**What is Docker?**

Docker is an open-source lightweight containerization tool. It allows you to automate the deployment of applications in lightweight and portable containers.

that is used for building, deploying, and running applications

**What are the advantages of using Docker container?**

Here, are a major advantage of using Docker.

* Offers an efficient and easy initial set up
* Allows you to describe your application lifecycle in detail
* Simple configuration and interacts with Docker Compose.
* Documentation provides every bit of information.

**What are the main drawbacks of Docker?**

* Doesn't provide a storage option
* Offer a poor monitoring option.
* No automatic rescheduling of inactive Nodes
* Complicated automatic horizontal scaling set up

Every docker images are stored in the Docker registry.

**What is Docker Engine?**

Docker daemon or Docker engine represents the server.

**Explain Registries**

There are two types of registry is

* Public Registry
* Private Registry

Docker's public registry is called Docker hub, which allows you to store images privately.

**What command should you run to see all running container in Docker?**

$ docker ps

**Write the command to stop the docker container**

$ sudo docker stop container name

\*\*docker stats …it will show CPU and memory utilization.

\*\*\*Service docker start (To start docker service)

\*\*\*Docker info

**How can you monitor the docker in production environments?**

Docker states and Docker Events are used to monitoring docker in the production environment.

**What is Virtualization?**

Virtualization is a method of logically dividing mainframes to allow multiple applications to run simultaneously.

**Where the docker volumes are stored?**

You need to navigate:

/var/lib/docker/volumes

**List out some important advanced docker commands**

|  |  |
| --- | --- |
| Command | Description |
| docker info | Information Command |
| docker pull | Download an image |
| docker stats | Container information |
| Docker images | List of images downloaded |

Q) How to run Jenkins on docker?

🡪 Find the image of Jenkins on docker hub.

Docker run -p 8080:8080 -p 50000:5000 -d -v Jenkins\_home:/var/Jenkins\_home Jenkins/Jenkins

\*\* Docker logs <container Id>

**What is the command you need to give to push the new image to Docker registry?**

docker push myorg/img

**Does Docker offer support for IPV6?**

Yes, Docker provides support IPv6. IPv6 networking is supported only on Docker daemons runs on Linux hosts. However, if you want to enable IPv6 support in the Docker daemon, you need to modify /etc/docker/daemon.json and set the ipv6 key to true.

**Can you lose data when the container exits?**

No, any data that your application writes to disk get stored in container. The file system for the contain persists even after the container halts.

**How can you run multiple containers using a single service?**

By using docker-compose, you can run multiple containers using a single service. All docker-compose files uses yaml language.

**What is the purpose of Docker\_Host?**

It contains container, images, and Docker daemon. It offers a complete environment to execute and run your application.

**What is a Container?**  
A container is a standard unit of software bundled with dependencies so that applications can be deployed fast and reliably b/w different computing platforms.

