**Problems Before Docker**

* An application works in developer's laptop but not in testing or production . This is due to difference in computing environment between Dev, Test and Prod.

What are microservices?

The idea behind microservices is that some types of applications become easier to build and maintain when they are broken down into smaller , composable pieces which work together . Each component is developed separately, and the application is then simply the sum

of its constituent components.

For example imagine an online shop with separate microservices for user - accounts , product - catalog order processing and shopping carts.

3 major reasons for adopting microservice architecture or 3 advantages of using microservice architecture:

* There are certain applications which are easier to build and maintain when they are broken down into smaller pieces or smaller services.
* If we want to update a particular software or if we want a new technology stack in one of our module or service, so we can easily do that because dependency concerns are very less as compare to whole application.
* If any of our service or module goes down then the whole application will remain largely unaffected.

Problems in adopting microservice architecture:

One way of implementing microservice architecture is to launch various virtual machines (VM) over a host machine to launch a single microservice on each VM.

The disadvantage over here is in virtual machines there are lot of wastage of resources (such as RAM, processor, disk space, etc.), they are not utilized completely by the microservice which is running in these VM.

**How Docker solves these problems (implementing of microservice problem)**

1. You can run several microservices in the same VM by running various Docker containers for each microservice .

There is a host machine on top of that there is a VM, and on top of that there are multiple docker containers and each of these docker containers contain dependency for one microservice.

So the difference here from VM (earlier we are using VM, and now we are using docker containers on top of VM) is, docker containers are actually lightweight alternative of VM, in docker container we don't need to pre-locate any RAM or any disk space, it takes RAM and disk space according to the requirement of application.

1. How docker solves the problem of not having a consistent computing environment throughout the whole SDLC (Software development lifecycle -> Project plan, Requirements, analysis, Design, coding, testing, deployment)

Docker containers are developed by developers itself.

So let’s see how docker solve the first problem that an application works fine in dev environment but not in testing and prod environment.

Docker containers can be used throughout the SDLC life cycle in order to provide a consistent computing environment. So the same environment will be present in dev, test, and prod. So there won’t be any difference in computing environment.

**What id Docker**

Docker does not uses the guest operating system, it uses the host operating system.

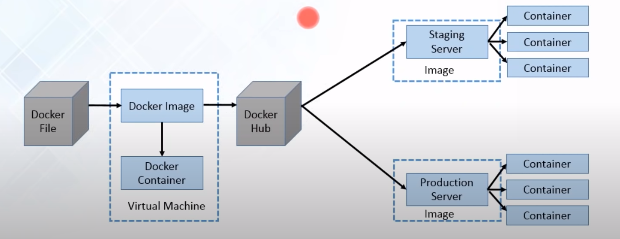
There will be host OS, and on top of that there will be a docker engine, and with the help of docker engine docker containers are formed, which run on top of the docker engine. These containers have application running in them, and the requirements for these applications such as all the binaries and libraries are also packaged with the same container. And there can be multiple containers running on the docker engine.

* Docker is a tool designed to make it easier to create, deploy and run applications by using containers.
* Docker containers are light weight alternatives to VM and it uses the host OS.
* You dont have to pre-allocate any RAM in containers.

**Docker in a nutshell or general workflow of docker or one way of using docker**

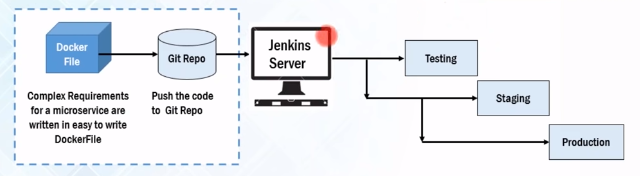
In this, a developer writes a code that defines its application, requirements and the dependencies in an easy to write dockerfile, and this dockerfile creates docker images, so whatever dependencies required for an application is present in that docker image, and docker containers are basically the run time instances of docker image. We can run this image to create as many docker containers as we want. This particular image is uploaded to docker hub, docker hub is basically a git repository for docker images, it contains public as well as private repositories. So from public repository you can pull your image and you can upload you own images to the dockerhub. So from dockerhub, various team such as qa and prod pull the images and prepare their own containers.

