Docker

- **Introduction of Docker** :
  Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. Build once, run anywhere . A clean, safe, hygienic and portable runtime environment for your app. No worries about missing dependencies, packages and other pain points during subsequent deployments. Run each app in its own isolated container, so you can run various versions of libraries and other dependencies for each app without worrying. And eliminate inconsistencies between development, test, production, and customer environments

- **Steps to install Docker** :

  1. To update apt, issue the command:

```
$ sudo apt update
```

  2. Mostly the docker package is available in all linux systems. So by below command you can install docker

```
$ sudo apt-get install docker.io
```

  3. If you find any problem in the above command you will have to run an installation script provided by Docker.

```
$ sudo apt-get install wget
$ wget -qO- https://get.docker.com/ | sh
```

  4. If you want to be able to run Docker containers as your user, not only as root, you should add yourself to the group called docker using the following command.

```
$ sudo groupadd docker
$ sudo usermod -aG docker $USER
```

  Then you'll want to log out and then log back in for the changes to take effect.

- **Docker components** :

1. **Docker images**
   - ➔ A Docker image is a read-only template. For example, an image could contain an Ubuntu operating system with Apache and your web application installed.
   - ➔ Images are used to create Docker containers. Docker provides a simple way to build new images or update existing images, or you can download Docker images that other people have already created.
   - ➔ Docker images are the build component of Docker.

2. **Docker File**
   - ➔ Docker can build images automatically by reading the instructions from a Dockerfile.
   - ➔ A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.
   - ➔ The docker build command builds an image from a Dockerfile and a context.

3. **Docker Containers**
   - ➔ Executing an image is called “container” .
   - ➔ Containers, in short, contain applications in a way that keep them isolated from the host system that they run on.
   - ➔ Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package.

4. **Docker Hub**
   - ➔ It is docker registry which holds images.
   - ➔ These are public or private stores from which you upload or download images.
   - ➔ The public Docker registry is provided with the Docker Hub. (https://hub.docker.com)
   - ➔ It serves a huge collection of existing images for your use. These can be images you create yourself or you can use images that others have previously created.
   - ➔ Docker registries are the distribution component of Docker.

- **Docker Architecture :**

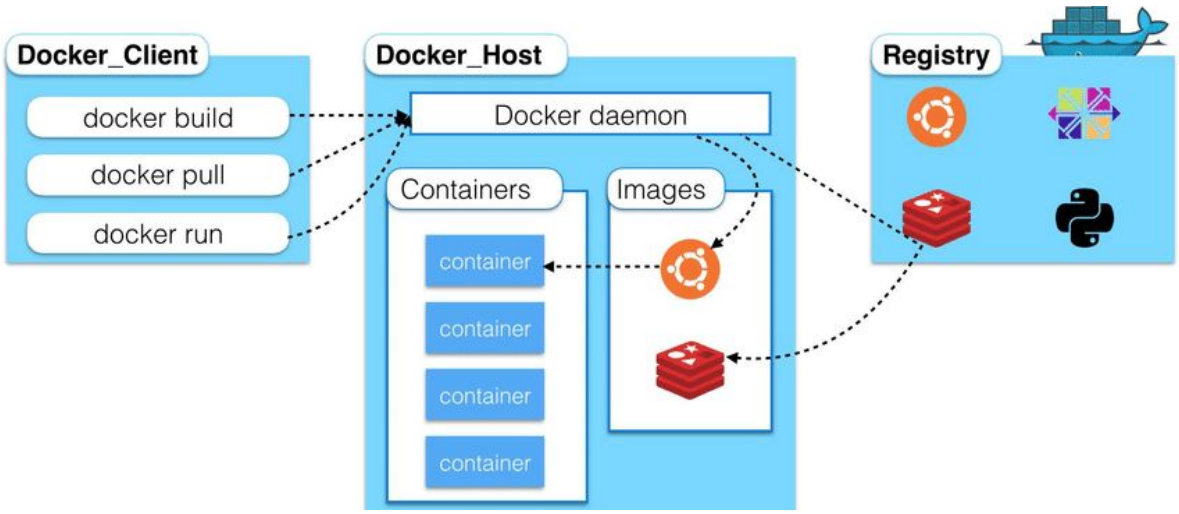

Fig. 13.C.1 : Docker architecture

- **Docker Commands** :

