

## Theory Questions

### a) Docker Fundamentals

- What is a Docker container, and how is it different from a virtual machine (VM)?

A Docker container is a lightweight, portable, and self-contained unit that packages an application along with its dependencies (such as libraries, configurations, and other required files). It ensures the application can run consistently across various environments, from a developer's local machine to production servers. Containers include everything needed for the application to function, except for the underlying operating system, making them highly efficient and isolated.

What is the purpose of a Dockerfile? Explain the significance of directives like FROM, COPY, RUN, and CMD.

A Dockerfile is a plain text file containing a sequence of instructions that Docker uses to build a Docker image. Each instruction in the Dockerfile contributes a layer to the image, ultimately resulting in a packaged environment ready for deployment as a Docker container. It defines how the application environment is set up, what dependencies are needed, and how the application should be executed. This ensures consistency and reproducibility across environments.

## Key Dockerfile Directives

### 1. FROM

- **Purpose:** Specifies the base image for building the Docker image. Every Dockerfile must start with a FROM directive to define the foundation (e.g., operating system, language runtime, etc.).
- **Example:**

```
dockerfile
Copy code
FROM python:3.9-slim
```

This uses the official Python 3.9 slim image as the base, providing a minimal Python runtime environment to build upon.

- **Significance:** The FROM directive sets the starting point for the image, such as an operating system (e.g., Ubuntu, Alpine) or a language runtime (e.g., Python, Node.js).

### 2. COPY

- **Purpose:** Transfers files or directories from the host system into the Docker image.
- **Example:**

```
dockerfile
Copy code
COPY ./app /app
```

This copies the contents of the ./app directory on the host machine to the /app directory inside the container.

- **Significance:** The COPY directive is essential for adding source code, configuration files, or other resources needed for the application to run inside the container.

### 3. RUN

- **Purpose:** Executes commands during the image build process. Commonly used for installing dependencies, updating packages, or performing setup tasks.
- **Example:**

```
dockerfile
Copy code
RUN apt-get update && apt-get install -y curl
```

This updates the package manager and installs the `curl` package inside the container.

- **Significance:** The `RUN` directive enables you to configure the image by adding software or preparing the environment. Since each `RUN` command creates a new layer, it's best to combine commands to reduce image size.

### 4. CMD

- **Purpose:** Specifies the default command to execute when a container is started from the built image.
- **Example:**

```
dockerfile
Copy code
CMD ["python", "app.py"]
```

This sets the default behavior of the container to execute `python app.py` when it starts.

- **Significance:** The `CMD` directive defines the main process to run within the container. If another command is provided during container startup, it overrides the `CMD` directive.

## Image Management

- Describe the layers of a Docker image. How does Docker optimize space and performance using these layers?

## How Docker Optimizes Space and Performance Using Layers

### 1. Layer Caching

- Docker caches each layer after it is built. If no changes are detected in a specific instruction, Docker reuses the cached layer instead of rebuilding it.
- **Example:** If the `RUN apt-get update` command hasn't changed, Docker skips re-executing it and uses the cached result.
- **Benefit:** Faster builds, especially for large applications with many dependencies, since Docker avoids redundant work.

### 2. Layer Reuse Across Images

- Layers are shared between different images. When multiple images use the same base image or dependencies, Docker stores these layers only once.
- **Example:** If multiple images are based on `python:3.9-slim` and install the same libraries, Docker stores the shared layers a single time.

- **Benefit:** Efficient storage by avoiding duplication of commonly used layers across images.

### 3. Layered Filesystem

- Docker uses a union filesystem (e.g., OverlayFS) to stack layers. Each layer adds or modifies files from the previous layer without duplicating them.
- **Example:** If a `node_modules` directory is added in one layer, installing additional dependencies later creates a new layer, leaving the original unchanged.
- **Benefit:** Only changes are recorded, reducing duplication and improving storage efficiency.

### 4. Minimizing Layer Size

- Combining commands into fewer instructions reduces the number of layers, resulting in smaller, more efficient images.
- **Example:** Instead of:

```
dockerfile
Copy code
RUN apt-get update
RUN apt-get install -y curl
```

Combine into:

```
dockerfile
Copy code
RUN apt-get update && apt-get install -y curl
```

- **Benefit:** Fewer layers mean smaller images with reduced overhead and better performance.

### 5. Writable Container Layer

- When a container starts, it adds a writable layer on top of the image layers. Any changes made within the container, such as file creation or configuration updates, occur in this layer.
- **Benefit:** Containers remain ephemeral—changes are discarded when the container is removed unless persisted using volumes. This design ensures minimal overhead while maintaining a clean state for containers.

