What Is container?

* A self-contained scaled unit of software
* Contains everything required to run the code
* Include batteries and operating systems
* A container includes code, configs and processes
* Networking for communication with other containers which they are supposed to talk to
* All the dependencies that the system needs, bundled up in that container
* Operating system to run your code

Docker is a client program

It’s a command you type at the terminal

A server program that listens for messages from that command and manages a running linux system

Docker has a program that builds containers from code

It takes your code along with its dependencies and bundles it up and seals it in a container

It’s a service that distributes these containers across the internet, so that you can find others work and right people can find your work.

A container image is lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings.

Docker is a tool that can package an application and its dependencies in a virtual container that can run on any host using the linux kernel (real or emulated)

It’s a containerization technology which create light weight images, resource isolation and other features

Application and dependencies of your app which are packaged into the container are independent of anything present in other container, even If they are running on the same host.

As a result you can have multiple docker container running on the same host which are truly isolated from each other.

Docker was developed by dotcloud, a paas company

Docker was open source in 2013

What does docker compose take as its input for creating containers

YAML Manifest file

Not a Part of docker architecture : host os userspace

Docker for Mac uses virtual box to provision VM for virtualization on mac OS is **False statement**

* The Docker client contacted the Docker daemon.
* The Docker daemon pulled the "hello-world" image from the Docker Hub.
* The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
* The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

**Docker Architecture** comprises of docker daemon, docker client, registries and objects which are building blocks of docker

**Docker Daemon**

**Docker Daemon** whose main job is to manage docker objects such as images, containers, networks and volumes. It listens for docker API requests and manages docker objects.

Its job is to communicate with other docker daemon to manage docker services.

**Docker Client**

**This is the point of interaction between user and docker**

This is the primary way that many docker users interact with docker

Docker client will take the commands coming from the user, and translate them to an api call, which is then sent over to docker daemon.

In short

Docker client sends docker run command to docker daemon

Docker client can communicate with more than one docker daemon which is useful when we are dealing with docker swarm which will contain cluster of nodes.

**Docker Registries**

Docker registries serves as stores for docker images. Docker images are essentially templates on which docker containers are built. These templates are stored in docker registries.

Docker hub and Docker cloud are public registries.

Docker trusted registry is a part of docker enterprise edition.

Inorder to upload and download images from docker registries we will make use of docker push and docker push and pull command. Even docker run command can be used to pull images from docker registry.

Docker also have docker store where one can buy and sell images and even distribute them for free.

**Docker Objects**

**Images, containers, volumes, services**

An image is essentially a template from which a container can be built. It is read only template.

You always start from a base image and then you add some customization of your own and that becomes your custom image.

An image is based on another image, with some additional customization.

For example a base image can be Ubuntu OS and then you could install http on it, along with a simple web page and that becomes your custom image.

You can create your own images, or use other images which are created by other people and are available in public registry.

The easiest way to create your own image is to use docker file.

Docker file contains sequence of instructions and in docker terminology each of those instruction create layer in the docker image.

**------------------------------**

**Docker file**

Docker file are text representation of your docker images.

Docker file defines what goes on in the environment inside your container.

This should include resources such as what networking interfaces should your container use.

What are the disk drive with the app running your container should have access to [something fishy to understand].

If the app or microservice running within your container needs to access the outside world through the use of ports, then you will need to mention those ports in the docker file.

Also if there are files which need to be copied into your container and these could include source code for your application or configuration file or other tools which your applications needs installed, then you could mention that in your docker file.

Next if you have copied your source code into the container, then you can also define build of your application, within the docker file.

**A container**

**Is an runnable instance of an image. While your image contains sets of instructions of what needs to be done, the container is what goes ahead and does it.**

**You can create, start, stop, or delete a container using docker CLI or the docker API.**

When we initially create a container from docker image, the create container is in **stopped state**. In order to start up this container we run the command **docker start**. This gives us running container.

If you want to move your image **directly into running container**, use **docker run** command.

To stop this running container, simply run **docker stop command.**

| **Rule** | **Behavior** |
| --- | --- |
| # comment | Ignored. |
| **\*/temp\*** | Exclude files and directories whose names start with temp in any immediate subdirectory of the root. For example, the plain file /somedir/temporary.txt is excluded, as is the directory /somedir/temp. |
| \*/\*/temp\* | Exclude files and directories starting with temp from any subdirectory that is two levels below the root. For example, /somedir/subdir/temporary.txt is excluded. |
| temp? | Exclude files and directories in the root directory whose names are a one-character extension of temp. For example, /tempa and /tempb are excluded. |

Lines starting with ! (exclamation mark) can be used to make exceptions to exclusions. The following is an example .dockerignore file that uses this mechanism:

If you want to ignore all markdown files you can use \*.md and incase if you don’t want any specific file not to be ignored you can use ! symbol **[example: !README.md].**

\*.md

!README.md

The placement of **!**exception rules influences the behavior: the last line of the .dockerignore that matches a particular file determines whether it is included or excluded. Consider the following **example:**

\*.md

!README\*.md

README-secret.md

No markdown files are included in the context **except README files other than README-secret.md.**

Now consider this example:

\*.md

README-secret.md

!README\*.md

All of the README files are included. The middle line has no effect because !README\*.md matches README-secret.md and comes last.

escape

# escape=\ (backslash)

Or

# escape=` (backtick)

The escape directive sets the character used to escape characters in a Dockerfile. If not specified, the default escape character is \.

The escape character is used both to escape characters in a line, and to escape a newline. This allows a Dockerfile instruction to span multiple lines. Note that regardless of whether the escape parser directive is included in a Dockerfile, *escaping is not performed in a RUN command, except at the end of a line.*

Setting the escape character to ` is especially useful on Windows, where \ is the directory path separator. ` is consistent with [Windows PowerShell](https://technet.microsoft.com/en-us/library/hh847755.aspx).

Consider the following example which would fail in a non-obvious way on Windows. The second \ at the end of the second line would be interpreted as an escape for the newline, instead of a target of the escape from the first \. Similarly, the \ at the end of the third line would, assuming it was actually handled as an instruction, cause it be treated as a line continuation. The result of this dockerfile is that second and third lines are considered a single instruction:

FROM microsoft/nanoserver

COPY testfile.txt c:\\

RUN dir c:\

Results in:

PS C:\John> docker build -t cmd .

Sending build context to Docker daemon 3.072 kB

Step 1/2 : FROM microsoft/nanoserver

---> 22738ff49c6d

Step 2/2 : COPY testfile.txt c:\RUN dir c:

GetFileAttributesEx c:RUN: The system cannot find the file specified.

PS C:\John>

One solution to the above would be to use / as the target of both the COPY instruction, and dir. However, this syntax is, at best, confusing as it is not natural for paths on Windows, and at worst, error prone as not all commands on Windows support /as the path separator.

By adding the escape parser directive, the following Dockerfile succeeds as expected with the use of natural platform semantics for file paths on Windows:

# escape=`

FROM microsoft/nanoserver

COPY testfile.txt c:\

RUN dir c:\

Results in:

PS C:\John> docker build -t succeeds --no-cache=true .