**Can you tell something about docker container?**

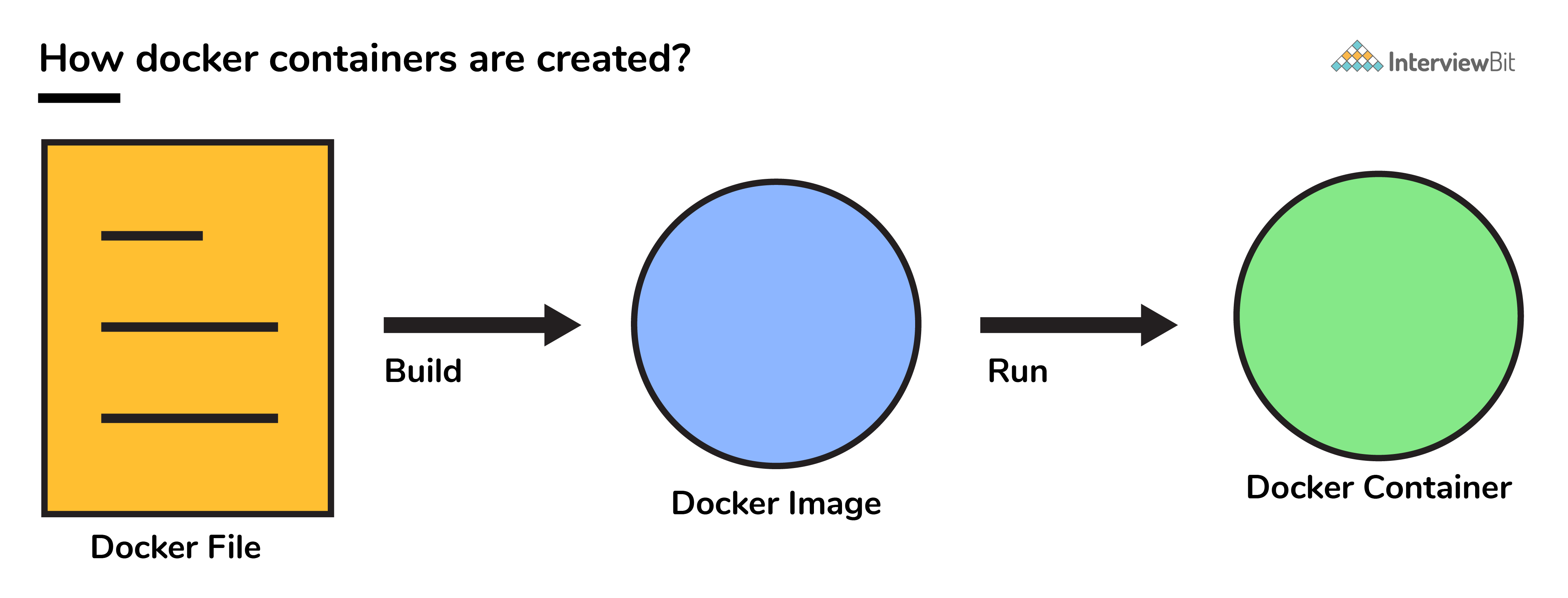
* In simplest terms, docker containers consist of applications and all their dependencies.
* They share the kernel and system resources with other containers and run as isolated systems in the host operating system.
* The main aim of docker containers is to get rid of the infrastructure dependency while deploying and running applications. This means that any containerized application can run on any platform irrespective of the infrastructure being used beneath.

### What are docker images?

They are executable packages(bundled with application code & dependencies, software packages, etc.) for the purpose of creating containers

**What is a DockerFile?**

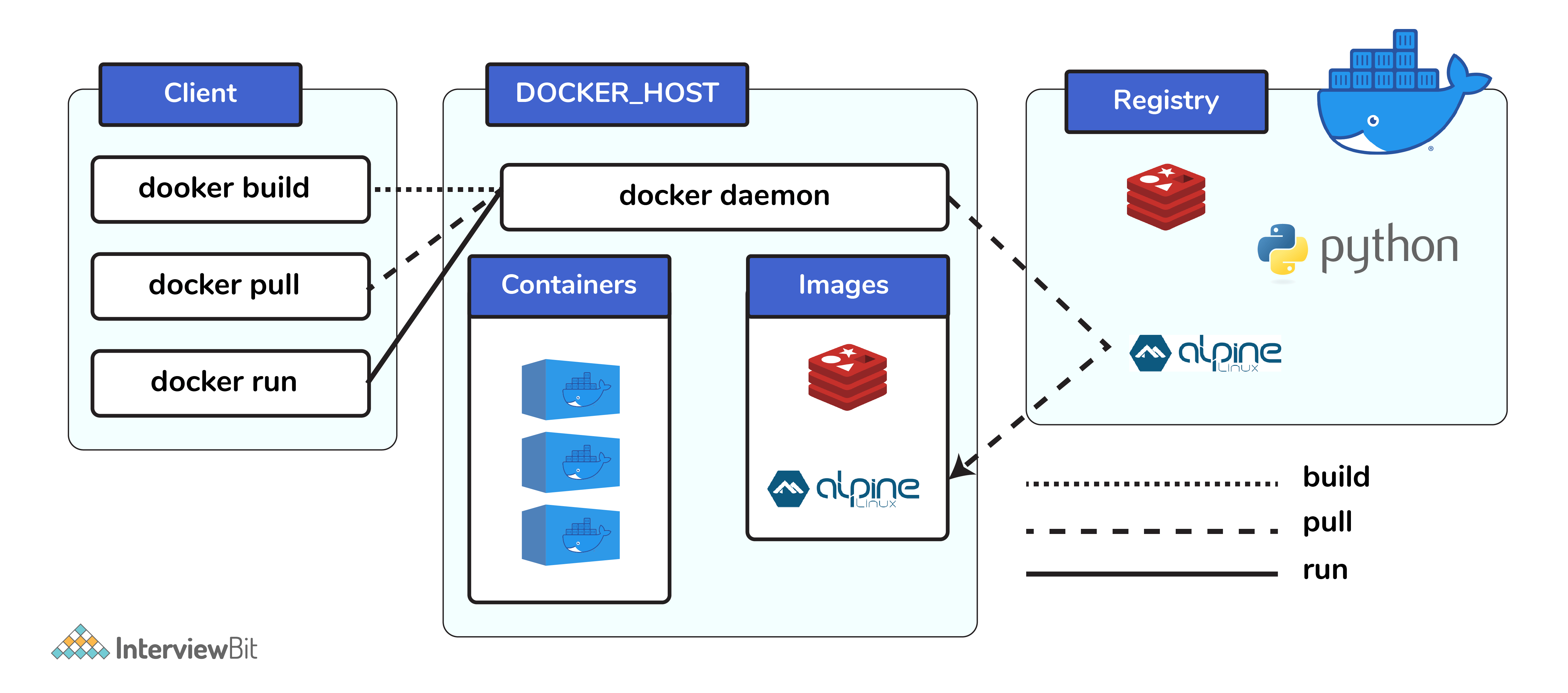
* It is a text file that has all commands which need to be run for building a given image.



