DOCKER - Complete Guide

From Beginner to Advanced

Table of Contents

1. Introduction to Docker

1.1 What is Docker?

Docker is an **open platform** for developing, shipping, and running applications. It enables you to separate your applications from your infrastructure, allowing you to deliver software quickly and efficiently.

Key characteristics of Docker:

- Released in **March 2013** by Solomon Hykes and Sebastian Pahl
- Written in the **Go programming language**
- Platform-as-a-Service (PaaS) using OS-level virtualization
- Open-source and centralized platform for application deployment
- Performs OS-level virtualization (containerization)

1.2 Why Use Docker?

Docker solves the age-old problem of **"it works on my machine"** by providing consistent environments across development, testing, and production.

Benefit	Description
Consistency	Applications run the same way in all environments (dev, test, prod)
Portability	Easily move applications between different machines and cloud platforms
Efficiency	Lightweight containers use resources more effectively than VMs
Isolation	Run applications independently, avoiding dependency conflicts
Scalability	Scale applications easily using orchestration tools like Kubernetes
DevOps Ready	Perfect for CI/CD pipelines and modern DevOps practices

1.3 Advantages of Docker

No pre-allocation of RAM - Containers use only the memory they need

Continuous Integration Efficiency - Build once, deploy anywhere

Cost-Effective - Reduce infrastructure costs with efficient resource usage

Lightweight - Containers are much smaller than VMs

Image Reusability - Create once, use multiple times

Fast Creation - Containers start in seconds, not minutes

1.4 Disadvantages & Limitations

Not ideal for GUI applications - Docker focuses on backend services

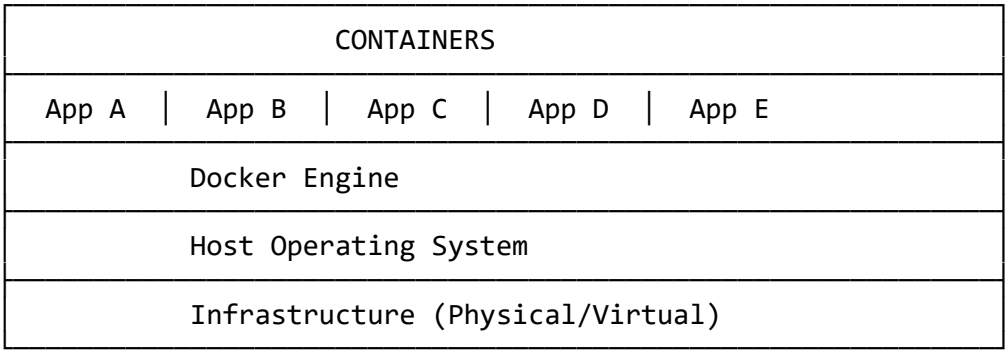
Container management complexity - Large deployments need orchestration

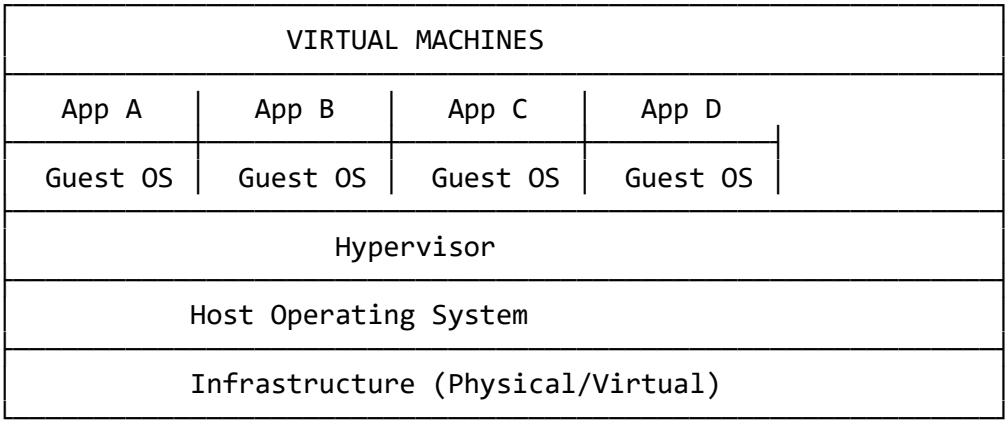
No cross-platform compatibility - Windows containers can't run on Linux and vice versa

Limited data recovery options - Requires proper volume management

1.5 Docker vs Virtual Machines

Understanding the difference is crucial:

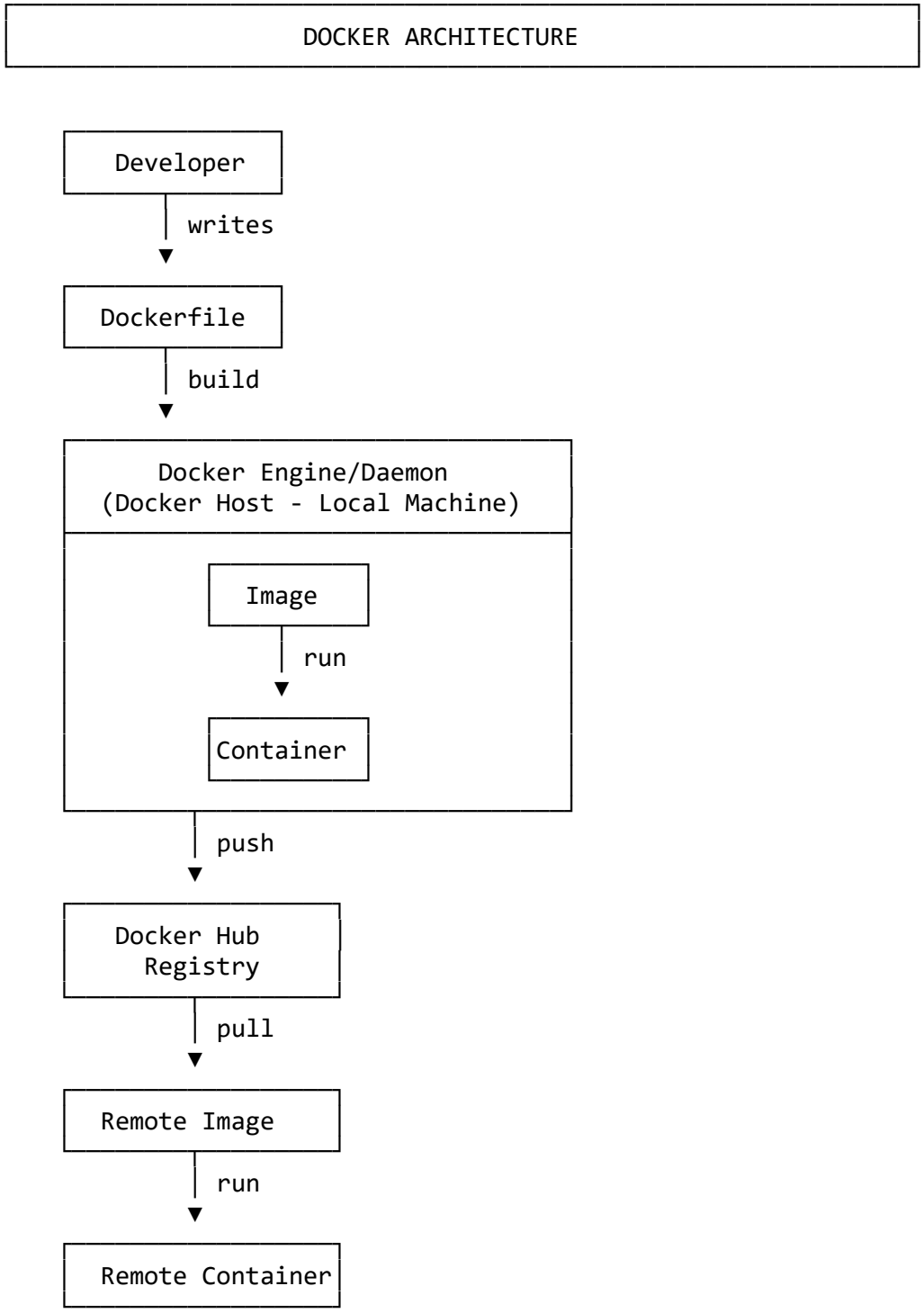




2. Docker Architecture & Components

2.1 Docker Architecture Overview

Docker uses a **client-server architecture**. The Docker client communicates with the Docker daemon, which handles building, running, and distributing containers.



