Final Project Report

Course: Big Data Analytics

Topic: Zeta Architecture

Submitted By:

Muhammad Zain Rajani (23379)

Muhammad Usman Khan (23391)

Zeta Architecture

Zζ

Domain Description

Zeta architecture is an enterprise architecture that offers a scalable way to integrate data for a business. Various components of the architecture, when properly deployed, help to reduce the complexity of systems and distribute data more efficiently. It represents a new modern data architecture that comprehensively supports a variety of solution architectures and enterprise applications that work together. Zeta architecture is an ideal implementation model that captures the importance of containerization as an inherent part of data center deployment.

The components of Zeta architecture include a distributed file system, real-time data storage and a pluggable compute model/execution engine, as well as data containers, enterprise applications and resource management tools

Below is an example of how Google uses the technology stack in zeta architecture in some of Google's services such as Gmail. Proposed architecture is built on pluggable components. All together, they produce a holistic architecture.

Objective

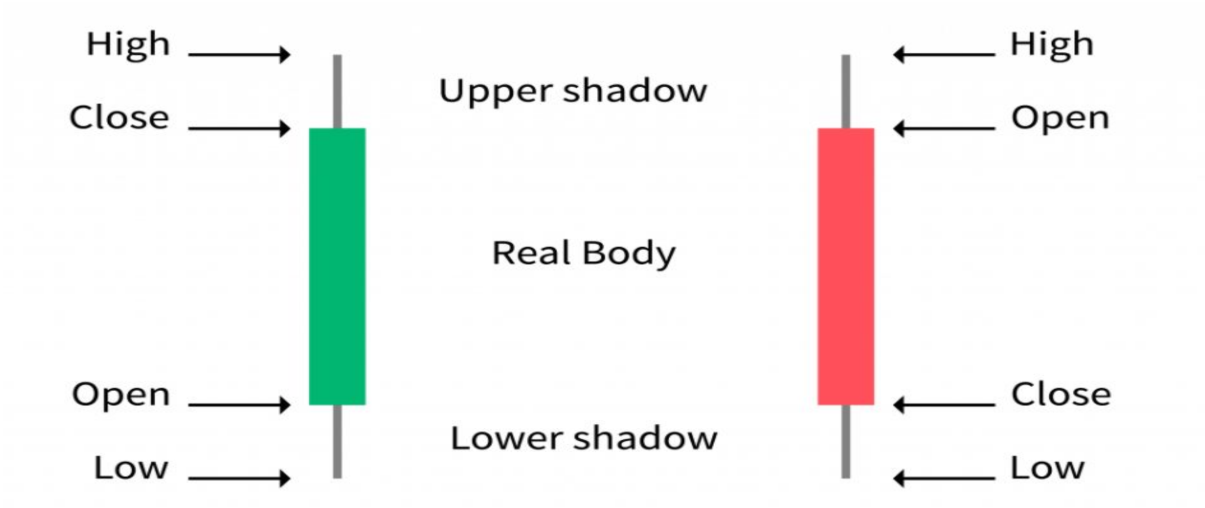
Our aim is to implement a Kappa Architecture use case by using dockerized containers from a pool of all available containers. We intend to show the use of pluggable architecture by using containers required to fulfill our objective. We will try to replace some containers and put in place some alternatives as a proof-of-concept of plug and play architecture.

Use Case

We have picked up a use case of real time analytics of cryptocurrencies data. For the scope of this project, we will try to simulate real time streaming data by making continuous call to our dataset via some API. Finally, we will present some visualization that will demonstrate real time ingestion and processing of the dataset. Our final output will show prices, trade or volume-based indicators reflecting the input dataset.

Dataset Description

The dataset has 1-minute candlesticks[1] data for 999 cryptocurrencies taken from binance.com. For every trading pair, the historical candlestick data is saved into a parquet file. That means for 999 cryptocurrencies, we will have 999 files.
Candlesticks are one of the most popular ways for investors and traders to understand the price movements of assets in the crypto market. The main features of a candlestick are visually demonstrated through this diagram



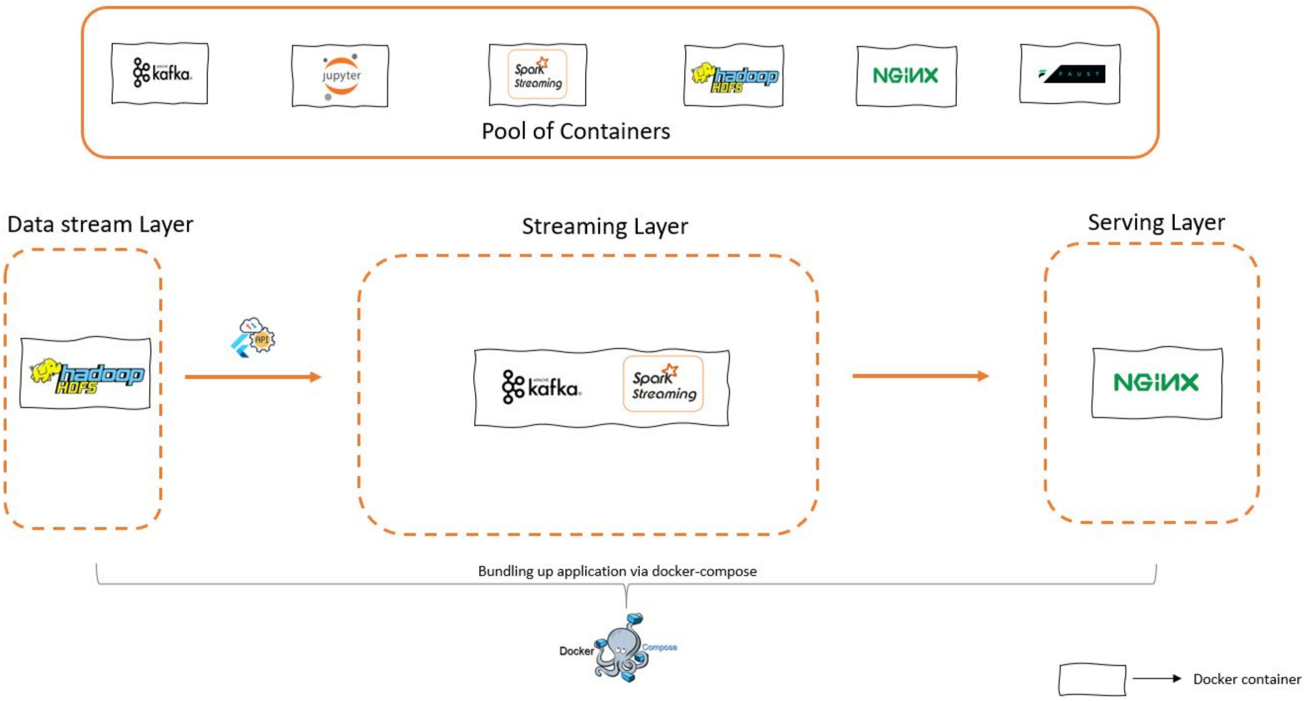
A candlestick becomes green when the current or closing price rises above its opening price, whereas, it becomes red when its current or closing price falls below the opening price. The dataset consists of the following fields:-

No.	Column	Description	Dtype
1	open_time	recording time of data	datetime64[ns]
2	Open	price of an asset when the trading period begins	float32
3	High	price of an asset when the trading period has concluded	float32
4	Low	highest price achieved in the trading period	float32
5	Close	lowest price achieved in the trading period	float32
6	Volume	total amount of coins traded in the trading period	float32
7	quote_asset_volume	volume in the second part in the pair i.e. BTC/USDT - quote volume would be in USDT	float32
8	number_of_trades	count of trades performed in the trading period	uint16
9	taker_buy_base_asset_volume	amount of coins received by the buyer	float32
10	taker_buy_quote_asset_volume	amount paid by the buyer in btc/eth/usdt depending on the market	float32