**How many Docker components are there?**

There are three docker components, they are - Docker Client, Docker Host, and Docker Registry.

* **Docker Client:** This component performs “build” and “run” operations for the purpose of opening communication with the docker host.
* **Docker Host:** This component has the main docker daemon and hosts containers and their associated images. The daemon establishes a connection with the docker registry.
* **Docker Registry:** This component stores the docker images. There can be a public registry or a private one. The most famous public registries are Docker Hub and Docker Cloud.



**What command is used to check for the version of docker client and server?**

* The command used to get all version information of the client and server is the docker version.
* To get only the server version details, we can run docker version --format '{{.Server.Version}}

**Can a container restart by itself?**

* Yes, it is possible only while using certain docker-defined policies while using the docker run command. Following are the available policies:  
    
  1. **Off:** In this, the container won’t be restarted in case it's stopped or it fails.  
  2. **On-failure**: Here, the container restarts by itself only when it experiences failures not associated with the user.  
  3. **Unless-stopped:** Using this policy, ensures that a container can restart only when the command is executed to stop it by the user.  
  4. **Always:** Irrespective of the failure or stopping, the container always gets restarted in this type of policy.  
    
  These policies can be used as:  
  docker run -dit — restart [restart-policy-value] [container\_name]

### Where are docker volumes stored in docker?

Volumes are created and managed by Docker and cannot be accessed by non-docker entities. They are stored in Docker host filesystem at /var/lib/docker/volumes/

### What does the docker info command do?

The command gets detailed information about Docker installed on the host system. The information can be like what is the number of containers or images and in what state they are running and hardware specifications like total memory allocated, speed of the processor, kernel version, etc.

**List the most commonly used instructions in Dockerfile?**

* **FROM:** This is used to set the base image for upcoming instructions. A docker file is considered to be valid if it starts with the FROM instruction.
* **LABEL:** This is used for the image organization based on projects, modules, or licensing. It also helps in automation as we specify a key-value pair while defining a label that can be later accessed and handled programmatically.
* **RUN:** This command is used to execute instructions following it on the top of the current image in a new layer. Note that with each RUN command execution, we add layers on top of the image and then use that in subsequent steps.
* **CMD:** This command is used to provide default values of an executing container. In cases of multiple CMD commands the last instruction would be considered.

### What is the best way of deleting a container?

We need to follow the following two steps for deleting a container:  
- docker stop <container\_id>  
- docker rm <container\_id>

**What is the use of the**docker save**and**docker load**commands?**

Hide answer

A Docker image can be exported as an archive via the docker save command. For example:

docker save -o <container-export-path>.tar <container-name>

The exported Docker image can then be imported to another Docker host via the docker load command:

docker load -i <container-path>.tar

**What is the default Docker network driver, and how can you change it when running a Docker image?**

Hide answer

Docker provides different network drivers like bridge, host, overlay, and macvlan. bridge is the default.

### ****Explain Docker Architecture****

### ****How do you get the number of containers running, paused and stopped?****

You can use the following command to get detailed information about the docker installed on your system.

