
Learn Docker

Ákos Takács

Apr 01, 2024

INTRO:

1	Getting started	3
1.1	Docker CE vs Docker Desktop	3
1.2	Requirements	3
1.3	Clone the git repository	4
1.4	Scripts	5
1.5	Example projects	5
2	LXD	7
2.1	Install LXD 4.0 LTS	7
2.2	Remote repositories	9
2.3	Search for images	9
2.4	Show image information	9
2.5	Start Ubuntu 20.04 container	9
2.6	List LXC containers	9
2.7	Enter the container	10
2.8	Delete the container	10
2.9	Start Ubuntu 20.04 VM	10
3	Docker	11
3.1	System information	11
3.2	Run a stateless DEMO application	11
3.3	Play with the “hello-world” container	11
3.4	Start an Ubuntu container	13
3.5	Start Apache HTTPD webserver	14
3.6	Start Ubuntu virtual machine	17
4	Start a simple web server with mounted document root	19
5	Build your own web server image and copy the document root into the image	21
6	Create your own PHP application with built-in PHP web server	23
7	Create a simple Docker Compose project	25
8	Communication of PHP and Apache HTTPD web server with the help of Docker Compose	27
9	Run multiple Docker Compose projects on the same port using nginx-proxy	29
10	Protect your web server with HTTP authentication	35
11	Memory limit test in a Bash container	39

11.1	Files	39
11.2	Description	40
11.3	Start the test	40
11.4	Explanation of the parameters	40
12	CPU limit test	41
12.1	Files	41
13	Learn what EXPOSE is and when it matters	43
13.1	Intro	43
13.2	Accessing services from the host using the container's IP address	44
13.3	Using user-defined networks to access services in containers	45
13.4	What is the connection between port forwards and exposed ports?	48
14	Docker network and network namespaces in practice	49
14.1	Linux Kernel Namespaces in general	49
14.2	Network traffic between a container and the outside world	52
14.3	Manipulating network namespaces	57
14.4	Testing a web-based application without internet in a container	66
14.5	Used sources	73
14.6	Recommended similar tutorials	73
15	Everything about Docker volumes	75
15.1	Intro	75
15.2	Where does Docker store data?	75
15.3	What is a Docker volume?	76
15.4	Custom volume path	77
15.5	Docker CE volumes on Linux	80
15.6	Docker Desktop volumes	81
15.7	Editing files on volumes	88
15.8	Conclusion	91



My name is Ákos Takács and I really like working with Docker containers and containers in general. I decided to share my experience so I created this page to help you through your journey in learning Docker and related tools and concepts. I am a moderator on forums.docker.com, I'm on YouTube ([@itsziget](#) (Hungarian), [@akos.takacs](#) (English)), and you can follow me on [Twitter](#)

In this project I share explanations, examples and scripts. The examples were originally made for the participants of the Ipszilon Seminar in 2017 in Hungary. The virtual machines were created in the Cloud For Education system. Since then I added more contents and I will continue to do so.

Before you start working with the example projects, read *Getting started*.

GETTING STARTED

1.1 Docker CE vs Docker Desktop

Very important to know that Docker CE often referred to as Docker Engine is not the same as Docker Desktop. Docker Desktop adds new features for development purposes, but it runs a virtual machine (yes, even on Linux) so you will lose some features that you would be able to use with Docker CE.

This tutorial will mostly use Docker CE, but you can use Docker Desktop depending on your needs, however, some of the examples might not work. Whenever an example requires Docker Desktop, it will be noted before the example.

1.2 Requirements

1.2.1 Install Docker

Docker can be installed from multiple sources. Not all sources are officially supported by Docker Inc. It is recommended to [follow the official way](#) whenever it is possible. Docker is not supported on all Linux distributions, although some distributions have their own way to install Docker or Docker Desktop. Keep in mind that in those cases it is likely that the developers of Docker or people on the Docker forum can't help you and you need to rely on the community of the distribution you have chosen.

Docker CE

Docker Community Edition. It is free to use even for commercial projects, although it does not have commercial support. Docker CE is open source and the source code of the daemon is available on GitHub in [moby/moby](#). The source code of the client is in [docker/cli](#). Installation instructions for Linux containers in the official documentation: [engine/install/#server](#). For Windows containers you can follow the instructions of Microsoft: [Get Started: Prep Windows for Containers](#). This is the recommended way to run containers natively on the preferred operating system.

Docker EE

Docker Enterprise Edition doesn't exist anymore. It was the enterprise version of Docker with commercial support until Mirantis bought Docker EE. See below.

Mirantis Container Runtime

[Mirantis Container Runtime](#) used for [Mirantis Kubernetes Engine](#). If you need commercial support (from Mirantis) this is the recommended way.

Docker Desktop

Docker Desktop was created for two purposes:

- It provides developer tools using a virtualized environment based on [LinuxKit](#).
- Running native Linux containers is not possible on Windows or macOS, only on Linux. Since Docker Desktop runs Docker CE in a virtual machine Docker Inc can support running Docker containers on macOS

and Windows. Docker Inc is doing their best to make you feel you are running native Docker containers, but you need to keep in mind that you are not.

Docker Desktop is not open source, even though LinuxKit is. You can use it on your computer for personal purposes, and it is free for small companies too. Check the official documentation for up-to-date information about whether it is free for you or not. [Docker Desktop](#). At the time of writing this tutorial the short version of Docker Desktop terms is the following:

“Commercial use of Docker Desktop in larger enterprises (more than 250 employees OR more than \$10 million USD in annual revenue) requires a paid subscription.”

Rancher Desktop

Rancher Desktop is similar to Docker Desktop, although it is created for running specific version of Kubernetes. You can use it for running Docker containers, but you will not be able to use the developer tools of Docker Desktop. For the installation instructions, follow the official documentation: [Rancher Desktop](#) If you want to know more about using Rancher Desktop with Docker Desktop on the same macOS machine, you can watch my Youtube video: [Docker Desktop and Rancher Desktop on the same macOS machine](#)

1.2.2 Install Docker Compose v2

Docker Compose v2 is Docker CLI plugin to run Docker Compose projects. This is the recommended way to use Docker Compose. Since Docker Compose could already be on your system after installing Docker, check the below command first:

```
docker compose version
```

Use space between “docker” and “compose”. If the “compose” subcommand doesn’t work, follow the official documentation for the installation instructions: [Install Docker Compose](#)

1.2.3 jq for parsing json files

Some of the examples will use jq. Click the link for the installation instruction.

1.2.4 Operating system

Linux is always supported and I do my best to support Docker Desktop on macOS, but not all the examples will work on macOS unless you run Linux in a virtual machine.

1.3 Clone the git repository

```
git clone https://github.com/itsziget/learn-docker.git
```

1.4 Scripts

system/usr/local/bin/nip.sh

nip.io generates domain names for the public DNS server based on the current WAN or LAN IP address of the host machine. It must be copied into `/usr/local/bin/` with the filename “`nip.sh`”. When you execute “`nip.sh`”, a domain name will be shown (Ex.: `192.168.1.2.nip.io`) which you can use for the examples. The command takes one optional parameter as a subdomain. Ex.: “`nip.sh web1`”. The result would be: `web1.192.168.1.2.nip.io`

system/etc/profile.d/nip.variable.sh

It uses the `nip` command to set the NIP environment variable so you can use the variable in a `docker-compose.yml` too.

Make sure you each script is executable before you continue. However, the above scripts are optional and you may not need them in a local virtual machine. If you don't want to rely on automatic IP address detection, set the NIP variable manually.

1.5 Example projects

Example projects are in the *learn-docker/projects* folder, so go to there.

Check the existence of `$NIP` variable since you will need it for some examples:

If it does not exist or empty, then set the value manually or run the script below:

All off the examples were tested with Docker 20.10.23. The version of Docker Compose was 2.15.1. You can try with more recent versions but some behaviour could be different in the future.

Before using Docker containers it's good to know a little about a similar tool. LXD can run containers and also virtual machines with similar commands. It uses LXC to run containers (as Docker did at the beginning) and Qemu-KVM to run virtual machines. To install LXD 4.0 LTS you need [snap](#).

2.1 Install LXD 4.0 LTS

```
sudo snap install --channel 4.0/stable lxd
```

Now you need to initialize the configuration:

```
lxd init
```

You will find the following questions:

1. **Question:** Would you like to use LXD clustering? (yes/no) [default=no]:
Answer: no
2. **Question:** Do you want to configure a new storage pool? (yes/no) [default=yes]:
Answer: yes
3. **Question:** Name of the new storage pool [default=default]
Answer: default
4. **Question:*** Name of the storage backend to use (btrfs, dir, lvm, zfs, ceph) [default=zfs]:
Answer: zfs
5. **Question:** Create a new ZFS pool? (yes/no) [default=yes]:
Answer: yes
6. **Question:** Would you like to use an existing empty block device (e.g. a disk or partition)? (yes/no) [default=no]:
Answer: no
7. **Question:** Size in GB of the new loop device (1GB minimum) [default=25GB]:
Answer: Choose a suitable size for you depending on how much space you have.
8. **Question:** Would you like to connect to a MAAS server? (yes/no) [default=no]:
Answer: no
9. **Question:** Would you like to create a new local network bridge? (yes/no) [default=yes]:
Answer: yes
10. **Question:** What should the new bridge be called? [default=lxdbr0]:

Answer: lxdbr0

11. **Question:** What IPv4 address should be used? (CIDR subnet notation, “auto” or “none”) [default=auto]:

Answer: auto

12. **Question:** What IPv6 address should be used? (CIDR subnet notation, “auto” or “none”) [default=auto]:

Answer: none

13. **Question:** Would you like LXD to be available over the network? (yes/no) [default=no]:

Answer: no

14. **Question:** Would you like stale cached images to be updated automatically? (yes/no) [default=yes]

Answer: no

15. **Question:** Would you like a YAML “lxd init” preseed to be printed? (yes/no) [default=no]:

Answer: Optional. Type “yes” if you want to see the result of the initialization.

Output:

```
config:
  images.auto_update_interval: "0"
networks:
- config:
  ipv4.address: auto
  ipv6.address: none
  description: ""
  name: lxdbr0
  type: ""
  project: default
storage_pools:
- config:
  size: 25GB
  description: ""
  name: default
  driver: zfs
profiles:
- config: {}
  description: ""
  devices:
  eth0:
  name: eth0
  network: lxdbr0
  type: nic
  root:
  path: /
  pool: default
  type: disk
  name: default
cluster: null
```

2.2 Remote repositories

There are multiple available remote repositories to download base images. For example: <https://images.linuxcontainers.org>

You can list all of them with the following command:

```
lxc remote list
```

2.3 Search for images

Pass <reponame>:<keywords> to `lxc image list`

```
lxc image list images:ubuntu
# or
lxc image list images:ubuntu focal
# or
lxc image list images:ubuntu 20.04
# or
lxc image list ubuntu:20.04
```

2.4 Show image information

To show information about a specific image use `lxc image info` with <reponame>:<knownalias>

```
lxc image info ubuntu:f
```

Aliases are the names of the images with which you can refer to a specific image. One image can have multiple aliases. The previous command's output is a valid YAML so you can use `yq` to process it.

```
lxc image info ubuntu:focal | yq '.Aliases'
```

2.5 Start Ubuntu 20.04 container

```
lxc launch ubuntu:20.04 ubuntu-focal
```

2.6 List LXC containers

```
lxc list
```

2.7 Enter the container

```
lxc exec ubuntu-focal bash
```

Then just use `exit` to quit the container.

2.8 Delete the container

```
lxc delete --force ubuntu-focal
```

2.9 Start Ubuntu 20.04 VM

You can even create a virtual machine instead of container if you have at least LXD 4.0 installed on your machine.

```
lxc launch --vm ubuntu:20.04 ubuntu-focal-vm
```

It will not work on all machines, only when Qemu KVM is supported on that machine.

3.1 System information

```
docker help
docker info
docker version
docker version --format '{{json .}}' | jq # requires jq installed
docker version --format '{{.Client.Version}}'
docker version --format '{{.Server.Version}}'
docker --version
```

3.2 Run a stateless DEMO application

```
docker run --rm -p "8080:80" itsziget/phar-examples:1.0
# Press Ctrl-C to quit
```

3.3 Play with the “hello-world” container

3.3.1 Start “hello-world” container

```
docker run hello-world
# or
docker container run hello-world
```

Output:

```
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
   (amd64)
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
```

(continues on next page)

(continued from previous page)

4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, and more with a free Docker ID:

<https://hub.docker.com/>

For more examples and ideas, visit:

<https://docs.docker.com/get-started/>

3.3.2 List containers

List running containers

```
docker ps
# or
docker container ps
# or
docker container ls
# or
docker container list
```

List all containers

```
docker ps -a
# or use the other alias commands
```

List containers based on the hello-world image:

```
docker ps -a -f ancestor=hello-world
# or
docker container list --all --filter ancestor=hello-world
```

3.3.3 Delete containers

Delete a stopped container

```
docker rm containername
# or
docker container rm containername
```

Delete a running container:

```
docker rm -f containername
```

If the generated name of the container is “angry_shaw”

```
docker rm -f angry_shaw
```

3.3.4 Start a container with a name

```
docker run --name hello hello-world
```

Running the above command again results an error message since “hello” is already used for the previously started container. Run the following command to check the stopped containers:

```
docker ps -a
```

Or you can start the stopped container again by using its name:

```
docker start hello
```

The above command will display the name of the container. You need to start it in “attached” mode in order to see the output:

```
docker start -a hello
```

Delete the container named “hello”

```
docker rm hello
```

3.3.5 Start a container and delete it automatically when it stops

```
docker run --rm hello-world
```

3.4 Start an Ubuntu container

3.4.1 Start Ubuntu in foreground (“attached” mode)

```
docker run -it --name ubuntu-container ubuntu:20.04
```

Press **Ctrl+P** and then **Ctrl+Q** to detach from the container or type **exit** and press **enter** to exit bash and stop the container.

3.4.2 Start Ubuntu in background (“detached” mode)

Linux distribution base Docker images usually don’t contain Systemd as LXD images so these containers cannot run in background unless you pass **-it** to get interactive terminal. It wouldn’t be necessary with a container which has a process inside running in foreground continuously. **-it** works with other containers too as long as the containers command is “bash” or some other shell.

```
docker rm -f ubuntu-container
docker run -it -d --name ubuntu-container ubuntu:20.04
```

Note: Actually only **-i** or **-t** would be enough to keep the container in the background, but if you want to attach the container later, it requires both of them. Of course, **-d** is always required.

3.4.3 Attach the container

You can attach the container and see the same as you could see when you run a container without `-d`, in foreground. You can even interact with the container's main process so be careful and don't execute a command like `exit`, or you will stop the whole container by stopping its main process.

```
docker attach ubuntu-container
```

Press `Ctrl+P` and then `Ctrl+Q` to quit without stopping the container.

The better way to “enter” the container is `docker exec` which is similar to the way of LXD.

```
docker exec -it ubuntu-container
```

Now you can use the “`exit`” command to quit the container and leave it running.

3.5 Start Apache HTTPD webserver

3.5.1 Start the container in the foreground

```
docker run --name web httpd:2.4
```

There will be no prompt until you press “`CTRL+C`” to stop the container running in the foreground.

Note: When you change your terminal window it will send `SIGWINCH` signal to the container and shut down the server. Use it only for some quick test.

3.5.2 Start it in the background

```
docker rm web
docker run -d --name web httpd:2.4
```

Note: You don't need to use `-it` and you should not use that either. Running HTTPD server container with and interactive terminal will send `SIGWINCH` signal to the container and shut down the HTTPD server immediately when you try to attach it.

Even without `-it`, attaching the HTTPD server container will shut down the server when you change the size of your terminal window.

Use `docker logs` instead.

3.5.3 Check container logs

`docker logs` shows the standard error and output of a container without attaching it. Actually it will read and show the content of the log file which was saved from the container's output.

```
docker logs web
# or
docker container logs web
```

Watch the output (logs) continuously

```
docker logs -f web
# Press Ctrl-C to stop watching
```

3.5.4 Open the webpage using an IP address

Get the IP address:

```
CONTAINER_IP=$(docker container inspect web --format '{{.NetworkSettings.IPAddress}}')
```

You can test if the server is working using `wget`:

```
wget -qO- $CONTAINER_IP
```

Output:

```
<html><body><h1>It works!</h1></body></html>
```

3.5.5 Use port forwarding

Delete the container named “web” and forward the port 8080 from the host to the containers internal port 80:

```
docker rm -f web
docker run -d -p "8080:80" --name web httpd:2.4
```

Then you can access the page using the host's IP address.

3.5.6 How we could enter a container in the past

Before `docker exec` was introduced, `nsenter` was the only way to enter a container. It does almost the same as `docker exec` except it does not support Pseudo-TTY so some commands may not work.

```
CONTAINER_PID=$(docker container inspect --format '{{.State.Pid}}' web)

sudo nsenter \
  --target $CONTAINER_PID \
  --mount \
  --uts \
  --ipc \
  --net \
```

(continues on next page)

(continued from previous page)

```
--pid \  
--cgroup \  
--wd \  
env -i - $(sudo cat /proc/$CONTAINER_PID/environ | xargs -0) bash
```

As you can see, `nsenter` runs a process inside specific Linux namespaces.

3.5.7 Share namespaces

```
docker rm -f web  
docker run -d --name web \  
  --net host \  
  --uts host \  
  --pid host \  
  httpd:2.4
```

This example shows how you can share the host's namespaces with the container.

- **net:** The container will not get a virtual network. `localhost` inside the container will be the same as `localhost` on the host operating system.
- **uts:** When you enter the container you will see that the hostname in the prompt is the same as you can see on the host. Without this, the container had a random hash as hostname.
- **pid:** The container can see every process running on the host and not just inside the container.

Note: Using [user namespace in a Docker container](#) is disabled by default

Note: Since Docker Desktop runs Docker CE in a virtual machine, sharing namespaces with the host means that you will use the namespace of the virtual machine, not the actual host operating system where you run the Docker client. As a result, you will still not be able to access ports on `localhost` of the host operating system, the hostname will be the hostname of the virtual machine and the processes inside the container will see the processes running inside the virtual machine only.

Now enter the container

```
docker exec -it web bash
```

and install the following tools, so you can see host processes and network interfaces from the container.

```
apt update  
apt install iproute2 procps psmisc
```

- **iproute2:** adds the `ip` command
- **procps:** installs the `ps` command
- **psmisc:** this makes `pstree` command available

Now run

- `ip addr` to see network interfaces

- `ps auxf` to see host processes
- `pstree` to see the process tree

You can exit the container and run the following command to get only the processes inside the container:

```
docker exec web ps auxf $(docker container inspect --format '{{ .State.Pid }}' web)
```

3.6 Start Ubuntu virtual machine

There are multiple ways to run a virtual machine with Docker. Using a parameter is not enough. You need to choose a different runtime. The default is `runc` which runs containers. One of the most popular and easiest runtime is [Kata Containers](#).

Follow the instructions to install the latest stable version of the Kata runtime: [Install Kata Containers](#) and configure Docker daemon to use it. An example `/etc/docker/daemon.json` is the following:

```
{
  "default-runtime": "runc",
  "runtimes": {
    "kata": {
      "path": "/usr/bin/kata-runtime"
    }
  }
}
```

Now run

```
docker run -d -it --runtime kata --name ubuntu-vm ubuntu:20.04
```

It is still lightweight. You can run `ps aux` inside to see there is no `systemd` or other process like that, however, run the following command on the host machine and see it has only one CPU core:

```
docker exec -it ubuntu-vm cat /proc/cpuinfo
```


START A SIMPLE WEB SERVER WITH MOUNTED DOCUMENT ROOT

Note: *Clone the git repository* if you haven't done it yet.

Go to Project 1 from the git repository root:

```
cd projects/p01
```

Project structure:

```
.
├── www
│   └── index.html
```

This project contains only one folder, “www” and an `index.html` in it with the following line:

```
Hello Docker (p01)
```

Start the container and mount “www” as document root:

```
docker run -d -v $(pwd)/www:/usr/local/apache2/htdocs:ro --name p01_httpd -p "8080:80" \
↳ httpd:2.4
# or
docker run -d --mount type=bind,source=$(pwd)/www,target=/usr/local/apache2/htdocs,
↳ readonly --name p01_httpd -p "8080:80" httpd:2.4
```

Generate a domain name:

```
nip.sh
# example output:
# 192.168.1.2.xip.io
```

In case are working in the cloned repository of this tutorial, you can also run the below command to set the variable

```
../../system/usr/local/bin/nip.sh
```

Test the web page:

```
http://192.168.1.2.nip.io:8080
```

Now you should see the content of the mounted `index.html`

Delete the container to make port 8080 free again.

```
docker rm -f p01_httpd
```

BUILD YUR OWN WEB SERVER IMAGE AND COPY THE DOCUMENT ROOT INTO THE IMAGE

Note: *Clone the git repository* if you haven't done it yet.

Go to Project 2 from the git repository root:

```
cd projects/p02
```

Project structure:

```
.
├── Dockerfile
└── www
    └── index.html
```

The content of the html file

```
Hello Docker (p02)
```

and the Dockerfile

```
FROM httpd:2.4
LABEL hu.itsziget.ld.project=p02
COPY www /usr/local/apache2/htdocs
```

Building an image:

```
docker build -t localhost/p02_httpd .
```

The dot character at the end of the line is important and required.

