Docker – Complete Learning Index (Interview-Oriented) - Explained

1. Introduction to Docker

1.1 What is Docker

Docker is a platform that uses containerization to package applications and their dependencies into standardized units called containers. Think of it like a shipping container for software - it ensures your application runs consistently anywhere, regardless of the environment.

Real Example: Imagine you're developing a Python web app that works perfectly on your Mac but fails on your colleague's Windows machine due to different Python versions or missing libraries. Docker solves this by packaging your app with its specific Python version, libraries, and configurations into a container that runs identically on any system with Docker installed.

1.2 Why Docker was created (Problems it solves)

Before Docker, developers faced:

"It works on my machine" problem - Applications behaving differently across environments

Complex dependency management - Installing and configuring multiple versions of libraries

Resource inefficiency - VMs requiring full OS copies for each application

Slow deployment - Lengthy setup and configuration processes

Real Example: A team building a microservices architecture might need 10 different services, each with unique dependencies. Without Docker, they'd need 10 separate VM setups or complex dependency management. With Docker, each service gets its own container with exactly what it needs.

1.3 Docker vs Virtual Machines

VM Approach:

Hardware → Hypervisor → Multiple Guest OS → Applications

Each VM includes full OS (Linux/Windows kernel, libraries, etc.)

Heavy (GBs), slow to start, significant overhead

Docker Approach:

Hardware → Host OS → Docker Engine → Multiple Containers

Containers share host OS kernel

Light (MBs), fast startup, minimal overhead

Real Example: Running a simple Redis database:

VM approach: Need to install full Ubuntu + Redis = 1-2GB, takes 1-2 minutes to boot

Docker approach: docker run redis = 100MB, starts in seconds

1.4 Real-world use cases of Docker

Development Environments: Every developer gets identical setup

CI/CD Pipelines: Consistent testing environments

Microservices: Each service in its own container

Legacy App Migration: Containerize old apps for modern infrastructure

Batch Processing: Run data processing jobs in isolated containers

Real Example: Netflix uses Docker to run thousands of microservices. Each service (user authentication, recommendation engine, video streaming) runs in its own container, allowing independent scaling and updates.

1.5 Docker architecture overview

Docker follows a client-server architecture:

Docker Client: CLI/UI you interact with (docker run, docker build)

Docker Daemon (dockerd): Background service managing containers

Docker Registry: Storage for Docker images (Docker Hub, private registries)

Docker Objects: Images, containers, networks, volumes

Real Example: When you run docker run nginx:

Client sends command to Docker Daemon

Daemon checks if nginx image exists locally

If not, pulls from Docker Hub registry

Creates container from image

Starts the container

2. Core Docker Concepts

2.1 Images

A Docker image is a read-only template with instructions for creating a container. It's built in layers using a Dockerfile.

Real Example: The official nginx:alpine image contains:

Base layer: Alpine Linux (minimal Linux distribution)

Add layer: Nginx installation

Add layer: Configuration files

Add layer: Startup script

Each layer is immutable. When you modify an image, Docker adds new layers on top.

2.2 Containers

A container is a running instance of an image. It's an isolated, lightweight process with its own filesystem, networking, and isolated process tree.

Real Example:

```
bash
# Create and start a container from nginx image
docker run -d -p 8080:80 --name my-web nginx
```

This creates a container named my-web that:

Runs Nginx web server

Maps host port 8080 to container port 80

Runs in background (-d flag)

2.3 Docker Engine

The Docker Engine is the core component that runs and manages containers. It consists of:

Docker Daemon (dockerd): Server that manages Docker objects

REST API: Interface for programs to talk to daemon

Docker CLI: Command-line interface

Real Example: When Jenkins needs to build a Docker image as part of CI/CD, it uses Docker Engine's REST API to communicate with the Docker Daemon.

2.4 Docker Client

The Docker Client is what users interact with. It can be:

Docker CLI (docker command)

Docker Desktop (GUI)

Third-party tools (Portainer, Rancher)

Real Example: Developers use Docker CLI (docker build, docker push) while operations teams might use Portainer GUI for container management.

2.5 Docker Daemon

The Docker Daemon (dockerd) is a background service that:

Listens for Docker API requests

Manages Docker objects (images, containers, networks, volumes)

Can communicate with other daemons to manage Docker services

Real Example: When you have 20 containers running, the Docker Daemon monitors them, restarts failed ones (if configured), and manages resource allocation.

2.6 Docker Registry

A Docker Registry stores Docker images. It's like GitHub for Docker images.

Real Example:

Docker Hub: Public registry (like docker pull nginx)

Amazon ECR: AWS private registry

Google Container Registry: GCP private registry

Self-hosted: Run your own registry with Docker Registry image

2.7 Docker Hub

Docker Hub is the default public registry for Docker images. It contains:

Official Images: Verified images from software vendors (nginx, mysql, node)

Verified Publisher Images: From certified companies

Community Images: User-created images

Real Example: A company might use:

Official postgres:13 image for production database

Their custom-built myapp:latest image from their private registry

Community jc21/nginx-proxy-manager for reverse proxy

3. Docker Installation & Setup

3.1 Installing Docker on Windows

Windows has two approaches:

Docker Desktop (Recommended):

Requires Windows 10/11 Pro, Enterprise, or Education (for WSL 2 backend)

Uses WSL 2 (Windows Subsystem for Linux) or Hyper-V

Includes Docker CLI, Docker Compose, Kubernetes

Docker Toolbox (Legacy):

For older Windows versions or Home edition

Uses VirtualBox to run Linux VM

Real Example: A Windows developer installs Docker Desktop:

Downloads from docker.com

Enables WSL 2 feature in Windows

Installs Docker Desktop with WSL 2 backend

Can now run Linux containers natively through WSL 2

3.2 Installing Docker on Linux

Ubuntu Example:

bash

```
# Update package index
sudo apt-get update

# Install prerequisites
sudo apt-get install apt-transport-https ca-certificates curl software-properties-common

# Add Docker's GPG key
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

# Add Docker repository
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"

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

```
# Add user to docker group (to run without sudo)
sudo usermod -aG docker $USER
```

Real Example: A cloud server (AWS EC2 Ubuntu instance) needs Docker for running containers. The admin runs these commands to install Docker Engine.

3.3 Installing Docker on Mac

Docker Desktop for Mac:

Uses a lightweight Linux VM (hyperkit) to run containers

Native macOS integration

Includes Docker CLI, Docker Compose, Kubernetes

Real Example: A macOS developer:

Downloads Docker.dmg from docker.com

Drags Docker to Applications folder

Runs Docker Desktop

Can now use docker commands in Terminal

3.4 Verifying Docker Installation

After installation, verify everything works:

bash

```
# Check Docker version
docker --version
# Output: Docker version 20.10.7, build f0df350

# Run test container
docker run hello-world

# Check Docker info
docker info

# List running containers
docker ps
```

Real Example: After installing Docker, you run docker run hello-world. This pulls a small test image and runs it to confirm Docker is working correctly.

3.5 Docker Desktop Overview

Docker Desktop includes:

Docker Engine: For running containers

Docker CLI: Command-line interface

Docker Compose: For multi-container apps

Kubernetes: Local Kubernetes cluster

Dashboard: GUI for managing containers

Extensions: Additional tools

Real Example: A developer uses Docker Desktop to:

Start/stop containers via GUI

View container logs and stats

Manage images and volumes

Switch between Linux and Windows containers

Enable Kubernetes for local development

(I'll continue with the remaining sections in subsequent responses to maintain readability. Would you like me to proceed with Section 4: Docker Basic Commands next?)

yes

4. Docker Basic Commands (Must for Interviews)

4.1 docker --version

Checks Docker installation version and confirms CLI is working.

Real Example:

bash

