**Docker**

reply2bhargav94@gmail.com

**Intro:**

1. Docker is nothing but a containerization tool.
2. It is designed to make it easier to create, deploy and run applications with all the parts it needs, such as libraries and other dependencies and ship it out as single packet. (Docker container will contain an image, applications and their dependencies)

**Scenerio1**:

* We developed an android project which has dependencies like java, tomcat.
* We must deliver the project to client.
* It is easier to containerize the application, and its dependencies as a container rather than asking the client to install java and tomcat in their machines. The clients may not have software knowledge.
* Let us assume that we have 1000 clients and we need to update the project every month.
* We will simply update the original container that we have and send the container to all the clients.

**Scenerio2**:

* The product which we developed works on our machine and it doesn’t work on other developer’s system.
* It may be because we might be running the application on Linux environment which is developed for windows environment.

**Things to know about docker**:

* Docker simply creates a container which can be used in an existing operating system under the same hardware.
* We can create any number of containers as it is very light in weight.
* Its like working on a separate new machine.
* We can run it anywhere (virtual, cloud)
* We can copy the container. (We can port the container we developed to client)
* Docker is a platform for developing, running and shipping.

**Difference between virtualization (Vagrant tool) and containerization**:

* Virtual concept needs complete OS and shares systems RAM to work (which makes a system slow if we create more virtual OS’s). Although it does the same job as docker does considering options like “light weight and create any number of containers” and “can run on any os and hardware” we go for docker.
* Docker doesn’t need complete OS to run. It will contain only base image i.e basic file system and bootfs.

**Docker Components**:

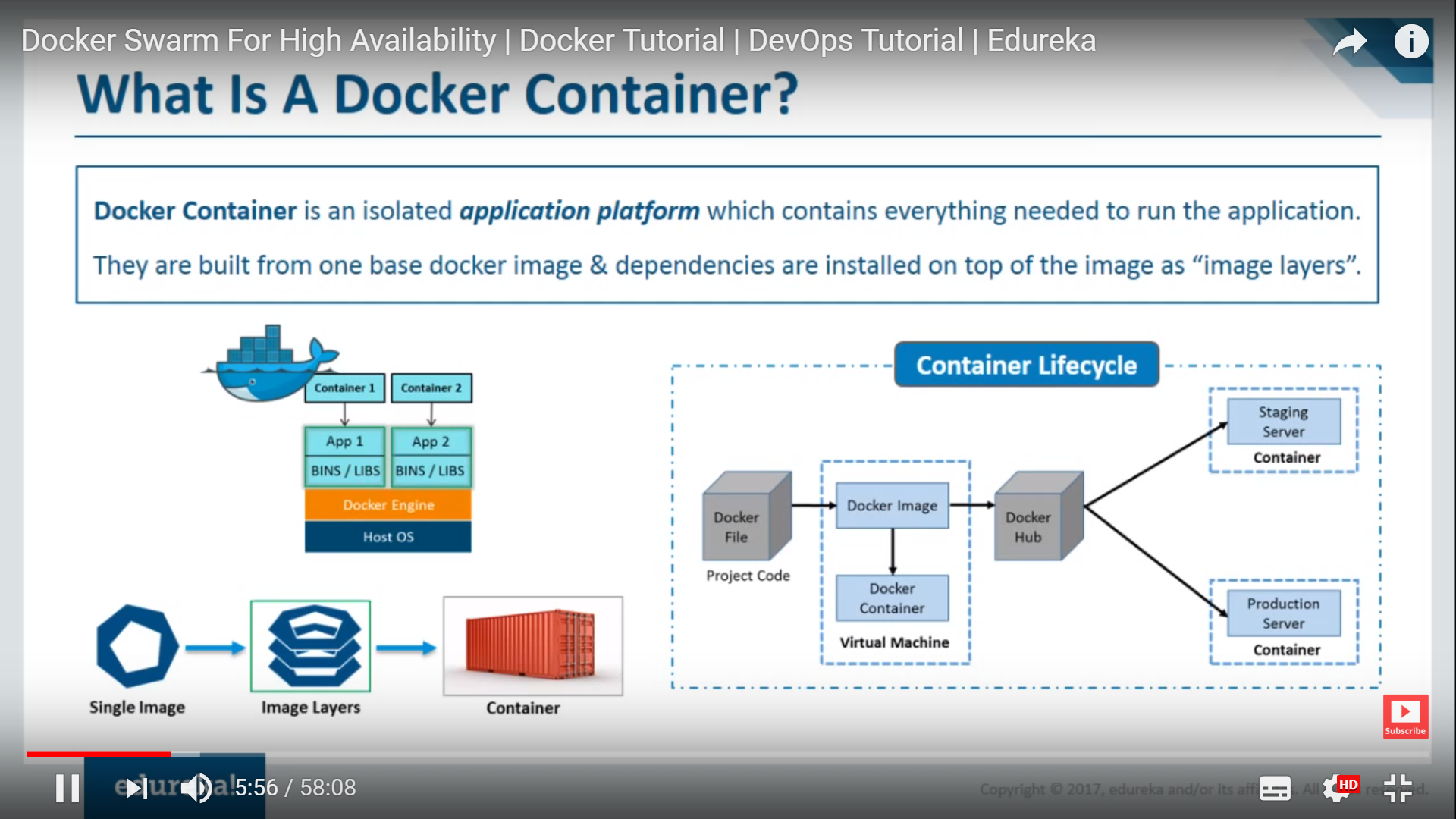
**Core components**:

* **Docker Daemon**: Process on which we try to run docker. It runs on host machine.
* **Docker Client**: Interface to interact with daemon. (bash)