2.2 Docker Components Explained

Docker Client

The Docker client is the primary interface for Docker users:

- **Command-line interface (CLI)** for user interaction
- Communicates via **REST API** with Docker daemon
- Sends commands (`docker run`, `docker build`, etc.) to daemon
- Can communicate with **multiple daemons**

How it works:

```
# User types command
$ docker run ubuntu

# Client sends command to daemon via REST API
# Daemon processes the command and returns result
```

Docker Daemon (Docker Engine)

The Docker daemon is the heart of the Docker system:

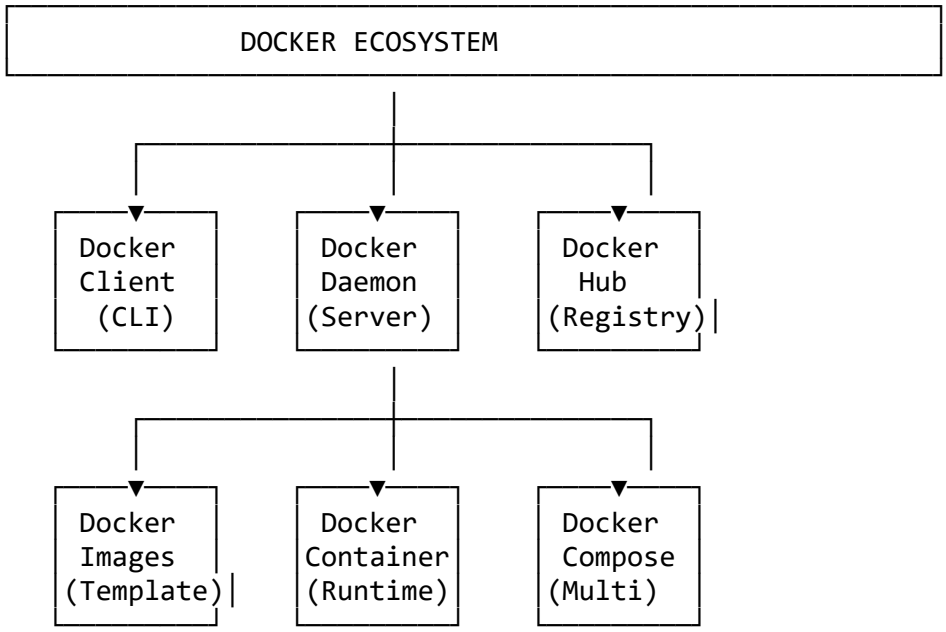
- Runs as a **background process** on the host OS
- Manages Docker objects (images, containers, networks, volumes)
- Responsible for **building, running, and distributing** containers
- Can communicate with other daemons for distributed systems

Docker Registry (Docker Hub)

Docker registries store and distribute Docker images:

- **Public Registry (Docker Hub):** Free cloud-based registry for public images
- **Private Registry:** Enterprise-level private image storage
- Stores **multiple versions** of images with tags
- Enables image **sharing** across teams and organizations

2.3 Docker Ecosystem



Docker Images

Docker images are **read-only templates** that contain everything needed to run an application:

- Lightweight, standalone packages
- Include code, runtime, libraries, and system tools
- Built in **layers** for efficiency and reusability
- **Immutable** - never changed after creation

Three ways to create Docker images:

1. **Pull from Docker Hub** (pre-built images)
2. **Create from Dockerfile** (build custom images)
3. **Commit from existing container** (save container state)

Docker Containers

Containers are **running instances** of Docker images:

- Portable, self-contained execution environments
- Pack code, runtime, libraries, and tools together
- **Isolated** from host system and other containers
- Share the host OS kernel (no separate OS per container)

Container characteristics: - Start in **seconds** - Use **minimal resources** - Can be **stopped, started, deleted** independently - **Ephemeral** by default (data lost when deleted)

Docker Compose

Docker Compose is a tool for defining and running **multi-container** applications:

- Uses **YAML files** for configuration
- Manages services, networks, and volumes together
- Simplifies **complex application deployments**
- Perfect for **microservices architectures**

Example use case: Running WordPress with MySQL

```yaml
services:
  wordpress:
    image: wordpress
    ports:
      - "8080:80"
  mysql:
    image: mysql
    environment:
      MYSQL_ROOT_PASSWORD: password
```

3. Installing Docker

3.1 Installation on Windows & macOS

Docker Desktop provides the easiest installation experience, including the Docker Engine, CLI, and Docker Compose.

Steps:

1. Download Docker Desktop from https://www.docker.com/products/docker-desktop/
2. Run the installer (`.exe` for Windows, `.dmg` for macOS)
3. **Windows only:** Ensure **WSL 2** is installed and enabled
4. Start Docker Desktop application
5. Verify installation: `docker --version`

3.2 Installation on Linux (Ubuntu/Debian)

For Linux distributions, install Docker Engine from the official repository:

```
# Update package index
sudo apt update

# Install prerequisite packages
sudo apt install apt-transport-https ca-certificates \
  curl software-properties-common -y

# Add Docker's official GPG key
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
  sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg

# Add Docker repository
echo "deb [arch=$(dpkg --print-architecture) \
  signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] \
  https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

# Install Docker Engine
sudo apt update
sudo apt install docker-ce docker-ce-cli containerd.io -y
```

3.3 Post-Installation Steps

Verify Installation

```
# Check Docker version
docker --version
# or
docker -v

# Check Docker service status
sudo service docker status

# Test Docker with hello-world
sudo docker run hello-world
```

3.4 Installation notes

```
# Manage Docker as a non-root user
# 1. Create the docker group (usually already exists)
sudo groupadd docker

# 2. Add your user to the group
sudo usermod -aG docker $USER

# 3. Apply the changes (or log out and back in)
newgrp docker
```

Configure Non-Root Access (Linux)

To run Docker commands without sudo:

```
# Add your user to the docker group
sudo usermod -aG docker $USER

# Log out and log back in for changes to take effect
# Verify with:
docker ps
```

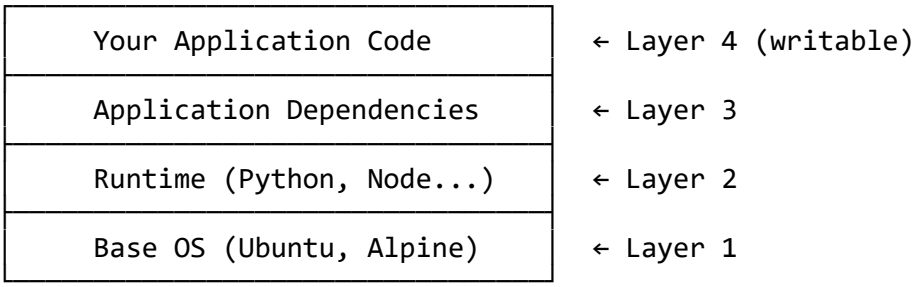
Security Note: Adding users to the docker group grants privileges equivalent to root access.

