Building a job-posting system using a Publisher-Subscriber pattern

CSE586 Project 2 November 12th, 2018

Pratik Vagyani
Department of Computer Science
Person Number: 50288741
pratikap@buffalo.edu

Aditya Manoharan
Department of Computer Science
Person Number: 50288615
avazhipo@buffalo.edu

Abstract

A **publisher-subscriber** system is a messaging pattern where publishers publish messages without knowledge of the recipients, and the subscribers "subscribe" to these messages.

Here, this pattern is used to build a shift-posting system where subscribers subscribe to postings in certain departments. The publishers publish to the system, and the system notifies the subscribers in case of a new posting.

The technologies used are Flask for the publishers, subscribers, and the publish-subscribe system, and Docker, in order to deploy multiple instances of each of them on different IPs. Bootstrap and JS is used to display an output in the browser.

1 Introduction

Subboard is a web application built to help part-time workers pickup and drop shifts in a stream-lined system. It has been built as a proof-of-concept for a publisher-subscriber pattern, to show how it can be implemented for a real world problem.

In this application, the users can either post(publish) or hunt(subscribe) for shift postings in a certain department using their email addresses. The notifications (in case of a new posting) is sent to their email addresses.

In order to demonstrate the working of this project adequately, the implementation has been divided into three phases, each to demonstrate one part of the deployed system.
The UX is also functional yet minimal, with just two navigations targets, one to publish, and one to subscribe.

The component structure, technical design choices, and performance are detailed further in this report, divided by Phase.

1.1 Publisher Subscriber System :

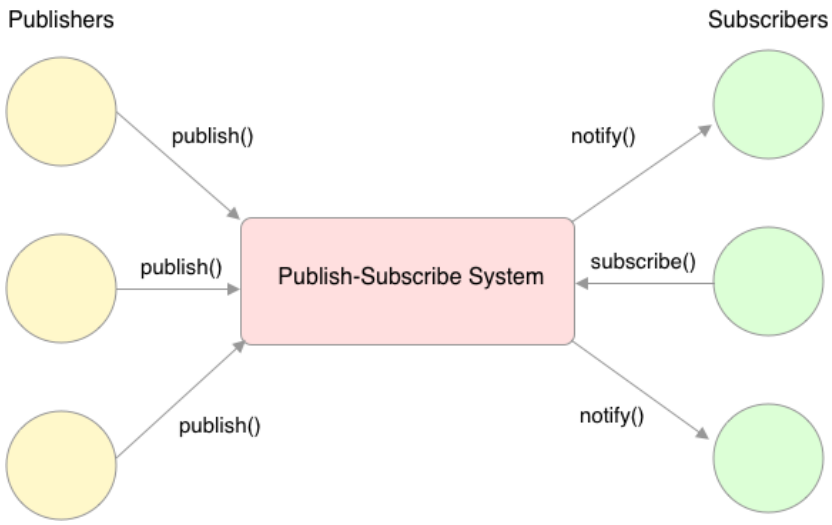


Fig 1.1 Model of a Publisher-Subscriber System

The above model describes a general publisher-subscriber system.

2 Technologies and Stack

2.1 Backend

Flask (Python Microframework) is used for the backend. Flask is chosen because it is a great tool for setting up a backend without compromising on performance. Since it's not a complete web framework, it does not limit the choices of other components of the applications, and allows to be used easily with any other technologies/frameworks; For instance PostgreSQL is used here using `psycopg2`

Flask is also based on Python, and since Python allows for quick prototyping, it is a good choice for the project. Also, this instantly allows to use any python library in the project. For example, the module `requests` is used to send requests between containers.

2.2 Database

PostgreSQL is used as the database for this project. This is the choice because a lot of the functionality of the project involves creating entries in databases, and PSQL allows for efficient storage and retrieval, while using familiar SQL commands.

2.3 Frontend

The Frontend is done using plain HTML/CSS and several Bootstrap components. Javascript is used for certain calls, and processing on the front-end.

