Docker and Containers Complete Deep Dive

1. What are Containers?

Definition

A **container** is a lightweight, standalone, executable package that includes everything needed to run a piece of software:

- Application code
- Runtime environment
- System libraries
- Dependencies
- Configuration files

Key Characteristics

Isolated: Each container runs in its own isolated environment

Portable: Run anywhere - laptop, server, cloud (same behavior everywhere)

Lightweight: Share host OS kernel, don't need full OS per container

Fast: Start in seconds (vs minutes for VMs)

Analogy

Think of containers like **shipping containers**:

- Standardized size/format
- Can hold anything inside
- Easy to move (ship, train, truck)
- Stackable and isolated
- Contents protected from outside

---

2. What is Docker?

Definition

> **Docker** is a platform that enables developers to build, package, share, and run applications in containers.

Docker is NOT the only containerization technology

- **Docker**: Most popular, user-friendly
- **Podman**: Daemon-less alternative
- **LXC/LXD**: Linux containers
- **containerd**: Container runtime (used by Docker internally)

What Docker Provides

1. **Tools** to create containers

2. **Platform** to run containers
3. **Registry** to share containers (Docker Hub)
4. **Ecosystem** of supporting tools

---

3. Why Containers Exist (The Problem They Solve)

The "Works on My Machine" Problem

Scenario Without Containers:

```
Developer's Laptop:
- Python 3.9
- PostgreSQL 12
- Ubuntu 20.04  App works perfectly!

Production Server:
- Python 3.7
- PostgreSQL 10
- CentOS 7
 App crashes!
```

Why? Different environments, dependencies, configurations

Traditional Problems

Problem 1: Dependency Hell

```
App A needs Python 3.7
App B needs Python 3.9
Both on same server? Conflict!
```

Problem 2: Environment Inconsistency

- Dev uses Windows
- Staging uses Ubuntu
- Production uses Red Hat
- Different behaviors everywhere

Problem 3: Resource Waste

- Running 10 apps needs 10 VMs?
- Each VM needs full OS (GB of RAM each)
- Expensive and slow

How Containers Solve This

Solution 1: Package Everything Together

```
Container = App + Python 3.9 + All Dependencies
Works identically everywhere!
```

Solution 2: Isolation Without Overhead

```
Same Host OS Kernel
├── Container 1 (Python 3.7 app)
├── Container 2 (Python 3.9 app)
└── Container 3 (Node.js app)
No conflicts, minimal overhead!
```

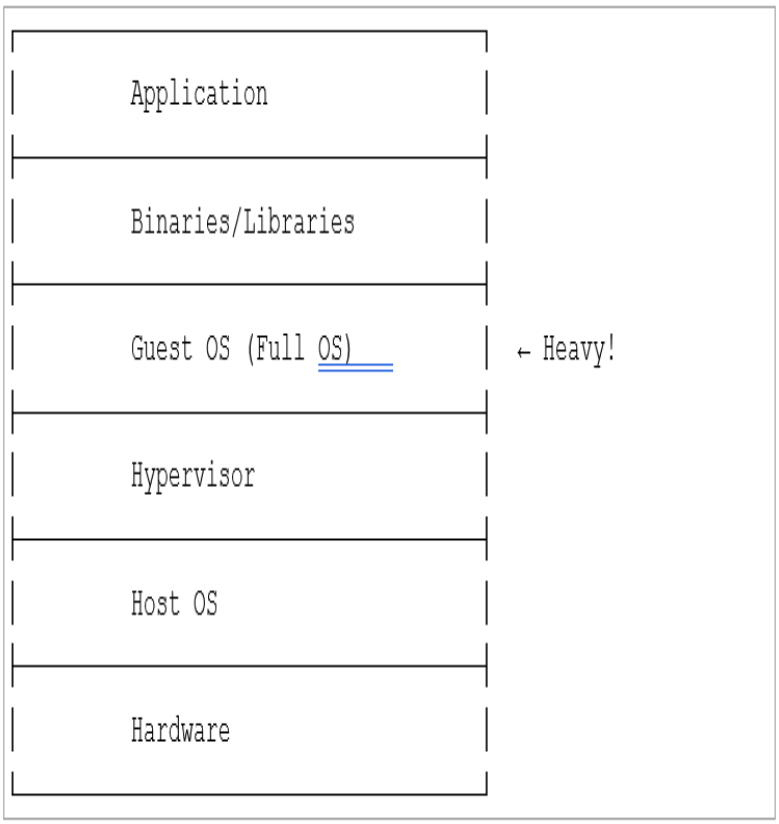
Solution 3: Consistency

```
Build once → Run anywhere
Same container in dev, test, production
```

