Docker

It works anywhere

Pascal Sun

UWA NLP-TLP Group

The University of Western Australia

November 28, 2024

Table of Contents

1 History of Docker

2 Architecture

3 Image

4 Container

5 Volume

6 Network

7 Docker Compose

8 Frequent used commands

9 Homework

10 Q&A

History of Docker

It works on my machine

Figure: It works on my machine

It is a serious problem

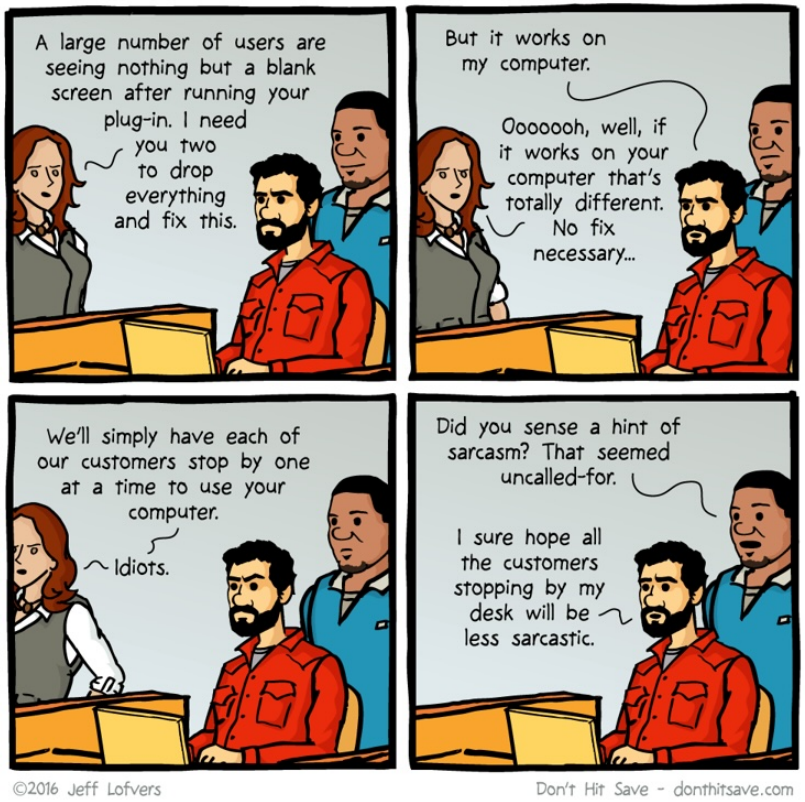

Figure: It still exists today with docker

Ship your development environment to customer

Figure: Where the idea and name from?[1]

