Complete Docker Mastery Guide - Interview Ready

Table of Contents

Docker Fundamentals

What is Docker?

Docker is a platform that packages your application and all its dependencies into containers. Think of it like shipping containers for code. Just like how physical shipping containers can be moved from ships to trucks to trains without unpacking, Docker containers can run anywhere: your laptop, a server, the cloud.

Why Docker Exists

The Problem Before Docker:

- "It works on my machine" syndrome
- Different environments (dev, staging, production) had different configurations
- Dependencies hell - installing the right versions of libraries, languages, tools
- Wasted resources running full virtual machines

What Docker Solves:

- Consistent environments across all stages
- Faster deployment (seconds vs minutes)
- Better resource utilization than VMs
- Easier scaling and orchestration
- Simplified dependency management

Docker vs Virtual Machines

Virtual Machines:

```
Host OS
  └── Hypervisor
      ├── Guest OS (entire OS)
      │   └── App A
      ├── Guest OS (entire OS)
      │   └── App B
```

Docker Containers:

```
Host OS
  └── Docker Engine
      ├── Container (shares host kernel)
      │   └── App A + dependencies
      ├── Container (shares host kernel)
      │   └── App B + dependencies
```

Key Differences:

- Containers share the host OS kernel, VMs include full OS
- Containers start in seconds, VMs take minutes
- Containers use MBs of space, VMs use GBs
- You can run 10x more containers than VMs on the same hardware

---

Core Concepts

1. Images

A Docker image is a read-only template. It's like a class in programming. Contains:

- Base operating system (usually minimal, like Alpine Linux)
- Application code
- Dependencies and libraries
- Configuration files
- Environment variables

2. Containers

A container is a running instance of an image. It's like an object in programming. Features:

- Isolated process with its own filesystem
- Has its own network interface
- Can be started, stopped, moved, deleted
- Ephemeral by default (data lost when deleted)

3. Docker Engine

The core Docker software that:

- Builds images
- Runs containers
- Manages Docker objects (images, containers, networks, volumes)

Components:

- **Docker Daemon** (`dockerd`): Background service that manages Docker objects
- **Docker CLI** (`docker`): Command-line tool you use
- **REST API**: How CLI talks to daemon

4. Docker Registry

A storage system for Docker images:

- **Docker Hub**: Public registry (like GitHub for Docker images)
- **Private Registries**: For internal use (AWS ECR, Google GCR, Harbor)

5. Dockerfile

A text file with instructions to build an image. Think of it as a recipe.

Docker Images

Understanding Layers

Docker images are built in layers. Each instruction in a Dockerfile creates a new layer.

dockerfile

```
FROM ubuntu:20.04          # Layer 1: Base OS
RUN apt-get update         # Layer 2: Updated package lists
RUN apt-get install python # Layer 3: Python installed
COPY app.py /app/          # Layer 4: Your code
CMD ["python", "/app/app.py"] # Layer 5: Startup command
```

Why Layers Matter:

- Layers are cached - rebuilds are fast
- Layers are shared between images - saves disk space
- Only changed layers are rebuilt

Layer Caching Strategy:

- Put things that change rarely at the top

- Put things that change often at the bottom
- This keeps cache hits high

Image Commands

bash

```
# List images
docker images
docker image ls

# Pull an image
docker pull nginx:latest
docker pull ubuntu:20.04

# Build an image
docker build -t myapp:v1 .
docker build -t myapp:v1 -f Dockerfile.prod .

# Tag an image
docker tag myapp:v1 myapp:latest
docker tag myapp:v1 myregistry.com/myapp:v1

# Push to registry
docker push myregistry.com/myapp:v1

# Remove images
docker rmi nginx:latest
docker image rm myapp:v1

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

# Inspect image
docker image inspect nginx:latest

# View image history (layers)
docker history nginx:latest

# Save/Load images (for offline transfer)
docker save myapp:v1 > myapp.tar
docker load < myapp.tar

# Export/Import (from container)
```

```
docker export container_name > container.tar
docker import container.tar myapp:imported
```

Image Naming Convention

```
[registry]/[username]/[repository]:[tag]

Examples:
nginx:latest                    # Official image from Docker Hub
ubuntu:20.04                    # Official Ubuntu image
myusername/myapp:v1.0           # Your image on Docker Hub
gcr.io/myproject/myapp:latest   # Google Container Registry
localhost:5000/myapp:dev        # Local private registry
```

Docker Containers

Container Lifecycle

```
Created → Running → Paused → Stopped → Deleted
```

Essential Container Commands

bash

```
# Run a container
docker run nginx
docker run -d nginx                    # Detached (background)
docker run -d --name web nginx         # Named container
docker run -d -p 8080:80 nginx         # Port mapping
docker run -d -e ENV_VAR=value nginx  # Environment variable
docker run -it ubuntu bash             # Interactive with terminal

# List containers
docker ps              # Running containers
docker ps -a           # All containers (including stopped)
docker ps -q           # Only container IDs

# Start/Stop containers
docker start container_name
docker stop container_name
docker restart container_name
docker pause container_name   # Pause processes
docker unpause container_name

# Remove containers
docker rm container_name
docker rm -f container_name      # Force remove running container
docker container prune           # Remove all stopped containers

# Execute commands in running container
docker exec container_name ls /app
docker exec -it container_name bash  # Interactive shell

# View logs
docker logs container_name
docker logs -f container_name        # Follow logs (like tail -f)
docker logs --tail 100 container_name
docker logs --since 10m container_name

# View container details
docker inspect container_name
docker stats                         # Resource usage (live)
docker top container_name            # Running processes

# Copy files
```

```
docker cp file.txt container_name:/path/
docker cp container_name:/path/file.txt ./

# View port mappings
docker port container_name

# Attach to running container
docker attach container_name
```

Docker Run Options (Deep Dive)

bash

```
# Full example with common options
docker run \
  -d \                              # Detached mode
  --name myapp \                    # Container name
  --restart unless-stopped \        # Restart policy
  -p 8080:80 \                      # Port mapping host:container
  -p 127.0.0.1:8081:81 \            # Bind to specific IP
  -e DATABASE_URL=postgres://... \  # Environment variable
  --env-file .env \                 # Load env vars from file
  -v /host/path:/container/path \   # Volume mount
  -v myvolume:/data \               # Named volume
  --mount type=bind,src=...,dst=... \ # Alternative mount syntax
  --network mynetwork \             # Custom network
  --network-alias dbserver \        # Network alias
  -w /app \                         # Working directory
  --user 1000:1000 \                # Run as specific user
  -m 512m \                         # Memory limit
  --cpus 2 \                        # CPU limit
  --health-cmd "curl -f http://localhost || exit 1" \
  --health-interval 30s \           # Health check
  --log-driver json-file \          # Logging driver
  --log-opt max-size=10m \          # Log options
  myimage:tag \                     # Image
  python app.py                     # Command to run
```

Restart Policies

bash

```bash
# no: Don't restart (default)
docker run --restart no nginx

# always: Always restart if stopped
docker run --restart always nginx

# on-failure: Restart if exit code is non-zero
docker run --restart on-failure nginx
docker run --restart on-failure:5 nginx  # Max 5 retries

# unless-stopped: Always restart unless manually stopped
docker run --restart unless-stopped nginx
```

Dockerfile Deep Dive

Dockerfile Structure

dockerfile

```
# Syntax version (optional but recommended)
# syntax=docker/dockerfile:1

# Base image
FROM node:18-alpine

# Metadata
LABEL maintainer="you@example.com"
LABEL version="1.0"
LABEL description="My awesome app"

# Environment variables
ENV NODE_ENV=production
ENV APP_PORT=3000

# Working directory
WORKDIR /app

# Copy files
COPY package*.json ./
COPY . .

# Run commands (during build)
RUN npm install --production
RUN npm run build

# Expose ports (documentation only)
EXPOSE 3000

# Create volumes
VOLUME /app/data

# Set user (don't run as root)
USER node

# Default command (runtime)
CMD ["node", "server.js"]
```

All Dockerfile Instructions

FROM

dockerfile

```dockerfile
# Start from base image
FROM ubuntu:20.04

# Multi-stage build
FROM node:18 AS builder
FROM nginx:alpine AS runtime

# Platform-specific
FROM --platform=linux/amd64 ubuntu:20.04
```

RUN

dockerfile

```dockerfile
# Shell form (runs in /bin/sh -c)
RUN apt-get update && apt-get install -y curl

# Exec form (doesn't invoke shell)
RUN ["/bin/bash", "-c", "echo hello"]

# Multi-line for readability
RUN apt-get update && apt-get install -y \
    curl \
    git \
    vim \
    && rm -rf /var/lib/apt/lists/*
```