So the major advantage from the workflow is that whatever the dependencies required for your application is actually present throughout the SDLC



**Docker Example or another way of using docker**

Docker images are huge in size and requires a lot of network bandwidth. So in order to save network bandwidth, we can use this kind of workflow.



Over here, we use jenkins server or any continuous integration (CI) server to build an environment that contains all the dependencies for a particular application or a microservice and that build environment is deployed into various teams like testing, staging and production.

In this, developer has written complex requirements for a microservice in an easy to write dockerfile and then the code is push onto the git repository. From git repository continuous integration server like jenkins pulled that code and builds an environment that contains all the dependencies for that particular microservice. And that environment is deployed onto testing, staging and production. So in this way whatever requirements are there for your microservice is present throughout the SDLC.

**Docker Registry**

* Docker Registry is nothing but the storage for all your docker images.
* You can store docker images either in public repository or in private repository.
* These repositories can be present locally or it can be present on the cloud. Docker provides a cloud hosted service called dockerhub.
* Dockerhub has public as well as private repositories.
* From public repository you can actually pull an image, and prepare your own containers. At the same same you can write your image and upload it to your private repository or in public repository over dockerhub.

**Docker Images**

* Read Only Template Used To Create Containers.
* These docker images contains all the dependencies for a particular application or microservice.
* We can create our own image and upload it onto the dockerhub. At the same time we can pull the images from the public repository from the dockerhub.
* Built By Docker Users
* Stored In Docker Hub Or Your Local Registry

**Docker Containers**

* Isolated Application Platform
* Docker containers are nothing but the runtime instances of docker images
* Contains Everything Needed To Run The Application or microservice
* It is also possible that more than one image is required to create one container. (Built From One Or More Images)

**Install docker on Ubuntu**

sudo get-apt update -> to update the packages

sudo apt-get install linux-image-extra-virtual -> to download recommended package to install docker (pre-requisities to install docker)

sudo apt-get install docker-engine -> to install docker

sudo service docker start -> to start the docker services

o/p: Job is already running

(Not working)

Official docker docs installation method

Update the apt package index and install packages to allow apt to use a repository over HTTPS:

sudo apt-get update

sudo apt-get install \

ca-certificates \

curl \

gnupg \

Lsb-release

Add Docker’s official GPG key:

sudo mkdir -p /etc/apt/keyrings

$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

Use the following command to set up the repository:

echo \

"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \

$(lsb\_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

Install DockerEngine

Update the apt package index, and install the *latest version* of Docker Engine, containerd, and Docker Compose, or go to the next step to install a specific version:  
 $ sudo apt-get update

$ sudo apt-get install docker-ce docker-ce-cli containerd.io docker-compose-plugin

Pull a centos image from dockerhub and run the centos container

sudo docker pull centos

(first it check for local repository, if it doesn’t found here then it goes to dockerhub and download the image from there)

sudo docker run run -it centos -> run centos container

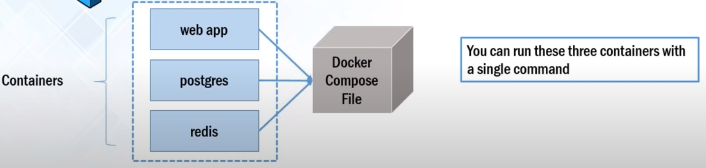
So by this we were in the centos container, we can use exit to quit.

**What is docker compose**

Suppose we have multiple applications on multiple containers and all those containers are actually linked together. So you dont want to actually execute each of those containers one by one but you want to run those containers at once with a single command. So here docker compose comes into the picture.

Docker Compose makes it easier to configure and run applications made up of multiple

containers . For example : imagine being able to define three containers - one running a web app, another running postgres , and a third running redis - all in one YAML file and then running those three connected containers with a single command .



**Install and setup Docker**

**At RedHat**

sudo yum update -> to update existing packages

curl -fsSl <https://get.docker.com/> | sh -> run docker installation script

(this script adds the docker.repo repository and installs docker)

sudo service docker start -> to start docker services

o/p: starting docker: ok

Pull a docker image for ubuntu operating system

sudo docker pull ubuntu -> pull an image for ubuntu

(first it check for local repository, if it doesn’t found here then it goes to dockerhub and download the image from there)

sudo docker run -it ubuntu-> to start using container

sudo docker ps -> to see all running containers

**At Windows**

Why use docker on Windows

The first reason is that it avoids “works on my machine but doesn’t work on prod” problem.

