Dockers and Containers Question Bank

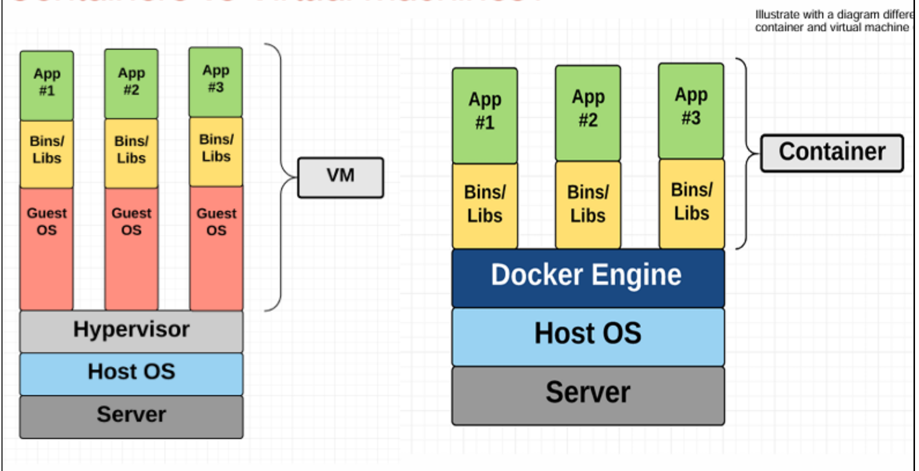
UNIT – V		Marks
1.	Discuss the problems solved by using the dockers and containers Docker and containers solve several problems in software development and deployment: 1. **Environment Consistency**: Containers ensure that applications run the same way in development, testing, and production environments. This eliminates the "it works on my machine" problem. 2. **Dependency Management**: Containers package all dependencies, libraries, and configurations needed for an application to run. This avoids conflicts between different versions of libraries and dependencies. 3. **Resource Efficiency**: Containers are lightweight and share the host OS kernel, making them more efficient in terms of resource usage compared to traditional virtual machines. 4. **Scalability**: Containers can be easily scaled up or down to handle varying loads. This makes it easier to manage and deploy applications in a microservices architecture. 5. **Isolation**: Containers provide process and network isolation, ensuring that applications run in their own isolated environments. This enhances security and stability. 6. **Portability**: Containers can run on any system that supports Docker, making it easy to move applications between different environments and platforms. 7. **Continuous Integration and Continuous Deployment (CI/CD)**: Containers streamline the CI/CD process by providing a consistent environment	6

for building, testing, and deploying applications.

8. **Simplified Management**: Docker provides tools for managing containers, images, and networks, making it easier to deploy and manage applications.

2. Discuss how containers differ from Hypervisor based virtualization. 6

Containers vs Virtual Machines?



Hypervisors virtualize hardware, Containers virtualize OS!!

Containers and hypervisor-based virtualization are two different approaches to running multiple isolated applications on a single host. Here are the key differences between them:

Containers

1. **Operating System Virtualization**: Containers virtualize the operating system, allowing multiple containers to share the same OS kernel while maintaining isolated user spaces.

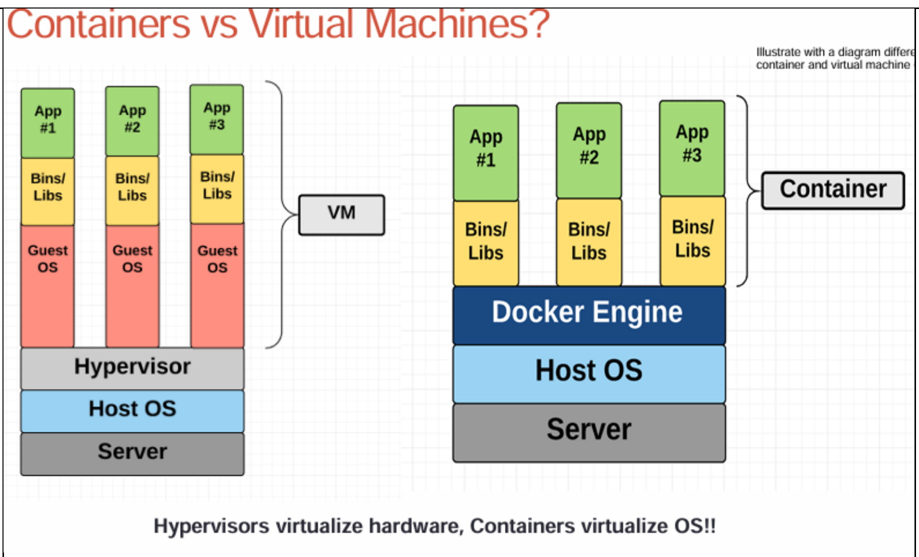
2. **Lightweight**: Containers are lightweight and have minimal overhead since they share the host OS kernel.

