EXTERNAL:PLAINTEXT

KAFKA_INTER_BROKER_LISTENER_NAME: LISTENER_DOCKER_INTERNAL

KAFKA_ZOOKEEPER_CONNECT: "zookeeper1:2181,zookeeper2:2182,zookeeper3:2183"

KAFKA_BROKER_ID: 1

KAFKA_LOG4J_LOGGERS:

"kafka.controller=INFO,kafka.producer.async.DefaultEventHandler=INFO,state.change.logger=INFO"

volumes:

- ./kafka-windows-docker-data/kafka1/data:/var/lib/kafka/data

depends_on:

- zookeeper1
- zookeeper2
- zookeeper3

kafka2:

image: confluentinc/cp-kafka:7.5.0

hostname: kafka2

ports:

- "9093:9093"

environment:

KAFKA_ADVERTISED_LISTENERS:

LISTENER_DOCKER_INTERNAL://kafka2:19093,LISTENER_DOCKER_EXTERNAL://{DOCKER_HOST_IP:-127.0.0.1}:9093

KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:

LISTENER_DOCKER_INTERNAL:PLAINTEXT,LISTENER_DOCKER_EXTERNAL:PLAINTEXT

KAFKA_INTER_BROKER_LISTENER_NAME: LISTENER_DOCKER_INTERNAL

KAFKA_ZOOKEEPER_CONNECT: "zookeeper1:2181,zookeeper2:2182,zookeeper3:2183"

KAFKA_BROKER_ID: 2

KAFKA_LOG4J_LOGGERS:

"kafka.controller=INFO,kafka.producer.async.DefaultEventHandler=INFO,state.change.logger=INFO"

volumes:

- ./kafka-windows-docker-data/kafka2/data:/var/lib/kafka/data

depends_on:

- zookeep1
- zookeep2
- zookeep3

kafka3:

image: confluentinc/cp-kafka:7.5.0

hostname: kafka3

ports:

- "9094:9094"

environment:

KAFKA_ADVERTISED_LISTENERS:

LISTENER_DOCKER_INTERNAL://kafka3:19094,LISTENER_DOCKER_EXTERNAL://{DOCKER_HOST_IP:-127.0.0.1}:9094

KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:

LISTENER_DOCKER_INTERNAL:PLAINTEXT,LISTENER_DOCKER_EXTERNAL:PLAINTEXT

KAFKA_INTER_BROKER_LISTENER_NAME: LISTENER_DOCKER_INTERNAL

KAFKA_ZOOKEEPER_CONNECT: "zookeeper1:2181,zookeeper2:2182,zookeeper3:2183"

KAFKA_BROKER_ID: 3

KAFKA_LOG4J_LOGGERS:

"kafka.controller=INFO,kafka.producer.async.DefaultEventHandler=INFO,state.change.logger=INFO"

volumes:

- ./kafka-windows-docker-data/kafka3/data:/var/lib/kafka/data

depends_on:

- zookeep1
- zookeep2
- zookeep3

STEP 3: Install kafka and zookeeper as per the configuration in docker file
docker-compose up -d — build



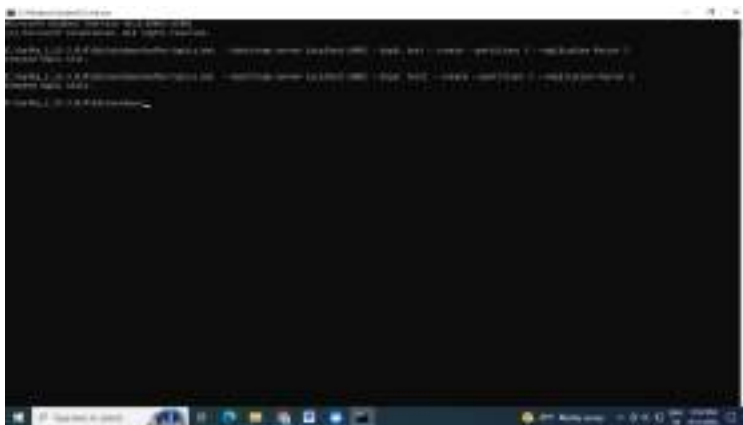
STEP 4: Creating Topics

Create new topic with topic Name “test” with 3 partitions and 3 replication-factor

```
C:\kafka_2.13-3.8.0\bin\windows>kafka-topics.bat --bootstrap-server localhost:9092 --topic test --create --  
partitions 3 --replication-factor 3
```

Create new topic with topicName “test1” with 3 partitions and 3 replication-factor

```
C:\kafka_2.13-3.8.0\bin\windows>kafka-topics.bat --bootstrap-server localhost:9092 --topic test1 --create --  
partitions 3 --replication-factor 3
```

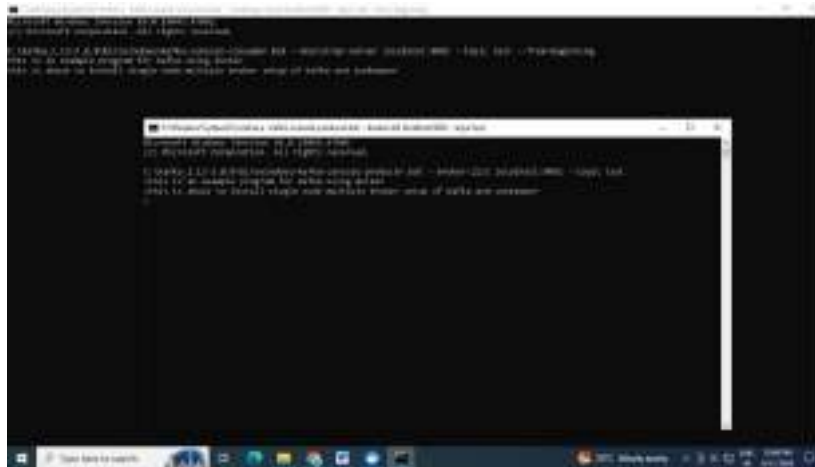


STEP 5 :To List down the topics

```
C:\kafka_2.13-3.8.0\bin\windows>kafka-topics.bat --bootstrap-server localhost:9092 --list
```


STEP 8: Consume messages from beginning

```
C:\kafka_2.13-3.8.0\bin\windows>kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic test --from-beginning
```



EXPERIMENT NO: 6

NAME OF THE EXPERIMENT: Developing Kafka Producer.

STEP 1: enter into c drive `cd kafka_2.13-3.8.0` `C:\kafka_2.13-3.8.0\config` open server

log.dirs=C:\kafka_2.13-3.8.0\kafka-logs (replace temp with C:\kafka_2.13-3.8.0) and SAVE the file

STEP 2: Enter into c drive `cd`

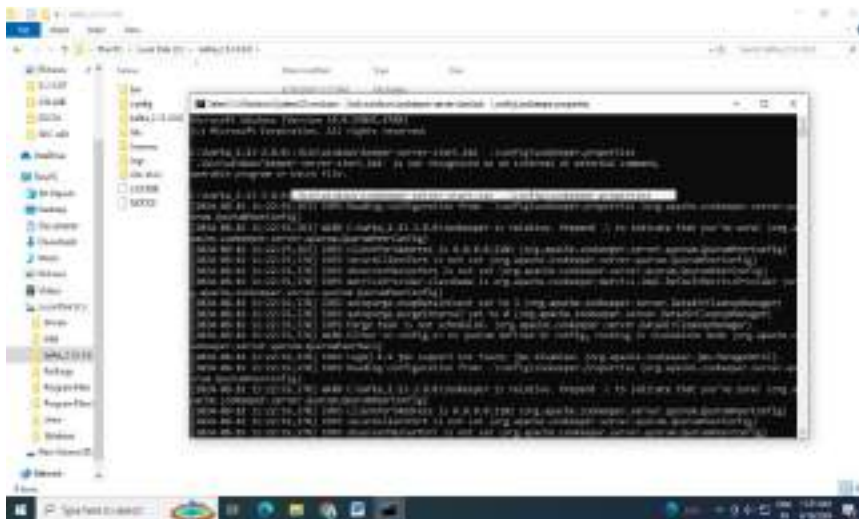
kafka_2.13-3.8.0 `cd C:\kafka_2.13-3.8.0\config` open zookeeper
dataDir=C:\kafka_2.13-3.8.0\zookeeper (replace temp with C:\kafka_2.13-3.8.0) and SAVE the file

STEP 3: Open cmd and enter into the path C:\kafka_2.13-3.8.0

PATH: C:\kafka_2.13-3.8.0

TYPE: `.\bin\windows\zookeeper-server-start.bat`

`.\config\zookeeper.properties`



NOTE: DON'T CLOSE CMD

STEP 4: Open A NEW CMD and enter into the path C:\kafka_2.13-3.8.0

PATH: C:\kafka_2.13-3.8.0

TYPE: `.\bin\windows\kafka-server-start.bat .\config\server.properties`

EXPERIMENT NO: 7

NAME OF THE EXPERIMENT: Sending Messages with Callback.

