Docker Volumes, Networks & Compose

Table of Contents

Docker Volumes

What Are Docker Volumes?

Docker Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. They are completely managed by Docker.

Why Use Volumes?

```
Problem: Container data is lost when container is removed
Solution: Use volumes to persist data outside containers

Benefits:
 Data persists after container deletion
 Easy to backup and migrate
 Shared between multiple containers
 Better performance than bind mounts
 Work on both Linux and Windows
```

Volume vs Container Storage

```
                    WITHOUT VOLUME


Container
    /var/lib/mysql
         database files

docker rm container  →   Data GONE!


                    WITH VOLUME
```

```
Container                    Volume (managed by Docker)
    /var/lib/mysql  ↔          database files

docker rm container  →   Data PERSISTS!
```

Creating and Using Volumes

Create Volume

```bash
# Create a named volume
docker volume create mydata

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

# Create with labels
docker volume create --label env=production mydata
```

List Volumes

```bash
# List all volumes
docker volume ls

# Output:
# DRIVER    VOLUME NAME
# local     mydata
# local     postgres_data
# local     redis_cache

# Filter volumes
docker volume ls --filter "label=env=production"
docker volume ls --filter "dangling=true"  # Unused volumes
```

Inspect Volume

```bash
# Get detailed information
docker volume inspect mydata

# Output:
# [
#     {
#         "CreatedAt": "2023-10-28T10:00:00Z",
```

```
#         "Driver": "local",
#         "Labels": {},
#         "Mountpoint": "/var/lib/docker/volumes/mydata/_data",
#         "Name": "mydata",
#         "Options": {},
#         "Scope": "local"
#     }
# ]

# Get specific field
docker volume inspect --format '{{.Mountpoint}}' mydata
```

Using Volumes with Containers

Named Volumes

```
# Run container with named volume
docker run -d \
  --name myapp \
  -v mydata:/app/data \
  nginx

# Multiple volumes
docker run -d \
  -v data:/app/data \
  -v logs:/app/logs \
  -v config:/app/config \
  myapp
```

Anonymous Volumes

```
# Docker creates a unique name
docker run -d -v /app/data nginx

# List to see generated name
docker volume ls
# DRIVER    VOLUME NAME
# local     a1b2c3d4e5f6...
```

Real-World Examples

Example 1: PostgreSQL Database

```
# Create volume for database
docker volume create postgres_data

# Run PostgreSQL with persistent storage
docker run -d \
```

```
  --name postgres \
  -e POSTGRES_PASSWORD=mysecretpassword \
  -v postgres_data:/var/lib/postgresql/data \
  -p 5432:5432 \
  postgres:14

# Insert data
docker exec -it postgres psql -U postgres -c "CREATE DATABASE myapp;"

# Remove container
docker rm -f postgres

# Run new container with same volume
docker run -d \
  --name postgres_new \
  -e POSTGRES_PASSWORD=mysecretpassword \
  -v postgres_data:/var/lib/postgresql/data \
  -p 5432:5432 \
  postgres:14

# Data still exists!
docker exec -it postgres_new psql -U postgres -c "\l"
```

Example 2: MongoDB with Backup

```
# Create volume
docker volume create mongo_data

# Run MongoDB
docker run -d \
  --name mongodb \
  -v mongo_data:/data/db \
  -p 27017:27017 \
  mongo:6

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

# Restore volume
docker run --rm \
  -v mongo_data:/data \
  -v $(pwd):/backup \
```

```
  ubuntu \
  tar xzf /backup/mongo-backup.tar.gz -C /
```

Example 3: Shared Volume Between Containers

```
# Create shared volume
docker volume create shared_data

# Container 1: Writer
docker run -d \
  --name writer \
  -v shared_data:/data \
  alpine \
  sh -c "while true; do echo $(date) >> /data/log.txt; sleep 5; done"

# Container 2: Reader
docker run -d \
  --name reader \
  -v shared_data:/data:ro \
  alpine \
  sh -c "while true; do tail -f /data/log.txt; sleep 1; done"

# Both containers share the same data!
docker logs reader
```

Volume Management

Remove Volumes

```
# Remove specific volume
docker volume rm mydata

# Remove all unused volumes
docker volume prune

# Remove volumes with filter
docker volume prune --filter "label=env=development"

# Force remove (even if in use)
docker volume rm -f mydata
```

Volume Drivers

```
# Local driver (default)
docker volume create --driver local myvolume

# NFS driver
```

```
docker volume create --driver local \
  --opt type=nfs \
  --opt o=addr=192.168.1.1,rw \
  --opt device=:/path/to/dir \
  nfs_volume

# Cloud drivers (plugins)
# AWS EBS, Azure File, GlusterFS, etc.
```

Mount Binds in Docker

What Are Bind Mounts?

