Real-Time Data Streaming with Apache Kafka

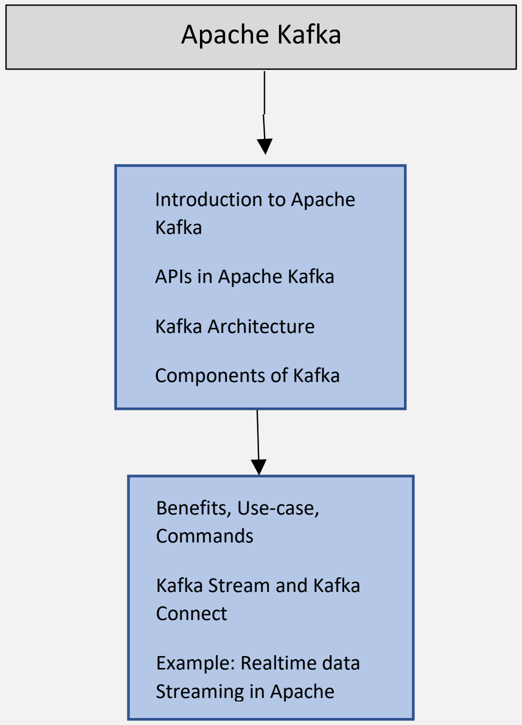
Apache Kafka (Kafka) is an open source, distributed streaming platform that enables (among other things) the development of real-time, event-driven applications.

As a part of Realtime Data Streaming with Apache Kafka, you covered:

- Introduction to Apache Kafka
- Kafka Benefits and Use-cases
- Apache Kafka Architecture and Components
- Example: Realtime data streaming with Apache Kafka

Common Interview Questions:

1. What are some of the features of Kafka?
2. What are the major components of Kafka?
3. Explain the four core API architecture that Kafka uses.
4. Tell me about some of the real-world usages of Apache Kafka.
5. What do you mean by zookeeper in Kafka and what are its uses?
6. Describe partitioning key in Kafka.
7. What are the traditional methods of message transfer? How is Kafka better from them?
8. Why is Topic Replication important in Kafka? What do you mean by ISR in Kafka?