This problem occurs due to the inconsistent environment throughout the software development workflow.

For eg, let’s say that a developer has built an application on windows environment and when he sends the application to the testng server , it fails to run. Now this happens because the testing server operates on an outdated version of windows. Now obviously the application does not support the dependencies needed to run on the outdated version of windows. SO because of the difference in the software versions in the development and testing server, the application will fail. But when it comes to docker we can run our application within a container which contains all teh dependencies of the application and the container can be run throughout the software development lifecycle.

Now this practice provides a consistent environment throughout. Apart from that it improves productivity. So b installing docker on windows we are running docker natively.

So before installing docker for windows there are pre-requisites:

* 64 bit windows 10 (for older version we can use docker toolbox instead)
* Enable hyper-v (Docker for windows requires type 1 hypervisor and in windows it is called hyper-v. Hyper-v s basically a lightweight virtualization solution built on top of the hypervisor framework so you dont need to virtualbox, you just need to enable hypervisor)
* Enable virtualization

Note: If you are using virtualbox on your system, you wont be able to run it because virtualbix will not work with the hypervisor enabled, but in order for your docker for windows to work on your system the hyoervisor must be enabled. So basically you cannot run docker for windows and virtualbox on the same system side by side

We need a docker installer to download (docker for windows).

Run the installer after download .

Open the docker for windows app.

So when we try to start the application we can see the whale icon at the status bar (right side toolbar). When the whale icon get stable it means docker started and we can start working on it. After that a prompt comes and we can login from there into our dockerhub account.After login, we can use any terminal to run the docker commands.

Open windows powershell as an administrator

Docker --version -> to check docker installed or not

Docker run hello-world -> to run the hello world image

Docker images -> to check the images have in our system

Docker pull ubuntu -> to pull image from dockerhub

Docker run -it -d ubuntu-> to run this ubuntu image

Docker ps -a -> it shows all the containers

Docker exec -it <container\_id> bash -> to access a running container (ubuntu container id)

We can perform all ubuntu commands in it. Eg, echo hello, and we can use exit to come put from the container.

Docker stop <container\_id> -> to stop a running container

Docker commit abhayrajsr/ubuntu -> it will crate a new image on the local system (abhayrajsr -> dockerhub repository name and ubuntu is the name of the image)

Docker images -> now we have seen a new ubuntu image also with old images

Docker login -> to login to dockerhub account

Docker push abhayrajsr/ubuntu -> to push this image to dockerhub

Docker rm <container\_id>-> to delete a container

Docker rmi <image\_id> -> to remove/delete docker image

Now, we are creating simple python web application using docker compose.

About this application, it uses flask framework and it maintains a hit count on redis. Flask is basically a web development framework which is written in python and redis is an in-memory storage component. It basically use as a database.

So we are basically going to use a docker compose to run two services which is web service and redis service.

What this application does is it’s going to maintain a hit counter every time you access a web page. So each time you access the website a hit counter gets incremented.

Logic is just increment the value of the hit counter when the webpage is accessed.

Mkdir webapplication -> creating a directory for your application

Cd webapplication -> to move into that directory

Start notepad++ -> to open the notepad++

We dont need to install python and redis, we are going to use docker images for python and redis. So first we need to create a python file named webapp.py

We are going to begin with importing dependencies, so we are going to import real time and we need to import redis and flask.

import time

import redis

from flask import Flask

After that we just initializing name of the application

app = Flask(app)

After this, we are just hosting the database and we are connecting to redis using the port number 6379, this is the default port.

cache = redis.Redis(host=’redis’, port=6379)

After that, we define the get hit count function, this basically return the number of hits. So we are also setting the retries to 5 incase the page does not load while all of this holds true the incremented hits are returned and if there’s an error then we have an exception so we have also defined exception in case of errors this function is basically to display the helo world message along with the hit comes.

def get\_hit\_count ( ) :

retries = 5

while True :

try :

return cache.incr (‘hits‘)

except redis.exceptions.ConnectionError as exc :

if retries == 0 :

raise exc

retries -= 1

time.sleep ( 0.5 )

@app.route ( ‘ / ‘ )

def hello ( ) :

count = get\_hit\_count ( )

return ' Hello World ! I have been seen { } times.\n' .format(count)

if app == “\_\_main\_\_":

app.run(host=”0.0.0.0”,debug=True)

Next file we gonna create is requirements.txt file

flask

redis

Next we have our Dockerfile. Now this dockerfile is used to create docker images.