3. **Fast Startup**: Containers can start and stop quickly, making them ideal for rapid development and deployment.

4. **Resource Efficiency**: Containers use fewer resources

	compared to virtual machines, as they do not require a full OS for each instance. 5. **Portability**: Containers are highly portable and can run consistently across different environments, such as development, testing, and production. 6. **Isolation**: Containers provide process and file system isolation, but they share the same OS kernel, which can lead to potential security risks if the kernel is compromised. **Hypervisor-Based Virtualization** 1. **Hardware Virtualization**: Hypervisors virtualize the underlying hardware, allowing multiple virtual machines (VMs) to run on a single physical host. 2. **Heavyweight**: VMs are heavier and have more overhead since each VM runs its own full OS, including the kernel. 3. **Slower Startup**: VMs take longer to start and stop compared to containers due to the need to boot a full OS. 4. **Resource Intensive**: VMs require more resources, such as CPU, memory, and storage, as each VM includes a full OS. 5. **Isolation**: VMs provide strong isolation between instances, as each VM runs its own OS, making them more secure in case of a kernel compromise. 6. **Flexibility**: VMs can run different operating systems on the same physical host, providing greater flexibility in terms of OS choice.	
3.	Describe what difference does Docker bring to Containers. Docker brings several key differences and advantages to containers: 1. **Ease of Use**: Docker simplifies the process of	6

	creating, deploying, and managing containers. It provides a user-friendly interface and a set of tools that make it easy to work with containers. 2. **Portability**: Docker containers can run on any system that supports Docker, making it easy to move applications between different environments and platforms. This ensures consistency across development, testing, and production environments. 3. **Isolation**: Docker containers provide process and network isolation, ensuring that applications run in their own isolated environments. This enhances security and stability. 4. **Resource Efficiency**: Docker containers are lightweight and share the host OS kernel, making them more efficient in terms of resource usage compared to traditional virtual machines. 5. **Scalability**: Docker makes it easy to scale applications up or down to handle varying loads. This is particularly useful in a microservices architecture where different components of an application can be scaled independently. 6. **Integration with CI/CD**: Docker integrates seamlessly with continuous integration and continuous deployment (CI/CD) pipelines, making it easier to automate the build, test, and deployment processes. 7. **Community and Ecosystem**: Docker has a large and active community, as well as a rich ecosystem of tools and services that extend its functionality. This includes Docker Hub, a repository of pre-built container images that can be easily pulled and used.	
4.	Illustrate with a diagram differences between container and virtual machine	6



Containers vs Virtual Machines?

Illustrate with a diagram differe
container and virtual machine

Hypervisors virtualize hardware, Containers virtualize OS!!

Containers and virtual machines (VMs) are both technologies used to run multiple isolated applications on a single host, but they have some key differences. Here are the main differences between containers and virtual machines:

Containers

1. **Operating System Virtualization**: Containers virtualize the operating system, allowing multiple containers to share the same OS kernel while maintaining isolated user spaces.
2. **Lightweight**: Containers are lightweight and have minimal overhead since they share the host OS kernel.
3. **Fast Startup**: Containers can start and stop quickly, making them ideal for rapid development and deployment.
4. **Resource Efficiency**: Containers use fewer resources compared to virtual machines, as they do not require a full OS for each instance.
5. **Portability**: Containers are highly portable and can run consistently across different environments, such as development, testing, and production.
6. **Isolation**: Containers provide process and file system isolation, but they share the same OS kernel, which can lead to potential security risks if the kernel is