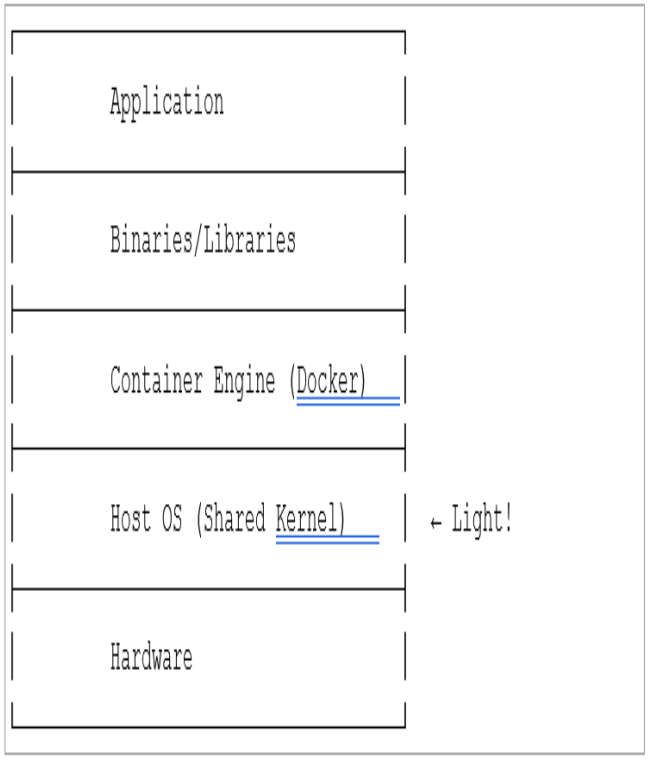
4. Container vs Virtual Machine

Architecture Comparison

Virtual Machine Architecture:



Container Architecture:



Key Differences

Feature	Virtual Machine	Container
Size	GBs (includes full OS)	MBs (shares host OS)
Startup	Minutes	Seconds
Performance	Slower (overhead)	Near-native
Isolation	Complete (hardware-level)	Process-level
Resource Usage	High	Low
Portability	Limited	Highly portable
Use Case	Different OS needed	Same OS kernel

When to Use What?

Use Virtual Machines when:

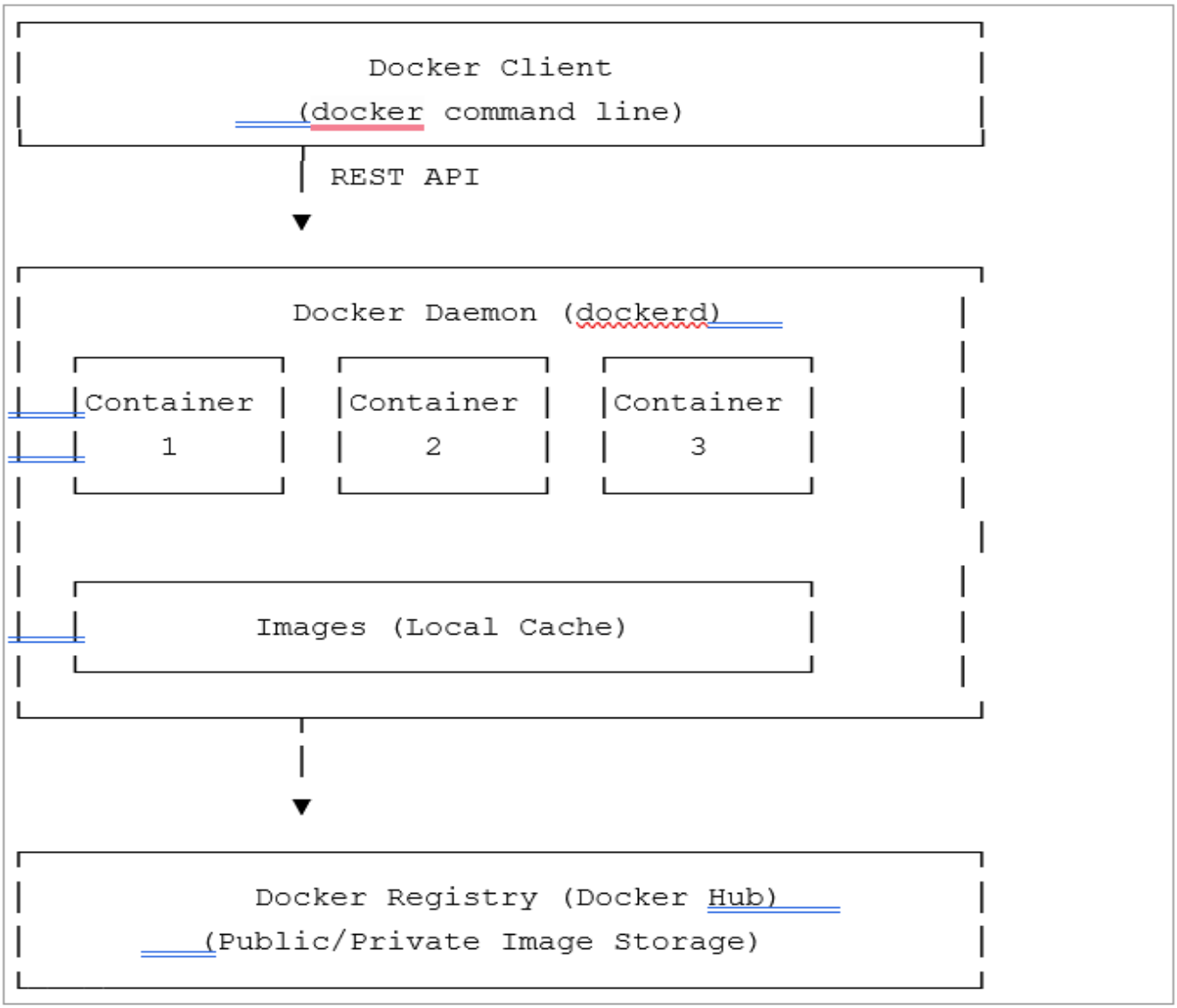
1. Need different operating systems (Windows on Linux host)
2. Need complete isolation for security
3. Running legacy applications
4. Need to simulate entire hardware

