🌐 Docker Networking A2Z – Masterclass for Developers & DevOps

---

📦 What is Docker Networking?

> 🧠 Docker networking allows **containers to communicate** with:

- Each other 📞
- The host machine 🖥️
- The external internet 🌍

Docker automatically creates networks and connects containers based on mode.

🧠 Key Terms

Term	Meaning	Emoji
Network	Virtual connection b/w containers	🛜
Bridge	Default, isolated internal network	🌉
Host	Shares host's network stack	🏠
None	No network access	🚫
Overlay	Cross-host communication (Swarm)	🕸️

🌉 Bridge Mode (Default)

> 🧱 **Bridge network** is like a **private switch** where containers talk to each other.

🛠️ Created Automatically:

```
docker network ls
```

Look for: `bridge`

🛠️ How it works:

- Containers get **private IPs** (like `172.17.0.x`)
- They **can access the internet** via NAT
- But **cannot be accessed from outside** without `-p` port mapping

---

🧪 Try it:

```
docker run -d --name container1 nginx
docker run -d --name container2 busybox sleep 9999

# Ping container1 from container2 by IP
docker exec -it container2 ping 172.17.0.x
```

❌ By default, they **can't talk by name** unless in **custom network**

🧱 Custom Bridge Network (Recommended)

> 🎯 Custom networks support **container name resolution (DNS)**!

📦 Create a custom bridge:

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

🚀 Launch containers into it:

```
docker run -d --name app1 --network my-network nginx
docker run -it --name app2 --network my-network busybox sh
```

Now, inside `app2`:

```
ping app1
```

☑️ Works! 🎉 Containers can **ping by name**!

☑️ Why Use Custom Bridge?

Feature	Benefit
🧠 DNS	Resolve container names
🔐 Isolation	Only containers in same network can talk
🔄 Flexibility	Use multiple networks
⚙️ Control	Inspect with `docker network inspect`

🏠 Host Network Mode

> Shares the **host's network stack** directly.

🚀 Use:

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

☑ Pros:

- 💧 Super fast — no NAT or port mapping
- 🩹 Useful for monitoring tools (Prometheus, Grafana)

❌ Cons:

- ⚠ **No isolation**
- 🧱 Cannot run 2 containers on same port!

🚫 None Network Mode

> Container has **no networking** at all.

```
docker run --network none busybox
```

- 🔒 Fully isolated
- Useful for **security testing** or **offline compute jobs**

🌐 Overlay Network (Advanced – Docker Swarm)

> ☸ Enables containers on **different hosts** to communicate

Use Case:

- Docker Swarm
- Distributed Microservices

```
docker network create --driver overlay my-overlay
```

Requires Swarm mode:

```
docker swarm init
```

Connect Containers to Multiple Networks

```
docker network create frontend
docker network create backend

docker run -d --name api \
  --network frontend \
  --network-alias api \
  nginx
```

Then attach to another network:

```
docker network connect backend api
```

Inspect a Network

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

Shows:

- Container list
- IPs
- Aliases
- Subnets

CLI Recap

Command	Purpose
`docker network ls`	List networks
`docker network create <name>`	Create a custom network
`docker run --network <name>`	Connect to specific network
`docker network inspect <name>`	Inspect config and members
`docker network rm <name>`	Delete a network

Command	Purpose
`docker network connect`	Connect running container
`docker network disconnect`	Disconnect container

Best Practices for Docker Networking

Practice	Why It's Great
☑ Use custom bridge networks	Enable name resolution + isolation
⛔ Avoid host mode unless needed	Can expose host stack
🧱 Use none mode for compute-only jobs	Maximum isolation
🔐 Limit network access	Avoid connecting everything together
🔍 Use `inspect` to debug IPs	Know who talks to whom
🧪 Use `busybox` or `alpine` to test ping	Lightweight network testing tools

Summary Table

Mode	Isolated?	Can Use DNS?	Host Access?	Notes
`bridge`	☑ Yes	❌ No (unless custom)	☑ via `-p`	Default
`custom bridge`	☑ Yes	☑ Yes	☑ via `-p`	Best for local
`host`	❌ No	☑ Yes	Direct	No port mapping
`none`	☑ Yes	❌ No	❌ No	For isolation
`overlay`	☑ Cross-host	☑ Yes	Swarm only	For multi-node