COPY vs ADD

dockerfile

```
# COPY - simple file copy (preferred)
COPY package.json /app/
COPY . /app/

# ADD - COPY with superpowers (use sparingly)
ADD https://example.com/file.txt /app/  # Can download URLs
ADD archive.tar.gz /app/                # Auto-extracts archives

# COPY with ownership
COPY --chown=node:node . /app/
```

CMD vs ENTRYPOINT

CMD: Default command, can be overridden

dockerfile

```
CMD ["python", "app.py"]
# docker run myimage          -> runs python app.py
# docker run myimage bash     -> runs bash (CMD overridden)
```

ENTRYPOINT: Command that always runs

dockerfile

```
ENTRYPOINT ["python", "app.py"]
# docker run myimage          -> runs python app.py
# docker run myimage --debug  -> runs python app.py --debug
```

Combined (best practice):

dockerfile

```dockerfile
ENTRYPOINT ["python", "app.py"]
CMD ["--mode", "production"]
# docker run myimage              -> python app.py --mode production
# docker run myimage --mode dev    -> python app.py --mode dev
```

ENV vs ARG

ENV: Available at build AND runtime

dockerfile

```dockerfile
ENV APP_PORT=3000
# Can use: $APP_PORT
# Persists in final image
```

ARG: Only available at build time

dockerfile

```dockerfile
ARG NODE_VERSION=18
FROM node:${NODE_VERSION}
# docker build --build-arg NODE_VERSION=20 .
# Does NOT persist in final image
```

WORKDIR

dockerfile

```dockerfile
WORKDIR /app
# All subsequent commands run in /app
# Creates directory if doesn't exist
RUN pwd  # prints /app
COPY . .  # copies to /app
```

EXPOSE

dockerfile

```dockerfile
EXPOSE 3000
EXPOSE 8080/tcp
EXPOSE 8081/udp
# Documentation only - doesn't actually publish ports
# Still need -p flag: docker run -p 3000:3000 myimage
```

VOLUME

dockerfile

```dockerfile
VOLUME /app/data
# Creates mount point
# Data persists even if container is deleted
```

USER

dockerfile

```dockerfile
USER node
# All subsequent commands run as 'node' user
# Container also runs as this user
# Security best practice - don't run as root
```

HEALTHCHECK

dockerfile

```
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
  CMD curl -f http://localhost/ || exit 1

# or
HEALTHCHECK NONE  # Disable healthcheck from base image
```

Best Practices for Dockerfiles

1. Use Specific Tags

dockerfile

```dockerfile
# Bad
FROM node:latest

# Good
FROM node:18.17-alpine
```

2. Minimize Layers

dockerfile

```dockerfile
# Bad - creates 3 layers
RUN apt-get update
RUN apt-get install -y curl
RUN apt-get install -y git

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

3. Order Matters (Caching)

dockerfile

```dockerfile
# Bad - code changes invalidate dependency layer
COPY . /app/
RUN npm install

# Good - dependencies cached unless package.json changes
COPY package*.json /app/
RUN npm install
COPY . /app/
```

4. Use .dockerignore

📋
✓

```
# .dockerignore file
node_modules
.git
.env
*.log
.DS_Store
dist
coverage
```

5. Don't Run as Root

📋
✓

dockerfile

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

6. Keep Images Small

📋
✓

dockerfile

```
# Use Alpine variants
FROM node:18-alpine

# Remove package manager cache
RUN apt-get update && apt-get install -y curl \
    && rm -rf /var/lib/apt/lists/*

# Use multi-stage builds (see below)
```

Real-World Dockerfile Examples

Node.js Application

dockerfile

```
FROM node:18-alpine

WORKDIR /app

# Copy package files
COPY package*.json ./

# Install dependencies
RUN npm ci --only=production

# Copy application code
COPY . .

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

USER nodejs

EXPOSE 3000

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

Python Application

dockerfile

```dockerfile
FROM python:3.11-slim

WORKDIR /app

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

# Copy application
COPY . .

# Create non-root user
RUN useradd -m -u 1000 appuser && \
    chown -R appuser:appuser /app

USER appuser

EXPOSE 8000

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

Go Application

dockerfile

```
FROM golang:1.21-alpine AS builder

WORKDIR /app

COPY go.mod go.sum ./
RUN go mod download

COPY . .
RUN CGO_ENABLED=0 GOOS=linux go build -o main .

# Final stage
FROM alpine:latest

RUN apk --no-cache add ca-certificates

WORKDIR /root/

COPY --from=builder /app/main .

EXPOSE 8080

CMD ["./main"]
```

Docker Networking

Network Types

1. Bridge (Default)

- Default network for containers
- Containers can communicate with each other by IP
- Need port publishing to access from host

bash

```bash
docker run -d --network bridge nginx
```

2. Host

- Container shares host's network

- No isolation
- Better performance (no network translation)

bash

```bash
docker run -d --network host nginx
# nginx accessible on host's port 80 directly
```

3. None

- No networking
- Completely isolated

bash

```bash
docker run -d --network none nginx
```

4. Custom Bridge (Recommended)

- Better isolation
- Automatic DNS resolution between containers
- Can control which containers can communicate

bash

```bash
docker network create myapp-network
docker run -d --network myapp-network --name web nginx
docker run -d --network myapp-network --name api node:18
# 'web' can reach 'api' by name: http://api:3000
```

Network Commands

bash

```
# List networks
docker network ls

# Create network
docker network create mynetwork
docker network create --driver bridge mynetwork
docker network create --subnet 172.18.0.0/16 mynetwork

# Inspect network
docker network inspect mynetwork

# Connect/disconnect container
docker network connect mynetwork container_name
docker network disconnect mynetwork container_name

# Remove network
docker network rm mynetwork
docker network prune  # Remove unused networks
```

Container Communication

Same Network

bash

```bash
# Create network
docker network create app-net

# Run database
docker run -d \
  --name postgres \
  --network app-net \
  -e POSTGRES_PASSWORD=secret \
  postgres:15

# Run application (can reach postgres by name)
docker run -d \
  --name api \
  --network app-net \
  -e DATABASE_URL=postgres://postgres:secret@postgres:5432/mydb \
  myapi:latest
```

DNS Resolution

Containers on custom networks can reach each other by:

- Container name
- Network aliases

bash

```bash
docker run -d \
  --name db \
  --network app-net \
  --network-alias database \
  --network-alias db-server \
  postgres:15

# Can connect using: db, database, or db-server
```

Port Publishing

bash

```
# Publish single port
docker run -p 8080:80 nginx
# Host:Container (localhost:8080 -> container:80)

# Publish to specific IP
docker run -p 127.0.0.1:8080:80 nginx
# Only accessible on localhost

# Publish all exposed ports to random host ports
docker run -P nginx

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

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

Network Troubleshooting

bash

```bash
# See which network container is on
docker inspect container_name | grep -A 10 NetworkSettings

# Test connectivity
docker exec container1 ping container2
docker exec container1 curl http://container2:8080

# DNS lookup
docker exec container1 nslookup container2

# View network details
docker network inspect bridge
```

Docker Volumes & Storage

Why Volumes?

- Container filesystem is ephemeral (deleted with container)

- Volumes persist data outside container lifecycle
- Can share data between containers
- Better performance than bind mounts on Mac/Windows

Volume Types

1. Named Volumes (Recommended)

Managed by Docker, stored in Docker's storage directory.

bash

```
# Create volume
docker volume create mydata

# Use volume
docker run -d \
  --name postgres \
  -v mydata:/var/lib/postgresql/data \
  postgres:15

# List volumes
docker volume ls

# Inspect volume
docker volume inspect mydata

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

2. Bind Mounts

Mount host directory into container.

bash

```bash
# Absolute path required
docker run -d \
  -v /host/path:/container/path \
  nginx

# Current directory
docker run -d \
  -v $(pwd):/app \
  node:18

# Read-only mount
docker run -d \
  -v $(pwd):/app:ro \
  nginx
```

3. tmpfs Mounts

Store in host memory (not persisted).

bash

```bash
docker run -d \
  --tmpfs /tmp:rw,size=100m \
  nginx
```

Mount Syntax Comparison

bash

```
# Old style (-v)
docker run -v myvolume:/data myimage

# New style (--mount) - more explicit, recommended
docker run \
  --mount type=volume,source=myvolume,target=/data \
  myimage

docker run \
  --mount type=bind,source=$(pwd),target=/app \
  myimage

docker run \
  --mount type=tmpfs,target=/tmp,tmpfs-size=100m \
  myimage
```

Volume Commands

bash

```bash
# Create with options
docker volume create \
  --driver local \
  --opt type=none \
  --opt device=/host/path \
  --opt o=bind \
  myvolume