	compromised. **Virtual Machines** 1. **Hardware Virtualization**: Virtual machines virtualize the underlying hardware, allowing multiple VMs to run on a single physical host. 2. **Heavyweight**: VMs are heavier and have more overhead since each VM runs its own full OS, including the kernel. 3. **Slower Startup**: VMs take longer to start and stop compared to containers due to the need to boot a full OS. 4. **Resource Intensive**: VMs require more resources, such as CPU, memory, and storage, as each VM includes a full OS. 5. **Isolation**: VMs provide strong isolation between instances, as each VM runs its own OS, making them more secure in case of a kernel compromise. 6. **Flexibility**: VMs can run different operating systems on the same physical host, providing greater flexibility in terms of OS choice.	
5.	Differentiate between process, virtual machine and containers. **Processes** 1. **Isolation**: Processes have isolated address spaces but do not isolate files or networks. 2. **Lightweight**: Processes are lightweight and have minimal overhead. 3. **Resource Sharing**: Processes share the same operating system and resources with other processes on the host. 4. **Security**: Processes have limited security isolation	6

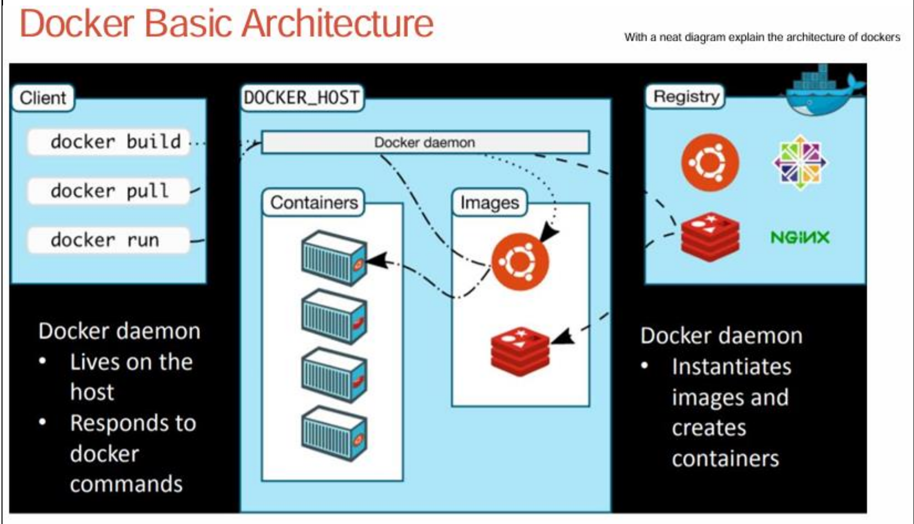
compared to containers and VMs.

Virtual Machines (VMs)

1. **Isolation**: VMs provide strong isolation by virtualizing the underlying hardware, allowing each VM to run its own full operating system.
2. **Heavyweight**: VMs are heavier and have more overhead since each VM includes a full OS.
3. **Resource Intensive**: VMs require more resources, such as CPU, memory, and storage, as each VM runs its own OS.
4. **Flexibility**: VMs can run different operating systems on the same physical host, providing greater flexibility in terms of OS choice.
5. **Security**: VMs provide strong security isolation, making them more secure in case of a kernel compromise.

Containers

1. **Isolation**: Containers provide process and file system isolation while sharing the same OS kernel with the host.
2. **Lightweight**: Containers are lightweight and have minimal overhead since they share the host OS kernel.
3. **Fast Startup**: Containers can start and stop quickly, making them ideal for rapid development and deployment.
4. **Resource Efficiency**: Containers use fewer resources compared to VMs, as they do not require a full OS for each instance.
5. **Portability**: Containers are highly portable and can run consistently across different environments, such as development, testing, and production.
6. **Security**: Containers provide process and file system isolation, but they share the same OS kernel, which

	can lead to potential security risks if the kernel is compromised.	
6.	With a neat diagram explain the architecture of dockers.  Docker Basic Architecture With a neat diagram explain the architecture of dockers Client docker build docker pull docker run DOCKER_HOST Docker daemon Containers Images Registry NGINX Docker daemon • Lives on the host • Responds to docker commands Docker daemon • Instantiates images and creates containers **Docker Architecture** 1. **Docker Client**: The Docker client is the primary interface for users to interact with Docker. It sends commands to the Docker daemon using the Docker CLI (Command Line Interface). Common commands include docker run, docker build, and docker pull. 2. **Docker Daemon (Docker Engine)**: The Docker daemon, also known as the Docker Engine, is the core component of Docker. It listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. The daemon communicates with the operating system's kernel to create and manage containers. 3. **Docker Images**: Docker images are read-only templates that contain the application code, runtime, libraries, and dependencies needed to run an application. Images are used to create Docker containers. They can be built from a Dockerfile or pulled from a Docker registry. 4. **Docker Containers**: Containers are lightweight,	8

portable, and isolated environments that run applications. They are created from Docker images and share the host OS kernel. Containers provide process and file system isolation, ensuring that applications run consistently across different environments.