4. Docker Images

4.1 Understanding Docker Images

A Docker image is a **lightweight, standalone, executable package** that includes everything needed to run a piece of software. Images are the building blocks of containers and serve as blueprints for creating container instances.

Image Layered Architecture:



Key characteristics:

- **Read-only templates** - Once created, images don't change
- **Layered architecture** - Built in layers for efficiency
- **Portable** - Can be shared and distributed easily
- **Versioned** - Support tags for different versions

4.2 Working with Images

Searching for Images

```
# Search Docker Hub for images
docker search <image_name>

# Example: Search for Ubuntu images
docker search ubuntu

# Search with filter (minimum 100 stars)
docker search --filter stars=100 ubuntu
```

Pulling Images

```
# Pull an image from Docker Hub
docker pull <image_name>

# Example: Pull Ubuntu image (latest version)
docker pull ubuntu

# Pull specific version with tag
docker pull ubuntu:20.04

# Pull from specific registry
docker pull myregistry.com/myimage:v1.0
```

Common Image Tags: - `latest` - Most recent stable version (default) - `20.04`, `22.04` - Specific version numbers - `alpine` - Minimal, lightweight version - `slim` - Reduced size version

Listing Images

```
# List all images on your system
docker images

# Alternative command
docker image ls

# Show all images including intermediate
docker images -a

# Filter images by name
docker images ubuntu

# Format output
docker images --format "table {{.Repository}}\t{{.Tag}}\t{{.Size}}"
```

Output columns explained: - `REPOSITORY` - Image name - `TAG` - Version/variant - `IMAGE ID` - Unique identifier - `CREATED` - When image was created - `SIZE` - Disk space used

Creating Images

You can create Docker images using two methods: capturing a running container (`commit`) or building from a recipe file (`build`).

A. Building from a Dockerfile (Professional/Standard Way) This is the preferred method as it is repeatable, version-controlled, and transparent.

```
# Build an image from a Dockerfile in the current directory
docker build -t <image_name> .

# Build with a specific tag
docker build -t <username>/<repository>:<tag> .
```

```
# Example:
docker build -t my-python-app:v1.0 .

# Build using a specific file (if not named 'Dockerfile')
docker build -f MyCustomFile -t my-app .
```

B. Creating from a Container (Snapshot/Commit Way) Useful for quick debugging or saving the state of a manually configured container.

```
# Create image from existing container
docker commit <container_name> <new_image_name>

# Example
docker commit mycontainer myimage

# With tag
docker commit mycontainer myimage:v1.0

# With author and message
docker commit -a "John Doe" -m "Added manual configuration" mycontainer
myimage
```

Inspecting Images

```
# View detailed image information
docker inspect <image_name>

# View image history (layers)
docker history <image_name>

# View specific field
docker inspect --format='{{.Config.Env}}' ubuntu
```

Removing Images

```
# Remove a single image
docker rmi <image_name>

# Remove by ID
docker rmi abc123def456

# Force remove (even if containers exist)
docker rmi -f <image_name>

# Remove all unused images
docker image prune

# Remove all images (dangerous!)
docker rmi $(docker images -q)
```

4.3 Image Naming Convention

```
[registry/][username/]repository[:tag]

Examples:
ubuntu                          # Official image, latest tag
ubuntu:20.04                    # Official image, specific tag
nginx:alpine                    # Official image, alpine variant
myusername/myapp:v1.0          # User image with tag
registry.example.com/app:prod   # Private registry
```

5. Docker Containers

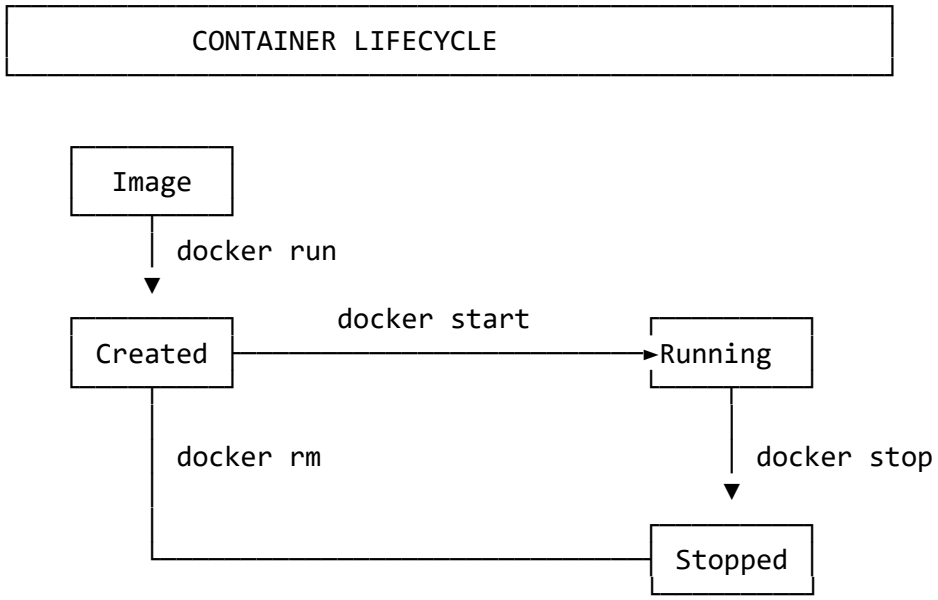
5.1 Understanding Containers

Containers are **running instances** of Docker images. They provide isolated, lightweight execution environments that include everything needed to run an application.

Container vs. Virtual Machine:

Aspect	Container	Virtual Machine
OS	Shares host OS kernel	Full OS per VM
Size	MBs	GBs
Startup Time	Seconds	Minutes
Resource Usage	Minimal	Heavy
Isolation	Process-level	Hardware-level
Performance	Near-native	Overhead from hypervisor

5.2 Container Lifecycle



5.3 Creating and Running Containers

Basic Container Creation

```
# Create and start container (interactive mode)
docker run -it <image_name> /bin/bash

# Example with Ubuntu
docker run -it ubuntu /bin/bash

# You're now inside the container!
root@abc123:/# whoami
root
root@abc123:/# cat /etc/os-release
# Shows Ubuntu version

# Exit container
root@abc123:/# exit
```

Create Container with Custom Name

```
# Create container with specific name
docker run -it --name mycontainer ubuntu /bin/bash

# Good practice: Use descriptive names
docker run -it --name web-frontend nginx /bin/bash
docker run -it --name db-mysql mysql /bin/bash
```

Create and Run in Detached Mode

```
#Create and Run in Detached Mode

To run a container in the background without it taking over your terminal,
use Detached Mode.


# Run a basic OS container in background
docker run -td --name my-linux ubuntu

# Run a web server in background (most common usage)
docker run -d --name webserver -p 8080:80 nginx
```

Flag Breakdown:

- `-d` (**Detached**): Runs the container in the background. Your terminal stays free.
- `-t` (**TTY**): Allocates a "pseudo-terminal." It keeps the container "alive" even if no process is active.
- `-i` (**Interactive**): Usually paired with `-t` (`-it`) to allow you to interact with the shell.

Why add the -i?

