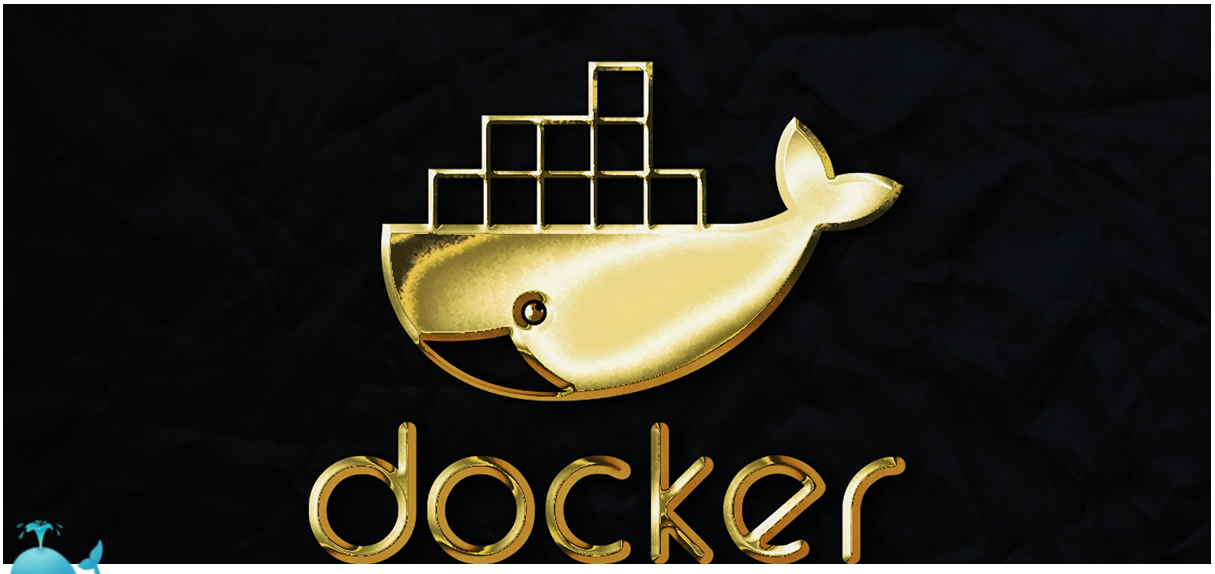


Docker

Table of Contents:

▼ ⚙️ Docker Config Templates

Description

Docker file → Image → Container.

Docker file

Dockerfile. It is a script that contains a series of instructions on how to build a Docker image.. Docker file has no extension and has a specific syntax. Based on docker files we can build custom images.

Docker Image

Docker image. It is an image for creating Docker Containers. Using one Docker image you can create multiple containers.

Docker Container

Docker Container. It is a standardized executable components combining application source code with the operating system (OS) libraries and dependencies required to run that code in any environment.

▼ Installation

1. Check virtualization enabled. Enable in not enabled

```
$ sudo lscpu | grep Virtualization
$ full or amd_v
```

2. [VPS/VDS]

```
sudo apt install gnome-terminal
```

3. [OPTIONAL]

```
sudo apt remove docker-desktop
```

4. Set up the Docker repository:
    a. Install required dependencies

```
$ sudo apt update
$ sudo apt install software-properties-common curl apt-transpo
rt-https ca-certificates -y
```

    b. Once the installation is complete, add Docker’s GPG signing key.

