Docker Images, Dockerfile & Building

Table of Contents

What is a Dockerfile?

Definition

A **Dockerfile** is a text file containing a set of instructions that Docker uses to build an image automatically. It's like a recipe for creating a Docker image.

Basic Structure

```dockerfile
# Comment
INSTRUCTION arguments

# Example:
FROM ubuntu:22.04
RUN apt-get update
COPY app.py /app/
CMD ["python", "/app/app.py"]
```

Why Use Dockerfiles?

```
 Manual Way (Tedious):
1. Start container
2. Install software manually
3. Configure manually
4. Commit container to image
5. Repeat for every change

 Dockerfile Way (Automated):
1. Write Dockerfile once
2. Build image: docker build
3. Rebuild anytime with one command
4. Share Dockerfile with team
```

```
5. Version control with Git
```

What is a Docker Image?

Definition

A **Docker Image** is a lightweight, standalone, executable package that includes everything needed to run a piece of software: - Code - Runtime - Libraries - Environment variables - Configuration files

Image Characteristics

```
Read-Only:  Images cannot be modified
Layered:    Built from layers stacked on top of each other
Portable:   Can run on any system with Docker
Versioned:  Tagged with versions (e.g., nginx:1.25)
Shareable:  Can be pushed to registries and shared
```

Image Naming Convention

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

Examples:
nginx                           # Official image, latest tag
nginx:1.25                      # Official image, specific version
ubuntu:22.04                    # Official Ubuntu, version 22.04
docker.io/library/nginx:latest # Full name with registry
myusername/myapp:1.0          # User image with version
myregistry.com/myapp:latest   # Private registry image
```

Difference Between Dockerfile and Image

Visual Comparison

```

                       DOCKERFILE
  (Blueprint/Recipe - Text file)

  FROM ubuntu:22.04
  RUN apt-get update
  RUN apt-get install -y python3
  COPY app.py /app/
  CMD ["python3", "/app/app.py"]
```

```
                         docker build


                   DOCKER IMAGE
(Built artifact - Binary)

Layer 5: CMD
Layer 4: COPY app.py
Layer 3: RUN apt install python3
Layer 2: RUN apt update
Layer 1: FROM ubuntu:22.04


                         docker run


                   CONTAINER
(Running instance)
Your application is running here!
```

Detailed Comparison

Aspect	Dockerfile	Docker Image
Type	Text file	Binary file
Purpose	Instructions to build image	Template to create containers
Format	Human-readable	Machine-readable
Editable	Yes, with text editor	No, must rebuild
Size	Few KBs	MBs to GBs
Shareable	Via Git, text	Via registries
Example	`Dockerfile`	`nginx:latest`

Analogy

```
Dockerfile = Recipe (how to make a cake)
Image      = Cake mold (template)
Container  = Actual cake (you can eat it!)
```

How to Create a Dockerfile

Step-by-Step Guide

Example 1: Simple Python Application   Project Structure:

```
my-python-app/
   Dockerfile
   app.py
   requirements.txt
```

app.py:

```python
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return "Hello from Docker!"

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

requirements.txt:

```
flask==2.3.0
```

Dockerfile:

```dockerfile
# Use official Python runtime as base image
FROM python:3.11-slim

# Set working directory in container
WORKDIR /app

# Copy requirements file
COPY requirements.txt .

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

# Copy application code
COPY app.py .

# Expose port
EXPOSE 5000

# Define command to run the application
CMD ["python", "app.py"]
```

Example 2: Node.js Application **Project Structure**:

```
my-node-app/
  Dockerfile
  package.json
  package-lock.json
  server.js
```

server.js:

```
const express = require('express');
const app = express();
const PORT = 3000;

app.get('/', (req, res) => {
    res.send('Hello from Node.js in Docker!');
});

app.listen(PORT, () => {
    console.log(`Server running on port ${PORT}`);
});
```

package.json:

```
{
  "name": "my-node-app",
  "version": "1.0.0",
  "main": "server.js",
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

Dockerfile:

```
# Use official Node.js runtime
FROM node:18-alpine

# Set working directory
WORKDIR /usr/src/app

# Copy package files
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy application code
COPY . .
```

```
# Expose port
EXPOSE 3000

# Start application
CMD ["node", "server.js"]
```

Example 3: React Web Application Dockerfile:

```
# Build stage
FROM node:18-alpine AS builder

WORKDIR /app

# Copy package files
COPY package*.json ./

# Install dependencies
RUN npm ci

# Copy source code
COPY . .

# Build application
RUN npm run build

# Production stage
FROM nginx:alpine

# Copy built files from builder stage
COPY --from=builder /app/build /usr/share/nginx/html

# Copy custom nginx config (optional)
# COPY nginx.conf /etc/nginx/conf.d/default.conf

# Expose port
EXPOSE 80

# Start nginx
CMD ["nginx", "-g", "daemon off;"]
```

Dockerfile Instructions

FROM - Base Image

```
# Official image
FROM ubuntu:22.04

# Specific version
FROM node:18.17.0

# Alpine (smaller size)
FROM python:3.11-alpine

# Multiple FROM for multi-stage builds
FROM node:18 AS builder
FROM nginx:alpine AS production
```

WORKDIR - Set Working Directory

```
# Set working directory
WORKDIR /app

# All subsequent commands run from /app
COPY . .
RUN npm install
# These run in /app directory

# Can set multiple times
WORKDIR /app
WORKDIR backend
# Now in /app/backend
```

COPY - Copy Files

```
# Copy single file
COPY app.py /app/

# Copy multiple files
COPY app.py config.json /app/

# Copy directory
COPY ./src /app/src/

# Copy with wildcard
COPY *.py /app/

# Copy everything (except .dockerignore items)
```

```
COPY . /app/

# Copy and rename
COPY config.json /app/config-prod.json
```

ADD - Copy and Extract

```
# Similar to COPY
ADD app.py /app/

# Can download from URL
ADD https://example.com/file.tar.gz /tmp/

# Automatically extracts tar files
ADD archive.tar.gz /app/
# Contents of archive.tar.gz extracted to /app/

#  Recommendation: Use COPY unless you need ADD's special features
```

RUN - Execute Commands

```
# Shell form (runs in /bin/sh -c)
RUN apt-get update
RUN apt-get install -y python3

# Exec form (doesn't invoke shell)
RUN ["apt-get", "update"]

# Multiple commands (preferred - creates one layer)
RUN apt-get update && \
    apt-get install -y \
        python3 \
        python3-pip \
        git && \
    rm -rf /var/lib/apt/lists/*

# Install Python packages
RUN pip install flask redis celery

# Create directories
RUN mkdir -p /app/logs /app/data

# Download and install
RUN wget https://example.com/tool.tar.gz && \
    tar -xzf tool.tar.gz && \
    rm tool.tar.gz
```

CMD - Default Command

```
# Exec form (preferred)
CMD ["python", "app.py"]
CMD ["node", "server.js"]
CMD ["nginx", "-g", "daemon off;"]

# Shell form
CMD python app.py

# Provide default arguments to ENTRYPOINT
CMD ["--port", "8080"]

#   Only one CMD instruction allowed
# Last CMD in Dockerfile is used
```

ENTRYPOINT - Configure Container as Executable

```
# Exec form
ENTRYPOINT ["python", "app.py"]

# Combined with CMD for default args
ENTRYPOINT ["python", "app.py"]
CMD ["--port", "8080"]

# Run container: docker run myapp
# Executes: python app.py --port 8080

# Run with custom args: docker run myapp --port 9000
# Executes: python app.py --port 9000

# Real example: Python CLI tool
ENTRYPOINT ["python", "cli.py"]
CMD ["--help"]
```

ENV - Environment Variables

```
# Set environment variable
ENV NODE_ENV=production
ENV PORT=3000
ENV DATABASE_URL=postgresql://localhost/mydb

