# **Docker architecture**

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface. Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.

## **Docker daemon**

The Docker daemon (**dockerd**) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

## **Docker client**

The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as **docker run**, the client sends these commands to **dockerd**, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

## **Docker Desktop**

Docker Desktop is an easy-to-install application for your Mac, Windows or Linux environment that enables you to build and share containerized applications and microservices. Docker Desktop includes the Docker daemon (dockerd), the Docker client (docker), Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper.

## **Docker registries**

A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker looks for images on Docker Hub by default. You can even run your own private registry.

When you use the docker pull or docker run commands, Docker pulls the required images from your configured registry. When you use the docker push command, Docker pushes your image to your configured registry.

## **Docker objects**

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects.

## **Images**

An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

## **Containers**

A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.

A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that aren't stored in persistent storage disappear.

## **The underlying technology**

Docker is written in the Go programming language and takes advantage of several features of the Linux kernel to deliver its functionality. Docker uses a technology called namespaces to provide the isolated workspace called the container. When you run a container, Docker creates a set of namespaces for that container.

These namespaces provide a layer of isolation. Each aspect of a container runs in a separate namespace and its access is limited to that namespace.

# **Install Docker**

* Install using the convenience script.

curl -fsSL https://get.docker.com -o get-docker.sh

sudo sh ./get-docker.sh

* Or you can visit below link for steps:

[https://docs.docker.com/engine/install/ubuntu/#install-using-the-convenience-script](https://docs.docker.com/engine/install/ubuntu/%23install-using-the-convenience-script%20)

* User can not execute docker command by default, For using docker from user you need to add user to docker group. Use below command to add user in docker group.

usermod -aG docker <user name>

# **Docker Container Commands**

* You can user **container id** and **container name** interchangeably in any command.
* You can use **docker run** or **docker** **container run** interchangeably.
* Default path for docker installation directory

/var/lib/docker/

## **Check Version of Docker Engine**

### **Complete server/client version details**

docker version

### **Only version details**

docker –version

### **Complete information of docker engine**

docker info

### **List of all command**

docker

## **List all running containers**

docker container ls or docker ps

**Options**

* -a - list all running & stopped containers
* -q - show only container id
* –no-trunc - display full container id / image id

## **Create/Run Container**

docker run -it -d -p <host port:container port> --name <container name> <image name> <command to execute>

**Options**

* -it - interactive mode with open terminal
* --detach/-d - run container in background or in detach mode
* --publish/-p <host\_port:container\_port> - port number
* –name <container name> - provide container name
* -e <variable> - to use environment variable
* --network <network name/id> - to attached container with specific network
* -c <command or script> - run command or script, this option will not give terminal back until process is done.
* -rm - to remove container automatically after work done.
* -v <folder name>:<container dir path> - mount volumes to container.
* run any preconfigured image in detach mode only.

Ex: docker run -it -d -p 80:80 --name <container name> <image name> httpd

## **Start/Stop/Restart container**

docker start/stop/restart <container id/container name>

* **start**: start an existing container
* **run**: start a new container always

## **Login/Execute command into container**

docker attach < container id/container name>

* do not use this method as container will stop at exit.
* ctrl+p+q to exit container without stopping it
* Execute command into container without login into it.

docker exec <container id> <command>

Ex: docker exec web1 echo “hi”

**Options**

* -it - interactive mode & bash command for login into container.

use this method to login as container will not stop after exit from container.

Ex: docker exec -it <contaner\_id> bash

## **Remove/Delete Containers**

docker rm <space separated container ids>

**Options**

* rm -f - remove container forcefully.
* Delete all container (running & stopped) using nested command.

docker rm -f `docker ps -a -q`

## **Details of container**

Docker inspect <container id>

**Options**

* -f <format string> to format and get specific info from entire json

Ex: '{{.NetworkSettings.IPAddress}}' - for IP address of container

Or you can search keyword using “|” in command.

