Docker: A Complete Guide

Containerization Technology Explained

Ines Tmimi

Comprehensive Technical Documentation

November 17, 2025

Contents

1 Introduction to Docker

1.1 What is Docker?

Docker is an open-source platform that enables developers to automate the deployment, scaling, and management of applications using containerization technology. It packages applications and their dependencies into standardized units called containers that can run consistently across different computing environments.

Key Concept

Container vs Virtual Machine: Containers share the host OS kernel and isolate application processes, making them lightweight and fast. Virtual machines include a full OS copy, making them heavier and slower to start.

1.2 Why Use Docker?

- **Consistency:** "It works on my machine" problem solved - containers run identically everywhere
- **Isolation:** Applications run in isolated environments without conflicts
- **Portability:** Move containers between development, testing, and production seamlessly
- **Efficiency:** Containers start in seconds and use minimal resources
- **Scalability:** Easy horizontal scaling for microservices architecture
- **Version Control:** Track changes to application environments

1.3 Docker Architecture

Docker uses a client-server architecture with the following components:

Component	Description
Docker Client	Command-line interface (CLI) that users interact with
Docker Daemon	Background service (dockerd) that manages containers, images, networks, and volumes
Docker Registry	Repository for storing and distributing Docker images (e.g., Docker Hub)
Docker Images	Read-only templates containing application code and dependencies
Docker Containers	Running instances of Docker images
Docker Volumes	Persistent data storage mechanism
Docker Networks	Virtual networks for container communication

Table 1: Docker Architecture Components

2 Installation

2.1 Installing Docker on Linux (Ubuntu/Debian)

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

# Install prerequisites
sudo apt-get install ca-certificates curl gnupg lsb-release

# Add Docker's official GPG key
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
  sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

# Set up repository
echo \
  "deb␣[arch=$(dpkg␣--print-architecture)␣\
␣␣signed-by=/etc/apt/keyrings/docker.gpg]␣\
␣␣https://download.docker.com/linux/ubuntu␣\
␣␣$(lsb_release␣-cs)␣stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

# Install Docker Engine
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io \
  docker-compose-plugin

# Verify installation
sudo docker run hello-world
```

2.2 Post-Installation Steps

```
# Add user to docker group (avoid using sudo)
sudo usermod -aG docker $USER

# Apply group changes (or logout/login)
newgrp docker

# Configure Docker to start on boot
sudo systemctl enable docker.service
sudo systemctl enable containerd.service

# Verify Docker version
docker --version
docker compose version
```

3 Docker Images

3.1 Understanding Docker Images

A Docker image is a read-only template that contains:

- Application code
- Runtime environment
- Libraries and dependencies
- Environment variables
- Configuration files

Images are built in layers, where each layer represents a set of file system changes.

3.2 Image Commands

Command	Description
`docker images`	List all local images
`docker pull <image>`	Download image from registry
`docker push <image>`	Upload image to registry
`docker rmi <image>`	Remove image
`docker image prune`	Remove unused images
`docker build -t <name> .`	Build image from Dockerfile
`docker tag <source> <target>`	Create tag for image
`docker history <image>`	Show image layer history
`docker inspect <image>`	Display detailed image information
`docker save -o <file> <image>`	Export image to tar archive
`docker load -i <file>`	Import image from tar archive

Table 2: Docker Image Commands

3.3 Working with Images - Detailed Examples

3.3.1 Pulling Images

```
# Pull latest version of Ubuntu
docker pull ubuntu:latest

# Pull specific version
docker pull ubuntu:20.04

# Pull from specific registry
docker pull myregistry.com/myapp:v1.0

# List all downloaded images
docker images
```

```
# List images with specific filter
docker images --filter "dangling=false"
```

3.3.2 Inspecting Images

```
# View detailed information
docker inspect ubuntu:latest

# View specific information using format
docker inspect --format='{{.Architecture}}' ubuntu:latest

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

# View image size
docker images --format "table␣{{.Repository}}\t{{.Tag}}\t{{.Size}}"
```

4 Docker Containers

4.1 Container Lifecycle

1. **Created:** Container is created but not started
2. **Running:** Container is actively executing
3. **Paused:** Container processes are suspended
4. **Stopped:** Container has exited
5. **Removed:** Container is deleted

4.2 Container Commands Reference

Command	Description
`docker run <image>`	Create and start container
`docker start <container>`	Start stopped container
`docker stop <container>`	Stop running container gracefully
`docker restart <container>`	Restart container
`docker pause <container>`	Pause container processes
`docker unpause <container>`	Resume paused container
`docker kill <container>`	Force stop container
`docker rm <container>`	Remove stopped container
`docker ps`	List running containers
`docker ps -a`	List all containers
`docker logs <container>`	View container logs
`docker exec -it <container> <cmd>`	Execute command in running container
`docker attach <container>`	Attach to running container
`docker cp <src> <dest>`	Copy files between container and host
`docker stats`	Display resource usage statistics
`docker top <container>`	Display running processes
`docker port <container>`	Display port mappings

Table 3: Essential Container Commands

4.3 Running Containers - Detailed Examples

4.3.1 Basic Container Operations

```
# Run container in foreground
docker run ubuntu:latest echo "Hello␣Docker"

# Run container in background (detached mode)
docker run -d nginx:latest

# Run container with custom name
docker run --name my-nginx -d nginx:latest
```

```
# Run interactive container with terminal
docker run -it ubuntu:latest /bin/bash

# Run container with automatic removal after exit
docker run --rm ubuntu:latest ls /
```

4.3.2 Port Mapping

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

# Map all exposed ports to random host ports
docker run -d -P nginx:latest

# Map to specific host IP
docker run -d -p 127.0.0.1:8080:80 nginx:latest

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

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

4.3.3 Volume Mounting

```
# Mount host directory to container
docker run -d -v /host/path:/container/path nginx:latest

# Mount with read-only access
docker run -d -v /host/path:/container/path:ro nginx:latest

# Create and use named volume
docker run -d -v my-volume:/data nginx:latest

# Mount current directory
docker run -d -v $(pwd):/app nginx:latest
```

4.3.4 Environment Variables

```
# Set single environment variable
docker run -e MYVAR=value ubuntu:latest env

# Set multiple environment variables
docker run -e VAR1=value1 -e VAR2=value2 ubuntu:latest

# Load environment variables from file
docker run --env-file ./env.list ubuntu:latest
```

```
# Example env.list file content:
# VAR1=value1
# VAR2=value2
# DATABASE_URL=postgresql://user:pass@host:5432/db
```

4.3.5 Resource Constraints

```
# Limit memory to 512MB
docker run -m 512m nginx:latest

# Limit CPU shares (relative weight)
docker run --cpus=1.5 nginx:latest

# Limit specific CPU cores
docker run --cpuset-cpus="0,1" nginx:latest

# Combine multiple constraints
docker run -m 1g --cpus=2 --name constrained-app nginx:latest
```

4.4 Container Management

