Exp19:-

Create two applications in two different docker containers. Push those applications and run to show the communications between two dockers.

Backend Application (Flask API)

1. **Create a directory for your project:**

   mkdir docker_communication_demo

   cd docker_communication_demo

2. **Create a file named `app.py` for the Flask API:**

```
# app.py
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/api/data')

def get_data():

   return jsonify({"message": "Hello from the backend!"})

if __name__ == '__main__':

   app.run(debug=True, host='0.0.0.0', port=5000)
```

3. **Create a `Dockerfile` for the backend:**

```
# Dockerfile

FROM python:3.8

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY . .
```

```
CMD ["python", "app.py"]
```

4. Create a `requirements.txt` file:

```
Flask>=2.1.5

Werkzeug>=2.0.2
```

5.Build and run the backend Docker container:

docker build -t backend-app .

docker run -p 5000:5000 backend-app

Frontend Application (Flask Web App)

1. **Create a directory for the frontend in** same directory in which backend directory are there:

   mkdir frontend-app

   cd frontend-app

2. **Create a file named `app.py` for the Flask web app:**

```
# app.py

from flask import Flask, render_template

import requests

app = Flask(__name__)

backend_url = "http://backend:5000"  # This is the Docker service name

@app.route('/')

def home():

   response = requests.get(f"{backend_url}/api/data")
```

```
    data = response.json()

    return render_template('index.html', message=data['message'])

if __name__ == '__main__':

    app.run(debug=True, host='0.0.0.0', port=5001)
```

3. **Create a `Dockerfile` for the frontend:**

```
# Dockerfile

FROM python:3.8

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["python", "app.py"]
```

4. **Create a `requirements.txt` file:**

```
Flask>=2.1.5

requests==2.26.0
```

5.Create a directory named `templates` and add a file named `index.html`:

```
<!-- templates/index.html -->

<!DOCTYPE html>

<html lang="en">

<head>
```

```
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Docker Communication Demo</title>
</head>
<body>
    <h1>{{ message }}</h1>
</body>
</html>
```

6. Build and run the frontend Docker container:

docker build -t frontend-app .

docker run -p 5001:5001 --link backend frontend-app

Docker-compose File

Create a docker-compose.yml file in your project directory along with frontend and backed directory **with the following content:**

```
version: '3'
services:
 backend:
   build:
     context: ./docker_communication_demo
   ports:
     - "5000:5000"
```

```
 frontend:
   build:
     context: ./frontend-app
   ports:
     - "5001:5001"
   depends_on:
     - backend
```

Run command:-

docker-compose up

Exp :- 37

Create a docker image of simple login form using Flask on port 7000.

Step 1:-

Create a file named `app.py` with the following code for a simple Flask login form:

```
# app.py

from flask import Flask, render_template, request



app = Flask(__name__)



@app.route('/')

def index():

   return render_template('login.html')
```

```
@app.route('/login', methods=['POST'])
def login():
    username = request.form['username']
    password = request.form['password']

    # Add your login logic here (for simplicity, we'll just print the credentials)
    print(f"Username: {username}, Password: {password}")

    return 'Login successful!'

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=7000)
```

Step 2: Create HTML Template Create a folder named `templates` and inside it, create a file named `login.html` with the following content:

```
<!-- templates/login.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Login</title>
</head>
<body>
    <h2>Login Form</h2>
    <form action="/login" method="post">
        <label for="username">Username:</label>
        <input type="text" id="username" name="username" required><br>

        <label for="password">Password:</label>
```

```
        <input type="password" id="password" name="password" required><br>

        <input type="submit" value="Login">
    </form>
</body>
</html>
```

Step 3: Create a Dockerfile Create a file named `Dockerfile` in the same directory as your `app.py` with the following content:

```
# Dockerfile
FROM python:3.8

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 7000

CMD ["python", "app.py"]
```

Step 4: Create a requirements.txt file in the same directory as your `app.py` Create a file named `requirements.txt` with the following content:

Flask>=2.0.1
Werkzeug>=2.0.1

Step 5: Build the Docker Image Open a terminal, navigate to the directory containing your `Dockerfile`, `app.py`, `templates`, and `requirements.txt` files, and run the following command to build the Docker image:

docker build -t flask-login-app .