Sending build context to Docker daemon 3.072 kB

Step 1/3 : FROM microsoft/nanoserver

---> 22738ff49c6d

Step 2/3 : COPY testfile.txt c:\

---> 96655de338de

Removing intermediate container 4db9acbb1682

Step 3/3 : RUN dir c:\

---> Running in a2c157f842f5

Volume in drive C has no label.

Volume Serial Number is 7E6D-E0F7

Directory of c:\

10/05/2016 05:04 PM 1,894 License.txt

10/05/2016 02:22 PM <DIR> Program Files

10/05/2016 02:14 PM <DIR> Program Files (x86)

10/28/2016 11:18 AM 62 testfile.txt

10/28/2016 11:20 AM <DIR> Users

10/28/2016 11:20 AM <DIR> Windows

2 File(s) 1,956 bytes

4 Dir(s) 21,259,096,064 bytes free

---> 01c7f3bef04f

Removing intermediate container a2c157f842f5

Successfully built 01c7f3bef04f

PS C:\John>

Environment replacement

Environment variables (declared with [the ENV statement](https://docs.docker.com/engine/reference/builder/#env)) can also be used in certain instructions as variables to be interpreted by the Dockerfile. Escapes are also handled for including variable-like syntax into a statement literally.

Environment variables are notated in the Dockerfile either with $variable\_name or ${variable\_name}. They are treated equivalently and the brace syntax is typically used to address issues with variable names with no whitespace, like ${foo}\_bar.

The ${variable\_name} syntax also supports a few of the standard bash modifiers as specified below:

* ${variable:-word} indicates that if variable is set then the result will be that value. If variable is not set then wordwill be the result.
* ${variable:+word} indicates that if variable is set then word will be the result, otherwise the result is the empty string.

In all cases, word can be any string, including additional environment variables.

Escaping is possible by adding a \ before the variable: \$foo or \${foo}, for example, will translate to $foo and ${foo}literals respectively.

Example (parsed representation is displayed after the #):

FROM busybox

ENV foo /bar

WORKDIR ${foo} # WORKDIR /bar

ADD . $foo # ADD . /bar

COPY \$foo /quux # COPY $foo /quux

Environment variables are supported by the following list of instructions in the Dockerfile:

* ADD
* COPY
* ENV
* EXPOSE
* FROM
* LABEL
* STOPSIGNAL
* USER
* VOLUME
* WORKDIR

as well as:

* ONBUILD (when combined with one of the supported instructions above)

**Note**: prior to 1.4, ONBUILD instructions did **NOT** support environment variable, even when combined with any of the instructions listed above.

Environment variable substitution will use the same value for each variable throughout the entire instruction. In other words, in this example:

ENV abc=hello

ENV abc=bye def=$abc

ENV ghi=$abc

will result in def having a value of hello, not bye. However, ghi will have a value of bye because it is not part of the same instruction that set abc to bye.

**Contents of docker file**

**from ubuntu**

**label maintainer bhushan**

**add https://drive.google.com/open?id=1BAb2PBv\_B40bD-7e1iRhmdOxhLgobw-qB2a61uJp\_9s /temp/hello.txt**

**cmd ["bash"]**

Docker build –f dockerfile –t my-ubuntu:latest

Docker images **#shows all images**

Docker ps –a **#shows active images**

Docker container ls –a **#shows active container**

Docker run –it –d –name Ubuntu my-ubuntu **#-it : interactive mode; -d : detach mode ; --name : name of container followed by name of image**

Docker container –rm Ubuntu **#to remove the container with Ubuntu name**

We use arg instruction to specify an argument

**ARG CODE\_VERSION=14.04 #**Name of this argument is **code\_version and a default value of 14.04** have been set.

The value for this argument can also be passed when we run the docker build command to build an image out of the docker file. However if no value is specified for this argument default value is used.

We can also use this as global variable in the docker file in case there are several references to code version and we only want specify value once.

**---------------------------------------------------**

ARG CODE\_VERSION=14.04

FROM ubuntu:${CODE\_VERSION}

LABEL maintainer=bhushan

RUN apt-get update && \

apt-get install -y curl && \

rm -rf /var/lib\*apt/list/\*

ENV HOME /root **# set the environment variable in key value format**

WORKDIR /root **#set the working directory for any instruction that fall in docker file**

CMD [“bash”] **#run bash command in the container based on this image**

**Difference between run and cmd command is that the state of a container after a run command will be commited to docker image.**

**On the other hand a command passed to cmd is only executed once the container is launched form that image**

**We use ENV instruction inorder to specify environment variable**

**#set the working directory for any instruction that fall in docker file**

**Entry point instruction, as the name suggest it effectively sets fix set of entry into the container which is find out from these image**

**Entry point configures your container to execute the command whenever container starts. (it is used to make container start at fixed point)**

**From ubuntu:trusty**

**ENTRYPOINT ["/bin/ping","-c","3"]**

**CMD ["192.168.99.100"]**

## ONBUILD

You will use on build in the docker file of images which will serve as the base image for other image. The purpose of having an onbuild instruction is that you want to define the instruction in the base image docker file, however you wanted it to be executed within the context of the downstream build.

Consider that you have set of onbuild instructions defined in the docker file of your base image, when you build your base image using that docker file, these onbuild instructions are not executed and do not part form of your base image build, however if you define another image with the docker file and you use that base image in your from instruction , then those onbuild commands defined in your base image will be executed as the part of your build of your downstream image and this is done immediately after the from instruction in your downstream docker file

ONBUILD [INSTRUCTION]

For example you might add something like this:

[...]

ONBUILD ADD . /app/src

ONBUILD RUN /usr/local/bin/python-build --dir /app/src

[...]

**Warning**: Chaining ONBUILD instructions using ONBUILD ONBUILD isn’t allowed.

**Warning**: The ONBUILD instruction may not trigger FROM or MAINTAINER instructions.

## STOPSIGNAL

When you create a container inorder to perform a very specific task and once that task is complete, you just want to exit from the container, in that case you will use stopsignal

STOPSIGNAL signal

The STOPSIGNAL instruction sets the system call signal that will be sent to the container to exit. This signal can be a valid unsigned number that matches a position in the kernel’s syscall table, for instance 9, or a signal name in the format SIGNAME, for instance SIGKILL.

**To save the state of a container and the share it with someone else , perform docker commit operation on a stopped container and then create a new docker image from it**

**Share a docker image**

When we create a container from a docker image, the container might have performed some operation and then changed its state in a manner which you cannot really define and state in a docker file and now you would like to save the state of a container and save it as another docker image and share it with someone else. For that you can perform docker commit operation on a stopped container and create a new docker image from it.

Now that we have an image which can be shared. We will want to make use of the fact that docker makes it portable and the way to test that is to upload the built image into a central repository and then pull it from another location and then run it there.

To do this we should know how to push images to registries and then pull from them in order to build containers out of those images.

**Lets say we have a repository setup on docker hub and we have a docker image which we want to share:**

* We first need to run docker tag command and tag should be of form <docker id>/<repo name>, where docker id is username of account on docker hub and repo name is the name given to repository in the docker hub account to which we want to push the image.
* When we run the docker tag command, we end up with a docker image but with new tag which we have specified.
* All of these is stored on our own local machine.
* And now in order to push this image to a docker repository we perform docker push operation and once again we specify the tag which we have created.
* The result of that push command is that any image which have the tag specified, will be pushed to docker hub repository.
* Now if there is someone who would to download this image and use it to spin up a container, they will need to use docker pull command along with the docker id and repo-name where the image is stored.
* And with that user will be able to download that image on to their own machine.

