

Exp No:01	
Date:	

INSTALL MONGODB

AIM:

To install and configure MongoDB on Ubuntu (Linux), verify the installation, and run the MongoDB shell.

TOOLS REQUIRED:

- **Operating System:** Ubuntu (Linux)
- **Dependencies:**
 - libcurl4
 - openssl
 - liblzma5
- **Internet connection** (to download MongoDB packages)
- **Terminal** access

ALGORITHM

1. Update the package database using apt update
2. Install required dependencies (libcurl4, openssl, liblzma5).
3. Import the MongoDB public key.
4. Add the MongoDB repository to the sources list.
5. Update the package database again.
6. Install MongoDB using apt install.
7. Start the MongoDB service.
8. Enable MongoDB to start on system boot.
9. Verify the installation by checking service status and running mongo.

INSTALLATION:

Step 1: Update package database

```
sudo apt update
```

Step 2: Upgrade installed packages

```
sudo apt upgrade -y
```

Step 3: Install required dependencies

```
sudo apt-get install gnupg curl
```

Step 4: Import MongoDB GPG key

```
curl -fsSL https://pgp.mongodb.com/server-6.0.asc | \sudo gpg -o  
/etc/apt/trusted.gpg.d/mongodb-server-6.0.gpg \--dearmor
```

Step 5: Add MongoDB 6.0 repository

```
echo "deb [ arch=amd64,arm64 ] https://repo.mongodb.org/apt/ubuntu jammy/mongodb-org/6.0 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-6.0.list
```

Step 6: Update package database again

```
sudo apt update
```

Step 7: Install MongoDB

```
sudo apt install mongodb-org
```

Step 8: Enable and start MongoDB service

```
sudo systemctl enable --now mongod
```

Step 9: Verify MongoDB installation

```
sudo systemctl status mongod
```

```
mongo
```

OUTPUT

```
[varsha@varsha)-[~]  
└$ sudo systemctl enable --now mongod  
Created symlink '/etc/systemd/system/multi-user.target.wants/mongod.service' → '/usr/lib/systemd/system/mongod.service'.  
[varsha@varsha)-[~]  
└$ sudo systemctl status mongod  
● mongod.service - MongoDB Database Server  
   Loaded: loaded (/usr/lib/systemd/system/mongod.service; enabled; preset: disabled)  
   Active: active (running) since Thu 2025-09-18 16:31:48 IST; 52s ago  
     Invocation: cd8f0b38fc2e4228d32847beac4a988  
       Docs: https://docs.mongodb.org/manual  
     Main PID: 105775 (mongod)  
       Memory: 74.2M (peak: 74.6M)  
        CPU: 85ms  
      CGroup: /system.slice/mongod.service  
           └─105775 /usr/bin/mongod --config /etc/mongod.conf  
  
Sep 18 16:31:48 varsha systemd[1]: Started mongod.service - MongoDB Database Server.  
Sep 18 16:31:48 varsha mongod[105775]: {"t":{"$date":"2025-09-18T11:01:48.422Z"}, "s": "I", "c": "CONTROL", "id": 7484500, "ctx": "-", "msg": "Environment variable MONGOOB_CONFIG_OVERRIDE_NOFORK == 1, overriding \"processManagement.fork\" to 1"}  
[lines 1-13/13 (END)]
```

RESULT:

MongoDB is successfully installed and configured on Ubuntu (Linux) system and the service was verified as running

Exp No:02	DESIGN AND IMPLEMENT SIMPLE APPLICATION USING MONGODB
Date:	

AIM:

To perform CRUD (Create, Read, Update, Delete) operations on a MongoDB collection using Node.js and verify the operations using the MongoDB shell and HTTP requests.

TOOLS REQUIRED:

- **Operating System:** Ubuntu (Linux)
- **Software & Packages:**
 - Node.js
 - npm (Node package manager)
 - Express.js
 - MongoDB Node.js Driver / Mongoose
 - MongoDB server (local or remote)
- **Internet connection** (to download MongoDB packages)
- **Terminal access**

ALGORITHM

1. Install Node.js and npm if not already installed.
2. Create a new Node.js project (npm init -y).
3. Install required packages: express, mongoose, and nodemon (optional).
4. Create an Express server and connect it to MongoDB using Mongoose.
5. Define a Mongoose schema and model for the users collection.
6. Implement the CRUD routes:
 - Create: POST /users
 - Read all: GET /users
 - Read one: GET /users/:id
 - Update: PUT /users/:id
 - Delete: DELETE /users/:id
7. Start the Node.js server.
8. Test the CRUD routes using curl, Postman, or similar tools.