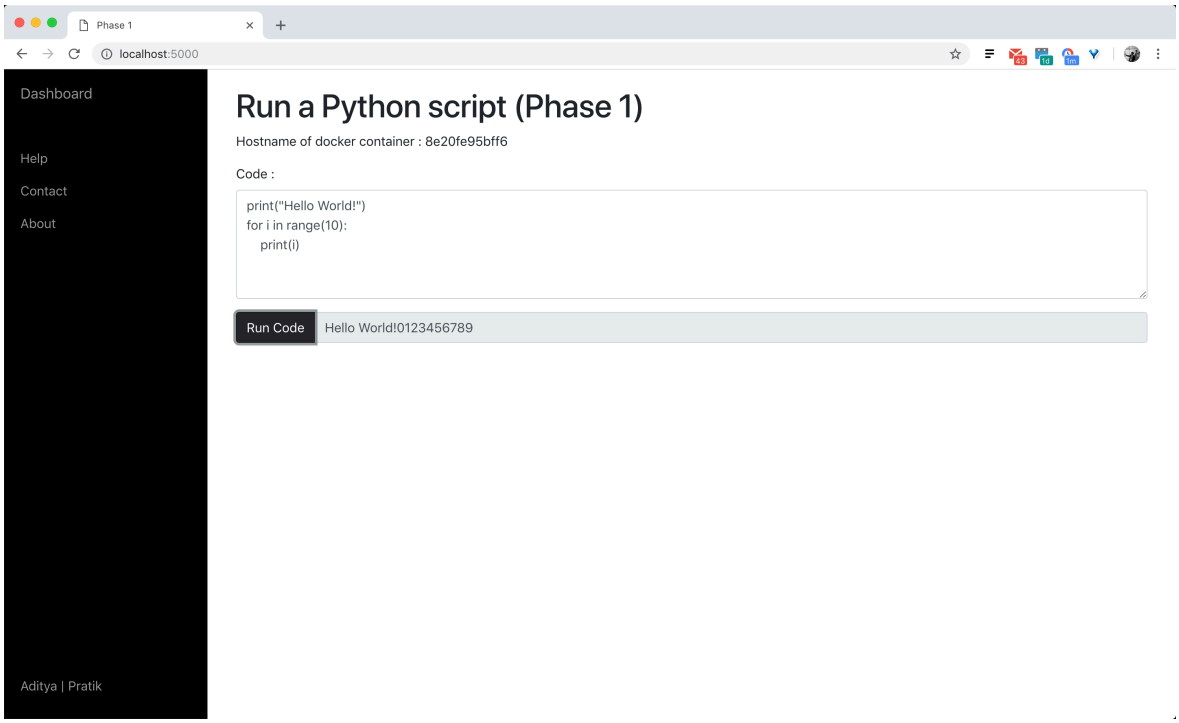
3 Implementation and Phases

The project is implemented in three phases, each of which is explained in detail below

3.1 Phase 1

This phase is primarily to **demonstrate the usage of Docker**. Here, a simple web application to execute a Python script is built.
On giving a Python script as input, the program displays the output in the textbox below.

Screenshot



The architecture is a simple client-server pattern. The server is deployed on Docker, and the site is accessed through the exposed port. The program reads the textbox and send it to Flask backend via a POST call. It is executed and returned to the front end.