```
# List running containers
docker ps

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

# Filter containers
docker ps --filter "status=running"
docker ps --filter "name=nginx"

# View container logs
docker logs my-container

# Follow logs in real-time
docker logs -f my-container

# View last 100 lines
docker logs --tail 100 my-container

# View logs with timestamps
docker logs -t my-container

# Execute command in running container
docker exec -it my-container /bin/bash

# Execute as specific user
docker exec -it -u root my-container bash
```

```
# Stop container gracefully (10 second timeout)
docker stop my-container

# Stop container immediately
docker kill my-container

# Restart container
docker restart my-container

# Remove container
docker rm my-container

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

# Remove all stopped containers
docker container prune
```

5 Dockerfile

5.1 What is a Dockerfile?

A Dockerfile is a text file containing instructions to build a Docker image. Each instruction creates a new layer in the image.

5.2 Dockerfile Instructions

Instruction	Description
FROM	Set base image (must be first instruction)
RUN	Execute command during build
CMD	Default command to run when container starts
ENTRYPOINT	Configure container as executable
COPY	Copy files from host to container
ADD	Copy files with additional features (tar extraction, URL support)
WORKDIR	Set working directory
ENV	Set environment variables
EXPOSE	Document which ports the container listens on
VOLUME	Create mount point for volumes
USER	Set user for subsequent instructions
ARG	Define build-time variables
LABEL	Add metadata to image
HEALTHCHECK	Define command to check container health
SHELL	Override default shell

Table 4: Dockerfile Instructions

5.3 Dockerfile Examples

5.3.1 Simple Python Application

```
# Use official Python runtime as base image
FROM python:3.9-slim

# Set metadata
LABEL maintainer="developer@example.com"
LABEL version="1.0"
LABEL description="Python␣Flask␣Application"

# Set working directory
WORKDIR /app

# Copy requirements file
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

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

# Set default command
CMD ["flask", "run", "--host=0.0.0.0"]
```

5.3.2 Node.js Application

```
# Multi-stage build for Node.js
FROM node:16-alpine AS builder

WORKDIR /app

# Copy package files
COPY package*.json ./

# Install dependencies
RUN npm ci --only=production

# Copy source code
COPY . .

# Build application
RUN npm run build

# Production stage
FROM node:16-alpine

WORKDIR /app

# Copy built application from builder
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/node_modules ./node_modules
COPY --from=builder /app/package.json ./

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

```
# Change ownership
RUN chown -R nodejs:nodejs /app

# Switch to non-root user
USER nodejs

EXPOSE 3000

CMD ["node", "dist/index.js"]
```

5.3.3 Java Spring Boot Application

```
# Build stage
FROM maven:3.8-openjdk-17 AS build

WORKDIR /app

# Copy pom.xml first (better caching)
COPY pom.xml .

# Download dependencies
RUN mvn dependency:go-offline

# Copy source code
COPY src ./src

# Build application
RUN mvn clean package -DskipTests

# Runtime stage
FROM openjdk:17-jdk-slim

WORKDIR /app

# Copy JAR from build stage
COPY --from=build /app/target/*.jar app.jar

# Add volume for logs
VOLUME /tmp

# Expose port
EXPOSE 8080

# JVM optimization flags
ENV JAVA_OPTS="-Xmx512m -Xms256m"

# Entry point with JVM options
ENTRYPOINT ["sh", "-c", "java $JAVA_OPTS -jar app.jar"]
```

5.4 Building Images

```
# Build image from Dockerfile in current directory
docker build -t myapp:latest .

# Build with specific Dockerfile name
docker build -f Dockerfile.dev -t myapp:dev .

# Build with build arguments
docker build --build-arg VERSION=1.0 -t myapp:latest .

# Build without using cache
docker build --no-cache -t myapp:latest .

# Build with custom build context
docker build -t myapp:latest -f /path/to/Dockerfile /path/to/context

# Build and tag with multiple tags
docker build -t myapp:latest -t myapp:v1.0 .

# View build cache
docker builder prune

# Remove build cache
docker builder prune --all
```

5.5 Best Practices for Dockerfile

1. **Use specific base image versions** - Don't use `latest` tag
2. **Minimize layers** - Combine RUN commands with && operator
3. **Order instructions** - Put least frequently changing instructions first
4. **Use .dockerignore** - Exclude unnecessary files from build context
5. **Multi-stage builds** - Reduce final image size
6. **Use COPY over ADD** - Unless you need ADD's features
7. **Run as non-root user** - Improve security
8. **Use HEALTHCHECK** - Enable container health monitoring
9. **Minimize installed packages** - Reduce attack surface
10. **Clean up in same layer** - Remove temporary files immediately

5.5.1 .dockerignore Example

```
# Git files
.git
.gitignore

# Documentation
*.md
docs/

# Build artifacts
target/
build/
dist/

# Dependencies
node_modules/
venv/

# IDE files
.idea/
.vscode/
*.swp

# Environment files
.env
.env.local

# Logs
*.log
logs/

# OS files
.DS_Store
Thumbs.db
```

6 Docker Volumes

6.1 Understanding Volumes

Volumes are the preferred mechanism for persisting data generated by Docker containers. They are:

- Managed by Docker
- Independent of container lifecycle
- Can be shared between containers
- Can be backed up or migrated easily
- Work on both Linux and Windows

6.2 Types of Data Persistence

Type	Description	Use Case
Volumes	Managed by Docker, stored in Docker area	Database data, application state
Bind Mounts	Mount host directory directly	Development, configuration files
tmpfs Mounts	Stored in host memory only	Sensitive temporary data

Table 5: Data Persistence Options

6.3 Volume Commands

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

# List all volumes
docker volume ls

# Inspect volume
docker volume inspect my-volume

# Remove volume
docker volume rm my-volume

# Remove all unused volumes
docker volume prune

# Remove all volumes (dangerous!)
docker volume prune -a
```

6.4 Using Volumes

```
# Run container with named volume
docker run -d -v my-volume:/data nginx:latest

# Run container with bind mount (absolute path)
docker run -d -v /host/path:/container/path nginx:latest

# Run container with bind mount (relative path)
docker run -d -v $(pwd):/app nginx:latest

# Read-only volume
docker run -d -v my-volume:/data:ro nginx:latest

# Create volume with specific driver
docker volume create --driver local \
  --opt type=nfs \
  --opt o=addr=192.168.1.1,rw \
  --opt device=:/path/to/dir \
  nfs-volume

# Share volume between containers
docker run -d --name app1 -v shared-data:/data nginx
docker run -d --name app2 -v shared-data:/data nginx
```

6.5 Volume Backup and Restore

```
# Backup volume to tar file
docker run --rm \
  -v my-volume:/data \
  -v $(pwd):/backup \
  ubuntu tar cvf /backup/backup.tar /data

# Restore volume from tar file
docker run --rm \
  -v my-volume:/data \
  -v $(pwd):/backup \
  ubuntu tar xvf /backup/backup.tar -C /

