

Sr.No.	Topics	Duration(Mins)	Session No(2 Hours)	Session No.(4 Hours)
1	Introduction to Virtualization and Containerization	20	1	1
2	What is Containerization?	20	1	1
3	Docker Architecture	30	1	1
4	Overview of Docker Hub	20	1	1
5	Docker Installation	30	1	1
6	Docker Commands	40	2	1
7	Container Modes	20	2	1
8	Port Binding	20	2	1
9	Dockerfile	40	2	1
10	Managing Docker Images	20	3	2
11	Running and managing Containers	30	3	2
12	Docker Volume	15	3	2
13	Docker Compose	30	3	2
14	Overview of Docker Swarm	15	3	2
15	Practical	60	4	2

# Containerization, Docker, and Docker Hub

## a. Introduction to Virtualization and Containerization

### Information

- Virtualization and containerization are two technologies that enable the efficient and flexible utilization of computing resources.

### A. Virtualization:

- Virtualization is the process of creating a virtual version of a physical resource, such as a server, operating system, storage device, or network, using software.
- It allows multiple virtual instances or machines to run on a single physical server, effectively utilizing the available hardware resources.
- Each virtual machine (VM) operates independently and can run its own operating system and applications.

Key concepts in virtualization include:

i. Hypervisor:

- Also known as a virtual machine monitor, the hypervisor is a software or firmware layer that enables the creation and management of virtual machines.
- It controls the allocation of physical resources to the virtual machines and facilitates their communication with the underlying hardware.

ii. Virtual Machine:

- A virtual machine is an isolated and self-contained software environment that emulates a complete computer system.
- It consists of a virtualized set of hardware resources, including CPU, memory, storage, and network interfaces.
- Benefits of virtualization include improved resource utilization, cost savings, scalability, isolation, and simplified management of the virtualized environment.

B. Containerization:

- Containerization is a lightweight form of virtualization that allows the packaging and isolation of applications and their dependencies into self-contained units called containers.
- Containers are portable and can run consistently across different computing environments, such as development, testing, and production systems.

Key concepts in containerization include:

i. Container:

- A container is a standalone and executable software package that encapsulates an application along with its runtime dependencies and libraries.
- It provides a consistent and isolated environment for the application to run, ensuring that it behaves the same regardless of the underlying infrastructure.

ii. Container Engine:

- The container engine, such as Docker, is responsible for creating, managing, and running containers on a host system.
- 

2. What is Containerization?

## Information

- Containerization is a software technology that enables the packaging and running of applications and their dependencies in isolated and portable environments called containers.
- A container provides a self-contained and lightweight runtime environment for an application, including its code, runtime libraries, system tools, and settings.
- In containerization, the application and its dependencies are bundled together into a single container image, which can be deployed and run consistently across different computing environments, such as development machines, testing environments, and production servers.
- Containers isolate the application and its dependencies from the underlying infrastructure, ensuring that the application behaves the same regardless of the host system.
- Containers are based on containerization platforms or engines, such as Docker or Kubernetes.
- These platforms provide the necessary tools and runtime environments to create, manage, and deploy containers.

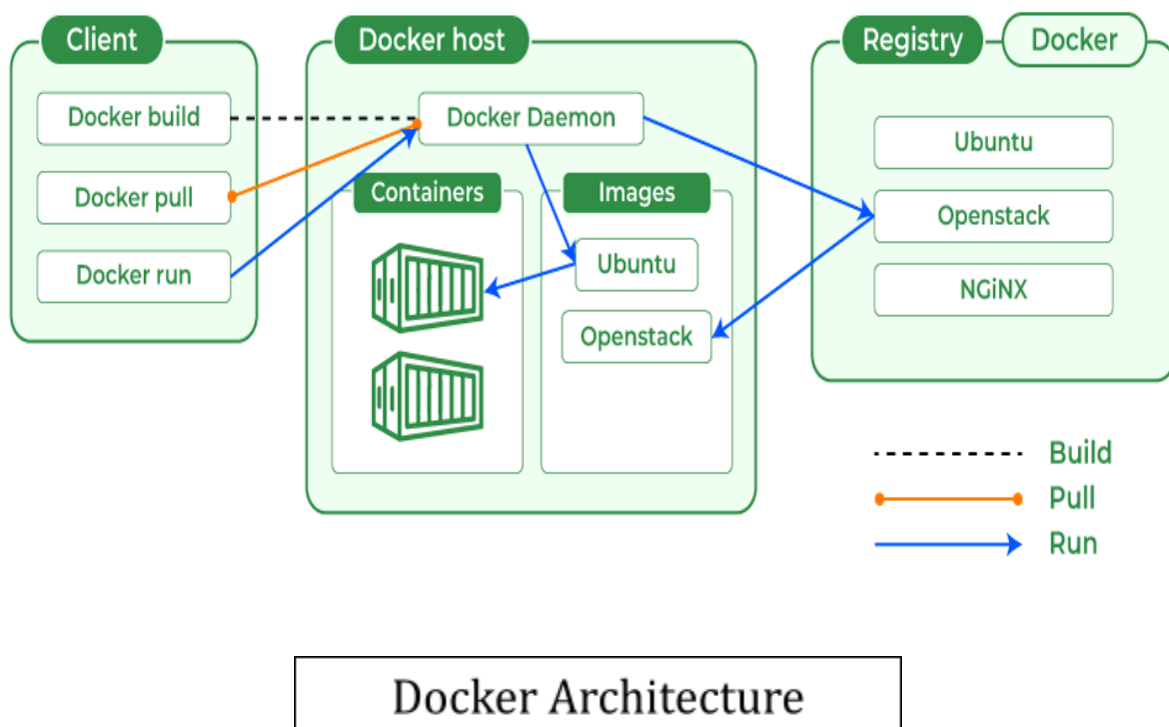
- Containers are typically created from a container image, which is a lightweight, standalone, and executable package that contains the application code and its dependencies.



### 3. Docker Architecture

#### Information

- Docker is an open-source containerization platform that enables the creation, deployment, and management of containers.
- It follows a client-server architecture, consisting of various components that work together to run and manage containers.



a. Docker Daemon (dockerd):

- The Docker Daemon, called dockerd, is the core component of Docker.
- It runs on the host machine and listens for Docker API requests.

- It manages the lifecycle of containers, including creating, running, stopping, and deleting them.
- The Docker Daemon also interacts with the underlying operating system's kernel to manage container resources.

b. Docker Client:

- The Docker Client is a command-line interface (CLI) tool that allows users to interact with the Docker Daemon.
- It sends commands and requests to the Docker Daemon using the Docker Remote API.
- The Docker Client can be run on the same machine as the Docker Daemon or on a remote machine that connects to the Docker Daemon over the network.

c. Docker Images:

- Docker Images are the building blocks of containers.
- An image is a read-only template that contains the application code, dependencies, libraries, and other files needed to run a container.
- Images are created from a set of instructions specified in a Dockerfile, which defines the image's configuration and the steps to build it.