FROM python:3.4-alpine ->just setting the base image,building an image starting with python 3.4

ADD . /code -> adding the current dir to the /code path of the image

WORKDIR /code -> now we are changing the working dir to this path

RUN pip install -r requirements.txt -> in this we are going to use packet manager of python to install the requirements that are mentioned in the requirements.txt file

CMD [“python”, “webapp.py”] -> finally we are setting the default command for the containers to python web app. Its basically going to run my webapp

Now we have a docker compose file docker-compose.yml. A docker compose or yml file is going to contain all of the services. So there is web service over here and there is redis service.

version: ‘3’'

services:

web:

Build: .

ports:

* "5000:5000"

redis:

image: "redis:alpine"

So we are basically running two containers over here or two services over here which is web and redis. So now the web service is basically building the dockerfile in the current directory (dot “.” signifies current directory). And it forward the exposed port 5000 on the container to the port 5000 on the host machine. Now the redis service is basically using a redis image pulled from dockerhub.

Now we are going to run bith of these services or bith of these containers by using the docker compose command. Make sure all of these 4 files and obviously have to create them in webappliaction directory.

In powershell,

Ls -> al 4 files must be visible here

Docker-compose up -> to run all of these containers

Now we are going to look at the output by using kitematic. Kitematic is basically a UI tool for docker for windows. So left click on the docker icon in the status bar and click kitematic.

After open kitematic we can see two applications there in left, one is app\_redis, and other is app\_web. When we go to app\_web, and click on the left side arow, it redirects to the output webpage. And when we refresh the output page the hitcount increases.

From kitematic, we can restart, stop or go into the container.

**Writing a DockerFile**

Dockerfile syntax consists of two kind of main line blocks : comments and commands + arguments

Syntax

# Line blocks used for commenting

command argument argument1 ...

Example

# Print " Welcome To Edureka ! "

RUN echo " Welcome To Edureka ! "

[where RUN is command and echo " Welcome To Edureka” is argument, two argument echo and welcome to edureka]

**DockerFile Commands - FROM**

FROM is the most important command, because without the FROM command we cannot write a dockerfile. The FROM command is used to specify a base docker image to use to start the build process.

Eg., # Usage : FROM [ image name ]

FROM ubuntu

So in this case I will be using ubuntu as my base docker image and all my customizations will be on top of my ubuntu image.

FROM ubuntu, it means that you have an ubuntu machine with you. SO this is just the base image which can be equivalent to you just having an ubuntu machine.

**DockerFile Commands - RUN**

If we want to run a particular image or if we want to run a particular command then we will use this RUN command.

In our case, if we have our ubuntu image and if I want to install say JAVA, or Jenkins or react or curl then I will be using RUN command.

The RUN command is the central executing directive for Dockerfiles . It takes a command as its argument

and runs it to form the image . Unlike CMD , it actually is used to build the image

Eg., # Usage : RUN [ command ]

RUN apt - get install -y riak

RUN command is basically for executing any command of mine.

But it has a slight difference when compared to CMD, because RUN is used to run a command, it could ne a shell command or it could be a command which basically runs my image into a container, and this is what the difference in respect to CMD.

**DockerFile Commands - CMD**

With CMD, we can execute a shell command, however we cannot use CMD command for building my docker image. So i cannot execute my docker image or i cannot build my docker image with the help of CMD command.

Eg., # Usage 1 : CMD application " argument " , " argument " , ..

CMD " echo " " Welcome To Edureka ! "

So if i want to execute a command on my shell, I can use either CMD or RUN, and if i want to build my docker image then i can only use RUN, for such things CMD won’t work.

**DockerFile Commands - ENTRYPOINT**

Entrypoint command is basically used to override whatever function your CMD command does or entrypoint basically suggests that when you have finished building your docker image then the command which is specified with the entrypoint that will be the onc which will be executed first then you run the docker container of that particular image.

Eg., # Usage : ENTRYPOINT application " argument " , " argument " , ..