If you use `-td` with `ubuntu`, the container starts and stays alive. However, if you ever want to "go inside" that running container later, you will need the **Interactive** (`-i`) flag to have been set, or use it with the `exec` command:

```
# Go inside the running detached container
docker exec -it webserver bash
```

5.4 Container Management Commands

Listing Containers

```
# List running containers
docker ps

# List all containers (running and stopped)
docker ps -a

# Show only container IDs
docker ps -q

# Custom format
docker ps --format "table {{.Names}}\t{{.Status}}\t{{.Ports}}"

# Filter by status
docker ps -a --filter "status=exited"
docker ps -a --filter "name=web"
```

Starting, Stopping, and Restarting

```
# Start a stopped container
docker start <container_name>

# Stop a running container (graceful shutdown)
docker stop <container_name>

# Kill a container (immediate termination)
docker kill <container_name>

# Restart a container
docker restart <container_name>

# Pause/unpause container
docker pause <container_name>
docker unpause <container_name>
```

Accessing Running Containers

```
# Attach to a running container
docker attach <container_name>
```

```
# Execute command in running container (preferred)
docker exec -it <container_name> /bin/bash

# Run single command without entering
docker exec <container_name> ls -la /var/log

# Run command as specific user
docker exec -u root <container_name> whoami
```

** Key Difference:** - `attach` - Connects to main process (exit = container stops) - `exec` - Creates new process (exit = new process stops, container continues)

5.5 Inspecting and Monitoring

```
# View detailed container information
docker inspect <container_name>

# View container logs
docker logs <container_name>

# Follow logs in real-time
docker logs -f <container_name>

# Show last 100 lines
docker logs --tail 100 <container_name>

# Show logs with timestamps
docker logs -t <container_name>

# View real-time resource usage stats
docker stats <container_name>

# Stats for all running containers
docker stats

# See changes made to container filesystem
docker diff <container_name>
# A = Added file
# D = Deleted file
# C = Changed file

# View running processes in container
docker top <container_name>
```

5.6 Copying Files Between Host and Container

```
# Copy from host to container
docker cp /path/on/host <container_name>:/path/in/container

# Copy from container to host
```

```
docker cp <container_name>:/path/in/container /path/on/host

# Example: Copy config file to container
docker cp config.yaml mycontainer:/etc/app/config.yaml

# Example: Extract logs from container
docker cp mycontainer:/var/log/app.log ./app.log
```

5.7 Removing Containers

```
# Remove a stopped container
docker rm <container_name>

# Force remove a running container
docker rm -f <container_name>

# Remove multiple containers
docker rm container1 container2 container3

# Remove all stopped containers
docker container prune

# Remove all containers (dangerous!)
docker rm -f $(docker ps -a -q)

# Remove containers after they exit (auto-cleanup)
docker run --rm ubuntu echo "This container will auto-delete"
```

5.8 Container Resource Limits

```
# Limit memory
docker run -m 512m ubuntu

# Limit CPU
docker run --cpus=".5" ubuntu  # 50% of one CPU

# Limit both
docker run -m 1g --cpus="1.5" ubuntu

# Set priority
docker run --cpu-shares=512 ubuntu

# Limit disk I/O
docker run --device-write-bps /dev/sda:1mb ubuntu
```

6. Docker Volumes

6.1 Understanding Docker Volumes

Docker volumes are **specialized directories** that containers can use to persist and share data. Unlike container filesystems, volumes persist even after containers are deleted.

The Problem:

```
Container Created → Data Written → Container Deleted →  DATA LOST!
```

The Solution:

```
Container Created → Data Written to Volume → Container Deleted →  DATA
PERSISTS!
```

Think of volumes as: - Shared folders between your host and containers - Persistent storage that survives container deletion - Data sharing mechanism between multiple containers

6.2 Volume Characteristics

Volume is a **directory inside the container**

Must be **declared when creating** the container

Can be **shared across multiple containers**

Data **persists** even when container is stopped or deleted

Not included when creating images from containers

Changes to volumes are made **directly** (not during image build)

6.3 Benefits of Using Volumes

Benefit	Description
Decoupling	Separate container from storage
Sharing	Share data among multiple containers
Persistence	Data survives container lifecycle
Flexibility	Easily backup and migrate data
Performance	Better I/O performance than container layers

6.4 Types of Volume Mapping

1. Container-to-Container Volume Sharing

```
Container 1 (Volume: /myvolume)
    ↓
    └─── Creates files in /myvolume

Container 2 (--volumes-from Container1)
    ↓
    └─── Can access same files in /myvolume
```

2. Host-to-Container Volume Mapping

```
Host Machine (/home/user/data)
    ↓
    └─── Maps to → Container (/app/data)
```

Changes in either location reflect in both!

6.5 Creating Volumes via Dockerfile

Step 1: Create a Dockerfile with VOLUME instruction

```
FROM ubuntu
VOLUME ["/myvolume1"]
```

Step 2: Build image from Dockerfile

```
docker build -t volumeimage .
```

Step 3: Create container from the image

```
# Create container with volume
docker run -it --name volcontainer1 volumeimage /bin/bash

# Inside container, navigate to volume
root@abc:/# cd myvolume1
root@abc:/myvolume1# ls

# Create some files
root@abc:/myvolume1# touch file1 file2 file3
root@abc:/myvolume1# echo "Hello Docker" > file1

# Exit container
root@abc:/myvolume1# exit
```

6.6 Sharing Volumes Between Containers

```
# Create second container sharing volume from first
docker run -it --name volcontainer2 \
  --privileged=true \
  --volumes-from volcontainer1 \
  ubuntu /bin/bash

# Inside volcontainer2
root@xyz:/# cd myvolume1
root@xyz:/myvolume1# ls
file1  file2  file3

# Read file created by volcontainer1
root@xyz:/myvolume1# cat file1
Hello Docker

# Create new file
root@xyz:/myvolume1# touch file4

# Exit
root@xyz:/myvolume1# exit
```

```
# Go back to volcontainer1
docker start volcontainer1
docker attach volcontainer1

# You'll see file4 created by volcontainer2!
root@abc:/# cd myvolume1
root@abc:/myvolume1# ls
file1  file2  file3  file4
```

Real-world use case: Log sharing between application and monitoring containers

6.7 Creating Volumes via Command Line

```
# Create container with volume (no Dockerfile needed)
docker run -it --name volcontainer3 -v /myvolume2 ubuntu /bin/bash

# Create container with named volume
docker run -it --name volcontainer4 -v mydata:/data ubuntu /bin/bash

# Multiple volumes
docker run -it --name volcontainer5 \
  -v /volume1 \
  -v /volume2 \
  -v /volume3 \
  ubuntu /bin/bash
```

6.8 Host-to-Container Volume Mapping

```
# Syntax: -v /host/path:/container/path
docker run -it --name hostcontainer \
  -v /home/ec2-user:/container \
  --privileged=true \
  ubuntu /bin/bash

# Inside container
root@abc:/# cd /container
root@abc:/container# ls
# You'll see files from /home/ec2-user

# Create file in container
root@abc:/container# echo "Created in container" > test.txt
root@abc:/container# exit

# Check on host machine
$ cd /home/ec2-user
$ cat test.txt
Created in container
```

Real-world examples:

```
# Mount code directory for development
docker run -v $(pwd):/app node:14 npm start

# Mount config file
docker run -v /etc/myapp/config.yml:/app/config.yml myapp

# Mount multiple directories
docker run \
  -v /host/code:/app/code \
  -v /host/logs:/app/logs \
  -v /host/data:/app/data \
  myapp
```

