MAYANK SINGH

What is Docker in brief.

Docker is a platform that enables developers to build, run, and manage applications in **containers**. Containers are lightweight, portable, and run the same regardless of the environment, making them ideal for software deployment.

Why Use Docker?

1. **Portability:** Applications run in isolated environments, ensuring they behave the same across development, testing, and production.
2. **Efficiency:** Containers are lightweight, consuming fewer resources than virtual machines.
3. **Faster Deployment:** Containers launch in seconds, enabling quicker application development and scaling.
4. **Consistency:** Docker ensures the same environment across multiple platforms, reducing bugs caused by differing system configurations.

Key Concepts of Docker:

1. **Docker Engine:** The core component responsible for running and managing containers.

2. **Containers:** Lightweight, isolated environments where applications run.
3. **Docker Image:** A template used to create containers, which includes the application code and its dependencies.
4. **Dockerfile:** A script used to define how a Docker image is built.
5. **Docker Hub:** A repository for storing and sharing Docker images.

Common Use Cases:

- Microservices architecture
- CI/CD pipelines
- Application isolation
- Testing and development environments

Basic Docker Commands:

- docker build: Build an image from a Dockerfile.
- docker run: Run a container from an image.
- docker ps: List running containers.
- docker stop: Stop a running container.
- docker pull: Download an image from Docker Hub.

1. docker build

This command creates a Docker image from a Dockerfile. A Dockerfile contains instructions on how to build the image, including installing dependencies, copying files, and setting up the environment.

Example:

docker build -t my-app .

- **-t my-app**: Tags the image as "my-app."
- **.**: Refers to the current directory where the Dockerfile is located.

This command builds an image using the Dockerfile in the current directory and tags it as my-app. The image is stored locally for later use.

2. docker run

This command starts a container from an image. If the image is not found locally, Docker will pull it from Docker Hub.

Example:

docker run -d -p 8080:80 my-app

- **-d**: Runs the container in detached mode (in the background).
- **-p 8080:80**: Maps port 8080 on the host to port 80 in the container.
- **my-app**: The image name.

This will start the container from the my-app image and make the application accessible on port 8080 of your local machine.

3. docker ps

This command lists all running containers. It gives information like container IDs, names, ports, and the image they are based on.

Example:

docker ps

Output might look like:

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

d34db33f1c3a my-app "nginx -g ..." 10 seconds ago Up 10 seconds 0.0.0.0:8080->80/tcp cool_pear

This shows that a container (cool_pear) is running from the my-app image, and port 8080 on your host is mapped to port 80 in the container.

4. docker stop

This command stops a running container. You can stop it using the container ID or name.

Example:

docker stop cool_pear

This stops the container with the name cool_pear. Alternatively, you can use the container ID:

docker stop d34db33f1c3a

5. docker pull

This command downloads an image from Docker Hub if it's not already available locally.

Example:

docker pull nginx

This pulls the official nginx image from Docker Hub and stores it locally for future use.

Brief Explanation of Each Step:

1. **Build (docker build)**: Compiles a Docker image from a set of instructions (Dockerfile).
2. **Run (docker run)**: Starts a container using a specified image, mapping ports if needed.
3. **List (docker ps)**: Lists all containers currently running on your system.
4. **Stop (docker stop)**: Stops a running container gracefully.
5. **Pull (docker pull)**: Downloads an image from Docker Hub so it can be used locally.

Docker vs Virtual Machines:

- **Docker:** Shares the host OS kernel, leading to faster and more efficient operations.
- **VM:** Each VM has its own OS, which makes them more resource-heavy.