STEP 1: Open Google chrome and type “dependency download” and open the below link



Maven Repository
<https://mvnrepository.com/artifact/org.apache.maven.plugins/maven-dependency-plugin>

Apache Maven Dependency Plugin

Apache Maven Dependency Plugin provides utility goals to work with dependencies like copying, unpacking, analyzing, resolving and many more.

STEP 2: Search in the search bar as “kafka-client” then click on “org.apache.kafka>>kafka-client “ in Apache-kafka

The screenshot shows the Maven Repository search results for 'kafka-client'. The search bar at the top is circled in red. Below the search bar, the results are displayed in a table. The first result is 'Kafka Schema Registry Client' by io.confluent. The second result is 'Vert.x Kafka Client' by io.vertx. The third result is 'Quarkus Kafka Client Runtime' by io.quarkus. The fourth result is 'Apache Kafka' by org.apache.kafka, which is circled in red. The table also shows the number of usages for each artifact.

Artifact	Group ID	Artifact ID	Version	Usages
Kafka Schema Registry Client	io.confluent	kafka-schema-registry-client	7.0.0	164 usages
Vert.x Kafka Client	io.vertx	vertx-kafka-client	4.4.0	70 usages
Quarkus Kafka Client Runtime	io.quarkus	quarkus-kafka-client	2.13.0	22 usages
Apache Kafka	org.apache.kafka	kafka-clients	3.6.0	2,388 usages

❖ Select 3.7.0 version

Home » org.apache.kafka » kafka-clients

Apache Kafka
Apache Kafka

License: [Apache 2.0](#)

Categories: [Messaging](#) [Queue](#) [Kafka](#) [Kinesis](#) [Client](#) [Message](#)

Tags: [#224 in MavenRepository \(See Top Artifacts\)](#) [#1 in Message Queue Clients](#)

Ranking: [#224 in MavenRepository \(See Top Artifacts\)](#) [#1 in Message Queue Clients](#)

Used By: 2,355 artifacts

Version	Vulnerabilities	Repository	Downloads	Date
3.6 = 3.6.0		Central	46	Jul 26, 2024
3.6 = 3.6.1		Central	46	Aug 20, 2024
3.6 = 3.7.0		Central	181	Feb 27, 2024
3.6 = 3.7.1		Central	46	Mar 04, 2024
3.6 = 3.6.1		Central	424	Dec 05, 2023
3.6 = 3.6.0		Central	344	Dec 04, 2023
3.6 = 3.6.2		Central	55	Mar 04, 2024

STEP 3: Select Gradle and click on jar(7.8 MB), then kafka-client dependencies are going to download

Home » Apache Kafka » 3.7.0

Apache Kafka

License: [Apache 2.0](#)

Categories: [Message Queue Clients](#)

Age: [streaming](#) [queue](#) [kafka](#) [apache](#) [client](#) [message](#)

Homepage: <https://kafka.apache.org>

Date: Feb 27, 2024

File: [pom \(1 KB\)](#) [jar \(7.8 MB\)](#) [View All](#)

Repositories: [Central](#)

Ranking: [#224 in MavenRepository \(See Top Artifacts\)](#) [#1 in Message Queue Clients](#)

Used By: 2,355 artifacts

Note: There is a new version for this artifact

New Version: [3.8.0](#)

Maven **Gradle** [Gradle \(Short\)](#) [Gradle \(Kotlin\)](#) [SBT](#) [Ivy](#) [Grape](#) [Lensing](#) [Buildr](#)

```
// https://mavenrepository.com/artifact/org.apache.kafka/kafka-clients
implementation group: 'org.apache.kafka', name: 'kafka-clients', version: '3.7.0'
```

STEP 4: :Again search in the search bar as “ org.slf4j ” then click on “org.slf4j >> slf4j-api” in the SLF4J API Module

SLF4J API Module
org.slf4j » slf4j-api 70,643 usages

API for SLF4J (The Simple Logging Facade Library) which serves as a simple facade or abstraction for various logging frameworks, allowing the end user to plug in the desired logging framework at deployment time.

Last Release on Aug 10, 2024

SLF4J LOG4J 1.2 Binding Relocated
org.slf4j » slf4j-log4j12 17,426 usages

SLF4J LOG4J 1.2 relocated to slf4j-reload4j

Last Release on Aug 10, 2024

Relocated - org.slf4j » slf4j-reload4j

SLF4J Simple Provider
org.slf4j » slf4j-simple 13,006 usages

Binding/provider for SLF4J which outputs all events to System.out. Only messages of level INFO and higher are printed.

Last Release on Aug 10, 2024

JCL 1.2 Implemented Over SLF4J
org.slf4j » jcl-over-slf4j 8,334 usages

jakarta/apache commons logging 1.2 implemented over SLF4J

Last Release on Aug 10, 2024

❖ Select 2.0.16 version

SLF4J API Module

API for SLF4J (The Simple Logging Facade for Java) which serves as a simple facade or abstraction for various logging frameworks, allowing the end user to plug in the desired logging framework at deployment time.

License: MIT

Categories: Logging Frameworks

Tags: logging, api, slf4j

Ranking: #2 in MavenRepository (See Top Artifacts)
#1 in Logging Frameworks

Used By: 70,643 artifacts

Central (105) | JBoss Repo (4) | Redhat GA (10) | Redhat EA (12) | Sonatype (1) | SonatypePuls (6)

Version	Vulnerabilities	Repository	Usages	Date
2.1.x				
2.1.0-alpha1		Central	98	Jan 02, 2024
2.1.0-alpha9		Central	10	Dec 28, 2023
2.0.16		Central	1,646	Aug 10, 2023
2.0.15		Central	104	Aug 08, 2023
2.0.14		Central	834	Aug 08, 2023

STEP 5: Select Gradle and click on jar, as like that search again as “org.slf4j ” then click on “org.slf4j >> slf4j-simple” in the SLF4J Simple Provider

SLF4J API Module
org.slf4j:slf4j-api

API for SLF4J (The Simple Logging Facade for Java) which serves as a simple facade or abstraction for various logging frameworks, allowing the end user to plug in the desired logging framework at deployment time.

Last Release on Aug 10, 2024

70,643 usages

SLF4J LOG4J 1.2 Binding Relocated
org.slf4j:slf4j-log4j12

SLF4J LOG4J 1.2 relocated to slf4j-reload4j

Last Release on Aug 10, 2024

Relocated -> org.slf4j:slf4j-reload4j

17,426 usages

SLF4J Simple Provider
org.slf4j:slf4j-simple

Binding/provider for SLF4J, which outputs all events to System.out. Only messages of level INFO and higher are printed.

Last Release on Aug 10, 2024

18,086 usages

JCL 1.2 Implemented Over SLF4J
org.apache.logging.log4j:log4j-core

Apache Commons logging 1.2 implemented over SLF4J

Last Release on Aug 10, 2024

9,354 usages

❖ Select 2.0.16 version

STEP 6: Select Gradle and click on jar

SLF4J Simple Provider

Binding/provider for SLF4J, which outputs all events to System.out. Only messages of level INFO and higher are printed.