Final Analogy

> **Bridge network** = Private Wi-Fi router 🌐 **Custom bridge** = Guest Wi-Fi with name tags 🏷 **Host mode** = Ethernet cable directly into host 💻 **None mode** = Airplane mode ✈ **Overlay** = Corporate VPN connecting multiple offices 🏢🏢

Docker Networking Overview

> 📦 **Container networking** is the foundation of container communication. Every container is equipped with a network interface, IP address, routing table, DNS config, etc.

☑ By default, containers can:

- Make outbound connections (internet)

- Be connected to **default or custom networks**
- Be isolated or exposed, depending on the configuration

🚦 Types of Docker Network Drivers

Network Driver	Description	Use Case
`bridge`	🧱 Default isolated network for single-host	Local development
`host`	🏠 Shares host's network namespace	Low-latency, host-level apps
`none`	🚫 No networking at all	Offline or compute-only tasks
`overlay`	🌍 Enables multi-host (Swarm) communication	Distributed systems
`macvlan`	🪪 Assigns MAC + IP from host's network	IoT, legacy systems
`ipvlan`	Similar to macvlan, IP only	More control over routing
`custom plugins`	🛠️ Extendable third-party solutions	SDN, advanced use

🧱 Default Bridge Network vs 🧩 User-Defined Bridge Network

Both are based on the `bridge` driver, but they differ **significantly** in features & behavior.

🧱 Default `bridge` Network

Created automatically by Docker when installed.

```
docker network ls
# OUTPUT will contain:
# bridge    bridge    local
```

> Example:

```
docker run -d --name app1 nginx
docker run -d --name app2 busybox sleep 999
```

🕳️ Limitations:

❌ Limitation	Explanation
🚫 No DNS	Can't resolve container names
🚷 Not isolated	All containers using default bridge are technically on the same LAN

❌ Limitation	Explanation
Poor discoverability	Need to link manually or use IPs
Less secure	Containers can talk across networks by default if not isolated

---

User-Defined Bridge Network

> Created with:

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

Advantages:

Feature	Benefit
Built-in DNS	Containers can resolve each other by **name**
Isolated environment	Containers only talk to others on the same network
Auto service discovery	Works like microservices
Fine-grained control	Inspect, attach, detach easily
Easier multi-container orchestration	Compose, Swarm, or manually

---

Comparison: Default vs User-Defined Bridge

Feature	Default `bridge`	User-Defined Bridge
DNS support	❌ No	☑ Yes
Service name resolution	❌ No	☑ Yes
Isolation	❌ Less secure	☑ Scoped per network
Compose support	⊘ Not recommended	☑ Fully supported
Security	Basic	Scoped & controlled
Container-to-container name access	❌ Only via IP	☑ Via name (`ping app1`)
Preferred for production/dev	❌ No	☑ Yes

Inspecting Networks

```
docker network inspect my-custom-net
```

Output includes:

- Subnet
- Gateway
- Connected containers
- DNS aliases

🩹 Example Test

1. Default Bridge (no DNS):

```
docker run -d --name alpha nginx
docker run -it --rm busybox
# ping alpha – ❌ fails
```

2. User-defined Bridge:

```
docker network create testnet
docker run -d --name alpha --network testnet nginx
docker run -it --rm --network testnet busybox
# ping alpha – ☑ works
```

🧱 Other Drivers – Quick Overview

Driver	Emoji	Key Use
`host`	🏠	High-perf apps (no NAT), not isolated
`none`	🚫	Secure offline or processing containers
`overlay`	🌍	Multi-host Swarm networking
`macvlan`	🪪	Assign physical IPs from host's LAN
`ipvlan`	🎯	Fine-grained routing control
`custom plugin`	🧰	CNI integrations, SDNs (like Calico)

🔐 When to Use What?