Step 6: Run the Docker Container After building the image, run the Docker container with the following command:

docker run -p 7000:7000 flask-login-app

Now, you should be able to access the simple login form at `http://localhost:7000` in your web browser.

Exp :- 38

Create a docker image of simple login form using django on port 6000.

Step 1: Create a Django Project

```
# Create a directory for your project
mkdir simple_login_django

# Navigate to the project directory
cd simple_login_django

# Create a virtual environment (optional but recommended)
python3 -m venv venv
source venv/bin/activate  # On Windows, use `venv\Scripts\activate`

# Install Django
pip install django

# Create a Django project
django-admin startproject simplelogin
```

Step 2: Create a Django App

 # Navigate to the project directory
cd simplelogin

 # Create a Django app
python manage.py startapp loginapp

Step 3: Update `loginapp/views.py`

Create a file named `views.py` inside the `loginapp` directory with the following content:

```
from django.shortcuts import render
from django.http import HttpResponse

def login(request):
    return render(request, 'loginapp/login.html', {})

def success(request):
    return HttpResponse("Successful Login!")
```

Step 4: Create `loginapp/templates/loginapp/login.html`

Create a `templates` directory inside the `loginapp` directory, and inside it, create a file named `login.html` with the following content:

```
<!DOCTYPE html>
<html>
<head>
    <title>Login Page</title>
</head>
<body>
    <h2>Login</h2>
    <form action="/success/" method="post">
        {% csrf_token %}
        <label for="username">Username:</label>
        <input type="text" name="username" id="username" required>
        <br>
```

```
        <label for="password">Password:</label>
        <input type="password" name="password" id="password" required>
        <br>
        <input type="submit" value="Login">
    </form>
</body>
</html>
```

Step 5: Update `loginapp/urls.py`

Create a file named `urls.py` inside the `loginapp` directory with the following content:

```
from django.urls import path

from . import views



urlpatterns = [

    path('login/', views.login, name='login'),

    path('success/', views.success, name='success'),

]
```

Step 6: Update `simplelogin/urls.py`

Update the `urls.py` file inside the `simplelogin` directory with the following content:

```
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('loginapp.urls')),
]
```

Update in the setting.py file with following content:-

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR / 'loginapp' / 'templates'],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

Step 8: Dockerize the Application

Create a `Dockerfile` in the project root in the level of manage.py file with the following content:

```
# Use an official Python runtime as a parent image
FROM python:3.8-slim

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . /app

# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Make port 6000 available to the world outside this container
EXPOSE 6000

# Define environment variable
ENV NAME simplelogin
```

```
# Run app.py when the container launches
CMD ["python", "manage.py", "runserver", "0.0.0.0:6000"]
```

Step 9: Create a `requirements.txt` file

Create a file named `requirements.txt` in the project root in the level of manage.py file with the following content:

```
Django==3.2.5
```

Step 10: Build and Run the Docker Image

 # Build the Docker image
docker build -t simple-login-django .

 # Run the Docker container
docker run -p 8000:6000 simple-login-django

Open your web browser and navigate to http://localhost:8000/login/. You should see the login page. Enter any username and password, and you'll be redirected to the success page.

Verify Directory Structure:

Ensure that your directory structure is correct. The `templates` directory should be inside the `loginapp` directory.

```
simple_login_django/
├── loginapp/
│   ├── templates/
│   │   └── loginapp/
│   │       └── login.html
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── migrations/
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── simplelogin/
│   ├── __init__.py
```

```
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
├── venv/
└── Dockerfile
```

Exp:-39

Create a container with ngnix web server and create one more container with mysql.