```
$ docker --version
Docker version 24.0.7, build afdd53b
```

Interview Insight: Shows Docker client version. Useful for troubleshooting compatibility issues. Always mention that server (daemon) version might differ (check with docker info).

4.2 docker pull

Downloads an image from a registry (default: Docker Hub) to local system.

Real Example:

bash

```
# Pull latest nginx image
docker pull nginx

# Pull specific version
docker pull nginx:1.21-alpine

# Pull from private registry
docker pull myregistry.com/myapp:v2.0
```

Real Scenario: Before deploying a new version in production, you pull it first to test locally:

bash

```
docker pull mycompany/api-service:v1.5

docker run mycompany/api-service:v1.5
```

4.3 docker images

Lists all Docker images stored locally.

Real Example:

bash

```
$ docker images
REPOSITORY   TAG      IMAGE ID      CREATED        SIZE
nginx       latest   6efc10a0510f  2 weeks ago    142MB
redis       alpine   2e2f252f3c88  3 weeks ago    32.3MB
ubuntu      20.04    d5447fc01ae6  2 months ago   72.8MB
```

Useful flags:

bash

```
# Show all images (including intermediate)
docker images -a

# Show only image IDs
docker images -q

# Filter images
docker images --filter "dangling=true"
```

4.4 docker run

The most important command. Creates and starts a container from an image.

Real Examples:

bash

```
# Basic run
docker run nginx

# Run in detached mode (background)
docker run -d --name web nginx

# Map ports (host:container)
docker run -d -p 8080:80 nginx

# Mount volume
docker run -d -v /data:/app/data mysql

# Set environment variables
docker run -e "DATABASE_URL=localhost" -e "DEBUG=true" myapp

# Interactive container with shell
docker run -it ubuntu bash

# Limit resources
docker run --memory="512m" --cpus="1.5" myapp
```

Interview Scenario: "Run a PostgreSQL container with persistence and custom password"

bash

```
docker run -d \
  --name postgres-db \
  -e POSTGRES_PASSWORD=mysecretpassword \
  -v pgdata:/var/lib/postgresql/data \
  -p 5432:5432 \
  postgres:13
```

4.5 docker ps

Lists running containers.

Real Example:

bash

```
$ docker ps
CONTAINER ID   IMAGE    COMMAND            CREATED       STATUS       PORTS            NAMES
a1b2c3d4e5f6   nginx    "/docker-entrypoint.…"   2 hours ago   Up 2 hours   0.0.0.0:8080->80/tcp   webserver
```

Common flags:

bash

```
# Show all containers (running + stopped)
docker ps -a

# Show last created container
docker ps -l
```

```
# Show only container IDs
docker ps -q

# Format output
docker ps --format "table {{.ID}}\t{{.Names}}\t{{.Status}}"
```

4.6 docker ps -a

Shows all containers including stopped ones.

Real Example:

bash

```
$ docker ps -a
CONTAINER ID   IMAGE     COMMAND       CREATED        STATUS                    PORTS     NAMES
a1b2c3d4e5f6   nginx     "nginx -g …"  2 hours ago    Up 2 hours                80/tcp    webserver
b2c3d4e5f6g7   redis     "redis-ser…"  5 hours ago    Exited (0) 2 hours ago              cache
c3d4e5f6g7h8   ubuntu    "bash"        1 day ago      Exited (137) 1 day ago              test-container
```

Interview Tip: Useful for debugging why containers stopped (check Exit codes).

4.7 docker stop

Gracefully stops a running container (sends SIGTERM, then SIGKILL after timeout).

Real Example:

bash

```
# Stop by container name
docker stop webserver
```

```
# Stop by container ID
docker stop a1b2c3d4e5f6

# Stop multiple containers
docker stop container1 container2

# Stop with timeout (default 10s)
docker stop -t 30 webserver  # Wait 30 seconds before force kill
```

Important: Always prefer docker stop over docker kill unless container is unresponsive.

4.8 docker start

Starts a stopped container (preserves its configuration and data).

Real Example:

bash

```
# Start stopped container
docker start webserver

# Start and attach to output
docker start -a webserver

# Start in interactive mode
docker start -i mycontainer
```

Interview Scenario: After fixing a configuration issue in a stopped container:

bash

```
# Container was stopped due to config error
$ docker ps -a | grep exited
b2c3d4e5f6g7   myapp   "node app.js"   1 hour ago   Exited (1)

# Fix the config, then restart
docker start b2c3d4e5f6g7
```

4.9 docker restart

Restarts a container (stop + start).

Real Example:

bash

```
# Restart by name
docker restart webserver

# Restart with new configuration
docker update --memory="1g" webserver && docker restart webserver
```

Use Case: Applying new environment variables:

bash

```
docker run -d -e "MODE=dev" --name app myapp
# Later need to change to production
docker stop app
docker rm app
docker run -d -e "MODE=prod" --name app myapp
# OR using restart with update (Docker 1.13+)
docker update --env-add "MODE=prod" app
docker restart app
```

4.10 docker rm

Removes stopped containers.

Real Example:

bash

```
# Remove by name
docker rm old-container

# Remove by ID
docker rm a1b2c3d4e5f6

# Remove multiple
docker rm container1 container2 container3

# Force remove running container
docker rm -f running-container

# Remove all stopped containers
docker container prune
# OR
docker rm $(docker ps -aq)
```

Interview Question: "Difference between docker rm and docker rmi?"

docker rm → removes containers

docker rmi → removes images

4.11 docker rmi

Removes Docker images.

Real Example:

bash

```
# Remove image by ID
docker rmi 6efc10a0510f

# Remove image by tag
docker rmi nginx:latest

# Force remove (even if in use)
docker rmi -f myimage

# Remove dangling images (untagged)
docker image prune

# Remove all unused images
docker image prune -a
```

Common Patterns:

bash

```
# Cleanup workflow
docker stop $(docker ps -aq)
docker rm $(docker ps -aq)
docker rmi $(docker images -q)

# Remove specific version
docker rmi myapp:v1.0

# Remove by pattern
docker images | grep "none" | awk '{print $3}' | xargs docker rmi
```

Interview Scenario: "You're getting 'no space left on device' error. How to clean up?"

bash

```
# Check disk usage
docker system df

# Remove unused data
docker system prune

# Remove everything (more aggressive)
docker system prune -a

# Specific cleanup
```

```
docker container prune   # Remove stopped containers
docker image prune      # Remove dangling images
docker volume prune     # Remove unused volumes
docker network prune    # Remove unused networks
```

Quick Interview Cheat Sheet for Commands:

Command	Purpose	Common Flags
docker run	Create+start container	-d, -p, -v, -e, --name
docker ps	List running containers	-a (all), -q (quiet)
docker stop	Stop container gracefully	-t (timeout)
docker start	Start stopped container	-a (attach), -i (interactive)
docker rm	Remove container	-f (force)
docker rmi	Remove image	-f (force)
docker exec	Run command in running container	-it (interactive)
docker logs	View container logs	-f (follow), --tail
docker build	Build image from Dockerfile	-t (tag)

Pro Tip for Interviews: Always mention:

Container lifecycle: run → stop → start/restart → rm

Image vs Container: Image is template, Container is instance

Use docker --help for command details

(Would you like me to continue with Section 5: Docker Image Deep Dive next?)

give the answer for all Section dont ask

5. Docker Image Deep Dive

5.1 What is a Docker Image

A Docker image is a read-only template containing application code, dependencies, libraries, and configuration files. Images are built in layers using a Dockerfile.

Real Example: Building a Python web app image:

dockerfile

```
FROM python:3.9-slim

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY app.py .

