Container Network - Containers Communication Setup

Communication Between Containers in a Custom Bridge Network

Communication Between Containers in a Custom Bridge Network
When working with Docker, containers by default are isolated. However, when containers need to communicate with each other, you can connect them to the same Docker network. A user-defined bridge network provides more control over how Docker containers communicate compared to the default bridge network. This guide will walk you through setting up a custom bridge network, launching multiple Nginx containers on that network, and verifying communication between them.

image

Why Use a User-Defined Bridge Network?
By default, Docker containers can communicate over a built-in network called the "default bridge network." However, using a user-defined bridge network provides the following benefits:

Name resolution: Containers connected to the same network can communicate by their container names, making it easier to manage multi-container setups.
Isolated environment: Containers on a user-defined network are isolated from others unless explicitly connected to other networks.
Security: You can control which containers can communicate by connecting them only to specific networks.
Now, let's move on to creating the network and launching our containers.

Creating the User-Defined Bridge Network
The first step is to create a custom bridge network. Docker allows you to create networks of different types, such as bridge, overlay, and host. Here, we'll use the bridge driver, which is the default type for local container communication on a single host.

Run the following command to create the network:

docker network create --driver bridge my-bridge-network
This command creates a bridge network named my-bridge-network. You can inspect the network details using the command below:

docker network inspect my-bridge-network
This will give you detailed information about the network, including its subnet, gateway, and connected containers.

Verifying Network Creation
You can list all existing Docker networks by running:

docker network ls

Expected Output:

image

The newly created my-bridge-network should appear in the list, showing that it uses the bridge driver.

Launching Containers and Connecting to the Network
In this section, we'll launch three containers (container1, container2, and container3), each running the Nginx web server, and connect them to our user-defined network.

Launching Container 1
We use the following command to launch container1, and immediately connect it to the my-bridge-network network:

docker run -d --name container1 --network=my-bridge-network nginx
This will run the Nginx web server in the background (-d) and assign the name container1 to the instance.

Launching Container 2
Similarly, to launch container2, use:

docker run -d --name container2 --network=my-bridge-network nginx
Launching Container 3
Finally, to launch container3, use:

docker run -d --name container3 --network=my-bridge-network nginx
With all three containers running, they are now connected to the same network, my-bridge-network. This enables them to communicate directly with one another.

Verifying Container Status
To check the status of the running containers, use the command:

docker ps
Expected output:

image

Here, you'll see the list of running containers along with their names, statuses, and other details like port mappings. The containers container1, container2, and container3 should be listed as running, confirming that Nginx is operational inside each container.

Verifying Communication Between Containers
Now that the containers are up and running, let's check if they can communicate with each other using their container names.

Accessing the Shell of Container 1
First, we'll access the shell of container1 to ping the other containers. Run:

docker exec -it container1 /bin/bash
This opens an interactive shell session inside container1. From this session, we can try pinging the

other containers by their names.

Pinging Container 2 from Container 1
To test connectivity from container1 to container2, run:

ping container2 -c 5
This command will send 5 ICMP echo requests to container2. A successful ping will indicate that container1 can communicate with container2.

Expected Output:

image

Pinging Container 3 from Container 1
Next, try pinging container3 from container1:

ping container3 -c 5
Expected Output:

image

The successful responses confirm that container1 can reach both container2 and container3 within the custom network.

Accessing the Shell of Container 2
We can repeat the process from another container. Access the shell of container2 by running:

docker exec -it container2 /bin/bash
Once inside the shell, you can ping container1 and container3.

Pinging Container 1 from Container 2
ping container1 -c 5
Expected Output:

image

Pinging Container 3 from Container 2
ping container3 -c 5
Expected Output:

image

These tests confirm that all the containers can communicate with each other over the custom bridge network.

