# SIO Lab
# Containers

version 1.0

Author: Pedro Escaleira

Version log:

- 1.0: Initial version

## 1 Introduction

In today's dynamic world of software development and deployment, containers have emerged as a transformative technology, reshaping how applications are packaged, deployed, and managed. Containers represent a pivotal shift in how we think about software development and infrastructure provisioning. Their importance cannot be overstated, as they have become a cornerstone of modern IT practices, offering numerous advantages over traditional approaches like virtual machines.

Containers are lightweight and portable units that encapsulate an application and all its dependencies, including libraries and runtime environments. This encapsulation allows applications to run consistently and reliably across various computing environments, from a developer's laptop to production servers, without the typical "it works on my machine" frustrations.

Other benefits aside, from a security standpoint, containers offer a high degree of isolation, ensuring that an application and its dependencies are isolated from the underlying host system. This isolation promotes consistency and reproducibility across different environments. However, you should be aware that containers do not provide the same isolation as virtual machines (VMs). On the one hand, with VMs, the hardware is virtualized, and, therefore, the VM does not interact with the host operating system (OS). Instead, each VM will have its own OS, which will interact with the virtualized hardware. On the other hand, with containers, the OS is virtualized. In reality, the host's Kernel will provide all the necessary abstractions to each container, and, therefore, when a process inside a container makes, for instance, a system call, it will be handled by the host OS. Therefore, a container does not contain an entire OS. Instead, it will interact with the host's Kernel when necessary and will have all the required dependencies and OS libraries in its filesystem.

Regarding their usage, we can roughly define two types of containers: application and system containers. The first is used to package, share, and execute single applications. Consequently, they usually run a limited number of processes. System containers, on the other hand, are helpful to mimic a complete operating system. Therefore, you can use them to install and run different services, languages, and databases, as you would usually do with VMs.

Therefore, we will try out two of the most recognized containerization technologies in this class: Docker for application containers and LXD for system containers.

## 2 Docker

Docker is a platform commonly used to manage, run, and create containers. Using it, you can quickly package an application with its dependencies and configurations, share it with others, or launch it on other machines.

## 2.1 Installing and setting up

This installation procedure was adapted from the official Docker documentation for Debian. If you chosen to use another Linux distribution, you may find the corresponding instructions in the Install guide.

1. Set up Docker's Apt repository:

```
# Add Docker's official GPG key
$ sudo apt update
$ sudo apt install ca-certificates curl gnupg
$ sudo install -m 0755 -d /etc/apt/keyrings
$ curl -fsSL https://download.docker.com/linux/debian/gpg \
    | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
$ sudo chmod a+r /etc/apt/keyrings/docker.gpg

# Add the repo to Apt sources
$ echo \
    "deb [arch="$(dpkg --print-architecture)" \
    signed-by=/etc/apt/keyrings/docker.gpg] \
    https://download.docker.com/linux/debian \
    "$(. /etc/os-release && echo "$VERSION_CODENAME")" stable" | \
    sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
$ sudo apt update
```

2. Install the latest Docker packages:

```
$ sudo apt install docker-ce docker-buildx-plugin docker-compose-plugin
```

3. By default, Docker commands need to be executed by the superuser user because the Docker daemon binds to a Unix socket. If you want to run Docker commands using your user (e.g., without using `sudo`), you can add it to the `docker` group:

```
# If the 'docker' group was not created during the installation process
$ sudo groupadd docker

# To add your current user to the docker group (you may replace $USER with a specific username, if you want to add
    another user to this group)
$ sudo usermod -aG docker $USER

# You will need to log out and log back in for the group membership to be re-evaluated; however, if you want to start
    working with Docker right-away in the same session, you can force the group ID change (this will only affect the
    shell session you are using at the moment, if you open another terminal, you will need to rerun this command)
$ newgrp docker
```

4. Finally, you can check if your installation succeeded by running the Docker's `hello-world` image. This will download this image from Docker Hub (the official Docker container registry) and execute it immediately. If it is executed with success, you will find a `Hello from Docker!` message on your terminal, among other information:

```
$ docker run hello-world
```

