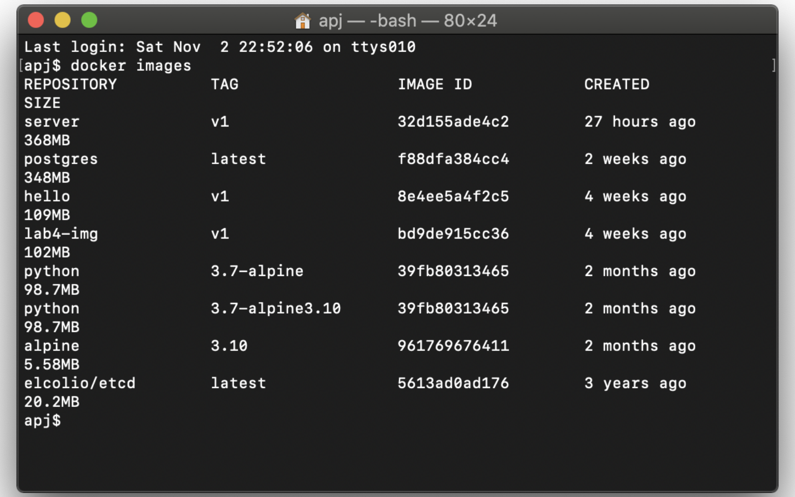
SERVICE SPANNING MULTIPLE CONTAINERS

- Arvind Ponnarassery Jayan
(aponnar1)

1 Images Pulled



```
Last login: Sat Nov  2 22:52:06 on ttys010
apj$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
server              v1                  32d155ade4c2        27 hours ago
368MB
postgres            latest              f88dfa384cc4        2 weeks ago
348MB
hello               v1                  8e4ee5a4f2c5        4 weeks ago
109MB
lab4-img            v1                  bd9de915cc36        4 weeks ago
102MB
python              3.7-alpine          39fb80313465        2 months ago
98.7MB
python              3.7-alpine3.10      39fb80313465        2 months ago
98.7MB
alpine              3.10                961769676411        2 months ago
5.58MB
elcolio/etcd        latest              5613ad0ad176        3 years ago
20.2MB
apj$
```

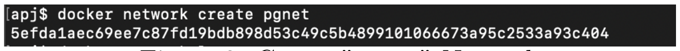
Figure 1: Docker Images

The images pulled are **postgres:latest** and **python:3.7-alpine**.
The command to pull the images are:

- `docker pull postgres`
- `docker pull python:alpine-3.7`

2 Docker Commands Issued

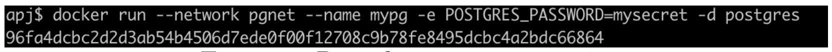
1. Creating a network named ”pgnet”.
   `docker network create pgnet`



```
apj$ docker network create pgnet
5efda1aec69ee7c87fd19bdb898d53c49c5b4899101066673a95c2533a93c404
```

Figure 2: Create ”pgnet” Network

2. Execute the postgres server in ”pgnet”.
   `docker run --network pgnet --name mypg -e POSTGRES_PASSWORD=mysecret -d postgres`

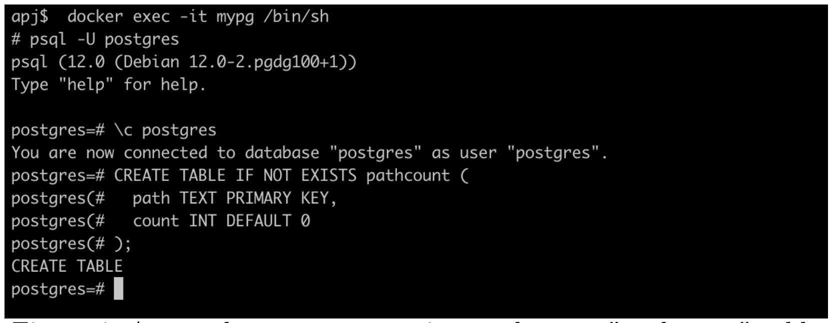


```
apj$ docker run --network pgnet --name mypg -e POSTGRES_PASSWORD=mysecret -d postgres
96fa4dcbc2d2d3ab54b4506d7ede0f00f12708c9b78fe8495dcbc4a2bdc66864
```

Figure 3: Run the postgres container

This runs the container in with the default user as "Postgres", default database as "Postgres" with the password as "mysecret" and the default port "5432".