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
```

    c. Next, add the official Docker’s repository to your system as follows

```
$ sudo add-apt-repository "deb [arch=amd64] https://download.doc
```

5. Install docker

```
$ sudo apt install docker-ce docker-ce-cli containerd.io uidmap -y
```

6. Verify

```
sudo systemctl status docker
```

▼ Basic command

Docker version info

Command	Description
docker —version	Print current docker version
docker info	Display detailed docker info

Docker images

Command	Description
docker images	Print all docker images
docker pull image_name:tag	Pull image from the Docker Hub
docker rmi image_name:tag	Remove image. (tag is optional)
docker run IMAGE	Run docker image (if no image, install from the docker server)
docker build -t image_name:tag	Build an image based on the Docker file.
docker attach CONTAINER_ID	Attach terminal to the container
CTRL+P CTRL+Q	Detach the container from the terminal

▼ Images exchange

There are 3 main ways to exchange images with other developers:

1. Docker HUB. Public or Private Docker Images repositories.
2. Exporting and Importing Images to a `.tar` file. You can save a Docker image as a file and share it directly with other developers. This method is useful when you cannot or do not want to use a Docker registry.
3. Using Private Docker registers. If you want to share images privately, you can set up a private Docker registry (e.g., using Docker's own `registry` image, or a third-party service like AWS ECR, Google Container Registry, or GitLab Container Registry).

Docker HUB

Pushing an Image to Docker Hub.

1. Tag the Image:

   `docker tag my_image:latest username/my_image:latest` .

2. Push the Image to Docker Hub:

   `docker push username/my_image:latest`

Pulling an Image:

1. Other developers can pull the Image from Docker Hub using:

   `docker pull username/my_image:latest`

2. After pulling, they can run a Container based on the Image:

`docket run -it username/my_image:latest`

Exporting and Importing Images

1. Save Docker Image to a Tar file:

   `docker save -o my_image.tar my_image:latest`

   This command creates a tar file (`my_image.tar`) containing the image and all its layers.

2. To load the Image on another machine use:

   `docker load -i my_image.tar`

   Now, developers can run Containers from this Image.

Using private Docker registers

1. Setting up a private register:
   a. You can run your own private Docker registry by running:

      `docker run -d 5000:5000 --name registry registry:2`

   b. Push to the private registry by tagging the image with your registry's address:

      `docker tag my_image:latest localhost:5000/my_image:latest`

      `docker push localhost:5000/my_image:latest`

2. Other developers can pull the image using:

   `docker pull localhost:5000/my_image:latest`

Docker Containers

Command	Description
docker ps	Print running Containers
docker ps -a	Print all existing Containers
docker start container_id	Start Docker Container
docker stop container_id	Stop Docker Container
docker rm container_id	Remove Docker Container
docker run image	Run a Container from an Image (Flags: **-i** Interactive mod, **-t** TTY, allocates psudo terminal, **-d** Detached mode, **—name** NAME Gives name to the Container, **-p** Publish a Container's port to the host, **-v** Bind mound volume).
docker attach CONTAINER_ID	Attach the Container to the terminal
CTRL+P CTRL+Q	Detach the Container from the terminal
docker logs CONTAINER_ID	View Container logs
docker exec -it CONTAINER_NAME /bin/sh	Execute command inside of a running Docker Container

Docker Networks

Command	Description
docker network ls	List all Docker Networks
docker network create NETWORK_NAME	Create a new Docker Network
docker network connect NETWORK_ID CONTAINER_ID	Connect a Container to a Network
docker network disconnect NETWORK_ID CONTAINER_ID	Disconnect a Container form a Network

Docker Volumes

Command	Network
docker volume ls	List Docker Volumes
docker volume create VOLUME_NAME	Create a new Docker Volume
docker volume inspect VOLUME_NAME	Inspect Docker Volume
docker volume rm VOLUME_NAME	Remove Docker Volume

▼ Dockerfile

A Dockerfile typically includes:

1. **Base Image**: Start from a base image (e.g., `python:3.9`).
2. **Maintainer Information**: (Optional) Information about the author.
3. **Environment Variables**: (Optional) Setting up environment variables.
4. **Working Directory**: Set the working directory within the container.
5. **Dependencies**: Install the necessary dependencies for your application.
6. **Copy Files**: Copy your application code into the container.
7. **Commands**: Specify commands to run within the container (e.g., running the application).

▼ Example (Dockerizing Django development server)

Project file structure

Suppose the Django project has following file structure:

```
my_django_project/
├── Dockerfile
├── requirements.txt
├── manage.py
```

```
└── myapp/
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

requirements.txt contains Python dependencies (e.g. Django etc.).