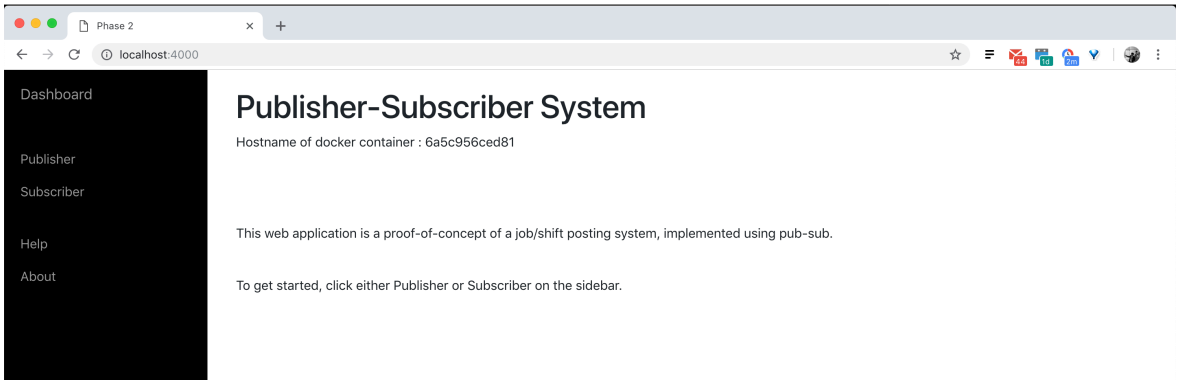
As shown in the screenshot above, this is deployed on Docker using a Dockerfile to build the image, and the hostname of the container (`8e20fe95bff6`) is displayed on the web page.

3.2 Phase 2

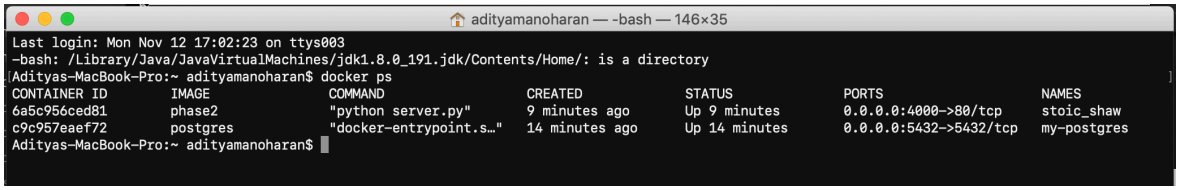
Phase 2 demonstrates a C**entralized publisher-subscriber system**.

A subscriber subscribes to a selection of four events(Departments), with their email address in the subscriber page.

The publisher publishes a shift posting for one of the events in the publisher page.
Once published, the system notifies the subscribers of that event by email.



The running containers are shown below. This phase has **two docker containers**: one for the database, and one for the server.

