Publisher Subscriber System using Kafka

University at Buffalo
Department of Computer Science and Engineering
CSE 586 - Distributed System
Project 2

Report by:
Abhishek Kumar
50419133
akumar58@buffalo.edu

Utkarsh Kumar
50419745
utkarshk@buffalo.edu

(Group -99)

Abstract

The Publish/Subscribe pattern, also known as pub/sub, is an architectural design pattern that provides a framework for exchanging messages between publishers and subscribers. This pattern involves the publisher and the subscriber relying on a message broker that relays messages from the publisher to the subscribers. The host (publisher) publishes messages (events) to a channel that subscribers can then sign up to.

Although Pub/Sub is based on earlier design patterns like message queuing and event brokers, it is more flexible and scalable. The key to this is the fact Pub/Sub enables the movement of messages between different components of the system without the components being aware of each other's identity.

The Pub/Sub pattern evolved out of the necessity to expand the scale of information systems. In the pre-Internet era, and even during the early days of the Internet, the systems were mostly scaled statically. However, with the expansion of the Internet and web-based applications, fueled by the massive adoption of mobile and IoT devices, systems needed to scale dynamically.

1. Introduction

1.1 Client Server model

Client–server model is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients.Often clients and servers communicate over a computer network on separate hardware, but both client and server may reside in the same system. The communication takes place using sockets using TCP protocol.

A server host runs one or more server programs, which share their resources with clients. A client usually does not share any of its resources, but it requests content or service from a server. Clients, therefore, initiate communication sessions with servers, which await incoming requests. Examples of computer applications that use the client–server model are email, network printing, and the World Wide Web.

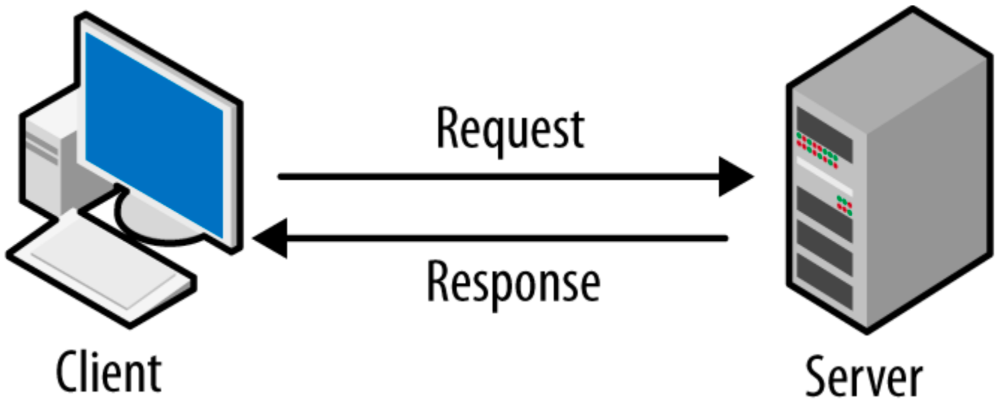


Fig 1: Client-Server model

1.2 Publisher Subscriber model

Pub/Sub is shorthand for publisher/Subscriber messaging, an asynchronous communication method in which messages are exchanged between applications without knowing the identity of the sender or recipient.

Four core concepts make up the pub/sub model:

1. Topic – An intermediary channel that maintains a list of subscribers to relay messages to that are received from publishers
2. Message – Serialized messages sent to a topic by a publisher which has no knowledge of the subscribers

3. Publisher – The application that publishes a message to a topic
4. Subscriber – An application that registers itself with the desired topic in order to receive the appropriate messages

Advantages and disadvantages of pub/sub

As with all technology, using pub/sub messaging comes with advantages and disadvantages. The two primary advantages are loose coupling and scalability.

1. Loose coupling

   Publishers are never aware of the existence of subscribers so that both systems can operate independently of each other. This methodology removes service dependencies that are present in traditional coupling. For example, a client generally cannot send a message to a server if the server process is not running. With pub/sub, the client is no longer concerned whether or not processes are running on the server.

2. Scalability

   Pub/sub messaging can scale to volumes beyond the capability of a single traditional data center. This level of scalability is primarily due to parallel operations, message caching, tree-based routing, and multiple other features built into the pub/sub model.

   Scalability does have a limit though. Increasing the number of nodes and messages also increases the chances of experiencing a load surge or slowdown. On top of that, the advantages of the pub/sub model can sometimes be overshadowed by the message delivery issues it experiences, such as:

   - A publisher may only deliver messages for a certain period of time regardless of whether the message was received or not.
   - Since the publisher does not have a window into the subscriber it will always assume that the appropriate subscriber is listening. If the subscriber isn't listening and misses an important message it can be disastrous for production systems.

