**Docker**

[**https://www.interviewbit.com/docker-interview-questions/**](https://www.interviewbit.com/docker-interview-questions/)

1. What is a container?

* A container is a way to **package** applications with everything they need inside that package including the **dependencies** and all the necessary **configuration**.
* This container is **portable** just like any other artifacts, which can be easily moved and shared around between a development team and development and operation team.
* This portability of container and everything packaged in one isolated environment gives it an advantage to make development and deployment **more efficient.**
* **Technically**, a container is made up of images so we have layers of staked images on top of each other.
* And at the base of the most container you would have a **Linux based image** which is either alpine with specific version or it could any other Linux distribution and it is important for those base image to be small that’s why most of them are actually alpine because that will make sure that the containers stay small in size which is an advantage to use a container.
* So on the top of the base image you would have **application image.** For example postgres.

1. Where do containers live?

* Container live in container repository, it is a special type of storage for containers.
* Many companies have their own private repositories where they host or store all the containers.
* There is also a public repository for Docker containers where we can browse and find any application containers.

1. How did we develop application before containers?

* Usually when you have a team of developers working on some application you will have to install most of the services on your OS directly. For example, you are developing a JavaScript application and you need PostgreSQL and need redis for messaging. So then every developer in the team have to install the binaries of those services and configure and run them on their local development environment.
* Depending on which OS they are using the **installation** **process** will look actually different, also other thing installation services like this is that you have **multiple** **steps** for installation. So you have couple of commands to execute and the chances of something going wrong and error happening is pretty high because of the number of steps to require to install each service.
* This approach can be difficult depending on how complex your application is for example if you ten services that your application is using then you would have to do that ten times in each OS environment.

1. How did we develop application after containers?

* You actually do not have to install any of the services directly OS. Because the container is its own isolated operating system layer packaged with all needed configuration. So you have PostgreSQL with a specific version packaged with the configuration in the start script inside of one container. So developer need not to go and look for binaries to download in the machine rather go browse the specific container and download on your machine.
* Download step is only one Docker command fetches the container and starts it at the same time. And regardless which operating system you are on the Docker command for starting the container will be same.
* So if we have ten applications that your JavaScript application uses or depends on you would just run ten Docker commands for each container that’s it which makes setting up your local development environment much easier and efficient.
* Even you can have same application on your local environment with different versions there will be no conflict.

1. Deployment process before the containers?

* Development team will produce artifacts together with a set of instructions of how to actually install and configure those artifacts on the server.
* So you would have a jar file or something similar for an application and in addition you would have some kind of database service or the other service also with set of instruction of how to configure and set it up on the server.
* So development would give those artifacts over to the operation team and the operations team will handle setting up the environment to deploy those applications.
* Now the problem with these kind of process is that first of all you need to configure everything and install everything directly on the operating system which we saw on the previous example that could actually lead to conflict with dependency version and multiple services running on same host.
* Another problem that can be arise from this kind of process is when there is miss understanding between development and operation team because everything is in a textual guide as an instruction so there could be cases where developer can forget to mention some important points for configuration or maybe when operation team misinterprets some of those instructions. So when that fails the operational team has to go to the development team and ask for more details and this can lead of back and forth communication until the application is successfully deployed on the server.

1. Deployment process after the containers?

* With containers this process is actually simplified because now you have the developer and the operation working in one team to package whole configuration, dependencies inside the application.
* Since already it is encapsulated in one single environment and you need not to configure any of them directly on the server. So the only thing you have to do is run a Docker command that pulls the container image that you have stored somewhere in the repository and then run it.
* This is actually a simple process where no environmental configuration needed on the server. The only thing you need to do is install and set up Docker runtime on the server before you able to run the containers but that’s just one time effort.

**Docker File:** It is a text document that contains all the instructions that are required to create an image

**FORMAT OF DOCKER FILE:**

FROM : Docker file must start with a from instruction(from which base image u want to start creating ur image)

COPY : copies from the current machine to the path mentioned inside container

ADD : add also copies from one location to image but it support 2 sources -

* U can use a url instead of a local file
* Can extract a tar file into the destination

EXPOSE : it informs docker or the person who is running the image that a container listens at a specific port at runtime.