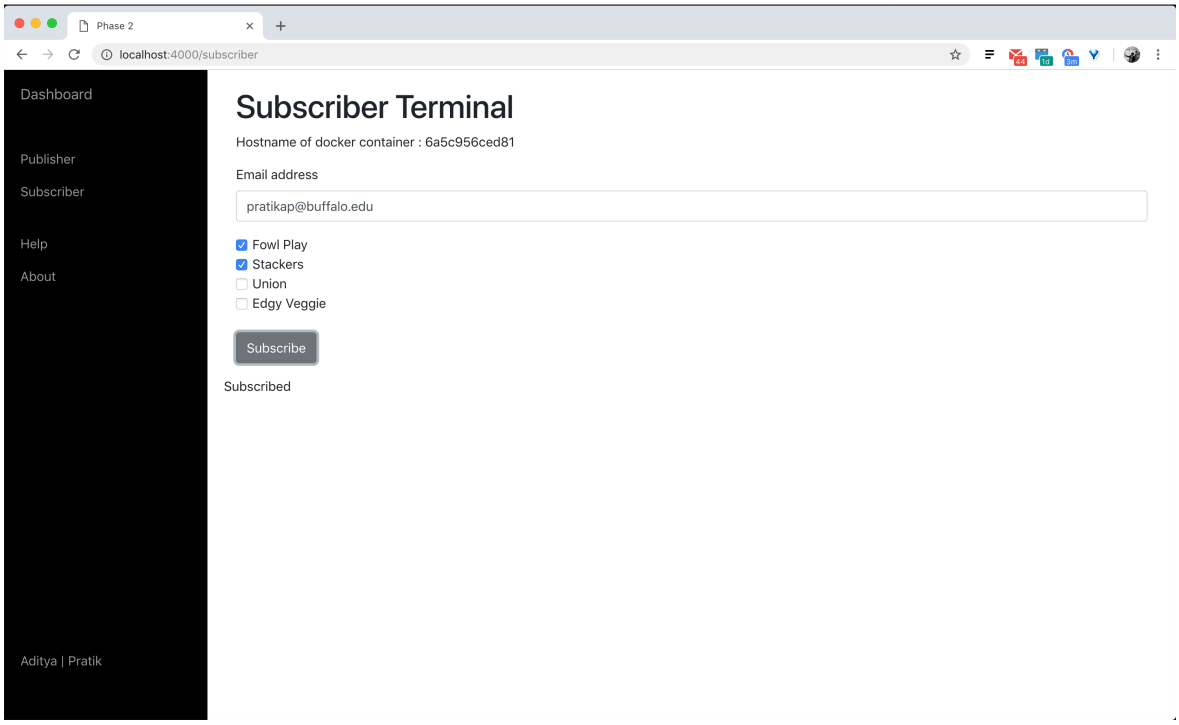


This follows the architecture if Fig 1.1, which is a general pub-sub system with no distributed components.

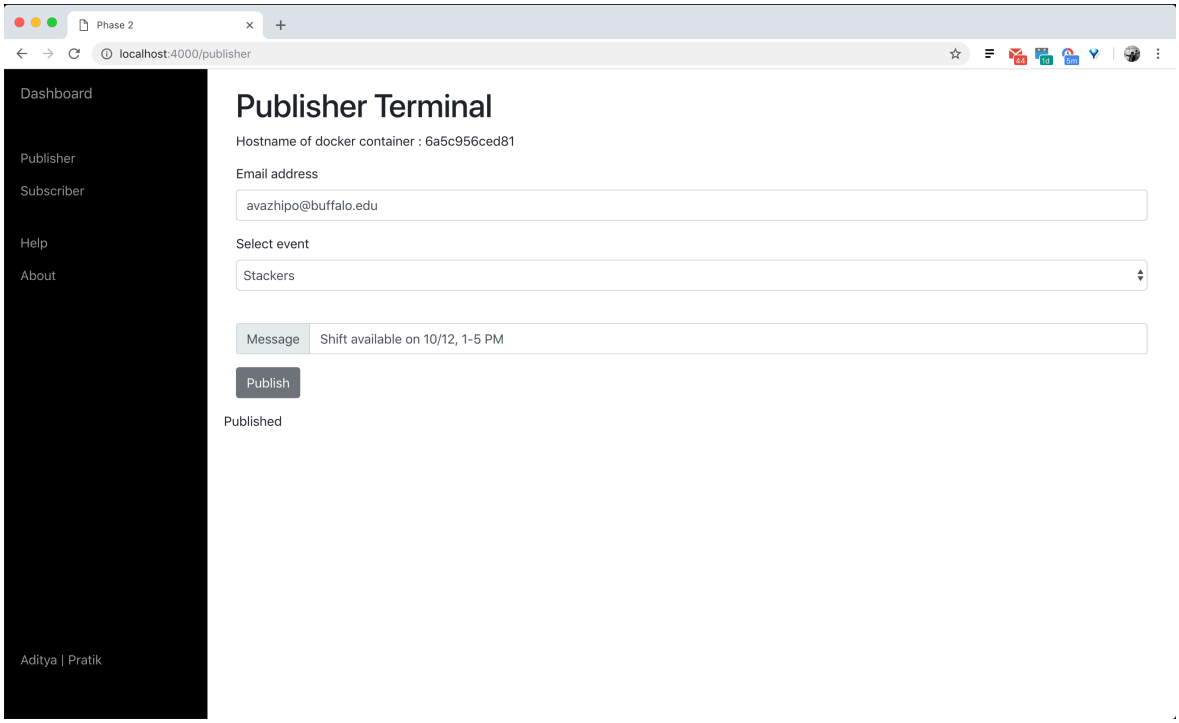
The server stores all the data related to subscription by communicating with the database container.

