

Publisher Subscriber System

CSE-586 Distributed Systems

(Group 99)

Abhishek Kumar

UB Person no. 50419133

MS Computer Science and Engineering

School of Engineering and Applied Science

akumar58@buffalo.edu

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.

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.

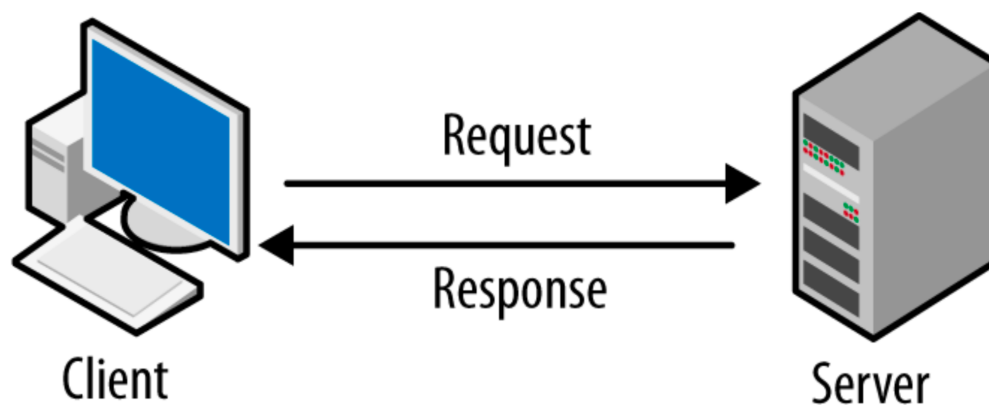


Fig 1: Client-Server model

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.

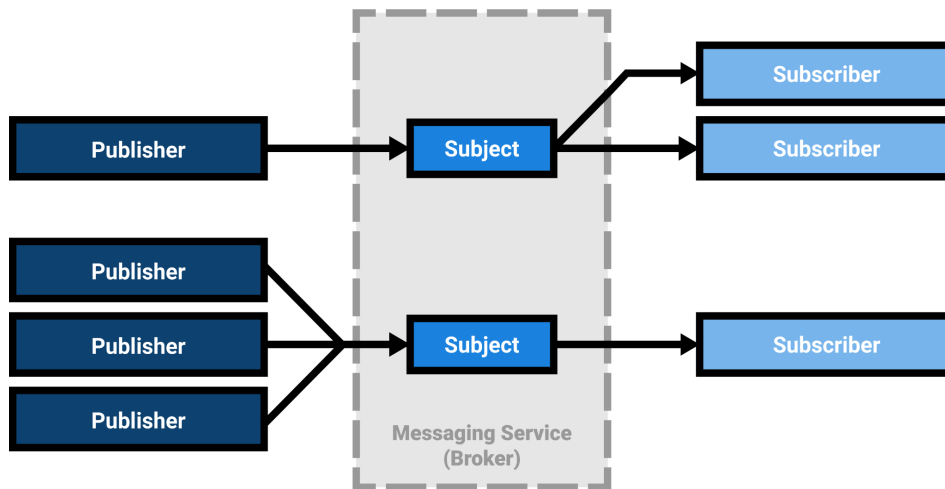


Fig 2: Publisher Subscriber System

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.

2. Implementation

2.1 Phase 1

Phase 1 was a basic client server (request-response) interaction system to demonstrate the use of docker containers.

The client and server interact using TCP socket based connection. Both client and server are running in docker containers in the local system. A filesystem storage was used to store the handshakes between client and server.

2.2 Phase 2

In phase 2 of the project we have implemented a centralized pub/sub system using Flask, sqlite3 and docker. The application processes the inputs from publishers and subscribers for 3 topics and provides output on the basis of user identity and functionality.

The application is scalable for any number of publishers and subscribers. The publisher publishes on any topic and a notification is sent to the subscriber subscribed to that topic. A subscriber is then able to login and view the notification and the latest updates on the topic.

2.3 Phase 3

In phase 3 our system contains 5 containers. Containers 1,2,3 are application servers which act as brokers to manage the pub/sub requests. Containers 4 and 5 are database containers which store the user and topic (event information). A routing mechanism handles the requests and the insert/updates of the events and users in the database.

The containers are created and managed using docker-compose.

2.3.1 System Architecture

- There are 3 broker containers which implement the user functions, publish, advertise, deadvertise, subscribe, unsubscribe, notify and viewnotification.
- The user sends a request to these brokers which routes the user requests to the database containers based on the topic and returns the appropriate response.