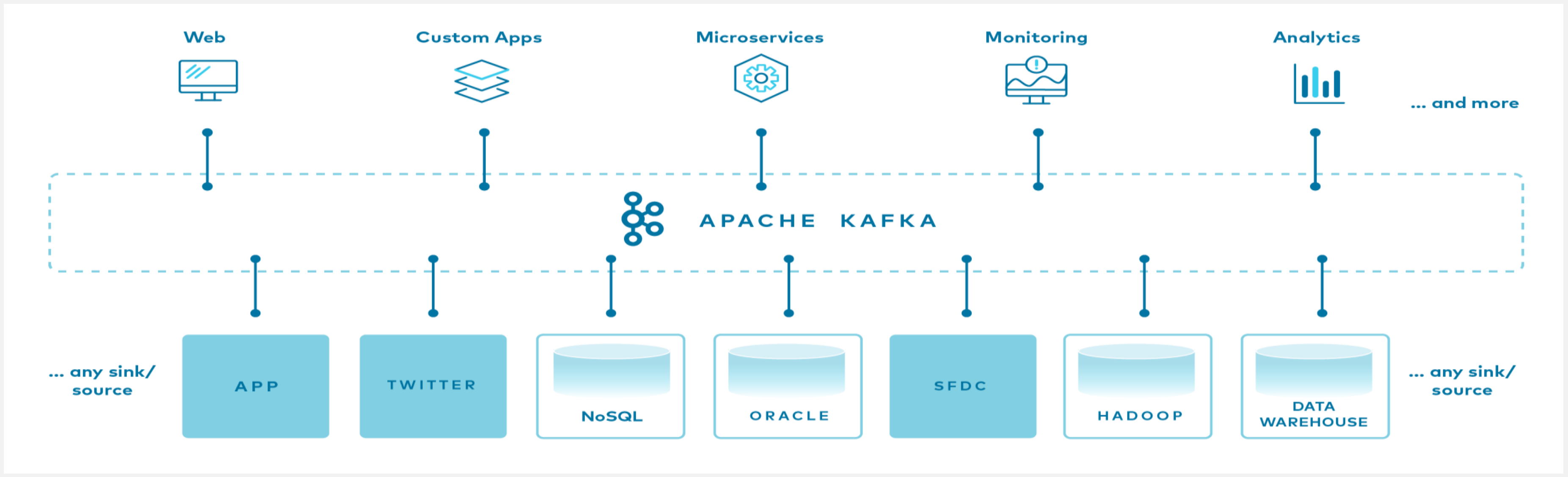

Apache Kafka
Introduction to Apache Kafka
APIs in Apache Kafka
Kafka Architecture
Components of Kafka
Benefits, Use-case, Commands
Kafka Stream and Kafka Connect
Example: Realtime data Streaming in Apache

Apache Kafka:

Apache Kafka (Kafka) is an open source, distributed streaming platform that enables (among other things) the development of real-time, event-driven applications.

Why Apache Kafka?

The platform is typically used to build real-time streaming data pipelines that support streaming analytics and mission-critical use cases with guaranteed ordering, no message loss, and exactly-once processing. Apache Kafka is massively scalable because it allows data to be distributed across multiple servers, and it's extremely fast because it decouples data streams, which results in low latency. It can also distribute and replicate partitions across many servers, which protects against server failure.



APIs in Apache Kafka:

Kafka has three primary capabilities:

- It enables applications to publish or subscribe to data or event streams.
- It stores records accurately (i.e., in the order in which they occurred) in a fault-tolerant and durable way.
- It processes records in real-time (as they occur).

Developers can leverage these Kafka capabilities through APIs:

Producer API: This enables an application to publish a stream to a Kafka topic. After a record is written to a topic, it can't be altered or deleted; instead, it remains in the topic for a preconfigured amount of time—for example, for two days—or until storage space runs out.

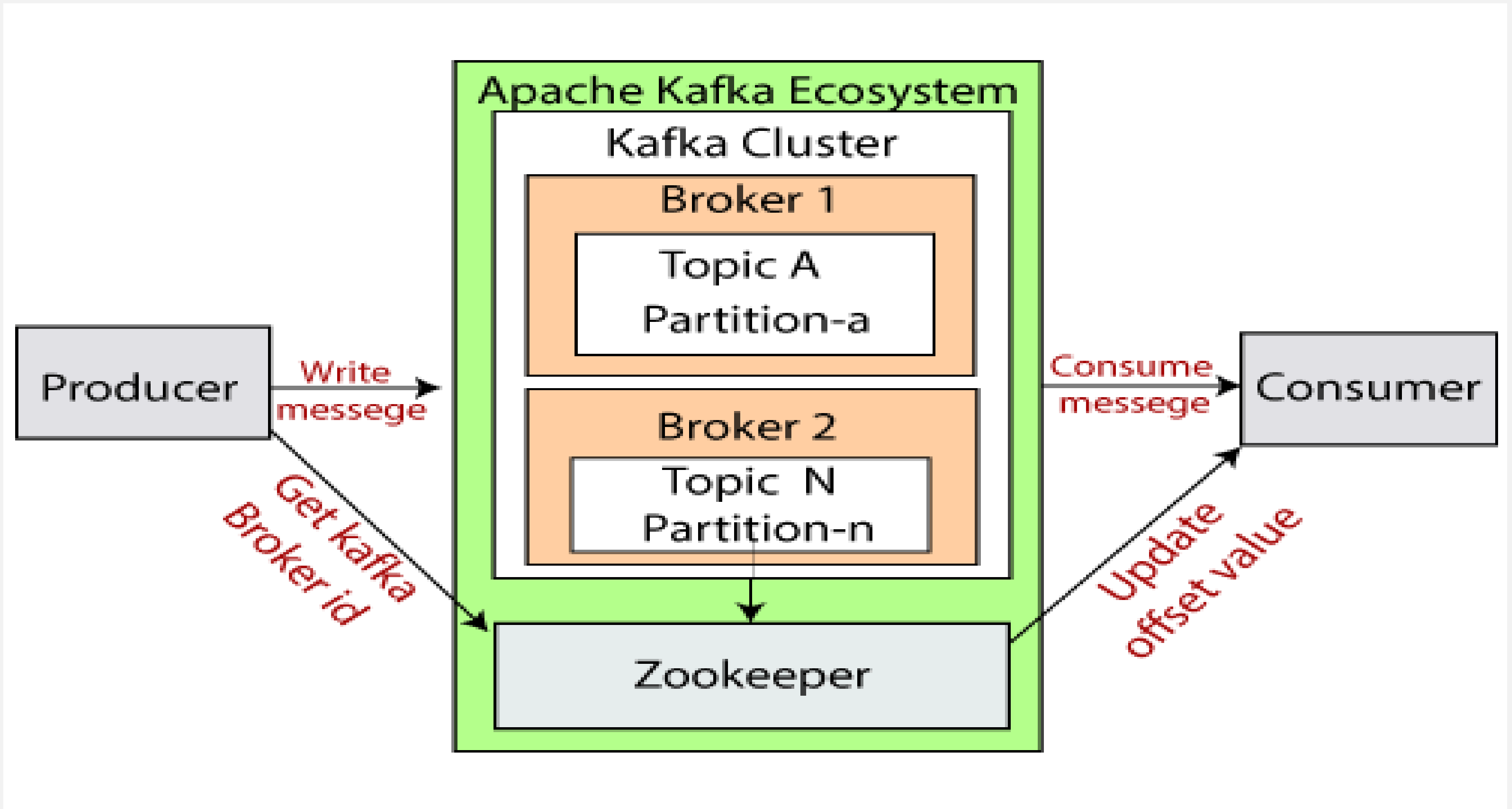
Consumer API: This enables an application to subscribe to one or more topics and to ingest and process the stream stored in the topic. It can work with records in the topic in real-time, or it can ingest and process past records.