Dockerfile

Here is the sample of the 'dockerfile' for Django development server.

```
# Use the official Python image from the Docker Hub
FROM python:3.9-slim

# Set environment variables
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

# Set the working directory inside the container
WORKDIR /usr/src/app

# Copy the requirements file and install the dependencies
COPY requirements.txt /usr/src/app/
RUN python -m venv venv && \
    . venv/bin/activate && \
    pip install --upgrade pip && \
    pip install -r requirements.txt

# Copy the rest of the application code
COPY . /usr/src/app/

# Expose the port the app runs on
EXPOSE 8000

# Command to run the Django development server
CMD ["./venv/bin/python", "manage.py", "runserver", "0.0.0.0:8000"]
```

Dockerfile explanation:

1. Base Image: `FROM 'FROM python:3.9-slim'`
    a. This is a lightweight version of Python 3.9 image.
2. Environment variables:
    a. `PYTHONDONTWRITEBYTECODE 1` prevents Python from writing `.pyc` files.

    b. `PYTHONUNBUFFERED 1` ensures that Python output is sent straight to the terminal (useful for logging).

3. Working directory: `WORKDIR /usr/src/app`
    a. All subsequent commands are run from this directory inside of this Docker Container.
4. Copy and install dependencies:
    a. The `requirements.txt` file is copied into the container, and dependencies are installed into a Python virtual environment (`venv`).
5. Copy application code:
    a. The rest of the project files are copied into the container.
6. **Expose Port**: `EXPOSE 8000`
    a. This inform the host that this Docker Container is listening on port 8000
7. CMD command to run the server:
    a. The `CMD` directive runs the Django development server using the Python interpreter from the virtual environment.

Building Docker Image

Navigate to the directory containing your `Dockerfile` and run:

```
docker build -t my_django_project .
```

Run the Docker Container

```
docker run -p 8000:8000 my_django_project
```

The `-p 8000:8000` flag maps port 8000 of the container to port 8000 on your host machine. You should be able to access the Django development server at `http://localhost:8000`.

Additional tips

Debugging: You can enter a running container using the following command to debug issues:

```
docker exec -it <container_id> /bin/bash
```

Persistence: If you need to persist data (e.g., using a database), consider using Docker volumes to manage data outside the container.

▼ Docker Volume

Docker Volumes allow us to save Containers' data outbound a Container's life cycle.

Volumes can be stores on remote servers or cloud servers.

Definition

A Docker volume is a special directory that's not part of the container's file system. It's designed to persist data generated and used by Docker containers. This means that even if a container is removed or restarted, the data within the volume remains intact.

Docker Volume commands

Command	Description
`docker volume --help`	Print help message.
`docker volume ls`	Print list of available docker volumes.
`docker volume create VOLUME_NAME`	Create a new Docker Volume.
`docker volume inspect VOLUME_NAME`	Print the volume settings.
`docker volume rm VOLUME_NAME`	Remove a Volume from the host.
`docker volume prune`	Remove unused local volumes.

Mounting volume to a new Container

`docker run -d -it --name CONTAINER_NAME -v VOLUME_NAME:path/to/container ubuntu`

- -d (Runs Container in the detached mode)
- -it (Runs Container in the interactive terminal mode)
- —name ([Optional] Set container name)
- -v VOLUME_NAME or PATH:PATH_IN_CONTAINER (Mounts a volume to the Container)
  - VOLUME_NAME - name of the volume (on the host machine) (PS: check `docker volume ls`) you want to mount. Or absolute path to the volume on the host machine.
  - path/in/container - Path inside of the Container you want this volume to be mounted. Can be any path inside of the Container. e.g. `/shared-volume`
- ubuntu (Image).

Example:

`docker run -d -it --name myUbuntu -v myVolume:/shared-volume ubuntu`

You can read this command this way. Run docker container named 'myUbuntu' in the detached mode, in interactive terminal mode, *saving all data inside of Container located in '/shared-volume` to myVolume location on the host*, based on image ubuntu.

