1 • Write a brief explanation of what Docker volumes are and why they are used in containerized environments. State different types of volumes in Docker and also make a note on difference between them.

->

Docker volumes are a persistent storage mechanism designed to store data generated by and used by Docker containers. They provide a way to keep the data integrated with the container lifecycle, ensuring that data remains intact even when containers are deleted or recreated.
They can be shared, backed up, and restored, ensuring that your containers are fully equipped to deliver an unforgettable experience.

A volume's lifecycle

A volume's contents exist outside the lifecycle of a given container. When a container is destroyed, the writable layer is destroyed with it. Using a volume ensures that the data is persisted even if the container using it is removed.
A given volume can be mounted into multiple containers simultaneously. When no running container is using a volume, the volume is still available to Docker and isn't removed automatically. You can remove unused volumes using

- docker volume prune.
- Named volumes are **persistent, reusable**, and **managed by Docker**

-> they used in containerized environment because -
-Volumes can be shared across multiple containers.
-deal for database files, uploaded content, config files, etc
-You can manage volumes using Docker CLI commands or the Docker API.
-volumes works on both windows and Linux containers.
-When your application requires high-performance I/O.

-> Types of Docker volumes
There are three main types of Docker volumes:

1. Bind mounts,
2. Named volumes
3. Anonymous volumes.

* Bind mounts :-

Bind mounts map a directory or file on the host system to a directory or file within the container.
This volume type is useful for
- transferring configuration files or source code between the host and container.
-When you want to create or generate files in a container and persist the files onto the host's filesystem.
-Docker bind mounts are recursive by default.
-Containers with bind mounts are strongly tied to the host.

Syntax :
--- >docker run --mount type=bind,src=<host-path>,dst=<container-path>
---> docker run --volume <host-path>:<container-path>

*Named volumes :-

Docker creates and manages named volumes, which provide a convenient way to persist data across container restarts and removals. Named volumes are more portable than bind mounts and can be easily shared between containers.
-to create a volume -- > docker volume create my-volume

to run with container - ->docker run -v my-volume:/app/data myimage

-Ideal for production, databases, and app data
-Named volumes are persistent, reusable, and managed by Docker.

*Anonymous volumes:-

Anonymous volumes are similar to named volumes in that they are not given a name. When a container is started without specifying a volume name or bind mount, they are created automatically.

- These volumes are beneficial when data does not need to be retained long-term or accessed by multiple contaners.
- Use them when you prioritize an easy setup and teardown process over data persistence or when the data's integrity beyond the container's execution isn't a concern.
- This makes them ideal for tasks like temporary data processing, testing, or development phases where data longevity isn't required.

To run a container with anonymous volume --> docker run -v /data ubunt

Difference between docker volumes and Bind mount

--->

By default, a local named volume is exactly as you describe, a bind mount to a special docker directory. The differences I see:

Behavior difference :- Named volumes are easier to work with because Docker sets them up automatically with the right files, owners, and permissions based on the image.
With bind mounts , you may run into file permission issues because files on your host system might not match what the container expects.

Portability:

Named volumes are more portable and flexible.You don't need to worry about file paths or users on your system — it works the same on any machine (Windows, Mac, Linux).
You just give the volume a name, and Docker takes care of the rest.

Management:

Bind mounts are managed on the host machine, which can lead to permission issues (user IDs inside container might not match the host).
With named volumes, Docker manages it for you — so you can easily control what goes in the volume, from inside another container if needed.

2 • creating a volume and attaching to container .

->

```
# Add current user to the Docker group (so you can run docker without sudo)
sudo usermod -aG docker $USER

# Refresh group membership in the current session
newgrp docker

# Check if Docker is working and running containers (optional check)
docker ps

#  Step 1: Create a named Docker volume called "mydata"
docker volume create mydata

# List all Docker volumes to confirm "mydata" was created
docker volume ls

# Inspect the volume to find its mount point
docker volume inspect mydata

#  Step 2: Run an NGINX container with "mydata" mounted to NGINX's default web root
docker run -d --name nginxcont -v mydata:/usr/share/nginx/html nginx
```

Check running containers
docker ps

Step 3: Create an HTML file on the host machine
nano index.html # Add: <h1>Hello, Docker Volumes!</h1>