When the server gets a publish request for an event, it looks in the database for that particular event and retrieves the email address list. It then sends out email notifications to the subscribers.

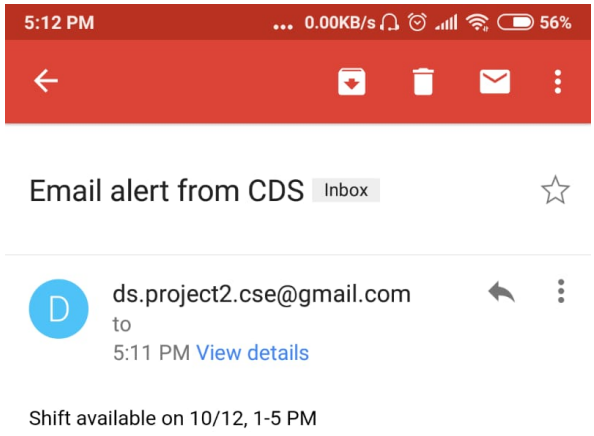
In the subscriber terminal, the user below subscribes to two departments with their email address



On success, this populates the table in the database and appends the user's email address to the tuple of that department.



Once a publisher publishes an event above, the users subscribed to the event are notified via email, as shown below.



3.3 Phase 3

Phase 3 is a **distributed implementation of a pub-sub system**, where there are multiple brokers. The components, and the operations in this implementation is shown below.

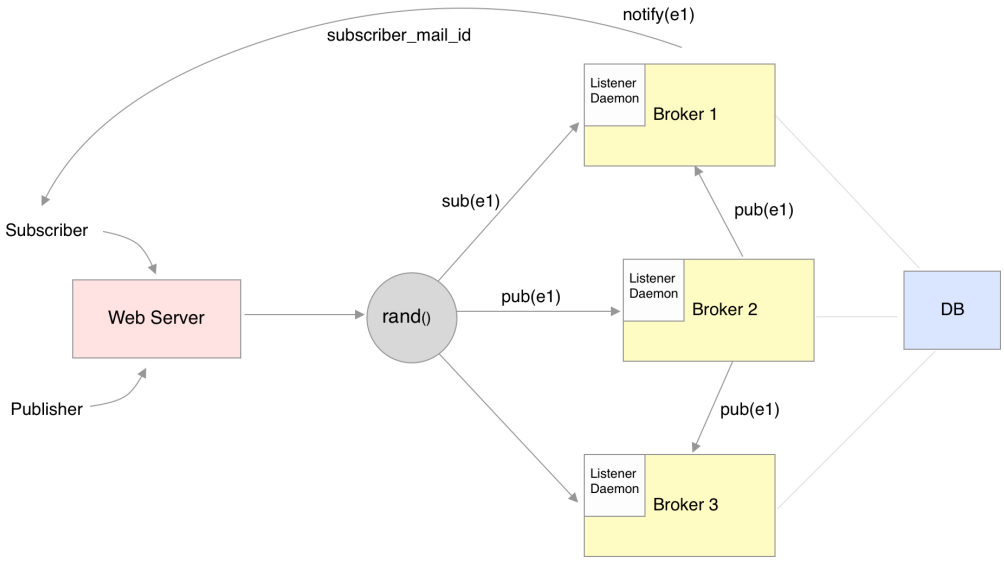


Fig 3.1 Component diagram for Phase 3 implementation

The Web server, the database, and the brokers are all separate Docker containers that communicate with each other. The routing technique implemented here is **filter-based routing**.

The server accepts subscribe and publish requests from users.

