**5.1 Starting our first container**

*docker pull ubuntu*

*docker run ubuntu*

**What is happening? Do you see something?**

docker pull ubuntu : This command downloads the official Ubuntu image from Docker Hub (the repository for Docker images) to your local machine

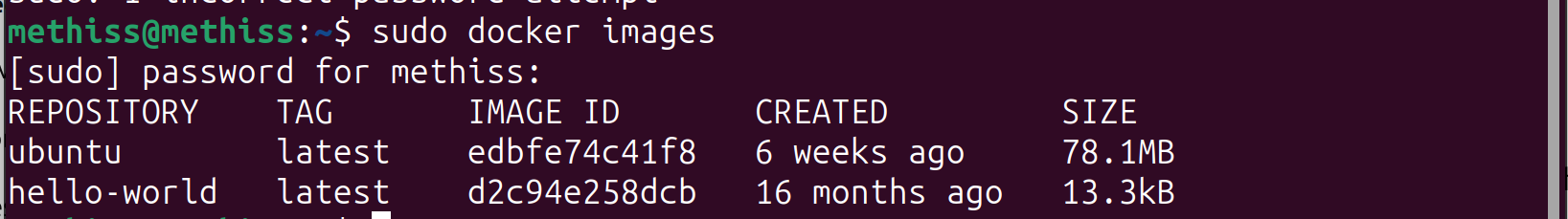
docker run ubuntu : This command runs the ubuntu image in a new container.

**What happens:**

* Docker creates a new container from the Ubuntu image.
* Since no specific command is provided, the container runs the default command specified in the image, which in this case is /bin/bash.
* After creating the container, Docker starts the container, runs the command, and then exits since there is no interactive shell or process running (in the default setup).

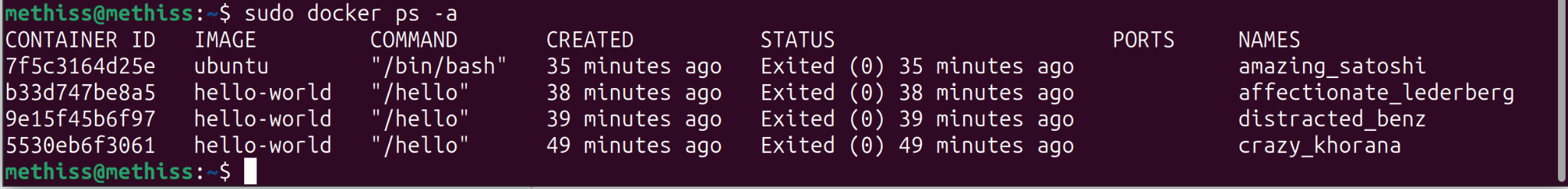
**Output:** Typically, you won’t see anything because the container starts, runs, and then immediately stops. This happens because there's no ongoing process inside the container to keep it running.

**Run docker images. What do you see?**



When you run the command docker images, Docker lists all the images that are currently available on your local machine.

**With the command docker ps -a, what does the command do?**

****

The command docker ps -a lists all Docker containers that exist on your system, including both running and stopped containers.

*docker run -it ubuntu /bin/bash*

**What’s the -it and /bin/bash doing?**

1. **docker run**: This tells Docker to create and start a new container based on the specified image (ubuntu in this case).
2. **-it**: These two flags are used together to create an interactive terminal session:
   * **-i**: Keeps the input stream open (interactive mode), so you can type commands inside the container.
   * **-t**: Allocates a pseudo-TTY (a terminal interface) to allow you to interact with the container in a terminal-like environment.
3. **ubuntu**: The Docker image that the container is based on (in this case, the official ubuntu image).
4. **/bin/bash**: The command to run inside the container. In this case, it launches the Bash shell, giving you an interactive terminal within the Ubuntu container.

### **What happens:**

* Docker creates a new container from the ubuntu image.
* The -it flag connects you to the terminal of the running container.
* It runs the bash shell inside the container, giving you access to a command line.

Now, you're essentially working *inside* the Ubuntu container. You will have a bash prompt where you can run commands, such as ls, apt update, cd, etc., just as if you were logged into a regular Ubuntu system.

### **Example output:**

Once you run the command, you’ll see something like this:



At this point, you are in an interactive terminal session inside the Ubuntu container. You can run commands like ls, pwd, or install software with apt-get as if you were on a fresh Ubuntu machine.

