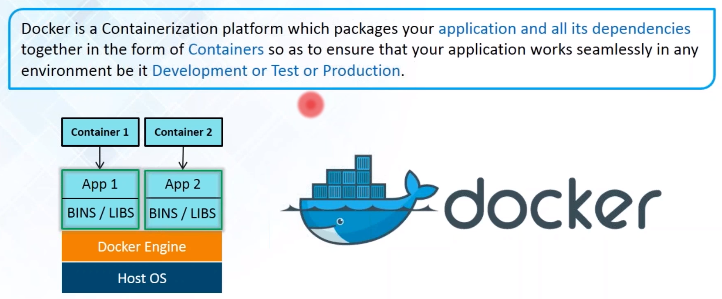
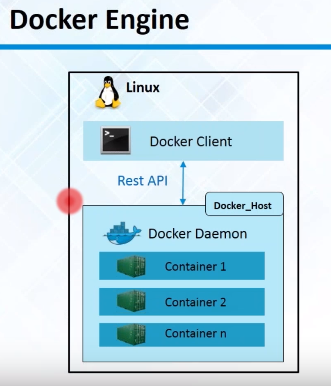
Docker tutorial

Virtualization has hypervisor and guest OS on the top of host OS / short commings - > unstable performace, hypervisor not efficient as host os, long bootup process

Containerization – apstraction to the software not hardwere as VM. It is virtualization on OS level. Has container Engine on top of host OS, short bootup, faster, ligher and smaller and also brings process (application) level isolation as VM.





Docker engine = client + daemon.

Rest API with Socket.,IO and TCP between.

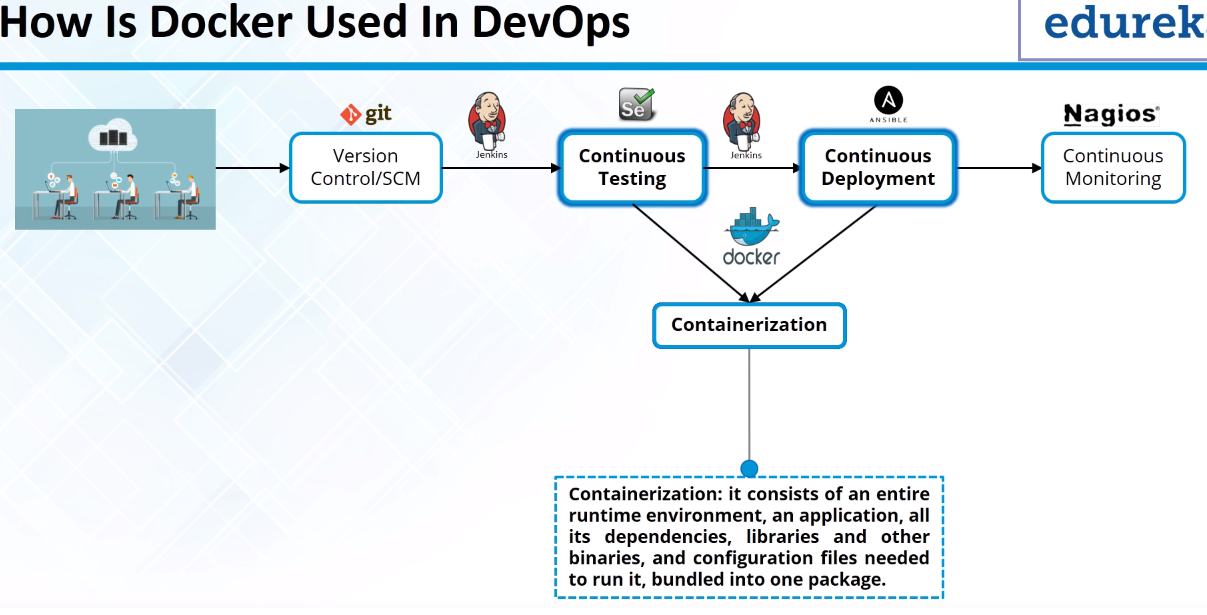
Docker client

Docker images – executable files, read only template used to create containers. Built by Docker Users, stored in Docker Hub or on mine Local Registry. Created with build command

Docker containers – With run command on docker image or images they are created. Isolated application platform, contains everything needed to run the application.

Docker registry – local or remote storage for placing docker images

Docker hub – dockers own cloud repo.