- What are the benefits of using Docker volumes? Give an example where data persistence is crucial in a Docker container.

## What Are Docker Volumes?

Docker volumes are a mechanism for persisting and managing data independently of a container's lifecycle. They provide a way to store data outside the container's writable layer, ensuring that the data remains intact even if the container is stopped, restarted, or deleted. Volumes enable data sharing between containers and allow data to persist beyond the lifespan of any single container.

## Benefits of Using Docker Volumes

### 1. Data Persistence

- Volumes ensure data is retained even when containers are stopped, restarted, or removed. Without volumes, any data written to the container's internal filesystem would be lost upon container deletion.
2. **Separation of Data and Application**
    - Storing data (e.g., databases or logs) outside the container's filesystem simplifies data management, backup, and recovery without affecting the application itself.
  3. **Data Sharing Between Containers**
    - Multiple containers can read from or write to the same volume. This is particularly useful when applications, like a web server and a database, need to share data.
  4. **Improved Performance**
    - Volumes are optimized for Input/Output (I/O) operations and are more efficient than using a container's internal filesystem for heavy or frequent read/write tasks.
  5. **Backup and Recovery**
    - Volumes facilitate easy data backup and migration. Data stored in volumes can be copied, moved, or restored between environments as needed.
  6. **Data Integrity**
    - Docker manages volumes independently of containers, ensuring that data remains intact and available even during container restarts or deletions.
  7. **Cross-Platform Compatibility**
    - Volumes work seamlessly across various environments, whether on local machines, in the cloud, or in production, maintaining data portability and consistency.

## Example: Persisting Data with Docker Volumes

A common scenario where data persistence is crucial is when running a database in a Docker container. Databases store critical information, and losing this data due to container deletion or restarts could lead to significant failures. Docker volumes prevent such data loss by ensuring persistence.

### Example: Using Docker Volumes with MySQL

**Scenario:** You are running a MySQL database container and need to ensure the database data persists.

1. **Run the MySQL container with a volume:**

```
bash
Copy code
docker run -d \
--name mysql-container \
-e MYSQL_ROOT_PASSWORD=root_password \
-v mysql-data:/var/lib/mysql \
mysql:8
```

- `-v mysql-data:/var/lib/mysql`: Creates a volume named `mysql-data` and mounts it to the `/var/lib/mysql` directory in the container, which is where MySQL stores its data.
2. **Data Persistence:**

- The `mysql-data` volume will store the MySQL database files. This ensures that the data remains intact even if the container is stopped or removed.

### 3. Verify Data Persistence:

- Stop and remove the container:

```
bash
Copy code
docker stop mysql-container
docker rm mysql-container
```

- Re-run the MySQL container using the same volume:

```
bash
Copy code
docker run -d \
--name mysql-container \
-e MYSQL_ROOT_PASSWORD=root_password \
-v mysql-data:/var/lib/mysql \
mysql:8
```

- The new container will access the same data stored in the `mysql-data` volume, preserving all the database contents from the previous container.

## c) Networking in Docker

- How does Docker handle networking? Explain the difference between bridge, host, and none network modes in Docker.

### How Does Docker Handle Networking?

Docker provides several networking options to enable communication between containers, the host system, and external networks. Docker employs network drivers to manage these options, each tailored to specific connectivity and isolation needs.

By default, Docker creates a virtual network bridge for containers, allowing them to communicate internally and with the outside world via port mapping. Additional networking modes provide flexibility based on performance, isolation, or direct connectivity requirements.

## Types of Docker Network Modes

### 1. Bridge Mode (Default)

#### What It Is:

- In bridge mode, Docker creates a private internal network (via a virtual bridge) on the host machine. Containers connected to this network can communicate with one another but require port mapping to interact with external systems.

#### Use Case:

- Ideal for containers that need to communicate with each other on the same host but require explicit configuration to interact with the host's network or external services.

#### How It Works:

- Docker sets up a virtual bridge (e.g., `docker0`) on the host system.

- Each container gets its own unique IP address within the bridge network.
- Port mapping (-p) allows external access to container services.

#### Example:

```
bash
Copy code
docker run -d --name mycontainer --network bridge myimage
```

#### Key Points:

- Containers on the same bridge network can communicate directly.
- Communication with external networks requires port mapping.
- Provides isolation between the container and the host's network.

