Feel free to delve into these topics, and if you have questions or need further guidance on any specific area, don't hesitate to ask!



list out all of the docker concept in beginner to intermediate give me all list

Here is a **complete list of Docker concepts**, organized from **beginner** to **intermediate** level, to help you build your skills step-by-step.

Beginner Level

1. What is Docker?

- Overview of containers vs virtual machines
- Why use Docker?

2. Installing Docker

- Docker Desktop (Windows/Mac)
- Docker Engine (Linux)

3. Basic Docker CLI Commands

- docker run
- docker ps , docker ps -a
- docker stop, docker start
- docker rm, docker rmi
- docker images, docker pull, docker push

4. Working with Docker Images

- Using official images from Docker Hub
- docker build -t name . to build custom images
- Image layers and caching

5. Dockerfile Basics

- FROM , RUN , COPY , CMD , EXPOSE , WORKDIR
- Creating minimal Docker images

6. Running Containers

- Detached vs interactive mode (-d , -it)
- Port binding: -p host:container
- Volume mounting: -v host:container
- Environment variables: -e VAR=value

7. Networking Basics

- Bridge network (default)
- Container-to-container communication
- Host and none network drivers

8. Using Volumes

- Anonymous, named, and host volumes
- Persistent storage
- docker volume create, inspect, rm

9. Docker Compose (Intro)

- docker-compose.yml structure
- Running multi-container apps
- docker-compose up, down, logs

Intermediate Level

10. Advanced Dockerfile

- ENTRYPOINT VS CMD
- ARG VS ENV

- LABEL for metadata
- .dockerignore file

11. Multi-Stage Builds

- Reduce image size
- Separate build and runtime steps

12. Container Logs and Debugging

- docker logs
- docker exec, docker inspect
- docker top, docker stats

13. Docker Compose (Intermediate)

- Defining networks and volumes
- depends_on, build, context
- Override files and environment variable substitution

14. Networking Deep Dive

- User-defined bridge networks
- Overlay network (Swarm-related)
- Container DNS resolution

15. Docker Volumes (Advanced)

- Bind mounts vs volumes
- Backup and restore volume data
- Using volumes in docker-compose

16. Image Optimization

- Smaller base images (alpine)
- Reducing layers and caching
- Cleaning up unused artifacts

17. Docker Registry

- Docker Hub
- Private registries
- Pushing/pulling images (docker login, tag, push)

18. Health Checks

- HEALTHCHECK in Dockerfile
- Checking app status inside a container

19. Security Best Practices

- Avoid running as root inside containers
- Scan images for CVEs (docker scout cves)
- Minimize image surface

20. Docker Contexts

Switching between local and remote Docker engines

Optional Next Steps Toward Advanced (Preview)

- Docker Swarm
- Docker Secrets
- Docker Events
- Custom Docker Networks and Plugins
- Introduction to Kubernetes (Docker to K8s migration)

Would you like exercises or projects based on this list? I can suggest hands-on labs for each section.

Keep all these labs in your mind i will ask you one by one ok?

Got it, Rushikesh!

I've noted all the labs and concepts you've completed, and I'm ready to help you with each of the remaining ones **one by one** whenever you ask.

Just message me with the next topic or say something like:

"Let's start with Docker Compose" or

"Give me an exercise on Docker Networking"

I'll guide you step-by-step with code, commands, and explanations.

Ready when you are! 🚀 🐒

i have completed till level 1 and 2 which you given NOW

For the above list

make new saperate labs for level 3 which is - 3. Basic Docker CLI Commands

give me steps to learn level 3 with details steps and informations

Perfect, Rushikesh! Since you've completed Level 1 and Level 2, you're now ready for:

✓ Level 3: Basic Docker CLI Commands

Goal: Understand and practice the most commonly used Docker CLI commands for containers, images, and system management.

LAB 3.1: Listing and Inspecting Docker Containers

Objective: Learn to list, inspect, and manage containers.

1. List all running containers

bash

docker ps

- Shows running containers.
- Columns: CONTAINER ID , IMAGE , COMMAND , STATUS , etc.

2. List all containers (including stopped ones)

```
bash

docker ps -a
```

3. Inspect a container

```
bash
docker inspect <container_id or name>
```

Outputs detailed JSON about the container (network, volumes, config).

4. View container logs

```
bash
docker logs <container_id>
```

5. View container stats (like CPU, memory)

```
bash

docker stats
```

Try this:

```
docker run -d --name test-nginx nginx
docker ps
docker logs test-nginx
docker inspect test-nginx
docker stats
```

LAB 3.2: Managing Docker Images

Objective: Learn to list, inspect, and remove Docker images.

→ Steps:

1. List Docker images

```
bash
docker images
```

2. Remove an image

```
bash
docker rmi <image_name or id>
```

3. Pull an image manually

```
bash
docker pull alpine
```

4. Inspect an image

```
bash
docker inspect alpine
```

Try this:

```
bash
docker pull alpine
docker images
docker inspect alpine
docker rmi alpine
```

LAB 3.3: Starting, Stopping, and Removing Containers

Objective: Learn container lifecycle management.

→ Steps:

1. Start a stopped container

```
bash

docker start <container_id or name>
```

2. Stop a running container

```
bash
docker stop <container_id or name>
```

3. Restart a container

```
bash
docker restart <container_id or name>
```

4. Remove a container

```
bash

docker rm <container_id or name>
```

5. Force remove (even if running)

```
bash

docker rm -f <container_id>
```

Try this:

```
docker run -d --name temp busybox sleep 1000
docker stop temp
docker start temp
docker restart temp
docker rm -f temp
```

LAB 3.4: Prune Unused Data

Objective: Clean up your Docker environment.

→ Steps:

1. Remove all stopped containers

```
bash

docker container prune
```

2. Remove all unused images

```
bash

docker image prune
```

3. Remove everything (volumes, networks, etc.)

```
bash

docker system prune
```

Try this:

```
docker system df  # Show used disk space
docker system prune  # Clean up everything not used
```

▲ Warning: Use with caution. You may lose containers and images if not careful.

6 What's Next After Level 3?

Once you're comfortable with CLI basics, you'll move to:

- Level 4: Docker Volumes
- Level 5: Docker Networking
- Z Level 6: Docker Compose
- Level 7: Dockerfile Best Practices

- Level 8: Build Multi-Container Applications
- Z Level 9: Docker and CI/CD
- Z Level 10: Kubernetes Basics (Optional for now)

Would you like to start with LAB 3.1 now or want to review any part from Level 1/2 again?

Just say the word!

2/2

Now next topic give me exercies on 4. Working with Docker Images

Great, Rushikesh! Now that you've mastered Levels 1–3, let's dive into:

Level 4: Working with Docker Images

Goal: Understand how Docker images work, how to pull, build, tag, optimize, and manage them.

- What Are Docker Images?
- Images are blueprints used to create containers.
- They are made up of layers, which makes them efficient and reusable.

LABS for Level 4: Docker Images

LAB 4.1: Pull Images from Docker Hub

→ Steps:

1. Pull an official image:

```
bash

docker pull nginx
```

2. Run a container from it:

```
bash

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

3. Open http://localhost:8080 to see the Nginx welcome page.

LAB 4.2: Build Your Own Image from a Dockerfile

Files:

Dockerfile:

```
FROM alpine
RUN echo "This is a custom Docker image!" > /message.txt
CMD ["cat", "/message.txt"]
```


1. Build the image:

```
bash

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

2. Run it:

```
docker run my-custom-image
```

Output: "This is a custom Docker image!"

