Container & Docker

Nguyễn Thanh Quân - ntquan@fit.hcmus.edu.vn

Agenda

---

1. Introduction to Docker
2. Managing Docker Images
3. Managing Docker Containers
4. Best Practices for Building Docker Images
5. Docker Registry
6. Docker Networking
7. Docker Volumes
8. Docker Compose
9. Docker Troubleshooting

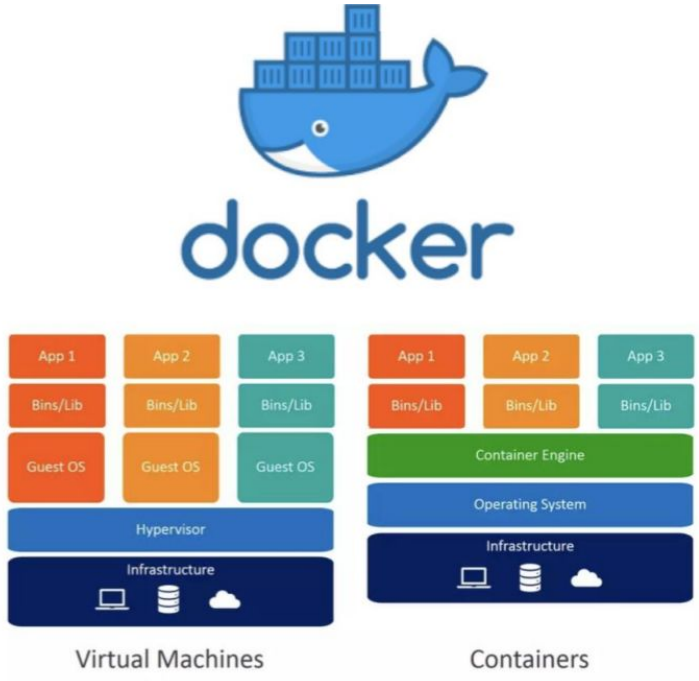
Introduction to Docker

Introduction to Docker

Docker is an open-source project that helps deploy Linux and Windows applications into virtualized containers.

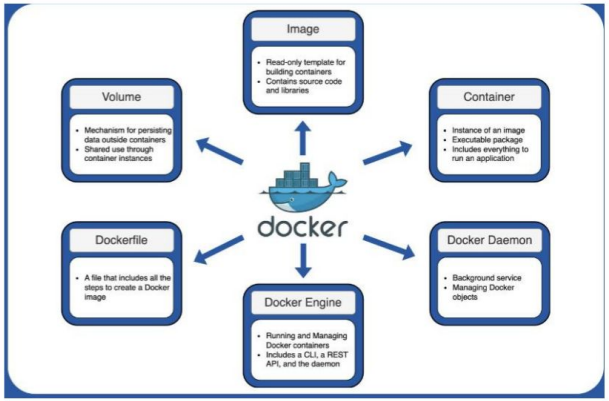
Advantages of Docker Containers:

- Containers are lighter and consume fewer resources compared to virtual machines.
- Fast startup time.
- Cost-effective deployment.



Main Components of Docker

1. Docker Engine: Docker Engine is the core component that runs Docker. It includes:
    a. Docker Daemon: Manages Docker objects such as containers, images, networks, and volumes. The daemon listens for requests from the Docker API and communicates with other components.
    b. Docker API: An interface that allows other tools or the Docker Client to interact with the Docker Daemon.
    c. Docker CLI (Client): A command-line interface that enables users to send commands to the Docker Daemon, such as docker run, docker build, and docker stop.



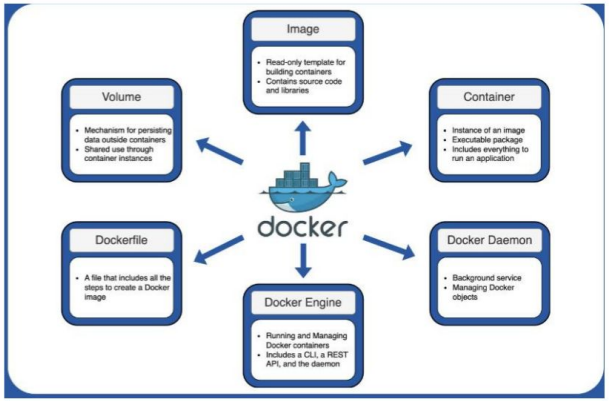
Main Components of Docker

---

2. Docker Container

How It Works:

a. A container is an active process running in an isolated environment.
b. Each container has its own file system, network resources, and execution environment.



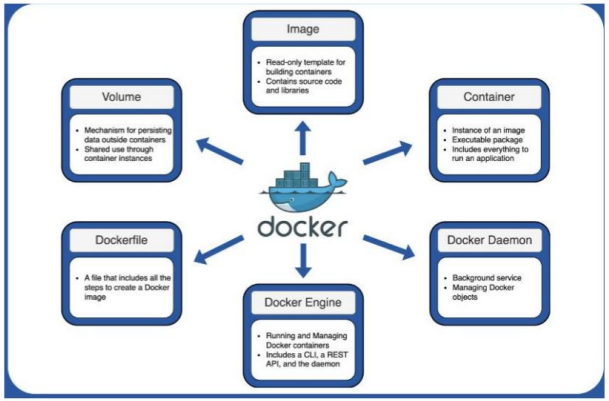
Main Components of Docker

---

3. Docker Image: A Docker Image is a read-only template used to create Docker Containers.

How Docker Images Are Created:

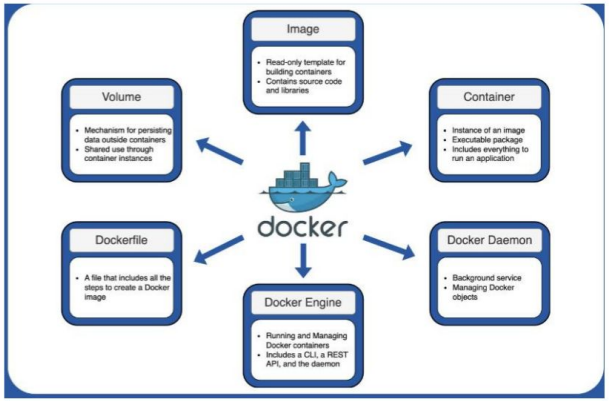
- Docker Images are built from a configuration file called a Dockerfile.
- An image includes everything needed to run an application, such as the operating system, source code, and libraries.