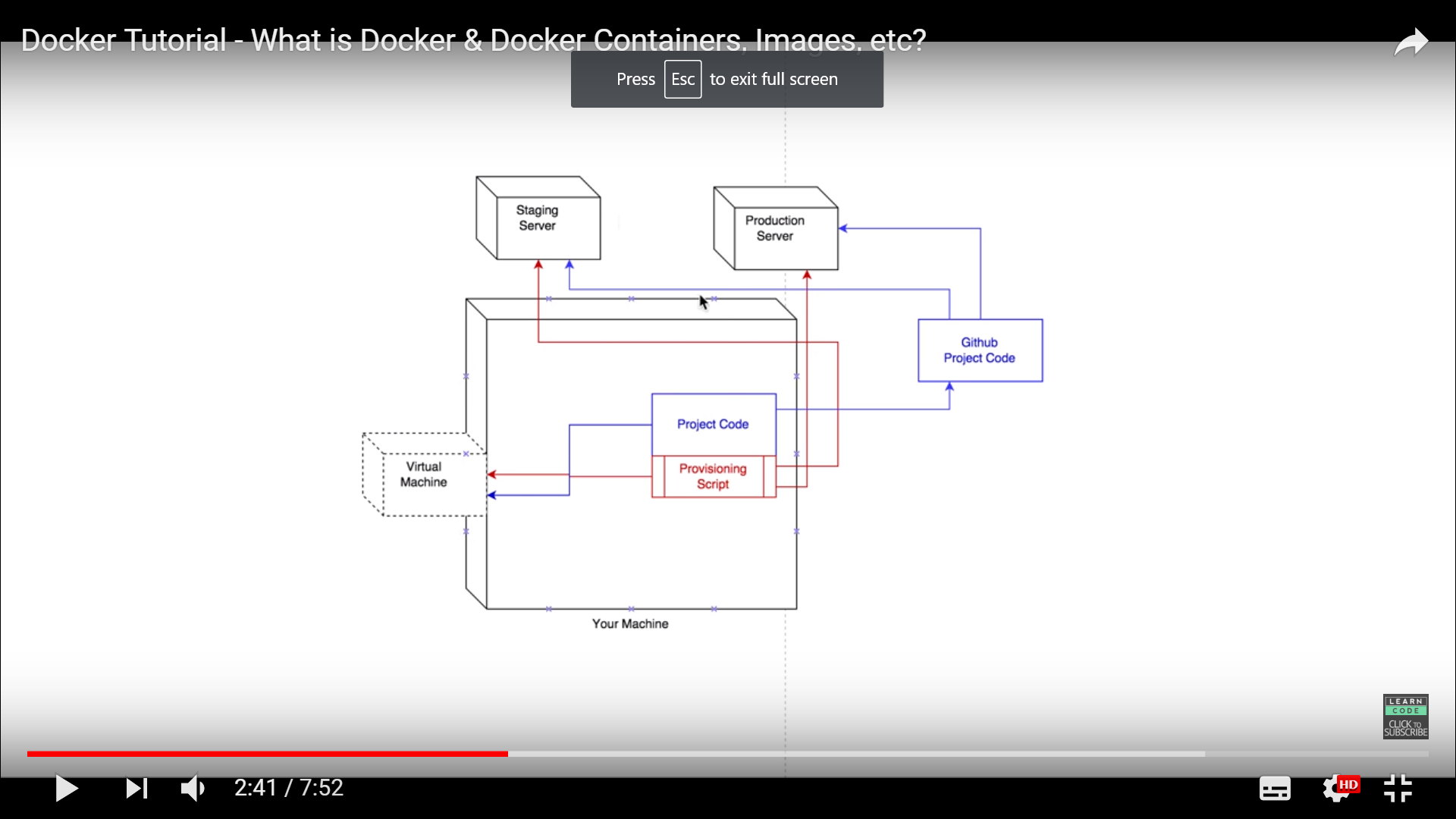
**Workflow components**:

* **Docker image**: Holds the environment and applications.
* **Docker container**: Created from images (It is an isolated application platform which contains everything needed to run the application. They are build from one base image and dependencies are installed on top of the image as “image layers” )
* **Docker Registry**: Store images
* **Docker file**: Create our own image.

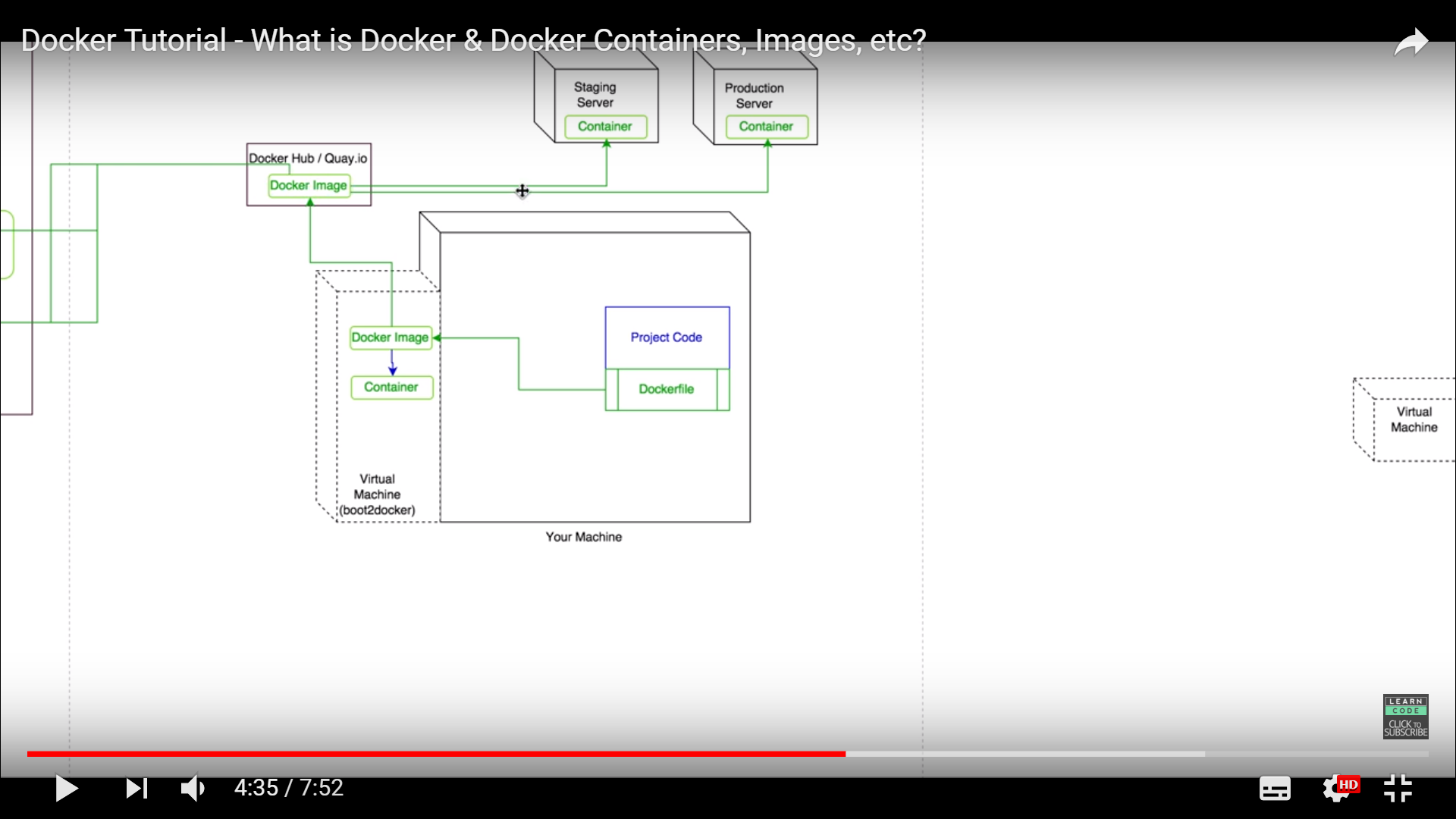
**Understanding Container:**



* We write the requirements of the image in docker file.
* An image is created which is not yet active.
* When we run the image, a container is created which is where the service is running and it is the actual instance.
* But, we cannot upload the container into docker hub , what we upload is the docker image.

