**Docker**

There are various life cycle of software development:

1. Design
2. Development
3. Deployment
4. Testing

Docker makes the process of application deployment very easy and efficient and resolves a lot of issues related to deploying application.

Docker is the world’s leading software container platform.

So a developer will package all of the software’s components, libraries into simple **CONTAINER**. Docker will take care for shipping this container to all the platform in a standard way.

So, now developer should only concern about creating the code and the software and will package the software along with all its dependencies and libraries and not worry about how it is deployed on what al platform.

* Docker is a standard for Linux containers.
* Container is an isolated runtime inside of Linux
* A container provides a private machine like space under Linux
* Containers will run in any modern Linux kernal.

**How it works?**

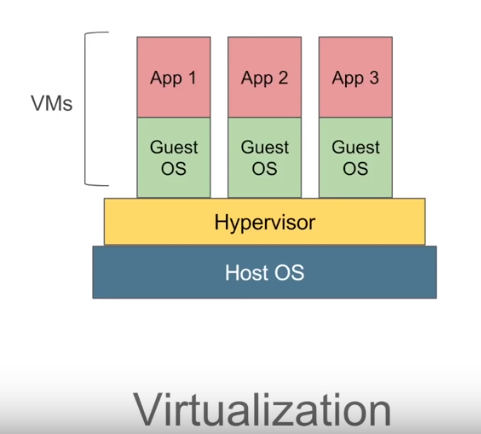
In general scenario, developer will define all the application and its dependencies in a file called **Dockerfile**. This Dockerfile will be used to create the docker image. So, in Docker image all the application and its dependencies is present. When you run the docker image you get docker containers. Docker containers are the runtime instances of the Docker image.

These Docker images can also be stored in the online cloud repository which is called Docker Hub.

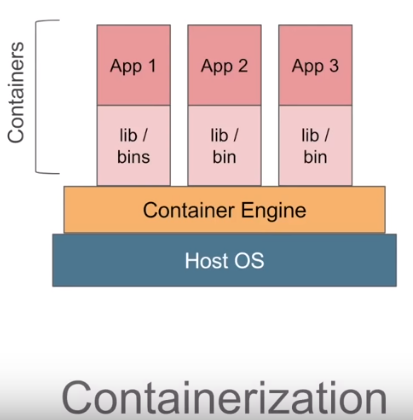
These images can be pulled to create containers in any environment.

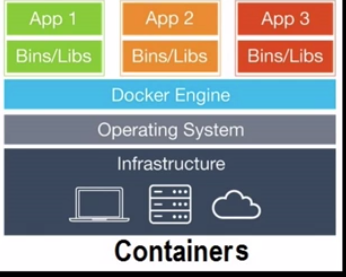
**Containerization VS Virtualization**

In virtualizatiuon, we have a software called Hypervisor which we use in our Host OS. Hypervisor is used to create and run virtual machines. Using Hypervisor, we can create multiple VM’s on the host OS. These virtual machine have their own OS i.e, it does not uses the host operating system. So, there can be a overhead on host platform. Also we have to allocate fixed memory for every VM’s so there can be a wastage of lot of memory and space.

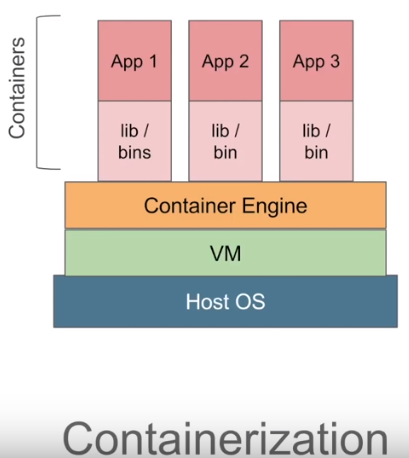


In **containerization**, we have a container engine(Docker Engine) and we do not have separate OS but we have container which have application and all its dependencies. It will use the host operating system unlike Virtualization which uses its own OS. Here, memory, space, other resources are not fixed i.e, they are taken dynamically so it is very fast and lightweight.





There can be a scenario where we need VM over our host OS and then have container over VM. For example, if you want to use windows operating system over linux OS we need to have VM first which will have a windows OS and we can have a containers over it.



Docker has a client server architecture,

In Docker, Command line interface is a client and we have a Docker server or a Docker Daemon which will have all the containers. Docker server receives the commands from the Docker client in the form of commands or a rest API request. Docker client with Docker server together form a ***DOCKER ENGINE***.

Docker Client and Daemon can be present in the same or different HOST(machine).

**Advantages:**

It resolves a problem of a code working on one system and not working on different system. So, with Docker you can build your application only once and then there is no need to build or configure multiple times on different encironments of platforms.

Suppose we have an application on Java. We build it and using maven, In Dockerfile we create an image which will contain that jar/war, jvm and all the environment in which it is running.

So, In short docker image will contain that jar/war and all the environment setup.

We now can hand this image to anyone and the jar/war will not give the trouble to run as docker image will have all the environment set up already.

We run this image which will lead to the creation of the instance of that image which is called as container.

Container does not contain the full OS, and that’s why it it very light weight and much more efficient than running VM’s.

Suppose we have Linux OS. Containers are kind of processes that runs on that Linux Kernal. All the containers shares the same Linux Kernal that is why containers are different than VM’s as each VM has its own kernal.

Instead of building war/jar file and handing it over for the deployment, we built a Docker Image. Then its easier for the deployer as they just run the container on that hardware.

These containers are nothing new, as they are there for a very long time as they are the features of Linux kernal since 2008.

Following steps is done by docker :-

1. The Docker client contacts the Docker daemon.
2. The Docker daemon pulls image from the Docker Hub.
3. The Docker daemon creates a new container from that image which runs the executable that produces the output.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.
5. Images in docker are immutable

**Docker Toolbox VS Docker Windows**

**Docker Toolbox:** In older version of windows, you need to have Docker Toolbox. The difference is that in the background it will install a copy of Oracle Virtual box and runs Linux kernal in a Virtual Machine(VM) all in the background.

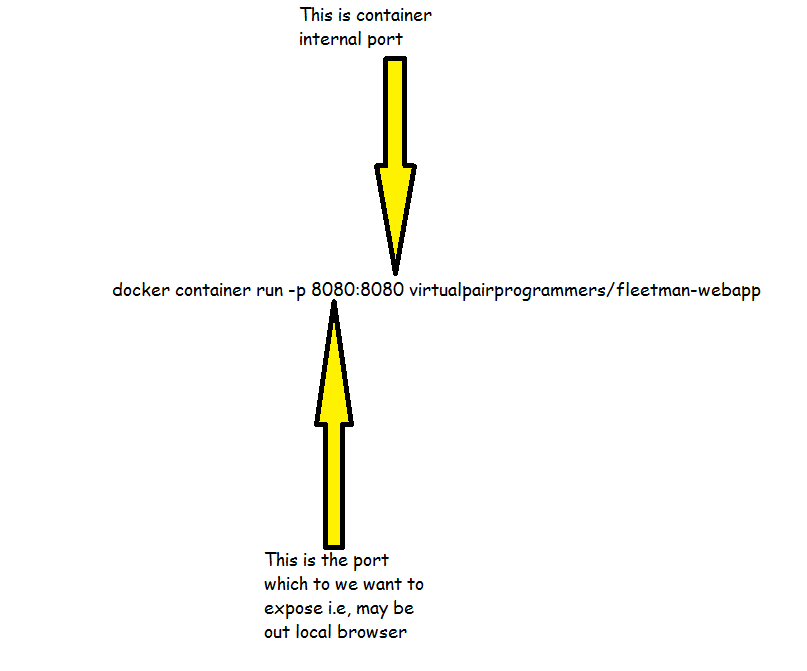
**Docker or Windows/Mac:** In this also Linux virtual machine is running, but you are using the native virtualization support from your HOST OS(Hyper-V on windows). So, it does not have to run Oracle Virtual box.