* You can only use this command for running container

## **Rename container**

docker rename <old name> <new name>

## **Check container logs**

docker logs <container id>

## **Check process running inside the container**

docker top <container id>

## **Containers Resource Consumption**

docker container stats <container id>

## **Limit cpu & mem**

* edit /etc/default/grub file and insert below line then reboot host system

GRUB\_CMDLINE\_LINUX="cdgroup\_enable=memory swapaccount=1"

* use below option with docker run command for memory hard limit, swap memory limit, memory soft limit and cpu limit for a container

docker run -it --memory=<"memory\_limit"> --memory-swap=<"memory\_limit"> --memory-reservation=<"memory\_limit"> --cpus=<"cpu limit"> <docker\_image>

Ex: docker run -it --memory="1g" --memory-swap="2g" --memory-reservation="750m" --cpus="1.0" ubuntu

--memory - hard limit for memory

--memory-swap - swap memory limit

--memory-reservation - soft limit for memory

--cpus - set cpu limit

# **Expose/Bind Port**

To access container, we need to expose or allow container port.

Use below method with docker run command for expose container port service.

* Container/website will be access only from container IP address, not host IP address.

docker run -d httpd

* Container port 80 will be bind to random port of host system.

docker run -d --name web1 -p 80 httpd

* Container port 80 will be bind to port 80 of host system

docker run -d --name web1 -p 80:80 httpd

* -p 80:80 🡪 host\_port:container\_port
* use multiple -p option for multiple port mapping.
* Host system random port will be bind to container expose port.

docker run -d --name web1 -P httpd

# **Docker Images**

* In Docker, everything is based on Images.
* An image is a combination of a file system and parameters.
* Images contains the binaries and dependencies.
* Images Contains the data require to run the Container.

### **List Docker Images**

docker images

### **Download image from Docker hub**

docker pull <image name>:<tag name>

* if tag not provided then latest version will be downloaded.
* provide tag if you want to download specific version.

### **Upload image to Docker hub**

docker image push <image name>

### **Remove downloaded image**

docker rmi <image name>:tag or <image id>

**Options**

* -f - for forcefully remove
* -q - to get full image id
* Remove all images at once.

docker rmi -f `docker images -q`

* if any container is created from image, then image will not delete it will un-tag.
* if container is stopped then you can delete image with -f option.

## **Docker Central Repositories**

* Docker Images can be Downloaded from Docker Hub

<https://hub.docker.com>

* Images can be committed with changes and have multiple versions. If you don't provide a specific version number, the client defaults to latest.

### **Login/Logout into Docker Repository**

* Login to docker hub repository

docker login

* you need to provide username & password of docker hub account
* Logout from docker hub repository

docker logout

## **Docker Images Differentiation**

### **Base Image**

* Images that have no parent image, usually images with an OS like ubuntu, busybox or debian.

### **Child Images**

* Images that build on base images and add additional functionality.

### **Official images**

* Images that are officially maintained and supported by the folks at Docker.
* These are typically one-word long.

### **User images**

* Images created and shared by users like you and me. They build on base images and add additional functionality.
* Typically, these are formatted as **user/image-name**.

## **Docker Image Layers**

* Each image consists of a series of layers. Docker makes use of union file systems to combine these layers into a single image.
* Union file systems allow files and directories of separate file systems, known as branches.
* History command shows different layers, command used for each layer, and the exact size of each image.

### **Show Image Layers**

docker history <image name>

## **Docker Image Tagging**

* Docker tags convey useful information about a specific image version/variant.
* Tags added to image during the Building of Image.
* Images can be tagged explicitly with the help of Tag Command.

docker tag SOURCE\_IMAGE[:TAG] TARGET\_IMAGE[:TAG]

* When you don’t specify Docker Image tag. By default, it takes ‘latest’ tag.
* You can rename image name or give image name by using tag command.

docker tag <old\_image\_name\_or\_image\_id>:<tag> <new\_image\_name>:<tag>

* if you want to upload your image to Docker hub account then you need to include account name in your Docker image name.

Ex: docker tag centos:v1 account\_name/centos:v1

* Give 2 tage to image (1) Latest and (2) Your version for version management is Docker hub. Push both images to Docker hub, latest tag will be overwritten in Docker hub, if anyone download image without version tag then latest will be downloaded.

## **Create custom image from running container**

docker commit -m "write message information" <container id>or<container name> <your image name>:<tag>

* you can provide any name to your image.
* latest tag will be automatically inserted if not provided.