6.9 Volume Management Commands

```
# List all volumes
docker volume ls

# Create a named volume
docker volume create myvolume

# Create volume with driver options
docker volume create --driver local \
  --opt type=nfs \
  --opt o=addr=192.168.1.1,rw \
  --opt device=:/path/to/dir \
  myvolume

# Inspect volume details
docker volume inspect myvolume

# Remove a volume
docker volume rm myvolume

# Remove all unused volumes
docker volume prune

# Remove volume with confirmation
docker volume rm myvolume
# Error if in use!

# Force remove (not recommended)
docker rm -v <container_name>  # Removes container and its volumes
```

6.10 Volume Inspection Output

```
$ docker volume inspect myvolume

[
    {
        "CreatedAt": "2025-01-15T10:30:00Z",
```

```
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/myvolume/_data",
        "Name": "myvolume",
        "Options": {},
        "Scope": "local"
    }
]
```

Key fields: - `Mountpoint` - Actual location on host filesystem - `Driver` - Storage driver (local, nfs, etc.) - `Scope` - Visibility (local or global in Swarm)

6.11 Read-Only Volumes

```
# Mount volume as read-only
docker run -v /host/data:/container/data:ro ubuntu

# Container can read but not write
root@abc:/# cd /container/data
root@abc:/container/data# touch file.txt
touch: cannot touch 'file.txt': Read-only file system
```

6.12 Volume Best Practices

Use named volumes for important data

```
docker volume create app-data
docker run -v app-data:/data myapp
```

Use bind mounts for development

```
docker run -v $(pwd):/app node:14
```

Backup volumes regularly

```
docker run --rm -v myvolume:/data -v $(pwd):/backup ubuntu \
  tar czf /backup/backup.tar.gz /data
```

Clean up unused volumes

```
docker volume prune -f
```

Don't store volumes in container layers

Don't rely on default anonymous volumes for important data

7. Docker Networking & Port Mapping

7.1 Understanding Port Mapping

Port mapping allows you to **access services** running inside Docker containers from outside the container. You map a port on your host machine to a port inside the container.

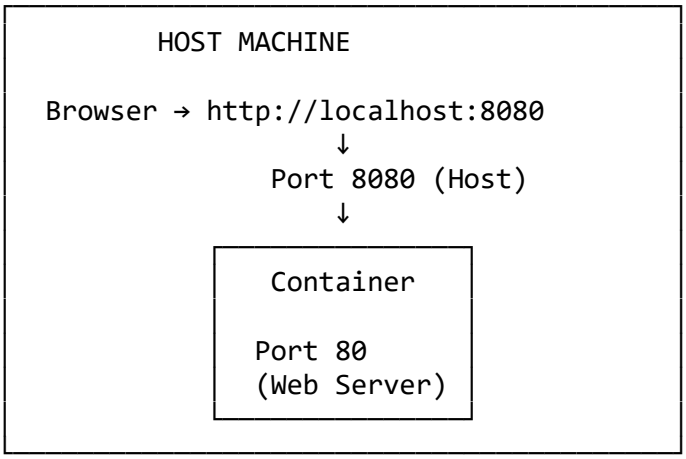
The Problem:

```
Container runs web server on port 80
↓
But you can't access it from outside!
```

The Solution:

```
Map Host Port 8080 → Container Port 80
↓
Access via http://localhost:8080 → Goes to container port 80
```

How it works:



7.2 Creating Containers with Port Mapping

```
# Syntax: -p host_port:container_port
docker run -td --name webserver -p 80:80 ubuntu

# Map to different host port
docker run -td --name webapp -p 8080:80 nginx

# Multiple port mappings
docker run -td --name multiport \
  -p 80:80 \
  -p 443:443 \
  -p 3000:3000 \
  myapp

# Map to specific interface
docker run -td -p 127.0.0.1:8080:80 nginx

# Random host port (Docker assigns)
docker run -td -p 80 nginx

# UDP port
docker run -td -p 53:53/udp dns-server
```

```
# Both TCP and UDP
docker run -td -p 53:53/tcp -p 53:53/udp dns-server
```

Flags explained:

- -t = Allocate pseudo-TTY
- -d = Detached mode (run in background)
- -p = Publish/expose port

7.3 Checking Port Mappings

```
# View ports mapped on a container
docker port <container_name>

# Example
$ docker port webserver
80/tcp -> 0.0.0.0:80
80/tcp -> :::80

# View in ps output
$ docker ps
CONTAINER ID   IMAGE   PORTS                     NAMES
abc123def456   nginx   0.0.0.0:8080->80/tcp    webapp
```

7.4 Practical Example: Apache Web Server

```
# Step 1: Create container with port 80 mapped
docker run -td --name apache -p 80:80 ubuntu

# Step 2: Enter the container
docker exec -it apache /bin/bash

# Step 3: Install and configure Apache
root@abc:/# apt-get update
root@abc:/# apt-get install apache2 -y

# Step 4: Start Apache
root@abc:/# service apache2 start

# Step 5: Optional - Create custom page
root@abc:/# echo "<h1>Hello from Docker!</h1>" > /var/www/html/index.html

# Step 6: Exit container
root@abc:/# exit

# Step 7: Access from browser
# Open: http://localhost:80 or http://your-ip:80
```

7.5 Practical Example: Jenkins Server

```
# Run Jenkins on port 8080
docker run -td --name jenkinsServer -p 8080:8080 jenkins/jenkins
```

```
# Access Jenkins
# Open: http://localhost:8080

# Get initial admin password
docker exec jenkinsServer cat /var/jenkins_home/secrets/initialAdminPassword
```

7.6 EXPOSE vs PUBLISH (-p)

Understanding the difference is **crucial**:

Configuration	Access Level	Use Case
Neither EXPOSE nor -p	Container only	Internal services
EXPOSE only	Other containers	Inter-container communication
EXPOSE + -p	Public access	Web servers, APIs
-p only	Public + containers	Implicit EXPOSE

EXPOSE in Dockerfile

```
FROM ubuntu
EXPOSE 80
# Documented but not published
# Other containers can access via container network
# External access requires -p flag at runtime
```

PUBLISH with -p

```
# Publishes port to host
docker run -p 8080:80 nginx

# Makes port accessible:
#  From host machine
#  From other containers
#  From external network
```

7.7 Network Driver Description

bridge: Default network. Containers can talk to each other if they are on the same bridge. host: Container uses the host's network directly (no isolation). Fastest performance. none: Container has no network access. Maximum isolation.

Example Scenarios

Scenario 1: Database (internal only)

```
FROM postgres
EXPOSE 5432
# No -p flag when running
# Only other containers can connect
```

Scenario 2: Web API (public)

```
docker run -p 3000:3000 myapi
# Accessible from anywhere
```

Scenario 3: Microservices

```
# Frontend (public)
docker run -p 80:80 frontend

# Backend (internal)
docker run backend

# Database (internal)
docker run database
```

7.8 Docker Networks

```
# List networks
docker network ls

# Create custom network
docker network create mynetwork

# Run container on specific network
docker run --network mynetwork --name web nginx

# Connect running container to network
docker network connect mynetwork mycontainer

# Disconnect from network
docker network disconnect mynetwork mycontainer

# Inspect network
docker network inspect mynetwork

# Remove network
docker network rm mynetwork
```

7.9 Port Mapping Best Practices

Use non-privileged ports (>1024) on host

```
docker run -p 8080:80 nginx  # Good
docker run -p 80:80 nginx    # Requires root/sudo
```

Document port mappings in docker-compose.yml

```
services:
  web:
    image: nginx
    ports:
      - "8080:80"  # host:container
```

Use environment variables for flexibility

```
docker run -p ${WEB_PORT}:80 nginx
```

Limit exposure to localhost when testing

```
docker run -p 127.0.0.1:8080:80 nginx
```

Don't expose unnecessary ports

Don't use same host port for multiple containers

8. Dockerfile & Image Creation

8.1 What is a Dockerfile?

A Dockerfile is a **text file containing a set of instructions** that define how to build a Docker image. It automates the image creation process, ensuring **consistency and reproducibility**.