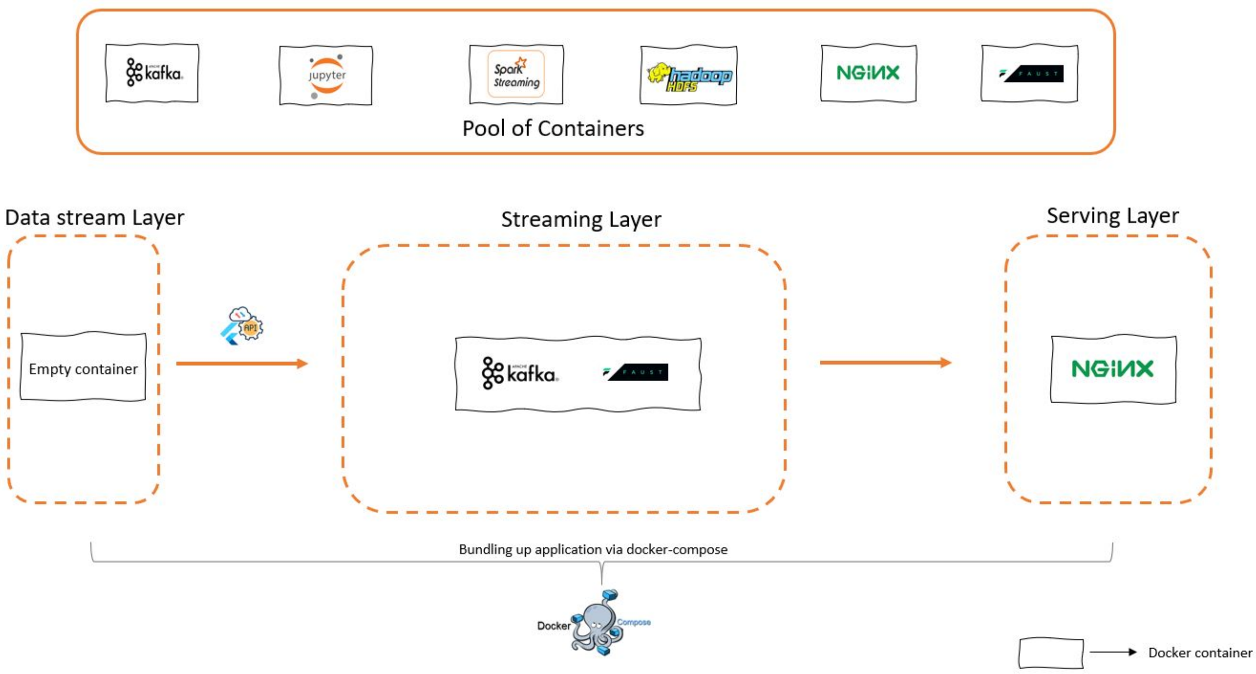
Link: https://www.kaggle.com/jorijnsmit/binance-full-history

Architecture

Use Case 1



Use Case 2



Technology Stack

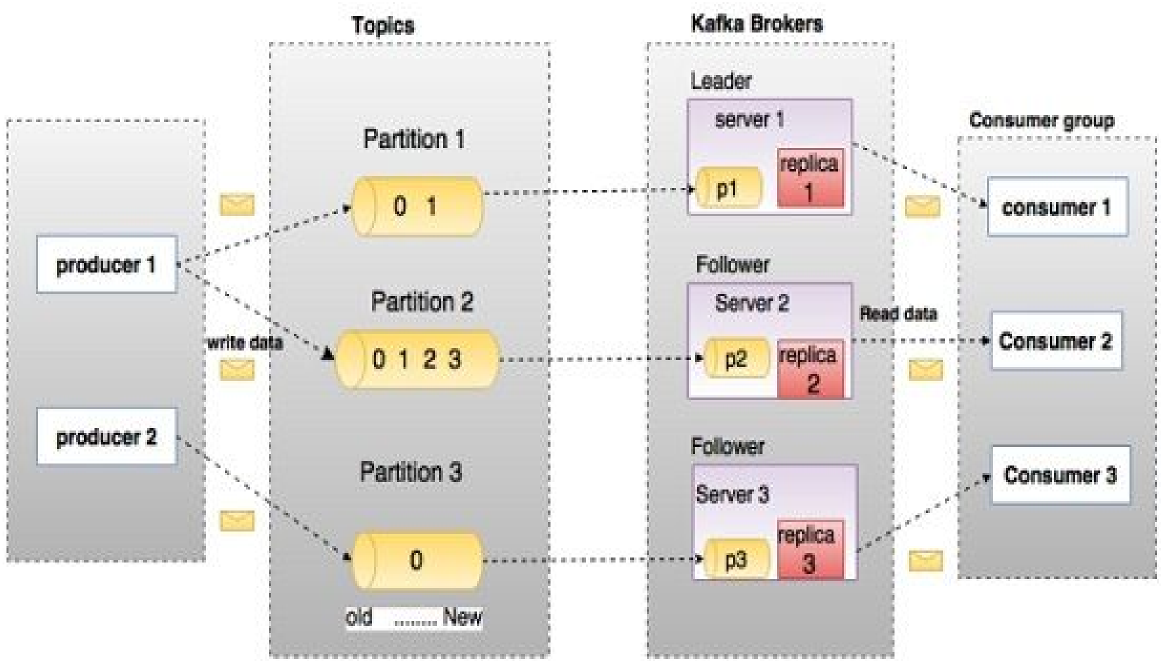
Apache Kafka

In Big Data, an enormous volume of data is used. Regarding data, we have two main challenges.The first challenge is how to collect large volume of data and the second challenge is to analyze the collected data. To overcome those challenges, you must need a messaging system.

In the messaging system, messages are queued asynchronously between client applications and messaging system. One of the pattern of messaging system is a publish-subscribe messaging system. In this system, message producers are called publishers and message consumers are called subscribers.

Apache Kafka is a distributed publish-subscribe messaging system and a robust queue that can handle a high volume of data and enables you to pass messages from one end-point to another. Kafka is suitable for both offline and online message consumption. Kafka messages are persisted on the disk and replicated within the cluster to prevent data loss. Kafka is built on top of the ZooKeeper synchronization service. It integrates very well with Apache Storm and Spark for real-time streaming data analysis.

Before moving forward, we need to be aware of the main terminologies within Kafka such as topics, brokers, producers and consumers. Below is an illustration of the main components of Kafka