Bind Mounts link a file or directory on the host machine to a file or directory in the container.

Bind Mounts vs Volumes

```
Volume:
- Managed by Docker
- Stored in Docker area: /var/lib/docker/volumes/
- Best for production
- Can be shared safely

Bind Mount:
- You specify exact host path
- Can be anywhere on host
- Great for development
- Direct file access
```

Using Bind Mounts

Basic Syntax

```
# Bind mount syntax
docker run -v /host/path:/container/path image

# Or using --mount (more explicit)
docker run --mount type=bind,source=/host/path,target=/container/path image
```

Development Example

```
# Project structure
my-app/
   src/
       app.py
```

```
    utils.py
  Dockerfile
  requirements.txt

# Bind mount for live code editing
docker run -d \
  --name dev-app \
  -v $(pwd)/src:/app/src \
  -p 5000:5000 \
  myapp:dev

# Edit src/app.py on host
# Changes immediately reflect in container!
# No need to rebuild image
```

Real-World Bind Mount Examples

Example 1: Node.js Development

```
# Project directory
cd /path/to/my-node-app

# Run with bind mount
docker run -d \
  --name node-dev \
  -v $(pwd):/usr/src/app \
  -v /usr/src/app/node_modules \
  -p 3000:3000 \
  -w /usr/src/app \
  node:18 \
  npm run dev

# Edit files on host → Hot reload in container!
```

docker-compose.yml for development:

```
version: '3.8'
services:
  web:
    build: .
    volumes:
      - ./src:/app/src        # Bind mount source code
      - /app/node_modules     # Anonymous volume for node_modules
    ports:
      - "3000:3000"
    command: npm run dev
```

Example 2: Database Configuration

```
# Custom MySQL configuration
docker run -d \
  --name mysql \
  -v $(pwd)/mysql.cnf:/etc/mysql/conf.d/mysql.cnf:ro \
  -v mysql_data:/var/lib/mysql \
  -e MYSQL_ROOT_PASSWORD=secret \
  -p 3306:3306 \
  mysql:8
```

Example 3: Nginx with Custom Config

```
# Custom nginx.conf on host
cat > nginx.conf << 'EOF'
server {
    listen 80;
    server_name localhost;

    location / {
        root /usr/share/nginx/html;
        index index.html;
    }

    location /api {
        proxy_pass http://backend:5000;
    }
}
EOF

# Run Nginx with custom config
docker run -d \
  --name nginx \
  -v $(pwd)/nginx.conf:/etc/nginx/conf.d/default.conf:ro \
  -v $(pwd)/html:/usr/share/nginx/html:ro \
  -p 80:80 \
  nginx:alpine
```

Bind Mount Options

```
# Read-only mount
docker run -v $(pwd)/config:/app/config:ro nginx

# Read-write (default)
docker run -v $(pwd)/data:/app/data:rw nginx

# Consistent, cached, delegated (Mac performance)
```

```
docker run -v $(pwd):/app:cached nginx      # Host authoritative
docker run -v $(pwd):/app:delegated nginx   # Container authoritative

# Using --mount (recommended for clarity)
docker run --mount type=bind,source="$(pwd)",target=/app,readonly nginx
```

Development Workflow

```
# Complete development setup

# 1. Create project
mkdir my-python-app && cd my-python-app

# 2. Create files
cat > app.py << 'EOF'
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, debug=True)
EOF

cat > requirements.txt << 'EOF'
flask==2.3.0
EOF

cat > Dockerfile << 'EOF'
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "app.py"]
EOF

# 3. Build image
docker build -t myapp:dev .

# 4. Run with bind mount for development
docker run -d \
  --name myapp-dev \
  -v $(pwd):/app \
```

```
  -p 5000:5000 \
  myapp:dev

# 5. Edit app.py on your machine
# 6. Flask auto-reloads!
# 7. See changes immediately in browser
```

Docker Networks

What is Docker Network?