[1]This lay the foundation principle of DevOps, CI/CD

Here it comes: Docker

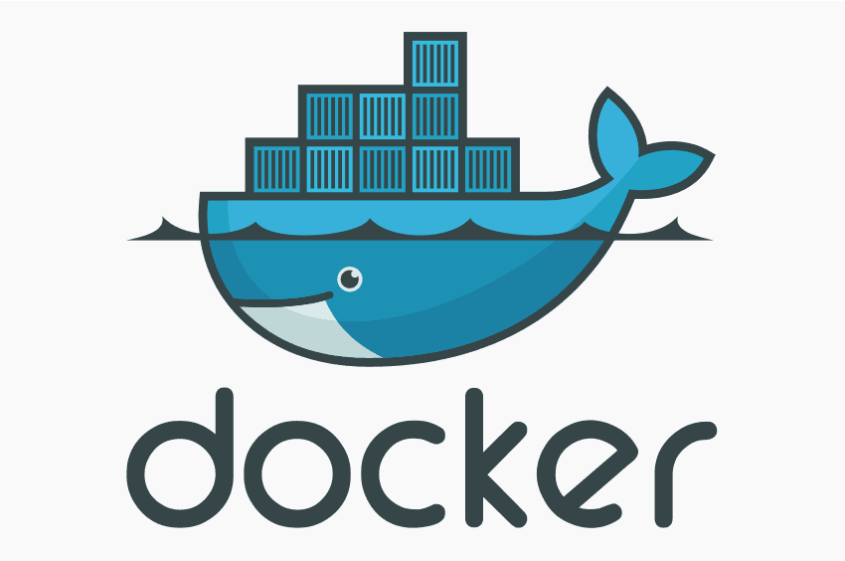

Figure: Docker Logo

History of Docker

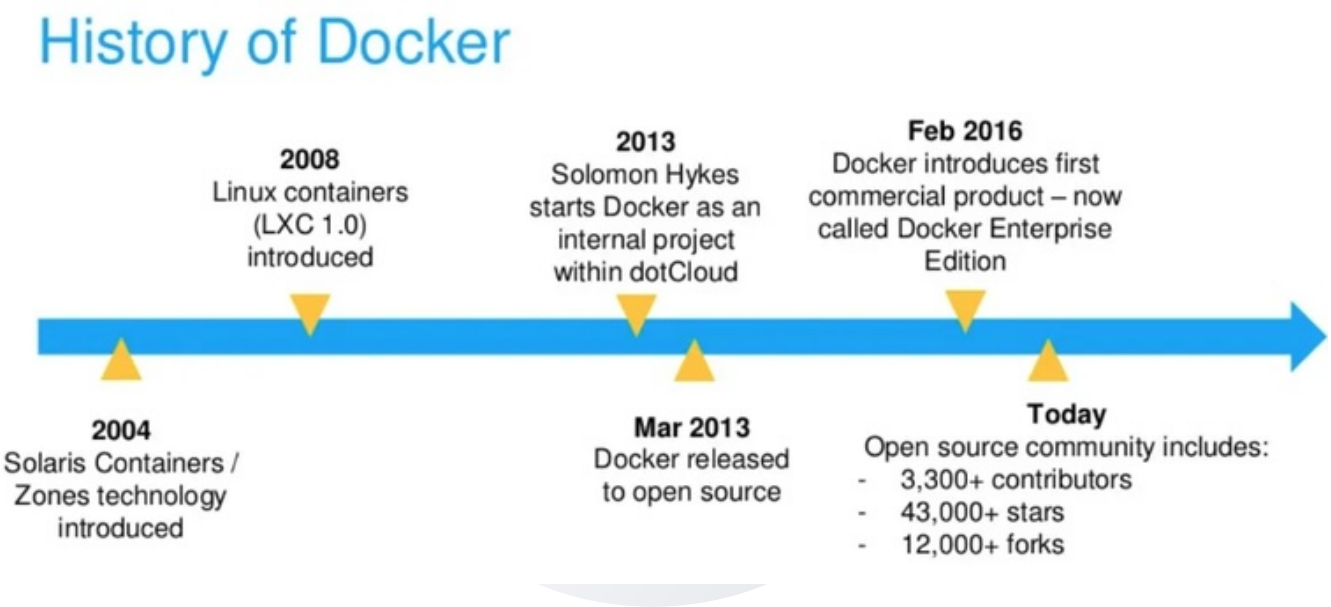

Figure: History of Docker

Architecture

Architecture of Docker

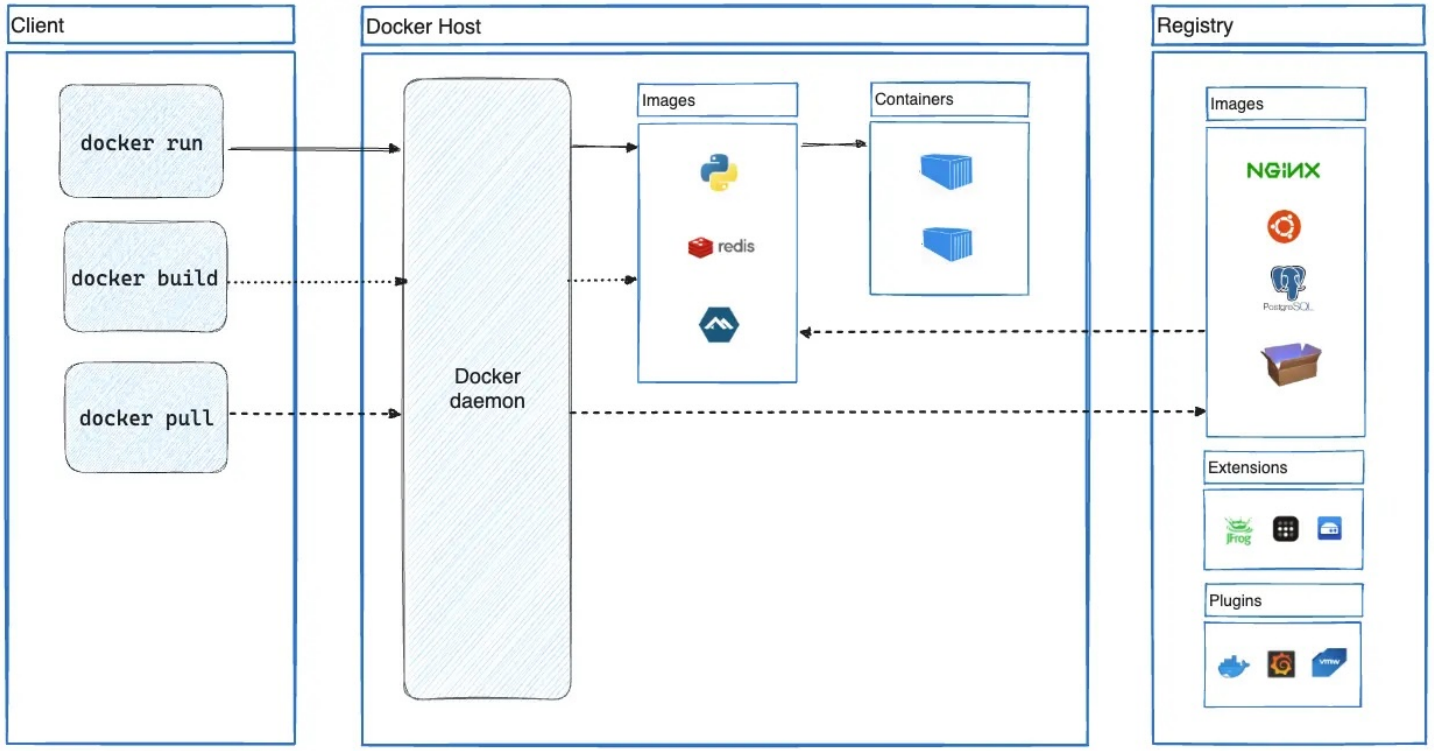

Figure: Architecture from Docker Official Documentation

Core concepts

- Image
- Container
- Volume
- Network

Image

What is a Docker Image? (1/2)

Ever Installed a New Operating System?

A Docker Image is similar to a complete OS installation package - but for your application!

Operating System:

- Windows/Mac installer
- Pre-installed programs
- System settings
- Works on any computer hardware

Docker Image:

- Base image
- Your application
- Configuration files
- Runs anywhere (operating system)

Key Point

Like a complete OS package, a Docker Image contains everything needed to run!

What is a Docker Image? (2/2)

Why Use Docker Images?

They solve the "But it works on my machine!" problem

- **Consistency:**
  - Same environment everywhere
  - No missing dependencies
  - Predictable behavior
- **Portability:**
  - Share with one file
  - Easy to distribute
  - Quick to deploy

Simple Example

Instead of saying "Install Python, then Redis, then..."
You just say "Here's the Docker Image - run it!"

Building a Docker Image

Example: Web Server Dockerfile

Let's create a simple Nginx web server image:

```
1  FROM ubuntu:22.04
2  ENV DEBIAN_FRONTEND=noninteractive
3
4  RUN apt-get update && apt-get install -y nginx
5
6  RUN echo "<!DOCTYPE html><title>Test</title>" > \
7     /var/www/html/index.html
8
9  EXPOSE 80
10 CMD ["nginx", "-g", "daemon off;"]
11
```

What's Happening Here?