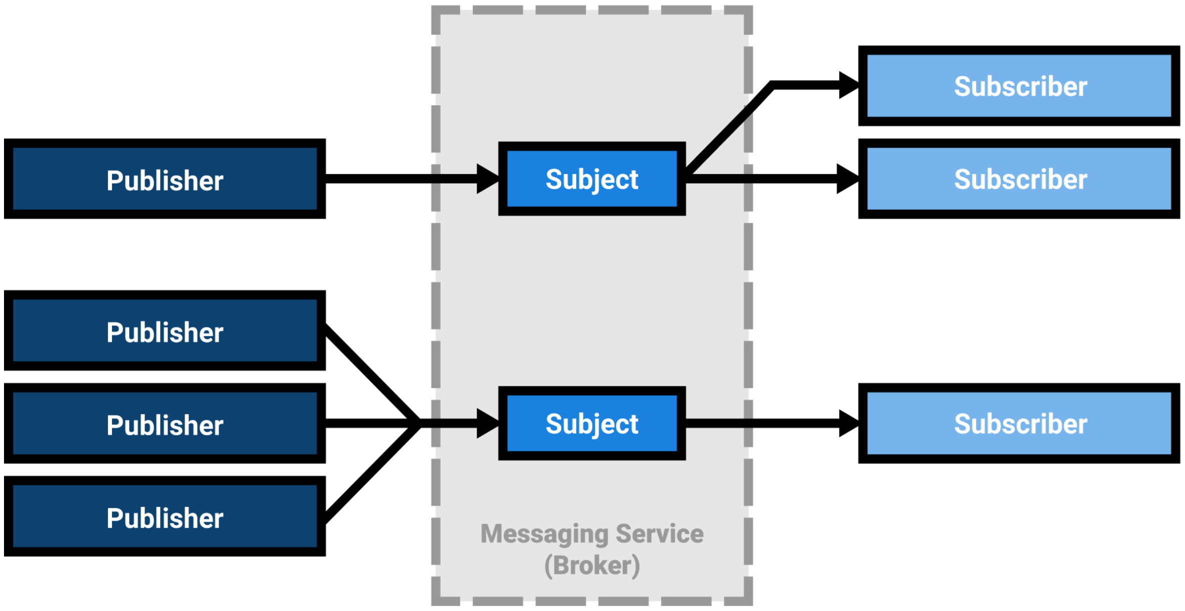


Fig 2: Publisher Subscriber System

1.3 Docker

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly.

With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

Docker Image - It is a file, composed of multiple layers, used to execute code in a docker container. They are a set of instructions used to create docker containers.

Docker containers - It is a runtime instance of an image. Allows developers to package applications with all parts needed such as libraries and other dependencies.

Docker file - It is a text document that contains necessary commands which on execution helps assemble a Docker Image. Docker image is created using a Docker file.

Docker engine - The software that hosts the containers is named Docker Engine. Docker Engine is a client-server based application.

Docker compose - Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.

1.4 Kafka

Apache Kafka is a stream-processing software platform originally developed by LinkedIn, open sourced in early 2011 and currently developed by the Apache Software Foundation. It is written in Scala and Java.

Key Concepts of Kafka

- Kafka is a distributed system that consists of servers and clients.
- Some servers are called brokers and they form the storage layer. Other servers run Kafka Connect to import and export data as event streams to integrate Kafka with your existing system continuously.
- On the other hand, clients allow you to create applications that read, write and process streams of events. A client could be a producer or a consumer.
- A producer writes (produces) events to Kafka while a consumer reads and processes (consumes) events from Kafka.
- Servers and clients communicate via a high-performance TCP network protocol and are fully decoupled and agnostic of each other.
- In Kafka, an event is an object that has a key, a value and a timestamp. Optionally, it could have other metadata headers. One or more events are organized in topics: producers can write messages/events on different topics and consumers can choose to read and process events of one or more topics.
- In Kafka, you can configure how long events of a topic should be retained, therefore, they can be read whenever needed and are not deleted after consumption.
- A consumer consumes the stream of events of a topic at its own pace and can commit its position (called offset). When we commit the offset we set a pointer to the last record that the consumer has consumed.
- From the servers side, topics are partitioned and replicated.
- A topic is partitioned for scalability reasons. Its events are spread over different Kafka brokers. This allows clients to read/write from/to many brokers at the same time.
- For availability and fault-tolerance every topic can also be replicated. It means that multiple brokers in different datacenters could have a copy of the same data.