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

When built: `docker build -t myapp:v1 .`

Creates immutable layers for each instruction

Result is a portable image that runs anywhere

5.2 Image Layers & Union File System

Docker uses a Union File System (Overlay2) to combine multiple read-only layers into a single filesystem.

Real Example: Analyzing an nginx image:

bash

```
# Inspect image layers
docker history nginx:alpine

# Output shows:
# Layer 1: Alpine base OS (5MB)
# Layer 2: Nginx installation files (3MB)
# Layer 3: Configuration files (few KB)
# Layer 4: Entrypoint script (few bytes)
```

Key Concept: When you run docker run, Docker adds a thin writable layer on top of read-only layers for runtime changes.

5.3 Base Images

Base images are the foundation layer, usually a minimal OS.

Common Base Images:

dockerfile

```
# Ubuntu (full-featured)
FROM ubuntu:20.04

# Alpine (lightweight, security-focused)
FROM alpine:3.14

# Distroless (minimal, no shell)
FROM gcr.io/distroless/base
```

```
# Scratch (empty)
FROM scratch
```

Real Example: Choosing base images:

Production: alpine (smaller, fewer vulnerabilities)

Development: ubuntu (easier debugging with tools)

Go apps: scratch (extremely small)

5.4 Creating Custom Images

Build custom images using docker build:

Real Example - Node.js app:

dockerfile

```
FROM node:16-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
COPY src/ ./src/
EXPOSE 3000
USER node
CMD ["node", "src/index.js"]
```

Build: docker build -t mynodeapp:v2 .

5.5 Tagging & Versioning Images

Tagging provides identity and versioning to images.

Real Examples:

bash

```
# Tag during build
docker build -t myapp:1.0 -t myapp:latest .

# Tag existing image
docker tag myapp:1.0 registry.company.com/myapp:prod

# Semantic versioning pattern
docker tag app myapp:1.2.3
docker tag app myapp:1.2
docker tag app myapp:1
docker tag app myapp:latest

# Push tagged images
docker push registry.company.com/myapp:1.2.3
```

6. Dockerfile (Most Important for Interviews)

6.1 What is a Dockerfile

A Dockerfile is a text file with instructions to build a Docker image.

Real Example: Complete Dockerfile:

dockerfile

```dockerfile
# Start from base image
FROM python:3.9-slim

# Set maintainer (deprecated but good practice)
LABEL maintainer="dev@company.com"

# Set working directory
WORKDIR /app

# Copy dependency file
COPY requirements.txt .

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

# Copy application code
COPY . .

# Expose port
EXPOSE 8000

# Set environment variable
ENV PYTHONUNBUFFERED=1
```

```
# Health check
HEALTHCHECK --interval=30s CMD curl -f http://localhost:8000/health || exit 1


# Define entry point
CMD ["gunicorn", "--bind", "0.0.0.0:8000", "app:app"]
```

6.2 Common Dockerfile Instructions

FROM

Sets the base image.

dockerfile

```
FROM ubuntu:20.04
FROM python:3.9 AS builder  # Multi-stage build
```

RUN

Executes commands during image build.

dockerfile

```
RUN apt-get update && apt-get install -y \
    curl \
    nginx \
    && rm -rf /var/lib/apt/lists/*
```

COPY vs ADD

COPY - Copies files/directories from host to image

dockerfile

```
COPY . /app
COPY config.json /etc/app/
```

ADD - Can also handle URLs and auto-extract archives

dockerfile

```
ADD https://example.com/file.tar.gz /tmp/
ADD app.tar.gz /app/  # Auto-extracts
```

Interview Tip: Prefer COPY unless you need ADD's extra features.

WORKDIR

Sets working directory for subsequent instructions.

dockerfile

```
WORKDIR /app
RUN pwd  # Output: /app
```

EXPOSE

Documents which ports the container listens on.

dockerfile

```
EXPOSE 80   # HTTP
EXPOSE 443  # HTTPS
```

CMD

Provides default command when container starts.

dockerfile

```
CMD ["npm", "start"]
CMD ["python", "app.py"]
```

ENTRYPOINT

Sets the main command (cannot be overridden by docker run).

dockerfile

```
ENTRYPOINT ["python"]
CMD ["app.py"]
# docker run myapp → runs: python app.py
# docker run myapp test.py → runs: python test.py
```

ENV

Sets environment variables.

dockerfile

```
ENV NODE_ENV=production
ENV PORT=3000
```

6.3 Difference between CMD vs ENTRYPOINT

Aspect	CMD	ENTRYPOINT
Purpose	Default arguments for ENTRYPOINT	Main executable
Override	Can be overridden by docker run	Cannot be overridden (without --entrypoint)
Shell form	CMD npm start (runs in shell)	ENTRYPOINT ["executable"] (exec form)
Best for	Default parameters	Container as executable

Real Example:

dockerfile

```
# Use case 1: ENTRYPOINT as main app
FROM alpine:3.14
ENTRYPOINT ["ping"]
```

```
CMD ["localhost"]
# docker run myping → ping localhost
# docker run myping google.com → ping google.com

# Use case 2: CMD as default command
FROM node:16
CMD ["npm", "start"]
# docker run myapp → npm start
# docker run myapp bash → overrides to bash
```

6.4 Best Practices for Writing Dockerfiles

Use specific tags, not latest

dockerfile

```
FROM ubuntu:20.04  # Good
FROM ubuntu:latest  # Bad
```

Combine RUN commands to reduce layers

dockerfile

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

Use .dockerignore

text

```
node_modules/
```

```
.git/
*.log
Dockerfile
.env
```

Minimize layer count and image size

Run as non-root user

dockerfile

```
RUN groupadd -r appuser && useradd -r -g appuser appuser
USER appuser
```

Use multi-stage builds for production

dockerfile

```
# Build stage
FROM node:16 AS builder
WORKDIR /build
COPY . .
RUN npm run build

# Production stage
FROM nginx:alpine
COPY --from=builder /build/dist /usr/share/nginx/html
```

7. Docker Containers Deep Dive

7.1 What is a Container

A container is a running instance of an image with:

Isolated filesystem

Isolated process space

Isolated network interface

Resource limits (CPU, memory)

Real Example:

bash

```
# Create container
docker run -d --name myapp --memory="512m" myimage

# Inspect container details
docker inspect myapp
```

7.2 Container Lifecycle

text

```
Created → Running → Paused → Stopped → Removed
   ↘       ↗
    └──── Restart ┘
```

Commands for each state:

bash

```
docker create    # Created state
docker start     # Created → Running
docker run       # Created + Started
docker pause     # Running → Paused
docker unpause   # Paused → Running
docker stop      # Running → Stopped (graceful)
docker kill      # Running → Stopped (force)
docker restart   # Running → Running
docker rm        # Stopped → Removed
```

7.3 Running Containers in Foreground/Background

Foreground Mode (default):

bash

```
docker run nginx  # Blocks terminal
# Ctrl+C stops container
```

Background Mode (detached):

bash

```
docker run -d --name web nginx
# Returns container ID, runs in background

# Attach to running container
docker attach web  # Re-attach
docker exec -it web bash  # New session
```

Real Use Case:

bash

```
# Development - foreground for logs
docker run -p 3000:3000 node-app

# Production - background
docker run -d -p 80:80 --restart unless-stopped nginx
```

7.4 Port Mapping

Maps container ports to host ports.