# Remember : arguments are optional . They can be provided by CMD

# or during the creation of a container .

ENTRYPOINT echo

# Usage example with CMD :

# Arguments set with CMD can be overridden during \* run \* CMD " Hello docker ! "

CMD " Welcome to edureka ! "

ENTRYPOINT echo

Entrypoint can basically overrides your CMD commands.

**DockerFile Commands - ADD**

ADD command or COPY command, these commands can be used interchangeably because the ADD command is used to copy whatever files which are there in one particular directory to another directory. So, it could be copying files from my host to my container.

The ADD command gets two arguments : a source and a destination . It basically copies the files from the source on the host into the container's own filesystem at the set destination.

Eg., # Usage : ADD [ source directory or URL ] [ destination directory ]

ADD /my\_app\_folder /my\_app\_folder

In this, we use ADD then the path of the source then path of destination after space.

**DockerFile Commands - ENV**

If my application needs a particular environmental variable then I can specify to my docker container that this application needs certain environment variables and this environment variable is present here.

The ENV command is used to set the environment variables (one or more). These variable consists of “key value” pairs which can be accessed within the container by scripts and applications alike.

Eg., # Usage: ENV key value

ENV SERVER\_WORKS 4

Eg., if we want to execute a java program then we need java, and we have to set our environment variables so i can specify my java environment variables like this inside my docker container.So ENV would the command and next will be arguments (liek SERVER\_WORKS will be first argument and 4 will be second argument)

**DockerFile Commands - WORKDIR**

Inside our docker container, when we want to go into a particular container and start execution inside that container, especially when we want to execute certain commands in the shell. So if we want to use CMD command inside your dockerfile then you want to execute a particular command on the shell, but where exactly you want to execute that command because these commands will be executed from inside the container and inside the container if you want to customize the place where you want to execute that command, if you want to change the place by you want to execute the command, if you want to change the place where the CMD command will be executing its arguments then you have to set the working directory (WORKDIR) over here.

Eg., # Usage

WORKDIR /path WORKDIR ~/

**DockerFile Commands - EXPOSE**

Expose command is very important command in case of front end applications. With the expose command you can specify your port number and you can specify that this application would be active on this particular port number inside the container.

If we want to execute the same application on a particular port number on our host then we have to do port mapping.

Eg., Usage: EXPOSE [Port]

EXPOSE 8080

**DockerFile Commands - MAINTAINER**

If we want to tag our name along with the image, which we are building then we can use this MAINTAINER to specify who’s the person that’s maintaining this particular docker image before we approach to dockerhub. Whoever download that image from dockerhub will egt to know that’s teh guy who builds the image.

Eg., # Usage: MAINTAINER [name]

MAINTAINER authors’s\_name

ANd, this has to be present only after the FROM command.

**DockerFile COmmands - USER**

If we want a particular user to execute or to run a container then we can use this USER command, and specify the user id of that particular user whom you want to execute that docker container.

Eg., # Usage: USER [UID]

USER 715

**DockerFile Commands - VOLUME**

THis VOLUME command is basically used to set a custom path where our container will store all the files. SO this is the place where all the files related to our docker container will be present. And if we want two containers to share the same path then we can use this VOLUME command. So, this path can be shared by multiple containers.

Eg., Usage: VOLUME [“/dir\_1”,”/dir\_2”..]

VOLUME [“/my\_files”]

If we have multiple containers which are hosting the same application then we might want them all to use the same path.

**Creating an image to install Apache Web Server**

DockerFile for installing Apache

FROM Ubuntu: 12.04

MAINTAINER Edureka

RUN apt-get update && apt-get install -y apache && apt-get clean && rm -rf /var/lib/apt/lists/\*

ENV APACHE\_RUN\_USER www-data

ENV APACHE\_RUN\_GROUP www-data

ENV APACHE\_LOG\_DIR /var/log/apache2

EXPOSE 80

CMD [“/usr/sbin/apache2”, “-D”, “FOREGROUND”]

(In CMD, we are give command to go to this particular path and execute this in foreground mode. -D is the flag which we have to specify. Foreground means to get the UI up and to get it hosted)

Demo:

Open VM(ubuntu), and open the terminal and go to the particular path where you want to make the Dockerfile. And create a DockerFile with the above commands.

Now, we need to first build the docker image , and then run this particular docker image into a container.

