📄 02-docker-images-and-dockerfile.md

Docker Images and Dockerfile

Understanding Docker Images

A Docker image is a lightweight, standalone, executable package that contains everything needed to run an application:

- Code
- Runtime
- System tools and libraries
- Environment variables
- Configuration files

Image Structure

Docker images are built in **layers**, where each layer represents a set of file system changes:

```
Layer 4: Your application code
Layer 3: Dependencies installed (pip, npm, etc.)
Layer 2: Runtime environment (Python, Node.js, etc.)
Layer 1: Base OS (Ubuntu, Alpine, etc.)
```

Introduction to Dockerfile

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

Basic Dockerfile Structure

```
# Base image
FROM ubuntu:22.04

# Set working directory
WORKDIR /app

# Copy files
COPY . .
```

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

# Set environment variables
ENV APP_PORT=8000

# Expose port
EXPOSE 8000

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

Dockerfile Instructions

FROM - Base Image

Specifies the base image for your Docker image.

```
# Official Python image
FROM python:3.11-slim

# Or Ubuntu with Python installed
FROM ubuntu:22.04

# Or lightweight Alpine Linux
FROM alpine:latest
```

Output: Creates the first layer of the image

WORKDIR - Working Directory

Sets the working directory inside the container.

```
WORKDIR /app

# All subsequent commands execute in /app
RUN pip install flask
COPY . .
```

Output: When container starts, current directory is `/app`

RUN - Execute Commands

Executes commands during image build time.

```
# Install packages
RUN apt-get update && apt-get install -y \
    curl \
    wget \
    git

# Install Python packages
RUN pip install flask requests numpy

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

Output: Each RUN creates a new layer

COPY and ADD - Copy Files

Copies files from host to container.

```
# Copy specific file
COPY requirements.txt /app/

# Copy entire directory
COPY . /app/

# ADD can also extract tar files
ADD archive.tar.gz /app/
```

Output: Files are added to the image layer

ENV - Environment Variables

Sets environment variables in the container.

```
# Single variable
ENV PYTHONUNBUFFERED=1

# Multiple variables
ENV DB_HOST=localhost \
    DB_PORT=5432 \
    DB_USER=admin
```

Output: Variables available in container at runtime

EXPOSE - Port Exposure

Documents which ports the container listens on.

```
EXPOSE 8000
EXPOSE 3000 5000
```

Note: This doesn't actually publish the port; you must use `-p` flag when running

CMD - Default Command

Specifies the default command to run when container starts.

```
# Exec form (preferred)
CMD ["python", "app.py"]

# Shell form
CMD python app.py

# Set default parameters
CMD ["nginx", "-g", "daemon off;"]
```

Output: Container executes this command on startup

ENTRYPOINT - Override Command

Configures a container to run as an executable.

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

# Now running: docker run myapp arg1 arg2
# Executes: python app.py arg1 arg2
```

USER - Run as User

Specifies which user the container should run as.

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

# Switch to that user
USER appuser

# All subsequent instructions run as appuser
CMD ["python", "app.py"]
```

ARG - Build Arguments

Variables available only during build time.

```
ARG NODE_VERSION=18
ARG BUILD_DATE

FROM node:${NODE_VERSION}-alpine
ENV BUILD_DATE=${BUILD_DATE}
```

Building Docker Images

Basic Build

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

# Output:
# [1/4] FROM ubuntu:22.04
# [2/4] WORKDIR /app
# [3/4] COPY . .
# [4/4] RUN apt-get update && apt-get install -y python3
# => exporting to image
# => writing image sha256:abc123def456...
# => naming to docker.io/library/myapp:1.0
# Successfully tagged myapp:1.0
```

Build with Custom Dockerfile

```
docker build -t myapp:1.0 -f Dockerfile.prod .
```

Build Arguments

```
docker build \
  --build-arg NODE_VERSION=18 \
  --build-arg BUILD_DATE=$(date -u +'%Y-%m-%dT%H:%M:%SZ') \
  -t myapp:1.0 .
```

Build from Git Repository

```
docker build https://github.com/username/repo.git
```

```
# Build from specific branch
docker build -t myapp:latest https://github.com/username/repo.git#main
```