Docker networking allows containers to communicate with each other and the outside world.

Default Networks

```
# List networks
docker network ls

# Output:
# NETWORK ID     NAME      DRIVER    SCOPE
# a1b2c3d4e5f6   bridge    bridge    local
# b2c3d4e5f6g7   host      host      local
# c3d4e5f6g7h8   none      null      local
```

Network Drivers

1. Bridge (Default)

```
# Default network for containers
# Containers on same bridge can communicate

# Run container on default bridge
docker run -d --name app1 nginx

# Check container IP
docker inspect app1 | grep IPAddress
# Output: "IPAddress": "172.17.0.2"
```

2. Host

```
# Container uses host network stack
# No network isolation

docker run -d --network host nginx
```

```
# Container accessible directly on host IP
# No port mapping needed!
```

3. None

```
# No networking
docker run -d --network none nginx

# Container is completely isolated
```

4. Custom Bridge

```
# Create custom bridge network
docker network create mynetwork

# Run containers on custom network
docker run -d --name app1 --network mynetwork nginx
docker run -d --name app2 --network mynetwork redis

# app1 can reach app2 by hostname!
docker exec app1 ping app2  # Works!
```

Creating Networks

```
# Basic network
docker network create mynetwork

# With specific subnet
docker network create --subnet=172.18.0.0/16 mynetwork

# With gateway
docker network create \
  --subnet=172.18.0.0/16 \
  --gateway=172.18.0.1 \
  mynetwork

# With driver options
docker network create \
  --driver=bridge \
  --subnet=172.28.0.0/16 \
  --ip-range=172.28.5.0/24 \
  --gateway=172.28.5.254 \
  mynetwork

# Inspect network
docker network inspect mynetwork
```

Connecting Containers

```
# Connect running container
docker network connect mynetwork mycontainer

# Connect with specific IP
docker network connect --ip 172.18.0.10 mynetwork mycontainer

# Connect with alias
docker network connect --alias db mynetwork postgres

# Disconnect
docker network disconnect mynetwork mycontainer
```

Real-World Network Examples

Example 1: Multi-Tier Application

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

# Database tier
docker run -d \
  --name postgres \
  --network app-network \
  -e POSTGRES_PASSWORD=secret \
  postgres:14

# Backend tier
docker run -d \
  --name backend \
  --network app-network \
  -e DATABASE_URL=postgresql://postgres:secret@postgres:5432/mydb \
  mybackend:latest

# Frontend tier
docker run -d \
  --name frontend \
  --network app-network \
  -p 80:80 \
  -e API_URL=http://backend:5000 \
  myfrontend:latest

# Communication:
# frontend → backend (via hostname "backend")
# backend → postgres (via hostname "postgres")
```

Example 2: Microservices

```
# Create network
docker network create microservices

# Service discovery by container name
docker run -d --name users-service --network microservices users:latest
docker run -d --name orders-service --network microservices orders:latest
docker run -d --name payments-service --network microservices payments:latest
docker run -d --name gateway --network microservices -p 8080:8080 gateway:latest

# gateway can reach:
# - http://users-service:3000
# - http://orders-service:3001
# - http://payments-service:3002
```

Example 3: Isolated Networks

```
# Frontend network (public facing)
docker network create frontend-net

# Backend network (private)
docker network create backend-net

# Web server (both networks)
docker run -d \
  --name nginx \
  --network frontend-net \
  -p 80:80 \
  nginx

docker network connect backend-net nginx

# API server (backend only)
docker run -d \
  --name api \
  --network backend-net \
  api:latest

# Database (backend only)
docker run -d \
  --name postgres \
  --network backend-net \
  postgres:14

# Result:
```

```
# - nginx can reach api and postgres
# - api can reach postgres
# - Internet can only reach nginx
```

Network Troubleshooting

```
# Inspect network
docker network inspect mynetwork

# Check container's networks
docker inspect mycontainer | grep Networks -A 10

# Test connectivity
docker exec container1 ping container2
docker exec container1 curl http://container2:80

# View network traffic
docker run --rm --net=host nicolaka/netshoot
```

What is Docker Compose

Definition

Docker Compose is a tool for defining and running multi-container Docker applications using a YAML file.

Why Docker Compose?

