Diving with the Whale - Docker Day V - Docker Compose and Multi-Container Apps

Author : Ahmed Ayman

Container Orchestration

Container orchestration is the automatic process of managing or scheduling the work of individual containers for applications based on micro-services within multiple clusters. The widely deployed container orchestration platforms are based on open-source versions like Kubernetes, Docker Swarm or the commercial version from Red Hat Open-shift.

If you want to manage tens, hundreds, or even thousands of containers, you don't want to do that manually of course ! That's why we need container orchestration.

Multi-Container Apps

If you have multiple containers to run your app, for example you have the database in one container, and other back-end container, and front-end container, so you need to run all of these and connect them using networks and map the right ports and volumes and all that stuff! sounds like a lot of work right ?! here is where Docker Compose comes in the picture , it allows you to setup every thing in a yaml file then run only one command to start your multi-containers app!

Docker Compose

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application’s services. Then, with a single command, you create and start all the services from your configuration. To learn more about all the features of Compose, see the list of features.

It's basically composing the parts of the application and configuring them to work beautifully together with one command!

It simplifies the managing from multiple repeated commands to a yaml file and one command.

▼ **Install Docker Compose on Linux**

Installation Guide From Docker Docs.

1. Download the current stable release of Docker Compose

```
sudo curl -L "https://github.com/docker/compose/releases/download/1.29.1/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/dock
```

2. Apply executable permissions to the binary

```
sudo chmod +x /usr/local/bin/docker-compose
```

and that's it!

3. Check if the installation is successful

```
docker-compose --version
# you should get smth like this
# - docker-compose version 1.29.1, build 1110ad01
```

▼ **Docker-Compose YAML File**

DOCKER-COMPOSE YAML FILE CHEAT SHEET

```
version: "3.9"

services:
  db:
    image: postgres
    volumes:
      - ./data/db:/var/lib/postgresql/data
    environment:
      - POSTGRES_DB=postgres
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
# docker run -v ./data/db:/var/lib/postgresql/data -e k=v postgres
  web:
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/code
    ports:
      - "8000:8000"
    depends_on:
      - db
# docker build -t tag .
# docker run tag -v .:/code -p 8000:8000 python manage.py runserver 0.0.0.0:8000
```

This is a simple docker-compose yaml file for a sample django app that uses postgres database.

so let's break it down !

```
version: "3.9"
```

Specifies the version of docker compose

```
services:
  db:

  web:
```

it means we have two services, which is kinda means we have two docker images to build, one called db and the other called web

let's dive into the db service

```
db:
    image: postgres
    volumes:
      - ./data/db:/var/lib/postgresql/data
    environment:
      - POSTGRES_DB=postgres
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
```

image defines the image we want to build from, and the same as docker run command, if the image doesn't exist locally it'll get downloaded from docker-hub, so here the image name is postgres

volumes defines the volume mapping rules so in this example we're saying map ./data/db in the local host to /var/lib/postgresql/data in the container. It's the same as running your container with docker run -v

environment sets the environment variables for the container, the same as using -e when running your container with docker run, and it gets an array of key value pairs

so to wrap it up, the db service says run a container from the image postgres, and map ./data/db in the host to /var/lib/postgresql/data in the container, and set POSTGRES_DB=postgres, POSTGRES_USER=postgres, POSTGRES_PASSWORD=postgres as env variables.

if we want to translate it to a docker command it'll be like this

```
# create a env.list file contains all the env variables
echo "POSTGRES_DB=postgres \nPOSTGRES_USER=postgres\nPOSTGRES_PASSWORD=postgres" > env.list
docker run -v $(pwd)/data/db:/var/lib/postgresql/data --env-file env.list postgres
```

now for the other service web, let's break it down

```
web:
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/code
    ports:
      - "8000:8000"
    depends_on:
      - db
```

build gets a path to the directory containing the docker file to build, so the difference between build and image is that image specifies an already built image, build specifies the docker-file to build an image from.

command gets a Linux command , this overwrites the CMD command inside the docker file

volumes maps a host directory to container directory

ports maps/binds a host port to a container exposed port, so here we're mapping port 8000 in the host machine to port 8000 in the container

depends_on specifies which services must be already running before running this container

so let's translate this service to a docker command

```
# build the image
docker build -t img .
# run the container with the mapped port and volume
docker run -p 8000:8000 -v $(pwd):/code img python manage.py runserver 0.0.0.0:8000
```

And there is so much more you can do with docker-compose yaml files.

▼ **Docker-Compose Commands**

Run Docker Compose Containers

now that we have a docker-compose.yaml file we can run them using one command without paying attention to any mapping , and can get a specific service to start it only as an argument, if no arguments are specified all the services will start

```
# change your directory to where the yaml file is
cd docker-compose-test
docker-compose up
# thats it really!
```

an important option of the up command is —scale=n this option will scale (i.e. will create more than 1 container from a specified service)

```
# docker-compose up --scale $service=$n
docker-compose up --scale web=5
```

and here is one of the powerful features of docker-compose!

Stop/Remove Docker Compose Containers

this will stop all the running containers and removes containers, networks, volumes, and images created by up.

```
docker-compose down
```

List Running Docker Compose Containers

```
docker-compose ps
```

Stop Docker Compose Containers

this will stop all the containers created by up but will not remove anything, and can get a specific service to start it only as an argument, if no arguments are specified all the services will stop

```
docker-compose stop
```

Start Docker Compose Containers

this will start a docker compose existing containers and can get a specific service to start it only as an argument, if no arguments are specified all the services will start

```
docker-compose start
```

Pause Docker Compose Containers

this will pause all the containers created by up and can get a specific service to start it only as an argument, if no arguments are specified all the services will pause

```
docker-compose pause
```

Un-pause Docker Compose Containers

this will un-pause all the containers created by up and can get a specific service to start it only as an argument, if no arguments are specified all the services will un-pause

```
docker-compose unpause # [service, ...]
```