### **Exiting the container:**

To exit the interactive session and stop the container, you can simply type:

Using this command, we have created a second ubuntu container. The first one still exist and we have entered the second one

**Review the folders in the container. What can you say about the folder structure? Compare it with a standard Linux distribution.**

you'll notice that the folder structure is similar to a standard Linux distribution, but it is much more minimal.

**Basic Linux Directories:** The container includes most of the standard directories you would expect in any Linux file system, such as:

* /bin: Essential binary commands (like ls, cat, etc.).
* /etc: Configuration files for the system.
* /home: The home directory (though in a container, this is typically empty unless you create users).
* /lib and /lib64: Shared libraries used by binaries.
* /proc: Information about running processes.
* /root: The home directory for the root user inside the container.
* /usr: Contains system-wide binaries, libraries, and other resources.
* /var: Variable data, like logs and package caches.

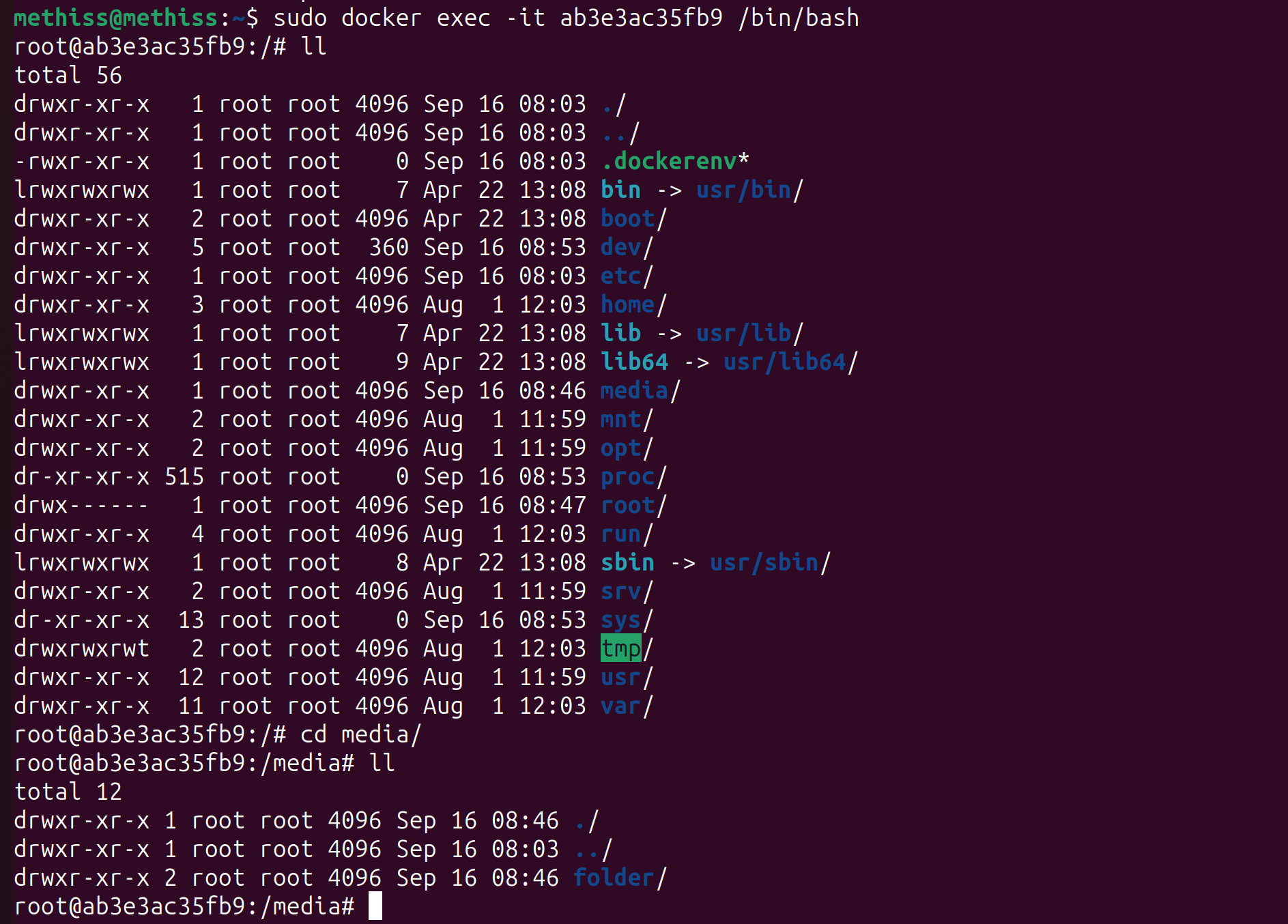
**Minimal Setup:** The Ubuntu Docker image is extremely lightweight compared to a full Linux installation. It’s meant to be a base system, so many unnecessary files, packages, and directories are omitted:

* You won’t find desktop environments, graphical libraries, or software that is typically part of a full Ubuntu distribution (like office software or media players).
* There's no /home/username because the container doesn't create users other than root by default.
* The /boot directory is empty or absent because there’s no bootloader or kernel needed in the container; it runs directly on the host kernel.