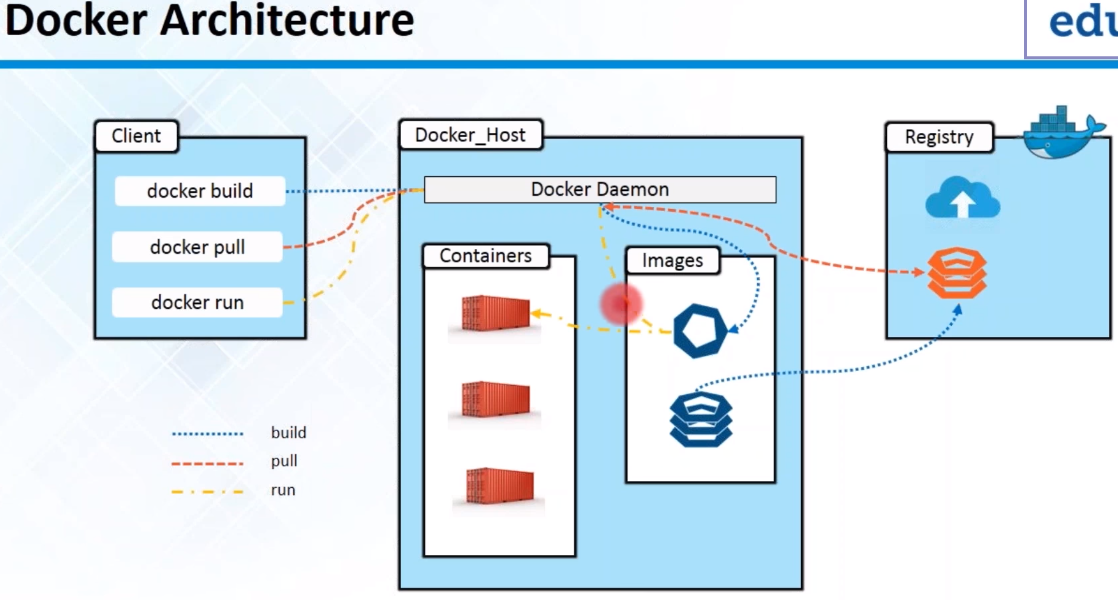


For creating the Docker Container:

We can build own Image and use it for creating Docker container

We can pull Image from Docker Hub and run the Docker Image

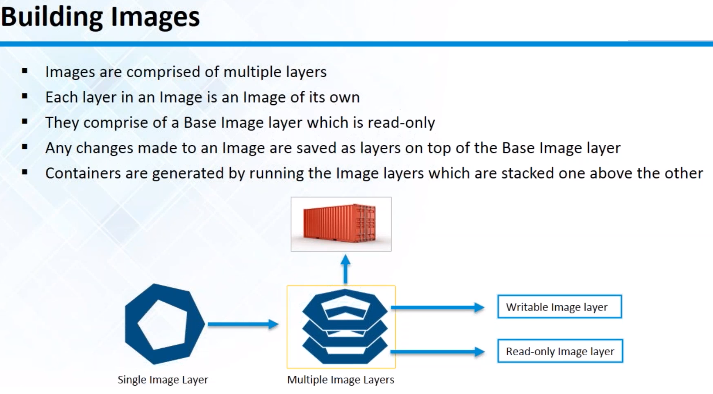
Docker Daeomn store the Docker Images in Docker Registry either local repo or remote Docker Hub.



Tag = version of the image.

|  |  |
| --- | --- |
| **Command** | **Description** |
| **docker –version |**  **docker version |**  **docker info** | Less | more details about docker installation and containers and images |
| **docker images |**  **docker image ls** | List down all the images.  To list all options with images run @docker image  [*build, history, import, inspect, load, ls, prune, pull, push, rm, save, tag*] |
| **docker pull <image-name: tag> or**  **docker run <image-id>** | To run the Image |
| **docker pull <image-name: tag>** | Pull Docker Image from Docker Hub |
| **docker ps |**  **docker container ls --all** | List down all the running containers/  To list all options with container run @docker container  [*attach, commit, cp, create, diff, exec, export, inspect, kill, logs, ls , pause, port, prune, rename, restart, rm, run, stsart, stats,* *stop, top, unpause, update, wait*] |
| **docker ps –a** | List down all containers |
| **docker rm $(docker container ls -aq)** | Remove all containers |
| **docker build –t [image\_name]:tag.** | Build image |
| **docker run - - name “container name” –p <host port>: <container port> <image\_name:tag>** | Run newly built Image |
| **docker run –d –p 4000:80 mypthonapp** | Run app in background in detached mode |
| **docker run –p 4000:80 <username/repository:tag>** | Run app on any machine from registry |
| **docker start <container\_id>** | Start the container |
| **docker stop <container\_id>** | Stop the container |
| **docker rm <container\_id> | <container\_name>** | Delete the container (after stopping) |
| **docker rmi <image\_id> |**  **docker rmi <repo:tag>** | Delete an image |
| **docker login** | Login to your docker registry |
| **docker tag <image\_name> <username/repository:tag>** | Tag the image |
| **docker push <username/repository:tag>** | Publish the image – upload to the repository |

# Buidling Images



Docker file. - To build the images (instructions for building FROM, RUN), and install application.