```
Problem: Managing multiple containers manually is tedious
docker run container1...
docker run container2...
docker run container3...
docker network create...
docker network connect...

Solution: Define everything in docker-compose.yml
docker compose up  # Done!
```

Benefits

```
Define multi-container apps in one file
Start everything with one command
Automatic network creation
Environment variable management
Volume management
Service dependencies
```

```
Easy scaling
Development and production configs
```

Installation

```
# Docker Compose v2 (included with Docker Desktop)
docker compose version

# Output: Docker Compose version v2.21.0

# If not installed:
# Linux
sudo apt-get install docker-compose-plugin

# Or download binary
sudo curl -L "https://github.com/docker/compose/releases/latest/download/docker-compose-$(ur
sudo chmod +x /usr/local/bin/docker-compose
```

Basic docker-compose.yml

```
version: '3.8'

services:
  web:
    image: nginx:alpine
    ports:
      - "80:80"
    volumes:
      - ./html:/usr/share/nginx/html

  db:
    image: postgres:14
    environment:
      POSTGRES_PASSWORD: secret
    volumes:
      - db-data:/var/lib/postgresql/data

volumes:
  db-data:
```

Compose Commands

```
# Start all services
docker compose up

# Start in detached mode
docker compose up -d
```

```bash
# Build and start
docker compose up --build

# Stop 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
docker compose logs -f web  # Follow logs for specific service

# List running services
docker compose ps

# Execute command in service
docker compose exec web sh

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

# Restart services
docker compose restart

# Pull latest images
docker compose pull
```

Docker Compose Examples

Example 1: WordPress with MySQL

```yaml
version: '3.8'

services:
  wordpress:
    image: wordpress:latest
    ports:
      - "8080:80"
    environment:
      WORDPRESS_DB_HOST: db
```

```
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: secret
      WORDPRESS_DB_NAME: wordpress
    volumes:
      - wordpress_data:/var/www/html
    depends_on:
      - db
    restart: always

  db:
    image: mysql:8
    environment:
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: secret
      MYSQL_ROOT_PASSWORD: rootsecret
    volumes:
      - db_data:/var/lib/mysql
    restart: always

volumes:
  wordpress_data:
  db_data:
```

```
# Start WordPress
docker compose up -d

# Open browser: http://localhost:8080
```

Example 2: MERN Stack Application

```
version: '3.8'

services:
  # MongoDB
  mongodb:
    image: mongo:6
    ports:
      - "27017:27017"
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: secret
    volumes:
      - mongo-data:/data/db
    networks:
      - mern-network
```

```yaml
  # Express Backend
  backend:
    build: ./backend
    ports:
      - "5000:5000"
    environment:
      MONGODB_URI: mongodb://admin:secret@mongodb:27017
      NODE_ENV: development
    volumes:
      - ./backend:/app
      - /app/node_modules
    depends_on:
      - mongodb
    networks:
      - mern-network
    command: npm run dev

  # React Frontend
  frontend:
    build: ./frontend
    ports:
      - "3000:3000"
    environment:
      REACT_APP_API_URL: http://localhost:5000
    volumes:
      - ./frontend:/app
      - /app/node_modules
    depends_on:
      - backend
    networks:
      - mern-network
    command: npm start

volumes:
  mongo-data:

networks:
  mern-network:
    driver: bridge
```

Example 3: Microservices with Multiple Containers

```yaml
version: '3.8'

services:
```

```yaml
# API Gateway
gateway:
  build: ./gateway
  ports:
    - "8080:8080"
  environment:
    USERS_SERVICE: http://users:3001
    ORDERS_SERVICE: http://orders:3002
    PRODUCTS_SERVICE: http://products:3003
  networks:
    - microservices
  depends_on:
    - users
    - orders
    - products

# Users Service
users:
  build: ./services/users
  environment:
    DB_HOST: postgres
    DB_NAME: users_db
    DB_USER: postgres
    DB_PASSWORD: secret
  networks:
    - microservices
  depends_on:
    - postgres

