**DOCKER DOCUMENT**

Containerization and virtualization are both techniques used to create isolated environments for applications, but they differ significantly in how they achieve this and their underlying architecture.

**Containerization**

* **Definition:** Containerization involves packaging an application along with its dependencies into a lightweight, portable container that can run consistently across different environments.
* **How it works:** Containers share the host operating system’s kernel. Each container runs as an isolated process, using container engine (like Docker) to manage isolation.
* **Key characteristics:**
  + **Lightweight:** Containers are smaller and start up faster because they share the OS kernel.
  + **Efficiency:** More containers can run on the same hardware since they reuse OS resources.
  + **Consistency:** Containers encapsulate everything needed to run an application, ensuring environment uniformity.
* **Use case:** Ideal for application deployment, microservices, CI/CD pipelines.
* **Example:** Docker containers.

**Virtualization**

* **Definition:** Virtualization involves creating fully isolated virtual machines (VMs), each with its own operating system, running on a physical host using a hypervisor.
* **How it works:** A hypervisor (like VMware, Hyper-V, or KVM) abstracts the hardware and runs multiple VMs, each with its own guest OS.
* **Key characteristics:**
  + **Heavyweight:** VMs contain complete OS images, so they consume more resources.
  + **Isolation:** VMs are fully isolated, providing strong security boundaries.
  + **Flexibility:** VMs can run different OSes on the same hardware (e.g., Windows VM on Linux host).
* **Use case:** Server consolidation, running multiple different OSes, situations requiring strong isolation.
* **Example:** VMware ESXi, VirtualBox, Hyper-V.

**Docker Architecture Overview**

Docker is designed around a client-server architecture consisting of several key components working together to build, run, and manage containers.

**1. Docker Engine (Docker Daemon / Server)**

* **Role:** The core service that manages Docker objects such as images, containers, networks, and volumes.
* **Implementation:** Runs as a background process (dockerd) on the host machine.
* **Responsibilities:**
  + Building images
  + Running containers
  + Managing images and containers
  + Communicating via REST API

**2. Docker Client (Command Line Interface, CLI)**

* **Role:** The primary interface for users to communicate with Docker.
* **How:** Users run commands like docker run, docker build, etc., which are sent to the Docker Daemon.
* **Interaction:** Sends REST API requests to the Docker Daemon.

**3. Docker API**

* **Role:** A RESTful API used for communication between Docker CLI and Docker Daemon.
* **Uses:** Tools or scripts can interface directly with Docker via this API.

**4. Docker Images**

* **Definition:** Read-only templates used to create containers.
* **Creation:** Built using Dockerfiles, or pulled from Docker registries (like Docker Hub).
* **Purpose:** Serve as the portable, shareable snapshots of containers.

**5. Containers**

* **Definition:** Running instances of Docker images.
* **Features:**
  + Isolated environments with their own filesystem, network, and process space.
  + Lightweight, sharing the host OS kernel.
  + Managed by Docker Engine.

**6. Union File System (UnionFS)**

* **Role:** Combines multiple layers (images) into a single filesystem.
* **Benefit:** Enables image layering and efficient storage.

**7. Storage & Networks**

* **Volumes:** Persistent data outside containers.
* **Networks:** Connect containers and external systems securely.

**Diagrammatic Summary**

Here's a simplified view:

User @ CLI (docker commands)

|

v

Docker CLI --------------> Docker Daemon (dockerd)

|

------------------------------

| |

Containers Images

|

Volumes, Networks

**Key Points:**

* **Client-Server:** User interacts via CLI, which talks to Docker Daemon.
* **Layers:** Docker images are composed of layered filesystems.
* **Containers:** Instances of images, isolated and portable.
* **Shared OS:** Containers share the host OS kernel for efficiency.

**For Linux and macOS:**

1. **Open your terminal** and run:

docker --version

* If Docker is installed, you'll see the version info, e.g.:

Docker version 20.10.8, build 3967b7d

1. **Alternatively**, you can check if the Docker daemon is running:

sudo systemctl status docker

* If Docker is active, you'll see a status indicating it's running. If not, you'll get an inactive or not found message.

**For Windows:**

* **Using Command Prompt or PowerShell:**

docker --version

* If Docker is installed, you'll see the version info.
* You can also check if Docker Desktop is running via the system tray or Control Panel.

**Additional check:**

* To verify Docker is properly installed and can run containers:

docker run hello-world

* If Docker is working correctly, you will see a message about Docker downloading and running the hello-world container.

By default, on most Linux systems, **only the root user** can run Docker commands.