- An external API is used to fetch topic data, which is then stored in the respective db containers.
- The user can login the application as a Publisher or Subscriber
- Users need to have a security key to login the application. This is to ensure that new topics are published by publishers only.

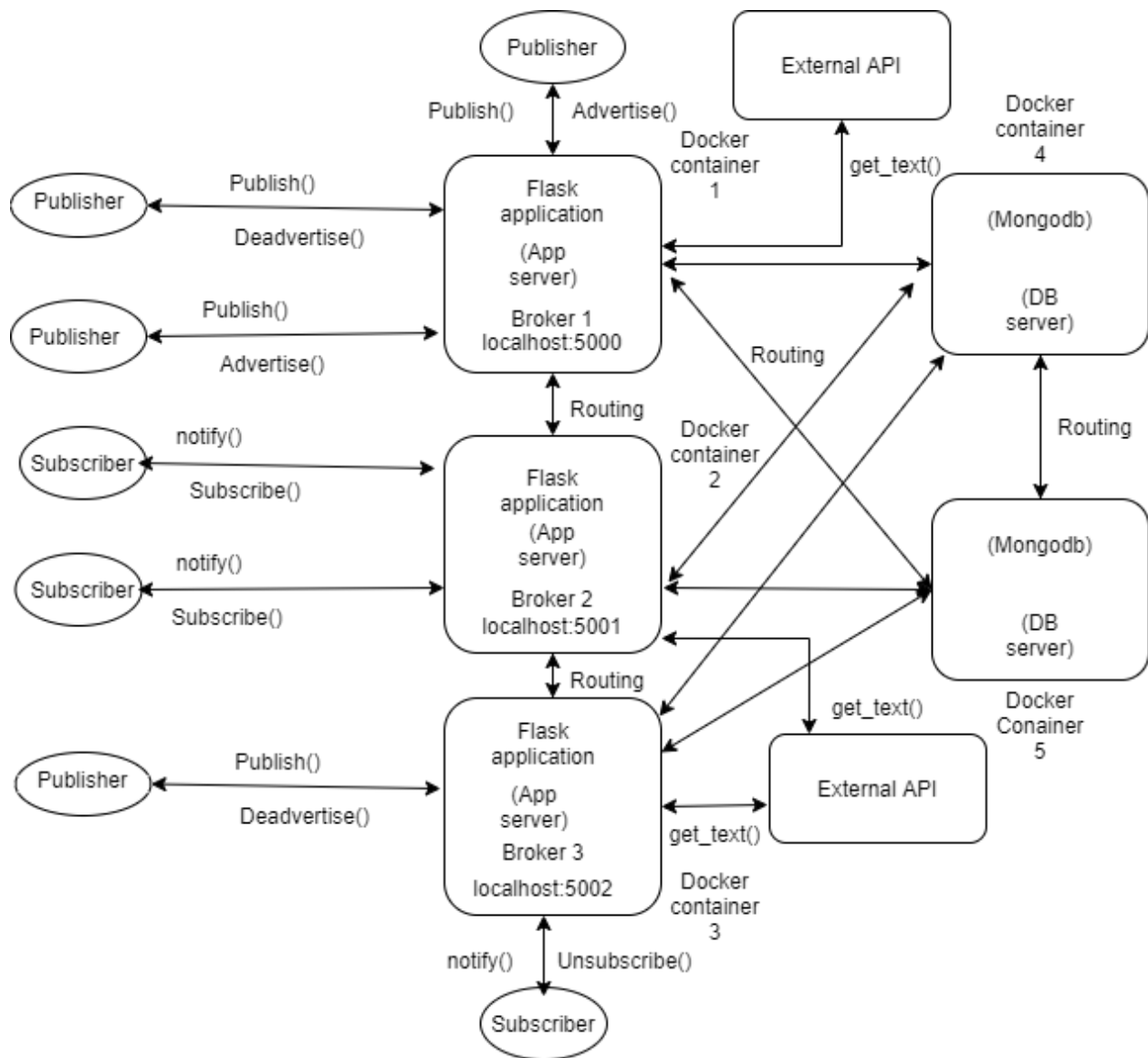


Fig 3: Architecture Diagram

2.3.2 UI details

The application is created using Flask, python, html and mongodb.

- The user logs in 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.

Text Generator Board



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.