**Vagrant Structure**:

* Vagrant is nothing but a virtual box (virtualization).
* To make the developed product run in different phases of development such as staging server and production server, we need to create a **virtual environment** like that of those servers and run the product in that **VE and test the product.**

**Docker Structure**  


* The docker file contains the required os, applications, and other dependencies and running the docker file creates docker image and running the image creates a container.
* Now, the container consists the developed product.
* We can run the container with product on any os under any hardware. The only requirement is that the machines should have docker service installed for the docker container to run.

**Hands-on: Regular Commands and their usage**

**Command:**

**(searching from existing docker hub, pull, run, create container)**

1. docker search [OPTIONS] TERM

**Eg: docker search hello-world**

**docker search --stars=3 hello-world** (Display results with 3 or more stars only)

1. docker images [OPTIONS] [REPOSITORY[:TAG]]

**Eg: docker images**

**docker images hello-world**

1. docker pull [OPTIONS] NAME[:TAG|@DIGEST]

**Eg:** **docker pull images**

**docker pull images hello-world**(and we can pull number of images at a time using different parameters and this command just downloads the image into ur machine)

1. **docker pull myregistry.local:5000/testing/test-image** (By default docker pulls the images from docker hub. Instead of that we can specify the path from where we need to pull the images.)
2. **docker run hello-world** (This commands pulls the image if it is not present locally as well as runs the image)(What happens when u run this command)

* The Docker client contacted the Docker daemon.
* The Docker daemon pulled the "hello-world" image from the Docker Hub.(amd64)
* 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.
* **Eg: docker run Jenkins** (this command runs the Jenkins through docker container, but we cannot open the Jenkins in browser.)
* **docker run -p 8080:8080 jenkins** (we have to mention the port number on which it runs, then we can open in browser using port ipaddress:8080)

6. **docker ps** (To know the running containers)

**docker ps -a** (To know the stopped containers)

7. **docker rm container-id container-id ….** (In order to remove container, it should be stopped first)

**docker rm -f container-id …..** (To force remove containers)

8. **docker run -d --name web1 -p 8080:8080 jenkins**(To run the Jenkins in **background** with name using port number)(Exposing port containers to allow service to run)

**docker run -d --name web2 -p 8081:8080 jenkins** (To run another Jenkins on another port using different name)

Also try **docker run -it –name test ubuntu /bin/bash**

9. **docker stop containername** (To stop the container from running) we can use start to make container run.

**docker stop web1**

10. **docker run -it image** (run image in interactive mode)

**docker run -it ubuntu**

11. **docker rmi imagename** (To remove images)

**docker rmi Jenkins**

12. **docker system prune** (To remove all resources — images, containers, volumes, and networks) not associated with a container

**docker system prune -a** (To remove all unused images and any stopped containers)

**Note :(Understanding unused and dangling images)**

* An unused image means that it has not been assigned or used in a container. For example, when running docker ps -a - it will list all your exited and currently running containers. Any images shown being used inside any of containers are a "used image".
* On the other hand, a dangling image just means that you've created the new build of the image, but it wasn't given a new name. So, the old images you have becomes the "dangling image". Those old image are the ones that are untagged and displays "<none>" on its name when you run docker images.
* When running docker system prune -a, it will remove both unused and dangling images. Therefore any images being used in a container, whether they have been exited or currently running, will NOT be affected.

13. **docker run -it –name test1 ubuntu /bin/bash**

* Create test.txt using touch command
* Exit

**docker cp test1:/test.txt (**To copy file from container**)**

14**. For Docker login, push, pull images, tag concept, save image to local and load image from local**.

* Create an account in <https://cloud.docker.com>
* Create a repository.
* Login to that account in docker machine using command **docker login --username=\*\*\*\* --password \*\*\*\***
* Create an image using base image or create a new image using docker file.
* Give a tag to the developed image using command **docker tag imageid username/createdrepository:tagname**

We generally use tag to give image a description like version and release version.

* Push the image into docker cloud or hub using command

**docker push username/createdrepo**

* We can pull the image from our own docker registry using command:

**docker pull username/repository:tagname**

* We can save the repository into our local machine also instead of uploading it into docker hub using command