# Copy data between volumes
docker run --rm \
  -v source-volume:/source \
  -v target-volume:/target \
  ubuntu cp -r /source/. /target/
```

7 Docker Networks

7.1 Network Drivers

Driver	Description
bridge	Default network driver. Isolated network on single host
host	Remove network isolation, use host's network directly
overlay	Connect multiple Docker daemons (Swarm mode)
macvlan	Assign MAC address to container, appears as physical device
none	Disable networking

Table 6: Docker Network Drivers

7.2 Network Commands

```
# List networks
docker network ls

# Create network
docker network create my-network

# Create network with specific driver
docker network create --driver bridge my-bridge-network

# Create network with subnet
docker network create --subnet=172.18.0.0/16 my-network

# Inspect network
docker network inspect my-network

# Connect container to network
docker network connect my-network my-container

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

# Remove network
docker network rm my-network

# Remove all unused networks
docker network prune
```

7.3 Container Communication

```
# Create custom bridge network
docker network create --driver bridge my-app-network

# Run containers on same network
```

```
docker run -d --name database \
  --network my-app-network \
  postgres:latest

docker run -d --name webapp \
  --network my-app-network \
  -p 8080:80 \
  nginx:latest

# Containers can communicate using container names
# From webapp container: ping database

# Run with network alias
docker run -d --name db \
  --network my-app-network \
  --network-alias postgres \
  postgres:latest

# Connect to multiple networks
docker run -d --name multi-net-app \
  --network network1 \
  nginx:latest

docker network connect network2 multi-net-app
```

7.4 DNS and Service Discovery

```
# Containers on user-defined networks can resolve each other by name
docker network create app-tier

# Start database
docker run -d --name mysql-db \
  --network app-tier \
  -e MYSQL_ROOT_PASSWORD=secret \
  mysql:8

# Application can connect using hostname "mysql-db"
docker run -d --name app \
  --network app-tier \
  -e DB_HOST=mysql-db \
  myapp:latest

# Test connectivity
docker exec app ping mysql-db
```

7.5 Advanced Network Configuration

```
# Create network with custom options
docker network create \
```

```
  --driver bridge \
  --subnet 172.20.0.0/16 \
  --ip-range 172.20.240.0/20 \
  --gateway 172.20.0.1 \
  --opt com.docker.network.bridge.name=custom-bridge \
  custom-network

# Assign static IP to container
docker run -d --name static-ip-app \
  --network custom-network \
  --ip 172.20.0.10 \
  nginx:latest

# Create overlay network for Swarm
docker network create \
  --driver overlay \
  --attachable \
  my-overlay-network

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

8 Docker Compose

8.1 What is Docker Compose?

Docker Compose is a tool for defining and running multi-container Docker applications. You use a YAML file to configure your application's services, networks, and volumes.

8.2 Docker Compose File Structure

```
# docker-compose.yml basic structure
version: '3.8'

services:
  # Service definitions

networks:
  # Network definitions

volumes:
  # Volume definitions
```

8.3 Docker Compose Commands

Command	Description
`docker compose up`	Create and start containers
`docker compose up -d`	Start containers in background
`docker compose down`	Stop and remove containers, networks
`docker compose start`	Start existing containers
`docker compose stop`	Stop running containers
`docker compose restart`	Restart containers
`docker compose ps`	List containers
`docker compose logs`	View container logs
`docker compose exec`	Execute command in service
`docker compose build`	Build or rebuild services
`docker compose pull`	Pull service images
`docker compose config`	Validate and view configuration

Table 7: Docker Compose Commands

8.4 Complete Docker Compose Examples

8.4.1 Web Application with Database

```
version: '3.8'

services:
  # Database service
  db:
```

```
    image: postgres:14
    container_name: postgres-db
    restart: always
    environment:
      POSTGRES_DB: myapp
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: secretpassword
    volumes:
      - db-data:/var/lib/postgresql/data
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql
    networks:
      - backend
    healthcheck:
      test: ["CMD-SHELL", "pg_isready␣-U␣admin"]
      interval: 10s
      timeout: 5s
      retries: 5

  # Backend API service
  api:
    build:
      context: ./api
      dockerfile: Dockerfile
    container_name: api-server
    restart: always
    ports:
      - "5000:5000"
    environment:
      DATABASE_URL: postgresql://admin:secretpassword@db:5432/myapp
      API_KEY: ${API_KEY}
    depends_on:
      db:
        condition: service_healthy
    networks:
      - backend
      - frontend
    volumes:
      - ./api:/app
      - /app/node_modules

  # Frontend service
  web:
    build:
      context: ./web
      dockerfile: Dockerfile
    container_name: web-frontend
    restart: always
    ports:
      - "80:80"
      - "443:443"
    environment:
```

```
      API_URL: http://api:5000
    depends_on:
      - api
    networks:
      - frontend
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro

  # Redis cache service
  cache:
    image: redis:7-alpine
    container_name: redis-cache
    restart: always
    ports:
      - "6379:6379"
    networks:
      - backend
    volumes:
      - cache-data:/data
    command: redis-server --appendonly yes

networks:
  frontend:
    driver: bridge
  backend:
    driver: bridge

volumes:
  db-data:
  cache-data:
```

8.4.2 Microservices Architecture

```
version: '3.8'

services:
  # Service discovery and load balancing
  traefik:
    image: traefik:v2.9
    container_name: traefik
    restart: always
    command:
      - "--api.insecure=true"
      - "--providers.docker=true"
      - "--entrypoints.web.address=:80"
    ports:
      - "80:80"
      - "8080:8080"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro
```

```
    networks:
      - web

  # User service
  user-service:
    build: ./services/user
    restart: always
    environment:
      DB_HOST: postgres
      REDIS_HOST: redis
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.user.rule=PathPrefix(`/api/users`)"
    depends_on:
      - postgres
      - redis
    networks:
      - web
      - backend
    deploy:
      replicas: 3

  # Order service
  order-service:
    build: ./services/order
    restart: always
    environment:
      DB_HOST: postgres
      RABBITMQ_HOST: rabbitmq
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.order.rule=PathPrefix(`/api/orders`)"
    depends_on:
      - postgres
      - rabbitmq
    networks:
      - web
      - backend

  # PostgreSQL database
  postgres:
    image: postgres:14
    restart: always
    environment:
      POSTGRES_MULTIPLE_DATABASES: users,orders
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: password
    volumes:
      - postgres-data:/var/lib/postgresql/data
    networks:
      - backend
```

```
  # Redis cache
  redis:
    image: redis:7-alpine
    restart: always
    networks:
      - backend

  # RabbitMQ message broker
  rabbitmq:
    image: rabbitmq:3-management
    restart: always
    ports:
      - "15672:15672"
    environment:
      RABBITMQ_DEFAULT_USER: admin
      RABBITMQ_DEFAULT_PASS: password
    networks:
      - backend

networks:
  web:
    driver: bridge
  backend:
    driver: bridge

volumes:
  postgres-data:
```