Steps for using Dockerfile:

1. Create a file named `Dockerfile` (no extension)
2. Add instructions to the Dockerfile
3. Build the Dockerfile to create an image
4. Run the image to create a container

8.2 Dockerfile Instructions

Instruction	Description	Example
`FROM`	Sets the base image	`FROM ubuntu:20.04`
`RUN`	Executes commands during build	`RUN apt-get update`
`CMD`	Default command when container starts	`CMD ["nginx", "-g", "daemon off;"]`
`ENTRYPOINT`	Configures container as executable	`ENTRYPOINT ["python", "app.py"]`
`COPY`	Copy files/directories to image	`COPY app.py /app/`
`ADD`	Copy + extract archives	`ADD archive.tar.gz /app/`
`WORKDIR`	Set working directory	`WORKDIR /app`
`ENV`	Set environment variables	`ENV PORT=8080`
`EXPOSE`	Document which ports are used	`EXPOSE 80 443`
`VOLUME`	Create mount point	`VOLUME ["/data"]`
`USER`	Set user for RUN/CMD/ENTRYPOINT	`USER appuser`
`LABEL`	Add metadata	`LABEL version="1.0"`
`ARG`	Build-time variables	`ARG VERSION=latest`
`HEALTHCHECK`	Check container health	`HEALTHCHECK CMD curl -f`

		http://localhost/
SHELL	Override default shell	SHELL ["/bin/bash", "-c"]
ONBUILD	Trigger for child images	ONBUILD RUN npm install
STOPSIGNAL	System call signal to exit	STOPSIGNAL SIGTERM

8.3 Your First Dockerfile

Example 1: Simple Ubuntu with custom file

```
FROM ubuntu
RUN echo "Learning Docker" > /tmp/testfile
```

Build the image:

```
# Build image from Dockerfile in current directory
docker build -t learningdocker .

# -t = tag (name) for the image
# .  = build context (current directory)

# Verify image was created
docker images

# Create container from image
docker run -it learningdocker /bin/bash

# Inside container, check the file
root@abc:/# cat /tmp/testfile
Learning Docker
```

8.4 Dockerfile Best Practices

Use Specific Base Images

```
#  Bad - unversioned
FROM ubuntu

#  Good - specific version
FROM ubuntu:20.04

#  Better - minimal image
FROM alpine:3.14
```

Minimize Layers

```
#  Bad - multiple layers
RUN apt-get update
RUN apt-get install -y python3
RUN apt-get install -y pip
RUN apt-get clean
```

```
#  Good - single layer
RUN apt-get update && \
    apt-get install -y python3 pip && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*
```

Use .dockerignore

Create `.dockerignore` file:

```
node_modules
.git
.env
*.log
.DS_Store
```

Order Instructions by Frequency of Change

```
# Changes rarely → top
FROM node:14
WORKDIR /app

# Changes occasionally → middle
COPY package*.json ./
RUN npm install

# Changes frequently → bottom
COPY . .

CMD ["npm", "start"]
```

8.5 Complete Dockerfile Examples

Example: Python Web Application

```
# Use official Python image
FROM python:3.9-slim

# Set working directory
WORKDIR /app

# Set environment variables
ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1

# Install dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY . .
```

```
# Create non-root user
RUN useradd -m appuser && \
    chown -R appuser:appuser /app
USER appuser

# Expose port
EXPOSE 8000

# Health check
HEALTHCHECK --interval=30s --timeout=3s \
  CMD python -c "import requests;
requests.get('http://localhost:8000/health')"

# Run application
CMD ["python", "app.py"]
```

Example: Node.js Application

```
FROM node:14-alpine

# Set working directory
WORKDIR /app

# Copy package files
COPY package*.json ./

# Install dependencies
RUN npm ci --only=production

# Copy application files
COPY . .

# Use non-root user
USER node

# Expose port
EXPOSE 3000

# Start application
CMD ["node", "server.js"]
```

Example: Multi-stage Build

```
# Build stage
FROM node:14 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build
```

```
# Production stage
FROM nginx:alpine
COPY --from=builder /app/dist /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

8.6 Building Images

```
# Basic build
docker build -t myapp .

# Build with tag
docker build -t myapp:v1.0 .

# Build with different Dockerfile
docker build -t myapp -f Dockerfile.prod .

# Build with build arguments
docker build --build-arg VERSION=1.0 -t myapp .

# Build without cache
docker build --no-cache -t myapp .

# Build and push
docker build -t username/myapp:latest . && docker push username/myapp:latest
```

8.7 CMD vs ENTRYPOINT

CMD - Can be overridden

```
FROM ubuntu
CMD ["echo", "Hello Docker"]

# Uses CMD
docker run myimage
# Output: Hello Docker

# Override CMD
docker run myimage echo "Goodbye"
# Output: Goodbye
```

ENTRYPOINT - Main executable

```
FROM ubuntu
ENTRYPOINT ["echo"]
CMD ["Hello Docker"]

# Uses both
docker run myimage
# Output: Hello Docker

# CMD is appended to ENTRYPOINT
```

```
docker run myimage "Custom message"
# Output: Custom message
```

8.8 ARG vs ENV

```
# ARG - Build-time only
ARG VERSION=1.0
RUN echo "Building version $VERSION"

# ENV - Runtime available
ENV APP_ENV=production
RUN echo "Environment: $APP_ENV"

# Set ARG at build time
docker build --build-arg VERSION=2.0 -t myapp .

# ENV available in container
docker run myapp env | grep APP_ENV
```

8.9 COPY vs ADD

```
# COPY - Simple file copy (preferred)
COPY app.py /app/

# ADD - Copy + special features
ADD https://example.com/file.tar.gz /tmp/  # Downloads from URL
ADD archive.tar.gz /app/                    # Auto-extracts
```

Best practice: Use COPY unless you need ADD's special features

---

9. Docker Hub & Registry

9.1 What is Docker Hub?

Docker Hub is a cloud-based container registry that allows you to:

- Store and distribute Docker images
- Access millions of public images
- Host private images for teams
- Collaborate with others
- Integrate with CI/CD pipelines

Docker Hub URL: https://hub.docker.com

9.2 Creating a Docker Hub Account

1. Go to https://hub.docker.com
2. Click "Sign Up"
3. Enter your details
4. Verify your email

5. Done! You now have a Docker Hub account

9.3 Pushing Images to Docker Hub

Step 1: Create a container

```
# Run Ubuntu container
docker run -it --name mycontainer ubuntu /bin/bash

# Inside container, create some files
root@abc:/# touch file1 file2
root@abc:/# echo "My custom image" > /tmp/info.txt
root@abc:/# exit
```

Step 2: Create image from container

```
# Commit container to image
docker commit mycontainer myimage

# Verify image was created
docker images
```

Step 3: Login to Docker Hub

```
# Login via CLI
docker login

# Enter your Docker Hub username and password
# Login Succeeded

# Alternative: Provide credentials directly (not recommended)
docker login -u username -p password
```

Step 4: Tag the image

```
# Tag format: username/imagename:tag
docker tag myimage yourusername/myimage:v1.0