Main Components of Docker

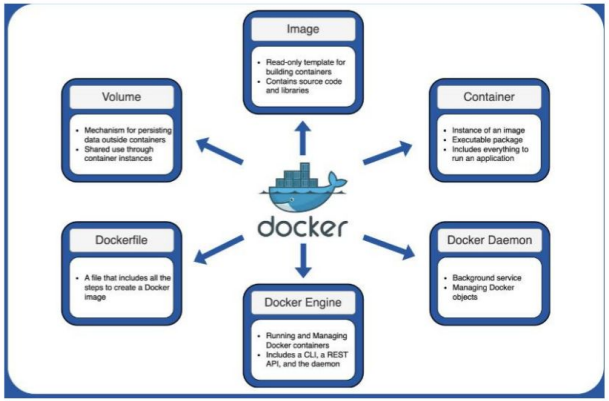
---

4. Docker Registry
    a. A Docker Registry is a storage system for Docker Images.
    b. Docker Hub is the most popular Docker Registry, providing many pre-built images for users.
    c. Users can also create a private Docker Registry to store and manage internal Docker Images.



Main Components of Docker

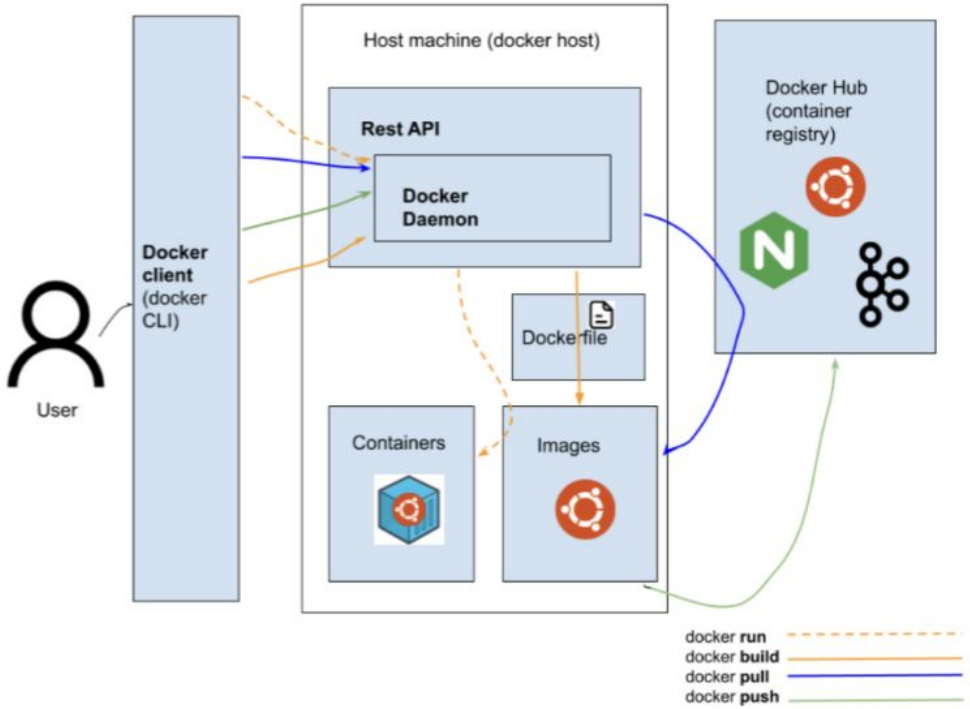
4. Docker Volumes
   a. A Volume is a storage solution for persistent data in Docker Containers.
   b. Volumes ensure that data is not lost when a container is stopped or deleted.
   c. Types of Docker Volumes:
      - Bind Mounts
      - Named Volumes
      - Anonymous Volumes



How Users Interact with Docker

---

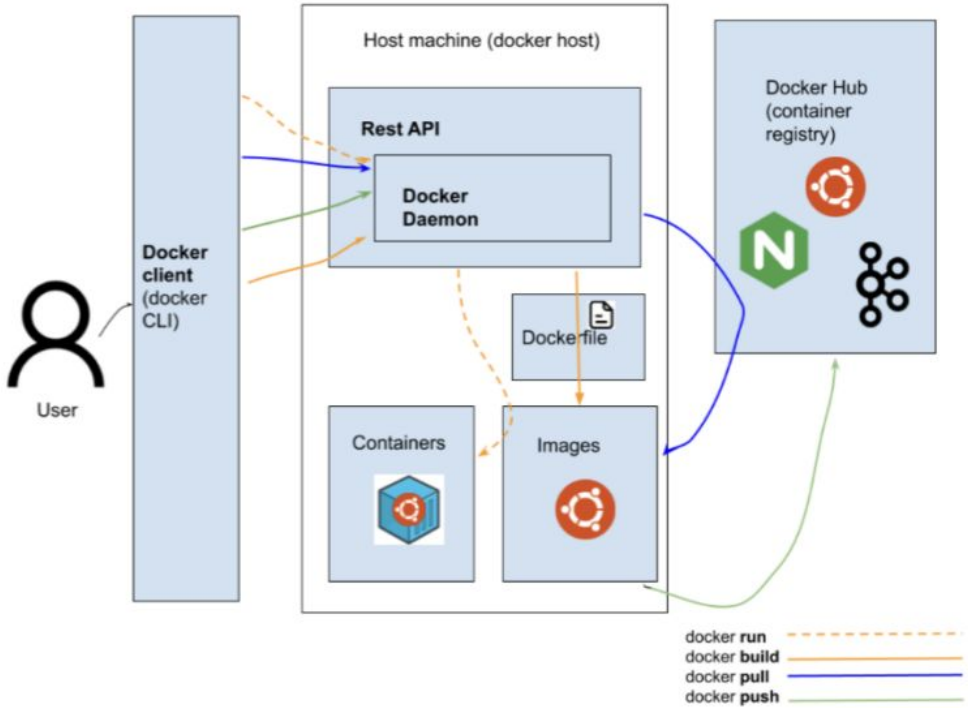
1. Users interact with Docker through the Docker CLI (Client). Commands such as docker run, docker build, and docker stop are sent from the client to the Docker Daemon via the Docker



