Docker

## Introduction

Docker is software to containerize applications. A container is a way to package the application with required dependencies and configurations.

Docker can be useful for development and deployment process.

## Problem without containers

Generally, applications were developed by multiple developers. It must run on their machine. Also, these applications need to be deployed in multiple servers based on environments like development, test, stage, production etc.

An application contains various dependencies and configurations which are required to be setup in every machine to run the application. Developers might be working on machines with various operating systems like Mac, Linux Based, Windows. The dependencies and required software must be installed and set up on their machines.

This process is prone to errors as there might be version conflicts with dependencies as well as human errors while installing them.

## Solution with containers

To solve the above problem, docker can be used to package the application along with the required dependencies (with specific versions) and other configurations.

Hence, the developers can use those containers without needing to setup the dependencies on their machines. Also, the environment servers can use the same containers for running the application. Only the docker engine is required to be installed on the machines or servers.

## Container Repository

A container repository is a location which hosts the containers (basically images). It is like GitHub which hosts the codes.

There are two types of containers repositories or registries:

1. Public Registry (Docker Hub)
2. Private Registry (E.g. AWS ECR)

## Virtual Machine vs Docker Container

Virtual machines are used to run the different operating systems on the host machines. E.g. A ubuntu based VM can run on Windows based host machine.

A VM virtualizes the kernel and application layer. So, the size of virtual machines is huge typically in GB’s.

But a docker container only virtualizes the application layer and uses the kernel of the host machine only. So, the size of containers (images) is small typically in MB’s.

## Docker Image and Container

**Docker Image** is a template used to create a container. A docker image consists of multiple layers where the base image is based on Linux distributions mostly.

**Base Image**: A base image acts as a base operating system for the container. Most docker images use Linux as base image like Alphine (because of small size), Ubuntu etc.

A **Container** is a running instance of the docker image. It creates an isolated environment which hosts the application code, dependencies and configurations.

## Docker Installation

Docker can be installed on Mac, Linux Based and Windows.

[Windows | Docker Docs](https://docs.docker.com/desktop/setup/install/windows-install/)

## Docker Commands

* docker ps  
  Show all running containers
* docker ps -a  
  Show all containers including the stopped containers
* docker pull <image>:<version>  
  E.g. docker pull redis:1.5.3  
  Pull the image from docker hub. If no version is specified, latest version will be pulled.
* docker run -d <image>:<version>  
  E.g. docker run -d redis  
  Run the container from image in detached mode
* docker stop <container\_id>  
  E.g. docker stop 4f1ff9e0ba06  
  Stop the running container
* docker start <container\_id>  
  E.g. docker start 4f1ff9e0ba06  
  Start the stopped container
* docker run -d -p<LocalHost Port>:<Container Port> redis  
  E.g. docker run -d -p8090:6379 redis   
  Configure the port to run the container
* docker run -d --name <container\_name> <image>   
  E.g. docker run -d --name redis-old redis  
  Name the container. By default, a random name will be assigned.
* docker run -d --name redis-new -p8090:6379 redis  
  Pass multiple arguments
* docker run -d \  
  --name redis-new \  
  -p8090:6379 \  
  redis

Environment Variables:

* docker run -d -8082:5432 \  
  -e POSTGRES\_USER=postgresuser \  
  -e POSTGRES\_PASSWORD=mypass123 \  
  --name mypostgresdb \  
  postgres

Debugging Container:

* docker logs <container\_id>  
  E.g. docker logs 90191e20902f  
  Display logs of the container
* docker exec -it <container\_id> bash   
  E.g. docker exec -it 52816c17b48b bash  
  E.g. docker exec -it 52816c17b48b sh  
  Login to the container terminal. Useful to inspect the files and run the commands inside the container.

## Docker Network

Docker Network is used to create a standalone network for multiple containers. This is useful when multiple containers have inter-communication.

* docker network create <network\_name>  
  E.g. docker network create my-network
* docker run -d -8082:5432 \  
  -e POSTGRES\_USER=postgresuser \  
  -e POSTGRES\_PASSWORD=mypass123 \  
  --name mypostgresdb \  
  --net my-network \  
  postgres

## Docker Compose

Docker Compose is a YAML file where containers are configured. It can be used to configure the containers (services) along with base image, ports, environment variables, volumes etc.

It automatically creates a network for multiple containers (services) specified in the docker-compose.yaml  
E.g.

|  |
| --- |
| version: '3'  services:  mongodb:  image: mongo  ports:  - 27017:27017  environment:  - MONGO\_INITDB\_ROOT\_USERNAME=admin  - MONGO\_INITDB\_ROOT\_PASSWORD=password    mongo-express:  image: mongo-express  restart: always  ports:  - 8080:8081  environment:  - ME\_CONFIG\_MONGODB\_ADMINUSERNAME=admin  - ME\_CONFIG\_MONGODB\_ADMINPASSWORD=password |

* docker-compose -f docker-compose.yaml up -d  
  Starts the containers specified in the file
* docker-compose -f docker-compose.yaml down  
  Stops the containers specified in the file

## Dockerfile

Dockerfile is used to build a custom image. It basically packages the application with all required dependencies and configurations. It uses base image to build new custom image.  
E.g.

|  |
| --- |
| FROM node:13-alpine  ENV MONGO\_DB\_USERNAME=admin \  MONGO\_DB\_PWD=password  RUN mkdir -p /home/app  COPY ./app /home/app  WORKDIR /home/app  RUN npm install  CMD ["node", "server.js"] |

FROM: Build an image based on the base image specified.