1. **docker pull ubuntu**
   - ➔ Executes "docker pull Ubuntu"
   - ➔ This pulls(downloads) an image from the docker library
   - ➔ The image contains bare copy of ubuntu

2. **docker images**
   - ➔ Executes "docker images"
   - ➔ This generates a list of images known to Docker on your machine

3. **docker run ubuntu**
   - ➔ Executes "docker run Ubuntu"
   - ➔ This executes an image called ubuntu. An executing image is called a "container".
   - ➔ You are now inside the container.
   - ➔ Execute "ls".
     A directory structure is set up (The files responsible for the container).

4. **docker ps –a**
   - ➔ Executes "docker ps –a"
   - ➔ This generates a list of all of the containers (both running and not-running).

5. **docker ps**
   - ➔ Executes "docker ps"
   - ➔ This generates a list of all running containers.

6. **docker rm <container_name (or) id>**
   - ➔ Executes “docker rm <container_name (or) id>”
   - ➔ This will remove the container specified. You can remove only stopped containers. (‘docker stop <container_id>’)

7. **docker exec -it <container_id (or) name> /bin/sh**
   - ➔ Executes “docker exec -it <container_id (or) name> /bin/sh”
   - ➔ You are now inside the container.
   - ➔ Execute “ls”. A directory structure is set up (The files responsible for the container).

8. **docker rmi <image_name (or) id>**
   - ➔ Executes “docker rmi <iamge_name (or) id>”
   - ➔ This will remove the image specified.

9. **docker logs -f <container_id (or) container_name>**
   - ➔ Executes “docker logs -f <container_id (or) container_name>”
   - ➔ To see all the logs of a particular container until the command executed and also ongoing logs .

- **Sharing image**

  Multiple team members may wish to share images. Images can be in production, under development or under test. Docker Hub is a repository where images can be stored and shared. You can also store images in your private registry . Each image is tagged to allow versioning Any image can be “pulled” to any host (with appropriate credentials). Tagging as “latest” allows updates to be propagated. Pull :latest gets the last image checked into repository with that name.

  Sharing involves mainly in three different types :

  (a) Sharing via docker hub :
  - ➔ Create a account in dockerhub (hub.docker.com). And remember your username and password.
  - ➔ Now go to terminal and run the command “docker login” .And then enter your username and password to login.
  - ➔ Then tag your image to a new name such that it should be in the format like {username}/{imagename:tag}

```
$ docker tag your_imagename:tag your_username/imagename:tag
```

  - ➔ Now push your image to docker hub.

```
$ docker push  your_username/imagename:tag
```

- ➔ Now the docker hub has your image.
- ➔ You can also push your same image with different tag.So that it will push Into same image repository with different tag. And then the repository contains different images with same imagename and different tags.You call pull the image by differentiating the tag.
- ➔ Now the person who needs to pull that image can pull with the below Command.

```
$ docker  pull  <usernameofimageholder>/<imagename:tag>
```

(b) Sharing via Private repository :

You might have the need to have your own private repositories. You may not want to host the repositories on Docker Hub. For this, there is a repository container itself from Docker.

- ➔ Use the Docker run command to download the private registry. This can be done using the following command :

```
$ docker pull registry:2
```

'registry' is the container managed by Docker which can be used to host private Repositories.

