Docker Questions

Q1: What is difference between containerization and virtualization

Q2: Explain Docker architecture

1. Docker Host
2. Docker Client
3. Daemon
4. Docker Hub or Registry

Q3:How can you check docker is installed or not ?

Q4: Which user by default can run the docker commands

Q5: I have a user by name Ubuntu I want Ubuntu user to run the docker commands how can he run the docker commands

Q6: List out the images in docker host

Q7:List the running containers in docker host

Q8:What are the different status of docker containers

Q9:Create and run a new container using below details

Image:jenkins/jenkins:jdk17

Containername: Jenkins

Port:8080

Mode:Detached mode

Q10:User wants to install git inside a container by name ‘jenkins’ how can he install?

Q11: How can I list the images,running,stopped and paused containers using single command?

Q12:How can I troubleshoot a container when it is not running?

Q13:What is persistent volume in docker.

Q14: What are different volumes in docker.

Q15: What is the use of docker-compose.Can you explain about docker-compose.yml file.

Q16: How can we use docker commands

Common Commands:

run

exec

ps

build

pull

push

images

login

logout

search

version

info

Q17:Write down a command to connect and disconnect a container to a host network

Q18: What is the default network in docker.What is inspect command used

Q19: What is the none network in docker .How can we disconnect from bridge network and connect to none network

Q20: Default location od docker ,Images and containers

Q21:What is a docker image

Q22: Why do we use following in Dockerfile

FROM

LABEL

WORKDIR

USER

COPY

CMD

Q23:I want to delete the unused networks and volumes

Q24: How can I check the resource usage of running containers.

Q25: What is ENV in a docker file and how is it used

Q26: Write a docker file for nodejs and java application

Q27: What is dockerfile optimization and how do you implement it

Q28: Forecfully delete all the running containers and images

# Docker Questions and Answers

## **Q1: What is the difference between containerization and virtualization?**

\*\***Containerization**\*\* and \*\*Virtualization\*\* are both technologies for running isolated applications, but they differ in architecture and resource usage:

- \*\***Virtualization**\*\*:

- Runs multiple complete operating systems (guests) on a single physical machine (host)

- Each VM includes a full OS, binaries, libraries, and application

- Uses a hypervisor (Type 1 like ESXi or Type 2 like VirtualBox)

- Higher resource overhead due to running multiple OS kernels

- Strong isolation between VMs

- \*\*Containerization\*\*:

- Shares the host OS kernel but isolates processes

- Containers include only the application and its dependencies

- Much lighter weight than VMs (faster startup, less disk/memory usage)

- Uses container runtime (like Docker) instead of hypervisor

- Isolation is at process level rather than machine level

Key difference: Virtualization virtualizes hardware to run multiple OSes, while containerization virtualizes the OS to run multiple isolated user spaces.

## **Q2: Explain Docker architecture**

Docker uses a client-server architecture with these main components:

### I. Docker Host

- The physical or virtual machine running the Docker daemon

- Responsible for running containers and managing Docker objects (images, containers, networks, volumes)

- Contains the Docker daemon, images, containers, networks, and storage

### II. Docker Client

- The primary interface for users (docker CLI)

- Sends commands to the Docker daemon via REST API

- Can connect to local or remote daemons

### III. Daemon (dockerd)

- The background service running on the host

- Listens for Docker API requests and manages Docker objects

- Can communicate with other daemons to manage Docker services

### IV. Docker Hub or Registry

- A repository for Docker images (Docker Hub is the default public registry)

- Stores images in repositories

- Can be public or private

- Alternatives include AWS ECR, Google Container Registry, Azure Container Registry, or self-hosted registries

## **Q3: How can you check if Docker is installed or not?**

Run the following command in your terminal:

```bash

docker --version

```

If Docker is installed, it will display the version information like:

```

Docker version 20.10.12, build e91ed57

```

If not installed, you'll get a "command not found" error. You can also check the Docker daemon status with:

```bash

systemctl status docker

```

or for more detailed information:

```bash

docker info

```

## **Q4: Which user by default can run the docker commands?**

By default, only the \*\*root user\*\* or users in the \*\*docker group\*\* can run Docker commands.