Use Containers when

- Microservices architecture
- CI/CD pipelines
- Scalable web applications
- Development environments
- Need fast startup and scaling

Docker Architecture

High-Level Architecture



Components Explained

1.Docker Client

- Command-line interface (CLI)
- What you interact with: `docker run`, `docker build`, etc.
- Sends commands to Docker Daemon

2.Docker Daemon (dockerd)

- Background service running on host
- Manages containers, images, networks, volumes
- Listens for Docker API requests
- Does the actual work

3.Docker Registry

- Stores Docker images
- **Docker Hub**: Public registry (like GitHub for containers)
- Private registries: Companies host their own

4 . Docker Objects

- **Images**: Read-only templates
- **Containers**: Running instances of images
- **Networks**: Communication between containers
- **Volumes**: Persistent data storage

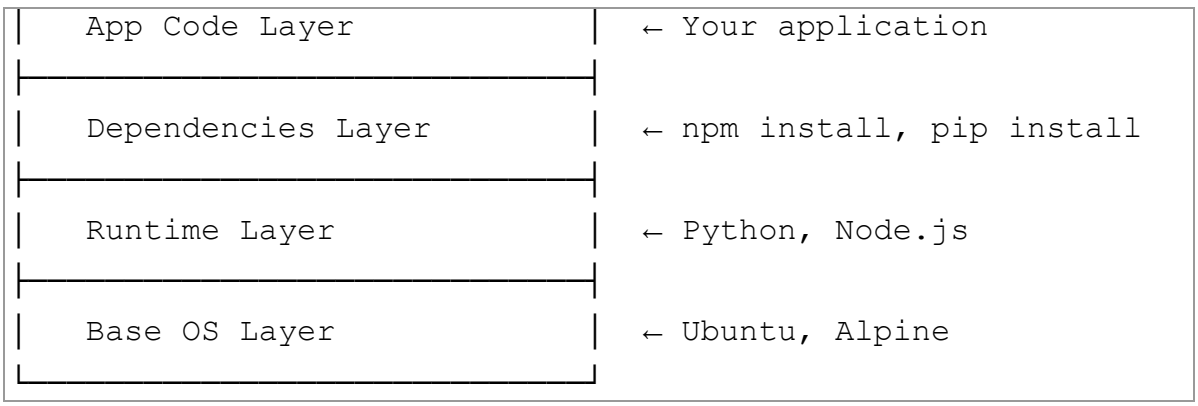
6. Docker Components in Detail

A. Docker Image

What is it?

- Read-only template with instructions
- Blueprint for creating containers
- Contains: OS, application, dependencies, configuration

Image Structure (Layers):



Why Layers?

- **Efficiency**: Shared layers between images
- **Caching**: Rebuild only changed layers
- **Size**: Reuse common layers

Example:

```
Image A: Ubuntu → Python → App A
Image B: Ubuntu → Python → App B
         └──Shared──┘
```

B. Docker Container

What is it?

- Running instance of an image
- Isolated process with its own filesystem
- Can be started, stopped, deleted

Container = Image + Writable Layer

```
┌─────────────────────────────────┐
│   Writable Container Layer      │  ← Changes, logs, temp files
├─────────────────────────────────┤
│   Image Layers (Read-only)      │
└─────────────────────────────────┘
```

Container Lifecycle:

```
Created → Running → Paused → Stopped → Removed
```

C. Dockerfile

What is it?

- Text file with instructions to build an image
- Step-by-step recipe
- Each instruction = new layer

Basic Structure:

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

# Set working directory
WORKDIR /app

# Copy files
COPY. /app

# Install dependencies
RUN apt-get update && apt-get install -y python3

# Define startup command
CMD ["python3", "app.py"]
```

D. Docker Registry

What is it?

- Storage and distribution system for images
- Like GitHub, but for Docker images

Types:

1. **Docker Hub**: Public registry (hub.docker.com)
2. **Private Registry**: Self-hosted or cloud
3. **Cloud Registries**: AWS ECR, Google GCR, Azure ACR **Image Naming:**

```
registry/repository:tag

Examples:
docker.io/ubuntu:20.04 myregistry.com/myapp:v1.0 nginx:latest
```

---

7. Docker Workflow

Complete Development Workflow

Step 1: Write Application