Docker (Containers)	Virtual Machines (VMs)
Shares the host OS kernel with isolated user spaces (containers)	Runs a full OS with its own kernel on a hypervisor
Starts in seconds (lightweight)	Takes minutes to boot (heavier due to full OS)
Lightweight, shares OS kernel; efficient use of system resources	Resource-heavy; each VM has its own OS
Process-level isolation with namespaces and cgroups	Complete OS-level isolation, strong security boundaries
Near-native performance	Performance overhead due to virtualized hardware

Requires less storage, as containers share the OS	Requires more storage due to full OS installation
Highly portable across different environments	Less portable; dependent on hypervisor and OS versions
Ideal for microservices, testing, and quick deployments	Suitable for running multiple full OS environments
Less isolation compared to VMs but improving (depends on the host OS security)	Stronger isolation as each VM runs a separate OS
Easier to manage with tools like Docker Compose, Docker Swarm, Kubernetes	Requires more management and setup (VM managers, hypervisors)
Minimal overhead as it uses the host OS kernel	Significant overhead due to running a full OS per VM

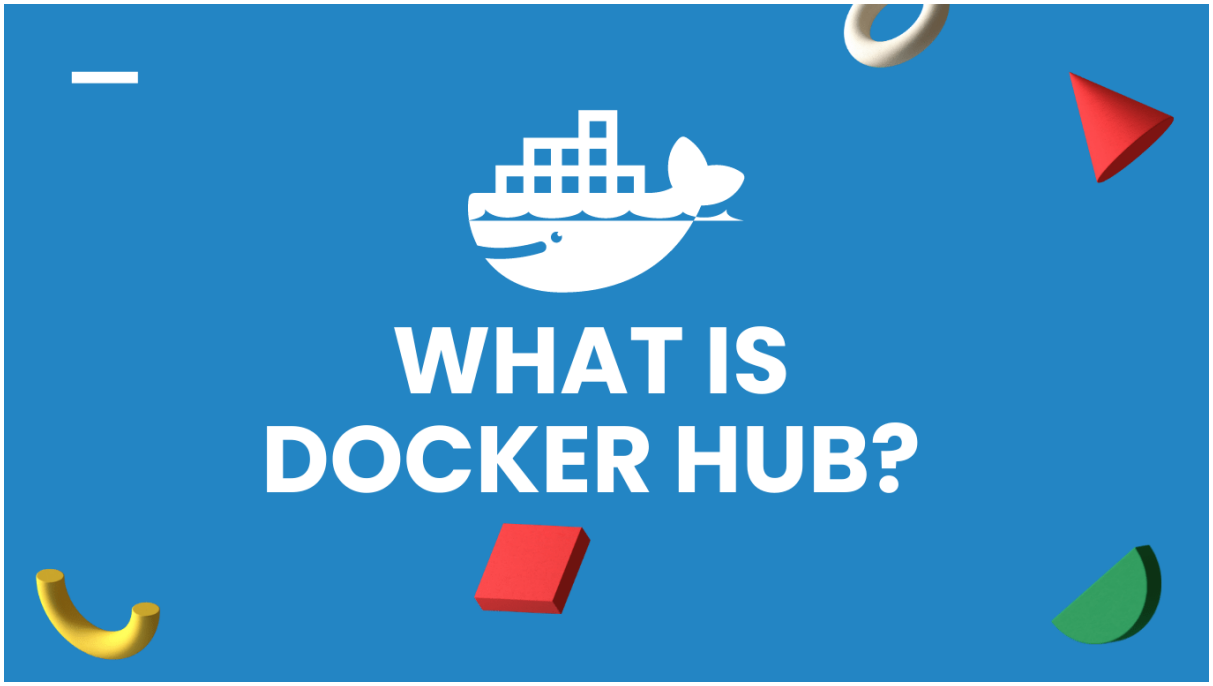
Hardware-level virtualization

Hardware-level virtualization refers to the technology that allows multiple operating systems to run simultaneously on a single physical machine by using a layer of software known as a **hypervisor**. This hypervisor creates and manages **virtual machines (VMs)**, each running its own OS, by partitioning the hardware resources (CPU, memory, storage) of the host machine.

In hardware-level virtualization:

- The hypervisor interacts directly with the **physical hardware**.
- It allocates hardware resources to each VM while ensuring isolation and independence between them.
- Examples of hypervisors include **VMware ESXi**, **Microsoft Hyper-V**, and **KVM**.

Docker Hub



Docker Hub is a cloud-based registry service where Docker users can **store, share, and distribute** Docker images. It is the default repository used by Docker to find and pull images when running containers.

