**Dockerfile**

* We can create customized images using docker file
* Dockerfile is simple text file, which uses predefined keywords for creating customized docker images
* Keywords in docker file is case sensitive and we need to use upper case only
* Whenever we are creating customized images we need to create using ***base image***
* From predefined images we need to build customized images
* Docker file contains set of keywords
* From dockerfile we will create customized images
* Filename should be dockerfile, we can customize the docker file name
* Dockerfile is file which will have instructions to create an image
* Docker will process these commands from top to bottom
* We will use docker DSL(Docker Specific Language) keywords in docker file
* Docker daemon will process these instructions from top to bottom while creating image.
* Dockerized any application we need ***dockerfile***
* If you have docker file with custom name using -f <fileName> while building docker image.

**docker build -f DockerfileMaven -t <imageName> .**

**Key words used in docker file (case sensitive )**

**FROM**

* Indicates ***“base image”*** on top and we can execute some instructions to create our own image
* Used to specify the base image from which the docker file has to be created.
* We can have more the one *“****FROM”*** instructions in docker file
* Each instructions in docker file we will create one layer

**#Syntax**

**FROM <Registry/Repository: tag>**

**#Specify a base image**

**FROM nginx**

**MAINTAINER logiclabs**

**Our Private repository own images**

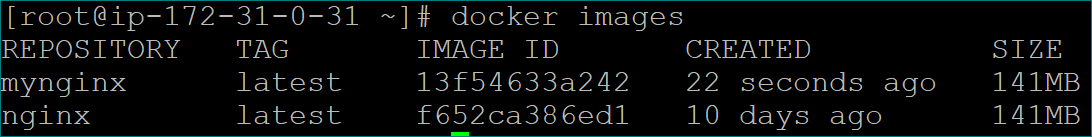
* FROM nexus.tcs.com/tomcat:8.5.54
* FROM 458430697982.dkr.ecr.ap-south-1.amazonaws.com/tomcat:8.5.54
* FROM gcr.io/tomcat:8.5.54

**To Build image from dockerfile using below command**

* docker build -t <Image\_Name> .
* docker build -t mynginx .
* docker build .

**To run image**

* docker run <Repository/Registry:tag> (OR) docker run <Image\_Id>
* docker run mynginx:latest

****

**COPY**

* Is used to copy local files/folders from build context (From Server where we are creating image) to the image.
* The ***COPY*** instruction copies files or directories into the Docker image.
* It takes a source and destination as arguments
* Source can be absolute or relative from current WORKDIR or wild cards.
* Destination path can be absolute or relative to current WORKDIR
* **Source:** The file/folder path from build context
* **Destination:** The path (image path) in which you what to copy

***Syntax:***

***COPY <source> <destination>***

***COPY . .***

***COPY target/maven-web-ap\*war /usr/local/tomcat/webapps/maven-web-application.war***

***COPY devops.pem /root/folder/***

***COPY ./requirement.txt /app/requirement.txt***

***COPY package.json package-lock.json /app***

***COPY package\*.json /app***

***COPY . /app***

**ADD**

* ADD can copy local files and also *it can download/copy files from remote https/endpoint locations while creating an image.*
* If we are adding or downloading tar files (\*.tar) it will add and also it will extract that file in image
* The ADD instruction copies files, directories, remote file or tar archive into the Docker image.
* Extension should be .tar only(if some other extension like *(****.tar.gz/.zip****) it wont extract in destination path)*
* It takes a source and destination as arguments.
* Source can be files and directories.
* Source can be a URL. The ADD instruction will download the file from the URL and save it to the destination. We don't need to use curl or wget to download a file.
* Source can be a local *(****.tar)*** archive. The ADD instruction will automatically extract it to the destination. We don't need to run unachieved commands manually.
* use ADD when you want download a file from a URL or extract local archive file.