1. Create the Nginx Container:

Step 1: Create an Nginx Dockerfile

Create a file named `Dockerfile.nginx` with the following content:

```
# Dockerfile.nginx

FROM nginx:latest

# Copy custom Nginx configuration

COPY nginx.conf /etc/nginx/nginx.conf

# Expose port 80

EXPOSE 80

# Start Nginx

CMD ["nginx", "-g", "daemon off;"]
```

Step 2: Create an Nginx Configuration File

Create a file named `nginx.conf` with your custom Nginx configuration. For simplicity, you can start with a basic configuration:

```
# nginx.conf

user  nginx;

worker_processes  1;

error_log  /var/log/nginx/error.log warn;

pid        /var/run/nginx.pid;

events {

   worker_connections  1024;

}

http {

   include       /etc/nginx/mime.types;

   default_type  application/octet-stream;

   log_format  main '$remote_addr - $remote_user [$time_local] "$request"'

                    '$status $body_bytes_sent "$http_referer" '

                    '"$http_user_agent" "$http_x_forwarded_for"';

   access_log  /var/log/nginx/access.log  main;

   sendfile        on;

   keepalive_timeout  65;

   include /etc/nginx/conf.d/*.conf;

}
```

Step 3: Build and Run the Nginx Container

docker build -t nginx-container -f Dockerfile.nginx .
docker run -d -p 80:80 --name nginx-container nginx-container

2. Create the MySQL Container:

Step 1: Create a MySQL Dockerfile

Create a file named `Dockerfile.mysql` with the following content:

```
# Dockerfile.mysql

FROM mysql:latest

# Set environment variables

ENV MYSQL_ROOT_PASSWORD=root_password \

   MYSQL_DATABASE=my_database \

   MYSQL_USER=my_user \

   MYSQL_PASSWORD=my_password

# Expose port 3306

EXPOSE 3306

# Start MySQL

CMD ["mysqld"]
```

Step 2: Build and Run the MySQL Container

docker build -t mysql-container -f Dockerfile.mysql .
docker run -d -p 3306:3306 --name mysql-container mysql-container

Verify Containers

You can access the Nginx welcome page by visiting `http://localhost` in your web browser.

View Running Containers

docker ps

docker inspect mysql-container

docker exec -it mysql-container bash

mysql -u root -p
password= `root_password`

Exp:-40
Create a simple web form to insert the records in mysql data base.

Step 1: Create a New Directory

Create a new directory for your project. For example:

mkdir lamp-web-form
cd lamp-web-form

Step 2: Create Dockerfile for PHP and Apache

Create a file named `Dockerfile` in the project directory:

```
# Dockerfile

FROM php:7.4-apache

# Install MySQLi extension

RUN docker-php-ext-install mysqli

COPY src/ /var/www/html/

EXPOSE 80
```

Step 3: Create the Source Directory

Create a directory named `src` in the project directory:

mkdir src

Step 4: Create PHP Script with Web Form

Inside the `src` directory, create a file named `index.php` with the following content:

```
<!-- src/index.php -->

<!DOCTYPE html>

<html>

<head>

   <title>Simple PHP Web Form</title>

</head>

<body>

   <h1>Web Form to Insert Records</h1>

   <form action="insert.php" method="post">

       <label for="name">Name:</label>

       <input type="text" id="name" name="name" required><br>



       <label for="email">Email:</label>

       <input type="email" id="email" name="email" required><br>

       <input type="submit" value="Submit">

   </form>
```

```
</body>

</html>
```

Inside the `src` directory, create a file named `insert.php` with the following content:

```
<!-- src/insert.php -->

<?php

$host = 'mysql';

$user = 'my_user';

$password = 'my_password';

$database = 'my_database';

$conn = new mysqli($host, $user, $password, $database);

if ($conn->connect_error) {

   die("Connection failed: " . $conn->connect_error);

}

if ($_SERVER["REQUEST_METHOD"] == "POST") {

   $name = $_POST["name"];

   $email = $_POST["email"];

   $sql = "INSERT INTO users (name, email) VALUES ('$name', '$email')";

   if ($conn->query($sql) === TRUE) {

       echo "<p>New record inserted successfully</p>";

   } else {

       echo "Error: " . $sql . "<br>" . $conn->error;

   }
```

```
}

$conn->close();

?>
```