**docker save bhargav30/testingdocker > testingdocker.tar**

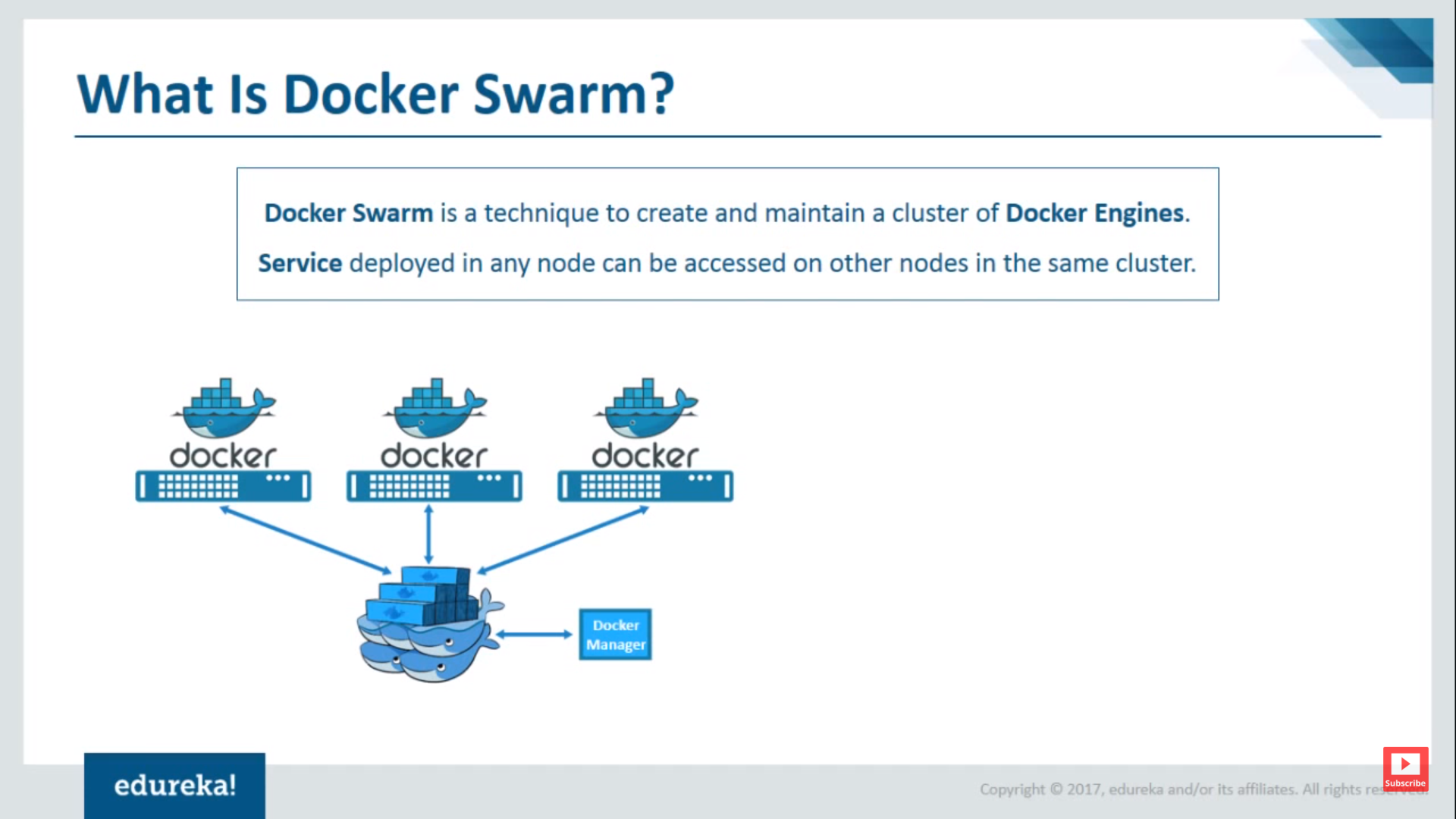
* We can load the repository from local into docker using **docker load --input testingdocker.tar**

15. **Docker detach and attach**

* **docker run --name test -d -it jenkins** (This command runs Jenkins in background. It gave container name as test)
* **docker attach test or containerid** (This command brings Jenkins foreground)

**Docker swarm for high availability**

* High availability refers to a system that is continuously operational for a desirably long length of time. Simply saying **100% operational** and **never failing**.



**Features:**

* **High-availability of services**: If a node goes down the other node takes place and the docker manager looks after the issue.
* **Auto load-balancing**: apart from node balancing the services it helps in scaling of services too i.e if I have 10 services, 9 services will be taken by 3 nodes from the figure and one service will be running by docker manager too. This feature is automatic in docker.
* **Decentralized access**: We can access docker manager or node on any server using ssh login.
* **Easy to scale-up deployments**: If I have 10 services, 9 services will be taken by 3 nodes from the figure and one service will be running by docker manager too. This feature is automatic in docker. We can increase the services and this feature looks after those with a single command.
* **Rolling updates**:We can specify a delay for any updates which will update one node at a time which helps for high availability.

**Hands On**

**Note:** The machines under docker swarm must allow certain protocols(traffic) to work normally. Here I allowed all traffic in security configure phase in aws. If you allow only ssh traffic, swarm won’t work.

* **docker swarm init --advertise-addr <ip address of machine we want to make as leader>** This command initializes a swarm mode and make a node as manager or leader as well as gives us code which can be used in attaching a machine into swarm.
* Now we must add nodes to the swarm. Since we have the code go and paste the code in node machines.
* We can check the status of nodes using command in leader machine:

**docker node ls** (or)To know status use:

**docker info** (To know information of docker in a machine)

* Use Command **setenforce 0** to give permission to docker swarm to run services through it. (This command requires more attention)
* **docker service ls ----** check for services --- we will find it empty

**Task 1:**

* Here we are trying to run Jenkins in docker swarm in manager machine. After successful running of Jenkins check in browser by giving ipaddresses of manager and node with port 8080. Even though we have just installed service in manager machine, we can still access service through workers ip. We may get error, try to solve it. Or scroll down for a solution.
* **docker volume create --name jenkins\_home**
* **docker service create --replicas 1 --name jenkins -p 8080:8080 -p 50000:50000 --mount source=jenkins\_home,dst=/var/jenkins\_home jenkins:alpine**
* **docker service ls**
* **docker service ps jenkins**
* **docker ps**
* **Open browser and test if service is running or not on all the nodes.**

1. **Note:**  Both docker service ps and docker ps commands must show that Jenkins is up and running. Test the same in browser.
2. **Note:** Always check node details using command docker node ls and check for status and availability as active. If u find nodes down remove them from swarm and add them again.

* A node can leave a swarm using command **docker swarm leave**
* For leader node or manager node use **docker swarm leave --force** to leave a swarm and the nodes become leaderless.
* We can even change the leader.

**Solution if the running service cannot be accessed through worker ip:**

* **docker network create -d overlay my\_network ----** creates an overlay network.
* **docker service create --replicas 2 --network my\_network --publish 8080:80 --name website yeasy/simple-web -----** connecting service through overlay network.

We can increase replicas using command:

* **docker service update --replicas 10 website**
* **docker service ps website (**if we observe the output of this command we can see the three nodes share the 10 replicas or services**) creating high availability, load balancing automatically.**
* Let’s see **scaling concept**. What if a node goes offline? Turn off a node and set the state=stop. Check for **docker service ps website,** even though a node goes down we can see the 10 replicas running shared by manager and the other node themselves.

