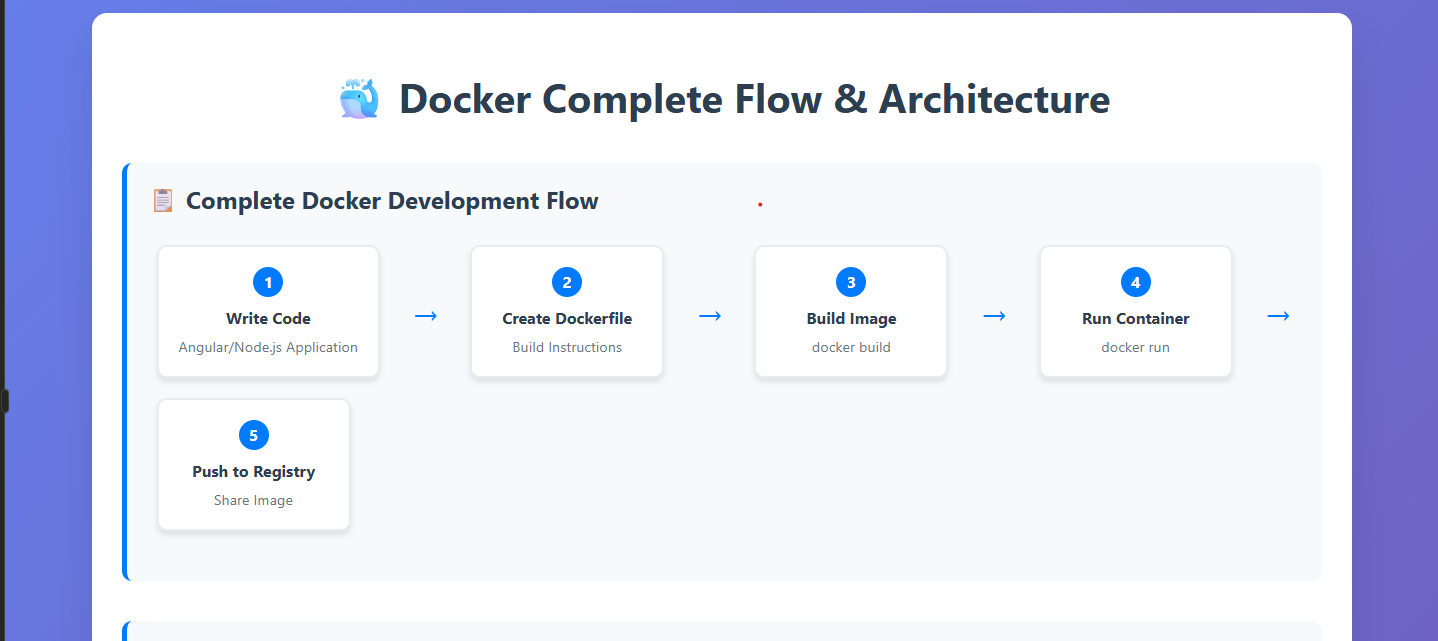
HOSTING – DOCKER AND KUBERNETES NOTES

GITHUB REPOSITORY : https://github.com/LavanyaPriyadharshini/DockerKubernetesNotes.git



A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**What is Docker Really?**

**Docker is a containerization platform** that packages applications and their dependencies into lightweight, portable containers. Think of it as creating a "shipping container" for your software

**Real-world analogy:**

* **Without Docker:** Like moving houses by packing everything loosely in different trucks - things get mixed up, broken, or lost
* **With Docker:** Like using standardized shipping containers - everything stays organized, secure, and can be moved anywhere

**Core Docker Concepts Explained**

**1. Docker Image**

**Definition:** A read-only template used to create containers

**Think of it as:** A recipe or blueprint

* Contains your application code
* Includes all dependencies (libraries, tools, etc.)
* Has operating system files
* Stored as layers (like an onion)

Example Angular App Image Contents:

my-angular-image

├── Alpine Linux (base OS)

├── Nginx web server

├── Node.js (for building)

├── Your Angular app files

└── Configuration files

2. Docker Container

**Definition:** A running instance of an image

**Think of it as:** The actual cake baked from the recipe

* Has its own isolated filesystem
* Gets its own network interface
* Runs as a separate process
* Can be started, stopped, moved, deleted