License: MIT

Categories: Logging Frameworks

Tags: logging, slf4j

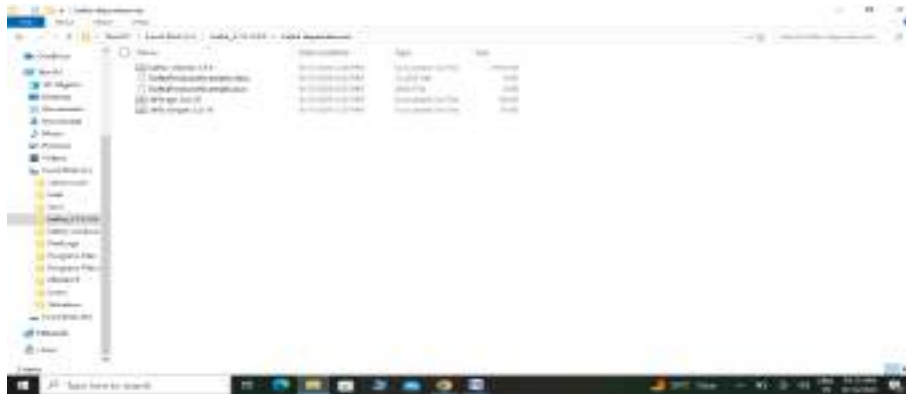
Ranking: #24 in MavenRepository (See Top Artifacts)
#4 in Logging Frameworks

Used By: 15,686 artifacts

Central (114) | JBoss Repo (3) | Redhat GA (33) | Redhat EA (11) | ICM (3)

Version	Vulnerabilities	Repository	Usages	Date
2.1.x				
2.1.0-alpha1		Central	9	Jan 02, 2024
2.1.0-alpha9		Central	3	Dec 28, 2023
2.0.16		Central	160	Aug 10, 2023
2.0.15		Central	135	Aug 08, 2023

STEP 7: Copy all the three dependencies from the downloads then create a new folder inside the kafka folder named as "kafka dependencies"



STEP 8: Write down the below code in the notepad and save it in the "kafka dependencies folder"

KafkaProducerExample.java

```
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;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.ArrayList;
import java.util.Properties;

public class KafkaProducerExample {

    private final static String TOPIC_NAME = "AnimalTopic";
    private final static String bootstrapServers = "localhost:9092"; // Example bootstrap server
    public static Logger logger = LoggerFactory.getLogger(KafkaProducerExample.class);

    public static void main(String[] args) {

        KafkaProducer<String, String> producer = null;
        try {

            // Create Producer properties
            Properties properties = new Properties();
            properties.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
            properties.setProperty(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
            properties.setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

            // Create the producer
            producer = new KafkaProducer<>(properties);

            ArrayList<String> animalList = getAnimalList();
```

```

int i = 0;

for (String animalName :animalList) {

    ++i;
    String key = "Id_" + i;
    String value = animalName; // Use animalName directly

    // Create a producer record
    ProducerRecord<String, String>producerRecord = new ProducerRecord<>(TOPIC_NAME, key, value);
    logger.info("key = " + key + ", value = " + value);

    // Send data - asynchronous
    producer.send(producerRecord, (RecordMetadata recordMetadata, Exception e) -> {
        if (e == null) {
            logger.info("Successfully received the details as: \n"
                + "Topic = " + recordMetadata.topic() + "\n"
                + "Partition = " + recordMetadata.partition() + "\n"
                + "Offset = " + recordMetadata.offset() + "\n"
                + "Timestamp = " + recordMetadata.timestamp());
        } else {
            logger.error("Can't produce, getting error ", e);
        }
    }).get(); // Make it synchronous for demo purposes

    System.out.println("Successfully sent the Animal name = '" + value + "' to the Topic");

    Thread.sleep(4000); // Simulating a delay for demo purposes
}

} catch (Exception exe) {
    exe.printStackTrace();
} finally {
    if (producer != null) {
        producer.flush();
        producer.close();
    }
}

}

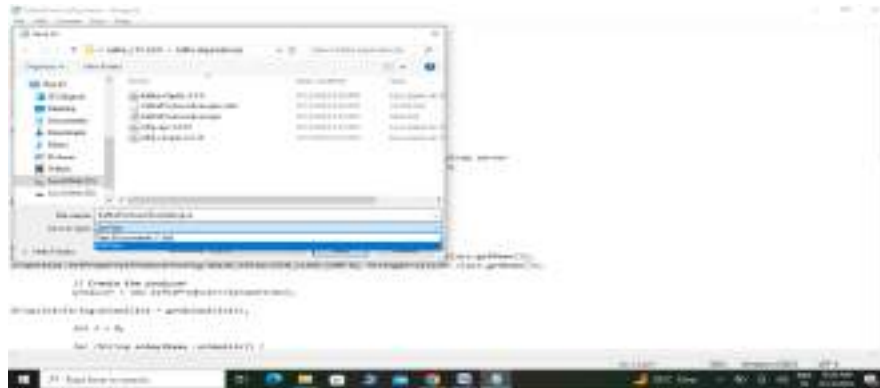
private static ArrayList<String>getAnimalList() {

    ArrayList<String>animalList = new ArrayList<>();
    animalList.add("DOG");
    animalList.add("LION");
    animalList.add("TIGER");
    animalList.add("SNAKE");
    animalList.add("CAT");

    return animalList;
}

```

}



STEP 9: Run Zookeeper and server in kafka by entering into the below command with two different cmd's (path--C:\kafka_2.13-3.8.0)

Command:

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



Command:

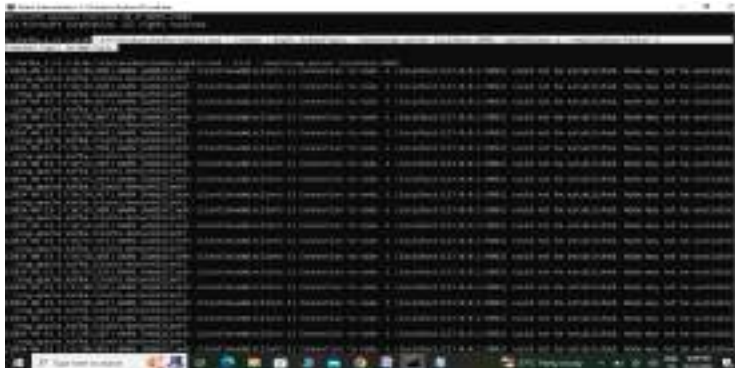
.bin\windows\kafka-server-start.bat .\config\server.properties



STEP 10: Creating a topic called AnimalTopic

Command:

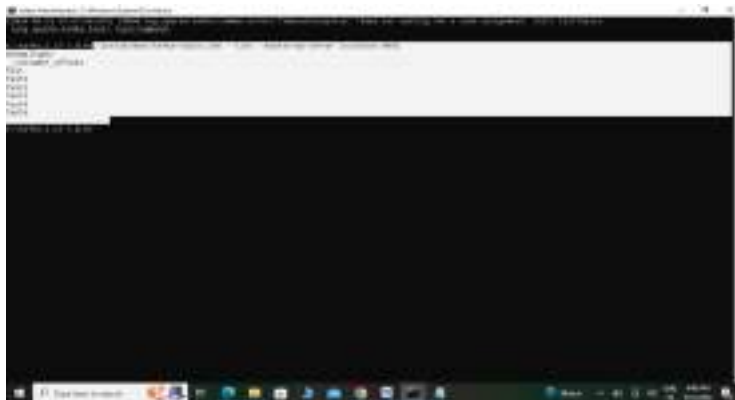
```
.\\bin\\windows\\kafka-topics.bat --create --topic AnimalTopic --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
```



STEP 11: To Listdown all the topics

Command:

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



STEP 12: Enter into the folder “kafka dependencies” by using the command “cd kafka dependencies”

Compilation of the Code:

```
javac -cp ".;kafka-clients-3.7.0.jar;slf4j-api-2.0.16.jar;slf4j-simple-2.0.16.jar" KafkaProducerExample.java
```



STEP 13:Run the Code using the below command

```
java -cp ".;kafka-clients-3.7.0.jar;slf4j-api-2.0.16.jar;slf4j-simple-2.0.16.jar" KafkaProducerExample
```

[illegible][illegible]

EXPERIMENT NO: 8

NAME OF THE EXPERIMENT: Kafka monitoring and schema Registry.

Objective:

1. Set up and configure Kafka monitoring to track Kafka metrics.
2. Set up and use Schema Registry to manage message schemas in Kafka.

Prerequisites

- Apache Kafka and Apache Zookeeper installed and running.
- Confluent Schema Registry installed (comes with Confluent Platform) or downloaded separately.
- Prometheus for monitoring metrics.
- Grafana for data visualization.
- Java Development Kit (JDK 8 or above).
- Kafka client libraries and Confluent libraries (for Schema Registry).