After Docker installation, you need to either:

1. Run Docker commands with `sudo` prefix, or

2. Add your user to the docker group with:

```bash

sudo usermod -aG docker $USER

```

Then log out and log back in for the changes to take effect.

## **Q5: I have a user by name Ubuntu I want Ubuntu user to run the docker commands how can he run the docker commands**

To allow the 'ubuntu' user to run Docker commands without sudo:

1. Add the user to the docker group:

```bash

sudo usermod -aG docker ubuntu

```

2. The user needs to log out and log back in for the group membership to be updated.

3. Verify the user can run Docker commands:

```bash

su - ubuntu

docker run hello-world

```

If you don't want to log out, you can run:

```bash

newgrp docker

```

to activate the new group membership in the current session.

## **Q6: List out the images in docker host**

To list all Docker images stored on your host machine:

```bash

docker images

```

or the newer syntax:

```bash

docker image ls

```

This will show:

- REPOSITORY: The image name

- TAG: The version tag (default is "latest")

- IMAGE ID: Unique identifier for the image

- CREATED: When the image was created

- SIZE: The image size

To list image IDs only:

```bash

docker images -q

```

## **Q7: List the running containers in docker host**

To list currently running containers:

```bash

docker ps

```

or

```bash

docker container ls

```

To list all containers (running and stopped):

```bash

docker ps -a

```

or

```bash

docker container ls -a

```

Columns displayed:

- CONTAINER ID: Unique container identifier

- IMAGE: The image used to create the container

- COMMAND: The command running in the container

- CREATED: When the container was created

- STATUS: Current state (Up, Exited, etc.)

- PORTS: Port mappings

- NAMES: The container name

## **Q8: What are the different statuses of docker containers**

Docker containers can have several statuses:

1. \*\*Created\*\*: Container has been created but not started

2. \*\*Running\*\*: Container is currently active and running

3. \*\*Restarting\*\*: Container is in the process of restarting

4. \*\*Paused\*\*: Container processes have been paused (freezed)

5. \*\*Exited\*\*: Container has stopped running

6. \*\*Dead\*\*: Container failed to stop correctly and is in an unrecoverable state

Common status messages you'll see:

- Up [time]: Running for specified time

- Up About a minute: Recently started

- Exited (0): Stopped normally with exit code 0

- Exited (137): Stopped by SIGKILL (often out of memory)

- Exited (143): Stopped by SIGTERM (graceful shutdown)

## **Q9: Create and run a new container using below details**

To create and run a Jenkins container with the specified parameters:

```bash

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

```

Explanation of flags:

- `-d`: Detached mode (run in background)

- `--name Jenkins`: Names the container "Jenkins"

- `-p 8080:8080`: Maps host port 8080 to container port 8080

- `jenkins/jenkins:jdk17`: The image to use (Jenkins with JDK 17)

After running, you can access Jenkins at `http://localhost:8080`.

To verify it's running:

```bash

docker ps

```

## **Q10: User wants to install git inside a container by name 'jenkins' how can he install?**

To install Git inside the running 'jenkins' container:

1. First, get a shell inside the container:

```bash

docker exec -it jenkins /bin/bash

```

(or `/bin/sh` if bash isn't available)

2. Once inside the container, install Git:

```bash

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

```

3. Verify installation:

```bash

git --version

```

4. Exit the container shell:

```bash

exit

```

Alternative one-liner:

```bash

docker exec jenkins apt-get update && docker exec jenkins apt-get install -y git

```

Note: The container must be based on a Debian/Ubuntu image for apt-get to work. For Alpine Linux, use `apk add git`.

## **Q11: How can I list the images, running, stopped and paused containers using single command?**

There isn't a single native Docker command that shows all these at once, but you can combine commands:

For a comprehensive overview:

```bash

echo "IMAGES:" && docker images && echo -e "\nCONTAINERS (all):" && docker ps -a

```

For just the counts:

```bash

echo "Images: $(docker images -q | wc -l)" && \

echo "Running: $(docker ps -q | wc -l)" && \

echo "Stopped: $(docker ps -f status=exited -q | wc -l)" && \

echo "Paused: $(docker ps -f status=paused -q | wc -l)"

```

For a custom format showing all containers with their status:

```bash

docker ps -a --format "table {{.ID}}\t{{.Names}}\t{{.Status}}\t{{.Image}}"