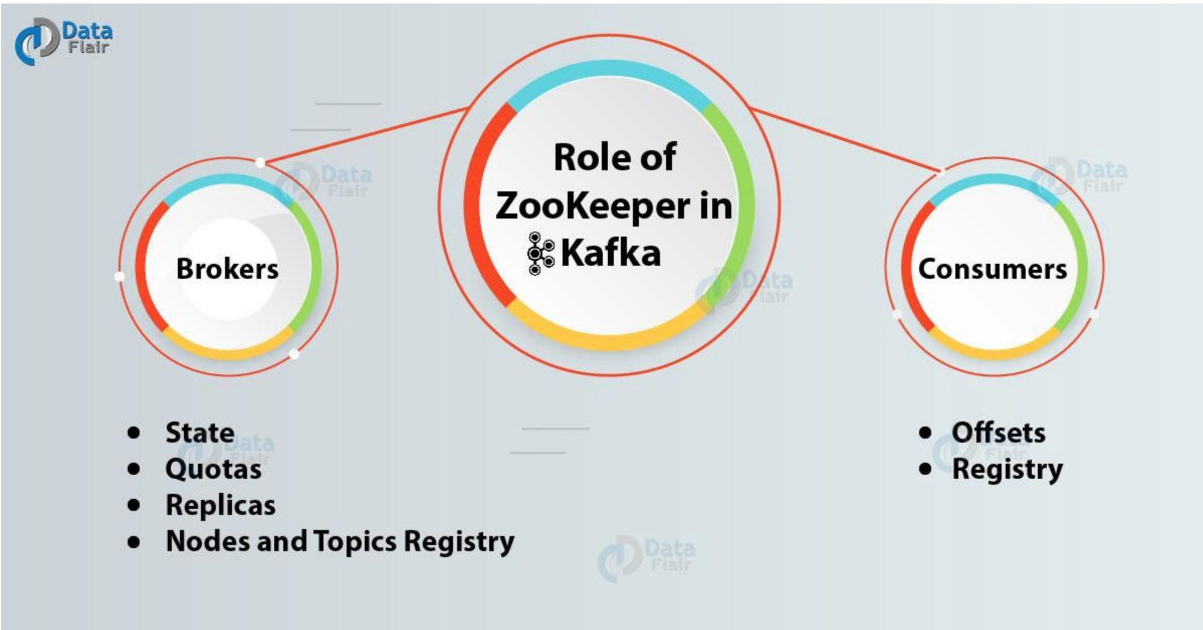


S. No	Components	Description
1	Topics	A stream of messages belonging to a particular category is called a topic. Data is stored in topics.
2	Broker	Brokers are a simple system responsible for maintaining the pub-lished data. Every instance of Kafka that is responsible for message exchange is called a Broker
3	Producers	Producers are the publisher of messages to one or more Kafka topics. Producers send data to Kafka brokers
4	Consumers	Consumers read data from brokers. Consumers subscribe to one or more topics and consume published messages by pulling data from the brokers.

Apache ZooKeeper

ZooKeeper is an open source Apache project that provides a centralized service for providing configuration information, naming, synchronization and group services over large clusters in distributed systemsIn our use case, ZooKeeper is used for managing and coordinating Kafka broker.

ZooKeeper service is mainly used to notify producer and consumer about the presence of any new broker in the Kafka system or failure of the broker in the Kafka system. As per the notification received by the Zookeeper regarding presence or failure of the broker then pro-ducer and consumer takes decision and starts coordinating their task with some other broker.



Hadoop HDFS

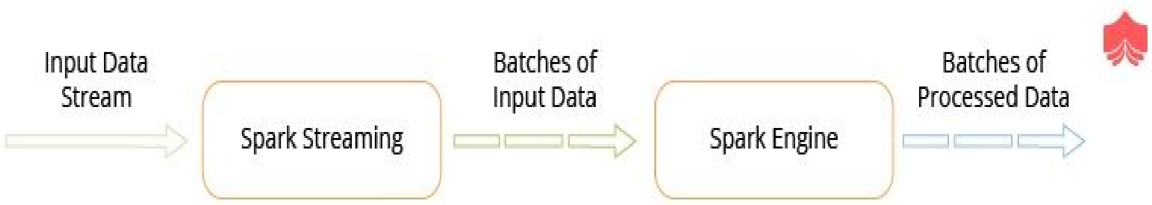
The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets. HDFS relaxes a few POSIX requirements to enable streaming access to file system data.

Applications that run on HDFS have large data sets. A typical file in HDFS is gigabytes to terabytes in size. Thus, HDFS is tuned to support large files. It should provide high aggregate data bandwidth and scale to hundreds of nodes in a single cluster. It should support tens of millions of files in a single instance.

Spark Streaming

Spark Streaming supports real time processing of streaming data, such as production web server log files (e.g. Apache Flume and HDFS/S3), social media like Twitter, and various messaging queues like Kafka. Under the hood, Spark Streaming receives the input data streams and divides the data into batches. Next, they get processed by the Spark engine and generate a final stream of results in batches, as depicted below.

Spark Streaming receives live input data streams, it collects data for some time, builds Resilient Distributed Dataset (RDD), divides the data into micro-batches, which are then processed by the Spark engine to generate the final stream of results in micro-batches. Following data flow diagram explains the working of Spark streaming.

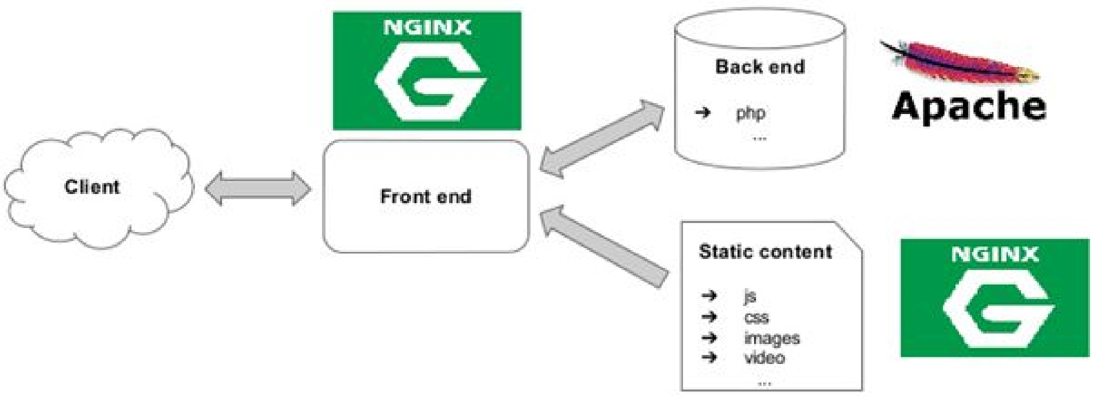


Spark Streaming provides a high-level abstraction called discretized stream or DStream, which represents a continuous stream of data. DStreams can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs. Think about RDD as the underlying concept for distributing data over a cluster of computers.

NGINX

Why not Apache Web Service be used? Here we want to use different containers to test our environment in Zeta Architecture so we decide to choose another flavor. NGINX is a free, open-source, high-performance HTTP server and reverse proxy, as well as an IMAP/POP3 proxy server. NGINX is known for its high performance, stability, rich feature set, simple configuration, and low resource consumption. In our case we will display all our processed data in web page on a dashboard.

NGINX is one of a handful of servers written to address the C10K problem. Unlike traditional servers, NGINX doesn't rely on threads to handle requests. Instead it uses a much more scalable event-driven (asynchronous) architecture. This architecture uses small, but more importantly, predictable amounts of memory under load. Even if you don't expect to handle thousands of simultaneous requests, you can still benefit from NGINX's high-performance and small memory footprint. NGINX scales in all directions: from the smallest VPS all the way up to large clusters of servers.



WorkFlow

Use Case 1

For this scenario we have intended to perform the following steps:

1. Use Hadoop hdfs container to store parquet files from local system to hdfs container
2. Start zookeeper and kafka server
3. Start producer and consumer for kafka
4. start spark streaming container that will read data from kafka consumer
5. Aggregate values and show updated sum of values to nginx server after specific interval

Steps

Open directory **“BDA Project”** in linux terminal

Execute ‘docker-compose-uc1.yml’ by executing this command:

`docker-compose -f docker-compose-uc1.yml up -d`

```
zain@zain-Inspiron-5593:~$ cd "BDA Project"
zain@zain-Inspiron-5593:~/BDA Project$ docker-compose -f docker-compose-uc1.yml up -d
Starting hadoop-local ... done

Starting zookeeper    ... done

Starting kafka        ... done

Starting spark        ... done
```