**Why?**

* Docker communicates with the Docker daemon (dockerd), which usually requires root privileges.
* The Docker socket (/var/run/docker.sock) is typically owned by root and has permissions that restrict access to root or members of the docker group.

**Common practice:**

* To allow a regular user to run Docker commands without root privileges, that user must be **added to the docker group**.

**How to check if your user can run Docker commands:**

docker info

* If you see output describing your Docker setup, your user has permission.
* If you get a permission error (e.g., “permission denied”), it means your user lacks access.

**How to grant a user Docker access:**

1. **Create or add your user to the docker group:**

sudo usermod -aG docker your\_username

1. **Log out and log back in**, or restart your system to apply group changes.
2. **Verify access:**

docker info

to allow the **Ubuntu** user to run Docker commands without needing root privileges, you need to add that user to the docker group. Here's the step-by-step process:

**Step 1: Add the Ubuntu user to the docker group**

Open a terminal and run:

sudo usermod -aG docker Ubuntu

* -aG means "append the user to the group" (here, the docker group).

**Step 2: Log out and log back in**

For the group change to take effect, you need to **log out** of the current session and **log back in**.

* Alternatively, you can restart your system.

**Step 3: Verify the user can run Docker commands**

After logging back in, test:

docker info

or

docker run hello-world

* If the command runs successfully without permission errors, the user Ubuntu now has access to run Docker commands.

**Important notes:**

* **Security:** Adding a user to the docker group grants that user root-level access via Docker, so do this only for trusted users.
* **User switching:** If you don't want to log out/in, you can **switch users** in the terminal using:

su - Ubuntu

or open a new terminal session as Ubuntu.

**Summary:**

sudo usermod -aG docker Ubuntu

# Log out and log back in

docker run hello-world

To list all Docker images available on your Docker host, you can use the following command:

docker images

**What does docker images display?**

* **REPOSITORY:** The name of the image repository (e.g., nginx, ubuntu).
* **TAG:** The specific version/tag of the image (e.g., latest, 20.04).
* **IMAGE ID:** Unique identifier for the image.
* **CREATED:** How long ago the image was created.
* **SIZE:** The size of the image.

**Example output:**

REPOSITORY TAG IMAGE ID CREATED SIZE

ubuntu latest 2ca708c1c9cc 2 weeks ago 64.2MB

nginx stable 4bb46517cac3 3 weeks ago 133MB

myapp v1 a9342f1d4e53 5 days ago 200MB

**Additional options:**

* To show all images, including intermediate or dangling images:

docker images -a

* To filter images (e.g., by repository name):

docker images ubuntu

**Summary:**

* Use docker images to list all images on your Docker host.

o list all **currently running containers** on your Docker host, use the following command:

docker ps

**What does docker ps show?**

* **CONTAINER ID:** Unique identifier for the container.
* **IMAGE:** The image used to create the container.
* **COMMAND:** The command the container is executing.
* **CREATED:** How long ago the container was created.
* **STATUS:** The current status (e.g., Up 5 minutes).
* **PORTS:** Port mappings.
* **NAMES:** The assigned name of the container.

**Example output:**

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

a1b2c3d4e5f6 nginx:latest nginx -g ... 10 minutes ago Up 10 minutes 0.0.0.0:80->80/tcp webpage

d7e8f9a0b1c2 mongo:latest mongod 2 hours ago Up 2 hours 27017/tcp mongo-db

**To list all containers, including stopped ones, add the -a (or --all) flag:**

docker ps -a

Docker containers can have various statuses that indicate their current state. Here are the most common container statuses you might see:

**Common Docker Container Statuses:**

1. **Created**
   * The container has been created but **has not started yet**.
2. **Running**
   * The container is **currently active** and executing.
3. **Paused**
   * The container has been **paused** (its process is suspended).
4. **Restarting**
   * The container is in the process of **restarting** automatically, often due to restart policies or errors.
5. **Exited**
   * The container has **stopped** after running; you can see the exit code (e.g., Exited (0) means it finished successfully).
6. **Dead**
   * The container is **dead**, which usually indicates a crash or an error, and it can’t be started again without cleanup.

**How to see container statuses:**

Use the command:

docker ps -a

This will list all containers with their current status in the **STATUS** column.