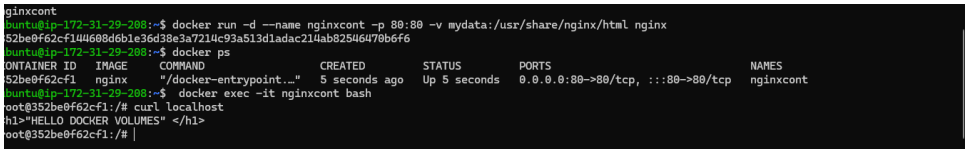
Confirm the file's content
cat index.html

# Step 4: Copy the HTML file into the Docker volume's _data directory
sudo cp index.html /var/lib/docker/volumes/mydata/_data/

# Verify the file is inside the volume
sudo ls /var/lib/docker/volumes/mydata/_data/

Step 5: Open a shell inside the running NGINX container
docker exec -it nginxcont bash

Inside the container, verify the HTML file exists and is readable
cat /usr/share/nginx/html/index.html



--> add port 80 in security group of instance and using public ip of instance access the index.html of container

3 • Brief explaination of docker networks and difference between host and bridge network
-->
Docker has a built-in networking system that manages communication between containers, the Docker host, and external networks. It supports different types of networks to handle a range of use cases, ensuring secure and flexible communication within containerized environments.
- Containers must be connected to a Docker network to receive network connectivity. The communication routes available to the container depend on its network connections]

-Docker supports six network types to manage container communication that implement core networking functionality but we mainly use three of them ie, Host,bridge,none:

1. bridge
2. Host
3. None
4. overlay
5. IPvLAN
6. macvlan

1. bridge

-Bridge network is a default network, Bridge networks in Docker create a virtual connection between the host system and the containers.

-This type of network is commonly used for container communication on a single host.
-. Each container gets its IP address within the bridge network.
-Containers in this network can communicate with each other but are isolated from external networks unless specifically
configured.
-Ideal for simple apps, microservices on the same Docker host.
-Containers can ping and connect to each other.
to create a custom bridge network --> docker network create --driver bridge mynetwork
to run continers using this bridge network
--> docker run -d --name nginxcont --network mynetwork nginx
--> docker run -it --name testcont --network mynetwork alpine sh

2. host
-Host network removes the network isolation between the container and the Docker host.
-The container uses the host's networking directly.
-Useful for performance-intensive applications that need low latency.
-Ports don't need to be published using -p since the container shares the host's ports.
-Works only on Linux hosts.
to run a container using host network
docker run --rm --network host nginx

3. none
-This network disables all networking for the container.
-The container has no network access (not even to the internet or other containers).
-Useful for security or when networking is not required.
-Containers still run and do other tasks like computation, file handling, etc.
to run a container using none network
docker run -it --rm --network none alpine sh

4. overlay
-Overlay networks allow containers on different Docker hosts to communicate securely.
-Used in Docker Swarm or multi-host networking.
-Provides encryption and routing between containers across hosts.
-Ideal for distributed systems and services running in production.
to create an overlay network (requires Swarm mode)
docker network create --driver overlay myoverlay
to use in Swarm services
docker service create --name web --network myoverlay nginx

5. macvlan
- macvlan network assigns a MAC address to the container, making it appear as a physical device on the network.

- The container gets an IP from the same network as the host.
- Useful when containers need to be accessed as standalone devices on the LAN (e.g., for legacy systems or network-based licensing).
- More complex to configure and usually used in advanced networking setups.

```
# to create a macvlan network
docker network create -d macvlan \
  --subnet=192.168.1.0/24 \
  --gateway=192.168.1.1 \
  -o parent=eth0 mymacvlan
# to run a container using macvlan
docker run --rm --network mymacvlan alpine ip a
```

* Host vs Bridge network *

Security:-

- Bridge Network:
  More secure — containers are isolated from the host and other networks unless explicitly connected.
- Host Network:
  Less secure — containers share the host's network directly, increasing the risk of exposure.

Use Case:-

- Bridge Network:
  Ideal for microservices, development, and apps needing container-to-container communication with isolation.
- Host Network:
  Suitable for performance-critical apps like monitoring agents or proxies that require host-level access.