Streams API: This builds on the Producer and Consumer APIs and adds complex processing capabilities that enable an application to perform continuous, front-to-back stream processing—specifically, to consume records from one or more topics, to analyse or aggregate or transform them as required, and to publish resulting streams to the same topics or other topics. While the Producer and Consumer APIs can be used for simple stream processing, it's the Streams API that enables development of more sophisticated data- and event-streaming applications.

Connector API: This lets developers build connectors, which are reusable producers or consumers that simplify and automate the integration of a data source into a Kafka cluster.

Admin API: The Admin API supports managing and inspecting topics, brokers, acls, and other Kafka objects

Apache Kafka Architecture:



Apache Kafka Architecture

Real-Time Data Streaming with Apache Kafka

Components of Apache Kafka:

Kafka Cluster: A Kafka cluster is a system that comprises of different brokers, topics, and their respective partitions. Data is written to the topic within the cluster and read by the cluster itself.

Producers: A producer sends or writes data/messages to the topic within the cluster. In order to store a huge amount of data, different producers within an application send data to the Kafka cluster.

Consumers: A consumer is the one that reads or consumes messages from the Kafka cluster. There can be several consumers consuming different types of data form the cluster. The beauty of Kafka is that each consumer knows from where it needs to consume the data.

Brokers: A Kafka server is known as a broker. A broker is a bridge between producers and consumers. If a producer wishes to write data to the cluster, it is sent to the Kafka server. All brokers lie within a Kafka cluster itself. Also, there can be multiple brokers.

Topics: It is a common name or a heading given to represent a similar type of data. In Apache Kafka, there can be multiple topics in a cluster. Each topic specifies different types of messages.

Partitions: The data or message is divided into small subparts, known as partitions. Each partition carries data within it having an offset value. The data is always written in a sequential manner. We can have an infinite number of partitions with infinite offset values. However, it is not guaranteed that to which partition the message will be written.

ZooKeeper: A ZooKeeper is used to store information about the Kafka cluster and details of the consumer clients. It manages brokers by maintaining a list of them. Also, a ZooKeeper is responsible for choosing a leader for the partitions. If any changes like a broker die, new topics, etc., occurs, the ZooKeeper sends notifications to Apache Kafka. A ZooKeeper is designed to operate with an odd number of Kafka servers. Zookeeper has a leader server that handles all the writes, and rest of the servers are the followers who handle all the reads.

Real-Time Data Streaming with Apache Kafka

Apache Kafka Benefits:

Apache Kafka benefits:

- **Reliability** – Kafka is distributed, partitioned, replicated and fault tolerance.
- **Scalability** – Kafka messaging system scales easily without down time.
- **Durability** – Kafka uses Distributed commit log which means messages persists on disk as fast as possible, hence it is durable.
- **Performance** – Kafka has high throughput for both publishing and subscribing messages. It maintains stable performance even many TB of messages are stored.