渗 LAB 4.3: Tagging Docker Images

→ Steps:

1. Tag your custom image:

```
bash
docker tag my-custom-image myname/hello-image:1.0
```

2. List all images:

```
bash

docker images
```

You'll see the image listed under your custom name.

LAB 4.4: Remove Docker Images

→ Steps:

1. Remove an image:

```
docker rmi my-custom-image
```

2. Remove with tag:

```
bash

docker rmi myname/hello-image:1.0
```

LAB 4.5: Optimize Docker Images (Minimize Size)

Compare these two Dockerfiles:

A: Unoptimized

```
FROM python:latest
RUN pip install flask
COPY app.py /app.py
CMD ["python", "/app.py"]
```

B: Optimized

```
dockerfile

FROM python:3.12-alpine

RUN pip install flask

COPY app.py /app.py

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

→ Steps:

1. Build both:

```
docker build -t python-fat -f Dockerfile-fat .
docker build -t python-slim -f Dockerfile-slim .
```

2. Compare size:

```
bash

docker images
```

python-slim will be much smaller.

LAB 4.6: View Image History & Layers

→ Steps:

1. View image history:

```
bash
```

docker history python-slim

2. Inspect layers:

docker inspect python-slim

LAB 4.7: Save and Load Docker Images

→ Steps:

1. Save image to .tar:

docker save -o myimage.tar python-slim

2. Load from .tar:

bash

docker load -i myimage.tar

LAB 4.8: Push Image to Docker Hub

You need a Docker Hub account.

1. Log in:

bash

docker login

2. Tag your image:

bash

docker tag my-custom-image yourdockerhubusername/myimage:1.0

3. Push to Docker Hub:

bash

docker push yourdockerhubusername/myimage:1.0

Summary of Level 4

Lab No	Concept	Command Example
4.1	Pull image	docker pull nginx
4.2	Build custom image	docker build -t my-image .
4.3	Tag image	<pre>docker tag my-image name/image:tag</pre>
4.4	Remove image	docker rmi my-image
4.5	Optimize size with alpine	FROM python:alpine
4.6	View image history & layers	docker history, docker inspect
4.7	Save/load image	docker save/load
4.8	Push to Docker Hub	docker push

Would you like to start with LAB 4.1 now? Or want me to give you files/templates for any lab above?

5. Dockerfile Basics FROM, RUN, COPY, CMD, EXPOSE, WORKDIR

Creating minimal Docker images

now give me labs again like previous

Perfect, Rushikesh! You're progressing like a pro. Let's now jump into:

Level 5: Dockerfile Basics

Learn how to write efficient Dockerfiles using FROM, RUN, COPY, CMD, EXPOSE, WORKDIR, and also how to create minimal and optimized Docker images.

LABS for Level 5: Dockerfile Basics

LAB 5.1: FROM – Base Image

Dockerfile:

```
dockerfile

FROM alpine

CMD ["echo", "This is Alpine Base Image"]
```

Steps:

```
docker build -t lab5-from .

docker run lab5-from
```

☑ Output: This is Alpine Base Image

LAB 5.2: RUN – Execute Commands While Building

Dockerfile:

```
FROM alpine
RUN apk add --no-cache curl
CMD ["curl", "https://example.com"]
```

→ Steps:

```
docker build -t lab5-run .
docker run lab5-run
```

Output: You'll see HTML from example.com

LAB 5.3: COPY – Add Files into Image

- Files:
- Dockerfile
- message.txt (create this manually with some text)

```
FROM alpine
COPY message.txt /message.txt
CMD ["cat", "/message.txt"]
```

→ Steps:

```
docker build -t lab5-copy .
docker run lab5-copy
```

✓ Output: Contents of message.txt

LAB 5.4: CMD – Default Command

Dockerfile:

```
dockerfile

FROM alpine

CMD ["echo", "Default command runs"]
```

Run normally:

```
bash
docker run lab5-cmd
```

Override CMD:

```
bash

docker run lab5-cmd echo "Overridden Command"
```

See both default and overridden results

LAB 5.5: EXPOSE – Declare Port

EXPOSE doesn't publish the port but documents which port the container listens on.

Dockerfile:

```
dockerfile

FROM python:3-alpine

WORKDIR /app

COPY app.py /app/

EXPOSE 5000

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

app.py:

python

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def home():
    return "EXPOSE test app"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```



```
docker build -t lab5-expose .
docker run -p 5000:5000 lab5-expose
```

✓ Visit http://localhost:5000

LAB 5.6: WORKDIR – Set Working Directory

Dockerfile:

```
dockerfile

FROM alpine
WORKDIR /data
COPY message.txt .
CMD ["cat", "message.txt"]
```

message.txt will be placed in /data and executed from there.

LAB 5.7: Create Minimal Image with Alpine and Multi-stage

Files:

app.py

```
print("Hello from a slim Python app!")
```

Dockerfile

```
# Stage 1: Build
FROM python:3.12 as builder
WORKDIR /src
COPY app.py .

# Stage 2: Final minimal image
FROM python:3.12-alpine
WORKDIR /app
COPY --from=builder /src/app.py .

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

Steps:

```
docker build -t lab5-minimal .

docker run lab5-minimal
```

Output: "Hello from a slim Python app!"

✓ Summary Table – Dockerfile Basics

Lab	Concept	Command/Use
5.1	FROM	FROM alpine
5.2	RUN	RUN apk add curl
5.3	СОРУ	COPY local_file /path/
5.4	CMD	CMD ["python", "app.py"]
5.5	EXPOSE	EXPOSE 8080

Lab	Concept	Command/Use
5.6	WORKDIR	WORKDIR /app
5.7	Minimal Image	Multi-stage build using Alpine

Would you like me to give files for these labs directly to copy-paste? Or would you like to move to Level 6: Volumes & Bind Mounts next?

Now moving to the next topic

6. Running Containers

Detached vs interactive mode (-d, -it)

Port binding: -p host:container

Volume mounting: -v host:container

Environment variables: -e VAR=value

now give me labs on this topic, like you gave me on last topic

Awesome, Rushikesh! You're moving ahead brilliantly 🚀 Now let's dive into:

Level 6: Running Containers

Learn the important Docker run-time flags and concepts: detached/interactive, port binding, volume mounting, and environment variables.

LABS for Level 6: Running Containers

LAB 6.1: Interactive Mode (-it)

Run a container interactively using a shell.

Command:

```
bash

docker run -it alpine /bin/sh
```

✓ Inside the container, run:

```
sh

echo "Hello from interactive container"
```

Exit:

```
sh exit
```

LAB 6.2: Detached Mode (-d)

Run a container in the background (detached).

Dockerfile:

```
dockerfile

FROM python:alpine

CMD ["python", "-m", "http.server", "8000"]
```


bash

```
docker build -t lab6-detached .
docker run -d -p 8080:8000 lab6-detached
```

- ✓ Visit: http://localhost:8080
- To check logs:

```
bash

docker ps
docker logs <container_id>
```

LAB 6.3: Port Binding (-p)

Map a port from host to container.

Use any simple server like Flask or Python's http.server.