Key Features of Docker Hub:

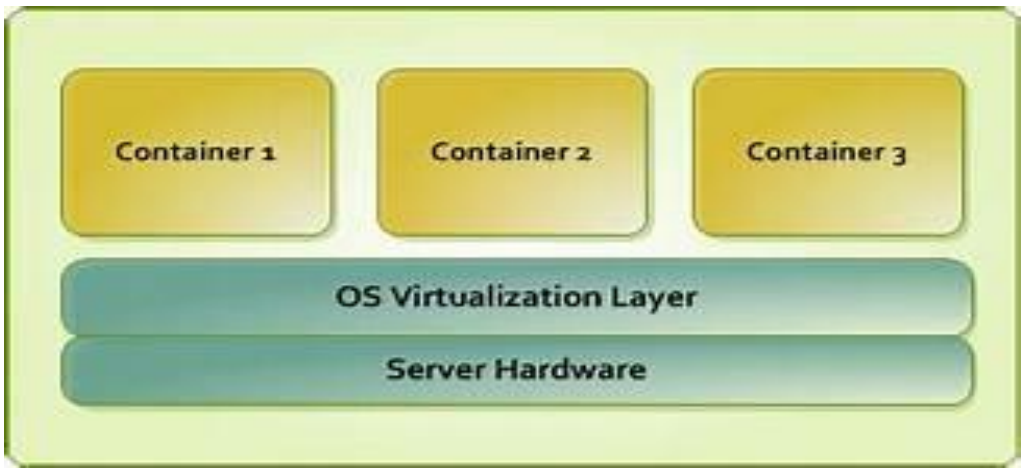
1. **Image Repository**: Hosts both **public** and **private** images that can be accessed by anyone (public) or restricted to specific users (private).
2. **Official Images**: Provides pre-built, trusted images for common applications (like **Nginx**, **MySQL**, etc.).
3. **Automated Builds**: Allows automatic building of images from GitHub or Bitbucket repositories.
4. **Tagging**: Supports multiple versions of the same image through tags (e.g., nginx:latest, nginx:1.19).
5. **Image Discovery**: Users can search for images shared by the Docker community and official publishers.

Example of Pulling an Image from Docker Hub:

docker pull nginx

This command pulls the official **Nginx** image from Docker Hub, allowing you to use it to run containers on your local system.

OS-level virtualization



OS-level virtualization, also known as **containerization**, is a type of virtualization where the **operating system (OS) kernel** allows multiple isolated user-space instances, called **containers**, to run on the same OS. Unlike hardware-level virtualization, which uses a hypervisor to run multiple OSes, OS-level virtualization uses a single OS kernel to run multiple isolated environments.

Key Features of OS-level Virtualization:

1. **Shared Kernel**: All containers share the same OS kernel, but they run isolated from each other.
2. **Efficiency**: Since containers share the OS, they consume fewer resources than virtual machines, resulting in **better performance** and **faster start times**.
3. **Lightweight**: Containers only contain the application code and dependencies, unlike VMs that require a full OS.
4. **Isolation**: Each container operates in its own isolated environment, but they share the underlying kernel with the host.

Examples of OS-level Virtualization:

- **Docker**: One of the most popular containerization platforms.
- **LXC (Linux Containers)**: Another method to create containers on Linux.

Use Case:

- OS-level virtualization is commonly used for **microservices**, **scalable applications**, and **continuous integration/continuous deployment (CI/CD)** pipelines because of its efficiency and portability.

A container contains an operating system or not?

Containers **do not contain a full OS**. Instead, they share the host operating system's kernel. However, each container includes the necessary **application code**, **libraries**, and **dependencies** required to run the application.

Here's how it works:

- Containers run on top of the **host OS kernel**, but they remain isolated from each other.
- While they do not have a full OS, containers have their own **user space**, meaning they can have their own filesystem, networking, and process space, which makes them appear like independent systems to applications running inside them.

Advantages of using Docker:

1. Portability

- Docker ensures that applications run the same way across different environments, from development to production, because containers bundle the application and its dependencies together.