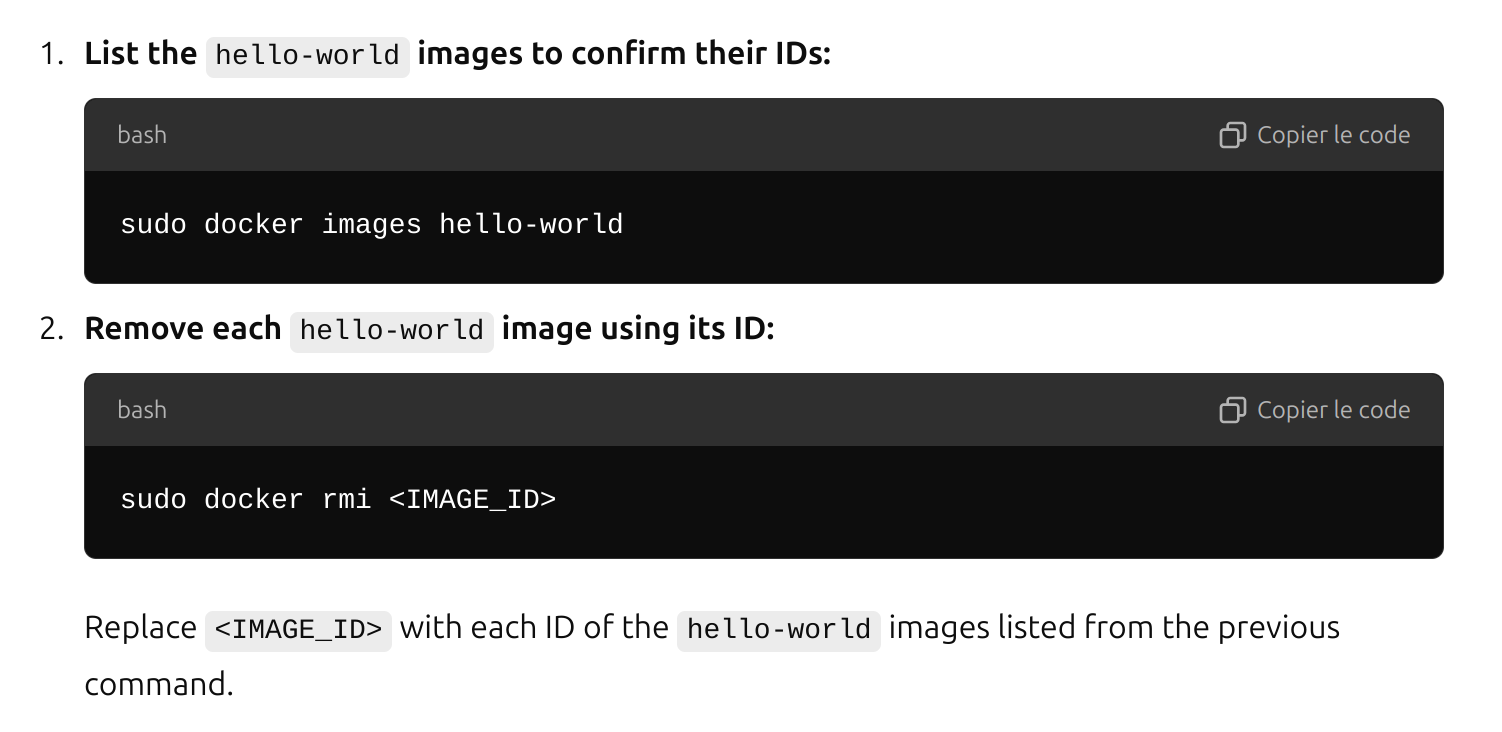
**Go to the /media and create a folder mkdir folder. Then exit the container and go back inside the same container. Do you find the folder /media/folder you have created?**