## 2. Host Mode

#### What It Is:

- In host mode, the container shares the host system's network stack. The container's networking is not isolated from the host, using the host's IP address and ports directly.

#### Use Case:

- Used when a container requires full access to the host's network resources, often for performance or to interact closely with the host's network interfaces.

#### How It Works:

- The container does not get a unique IP address.
- Ports opened by the container are directly mapped to the host's ports, leading to potential conflicts.

#### Example:

```
bash
Copy code
docker run -d --name mycontainer --network host myimage
```

#### Key Points:

- No network isolation between the container and the host.
- Performance benefits as there's no network address translation (NAT) overhead.
- Useful for services like web servers that need direct access to host network interfaces.

## 3. None Mode

#### What It Is:

- In none mode, the container is completely isolated from any network. It has no network interfaces and cannot communicate with the host, other containers, or external systems.

#### Use Case:

- Ideal for containers that do not require networking, such as running isolated applications or for testing purposes.

#### How It Works:

- The container is not assigned an IP address or network interface.
- Communication must be handled explicitly through other mechanisms (e.g., `docker exec`).

### Example:

bash

Copy code

```
docker run -d --name mycontainer --network none myimage
```

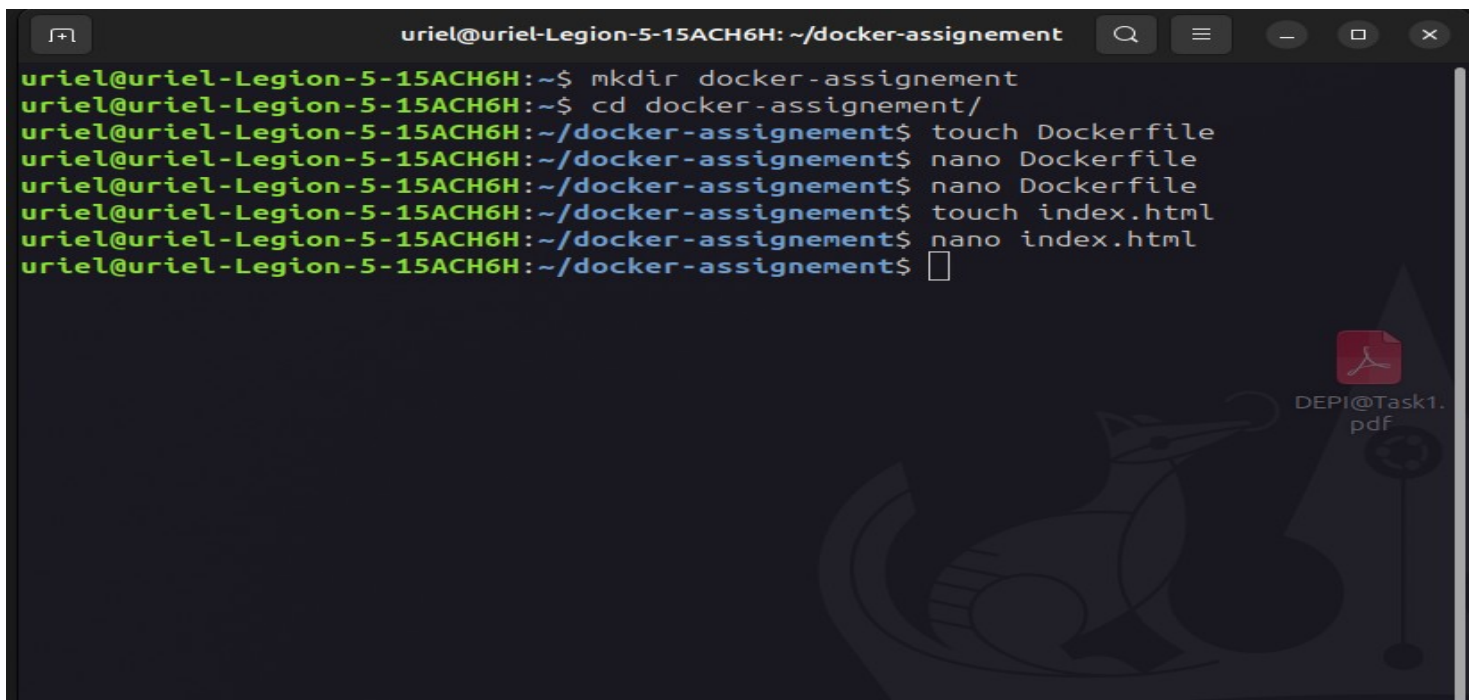
### Key Points:

- No network connectivity is available to the container.
  - Completely isolated from the host and other containers.
  - Suitable for use cases where networking is unnecessary, like batch processing or testing.
- Describe how you would configure container-to-container communication within a Docker network.