Apache Kafka Use-case:

Apache Kafka some Use-Case Examples:

- IoT and sensor networks
- Large scale message processing
- Stream processing
- Processing business and user event streams in real time
- Aggregating metrics and logs from distributed servers and applications
- Manage and distribute content to multiple applications in real-time
- Broker data to analytics clusters, real-time web analytics, and real-time predictive analytic

Apache Kafka Commands:

Apache Kafka CLI Commands:

To List All Topics in a Kafka Cluster: *$ kafka-topics *

--bootstrap-server localhost:9092 \ --list

To Delete a Topic From Kafka*: $ kafka-topics *

*--bootstrap-server localhost:9092,localhost:9093,localhost:9094 *

--delete \ --topic topic_for_deletion

To Produce Data to a Kafka Topic: *$ kafka-console-producer *

*--bootstrap-server localhost:9092 *

--topic topic-name

To Consume Data From a Kafka Topic: *$ kafka-console-consumer *

*--bootstrap-server localhost:9092 *

*--topic topic-name *

--from-beginning

Apache Kafka Python Libraries:

Kafka-Python: This is an open-source library designed by the Python community.

PyKafka: This library is maintained by Parsly and it has claimed to be a Pythonic API. However, we cannot create dynamic topics in this library like Kafka-Python.

Confluent Python Kafka: This library is provided by Confluent as a thin wrapper around librdkafka. Thus, it performs better than the above two.

$ pip install kafka-python

from kafka import KafkaProducer

from kafka import KafkaConsumer

Real-Time Data Streaming with Apache Kafka

Apache Kafka Stream:

Apache Kafka Stream:

Kafka Streams, or the Streams API, makes it easier to transform or filter data from one Kafka topic and publish it to another Kafka topic, although you can use Streams for sending events to external systems if you wish.

You can think of Kafka Streams as a Java-based toolkit that lets you change and modify messages in Kafka in real time, before the messages reach your external consumers.

To use Kafka Streams, you first import it into your Java application as a library (JAR file). The library gives you the Kafka Streams Java API.

With Kafka Streams, all your stream processing takes place inside your app, not on the brokers. You can even run multiple instances of your Kafka Streams-based application if you've got a firehose of messages and you need to handle high volumes.

Apache Kafka Connect:

Apache Kafka some Connect:

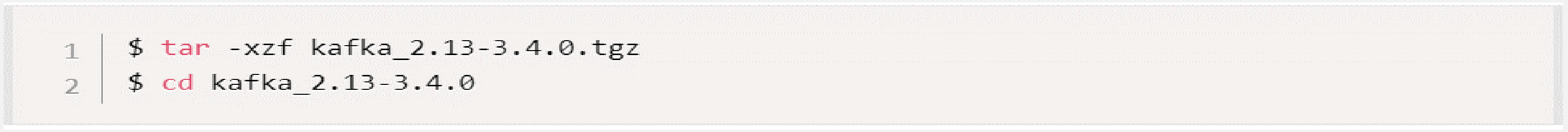
Kafka Connect is a tool for connecting different input and output systems to Kafka. Think of it like an engine that can run a number of different components, which can stream Kafka messages into databases, Lambda functions, S3 buckets, or apps like Elasticsearch or Snowflake.

To use Kafka Connect, you download the Connect distribution, set the configuration files how you want them, and then start a Kafka Connect instance.