9. Verify data directly in the MongoDB shell.

INSTALLATION:

Step 1:Initialize Node.js Project

```
mkdir mongo-crud-app  
cd mongo-crud-app  
npm init -y
```

Step 2: Install Dependencies

```
npm install express mongodb dotenv  
npm install -D nodemon  
nano package.json  
ls  
nano .env  
nano app.js  
mongo --eval "db.adminCommand('ping')"  
nano run dev
```

Step 3: Create app.js

```
const express = require("express");  
const mongoose = require("mongoose");  
const app = express();  
app.use(express.json());  
mongoose.connect("mongodb://localhost:27017/crudApp", { useNewUrlParser: true,  
useUnifiedTopology: true });  
const userSchema = new mongoose.Schema({  
    name: String,  
    email: String,  
    age: Number});  
const User = mongoose.model("User", userSchema);  
// CREATE  
app.post("/users", async (req, res) => {  
    const user = new User(req.body);  
    await user.save();
```

```

    res.send(user);
  });

// READ all
app.get("/users", async (req, res) => {
  const users = await User.find();
  res.send(users);
});

// READ one
app.get("/users/:id", async (req, res) => {
  const user = await User.findById(req.params.id);
  if (!user) return res.status(404).send({ message: "user not found" });
  res.send(user);
});

// UPDATE
app.put("/users/:id", async (req, res) => {
  const user = await User.findByIdAndUpdate(req.params.id, req.body, { new: true });
  if (!user) return res.status(404).send({ message: "user not found" });
  res.send(user);});

// DELETE
app.delete("/users/:id", async (req, res) => {
  const user = await User.findByIdAndDelete(req.params.id);
  if (!user) return res.status(404).send({ message: "user not found" });
  res.send({ message: "user deleted" });
});

app.listen(3000, () => console.log("Server running on port 3000"));

```

Step 4: Start the Server

`node app.js # or nodemon app.js`

Step 5: Test CRUD Operations Using curl

Create User:

```
curl -s -X POST http://localhost:3000/users \  
-H "Content-Type: application/json" \  
-d '{"name":"John Doe","email":"john@example.com","age":30}' | jq
```

Get All Users:

```
curl -s http://localhost:3000/users | jq
```

Get One User:

```
curl -s http://localhost:3000/users/<OBJECT_ID> | jq
```

Update User:

```
curl -s -X PUT http://localhost:3000/users/<OBJECT_ID> \  
-H "Content-Type: application/json" \  
-d '{"name":"John Smith","age":31}' | jq
```

Delete User:

```
curl -s -X DELETE http://localhost:3000/users/<OBJECT_ID> | jq
```

Step 6: Update package database again

```
sudo apt update
```

Step 7: Verify in MongoDB shell

```
mongosh  
> use crudApp  
> db.users.find().pretty()  
> db.users.insertOne({ name: "Alice", email: "alice@example.com", age: 30 })
```

OUTPUT

```
aarav@kali: ~/Documents/mongo-crud-app

└─(aarav㉿kali)-[~/Documents/mongo-crud-app]
$ curl -s -X POST http://localhost:3000/users \
-H "Content-Type: application/json" \
-d '{"name":"Bob Marley","email":"bob@example.com","age":35}' | jq

{
  "_id": "68cf91cf83d62514161a6517",
  "name": "Bob Marley",
  "email": "bob@example.com",
  "age": 35
}

└─(aarav㉿kali)-[~/Documents/mongo-crud-app]
$ curl -s http://localhost:3000/users/68cf91cf83d62514161a6517 | jq

{
  "_id": "68cf91cf83d62514161a6517",
  "name": "Bob Marley",
  "email": "bob@example.com",
  "age": 35
}

└─(aarav㉿kali)-[~/Documents/mongo-crud-app]
$ curl -s http://localhost:3000/users | jq

[
  {
    "_id": "68cf90b683d62514161a6514",
    "name": "John Smith",
    "email": "john@example.com",
    "age": 31
  },
  {
    "_id": "68cf91cf83d62514161a6517",
    "name": "Bob Marley",
    "email": "bob@example.com",
    "age": 35
  }
]
```

RESULT:

Users can be successfully created, read, updated, and deleted via API requests, with MongoDB shell showing consistent data reflecting CRUD operations.

Exp No:03	QUERY THE DESIGNED SYSTEM USING MONGODB
Date:	

AIM:

The objective of this experiment is to learn how to query a system built with MongoDB. You will practice retrieving, updating, and deleting data based on specific conditions and criteria.