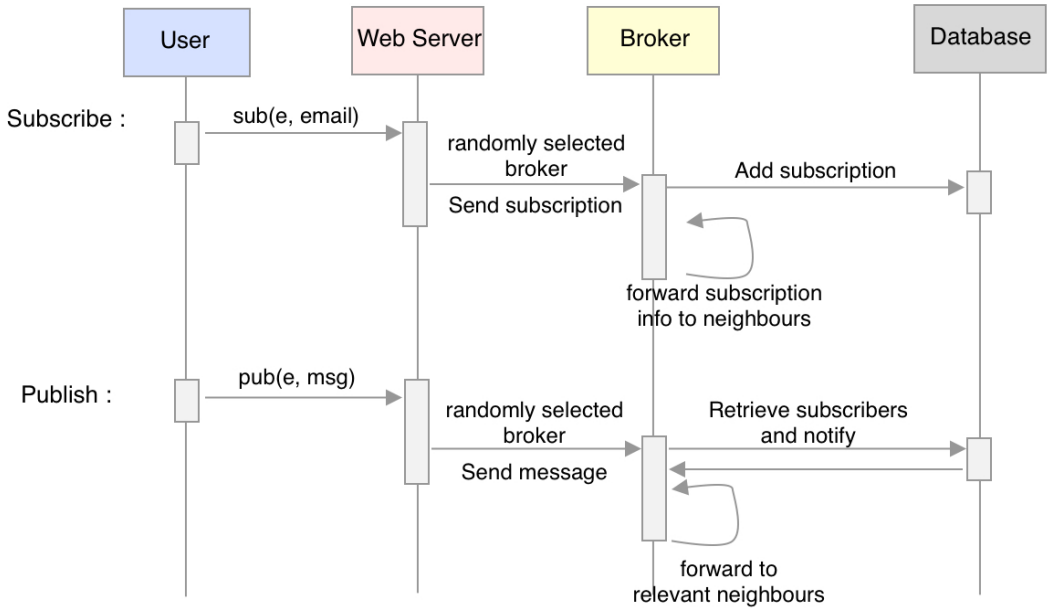


Fig 3.2 Sequence diagram for Phase 3 implementation

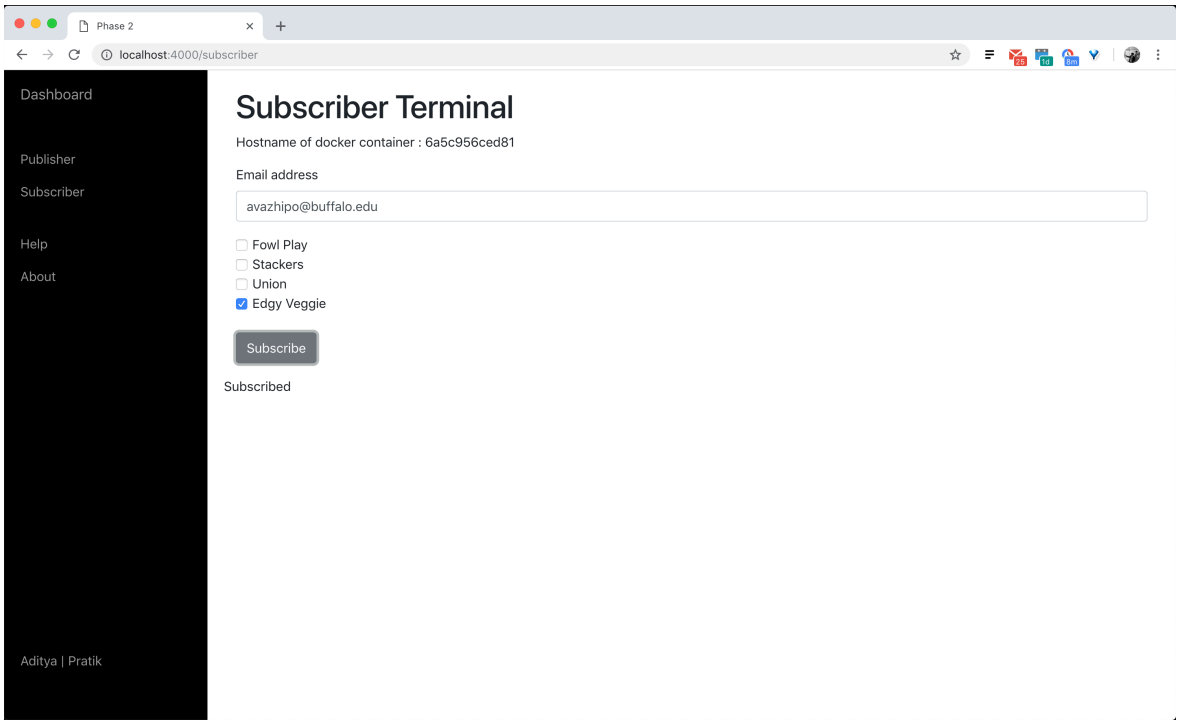
The server maintains list of brokers in network. Once it receives subscription/publish request, the server randomly selects a broker from network and sends a request to that broker via socket programming.