Use Case	Best Network Driver
Local dev, isolated apps	🧩 User-defined bridge
Multi-container orchestration	🧩 Custom bridge + Docker Compose
High-speed, low-latency app (e.g. Prometheus)	🏠 Host network

Use Case	Best Network Driver
No internet access container	None
Containers across multiple hosts	Overlay (Swarm mode)
Assign IPs from LAN for legacy systems	Macvlan

Key Docker Networking Commands Cheat Sheet

Command	Description
`docker network ls`	List all available networks
`docker network create <name>`	Create a custom bridge network
`docker network rm <name>`	Remove a network
`docker network inspect <name>`	View detailed info about a network
`docker run --network <name> <image>`	Create container in a specific network (unnamed)
`docker run -d --name <container> --network <network> <image>`	Create **named container** in a custom network
`docker network connect <network> <container>`	Attach an **existing container** to a network
`docker network disconnect <network> <container>`	Detach a container from a network

Real-World Example: Create and Use a Custom Bridge Network

```
# 1 Create a custom bridge network
docker network create my-bridge

# 2 Run a container (nginx) attached to that network
docker run -d --name my-app --network my-bridge nginx

# 3 Run another container (busybox) in the same network
docker run -it --name client --network my-bridge busybox

# 4 Inside 'client', you can ping 'my-app' by name
ping my-app
```

Result: `client` can resolve and communicate with `my-app` using container name thanks to Docker's internal DNS in **custom bridge networks**.

Bonus Tip: Disconnect & Reconnect

```
docker network disconnect my-bridge my-app    # Disconnect from network
docker network connect my-bridge my-app       # Reconnect to same or new network
```

🧠 Final Takeaways

- 🧱 **Default bridge** is basic → no name resolution, low security
- 🧩 **User-defined bridge** is preferred for real-world apps
- 🌍 Use **overlay** for distributed microservices
- 🧰 Know when to use each driver to optimize performance & security
- 🧪 Always test communication with tools like `ping`, `curl`, `netcat`

🌐 Docker Networking Logic – Container Communication

☑️ 1. **User-Defined Bridge Network = Container DNS Heaven** 🧠

> 🧩 When you create a **user-defined bridge network**, Docker automatically enables an **internal DNS service**.

📦 That means containers can **talk to each other by name**! 🎉

Example:

```
docker network create my-net

docker run -d --name db --network my-net mongo
docker run -d --name web --network my-net node-app
```

🎯 Now, inside `web`, you can:

```
ping db
```

☑️ Boom! It works because Docker auto-resolves `db` → container's IP inside the same network.

❌ 2. Default Bridge Network = Only IP-Based Access 🔢

> Containers in the **default bridge** cannot resolve each other by name.

Example:

```
docker run -d --name app1 nginx       # default bridge
docker run -it --name app2 busybox    # default bridge
```

Inside app2:

```
ping app1    # ❌ FAILS
```

Why? Because **DNS resolution doesn't work** in the default `bridge` without using the old `--link` option (now deprecated 🚫).

👄 Legacy `--link` (Avoid Using It)

```
docker run -d --name db mongo
docker run -d --name web --link db node-app
```

⚠️ Works — but deprecated & removed in newer Docker versions.

🔄 3. Containers in Different Networks = 🚫 No Communication

Example:

```
docker network create net-a
docker network create net-b

docker run -d --name app1 --network net-a nginx
docker run -d --name app2 --network net-b busybox
```

Inside app2:

```
ping app1    # ❌ FAILS – isolated networks
```

> 🧱 Networks are isolated **by default**. Containers on different bridge networks **cannot talk to each other** unless you **connect one container to both** using:

```
docker network connect net-a app2
```

Now, app2 belongs to both networks!