**To create an image out of stopped container**

**docker container commit continarid newimagename**

**To tag the new image with dockerid and repo name**

**docker tag imagename dockerid/reponame**

**To push a new tag to this repository,**

docker push taggedimagename

**------------------------------------------------------------------------------------**

**How you can use docker save inorder to save your docker image to a tar arhieve**

One reason to do this, is if you would like record of all your images and then store them somewhere in compressed format, then its safest to tar it up and then compress the file.

Its also an alternative means to share your image with someone else without having to use docker hub.

**docker save imagename > imagename.tar**

**Inorder to load image from tar file use following command**

docker load < imagename.tar

**--------------------------------------------------**

**command to search an image in docker hub local registry**

docker search textinimagename

**command to search an image in docker hub local registry based on filter**

docker search --filter is-automated=true imagename

docker search --filter is-official=true imagename

**command to search an image in docker hub local registry based on multiple filter**

docker search --filter is-official=true --filter stars=3 imagename

**we can also format the output of search result using format**

docker search –format “table{{.Name}}\t{{.StarCount}}” imagename

**-----------------------------------------**

**Command to get rid of dangling images**

docker image prune

dangling images are ones which were built out but not given a name.

**images from which no containers were created are called unused docker images and command to remove unused images: following commands gets rid of all unused and dangling images**

docker image prune –all

**command to remove images dangling/unused images based on filter and without any warning**

docker prune --all --force –filter “until=200h”

**----------------------------**

**Command to remove images from docker**

docker image rm imagename

docker image rm image1name image2name **#command to remove multiple images**

docker rmi imagename

Top of Form

**Which of the following is a function of Docker daemon?**

Storing Docker images

Provisioning user’s interaction with docker

**Responding to Docker API requests**

Container orchestration

Top of Form

**Which of the following is NOT an executable?**

Docker Container

**Docker Image**

Jar

None of the above

Bottom of Form

Top of Form

**In a Dockerfile, which keyword is used to define the base operating system image of the container?**

**FROM**

RUN

EXPOSE

ENTRYPOINT

Bottom of Form

Bottom of Form

Top of Form

**In a Dockerfile, which keyword is used to setup dependencies of the containerized application?**

FROM

**RUN**

EXPOSE

ENTRYPOINT

Top of Form

**In a Dockerfile, which keyword is used to determine user’s landing location once he/she spins up the container?**

FROM

RUN

EXPOSE

**ENTRYPOINT**

Bottom of Form

Bottom of Form

Top of Form

**What are the source and destination of the COPY command?**

**Image’s file system, container’s file system**

Container’s filesystem, image’s file system

Host file system

Docker local registry

**What is the tagging format to push your image to docker-hub?**

<Docker-ID>/<Image-name>

**<Docker-ID>/<Repository-name>**

<Repository-name>/<Image-name>

<Docker-ID>/<dockerfile>

**Docker container**

Command to create container from image which is on local machine.

**The command below will create container which will be in stopped state.**

docker container **create** --name containername --interactive --tty imagename

optionally

docker container **create** –name containername –i -t imagename

**command to start container which is stopped state:**

docker container **start** containername

**command to stop container which is running state:**

docker container **stop** containername

**command to restart a container from running\stopped states**

docker container restart containername

**docker container ps --all --format “table{{.Names}}\t{{.ID}}\t{{.Status}}\t{{.Ports}}”**

------------------------------------------------

**DOCKER ATTACH COMMAND**

This command is used to attach our local standard inputs, standard outputs and standard error streams to that of a running container,

Docker attach can effectively allow us to interact with the shell of a container and run commands on it

Docker run can be considered as combination of docker create followed by docker start

**docker run -d -i -t --name containername imagename**

-d option : so that container runs in detach mode in the background

We specify -i so that standard in is available for the container.

We specify -t for the teletype option in order to type input into the standard in and then view the text output

After this we will attach our created container to the shell using following command

**docker attach containername #this command will connect our standard input, output and error stream to that of a container**

**following is tutorial on how we can execute a command on a running docker container without attaching to it.**

**We will also see how we can interact with a shell of the container and exit from it without stopping from it.**

**Docker exec command is used to execute new command in a running container**

First start the container : **docker start containername**

Then type **: docker exec –I –t containername bash**

**Docker attach connect to the main process of the container whereas docker exec creates the new process, and we don’t attach to the main process of the container,**

**When we type exit in the shell of bash, container is not closed, its still up and running.**

**Two different ways of starting a container and check whether in running state or not**

docker container create --name containername –I –t imagename **#this creates a container in stopped state**

docker run –I –t –d –name containername imagename **#this creates a container in running state**

**command to inspect containers**

docker inspect –format “{{range .NetworkSettings.Networks}}{{.IPAdress}}{{end}}” containeridorcontainername

**above command will provide ip address of container which are in running state**

**Normal command : docker inspect containeridorname container2idorname**

**Command to fetch logs of a container**

docker container logs containeridorname

**command to fetch last 10 logs of a container**

docker container logs --tail=10 containeridorname

for debugging process, one may often want to check all the running processes In a docker container and for that container top command is used **[only on running containers]**

**docker container top containernameorid**

command to get rid of all stopped containers

**docker container prune**

alternatively to remove containers without any warning we use

**docker container prune –force**

**difference between prune and rm :**

**both remove stopped containers but for rm when used with –force removes running containers as well**

**docker container rm does not throw any warning before removing container whereas docker container throw warning.**

command to remove container

following commands will remove only stopped containers

**docker container rm containernameorid**

**docker container rm containernameorid container2nameorid**

command to remove container forcefully both in running and stopped state

**docker container rm --force containernameorid**

**docker container rm --force containernameorid container2nameorid**

alternatively we can also use following command to remove all containers

**docker rm --force `docker ps –a –q`**

above command will remove all containers and return their id

following is the tutorial on docker container kill command which is used to kill the main process of the container and bring it to the stopped state

**difference between docker kill and docker stop**

when we use **docker stop command, docker will attempt to gracefully shutdown** the container by sending its main process a sick term signal, if the process does not respond for a while, then it will forcefully kill the process by sending it a sick kill signal.

But when **we use docker kill, it just moves on right away to the sick kill signal**

The reason one might use docker container kill command is when a container is effectively a zombie, where its gone into some kind of frozen state and is unable to get out of it.

