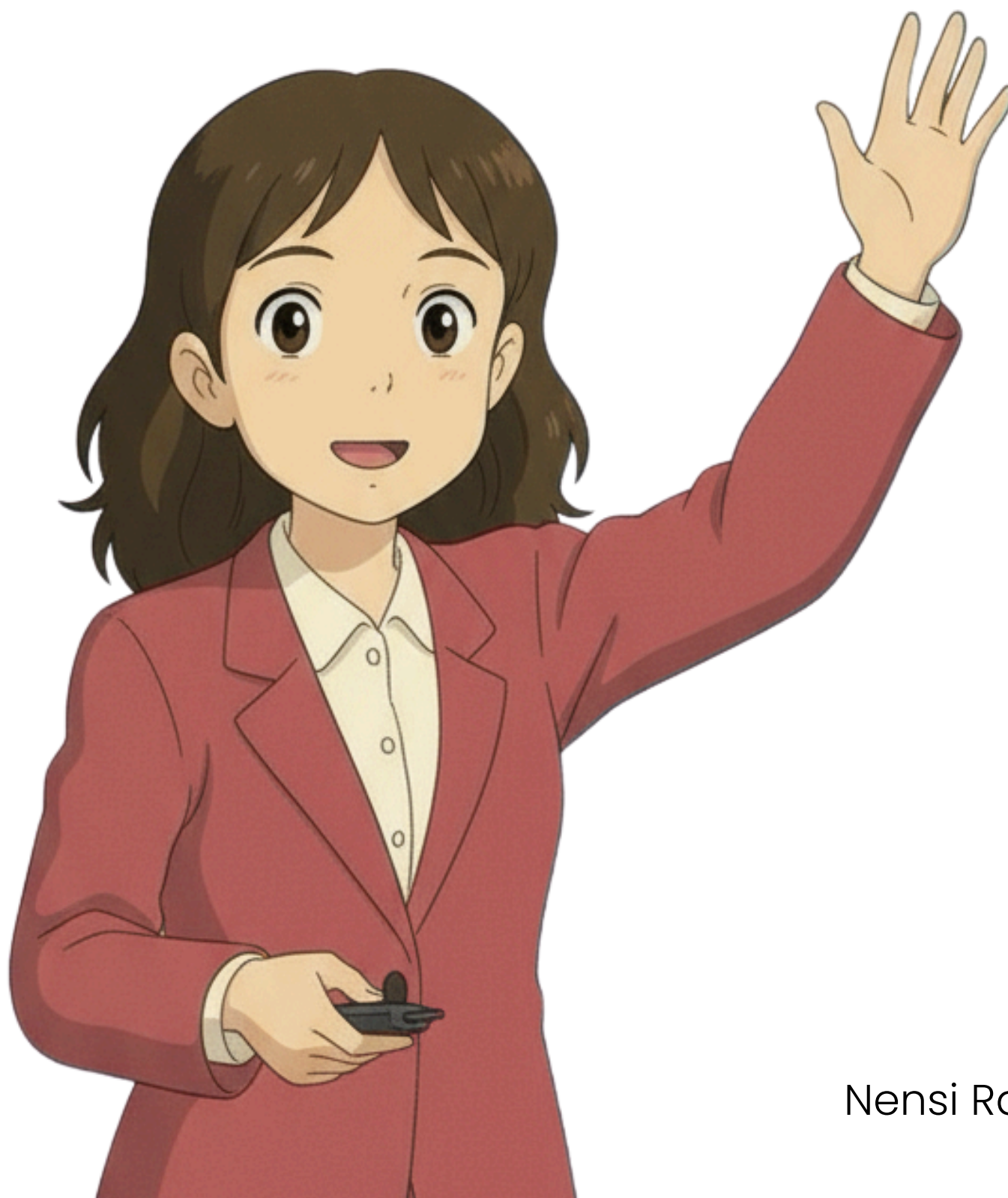
Top 30

Docker Interview

Questions & Answers



Nensi Ravaliya

Top 30 Docker Interview Questions with Answers

Asked at Top MNCs (Google, Amazon, Microsoft, Meta, Netflix, Uber, Airbnb)

SECTION 1: DOCKER FUNDAMENTALS

Q1: What is Docker and why is it important?

Answer:
Docker is an open-source containerization platform that packages applications and their dependencies into lightweight, portable containers that can run consistently across different environments.