***Syntax:***

***ADD <URL> <DESTINATION-PATH>***

***ADD <SOURCE> <DESTINATION-PATH>***

***ADD*** [***https://dlcdn.apache.org/tomcat/tomcat-9/v9.0.56/bin/apache-tomcat-9.0.56-deployer.tar.gz***](https://dlcdn.apache.org/tomcat/tomcat-9/v9.0.56/bin/apache-tomcat-9.0.56-deployer.tar.gz) ***.***

***ADD*** [***https://dlcdn.apache.org/tomcat/tomcat-9/v9.0.56/bin/apache-tomcat-9.0.56-deployer.tar.gz***](https://dlcdn.apache.org/tomcat/tomcat-9/v9.0.56/bin/apache-tomcat-9.0.56-deployer.tar.gz) ***/opt/***

***ADD ./example.tar.gz /tmp/***

***ADD example.txt /tmp/***

* Here, ADD command its copied and extracted where COPY only do copy the files
* If you want to extract the tar file need to add one more extra command in Dockerfile

**Create tar file:**

*tar -cvzf sample.tar.gz ./*

**Dockerfile:**

***FROM busybox***

***COPY sample.tar.gz /tmp/***

***WORKDIR /tmp/***

***RUN tar -xvzf sample.tar.gz . && rm -rf sample.tar.gz***

***CMD ["sh"]***

**Build Image: docker build -t imagename .**

**Run image : docker run -it imagename sh**

* One more note worthily difference between ADD and COPY is that COPY has the **from=<name|index>** flag that lets you copy files from a previous build stage in a multi-stage build.
* ADD does not have this option

**Without using COPY and ADD command:**

***FROM ubuntu***

***WORKDIR /tmp/***

***RUN apt-get update && apt-get install wget -y && wget https://dlcdn.apache.org/tomcat/tomcat-9/v9.0.56/bin/apache-tomcat-9.0.56-deployer.tar.gz && tar xvf apache-tomcat-9.0.56-deployer.tar.gz && rm -rf apache-tomcat-9.0.56-deployer.tar.gz***

***CMD ["sh"]***

**RUN**

* Using RUN we can executes commands on top of base image
* RUN instructions will be executed while creating(building) an image itself
* **Using RUN instructions we can configure the software while creating image**
* We can have **‘n’** number of run instructions in ***“Dockerfile”*** all instructions will be executed ***top to bottom order***
* Used for running Linux commands within the container. It is generally helpful for installing the software in the container.
* **Usecase**: while creating image if we want to install any software within image in this care we will use RUN command.
* **Difference between docker run and RUN** :

**docker run:**

* Using this command we will create the container

**RUN:**

* This is dockerfile instruction command using RUN instruction we run/execute some command on top of previous layers (Image)
* RUN instruction will be executed while creating image.

**Syntax:**

**RUN <command> <arguments>**

FROM ubuntu

MAINTAINER logiclabs

**RUN apt-get update**

**RUN apt-get install -y git**

**RUN useradd Balaji**

**RUN mkdir -p /opt/tomcat**

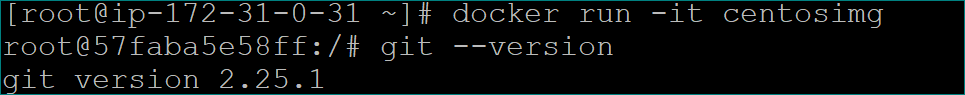
docker build -it --rm <image\_name> .

**docker build -it centosimg**

**docker build -it --rm centosimg**

[

**docker run -it centosimg**



**View specific IP Address instead of all**

**docker run -p 127.0.0.1:5050:80 nginx**

**All RUN instructions will be executed while building the image**

**FROM centos**

**1.RUN yum install git -y**

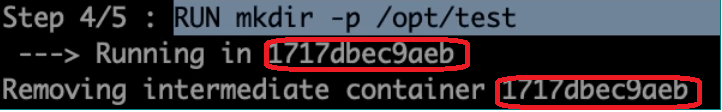
**2.RUN echo "Welcome From RUN one"**

**3.RUN mkdir -p /opt/test**

**4.RUN echo "Welcome From RUN two"**

**#docker build -t imageone .**

* If we observed, each instruction we have ***running in* "1717abec9aeb"** it creating some intermediate container to create that layer
* Once the layer has been created, Removing intermediate container
* Removing intermediate container ***"1717abec9aeb"***
* This is applicable for each run instructions.



**Docker has a concept of build cache:**

**docker build -t image .**

* If run above command again, if there are no changes then same layers will be created again (i.e if no changes in docker instruction file) with the help of build cache.
* Each instruction in a docker file going to form a layer, if that layer is already processed and no change in the instruction or layer it is not going to create that layer again.
* Using build cache, improve the build time because first time it took some time and second time it has created quickly image using ***BuildCahe concept***

**Case-1:**

If we change some instruction in docker file, line number "1" “*RUN yum install git wget –y”* in this case all layers will be processed again because this is first instruction in Dockerfile.