- Images can be stored in local or remote registries, such as Docker Hub or private registries.

d. Docker Registry:

- A Docker Registry is a repository for storing and distributing Docker images.
- The most commonly used registry is Docker Hub, which hosts a vast collection of publicly available Docker images.
- Organizations can also set up private registries to store their own custom images.
- Docker images can be pulled from a registry to run containers or pushed to a registry for sharing with others.

e. Docker Containers:

- A Docker Container is an isolated and lightweight runtime environment that runs an application and its dependencies.
- Containers are created from Docker Images and run as separate processes on the host machine.
- Each container is isolated from other containers and the host system, providing security and resource isolation.
- Containers can be started, stopped, restarted, and deleted using Docker commands or API calls.

f. Docker Network:

- Docker provides networking capabilities to allow containers to communicate with each other and with the external network.
- Docker Network enables the creation of virtual networks and the assignment of IP addresses to containers.
- Containers can be connected to specific networks to enable communication between them, or they can be exposed to the host network for external access.

g. Docker Volumes:

- Docker Volumes provide persistent storage for containers.
  - Volumes allow data to be shared and stored independently of the container's lifecycle.
  - They can be used to store databases, configuration files, logs, or any other data that needs to persist across container restarts or upgrades.
- 
- The Docker architecture combines these components to provide a powerful and flexible containerization platform.
  - It simplifies the process of creating, deploying, and managing applications in isolated and portable containers.

## 4. Overview of Docker Hub

### Information

- Docker Hub is a cloud-based repository and marketplace for Docker images.
  - It provides a central location for developers, organizations, and the Docker community to store, share, and access Docker images.
- a. Image Repository:
- Docker Hub serves as a repository for Docker images. It hosts a vast collection of publicly available images that can be used as a base for building containers.
  - These images cover a wide range of applications, frameworks, programming languages, and system configurations.

- Users can search for images based on their specific needs and requirements.

#### b. Official Images:

- Docker Hub hosts official images that are maintained and verified by the creators of the respective software or framework.
- These images undergo rigorous testing and are considered reliable and trustworthy.
- Official images are typically tagged with the name of the software or project they represent.

## 5. Docker Installation

### Information

- To install and set up Docker on your system, follow these general steps:

#### i. Check System Requirements:

- Ensure that your system meets the minimum requirements for running Docker.

- Docker is available for different operating systems like Windows, macOS, and various Linux distributions. Check the official Docker documentation for specific system requirements.

ii. Download Docker:

- Visit the official Docker website (<https://www.docker.com/get-started>) and download the appropriate Docker version for your operating system.
- Follow the instructions for the specific platform you are using.

iii. Install Docker:

- Run the installer or package that you downloaded in the previous step.
- The installation process will vary depending on your operating system.
- Follow the on-screen instructions to complete the installation.

iv. Start Docker:

- After the installation is complete, start the Docker service or application.

- On Windows and macOS, Docker may require you to log in with your Docker account.

v. Verify Installation:

- Open a terminal or command prompt and run the following command to verify that Docker is installed correctly:

```
docker version
```

- This command will display the version of Docker installed on your system, along with other information.

vi. Test Docker:

- To test if Docker is working properly, run the following command to pull and run a simple "Hello World" container:

```
docker run hello-world
```

- Docker will download the "Hello World" image if it's not already available on your system and run a container based on that image.
- If everything is set up correctly, you will see a message indicating that Docker is running and working properly.

vii. Optional Configuration:

- Depending on your needs, you may want to configure additional settings for Docker, such as managing Docker images and containers storage locations, adjusting resource limits, configuring network settings, etc.
  - Refer to the Docker documentation for more details on configuration options.
- 
- Once Docker is installed and set up, you can start using it to build, deploy, and manage containers on your system.
  - You can pull existing Docker images from registries, create your own custom images using Dockerfiles, and run containers based on those images.
  - Docker provides a powerful command-line interface (CLI) and a graphical user interface (GUI) tool called

Docker Desktop (for Windows and macOS) to interact with Docker and manage containers and images.

## 6. Docker Commands

### Information

- Docker provides a powerful command-line interface (CLI) that allows you to interact with the Docker engine and manage containers, images, networks, volumes, and other Docker components.
- a. `docker run`: This command is used to create and run a container based on a Docker image.
- For example:

```
docker run image_name
```

- b. `docker pull`: This command is used to download a Docker image from a registry.
- For example:



```
docker pull image_name
```

c. **docker build**: This command is used to build a Docker image from a Dockerfile.

- For example:

```
docker build -t image_name .
```

d. **docker images**: This command lists all the Docker images available on your system.

- For example:

```
docker images
```

e. **docker ps**: This command lists all the running containers. Adding the '-a' option will display all containers, including stopped ones.

- For example:

```
docker ps
```

f. `docker stop`: This command is used to stop a running container.

- You need to provide either the container ID or the container name.
- For example:

```
docker stop container_id/container_name
```

g. `docker rm`: This command is used to remove one or more containers.

- You can specify either the container ID or the container name.
- For example:

```
docker rm container_id/container_name
```

h. `docker rmi`: This command is used to remove one or more Docker images.

- You need to provide the image ID or the image name.
- For example:

```
docker rmi image_id/image_name
```

i. `docker exec`: This command is used to execute a command inside a running container.

- For example:

```
docker exec container_id/container_name  
command
```

- j. `docker logs`: This command is used to view the logs of a container.
- You need to provide the container ID or the container name.
  - For example:

```
docker logs container_id/container_name
```

- These are just a few examples of commonly used Docker commands.
- Docker provides a rich set of commands to manage containers, images, networks, volumes, and more.
- You can explore the Docker documentation or use the '`docker -help`' command to see a complete list of available commands and their options.

## 7. Container Modes

### Information

- In Docker, there are two main container modes or run modes:

a. Interactive Mode:

- Interactive mode allows you to interact with a container's shell or command prompt.
- When running a container in interactive mode, you can provide input and receive output directly from the container's shell or the application running inside it.
- This mode is commonly used for troubleshooting, debugging, or executing commands inside a container.
- To run a container in interactive mode, you can use the '-it' option with the 'docker run' command.
- For example:

```
docker run -it image_name
```

- Detached mode, also known as background mode, allows you to run a container in the background without attaching to its shell or command prompt.

- When running a container in detached mode, it runs as a background process, and you won't see the output or interact with the container directly.
- To run a container in detached mode, you can use the '-d' option with the 'docker run' command.
- For example:

```
docker run -d image_name
```

- In detached mode, you can still access the logs and interact with the container later using various Docker commands such as docker logs, docker exec, etc.
- Additionally, there is another mode called "Swarm Mode" in Docker, which is used for orchestration and managing a cluster of Docker nodes.
- Swarm Mode allows you to create and manage a swarm of Docker nodes as a single entity, enabling high availability, load balancing, and scaling of containerized applications.
- However, swarm mode is beyond the scope of container modes for individual containers.

- These container modes provide flexibility in how you interact with and run Docker containers, depending on your specific use cases and requirements.