```
# app.py print("Hello from
Docker!")
```

Step 2: Create Dockerfile

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

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

Step 3: Build Image

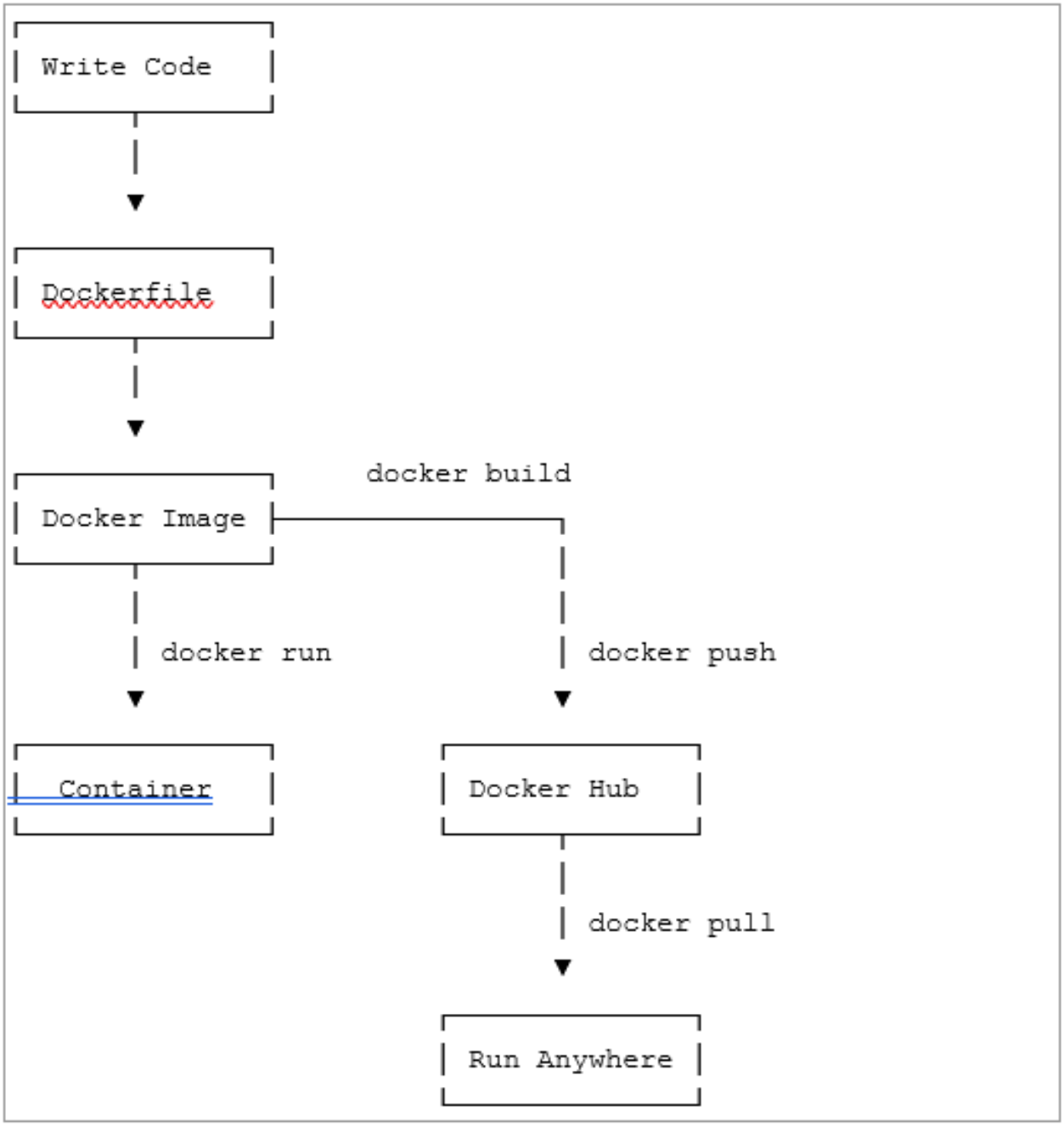
```
docker build -t myapp:v1 .
```

Step 4: Run Container

```
docker run myapp:v1
```

Step 5: Push to Registry (Share)

Visual Workflow

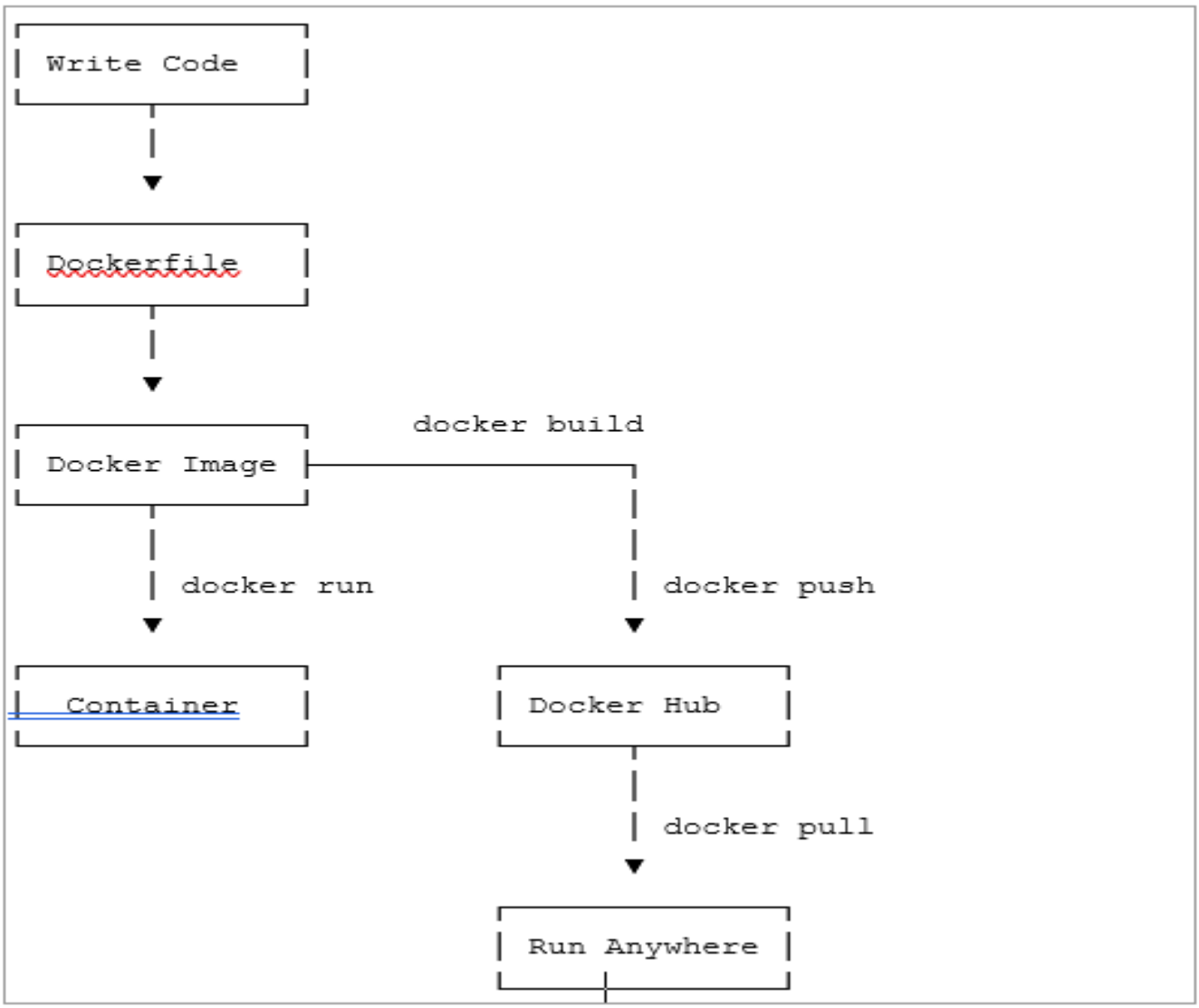