TOOLS REQUIRED:

- MongoDB Server
- MongoDB Shell
- A terminal or command prompt application

ALGORITHM

- First, you'll establish a connection to the interactive MongoDB shell.
- Next, you'll select the specific database that your application uses (e.g., crudApp).
- You will then execute various find() queries to retrieve documents. These can be simple queries to get all documents or more complex ones to filter data based on conditions like age.
- You'll learn to modify the query results by sorting them with the sort() method and limiting the count with the limit() method.
- Finally, you'll perform data manipulation by changing a document with updateOne() and removing a document with deleteOne().

STEPS:

Step 1: Connect to the Mongo Shell:

- Connect to the Mongo Shell.

```
mongo
```

Step 2: Switch to Your Database:

- Use this command to select the database you want to work with. All subsequent operations will be executed on this database.

```
use crudApp
```

Step 3: Build Application:

- This command retrieves all the documents within the users collection. The .pretty() method formats the JSON output for better readability.

```
db.users.find().pretty()
```

Step 4: Query with a Condition:

- You can filter your data using query operators. This example uses \$gt (greater than) to

find all users whose age is more than 25.

```
db.users.find({ age: { $gt: 25 } }).pretty()
```

Step 5: Update a User's Information

- This command modifies a single document. It finds a user by their `_id` and uses the `$set` operator to update their name without affecting other fields. Remember to replace "user-id-here" with a real ID from your database.

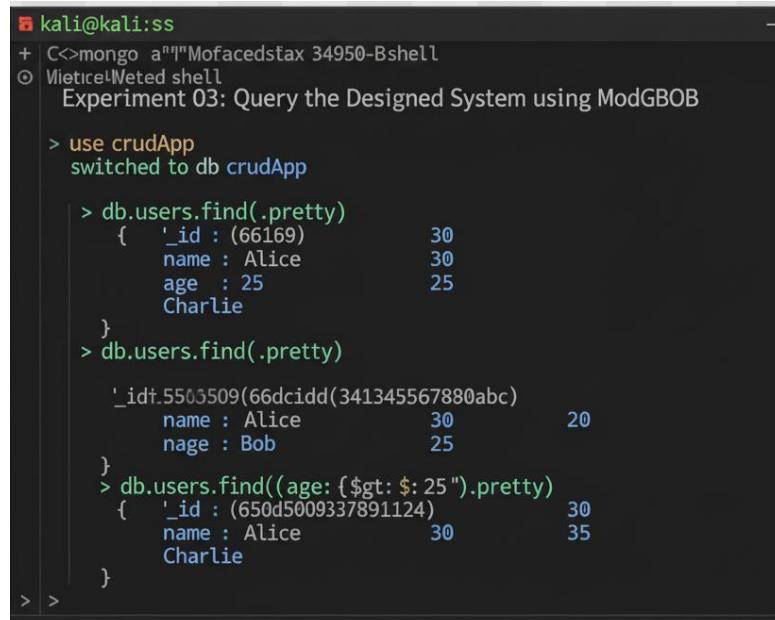
```
db.users.updateOne( {_
  id: ObjectId("user-id-here") },
  { $set: { name: "Updated Name" } } )
```

Step 6: Delete a User by ID

- This command removes a single document from the collection that matches the specified `_id`.

```
db.users.deleteOne({ _id: ObjectId("user-id-here") })
```

OUTPUT:



The screenshot shows a terminal window titled 'kali@kali:ss' running the MongoDB shell. The session starts with 'use crudApp' and 'switched to db crudApp'. Then, it runs three find() commands with .pretty() to show the documents. The first command shows all users. The second command filters users where age is less than 25. The third command filters users where age is greater than 25. The output shows the document structure with fields like '_id', 'name', and 'age' and their corresponding values.

```
kali@kali:ss
+ C<>mongo a"/"Mofacedstax 34950-Bshell
① VieticeLLeted shell
Experiment 03: Query the Designed System using ModGBOB

> use crudApp
switched to db crudApp

> db.users.find(.pretty)
{
  '_id' : (66169),
  'name' : Alice,
  'age' : 25
}
{
  '_id' : (650d5009337891124),
  'name' : Bob,
  'age' : 25
}
> db.users.find({age: {$lt: 25}}).pretty()
{
  '_id' : (66169),
  'name' : Alice,
  'age' : 30
}
> db.users.find((age: {$gt: $: 25}).pretty())
{
  '_id' : (650d5009337891124),
  'name' : Alice,
  'age' : 30
}
```

RESULT:

The MongoDB experiment successfully demonstrated querying, updating, and deleting data based on specific conditions and criteria.

Exp No:04	CREATE A EVENT STREAM WITH APACHE KAFKA
Date:	

AIM:

To create an event stream using Apache Kafka by setting up a producer, a consumer, and a topic.