## 8. Port Binding

### Information

- Port binding in Docker is the process of mapping ports on the host machine to ports exposed by containers.
- It allows external systems to communicate with containers through specific ports.
- Port binding is important for enabling network connectivity and accessing services running inside containers.

#### a. Container Port:

- Containers can expose specific ports to allow communication with services or applications running inside them.
- These ports are defined in the Dockerfile or the container runtime configuration.

b. Host Port:

- Host ports are the ports on the host machine that are used to communicate with containers.
- These ports can be any available port on the host system.

c. Port Binding:

- Port binding is the process of associating a container's exposed port with a host port.
- It enables external systems to access services running inside the container by communicating with the corresponding host port.

d. Binding Syntax:

- Port binding is specified using the '-p' flag when running a container with the docker run command.
- The syntax is as follows:

```
docker run -p host_port:container_port image_name
```



- 'host\_port' refers to the port on the host machine that will be bound to the container.
- 'container\_port' refers to the port exposed by the container.
- For example, to bind port 8080 of the host machine to port 80 of a container, the command would be:

```
docker run -p 8080:80 image_name
```

- This configuration allows external systems to access the container's service using 'http://localhost:8080'.

e. Multiple Port Binding:

- It's possible to bind multiple ports by specifying multiple '-p' flags in the docker run command.
- Each flag binds a specific port.
- For example:

```
docker run -p host_port1:container_port1 -p  
host_port2:container_port2 image_name
```

- This enables multiple services running inside the container to be accessible on different ports of the host machine.
- Port binding is crucial for exposing containerized applications or services to the outside world.
- It enables seamless communication between containers and the host machine or other systems

## 9. Dockerfile

### Information

- A Dockerfile is a text file that contains a set of instructions used to build a Docker image.
- It provides a standardized and reproducible way to define the environment and dependencies required for running an application inside a Docker container.

#### a. Base Image:

- The Dockerfile typically starts with a base image that serves as the starting point for building the application's container.
- The base image contains a minimal operating system or a specific runtime environment.

b. Instructions:

- Dockerfile instructions are used to define the steps required to build the image.
- These instructions are executed sequentially and create layers within the image.
- Some commonly used instructions include:
  - FROM: Specifies the base image.
  - RUN: Executes commands during the image build process.
  - COPY or ADD: Copies files or directories from the host machine to the image.
  - WORKDIR: Sets the working directory inside the container.
  - EXPOSE: Declares the ports on which the container listens.
  - CMD or ENTRYPOINT: Specifies the command to run when the container starts.

c. Building the Image:

- The Dockerfile is used as input to the 'docker build' command, which builds the Docker image based on the instructions in the file.
- Each instruction in the Dockerfile creates a new layer in the image, allowing for efficient caching and incremental builds.

d. Image Layers:

- Docker images are composed of multiple layers, where each layer represents a specific instruction in the Dockerfile.
- This layer-based approach enables faster and more efficient image builds and facilitates reusability.

e. Environment Configuration:

- The Dockerfile can include instructions to configure environment variables, set up networking, install software packages, copy application code, and perform other configuration tasks required for the application to run correctly inside the container.

f. Best Practices:

- When writing a Dockerfile, it's important to follow best practices such as keeping the image lightweight, minimizing the number of layers, using specific versions

of dependencies, and ensuring proper cleanup of temporary files to optimize image size and security.

- The Dockerfile provides a declarative and version-controlled way to define the build process for a Docker image.
- It simplifies the deployment and distribution of applications, allowing them to be easily reproduced and run in different environments using Docker containers.



## Writing Docker File

- Here's an example of a simple Dockerfile for a Node.js application:

```
# Set the working directory inside the container
WORKDIR /app
```



- In this example, the Dockerfile starts with the 'FROM' instruction, which specifies the base image to use (in this case, 'node:14').
- The 'WORKDIR' instruction sets the working directory inside the container to '/app'.

- The 'COPY' instruction is used to copy the 'package.json' and 'package-lock.json' files from the host machine to the container's working directory.
- The RUN instruction then runs 'npm install' to install the application's dependencies.
- Next, the remaining application code is copied to the working directory using 'COPY . .'.
- The 'EXPOSE' instruction declares that the application will listen on port 3000.
- Finally, the 'CMD' instruction specifies the command to run when the container starts, which in this case is 'npm start'.
- To build an image from this Dockerfile, navigate to the directory containing the Dockerfile and run the following command:

```
docker build -t my-app-image .
```

- This command builds the Docker image and tags it as 'my-app-image'.
- The '.' indicates that the build context is the current directory.
- Once the image is built, you can run a container using the image:

```
docker run -p 3000:3000 my-app-image
```

- This command starts a container from the 'my-app-image' image and maps port 3000 of the host machine to port 3000 of the container.
- Note: This is a simplified example, and depending on your specific application and requirements, you may need to modify the Dockerfile accordingly.

## 10. Managing Docker Images



## Information

- Managing Docker images involves tasks such as listing images, pulling images from registries, building custom images, tagging images, pushing images to registries, and removing unnecessary images.
- Here are some commonly used Docker commands for managing Docker images:

a. 'docker images':

- Lists all the Docker images available on your system.

b. 'docker pull <image\_name>':

- Pulls an image from a registry.
- If the image is not available locally, it will be downloaded from the specified registry.

c. 'docker build -t <image\_name>' .:

- Builds a Docker image from a Dockerfile located in the current directory.
- The '-t' flag is used to tag the image with a name.

d. 'docker tag <existing\_image\_name>  
<new\_image\_name>':

- Tags an existing image with a new name.
- This is useful when you want to assign a different name or version to an existing image.

e. 'docker push <image\_name>':

- Pushes an image to a registry, making it available for others to download.
- You need appropriate permissions to push images to a registry.

f. 'docker rmi <image\_name>':

- Removes a specific image from your local system.
- You need to specify either the image ID or the image name.

g. 'docker rmi \$(docker images -q)':

- Removes all the unused and dangling images from your local system.

h. 'docker save -o <output\_file.tar> <image\_name>':

- Saves a Docker image as a tar file.
- This is useful when you want to share the image with others or move it to another system.

i. 'docker load -i <input\_file.tar>':

- Loads a Docker image from a tar file.
- This command is used to restore a previously saved image.

j. 'docker history <image\_name>':

- Shows the history of an image, including the layers and commands used to build it.
- These commands help you manage your Docker images, including pulling, building, tagging, pushing, and removing them.
- Proper management of images ensures efficient use of resources and simplifies the deployment and distribution of containerized applications.