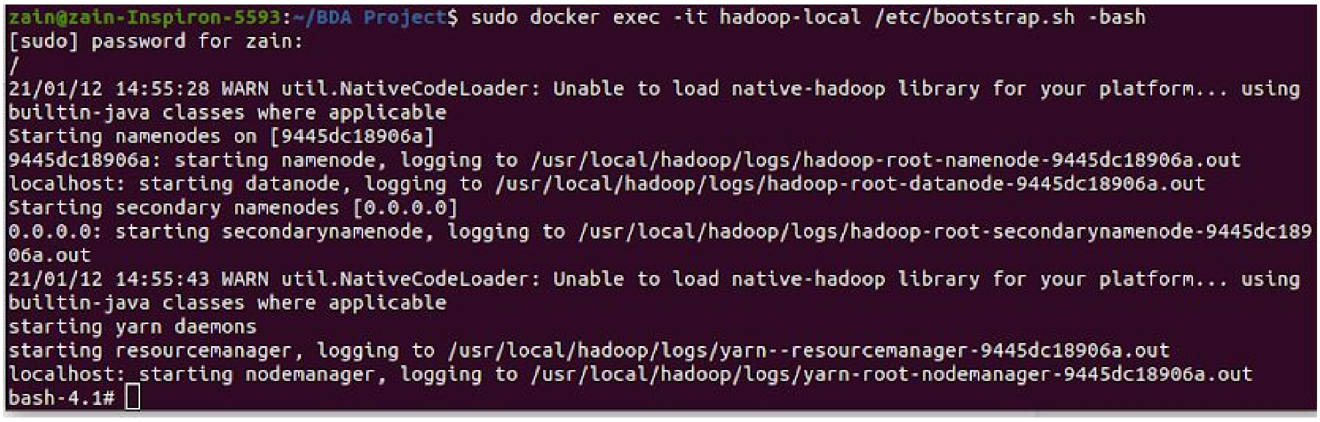
Docker container status:

`Docker ps`

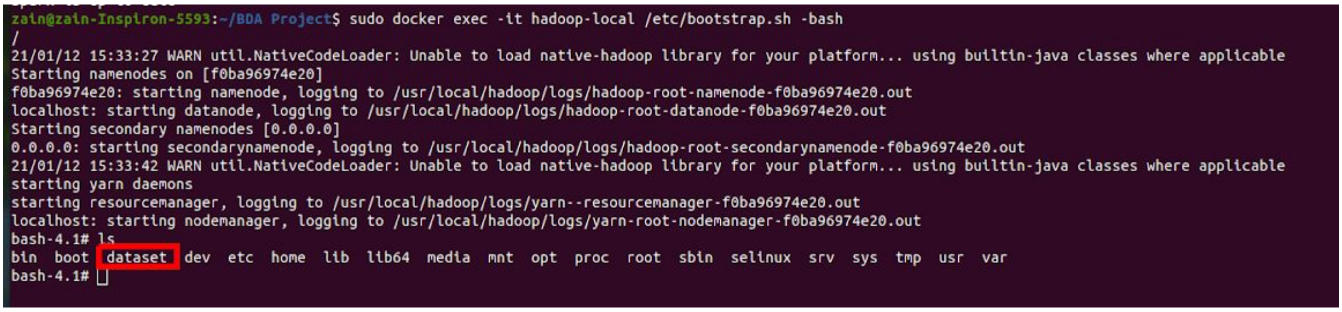
```
zain@zain-Inspiron-5593:~/BDA Project$ docker ps
CONTAINER ID   IMAGE                          COMMAND                  CREATED         STATUS         PORTS


                         NAMES
041177e336e9   jupyter/pyspark-notebook       "tini -g -- start-no…"   2 hours ago   Up 4 minutes   0.0.0.0:
4040-4041->4040-4041/tcp, 0.0.0.0:8888->8888/tcp

                         spark
c8cf7fcd3ec1   wurstmeister/kafka:0.10.2.0    "start-kafka.sh"         2 hours ago   Up 4 minutes   0.0.0.0:
9092->9092/tcp

                         kafka
9445dc18906a   sequenceiq/hadoop-docker:2.7.1   "/etc/bootstrap.sh -d"   2 hours ago   Up 4 minutes   2122/tcp
, 8030-8033/tcp, 8040/tcp, 8042/tcp, 0.0.0.0:8020->8020/tcp, 8088/tcp, 19888/tcp, 0.0.0.0:9000->9000/tcp, 0.0.
0.0:50010->50010/tcp, 0.0.0.0:50020->50020/tcp, 0.0.0.0:50070->50070/tcp, 0.0.0.0:50075->50075/tcp, 49707/tcp,
 0.0.0.0:50090->50090/tcp   hadoop-local
603790aa27e1   wurstmeister/zookeeper         "/bin/sh -c '/usr/sb…"   2 hours ago   Up 4 minutes   22/tcp,
2888/tcp, 3888/tcp, 0.0.0.0:2181->2181/tcp

                         zookeeper
```

All of the containers are up and running, now we will go to bash terminal in ‘hadoop-local’ container by execution this command:

```
sudo docker exec -it hadoop-local /etc/bootstrap.sh -bash
```

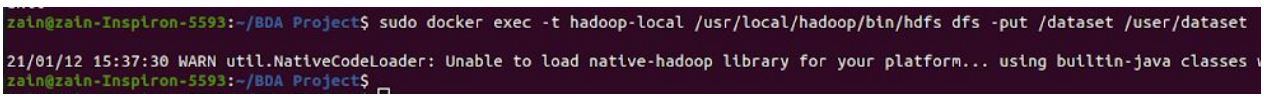


Check if the the local disk volume is mounted to container volume



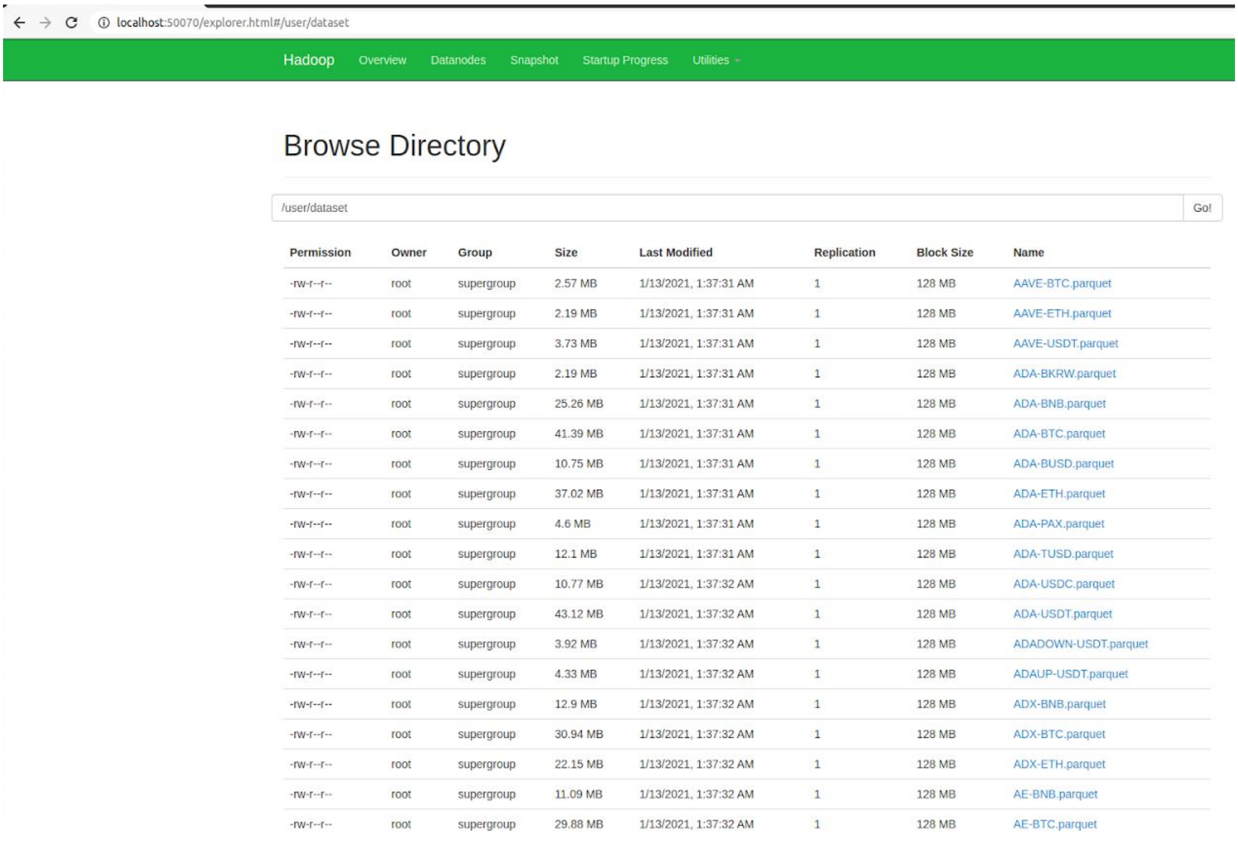
Now move data from hadoop container to hdfs. To do this, execute command from local linux terminal:

```
sudo docker exec -t hadoop-local /usr/local/hadoop/bin/hdfs dfs -put
/dataset /user/dataset
```