Start container:

```
docker run -d --name p02_httpd -p "80:80" localhost/p02_httpd
```

You can open the website from a web browser on port 80. The output should be “Hello Docker (p02)”

Delete the container to make port 8080 free again.

```
docker rm -f p02_httpd
```

CREATE YOUR OWN PHP APPLICATION WITH BUILT-IN PHP WEB SERVER

Note: *Clone the git repository* if you haven't done it yet.

Go to Project 3 from the git repository root:

```
cd projects/p03
```

Project structure:

```
.
├── Dockerfile
└── www
    └── index.php
```

The content of the index.php

```
<?php
file_put_contents(__DIR__ . '/access.txt', date('Y.m.d. H:i:s') . "\n", FILE_APPEND);

echo 'P03<br/>';
echo nl2br(file_get_contents(__DIR__ . '/access.txt'));
```

and the Dockerfile

```
FROM php:7.4-alpine
LABEL hu.itsziget.ld.project=p03
COPY www /var/www
CMD ["php", "-S", "0.0.0.0:80", "-t", "/var/www"]
# CMD php -S 0.0.0.0:80 -t /var/www
# is the same as
# CMD ["sh", "-c", "php -S 0.0.0.0:80 -t /var/www"]
```

Build an image:

```
docker build -t localhost/p03_php .
```

Start the container:

```
docker run -d --name p03_php -p "8080:80" localhost/p03_php
```

Open in a web browser and reload the page multiple times. You can see the output is different each time with more lines.

Now delete the container. Probably you already know how, but as a reminder I show you again:

```
docker rm -f p03_php
```

Execute the “docker run ...” command again and reload the example web page to see how you have lost the previously generated lines and delete the container again.

Now start the container with a volume to preserve data:

```
docker run -d --mount source=p03_php_www,target=/var/www --name p03_php -p "8080:80" ↵  
↵localhost/p03_php
```

This way you can delete and create the container repeatedly as you will never lose your data until you delete the volume. You can see all volumes with the following command:

```
docker volume ls  
# or  
docker volume list
```

After you have deleted the container, you can delete the volume:

```
docker rm -f p03_php  
docker volume rm p03_php_www
```

CREATE A SIMPLE DOCKER COMPOSE PROJECT

Note: *Clone the git repository* if you haven't done it yet.

Go to Project 4 from the git repository root:

```
cd projects/p04
```

Project structure:

```
.
├── Dockerfile
├── docker-compose.yml
└── www
    └── index.php
```

The content of index.php

```
<?php
file_put_contents(__DIR__ . '/access.txt', date('Y.m.d. H:i:s') . "\n", FILE_APPEND);

echo 'P04<br/>';
echo nl2br(file_get_contents(__DIR__ . '/access.txt'));
```

the compose file

```
volumes:
  php_www:

services:
  php:
    image: localhost/p04_php
    build:
      context: .
      dockerfile: Dockerfile
    volumes:
      - php_www:/var/www
    ports:
      - "8080:80"
```

and the Dockerfile

```
FROM php:7.4-alpine
LABEL hu.itsziget.ld.project=p04
COPY www /var/www
CMD ["php", "-S", "0.0.0.0:80", "-t", "/var/www"]
```

Build an image and start the container using Docker Compose:

```
docker compose up -d
```

Check the container:

```
docker compose ps
# The name of the container: p04_php_1
```

Check the networks:

```
docker network ls
# New bridge network: p04_default
```

Delete the container, and networks with Docker Compose:

```
docker compose down
```

Or delete the volumes too.

```
docker compose down --volumes
```

COMMUNICATION OF PHP AND APACHE HTTPD WEB SERVER WITH THE HELP OF DOCKER COMPOSE

Note: *Clone the git repository* if you haven't done it yet.

Go to Project 5 from the git repository root:

```
cd projects/p05
```

Project structure:

```
.
├── Dockerfile
├── docker-compose.yml
└── www
    └── index.php
```

The content of index.php

```
<?php
file_put_contents(__DIR__ . '/access.txt', date('Y.m.d. H:i:s') . "\n", FILE_APPEND);
echo 'P05: ' . getenv('HOSTNAME') . '<br/>';
echo nl2br(file_get_contents(__DIR__ . '/access.txt'));
```

the compose file

```
volumes:
  www:

services:
  php:
    image: localhost/p05_php
    build:
      context: .
      dockerfile: Dockerfile
    volumes:
      - www:/var/www/html
  httpd:
    image: itsziget/httpd24:2.0
```

(continues on next page)

(continued from previous page)

```
volumes:
  - www:/var/www/html
environment:
  SRV_PHP: "true"
  SRV_DOCROOT: /var/www/html
ports:
  - "8080:80"
```

and the Dockerfile

```
FROM itsziget/php:7.4-fpm
LABEL hu.itsziget.ld.project=p05
COPY www /var/www/html
# The scripts interpreted by PHP-FPM executes on behalf of "www-data" user.
RUN chown www-data:www-data -R /var/www/html
```

Build PHP image and start the containers:

```
docker compose up -d
```

Start multiple container for PHP:

```
docker compose up -d --scale php=2
```

List the containers to see PHP has multiple instance:

```
docker compose ps
```

Open the page in your browser and you can see the hostname in the first line is not constant. It changes but not every time, although the data is permanent.

Delete everything created by Docker Compose for this project:

```
docker compose down --volumes
```

RUN MULTIPLE DOCKER COMPOSE PROJECTS ON THE SAME PORT USING NGINX-PROXY

Note: *Clone the git repository* if you haven't done it yet.

See [nginx-proxy](#)

Go to Project 6 from the git repository root:

```
cd projects/p06
```

Project structure:

```
.
├── nginxproxy
│   └── docker-compose.yml
├── web1
│   ├── Dockerfile
│   ├── docker-compose.yml
│   └── www
│       └── index.php
└── web2
    ├── Dockerfile
    ├── docker-compose.yml
    └── www
        └── index.php
```

We will need a proxy network which will be used by all of the compose projects for the communication between NGinX and the webservers:

```
docker network create public_proxy
```

Check the networks:

```
docker network ls
```

Navigate to the nginxproxy folder

```
cd nginxproxy
```

The compose file is the following:

```
name: p06proxy

networks:
  default:
    external: true
    name: public_proxy

services:
  nginx-proxy:
    image: nginxproxy/nginx-proxy:1.2.0
    ports:
      - "80:80"
    volumes:
      - /var/run/docker.sock:/tmp/docker.sock:ro
```

Start the proxy:

```
docker compose up -d
```

Navigate to the web1 folder:

```
cd ../web1
```

Here you will have a compose file:

```
name: p06web1

volumes:
  www:

networks:
  proxy:
    external: true
    name: public_proxy

services:
  php:
    image: localhost/p06_php_web1
    build:
      context: .
      dockerfile: Dockerfile
    volumes:
      - www:/var/www/html
  httpd:
    image: itsziget/httpd24:2.0
    volumes:
      - www:/var/www/html
    environment:
      SRV_PHP: "true"
      SRV_DOCROOT: /var/www/html
      VIRTUAL_HOST: web1.$NIP
    networks:
      - default
```

(continues on next page)

(continued from previous page)

```
- proxy
```

a Dockerfile

```
FROM itsziget/php:7.4-fpm
LABEL hu.itsziget.ld.project=p06
COPY www /var/www/html
# The scripts interpreted by PHP-FPM executes on behalf of "www-data" user.
RUN chown www-data:www-data -R /var/www/html
```

and the PHP file

```
<?php
file_put_contents(__DIR__ . '/access.txt', date('Y.m.d. H:i:s') . "\n", FILE_APPEND);
echo 'P06WEB1: ' . getenv('HOSTNAME') . '<br/>';
echo nl2br(file_get_contents(__DIR__ . '/access.txt'));
```

At this point you need to have the NIP variable set as *Welcome to Learn Docker's documentation!* refers to it. Alternative option: set the NIP variable in a “.env” file.

Start the containers:

```
docker-compose up -d
```

In case of working in the cloned repository of this tutorial, you can also run the below command to set the variable

```
NIP=$(../../../../../system/usr/local/bin/nip.sh) docker compose up -d
```

Navigate to the web2 folder:

```
cd ../web2
```

The compose file is similar to the previous one:

```
name: p06web2
volumes:
  www:
networks:
  proxy:
    external: true
    name: public_proxy
services:
  php:
    image: localhost/p06_php_web2
    build:
```

(continues on next page)

(continued from previous page)

```
context: .
dockerfile: Dockerfile
volumes:
- www:/var/www/html
httpd:
image: itsziget/httpd24:2.0
volumes:
- www:/var/www/html
environment:
SRV_PHP: "true"
SRV_DOCROOT: /var/www/html
VIRTUAL_HOST: web2.$NIP
networks:
- default
- proxy
```

we also have another Dockerfile

```
FROM itsziget/php:7.4-fpm
LABEL hu.itsziget.ld.project=p06
COPY www /var/www/html
# The scripts interpreted by PHP-FPM executes on behalf of "www-data" user.
RUN chown www-data:www-data -R /var/www/html
```

and a PHP file

```
<?php
file_put_contents(__DIR__ . '/access.txt', date('Y.m.d. H:i:s') . "\n", FILE_APPEND);
echo 'P06WEB2: ' . getenv('HOSTNAME') . '<br/>';
echo nl2br(file_get_contents(__DIR__ . '/access.txt'));
```

Start the containers:

```
docker compose up -d
```

Or you can use `nip.sh` as we did in `web1`.

Both of the services are available on port 80. Example:

```
http://web1.192.168.1.6.nip.io
http://web2.192.168.1.6.nip.io
```

This way you do not need to remove a container just because it is running on the same port you want to use for a new container.

Clean the project:

```
docker compose down --volumes
cd ../web1
```

(continues on next page)

(continued from previous page)

```
docker compose down --volumes  
cd ../nginxproxy  
docker compose down --volumes
```


PROTECT YOUR WEB SERVER WITH HTTP AUTHENTICATION

Note: *Clone the git repository* if you haven't done it yet.

Go to Project 7 from the git repository root:

```
cd projects/p07
```

Project structure:

```
.
├── nginxproxy
│   └── docker-compose.yml
├── web
│   ├── .env
│   ├── docker-compose.yml
│   └── www
│       └── index.html
```

The first step is the same as it was in *Run multiple Docker Compose projects on the same port using nginx-proxy*. Let's go to nginxproxy

```
cd nginxproxy
```

The compose file is:

```
name: p07proxy

networks:
  default:
    external: true
    name: public_proxy

services:
  nginx-proxy:
    image: nginxproxy/nginx-proxy:1.2.0
    ports:
      - "80:80"
    volumes:
      - /var/run/docker.sock:/tmp/docker.sock:ro
```

Start the proxy server:

```
docker compose up -d
```

Go to the web folder:

```
cd ../web
```

The compose file is

```
name: p07web

volumes:
  apache2:

networks:
  proxy:
    external: true
    name: public_proxy

services:
  htpasswd:
    image: itsziget/httpd24:2.0
    volumes:
      - apache2:/usr/local/apache2
    command:
      - "/bin/bash"
      - "-c"
      - "htpasswd -nb $HTTPD_USER $HTTPD_PASS >> /usr/local/apache2/.htpasswd"
    network_mode: none
  httpd:
    depends_on:
      - htpasswd
    image: itsziget/httpd24:2.0
    volumes:
      - apache2:/usr/local/apache2
      - ./www:/usr/local/apache2/htdocs
    networks:
      - proxy
    environment:
      SRV_AUTH: "true"
      VIRTUAL_HOST: p07.$NIP
  fixperm:
    depends_on:
      - httpd
    image: bash
    volumes:
      - ./www:/htdocs
    network_mode: none
    command:
      - "bash"
      - "-c"
      - "find htdocs/ -type f -exec chmod -R 0655 {} \\\; && chmod 0775 /htdocs && chown -
↵R 33:33 /htdocs"
```

In this case we have a simple html file

```
<p style="text-align: center; font-size: 20pt">Hello Docker User!</p>
```

You can simply start a web server protected by HTTP authentication. The name and the password will come from environment variables. I recommend you to use a more secure way in production. Create the `.htpasswd` file manually and mount it inside the container.

The `htpasswd` container will create `.htpasswd` automatically and exit.

In the `“.env”` file you can find two variables.

```
HTTPD_USER=user
HTTPD_PASS=secretpass
```

The variables will be used in `“docker-compose.yml”` by the `“htpasswd”` service to generate the password file and then the `“httpd”` service will read it from the common volume.

The `“fixperm”` service runs and exits similarly to `“htpasswd”`. It sets the permission of the files after the web server starts.

Use the `“depends_on”` option to control which service starts first.

At this point you need to have the `NIP` variable set as the *Welcome to Learn Docker’s documentation!* refers to it.

Alternative option: set the `NIP` variable in the `“.env”` file.

Start the web server

```
docker compose up -d
```

In case are working the in cloned repository of this tutorial, you can also run the below command to set the variable

```
NIP=$(../../system/usr/local/bin/nip.sh) docker compose up -d
```

Open the web page in your browser (Ex.: `p07.192.168.1.6.nip.io`). You will get a password prompt.

Clean the project:

```
docker compose down --volumes
cd ../nginxproxy
docker compose down --volumes
```


MEMORY LIMIT TEST IN A BASH CONTAINER

Note: *Clone the git repository* if you haven't done it yet.

Go to Project 8 from the git repository root:

```
cd projects/p08
```

Project structure:

```
├── docker-compose.yml
```

11.1 Files

docker-compose.yml

```
services:
  test:
    image: bash:5.2
    environment:
      - ALLOCATE
    command:
      - -c
      - 'fallocate -l $ALLOCATE /app/test && echo $(( $(stat /app/test -c "%s") / 1024 /↵
↵1024))'
    deploy:
      resources:
        limits:
          memory: 50MB
    mem_swappiness: 0
    tmpfs:
      - /app
```

11.2 Description

This example shows the memory testing in a bash container, where the “fallocate” command generates a file with a defined size stored in memory using tmpfs. We use an environment variable to set the allocated memory and the memory limit is defined in the compose file as 50 MiB.

11.3 Start the test

The container will have 50MB memory limit. (It must be at least 6MB in Docker Compose 1.27.4). The examples below will test the memory usage from 10MB to 50MB increased by 10MB for each test.

```
ALLOCATE=10MiB docker compose run --rm test
ALLOCATE=20MiB docker compose run --rm test
ALLOCATE=30MiB docker compose run --rm test
ALLOCATE=40MiB docker compose run --rm test
ALLOCATE=50MiB docker compose run --rm test
```

Running it in Docker Desktop on macOS I get the following output:

```
fallocate: fallocate '/app/test': Out of memory
Out of memory
```

Running it on a virtual machine with an Ubuntu 20.04 host the output is:

```
Out of memory
```

Since there is some additional memory usage in the container, it kills the process at 50MiB even though 50 is still allowed.

11.4 Explanation of the parameters

The “docker compose run” is similar to “docker run”, but it runs a service from the compose file. “-rm” means the same as it meant for “docker run”. Deletes the container right after it stopped.

Clean the project:

```
docker compose down
```

The containers were deleted automatically, but it can still delete the network.

CPU LIMIT TEST

Note: *Clone the git repository* if you haven't done it yet.

Go to Project 9 from the git repository root:

```
cd projects/p09
```

Project structure:

```
├── Dockerfile
```

12.1 Files

Dockerfile

```
# Based on "Petar Maric outdated image"
# https://github.com/petarmaric/docker.cpu_stress_test

FROM ubuntu:20.04

# Update the Ubuntu package index and install the required Ubuntu packages
RUN apt-get update \
  && apt-get install -y --no-install-recommends stress

# Parameterize this Dockerfile, by storing the app configuration within environment
  ↪ variables
ENV STRESS_TIMEOUT 120
ENV STRESS_MAX_CPU_CORES 1

CMD stress --cpu $STRESS_MAX_CPU_CORES --timeout $STRESS_TIMEOUT
```

We test the CPU limit in this example using an image based on [petarmaric/docker.cpu-stress-test](https://github.com/petarmaric/docker.cpu-stress-test). Since that image is outdated, we create a new image similar to Peter Maric's work.

```
docker build -t localhost/stress .
```

Execute the following command to test a whole CPU core:

```
docker run -it --rm \
-e STRESS_MAX_CPU_CORES=1 \
-e STRESS_TIMEOUT=30 \
--cpus=1 \
localhost/stress
```

Run “top” in an other terminal to see that the “stress” process uses 100% of one CPU. To see the same result on any host operating system, we will run top in an Ubuntu container using the process namespace of the host.

```
docker run --rm -it --pid host ubuntu:20.04 top
```

Press Ctrl-C and execute the following command to test two CPU core and allow the container to use only 1 and a half CPU.

```
docker run -it --rm \
-e STRESS_MAX_CPU_CORES=2 \
-e STRESS_TIMEOUT=30 \
--cpus=1.5 \
localhost/stress
```

Use “top” again to see that the “stress” process uses 75% of two CPU.

You can test on one CPU core again and allow the container to use 50% of a specific CPU core by setting the core index.

```
docker run -it --rm \
-e STRESS_MAX_CPU_CORES=1 \
-e STRESS_TIMEOUT=60 \
--cpus=0.5 \
--cpuset-cpus=0 \
localhost/stress
```

You can use top again, but do not forget to add the index column to the list:

- run `docker run --rm -it --pid host ubuntu:20.04 top`
- press “f”
- Select column “P” by navigating with the arrow keys
- Press “SPACE” to select “P”
- Press “ESC”

Now you can see the indexes in the column “P”.

Press “1” to list all the CPU-s at the top of the terminal so you can see the usage of all the CPU-s.

LEARN WHAT EXPOSE IS AND WHEN IT MATTERS

Note: *Clone the git repository* if you haven't done it yet.

13.1 Intro

Go to Project 10 from the git repository root:

```
cd projects/p10
```

Project structure:

```
.
├── expose
│   ├── docker-compose.yml
│   ├── Dockerfile
│   └── index.html
└── noexpose
    ├── docker-compose.yml
    ├── Dockerfile
    └── index.html
```

It is a misconception that exposing a port is required to make a service available on that specific port from the host or from another container.

Firewalls can make additional restrictions in which case the following examples can work differently. If some special firewall looks for the exposed ports today or in the future, it doesn't change the fact that exposed ports are just metadata for an interactive user or another software to make decisions.

This tutorial does not state that you should not use the **EXPOSE** instruction in the Dockerfile or the `--expose` option of `docker run` or the `expose` parameter in a Docker Compose file. It only states that it is not required, so use it when you know it is necessary either because some automation depends on it or just because you like to document which port is used in a container.

13.2 Accessing services from the host using the container's IP address

We will use `noexpose/Dockerfile` to build an example Python HTTP server:

`noexpose/Dockerfile`:

```
FROM python:3.8-alpine

WORKDIR /var/www
COPY index.html /var/www/index.html

CMD ["python3", "-m", "http.server", "8080"]
```

Let's build the image

```
docker build ./noexpose -t localhost/expose:noexpose
```

Run the container

```
docker run --name p10_noexpose -d --init localhost/expose:noexpose
```

List the exposed ports (you should not find any)

```
docker container inspect p10_noexpose --format '{{ json .Config.ExposedPorts }}'
# output: null
```

Get the IP address of the container:

```
IP=$(docker container inspect p10_noexpose --format '{{ .NetworkSettings.IPAddress }}')
```

Get the index page from the server:

```
curl $IP:8080
# or
wget -qO- $IP:8080
```

The output should be

```
no expose
```

It means exposing a port is not necessary for making a service available on that port inside the container. It is just a metadata for you and for some proxy services to automate port forwarding based on exposed ports. Docker itself will not forward any port based on this information automatically.

Let's remove the container

```
docker container rm -f p10_noexpose
```

13.3 Using user-defined networks to access services in containers

You could think the previous example worked because we used the default Docker bridge which is a little different than user-defined networks. The following example shows you that it doesn't matter. Docker Compose creates a user-defined network for each project, so let's use Docker Compose to run the containers. One for the server and one for the client.

The `noexpose/docker-compose.yml` is the following:

```
name: p10noexpose

networks:
  default:
  client:

x-client-base: &client-base
depends_on:
  - server
image: nicolaka/netshoot:v0.8
command:
  - sleep
  - inf
init: true

services:

  server:
    build: .
    init: true
    networks:
      - default

  client1:
    <<: *client-base
    networks:
      - default

  client2:
    <<: *client-base
    networks:
      - client
```

Note: I used some special YAML syntax to make the compose file shorter. This way I could define the common parameters of the client containers.

Run the containers:

```
docker compose -f noexpose/docker-compose.yml up -d
```

Get the IP address of the server container:

```
ID=$(docker compose -f noexpose/docker-compose.yml ps -q server)
IP=$(docker container inspect "$ID" --format '{{ .NetworkSettings.Networks.p10noexpose_
```

(continues on next page)

(continued from previous page)

```
↪default.IPAddress }}')
```

Get the index page from the server:

```
curl $IP:8080
# or
wget -qO- $IP:8080
```

The output should be the same as before:

```
no expose
```

Now let's get the main page using curl from another container:

```
docker-compose -f noexpose/docker-compose.yml exec client1 curl $IP:8080
```

Again, we get the output as we expected:

```
no expose
```

What if we try from another Docker network? "client2" has its own network and doesn't use default as the other two containers. Let's try from that.

```
docker compose -f noexpose/docker-compose.yml exec client2 curl --max-time 5 $IP:8080
```

I set `--max-time` to 5 so after about 5 seconds, it times out. Without `--max-time`, it would try much longer.

```
curl: (28) Connection timed out after 5001 milliseconds
```

I guess you think we finally found the case when we need to expose the port to make it available in another docker network. Wrong.