3. Add a new table **pathcount** in the postgres database.
   The pathcount table has 2 columns "path" and "count".
   `docker exec -it mypg /bin/sh`

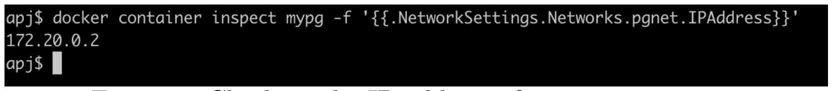


```
apj$  docker exec -it mypg /bin/sh
# psql -U postgres
psql (12.0 (Debian 12.0-2.pgdg100+1))
Type "help" for help.

postgres=# \c postgres
You are now connected to database "postgres" as user "postgres".
postgres=# CREATE TABLE IF NOT EXISTS pathcount (
postgres(#   path TEXT PRIMARY KEY,
postgres(#   count INT DEFAULT 0
postgres(# );
CREATE TABLE
postgres=#
```

Figure 4: Access the postgres container and create "pathcount" table

I have created the "pathcount" table in the "postgres" table from inside the container. This can be replicated by creating a `init.sql` file that contians the SQL query to create a table. The command is as follows, takes the password as an environment variable and runs the psql command to run the queries in the SQL file to update the database.
`docker run -it --rm --network pgnet -e PGPASSWORD=mysecret postgres psql -h 172.20.0.2 -U postgres < init.sql`

4. Check the IP address of the postgres container.
   `docker container inspect mypg -f '{{.NetworkSettings.Networks.pgnet.IPAddress}}'`



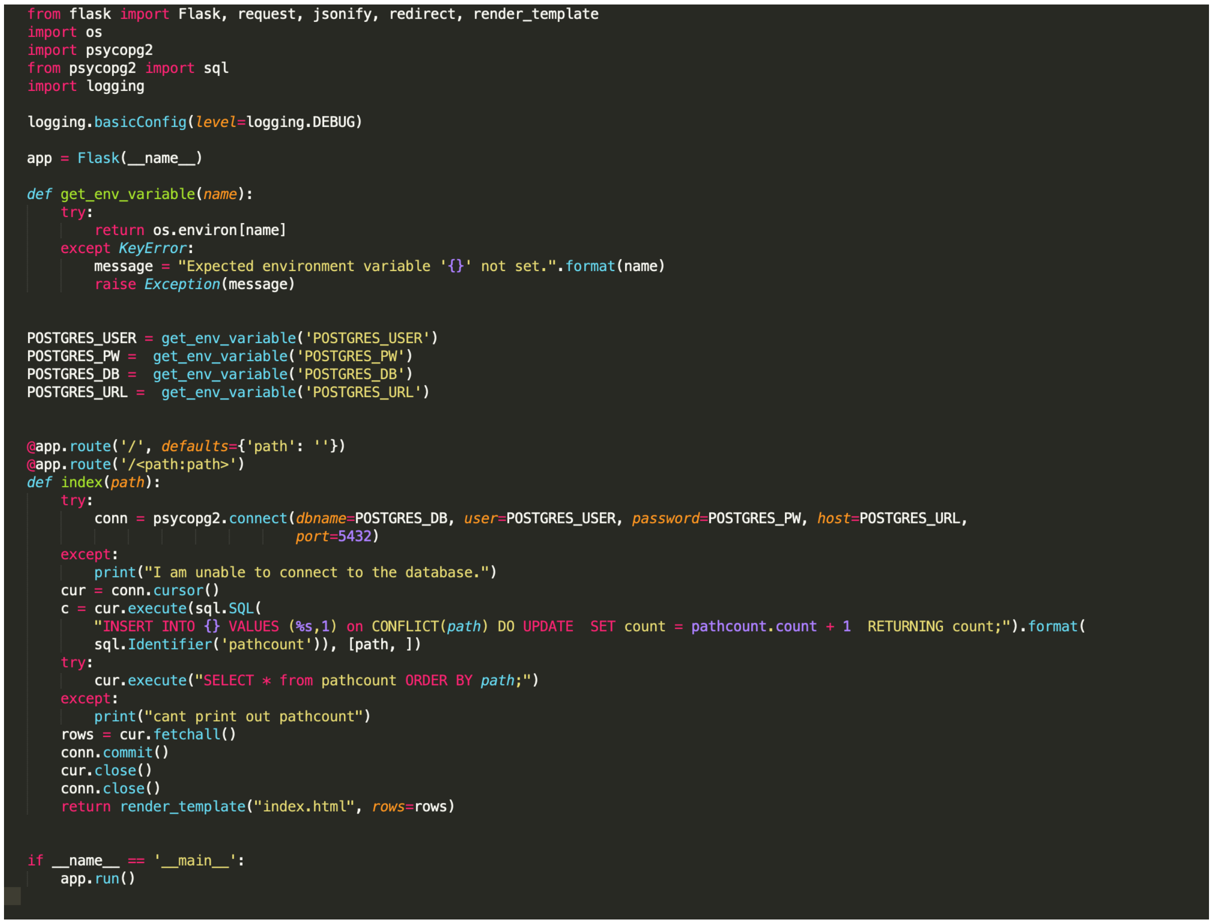
```
apj$ docker container inspect mypg -f '{{.NetworkSettings.Networks.pgnet.IPAddress}}'
172.20.0.2
apj$
```