Key Benefits:

- **Consistency**: "Works on my machine" problem solved - same environment from dev to production
- **Efficiency**: Containers share the host OS kernel, unlike VMs that require full OS instances
- **Portability**: Run anywhere - local machine, data center, or cloud (AWS, Azure, GCP)
- **Scalability**: Quickly spin up/down containers based on demand
- **Isolation**: Each container runs in its own isolated environment
- **Speed**: Containers start in seconds vs minutes for VMs

Industry Adoption:

- 87% market share in containerization
- Used by Uber, Airbnb, Google, Netflix, Amazon, Spotify, PayPal
- Essential for modern DevOps and CI/CD pipelines

Docker vs Virtual Machines:

Aspect	Docker Container	Virtual Machine
Size	Lightweight (MBs)	Heavy (GBs)
Startup	Seconds	Minutes
OS	Shares host kernel	Full OS per VM
Resources	Minimal overhead	Significant overhead
Isolation	Process-level	Hardware-level
Use Case	Microservices, apps	Full OS instances

Q2: Explain Docker Architecture

Answer:
Docker follows a **client-server architecture** with three main components:

1. Docker Client

- Command-line interface (CLI) tool
- Sends commands to Docker daemon via REST API
- Examples: `docker run`, `docker build`, `docker pull`
- Can communicate with remote Docker hosts

2. Docker Host (Docker Daemon)

- Core engine that runs on the host machine
- Manages containers, images, networks, and volumes
- Listens for Docker API requests
- Handles container lifecycle operations

3. Docker Registry

- Stores and distributes Docker images
- **Docker Hub**: Public registry (default)
- **Private registries**: AWS ECR, Google Container Registry, Azure Container Registry

- Push/pull images using `docker push` and `docker pull`

Docker Engine Components:

- **containerd**: High-level container runtime
- **runc**: Low-level container runtime that creates containers
- **Docker CLI**: User interface for Docker

Q3: What is the difference between a Docker Image and a Docker Container?

Answer:

Docker Image:

- **Read-only template** used to create containers
- Contains application code, libraries, dependencies, and configuration
- Built from Dockerfile instructions
- Stored in Docker registry
- Immutable - never changes once created
- Composed of layers (each instruction = one layer)

Docker Container:

- **Running instance** of a Docker image
- Writable layer on top of image
- Can be started, stopped, deleted, moved
- Isolated process with its own filesystem, network, and process space
- Multiple containers can run from the same image

Analogy:

- **Image** = Class in programming (blueprint)
- **Container** = Object/Instance of that class (running application)

Example:

```
# Pull an image (blueprint)
docker pull nginx:latest

# Run containers from that image (instances)
docker run -d --name web1 nginx:latest
docker run -d --name web2 nginx:latest
docker run -d --name web3 nginx:latest

# Now you have 1 image and 3 running containers
```

Key Commands:

```
# List images
docker images

# List running containers
docker ps

# List all containers (including stopped)
docker ps -a
```

Q4: What is a Dockerfile?

Answer:
A **Dockerfile** is a text file containing instructions to build a Docker image automatically.

Key Dockerfile Instructions:

Instruction	Purpose	Example
FROM	Base image to start from	`FROM ubuntu:20.04`
WORKDIR	Set working directory	`WORKDIR /app`
COPY	Copy files from host to image	`COPY . /app`
ADD	Like COPY but can extract tar & download URLs	`ADD app.tar.gz /app`

Instruction	Purpose	Example
RUN	Execute commands during build	`RUN apt-get update`
CMD	Default command when container starts	`CMD ["python", "app.py"]`
ENTRYPOINT	Configure container as executable	`ENTRYPOINT ["nginx"]`
EXPOSE	Document which ports app listens on	`EXPOSE 8080`
ENV	Set environment variables	`ENV APP_ENV=production`
ARG	Build-time variables	`ARG VERSION=1.0`
VOLUME	Create mount point for persistent data	`VOLUME /data`
LABEL	Add metadata to image	`LABEL version="1.0"`

Example Dockerfile (Python Web App):

```
# Base image
FROM python:3.9-slim

# Set working directory
WORKDIR /app

# Copy requirements first (for layer caching)
COPY requirements.txt .

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

# Copy application code
COPY . .

# Set environment variables
ENV FLASK_APP=app.py
ENV FLASK_ENV=production

# Expose port
EXPOSE 5000
```

```
# Health check
HEALTHCHECK --interval=30s --timeout=3s \
  CMD curl -f http://localhost:5000/health || exit 1

# Run application
CMD ["flask", "run", "--host=0.0.0.0"]
```

Build and Run:

```
# Build image
docker build -t myapp:1.0 .

# Run container
docker run -d -p 5000:5000 myapp:1.0
```

Q5: What is Docker Hub?

Answer:Docker Hub is a cloud-based public registry service for storing and sharing Docker images.

Key Features:

- **Public Repositories**: Free hosting for open-source images
- **Private Repositories**: Secure storage for proprietary images (paid plans)
- **Official Images**: Verified images from software vendors (nginx, mysql, redis, etc.)
- **Automated Builds**: Link GitHub/Bitbucket for automatic image builds
- **Webhooks**: Trigger actions after successful push
- **Organizations & Teams**: Collaborate with team members
- **Image Scanning**: Security vulnerability scanning

Common Docker Hub Commands:

```
# Login to Docker Hub
docker login
```

```
# Tag image for Docker Hub
docker tag myapp:1.0 username/myapp:1.0

# Push image to Docker Hub
docker push username/myapp:1.0

# Pull image from Docker Hub
docker pull username/myapp:1.0

# Search for images
docker search nginx
```

Alternative Registries:

- **AWS Elastic Container Registry (ECR)**
- **Google Container Registry (GCR)**
- **Azure Container Registry (ACR)**
- **Harbor** (self-hosted)
- **GitLab Container Registry**

SECTION 2: DOCKER COMMANDS & OPERATIONS

Q6: What are the most important Docker commands?