## Creating Docker Image :

- To create a Docker image, you need to follow these steps:

a. Create a Dockerfile:

- A Dockerfile is a text file that contains a set of instructions for building the Docker image.
- It specifies the base image, copies files, installs dependencies, configures the environment, and defines the commands to run within the container.
- Create a file named "Dockerfile" (no extension) in your project directory.

b. Write the Dockerfile:

- Open the Dockerfile in a text editor and define the instructions based on your application requirements.
- Here's an example of a simple Dockerfile for a Node.js application:

```
# Set the working directory inside the
container
WORKDIR /app

# Copy package.json and
package-lock.json to the working directory
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the application code to the
working directory
COPY ..
```

c. Build the Docker image:

- Open a terminal or command prompt, navigate to the directory containing the Dockerfile, and run the following command to build the Docker image:

```
docker build -t image_name .
```

- Replace "image\_name" with the desired name for your Docker image.
- The "." indicates that the build context is the current directory.
- Docker will read the Dockerfile and execute the instructions to build the image.

d. Verify the Docker image:

- After the build process completes successfully, you can verify that the image has been created by running the following command:

```
docker images
```

- You should see the newly built image listed among the available Docker images.
- That's it! You have created a Docker image based on the instructions in the Dockerfile. You can now use this image to run containers and deploy your application.

## 11. Running and Managing Containers

### Information

- Running and managing containers is an essential part of working with Docker.
  - Here are some common Docker commands to help you run and manage containers:
- a. docker run: This command is used to create and start a new container from an image. For example:

```
docker run image_name
```

- b. `docker ps`: This command lists all the running containers. Adding the `'-a'` option will display all containers, including stopped ones. For example:

```
docker ps
```

- c. `docker start`: This command starts one or more stopped containers. You need to provide either the container ID or the container name. For example:

```
docker start container_id/container_name
```

- d. `docker stop`: This command stops a running container. You need to provide either the container ID or the container name. For example:

```
docker stop container_id/container_name
```



- e. docker restart: This command restarts a running container. You need to provide either the container ID or the container name. For example:

```
docker restart container_id/container_name
```

- f. docker pause: This command pauses a running container, suspending all processes within it. For example:

```
docker pause container_id/container_name
```

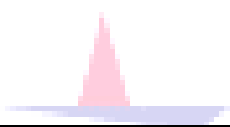
- g. docker unpause: This command resumes a paused container, allowing the processes to continue. For example:

```
docker unpause container_id/container_name
```

- h. `docker rm`: This command removes one or more containers. You can specify either the container ID or the container name. For example:

```
docker rm container_id/container_name
```

- i. `docker logs`: This command displays the logs of a container. You need to provide the container ID or the container name. For example:



```
docker logContainerization, Docker and Docker Hubs  
container_id/container_name
```

- j. `docker exec`: This command allows you to execute a command inside a running container. For example:

```
docker exec container_id/container_name command
```

- These commands provide the basic functionality to run and manage Docker containers.
- There are many more Docker commands available for various container management tasks, such as attaching to a container's shell, copying files to/from a container, managing container networks, etc.
- You can refer to the Docker documentation or use the 'docker -help' command to explore additional commands and their options.

#### Running the Container :

- To run a Docker container, you need to use the 'docker run' command.
- Here's the basic syntax:

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

- Let's break down the command and its options:
  - a. **docker run**: This is the base command used to run a container.
  - b. **OPTIONS**: These are optional flags that you can provide to customize the container's behavior. Some commonly used options include:
    - **'-d' or '--detach'**: Runs the container in the background (detached mode).
    - **'-p' or '--publish'**: Publishes a container's port(s) to the host. You can specify the port mapping in the format **'host\_port:container\_port'**.
    - **'-v' or '--volume'**: Mounts a volume or directory from the host into the container. You can specify the mapping in the format **'host\_path:container\_path'**.
    - **'--name'**: Assigns a name to the container.
    - **'-e' or '--env'**: Sets environment variables inside the container.
    - **'--network'**: Connects the container to a specific Docker network.
  - c. **IMAGE**: This is the name or ID of the Docker image you want to run as a container.

d. **COMMAND** and **ARG**: These are optional parameters that specify the command to run inside the container and any arguments for that command. If not provided, the container will run the default command defined in the image.

- Here's an example command to run a basic Nginx web server container:

```
docker run -d -p 80:80 nginx
```

- This command starts an Nginx container in detached mode ('-d'), maps port 80 of the host to port 80 of the container ('-p 80:80'), and uses the nginx image.
- Once the container is running, you can access the Nginx web server by opening a web browser and navigating to 'http://localhost or http://<host-ip>'.

- Remember to replace 'nginx' with the actual image name and adjust the options based on your specific requirements.
- Note: If the specified image is not available locally, Docker will automatically download it from a registry before running the container.

## 12. Docker Volume :

### Information

- A Docker volume is a directory or filesystem that can be mounted into one or more containers.
- Volumes provide persistent storage for containers, allowing data to be shared and retained even if the container is stopped or restarted.

#### a. Data Persistence:

- By using volumes, you can ensure that data stored within a container is not lost when the container is destroyed or replaced.
- Volumes provide a durable storage solution for containers.

#### b. Mounting Volumes:

- Volumes can be mounted into containers at specific paths, allowing the container to read from and write to the volume.
- Multiple containers can share the same volume, enabling data sharing and communication.

c. Volume Types:

- Docker supports different types of volumes, including host-mounted volumes, named volumes, and anonymous volumes.

d. Host-mounted volumes:

- With host-mounted volumes, a directory on the host machine is mounted into the container.
- This allows the container to read and write data directly to the host's filesystem.

e. Named volumes:

- Named volumes are created and managed by Docker.
- They have a specific name assigned to them and are stored in a dedicated location within Docker's storage infrastructure.
- Named volumes are independent of the host filesystem and can be shared between containers.

f. Anonymous volumes:

- Anonymous volumes are similar to named volumes but do not have an explicitly assigned name.
- They are created automatically by Docker and are typically used for temporary or ephemeral data storage.

g. Volume Creation:

- Volumes can be created explicitly using the docker volume create command or implicitly when a container is created and a volume is referenced.

h. Volume Mounting Syntax:

- Volumes can be mounted in the Docker run command using the '-v' or '--volume' flag, followed by the volume source and destination path.
- For host-mounted volumes: '-v /host/path:/container/path'
- For named volumes: '-v volume\_name:/container/path'
- For anonymous volumes: '-v /container/path'

i. Data Sharing and Persistence:

- Volumes provide a way to share data between containers, even when they are running on different hosts.



- They also allow for data persistence, as the data stored in the volume is retained even if the container is stopped or restarted.
- Using Docker volumes, you can manage and persist data across containers, making it easier to separate application logic from the underlying storage and ensuring data integrity and availability in your Docker environment.

#### Data Persistence using Docker Volume:

- Data persistence in Docker can be achieved using Docker volumes.
- Docker volumes provide a way to store and manage data separately from the containers, ensuring that the data persists even when the containers are stopped, removed, or replaced.
- Here's how you can achieve data persistence using Docker volumes:

##### a. Create a Volume:

- Use the 'docker volume create' command to create a named volume. For example:

```
docker volume create mydata
```

b. Mount the Volume in Containers:

- When running a container, specify the volume to be mounted using the '-v' or '--volume' flag. For example:

```
docker run -v mydata:/app/data myimage
```

- In this example, the 'mydata' volume is mounted at the '/app/data path' inside the container.

c. Store Data in the Volume:

- Any data written to the mounted volume path (/app/data in the example above) within the container will be stored in the volume.