$ docker info

### ****How do you create a docker container from an image?****

Pull an image from docker repository with the above command and run it to create a container. Use the following command:

$ docker run -it -d <image\_name>

Most probably the next question would be, what does the ‘-d’ flag mean in the command?

**-d** means the container needs to start in the detached mode. Explain a little about the detach mode. Have a look at [this](https://www.edureka.co/blog/docker-commands/) blog to get a better understanding of different docker commands.

### ****Suppose you have 3 containers running and out of these, you wish to access one of them. How do you access a running container?****

The following command lets us access a running container:

$ docker exec -it <container id> bash

The exec command lets you get inside a container and work with it.

**How to start, stop and kill a container?**

The following command is used to start a docker container:

$ docker start <container\_id>

and the following for stopping a running container:

$ docker stop <container\_id>

kill a container with the following command:

$ docker kill <container\_id>

### ****Once you’ve worked with an image, how do you push it to docker hub?****

$ docker push <username/image name>

### ****How to build a Dockerfile?****

Once you’ve written a Dockerfile, you need to build it to create an image with those specifications. Use the following command to build a Dockerfile:

$ docker build <path to docker file>

### ****Can you remove a paused container from Docker?****

The answer is no. You cannot remove a paused container. The container has to be in the stopped state before it can be removed.

Docker compose

* 1. To define and run multi container docker application.
  2. Use yaml file to configure application services (docker-compose.yml).
  3. Can start all services with a single command :- docker compse up
  4. Can stop all service with single command :- docker compose down
  5. Can scale up selected services when required.

Commands ;\_

Docker-compse -v (To show docker compose version)

Docker-compose config (To validate the compose file )

Docker-compse up -d

Docker -compose up -d –scale database=4 (to scale database server upto 4 means 4 instance will of database will start.)

You can build docker file from docker compose file.

FROM is the first command in docker file.

## **FROM**

This is the first command in the Dockerfile. Without this, we can’t build an image. We can build the image just with this command. when we build just with **FROM,**we are actually taking the base image CMD whenever the image is instantiated.

## **CMD**

**CMD** command is used to give the default commands when the image is instantiated, it doesn’t execute while build stage. There should be only one CMD per Dockerfile, you can list multiple but the last one will be executed.

To build and Image

Docker build -t dokcerfile3 -f

## **ENTRYPOINT**

ENTRYPOINT is used as an executable for the container. Let’s look at the below example

we are using ENTRYPOINT for executable command and using CMD command to pass some default commands to the executable.

|  |
| --- |
| FROM node:8.11-slim |
|  |  |
|  | # command executable and version |
|  | CMD ["-v"] |
|  | ENTRYPOINT ["node"] |

* if we specify executable in ENTRYPOINT, we can use CMD to pass default parameters to it (look at ENTRYPOINT section)
* if not, we can specify executable and default params in the CMD.
* we can override the default command given in CMD while running the container

## **WORKDIR**

**WORKDIR** sets the working directory for all the consecutive commands. we can have multiple **WORKDIR** commands and will be appended with a relative path. Consider the following example where we have two **WORKDIR** commands leads to **/usr/node/app**

|  |
| --- |
| FROM node:8.11-slim |
|  |  |
|  | WORKDIR /usr/node |
|  | WORKDIR app |
|  |  |
|  | RUN pwd |
|  |  |
|  | # command executable and version |
|  | ENTRYPOINT ["node"] |

## **ENV**

ENV sets the environment variables for the subsequent instructions in the build stage. Consider the below example where we define the environment variable workdirectory and we used that later with $

|  |
| --- |
| FROM node:8.11-slim |
|  |  |
|  | ENV workdirectory /usr/node |
|  |  |
|  | WORKDIR $workdirectory |
|  | WORKDIR app |
|  |  |
|  | RUN pwd |
|  |  |
|  | # command executable and version |
|  | ENTRYPOINT ["node"] |

## **LABEL**

LABEL is used to add some metadata to the image. if we use the same label as the base image and the most recent label value is applied.

|  |
| --- |
| # from base image node |
|  | FROM node:8.11-slim |
|  |  |
|  | LABEL "about"="This file is just am example to demonstarte the LABEL" |
|  |  |
|  | ENV workdirectory /usr/node |
|  |  |
|  | WORKDIR $workdirectory |
|  | WORKDIR app |
|  |  |
|  | COPY package.json . |
|  |  |
|  | RUN ls -ll |
|  |  |
|  | # command executable and version |
|  | ENTRYPOINT ["node"] |