**Example:**

CONTAINER ID IMAGE COMMAND CREATED STATUS NAMES

abc12345 nginx:latest nginx -g ... 2 minutes ago Up 2 minutes webserver

def67890 mongo:latest mongod 10 minutes ago Exited (0) 8 minutes ago mongo-db

ghi13579 redis redis-server 3 hours ago Dead cache

**Summary:**

| **Status** | **Description** |
| --- | --- |
| **Created** | Container created but not started yet |
| **Running** | Container actively running |
| **Paused** | Container is paused (suspended processes) |
| **Restarting** | Container is restarting automatically |
| **Exited** | Container has stopped after completion or error |
| **Dead** | Container is dead due to a crash or error |

reate and run a new container using below details Image:jenkins/jenkins:jdk17 Containername: Jenkins Port:8080 Mode:Detached mode

To create and run a new Docker container with the specified details, you can use the following docker run command:

docker run -d --name Jenkins -p 8080:8080 jenkins/jenkins:jdk17

**Explanation:**

* -d: Run the container in **detached mode** (in the background).
* --name Jenkins: Name the container **Jenkins**.
* -p 8080:8080: Map port **8080** of the container to port **8080** on the host.
* jenkins/jenkins:jdk17: The image to use.

**Complete command:**

docker run -d --name Jenkins -p 8080:8080 jenkins/jenkins:jdk17

**After running:**

* You can verify the container is running:

docker ps

To install git inside an existing container named Jenkins, follow these steps:

**Method 1: Using docker exec (interactive install)**

1. **Run an interactive shell inside the container:**

docker exec -it Jenkins bash

*(If the container uses sh instead of bash, use sh)*

docker exec -it Jenkins sh

1. **Once inside the container, install Git:**

* **For Debian/Ubuntu-based images:**

apt-get update

apt-get install -y git

* **For Alpine-based images:**

apk add --no-cache git

* **For CentOS/RHEL-based images:**

yum install -y git

1. **Exit the container:**

exit

**Method 2: Commit changes (if you want to save the updated image)**

If you'd like to create a new image with git installed:

docker commit Jenkins myjenkins\_with\_git

Then, you can run containers from this new image.

**Note:**

* Installing packages inside a running container is **temporary** unless you commit the container.
* For persistent setups, it's better to update the Docker image itself via a Dockerfile and rebuild.

**Summary:**

docker exec -it Jenkins bash

# inside container

apt-get update && apt-get install -y git

# or use sh depending on the base image

To list images, running containers, stopped containers, and paused containers all at once using **single commands**, you can break down and run multiple commands sequentially, or combine some commands with && or use a script. However, in a single line, you can do:

**1. List Docker images:**

docker images

**2. List running containers:**

docker ps

**3. List stopped containers:**

docker ps -a --filter "status=exited"

**4. List paused containers:**

docker ps -a --filter "status=paused"

**Combine all into a single command with separators:**

echo "Images:" && docker images && \

echo "\nRunning Containers:" && docker ps && \

echo "\nStopped Containers:" && docker ps -a --filter "status=exited" && \

echo "\nPaused Containers:" && docker ps -a --filter "status=paused"

**Explanation:**

* docker images shows all images.
* docker ps shows running containers.
* docker ps -a --filter "status=exited" shows stopped containers.
* docker ps -a --filter "status=paused" shows paused containers.
* The echo commands label each section for clarity.

When a Docker container is not running, troubleshooting involves checking logs, inspecting the container's status, and gathering information to identify the root cause. Here's a step-by-step guide:

**1. Check the container's status**

Use:

docker ps -a

* Look for your container (by name or ID).
* Check the **STATUS** column. If it shows Exited, Dead, or similar, note the exit code.

**2. View container logs**

Logs often provide clues:

docker logs <container\_name\_or\_id>

* For example:

docker logs Jenkins

* Check for errors or failure messages that indicate what's wrong.

**3. Examine detailed container info**

Use:

docker inspect <container\_name\_or\_id>

* This outputs detailed JSON info about the container.
* Look for **State** info:
  + **ExitCode**: Any non-zero usually indicates an error.
  + **Error**: Specific error messages.
  + **RestartCount**, etc.

**4. Try to restart the container**

Attempt to restart and watch for errors:

docker restart <container\_name\_or\_id>

**5. Check resource constraints**

* Ensure there's enough CPU, memory, disk space.
* Check system logs for resource issues.

**6. Check for port conflicts**

* The container may fail to start if ports are already in use.
* Use:

netstat -tuln | grep 8080

to see if port 8080 is occupied.

**7. If container crashed due to fatal error**

* Review logs thoroughly.
* Consider starting an interactive shell inside the container (if it exists):

docker run -it --entrypoint /bin/bash <image>

or

docker exec -it <container\_id> bash

* Run diagnostic commands inside.

**8. Rebuild or recreate the container**

* If configuration or image issues are suspected, rebuild or recreate the container from the image.