Before we continue, let's remove the containers

```
docker compose -f noexpose/docker-compose.yml down
```

We will use the other compose project in which `EXPOSE 8080` is defined in the Dockerfile, which is the following. `expose/Dockerfile`:

```
FROM python:3.8-alpine
WORKDIR /var/www
COPY index.html /var/www/index.html
CMD ["python3", "-m", "http.server", "8080"]
EXPOSE 8080
```

It will `COPY` an `index.html` containing one line:

```
expose PORT 8080
```

and we use the `docker-compose.yml` below:

```

name: p10expose

networks:
  default:
  client:

x-client-base: &client-base
  depends_on:
    - server
  image: nicolaka/netshoot:v0.8
  command:
    - sleep
    - inf
  init: true

services:

  server:
    build: .
    init: true
    networks:
      - default

  client1:
    <<: *client-base
    networks:
      - default

  client2:
    <<: *client-base
    networks:
      - client

```

Run the project

```
docker compose -f expose/docker-compose.yml up -d
```

Get the IP address of the server

```

ID=$(docker compose -f expose/docker-compose.yml ps -q server)
IP=$(docker container inspect "$ID" --format '{{ .NetworkSettings.Networks.p10expose_
↪default.IPAddress }}')

```

Before you run the tests, list the exposed ports (Now you should not find 8080/tcp)

```
docker container inspect "$ID" --format '{{ json .Config.ExposedPorts }}'
```

```
{"8080/tcp":{}}
```

Test the port from the host:

```
curl $IP:8080
# or
```

(continues on next page)

(continued from previous page)

```
wget -qO- $IP:8080
```

And finally test it from client1 before client2, which is in a different network:

```
docker compose -f expose/docker-compose.yml exec client1 curl $IP:8080
docker compose -f expose/docker-compose.yml exec client2 curl --max-time 5 $IP:8080
```

You probably figured it out why I used `--max-time` again.

It doesn't matter whether you expose the port or not. It will not help you to reach the server from a different Docker network. You need to attach a common network to the containers in order to communicate or forward a port from the host and use that host port as target.

It's time to delete the containers:

```
docker compose -f expose/docker-compose.yml down
```

13.4 What is the connection between port forwards and exposed ports?

Let's run a simple container to demonstrate it.

```
docker run -d --name p10_port_forward --init -p 8081:8080 python:3.8-alpine python3 -m
↳ http.server 8080
```

Check the exposed ports:

```
docker container inspect p10_port_forward --format '{{ json .Config.ExposedPorts }}'
```

The output is familiar. We have seen it before:

```
{"8080/tcp":{}}
```

It means even if we don't expose the port directly, but forward a port from the host, the target port will be exposed. It is not enough though to use that port from another machine. There is another setting for the container, and that is "PortBindings". Let's inspect that:

```
docker container inspect p10_port_forward --format '{{ json .HostConfig.PortBindings }}'
```

```
{"8080/tcp":[{"HostIp":"","HostPort":"8081"}]}
```

As you can see PortBindings in the HostConfig section. It is because it doesn't affect the container itself. Instead, it configures the host to forward a specified port to the container's IP address. You could do it without Docker. The problem is that you don't know what the IP address will be, so Docker solves it for you automatically.

We can finally remove the last container since we know confidently how exposing a port does not affect whether we can access that port or not.

```
docker container rm -f p10_port_forward
```

However, when we use Docker's built-in port-forward, Docker also exposes the target port, which is just how Docker works internally at the moment. It is not necessary in order to be able to access the port.

DOCKER NETWORK AND NETWORK NAMESPACES IN PRACTICE

This tutorial can be watched on YouTube: <https://youtu.be/HtJEmjW3qmg>

14.1 Linux Kernel Namespaces in general

Let's start with a simple example so we can understand why namespaces are useful.

PHP running "on the host"



Let's say, you have a running PHP process on your Linux machine. That PHP process will be able to see your entire host, including files, other running processes and all the network interfaces. It will not just see the interfaces, it will be able to listen on all the IP addresses, so you have to configure PHP to listen on a specific IP address. PHP will also have internet access which is usually what you want, but not always.

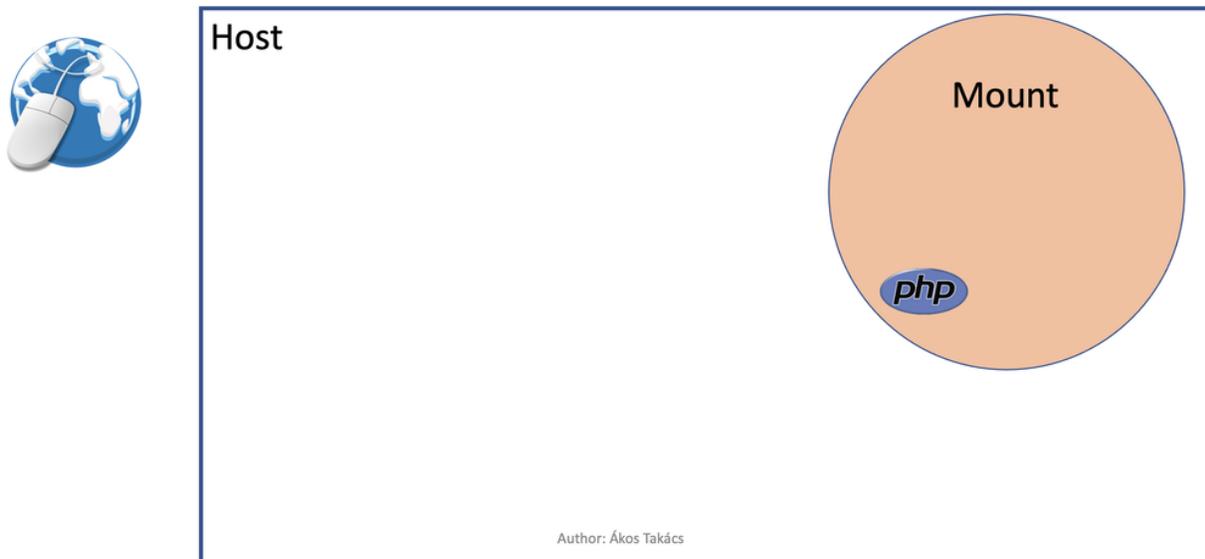
This is when Linux kernel namespaces can help us. A kernel namespace is like a magic wall between a running process and the rest of the host. This magic wall will only hide some parts of the host from a specific point of view. In this tutorial we will mainly cover the three best known point of views.

- Filesystem (Mount namespace)
- Network (Network namespace)

- Process (PID namespace, which means “Process ID” namespace)

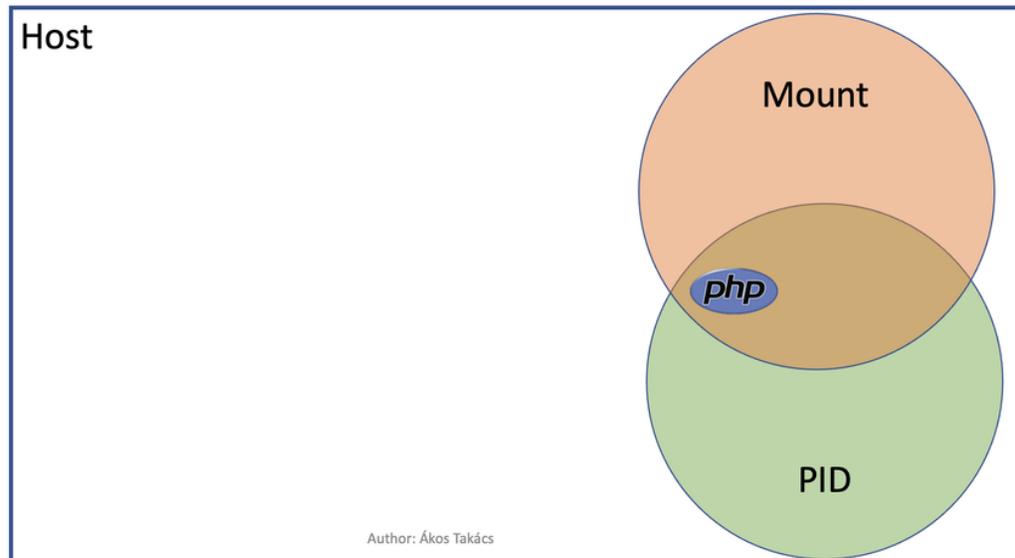
The first namespace is mount namespace. You could also remember it as “Jail” or “chroot”. It means the process will see only a folder on the host and not the root filesystem. That folder could be empty, but it is very often a folder that contains very similar files as the root filesystem does. This way PHP will “think” it is running on a different host and it won’t even know if there is anything outside that folder.

Mount namespace / Jail / chroot



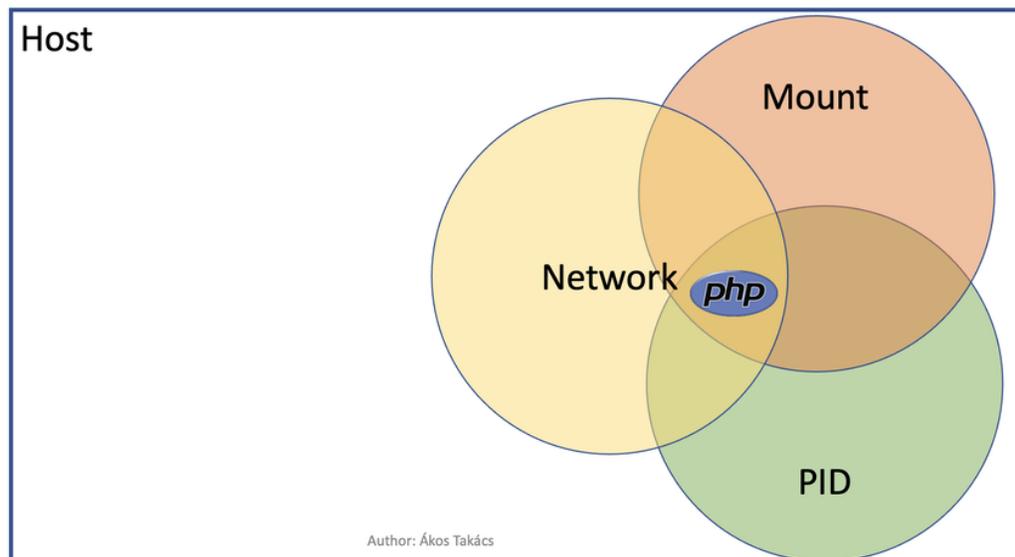
This is not always enough. Sometimes you don’t want a specific process to see other processes on the host. In this case you can use the PID namespace so the PHP process will not be able to see other processes on the host, only the processes in the same PID namespace. Since on Linux there is always a process with the process ID 1 (PID 1), for any process in the network namespace, PHP will have PID 1. For any process outside of that PID namespace, PHP will have a larger process ID like 2324.

PHP can't see processes and files "on the host"



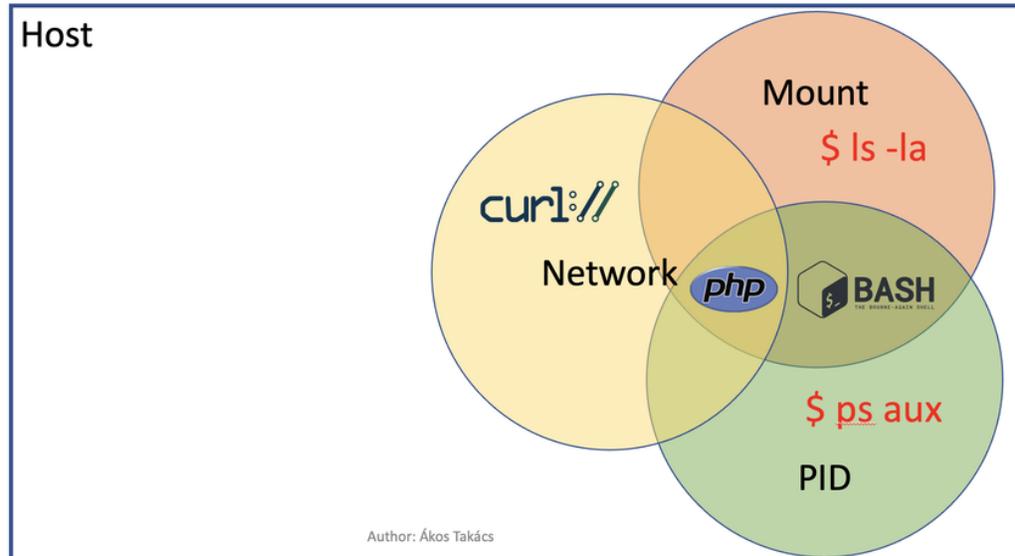
And finally we arrived to the network namespace. Network namespace can hide network interfaces from processes in the namespace. The network namespace can have its own IP address, but it will not necessarily have one. Thanks to the network namespace, PHP will only be able to listen its own IP address in a Docker network.

PHP can't see processes, files and network interfaces "on the host"



These namespaces are independent and not owned by the PHP process. Any other process could be added to these namespaces, and you don't need to run those processes in each namespace.

Run any command in any namespace of any process

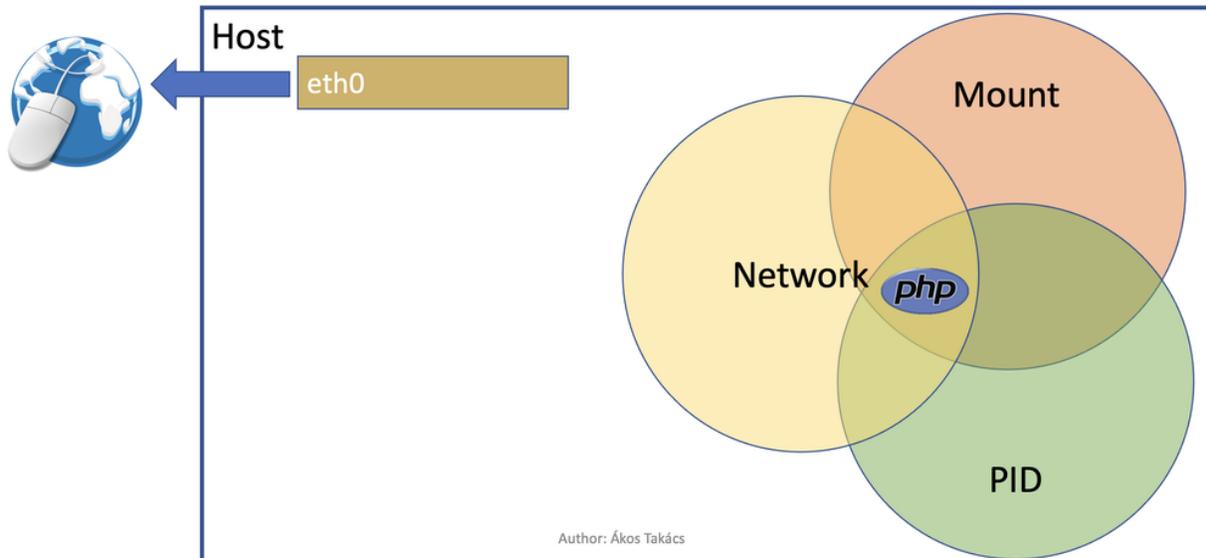


You can choose the network namespace even without the mount namespace, so you can use an application (web browser, curl) on the host and run it in the network namespace of the PHP process, so even if the PHP is not available outside of the container, running the browser in the network namespace of the container will allow you to access your website.

14.2 Network traffic between a container and the outside world

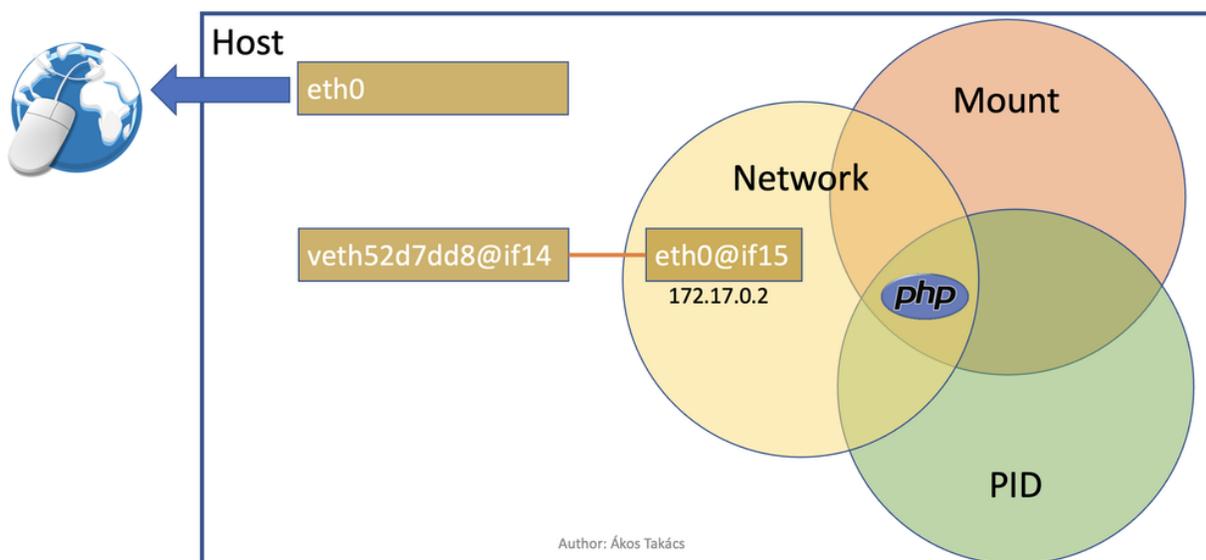
Your host machine usually has a network interface which is connected to the internet or at least to a local network. We will call it “the outside world”.

Network namespace with network interfaces



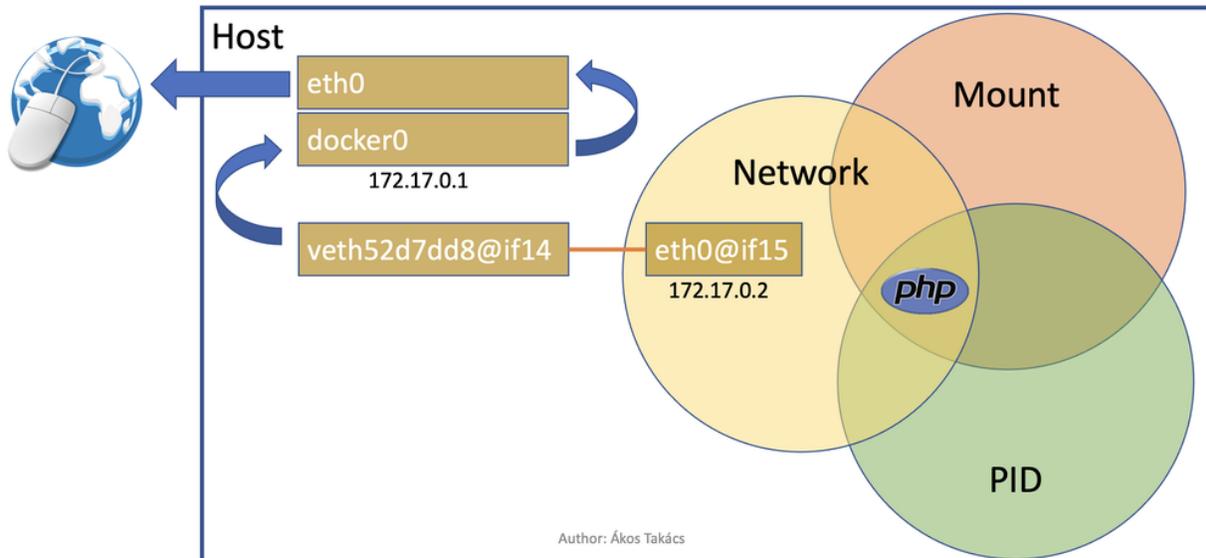
This interface could be “eth0”, although recently it is more likely to be something like “enp1s0” or “eno1”. The point is that traffic routed through this interface can leave the host machine. The container needs its own network interface which is connected to another outside of the container on the host. The name of the interface on the host will start with “veth” followed by a hash.

Virtual ethernet cable



The veth interface will not have IP address. It could have, but Docker uses a different way so containers in the same Docker network can communicate with each other. There will be a bridge network between the veth interface and `eth0`. The bridge of the default Docker network is “docker0”.

