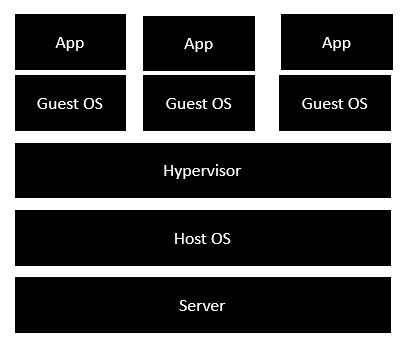
**What is Docker?**

Docker is a container management service. The keywords of Docker are **develop, ship** and **run** anywhere. The whole idea of Docker is for developers to easily develop applications, ship them into containers which can then be deployed anywhere.

Docker is an open platform for developers and sysadmins to build, ship, and run distributed applications. Consisting of Docker Engine, a portable, lightweight runtime and packaging tool, and Docker Hub, a cloud service for sharing applications and automating workflows, Docker enables apps to be quickly assembled from components and eliminates the friction between development, QA, and production environments. As a result, IT can ship faster and run the same app, unchanged, on laptops, data center VMs, and any cloud.

# Architecture



**This command is used to stop the Docker daemon process.**

**Syntax**

**service docker stop**

**Return Value**

**A message showing that the Docker process has stopped.**

**Example**

**sudo service docker stop**

**Docker terminology**

**Container image**: A package with all the dependencies and information needed to create a container. An image includes all the dependencies (such as frameworks) plus deployment and execution configuration to be used by a container runtime. Usually, an image derives from multiple base images that are layers stacked on top of each other to form the container's filesystem. An image is immutable once it has been created.

**Dockerfile**: A text file that contains instructions for how to build a Docker image. It's like a batch script, the first line states the base image to begin with and then follow the instructions to install required programs, copy files and so on, until you get the working environment you need.

**Build**: The action of building a container image based on the information and context provided by its Dockerfile, plus additional files in the folder where the image is built. You can build images with the Docker **docker build**command.

**Container**: An instance of a Docker image. A container represents the execution of a single application, process, or service. It consists of the contents of a Docker image, an execution environment, and a standard set of instructions. When scaling a service, you create multiple instances of a container from the same image. Or a batch job can create multiple containers from the same image, passing different parameters to each instance.

**Volumes**: Offer a writable filesystem that the container can use. Since images are read-only but most programs need to write to the filesystem, volumes add a writable layer, on top of the container image, so the programs have access to a writable filesystem. The program doesn't know it is accessing a layered filesystem, it is just the filesystem as usual. Volumes live in the host system and are managed by Docker.

**Tag**: A mark or label you can apply to images so that different images or versions of the same image (depending on the version number or the target environment) can be identified.

**Multi-stage Build**: Is a feature, since Docker 17.05 or higher, that helps to reduce the size of the final images. In a few sentences, with multi-stage build you can use, for example, a large base image, containing the SDK, for compiling and publishing the application and then using the publishing folder with a small runtime-only base image, to produce a much smaller final image

**Repository (repo)**: A collection of related Docker images, labeled with a tag that indicates the image version. Some repos contain multiple variants of a specific image, such as an image containing SDKs (heavier), an image containing only runtimes (lighter), etc. Those variants can be marked with tags. A single repo can contain platform variants, such as a Linux image and a Windows image.

**Registry**: A service that provides access to repositories. The default registry for most public images is Docker Hub(owned by Docker as an organization). A registry usually contains repositories from multiple teams. Companies often have private registries to store and manage images they've created. Azure Container Registry is another example.

**Multi-arch image**: For multi-architecture, is a feature that simplifies the selection of the appropriate image, according to the platform where Docker is running, e.g. when a Dockerfile requests a base image **FROM mcr.microsoft.com/dotnet/core/sdk:2.2** from the registry it actually gets **2.2-sdk-nanoserver-1709**, **2.2-sdk-nanoserver-1803**, **2.2-sdk-nanoserver-1809** or **2.2-sdk-stretch**, depending on the operating system and version where Docker is running.

**Docker Hub**: A public registry to upload images and work with them. Docker Hub provides Docker image hosting, public or private registries, build triggers and web hooks, and integration with GitHub and Bitbucket.

**Azure Container Registry**: A public resource for working with Docker images and its components in Azure. This provides a registry that is close to your deployments in Azure and that gives you control over access, making it possible to use your Azure Active Directory groups and permissions.