Start Kafka and Zookeeper:

Start Zookeeper:

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

Start Kafka:

```
bin/kafka-server-start.sh config/server.properties
```

Create a Kafka Topic for testing schemas (e.g., schema-topic):

```
bin/kafka-topics.sh --create --topic schema-topic --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
```

Configure Kafka Monitoring with Prometheus and Grafana

Install and Configure JMX Exporter for Kafka:

1. Download the JMX Exporter .jar file from Prometheus's website.
2. Place the .jar file in your Kafka directory and create a configuration file kafka-jmx-exporter.yml:

```
lowercaseOutputName: true
rules:
  - pattern: "kafka.server<type=(.+), name=(.+)>"
    name: "kafka_server_$1_$2"
    labels:
      instance: "kafka"
    help: "Kafka server metrics"
```

Edit the Kafka Server Configuration (config/server.properties) to enable JMX Exporter by adding the following line:

```
KAFKA_OPTS="-javaagent:/path/to/jmx_prometheus_javaagent.jar=8080:/path/to/kafka-jmx-exporter.yml"
```

Restart Kafka for the JMX exporter changes to take effect.

Set Up Prometheus to Scrape Kafka Metrics:

Edit the prometheus.yml configuration file to add Kafka as a target:

scrape_configs:

- job_name: 'kafka'

static_configs:

- targets: ['localhost:8080']

Start Prometheus:

./prometheus --config.file=prometheus.yml

Set Up Grafana and Add Prometheus as a Data Source

1. Download and install Grafana if not already installed.
2. Open Grafana at <http://localhost:3000> and log in.
3. Add Prometheus as a Data Source.
4. Import or create Kafka dashboards to monitor Kafka metrics (e.g., throughput, consumer lag, etc.).

Set Up Schema Registry

Start Schema Registry: In the Confluent Platform, Schema Registry can be started using the command below. If using a standalone Schema Registry, adjust the paths accordingly

bin/schema-registry-start config/schema-registry.properties

Ensure Schema Registry is accessible on <http://localhost:8081>.

Check Schema Registry:

Verify the Schema Registry is running by visiting <http://localhost:8081/subjects> in a browser or with a curl command:

```
curl -X GET http://localhost:8081/subjects
```

Define and Register a Schema

Create a Schema (e.g., a JSON file named transaction-schema.avsc):

```
{  
  
  "type": "record",  
  
  "name": "Transaction",  
  
  "namespace": "com.example",  
  
  "fields": [  
  
    {"name": "id", "type": "int"},
```

```

    {"name": "amount", "type": "double"},

    {"name": "user_id", "type": "int"},

    {"name": "timestamp", "type": "string"}

]

}

```

Register the Schema with Schema Registry:

Use the curl command to register the schema:

```

curl -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" \

--data '{"schema":

"{\"type\":\"record\",\"name\":\"Transaction\",\"fields\": [{\"name\":\"id\",\"type\":\"int\"}, {\"name\":\"amount\",\"type\":\"double\"}, {\"name\":\"user_id\",\"type\":\"int\"}, {\"name\":\"timestamp\",\"type\":\"string\"}]}"' \

http://localhost:8081/subjects/schema-topic-value/versions

```

Verify the Schema Registration:

Check if the schema was successfully registered:

```

curl -X GET http://localhost:8081/subjects/schema-topic-value/versions

```

Produce and Consume Messages with Schema Validation:

Write a Producer to Send Messages Using the Schema

Create a new Java file, SchemaProducer.java, with the following code:

```

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import io.confluent.kafka.serializers.KafkaAvroSerializer;
import org.apache.avro.generic.GenericData;
import org.apache.avro.generic.GenericRecord;

import java.util.Properties;

public class SchemaProducer {
    public static void main(String[] args) {
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.put("key.serializer", KafkaAvroSerializer.class.getName());
        props.put("value.serializer", KafkaAvroSerializer.class.getName());
        props.put("schema.registry.url", "http://localhost:8081");
    }
}

```

```

KafkaProducer<String, GenericRecord> producer = new KafkaProducer<>(props);

String topic = "schema-topic";
GenericRecord transaction = new GenericData.Record(schema);

transaction.put("id", 1);
transaction.put("amount", 150.0);
transaction.put("user_id", 1001);
transaction.put("timestamp", "2024-11-01T10:00:00Z");

ProducerRecord<String, GenericRecord> record = new ProducerRecord<>(topic, transaction);

producer.send(record, (RecordMetadata metadata, Exception exception) -> {
    if (exception == null) {
        System.out.println("Message sent successfully to " + metadata.topic());
    } else {
        exception.printStackTrace();
    }
});

producer.close();
}
}

```

Run the Producer: This will produce messages using the schema registered in Schema Registry.

Verify Messages Using Kafka Consumer

Create a Kafka consumer to read from the topic:

```
bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic schema-topic --from-beginning
```

Monitor Messages in Grafana: Check your Kafka Grafana dashboard for the latest metrics, including throughput and consumer lag.

EXPERIMENT NO: 9

NAME OF THE EXPERIMENT: Kafka streams and kafka connectors.

Kafka Streams application that reads messages from a Kafka topic, transforms the data (in this case, converting the message to uppercase), and writes the transformed data to another Kafka topic.

Prerequisites:

- Kafka cluster is running.
- Kafka topics input-topic and output-topic are created.
- Dependencies for Kafka Streams (e.g., org.apache.kafka:kafka-streams in your pom.xml for Maven).

Kafka Streams Example:

```
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.Consumed;
import org.apache.kafka.streams.kstream.Produced;
import java.util.Properties;

public class KafkaStreamsExample {

    public static void main(String[] args) {

        // Set up Kafka Streams configuration
        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "uppercase-stream-app");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass().getName());
        // Define the stream processing logic
        KStream<String, String> inputStream = new KafkaStreams(props)
            .stream("input-topic", Consumed.with(Serdes.String(), Serdes.String()))
            .mapValues(value -> value.toUpperCase()); // Transform value to uppercase
        // Write the processed stream to the output topic
        inputStream.to("output-topic", Produced.with(Serdes.String(), Serdes.String()));
        // Start the Kafka Streams application
        KafkaStreams streams = new KafkaStreams(inputStream, props);
        streams.start();

        // Shutdown hook to close Kafka Streams
```

```

    Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
}
}

```

How to Run:

1. Start your Kafka cluster.
2. Create Kafka topics input-topic and output-topic.
3. Run the Kafka Streams application.
4. Send messages to input-topic (e.g., using Kafka Console Producer).
5. Check output-topic for uppercase transformed messages.

Kafka Connect JDBC Source Connector:

Kafka Connect uses connectors to integrate with external systems. In this example, we'll set up a JDBC Source Connector to read data from a relational database (e.g., MySQL) and stream it into a Kafka topic.

Prerequisites:

- A running Kafka cluster.
- A relational database (e.g., MySQL) with a sample table.
- Kafka Connect with the JDBC source connector configured.

Steps:

1. Set up the JDBC source connector in Kafka Connect (via REST API or properties files).
2. Configure the JDBC Source Connector to pull data from the database.

JDBC Source Connector Configuration (via REST API):

First, let's assume you have a database test_db and a table users with the following structure:

```

CREATE TABLE users (
  id INT PRIMARY KEY,
  name VARCHAR(100),
  email VARCHAR(100)
);

```

Now, configure the JDBC Source Connector to pull data from this users table.

```

{
  "name": "jdbc-source-connector",
  "config": {
    "connector.class": "io.confluent.connect.jdbc.JdbcSourceConnector",
    "tasks.max": "1",
    "connection.url": "jdbc:mysql://localhost:3306/test_db",
    "connection.user": "root",
    "connection.password": "password",
    "topic.prefix": "users_",

```

```

    "table.whitelist": "users",
    "mode": "bulk",
    "poll.interval.ms": "1000"
  }
}

```

How to Run:

1. Start Kafka Connect in distributed mode.
2. Post the connector configuration via REST to the Kafka Connect cluster.

```
curl -X POST -H "Content-Type: application/json" --data @jdbc-source-config.json http://localhost:8083/connectors
```

- 3 Kafka Connect will start reading the users table and stream the data to Kafka topics (users_users in this case).
4. Use a Kafka consumer (e.g., kafka-console-consumer) to consume the data from the generated topic.

```
kafka-console-consumer --bootstrap-server localhost:9092 --topic users_users --from-beginning
```

Kafka Connect JDBC Sink Connector:

In this example, we'll configure a JDBC Sink Connector to write data from Kafka topics to a relational database (e.g., MySQL).

