

CSE 586: Distributed Systems

Phase-3 Report

By

SRIKAR PANUGANTI- 50411922
SUHAS REDDY EDAVALLI- 50363272
(TEAM-11)

Decentralized Pub-Sub using Containerization for Meteorological Notifications

Summary:

In this phase we have implemented Decentralized Pub-Sub system using docker to provide real-time Meteorological data notifications moving from Centralized model to achieve scalability.

The Decentralized pub-sub system has multiple broker nodes to handle the Subscribers and reduce the single dependency on one server when compared to Centralized Pub-Sub system.

We have routing/broker network table which has the information of connected broker nodes in the network which is used to propagate the notifications to client.

Topics of Interest:

1. Elevation Info
2. Soil Biological Data
3. Sea Level Info
4. Astronomical Data

We are using an External API - Storm Glass (<https://stormglass.io/>) to pull the above data to be published as events for which subscribers can be notified.

- It uses API- key based authentication mechanism. A user can get a key registering on the above website.

Major Components in System:

1. Publishers:

We have built a polling server to fetch all the Publishers from the above External API. This server is responsible to pull the data from the API's periodically and push them to the Middleware network.

2. Middleware/Broker Network

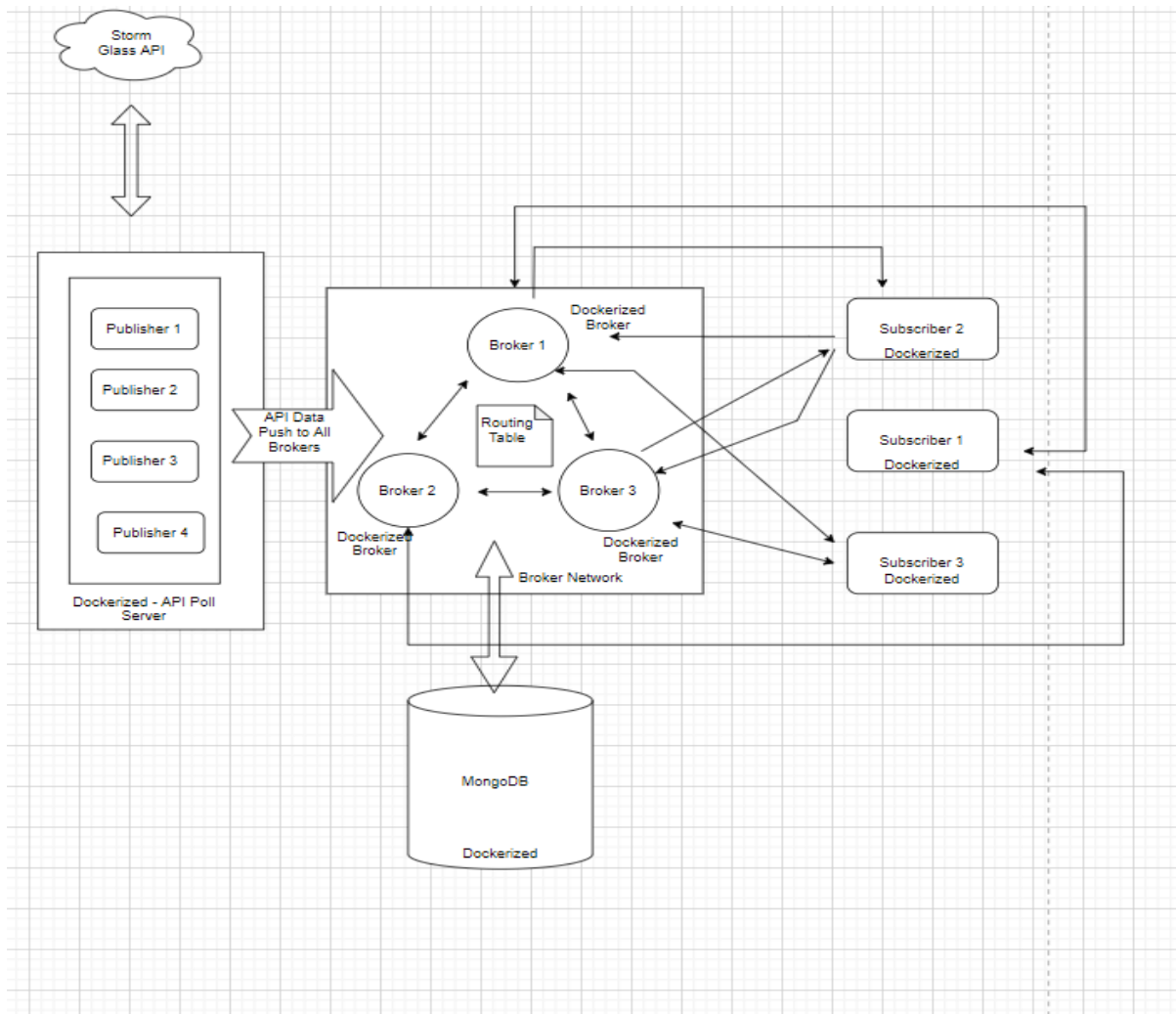
Middleware/Centralized server has below responsibilities.