```
docker push username/myapp:v1
```

Step 6: Pull and Run Anywhere

```
docker pull username/myapp:v1
docker run username/myapp:v1
```

Visual Workflow



8. Dockerfile in Deep Detail

Dockerfile Instructions

FROM - Base image

```
FROM ubuntu:20.04
FROM python:3.9-alpine
FROM node:16
```

WORKDIR - Set working directory

```
WORKDIR /app
# All subsequent commands run from /app
```

COPY - Copy files from host to image

```
COPY app.py /app/
COPY . /app/
COPY requirements.txt .
```

ADD - Like COPY but with extras (can extract tar, download URLs)

```
ADD archive.tar.gz /app/
ADD https://example.com/file.zip /app/
```

RUN - Execute commands during build

```
RUN apt-get update
RUN pip install flask
RUN npm install
```

CMD - Default command when container starts

```
CMD ["python", "app.py"]
CMD ["npm", "start"]
# Only one CMD per Dockerfile (last one wins)
```

ENTRYPOINT - Configure container as executable

```
ENTRYPOINT ["python", "app.py"]
# Cannot be overridden easily (more rigid than CMD)
```

CMD vs ENTRYPOINT:

```
# CMD - can be overridden
CMD ["echo", "Hello"]
# docker run myimage          → Hello
# docker run myimage echo Hi  → Hi

# ENTRYPOINT - fixed command
ENTRYPOINT ["echo", "Hello"]
# docker run myimage          → Hello
# docker run myimage World    → Hello World
```

ENV - Set environment variables

```
ENV NODE_ENV=production
ENV PORT=8080
```

EXPOSE - Document which ports container listens on

```
EXPOSE 8080
EXPOSE 3000
```

VOLUME - Create mount point for persistent data

```
VOLUME /data
```

USER - Set user for running commands

```
USER appuser
```

ARG - Build-time variables

```
ARG VERSION=1.0
RUN echo "Building version $VERSION"
```

Complete Real-World Dockerfile Example

```
# Multi-stage build for Python Flask app

# Stage 1: Build stage
FROM python:3.9-slim AS builder

# Set working directory
WORKDIR /app

# Install system dependencies RUN apt-get
update && \     apt-get install -y --no-
install-recommends gcc && \     rm -rf
/var/lib/apt/lists/*

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

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

# Stage 2: Runtime stage
FROM python:3.9-slim

# Create non-root user
RUN useradd -m -u 1000 appuser

# Set working directory
WORKDIR /app

# Copy Python dependencies from builder
COPY --from=builder /root/.local /home/appuser/.local

# Copy application code
COPY --chown=appuser:appuser . .

# Switch to non-root user
USER appuser

# Update PATH
ENV PATH=/home/appuser/.local/bin:$PATH

# Expose port
EXPOSE 5000
# Health check
```

```
HEALTHCHECK --interval=30s --timeout=3s \  CMD curl -f
http://localhost:5000/health || exit 1
# Set environment variables
ENV FLASK_APP=app.py
ENV PYTHONUNBUFFERED=1

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

Best Practices for Dockerfile

1. **Use Specific Tags (Not `latest`)**

```
# Bad
FROM python:latest

# Good
FROM python:3.9-slim
```

2. **Minimize Layers**

```
# Bad - 3 layers
RUN apt-get update
RUN apt-get install -y python3
RUN apt-get clean

# Good - 1 layer RUN apt-get update && \     apt-get install
-y python3 && \     apt-get clean && \     rm -rf
/var/lib/apt/lists/*
```

3. **Order Matters for Caching**

```
# Copy requirements first (changes less often)
COPY requirements.txt .
RUN pip install -r requirements.txt

# Copy code later (changes more often) COPY .
.
```

4. **Use .dockerignore**

```
# .dockerignore file
node_modules
.git
*.log
.env
```

5. Multi-stage Builds (Smaller Images)

```
# Build stage - includes build tools
FROM node:16 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm
install COPY .
.
RUN npm run build

# Runtime stage - only runtime dependencies
FROM node:16-alpine
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/node_modules ./node_modules
CMD ["node", "dist/server.js"]
```