- This data will persist even if the container is removed or replaced.

d. Mounting Volumes in Multiple Containers:

- You can mount the same volume in multiple containers, allowing them to access and share the same data.
- Simply specify the same volume name and mount path when running the containers.

e. Backup and Restore:

- Docker volumes can be backed up by copying the data from the volume directory on the host machine.
- To restore the data, simply copy it back to the appropriate volume directory.

f. Removing Volumes:

- To remove a volume, use the 'docker volume rm' command followed by the volume name.
- Note that removing a volume will delete all the data stored in it.

- Docker volumes provide a convenient way to achieve data persistence in Docker containers.
- They decouple the data from the container lifecycle, allowing you to manage and reuse data independently.

- This approach makes it easier to handle data storage and ensures that your data is retained even when containers are modified or replaced.

## 13. Docker Compose

### Information

- Docker Compose is a tool that allows you to define and manage multi-container Docker applications.
- It uses a YAML file to specify the services, networks, and volumes required for your application, making it easier to define, configure, and run complex containerized environments.
- With Docker Compose, you can:
  - a. Define Services:
    - In a Docker Compose file, you can define the services that make up your application, specifying the image to use, environment variables, exposed ports, mounted volumes, and other configurations.
    - Each service can have its own configuration, allowing you to compose multiple containers that work together.

b. Manage Networks:

- Docker Compose creates a default network for your application, allowing the services defined in the Compose file to communicate with each other using service names as hostnames.
- You can also define custom networks for more complex networking scenarios.

c. Configure Volumes:

- Docker Compose enables you to define volumes for your services, allowing you to persist data across container restarts or share data between services.
- Volumes can be defined as named volumes, anonymous volumes, or host-mounted volumes.

d. Define Dependencies:

- Docker Compose allows you to specify dependencies between services, ensuring that dependent services start before the services that require them.
- This helps to orchestrate the startup and shutdown order of containers within your application.

e. Build and Deploy:

- Docker Compose provides a simple command-line interface for building and deploying your application.

- You can use commands like 'docker-compose up' to start the services defined in your Compose file, 'docker-compose down' to stop and remove the containers, and 'docker-compose build' to build or rebuild the images defined in the Compose file.

f. Manage Environments:

- Docker Compose allows you to define environment variables for your services, making it easy to configure different environments (e.g., development, staging, production) using the same Compose file.
- By using Docker Compose, you can define your application's infrastructure as code, making it easier to manage and reproduce your containerized environment across different machines and environments.
- Docker Compose simplifies the process of running and managing multi-container applications, enabling you to focus on developing and deploying your application with ease.

## 14. Overview of Docker Swarm

- Docker Swarm is a native clustering and orchestration tool provided by Docker.

- It allows you to create and manage a swarm of Docker nodes (hosts) to deploy and scale your containerized applications across multiple machines.
- Docker Swarm provides features for load balancing, service discovery, rolling updates, and fault tolerance.

a. Swarm Mode:

- Docker Swarm is built into Docker Engine and is activated by enabling Swarm mode.
- Once Swarm mode is enabled on a Docker host, it becomes a Swarm manager and can manage other Docker hosts as worker nodes.

b. Manager and Worker Nodes:

- In a Docker Swarm, you have one or more manager nodes and one or more worker nodes.
- Manager nodes are responsible for managing the cluster, storing the swarm state, and orchestrating the deployment of services.
- Worker nodes are where the containers are actually run.

c. Service Definition:

- With Docker Swarm, you define services instead of individual containers.
- A service represents a desired state for running containers, including the number of replicas, container

image, exposed ports, mounted volumes, and other configurations.

d. Scaling:

- Docker Swarm makes it easy to scale your services horizontally by increasing or decreasing the number of replicas.
- You can scale a service up or down depending on the demand and resource availability.

e. Load Balancing and Service Discovery:

- Docker Swarm automatically load balances traffic across containers running the same service.
- It also provides built-in service discovery, allowing containers to communicate with each other using service names as hostnames.

f. Rolling Updates:

- Docker Swarm supports rolling updates, which allow you to update services with zero downtime.
- You can define a strategy for rolling updates, specifying how many replicas to update at a time and the delay between updates.

g. Fault Tolerance:



- Docker Swarm provides fault tolerance by automatically rescheduling containers on other nodes if a node becomes unavailable.
- This ensures that your services remain available even in the event of node failures.

#### h. Integration with Docker Compose:

- Docker Swarm integrates with Docker Compose, allowing you to use the same Compose file syntax to define and deploy multi-container applications on a Swarm cluster.
- Docker Swarm provides a simple and powerful way to manage a cluster of Docker nodes and deploy containerized applications at scale.
- It simplifies the orchestration and management of containers, making it easier to build highly available and scalable distributed systems using Docker

#### Cluster Creation :

- To create a Docker Swarm cluster, you need to follow these steps:

a. Initialize the Swarm:

- On one of the machines that you want to include in the cluster, run the following command to initialize the Swarm and make it a manager node:

```
docker swarm init --advertise-addr  
<MANAGER_NODE_IP>
```

- Replace <MANAGER\_NODE\_IP> with the IP address of the machine. This command will generate a token that you will need to join worker nodes to the cluster.

b. Join Worker Nodes:

- On each machine that you want to add as a worker node, run the following command:

```
docker swarm join --token <TOKEN>  
<MANAGER_NODE_IP>:<PORT>
```

and <MANAGER\_NODE\_IP>:<PORT> with the IP address and port of the manager node.

c. Verify Cluster Status:

- Run the following command on the manager node to verify the status of the cluster and see the list of nodes:

```
docker node ls
```

- You should see the manager node and the joined worker nodes listed.

d. Deploy Services:

- Now that the cluster is set up, you can deploy services to it. Define your services in a Docker Compose file, specifying the desired configurations such as image, ports, volumes, etc. Then use the following command to deploy the services:

```
docker stack deploy -c <COMPOSE_FILE>  
<STACK_NAME>
```

- Replace <COMPOSE\_FILE> with the path to your Compose file and <STACK\_NAME> with a name for your stack.

e. Scale Services:

- You can scale your services by updating the desired number of replicas in your Compose file and running the deploy command again.
- Docker Swarm will automatically distribute the replicas across the cluster.

f. Manage the Cluster:

- You can manage the cluster by running commands on the manager node using 'docker service' or 'docker stack' commands.
- These commands allow you to inspect services, update configurations, scale services, perform rolling updates, etc.
- By following these steps, you can create a Docker Swarm cluster and start deploying and managing services across the cluster.
- Docker Swarm provides a simple and powerful way to orchestrate containerized applications, enabling you to scale and manage them efficiently.