Answer:

Container Lifecycle:

```
# Run container
docker run -d --name web -p 8080:80 nginx

# List running containers
docker ps
```

```
# List all containers
docker ps -a

# Stop container
docker stop web

# Start stopped container
docker start web

# Restart container
docker restart web

# Pause container
docker pause web

# Unpause container
docker unpause web

# Kill container (force stop)
docker kill web

# Remove container
docker rm web

# Remove running container (force)
docker rm -f web
```

Image Management:

```
# Build image from Dockerfile
docker build -t myapp:1.0 .

# List images
docker images

# Pull image from registry
```

```
docker pull nginx:latest

# Push image to registry
docker push username/myapp:1.0

# Remove image
docker rmi myapp:1.0

# Remove dangling images
docker image prune

# Tag image
docker tag myapp:1.0 myapp:latest
```

Debugging & Inspection:

```
# View container logs
docker logs web
docker logs -f web  # Follow logs
docker logs --tail 100 web  # Last 100 lines

# Execute command in running container
docker exec -it web bash

# Inspect container details
docker inspect web

# View container resource usage
docker stats web

# View container processes
docker top web

# Copy files between container and host
```

```
docker cp web:/app/file.txt ./file.txt
docker cp ./file.txt web:/app/
```

System Management:

```
# View Docker system info
docker info

# View Docker version
docker version

# Remove all stopped containers
docker container prune

# Remove unused images
docker image prune -a

# Remove unused volumes
docker volume prune

# Remove everything unused
docker system prune -a --volumes

# View disk usage
docker system df
```

Q7: What is the difference between CMD and ENTRYPOINT?

Answer:

CMD:

- Provides **default arguments** to ENTRYPOINT or standalone command
- Can be **overridden** by command-line arguments
- Only the **last CMD** in Dockerfile is used

- Executed when container starts

ENTRYPOINT:

- Configures container to run as an **executable**
- **Cannot be overridden** easily (needs --entrypoint flag)
- Command-line arguments are **appended** to ENTRYPOINT
- Makes container behave like a binary

Comparison Table:

Feature	CMD	ENTRYPOINT
Purpose	Default command/args	Main executable
Override	Easy (docker run args)	Difficult (--entrypoint)
Multiple	Only last one used	Only last one used
Use Case	Flexible commands	Fixed executable

Examples:

1. CMD Only:

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

```
docker run myimage          # Output: Hello World
docker run myimage echo Hi   # Output: Hi (CMD overridden)
```

2. ENTRYPOINT Only:

```
FROM ubuntu
ENTRYPOINT ["echo"]
```

```
docker run myimage          # Output: (nothing)
docker run myimage Hello     # Output: Hello (appended to ENTRYPOINT)
```

3. ENTRYPOINT + CMD (Best Practice):

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

```
docker run myimage          # Output: Hello World
docker run myimage Goodbye   # Output: Goodbye (CMD replaced)
```

Real-World Example (Python App):

```
FROM python:3.9
WORKDIR /app
COPY . .

# ENTRYPOINT sets main executable
ENTRYPOINT ["python"]

# CMD provides default script
CMD ["app.py"]
```

```
docker run myapp            # Runs: python app.py
docker run myapp test.py     # Runs: python test.py
```

Q8: How do you debug a failing Docker container?

Answer:

Step-by-Step Debugging Process:

1. Check Container Status:

```
# List all containers
docker ps -a
```

```
# Check container status (running, exited, restarting, etc.)
# Look for exit code (0 = success, non-zero = error)
```

2. View Container Logs:

```
# View logs
docker logs <container_name>

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

# View last 50 lines
docker logs --tail 50 <container_name>

# Logs from specific time
docker logs --since 30m <container_name>

# Include timestamps
docker logs -t <container_name>
```

3. Inspect Container Details:

```
# Full container configuration
docker inspect <container_name>

# Get specific field
docker inspect --format='{{.State.Status}}' <container_name>
docker inspect --format='{{.NetworkSettings.IPAddress}}' <container_name>
```

4. Check Container Events:

```
# View Docker events
docker events --filter container=<container_name>
```

5. Execute Commands Inside Container:

```
# Access interactive shell
docker exec -it <container_name> /bin/bash
docker exec -it <container_name> sh  # If bash not available

# Run specific commands
docker exec <container_name> ps aux
docker exec <container_name> df -h
docker exec <container_name> netstat -tuln
```

6. Check Resource Usage:

```
# Real-time resource stats
docker stats <container_name>

# Check if container ran out of memory
docker inspect --format='{{.State.OOMKilled}}' <container_name>
```

7. Review Dockerfile and Build:

```
# Check image history
docker history <image_name>

# Rebuild with no cache
docker build --no-cache -t <image_name> .
```

Common Container Exit Codes:

Exit Code	Meaning	Common Cause
0	Success	Normal exit
1	Application error	Bug in application
137	SIGKILL (OOM)	Out of memory
139	SIGSEGV	Segmentation fault
143	SIGTERM	Graceful shutdown
126	Command not executable	Permission issue

Exit Code	Meaning	Common Cause
127	Command not found	Wrong command/path

8. Check Logs of Previous Container (if crashed):

```
docker logs <container_name> --tail 100 > container_logs.txt
```

Troubleshooting Checklist:

Container logs show errors?