## **Backup & Load image to your local system**

### **Backup**

docker save --output <filename> <image name>:<tag>

* file name standard: <image name>.<version or tag>.tar
* do not use image id for this command as null value will comes after restoring, in repository & tag value.
* use image name & tag in backup command to get details after restoring.

### **Load image**

docker load --input <file name>

## **Docker File**

* Docker can build images automatically by reading the instructions from a Dockerfile.
* A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.
* standard name of docker file "Dockerfile"
* use capitol latter for directives in docker file.

### **Docker File Directives**

* **FROM**

FROM <image name with tag>

* Ex: FROM centos:7
* name of base image or which image you want to use.
* **LABEL**

LABEL <label key value pair>

* Ex: LABEL maintainer:"vikas"
* give information like author or company name.
* use key-value pair.
* **RUN**

RUN <command>

* Ex: RUN adduser vikas
* use for run command like mkdir, ls, adduser
* RUN will execute while image is created.
* if you define multiple command, all command will be executed
* RUN instruction will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the Dockerfile.
* **USER**

USER <user name>

* Ex: USER vikas
* default login user for container
* build perform all step forward using this user while image is created.
* to avoid any permission issue, use this directive at last.
* **CMD**

CMD <command>

* Ex: CMD echo "hello"
* CMD instruction should be used to run the software contained by your image, along with any arguments.
* CMD will run when container is created.
* if you define multiple CMD only last one will be executed.
* CMD can be overwrite if you define any command in docker run command to execute.
* use CMD at last in docker file.
* **ENTRYPOINT**

ENTRYPOINT <command>

* Ex: ENTRYPOINT echo "hello"
* ENTRYPOINT instruction should be used to run the software contained by your image, along with any arguments.
* ENTRYPOINT will run when container is created.
* if you define multiple ENTRYPOINT only last one will be executed.
* ENTRYPOINT cannot be overwrite if you define any command in docker run command.
* use ENTRYPOINT at last in docker file.
* you need to run any service in foreground using CMD or ENTRYPOINT for container to run continuously.
* **WORKDIR**

WORKDIR <directory path>

* Ex. WORKDIR /var
* define default work directory for container when container is created.
* default directory will be “/” if WORKDIR not provided
* **ENV**

ENV <env variable>

* Ex. ENV PATH /usr/local/nginx/bin:$PATH
* sets the environment variable to the value.
* **EXPOSE**

EXPOSE <port no>

* Ex. EXPOSE 80 443
* to expose container port
* you can also define multiple ports.
* **COPY**

COPY <file or folder path in host system> <path of container dir>

* Ex. COPY /home/index.html /var/www/html/
* to copy file or folder in container directory while creating image
* **ADD**

ADD <file/folder path in host system or any link> <path of container dir>

* Ex. ADD https://www.google.com/image1.png /var/www/html/
* to copy file or folder or any links document in container directory while creating image
* **VOLUME**

VOLUME <dir path>

* Ex. VOLUME /var/lib/postgresql/data
* VOLUME instruction should be used to expose any database storage area, configuration storage, or files/ folders created by your docker container.

## **Build image from docker file**

docker build -t <image name>:<tag> <dir where docker file is present>

* Option
* -t - to mention tag to the image
* -f - path and file name of docker file if custom name provided (like mydockerfile)
* --no-cache - use this option for not using cache for creating layers, or create all layers once again
* Image Name - Name you want to give to your image
* Tag - tag you want to give to your image
* dir - the directory where the docker file is present, if you are running command from same dir where docker file is present then don’t have to provide dir

# **Volumes**

* Volumes are stored in a part of the host filesystem which is managed by Docker.
* Volumes are created and Managed by Containers.
* When you create a volume, it is stored within a directory on the Docker host machine.
* Volumes cannot be removed when user destroy the containers.
* volume directory path in host system:

/var/lib/docker/volumes/

### **list all volumes**

docker volume ls

### **Create volumes**

docker volume create <volume name>

### **Inspect volume**

docker volume inspect <volume name>

### **Remove Volume**

docker volume rm <volume name>

## **Data Volumes mount**

* **Map container path to host random path**

-v <container dir path>

Ex: docker run –d –name nginx **-v /var/www/html** nginx

* It is mapping container path (dir) to host system random path, this random path will be in volumes directory.
* **Map host system fix path to container path**