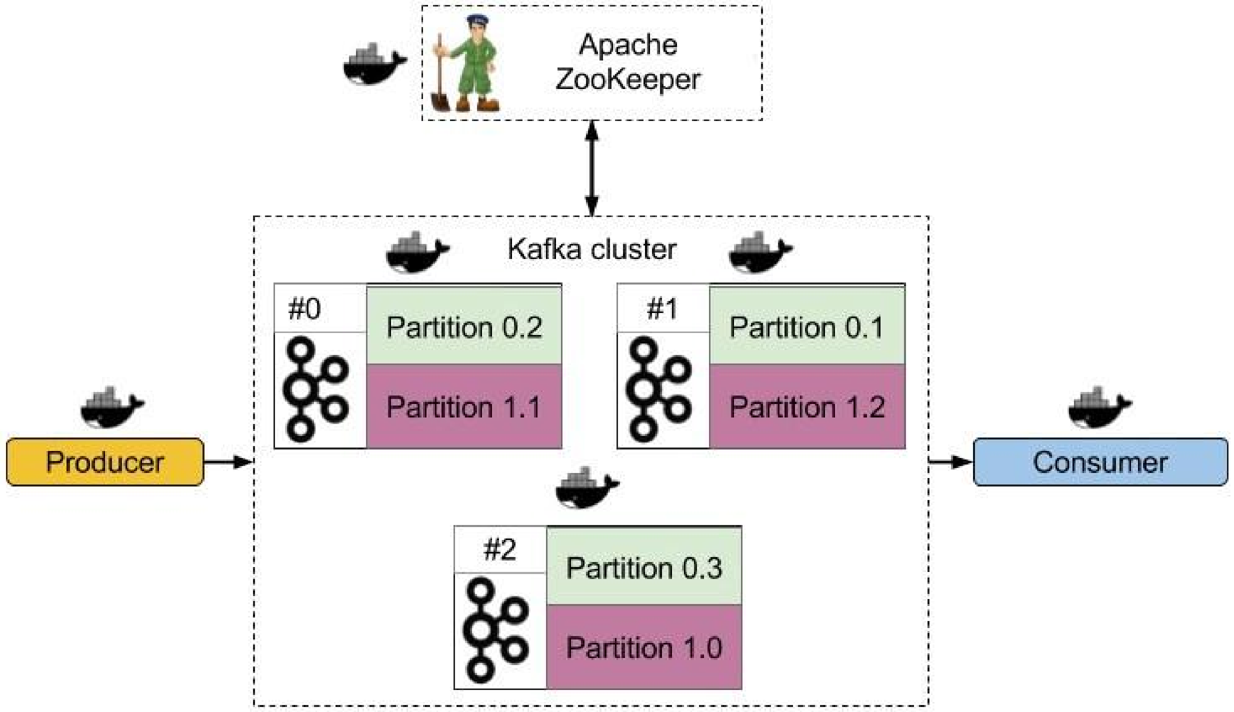


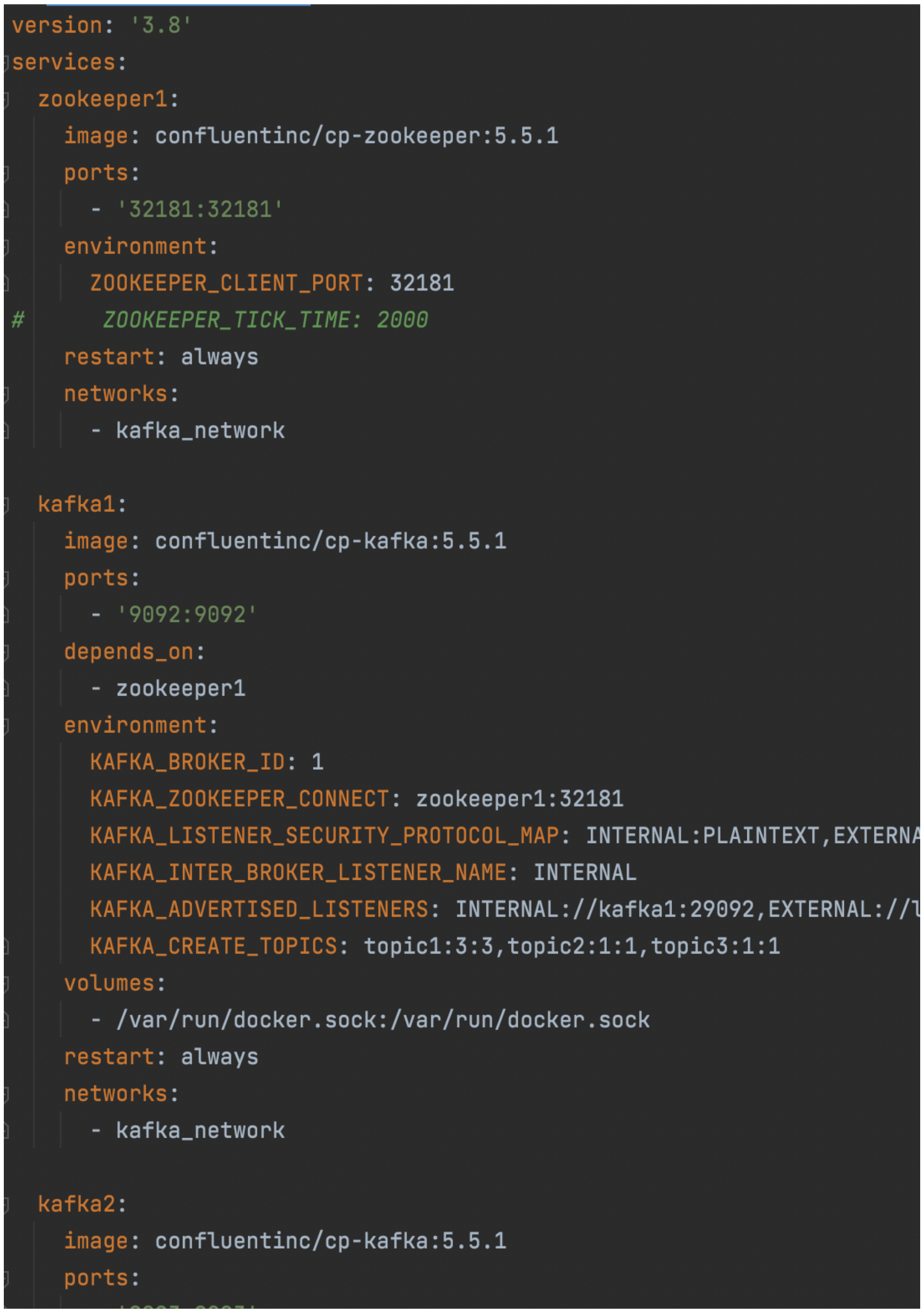
Fig: Kafka/Zookeeper in Docker

2. Implementation

2.1 docker-compose.yml configuration

- To start an Apache Kafka server, we'd first need to start a Zookeeper server.
- We can configure this dependency in a docker-compose.yml file, which will ensure that the Zookeeper server always starts before the Kafka server and stops after it.
- In this setup, our Zookeeper server is listening on port=32181 for the kafka service, which is defined within the same container setup. However, for any client running on the host, it'll be exposed on port **32181**.
- Similarly, the kafka service is exposed to the host applications through port 29092, but it is actually advertised on port 9092 within the container environment configured by the KAFKA_ADVERTISED_LISTENERS property.
- We define 3 kafka containers with the below containers for 3 different ports
- KAFKA PORTS : **9092,9093,9094**
- Each service must **expose a unique port to the host machine.**
- Our Flask application is running on port **5001**

- All the containers are connected via network bridge “kafka_network” , name : my_network.
- The topics are spread across each kafka node
- The partition and replication is defined in the docker-compose.yml file itself, so that kafka creates topics while node initialization.