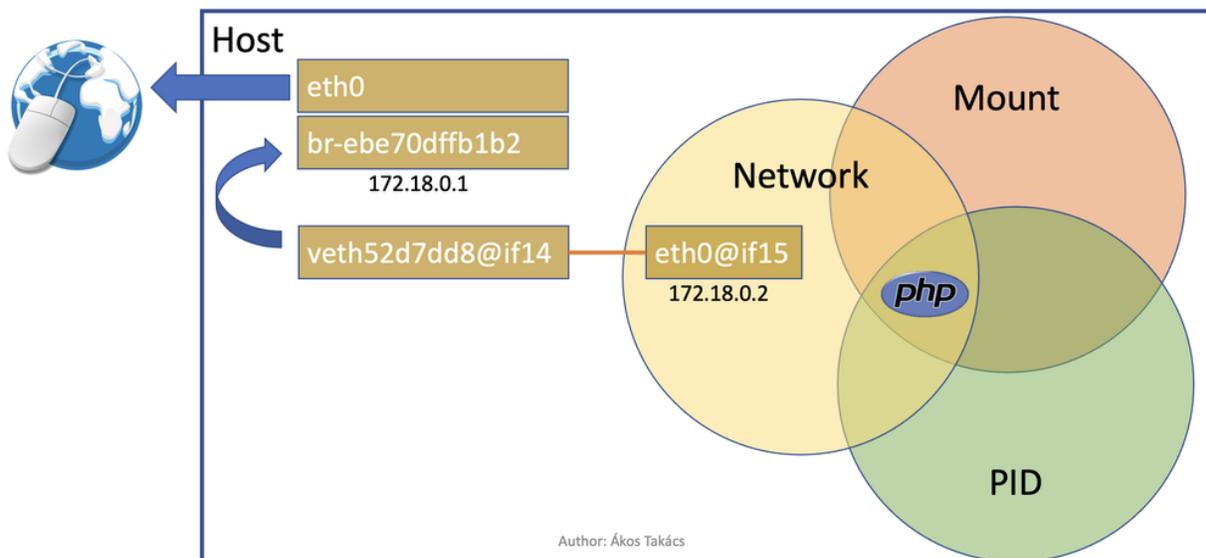
docker0 bridge



This bridge will have an IP address which will also be the gateway for each container in the default Docker network. Since this is on the host outside of the network namespace, any process running on the host could listen on this IP address so processes running inside the container could use this IP to access a webservice listening on it.

Sometimes you don't want a containerized process to access the internet, because you don't trust an application and you want to test or run it without internet access for security reasons indefinitely. This is when "internal networks" can help.

Internal Docker network



Containers don't accept port forwards on IP addresses in internal networks so it is not just rejecting outgoing traffic to the outside world, but also rejecting incoming requests from other networks.

You can create a user-defined Docker network which will have a new bridge. If you also define that network as “internal” using the following command for example

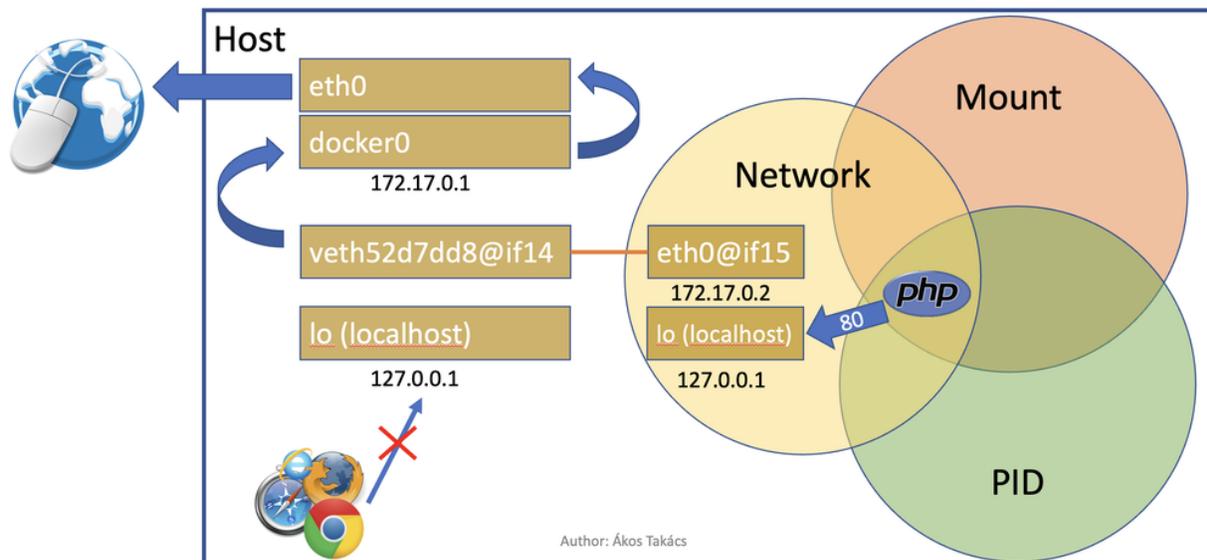
```
docker network create secure_net --internal
```

the network traffic will not be forwarded from the bridge to eth0 so PHP will only be able to access services running on the host or in the container.

There is another very important interface called “lo” better known as “localhost” which usually has an IP address like 127.0.0.1. This is however not the only IP address that is bound to this interface. Every part of the IP could be changed after 127. and still pointing to the same interface. It is important to know that every network namespace has its own localhost.

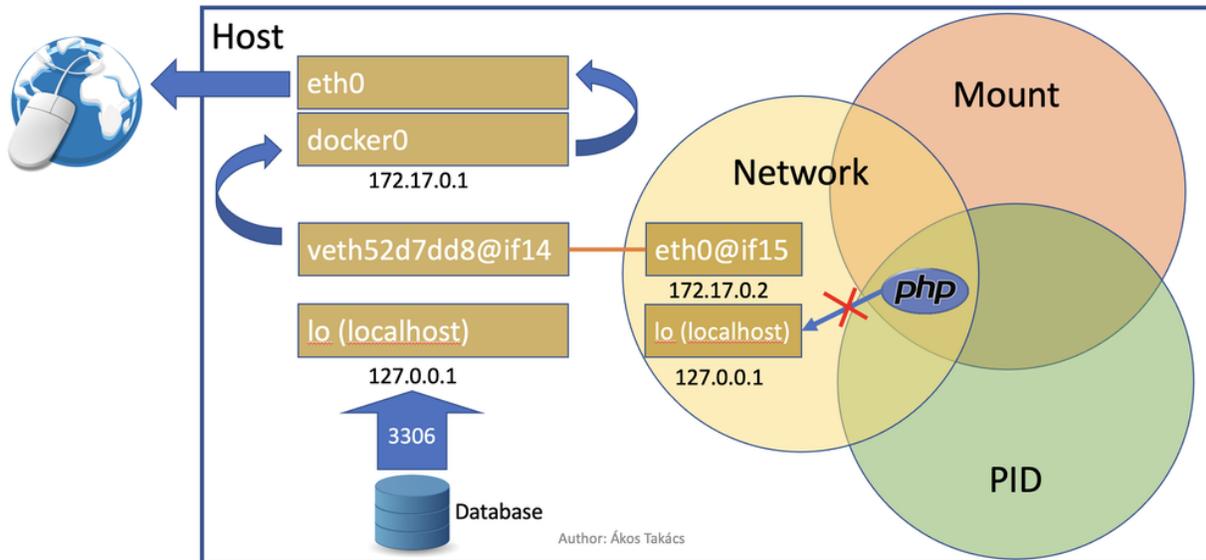
Let’s see what it means.

Each net namespace has its own "localhost" (2)



If the web application is listening on port 80 on localhost, a web browser outside of the container will not be able to access it, since it has a different localhost. The same is true when for example a database server is running on the host listening on port 3306 on localhost. The PHP process inside the container will not be able to reach it.

Each net namespace has its own "localhost" (3)

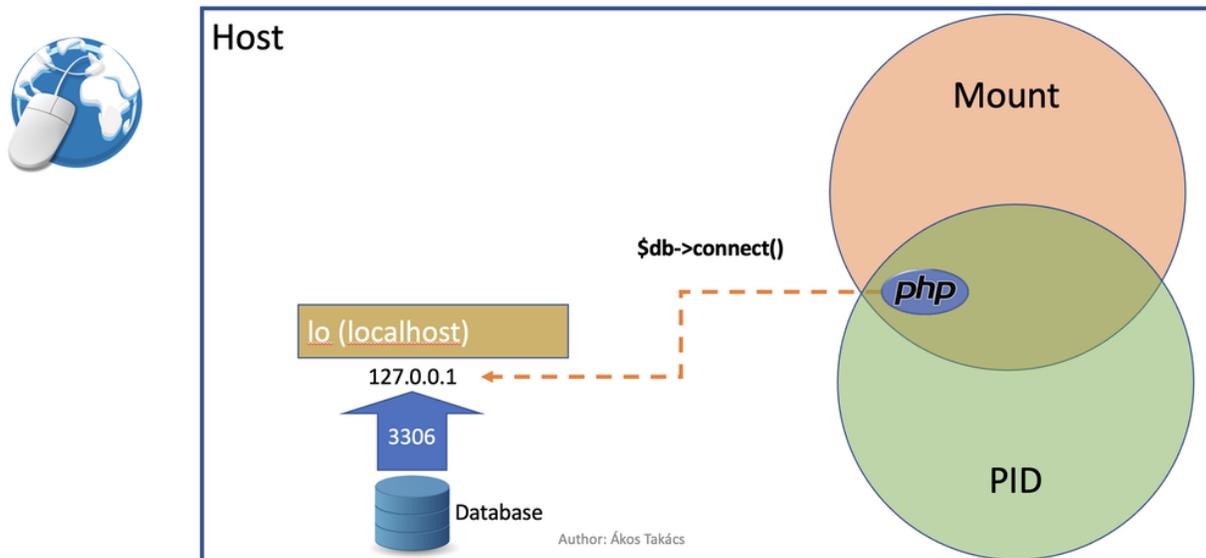


Since the reason is the network namespace, you could just run the container in host network mode

```
docker run -d --name php-hostnet --network host itsziget/phar-examples:1.0
```

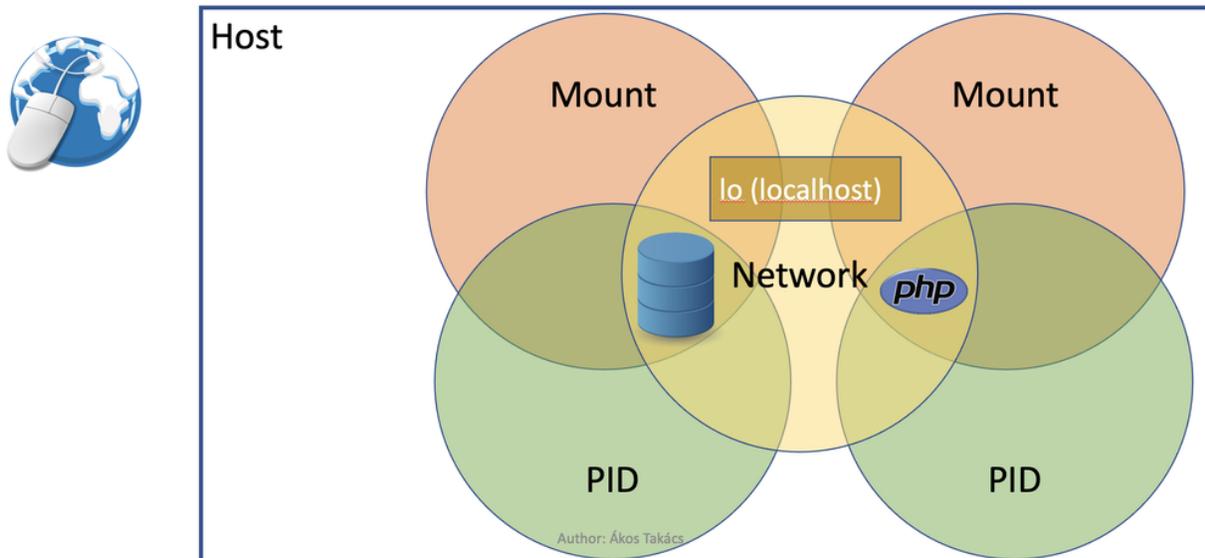
which means you just don't get the network isolation. The host network mode does not mean that you are using a special Docker network. It only means you don't want the container to have its own network namespace.

Using host network equals no network namespace



Of course we wanted to have the network isolation and we want to keep it. The other solution is running another container which will use the same network namespace.

Common network namespace for containers



14.3 Manipulating network namespaces

Docker is not the only tool to manipulate namespaces. I will show you the following tools.

- Container engines (Docker)
- “ip” command
- “nsenter” command
- “unshare” command

14.3.1 Two containers using the same network namespace

Of course the first we have to talk about is still Docker. The following commands will start a PHP demo application and run a bash container using the same network namespace as the PHP container so we can see the network interfaces inside the PHP container.

```
docker run -d --name php itsziget/phar-examples:1.0
docker run --rm -it --network container:php bash:5.1 ip addr
```

There is a much easier solution of course. We can just use `docker exec` to execute a command in all of the namespaces of the PHP container.

```
docker exec php ip addr
```

This command works only because “ip”, which is part of the “iproute2” package is installed inside the PHP container, so it wouldn’t work with every base image and especially not with every command.

14.3.2 “nsenter”: run commands in any namespace

The “nsenter” (namespace enter) command will let you execute commands in specific namespaces. The following command would execute `ip addr` in the network namespace of a process which has the process ID `$pid`.

```
sudo nsenter -n -t $pid ip addr
```

We have to get the id of a process running inside a container. Remember, the process has a different ID inside and outside of the container because of the PID namespace, so we can’t just run the `ps aux` command inside the container. We need to “inspect” the PHP container’s metadata.

```
pid="$(docker container inspect php --format '{{ .State.Pid }}')"
```

The above command will save the process ID in the environment variable called “pid”. Now let’s run nsenter again.

```
sudo nsenter -n -t $pid ip addr
sudo nsenter -n -t $pid hostname
sudo nsenter -n -u -t $pid hostname
```

The first command will show us the network interfaces inside the network namespace of the PHP container. The second command will try to get the hostname of the container, but it will return the hostname of the host machine. Although the hostname is related to the network in our mind, it is not part of the network namespace. It is actually the part of the UTS namespace. Since the long name of the namespace would just confuse you, I will not share it at this point of the tutorial. The good news is that we can also use the UTS namespace of the container by adding the `-u` flag to the “nsenter” command, and this is what the third line does.

14.3.3 “ip netns” to create new network namespaces

“nsenter” was great for running commands in existing namespaces. If you want to create network namespaces, you can use the `ip netns` command, but before we create one, let’s list existing network namespaces:

```
ip netns list
```

The above command will give you nothing even if you have running containers using network namespaces. To understand why, first let’s look at content of two folders

```
ls /run/netns
sudo ls /run/docker/netns
```

The first line, used by the “ip” command will not give you anything, but the second will give you at least one file, which is the file of the network namespace of our previously started PHP container.

As you can see, if you want to work with namespaces, you need to refer to a file or the name of the file. Docker and the “ip” command uses a different folder to store those files. These files are not the only way to refer to network namespaces and we will discuss it later.

It’s time to create our first network namespace without Docker.

```
sudo ip netns add test
ls /run/netns
```

The “ls” command isn’t required here, but can show us that we indeed created a file. Let’s run `ip addr` inside our new network namespace:

```
sudo ip netns exec test ip addr
```

Note: You could actually use `nsenter` to run `ip addr` in a network namespace even if you don't have an existing process.

```
sudo nsenter --net=/run/netns/test ip addr
```

The output will be

```
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

As you can see this new network namespace doesn't even have a loopback IP address so basically it doesn't have "localhost". It shows us that a network namespace does not give us a fully configured private network, it only gives us the network isolation. Now that we know it, it is not surprising that the following commands will give us error messages.

```
sudo ip netns exec test ping dns.google
# ping: dns.google: Temporary failure in name resolution
sudo ip netns exec test ping 8.8.8.8
# ping: connect: Network is unreachable
```

Since this network namespace is useless without further configuration and configuring the network is not part of this tutorial, we can delete it:

```
sudo ip netns del test
```

14.3.4 "unshare": Temporary network namespace creation

If you want to create a temporary network namespace and run a command inside it, you can use `unshare`. This command has similar parameters as `nsenter` but it doesn't require existing namespaces. It will create new namespaces for the commands that you want to run. IT could be useful when you just want to test an application that you shouldn't use the network so you can run it in a safer environment.

```
sudo unshare -n ip addr
```

It will give you the same output as our previous attempt to create a network namespace.

```
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

14.3.5 Working with Docker's network namespaces

Allow the “ip” command to use Docker's network namespaces

If you want, you could remove `/run/netns` and create a symbolic link instead pointing to `/run/docker/netns`.

```
sudo rm -r /run/netns
sudo ln -s /run/docker/netns /run/netns
ip netns list
```

Sometimes you can get an error message saying that

```
Error: rm: cannot remove '/run/netns': Device or resource busy
```

Since we started to use the “ls” and “ip” commands to list namespaces, it is likely that we get this error message even though we are not actively using that folder. There could be two solutions to be able to remove this folder:

- Exiting from current shell and opening a new one
- Rebooting the machine

The first will not always work, and the second is obviously something that you can't do with a running production server.

A better way of handling the situation is creating symbolic links under `/run/netns` pointing to files under `/run/docker/netns`. In Docker's terminology the file is called “sandbox key”. We can get the path of a container's sandbox key by using the following command:

```
sandboxKey=$(docker container inspect php --format '{{.NetworkSettings.SandboxKey}}')
```

The end of that path is the filename which we will need to create a link under `/run/netns`.

```
netns=$(basename "$sandboxKey")
```

Using the above variables we can finally create our first symbolic link

```
sudo ln -s $sandboxKey /run/netns/$netns
```

Finally, `ip netns ls` will give us an output similar to the following:

```
a339e5fc43f0 (id: 0)
```

Name resolution issue with “ip netns exec”

It's time to run `ip netns exec` to test the network of a Docker container.

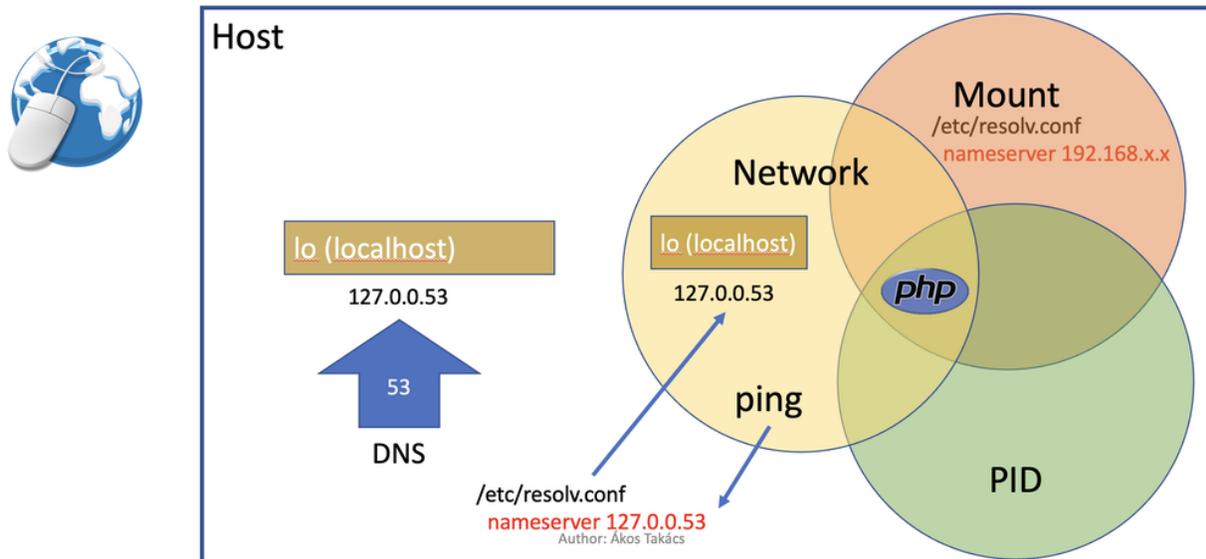
```
sudo ip netns exec $netns ip addr
sudo ip netns exec $netns ping 8.8.8.8
sudo ip netns exec $netns ping dns.google
```

The first two lines will give the expected results, but the third line will give us the following error message.

```
Error: ping: dns.google: Temporary failure in name resolution
```

What happened?

DNS resolution without mount namespace



We ran the ping command only in the network namespace of the container, which means the configuration files that are supposed to control how name resolution works are loaded from the host. My host was an Ubuntu 20.04 LTS virtual machine created by [Multipass](#). By default, the IP address of the nameserver is `127.0.0.53`. Remember, that this IP address belongs to the loopback interface which is different in each network namespace. In the network namespace of our PHP container there is no service listening on this IP address.

Solution 1: Change the configuration on the host

Danger: DO NOT test it in a production environment as it could also break your name resolution if you are doing something wrong.

`/etc/resolv.conf` is usually a symbolic link pointing one of the following files:

- `/run/systemd/resolve/stub-resolv.conf`
- `/run/systemd/resolve/resolv.conf`

Depending on your system it could point to an entirely different file or it could also be a regular file instead of a symbolic link. I will only discuss the above files in this tutorial.