Conclusion
By following these steps, we successfully created a user-defined bridge network and launched multiple Nginx containers connected to the network. We verified that they can communicate with each other by pinging container names. This demonstrates how Docker networking facilitates smooth communication between containerized applications, making it easier to manage interconnected services.

explain these for a simple child in a very easiest manner in details with basics also correct any error

Alright! Imagine you have a big toy box, and inside it, you have three walkie-talkies (like small radios) that can talk to each other. Each walkie-talkie is like a tiny computer, and you want them to talk to each other, but they need to be on the same frequency (channel) to work.

Now, let's turn this into a simple explanation of **how containers talk to each other in Docker using a custom bridge network.**

What is a Container?

Think of a container as a **lunchbox**. It holds everything needed for a meal (food, fork, spoon, etc.). But in computers, a container holds everything needed to run an app (files, settings, and programs).

In our case, each container will have **a small web server called Nginx** inside it.

Why Do Containers Need a Network?

Imagine you and your friends are using walkie-talkies. If you are on the same channel, you can talk to each other. But if you are on different channels, you cannot hear each other.

Similarly, **containers need to be on the same network to talk to each other.** Docker gives them a default network, but we want a special **custom network** where only our chosen containers can communicate.

Step 1: Creating a Custom Network

Before putting our walkie-talkies (containers) in the toy box (computer), we need to make sure they are using the same channel (network).

To do this, we create a special network using this command:

```bash
docker network create --driver bridge my-bridge-network
```

- This creates a private space (a road where only our containers can drive).
- We name this network **my-bridge-network**.
- The **bridge** type means containers can talk inside but stay isolated from others.

To check if our network is made, we run:

```bash
docker network ls
```

- This shows all the networks, including our new **my-bridge-network**.

Step 2: Creating Containers and Connecting Them

Now, we add our walkie-talkies (containers) to the network.

Launching Container 1

We create a new **container** and put it in the **my-bridge-network**.

```bash
docker run -d --name container1 --network=my-bridge-network nginx
```

- `-d` means **run in the background**.
- `--name container1` gives it a name.
- `--network=my-bridge-network` connects it to our special network.
- `nginx` is the small web server running inside it.

Launching Container 2

```bash
docker run -d --name container2 --network=my-bridge-network nginx
```

Launching Container 3

```bash
docker run -d --name container3 --network=my-bridge-network nginx
```

Now, **all three containers are on the same network** and can talk to each other.

Step 3: Checking If Containers Are Running

To see if our containers are running, we use:

```bash
docker ps
```

- This shows all running containers.
- We should see **container1, container2, and container3** listed.

Step 4: Testing Communication (Can They Talk?)

Now, let's check if the containers can talk to each other.

Entering Container 1

We go inside **container1** to test communication:

```bash
docker exec -it container1 /bin/bash
```

- This opens a terminal inside container1 (like opening a chat window).

Pinging Container 2

From inside container1, we try:

```bash
ping container2 -c 5
```

- This asks container2, "Hey, are you there?"
- If container2 is working, it will reply!

Pinging Container 3

We also check container3:

```bash
ping container3 -c 5
```

- If the response comes, that means container1 can talk to container3.

Step 5: Testing from Another Container

Let's do the same from **container2**.

```bash
docker exec -it container2 /bin/bash
```

- Now, we are inside **container2**.

Pinging Container 1

```bash
ping container1 -c 5
```

- If container1 responds, it means they can talk.

Pinging Container 3

```bash
ping container3 -c 5
```

Final Result

If all pings work, it means: ✅ container1 can talk to container2 and container3
✅ container2 can talk to container1 and container3
✅ container3 can talk to container1 and container2

Conclusion

1. **We created a special network** so our containers can talk.
2. **We started three containers** with Nginx inside them.
3. **We checked if they could talk to each other** using ping.
4. **Everything worked!** 🎉

This setup is useful when you want different apps (containers) to talk without disturbing others.

Example in Real Life:
Imagine a restaurant with three chefs working together in a private kitchen. They can talk to each other inside (our private network) but cannot talk to chefs in another restaurant (other networks).