PUBLISHER PAGE

Edit or Publish :

Publisher ID : @ OR &

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 : @ OR || OR

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

Fig 6: Subscriber Page

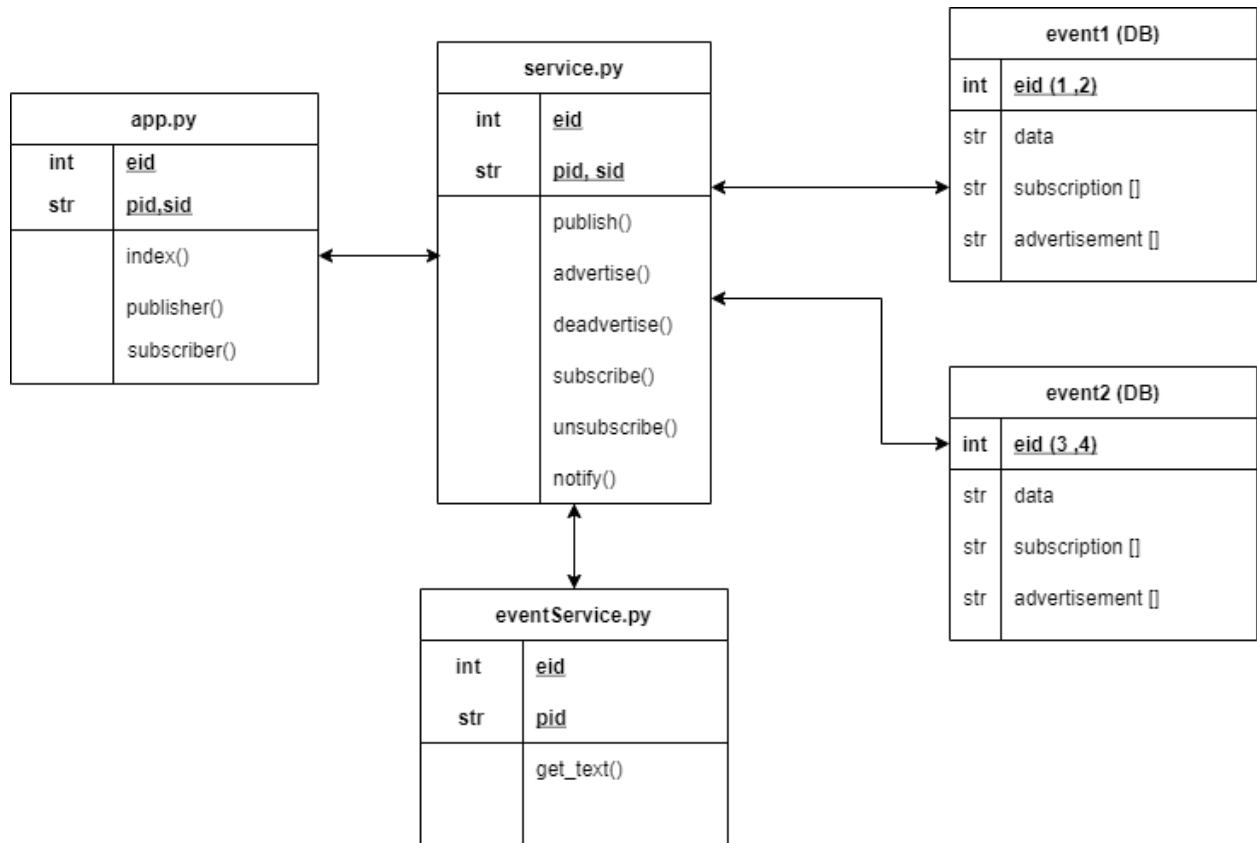
- Each Publisher and Subscriber page has a logout button, which will redirect the user to the login page.

2.3.3 Project Structure

The project consists of the following file:

1. app.py
2. service.py
3. Dockerfile
4. docker-compose.yml
5. requirements.txt
6. init-db.js
7. init-db2.js
8. templates
 - a. index.html
 - b. pub.html
 - c. sub.html