## 2.2 Launching a simple application

After downloading this class' resources, enter the `docker-app` directory. In it, you will find a simple Flask application, its requirements, and the Dockerfile with the procedures to create a container image for this application. With your terminal in that directory, you can build the image with the following command:

```
# In this case, you are naming the image 'flask-app'
$ docker build -t flask-app .
```

This process will take some time to finish. When it does so, you can list the images you already have on your system:

```
# You should find, at least, the flask-app image you just built
$ docker image ls
REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
flask-app    latest    22ce10979060   3 minutes ago   569MB
```

With the image created, you can create a container from this image:

```
# The -p flag tells Docker to map port 8000 of your host system to the container's port 9000, where the Flask app is
     listening on;
# As for the -i flag, it is to tell Docker to launch the container in interactive mode, meaning that you will see all the
     container's logs and you can kill the container with just a CTRL-C;
# Finally, the -u flag tells Docker to use the user ID 1001 and group ID 1001 (this is a crucial step, as by default, Docker
     will use the root user to run everything inside a container)
$ docker run -p 8000:9000 -i -u 1001:1001 flask-app
```

Finally, you can access the web application on your browser using your local IP address and the port you selected (e.g., 127.0.0.1:8000).

## 2.3 Analysing some of the isolation provided by the container

With your container still running, you can now open another terminal and run the following commands:

```
# First, list all the running containers, and copy the ID of the one corresponding to the 'flask-app'
$ docker container ls
CONTAINER ID   IMAGE       COMMAND            CREATED         STATUS         PORTS                                       NAMES
3a8836c4d75c   flask-app   "python main.py"   3 minutes ago   Up 3 minutes   0.0.0.0:8000->9000/tcp, :::8000->9000/tcp
     romantic_cohen

# Then, past the container ID onto the following command (in this case, the container ID was 3a8836c4d75c)
$ docker exec -ti 3a8836c4d75c /bin/bash
```

The last command can be used to open an interactive shell for you to interact with the container directly. Instead of `/bin/bash`, you may experiment with other known commands and respective arguments, such as "echo ola". These commands will be executed on your container, and the result will appear on your terminal.

Now, going back to the `/bin/bash` command, you will be presented with a shell of your container. Open another terminal on your host to compare the results with the container's ones:

### 2.3.1 Check for the processes being executed

#### 2.3.1.1 On the container

```
$ ps auxf
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
1001           7  0.0  0.0   4188  3528 pts/0    Ss   17:04   0:00 /bin/bash
1001          17  0.0  0.1   8100  4016 pts/0    R+   17:07   0:00  \_ ps auxf
1001           1  0.1  0.7 112148 32004 ?        Ss   17:03   0:00 python main.py
```

As you can see, only three processes are running. However, one is the command you just executed, while the other corresponds to the shell you have opened. Therefore, if you do not execute anything else inside the container, the only process running is the one corresponding to your application (which, in addition, possesses the process ID (PID) 1, being the first process created inside your container).

#### 2.3.1.2 On the host

```
$ ps auxf
[...]
root       53720  0.0  0.2 720776 11004 ?        Sl   18:03   0:00 /usr/bin/containerd-shim-runc-v2 -namespace moby -id
     ad67d1aba4b48af5d8d8934c4756c83afc042dd2546c2ca591d0455523b346bd -addr
1001       53741  0.0  0.7 112148 32004 ?        Ss   18:03   0:00  \_ python main.py
1001       53795  0.0  0.0   4188  3528 pts/0    Ss+  18:04   0:00  \_ /bin/bash
```

In this case, you will get many processes, as this includes all the host's processes. Near the end, you will probably find the processes that are running inside your container (notice the user ID). As you can see, the process that launched the container's processes corresponds to the containerd runtime. This is because, underneath the hood, Docker uses the containerd runtime to manage its containers. For each container you launch using Docker, you will get a process similar to this, which has as a child processes the corresponding container's processes.

Moreover, compare the PID of the container processes with the analogous ones on your host. For instance, in this example, our Flask application has PID 1 inside the container and PID 53741 on the host. This is because Docker/containerd creates limits the view all the processes inside the container will have. A process inside a container will only be able to observe the other container's processes.