**Command to rename container**

docker container rename oldcontainername newcontainername

**There will be occasion where you want to know what changes were made to a container’s file system, since it was created.**

docker container diff containername

**above command inspect the changes to files or directories on a container’s filesystem.**

**bhush@LAPTOP-15Q0IKL6 /e/Docker/Docker Toolbox**

**$ docker container start test-ubuntu**

test-ubuntu

**bhush@LAPTOP-15Q0IKL6 /e/Docker/Docker Toolbox**

**$ docker exec -it test-ubuntu bash**

root@46defe5d2c75:/# mkdir bhush

root@46defe5d2c75:/# ls

bhush bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var

root@46defe5d2c75:/# exit

exit

**bhush@LAPTOP-15Q0IKL6 /e/Docker/Docker Toolbox**

**$ docker container diff test-ubuntu**

A /bhush **#denotes bhush directory has been added by letter “ D”**

C /root **#denotes that changes had been made to root directory by letter “C”**

A /root/.bash\_history **# denotes that all the changes made are added to bashhistory file by letter “A”**

bhush@LAPTOP-15Q0IKL6 /e/Docker/Docker Toolbox

**There will be occasion where container had made changes to its own settings or to its file system and you would like to commit those changes to a new image which can be used to spin up other container.**

**Command :** docker container commit containername imagename

**In this demo we will see how a port in a docker container can be published abound to a port of the host machine it is running on.**

-p <host\_port>:<container\_port>

-p <portnumber on host machine>:<portnumber on container>

**Publish an exposed port that is accessible publicly over the internet**

**Any connction that come into port 80 of the host machine will be directed to port 80 of the container which is were httpd will be istening of connection.**

**docker run –I –t –d --name test-httpd –p 80:80 httpd**

**alternative way to publish ports from container to a host and that is using upper case P option**

**docker run –I –t –d --name test-httpd1 –P httpd**

**rather than explicitly specifying port mapping from the host to the container, when using upper case P then all the exposed ports of your container will be bound to some random open port on your host.**

$ docker ps --all

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

f7a5f624e9c3 httpd "httpd-foreground" 7 seconds ago Up 5 seconds **0.0.0.0:32768->80/tcp** test-httpd1

6b8778950d9d httpd "httpd-foreground" 4 minutes ago Up 4 minutes **0.0.0.0:80->80/tcp** test-httpd

46defe5d2c75 ubuntu "/bin/bash" 29 minutes ago Up 24 minutes test-ubuntu

41f6cc551800 redis "docker-entrypoint.s…" 31 minutes ago Up 29 minutes 6379/tcp test-redis

In first httpd1 docker maps random host port to port 80 of container, whereas in second docker maps port 80 of host to port 80 of container.

**Command to list out port mapping of a container**

docker container port containername

------

bhush@LAPTOP-15Q0IKL6 /e/Docker/Docker Toolbox

**$ docker container port test-httpd**

**80/tcp -> 0.0.0.0:80**

bhush@LAPTOP-15Q0IKL6 /e/Docker/Docker Toolbox

**$ docker container port test-httpd1**

**80/tcp -> 0.0.0.0:32768**

**------------------------------------------------------------------------------------------------------------------------------------------------------**

Volume is used to define volume for the image

Example of volume instruction used in docker file

**VOLUME [“/vol-nginx”] #instructions says volume is mounted at the location /vol-nginx within the docker container**

**Volume allow us a way to possess data which is created within docker containers**

**Copy command**

Difference between copy and the add instruction is that in a copy instruction **source can only be a location** within your **file system** while for **the add instruction source can also be an url**.

Nocache option is used to ensure that docker file that is used in order to build an image and that it is not building out of layer which may be present in the cache.

Top of Form

Which of the following commands will you use to get a list of containers?

docker compose

docker pull

**docker ps**

docker run -a

Top of Form

The command “docker create” generates a running container from a docker image. True or False?

True

**False**

Top of Form

Question 3:

What happens when we use “docker exec” command?

A new process spawns up within the container and all of the processes exit once the command is executed

A new process spawns up within the container and exits without disturbing the container init process.

The command executes within the container init process and the process does not exit.

The command executes within the container init process and the process exits after the command’s execution.

Bottom of Form

Top of Form

Which docker command is used to save changes of container in a new Docker image?

docker save

**docker commit**

docker create

None of the above

Top of Form

Which docker command is used to list the running processes within a specific container?

docker container ps -a

docker ps -a

**docker container top**

docker container top -a

Which argument is used with “docker run” to let docker decide the port on host to be mapped to the exposed container port?

-p

**-P**

-a

A

Top of Form

What is the command to show last 10 log entries of any running container?

docker container logs 10 <container\_name>

docker container logs --show=10 <container\_name>

docker container info --last=10 <container\_name>

**docker container logs --tail=10 <container\_name>**

**Different forms of storage in docker**

**From within a container it is possible to write data into the containers writable layer. So when a container has been built out of an image, that image actually contains multiple layers in it. We include the base image and also the layer for each of the instruction which make up the image. All of these layers are read only. On top of all of this there is writable layer in which the container can write information. Drawback: difficult to get data in this writable layer out of the container and perhaps into another container on the same host. Additionally that data cannot me moved out of the host running the container and perhaps use somewhere else. To overcome this, docker offers 3 different forms of storage:**

**Volume**

**Bind mounts**

**Tmpfs volume**

**Volumes are always the default choice**

* **Volumes** are stored in a part of the host filesystem which is *managed by Docker* (/var/lib/docker/volumes/ on Linux). Non-Docker processes should not modify this part of the filesystem. Volumes are the best way to persist data in Docker.
* **Bind mounts** may be stored *anywhere* on the host system. They may even be important system files or directories. Non-Docker processes on the Docker host or a Docker container can modify them at any time.
* **tmpfs mounts** are stored in the host system’s memory only, and are never written to the host system’s filesystem.

**One area where volume has advantage over bind mounts is that it is not tried to directory structure of host on which the container is running.**

When you need to back up, restore, or migrate data from one Docker host to another, volumes are a better choice.

**Volumes: Created and managed by Docker**. You can create a volume explicitly using the docker volume create command, or Docker can create a volume during container or service creation.

When you create a volume, it is stored within a directory on the Docker host. When you mount the volume into a container, this directory is what is mounted into the container. This is similar to the way that bind mounts work, except that volumes are managed by Docker and are isolated from the core functionality of the host machine.

A given volume can be mounted into multiple containers simultaneously. When no running container is using a volume, the volume is still available to Docker and is not removed automatically. You can remove unused volumes using docker volume prune.

When you mount a volume, it may be named or anonymous. Anonymous volumes are not given an explicit name when they are first mounted into a container, so Docker gives them a random name that is guaranteed to be unique within a given Docker host. Besides the name, named and anonymous volumes behave in the same ways.

Volumes also support the use of volume drivers, which allow you to store your data on remote hosts or cloud providers, among other possibilities.

**Bind** **mounts**: Available since the early days of Docker. Bind mounts have limited functionality compared to volumes. When you use a bind mount, a file or directory on the host machine is mounted into a container. The file or directory is referenced by its full path on the host machine. The file or directory does not need to exist on the Docker host already. It is created on demand if it does not yet exist. Bind mounts are very performant, but they rely on the host machine’s filesystem having a specific directory structure available. If you are developing new Docker applications, consider using named volumes instead. You can’t use Docker CLI commands to directly manage bind mounts.

Bind mounts allow access to sensitive files

One side effect of using bind mounts, for better or for worse, is that you can change the host filesystem via processes running in a container, including creating, modifying, or deleting important system files or directories. This is a powerful ability which can have security implications, including impacting non-Docker processes on the host system.