# Orders Service
orders:
  build: ./services/orders
  environment:
    DB_HOST: postgres
    DB_NAME: orders_db
    REDIS_URL: redis://redis:6379
  networks:
    - microservices
  depends_on:
    - postgres
    - redis

# Products Service
products:
  build: ./services/products
  environment:
```

```
      DB_HOST: mongodb
      DB_NAME: products
    networks:
      - microservices
    depends_on:
      - mongodb

  # PostgreSQL
  postgres:
    image: postgres:14
    environment:
      POSTGRES_PASSWORD: secret
    volumes:
      - postgres-data:/var/lib/postgresql/data
    networks:
      - microservices

  # MongoDB
  mongodb:
    image: mongo:6
    volumes:
      - mongo-data:/data/db
    networks:
      - microservices

  # Redis
  redis:
    image: redis:7-alpine
    networks:
      - microservices

volumes:
  postgres-data:
  mongo-data:

networks:
  microservices:
    driver: bridge
```

Docker Compose with Network

```
version: '3.8'

services:
  web:
    image: nginx:alpine
```

```
    ports:
      - "80:80"
    networks:
      - frontend
      - backend

  api:
    image: myapi:latest
    networks:
      - backend
    depends_on:
      - db

  db:
    image: postgres:14
    environment:
      POSTGRES_PASSWORD: secret
    networks:
      - backend
    volumes:
      - db-data:/var/lib/postgresql/data

networks:
  frontend:
    driver: bridge
  backend:
    driver: bridge
    internal: true  # No external access

volumes:
  db-data:
```

Docker Compose with Volume

```
version: '3.8'

services:
  app:
    image: myapp:latest
    volumes:
      # Named volume
      - app-data:/app/data

      # Bind mount
      - ./config:/app/config:ro
```

```
      # Anonymous volume
      - /app/temp

      # Host machine volume
      - /host/path:/container/path

volumes:
  app-data:
    driver: local
    driver_opts:
      type: none
      device: /path/on/host
      o: bind
```

Docker Compose with Port Binding

```
version: '3.8'

services:
  web:
    image: nginx:alpine
    ports:
      # Host:Container
      - "80:80"
      - "443:443"

      # Bind to specific interface
      - "127.0.0.1:8080:80"

      # Random host port
      - "80"

      # UDP port
      - "53:53/udp"

      # Port range
      - "3000-3005:3000-3005"
```

Pre-defined Images

Official Images from Docker Hub

```
# Operating Systems
docker pull ubuntu:22.04
docker pull debian:bullseye
```

```
docker pull alpine:latest  # Minimal (~5MB)
docker pull centos:8

# Programming Languages
docker pull python:3.11
docker pull node:18
docker pull golang:1.21
docker pull openjdk:17

# Databases
docker pull postgres:14
docker pull mysql:8
docker pull mongodb:6
docker pull redis:7

# Web Servers
docker pull nginx:alpine
docker pull httpd:latest  # Apache
docker pull caddy:latest

# Message Queues
docker pull rabbitmq:3-management
docker pull redis:7

# Monitoring & Logging
docker pull grafana/grafana:latest
docker pull prometheus:latest
docker pull elasticsearch:8.9.0
```

Running Pre-defined Images

```
# Nginx Web Server
docker run -d -p 80:80 nginx:alpine

# PostgreSQL Database
docker run -d \
  --name postgres \
  -e POSTGRES_PASSWORD=mysecretpassword \
  -p 5432:5432 \
  postgres:14

# Redis Cache
docker run -d -p 6379:6379 redis:7-alpine

# MongoDB
docker run -d \
```

```
  --name mongodb \
  -p 27017:27017 \
  -e MONGO_INITDB_ROOT_USERNAME=admin \
  -e MONGO_INITDB_ROOT_PASSWORD=secret \
  mongo:6

# MySQL
docker run -d \
  --name mysql \
  -e MYSQL_ROOT_PASSWORD=secret \
  -e MYSQL_DATABASE=mydb \
  -p 3306:3306 \
  mysql:8
```

Running Containers in Interactive Mode