Real Examples:

bash

```
# Map specific host port to container port
docker run -p 8080:80 nginx  # host:container

# Map to all interfaces
docker run -p 0.0.0.0:8080:80 nginx

# Random host port
docker run -p 80 nginx  # Docker assigns random port

# Multiple ports
docker run -p 8080:80 -p 8443:443 nginx

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

```
# Check mapped ports
docker port container_name
```

7.5 Environment Variables in Containers

Set runtime configuration.

Real Example:

bash

```
# Single variable
docker run -e "DATABASE_URL=postgres://user:pass@db/app" myapp

# Multiple variables
docker run -e "NODE_ENV=production" -e "PORT=3000" myapp

# File with variables
docker run --env-file  .env myapp

# From host environment
docker run -e "HOSTNAME" myapp  # Passes host's HOSTNAME
```

.env file:

text

```
DB_HOST=localhost
DB_PORT=5432
DEBUG=false
```

7.6 Inspecting Containers

Get detailed container information.

Real Examples:

bash

```
# All container details (JSON)
docker inspect container_name

# Specific information
docker inspect --format='{{.NetworkSettings.IPAddress}}' container_name
docker inspect --format='{{.State.Status}}' container_name
docker inspect --format='{{json .Config}}' container_name

# Live resource usage
docker stats container_name

# Process list inside container
docker top container_name
```

7.7 Container Logs

View container output.

Real Examples:

bash

```
# View logs
docker logs container_name
```

```
# Follow logs (like tail -f)
docker logs -f container_name

# Show last N lines
docker logs --tail 100 container_name

# Show logs with timestamps
docker logs -t container_name

# Show logs since specific time
docker logs --since 2024-01-15T10:30:00 container_name

# Export logs to file
docker logs container_name > app.log
```

Interview Scenario: Debugging a crashing container:

bash

```
# Check why container exited
docker ps -a | grep Exit

# See exit logs
docker logs container_name

# Check exit code
docker inspect --format='{{.State.ExitCode}}' container_name
```

8. Docker Networking

8.1 What is Docker Networking

Docker provides network isolation between containers and host.

Default Networks:

bash

```
$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
abc123         bridge    bridge    local
def456         host      host      local
ghi789         none      null      local
```

8.2 Types of Docker Networks

Bridge Network (Default)

Default network for containers

Containers get IP addresses via DHCP

Port mapping needed for external access

bash

```
# Run container on bridge network
docker run -d --name web --network bridge nginx

# Custom bridge network
docker network create mynetwork
docker run -d --name app1 --network mynetwork myapp
docker run -d --name app2 --network mynetwork myapp
# app1 and app2 can communicate by name
```

Host Network

Container uses host's network directly

No network isolation, better performance

bash

```
docker run -d --name web --network host nginx
# Container port 80 available on host port 80 directly
```

None Network

No networking

Only loopback interface (127.0.0.1)

bash

```
docker run -d --name isolated --network none myapp
# Completely isolated from network
```

Overlay Network

For multi-host Docker Swarm clusters

Containers across hosts can communicate

bash

```
# In Swarm mode
```

```
docker network create --driver overlay myoverlay
```

8.3 Custom Networks

Create isolated networks for applications.

Real Example:

bash

```
# Create custom network
docker network create --driver bridge \
  --subnet 172.20.0.0/16 \
  --gateway 172.20.0.1 \
  app-network

# Run containers on custom network
docker run -d --name db --network app-network postgres
docker run -d --name api --network app-network \
  --link db:database \
  -e "DB_HOST=database" \
  api-server

# Containers can communicate using names
# api can connect to db at hostname "database"
```

8.4 Container-to-Container Communication

Methods:

Default Bridge: Use IP addresses

Custom Bridge: Use container names

Links: Legacy method (deprecated)

Network Aliases: Multiple names

Real Example:

bash

```
# Create network
docker network create mynet

# Run containers
docker run -d --name redis --network mynet redis:alpine
docker run -d --name app --network mynet \
  -e "REDIS_HOST=redis" \
  myapp

# In myapp, connect to redis using hostname "redis"
```

Testing Connectivity:

bash

```
# From host, test container connectivity
docker exec app ping redis
```

```
# Check network details
docker network inspect mynet
```

9. Docker Volumes & Storage

9.1 Why Volumes are Needed

Containers are ephemeral - all changes are lost when container is removed. Volumes provide persistent storage.

Real Problem: Database container gets recreated, all data lost. Solution: Use volume.

9.2 Types of Storage

Volumes (Managed by Docker)

bash

```
# Create volume
docker volume create dbdata

# Use volume
docker run -d \
  --name postgres \
  -v dbdata:/var/lib/postgresql/data \
  postgres

# List volumes
docker volume ls

# Inspect volume
docker volume inspect dbdata
```

```
# Remove unused volumes
docker volume prune
```

Bind Mounts (Host path)

bash

```
# Mount host directory
docker run -d \
  --name dev-server  \
  -v /home/user/app:/app \
  node-app

# Read-only bind mount
docker run -d \
  -v /config:/app/config:ro \
  myapp
```

tmpfs (In-memory)

bash

```
# Temporary in-memory storage
docker run -d \
  --name temp-app \
  --tmpfs /tmp \
  myapp
```

Comparison:

Type	Location	Managed by	Use Case
Volume	Docker area	Docker	Production, persistent data

Bind Mount	Host filesystem	User	Development, config files
tmpfs	Memory	Docker	Temporary, sensitive data

9.3 Creating & Using Volumes

Real Example - Database with volume:

bash

```
# Create named volume
docker volume create mysql_data

# Run MySQL with volume
docker run -d \
  --name mysql_db \
  -e MYSQL_ROOT_PASSWORD=secret \
  -v mysql_data:/var/lib/mysql \
  mysql:8.0

# Even if container removed, data persists
docker rm -f mysql_db
docker run -d \
  --name new_mysql \
  -v mysql_data:/var/lib/mysql \
  mysql:8.0
# Data still exists!
```

Backup Volume:

bash

```
# Backup volume data
docker run --rm \
  -v mysql_data:/source \
  -v $(pwd):/backup \
  alpine tar czf /backup/backup.tar.gz -C /source .