hadoop namenode port is 50070

Now go to `localhost:50070` and check if data is moved to hdfs or not



Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	root	supergroup	2.57 MB	1/13/2021, 1:37:31 AM	1	128 MB	AAVE-BTC.parquet
-rw-r--r--	root	supergroup	2.19 MB	1/13/2021, 1:37:31 AM	1	128 MB	AAVE-ETH.parquet
-rw-r--r--	root	supergroup	3.73 MB	1/13/2021, 1:37:31 AM	1	128 MB	AAVE-USDT.parquet
-rw-r--r--	root	supergroup	2.19 MB	1/13/2021, 1:37:31 AM	1	128 MB	ADA-BKRW.parquet
-rw-r--r--	root	supergroup	25.26 MB	1/13/2021, 1:37:31 AM	1	128 MB	ADA-BNB.parquet
-rw-r--r--	root	supergroup	41.39 MB	1/13/2021, 1:37:31 AM	1	128 MB	ADA-BTC.parquet
-rw-r--r--	root	supergroup	10.75 MB	1/13/2021, 1:37:31 AM	1	128 MB	ADA-BUSD.parquet
-rw-r--r--	root	supergroup	37.02 MB	1/13/2021, 1:37:31 AM	1	128 MB	ADA-ETH.parquet
-rw-r--r--	root	supergroup	4.6 MB	1/13/2021, 1:37:31 AM	1	128 MB	ADA-PAX.parquet
-rw-r--r--	root	supergroup	12.1 MB	1/13/2021, 1:37:31 AM	1	128 MB	ADA-TUSD.parquet
-rw-r--r--	root	supergroup	10.77 MB	1/13/2021, 1:37:32 AM	1	128 MB	ADA-USDC.parquet
-rw-r--r--	root	supergroup	43.12 MB	1/13/2021, 1:37:32 AM	1	128 MB	ADA-USDT.parquet
-rw-r--r--	root	supergroup	3.92 MB	1/13/2021, 1:37:32 AM	1	128 MB	ADADOWN-USDT.parquet
-rw-r--r--	root	supergroup	4.33 MB	1/13/2021, 1:37:32 AM	1	128 MB	ADAUP-USDT.parquet
-rw-r--r--	root	supergroup	12.9 MB	1/13/2021, 1:37:32 AM	1	128 MB	ADX-BNB.parquet
-rw-r--r--	root	supergroup	30.94 MB	1/13/2021, 1:37:32 AM	1	128 MB	ADX-BTC.parquet
-rw-r--r--	root	supergroup	22.15 MB	1/13/2021, 1:37:32 AM	1	128 MB	ADX-ETH.parquet
-rw-r--r--	root	supergroup	11.09 MB	1/13/2021, 1:37:32 AM	1	128 MB	AE-BNB.parquet
-rw-r--r--	root	supergroup	29.88 MB	1/13/2021, 1:37:32 AM	1	128 MB	AE-BTC.parquet

data is moved to hdfs

Now we will go to kafka container

Execute this command to start bash:

```
Sudo docker exec -it kafka /bin/sh
```

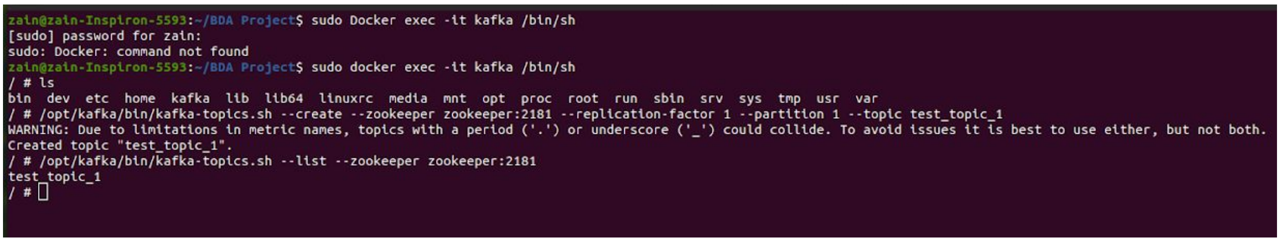
```
Cd opt/kafka
```

Create a topic in kafka with name **test_topic_1.** All the messages will be published to this topic

```
/opt/kafka/bin/kafka-topics.sh --create --zookeeper zookeeper:2181
--replication-factor 1 --partition 1 --topic test_topic_1
```

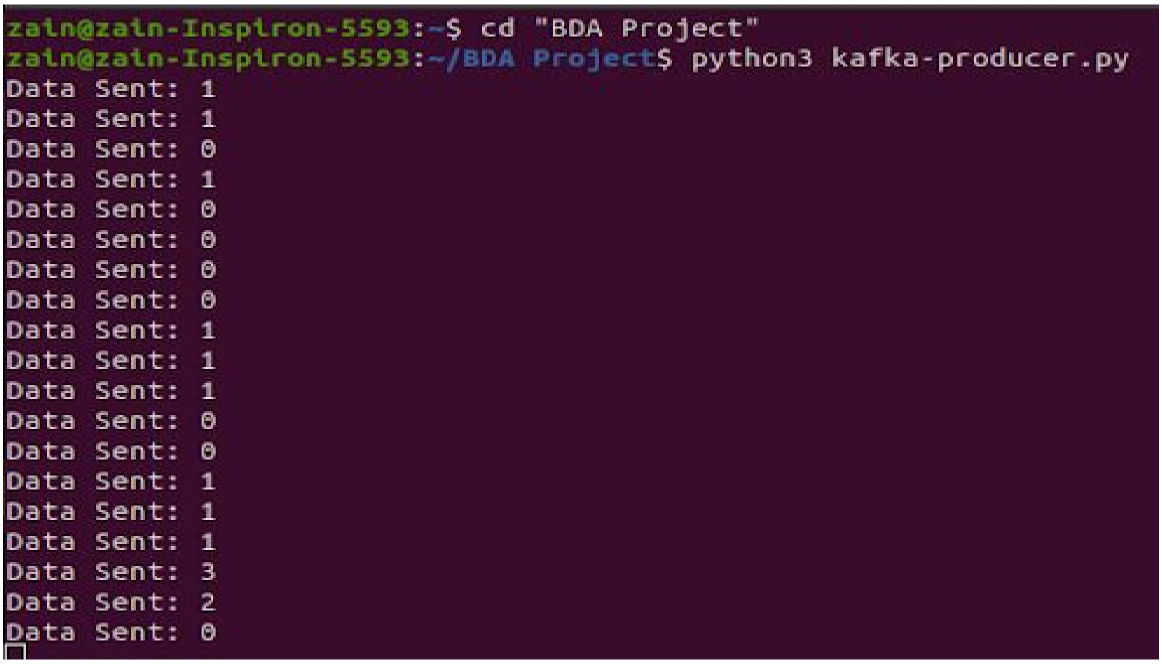
List all topics in kafka by executing this command in kafka container:

```
/opt/kafka/bin/kafka-topics.sh --list --zookeeper zookeeper:2181
```



