**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 willpackage the software along with all its dependencies and libraries and not worry about how it is deployed on what al platform.

**How it works?**

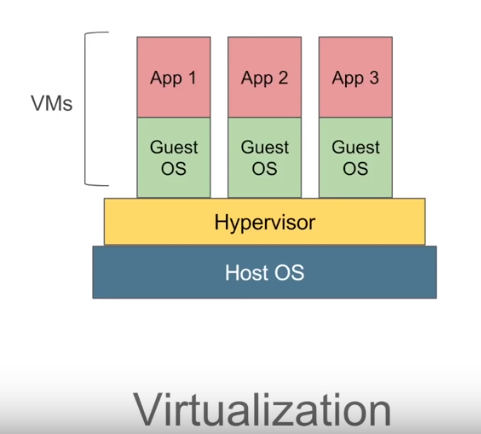
In general scenario, developer will define all the application and its dependencies in a file called **Dockerfile**. This Dockerfile will be used to create the docker image. So, it Docker image all the allpication and its dependencies id present. When you run the docker image you get docker containers.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

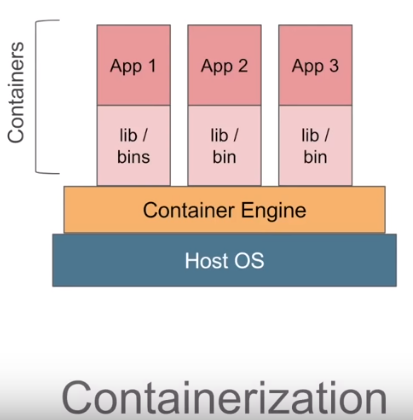
These images can be pulled to create containers in any environment.

**Containerization VS Virtualization**

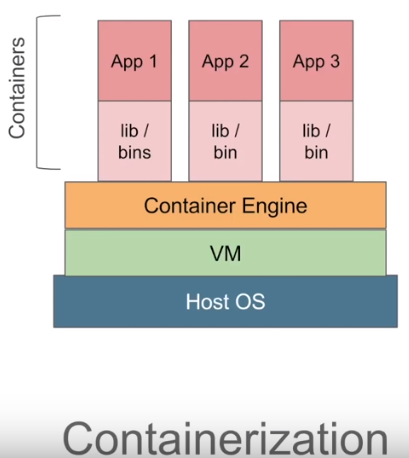
In virtualizatiuon, we have a software called Hypervisor which we used in our Host OS. Hypervisor is used to create and run virtual machines. Using Hypervisor, we can create multiple VM’s on the host OS. These virtual machine have their own OS i.e, it does not uses the host operating system. So, there 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 conatainer engine and we do not have separate OS but we have container which have application and all its dependencies. It will use the hist operating system unlike Virtualization which uses its own OS. Here, memory, space, other resources are not fixed i.e, they are taken dynamically so t 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, 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 ip 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.

**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.

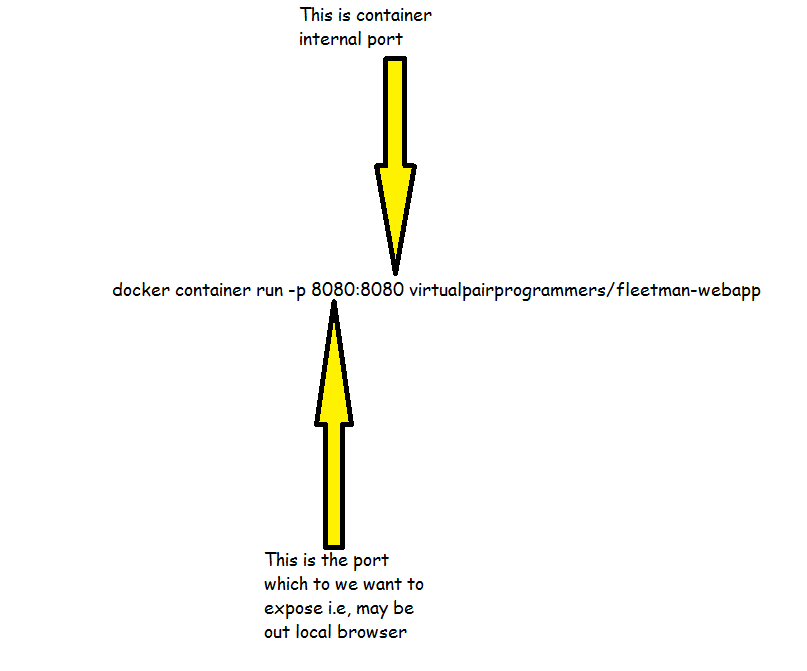
**Docker or Windows/Mac:** In this also Linux virtual machine is running, but you are using the native virtualization support from your HOST OS(Hyper-V on windows). So, it does not have to run Oracle Virtual box.

**Commands**

1. docker image pull <IMAGE NAME>

* download an image from DockerHub

1. docker image ls
2. docker-machine ip



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 |

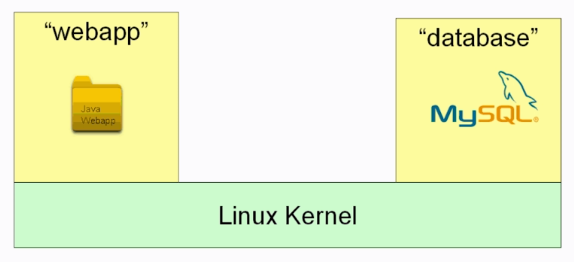
+------------------------------------------+

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

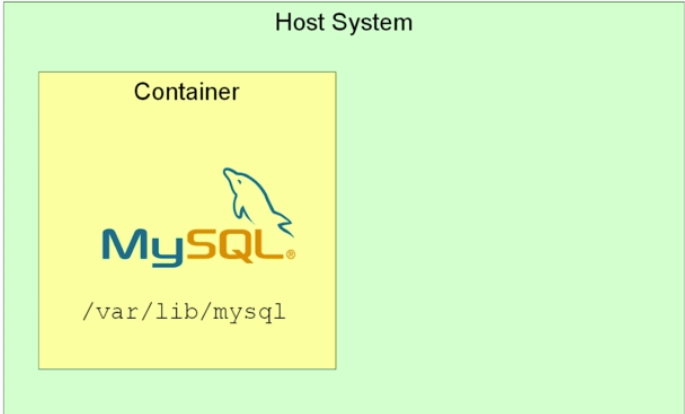
**Volumes**

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 ypu 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, we we build it 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 plughin

<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 expensice 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.*

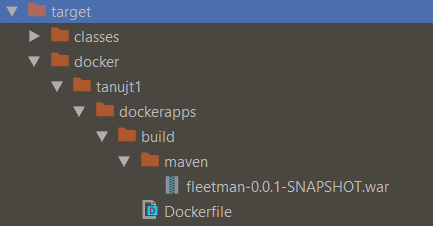
*We also needs to change*

**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*.