- Maintain a routing/network table which has address of the broker nodes in the network.
- Each broker node handles its designated topics that are propagated to the client.
- Provide an API for Subscribers to register, subscribe, unsubscribe, and get the notifications or events which the Clients-side application interacts using HTTP calls.
- Maintain a Queue for handling the events and pushing the notifications to the subscribers and the topics being published.
- Interaction with Database to store subscribers' info, topics being published along with storing the events data being published.

3. Client-side Application / Subscribers:

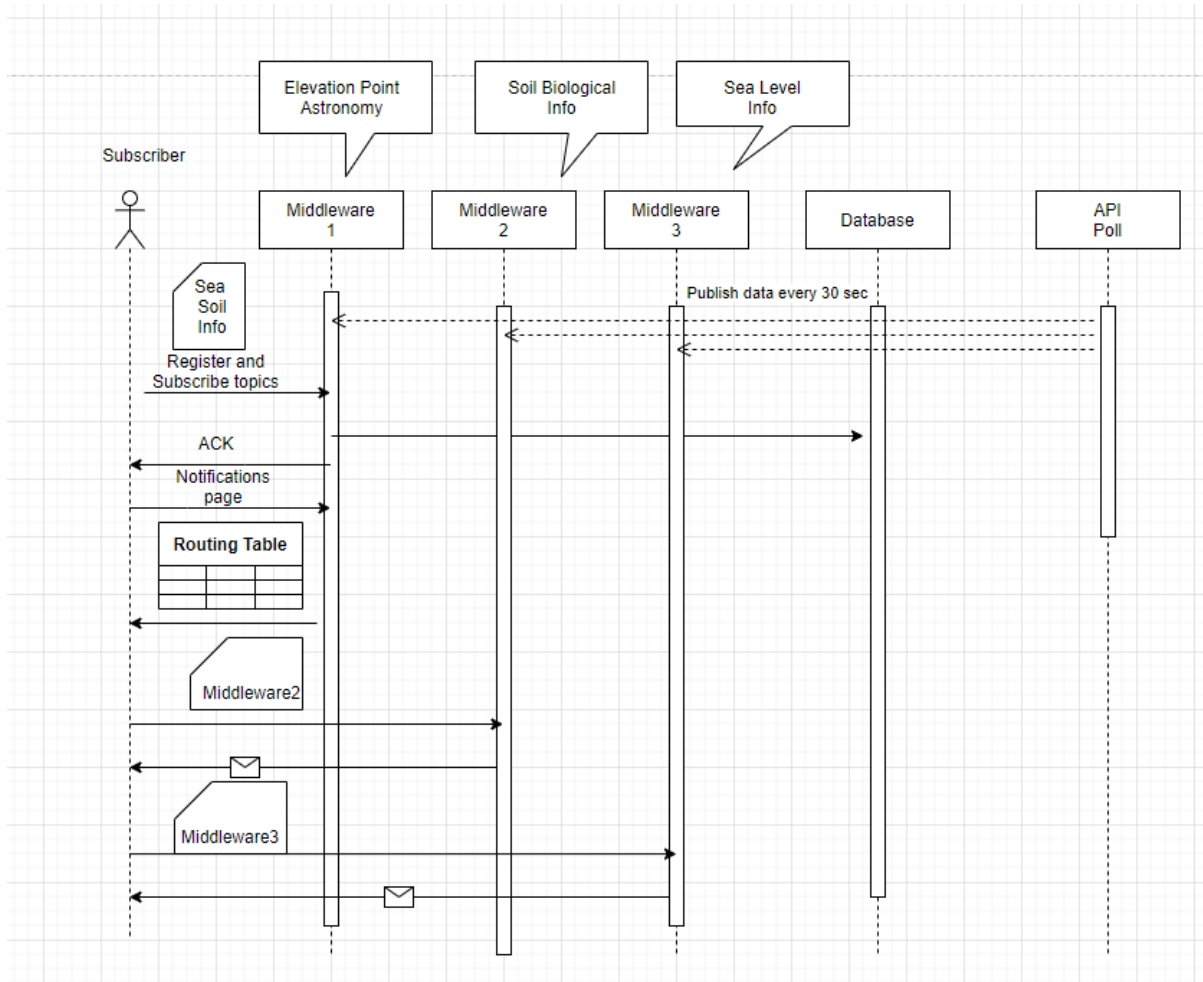
Client-side application provides some basic functionalities to interact with server to register the subscribers and allow them to subscribe for topics of their interest. It also has functionality to allow a subscriber for subscription to multiple topics and view the event notifications. They were also given the option to unsubscribe from the UI.

