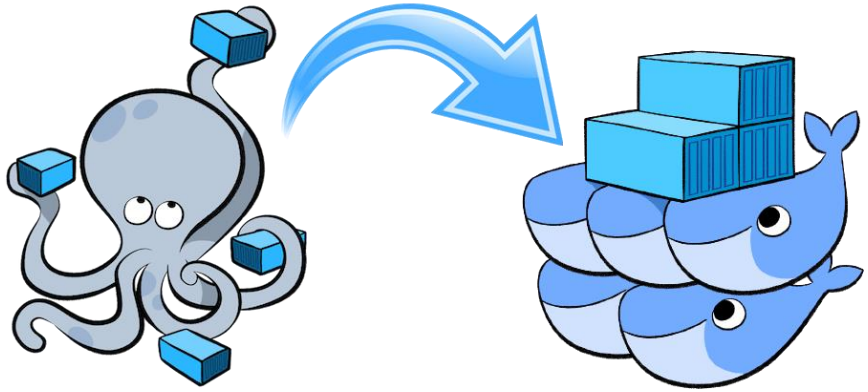
Docker Compose

Mithun Technologies



devopstrainingblr@gmail.com
+91-9980923226

Let's talk about real life applications first!

- One application consists of multiple containers.
- One container is dependent on another.
- Mutual dependency/ startup order.
- Process involves building images and then deploy them
- Long docker run commands
- Complexity is proportional to the number of containers involved.

Docker Compose

- Tool for defining and running multi-container Docker application.
- It is a YML file.
- Compose contains information about how to build the containers and deploy containers.
- Integrated with Docker Swarm.
- Competes with Kubernetes.

Note: Generally the containers in an application built using Docker Compose will all run on the same host. Managing containers running on different hosts usually requires an additional tool, such as Docker Swarm or Kubernetes.

Installation

You can run Compose on macOS, Windows, and 64-bit Linux.

Prerequisites

- Docker Compose relies on Docker Engine for any meaningful work, so make sure you have Docker Engine installed either locally or remote, depending on your setup.
- On desktop systems like Docker for Mac and Windows, Docker Compose is included as part of those desktop installs.
- On Linux systems, first install the Docker for your OS as described on the Get Docker page, then come back here for instructions on installing Compose on Linux systems.

1) Run this command to download the latest version of Docker Compose:
$sudo curl -L "https://github.com/docker/compose/releases/download/1.22.0/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

Use the latest Compose release number in the download command.
The above command is an example, and it may become out-of-date. Please refer below link in case of any issues with installation. https://docs.docker.com/compose/install/

2) Apply executable permissions to the binary:
$sudo chmod +x /usr/local/bin/docker-compose

3) Test the installation.
$docker-compose --version

Note: Generally the containers in an application built using Docker Compose will all run on the same host. Managing containers running on different hosts usually requires an additional tool, such as Docker Swarm or Kubernetes.

Basic Usage

1. Open docker-compose.yml in a text editor and add the following content:

```
version: '3'

services:
  db:
    image: mysql:5.7
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress

  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    ports:
      - "8000:80"
    restart: always
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
volumes:
    db_data:
```

2. Save the file and run Docker Compose from the same directory:
docker-compose config (To validare yml file)
docker-compose up –d

This will build and run the db and wordpress containers. Just as when running a single container with docker run, the -d flag starts the containers in detached mode.

3. You now have a WordPress container and MySQL container running on your host. Navigate to http://<DockerServerPublicIP>:8000/wordpress in a web browser to see your newly installed WordPress application.

You can also use docker ps to further explore the resulting configuration:

docker ps
4. Stop and remove the containers:
docker-compose down

Compose File Syntax

A docker-compose.yml file is organized into four sections:

Directive	Use
version	Specifies the Compose file syntax version.
services	In Docker a service is the name for a “Container in production”. This section defines the containers that will be started as a part of the Docker Compose instance.
networks	This section is used to configure networking for your application. You can change the settings of the default network, connect to an external network, or define app-specific networks.
volumes	Mounts a linked path on the host machine that can be used by the container.

Most of this guide will focus on setting up containers using the services section. Here are some of the common directives used to set up and configure containers:

Directive	Use
image	Sets the image that will be used to build the container. Using this directive assumes that the specified image already exists either on the host or on Docker Hub.
build	This directive can be used instead of image. Specifies the location of the Dockerfile that will be used to build this container.
restart	Tells the container to restart if the system restarts.
volumes	Mounts a linked path on the host machine that can be used by the container

Directive	Use
environment	Define environment variables to be passed in to the Docker run command.
depends_on	Sets a service as a dependency for the current block-defined container
port	Maps a port from the container to the host in the following manner: host:container
links	Link this service to any other services in the Docker Compose file by specifying their names here.

Caution

The example docker-compose.yml above uses the environment directive to store MySQL user passwords directly in the YAML file to be imported into the container as environment variables. This is not recommended for sensitive information in production environments. Instead, sensitive information can be stored in a separate .env file (which is not checked into version control or made public) and accessed from within docker-compose.yml by using the env_file directive.