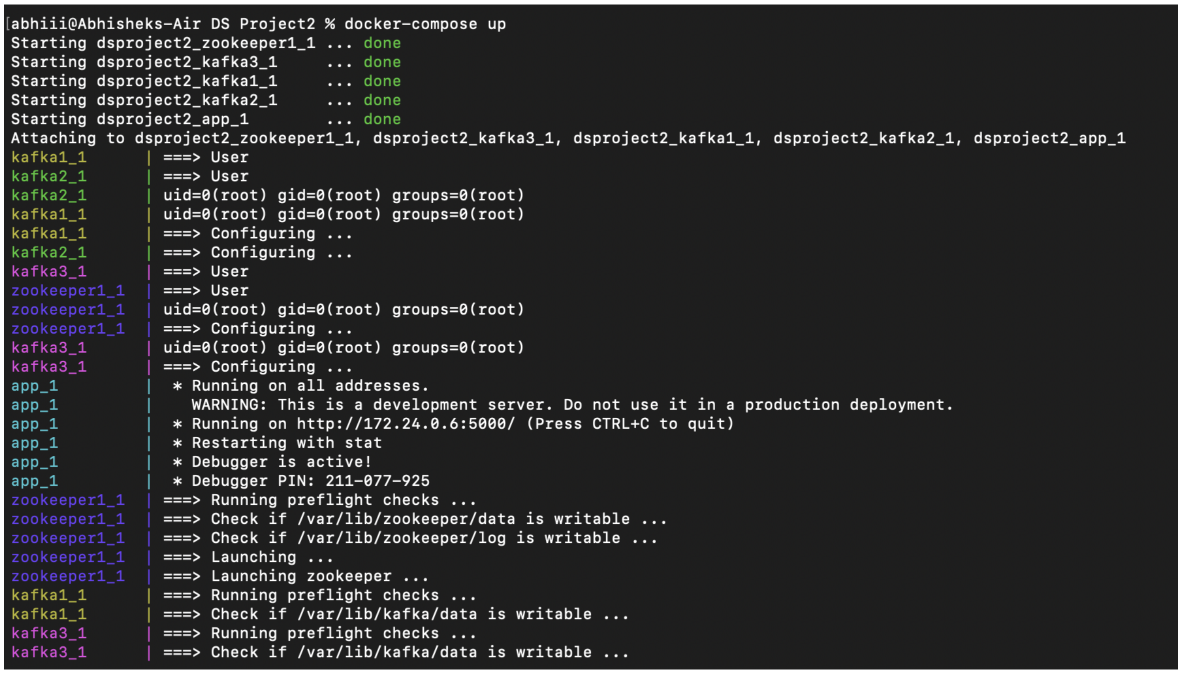


```yaml
version: '3.8'
services:
  zookeeper1:
    image: confluentinc/cp-zookeeper:5.5.1
    ports:
      - '32181:32181'
    environment:
      ZOOKEEPER_CLIENT_PORT: 32181
#      ZOOKEEPER_TICK_TIME: 2000
    restart: always
    networks:
      - kafka_network

  kafka1:
    image: confluentinc/cp-kafka:5.5.1
    ports:
      - '9092:9092'
    depends_on:
      - zookeeper1
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper1:32181
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: INTERNAL:PLAINTEXT,EXTERNA
      KAFKA_INTER_BROKER_LISTENER_NAME: INTERNAL
      KAFKA_ADVERTISED_LISTENERS: INTERNAL://kafka1:29092,EXTERNAL://l
      KAFKA_CREATE_TOPICS: topic1:3:3,topic2:1:1,topic3:1:1
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    restart: always
    networks:
      - kafka_network

  kafka2:
    image: confluentinc/cp-kafka:5.5.1
    ports:
```

Fig: docker-compose.yml configuration

2.2 Starting the cluster

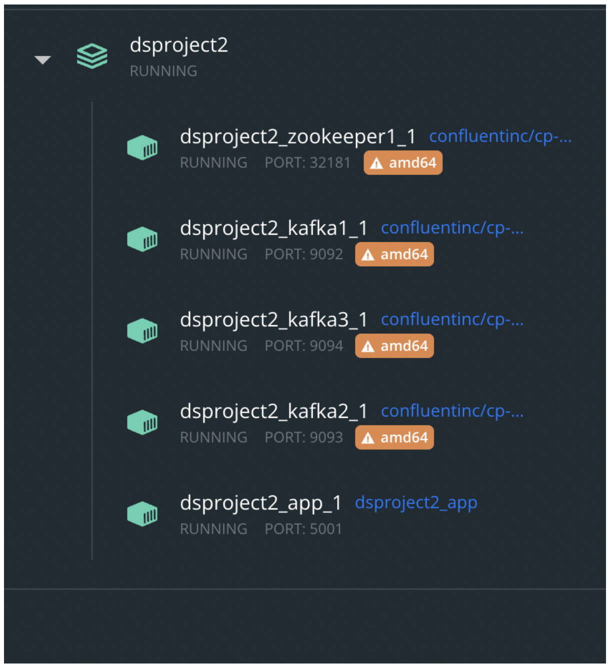
After our docker cluster setup we will run our setup using **docker-compose up** in terminal.



```
[abhiii@Abhisheks-Air DS Project2 % docker-compose up
 Starting dsproject2_zookeeper1_1 ... done
 Starting dsproject2_kafka3_1     ... done
 Starting dsproject2_kafka1_1     ... done
 Starting dsproject2_kafka2_1     ... done
 Starting dsproject2_app_1        ... done
 Attaching to dsproject2_zookeeper1_1, dsproject2_kafka3_1, dsproject2_kafka1_1, dsproject2_kafka2_1, dsproject2_app_1
 kafka1_1      | ===> User
 kafka2_1      | ===> User
 kafka2_1      | uid=0(root) gid=0(root) groups=0(root)
 kafka1_1      | uid=0(root) gid=0(root) groups=0(root)
 kafka1_1      | ===> Configuring ...
 kafka2_1      | ===> Configuring ...
 kafka3_1      | ===> User
 zookeeper1_1  | ===> User
 zookeeper1_1  | uid=0(root) gid=0(root) groups=0(root)
 zookeeper1_1  | ===> Configuring ...
 kafka3_1      | uid=0(root) gid=0(root) groups=0(root)
 kafka3_1      | ===> Configuring ...
 app_1         |  * Running on all addresses.
 app_1         |    WARNING: This is a development server. Do not use it in a production deployment.
 app_1         |  * Running on http://172.24.0.6:5000/ (Press CTRL+C to quit)
 app_1         |  * Restarting with stat
 app_1         |  * Debugger is active!
 app_1         |  * Debugger PIN: 211-077-925
 zookeeper1_1  | ===> Running preflight checks ...
 zookeeper1_1  | ===> Check if /var/lib/zookeeper/data is writable ...
 zookeeper1_1  | ===> Check if /var/lib/zookeeper/log is writable ...
 zookeeper1_1  | ===> Launching ...
 zookeeper1_1  | ===> Launching zookeeper ...
 kafka1_1      | ===> Running preflight checks ...
 kafka1_1      | ===> Check if /var/lib/kafka/data is writable ...
 kafka3_1      | ===> Running preflight checks ...
 kafka3_1      | ===> Check if /var/lib/kafka/data is writable ...
```