Build docker image from dockerfile

Sudo docker build -t myapacheimage **.** -> [myapacheimage-docker image name ; (**.**) -> path of dockerfile]

Sudo docker images -> to verify the docker image has been built

To built up the docker container using built docker image

Sudo docker run -p 80:80 –name=App1 myapacheimage

In the above command we have to specify a port number because inside our dockerfile we have specified that the application has to be active on port number 80. And, if we want to access that application on my host machine, then we have to do port mapping, port mapping of my host port to container port by using -p 80:80, it means 80 port of my container will be mapped to 80 port of my host. After that we have give the container name as App1 by using –name tag and after that we have specified our docker image name from which the container build.

Check it in ubuntu firefox: localhost:80

So our servies are running fine, we can verify it too by using docker ps to see the running container.

We can stop the container by using the ctrl+C or using the stop command:

sudo docker stop <container\_id>

Use stop command on another terminal

After stopping the container we cannot access the webpage.

Docker ps -a -> to see the stopped containers

DockerFile for installing Nginx

FROM ubuntu:14.04

MAINTAINER Edureka

RUN apt-get update

RUN apt-get install -y nginx

ADD index.html /usr/share/nginx//html/index.html

ENTRYPOINT [“/usr/sbin/nginx”, “-g”, “daemon off;”]

EXPOSE 80

In this, we will be first going to use our ubuntu base image, then we are installing our nginx service on our ubuntu machine. And then specifying our maintainer as edureka and then using our run commands to update the packages, install the nginx and then we are adding index.html

In nginx index.html is not created by default, so we have created a index.html file and put our own code in index.html file, and placing in over the container over here. (index.html -> source path(host path) ; /usr/share/nginx//html/index.html -> container path (destination path) ). After this we have an entrypoint command, so whenever the container is running then this entrypoint line will be executed. So as the docker container start it will first go to /use/sbin/nginx and start the nginx service by giving the flag -g and daemon off. Daemon off basically helps bring the application to the foreground. So if daemon on then the application will be running in background. So because the this we can see the UI to the foreground and we can only see the UI when there is port number so at last we expose the application to the port 80 using EXPOSE, means our container is hosted on port 80.

Go to the terminal and go the directory where you want to make this docker file and create a docker file over there and paste the above content in it.

Then create a index.html file and paste the below content in it.

<html>

<head>

<link href=”<https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap.min.css>” rel=”stylesheet” integrity=”sha256-MfvZlkHCEwatNoGiOXveE8FIwMzZg4W85qfrfIFBfYc= sha512dTfge/zgoMYpP7bHy4gWMEGsbsdZeCXz7irItjcC3sPUFtf0kuFbDz/ixG7ArTxmDjLXDmezHubeNikyKGVyQ==” crossorigin=”anonymous”>

<title>Edureka’s Docker Nginx Tutorial</title>

</head>

<body>

<div class=”container”>

<h1>Hello Edureka Learners</h1>

<p>This nginx page is brought to you by Abhay and Edureka</p>

</div>

</body>

</html>

Sudo docker build -t mynginximage **.**  -> to build docker image

Sudo docker run -p 80:80 –name=App2 mynginximage -> ro run the docker container

Now go to firefox and check localhost:80

Now open another terminal

Sudo docker ps

Sudo docker stop <container\_id>

**Basic** **Docker Commands**

Docker –version

Docker –help

Docker pull

Docker run

Docker build

docker login

Docker push

Docker ps (ps stands for processes and this command use to see active containers)

Docker images

Docker stop

Docker kill

Docker rm (which stands for docker remove)

Docker rmi (which stands for remove images)

Docker exec (is used to access the bash of any active container)

Docker commit

Docker import

Docker export

Docker container

Docker compose

Docker swarm

Docker service

**docker --version**

This command is used to find out the version of your docker engine.

**docker --help**

This command list out all commands available in docker along with the description.

Open terminal and perform these two commands.

**docker pull**

This command used to pull a new docker image from dockerhub.

Eg., docker pull ubuntu

**docker images**

This command list down all the images available in our local repo.

Eg., docker images

**docker run**

This command is used to execute a docker image on your local repo and creates a running container out of it.