```
bash

docker run -d -p 9090:8000 python:alpine python -m http.server 8000
```

✓ Visit: http://localhost:9090

🥕 LAB 6.4: Volume Mounting (-v)

Mount your host file into the container.

→ Steps:

1. Create message.txt:

```
echo "Live update via volume!" > message.txt
```

2. Run the container:

```
docker run --rm -v $(pwd)/message.txt:/data/message.txt alpine cat /data/message.txt
```

- ✓ Output: Live update via volume!
- Edit message.txt and re-run for updated result.

LAB 6.5: Bind Mount with Python Reader

♦ Files:

script.py

```
python
with open("/data/file.txt") as f:
    print("From file:", f.read())
```

1. Create a file:

```
echo "Dynamic content via bind mount" > file.txt
```

2. Run:

```
bash

docker run --rm -v $(pwd)/file.txt:/data/file.txt -v $(pwd)/script.py:/script.py
python:alpine python /script.py
```

Output: From file: Dynamic content via bind mount

LAB 6.6: Environment Variables (-e)

♦ Files:

env_app.py

```
import os
name = os.getenv("USERNAME", "Guest")
print(f"Hello, {name}!")
```

Dockerfile:

```
FROM python:alpine
COPY env_app.py .
CMD ["python", "env_app.py"]
```

→ Build:

```
bash

docker build -t lab6-env .
```

Nun with environment variable:

```
bash

docker run --rm -e USERNAME=Rushikesh lab6-env
```

Output: Hello, Rushikesh!

Bonus: Combine All

```
docker run -it -d \
    -p 8888:8000 \
    -v $(pwd)/file.txt:/data/file.txt \
    -e USERNAME=Rushikesh \
    --name full-lab \
    python:3-alpine \
    python -m http.server 8000
```

✓ All features combined.

Summary Table

Lab	Concept	Command/Example
6.1	-it	docker run -it alpine sh
6.2	-d	docker run -d -p 8080:8000
6.3	Port Binding	-p 9090:8000
6.4	Volume Mount	<pre>-v \$(pwd)/file.txt:/data/file.txt</pre>
6.5	Bind Mount	For real-time host-file read
6.6	Env Variables	-e USERNAME=Rushikesh

Let me know when you're ready for **Level 7**: **Docker Volumes** or want me to generate files/templates for these labs .

Now moving to the next topic7. Networking BasicsBridge network (default)

Container-to-container communication

Host and none network drivers

now give me labs on this topic, like you gave me on last topic

Great work progressing to Level 7: Networking Basics!

These labs will help you understand how Docker containers communicate with each other and the outside world using different **network drivers**.

Level 7: Docker Networking Basics

Key concepts:

- bridge (default network)
- container-to-container communication
- host and none network drivers

LABS for Level 7: Networking

🥕 LAB 7.1: Default Bridge Network

Verify that containers on the same default bridge cannot talk to each other by name (without extra configuration).

Steps:

```
bash
docker run -dit --name container1 alpine sh
docker run -dit --name container2 alpine sh
```

Then inside container1:

```
bash
docker exec -it container1 sh
ping container2
```

Expected result: ping will fail because default bridge doesn't resolve container names.

LAB 7.2: Custom Bridge Network with Container-to-Container Communication

Create a custom bridge network so containers can talk by name.

Create custom network:

```
bash

docker network create my_bridge
```

↑ Start containers in same network:

```
docker run -dit --name web1 --network my_bridge alpine sh
docker run -dit --name web2 --network my_bridge alpine sh
```

♦ Test communication:

```
bash

docker exec -it web1 ping web2
```

Expected result: Success! web2 will be resolved by name.

LAB 7.3: Simple Web Server to Test Networking

Create a tiny HTTP server in one container and access it from another.

Step 1: Create network

```
docker network create mynet
```

Step 2: Create a simple Python HTTP server

bash

docker run -dit --name server --network mynet python:alpine sh -c "python -m http.server 8000"

Step 3: Curl from another container:

```
bash

docker run --rm --network mynet curlimages/curl curl server:8000
```

Output should be the directory listing from the server container.

LAB 7.4: Host Network Driver (Linux Only)

Use the host's own networking stack (Linux only).

```
bash

docker run --rm --network host nginx
```

✓ Visit http://localhost — Nginx is directly bound to the host's port 80.

⚠ On Windows/Mac, host mode behaves differently or is unsupported. You can skip this if not on Linux.

LAB 7.5: None Network Driver

Use --network none to completely **isolate** the container from all networks.

```
docker run -dit --name isolated --network none alpine sh
docker exec -it isolated ping google.com
```

Expected: ping fails — container has no network access.

Summary Table

Lab	Concept	Key Commands
7.1	Default Bridge (no DNS)	ping container-name fails
7.2	Custom Bridge (works)	docker network create +network mynet
7.3	Container-to-container HTTP	Server & Curl in same network
7.4	Host Network (Linux only)	network host
7.5	None Network (fully isolated)	network none

Let me know when you're ready to move on to:

Level 8: Docker Volumes

Or ask if you'd like diagrams or files for these labs.

Awesome! 🞉

You're doing a great job progressing through Docker levels.

When you're ready to move on to:

Level 8: Docker Volumes

(just say "Next") and I'll give you labs just like before — hands-on and step-by-step.

If you ever want a recap of any previous level or a cheat sheet, just ask. 💆 🦾



Now moving to the next topic

8. Using Volumes

Anonymous, named, and host volumes

Persistent storage

docker volume create, inspect, rm now give me labs on this topic, like you gave me on last topic

Great! You're now on **Level 8**: **Using Volumes** — this is an important part of managing data in containers. Let's jump into labs with **hands-on examples** for:

Level 8 Labs: Using Volumes in Docker

Lab 1: Using Anonymous Volumes

Anonymous volumes are created without a specific name. They are managed by Docker and deleted with the container (unless persisted).

Steps:

1. Create a simple Dockerfile:

```
Dockerfile

# Dockerfile

FROM alpine
RUN mkdir /data
CMD ["sh", "-c", "echo Hello from container > /data/output.txt && sleep 5"]
```

2. Build and run with an anonymous volume:

```
docker build -t anon-volume .

docker run -v /data anon-volume
```

3. Check volumes:

```
bash

docker volume ls
```

4. Use docker inspect on the container to see where the volume is mounted.

Lab 2: Using Named Volumes



1. Run container using a named volume:

```
bash

docker volume create mydata

docker run -v mydata:/app/data alpine sh -c "echo Docker Rocks > /app/data/notes.txt"
```

2. Inspect the volume:

```
bash
docker volume inspect mydata
```

3. Re-run and check persistence:

```
bash
docker run -v mydata:/app/data alpine cat /app/data/notes.txt
```

Lab 3: Host Volumes (Bind Mounts)

Steps:

1. Create a file on host:

```
echo "Host file content" > hostfile.txt
```

2. Mount it inside a container:

bash
docker run -v \$(pwd)/hostfile.txt:/data/inside.txt alpine cat /data/inside.txt

Lab 4: Persistent Storage in Web Server

Simulate data persistence with a running container like nginx.

Steps:

1. Create a folder:

```
mkdir site-data
echo "<h1>Hello Volume</h1>" > site-data/index.html
```

2. Run nginx using bind mount:

```
bash

docker run -d -p 8080:80 -v $(pwd)/site-data:/usr/share/nginx/html nginx
```

- 3. Visit: http://localhost:8080
- **4.** Modify index.html and refresh browser to see live changes.

Lab 5: Volume Lifecycle

Commands to learn:

bash

Create a named volume
docker volume create myvol

Inspect volume details

```
docker volume inspect myvol

# Remove volume (only if not in use)
docker volume rm myvol

# List all volumes
docker volume ls

# Remove all unused volumes
docker volume prune
```

Summary:

Туре	Command Example	Use Case
Anonymous	-v /data	Temporary data
Named	-v mydata:/app/data	Reuse across containers
Host (Bind)	<pre>-v \$(pwd)/file:/container/path</pre>	Direct host-container access

Say "Next" when you're ready to move to

Level 9: Docker Compose Basics

and I'll give you step-by-step labs for that too.

Now moving to the next topic 9. Docker Compose (Intro) docker-compose.yml structure

Running multi-container apps

docker-compose up, down, logs now give me labs on this topic, like you gave me on last topic Excellent! You're now moving to **Level 9**: **Docker Compose (Intro)** — one of the most **powerful** tools in Docker for managing **multi-container** applications with simple YAML files.

Let's dive into hands-on labs that will help you master the core commands and concepts of Docker Compose.

Level 9 Labs: Docker Compose (Intro)

☑ Lab 1: Basic docker-compose.yml Structure

→ Goal:

Create a Docker Compose file that runs a Flask web server.

Steps:

1. Project structure:

2. app.py:

```
python

from flask import Flask
app = Flask(__name__)

@app.route("/")
def home():
    return "Hello from Docker Compose!"
```

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

3. requirements.txt:

```
nginx
flask
```

4. docker-compose.yml:

```
version: "3.8"

services:
    web:
    image: python:3.10
    container_name: flask-app
    ports:
        - "5000:5000"
    volumes:
        - .:/app
    working_dir: /app
    command: sh -c "pip install -r requirements.txt && python app.py"
```

5. Run the project:

```
bash

docker-compose up
```

6. Visit http://localhost:5000

✓ Lab 2: Run a Multi-Container App (Flask + Redis)

√ Goal:

Connect a Flask app to a Redis container using Docker Compose.

Structure:

app.py:

```
python

from flask import Flask
import redis

app = Flask(__name__)
    cache = redis.Redis(host='redis', port=6379)

@app.route("/")
    def hello():
        count = cache.incr('hits')
        return f'Hello Docker! This page has been viewed {count} times.'

if __name__ == "__main__":
        app.run(host='0.0.0.0', port=5000)
```

requirements.txt:

```
nginx
flask
redis
```

docker-compose.yml:

```
yaml
version: '3.8'
```

```
services:
    web:
    build: .
    ports:
        - "5000:5000"
    volumes:
        - .:/app
    depends_on:
        - redis
        working_dir: /app
    command: sh -c "pip install -r requirements.txt && python app.py"

redis:
    image: redis:alpine
```

Dockerfile:

```
Dockerfile

FROM python:3.10

WORKDIR /app

COPY . .
```

Run:

```
bash

docker-compose up
```

Stop:

```
bash
docker-compose down
```

Lab 3: View Logs

→ Goal:

Check logs from your containers.

Steps:

```
# Show all logs
docker-compose logs

# Follow logs
docker-compose logs -f

# View logs for a specific service
docker-compose logs web
```

☑ Lab 4: Detached Mode

Run the app in the background:

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

Stop it:

bash

docker-compose down

Summary of Compose Commands:

Command	Purpose		
docker-compose up	Start containers		
docker-compose up -d	Start in detached mode		
docker-compose down	Stop and remove containers/networks		
docker-compose logs	Show logs from all services		

Command	Purpose
docker-compose logs -f	Tail logs (like tail -f)
docker-compose build	Build the images defined in services
docker-compose ps	List running services

When you're ready, say "Next"

to move on to:

10. Dockerfile Optimizations and Best Practices

and I'll give you labs for that too.

Now moving to the next topic

10. Advanced Dockerfile

ENTRYPOINT vs CMD

ARG vs ENV

LABEL for metadata

.dockerignore file now give me labs on this topic, like you gave me on last topic

Great! Level 10 dives deeper into Dockerfile best practices and advanced instructions that give you more control and cleaner images.

Level 10 Labs: Advanced Dockerfile Concepts

Lab 1: ENTRYPOINT vs CMD



Understand difference between ENTRYPOINT and CMD.

Dockerfile:

```
Dockerfile

FROM alpine:latest

ENTRYPOINT ["echo"]

CMD ["Hello from CMD!"]
```

Build & Run:

```
docker build -t entrypoint-cmd .
docker run entrypoint-cmd
```

Expected output:

```
csharp

Hello from CMD!
```

Try overriding CMD:

```
bash

docker run entrypoint-cmd "Overridden CMD"
```

Output:

objectivec

Overridden CMD

Now override ENTRYPOINT:

```
bash

docker run --entrypoint ls entrypoint-cmd -l
```

Output: Lists directory contents, because **ENTRYPOINT** is overridden.

Summary:

- ENTRYPOINT is the executable; rarely overridden by default.
- CMD provides default arguments to ENTRYPOINT, can be overridden when running.

Lab 2: ARG vs ENV

→ Goal:

Learn build-time vs runtime variables.

Dockerfile:

```
Prockerfile

FROM alpine:latest

ARG my_arg=default_arg
ENV my_env=default_env

RUN echo "Build-time ARG: $my_arg"
CMD echo "Runtime ENV: $my_env"
```

Build:

bash

docker build --build-arg my_arg=passed_arg -t arg-env-demo .

Run:

bash

docker run arg-env-demo

Output:

yaml

Build-time ARG: passed_arg
Runtime ENV: default_env

Override ENV at runtime:

bash

docker run -e my_env=runtime_env arg-env-demo

Output:

yaml

Runtime ENV: runtime_env

Summary:

- ARG available only during build.
- ENV available during build and runtime.

Lab 3: LABEL for Metadata

→ Goal:

Add metadata to your images.

Dockerfile:

```
FROM alpine:latest

LABEL maintainer="your.email@example.com"

LABEL version="1.0"

LABEL description="Sample image with labels"
```

Build:

```
bash
docker build -t label-demo .
```

Inspect:

```
bash

docker inspect label-demo
```

Look for "Labels" section with your metadata.

Lab 4: Using .dockerignore

→ Goal:

Learn to exclude files/folders during build.

1. Create .dockerignore:

```
*.log
node_modules
secret.txt
```

- 2. Add these files/folders to your project.
- 3. Build an image copying all files:

```
Dockerfile

FROM alpine:latest

COPY . /app
```

4. Build:

```
bash

docker build -t dockerignore-demo .
```

Files/folders in .dockerignore won't be sent to the Docker daemon, making build faster and avoiding secrets in images.

Summary of Advanced Dockerfile Concepts

Instruction	Use Case
ENTRYPOINT	Define main executable, usually fixed
CMD	Provide default arguments for ENTRYPOINT
ARG	Pass variables during build time
ENV	Set environment variables in container

Instruction	Use Case
LABEL	Add metadata (author, version, etc.)
.dockerignore	Exclude files/folders from build context

When ready for the next topic, just say Next!

Now moving to the next topic

11. Multi-Stage Builds
 Reduce image size

Separate build and runtime steps

now give me labs on this topic, like you gave me on last topic

Sure! Multi-stage builds are super helpful for creating small, efficient Docker images by separating the build environment from the runtime environment.

Level 11 Labs: Multi-Stage Builds

☑ Lab 1: Simple Multi-Stage Build

→ Goal:

Build a Go app using multi-stage build to keep image size small.

Create a simple Go app main.go:

go

```
package main

import "fmt"

func main() {
   fmt.Println("Hello from multi-stage build!")
}
```

Dockerfile:

```
# Stage 1: Build
FROM golang:alpine AS builder

WORKDIR /app

COPY main.go .

