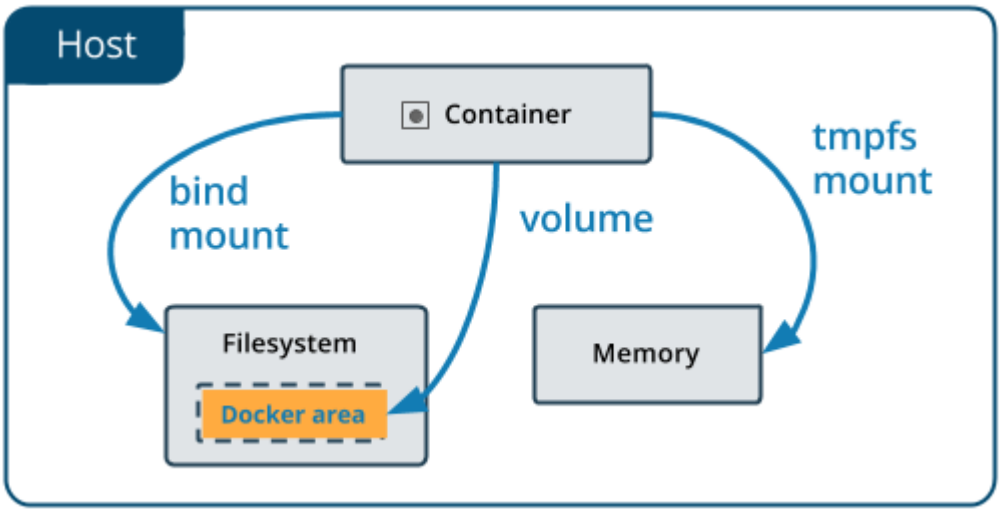
#90DaysOfDevOps Challenge - Day 19 - Docker for DevOps Engineers (Docker Volume and Docker Network)

Welcome to Day 19 of the #90DaysOfDevOps Challenge. Today, we will explore two important concepts in Docker: Docker Volume and Docker Network. These features enable us to manage data persistence and establish communication between containers efficiently. Let's dive into the details.

Docker Volume



In **Docker**, a **volume** is a directory that is stored outside the container's filesystem. It allows us to persist data and **share** it between **containers** or between the **host** machine and containers. Docker volumes provide advantages such as **data persistence**, **improved container performance**, and **easier data management**.

When a container is created, Docker sets up a writable storage layer. However, any changes made to this storage layer will not persist if the container is deleted or replaced. This is where Docker volumes come into play. Volumes provide a way to store and manage data separately from the container itself.

Docker volumes can be used in various scenarios. For example, they are commonly used to **store application data, configuration files**, **logs**, or any **other data** that needs to be **persistent**. By utilizing volumes, we can ensure that our data remains intact even if the container is restarted or replaced.

Additionally, Docker volumes can be shared between multiple containers, enabling efficient data exchange. This is particularly useful in microservices architectures, where different containers need access to shared data.

To use Docker volumes, we can specify the `-v` or `--volume` flag when running a container, followed by the source and destination paths. For example:

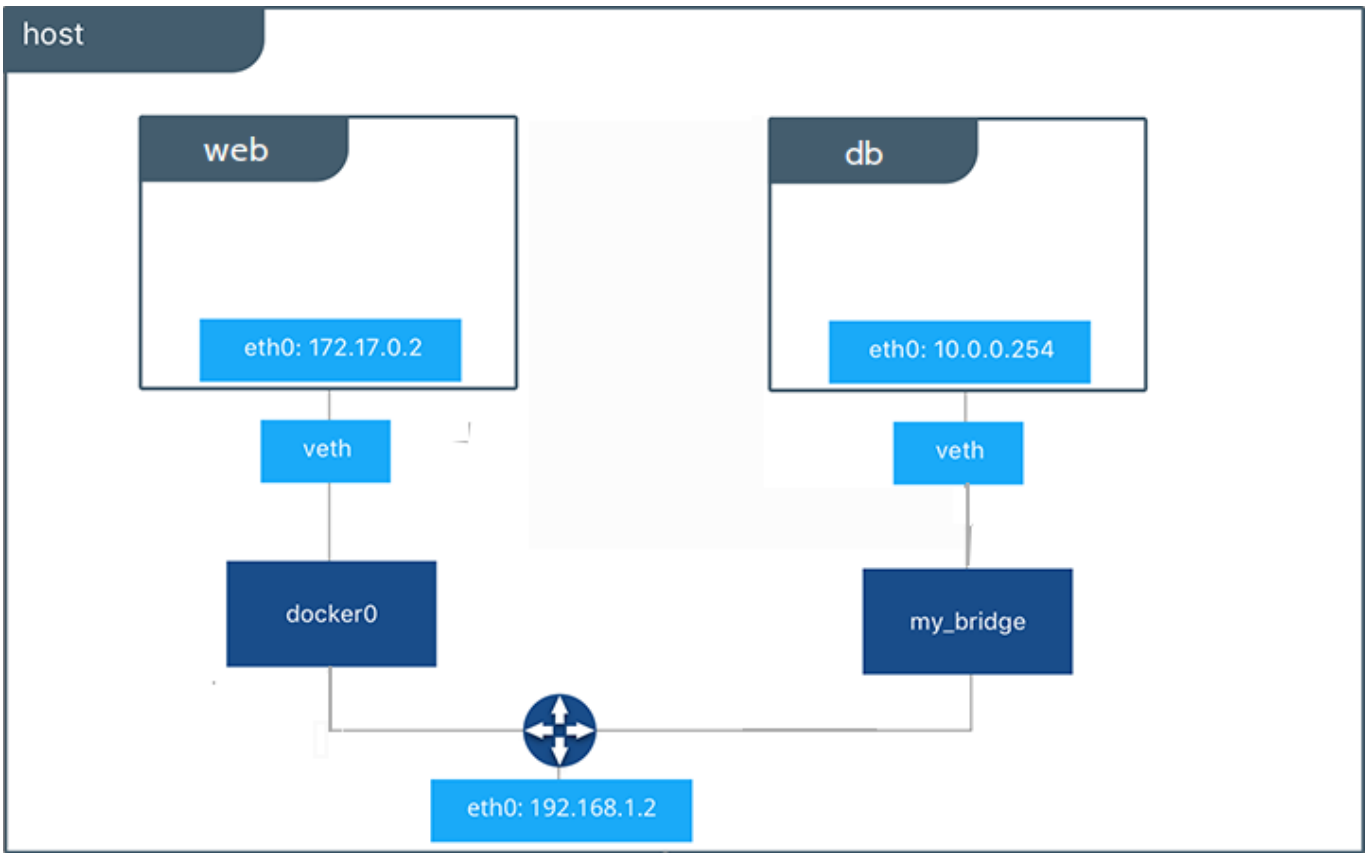
```
docker run -v /host/path:/container/path image_name
```

This command mounts a volume from the host machine to a specific path inside the container. Any changes made to the data in that path will be reflected in both the container and the host. By specifying different paths, we can easily share and persist data between containers.

In addition to using volumes with `docker run`, Docker provides commands like `docker volume create`, `docker volume inspect`, and `docker volume rm` to manage volumes and perform operations such as creating, inspecting, and removing volumes.

Docker volumes are a powerful feature that enhances data management and facilitates seamless communication between containers.

Docker Network



Docker Network facilitates **communication** between containers and connects them with the outside world. By **default**, Docker creates a **bridge network** for containers to communicate with each other. Additionally, Docker supports **other network types** like **host**, **overlay**, and **macvlan**, allowing flexible networking configurations based on your application requirements.

When containers are connected to a network, they can communicate with each other using container names as hostnames. This simplifies the process of establishing communication between containers, as Docker takes care of name resolution and routing.

Docker networks also enable containers to communicate with external systems or services. By exposing ports on the host machine and mapping them to container ports, we can access containers from the host or other machines on the network.

In addition to the default bridge network, Docker provides other network drivers to suit different networking scenarios. For example, the host network driver allows a container to share the network namespace with the

host, eliminating the need for network address translation (NAT) and providing better performance.

Overlay networks are used in **distributed environments** where containers need to **communicate across multiple hosts**. They enable seamless communication between containers running on different hosts by abstracting the underlying network infrastructure.

Macvlan networks allow containers to have their **own MAC addresses**, making them appear as separate physical devices on the network. This is useful when containers need to be directly connected to the physical network.

By understanding and utilizing Docker networks effectively, we can create **flexible and scalable architectures** for our containerized applications.

Now that we have explored the concepts of Docker Volume and Docker Network, let's proceed to the tasks to gain hands-on experience in using them.

