Richard Bradt
Nov. 1, 2019
Cloud Computing Security

Lab 6 - Create a Service Spanning Multiple Containers

Set-up:
For the set up of this lab, I followed the steps given in the lab 6 document.

1. Create user-defined Docker network (named pgnet)
   $ docker network create pgnet

```
(env) dix@dix-ubuntu:cloud-sec-2019-06$ docker network create pgnet
d92fe406dd2101605e98faabcec2e19100ae0c810d822d8ea5ffc345126a6574
(env) dix@dix-ubuntu:cloud-sec-2019-06$
```

2. Postgres Container
   a. Pull the postgres image from Docker repositories
      $ docker pull postgres

```
(env) dix@dix-ubuntu:cloud-sec-2019-06$ docker pull postgres
Using default tag: latest
latest: Pulling from library/postgres
Digest: sha256:a4a944788084a92bcaff6180833428f17cceb610e43c828b3a42345b33a608a7
Status: Image is up to date for postgres:latest
docker.io/library/postgres:latest
(env) dix@dix-ubuntu:cloud-sec-2019-06$
```

   b. Run container using postgres image. Connect it to your Docker network
      *$ docker run --network pgnet --name mypg *
      > -e POSTGRES_PASSWORD=password -d postgres

```
(env) dix@dix-ubuntu:cloud-sec-2019-06$ docker run \
> --network pgnet \
> --name mypg \
> -e POSTGRES_PASSWORD=password \
> -d \
> postgres
333d8b83d2cb8a8936ebff3b95bc5f5d7959ced27d7aefc1aa00f1e990809e64
(env) dix@dix-ubuntu:cloud-sec-2019-06$
```

3. Init Container
   a. Find IP address for postgres database container
      *$ docker container inspect mypg *
      > -f '{{.NetworkSettings.Networks.pgnet.IPAddress}}'

```
(env) dix@dix-ubuntu:cloud-sec-2019-06$ docker container inspect mypg \
>   -f '{{.NetworkSettings.Networks.pgnet.IPAddress}}'
172.18.0.2
(env) dix@dix-ubuntu:cloud-sec-2019-06$
```

b. Write *init.sql* to build table in database

```
(env) dix@dix-ubuntu:cloud-sec-2019-06$ cat init.sql
CREATE TABLE IF NOT EXISTS pathcount (
  path TEXT PRIMARY KEY,
  count INT DEFAULT 0
);
(env) dix@dix-ubuntu:cloud-sec-2019-06$
```

c. Run init container to initialize your database for use
*$ docker run -i --rm --network pgnet -e PGPASSWORD=password postgres *
> psql -h 172.18.0.2 -U postgres < init.sql

```
(env) dix@dix-ubuntu:cloud-sec-2019-06$ docker run \
> -i \
> --rm \
> --network pgnet \
> -e PGPASSWORD=password \
> postgres \
> psql -h 172.18.0.2 -U postgres < init.sql
CREATE TABLE
(env) dix@dix-ubuntu:cloud-sec-2019-06$
```

d. I did run into one error when following the lab document. When trying to run the init container in interactive mode using --tty (or -it), I would get an 'Input device is not a TTY' error. The only work around I could find without digging too deep was to just omit --tty flag. Screenshot of error below:

```
(env) dix@dix-ubuntu:cloud-sec-2019-06$ docker run \
> -it \
> --rm \
> --network pgnet \
> -e PGPASSWORD=password \
> postgres \
> psql -h 172.18.0.2 -U postgres < init.sql
the input device is not a TTY
(env) dix@dix-ubuntu:cloud-sec-2019-06$
```

Service Container:

1. For my service container, the package requirements include the following python modules; installed using *$ pip install [package]*:
    1. Flask for creating my web application
    2. Python-dotenv for handling virtual environment variables
    3. Psycopg2 to work with postgresql databases.