# Backup volume
docker run --rm \
  -v myvolume:/data \
  -v $(pwd):/backup \
  ubuntu \
  tar czf /backup/mydata.tar.gz /data

# Restore volume
docker run --rm \
  -v myvolume:/data \
  -v $(pwd):/backup \
  ubuntu \
  tar xzf /backup/mydata.tar.gz -C /

# Copy data between volumes
docker run --rm \
  -v oldvolume:/from \
  -v newvolume:/to \
  alpine \
  sh -c "cp -av /from/* /to/"
```

Storage Drivers

Different storage drivers for the container's writable layer:

- `overlay2`: Default and recommended
- `aufs`: Older systems
- `btrfs`: For btrfs filesystems
- `devicemapper`: For production on older kernels
- `zfs`: For ZFS filesystems

bash

```
# Check storage driver
docker info | grep "Storage Driver"
```

Best Practices

bash

```
# Use named volumes for databases
docker run -d \
  -v postgres_data:/var/lib/postgresql/data \
  postgres:15

# Use bind mounts for development
docker run -d \
  -v $(pwd):/app \
  node:18

# Never store data in container writable layer
# Always use volumes

# Clean up unused volumes regularly
docker volume prune
```

Docker Compose

Docker Compose lets you define multi-container applications in a YAML file.

Why Docker Compose?

- Define entire stack in one file
- One command to start everything
- Easier than remembering long docker run commands
- Version control your infrastructure

Basic docker-compose.yml

yaml

```yaml
version: '3.8'

services:
  web:
    image: nginx:alpine
    ports:
      - "8080:80"
    volumes:
      - ./html:/usr/share/nginx/html
    depends_on:
      - api
    networks:
      - frontend

  api:
    build: ./api
    environment:
      - DATABASE_URL=postgres://user:pass@db:5432/mydb
    depends_on:
      - db
    networks:
      - frontend
      - backend

  db:
    image: postgres:15
    environment:
      - POSTGRES_PASSWORD=secret
      - POSTGRES_DB=mydb
    volumes:
      - db_data:/var/lib/postgresql/data
    networks:
      - backend

volumes:
  db_data:

networks:
  frontend:
  backend:
```

Compose File Structure

Services

yaml

```yaml
services:
  myapp:
    # Use existing image
    image: nginx:alpine

    # Or build from Dockerfile
    build: .

    # Or build with context and Dockerfile name
    build:
      context: ./api
      dockerfile: Dockerfile.prod
      args:
        - NODE_ENV=production

    # Container name
    container_name: my-app

    # Restart policy
    restart: unless-stopped

    # Ports
    ports:
      - "8080:80"           # host:container
      - "127.0.0.1:8081:81" # bind to localhost

    # Environment variables
    environment:
      - NODE_ENV=production
      - API_KEY=secret

    # Or from file
    env_file:
      - .env
      - .env.prod

    # Volumes
    volumes:
      - ./code:/app                # bind mount
      - app_data:/data             # named volume
      - /etc/config:/etc/config:ro  # read-only
```

```yaml
# Networks
networks:
  - frontend
  - backend

# Dependencies
depends_on:
  - db
  - redis

# Command override
command: npm start

# Entrypoint override
entrypoint: /app/entrypoint.sh

# Working directory
working_dir: /app

# User
user: "1000:1000"

# Health check
healthcheck:
  test: ["CMD", "curl", "-f", "http://localhost"]
  interval: 30s
  timeout: 10s
  retries: 3
  start_period: 40s

# Resource limits
deploy:
  resources:
    limits:
      cpus: '0.5'
      memory: 512M
    reservations:
      cpus: '0.25'
      memory: 256M
```

Networks

yaml

```yaml
networks:
  frontend:
    driver: bridge

  backend:
    driver: bridge
    internal: true  # No external access

  custom:
    driver: bridge
    ipam:
      config:
        - subnet: 172.20.0.0/16
```

Volumes

yaml

```yaml
volumes:
  db_data:
    driver: local

  nfs_data:
    driver: local
    driver_opts:
      type: nfs
      o: addr=10.0.0.1,rw
      device: ":/path/to/dir"
```

Compose Commands

bash

```
# Start services
docker-compose up
docker-compose up -d              # Detached
docker-compose up --build         # Rebuild images
docker-compose up --force-recreate  # Recreate containers

# Stop services
docker-compose stop
docker-compose down               # Stop and remove
docker-compose down -v            # Stop and remove volumes
docker-compose down --rmi all     # Stop and remove images

# View logs
docker-compose logs
docker-compose logs -f            # Follow
docker-compose logs -f api        # Specific service

# List services
docker-compose ps

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

# Run one-off command
docker-compose run api npm test
docker-compose run --rm api npm test  # Remove after

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

# Validate compose file
docker-compose config

# Build images
docker-compose build
docker-compose build --no-cache api

# Pull images
docker-compose pull
```

```bash
# Restart services
docker-compose restart
docker-compose restart api
```

Environment Variables

.env File

bash

```bash
# .env
POSTGRES_VERSION=15
API_PORT=3000
NODE_ENV=production
```

docker-compose.yml

yaml

```yaml
services:
  db:
    image: postgres:${POSTGRES_VERSION}

  api:
    build: .
    ports:
      - "${API_PORT}:3000"
    environment:
      - NODE_ENV=${NODE_ENV}
```

Real-World Examples

Full Stack Application

yaml

```yaml
version: '3.8'

services:
  nginx:
    image: nginx:alpine
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
      - ./ssl:/etc/nginx/ssl:ro
    depends_on:
      - frontend
      - backend
    networks:
      - app-network

  frontend:
    build: ./frontend
    environment:
      - REACT_APP_API_URL=http://localhost/api
    networks:
      - app-network

  backend:
    build: ./backend
    environment:
      - DATABASE_URL=postgres://user:pass@postgres:5432/mydb
      - REDIS_URL=redis://redis:6379
    depends_on:
      - postgres
      - redis
    networks:
      - app-network

  postgres:
    image: postgres:15-alpine
    environment:
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=pass
      - POSTGRES_DB=mydb
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - app-network

  redis:
    image: redis:7-alpine
    volumes:
      - redis_data:/data
    networks:
      - app-network

volumes:
  postgres_data:
  redis_data:

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

Development with Hot Reload

yaml

```yaml
version: '3.8'

services:
  app:
    build:
      context: .
      dockerfile: Dockerfile.dev
    volumes:
      - ./src:/app/src      # Mount source code
      - /app/node_modules   # Don't override node_modules
    ports:
      - "3000:3000"
    environment:
      - NODE_ENV=development
    command: npm run dev
```

Multi-Stage Builds

Multi-stage builds let you use multiple FROM statements. Each stage can copy files from previous stages.

Why Multi-Stage Builds?

- Smaller final images (no build tools in production)
- Separate build and runtime dependencies
- Better security (fewer attack surfaces)
- Cleaner Dockerfiles

Basic Example

✓

dockerfile

```dockerfile
# Stage 1: Build
FROM node:18 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# Stage 2: Production
FROM node:18-alpine
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/node_modules ./node_modules
COPY package*.json ./
CMD ["node", "dist/server.js"]
```

Result: Final image only contains built files and runtime dependencies. Build tools stay in builder stage.

Real-World Examples

Go Application

dockerfile

```dockerfile
# Build stage
FROM golang:1.21 AS builder
WORKDIR /app
COPY go.* ./
RUN go mod download
COPY . .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .

# Production stage
FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /app/main .
EXPOSE 8080
CMD ["./main"]
```

Size Comparison:

- With build tools: ~1GB
- Multi-stage: ~15MB

React Application

dockerfile

```dockerfile
# Build stage
FROM node:18 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

# Production stage
FROM nginx:alpine
COPY --from=builder /app/build /usr/share/nginx/html
COPY nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Python with Dependencies

dockerfile

```dockerfile
# Build stage - compile dependencies
FROM python:3.11 AS builder
WORKDIR /app
COPY requirements.txt .
RUN pip install --user --no-cache-dir -r requirements.txt

# Production stage
FROM python:3.11-slim
WORKDIR /app
COPY --from=builder /root/.local /root/.local
COPY . .
ENV PATH=/root/.local/bin:$PATH
CMD ["python", "app.py"]
```

Java Application

dockerfile

```dockerfile
# Build stage
FROM maven:3.9-eclipse-temurin-17 AS builder
WORKDIR /app
COPY pom.xml .
RUN mvn dependency:go-offline
COPY src ./src
RUN mvn package -DskipTests

