1. What is Docker?

Docker is an open-source platform that allows developers to automate the deployment of applications inside lightweight, portable containers. Containers package an application and its dependencies together, ensuring consistency across development, testing, and production environments.

---

2. Key Features of Docker

- **Lightweight Containers**: Containers share the host OS kernel, making them more efficient than traditional virtual machines.
- **Portability**: "Write once, run anywhere" – containers can run on any system that supports Docker.
- **Isolation**: Applications in containers are isolated from each other, enhancing security and stability.
- **Scalability**: Easily scale up or down by creating or destroying containers.
- **Versioning**: Docker images can be versioned for better control and rollback capabilities.
- **Efficiency**: Use system resources efficiently, leading to faster application performance.

---

3. Core Docker Components

- **Docker Engine**: The runtime environment to build, run, and manage containers.
  - **Docker Daemon**: The background service managing containers.
  - **Docker CLI**: Command-line interface to interact with the daemon.
- **Images**: Pre-built templates for creating containers (e.g., an image for Ubuntu with Node.js pre-installed).
- **Containers**: Running instances of images that are isolated and lightweight.
- **Dockerfile**: A script used to create Docker images.
- **Volumes**: Persistent storage for containers.
- **Networks**: Allow containers to communicate with each other or external systems.

---

4. Docker Workflow

1. **Write a `Dockerfile`**: Define the environment, dependencies, and instructions.
2. **Build an Image**: Use the `docker build` command.
3. **Run a Container**: Start a container using the `docker run` command.
4. **Manage Containers**: Use commands like `docker ps`, `docker stop`, and `docker rm`.
5. **Push to a Registry**: Store images in Docker Hub, AWS ECR, or a private registry.

---

5. Common Docker Commands

- **Images**
  - `docker images`: List all available images.
  - `docker build -t image_name .`: Build an image from a Dockerfile.
  - `docker pull image_name`: Download an image from a registry.
- **Containers**
  - `docker run image_name`: Run a container from an image.
  - `docker ps`: List running containers.
  - `docker stop container_id`: Stop a container.
  - `docker rm container_id`: Remove a stopped container.
  - `docker exec -it container_id command`: Execute a command inside a running container.
- **Volumes**
  - `docker volume create volume_name`: Create a volume.
  - `docker volume ls`: List all volumes.
- **Networking**
  - `docker network create network_name`: Create a custom network.
  - `docker network ls`: List all networks.

---

6. Dockerfile Example

```
dockerfile
# Use a base image
FROM node:20

# Set the working directory
WORKDIR /app

# Copy application code
COPY . .

# Install dependencies
RUN npm install

# Expose application port
EXPOSE 3000

# Start the application
CMD ["npm", "start"]
```

---

7. Docker Compose

Docker Compose is a tool to define and manage multi-container Docker applications using a `docker-compose.yml` file.

Example `docker-compose.yml`:

```
version: '3.8'
```

```
services:
  app:
    build:
      context: .
    ports:
      - "3000:3000"
    volumes:
      - .:/app
    depends_on:
      - db

  db:
    image: postgres:14
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: app_db
```

---

8. Docker Registry

Docker images are stored in registries:

- **Docker Hub**: Public registry.
- **Amazon ECR**: AWS-hosted private registry.
- **Self-hosted**: Using tools like Harbor.

Commands for working with registries:

- `docker login`: Authenticate to a registry.
- `docker tag image_name repository/image_name:tag`: Tag an image.
- `docker push repository/image_name:tag`: Push to a registry.

---

9. Advanced Docker Concepts

- **Orchestration**: Tools like Kubernetes and Docker Swarm manage multiple containers across a cluster.
- **Multi-stage Builds**: Optimize image sizes by separating build and runtime stages in the Dockerfile.
- **Security**: Use tools like Docker Bench to assess security best practices.
- **Performance Optimization**: Use caching layers and smaller base images.

---

10. Use Cases

- **Microservices**: Deploy individual services in separate containers.
- **CI/CD Pipelines**: Automate builds and tests in isolated environments.
- **Dev/Test Environments**: Quickly spin up reproducible environments.

- **Hybrid Cloud Deployments**: Ensure consistency across cloud and on-premises systems.

---

1. Layers in Docker Containers

- **Base Image Layer**:
  - The foundational layer of the Docker image. For example, a base image could be `ubuntu`, `node`, or `alpine`.
  - Provides the operating system environment for the container.
- **Intermediate Layers**:
  - These are the layers created by each instruction in the Dockerfile (e.g., `RUN`, `COPY`, `ADD`).
  - Each instruction in the Dockerfile creates a new layer.
- **Application Layer**:
  - Contains the application code and its dependencies added using commands like `COPY` or `RUN npm install`.
- **Writable Container Layer**:
  - When a container is started from an image, a writable layer is added on top.
  - This writable layer stores changes made to the container during runtime (e.g., adding files, modifying configuration).

