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**

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

Bottom of Form

Bottom of Form