**Docker Trusted Registry (DTR)**: A Docker registry service (from Docker) that can be installed on-premises so it lives within the organization's datacenter and network. It is convenient for private images that should be managed within the enterprise. Docker Trusted Registry is included as part of the Docker Datacenter product

**Docker Community Edition (CE)**: Development tools for Windows and macOS for building, running, and testing containers locally. Docker CE for Windows provides development environments for both Linux and Windows Containers.

**Docker Enterprise Edition (EE)**: An enterprise-scale version of Docker tools for Linux and Windows development.

**Compose**: A command-line tool and YAML file format with metadata for defining and running multi-container applications. You define a single application based on multiple images with one or more .yml files that can override values depending on the environment. After you have created the definitions, you can deploy the whole multi-container application with a single command (docker-compose up) that creates a container per image on the Docker host.

**Cluster**: A collection of Docker hosts exposed as if it were a single virtual Docker host, so that the application can scale to multiple instances of the services spread across multiple hosts within the cluster. Docker clusters can be created with Kubernetes, Azure Service Fabric, Docker Swarm and Mesosphere DC/OS.

**Orchestrator**: A tool that simplifies management of clusters and Docker hosts. Orchestrators enable you to manage their images, containers, and hosts through a command line interface (CLI) or a graphical UI. You can manage container networking, configurations, load balancing, service discovery, high availability, Docker host configuration, and more. An orchestrator is responsible for running, distributing, scaling, and healing workloads across a collection of nodes. Typically, orchestrator products are the same products that provide cluster infrastructure, like Kubernetes and Azure Service Fabric, among other offerings in the market.

### Docker Daemon

Docker daemon runs on host system. The users cannot interact directly with Docker daemon but only through Docker clients.

### Docker Client

Docker Client is the chief user interfacing for Docker and it is in docker binary format. Docker daemon will accept the docker commands from users and establishes to and fro communication with Docker daemon.

### Docker Swarm

Docker Swarm is domestic cluster for Docker. This will allow creation and accessing to a collection of Docker hosts with the help of Docker tools. As Docker Swarm acts as worthful API for Docker, any of Docker tools which are communicating with Docker daemon could use Swarm for transparently scaling different hosts.

**Docker Swarm**

With **Docker Swarm** Mode, a service is a long-running**Docker** container that can be deployed to any node worker. It's something that either remote systems or other containers within the **swarm** can connect to and consume. For this **example**, we're going to deploy a Redis service.

* To create a Docker container, download the ‘hello world’ image, by typing the following command in the terminal –

$ docker run hello world

* For checking the number of images on your system, use the following command –

$ docker images

* For searching an image in the Docker Hub –

$ docker search <image>

docker info

sudo docker pull Jenkins

sudo docker run -p 8080:8080 -p 50000:50000 jenkins

docker rmi ImageID

sudo docker inspect jenkins

### Attach to and detach from a running container

$ docker run -d --name topdemo ubuntu /usr/bin/top -b

$ docker attach topdemo

* docker run – Runs a command in a new container.
* docker start – Starts one or more stopped containers
* docker stop – Stops one or more running containers
* docker build – Builds an image form a Docker file
* docker pull – Pulls an image or a repository from a registry
* docker push – Pushes an image or a repository to a registry
* docker export – Exports a container’s filesystem as a tar archive
* docker exec – Runs a command in a run-time container
* docker search – Searches the Docker Hub for images
* docker attach – Attaches to a running container
* docker commit – Creates a new image from a container’s changes

List:

$ docker images –a

$ docker images -a | grep "pattern"

Remove Image :

$docker rmi Image Image

## Removing Containers

$docker ps –a

$ docker container ls -a

Remove

$docker rm ID\_or\_Name ID\_or\_Name

$ docker container rm cc3f2ff51cab cd20b396a061

Remove all stoped container

$docker container ls -a --filter status=exited --filter status=created

$ docker container prune

Docker container stop

$docker container stop

## Docker stats

You can use the docker stats command to live stream a container’s runtime metrics. The command supports CPU, memory usage, memory limit, and network IO metrics.

The following is a sample output from the docker stats command

$ docker stats redis1 redis2

docker top

With this command, you can see the top processes within a container.

Syntax

docker top ContainerID

docker stop

This command is used to stop a running container.

Syntax

docker stop ContainerID

docker stats

This command is used to provide the statistics of a running container.

Syntax

docker stats ContainerID

docker pause

This command is used to pause the processes in a running container.

Syntax

docker pause ContainerID

docker unpause

This command is used to unpause the processes in a running container.