Container has enough memory/CPU?

Ports are available and not conflicting?

Environment variables set correctly?

Volumes mounted correctly?

Network connectivity working?

Base image compatible?

Dependencies installed correctly?

SECTION 3: DOCKER NETWORKING

Q9: Explain Docker networking modes

Answer:
Docker provides several networking modes for container communication:

1. Bridge Network (Default)

- Default network for containers
- Isolated network on host
- Containers can communicate via IP
- Uses software bridge on host
- Good for single-host deployments

```
# Create bridge network
docker network create my-bridge

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

# Containers can ping each other by name
docker exec web1 ping web2
```

2. Host Network

- Container shares host's network namespace
- No network isolation
- Container uses host's IP directly
- Better performance (no NAT overhead)
- Port conflicts possible

```
# Run container with host network
docker run -d --name web --network host nginx

# Container accessible on host's IP
curl http://localhost:80
```

3. None Network

- No networking
- Complete network isolation
- Container has only loopback interface
- Used for security/testing

```
docker run -d --name isolated --network none nginx
```

4. Overlay Network

- Multi-host networking
- Used with Docker Swarm or Kubernetes
- Containers on different hosts can communicate
- Encrypted communication possible

```
# Create overlay network (Swarm mode)
docker network create --driver overlay my-overlay

# Deploy service using overlay
docker service create --name web --network my-overlay nginx
```

5. Macvlan Network

- Assigns MAC address to container
- Container appears as physical device on network
- Direct Layer 2 connectivity
- Used for legacy apps requiring direct network access

```
docker network create -d macvlan \
  --subnet=192.168.1.0/24 \
  --gateway=192.168.1.1 \
  -o parent=eth0 my-macvlan
```

Network Commands:

```
# List networks
docker network ls

# Inspect network
docker network inspect bridge

# Connect running container to network
docker network connect my-network web
```

```
# Disconnect container from network
docker network disconnect my-network web

# Remove network
docker network rm my-network

# Clean up unused networks
docker network prune
```

Q10: How do you expose container ports?

Answer:

Port Mapping Syntax:

```
docker run -p [host_port]:[container_port] image_name
```

Port Mapping Examples:

1. Basic Port Mapping:

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

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

2. Multiple Port Mappings:

```
# Map multiple ports
docker run -d \
  -p 8080:80 \
  -p 8443:443 \
  nginx
```

3. Bind to Specific IP:

```
# Bind to specific host IP
docker run -d -p 127.0.0.1:8080:80 nginx

# Only accessible from localhost
```

4. Random Host Port:

```
# Let Docker choose random host port
docker run -d -p 80 nginx

# Check assigned port
docker port <container_name>
```

5. Expose All Ports:

```
# Expose all ports defined in Dockerfile
docker run -d -P nginx

# -P maps all EXPOSE ports to random host ports
```

Dockerfile EXPOSE:

```
FROM nginx
EXPOSE 80 443

# EXPOSE is documentation only
# Doesn't actually publish ports
# Must use -p or -P at runtime
```

Check Port Mappings:

```
# View port mappings
docker port <container_name>

# Example output:
```

```
# 80/tcp → 0.0.0.0:8080
# 443/tcp → 0.0.0.0:8443

# Using inspect
docker inspect <container_name> | grep -A 10 Ports
```

UDP Port Mapping:

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

SECTION 4: DOCKER VOLUMES & DATA PERSISTENCE

Q11: What are Docker Volumes and why use them?

Answer:Docker Volumes provide persistent data storage that exists independently of container lifecycle.

