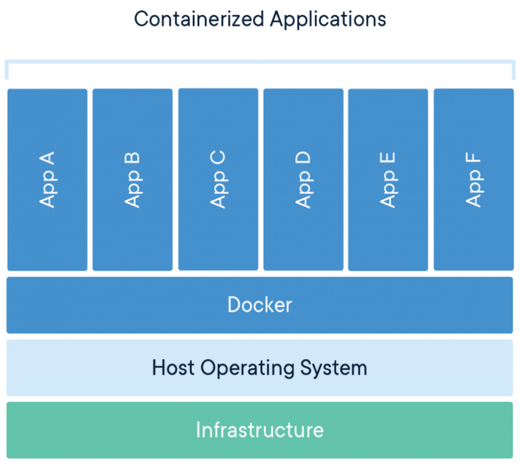
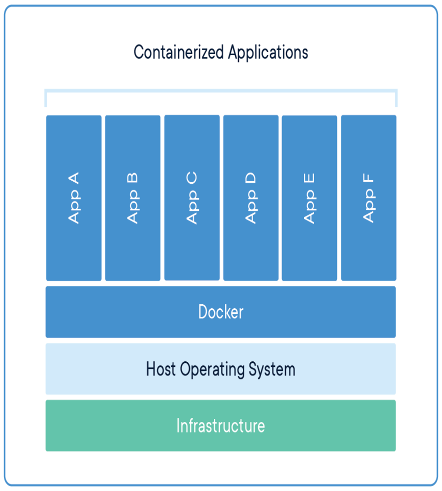
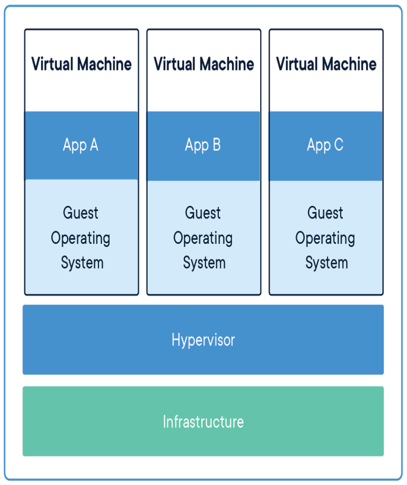
Dockers and Containers Question Bank

	UNIT – V	Marks
1.	**Discuss the problems solved by using the dockers and containers** Now if you have a little development experience, you already know that these technologies each have a version the application depends on. Also the application isn't an isolated thing that just floats around. It needs to run in an environment, which could be a development, test or a production environment. Since the environments can differ in OS, version, hardware etc, it's obvious that the application and its technologies with their respective versions should work the same on different environments. Without docker, this means that each environment that the application runs on (local dev environment, a test or production server) needs to be configured with the correct versions of these services so that application can run properly. **So the following PROBLEMs arise** ● Compatibility of each service with the underlying OS ● Compatibility of each service with the libraries and dependencies of OS (One service requires versionX of OS library. Another service - versionY of same library) ● Every time version of any service updates, you might need to recheck compatibilities with underlying OS infrastructure ● "Matrix from Hell" ● For a new developer to setup the environment with right OS and Service versions **Docker SOLUTION** ● Each service has and can manage its required OS dependencies for itself, bundled and isolated in its own container ● Change the components without affecting the other services ● Change underlying OS without affecting any of the services As a result, docker should avoid the typical "works on my machine" cases. In the development process, for example, developers and testers will have the identical environments where the application runs, since this environment is packaged in docker containers, which just like a file, can be transferred around as an artifact.	6
2.	**Discuss how containers differ from Hypervisor based virtualization.**	6