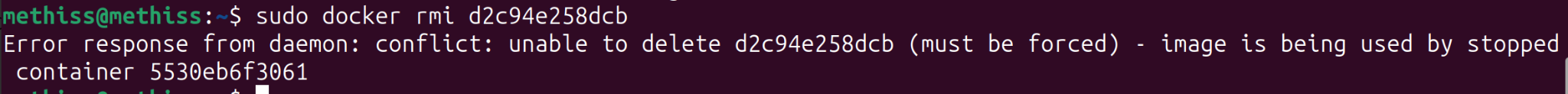
Exit the docker you are currently using and run the command again: docker run -it ubuntu /bin/bash . This will recreate a new conternairs with a different id but based on the same image so there will be no folder

whereas if you use the command: sudo docker exec -it ab3e3ac35fb9 /bin/bash . This returns you to the correct container where the folder file was created



**As we won’t need it anymore. Learn how to delete an image : remove the hello -world images from your computer**





force the remove :

sudo docker rmi -f d2c94e258dcb

**From this image, if we run twice the image, is the two containers identical?**

**What if we modify something inside a container, exit the container and run again the image. Do you think the modification will transfer to the newly created container?**

**What do you think is missing at this point?**

**Running the same image twice:** The two containers will start with identical configurations and files, but they are independent of each other.

**Modifications in a container:** Changes made inside a container do not transfer to new containers created from the same image. New containers will be identical to the image and will not include modifications made in previous containers.

**5.2 Volumes**

*docker volume create ubuntu-folder*

**Using the docker command and the manual, try to attach the volume you just created to an ubuntu container and run it interactively.**

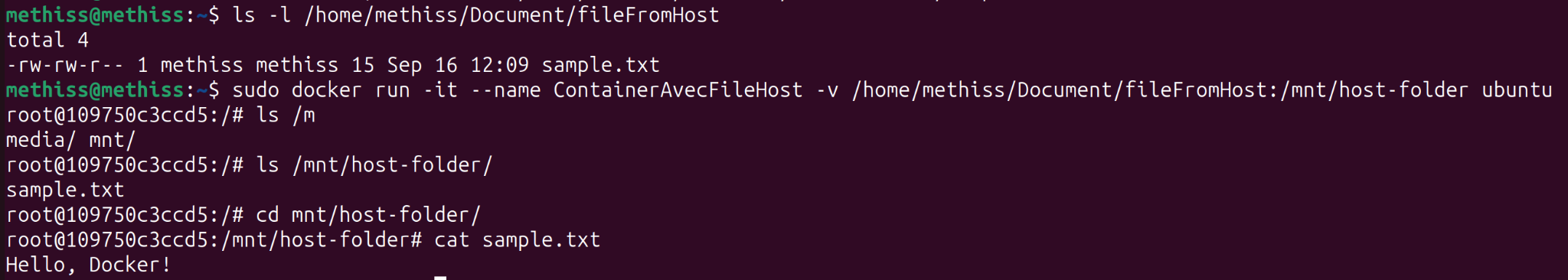
*docker run -it --name my-ubuntu-container -v ubuntu-folder:/mnt/ubuntu-folder ubuntu*

### **Explanation:**

* docker run: Command to create and start a new container.
* -it: Runs the container in interactive mode with a terminal.
* --name my-ubuntu-container: Assigns a name to the container (optional, but useful for identification).
* -v ubuntu-folder:/mnt/ubuntu-folder: Mounts the ubuntu-folder volume to the /mnt/ubuntu-folder directory inside the container.
* ubuntu: Specifies the image to use, in this case, the official Ubuntu image.

After running this command, you'll be inside the Ubuntu container with the volume mounted at /mnt/ubuntu-folder. You can interact with the volume and the container from this point.

**Try to bind-mount an existing folder on your computer to the container.**

**

*sudo docker run -it --name ContainerAvecFileHost -v /home/methiss/Document/fileFromHost:/mnt/host-folder ubuntu*