RUN go build -o hello

# Stage 2: Runtime
FROM alpine:latest

WORKDIR /app

COPY --from=builder /app/hello .

CMD ["./hello"]
```

Build & run:

```
docker build -t multi-stage-go .
docker run --rm multi-stage-go
```

Output:

```
csharp

Hello from multi-stage build!
```

Check image size:

```
bash
docker images multi-stage-go
```

Notice how the final image is small, as it only contains the compiled binary and not the whole Go toolchain.

☑ Lab 2: Multi-Stage Build with Node.js

→ Goal:

Build a React app in one stage and serve it with Nginx in final stage.

Dockerfile:

```
# Stage 1: Build
FROM node:16-alpine AS builder

WORKDIR /app

COPY package.json package-lock.json ./
RUN npm install

COPY . .

RUN npm run build
```

```
# Stage 2: Serve
FROM nginx:alpine

COPY --from=builder /app/build /usr/share/nginx/html

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

Steps:

- Place your React app files here.
- Run:

```
docker build -t react-multi-stage .
docker run -p 8080:80 react-multi-stage
```

Open http://localhost:8080 to see the app served by Nginx.

☑ Lab 3: Compare single-stage vs multi-stage

→ Goal:

Build a Dockerfile that installs build tools and runs an app in one stage vs multi-stage, then compare image sizes.

Single-stage example (big image):

Dockerfile

FROM node:16-alpine

```
WORKDIR /app

COPY package.json package-lock.json ./

RUN npm install

COPY . .

RUN npm run build

CMD ["npm", "start"]
```

Multi-stage: Use the previous multi-stage example.

Build and compare:

```
docker build -t single-stage-app -f Dockerfile.single .

docker build -t multi-stage-app -f Dockerfile.multi .

docker images | grep app
```

Summary

- Multi-stage builds allow you to build your app with all dependencies in one stage.
- Then **copy only** the necessary artifacts (binaries, build output) to a smaller, cleaner image.
- This reduces image size, improves security (no build tools in runtime), and speeds up deployment.

When you want the next topic, just say Next!

Now moving to the next topic

12. Container Logs and Debugging docker logs
docker exec, docker inspect

docker top, docker statsnow give me labs on this topic, like you gave me on last topic

Great! You're moving into a very important section — **observability and debugging** in Docker. This helps you monitor containers, troubleshoot issues, and understand container behavior in real-time.

Level 12 Labs: Container Logs and Debugging

✓ Lab 1: View Container Logs

→ Goal:

Use docker logs to inspect output from a container.

Step 1: Create a simple logging container

Create log-app.py:

```
import time

i = 1
while True:
    print(f"Log line {i}")
    i += 1
    time.sleep(2)
```

Dockerfile:

```
Dockerfile

FROM python:3.9-slim

COPY log-app.py .
```

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

Build & Run:

```
docker build -t log-app .
docker run --name test-logs log-app
```

Check logs in another terminal:

```
bash
docker logs -f test-logs
```

Use Ctrl + C to stop following logs.

Lab 2: Executing Commands in a Running Container

→ Goal:

Use docker exec to run commands inside a live container.

Run an Alpine container:

```
bash

docker run -dit --name alpine-test alpine sh
```

Open a shell inside it:

```
bash

docker exec -it alpine-test sh
```

Try commands like:

bash

```
ls
apk add curl
curl google.com
```

Lab 3: Inspect Container Metadata



Use docker inspect to view container details like IP, mounts, env vars.

Run a container:

```
bash

docker run -d --name inspect-me nginx
```

Inspect it:

```
bash

docker inspect inspect-me
```

Look at:

- NetworkSettings.IPAddress
- Mounts
- Config.Env

Tip to format:

```
bash

docker inspect -f '{{.NetworkSettings.IPAddress}}' inspect-me
```

■ Lab 4: Monitor Container Processes with docker top



View running processes inside a container.

Run a container:

bash

docker run -d --name top-test nginx

Check processes:

bash

docker top top-test

■ Lab 5: Monitor Container Resources with docker stats

→ Goal:

Monitor CPU, memory, and network usage.

bash

docker run -d --name stats-test nginx
docker stats stats-test

This gives a live view of CPU %, memory usage, etc.

You can also use:

bash

docker stats

To monitor all running containers.

Summary

Command	Use Case
docker logs	See stdout/stderr from container
docker exec	Run commands inside live container
docker inspect	View detailed container metadata
docker top	List container's running processes
docker stats	Monitor live container resource usage

When you're ready for the next topic, just say Next!

Now moving to the next topic

13. Docker Compose (Intermediate)

Defining networks and volumes

depends_on, build, context

Override files and environment variable substitution

now give me labs on this topic, like you gave me on last topic

Great! You're progressing steadily into **intermediate Docker Compose** — which is crucial for managing multi-container apps efficiently.

Level 13 Labs: Docker Compose (Intermediate)

We'll now go deeper with Docker Compose: adding networks, shared volumes, build contexts, dependency ordering, and environment substitution.

✓ Lab 1: Define Custom Networks & Volumes

→ Goal: Use custom networks and volumes between multiple services.

Directory structure:

backend/app.py:

```
python

from flask import Flask
app = Flask(__name__)

@app.route('/')
def home():
    return "Backend is running!"

app.run(host='0.0.0.0', port=5000)
```

frontend/index.html:

```
html
```

docker-compose.yml:

```
yaml
version: "3.8"
services:
  backend:
    build: ./backend
    networks:
      - app-net
  frontend:
    image: nginx:alpine
    volumes:
      - ./frontend:/usr/share/nginx/html
    ports:
      - "8080:80"
    networks:
      - app-net
networks:
  app-net:
    driver: bridge
```

Build & Run:

```
bash

docker-compose up --build
```

Visit:

- http://localhost:8080 → Frontend
- (You can curl backend:5000 from inside frontend container to verify internal network)

```
✓ Lab 2: Use depends_on, build, and context
```

→ Goal: Control service build and startup order using depends_on , and set custom build context.