Docker offers several methods for enabling communication between containers on the same network. When containers are part of the same Docker network, they can communicate using their container names as hostnames. Docker supports various network drivers, including bridge, host, and overlay. For communication between containers on the same host, the bridge network driver is typically used. However, for more complex use cases, such as multi-host communication, overlay networks are employed.

## Practical Task

### Dockerfile Creation

A terminal window titled 'uriel@uriel-Legion-5-15ACH6H: ~/docker-assignement' showing a series of commands to create a Dockerfile. The commands are: 'mkdir docker-assignement', 'cd docker-assignement/', 'touch Dockerfile', 'nano Dockerfile', 'touch index.html', and 'nano index.html'. The terminal has a dark background with green text. In the bottom right corner, there is a red PDF icon and the text 'DEPI@Task1.pdf'.

```
uriel@uriel-Legion-5-15ACH6H: ~/docker-assignement
uriel@uriel-Legion-5-15ACH6H:~$ mkdir docker-assignement
uriel@uriel-Legion-5-15ACH6H:~$ cd docker-assignement/
uriel@uriel-Legion-5-15ACH6H:~/docker-assignement$ touch Dockerfile
uriel@uriel-Legion-5-15ACH6H:~/docker-assignement$ nano Dockerfile
uriel@uriel-Legion-5-15ACH6H:~/docker-assignement$ touch index.html
uriel@uriel-Legion-5-15ACH6H:~/docker-assignement$ nano index.html
uriel@uriel-Legion-5-15ACH6H:~/docker-assignement$
```



```
uriel@uriel-Legion-5-15ACH6H: ~/docker-assignment
GNU nano 7.2 Dockerfile
# Use Ubuntu as the base image
FROM ubuntu:latest

# Update and install Nginx
RUN apt-get update && apt-get install -y nginx

# Copy the custom index.html to the appropriate Nginx directory
COPY index.html /var/www/html/index.html

# Expose port 8080
EXPOSE 8080

# Start Nginx in the foreground
CMD ["nginx", "-g", "daemon off;"]

[ Read 14 lines ]
^G Help      ^O Write Out ^W Where Is  ^K Cut      ^T Execute  ^C Location
^X Exit      ^R Read File ^\ Replace   ^U Paste     ^J Justify  ^_ Go To Line

uriel@uriel-Legion-5-15ACH6H: ~/docker-assignment
GNU nano 7.2 index.html *
<!DOCTYPE html>
<html>
<head>
  <title>DevOps World</title>
</head>
<body>
  <h1>Welcome to DevOps World!</h1>
</body>
</html>
^G Help      ^O Write Out ^W Where Is  ^K Cut      ^T Execute  ^C Location
^X Exit      ^R Read File ^\ Replace   ^U Paste     ^J Justify  ^_ Go To Line

uriel@uriel-Legion-5-15ACH6H:~/docker-assignment$ sudo docker build -t nginx-devops-world .
[+] Building 26.1s (8/8) FINISHED
docker:default
=> [internal] load build definition from Dockerfile 0.0s
=> => transferring dockerfile: 370B 0.0s
=> [internal] load metadata for docker.io/library/ubuntu:latest 2.2s
=> [internal] load .dockerignore 0.0s
=> => transferring context: 2B 0.0s
=> [1/3] FROM docker.io/library/ubuntu:latest@sha256:80dd3c3b9c6cecb9f1 10.8s
=> => resolve docker.io/library/ubuntu:latest@sha256:80dd3c3b9c6cecb9f16 0.0s
=> => sha256:de44b265507ae44b212defcb50694d666f136b35 29.75MB / 29.75MB 10.1s
=> => sha256:80dd3c3b9c6cecb9f1667e9290b3bc61b78c2678c02 6.69kB / 6.69kB 0.0s
=> => sha256:6e75a10070b0fcb0bead763c5118a369bc7cc30dfc1b074 424B / 424B 0.0s
=> => sha256:b1d9df8ab81559494794e522b380878cf9ba82d4c1f 2.30kB / 2.30kB 0.0s
=> => extracting sha256:de44b265507ae44b212defcb50694d666f136b35c1090d97 0.5s
=> [internal] load build context 0.0s
=> => transferring context: 170B 0.0s
=> [2/3] RUN apt-get update && apt-get install -y nginx 12.9s
=> [3/3] COPY index.html /var/www/html/index.html 0.0s
=> exporting to image 0.1s
=> => exporting layers 0.1s
=> => writing image sha256:15c89222e989734f7bb4b6df8ccc67cdcea91463a8945 0.0s
=> => naming to docker.io/library/nginx-devops-world 0.0s
uriel@uriel-Legion-5-15ACH6H:~/docker-assignment$ sudo docker run -d -p 8080:8080 nginx-devops-world
db2b070374dd9a13e48c7372c50873ee172a5283ef64edebeac4410f56b3d458
uriel@uriel-Legion-5-15ACH6H:~/docker-assignment$
```