FROM – Keyword indicates the base image from which the container is built

RUN - > Keyword indicates the command that needs to be executed on the Image.

Build an Image: **docker build –t anel\_image:1.0.**

Run newly built Image: **docker run - - name “anel\_container” –p 8080:8080 anel\_image:1.0.**

# Installing the docker using the repository

**Install Docker community edition (CE)**

sudo apt-get update

**Install packages to allow apt to use repos over HTTPS:**

sudo apt-get install \

apt-transport-https \

ca-certificates \

curl \

software-properties-common

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

**Add Dockers official GPG key:**

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add –

OK

Verify that you now have the key with the fingerprint 9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88, by searching for the last 8 characters of the fingerprint.

$ sudo apt-key fingerprint 0EBFCD88

pub 4096R/0EBFCD88 2017-02-22

Key fingerprint = 9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88

uid Docker Release (CE deb) <docker@docker.com>

sub 4096R/F273FCD8 2017-02-22

lsb\_release –cs => bionic

**Setup stable repository:**

sudo add-apt-repository \

"deb [arch=amd64] https://download.docker.com/linux/ubuntu \

$(lsb\_release -cs) \

stable"

**Install latest version of Docker CE**

$ sudo apt-get update

$ sudo apt-get install docker-ce

$ sudo docker run hello-world

This command downloads a test image and runs it in a container. When the container runs, it prints an informational message and exits.

To run the Image from the same folder where we have Dockerfile:

$ docker build –t image\_name:tag .

To run the image and to map container port to host port:

$ docker run –p 8080:8080 –-name “containername” image\_name:tag

To see the effect:

$ docker ps

# Manage Docker as a non-root user

Docker daemon binds to Unix socket instead of TCP port, which is owned by the root user and is accessible only with sudo.

Create a Unix group called docker and add users to it. When Docker daemon starts, it creates Unix socket accessible by members of the docker group.

**To See which groups your user account belongs to**

$ groups

**List all members of the group using /etc/group file**

$ grep ‘docker’ /etc/group

**See all groups you have (if there is docker group skip following step)**

$ cut -d: -f1 /etc/group | sort

**Create docker group**

$ sudo groupadd docker

**Add user to docker group**

$ sudo usermod -aG docker $USER

Log out and log in to see the effect.

# Building an app the Docker way

At the bottom of the hierarchy is a container.

Above this level is a service, which defines how containers behave in production

At the top level is the stack – defining the interaction of all services

Dockerfile – file where portable images are defined., what goes on in the environment inside your container.

$ curl http://localhost:4000

Create docker account : <https://hub.docker.com/> in order to create a docker registry which is a collection of docker repositories and repository is a collection of docker images.

You can run docker login to log to hub.docker.com. Password is saved locally in ~/.docker/config.json as unencrypted.

Tag the image:

**docker tag <image\_name> <username/repository:tag>**

**docker tag mypthon an3l/get-started:part1**

anel@ubuntu:~/workspace/demo\_docker$ **docker image ls**

*REPOSITORY TAG IMAGE ID CREATED SIZE*

*an3l/get-started part1 1d2469d00ac2 24 minutes ago 132MB*

*mypython latest 1d2469d00ac2 24 minutes ago 132MB*

*hello-world latest 4ab4c602aa5e 11 days ago 1.84kB*

*python 2.7-slim c9cde4658340 2 weeks ago 120MB*

Upload image to the repo:

**docker push <username/repository:tag>**

**docker push <ann3l/get-started:part1>**

To run app from image from repo from any machine (if not locally and then it will be pulled):