Folder:

api/app.py:

```
python
print("API is starting...")
```

api/Dockerfile:

```
Prockerfile

FROM python:3.9-slim

COPY app.py .

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

docker-compose.yml:

```
yaml

version: "3.8"

services:
app:
build:
```

```
context: ./api
depends_on:
    - db

db:
    image: postgres
    environment:
    POSTGRES_PASSWORD: example
```

Run:

```
bash

docker-compose up --build
```

You'll see db container start before the app.

☑ Lab 3: Environment Variable Substitution

Goal: Use .env variables in Compose.

Structure:

```
env-substitution-lab/

— docker-compose.yml
— .env
```

.env:

```
ini
WEB_PORT=8088
```

docker-compose.yml:

```
yaml

version: "3.8"

services:
    web:
    image: nginx
    ports:
    - "${WEB_PORT}:80"
```

Run:

```
bash

docker-compose up
```

Check on http://localhost:8088.

Lab 4: Compose Override Files

Goal: Use docker-compose.override.yml to apply dev overrides.

Structure:

docker-compose.yml:

```
yaml
```

```
version: "3.8"

services:
  web:
  image: nginx
```

docker-compose.override.yml:

```
yaml

version: "3.8"

services:
    web:
    ports:
    - "8089:80"
```

Run:

```
bash

docker-compose up
```

This will apply the override and expose NGINX at http://localhost:8089.

Summary

Concept	Description		
networks:	Define custom bridge networks		
volumes:	Share data between host & containers		
depends_on:	Control startup order		
build/context:	Specify build source folders		
.env file	Inject dynamic values like ports and secrets		
override.yml	Use for development/local-specific overrides		

When you're ready, say **Next topic** to continue to • **14. Docker Volumes and Bind Mounts** (Advanced) or whichever is next in your list.

Now moving to the next topic
14. Networking Deep Dive
User-defined bridge networks

Overlay network (Swarm-related)

Container DNS resolution

now give me labs on this topic, like you gave me on last topic

Excellent! You're now moving into a **deeper networking layer** of Docker — crucial for building production-ready distributed apps.

Level 14 Labs: Networking Deep Dive

Topics:

- User-defined bridge networks
- Overlay networks (Swarm)
- Container DNS resolution

Let's explore these step-by-step.

Lab 1: User-defined Bridge Networks & DNS Resolution

Goal: Understand benefits of user-defined bridge networks and how Docker provides DNS-based service discovery.

Directory structure:

```
sql
user-bridge-lab/
├── docker-compose.yml
```

docker-compose.yml:

```
yaml
version: "3.8"
services:
  backend:
    image: python:3.9-slim
    command: ["python3", "-m", "http.server", "5000"]
    networks:
      - custom-net
  frontend:
    image: curlimages/curl
    command: ["sh", "-c", "sleep 5 && curl backend:5000"]
    networks:
      - custom-net
networks:
  custom-net:
    driver: bridge
```

Run:

```
bash

docker-compose up
```

You'll see frontend resolves backend using Docker-internal DNS (via backend service name).

Try changing the network to default and see if name resolution fails.

✓ Lab 2: Manual Network Creation and Use

© Goal: Create a user-defined bridge network manually and attach containers.

```
# Step 1: Create a custom bridge network
docker network create my_bridge_net

# Step 2: Run 2 containers in it
docker run -dit --name backend --network my_bridge_net python:3.9-slim python3 -m
http.server 5000
docker run --rm --network my_bridge_net curlimages/curl curl backend:5000
```

The second container can **resolve** backend thanks to the DNS provided by Docker's user-defined bridge.

☑ Lab 3: Overlay Networks with Swarm Mode

- Overlay networks require Docker Swarm to be initialized.
- **6 Goal:** Setup overlay network in Swarm and connect services.

Step 1: Init Swarm (if not already)

bash

docker swarm init

Step 2: Create overlay network

```
bash
docker network create -d overlay my_overlay_net
```

Step 3: Deploy services using that network

```
bash

docker service create --name backend --network my_overlay_net python:3.9-slim python3
-m http.server 5000

docker service create --name frontend --network my_overlay_net curlimages/curl sh -c
"sleep 5 && curl backend:5000"
```

Services communicate over the overlay network, even across nodes (in multi-host Swarm clusters).

Lab 4: Inspecting Container Networking

Goal: Understand internal details of Docker networks.

```
# List networks

docker network ls

# Inspect a network (e.g., bridge)

docker network inspect bridge

# Inspect a custom network

docker network inspect my_bridge_net
```

Q Look at:

- Connected containers
- Subnet and gateway
- Internal DNS info

Summary

Concept	Description		
User-defined bridge	Enables DNS resolution by service/container name		
Overlay network	Used in Docker Swarm for multi-host communication		
Container DNS	Docker provides internal DNS in user-defined networks		
docker network inspect	View subnet, connected containers, and metadata		

Let me know when you're ready to move to the next topic:

◆ 15. Docker Security Basics or whichever comes next in your path!

Now moving to the next topic

15. Docker Volumes (Advanced)

Bind mounts vs volumes

Backup and restore volume data

Using volumes in docker-compose

now give me labs on this topic, like you gave me on last topic

Awesome, you're diving into **Advanced Docker Volumes** now — critical for managing persistent, reusable, and sharable data across containers.

Level 15 Labs: Docker Volumes (Advanced)

Topics:

- Bind mounts vs named volumes
- Backup & restore volume data
- Using volumes in docker-compose

✓ Lab 1: Bind Mount vs Named Volume

Goal:

Understand how bind mounts and volumes work and their differences.

Create directory and test file:

```
mkdir -p bind-mount-lab
echo "Hello from bind mount!" > bind-mount-lab/hello.txt
```

Run with Bind Mount:

```
bash
docker run --rm -v $(pwd)/bind-mount-lab:/data alpine cat /data/hello.txt
```

Output:

```
bash
Hello from bind mount!
```

* Note: You're directly mounting a host folder.

Run with Named Volume:

```
docker volume create my_named_volume
docker run --rm -v my_named_volume:/data alpine sh -c "echo 'Hello from named volume!'
> /data/msg.txt"
docker run --rm -v my_named_volume:/data alpine cat /data/msg.txt
```

Output:

```
csharp
Hello from named volume!
```

Unlike bind mounts, named volumes are fully managed by Docker and not tied to a host path.

☑ Lab 2: Backup & Restore Docker Volume

Goal:

Backup and restore a named volume's content.

Create volume and data:

```
docker volume create backup_test
docker run --rm -v backup_test:/data alpine sh -c "echo 'Backup this!' >
/data/info.txt"
```

Backup volume to a tarball:

bash

```
docker run --rm -v backup_test:/volume -v $(pwd):/backup alpine \
  tar czf /backup/volume-backup.tar.gz -C /volume .
```

Check volume-backup.tar.gz in your host directory.

Restore to a new volume:

```
docker volume create restore_test
docker run --rm -v restore_test:/volume -v $(pwd):/backup alpine \
    sh -c "cd /volume && tar xzf /backup/volume-backup.tar.gz"

docker run --rm -v restore_test:/data alpine cat /data/info.txt
```

Output:

```
kotlin
Backup this!
```

Lab 3: Using Volumes in Docker Compose

Project structure:

data/hello.txt;

csharp

Hello from Compose volume!

docker-compose.yml:

```
version: "3.8"

services:
   app:
   image: alpine
   command: cat /data/hello.txt
   volumes:
      - ./data:/data
```

Run:

docker-compose up

Output:

csharp

Hello from Compose volume!

Try changing it to use a named volume:

```
yaml

services:
app:
image: alpine
```

Summary Table

Туре	Mount Path	Lives on Host?	Persistent?	Use Case
Bind Mount	Host path	Yes	Yes	Dev/testing, live editing
Named Volume	Docker- managed	Internally yes	Yes	Production data, safe backups
Anonymous Volume	Auto-generated	Yes	Yes	Temporary data, intermediate

Let me know when you're ready to move to:

• 16. Docker Security Basics

(Security options, rootless containers, user management) 🚀

Now moving to the next topic

• 16. Image Optimization Smaller base images (alpine)

Reducing layers and caching

Cleaning up unused artifacts now give me labs on this topic, like you gave me on last topic

Perfect! You're now moving into • Level 16: Image Optimization — essential for building fast, secure, and efficient Docker images.

Level 16 Labs: Image Optimization

Topics:

- Using smaller base images like Alpine
- Reducing layers & leveraging caching
- Cleaning up build artifacts

✓ Lab 1: Using Smaller Base Images (Alpine)

Goal:

Compare image sizes between ubuntu and alpine.

Dockerfile.ubuntu:

```
dockerfile

FROM ubuntu:latest

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

Dockerfile.alpine:

```
dockerfile

FROM alpine:latest

RUN apk add --no-cache curl
```

Build both:

```
docker build -f Dockerfile.ubuntu -t ubuntu-curl .
docker build -f Dockerfile.alpine -t alpine-curl .
```

Compare sizes:

```
bash

docker images | grep curl
```

You'll see that Alpine is significantly smaller (typically ~5MB vs ~100MB+).

Lab 2: Reducing Layers & Leveraging Caching

Goal:

Build an optimized image by combining RUN steps and using Docker cache.

bad.Dockerfile (inefficient):

```
FROM node:18

RUN apt-get update

RUN apt-get install -y git

RUN npm install -g http-server
```

good.Dockerfile (optimized):

```
dockerfile

FROM node:18

RUN apt-get update && \
```

```
apt-get install -y git && \
npm install -g http-server && \
apt-get clean && rm -rf /var/lib/apt/lists/*
```

Build both:

```
docker build -f bad.Dockerfile -t bad-image .
docker build -f good.Dockerfile -t good-image .
```

Compare layer count:

```
docker history bad-image
docker history good-image
```

You'll see fewer layers in good-image.

Lab 3: Clean Up Unused Artifacts

Goal:

Use multi-stage builds and cleanup to reduce image size.

Dockerfile.cleanup:

```
dockerfile

FROM node:18 AS builder

WORKDIR /app

COPY . .
```

```
RUN npm install

FROM node:18-slim

WORKDIR /app

COPY --from=builder /app .

CMD ["node", "app.js"]
```

☑ This avoids carrying over node_modules, build cache, and temp files into final image.

Bonus Tip: Minimize Final Image

You can also add this in Alpine or Debian:

```
dockerfile

RUN rm -rf /tmp/* /var/tmp/* /root/.npm /root/.cache
```

Summary Tips

Optimization Strategy	Benefit
Use Alpine/scratch image	Minimize base image size
Combine RUN statements	Reduce image layers
Use .dockerignore	Avoid sending unneeded files
Clean up cache/temp files	Avoid leftover build dependencies
Multi-stage builds	Only keep runtime dependencies

Let me know when you're ready to move to:

• 17. Docker Security Basics

(Non-root users, image scanning, seccomp, capabilities)

Now moving to the next topic

17. Docker Registry

Docker Hub

Private registries

Pushing/pulling images (docker login, tag, push) now give me labs on this topic, like you gave me on last topic

Awesome! Let's tackle • Level 17: Docker Registry — understanding how to push/pull images, use Docker Hub, and work with private registries.

Level 17 Labs: Docker Registry

Topics:

- Docker Hub basics
- Private registry setup
- docker login, tag, push, pull

Lab 1: Using Docker Hub

Goal:

Push and pull a custom image to/from Docker Hub.

Steps:

1. Create a Docker Hub account

Go to https://hub.docker.com and sign up/log in.

2. Log in from terminal:

```
bash

docker login
```

Enter your Docker Hub username & password.

3. Create a simple Dockerfile:

```
dockerfile

# Dockerfile

FROM alpine

CMD ["echo", "Hello from Docker Hub!"]
```

4. Build your image:

```
bash

docker build -t myimage .
```

5. Tag the image:

```
bash

docker tag myimage <your-dockerhub-username>/myimage:latest
```

6. Push it to Docker Hub:

```
bash

docker push <your-dockerhub-username>/myimage:latest
```

7. Verify on Docker Hub UI, and then:

bash

docker rmi <your-dockerhub-username>/myimage:latest
docker pull <your-dockerhub-username>/myimage:latest

Lab 2: Using a Local Private Registry

Goal:

Run a private registry locally and push an image to it.

Start a private registry:

docker run -d -p 5000:5000 --name registry registry:2

Build and tag image for local registry:

docker build -t myimage .

docker tag myimage localhost:5000/myimage:latest

1 Push to the local registry:

docker push localhost:5000/myimage:latest

- 🦴 If you get insecure registry errors:
- Add this to Docker daemon config (/etc/docker/daemon.json):

json

```
{
   "insecure-registries" : ["localhost:5000"]
}
```

Restart Docker:

```
bash
sudo systemctl restart docker
```

U Pull from local registry:

```
docker rmi localhost:5000/myimage:latest
docker pull localhost:5000/myimage:latest
```

✓ Lab 3: Browse & Inspect Registries

Goal:

List pushed images and interact with registry API.

List images (via curl):

```
curl http://localhost:5000/v2/_catalog
```

List tags for myimage:

bash

curl http://localhost:5000/v2/myimage/tags/list

Summary

Command	Description
docker login	Authenticate with Docker Hub/registry
docker tag	Tag image for registry
docker push	Push image to registry
docker pull	Pull image from registry
registry:2 image	Official Docker registry image

Let me know when you're ready to move on to:

18. Docker Security Basics

(non-root containers, image scanning, seccomp, capabilities)

Now moving to the next topic

• 18. Health Checks
HEALTHCHECK in Dockerfile

Checking app status inside a container

now give me labs on this topic, like you gave me on last topic

Great! Let's move into • Level 18: Docker Health Checks, which ensures that your containerized applications are functioning correctly even after startup.

Level 18 Labs: Health Checks

What You'll Learn:

- How to use HEALTHCHECK in a Dockerfile
- How Docker monitors container health
- How to view container health status
- How to troubleshoot failed health checks

☑ Lab 1: Basic HEALTHCHECK in Dockerfile

Goal:

Use a HEALTHCHECK instruction in a Dockerfile and observe health status changes.

Step-by-Step:

1. Create a simple Dockerfile:

```
# Dockerfile

# Dockerfile

FROM nginx:alpine

HEALTHCHECK --interval=10s --timeout=3s \
CMD wget --no-verbose --spider http://localhost || exit 1
```

2. Build the image:

bash

docker build -t nginx-healthcheck .

3. Run the container:

```
bash

docker run -d --name nginx-container nginx-healthcheck
```

4. Check health status:

```
bash

docker inspect --format='{{json .State.Health}}' nginx-container | jq
```

Or:

```
bash
docker ps
```

You'll see the "healthy" or "unhealthy" status under the STATUS column.

Lab 2: Simulate Failing Health Check

Goal:

See how Docker reacts when the health check fails repeatedly.

