**Real – Time Streaming Data Pipeline with Kafka and Docker**

Submitted by **Ram Charan Shashank Jangam**

Keywords: Docker, Apache Kafka, Zookeeper, Producer, Consumer, Data Generator, Processing

**Outline:**

This task involves setting up a real-time streaming data pipeline using Kafka and Docker. This procedure happens in three key steps: streaming real-time data, processing the collected data and storing the processed data in a new Kafka topic.

**Data Flow Design:**

1. Producer generates streaming data (user login events) to the `user-login` topic.

2. Consumer consumes the data from `user-login`, processes it, and pushes the transformed data to the `processed-data` topic.

3. Data Processing involves handling missing fields and transforming timestamps.

**Environment Setup:**

**Pre-req:** Install Docker Compose on your local machine: <https://docs.docker.com/compose/install/>

**Step1:** Setup a local development environment on Docker which includes Apache Kafka, Zookeeper and a data generator to simulate streaming data.

Initially create a remote project repository on GitHub or any version control tool.

On your terminal window clone the project repository:

$ git clone <https://github.com/ramcharanj3121/real-time-streaming-data-pipeline-with-kafka-and-docker.git>

$ cd real-time-streaming-data-pipeline-with-Kafka-and-docker/

Setup the Kafka and Zookeeper using the docker-compose.yml file provided:

version: '2' # Docker Compose version.

services:

  zookeeper:

    image: confluentinc/cp-zookeeper:latest # Zookeeper service.

    environment:

      ZOOKEEPER\_CLIENT\_PORT: 2181 # Port for Zookeeper.

      ZOOKEEPER\_TICK\_TIME: 2000 # Zookeeper tick time.

    ports:

      - 22181:2181 # Port mapping for Zookeeper.

    networks:

      - kafka-network # Connect to Kafka network.

  kafka:

    image: confluentinc/cp-kafka:latest # Kafka service.

    depends\_on:

      - zookeeper # Wait for Zookeeper.

    ports:

      - 9092:9092 # Internal Kafka port.

      - 29092:29092 # External Kafka port.

    networks:

      - kafka-network # Connect to Kafka network.

    environment:

      KAFKA\_BROKER\_ID: 0 # Kafka broker ID.

      KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181 # Zookeeper connection.

      KAFKA\_ADVERTISED\_LISTENERS: LISTENER\_INTERNAL://kafka:9092,LISTENER\_EXTERNAL://localhost:29092 # Listeners.

      KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: LISTENER\_INTERNAL:PLAINTEXT,LISTENER\_EXTERNAL:PLAINTEXT # Protocols.

      KAFKA\_INTER\_BROKER\_LISTENER\_NAME: LISTENER\_INTERNAL # Internal broker communication.

      KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1 # Replication factor.

      KAFKA\_TRANSACTION\_STATE\_LOG\_REPLICATION\_FACTOR: 1 # Transaction log replication.

      KAFKA\_TRANSACTION\_STATE\_LOG\_MIN\_ISR: 1 # Minimum in-sync replicas.

  my-python-producer:

    image: mpradeep954/fetch-de-data-gen # Custom Python producer.

    depends\_on:

      - kafka # Wait for Kafka.

    restart: on-failure:10 # Restart on failure.

    ports:

      - 9093:9093 # Producer port.

    environment:

      BOOTSTRAP\_SERVERS: kafka:9092 # Kafka connection.

      KAFKA\_TOPIC: user-login # Kafka topic.

    networks:

      - kafka-network # Connect to Kafka network.

networks:

  kafka-network:

    driver: bridge # Bridge network for services.

The following command will initiate Kafka, Zookeeper and the data generator:

$ docker-compose up

This will set up a local Kafka environment and a producer generating user login messages into the user-login topic.

**Step2:** Model a Kafka consumer to consume messages from a topic, perform basic data processing, and store the processed data in a new Kafka topic.

I have used Python scripting for writing the Kafka Consumer as:

Python Script:

from kafka import KafkaConsumer, KafkaProducer

import json

def process\_data(message):

    """Process incoming data and handle missing 'device\_type'."""

    try:

        data = json.loads(message) # Parse the JSON message.

        if 'device\_type' not in data:

            data['device\_type'] = 'unknown' # Assign default value if missing.

        return data

    except Exception as e:

        print(f"Error processing message: {e}") # Handle errors.

        return None

# Kafka consumer setup.

consumer = KafkaConsumer(

    'user-login', # Topic to consume.

    bootstrap\_servers='localhost:29092', # Kafka broker.

    auto\_offset\_reset='earliest', # Start from the earliest message.

    enable\_auto\_commit=True, # Auto commit offsets.

    group\_id='user-logins-group', # Consumer group ID.

    value\_deserializer=lambda x: x.decode('utf-8') # Deserialize message.

)

# Kafka producer setup.

producer = KafkaProducer(

    bootstrap\_servers='localhost:29092', # Kafka broker.

    value\_serializer=lambda x: json.dumps(x).encode('utf-8') # Serialize message.

)

# Process each consumed message.

for message in consumer:

    data = process\_data(message.value) # Process message.

    if data:

        producer.send('processed-user-login', value=data) # Send to new topic.

        print(f"Produced processed data: {data}") # Log the produced data.

Here, **try-except blocks** are used to handle missing or malformed fields, ensuring faulty data doesn’t crash the pipeline. We also have an added advantage of scaling the pipeline to balance the workload as the dataset grows by performing partitioning, group-based consumption and adding more consumers to user-logins-group.

In a new window, go to same directory

Create a consumer.py file with the above code and execute it using:  
$ python consumer.py

**Step3:** The pipeline should continuously ingest data efficiently and processes the data error-free.

**Step4:** Additionally, it is designed to handle missing fields and other potential errors in the streaming data.

Overall, the pipeline in composed of Kafka, Docker Compose (includes kafka, Zookeeper and other components) and a Kafka Consumer – written in Python. The kafka consumer will read, process and store data from the user-login topic into a new kafka topic(processed-user-login).

We can check the processed data by running the following command in a new terminal:

$docker exec -it kafka /bin/bash

kafka-console-consumer --bootstrap-server localhost:9092 --topic processed-user-login --from-beginning

Now that we have created a real-time-data- streaming pipeline, we will be able to see the logs. Sample logs of python-consumer and kafka-consumer are attached in the repository.

**Additionally, for:**

**Deployment:**

I would deploy this application by:

* Containerize all components (Kafka, Zookeeper, producer, consumer) using Docker.
* Use Kubernetes for orchestration, scaling, and self-healing.
* Manage environment configuration using variables and configuration files.
* Implement monitoring and logging with tools like Prometheus and ELK stack.
* Set up load balancing to evenly distribute traffic among Kafka producers and consumers.
* Establish regular data backups and disaster recovery plans.

**Production:**

To make it production ready:

* Add a schema registry to manage data schemas (e.g., Confluent Schema Registry).
* Implement TLS/SSL encryption, authentication, and authorization for security.
* Configure auto-scaling for Kafka consumer instances based on load.
* Add retry mechanisms for error handling to ensure reliable data processing.
* Introduce data quality checks to validate incoming data.

**Scalability:**

As the dataset keeps growing, to make this application scalable:

* Use Kafka topic partitioning to enable horizontal scaling.
* Deploy more consumer instances for parallel data processing.
* Implement Kafka Streams for scalable real-time data processing.
* Scale Kafka producers to handle higher data generation.
* Consider using managed Kafka services (e.g., Confluent Cloud, AWS MSK) for easier scalability.