Step 5: Create Dockerfile for MySQL

Create a file named `Dockerfile.mysql` in the project directory:

```
# Dockerfile.mysql

FROM mysql:latest

# Set environment variables

ENV MYSQL_ROOT_PASSWORD=root_password

ENV MYSQL_DATABASE=my_database

ENV MYSQL_USER=my_user

ENV MYSQL_PASSWORD=my_password

# Copy initialization SQL script

COPY init.sql /docker-entrypoint-initdb.d/

# Expose the MySQL port

EXPOSE 3306
```

Create a file named `init.sql` in the same directory as your `Dockerfile.mysql` with the following content:

```
-- init.sql

CREATE DATABASE IF NOT EXISTS my_database;

USE my_database;

CREATE TABLE IF NOT EXISTS users (
```

```
    id INT AUTO_INCREMENT PRIMARY KEY,

    name VARCHAR(255) NOT NULL,

    email VARCHAR(255) NOT NULL

);
```

Step 6: Create Docker Compose File

Create a file named `docker-compose.yml` in the project directory:

```
version: '3'

services:
 web:
   build:
     context: .
     dockerfile: Dockerfile
   ports:
     - "8080:80"
   depends_on:
     - mysql
   environment:
     MYSQL_HOST: mysql
     MYSQL_USER: root
     MYSQL_PASSWORD: root_password
     MYSQL_DATABASE: my_database

 mysql:
   build:
     context: .
     dockerfile: Dockerfile.mysql
   ports:
     - "3306:3306"

 phpmyadmin:
   image: phpmyadmin/phpmyadmin
   ports:
     - "8081:80"
```

```
  environment:
    PMA_HOST: mysql
    PMA_USER: root
    PMA_PASSWORD: root_password
```

Step 7: Build and Run the Docker Containers

Run the following commands to build and run the Docker containers:

docker-compose build

docker-compose up -d

Visit `http://localhost:8080` in your web browser to access the web form

Visit http://localhost:8081 in your web browser to access the phpMyAdmin to see your data is inserted or not

Exp:-42

Write a Docker File to pull the Ubuntu with open jdk and write any java application.

Step 1: Create the Java Application

Create a directory for your Java application and add a file named `MyApp.java`:

```
// MyApp.java

public class MyApp {

   public static void main(String[] args) {

        System.out.println("Hello, Docker!");

   }
```

```
}
```

Step 2: Create Dockerfile

In the same directory as your Java application, create a file named `Dockerfile`:

```
# Use the official Ubuntu base image

FROM ubuntu:latest

# Install OpenJDK

RUN apt-get update && \

   apt-get install -y openjdk-11-jdk

# Set the working directory

WORKDIR /app

# Copy the Java application into the container

COPY MyApp.java .

# Compile the Java application

RUN javac MyApp.java

# Define the command to run the application

CMD ["java", "MyApp"]
```

Step 3: Build the Docker Image

docker build -t my-java-app .

Step 4: Run the Docker Container

docker run my-java-app

43. Run a LAMP Stack Container at port 8080 and host media wiki site on native machine.

1] create docker-compose file:-

```
# MediaWiki with MySQL

version: '3'

services:

 mediawiki:

   image: mediawiki:1.38

   restart: always

   networks:

     - docker_network

   ports:

     - 8080:80

   # volumes:

   #   - ./LocalSettings.php:/var/www/html/LocalSettings.php

# After initial setup, download LocalSettings.php to the same directory as

# this yaml and uncomment the following line and use compose to restart

# the mediawiki service

 database:

   image: mysql:8.0.29

   restart: always

   networks:

     - docker_network
```

```
  environment:
    MYSQL_DATABASE: wiki_db
    MYSQL_ROOT_PASSWORD: root
    MYSQL_USER: wikimedia
    MYSQL_PASSWORD: wikimedia
  volumes:
    - /var/lib/mysql
 # phpmyadmin
 phpmyadmin:
  depends_on:
    - database
  image: phpmyadmin/phpmyadmin
  restart: always
  ports:
    - '8000:80'
  environment:
    PMA_HOST: database
    MYSQL_ROOT_PASSWORD: root
    UPLOAD_LIMIT: 64M
  networks:
    - docker_network
networks:
 docker_network:
```

```
  driver: bridge
```

2. Follow the installation instruction

While database configuration

Change database host:-

Localhost to database

Change database name:-

wiki_db

Password:-

root