# Production stage
FROM eclipse-temurin:17-jre-alpine
WORKDIR /app
COPY --from=builder /app/target/*.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Multiple Build Stages

dockerfile

```dockerfile
# Dependencies stage
FROM node:18 AS dependencies
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
RUN cp -R node_modules /tmp/node_modules
RUN npm ci

# Build stage
FROM node:18 AS builder
WORKDIR /app
COPY --from=dependencies /app/node_modules ./node_modules
COPY . .
RUN npm run build
RUN npm run test

# Production stage
FROM node:18-alpine
WORKDIR /app
COPY --from=dependencies /tmp/node_modules ./node_modules
COPY --from=builder /app/dist ./dist
COPY package*.json ./
CMD ["node", "dist/server.js"]
```

Named Stages and Build Targets

dockerfile

```
FROM node:18 AS base
WORKDIR /app
COPY package*.json ./

FROM base AS development
RUN npm install
COPY . .
CMD ["npm", "run", "dev"]

FROM base AS builder
RUN npm ci
COPY . .
RUN npm run build

FROM base AS production
RUN npm ci --only=production
COPY --from=builder /app/dist ./dist
CMD ["node", "dist/server.js"]
```

Build specific target:

bash

```
# Development
docker build --target development -t myapp:dev .

# Production
docker build --target production -t myapp:prod .
```

Docker Security

Security Best Practices

1. Don't Run as Root

dockerfile

```dockerfile
# Create user
FROM node:18-alpine
RUN addgroup -g 1001 -S nodejs && \
    adduser -S nodejs -u 1001
USER nodejs
```

dockerfile

```dockerfile
# Python
FROM python:3.11-slim
RUN useradd -m -u 1000 appuser
USER appuser
```

2. Use Official Base Images

dockerfile

```dockerfile
# Good - official image
FROM node:18-alpine

# Bad - unknown source
FROM someuser/node:latest
```

3. Scan Images for Vulnerabilities

bash

```
# Docker Scout (built-in)
docker scout cves myimage:latest
docker scout recommendations myimage:latest

# Trivy
trivy image myimage:latest

# Snyk
snyk container test myimage:latest
```

4. Use Specific Tags

dockerfile

```dockerfile
# Bad
FROM node:latest

# Good
FROM node:18.17.0-alpine3.18
```

5. Minimize Attack Surface

dockerfile

```dockerfile
# Use minimal base images
FROM alpine:3.18
FROM scratch  # Empty image (for compiled binaries)

# Remove unnecessary packages
RUN apt-get update && apt-get install -y \
    curl \
    && rm -rf /var/lib/apt/lists/*
```

6. Don't Store Secrets in Images

dockerfile

```dockerfile
# Bad - secret in image
ENV API_KEY=secret123

# Good - pass at runtime
docker run -e API_KEY=secret123 myimage

# Better - use Docker secrets or external secret manager
```

7. Use Read-Only Filesystem

bash

```bash
docker run --read-only \
  --tmpfs /tmp \
  --tmpfs /run \
  myimage
```

8. Limit Resources

bash

```bash
docker run \
  --memory="512m" \
  --cpus="1.0" \
  --pids-limit=100 \
  myimage
```

9. Use Security Options

bash

```
docker run \
  --security-opt=no-new-privileges:true \
  --cap-drop=ALL \
  --cap-add=NET_BIND_SERVICE \
  myimage
```

10. Enable Content Trust

bash

```bash
# Enable Docker Content Trust
export DOCKER_CONTENT_TRUST=1

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

Docker Bench Security

Automated security audit:

bash

```bash
docker run -it --net host --pid host --userns host --cap-add audit_control \
    -e DOCKER_CONTENT_TRUST=$DOCKER_CONTENT_TRUST \
    -v /var/lib:/var/lib \
    -v /var/run/docker.sock:/var/run/docker.sock \
    -v /usr/lib/systemd:/usr/lib/systemd \
    -v /etc:/etc --label docker_bench_security \
    docker/docker-bench-security
```

Secrets Management

Docker Secrets (Swarm Mode)

bash

```
# Create secret
echo "my-secret-password" | docker secret create db_password -

# Use in service
docker service create \
  --name db \
  --secret db_password \
  postgres:15

# Access in container at /run/secrets/db_password
```

Environment Variables (Less Secure)

bash

```bash
# Better than hardcoding
docker run -e API_KEY=$(cat api-key.txt) myimage
```

External Secret Managers

- HashiCorp Vault
- AWS Secrets Manager
- Azure Key Vault
- Google Secret Manager

Docker in Production

Production Checklist

Image Optimization

dockerfile

```dockerfile
# Use minimal base images
FROM alpine:3.18

# Multi-stage builds
FROM node:18 AS builder
# ... build
FROM node:18-alpine
COPY --from=builder /app/dist ./dist

# Combine RUN commands
RUN apk add --no-cache curl && \
    rm -rf /var/cache/apk/*

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

Health Checks

dockerfile

```dockerfile
HEALTHCHECK --interval=30s --timeout=3s --retries=3 \
  CMD curl -f http://localhost:8080/health || exit 1
```

yaml

```yaml
# docker-compose.yml
services:
  api:
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080/health"]
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 40s
```

Logging

bash

```bash
# JSON file driver (default)
docker run \
  --log-driver json-file \
  --log-opt max-size=10m \
  --log-opt max-file=3 \
  myimage

# Syslog
docker run --log-driver syslog myimage

# Fluentd
docker run --log-driver fluentd myimage

# Disable logging (dangerous)
docker run --log-driver none myimage
```

Restart Policies

bash

```bash
docker run --restart=unless-stopped myimage

# docker-compose.yml
services:
  app:
    restart: unless-stopped
```

Resource Limits

yaml

```yaml
services:
  app:
    deploy:
      resources:
        limits:
          cpus: '2.0'
          memory: 1G
        reservations:
          cpus: '1.0'
          memory: 512M
```

Monitoring

View Resource Usage

bash

```bash
# All containers
docker stats

# Specific container
docker stats container_name

# No streaming
docker stats --no-stream
```

Monitoring Solutions

- **Prometheus + Grafana**: Metrics and dashboards
- **ELK Stack**: Log aggregation
- **Datadog**: Full observability platform
- **New Relic**: Application monitoring

CI/CD Integration

GitHub Actions

yaml

```yaml
name: Docker Build and Push

on:
  push:
    branches: [ main ]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v2

      - name: Login to DockerHub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      - name: Build and push
        uses: docker/build-push-action@v4
        with:
          context: .
          push: true
          tags: myuser/myapp:latest
          cache-from: type=registry,ref=myuser/myapp:buildcache
          cache-to: type=registry,ref=myuser/myapp:buildcache,mode=max
```

GitLab CI

yaml

```yaml
# .gitlab-ci.yml
build:
  image: docker:latest
  services:
    - docker:dind
  script:
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
    - docker build -t $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA .
    - docker push $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
```

Docker Registry

Set Up Private Registry

bash

```bash
# Run registry
docker run -d \
  -p 5000:5000 \
  --name registry \
  -v registry_data:/var/lib/registry \
  registry:2

# Tag image
docker tag myapp:latest localhost:5000/myapp:latest

# Push
docker push localhost:5000/myapp:latest

# Pull
docker pull localhost:5000/myapp:latest
```

Registry with Authentication

bash

```
# Create password file
htpasswd -Bc registry.password admin

# Run with auth
docker run -d \
  -p 5000:5000 \
  --name registry \
  -v $(pwd)/registry.password:/auth/registry.password \
  -e REGISTRY_AUTH=htpasswd \
  -e REGISTRY_AUTH_HTPASSWD_PATH=/auth/registry.password \
  -e REGISTRY_AUTH_HTPASSWD_REALM="Registry Realm" \
  registry:2

# Login
docker login localhost:5000
```

Advanced Topics

Docker Buildx

Enhanced build capabilities with BuildKit.

bash

```
# Create builder
docker buildx create --name mybuilder --use

# Build multi-platform images
docker buildx build \
  --platform linux/amd64,linux/arm64,linux/arm/v7 \
  -t myuser/myapp:latest \
  --push .

# Build with cache
docker buildx build \
  --cache-from type=registry,ref=myuser/myapp:buildcache \
  --cache-to type=registry,ref=myuser/myapp:buildcache,mode=max \
  -t myuser/myapp:latest \
  --push .
```

BuildKit Features

Enable BuildKit:

bash

```
export DOCKER_BUILDKIT=1
```

Secret Mounts

dockerfile

```
# syntax=docker/dockerfile:1
FROM alpine
RUN --mount=type=secret,id=mysecret \
    cat /run/secrets/mysecret
```

bash

```
docker build --secret id=mysecret,src=./secret.txt .
```

SSH Mounts

dockerfile

```dockerfile
# syntax=docker/dockerfile:1
FROM alpine
RUN apk add git
RUN --mount=type=ssh \
    git clone git@github.com:user/private-repo.git
```

bash

```bash
docker build --ssh default .
```

Cache Mounts

dockerfile

```dockerfile
# syntax=docker/dockerfile:1
FROM node:18
WORKDIR /app
COPY package*.json ./
RUN --mount=type=cache,target=/root/.npm \
    npm install
```