### **Explanation**

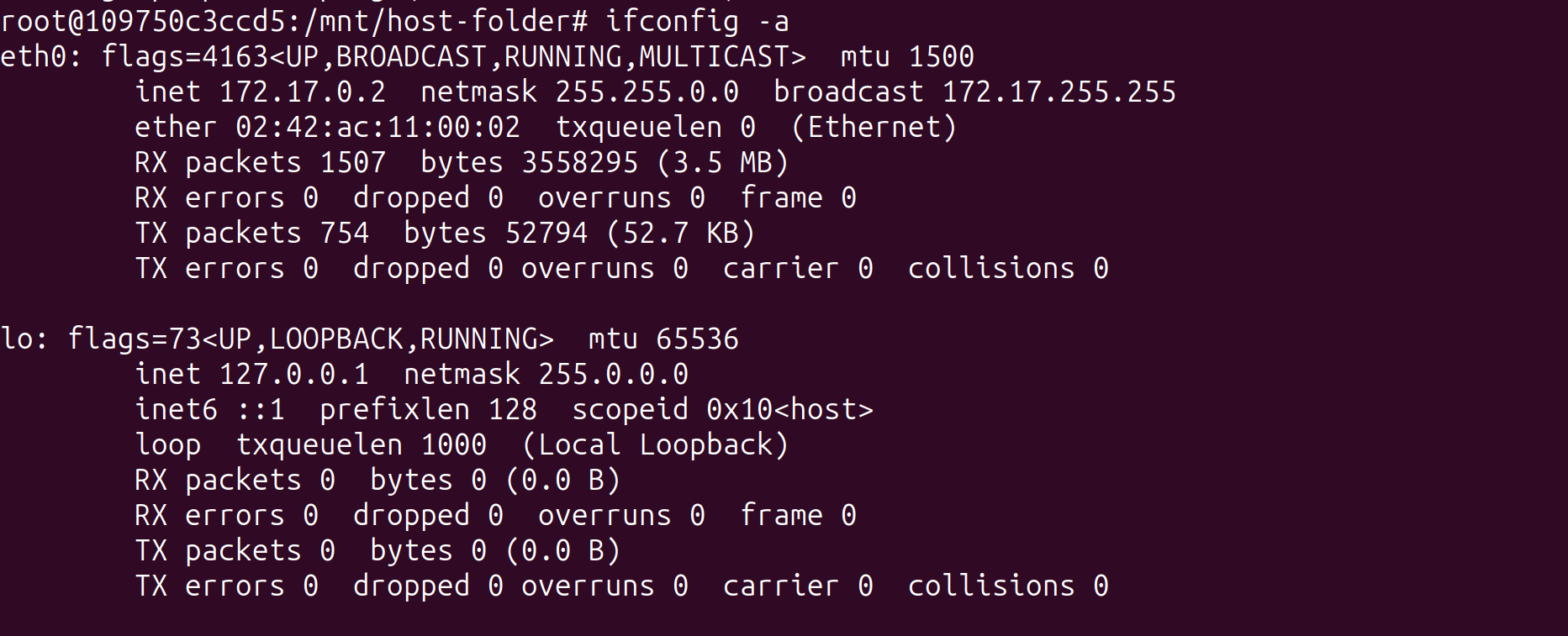
* docker run: Command to create and start a new container.
* -it: Runs the container in interactive mode with a terminal.
* --name my-ubuntu-container: Assigns a name to the container (optional but useful).
* -v /path/on/host:/mnt/host-folder: Mounts the directory /path/on/host from your host to /mnt/host-folder inside the container.
* ubuntu: Specifies the image to use, in this case, the official Ubuntu image.

**Try to modify a file from inside the container and try to modify the same file outside the container. Do you see the changes on either sides?**

No matter where you modify the file, the changes will be applied, whether from the volume or from the host.

**5.3 Networks**

**Run the command ifconfig -a. What can you say on the interfaces?**

****

Here’s a general breakdown of what you’ll see when running ifconfig -a in a container:

* **eth0**:
  + The default network interface inside the container.
  + Typically configured by Docker to allow communication between containers and the host machine.
  + It usually has an internal IP address from Docker's bridge network (e.g., 172.x.x.x).
* **lo (Loopback Interface)**:
  + This is the loopback interface with the IP address 127.0.0.1.
  + It allows the container to communicate with itself internally.

Depending on how Docker networking is set up, you might see other interfaces such as docker0 (the bridge network on the host), but typically inside the container, you'll mostly see eth0 and lo.

1. **Create a custom network with a subnet and assign a static IP to a container.**
2. **Run a second container on the same network.**
3. **Verify communication between the two containers.**
4. **Create a diagram of multiple containers across different networks.**

### **Step 1: Create a custom network with a subnet and assign a static IP to a container**

We will create a Docker network with a custom subnet and specify an IP for the first Ubuntu container.

**Create the custom Docker network:**bash  
Copier le code  
sudo docker network create --subnet=192.168.20.0/24 my\_custom\_network

* + my\_custom\_network: the name of the custom network.
  + --subnet=192.168.20.0/24: defines the IP range for this network.