-v <host dir path>:<container dir path>

Ex: docker run –d –name nginx **-v /mydata:/var/www/html** nginx

* host system fix path (/mydata) will map with container path (/var/www/html).
* if you use path other then /var/lib/docker/volumes then it will not reflect in docker volume ls command.
* **Map host system volume path to container path**

-v <folder name>:<container dir path>

Ex: docker run –d –name nginx **-v web1:/var/www/html** nginx (host system path is /var/lib/docker/volumes/web1)

OR

docker run –d –name nginx **--mount source=web1,target=/var/www//html** nginx

* create or attach folder from /var/lib/docker/volumes/ dir to container path.
* This is the standard method of defining volumes.
* **Map read only file system**

-v <folder name>:<container path>:ro

Ex: -v web1:/var/www/html:ro

* use this method if data needs to be only read from container or you want to attach same folder to multiple containers to assure data change will not affect all container.

## **Data Bind Mount**

* Bind mount means a file or directory on the host machine is mounted into a container.
* Bind mounts may be stored anywhere on the host system.
* Non-Docker processes on the Docker host or a Docker container can modify them at any time.
* Bind Mount can’t be use in DockerFile.
* Used for Sharing configuration files, source code or build artifacts from the host machine to containers.

**Use below option in docker run command for bind mount**

--mount type=bind,source=<source folder name>,target=<container path>

Ex: docker run -d --mount type=bind,source=/root/web,target=/var/www/html httpd

# **Networks**

## **Network Types**

### **Bridge Networking**

* By default, containers are connected to a Docker bridge network, allowing them to communicate with each other on the same host.
* However, containers on different hosts cannot communicate without additional configuration.

### **Overlay Networking**

* This model enables communication between containers across different hosts.
* Technologies like VXLAN or IPSec are often used to create virtual networks that span multiple hosts.

### **Container Network Interfaces (CNI)**

* CNI is a specification that defines how container runtimes interact with networking plugins.
* It allows different container runtimes to be combined with various networking solutions.

### **Default network in Docker**

There will be 3 network created by default

NETWORK ID NAME DRIVER SCOPE

fa17e88926a3 bridge bridge local

38c06d007333 host host local

77e355429f3b none null local

* default bridge network range:

Subnet:172.17.0.0/16

Gateway:172.17.0.1

* we can communicate using IP address but not container name in default bridge network.
* we can communicate using IP address & container name in custom bridge network.

## **List down all network interface**

docker network ls

-f <key=value> - filter networks with key-value pair

Ex: docker network ls -f driver=bridge

## **Info about one or more network interface (inspect)**

docker network inspect <network name>

## **Create your custom network**

docker network create --subnet <range> --gateway <ip> <your network name>

Ex: docker network create --subnet 192.168.2.0/24 --gateway 192.168.2.1 mybridge

### **Use below option in docker run command for using custom network**

--net/--network <network name>

Ex: docker run -d –net mybridge --name web1 httpd

--ip <IP Address> - option to assign IP address to your container

Ex: docker run -d --net mybridge -ip 192.168.2.100 --name web1 httpd

* **Note:** you can only assign fix IP in custom network.

## **Connect container to network**

docker network connect <network name> <container name>

Ex: docker network connect bridge web1

* **Note:** you can connect container with more than 1 network

## **Disconnect container to network**

docker network disconnect <netwok name> <container name>

Ex: docker network disconnect bridge web1

* **Note:** you can disconnect container from all networks.

## **find the traffic port and protocol on container**

docker container port <container id>

## **Remove Network**

docker network rm <network name>

Ex: docker network rm mybridge

## **Notes**

* if you want to communicate with name in default network then create communicate with container first ex. db container then create communicate from container ex. web container.
* in creation of communicate from container user --link <container name> option in docker run command.

Ex: docker run -it --link db1 --name web1 centos bash

* this communication will only work only one-way. (Web 🡪 DB)
* Containers uses DNS to communicate. Containers don’t use IP address to Communicate.