2. Efficiency

- Containers share the host OS kernel, making them lightweight compared to virtual machines. This reduces resource usage (CPU, memory) and improves performance.

3. Fast Deployment

- Containers can be launched in seconds due to their lightweight nature. This speeds up development, testing, and scaling processes.

4. Scalability

- Docker makes it easy to scale applications horizontally by creating multiple instances of containers and distributing the load across them.

5. Consistency

- Containers ensure that the same code runs with the same dependencies, libraries, and configurations across different environments, reducing compatibility issues.

6. Isolation

- Docker containers are isolated from one another and from the host system, improving security and preventing interference between applications.

7. Version Control and Rollbacks

- Docker allows you to maintain different versions of an application in the form of images. You can quickly roll back to previous versions if needed.

8. Simplified DevOps

- Docker facilitates Continuous Integration and Continuous Deployment (CI/CD) workflows by streamlining the build, test, and deployment processes.

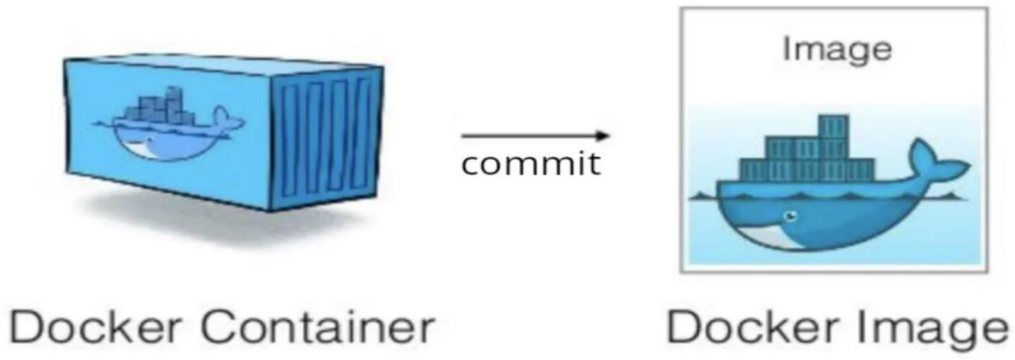
9. Cost-Effectiveness

- By efficiently using system resources, Docker helps reduce costs, especially in large-scale environments, as it requires fewer resources than traditional virtualization.

10. Extensive Ecosystem

- Docker has a large ecosystem, including Docker Hub, which provides thousands of pre-built images that can be reused and customized for different purposes.

Here's a brief explanation of **Docker Image** and **Docker Container**:



1. Docker Image:

- A **Docker Image** is a **read-only template** that contains the application code, libraries, dependencies, and other configuration files required to run an application.
- It is the blueprint for creating Docker containers.
- Images are **built using a Dockerfile**, which contains a set of instructions for assembling the image.
- Images are **immutable**, meaning they cannot be changed once created.

Example:

An image might contain a Python application with all necessary Python libraries installed.

2. Docker Container:

- A **Docker Container** is a **running instance** of a Docker image.
- Containers are **isolated environments** where applications run, using the code and dependencies defined in the image.
- Containers are **ephemeral** and can be started, stopped, and deleted. Multiple containers can be created from the same image.
- Containers are portable and lightweight because they share the host OS kernel.

Example:

If you have an image with a Python web app, when you run the container, it will start the app in an isolated environment.

Key Difference:

- **Image**: A template or blueprint.
- **Container**: A running instance created from an image.

A few **disadvantages** and limitations

1. Limited to Linux-based Kernel

- Docker relies on the **host OS kernel**, which means it works best for **Linux-based applications**. Windows containers are supported, but they are less mature and have limitations compared to Linux containers.

2. Weaker Isolation Compared to VMs

- Containers share the host OS kernel, so they offer **less isolation** compared to Virtual Machines (VMs), which have a full OS. This can pose security risks, as vulnerabilities in the host OS could potentially affect all containers.

3. Persistent Data Storage

- Containers are **ephemeral**, meaning they don't store data persistently by default. Managing **persistent storage** in Docker can be complex and often requires the use of **volumes** or external storage solutions.