```

## **Q12: How can I troubleshoot a container when it is not running?**

Several approaches to troubleshoot a non-running container:

1. Check container logs:

```bash

docker logs <container\_name\_or\_id>

```

2. View the last few lines of logs:

```bash

docker logs --tail 50 <container\_name\_or\_id>

```

3. Inspect container details:

```bash

docker inspect <container\_name\_or\_id>

```

4. Check the exit code (may indicate why it stopped):

```bash

docker ps -a --filter "name=<container\_name>"

```

5. Run the container interactively to see errors:

```bash

docker run -it <image\_name> /bin/sh

```

6. Check events for system-wide issues:

```bash

docker events

```

7. Check container resource usage before it crashed:

```bash

docker stats <container\_name\_or\_id>

```

8. If the container exits immediately, try:

```bash

docker run --entrypoint /bin/sh -it <image\_name>

```

Then try running the normal command manually to see errors.

## **Q13: What is persistent volume in Docker?**

A \*\*persistent volume\*\* in Docker is a mechanism to store data outside a container's writable layer, ensuring data persists even when the container is removed.

Key points:

- By default, container filesystems are ephemeral - changes are lost when container stops

- Volumes provide persistent storage independent of container lifecycle

- Volumes can be shared among multiple containers

- Better performance than writing to container's writable layer

- Can be backed up or migrated

- Managed through Docker's volume system

Example creating and using a volume:

```bash

docker volume create my\_volume

docker run -v my\_volume:/path/in/container my\_image

```

Types of persistence in Docker:

1. \*\*Volumes\*\*: Managed by Docker (/var/lib/docker/volumes/)

2. \*\*Bind mounts\*\*: Mount host directory directly into container

3. \*\*tmpfs mounts\*\*: Store in host memory (not persistent)

## **Q14: What are different volumes in Docker?**

Docker provides several volume types for data persistence:

1. \*\*Named Volumes\*\*:

- Created and managed by Docker

- Stored in Docker's storage directory (/var/lib/docker/volumes/)

- Best for production (managed by Docker)

```bash

docker volume create my\_volume

docker run -v my\_volume:/container/path image

```

2. \*\*Anonymous Volumes\*\*:

- Created automatically when container starts with -v /container/path

- Not named, harder to reference

- Removed with `docker volume prune`

3. \*\*Bind Mounts\*\*:

- Map a host directory directly into container

- Can modify host filesystem directly

- Good for development

```bash

docker run -v /host/path:/container/path image

```

4. \*\*tmpfs Mounts\*\*:

- Stores data in host memory only

- Never written to host filesystem

- Disappears when container stops

```bash

docker run --tmpfs /container/path image

```

5. \*\*Volume Drivers\*\*:

- Plugins that enable volumes on remote hosts/cloud providers

- Examples: Azure File Storage, NFS, AWS EBS

## **Q15: What is the use of docker-compose. Can you explain about docker-compose.yml file.**

\*\*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.

\*\*docker-compose.yml\*\* file:

- Defines services, networks, and volumes in a declarative way

- Allows you to manage multi-container apps with a single command

- Enables service dependencies and startup order

- Simplifies complex docker run commands

Key features:

1. \*\*Services\*\*: Define each container as a service with its configuration

2. \*\*Networks\*\*: Create custom networks for services to communicate

3. \*\*Volumes\*\*: Configure persistent storage

4. \*\*Environment variables\*\*: Set runtime environment

5. \*\*Dependencies\*\*: Control startup order

Example docker-compose.yml:

```yaml

version: '3.8'

services:

web:

image: nginx:alpine

ports:

- "80:80"

volumes:

- ./html:/usr/share/nginx/html

depends\_on:

- db

db:

image: postgres:13

environment:

POSTGRES\_PASSWORD: example

volumes:

- db\_data:/var/lib/postgresql/data

volumes:

db\_data:

```

Common commands:

```bash

docker-compose up -d # Start services in background

docker-compose down # Stop and remove containers

docker-compose ps # List running services

docker-compose logs # View logs

```

## **Q16: How can we use docker commands**

Here are explanations of common Docker commands:

1. \*\*run\*\*: Create and start a new container

```bash

docker run [options] image [command]