**docker run –p 4000:80 <username/repository:tag>**

|  |
| --- |
| **Recap and cheat sheet**  docker build -t friendlyhello . # Create image using this directory's Dockerfile  docker run -p 4000:80 friendlyhello # Run "friendlyname" mapping port 4000 to 80  docker run -d -p 4000:80 friendlyhello # Same thing, but in detached mode  docker container ls # List all running containers  docker container ls -a # List all containers, even those not running  docker container stop <hash> # Gracefully stop the specified container  docker container kill <hash> # Force shutdown of the specified container  docker container rm <hash> # Remove specified container from this machine  docker container rm $(docker container ls -a -q) # Remove all containers  docker image ls -a # List all images on this machine  docker image rm <image id> # Remove specified image from this machine  docker image rm $(docker image ls -a -q) # Remove all images from this machine  docker login # Log in this CLI session using your Docker credentials  docker tag <image> username/repository:tag # Tag <image> for upload to registry  docker push username/repository:tag # Upload tagged image to registry  docker run username/repository:tag # Run image from a registry |

# Scaling application , enable load balancing

Before start get Docker Compose.

* Stack
* **Services** (you are here)
* Container

In a distributed application, different pieces of the app are called “services.” For example, if you imagine a video sharing site, it probably includes a service for storing application data in a database, a service for video transcoding in the background after a user uploads something, a service for the front-end, and so on.

Services are really just “containers in production.” A service only runs one image, but it codifies the way that image runs—what ports it should use, how many replicas of the container should run so the service has the capacity it needs, and so on. Scaling a service changes the number of container instances running that piece of software, assigning more computing resources to the service in the process.

Luckily it’s very easy to define, run, and scale services with the Docker platform -- just write a docker-compose.yml file. A docker-compose.yml file is a YAML file that defines how Docker containers should behave in production.

|  |
| --- |
| version: "3"  services:  web:  # replace username/repo:tag with your name and image details  image: username/repo:tag  deploy:  replicas: 5  resources:  limits:  cpus: "0.1"  memory: 50M  restart\_policy:  condition: on-failure  ports:  - "4000:80"  networks:  - webnet  networks:  webnet: |

This docker-compose.yml file tells Docker to do the following:

* Pull [the image](https://docs.docker.com/get-started/part2/)  from the registry.
* Run 5 instances of that image as a service called web, limiting each one to use, at most, 10% of the CPU (across all cores), and 50MB of RAM.
* Immediately restart containers if one fails.
* Map port 4000 on the host to web’s port 80.
* Instruct web’s containers to share port 80 via a load-balanced network called webnet. (Internally, the containers themselves publish to web’s port 80 at an ephemeral port.)
* Define the webnet network with the default settings (which is a load-balanced overlay network).

## Get Docker Compose

<https://github.com/docker/compose>

Compose is a tool for defining and running multi-container Docker apps. With Compose, you can use a Compose file to configure your applications services.

Installing <https://github.com/docker/docker.github.io/blob/master/compose/install.md>

<https://docs.docker.com/compose/install/#install-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

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

Test installation:

docker-compose --version

**To run the composer :**

**$ docker-compose up -d**

Use –d for background process (detached mode)

Using bind mounts for docker-composer

[Volumes](https://docs.docker.com/storage/volumes/) and [bind mounts](https://docs.docker.com/storage/bind-mounts/) let you share files between the host machine and container so that you can persist data even after the container is stopped.

Persisted data, temporary file system (tmpfs, only for linux),  swarm services (mount flag)

<https://docs.docker.com/storage/bind-mounts/>

<https://docs.docker.com/storage/volumes/>

<https://docs.docker.com/storage/tmpfs/>

<https://docs.docker.com/storage/storagedriver/>

We are using volumes service in order to have modification from host to docker instantly.

Other commands:

$ docker-compose run

$ docker-compose --help

## Install command completion

<https://docs.docker.com/compose/completion/#zsh>

Completition script is in folder /etc/bash\_completition.d or /usr/local/etc/bash\_completition.d

Place completition script :

sudo curl -L https://raw.githubusercontent.com/docker/compose/1.22.0/contrib/completion/bash/docker-compose -o /etc/bash\_completion.d/docker-compose

# Compose and django

<https://docs.docker.com/compose/django/>

# Dockerfile reference

<https://docs.docker.com/engine/reference/builder/>

|  |
| --- |
| AdditionalExample: Tomcat Ubuntu Image for JavaChanging Linux Colors $ dircolors --print-database > DIR\_COLORS  $  force\_color\_prompt - > .bashrc  Just change colors in color\_prompt  01;32 – zelena, 01; 31 crvena, 01;36 cyan , 35 purple, 33 browm; 1;33 yellow, 0;37 gray; 1;37 white; 0;30 black; 1:30 darkgrey; 0;34 blue  01 - > bold , 0 nije bold |