**tmpfs** **mounts**: A tmpfs mount is not persisted on disk, either on the Docker host or within a container. It can be used by a container during the lifetime of the container, to store non-persistent state or sensitive information. For instance, internally, swarm services use tmpfs mounts to mount secrets into a service’s containers.

Bind mounts and volumes can both mounted into containers using the -v or --volume flag, but the syntax for each is slightly different. For tmpfs mounts, you can use the --tmpfs flag. However, in Docker 17.06 and higher, we recommend using the --mount flag for both containers and services, for bind mounts, volumes, or tmpfs mounts, as the syntax is more clear.

**Good use cases for volumes**

Volumes are the preferred way to persist data in Docker containers and services. Some use cases for volumes include:

**Sharing data among multiple running containers**. If you don’t explicitly create it, a volume is created the first time it is mounted into a container. When that container stops or is removed, the volume still exists. Multiple containers can mount the same volume simultaneously, either read-write or read-only. Volumes are only removed when you explicitly remove them.

When the Docker host is not guaranteed to have a given directory or file structure. Volumes help you decouple the configuration of the Docker host from the container runtime.

When you want to store your container’s data on a remote host or a cloud provider, rather than locally.

When you need to back up, restore, or migrate data from one Docker host to another, volumes are a better choice. You can stop containers using the volume, then back up the volume’s directory (such as /var/lib/docker/volumes/<volume-name>).

**Good use cases for bind mounts**

In general, you should use volumes where possible. Bind mounts are appropriate for the following types of use case:

Sharing configuration files from the host machine to containers. This is how Docker provides DNS resolution to containers by default, by mounting /etc/resolv.conf from the host machine into each container.

Sharing source code or build artifacts between a development environment on the Docker host and a container. For instance, you may mount a Maven target/ directory into a container, and each time you build the Maven project on the Docker host, the container gets access to the rebuilt artifacts.

If you use Docker for development this way, your production Dockerfile would copy the production-ready artifacts directly into the image, rather than relying on a bind mount.

When the file or directory structure of the Docker host is guaranteed to be consistent with the bind mounts the containers require.

**Good use cases for tmpfs mounts**

tmpfs mounts are best used for cases when you do not want the data to persist either on the host machine or within the container. This may be for security reasons or to protect the performance of the container when your application needs to write a large volume of non-persistent state data.

Tips for using bind mounts or volumes

If you use either bind mounts or volumes, keep the following in mind:

If you mount an empty volume into a directory in the container in which files or directories exist, these files or directories are propagated (copied) into the volume. Similarly, if you start a container and specify a volume which does not already exist, an empty volume is created for you. This is a good way to pre-populate data that another container needs.

If you mount a bind mount or non-empty volume into a directory in the container in which some files or directories exist, these files or directories are obscured by the mount, just as if you saved files into /mnt on a Linux host and then mounted a USB drive into /mnt. The contents of /mnt would be obscured by the contents of the USB drive until the USB drive were unmounted. The obscured files are not removed or altered, but are not accessible while the bind mount or volume is mounted.

Bottom of Form

Bottom of Form

**Command to view volumes**

docker volume ls

**command to create volume**

docker volume create --name volumename

**command to create container with volume**

create a volume and mount it on a container when container is initiated

-v <volumename>:<container-directory>

docker run –d --name containername –v volumename:/containerdirectory imagename

**command to create container with volume mounted on it**

**mount the volume using mount flag on a container**

docker run –it –d --name test-busybox1 --mount source=vol-busybox-1,target-/app busybox

**command to create container with volume mounted on it with read-only restriction**

docker run –it –d –name containername –v volumename:contianerdirectory:**ro** imagename

docker run –it –d --name test-busybox1 --mount source=vol-busybox-1,target-/app:ro busybox

**alternative way for readonly volume**

docker run –it –d --name test-box --**mount** source=test-volume,target=/app,**readonly** busybox

**command to inspect volume**

docker volume inspect volumename

**Command to delete volumes not attached to a container**

docker volume prune

**In this demo we will be creating a new volume and then attaching it to a docker container, the container will then make some changes to that volume and then we will get out of that container to view the changes, which where made within the container.**

**Command to create volume :** docker create volume --name test-ubuntu xor docker volume create test-ubuntux

**Command to attach volume to container :**

docker run –I –t –d –name test-ubuntu **–v test-ubuntux:\tmp** Ubuntu

**command to attach container to standard in/out/error**: docker attach test-ubuntu

**create a file and store it in tmp directory where volume was mounted**

**root@7b8912f18391:/#** echo "this is text file">/tmp/msg.txt

ssh to default docker machine : docker-machine ssh default

bhush@LAPTOP-15Q0IKL6 /e/Docker/Docker Toolbox/test-nginx

**$ docker-machine ssh default**

( '>')

/) TC (\ Core is distributed with ABSOLUTELY NO WARRANTY.

(/-\_--\_-\) www.tinycorelinux.net

**docker@default:~$ sudo su -**

root@default:~# cd **/var/lib/docker/volumes**

root@default:/var/lib/docker/volumes# ls

**metadata.db somvolume testbox ubuntu-volume**

root@default:/var/lib/docker/volumes# cd ubuntu-volume

root@default:/var/lib/docker/volumes/ubuntu-volume# ls

**\_data**

root@default:/var/lib/docker/volumes/ubuntu-volume# cd \_data

root@default:/var/lib/docker/volumes/ubuntu-volume/\_data# ls

**msg.txt**

root@default:/var/lib/docker/volumes/ubuntu-volume/\_data#

**start a container with a bind mount**

**command :**

docker run -it –d --name test-busybox --mount type-bind,souce-/tmp,target=/app nginx

**start a container by mounting a host(source) directory on conainer’s (target) directory**

**start a container with a bind mount with read only permissions**

docker run -it –d --name test-busybox --mount type-bind,souce-/tmp,target=/app,**readonly** nginx

**start a container with tmpfs mount**

docker run -it -d --name test-busybox --mount **type=tmpfs**,**target=/app** busybox

**this means any files that are written or read from /app directory within the container actually reside in the main memory of the host.**

**Alternative way to create tmpfs using tmpfs flag**

docker run –it –d --name test-ubuntu **--tmpfs /app** Ubuntu

**What should we use to save the data created by a running container?**

Docker-hub

Docker trusted registries

**Volumes**

All of the above

**Where will the container store its runtime generated data by default?**

Docker-hub

**Container’s writable layer**

Host machine’s Docker repository

Volumes

**Containers using bind mount as storage are as portable as containers using Volumes as storage.**

True

**False**

**Which are the types of storages offered by Docker?**

Volumes and Bind Mounts

Bind Mounts and tmpfs

tmpfs and Volumes

**Volumes, Bind Mounts and tmpfs**

**Which command is used to list out available volumes on a docker host?**

docker volumes -a

docker volumes ls

**docker volume ls**

docker volume –a

**When is the volume to be mounted with a container specified in Docker?**

Writing Dockerfile

Creating Docker image

Running Docker Container

**A and C both**

**What are the default volume privileges when it is mounted with a container?**

Read only

**Read-Write**

Write only

**What is the use of docker volume prune command?**

Remove all volumes

Remove unmounted volumes

**Remove unused volumes**

Remove all storage objects

Bottom of Form

**DOCKER SWARM**

Docker swarm is a container orchestration tool supplied by docker.