## Multi-Container Setup

```

uriel@uriel-Legion-5-15ACH6H: ~/MS-docker
uriel@uriel-Legion-5-15ACH6H:~/docker-assignement$ cd ~
uriel@uriel-Legion-5-15ACH6H:~$ mkdir MS-docker
uriel@uriel-Legion-5-15ACH6H:~$ cd MS-docker/
uriel@uriel-Legion-5-15ACH6H:~/MS-docker$ touch docker-compose.yml
uriel@uriel-Legion-5-15ACH6H:~/MS-docker$ nano docker-compose.yml
uriel@uriel-Legion-5-15ACH6H:~/MS-docker$ nano docker-compose.yml
uriel@uriel-Legion-5-15ACH6H:~/MS-docker$ mkdir html
uriel@uriel-Legion-5-15ACH6H:~/MS-docker$ cd html/
uriel@uriel-Legion-5-15ACH6H:~/MS-docker/html$ touch index.html
uriel@uriel-Legion-5-15ACH6H:~/MS-docker/html$ nano index.html
uriel@uriel-Legion-5-15ACH6H:~/MS-docker/html$ cd MS-docker
bash: cd: MS-docker: No such file or directory
uriel@uriel-Legion-5-15ACH6H:~/MS-docker/html$ cd ~
uriel@uriel-Legion-5-15ACH6H:~$ cd MS-docker/
uriel@uriel-Legion-5-15ACH6H:~/MS-docker$ docker-compose up -d
Command 'docker-compose' not found, but can be installed with:
sudo snap install docker # version 27.2.0, or
sudo apt install docker-compose # version 1.29.2-6
See 'snap info docker' for additional versions.
uriel@uriel-Legion-5-15ACH6H:~/MS-docker$ sudo snap install docker
[sudo] password for uriel:
docker 27.2.0 from Canonical✓ installed
uriel@uriel-Legion-5-15ACH6H:~/MS-docker$ docker-compose up -d
permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Get "http://%2Fvar%2Frun%2Fdocker.sock/v1.24/containers/json?all=1&filters=%7B%22label%22%3A%7B%22com.docker.compose.config-hash%22%3Atrue%2C%22com.docker.compose.project%3Dms-docker%22%3Atrue%7D%7D": dial unix /var/run/docker.sock: connect: permission denied
uriel@uriel-Legion-5-15ACH6H:~/MS-docker$ sudo docker-compose up -d
[+] Running 22/22
 ✓ db 13 layers [██████████] 0B/0B Pulled 66.2s
   ✓ 002e1a8eb6f9 Pull complete 5.5s
   ✓ a24f300391ed Pull complete 9.5s
   ✓ 627f580b7ad7 Pull complete 11.2s
   ✓ cfb3c2203f88 Pull complete 18.9s
   ✓ 9e592465b243 Pull complete 16.7s
   ✓ 8d4265d09d9c Pull complete 17.9s
   ✓ e3a8293e92fd Pull complete 18.9s
   ✓ 2cb801c39436 Pull complete 62.0s
   ✓ c5fdb20d8658 Pull complete 19.6s
   ✓ 67c5fe618f0c Pull complete 22.3s
   ✓ c9cdd1fe82e4 Pull complete 23.5s
   ✓ 8f152c4aceed Pull complete 25.2s
   ✓ 2cd360f3b7db Pull complete 26.4s
 ✓ web 7 layers [██████████] 0B/0B Pulled 30.5s