Docker Context

Manage multiple Docker environments.

bash

```bash
# List contexts
docker context ls

# Create context (SSH)
docker context create remote \
  --docker "host=ssh://user@remote-host"

# Create context (TCP)
docker context create remote \
  --docker "host=tcp://remote-host:2376"

# Use context
docker context use remote

# Now all commands run on remote host
docker ps

# Switch back
docker context use default
```

Docker System Commands

bash

```bash
# Disk usage
docker system df
docker system df -v  # Verbose

# Clean everything
docker system prune
docker system prune -a  # Remove all unused images
docker system prune -a --volumes  # Remove volumes too

# System info
docker system info
docker info

# System events
docker system events
docker events --since 1h
```

Docker Save/Load vs Export/Import

Save/Load (Images)

bash

```bash
# Save image with all layers and metadata
docker save myapp:latest > myapp.tar
docker save myapp:latest | gzip > myapp.tar.gz

# Load image
docker load < myapp.tar
docker load < myapp.tar.gz
```

Export/Import (Containers)

bash

```
# Export container filesystem (flattened, no history)
docker export container_name > container.tar

# Import as image
docker import container.tar myapp:imported
cat container.tar | docker import - myapp:imported
```

Docker Plugins

bash

```
# List plugins
docker plugin ls

# Install plugin
docker plugin install vieux/sshfs

# Enable/Disable
docker plugin disable vieux/sshfs
docker plugin enable vieux/sshfs

# Remove plugin
docker plugin rm vieux/sshfs
```

Troubleshooting

Container Won't Start

Check Logs

bash

```
docker logs container_name
docker logs --tail 100 container_name
docker logs --since 10m container_name
```

Check Events

bash

```bash
docker events --since 10m
```

Inspect Container

bash

```bash
docker inspect container_name
docker inspect container_name | grep -A 10 State
```

Check Exit Code

bash

```bash
docker ps -a
# Look at STATUS column
# Exit (0) = normal exit
# Exit (1) = application error
# Exit (137) = killed by SIGKILL (often OOM)
# Exit (139) = segmentation fault
```

Networking Issues

Can't Connect to Container

bash

```
# Check if container is running
docker ps

# Check port mappings
docker port container_name

# Check if process is listening inside container
docker exec container_name netstat -tlnp

# Test from host
curl localhost:8080

# Test from another container
docker run --rm --network container:mycontainer alpine wget -O- localhost:8080
```

Containers Can't Communicate

bash

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

# Check if DNS is working
docker exec container1 nslookup container2
docker exec container1 ping container2

# Check firewall rules
iptables -L -n
```

Performance Issues

High CPU Usage

bash

```
# Check stats
docker stats

# Check processes inside container
docker exec container_name top

# Limit CPU
docker update --cpus 1.5 container_name
```

High Memory Usage

bash

```bash
# Check memory usage
docker stats

# Check for memory leaks
docker exec container_name ps aux

# Limit memory
docker update --memory 512m container_name
```

Slow Build Times

bash

```bash
# Use BuildKit
export DOCKER_BUILDKIT=1

# Use build cache
docker build --cache-from myapp:latest .

# Order Dockerfile properly (stable layers first)

# Use .dockerignore to exclude unnecessary files
```

Storage Issues

Out of Disk Space

bash

```
# Check disk usage
docker system df

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

# Remove specific items
docker image prune
docker container prune
docker volume prune
docker network prune
```

Volume Issues

bash

```
# List volumes
docker volume ls

# Inspect volume
docker volume inspect volume_name

# Find which container uses volume
docker ps -a --filter volume=volume_name

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

Image Issues

Image Won't Build

bash

```bash
# Check Dockerfile syntax
docker build --no-cache .

# Build with verbose output
docker build --progress=plain .

# Check specific layer
docker build --target stage_name .
```

Can't Pull Image

bash

```bash
# Check network
ping registry.hub.docker.com

# Check auth
docker login

# Try different registry
docker pull registry.example.com/image:tag

# Check rate limits (Docker Hub)
TOKEN=$(curl "https://auth.docker.io/token?service=registry.docker.io&scope=repository:ratelimitpreview/test:pull" | jq -
curl --head -H "Authorization: Bearer $TOKEN" https://registry-1.docker.io/v2/ratelimitpreview/test/manifests/latest
```

Debug Running Container

bash

```
# Get shell
docker exec -it container_name bash
docker exec -it container_name sh  # Alpine

# Check environment variables
docker exec container_name env

# Check filesystem
docker exec container_name ls -la /app

# Check running processes
docker exec container_name ps aux

# Check network connections
docker exec container_name netstat -tlnp

# Copy files out
docker cp container_name:/app/logs/error.log ./

# View real-time logs
docker logs -f container_name
```

Common Error Messages

"Container already exists"

bash

```
docker rm container_name
# or
docker rm -f container_name  # Force remove
```

"Port already allocated"

bash

```
# Find what's using the port
sudo lsof -i :8080
# or
sudo netstat -tlnp | grep 8080

# Use different port
docker run -p 8081:80 nginx
```

"No space left on device"

bash

```
docker system prune -a --volumes
```

"Cannot connect to Docker daemon"

bash

```
# Start Docker
sudo systemctl start docker

# Check if running
sudo systemctl status docker

# Add user to docker group
sudo usermod -aG docker $USER
# Log out and back in
```

"denied: access forbidden"

bash

```
# Login to registry
docker login

# Check credentials
cat ~/.docker/config.json
```

Interview Questions & Answers

Fundamentals

Q: What is Docker and why do we use it? A: Docker is a containerization platform that packages applications with their dependencies. We use it for consistency across environments, faster deployments, better resource utilization than VMs, easier scaling, and simplified dependency management. It solves the "works on my machine" problem.

Q: Difference between Docker image and container? A: An image is a read-only template (like a class in programming) containing the application and dependencies. A container is a running instance of an image (like an object). Images are built once, containers are created from images and can be started/stopped/deleted.

Q: What is a Dockerfile? A: A Dockerfile is a text file containing instructions to build a Docker image. It specifies the base image, copies files, installs dependencies, sets environment variables, and defines the command to run. It's like a recipe for creating an image.

Q: Difference between CMD and ENTRYPOINT? A: CMD provides default arguments that can be overridden when running the container. ENTRYPOINT defines the main command that always runs. Best practice is to use both: ENTRYPOINT for the main command, CMD for default arguments.

dockerfile

```dockerfile
ENTRYPOINT ["python", "app.py"]
CMD ["--mode", "production"]
# docker run myimage -> python app.py --mode production
# docker run myimage --mode dev -> python app.py --mode dev
```

Q: Difference between COPY and ADD? A: COPY simply copies files from host to image. ADD does the same but also supports URLs and auto-extracts tar archives. Use COPY for simple file copying (preferred), use ADD only when you need its extra features.

Q: What is a Docker registry? A: A Docker registry is a storage system for Docker images. Docker Hub is the default public registry. Private registries include AWS ECR, Google GCR, Azure ACR, and self-hosted solutions like Harbor. You push images to registries and pull them when deploying.

Intermediate

Q: Explain Docker networking modes. A:

- Bridge (default): Isolated network, containers communicate via IP, need port publishing for host access

- Host: Container shares host network, no isolation, better performance
- None: No networking, completely isolated
- Custom bridge: User-defined network with DNS resolution, better isolation and control

Q: What are Docker volumes and why use them? A: Volumes persist data outside container lifecycle. Three types: named volumes (managed by Docker), bind mounts (host directory), tmpfs (memory). Use volumes for databases, user uploads, logs, or any data that should survive container restarts/deletions.

Q: What is Docker Compose? A: Docker Compose is a tool for defining multi-container applications in YAML. Instead of running multiple docker run commands, you define services, networks, and volumes in docker-compose.yml and manage everything with single commands like docker-compose up.

Q: Explain multi-stage builds. A: Multi-stage builds use multiple FROM statements in one Dockerfile. You build in one stage (with build tools) and copy artifacts to a minimal final stage. This keeps images small and secure by excluding build dependencies from production images.

dockerfile

```dockerfile
FROM node:18 AS builder
WORKDIR /app
COPY . .
RUN npm install && npm run build

FROM node:18-alpine
COPY --from=builder /app/dist ./dist
CMD ["node", "dist/server.js"]
```

Q: How do containers communicate? A: Containers on the same custom network can reach each other by container name through Docker's embedded DNS server. Use docker network create to create custom networks. For external access, publish ports with -p flag.