Run the following command to get the real path of the configuration file.

```
readlink -f /etc/resolv.conf
```

Learn Docker

Note: Alternatively, you could also run `realpath /etc/resolv.conf`

If the output is `/run/systemd/resolve/stub-resolv.conf`, you are using the stub resolver and the content of the file looks like this without the comments:

```
nameserver 127.0.0.53
options edns0 trust-ad
search .
```

On the other hand, `/run/systemd/resolve/resolv.conf` will directly contain the nameservers:

```
nameserver 192.168.205.1
search .
```

Now I will change the symbolic link:

```
sudo unlink /etc/resolv.conf
sudo ln -s /run/systemd/resolve/resolv.conf /etc/resolv.conf
```

After this I will be able to successfully ping the domain name of Google's name server:

```
sudo ip netns exec $netns ping dns.google
```

I don't want to keep this configuration, so I will restore the stub resolver:

```
sudo unlink /etc/resolv.conf
sudo ln -s /run/systemd/resolve/stub-resolv.conf /etc/resolv.conf
```

Solution 2: Using per-namespace resolv.conf

We can create additional configuration files for each network namespace. First we have to create a new folder using the name of the namespace under `/etc/netns`

```
sudo mkdir -p /etc/netns/$netns
```

After that we have to create a `resolv.conf` file in the new folder and add a nameserver definition like `nameserver 8.8.8.8`

```
echo "nameserver 8.8.8.8" | sudo tee /etc/netns/$netns/resolv.conf
```

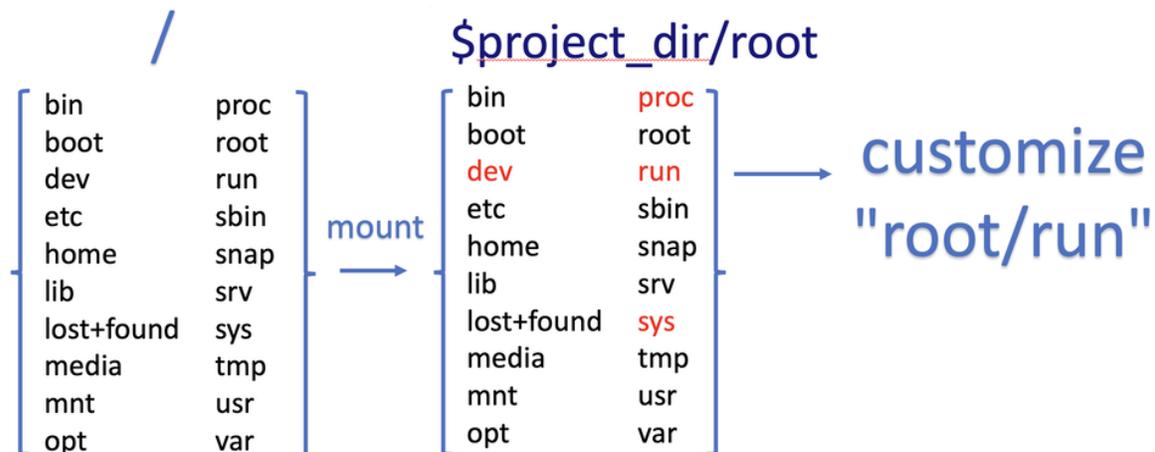
And finally we can ping the domain name

```
sudo ip netns exec $netns ping dns.google
```

Solution 3: Using a custom mount namespace based on the original root filesystem

This is a very tricky solution which I would not recommend usually, but it could be useful to learn about the relation of different types of namespaces. The solution is based on the following facts.

- The “nsenter” command allows us to define a custom root directory (mount namespace) instead of using an existing mount namespace
- The “mount” command has a `--bind` flag which allows us to “bind mount” a folder to a new location. This is similar to what Docker does if you choose “bind” as the type of a volume. See [Bind mounts | Docker](#)
- There are some folders that are not part of the root filesystem, so when we mount the root filesystem we don't mount those folders. `/run` is on `tmpfs`, so it is stored in memory.
- Mounting a file over a symbolic link is not possible, but mounting over an empty file which is a target of a symbolic link works.



Author: Ákos Takács

First we will set the variables again with an additional `project_dir` which you can change if you want

```
sandboxKey=$(docker container inspect php --format '{{.NetworkSettings.SandboxKey}}')
pid=$(docker container inspect php --format '{{.State.Pid}}')

project_dir="$HOME/projects/netns"
```

Then we create the our project directory

```
mkdir -p "$project_dir"
cd "$project_dir"
```

Mount the system root to a local folder called “root”.

```
mkdir -p root
sudo mount --bind / root
```

Since “run” is on `tmpfs` and it wasn't mounted, we create an empty file to work as a placeholder for the target of the symbolic link at `/etc/resolv.conf`

```
sudo mkdir -p "root/run/systemd/resolve/"
sudo touch "root/run/systemd/resolve/stub-resolv.conf"
```

Now we can copy the `resolv.conf` that contains the actual name servers and mount it over our placeholder `stub-resolv.conf`.

```
cp "/run/systemd/resolve/resolv.conf" "resolv.conf"
sudo mount --bind "resolv.conf" "root/run/systemd/resolve/stub-resolv.conf"
```

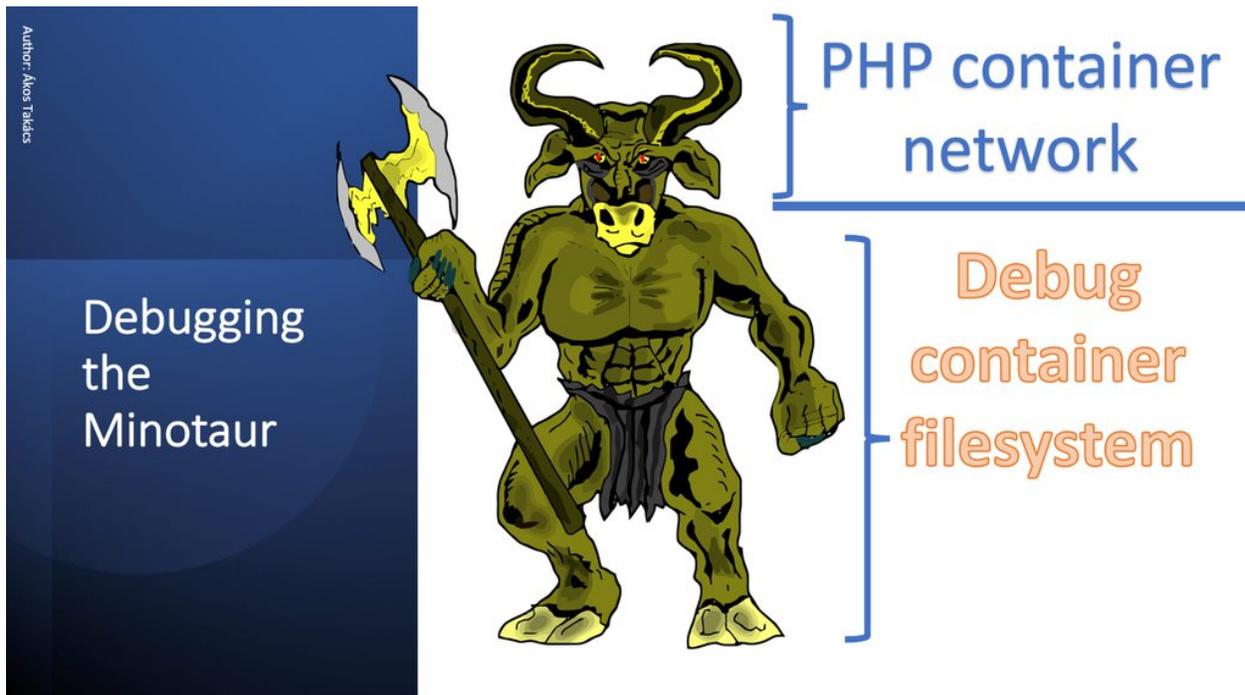
And finally we can run the following `nsenter` command.

```
sudo nsenter -n --root=$PWD/root --target=$pid ping dns.google
```

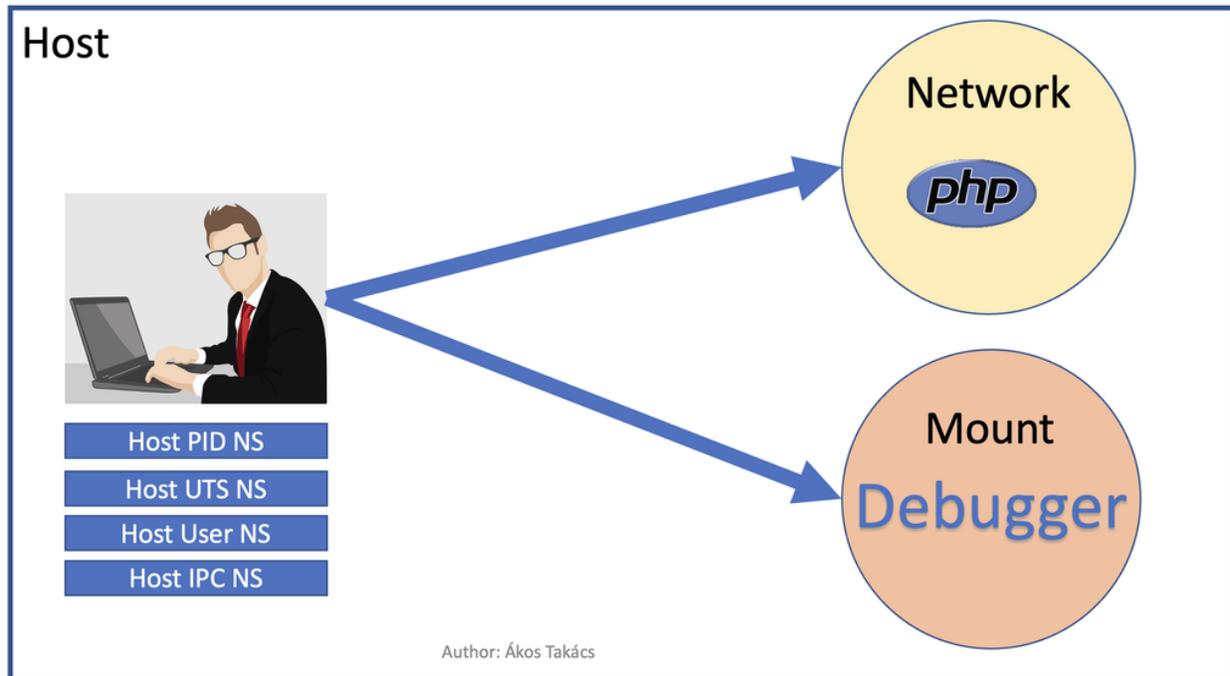
Now `nsenter` will use `$PWD/root` as the filesystem of the new mount namespace and use the network namespace of the PHP container to run `ping`.

```
PING dns.google (8.8.4.4) 56(84) bytes of data.
64 bytes from dns.google (8.8.4.4): icmp_seq=1 ttl=112 time=11.5 ms
64 bytes from dns.google (8.8.4.4): icmp_seq=2 ttl=112 time=12.1 ms
64 bytes from dns.google (8.8.4.4): icmp_seq=3 ttl=112 time=11.7 ms
```

14.3.6 Debugging the Minotaur



I call this technique “Debugging the Minotaur” because unlike before when we ran a new container to attach it to another container’s network namespace, we are still on the host and we use most of the host’s namespaces and we choose to use one container’s mount namespace (and only the mount namespace) and another container’s network namespace (and only the network namespace). As we were creating a Minotaur where the body of the Minotaur is the mount namespace of the debugger container with all of its tools and the head is the other container’s network namespace which we want to debug. To do this, we use only `nsenter` and nothing else.



We know that we can use an executable on the host's filesystem and run it in a network namespace. We can also choose the mount namespace and that can be the filesystem of a running container. First we want to have a running debugger container. [nicolaka/netshoot](#) is an excellent image to start a debugger container from. We need to run it in detached mode (`-d`) so it will run in the background (not attaching to the container's namespaces) and also in interactive mode (`-i`) so it will keep running instead of exiting immediately.

```
docker run -d -i --name debug nicolaka/netshoot:v0.9
```

Now we need to get the sandbox key for the network namespace and since we want to debug the PHP container, we will get the sandbox key from it. We also need something for the mount namespace of the debugger container. This is a good time to learn that if we have an existing process, we can find all of its namespaces using a path like this:

```
/proc/<PID>/ns/<NAMESPACE>
```

where `<PID>` is the process id and `<NAMESPACE>` in case of the discussed best known namespaces is one of the following: `mnt`, `net`, `pid`. We could use `/proc/$pid/ns/net` instead of the sandbox key, but in this example I will keep it to demonstrate that you can do both.

```
php_sandbox_key=$(docker container inspect php --format '{{ .NetworkSettings.SandboxKey }}')
debug_pid=$(docker container inspect debug --format '{{ .State.Pid }}')
```

Now that we have the variables, let's use `nsenter` a new way. So far we used the sandbox key only to help the `ip` command to recognize the network namespaces. Now we have to refer to it directly and `nsenter` can do that.

```
sudo nsenter --net=$php_sandbox_key --mount=/proc/$debug_pid/ns/mnt ping dns.google
```

This way we have a ping command running, but sometimes we need to do more debugging. The ping command is almost always available on Linux systems, although you can use `tshark` or `tcpdump` to see the network packets, but I prefer to use `tshark`. The following command will show us packets going through the debugger container's `eth0` interface so you can actually see the source of everything before those packets are reaching the `veth*` interface on the host. Since you can use `tshark` from the debugger container, you don't have to install it. In case you have a more

advanced debugger script which for some reason needs to access other namespaces on the host, you can do that too.

```
sudo nsenter --net=$php_sandbox_key --mount=/proc/$debug_pid/ns/mnt tshark -i eth0
```

As a final step, open a new terminal and generate some traffic on the container network. Get the ip address of the container and use curl to get the main page of the website in the container.

```
ip=$(docker container inspect php --format '{{ .NetworkSettings.IPAddress }}')  
curl "$ip"
```

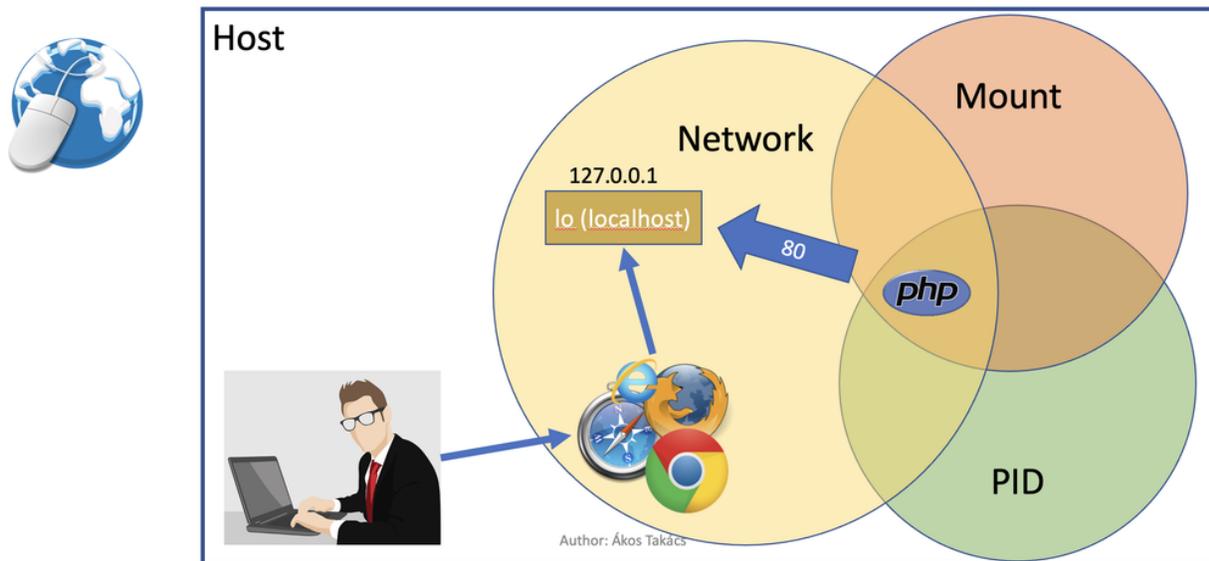
As a result, in the previous terminal window you should see the request packets and the response.

14.4 Testing a web-based application without internet in a container

14.4.1 Running a web browser in a net namespace on Linux (Docker CE)

If you are running Docker CE on Linux (not Docker Desktop), you can just use a web browser on your host operating system and run it in the network namespace of a container. If the application inside is listening on localhost, you can access it from the web browser in the same network namespace.

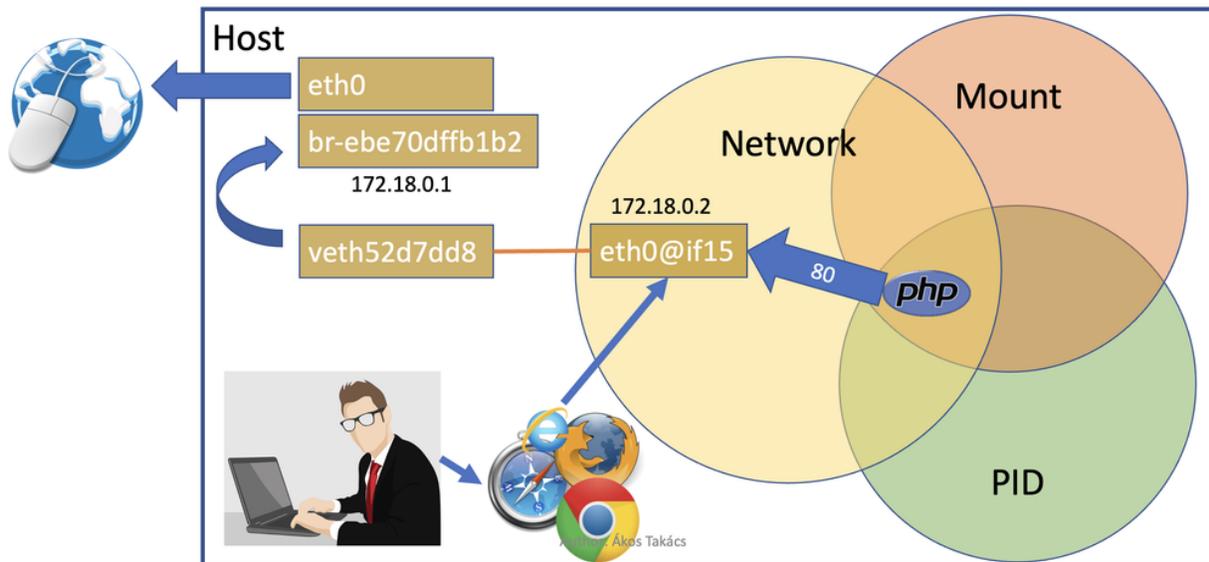
Web browser and net namespace + network "none"



```
docker run -d --name php-networkless --network none itsziget/phar-examples:1.0  
pid_networkless=$(docker container inspect php --format '{{ .State.Pid }}')  
sudo nsenter --net=/proc/$pid_networkless/ns/net curl localhost
```

Or sometimes you know that the frontend is safe to use, so you only want to test the backend.

Web browser and net namespace + internal network



In that case you can run the container with network, but only with an “internal” network, so the host and the container can communicate, but no traffic will be forwarded to the internet from the Docker bridge. This way you can run your browser “on the host” and use the container’s ip address instead of “localhost”.

Note: Actually everything is running on the host. Only the isolated processes will see it differently.