## **RUN**

RUN executes the instructions in a new layer on top of the existing image and commit those layers and the resulted layer will be used for the next instructions in the Dockerfile. consider this example we are doing npm install and ls -l with one RUN to avoid any additional layers.

|  |
| --- |
| FROM node:8.11-slim |
|  |  |
|  | LABEL "about"="This file is just am example to demonstarte the LABEL" |
|  |  |
|  | ENV workdirectory /usr/node |
|  |  |
|  | WORKDIR $workdirectory |
|  | WORKDIR app |
|  |  |
|  | COPY package.json . |
|  |  |
|  | RUN ls -ll &&\ |
|  | npm install |
|  |  |
|  | # command executable and version |
|  | ENTRYPOINT ["node"] |

## **ADD**

ADD is used to add files or directories and remote files from URL from source host filesystem to a destination in the container file system. Consider this example where we are adding index.js from our system to container file system. We can verify that while building with the RUN command ls -l

|  |
| --- |
| FROM node:8.11-slim |
|  |  |
|  | LABEL "about"="This file is just am example to demonstarte the LABEL" |
|  |  |
|  | ENV workdirectory /usr/node |
|  |  |
|  | WORKDIR $workdirectory |
|  | WORKDIR app |
|  |  |
|  | COPY package.json . |
|  |  |
|  | RUN ls -ll &&\ |
|  | npm install |
|  |  |
|  | ADD index.js . |
|  |  |
|  | RUN ls -l |
|  |  |
|  | # command executable and version |
|  | ENTRYPOINT ["node"] |

## **dockerignore**

Whenever we build the image at the root level, the entire context is sent to the Docker daemon. Sometimes we don’t need to send all the content to Docker daemon, those files or directories should be added to this .dockerignore file.

for example, we have node\_modules in the context and we have added that to .dockerignore file.

## **ARG**

ARG is used to pass some arguments to consecutive instructions and this is only command other than a comment can be used before FROM. We can see ARG usage in the below file and also we can pass that with the build command.

|  |
| --- |
| ARG NODE\_VERSION=8.11slim |
|  | FROM node:$NODE\_VERSION |
|  |  |
|  | LABEL "about"="This file is just am example to demonstarte the LABEL" |
|  |  |
|  | ENV workdirectory /usr/node |
|  |  |
|  | WORKDIR $workdirectory |
|  | WORKDIR app |
|  |  |
|  | COPY package.json . |

## **VOLUME**

VOLUME is used to create a mount point with the specified name. Following are the examples of Dockerfile and running instructions.

|  |
| --- |
| ARG NODE\_VERSION=8.11-slim |
|  | FROM node:$NODE\_VERSION |
|  |  |
|  | LABEL "about"="This file is just am example to demonstarte the LABEL" |
|  |  |
|  | ENV workdirectory /home/bhargav |
|  |  |
|  | RUN mkdir /dockerexample |
|  | VOLUME /dockerexample |
|  |  |
|  | COPY package.json . |
|  |  |
|  | RUN ls -ll &&\ |
|  | npm install |
|  |  |
|  | RUN useradd bhargav &&\ |
|  | mkdir -p $workdirectory &&\ |
|  | chown bhargav $workdirectory |
|  |  |
|  | USER bhargav |
|  | WORKDIR $workdirectory |
|  |  |
|  | ADD index.js . |
|  |  |
|  | RUN ls -l |
|  |  |
|  | EXPOSE 3070 |
|  |  |
|  | # command executable and version |
|  | ENTRYPOINT ["node"] |

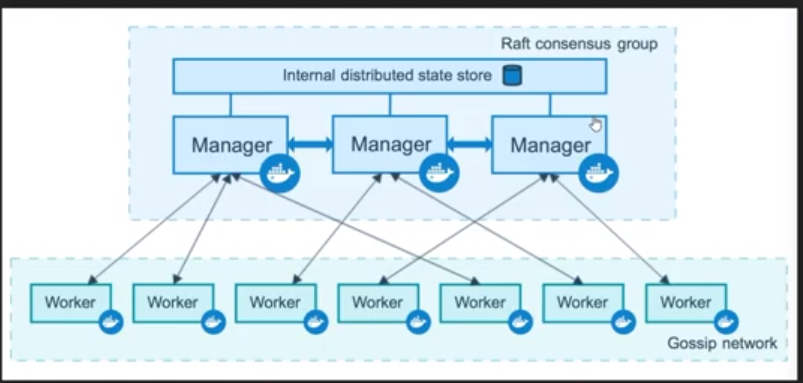
**Docker swarm is** a container orchestration tool, meaning that it allows the user to manage multiple containers deployed across multiple host machines. One of the key benefits associated with the operation of a **docker swarm is** the high level of availability offered for applications.

