**Docker Interview Questions:**

**1) What is Virtual Machine**

Virtualization involves creating multiple virtual machines (VMs) on a single physical server. Each VM emulates a complete computer with its own operating system and resources, such as CPU, memory, and storage.

**Example**: Imagine you have a powerful computer, and you want to run both Windows and Linux on it. With virtualization, you can create two virtual machines on that single physical computer. One VM runs Windows, and the other runs Linux. Each VM operates independently as if it were on its own physical computer.

**2) what is hypervisor and explain with and example**

A hypervisor, also known as a virtual machine monitor (VMM), is a software or hardware layer that creates and manages virtual machines (VMs). It allows multiple operating systems to run on a single physical machine concurrently, effectively enabling virtualization. The primary purpose of a hypervisor is to abstract and virtualize the underlying hardware, allowing multiple VMs to share the same physical resources while remaining isolated from one another.

There are two main types of hypervisors:

**Type 1 Hypervisor (Bare-Metal Hypervisor):** This type runs directly on the physical hardware without the need for a host operating system. Examples of Type 1 hypervisors include VMware vSphere/ESXi, Microsoft Hyper-V, and Xen.

**Type 2 Hypervisor (Hosted Hypervisor):** This type runs on top of a host operating system and relies on it for hardware access. Users install and run VMs as applications within the host OS. Examples of Type 2 hypervisors include Oracle VirtualBox and VMware Workstation.

**3)** **why we have to move towards containerization but if virtualization is working very well?**

**4) what is container**

**A)** virtual Machine have a full operating system so that means they have complete isolation Whereas with containers you will see that containers do not have a complete operating system and they do not run their full operating system

Containers are a lightweight form of virtualization technology that allows you to package and run applications and their dependencies in isolated environments. They provide a way to bundle everything needed for an application to run, including the code, runtime, libraries, and system tools, into a single package called a container

**5)what is Docker or what is container**

A) Docker is a open-source containerization platform. It enables developers to package applications into containers.

why you are using containerization Platfrom?

is to build containers or to manage life cycle of the container

**(OR)**

Docker is a platform for developing, shipping, and running applications in containers. Containers are lightweight, standalone, and executable packages that contain everything needed to run an application, including the code, runtime, libraries, and dependencies.

**6) How containers are different from virtual machines?**

A) containers are very lightweight in nature because they don't have complete operating system but they have very minimal system dependencies that are required to run your application

example: 'Im running a Java application so what I would need is I need my application I need some Java runtime dependency to run the Java application and I just need some system libraries that are very required or that are like mandatory to run your application so these are the only three things that you require

**7) what is Docker life cycle**

**8) what are the different components**

**Docker Engine**

The Docker Engine is the core of the Docker system. It is a daemon that runs in the background and manages Docker containers, images, and networks.

**Docker Images**

A Docker image is a read-only template that can be used to create Docker containers. An image is made up of a filesystem, runtime, and environment.

**Docker Containers**

A Docker container is a running, executable instance of a Docker image. A container is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries, and settings.

**Dockerfile**

A Dockerfile is a text file that is used to create a Docker image. A Dockerfile contains instructions for the Docker Engine to create an image.

**Docker Hub**

The Docker Hub is a registry of Docker images. The Docker Hub can be used to store, share, and search for Docker images.

**Docker Compose**

Docker Compose is a tool that can be used to create and manage multiple Docker containers.

**Docker Swarm**

Docker Swarm is a cluster management tool that can be used to manage a group of Docker hosts.

**9)** **what is docker file and docker compose file**

**A)** Dockerfile and a Docker Compose file are two essential components of Docker, a containerization platform that allows you to package and distribute applications and their dependencies in a consistent and isolated environment.

**Dockerfile:**

A Dockerfile is a text file that contains a set of instructions for building a Docker image.

Docker images are like templates that contain all the necessary code and dependencies for running an application.

The Dockerfile specifies the base image, sets up the environment, copies application code and files, and defines how the container should run.