2.3.4 Class Diagram



Class Diagram

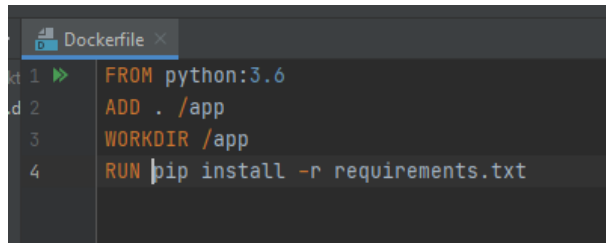
Fig 7: Class Diagram

Application functions

1. *publish(pid,eid)* - Takes publisher id (pid) and publishes new data at topic (eid)
2. *advertise(pid,eid)* - Takes publisher id (pid) and adds user to advertisement table in db based on topic (eid)
3. *deadvertise(pid,eid)* - Takes publisher id (pid) and removes user from advertisement table in db based on topic (eid)
4. *subscribe(sid,eid)* - Takes subscriber id (sid) and adds user to the subscription table in db based on topic (eid)
5. *unsubscribe(sid,eid)* - Takes subscriber id (sid) and removes user from the subscription table in db based on topic (eid)
6. *notify(sid,eid)* - Allows Subscriber to view the recent updates based on topic (eid)

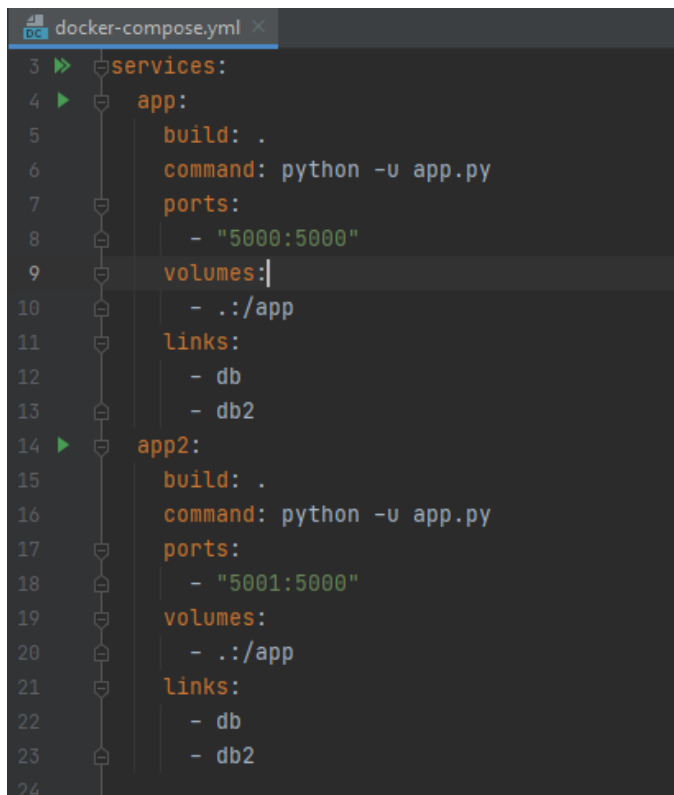
2.3.5 Docker details

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.



```
Dockerfile
1 FROM python:3.6
2 ADD . /app
3 WORKDIR /app
4 RUN pip install -r requirements.txt
```

Docker-compose.yml - This file is used to run the Dockerfile and also creates the 5 containers(app, app2, app3, db and db2) for the application.



```
docker-compose.yml
3 services:
4   app:
5     build: .
6     command: python -u app.py
7     ports:
8       - "5000:5000"
9     volumes:|
10      - ./app
11     links:
12       - db
13       - db2
14   app2:
15     build: .
16     command: python -u app.py
17     ports:
18       - "5001:5000"
19     volumes:
20       - ./app
21     links:
22       - db
23       - db2
24
```

```
docker-compose.yml
25 db:
26   image: mongo:latest
27   hostname: test_mongodb
28   environment:
29     - MONGO_INITDB_DATABASE=event1_db
30     - MONGO_INITDB_ROOT_USERNAME=root
31     - MONGO_INITDB_ROOT_PASSWORD=pass
32   volumes:
33     - ./docker-entrypoint-initdb.d/init-db.js:/docker-entrypoint-initdb.d/init-db.js:ro
34     # - ./init-db.js:/docker-entrypoint-initdb.d/init-db.js:ro
35   ports:
36     - 27017:27017
37
38 db2:
39   image: mongo:latest
40   hostname: test_mongodb2
41   environment:
42     - MONGO_INITDB_DATABASE=event2_db
43     - MONGO_INITDB_ROOT_USERNAME=root
44     - MONGO_INITDB_ROOT_PASSWORD=pass
45   volumes:
46     - ./docker-entrypoint-initdb.d/init-db2.js:/docker-entrypoint-initdb.d/init-db2.js:ro
47     # - ./init-db2.js:/docker-entrypoint-initdb.d/init-db2.js:ro
48   command: mongod --port 27018
49   # ports:
50     # - 27018:27018
```