Convenience:-

- Bridge Network:
  Requires port mapping (-p) to expose services, giving more control but needing extra configuration.
- Host Network:
  No need for port mapping — services are directly accessible, making setup faster but less controlled.

Network Isolation:-

- Bridge Network:
  Containers run in a private subnet with controlled communication.
- Host Network:
  No isolation — containers and host share the same network namespace.

Accessibility:-

- Bridge Network:
  Not directly accessible from outside unless ports are exposed.
- Host Network:
  Services are accessible via host IP and ports instantly.

Platform Support:-

- Bridge Network:
  Supported on all platforms (Linux, Windows, macOS).
- Host Network:
  Only available on Linux.

Container Port Binding:-

- Bridge Network:
  Needs manual -p flags for each port to be reachable.
- Host Network:
  Automatically uses the host ports — simpler but risky.

Performance:-

- Bridge Network:
  Slightly lower performance due to virtual networking overhead.

- Host Network:
  Higher performance — no virtualization between host and container networking.

4 • use of custom network creating a network and two containers in it and performing curl command on them to check connectivity .

- docker network create mynetwork #creating a network
- docker run -d --name httpdcont httpd # httpd container
- docker run -d --name nginxcont nginx # nginx container
- docker network connect mynetwork nginxcont # connecting to mynetwork
- docker network connet mynetwork httpdcont
- docker exec -it httpdcont bash # going inside httpd container
- sudo apt update # updating packages
- sudo apt install curl -y # installing curl
- curl 172.18.0.3 # curl command using ip of nginx container to check response.
- <html><body><h1>It works!</h1></body></html> # response from nginx container

```
  180  history
ubuntu@ip-172-31-29-208:~$ docker network ls
NETWORK ID     NAME        DRIVER    SCOPE
402f88ca6d06   bridge      bridge    local
15842bae243e   host        host      local
a452ae40b2af   mynetwork   bridge    local
2730ca08a362   none        null      local
ubuntu@ip-172-31-29-208:~$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED       STATUS       PORTS                                   NAMES
4380859aeb20   httpd     "httpd-foreground"       5 hours ago   Up 5 hours   80/tcp                                  httpdcont
ce8fb57817c4   nginx     "/docker-entrypoint.…"   6 hours ago   Up 6 hours   0.0.0.0:80->80/tcp, :::80->80/tcp   nginxcont
ubuntu@ip-172-31-29-208:~$ docker exec -it nginxcont bash
root@ce8fb57817c4:/# curl 172.18.0.3
<html><body><h1>It works!</h1></body></html>
root@ce8fb57817c4:/# exit
exit
```

5• Write a note on Dockerfile with usage of its attributes.

-> ,A Dockerfile is a plain text file that contains a set of instructions Docker uses to build a custom image. It automates the process of setting up the environment and installing software, helping developers package their applications along with dependencies in a reproducible way.

Why Use a Dockerfile:-
-Automates image creation
-Ensures consistency across development and production
-Makes deployment faster and repeatable
- Simplifies application sharing and versioning

* Dockerfile attributes with their usage

1. FROM

A FROM statement defines which image to download and start from. It must be the first command in your Dockerfile. A Dockerfile can have multiple FROM statements which means the Dockerfile produces more than one image

```
FROM java: 8
```

2. MAINTAINER

This statement is a kind of documentation, which defines the author who is creating this Dockerfile or who should you contact if it has bugs.

Example:

```
MAINTAINER Firstname Lastname <example@geeksforgeeks.com>
```

3. RUN

The RUN statement defines running a command through the shell, waiting for it to finish, and saving the result. It tells what process will be running inside the container at the run time.

Example:

```
RUN unzip install.zip /opt/install
RUN echo hello
```

4. ADD

If we define to add some files, ADD statement is used. It basically gives instructions to copy new files, directories, or remote file URLs and then adds them to the filesystem of the image.

To sum up it can add local files, contents of tar archives as well as URLs.

Example:

```
Local Files: ADD run.sh /run.sh
Tar Archives: ADD project.tar.gz /install/
URLs: ADD   https://project.example.com/image
```

5. ENV

ENV statement sets the environment variables both during the build and when running the result. It can be used in the Dockerfile and any scripts it calls. It can be used in the Dockerfile as well as any scripts that the Dockerfile calls. These are also persistent with the container and can be referred to at any moment.

