😊 Docker Compose

Docker Compose is a tool for defining and running multi-container Docker applications. It allows you to define the services, networks, and volumes required for your application in a YAML file called docker-compose.yml. With Docker Compose, you can manage the entire lifecycle of a multi-container Docker application with simple commands.

Here's a brief overview of what Docker Compose can do:

1. **Define Services:** You can specify the services that make up your application in the docker-compose.yml file. Each service is defined as a separate container, and you can configure various parameters such as the image to use, environment variables, ports to expose, volumes to mount, and more.
2. **Manage Dependencies:** Docker Compose automatically handles dependencies between services. For example, if one service depends on another, Docker Compose ensures that the dependent service is started first.
3. **Networking:** Docker Compose automatically creates a default network for your application, allowing containers to communicate with each other using service names as hostnames. You can also define custom networks for more complex network topologies.
4. **Volume Management:** Docker Compose allows you to define named volumes or bind mounts for your containers, making it easy to persist data across container restarts or share data between containers.
5. **Lifecycle Management:** With Docker Compose, you can easily start, stop, and restart your entire application or individual services with simple commands. You can also scale services up or down to adjust to changing workloads.

😄 Use cases of Docker Compose:

Docker Compose is incredibly versatile and can be used in various scenarios where you need to manage multi-container Docker applications. Here are some common use cases:

1. **Development Environments:** Docker Compose is widely used for setting up development environments. Developers can define a docker-compose.yml file that specifies all the services required for their application, including databases, web servers, caching systems, and more. This allows developers to quickly spin up a consistent development environment on their local machine, regardless of the underlying operating system.
2. **Testing Environments:** Docker Compose is also valuable for creating testing environments. Testers can define a docker-compose.yml file that sets up the necessary infrastructure for running automated tests, such as test databases, mock services, and other dependencies. This ensures that tests are run in an isolated and reproducible environment, leading to more reliable results.
3. **Continuous Integration/Continuous Deployment (CI/CD):** Docker Compose is often used in CI/CD pipelines to deploy applications to various environments, such as staging

and production. CI/CD tools can use Docker Compose to define the infrastructure required for running tests, building Docker images, deploying containers, and orchestrating the entire deployment process.

4. **Microservices Architecture:** In a microservices architecture, applications are composed of multiple loosely coupled services that communicate with each other over the network. Docker Compose simplifies the management of microservices-based applications by allowing you to define and orchestrate the deployment of all the services in a single YAML file. This makes it easier to develop, deploy, and scale microservices applications.
5. **Local Development with Dependency Management:** Docker Compose is useful for managing dependencies between services in a local development environment. For example, if your application requires a database, a caching service, and a message queue, you can define all these services in a docker-compose.yml file and start them with a single command. This eliminates the need to install and configure these dependencies manually on each developer's machine.
6. **Demo and Training Environments:** Docker Compose is great for creating demo and training environments where you need to showcase or teach how to set up and run a multi-container application. You can package all the necessary components into a Docker Compose file and distribute it to users, allowing them to quickly spin up the environment on their own machines.

😄 To begin with the lab:

1. In this lab you are going to learn about docker compose. It is another utility of docker.
2. So, docker compose is a separate utility which you need to install in order to use it.
3. Docker compose is a tool to run multi-containers together.
4. Now for this lab you should have an instance ready or you can create one on AWS Console. You should use Ubuntu OS.
5. After that login or SSH into the instance and then use the below link to install docker compose.

   https://docs.docker.com/compose/install/linux/#install-the-plugin-manually

   DOCKER_CONFIG=${DOCKER_CONFIG:-$HOME/.docker}
   mkdir -p $DOCKER_CONFIG/cli-plugins
   curl -SL https://github.com/docker/compose/releases/download/v2.24.5/docker-compose-linux-x86_64 -o $DOCKER_CONFIG/cli-plugins/docker-compose

   chmod +x $DOCKER_CONFIG/cli-plugins/docker-compose

   docker compose version

```
ubuntu@ip-172-31-41-113:~$ DOCKER_CONFIG=${DOCKER_CONFIG:-$HOME/.docker}
mkdir -p $DOCKER_CONFIG/cli-plugins
curl -SL https://github.com/docker/compose/releases/download/v2.24.5/docker-compose-linux-x86_64 -o $DOCKER_CONFIG/cli-plugins/docker-compose
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100 58.5M  100 58.5M    0     0  40.0M      0  0:00:01  0:00:01 --:--:-- 49.8M
ubuntu@ip-172-31-41-113:~$
```

```
ubuntu@ip-172-31-41-113:~$ chmod +x $DOCKER_CONFIG/cli-plugins/docker-compose
ubuntu@ip-172-31-41-113:~$ docker compose version
Docker Compose version v2.24.5
```

6. Now use this link below to work on docker compose. You can find everything here related to this lab.
7. The steps are mentioned very clearly on how you can containerize docker compose.
8. Build an image starting with the Python 3.10 image.
9. Set the working directory to /code.
10. Set environment variables used by the flask command.
11. Install gcc and other dependencies
12. Copy requirements.txt and install the Python dependencies.
13. Add metadata to the image to describe that the container is listening on port 5000
14. Copy the current directory . in the project to the workdir . in the image.
15. Set the default command for the container to flask run.
    https://docs.docker.com/compose/gettingstarted/

16. First you need to create a directory.
    mkdir composetest
    cd composetest
17. Then you need to create this app file in python
    vim app.py

```
import time

import redis
from flask import Flask

app = Flask(__name__)
cache = redis.Redis(host='redis', port=6379)

def get_hit_count():
    retries = 5
    while True:
        try:
            return cache.incr('hits')
        except redis.exceptions.ConnectionError as exc:
            if retries == 0:
                raise exc
            retries -= 1
```

```
            time.sleep(0.5)

    @app.route('/')
    def hello():
       count = get_hit_count()
       return 'Hello World! I have been seen {} times.\n'.format(count)
```

18. Then you need to create a requirement file for python.
    vim requirements.txt

    ```
    flask
    redis
    ```

19. Now you need to create a docker file.

    vim Dockerfile

    ```
    # syntax=docker/dockerfile:1
    FROM python:3.10-alpine
    WORKDIR /code
    ENV FLASK_APP=app.py
    ENV FLASK_RUN_HOST=0.0.0.0
    RUN apk add --no-cache gcc musl-dev linux-headers
    COPY requirements.txt requirements.txt
    RUN pip install -r requirements.txt
    EXPOSE 5000
    COPY . .
    CMD ["flask", "run"]
    ```

20. After that create a YAML file.
    compose.yaml

    ```
    services:
      web:
        build: .
        ports:
          - "8000:5000"
      redis:
        image: "redis:alpine"
    ```

21. In the end run the execution command.
    docker compose up

```
ubuntu@ip-172-31-41-113:~$ mkdir composetest
ubuntu@ip-172-31-41-113:~$ cd composetest
ubuntu@ip-172-31-41-113:~/composetest$ vim app.py
ubuntu@ip-172-31-41-113:~/composetest$ vim requirements.txt
ubuntu@ip-172-31-41-113:~/composetest$ vim Dockerfile
ubuntu@ip-172-31-41-113:~/composetest$ vim compose.yaml
```

```
ubuntu@ip-172-31-41-113:~/composetest$ docker compose up
```

22. Now you will that the process is running in the foreground and you can go to AWS Console and then on EC2 from there copy the Public IP address of the instance and paste it in a new tab. Use the port number which you will use from the documentation.

Hello World! I have been seen 3 times.