**Commands**

1. docker image pull <IMAGE NAME>

* download an image from DockerHub

1. docker image ls
2. docker-machine ip
3. docker rmi $(docker images -q)



To install jdk in the container, we use interactive bash of link:-

$ docker container run –it ubuntu

root@12332:/# apt-get install openjdk-8

root@12332:/# javac

root@12332:/# exit

$ docker container commit -a "tanuj tripathi tanuj.tiwari99@gmail.com" <container-Id> myjdkubuntuimage

In the above we are making use of the ubuntu image which is available publicly and adding jdk in it.

If we want to do it through Dockerfile:-

FROM ubuntu:latest

MAINTAINER Tanuj Tripathi "tanuj tripathi [tanuj.tiwari99@gmail.com](mailto:tanuj.tiwari99@gmail.com)"

RUN apt-get update && apt-get install –y openjdk-8-jdk

CMD [“/bin/bash”]

$ docker image build -t jdk-image-from-dockerfile .

The above Dockerfile will do the exact same thing as the commnds used before.

Here, . represents the current directory

Now that we have image of ubuntu and jdk installed in it, we also want our java program jar to be added in it.

FROM ubuntu:latest

MAINTAINER tanuj tripathi "tanuj.tiwari99@gmail.com"

RUN apt-get update && apt-get install -y openjdk-8-jdk

WORKDIR /usr/local/bin/

ADD test-program.jar .

ENTRYPOINT ["java", "-jar", "test-program.jar"]

#**How to deploy your WAR file into tomcat container:**

Suppose you have WAR file and most of the time we don’t have embedded tomcat as spring boot provides. In case of spring boot application we can user packaging as jar and build the image out of it and we can simply run it.

In case our jar/war does not have embedded tomcat:

**FROM** tomcat:latest  
  
**MAINTAINER** tanuj tripathi "tanuj.tiwari99@gmail.com"  
  
**EXPOSE** 8080  
  
# Removing contents inside webapps to stop exposing ROOT inside it to any user  
**RUN** rm **-**rf **/**usr**/**local**/**tomcat**/**webapps**/\***#Sending WAR file accross the tomcat  
**COPY** .**/**target**/**fleetman-0.0.1**-**SNAPSHOT.war **/**usr**/**local**/**tomcat**/**webapps**/**ROOT.war  
  
**ENV** *JAVA\_OPTS*="-Dspring.profiles.active=docker-demo"  
  
**CMD** ["catalina.sh", "run"]

612415803@BTG714382 MINGW64 /c/NotBackedUp/myWorkSpace\_intellij/Chapter7/fleetman-webapp

$ docker image build –t fleetman-webapp **.**

$ docker container run –p 8080:8080 fleetman-webapp

Now, inside above Dockerfile we are copying our WAR file of our application to the tomcat webapps directory as ROOT.war

#Sending WAR file accross the tomcat  
**COPY** .**/**target**/**fleetman-0.0.1**-**SNAPSHOT.war **/**usr**/**local**/**tomcat**/**webapps**/**ROOT.war

**Networking of the Containers:**

We will create mysql container now which will have fleetman database.

$ docker container run -e MYSQL\_ROOT\_PASSWORD=password -d mysql:5

$ docker container exec –it <container-id> bash

root@12332:/# mysql –uroot –ppassword

mysql> show databases

+------------------------------------------+

| Database |

+------------------------------------------+

| information\_schema |

| mysql |

| performance\_schema |

| sys |

+------------------------------------------+

mysql> exit

Bye

root@12332:/# exit

$ docker container run -e MYSQL\_ROOT\_PASSWORD=password –e **MYSQL\_DATABASE=fleetman** -d mysql:5

$ docker container exec –it <container-id> bash

root@12332:/# mysql –uroot –ppassword

mysql> show databases

+------------------------------------------+

| Database |

+------------------------------------------+

| information\_schema |

| mysql |

| performance\_schema |

| sys |

| fleetman |

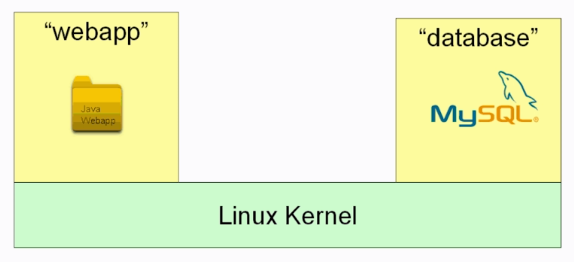
+------------------------------------------+

mysql> exit

Bye

root@12332:/# exit

So, now we have two Containers, first being our fleetman and another being mysql container.



We want to network these two containers together.

We can do it by simply using the container name as a domain name so we do not need to deal with IP address.

So, while running we will give it name and assign our private network to it

$ docker network create my-network

$ docker container run **--network my-network --name database** -e MYSQL\_ROOT\_PASSWORD=password -e MYSQL\_DATABASE=fleetman -d mysql:5

$ docker container run -d -p 8080:8080 **--network my-network --name fleetman-webapp** fleetman-webapp

$ docker container exec –it <container-id> bash

root@12332:/# ping database

OR

--rm : when the container stops it will be removed also

$ docker network create my-network

$ docker container run **--network my-network --name database** -e MYSQL\_ROOT\_PASSWORD=password -e MYSQL\_DATABASE=fleetman -d **--rm** mysql:5

$ docker container run -d -p 8080:8080 **--network my-network --name fleetman-webapp** --rm fleetman-webapp

$ docker container exec –it <container-id> bash

root@12332:/# ping database

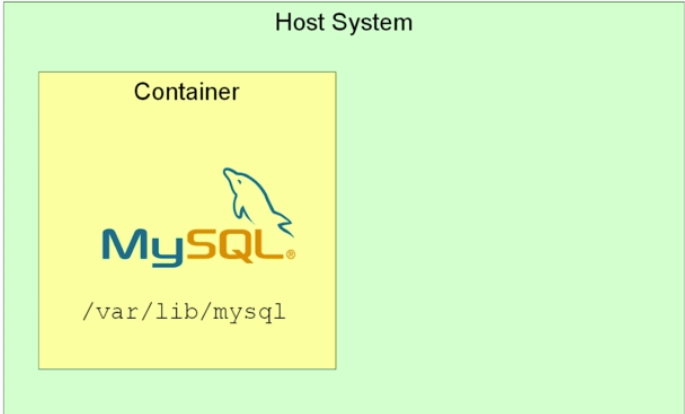
**Volumes**

When we use any container and after that if we remove that all the information is lost along with it. Suppose, in our condition we are using database container. So, we have pulled the image of mysql:5 and given cmd line argument to create DB fleetman. Now we have fleetman database as a part of this image and when we run instance of this image and create any table indside fleetman database and put data inside it, it will work till you have not removed that container. The instant you remove the container all the data inside fleetman DB get also deleted

If you look at Dockerfile of mysql in docker hub you will find this:

VOLUME /var/lib/mysql

So, when we run the mysql container it will have data persisted in /var/lib/mysql



So, in the HOST system they are stored somewhere safe. So, when the container is deleted the data is still present in /var/lib/mysql somewhere in the HOST system. We can mount this /var/lib/mysql to some folder of the HOST sytem.