**Volumes** (Base concept)

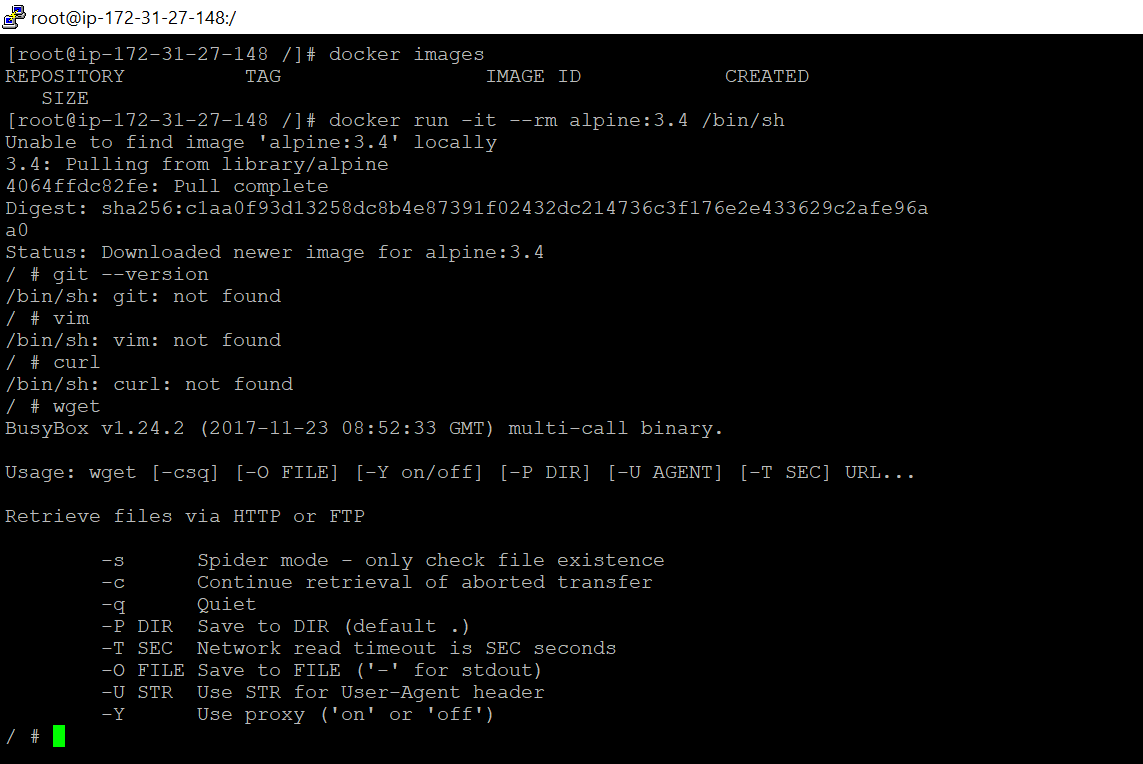
* It is very important concept related to containers.
* What happens when a container is removed? All the data gets deleted.
* To save the data for any future references or usage we go for volumes.
* Even after removing the containers, the volumes we create contains the data.
* For better understanding, we will run Jenkins service with our own created volume and test whether it holds the data or not.

**Hands on**

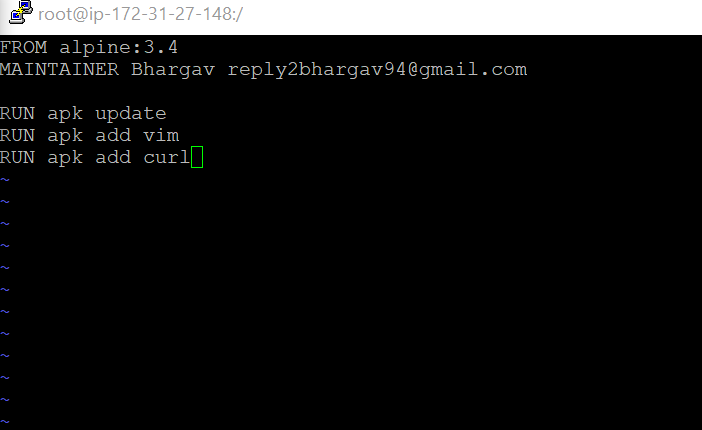
* **docker volume create myvol1** --- create a volume
* **docker volume ls** --- Check if volume is created or not
* **docker volume inspect myvol1** --- see volume details like location of volume created.
* **docker run --name myjenkins -v myvol1:/var/jenkins\_home -p 8080:8080 -p 50000:50000 jenkins** ---- This command runs Jenkins service on port 8080 with api’s using port 50000 and what ever data that is stored in Jenkins\_home , is being assigned to myvol1.
* Open Jenkins in browser and create a job.
* **docker volume inspect myvol1** --- Use this command and we can see the location of myvol1 in mountpoint section. In my case it is: **"Mountpoint": "/var/lib/docker/volumes/myvol1/\_data"**
* Goto **\_data folder** and we can see the Jenkins data.
* Even if we delete the Jenkins service we can still see the **myvol1** volume and its data can be used.
* **docker volume rm volume-name** --- This command removes docker volume by name
* **docker volume prune** --- This command removes all volumes which are not under use.

**Docker File (**with dangling image concept**)**

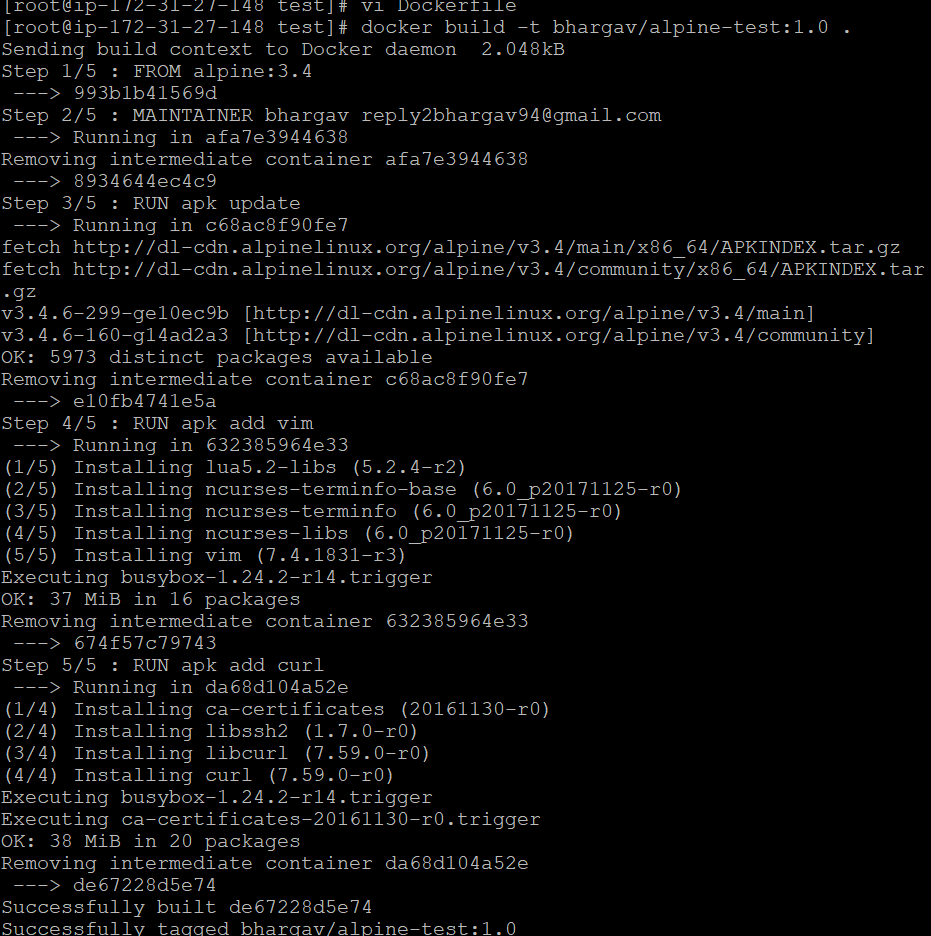
* Docker file is used to create our own images. This can be done using base image or completely from scratch.