	Hypervisor-based virtualization allows to fully emulate another computer, therefore it is possible to emulate other types of devices (for example a smartphone), other CPU architectures or other operating systems. This is useful for example when developing applications for mobile platforms — the developer can test his application on his development system without the need of physically having access to a target device. Another common use case is to have virtual machines with other guest operating systems than the host. Some users need special software that does not run on their preferred operating system, virtualization allows to run nearly every required environment and software in this environment independently from the host system. **Container-based virtualization** utilizes kernel features to create an isolated environment for processes. In contrast to hypervisor-based virtualization, containers do not get their own virtualized hardware but use the hardware of the host system. Therefore, software running in containers does directly communicate with the host kernel and has to be able to run on the operating system (see figure 2) and CPU architecture the host is running on. Not having to emulate hardware and boot a complete operating system enables containers to start in a few milliseconds and be more efficient than classical virtual machines.	
3.	**Describe what difference does Docker bring to Containers.**  Containerized Applications App A App B App C App D App E App F Docker Host Operating System Infrastructure A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone,	6

	executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. Container images become containers at runtime and in the case of Docker containers - images become containers when they run on Docker Engine. Available for both Linux and Windows-based applications, containerized software will always run the same, regardless of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging. Docker containers that run on Docker Engine: ● Standard: Docker created the industry standard for containers, so they could be portable anywhere ● Lightweight: Containers share the machine’s OS system kernel and therefore do not require an OS per application, driving higher server efficiencies and reducing server and licensing costs ● Secure: Applications are safer in containers and Docker provides the strongest default isolation capabilities in the industry	
4.	**Illustrate with a diagram differences between container and virtual machine**	6





CONTAINERS

Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), can handle more applications and require fewer VMs and Operating systems.

VIRTUAL MACHINES

Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, the application, necessary binaries and libraries - taking up tens of GBs. VMs can also be slow to boot.

5. **Differentiate between process, virtual machine and containers.** 6

Container

- Isolate address space
- isolate files and networks
- Lightweight

Process

- Isolate address space
- No isolation for files or networks
- Lightweight

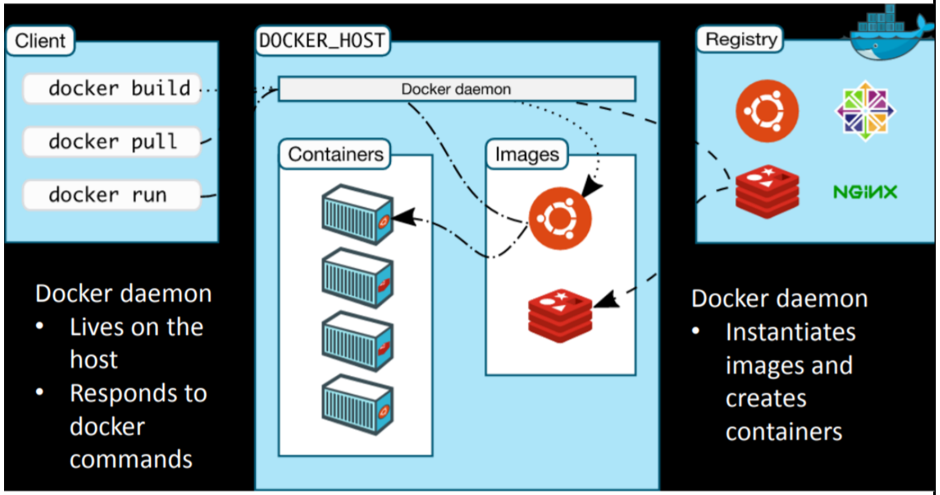
Virtual Machine

- Isolate address space
- isolate files and networks
- Heavyweight

6. **With a neat diagram explain the architecture of dockers.** 8

The Docker architecture uses a client-server model and comprises the Docker Client, Docker Host, Network and Storage components, and the Docker Registry / Hub. Let's look at each of these in some detail.

Docker Architecture



Docker Client

The Docker client enables users to interact with Docker. The Docker client can reside on the same host as the daemon or connect to a daemon on a remote host. A docker client can communicate with more than one daemon. The Docker client provides a command line interface (CLI) that allows you to issue build, run, and stop application commands to a Docker daemon.

The main purpose of the Docker Client is to provide a means to direct the pull of images from a registry and to have it run on a Docker host. Common commands issued by a client are:

docker build

docker pull

docker run

DockerHost

The Docker host provides a complete environment to execute and run applications. It comprises the Docker daemon, Images, Containers, Networks, and Storage. As previously mentioned, the daemon is responsible for all container-related actions and receives commands via the CLI or the REST API. It can also communicate with other daemons to manage its services. The Docker daemon pulls and builds container images as requested by the client. Once it pulls a requested image, it builds a working model for the container by utilizing a set of instructions known as a build file. The build file can also include instructions for the daemon to pre-load other components prior to running the container, or instructions to be sent to the local command line once the container is built.