**Run the first Ubuntu container with a specific IP:**We will assign a static IP (192.168.20.10) to the first Ubuntu container.  
bash  
Copier le code  
sudo docker run -it --network=my\_custom\_network --ip=192.168.20.10 --name ubuntu\_container1 ubuntu /bin/bash

* + --network=my\_custom\_network: connects the container to the specified network.
  + --ip=192.168.20.10: assigns this static IP to the container.
  + --name ubuntu\_container1: names the container for easier reference.

1. The container will start in interactive mode, and you'll have access to the shell inside it.

### **Step 2: Run a second container on the same network**

Now, we will run a second Ubuntu container on the same custom network.

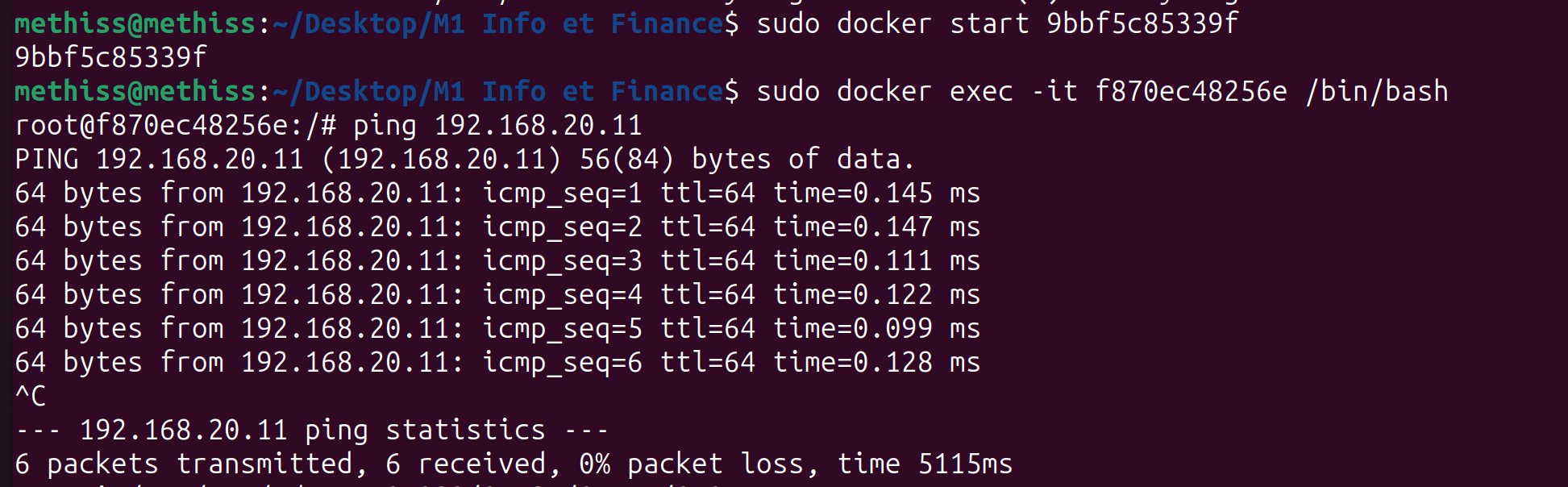
**Run the second Ubuntu container:**This container will also be on the same network, and Docker will automatically assign an IP from the 192.168.20.x subnet.  
bash  
Copier le code  
sudo docker run -it --network=my\_custom\_network --ip=192.168.20.11 --name ubuntu\_container2 ubuntu /bin/bash

1. This command will:
   * Assign 192.168.20.11 to the second container.
   * Name the container ubuntu\_container2.

### **Step 3: Verify communication between the two containers**

Once both containers are running on the same network, we can verify communication by using the ping command.

**From ubuntu\_container1 (IP: 192.168.20.10)**, ping ubuntu\_container2 (IP: 192.168.20.11):  
bash



1. **From ubuntu\_container2 (IP: 192.168.20.11)**, ping ubuntu\_container1 (IP: 192.168.20.10):  
   Similarly, you should see successful ping responses, confirming network communication.

### **Step 4: Diagram with multiple containers and networks**

Imagine we have a park of containers across different networks. We’ll draw a simple diagram and assign containers to different networks, specifying their IP addresses and whether they can communicate.