Architecture Diagram:



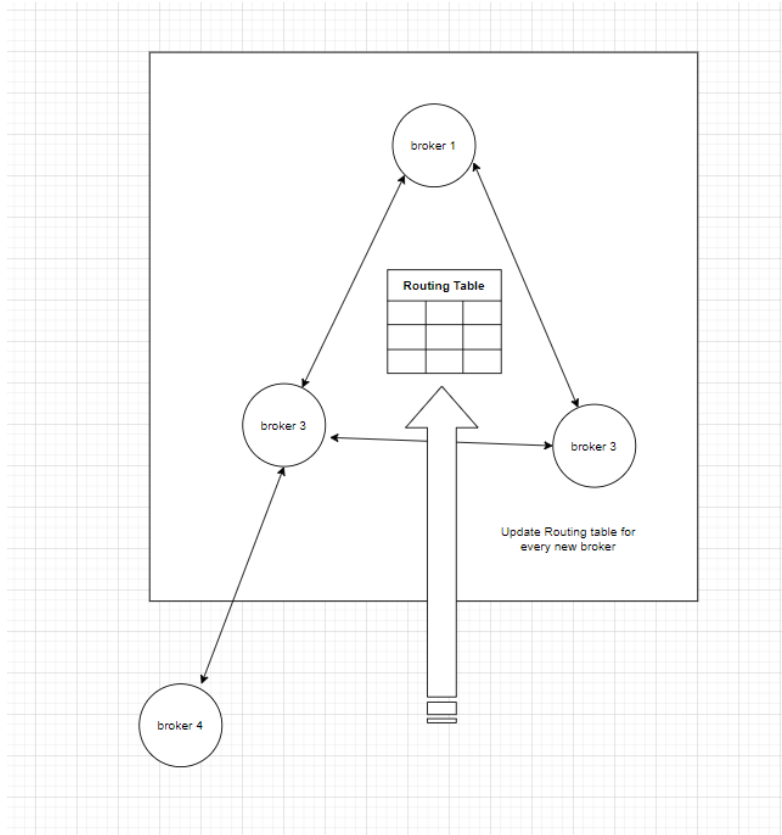
- As the picture depicts, we maintain a routing table among all brokers which contain information of the address of the brokers in the network and topics each broker is handling.
- Subscriber gets the notifications using the HTTP Polling. When the subscriber polls the broker regarding the notification, Broker 1 sends the information of brokers it needs to poll to get the notifications of interest.
- Then Subscriber directly polls the respective broker and gets its required information.
- As this model is developed using HTTP polling, exchange of information/data between brokers becomes redundant as client can directly poll the broker for his topic of interest.
- If we want to scale this model, we just introduce a new broker to the existing “bridge” network and update the routing information.