8.5 Advanced Compose Features

8.5.1 Environment Variables and Secrets

```
# Using .env file
# Create .env file in same directory as docker-compose.yml
# Contents of .env:
# DB_PASSWORD=secret123
# API_KEY=myapikey
# VERSION=1.0.0

version: '3.8'

services:
  app:
    image: myapp:${VERSION}
    environment:
      DATABASE_PASSWORD: ${DB_PASSWORD}
      API_KEY: ${API_KEY}
    env_file:
      - ./common.env
```

```
      - ./app.env

  # Using secrets (Docker Swarm)
  secure-app:
    image: myapp:latest
    secrets:
      - db_password
      - api_key

secrets:
  db_password:
    file: ./secrets/db_password.txt
  api_key:
    external: true
```

8.5.2 Health Checks and Dependencies

```
version: '3.8'

services:
  database:
    image: postgres:14
    healthcheck:
      test: ["CMD-SHELL", "pg_isready␣-U␣postgres"]
      interval: 10s
      timeout: 5s
      retries: 5
      start_period: 30s

  backend:
    image: myapi:latest
    depends_on:
      database:
        condition: service_healthy
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8000/health"]
      interval: 30s
      timeout: 10s
      retries: 3

  frontend:
    image: myweb:latest
    depends_on:
      backend:
        condition: service_healthy
```

8.5.3 Resource Limits and Constraints

```
version: '3.8'
```

```
services:
  limited-app:
    image: myapp:latest
    deploy:
      resources:
        limits:
          cpus: '0.5'
          memory: 512M
        reservations:
          cpus: '0.25'
          memory: 256M
      restart_policy:
        condition: on-failure
        delay: 5s
        max_attempts: 3
        window: 120s
```

8.6 Using Docker Compose

```
# Start all services
docker compose up

# Start in detached mode
docker compose up -d

# Build images before starting
docker compose up --build

# Start specific service
docker compose up -d database

# Scale a service
docker compose up -d --scale api=3

# Stop all services
docker compose stop

# Stop and remove containers, networks
docker compose down

# Stop and remove everything including volumes
docker compose down -v

# View logs
docker compose logs

# Follow logs
docker compose logs -f
```

```
# View logs for specific service
docker compose logs -f api

# Execute command in running service
docker compose exec api bash

# Run one-off command
docker compose run --rm api python manage.py migrate

# View running services
docker compose ps

# View service configuration
docker compose config

# Validate compose file
docker compose config --quiet

# Pull all images
docker compose pull

# Build all services
docker compose build

# Build specific service
docker compose build api

# Build without cache
docker compose build --no-cache

# Restart services
docker compose restart

# Pause services
docker compose pause

# Unpause services
docker compose unpause
```

9 Docker Registry

9.1 Docker Hub

Docker Hub is the default public registry for Docker images.

```
# Login to Docker Hub
docker login

# Login with username
docker login -u username

# Tag image for Docker Hub
docker tag myapp:latest username/myapp:latest
docker tag myapp:latest username/myapp:v1.0

# Push image to Docker Hub
docker push username/myapp:latest
docker push username/myapp:v1.0

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

# Logout
docker logout

# Search for images
docker search nginx

# Search with filter
docker search --filter stars=100 nginx
```

9.2 Private Registry

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

# Run registry with persistent storage
docker run -d \
  -p 5000:5000 \
  --name registry \
  -v registry-data:/var/lib/registry \
  registry:2

# Tag image for private registry
docker tag myapp:latest localhost:5000/myapp:latest

# Push to private registry
docker push localhost:5000/myapp:latest

# Pull from private registry
```

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

# List images in private registry
curl -X GET http://localhost:5000/v2/_catalog

# List tags for image
curl -X GET http://localhost:5000/v2/myapp/tags/list
```

9.3 Secure Private Registry

```
# Create certificates directory
mkdir -p certs

# Generate self-signed certificate
openssl req -newkey rsa:4096 -nodes -sha256 \
  -keyout certs/domain.key -x509 -days 365 \
  -out certs/domain.crt

# Run registry with TLS
docker run -d \
  -p 5000:5000 \
  --name secure-registry \
  -v $(pwd)/certs:/certs \
  -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/domain.crt \
  -e REGISTRY_HTTP_TLS_KEY=/certs/domain.key \
  registry:2

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

# Create htpasswd file
docker run --rm --entrypoint htpasswd \
  httpd:2 -Bbn username password > auth/htpasswd

# Login to secured registry
docker login myregistry.com:5000
```

10 Docker Security

10.1 Security Best Practices

1. **Use official and verified images**
2. **Keep images and Docker updated**
3. **Run containers as non-root user**
4. **Use read-only file systems**
5. **Limit container resources**
6. **Scan images for vulnerabilities**
7. **Use secrets management**
8. **Enable Docker Content Trust**
9. **Use security profiles (AppArmor, SELinux)**
10. **Minimize attack surface**

10.2 Security Commands and Techniques

```
# Run container as non-root user
docker run -u 1000:1000 nginx:latest

# Run with read-only root filesystem
docker run --read-only nginx:latest

# Run with read-only root and writable tmp
docker run --read-only --tmpfs /tmp nginx:latest

# Drop all capabilities
docker run --cap-drop=ALL nginx:latest

# Add specific capability
docker run --cap-drop=ALL --cap-add=NET_BIND_SERVICE nginx:latest

# Run without new privileges
docker run --security-opt=no-new-privileges nginx:latest

# Use AppArmor profile
docker run --security-opt apparmor=docker-default nginx:latest

# Use SELinux labels
docker run --security-opt label=level:s0:c100,c200 nginx:latest

# Limit PID namespace
docker run --pids-limit 100 nginx:latest
```

```
# Set resource limits
docker run -m 512m --cpus=0.5 nginx:latest

# Use secrets in Docker Swarm
echo "mysecretpassword" | docker secret create db_password -

# Run container with secret
docker service create \
  --name mysql \
  --secret db_password \
  -e MYSQL_ROOT_PASSWORD_FILE=/run/secrets/db_password \
  mysql:8
```

10.3 Image Scanning

```
# Scan image with Docker Scout (built-in)
docker scout cves myapp:latest

# View detailed CVE information
docker scout cves --details myapp:latest

# Compare images
docker scout compare myapp:v1 myapp:v2

# Scan with Trivy
docker run aquasec/trivy image myapp:latest

# Scan with Snyk
docker scan myapp:latest

# Enable Docker Content Trust
export DOCKER_CONTENT_TRUST=1

# Sign and push image
docker push myrepo/myapp:latest

# Pull and verify signed image
docker pull myrepo/myapp:latest
```