- ➔ And it is better to mount a volume which keeps a backup of all your pushed Images. So that if the registry container is stopped it will restore all your pushed images whenever it restarts.

```
$ mkdir registry_backup
$ cd registry_backup
```

and run the registry image along with mounting :

```
$   docker run -d -p  9999:5000 -v  $(pwd)/:/var/lib/registry registry:2
```

Let's do a 'docker ps' to see that the registry container is indeed running

Now your private registry setup is done.

- ➔ Now let's tag one of our existing images (let's take centos) so that we can push it to our local Repository.

```
$ docker tag centos  localhost:9999/centos
```

➔ Now let's use the Docker push command to push the repository to our private Repository.

```
$ docker push localhost:9999/centos
```

➔ You can share your private registry image to others also.To do that go to the file /etc/docker/daemon.json and add this line.

```
{
        "insecure-registries" : [ "<your_IP_address>:9999" ]
}
```

And in the system from where you need to pull this private image also add the above line in /etc/docker/daemon.json in its system.(The ip address which is mentioning here should be the image holder's ip address.)

➔ After that you can pull that private image with the following command.

```
$ docker pull <ipaddressofimageholder>:9999/centos
```

(c) Share image via .tar file

➔ Sometimes I want to save a docker image and then use it on another computer without going through the hassle of uploading it to docker hub or private registry

➔ Docker images aren't really stored as files that you can just grab and move to another computer .But you can save and then load the images like this:

If you want to save the image as a tar archive, using docker save -o:

```
$ docker save -o  abcd.tar  <image_name:tag>
```

Then copy the adcd.tar file into another computer and load there :

```
$ docker load abcd.tar
```

➔ Run " docker images " . It will show up in your docker images list .

Docker clustering :

- cluster is a group of docker-enabled nodes . Each node has either a manager or worker role in the cluster. At least one master node is required for a cluster to operate.
- manager refers to the node maintaining the state of the cluster. There can be one or more managers in a cluster.

- worker refers to the node sharing the workload in the cluster.

Docker swarm:
With Docker Swarm, you can create and manage Docker clusters. Swarm can be used to distribute containers across multiple hosts.

Cluster initialisation :

```
$ docker swarm init
```

- This node will become the manager node.

Join cluster :

```
$ docker swarm join-token manager     (To join the cluster as manager)
$ docker swarm join-token worker      (To join the cluster as worker)
```

The token used here is generated when that cluster is initialized.

Leave cluster :

```
$ docker swarm leave       (For worker )
$ docker swarm leave -f    (For manager)
```

- The node leaves the cluster is NOT removed automatically from the node table. Instead, the node is marked as Down. If you want the node to be removed from the table, you should run the command **docker node rm**.

In swarm cluster, a service is created by deploying a container in the cluster. The container can be deployed as a single instance (i.e. task) or multiple instances to achieve load-balancing.
Services are really just "containers in production." A service only runs one image, but it codifies the way that image runs—what ports it should use, how many replicas of the container should run so the service has the capacity it needs, and so on.
It's very easy to define, run, and scale services with the Docker platform, just write a docker-compose.yml file.A docker-compose.yml file is a YAML file that defines how Docker containers should behave in production.

Ex: (docker-compose.yml)

```
version: "3"
services:
  web:
    image: username/repo:tag
    deploy:
      replicas: 5
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
      restart_policy:
        condition: on-failure
    ports:
      - "4000:80"
    networks:
      - application_default
networks:
  Application_default:
```

- (replicas:5) - Run 5 instances of "repo" image as a service called "web". Resulting to run that image as 5 containers which helps to distribute the incoming requests among them for load-balancing.
- Restart_policy - Immediately restart containers if one fails.
- Network - Instruct web's containers to share port 80 via a network called "application_default".
- Define the webnet network with the default settings

Create app with that service :

$ docker stack deploy -c docker-compose.yml <app-name>

➔ single service stack runs replicas no.of container instances of our deployed image on one host.

-To list all the services running on the host.

```
$ docker service ls
```

-To list all the stacks with no.of services in that host.

```
$ docker stack ls
```