How Users Interact with Docker

---

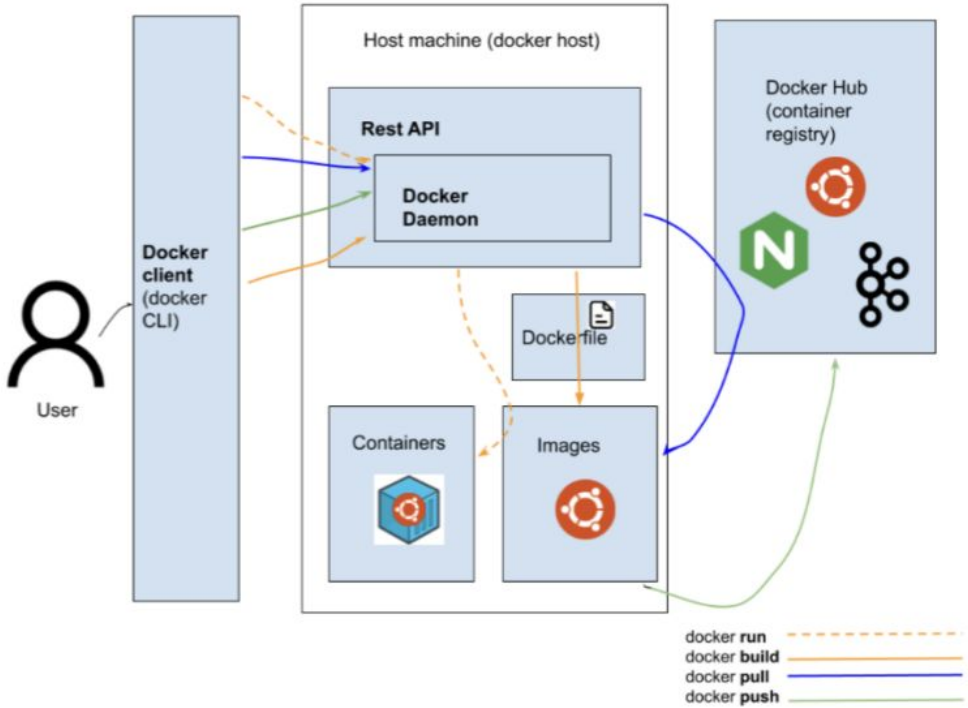
2. The Docker Daemon is responsible for handling these requests, such as:
    a. Creating containers from images.
    b. Connecting containers to networks.
    c. Managing volumes.



How Users Interact with Docker

---

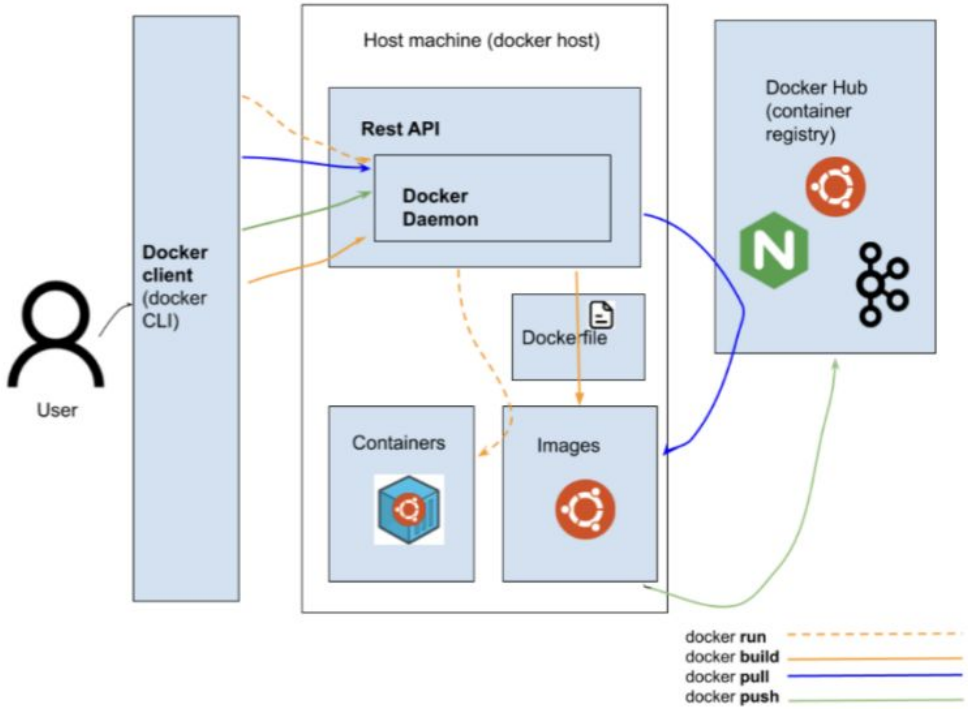
3. The Docker Daemon can pull Docker Images from a Docker Registry if the image is not available locally.
4. A container is created from a Docker Image and starts running the application packaged inside the image.



How Users Interact with Docker

---

5. A volume can be attached to a container to store data, ensuring that the data persists even when the container is deleted.



Process from Docker Image to Docker Container

1. Build Docker Image:

A Docker Image is built from a Dockerfile.

The Dockerfile defines how to install components, dependencies, and start the application.

2. Pull Docker Image from Registry:

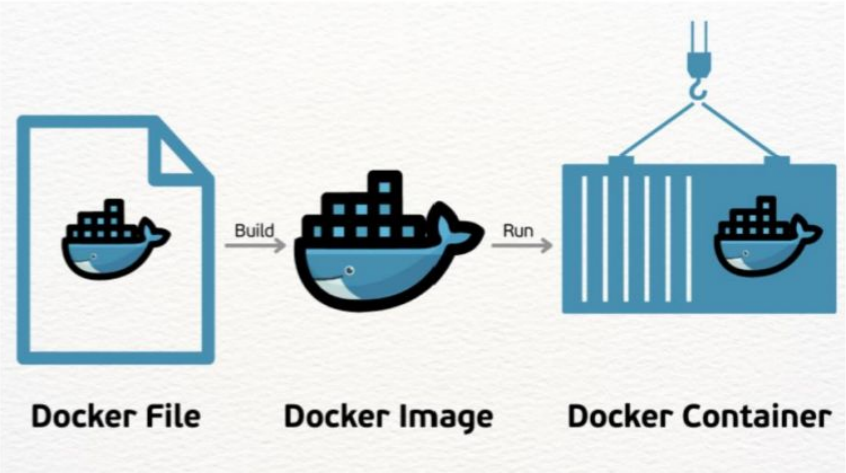
If the image already exists in a Docker Registry, the Docker Daemon can pull and store it locally.

3. Run Docker Container:

The Docker Daemon creates a new container from the image and starts the application inside the container.