Eg., docker run ubuntu

Open terminal and executes these three commands

**docker build**

This docker build command is used to compile the dockerfile for building a custom image. Eg., suppose you have an ubuntu image but you do not want it as it is and we want to make few adjustments to that (like host a node.js application and nginx application on top of this image).

docker build -t MyUbuntuImage **.**

**docker container**

This container command is used to manage your container or to perform various operations on the container.

docker container logs -> to see container logs

docker container kill -> to remove the container

docker container rm ->

docker container run -> to start any inactive container

docker container start -> to start a stopped container

docker container kill $(docker ps -q) -> Kill all running containers

Eg., docker container logs <container\_id>

**docker login**

This command is used to login to dockerhub repo from the CLI.

docker login

Why we need to login to dockerhub account from CLI?

Because if we want to push any image present locally to docker hub.

**docker push**

This command is used to push a docker image from your local repo to the docker hub.

docker push abhayrajsr/MyUbuntuImage

<docker\_id>/<image\_name>

docker tag mycustomubuntu abhayrajsr/mycustomubuntu -> to make a copy the theimage with a given name.

Before pushing the image to dockerhub, we must change the name by using a tag command.

**docker ps**

This command lists all the running containers in the host. If ‘-a’ flag is specified, shutdown containers are also displayed. (ps stands for processes)

docker ps -> to see the active containers

docker ps -a -> to see all the containers including shutdown containers

**docker stop**

This command shuts down the container whose container id is specified in arguments. It takes sometimes because container is shutdown gracefully by waiting for other dependencies to shut.

docker stop container\_id

**docker kill**

This command kills the container by stopping its execution immediately. Its similar to force kill.

docker kill container\_id

**docker rm**

This command removes the container whose container id is specified in the arguments.

If we want to remove any container from the host, we have to first stop it by using stop or kill, after that we can remove it from our repository.

**docker rmi**

THis command removes the image from the repository whose name is specified in the arguments.

**docker exec**

This command is used to access an actively running container or access the bash of the container and perform the operations in the container.

docker exec -it <container\_id> bash or docker exec -i -t <container\_id> bash

-i means access your container in interactive mode.

Inside the container, we cannot access the files or any thing else in gui mode, like we cannot open a file by using gedit, since gedit opens it in gui mode, and vi or vim opens it in cli mode, so we can’t use gedit command inside the container or commands like these.

**docker commit**

This command creates a new image of an edited container on the local repository.

docker commit <container\_id> abhayrajsr/myeditedimage

We can verify by doing docker images, new image with this name must be present in local repo.

**docker export**

This command is used to export a docker image in your system into a tar file. THis tar file is going to be saved in your local file system, and it’s not going to be inside in docker anymore. This is another way of sharing your docker images other than uploading to dockerhub. THis method we used when image is very heavy so in that case this is the alternative for sharing in the industry.

docker export --output=”latest.tar” <container\_id>

latest.tar -> name of tar, we can specify any name. We can use .tar extension or not (optional)

**docker import**

THis command is used to import the contents of the tar file (usually a docker image) into your local repo.

docker import /home/edureka/Downloads/demo

/home/edureka/Downloads/demo -> path of tar file

Perform all the commands in the terminal.

**Advanced Docker Commands**

**docker compose**

This command is used to power multi-container applications where various services will be hosted inside different containers.

There are two variations to it, and the two syntax are:

docker-compose build

docker-compose up

These two commands work very similar to docker build and docker run.

Docker build is basically used to build a new image from a dockerfile. Similarly docker-compose build is ised to build your docker compose by using your docker yaml file.

YAML file stands for Yet Another Markup Language, in the YAML file,we can specify which all containers we want to be active and we have to specify the path of the different dockerfiles which will be used to create those containers or those services. That’s what docker compose does, it creates multiple services and it basically used to manage all those services and start them at one go so it would use more than one dockerfile.

For start your docker compose and start the container services then we can use docker-compose up command. This is very similar to docker run command.

Open terminal

docker ps -> to check running container

Go to a directory and make a folder with name MEAN-Stack-App

Here we will have our all required files.

Create a file docker-compose.yml

version: ‘3.0’ # specify docker-compose version

# Define teh services/containers to run

services:

angular: # name of the first service

build: angular-app # specify the directory of teh Dockerfile

ports:

- “4200:4200” # specify port mapping

express: # name of the second service

build: express-server # specify the directory of the Dockerfile

ports:

- “3000:3000” # specify port mapping

links:

* database # link this service to the database service

database: # name of the third service

image: mongo # specify image to build container from

ports:

* “27017:27017” # specify port forwarding

In this docker-compose.yml files we have commands to create three different services. One is angular service, second is express service and third is database service.