If we do:

$ docker container inspect <container-id of mysql database>

We will get large chunk of JSON and somewhere in that we have mount

"Mounts": [

{

"Type": "volume",

"Name": "0583502dd6d863896c1949233be95360a3a091e60356626b3db49aebb659cfb7",

"Source": "/mnt/sda1/var/lib/docker/volumes/0583502dd6d863896c1949233be95360a3a091e60356626b3db49aebb659cfb7/\_data",

"Destination": "/var/lib/mysql",

"Driver": "local",

"Mode": "",

"RW": true,

"Propagation": ""

}

],

So, as you can see its Type is “volume” and Destination is ‘’var/lib/mysql”

And if you see Source you can see the destination of data is somewhere in linux VM. If you are in linux you can directly see, but if you are in windows/mac you can use bash to navigate to it.

You can see all the volumes that has been store in the HOST by doing:

$ docker volume ls

DRIVER VOLUME NAME

local 0583502dd6d863896c1949233be95360a3a091e60356626b3db49aebb659cfb7

local 404764461bc0bd6f2d3c7c25ed1b04da1b19cd62bc8d3ec3589b139c433ee585

local a2d260022c58a039e96a5250697cfadbb8bf9eb324dbfbd48c4193d616b61e4a

local ab7b67576ea501624274316db0c395625f5577f94bf85de165cb3f7f005f12c9

local c5796dbe4e36e61dcec61b7cc52b799015e737c879c514d1294145e9137fe729

Above, you can see the list of volumes we have created each time we run the mysqk container.

Good practise is to use Name for the volumes of these containers.

If we use a container and delete it and at some point of time we need that data, we can remount that data to the image.

So, to give nam eto our volume:

$ docker container run **-v mydata:/var/lib/mysql** -e MYSQL\_ROOT\_PASSWORD=password -e MYSQL\_DATABASE=fleetman -d mysql:5

$ docker volume ls

DRIVER VOLUME NAME

local mydata

$ docker volume inspect mydata

[

{

"CreatedAt": "2019-06-28T06:29:01Z",

"Driver": "local",

"Labels": null,

"Mountpoint": "/mnt/sda1/var/lib/docker/volumes/mydata/\_data",

"Name": "mydata",

"Options": null,

"Scope": "local"

}

]

So, now if we delete this container and run it again using the same comand having **-v mydata:/var/lib/mysql** will remount the data to the image.

**Maven**

We use Fabric8 Docker maven plugin to user Docker with Maven.

<plugin>  
 <groupId>io.fabric8</groupId>  
 <artifactId>docker-maven-plugin</artifactId>  
 <version>0.21.0</version>  
  
 <configuration>

<!--<dockerHost>tcp://192.168.99.100:2376</dockerHost>-->

<!-- this is for Mac and Amazon Linux -->  
 <!-- <dockerHost>unix:///var/run/docker.sock</dockerHost> -->  
  
 <verbose>true</verbose>   
  
 <images>  
 <image>  
 <name>tanujt1/dockerapps</name>  
 <build>  
 <dockerFileDir>${project.basedir}/src/main/docker/</dockerFileDir>  
  
 <!--copies Jar to the maven directory (uses Assembly system)-->  
 <assembly>  
 <descriptorRef>artifact</descriptorRef>  
 </assembly>  
 <tags>  
 <tag>latest</tag>  
 </tags>  
 </build>  
 </image>  
 </images>  
 </configuration>  
</plugin>

This is fabric8 plugin which has above group and artifact id.

Image name will be the name which you have defined in docker hub as one of you repository

We have changed the location of the Dockerfile and moved it to **${project.basedir}/src/main/docker**

Now, we were following these steps before:-

1. Build the image manually by mvn clean install
2. Build docker image manually by:-

* docker image build –t fleetman-webapp .

1. For docker build we were using mvn clean package docker:build
2. Now finally we were doing:-

* docker push

From maven we can do automation of that, i.e, if we build, it will automatically execute docker:build and docker:push command. Now, to push your war/jar into docker hub maven needs to login to docker hub. One way to do it is to give the username and password in POM itself but its not good practise so we give that in the settings.xml of maven.

In POM under fabric8 plugin

<configuration>

<authconfig>

<username>tanujt1</username>

<password>\*\*\*\*\*\*\*\*\*\*</password>

</authconfig>

<configuration>

**OR**

Under setting.xml

<servers>

<server>

<id>docker.io</id>

<username>tanujt1</username>

<password>\*\*\*\*\*\*\*\*\*\*</password>

</server>

</servers>

***Ques: Why we want to move dockerfile in the src/main/docker ?***

*ANS: In the root folder i.e, our fleetman-app there are lot of assets. Whenever we run a build docker runs as a daemon process in our machine.*

*We use docker image build –t fleetman-webapp .*

*So, this . tells docker that under current folder docker is going to need all the files to build an image so, this command will zip all of the folders/subfolders inside the root directory and sends it across the docker daemon through a network connection. It seems pretty inefficient because every time we build the image its going to zip the entire folder structure and do the very expensive network transfer. In reality, to build an image all it needs is Dockerfile and the Jar file that we are building.*

*So, we move our Dockerfile to src/main/<some-directory>*

*Now, when we build the image, it will only zip up the <some-directory> and its components which in our case is only Dockerfile.*

*Now, that we have Dockerfile we need jar file which resides under target folder.*

<!--copies Jar from the target folder-->  
<assembly>  
 <descriptorRef>artifact</descriptorRef>  
</assembly>

*Here, above will help image build to give jar file from the target folder. So, alongside Dockerfile we have jar file also.*

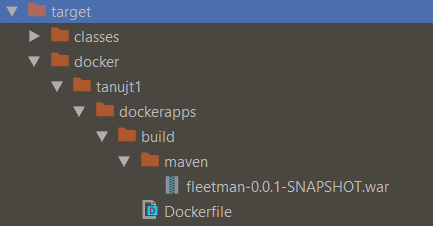
*We also needs to change*

**COPY** .**/**target**/**fleetman-0.0.1**-**SNAPSHOT.war webapp.jar

*To*

**COPY** maven**/**fleetman-0.0.1**-**SNAPSHOT.war webapp.jar

*Instead of ./target we are using maven as you can see from the below image fleetman-0.0.1-SNAPSHOT.jar is created under target/docker/tanujt1/dockerapps/build/maven/*



*For building image for CMD we need:-*

*$ docker image build –t fleetman-webapp .*

*For building image drom maven we need:*

*mvn clean package docker:build*

*After building it we push it into docker hub so we need from CMD:*

*$ docker push*

*And from maven we need*

*mvn clean docker:push*

*Steps:*

1. *mvn clean package docker:build*
2. *mvn clean docker:push*

*We want to automate this also so we use this;-*

<executions>  
 <execution>  
 <phase>package</phase>  
 <goals>  
 <goal>build</goal>  
 </goals>  
 </execution>  
</executions>

*We are not giving docker:build in <goal> as it is already under docker-maven-plugin*

<plugin>  
 <groupId>io.fabric8</groupId>  
 <artifactId>docker-maven-plugin</artifactId>

*Now, when we run mvn clean package, it will build our application jar as well as it will build the docker image. Now we only need to push it.*

*Now, if we want to automate push also we can use use maven predefined mvn deploy as set it as an another goal.*

*But, Generally we don’t want to automate push as when we are sure that this is the image we want to push to hub then only we shoukd do it as in real scenario it can takes upto 10 times of image build and in the end push it. If we automate it,, every time when we build an image it will push it to dockdr hub.*

<executions>  
 <execution>  
 <phase>package</phase>  
 <goals>  
 <goal>build</goal>  
 </goals>  
 </execution>  
  
 <execution>  
 <id>mydeploy</id>  
 <phase>deploy</phase>  
 <goals>  
 <goal>push</goal>  
 </goals>  
 </execution>  
 </executions>  
 </plugin>  
  
 <plugin>  
 <artifactId>maven-deploy-plugin</artifactId>  
 <configuration>  
 <skip>true</skip>  
 </configuration>  
 </plugin>  
</plugins>

*mvn deploy:*

* deploy: *copies the final package to the****remote****repository for sharing with other developers and projects*.