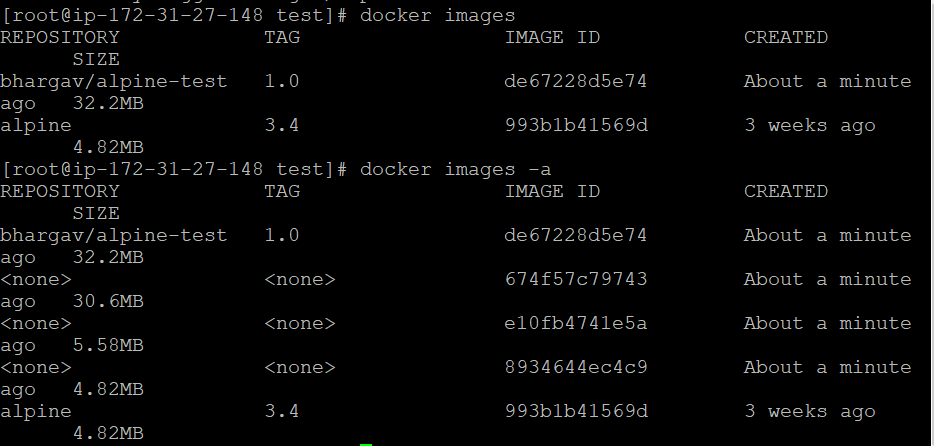
* Here we don’t have any local images.
* We are trying to run alpine:3.4 image. **- - rm helps in removing containers if they are stopped.**
* Vim, git, curl are not installed in base alpine image. Let us install them using Dockerfile.



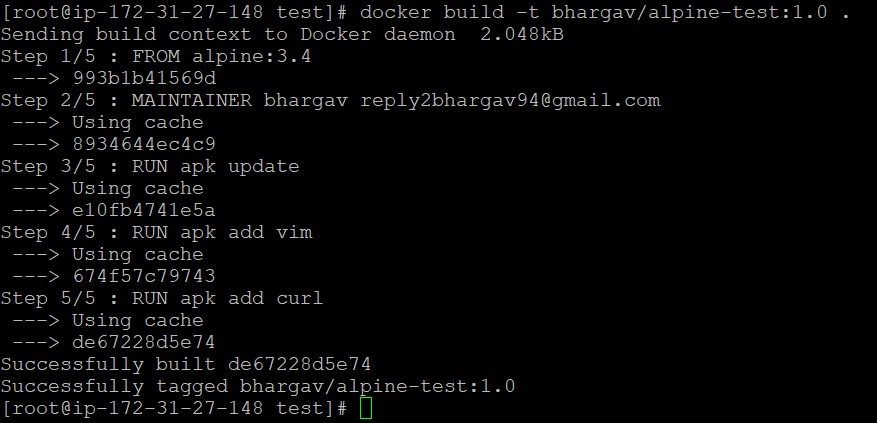
* Create a folder and inside the folder create a file named **Dockerfile** and place the code as shown in figure in order to install vim and curl



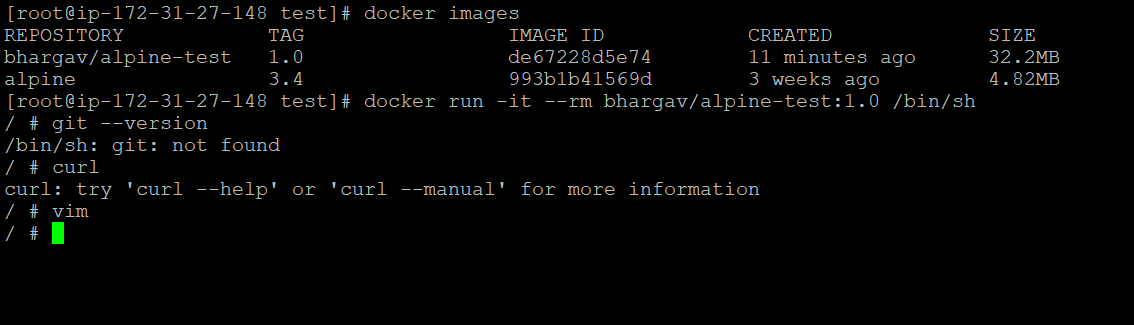
* Run the build docker command to create an image with tag using option **-t**
* We can observe that the process completed in 5 installations which is equal to our command lines in Dockerfile i.e each command is executed as a separate image and finally merged.