### 2.3.2 Check for the network interfaces

#### 2.3.2.1 On the container

```
$ ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
inet 127.0.0.1/8 scope host lo
   valid_lft forever preferred_lft forever
134: eth0@if135: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
```

Inside the container, we only have two interfaces: the loopback interface and an ethernet interface.

#### 2.3.2.2 On the host

```
$ ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
inet 127.0.0.1/8 scope host lo
   valid_lft forever preferred_lft forever
inet6 ::1/128 scope host
   valid_lft forever preferred_lft forever
2: enp1s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 52:54:00:10:91:84 brd ff:ff:ff:ff:ff:ff
    inet 192.168.122.154/24 brd 192.168.122.255 scope global dynamic noprefixroute enp1s0
       valid_lft 2425sec preferred_lft 2425sec
    inet6 fe80::5054:ff:fe10:9184/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:97:51:18:00 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:97ff:fe51:1800/64 scope link
       valid_lft forever preferred_lft forever
135: vethc21adc3@if134: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP group default
    link/ether 92:4a:cc:a1:9a:3c brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::904a:ccff:fea1:9a3c/64 scope link
       valid_lft forever preferred_lft forever
```

All interfaces on the host are distinct from the container's. However, the `vethc21adc3` (in your machine, it will have a name also starting with `veth`) is a "special interface," belonging to a virtual ethernet device (veth), which Docker created. You can think of a veth as a virtual ethernet cable between two interfaces: in this case, one of them is on the host (in this example, named `vethc21adc3`), while the other is on the container (in this example, it was the `eth0`). Therefore, the container does not have access to the host's network. Instead, for the necessary network connections to reach the container, everything goes through the host's veth interface, ending up on the container on the eth0 interface. Similarly, any response from the container will be sent to the `eth0` interface and end up on the host's corresponding veth interface.

#### 2.3.3 Check for the filesystem

##### 2.3.3.1 On the container

You can verify that inside the container, your filesystem is distinct from your host's. If we analyze the root directory (`$ ls /`), we can check that the container has all the directories and files needed to run a Linux distribution (in our case, Debian). Furthermore, we have the `/app` directory, which was the directory we instructed Docker to save our application code and dependencies.

##### 2.3.3.2 On the host

As with the network and processes, a container's filesystem is also present on the host, which shares it with the container. Therefore, we can access the complete container filesystem through the host filesystem. To achieve that, we can execute the following command:

```
# You will need to obtain, once again the container's ID
$ docker container inspect 1d40ae6f2809
[
    [...]
    "GraphDriver": {
        "Data": {
            "LowerDir":
     "/var/lib/docker/overlay2/5d7d113ece16709db63862c379e6e355e7b9363c2b4bf5c37aa08189d30dd9bd-init/diff:[...]",
            "MergedDir": "/var/lib/docker/overlay2/5d7d113ece16709db63862c379e6e355e7b9363c2b4bf5c37aa08189d30dd9bd/merged",
            "UpperDir": "/var/lib/docker/overlay2/5d7d113ece16709db63862c379e6e355e7b9363c2b4bf5c37aa08189d30dd9bd/diff",
            "WorkDir": "/var/lib/docker/overlay2/5d7d113ece16709db63862c379e6e355e7b9363c2b4bf5c37aa08189d30dd9bd/work"
        },
        "Name": "overlay2"
    },
    [...]
]
```

In our scenario, the most relevant directory of the ones listed is the `MergedDir`. If we access it, we will find the directories and files we initially found when using the container's shell. Once again, Docker limits the view the processes inside the container have, in this case, in relation to the host's filesystem.

### 2.4 Cleaning and uninstalling

Since we will no longer use Docker in these practical classes, you should clean all the resources it created, as well as uninstall it. Moreover, your LXD containers will have network issues if you still have Docker installed on the same machine, so you should uninstall Docker before using LXD and reboot your virtual machine:

```
# Just to be sure, kill any potential running container
$ docker kill $(docker container ls | tail -n +2 | awk '{print $1}')

# Clean all the Docker created resources
$ docker system prune -a

# Uninstall Docker
$ sudo apt purge --purge --auto-remove docker-ce docker-buildx-plugin docker-compose-plugin iptables pigz
$ sudo rm -rf /var/lib/docker
$ sudo rm -rf /var/lib/containerd

# Reboot the machine, critical
$ sudo reboot
```

## 3 LXD

### 3.1 Installing and setting up

This installation procedure was adapted from the Debian wiki LXD documentation. Furthermore, LXD LST is only supported from Debian 12 onwards. If you choose to use another Linux distribution or Debian version, you may need to search for a different installation process.