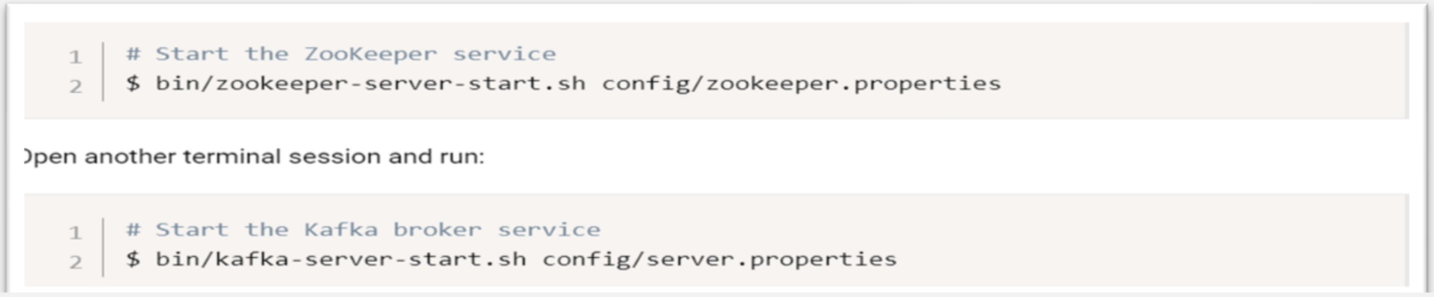
You can also run Kafka Connect in containers. Some projects which use Kafka Connect, offer their own pre-built Docker image. Debezium has a ready-made Connect image that you can pull and run.

Example on Creating a Real-Time Data Streaming with Apache Kafka:

Installing Kafka:



```
$ tar -xzf kafka_2.13-3.4.0.tgz
$ cd kafka_2.13-3.4.0
```



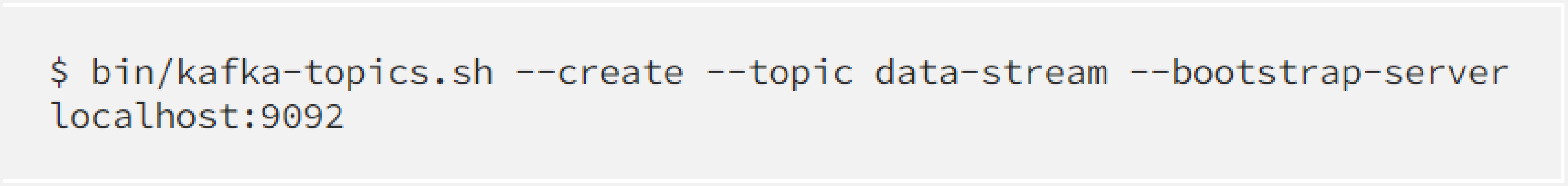
```
# Start the ZooKeeper service
$ bin/zookeeper-server-start.sh config/zookeeper.properties
```

pen another terminal session and run:

```
# Start the Kafka broker service
$ bin/kafka-server-start.sh config/server.properties
```

Creating a Topic:

Creating a topic named “data-stream”:



```
$ bin/kafka-topics.sh --create --topic data-stream --bootstrap-server
localhost:9092
```