# Multiple variables
ENV NODE_ENV=production \
    PORT=3000 \
    LOG_LEVEL=info
```

```
# Use in subsequent instructions
ENV APP_HOME=/app
WORKDIR $APP_HOME
COPY . $APP_HOME

# Override at runtime
# docker run -e NODE_ENV=development myapp
```

EXPOSE - Document Ports

```
# Single port
EXPOSE 80

# Multiple ports
EXPOSE 80 443

# With protocol
EXPOSE 80/tcp
EXPOSE 53/udp

#  EXPOSE doesn't actually publish ports
# It's documentation for docker run -p
# docker run -p 8080:80 myapp
```

VOLUME - Create Mount Point

```
# Create mount point for persistent data
VOLUME /app/data

# Multiple volumes
VOLUME ["/app/data", "/app/logs"]

# Example: Database
FROM postgres:14
VOLUME /var/lib/postgresql/data
```

USER - Set User

```
# Run as non-root user (security best practice)
FROM ubuntu:22.04

# Create user
RUN useradd -m -u 1000 appuser

# Switch to user
USER appuser
```

```
# All subsequent commands run as appuser
WORKDIR /home/appuser/app
COPY --chown=appuser:appuser . .

# Can switch back to root if needed
USER root
RUN apt-get update
USER appuser
```

ARG - Build Arguments

```
# Define build argument
ARG VERSION=latest
ARG PORT=8080

# Use in Dockerfile
FROM node:${VERSION}
EXPOSE ${PORT}

# Build with custom values
# docker build --build-arg VERSION=18 --build-arg PORT=3000 .

# Real example: Multi-environment builds
ARG ENVIRONMENT=production
ENV NODE_ENV=${ENVIRONMENT}

# Available only during build (unlike ENV)
```

LABEL - Add Metadata

```
# Add metadata to image
LABEL version="1.0"
LABEL description="My awesome application"
LABEL maintainer="your-email@example.com"

# Multiple labels
LABEL version="1.0" \
      description="My app" \
      maintainer="me@example.com"

# View labels
# docker image inspect myapp
```

HEALTHCHECK - Health Check

```
# Check if container is healthy
HEALTHCHECK --interval=30s --timeout=3s \
  CMD curl -f http://localhost/ || exit 1

# More complex check
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
  CMD python healthcheck.py || exit 1

# Disable healthcheck from base image
HEALTHCHECK NONE

# Check health status
# docker ps  # Shows "healthy" or "unhealthy"
```

SHELL - Change Default Shell

```
# Change shell (Windows containers)
SHELL ["powershell", "-command"]

# Linux containers
SHELL ["/bin/bash", "-c"]

# Use custom shell for RUN commands
SHELL ["/bin/bash", "-o", "pipefail", "-c"]
RUN wget -O- https://example.com | tar xz
```

ONBUILD - Trigger Instructions

```
# Instructions that run when image is used as base

# In base-image Dockerfile:
FROM node:18
ONBUILD COPY package*.json ./
ONBUILD RUN npm install
ONBUILD COPY . .

# In application Dockerfile:
FROM base-image  # ONBUILD instructions execute here

# Useful for creating base images for teams
```

How to Build Docker Images

Basic Build

```
# Build from Dockerfile in current directory
docker build -t myapp:1.0 .
```

```
# Output:
# [+] Building 12.3s (10/10) FINISHED
#  => [internal] load build definition from Dockerfile
#  => [internal] load .dockerignore
#  => [internal] load metadata for docker.io/library/python:3.11
#  => [1/5] FROM docker.io/library/python:3.11
#  => [2/5] WORKDIR /app
#  => [3/5] COPY requirements.txt .
#  => [4/5] RUN pip install -r requirements.txt
#  => [5/5] COPY app.py .
#  => exporting to image
#  => => naming to docker.io/library/myapp:1.0
```

Build Options

```
# Build with custom Dockerfile name
docker build -f Dockerfile.prod -t myapp:prod .