Would you like any extra explanations? 😊

root@17cadad9c0ff:/# docker exec -it container1 /bin/bash
bash: docker: command not found
root@17cadad9c0ff:/# exit
exit
root@836b6b039e468e57:~# docker exec -it container1 /bin/bash
root@17cadad9c0ff:/# ping container2 -c 5
bash: ping: command not found
root@17cadad9c0ff:/# exit
exit
root@836b6b039e468e57:~# docker exec -it container2 /bin/bash
root@9d1c674c92d9:/#
root@9d1c674c92d9:/# ping container1 -c 5
bash: ping: command not found
root@9d1c674c92d9:/#

solve this

It looks like your **Docker containers do not have the `ping` command installed**. This happens because the official `nginx` image (which is based on `Debian` or `Alpine Linux`) is a minimal image and does not include unnecessary utilities like `ping`.

Solution

Step 1: Install `ping` inside the containers

You need to install `iputils-ping` or `ping` inside each container.

For Debian-based containers (default `nginx` image)

If your `nginx` container is based on Debian, run:

```bash
docker exec -it container1 bash
apt update && apt install -y iputils-ping
```

Then exit and repeat for other containers:

```bash
exit
docker exec -it container2 bash
apt update && apt install -y iputils-ping
```

For Alpine-based containers (`nginx:alpine`)

If you used the `nginx:alpine` image, use this instead:

```bash
docker exec -it container1 sh
apk add --no-cache iputils
```

Then do the same for `container2` and `container3`.

Step 2: Test Connectivity Again

Once `ping` is installed, try:

```bash
docker exec -it container1 bash  # Or sh for Alpine
ping container2 -c 5
```

You should now see the expected responses.

Alternative Method: Using `curl` Instead of `ping`

If you don't want to install `ping`, you can test connectivity using `curl`:

```bash
docker exec -it container1 bash
```

```
curl -I http://container2
```

This will test if `container2`'s Nginx server is reachable from `container1`.

This should solve your issue! 🚀 Let me know if you need more help. 😊

explain these word by word and don't miss any important part in a very easiest manner in details with basics also correct if any error occur; what does these codes do with basics of each part and portion with basic example if necessary:

Exploring Bridge Networks in Docker

Exploring Bridge Networks in Docker
In this documentation, we'll explore bridge networks in Docker by creating and examining custom bridge networks, attaching containers to multiple networks, and using tools like ip and nmap to examine network configurations and discover other containers.

Creating and Examining a Custom Bridge Network
Let's start by creating a custom bridge network named user-network and exploring its configuration:

docker network create \
 --driver bridge \
 --label project=dockerinaction \
 --label chapter=5 \
 --attachable \
 --scope local \
 --subnet 10.0.42.0/24 \
 --ip-range 10.0.42.128/25 \
 user-network
This Docker command creates a custom bridge network with the following configurations:

Bridge driver for local container communication.
Labels (project=dockerinaction, chapter=5) to categorize and organize the network.
Attachable network allows standalone containers to join.
Scope: Local, restricting the network to a single Docker host.
Custom subnet (10.0.42.0/24) and IP range (10.0.42.128/25), controlling the IPs assigned to containers.
Network name: user-network.
This setup allows precise control over container networking and IP management within the defined range.

Inspect the newly created bridge:
image

Now, let's launch a container attached to this network and examine its network interfaces:

```
docker run -it \
  --network user-network \
  --name network-explorer \
  alpine:3.8 \
    sh
```

image

Inside the container, run the following command to list the available IPv4 addresses:

```
ip -f inet -4 -o addr
```

We'll see output similar to:

image

This output shows that the container has an IP address within the range and subnet specified by the user-network configuration.

Attaching Containers to Multiple Networks

Now, let's attach the network-explorer container to another bridge network named user-network2:

```
docker network create \
  --driver bridge \
  --label project=dockerinaction \
  --label chapter=5 \
  --attachable \
  --scope local \
  --subnet 10.0.43.0/24 \
  --ip-range 10.0.43.128/25 \
  user-network2
```

To see the networks :

```
docker network ls
```

Expected Output:

image

Next, attach the network-explorer container to user-network2:

```
docker network connect \
  user-network2 \
  network-explorer
```

image

Installing nmap Inside the Container

Inside the network-explorer container, install the nmap package:

```
docker exec -it network-explorer sh -c "apk update && apk add nmap"
```

This command executes a shell within the network-explorer container and installs the nmap package.

nmap: Installing nmap inside a Docker container provides network administrators and developers with a versatile tool for network exploration, security auditing, troubleshooting, and monitoring within the containerized environment.