Dockerfiles are used to create immutable images, ensuring consistency between development, testing, and production environments.

**Docker Compose file:**

A Docker Compose file, often named **docker-compose.yml**, is used to define and manage multi-container Docker applications.

It allows you to specify multiple services, their configuration, relationships, and how they should interact with each other.

Docker Compose simplifies the process of running complex applications with multiple containers by defining them all in one configuration file.

**10) Explain various layers in docker file**

In a Dockerfile, each instruction represents a layer in the Docker image. These layers are created incrementally as Docker builds the image, and they contribute to the overall size and functionality of the image. Understanding the layers in a Dockerfile is crucial for optimizing image size and build efficiency. Here are the common layers you'll encounter in a typical Dockerfile:

**Base Image Layer:**

Every Docker image starts with a base image, which serves as the foundation for your application.

This base image contains the minimal operating system and runtime environment required to run your application.

You specify the base image using the **FROM** instruction, like **FROM ubuntu:20.04** or **FROM python:3.8-slim**.

This layer is essential because it provides the starting point for all other layers in the image.

**Layer for Environment Setup:**

After the base image, you often configure the environment within the container. This includes setting environment variables, creating directories, and defining working directories.

Common instructions used in this layer include **ENV**, **RUN**, **WORKDIR**, and **COPY**.

**Layer for Application Code and Files:**

In this layer, you copy your application code and files into the image.

The **COPY** or **ADD** instruction is typically used for this purpose. For example, **COPY . /app** copies the contents of the current directory into a directory called **/app** within the image.

**Layer for Dependency Installation:**

If your application relies on external dependencies, like libraries, packages, or modules, you'll often install them in a separate layer.

The **RUN** instruction is used to execute commands for installing dependencies, such as **RUN apt-get update && apt-get install -y <package-name>** for package installations or **RUN pip install <package-name>** for Python packages.

**Layer for Configuration:**

Configuration files needed for your application can be added in a dedicated layer.

The **COPY** instruction is commonly used to add configuration files, like **COPY config.ini /app/config.ini**.

**Layer for Cleanup:**

To minimize the image size, it's a good practice to remove any temporary files or cleanup steps in a separate layer.

Use the **RUN** instruction to remove unnecessary files or perform cleanup actions. For example, **RUN apt-get clean** to remove cached package files.

**Layer for Application Execution Command:**

Finally, the **CMD** or **ENTRYPOINT** instruction defines the command that will be executed when a container is started from the image.

This layer determines what the container will do when it runs. For example, **CMD ["python", "app.py"]** specifies that the **app.py** script should be run when the container starts.

**11) what is difference between Docker copy and add?**

A) Docker ADD can copy the files from a URL unlike Docker COPY which can only copy files from host system into the container

**12) what is the difference between CMD and entry point**

A) CMD: let’s say you want to pass some arguments to the container and this arguments can be overwritten

Entry point: let’s say you want to pass some arguments to the container and this arguments cannot be overwritten

Example: lets us run a python related application it’s a calculator function

For this calculator function you want to read the arguments from the user to perform addition functionality and I want to get the output of 2 plus 3. So in such cases  what you can do is for the entry point you can provide the name of the function and because the other parameters are configurable like user can provide addition user can provide subtraction multiplication or division so in such cases how you do is the parameter that should not be overridden pass it as an entry point or the function or the you know executable which should not be overwritten should always be passed as an entry point and the parameters that can be overridden or you know any fields that can be configured that should be passed using the CMD

\*\*\*\* so there can be a question like okay should I mandatory use both command and entry point no you can either use one of them or you can use a combination of both of them

**Docker Networking**

**1) what is Docker networking**

Networking allows containers to communicate with each other and with the host system. Containers run isolated from the host system and need a way to communicate with each other and with the host system.

**2)How does Docker network work?**

Docker networking is based on the use of virtual networks that containers connect to. Each container can be connected to one or more networks, and Docker provides drivers for various types of networks, such as bridge, host, overlay, and more. Docker creates network bridges or overlays, allowing containers on the same network to communicate with each other using IP addresses.