**Case-2:**

if we change some instruction in docker file, line number "4" *“RUN echo Welcome From RUN 2"* in this case only line "4" alone will processed again remaining all will use build cache alone because this is last instruction in Dockerfile.

**Option to rebuild Dockerfile with/without changes:**

Docker RUN instruction will be build again eventhrought no changes layers will be created again because option ***--no-cache (i.e it will not use the BuildCahe)***

docker build -t imageone --no-cache .

* Recommended to use BuildCahe
* Using this we can save the build time

**CMD**

* Using **CMD** we can execute commands
* This is used to specify the initial command that should be executed when the container starts.
* CMD instructions will be executed while starting a container
* **USECASE:** CMD command can be used to ***START*** the process (Application) while creating a container
* ***We can have more than one CMD instructions in dockerfile in this case docker will be process or execute only one (i.e last instruction alone)***
* When we starting a container CMD command will be executed.

FROM centos

MAINTAINER loginlabs

**CMD ["date"]**

**CMD ["ls","-a"]**

**CMD ["echo","Welcome From CMD"]**

docker build -t imgone -f **Dockerfile\_CMD** .

**docker run imgone**



We can inspect the image and verify of CMD command instruction

docker inspect <image-name>

**ENTRYPOINT**

* **ENTRYPOINT** instructions will be executed while creating/staring a container.
* We can set an entrypoint (command) for our container while we want to execute
* ENTRYPOINT also used to start the process (application) inside a container
* If we have more than one **ENTRYPOINT** then it will execute latest one
* It can also be used for accepting arguments from the CMD instruction.

**Difference between CMD and ENTRYPOINT:**

* **CMD:** command can be overridden(passing at runtime) while creating a container
* **ENTRYPOINT:** command can’t be overridden while creating a container and **it will take as argument.**
* **[RUN,CMD,ENTRYPOINT ]** can be written in 2 forms (i.e shell form and executable form)
* Order doesn’t not matter.

**Syntax:**

**ENTRYPOINT [“command”,”arguments”]**

**ENTRYPOINT [“echo”,”Welcome To ENTRYPOINT”]**

FROM centos

RUN yum install curl -y

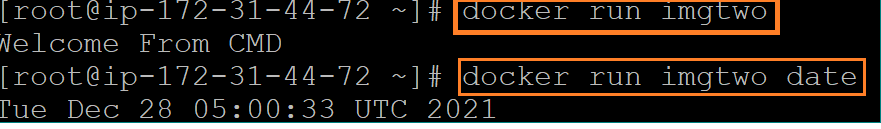
RUN echo "Welcome to Run one"

CMD ["date"]

CMD ["echo","Welcome From CMD"]

**Override CMD at RUNTIME:**

**docker build -t imgtwo -f Dockerfile\_ENTPNT .**

****

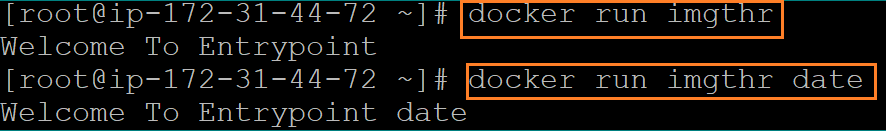
FROM centos

RUN yum install curl -y

RUN echo "Welcome to Run one"

**ENTRYPONT ["echo","Welcome From ENTRYPOINT"]**

**docker build -t imgthr -f Dockerfile\_ENTPNT .**

****

**Note:**

* + We can’t override while creating container
  + If we pass anything at runtime for ENTRYPONT, it will consider as argument to the ENTRYPONT

**If we have both CMD and ENTRYPOINT**

**FROM centos**

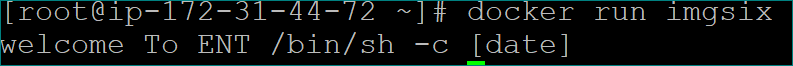
**CMD [date]**

**ENTRYPOINT ["echo","welcome To ENT"]**

**Build image:**

**docker build -t imgname -f Dockerfile\_cust .**

**docker run imgsix**

****

**Note:** Date is taken as argument to the ENTRYPOINT

**Use Case:** Always i want to execute some command but i should have a control on arguments

sh catalina.sh run

sh catalina.sh start

CMD ["run"]