Docker Objects

Various objects are used in the assembling of your application. The main requisite Docker objects are:

Images

Images are a read-only binary template used to build containers. Images also contain metadata that describe the container's capabilities and needs. Images are used to store and ship applications. An image can be used on its own to build a container or customized to add additional elements to extend the current configuration. Container images can be shared across teams within an enterprise using a private container registry, or shared with the world using a public registry like Docker Hub. Images are a core part of the Docker experience as they enable collaboration between developers in a way that was not possible before.

Containers

Containers are encapsulated environments in which you run applications. The container is defined by the image and any additional configuration options provided on starting the container, including and not limited to the network connections and storage options. Containers only have access to resources that are defined in the image, unless additional access is defined when building the image into a container. You can also create a new image based on the current state of a container. Since containers are much smaller than VMs, they can be spun up in a matter of seconds, and result in much better server density.

Networking

Docker implements networking in an application-driven manner and provides various options while maintaining enough abstraction for application developers. There are basically two types of networks available – the default Docker network and user-defined networks. By default, you get three different networks on the installation of Docker – none, bridge, and host. The none and host networks are part of the network stack in Docker. The bridge network automatically creates a gateway and IP subnet and all containers that belong to this network can talk to each other via IP addressing. This network is not commonly used as it does not scale well and has constraints in terms of network usability and service discovery.

The other type of networks is user-defined networks. Administrators can configure multiple user-defined networks. There are three types:

- **Bridge network**: Similar to the default bridge network, a user-defined Bridge network differs in that there is no need for port forwarding for containers within the network to communicate with each other. The other difference is that it has full support for automatic network discovery.
- **Overlay network**: An Overlay network is used when you need containers on separate hosts to be able to communicate with each other, as in the case of a distributed network. However, a caveat is that swarm mode must be enabled for a cluster of Docker engines, known as a swarm, to be able to join the same group.
- **Macvlan network**: When using Bridge and Overlay networks a bridge resides between the container and the host. A Macvlan network removes this bridge, providing the benefit of exposing container resources to external networks without dealing with port forwarding. This is realized by using MAC addresses instead of IP addresses.

Storage

You can store data within the writable layer of a container but it requires a storage driver. Being non-persistent, it perishes whenever the container is not running. Moreover, it is not easy to transfer this data. In terms of persistent storage, Docker offers four options:

- **Data Volumes**: Data Volumes provide the ability to create persistent storage, with the ability to rename volumes, list volumes, and also list the container that is associated with the volume. Data Volumes sit on the host file system, outside the container's copy on write mechanism and are fairly efficient.
- **Data Volume Container**: A Data Volume Container is an alternative approach wherein a dedicated container hosts a volume and to mount that volume to other containers. In this case, the volume container is independent of the application container and therefore can be shared across more than one container.

- **Directory Mounts**: Another option is to mount a host's local directory into a container. In the previously mentioned cases, the volumes would have to be within the Docker volumes folder, whereas when it comes to Directory Mounts any directory on the Host machine can be used as a source for the volume.
- **Storage Plugins**: Storage Plugins provide the ability to connect to external storage platforms. These plugins map storage from the host to an external source like a storage array or an appliance. A list of storage plugins can be found on Docker's Plugin page.

Storage Plugins

There are storage plugins from various companies to automate the storage provisioning process. For example,

- HPE 3PAR
- EMC (ScaleIO, XtremIO, VMAX, Isilon)
- NetApp

There are also plugins that support public cloud providers like:

- Azure File Storage
- Google Compute Platform.