Figure 5: Checking the IP address of postgres container

The IP address is found as `172.20.0.2`.

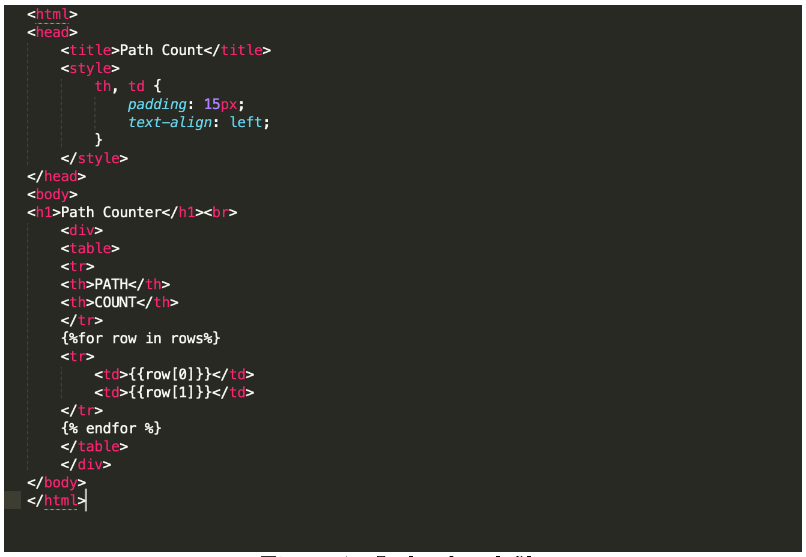
5. Construct the code for the flask server.

   The code takes the parameter for connecting to the postgres server through the environment variables. The environment variables passed are the postgers user: POSTGRES_USER, postgres password: POSTGRES_PW, postgres database: POSTGRES_DB and postgres URL: POSTGRES_URL. The server accepts the GET requests from any path, increases the count of the relative path to the postgres server if it doesn't already exist. The server renders "index.html" that prints the values in the "pathcount" table. The SQL queries used in the main.py file are to insert the new count for each path and also to select all the values from the "pathcount" table. The SQL queries are safe from SQL injections because it uses the prepared statements syntax.



```python
from flask import Flask, request, jsonify, redirect, render_template
import os
import psycopg2
from psycopg2 import sql
import logging

logging.basicConfig(level=logging.DEBUG)

app = Flask(__name__)

def get_env_variable(name):
    try:
        return os.environ[name]
    except KeyError:
        message = "Expected environment variable '{}' not set.".format(name)
        raise Exception(message)


POSTGRES_USER = get_env_variable('POSTGRES_USER')
POSTGRES_PW =  get_env_variable('POSTGRES_PW')
POSTGRES_DB =  get_env_variable('POSTGRES_DB')
POSTGRES_URL =  get_env_variable('POSTGRES_URL')


@app.route('/', defaults={'path': ''})
@app.route('/<path:path>')
def index(path):
    try:
        conn = psycopg2.connect(dbname=POSTGRES_DB, user=POSTGRES_USER, password=POSTGRES_PW, host=POSTGRES_URL,
                                port=5432)
    except:
        print("I am unable to connect to the database.")
    cur = conn.cursor()
    c = cur.execute(sql.SQL(
        "INSERT INTO {} VALUES (%s,1) on CONFLICT(path) DO UPDATE  SET count = pathcount.count + 1  RETURNING count;").format(
        sql.Identifier('pathcount')), [path, ])
    try:
        cur.execute("SELECT * from pathcount ORDER BY path;")
    except:
        print("cant print out pathcount")
    rows = cur.fetchall()
    conn.commit()
    cur.close()
    conn.close()
    return render_template("index.html", rows=rows)


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

Figure 6: Flask server code

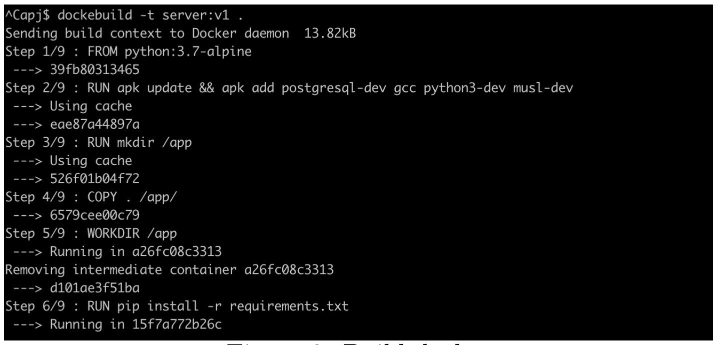