What is blue green deployment AWS?

A **blue**/**green deployment** is a **deployment** strategy in which you create two separate, but identical environments. One environment (**blue**) is running the current application version and one environment (**green**) is running the new application version.

gradually transfers user traffic from a previous version of an app or microservice to a nearly identical new release—both of which are running in production.

Docker swarm architecture:-



Docker swarm commands :-

Docker swarm init (to initialize docker swarm)

Docker nodes ls (it will give all details of nodes which are running on master node.)

Docker nodes ls command does not run on worker node ….command should be run on master nodes.

We can add woker to nodes to master nodes with the help of token.

Docker swarm join-token worker (It will give token of worker by using this token you can add another worker node).

Docker swarm join -token manager (it will give you token/command by using this token you can add master /manager ).

Docker swarm leave (it should run on worker node …once you run this command worker node will remove from master node)

Docker node rm <worker Node name>(to remover worker node from swarm cluster)

Docker node inspect <worker node name> | less

How to promote worker node to master node

Docker node promote <worker\_name>

Suppose you want run one command on multiple worker node …then with the help of service command you can do it. But you have to run all these command on Manager node.

Docker service create -d <image\_name> <> <command which you want to run>

Docker service ls

Docker service create -d –replicas 4 <image\_name> <> <command which you want to run>

You can increase the replicas with the help of below command

Docker service scale <container\_id>=<number which you want to scale.>

If you create the service then this service can be accessible from any worker nodes IP address.

Ex >- docker service crate -d -p 8090 :80 nginx

Then nginx can be accessible from any workernode.\

Docker swarm visualisaer (with the help of this funactnaloty ,,,you can run at least one container on each workernode)

\*\*\* docker service create mode=global alphine ping 8.8.8.8

=====================================================================;

By default, Docker will run a command in the foreground. To run in the background, the option -d needs to be specified.

By default, Docker will run the latest version available. If a particular version was required, it could be specified as a tag, for example, version 3.2 would be docker run -d redis:3.2.

docker inspect <friendly-name|container-id> provides more details about a running container, such as IP address.

docker logs <friendly-name|container-id> will display messages the container has written to standard error or standard out

Jane finds the best way to solve her problem of running Redis in the background, with a name of redisHostPort on port 6379 is using the following command.

docker run -d --name redisHostPort -p 6379:6379 redis:latest

The problem with running processes on a fixed port is that you can only run one instance. Jane would prefer to run multiple Redis instances and configure the application depending on which port Redis is running on.

#### Task

After experimenting, Jane discovers that just using the option -p 6379 enables her to expose Redis but on a randomly available port. She decides to test her theory using docker run -d --name redisDynamic -p 6379 redis:latest

While this works, she now doesn't know which port has been assigned. Thankfully, this is discovered via docker port redisDynamic 6379

Jane also finds that listing the containers displays the port mapping information, docker ps

Jane has been working with Redis as a background process. Jane wonders how containers work with foreground processes, such as ps or bash.

Previously, Jane used the -d to execute the container in a detached, background, state. Without specifying this, the container would run in the foreground. If Jane wanted to interact with the container (for example, to access a bash shell) she could include the options -it.

As well as defining whether the container runs in the background or foreground, certain images allow you to override the command used to launch the image. Being able to replace the default command makes it possible to have a single image that can be re-purposed in multiple ways. For example, the Ubuntu image can either run OS commands or run an interactive bash prompt using /bin/bash

#### Example

The command docker run ubuntu ps launches an Ubuntu container and executes the command ps to view all the processes running in a container.

Using docker run -it ubuntu bash allows Jane to get access to a bash shell inside of a container.

The Dockerfile is used by the Docker CLI build command. The build command executes each instruction within the Dockerfile. The result is a built Docker Image that can be launched and run your configured app.

The build command takes in some different parameters. The format is docker build -t <build-directory>. The -t parameter allows you to specify a friendly name for the image and a tag, commonly used as a version number. This allows you to track built images and be confident about which version is being started.

## Task

Build our static HTML image using the build command below.

docker build -t webserver-image:v1 .