Fig : docker-compose up in terminal

- We can view all the running containers in our docker desktop application.

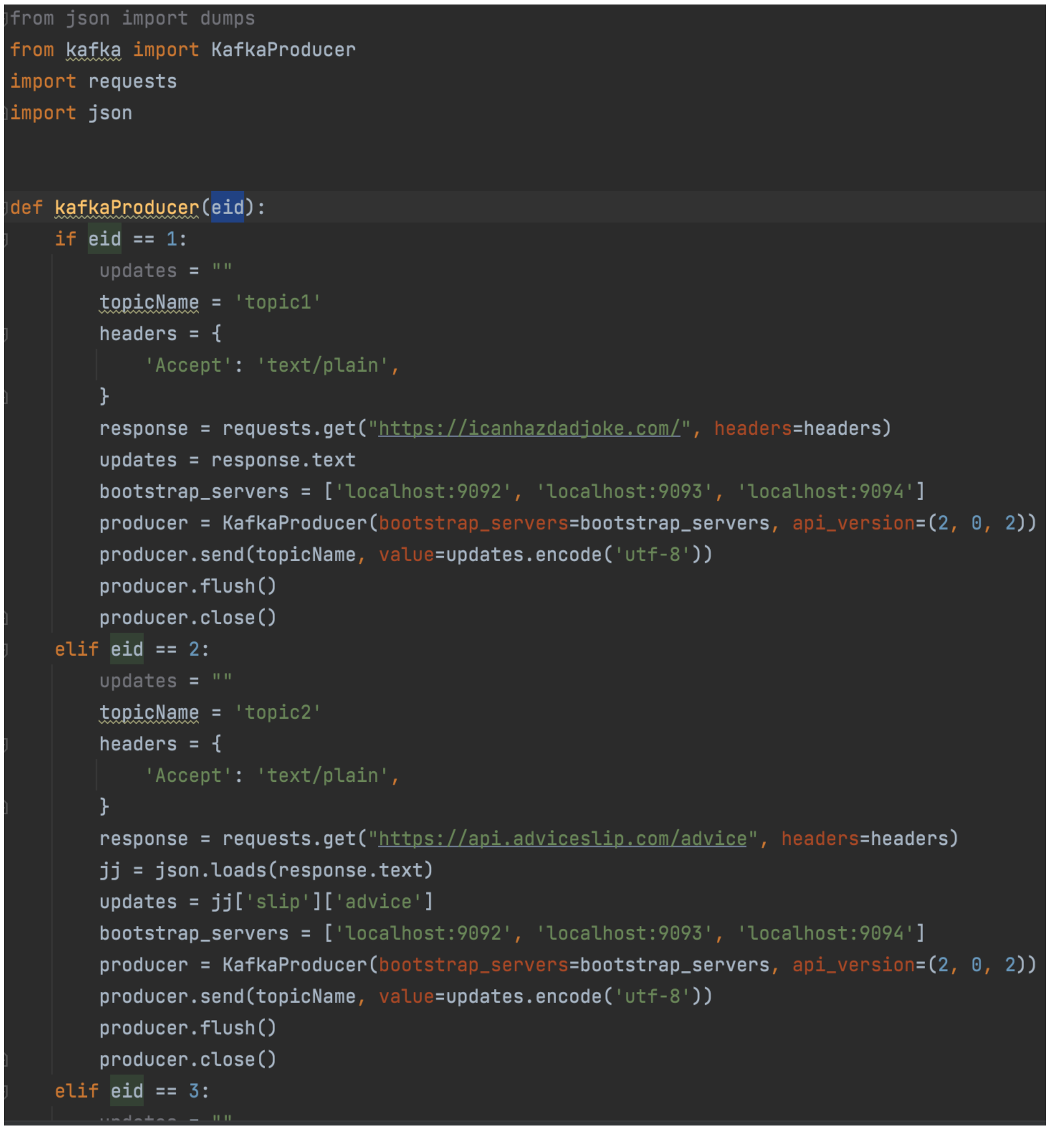


```
dsproject2
RUNNING

dsproject2_zookeeper1_1  confluentinc/cp-...
RUNNING  PORT: 32181  amd64

dsproject2_kafka1_1  confluentinc/cp-...
RUNNING  PORT: 9092  amd64

dsproject2_kafka3_1  confluentinc/cp-...
RUNNING  PORT: 9094  amd64

dsproject2_kafka2_1  confluentinc/cp-...
RUNNING  PORT: 9093  amd64

dsproject2_app_1  dsproject2_app
RUNNING  PORT: 5001
```