***Docker Compose***

*Docker compose is a separate tool from the Docker. We write long commands in CMD. Docker compose allows us to write these tedious commands in simple text files.*

*$ docker container run -d*

*--network my-network*

*--name database*

*-e MYSQL\_ROOT\_PASSWORD=password MYSQL\_DATABASE=fleetman mysql:5*

*We use docker-compose.yml file.*

**version**: "3"  
  
*# docker container run -d --network fleetman-network --name fleetman-webapp -p 8080:8080  
# virtualpairprogrammers/fleetman-webapp***services**:  
 **fleetman-webapp**:  
 **image**: virtualpairprogrammers/fleetman-webapp  
 **networks**:  
 - fleetman-network  
 **ports**:  
 - 8080:8080  
  
 **database**:  
 **image**: mysql:5  
 **networks**:  
 - fleetman-network  
 **environment**:  
 **MYSQL\_ROOT\_PASSWORD**: password  
 **MYSQL\_DATABASE**: fleetman  
  
**networks**:  
 **fleetman-network**:

*Here, we are using spring boot and while starting spring boot will look for database. In this case it will fail as database will not start and there is no way spring boot will retry. So, we have to have a mechanism to tell spring boot that don’t start the fleetman-webapp component it database component is not UP.*

*So, according to the documentation we should use depends\_on in yml which will ensure that when database container will run then only fleetman webapp will run.*

**depends\_on**:  
 - database

*But, this also does not solve the problem because the thing is as soon as the database container starts running, fleetman-webapp also starts running and fails as database container takes few seconds to start.*

**version**: "3"  
  
*# docker container run -d --network fleetman-network --name fleetman-webapp -p 8080:8080  
# virtualpairprogrammers/fleetman-webapp***services**:  
 **fleetman-webapp**:  
 **image**: virtualpairprogrammers/fleetman-webapp  
 **networks**:  
 - fleetman-network  
 **ports**:  
 - 8080:8080  
 **depends\_on**:  
 - database  
  
 **database**:  
 **image**: mysql:5  
 **networks**:  
 - fleetman-network  
 **environment**:  
 **MYSQL\_ROOT\_PASSWORD**: password  
 **MYSQL\_DATABASE**: fleetman  
  
**networks**:  
 **fleetman-network**:

***Swarm***

*Docker swarm enables us to run Docker container in a cluster. It can give re deployment in zero down time. So, this is a tool that can manage your containers in production. There is a very popular tool called kubernetes which came from google. Kubernetes is an orchestration system for docker. It can use not only Docker environment as well but it works brilliantly with Docker.*

*Docker contains its inbuilt orchestration system that is called Docker swarm. You can think Kubernetes and swarm in competition. If you want to run your Docker containers Live then you have to choose between the Kubernetes and Docker Swarm.*

*Most of the Live projects out in the world that are using orchestration systems has chosen Kubernetes.*

*Here, We have two services(Containers)*

1. *Spring boot App --name fleetman-webapp*
2. *Mysql:5 --name database*

*The main difference between service and container is that Services are resilient to the failures i.e, if any service is down it automatically use retry mechanism.*

*So, in production there can be several Containers and if for some reason one of the Containers fails Docker Swarm restart it automatically.*

*To run 1000 of containers you need to have a very expensive Server so in real scenario we distribute containers horizontally i.e, in different physical machines.*

*So, here we will distributes among multiple Nodes.*

*Suppose, 4 Nodes are there and we have two services(Containers). Now, if you run the container Docker-Swarm will decide on which Node the Docker Container will run. So, roughly Docker Swarm behaves like the Load balancer.*

*Just for the sake of learning, We will start by having Single Docker Swarm Node which is pretty much useless :p*

*Commands:*

*ca Display and rotate the root CA*

*init Initialize a swarm*

*join Join a swarm as a node and/or manager*

*join-token Manage join tokens*

*leave Leave the swarm*

*unlock Unlock swarm*

*unlock-key Manage the unlock key*

*update Update the swarm*

$ docker swarm init –advertise-addr 192.168.99.100

Swarm initialized: **current** node (qwbgjdkgn363fllnwftbnoi7a) **is** now a **manager**.  
  
To add a worker tothis swarm, run the **following** command:  
  
 docker swarm **join** --token SWMTKN-1-5tiz5edjx1vljat2ctdlielgffwcjbvvh5zpsvjefncqjzhgqp-df8zp4xeeg2j1httdm5n07jx8 192.168.99.100:2377  
  
To adda **manager to** this swarm, run 'docker swarm join-token manager' **and** follow the instructions.

*--advertise-addr is used when your system is using multiple adresses on different interfaces.*

*# Launch our Container into our Swarm*

$ docker service create -d --network fleetman-network -e MYSQL\_ROOT\_PASSWORD=password -e MYSQL\_DATABASE=fleetman --name database mysql:5

Error: No such network: fleetman-network

*We are getting this error because fleetman-network is normal bridge network and for swarm we need overlay network.*

*In swarm different phiysical machine are connected i.e, host machines are not same for every container. So Swarm has a different type of network: overlay network.So, this overlay network acts as a bridge between different physical machines.*

*To create this overlay network we use slight different command.*

$ docker network create --driver overlay fleetman-network

612415803@BTG714382 MINGW64 /c/NotBackedUp/myWorkSpace\_intellij/Chapter7/fleetman-webapp

$ docker service create -d --network fleetman-network --name database -e MYSQL\_ROOT\_PASSWORD=password -e MYSQL\_DATABASE=fleetman mysql:5  
jko4u8bya42tzhmruqyb93nj9  
  
612415803@BTG714382 MINGW64 /c/NotBackedUp/myWorkSpace\_intellij/Chapter7/fleetman-webapp  
$ docker service ls  
ID NAME MODE REPLICAS IMAGE PORTS  
jko4u8bya42t database replicated 1/1 mysql:5

*So, now when you have run this command, docker swarm would have created the container in the background.*

*We can test different Nodes in docker play in* <https://labs.play-with-docker.com/>

Here ,

$ docker swarm init –advertise-addr 192.168.99.100  
**Swarm initialized**: current node (qwbgjdkgn363fllnwftbnoi7a) is now a manager.  
  