# Restore to volume
docker run --rm \
  -v mysql_data:/target \
  -v $(pwd):/backup \
  alpine tar xzf /backup/backup.tar.gz -C /target
```

9.4 Data Persistence in Docker

Best Practices:

Use named volumes for production data

Regular backups of volumes

Avoid storing data in container layer

Use read-only mounts for configs

Real Scenario: Multi-container app with volumes:

yaml

```
# docker-compose.yml
```

```
version: '3.8'
services:
  db:
    image: postgres:13
    volumes:
      - postgres_data:/var/lib/postgresql/data
    environment:
      POSTGRES_PASSWORD: secret

  app:
    image: myapp:latest
    volumes:
      - app_logs:/app/logs
      - ./config:/app/config:ro

volumes:
  postgres_data:
  app_logs:
```

10. Docker Compose (Multi-Container Applications)

10.1 What is Docker Compose

A tool for defining and running multi-container Docker applications using YAML files.

Real Example: Instead of multiple docker run commands:

bash

```
# Traditional way (painful!)
```

```
docker run -d --name db postgres

docker run -d --name redis redis

docker run -d --name app --link db --link redis myapp
```

With Compose: Single docker-compose up

10.2 docker-compose.yml Structure

Basic Structure:

yaml

```
version: '3.8'  # Compose file version

services:       # Containers to run
  web:
    image: nginx:alpine
    ports:
      - "80:80"

  api:
    build: ./api
    environment:
      - DB_HOST=db

  db:
    image: postgres:13
    environment:
      POSTGRES_PASSWORD: secret
```

```
networks:      # Custom networks
  app-network:
    driver: bridge

volumes:       # Named volumes
  db-data:
```

10.3 Services

Each service becomes a container.

Real Example:

yaml

```
services:
  frontend:
    build: ./frontend
    ports:
      - "3000:3000"
    depends_on:
      - backend

  backend:
    build: ./backend
    environment:
      DATABASE_URL: postgres://user:pass@db/app
    depends_on:
      - database

  database:
    image: postgres:13
    environment:
      POSTGRES_DB: app
      POSTGRES_USER: user
      POSTGRES_PASSWORD: pass
    volumes:
      - postgres_data:/var/lib/postgresql/data

  redis:
    image: redis:alpine
    command: redis-server --appendonly yes
```

10.4 Networks in Compose

Define custom networks for service communication.

Real Example:

yaml

```
version: '3.8'
services:
  web:
    image: nginx
    networks:
      - frontend
  api:
    image: node-app
    networks:
      - frontend
      - backend

  db:
    image: postgres
    networks:
      - backend

networks:
  frontend:
    driver: bridge
    ipam:
      config:
        - subnet: 172.20.0.0/16

  backend:
    driver: bridge
```

10.5 Volumes in Compose

Define persistent storage.

Real Example:

yaml

```
services:
  database:
    image: mysql:8.0
    volumes:
      - db_data:/var/lib/mysql
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql:ro
      - ./my.cnf:/etc/mysql/conf.d/my.cnf:ro

  app:
    image: myapp
    volumes:
      - ./app:/app
      - /app/node_modules  # anonymous volume

volumes:
  db_data:
    driver: local
```

10.6 Running Multiple Containers Together

Compose Commands:

bash

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

# Stop all services
```

```
docker-compose down


# View logs
docker-compose logs -f
docker-compose logs service_name


# Scale services
docker-compose up --scale api=3 --scale worker=5


# Build and start
docker-compose up --build


# Execute command in service
docker-compose exec api bash
docker-compose exec db psql -U postgres
```

10.7 Real-world Example: Web App + Database

Complete docker-compose.yml:

yaml

```
version: '3.8'


services:
  # React Frontend
  frontend:
    build: ./frontend
    ports:
      - "80:3000"
    volumes:
      - ./frontend:/app
      - /app/node_modules
    environment:
      - REACT_APP_API_URL=http://api:5000
    depends_on:
      - api

  # Node.js Backend API
  api:
    build: ./backend
    ports:
      - "5000:5000"
    volumes:
      - ./backend:/app
      - /app/node_modules
    environment:
      - NODE_ENV=development
      - DB_HOST=postgres
      - DB_PORT=5432
      - DB_USER=appuser
      - DB_PASSWORD=secret
      - DB_NAME=mydb
      - REDIS_URL=redis://redis:6379
    depends_on:
```

```
  - postgres
  - redis
 healthcheck:
  test: ["CMD", "curl", "-f", "http://localhost:5000/health"]
  interval: 30s
  timeout: 10s
  retries: 3

# PostgreSQL Database
postgres:
 image: postgres:13-alpine
 environment:
  - POSTGRES_USER=appuser
  - POSTGRES_PASSWORD=secret
  - POSTGRES_DB=mydb
 volumes:
  - postgres_data:/var/lib/postgresql/data
  - ./init.sql:/docker-entrypoint-initdb.d/init.sql
 ports:
  - "5432:5432"

# Redis Cache
redis:
 image: redis:6-alpine
 command: redis-server --appendonly yes
 volumes:
      - redis_data:/data
    ports:
      - "6379:6379"

  # pgAdmin (Database GUI)
  pgadmin:
    image: dpage/pgadmin4
    environment:
      - PGADMIN_DEFAULT_EMAIL=admin@example.com
      - PGADMIN_DEFAULT_PASSWORD=admin
    ports:
      - "8080:80"
    depends_on:
      - postgres

volumes:
  postgres_data:
  redis_data:

networks:
  default:
    name: app-network
    driver: bridge
```

Usage:

bash

```
# Start the entire stack
docker-compose up -d

# Check status
docker-compose ps

# View logs
docker-compose logs -f

# Scale API instances
docker-compose up -d --scale api=3

# Stop everything
docker-compose down

# Stop and remove volumes
docker-compose down -v
```

11. Docker Registry & Image Management

11.1 Public vs Private Registries

Public Registries:

Docker Hub: Default public registry

GitHub Container Registry: Integrated with GitHub

Quay.io: Red Hat's registry

Private Registries:

Docker Trusted Registry: Docker's enterprise solution

Azure Container Registry: Microsoft Azure

Amazon ECR: AWS

Google Container Registry: GCP

Self-hosted: Run registry as container

Real Example - Company Setup:

Public images (nginx, redis) from Docker Hub

Internal base images from private registry

Application images in organization's private registry

11.2 Pushing Images to Docker Hub

Step-by-step:

bash

```
# 1. Create Docker Hub account
# 2. Login from CLI
docker login

# 3. Tag image with username/repository
docker tag myapp:latest username/myapp:1.0
docker tag myapp:latest username/myapp:latest

# 4. Push image
docker push username/myapp:1.0
docker push username/myapp:latest

# 5. Verify on Docker Hub
docker pull username/myapp:1.0
```

Organization Images:

bash

```
# Tag for organization
docker tag myapp:latest mycompany/api-service:v2.1

# Push to organization
docker push mycompany/api-service:v2.1
```

11.3 Pulling from Private Registry

Different Registry Examples:

Docker Hub (Private):

bash

```
docker login
docker pull mycompany/private-app:latest
```

Amazon ECR:

bash

```
# Get login command
aws ecr get-login-password --region us-east-1 | \
  docker login --username AWS --password-stdin 123456789.dkr.ecr.us-east-1.amazonaws.com

# Pull image
docker pull 123456789.dkr.ecr.us-east-1.amazonaws.com/myapp:latest
```

Self-hosted Registry:

bash

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

# Tag and push to local registry
docker tag myapp:latest localhost:5000/myapp:latest
docker push localhost:5000/myapp:latest

# Pull from local registry
docker pull localhost:5000/myapp:latest
```

11.4 Image Security & Scanning

Security Best Practices:

Scan images for vulnerabilities:

bash

```
# Using Docker Scout (formerly Snyk)
docker scout quickview myapp:latest

# Using Trivy
trivy image myapp:latest

# Using Clair
clair-scanner --ip host.docker.internal myapp:latest
```

Use official images from trusted sources

Keep images updated with security patches

Implement image signing (Docker Content Trust)

Use minimal base images (Alpine, Distroless)

Real Example - Secure Pipeline:

bash