Example:

```
ENV URL_POST=production.example-gfg.com
```

6. ENTRYPOINT

It specifies the starting of the expression to use when starting your container. Simply ENTRYPOINT specifies the start of the command to run. If your container acts as a command-line program, you can use ENTRYPOINT.

Example:

```
ENTRYPOINT ["/start.sh"]
```

7. CMD

CMD specifies the whole command to run. We can say CMD is the default argument passed into the ENTRYPOINT. The main purpose of the CMD command is to launch the software required in a container.

Example:

```
CMD ["program-foreground"]
CMD ["executable", "program1", "program2"]
```

Note: If you have both ENVIRONMENT and CMD, they are combined together.

8. EXPOSE

EXPOSE statement maps a port into the container. The ports can be TCP or UDP but by default, it is TCP.

Example:

```
EXPOSE 8080
```

9. VOLUME

The VOLUME statement defines shared volumes or ephemeral volumes depending upon whether you have one or two arguments.

Example:

```
1. If you have two arguments, it maps a host path into a container path.
       VOLUME ["/host/path" "/container/path/"]
```

```
2. If you have one arguments, it creates a volume that can be inherited by
the later containers.
          VOLUME ["/shared-data"]
```

10. WORKDIR

As the name suggests, WORKDIR sets the directory that the container starts in. Its main purpose is to set the working directory for all future Dockerfile commands.

Example:

```
WORKDIR /directory-name
```

6 • What is difference between CMD and ENTRYPOINT?

-->

CMD vs ENTRYPOINT in Dockerfile

Both CMD and ENTRYPOINT define what happens when your container starts, but they work a bit differently.

*** CMD**

- Provides default arguments or commands.
- Can be easily overridden when running the container.

Example:

CMD ["echo", "Hello World"]
Run it:
docker run myimage
Output: Hello World
docker run myimage Goodbye
Output: Goodbye
Great when you want flexible behavior.

*** ENTRYPOINT :-**

- Defines the main executable that always runs.
- Can only be changed using --entrypoint.

Example:

ENTRYPOINT ["ping"]
Run it:
docker run myimage google.com
Output: pinging google.com
docker run myimage echo Hello
Won't work. To override:
docker run --entrypoint echo myimage Hello

Best for fixed behavior, like a script or service.

*** Using Both Together**
You can combine them:
ENTRYPOINT ["ping"]
CMD ["google.com"]
Now:
docker run myimage
pings google.com (default)
docker run myimage yahoo.com
pings yahoo.com (overrides CMD)

Tip:

- Use ENTRYPOINT when your container acts like a single-purpose tool.
- Use CMD when you want to allow users to customize what runs.

7 • What is difference between ADD and COPY?

--> * COPY vs ADD in Dockerfile

Both COPY and ADD are used in Dockerfiles to copy files into the image, but they're not exactly the same.

COPY

- Only used to copy files or directories from your local context (host) into the container.
- It is simple, predictable, and preferred when you just want to move files.

Example:
COPY index.html /usr/share/nginx/html/
Copies index.html from your local folder into the container at /usr/share/nginx/html/.

ADD

- Works like COPY, but with extra features:
  1. It can extract local .tar, .tar.gz files automatically.
  2. It can download files from a URL.

Example 1: Extract tar archive
ADD site.tar.gz /usr/share/nginx/html/

Overwriting Files

- If you use multiple COPY or ADD commands on the same file path, Docker will overwrite it with the last one in the Dockerfile.

COPY index.html /usr/share/nginx/html/
ADD index.html /usr/share/nginx/html/
The second command (ADD) will **overwrite** the file from the first.

***** when to use :-

- Use COPY when you just want to move files — it's cleaner and simpler.
- Use ADD only when you need to extract a tar file or download from a URL.

8•Write a Dockerfile to run Nodejs application build an image from it and create a container using that image (also include persistent volume and network in Dockerfile).