You need to

- create an internal network,
- run the container using the internal network
- get the ip address of the container
- open your web browser or use curl to access the website

```
docker network create internal --internal
docker run -d --name php-internal --network internal itsziget/phar-examples:1.0
ip_internal=$(docker container inspect php-internal --format '{{ .NetworkSettings.
↳Networks.internal.IPAddress }}')
curl "$ip_internal"
```

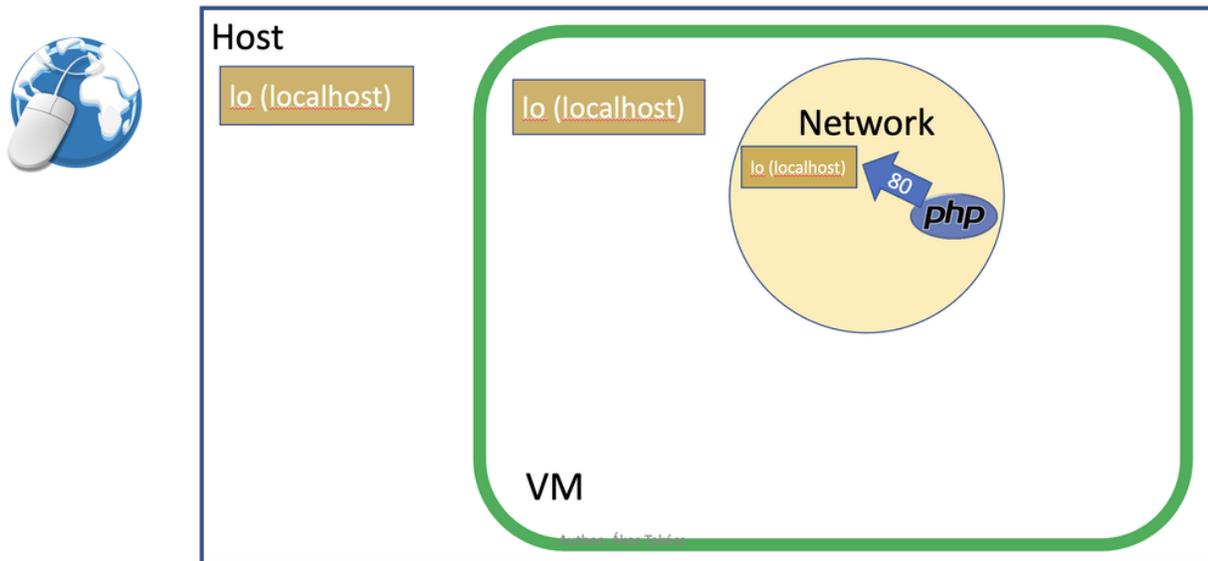
Since curl will not execute javascript, you can even check the generated source code, but nothing in the container will be able to send request to the outside world except the host machine:

```
docker exec php-internal ping 8.8.8.8
```

14.4.2 Running a web browser in a net namespace in a VM (Docker Desktop)

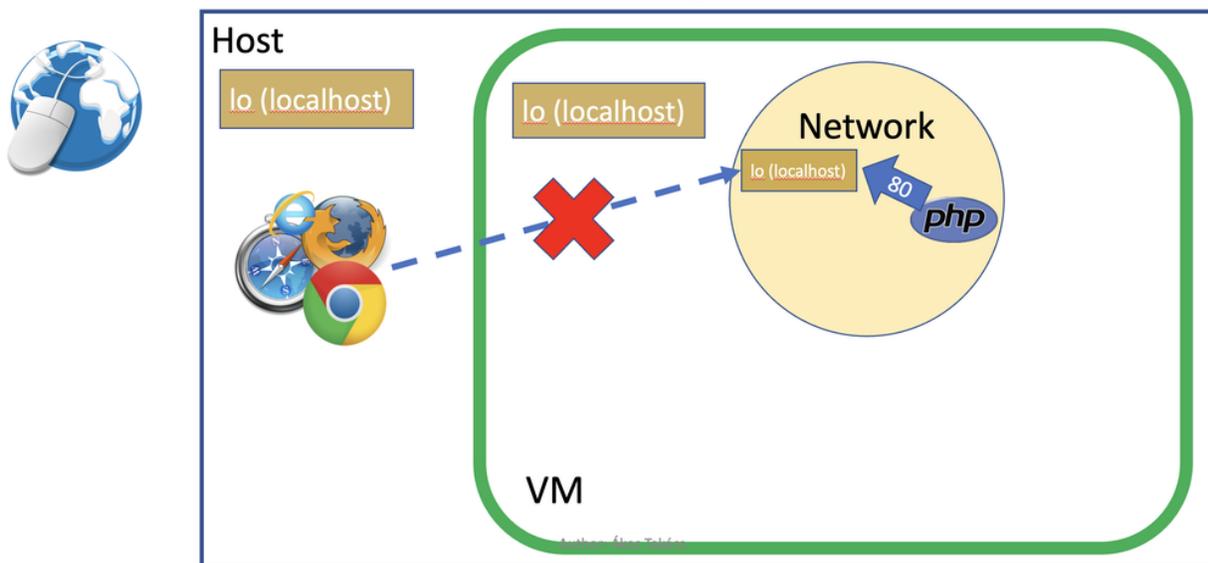
When you start to use Docker Desktop, one of the most important facts is that your containers will run in a virtual machine even on Linux (See: *Getting Started: Docker Desktop*). It means your actual host, the virtual machine and the container's will have their own "localhost".

Docker Desktop + network "none" from the network point of view



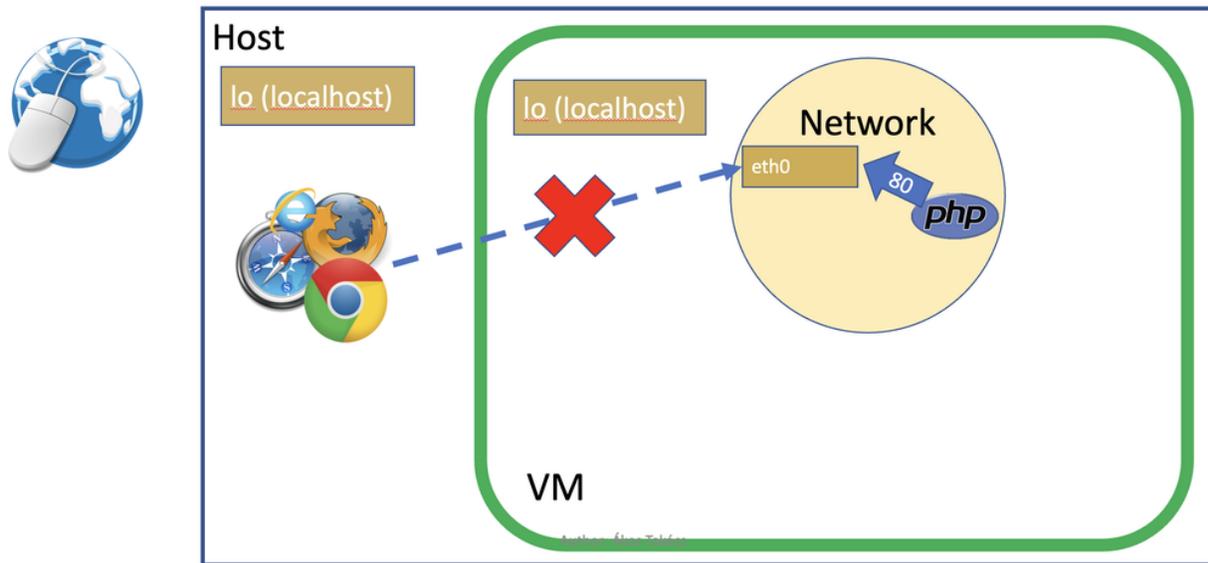
The network namespaces will be in that virtual machine, so you can't just run your web browser on your host operating system inside the network namespace.

Docker Desktop + network "none" from the network point of view



You can't even run the web browser in the virtual machine (in case of Docker Desktop) since that is just a server based on LinuxKit without GUI inside so you can't simply just use an internal network and connect to the IP address from the browser.

Docker Desktop + internal network from the network point of view

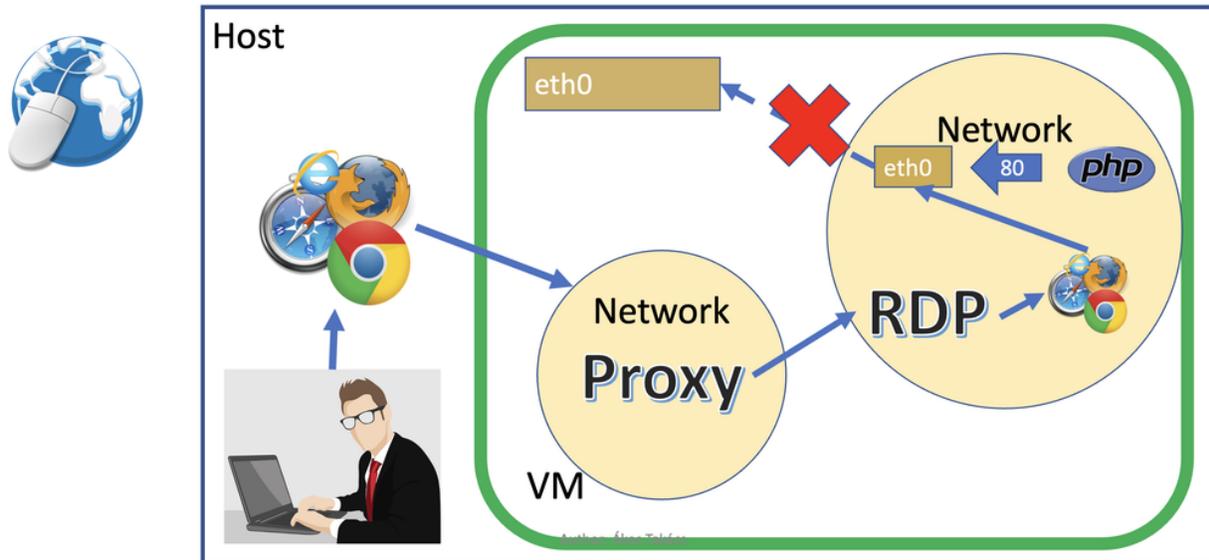


We need a much more complex solution which requires everything that we have learnt so far and more.

- We know that our PHP app has to run in a container without internet access
- We also know that we can achieve that by using internal networks or no network at all except loopback interface.
- Since Docker Desktop runs containers in a virtual machine, we definitely need network in the PHP container so we can access it from the outside. It means we obviously need to forward a port from the host to Docker Desktop's virtual machine, but we have also learnt that internal networks *do not accept forwarded ports*.
- We can however run a container with only an internal network and a proxy container with an internal and a public network which will be accessible from the outside. This container will forward all traffic to another container in the internal network.
- There is a way to run a web browser in a container and you can run this container in the PHP container's network namespace. The problem is that you need to access the graphical interface inside the container.
- Fortunately there is also a sponsored OSS project called [linuxserver/firefox](#). This project lets you run Firefox and a remote desktop server in the container.

How will this all look like? The following diagram illustrates it.

Docker Desktop + internal network + proxy



- You will use a web browser on the host as a remote desktop client to access the forwarded port of the proxy server on the IP address of the public network.
- The PHP container will have an internal network
- The Firefox container with the remote desktop client will use the network namespace of the PHP container so Firefox will not have internet access.
- The proxy server (with both internal and public network) will forward your request to the PHP container's network namespace to access the remote desktop server.
- The remote desktop server will stream back the screen only through the proxy server so the graphical interface of the containerized Firefox will appear in the web browser running on your host. If a harmful application tries to use JavaScript to access another website it won't be able to since all you can see is a picture of a web browser running in an isolated environment.

I have created a compose file which we can use to create this whole environment.

Create a project folder anywhere you like. This is mine:

```
project_dir="$HOME/Data/projects/testprojects/netns"
mkdir -p "$project_dir"
cd "$project_dir"
```

Download the compose file from GitHub

```
curl --output compose.yml \
  https://gist.githubusercontent.com/rimelek/91702f6e9c9e0ae75a72a42211099b63/raw/
  ↪339beaf0c50790e86ab8a011ed298c250da3b7ec/compose.yml
```

Compose file content:

```
networks:
  default:
```

(continues on next page)

(continued from previous page)

```
    internal: true
  public:

services:
  php:
    image: itsziget/phar-examples:1.0

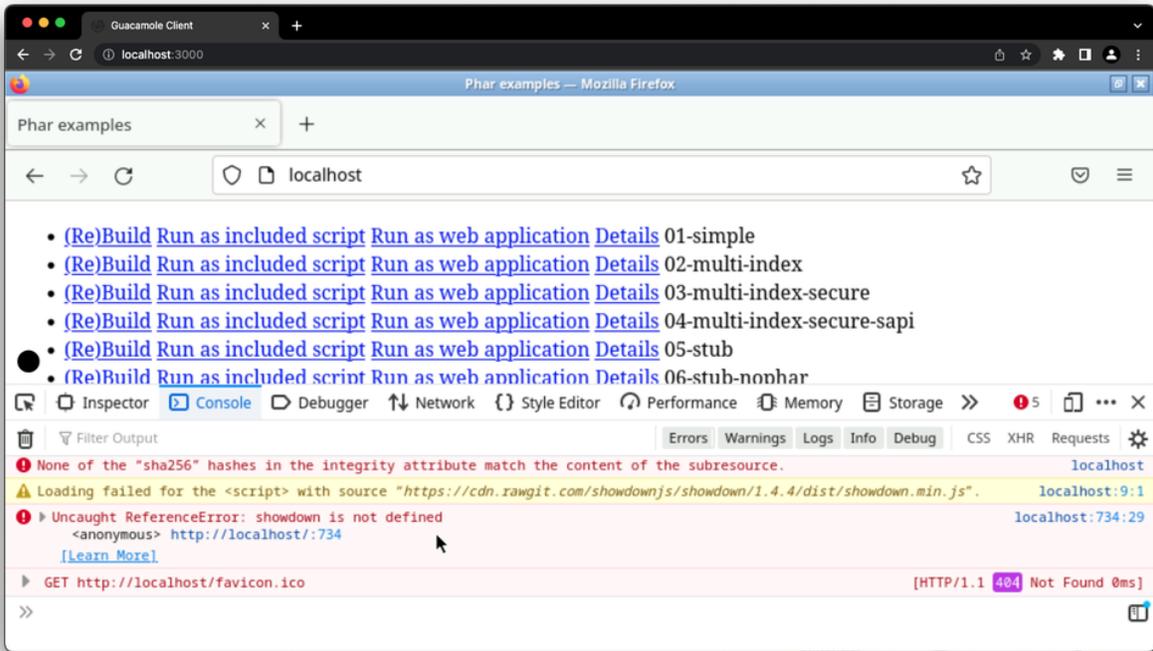
  firefox:
    network_mode: service:php
    environment:
      PUID: 1000
      PGID: 1000
      TZ: Europe/London
    shm_size: "1gb"
    image: lscr.io/linuxserver/firefox:101.0.1

  proxy:
    image: alpine/socat:1.7.4.4-r0
    command: "TCP-LISTEN:3000, fork, reuseaddr TCP:php:3000"
    ports:
      - 3000:3000
    networks:
      - default
      - public
```

Start the containers:

```
docker compose up -d
```

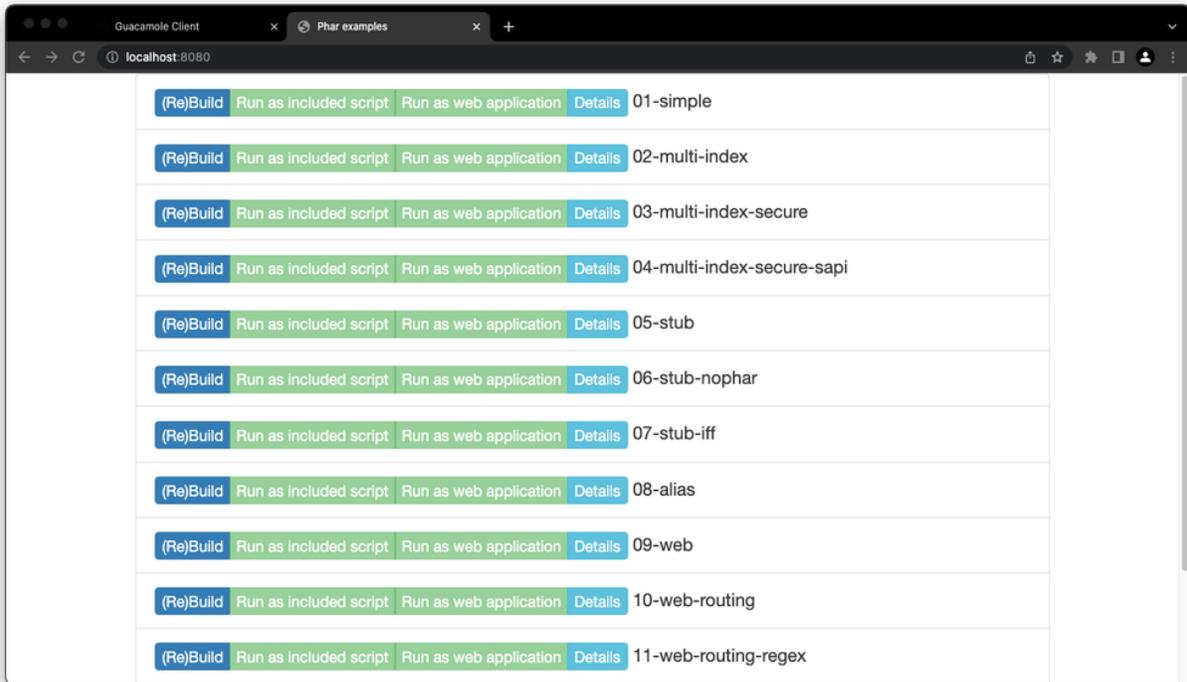
If you open `localhost:3000` in your browser, you will see the containerized browser and the demo application without CSS and JavaScript since those files would be loaded from an external source and they are not available.



Now that you know it is trying to load CSS and some harmless JavaScripts, you can run it with a public network

```
docker run -d --name php-internet -p 8080:80 itsziget/phar-examples:1.0
```

and open it in an other tab on port 8080.

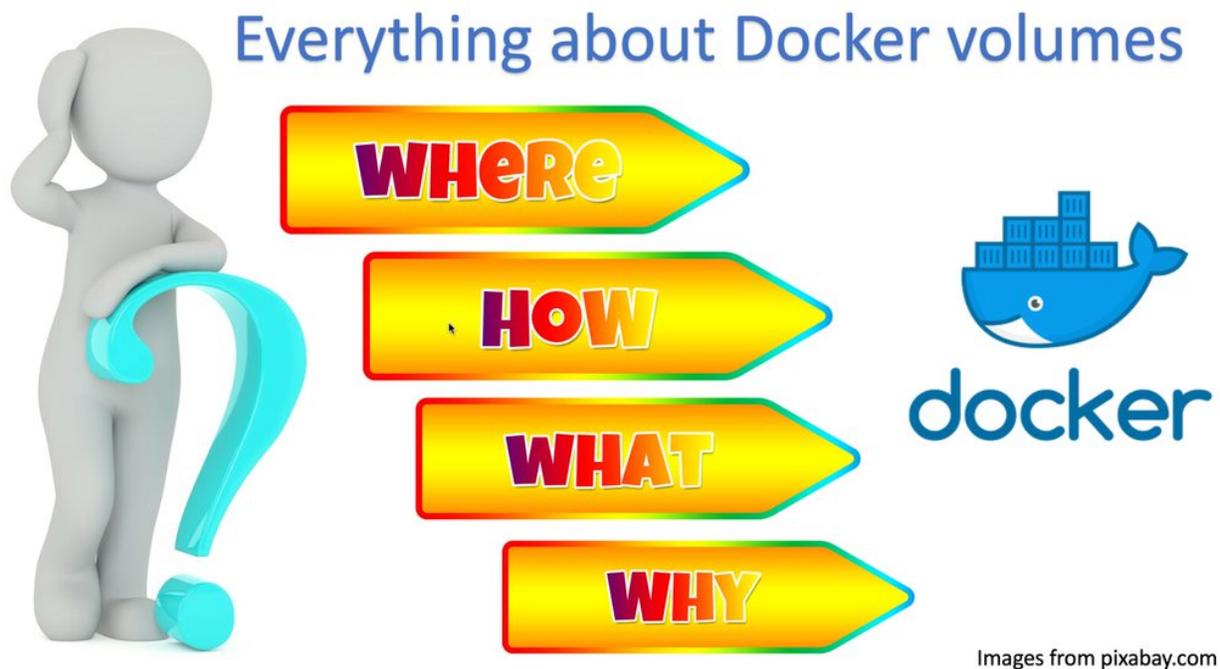


14.5 Used sources

- <https://www.redhat.com/sysadmin/net-namespaces>
- <https://serverfault.com/a/704717>
- <https://serverfault.com/questions/1007562/linux-networking-bridge-with-veths-not-able-to-send-outbound-packets>
- <https://github.com/p8952/bocker/blob/master/bocker>
- <https://collabnix.com/a-beginners-guide-to-docker-networking/>

14.6 Recommended similar tutorials

- <https://iximiuz.com/en/posts/container-networking-is-simple/>



EVERYTHING ABOUT DOCKER VOLUMES

15.1 Intro

“Where are the Docker volumes?” This question comes up a lot on the [Docker forum](#). There is no problem with curiosity, but this is usually asked when someone wants to edit or at least read files directly on the volume from a terminal or an IDE, but not through a container. So I must start with a statement:

Important: You should never handle files on a volume directly without entering a container, unless there is an emergency, and even then, only at your own risk.

Why I am saying it, you will understand if you read the next sections. Although the original goal with this tutorial was to explain where the volumes are, it is hard to talk about it without understanding what the volumes are and what different options you have when using volumes. As a result of that, by reading this tutorial, you can learn basically everything about the local volumes, but you can also search for volume plugins on [Docker Hub](#).

15.2 Where does Docker store data?

Before we talk about the location of volumes, we first have to talk about the location of all data that Docker handles. When I say “Docker”, I usually mean “Docker CE”.

Docker CE is the community edition of Docker and can run directly on Linux. It has a data root directory, which is the following by default:

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

You can change it in the [daemon configuration](#), so if it is changed on your system, you will need to replace this folder in the examples I show. To find out what the data root is, run the following command:

```
docker info --format '{{ .DockerRootDir }}'
```

In case of Docker Desktop of course, you will always have a virtual machine, so the path you get from the above command will be in the virtual machine.

15.3 What is a Docker volume?

For historical reasons, the concept of volumes can be confusing. There is a [page in the documentation](#) which describes what volumes are, but when you see a Compose file or a docker run command, you see two types of volumes, but only one of them is actually a volume.

Example Compose file:

```
services:
  server:
    image: httpd:2.4
    volumes:
      - ./docroot:/usr/local/apache2/htdocs
```

Did I just define a volume? No. It is a [bind mount](#). Let's just use the long syntax:

```
services:
  server:
    image: httpd:2.4
    volumes:
      - type: bind
        source: ./docroot
        target: /usr/local/apache2/htdocs
```

The “volumes” section should have been “storage” or “mounts” to be more clear. In fact, the “docker run” command supports the `--mount` option in addition to `-v` and `--volume`, and only `--mount` supports the `type` parameter to directly choose between volume and bind mount.