**3) Why Docker Networking with Example?**

Docker networking is crucial because it enables communication between containers and with external services. For example, in a microservices architecture, different containers running various parts of an application need to talk to each other. Docker's networking capabilities facilitate this communication. Here's a basic example:

Suppose you have a web application where one container runs the front-end and another runs the back-end. Docker networking allows these containers to communicate, so the front-end can make API requests to the back-end for data.

**4) Comparing Docker Networking with VM networking**

In Docker, containers on the same network can communicate directly using private IP addresses. Docker uses a bridge network by default.

In VMs, communication between virtual machines typically involves more complex networking configurations. VMs may have their own IP addresses, and routing between VMs might require additional configuration.

**5) Different types of Networking in Docker**

Docker offers various network types, including:

Bridge Network: Default for standalone containers, isolates containers on the same host.

Host Network: Containers share the host's network stack.

Overlay Network: Used for Docker Swarm mode to enable multi-host communication.

Macvlan Network: Allows containers to have their own MAC and IP addresses.

Custom Bridge Networks: User-defined bridge networks for isolated container communication.

Host Networking in Docker

When you use host networking mode (--network host), the container shares the host's network stack, which means it can access resources on the host with the host's IP address. This mode offers better performance but sacrifices network isolation between the container and the host.

**6) Difference between this networks?**

A) **Bridge Network:**

Think of a bridge network as a virtual switch or a hub.

Containers connected to a bridge network are like computers connected to the same hub. They can talk to each other within the same network.

However, they are isolated from the host machine's network and other bridge networks, like different rooms in a house with separate phones.

**Host Network:**

Using the host network is like sharing your computer's network connection with a container.

Containers on the host network act as if they are directly connected to the same network as your computer. They share the same IP address and network stack.

It's like your computer and the container are in the same room, talking through the same telephone line.

**Overlay Network:**

Imagine you have multiple houses (computers) in different locations, and you want to connect them as if they are all in the same neighborhood.

Overlay networks connect containers across different physical machines (hosts) so they can communicate with each other as if they are on the same network, just like houses in the same neighborhood can talk to each other even if they're in different locations.

It's like having a secret tunnel connecting houses in different cities.

In summary:

Bridge Network: Isolated containers on a private network, like rooms with separate phones.

Host Network: Containers share the host's network, like being in the same room with the same phone.

Overlay Network: Connects containers across different machines, like houses in different locations in the same neighborhood.

**7) Can you explain how to isolate networks between containers?**

Custom Bridge Networks:

Docker allows you to create custom bridge networks to isolate containers. Containers attached to the same custom bridge network can communicate with each other, but they are isolated from containers on other networks.

Create a custom bridge network:

docker network create mynetwork

Attach containers to the custom network when starting them:

docker run -d –network=mynetwork --name container1 my\_image1

**9) How containers talk to Host in terms of networking?**

Containers communicate with the host system through a bridge network. They use the host's IP address to connect to the outside world. When a container needs to access external resources, it routes traffic through the host's network stack.

**10) Default Networking in Docker and What happens without that?**

The default networking mode in Docker is the "bridge" network. Containers attached to this network can communicate with each other using private IP addresses. If you create a container without specifying a network, it will be attached to the default bridge network. Without networking, containers would be isolated and unable to communicate with each other or the host.

**11)** **what is Multi Stage Build in Docker**

Multi-stage builds in Docker are a feature that allows you to create more efficient and smaller Docker images. This feature is particularly useful when you need to build and package an application inside a Docker image, but you want the final image to contain only the necessary runtime artifacts, minimizing the image size. Multi-stage builds involve using multiple **FROM** statements in a single Dockerfile to define multiple build stages, each with its own set of instructions.