Why Use Volumes:

- **Data Persistence**: Data survives container deletion
- **Data Sharing**: Multiple containers can share same volume
- **Performance**: Better I/O than bind mounts
- **Backup/Restore**: Easier to back up and migrate
- **Docker-managed**: Docker handles storage drivers

Types of Mounts:

1. Volumes (Recommended)

- Managed by Docker
- Stored in `/var/lib/docker/volumes/`
- Best for production
- Can be named or anonymous

```
# Create named volume
docker volume create my-data

# Run container with volume
docker run -d \
  --name web \
  -v my-data:/app/data \
  nginx

# List volumes
docker volume ls

# Inspect volume
docker volume inspect my-data

# Remove volume
docker volume rm my-data

# Remove unused volumes
docker volume prune
```

2. Bind Mounts

- Mount any host directory into container
- Full path required
- Good for development (code changes reflect immediately)
- Host-dependent (less portable)

```
# Bind mount syntax
docker run -d \
  --name web \
  -v /host/path:/container/path \
  nginx
```

```
# Or using --mount (more explicit)
docker run -d \
  --name web \
  --mount type=bind,source=/host/path,target=/container/path \
  nginx
```

3. tmpfs Mounts

- Stored in host memory only
- No disk I/O
- Temporary data (passwords, session data)
- Data lost when container stops

```
docker run -d \
  --name web \
  --tmpfs /app/temp \
  nginx
```

Volume vs Bind Mount:

Feature	Volume	Bind Mount
Managed by	Docker	User
Location	`/var/lib/docker/volumes/`	Any host path
Portability	High	Low
Performance	Better	Good
Use Case	Production	Development

Real-World Example (Database):

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

```
# Data persists even after container removal
docker rm -f mysql
docker run -d \
  --name mysql-new \
  -e MYSQL_ROOT_PASSWORD=secret \
  -v mysql-data:/var/lib/mysql \
  mysql:8.0
# Data still there!
```

Q12: What happens to data when a container is deleted?

Answer:

Without Volumes:

- **Data is lost** when container is deleted
- Container has writable layer on top of image
- Writable layer deleted with container

Example (Data Loss):

```
# Run container and create file
docker run -it --name test ubuntu bash
# Inside container: echo "important data" > /data.txt
# Exit container

# Remove container
docker rm test

# Data is GONE forever
```

With Volumes:

- **Data persists** after container deletion
- Volume exists independently

- Can be reused by new containers

Example (Data Persistence):

```
# Create volume
docker volume create app-data

# Run container with volume
docker run -it --name test -v app-data:/data ubuntu bash
# Inside container: echo "important data" > /data/file.txt
# Exit container

# Remove container
docker rm test

# Run new container with same volume
docker run -it --name test2 -v app-data:/data ubuntu bash
# Inside container: cat /data/file.txt
# Output: important data (STILL THERE!)
```

Best Practices:

- Always use volumes for important data
- Use bind mounts for development only
- Regular volume backups
- Monitor volume disk usage

SECTION 5: DOCKER COMPOSE

Q13: What is Docker Compose?

Answer:Docker Compose is a tool for defining and running multi-container Docker applications using a YAML file.

Key Benefits:

- **Single file** defines entire application stack

- **Easy orchestration** of multiple containers
- **Simplified commands** (one command to start/stop all)
- **Networking** automatically configured
- **Environment-specific** configurations
- **Development and testing** made easy

Docker Compose File (docker-compose.yml):

```
version: '3.8'

services:
  # Web application
  web:
    build: .
    ports:
      - "8080:80"
    environment:
      - DATABASE_URL=mysql://db:3306/myapp
    depends_on:
      - db
      - redis
    volumes:
      - ./app:/app
    networks:
      - app-network
    restart: always

  # Database
  db:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: secret
      MYSQL_DATABASE: myapp
    volumes:
      - db-data:/var/lib/mysql
```

```
    networks:
      - app-network
    restart: always

  # Cache
  redis:
    image: redis:alpine
    ports:
      - "6379:6379"
    networks:
      - app-network
    restart: always

volumes:
  db-data:

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

Docker Compose Commands:

```
# Start all services (detached mode)
docker-compose up -d

# View running services
docker-compose ps

# View logs
docker-compose logs
docker-compose logs -f web  # Follow specific service

# Stop all services
docker-compose stop

# Stop and remove containers, networks
```

```
docker-compose down

# Stop and remove containers, networks, volumes
docker-compose down -v

# Restart services
docker-compose restart

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

# Execute command in service
docker-compose exec web bash

# View resource usage
docker-compose top

# Build/rebuild images
docker-compose build
docker-compose build --no-cache
```

Use Cases:

- Development environments
- Testing environments
- Microservices applications
- CI/CD pipelines
- Demo/training setups

Q14: What is the difference between docker-compose up, run, and start?

Answer:

1. `docker-compose up`

- **Creates and starts** all services
- Builds images if needed
- Creates networks and volumes
- Attaches to containers (shows logs)
- Use `d` for detached mode

```
docker-compose up          # Start all, show logs
docker-compose up -d       # Start all, detached
docker-compose up web      # Start only 'web' and dependencies
docker-compose up --build  # Rebuild images before starting
```

2. `docker-compose run`

- **Runs one-off commands** in a new container
- Service must be defined in docker-compose.yml
- Does NOT start other services (unless depends_on)
- Useful for running tests, migrations, etc.