# **Example of WordPress website with DB**

DB Server

requirement:

IP (container name), DB Username, DB Password, Database Name

data folder of mysql:/var/lib/mysql

execute below command to create DB container

docker run -d --name db -e MYSQL\_ROOT\_PASSWORD=Redhat123 -e MYSQL\_DATABASE=wordpress -v db:/var/lib/mysql mysql

Web Server

requirement:

Port no to Expose, DB container name and IP address, DB Username, DB Password, DB Name

Execute below command to create web container

docker run -d --name wordpress -p 8080:80 --link=db -e WORDPRESS\_DB\_HOST=db -e WORDPRESS\_DB\_USER=root -e WORDPRESS\_DB\_PASSWORD=Redhat123 -e WORDPRESS\_DB\_NAME=wordpress wordpress

Note: you can find environment variable details for container in Docker Hub image details.

# **Docker Compose**

* Single Command for all Image Building and Container Creation.
* Manage the whole Application Lifecycle.
* Start, Stop and Rebuild Services.
* View the Status of Running Services.
* Stream the Log output of Running Services.
* Run a one-off command on a service.
* file name should be “docker-compose.yaml”
* for more information about Docker Compose

[https://docs.docker.com/compose/compose-file/#resources](https://docs.docker.com/compose/compose-file/%23resources)

## **Install Docker Compose**

curl -L https://github.com/docker/compose/releases/download/1.29.2/docker-compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose

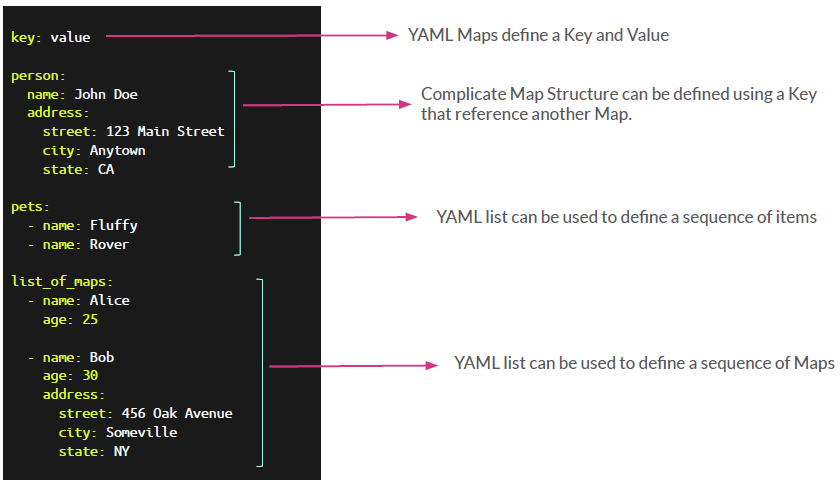
chmod +x /usr/local/bin/docker-compose

<https://www.digitalocean.com/community/tutorials/how-to-install-docker-compose-on-ubuntu-16-04>

you can find installation step from google search for different host environment.

## **YAML File Overview**

* YAML files are composed of maps and lists.
* Indentation matters (be Consistent).
* Always use spaces
* **Maps:**
* name: value pairs
* Maps can contain other maps for more complex data structure
* **Lists:**
* Sequence of Items
* Multiple Maps can be defined in list



## **Docker Compose Commands**

* Check docker compose version

docker-compose –version

* Build Images

docker-compose build

* Push Images to docker hub

docker-compose push

* Up & run docker compose file containers

docker-compose up -d

* Down docker compose file containers

docker-compose down

* View the Logs of all Docker Compose Services

docker-compose logs

* View the Logs of Docker Compose Specific Services

docker-compose logs <service name>

* Limit Container log output

docker-compose logs --tail=5

* Follow the Container log output (continues log )

docker-compose logs – follow

* Login into Docker Compose Service container

docker-compose exec <service name> <shell>

* **Notes:**
* you can use custom file name for docker-compose for this use -f <file name> option in docker-compose command.
* you need to execute docker-compose command from docker-compose file location to work this command or you can specify path with -f option.

## **Docker Compose File**

* Docker Compose is a tool for defining and running multi-container Docker applications.
* It allows you to define an entire application stack, including services, networks, and volumes, in a single file called ‘docker-compose.yml’.