- Base: Ubuntu 22.04
- Install: Nginx web server
- Config: Simple webpage
- Run: Start Nginx server

Dockerfile Commands

Essential Dockerfile Instructions

Most commonly used commands in Dockerfiles:

- **FROM:** Base image
- **RUN:** Execute commands
- **COPY/ADD:** Add files
- **WORKDIR:** Set working directory
- **ENV:** Environment variables
- **EXPOSE:** Port mapping
- **CMD:** Default command
- **ENTRYPOINT:** Main executable
- **ARG:** Build variables
- **LABEL:** Add metadata
- **USER:** Set user/group
- **VOLUME:** Mount points
- **HEALTHCHECK:** Container health
- **SHELL:** Default shell
- **ONBUILD:** Trigger commands
- **STOPSIGNAL:** Exit signal

Reference

For detailed documentation: `https://docs.docker.com/reference/dockerfile/`

Docker Image Registries

What is Docker Hub?

The default public registry for Docker images - like GitHub for Docker images!

Docker Hub Features:

- Official Images
- Free Public Repositories
- Automated Builds
- Team Collaboration
- Integration with GitHub

Common Commands:

- `docker pull ubuntu`
- `docker push myapp:1.0`
- `docker login`
- `docker tag myapp registry/myapp`

Alternative Registries

- **Amazon ECR:** AWS container registry
- **Google GCR:** Google Cloud registry
- **Azure ACR:** Microsoft Azure registry

Private Registry Example

```
docker pull private-registry.company.com/team/app:latest
```

Working with Registries

Image Naming Convention

`registry-domain/username/repository:tag`

```
# Tag image for your private registry
$ docker tag myapp:1.0 private-registry.com/myapp:1.0

# Push to private registry
$ docker push private-registry.com/myapp:1.0

# Pull from Google Container Registry
$ docker pull gcr.io/project-id/app:v1

```

Key Points

- Always use specific tags (not 'latest')
- Secure your registry credentials
- Consider using private registries for sensitive images
- Regular cleanup of unused images

Container

Container vs Image

- Container images become containers at runtime
- When running on Docker Engine:
  - Images are like installation packages
  - Containers are the running instances
- Analogy:
  - Image = Windows 10 .iso file
  - Container = Computer running Windows 10

Working with Containers

```
# Pull an image
docker pull pascalsun/docker101:latest

# Run a container
docker run -d -p 8080:80 pascalsun/docker101:latest

# List running containers
docker ps
```

Container Properties

```
❯ docker ps
CONTAINER ID   IMAGE                                          COMMAND                  CREATED          STATUS                    PORTS                                                  NAMES
3fbccf58a82c   hasura/graphql-engine:v2.43.0.cli-migrations-v3   "docker-entrypoint.s…"   12 seconds ago   Up 11 seconds (healthy)   0.0.0.0:8080->8080/tcp                                 hasura
6785ed160dbc   chatgeoplatform-platform                        "/usr/src/app/script…"   12 seconds ago   Up 11 seconds             0.0.0.0:8000->8000/tcp                                 API
08466fca9e5c   neo4j:5.20.0                                   "tini -g -- /startup…"   12 seconds ago   Up 11 seconds             0.0.0.0:7474->7474/tcp, 7473/tcp, 0.0.0.0:7687->7687/tcp   neo4j
3df1eed83240   postgis/postgis                                "docker-entrypoint.s…"   12 seconds ago   Up 11 seconds             0.0.0.0:5432->5432/tcp                                 DB
795aafe92949   qdrant/qdrant                                  "./entrypoint.sh"        12 seconds ago   Up 11 seconds             0.0.0.0:6333->6333/tcp, 6334/tcp                       qdrant
                                                                                                                                                                                 base ⊙ 15:45:39
```

Figure: docker ps results

Each container has:

- Unique Container ID
- Source Image reference
- Startup command
- Creation timestamp
- Status (running/failed)
- Port mappings
- Container name (random if not specified)

Interacting with Containers

- Enter container shell:

```
docker exec -it container_id bash
```

Containers from high level

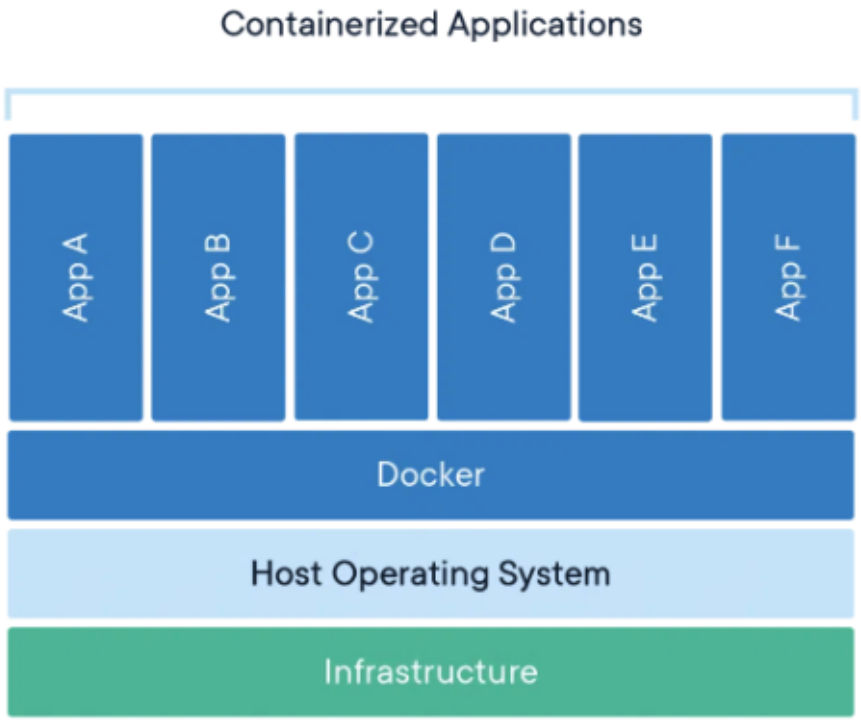

Figure: Containers in Infrastructure Level

Volume

Why Do We Need Volumes?

- Challenge with Containers:
  - Changes inside container are temporary
  - Data lost when container is removed
  - New container starts fresh from image
- Business Impact:
  - Critical for production environments
  - Essential for database containers
  - Prevents accidental data loss