**Summary of commands:**

docker ps -a # Check status

docker logs <container> # View logs

docker inspect <container> # Inspect detailed info

docker restart <container> # Restart container

In Docker, the term **Persistent Volume** is typically associated with Docker **volumes**, especially in the context of Docker containers that require data to persist beyond the container's lifecycle.

**What is a Docker Volume?**

* A **Docker volume** is a special storage mechanism used to persist data generated or used by Docker containers.
* Volumes are stored outside the container's writable layer, usually on the host filesystem, allowing data to persist even if the container is removed.
* Volumes can also be shared between containers.

**Why use volumes (Persistent Storage)?**

* **Data persistence:** Ensures data remains available even if containers are recreated or deleted.
* **Separation of concerns:** Keeps data separate from container lifecycle.
* **Sharing data:** Multiple containers can access the same volume.
* **Backup and restore:** Easier backup and recovery of persistent data.

**How to create and use a volume in Docker**

**1. Create a volume:**

docker volume create mydata

**2. Use a volume when running a container:**

docker run -d --name mycontainer -v mydata:/app/data myimage

This mounts the volume mydata inside the container at /app/data.

**3. Alternatively, create a bind mount for a directory from host:**

docker run -d --name mycontainer -v /host/path:/container/path myimage

**Summary:**

* A **Persistent Volume in Docker** refers to storage (via Docker volumes or bind mounts) that persists data independently of containers.
* Volumes are the recommended way to persist data in Docker because they are managed by Docker and offer advantages like better performance, sharing, and easy backup.

**1. Docker Managed Volumes (Named Volumes)**

**Description:**

* Managed by Docker.
* Created explicitly with docker volume create or automatically when you use the -v flag with a volume name.
* Stored in Docker's default directory (/var/lib/docker/volumes/ on Linux).
* Easy to use, share, and manage.

**Characteristics:**

* Do not depend on host filesystem paths.
* Persistent beyond container lifecycle.
* Can be shared between containers.

**Example:**

docker volume create mydata

docker run -d --name container1 -v mydata:/app/data myimage

**2. Bind Mounts (Host Directory Mounts)**

**Description:**

* Directly links a directory or file from the host machine into the container.
* Created by specifying the full path of the host directory.

**Characteristics:**

* Provides direct access to host files.
* Changes on host or container are immediately reflected.
* Useful for development and testing but less portable.

**Example:**

docker run -d --name container2 -v /host/path:/app/data myimage

**Summary of Differences:**

| **Feature** | **Named Volumes** | **Bind Mounts** |
| --- | --- | --- |
| Managed by Docker | Yes | No |
| Storage location | Docker's managed directory | Host filesystem path |
| Persistence | Yes | Yes |
| Portability | Yes (easier to move with Docker) | Less portable (depends on host path) |
| Use case | Application data, databases | Development, configuration files, logs |

**What is Docker Compose?**

**Docker Compose** is a tool designed to define and run multi-container Docker applications. It allows you to:

* Define multiple containers, networks, and volumes in a single file.
* Start, stop, and manage all services together with a single command.
* Simplify complex multi-container setups, especially in development, testing, or staging environments.

**Why use Docker Compose?**

* **Simplify orchestration:** Manage multiple containers as a single application.
* **Configuration as code:** All settings are stored in a YAML file (docker-compose.yml).
* **Ease of setup:** Spin up entire environments with docker-compose up.
* **Repeatability:** Ensure consistent setups across environments.

**What is docker-compose.yml?**

The docker-compose.yml file is a YAML configuration file that defines:

* **Services:** The containers to run, their images, ports, environment variables, etc.
* **Networks:** Custom networks to connect services.
* **Volumes:** Persistent storage preferences.

**Basic structure of docker-compose.yml**

version: '3' # Compose file format version

services:

service\_name:

image: image\_name:tag

ports:

- "host\_port:container\_port"

environment:

- ENV\_VAR=value

volumes:

- host\_path:container\_path

command: your\_command

networks:

- network\_name

networks:

network\_name:

driver: bridge

volumes:

volume\_name:

**Example: Simple docker-compose.yml with a web app and database**

version: '3'

services:

web:

image: nginx:latest

ports:

- "8080:80"

networks:

- my\_network

db:

image: mysql:5.7

environment:

MYSQL\_ROOT\_PASSWORD: example

volumes:

- db\_data:/var/lib/mysql

networks:

- my\_network

networks:

my\_network:

driver: bridge

volumes:

db\_data:

**How to use docker-compose.yml**

* **Start all services:**

docker-compose up -d

* **Stop all services:**

docker-compose down

* **View logs:**

docker-compose logs

**1. docker run**

**Purpose:** Create and start a new container.

**Example:**

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

* Runs an Nginx container in detached mode (-d).