[Whole files of this app is in later explaination]

Now execute the yml file

docker-compose build -> to build the image

docker-compose up -> now the app is up and running in the container

Now check in browser

localhost:4200 -> to check for angular (it is ,mean-stack app, 4200 is front-end)

localhost:3000 -> it’s our server-end

localhost:27017 -> it’s our mongodb

docker ps -> to verify our containers running

**docker swarm**

Docker compose is for having the multi-container application, and Docker Swarm is however used to manage multiple docker engines on the various hosts. Till now, our docker engine is hosted on one particular host and we are executing our docker commands over there, even docker-compose did that on the same host, three different services were start on the same host but with docker swarm we can start those services in multiple machine (or hosts). So we will have a master machine, which is nothing but the docker manager and then we will have different slaves or the charcoal as worker in docker terms. So whatever service you start at the manager will be executed across all the machines which are there in that docker swarm cluster. So, it creates a network of docker engines or hosts to execute the containers in parallel and the biggest benefit of docker swarm is scaling up and ensuring high availability.

docker swarm init --advertise-addr 192.168.1.100 -> start off creating a docker swarm

192.168.1.100 - manager machine ip

As we execute this docker we got output as docker swarm is initialized and we will get a token. This token is for other machine to join the docker swarm by using the token. We just need to copy the token in another machine and enter then that machine will join as a worker.

docker swarm join -> for joining the swarm as a worker followed by token.

docker swarm join-token -> need to execute this command at manager machine when need to re-generate token

docker swarm leave -> if we want to leave the docker swarm cluster then run this command at worker machine

docker swarm leave --force -> to leave the docker swarm for manager machine

**docker service**

THis command is used to control any existing docker service, be a container, or be a docker compose or docker swarm or anything.

If we want to control your different nodes when you are in docker swarm then we can use the docker service command.

docker service ls -> to list down all teh nodes are there in your cluster

docker service ps -> to find out what container are being executed in a particular node

docker service scale -> if we want to sclae the number of containers supposing you have a cluster of 5 machines and 5 containers running in those machines, one in each, now if we want to scale those containers to 25, means 5 containers in each machines

docker service stop -> if we want to stop a container on any particular node

docker service logs -> to find out the logs

docker service rm

Docker service command is used in sync with docker swarm and docker compose primarily.

**DOCKER COMPOSE**

Docker compose is used to run multi container applications. Each container will run a stand alone application and it can communicate with other containers present in the same host. Eg., MEAN Stack Application [MongoDB, ExpressJS, Angular and NodeJS] using different containers.

We use one container usually to host one service.

Let’s take a case of massive big application, and it has multiple services and there are multiple web servers which need to be separately placed on a particular server or on a particular VM. because ⅔ servers cannot be hosted on the same machine since there might be an overhead. SO at that time what we usually do is we create a new VM and we hosted there or we have a new server. For example if we ewant to monitor our application then we probably use nagios, so we have to host it separately on a different machine. Similarly we can have many servers and services like jenkins, etc. SO at that time, instead of having a different machine or having a different VM, we can simply have a different container. We can have these multiple servers or services hosted in multiple containers. SO each container will be hosting one service and these containers would be run such that they will be interact or communicate with one another.

Now we will be setting up MEAN stack application.

In Mean stack, it has four different things. M stands for MongoDB which is the database, and E stands for Express, A stands for angular and N stands for NodeJS. Together it is a full stack application. Now since we are using the combination of these four technologies, the acronym is MEAN stack application. THis full stack application is a web service , in which we have a front end client and a back end server nad then we have a database. So whenever we have clients and our customers interacting with our web server, they would be interacting with the client. The data they (client) pass over there whatever actions they perform or whatever requests they make that would go to the server and the server will do teh necessary function, and sometimes they need to fetch data from the database. So in that case it would fetch data and provide a response. So the actual functionality would be done by the serverand the displaying part would be done by the client nad the actual data would be stored inside the database. So that’s how full stack application works, the combination of these three services, the front end client, the back end server and the database.

For these three services we need three different containers. One for mongodb database, one for back end server express and nodejs, and one for front end angular. These three containers will be build from three dockerfile which would be called by docker compose yml file.