Multi stage build allows you to build your docker container in multiple stages allowing you to copy artifacts from one stage to other. The major advantage of this is to build light weight containers

**Example:**

Let’s say I have multi-tire application where you have front-end back-end and database and final image will be just jar Or war file .

let's assume so for front end you will have dependencies with react or node or angular or something so you will copy all the dependencies you will know basically using pip command or basically using the APT command you will install all of those things then when you are trying to build things related to your back end let's say it's a Java application again you will install all the dependencies like you will install jdk you will install JRE all of those things right and finally let's say it also has some information like connectors with respect to the DB so you'll also install those connectors and when you try to build this image and see the image size will go over 1GB because you have lot of dependencies that you have installed on this image but the final executable just requires JRE or react runtime. let's assume that it requires two different runtimes but all the dependencies that you have installed was only request using the required using the build phase once the application is built all these dependencies were not required so what you can simply do is you can use multi-stage build and in the last stage you can simply copy the binaries or the executables that you have created and just take them into the final stage and in the final stage just install Java runtime not all the packages so you will see that the image size from 1GB will come down to 200 MB or 300 MB so this is the concept and advantage of multi-stage

**12) What are Distro less images in Docker?**

**A)** Distroless images contains only your application and its runtime dependencies with a very minimum operating system libraries. They do not contain package managers, shells or any other programs you would expect to find in a standard linux distribution. They are very small and lightweight images.

**13) what are docker volumes**

Docker volumes are a way to manage and persist data in Docker containers. They provide a method for sharing data between containers, allowing you to separate data storage from the container itself. Docker volumes are especially useful for preserving data and ensuring data persistence, even if the container is stopped or removed.

**14) How One Container Talks with Another Container:**

Containers can communicate with each other using various methods:

Docker Networking: Containers on the same network can communicate using container names or IP addresses.

Ports: You can expose ports in one container and connect to them from another using the host's IP address and the exposed port.

Docker Compose: Use Docker Compose to define and manage multi-container applications, specifying how containers should interact.

**15) Saving a Container as an Image and ZIP File:**

To save a container as an image and then as a ZIP file:

Save the container as an image using docker commit.

Export the image to a tarball using docker save.

Compress the tarball using a tool like tar or zip.

**Real Time Challenges With Docker?**

**1)** Docker Is a single daemon process. Which can cause a single point of failure, If the Docker Daemon goes down for some reason all the applications are Down

**Explanation:**

do you know any modern day solutions that address this problem you can say that yeah Buildah and podman is one such thing that does not have a demon that does not have a single point of failure

podman is a tool which address the docker problem

**2)** Docker Daemon runs as a root user. Which is a security threat. Any process running as a root can have adverse effects. When it is comprised for security reasons, it can impact other applications or containers on the host.

**Explanation**

**3) Resouce constraints:** if you are running too many containers on as single host you may experiences issues with resource constraints. This can result in slow performance or crashes

**Explanation**

if you're running 10 different containers on a single host if one container is leaking a lot of memory or it is using a lot of memory then the other containers which actually require the resources will not get the resources because one container is using 10 GB one container you know for some problem with this container it is it keeps on using more and more resources it will have impact on the other containers so you have to configure the resources very well for your containers if not the other containers will be impacted

**What steps Would You take to Secure Containers?**

Some of the steps,

1. Use Distroless or images with not too many packages as your final image in multi stage build. So that there is less chance of CVE or security isssues.

2. Ensure that are networking is configured properly. This is one of the most common reasons for security issues. If required configure custom bridge networks and assign them to isolate conatiners.

3. use utilities like sync to scan your containers images.

**Troubleshooting Containers**

Docker log <Container name>

Docker exec –it <container name> <command>

Docker exec –it <container name> /bin/bash or /sh

**2) way Restart Policies**

1) No: The default behavior is to not start containers automatically

2) Always: Always restart a stopped container unless the container was stopped explicitly

3)Unless-stopped: Restart the container unless the container was in stopped state before the docker daemon was stopped

4)On-failure: Restart the container if it exited with a non-zero exit code or if the docker daemon restarts