**To add a worker to this swarm, run the following command**:  
  
 docker swarm join --token SWMTKN-1-5tiz5edjx1vljat2ctdlielgffwcjbvvh5zpsvjefncqjzhgqp-df8zp4xeeg2j1httdm5n07jx8 192.168.99.100:2377  
  
 To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

**Swarm initialized**: current node (qwbgjdkgn363fllnwftbnoi7a) is now a manager.

So one Node is a manager Node and other Node are worker Node. We can make other Nodes as worker Nodes by using:-

docker swarm join --token SWMTKN-1-5tiz5edjx1vljat2ctdlielgffwcjbvvh5zpsvjefncqjzhgqp-df8zp4xeeg2j1httdm5n07jx8 192.168.99.100:2377

*Managers VS Workers*

*There can be multiple Managrs Nodes and workers node. They both will have containers running in them. The only difference between them is we can manage services from manager nodes.*

*In docker playground you can use several physical Nodes.*

*Here, in Node1 we are initializing swarm:-*

*[node1] (local) root@192.168.0.37 ~*

*$ docker swarm init --advertise-addr 192.168.0.37*

*Swarm initialized: current node (sht8pibwcczqdio9pv58ibu8t) is now a manager.*

To add a worker to this swarm, run the following command:

***docker swarm join --token SWMTKN-1-abcdi 192.168.0.37:2377***

*To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.*

*Now, we are joining Node2 as a worker node*

*[node2] (local)* [*root@192.168.0.37*](mailto:root@192.168.0.37) *~*

*$ docker swarm join --token SWMTKN-1-abcdi 192.168.0.37:2377*

*This node joined a swarm as a worker.*

*So, now Node1 is a manager Node and Node2 is a worker Node.*

*$ docker node ls*

***ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS ENGINE VERSION***

*csmyzw node1 Ready Active 19.03.0-beta2*

*sht8p \* node2 Ready Active Leader 19.03.0-beta2*

***#NOTE:*** *We can run docker node ls in manager node only and we can not run swarm commands in worker Node.*

**Stacks**

We remember that in docker-compose can help us managing all the commands in a file.

**version**: "3"  
  
*# docker container run -d --network fleetman-network --name fleetman-webapp -p 8080:8080  
# virtualpairprogrammers/fleetman-webapp***services**:  
 **fleetman-webapp**:  
 **image**: virtualpairprogrammers/fleetman-webapp  
 **networks**:  
 - fleetman-network  
 **ports**:  
 - 8080:8080  
  
 **database**:  
 **image**: mysql:5  
 **networks**:  
 - fleetman-network  
 **environment**:  
 **MYSQL\_ROOT\_PASSWORD**: password  
 **MYSQL\_DATABASE**: fleetman  
  
**networks**:  
 **fleetman-network**:

Stack is like single application, we can use stack just as docker-compose file

So, In docker swarm instead of writing those big commands, we can use stack.

*[manager1] (local)* [*root@192.168.0.37*](mailto:root@192.168.0.37)*~*

$ apk add --no-cache nano

fetch <http://dl-cdn.alpinelinux.org/alpine/v3.9/main/x86_64/APKINDEX.tar.gz>

fetch <http://dl-cdn.alpinelinux.org/alpine/3.9/community/x86_64/APKINDEX.tar.gz>

(1/1) Installing nano (3.2-r0)

Executing busybox-1.29.3-r10.trigger

OK: 301 MiB in 110 packages

*[manager1] (local)* [*root@192.168.0.37*](mailto:root@192.168.0.37)*~*

$ nano docker-compose.yml

// paste above yml file and press ctrl+o enter ctrl+x

*[manager1] (local)* [*root@192.168.0.37*](mailto:root@192.168.0.37)*~*

$ docker stack deploy -c docker-compose.yml fleetman-stack

Creating network fleetman-stack\_fleetman-network

Creating service fleetman-stack\_database

Creating service fleetman-stack\_fleetman-webapp

*[manager1] (local)* [*root@192.168.0.37*](mailto:root@192.168.0.37)*~*

$ docker service ls

ID NAME MODE REPLICAS IMAGE PORTS

un fleetman-stack\_database replicated 1/1 mysql:5

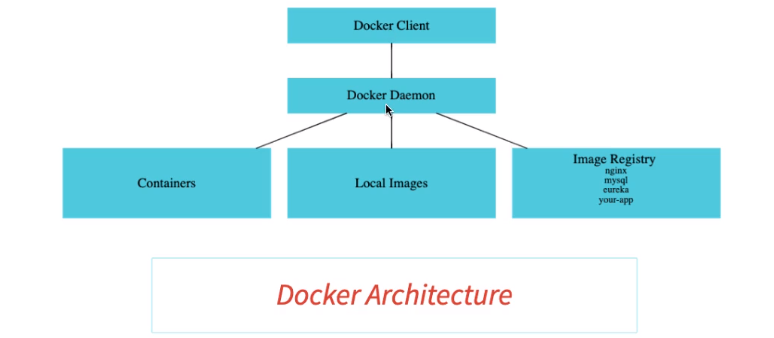
br fleetman-stack\_fleetman-webapp replicated 1/1 virtualpairprogrammers/fleetman-webapp:latest \*:8080->8080/tcp

Visualiser.

Doceker swarm visualizer is an opern source product which helps us to see docker containervisually.

docker run -it -d -p 8080:8080 -v /var/run/docker.sock:/var/run/docker.sock dockersamples/visualizer

**II Docker Course**



1. We type commands in Docker client.
2. These commands are sent out to something called docker daemon/Docker Engine for execution.
3. It’s a client server architecture.

**Namespaces PID’s**

Suppose you have to set an end to end application stack including various different technologies like a web server like Node Js and the database like mongo DB, messageing system like Redis, and orchestration tool. You will face lot of issues developing this with all this different components. First of all their underline

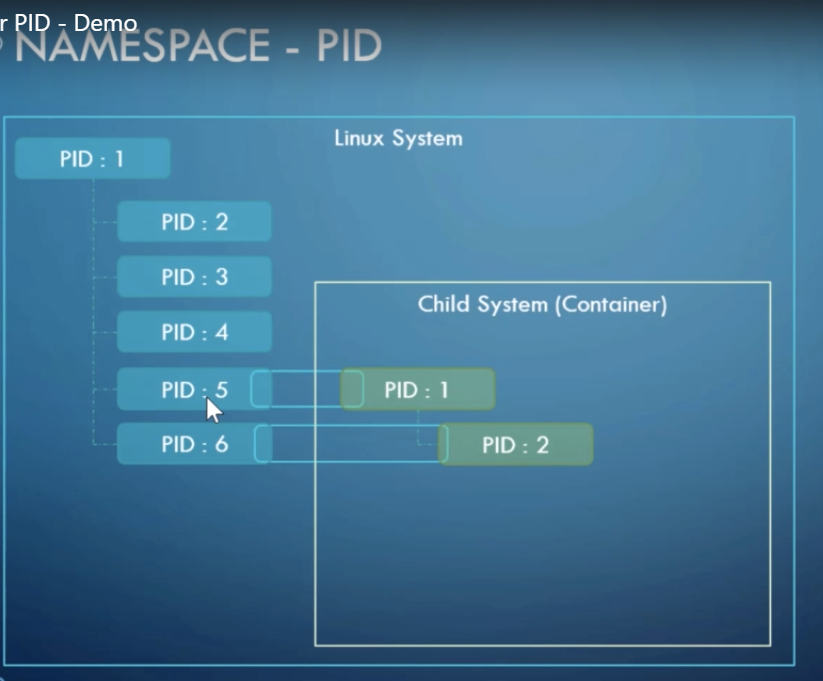
Compatibility with the OS. You will have to make sure that all these services are compatible with the version of OS you are using. There are times when certain version of certain technologies are not compatiable wiith the OS version of this application stack. You have to go back and look at the different OS that is compatiable with all of these techs. So every time we upgrade the Technologies, same process we have to go again.

Every Host System have one one root process, i.e, as soon as your OS boots up, It has its root process ID that is always 1. Other processess runs over it. Docker Engine uses the namespace concept to map the HOST PID’s to it.

Suppose you have one root process Id PID:1 and other process ID’s PID:2, PID:3, PID:4 runs over it. Now you Launch the Docker Engine That will also have root process ID: 1 and runs container over it having PID 2.

**Ques: If Docker Engine runs on the same HOST OS, How is this Possible to have same PID’s 1 and 2 if that already exists?**

If you check the process ID’s inside the engine you will see PID1 and PID 2 and not PID 5 and PID 6 because Docker create namespace PID: 5 and maps it as a root PID: 1 and rund PID: 6 under it giving it name PID: 2 in a Docker Engine. Under the hood all the process ID’s are run over same host OS root PID 1.