Docker Registries

Docker registries are services that provide locations from where you can store and download images. In other words, a Docker registry contains Docker repositories that host one or more Docker Images. Public Registries include Docker Hub and Docker Cloud and private Registries can also be used. Common commands when working with registries include:

docker push

docker pull

docker run

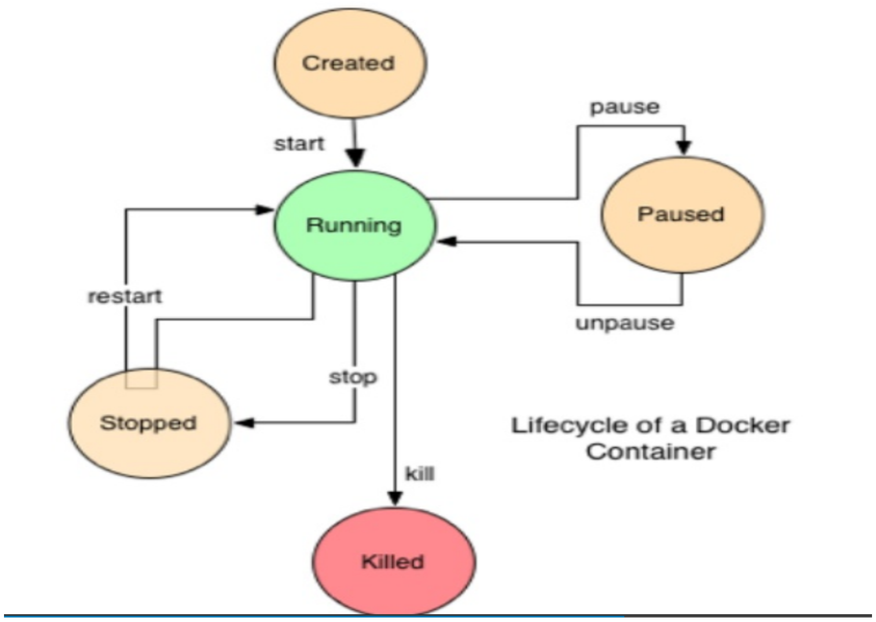
Service Discovery

Service Discovery allows containers to find out about the environment they are in and find other services offered by other containers.

It is an important factor when trying to build scalable and flexible applications.

7.	**Define the following terms:** i) Image ii) Container iii) Dockerfile iv)Docker Client v) Docker Daemon/Engine Answer: i) An image is a read-only template with instructions for creating a Docker container. Often, an image is based *on* another image, with some additional customization. For example, you may build an image which is based on the Ubuntu image, but install the Apache web server and your application, as well as the configuration details needed to make your application run. ii) A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state. By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine. iii) A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Using docker build users can create an automated build that executes several command-line instructions in succession. iv) The Docker client enables users to interact with Docker. The Docker client can reside on the same host as the daemon or connect to a daemon on a remote host. A docker client can communicate with more than one daemon. The Docker client provides a command line interface (CLI) that allows you to issue build, run, and stop application commands to a Docker daemon. The main purpose of the Docker Client is to provide a means to direct the pull of images from a registry and to have it run on a Docker host. Common commands issued by a client are: docker build docker pull docker run v) The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.	5
8.	**List out the similarities and differences of docker containers between Linux and Windows operating systems**	8

	Similarities • Similarities • Both are application containers, run natively, do not depend on hypervisors or virtual machines. • Both administered through Docker CLI/APIs • Only support containers with same platform architecture as the host OS. Docker for Windows can only host Windows applications inside Docker containers, and Docker on Linux supports only Linux apps. They provide the same portability and modularity features on both operating systems. **Differences** • Differences • Docker supports only Windows Server 2016 and Windows 10 now. • But Docker can run on any type of modern Linux-based operating system. • Most container orchestration systems used for Docker on Linux are not supported on Windows. • Only Docker Swarm is supported. Windows support for orchestrators such as Kubernetes and Apache Mesos is under development.	
9.	**With a neat diagram, describe Docker Container Life Cycle.**	7



- Created: A container that has been created but not started
- Running: A container running with all its processes
- Paused: A container whose processes have been paused
- Stopped: A container whose processes have been stopped
- Deleted: A container in a dead state

Create container:Create a container to run it later on with required image

```
docker create --name <container-name> <image-name>
```

Run docker container: Run the docker container with the required image and specified command / process. '-d' flag is used for running the container in background.