```html
<html>
<head>
    <title>Path Count</title>
    <style>
        th, td {
            padding: 15px;
            text-align: left;
        }
    </style>
</head>
<body>
<h1>Path Counter</h1><br>
    <div>
    <table>
    <tr>
    <th>PATH</th>
    <th>COUNT</th>
    </tr>
    {%for row in rows%}
    <tr>
        <td>{{row[0]}}</td>
        <td>{{row[1]}}</td>
    </tr>
    {% endfor %}
    </table>
    </div>
</body>
</html>
```

Figure 7: Index.html file

6. Creating and building the docker file.

   The dockerfile uses the base image "python:3.7-alpine". All the required files are copied and python flask is executed. Then the image "server:v1" is build.
   `docker build -t server:v1 .`

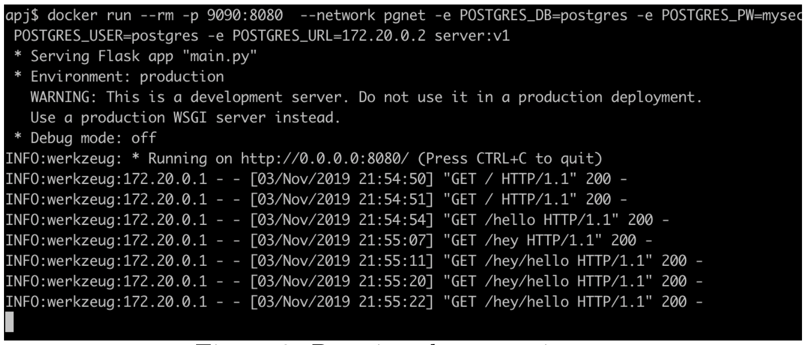