Open new terminal and Run **kafka-producer.py**

Run command: `python3 kafka-producer.py`



Lets have a look at the code for **kafka-producer.py**

kafka-producer.py

```python
import pandas as pd
import pyarrow.parquet as pq
import glob
from time import sleep
from json import dumps
from json import loads
from kafka import KafkaProducer

# provide file path and store all files in one dataframe

path = r'./dataset' # use your path
all_files = glob.glob(path + "/*.parquet")

li = []

for fileobject in all_files:
    table = pq.read_table(fileobject)
    df = table.to_pandas()
    df = df.reset_index()
    filename = fileobject.split('/')[-1].split('.')[0]
    df['file_name'] = filename
    li.append(df)

frame = pd.concat(li, axis=0, ignore_index=True)

# initialize KafkaProducer
producer = KafkaProducer(bootstrap_servers=['localhost:9092'],
        value_serializer=lambda x: dumps(x).encode('utf-8'))


# send one row from data after every 1 second
for index, row in frame.head(100000).iterrows():
    print('Data Sent: {}'.format(row['number_of_trades']))
    producer.send('test_topic_1',value=row['number_of_trades'])
    sleep(1)
```

Open new terminal and Run **kafka-consumer.py**

Run command: `python3 kafka-consumer.py`

```
zain@zain-Inspiron-5593:~/BDA Project$ python3 kafka-consumer.py
Data Received: 1
Data Received: 1
Data Received: 0
Data Received: 1
Data Received: 0
Data Received: 0
Data Received: 0
Data Received: 0
Data Received: 1
Data Received: 1
Data Received: 1
Data Received: 0
Data Received: 0
Data Received: 1
Data Received: 1
Data Received: 1
Data Received: 3
Data Received: 2
Data Received: 0
```

Lets have a look at the code for **kafka-consumer.py**

kafka-producer.py | kafka-consumer.py

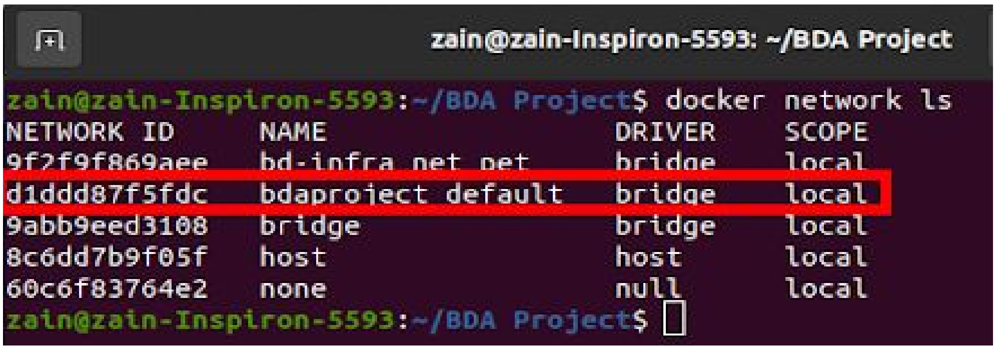
```python


from kafka import KafkaConsumer
from json import loads



if __name__ == '__main__':

	# initialize KakfaConsumer
	consumer = KafkaConsumer('test_topic_1', bootstrap_servers=['localhost:9092'],
		auto_offset_reset='earliest', enable_auto_commit=True,
		value_deserializer=lambda x: loads(x.decode('utf-8')))


	# extract message from consumer
	for message in consumer:
		msg = message.value
		# collection.insert_one(message)
		print('Data Received: {}'.format(msg))


```

All of these containers are under a bridge network

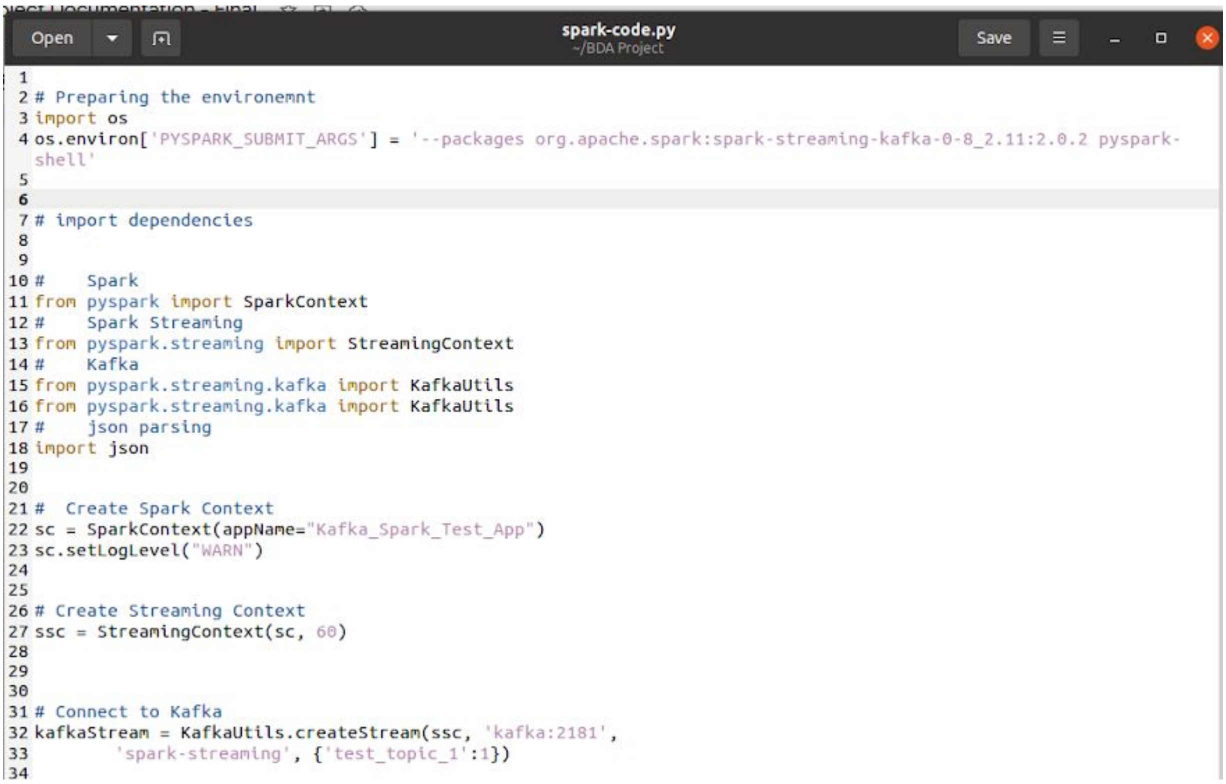
Check name of our bridge network by using command:

```
Docker network ls
```

Docker created **bdaproject_default** bridge network by default



Run **spark-job.py**



Use Case 2

For this scenario we have intended to perform the following steps:

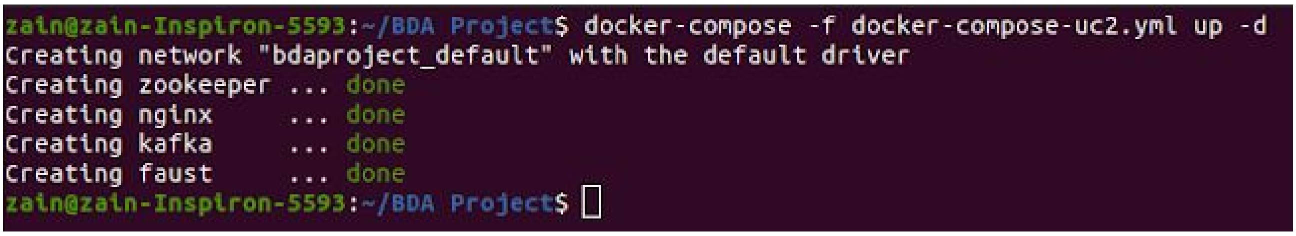
1. Use empty container to store dataset from local system to container
2. Start zookeeper and kafka server
3. Start producer and consumer for kafka
4. Use container where jupyter notebook is installed with faust library (faust works as an in-python streaming platform, similar to spark streaming)
5. Aggregate values and show updated sum of values to nginx server after specific interval

Steps

Open directory “BDA Project” in linux terminal

Execute ‘docker-compose-uc1.yml’ by executing this command:

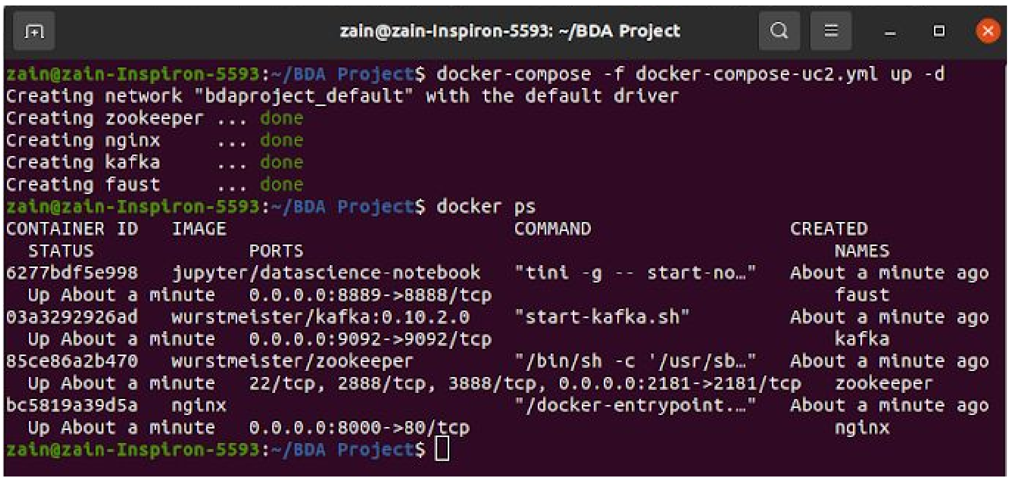
`docker-compose -f docker-compose-uc2.yml up -d`



```
zain@zain-Inspiron-5593:~/BDA Project$ docker-compose -f docker-compose-uc2.yml up -d
Creating network "bdaproject_default" with the default driver
Creating zookeeper ... done
Creating nginx     ... done
Creating kafka     ... done
Creating faust     ... done
zain@zain-Inspiron-5593:~/BDA Project$
```

Docker container status:

`Docker ps`



```
zain@zain-Inspiron-5593: ~/BDA Project
zain@zain-Inspiron-5593:~/BDA Project$ docker-compose -f docker-compose-uc2.yml up -d
Creating network "bdaproject_default" with the default driver
Creating zookeeper ... done
Creating nginx     ... done
Creating kafka     ... done
Creating faust     ... done
zain@zain-Inspiron-5593:~/BDA Project$ docker ps
CONTAINER ID   IMAGE                              COMMAND                  CREATED
  STATUS              PORTS                                              NAMES
6277bdf5e998   jupyter/datascience-notebook       "tini -g -- start-no…"   About a minute ago
  Up About a minute   0.0.0.0:8889->8888/tcp                             faust
03a3292926ad   wurstmeister/kafka:0.10.2.0        "start-kafka.sh"         About a minute ago
  Up About a minute   0.0.0.0:9092->9092/tcp                             kafka
85ce86a2b470   wurstmeister/zookeeper             "/bin/sh -c '/usr/sb…"   About a minute ago
  Up About a minute   22/tcp, 2888/tcp, 3888/tcp, 0.0.0.0:2181->2181/tcp   zookeeper
bc5819a39d5a   nginx                              "/docker-entrypoint.…"   About a minute ago
  Up About a minute   0.0.0.0:8000->80/tcp                               nginx
zain@zain-Inspiron-5593:~/BDA Project$
```

Now we will move to kafka container

Execute this command to start bash:

```
Sudo docker exec -it kafka /bin/sh
```

```
Cd opt/kafka
```

Create a topic in kafka with name **test_topic_1.** All the messages will be published to this topic

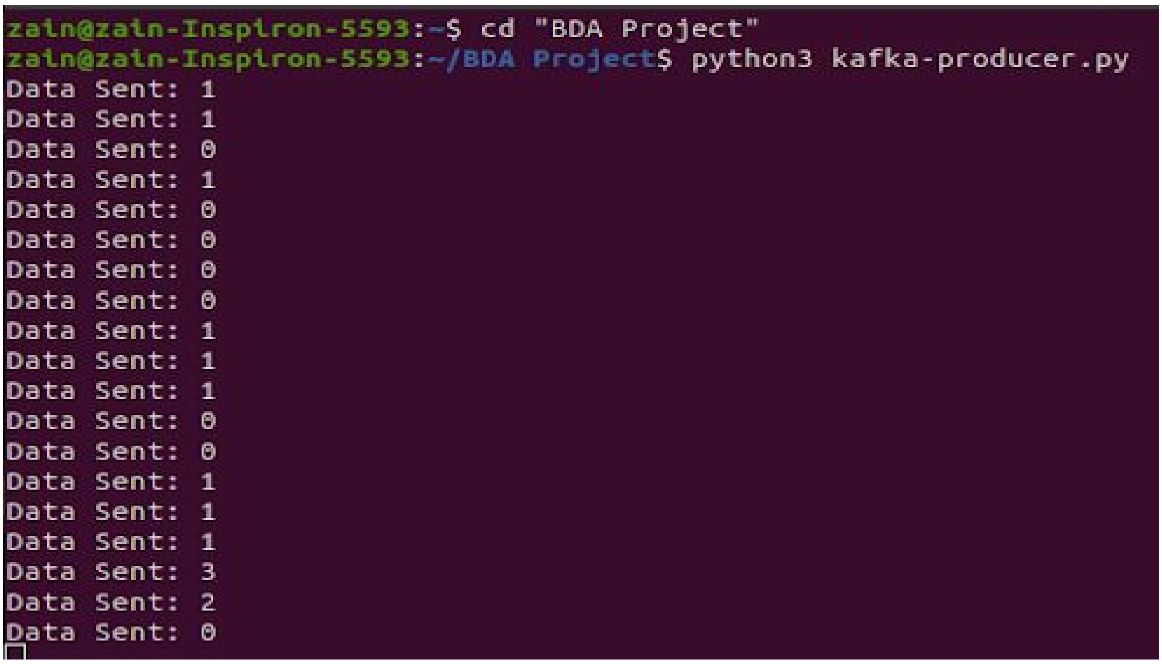
```
/opt/kafka/bin/kafka-topics.sh --create --zookeeper zookeeper:2181
--replication-factor 1 --partition 1 --topic test_topic_1
```

List all topics in kafka by executing this command in kafka container:

```
/opt/kafka/bin/kafka-topics.sh --list --zookeeper zookeeper:2181
```

Open new terminal and Run **kafka-producer.py**

Run command: `python3 kafka-producer.py`



Lets have a look at the code for **kafka-producer.py**

kafka-producer.py

```python
import pandas as pd
import pyarrow.parquet as pq
import glob
from time import sleep
from json import dumps
from json import loads
from kafka import KafkaProducer

# provide file path and store all files in one dataframe

path = r'./dataset' # use your path
all_files = glob.glob(path + "/*.parquet")

li = []

for fileobject in all_files:
  table = pq.read_table(fileobject)
  df = table.to_pandas()
  df = df.reset_index()
  filename = fileobject.split('/')[-1].split('.')[0]
  df['file_name'] = filename
  li.append(df)

frame = pd.concat(li, axis=0, ignore_index=True)

# initialize KafkaProducer
producer = KafkaProducer(bootstrap_servers=['localhost:9092'],
        value_serializer=lambda x: dumps(x).encode('utf-8'))


# send one row from data after every 1 second
for index, row in frame.head(100000).iterrows():
  print('Data Sent: {}'.format(row['number_of_trades']))
  producer.send('test_topic_1',value=row['number_of_trades'])
  sleep(1)
```