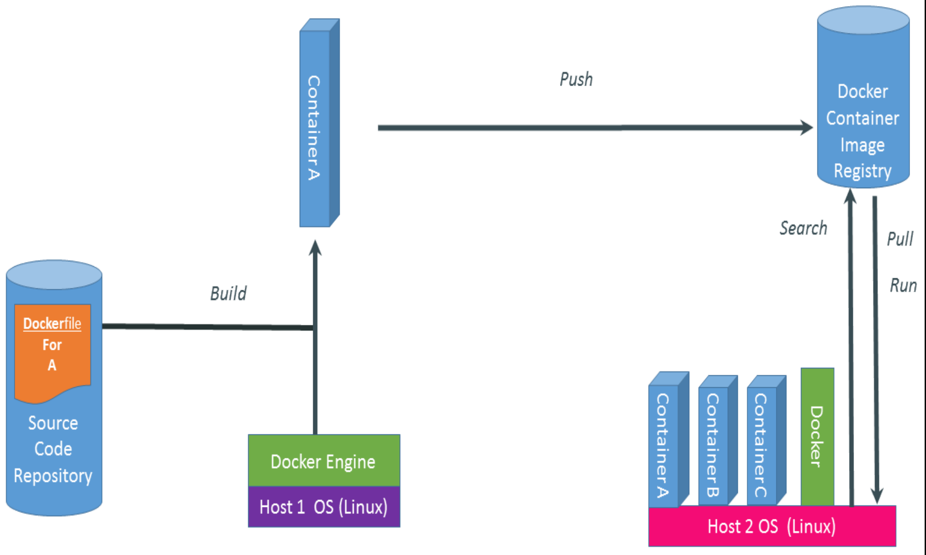
```
docker run -it -d --name <container-name> <image-name> bash
```

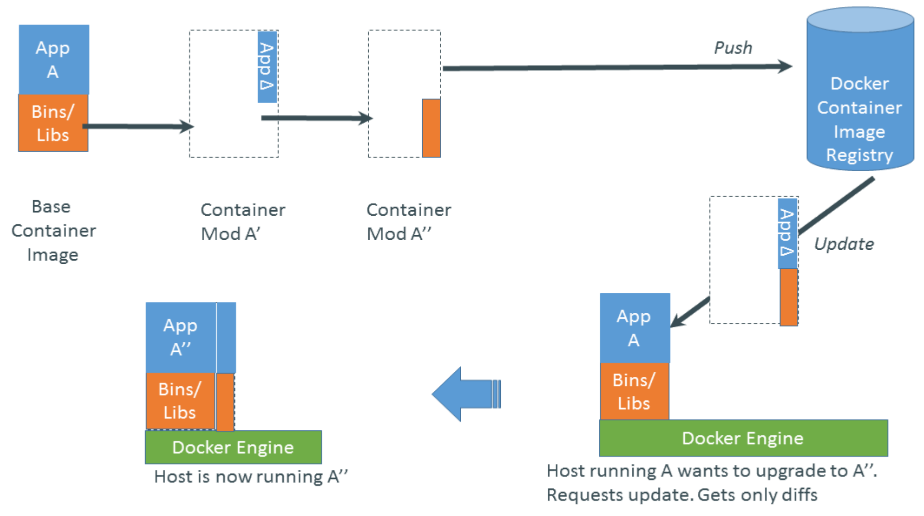
Pause container: Used to pause the processes running inside the container.

```
docker pause <container-id/name>
```

	Unpause container: Used to unpause the processes inside the container.
	
	`docker unpause <container-id/name>`
	
	Start container: Start the container, if present in stopped state.
	
	`docker start <container-id/name>`
	
	Stop container: To stop the container and processes running inside the container:`docker stop <container-id/name>`
	
	Restart container:It is used to restart the container as well as processes running inside the container.
	
	`docker restart <container-id/name>`
	
	Kill container: We can kill the running container.
	
	`docker kill <container-id/name>`
	
	Destroy container: Its preferred to destroy container, only if present in stopped state instead of forcefully destroying the running container.
	