JDBC Sink Connector Configuration (via REST API):

Here's how you configure a JDBC Sink Connector to write data from Kafka to the users table in a MySQL database.

```

{
  "name": "jdbc-sink-connector",
  "config": {
    "connector.class": "io.confluent.connect.jdbc.JdbcSinkConnector",
    "tasks.max": "1",
    "connection.url": "jdbc:mysql://localhost:3306/test_db",
    "connection.user": "root",
    "connection.password": "password",
    "topics": "users-topic",
    "insert.mode": "insert",
    "auto.create": "true",
    "auto.evolve": "true",
    "pk.mode": "none",
    "batch.size": "1000"
  }
}

```

1. Start Kafka Connect in distributed mode.
2. Post the connector configuration via REST to the Kafka Connect cluster.

```
curl -X POST -H "Content-Type: application/json" --data @jdbc-sink-config.json http://localhost:8083/connectors
```

3. Kafka Connect will start consuming records from the users-topic Kafka topic and insert them into the users table in the MySQL database.

Comparison of Kafka Streams and Kafka Connect:

Feature	Kafka Streams	Kafka Connect
Purpose	Real-time stream processing in the Kafka ecosystem.	Integration of external systems (data sources and sinks) with Kafka.
Usage	Java client library for building stream processing applications.	Framework for connecting Kafka with external systems using connectors.
Data Flow	Processes data <i>within</i> Kafka topics (consuming and producing).	Moves data <i>in and out</i> of Kafka topics to/from external systems.
Operations	Complex stream processing (filtering, aggregating, joining).	Simple data transfer between Kafka and external systems.
State	Supports both stateless and stateful processing.	Stateless, focuses on data movement between systems.
Deployment	Embedded in applications as a library.	Can run in standalone or distributed mode.
Examples of Use Cases	Real-time analytics, ETL jobs, event-driven processing.	Data ingestion, data export, CDC, file system integration.

EXPERIMENT NO: 10

NAME OF THE EXPERIMENT: Integration of kafka with storm.

Step-by-Step Example of Kafka + Storm Integration in IntelliJ IDEA

Step 1: Set Up IntelliJ IDEA Project

1. Create a New Project in IntelliJ IDEA:
 - o Open IntelliJ IDEA.
 - o Click on File → New → Project.
 - o Select Java as the project type.
 - o Click Next and follow the steps to create the project.
2. Set Up the Project SDK:
 - o Make sure the JDK is correctly configured for your project (for example, JDK 8 or JDK 11).
 - o Click Next and then Finish to create the project.

Step 2: Download Kafka and Storm JAR Files

1. Download Apache Kafka JAR:
 - o Go to Kafka Maven Repository and download the appropriate Kafka JAR (e.g., kafka-clients-3.1.0.jar).
2. Download Apache Storm JAR:
 - o Go to Storm Maven Repository and download the Storm Core JAR (e.g., storm-core-2.3.0.jar).
3. Download Kafka Spout JAR:
 - o Go to Storm Kafka Spout Repository and download the Storm Kafka Spout JAR (e.g., storm-kafka-client-2.3.0.jar).
4. Download Other Dependencies:
 - o You will also need other dependencies like slf4j (Simple Logging Facade for Java) and log4j, which Kafka and Storm depend on.

Step 3: Add JAR Files to IntelliJ IDEA Project

1. Create a lib Directory:
 - o Inside your project, create a folder named lib (e.g., ProjectName/lib).
 - o Move all the JAR files you downloaded into this lib directory.
2. Add JARs to Classpath:
 - o Right-click on your project in IntelliJ and select Open Module Settings (or press F4).
 - o Go to Modules → Dependencies tab.
 - o Click the + button and choose JARs or directories.
 - o Select the lib folder you created and click OK.
3. Check the Module SDK:
 - o Make sure the module SDK is set correctly (e.g., Java 11 or Java 8).

Step 4: Write the Kafka + Storm Integration Code

Now, let's create a simple Kafka-Spout and Storm-Bolt example.

Directory Structure:

Css:

```
src
├── main
│   └── java
│       ├── com
│       │   └── example
│       │       └── KafkaStormExample.java
```

```
lib
├── kafka-clients-3.1.0.jar
├── storm-core-2.3.0.jar
├── storm-kafka-client-2.3.0.jar
├── slf4j-api.jar
└── log4j.jar
```

KafkaStormExample.java

Java:

```
package com.example;
```

```
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.apache.storm.kafka.spout.KafkaSpout;
import org.apache.storm.kafka.spout.KafkaSpoutConfig;
import org.apache.storm.kafka.bolt.KafkaBolt;
import org.apache.storm.kafka.bolt.selector.DefaultTopicSelector;
import org.apache.storm.topology.TopologyBuilder;
import org.apache.storm.topology.BasicOutputCollector;
import org.apache.storm.topology.OutputFieldsDeclarer;
import org.apache.storm.topology.base.BaseBasicBolt;
import org.apache.storm.tuple.Tuple;
import org.apache.storm.tuple.Values;
import org.apache.storm.LocalCluster;
import org.apache.storm.Config;
```

```
import java.util.HashMap;
import java.util.Map;
```

```
public class KafkaStormExample {
```

```
    // Bolt to process the message from Kafka
```

```
    public static class ProcessMessageBolt extends BaseBasicBolt {
```

```

@Override
public void execute(Tuple tuple, BasicOutputCollector collector) {
    // Extract message from the tuple
    String message = tuple.getStringByField("message");

    // Process the message (convert it to uppercase)
    String processedMessage = message.toUpperCase();

    // Emit the processed message to the next bolt (Kafka Producer)
    collector.emit(new Values(processedMessage));
}

@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new org.apache.storm.tuple.Fields("processed-message"));
}
}

public static void main(String[] args) throws Exception {
    // Kafka configuration
    Map<String, Object> kafkaProps = new HashMap<>();
    kafkaProps.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    kafkaProps.put(ConsumerConfig.GROUP_ID_CONFIG, "storm-consumer-group");
    kafkaProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
    kafkaProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());

    // Kafka Spout Configuration
    KafkaSpoutConfig<String, String> kafkaSpoutConfig = KafkaSpoutConfig
        .builder("localhost:9092", "input-topic") // Kafka broker and topic name
        .setProp(kafkaProps)
        .setStartOffset(KafkaSpoutConfig.StartOffset.LATEST) // Start reading from the latest message
        .build();

    // Topology builder
    TopologyBuilder builder = new TopologyBuilder();

    // Set up the Kafka Spout
    builder.setSpout("kafka-spout", new KafkaSpout<>(kafkaSpoutConfig), 1);

    // Set up the ProcessMessageBolt to process messages from Kafka
    builder.setBolt("process-bolt", new ProcessMessageBolt(), 1).shuffleGrouping("kafka-spout");

    // Set up KafkaProducerBolt to send processed messages to Kafka
    builder.setBolt("kafka-producer-bolt", new KafkaBolt<String, String>()
        .withProducerProperties(kafkaProps)

```

```

        .withTopicSelector(new DefaultTopicSelector("output-topic"), 1)
        .shuffleGrouping("process-bolt");

// Configure the Storm topology
Config config = new Config();
config.setDebug(true);

// Submit the topology to the Storm cluster (LocalCluster for local testing)
LocalCluster cluster = new LocalCluster();
cluster.submitTopology("kafka-storm-topology", config, builder.createTopology());

// Wait for a while to process messages
Thread.sleep(10000);

// Shutdown the cluster after processing
cluster.shutdown();
}
}

```

Explanation of the Code:

1. KafkaSpout: Reads messages from Kafka topic input-topic and emits them as tuples.
2. ProcessMessageBolt: Converts each message to uppercase (as a simple transformation) and emits the processed message.
3. KafkaBolt: Sends the processed message to a Kafka topic output-topic.