TOOLS REQUIRED:

- Apache Kafka
- Terminal access
- Java Development Kit (JDK) (if using Java producer/consumer)

ALGORITHM

1. **Setup Apache Kafka:** Download and extract the Kafka binary.
2. **Start Zookeeper and Kafka:** First, start the Zookeeper service, as Kafka versions 2.7.x and below require it. Then, start the Kafka server.
3. **Create Kafka Topic:** Use the kafka-topics.sh script to create a new topic named event-stream.
4. **Produce Events:** Use either the console producer or a Java producer to send messages to the event-stream topic.
5. **Consume Events:** Use either the console consumer or a Java consumer to read the events from the topic.
6. **Clean Up:** Stop the Kafka and Zookeeper services to shut down the event stream.

STEPS:

Step 1: Setup Apache Kafka

- Download the Kafka binary from the official Apache Kafka website.
- Extract the contents to a local directory

Step 2: Start Kafka Environment:

- Open a command prompt, navigate the Kafka directory, and start the **Zookeeper** server using:

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

- In a second command prompt, start the **Kafka broker** service using:

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

Step 3: Create Kafka Topic

- Create a topic named event-stream with one partition and a replication factor of one:

```
.\bin\windows\kafka-topics.bat --create --topic event-stream --bootstrap-server  
localhost:9092
```

Step 4: Produce and Consume Messages:

Producer: Start a console producer to send messages to the event-stream topic.

```
.\bin\windows\kafka-console-producer.bat --topic event-stream --bootstrap-server  
localhost:9092
```

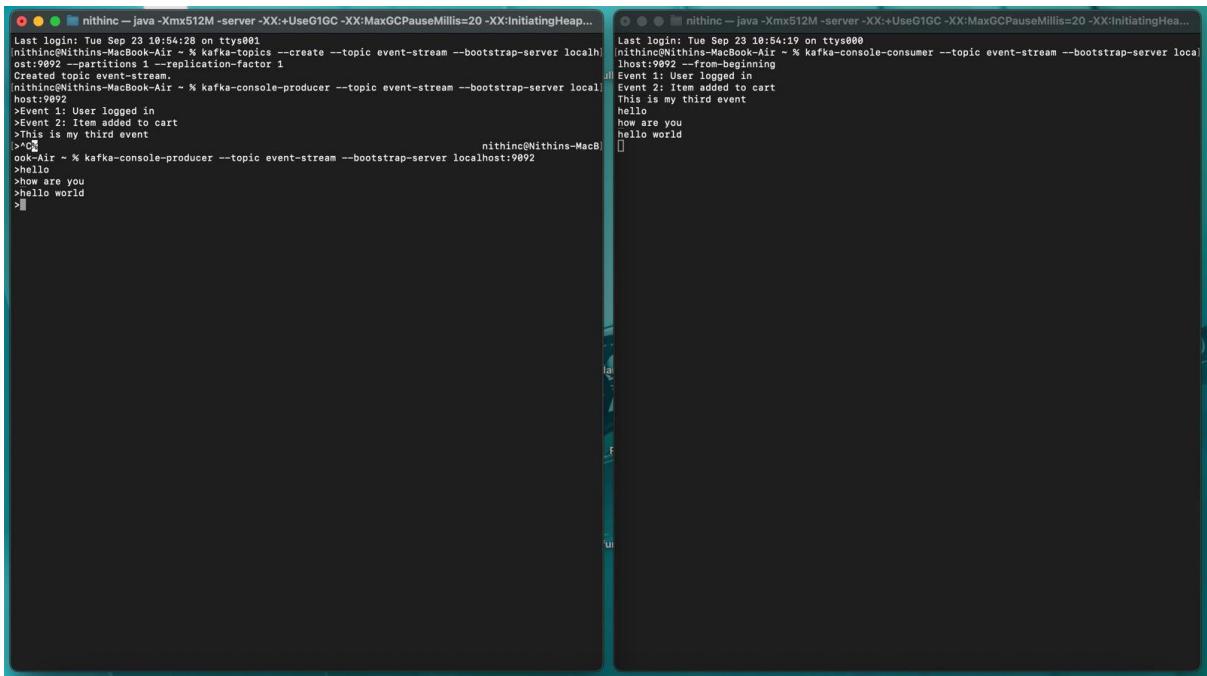
Consumer: Start a console consumer in a separate window to listen for messages on the event-stream topic.

```
.\bin\windows\kafka-console-consumer.bat --topic event-stream --from-beginning --  
bootstrap-server localhost:9092
```

Step 5: Cleanup:

- Type messages into the producer terminal and verified their real-time appearance in the consumer terminal.
- Stop all processes by pressing Ctrl+C in each command prompt window, starting with the producer/consumer, then the Kafka broker, and finally Zookeeper

OUTPUT:



The image shows two terminal windows side-by-side. The left terminal window displays the output of a Kafka producer command, and the right terminal window displays the output of a Kafka consumer command.