Sequence Diagram:



- In the above diagram we consider the Broker / Middleware network is already established and we look at the sequence diagram from the view of the subscriber.
- We discuss the broker network establishment and its communication in the next section.
- API poll server polls the External Storm Glass API every 30 seconds and pushes the data to the middleware.
- Each Middleware has its designated Topic to handle and filters the data from the poll service accordingly.
- Subscriber registers and subscribes of his topic of interest with the Middleware 1.
- Middleware 1 sends the broker's data of where he can get his subscribed info back to the Subscriber.
- Subscriber now directly polls respective brokers and the get the notifications.

Broker Network and Communication:



- Brokers join the network by updating the routing the centralized routing table with its broker name, topic(s) it is going to handle and the brokers' address.
- With this model any broker can enter and leave the network freely without much process.
- Apart from the conceptualized broker network discussed above, all the brokers are connected via “*bridge*” network which allows them to communicate among the barriers of the docker container.

```
networks:
  backend:
    driver: bridge
volumes:
  mongodbddata:
    driver: local
  appdata:
    driver: local
```

- In terms of Docker, a bridge network uses a software bridge which allows containers connected to the same bridge network to communicate, while providing isolation from containers which are not connected to that bridge network

We have established a broker network of 3 nodes and distributed above 4 topics among them. So, each broker node has the responsibility of notifying the Subscriber when the topic of his interest is published.

Topics Distribution among broker nodes:

	Broker Node	Topics Handled	Location
1	Middleware	Elevation Info, Astronomy	Host=0.0.0.0 Port=5000
2	Middleware01	Soil Biological Data	Host=0.0.0.0 Port=5001
3	Middleware02	Sea Level Info	Host=0.0.0.0 Port=5002

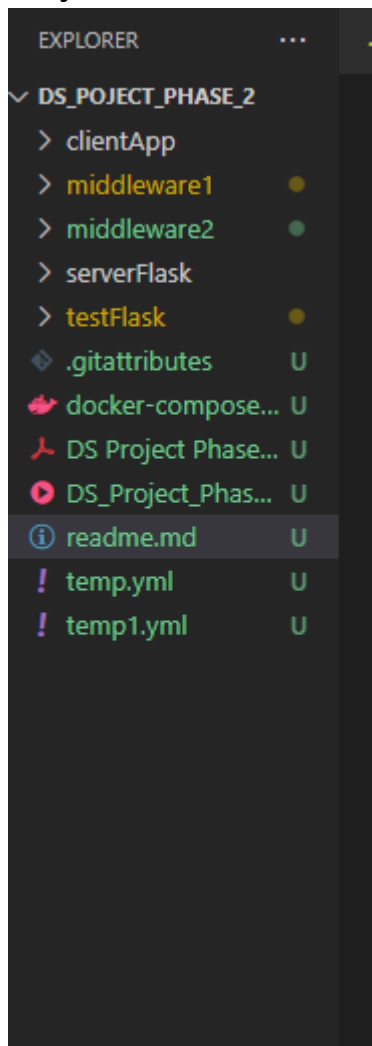
Technology Stack:

- Front-end: ReactJs, HTML, CSS
- Server / Broker node: Flask, python, REST API
- Database: MongoDB
- Containerization: Docker

API's Exposed:

- Advertise() - This advertises the topics the publisher would publish.
- CreateSubscriber() - To register subscribers in order to register for topics.
- Subscribe() - Helps subscribers to subscribe to his topic of interest.
- Notify() - This publishes the events to the subscriber.
- Unsubscribe() - This is used to unsubscribe for a topic.
- Deadvertise() – This is used to deadvertise/revoke advertisement.
- pushDefaultBrokers() – routing data to establish basic broker network.
- pushDefaultPublishers() – Publisher data for subscriber matching.

Project Structure:



clientApp has all the source code related to the react front-end.

testFlask is the main middleware directory aka “Middleware”.

Middleware1 and Middleware2 are other two broker nodes which are distinguished to handle the respected topics, self-identifying server instance and to dockerize individually.

serverFlask has all the source code related to API poller service.

docker-compose.yml has all the instructions to dockerize and build the applications with their dependencies.

Application Screenshots:

1. Sign-up Page:

Meteorological Pub-Sub App [Login](#) [Sign up](#) [Notifications](#)

Sign Up

Name
First name

Email address
Enter email

Username
Enter Username

Password
Enter password

[Sign Up](#)

2. Login Page:

Meteorological Pub-Sub App [Login](#) [Sign up](#) [Notifications](#)

Subscriber login

Username
Enter Username

Password
Enter password

[Submit](#)

3. Subscribe / Unsubscribe Page:

Meteorological Pub-Sub App Login Sign up Notifications

Name: elevationPoint

Subscribe UnSubscribe

Name: bio

Subscribe UnSubscribe

Name: seaLevel

Subscribe UnSubscribe

Name: astronomy

Subscribe UnSubscribe

Notifications

4. Notifications Page

Meteorological Pub-Sub App Login Sign up Notifications

```
("api":"elevationPoint","data":{"elevation":-1.9980000257492065},"meta":{"cost":1,"dailyQuota":50,"distance":0.12,"elevation":{"source":"GEBCO Compilation Group (2019) GEBCO 2019 Grid (doi:10.5285/836f016a-33be-6ddc-e053-6c86abc0788e)","unit":"m"},"lat":58.7984,"lng":17.8081,"requestCount":17})

("api":"elevationPoint","data":{"elevation":-1.9980000257492065},"meta":{"cost":1,"dailyQuota":50,"distance":0.12,"elevation":{"source":"GEBCO Compilation Group (2019) GEBCO 2019 Grid (doi:10.5285/836f016a-33be-6ddc-e053-6c86abc0788e)","unit":"m"},"lat":58.7984,"lng":17.8081,"requestCount":17})

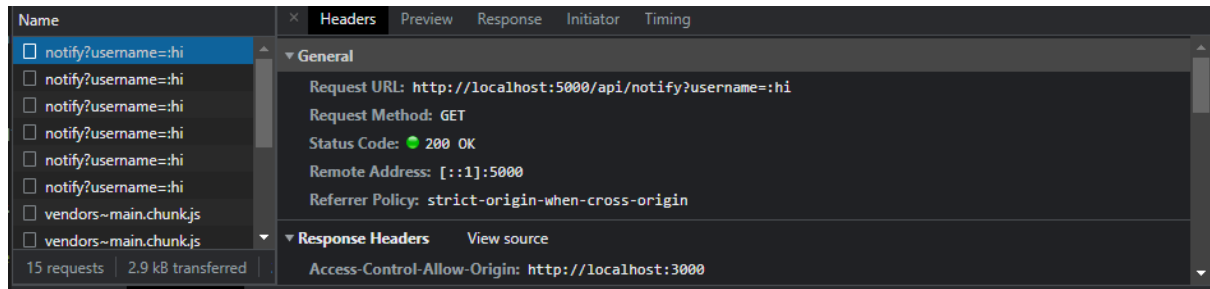
("api":"bio","hours":{"iron":{"mercator":0,"sg":0},"nitrate":{"mercator":0.42,"sg":0.42},"time":{"time":"2021-11-06T23:00:00+00:00"},"iron":{"mercator":0,"sg":0},"nitrate":{"mercator":0.42,"sg":0.42},"time":"2021-11-07T00:00:00+00:00"},"meta":{"cost":1,"dailyQuota":50,"end":"2021-11-07 00:46","lat":58.7984,"lng":17.8081,"params":{"iron","nitrate"},"requestCount":18,"start":"2021-11-06 23:00"})

("api":"seaLevel","data":{"sg":-1.86,"time":"2021-11-06T23:00:00+00:00"},"sg":-1.3,"time":"2021-11-07T00:00:00+00:00"},"meta":{"cost":1,"dailyQuota":50,"datum":"MSL","end":"2021-11-07 00:46","lat":43.38,"lng":-3.01,"requestCount":19,"start":"2021-11-06 23:00","station":{"distance":4,"lat":43.36,"lng":-3.05,"name":"bilbao","source":"sg"}})
```