Server cluster benefits:

Makes our application highly available:

* no single point of failure **[if one of the host goes down, our application is still up and running]**
* possible to take down individual nodes for the purposes of maintenance **[in this case our application will still be running because there is one replica of each microservice which is still in operation].**

Allows horizontal scaling of your application

* if more and more people are using our application and demand for the resources get higher then we can easily provision new server and then add them to our cluster. We can then create more replica of our microservices and the distribute them in this cluster and then have a load balancer which will distribute the load between all the replicas of our container.

We will need to create and manage the cluster and ensure that all our containers are able to talk to each other irrespective of which host they are running on.

We also need to make sure that we set a container to run on a host which does have enough speed resources available which is also known as scheduling of containers.

If a container were to go down for some reason we need to make sure that we are able to detect that and then restart the container in a correct way.

We would also like to scale up and scale down replicas of our microservices depending on the demand of our application.

And also the app running on our container will be listening to connections on certain port and we would like to expose these ports to the outside.

**These are the lots of things to manage and for this reason we have docker swarm.**

**A cluster of one or more docker engines**

So if we have number of hosts which are running the docker engine then docker will allow us to form the cohesive unit out of them and they will form a cluster or a docker swarm.

**Schedule containers in cluster**

We just need to tell docker swarm how many replicas of a particular container we want running and it will take care of scheduling them across all the nodes in the cluster.

**Maintain cluster state**

Docker swarm also ensures that the specified number of replicas of each of our containers are up and running at all time. So it will handle restart of any failed containers as well.

**User services to scale containers.**

We can also make use of services in docker swarm to scale up or scale down the number of replicas of a container we have running.

**Expose ports to the outside**

And also docker swarm makes it very easy to expose the ports which are containers listening on to the outside world.

In order to get all these pieces to work, docker swarm makes use of **manager nodes** whose **responsibility** includes:

* **Scheduling** of all the containers in the cluster
* **Maintain the cluster state** and ensuring that the specified number of replicas of each container are up and running.
* The managers also **serves as http api endpoints** if you are making api calls
* Docker recommends that we have 3 managers in any docker swarm.

The other kind of nodes in docker swarm are **worker** **nodes**: whose job is simply

To get instructions from the manager and execute them.**[Simply execute containers]**

**Services**

Docker swarm **object which makes it possible to scale up and scale down** our application **is a service.**

**So think of a service as a set of identical containers which are distributed across the nodes in your cluster.**

**So this simply collection of all the replicas of your app or microservice.**

**Two forms of services available in docker:**

* **Replicated service:** Specifies number of replicas are scheduled in the docker swarm. **[where you explicitly state the number of replicas to be scheduled in docker swarm.**
* **Global Service:** Runs one task on every node useful for monitoring agents, virus scanners. **[ global service will run one task or one container of that service on every single node in the cluster. This is especially useful when you have an application such as monitoring agent or a virus scanner which you want to be running on every single node.]**

**Secrets in docker**

* Sensitive data can be stored as docker secrets
* You will often have some sensitive information such as a password or a private ssh key which you want your container to access but you want to store in docker file or someplace where it is easily visible.
* **For that purpose, we have** **docker secrets**
* So secret will allow your password or sensitive information to be passed on to the services. And it will store that information in a file which will be in a **directory /run/secrets/<secret-name>** **within** **your** **container**.
* So you will need to build your application in such a way that it picks up that secret from that file which is located in that directory and you never really need to know the contents of that file.
* All you **need to know is the name of that secret**. So if your secret contains database password then It can be database administrator’s responsibility to frequently change the password and update the secret and the app developer doesn’t really need to be made aware the change in password as the application will be referencing the password by the name of the secret.
* **Secrets are only available to use in docker swarm** and not a standalone container

**Docker swarm vs kubernetes**

Easier to install and setup

Built into official docker cli

More lightweight

**Kubernetes**

More mature product

Has managed solutions on aws, azure, gcp

Has more features (e.g. autoscaling)

In this lab we will be using docker machine in order provision a number of vm on which we will create docker swarm,

We will be creating one manager and 3 worker

**Command to create manager and worker nodes**

docker-machine create --driver virtualbox manager1

docker-machine create --driver virtualbox worker1

docker-machine create --driver virtualbox worker2

docker-machine create --driver virtualbox worker3

**command to see all docker machine nodes**

docker-machine ls

**command to view ip address of each node**

docker-machine ip manager1

**in order to create docker swarm we will need to first log in to manager node for that swarm**

**command to log in to one of the node**

docker-machine ssh manager1

**Initialize swarm on your manager node**

**docker swarm init --advertise-addr 192.168.99.101**

manager node now becomes manager of the swarm

**command used to get command which can be used from nodes In order to join swarm as a worker**

docker swarm join-token worker

**example**

docker@manager1:~$ **docker swarm join-token worker**

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

docker swarm join --token SWMTKN-1-00i5eg7x10qjceumutqrsogaytqo3hkpl5usi4kyo201kispyc-a1fl5il0m881xz1ydtjay3yhe 192.168.99.101:2377

**if you would like to add another manager node to the swarm following command is used:**

docker swarm join-token manager

**example**

docker@manager1:~$ **docker swarm join-token manager**

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

docker swarm join --token SWMTKN-1-00i5eg7x10qjceumutqrsogaytqo3hkpl5usi4kyo201kispyc-36mtyasye8ff0l9yu9ejwi52a 192.168.99.101:2377

**Note : even though swarm has multiple manager nodes only one of them is designated as their leader.**

**Open three different docker terminal and type following commands**

docker-machine ssh worker1 [1st terminal]

docker-machine ssh worker2 [2nd terminal]

docker-machine ssh worker3 [3rd terminal]

**and run the token provided in docker swarm join-token worker command in each of this terminal**

we will get following result in each of the terminal after entering following token

docker swarm join --token SWMTKN-1-00i5eg7x10qjceumutqrsogaytqo3hkpl5usi4kyo201kispyc-a1fl

5il0m881xz1ydtjay3yhe 192.168.99.101:2377

**This node joined a swarm as a worker.**

**Command to get swarm config information**

docker info **# from the manager node**

**creating and deploying service on docker swarm**

docker service create --replicas 4 –p 80:80 --name servicename nginx

**command to view all services**

docker service ls

**command to list out all task running on every node**

**we use docker service ps command and servicename in order to view container processes associated with the service**

**example:**

**docker@manager1:~$ docker service ls**

**ID NAME MODE REPLICAS IMAGE PORTS**

**sbe7v6z8lxhq web replicated 4/4 nginx:latest \*:80->80/tcp**

**docker@manager1:~$ docker service ps web**

**ID NAME IMAGE NODE DESIRED STATE CURRENT STATE ERROR PORTS**

**b8kae1yoyxak web.1 nginx:latest worker1 Running Running 2 minutes ago**

**oa5auimwdypm web.2 nginx:latest worker2 Running Running 2 minutes ago**

**nvodl6ayw6f1 web.3 nginx:latest worker3 Running Running 2 minutes ago**

**vrl1p7brxwze web.4 nginx:latest manager1 Running Running 2 minutes ago**

Bottom of Form

Bottom of Form

**Incase application becomes popular and 4 replicas are not enough in such a cases we may scale up to 8 replicas for that we use Command :**

docker service scale servicename=8

**docker node inspect command to get details of all the nodes in the docker swarm**

**this command can only be executed from manager node of the swarm**

docker node inspect self

**command to inspect node details in worker from manager terminal**

docker node inspect worker1

every now and then we need to perform maintenance task on your host in the infrastructure and this host may include the cluster on which you have docker swarm.