ENTRYPOINT : this is the main command but cannot be overridden during run time

CMD : this is the main command but can be overridden during run time

WORKDIR : It sets the working directory for any command that you execute. Any command after mentioning workdir will be executed in that folder.

ENV : environment variable

**Docker image**: Image is actual package, the application packages along with the configuration, dependencies this is an artifact which can be moved around. Docker image is which is not running.

**Docker container**: When an image is pulled to local machine and start it, the application inside actually starts, this creates a container environment. Docker container is when It is started or running. So container is a running environment for an image

**Commands:**

**Docker build -t dev:v1 . –** builds the image and gives name as dev and tag as v1

Docker tag imageid dev:v1 – tags an already build image

**Docker ps**: - to see running containers

**Docker images**: - to check all the existing images

**Docker pull imagename**: - version or Docker pull imagename: pulls the image from repo to local

**Docker run**: -start an image in a container or starts new container

**Docker run –d containername**: - to run container in detach mode

**Docker stop containerID**: - stops the container

**Docker start containerID**: - start the container

**Docker ps –a**: - shows all the containers which are running or stopped

**Docker run imagename: version or Docker run Imagename**: - pulls the image and runs it.

**Docker run –p host port: containerport imagename**: - binding the port of host to container.-p is for port binding.

**Docker logs containername or docker logs containerID**: - To check the logs, for example somethings happens and our application cannot connect to that container. To check what logs the container is producing.

**Docker run -d -p host port: containerport –name give any name imagename: version**: - to give a name to the container

**Docker exec –it containerID /bin/bash or Docker exec –it containername /bin/bash**: - To go inside a container. Where –it is interactive terminal to come out of container simply give exit.

**DOCKER VOLUMES**

Volumes are used to separate data from container and store it . This is to prevent data loss when container is deleted or to share data between containers.

Commands:

docker volume create myvolume - creates new volume

docker run --name jenkinsvolume1 -v myvolume:/var/jenkins\_home -p 8080:8080 jenkins/jenkins

this creates new Jenkins container and mounts the volume into our new docker volume(myvolume)

docker run --name jenkinsvolume2 -v myvolume:/var/jenkins\_home -p 8081:8080 jenkins/Jenkins

this creates 2nd Jenkins which also uses same volume. So it will use same password and will have same jobs in container as 1st Jenkins

docker run --name jenkinsvolume3 -v /Users/m1055938/Desktop/JENKINS\_HOME:/var/jenkins\_home -p 8082:8080 jenkins/Jenkins

This will create 3rd Jenkins instance but volume is mounted on our local path provided.

**DOCKER NETWORKING**

**Bridge**: uses software bridge which allows containers connected to the same bridge network to communicate, while providing isolation from containers which are not connected to that bridge network. (172.17.0.0/16)

Creation:

docker network create –driver bridge mybridge

even if we do not specify a driver it will be bridge by default

**Host**: In this the containers run directly on the host network. We don’t have to expose port with -p

**None**: if u want to completely disable the networking stack on a container you can use none network. This mode will not assign any ip to containers and doesn’t have access to external network or any other containers.

-p – publishes specific host port to the container port (eg : -p 8080:80– <https://ip:8080>)

-P – this assigns a random host port to the container port

Connecting the containers to network:

Add flag during new container creation:

--link container02:container (link is legacy way to connect the containers we have new option now for user defined networks)

docker network connect network\_name container\_name

DOCKER COMPOSE

docker run -- name nginx1 -p 8081:80 nginx – converting this to a docker file

docker-compose.yaml

version: '3'

services:

nginx\_service:

image: nginx

ports:

- 8081:80

nginx\_service2:

image: nginx

ports:

- 8082:80

version: '3'

services:

nginx\_service:

image: nginx

ports:

- 8081:80

networks:

- my-network

nginx\_service2:

image: nginx

ports:

- 8082:80

networks:

my-network:

driver: bridge

docker-compose file.yml up

prepare to write few docker files

prepare to write few docker compose files