Let’s assume we have three custom networks:

1. **Network 1: network\_A**
   * Subnet: 192.168.10.0/24
   * Containers:
     + container\_A1: IP 192.168.10.10
     + container\_A2: IP 192.168.10.11
   * **Communication:** container\_A1 ↔ container\_A2 (can communicate)
2. **Network 2: network\_B**
   * Subnet: 192.168.30.0/24
   * Containers:
     + container\_B1: IP 192.168.30.10
     + container\_B2: IP 192.168.30.11
   * **Communication:** container\_B1 ↔ container\_B2 (can communicate)
3. **Network 3: network\_C**
   * Subnet: 192.168.40.0/24
   * Containers:
     + container\_C1: IP 192.168.40.10
     + container\_C2: IP 192.168.40.11
   * **Communication:** container\_C1 ↔ container\_C2 (can communicate)

### **Diagram:**

lua

Copier le code

+---------------------------+

| Network A |

| Subnet: 192.168.10.0/24 |

| |

| container\_A1: 192.168.10.10 <-------> container\_A2: 192.168.10.11 |

+---------------------------+

+---------------------------+

| Network B |

| Subnet: 192.168.30.0/24 |

| |

| container\_B1: 192.168.30.10 <-------> container\_B2: 192.168.30.11 |

+---------------------------+

+---------------------------+

| Network C |

| Subnet: 192.168.40.0/24 |

| |

| container\_C1: 192.168.40.10 <-------> container\_C2: 192.168.40.11 |

+---------------------------+

### **Key Details:**

* Containers within the same network can communicate with each other.
* **Communication between networks**: By default, Docker isolates containers in different networks. To allow communication between networks, you'd need to configure additional settings like setting up container links or using docker-compose with shared networks.

**Using the terminal, type the command sudo unshare --pid --fork --mount-**

**proc /bin/bash. Try to understand what this command is doing (type of names-**

**pace for example)**

The command sudo unshare --pid --fork --mount-proc /bin/bash is creating a new bash shell that runs in a separate set of namespaces, particularly with respect to process IDs (PID namespace) and the /proc filesystem (Mount namespace). Let's break down what each part of the command is doing:

### **Breakdown of the Command:**

1. **sudo**:
   * You are running this command with superuser privileges, which is necessary for working with namespaces.
2. **unshare**:
   * The unshare command allows a program to run in new namespaces that are different from the current set of namespaces of the calling process. Essentially, it "unshares" certain aspects (like process IDs, mount points, etc.) of the environment, creating isolated environments.
3. **--pid**:
   * This option tells unshare to create a new **PID namespace**.
   * A PID namespace provides isolation of process IDs. When you create a new PID namespace, processes inside this namespace will have their own process ID numbering, starting from 1 for the first process, similar to the root process in a typical Linux environment.
   * Processes inside the new PID namespace will see only processes within the namespace. Processes outside (on the host) can see all processes, including those inside the new PID namespace.
4. **--fork**:
   * This option forks the process after creating the new namespace.
   * This ensures that a new bash session starts within the new namespace environment. Without forking, the command would stay in the same process and would not fully utilize the new namespace.
5. **--mount-proc**:
   * This option remounts the /proc filesystem inside the new PID namespace.
   * The /proc filesystem provides information about processes running on the system. By remounting /proc, the new namespace will show the processes as they exist **within the new PID namespace**, rather than those of the host.
   * This is crucial because the /proc directory in Linux is a pseudo-filesystem that reflects the process-related information of the running system. In this isolated namespace, /proc will now only reflect the processes within that namespace.
6. **/bin/bash**:
   * This starts a new bash shell inside the new namespaces (PID and mount namespaces).

### **What Happens When You Run This Command:**

* **New PID namespace**: You create a new PID namespace, meaning the processes in this shell are isolated from those on the host. For example, if you run ps or top inside this new shell, you will see only the processes started within this namespace, and they will have their own PID numbering starting from 1. The host system can still see these processes, but they are isolated from the perspective of the new namespace.
* **New /proc mount**: The /proc filesystem is remounted, and it reflects the process state of the newly created PID namespace. Therefore, if you list the processes using ps, it will appear as if you're on a new system with only your bash process (and perhaps a few others) running, isolated from the host system's processes.

**• How can you identify if you are in an isolated environment or not?**

### **1. Check the Process ID (PID) of the Init Process**

In a new PID namespace, the init process (the first process) will typically have the PID of **1**, whereas on a host system, the init or systemd process typically has PID **1**.

**Command:**

bash

Copier le code

ps -p 1

* **In an isolated environment (new PID namespace):** The init process (usually bash or some other process) will have PID 1.
* **On the host system:** You will see systemd or init with PID 1.