```
# Run migration in web service
docker-compose run web python manage.py migrate

# Run bash in web service
docker-compose run web bash

# Run command without starting dependencies
docker-compose run --no-deps web python test.py

# Remove container after command finishes
docker-compose run --rm web pytest
```

3. `docker-compose start`

- **Starts existing stopped containers**

- Does NOT create new containers
- Does NOT build images
- Quick restart of stopped services

```
docker-compose start       # Start all stopped services
docker-compose start web   # Start only 'web' service
```

Comparison:

Command	Creates Containers	Starts Services	Builds Images	One-off Commands
up	Yes	Yes	If needed	No
run	Yes (one)	No	No	Yes
start	No	Yes	No	No

Workflow Example:

```
# Initial setup
docker-compose up -d            # Create and start all

# Stop services
docker-compose stop             # Stop all services

# Restart services
docker-compose start            # Restart stopped services

# Run database migration
docker-compose run web python manage.py migrate

# Cleanup
docker-compose down             # Stop and remove all
```

SECTION 6: DOCKER SECURITY & BEST PRACTICES

Q15: What are Docker security best practices?

Answer:

1. Use Official and Verified Images

```
# Good: Official image
FROM nginx:1.21-alpine

# Bad: Unknown source
FROM randomuser/nginx
```

2. Use Specific Image Tags (Not 'latest')

```
# Good: Specific version
FROM python:3.9.7-alpine

# Bad: Unpredictable
FROM python:latest
```

3. Run as Non-Root User

```
FROM node:14-alpine

# Create non-root user
RUN addgroup -g 1001 -S nodejs && \
    adduser -S nodejs -u 1001

# Set ownership
WORKDIR /app
COPY --chown=nodejs:nodejs . .

# Switch to non-root user
USER nodejs

CMD ["node", "server.js"]
```

4. Minimize Image Layers and Size

```
# Bad: Multiple RUN commands
RUN apt-get update
RUN apt-get install -y package1
RUN apt-get install -y package2

# Good: Combined and cleaned up
RUN apt-get update && \
    apt-get install -y package1 package2 && \
    rm -rf /var/lib/apt/lists/*
```

5. Use Multi-Stage Builds

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

# Production stage
FROM node:14-alpine
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY package*.json ./
RUN npm install --production
USER node
CMD ["node", "dist/server.js"]
```

6. Scan Images for Vulnerabilities

```
# Docker Scout
docker scout cve nginx:latest
```

```
# Trivy
trivy image nginx:latest

# Snyk
snyk container test nginx:latest
```

7. Use Docker Secrets (Not Environment Variables)

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

# Use in service
docker service create \
  --name mysql \
  --secret db_password \
  mysql:8.0
```

8. Limit Container Resources

```
docker run -d \
  --name web \
  --memory="512m" \
  --cpus="1.0" \
  nginx
```

9. Use Read-Only Filesystem

```
docker run -d \
  --name web \
  --read-only \
  --tmpfs /tmp \
  nginx
```

10. Enable Docker Content Trust

```
# Enable image signing
export DOCKER_CONTENT_TRUST=1

# Now only signed images can be pulled/run
docker pull nginx:latest
```

11. Use .dockerignore File

```
# .dockerignore
.git
.env
node_modules
*.log
*.md
.dockerignore
Dockerfile
docker-compose.yml
```

12. Regular Updates

```
# Update base images regularly
docker pull nginx:alpine
docker build --pull -t myapp .
```

Security Checklist:

Use minimal base images (Alpine)

Run as non-root user

Don't store secrets in images

Scan images for vulnerabilities

Limit container capabilities

Use read-only filesystems when possible

Keep Docker engine updated

Implement network policies

Log and monitor container activity

Use trusted registries only

SECTION 7: ADVANCED TOPICS

Q16: What is multi-stage Docker build?

Answer:Multi-stage builds allow you to use multiple FROM statements in a Dockerfile to create smaller, more secure final images.

Benefits:

- **Smaller images**: Only production dependencies in final image
- **Faster deployments**: Smaller = faster push/pull
- **More secure**: No build tools in production image
- **Organized**: Separate build and runtime environments
- **No cleanup needed**: Build artifacts automatically excluded

Example: Java Application

Without Multi-Stage (Bad):

```
FROM maven:3.8-jdk-11
WORKDIR /app
COPY pom.xml .
COPY src ./src
RUN mvn clean package
CMD ["java", "-jar", "target/app.jar"]

# Problem: Image includes Maven, source code, build cache
# Size: ~700MB
```

With Multi-Stage (Good):

```
# Stage 1: Build
FROM maven:3.8-jdk-11 AS builder
WORKDIR /app
COPY pom.xml .
COPY src ./src
RUN mvn clean package -DskipTests

# Stage 2: Runtime
FROM openjdk:11-jre-slim
WORKDIR /app
COPY --from=builder /app/target/app.jar .
EXPOSE 8080
CMD ["java", "-jar", "app.jar"]

# Result: Only JRE + JAR in final image
# Size: ~200MB (65% smaller!)
```

Example: Node.js Application

```
# Build stage
FROM node:16 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build
RUN npm prune --production

# Production stage
FROM node:16-alpine
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/node_modules ./node_modules
COPY package*.json ./
EXPOSE 3000
```

```
USER node
CMD ["node", "dist/server.js"]
```

Example: Python Application

```
# Build stage
FROM python:3.9 AS builder
WORKDIR /app
COPY requirements.txt .
RUN pip install --user --no-cache-dir -r requirements.txt

# Production stage
FROM python:3.9-slim
WORKDIR /app
COPY --from=builder /root/.local /root/.local
COPY . .
ENV PATH=/root/.local/bin:$PATH
CMD ["python", "app.py"]
```

Build Specific Stage:

```
# Build only 'builder' stage (for testing)
docker build --target builder -t myapp:build .