4. Manage Containers:

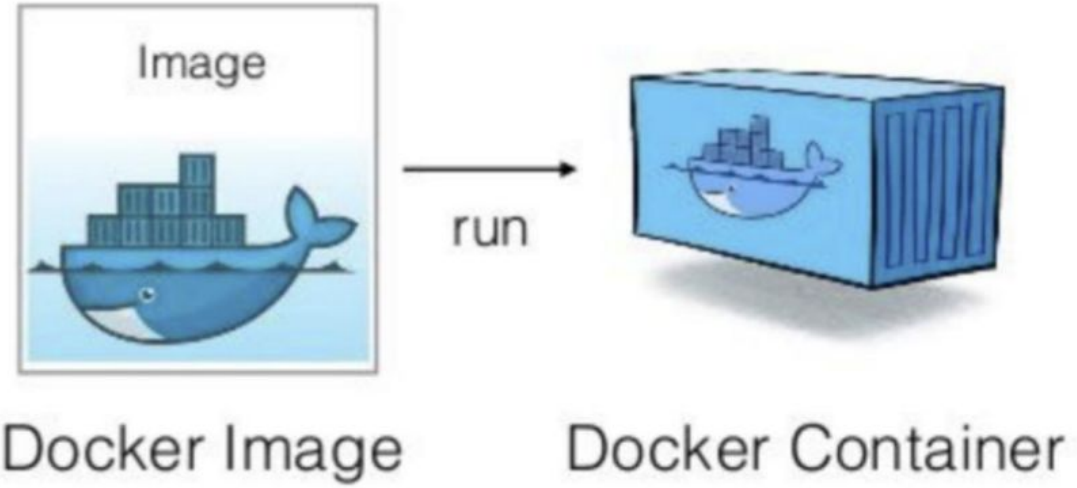
Users can use the Docker CLI to inspect, restart, or stop containers at any time.



Managing Docker Images

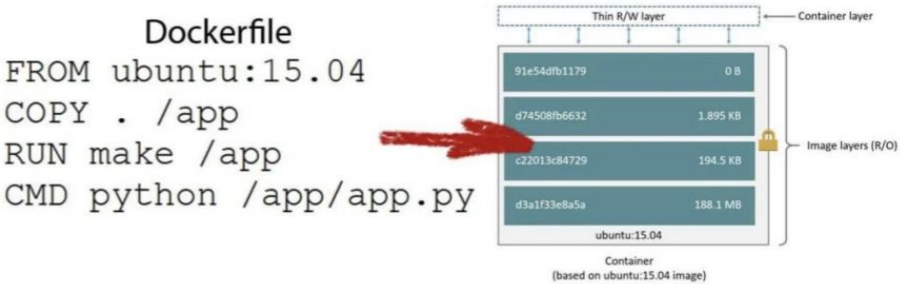
Managing Docker Images

A Docker Image is a read-only template used to create Docker Containers. It contains everything needed to run an application, including source code, libraries, and necessary dependencies.



Managing Docker Images

A Docker Image is composed of multiple layers. Each layer represents a change made during the image build process, such as installing software or copying files.



Managing Docker Images

Basic Docker Image Management Commands

- Pull a Docker Image: By default, Docker pulls images from Docker Hub.

  Example: Pull the Nginx image from Docker Hub:

  docker pull nginx

- List Downloaded Images:

  Show all images available locally:

  docker images

- Remove a Docker Image:

  Delete an image by its name or ID:

  docker rmi <image_name_or_id>

Creating and Managing a Dockerfile

A Dockerfile is a text file containing commands to build a Docker Image. It defines how Docker sets up the environment for an application.

Command	Description
FROM	Base image
RUN	Run command
COPY or ADD	Copy files from host to container
WORKDIR	Working directory
CMD or ENTRYPOINT	Start container
EXPOSE	Give port number will open
EVN	Set Environment for image

Build Docker Image from Dockerfile

---

docker build -t [image_name]:[tag] .

Example: ***docker build -t mynodeapp:1.0 .***

Explanation:

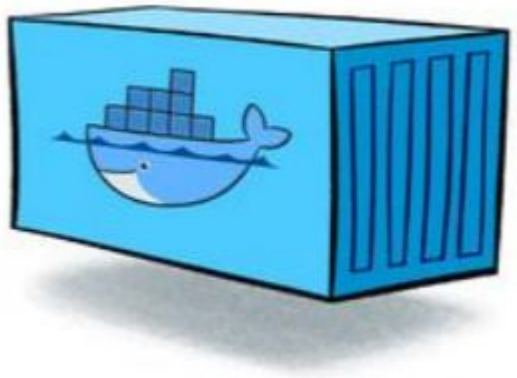
- -t mynodeapp:1.0 → Assigns a name and tag to the image (mynodeapp with version 1.0).
- . → Specifies the directory containing the Dockerfile (the current directory).

Managing Docker Containers

Managing Docker Containers

A Docker Container is a running instance of a Docker Image.

Each container includes everything needed to run an application, such as code, libraries, and environment variables.



Managing Docker Containers

Structure of a Docker Container: Each Docker Container consists of three main components:

- Docker Image: The base of the container. When a container starts, Docker pulls the corresponding image and creates a container from it.
- Write Layer (Writable Layer): When a container runs, Docker adds a write layer on top of the image’s read-only layers. Any changes made inside the container (e.g., file modifications) are stored in this layer.
- Execution Environment: The container has an isolated execution environment, including
  - IP address & networking
  - Filesystem
  - Resources (CPU, memory, etc.)

Managing Docker Containers

Run Docker Containers: docker run <image_name>

Example: ***docker run nginx***

Important flag when running container

-d : detached mode

--name: name of container

-p: mapping port between container and host

-v: create volume between container and host

--rm: delete container when finishing run

Example: Bạn có thể chạy Nginx trong chế độ nền (detached mode) và ánh xạ cổng 8080 trên host với cổng 80 trong container bằng lệnh sau:

docker run -d -p 8080:80 --name mynginx nginx

Managing Docker Containers

- List running containers: ***docker ps***
- List all containers (including stopped ones): ***docker ps -a***
- Stop a container: ***docker stop <container_name_or_id>***
- Start a container: ***docker start <container_name_or_id>***
- Delete a container: ***docker rm <container_name_or_id>***
- Delete stopped containers: ***docker container prune***
- Check log of container: ***docker logs <container_name_or_id>***

Managing Docker Containers

- Run a command inside a running container: ***docker exec -it <container_id_or_name> <command>***