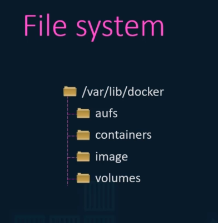
**Storage**

**How and where Docker stores the file structure??**

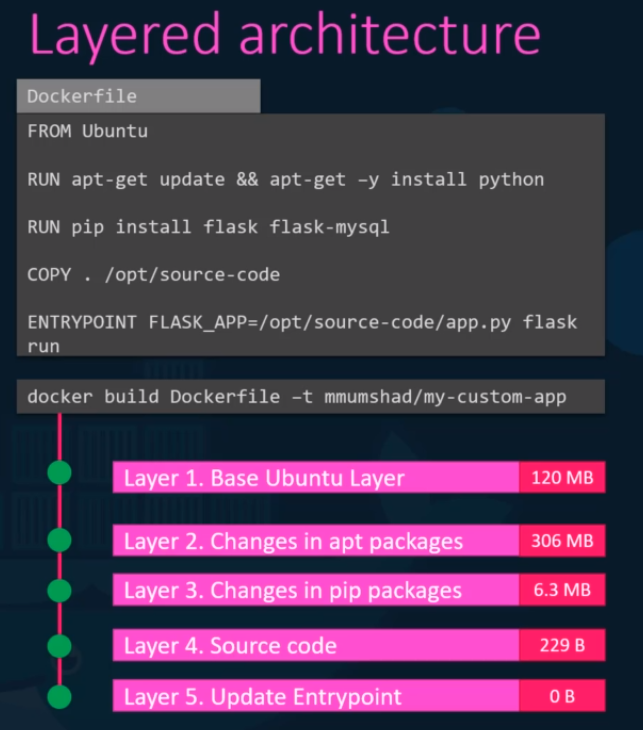
When you install the Docker in a System, It creates a folder structure as /var/lib/docker

It has multiple folders inside it that contains aufs, containers, image, volumes

This is where Docker stores its data(file, containers, images etc) by default.



When docker build the images, it builds in a Layered architecture. Each line inside Dockerfile creates a new Layer in the docker image



Suppose you change any instruction say layer 4, Till Layer 3 Docker will use cache and from layer 4 onwards it does not uses cache.

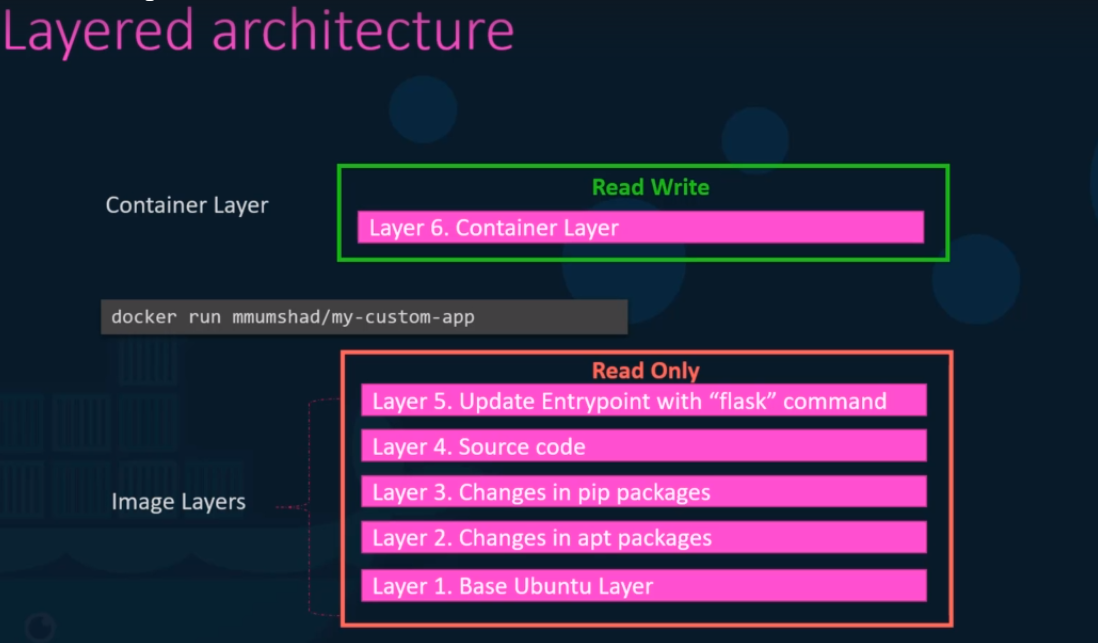
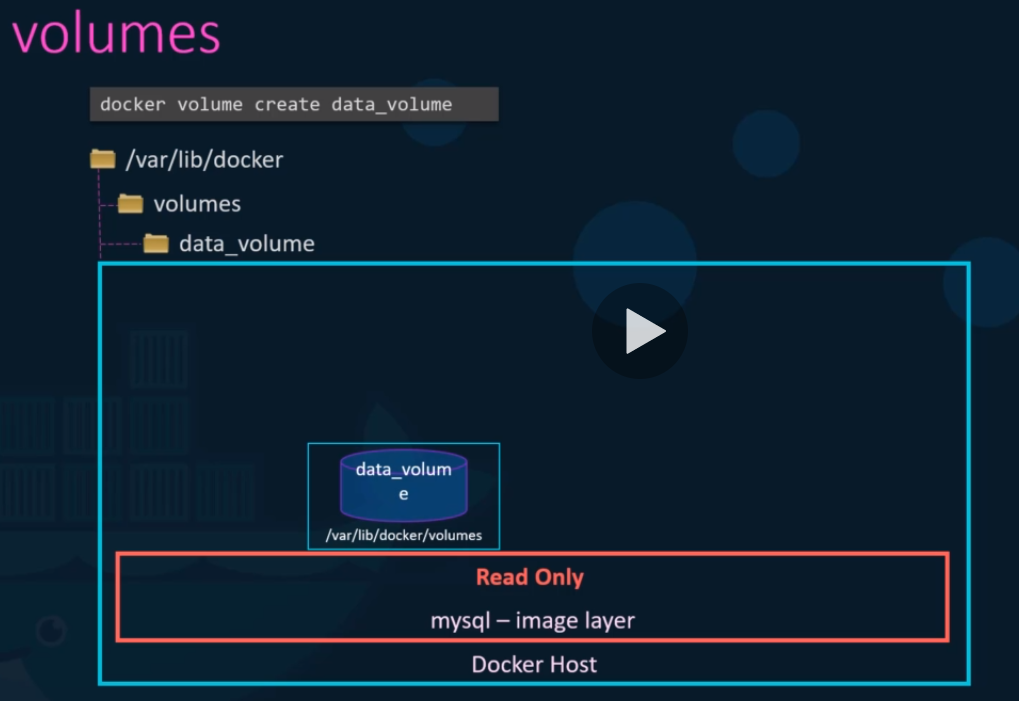


Image Layer is read only i.e, once you build an Image you can not change it. You need to re build to make the changes.

While container layer is READ and WRITE. We can modify the file inside container. This won’t affect another container as when you modfy Docker will create a copy of that file in that container and you will be modifying the different version of the file. This is called **COPY-ON-WRITE** mechanism.

**Volumes**



There are two types of volume mounting.