---

2. Layer Characteristics

- **Read-Only Layers**:
  - All layers except the writable container layer are read-only.
  - These layers are shared across containers that use the same image, saving storage and improving efficiency.
- **Union File System**:
  - Docker uses a UnionFS (e.g., OverlayFS) to stack layers, presenting them as a single unified file system.
- **Layer Caching**:
  - Docker caches each layer during the build process.
  - If a layer hasn't changed, Docker reuses it, significantly speeding up builds.

---

3. Example of Layers in a Dockerfile

```
dockerfile
# Base image (Layer 1)
FROM node:20

# Working directory (Layer 2)
WORKDIR /app

# Copy application files (Layer 3)
COPY . .
```

```
# Install dependencies (Layer 4)
RUN npm install

# Expose the application port (Layer 5)
EXPOSE 3000

# Run the application (Layer 6)
CMD ["npm", "start"]
```

Explanation:

1. **Layer 1**: The `node:20` base image.
2. **Layer 2**: The `WORKDIR` instruction sets the working directory.
3. **Layer 3**: The `COPY` instruction adds application code.
4. **Layer 4**: The `RUN npm install` instruction installs dependencies.
5. **Layer 5**: The `EXPOSE` instruction opens the port for the application.
6. **Layer 6**: The `CMD` instruction defines the runtime command for the container.

---

4. Types of Docker Layers

- **Image Layers**: Represent static parts of the image (e.g., base OS, application files).
- **Container Layer**: The dynamic writable layer unique to each running container.

---

5. Benefits of Layers

- **Reusability**: Layers can be reused across multiple images, reducing redundancy.
- **Efficiency**: Changes in one layer don't require rebuilding the entire image.
- **Portability**: Layers can be shared across systems using registries like Docker Hub.

---

6. How Layers Work During Build and Runtime

During Build:

- Docker executes each instruction in the Dockerfile sequentially.
- Each instruction creates a new layer.
- Unchanged layers are cached for faster builds.

During Runtime:

- The writable container layer is created on top of the image layers.
- Changes made during container runtime (e.g., new files, modifications) are stored in this layer.

---

7. Managing Layers

- **View Layers**: Use `docker history image_name` to view the layers of an image.
- **Reduce Layers**:
  - Combine commands into a single `RUN` statement:

    ```
    dockerfile
    RUN apt-get update && apt-get install -y curl && apt-get clean
    ```

  - Minimize the use of `COPY` or `ADD` commands.

Managing Docker Space

Over time, Docker can consume significant disk space due to unused containers, images, volumes, and networks. Here are some strategies to manage Docker disk usage:

---

1. Viewing Disk Usage

To identify what is consuming space:

```
docker system df
```

This command shows:

- Images: Number of images and their sizes.
- Containers: Running and stopped containers.
- Volumes: Disk space used by volumes.

---

2. Removing Unused Docker Resources

a. Remove Unused Images

Unused images, also called **dangling images**, are those not associated with a container.

- To remove dangling images:

  ```
  docker image prune
  ```

- To remove all unused images:

  ```
  docker image prune -a
  ```

b. Remove Stopped Containers

- To remove stopped containers:

```
docker container prune
```

c. Remove Unused Volumes

- To remove unused volumes:

  ```
  docker volume prune
  ```

d. Remove Unused Networks

- To remove unused networks:

  ```
  docker network prune
  ```

e. Full Cleanup

- To remove all unused data (images, containers, volumes, networks):

  ```
  docker system prune -a --volumes
  ```

---

3. Additional Space Management Tips

- **Set a Retention Policy for Images**:
  - Regularly clean up old images and keep only those actively in use.
- **Use Lightweight Base Images**:
  - Use images like `alpine` or `debian-slim` to reduce image size.
- **Multi-Stage Builds**:
  - Optimize the Dockerfile by separating build and runtime stages.
- **Use Docker Disk Space Monitoring Tools**:
  - Tools like `dockviz` or third-party monitoring platforms.

What is Docker Compose?

Docker Compose is a tool for defining and running multi-container Docker applications. It uses a **YAML file** (`docker-compose.yml`) to configure the application's services, networks, and volumes.

Key Features

- Orchestrates multiple containers as part of a single application.
- Defines services, networks, and volumes in one file.
- Simplifies application deployment in development and production.