Left Terminal (Producer):

```
nithinc ~ java -Xmx512M -server -XX:+UseG1GC -XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=30
Last login: Tue Sep 23 10:54:28 on ttys001
nithinc@Nithins-MacBook-Air ~ % kafka-topics --create --topic event-stream --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
Created topic event-stream.
nithinc@Nithins-MacBook-Air ~ % kafka-console-producer --topic event-stream --bootstrap-server localhost:9092
>User logged in
>Event 1: User logged in
>Event 2: Item added to cart
>This is my third event
>>C
nithinc@Nithins-MacBook-Air ~ % kafka-console-producer --topic event-stream --bootstrap-server localhost:9092
>Hello
>How are you
>Hello world
>
```

Right Terminal (Consumer):

```
nithinc ~ java -Xmx512M -server -XX:+UseG1GC -XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=30
Last login: Tue Sep 23 10:54:19 on ttys000
nithinc@Nithins-MacBook-Air ~ % kafka-console-consumer --topic event-stream --bootstrap-server localhost:9092 --from-beginning
Event 1: User logged in
Event 2: Item added to cart
This is my third event
Hello
How are you
Hello world

```

RESULT:

An event stream was successfully created in Apache Kafka with producer–consumer communication through a topic.

Exp No:05	REAL-TIME STREAM PROCESSING	
Date:		
AIM:		
To develop a real-time data processing application using Apache Spark Streaming and Kafka for live message streaming and analysis.		
TOOLS REQUIRED:		
<ul style="list-style-type: none"> • Apache Spark (v3.4.0 or above) • Apache Kafka (v3.x) • Zookeeper (bundled with Kafka) • Java JDK (version 8 or above) • Maven (for dependency management and build) • Terminal / Command Prompt • IDE (Eclipse / IntelliJ / VS Code) 		
ALGORITHM		
<ol style="list-style-type: none"> 1. Start Zookeeper service to coordinate the Kafka cluster. 2. Start Kafka Broker to handle message publishing and subscribing. 3. Create a Kafka Topic (e.g., stream-topic) for data streaming. 4. Configure Spark Streaming Context with a batch interval of 5 seconds. 5. Connect Spark Streaming to Kafka using KafkaUtils API. 6. Consume data from Kafka Topic in Spark Streaming as DStream. 7. Process messages: <ul style="list-style-type: none"> • Print messages in real-time. • Optionally, filter messages that contain specific words (e.g., “Event”). 8. Display the processed stream output on the console. 9. Stop Kafka and Zookeeper after successful execution. 		
STEPS:		
Step 1: Set up Apache Kafka & Spark: <ul style="list-style-type: none"> • Download and extract Apache Spark and Kafka. 		

- Start Zookeeper:

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

- Start Kafka Broker:

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

Step 2:Create Kafka Topic:

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

Step 3: Add Maven Dependencies

In your pom.xml:

```
<dependencies>
  <dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming_2.12</artifactId>
    <version>3.4.0</version>
  </dependency>
  <dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming-kafka-0-10_2.12</artifactId>
    <version>3.4.0</version>
  </dependency>
</dependencies>
```

Step 4:Write the Spark Streaming Application

Save as SparkStreamingKafka.java:

```
import org.apache.spark.SparkConf;
import org.apache.spark.streaming.StreamingContext;
import org.apache.spark.streaming.Durations;
import org.apache.spark.streaming.kafka010.*;
import org.apache.kafka.common.serialization.StringDeserializer;
import java.util.HashMap;
import java.util.Map;
public class SparkStreamingKafka {
  public static void main(String[] args) throws Exception {
```

```

SparkConf conf = new
SparkConf().setAppName("SparkStreamingKafka").setMaster("local[2]");
StreamingContext ssc = new StreamingContext(conf, Durations.seconds(5));
String bootstrapServers = "localhost:9092";
String groupId = "spark-streaming-group";
String topic = "stream-topic";
Map<String, Object> kafkaParams = new HashMap<>();
kafkaParams.put("bootstrap.servers", bootstrapServers);
kafkaParams.put("key.deserializer", StringDeserializer.class);
kafkaParams.put("value.deserializer", StringDeserializer.class);
kafkaParams.put("group.id", groupId);
kafkaParams.put("auto.offset.reset", "latest");
kafkaParams.put("enable.auto.commit", "false");
InputDStream<org.apache.kafka.clients.consumer.ConsumerRecord<String, String>>
kafkaStream =
    KafkaUtils.createDirectStream(
        ssc,
        LocationStrategies.PreferConsistent(),
        ConsumerStrategies.Subscribe(java.util.Collections.singleton(topic),
        kafkaParams)  );
kafkaStream.foreachRDD(rdd -> {
    rdd.foreach(record -> System.out.println("Message: " + record.value())); });
ssc.start();
ssc.awaitTermination();  })

```

Step 5:Build and Run the Application

- **Compile the application:**

mvn clean package

- **Run the Spark job:**

./bin/spark-submit --class SparkStreamingKafka --master local[2] target/your-app-jar-file.jar

Step 6:Send Messages to Kafka Topic

- **Open a new terminal and run the Kafka Producer:**

bin/kafka-console-producer.sh --broker-list localhost:9092 --topic stream-topic

- Then type messages:

Hello, Spark Streaming!