```
# Build image
docker build -t myapp:latest .
```

```
# Scan for vulnerabilities
docker scout cves  myapp:latest

# If passes, push to registry
docker push myregistry.com/myapp:latest

# In production, pull with content trust
export DOCKER_CONTENT_TRUST=1
docker pull myregistry.com/myapp:latest
```

12. Docker Security

12.1 Container Isolation

Containers provide process isolation through Linux namespaces and cgroups.

Isolation Mechanisms:

PID namespace: Isolated process tree

Network namespace: Isolated network stack

Mount namespace: Isolated filesystem

UTS namespace: Isolated hostname

User namespace: Isolated user IDs

Real Example: Even if container runs as root, it's root inside container namespace, not on host.

12.2 Image Vulnerabilities

Common Vulnerabilities:

Outdated packages in base image

Unnecessary software included

Default credentials in images

SUID binaries that can be exploited

Mitigation:

dockerfile

```
# Use specific version, not latest
FROM alpine:3.14  # Good
FROM alpine:latest  # Bad

# Regularly update base images
# Run security scans
# Remove unnecessary packages
```

12.3 Least Privilege Principle

Run containers with minimal privileges.

Best Practices:

dockerfile

```
# Create non-root user
RUN groupadd -r appuser && useradd -r -g appuser appuser
USER appuser

# In docker run
docker run --user 1000:1000 myapp
```

Capabilities Management:

bash

```
# Drop all capabilities, add only needed
docker run --cap-drop=ALL --cap-add=NET_BIND_SERVICE nginx

# Read-only filesystem
docker run --read-only myapp

# No new privileges
docker run --security-opt=no-new-privileges  myapp
```

12.4 Secrets Management

Never store secrets in images or environment variables.

Docker Secrets (Swarm):

bash

```
# Create secret
echo "mysecretpassword" | docker secret create db_password -

# Use in service
docker service create \
  --name mysql \
  --secret source=db_password,target=db_password \
  mysql
```

For Docker Compose:

yaml

```
version: '3.8'
services:
  db:
    image: postgres
    secrets:
      - db_password
    environment:
      POSTGRES_PASSWORD_FILE: /run/secrets/db_password

secrets:
  db_password:
    file: ./db_password.txt
```

For standalone containers: Use external secret managers (AWS Secrets Manager, HashiCorp Vault).

12.5 Docker Bench for Security

Automated security checker for Docker.

Usage:

bash

```
# Run security audit
docker run -it --net host --pid host --userns host --cap-add audit_control \
  -e DOCKER_CONTENT_TRUST=1 \
  -v /var/lib:/var/lib \
  -v /var/run/docker.sock:/var/run/docker.sock \
  --label docker_bench_security \
  docker/docker-bench-security

# Fix issues reported
# Example: Ensure containers use trusted base images
# Example: Ensure Docker daemon runs with TLS authentication
```

13. Docker Performance & Optimization

13.1 Reducing Image Size

Techniques:

Use Alpine base images:

dockerfile

```
FROM node:16-alpine  # ~120MB
```

```
# vs
FROM node:16  # ~900MB
```

Clean up in same RUN layer:

dockerfile

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

Remove unnecessary files:

dockerfile

```
# Bad: Leaves npm cache
RUN npm install

# Good: Clean npm cache
RUN npm install && npm cache clean --force
```

Real Example Comparison:

Unoptimized: 1.2GB

Optimized: 180MB (85% reduction)

13.2 Multi-Stage Builds

Separate build environment from runtime.

Real Example - Go application:

```dockerfile
# Stage 1: Build
FROM golang:1.19 AS builder
WORKDIR /app
COPY . .
RUN go build -o myapp .

# Stage 2: Runtime
FROM alpine:3.16
WORKDIR /root/
COPY --from=builder /app/myapp .
RUN apk --no-cache add ca-certificates
CMD ["./myapp"]
```

Result: Final image contains only the binary, not build tools (~15MB vs ~800MB).

13.3 Caching in Docker

Docker caches layers to speed up builds.

Optimizing Cache Usage:

Order instructions properly:

```dockerfile
# Bad - COPY changes often, cache invalidated
```

```
COPY . .
RUN npm install


# Good - Install dependencies first
COPY package.json package-lock.json .
RUN npm install
COPY . .
```

Use .dockerignore:

text

```
node_modules/
.git/
*.log
Dockerfile
```

Leverage build cache in CI/CD:

bash

```
docker build --cache-from myapp:latest -t myapp:new .
```

13.4 Monitoring Containers

Monitoring Commands:

bash

```
# Live resource usage
docker stats


# Container process list
```

```
docker top container_name

# Inspect container resource limits
docker inspect --format='{{.HostConfig.Memory}}' container_name

# Log monitoring
docker logs --tail 100 -f container_name
```

Prometheus Monitoring Setup:

yaml

```
# docker-compose.yml
version: '3.8'
services:
  prometheus:
    image: prom/prometheus
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
    ports:
      - "9090:9090"

  node-exporter:
    image: prom/node-exporter
    volumes:
      - /proc:/host/proc:ro
      - /sys:/host/sys:ro
      - /:/rootfs:ro
    command:
      - '--path.procfs=/host/proc'
      - '--path.sysfs=/host/sys'
      - '--collector.filesystem.mount-points-exclude=^/(sys|proc|dev|host|etc)($$|/)'

  cadvisor:
    image: gcr.io/cadvisor/cadvisor
    volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:rw
      - /sys:/sys:ro
      - /var/lib/docker/:/var/lib/docker:ro
    ports:
      - "8080:8080"
```

14. Docker in DevOps & CI/CD

14.1 Docker in CI/CD Pipeline

Typical Pipeline:

Developer pushes code to Git

CI Server builds Docker image

Security scan on image

Push to registry

Deploy to environments

Real Example - Jenkins Pipeline:

```groovy
pipeline {
  agent any

  stages {
    stage('Build') {
      steps {
        sh 'docker build -t myapp:$BUILD_NUMBER .'
      }
    }

    stage('Test') {
      steps {
        sh 'docker run myapp:$BUILD_NUMBER npm test'
      }
    }

    stage('Scan') {
      steps {
        sh 'docker scan myapp:$BUILD_NUMBER'
      }
    }
    stage('Push') {
      steps {
        sh 'docker tag myapp:$BUILD_NUMBER myregistry.com/myapp:$BUILD_NUMBER'
        sh 'docker push myregistry.com/myapp:$BUILD_NUMBER'
      }
    }

    stage('Deploy') {
      steps {
        sh 'kubectl set image deployment/myapp myapp=myregistry.com/myapp:$BUILD_NUMBER'
      }
    }
  }
}
```

14.2 Docker with Jenkins

Jenkins Docker Setup:

Install Docker on Jenkins agents

Configure Docker Cloud in Jenkins

Use Docker for ephemeral build agents

docker-compose.yml for Jenkins:

yaml

```
version: '3.8'

services:
  jenkins:
    image: jenkins/jenkins:lts
    ports:
      - "8080:8080"
      - "50000:50000"
    volumes:
      - jenkins_home:/var/jenkins_home
      - /var/run/docker.sock:/var/run/docker.sock
    environment:
      - DOCKER_HOST=unix:///var/run/docker.sock

  jenkins-agent:
    build: ./agent
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
```

14.3 Docker with GitHub Actions

GitHub Actions Workflow:

yaml

```
name: Docker CI/CD
```

```
on:
 push:
  branches: [ main ]

jobs:
 build-and-push:
  runs-on: ubuntu-latest

  steps:
  - name: Checkout code
   uses: actions/checkout@v2

  - name: Login to Docker Hub
   uses: docker/login-action@v1
   with:
    username: ${{ secrets.DOCKER_USERNAME }}
    password: ${{ secrets.DOCKER_PASSWORD }}