10.4 Dockerfile Security

```
# Secure Dockerfile example
FROM node:18-alpine AS builder

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

WORKDIR /app
```

```
# Copy and install dependencies
COPY --chown=nodejs:nodejs package*.json ./
RUN npm ci --only=production

# Copy application
COPY --chown=nodejs:nodejs . .

# Build application
RUN npm run build

# Production stage
FROM node:18-alpine

# Install dumb-init for proper signal handling
RUN apk add --no-cache dumb-init

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

WORKDIR /app

# Copy from builder
COPY --from=builder --chown=nodejs:nodejs /app/dist ./dist
COPY --from=builder --chown=nodejs:nodejs /app/node_modules ./node_modules

# Switch to non-root user
USER nodejs

# Expose port
EXPOSE 3000

# Use dumb-init as entrypoint
ENTRYPOINT ["dumb-init", "--"]

# Health check
HEALTHCHECK --interval=30s --timeout=3s \
  CMD node healthcheck.js

# Start application
CMD ["node", "dist/index.js"]
```

11 Docker Monitoring and Logging

11.1 Container Monitoring

```
# View real-time resource usage
docker stats

# View stats for specific containers
docker stats container1 container2

# View stats without streaming
docker stats --no-stream

# View stats with custom format
docker stats --format "table␣{{.Name}}\t{{.CPUPerc}}\t{{.MemUsage}}"

# View container processes
docker top my-container

# View container resource limits
docker inspect --format='{{.HostConfig.Memory}}' my-container

# View container events
docker events

# Filter events by type
docker events --filter 'type=container'

# Filter events by container
docker events --filter 'container=my-container'

# View events since specific time
docker events --since '2024-01-01T00:00:00'
```

11.2 Logging

```
# View container logs
docker logs my-container

# Follow logs in real-time
docker logs -f my-container

# View last N lines
docker logs --tail 100 my-container

# View logs with timestamps
docker logs -t my-container

# View logs since specific time
docker logs --since 2024-01-01T00:00:00 my-container
```

```
# View logs for last N minutes
docker logs --since 30m my-container

# View logs until specific time
docker logs --until 2024-01-01T23:59:59 my-container

# Configure logging driver
docker run -d \
  --log-driver json-file \
  --log-opt max-size=10m \
  --log-opt max-file=3 \
  nginx:latest

# Use syslog driver
docker run -d \
  --log-driver syslog \
  --log-opt syslog-address=tcp://192.168.1.100:514 \
  nginx:latest

# Use fluentd driver
docker run -d \
  --log-driver fluentd \
  --log-opt fluentd-address=localhost:24224 \
  --log-opt tag="docker.{{.Name}}" \
  nginx:latest
```

11.3 Monitoring Stack with Prometheus and Grafana

```
# docker-compose.yml for monitoring stack
version: '3.8'

services:
  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
      - prometheus-data:/prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
    networks:
      - monitoring

  grafana:
    image: grafana/grafana:latest
    container_name: grafana
```

```
    ports:
      - "3000:3000"
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin
    volumes:
      - grafana-data:/var/lib/grafana
    depends_on:
      - prometheus
    networks:
      - monitoring

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

  node-exporter:
    image: prom/node-exporter:latest
    container_name: node-exporter
    ports:
      - "9100:9100"
    networks:
      - monitoring

networks:
  monitoring:
    driver: bridge

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

12 Docker Troubleshooting

12.1 Common Issues and Solutions

Issue	Cause	Solution
Container exits immediately	Wrong CMD/ENTRYPOINT	Check container logs
Port already in use	Port conflict	Use different host port
Permission denied	User permissions	Add user to docker group
Out of disk space	Too many images/containers	Run docker system prune
Cannot connect to daemon	Docker not running	Start Docker service
DNS resolution fails	Network configuration	Check DNS settings

Table 8: Common Docker Issues

12.2 Debugging Commands

```
# Inspect container details
docker inspect my-container

# View specific information
docker inspect --format='{{.State.Status}}' my-container
docker inspect --format='{{.NetworkSettings.IPAddress}}' my-container

# Check container logs
docker logs --tail 50 my-container

# Check container processes
docker top my-container

# Execute shell in running container
docker exec -it my-container /bin/bash
docker exec -it my-container sh

# Run command as root in container
docker exec -it -u root my-container bash

# Copy files from container for inspection
docker cp my-container:/var/log/app.log ./app.log

# View container changes
docker diff my-container

# Export container filesystem
docker export my-container > container.tar

# View Docker system information
docker info
```

```
# View Docker version
docker version

# Check Docker disk usage
docker system df

# Detailed disk usage
docker system df -v

# View container resource usage
docker stats --no-stream my-container
```

12.3 Network Debugging

```
# Test connectivity between containers
docker exec container1 ping container2

# Check DNS resolution
docker exec my-container nslookup google.com

# View container IP address
docker inspect -f '{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}' \
  my-container

# Test port connectivity
docker exec my-container nc -zv hostname port

# Inspect network
docker network inspect bridge

# View all container IPs on network
docker network inspect bridge | \
  grep -A 3 "Containers"

# Run debugging container
docker run -it --rm --network container:my-container \
  nicolaka/netshoot

# Capture network traffic
docker exec my-container tcpdump -i eth0 -w /tmp/capture.pcap
```

12.4 Performance Troubleshooting

```
# Check container resource constraints
docker inspect --format='{{.HostConfig.Memory}}' my-container
docker inspect --format='{{.HostConfig.NanoCpus}}' my-container

# View real-time stats
docker stats my-container
```

```
# Check I/O stats
docker stats --format "table␣{{.Name}}\t{{.BlockIO}}"

# Run container with limited resources
docker run -d \
  -m 512m \
  --cpus=0.5 \
  --memory-swap=1g \
  myapp:latest

# View container events
docker events --filter 'container=my-container'

# Check for OOM (Out of Memory) kills
docker inspect --format='{{.State.OOMKilled}}' my-container

# View kernel messages (requires privileged)
docker run --rm --privileged ubuntu dmesg | grep -i oom
```

13 Docker Best Practices

13.1 Development Best Practices

1. **Use .dockerignore** - Exclude unnecessary files from build context
2. **Multi-stage builds** - Reduce final image size
3. **Minimize layers** - Combine RUN commands
4. **Use specific tags** - Avoid latest tag for production
5. **One process per container** - Follow single responsibility principle
6. **Use COPY instead of ADD** - Unless you need ADD features
7. **Leverage build cache** - Order instructions wisely
8. **Use health checks** - Enable monitoring
9. **Don't store secrets in images** - Use environment variables or secrets
10. **Clean up in same layer** - Remove temporary files immediately

13.2 Production Best Practices

1. **Run as non-root** - Improve security
2. **Use read-only filesystems** - When possible
3. **Scan images** - Check for vulnerabilities regularly
4. **Resource limits** - Set memory and CPU constraints
5. **Use restart policies** - Handle failures gracefully
6. **Implement health checks** - Enable automatic recovery
7. **Use logging drivers** - Centralize logs
8. **Monitor containers** - Track resource usage
9. **Use orchestration** - Kubernetes or Docker Swarm for production
10. **Backup volumes** - Protect persistent data

13.3 Optimization Checklist

Area	Optimization
Image Size	Use alpine base images, multi-stage builds, clean up caches
Build Time	Leverage cache, order instructions properly, use .dockerignore
Runtime	Set resource limits, use health checks, implement logging
Security	Run as non-root, scan images, use secrets management
Networking	Use user-defined networks, implement service discovery
Storage	Use volumes for data, implement backup strategy

Table 9: Docker Optimization Checklist

14 Advanced Docker Concepts

14.1 Docker Buildx

Docker Buildx is an enhanced build system with additional features.

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

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

# Build with cache
docker buildx build \
  --cache-from type=registry,ref=myapp:cache \
  --cache-to type=registry,ref=myapp:cache \
  -t myapp:latest .

# List builders
docker buildx ls

# Inspect builder
docker buildx inspect mybuilder

# Remove builder
docker buildx rm mybuilder
```