1. Installation:

```
$ sudo apt install lxd
```

2. Initial configuration:

```
# You will be presented with a multitude of configuration choices. However, for these classes, you can select all the
    default configurations (just hit enter on all questions).
$ sudo lxd init
```

3. As with Docker, by default, LXD commands need to be executed by the superuser user because the LXD binds to a Unix socket. If you want to run LXD commands using your user (e.g., without using `sudo`), you can add it to the `lxd` group:

```
$ sudo usermod -aG lxd $USER

# As with Docker, you will need to log out and log back in for the changes to take place. However, you can force your
    shell session to use the new group with the following command
$ newgrp lxd
```

## 3.2 Launching a simple application

With LXD installed, we can start by creating a container. Unlike Docker, LXD allows us to create containers resembling a virtual machine. Therefore, we can begin by launching a container based on a Linux distribution image. In this class, we will use an Ubuntu image (notice that we are using a Linux distribution different from the host's — this is possible because containers only share the Linux Kernel, everything else can be different):

```
# First, let's find an Ubuntu image. We could also search for other Linux distributions or versions by changing the filter
    entry (in this command, we are using the filter "ubuntu/jammy/amd64")
$ lxc image list images: ubuntu/jammy/amd64 -c lfdat
+----------------------+--------------+-------------------------------------+--------------+-----------------+
|         ALIAS        | FINGERPRINT  |             DESCRIPTION             | ARCHITECTURE |      TYPE       |
+----------------------+--------------+-------------------------------------+--------------+-----------------+
| ubuntu/jammy (7 more) | 656a75cf6d0d | Ubuntu jammy amd64 (20230917_07:42) | x86_64       | CONTAINER       |
+----------------------+--------------+-------------------------------------+--------------+-----------------+
| ubuntu/jammy (7 more) | b007bcadd52a | Ubuntu jammy amd64 (20230917_07:42) | x86_64       | VIRTUAL-MACHINE |
+----------------------+--------------+-------------------------------------+--------------+-----------------+

# Create and Launch a container called 'aula2' based on this image
$ lxc launch images:ubuntu/jammy aula2

# List your existing containers and verify that the newly created container is there
$ lxc list
+-----------------+---------+---------------------+-----------------------------------------------+-----------+-----------+
|       NAME      |  STATE  |        IPV4         |                     IPV6                      |   TYPE    | SNAPSHOTS |
+-----------------+---------+---------------------+-----------------------------------------------+-----------+-----------+
| aula2           | RUNNING | 10.72.250.44 (eth0) | fd42:6e3:8589:2d10:216:3eff:fe1e:39c8 (eth0) | CONTAINER | 0         |
+-----------------+---------+---------------------+-----------------------------------------------+-----------+-----------+
```

With the container created, you can now interact with it. As with Docker, you can start an interactive shell to that container:

```
$ lxc shell aula2
```

By interacting with it, you may verify that this container possesses a complete operating system running without an application (as with Docker). If you want to execute an application in this container, you will do steps closer to what you would do with a virtual machine. For example, if we consider the Flask application we launched with Docker:

```
# You need to be in the directory where you downloaded this class' resources
$ lxc file push -r main.py requirements.txt aula2/home/ubuntu

# Enter the container's shell
$ lxc shell aula2
```

```
# Install Python virtual environment
$ apt install python3-virtualenv

# Activate the Python virtual environment
$ su ubuntu             # it is a good practice to use an unprivileged user to run publicly available services
$ cd && ls
$ virtualenv venv && source venv/bin/activate

# Install application dependencies and launch it
$ pip install -r requirements.txt
$ python main.py
```

Finally, in your host, you can access the application by entering the IP address of your container (previously obtained with the `lxc list`) and the application port (9000).