1. Volume Mount
2. Bind Mount

**Volume mount** : Here the volume is created inside the /var/lib/docker/volumes/data\_volume inside the docker HOST

**$** docker run -v data\_volume:/var/lib/mysql mysql

**Bind Mount:** Here the the volume is created in specified directory in Docker HOST.

**$** docker run -v /data/myvolume:/var/lib/mysql mysql

**Command to run a container**

$ docker container run –p 5000:5000 <application-name> :tag

OR

$ docker run –p 5000:5000 <application-name> :tag

**In detached mode:** **-d**

$ docker run –p 5000:5000 -d <application-name> :tag

**Stop**

$ docker container stop <Container-ID>

**Restart**

# What restart do is whenever you restart you docker-desktop this container will automatically start

$ docker run –p 5000:5000 -d --restart=always <application-name> :tag

$ docker run –p 5000:5000 -d --restart=no <application-name> :tag

**Prune**

# Prune will delete all the non-running containers

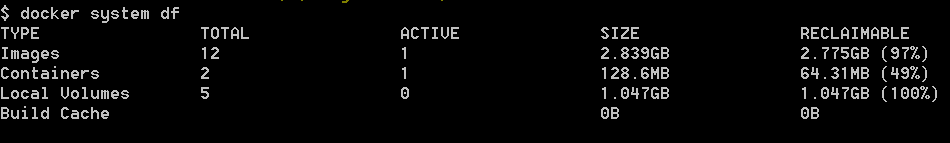
$ docker prune

**Show all containers**

$ docker container ls

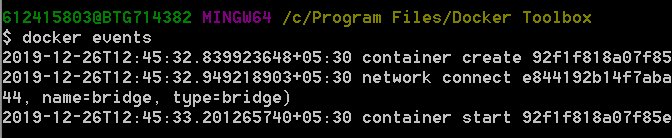
$ docker container ls –a : to ensure there are no stopped containers

**df**



**Events**

# Shows all the events



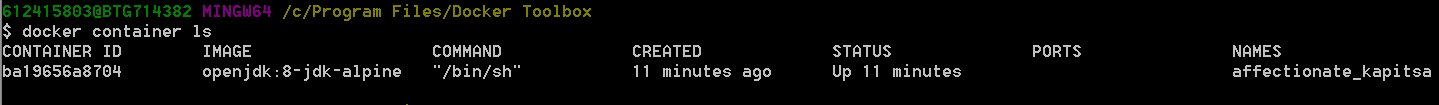
**Stats**

$ docker stats

Shows cpu utilization

**Openjdk**

$ docker run –dit openjdk:8-jdk-alpine



-d : detched

-i: --interactive

-t: --tty

-it : this combination allows you to run commands on running container.

Now, we want to run our spring boot app in this container

$ docker container cp target/docker-demo-0.0.1-SNAPSHOT.jar affectionate\_kapitsa:/tmp

**To see inside the openjdk container:**

$ docker container exec affectionate\_kapitsa ls /tmp

**Comimt this container so that it can be saved as an image:**

$ docker container commit affectionate\_kapitsa com/hello-world:manual1

This won’t run as we have not provided the start up command

$ docker container commit --change='CMD ["java","-jar","/tmp/docker-demo.jar"]' affectionate\_kapitsa com/docker-demo:manual1

$ docker container run –p 8080:8080 <Image-id-of-sp-boot-app>

If you are running docker engine, instead of localhost give docker-machine ip.

To find docker-engine ip

$ docker-machine ip

Above is the tedious and manual process.

We can use Dockerfile

FROM openjdk:8-jdk-alpine  
ADD target/docker-demo1.jar docker-demo1.jar  
ENTRYPOINT ["sh","-c","java -jar /docker-demo1.jar"]

We need to build the image now,

$ docker build -t some-random-name-of-image:optional-tag **.**

***PUSH***

$ docker push tanujt1/my-test-app:manual

**Docker uses cache internally i.e, next time when you will build if there is no change in Dockerfile it uses cache**



However, when you change the steps, till that step cache is used but from the new step cache is not used. For eg:

FROM openjdk:8-jdk-alpine  
EXPOSE 8080  
ADD target/docker-demo1.jar docker-demo1.jar  
ENTRYPOINT ["sh","-c","java -jar /docker-demo1.jar"]

Generalize Dockerfile

FROM openjdk:8-jdk-alpine  
EXPOSE 8080  
COPY target/\*.jar app.jar  
ENTRYPOINT ["sh","-c","java -jar /docker-demo1.jar"]

Using External Tomcat

FROM tomcat:8.0.51-jre8-alpine  
EXPOSE 8080

RUN rm –rf /usr/local/tomcat/webapps/\*  
COPY target/\*.war /usr/local/tomcat/webapps/ROOT.war  
ENTRYPOINT ["catalina.sh","run"]

**Spotify plugin to create docker images**

Whenever I run mvn clean package/install this plugin will automatically build the image. Hence we don’t have to build image manually after building the jar.

<plugin>  
 <groupId>com.spotify</groupId>  
 <artifactId>dockerfile-maven-plugin</artifactId>  
 <version>1.4.10</version>  
 <executions>  
 <execution>  
 <id>default</id>  
 <goals>  
 <goal>build</goal>  
 <!-- <goal>push</goal> -->  
 </goals>  
 </execution>  
 </executions>  
 <configuration>  
 <repository>com/${project.artifactId}</repository>  
 <tag>${project.version}</tag>  
 <skipDockerInfo>true</skipDockerInfo>  
 </configuration>  
</plugin>

**Improving cache of Docker Images by Adding Libraries in a Seperate**

Currently we have our jdk images which is downloaded and on the top of it we are copying(COPY) the Fat jar(i.e, our application Jar).

This Fat jar contains 2 different things:

1. Maven dependencies
2. Class files and the properties

**Microservices**

1. A microservice

* Dockerfile

**FROM** openjdk:8-jdk-alpine

**Volume** /tmp  
**EXPOSE** 8000  
**ADD** target**/**\*.jar app.jar

**ENV** JAVA\_OPTS=””  
**ENTRYPOINT** ["sh","-c","java $JAVA\_OPTS –Djava.security.egd=file:/dev/./urandom -jar /app.jar"]

If you have older version of spring boot you need to give -Djava.security.egd=file:/dev/./urandom

1. Currency Exchange: <http://localhost:8000>

* Rest Api: /currency-exchange/from/{from}/to/{to}

1. Currency Conversion

@FeignClient(name = "currency-exchange-service", url = "${CURRENCY\_EXCHANGE\_URI:http://localhost:8000}")  
//@FeignClient(name = "currency-exchange-service")  
//@FeignClient(name="netflix-zuul-api-gateway-server")  
public interface CurrencyExchangeServiceProxy {  
  
 @GetMapping("/currency-exchange/from/{from}/to/{to}")  
 // @GetMapping("/currency-exchange-service/currency-exchange/from/{from}/to/{to}")  
 public CurrencyConversionBean retrieveExchangeValue(@PathVariable("from") String from,  
 @PathVariable("to") String to);  
}

**# As we are not using Eureka, we have to give url in Feign Client**

**$CURRENCY\_EXCHANGE\_URI** is an environment variable.

After image build:

$ docker network create currency-network

$ docker run

-p 8000:8000

--network=currency-network

--name=currency-exchange-service

e6

$ docker run

-p 8100:8100

--network=currency-network

--name=currency-conversion-service

--env *CURRENCY\_EXCHANGE\_URI*=http://currency-exchange-service:8000

-d

0f

When one service(container) wants to call other service both have to be on the same network and the caller container should use the name of the container followed by the port it has exposed.

<http://currency-exchange-service:8000>, here currency-exchange-service is the name of the container and 8000 is the port it has exposed.