14.2 Docker Context

Manage multiple Docker environments.

```
# Create new context
docker context create remote-server \
  --docker "host=ssh://user@remote-server"

# List contexts
docker context ls

# Use context
docker context use remote-server
```

```
# Inspect context
docker context inspect remote-server

# Remove context
docker context rm remote-server

# Export context
docker context export remote-server

# Import context
docker context import remote-server context.tar
```

14.3 Docker Plugins

```
# List plugins
docker plugin ls

# Install plugin
docker plugin install grafana/loki-docker-driver:latest --alias loki

# Enable plugin
docker plugin enable loki

# Disable plugin
docker plugin disable loki

# Configure plugin
docker plugin set loki LOKI_URL=http://localhost:3100

# Remove plugin
docker plugin rm loki

# Use plugin in container
docker run -d \
  --log-driver=loki \
  --log-opt loki-url="http://localhost:3100/loki/api/v1/push" \
  nginx:latest
```

15 Docker Swarm

15.1 Introduction to Swarm

Docker Swarm is Docker's native clustering and orchestration solution.

```
# Initialize swarm
docker swarm init

# Initialize with specific IP
docker swarm init --advertise-addr 192.168.1.100

# Join swarm as worker
docker swarm join --token SWMTKN-... 192.168.1.100:2377

# Join as manager
docker swarm join-token manager

# Leave swarm
docker swarm leave

# Force leave (on manager)
docker swarm leave --force
```

15.2 Managing Services

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

# List services
docker service ls

# Inspect service
docker service inspect web

# View service logs
docker service logs web

# Scale service
docker service scale web=5

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

# Remove service
docker service rm web

# List service tasks
docker service ps web

# Create service with constraints
```

```
docker service create \
  --name db \
  --constraint 'node.role==manager' \
  --replicas 1 \
  postgres:latest

# Create service with resource limits
docker service create \
  --name app \
  --limit-cpu 0.5 \
  --limit-memory 512M \
  --reserve-cpu 0.25 \
  --reserve-memory 256M \
  myapp:latest
```

15.3 Docker Stack

```
# Deploy stack from compose file
docker stack deploy -c docker-compose.yml mystack

# List stacks
docker stack ls

# List stack services
docker stack services mystack

# List stack tasks
docker stack ps mystack

# Remove stack
docker stack rm mystack

# Example stack compose file
version: '3.8'

services:
  web:
    image: nginx:latest
    ports:
      - "80:80"
    deploy:
      replicas: 3
      update_config:
        parallelism: 1
        delay: 10s
      restart_policy:
        condition: on-failure
        max_attempts: 3

  db:
```

```
    image: postgres:14
    deploy:
      placement:
        constraints:
          - node.role == manager
    volumes:
      - db-data:/var/lib/postgresql/data

volumes:
  db-data:
```

16 Docker System Management

16.1 System Cleanup

```
# Remove all stopped containers
docker container prune

# Remove all unused images
docker image prune

# Remove all unused images (including tagged)
docker image prune -a

# Remove all unused volumes
docker volume prune

# Remove all unused networks
docker network prune

# Remove everything unused
docker system prune

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

# View disk usage
docker system df

# View detailed disk usage
docker system df -v

# Remove specific resources older than 24 hours
docker image prune --filter "until=24h"

# Remove dangling images
docker rmi $(docker images -f "dangling=true" -q)

# Remove exited containers
docker rm $(docker ps -a -f status=exited -q)

# Remove containers by name pattern
docker rm $(docker ps -a | grep "pattern" | awk '{print␣$1}')
```

16.2 Configuration Files

```
# Docker daemon configuration
# Location: /etc/docker/daemon.json

{
  "log-driver": "json-file",
```

```
  "log-opts": {
    "max-size": "10m",
    "max-file": "3"
  },
  "storage-driver": "overlay2",
  "dns": ["8.8.8.8", "8.8.4.4"],
  "insecure-registries": ["myregistry.local:5000"],
  "registry-mirrors": ["https://mirror.example.com"],
  "max-concurrent-downloads": 10,
  "max-concurrent-uploads": 10,
  "default-address-pools": [
    {
      "base": "172.80.0.0/16",
      "size": 24
    }
  ]
}

# Reload daemon after configuration change
sudo systemctl daemon-reload
sudo systemctl restart docker
```

17 Useful Docker Commands Summary

17.1 Quick Reference Table

Category	Commands
Images	pull, push, build, images, rmi, tag, save, load, history, inspect
Containers	run, start, stop, restart, pause, unpause, kill, rm, ps, logs, exec, attach, cp, stats, top, port
Volumes	create, ls, inspect, rm, prune
Networks	create, ls, inspect, rm, prune, connect, disconnect
Compose	up, down, start, stop, restart, ps, logs, exec, build, pull, config
System	info, version, df, prune, events
Registry	login, logout, search, push, pull
Swarm	init, join, leave, update
Service	create, ls, inspect, logs, rm, scale, update, ps
Stack	deploy, ls, ps, rm, services

Table 10: Docker Commands Quick Reference

17.2 Essential Command Patterns

```
# Run container with common options
docker run -d \
  --name <name> \
  -p <host-port>:<container-port> \
  -v <volume-name>:<container-path> \
  -e <ENV_VAR>=<value> \
  --restart unless-stopped \
  --network <network-name> \
  <image>:<tag>

# Build image with best practices
docker build \
  -t <image>:<tag> \
  -f <Dockerfile> \
  --build-arg <ARG>=<value> \
  --no-cache \
  <context-path>

# Execute interactive shell
docker exec -it <container> /bin/bash
docker exec -it <container> sh

# View and follow logs
docker logs -f --tail 100 <container>

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

# Export and import containers
docker export <container> > container.tar
docker import container.tar <image>:<tag>

# Save and load images
docker save -o image.tar <image>:<tag>
```

```
docker load -i image.tar

# Inspect with formatting
docker inspect --format='{{json .State}}' <container> | jq
docker inspect --format='{{.NetworkSettings.IPAddress}}' <container>
```

18 Docker Use Cases and Examples

18.1 Development Environment

```
# Docker Compose for full development stack
version: '3.8'

services:
  # Development database
  postgres:
    image: postgres:14
    environment:
      POSTGRES_DB: devdb
      POSTGRES_USER: devuser
      POSTGRES_PASSWORD: devpass
    ports:
      - "5432:5432"
    volumes:
      - postgres-data:/var/lib/postgresql/data
      - ./init-scripts:/docker-entrypoint-initdb.d