Open new terminal and Run **kafka-consumer.py**

Run command: `python3 kafka-consumer.py`

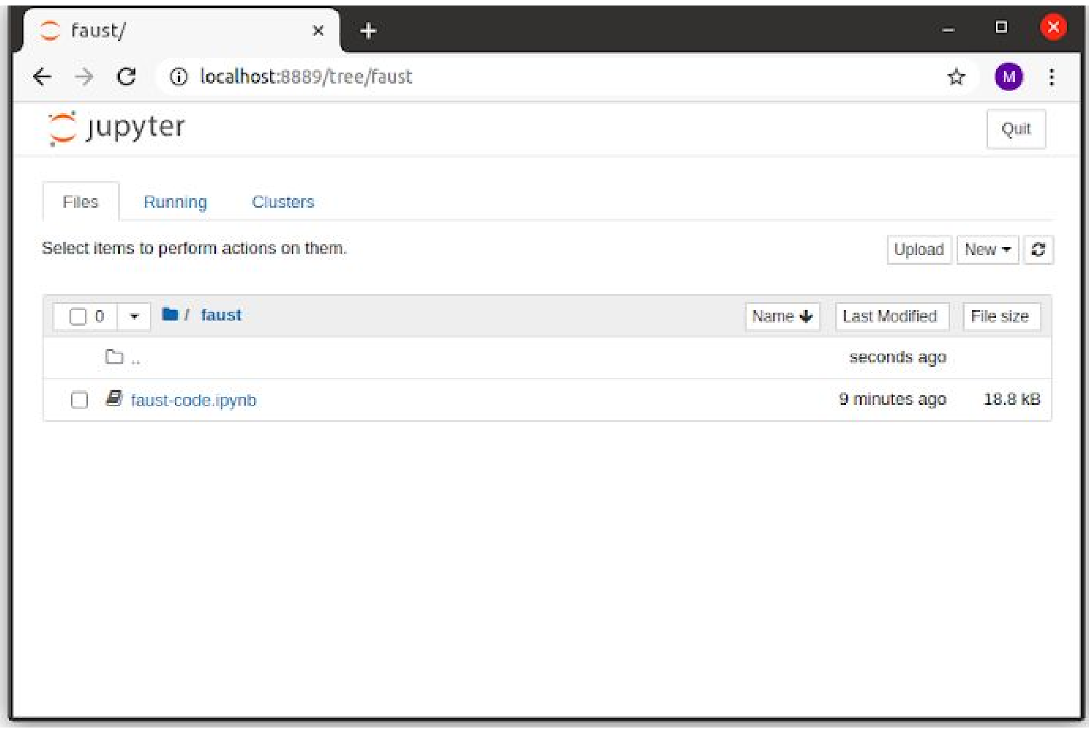
```
zain@zain-Inspiron-5593:~/BDA Project$ python3 kafka-consumer.py
Data Received:  1
Data Received:  1
Data Received:  0
Data Received:  1
Data Received:  0
Data Received:  0
Data Received:  0
Data Received:  0
Data Received:  1
Data Received:  1
Data Received:  1
Data Received:  0
Data Received:  0
Data Received:  1
Data Received:  1
Data Received:  1
Data Received:  3
Data Received:  2
Data Received:  0
```

Lets have a look at the code for **kafka-consumer.py**

```
kafka-producer.py        kafka-consumer.py


from kafka import KafkaConsumer
from json import loads



if __name__ == '__main__':

        # initialize KakfaConsumer
        consumer = KafkaConsumer('test_topic_1', bootstrap_servers=['localhost:9092'],
                auto_offset_reset='earliest', enable_auto_commit=True,
                value_deserializer=lambda x: loads(x.decode('utf-8')))


        # extract message from consumer
        for message in consumer:
                msg = message.value
                # collection.insert_one(message)
                print('Data Received: {}'.format(msg))


```

We have created a container with a python notebook which has a faust library.

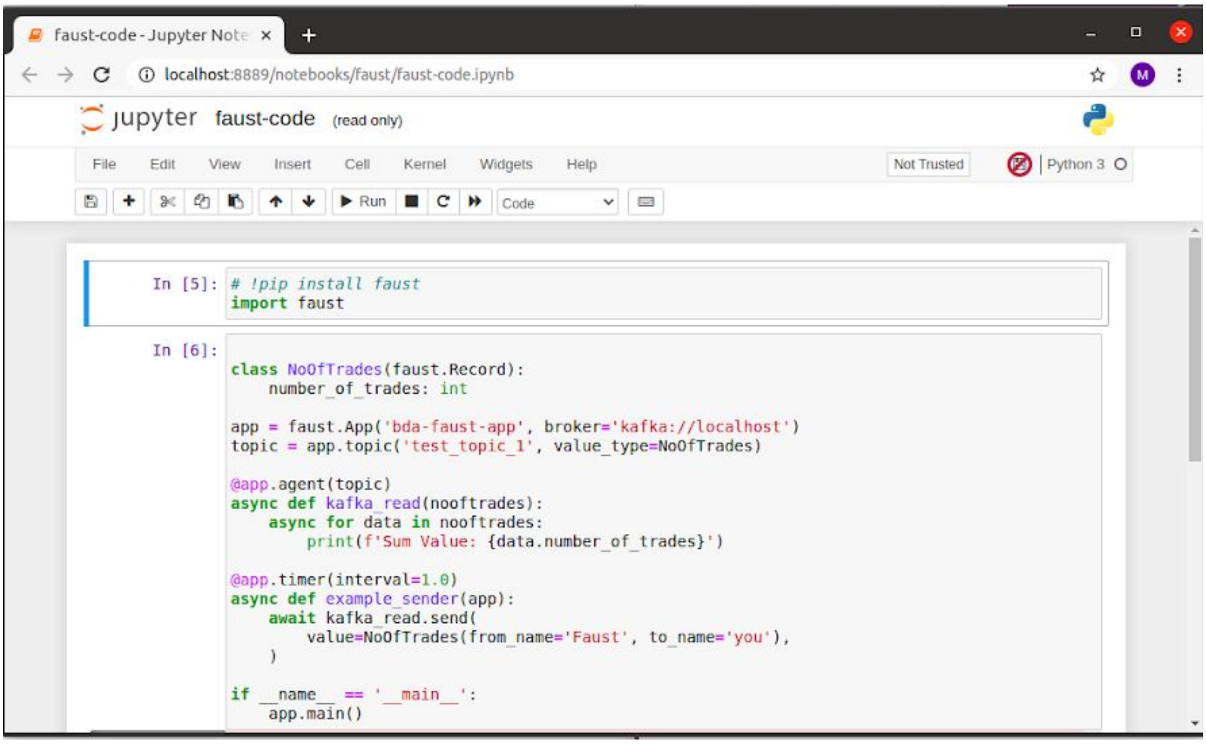
Faust library works as an alternative to apache spark. It serves similar feature to apache spark / storm / flink / fume i.e it provides streaming platform within python environment



We have mapped 8888 port coming from container to 8889 port of host system

We can see that our local system volume is bind to container volume, therefore, we can see our code placed inside of container

Let's have a look at **faust.py**



In [5]:

```python
# !pip install faust
import faust
```

In [6]:

```python
class NoOfTrades(faust.Record):
    number_of_trades: int

app = faust.App('bda-faust-app', broker='kafka://localhost')
topic = app.topic('test_topic_1', value_type=NoOfTrades)

@app.agent(topic)
async def kafka_read(nooftrades):
    async for data in nooftrades:
        print(f'Sum Value: {data.number_of_trades}')

@app.timer(interval=1.0)
async def example_sender(app):
    await kafka_read.send(
        value=NoOfTrades(from_name='Faust', to_name='you'),
    )

if __name__ == '__main__':
    app.main()
```

Faust will consume message coming from kafka consumer

Key Challenges

Some of the key challenges faced during this project are mentioned below:

1. Implementation of Zeta Architecture in a multi container environment is a highly challenging task. Complexity of maintaining and integrating each container as a bundled application makes it difficult to handle.
2. Handling of different Libraries at each level was another challenge which involved a lot of time exploring to issue resolution
3. Scope of the project within the stipulated time frame was very tough but on a brighter side it was full of learning
4. Uncertainty on doing things on dockerized platforms.
5. Some hardware limitations were a hindrance for us in this project. Low disk space restricted us to explore more docker images
6. This project had a dependency of using ubuntu over a dual boot environment. This took quite some time to set up as it was not readily available