▼ Volume backup

▼ Volume backup into tar.gz

```
docker run --rm -v your_postgres_volume:/volume -v $(pwd):/backup
```

Explanation:

- `docker run --rm`: This command starts a new container and automatically removes it once the command inside it finishes executing. The container won't linger after the backup is complete.
- `v your_postgres_volume:/volume`: This mounts the Docker volume named `your_postgres_volume` into the container at the `/volume` path. Docker volumes are directories on the host system that are managed by Docker to persist data across container restarts.
- `v $(pwd):/backup`: This mounts your current working directory (on the host machine) into the container at `/backup`. The `$(pwd)` command dynamically inserts the path to your current directory. On Linux/macOS systems, this command fetches the absolute path of the directory you're in.
- `alpine`: This is the base image you're using to run the commands inside the container. Alpine is a lightweight Linux distribution often used for simple tasks like this.
- `tar czf /backup/postgres_backup.tar.gz`: This creates a compressed archive (`tar.gz`) of the contents. Let's break down the `tar` options:
  - `c`: Create a new archive.
  - `z`: Compress the archive using gzip.
  - `f`: Specifies the filename of the archive (`/backup/postgres_backup.tar.gz`), which will be stored in the `/backup` directory (your current working directory on the host machine).
- `C /volume`: Change to the `/volume` directory inside the container before starting the archive process. This is the path where your Docker volume is mounted.
- `.`: This means "include everything" in the current directory (which is `/volume` after the `C /volume` argument) in the archive.

In Summary:

This command creates a backup of all the files in the `your_postgres_volume` volume, compresses them into a `postgres_backup.tar.gz` file, and saves it in your current working directory on the host machine.

▼ Volume transfer

1. Using SSH copy:
   `scp postgres_backup.tar.gz user@new_vps_ip:/path/to/destination/`

▼ Volume unpacking

```
docker volume create your_postgres_volume
docker run --rm -v your_postgres_volume:/volume -v /path/to/desti
```

1. `docker volume create your_postgres_volume`:

This creates a new Docker volume called `your_postgres_volume`. Docker volumes are used to persist data even when the container stops or is removed.

2. `docker run --rm -v your_postgres_volume:/volume -v /path/to/destination:/backup alpine tar xzf /backup/postgres_backup.tar.gz -C /volume`

- `docker run --rm`: Same as before, this runs a temporary container that will be removed after the restore is done.
- `v your_postgres_volume:/volume`: Mounts the newly created Docker volume `your_postgres_volume` to `/volume` in the container, where the data will be restored.
- `v /path/to/destination:/backup`: Mounts the directory on your host system that contains the backup file (`postgres_backup.tar.gz`) to the container's `/backup` directory. Replace `/path/to/destination` with the actual path where your backup file is stored.
- `tar xzf /backup/postgres_backup.tar.gz -C /volume`:
  - `x`: Extract files from the archive.
  - `z`: Decompress the `gzip` archive.
  - `f`: Specifies the archive file to extract (`/backup/postgres_backup.tar.gz`).
  - `C /volume`: Change to the `/volume` directory before extracting, so that the extracted files are restored into the Docker volume.

In Summary:

This command restores the contents of the `postgres_backup.tar.gz` archive into the `your_postgres_volume` volume by extracting the backup files into the volume's directory.

▼ Docker Network

Definition

Docker Network is a Docker component vital for building networks between Containers and the host network, and external network.

⚠ Do no use the default bridge for production. Use a user-defined bridge.