# Build final stage (default)
docker build -t myapp:latest .
```

Q17: How do you optimize Docker image size?

Answer:

1. Use Alpine or Slim Base Images

```
# Huge: 1GB+
FROM ubuntu:20.04
```

```
# Better: 200MB
FROM python:3.9

# Best: 50MB
FROM python:3.9-alpine
```

2. Multi-Stage Builds

```
FROM node:16 AS build
COPY . .
RUN npm run build

FROM node:16-alpine
COPY --from=build /app/dist ./dist
```

3. Combine RUN Commands

```
# Bad: Creates 3 layers
RUN apt-get update
RUN apt-get install -y curl
RUN rm -rf /var/lib/apt/lists/*

# Good: Creates 1 layer
RUN apt-get update && \
    apt-get install -y curl && \
    rm -rf /var/lib/apt/lists/*
```

4. Use .dockerignore

```
.git
*.md
tests/
.env
node_modules
```

5. Don't Install Unnecessary Packages

```
# Install only what's needed
RUN apt-get update && \
    apt-get install -y --no-install-recommends \
    package1 \
    package2 && \
    rm -rf /var/lib/apt/lists/*
```

6. Clean Up in Same Layer

```
RUN apt-get update && \
    apt-get install -y package && \
    # ... use package ... && \
    apt-get purge -y package && \
    apt-get autoremove -y && \
    rm -rf /var/lib/apt/lists/*
```

7. Leverage Build Cache

```
# Copy dependency files first
COPY package.json package-lock.json ./
RUN npm install

# Then copy source (changes more often)
COPY . .
```

8. Use Specific COPY (Not ADD)

```
# Good: Only copy what's needed
COPY package.json ./
COPY src/ ./src/

# Bad: Copies everything
COPY . .
```

Size Comparison Example:

Approach	Image Size
Ubuntu + full dependencies	1.2 GB
Debian slim + selective install	450 MB
Alpine + minimal packages	85 MB
Multi-stage Alpine	45 MB

Q18: How do you handle container health checks?

Answer:

Dockerfile HEALTHCHECK:

```
FROM nginx:alpine

# Health check: curl endpoint every 30s
HEALTHCHECK --interval=30s \
            --timeout=3s \
            --start-period=5s \
            --retries=3 \
  CMD curl -f http://localhost/health || exit 1

# Or using wget
HEALTHCHECK --interval=30s --timeout=3s \
  CMD wget --quiet --tries=1 --spider http://localhost/health || exit 1
```

Health Check Parameters:

- `-interval`: How often to run check (default: 30s)
- `-timeout`: Max time for check to complete (default: 30s)
- `-start-period`: Grace period before checks start (default: 0s)
- `-retries`: Failed checks before "unhealthy" (default: 3)

Docker Compose Health Check:

```
version: '3.8'
services:
```

```
  web:
    image: myapp
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost/health"]
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 40s

  db:
    image: postgres
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 10s
      timeout: 5s
      retries: 5
```

Check Health Status:

```
# View health status
docker ps

# Detailed health info
docker inspect --format='{{json .State.Health}}' web | jq

# View health check logs
docker inspect web | grep -A 10 Health
```

Health States:

- `starting`: Health check not yet run
- `healthy`: Health check passing
- `unhealthy`: Health check failing

SECTION 8: DOCKER IN PRODUCTION

Q19: How do you monitor Docker containers in production?

Answer:

1. Built-in Docker Commands:

```
# Real-time resource usage
docker stats

# Specific container stats
docker stats web

# Container logs
docker logs -f web

# System-wide info
docker system df
docker system events
```

2. Prometheus + Grafana (Recommended)

docker-compose.yml:

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

  grafana:
    image: grafana/grafana
    ports:
      - "3000:3000"
    volumes:
```

```
      - grafana-data:/var/lib/grafana
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=secret

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

volumes:
  prometheus-data:
  grafana-data:
```

3. ELK Stack (Logging)

```
version: '3.8'
services:
  elasticsearch:
    image: elasticsearch:7.14.0
    environment:
      - discovery.type=single-node
    ports:
      - "9200:9200"

  logstash:
    image: logstash:7.14.0
    volumes:
      - ./logstash.conf:/usr/share/logstash/pipeline/logstash.conf

  kibana:
    image: kibana:7.14.0
```

```
    ports:
      - "5601:5601"
```

Key Metrics to Monitor:

- CPU usage per container
- Memory usage and limits
- Network I/O
- Disk I/O
- Container restart count
- Application-specific metrics

Q20: What is Docker Swarm?