How Docker Compose Works

1. Define the application services in a `docker-compose.yml` file.
2. Run the services with a single command.

Example `docker-compose.yml` File

Here's an example of a multi-container application with a Node.js app and a PostgreSQL database:

```
yaml
version: '3.8'

services:
  app:
    image: node:20
    working_dir: /app
    volumes:
      - .:/app
    ports:
      - "3000:3000"
    command: npm start
    depends_on:
      - db

  db:
    image: postgres:14
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: my_database
    ports:
      - "5432:5432"
```

Commands to Use Docker Compose

- **Start Services**:

  ```
  docker-compose up
  ```

  Add `-d` to run in detached mode:

  ```
  bash
  docker-compose up -d
  ```

- **Stop Services**:

  ```
  bash
  docker-compose down
  ```

- **List Running Services**:

  ```
  bash
  docker-compose ps
  ```

- **Rebuild Services**:

  ```
  docker-compose build
  ```

Benefits of Docker Compose

- Simplifies deployment and management of multi-container applications.
- Centralized configuration for services, volumes, and networks.
- Handles container dependencies automatically (via `depends_on`).
- Works seamlessly with CI/CD pipelines for automated deployments.

---

1. Docker Networking Basics

Docker creates an isolated network environment for containers, allowing them to communicate securely.

- Containers can communicate with each other through networks.
- Each container has its own **virtual network interface** and IP address.

---

2. Docker Network Types

Docker provides several built-in network drivers:

a. Bridge Network (Default)

- **Purpose**: Used for standalone containers on a single host.
- **Features**:
  - Containers on the same bridge network can communicate using container names.
  - Default network created when Docker is installed.
- **Use Case**: Running multiple containers that need to interact (e.g., web server and database).
- **Commands**:
  - List bridge networks:

    ```
    docker network ls
    ```

  - Inspect bridge network details:

    ```
    docker network inspect bridge
    ```

b. Host Network

- **Purpose**: Removes network isolation; the container shares the host's network stack.
- **Features**:

    - No additional IP address is assigned to the container.
    - Direct access to the host's network.
- **Use Case**: High-performance networking or when container-to-host communication latency is critical.
- **Commands**:
    - Use the host network:

```
docker run --network host <image>
```

c. None Network

- **Purpose**: Completely disables networking for the container.
- **Features**:
    - The container is isolated with no network access.
- **Use Case**: For testing or running secure, network-independent applications.
- **Commands**:
    - Use no network:

```
bash
Copy code
docker run --network none <image>
```

d. Overlay Network

- **Purpose**: Enables communication between containers across multiple Docker hosts (requires Docker Swarm or Kubernetes).
- **Features**:
    - Containers across different nodes in a cluster can communicate securely.
- **Use Case**: Distributed applications in a swarm or Kubernetes cluster.
- **Commands**:
    - Create an overlay network:

```
docker network create -d overlay my-overlay-network
```

e. Macvlan Network

- **Purpose**: Assigns a unique MAC address to each container, allowing it to appear as a physical device on the network.
- **Features**:
    - Direct communication with the physical network.
- **Use Case**: When containers need to appear as physical devices, such as for legacy systems.
- **Commands**:
    - Create a macvlan network:

```
docker network create -d macvlan \
  --subnet=192.168.1.0/24 \
  --gateway=192.168.1.1 \
  -o parent=eth0 my-macvlan-network
```

---

3. Connecting Containers

- **By Default**: Containers can communicate if they are on the same network.
- **Manually Connecting**:
    - Connect a running container to a network:

      ```
      docker network connect <network> <container>
      ```

    - Disconnect a container from a network:

      ```
      docker network disconnect <network> <container>
      ```

---

4. Communication Between Containers

- Use **container names** or **service discovery**.
- Example: In a bridge network, one container can connect to another by using its name:

  ```
  ping container_name
  ```

---

5. Inspecting Networks

- **View all networks**:

  ```
  docker network ls
  ```

- **Inspect a specific network**:

  ```
  docker network inspect <network_name>
  ```

---

6. Custom Networks

You can create custom networks for better isolation and control.

- **Create a custom bridge network**:

  ```
  docker network create my-custom-network
  ```

- **Run a container on a custom network**:

  ```
  bash
  Copy code
  docker run --network my-custom-network <image>
  ```

---

7. Docker DNS

- Docker includes an embedded **DNS server**.

- Containers can resolve each other by name when on the same network.

---

8. Ports and Networking

- Map container ports to the host to allow external access.