While the maintenance task are performed on worker node in swarm, you want the manager to know no more container should be created on it while the maintenance is going on

**Command to set availability of worker node from active to drain**

docker node update –availability drain worker1

example

docker@manager1:~$ docker node update --availability drain worker1

worker1

docker@manager1:~$ docker service ps web

ID NAME IMAGE NODE DESIRED STATE CURRENT STATE ERROR PORTS

2fcv2p1pkakm **web**.1 nginx:latest **worker3** Running Starting 3 seconds ago

**b8kae1yoyxak \\_ web.1 nginx:latest worker1 Shutdown Shutdown 4 seconds ago**

oa5auimwdypm web.2 nginx:latest worker2 Running Running 20 minutes ago

nvodl6ayw6f1 web.3 nginx:latest worker3 Running Running 20 minutes ago

vrl1p7brxwze web.4 nginx:latest manager1 Running Running 20 minutes ago

yfw1uppuxw6x web.5 nginx:latest worker3 Running Running 11 minutes ago

xm2gy9gh39f2 web.6 nginx:latest manager1 Running Running 11 minutes ago

hwn1u73hwk59 **web**.7 nginx:latest **worker2** Running Starting 2 seconds ago

**p0s6stlg37xw \\_ web.7 nginx:latest worker1 Shutdown Shutdown 3 seconds ago**

4nfe8s9vzk2n web.8 nginx:latest worker2 Running Running 11 minutes ago

**How do we update all our containers in nginx service so that they use the latest image of nginx**

**Command : docker service update --image nginx:latest web**

**Roll out updated nginx no every node of cluster**

docker service update --image nginx:latest web

**command to delete a service which we no longer need**

docker service rm servicename

docker@manager1:~$ **docker service rm web**

web

docker@manager1:~$ **docker service inspect web**

[]

**Status: Error: no such service: web, Code: 1**

**Create a service in global mode**

docker service create --mode global --name test-redis redis

**global mode services are useful when you want some task to run globally that is on every node in your swarm**

**command to create secret**

**echo “this is secret file” | docker secret create secretname –**

docker@secondmanager:~$ **echo "this is secret file" | docker secret create my-secret-data -**

qv3ir6u5chclner8zdrfwr3vc

docker@secondmanager:~$ **docker secret ls**

ID NAME DRIVER CREATED UPDATED

qv3ir6u5chclner8zdrfwr3vc **my-secret-data** 18 seconds ago 18 seconds ago

docker@secondmanager:~$

**create a service in swarm with secret**

**docker service create --name test-redis --replicas 3 --secret my-secret-data redis:alpine**

**trying remove secret file without deleting service throws error**

docker@secondmanager:~$ **docker secret rm my-secret-data**

**Error response from daemon: rpc error: code = InvalidArgument desc = secret 'my-secret-data' is in use by the following service: test-redis.**

docker@secondmanager:~$ **docker service rm test-redis**

test-redis

docker@secondmanager:~$ **docker secret rm my-secret-data**

my-secret-data

**command to remove node from docker swarm**

login into the worker node which you want to remove from the swarm and type :

docker swarm leave

**docker@secondworker2:~$ docker swarm leave**

**Node left the swarm.**

**Command to remove worker node from our swarm entirely**

**docker node rm workernodename**

docker@secondmanager:~$ **docker node rm secondworker1**

secondworker1

docker@secondmanager:~$ **docker node rm secondworker3**

secondworker3

docker@secondmanager:~$ **docker node ls**

ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS ENGINE VERSION

c0lg5jdbmav3quysn7g7o76xk \* secondmanager Ready Active Leader 18.09.1

**command to promote worker node to become a manager and also how to demote a manager to become a worker**

**command to promote worker node to manager**

docker node promote workernode

docker@secondmanager:~$ **docker node promote secondworker1**

**Node secondworker1 promoted to a manager in the swarm.**

docker@secondmanager:~$ docker node ls

ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS ENGINE VERSION

c0lg5jdbmav3quysn7g7o76xk \* secondmanager Ready Active **Leader** 18.09.1

octcqyhd8bm5s3g26ax1p4v04 secondworker1 Ready Active **Reachable** 18.09.1

rm26212c2gnff80n71zar6lwv secondworker2 Ready Active 18.09.1

docker@secondmanager:~$

**command to demote manager to worker**

docker@secondmanager:~$ **docker node demote secondmanager**

Manager secondmanager demoted in the swarm.

docker@secondmanager:~$ **docker node ls**

**Error response from daemon**: rpc error: code = Unavailable desc = all SubConns are in TransientFailure, latest connection error: connection error: desc = "transport: Error while dialing dial unix /var/run/docker/swarm/control.sock: connect: no such file or directory"

docker@secondmanager:~$

**Notes : after denoting manager node to worker node we cannot run manager level commands like docker node ls from manager terminal , instead we should run those terminal node which has been promoted to be an manager**

docker@**secondworker1**:~$ **docker node ls**

ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS ENGINE VERSION

c0lg5jdbmav3quysn7g7o76xk secondmanager Ready Active 18.09.1

octcqyhd8bm5s3g26ax1p4v04 \* **secondworker1** Ready Active **Leader** 18.09.1

rm26212c2gnff80n71zar6lwv secondworker2 Ready Active 18.09.1

**Note : if we try to remove the one and only one remaining manager in swarm it will throw error**

docker@secondworker1:~$ **docker node demote secondworker1**

**Error response from daemon**: rpc error: code = FailedPrecondition desc = attempting to demote the last manager of the swarm.

**What is the advantage of having a cluster of Docker instances over a single instance?**

Avoiding single point of failure incidents

Maintaining nodes without taking the application down

Easy to scale horizontally

**All of the above**

**Which of the following tasks is NOT part of container orchestration?**

Scaling containers

Scheduling containers

Maintaining Cluster nodes

**Writing Container images**

Bottom of Form

**Which Docker Swarm object is used to make the containerized application available to communicate with outside world (outside the cluster)?**

Replicas

Containers

**Services**

Volumes

**Which parameter does a service use to sort the containers to be controlled by it?**

Base image

Application Port

**Tag**

Volume

**What type of service should be used to create “dev” and “prod” copies of container instances?**

Global Service

Deployment Service

**Replicated Service**

Enterprise Service

**What kind of service will you use for running a virus-scanner on all cluster nodes?**

Secure Service

Replicated Service

Monitoring Service

**Global Service**

**Which of the following type of information should NOT be passed using a secret?**

Password

**Number of Replicas**

SSH private key

Encoded Text

Bottom of Form

Bottom of Form

**Where does Docker swarm store the secrets?**

/run/<secret-name>

/user/run/secret/<secret-name>

/run/secret/<secret-name>

**/run/secrets/<secret-name>**

**Which is the advantage of using Docker Swarm over Kubernetes for container orchestration?**

More features

Managed solution on popular public cloud platforms

**More lightweight**

None of the above

**What is the minimum number of manager nodes recommended by Docker Swarm?**

2

**3**

5

1

**Which command is used to connect to a node of a Docker Swarm cluster?**

docker-login

**docker-machine ssh**

docker-node login

docker-node ssh

**Which command is used to spin-up a Docker Swarm Cluster?**

docker swarm run

docker swarm begin

**docker swarm init**

docker swarm make

**What makes Leader node unique among the manager nodes?**

It can schedule and maintain containers on worker nodes

**It can promote or demote a node by changing it’s role**

It can expose containers to specific ports

It can drain nodes for maintenance

**Docker Networks**

Task of docker network is to broker the connectivity between all our running containers and to that without exposing us to the complexity of the underlying network.