---

9. Docker Images and Layers

Layer System Explained

Each Dockerfile instruction creates a layer:

```
FROM ubuntu:20.04           # Layer 1: Base OS
RUN apt-get update          # Layer 2: Updated packages
RUN apt-get install python # Layer 3: Python installed
COPY app.py /app/           # Layer 4: Application code CMD
["python", "/app/app.py"] # Layer 5: Metadata (no size)
```

Image Structure:

```
Image ID: abc123
├── Layer 5 (CMD metadata)      ← 0 bytes
├── Layer 4 (app.py)            ← 1 KB
├── Layer 3 (python install)    ← 100 MB ├── Layer 2 (apt
update)       ← 50 MB
└── Layer 1 (Ubuntu base)       ← 200 MB
Total: 350 MB
```

Layer Caching

How It Works:

1. Docker checks if instruction changed
2. If unchanged, reuse cached layer
3. If changed, rebuild this and all subsequent layers

Example:

```
FROM python:3.9                # Cached (unchanged)
WORKDIR /app                   # Cached (unchanged)
COPY requirements.txt .        # Cached (unchanged)
RUN pip install -r requirements.txt  # Cached (unchanged)
COPY . .                       # REBUILD (code changed)
CMD ["python", "app.py"]       # REBUILD (after changed layer)
```

Image Commands

```
# List images
docker images

# Remove image
docker rmi
image_name

# View image history
(layers) docker history
image_name

# Inspect image
details docker inspect
image_name

# Tag image docker tag
source_image:tag
target_image:tag

# Save image to tar file
docker save -o myimage.tar
myimage:tag

# Load image from tar
file docker load -i
myimage.tar
```

---

10. Essential Docker Commands

Container Management

Run Container:

```
# Basic run
docker run
ubuntu

# Run with name docker run
--name mycontainer ubuntu

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

# Run with port
mapping docker run -p
8080:80 nginx
# Host port 8080 → Container port 80


# Run with environment variables docker
run -e DB_HOST=localhost -e
DB_PORT=5432 myapp

# Run with volume mount docker run
-v /host/path:/container/path
myapp

# Run interactive
terminal docker run -it
ubuntu /bin/bash

# Run and remove after
exit docker run --rm
ubuntu

# Combined example
docker run -d \
  --name webapp \
  -p 8080:80 \
  -e
ENV=production \
-v
/data:/app/data \
myapp:v1
```

Container Lifecycle:

```
# List running
containers docker ps

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

# Start stopped
container docker start
container_name

# Stop running
container docker stop
container_name

# Restart container
docker restart
container_name

# Pause container
docker pause
container_name

# Unpause container
docker unpause
container_name

# Remove container
docker rm
container_name

# Remove running container
(force) docker rm -f
container_name

# Remove all stopped
containers docker container
prune
```

Interact with Containers:

```
# View logs docker
logs container_name

# Follow logs (real-
time) docker logs -f
container_name

# Execute command in running
container docker exec
container_name ls /app

# Interactive shell in running
container docker exec -it
container_name /bin/bash

# Copy files from container to
host docker cp
container_name:/app/file.txt ./

# Copy files from host to
container docker cp ./file.txt
container_name:/app/

# View container resource
usage docker stats
container_name

# Inspect container
details docker inspect
container_name
```

Image Management

```
# Build image from
Dockerfile docker build -t
myapp:v1 .

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

# Build without cache
docker build --no-cache -
t myapp .

# Pull image from
registry docker pull
ubuntu:20.04

# Push image to
registry docker push
username/myapp:v1

# Tag image docker tag
myapp:v1 myapp:latest

# Search images on Docker
Hub docker search nginx

# Remove unused
images docker image
prune

# Remove all images
docker rmi $(docker
images -q)
```

System Management

```
# View Docker info
docker info

# View Docker
version docker
version

# Clean up everything (unused containers, images,
networks) docker system prune

# Clean up with volumes
docker system prune -a
--volumes

# View disk usage
docker system df
```

11. Docker Networking

Network Types

1. **Bridge (Default)**

- Default network for containers
- Containers can communicate with each other
- Isolated from host network

```
docker run -d --name web nginx docker run -d --
name app myapp
# web and app can talk to each other
```

2. **Host**

- Container uses host's network directly
- No isolation
- Better performance

```
docker run --network host nginx
# nginx binds to host's port 80 directly
```

3. **None**

- No networking
- Completely isolated

```
docker run --network none ubuntu
```

4. **Custom Bridge**

- User-defined networks
- Better isolation and DNS

```
# Create custom network
docker network create
mynetwork

# Run containers on custom network
docker run -d --name db --network
mynetwork postgres docker run -d --name
web --network mynetwork nginx # db and
web can communicate using container
names
```

Network Commands

```
# List networks
docker network
ls

# Create network docker
network create
mynetwork

# Inspect network
docker network inspect
mynetwork

# Connect container to network
docker network connect mynetwork
container_name

# Disconnect container from network
docker network disconnect mynetwork
container_name

# Remove network
docker network rm
mynetwork

# Remove unused
networks docker network
prune
```

Container Communication Example

```
# Create custom network
docker network create
app-network

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

# Run application (can connect to 'database'
by name) docker run -d \
  --name webapp \
  --network app-network \
  -p 8080:80 \
  -e DB_HOST=database
\   myapp
```