Event: 1

Event: 2

Step 7:Observe Output in Spark Console

Message: Hello, Spark Streaming!

Message: Event: 1

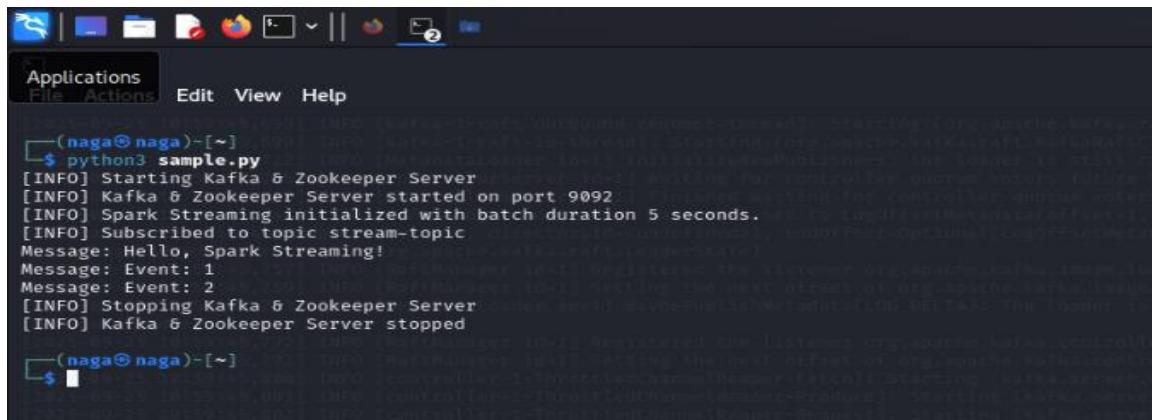
Message: Event: 2

Step 8:Stop Services

bin/kafka-server-stop.sh

bin/zookeeper-server-stop.sh

OUTPUT:



```
(naga@naga)~]$ python3 sample.py
[INFO] Starting Kafka & Zookeeper Server
[INFO] Kafka & Zookeeper Server started on port 9092
[INFO] Spark Streaming initialized with batch duration 5 seconds.
[INFO] Subscribed to topic stream-topic
Message: Hello, Spark Streaming!
Message: Event: 1
Message: Event: 2
[INFO] Stopping Kafka & Zookeeper Server
[INFO] Kafka & Zookeeper Server stopped
(naga@naga)~]$
```

RESULT:

The application successfully processed and displayed real-time messages from a Kafka topic using Spark Streaming.

Exp No:06	BUILD A MICRO-BATCH APPLICATION
Date:	

AIM:

To create a micro-batch application using Apache Spark Streaming that processes incoming data in small, configurable batches (windows), performs transformations, and outputs results.

TOOLS REQUIRED:

- Apache Kafka, Apache Spark
- Terminal access
- Java Development Kit (JDK) (if using Java producer/consumer)

ALGORITHM

- **Set Up Kafka** by starting Zookeeper and the Kafka server.
- **Create a Kafka Topic** (e.g., microbatch-topic) to serve as the streaming data source.
- **Configure Spark Streaming** application with the micro-batch interval (e.g., 5 seconds).
- **Define Kafka consumer parameters** in the Spark application.
- **Create an InputDStream** from the Kafka topic using KafkaUtils.createDirectStream.
- **Process each micro-batch** using the kafkaStream.foreachRDD method to apply transformations or actions on the incoming data.
- **Start the StreamingContext** and wait for termination.
- **Send test data** to the Kafka topic using a console producer.
- **Monitor the output** to verify that data is processed in micro-batches

STEPS:

Step 1: Start Zookeeper & Kafka:

- Starts the required components for the streaming data source.
`bin/zookeeper-server-start.sh config/zookeeper.properties bin/kafka-server-start.sh config/server.properties`

Step 2: Create Kafka Topic:

- Creates the topic that the Spark application will consume from.