Then what do we call a volume? Let's start with answering another question. What do we not call a volume? A file can never be a volume. A volume is always a directory, and it is a directory which is created by Docker and handled by Docker throughout the entire lifetime of the volume. The main purpose of a volume is to populate it with the content of the directory to which you mount it in the container. That's not the case with bind mounts. Bind mounts just completely override the content of the mount point in the container, but at least you can choose where you want to mount it from.

You should also know that you can disable copying data from the container to your volume and use it as a simple bind mount, except that Docker creates it in the Docker data root, and when you delete the volume after you wrote something on it, you will lose the data.

```
volumes:
  docroot:

services:
  server:
    image: httpd:2.4
    volumes:
      - type: volume
        source: docroot
        target: /usr/local/apache2/htdocs
        volume:
          nocopy: true
```

You can find this and other parameters in the [documentation of volumes in a compose file](#). Scroll down to the “Long syntax” to read about “nocopy”.

15.4 Custom volume path

15.4.1 Custom volume path overview

There is indeed a special kind of volume which seems to mix bind mounts and volumes. The following example will assume you are using Docker CE on Linux.

```
volume_name="test-volume"
source="$PWD/$volume_name"

mkdir -p "$volume_name"
docker volume create "$volume_name" \
  --driver "local" \
  --opt "type=none" \
  --opt "device=$source" \
  --opt "o=bind"
```

Okay, so you created a volume and you also specified where the source directory is (device), and you specified that it is a bind mount. Don't worry, you find it confusing because it is confusing. `o=bind` doesn't mean that you will bind mount a directory into the container, which will always happen, but that you will bind mount the directory to the path where Docker would have created the volume if you didn't define the source.

This is basically the same what you would do on Linux with the `mount` command:

```
mount -o bind source/ target/
```

Without `-o bind` the first argument must be a block device. This is why we use the "device" parameter, even though we mount a folder.

This is one way to know where the Docker volume is.

Note: Even the the above example assumed Linux, custom volume path would work on other operating systems as well, since Docker Desktop would mount the required path into the virtual machine.

Let's just test if it works and inspect the volume:

```
docker volume inspect test-volume
```

You will get a json like this:

```
[
  {
    "CreatedAt": "2024-01-05T00:55:15Z",
    "Driver": "local",
    "Labels": {},
    "Mountpoint": "/var/lib/docker/volumes/test-volume/_data",
    "Name": "test-volume",
    "Options": {
      "device": "/home/ta/test-volume",
      "o": "bind",
      "type": "none"
    },
    "Scope": "local"
  }
]
```

(continues on next page)

(continued from previous page)

```
}  
]
```

The “Mountpoint” field in the json is not the path in a container, but the path where the specified device should be mounted at. In our case, the device is actually a directory. So let’s see the content of the mount point:

```
sudo ls -la $(docker volume inspect test-volume --format '{{ .Mountpoint }}')
```

You can also check the content of the source directory:

```
ls -la test-volume/
```

Of course, both are empty as we have no container yet. How would Docker know what the content should be? As we already learned it, we need to mount the volume into a container to populate the volume.

```
docker run \  
-d --name test-container \  
-v test-volume:/usr/local/apache2/htdocs \  
httpd:2.4
```

Check the content in the container:

```
docker exec test-container ls -lai /usr/local/apache2/htdocs/
```

Output:

```
total 16  
256115 drwxr-xr-x 2 root root 4096 Jan 5 00:33 .  
5112515 drwxr-xr-x 1 www-data www-data 4096 Apr 12 2023 ..  
256139 -rw-r--r-- 1 501 staff 45 Jun 11 2007 index.html
```

Notice that we added the flag “i” to the “ls” command so we can see the inode number, which identifies the files and directories on the filesystem in the first column.

Check the directory created by Docker:

```
sudo ls -lai $(docker volume inspect test-volume --format '{{ .Mountpoint }}')
```

```
256115 drwxr-xr-x 2 root root 4096 Jan 5 00:33 .  
392833 drwx----x 3 root root 4096 Jan 5 00:55 ..  
256139 -rw-r--r-- 1 501 staff 45 Jun 11 2007 index.html
```

As you can see, only the parent directory is different, so we indeed see the same files in the container and in the directory created by Docker. Now let’s check our source directory.

```
ls -lai test-volume/
```

Output:

```
total 12  
256115 drwxr-xr-x 2 root root 4096 Jan 5 00:33 .  
255512 drwxr-xr-x 11 ta ta 4096 Jan 5 00:32 ..  
256139 -rw-r--r-- 1 501 staff 45 Jun 11 2007 index.html
```

Again, the same files, except the parent. We confirmed, that we could create an empty volume directory, we could populate it when we started a container and mounted the volume, and the files appeared where Docker creates volumes. Now let's check one more thing. Since this is a special volume where we defined some parameters, there is an `opts.json` right next to `_data`

```
sudo cat "$(dirname "$(docker volume inspect test-volume --format '{{ .Mountpoint }}')")"
↪ /opts.json
```

Output:

```
{"MountType":"none","MountOpts":"bind","MountDevice":"/home/ta/test-volume","Quota":{"Size":0}}
```

Now remove the test container:

```
docker container rm -f test-container
```

Check the directory created by Docker:

```
sudo ls -lai $(docker volume inspect test-volume --format '{{ .Mountpoint }}')
```

It is empty now.

```
392834 drwxr-xr-x 2 root root 4096 Jan  5 00:55 .
392833 drwx-----x 3 root root 4096 Jan  5 00:55 ..
```

And notice that even the inode has changed, not just the content disappeared. On the other hand, the directory we created is untouched and you can still find the `index.html` there.

15.4.2 Avoid accidental data loss on volumes

Let me show you an example using Docker Compose. The compose file would be the following:

```
volumes:
  docroot:
    driver: local
    driver_opts:
      type: none
      device: ./docroot
      o: bind

services:
  httpd:
    image: httpd:2.4
    volumes:
      - type: volume
        source: docroot
        target: /usr/local/apache2/htdocs
```

You can populate `./docroot` in the project folder by running

```
docker compose up -d
```

You will then find `index.html` in the `docroot` folder. You probably know that you can delete a compose project by running `docker compose down`, and delete the volumes too by passing the flag `-v`.

```
docker compose down -v
```

You can run it, and the volume will be destroyed, but not the content of the already populated “docroot” folder. It happens, because the folder which is managed by Docker in the Docker data root does not physically have the content. So the one that was managed by Docker could be safely removed, but it didn’t delete your data.

15.5 Docker CE volumes on Linux

This question seems to be already answered in the previous sections, but let’s evaluate what we learned and add some more details.

So you can find the local default volumes under `/var/lib/docker/volumes` if you didn’t change the *data root*. For the sake of simplicity of the commands, I will keep using the default path.

The Docker data root is not accessible by normal users, only by administrators. Run the following command:

```
sudo ls -la /var/lib/docker/volumes
```

You will see something like this:

```
total 140
drwx-----x 23 root root  4096 Jan  5 00:55 .
drwx--x--- 13 root root  4096 Dec 10 14:27 ..
drwx-----x  3 root root  4096 Jan 25  2023 _
↪0c5f9867e761f6df0d3ea9411434d607bb414a69a14b3f240f7bb0ffb85f0543
drwx-----x  3 root root  4096 Sep 19 13:15 _
↪1c963fb485fbbd5ce64c6513186f2bc30169322a63154c06600dd3037ba1749a
...
drwx-----x  3 root root  4096 Jan  5  2023 apps_cache
brw-----  1 root root    8, 1 Dec 10 14:27 backingFsBlockDev
-rw-----  1 root root 65536 Jan  5 00:55 metadata.db
```

These are the names of the volumes and two additional special files.

- `backingFsBlockDev`
- `metadata.db`

We are not going to discuss it in more details. All you need to know at this point is that this is where the volume folders are. Each folder has a sub-folder called “_data” where the actual data is, and there could be an `opts.json` with metadata next to the “_data” folder.

Note: When you use *rootless Docker*, the Docker data root will be in your user’s home.

```
$HOME/.local/share/docker
```

15.6 Docker Desktop volumes

Docker Desktop volumes are different depending on the operating system and whether you want to run Linux containers or Windows containers.

Docker Desktop always runs a virtual machine for Linux containers and runs Docker CE in it in a quite complicated way, so your volumes will be in the virtual machine too. Because of that fact when you want to access the volumes, you either have to find a way to run a shell in the virtual machine, or find a way to share the filesystem on the network and use your filebrowser, IDE or terminal on the host.

Parts of what I show here and more can be found in my presentation which I gave on the 6th Docker Community All-Hands. Tyler Charboneau wrote a [blog post](#) about it, but you can also [find the video](#) in the blog post.

15.6.1 Docker Desktop volumes on macOS

On macOS, you can only run Linux containers and there is no such thing as macOS container yet (2024. january).

You can get to the volumes folder by running the following command:

```
docker run --rm -it --privileged --pid host ubuntu:22.04 \
  nsenter --all -t 1 \
  sh -c 'cd /var/lib/docker/volumes && sh'
```

Or just simply mount that folder to a container:

```
docker run --rm -it \
  -v /var/lib/docker/volumes:/var/lib/docker/volumes \
  --workdir /var/lib/docker/volumes \
  ubuntu:22.04 \
  bash
```

You can also run an NFS server in a container that mounts the volumes so you can mount the remote fileshare on the host. The following compose .yml file can be used to run the NFS server:

```
services:
  nfs-server:
    image: openebs/nfs-server-alpine:0.11.0
    volumes:
      - /var/lib/docker/volumes:/mnt/nfs
    environment:
      SHARED_DIRECTORY: /mnt/nfs
      SYNC: sync
      FILEPERMISSIONS_UID: 0
      FILEPERMISSIONS_GID: 0
      FILEPERMISSIONS_MODE: "0755"
    privileged: true
    ports:
      - 127.0.0.1:2049:2049/tcp
      - 127.0.0.1:2049:2049/udp
```

Start the server:

```
docker compose up -d
```

Create the mount point on the host:

```
sudo mkdir -p /var/lib/docker/volumes
sudo chmod 0700 /var/lib/docker
```

Mount the base directory of volumes:

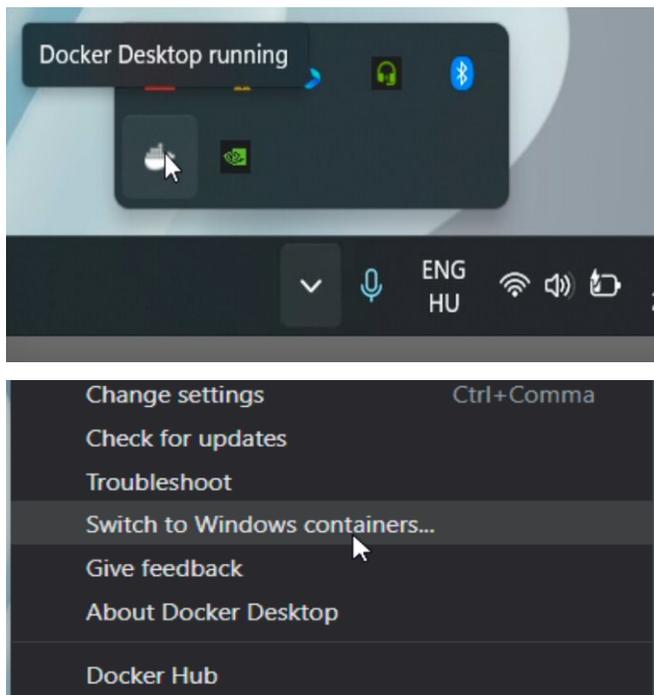
```
sudo mount -o vers=4 -t nfs 127.0.0.1:/ /var/lib/docker/volumes
```

And list the content:

```
sudo ls -l /var/lib/docker/volumes
```

15.6.2 Docker Desktop volumes on Windows

Docker Desktop on Windows allows you to switch between Linux containers and Windows containers.



To find out which one you are using, run the following command:

```
docker info --format '{{ .OSType }}'
```

If it returns “windows”, you are using Windows containers, and if it returns “linux”, you are using Linux containers.

Linux containers

Since Linux containers always require a virtual machine, you will have your volumes in the virtual machine the same way as you would on macOS. The difference is how you can access them. A common way is through a Docker container. Usually I would run the following command.

```
docker run --rm -it --privileged --pid host ubuntu:22.04 `
  nsenter --all -t 1 `
  sh -c 'cd /var/lib/docker/volumes && sh'
```

But if you have an older kernel in WSL2 which doesn't support the time namespace, you can get an error message like:

```
nsenter: cannot open /proc/1/ns/time: No such file or directory
```

If that happens, make sure you have the latest kernel in WSL2. If you built a custom kernel, you may need to rebuild it from a new version.

If you can't update the kernel yet, exclude the time namespace, and run the following command:

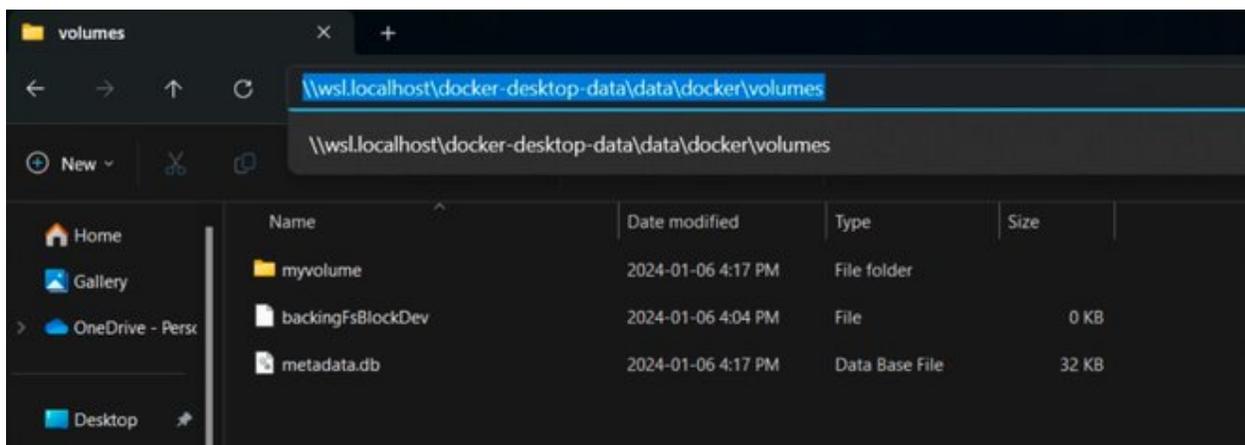
```
docker run --rm -it --privileged --pid host ubuntu:22.04 `
  nsenter -m -n -p -u -t 1 `
  sh -c 'cd /var/lib/docker/volumes && sh'
```

You can simply mount the base directory in a container the same way as we could on macOS:

```
docker run --rm -it `
  -v /var/lib/docker/volumes:/var/lib/docker/volumes `
  --workdir /var/lib/docker/volumes `
  ubuntu:22.04 `
  bash
```

We don't need to run a server in a container to share the volumes, since it works out of the box in WSL2. You can just open the Windows explorer and go to

```
\\wsl.localhost\docker-desktop-data\data\docker\volumes
```



Warning: WSL2 let's you edit files more easily even if the files are owned by root on the volume, so do it at your own risk. My recommendation is using it only for debugging.

Windows Containers

Windows containers can mount their volumes from the host. Let's create a volume

```
docker volume create windows-volume
```

Inspect the volume:

You will get something like this:

```
[
  {
    "CreatedAt": "2024-01-06T16:27:03+01:00",
    "Driver": "local",
    "Labels": null,
    "Mountpoint": "C:\\ProgramData\\Docker\\volumes\\windows-volume\\_data",
    "Name": "windows-volume",
    "Options": null,
    "Scope": "local"
  }
]
```

So now you got the volume path on Windows in the “Mountpoint” field, but you don't have access to, it unless you are Administrator. The following command works only from Powershell run as Administrator

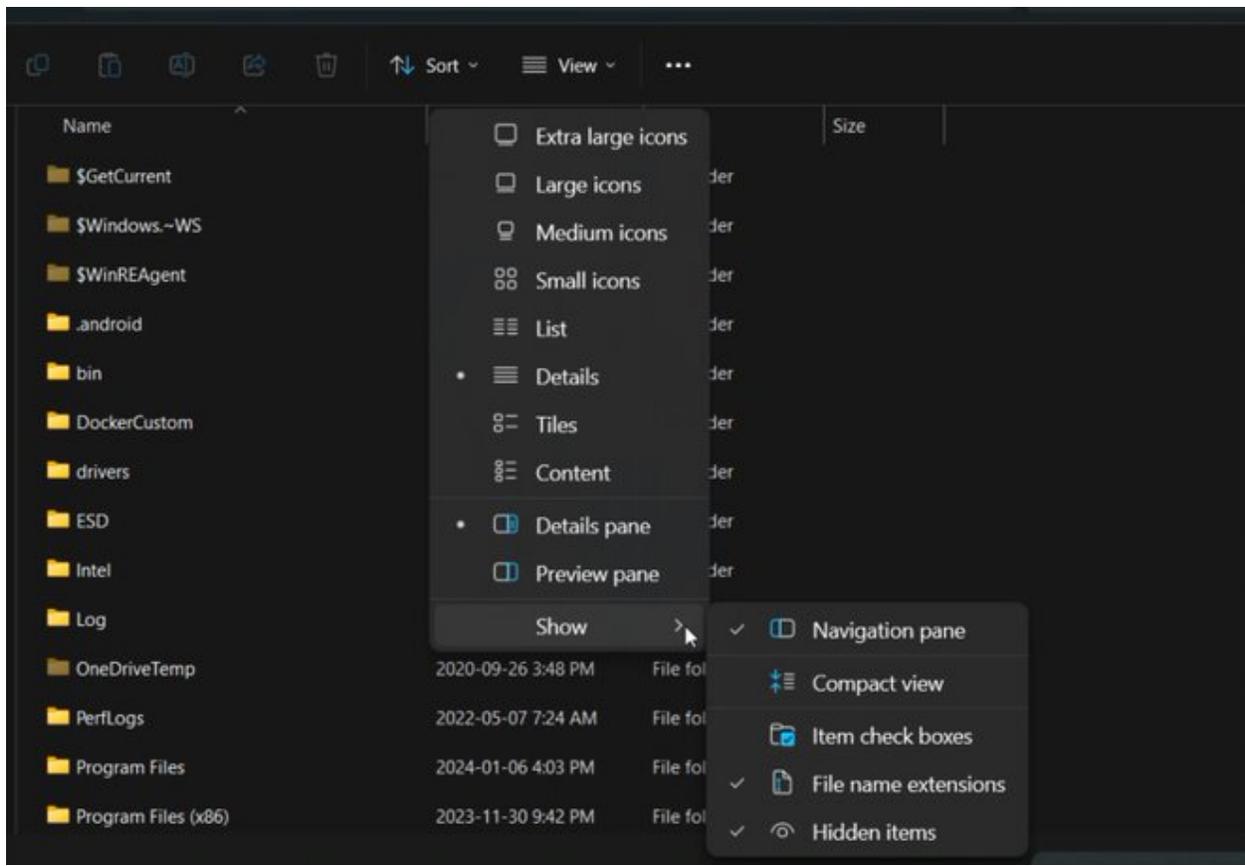
```
cd $(docker volume inspect windows-volume --format '{{ .Mountpoint }}')
```

If you want to access it from Windows Explorer, you can first go to

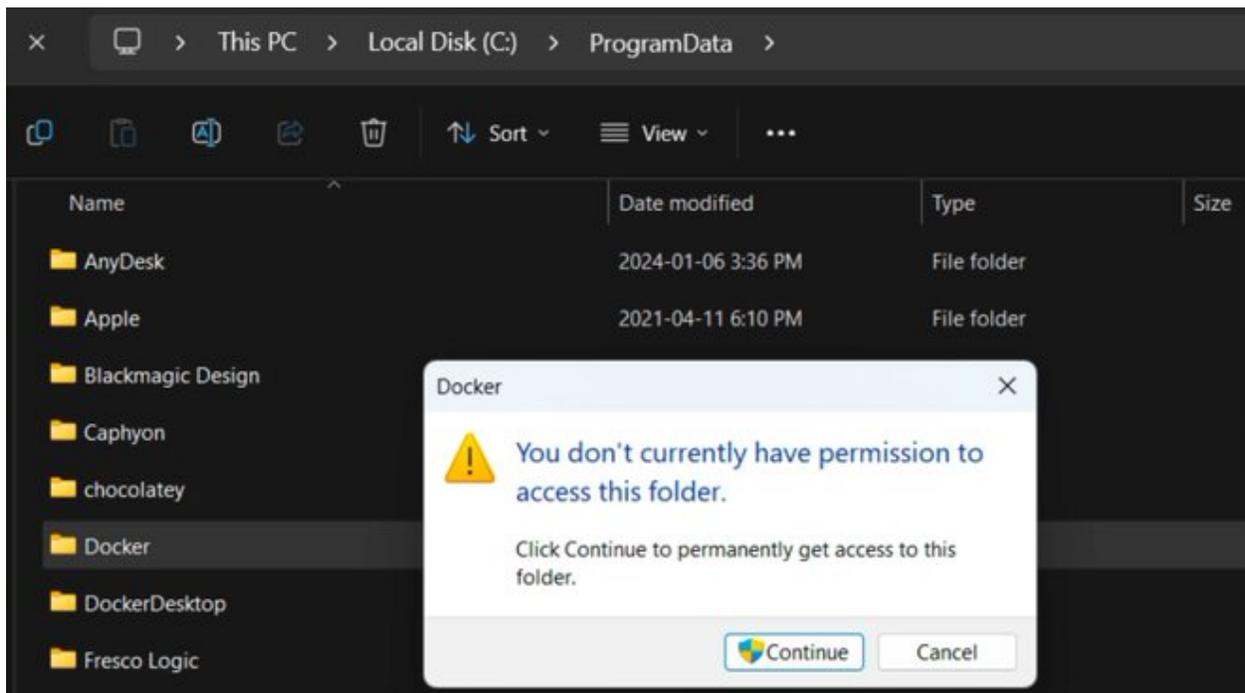
```
C:\ProgramData
```

Note: This folder is hidden by default, so if you want to open it, just type the path manually in the navigation bar, or enable hidden folders on Windows 11 (works differently on older Windows):