4. Complex Networking

- Networking in Docker can be more **complex to manage**, especially when multiple containers are interacting across different networks or need to communicate with external systems.

5. Limited GUI Support

- Docker containers are primarily used for command-line and backend applications. **Running GUI-based applications** in Docker containers is challenging and not as seamless as in VMs.

6. Security Vulnerabilities

- Since Docker containers share the host kernel, a **compromised container** can potentially impact other containers or the host system. Additionally, using public images from Docker Hub can pose risks if the image contains malicious or outdated code.

7. Learning Curve

- Docker requires a **learning curve** for developers and system administrators, particularly when integrating with **orchestration tools** like Kubernetes, Docker Swarm, or complex multi-container applications.

8. Performance Overhead in Some Cases

- While Docker is lightweight, certain applications might still experience **performance overhead**, especially in **I/O-heavy workloads** or when compared to running natively on the host OS.

9. Not a Complete Virtualization Solution

- Docker is designed for **isolating applications**, not full OS environments like virtual machines. This makes it less suitable for applications that require a fully isolated OS, where VMs may still be a better fit.

Key Component of Docker and their Parameter

1. Docker Engine

- **Definition**: Docker Engine is the core part of Docker that runs and manages containers. It consists of several components like the Docker daemon, REST API, and CLI (command-line interface).
- **Function**: It is responsible for building, running, and managing Docker containers.
- It handles the communication between the Docker client and the operating system.

2. Docker Daemon (dockerd)

- **Definition**: The Docker daemon is the background service that runs on the host machine and is responsible for managing Docker objects such as containers, images, networks, and volumes.
- **Function**: It listens to Docker API requests, manages Docker containers, and communicates with other daemons for orchestration.

3. Docker Client (docker)

- **Definition**: The Docker client is the command-line tool that allows users to interact with the Docker daemon.
- **Function**: Users run Docker commands via the client (like docker run, docker build), and the client sends these commands to the Docker daemon.

4. Docker Image

- **Definition**: A Docker image is a lightweight, standalone, and executable package that includes everything needed to run a container—such as the code, runtime, libraries, environment variables, and configuration files.
- **Function**: It serves as a template for creating Docker containers. Images are immutable and can be shared and reused.

5. Container

- **Definition**: A container is a runtime instance of a Docker image. It's the running environment where applications are executed.
- **Function**: Containers isolate the application from the host environment, providing a consistent environment for development, testing, and production.

6. Docker Hub

- **Definition**: Docker Hub is a cloud-based repository where Docker users and developers can store and distribute their Docker images.
- **Function**: It allows developers to upload, search, and download pre-built images for their applications or use images from other developers.

7. Dockerfile

- **Definition**: A Dockerfile is a text file that contains a series of instructions used to create a Docker image.
- **Function**: It automates the process of building a Docker image. Instructions in a Dockerfile can specify the base image, copy files, install dependencies, set environment variables, and run commands.

8. Docker Compose

- **Definition**: Docker Compose is a tool for defining and running multi-container Docker applications. It uses a YAML file to configure the application's services, networks, and volumes.
- **Function**: It allows you to run multiple containers as a service, manage them with one command, and define dependencies between them.

9. Docker Swarm

- **Definition**: Docker Swarm is a native clustering and orchestration tool for Docker. It allows you to create a cluster of Docker hosts, and Docker Swarm manages the cluster for you.
- **Function**: It helps with container orchestration by managing multiple Docker hosts as a single system, automating scaling, and load balancing.

10. Docker Volume

- **Definition**: Docker Volumes are used to persist data generated by Docker containers. Volumes are stored on the host filesystem, outside the container's file system.
- **Function**: Volumes allow you to share data between containers or between the container and the host, ensuring that important data is not lost when containers are stopped or removed.

11. Docker Network

- **Definition**: Docker Networks allow Docker containers to communicate with each other and with other systems.
- **Function**: Docker provides different network drivers like bridge, host, and overlay to manage container-to-container communication, enabling applications to be connected in a distributed way.