### **Components of a Docker Compose file**

* **Version**
* The version of the Docker Compose file syntax.
* This version indicates which features and syntax are supported.
* **Services**
* This section defines the various containers (services) that make up your application.
* Each service has a name, and under each service, you can configure parameters such as the Docker image to use, environment variables, ports to expose, volumes to mount, and more.

services:

  web:

    image: nginx:latest

    ports:

      - "80:80"

* **Volumes**
* This section allows you to define named volumes or bind mounts for your services.
* Volumes are used to persist data between container restarts.

volumes:

  data-volume:

* **Networks**
* This section lets you define custom networks and connect services to them.
* This is useful for controlling communication between services.

networks:

  custom-network:

* **Environment Variables**
* You can set environment variables for your services using the ‘environment’ key.
* This is useful for configuring your applications dynamically.

environment:

  -NODE\_ENV=production

* **Ports**
* This key allows you to map ports from the host to the container.
* This is essential for accessing services from outside the Docker environment.

    ports:

      - "80:80"

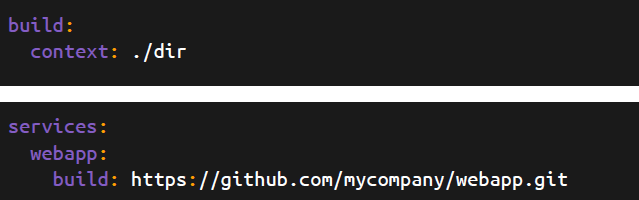
* **Command**
* This key lets you override the default command for the container.
* This is useful when you need to specify how the container should start.

command: npm start

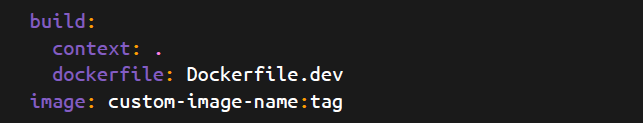
## **Build Images with Docker Compose**

### **Docker Compose Build Properties**

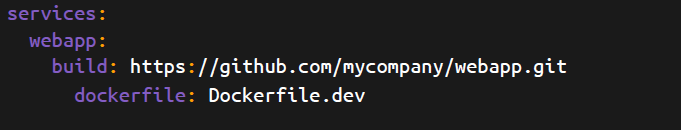
* **build/context**
* It defines either a path to a directory containing a Dockerfile, or a URL to a git repository.



* **Image**
* Overrides the image name specified in the Dockerfile.



* **Dockerfile**
* Specifies an alternative Dockerfile for the build.



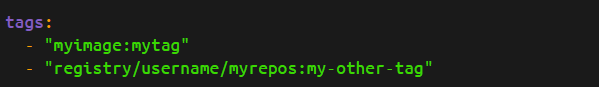
* **Args**
* Passes build arguments to the build context. Useful for dynamically setting values during the build.







* **Tags**
* This defines a list of tag mappings that must be associated to the build image.



## **Example of Wordpress Application**

version: '3' #Version of YML file

services:

  # MySQL Service

  db:

    image: mysql:5.7

    container\_name: mysql-container

    restart: always

    environment:

      MYSQL\_ROOT\_PASSWORD: mypassword

      MYSQL\_DATABASE: wordpress

      MYSQL\_USER: wordpressuser

      MYSQL\_PASSWORD: wordpress

    volumes:

      - mysql\_data:/var/lib/mysql

    networks:

      - wordpress\_network

    ports:

      - "3306:3306"

  # WordPress Service

  wordpress:

    depends\_on:

      - db

    image: wordpress:latest

    container\_name: wordpress-container

    restart: always

    environment:

      WORDPRESS\_DB\_HOST: db:3306

      WORDPRESS\_DB\_USER: wordpressuser

      WORDPRESS\_DB\_PASSWORD: wordpress

    #mem\_limit: 300m

    #mem\_reservation: 100m

    #cpus: 0.5

    volumes:

      - wordpress\_data:/var/www/html

    ports:

      - "8080:80"

    networks:

      - wordpress\_network

volumes:

  mysql\_data:

  wordpress\_data:

networks:

  wordpress\_network:

    driver: bridge