3. Dockerfile

**Definition:** A text file with instructions to build an image

**Analogy:** Step-by-step cooking instructions

IMPORTANT - Let's Build a Complete Angular Docker Example

STEP 1 : # Create new Angular project (if you don't have one)

ng new my-docker-app

cd my-docker-app

# Build for production

ng build –prod

Your project structure:

my-docker-app/ ---- ANGULAR PROJECT NAME

├── src/ # Source code

├── dist/ # Built files (after ng build)

├── package.json # Dependencies

├── angular.json # Angular config

└── node\_modules/ # Installed packages

Step 2: Create Dockerfile (Multi-stage Build)

Create Dockerfile in project root:

# ================================

# Stage 1: Build the Angular app

# ================================

FROM node:16-alpine AS builder

# Why node:16-alpine?

# - node:16 = Node.js version 16

# - alpine = Lightweight Linux (5MB vs 100MB+)

# - AS builder = Name this stage "builder"

# Set working directory inside container

WORKDIR /app

# Copy package files first (for better caching)

COPY package\*.json ./

# Install dependencies

# npm ci is faster and more reliable than npm install

RUN npm ci --only=production && npm cache clean --force

# Copy source code

COPY . .

# Build Angular app for production

RUN npm run build --prod

# ================================

# Stage 2: Serve with Nginx

# ================================

FROM nginx:alpine

# Why nginx:alpine?

# - nginx = Fast, lightweight web server

# - alpine = Small base image

# - Perfect for serving static files

# Remove default nginx website

RUN rm -rf /usr/share/nginx/html/\*

# Copy built app from builder stage

COPY --from=builder /app/dist/my-docker-app /usr/share/nginx/html

# Copy custom nginx configuration

COPY nginx.conf /etc/nginx/nginx.conf

# Expose port 80 (nginx default)

EXPOSE 80

# Start nginx (stays running in foreground)

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

**Step 3: Create nginx.conf**

events {

worker\_connections 1024;

}

http {

include /etc/nginx/mime.types;

default\_type application/octet-stream;

sendfile on;

keepalive\_timeout 65;

server {

listen 80;

server\_name localhost;

root /usr/share/nginx/html;

index index.html;

# Handle Angular routing - very important!

location / {

try\_files $uri $uri/ /index.html;

}

# Cache static assets for better performance

location ~\* \.(js|css|png|jpg|jpeg|gif|ico|svg)$ {

expires 1y;

add\_header Cache-Control "public, immutable";

}

# Security headers

add\_header X-Frame-Options "SAMEORIGIN" always;

add\_header X-XSS-Protection "1; mode=block" always;

add\_header X-Content-Type-Options "nosniff" always;

}

}

Why this nginx config is important:

 **try\_files $uri $uri/ /index.html**: Angular is a SPA - all routes should serve index.html

 **Caching**: Static assets cached for 1 year for better performance

 **Security headers**: Basic security improvements

**Step 4: Create .dockerignore**

# Exclude from Docker build context

node\_modules

npm-debug.log\*

dist

.git

.gitignore

README.md

.env

.DS\_Store

coverage

.nyc\_output

.angular

**Why .dockerignore?**

* Reduces build context size
* Faster builds
* Avoids copying unnecessary files

Docker Commands Deep Dive

STEP 1 : Building the Image

# Build image with tag

docker build -t my-angular-app:v1.0 .

# Build with different tag

docker build -t my-angular-app:latest .

# Build with no cache (fresh build)

docker build --no-cache -t my-angular-app:latest .

**What happens during build:**

1. Docker reads Dockerfile
2. Creates temporary containers for each instruction
3. Executes instruction in container
4. Commits changes as new layer
5. Removes temporary container
6. Repeats for next instruction

Build output example:

Step 1/8 : FROM node:16-alpine AS builder

---> abc123def456

Step 2/8 : WORKDIR /app

---> Using cache

---> def456ghi789

Step 3/8 : COPY package\*.json ./

---> 123abc456def

...