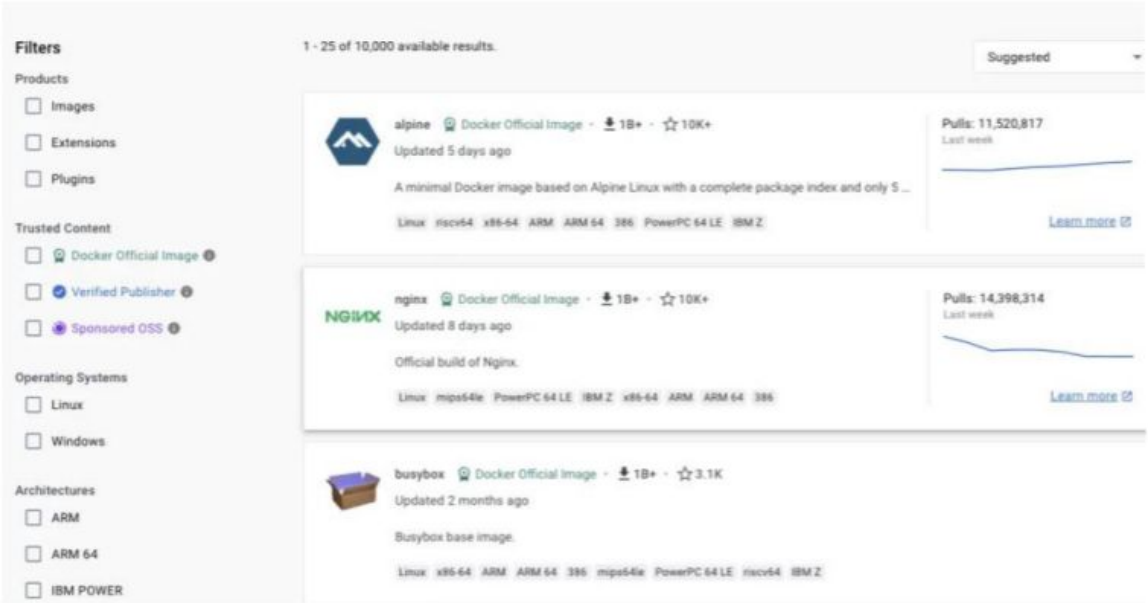
  Example: ***docker exec mynginx ls /usr/share/nginx/html***

- Open an interactive terminal inside a running container: ***docker exec -it <container_name_or_id> /bin/bash***

Build Docker Image Best Practice

Build Docker Image Best Practice

- Use Base Images from Trusted Sources



Build Docker Image Best Practice

- Use the Smallest Suitable Base Image

  Example: image ubuntu:alpine

- Use Multi-Stage Builds: Reduce final image size by separating build and runtime dependencies.
- Use “.dockerignore” to Exclude Unnecessary Files

Docker Registry

Docker Registry

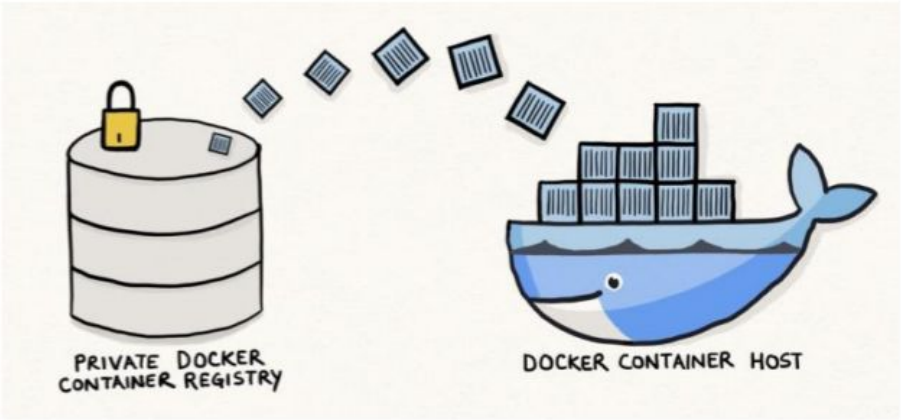
A Docker Registry is a storage system for Docker Images. It acts as a repository where images can be pushed (uploaded) and pulled (downloaded).

Common Uses of Docker Registry:

- Sharing and distributing images across teams and environments.
- Managing and versioning Docker Images in projects or organizations.
- Storing private or public images for deployment.

Examples of Docker Registries:

- Docker Hub (Public registry)
- Amazon Elastic Container Registry (ECR)
- Google Container Registry (GCR)
- Private Docker Registry (Self-hosted using registry image)



Docker Registry

Benefits of Docker Registry

- **Storage for Docker Images**: Enables storing created images for reuse or sharing across teams.
- **Optimized CI/CD Workflow**: Easily integrates with CI/CD pipelines to automate building and pushing images when source code changes.
- **Security and Privacy**: Using a Private Docker Registry ensures protection of sensitive images within an organization.
- **Version Management**: Allows storing and managing multiple versions of the same image for rollback or updates.

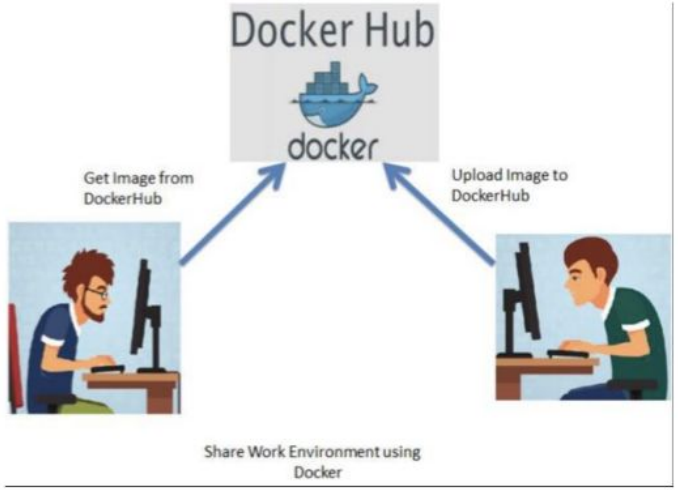
Docker Registry

Types of Docker Registry

- Docker Hub (Public Registry)
  - The official public Docker Registry.
  - Allows users to store and download Docker Images.
  - Contains:
  - Official Images (maintained by Docker and verified organizations).
  - Community Images (contributed by developers).
- Private Docker Registry
  - A self-hosted or organization-specific registry.
  - Provides secure storage for private images.
  - Used for internal image management and enhanced security.
  - Can be set up using the Docker Registry image:

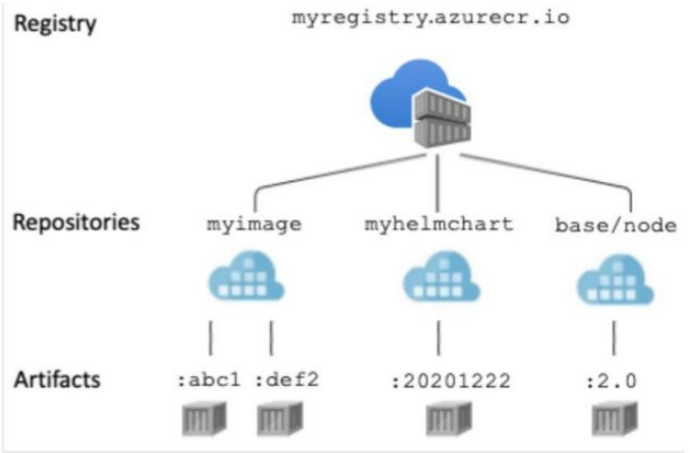
Docker Hub

- **Public**: Anyone can access and download public Docker Images from Docker Hub.
- **Community Sharing**: Docker Hub is widely used to share images with the community and open-source projects.
- **Official Images**: Docker Hub provides official, curated images from major organizations (e.g., MySQL, Redis, Nginx).



Docker Private Registry

- **Private**: Access requires authorization. Suitable for projects that demand high security.
- **Full Control**: Allows management of storage, distribution, and user access control for Docker Images.
- **Internal Optimization**: Reduces latency within internal networks and optimizes the upload/download process of Docker Images between internal systems.



Installing a Private Docker Registry

- Step 1: Pull the Docker Registry Image

  Download the official Docker Registry image from Docker Hub:

  docker pull registry:2

- Step 2: Run the Docker Registry Container

  Start a private Docker Registry on port 5000:

  docker run -d -p 5000:5000 --name my-registry registry:2

- Explanation:
- -d → Run in detached mode (background).
- -p 5000:5000 → Map port 5000 on the host to port 5000 in the container.
- --name my-registry → Assign a custom name to the container.
- registry:2 → Use version 2 of Docker Registry.

Push and Pull Docker Images from a Private Registry

- Tag a Docker Image to specify that it will be pushed to the Docker Registry:

  docker tag nginx localhost:5000/my-nginx:1.0

- Push the image to the Docker Registry:

  docker push localhost:5000/my-nginx:1.0

- Pull the image from the Docker Registry:

  docker pull localhost:5000/my-nginx:1.0

Check Docker Images in the Registry

Docker Registry stores data at /var/lib/registry inside the container.

Use the Registry HTTP API to check the existing repositories in the registry.

curl http://localhost:5000/v2/_catalog

List tags in a specific repository:

curl http://localhost:5000/v2/<repository_name>/tags/list

Example: ***curl http://localhost:5000/v2/myapp/tags/list***

Secure Docker Registry with Basic Authentication

- Create a password file using .htpasswd

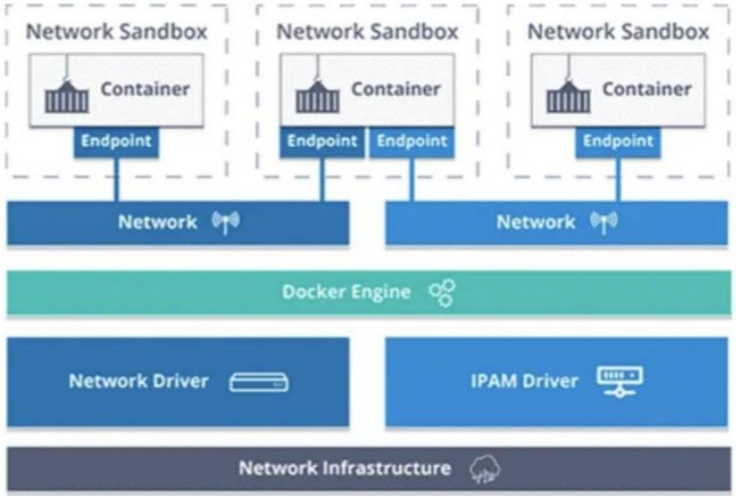
```
sudo apt-get install apache2-utils
htpasswd -Bc /auth/htpasswd myuser

docker run -d \
-p 5000:5000 \
--name registry \
-v /auth:/auth \
-e "REGISTRY_AUTH=htpasswd" \
-e "REGISTRY_AUTH_HTPASSWD_REALM=Registry
Realm" \
-e
"REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd" \
-e ”REGISTRY_STORAGE_DELETE_ENABLED=true” \
registry:2
```

Docker Network

Docker Network

Role of Networking in Docker

- Communication Between Containers: Containers on the same network can communicate with each other using container names as hostnames.
- Communication with External Systems: Containers can connect to the internet or interact with external services (e.g., databases, APIs).



Docker Network

Types of Networks in Docker

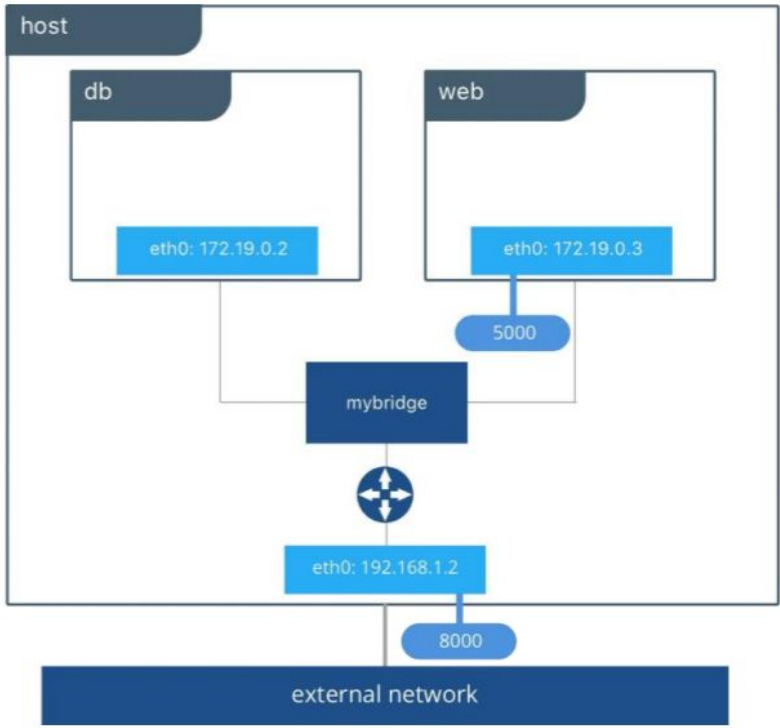
1. Bridge Network (Default)
2. Host Network
3. Overlay Network
4. Macvlan Network
5. None (No Network)

Bridge Network

- This is the default network type when running a container.
- Containers on the same bridge network can communicate with each other via container name or IP address.
- The bridge network does not allow containers to directly access the host's network.