ENV: Configure environment variables.

RUN: Execute the commands inside the container image.

COPY: Copy the files from local host machine to container image.

WORKDIR: Sets the working directory inside the container image.

CMD: Specifies the entry point of the application. Generally, used to execute the command to run the application.

* docker build -t <app\_name>:<version> .  
  E.g. docker build -t my-app:1.0.0 .  
    
  The last . specified indicate the Dockerfile in the current folder.

## Pushing Image to Docker Repository

A custom image can be pushed to the docker repository either in a public docker hub or private docker registry (E.g. AWS Elastic Container Registry).

* Docker push <registry\_domain>/<app\_name>:<version>

### Docker Hub

* docker push my-app:1.0.0
* docker push docker.io/library/my-app:1.0.0

### Private Container Registry (AWS ECR)

1. Login to the AWS ECR. (One time activity like git login)
2. docker tag my-app:1.0.0 <Registry Domain>/my-app:1.0.0
3. docker push <Registry Domain>/my-app:1.0.0

## Deleting the Container and Image

* docker rm <container\_id>  
  Deletes the container. Stop the container first and then remove.
* docker rmi <image\_id>   
  Deletes the docker image. Remove the associated containers first and then remove.

## Docker Volume

The data is wiped off if container is restarted. Docker volumes are used to persist the data. Docker Volume uses a virtual file system in the container which is mounted on the host file system.

1. Mounts the Container Virtual File System on the Host Machine File System.   
   > docker run -v <Host File System Path>:<Container Virtual File System Path>
2. Anonymous Volumes - Creates a new folder in host file system and mounts the container path.  
   > docker run -v <Container Virtual File System Path>
3. Named Volumes (Recommended way) - Named version of anonymous volume.   
   > docker run -v <name>:<Container Virtual File System Path>

The docker volumes can be specified in the docker-compose.yaml file as well. The volume will be linked to the specified container (service).  
E.g.

|  |
| --- |
| version: '3'  services:  mongodb:  image: mongo  …  volumes:  - mongo-data:/data/db  volumes:  mongo-data:  driver: local |

In the above file, **mongo-data:/data/db**   
> mongo-data is the named volume  
> /data/db is the path where Mongo DB stores the data. This path differs from database vendor to vendor. E.g. Postgres store the data in /var/lib/postgresql/data

The docker volumes persisted on the host file system in the following paths:

1. Windows: C:\ProgramData\docker\volumes

2. Linux and Mac: /var/lib/docker/volumes

# Containerizing the Python FastAPI application

Functionality: REST API that interacts with Postgres SQL database.

Step-1: Build the application and test locally if needed.

Step-2: Create a network.

|  |
| --- |
| docker network create emp-app-net |

Step-3: Pull postgres image from docker hub and run below command to start the container.

|  |
| --- |
| docker run -d -p5432:5432 -e POSTGRES\_USER=postgres -e POSTGRES\_PASSWORD=mohan -e POSTGRES\_DB=python\_learning --name postgres-db --net emp-app-net postgres |

Step-4: Create docker file for building the app custom image.

Dockerfile

|  |
| --- |
| FROM python:3.10.11  WORKDIR /app  COPY requirements.txt .  RUN pip install --no-cache-dir -r requirements.txt  COPY . .  CMD ["uvicorn", "app.run:app", "--host", "0.0.0.0", "--port", "8000"] |

Step-5: Use the command below to build the app image.

|  |
| --- |
| docker build -t emp-app:1.0.0 . |

Step-6: Start the app container using above custom image

|  |
| --- |
| docker run -d -p8000:8000 -e DB\_HOST=host.docker.internal -e DB\_PORT=5432 -e DB\_USERNAME=postgres -e DB\_PASSWORD=mohan -e DB\_DATABASE\_NAME=python\_learning -e ENV=dev --name emp-api --net emp-app-net emp-app:1.0.0 |

All the environment variables are app specific.

**Note**: The DB\_HOST is referred by host.docker.internal (not localhost) because the postgres DB is running inside the container.

Step-7: Check the running status and logs. Make sure everything is up and running without any errors.

User docker ps, docker logs, docker exec to verify the status of the containers.

Step-8: Test the application with few transactions if required.

## Using docker-compose.yaml file

The containers (services) can be configured by docker-compose.yaml as well.

docker-compose.yaml

|  |
| --- |
| version: '3'  services:    api:      build: .      container\_name: "employee-api"      ports:        - "8000:8000"      command: uvicorn app.run:app --host 0.0.0.0 --port 8000      environment:        - DB\_HOST=host.docker.internal        - DB\_PORT=5432        - DB\_USERNAME=postgres        - DB\_PASSWORD=mohan        - DB\_DATABASE\_NAME=python\_learning        - ENV=dev      depends\_on:        - postgres    postgres:      image: postgres      container\_name: "employee-api-postgres-db"      environment:        - POSTGRES\_USER=postgres        - POSTGRES\_PASSWORD=mohan        - POSTGRES\_DB=python\_learning      volumes:        - postgres-db:/var/lib/postgresql/data      ports:        - "5432:5432"  volumes:    postgres-db: |

Noe: The api service takes Dockerfile present in the current folder and build custom image. Once the image is created, the container will be started.

* docker-compose up -d
* docker-compose down