Docker Volume

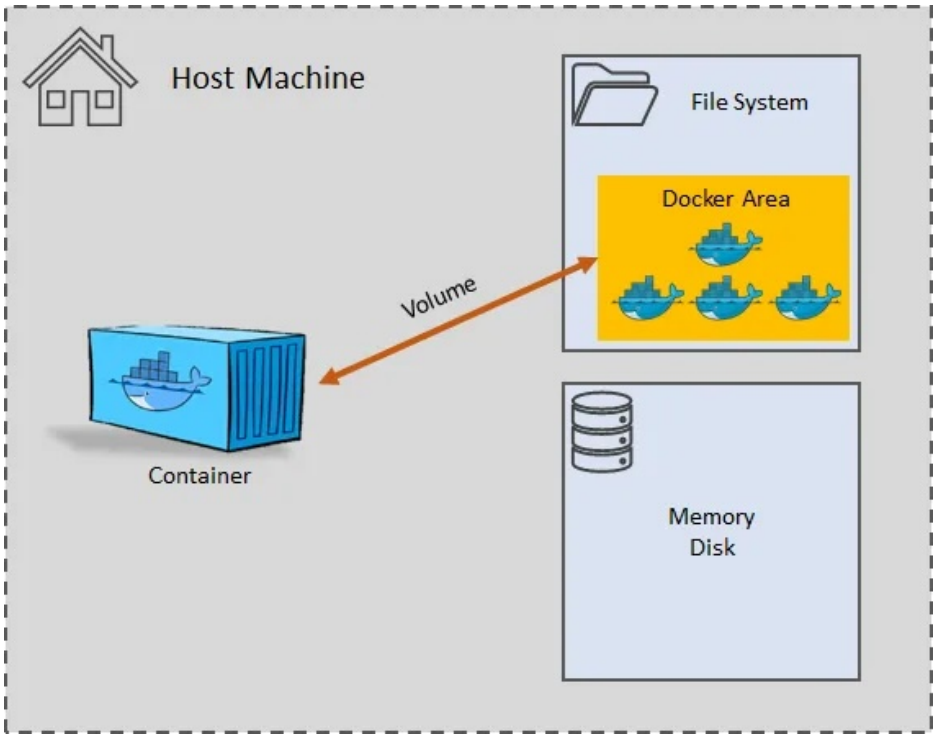

Figure: Docker Volume Illustration

Understanding Volumes

- What is a Volume?
  - Preferred mechanism for persisting container data
  - Like a "USB drive" for containers
  - Survives container lifecycle
- Two Types of Volume Mapping:
  - Docker-managed volumes
  - Host machine directory mapping

Docker-Managed Volumes

```
# Create container with volume
docker run -d -p 8080:80 \
    -v demovolume:/var/www/html \
    pascalsun/docker101:latest

# List volumes
docker volume ls

# Remove volume
docker volume rm volume_name
```

Host Directory Mapping

```
# Map local directory to container
docker run -d -p 8080:80 \
    -v ./demovolume:/var/www/html \
    pascalsun/docker101:latest
```

- Use Cases:
  - Development environments
  - Direct file editing on host
  - Code synchronization
  - Immediate feedback for changes

Volume Data Persistence

- Container Lifecycle:
  - Stop container `docker stop container_id`
  - Remove container `docker rm container_id`
  - Create new container
- Volume Behavior:
  - Data persists between container restarts
  - Volume remains after container removal
  - New containers can reuse existing volumes
  - Changes visible in both host and container

Best Practices

- Choose Volume Type:
  - Docker volumes for production data
  - Host mapping for development
- Common Applications:
  - Database storage
  - Application source code
  - Configuration files
  - Shared resources

Network

Understanding Docker Networks

- Basic Network Concept:
  - Similar to physical world communication
  - Containers need to know where to find each other
  - Requires host address and port information
- Key Components:
  - Host Address (like building name)
  - Ports (like doors in a building)
  - Networks (like different universities)

Port Mapping in Docker

- Basic Syntax: `-p host_port:container_port`
  - Left side: Host machine port
  - Right side: Container port
- Example:
  - `-p 8080:80`
  - Traffic to host port 8080
  - Redirects to container port 80
  - Common for web servers (nginx)

Common Network Scenarios

Basic Container with Port Mapping:

```
docker run -d -p 8080:80 \
    -v ./demovolume:/var/www/html \
    pascalsun/docker101:latest
```

Common Issues:

- Port Already in Use:
  - Example: Local PostgreSQL on 5432
  - Solution: Use different host port (5433:5432)

Network Isolation - University Analogy

- Understanding Network Isolation:
  - Like different universities (UWA vs Curtin)
  - Same building names allowed (CSSE building)
  - Same door numbers possible
  - But different physical locations
- Docker Implementation:
  - Different networks can have same container names
  - Same ports can be used in different networks
  - Provides isolation and organization

Docker Network Commands

```
# List all networks
docker network ls

# Default networks output:
# NETWORK ID     NAME      DRIVER    SCOPE
# 97ea4524789b   host      host      local
# b237ce03fed3   bridge    bridge    local
```

- Default Networks:
  - `host`: Host machine network
  - `bridge`: Default container network

Key Takeaways

- Port Mapping:
  - Essential for container access
  - Format: host_port:container_port
  - Handles traffic redirection
- Network Isolation:
  - Separates container environments
  - Allows name/port reuse
  - Provides security boundaries

Docker Compose

Why Docker Compose?