### **2. Check /proc/1/sched**

The file /proc/1/sched contains scheduling information for the first process (PID 1). You can use it to check whether the first process in the namespace is systemd (which runs as PID 1 on most Linux hosts) or something else.

**Command:**

bash

Copier le code

cat /proc/1/sched

* **In an isolated environment:** You will likely see a process like bash or sh running as PID 1, indicating you're not on the host.
* **On the host system:** You will see systemd or init.

**Considering that a Linux operating system NEEDS a PID starting at 1 (this is**

**where the kernel start the initialization of the system), explain the role of this**

**namespace in the creation of containers?**

In the context of containers, the PID namespace is vital because it allows containers to have their own isolated process space, with their own init-like process (PID 1) that manages the container’s lifecycle. This namespace creates the illusion that each container is running on a separate system, even though all containers share the same host kernel. The result is lightweight, isolated environments that are central to how containers (like Docker) provide isolation, security, and efficient use of system resources.

**Using the terminal, type the command sudo unshare --net /bin/bash. Try to**

**understand what this command is doing (type of namespace)?**

### **What is a Network Namespace?**

A **network namespace** is a Linux feature that allows for the complete isolation of networking resources. This means that any process running in the new network namespace will not be able to see or interact with the network interfaces, routing tables, or network configurations of the host system.

Each network namespace has its own:

* Network interfaces (like eth0, lo)
* IP addresses
* Routing tables
* Firewall rules (like iptables)
* Network sockets (TCP/UDP)

### **What Happens When You Run sudo unshare --net /bin/bash?**

1. **Isolated Network Stack**:
   * A new network namespace is created. The bash shell (and any processes spawned by it) will now operate inside this new, isolated namespace.
   * The network interfaces, IP addresses, routing tables, and other network-related settings of the host system will **not be visible** inside this namespace.
2. **No Network Interfaces by Default**:
   * Inside the new network namespace, you typically won’t have any network interfaces available, except for a new **loopback interface (lo)**. You will not have access to the default Ethernet or Wi-Fi interfaces of the host (e.g., eth0, wlan0).
   * Initially, the loopback interface (lo) will be down, and you'll need to bring it up manually if you want to use it for local networking.
3. **No Internet Connectivity**:
   * Since the new network namespace is isolated, it will not have any external network connectivity unless you manually configure it (e.g., by setting up virtual Ethernet interfaces, bridges, or assigning an IP address).
4. **Manual Network Configuration**:
   * If you want the processes inside this new namespace to communicate with the host or the internet, you'll need to manually create and configure network interfaces, such as:
     + Creating a virtual Ethernet pair (veth).
     + Connecting one end of the virtual Ethernet to the host network.
     + Configuring IP addresses and routes.

### **Example Workflow After Running sudo unshare --net /bin/bash:**

#### **1. Check Network Interfaces:**

Inside the newly created network namespace, you can check the available interfaces using the ifconfig or ip command:

bash

Copier le code

ip a

You'll see only the loopback interface (lo), which is typically **down**:

bash

Copier le code

1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state DOWN

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

#### **2. Bring the Loopback Interface Up:**

To enable the loopback interface for local communication within the namespace:

bash

Copier le code

ip link set lo up

#### **3. Verify the Loopback is Up:**

You can verify the loopback is up by running:

bash

Copier le code

ip a

Now, lo should be shown as **UP**:

bash

Copier le code

1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

#### **4. Exit the Namespace:**

When you exit the Bash shell (by typing exit), you return to the host's default network namespace, where you have access to the usual networking interfaces like eth0 or wlan0.

### **Practical Uses of Network Namespaces:**

1. **Containers**: Containers (e.g., Docker) use network namespaces to provide isolated network stacks for each container. Each container has its own virtual network interface, IP address, and routing table, isolated from other containers and the host.
2. **Virtual Private Networks (VPNs)**: You can use network namespaces to create isolated environments where only certain processes or groups of processes use a specific network route or VPN while others remain unaffected.
3. **Testing and Development**: Developers and network engineers use network namespaces to test network configurations in a controlled environment without affecting the host system's network.
4. **Multi-Tenant Networking**: In cloud environments or large-scale deployments, network namespaces can be used to create isolated networks for different tenants or applications.

### **Summary:**

The command sudo unshare --net /bin/bash creates a new **network namespace** and launches a Bash shell within it. Inside this namespace:

* The network interfaces and configurations of the host system are not visible.
* You have an isolated loopback interface (lo), which you must manually bring up.
* Any further network configuration (e.g., external networking) must be set up manually.