Port Mapping

```
# Map single port
docker run -p
8080:80 nginx
# localhost:8080 → container:80


# Map all ports
docker run -P
nginx
# Automatically map all EXPOSE'd ports to random host ports


# Map to specific interface
docker run -p
127.0.0.1:8080:80 nginx
# Only accessible from localhost


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

---

12. Docker Volumes (Persistent Data)

Why Volumes?

Problem: Container data is ephemeral

```
docker run -d --name db postgres # Write data
to database docker rm db
# Data is LOST!
```

Solution: Volumes persist data outside containers

Volume Types

1. **Named Volumes (Managed by Docker)**

```
# Create volume
docker volume create
mydata

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

# Data persists even if container is removed
docker rm db docker run -d --name db2 -v
mydata:/var/lib/postgresql/data postgres
# Data still there!
```

2. **Bind Mounts (Host Directory)**

```
# Mount host directory into container
docker run -d \
  -v /host/path:/container/path \
myapp

# Example: Mount current directory
docker run -v $(pwd):/app myapp
```

3. tmpfs Mounts (Memory)

```
# Data stored in memory (fast, but lost on stop)
docker run --tmpfs /app/cache myapp
```

Volume Commands

```
# List volumes
docker volume
ls

# Create volume:
docker volume create
myvolume

# Inspect volume
docker volume inspect
myvolume

# Remove volume
docker volume rm
myvolume

# Remove unused
volumes docker volume
prune

# View volume location on host
docker volume inspect myvolume | gep
Mountpoint
```

Real-World Volume Example

```
# Development: Bind mount for live code
updates docker run -d \
  --name dev-server \
  -v $(pwd)/src:/app/src \
  -p 3000:3000 \
node-app

# Edit files on host → Changes immediately reflected in container


# Production: Named volume for
database docker run -d \
  --name prod-db \
  -v prod-
data:/var/lib/mysql \   -e
MYSQL_ROOT_PASSWORD=secret
\   mysql

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

13. Docker Compose

What is Docker Compose?

Purpose: Define and run multi-container applications

Problem Without Compose:

```
# Manually start each service docker network create
myapp docker run -d --name db --network myapp postgres
docker run -d --name redis --network myapp redis docker
run -d --name web --network myapp -p 8080:80 myapp
# Tedious! 
```

Solution With Compose:

```
# docker-compose.yml
version: '3.8'
services:   db:
    image: postgres
redis:
    image: redis   web:
    image: myapp
ports:       -
"8080:80"
```

```
docker-compose up
# All services start with one command!
```

Complete docker-compose.yml Example

```
version: '3.8'

services:
  # Database service
database:
    image: postgres:13
container_name: my_postgres
environment:
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: secret
POSTGRES_DB: myapp     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

  # Redis cache   cache:
    image: redis:alpine
container_name: my_redis     networks:
- backend     command: redis-server --
appendonly yes     volumes:
-    cache-data:/data


  # Web application
web:     build:
      context: ./app
dockerfile: Dockerfile
args:
        VERSION: 1.0
container_name: my_webapp
ports:
-    "8080:80"
-    "8443:443"
```

```
    environment:
-    NODE_ENV=production
-    DB_HOST=database       -
REDIS_HOST=cache     depends_on:
database:
        condition: service_healthy
cache:
        condition: service_started
volumes:
-    ./app:/usr/src/app       -
/usr/src/app/node_modules
networks:
-    frontend       - backend
restart: unless-stopped

  # Nginx reverse proxy
nginx:
    image: nginx:alpine
container_name: my_nginx
ports:       - "80:80"
volumes:
-    ./nginx.conf:/etc/nginx/ngi
nx.conf:ro     depends_on:
- web     networks:       -
frontend

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


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

Docker Compose Commands

```
# Start all services
docker-compose up

# Start in background
docker-compose up -d

# Build images before starting
docker-compose up --build

# Stop all services
docker-compose down

# Stop and remove volumes
docker-compose down -v

# View running services
docker-compose ps

# View logs docker-
compose logs

# Follow logs
docker-compose logs
-f

# Execute command in service
docker-compose exec web bash

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

# Restart services
docker-compose restart

# Pause services
docker-compose pause

# Unpause services
docker-compose unpause
```

Real-World Full Stack Example

```
# docker-compose.yml for a full web app

version: '3.8'

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

  # Backend (Python Flask)
backend:
    build: ./backend
ports:
-    "5000:5000"     environment:
-    DATABASE_URL=postgresql://user:pass@db:5432/myapp
- REDIS_URL=redis://redis:6379     depends_on:       -
db
-    redis

  # Database
db:
    image: postgres:13
environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD:
pass       POSTGRES_DB:
myapp     volumes:
-    postgres-data:/var/lib/postgresql/data

  # Cache
redis:
    image: redis:alpine
volumes:
-    redis-data:/data
```

```
  # Database admin UI
adminer:
    image: adminer
ports:
      -
"8080:8080"
depends_on:
- db