Task 1: Creating a Multi-Container Docker-Compose File

To manage multiple containers effortlessly, we will use Docker Compose. Follow the steps below:

1. Write a docker-compose.yml file defining the services for your application and database containers.

```yaml
version : "3.3"
services:
  web:
    image: varsha0108/local_django:latest
    deploy:
        replicas: 2
    ports:
      - "8001-8005:8001"
    volumes:
      - my_django_volume:/app
  db:
    image: mysql
    ports:
      - "3306:3306"
    environment:
      - "MYSQL_ROOT_PASSWORD=test@123"
volumes:
  my_django_volume:
    external: true
```

The given `docker-compose.yaml` file defines a multi-service Docker application with two services:

1. **Web service**:
    - Uses the image `varsha0108/local_django:latest`.
    - Deploys two replicas.

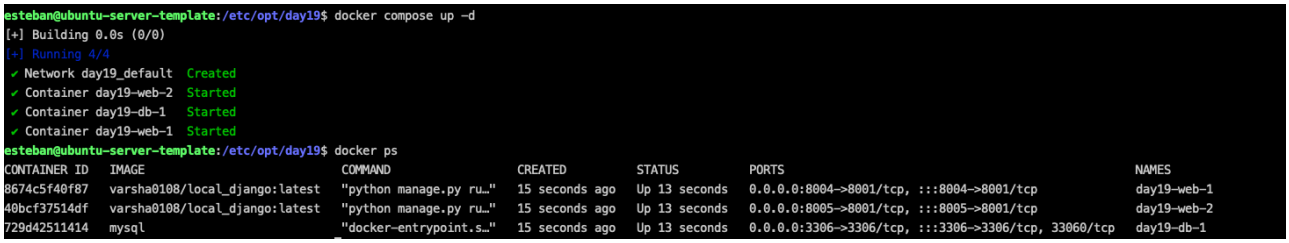
- Maps host ports 8001-8005 to container port 8001.
- Mounts the external volume `my_django_volume` to `/app` in the container.

2. **DB service**:
   - Uses the `mysql` image.
   - Maps host port 3306 to container port 3306.
   - Sets the root password as `test@123` using an environment variable.

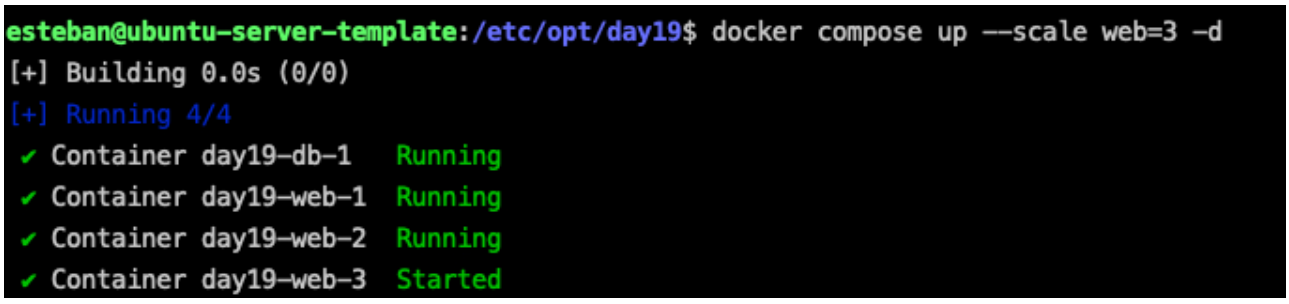
The application utilizes Docker volumes for persistent storage, and the `my_django_volume` volume is assumed to be created externally. If not, use the below command:

```
docker volume create my_django_volume
```

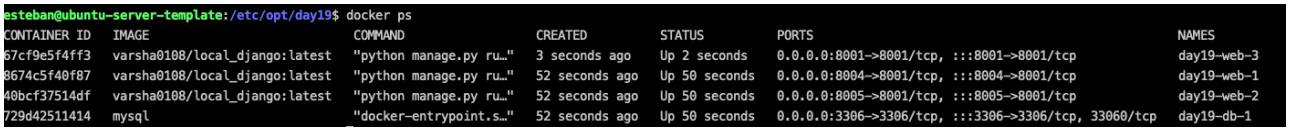
1. Use the `docker-compose up -d` command to start the containers in detached mode.