**2. docker exec**

**Purpose:** Run a command inside a running container.

**Example:**

docker exec -it mynginx bash

* Opens an interactive Bash shell inside the container named mynginx.

**3. docker ps**

**Purpose:** List running containers.

**Example:**

docker ps

* Shows active containers.

**To list all containers (including stopped):**

docker ps -a

**4. docker build**

**Purpose:** Build an image from a Dockerfile.

**Example:**

docker build -t myapp:latest .

* Builds an image named myapp tagged latest from current directory.

**5. docker pull**

**Purpose:** Download an image from Docker Hub.

**Example:**

docker pull ubuntu:20.04

**6. docker push**

**Purpose:** Upload an image to Docker Hub or a registry.

**Example:**

docker push myusername/myapp:latest

*(Make sure you're logged in and have the image tagged with your registry name.)*

**7. docker images**

**Purpose:** List downloaded images.

docker images

**8. docker login**

**Purpose:** Log in to Docker registry.

docker login

* Prompts for username and password.

**9. docker logout**

**Purpose:** Log out from Docker registry.

docker logout

**10. docker search**

**Purpose:** Search Docker Hub for images.

**Example:**

docker search nginx

**11. docker version**

**Purpose:** Show Docker version info.

docker version

**12. docker info**

**Purpose:** Show detailed Docker system info.

docker info

**Summary:**

| **Command** | **Purpose** | **Example** |
| --- | --- | --- |
| run | Run new container | docker run -d nginx |
| exec | Run commands inside running containers | docker exec -it mycontainer bash |
| ps | List active containers | docker ps |
| build | Build images from Dockerfile | docker build -t myapp . |
| pull | Download images from registry | docker pull ubuntu |
| push | Upload images to registry | docker push myimage |
| images | List downloaded images | docker images |
| login | Login to Docker registry | docker login |
| logout | Logout from registry | docker logout |
| search | Search Docker Hub | docker search nginx |
| version | Show Docker version info | docker version |
| info | Show system-wide info | docker info |

**Connect a container to the host network**

**Command:**

docker network connect host <container\_name\_or\_id>

**Example:**

docker network connect host mycontainer

This adds the container mycontainer to the host network, allowing it to share the host's network interfaces directly.

**Disconnect a container from the host network**

**Command:**

docker network disconnect host <container\_name\_or\_id>

**Example:**

docker network disconnect host mycontainer

This removes the container mycontainer from the host network.

**Note:**

* Connecting a running container to the host network might require restart or may have limitations depending on your Docker setup.
* Not all network drivers support disconnecting when the container is running; some may need container restart or stop.

**Summary:**

| **Action** | **Command** | **Example** |
| --- | --- | --- |
| Connect to host network | docker network connect host <container> | docker network connect host mycontainer |
| Disconnect from host network | docker network disconnect host <container> | docker network disconnect host mycontainer |

**What is the Default Network in Docker?**

* **Default Network:** When you install Docker, it automatically creates a network called **bridge**.
* **Purpose:** The bridge network is the default network driver used for containers that are run without specifying a custom network.
* **Behavior:**
  + Containers connected to the bridge network can communicate with each other by default.
  + Containers get a private IP address within this network.
  + Port mapping (like -p options) allows access to the host machine.

**Additional Networks:**

* **host:** Shares the host's network stack directly (no network isolation).
* **none:** No network; container has no network interfaces.

**What is the docker inspect command used for?**

**Purpose:**

* The docker inspect command provides detailed information about Docker objects (**containers**, **images**, **volumes**, **networks**, etc.).
* It returns a **JSON format** output containing configuration, status, resource allocations, network settings, mount points, environment variables, and more.

**Usage:**

docker inspect <object\_name\_or\_id>

**Example:**

Inspect a container:

docker inspect mycontainer

Inspect a network:

docker inspect bridge

**What is the none network in Docker?**

* The none network is a **Docker network driver** that provides **no network connectivity** to the container.
* When a container is connected to the none network:
  + It **has no network interfaces**.
  + It cannot communicate with other containers or the outside world.
  + It is typically used when you want an **isolated container** with no network access.

**Use case:**

* Containers that should not have network access.
* Testing or security-sensitive scenarios where network traffic needs to be restricted.

**How to disconnect a container from the bridge network and connect it to the none network?**

**1. Disconnect from the bridge network:**

docker network disconnect bridge <container\_name\_or\_id>

**Example:**

docker network disconnect bridge mycontainer

**2. Connect to the none network:**

docker network connect none <container\_name\_or\_id>

**Example:**

docker network connect none mycontainer

**Important notes:**

* The container must be **running** to connect or disconnect from networks.
* Sometimes, **disconnecting from bridge and connecting to none** may require container restart depending on configurations.