You can view a list of all the images on the host using docker images.

The built image will have the name webserver-image with a tag of v1.

The built Image can be launched in a consistent way to other Docker Images. When a container launches, it's sandboxed from other processes and networks on the host. When starting a container you need to give it permission and access to what it requires.

For example, to open and bind to a network port on the host you need to provide the parameter -p <host-port>:<container-port>.

## Task

Launch our newly built image providing the friendly name and tag. As it's a web server, bind port 80 to our host using the -p parameter.

docker run -d -p 80:80 webserver-image:v1

Once started, you'll be able to access the results of port 80 via curl docker

#### Docker Images

Docker images are built based on a Dockerfile. A Dockerfile defines all the steps required to create a Docker image with your application configured and ready to be run as a container. The image itself contains everything, from operating system to dependencies and configuration required to run your application.

Having everything within the image allows you to migrate images between different environments and be confident that if it works in one environment, then it will work in another.

The Dockerfile allows for images to be composable, enabling users to extend existing images instead of building from scratch. By building on an existing image, you only need to define the steps to setup your application. The base images can be basic operating system installations or configured systems which simply need some additional customisations.

To help you complete the steps, an environment has been created with Docker configured. The editor allows you to write a Dockerfile which defines how to build the Docker image.

#### Base Images

All Docker images start from a base image. A base image is the same images from the Docker Registry which are used to start containers. Along with the image name, we can also include the image tag to indicate which particular version we want, by default, this is latest.

With the base image defined, we need to run various commands to configure our image. There are many commands to help with this, the main commands two are COPY and RUN.

RUN <command> allows you to execute any command as you would at a command prompt, for example installing different application packages or running a build command. The results of the RUN are persisted to the image so it's important not to leave any unnecessary or temporary files on the disk as these will be included in the image.

COPY <src> <dest> allows you to copy files from the directory containing the Dockerfile to the container's image. This is extremely useful for source code and assets that you want to be deployed inside your container.

#### Exposing Ports

With our files copied into our image and any dependencies downloaded, you need to define which port application needs to be accessible on.

Using the EXPOSE <port> command you tell Docker which ports should be open and can be bound to. You can define multiple ports on the single command, for example, EXPOSE 80 433 or EXPOSE 7000-8000

#### Default Commands

With the Docker image configured and having defined which ports we want accessible, we now need to define the command that launches the application.

The CMD line in a Dockerfile defines the default command to run when a container is launched. If the command requires arguments then it's recommended to use an array, for example ["cmd", "-a", "arga value", "-b", "argb-value"], which will be combined together and the command cmd -a "arga value" -b argb-value would be run.

Using the docker build command to build the image. You can give the image a friendly name by using the -t <name> option.

We can define a working directory using WORKDIR <directory> to ensure that all future commands are executed from the directory relative to our application.

#### OnBuild

While Dockerfile's are executed in order from top to bottom, you can trigger an instruction to be executed at a later time when the image is used as the base for another image.

The result is you can delay your execution to be dependent on the application which you're building, for example the application's package.json file.

Below is the [Node.js OnBuild Dockerfile](https://github.com/docker-library/node/blob/70741d88bf688389bfac7b147573f3b761f9ede9/0.10/onbuild/Dockerfile). Unlike in our previous scenario the application specify commands have been prefixed with ONBUILD.

#### Create Container

Data Containers are containers whose sole responsibility is to be a place to store/manage data.

Like other containers they are managed by the host system. However, they don't run when you perform a docker ps command.

To create a Data Container we first create a container with a well-known name for future reference. We use busybox as the base as it's small and lightweight in case we want to explore and move the container to another host.

When creating the container, we also provide a -v option to define where other containers will be reading/saving data.

#### Create Network

The first step is to create a network using the CLI. This network will allow us to attach multiple containers which will be able to discover each other.

In this example, we're going to start by creating a backend-network. All containers attached to our backend will be on this network.

Docker Volumes allow directories to be shared between containers and container versions.

Docker Volumes allows you to upgrade containers, restart machines and share data without data loss. This is essential when updating database or application versions.

#### Data Volumes

Docker Volumes are created and assigned when containers are started. Data Volumes allow you to map a host directory to a container for sharing data.

This mapping is bi-directional. It allows data stored on the host to be accessed from within the container. It also means data saved by the process inside the container is persisted on the host.