What Docker Network Does:

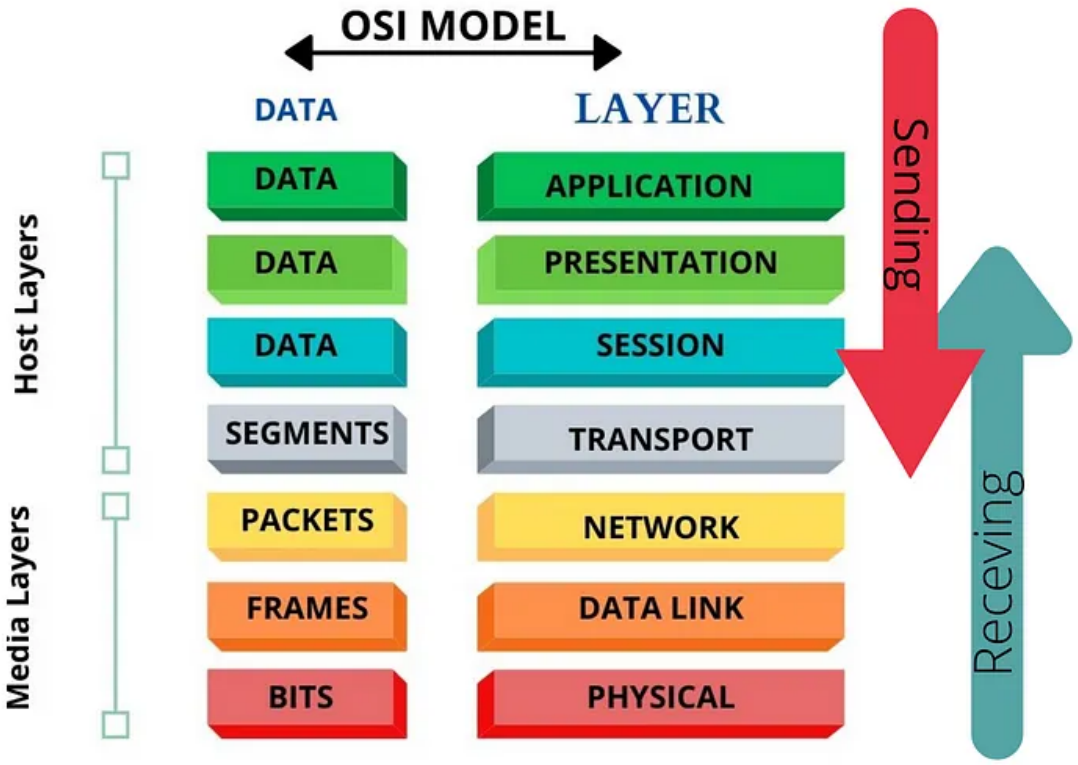
- **Container Communication**: Allows containers to communicate with each other, whether they are on the same host or different hosts.
- **Isolation**: By default, Docker containers are isolated from each other. Docker networks enable controlled communication between containers.

- **External Access**: Provides access to the outside world, enabling containers to reach external services or be accessed from outside the Docker host.
- **Service Discovery**: When containers join the same network, they can discover each other by their container name or service name (in Docker Compose).

Docker Network types:

1. **Bridge Network:** The default network. Containers connected to the same bridge network are able to communicate between each other but isolated from Containers on different bridge networks. (Default Bridge Networks do not support DNS between Containers, create custom instead). DNS names are Container names that can be used inside the Network.
2. **Host Network:** Uses the host machine network. Containers connected to the host network share the same IP address as the host.
3. **Overlay Network:** Uses for communication between Containers across multiple Docker hosts is a swarm cluster.
4. **MACVlan Network:** Assigns a MAC address to each container, making them appear as physical devises on the network.
    a. Bridge mode.
    b. 802.1q mode. In this mode make able to assign network interfaces to each Container like `enp3s1` and so on.
5. **IPVLAN**:
    a. L2 mode (default): The same as MACVlan but the MAC addresses of Container will match with the host MAC address. Which solves MAC address issue in MACVlan networks.
    b. L3 mode:
6. **None:** Disables networking for a Container. Container has no network interface.

▼ OSI



Docker Network commands

Command

Command	Description
`docker network create [OPTIONS] NETWORK_NAME`	Create a new Docker Network.
`docket network ls`	Print list of Docker Networks.
`docker inspect NETWORK_NAME`	Print detailed information about the Network.
`docker network connect [OPTIONS] NETWORK_NAME CONTAINER_NAME`	Connect Container to a Network.
`docker rm NETWORK_NAME`	Remove Docker Network.
`bridge link`	Show active network bridges on the host.