# Build with no cache (force rebuild)
docker build --no-cache -t myapp:1.0 .

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

# Build and show progress
docker build --progress=plain -t myapp .

# Build for specific platform
docker build --platform linux/amd64 -t myapp .
docker build --platform linux/arm64 -t myapp .

# Build with labels
docker build --label version=1.0 --label env=production -t myapp .

# Build with target stage (multi-stage builds)
docker build --target builder -t myapp:builder .

# Build with squash (experimental)
docker build --squash -t myapp .
```

BuildKit (Enhanced Build Engine)

```
# Enable BuildKit
export DOCKER_BUILDKIT=1
```

```
# Or per command
DOCKER_BUILDKIT=1 docker build -t myapp .

# Features:
#   Parallel builds
#   Better caching
#   Secrets management
#   SSH forwarding

# BuildKit example with secrets
docker build --secret id=mysecret,src=secret.txt -t myapp .

# In Dockerfile:
# RUN --mount=type=secret,id=mysecret \
#     cat /run/secrets/mysecret
```

.dockerignore File

Create `.dockerignore` to exclude files from build context:

```
# .dockerignore
node_modules
npm-debug.log
Dockerfile
.dockerignore
.git
.gitignore
README.md
.env
.vscode
**/*.log
**/*.md
!README.md
dist
coverage
.pytest_cache
__pycache__
*.pyc
.DS_Store
```

Building Real Projects

Example 1: Full Stack React + Node.js App

Project Structure:

```
fullstack-app/
  backend/
     Dockerfile
     package.json
     server.js
  frontend/
     Dockerfile
     package.json
     public/
     src/
  docker-compose.yml
```

backend/Dockerfile:

```
FROM node:18-alpine

WORKDIR /app

# Install dependencies
COPY package*.json ./
RUN npm ci --only=production

# Copy source
COPY . .

# Expose API port
EXPOSE 5000

# Health check
HEALTHCHECK --interval=30s --timeout=3s \
  CMD node healthcheck.js || exit 1

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

frontend/Dockerfile:

```
# Build stage
FROM node:18-alpine AS build

WORKDIR /app

COPY package*.json ./
RUN npm ci

COPY . .
RUN npm run build
```

```
# Production stage
FROM nginx:alpine

# Copy built files
COPY --from=build /app/build /usr/share/nginx/html

# Copy nginx config
COPY nginx.conf /etc/nginx/conf.d/default.conf

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

Example 2: Python FastAPI Application

Dockerfile:

```
FROM python:3.11-slim

WORKDIR /app

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

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

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

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

# Switch to non-root user
USER appuser

# Expose port
EXPOSE 8000

# Health check
HEALTHCHECK --interval=30s --timeout=3s \
  CMD python -c "import requests; requests.get('http://localhost:8000/health')" || exit 1

# Start application
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Example 3: Java Spring Boot Application

Dockerfile:

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

WORKDIR /app

# Copy pom.xml and download dependencies (cached)
COPY pom.xml .
RUN mvn dependency:go-offline

# Copy source and build
COPY src ./src
RUN mvn package -DskipTests

# Runtime stage
FROM openjdk:17-jdk-slim

WORKDIR /app

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

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

EXPOSE 8080

# Health check
HEALTHCHECK --interval=30s --timeout=3s \
  CMD java -cp app.jar org.springframework.boot.SpringApplication --help || exit 1

# Run application
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Best Practices

1. Use Appropriate Base Images

```
#  Bad: Large base image
FROM ubuntu:22.04  # ~77MB

#  Good: Minimal base image
FROM python:3.11-slim  # ~45MB

#  Better: Alpine Linux
FROM python:3.11-alpine  # ~17MB
```