Step 5: Set Up Kafka Broker and Topics

1. Start Kafka Broker:
 - o Download and start a Kafka broker if you don't have one running already.
 - o Follow the [Kafka quick start guide](#) to set up and start Kafka locally.
2. Create Kafka Topics:
 - o Create two Kafka topics using the Kafka command line tools:

```
bash
```

```
Copy code
```

```
kafka-topics --create --topic input-topic --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
```

```
kafka-topics --create --topic output-topic --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
```

3. Produce Messages to input-topic:
 - o Use the Kafka producer tool to send test messages to the input-topic:

```
kafka-console-producer --broker-list localhost:9092 --topic input-topic
```

Type a few test messages, like:

```
hello  
world  
storm
```

Step 6: Run the Project in IntelliJ IDEA

1. Run the KafkaStormExample.java:

- o In IntelliJ IDEA, click on the green Run button or right-click on the KafkaStormExample.java class and select Run 'KafkaStormExample'.
- o The topology will start and begin reading from the input-topic, processing the messages, and writing the processed messages to output-topic.

2. Consume from the output-topic:

- o Use the Kafka consumer tool to check the processed messages:

```
kafka-console-consumer --bootstrap-server localhost:9092 --topic output-  
topic --from-beginning
```

- o You should see the messages like:

```
HELLO  
WORLD  
STORM
```

EXPERIMENT NO 11

NAME OF THE EXPERIMENT: Kafka integration with spark and flume.

Integrating Kafka, Spark, and Flume:

Objective:

To set up an end-to-end data streaming pipeline that uses Flume to ingest data, Kafka as a message broker, and Spark for real-time processing.

Prerequisites

1. Apache Kafka and Apache Zookeeper installed.
2. Apache Flume and Apache Spark installed.
3. Java Development Kit (JDK 8 or above).
4. Scala installed for the Spark job.
5. IntelliJ IDEA or any IDE with Scala support.

Setup

1. Kafka will serve as the message broker.
2. Flume will capture and send data (e.g., log entries) to Kafka.
3. Spark will consume messages from Kafka and process them.

Set Up Kafka :

Start Zookeeper: `bin/zookeeper-server-start.sh config/zookeeper.properties`

Start Kafka: `bin/kafka-server-start.sh config/server.properties`

Create Kafka Topic (e.g., flume-kafka-topic):

`bin/kafka-topics.sh --create --topic flume-kafka-topic --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1`

Configure Flume to Send Data to Kafka:

- Create a configuration file for Flume named flume-kafka.conf.
- Copy the following configuration into flume-kafka.conf:

Define the agent, source, sink, and channel

`agent.sources = source1`

`agent.channels = channel1`

`agent.sinks = kafka-sink`

Configure the source to monitor a log file

`agent.sources.source1.type = exec`

`agent.sources.source1.command = tail -F /path/to/your/logfile.log`

`agent.sources.source1.channels = channel1`

Configure the channel as a memory channel

`agent.channels.channel1.type = memory`

`agent.channels.channel1.capacity = 1000`

```
agent.channels.channel1.transactionCapacity = 100
```

Configure Kafka as the sink

```
agent.sinks.kafka-sink.type = org.apache.flume.sink.kafka.KafkaSink
agent.sinks.kafka-sink.kafka.bootstrap.servers = localhost:9092
agent.sinks.kafka-sink.kafka.topic = flume-kafka-topic
agent.sinks.kafka-sink.channel = channel1
agent.sinks.kafka-sink.kafka.producer.acks = 1
```

Run Flume with the configuration file:

```
bin/flume-ng agent --conf conf --conf-file flume-kafka.conf --name agent -Dflume.root.logger=INFO,console
```

Create a Spark Application to Consume Data from Kafka:

Add Dependencies for Kafka and Spark Streaming (if using Scala):

➤ In build.sbt for SBT:

```
libraryDependencies += "org.apache.spark" %% "spark-streaming" % "3.0.1"
libraryDependencies += "org.apache.spark" %% "spark-streaming-kafka-0-10" % "3.0.1"
```

In pom.xml for Maven:

```
<!-- Spark and Kafka dependencies for Maven -->
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-streaming_2.12</artifactId>
  <version>3.0.1</version>
</dependency>
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-streaming-kafka-0-10_2.12</artifactId>
  <version>3.0.1</version>
</dependency>
```

Write the Spark Streaming Application:

In a new Scala file named KafkaSparkConsumer.scala, add the following code

```
import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.kafka010._
import org.apache.kafka.common.serialization.StringDeserializer
```

```
object KafkaSparkConsumer {
  def main(args: Array[String]): Unit = {
    // Spark configuration
    val sparkConf = new SparkConf().setAppName("KafkaSparkConsumer").setMaster("local[*]")
    val ssc = new StreamingContext(sparkConf, Seconds(5))
```

```
// Kafka parameters
val kafkaParams = Map[String, Object](
  "bootstrap.servers" -> "localhost:9092",
  "key.deserializer" -> classOf[StringDeserializer],
  "value.deserializer" -> classOf[StringDeserializer],
  "group.id" -> "spark-consumer-group",
  "auto.offset.reset" -> "latest",
  "enable.auto.commit" -> (false: java.lang.Boolean)
)

// Define the topic
val topics = Array("flume-kafka-topic")

// Create a direct stream
val stream = KafkaUtils.createDirectStream[String, String](
  ssc,
  LocationStrategies.PreferConsistent,
  ConsumerStrategies.Subscribe[String, String](topics, kafkaParams)
)

// Process each message
stream.map(record => record.value).foreachRDD { rdd =>
  rdd.foreach { message =>
    println(s"Received message: $message")
  }
}

// Start streaming
ssc.start()
ssc.awaitTermination()
}
```

Run the Spark Application:

Compile and run your KafkaSparkConsumer application.

Verify the Data Flow and Output

In a new terminal, add data to the file that Flume is monitoring:

```
echo "Log Entry 1" >> /path/to/your/logfile.log
```

```
echo "Log Entry 2" >> /path/to/your/logfile.log
```

Expected Output in Spark Console:

After a few seconds, you should see output in the Spark Streaming application console that looks like this:

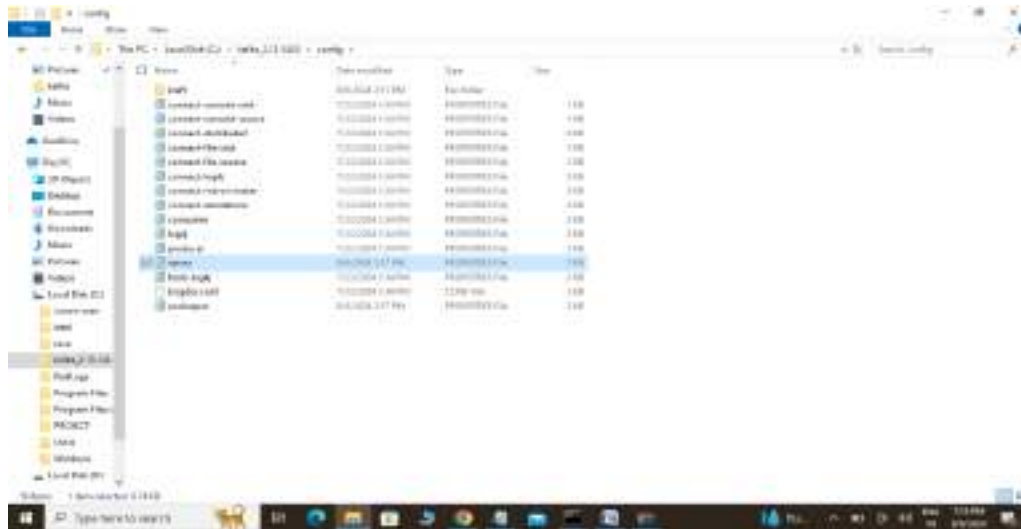
```
Received message: Log Entry 1
```

```
Received message: Log Entry 2
```

EXPERIMENT NO 12

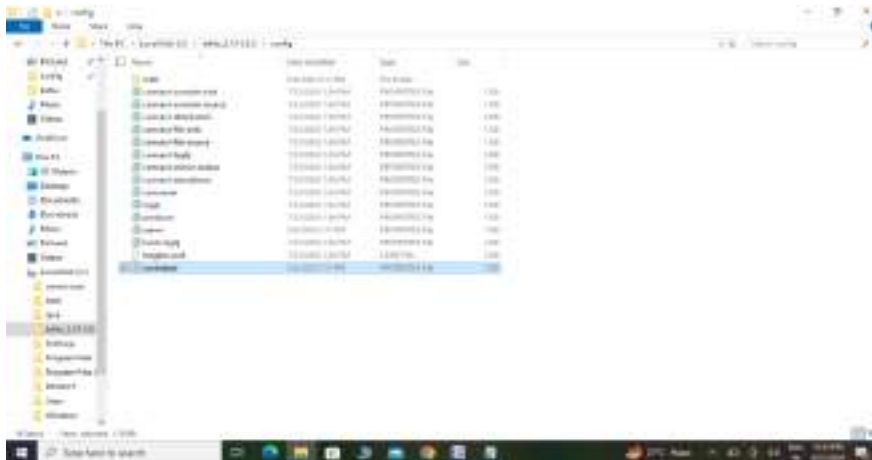
NAME OF THE EXPERIMENT: Kafka Application as Consumer and Producer.