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

  - name: Push Docker image
   run: |
    docker push myapp:${{ github.sha }}
```

```
        docker push myapp:latest

    - name: Deploy to Kubernetes
      run: |
        kubectl set image deployment/myapp myapp=myapp:${{ github.sha }}
```

14.4 Docker in Cloud (AWS, Azure, GCP)

AWS ECS Example:

json

```
{
  "family": "myapp",
  "containerDefinitions": [
    {
      "name": "web",
      "image": "123456789.dkr.ecr.us-east-1.amazonaws.com/myapp:latest",
      "cpu": 256,
      "memory": 512,
      "portMappings": [
        {
          "containerPort": 80,
          "hostPort": 80
        }
      ]
    }
  ]
}
```

Azure Container Instances:

bash

```
az container create \
  --resource-group myResourceGroup \
  --name myapp \
  --image myregistry.azurecr.io/myapp:latest \
  --dns-name-label myapp-dns \
  --ports 80
```

Google Cloud Run:

bash

```
# Deploy container
gcloud run deploy myapp \
  --image gcr.io/myproject/myapp:latest \
  --platform managed \
  --region us-central1 \
  --allow-unauthenticated
```

15. Docker vs Kubernetes (Interview Comparison)

15.1 Docker vs Kubernetes

Aspect	Docker	Kubernetes
Purpose	Container runtime & packaging	Container orchestration
Scope	Single host	Multiple hosts (cluster)
Scaling	Manual (docker-compose scale)	Automatic (HPA)
Networking	Basic (bridge, host, none)	Advanced (CNI plugins)
Storage	Volumes, bind mounts	Persistent volumes, storage classes

Load Balancing	Basic (port mapping)	Advanced (Ingress, Services)
Self-healing	Limited (restart policies)	Advanced (health checks, pod restart)

Analogy: Docker is like a shipping container, Kubernetes is like a global shipping management system.

15.2 When to Use Docker Only

Use Docker when:

Local development environment

Simple applications (single container)

Learning containers concepts

CI/CD build environments

Small-scale deployments (1-5 containers)

Real Example:

Developer laptop setup

Small company's internal tool

Prototype/MVP applications

Build agents in CI pipeline

15.3 When to Use Kubernetes

Use Kubernetes when:

Microservices architecture (10+ services)

High availability requirements

Auto-scaling needed

Multi-cloud or hybrid deployments

Complex networking requirements

Enterprise-grade applications

Real Example:

E-commerce platform with 50+ microservices

SaaS application with thousands of users

Global application needing multi-region deployment

Applications requiring zero-downtime updates

Interview Answer: "We use Docker to package our applications into containers, and Kubernetes to orchestrate and manage those containers across our production cluster, providing scaling, self-healing, and service discovery."

16. Advanced Docker Concepts

16.1 Namespaces & cgroups

Namespaces provide isolation:

bash

```
# PID namespace example
docker run -it --name container1 alpine sh
# Inside container: ps aux (only container processes)

# Network namespace
docker run -it --name container2 alpine sh
# ifconfig shows container's network, not host
```

cgroups control resources:

bash

```
# Limit memory to 100MB
docker run -it --memory="100m" alpine sh

# Limit CPU shares
docker run -it --cpu-shares=512 alpine sh
```

```
# Limit CPU cores
docker run -it --cpus="1.5" alpine sh
```

Real Example: Preventing a container from consuming all host memory:

bash

```
docker run -d --memory="1g" --memory-swap="2g" myapp
```

16.2 Swarm Mode

Docker's built-in orchestration.

Initialize Swarm:

bash

```
# Initialize swarm on manager
docker swarm init --advertise-addr <MANAGER-IP>

# Join worker nodes
docker swarm join --token <TOKEN> <MANAGER-IP>:2377

# Create service
docker service create --name web --replicas 3 -p 80:80 nginx

# Scale service
docker service scale web=5

# Update service
docker service update --image nginx:alpine web
```

16.3 Rolling Updates

Update containers without downtime.

Docker Swarm:

bash

```
docker service update \
  --image myapp:v2 \
  --update-parallelism 2 \
  --update-delay 10s \
  --update-failure-action rollback \
  myapp
```

Docker Compose:

bash

```
docker-compose up -d --no-deps --build service_name
```

16.4 Load Balancing

Docker Swarm has built-in load balancing across replicas.

Real Example:

bash

```
# Create service with 3 replicas
docker service create --name api --replicas 3 -p 8080:3000 myapi

# Traffic to host:8080 is load balanced across 3 containers
# Check which container handles request
curl http://localhost:8080/which-container
```

17. Common Docker Interview Questions

17.1 Freshers Level Questions

Q1: What is Docker and why use it?

A: Docker is a containerization platform that packages applications and dependencies into isolated containers. Benefits: consistency across environments, lightweight compared to VMs, fast deployment, efficient resource usage.

Q2: Difference between Docker image and container?

A: Image is a read-only template with application code and dependencies. Container is a running instance of an image. Analogy: Image is a Class, Container is an Object instance.

Q3: Basic Docker commands?

A: docker run, docker ps, docker build, docker pull, docker push, docker exec, docker logs

Q4: What is Dockerfile?

A: Text file with instructions to build a Docker image (FROM, COPY, RUN, CMD, etc.)

17.2 2–4 Years Experience Level

Q1: Explain Docker architecture?

A: Client-server architecture with Docker Client (CLI), Docker Daemon (server), Registry (image storage), Images (templates), Containers (running instances).

Q2: Docker network types?

A: Bridge (default), Host (shares host network), None (no network), Overlay (multi-host), Macvlan (assign MAC to container).

Q3: Docker volumes vs bind mounts?

A: Volumes are managed by Docker, stored in Docker area. Bind mounts mount host directories. Volumes are preferred for production.

Q4: Multi-stage builds?

A: Use multiple FROM statements to separate build environment from runtime, reducing final image size.

17.3 Scenario-Based Questions

Q1: Container is running but application not accessible?

A:

Check port mapping: docker ps to see ports

Check if app listens on correct interface (should be 0.0.0.0)

Check firewall rules

Check container logs: docker logs container_name

Q2: Docker build is slow?

A:

Optimize Dockerfile order (frequently changing layers last)

Use .dockerignore to exclude unnecessary files

Use build cache effectively

Consider multi-stage builds

Q3: "No space left on device" error?

A:

bash

```
# Cleanup commands
docker system prune
docker system prune -a
docker volume prune
docker image prune
```

17.4 Debugging Questions

Q1: How to debug a crashing container?

A:

bash

```
# Check exit code
docker inspect --format='{{.State.ExitCode}}' container

# Check logs
docker logs container
```

```
# Run with interactive shell
docker run -it --entrypoint=/bin/sh image_name

# Inspect container
docker inspect container
```

Q2: Container can't connect to another container?

A:

Check if on same network: `docker network inspect network_name`

Use container names not IPs (IPs change)

Check exposed ports

Test connectivity: `docker exec container1 ping container2`

18. Hands-On Practice Projects

18.1 Run a Node.js App in Docker

Project Structure:

text

```
node-app/
├── Dockerfile
├── package.json
├── index.js
└── .dockerignore
```

Dockerfile:

dockerfile

```
FROM node:16-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
COPY . .
EXPOSE 3000
USER node
CMD ["node", "index.js"]
```

index.js:

javascript

```
const express = require('express');
const app = express();
app.get('/', (req, res) => {
  res.send('Hello Docker!');
});
app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