```
# Ubuntu interactive shell
docker run -it ubuntu:22.04 bash

# Python interactive shell
docker run -it python:3.11 python

# Node.js REPL
docker run -it node:18 node

# Alpine shell
docker run -it alpine:latest sh

# MySQL client
docker run -it --rm mysql:8 mysql -h host.docker.internal -u root -p

# PostgreSQL client
docker run -it --rm postgres:14 psql -h host.docker.internal -U postgres
```

Push and Pull Images

Docker Registry

```
Docker Hub (default): hub.docker.com
Private Registries:
- AWS ECR
- Google Container Registry
- Azure Container Registry
- Self-hosted Registry
```

Pull Images Remotely

```
# Pull from Docker Hub (default)
docker pull nginx:latest

# Pull specific version
docker pull nginx:1.25.3

# Pull from specific registry
docker pull docker.io/library/nginx:latest

# Pull from private registry
docker pull myregistry.com:5000/myapp:latest

# Pull all tags
docker pull --all-tags nginx

# Pull for specific platform
docker pull --platform linux/amd64 nginx
docker pull --platform linux/arm64 nginx

# Pull with quiet output
docker pull -q nginx
```

Push Images to DockerHub

Step 1: Create Docker Hub Account

```
1. Go to https://hub.docker.com
2. Sign up for free account
3. Create repository (public or private)
```

Step 2: Login

```
# Login to Docker Hub
docker login

# Enter username and password
# Login Succeeded

# Login to specific registry
docker login myregistry.com

# Login with credentials
docker login -u username -p password
```

Step 3: Tag Image

```
# Tag image with your username
docker tag myapp:latest username/myapp:latest

# Tag with version
docker tag myapp:latest username/myapp:1.0.0

# Tag with multiple tags
docker tag myapp:latest username/myapp:latest
docker tag myapp:latest username/myapp:1.0
docker tag myapp:latest username/myapp:stable
```

Step 4: Push Image

```
# Push to Docker Hub
docker push username/myapp:latest

# Output:
# The push refers to repository [docker.io/username/myapp]
# a1b2c3d4e5f6: Pushed
# b2c3d4e5f6g7: Pushed
# c3d4e5f6g7h8: Pushed
# latest: digest: sha256:abc123... size: 1234

# Push all tags
docker push --all-tags username/myapp
```

Complete Example: Build, Push, Pull

```
# Step 1: Build image
docker build -t myapp:1.0 .

# Step 2: Tag for Docker Hub
docker tag myapp:1.0 username/myapp:1.0
docker tag myapp:1.0 username/myapp:latest

# Step 3: Login
docker login

# Step 4: Push
docker push username/myapp:1.0
docker push username/myapp:latest

# Step 5: Pull on another machine
docker pull username/myapp:latest

# Step 6: Run
```

```
docker run -d -p 8080:80 username/myapp:latest
```

Private Registry

Setup Private Registry

```
# Run registry container
docker run -d \
  -p 5000:5000 \
  --name registry \
  -v registry-data:/var/lib/registry \
  registry:2

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

# Pull from private registry
docker pull localhost:5000/myapp:latest
```

Secure Private Registry

```
# docker-compose.yml for secure registry
version: '3.8'

services:
  registry:
    image: registry:2
    ports:
      - "5000:5000"
    environment:
      REGISTRY_AUTH: htpasswd
      REGISTRY_AUTH_HTPASSWD_PATH: /auth/htpasswd
      REGISTRY_AUTH_HTPASSWD_REALM: Registry Realm
      REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY: /data
    volumes:
      - ./auth:/auth
      - registry-data:/data

volumes:
  registry-data:
```

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

# Login to private registry
```

```
docker login localhost:5000

# Push image
docker push localhost:5000/myapp:latest
```

Key Takeaways

1. **Volumes**: Persist data outside containers, managed by Docker
2. **Bind Mounts**: Direct link to host filesystem, great for development
3. **Networks**: Enable container communication with DNS resolution
4. **Docker Compose**: Define multi-container apps in YAML
5. **Pre-defined Images**: Use official images from Docker Hub
6. **Push/Pull**: Share images via Docker Hub or private registries

---

What's Next?

- Working with APIs in containers
- Container orchestration with Kubernetes
- Production deployment strategies
- CI/CD with Docker
- Security best practices

Master Docker ecosystem, master modern DevOps!