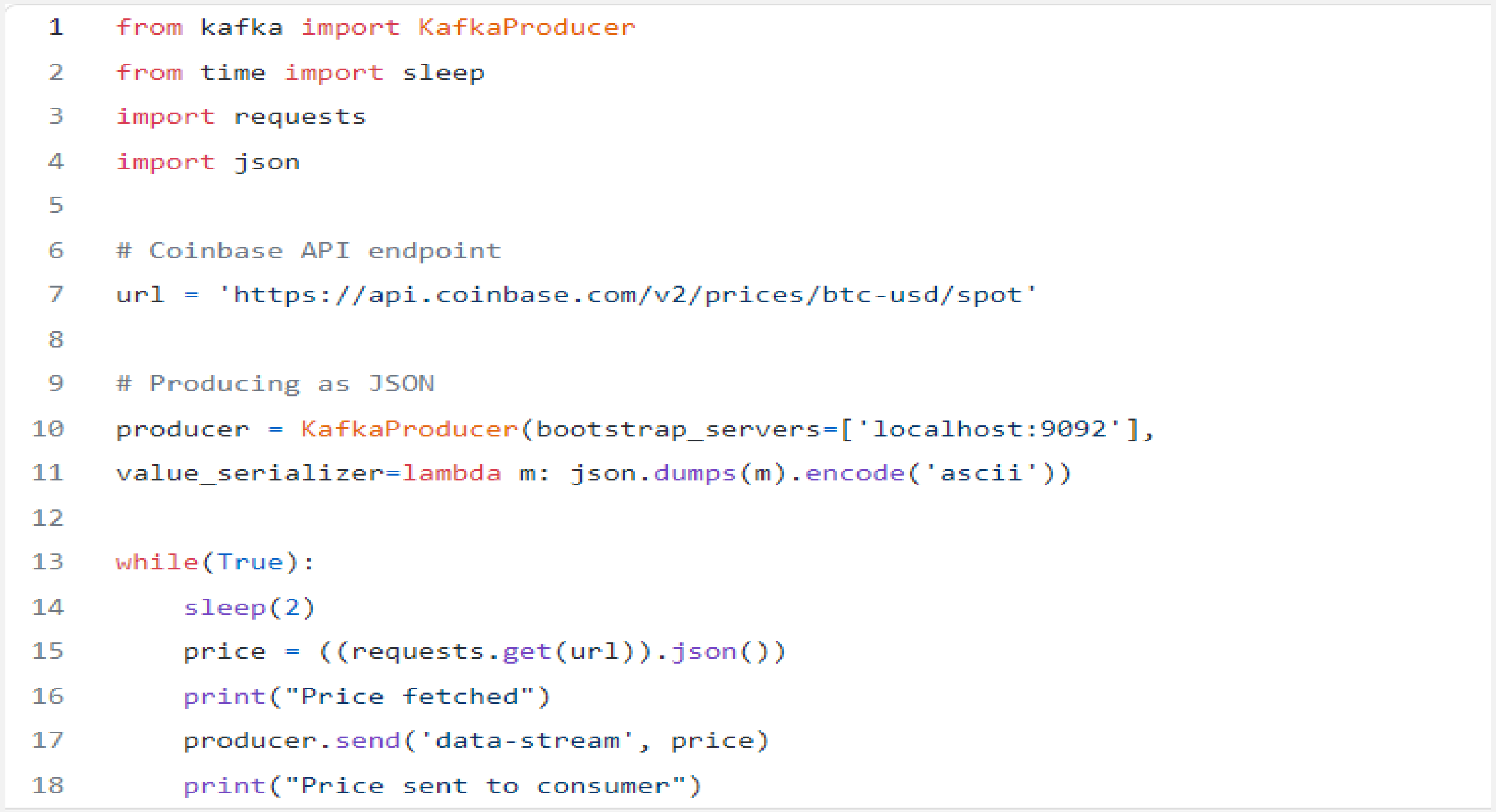
Real-Time Data Streaming with Apache Kafka

Streaming Data using Python library Kafka-python:

For now, we just need to know two things about Kafka:

- A producer is something that produces data. Pretty straightforward. The producer is where we fetch the data and send it to the Kafka cluster in real time.
- A consumer reads the data sent by the producer. Here, we can do whatever we want with the data we keep receiving from the producer.

Creating Producer.py:



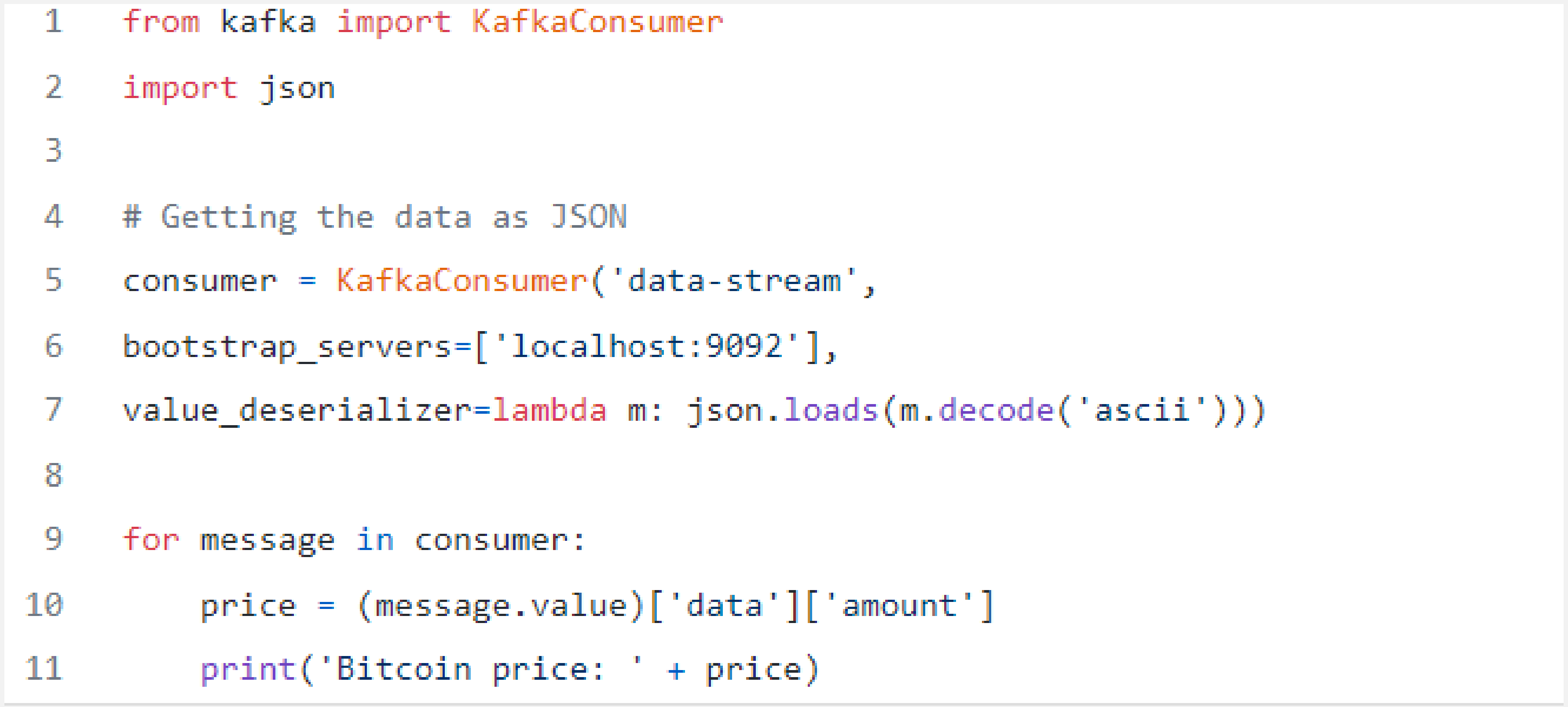
```python
from kafka import KafkaProducer
from time import sleep
import requests
import json

# Coinbase API endpoint
url = 'https://api.coinbase.com/v2/prices/btc-usd/spot'

# Producing as JSON
producer = KafkaProducer(bootstrap_servers=['localhost:9092'],
value_serializer=lambda m: json.dumps(m).encode('ascii'))

while(True):
    sleep(2)
    price = ((requests.get(url)).json())
    print("Price fetched")
    producer.send('data-stream', price)
    print("Price sent to consumer")
```

Creating Consumer.py:



```python
from kafka import KafkaConsumer
import json

# Getting the data as JSON
consumer = KafkaConsumer('data-stream',
bootstrap_servers=['localhost:9092'],
value_deserializer=lambda m: json.loads(m.decode('ascii')))

for message in consumer:
    price = (message.value)['data']['amount']
    print('Bitcoin price: ' + price)
```

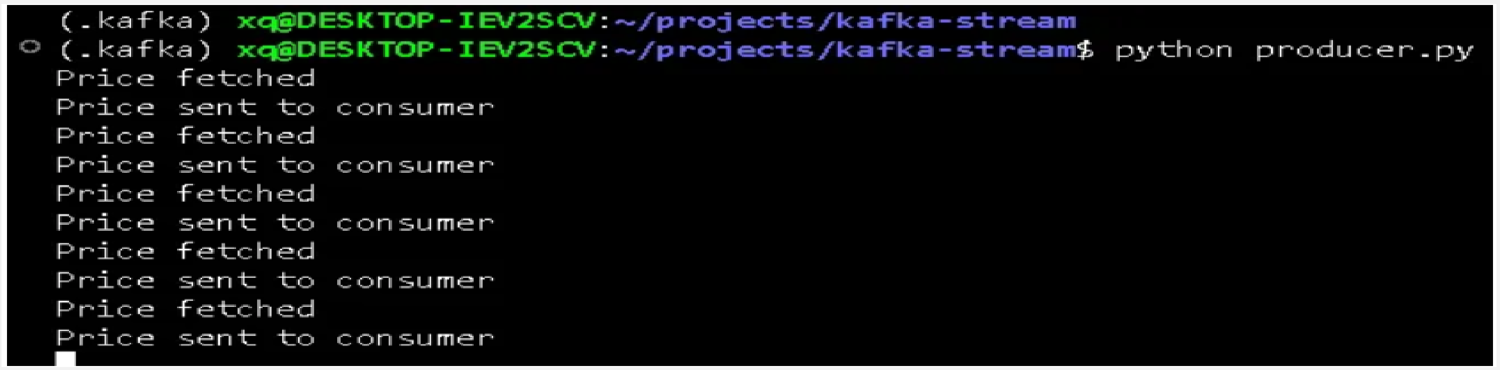
- message.value contains the actual JSON object received. You can filter it further to get the field you want. So, you must know the JSON structure beforehand. The price value is stored under the "amount" parameter in this case.
- Once you read the data in your consumer, you can do anything with it. For example, you can extract some variables from this data and pass it on to a deep learning model to predict something and print your prediction.
- You could write some logic and send an alert if some conditions are met. You could also plot the data as a graph on some web page to get a real-time graph.