Fig : Running containers in docker desktop

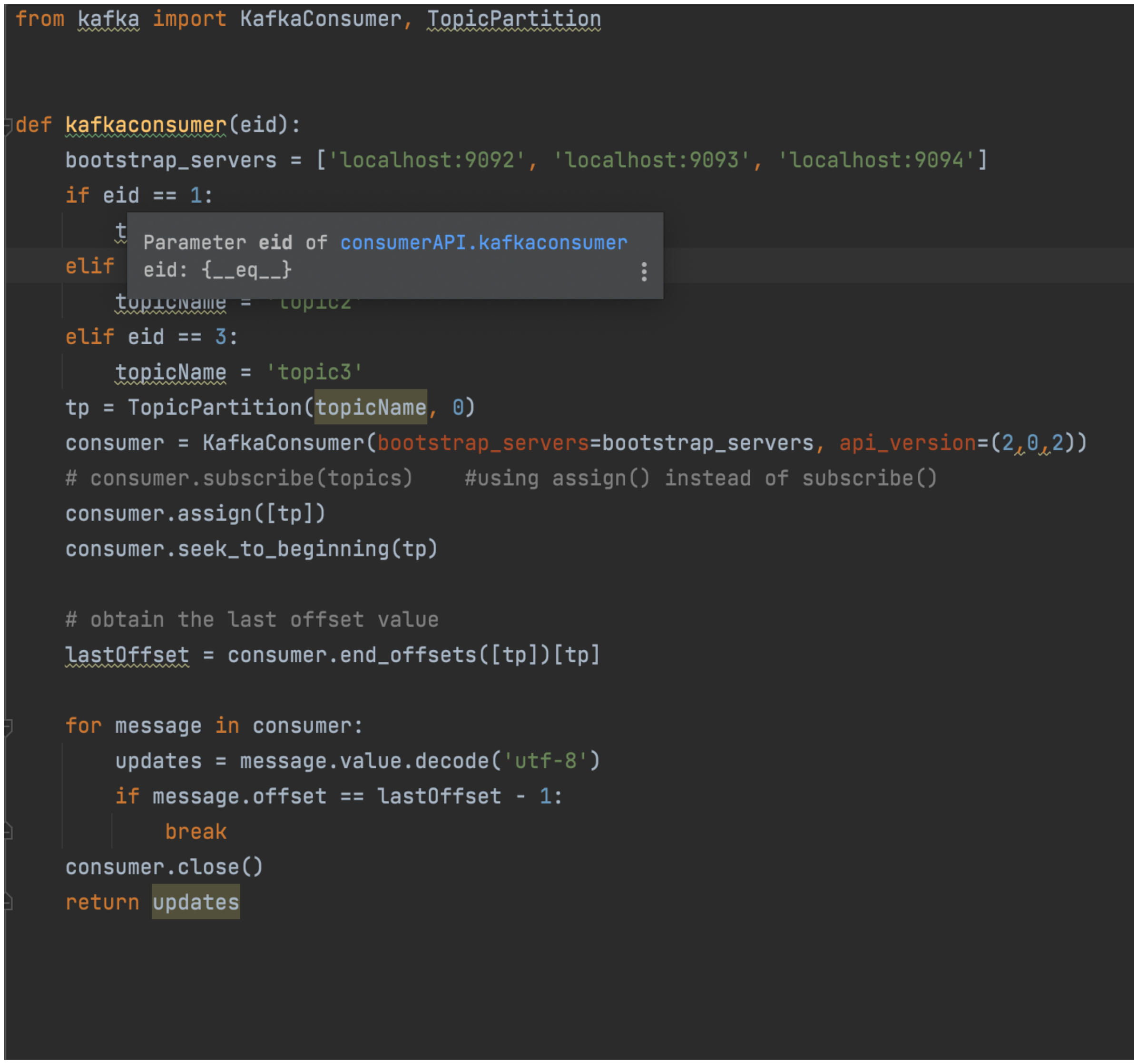
2.3 Producer and Consumer in python application

- Next we define our Producer and consumer using kafka APIs
- We use **kafka-python=2.0.2** for our project
- We create producer and consumer objects for publishing new topics and subscribing and view the published data into the topics
- We subscribe user using **assign()** function of consumerAPI
- The **producerAPI.py** creates producer objects and sends data to the kafka cluster depending on the bootstrap_servers.
- The **consumerAPI.py** file is used to create consumer objects for subscribers to subscribe to a topic and receive data from kafka clusters
- All the topics are already defined in the **docker-compose.yml**
- Topic partition and replication is also defined in docker-compose.yml**.**



```python
from json import dumps
from kafka import KafkaProducer
import requests
import json


def kafkaProducer(eid):
    if eid == 1:
        updates = ""
        topicName = 'topic1'
        headers = {
            'Accept': 'text/plain',
        }
        response = requests.get("https://icanhazdadjoke.com/", headers=headers)
        updates = response.text
        bootstrap_servers = ['localhost:9092', 'localhost:9093', 'localhost:9094']
        producer = KafkaProducer(bootstrap_servers=bootstrap_servers, api_version=(2, 0, 2))
        producer.send(topicName, value=updates.encode('utf-8'))
        producer.flush()
        producer.close()
    elif eid == 2:
        updates = ""
        topicName = 'topic2'
        headers = {
            'Accept': 'text/plain',
        }
        response = requests.get("https://api.adviceslip.com/advice", headers=headers)
        jj = json.loads(response.text)
        updates = jj['slip']['advice']
        bootstrap_servers = ['localhost:9092', 'localhost:9093', 'localhost:9094']
        producer = KafkaProducer(bootstrap_servers=bootstrap_servers, api_version=(2, 0, 2))
        producer.send(topicName, value=updates.encode('utf-8'))
        producer.flush()
        producer.close()
    elif eid == 3:
```