5. **Docker Registries**: Docker registries are repositories that store Docker images. The most common registry is Docker Hub, a public registry where users can share and access images. Private registries can also be set up for internal use.
6. **Dockerfile**: A Dockerfile is a text document that contains a set of instructions for building a Docker image. It specifies the base image, application code, dependencies, and configuration settings needed to create the image.

7. Define the following terms: 5

i) Image ii) Container iii) Dockerfile iv)Docker Client v) Docker Daemon/Engine

i) Image

A Docker image is a read-only template that contains the application code, runtime, libraries, and dependencies needed to run an application. Images are used to create Docker containers. They can be built from a Dockerfile or pulled from a Docker registry.

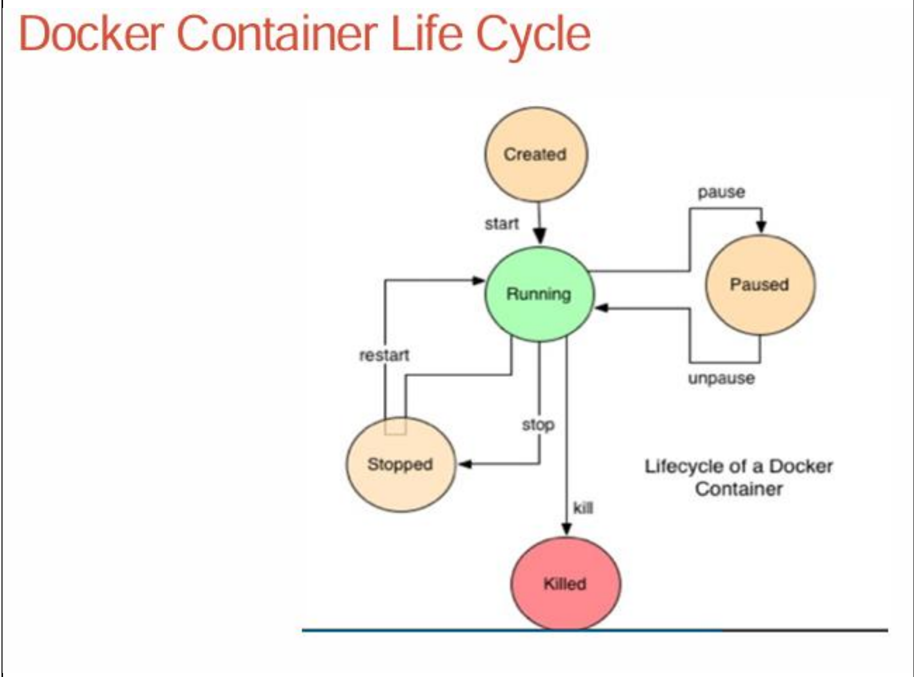
ii) Container

A Docker container is a lightweight, portable, and isolated environment that runs applications. Containers are created from Docker images and share the host OS kernel. They provide process and file system isolation, ensuring that applications run consistently across different environments.