- Problem with Manual Docker:
  - Multiple commands needed
  - Complex setup process
  - Hard to remember configurations
  - Time-consuming to recreate environments
- Solution:
  - Centralize configurations
  - Single command to manage everything
  - Easy environment sharing
  - Reproducible development setup

docker-compose.yml example

```
services:
  pgdb:
    container_name: pgdb
    build:
      context: ./
      dockerfile: Dockerfile_PG
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
    volumes:
      - postgres_data:/var/lib/postgresql/data
    ports:
      - '5432:5432'
  sqlserverdb:
    container_name: sqlserver
    platform: linux/amd64
    build:
      context: ./
      dockerfile: Dockerfile_MS
    environment:
      ACCEPT_EULA: Y
      SA_PASSWORD: "YourStrongPassw0rd"
    ports:
      - "1433:1433"

volumes:
  postgres_data:
```

Basic Commands

```
# Start services
docker compose up

# Start and rebuild
docker compose up --build

# Run in background
docker compose up -d

# Stop services
docker compose down
```

Docker Compose Lifecycle

- After `docker compose up`:
  - Images build or pull if not exists locally
  - Containers are created
  - Networks are created
  - Volumes are created
- After `docker compose down`:
  - Containers removed
  - Networks removed
  - Volumes persist
  - Images remain

Frequent used commands

Docker Installation

- Official Docker Website:
  - Download Docker Desktop
  - Follow installation guide
  - Verify installation
- Post Installation:
  - Test docker: `docker --version`
  - Test compose: `docker compose version`
  - Test build: `docker build --help`

Container Management

```
# List containers
docker ps                      # Show running containers
docker ps -a                   # Show all containers (include stopped)
docker ps -aq                 # Show all container IDs

# Container operations
docker logs -f container_id  # Follow container logs
docker exec -it container_id bash  # Enter container shell, sometimes use sh
docker rm $(docker ps -aq)  # Remove all containers
```

Image Management

```
# Basic image operations
docker image ls                # List images
docker image rm image_id       # Remove image
docker pull image_repo_id      # Pull image

# Building and pushing
docker build -t docker101 .    # Build image
docker tag docker101:latest user/docker101:latest
docker push image_id           # Push to registry

# Cleanup
docker image prune           # Remove unused images
```

Volume and Network Commands

```
# Volume commands
docker volume ls            # List volumes
docker volume rm volume_name
docker volume prune         # Remove unused volumes

# Network commands
docker network ls           # List networks
docker network rm network_name
docker network prune        # Remove unused networks
```

Docker Compose Commands

```
# Basic compose operations
docker compose up           # Start services
docker compose up --build   # Rebuild and start
docker compose up -d        # Start in background
docker compose down         # Stop services

# File transfer
docker cp container_id:src_path dest_path  # From container
docker cp src_path container_id:dest_path  # To container
```

System Cleanup

```
# General cleanup
docker system prune          # Clean unused resources

# Complete cleanup
docker rm $(docker ps -aq)  # Remove all containers
docker volume rm $(docker volume ls -q)  # Remove volumes
docker image rm $(docker image ls -q)    # Remove images
docker network rm $(docker network ls -q) # Remove networks
```

Warning: Be careful with force removal commands!

Best Practices

The above commands cover 99.9% of use cases.

- Regular Maintenance:
  - Clean unused resources
  - Monitor disk space
  - Remove unused images
- Safety Measures:
  - Backup important volumes

Homework

Homework

- Tutorial Exercise:
  - Visit: `https://tutorial.nlp-tlp.org/docker-101`
  - Complete all commands step by step
  - Test your understanding of:
    - Container creation and management
    - Image building and tagging
    - Volume mapping
    - Network configuration
- Learning Outcomes:
  - Practical Docker command usage
  - Understanding Docker workflow
  - Troubleshooting common issues

Q&A

Q&A

Questions?

Feel free to ask anything