12. Registry

- **Definition**: A registry is a repository for storing and distributing Docker images.
- Docker Hub is a public registry, while users can also set up private registries.
- **Function**: It allows developers to push and pull Docker images, helping to manage version control and distribution of images.

13. Layered File System

- **Definition**: Docker uses a union file system to create images layer by layer. Each layer represents a change or instruction in the Dockerfile.
- **Function**: This layered approach optimizes storage and reusability, as each layer is immutable and can be reused across multiple containers.

14. Namespaces

- **Definition**: Namespaces provide isolation for Docker containers. It is a Linux kernel feature that ensures containers do not interfere with each other.
- **Function**: Each container gets its own separate namespace for processes, network, and file systems, ensuring isolation and security.

15. Control Groups (Cgroups)

- **Definition**: Cgroups limit, prioritize, and isolate the resources (like CPU, memory, I/O) a Docker container can use.
- **Function**: This ensures that no container consumes more than its fair share of system resources, protecting the host and other containers.

16. OverlayFS

- **Definition**: OverlayFS is the file system Docker uses to manage image layers.
- It allows Docker to layer multiple file systems on top of each other.
- **Function**: It optimizes storage by sharing the same layers across multiple containers and copying only the necessary changes to a writable layer.

17. Entry Point

- **Definition**: An entry point is a command that is executed when a container starts.
- **Function**: It defines the default application that will run inside the container when no command is specified.

18. CMD

- **Definition**: The CMD instruction in a Dockerfile provides the default command to run in the container.
- **Function**: If no command is provided when running a container, Docker will execute the command specified in CMD.

19. Docker Run

- **Definition**: The docker run command is used to create and start a new container from a Docker image.
- **Function**: It starts the container, runs the process defined by the Dockerfile or command, and manages the lifecycle of the container

Docker Basic Commands with Examples and Explanations:

1. docker --version

- Displays the installed Docker version.
- docker --version

2. docker pull

- Downloads a Docker image from a registry like Docker Hub.
- docker pull ubuntu

3. docker images

- Lists all available images on your system.

- docker images

4. docker rmi <image_id>

- Removes a Docker image by its ID.
- docker rmi abc123

5. docker ps

- Lists all running containers.
- docker ps

6. docker ps -a

- Lists all containers, including stopped ones.
- docker ps -a

7. docker run <image_name>

- Creates and starts a new container from a specified image.
- docker run ubuntu

8. docker run -it <image_name>

- Starts an interactive container session.
- docker run -it ubuntu /bin/bash

9. docker start <container_id>

- Starts an existing stopped container.
- docker start abc123

10. docker stop <container_id>

- Stops a running container.

- docker stop abc123

11. docker restart <container_id>

- Restarts a running or stopped container.
- docker restart abc123

12. docker rm <container_id>

- Removes a stopped container.
- docker rm abc123

13. docker exec -it <container_id> <command>

- Executes a command in a running container.
- docker exec -it abc123 /bin/bash

14. docker logs <container_id>

- Displays the logs of a container.
- docker logs abc123

15. docker inspect <container_id>

- Provides detailed information about a container.
- docker inspect abc123

16. docker build -t <image_name> <Dockerfile_path>

- Builds an image from a Dockerfile.
- docker build -t my_image .

17. docker tag <image_id> repository:tag

- Tags an image for a repository.
- docker tag abc123 myrepo/myimage:v1

18. **docker push** repository:tag
    - Pushes an image to a Docker registry.
    - docker push myrepo/myimage:v1
19. **docker pull repository:tag**
    - Pulls an image from a Docker registry.
    - docker pull myrepo/myimage:v1
20. **docker network ls**
    - Lists all Docker networks.
    - docker network ls
21. **docker network create <network_name>**
    - Creates a new Docker network.
    - docker network create my_network
22. **docker network rm <network_name>**
    - Removes an existing Docker network.
    - docker network rm my_network
23. **docker volume ls**
    - Lists all Docker volumes.
    - docker volume ls
24. **docker volume create <volume_name>**
    - Creates a new Docker volume.
    - docker volume create my_volume
25. **docker volume rm <volume_name>**