```

2. \*\*exec\*\*: Run a command in a running container

```bash

docker exec [options] container command

```

3. \*\*ps\*\*: List containers

```bash

docker ps [options] # -a for all containers

```

4. \*\*build\*\*: Build an image from a Dockerfile

```bash

docker build [options] path

```

5. \*\*pull\*\*: Download an image from a registry

```bash

docker pull image[:tag]

```

6. \*\*push\*\*: Upload an image to a registry

```bash

docker push image[:tag]

```

7. \*\*images\*\*: List images

```bash

docker images [options]

```

8. \*\*login\*\*: Log in to a registry

```bash

docker login [options] [server]

```

9. \*\*logout\*\*: Log out from a registry

```bash

docker logout [server]

```

10. \*\*search\*\*: Search Docker Hub for images

```bash

docker search term

```

11. \*\*version\*\*: Show Docker version information

```bash

docker version [options]

```

12. \*\*info\*\*: Display system-wide information

```bash

docker info

```

## **Q17: Write down a command to connect and disconnect a container to a host network**

\*\*Connect a container to a host network:\*\*

```bash

docker network connect host network\_name container\_name

```

Example:

```bash

docker network connect host my\_container

```

\*\*Disconnect a container from a network:\*\*

```bash

docker network disconnect network\_name container\_name

```

Example:

```bash

docker network disconnect host my\_container

```

Note: The "host" network is a special network that shares the host's networking namespace. For custom networks, replace "host" with your network name.

## **Q18: What is the default network in Docker. What is inspect command used for?**

\*\*Default network in Docker:\*\*

Docker creates three default networks when installed:

1. \*\*bridge\*\*: The default network containers connect to (docker0)

- Containers get IP addresses in the 172.17.0.0/16 range

- Containers can communicate via IP

- Requires port publishing (-p) to access from host

2. \*\*host\*\*: Shares host's network namespace (no network isolation)

3. \*\*none\*\*: No networking (container has no network interfaces)

\*\*docker inspect command:\*\*

- Provides detailed information about Docker objects (containers, images, networks, volumes, etc.)

- Returns JSON format data with all configuration details

- Useful for troubleshooting and viewing settings

Examples:

```bash

docker inspect container\_name # Inspect container

docker inspect image\_name # Inspect image

docker inspect network\_name # Inspect network

docker inspect volume\_name # Inspect volume

```

To get specific information, use format flags:

```bash

docker inspect -f '{{.NetworkSettings.IPAddress}}' container\_name

```

## **Q19: What is the none network in Docker. How can we disconnect from bridge network and connect to none network?**

\*\*none network in Docker:\*\*

- A special network that provides complete network isolation

- Container has no network interfaces (except loopback)

- Useful for:

- Security-sensitive applications

- Containers that only need local filesystem access

- Testing network-independent functionality

\*\*Steps to disconnect from bridge and connect to none:\*\*

1. Create a container (if not already created):

```bash

docker run -d --name my\_container nginx

```

2. Disconnect from bridge network:

```bash

docker network disconnect bridge my\_container

```

3. Connect to none network:

```bash

docker network connect none my\_container

```

Alternatively, start a container directly with none network:

```bash

docker run -d --network none --name isolated\_container nginx

```

Verify with:

```bash

docker exec isolated\_container ip a

```

(Should only show loopback interface)

## **Q20: Default location of Docker, images and containers**

Default storage locations on Linux systems:

1. \*\*Docker root directory\*\*:

```

/var/lib/docker/

```

2. \*\*Images\*\*:

```

/var/lib/docker/image/

```

3. \*\*Containers\*\*:

```

/var/lib/docker/containers/

```

4. \*\*Volumes\*\*:

```

/var/lib/docker/volumes/

```

5. \*\*Networks\*\*:

Network configuration files are stored in:

```

/var/lib/docker/network/

```

On Windows (Docker Desktop):

```

C:\ProgramData\DockerDesktop\

```

On macOS (Docker Desktop):

```

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

```

To check your Docker root directory:

```bash

docker info | grep "Docker Root Dir"

```

Note: These locations can be changed by configuring the Docker daemon with the `--data-root` option.

## **Q21: What is a Docker image?**

A \*\*Docker image\*\* is a read-only template used to create Docker containers. It's a packaged application with all its dependencies and configuration.