Swarm Initialization running containers:

- To initialize a Docker Swarm and start running containers on the cluster, you need to follow these steps:

a. Initialize the Swarm:

- On the machine you want to set as the Swarm manager, run the following command to initialize the Swarm:

```
docker swarm init --advertise-addr <MANAGER_NODE_IP>
```

- Replace <MANAGER\_NODE\_IP> with the IP address of the machine. This command will generate a command with a token that you need to join worker nodes to the Swarm.

b. Join Worker Nodes:

- On each machine you want to add as a worker node, run the command provided by the docker swarm init command output.
- It will look something like this:

```
docker swarm join --token <TOKEN>  
<MANAGER_NODE_IP>:<PORT>
```

- Replace <TOKEN> with the token generated in step 1 and <MANAGER\_NODE\_IP>:<PORT> with the IP address and port of the manager node.

c. Verify Cluster Status:

- On the manager node, run the following command to verify the status of the Swarm cluster:

```
docker node ls
```

- You should see the manager node and the joined worker nodes listed.

d. Deploy Services:

- Create a Docker Compose file that defines the services you want to deploy on the Swarm cluster.
- Specify the desired configurations such as image, ports, volumes, etc.
- Then, use the following command to deploy the services:

```
docker stack deploy -c <COMPOSE_FILE> <STACK_NAME>
```

- Replace <COMPOSE\_FILE> with the path to your Compose file and <STACK\_NAME> with a name for your stack.

e. Scale Services:

- You can scale your services by updating the desired number of replicas in your Compose file and running the deploy command again.
- Docker Swarm will automatically distribute the replicas across the cluster.

f. Monitor the Cluster:

- You can monitor the cluster and the running containers by using commands such as `docker service ls`, `docker stack ps`, and `docker service logs`.
- These commands allow you to inspect services, view container status, logs, and perform other management tasks.

- By following these steps, you can initialize a Docker Swarm, join worker nodes to the cluster, and start running containers on the Swarm.
- Docker Swarm provides a robust and scalable platform for orchestrating containers, allowing you to efficiently manage your application deployments across a cluster of machines.

### Scaling Containers :

- Scaling containers refers to the process of increasing or decreasing the number of container instances running in a cluster or an orchestration platform.
- Docker Swarm provides built-in scaling capabilities that allow you to scale your services horizontally based on the demand and resource availability.

#### a. Replicas:

- In Docker Swarm, you define services instead of individual containers.
- A service represents a desired state for running containers. When you create a service, you specify the number of replicas you want to run.
- Replicas are the instances of a service running across different worker nodes in the Swarm cluster.



b. Scaling Up:

- To scale up the number of replicas for a service, you can use the 'docker service scale' command followed by the service name and the desired number of replicas.
- For example:

```
docker service scale myservice=5
```

- This command scales up the 'myservice' to 5 replicas.
- Docker Swarm will automatically distribute the replicas across available worker nodes.

c. Scaling Down:

- To scale down the number of replicas, you can use the same docker service scale command and specify the desired number of replicas.
- For example:

```
docker service scale myservice=2
```

- This command scales down the myservice to 2 replicas.
- Docker Swarm will stop and remove the excess replicas, ensuring that the desired state is maintained.

d. Dynamic Scaling:

- Docker Swarm supports dynamic scaling, which allows you to scale services automatically based on certain conditions.
- You can use external tools or monitoring systems to monitor resource utilization or application metrics and trigger scaling actions accordingly.
- These tools can interact with the Docker API to adjust the number of replicas dynamically.

e. Load Balancing:

- Docker Swarm provides built-in load balancing across replicas of a service.
- When you scale a service, Docker Swarm automatically distributes incoming traffic across all running replicas, ensuring that the load is evenly balanced.

f. Rolling Updates:

- When scaling a service, Docker Swarm supports rolling updates, which means that the replicas are updated incrementally without causing downtime.
- Docker Swarm will gradually update the replicas one by one, ensuring that the service remains available during the update process.
- By leveraging the scaling capabilities of Docker Swarm, you can easily adjust the number of container replicas for your services to meet the changing demands of your application.
- This flexibility allows you to scale your application horizontally and ensure high availability and performance.

#### Adding and Leaving Nodes :

- In a Docker Swarm cluster, you can add or remove nodes (hosts) to dynamically adjust the resources available for running containers.

#### A. Adding Nodes:

##### a. Join as a Worker:

- To add a node as a worker to an existing Docker Swarm cluster, you need to run the following command on the node you want to add:

```
docker swarm join --token <TOKEN>
<MANAGER_NODE_IP>:<PORT>
```

- Replace <TOKEN> with the token generated when initializing the Swarm cluster, and <MANAGER\_NODE\_IP>:<PORT> with the IP address and port of one of the manager nodes in the cluster.

b. Join as a Manager:

- If you want to add a node as a manager to the Swarm cluster, you can use the following command instead:

```
docker swarm join-token manager
```

- This command will generate a token that you can use on the node you want to add as a manager.
- Run the generated command on the node to join it as a manager.

c. Verification:

- To verify that the new node has joined the Swarm cluster successfully, run the following command on any manager node:

```
docker node ls
```

- This command will display the list of nodes in the cluster, including the newly added node.

#### Limitations of Docker Swarm:

##### a. Limited Scaling Capabilities:

- While Docker Swarm provides scaling capabilities, it may not be suitable for extremely large-scale deployments or highly dynamic workloads.
- In such cases, more advanced orchestration platforms like Kubernetes might be preferred.

##### b. Limited Feature Set:

- Compared to other container orchestration platforms like Kubernetes, Docker Swarm has a more limited set of features and functionalities.
- It may not offer the same level of flexibility and extensibility.

c. Lack of Advanced Scheduling Options:

- Docker Swarm's scheduling options are relatively basic compared to other platforms.
- It may not support more advanced scheduling strategies and policies required for complex deployment scenarios.

d. Limited Monitoring and Logging:

- Docker Swarm's built-in monitoring and logging capabilities are not as comprehensive as dedicated monitoring and logging solutions.
- Additional tools or integrations may be needed for advanced monitoring and logging requirements.

e. Limited Community Support:

- While Docker Swarm has a strong community, it may not have the same level of community support and ecosystem as other orchestration platforms like Kubernetes.

- Finding resources and expertise specific to Docker Swarm may be more challenging.
- It's important to consider these benefits, features, and limitations when evaluating Docker Swarm for container orchestration and deciding whether it fits your specific requirements and use cases.



## Practical :

### 1. Installation of Docker and Docker Compose on AWS EC2

- To install Docker and Docker Compose on an AWS EC2 instance, you can follow these steps:

#### a. Launch an EC2 Instance:

- Log in to your AWS Management Console.
- Navigate to EC2 and click on "Launch Instance".
- Choose an Amazon Machine Image (AMI) based on your requirements (e.g., Amazon Linux, Ubuntu, etc.).
- Select the desired instance type and configure the instance details.
- Configure security groups to allow inbound traffic on ports 22 (SSH) and 2376/2377 (Docker).
- Review the instance details and launch the EC2 instance.