2. Employ the `docker-compose scale` command to scale the number of replicas for specific services or enable auto-scaling by adding the `replicas` parameter in the deployment file.



3. Verify the container status using `docker compose ps` and view logs using `docker compose logs`.



```
esteban@ubuntu-server-template:/etc/opt/day19$ docker compose logs
day19-db-1  | 2023-06-09 19:13:01+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL Server 8.0.33-1.el8 started.
day19-db-1  | 2023-06-09 19:13:01+00:00 [Note] [Entrypoint]: Switching to dedicated user 'mysql'
day19-db-1  | 2023-06-09 19:13:01+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL Server 8.0.33-1.el8 started.
day19-db-1  | 2023-06-09 19:13:02+00:00 [Note] [Entrypoint]: Initializing database files
day19-db-1  | 2023-06-09T19:13:02.289015Z 0 [Warning] [MY-011068] [Server] The syntax '--skip-host-cache' is deprecated and will be removed in a future release. Please use SET GLOBAL host_cache_size=0 instead.
day19-db-1  | 2023-06-09T19:13:02.289124Z 0 [System] [MY-013169] [Server] /usr/sbin/mysqld (mysqld 8.0.33) initializing of server in progress as process 79
day19-db-1  | 2023-06-09T19:13:02.351116Z 1 [System] [MY-013576] [InnoDB] InnoDB initialization has started.
day19-db-1  | 2023-06-09T19:13:03.118281Z 1 [System] [MY-013577] [InnoDB] InnoDB initialization has ended.
day19-db-1  | 2023-06-09T19:13:04.589021Z 6 [Warning] [MY-010453] [Server] root@localhost is created with an empty password ! Please consider switching off the --initialize-insecure option.
day19-db-1  | 2023-06-09 19:13:09+00:00 [Note] [Entrypoint]: Database files initialized
day19-db-1  | 2023-06-09 19:13:09+00:00 [Note] [Entrypoint]: Starting temporary server
day19-db-1  | 2023-06-09T19:13:09.706094Z 0 [Warning] [MY-011068] [Server] The syntax '--skip-host-cache' is deprecated and will be removed in a future release. Please use SET GLOBAL host_cache_size=0 instead.
day19-db-1  | 2023-06-09T19:13:09.707714Z 0 [System] [MY-010116] [Server] /usr/sbin/mysqld (mysqld 8.0.33) starting as process 121
```

4. Finally, use `docker compose down` to stop and remove all containers, networks, and volumes associated with the application.

```
esteban@ubuntu-server-template:/etc/opt/day19$ docker compose down
[+] Running 5/5
 ✔ Container day19-db-1   Removed
 ✔ Container day19-web-2  Removed
 ✔ Container day19-web-3  Removed
 ✔ Container day19-web-1  Removed
 ✔ Network day19_default  Removed
esteban@ubuntu-server-template:/etc/opt/day19$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
esteban@ubuntu-server-template:/etc/opt/day19$
```

Task 2: Using Docker Volumes

To share data between containers using Docker Volumes, follow these steps:

1. Create a new volume

```
docker volume create ubuntu_volume
docker volume ls
docker volume inspect ubuntu_volume
```

```
esteban@ubuntu-server-template:/etc/opt/day19$ docker volume create ubuntu_volume
ubuntu_volume
esteban@ubuntu-server-template:/etc/opt/day19$ docker volume ls
DRIVER    VOLUME NAME
local     ubuntu_volume
esteban@ubuntu-server-template:/etc/opt/day19$ docker volume inspect ubuntu_volume
[
    {
        "CreatedAt": "2023-06-10T10:33:22Z",
        "Driver": "local",
        "Labels": null,
        "Mountpoint": "/var/lib/docker/volumes/ubuntu_volume/_data",
        "Name": "ubuntu_volume",
        "Options": null,
        "Scope": "local"
    }
]
```