Example:

docker run -d --name webserver nginx

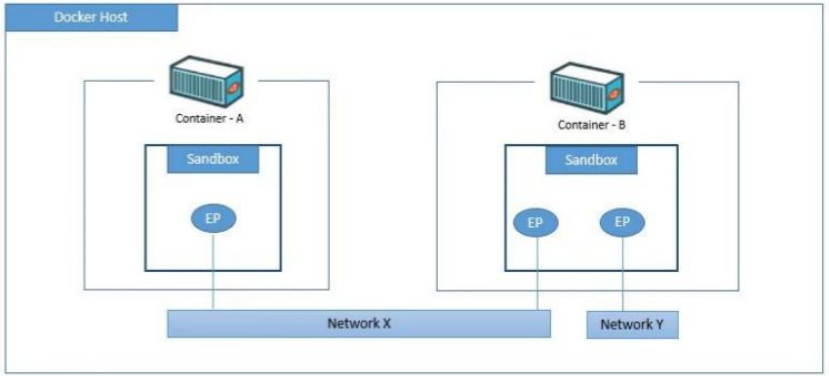


Host Network

- The container is directly attached to the host's network, allowing it to access the host network without going through NAT.
- When using the Host Network, the container shares the same IP address as the host.

Example:

docker run -d --name webserver --network host nginx



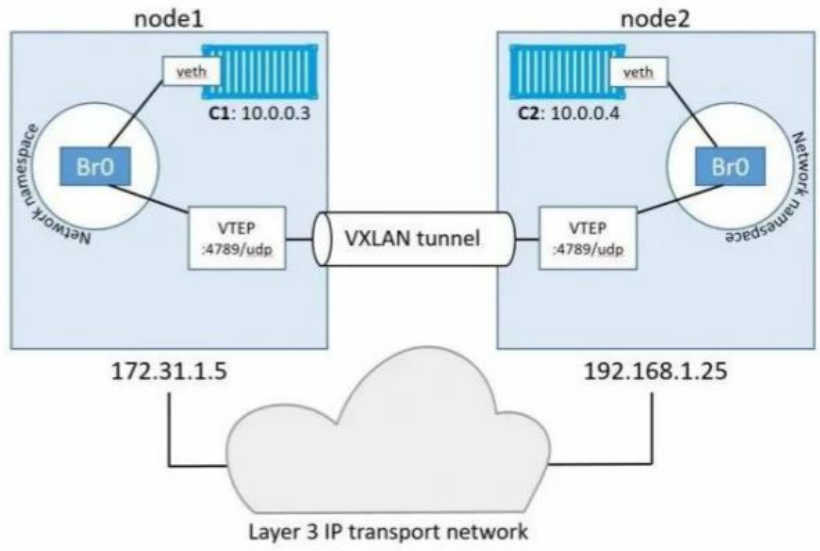
Overlay Network

- Overlay Network allows containers on different hosts to communicate with each other.
- It is commonly used in Kubernetes environments.

Example:

docker network create -d overlay my_overlay_network

docker network connect my_overlay_network my_container

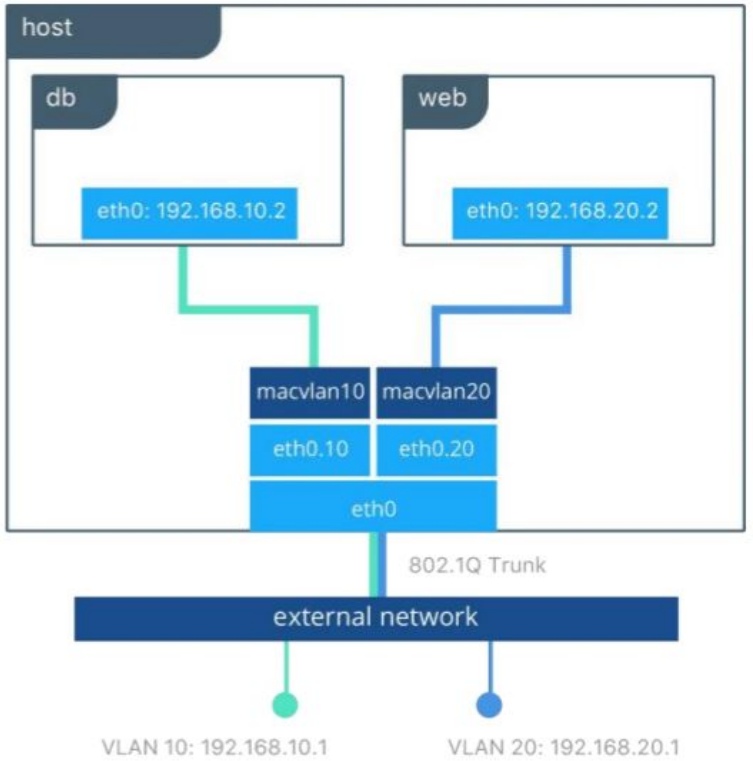


Macvlan Network

Macvlan Network allows assigning a separate MAC address to each container, creating a network environment where each container acts as an independent device on the physical network.

Example:

```
docker network create -d macvlan \
--subnet=192.168.1.0/24 \
--gateway=192.168.1.1 \
-o parent=eth0 macvlan_network
```