Clear

Network Calls to each middleware:

Broker 1 at port 5000

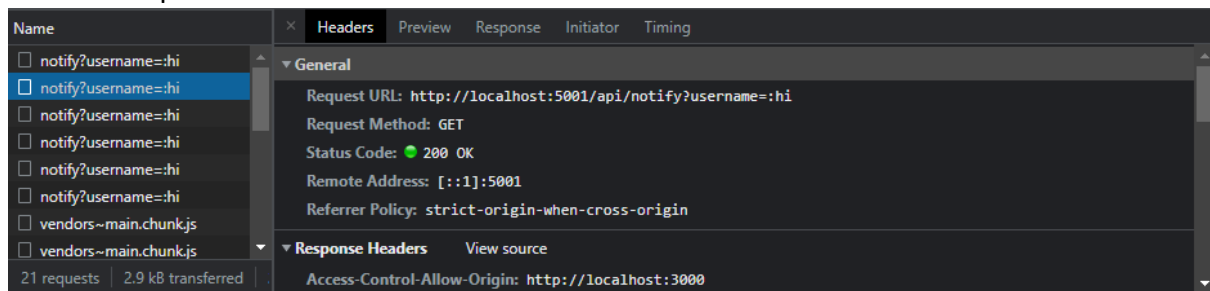


Network inspector showing a list of requests on the left and details for the selected request on the right. The selected request is a GET to `http://localhost:5000/api/notify?username=:hi` with status 200 OK. The response header `Access-Control-Allow-Origin` is `http://localhost:3000`.

Name	Headers	Preview	Response	Initiator	Timing
<input type="checkbox"/> notify?username=:hi	▼ General				
<input type="checkbox"/> notify?username=:hi	Request URL: <code>http://localhost:5000/api/notify?username=:hi</code>				
<input type="checkbox"/> notify?username=:hi	Request Method: GET				
<input type="checkbox"/> notify?username=:hi	Status Code: 200 OK				
<input type="checkbox"/> notify?username=:hi	Remote Address: <code>[::1]:5000</code>				
<input type="checkbox"/> notify?username=:hi	Referrer Policy: <code>strict-origin-when-cross-origin</code>				
<input type="checkbox"/> vendors~main.chunk.js	▼ Response Headers View source				
<input type="checkbox"/> vendors~main.chunk.js	Access-Control-Allow-Origin: <code>http://localhost:3000</code>				