-->
Building a project directory:-
-> mkdir docker-node-app

- Dockerfile

```
# Base image
FROM node:16-alpine

# Set working directory
WORKDIR /app

# Copy package.json and install dependencies
COPY package*.json ./
RUN npm install --production

# Copy application code
COPY . .

# Declare a volume mount point
VOLUME [ "/app/data" ]

# Expose the app port
EXPOSE 3000

# Start the app
CMD ["node", "app.js"]
```

- package.json

```
{
 "name": "docker-node-app",
 "version": "1.0.0",
 "main": "app.js",
 "dependencies": {
  "express": "^4.17.1"
 }
}
```

- app.js

```
const express = require('express');
const app = express();

// Route for the homepage
app.get('/', (req, res) => {
  res.send(`
    <!DOCTYPE html>
    <html lang="en">
    <head>
      <meta charset="UTF-8" />
      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
      <title>Welcome to Docker + Express</title>
      <style>
        body {
          background: linear-gradient(to right, #4facfe, #00f2fe);
          font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
          color: white;
          text-align: center;
          padding: 80px;
        }
        h1 {
```

```
      font-size: 3rem;
      margin-bottom: 10px;
    }
    p {
      font-size: 1.5rem;
    }
    .box {
      background-color: rgba(255, 255, 255, 0.1);
      border-radius: 10px;
      padding: 30px;
      display: inline-block;
      box-shadow: 0 8px 20px rgba(0, 0, 0, 0.2);
    }
  </style>
 </head>
 <body>
  <div class="box">
   <h1>🚀 Hello from Express inside Docker!</h1>
   <p>Your app is running beautifully on <strong>Node.js & Docker</strong>.</p>
  </div>
 </body>
 </html>
 `);
});

// Start the server
app.listen(3000, () => {
 console.log('🌐 Server is running on http://localhost:3000');
});
```

- Building image from Dockerfile:-

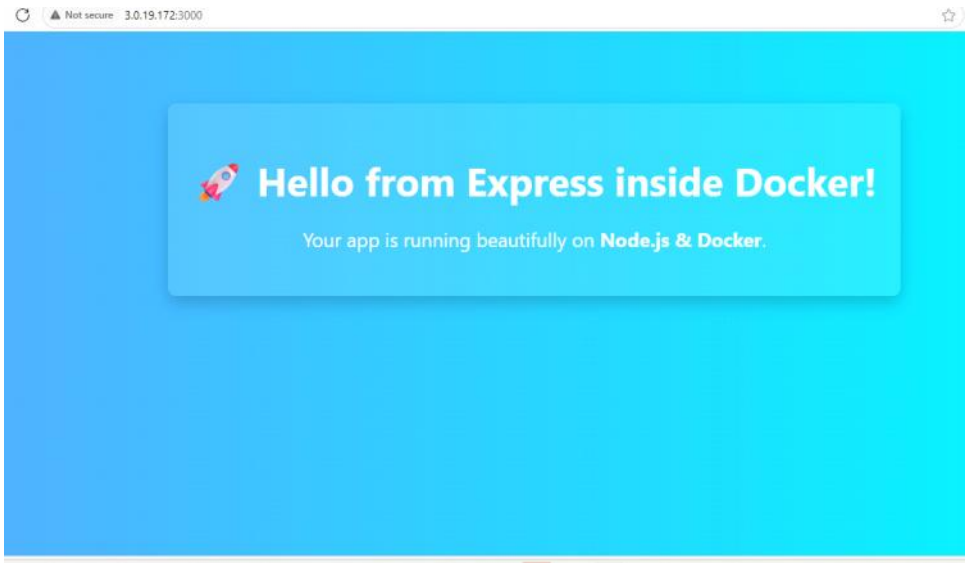
  -> docker build -t my-node-app .

- Running a nodejs container using image

- docker run -d --name node-app --network mynetwork -v mydata:/app/data -p 3000:3000 my-node-app

- add port 3000 to security group of server
- To access the node app on internet.

  * <public ip of server: 3000>



9 • Write a Dockerfile to create a python application build image from it and push that image to private
repository of Docker hub.
-->

- Dockerfile for python app

```
FROM python:3.9-slim-buster

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 5000

CMD ["python", "app.py"]
```

- To tag the image
  docker tag my-python-app <repo name>/my-python-app:latest
- To push the image to dockerhub
  docker push <your repo name>/my-python-app:latest