🧠 Real-World Analogy

Concept	Analogy
🧱 Default Bridge	Talking in a crowd with no names, only IPs (👤 192.168.x.x)
🧩 User-defined Bridge	Talking in a chatroom where everyone has a username (👄 @web, @db)
🔌 Multi-Network	Having one foot in two rooms 🦶🚪

🩹 Recap – Container Communication Matrix

Scenario	Communicate by name?	Communicate by IP?	Notes
Same user-defined bridge	☑ Yes	☑ Yes	Best practice
Same default bridge	❌ No	☑ Yes	No name resolution
Different networks	❌ No	❌ No	Unless manually connected
With `--link` (legacy)	⚠ Yes	☑ Yes	Deprecated, avoid

☑ Summary

- 🧩 **User-defined networks** allow **name-based** communication via Docker's built-in DNS.
- 🧱 **Default bridge** networks **don't support DNS** — only IPs.
- 🔐 **Each network is isolated** — containers inside a network can talk, but can't reach containers in **other** networks unless **manually connected**.
- 💡 Use `docker network connect` to join a container to multiple networks if needed.

🌐 Overview of Advanced Docker Network Drivers

Driver	Purpose	Host-to-Container	Container-to-Host	Cross-Host Support
`overlay`	Cross-host container communication (Swarm)	☑ Yes	☑ Yes	☑ Yes
`macvlan`	Assign real MAC & IP from LAN to container	❌ No (by default)	☑ Yes	❌ No
`ipvlan`	IP-level control without creating MAC addresses	☑ Yes	☑ Yes	❌ No

---

1 🌍 Overlay Network

🔍 What It Is:

> Allows **containers on different Docker hosts** to securely communicate as if they were on the same LAN.

💡 Requires Docker Swarm (or other orchestrators). Creates an **encrypted VXLAN tunnel** between hosts.

🧠 Real-world Analogy:

> Like a VPN that connects branch offices (containers) across cities (hosts).

☑️ Use Cases:

- Multi-host microservices
- Docker Swarm services
- HA + distributed architecture

🚀 How to Use (with Swarm):

```
# 1. Initialize Swarm
docker swarm init

# 2. Create overlay network
docker network create --driver overlay my-overlay

# 3. Deploy service to use overlay
docker service create \
  --name webapp \
  --replicas 3 \
  --network my-overlay \
  nginx
```

🔐 Key Features:

Feature	Benefit
🔒 Encrypted traffic	Uses IPSEC tunneling
🔄 Auto service discovery	Works across nodes
📦 Built-in load balancing	Container-to-service routing
🔧 Works with Docker Compose (v3+)	Great for multi-host orchestration

---

2️⃣ 🪪 Macvlan Network

🔍 What It Is:

> Allows containers to appear as **physical devices on the host's network**, each with their **own IP and MAC**.

💡 The container **bypasses Docker's NAT**, appearing directly on your LAN.

🧠 Real-world Analogy:

> Like plugging a new computer (container) directly into your office switch with its own IP.

☑️ Use Cases:

- Legacy apps that require static IPs or MACs
- IoT, embedded, or bare metal simulation
- When containers must be reachable **from the LAN directly**

🚀 How to Use:

```
# 1. Create macvlan network
docker network create -d macvlan \
  --subnet=192.168.1.0/24 \
  --gateway=192.168.1.1 \
  -o parent=eth0 \
  macvlan-net

# 2. Run a container on that network
docker run -d --name myrouter \
  --network macvlan-net \
  busybox sleep 3600
```

📌 `parent=eth0`: your physical host's network interface

⚠️ Limitations:

❌ Limitation	Description
🚫 No container-to-host	Can't ping container from host by default
🔓 Can bypass Docker security stack	Be cautious in shared infra
🌍 Requires IP address planning	Must avoid IP conflicts

🛠️ Tip to Enable Host ↔ Container Communication (Workaround):

Create a **dummy interface** on the host:

```
ip link add macvlan-shim link eth0 type macvlan mode bridge
ip addr add 192.168.1.200/24 dev macvlan-shim
ip link set macvlan-shim up
```

---

3 IPvlan Network

What It Is:

> Similar to macvlan, but **no extra MACs per container**. All containers **share host's MAC**, just get different IPs.

More compatible with cloud and DHCP setups where duplicate MACs are not allowed.

Analogy:

> Multiple workers using one ID card (MAC) but different phone numbers (IP).