- Removes a Docker volume.
- docker volume rm my_volume

26. **docker-compose up**
    - Starts services defined in a docker-compose.yml file.
    - docker-compose up
27. **docker-compose down**
    - Stops and removes services created by Docker Compose.
    - docker-compose down
28. **docker-compose ps**
    - Lists the status of Docker Compose services.
    - docker-compose ps
29. **docker commit <container_id> <new_image_name>**
    - Creates a new image from a container's changes.
    - docker commit abc123 my_image:v2
30. **docker system prune**
    - Removes unused data (stopped containers, networks, etc.).
    - docker system prune
31. **docker stats**

- Displays a live stream of container resource usage statistics.
- docker stats

32. **docker save -o <output_file> <image_name>**
    - Saves an image to a tar archive.
    - docker save -o my_image.tar my_image:v1
33. **docker load -i <input_file>**
    - Loads an image from a tar archive.
    - docker load -i my_image.tar
34. **docker export <container_id> -o <output_file>**
    - Exports a container's filesystem to a tar archive.
    - docker export abc123 -o container_fs.tar
35. **docker import <input_file>**
    - Imports a container’s filesystem from a tar archive as an image.
    - docker import container_fs.tar
36. **docker pause <container_id>**

- Pauses all processes within a container.
- docker pause abc123

37. **docker unpause <container_id>**

- Unpauses a paused container, resuming its processes.
- docker unpause abc123

38. **docker rename <old_name> <new_name>**

- Renames an existing container.
- docker rename old_container_name new_container_name

39. **docker update <container_id> --cpus <value>**

- Updates the resource limits for a container (e.g., CPU).
- docker update abc123 --cpus 2

40. **docker attach <container_id>**

- Attaches to a running container’s standard input, output, and error streams.
- docker attach abc123

41. **docker cp <container_id>:/path/to/file <host_destination>**

- Copies files from a container to the host machine.
- docker cp abc123:/var/log/nginx.log /home/user/nginx.log

42. **docker diff <container_id>**

- Shows the changes made to a container's filesystem.
- docker diff abc123

43. **docker login**

- Logs into a Docker registry (Docker Hub or private).
- docker login

44. **docker logout**

- Logs out from a Docker registry.
- docker logout

45. **docker history <image_name>**

- Displays the history of an image's layers.
- docker history ubuntu

46. **docker export <container_id> -o <filename.tar>**

- Exports a container's filesystem as a tar archive.
- docker export abc123 -o container_backup.tar

47. **docker import <filename.tar> <image_name>**

- Imports a tarball as a Docker image.
- docker import container_backup.tar new_image_name

48. **docker commit <container_id> <image_name>**

- Creates a new image from a container's changes.
- docker commit abc123 my_custom_image

49. **docker exec <container_id> <command>**

- Executes a command inside a running container.
- docker exec abc123 ls /app

50. **docker stats**

- Displays a live stream of resource usage statistics for running containers.
- docker stats

51. **docker events**

- Displays real-time events from the Docker daemon.
- docker events

52. **docker version**

- Displays the Docker version information.
- docker version

53. **docker info**

- Displays system-wide information about Docker.
- docker info

54. **docker wait <container_id>**

- Blocks until a container stops, then prints the exit code.
- docker wait abc123

55. **docker prune**

- Cleans up unused containers, networks, images (dangling), and volumes.
- docker system prune

56. **docker image prune**

- Removes unused or dangling images.
- docker image prune

57. **docker volume prune**

- Removes unused volumes.
- docker volume prune

58. **docker network prune**

- Removes unused networks.
- docker network prune

59. **docker network create <network_name>**

- Creates a new network for containers.
- docker network create my_network

60. **docker network ls**

- Lists all available Docker networks.
- docker network ls

61. **docker network inspect <network_name>**

- Shows detailed information about a specific network.
- docker network inspect my_network

62. **docker volume create <volume_name>**

- Creates a new Docker volume.
- docker volume create my_volume

63. **docker volume ls**

- Lists all Docker volumes.
- docker volume ls

64. **docker volume inspect <volume_name>**

- Shows detailed information about a specific volume.
- docker volume inspect my_volume

65. **docker-compose up**

- Starts containers defined in a docker-compose.yml file.