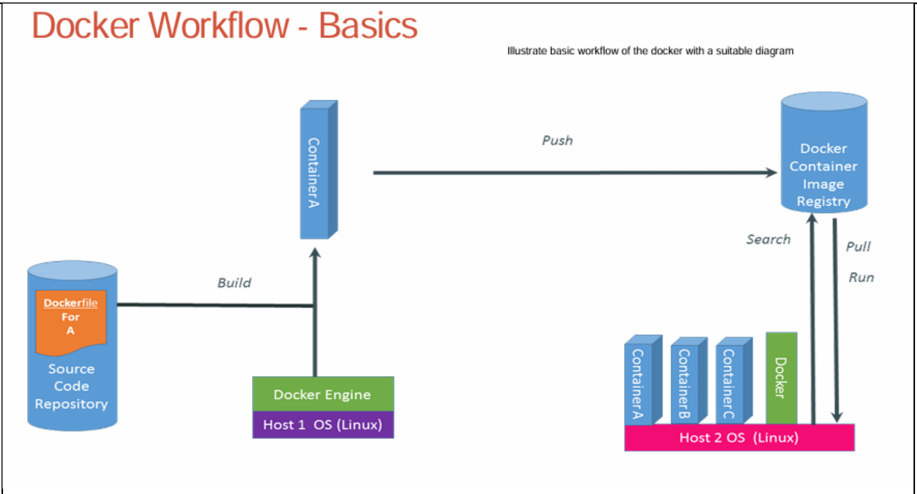
iii) Dockerfile

A Dockerfile is a text document that contains a set of instructions for

	building a Docker image. It specifies the base image, application code, dependencies, and configuration settings needed to create the image. **iv) Docker Client** The Docker client is the primary interface for users to interact with Docker. It sends commands to the Docker daemon using the Docker CLI (Command Line Interface). Common commands include docker run, docker build, and docker pull. **v) Docker Daemon/Engine** The Docker daemon, also known as the Docker Engine, is the core component of Docker. It listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. The daemon communicates with the operating system's kernel to create and manage containers.	
8.	List out the similarities and differences of docker containers between Linux and Windows operating systems **Similarities:** 1. **Isolation**: Both Linux and Windows containers provide process and network isolation. 2. **Portability**: Containers can be moved between different environments and platforms. 3. **Resource Efficiency**: Containers are lightweight and share the host OS kernel. 4. **Scalability**: Containers can be easily scaled up or down to handle varying loads. 5. **Integration with CI/CD**: Both Linux and Windows containers integrate seamlessly with CI/CD pipelines. **Differences:** 1. **Kernel**: Linux containers use the Linux kernel, while	8

	Windows containers use the Windows kernel. 2. **File System**: Linux containers use OverlayFS, while Windows containers use a different file system mechanism. 3. **Networking**: Linux containers use the Linux networking stack, while Windows containers use the Windows networking stack. 4. **Compatibility**: Linux containers can run on any system that supports Docker, while Windows containers require a Windows host. 5. **Performance**: Linux containers generally have better performance due to the more mature container ecosystem on Linux.	
9.	With a neat diagram describe Docker Container Life Cycle.  Docker Container Life Cycle Created start pause Paused Running restart unpause stop Stopped Lifecycle of a Docker Container kill Killed **Docker Container Life Cycle** 1. **Create**: The container is created from a Docker image but is not yet running. This is done using the docker create	7

	command. 2. **Start**: The container is started using the docker start command. The container begins executing its main process. 3. **Running**: The container is in a running state, executing its main process. This is the active state where the container performs its intended tasks. 4. **Pause**: The container's processes are paused using the docker pause command. The container remains in memory but does not execute any processes. 5. **Unpause**: The container's processes are resumed using the docker unpause command. The container returns to the running state. 6. **Stop**: The container is stopped using the docker stop command. The container's main process is terminated, and the container is no longer running. 7. **Restart**: The container is restarted using the docker restart command. This stops and then starts the container again. 8. **Kill**: The container is forcefully terminated using the docker kill command. This immediately stops the container's main process. 9. **Remove**: The container is removed using the docker rm command. This deletes the container from the Docker host.	
10.	Illustrate basic workflow of the docker with a suitable diagram	6

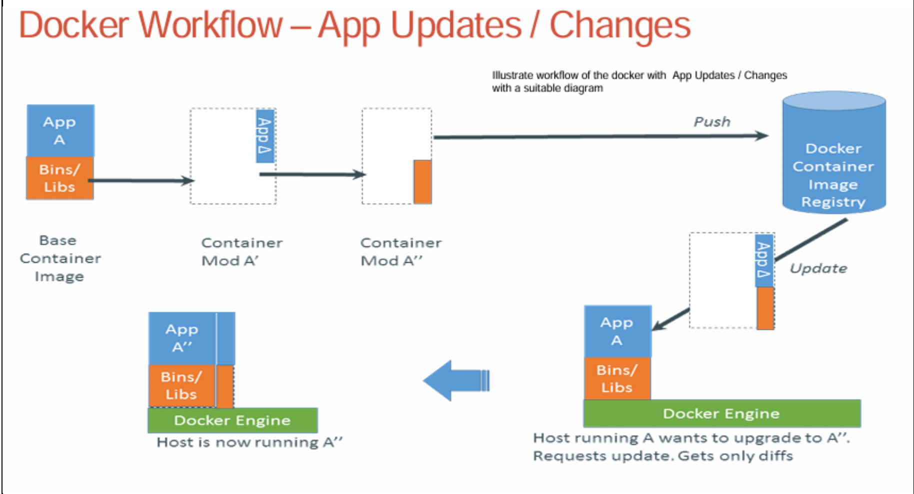


Illustrate basic workflow of the docker with a suitable diagram

Sure! Here's a basic workflow of Docker:

Docker Workflow

1. **Develop**: Developers write code and create a Dockerfile that defines the application's environment, dependencies, and configuration.
2. **Build**: The Dockerfile is used to build a Docker image. This image contains the application code and all necessary dependencies.
3. **Test**: The Docker image is tested in a container to ensure that the application works as expected. Automated tests can be run to verify functionality.
4. **Push**: Once the image passes testing, it is pushed to a Docker registry (e.g., Docker Hub) for storage and distribution.
5. **Deploy**: The Docker image is pulled from the registry and deployed to production environments. Containers are created from the image and run the application.
6. **Monitor**: The running containers are monitored for performance, errors, and other metrics. Logs and monitoring tools are used to track the application's health.
7. **Update**: When updates or changes are made to the

	application code, the Dockerfile is updated, and a new Docker image is built. 8. **Redeploy**: The updated Docker image is pushed to the registry and redeployed to the production environment. Containers running the old version are replaced with containers running the new version.	
11.	Illustrate workflow of the docker with App Updates / Changes with a suitable diagram  Docker Workflow – App Updates / Changes Illustrate workflow of the docker with App Updates / Changes with a suitable diagram App A Bins/ Libs Base Container Image App Δ Container Mod A' Container Mod A'' Push Docker Container Image Registry Update App Δ App A Bins/ Libs Docker Engine Host running A wants to upgrade to A''. Requests update. Gets only diffs App A'' Bins/ Libs Docker Engine Host is now running A'' **Docker Workflow with App Updates/Changes** 1. **Develop**: Developers write code and create a Dockerfile that defines the application's environment, dependencies, and configuration. 2. **Build**: The Dockerfile is used to build a Docker image. This image contains the application code and all necessary dependencies. 3. **Test**: The Docker image is tested in a container to ensure that the application works as expected. Automated tests can be run to verify functionality. 4. **Push**: Once the image passes testing, it is pushed to a Docker registry (e.g., Docker Hub) for storage and distribution. 5. **Deploy**: The Docker image is pulled from the registry and deployed to production environments. Containers are created from the image and run the application. 6. **Monitor**: The running containers are monitored for	8

	performance, errors, and other metrics. Logs and monitoring tools are used to track the application's health. 7. **Update**: When updates or changes are made to the application code, the Dockerfile is updated, and a new Docker image is built. 8. **Redeploy**: The updated Docker image is pushed to the registry and redeployed to the production environment. Containers running the old version are replaced with containers running the new version.	
12.	Describe different types of networks available for docker and containers. ☐ **None**: No network interface or IP address is provided to the container. Useful when the container doesn't need to provide a service over the network. ☐ **Host**: The container uses the Docker host's network stack, making services running on any port within the container directly accessible through the host's IP. ☐ **Bridge (default)**: Containers are connected to a private internal network on the Docker host. This is the default network type and requires linking for container DNS resolution. ☐ **Bridge (user-defined)**: Allows for multiple networks on the Docker host, useful for isolating different deployments of containers. ☐ **Overlay (swarm)**: Enables multi-host networking across Docker hosts that are part of a swarm. ☐ **Overlay (external key-value mechanism)**: Similar to the swarm overlay but uses an external key-value store for cluster management. ☐ **Custom network plugin**: If the existing networking options don't fit your needs, you can write your own network plugin using the Docker plugin API.	8
13.	Explain the various ways in which a user can configure	6

containers to be accessible

☐ **Port Mapping**: Map a port on the host to a port on the container. This allows external traffic to reach the container.

sh

```
docker run -d -p 8080:80 --name my_container nginx
```

In this example, port 8080 on the host is mapped to port 80 on the container.