Real-World Examples

Python Flask Application

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

# Set working directory
WORKDIR /app

# Copy requirements
COPY requirements.txt .

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

# Copy application code
COPY . .

# Expose port
EXPOSE 5000

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

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

Node.js Application

```
# Multi-stage build for Node.js
FROM node:18-alpine AS development

WORKDIR /app

# Copy package files
COPY package*.json ./

# Install dependencies
RUN npm ci
```

```
# Copy source code
COPY . .

# Build stage
FROM node:18-alpine

WORKDIR /app

# Copy package files
COPY package*.json ./

# Install only production dependencies
RUN npm ci --only=production

# Copy built application from development stage
COPY --from=development /app/dist ./dist

EXPOSE 3000

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

Go Application

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

WORKDIR /build

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

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

# Final stage
FROM alpine:latest

RUN apk --no-cache add ca-certificates

WORKDIR /root/

COPY --from=builder /build/app .

EXPOSE 8080

CMD ["./app"]
```

Image Management

List Images

```
docker images

# Output:
# REPOSITORY          TAG       IMAGE ID       CREATED        SIZE
# myapp               1.0       abc123def456   2 hours ago    456MB
# nginx               latest    605c77e624dd   3 days ago    187MB
# python              3.11      1234567890ab   1 week ago    912MB
```

View Image History

```
docker history myapp:1.0

# Output:
# IMAGE               CREATED             CREATED BY
# abc123def456        2 hours ago         /bin/sh -c python app.py
# def456ghi789        2 hours ago         /bin/sh -c apt-get update && apt-get
# ghi789jkl012        2 hours ago         WORKDIR /app
# jkl012mno345        2 hours ago         COPY . .
# ubuntu:22.04        1 week ago
```

Tag an Image

```
# Create a new tag for existing image
docker tag myapp:1.0 myapp:latest

# Tag for registry
docker tag myapp:1.0 myregistry.com/myapp:1.0
```

Remove Images

```
# Delete specific image
docker rmi myapp:1.0

# Force delete
docker rmi -f myapp:1.0

# Remove all unused images
docker image prune
```

```
# Remove all unused images including dangling layers
docker image prune -a
```

Image Inspection

Inspect Image Details

```
docker inspect myapp:1.0

# Output (abbreviated):
# [
#     {
#         "Id": "sha256:abc123def456...",
#         "RepoTags": ["myapp:1.0"],
#         "RepoDigests": ["myapp@sha256:1234567..."],
#         "Created": "2024-02-17T14:30:00...",
#         "DockerVersion": "26.0.0",
#         "Config": {
#             "Hostname": "",
#             "Domainname": "",
#             "User": "",
#             "AttachStdin": false,
#             "AttachStdout": false,
#             "AttachStderr": false,
#             "Tty": false,
#             "OpenStdin": false,
#             "StdinOnce": false,
#             "Env": ["PATH=/usr/local/sbin:/usr/local/bin:..."],
#             "Cmd": ["python", "app.py"],
#             "Image": "def456ghi789...",
#             "ExposedPorts": {"5000/tcp": {}},
#             "Volumes": null,
#             "WorkingDir": "/app",
#             "Entrypoint": null
#         }
#     }
# ]
```

Export and Import Images

```
# Save image to tar file
docker save myapp:1.0 -o myapp-1.0.tar

# Load image from tar file
```

```
docker load -i myapp-1.0.tar

# Output:
# Loaded image: myapp:1.0
```

Best Practices for Dockerfiles

1. **Use specific base image versions** - Avoid `latest` tag
2. **Minimize layers** - Combine RUN commands with `&&`
3. **Order instructions** - Place frequently changing instructions last
4. **Use .dockerignore** - Exclude unnecessary files
5. **Multi-stage builds** - Keep final image small
6. **Non-root user** - Run applications as unprivileged user

.dockerignore Example

```
node_modules
npm-debug.log
.git
.gitignore
.env
.DS_Store
dist
build
coverage
.vscode
.idea
```

Next Steps

- Learn about running and managing Docker Containers
- Explore Networking and Storage