ENTRYPOINT ["sh","catalina.sh"]

sh catalina.sh run **//output**

**#If ENTRYPOINT in Dockerfile**

**docker run <image-name> <arugument>**

**docker run image start**

sh catalina.sh **start** **//start will not consider as command it will argument to the ENTRYPOINT**

**CMD**

**Example:**

**#Dockerfile**

**FROM ubuntu**

**MAINTAINER MithunTech devops@gmail.com**

**RUN apt-get update -y**

**RUN apt-get install apt-utils -y**

**RUN apt-get install git -y**

**RUN echo "Welcome to run One"**

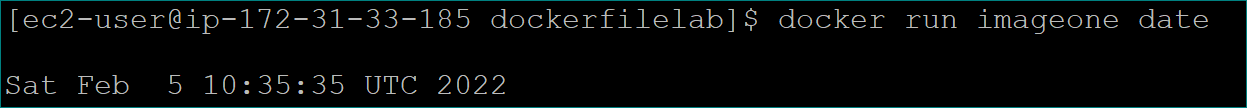
**RUN mkdir -p /tmp/dir1**

**RUN echo "Welcome to run Two"**

**CMD ["echo","Welcome To CMD command"] //Replaced with ENTRYPONT instead of CMD**

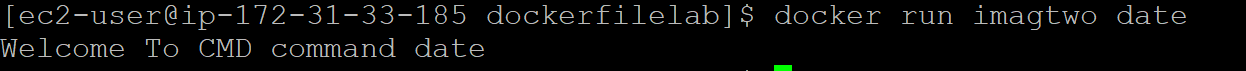
**#docker build -t imageone .**

**#docker run imageone date**

****

**ENTRYPOINT:**

**#docker run imagetwo date**

****

* **If we have multiple CMD and ENTRYPOIT in side Dockerfile, latest instruction will be execute**