Commands:

bash

```
docker build -t node-app .
docker run -d -p 3000:3000 node-app
curl http://localhost:3000
```

18.2 Dockerize a Java Spring Boot App

Dockerfile:

dockerfile

```
# Build stage
FROM maven:3.8-openjdk-17 AS builder
WORKDIR /app
COPY pom.xml .
RUN mvn dependency:go-offline
COPY src ./src
RUN mvn package -DskipTests

# Runtime stage
FROM openjdk:17-jdk-slim
WORKDIR /app
COPY --from=builder /app/target/*.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Build and Run:

bash

```
docker build -t spring-app .
docker run -d -p 8080:8080 spring-app
```

18.3 Docker + MySQL Project

docker-compose.yml:

yaml

```
version: '3.8'
services:
  db:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: rootpass
      MYSQL_DATABASE: mydb
      MYSQL_USER: user
      MYSQL_PASSWORD: userpass
    ports:
      - "3306:3306"
    volumes:
      - db_data:/var/lib/mysql
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql

  adminer:
    image: adminer
    ports:
      - "8080:8080"
    depends_on:
      - db

volumes:
  db_data:
```

Access: Adminer at http://localhost:8080

18.4 Full Stack App with Docker Compose

Complete Stack:

yaml

```
version: '3.8'

services:
  # React Frontend
  frontend:
    build: ./frontend
    ports:
      - "3000:3000"
    volumes:
      - ./frontend:/app
      - /app/node_modules
    environment:
      REACT_APP_API_URL: http://localhost:5000

  # Node.js API
  api:
    build: ./api
    ports:
      - "5000:5000"
```

```
 volumes:
  - ./api:/app
  - /app/node_modules
 environment:
  DB_HOST: db
  DB_USER: root
  DB_PASSWORD: secret
  DB_NAME: appdb
 depends_on:
  - db

# MySQL Database
db:
 image: mysql:8.0
 environment:
  MYSQL_ROOT_PASSWORD: secret
  MYSQL_DATABASE: appdb
 volumes:
  - mysql_data:/var/lib/mysql

# Nginx Reverse Proxy
nginx:
 image: nginx:alpine
 ports:
  - "80:80"
 volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
    depends_on:
      - frontend
      - api

volumes:
  mysql_data:
```

19. Docker Troubleshooting

19.1 Container Not Starting

Common Causes & Solutions:

bash

```
# 1. Check container logs
docker logs container_name

# 2. Check exit code
docker inspect --format='{{.State.ExitCode}}' container_name

# 3. Common exit codes:
# 0: Success
# 1: Application error
# 125: Docker run error
# 126: Command invoked cannot execute
# 127: Command not found
# 137: SIGKILL (often out of memory)
# 143: SIGTERM (graceful shutdown)
# 4. Run with interactive shell to debug
docker run -it --entrypoint=/bin/sh image_name

# 5. Check resource limits
docker inspect --format='{{.HostConfig.Memory}}' container_name
```

19.2 Port Not Exposed

Debugging Steps:

bash

```
# 1. Check if container is running
docker ps

# 2. Check port mapping
docker port container_name

# 3. Test from inside container
docker exec container_name curl localhost:80

# 4. Check if app listens on 0.0.0.0 (not 127.0.0.1)
docker exec container_name netstat -tulpn

# 5. Check firewall
sudo ufw status
```

19.3 Image Not Found

Solutions:

bash

```
# 1. Check if image exists locally
docker images | grep image_name

# 2. Pull image
docker pull image_name:tag

# 3. Check private registry login
docker login registry.company.com

# 4. Check tag exists in registry
curl https://registry.hub.docker.com/v2/repositories/library/nginx/tags/

# 5. Use correct image name format
# Wrong: docker pull nginx (if private registry needed)
# Right: docker pull registry.company.com/nginx
```

19.4 Permission Denied Errors

Solutions:

bash

```
# 1. Docker daemon not running
sudo systemctl status docker

# 2. User not in docker group
sudo usermod -aG docker $USER
```

```
# Logout and login again


# 3. Permission on Docker socket
ls -la /var/run/docker.sock


# 4. In container: run as non-root
# In Dockerfile:
USER 1000:1000


# 5. Volume permissions
# Mount with correct user ID
docker run -v $(pwd):/app:z -u $(id -u):$(id -g) myapp
```

19.5 Low Disk Space Issues

Cleanup Commands:

bash

```
# Check disk usage
docker system df


# Remove unused data
docker system prune


# Remove everything (more aggressive)
docker system prune -a --volumes


# Remove specific resources
```

```
docker container prune
docker image prune
docker volume prune
docker network prune

# Remove by filter
docker images --filter "dangling=true" -q | xargs docker rmi
docker images --filter "before=image:tag" -q | xargs docker rmi
```

20. Final Revision & Interview Crash Notes

20.1 One-Page Docker Cheat Sheet

text

```
# IMAGES
docker build -t name .        # Build image
docker images                 # List images
docker rmi image              # Remove image
docker history image          # Show image layers
docker tag old new            # Tag image

# CONTAINERS
docker run image              # Run container
docker ps                     # List running
docker ps -a                  # List all
docker stop container         # Stop container
docker start container        # Start stopped
docker rm container           # Remove container
docker exec -it container sh  # Exec into container
docker logs container        # View logs

# VOLUMES
docker volume ls             # List volumes
docker volume create name     # Create volume
docker volume rm name        # Remove volume

# NETWORK
docker network ls            # List networks
docker network create name    # Create network

# SYSTEM
docker info                # System info
docker version              # Version info
docker system df            # Disk usage
docker system prune          # Cleanup

# COMPOSE
docker-compose up           # Start services
docker-compose down          # Stop services
docker-compose logs          # View logs
```

20.2 Important Command Summary

Essential for Interviews:

bash

```
# Build and Run workflow
```

```
docker build -t app:tag .
docker run -d -p 80:80 --name myapp app:tag
docker logs myapp
docker exec -it myapp bash
docker stop myapp
docker rm myapp
docker rmi app:tag

# Debugging commands
docker inspect container
docker stats
docker top container
docker port container
docker diff container
```

20.3 Most Asked Interview Answers

Q: What is Docker?

A: Containerization platform that packages apps with dependencies into isolated, portable containers.

Q: Docker vs VM?

A: Docker shares host OS kernel, lightweight, fast startup. VMs have full OS, heavier, slower.

Q: Dockerfile instructions?

A: FROM, RUN, COPY, ADD, WORKDIR, EXPOSE, CMD, ENTRYPOINT, ENV, VOLUME, USER.

Q: Docker volumes?

A: Persistent storage that survives container lifecycle. Types: volumes (managed), bind mounts (host path), tmpfs (memory).

Q: Docker Compose?

A: Tool for defining/running multi-container apps using YAML file.

Q: Docker networking?

A: Bridge (default), Host (shared), None (isolated), Overlay (multi-host).

20.4 Common Mistakes

Using latest tag in production → Use specific versions

Running as root in container → Create non-root user

Not using .dockerignore → Large images, slow builds

Storing data in container layer → Use volumes

Exposing all ports → Only expose necessary ports

Not cleaning up → Regular docker system prune

Hardcoding configs → Use environment variables

Building from source in production image → Use multi-stage builds

Not scanning for vulnerabilities → Regular security scans

Ignoring container resource limits → Set memory/CPU limits

Golden Rule: Containers should be ephemeral (stateless) and immutable (don't modify running containers, rebuild image instead).