	`docker rm <container-id/name>`
	
	To remove all the stopped docker containers	

10.	**Illustrate basic workflow of the docker with a suitable diagram**	6

	 Container A Push Docker Container Image Registry Search Pull Run Build Dockerfile For A Source Code Repository Docker Engine Host 1 OS (Linux) Container A Container B Container C Docker Host 2 OS (Linux) 1. A developer creates the dockerfile with all dependencies and how to create an image. 2. This dockerfile is then used by the docker engine to create the docker image of the application along with the necessary dependencies. 3. The created image can either be used by the local host or can be sent into the docker hub for it to be used by other users. It will be made publicly available on the docker hub. 4. Docker hub is a hub of images to which images can be pushed to or pulled from. 5. Users who are interested will pull these images and run it on their machine which has docker installed, thereby creating containers. 6. Containers are running instances of images. Running container means running an application flawlessly with all the dependencies.	
11.	**Illustrate workflow of the docker with App Updates / Changes with a suitable diagram**	8



12 **Describe different types of networks available for docker and containers**. 8

Docker comes with network drivers geared towards different use cases. The most common network types being: **bridge**, **overlay**, and **macvlan**.

1.None

•Used when your container doesn’t provide a service over the network

•The container isn’t provided with an n/w interface and an IP address

docker run --network none --name testApp training/webapp python app.py

2.Host

•Used when you would like to use the Docker host’s network stack

•Services running on any port within your container will be directly accessible through the host’s IP

•Means you will not be able to run multiple web containers on the same host

docker run -d --network host --name testApp training/webapp python app.py

3. Bridge Networks

Bridge networking is the most common network type. It is limited to containers within a single host running the Docker engine. Bridge networks are easy to create, manage and troubleshoot.

For the containers on bridge network to communicate or be reachable from the outside world, port mapping needs to be configured. As an example, consider you can have a Docker container running a web service on port **80** . Because this container is attached to the bridge network on a private subnet, a port on the host system like **8000** needs to be mapped to port 80 on the container for outside traffic to reach the web service.

To create a bridge network named my-bridge-net , pass the argument bridge to the -d (driver) parameter as shown below:

```
$ docker network create -d bridge my-bridge-net
```

- Bridge - default

  Used when your container doesn't provide a service on the network

  Needs "linking" for container DNS resolution

  ```
  docker run -d --network bridge --name testApp training/webapp python app.py
  ```

  ```
  docker run -d --network bridge -p 5001:5000 --name testApp training/webapp python app.py
  ```

- Bridge – user defined

  Used when you have a requirement for multiple networks on your Docker host, possibly because you'd like to isolate different deployments of containers on that host

  ```
  docker network create --driver bridge isolated_nw01
  ```

  ```
  docker run -d --network isolated_nw01 -p 5001:5000 --name testApp training/webapp python app.py
  ```

5.Overlay Networks

An overlay network uses software virtualization to create additional layers of network abstraction running on top of a physical network. In Docker, an overlay network driver is used for multi-host network communication. This driver utilizes Virtual Extensible LAN (VXLAN) technology which provides portability between cloud, on-premise and virtual environments. VXLAN solves common portability limitations by extending layer 2 subnets across layer 3 network boundaries, hence containers can run on foreign IP subnets.

To create an overlay network named my-overlay-net, you'll also need the --subnet parameter to specify the network block that Docker will use to assign IP addresses to the containers:

```
$ docker network create -d overlay --subnet=192.168.10.0/24 my-overlay-net
```

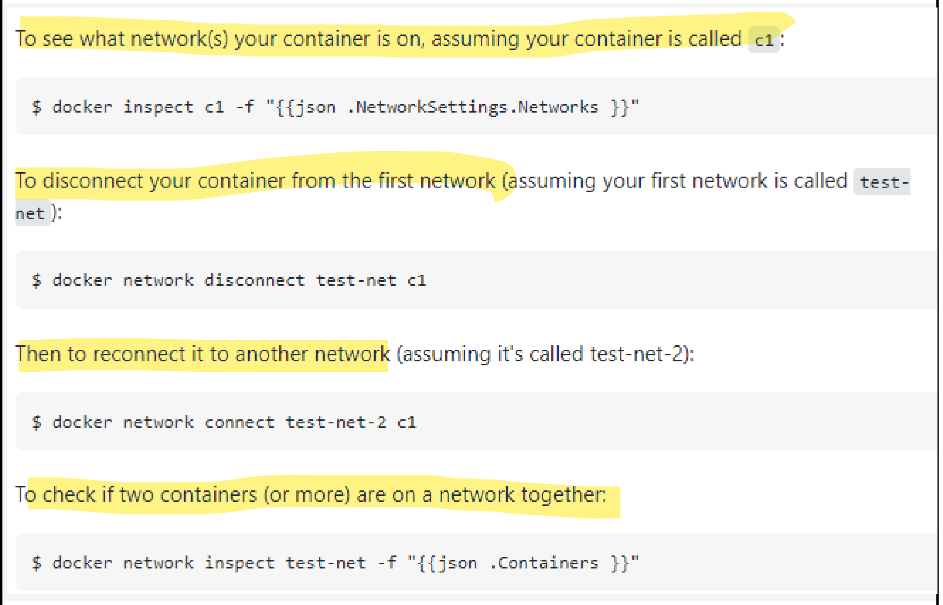
See Docker Documentation: **An overlay network without swarm mode ›**