**Combination of both CMD and ENTRYPOINT in Dockerfile:**

**#Dockerfile\_CMDENT**

**FROM ubuntu**

**CMD ["pwd"]**

**CMD ["date"]**

**ENTRYPOINT ["echo","WelcomeTo Entrypoint"]**

**Build image from Dockerfile:**

**#docker build -t <image\_name> -f <custome\_dockerfile\_name> .**

**#docker build -t imgone -f Dockerfile\_CMDENT .**

**#Run image**

**#docker run imgone**

**WelcomeTo Entrypoint date**

**Note: date consider as argument to the ENTRYPOINT**

**#Run image**

**#docker run imgone pwd**

**WelcomeTo Entrypoint pwd**

**Note:**

**ENTRYPOINT will be overridden the argument**

**FROM ubuntu**

**ENTRYPOINT ["echo","WelcomeTo Entrypoint"]**

**CMD ["date"]**

**CMD ["pwd"]**

**#docker build -t imgone -f Dockerfile\_CMDENT .**

**#docker run imgone**

**WelcomeTo Entrypoint pwd**

**Note:**

**We can fallow any order (CMD first and ENTRYPOINT next or ENTYPPONT first and CMD next]**

**USECASE :**

**[RUN CMD ENTRYPOINT]** can be written in 2 forms

**1.Shell form**

**2.Executable form**

**SHELL FORM:**

**Process is running as a sub process and process will not notify.**

**RUN <command> <args>**

**Example:**

**RUN echo "Welcome From RUN"**

**RUN mkdir -p /opt/test**

**RUN yum install git curl -y**

**CMD <command> <args>**

**CMD echo "Welcome From CMD"**

**CMD java -jar app.jar**

**CMD node app.js**

**ENTRYPOINT <command> <args>**

**ENTRYPOINT echo "Welcome From CMD"**

**ENTRYPOINT java -jar app.jar**

**ENTRYPOINT node app.js**

**SHELL FORM internally will execute like below**

**CMD echo "Welcome From CMD"**

**/bin/bash –c echo "Welcome From CMD"**

**EXECUTABLE FORM:**

**Process is running as a parent process and process will be notified.**

**Recommended way to run executable form**

**[CMD, ENTRYPOINT] recommended way to write EXECUTABLE form**

**RUN ["<executable>","<arg1>","<arg2">]**

**RUN [“echo”,”Welcome To RUN”]**

**RUN ["mkdir","-p","/opt/test"]**

**RUN ["yum","install","git","curl","-y"]**

**CMD ["<executable>","<arg1>","<arg2">]**

**CMD ["mkdir","-p","/opt/test"]**

**CMD ["yum","install","git","curl","-y"]**

**ENTRYPOINT ["<executable>","<arg1>","<arg2">]**

**ENTRYPOINT ["mkdir","-p","/opt/test"]**

**ENTRYPOINT ["yum","install","git","curl","-y"]**

**EXECUTABLE FORM internally will execute like below**

**RUN [“echo”,”Welcome To RUN”]**

**/bin/echo Welcome To RUN**

**ENV**

* We can set environment variable using **ENV** these variables can be accessible in image while creating **image** and also we can access in container.
* Used for specifying the environment variables that should be passed to the container.
* Environment variable will be available inside the container
* Environment variable will be passing while creating container or we can define within docker file and while building image we can’t pass the environment variable.
* Instead of hard coding username and password sensitive details in docker file instead of this (i.e which is going to change env to env) we can pass this env variable run time

**Syntax:**

**ENV <KEY> <VALUE>**

FROM ubuntu as UbuntuOS

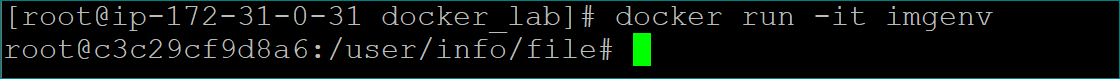
**ENV CATALINA\_HOME /user/info/file**

RUN mkdir -p "$CATALINA\_HOME"

**WORKDIR $CATALINA\_HOME**

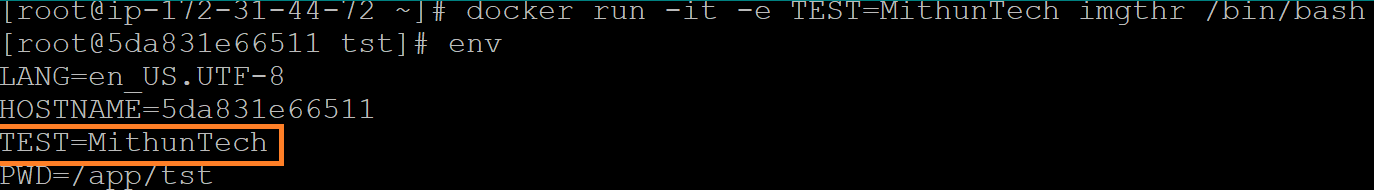
**docker images**

**docker run –it <image\_name>**

****

**docker inspect <image\_name>**

**docker run -it -e TEST=MithunTech imgthr /bin/bash #passing environment variable at runtime**

****

**We can pass the multiple environment variables at runtime**

**docker run -it -e TEST=mithuntech -e DB\_USERNAE=super imgthr /bin/bash**

**LABEL**

* **LABLES** are **key value pair** , which we can add to the image
* We can add additional information to the image
* LABLES are Metadata(i.e data about data) to the image

FROM tomee

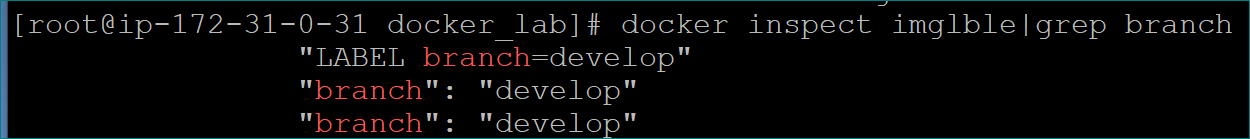
**LABEL branch develop**

**LABEL application discoverdsl**

**LABEL team devops**

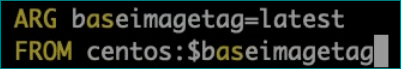
**docker inspect <REPOSITORY/IMAGE\_ID>**

**docker inspect imglble|grep branch**

****

**ARG**

* We can use ARG to define variables
* we can access ARGS with in Dockerfile and we can define key value pairs using ARG
* We can define default value and if required we can override while building the image
* ARG is only one instructions where we can define before **FROM(i.e parameterized)**



* If we want to **pass argument while creating image** to dockerfile
* We can pass runtime as well, while creating image
* **Is like variable which we can define and refer in dockerfile at runtime(while creating an image) also we can pass arguments**
* **Argument(ARG)** can be accessible only while creating an image
* **We cannot refer/access in side container**