Q: What is the difference between docker stop and docker kill? A: docker stop sends SIGTERM (graceful shutdown, allows cleanup) then SIGKILL after timeout. docker kill sends SIGKILL immediately (force termination). Always use stop in production to allow proper shutdown.

Advanced

Q: How do you optimize Docker images? A:

1. Use minimal base images (alpine, scratch)
2. Use multi-stage builds
3. Order Dockerfile layers properly (stable first, changing last)
4. Combine RUN commands to reduce layers
5. Use .dockerignore
6. Remove package manager caches
7. Use specific image tags, not latest

Q: Docker security best practices? A:

1. Don't run as root (USER instruction)

2. Use official base images
3. Scan images for vulnerabilities
4. Don't store secrets in images
5. Use read-only filesystems where possible
6. Limit container resources
7. Keep images updated
8. Use minimal base images
9. Enable Docker Content Trust

Q: How do you handle secrets in Docker? A: Never hardcode secrets in Dockerfiles or images. Options:

1. Docker secrets (Swarm mode)
2. Environment variables at runtime
3. External secret managers (Vault, AWS Secrets Manager)
4. Mount secrets as files at runtime
5. Use BuildKit secret mounts for build-time secrets

Q: Explain Docker layer caching. A: Each Dockerfile instruction creates a layer. Docker caches layers and reuses them if the instruction and its context haven't changed. If a layer changes, all subsequent layers are invalidated. Order instructions from least to most frequently changing for optimal caching.

Q: What is BuildKit? A: BuildKit is Docker's improved build backend with better performance, caching, and features. It supports parallel builds, build secrets, SSH mounts, cache mounts, and multi-platform builds. Enable with DOCKER_BUILDKIT=1.

Q: How would you debug a failing container? A:

1. Check logs: docker logs container_name
2. Check exit code: docker ps -a
3. Inspect container: docker inspect container_name
4. Get shell if possible: docker exec -it container_name bash
5. Check resource usage: docker stats
6. Review events: docker events
7. Test network connectivity
8. Check volume mounts

Q: Difference between docker-compose down and docker-compose stop? A: stop stops containers but keeps them. down stops AND removes containers, networks, and optionally volumes (with -v). Use stop for temporary pause, down for complete teardown.

Q: How do you limit container resources? A:

bash

```bash
docker run \
  --memory="512m" \
  --memory-swap="1g" \
  --cpus="1.5" \
  --pids-limit=100 \
  myimage
```

Or in Compose:

yaml

```yaml
services:
  app:
    deploy:
      resources:
        limits:
          cpus: '1.5'
          memory: 512M
```

Q: What happens when you run docker build? A:

1. Docker sends build context to daemon
2. Daemon reads Dockerfile
3. For each instruction, creates intermediate container
4. Runs instruction in container
5. Commits container to new layer
6. Removes intermediate container
7. Repeats for all instructions
8. Final layer becomes the image

Q: How do you implement health checks? A: In Dockerfile:

dockerfile

```dockerfile
HEALTHCHECK --interval=30s --timeout=3s --retries=3 \
  CMD curl -f http://localhost/health || exit 1
```

Or in docker run:

bash

```bash
docker run --health-cmd="curl -f http://localhost/health" myimage
```

Q: What is the Docker storage driver? A: Storage driver manages the container's writable layer. overlay2 is the default and recommended. It uses union filesystem to layer images efficiently. Other drivers: aufs, btrfs, devicemapper, zfs.

Q: Explain Docker context. A: Docker context lets you manage multiple Docker environments (local, remote, different clusters) and switch between them. Useful for managing development, staging, and production environments.

bash

```
docker context create remote --docker "host=ssh://user@server"
docker context use remote
docker ps  # Now running on remote host
```

Q: How would you deploy Docker in production? A:

1. Use orchestration (Kubernetes, Docker Swarm)
2. Implement health checks
3. Set restart policies
4. Configure logging
5. Limit resources
6. Use specific image tags
7. Implement monitoring
8. Set up CI/CD pipelines
9. Use secrets management
10. Regular security scans

Q: What is the difference between Union FS and Volume? A: Union FS (like overlay2) is how Docker layers images and container writable layers. It's copy-on-write and ephemeral. Volumes are for persistent data, stored outside the union filesystem, and survive container deletion.

Quick Reference Commands

Image Commands

bash

```
docker images                    # List images
docker build -t name:tag .       # Build image
docker pull image:tag            # Download image
docker push image:tag            # Upload image
docker rmi image:tag             # Remove image
docker image prune -a            # Remove unused images
docker history image:tag         # Show image layers
docker inspect image:tag         # Image details
docker save image > file.tar     # Export image
docker load < file.tar           # Import image
```

Container Commands

bash

```bash
docker ps                    # Running containers
docker ps -a                 # All containers
docker run -d -p 8080:80 nginx  # Run container
docker start container_name     # Start container
docker stop container_name      # Stop container
docker restart container_name   # Restart container
docker rm container_name        # Remove container
docker rm -f container_name     # Force remove
docker exec -it container bash  # Execute command
docker logs -f container_name   # View logs
docker inspect container_name   # Container details
docker stats                 # Resource usage
docker cp file container:/path  # Copy files
```

Network Commands

bash

```bash
docker network ls                # List networks
docker network create name       # Create network
docker network connect net cont  # Connect container
docker network disconnect net cont # Disconnect container
docker network inspect name      # Network details
docker network rm name           # Remove network
docker network prune             # Remove unused networks
```

Volume Commands

bash

```
docker volume ls          # List volumes
docker volume create name      # Create volume
docker volume inspect name      # Volume details
docker volume rm name         # Remove volume
docker volume prune          # Remove unused volumes
```

Docker Compose Commands

bash

```
docker-compose up           # Start services
docker-compose up -d          # Start detached
docker-compose down          # Stop and remove
docker-compose down -v         # Stop and remove volumes
docker-compose ps           # List services
docker-compose logs -f         # View logs
docker-compose exec service bash  # Execute command
docker-compose build          # Build images
docker-compose restart         # Restart services
docker-compose stop          # Stop services
```

System Commands

bash

```
docker system df          # Disk usage
docker system prune         # Remove unused data
docker system prune -a        # Remove all unused data
docker system info          # System information
docker events            # System events
docker version           # Docker version
```

Best Practices Summary

Dockerfile

- Use official base images

- Use specific tags, not latest
- Order instructions from least to most frequently changing
- Combine RUN commands to reduce layers
- Use multi-stage builds
- Don't run as root
- Use .dockerignore
- Clean up in same RUN command

Security

- Don't store secrets in images
- Scan images regularly
- Use minimal base images
- Run as non-root user
- Keep images updated
- Limit container resources
- Use read-only filesystems where possible

Performance

- Use layer caching effectively
- Keep images small
- Use volumes for data
- Optimize for network I/O
- Use multi-stage builds
- Consider build cache strategies

Production

- Always use health checks
- Set restart policies
- Configure logging properly
- Monitor resource usage
- Use orchestration (Kubernetes/Swarm)
- Implement CI/CD
- Use specific image tags
- Regular backups of volumes

Common Patterns

Database Container

bash

```
docker run -d \
  --name postgres \
  -e POSTGRES_PASSWORD=secret \
  -e POSTGRES_DB=mydb \
  -v postgres_data:/var/lib/postgresql/data \
  -p 5432:5432 \
  --restart unless-stopped \
  postgres:15-alpine
```

Application with Database

yaml

```yaml
version: '3.8'
services:
  app:
    build: .
    ports:
      - "3000:3000"
    environment:
      - DATABASE_URL=postgres://user:pass@db:5432/mydb
    depends_on:
      - db

  db:
    image: postgres:15-alpine
    environment:
      - POSTGRES_PASSWORD=pass
      - POSTGRES_USER=user
      - POSTGRES_DB=mydb
    volumes:
      - db_data:/var/lib/postgresql/data

volumes:
  db_data:
```

Development Environment

yaml

```yaml
version: '3.8'
services:
  app:
    build:
      context: .
      dockerfile: Dockerfile.dev
    volumes:
      - ./src:/app/src
      - /app/node_modules
    ports:
      - "3000:3000"
    environment:
      - NODE_ENV=development
    command: npm run dev
```

Docker Swarm (Orchestration Basics)

Docker Swarm is Docker's native orchestration tool for managing clusters of Docker hosts.