STEP 1: enter into c drive □ kafka_2.13-3.8.0 □ C:\kafka_2.13-3.8.0\config □ open server



log.dirs=C:\kafka_2.13-3.8.0\kafka-logs (replace temp with C:\kafka_2.13-3.8.0) and SAVE the file

STEP 2: Enter into c drive □ kafka_2.13-3.8.0 □ C:\kafka_2.13-3.8.0\config □ open zookeeper

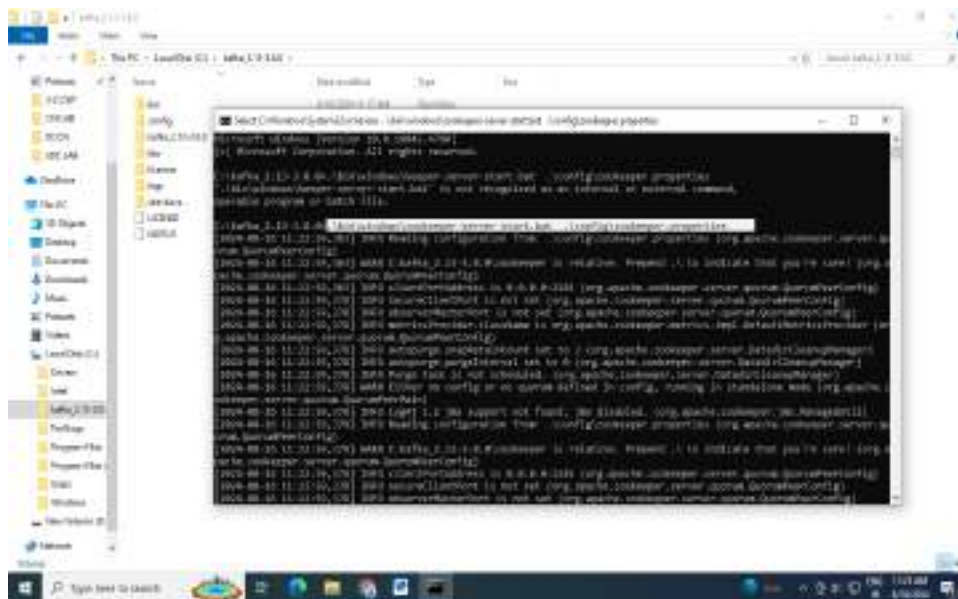


dataDir=C:\kafka_2.13-3.8.0\zookeeper (replace temp with C:\kafka_2.13-3.8.0) and SAVE the file

STEP 3: Open cmd and enter into the path C:\kafka_2.13-3.8.0

PATH: C:\kafka_2.13-3.8.0

TYPE: .\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties

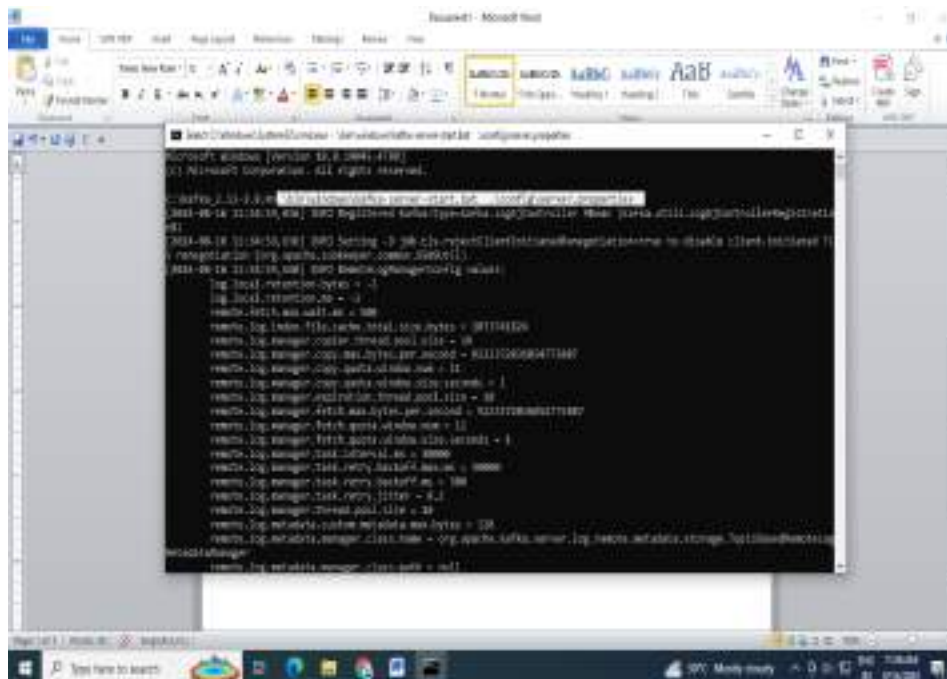


NOTE: DON'T CLOSE CMD

STEP 4: Open A NEW CMD and enter into the path C:\kafka_2.13-3.8.0

PATH: C:\kafka_2.13-3.8.0

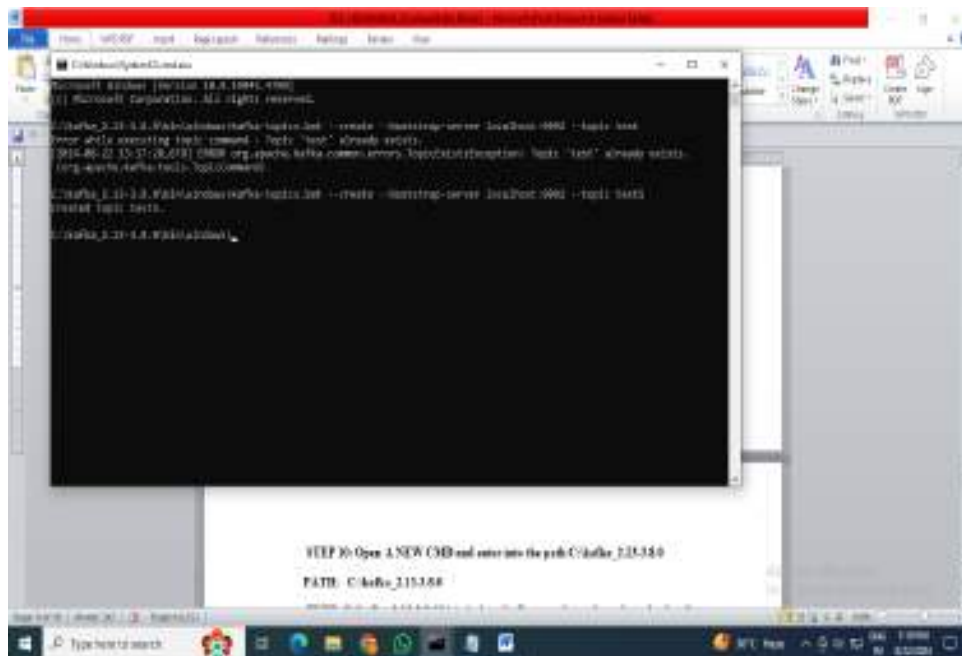
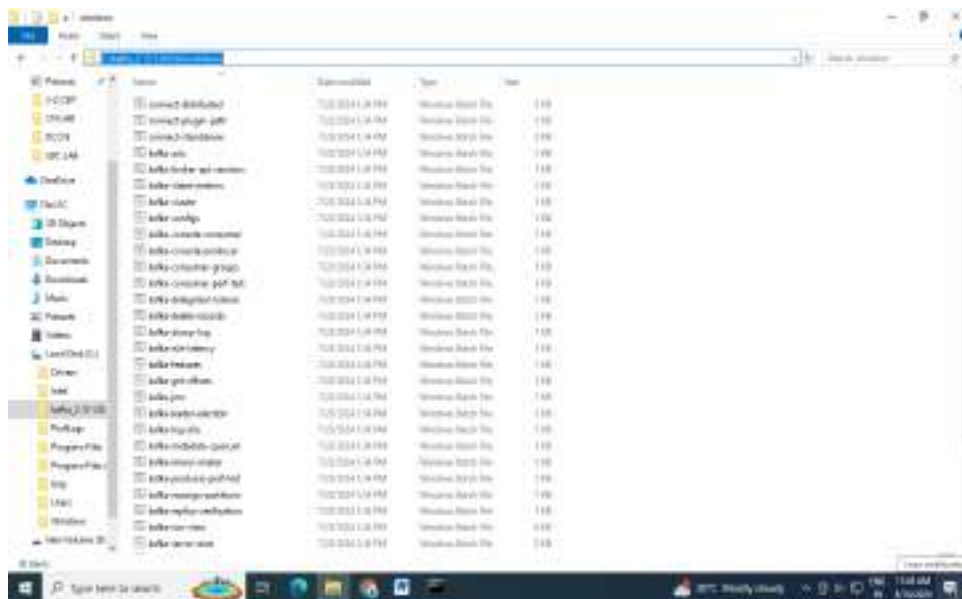
TYPE: .bin\windows\kafka-server-start.bat .\config\server.properties



