🐳 Complete Docker Guide — Volumes, Run, Networking (Well-Formatted Edition)

1️⃣ What Are Docker Volumes?

A **volume** is a persistent storage mechanism managed by Docker.

Containers are ephemeral (temporary). When a container is removed, its internal filesystem is r
too.

Volumes solve this problem.

They allow:

- Data persistence
- Sharing data between containers
- Mounting host folders into containers

🔹 Types of Storage in Docker

1 . Anonymous Volume

```
docker run -v /data my-app
```

Docker creates a random volume.

---

2 . Named Volume

```
docker volume create my-volume

docker run -v my-volume:/app/data my-app
```

Docker manages storage in:

```
/var/lib/docker/volumes/
```

---

3 . Bind Mount (Host Folder Mount)

```
docker run -v /home/user/project:/workspace my-app
```

This connects a host folder directly into the container.

📦 Clear Architecture Diagram (Bind Mount)

```
+--------------------------------------------------+
|                     HOST                         |
|                                                  |
|   /home/user/project                             |
|          │                                        |
|          │ (Bind Mount)                           |
|          ▼                                        |
|   +-------------------------------+               |
|   |        Docker Engine          |               |
|   |                               |               |
|   |   +-----------------------+   |               |
|   |   |       Container       |   |               |
|   |   |                       |   |               |
|   |   |     /workspace        |   |               |
|   |   |                       |   |               |
|   |   +-----------------------+   |               |
|   +-------------------------------+               |
+--------------------------------------------------+
```

🔥 What This Means

- `/home/user/project` exists on your computer
- `/workspace` exists inside the container
- They point to the SAME physical data
- If you edit file on host → container sees it instantly
- If container modifies file → host sees it instantly

Both see the same data.

2️⃣ How `docker run my-app node server.js` Works (No docker exec)

When you run:

```
docker run my-app node server.js
```

Docker performs these steps internally:

Step 1 : Image Lookup

Docker checks if `my-app` image exists locally.

If not → it pulls from Docker Hub.

Step 2 : Container Creation

Docker creates a new isolated container layer on top of the image.

Each container has:

- Its own filesystem
- Its own process namespace
- Its own network namespace
- Its own PID 1 process

Step 3 : Filesystem Mounting

Docker:

- Mounts image layers (read-only)
- Adds writable layer on top
- Applies bind mounts or volumes

Step 4 : Process Execution

Docker starts:

```
node server.js
```

as PID 1 inside container.

This is NOT using `docker exec`.

`docker exec` is only used to enter an already running container.

3️⃣ Docker Networking (Full Explanation)

Every container gets its own network namespace.

By default, Docker creates a bridge network called:

```
bridge
```

🔹 Default Bridge Architecture

```
                 HOST MACHINE

                192.168.1.10
                      |
                      |
              +----------------+
              |  docker0       |  (Virtual Bridge)
              |  172.17.0.1    |  (Gateway)
              +----------------+
                  |        |
          ---------        ---------
          |                         |
   Container A                Container B
   172.17.0.2                 172.17.0.3
```

🔥 Important Concepts

docker `0`

A virtual network bridge created by Docker.

It acts like a router/switch.

Default Gateway

Inside every container:

```
Default Gateway = 172.17.0.1
```

That is docker `0` .

When container wants internet:

```
Container → docker0 → Host → Internet
```

4️⃣ How Containers Communicate

Case 1 : Default Bridge

Containers can communicate using IP addresses:

```
http://172.17.0.3:3000
```

BUT not by container name.

Case 2 : User-Defined Bridge (Recommended)

```
docker network create my-network

docker run --network my-network --name app1 my-app

docker run --network my-network --name db my-db
```

Now containers can communicate using names:

```
http://db:5432
```

Docker provides internal DNS automatically.

5️⃣ Port Mapping (Host ↔️ Container)

```
docker run -p 5000:3000 my-app
```

Means:

```
Host:5000 → Container:3000
```

Flow:

```
Browser → Host:5000 → Docker NAT → Container:3000
```

6️⃣ Full System Communication Diagram

```
              INTERNET
                   │
                   ▼
           Host (192.168.1.10)
                   │
             Port 5000
                   │
                   ▼
              docker0 (172.17.0.1)
                   │
        ┌──────────┴──────────┐
        ▼                     ▼
Container A               Container B
172.17.0.2                172.17.0.3
  app:3000                 db:5432
```

`docker network create my-network`

🔹 Syntax

```
docker network create [OPTIONS] NETWORK_NAME
```

In your case:

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

What this means:

- Create a **user-defined bridge network**
- Name it: `my-network`

🔥 What Actually Happens Internally

When you create this network, Docker:

1 . Creates a new virtual Linux bridge

Something like:

```
br-<random_id>
```

You can see it:

```
ip addr
```

Example:

```
br-7d83c9e2c1f4
```

This is similar to `docker0`, but isolated.