## Task

This example will use Redis as a way to persist data. Start a Redis container below, and create a data volume using the -v parameter. This specifies that any data saved inside the container to the /data directory should be persisted on the host in the directory /docker/redis-data.

docker run -v /docker/redis-data:/data \ --name r1 -d redis \ redis-server --appendonly yes

We can pipe data into the Redis instance using the following command.

cat data | docker exec -i r1 redis-cli --pipe

Redis will save this data to disk. On the host we can investigate the mapped direct which should contain the Redis data file.

ls /docker/redis-data

This same directory can be mounted to a second container. One usage is to have a Docker Container performing backup operations on your data.

docker run -v /docker/redis-data:/backup ubuntu ls /backup

When running containers in production, it can be useful to add additional metadata relating to the container to help their management. This metadata could be related to which version of the code is running, which applications or users own the container or define special criteria such as which servers they should run on.

This additional data is managed via Docker Labels. Labels allow you to define custom metadata about a container or image which can later be inspected or used as part of a filter.

Load balancer:-

we'll explore how you can use the NGINX web server to load balance requests between two containers running on the host.

With Docker, there are two main ways for containers to communicate with each other. The first is via links which configure the container with environment variables and host entry allowing them to communicate. The second is using the Service Discovery pattern where uses information provided by third parties, in this scenario, it will be Docker's API.

The Service Discovery pattern is where the application uses a third party system to identify the location of the target service. For example, if our application wanted to talk to a database, it would first ask an API what the IP address of the database is. This pattern allows you to quickly reconfigure and scale your architectures with improved fault tolerance than fixed locations.

## What is Swarm Mode

In 1.12, Docker introduced Swarm Mode. Swarm Mode enables the ability to deploy containers across multiple Docker hosts, using overlay networks for service discovery with a built-in load balancer for scaling the services.

Swarm Mode is managed as part of the Docker CLI, making it a seamless experience to the Docker ecosystem.

## Key Concepts

Docker Swarm Mode introduces three new concepts which we'll explore in this scenario.

* **Node**: A Node is an instance of the Docker Engine connected to the Swarm. Nodes are either managers or workers. Managers schedules which containers to run where. Workers execute the tasks. By default, Managers are also workers.
* **Services**: A service is a high-level concept relating to a collection of tasks to be executed by workers. An example of a service is an HTTP Server running as a Docker Container on three nodes.
* **Load Balancing**: Docker includes a load balancer to process requests across all containers in the service.

## Create Swarm Mode Cluster

Swarm Mode is built into the Docker CLI. You can find an overview the possibility commands via docker swarm --help

The most important one is how to initialise Swarm Mode. Initialisation is done via init.

docker swarm init

After running the command, the Docker Engine knows how to work with a cluster and becomes the manager. The results of an initialisation is a token used to add additional nodes in a secure fashion. Keep this token safe and secure for future use when scaling your cluster.

In the next step, we will add more nodes and deploy containers across these hosts.

#### Join Cluster

With Swarm Mode enabled, it is possible to add additional nodes and issues commands across all of them. If nodes happen to disappear, for example, because of a crash, the containers which were running on those hosts will be automatically rescheduled onto other available nodes. The rescheduling ensures you do not lose capacity and provides high-availability.

On each additional node, you wish to add to the cluster, use the Docker CLI to join the existing group. Joining is done by pointing the other host to a current manager of the cluster. In this case, the first host.

Docker now uses an additional port, 2377, for managing the Swarm. The port should be blocked from public access and only accessed by trusted users and nodes. We recommend using VPNs or private networks to secure access.

## Task

The first task is to obtain the token required to add a worker to the cluster. For demonstration purposes, we'll ask the manager what the token is via swarm join-token. In production, this token should be stored securely and only accessible by trusted individuals.

token=$(ssh -o StrictHostKeyChecking=no 172.17.0.8 "docker swarm join-token -q worker") && echo $token

On the second host, join the cluster by requesting access via the manager. The token is provided as an additional parameter.

docker swarm join 172.17.0.8:2377 --token $token

By default, the manager will automatically accept new nodes being added to the cluster. You can view all nodes in the cluster using docker node ls

## Task

The following command will create a new overlay network called skynet. All containers registered to this network can communicate with each other, regardless of which node they are deployed onto.

docker network create -d overlay skynet

A private Registry enables you to distribute Docker Images without being dependent on external providers or the public cloud. This allows you to increase security and confidence of your image sources and versioning.