## 3.3 Analysing some of the isolation provided by the container

As with the Docker example, open two terminals. Enter the container's shell in one, and the other will be used to interact with the host. Once again, we will compare both environments:

### 3.3.1 Check for the processes being executed

#### 3.3.1.1 On the container

```
$ ps auxf
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         195  0.0  0.0  10624  3992 pts/1    Ss   17:14   0:00 /bin/bash
root         197  0.0  0.0  13060  3160 pts/1    R+   17:16   0:00  \_ ps auxf
root           1  0.0  0.2 165412 10680 ?        Ss   17:06   0:00 /sbin/init
root         117  0.0  0.3  31332 14464 ?        Ss   17:06   0:00 /lib/systemd/systemd-journald
root         151  0.0  0.1  21432  5384 ?        Ss   17:06   0:00 /lib/systemd/systemd-udevd
systemd+     160  0.0  0.2  16116  8332 ?        Ss   17:06   0:00 /lib/systemd/systemd-networkd
root         165  0.0  0.0   9492  1256 ?        Ss   17:06   0:00 /usr/sbin/cron -f -P
message+     166  0.0  0.1   8428  4680 ?        Ss   17:06   0:00 @dbus-daemon --system --address=systemd: --nofork
     --nopidfile --syst
root         169  0.0  0.4  34304 19064 ?        Ss   17:06   0:00 /usr/bin/python3 /usr/bin/networkd-dispatcher
     --run-startup-triggers
syslog       170  0.0  0.1 152764  7004 ?        Ssl  17:06   0:00 /usr/sbin/rsyslogd -n -iNONE
root         171  0.0  0.1  14900  6364 ?        Ss   17:06   0:00 /lib/systemd/systemd-logind
systemd+     176  0.0  0.3  25528 12712 ?        Ss   17:06   0:00 /lib/systemd/systemd-resolved
root         182  0.0  0.0   8396  1080 pts/0    Ss+  17:06   0:00 /sbin/agetty -o -p -- \u --noclear --keep-baud console
     115200,38400,
```

If you compare the processes being executed in this container with the Docker container, there are a lot more in this case. This is because a system container will run just the necessary processes to launch the operating system (in this case, Ubuntu). Moreover, in this container, the first process launched (the one with PID 1) was the init daemon, responsible for bootstrapping the user space. On the other hand, on the Docker container, the first process was right away our application, as it does not launch an operating system.

#### 3.3.1.2 On the host

```
$ ps auxf
[...]
root        1182  0.0  0.4 1284532 19800 ?       Ss   18:06   0:00 [lxc monitor] /var/lib/lxd/containers aula2
165536      1193  0.0  0.2 165412 10680 ?        Ss   18:06   0:00  \_ /sbin/init
165536      1328  0.0  0.3  31332 12440 ?        Ss   18:06   0:00      \_ /lib/systemd/systemd-journald
165536      1363  0.0  0.1  21432  5384 ?        Ss   18:06   0:00      \_ /lib/systemd/systemd-udevd
165636      1372  0.0  0.2  16116  8332 ?        Ss   18:06   0:00      \_ /lib/systemd/systemd-networkd
165536      1377  0.0  0.0   9492  1256 ?        Ss   18:06   0:00      \_ /usr/sbin/cron -f -P
165638      1378  0.0  0.1   8428  4680 ?        Ss   18:06   0:00      \_ @dbus-daemon --system --address=systemd: --nofork
     --nopidfil
165536      1381  0.0  0.4  34304 19016 ?        Ss   18:06   0:00      \_ /usr/bin/python3 /usr/bin/networkd-dispatcher
     --run-startup-
165640      1382  0.0  0.1 152764  5016 ?        Ssl  18:06   0:00      \_ /usr/sbin/rsyslogd -n -iNONE
165536      1383  0.0  0.1  14900  6364 ?        Ss   18:06   0:00      \_ /lib/systemd/systemd-logind
```

```
165637      1388  0.0  0.3  25528 12712 ?        Ss   18:06   0:00      \_ /lib/systemd/systemd-resolved
165536      1395  0.0  0.0   8396  1080 pts/0    Ss+  18:06   0:00      \_ /sbin/agetty -o -p -- \u --noclear --keep-baud
    console 11520
```

Like with Docker, we have complete observability over the processes running inside the container on the host. However, as previously, the PIDs on the host are distinct from the container's. Furthermore, instead of containerd launching the first container process, in LXD is the LXC monitor.