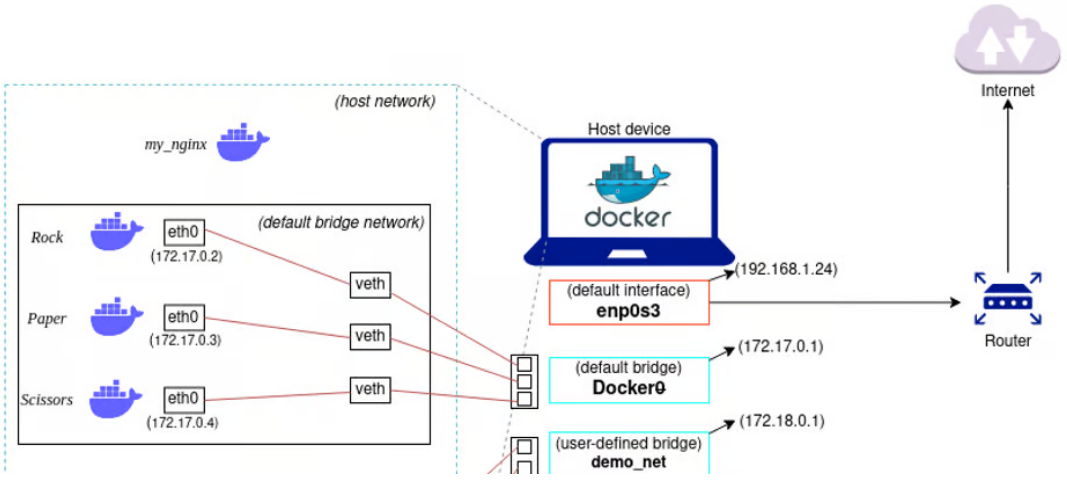
Flags:

- OPTIONS for `docker network create`:
  - `-driver`: Specifies the network driver (`bridge`, `overlay`, `macvlan`, etc.).
  - `-subnet`: Specifies the subnet (e.g., `192.168.1.0/24`).
  - `-gateway`: Specifies the gateway for the network.

- `-ip-range`: Specifies a range of IP addresses for containers.
- `-internal`: Makes the network internal, preventing external access

- OPTIONS for `docker network connect NETWORK_NAME CONTAINER_NAME`
  - `-ip`: Assign a specific IP address to the container.
  - `-alias`: Add a network-scoped alias for the container.

Docket Network Package travel scheme



▼ Configuring Networks inside of `docker-compose.yml`

Let's say we want to configure a Docker Network inside of the `docker-compose.yml` file. The docker-compose will cover following services: NGINX, Django, PostgreSQL.

```
version: '3.8'

services:
  nginx:
    image: nginx:latest
    ports:
      - "80:80"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
    depends_on:
      - web
    networks:
      - frontend
      - backend

  web:
    image: django:latest
    volumes:
```