Output:

Now, Run both the producer and the consumer in separate terminals.

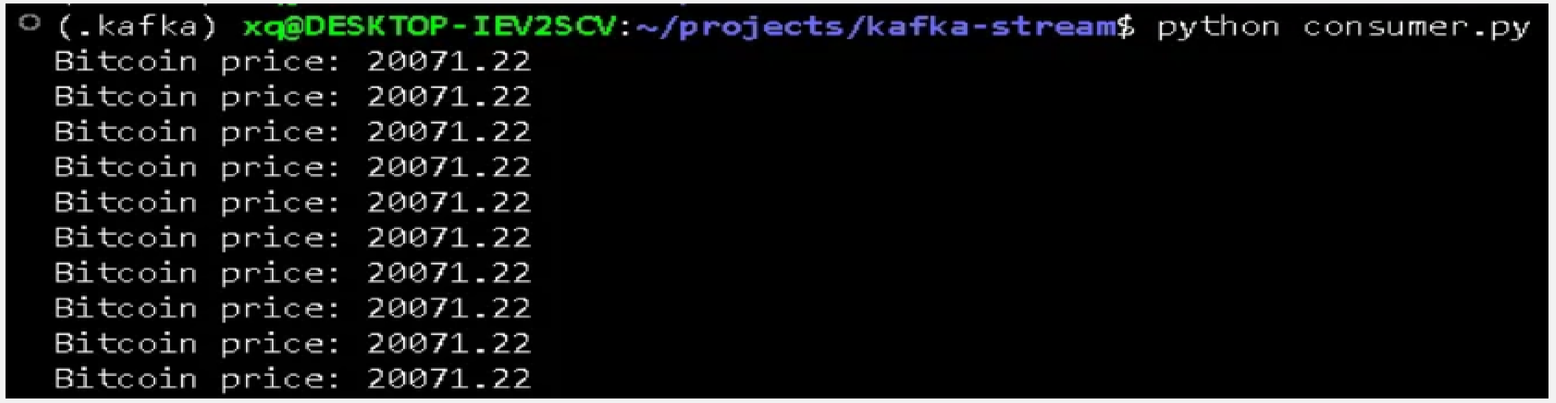
Note that the other terminals for Zookeeper and Kafka servers need to be running in the background.

Running Producer.py:



```
  (.kafka) xq@DESKTOP-IEV2SCV:~/projects/kafka-stream
○ (.kafka) xq@DESKTOP-IEV2SCV:~/projects/kafka-stream$ python producer.py
  Price fetched
  Price sent to consumer
  Price fetched
  Price sent to consumer
  Price fetched
  Price sent to consumer
  Price fetched
  Price sent to consumer
  Price fetched
  Price sent to consumer
```

Running Consumer.py:



```
○ (.kafka) xq@DESKTOP-IEV2SCV:~/projects/kafka-stream$ python consumer.py
  Bitcoin price: 20071.22
  Bitcoin price: 20071.22
  Bitcoin price: 20071.22
  Bitcoin price: 20071.22
  Bitcoin price: 20071.22
  Bitcoin price: 20071.22
  Bitcoin price: 20071.22
  Bitcoin price: 20071.22
  Bitcoin price: 20071.22
  Bitcoin price: 20071.22
```