Another difference between LXD and Docker, in terms of processes, is the effective user ID of each process. In this case, you may notice that the processes' user IDs are different between the container and the host (contrary to Docker). Once again, this is an additional sandboxing measure that containers can provide, which is present, by default, in LXD. Therefore, there is a mapping between the user ID inside and outside the container. Consequently, for the host, the effective owner of each container's process will be the one observed on the host and not the one on the container. Although this feature is not in Docker by default, it can easily be activated.

### 3.3.2 Check for the network interfaces

#### 3.3.2.1 On the container

```
$ ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
inet 127.0.0.1/8 scope host lo
   valid_lft forever preferred_lft forever
inet6 ::1/128 scope host
   valid_lft forever preferred_lft forever
4: eth0@if5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 00:16:3e:1e:39:c8 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.72.250.44/24 metric 100 brd 10.72.250.255 scope global dynamic eth0
       valid_lft 2013sec preferred_lft 2013sec
    inet6 fd42:6e3:8589:2d10:216:3eff:fe1e:39c8/64 scope global mngtmpaddr noprefixroute
       valid_lft forever preferred_lft forever
    inet6 fe80::216:3eff:fe1e:39c8/64 scope link
       valid_lft forever preferred_lft forever
```

Similarly to Docker, inside the container, we only have two interfaces: the loopback interface and an ethernet interface.

#### 3.3.2.2 On the host

```
$ ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
inet 127.0.0.1/8 scope host lo
   valid_lft forever preferred_lft forever
inet6 ::1/128 scope host noprefixroute
   valid_lft forever preferred_lft forever
2: enp1s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 52:54:00:10:91:84 brd ff:ff:ff:ff:ff:ff
    inet 192.168.122.154/24 brd 192.168.122.255 scope global dynamic noprefixroute enp1s0
       valid_lft 3566sec preferred_lft 3566sec
    inet6 fe80::5054:ff:fe10:9184/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: lxdbr0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 00:16:3e:36:6b:27 brd ff:ff:ff:ff:ff:ff
    inet 10.72.250.1/24 scope global lxdbr0
       valid_lft forever preferred_lft forever
    inet6 fd42:6e3:8589:2d10::1/64 scope global
       valid_lft forever preferred_lft forever
    inet6 fe80::216:3eff:fe36:6b27/64 scope link
       valid_lft forever preferred_lft forever
5: vethad171874@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master lxdbr0 state UP group default qlen 1000
    link/ether aa:c3:70:6d:87:3e brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

Once again, upon the container creation and launching, the manager (in this case, LXD), created a virtual ethernet device (veth). Therefore, the veth interface on the host is paired with the eth0 container's interface. With this device in action, the container does not have access to the host's network. Instead, any necessary connection has to go through the virtual device.

### 3.3.3 Check for the filesystem

#### 3.3.3.1 On the container

As with the Docker container, inside this container, the filesystem is distinct from your host's. Once again, you may analyze the root directory and verify that it corresponds to a typical Linux distribution.

#### 3.3.3.2 On the host

As with the network and processes, a container's filesystem is also present on the host, which shares it with the container. Therefore, we can access the complete container filesystem through the host filesystem. To achieve that, we can execute the following command:

```
# You can access the container filesystem in the following location.
# Note that this path might be different if you used another installation method. Furthermore, this location supposes you
     named your container 'aula2' (meaning that you should adapt it otherwise)
$ cd /var/lib/lxd/storage-pools/default/containers/aula2/rootfs
```

Similarly to Docker, if we access this directory, we will find the directories and files we initially found when using the container's shell. Once again, LXD limits the view the processes inside the container have, in this case, in relation to the host's filesystem.

## 3.4 Cleaning

Since we will continue to use LXD in some of the following classes, you can maintain it on your virtual machine. However, you no longer need the container you created in this class. Therefore, you can clean your environment with the following commands:

```
$ lxc stop aula2 && lxc delete aula2
```