Fig: produceAPI.py



```
from kafka import KafkaConsumer, TopicPartition


def kafkaconsumer(eid):
    bootstrap_servers = ['localhost:9092', 'localhost:9093', 'localhost:9094']
    if eid == 1:
        t
    elif
        topicName = 'topic2'
    elif eid == 3:
        topicName = 'topic3'
    tp = TopicPartition(topicName, 0)
    consumer = KafkaConsumer(bootstrap_servers=bootstrap_servers, api_version=(2,0,2))
    # consumer.subscribe(topics)    #using assign() instead of subscribe()
    consumer.assign([tp])
    consumer.seek_to_beginning(tp)

    # obtain the last offset value
    lastOffset = consumer.end_offsets([tp])[tp]

    for message in consumer:
        updates = message.value.decode('utf-8')
        if message.offset == lastOffset - 1:
            break
    consumer.close()
    return updates
```

```
Parameter eid of consumerAPI.kafkaconsumer
eid: {__eq__}
```

Fig : consumerAPI.py

3. UI Details

The application is created using Flask, python, html and kafka/zookeeper.

- The user logins as a Publisher or Subscriber using a security key. The security key is different for Publishers and Subscriber.
- The application displays “Wrong Password” if the correct password is not entered.

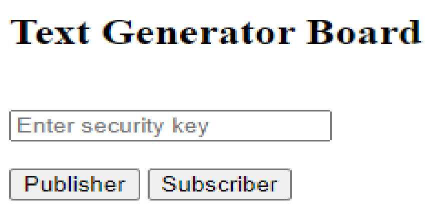
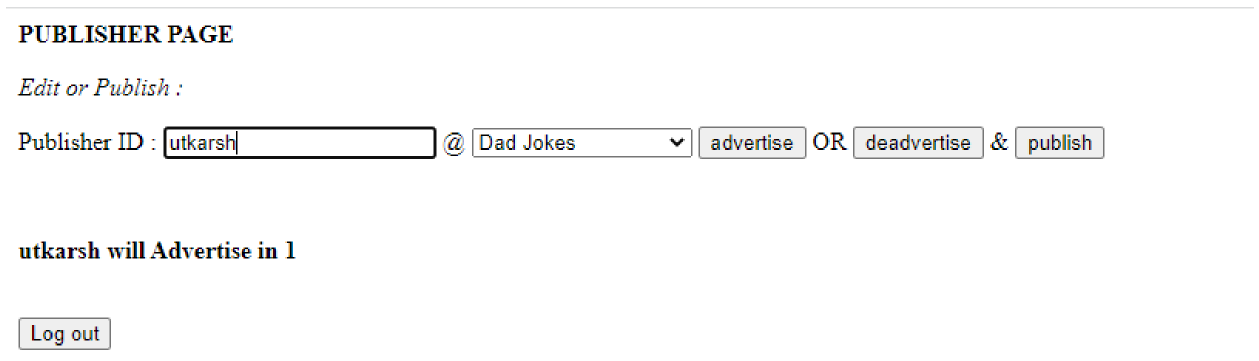


Fig 4: Application Login page

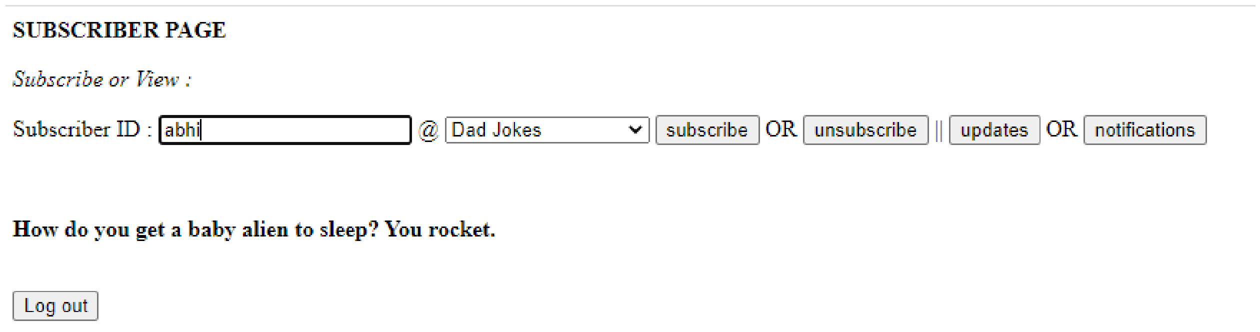
- If the user is Publisher, he/she will be redirected to a publisher page. This page allows users to Publish , advertise , de advertise to a particular topic with appropriate messages.
- User needs to enter the publisher id and select the topic he/she wants to publish.
- If the user is advertised to the topic, an advertisement will be displayed on the login page.



utkarsh will Advertise in 1

Fig 5: Publisher Page

- If the User is a Subscriber, he/she will be redirected to a Subscriber page.
- This page allows users to subscribe, unsubscribe, view updates and notifications.
- To view an update a user must be subscribed to that topic. If not, then the system will prompt the user with appropriate messages.


SUBSCRIBER PAGE

Subscribe or View :

Subscriber ID : abhi @ Dad Jokes subscribe OR unsubscribe || updates OR notifications

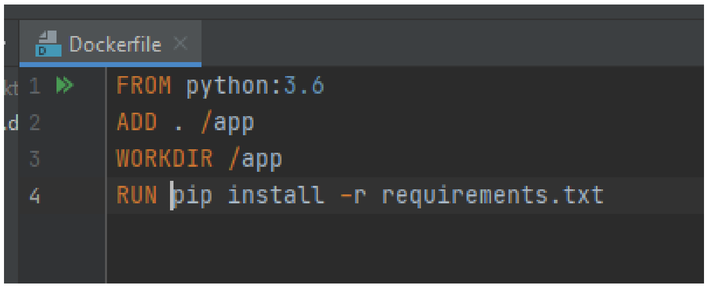
How do you get a baby alien to sleep? You rocket.