#### b. Connect to the EC2 Instance:

- Use an SSH client (e.g., PuTTY, Terminal) to connect to your EC2 instance using the SSH key pair you specified during instance launch.
- Provide the appropriate permissions to the key pair using the command: `chmod 400 <path-to-key-pair.pem>`.



- Connect to the EC2 instance using the SSH command:  
'ssh -i <path-to-key-pair.pem>  
ec2-user@<public-ip-address>'.

c. Install Docker:

- Update the package index using the command: 'sudo yum update -y' (for Amazon Linux) or 'sudo apt update' (for Ubuntu).
- Install Docker using the package manager:
- For Amazon Linux:

```
sudo amazon-linux-extras install docker
sudo service docker start
sudo usermod -a -G docker ec2-user
```

- For Ubuntu:

```
sudo apt install docker.io
sudo systemctl start docker
sudo usermod -a -G docker ubuntu
```

d. Verify Docker Installation:

- Check the Docker version using the command: 'docker -version'.
- Test Docker by running a sample container: 'docker run hello-world'.

e. Install Docker Compose:

- Download the Docker Compose binary using the command:

```
sudo curl -L
"https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname -s)-$(uname -m)" -o
/usr/local/bin/docker-compose
```

- Provide executable permissions to the Docker Compose binary: 'sudo chmod +x /usr/local/bin/docker-compose'.
- Verify the Docker Compose installation: 'docker-compose -version'.
- Now you have successfully installed Docker and Docker Compose on your AWS EC2 instance. You can start using Docker and Docker Compose to manage and deploy your containerized applications.

## 2. Running Docker Commands

- To run Docker commands, follow these steps:
  - a. Open a terminal or command prompt on your machine.
  - b. Start by checking the Docker version to ensure it is installed and running properly:

```
docker version
```

- c. To view a list of available Docker commands, you can use the docker command with the '--help' flag:

```
docker --help
```

- d. Some commonly used Docker commands include:
  - docker run: Run a container from a Docker image.
  - docker pull: Download a Docker image from a registry.

- docker images: List the Docker images available on your machine.
- docker ps: List the running containers.
- docker stop: Stop a running container.
- docker start: Start a stopped container.
- docker rm: Remove a container.
- docker rmi: Remove an image.
- docker exec: Run a command inside a running container.
- docker logs: View the logs of a container.
- docker build: Build a Docker image from a Dockerfile.
- You can append ‘—help’ to any of these commands to get more information about their usage, options, and arguments. For example:

```
docker run --help
```

- Ensure that you have the necessary permissions to run Docker commands.
- On Linux, you may need to prefix Docker commands with sudo unless you have added your user to the Docker group:

```
sudo docker run hello-world
```

- With these basic commands, you can start working with Docker containers and images.
- Make sure to refer to the Docker documentation for more detailed information on each command and their options.



### 3. Writing Docker Files for various applications

- Writing Dockerfiles for various applications depends on the specific requirements and technologies used in those applications.
- Here are some examples of Dockerfiles for different types of applications

a. Dockerfile for a Node.js Application:

```
# Use the official Node.js base image
FROM node:14

# Set the working directory in the container
WORKDIR /app

# Copy package.json and package-lock.json to the container
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the application code to the container
COPY . .

# Expose a port for the application
EXPOSE 3000

# Start the application
```

b. Dockerfile for a Python Flask Application:

```
# Use the official Python base image
FROM python:3.9

# Set the working directory in the container
WORKDIR /app

# Copy the requirements.txt file to the container
COPY requirements.txt ./

# Install dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Copy the application code to the container
COPY ..

# Expose a port for the application
EXPOSE 5000
```

c. Dockerfile for a Java Spring Boot Application:

```
# Use the official OpenJDK base image
FROM openjdk:11

# Set the working directory in the container
WORKDIR /app

# Copy the JAR file to the container
COPY target/myapp.jar .

# Expose a port for the application
EXPOSE 8080

# Start the application
CMD [ "java", "-jar", "myapp.jar" ]
```

file



- These are just examples, and you may need to modify them based on your specific application requirements, such as additional dependencies, build steps, or environment variables.
- The Dockerfile provides instructions to build a Docker image, which can then be used to run containers for your applications.

#### 4. Building Docker Images

- To build Docker images, you need to create a Dockerfile that contains instructions on how to build the image.
- Here's a step-by-step guide on building Docker images:

##### a. Create a Dockerfile:

- Open a text editor and create a new file named "Dockerfile" (without any file extension).
- Add the necessary instructions to the Dockerfile to define the image's configuration and dependencies.

b. Specify the base image:

- Use the FROM instruction in the Dockerfile to specify the base image upon which your image will be built.
- For example, you can use the official Node.js base image: FROM node:14.

c. Set the working directory:

- Use the WORKDIR instruction to set the working directory within the container.
- This is where subsequent commands will be executed. For example, WORKDIR /app.

d. Copy application files:

- Use the COPY instruction to copy the necessary files and directories from your local machine to the image.
- For example, COPY package\*.json ./ to copy the package.json file.

e. Install dependencies:

- If your application has any dependencies, you need to install them within the image.
- Use the appropriate package manager command (e.g., RUN npm install) to install the dependencies.

f. Expose ports (if needed):

- If your application listens on a specific port, you can use the EXPOSE instruction to expose that port.
- For example, EXPOSE 3000 to expose port 3000.

g. Define the default command:

- Use the CMD or ENTRYPOINT instruction to specify the command that will be executed when a container is created from the image.
- This command can include running your application or any other desired behavior.

h. Build the Docker image:

- Open a terminal or command prompt and navigate to the directory containing the Dockerfile.
- Run the following command to build the Docker image:

```
docker build -t <image-name> .
```

- Replace <image-name> with the desired name for your image.
- i. Wait for the image to build:
  - Docker will execute each instruction in the Dockerfile and build the image.
  - The time it takes to build depends on the complexity of your application and the number of layers in the image.
- j. Verify the image:
  - Run the following command to list the Docker images on your machine:

```
docker images
```

- Verify that the image you built is listed with the specified image name.
- You have now successfully built a Docker image based on the instructions in your Dockerfile.

- You can use this image to create and run containers with your application.

## 5. Pushing Images to Docker Hub

- To push Docker images to Docker Hub, you need to follow these steps:
  - a. Tag the Image:
    - Before pushing the image, you need to tag it with your Docker Hub repository name.
    - Use the following command:

```
docker tag <image-name>  
<docker-hub-username>/<repository-name>:<tag>
```

- Replace <image-name> with the name of the image you want to push, <docker-hub-username> with your Docker Hub username, <repository-name> with the desired name of your repository on Docker Hub, and <tag> with an optional tag for versioning.