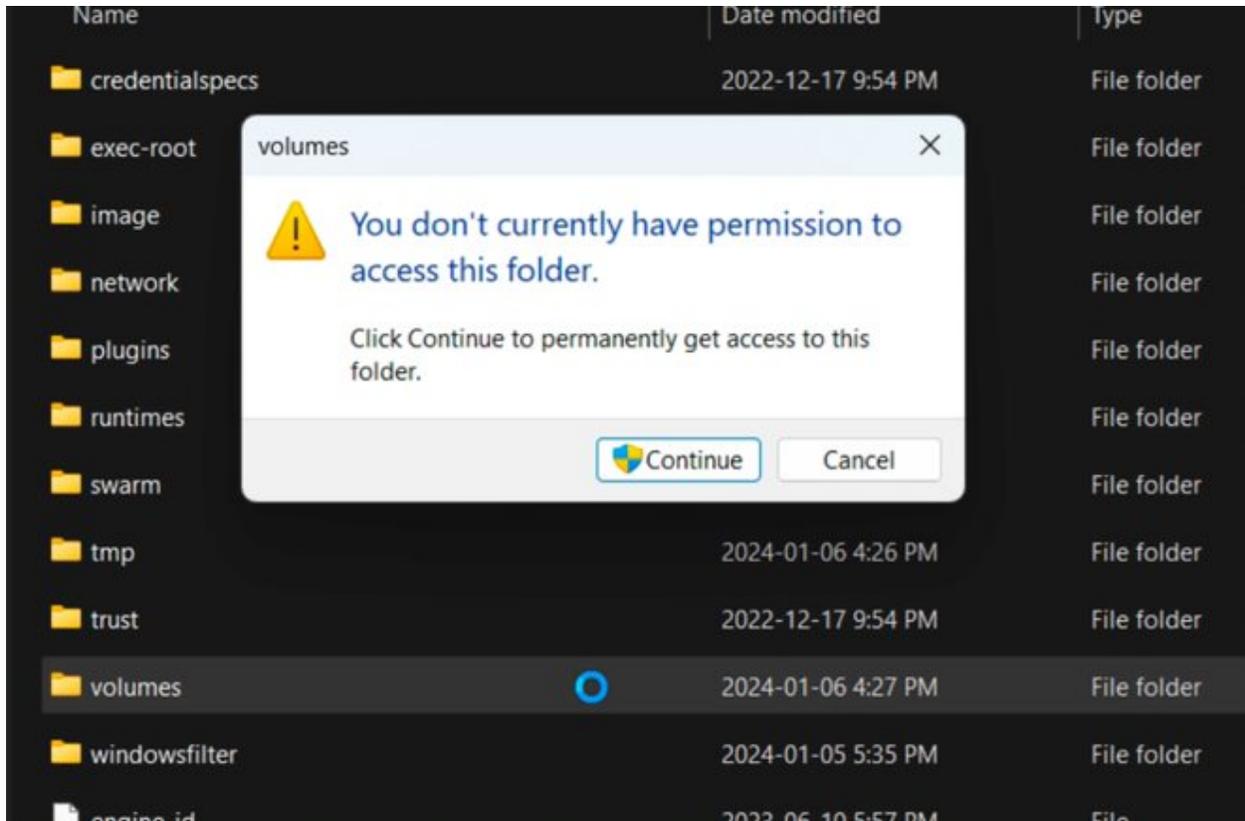
```
Menu bar » View » Show » Hidden Items
```



Then try to open the folder called “Docker” which gives you a prompt to ask for permission to access to folder.



and then try to open the folder called “volumes” which will do the same.



After that you can open any Windows container volume from Windows explorer.

15.6.3 Docker Desktop volumes on Linux

On Windows, you could have Linux containers and Window containers, so you had to switch between them. On Linux, you can install Docker CE in rootful and rootless mode, and you can also install Docker Desktop. These are 3 different and separate Docker installations and you can switch between them by changing context or logging in as a different user.

You can check the existing contexts by running the following command:

```
docker context ls
```

If you have Docker CE installed on your Linux, and you are logged in as a user who installed the rootless Docker, and you also have Docker Desktop installed, you can see at least the following three contexts:

NAME	TYPE	DESCRIPTION	DOCKER_
↔ENDPOINT		KUBERNETES ENDPOINT ORCHESTRATOR	
default	moby	Current DOCKER_HOST based configuration	unix://
↔/var/run/docker.sock			
desktop-linux *	moby	Docker Desktop	unix://
↔/home/ta/.docker/desktop/docker.sock			
rootless	moby	Rootless mode	unix://
↔/run/user/1000/docker.sock			

In order to use Docker Desktop, you need to switch to the context called “desktop-linux”.

```
docker context use desktop-linux
```

Important: The default is usually rootful Docker CE and the other two are obvious. Only the rootful Docker CE needs to run as root, so if you want to interact with Docker Desktop, don't make the mistake of running the docker commands with sudo:

```
sudo docker context ls
```

NAME	TYPE	DESCRIPTION	DOCKER_
↪ENDPOINT	KUBERNETES	ENDPOINT ORCHESTRATOR	
default *	moby	Current DOCKER_HOST based configuration	unix://
↪/var/run/docker.sock			

In terms of accessing volumes, Docker Desktop works similarly on macOS and Linux, so you have the following options:

Run a shell in the virtual machine using nsenter:

```
docker run --rm -it --privileged --pid host ubuntu:22.04 \
  nsenter --all -t 1 \
  sh -c 'cd /var/lib/docker/volumes && sh'
```

Or just simply mount that folder to a container:

```
docker run --rm -it \
  -v /var/lib/docker/volumes:/var/lib/docker/volumes \
  --workdir /var/lib/docker/volumes \
  ubuntu:22.04 \
  bash
```

And of course, you can use the nfs server compose project with the following `compose.yml`

```
services:
  nfs-server:
    image: openebs/nfs-server-alpine:0.11.0
    volumes:
      - /var/lib/docker/volumes:/mnt/nfs
    environment:
      SHARED_DIRECTORY: /mnt/nfs
      SYNC: sync
      FILEPERMISSIONS_UID: 0
      FILEPERMISSIONS_GID: 0
      FILEPERMISSIONS_MODE: "0755"
    privileged: true
    ports:
      - 127.0.0.1:2049:2049/tcp
      - 127.0.0.1:2049:2049/udp
```

and prepare the mount point. Remember, you can have Docker CE running as root, which means `/var/lib/docker` probably exists, so let's create the mount point as `/var/lib/docker-desktop/volumes`:

```
sudo mkdir -p /var/lib/docker-desktop/volumes
sudo chmod 0700 /var/lib/docker-desktop
```

And mount it:

```
sudo mount -o vers=4 -t nfs 127.0.0.1:/ /var/lib/docker-desktop/volumes
```

And check the content:

```
sudo ls -l /var/lib/docker-desktop/volumes
```

You could ask why we mount the volumes into a folder on the host, which requires sudo if the docker commands don't. The reason is that you will need sudo to use the mount command, so it shouldn't be a problem to access the volumes as root.

15.7 Editing files on volumes

15.7.1 The danger of editing volume contents outside a container

Now you know how you can find out where the volumes are. You also know how you can create a volume with a custom path, even if you are using Docker Desktop, which creates the default volumes inside a virtual machine.

But most of you wanted to know where the volumes were to edit the files.

Danger: Any operation inside the Docker data root is dangerous, and can break your Docker completely, or cause problems that you don't immediately recognize, so you should never edit files without mounting the volume into a container, except if you defined a *custom volume path* so you don't have to go into the Docker data root.

Even if you defined a custom path, we are still talking about a volume, which will be mounted into a container, in which the files can be accessed by a process which requires specific ownership and permissions. By editing the files from the host, you can accidentally change the permission or the owner making it inaccessible for the process in the container.

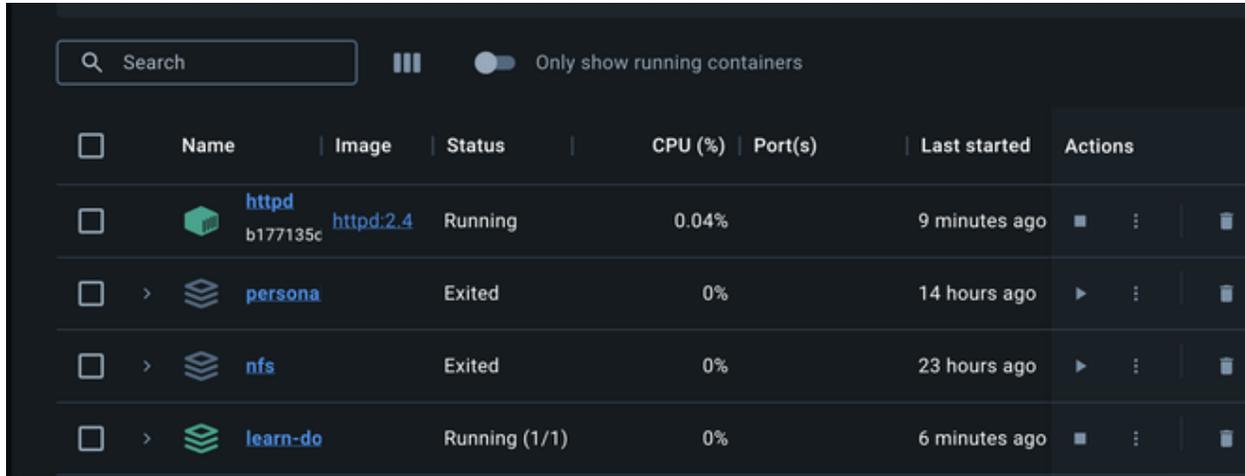
Even though I don't recommend it, I understand that sometimes we want to play with our environment to learn more about, but we still have to try to find a less risky way to do it.

You know where the volumes are, and you can edit the files with a text editor from command line or even from the graphical interface. One problem on Linux and macOS could be setting the proper permissions so you can edit the files even if you are not root. Discussing permissions could be another tutorial, but this is one reason why we have to try to separate the data managed by a process in a Docker container from the source code or any files that requires an interactive user. Just think of an application that is not running in a container, but the files still have to be owned by another user. An example could be a webserver, where the files has to be owned by a user or group so the webserver has access to the files, while you still should be able to upload files.

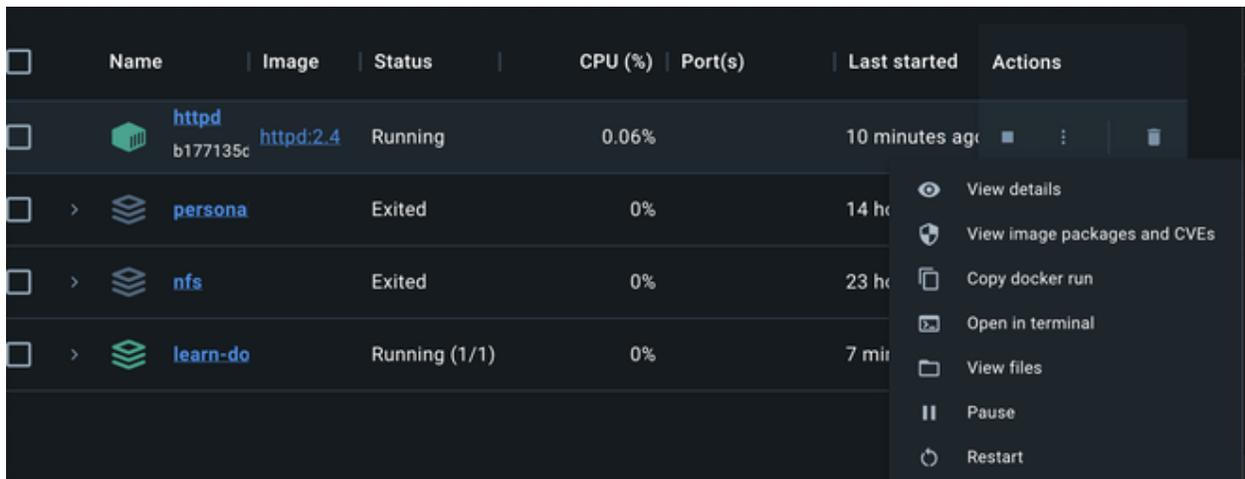
15.7.2 View and Edit files through Docker Desktop

Docker Desktop let's you browse files from the GUI, which is great for debugging, but I don't recommend it for editing files, even though Docker Desktop makes that possible too. Let's see why I am saying it.

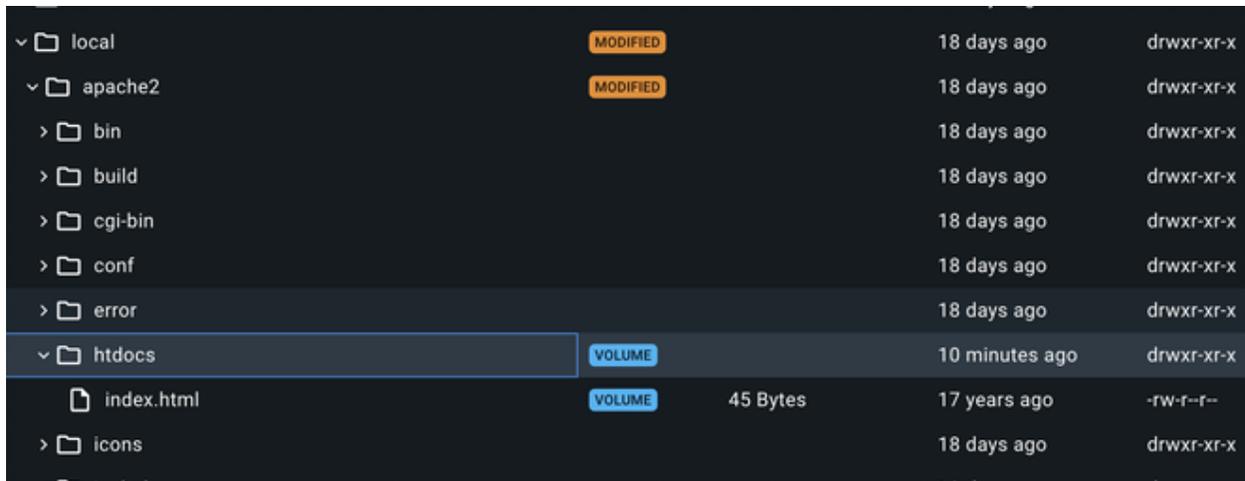
Open the Containers tab of Docker Desktop.



Click on the three dots in the line of the container in which you want to browse files



Go to a file that you want to edit



Item	Label	Modified	Permissions
local	MODIFIED	18 days ago	drwxr-xr-x
apache2	MODIFIED	18 days ago	drwxr-xr-x
bin		18 days ago	drwxr-xr-x
build		18 days ago	drwxr-xr-x
cgi-bin		18 days ago	drwxr-xr-x
conf		18 days ago	drwxr-xr-x
error		18 days ago	drwxr-xr-x
htdocs	VOLUME	10 minutes ago	drwxr-xr-x
index.html	VOLUME	45 Bytes, 17 years ago	-rw-r--r--
icons		18 days ago	drwxr-xr-x

Note: Notice that Docker Desktop shows you whether the files are modified on the container’s filesystem, or you see a file on a volume.

Right click on the file and select “Edit file”.

Before you do anything, run a test container:

```
docker run -d --name httpd -v httpd_docroot:/usr/local/apache2/htdocs httpd:2.4
```

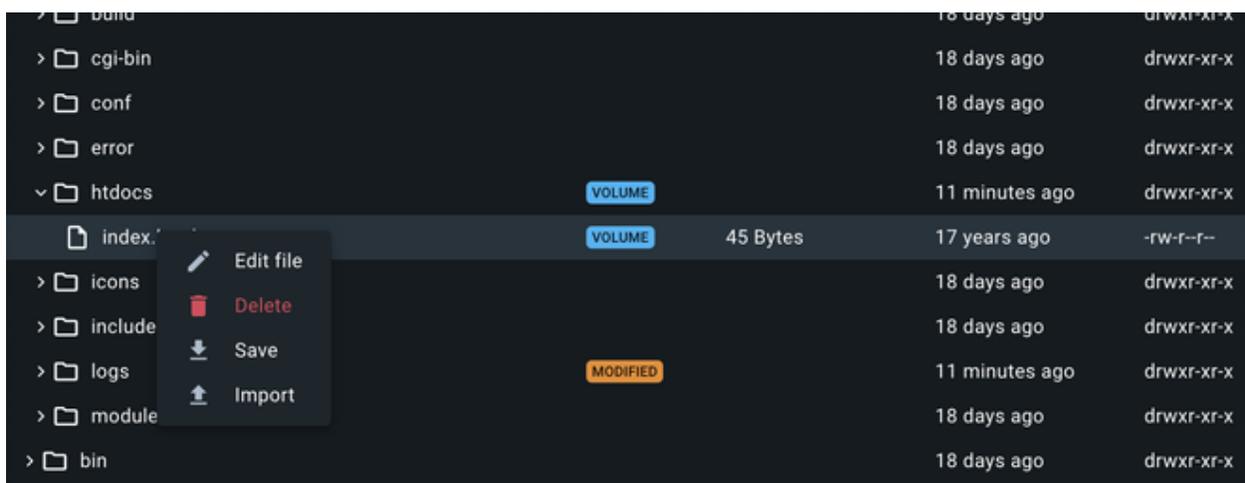
And check the permissions of the index file:

```
docker exec -it httpd ls -l /usr/local/apache2/htdocs/
```

You will see this:

```
-rw-r--r-- 1 504 staff 45 Jun 11 2007 index.html
```

You can then edit the file and click on the floppy icon on the right side or just press CTRL+S (Command+S on macOS) to save the modification.



Item	Label	Modified	Permissions
htdocs	VOLUME	11 minutes ago	drwxr-xr-x
index.html	VOLUME	45 Bytes, 17 years ago	-rw-r--r--
icons		18 days ago	drwxr-xr-x
include		18 days ago	drwxr-xr-x
logs		11 minutes ago	drwxr-xr-x
module	MODIFIED	18 days ago	drwxr-xr-x
bin		18 days ago	drwxr-xr-x

Then run the following command from a terminal:

```
docker exec -it httpd ls -l /usr/local/apache2/htdocs/
```

And you will see that the owner of the file was changed to root.

```
total 4
-rw-r--r-- 1 root root 69 Jan  7 12:21 index.html
```

One day it might work better, but I generally don't recommend editing files in containers from the Graphical interface.

Edit only source code that you mount into the container during development or [Use Compose watch](#) to update the files when you edit them, but let the data be handled only by the processes in the containers.

Some applications are not optimized for running in containers and there are different folders and files at the same place where the code is, so it is hard to work with volumes and mounts while you let the process in the container change a config file, which you also want to edit occasionally. In that case you need to learn how permissions are handled on Linux using the `chmod` and `chown` commands so you both have permission to access the files.

15.7.3 Container based dev environments

Docker Desktop Dev environment

One of the features of Docker Desktop is that you can run a development environment in a container. In this tutorial we will not discuss it in details, but it is good to know that it exists, and you can basically work inside a container into which you can mount volumes.

More information in the [documentation of the Dev environment](#)

Visual Studio Code remote development

The dev environment of Docker Desktop can be opened from Visual Studio Code as it supports opening projects in containers similarly to how it supports remote development through SSH connection or in Windows Subsystem for Linux. You can use it without Docker Desktop to simply open a shell in a container or even open a project in a container.

More information is in the [documentation of VSCode about containers](#).

Visual Studio Code dev containers

Microsoft also created container images for creating a dev container, which is similar to what Docker Desktop supports, but the process of creating a dev container is different.

More information in the [documentation of VSCode about dev containers](#).

15.8 Conclusion

There are multiple ways to browse the content of the Docker volumes, but it is not recommended to edit the files on the volumes. If you know enough about how containers work and what are the folders and files that you can edit without harming your system, you probably know enough not to edit the files that way in the first place.

For debugging reasons or to learn about Docker by changing things in the environment, you can still edit the files at your own risk.

Everything I described in this tutorial is true even if the user is not an interactive user, but an external user from the container's point of view, trying to manage files directly in the Docker data root.

So with that in mind if you ever think of doing something like that, stop for a moment, grab a paper and write the following sentence 20 times to the paper:

“I do not touch the Docker data root directly.”

If you enjoyed this tutorial, I also recommend reading about [Volume-only Compose projects](#).