* Brokers connectivity for Docker containers
* Abstracts away underlying network complexity

To do that docker offers 4 kinds of network:

* **Bridge Network (default network driver)**
* **Overlay Network (useful with docker swarm)**
* **MACVLAN Network**
* **Host Network**

**Bridge Networks (Defualt network driver)**

Creates a private network within the host on which it is created.

Allows for communication between containers which are all on the same bridge network but prevents communication to other containers which are not on it.

It allows a container to talk to outside world by exposing the ports on which they are listening for connections.

In order to make all of this happen, bridge network automatically take care of the integrity of networking such as bridges, ip tables, service discovery and so on.

One potential limitation of a bridge network is that its very much local in scope and this network does not extend beyond the host on which it is created.

**Overlay Networks**

Allow networking across multiple host. And this is exactly why they are used in docker swarm where your services may span multiple nodes.

By abstracting away the underlying physical network, overlay network make a container more portable, so you don’t need worry about underlying details of physical network they are running on, as long as they are able to run on an overlay network.

Overlay network also encrypt communication between the containers which are running on it.

**MACLVAN Network**

This networks essentially assigns a mac address to your container and make it appear like a physical device which is connected to the network.

This is usually meant for legacy application where a physical connection to the network is expected and application are addressed using their mac address.

**Host Networks**

As the name suggest we end up using network of the host on which the container is running. So there is no isolation here between the container and the host.

So the way it works is, if your container is listening to connections on port 80, then the app running in your container can be reached by hitting port 80 on your host IP address or url

If your container does not publish any ports then using a host network will not have any effect. A host network is simply meant to automatically open up the ports on which the containers are listening.

**How to create a network in docker [Bridge network]**

**Create a network and specify the driver to manage the network**

docker network create -d=bridge my-new-bridge #**specify the driver using –d option and drive type bridge if you don’t specifydriver type bridge network will be created by default. [other options subnet,ip range, gateway**

**how to attach container to newly created network**

**first create a container :** docker run –it –d --name mybusybox busybox

**connect the container to the network :** docker network connect my-new-bridge mbusybox

**run** docker inspect mybusybox **#in the network section [last section] you will see the network name and its config.**

**If we want to a container to run on a custom network**

docker run –it -d --name mynginx –network=my-new-bridge nginx

**how to disconnect container from network**

docker network disconnect my-new-bridge mynginx

**command to inspect docker network**

docker network inspect bridge

**how to create overlay network:**

**in order to create an overlay network we need to be connected to the manager of a docker swarm and for that we use** docker-machine ssh managernode and docker-machine ssh workernode.

We cannot delete a network which has active containers on it.

docker@manager1:~$ **docker network rm my-overlay-network**

Error response from daemon: rpc error: code = FailedPrecondition desc = network n5dwdg317hjwf0vkpt9s5smim is in use by service mh7z8wmacsrxd2joa5y7xg8dk

**Note: we cannot delete a secret before deleting the service, Similarly we cannot delete a network before deleting the service**

docker@manager1:~$ **docker service rm test-nginx**

test-nginx

docker@manager1:~$ **docker network rm my-overlay-network**

my-overlay-network

**creating a host network**

docker run -it -d --name test-nginx --network host nginx

**how to remove an network**

**command to remove individual network**

docker network disconnect networkname containername

docker network rm networkname

example

**bhush@LAPTOP-15Q0IKL6 /e/Docker/Docker Toolbox**

$ **docker network rm my-new-brige**

Error response from daemon: error while removing network: network my-new-brige id 0ce569f1edb76de49ddc4cde419a422daa0cab65adaca2b88792d50a875d1f9d has active endpoints

**Note first disconnect network before removing it**

bhush@LAPTOP-15Q0IKL6 /e/Docker/Docker Toolbox

$ **docker network disconnect my-new-brige mybusybox**

bhush@LAPTOP-15Q0IKL6 /e/Docker/Docker Toolbox

$ **docker network rm my-new-brige**

my-new-brige

**command to remove unused docker networks**

docker network prune

**Which object of docker manages the internal connectivity of Docker containers?**

Docker Swarm

**Docker Networks**

Docker Images

Service Objects

**Docker containers can communicate with the outside world with the default docker network. True or False?**

True

**False**

Bottom of Form

Bottom of Form

**Which Docker network is the default network in Docker?**

MACVLAN

Host

Overlay

**Bridge**

Bottom of Form

**Which Docker network is preferred for spinning up a Docker Swarm cluster?**

MACVLAN

Host

**Overlay**

Bridge

Bottom of Form

**Which Docker network should be used to treat containers as physical devices in a cluster or network?**

**MACVLAN**

Host

Overlay

Bridge

**It is possible to remove network isolation between a container and its host. True or False?**

**True**

False

**UCP – universal control plane – container management solution**

**Command to pull official latest ucp image from docker hub**

docker image pull UCP image from docker/ucp:2.2.6

**How does docker store its license key for the enterprise edition?**

Using a licence key file

Using Secret

**Using environment variable**

None of the above

**Which of the following TCP port is used for UCP’s API communication with host machine?**

**443**

80

8080

36071

Bottom of Form

**How is UCP treated by Docker?**

A service

**A container image**

An object

All of the above

Bottom of Form

Bottom of Form

**Which of the following facility does Docker EE offer?**

Large scale container management

UCP

DTR

**All of the above**

Bottom of Form

**Which of the following is NOT a unit of user management in UCP?**

Organizations

**Crews**

Users

Teams

Docker Trusted Registry

DTR allows us to store and manage our docker images, it is like docker hub, however with one major difference, in that we can store our images on a host of our choice so this could be either on premises or somewhere in the cloud where we can apply our own compliance constraint and security requirements.

Mention the ucp node name on which dtr is being installed and disable tls verification for ucp.

You cannot delete a repo in which tags are immutable.

**What is the format to write the full name of repository in DTR?**

**<dtr\_domain\_name>/<user\_or\_org>/<repository\_name>**

<dtr\_domain\_name>/<repository\_name>

<dtr\_domain\_name>/<repository\_name>/<Image\_name>

None of the above

**Which feature of Docker EE provides us the flexibility of choosing host architecture and security policies for storing our Docker images?**

**DTR**

UCP

Docker Swarm

None of the above

Bottom of Form

Bottom of Form