  # Redis cache
  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"

  # Development application with hot reload
  app:
    build:
      context: .
      dockerfile: Dockerfile.dev
    ports:
      - "3000:3000"
    volumes:
      - ./src:/app/src
      - /app/node_modules
    environment:
      NODE_ENV: development
      DB_HOST: postgres
      REDIS_HOST: redis
    depends_on:
      - postgres
      - redis
    command: npm run dev

  # pgAdmin for database management
  pgadmin:
    image: dpage/pgadmin4
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@example.com
```

```
      PGADMIN_DEFAULT_PASSWORD: admin
    ports:
      - "5050:80"

volumes:
  postgres-data:
```

18.2 CI/CD Pipeline

```
# Dockerfile for CI/CD
FROM node:18-alpine AS builder

WORKDIR /app

COPY package*.json ./
RUN npm ci

COPY . .
RUN npm run test
RUN npm run lint
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 --from=builder /app/package.json ./

USER node

EXPOSE 3000

CMD ["node", "dist/index.js"]

# GitLab CI example
# .gitlab-ci.yml
stages:
  - build
  - test
  - deploy

variables:
  DOCKER_IMAGE: $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA

build:
  stage: build
  image: docker:latest
  services:
```

```
    - docker:dind
  script:
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
    - docker build -t $DOCKER_IMAGE .
    - docker push $DOCKER_IMAGE

test:
  stage: test
  image: $DOCKER_IMAGE
  script:
    - npm test

deploy:
  stage: deploy
  image: docker:latest
  script:
    - docker pull $DOCKER_IMAGE
    - docker tag $DOCKER_IMAGE $CI_REGISTRY_IMAGE:latest
    - docker push $CI_REGISTRY_IMAGE:latest
```

18.3 Microservices Architecture

```
# Complete microservices setup with service mesh
version: '3.8'

services:
  # API Gateway
  gateway:
    image: nginx:alpine
    ports:
      - "80:80"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
    depends_on:
      - user-service
      - product-service
      - order-service
    networks:
      - frontend

  # User Service
  user-service:
    build: ./services/user
    environment:
      DATABASE_URL: postgresql://postgres:password@user-db:5432/users
      REDIS_URL: redis://cache:6379
      JWT_SECRET: ${JWT_SECRET}
    depends_on:
      - user-db
      - cache
```

```
    networks:
      - frontend
      - backend
    deploy:
      replicas: 3
      restart_policy:
        condition: on-failure

  user-db:
    image: postgres:14
    environment:
      POSTGRES_DB: users
      POSTGRES_PASSWORD: password
    volumes:
      - user-db-data:/var/lib/postgresql/data
    networks:
      - backend

  # Product Service
  product-service:
    build: ./services/product
    environment:
      MONGO_URL: mongodb://product-db:27017/products
    depends_on:
      - product-db
    networks:
      - frontend
      - backend
    deploy:
      replicas: 2

  product-db:
    image: mongo:6
    volumes:
      - product-db-data:/data/db
    networks:
      - backend

  # Order Service
  order-service:
    build: ./services/order
    environment:
      DATABASE_URL: postgresql://postgres:password@order-db:5432/orders
      RABBITMQ_URL: amqp://rabbitmq:5672
    depends_on:
      - order-db
      - rabbitmq
    networks:
      - frontend
      - backend
    deploy:
```

```
    replicas: 2

order-db:
  image: postgres:14
  environment:
    POSTGRES_DB: orders
    POSTGRES_PASSWORD: password
  volumes:
    - order-db-data:/var/lib/postgresql/data
  networks:
    - backend

# Message Queue
rabbitmq:
  image: rabbitmq:3-management
  ports:
    - "15672:15672"
  environment:
    RABBITMQ_DEFAULT_USER: admin
    RABBITMQ_DEFAULT_PASS: password
  networks:
    - backend

# Shared Cache
cache:
  image: redis:7-alpine
  networks:
    - backend

# Monitoring
prometheus:
  image: prom/prometheus
  ports:
    - "9090:9090"
  volumes:
    - ./prometheus.yml:/etc/prometheus/prometheus.yml
    - prometheus-data:/prometheus
  networks:
    - monitoring

grafana:
  image: grafana/grafana
  ports:
    - "3000:3000"
  environment:
    GF_SECURITY_ADMIN_PASSWORD: admin
  volumes:
    - grafana-data:/var/lib/grafana
  networks:
    - monitoring
```

```
networks:
  frontend:
  backend:
  monitoring:

volumes:
  user-db-data:
  product-db-data:
  order-db-data:
  prometheus-data:
  grafana-data:
```

18.4 Testing Environment

```
# Docker Compose for integration testing
version: '3.8'

services:
  test-db:
    image: postgres:14
    environment:
      POSTGRES_DB: testdb
      POSTGRES_USER: testuser
      POSTGRES_PASSWORD: testpass
    tmpfs:
      - /var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready␣-U␣testuser"]
      interval: 5s
      timeout: 5s
      retries: 5

  test-redis:
    image: redis:7-alpine
    tmpfs:
      - /data

  app-test:
    build:
      context: .
      target: test
    environment:
      DATABASE_URL: postgresql://testuser:testpass@test-db:5432/testdb
      REDIS_URL: redis://test-redis:6379
      NODE_ENV: test
    depends_on:
      test-db:
        condition: service_healthy
      test-redis:
        condition: service_started
```

```
    command: npm test

# Run tests
# docker compose -f docker-compose.test.yml up --abort-on-container-exit
```

19 Docker Performance Optimization

19.1 Image Size Optimization

```
# BAD: Large image
FROM ubuntu:22.04
RUN apt-get update
RUN apt-get install -y python3 python3-pip
RUN pip3 install flask
COPY app.py /app/
CMD ["python3", "/app/app.py"]

# GOOD: Optimized image
FROM python:3.11-alpine
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY app.py .
CMD ["python", "app.py"]

# BEST: Multi-stage optimized
FROM python:3.11 AS builder
WORKDIR /app
COPY requirements.txt .
RUN pip install --user --no-cache-dir -r requirements.txt

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

19.2 Build Performance

```
# Optimize build order (cache-friendly)
FROM node:18-alpine

WORKDIR /app

# Copy package files first (changes less frequently)
COPY package*.json ./

# Install dependencies (cached if package files unchanged)
RUN npm ci --only=production

# Copy source code last (changes frequently)
COPY . .

# Build application
```

```
RUN npm run build

# Use BuildKit for faster builds
# export DOCKER_BUILDKIT=1

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

19.3 Runtime Performance

```
# Set appropriate resource limits
docker run -d \
  --name optimized-app \
  --memory="512m" \
  --memory-swap="1g" \
  --cpus="1.5" \
  --pids-limit 100 \
  myapp:latest

# Use host network for better performance (when security allows)
docker run -d --network host myapp:latest

# Optimize storage driver
# In daemon.json:
{
  "storage-driver": "overlay2",
  "storage-opts": [
    "overlay2.override_kernel_check=true"
  ]
}