**Summary:**

* **none network:** Provides **no network connectivity** to the container.
* **Disconnect from bridge:**

docker network disconnect bridge mycontainer

* **Connect to none:**

docker network connect none mycontainer

**1. Default Location of Docker Storage**

**Linux:**

* Docker stores **images, containers, volumes, and networks** by default in:

/var/lib/docker/

This directory contains subdirectories like:

* images/
* containers/
* volumes/
* overlay2/ (or other storage drivers)

**Note:**

* You can change this storage location by modifying Docker's configuration (daemon.json).

**2. Default Location of Docker Images**

* Located inside the Docker directory:

/var/lib/docker/images/

* Actual image data is stored in the **overlay2/** (or other storage driver-specific directories such as aufs, btrfs, etc.) subdirectory.

**3. Default Location of Containers**

* Container data is stored in:

/var/lib/docker/containers/

* Each container has a subdirectory named after its container ID, containing logs and configuration files.

**4. Note for Windows & Mac**

* Docker Desktop uses a VM to run Docker containers.
* The actual data is stored inside that VM, typically in:

~/Library/Containers/com.docker.docker/Data/vms/0/

* Accessing storage directly on Windows/Mac is more complex; it's managed by Docker Desktop.

**Summary:**

| **Item** | **Default Location (Linux)** |
| --- | --- |
| Docker root directory | /var/lib/docker/ |
| Images | /var/lib/docker/image/ or within /var/lib/docker/overlay2/ |
| Containers | /var/lib/docker/containers/ |

A **Docker image** is a **read-only, portable template** that contains all the necessary instructions to run a container. Think of it as a **snapshot** or **blueprint** of an application, including the application code, runtime, libraries, dependencies, and configurations.

**Key Points about Docker Images:**

* **Immutable:** Once built, images do not change. Containers are instances of images.
* **Layered Structure:** Images are composed of multiple read-only layers stacked together, which makes them efficient for storage and sharing.
* **Build from Dockerfile:** You create images by writing a Dockerfile, which specifies instructions for building the image.
* **Reusable:** Images can be shared across systems, stored in registries like Docker Hub, and used to deploy containers consistently.

**How does an image work?**

* When you run a container, Docker creates a writable layer on top of the image.
* Multiple containers can run from the same image, each with its own writable layer.

**Example:**

docker pull ubuntu:20.04

docker images

* The first command downloads the Ubuntu image.
* The second lists all images stored locally.

**1. FROM**

**Purpose:**

* Specifies the **base image** to build your custom image upon.
* Acts as the starting point; for example, an official Linux distro, a runtime environment like Node.js, Python, etc.

**Why use it?**

* Ensures your image has a standardized environment.
* Every Dockerfile must begin with a FROM (except for FROM scratch, which is minimal).

**Example:**

FROM ubuntu:20.04

**2. LABEL**

**Purpose:**

* Adds metadata or descriptive information to the image in the form of key-value pairs.

**Why use it?**

* For documentation, versioning, licensing, maintainer info, or custom metadata.
* Helps in managing and organizing images.

**Example:**

LABEL maintainer="admin@example.com" description="My custom app image"

**3. WORKDIR**

**Purpose:**

* Sets the **working directory** within the container where subsequent commands will execute.

**Why use it?**

* Simplifies commands by setting a default directory.
* Ensures files are copied/executed relative to this directory.

**Example:**

WORKDIR /app

**4. USER**

**Purpose:**

* Specifies the **user** under which subsequent commands will run inside the container.

**Why use it?**

* Security: avoid running containers as root.
* Control: run applications with the appropriate privileges.

**Example:**

USER appuser

**5. COPY**

**Purpose:**

* Copies files or directories from the host machine into the image.

**Why use it?**

* To include application code, configs, scripts, or other assets into the image.

**Example:**

COPY . /app

**6. CMD**

**Purpose:**

* Specifies the **default command** to run when the container starts.

**Why use it?**

* Defines the main process the container should execute.
* Can be overridden at runtime.

**Example:**

CMD ["python", "app.py"]

**Summary table:**

| **Instruction** | **Purpose** | **Why use it?** |
| --- | --- | --- |
| FROM | Specify base image | Sets up the environment |
| LABEL | Add metadata | Documentation, management |
| WORKDIR | Set working directory | Simplifies commands and paths |
| USER | Specify user | Security and permissions |
| COPY | Copy files into image | Prepare application files |
| CMD | Default command to run | Start application/process |

**Delete Unused Docker Networks**

Docker provides a command to remove all **dangling** or **unused** networks:

docker network prune

* **What it does:** Removes all networks not used by at least one container.
* **Prompt:** It asks for confirmation before deleting.