# Example
docker tag myimage johndoe/customubuntu:latest

# Verify tag
docker images | grep yourusername
```

Step 5: Push image to Docker Hub

```
# Push image
docker push yourusername/myimage:v1.0

# Example
docker push johndoe/customubuntu:latest
```

```
# Output shows upload progress:
# The push refers to repository [docker.io/johndoe/customubuntu]
# abc123: Pushed
# latest: digest: sha256:abc... size: 1234
```

Step 6: Verify on Docker Hub

1. Go to https://hub.docker.com
2. Login to your account
3. Navigate to "Repositories"
4. You'll see your uploaded image!

9.4 Pulling Images from Docker Hub

```
# Pull your image
docker pull yourusername/myimage:v1.0

# Pull without tag (gets latest)
docker pull yourusername/myimage

# Pull official image
docker pull ubuntu

# Pull from specific user
docker pull nginx
```

9.5 Using Your Pushed Image

```
# Create container from your pushed image
docker run -it --name testcontainer yourusername/myimage:v1.0 /bin/bash

# Inside container, verify your custom files
root@xyz:/# ls
file1  file2

root@xyz:/# cat /tmp/info.txt
My custom image
```

9.6 Managing Docker Hub Repositories

Repository Types

Public Repository: - Free for unlimited public repos - Anyone can pull - Great for open-source projects

Private Repository: - Only you and collaborators can access - Free tier: 1 private repo - Paid plans: Unlimited private repos

Repository Settings

```
# Make repository private (via Docker Hub web interface)
1. Go to repository
```

```
2. Click "Settings"
3. Set visibility to "Private"

# Add collaborators
1. Go to repository
2. Click "Collaborators"
3. Enter username
4. Set permissions (Read/Write/Admin)
```

9.7 Docker Hub Tags and Versions

```
# Push multiple versions
docker tag myimage yourusername/myimage:v1.0
docker push yourusername/myimage:v1.0

docker tag myimage yourusername/myimage:v2.0
docker push yourusername/myimage:v2.0

docker tag myimage yourusername/myimage:latest
docker push yourusername/myimage:latest

# Pull specific version
docker pull yourusername/myimage:v1.0
docker pull yourusername/myimage:v2.0
docker pull yourusername/myimage:latest
```

9.8 Automated Builds

Setup automated builds from GitHub:

1. Connect GitHub account to Docker Hub
2. Create repository on Docker Hub
3. Link to GitHub repository
4. Configure build rules
5. Every push to GitHub triggers Docker build!

Example .github/workflows/docker.yml:

```yaml
name: Docker Build and Push

on:
  push:
    branches: [ main ]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2

      - name: Login to Docker Hub
```

```yaml
        uses: docker/login-action@v1
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      - name: Build and push
        uses: docker/build-push-action@v2
        with:
          context: .
          push: true
          tags: username/myapp:latest
```

9.9 Private Docker Registry

Run your own private registry:

```
# Run registry container
docker run -d -p 5000:5000 --name registry registry:2

# Tag image for private registry
docker tag myimage localhost:5000/myimage

# Push to private registry
docker push localhost:5000/myimage

# Pull from private registry
docker pull localhost:5000/myimage
```

Production-ready registry:

```yaml
version: '3'
services:
  registry:
    image: registry:2
    ports:
      - "5000:5000"
    environment:
      REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY: /data
    volumes:
      - ./registry-data:/data
```

9.10 Docker Hub Alternatives

- **GitHub Container Registry** (ghcr.io)
- **Amazon ECR** (Elastic Container Registry)
- **Google Container Registry** (gcr.io)
- **Azure Container Registry**
- **GitLab Container Registry**
- **Harbor** (Self-hosted)
- **Quay.io**

10. Docker Commands Reference

10.1 Container Commands

```
# Lifecycle
docker run <image>               # Create and start
docker start <container>          # Start stopped container
docker stop <container>           # Stop running container
docker restart <container>        # Restart container
docker pause <container>          # Pause container
docker unpause <container>        # Unpause container
docker kill <container>           # Force stop
docker rm <container>             # Remove container
docker rm -f <container>          # Force remove

# Execution
docker run -it <image> bash       # Interactive terminal
docker run -d <image>             # Detached mode
docker exec -it <container> bash # Execute in running container
docker attach <container>         # Attach to running container

# Information
docker ps                         # List running containers
docker ps -a                      # List all containers
docker inspect <container>        # Detailed info
docker logs <container>           # View logs
docker logs -f <container>        # Follow logs
docker top <container>            # Running processes
docker stats <container>          # Resource usage
docker port <container>           # Port mappings
docker diff <container>           # Filesystem changes

# Bulk Operations
docker stop $(docker ps -q)       # Stop all running
docker rm $(docker ps -a -q)      # Remove all containers
docker container prune            # Remove stopped containers
```

10.2 Image Commands

```
# Management
docker images                     # List images
docker pull <image>               # Download image
docker push <image>               # Upload image
docker rmi <image>                # Remove image
docker rmi -f <image>             # Force remove
docker image prune                # Remove unused images

# Building
docker build -t <name> .          # Build from Dockerfile
docker build --no-cache .         # Build without cache
```

```
docker commit <container> <img>  # Create image from container

# Information
docker inspect <image>           # Detailed info
docker history <image>           # Image layers

# Tagging
docker tag <image> <new-tag>     # Tag image

# Search
docker search <term>              # Search Docker Hub

# Bulk Operations
docker rmi $(docker images -q)   # Remove all images
```

10.3 Volume Commands

```
# Management
docker volume create <name>      # Create volume
docker volume ls                 # List volumes
docker volume inspect <name>     # Volume details
docker volume rm <name>          # Remove volume
docker volume prune              # Remove unused volumes

# Usage in containers
docker run -v <vol>:<path> <img> # Mount volume
docker run -v <host>:<cont> <img> # Bind mount
docker run --volumes-from <cont> # Share volumes
```

10.4 Network Commands

```
# Management
docker network create <name>     # Create network
docker network ls                # List networks
docker network inspect <name>    # Network details
docker network rm <name>         # Remove network
docker network prune             # Remove unused networks

# Container networking
docker network connect <net> <cont>    # Connect container
docker network disconnect <net> <cont> # Disconnect container

# Run with network
docker run --network <name> <image>    # Use custom network
```

10.5 Docker Compose Commands

```
# Lifecycle
docker-compose up                # Start services
docker-compose up -d             # Start in background
docker-compose down              # Stop and remove
docker-compose start             # Start services
docker-compose stop              # Stop services
```

```
docker-compose restart          # Restart services

# Information
docker-compose ps               # List containers
docker-compose logs             # View logs
docker-compose logs -f          # Follow logs
docker-compose top              # Running processes

# Execution
docker-compose exec <service> sh # Execute command

# Building
docker-compose build            # Build images
docker-compose build --no-cache  # Build without cache
docker-compose pull             # Pull images
```

10.6 System Commands

```
# Information
docker version                  # Docker version
docker info                     # System information
docker system df                # Disk usage

# Cleanup
docker system prune             # Remove unused data
docker system prune -a          # Remove all unused data
docker system prune --volumes    # Include volumes

# Login/Logout
docker login                    # Login to registry
docker logout                   # Logout from registry
```

10.7 Registry Commands

```
# Docker Hub
docker login                    # Login
docker push <image>             # Push image
docker pull <image>             # Pull image
docker search <term>            # Search images

# Tag for registry
docker tag <img> user/name:tag   # Tag for Docker Hub
docker tag <img> reg.com/name:tag # Tag for private registry
```

10.8 Useful Command Combinations

```
# Stop and remove all containers
docker stop $(docker ps -a -q) && docker rm $(docker ps -a -q)