FROM tomee

ARG branch=develop

**LABEL branchname $branch**

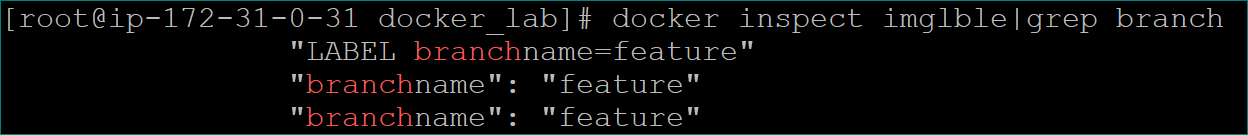
* At runtime (while creating an image) also we can pass argument**(FIRST PRORITY)**
* We can define default value for ARG and if required we can override.
* We can access this ARG within docker file
* Only instruction which is allowed before FROM is ARG
* Using ARG we can parameterized
* **We can pass ‘n’ number of arguments**

**Syntax:**

**ARG <name> <value>**

**docker build -t imglble --build-arg branch=feature .**

**docker build -t imglble --build-arg branch=feature .**

****

FROM tomee

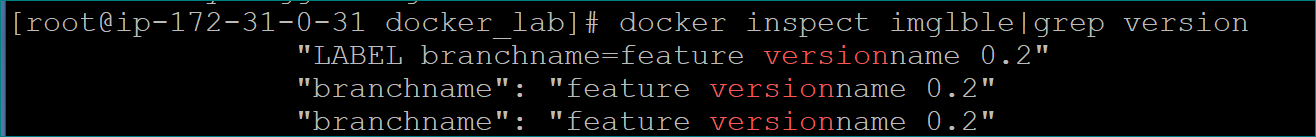
ARG branch=develop

**ARG version=0.1**

**LABEL branchname $branch**

**V**

**docker\_lab]# docker build -t imglble --build-arg branch=feature --build-arg version=0.2**

****

**Difference between ENV and ARG ?**

**ENV:**

Environment variable we can access and refer inside the container

**We** can pass the environment variable while creating container not possible while creating image.

**ARG:**

arguments we can’t access inside the container

we can pass the args while creating a images itself

**LABEL:** Label we can’t access inside the container

FROM tomee

ARG branch=develop

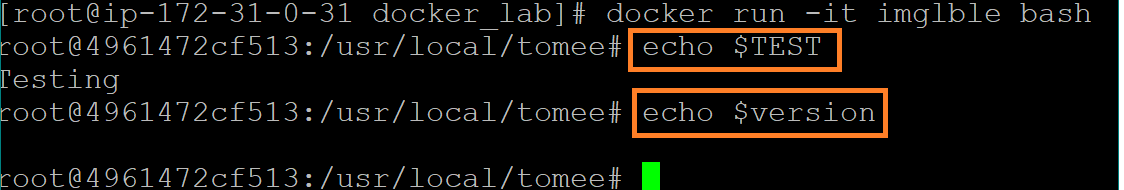
**ENV TEST TestingENV**

**ARG version=0.1**

**LABEL branchname $branch**

**docker run -it < REPOSITORY/IMAGE\_ID> bash**

**docker run -it imglble bash**

****

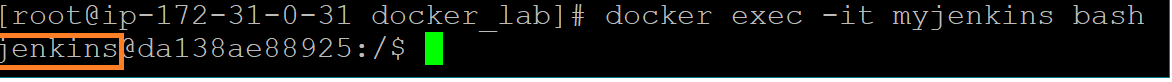
**USER**

* We can set an **USER** for an image or container
* The instructions will be executed as whatever user we have set using this **USER**
* Used to specify the default user who should login into the container.
* Real time not recommended to run as root user, it’s should be normal user
* **By default container will start root user(but real time it’s not recommended to run as root user)**
* **If containers are running as root user there may be chance to access host also and lot of staff in side container that why not recommended to run as root user it should be normal user(i.e non-root user)**

**docker pull Jenkins/Jenkins**

**docker run --name myjenkins -d -p 9090:8080 jenkins/Jenkins**

**docker exec -it myjenkins bash**



**EXPOSE**

* Used to specify the internal port of the container
* **EXPOSE** indicates on which part our container is listening. It’s like documentation using this we can understand what port we are using.

**git clone** https://github.com/Venki22/nodejs-app-mss.git

**Dockerfile:**

FROM node:12

RUN mkdir -p /usr/app

WORKDIR /usr/app

COPY . .