Key characteristics:

- Composed of layered filesystem changes

- Each layer represents an instruction in the Dockerfile

- Images are immutable (cannot be changed after creation)

- Shared layers optimize storage and transfer

- Stored in a registry (Docker Hub, private registry, etc.)

Image components:

1. \*\*Base Image\*\*: The starting point (e.g., ubuntu, alpine)

2. \*\*Layers\*\*: Filesystem changes from each Dockerfile instruction

3. \*\*Metadata\*\*: Configuration for the container (env vars, commands, etc.)

Common operations:

- Build: `docker build -t my\_image .`

- Pull: `docker pull nginx`

- Push: `docker push myrepo/my\_image`

- List: `docker images`

- Remove: `docker rmi my\_image`

## **Q22: Why do we use following in Dockerfile**

Dockerfile instructions and their purposes:

1. \*\*FROM\*\*:

- Specifies the base/parent image to build upon

- Must be the first non-comment instruction

- Example: `FROM ubuntu:20.04`

2. \*\*LABEL\*\*:

- Adds metadata to the image (key-value pairs)

- Used for organization, documentation, licensing

- Example: `LABEL maintainer="me@example.com"`

3. \*\*WORKDIR\*\*:

- Sets the working directory for subsequent instructions

- Creates directory if it doesn't exist

- Example: `WORKDIR /app`

4. \*\*USER\*\*:

- Sets the username or UID for subsequent instructions

- Affects RUN, CMD, and ENTRYPOINT instructions

- Example: `USER node`

5. \*\*COPY\*\*:

- Copies files/directories from host to container

- Example: `COPY . /app`

6. \*\*CMD\*\*:

- Provides default command/arguments for container

- Only one CMD per Dockerfile (last one wins)

- Example: `CMD ["npm", "start"]`

7. \*\*RUN\*\*:

- Executes commands during image build

- Each RUN creates a new layer

- Example: `RUN apt-get update && apt-get install -y curl`

## **Q23: I want to delete the unused networks and volumes**

To clean up unused Docker objects:

1. \*\*Delete unused networks\*\*:

```bash

docker network prune

```

Add `-f` to skip confirmation

2. \*\*Delete unused volumes\*\*:

```bash

docker volume prune

```

3. \*\*Delete all unused objects (networks, volumes, containers, images)\*\*:

```bash

docker system prune

```

For more aggressive cleanup:

```bash

docker system prune -a

```

To see what would be removed without actually deleting:

```bash

docker network prune --dry-run

docker volume prune --dry-run

docker system prune --dry-run

```

## **Q24: How can I check the resource usage of running containers?**

Several ways to check container resource usage:

1. \*\*docker stats\*\* (real-time monitoring):

```bash

docker stats

```

Shows CPU%, memory usage/limit, memory %, network I/O, block I/O, PIDs

2. For specific containers:

```bash

docker stats container1 container2

```

3. \*\*docker top\*\* (view running processes):

```bash

docker top container\_name

```

4. \*\*docker inspect\*\* (view configuration and resource limits):

```bash

docker inspect container\_name

```

Look for "HostConfig" section (Memory, CPU, etc.)

5. \*\*cAdvisor\*\* (Google's container monitoring tool):

```bash

docker run -d --name=cadvisor -p 8080:8080 --volume=/:/rootfs:ro --volume=/var/run:/var/run:rw --volume=/sys:/sys:ro --volume=/var/lib/docker/:/var/lib/docker:ro google/cadvisor

```

Then access http://localhost:8080

## **Q25: What is ENV in a Dockerfile and how is it used?**

\*\*ENV\*\* in a Dockerfile sets environment variables that will be available:

- During the build process (for subsequent instructions)

- In containers created from the image

Usage:

1. Simple form (single variable):

```dockerfile

ENV KEY value

```

Example:

```dockerfile

ENV APP\_HOME /usr/src/app

```

2. Multiple variables:

```dockerfile

ENV KEY1=value1 KEY2=value2

```

Example:

```dockerfile

ENV NODE\_ENV=production PORT=3000

```

Key points:

- Variables can be referenced in subsequent instructions:

```dockerfile

WORKDIR $APP\_HOME