Using nmap to Explore the Network
Now, scan the subnets of both bridge networks:

nmap -sn 10.0.42.* -sn 10.0.43.* -oG /dev/stdout | grep Status
image

Ping Scan: nmap is used to perform a ping scan (-sn) on IP ranges 10.0.42.* and 10.0.43.*.

Output Filtering: Results are formatted in a greppable format (-oG /dev/stdout) and filtered using grep to display only lines containing "Status".

We'll see output indicating the status of devices on each subnet, including containers and gateway adapters.

Conclusion
Exploring bridge networks in Docker allows us to understand network configurations, attach containers to multiple networks, and use tools like ip and nmap to examine network settings and discover other containers. This knowledge is essential for managing containerized applications and designing network architectures in Docker environments.

Great! Let's go step-by-step **like we're teaching a curious child**, and explain **every part** of these Docker commands and concepts using very **easy, basic examples**. I'll also correct anything that's unclear and break it into super understandable bits.

🚢 What Is a Bridge Network in Docker?

Imagine you have a bunch of toy robots (containers). They are all in a room (a **network**) and can talk to each other **if they are in the same room**.

A **bridge network** in Docker is like a virtual room (a network) where containers (robots) can talk using **IP addresses** (like walkie-talkie numbers).

🧱 PART 1: Creating a Custom Bridge Network

Command:

```bash
docker network create \
  --driver bridge \
  --label project=dockerinaction \
  --label chapter=5 \
  --attachable \
  --scope local \
  --subnet 10.0.42.0/24 \
  --ip-range 10.0.42.128/25 \
  user-network
```

Let's explain it part-by-part:

1. `` `docker network create` ``
   📦 This tells Docker to **create a new network**.
2. `` `--driver bridge` ``
   🛣️ The **bridge driver** makes a local network on your computer where containers can talk to each other like they're on the same LAN (like robots in the same room).
3. `` `--label project=dockerinaction` ``
   🏷️ A **label** is like a sticky note on the network saying **"this is part of the dockerinaction project"**.
4. `` `--label chapter=5` ``
   🏷️ Another label saying this is from **chapter 5** (helps organize or filter networks later).
5. `` `--attachable` ``
   🤝 This makes the network **joinable**, meaning you can attach **individual containers** to it later (important for flexibility).
6. `` `--scope local` ``
   🌍 This says the network is only available **on your local machine** (not shared across a cluster of machines).
7. `` `--subnet 10.0.42.0/24` ``
   🧮 Subnet defines the whole group of possible IPs (like phone numbers) — in this case, IPs from `` `10.0.42.0` `` to `` `10.0.42.255` `` (a total of 256 IPs).
8. `` `--ip-range 10.0.42.128/25` ``
   🎯 This limits what IPs Docker can assign to containers (from `` `10.0.42.128` `` to `` `10.0.42.255` ``, half of the subnet).
9. `` `user-network` ``
   🏷️ The name of this custom network is **user-network**.

🧠 **Why do we need all this?**
To have **control** over container communication, IPs, and organization — great for larger apps or secure setups.

🔍 PART 2: Running a Container on This Network

Command:

```bash
docker run -it \
  --network user-network \
  --name network-explorer \
  alpine:3.8 \
  sh
```

Breaking this down:

1. `` `docker run` ``
   🏁 Start a new container.

2. `-it`
   - `-i` = **interactive** (stay open).
   - `-t` = **terminal** (gives you a shell).
3. `--network user-network`
   🌐 Connect the container to the **user-network** we created earlier.
4. `--name network-explorer`
   🏷️ Give the container a name so we can refer to it easily.
5. `alpine:3.8`
   🏕️ Use a very small Linux OS image called **Alpine**, version 3.8 (lightweight and fast).
6. `sh`
   💬 Run the **shell** inside the container (like opening a terminal inside the robot).