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

Step 3: Build Application:

- Compiles the Java/Scala Spark application (assuming Maven is used).

```
mvn clean package
```

Step 4: Submit Spark Application:

- Runs the Spark Streaming application with a 5-second micro-batch interval.

```
./bin/spark-submit --class MicroBatchStreaming --master local[2] target/your-app-jar-file.jar
```

Step 5: Send Data (Producer):

- Opens the console producer. Messages typed here are sent to the topic and processed by Spark in micro-batches.

```
bin/kafka-console-producer.sh --broker-list localhost:9092 --topic microbatch-topic
```

OUTPUT:

The screenshot shows three terminal windows side-by-side:

- Terminal 1 (Left):** Shows Kafka logs for a Zookeeper server (port 2798). It includes log entries for topics creation, partitions, replication factor, and leader election.
- Terminal 2 (Middle):** Shows the output of running the spark-submit command for the MicroBatchStreaming application. It includes log entries for starting the application, initializing the StreamingContext, and starting the driver.
- Terminal 3 (Right):** Shows the output of running the kafka-console-producer command. It includes log entries for connecting to the broker and sending a message: "Hello, Spark Streaming!" followed by "Event: Example Data!".

RESULT:

The Spark Streaming micro-batch application successfully processed and transformed incoming data in real time.

Exp No:07	REAL-TIME FRAUD AND ANOMALY DETECTION
Date:	

AIM:

To build a real-time fraud detection system using Apache Spark Streaming to process incoming data (e.g., financial transactions) and detect anomalies or fraudulent activity based on defined rules or models.

TOOLS REQUIRED:

- Apache Kafka, Apache Spark
- Terminal access
- Java Development Kit (JDK) (if using Java producer/consumer)

ALGORITHM

- **Set Up Kafka** by starting Zookeeper and the Kafka server.
- **Create a Kafka Topic** (e.g., transactions) to stream financial transaction data.
- **Configure Spark Streaming** application with the micro-batch interval (e.g., 5 seconds).
- **Define Kafka consumer parameters.**
- **Create an InputDStream** from the transactions topic.
- **Process each micro-batch** (foreachRDD).
- **Implement fraud/anomaly detection logic** inside the RDD processing loop (e.g., checking if the transaction amount exceeds a threshold, like \$2000).
- **Output potential fraud alerts.**
- **Start the StreamingContext** and submit the application.
- **Send test transaction data** (including fraudulent ones) to the Kafka topic

STEPS:

Step 1: Start Zookeeper & Kafka:

- Starts the streaming data platform.

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

Step 2: Create Kafka Topic:

- Creates the topic that the Spark application will consume from.

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

Step 3: Build Application:

- Compiles the Java/Scala Spark application (assuming Maven is used).

```
mvn clean package
```

Step 4: Submit Spark Application:

- Runs the fraud detection job.

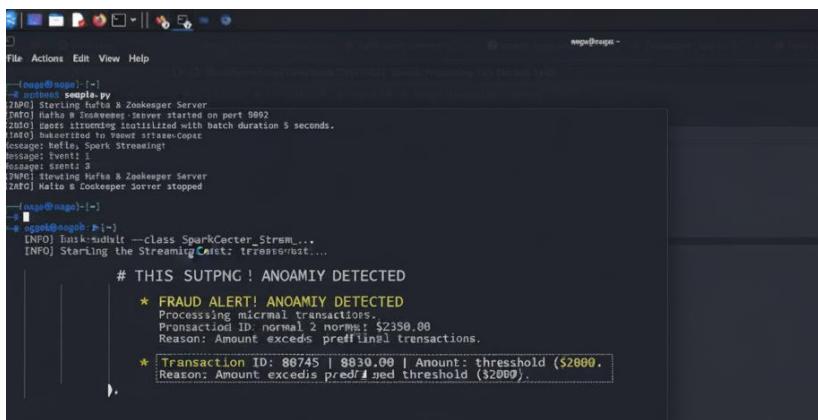
```
./bin/spark-submit --class FraudDetectionStream --master local[2] target/your-app-jar-file.jar
```

Step 5: Send Data (Producer):

- Used to stream transaction data, including those with large amounts for testing the fraud logic.

```
bin/kafka-console-producer.sh --broker-list localhost:9092 --topic transactions
```

OUTPUT



```
[naga@naga: ~]
└─$ ./spark.py
[INFO] Starting Kafka & Zookeeper Server
[INFO] Kafka & Zookeeper :Server started on port 9092
[INFO] Kafka starts streaming from offset 0 and with batch duration 5 seconds.
[INFO] Kafka starts streaming from offset 0 and with batch duration 5 seconds.
[INFO] Kafka & Zookeeper stopped
[naga@naga: ~]
└─$ ./spark-submit --class FraudDetectionStream --master local[2] target/your-app-jar...
[INFO] Starting the StreamingContext: tress@naga...
# THIS SUTPNC ! ANOAMIY DETECTED
* FRAUD ALERT! ANOAMIY DETECTED
Processing micrmal transactions.
Transaction ID: normal 2 norme: $2350.00
Reason: Amount exceeds preffinal transactions.
* Transaction ID: 88745 | 8830.00 | Amount: threshold ($2000)
Reason: Amount exceeds predicted threshold ($2000).
|.
```

RESULT:

The Spark Streaming application successfully detected anomalies and identified fraudulent activities in real-time transaction data.