```

- Values persist in running containers

- Can be overridden at runtime with `-e` flag:

```bash

docker run -e "NODE\_ENV=development" my\_image

```

Difference between ENV and ARG:

- ENV persists in container, ARG is only during build

- ARG can be set from build command (`--build-arg`)

- ENV can reference ARG values

## **Q26: Write a Dockerfile for Node.js and Java application**

Here's a multi-stage Dockerfile for an application that needs both Node.js (for frontend) and Java (for backend):

```dockerfile

# Stage 1: Build frontend with Node.js

FROM node:16 as frontend-builder

WORKDIR /app

COPY package.json package-lock.json ./

RUN npm ci

COPY . .

RUN npm run build

# Stage 2: Build backend with Java

FROM maven:3.8.4-openjdk-17 as backend-builder

WORKDIR /app

COPY pom.xml .

RUN mvn dependency:go-offline

COPY src ./src

RUN mvn package -DskipTests

# Stage 3: Final runtime image

FROM openjdk:17-jdk-slim

WORKDIR /app

# Copy built backend

COPY --from=backend-builder /app/target/myapp.jar ./app.jar

# Copy built frontend

COPY --from=frontend-builder /app/dist ./public

# Set environment variables

ENV PORT=8080

ENV SPRING\_PROFILES\_ACTIVE=prod

# Expose port

EXPOSE $PORT

# Run application

CMD ["java", "-jar", "app.jar"]

```

Key features:

1. Multi-stage build to keep final image small

2. Node.js stage for frontend build

3. Maven stage for Java backend build

4. Slim JDK image for runtime

5. Combined frontend (static files) and backend (JAR) in final image

6. Environment variables for configuration

7. Proper layer caching for dependencies

## **Q27: What is Dockerfile optimization and how do you implement it?**

\*\*Dockerfile optimization\*\* involves techniques to:

- Reduce image size

- Speed up build process

- Improve security

- Enhance maintainability

Key optimization techniques:

1. \*\*Multi-stage builds\*\*:

- Separate build environment from runtime

- Copy only necessary artifacts to final image

```dockerfile

FROM node:16 as builder

# build steps...

FROM node:16-alpine

COPY --from=builder /app/dist /app

```

2. \*\*Minimize layers\*\*:

- Combine related RUN commands

- Clean up in same layer

```dockerfile

RUN apt-get update && \

apt-get install -y package && \

rm -rf /var/lib/apt/lists/\*

```

3. \*\*Use .dockerignore\*\*:

- Exclude unnecessary files from build context

- Similar to .gitignore

4. \*\*Choose appropriate base image\*\*:

- Prefer Alpine or slim versions

- Example: `python:3.9-alpine` vs `python:3.9`

5. \*\*Leverage build cache\*\*:

- Order instructions from least to most frequently changed

- Copy package files (package.json, requirements.txt) before source code

6. \*\*Avoid unnecessary files\*\*:

- Remove temporary files in same RUN command

- Don't include development tools in production images

7. \*\*Use specific tags\*\*:

- Avoid `latest` tag which can cause inconsistencies

- Example: `node:16.14.2-alpine`

8. \*\*Set proper permissions\*\*:

- Create non-root user

- Restrict file permissions

```dockerfile

RUN adduser -D myuser && chown -R myuser /app

USER myuser

```

## **Q28: Forcefully delete all the running containers and images**

\*\*Warning\*\*: These commands will remove ALL containers and images without confirmation.

1. \*\*Stop and remove all running containers\*\*:

```bash

docker rm -f $(docker ps -aq)

```

or with newer Docker versions:

```bash

docker container rm -f $(docker container ls -aq)

```

2. \*\*Remove all images\*\*:

```bash

docker rmi -f $(docker images -aq)

```

or:

```bash

docker image rm -f $(docker image ls -aq)

```

3. \*\*Remove unused networks, volumes, and build cache\*\*:

```bash

docker system prune -a --volumes -f

```

Alternative one-liner for everything:

```bash

docker system prune -a --volumes -f && docker rm -f $(docker ps -aq) && docker rmi -f $(docker images -aq)

```

\*\*Important notes\*\*:

- These commands are destructive and cannot be undone

- Back up any important data in containers or volumes first

- Consider removing specific containers/images instead if possible

- Add `-f` to force removal even if containers are running