- b. Log in to Docker Hub:

- Log in to your Docker Hub account using the following command:

```
docker login
```

- You will be prompted to enter your Docker Hub username and password.

c. Push the Image:

- Use the following command to push the tagged image to Docker Hub:

```
docker push  
<docker-hub-username>/<repository-name>:<tag>  
>
```

- Replace <docker-hub-username>, <repository-name>, and <tag> with the values you used in the previous step.

d. Wait for the Image to Push:

- Docker will upload the image and its layers to Docker Hub.
- The time it takes depends on the size of the image and your internet connection speed.

e. Verify the Image on Docker Hub:

- Open your web browser and navigate to Docker Hub (hub.docker.com).
- Log in to your Docker Hub account if you haven't already.
- Search for your repository by name or go to your profile page to see a list of repositories.
- Verify that the image you pushed is listed in the repository with the correct tag.
- You have successfully pushed your Docker image to Docker Hub.
- It is now available for others to use and pull from Docker Hub.

## 6. Running Docker Containers

- To run Docker containers, follow these steps:
  - a. Pull the Docker Image (if not already pulled):
- Use the docker pull command to download the desired Docker image from a registry. For example:

```
docker pull <image-name>:<tag>
```

- Replace <image-name> with the name of the image and <tag> with the specific version or tag you want to pull. If you don't specify a tag, it will default to the "latest" tag.

b. Run a Docker Container:

- Use the docker run command to create and start a new container from the pulled image. For example:

```
docker run <image-name>:<tag>
```

- Replace <image-name> with the name of the image and <tag> with the specific version or tag you want to run.

c. Specify Container Options:

- You can include additional options and configurations when running the container.
- Some commonly used options include:



- -d: Run the container in detached mode (in the background).
- -p: Publish container ports to the host machine. For example, -p 8080:80 maps port 80 inside the container to port 8080 on the host.
- --name: Assign a custom name to the container.
- -e: Set environment variables inside the container.
- -v: Mount volumes from the host to the container.
- --restart: Set the restart policy for the container.
- -it: Run the container interactively with a terminal session.

#### d. Access Container Output:

- By default, the container's output is printed to the terminal.
- You can view the logs of a running container using the docker logs command followed by the container ID or name.

#### e. Manage Running Containers:

- Use the docker ps command to list the running containers.
- To stop a running container, use the docker stop command followed by the container ID or name.

- To start a stopped container, use the docker start command followed by the container ID or name.
- To remove a container, use the docker rm command followed by the container ID or name.
- Remember to refer to the Docker documentation for a comprehensive list of options and commands available for managing and interacting with Docker containers.

## 7. Container Port Binding

- Container port binding allows you to map a port inside a container to a port on the host machine, enabling communication between the container and the host or other containers.
- Here's how you can bind container ports:

### a. Run a Container with Port Binding:

- When using the docker run command, specify the -p option followed by the host port and container port separated by a colon.
- For example, to bind port 8080 on the host to port 80 inside the container:

```
docker run -p 8080:80 <image-name>
```

- Replace <image-name> with the name of the Docker image you want to run.

b. Access the Containerized Application:

- After running the container with port binding, you can access the containerized application by using the host machine's IP address or localhost followed by the mapped port.
- In the example above, you would access the application in the container by navigating to `http://localhost:8080` or `http://<host-ip>:8080` in your web browser.

c. Binding Multiple Ports:

- You can bind multiple ports by specifying additional `-p` options.
- For example, to bind ports 8080 and 8081 on the host to ports 80 and 443 inside the container:

```
docker run -p 8080:80 -p 8081:443 <image-name>
```

#### d. Binding to Specific IP Addresses:

- By default, port binding binds to all available network interfaces on the host.
- However, you can bind to specific IP addresses by specifying the IP address before the host port.
- For example, to bind port 8080 on the IP address 192.168.0.100 to port 80 inside the container:

```
docker run -p 192.168.0.100:8080:80 <image-name>
```

#### e. Binding UDP Ports:

- By default, port binding uses TCP as the protocol. To bind a UDP port instead, specify udp after the port number.
- For example, to bind UDP port 53 on the host to UDP port 53 inside the container:

```
docker run -p 53:53/udp <image-name>
```

- Port binding is a crucial feature of Docker that allows containers to expose their services to the outside world.
- It facilitates easy communication between containers and the host machine or other containers running on the same network.

```
docker-compose up
```

- Docker Compose will read the configuration from the Docker Compose file and start the specified containers.
  - Container logs will be displayed in the terminal.
- a. Stop and remove the containers:
- To stop the running containers, press Ctrl + C in the terminal.
  - To remove the containers and associated resources, run the following command in the same directory as the Docker Compose file:

```
docker-compose down
```

- By using a Docker Compose file, you can define and manage multiple containers as a single unit.
- Docker Compose simplifies the process of running and managing multiple containers, enabling you to define complex multi-container environments with ease.

## 8. Persisting container data using Docker Volume

- To persist data in Docker containers using volumes, follow these steps:
  - a. Create a Docker Volume:
    - Use the docker volume create command to create a named volume. For example:

```
docker volume create mydata
```

### b. Mount the Volume in a Container:

- When running a container,
- use the '-v' or '--volume' option to specify the volume to mount inside the container. For example:

```
docker run -v mydata:/app/data <image-name>
```

- Replace <image-name> with the name of the Docker image you want to run.
- c. Access the Persisted Data:
- Any data written to the mounted volume inside the container will be persisted even after the container is stopped or removed.
  - To access the persisted data, start a new container with the same volume mounted at the same location. The data will be available in the container.

d. List Docker Volumes:

- To view the list of Docker volumes on your system, use the docker volume ls command.

e. Remove Docker Volumes:

- To remove a Docker volume, use the `docker volume rm` command followed by the volume name. For example:

```
docker volume rm mydata
```

- By using Docker volumes, you can persist data even when containers are stopped, started, or removed.
- Volumes provide a convenient way to manage and share data between containers or between containers and the host system.

## 9. Initialize a docker swarm and demonstrate workload deployments

- To initialize a Docker Swarm and demonstrate workload deployments, follow these steps:
  - a. Initialize Docker Swarm:
    - Open a terminal or command prompt and run the following command to initialize Docker Swarm:



```
docker swarm init
```

b. Create a Docker Swarm Service:

- In the same terminal, run the following command to create a Docker Swarm service:

```
docker service create --name myservice --replicas 3  
    <image-name>
```

- Replace <image-name> with the name of the Docker image you want to deploy as a service. This command creates a service named "myservice" with three replicas.

c. List Docker Services:

- To view the list of Docker services running in the Swarm, use the following command:

```
docker service ls
```

d. Scale the Service:

- You can scale the number of replicas of a service using the following command:

```
docker service scale myservice=5
```

- This command scales the "myservice" service to have five replicas.

e. Update the Service:

To update the service with a new image or configuration, use the following command:

```
docker service update --image <new-image-name>  
myservice
```

- Replace <new-image-name> with the name of the new Docker image you want to use for the service.

f. Remove the Service:

- To remove a service, use the following command:

```
docker service rm myservice
```

- These steps demonstrate the basic workflow of deploying workloads using Docker Swarm.
- Docker Swarm enables you to manage and orchestrate containerized applications across a cluster of Docker hosts, providing scalability, high availability, and fault tolerance.