☐ **Host Network**: Use the host's network stack. This makes the container's services accessible through the host's IP address.

sh

```
docker run -d --network host --name my_container nginx
```

☐ **Bridge Network**: Use Docker's default bridge network. Containers on the same bridge network can communicate with each other.

sh

```
docker run -d --network bridge --name my_container nginx
```

☐ **User-Defined Bridge Network**: Create a custom bridge network for better isolation and control.

sh

```
docker network create my_bridge

docker run -d --network my_bridge --name my_container
nginx
```

☐ **Overlay Network**: Use an overlay network for multi-host communication in a Docker Swarm.

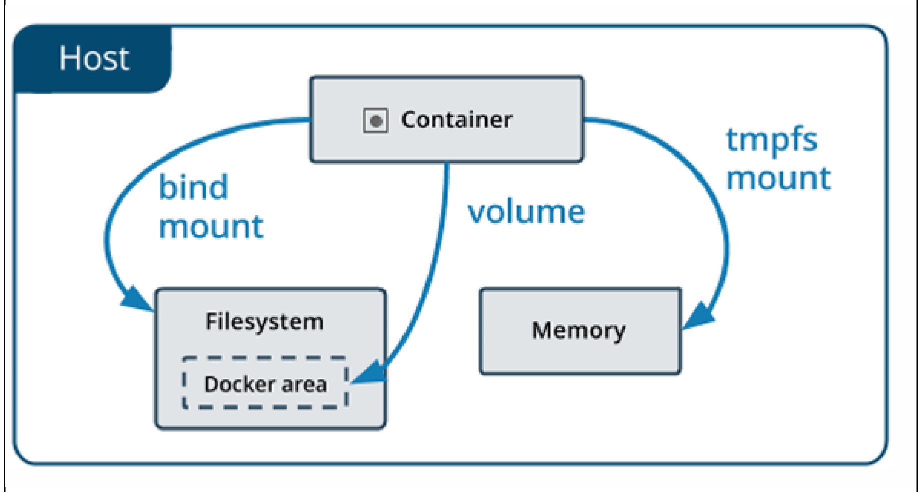
sh

```
docker network create --driver overlay my_overlay

docker run -d --network my_overlay --name my_container
nginx
```

	☐ **DNS Configuration**: Use Docker's built-in DNS service to resolve container names to IP addresses. sh docker run -d --name my_container --dns 8.8.8.8 nginx ☐ **Environment Variables**: Pass environment variables to the container to configure network settings. sh docker run -d --name my_contain	
14.	Show how a user can identify which network mode is being used by a container. To identify which network mode is being used by a Docker container, you can use the following commands: 1. **Inspect the container**: sh docker inspect <container_name_or_id> This command provides detailed information about the container, including its network settings. 2. **List all networks**: sh docker network ls This command lists all the networks available on the Docker host. 3. **Inspect a specific network**: sh docker network inspect <network_name> This command provides detailed information about a specific network, including which containers are connected to it.	6

15.	Explain why does a docker need a Union File System Docker uses a Union File System (UFS) to efficiently manage and layer file systems. Here's why it's essential: 1. **Layering**: UFS allows Docker to create layers for each change made to a container. This means that when you build a Docker image, each instruction in the Dockerfile creates a new layer. These layers are stacked on top of each other, forming a single unified file system. 2. **Efficiency**: By using layers, Docker can reuse existing layers across multiple containers. This reduces the amount of storage needed and speeds up the build process since only the new layers need to be created and stored. 3. **Copy-on-Write**: UFS supports the copy-on-write mechanism, which means that when a container writes to a file, a copy of the file is created in the container's writable layer. This ensures that the original file remains unchanged, allowing multiple containers to share the same base image without interfering with each other. 4. **Isolation**: Each container gets its own isolated file system, which is a combination of the base image layers and the container's writable layer. This isolation ensures that changes made in one container do not affect other containers. 5. **Performance**: UFS provides good performance by allowing containers to share common layers and only creating new layers when necessary. This reduces the overhead associated with creating and managing containers. Overall, the Union File System is a key component of Docker's architecture, enabling efficient storage, isolation,	6

	and management of container file systems	
16.	Discuss the types of docker mount with a neat diagram  Host Container bind mount volume tmpfs mount Filesystem Docker area Memory **Types of Docker Mounts** 1. **Volumes** ○ Created and managed by Docker. ○ Can be named or anonymous. ○ Supports volume drivers. ○ Decoupled from the host file system. ○ Provides persistent storage. ○ Suitable for sharing data across containers and for backup and restore. 2. **Bind Mounts** ○ Created by Docker if specified during container creation. ○ Data is stored on the host machine. ○ Directly accessible by the host. ○ Can have security implications. ○ Requires a specific directory structure on the	6