Answer:Docker Swarm is Docker's native clustering and orchestration tool for managing a cluster of Docker engines.

Key Features:

- **Cluster management**: Multiple Docker hosts as single virtual host
- **Declarative service model**: Define desired state
- **Scaling**: Scale services up/down easily
- **Load balancing**: Built-in load balancer
- **Rolling updates**: Zero-downtime deployments
- **Service discovery**: Automatic DNS-based discovery

Initialize Swarm:

```
# Initialize manager node
docker swarm init --advertise-addr 192.168.1.100

# Join worker nodes (run on worker machines)
docker swarm join --token <worker-token> 192.168.1.100:2377
```

```
# View nodes
docker node ls
```

Deploy Service:

```
# Create service
docker service create \
  --name web \
  --replicas 3 \
  --publish 8080:80 \
  nginx

# List services
docker service ls

# View service details
docker service ps web

# Scale service
docker service scale web=5

# Update service (rolling update)
docker service update --image nginx:1.21 web

# Remove service
docker service rm web
```

Docker Swarm vs Kubernetes:

Feature	Docker Swarm	Kubernetes
Complexity	Simple	Complex
Setup	Easy	Difficult
Learning Curve	Low	High
Scalability	Good	Excellent
Ecosystem	Limited	Extensive

Feature	Docker Swarm	Kubernetes
Use Case	Small-medium	Enterprise

SECTION 9: SCENARIO-BASED QUESTIONS

Q21-30: Quick-Fire Scenarios

Q21: How do you share data between multiple containers?Answer: Use Docker volumes. Multiple containers can mount the same volume:

```
docker volume create shared-data
docker run -v shared-data:/data container1
docker run -v shared-data:/data container2
```

Q22: Container keeps restarting. How to debug?Answer:

```
# Check logs
docker logs --tail 100 container_name

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

# Run with different command to debug
docker run -it --entrypoint /bin/sh image_name
```

Q23: How to limit container memory and CPU?Answer:

```
docker run -d \
  --memory="512m" \
  --memory-swap="1g" \
  --cpus="1.5" \
  --cpu-shares=512 \
  nginx
```

Q24: How to copy files from container to host?Answer:

```
# Container to host
docker cp container_name:/path/file.txt ./file.txt

# Host to container
docker cp ./file.txt container_name:/path/
```

Q25: How to run Docker commands without sudo?Answer:

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

# Logout and login again
newgrp docker
```

Q26: How to save and load Docker images?Answer:

```
# Save image to tar file
docker save -o myapp.tar myapp:latest

# Load image from tar
docker load -i myapp.tar

# Export container filesystem
docker export container_name > container.tar

# Import container
docker import container.tar myapp:imported
```

Q27: How to view real-time container events?Answer:

```
docker events --filter container=web
docker events --filter type=container
docker events --since 1h
```

Q28: How to remove all stopped containers?Answer:

```
# Remove all stopped containers
docker container prune

# Remove all containers (including running)
docker rm -f $(docker ps -aq)

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

Q29: How to update running container without downtime?Answer:
Use Docker Compose or Swarm for rolling updates:

```
# Swarm
docker service update --image nginx:new web

# Compose
docker-compose up -d --no-deps --build web
```

Q30: How to troubleshoot networking issues?Answer:

```
# Check container IP
docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
web

# Test connectivity
docker exec web ping db

# Check DNS
docker exec web nslookup db

# View network details
docker network inspect bridge
```

KEY TAKEAWAYS

For Interview Success:

1. Understand Docker architecture deeply
2. Know difference between images and containers
3. Master essential Docker commands
4. Understand networking modes
5. Know volume types and persistence
6. Learn Docker Compose for multi-container apps
7. Practice security best practices
8. Be ready for troubleshooting scenarios
9. Understand Docker vs VM differences
10. Know when to use Docker Swarm vs Kubernetes

Hands-On Practice:

- Install Docker locally
- Build custom images
- Run multi-container apps with Compose
- Practice debugging failing containers
- Implement health checks
- Optimize image sizes
- Test different networking modes

Resources:

- Official Docker Docs: https://docs.docker.com
- Docker Hub: https://hub.docker.com
- Play with Docker: https://labs.play-with-docker.com
- Docker Cheat Sheet: Essential commands reference

Interview Preparation Checklist:

Can explain Docker architecture

Comfortable with Dockerfile creation

Know essential Docker CLI commands

Understand networking modes

Can work with volumes and persistence

Familiar with Docker Compose

Know security best practices

Can troubleshoot common issues

Understand multi-stage builds

Ready for scenario-based questions

Target Audience: Freshers to 5+ years experience

Interview Level: Entry to Senior positions at MNCs

Repost and Follow

Nensi Ravaliya

for more content

Want to build your career in cloud?

Subscribe to Yatri Cloud Channel