    Dependencies for psycopg2 included the postgresql package. The current version in the Ubuntu 18.04 apt repositories is 10. With *$ sudo apt-get install*, I installed the two dependencies:
    1. postgresql
    2. postgresql-server-dev-10

2. I wrote my web service application using Python and flask. I have all paths going to the default route which calls functions that will query and update the postgres database and display all paths onto an HTML page.
    a. As required, I defined all configuration variables in ENVIRONMENT VARIABLES. To use python-dotenv, I defined those variables in a '.env' file in the project directory. That file is then loaded in 'main.py'. Here is that file:

```
(env) dix@dix-ubuntu:cloud-sec-2019-06$ cat .env
# Environment variables for Docker lab 06
POSTGRES_USER="postgres"
POSTGRES_DATABASE="postgres"
POSTGRES_PASSWORD="password"
POSTGRES_HOST="172.18.0.2"
(env) dix@dix-ubuntu:cloud-sec-2019-06$
```

    b. The primary code files include:
        i. Main.py : web application file
        ii. Index.html : HTML for displaying path counts
        iii. Style.css : styling for the web page

        All of which are located in my class github: https://github.com/richardbradt/cloud_computing/tree/master/cloud-sec-2019-06

3. I then built my container using the following command and Dockerfile:
$ docker build -t pathcount .

```
FROM python
RUN mkdir /app
COPY . /app/
WORKDIR /app
RUN pip install Flask \
  python-dotenv \
  psycopg2
EXPOSE 8080
CMD ["python", "main.py"]
```

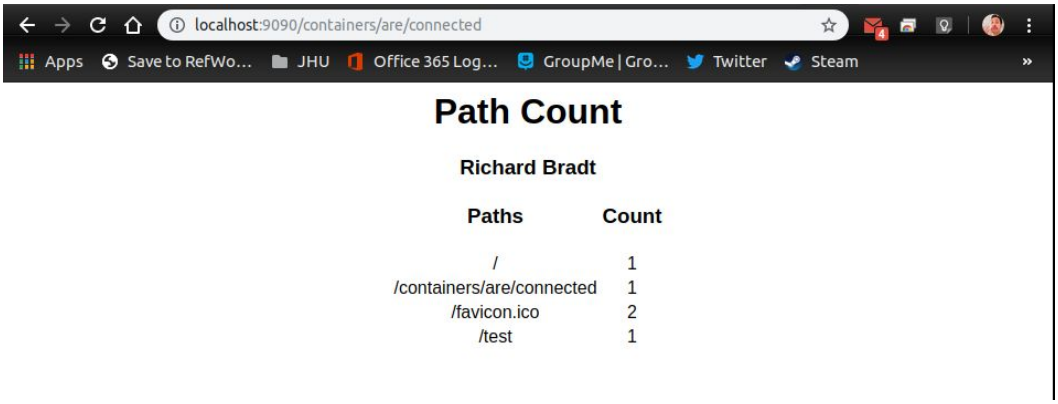
Dockerfile for Service Container

4. I then ran my service container with the following command, exposing port 8080 to host port 9090.

   *$ docker run -it --name=pathcount_cont --network pgnet *
   > -p 9090:8080 pathcount

```
(env) dix@dix-ubuntu:cloud-sec-2019-06$ docker run \
> -it \
> --name=pathcount_cont \
> --network pgnet \
> -p 9090:8080 \
> pathcount
 * Serving Flask app "main" (lazy loading)
 * 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
 * Running on http://0.0.0.0:8080/ (Press CTRL+C to quit)
```

5. Finally, going to http://localhost:9090 on my host machine's browser produces this page, and by manually typing various paths in the address bar, the service continues to update and display the database contents.



Questions:

1. Why did we create a special network instead of exposing the host network?
   This further isolates the containers from elements on the host network.

2. Why didn't we use exposed ports everywhere (that they exist)?
   By not exposing ports on the postgres container, we're isolating the database to our user-defined docker bridge network, protecting it from outside influences.

3. What could happen if you didn't use SQL parameters, but relied on string formatting for setting the path in your queries?
   We would be subject to SQL Injection attacks.

4. Why is that particularly important in this setup? What makes those parameters potentially dangerous?
   Parameters could still be dangerous if they're not sanitized.

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*?
   Docker would have to use the host network as opposed to using a user-defined bridge network like we did in this lab.

6. What parts of this did you wish were simpler? Which parts seemed unnecessarily difficult?
   The most difficult portions of this lab falls to whichever part is most unfamiliar. For me, it was wrapping my head around the 'docker-isms'; most notably, understanding docker networks and docker postgres. Finding usable python modules for postgresql was just a Google search. Luckily, I found very useful articles and tutorials. But, sifting through docker documentation can be daunting and somewhat lacking.