15 requests | 2.9 kB transferred

Broker 2 at port 5001

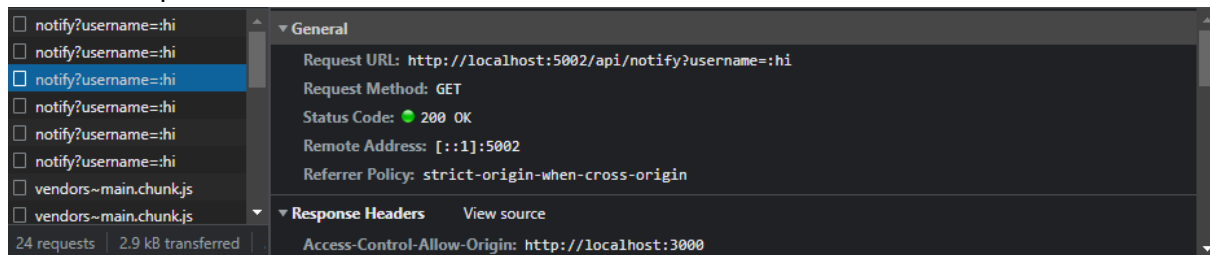


Network inspector showing a list of requests on the left and details for the selected request on the right. The selected request is a GET to `http://localhost:5001/api/notify?username=:hi` with status 200 OK. The response header `Access-Control-Allow-Origin` is `http://localhost:3000`.

Name	Headers	Preview	Response	Initiator	Timing
<input type="checkbox"/> notify?username=:hi	▼ General				
<input type="checkbox"/> notify?username=:hi	Request URL: <code>http://localhost:5001/api/notify?username=:hi</code>				
<input type="checkbox"/> notify?username=:hi	Request Method: GET				
<input type="checkbox"/> notify?username=:hi	Status Code: 200 OK				
<input type="checkbox"/> notify?username=:hi	Remote Address: <code>[::1]:5001</code>				
<input type="checkbox"/> notify?username=:hi	Referrer Policy: <code>strict-origin-when-cross-origin</code>				
<input type="checkbox"/> vendors~main.chunk.js	▼ Response Headers View source				
<input type="checkbox"/> vendors~main.chunk.js	Access-Control-Allow-Origin: <code>http://localhost:3000</code>				

21 requests | 2.9 kB transferred

Broker 3 at port 5002



Network inspector showing a list of requests on the left and details for the selected request on the right. The selected request is a GET to `http://localhost:5002/api/notify?username=:hi` with status 200 OK. The response header `Access-Control-Allow-Origin` is `http://localhost:3000`.

Name	Headers	Preview	Response	Initiator	Timing
<input type="checkbox"/> notify?username=:hi	▼ General				
<input type="checkbox"/> notify?username=:hi	Request URL: <code>http://localhost:5002/api/notify?username=:hi</code>				
<input type="checkbox"/> notify?username=:hi	Request Method: GET				
<input type="checkbox"/> notify?username=:hi	Status Code: 200 OK				
<input type="checkbox"/> notify?username=:hi	Remote Address: <code>[::1]:5002</code>				
<input type="checkbox"/> notify?username=:hi	Referrer Policy: <code>strict-origin-when-cross-origin</code>				
<input type="checkbox"/> vendors~main.chunk.js	▼ Response Headers View source				
<input type="checkbox"/> vendors~main.chunk.js	Access-Control-Allow-Origin: <code>http://localhost:3000</code>				

24 requests | 2.9 kB transferred

Advantages:

- Due to establishment of broker network, our application is scalable now compared to the earlier centralized network.
- With containerization technology, we can be able to span up the Publishers and Broker nodes as per the load/ traffic in the network.
- We can achieve high availability of data at any given point of time as we can have multiple publishers and brokers supporting a topic even if a node goes down.

Scope for Improvement:

- Improvements in the UI/UX, to improve user experience.
- Introduce hybrid push/pull mechanism for delivering the notifications.
- Use database shard mechanism to improve latency.

Project Contributors:

1. Srikar Panuganti – 50411922
 - Front-end: React Application
 - Components in Middleware in FLASK
2. Suhas Reddy Edavalli – 50363272
 - API poller service in FLASK
 - Components in Middleware in FLASK

We contributed equally right from design, development to documentation.