Swarm Concepts

- **Node**: A Docker host in the swarm (manager or worker)
- **Manager**: Orchestrates and schedules services
- **Worker**: Executes containers
- **Service**: Definition of tasks to run on nodes
- **Task**: A single container running on a node
- **Stack**: Group of services defined in a Compose file

Basic Swarm Commands

bash

```
# Initialize swarm
docker swarm init

# Join swarm as worker
docker swarm join --token TOKEN manager-ip:2377

# List nodes
docker node ls

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

# List services
docker service ls

# Scale service
docker service scale web=5

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

# Remove service
docker service rm web

# Leave swarm
docker swarm leave
```

Deploy Stack with Compose File

yaml

```yaml
# docker-stack.yml
version: '3.8'

services:
  web:
    image: nginx:alpine
    ports:
      - "80:80"
    deploy:
      replicas: 3
      update_config:
        parallelism: 1
        delay: 10s
      restart_policy:
        condition: on-failure
      placement:
        constraints:
          - node.role == worker

  api:
    image: myapi:latest
    deploy:
      replicas: 2
      resources:
        limits:
          cpus: '0.5'
          memory: 512M
        reservations:
          cpus: '0.25'
          memory: 256M
```

bash

```
# Deploy stack
docker stack deploy -c docker-stack.yml myapp

# List stacks
docker stack ls

# List services in stack
docker stack services myapp

# Remove stack
docker stack rm myapp
```

Docker Performance Optimization

Build Performance

1. Use BuildKit

bash

```bash
export DOCKER_BUILDKIT=1
docker build .
```

2. Effective Layer Caching

dockerfile

```dockerfile
# Bad - cache invalidated on any code change
COPY . /app
RUN npm install

# Good - dependencies cached separately
COPY package*.json /app/
RUN npm install
COPY . /app
```

3. Parallel Builds

dockerfile

```dockerfile
# syntax=docker/dockerfile:1
FROM node:18 AS deps
COPY package*.json ./
RUN npm install

FROM node:18 AS build
COPY --from=deps /node_modules ./node_modules
COPY . .
RUN npm run build

# Both stages can build in parallel
```

4. Cache Mounts

dockerfile

```dockerfile
# syntax=docker/dockerfile:1
FROM node:18
RUN --mount=type=cache,target=/root/.npm \
    npm install
```

Runtime Performance

1. Resource Limits

bash

```bash
docker run \
  --memory="512m" \
  --memory-reservation="256m" \
  --cpus="1.5" \
  --cpu-shares=1024 \
  myimage
```

2. Use Alpine Images

dockerfile

```dockerfile
# Standard: ~900MB
FROM node:18

# Alpine: ~150MB
FROM node:18-alpine
```

3. Minimize Layers

dockerfile

```dockerfile
# Bad - 3 layers
RUN apt-get update
RUN apt-get install curl
RUN apt-get install git

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

4. Use .dockerignore

```
node_modules
.git
.gitignore
README.md
.env
.env.local
dist
build
coverage
.DS_Store
*.log
```

Network Performance

1. Use Host Network for Performance

bash

```bash
# When you need maximum network performance
docker run --network host myimage
```

2. Optimize DNS

bash

```bash
docker run --dns 8.8.8.8 --dns 8.8.4.4 myimage
```

Storage Performance

1. Use Volumes (Not Bind Mounts)

bash

```
# Better performance
docker run -v myvolume:/data myimage

# Slower on Mac/Windows
docker run -v $(pwd):/data myimage
```

2. Use tmpfs for Temporary Data

bash

```bash
docker run --tmpfs /tmp:rw,size=100m myimage
```

Docker Anti-Patterns (What NOT to Do)

1. Running as Root

dockerfile

```dockerfile
# BAD
FROM ubuntu
COPY app /app
CMD ["/app"]

# GOOD
FROM ubuntu
RUN useradd -m appuser
USER appuser
COPY app /app
CMD ["/app"]
```

2. Using latest Tag

dockerfile

```
# BAD
FROM node:latest

# GOOD
FROM node:18.17-alpine
```

3. Storing Data in Containers

bash

```bash
# BAD - data lost when container removed
docker run -d postgres

# GOOD
docker run -d -v postgres_data:/var/lib/postgresql/data postgres
```

4. Installing Unnecessary Packages

dockerfile

```dockerfile
# BAD
RUN apt-get update && apt-get install -y \
    curl \
    wget \
    vim \
    nano \
    build-essential

# GOOD - only what you need
RUN apt-get update && apt-get install -y curl \
    && rm -rf /var/lib/apt/lists/*
```

5. Not Using .dockerignore

Without .dockerignore, you send node_modules, .git, etc. to build context.

6. Multiple Processes in One Container

dockerfile

```dockerfile
# BAD
CMD service nginx start && service mysql start

# GOOD - use docker-compose for multiple services
```

7. Hardcoding Configuration

dockerfile

```dockerfile
# BAD
ENV DATABASE_HOST=192.168.1.100
ENV API_KEY=secret123

# GOOD - pass at runtime
docker run -e DATABASE_HOST=db -e API_KEY=$API_KEY myimage
```

8. Not Using Health Checks

dockerfile

```dockerfile
# BAD - no health check
FROM nginx

# GOOD
FROM nginx
HEALTHCHECK CMD curl -f http://localhost || exit 1
```

9. Ignoring Logs

bash

```
# BAD - logs fill disk
docker run -d myapp

# GOOD
docker run -d \
  --log-opt max-size=10m \
  --log-opt max-file=3 \
  myapp
```

10. Not Cleaning Up

dockerfile

```
# BAD
RUN apt-get update
RUN apt-get install -y curl
RUN wget https://example.com/file.tar.gz
RUN tar xzf file.tar.gz

# GOOD
RUN apt-get update && \
    apt-get install -y curl && \
    wget https://example.com/file.tar.gz && \
    tar xzf file.tar.gz && \
    rm file.tar.gz && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*
```

Real-World Scenarios

Scenario 1: Microservices Architecture

yaml

```yaml
version: '3.8'

services:
  nginx:
    image: nginx:alpine
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
      - ./ssl:/etc/nginx/ssl:ro
    depends_on:
      - frontend
      - auth-service
      - user-service
      - product-service
    networks:
      - frontend

  frontend:
    build: ./frontend
    environment:
      - API_URL=http://api-gateway:3000
    networks:
      - frontend

  api-gateway:
    build: ./api-gateway
    environment:
      - AUTH_SERVICE=http://auth-service:3001
      - USER_SERVICE=http://user-service:3002
      - PRODUCT_SERVICE=http://product-service:3003
    networks:
      - frontend
      - backend

  auth-service:
    build: ./services/auth
    environment:
      - DATABASE_URL=postgres://user:pass@postgres:5432/auth
      - REDIS_URL=redis://redis:6379
    depends_on:
      - postgres
      - redis
    networks:
      - backend

  user-service:
    build: ./services/user
    environment:
      - DATABASE_URL=postgres://user:pass@postgres:5432/users
    depends_on:
      - postgres
    networks:
      - backend

  product-service:
    build: ./services/product
    environment:
      - DATABASE_URL=postgres://user:pass@postgres:5432/products
      - ELASTICSEARCH_URL=http://elasticsearch:9200
    depends_on:
      - postgres
      - elasticsearch
    networks:
      - backend

  postgres:
    image: postgres:15-alpine
    environment:
      - POSTGRES_PASSWORD=secret
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - backend

  redis:
    image: redis:7-alpine
    volumes:
      - redis_data:/data
    networks:
      - backend
  elasticsearch:
    image: elasticsearch:8.10.0
    environment:
      - discovery.type=single-node
      - ES_JAVA_OPTS=-Xms512m -Xmx512m
    volumes:
      - es_data:/usr/share/elasticsearch/data
    networks:
      - backend

volumes:
  postgres_data:
  redis_data:
  es_data:

networks:
  frontend:
  backend:
```

Scenario 2: CI/CD Pipeline with Testing

yaml

```yaml
# docker-compose.test.yml
version: '3.8'

services:
  app:
    build:
      context: .
      target: test
    environment:
      - NODE_ENV=test
      - DATABASE_URL=postgres://test:test@test-db:5432/testdb
    depends_on:
      - test-db
    command: npm test

  test-db:
    image: postgres:15-alpine
    environment:
      - POSTGRES_USER=test
      - POSTGRES_PASSWORD=test
      - POSTGRES_DB=testdb
    tmpfs:
      - /var/lib/postgresql/data
```

dockerfile

```
# Multi-stage with test stage
FROM node:18-alpine AS base
WORKDIR /app
COPY package*.json ./

FROM base AS development
RUN npm install
COPY . .
CMD ["npm", "run", "dev"]

FROM base AS test
RUN npm install
COPY . .
CMD ["npm", "test"]