**To bypass confirmation:**

docker network prune -f

**2. Delete Unused Docker Volumes**

Similarly, to remove unused volumes:

docker volume prune

* **What it does:** Deletes volumes that are not referenced by any container.
* **Warning:** Be careful; volumes containing important data will be lost.

**To bypass confirmation:**

docker volume prune -f

**3. Additional cleanup: Remove all unused images, containers, networks, and volumes at once**

docker system prune -a

* **-a** flag also deletes unused images (not just dangling).
* Use with caution as it removes a lot of unused resources.

**Summary:**

| **Command** | **Purpose** | **To auto-confirm (no prompt)** |
| --- | --- | --- |
| docker network prune | Remove unused networks | docker network prune -f |
| docker volume prune | Remove unused volumes | docker volume prune -f |
| docker system prune -a | Remove unused images, containers, networks, and volumes | add -f to skip confirmation |

**1. docker stats**

**Purpose:** Provides real-time, live resource usage metrics for all running containers.

**Command:**

docker stats

**What it shows:**

* CPU %
* Memory usage / limit
* Memory % (percentage)
* Network I/O
* Block I/O (disk read/write)
* PIDs (process IDs)

**Example output:**

CONTAINER ID NAME CPU % MEM USAGE / LIMIT MEM % NET I/O BLOCK I/O PIDS

abc123 mycontainer 2.34% 50.2MiB / 1GiB 4.9% 1.2MB / 800kB 2.3MB / 1.0MB 10

**To monitor specific containers:**

docker stats container\_name\_or\_id

Great question!

**What is ENV in a Dockerfile?**

ENV is an instruction used in a Dockerfile to set environment variables inside the Docker image. These variables can be used during the build process or when the container runs.

**Purpose of ENV:**

* To define variables that configure the application's behavior.
* To set default environment variables accessible inside the container.
* To customize container behavior without modifying the image.

**1. Dockerfile for a Node.js Application**

**Assumption:** Your Node.js app has a package.json file and an entry script called app.js.

# Use official Node.js LTS image as base

FROM node:18

# Set working directory inside container

WORKDIR /usr/src/app

# Copy package.json and package-lock.json

COPY package\*.json ./

# Install dependencies

RUN npm install

# Copy the rest of the application code

COPY . .

# Expose port 3000 (or your app's port)

EXPOSE 3000

# Run the application

CMD ["node", "app.js"]

1. **Dockerfile for a Java Application (using Maven for build)**

**What is Dockerfile Optimization?**

Dockerfile optimization involves writing efficient, minimal, and faster-building Docker images. The goal is to reduce image size, improve build speed, and enhance security and maintainability.

**Why optimize a Dockerfile?**

* Faster builds and deploys
* Reduced image size
* Lower attack surface (fewer layers, smaller images)
* Better cache efficiency for incremental builds
* Easier maintenance and readability

**How to implement Dockerfile optimization**

Here are some best practices and techniques:

1. Use Minimal Base Images

* Use lightweight images like alpine variants (e.g., node:18-alpine, openjdk:17-alpine) where possible.

FROM node:18-alpine

2. Combine Commands to Reduce Layers

* Use && to chain commands into a single RUN to minimize layers.

RUN apk add --no-cache bash && \

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

3. Order Commands Effectively

* Place commands that change less frequently higher up to leverage Docker cache.
* For example, copy package.json and package-lock.json first and run npm install then copy source code to avoid reinstalling unless dependencies change.

COPY package\*.json ./

RUN npm install

COPY . .

4. Clean Up Temporary Files

* Remove temporary files, cache, or build artifacts within the same layer to reduce image size.

RUN apt-get update && apt-get install -y build-essential && \

apt-get clean && rm -rf /var/lib/apt/lists/\*

5. Use .dockerignore

* Exclude unnecessary files/folders (like node\_modules, test/, version control directories .git) to speed up build context.

.git

node\_modules

tests

6. Multi-stage Builds

* Use multi-stage builds to compile or build the app in one stage, then copy only the necessary artifacts into a minimal final image.

# Build stage

FROM maven:3.8.5-openjdk-17 AS build

WORKDIR /app

COPY pom.xml .