🔎 PART 3: Checking IP Address Inside the Container

Command (inside container):

```bash
ip -f inet -4 -o addr
```

What this means:

- `ip` = tool to see networking info.
- `-f inet` = only show **IPv4** (not IPv6).
- `-4` = shorthand for IPv4.
- `-o` = output in one line (easy to read).
- `addr` = show address info.

📋 This tells us **what IP address the container got**, and confirms if it's within the range we set (`10.0.42.128/25`).

🔁 PART 4: Attaching a Container to a Second Network

First, create a new network:

```bash
docker network create \
  --driver bridge \
  --label project=dockerinaction \
  --label chapter=5 \
  --attachable \
  --scope local \
  --subnet 10.0.43.0/24 \
  --ip-range 10.0.43.128/25 \
  user-network2
```

⚙️ Same idea as before, but now it's a second network called **user-network2** with a different subnet (`10.0.43.0/24`).

Then, list all networks:

```bash
docker network ls
```

🧾 Shows all the networks Docker knows about.

Now attach the container to this second network:

```bash
docker network connect \
  user-network2 \
  network-explorer
```

✅ This means our **network-explorer** container is now connected to **two networks**:

- `user-network`
- `user-network2`

It's like a robot standing in **two rooms** and can hear both sets of friends.

🔧 PART 5: Installing `nmap` Inside the Container

Command:

```bash
docker exec -it network-explorer sh -c "apk update && apk add nmap"
```

Let's break it:

1. `docker exec -it network-explorer`
   🛠️ Run something **inside** the running container named **network-explorer**.
2. `sh -c "apk update && apk add nmap"`
   🚧 Inside the container:
   - `apk update`: Updates the list of available packages (like refreshing an app store).
   - `apk add nmap`: Installs **nmap**, a tool to explore networks.

🔬 PART 6: Using `nmap` to Explore Both Networks

Command:

```bash
nmap -sn 10.0.42.* -sn 10.0.43.* -oG /dev/stdout | grep Status
```

What this does:

1. **`nmap -sn 10.0.42.* -sn 10.0.43.*`**
   📡 Ping scan on both networks.
   - **`-sn`** = ping only (don't check open ports).
   - **`10.0.42.*`** and **`10.0.43.*`** = scan all addresses in those subnets.
2. **`-oG /dev/stdout`**
   📄 Output in a "greppable" format and print it to the screen.
3. **`| grep Status`**
   🔍 Only show lines with **"Status"**, to know which devices are online.

✅ You'll see containers and gateways that replied.

✅ Conclusion (Easy Words)

- We created a **custom room (network)**.
- We put a **robot (container)** in it and gave it an IP.
- Then we made **another room** and let the robot stand in **both rooms**.
- We gave it **nmap**, a tool like a radar scanner.
- It used **nmap** to find out who else is in those rooms.

This helps us build and manage **container-based apps** like mini cities where parts can talk clearly, safely, and only to who they're supposed to.

Would you like a **real drawing or diagram** to understand this better? Or I can make a demo with container names and IPs if you want!

what is nmap and what it does in these case

Great question! Let's break down **what `nmap` is**, and **what it does in this case**, using **simple, basic words**, like you're new to it. 😊

🔍 What is `nmap`?

➤ Full Form:

Nmap = **Network Mapper**

➤ Think of it like this:

Imagine you're in a dark room and you want to know who else is there. You **shout "hello!" in every direction** and listen for who replies. That's what `nmap` does — but instead of shouting, it **sends signals** (called packets) on a network to see who responds.

🛠️ What is `nmap` used for?

Here's what `nmap` can do:

What it does	Like this...
Detect devices	See which devices (computers, containers) are on the network
Check if online	Find out who is **active or not** (pinging)
Scan ports	See which doors (ports) are open on a computer (optional)
Map a network	Show how devices are connected (advanced use)

🤖 What does `nmap` do in this specific Docker case?