The broker has listener daemon that listens all messages on port and behaves accordingly. For each message, listener creates thread and new thread processes message. Every broker has table ("broker table") to maintain subscriber information. Every broker also maintains a list of neighbor brokers.

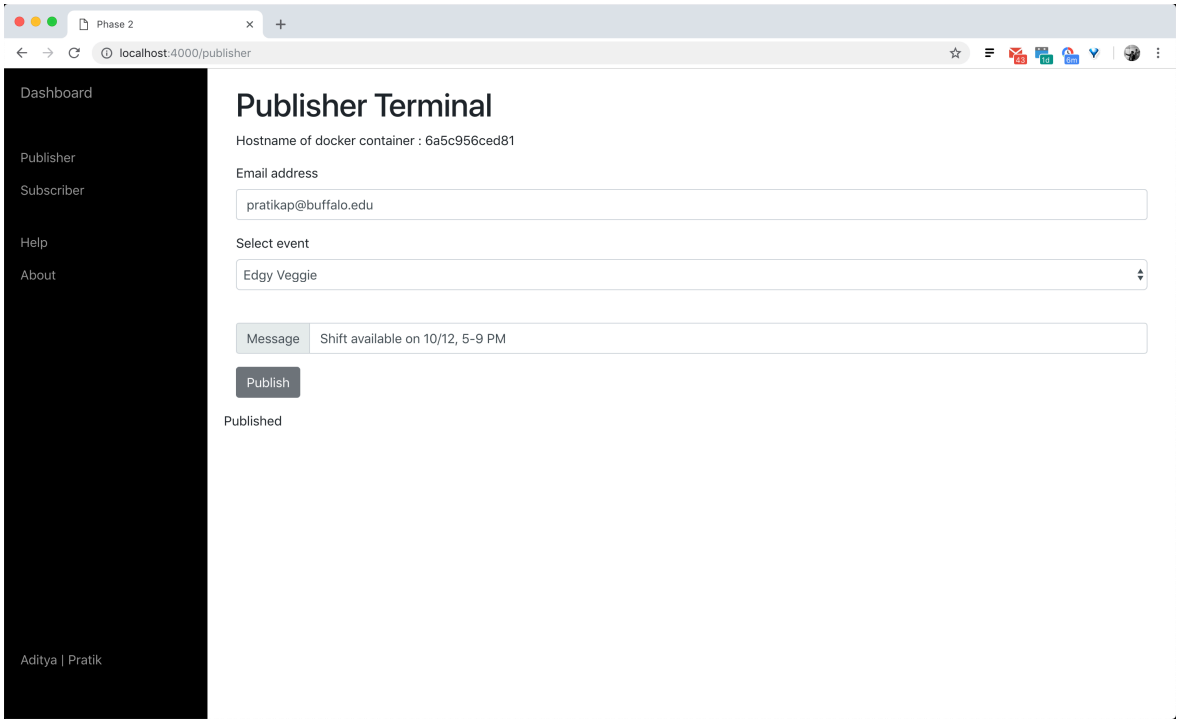
Subscribe():
The broker receives subscriber's mail id and event. If event exists in table, then subscriber gets added to subscriber list of that event. If event doesn't exist, broker adds event and subscriber mail id in table. Broker forwards event to neighbors.