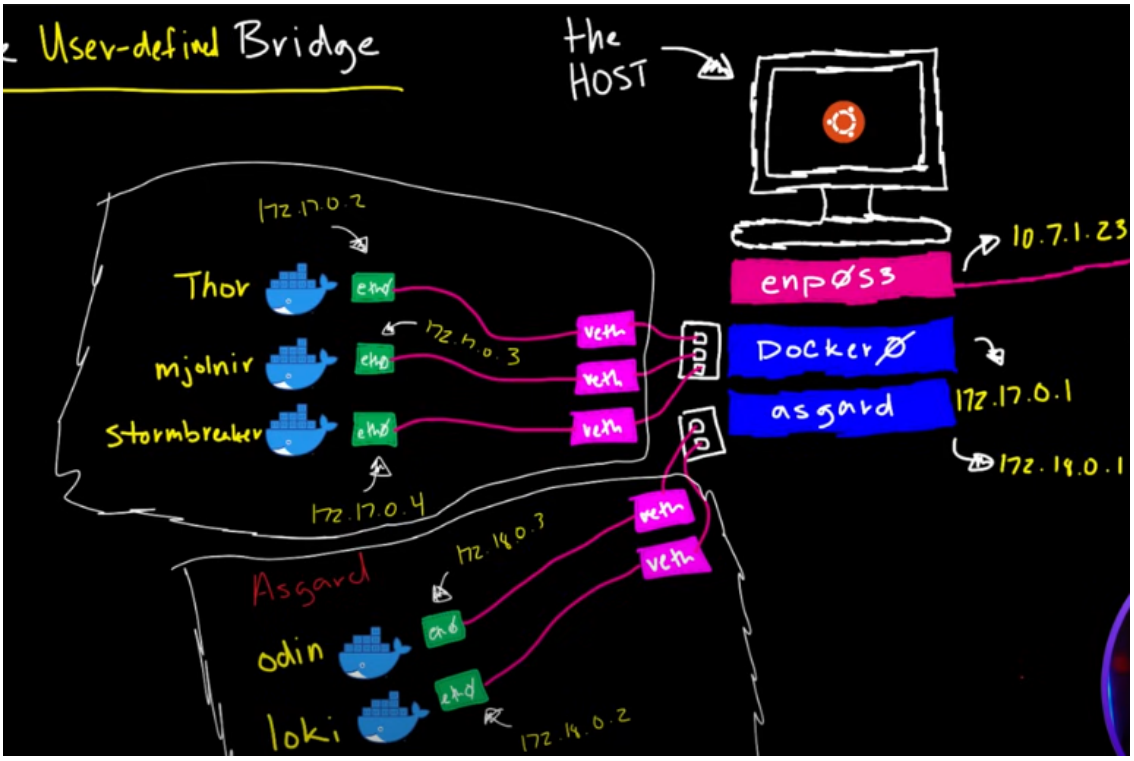
```
      - .:/app
    depends_on:
      - db
    networks:
      - backend

  db:
    image: postgres:latest
    environment:
      POSTGRES_DB: mydatabase
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
    networks:
      - backend

networks:
  frontend:
    driver: bridge
  backend:
    driver: bridge
```

▼ Example of creating a user-defined bridge network

This type of network should be used instead of the default bridge. The whole point of using a user-defined bridge is to isolate Containers from the default bridge network. Also a user-defined Network Bridge supports DNS for Containers.



Isolating Containers from the default Network bridge.

1. Create the network:

   `docker network create NETWORK_NAME`

2. Run Containers within your Network:

   `docker run -itd --network NETWORK_NAME --name CONTAINER_NAME IMAGE_NAME`

▼ Example of creating MACVlan network

Bridge mode

❗ Make sure your network supports multiple MAC addresses on one port. (The issue is that all Container are connected to the same physical (digital) port on your switch or router. While some switches or routers cannot assign different MAC addresses to a single one physical (digital) port).

Try to enable the promiscuous mode on the router.

1. Enable promiscuous mode on your host machine.

   `sudo ip link set enp3s0 promisc on`

2. Might need to enable the promiscuous mode on the router.

1. Create the network

   `docker network create -d macvlan --subnet 10.7.1.0/24 --gateway 10.7.1.3 -o parent=enp3s0 NETWORK_NAME`

   Flags:

   - -d - Specify drive. (macvla, bridge, host, etc.).
   - —subnet - Specify your (home) network subnet. `ip a` enp3s0;

- —gateway - Specify the gateway address (Router network).
- - o - Set options:
  - parent - Set parent network interface. enp3s0;

2. Run the Containers within this Network

   `docker run -itd --name CONTAINER_NAME --network NETWORK_NAME --ip 10.7.0.5 IMAGE_NAME`

   Specify the Container's IP address manually. Make sure this IP address is free in your network.

3. Check the promiscuous mode if you cannot access other devises on the network.
4. Remember this type of network does not have its own DHCP server, thus you have to assign IP addresses to each Container manually. In case, you do not specify the IP address manually Docker will use its own DHCP server, this means that there are two DHCP server on the network which may lead to collisions.