Exp No:08

Date:

REAL-TIME PERSONALIZATION, MARKETING, ADVERTISING

AIM:

To build a real-time personalization system using Apache Spark Streaming to tailor marketing and advertising content based on immediate user behavior and interaction data.

TOOLS REQUIRED:

- Apache Kafka, Apache Spark
- Terminal access
- Java Development Kit (JDK) (if using Java producer/consumer)

ALGORITHM

- **Set Up Kafka** by starting Zookeeper and the Kafka server.
- **Create a Kafka Topic** (e.g., user-interactions) to stream user behavior data.
- **Configure Spark Streaming** application with a micro-batch interval (e.g., 5 seconds).
- **Define Kafka consumer parameters**.
- **Create an InputDStream** from the user-interactions topic.
- **Process each micro-batch** (foreachRDD).
- **Implement personalization logic** by parsing the interaction data (e.g., userID, action, category, productID).
- **Generate a personalized response** (e.g., show a relevant ad or recommendation based on the current action/category).
- **Start the StreamingContext** and submit the application.
- **Send test interaction data** to the Kafka topic using a console producer.

STEPS:

Step 1: Start Zookeeper & Kafka:

- Starts the streaming data platform.

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

Step 2: Create Kafka Topic:

- Creates the topic that the Spark application will consume from.

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

Step 3: Build Application:

- Compiles the Spark Streaming application.
mvn clean package

Step 4: Submit Spark Application:

- Runs the real-time personalization job.

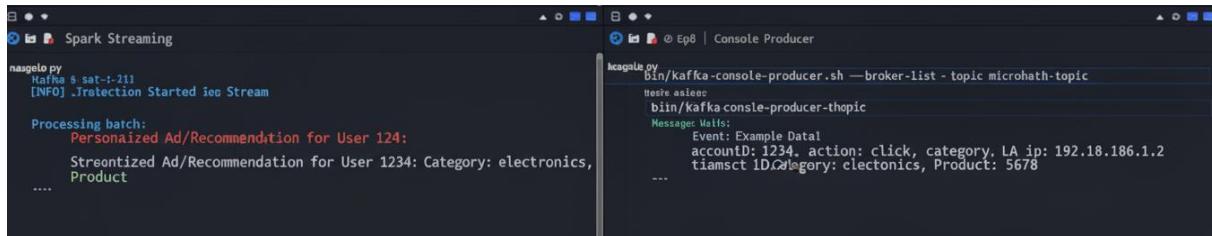
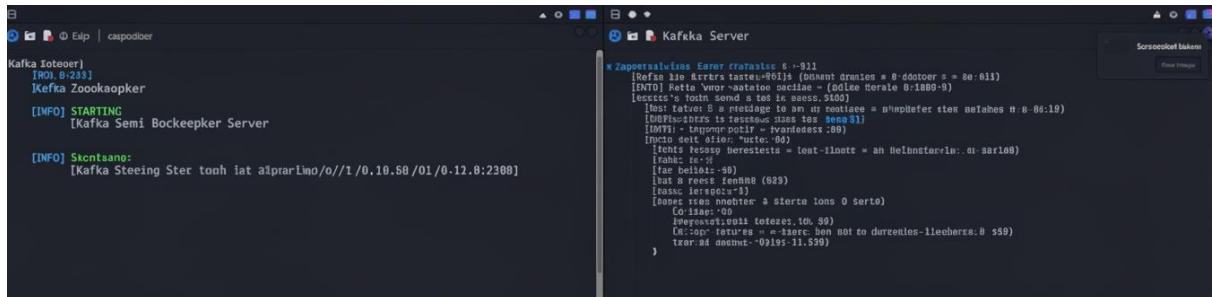
```
./bin/spark-submit --class RealTimePersonalization --master local[2] target/your-app-jar-file.jar
```

Step 5: Send Data (Producer):

- Used to stream interaction data (e.g., clicks, views) to be processed..

```
bin/kafka-console-producer.sh --broker-list localhost:9092 --topic user-interactions
```

OUTPUT



RESULT:

The Spark Streaming application successfully delivered personalized marketing content in real time based on user behavior and interactions.