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

- Use `EXPOSE` in a Dockerfile to document which ports the container uses.

---

9. Advanced Networking Tools

- **Third-party plugins**: Docker supports network plugins for custom solutions.
- **Tools like Traefik or Nginx**: Often used for load balancing and reverse proxying in containerized applications.

---

10. Security in Docker Networking

- Limit container communication by using **custom networks**.
- Use **firewall rules** and **network policies**.
- Use **encrypted overlay networks** for secure communication.

Docker Commands Cheat Sheet

Basic Operations

✔ `docker --version` - Display the installed Docker version
✔ `docker info` - Display detailed system-wide information about Docker
✔ `docker help` - List all available Docker commands or get help for a specific command

---

Image Management

✔ `docker images` - List all locally available Docker images
✔ `docker pull <image>` - Download an image from a Docker registry (e.g., Docker Hub)
✔ `docker push <image>` - Upload a locally tagged image to a registry
✔ `docker rmi <image>` - Remove a Docker image
✔ `docker tag <image> <new_tag>` - Add a new tag to an existing image
✔ `docker build -t <image_name> .` - Build an image from a Dockerfile

Container Management

✔ `docker ps` - List all running containers
✔ `docker ps -a` - List all containers, including stopped ones
✔ `docker run <image>` - Run a container from an image
✔ `docker run --name <name> -d <image>` - Run a container in detached mode with a custom name
✔ `docker run -it <image>` - Run a container in interactive mode (e.g., for a shell session)
✔ `docker stop <container>` - Stop a running container
✔ `docker start <container>` - Start a stopped container
✔ `docker restart <container>` - Restart a container
✔ `docker rm <container>` - Remove a stopped container

Volume Management

✔ `docker volume ls` - List all Docker volumes
✔ `docker volume create <name>` - Create a new volume
✔ `docker volume inspect <name>` - Display detailed information about a volume
✔ `docker volume rm <name>` - Remove a volume

Network Management

✔ `docker network ls` - List all Docker networks
✔ `docker network create <name>` - Create a new network
✔ `docker network inspect <name>` - Display detailed information about a network
✔ `docker network connect <network> <container>` - Connect a container to a network
✔ `docker network disconnect <network> <container>` - Disconnect a container from a network

Logging and Monitoring

✔ `docker logs <container>` - View logs for a container
✔ `docker stats` - Display real-time statistics for running containers
✔ `docker top <container>` - Display running processes in a container

Exec and Debugging

✔ `docker exec -it <container> <command>` - Run a command in a running container (e.g., a shell session)
✔ `docker inspect <container|image>` - Display detailed information about a container or image
✔ `docker events` - Stream real-time Docker events

Pruning Unused Resources

✔ `docker system prune` - Remove unused data (containers, networks, images)
✔ `docker system prune -a` - Remove all unused images, containers, networks, and volumes
✔ `docker container prune` - Remove stopped containers
✔ `docker image prune` - Remove dangling images
✔ `docker volume prune` - Remove unused volumes
✔ `docker network prune` - Remove unused networks

Docker Compose Commands

✔ `docker-compose up` - Start all services defined in the `docker-compose.yml` file
✔ `docker-compose up -d` - Start services in detached mode
✔ `docker-compose down` - Stop and remove all services and networks
✔ `docker-compose ps` - List running services
✔ `docker-compose build` - Build images for services
✔ `docker-compose logs` - View logs for all services
✔ `docker-compose exec <service> <command>` - Execute a command in a running service

Exporting and Importing

✔ `docker save <image> -o <file.tar>` - Save an image to a tar archive
✔ `docker load -i <file.tar>` - Load an image from a tar archive
✔ `docker export <container> -o <file.tar>` - Export a container’s filesystem
✔ `docker import <file.tar>` - Import a tar archive as an image

Swarm and Orchestration

✔ `docker swarm init` - Initialize a Docker Swarm cluster
✔ `docker swarm join` - Join a Swarm as a worker or manager
✔ `docker node ls` - List nodes in a Swarm
✔ `docker service create --name <name> <image>` - Deploy a service in Swarm
✔ `docker service ls` - List all services in the Swarm

Security and User Management

✔ `docker login` - Log in to a Docker registry
✔ `docker logout` - Log out from a Docker registry
✔ `docker secret create <name> <file>` - Create a secret for a Swarm
✔ `docker secret ls` - List all secrets

💜 **Bonus Tips** 💜
✔ Use `docker context` to manage multiple Docker environments
✔ Leverage multi-stage builds to reduce image sizes
✔ Use lightweight base images (e.g., `alpine`) for faster build times and smaller images