Use Cases:

- Performance-sensitive systems
- Cloud infra with MAC restrictions
- Advanced network routing with minimal overhead

How to Use:

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

docker run -it --rm --network ipvlan-net alpine
```

IPvlan supports two modes:

- `l2`: Same subnet, like macvlan
- `l3`: Different subnet, routing via host

Benefits:

Feature	Benefit
Efficient	No MAC duplication

Feature	Benefit
More predictable IP rules	Good for secured environments
Custom routing support	Great for advanced network setups

Advanced Network Drivers Comparison Table

Feature/Driver	`overlay`	`macvlan`	`ipvlan`
Cross-host support	☑	✗	✗
Requires Swarm?	☑	✗	✗
Uses physical IP/MAC	✗	☑	☑ (IP only)
Host ↔ container communication	☑	✗ (manual fix)	☑ (limited)
Best for	Microservices on multiple hosts	LAN-level communication	Custom IP routing or cloud infra
Security model	Swarm-controlled	Exposes real IP on LAN	More controlled than macvlan
Complexity	Medium	High	Medium-High

Security Considerations

Driver	Security Tip
Overlay	Isolated per service; enable encryption
Macvlan	Bypasses Docker's firewall — isolate via VLAN
IPvlan	Good firewall compatibility; still isolate with subnet rules

When to Use What?

Situation	Use Driver
Multi-host Swarm deployments	`overlay`
Direct LAN access required	`macvlan`
Controlled IP mapping, no MAC exposure	`ipvlan`

Summary

- `overlay`: Best for **Swarm**, cross-node services, scalable infra
- `macvlan`: Best for **LAN visibility**, legacy hardware, IP-bound apps

- `ipvlan`: Best for **performance** and **controlled environments** (e.g., cloud)

🚫 Docker `none` Network Driver – Ultimate Guide

❓ What is the `none` Network?

> The `none` network is a special Docker network driver that **completely disables networking** for a container.

🚫 No IP address 🚫 No routing 🚫 No DNS 🚫 No internet access 🚫 No communication with host or other containers

---

🔐 Use Case:

When you want your container to **run in complete isolation**, especially for:

- ☑️ CPU-intensive or file-only tasks
- ☑️ Secure environments with no external communication
- ☑️ Containers that interact only via **volume sharing** or **IPC**
- ☑️ Avoiding **network-related attacks** (like SSRF, port scanning, etc.)

🧠 Real-World Analogy

> It's like putting a person in a **soundproof, windowless room** 🧍🔇 They can compute, read, or write files – but **cannot talk or hear** the outside world.

🛠️ How to Use It

```
docker run -it --rm --network none alpine
```

Then inside the container:

```
ping google.com   # ❌ Fails
ip addr            # Shows no IP
```

☑️ The container runs, but it's **completely cut off** from any kind of networking.

🔍 Check from the Host

```
docker inspect <container-id> | grep -i "NetworkMode"
# Output: "NetworkMode": "none"
```

Useful Scenarios

Use Case	Why `none` Works
Data processing / math computation	Doesn't need the internet
Security sandboxing	No attack vector via networking
Testing firewall rules	Simulate “no internet” condition
Build-only stages	Avoid leaking credentials over network
DevOps CI/CD	Run isolated build/test tasks with no exposure

Warning

- You **cannot** `ping`, `curl`, `apt update`, **or download** anything inside containers using `--network none`
- Any tools that require internet or inter-container access will **fail**
- It's **not usable** for most microservices or web APIs

Tip: Combine with Volumes or IPC

If you want to **exchange data** without a network:

```
# Create a shared volume
docker volume create shared-data

# Use it with the isolated container
docker run -it --rm --network none -v shared-data:/data alpine
```

☑ This lets you read/write to shared storage **without needing any network access.**

Summary Table

Feature	`none` Driver
IP address	❌ None
DNS	❌ None
Host access	❌ None
Container-to-container	❌ None
Internet	❌ None
Use case	Security, sandboxing, isolated compute

🔐 Final Verdict

If you need **absolute network isolation**, `--network none` is your **zero-trust go-to option**. It's perfect for:

- 🔐 Security-first workloads
- 🧪 Testing internal-only logic
- 🚫 Disabling all remote calls