COPY src ./src

RUN mvn clean package -DskipTests

# Final stage

FROM openjdk:17-jdk-slim

COPY --from=build /app/target/myapp.jar /app/myapp.jar

ENTRYPOINT ["java", "-jar", "/app/myapp.jar"]

7. Use Specific Versions

* Pin dependencies and base image versions to avoid unpredictable builds and facilitate caching.

8. Utilize Build Cache

* Structure instructions to maximize Docker’s cache capabilities, avoiding unnecessary rebuilds.

Example of an optimized Dockerfile snippet:

FROM node:18-alpine

WORKDIR /app

# Copy dependency files first to leverage cache

COPY package.json package-lock.json ./

RUN npm install --production

# Then copy source code

COPY . .

EXPOSE 3000

CMD ["node", "app.js"]

Summary:

* Use minimal base images.
* Combine RUN commands.
* Leverage build cache with proper ordering.
* Use multi-stage builds.
* Clean up after installations.
* Use .dockerignore.
* Pin dependencies/versions.

1. Stop and remove all running containers forcibly

docker rm -f $(docker ps -aq)

Explanation:

* docker ps -aq: Lists all container IDs (both running and stopped).
* docker rm -f: Forcefully removes all containers by ID.

2. Remove all images

docker rmi -f $(docker images -aq)

Explanation:

* docker images -aq: Lists all image IDs.
* docker rmi -f: Forcefully removes all images, even if they are used by containers (which must have been stopped and removed first).

Full command sequence:

docker rm -f $(docker ps -aq) # Stop and remove all containers

docker rmi -f $(docker images -aq) # Remove all images

Note:

* Use these commands wit**h caution, as they will delete all containers and images and may cause data loss if images or container data are important.**
* **Best to ensure unneeded containers/images are safe to delete.**

**1. Use Minimal Base Images**

* **Choose lightweight images like alpine variants (e.g., node:18-alpine, python:3.11-alpine) to reduce size.**
* **Example:**

**FROM node:18-alpine**

**2. Leverage Docker Cache by Ordering Commands**

* **Place commands that change less frequently at the top.**
* **Separate copying dependencies from copying source code.**
* **Example:**
* **COPY package.json package-lock.json ./**
* **RUN npm install**

**COPY . .**

**3. Combine Commands to Minimize Layers**

* **Use && to chain related commands into a single RUN.**
* **Example:**

**RUN apk add --no-cache bash && rm -rf /var/cache/apk/\***

**4. Clean Up Temporary Files and Cache in Same Layer**

* **Remove temporary files, caches, or build artifacts immediately after their use to keep image lean.**
* **Example:**

**RUN apt-get update && apt-get install -y build-essential && apt-get clean && rm -rf /var/lib/apt/lists/\***

**5. Use Multi-Stage Builds**

* **Build large dependencies or compile code in intermediate stages.**
* **Copy only necessary artifacts into lightweight final images.**
* **Example:**
* **FROM maven:3.8.5-openjdk-17 AS build**
* **# build process**
* **FROM openjdk:17-jdk-slim**

**COPY --from=build /app/target/app.jar /app/app.jar**

**6. Apply .dockerignore**

* **Exclude unnecessary files and folders (like .git, node\_modules, tests, etc.) from the build context to speed up build time and reduce image size.**

**7. Pin Dependencies and Base Image Versions**

* **For reproducibility and cache effectiveness, specify exact versions:**
* **FROM node:18.15.0**

**RUN npm install some-lib@1.2.3**

**Example of an optimized Dockerfile snippet for Node.js app:**

**FROM node:18-alpine**

**WORKDIR /app**

**# Copy dependency files and install dependencies**

**COPY package.json package-lock.json ./**

**RUN npm install --production**

**# Copy only necessary source files**

**COPY src ./src**

**EXPOSE 3000**

**CMD ["node", "src/index.js"]**

**Summary:**

An optimized Dockerfile minimizes image size, build time, and layers by:

* Choosing appropriate base images.
* Ordering commands to maximize cache efficiency.
* Combining commands where possible.
* Using multi-stage builds.
* Cleaning up caches.
* Excluding unnecessary files via .dockerignore.

| **Aspect** | **Containerization** | **Virtualization** |
| --- | --- | --- |
| Architecture | Shares host OS kernel; isolates at process level | Full OS virtualization; separate kernel for each VM |
| Resource Usage | Less; lightweight, fast to start | More; heavier, slower to start |
| OS Compatibility | Runs on host OS; usually same OS as host | Can run different OSes; guest OS included |
| Efficiency | High; supports thousands of containers per host | Lower; fewer VMs per host |
| Use Cases | App deployment, microservices, DevOps | Server consolidations, different OS environments |
| Isolation | Process-level, less isolation | Full isolation |