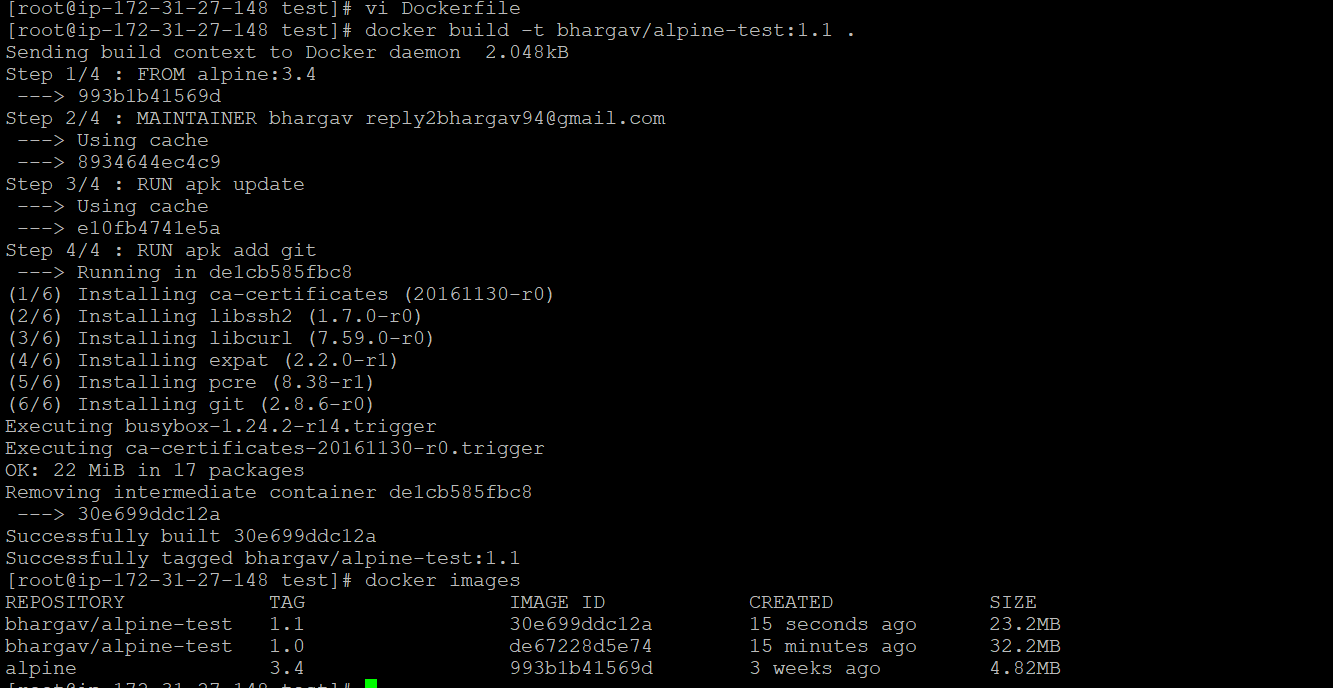
* Using command **docker images -a** we can see intermediate images.



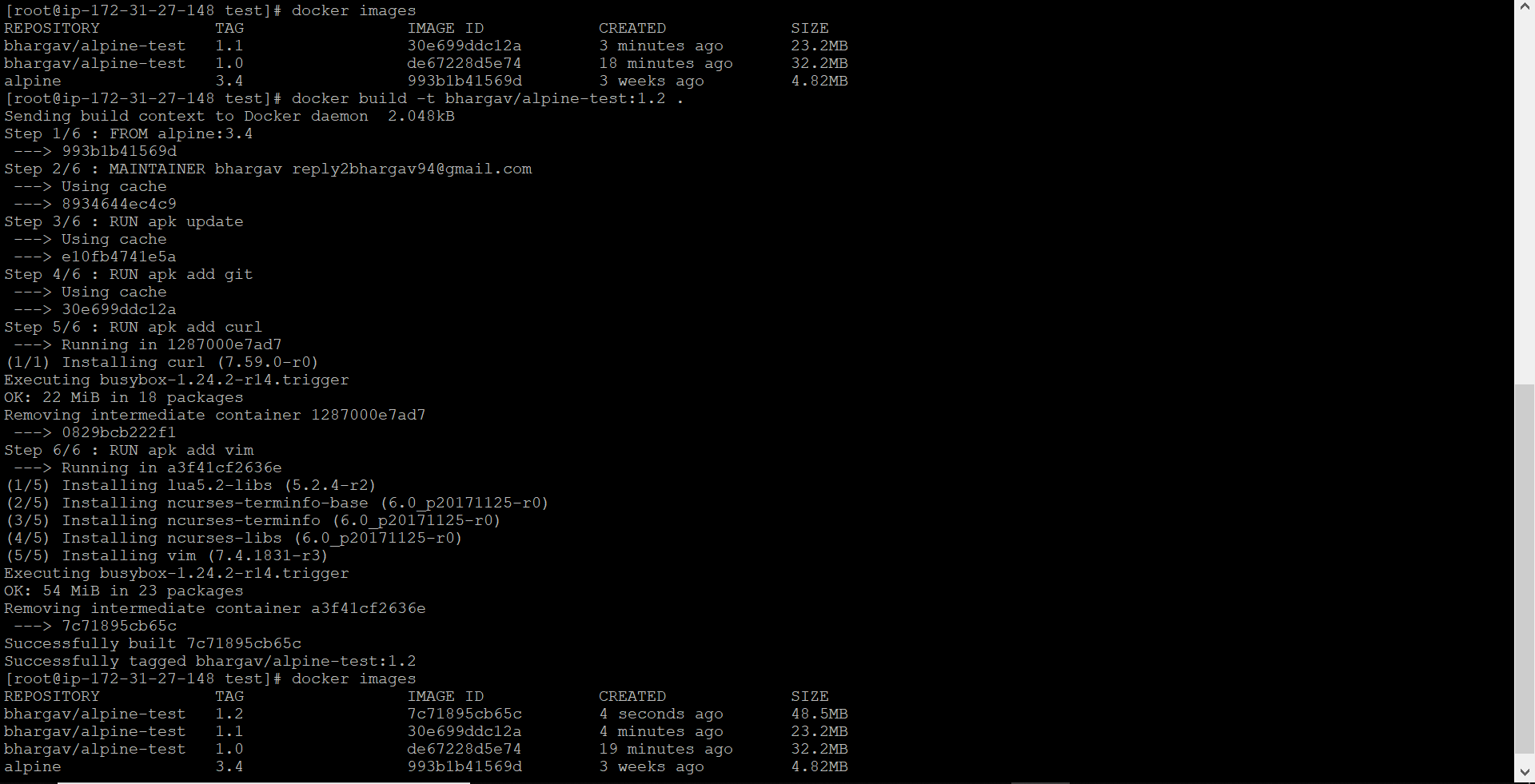
* If we build the same Dockerfile without any modifications, we can observe docker uses cache as they are already installed.