volumes:
postgres-data:
redis-data:
```

Usage:

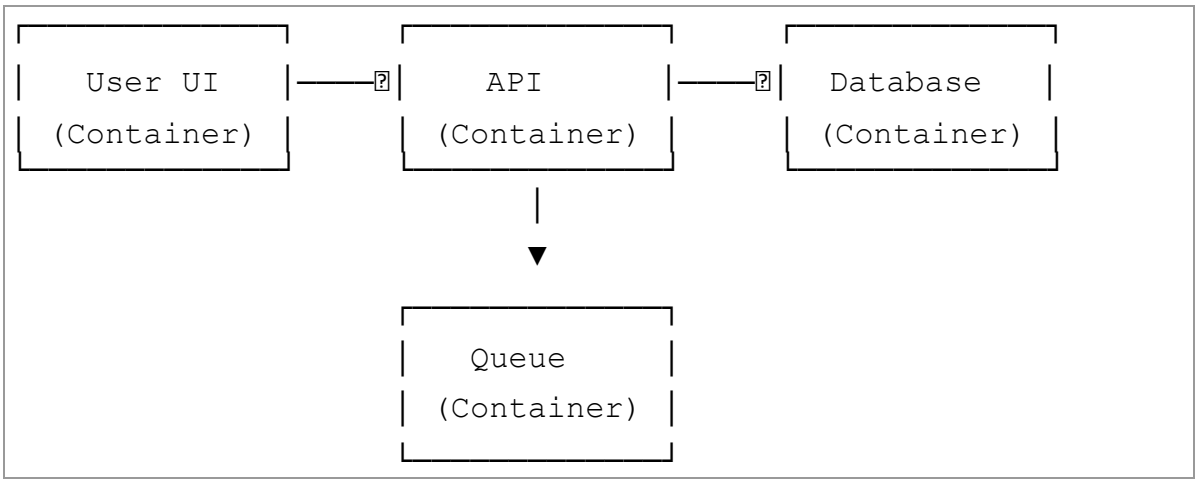
```
# Start everything docker-compose up -d

# Access:
# Frontend: http://localhost:3000 # Backend:
http://localhost:5000
# DB Admin: http://localhost:8080
```

---

14. Real-World Use Cases

Use Case 1: Microservices Architecture



Each service runs in its own container, scales independently.

Use Case 2: CI/CD Pipeline

```
Code Push → Docker Build → Run Tests in Container →
Push to Registry → Deploy to Production
```

```
# .gitlab-ci.yml
example test:
  image: python:3.9
script:
- pip install -r requirements.txt
- pytest


build:
script:
- docker build -t myapp:$CI_COMMIT_SHA .
- docker push myapp:$CI_COMMIT_SHA
```

Use Case 3: Development Environment

```
# Instead of installing Python, Node, PostgreSQL locally...
docker-compose up
# Entire dev environment ready!
```

Benefits:

- Same environment for all developers
- Easy onboarding for new developers
- No "works on my machine" issues

Use Case 4: Database Testing

```
# Spin up test database docker run -d -p
5432:5432 -e POSTGRES_PASSWORD=test postgres

# Run tests
pytest

# Destroy database
docker rm -f
postgres

# Fresh, clean state every time!
```

Use Case 5: Legacy Application Modernization

```
# Containerize old PHP app
FROM php:5.6-apache
COPY . /var/www/html/
RUN docker-php-ext-install mysql
```

Now runs anywhere, even though PHP 5.6 is deprecated.

---

15. Best Practices

Security

1. Don't Run as Root

```
# Bad
USER root

# Good
RUN useradd -m appuser
USER appuser
```

2. Use Specific Tags

```
# Bad
FROM python:latest

# Good
FROM python:3.9.7-slim
```

3. Scan Images for Vulnerabilities

```
docker scan myimage:v1
```

4. Don't Store Secrets in Images

```
# Bad
ENV DB_PASSWORD=secret123

# Good - use environment variables at runtime docker run -e
DB_PASSWORD=$DB_PASSWORD myapp
```

5. Use .dockerignore

```
.git .env node_modules
*.log
```

Performance

1. Minimize Layers

```
# Combine RUN commands RUN apt-get update
&& \     apt-get install -y package1
package2 && \     apt-get clean
```

2. Use Multi-Stage Builds

```
FROM golang:1.17 AS builder
WORKDIR /app
COPY . .
RUN go build -o app

FROM alpine:latest
COPY --from=builder /app/app /app
CMD ["/app"]
```

3. Leverage Cache

```
# Copy dependencies first
COPY requirements.txt .
RUN pip install -r requirements.txt

# Copy code last (changes more often) COPY . .
```

4. Use Smaller Base Images

```
# python:3.9        → 900 MB
# python:3.9-slim   → 120 MB
# python:3.9-alpine →  45 MB

FROM python:3.9-alpine
```

Maintenance

1. Health Checks

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

2. Logging

```
# Log to stdout/stderr (Docker captures them)
CMD ["python", "-u", "app.py"]
```

3. Resource Limits

```
docker run -m 512m --cpus=1 myapp
```

4. Clean Up Regularly

```
# Remove unused containers, images, networks
docker system prune -a
```

Summary Cheat Sheet

Key Concepts

```
Container = Isolated running instance
Image = Blueprint/template for containers
Dockerfile = Instructions to build image
Registry = Storage for images
Volume = Persistent data storage
Network = Container communication
Compose = Multi-container orchestration
```

Most Common Commands

```
# Images docker
build -t name .
docker pull name
docker push name

# Containers docker
run -d -p 8080:80 name
docker ps docker logs
container docker exec
-it container bash
docker stop container
docker rm container

# Compose docker-
compose up -d
docker-compose
down docker-
compose logs -f
```

When to Use Docker

Microservices CI/CD pipelines Development environments Application portability Testing isolation

When NOT to Use Docker

Need different OS kernels (use VMs) GUI applications (complex) Maximum performance critical (small overhead

exists) Simple static sites (might be overkill)

Docker revolutionizes how we build, ship, and run applications. Master these concepts and you'll understand modern

software deployment!