2. Minimize Layers

```
#  Bad: Multiple RUN commands (multiple layers)
RUN apt-get update
RUN apt-get install -y python3
RUN apt-get install -y python3-pip
RUN apt-get clean

#  Good: Combined RUN command (one layer)
RUN apt-get update && \
    apt-get install -y \
        python3 \
        python3-pip && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*
```

3. Order Instructions by Change Frequency

```
#  Good: Least changing first
FROM node:18-alpine

WORKDIR /app

# Dependencies change less frequently
COPY package*.json ./
RUN npm install

# Source code changes frequently
COPY . .

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

4. Use .dockerignore

```
# Exclude unnecessary files
```

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

5. Run as Non-Root User

```dockerfile
#  Security best practice
FROM node:18-alpine

WORKDIR /app

# Install as root
COPY package*.json ./
RUN npm install

# Create and switch to non-root user
RUN addgroup -g 1000 appuser && \
    adduser -D -u 1000 -G appuser appuser

USER appuser

COPY --chown=appuser:appuser . .

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

6. Use Multi-Stage Builds

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

# Production stage (only runtime needed)
FROM node:18-alpine
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/node_modules ./node_modules
CMD ["node", "dist/server.js"]
```

7. Add Health Checks

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

8. Use Specific Tags

```
#   Bad: Latest tag (unpredictable)
FROM node:latest

#   Good: Specific version
FROM node:18.17.0-alpine
```

Multi-Stage Builds

Why Multi-Stage Builds?

```
Problem: Build tools bloat production images
Solution: Use one stage to build, another for runtime

Benefits:
  Smaller production images
  Separate build and runtime dependencies
  Better security (fewer packages in production)
```

Example 1: Go Application

```
# Build stage
FROM golang:1.21 AS builder

WORKDIR /app

# Download dependencies
COPY go.mod go.sum ./
RUN go mod download

# Copy source and build
COPY . .
RUN CGO_ENABLED=0 GOOS=linux go build -o myapp .

# Final stage (minimal image)
FROM alpine:latest

# Add ca-certificates for HTTPS
RUN apk --no-cache add ca-certificates
```

```
WORKDIR /root/

# Copy only the binary
COPY --from=builder /app/myapp .

EXPOSE 8080

CMD ["./myapp"]
```

Result:

```
builder stage: 1.2GB
final image:   15MB  (80x smaller!)
```

Example 2: React Application

```
# Stage 1: Build React app
FROM node:18-alpine AS build

WORKDIR /app

COPY package*.json ./
RUN npm ci

COPY . .
RUN npm run build

# Stage 2: Serve with Nginx
FROM nginx:alpine

# Copy built files
COPY --from=build /app/build /usr/share/nginx/html

# Copy nginx configuration
COPY nginx.conf /etc/nginx/conf.d/default.conf

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

Example 3: Multiple Stages

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

```
# Dependencies stage
FROM base AS dependencies
RUN npm ci --only=production
RUN cp -R node_modules /prod_node_modules
RUN npm ci

# Build stage
FROM base AS build
COPY --from=dependencies /app/node_modules ./node_modules
COPY . .
RUN npm run build

# Test stage
FROM build AS test
RUN npm run test

# Production stage
FROM base AS production
COPY --from=dependencies /prod_node_modules ./node_modules
COPY --from=build /app/dist ./dist
USER node
CMD ["node", "dist/server.js"]
```

Build specific stage:

```
# Build and run tests
docker build --target test -t myapp:test .

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

Key Takeaways

1. **Dockerfile** is a recipe for building images
2. **Docker Image** is the built template for containers
3. **Use appropriate base images** (alpine for smaller size)
4. **Optimize layer caching** (order matters!)
5. **Multi-stage builds** reduce final image size dramatically
6. **Security**: Run as non-root, scan for vulnerabilities
7. **Best practices**: .dockerignore, health checks, specific tags

---

What's Next?

- Creating DEMO projects with Docker

- Working with Docker registries
- Managing containers in production
- Docker Compose for multi-container applications

Master Dockerfiles, master containerization!