FROM base AS builder
RUN npm ci --only=production
COPY . .
RUN npm run build

FROM node:18-alpine AS production
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/node_modules ./node_modules
COPY package*.json ./
USER node
CMD ["node", "dist/server.js"]
```

Scenario 3: Development Environment with Hot Reload

yaml

```yaml
version: '3.8'

services:
  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile.dev
    volumes:
      - ./frontend/src:/app/src
      - /app/node_modules
    ports:
      - "3000:3000"
    environment:
      - CHOKIDAR_USEPOLLING=true
      - REACT_APP_API_URL=http://localhost:8000
    command: npm start

  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile.dev
    volumes:
      - ./backend:/app
      - /app/node_modules
    ports:
      - "8000:8000"
    environment:
      - NODE_ENV=development
      - DATABASE_URL=postgres://dev:dev@postgres:5432/devdb
    depends_on:
      - postgres
    command: npm run dev

  postgres:
    image: postgres:15-alpine
    environment:
      - POSTGRES_USER=dev
      - POSTGRES_PASSWORD=dev
      - POSTGRES_DB=devdb
    ports:
      - "5432:5432"
    volumes:
      - postgres_dev:/var/lib/postgresql/data

volumes:
  postgres_dev:
```

Scenario 4: Production with Monitoring

yaml

```yaml
version: '3.8'

services:
  app:
    image: myapp:${VERSION}
    deploy:
      replicas: 3
      restart_policy:
        condition: on-failure
        max_attempts: 3
      resources:
        limits:
          cpus: '1'
          memory: 1G
        reservations:
          cpus: '0.5'
          memory: 512M
    environment:
      - NODE_ENV=production
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:3000/health"]
      interval: 30s
      timeout: 10s
      retries: 3
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "3"
    networks:
      - app-network

  prometheus:
    image: prom/prometheus
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
      - prometheus_data:/prometheus
    ports:
      - "9090:9090"
    networks:
      - app-network
  grafana:
    image: grafana/grafana
    ports:
      - "3001:3000"
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin
    volumes:
      - grafana_data:/var/lib/grafana
    networks:
      - app-network

  node-exporter:
    image: prom/node-exporter
    ports:
      - "9100:9100"
    networks:
      - app-network

volumes:
  prometheus_data:
  grafana_data:

networks:
  app-network:
```

Docker Tips & Tricks

1. Quick Container Inspection

bash

```bash
# Get container IP
docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' container_name

# Get container ports
docker inspect -f '{{range $p, $conf := .NetworkSettings.Ports}}{{$p}} -> {{(index $conf 0).HostPort}}{{end}}' contain

# Get environment variables
docker inspect -f '{{range .Config.Env}}{{println .}}{{end}}' container_name
```

2. Clean Up Everything

bash

```bash
# Nuclear option - remove everything
docker system prune -a --volumes -f

# Remove all stopped containers
docker container prune -f

# Remove all dangling images
docker image prune -f

# Remove all unused volumes
docker volume prune -f
```

3. Run Command in All Containers

bash

```
# Restart all containers
docker ps -q | xargs docker restart

# Stop all containers
docker ps -q | xargs docker stop

# Remove all stopped containers
docker ps -aq | xargs docker rm
```

4. Copy Files Between Containers

bash

```
# Container to container
docker cp container1:/path/file.txt /tmp/
docker cp /tmp/file.txt container2:/path/

# One-liner
docker exec container1 cat /path/file.txt | docker exec -i container2 sh -c 'cat > /path/file.txt'
```

5. Monitor Multiple Containers

bash

```
# Follow logs from multiple containers
docker-compose logs -f service1 service2

# Stats for specific containers
docker stats container1 container2

# Watch container status
watch 'docker ps --format "table {{.Names}}\t{{.Status}}\t{{.Ports}}"'
```

6. Quick Database Backup

bash

```
# PostgreSQL
docker exec postgres pg_dump -U user dbname > backup.sql

# MySQL
docker exec mysql mysqldump -u user -ppassword dbname > backup.sql

# MongoDB
docker exec mongo mongodump --out /tmp/backup
docker cp mongo:/tmp/backup ./backup
```

7. Interactive Debugging

📋 ✓

bash

```
# Run ephemeral debug container
docker run --rm -it \
  --network container:target_container \
  --pid container:target_container \
  nicolaka/netshoot

# Attach debugger to running container
docker run --rm -it \
  --cap-add=SYS_PTRACE \
  --pid=container:target \
  alpine gdb -p 1
```

8. Test DNS Resolution

📋 ✓

bash

```
# Check DNS from container
docker run --rm alpine nslookup google.com

# Check DNS in specific network
docker run --rm --network mynet alpine nslookup service_name
```

9. Build Context Tricks

bash

```bash
# Build from URL
docker build github.com/user/repo

# Build from tar
docker build - < context.tar.gz

# Build with stdin Dockerfile
docker build -t myimage:latest - < Dockerfile
```

10. Container Forensics

bash

```bash
# Export container filesystem
docker export container_name > filesystem.tar

# Commit container to image (for debugging)
docker commit container_name debug-image

# Run container with different command
docker run --rm -it --entrypoint bash myimage
```

Docker Cheat Sheet

Essential Commands

bash

```
# Images
docker images                      # List
docker build -t name:tag .         # Build
docker pull name:tag               # Download
docker push name:tag               # Upload
docker rmi name:tag                # Remove

# Containers
docker ps                          # List running
docker ps -a                       # List all
docker run -d -p 8080:80 nginx     # Run
docker start/stop/restart name     # Control
docker rm name                     # Remove
docker exec -it name bash          # Execute
docker logs -f name                # Logs

# Networks
docker network ls                  # List
docker network create name         # Create
docker network connect net cont    # Connect
docker network inspect name        # Inspect

# Volumes
docker volume ls                   # List
docker volume create name          # Create
docker volume inspect name         # Inspect
docker volume rm name              # Remove

# System
docker system df                   # Disk usage
docker system prune                # Clean up
docker info                        # System info
docker version                     # Version

# Compose
docker-compose up -d               # Start
docker-compose down                # Stop
docker-compose logs -f             # Logs
```

```
docker-compose ps                 # Status
docker-compose exec service bash      # Execute
```

Common Dockerfile Instructions

dockerfile

```
FROM image:tag                  # Base image
WORKDIR /app                   # Working directory
COPY src dest                  # Copy files
RUN command                    # Execute command
ENV KEY=value                  # Environment variable
EXPOSE 8080                    # Document port
VOLUME /data                   # Mount point
USER username                  # Switch user
CMD ["executable"]             # Default command
ENTRYPOINT ["executable"]         # Main command
HEALTHCHECK CMD command             # Health check
```

Docker Run Options

bash

```
-d                    # Detached mode
-it                   # Interactive with TTY
-p 8080:80            # Port mapping
-e KEY=value          # Environment variable
-v /host:/container  # Volume mount
--name myapp          # Container name
--network mynet       # Custom network
--restart always      # Restart policy
-m 512m               # Memory limit
--cpus 1.5            # CPU limit
--rm                  # Remove after exit
-w /app               # Working directory
--user 1000:1000      # Run as user
```

Summary

You now have complete Docker mastery. Here's what you've learned:

Core Concepts

1. **Docker Basics**: Containers, images, Dockerfile, registry
2. **Difference from VMs**: Lighter, faster, shares kernel
3. **Layer System**: Caching, optimization, multi-stage builds

Practical Skills

4. **Image Building**: Efficient Dockerfiles, optimization techniques
5. **Container Management**: Run, stop, debug, monitor
6. **Networking**: Bridge, host, custom networks, DNS resolution
7. **Storage**: Volumes, bind mounts, data persistence
8. **Docker Compose**: Multi-container applications

Advanced Topics

9. **Security**: Run as non-root, scan images, secrets management
10. **Performance**: Caching, alpine images, resource limits
11. **Production**: Health checks, logging, monitoring, CI/CD
12. **Orchestration**: Docker Swarm basics
13. **Troubleshooting**: Debug techniques, common issues

Best Practices

14. Use specific tags, not latest
15. Order Dockerfile for optimal caching
16. Don't run as root
17. Use multi-stage builds
18. Keep images small
19. Never store secrets in images
20. Always use .dockerignore
21. Implement health checks
22. Clean up regularly

Interview Ready

You can now confidently answer questions about:

- Docker architecture and how it works
- Differences between images and containers
- Networking and storage strategies
- Security best practices
- Production deployment patterns
- Performance optimization
- Troubleshooting techniques

Next Steps

- Practice building real applications

- Learn Kubernetes (container orchestration at scale)
- Explore CI/CD pipelines with Docker
- Deep dive into security scanning
- Experiment with different deployment strategies

You're now a Docker expert! Go build amazing things!