```

```
uriel@uriel-Legion-5-15ACH6H: ~/MS-docker
GNU nano 7.2 docker-compose.yml
Version: '3.8'

services:
  web:
    image: nginx:latest
    container_name: nginx-web
    ports:
      - "8080:80"
    volumes:
      - ./html:/usr/share/nginx/html
    networks:
      - app-network

  db:
    image: postgres:latest
    container_name: postgres-db
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: appdb
    volumes:
      - pgdata:/var/lib/postgresql/data
    networks:
      - app-network

networks:
  app-network:
    driver: bridge

volumes:
  pgdata:
    driver: local
```

```
uriel@uriel-Legion-5-15ACH6H: ~/MS-docker
uriel@uriel-Legion-5-15ACH6H:~/MS-docker$ sudo docker-compose up -d
[+] Running 1/0
 ✓ Container postgres-db Running 0.0s
 ⋄ Container nginx-web Starting 0.1s
Error response from daemon: driver failed programming external connectivity on endpoint nginx-web (45f625db62d99eb30a7167d1ca425e20dba376f5ea8b183723e8a0dee3873656): failed to bind port 0.0.0.0:8080/tcp: Error starting userland proxy: listen tcp4 0.0.0.0:8080: bind: address already in use
uriel@uriel-Legion-5-15ACH6H:~/MS-docker$ sudo lsof -i :8080
COMMAND      PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
docker-pr 49590 root    4u    IPv4 211711      0t0  TCP *:http-alt (LISTEN)
docker-pr 49604 root    4u    IPv6 207266      0t0  TCP *:http-alt (LISTEN)
uriel@uriel-Legion-5-15ACH6H:~/MS-docker$ sudo kill -9 49590 49604
uriel@uriel-Legion-5-15ACH6H:~/MS-docker$ sudo docker-compose up -d
[+] Running 2/2
 ✓ Container postgres-db Running 0.0s
 ✓ Container nginx-web Started 0.3s
uriel@uriel-Legion-5-15ACH6H:~/MS-docker$ xdg-open http://localhost:8080
uriel@uriel-Legion-5-15ACH6H:~/MS-docker$ Opening in existing browser session.
█
```



## Resource Limiting

```
GNU nano 7.2
version: '3.8'

services:
  web:
    image: nginx:latest
    ports:
      - "8080:80"
    depends_on:
      - db
    deploy:
      resources:
        limits:
          memory: 512M
          cpus: "1.0"
    networks:
      - webnet

  db:
    image: postgres:latest
    environment:
      POSTGRES_USER: exampleuser
      POSTGRES_PASSWORD: examplepass
      POSTGRES_DB: exampledb
    volumes:
      - db_data:/var/lib/postgresql/data
    networks:
      - webnet

networks:
  webnet:

volumes:
  db_data:
```

uriel@uriel-Legion-5-15ACH6H: ~/MS-docker						
CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	
0	BLOCK I/O	PIDS				
24c84466bda1	ms-docker_web_1	0.00%	12.95MiB / 512MiB	2.53%	5.22kB	
/ 0B	0B / 4.1kB	17				
5ef6cc086363	ms-docker_db_1	0.03%	18.75MiB / 13.5GiB	0.14%	5.35kB	
/ 0B	0B / 221kB	6				

## Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to [nginx.org](https://nginx.org). Commercial support is available at [nginx.com](https://nginx.com).

*Thank you for using nginx.*

## Benefits of Docker Resource Limits in a Production Environment

1. **Prevents Resource Contention:** In a production environment with multiple containers running on the same host, it's crucial to allocate resources efficiently. By setting resource limits, you ensure that no container can monopolize the host's CPU or memory, which could cause other containers to become unresponsive.
2. **Improves Stability:** Resource limits can help prevent containers from consuming excessive amounts of resources, especially when under heavy load. This can prevent containers from crashing the host system due to resource exhaustion.
3. **Better Resource Allocation:** You can use the `reservations` feature to guarantee certain resources for critical containers. This ensures that important services (like databases or web servers) always have the necessary resources to run efficiently, even under heavy load.
4. **Cost Optimization:** In cloud environments where you pay for resources (like CPU, memory, and storage), resource limits help to optimize the usage and reduce unnecessary overprovisioning, thus reducing costs.
5. **Predictable Performance:** Setting CPU and memory limits helps ensure predictable behavior, even when the system is under load. It can also help in load testing scenarios, where you need to ensure that the container performs within defined boundaries.
6. **Prevents "Resource Leak":** Without resource limits, containers could potentially consume more resources over time (e.g., memory leaks), which might eventually affect the entire system. Setting limits allows you to catch these issues early, as containers will be automatically restarted when they exceed the limits.

By using resource limits effectively, you can maintain a stable and performant environment, especially in production where reliability is critical.