Fig 6: Subscriber Page

- Each Publisher and Subscriber page has a logout button, which will redirect the user to the login page.
- Our app.py connects to kafka to send and receive topic data.

4. Docker details

Dockerfile - This dockerfile pulls the python image and builds the application and installs all dependencies mentioned in *requirements.txt* file. After building it will run the *app.py* file.



```
FROM python:3.6
ADD . /app
WORKDIR /app
RUN pip install -r requirements.txt
```

Fig: Dockerfile for app.py

5. Project Structure

The project consists of the following file:

1. app.py
2. create_db
3. Dockerfile

4. docker-compose.yml
5. requirements.txt
6. consumerAPI
7. producerAPI
8. subService
9. pubService
10. templates
    a. index.html
    b. pub.html
    c. sub.html

6. Architecture Diagram

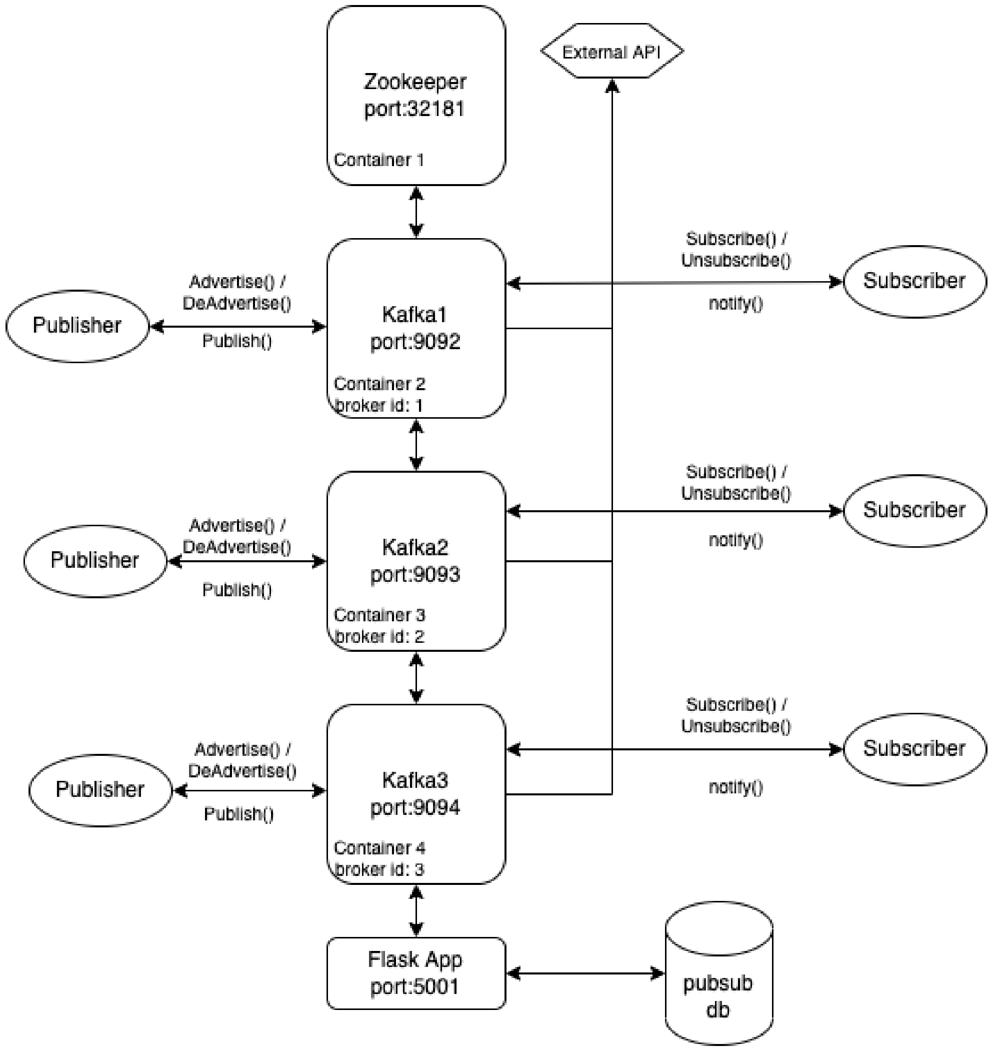


Fig: Architecture Diagram

7. Deployment Details

- Go to project folder (locate yml file)
- run **docker-compose up** in terminal
- Open **localhost:5001** in browser

SECURITY KEY :
Publisher - “pppp”
Subscriber - “ssss”

8. Conclusion

Publish/subscribe messaging is when a publisher sends a message to a topic and the message is forwarded to a subscriber.

The concept of pub/sub is easy to understand but every coding and programming language handles it differently, making it a little more challenging to learn across all platforms.Building an information system at scale using the Pub/Sub pattern benefits all stakeholders. The software development process benefits from the simplicity of the Pub/Sub pattern.

Here we have implemented a distributed pub/sub system with 3 Kafka broker containers,1 zookeeper container and 1 python app container which are able to send messages asynchronously and without the awareness of the publisher to the subscriber.

References

1. Kafka : https://hub.docker.com/r/confluentinc/cp-kafka/
2. Kafka blog: https://www.confluent.io/blog/
3. Docker Documentation , https://docs.docker.com/
4. Docker wikipedia, https://en.wikipedia.org/wiki/Docker_(software)
5. Flask documentation, https://flask.palletsprojects.com/en/2.0.x/tutorial/index.html