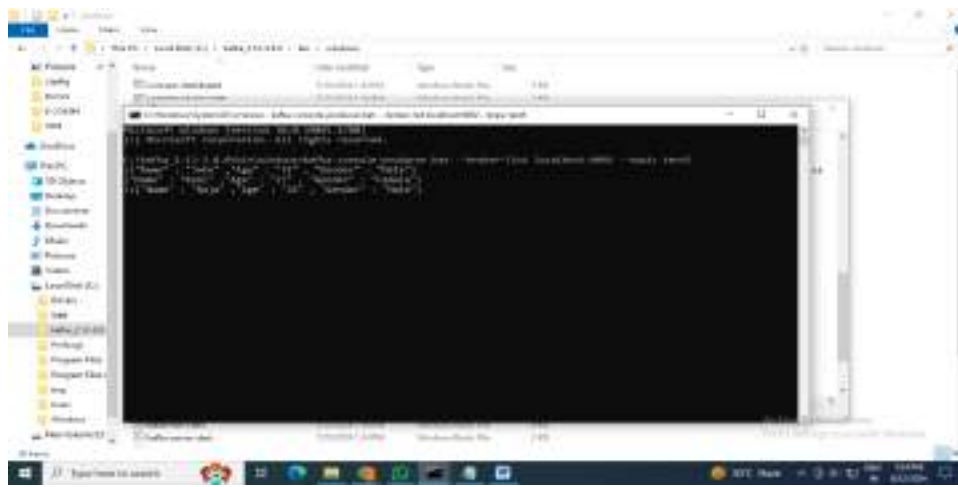
STEP 5: Open A NEW CMD and enter into the path C:\kafka_2.13-3.8.0

PATH: C:\kafka_2.13-3.8.0

This step is to create a topic by using the below command



```
TYPE: C:\kafka_2.13-3.8.0\bin\windows\kafka-console-producer.bat --broker-list localhost:9092 --topic test
TYPE: (PRODUCER TAB)
{"Name": "John", "Age": "31", "Gender": "Male"}
{"Name": "Raju", "Age": "41", "Gender": "Male"}
{"Name": "Rani", "Age": "19", "Gender": "Female"}
```

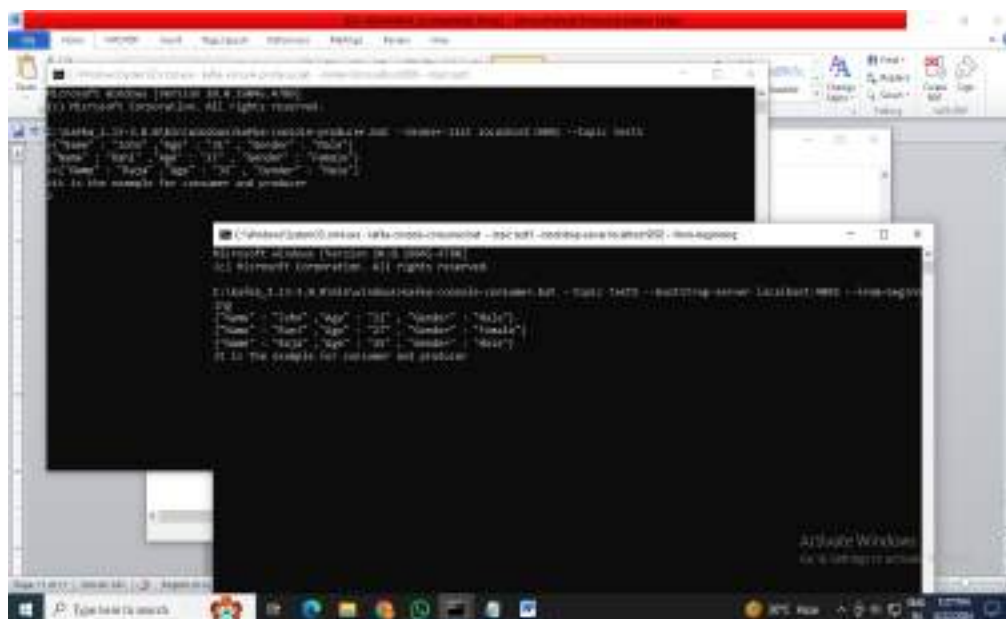


STEP 7: Open A NEW CMD and enter into the path C:\kafka_2.13-3.8.0

PATH: C:\kafka_2.13-3.8.0

(CONSUMER TAB)

TYPE: C:\kafka_2.13-3.8.0\bin\windows\kafka-console-consumer.bat --topic test --bootstrap-server localhost:9092 --from-beginning



EXPERIMENT NO. 13

NAME OF THE EXPERIMENT: Word Count per Record

word count per record in a Kafka stream or a Kafka topic, you would typically perform the following steps:

1. Consume the Kafka records

You first need to consume the records (messages) from your Kafka topic. This is usually done by a consumer application written in a language like Java, Python, or Scala.

2. Parse the message

Kafka messages are typically serialized, so you'll need to deserialize the record into a usable format, like a string or JSON, depending on how the messages are structured.

3. Count the words

Once you have the message in a string format, you can split the string into words and then count them. This would depend on the language you're using, but the general idea is to split the string by spaces (or other delimiters) and count the resulting elements.

4. Process each record

If you're processing a stream, you'll want to calculate the word count for each record as it comes in, rather than processing a bulk batch.

Install Required Libraries

First, make sure you have the confluent-kafka-python library installed. You can install it using pip:

```
pip install confluent-kafka
```

Python Code Example:

```
from confluent_kafka import Consumer, KafkaException, KafkaError
```

```
# Configure the Kafka Consumer
```

```
conf = {  
    'bootstrap.servers': 'localhost:9092', # Kafka broker  
    'group.id': 'word-count-group',      # Consumer group ID  
    'auto.offset.reset': 'earliest',     # Start reading from the beginning  
}
```

```
# Create the consumer instance
```

```
consumer = Consumer(conf)
```

```
# Kafka topic to consume messages from
```

```
topic = 'your_topic_name' # Replace with your actual topic name  
consumer.subscribe([topic])
```

```
# Function to calculate the word count of a message
```

```
def word_count(text):  
    # Split the message by whitespace and count the number of words  
    return len(text.split())
```

```
# Consume messages and calculate word count
```

```
try:
    while True:
        # Poll for messages (timeout in seconds)
        msg = consumer.poll(timeout=1.0)

        if msg is None:
            continue # No new message, continue polling

        if msg.error():
            if msg.error().code() == KafkaError._PARTITION_EOF:
                print(f"End of partition reached {msg.partition}")
            else:
                raise KafkaException(msg.error())
        else:
            # Deserialize the message value (assuming it's a UTF-8 string)
            message_value = msg.value().decode('utf-8')

            # Calculate word count
            count = word_count(message_value)
            print(f"Message: {message_value}\nWord Count: {count}\n")

except KeyboardInterrupt:
    print("Consumer interrupted.")
finally:
    # Close the consumer to commit offsets and clean up
    consumer.close()
```

Example Output:

```
Message: This is a test message.
Word Count: 5
```

```
Message: Another example with more words.
Word Count: 6
```