6. Macvlan Networks

The macvlan driver is used to connect Docker containers directly to the host network interfaces through layer 2 segmentation. No use of port mapping or network address

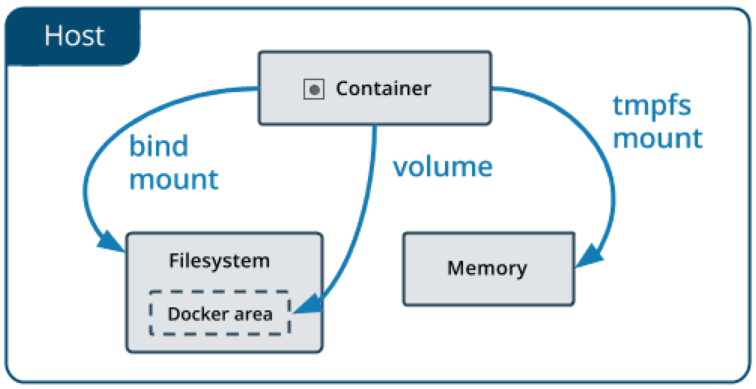
	translation (NAT) is needed and containers can be assigned a public IP address which is accessible from the outside world. Latency in macvlan networks is low since packets are routed directly from Docker host network interface controller (NIC) to the containers.
	
	Note that macvlan has to be configured per host, and has support for physical NIC, sub-interface, network bonded interfaces and even teamed interfaces. Traffic is explicitly filtered by the host kernel modules for isolation and security. To create a macvlan network named macvlan-net, you'll need to provide a --gateway parameter to specify the IP address of the gateway for the subnet, and a -o parameter to set driver specific options. In this example, the parent interface is set to eth0 interface on the host:
	
	`$ docker network create -d macvlan \`
	`  --subnet=192.168.40.0/24 \`
	`  --gateway=192.168.40.1 \`
	`  -o parent=eth0 my-macvlan-net`
	
	7.Custom network plugin
	•In case you find the current networking options don't fit your need, you could write your own network plugin using the Docker plugin API	

13.	**Explain the various ways in which a user can configure containers to be accessible** 1.--network •Ask Docker to connect your container to a specific network stack 2."-p" •Ask Docker to "publish" specific ports on your container to the host's ports 3."-P" •Ask Docker to publish "All" the container's active ports to the host 4.--link •Old way of establishing DNS registration for created containers in new containers. Still relevant for the default bridge network.	6

14.	**Show how a user can identify which network mode is being used by a container.**
	
	To see what network(s) your container is on, assuming your container is called `c1`:
	`$ docker inspect c1 -f "{{json .NetworkSettings.Networks }}"`
	To disconnect your container from the first network (assuming your first network is called `test-net`):
	`$ docker network disconnect test-net c1`
	Then to reconnect it to another network (assuming it's called test-net-2):
	`$ docker network connect test-net-2 c1`
	To check if two containers (or more) are on a network together:
	`$ docker network inspect test-net -f "{{json .Containers }}"`
		6