Successfully built 789def123abc

Successfully tagged my-angular-app:v1.0

Running Containers

# Basic run

docker run my-angular-app:v1.0

# Run with port mapping

docker run -p 8080:80 my-angular-app:v1.0

# Run in background (detached)

docker run -d -p 8080:80 my-angular-app:v1.0

# Run with custom name

docker run -d -p 8080:80 --name my-app my-angular-app:v1.0

# Run with environment variables

docker run -d -p 8080:80 -e ENV=production my-angular-app:v1.0

# Run with volume (for development)

docker run -d -p 8080:80 -v $(pwd)/src:/app/src my-angular-app:v1.0

Port mapping explained:

-p 8080:80 means:

* 8080 = Port on your computer
* 80 = Port inside container
* Traffic to localhost:8080 → container:80

Container Management

# List running containers

docker ps

# List all containers (including stopped)

docker ps -a

# Stop container

docker stop my-app

# Start stopped container

docker start my-app

# Restart container

docker restart my-app

# Remove container

docker rm my-app

# Remove container forcefully

docker rm -f my-app

# View container logs

docker logs my-app

# Follow logs in real-time

docker logs -f my-app

# Execute command in running container

docker exec -it my-app /bin/sh

Image Management

# List images

docker images

# Remove image

docker rmi my-angular-app:v1.0

# Remove unused images

docker image prune

# Show image details

docker inspect my-angular-app:v1.0

# Show image history (layers)

docker history my-angular-app:v1.0

Docker Networking Explained

Default Network Behavior

# Create network

docker network create my-network

# Run container on specific network

docker run -d --network my-network --name web-app my-angular-app:v1.0

# Run database on same network

docker run -d --network my-network --name database postgres:13

# Now web-app can connect to database using hostname "database"

**Network Types:**

* **bridge** (default): Isolated network for containers
* **host**: Use host machine's network directly
* **none**: No network access

Docker Volumes (Data Persistence)

# Create named volume

docker volume create app-data

# Run with volume

docker run -d -p 8080:80 -v app-data:/data my-angular-app:v1.0

# Mount host directory

docker run -d -p 8080:80 -v /host/path:/container/path my-angular-app:v1.0

**Volume types:**

 **Named volumes**: Managed by Docker

 **Bind mounts**: Direct host directory mapping

 **tmpfs**: In-memory storage

Docker Registry & Image Sharing

Push to Docker Hub

# Login to Docker Hub

docker login

# Tag image with your username

docker tag my-angular-app:v1.0 yourusername/my-angular-app:v1.0

# Push to Docker Hub

docker push yourusername/my-angular-app:v1.0

# Pull from Docker Hub (on any machine)

docker pull yourusername/my-angular-app:v1.0

Private Registry (GitLab)

# Login to GitLab registry

docker login registry.gitlab.com

# Tag for GitLab

docker tag my-angular-app:v1.0 registry.gitlab.com/yourusername/project:v1.0

# Push to GitLab

docker push registry.gitlab.com/yourusername/project:v1.0

Advanced Docker Features

**Multi-container Applications with Docker Compose** Create docker-compose.yml:

version: '3.8'

services:

# Angular frontend

frontend:

build: .

ports:

- "8080:80"

depends\_on:

- api

environment:

- API\_URL=http://api:3000

# Node.js backend

api:

image: node:16-alpine

working\_dir: /app

volumes:

- ./backend:/app

ports:

- "3000:3000"

command: npm start

depends\_on:

- database

# PostgreSQL database

database:

image: postgres:13

environment:

POSTGRES\_DB: myapp

POSTGRES\_USER: user

POSTGRES\_PASSWORD: password

volumes:

- db-data:/var/lib/postgresql/data

volumes:

db-data:

Run with compose:

# Start all services

docker-compose up -d

# View logs

docker-compose logs -f

# Stop all services

docker-compose down

# Rebuild and start

docker-compose up –build

Troubleshooting Common Issues

1. Build Fails

# Check build logs

docker build --no-cache -t my-app .

# Build with verbose output

docker build --progress=plain -t my-app .

2. Container Won't Start

# Check container logs

docker logs container-name

# Run interactively for debugging

docker run -it my-app /bin/sh

3. Port Already in Use

# Find process using port

lsof -i :8080

# Use different port

docker run -p 8081:80 my-app

4. Permission Issues

# Run as root (not recommended for production)

docker run --user root my-app

# Fix file permissions

docker run -v $(pwd):/app --user $(id -u):$(id -g) my-app