Assigns a new subnet

Docker automatically chooses a subnet like:

```
172.18.0.0/16
```

You can verify:

```
docker network inspect my-network
```

You'll see:

```
"Subnet": "172.18.0.0/16",
"Gateway": "172.18.0.1"
```

So now:

- Gateway = `172.18.0.1`
- Containers will get IPs like:
- `172.18.0.2`
- `172.18.0.3`

`docker run --network my-network --name app1 my-app`

🔹 Syntax

```
docker run
  --network <network_name>
  --name <container_name>
  <image_name>
```

Your command:

```
docker run --network my-network --name app1 my-app
```

This means:

- Create a container from image `my-app`
- Attach it to `my-network`
- Give container name: `app1`

🔥 What Happens Internally

Step 1 : Container Created

Docker:

- Creates container filesystem
- Sets up namespaces
- Creates writable layer

Step 2 : Network Namespace Created

Each container gets its own:

- IP stack
- Routing table
- Loopback interface

Step 3 : Docker Creates veth Pair

This is VERY important.

Docker creates something called a **veth pair**.

Think of it like a virtual ethernet cable.

```
Host Side         Container Side
---------         --------------
vethXYZ  <---->   eth0
```

- One end stays on host
- One end goes inside container

Step 4 : Connect to Bridge

The host side of veth connects to:

```
br-7d83c9e2c1f4
```

Now your container is plugged into the virtual switch.

Step 5 : Assign IP

Docker assigns:

```
172.18.0.2
```

Inside container:

```
eth0 → 172.18.0.2
Gateway → 172.18.0.1
```

```
docker run --network my-network --name db my-db
```

Same process happens.

This container now gets:

```
172.18.0.3
```

Both are on same virtual LAN.

How Naming Works

This is the most important part.

When you use:

```
--name app1
```

Docker registers this name in its internal DNS server.

Docker runs an embedded DNS server inside each user-defined network.

🔥 Internal DNS Magic

When `app1` tries to access:

```
http://db:5432
```

Inside container:

1. Container asks: “Who is db?”
2. Docker DNS server checks network registry.
3. Finds container named `db`.
4. Returns IP: `172.18.0.3`.

So:

```
db → 172.18.0.3
```

This ONLY works in **user-defined networks**.

It does NOT work in default `bridge`.

Full Communication Flow

```
         USER DEFINED NETWORK (my-network)

                 br-xxxxxx
               172.18.0.1 (gateway)
                     │
          ┌──────────┴──────────┐
          ▼                      ▼
     app1 container         db container
     172.18.0.2              172.18.0.3
```

When app 1 connects to db:

```
app1 → DNS lookup (db)
     → Docker DNS returns 172.18.0.3
     → Packet sent to 172.18.0.3
     → Bridge forwards to db container
```

No NAT needed (same subnet).

Default Gateway

Inside each container:

```
ip route
```

You'll see:

```
default via 172.18.0.1 dev eth0
```

If container wants internet:

```
Container → Gateway (bridge) → Host → Internet
```

Docker performs NAT at host level.

Why This Is Better Than Default Bridge

Default `bridge`:

- No automatic DNS
- Must use IP
- Less isolation

User-defined bridge:

- Automatic DNS
- Container name resolution

- Better isolation
- Cleaner architecture

This is how Docker Compose works internally.

What Happens When You Remove Container?

If you remove `db`:

```
docker rm db
```

Docker:

- Removes veth pair
- Removes DNS entry
- Frees IP

If you recreate `db`, it may get a new IP.

But name `db` will always work.

Important Production Detail

IP addresses are dynamic.

NEVER use IP like:

```
172.18.0.3
```

Always use:

```
db
```

Because DNS abstracts IP changes.

🔥 In One Sentence

When you create a user-defined network:

- Docker creates a new isolated virtual LAN
- Containers plugged into it get unique IPs

- Docker runs an internal DNS
- `--name` becomes hostname inside that network
- Containers communicate using names, not IPs

7️⃣ Summary

Volumes

Persistent storage mechanism for containers.

Bind Mount

Maps host folder into container.

docker run

Creates container and starts process directly.

docker exec

Enters running container.

docker0

Default bridge acting as gateway.

Default Gateway

Usually `172.17.0.1` inside containers.

User-Defined Network

Enables container name-based communication.

🚀 Final Understanding

Docker provides:

- Filesystem isolation
- Process isolation
- Network isolation
- Controlled resource usage
- Persistent storage via volumes

This is the foundation of containerized systems and platforms like mini-Replit architectures.

If you want next level:

- Linux namespaces deep dive
- cgroups internals
- Overlay 2 filesystem layers
- Kubernetes networking model

Tell me what you want to master next 🔥