802.1q mode

1. Create the network

   `docker network create -d macvlan --subnet 192.168.20.0/24 --gateway 192.168.20.1 -o parent= enp3s0.10  NETWORK_NAME`

   Flags:

   - -d - Specify drive. (macvla, bridge, host, etc.).
   - —subnet - Specify your (home) network subnet. `ip a` enp3s0;
   - —gateway - Specify the gateway address (Router network).
   - - o - Set options:
     - parent - Set parent network interface. enp3s0;

2. Run the Containers within this Network

   `docker run -itd --name CONTAINER_NAME --network NETWORK_NAME --ip 10.7.0.5 IMAGE_NAME`

▼ Example of creating IPVLAN network

L2 (default) (PS: L stands for payer of the OSI model)

1. Create the network

   `docker network create -d ipvlan --subnet 10.7.1.0/24 --gateway 10.7.1.3 -o parent=enp3s0 NETWORK_NAME`

   Flags:

   - -d - Specify drive. (macvla, bridge, host, etc.).
   - —subnet - Specify your (home) network subnet. `ip a` enp3s0;
   - —gateway - Specify the gateway address (Router network).
   - - o - Set options:
     - parent - Set parent network interface. enp3s0;

2. Run the Containers within this Network

   `docker run -itd --name CONTAINER_NAME --network NETWORK_NAME --ip 10.7.0.5 IMAGE_NAME`

   Specify the Container's IP address manually. Make sure this IP address is free in your network.

▼ Docker Compose

Definition. Docker Compose is a Docker service for defining and running multi-container applications.

Docker Compose commands

Command	Description
docker compose config	Validate the docker-compose file
docker compose up	Up docker containers. Flag -d for detached mode.
docker compose down	Stop every docker container that was upped by the Compose
docker-compose stop	Stop all Containers
docker-compose stat	Start all existing Containers
docker-compose restart	Restarts all Containers.
docker-compose logs APPLICATION_NAME	Check Application logs. Useful when containers were upped in the detached mode.

⚠ Note: `docker-compose down` will remove all Containers, Networks and Unnamed volumes with flags -v. Use docker-compose stop for safety.

Docker compose flags

Flag	Description	Example
-f	Specify `docker-compose.yml` file.	`docker-compose -f docker-compose.yml up`
-e	Specify the virtual environment file.	`docker-compose -e .env up`

YML files

Docker compose file can be create anywhere on the host. The standard name for this file is `docker-compose.yml`

▼ Example of using docker-compose (Composing Django/Gunicorn/Redis/PostgreSQL/NGINX project)

1. Crete `docker-compose` file:
2. Validate `docker-compose` file:

```
docker-compose config
```

3. Create `Dockerfile` s for each micro-service
    a. NGINX config:

```
user  nginx;
worker_processes  auto;

events {
    worker_connections  1024;
}

http {
    log_format  main  '$remote_addr - $remote_user [$time_loca
                      '$status $body_bytes_sent
 "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_f

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


    server {
        listen 80;

        location /static/ {
            root /code/app/static;
        }

        location / {
            proxy_pass http://web:8000;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forw

        }
    }
}
```

    b. NGINX:

```
FROM nginx:alpine
COPY nginx.conf /etc/nginx/nginx.conf
```

    c. Gunicorn:

```
FROM python:3.9-slim-buster

WORKDIR /code

COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt

COPY . .

CMD ["gunicorn", "--workers", "3", "--bind", ":8000", "app.ws
```

d. Postgres

```
FROM postgres:latest

COPY init-db.sh /docker-entrypoint-initdb.d/
```

e. Redis:

```
FROM redis:latest
```

4. Create `.env` file to store environment variables like database credentials.
5. Run `docker-compose up -d` to start all Containers in detached mode.

Horizontal scales

It is possible to run several Containers for a single service. To do it use `docker-compose up -d --scale radis=4` .