Other way with docker compose

Docker-compose.yml

**version**: '3.3'  
**services**:  
 **currency-exchange-service**:  
 **image**: in28min/currency-exchange-service:0.0.1-SNAPSHOT  
 **ports**:  
 - "8000:8000"  
 **restart**: always  
 **networks**:  
 - currency-compose-network  
  
 **currency-conversion-service**:  
 **image**: in28min/currency-conversion-service:0.0.1-SNAPSHOT  
 **ports**:  
 - "8100:8100"  
 **restart**: always  
 **environment**:  
 **CURRENCY\_EXCHANGE\_URI**: http://currency-exchange-service:8000  
 **depends\_on**:  
 - currency-exchange-service  
 **networks**:  
 - currency-compose-network  
   
*# Networks to be created to facilitate communication between containers***networks**:  
 **currency-compose-network**:

$ docker-compose up -d

**Eureka**

In this we don’t have to give the URL of other service, we can use the name of the Service and Feign with the help of Ribbon and eureka naming server will take care of calling other services.

//@FeignClient(name = "currency-exchange-service", url = "${CURRENCY\_EXCHANGE\_URI:http://localhost:8000}")  
@FeignClient(name = "currency-exchange-service")  
//@FeignClient(name="netflix-zuul-api-gateway-server")  
public interface CurrencyExchangeServiceProxy {  
  
 @GetMapping("/currency-exchange/from/{from}/to/{to}")  
 // @GetMapping("/currency-exchange-service/currency-exchange/from/{from}/to/{to}")  
 public CurrencyConversionBean retrieveExchangeValue(@PathVariable("from") String from,  
 @PathVariable("to") String to);  
  
  
}

eureka.client.service-url.defaultZone=http://naming-server:8761/eureka/

**version**: '3.3'  
**services**:  
 **naming-server**:  
 **image**: in28min/netflix-eureka-naming-server:0.0.1-SNAPSHOT  
 **ports**:  
 - "8761:8761"  
 **restart**: always  
 **networks**:  
 - currency-compose-network  
  
 **currency-exchange-service**:  
 **image**: in28min/currency-exchange-service:0.0.1-SNAPSHOT  
 **ports**:  
 - "8000:8000"  
 **restart**: always  
 **depends\_on**:  
 - naming-server  
 **networks**:  
 - currency-compose-network  
  
 **currency-conversion-service**:  
 **image**: in28min/currency-conversion-service:0.0.1-SNAPSHOT  
 **ports**:  
 - "8100:8100"  
 **restart**: always  
 **environment**:  
 **CURRENCY\_EXCHANGE\_URI**: http://currency-exchange-service:8000  
 **depends\_on**:  
 - naming-server  
 - currency-exchange-service  
 **networks**:  
 - currency-compose-network  
  
*# Networks to be created to facilitate communication between containers***networks**:  
 **currency-compose-network**:

API Gateways

eureka.client.service-url.defaultZone=http://naming-server:8761/eureka/

//@FeignClient(name = "currency-exchange-service", url = "${CURRENCY\_EXCHANGE\_URI:http://localhost:8000}")  
//@FeignClient(name = "currency-exchange-service")  
@FeignClient(name="netflix-zuul-api-gateway-server")  
public interface CurrencyExchangeServiceProxy {  
  
 //GetMapping("/currency-exchange/from/{from}/to/{to}")  
 @GetMapping("/currency-exchange-service/currency-exchange/from/{from}/to/{to}")  
 public CurrencyConversionBean retrieveExchangeValue(@PathVariable("from") String from,  
 @PathVariable("to") String to);  
  
  
}

Here, When currency-conversion-service uses feign client to call curreny exchange service, It uses netflix-zuul-api-gateway-server which goes to first part of URL i.e, "/currency-exchange-service”. Zuul asks Eureka the location of "/currency-exchange-service”.

**version**: '3.3'  
**services**:  
 **naming-server**:  
 **image**: in28min/netflix-eureka-naming-server:0.0.1-SNAPSHOT  
 **ports**:  
 - "8761:8761"  
 **restart**: always  
 **networks**:  
 - currency-compose-network  
  
 **zuul-api-gateway**:  
 **image**: in28min/netflix-zuul-api-gateway-server:0.0.1-SNAPSHOT  
 **ports**:  
 - "8765:8765"  
 **restart**: always  
 **depends\_on**:  
 - naming-server  
 **networks**:  
 - currency-compose-network  
  
 **currency-exchange-service**:  
 **image**: in28min/currency-exchange-service:0.0.1-SNAPSHOT  
 **ports**:  
 - "8000:8000"  
 **restart**: always  
 **depends\_on**:  
 - naming-server  
 **networks**:  
 - currency-compose-network  
  
 **currency-conversion-service**:  
 **image**: in28min/currency-conversion-service:0.0.1-SNAPSHOT  
 **ports**:  
 - "8100:8100"  
 **restart**: always  
 **environment**:  
 **CURRENCY\_EXCHANGE\_URI**: http://currency-exchange-service:8000  
 **depends\_on**:  
 - naming-server  
 - currency-exchange-service  
 **networks**:  
 - currency-compose-network  
  
*# Networks to be created to facilitate communication between containers***networks**:  
 **currency-compose-network**:

Zipkin

**version**: '3.3'  
**services**:  
 **rabbitmq**:  
 **image**: rabbitmq:3.5.3-management  
 **environment**:  
 **RABBIT\_URI**: amqp://guest:guest@rabbitmq:5672  
 **ports**:  
 - "5672:5672"  
 - "15672:1562"  
 **restart**: always  
 **networks**:  
 - currency-compose-network  
  
 **zipkin-server**:  
 **image**: openzipkin/zipkin  
 **container\_name**: zipkin  
 **environment**:  
 **STORAGE\_TYPE**: mem  
 **RABBIT\_URI**: amqp://guest:guest@rabbitmq:5672  
 **ports**:  
 - "9411:9411"  
 **restart**: always  
 **depends\_on**:  
 - rabbitmq  
 **networks**:  
 - currency-compose-network  
  
 **naming-server**:  
 **image**: in28min/netflix-eureka-naming-server:0.0.1-SNAPSHOT  
 **ports**:  
 - "8761:8761"  
 **restart**: always  
 **networks**:  
 - currency-compose-network  
  
 **zuul-api-gateway**:  
 **image**: in28min/netflix-zuul-api-gateway-server:0.0.1-SNAPSHOT  
 **ports**:  
 - "8765:8765"  
 **restart**: always  
 **depends\_on**:  
 - naming-server  
 - rabbitmq  
 - zipkin-server  
 **networks**:  
 - currency-compose-network  
  
 **currency-exchange-service**:  
 **image**: in28min/currency-exchange-service:0.0.1-SNAPSHOT  
 **environment**:  
 **RABBIT\_URI**: amqp://guest:guest@rabbitmq:5672  
 **ports**:  
 - "8000:8000"  
 **restart**: always  
 **depends\_on**:  
 - naming-server  
 - rabbitmq  
 - zipkin-server  
 **networks**:  
 - currency-compose-network  
  
 **currency-conversion-service**:  
 **image**: in28min/currency-conversion-service:0.0.1-SNAPSHOT  
 **ports**:  
 - "8100:8100"  
 **restart**: always  
 **environment**:  
 *# CURRENCY\_EXCHANGE\_URI: http://currency-exchange-service:8000* **RABBIT\_URI**: amqp://guest:guest@rabbitmq:5672  
 **depends\_on**:  
 - naming-server  
 - currency-exchange-service  
 **networks**:  
 - currency-compose-network  
  
*# Networks to be created to facilitate communication between containers***networks**:  
 **currency-compose-network**:

**III.** **Docker Course (Beginners)**