	host. ○ Suitable for sharing configuration files and for DevOps build lifecycles. 3. **Tmpfs Mounts** ○ Stores data only in memory. ○ Provides extremely fast access. ○ Data is lost when the container stops or restarts. ○ No cleanup needed. ○ Suitable for I/O sensitive projects and storing sensitive data temporarily.	
17.5	Describe the characteristics and use cases of volumes **Characteristics of Volumes** 1. **Created and Managed by Docker**: Volumes are created and managed by Docker, either explicitly using the docker volume create command or implicitly during container creation. 2. **Named or Anonymous**: Volumes can be named or anonymous. Named volumes are easier to manage and reference. 3. **Supports Volume Drivers**: Volumes support various volume drivers, allowing for different storage backends. 4. **Decoupled from Host File System**: Volumes provide a decoupled host-container file system architecture, making them independent of the host's directory structure. 5. **Persistent Storage**: Volumes offer persistent storage, ensuring data is retained across container	8

restarts and removals.

6. **Backup and Restore**: Volumes can be easily backed up and restored, making them suitable for data preservation.
7. **Cross-Platform Compatibility**: Volumes work on both Linux and Windows containers.
8. **High-Performance I/O**: Volumes are optimized for high-performance I/O operations, making them suitable for applications requiring fast data access.

Use Cases of Volumes

1. **Persistency Across Container Lifecycle**: Volumes ensure data persistency across the lifecycle of containers, even if the container is removed or restarted.
2. **Sharing Data Across Containers**: Volumes allow data sharing between multiple containers, facilitating communication and data exchange.
3. **Decoupled Storage**: Volumes provide a decoupled storage solution, enabling data to be stored outside the host machine, such as in central storage or the cloud.
4. **Backup and Restore**: Volumes can be used for backing up and restoring data, ensuring data integrity and availability.
5. **High-Performance Applications**: Volumes are suitable for applications requiring high-performance I/O operations, such as databases and file servers.
6. **DevOps and CI/CD Pipelines**: Volumes are used in DevOps and CI/CD pipelines to share configuration files, build artifacts, and other data between containers.
7. **Sensitive Data Storage**: Volumes can be used to store sensitive data securely, ensuring it is not exposed to the host machine.

	8. **Temporary Data Storage**: Volumes can be used for temporary data storage in standalone containers, providing fast and efficient data access.	
18.	Describe the characteristics and use cases of bind mounts **Characteristics of Bind Mounts** 1. **Created by Docker if Needed**: Bind mounts are created by Docker if they are specified during container creation. 2. **Maintained by Host**: The data in bind mounts is stored on the host machine and is directly accessible by the host. 3. **Security Implications**: Bind mounts can have security implications since they expose the host's file system to the container. 4. **Performance**: Bind mounts are performant and provide fast access to the host's file system. 5. **Specific Directory Structure**: Bind mounts require a specific directory structure on the host machine. **Use Cases of Bind Mounts** 1. **Sharing Configuration from Host**: Bind mounts are used to share configuration files from the host machine to the container. 2. **DevOps Build Lifecycle**: Bind mounts are used in DevOps build lifecycles to mount target folders into containers. 3. **Persistency Across Container Lifecycle**: Bind mounts ensure data persistency across the lifecycle of containers. 4. **Sharing Data Across Containers**: Bind mounts allow data sharing between multiple containers,	8

	facilitating communication and data exchange.	
19.	Describe the characteristics and use cases of Tempfs mounts **Characteristics of Tempfs Mounts** 1. **In-Memory Storage**: Tempfs mounts store data only in memory, not on the disk. 2. **Fast Access**: Since data is stored in memory, Tempfs mounts provide extremely fast access. 3. **Ephemeral**: Data in Tempfs mounts is temporary and is lost when the container stops or restarts. 4. **No Cleanup Needed**: Since data is not stored on disk, there is no need for cleanup. **Use Cases of Tempfs Mounts** 1. **I/O Sensitive Projects**: Tempfs mounts are ideal for projects that require fast I/O operations. 2. **Storing Sensitive Data**: Tempfs mounts are great for storing sensitive data that should not be written to disk. 3. **Standalone Containers**: Tempfs mounts are useful for standalone containers that need to store runtime information temporarily.	6