```
^Capj$ dockebuild -t server:v1 .
Sending build context to Docker daemon  13.82kB
Step 1/9 : FROM python:3.7-alpine
 ---> 39fb80313465
Step 2/9 : RUN apk update && apk add postgresql-dev gcc python3-dev musl-dev
 ---> Using cache
 ---> eae87a44897a
Step 3/9 : RUN mkdir /app
 ---> Using cache
 ---> 526f01b04f72
Step 4/9 : COPY . /app/
 ---> 6579cee00c79
Step 5/9 : WORKDIR /app
 ---> Running in a26fc08c3313
Removing intermediate container a26fc08c3313
 ---> d101ae3f51ba
Step 6/9 : RUN pip install -r requirements.txt
 ---> Running in 15f7a772b26c
```

Figure 8: Build docker

7. Running the server:v1 image.
   `docker run --rm -p 9090:8080 --network pgnet -e POSTGRES_DB=postgres -e POSTGRES_PW=mysecret -e POSTGRES_USER=postgres -e POSTGRES_URL=172.20.0.2 server:v1`

   The server:v1 image is run in the same network using the network flag.



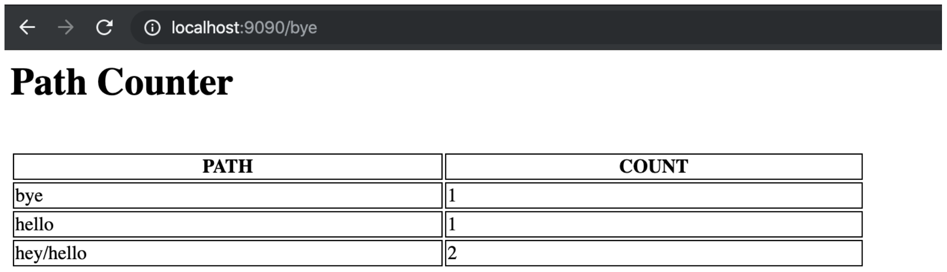
```
apj$ docker run --rm -p 9090:8080  --network pgnet -e POSTGRES_DB=postgres -e POSTGRES_PW=mysec
 POSTGRES_USER=postgres -e POSTGRES_URL=172.20.0.2 server:v1
 * Serving Flask app "main.py"
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
INFO:werkzeug: * Running on http://0.0.0.0:8080/ (Press CTRL+C to quit)
INFO:werkzeug:172.20.0.1 - - [03/Nov/2019 21:54:50] "GET / HTTP/1.1" 200 -
INFO:werkzeug:172.20.0.1 - - [03/Nov/2019 21:54:51] "GET / HTTP/1.1" 200 -
INFO:werkzeug:172.20.0.1 - - [03/Nov/2019 21:54:54] "GET /hello HTTP/1.1" 200 -
INFO:werkzeug:172.20.0.1 - - [03/Nov/2019 21:55:07] "GET /hey HTTP/1.1" 200 -
INFO:werkzeug:172.20.0.1 - - [03/Nov/2019 21:55:11] "GET /hey/hello HTTP/1.1" 200 -
INFO:werkzeug:172.20.0.1 - - [03/Nov/2019 21:55:20] "GET /hey/hello HTTP/1.1" 200 -
INFO:werkzeug:172.20.0.1 - - [03/Nov/2019 21:55:22] "GET /hey/hello HTTP/1.1" 200 -
```

Figure 9: Running the server image

   Since both the containers run in the same network they can interact and now provide the service together. The environment variables required for the container is passed using -e flag.

3 Results

The multiple containers communicate with each other and the service runs properly. We can see that the postgres container that is having the IP `172.20.0.2` in the network `pgnet` and the server container in the same network can communicate with the it. The environment variables passed to the server container is used to make a connection to the postgres server. Each GET request made to the server causes an updation of count in the database associated to the path name. The SQL injection attack is prevented by using the `%s` literal and {} in the SQL queries in the server code.



Path Counter

PATH	COUNT
bye	1
hello	1
hey/hello	2

Figure 10: Output

4 Security Questions

1. Why did we create a special network instead of exposing the host network?

   By creating a special networks for the containers to share, the host is isolated from the containers. If the any of the containers were to be compromised then only the network is under threat leaving the host to be safe.

2. Why didn't we use exposed ports everywhere (that they exist)?

   Exposing the ports unnecessarily can be a security risk, by exposing ports there will be a channel for unsecured communications and unrestricted traffic. If the container were to be compromised then the host can be easily accessed through the open ports.

3. What could happen if you didn't use SQL parameters, but relied on string formatting for setting the path in your queries?

   By using the SQL parameters instead of string formatting in the queries, the user prevents SQL injection attacks.

4. Why is that particularly important in this setup? What makes those parameters potentially dangerous?

   SQL parameters separates the data and the code unlike the string formatting which prevents the attacker to gain access to the database through SQL injection attacks. Since the application takes a user input through the URL, it is possible for the attacker to inject malicious code through the URL.

5. The bridge network we define only works on a single host. What would you have to do to make these containers talk to each other if they were running on different host machines?

   There are several options to do this:

   - using weave
   - setting up docker overlay network
   - using docker swarm
   - creating a docker network

   ref: stack-overflow

6. What parts of this did you wish were simpler? Which parts seemed unnecessarily difficult?

   I had a hard time connecting the flask server to postgres. This was unnecessarily difficult because I was trying to hardcode the values instead of passing the environment variables.