15.	**Explain why does a docker need a Union File System.** It is used to: ● avoid duplicating a complete set of files each time you run an image as a new container ● isolate changes to a container filesystem in its own layer, allowing for that same container to be restarted from a known content (since the layer with the changes will have been dismissed when the container is removed) That UnionFS:	6

	implements a union mount for other file systems. It allows files and directories of separate file systems, known as branches, to be transparently overlaid, forming a single coherent file system. Contents of directories which have the same path within the merged branches will be seen together in a single merged directory, within the new, virtual filesystem. This allows a file system to appear as writable, but without actually allowing writers to change the file system, also known as **copy-on-write.**	
16.	**Discuss the types of docker mount with a neat diagram.** Basically, there are 3 types of mounts which you can use in your Docker container viz. **Volumes, Bind mount and tmpfs mounts**. **Volumes** These are the most common mount options available for Docker containers and are fully managed by Docker engine. You can create a volume through Docker commands and can share it within the Docker containers. When you create a volume, it is stored within a directory on the Docker host (**/var/lib/docker/volumes/** on Linux) and is completely isolated from Docker host. Multiple Docker containers can use the same volumes and can read-write simultaneously. When the container is not running, data still persists in volumes. **Bind Mounts** These are the host machine file systems which are mounted on a Docker container and this Docker doesn't have control over it and the host machine only manages it. When you use a bind mount, a file or directory on the host machine is mounted into a Docker container. Their performance is good, but containers have to rely on the host machine's filesystem having a specific directory structure available. **tmpfs Mounts** As the name suggests, these mounts are temporary and once the Docker container is stopped the data present on these mounts is also lost. tmpfs mounts are stored in the host system's memory only and are never written to the host system's filesystem and thus does not hold data permanently.	6

	Below image from Docker website shows all these 3 mounts and depicts where the data lives on the Docker host.  Host Container bind mount volume tmpfs mount Filesystem Docker area Memory	
17	**Describe the characteristics and use cases of volumes.** Characteristics: ● Volumes are easier to back up or migrate than bind mounts. ● You can manage volumes using Docker CLI commands or the Docker API. ● Volumes work on both Linux and Windows containers. ● Volumes can be more safely shared among multiple containers. ● Volume drivers let you store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality. ● New volumes can have their content pre-populated by a container. ● Volumes on Docker Desktop have much higher performance than bind mounts from Mac and Windows hosts. ● Volumes are often a better choice than persisting data in a container's writable layer, because a volume does not increase the size of the containers using it, and the volume's contents exist outside the lifecycle of a given container. Use cases: ● Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. ● While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker.	8

	● docker run -d \ -it \ --name devtest \ --mount type=myvol2,source="$(pwd)"/target,target=/app \ nginx:latest	
18.	**Describe the characteristics and use cases of bind mounts.** Bind mounts have been around since the early days of Docker. Bind mounts have limited functionality compared to volumes. When you use a bind mount:(characteristics?) ● a file or directory on the *host machine* is mounted into a container. ● The file or directory is referenced by its absolute path on the host machine. ● By contrast, when you use a volume, a new directory is created within Docker's storage directory on the host machine, and Docker manages that directory's contents. ● The file or directory does not need to exist on the Docker host already. It is created on demand if it does not yet exist. ● Bind mounts are very performant, but they rely on the host machine's filesystem having a specific directory structure available. ● You can't use Docker CLI commands to directly manage bind mounts. Use cases: ● With **bind mounts**, we control the exact mount point on the host. We can use this to persist data, but it's often used to provide additional data into containers.	8

	● When working on an application, we can use a bind mount to mount our source code into the container to let it see code changes, respond, and let us see the changes right away. ● $ docker run -d \ -it \ --name devtest \ --mount type=bind,source="$(pwd)"/target,target=/app \ nginx:latest	
19.	**Describe the characteristics and use cases of Tmpfs mounts.** Characteristics: ● When you create a container with a tmpfs mount, the container can create files outside the container's writable layer. ● As opposed to volumes and bind mounts, a tmpfs mount is temporary, and only persisted in the host memory. When the container stops, the tmpfs mount is removed, and files written there won't be persisted. ● Unlike volumes and bind mounts, you can't share tmpfs mounts between containers. ● This functionality is only available if you're running Docker on Linux. Use cases: ● If you're running Docker on Linux, you have a third option: tmpfs mounts. ● We can specify when running containers--tmpfsParameter or--mountParameters to use TMPFS mounts: $ docker run -d \ -it \ --name tmptest \ --mount type=tmpfs,destination=/app \ nginx:latest	6