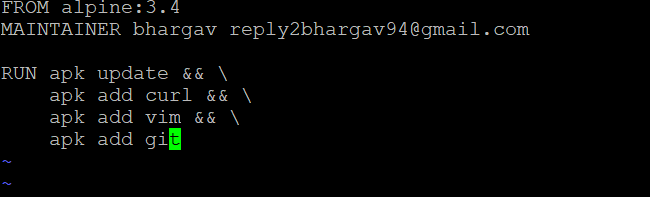


* Test for curl, vim installations by running our image

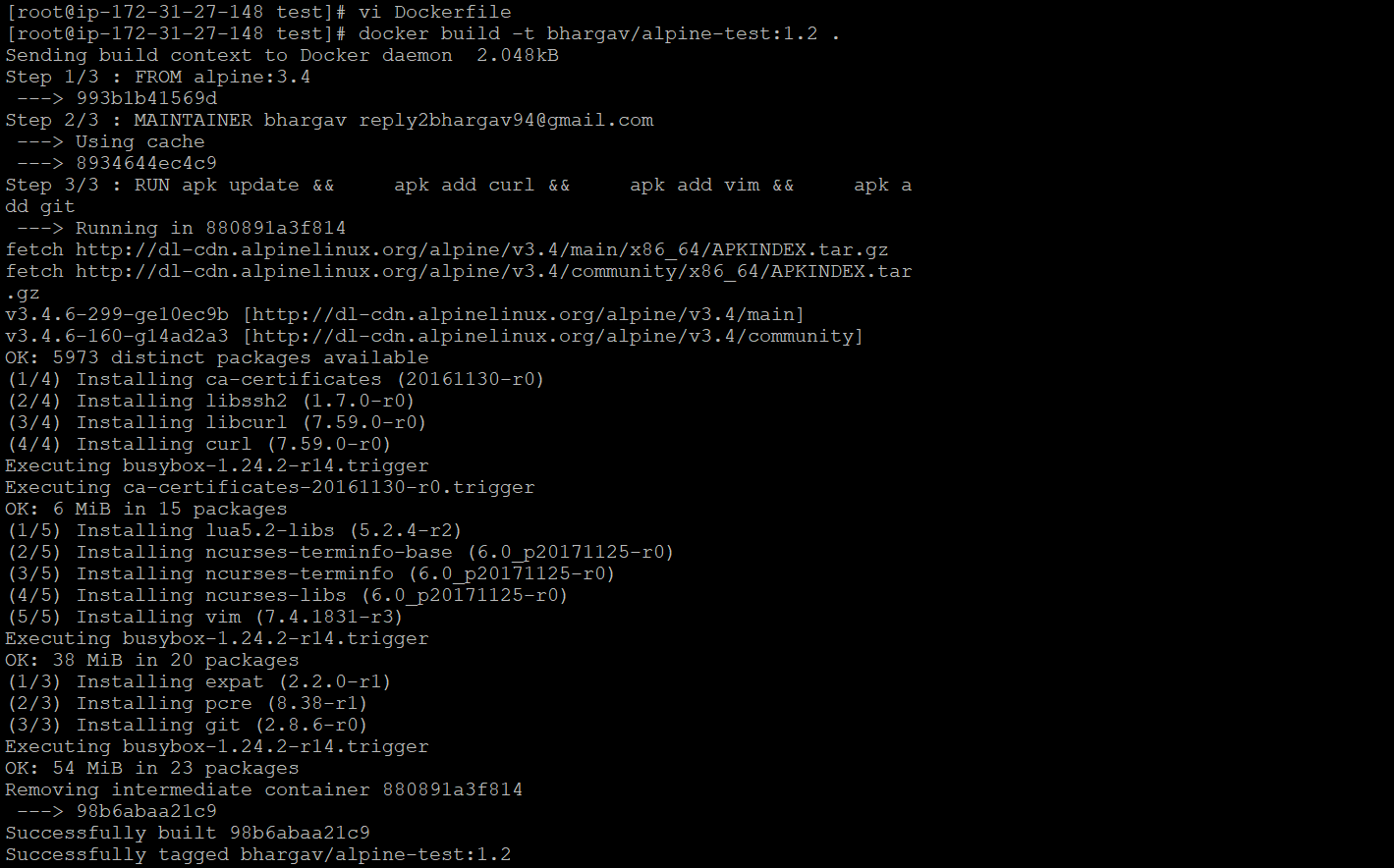


* Add git in Dockerfile and try build. Give different tag name.

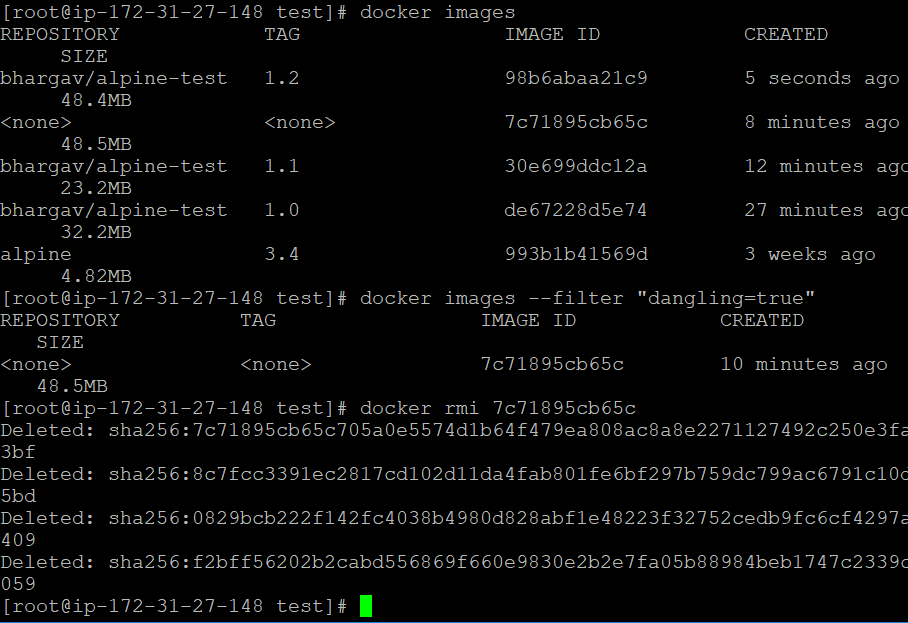




* In order to eliminate intermediate images , we can write dockerfile as above.



* Build the dockerfile with same tagname as previous.



* We can see an image with repo and tag as **none which is a dangling image and is not used anymore since we have our latest image. It is created as we have used the same tag name in last step.**
* We can remove dangling images using command mentioned in figure.
* As shown above, we can write dockerfile in many formats as required.

**Thank You**