The command used:

```bash
nmap -sn 10.0.42.* -sn 10.0.43.* -oG /dev/stdout | grep Status
```

Let's explain what **each part** does with an example.

👣 Step-by-step meaning:

Part	Meaning
`nmap`	Run the network scanner
`-sn`	Do a **"ping scan"** — just check **who is alive**, not what ports are open
`10.0.42.*` and `10.0.43.*`	Look at IPs from `10.0.42.0` to `10.0.42.255` and `10.0.43.0` to `10.0.43.255`
`-oG /dev/stdout`	Show output in a simple text format (**G**reppable), directly to the screen
`	grep Status`

🧠 What this tells us:

It checks:

- Which containers or gateways are **up (alive)**
- Which IPs are responding (i.e., containers in the same networks)
- Whether two networks (`user-network` and `user-network2`) have reachable devices

🧪 Why is this useful?

- To **confirm** your containers are correctly connected to the network.
- To **see who is in the same subnet**.
- To **debug** networking issues: "Why can't my app container talk to my database container?"

🧒 Simple Example

Imagine your room has 256 chairs (IPs from `` `10.0.42.0` `` to `` `10.0.42.255` ``). You want to know which chairs have someone sitting in them.

You use `` `nmap` `` like a flashlight — it shines over every chair and tells you:
✅ Someone is here
❌ This chair is empty

That's exactly what `` `nmap` `` does — **finds which IPs are active** in the network.

✅ Summary (TL;DR)

Feature	Simple Explanation
nmap	A tool to scan networks
Why we use it	To see which containers are alive in Docker
What it does here	It checks if any containers respond from `` `user-network` `` and `` `user-network2` ``

Want a visual map or output sample from real Docker networks with `` `nmap` ``? I can show that too!

Conclusion
Exploring bridge networks in Docker allows us to understand network configurations, attach containers to multiple networks, and use tools like ip and nmap to examine network settings and discover other containers.

Sure! Here's a **simple and clear conclusion** in an **easy-to-understand** way, word by word:

✅ Conclusion (In Simple Words)

Exploring **bridge networks** in Docker helps us learn how containers talk to each other.

We saw that:

- 🧱 **Bridge networks** act like private rooms where containers (tiny apps) can communicate.
- ➕ We can create **custom bridge networks** with our own rules — like giving containers their own phone numbers (IP addresses).

- 🔌 We can **attach one container to more than one network** — like letting it be in two rooms at once.
- 🔍 We used tools like:
  - `ip` → to see the container's own IP address.
  - `nmap` → to scan the network and find other containers nearby (who else is "alive").

This knowledge helps us build **secure**, **organized**, and **connected** container systems — perfect for real apps where different parts need to talk (like web server + database).