RUN npm install

CMD ["node","app.js"]

docker build -t nodejsapp .

docker login -u <user\_name>

Eneter password of hub.doceker

docker push venki22/nodejsapp:1

docker run –name nodejsapp-1 -p 8181:9981 venki22/nodejsap:1

**MAINTAINER**

* This represents name of the organization or the author who created this dockerfile
* Using MAINTAINER we can define who is maintainer(author) of docker file image
* It is just documentation purpose

**Syntax:**

* MAINTAINER mithun Technologes <devopstraining@gmail.com)

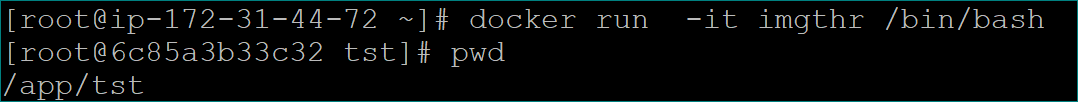
**WORKDIR**

* Used to specify default working directory in the container
* We can set a working directory using for image or container. All subsequent instructions will be performed/executed in that directory.
* Default working directory is root (/)

FROM centos

WORKDIR /app/tst

**docker run exec -it imgthr /bin/bash**

****

**VOLUME**

* Used to specify the default volume that should be attached to the container.
* We can set the mount point

**STOPSIGNAL**

* Used to specify the key sequences that have to be passed in order to stop the container.

**Multi Stage Dockerfile**

* Multi-stage Docker builds let you write Dockerfile with multiple FROM statements. This means you can create images which derive from several bases, which can help cut the size of your final build.
* Docker images are created by selecting a base image using the FROM statement. You then add layers to that image by adding commands to your Dockerfile.
* With multi-stage builds, you can split your Dockerfile into multiple sections. Each stage has its own FROM statement so you can involve more than one image in your builds. Stages are built sequentially and can reference their predecessors, so you can copy the output of one layer into the next.

For Example

***#Maven***

***FROM maven:3.5-jdk-8-alpine as build***

***WORKDIR /app***

***COPY . .***

***RUN mvn install***

***#Tomcat***

***FROM tomcat:8.0.20-jre8***

***COPY --from=build /app/target/maven-web-application\*.war /usr/local/tomcat/webapps/maven-web-application.war***

Create a dockerfile by taking nginx as the base image

and specify the maintainer as logiclabs. Construct an image from the dockerfile.

Creating customized docker images by using docker file.

$ sudo su -

dockerfile.sh

# **vim dockerfile**

FROM nginx

MAINTAINER logiclabs

**To build an image from the dockerfile**

**# docker build -t mynginx .**

**Note:**

t 🡪 stands for tag,

. 🡪 Stands for current working dir

mynginx 🡪 is the new image name

**To see the image**

# docker images

# **vim dockerfile**

FROM nginx

MAINTAINER logiclabs

CMD [“date”]

**Build image using below command**

docker build –t <image allies name> .

t 🡪 Tagging

. 🡪 Current directory

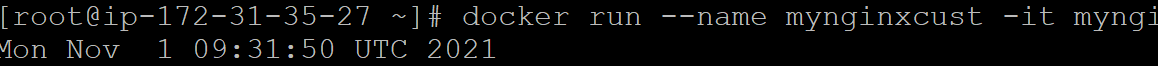
**CMD** 🡪 Whenever container start command should be executed automatically, then we have to configure under CMD

When the container starts CMD instructions will be executed.

**#To run container using image**

docker run –it

**# Display date using docker file**



**Multiple commands in DOCKER FILE:**

Whenever we write multiple commands in dock file, the latest file will be executed.

*# vim dockerfile*

*FROM centos*

*MAINTAINER logiclabs*

*CMD ["date"]*

*CMD ["ls", "-la"]*

# **docker build -t mycentos .**

# docker run -it mycentos

**Observation**, we get ls -la output

***RUN 🡪*** *RUN instructions will be executed when building the images*

*#vim dockerfile*

*FROM ubuntu*

*MAINTAINER ossoftware*

*RUN apt-get update*

*RUN apt-get install -y git*

***# docker build -t myubuntu .***

*Lets see the images list and space consumed by our image*

*# docker images*

*# docker run -it myubuntu*

*# git --version*

*# exit*