# Remove all images
docker rmi $(docker images -q)

# Remove everything
```

```
docker system prune -a --volumes

# View container IP address
docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
<container>

# Follow logs with timestamps
docker logs -f -t <container>

# Copy files
docker cp <container>:/path/to/file /local/path
docker cp /local/path <container>:/path/to/file

# Execute command as specific user
docker exec -u root <container> command

# Export/Import containers
docker export <container> > container.tar
docker import container.tar

# Save/Load images
docker save <image> > image.tar
docker load < image.tar

# Get container shell (try different shells)
docker exec -it <container> /bin/bash || docker exec -it <container> /bin/sh
```

10.9 Docker Run Flags Reference

```
# Basic flags
-d                    # Detached mode
-it                   # Interactive with TTY
--name <name>         # Container name
--rm                  # Auto-remove when stopped

# Resource limits
-m, --memory          # Memory limit (e.g., 512m, 2g)
--cpus                # CPU limit (e.g., 0.5, 2.0)
--cpu-shares          # CPU priority

# Networking
-p <host>:<cont>      # Port mapping
--network <name>      # Custom network
--link <container>    # Link containers (deprecated)

# Volumes
-v <vol>:<path>       # Volume mount
-v <host>:<cont>      # Bind mount
--volumes-from <cont> # Share volumes
```

```
# Environment
-e KEY=value          # Environment variable
--env-file <file>     # Environment file

# User
-u, --user            # Run as user

# Working directory
-w, --workdir         # Working directory

# Restart policy
--restart             # Restart policy (no, always, on-failure, unless-
stopped)
```

11. Advanced Topics & Best Practices

11.1 Multi-Stage Builds

Problem: Build artifacts increase image size

Solution: Use multi-stage builds

```
# Build stage
FROM node:14 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# Production stage
FROM node:14-alpine
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/node_modules ./node_modules
EXPOSE 3000
CMD ["node", "dist/server.js"]
```

Benefits: - Smaller final image (only production files) - No build tools in production - Better security - Faster deployments

11.2 Health Checks

```
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
  CMD curl -f http://localhost/ || exit 1

# Check health status
docker ps
```

```
# Inspect health
docker inspect --format='{{.State.Health.Status}}' <container>
```

11.3 Docker Compose for Multi-Container Apps

docker-compose.yml:

```
version: '3.8'

services:
  web:
    build: .
    ports:
      - "80:80"
    depends_on:
      - db
      - redis
    environment:
      - DATABASE_URL=postgres://db:5432/myapp
    volumes:
      - ./app:/app
    networks:
      - backend

  db:
    image: postgres:13
    environment:
      POSTGRES_PASSWORD: secret
    volumes:
      - db-data:/var/lib/postgresql/data
    networks:
      - backend

  redis:
    image: redis:alpine
    networks:
      - backend

volumes:
  db-data:

networks:
  backend:
```

Usage:

```
# Start all services
docker-compose up -d

# View logs
```

```
docker-compose logs -f web

# Scale service
docker-compose up -d --scale web=3

# Stop all
docker-compose down
```

11.4 Security Best Practices

1. Don't Run as Root

```
# Create user
RUN useradd -m -u 1000 appuser

# Switch to user
USER appuser

# Run application
CMD ["python", "app.py"]
```

2. Scan Images for Vulnerabilities

```
# Use Docker Scout
docker scout cves <image>

# Use Trivy
trivy image <image>

# Use Snyk
snyk container test <image>
```

3. Use Official Base Images

```
#  Good
FROM node:14-alpine

#  Bad
FROM random-user/node:custom
```

4. Don't Include Secrets

```
#  NEVER do this
ENV API_KEY=abc123secret

#  Use Docker secrets or environment variables at runtime
docker run -e API_KEY=abc123secret myapp
```

5. Minimize Attack Surface

```
# Use minimal base images
FROM alpine:3.14

# Remove unnecessary packages
RUN apk add --no-cache python3 && \
```

```
    rm -rf /var/cache/apk/*

# Use read-only filesystem where possible
docker run --read-only myapp
```

11.5 Image Optimization

Reduce Layer Count

```
#  Bad - Many layers
RUN apt-get update
RUN apt-get install -y package1
RUN apt-get install -y package2

#  Good - Single layer
RUN apt-get update && \
    apt-get install -y package1 package2 && \
    rm -rf /var/lib/apt/lists/*
```

Use .dockerignore

```
# .dockerignore
node_modules
npm-debug.log
.git
.env
*.md
.DS_Store
```

Leverage Build Cache

```
# Copy dependency files first (changes less)
COPY package*.json ./
RUN npm install

# Copy source code (changes more)
COPY . .
```

11.6 Logging Best Practices

```
# JSON logging driver
docker run --log-driver=json-file myapp

# Limit log size
docker run --log-opt max-size=10m --log-opt max-file=3 myapp

# Send logs to syslog
docker run --log-driver=syslog myapp

# Send to external service
docker run --log-driver=fluentd --log-opt fluentd-address=localhost:24224
myapp
```

11.7 Resource Management

```
# Memory limits
docker run -m 512m myapp

# CPU limits
docker run --cpus="1.5" myapp

# Combined
docker run -m 1g --cpus="2.0" myapp

# View resource usage
docker stats
```

11.8 Container Orchestration

When you need to manage many containers:

Docker Swarm:

```
# Initialize swarm
docker swarm init

# Deploy stack
docker stack deploy -c docker-compose.yml myapp

# Scale service
docker service scale myapp_web=5
```

Kubernetes:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp
        image: myapp:latest
        ports:
        - containerPort: 80
```

11.9 CI/CD Integration

GitHub Actions Example:

```yaml
name: Docker CI/CD

on:
  push:
    branches: [ main ]

jobs:
  build-and-push:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2

      - name: Build image
        run: docker build -t myapp:${{ github.sha }} .

      - name: Run tests
        run: docker run myapp:${{ github.sha }} npm test

      - name: Push to registry
        run: |
          docker login -u ${{ secrets.DOCKER_USERNAME }} -p ${{
secrets.DOCKER_PASSWORD }}
          docker push myapp:${{ github.sha }}
```

11.10 Troubleshooting Common Issues

Container Won't Start

```
# Check logs
docker logs <container>

# Inspect container
docker inspect <container>

# Try running interactively
docker run -it <image> /bin/bash
```

Out of Disk Space

```
# Clean up
docker system prune -a --volumes

# Check disk usage
docker system df

# Remove specific items
docker volume prune
docker image prune
```

Network Issues

```
# Check container network
docker inspect <container> | grep IPAddress

# Test connectivity
docker exec <container> ping google.com

# Check DNS
docker exec <container> nslookup google.com
```

Permission Issues

```
# Run as specific user
docker run -u 1000:1000 myapp

# Fix volume permissions
docker run -v /data:/data --rm alpine chown -R 1000:1000 /data
```

11.11 Production Checklist

- ☐ Use specific image tags, not `latest`
- ☐ Run containers as non-root user
- ☐ Set resource limits (CPU, memory)
- ☐ Implement health checks
- ☐ Use read-only filesystem where possible
- ☐ Scan images for vulnerabilities
- ☐ Use secrets management (not environment variables)
- ☐ Configure logging
- ☐ Set restart policy (`--restart=unless-stopped`)
- ☐ Use Docker Compose or orchestration for multi-container apps
- ☐ Regular security updates
- ☐ Monitor resource usage
- ☐ Backup volumes
- ☐ Document all configurations

Useful Resources:

- Official Docker Documentation
- Docker Labs
- Docker Community Forums
- Awesome Docker