- docker-compose up

66. **docker-compose down**

- Stops and removes containers, networks, images, and volumes created by docker-compose up.
- docker-compose down

67. **docker-compose build**

- Builds or rebuilds services defined in the docker-compose.yml file.
- docker-compose build

68. **docker-compose stop**

- Stops running containers without removing them.
- docker-compose stop

69. **docker-compose restart**

- Restarts all services in a docker-compose setup.
- docker-compose restart

70. **docker-compose ps**

- Lists all containers in a docker-compose environment.
- docker-compose ps

71. **docker-compose logs**

- Displays output from the running containers managed by docker-compose.
- docker-compose logs

72. **docker-compose rm**

- Removes stopped service containers.
- docker-compose rm

73. **docker-compose pull**

- Pulls images for services defined in the docker-compose.yml file.
- docker-compose pull

74. **docker-compose exec <service_name> <command>**

- Executes a command inside a running service container.
- docker-compose exec web ls /app

75. **docker-compose scale <service_name>=<num_instances>**

- Scales the number of containers for a service.
- docker-compose scale web=3

76. **docker tag <image> <new_image_tag>**

- Tags an image with a new tag.
- docker tag my_image my_image:v2

77. **docker save -o <filename.tar> <image_name>**

- Saves an image to a tar archive.
- docker save -o my_image.tar my_image

78. **docker load -i <filename.tar>**

- Loads an image from a tar archive.

- docker load -i my_image.tar

79. **docker stop $(docker ps -q)**

- Stops all running containers.
- docker stop $(docker ps -q)

80. **docker rm $(docker ps -a -q)**

- Removes all stopped containers.
- docker rm $(docker ps -a -q)

81. **docker rmi $(docker images -q)**

- Removes all images.
- docker rmi $(docker images -q)

82. **docker rename <old_container_name> <new_container_name>**

- Renames a running or stopped container.
- docker rename my_old_container my_new_container

83. **docker update --cpu-shares <value> <container_id>**

- Updates CPU shares for a container.
- docker update --cpu-shares 512 abc123

84. **docker update --memory <value> <container_id>**

- Updates memory limit for a container.
- docker update --memory 512m abc123

85. **docker daemon**

- Runs the Docker daemon manually (typically done automatically).
- docker daemon

86. **docker logs -f <container_id>**

- Follows the logs of a container in real-time.
- docker logs -f abc123

87. **docker history <image_name>**

- Shows the history of an image's layers.
- docker history ubuntu

88. **docker inspect <container_id>**

- Shows detailed information about a container or image.
- docker inspect abc123

89. **docker diff <container_id>**

- Shows changes made to a container’s filesystem.
- docker diff abc123

90. **docker cp <container_id>:<source_path> <destination_path>**

- Copies files from a container to the host system.
- docker cp abc123:/app/file.txt /tmp

91. **docker wait <container_id>**

- Blocks until a container stops, then prints its exit code.

- docker wait abc123

92. **docker kill <container_id>**

- Forcefully stops a running container.
- docker kill abc123

93. **docker attach <container_id>**

- Attaches to a running container to view logs or interact with it.
- docker attach abc123

94. **docker pause <container_id>**

- Pauses all processes within a container.
- docker pause abc123

95. **docker unpause <container_id>**

- Resumes all processes within a paused container.
- docker unpause abc123

96. **docker top <container_id>**

- Shows running processes within a container.
- docker top abc123

97. **docker port <container_id>**

- Displays a list of port mappings for a container.
- docker port abc123

98. **docker exec -it <container_id> /bin/bash**

- Opens an interactive shell inside a running container.

- docker exec -it abc123 /bin/bash

99. **docker network create <network_name>**

- Creates a new network for Docker containers to communicate.
- docker network create my_network

100. **docker network ls**

- Lists all available networks.
- docker network ls