2. Create two or more containers that need to read and write data. Use the below command to specify the same volume configuration for each container.

```
docker run -d --name nginx_container_1 -v ubuntu_volume:/app
nginx:latest
docker run -d --name nginx_container_2 -v ubuntu_volume:/app
nginx:latest
docker ps
```

```
esteban@ubuntu-server-template:/etc/opt/day19$ docker run -d --name nginx_container_1 -v ubuntu_volume:/app nginx:latest
dabd72adfc2e4d8bca9cda34486927fae94b4201df0ac3b00af8ddfd88eea42e
esteban@ubuntu-server-template:/etc/opt/day19$ docker run -d --name nginx_container_2 -v ubuntu_volume:/app nginx:latest
576ca74066697dfacc0dc8747d221faf9641bd55c5d4a4b1d02db653304322c1
esteban@ubuntu-server-template:/etc/opt/day19$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          PORTS     NAMES
576ca7406669   nginx:latest   "/docker-entrypoint.…"   2 seconds ago    Up 1 second     80/tcp    nginx_container_2
dabd72adfc2e   nginx:latest   "/docker-entrypoint.…"   12 seconds ago   Up 11 seconds   80/tcp    nginx_container_1
esteban@ubuntu-server-template:/etc/opt/day19$
```

3. Verify data consistency by executing commands inside each container using `docker exec`.

```
docker exec -it dabd72adfc2e bash
```

```
esteban@ubuntu-server-template:/etc/opt/day19$ docker exec -it dabd72adfc2e bash
root@dabd72adfc2e:/# cd app
root@dabd72adfc2e:/app# echo "File created in container 1" > volume-test.txt
root@dabd72adfc2e:/app# ls
volume-test.txt
root@dabd72adfc2e:/app# exit
exit
esteban@ubuntu-server-template:/etc/opt/day19$
```

I have created a new file called `'volume-test.txt'` while connected to `nginx_container_1`. Let's connect to `nginx_container_2` and verify the text file is there.

```
docker exec -it nginx_container_2 bash
```

```
esteban@ubuntu-server-template:/etc/opt/day19$ docker exec -it nginx_container_2 bash
root@576ca7406669:/# cd app
root@576ca7406669:/app# ls
volume-test.txt
root@576ca7406669:/app# cat volume-test.txt
File created in container 1
root@576ca7406669:/app# exit
exit
```

In this example, we have used the `container ID` and the `container name` with the `docker exec` command to verify the content of the volume

4. Utilize `docker volume ls` to list all volumes and `docker volume rm` to remove the volume once you have finished using it.

```
esteban@ubuntu-server-template:/etc/opt/day19$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          PORTS     NAMES
576ca7406669   nginx:latest   "/docker-entrypoint.…"   18 minutes ago   Up 18 minutes   80/tcp    nginx_container_2
dabd72adfc2e   nginx:latest   "/docker-entrypoint.…"   18 minutes ago   Up 18 minutes   80/tcp    nginx_container_1
esteban@ubuntu-server-template:/etc/opt/day19$ docker stop nginx_container_1
nginx_container_1
esteban@ubuntu-server-template:/etc/opt/day19$ docker stop nginx_container_2
nginx_container_2
esteban@ubuntu-server-template:/etc/opt/day19$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

```
esteban@ubuntu-server-template:/etc/opt/day19$ docker ps -a --filter "volume=ubuntu_volume"
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS                          PORTS     NAMES
576ca7406669   nginx:latest   "/docker-entrypoint.…"   20 minutes ago   Exited (0) About a minute ago             nginx_container_2
dabd72adfc2e   nginx:latest   "/docker-entrypoint.…"   20 minutes ago   Exited (0) About a minute ago             nginx_container_1
esteban@ubuntu-server-template:/etc/opt/day19$ docker rm nginx_container_1
nginx_container_1
esteban@ubuntu-server-template:/etc/opt/day19$ docker rm nginx_container_2
nginx_container_2
```

```
esteban@ubuntu-server-template:/etc/opt/day19$ docker volume ls
DRIVER    VOLUME NAME
local     ubuntu_volume
esteban@ubuntu-server-template:/etc/opt/day19$ docker volume rm ubuntu_volume
ubuntu_volume
esteban@ubuntu-server-template:/etc/opt/day19$ docker volume ls
DRIVER    VOLUME NAME
```

Congratulations on completing Day 19 of the #90DaysOfDevOps challenge. In Day 19, we focused on further exploring Docker for DevOps Engineers. We learned how to work with Docker volumes to persist data across containers and how to scale our services using the Docker Compose scale command.

Moving on to Day 20, we will review the Docker commands we have learned so far and create a comprehensive Docker cheatsheet. This cheatsheet will serve as a handy reference guide for you to quickly access and use Docker commands effectively in your DevOps workflows. Stay tuned!