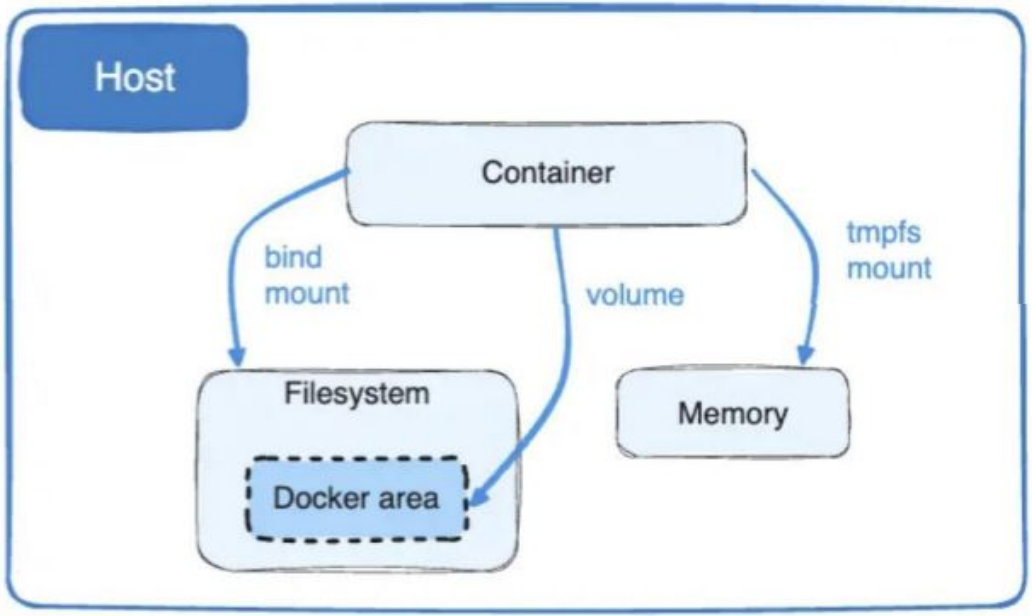
Basic Commands for Managing Docker Networks

- `docker network ls`
- `docker network create my_network`
- `docker network rm my_network`
- `docker network inspect my_network`
- `docker network connect my_network my_container`

Docker Volumes

Introduction

- Docker Volume is a method for persistently storing data related to containers.
- Data in a Volume is not deleted when a container stops or is removed, helping to protect important data.
- Volumes make it easy to manage and share data between containers.



Manage docker volumes

- `docker volume create my_volume`
- `docker volume ls`
- `docker run -d --name my_container -v my_volume:/data nginx`
- `docker volume inspect my_volume`
- `docker volume rm my_volume`

Docker compose

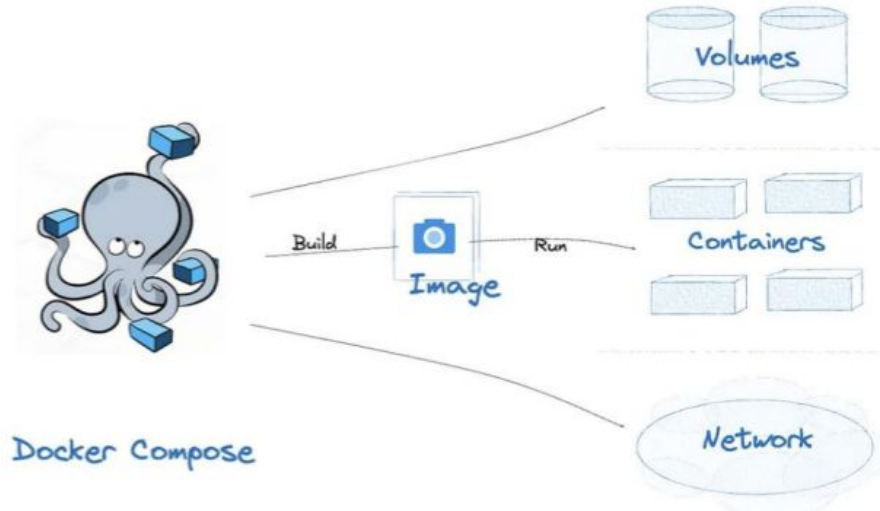
Introduction

- Docker Compose is a tool that allows you to define and manage multiple containers using a YAML configuration file.
- Instead of running each container with separate Docker commands, Docker Compose lets you define all the containers, networks, and volumes your application needs in a single file.

docker
Compose

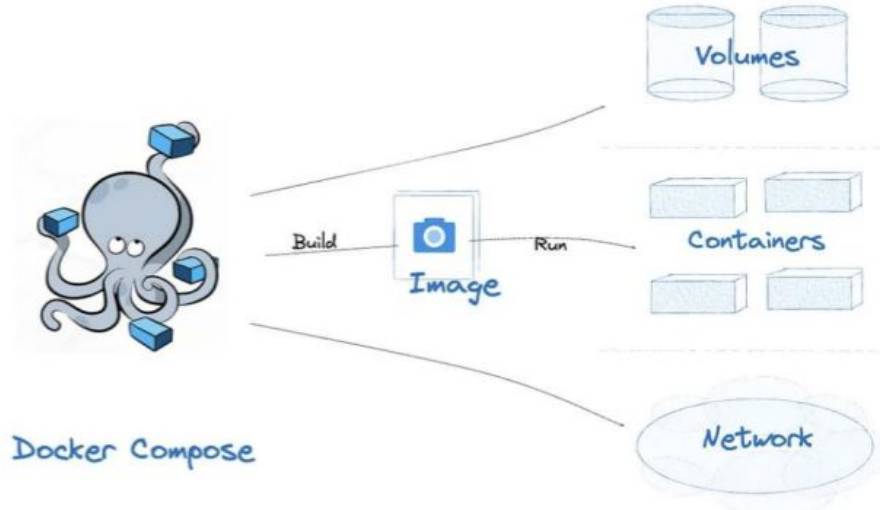
Benefits

- **Easier management of multiple containers**: Instead of starting each container manually, Docker Compose allows you to manage multiple containers at once with a single command.
- **Reusable configuration**: You can define your application's configuration in a YAML file, making it easy to share or reuse across different environments (development, staging, production).



Benefits

- **Easy network management:** Docker Compose automatically sets up a network between containers and ensures they can communicate with each other seamlessly.
- **Create volumes and temporary environments:** You can easily create volumes, networks, and configure environment variables when starting containers, making setup and teardown straightforward.



Main Components

- **services**: Defines the services (containers) in the application.
- **volumes**: Defines volumes for storing data outside of the container.
- **networks**: Defines networks that allow services to communicate with each other.
- **environment**: Sets environment variables for the container.
- **ports**: Configures port mapping from the container to the host.

Example

```
services:
  web:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "5000:5000"
    volumes:
      - .:/code
    environment:
      FLASK_ENV: development

  db:
    image: postgres:13
    volumes:
      - ./data:/var/lib/postgresql/data
```

Command line

- `docker compose config`
- `docker-compose up [-d]`
- `docker-compose down [--volumes]`
- `docker-compose build`
- `docker-compose up --build`
- `docker-compose ps`
- `docker-compose logs`
- `docker-compose stop`

Docker Container Trobleshooting

Docker Container Trobleshooting

- `docker inspect <container_name_or_id>`
- `docker stats <container_name_or_id>`
- `docker events`
- `docker system df`
- `docker logs <container_name_or_id>`

Thank you