Main.py

```python
"""App that produces a path count."""
"""Default route will increment count for each path and store data in database."""

from flask import Flask, request, render_template
from os.path import join, dirname
from dotenv import load_dotenv
import os, psycopg2

"""Import Environment Variables for DB"""
dotenv_path = join(dirname(__file__), '.env')
load_dotenv(dotenv_path)

user = os.getenv('POSTGRES_USER')
db = os.getenv('POSTGRES_DATABASE')
secret = os.getenv('POSTGRES_PASSWORD')
host = os.getenv('POSTGRES_HOST')

app = Flask(__name__)

"""Default route will get path and check for database entry"""
"""If in database, increment count. If not, set count to 1 and add to db"""
@app.route('/', defaults={'path': '/'}, methods=['GET'])
@app.route('/<path:path>', methods=['GET'])
def root(path):
    c_path = request.path
    print('CURRENT PATH: {}'.format(c_path))
    count_path(c_path)
    return display_paths()

"""Check database for path and count"""
"""Connect to Postgresql database and query current path"""
def count_path(path):
    sql = """INSERT INTO pathcount (path, count)
                VALUES (%s, 1)
             ON CONFLICT (path) DO UPDATE
                SET count = pathcount.count + 1
             RETURNING count;"""
    conn = None

    try:
        conn = psycopg2.connect(host=host ,database=db, user=user, password=secret)
        cur = conn.cursor()
        cur.execute(sql, (path,))
        conn.commit()
        cur.close()
    except (Exception, psycopg2.DatabaseError) as error:
        print(error)
    finally:
        if conn is not None:
            conn.close()
```

```python

"""Display current database contents"""
"""Query Postgresql database and build JSON of all pathcounts"""
def display_paths():
    sql = """SELECT path, count FROM pathcount ORDER BY path"""
    conn = None

    try:
        conn = psycopg2.connect(host=host ,database=db, user=user, password=secret)
        cur = conn.cursor()
        cur.execute(sql)
        entities = cur.fetchall()
        path_json = build_json(entities)
        conn.commit()
        cur.close()
    except (Exception, psycopg2.DatabaseError) as error:
        print(error)
    finally:
        if conn is not None:
            conn.close()

    return render_template('index.html', data=path_json)

"""Builds JSON to send to HTML/JS.  Takes entity list from DB"""
def build_json(path_list):
    new_json = '['
    counter = 1
    for ent in path_list:
        print("CURRENT KEY: {}".format(ent[0]))
        print("CURRENT KEY VALUE: {}".format(ent[1]))
        if counter==len(path_list):
            new_json+='{"path":"%s","count":"%d"}]' % (ent[0], ent[1])
            break
        else:
            new_json+='{"path":"%s","count":"%d"},' % (ent[0], ent[1])
            counter+=1

    return new_json

if __name__ == '__main__':
    app.run(host='0.0.0.0', port='8080')
```

Index.html

```html

<html>
<head>
<title>Richard Bradt - Cloud Sec 2019</title>
<link type="text/css" rel="stylesheet" href="{{ url_for('static', filename='style.css') }}"></link>
</head>
<body>
<div id="main" class="mainDiv">
  <h1>Path Count</h1>
  <h3>Richard Bradt</h3>
  <div class="container" id="table_div">
  <table class="container" id="path_table">
      <script>
        var paths = JSON.parse('{{ data | safe }}');
        let html = '<tr><th><h3>Paths</h3></th><th><h3>Count</h3></th></tr>';
        for (let x of paths) {
          html += '<tr><td>'+x.path+'</td><td>'+x.count+'</td></tr>';
        }
        document.getElementById('path_table').innerHTML = html;
      </script>
  </table>
  </div>
</div>
</body>
</html>

```

Style.css

```css
.mainDiv {
  margin-left: auto;
  margin-right: auto;
  max-width: 1000px;
  float: none;
  font-family: "helvetica", sans-serif;
  text-align: center;
}
.container {
  text-align: center;
}
.container table {
  margin: 0 auto;
}

```