If you'd like, I can turn this into a visual mind map or step diagram too!

```
root@91378858fe5dc7e1:~/code# ^C
root@91378858fe5dc7e1:~/code# docker network create \
>   --driver bridge \
>   --label project=dockerinaction \
>   --label chapter=5 \
>   --attachable \
>   --scope local \
>   --subnet 10.0.42.0/24 \
>   --ip-range 10.0.42.128/25 \
>   user-network
57321b98c7298008b123d2a3ca070aaf0cce7ce79e6dd6d28961f991b1adefe1
root@91378858fe5dc7e1:~/code# docker run -it \
>   --network user-network \
>   --name network-explorer \
>   alpine:3.8 \
>   sh
Unable to find image 'alpine:3.8' locally
3.8: Pulling from library/alpine
486039affc0a: Pull complete
Digest: sha256:2bb501e6173d9d006e56de5bce2720eb06396803300fe1687b58a7ff32bf4c14
Status: Downloaded newer image for alpine:3.8
/ # ip -f inet -4 -o addr
1: lo    inet 127.0.0.1/8 scope host lo\       valid_lft forever preferred_lft forever
7: eth0    inet 10.0.42.129/24 brd 10.0.42.255 scope global eth0\       valid_lft forever preferred_lft
forever
/ # exit
root@91378858fe5dc7e1:~/code# docker network create \
>   --driver bridge \
>   --label project=dockerinaction \
>   --label chapter=5 \
>   --attachable \
>   --scope local \
>   --subnet 10.0.43.0/24 \
>   --ip-range 10.0.43.128/25 \
>   user-network2
d7b3bd004c6954f669d4e83290b113fc7bba2ea46071e384acef8ada2ea51f8a
```

```
root@91378858fe5dc7e1:~/code# docker network ls
NETWORK ID     NAME          DRIVER    SCOPE
85b57c4b1882   bridge        bridge    local
5a1111fc7a81   host          host      local
2b3d74d15657   none          null      local
57321b98c729   user-network    bridge    local
d7b3bd004c69   user-network2   bridge    local
root@91378858fe5dc7e1:~/code# docker network connect \
>   user-network2 \
>   network-explorer
root@91378858fe5dc7e1:~/code# docker exec -it network-explorer sh -c "apk update && apk add
nmap"
Error response from daemon: Container
e99fd30d8e6873679395cd2ebdf6c2f884f75118e3b3135ecc05063437554a8e is not running
root@91378858fe5dc7e1:~/code# docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
root@91378858fe5dc7e1:~/code# docker ps -a
CONTAINER ID   IMAGE        COMMAND   CREATED         STATUS                     PORTS     NAMES
e99fd30d8e68   alpine:3.8   "sh"      5 minutes ago   Exited (0) 3 minutes ago             network-explorer
root@91378858fe5dc7e1:~/code#
```

Awesome progress! You're doing all the right steps 👏 — now let's understand **what happened**, **why the error occurred**, and how to **fix it step by step**, in the **easiest way possible**.

❌ Problem:

You ran this command:

```bash
docker exec -it network-explorer sh -c "apk update && apk add nmap"
```

And got this error:

```vbnet
Error response from daemon: Container e99fd30d8e6873679395cd2ebdf6c2f884f75118e3b3135ecc05063437554a
```

💡 Why did this happen?

Your container **`network-explorer`** **exited**. That means it **stopped running** before you tried to install **`nmap`**.

How do we know? You checked with:

```bash
docker ps -a
```

And it shows:

```makefile
STATUS: Exited (0)
```

This means the container **ran**, did nothing (it had no ongoing job), and **automatically stopped**.

Alpine's `sh` shell exits if it's not kept open with a task — that's normal behavior.

✅ Solution: Restart the container and keep it running

Step 1: Start the container again in the background:

```bash
docker start network-explorer
```

This just **restarts** the container.

Step 2: Install `nmap` inside it:

```bash
docker exec -it network-explorer sh -c "apk update && apk add nmap"
```

This:

- Enters the container and
- Updates Alpine's package list
- Installs `nmap`

Step 3: Use `nmap` to scan both subnets:

```bash
docker exec -it network-explorer sh
nmap -sn 10.0.42.* -sn 10.0.43.* -oG /dev/stdout | grep Status
```

If `nmap` shows **"Host is up"**, it means another container or gateway on the network replied.

🧠 Quick Explanation (Word by Word)

- `docker start network-explorer` → Starts the stopped container
- `docker exec -it` → Run a command **inside** a running container
- `apk update` → Update Alpine's package list (like refreshing app store)
- `apk add nmap` → Install the `nmap` tool inside the container

- `nmap -sn` → Just ping scan (don't scan ports)
- `10.0.42.*` → Scan IPs in the first network
- `10.0.43.*` → Scan IPs in the second network
- `-oG /dev/stdout` → Show output in greppable format on screen
- `| grep Status` → Show only "Host is up" lines (who is alive)

✅ Summary:

- Your container exited because Alpine `sh` finished.
- You just need to **start the container again** before using `exec`.
- Then install and use `nmap` as planned.

Would you like a visual cheat sheet showing this workflow? Or want to test pinging between containers next?