1. Edit Dockerfile for failure simulation:

```
Dockerfile

# Dockerfile

FROM alpine
```

```
RUN apk add --no-cache curl

HEALTHCHECK --interval=5s --timeout=2s --retries=3 \
   CMD curl --fail http://localhost:9999 || exit 1

CMD ["sleep", "3600"]
```

2. Build & run:

```
docker build -t fail-healthcheck .

docker run -d --name unhealthy-container fail-healthcheck
```

3. Check status after ~20s:

```
bash

docker inspect --format='{{.State.Health.Status}}' unhealthy-container
```

You should now see "unhealthy".

Lab 3: Log Health Status Events

Goal:

Track health state transitions in logs.

1. Use Docker events to monitor container health:

```
docker events --filter container=unhealthy-container
```

2. In another terminal, restart or cause failure and see the event logs update.

☑ Bonus: Use docker-compose with Healthcheck

```
version: "3.8"

services:
    web:
    image: nginx:alpine
    ports:
        - "8080:80"
    healthcheck:
        test: ["CMD", "wget", "--spider", "-q", "http://localhost"]
        interval: 10s
        timeout: 3s
        retries: 3
```

Run:

```
bash

docker-compose up -d

docker inspect <container_id> | grep Health
```

★ Summary

Directive	Purpose
HEALTHCHECK	Defines how Docker tests if a container is healthy
CMD	Command to run inside container for health check
interval	How often to run the check
timeout	Time allowed for the check

Directive	Purpose
retries	Number of failures before being marked unhealthy

Let me know when you're ready for:

- 19. Docker Security Basics
- 🔐 (non-root containers, image scanning, seccomp, capabilities)

Now moving to the next topic

19. Security Best Practices
 Avoid running as root inside containers

Scan images for CVEs (docker scout cves)

Minimize image surface

now give me labs on this topic, like you gave me on last topic

Awesome! Let's move on to • Level 19: Security Best Practices — an essential topic for running containers safely in production environments.

Level 19 Labs: Security Best Practices

What You'll Learn:

- How to avoid running containers as root
- How to scan images for vulnerabilities (CVEs)
- How to minimize the image surface and attack area

☑ Lab 1: Avoid Running as Root



Modify a Dockerfile to run the application as a **non-root** user.

Example Dockerfile (unsafe):

```
dockerfile

FROM node:18-alpine

WORKDIR /app
COPY . .
RUN npm install

CMD ["node", "index.js"]
```

This runs as **root** by default (bad practice \times).

☑ Updated Secure Dockerfile:

```
dockerfile

FROM node:18-alpine

# Create app directory and non-root user

RUN addgroup -S appgroup && adduser -S appuser -G appgroup

WORKDIR /app

COPY . .

RUN npm install

# Switch to non-root user
```

```
USER appuser

CMD ["node", "index.js"]
```

Q Check user inside container:

```
docker build -t secure-node-app .
docker run -it secure-node-app sh
whoami
```

You should see appuser <

☑ Lab 2: Scan Images for CVEs Using Docker Scout

Goal:

Scan your Docker images for known vulnerabilities.

1. Install Docker Scout (comes with Docker Desktop CLI v4.18+):

bash
docker scout version

If installed, proceed.

2. Scan an official image:

bash

docker scout cves node:18-alpine

3. Scan your custom image:

bash

docker scout cves secure-node-app

4. Interpret Results:

- Critical, High, Medium CVEs will be listed
- Check if there's a newer base image version

Lab 3: Minimize Image Surface (Alpine & Multi-Stage Builds)

Goal:

Create minimal images with no unnecessary packages, making them smaller and more secure.

1. Use Alpine base image:

dockerfile

FROM node:18-alpine

Removes ~100MB of unused layers compared to node:18

2. Use multi-stage builds (example with Go):

dockerfile

```
# Stage 1: Builder
FROM golang:1.20 AS builder
WORKDIR /src
COPY . .
RUN go build -o app .

# Stage 2: Minimal image
FROM alpine
RUN adduser -D appuser
USER appuser
WORKDIR /app
COPY --from=builder /src/app .
CMD ["./app"]
```

3. Build & check size:

```
docker build -t minimal-go-app .

docker image ls
```

Result: A small, non-root, production-safe image

☑ Bonus Tip: Add .dockerignore

Avoid copying unnecessary files like .git , node_modules , etc.

```
dockerignore

.git
node_modules
Dockerfile
README.md
```



Best Practice	Why it matters
Run as non-root user	Reduces attack surface
Use Alpine or minimal base images	Smaller, less vulnerable images
Use Docker Scout	Detect and fix known vulnerabilities
Avoid copying secrets	Prevent credentials leakage
Use .dockerignore	Avoid bloated images

Let me know when you're ready for:

20. Docker Swarm Intro

(init, join, services, scaling, stack)

Now moving to the next topic

20. Docker Contexts

Switching between local and remote Docker engines

now give me labs on this topic, like you gave me on last topic

Great! Let's dive into • Level 20: Docker Contexts — essential when you're managing multiple Docker environments like local, remote servers, or cloud instances.

Level 20 Labs: Docker Contexts

What You'll Learn:

- What Docker contexts are
- How to create, list, inspect, and switch contexts
- How to connect to remote Docker Engines securely



Lab 1: Understand Docker Contexts



What is a Context?

A context is a set of parameters (host, TLS credentials, etc.) used by the Docker CLI to communicate with a local or remote Docker daemon.

Step-by-step:

1. List available contexts:

```
bash
docker context ls
```

You'll see something like:

```
срр
NAME
           DESCRIPTION
                                                      DOCKER ENDPOINT
default *
           Current DOCKER_HOST based configuration
                                                     unix:///var/run/docker.sock
```

The * shows the currently active context.

2. Inspect current context:

```
bash
docker context inspect default
```

This shows connection settings like host, TLS, etc.



Lab 2: Create a New Remote Context (SSH-based)



Goal:

Connect to a remote server via SSH and manage Docker there from your local CLI.

Prerequisites:

- You have SSH access to a remote server
- Docker is installed and running on that server
- Your public key is authorized on the remote server

Steps:

1. Create a new context via SSH:

```
bash
docker context create myremote \
  --docker "host=ssh://user@remote-server-ip"
```

This uses ssh as a transport layer.

2. Verify it was created:

```
bash
docker context ls
```

You'll now see:

nginx

```
myremote ssh://user@remote-server-ip ...
```

3. Switch to the new context:

```
bash

docker context use myremote
```

✓ All Docker CLI commands now run on the remote server!

Try this:

```
bash

docker info
```

You'll get remote Docker daemon info.

4. Switch back to local:

docker context use default

Lab 3: Remove and Manage Contexts

***** Remove a context:

bash

docker context rm myremote

Rename a context:

bash

docker context rename oldname newname

Inspect any context:

bash

docker context inspect myremote

☑ Bonus: Use Contexts in docker-compose

If you're using docker-compose, it respects the current context.

docker context use myremote
docker-compose up -d

Runs the compose stack on the remote host!

★ Summary

Command	Purpose
docker context ls	List all contexts
docker context create	Create new context
docker context use	Switch between contexts

Purpose
View context details
Remove a context

Let me know when you're ready for:

- 21. Docker Swarm (Intro)
- (init, join, services, scaling, stacks, overlay)