Docker Compose with Spring Boot, MongoDB Application


```
version: '3.1'

services:
  springboot:
    image: dockerhandson/spring-boot-mongo:latest
    restart: always # This will be ignored if we deploy in docker swarm
    container_name: springboot
    environment:
    - MONGO_DB_HOSTNAME=mongo
    - MONGO_DB_USERNAME=devdb
    - MONGO_DB_PASSWORD=devdb1234
    ports:
      - 8080:8080
    working_dir: /opt/app
    depends_on:
      - mongo
    deploy:  # This will be considered only in docker swarm.
      replicas: 2
      update_config:
        parallelism: 1
        delay: 20s
      restart_policy:
        condition: on-failure
        delay: 10s
        max_attempts: 5
    networks:
    - springappnetwork

  mongo:
    image: mongo
    container_name: springboot-mongo
    environment:
    - MONGO_INITDB_ROOT_USERNAME=devdb
    - MONGO_INITDB_ROOT_PASSWORD=devdb1234
    volumes:
      - mongobkp:/data/db
    restart: always
    networks:
    - springappnetwork

volumes:
  mongobkp:
    driver: local

networks:
  springappnetwork:
    driver: bridge
```

Persistent Data Storage

Storing MySQL,MongoDB or PostgreSQL data directly inside a container is not recommended. Docker containers are intended to be treated as ephemeral: your application's containers are built from scratch when running docker-compose up and destroyed when running docker-compose down. In addition, any unexpected crash or restart on your system will cause any data stored in a container to be lost.

For these reasons it is important to set up a persistent volume on the host that the database containers will use to store their data.

```
version: '2'
services:
  mongodb:
    image: mongo
    container_name: mongo
    volumes:
      - data:/data/db
    restart: always
  volumes:
  data:
    external: true
```

external: true tells Docker Compose to use a pre-existing external data volume. If no volume named data is present, starting the application will cause an error. Create the volume:

```
docker volume create --name=data
```

Start the application as before:

```
docker-compose up -d
```

Attach bash to the running container

docker exec -i -t <name or id of the contianer> /bin/bash

How to update containers

Making some changes in docker-compose file

In case when we update/add/remove any property in docker-compose.yml file the docker-compose up -d will recreate and will restart our running containers. The output will look like:

Recreating springboot-mongo ...
Recreating springboot-mongo ... done
Recreating springboot ... Recreating springboot ... Done

Update external images

When Docker pulls any image first time, the image and running containers from this image leave without changes. Docker images often get some updates like hot/security fixes, new versions, etc. To update any external image (by external I mean any image which will be pulled from any Docker registry) we need use pull command like this:

docker-compose pull

It will verify updates for any used image in docker-compose.yml file and download it.

The output will look like:
Pulling mongo (mongo:latest)... latest: Pulling from library/mongo c4bb02b17bb4:
Pull complete 158f54c96e9a:
Pull complete Digest:
sha256:d16539343d6b47ac150a9fae8e1278253e5f00a4c1d9d3f4a3858bd90d5f3097
Status: Downloaded newer image for mongo:latest

Update local images

To update our local image we need do the following:
Run command
docker-compose build

It will verify changes in Dockerfile and recreate image in case when Dockerfile was changed.
run docker-compose up -d command.
It will recreate and restart only our application container:

docker-compose commands

Following commands can be used with docker-compose <command> d

Ex: docker-compose up

To get more help about particular command > docker-compose <command> --help

Ex: docker-compose up --help

```
Commands:
  build              Build or rebuild services
  bundle             Generate a Docker bundle from the Compose file
  config             Validate and view the Compose file
  create             Create services
  down               Stop and remove containers, networks, images, and volumes
  events             Receive real time events from containers
  exec               Execute a command in a running container
  help               Get help on a command
  images             List images
  kill               Kill containers
  logs               View output from containers
  pause              Pause services
  port               Print the public port for a port binding
  ps                 List containers
  pull               Pull service images
  push               Push service images
  restart            Restart services
  rm                 Remove stopped containers
  run                Run a one-off command
  scale              Set number of containers for a service
  start              Start services
  stop               Stop services
  top                Display the running processes
  unpause            Unpause services
  up                 Create and start containers
  version            Show the Docker-Compose version information
```

Using Multiple Docker Compose Files

Use multiple Docker Compose files when you want to change your app for different environments (e.g., dev, staging, and production) or when you want to run admin tasks against a Compose application. This gives us one way to share common configurations.

Docker Compose already reads two files by default: docker-compose.yml and docker-compose.override.yml. The docker-compose-override.yml file can be used to store overrides for existing services or define new services. Use multiple files (or an override file with a different name) by passing the -f option to docker-compose up (order matters):

$ docker-compose up -f my-override-1.yml my-overide-2.yml
When two configuration options match, the most recent value either replaces or extends the first.

In the following example, the new value overrides the old, and command runs my_new_app.py:

```
# original service
command: python my_app.py

# new service
command: python my_new_app.py
```

Different Environments

Start with your base Docker Compose file for your application (docker-compose.yml):

```
web:
  image: "my_dockpy/my_django_app:latest"
  links:
    - db
    - cache
db:
  image: "postgres:latest"
cache:
  image: "redis:latest"
```

On our development server, we want to expose some ports, mount our code as a volume, and build our web image (docker-compose.override.yml):

```
web:
  build: .
  volumes:
    - ".:/code"
  ports:
    - "8883:80"
  environment:
    DEBUG: "true"
db:
  command: "-d"
  ports:
    - "5432:5432"
cache:
  ports:
    - "6379:6379"
```

docker-compose up automatically reads the override file and applies it. We also need a production version of our Docker Compose app, and we want to call that docker-compose.production.yml:

```
web:
  ports:
    - "80:80"
  environment:
    PRODUCTION: "true"

cache:
  environment:
    TTL: "500"
```

When you want to deploy your production file, simply run the following:

```
$ docker-compose -f docker-compose.yml -f docker-compose.production.yml up -d
```

Note: Docker Compose reads docker-compose.production.yml but not docker-compose.override.yml.

Questions ?

Thank You