Publish():
Broker receives message for a given event. Broker lists subscribers for event by querying "broker table". It notifies (**Notify()**) all subscriber by send message via email. Later, It forward given message to neighbors which have subscribers for that same event.

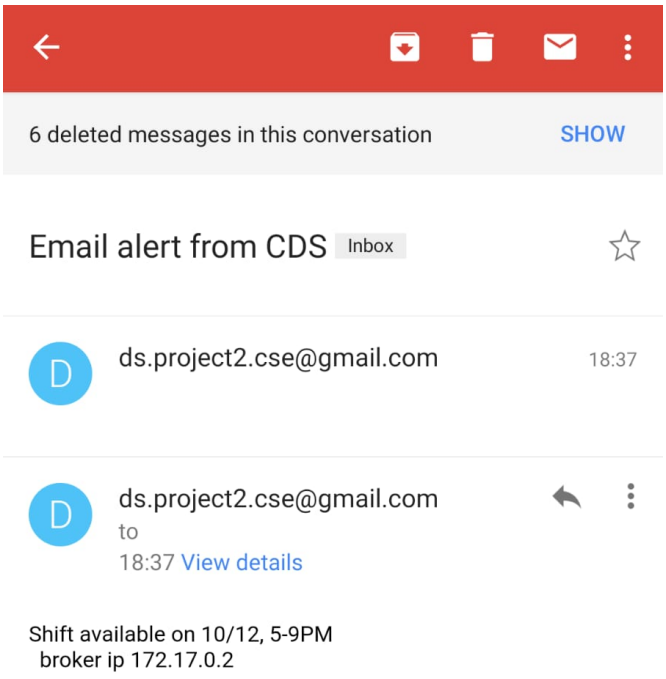
A user subscribes to the department “Edgy Veggie” below.



When a user subscribes as shown above, subscription request is sent to a broker selected at random, and broker stores subscription in database. Broker sends subscription information to the neighboring brokers.

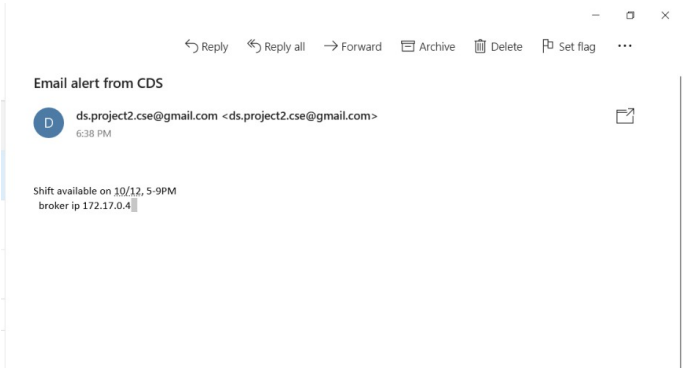


When a message is published, the notification sent to the user will be from the broker who has the subscription.



In the above notification, the IP of the broker is displayed as `172.12.0.2`

If a different user sends a subscribe request, it is possible that the user gets notified by a different broker since it's a distributed system.



An example of that case is shown above.

3 Instructions and Environment setup

The project runs on Docker containers, and the Dockerfiles, along with the rest of the instructions are detailed in the **`README`** file in the zip file submitted.

5 Future improvements

For going forward with this project in the future, certain improvements can be made.

First, on implementing a login functionality in the backend, this will open doors to new possibilities such as storing user preferences and notifying the users directly within the web application.

6 References

https://docs.docker.com/get-started/#images-and-containers

https://docs.docker.com/samples/library/postgres/