Syntax

docker unpause ContainerID

docker kill

This command is used to kill the processes in a running container.

Syntax

docker kill ContainerID

nginx setup

Dockerfile

#This is a sample Image

FROM ubuntu

MAINTAINER demousr@gmail.com

RUN apt-get update

RUN apt-get install nginx

CMD [“echo”,”Image created”]

Example

sudo docker build –t myimage:0.1.

Mongodb databse

MongoDB is a famous document-oriented database that is used by many modern-day web applications. Since MongoDB is a popular database for development, Docker has also ensured it has support for MongoDB.

sudo docker run -it -d mongo

* The **–it** option is used to run the container in interactive mode.
* The **–d** option is used to run the container as a daemon process.
* And finally we are creating a container from the Mongo image.

Now let’s spin up another container which will act as our client which will be used to connect to the MongoDB database. Let’s issue the following command for this −

sudo docker run –it –link=tender\_poitras:mongo mongo /bin/bash

connect to the MongoDB server from the client container. We can do this via the following command −

mongo 172.17.0.2:27017

The following points need to be noted about the above command

The mongo command is the client mongo command that is used to connect to a MongoDB database.

The IP and port number is what you get when you use the env command.

use demo

nginx

sudo docker pull nginx

sudo docker run –p 80:80 –d nginx/

http://dockerhost:80 This shows that the nginx container is up and running.

create an HTML file called **HelloWorld.html**

sudo docker run –p 8080:80 –v

“$PWD”:/usr/share/nginx/html:ro –d nginx

**http://dockerhost:8080/HelloWorld.html**

-----------

Mysql

257 mkdir docker-db

258 sudo chmod 777 docker-db/

259 cd docker-db/

260 sudo nano docker-compose.yml

261 sudo docker-compose up

262 sudo docker ps

263 sudo nano docker-compose.yml

264 sudo docker-compose up -d

265 sudo ps

266 sudo docker ps

267 mysql --host 0.0.0.0:3306 -u root -p

268 mysql -host 0.0.0.0:3306 -u root -p

269 mysql -host=0.0.0.0:3306 -u root -p

270 mysql -u root -p

271 mysql 3.14.66.182:3306-u root -p

272 mysql 3.14.66.182:3306 -u root -p

273 sudo docker mysql 3.14.66.182:3306 -u root -p

274 sudo mysql 3.14.66.182 -u root -p

275 mysql 3.14.66.182 -u root -p

276 sudo apt install mysql-client-core-57

sudo apt install mysql-client-core-5.7

mysql --host 52.14.212.58 -u root -p

history

sudo docker container stop contaiserid

Wordpress – compose

### Define the project

1. Create an empty project directory.

You can name the directory something easy for you to remember. This directory is the context for your application image. The directory should only contain resources to build that image.

This project directory contains a docker-compose.yml file which is complete in itself for a good starter wordpress project.

**Tip**: You can use either a .yml or .yaml extension for this file. They both work.

1. Change into your project directory.

For example, if you named your directory my\_wordpress:

cd my\_wordpress/

1. Create a docker-compose.yml file that starts your WordPress blog and a separate MySQL instance with a volume mount for data persistence:

version: '3.3'

services:

db:

image: mysql:5.7

volumes:

- db\_data:/var/lib/mysql

restart: always

environment:

MYSQL\_ROOT\_PASSWORD: somewordpress

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

WORDPRESS\_DB\_NAME: wordpress

volumes:

db\_data: {}

### Build the project

Now, run docker-compose up -d from your project directory.

This runs docker-compose up in detached mode, pulls the needed Docker images, and starts the wordpress and database containers, as shown in the example below.

$ docker-compose up -d

Creating network "my\_wordpress\_default" with the default driver

Pulling db (mysql:5.7)...

5.7: Pulling from library/mysql

efd26ecc9548: Pull complete

a3ed95caeb02: Pull complete

...

Digest: sha256:34a0aca88e85f2efa5edff1cea77cf5d3147ad93545dbec99cfe705b03c520de

Status: Downloaded newer image for mysql:5.7

Pulling wordpress (wordpress:latest)...

latest: Pulling from library/wordpress

efd26ecc9548: Already exists

a3ed95caeb02: Pull complete

589a9d9a7c64: Pull complete

...

Digest: sha256:ed28506ae44d5def89075fd5c01456610cd6c64006addfe5210b8c675881aff6

Status: Downloaded newer image for wordpress:latest

Creating my\_wordpress\_db\_1

Creating my\_wordpress\_wordpress\_1