Port details for the containers :

1. app - port 5000
2. app2 - port 5001
3. app3 - port 5002
4. db - port 27017
5. db2 - port 27018

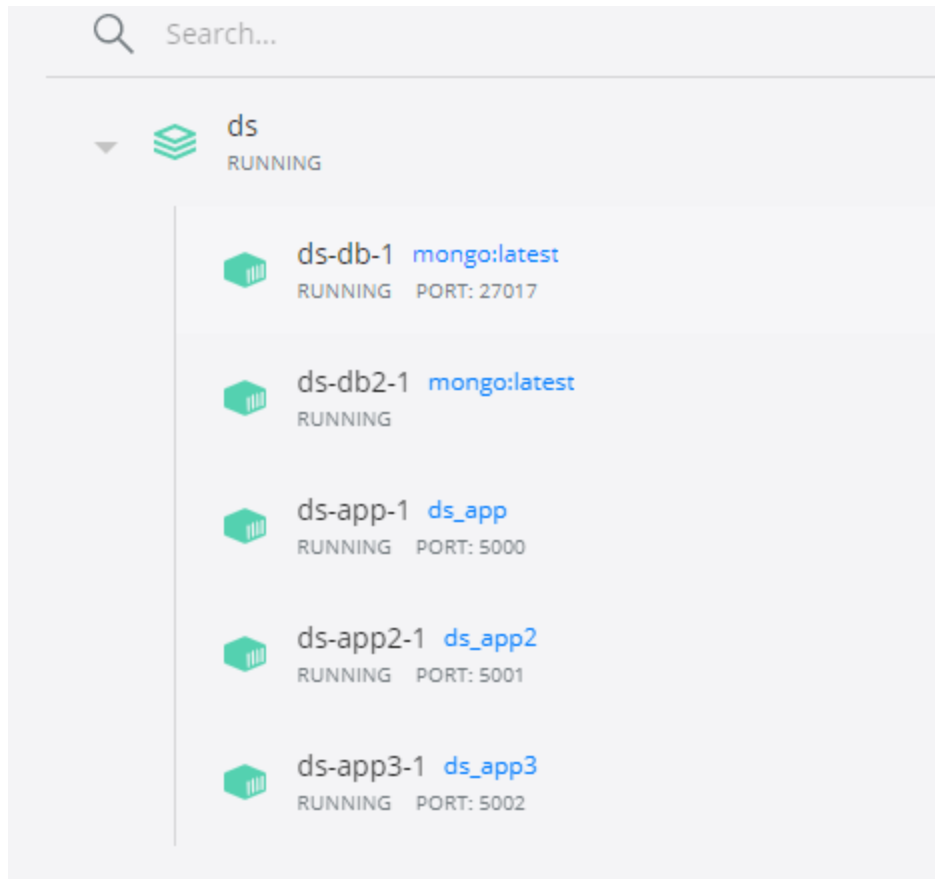
2.3.6 Deployment Details

- Go to project folder (locate yml file)
- run docker-compose up in terminal
Both dbs exit with code -273 during initialization (although db containers have been initialized and volumes are created)
- stop the run by pressing ctrl+c
- Run docker-compose up again in terminal
All containers run without exiting (check in docker app)
- open localhost:5000 in browser for broker 1
- open localhost:5001 in browser for broker 2
- open localhost:5002 in browser for broker 3

SECURITY KEY :

Publisher - “pppp”

Subscriber - “ssss”



3. 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 broker containers and 2 database containers which are able to send messages asynchronously and without the awareness of the publisher to the subscriber.

References

1. Pub/Sub blog, <https://ably.com/topic/pub-sub>
2. Pub/Sub blog, <https://blog.stackpath.com/pub-sub/>
3. Docker Documentation , <https://docs.docker.com/>
4. Docker wikipedia, [https://en.wikipedia.org/wiki/Docker_\(software\)](https://en.wikipedia.org/wiki/Docker_(software))
5. Flask documentation, <https://flask.palletsprojects.com/en/2.0.x/tutorial/index.html>
6. MongoDB documentation, <https://docs.mongodb.com/>

Contributions

1. Abhishek Kumar (50419133) - Backend, Database, Docker
2. Utkarsh Kumar (50419745) - Frontend, routing, api