# Use volumes instead of bind mounts for better I/O
docker run -d -v data-volume:/data myapp:latest

# Enable user namespace remapping for security and performance
# In daemon.json:
{
  "userns-remap": "default"
}
```

20 Docker Glossary

Term	Definition
Container	Runnable instance of an image with its own filesystem, networking, and process space
Image	Read-only template containing application code, libraries, and dependencies
Dockerfile	Text file with instructions to build a Docker image
Layer	Each instruction in a Dockerfile creates a layer in the final image
Registry	Service for storing and distributing Docker images (e.g., Docker Hub)
Repository	Collection of related images with different tags
Tag	Label to identify image versions
Volume	Persistent data storage managed by Docker
Network	Virtual network for container communication
Compose	Tool for defining multi-container applications using YAML
Swarm	Docker's native clustering and orchestration solution
Service	Definition of how containers should run in Swarm mode
Stack	Group of interrelated services deployed together
Node	Instance of Docker engine participating in a swarm
Task	Single container running as part of a service
Bind Mount	Direct mount of host directory into container
Bridge Network	Default network driver that creates isolated network
Host Network	Container uses host's network directly
Overlay Network	Multi-host network for swarm services
Daemon	Background service that manages Docker objects

Table 11: Docker Terminology

21 Troubleshooting Scenarios

21.1 Container Won't Start

```
# Check logs immediately
docker logs container-name

# Check last 50 lines with timestamps
docker logs --tail 50 -t container-name

# Inspect container state
docker inspect container-name | grep -A 10 "State"

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

# Common exit codes:
# 0 - Success
# 1 - Application error
# 125 - Docker daemon error
# 126 - Command cannot be executed
# 127 - Command not found
# 137 - SIGKILL (out of memory)
# 139 - SIGSEGV (segmentation fault)

# Try running with interactive shell
docker run -it --entrypoint /bin/sh image-name

# Check if dependencies are ready
docker run --rm image-name nc -zv dependency-host dependency-port
```

21.2 Network Connectivity Issues

```
# Verify container is on correct network
docker inspect container-name | grep -A 20 "Networks"

# Test DNS resolution
docker exec container-name nslookup other-container
docker exec container-name ping other-container

# Check network configuration
docker network inspect network-name

# Verify port mapping
docker port container-name

# Test from host
curl localhost:8080

# Check firewall rules
```

```
sudo iptables -L -n

# Verify container can reach internet
docker exec container-name ping -c 3 8.8.8.8
docker exec container-name curl https://google.com

# Check if port is already in use
sudo netstat -tulpn | grep :8080
sudo lsof -i :8080
```

21.3 Performance Issues

```
# Check resource usage
docker stats --no-stream

# Check if container is throttled
docker inspect container-name | grep -i throttl

# Monitor I/O
docker stats --format "table␣{{.Name}}\t{{.BlockIO}}"

# Check for OOM kills
docker inspect --format='{{.State.OOMKilled}}' container-name

# View system events
docker events --filter 'event=oom' --since '1h'

# Increase resources
docker update --memory 1g --cpus 2 container-name

# Check disk usage
docker system df -v

# Clean up to free space
docker system prune -a --volumes
```

21.4 Volume and Data Issues

```
# List all volumes
docker volume ls

# Inspect volume
docker volume inspect volume-name

# Check volume mount points
docker inspect container-name | grep -A 20 "Mounts"

# Verify permissions inside container
docker exec container-name ls -la /mount/path
```

```
# Check if volume is in use
docker ps -a --filter volume=volume-name

# Backup volume data
docker run --rm \
  -v volume-name:/source:ro \
  -v $(pwd):/backup \
  ubuntu tar czf /backup/backup.tar.gz -C /source .

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

# Fix permission issues
docker run --rm \
  -v volume-name:/data \
  ubuntu chown -R 1000:1000 /data
```

22 Additional Resources

22.1 Official Documentation

- **Docker Documentation:** https://docs.docker.com
- **Docker Hub:** https://hub.docker.com
- **Docker Compose:** https://docs.docker.com/compose
- **Dockerfile Reference:** https://docs.docker.com/engine/reference/builder
- **Docker CLI Reference:** https://docs.docker.com/engine/reference/commandline/cli

22.2 Best Practices Guides

- **Security:** https://docs.docker.com/engine/security
- **Networking:** https://docs.docker.com/network
- **Storage:** https://docs.docker.com/storage
- **Production:** https://docs.docker.com/config/containers/start-containers-automatic

22.3 Community Resources

- **Docker Forums:** https://forums.docker.com
- **GitHub Issues:** https://github.com/docker
- **Stack Overflow:** Tag: `docker`
- **Docker Blog:** https://www.docker.com/blog

22.4 Learning Resources

- **Docker Training:** Official Docker courses
- **Play with Docker:** https://labs.play-with-docker.com
- **Katacoda:** Interactive Docker scenarios
- **Docker Samples:** https://github.com/docker/awesome-compose

23 Quick Command Reference Card

Essential Docker Commands

Images:

```
docker pull <image> # Download image
docker build -t <name> . # Build image
docker images # List images
docker rmi <image> # Remove image
```

Containers:

```
docker run -d -p 8080:80 <image> # Run container
docker ps # List running containers
docker ps -a # List all containers
docker stop <container> # Stop container
docker start <container> # Start container
docker rm <container> # Remove container
docker logs -f <container> # View logs
docker exec -it <container> bash # Access shell
```

Compose:

```
docker compose up -d # Start services
docker compose down # Stop services
docker compose logs -f # View logs
docker compose ps # List services
```

System:

```
docker system df # Check disk usage
docker system prune # Clean up
docker info # System information
docker version # Docker version
```

24 Conclusion

Docker has revolutionized the way we develop, deploy, and manage applications. This comprehensive guide has covered:

- **Fundamentals:** Understanding containers, images, and Docker architecture
- **Core Concepts:** Working with Dockerfile, volumes, networks, and registries
- **Orchestration:** Using Docker Compose and Docker Swarm
- **Best Practices:** Security, optimization, and production deployment
- **Advanced Topics:** Multi-stage builds, BuildKit, monitoring, and troubleshooting
- **Real-World Examples:** Development environments, CI/CD, and microservices

24.1 Key Takeaways

1. Docker provides consistent environments across development, testing, and production
2. Containerization enables efficient resource utilization and rapid deployment
3. Security should be a priority from development through production
4. Proper monitoring and logging are essential for maintaining containerized applications
5. Multi-stage builds and optimization techniques reduce image size and improve performance
6. Docker Compose simplifies multi-container application management
7. Regular maintenance and cleanup prevent resource exhaustion

24.2 Next Steps

To continue your Docker journey:

- Practice building and deploying your own applications
- Explore container orchestration with Kubernetes
- Implement CI/CD pipelines with Docker
- Study advanced networking and security configurations
- Contribute to open-source Docker projects
- Join the Docker community and share your knowledge

Happy Dockerizing!