Getting started with Docker

In this tutorial you'll learn:

- How to run **containers**.
- How to define and build **images**.
- How to create and use **volumes**.
- How to define and use **networks**.

Prerequisites:

- Having installed Docker on your computer. See the installation guide.
- Being familiar with the notions of containers, images, volumes and networks in Docker. See the Docker primer for an introduction.
- Being familiar with the basic notions of Linux.

Terminology
- You'll use the **terminal** to run Docker commands. Referring to the Docker architecture, the terminal is the **client** that communicates with the Docker daemon.
- Docker runs **containers** on your computer. We'll refer to your computer as the **host**, the containers being the **guests**.
- A **containerized application** is an application running in a container.

1 Running containers

The command used to run a container is `docker run` followed by four parameters:

```
docker run [options] image-name [command] [arg]
```

The four parameters are:

- *options*. List of options.
- *image-name*. The fully qualified name of the image used to run the container.
- *command*. The command to be executed in the container.
- *arg*. The arguments taken by the command executed in the container.

Only the parameter *image-name* is mandatory. The fuly qualified name of an image is specified as a sequence of four fields, formatted as follows:

```
registry_url/user/name:tag
```

where:

- *registry_url* (optional). The URL of the registry that provides the image. If its value is not specified, the image will be looked up for in the DockerHub registry.
- *user*. The identifier of the user or organization that created the image.
- *name* (mandatory). The name of the image.
- *tag* (optional). It specifies the version of the image. If its value is not specified, the tag *latest* is taken, pointing to the latest version of the image.

Exercise
For each of the following images,
specify the name of the registry, the user, the name and the tag.

1. registry.redhat.io/rhel8/mysql-80

2. alpine:3.11

::::: {.last-child}
3. alpine
:::::

Solution

1. Registry: *registry.redhat.io*, user: *rhel8*, name: *mysql-80*, tag: *latest*
2. Registry: DockerHub, user: not specified, name: *alpine*, tag: *3.11*
3. Registry: DockerHub, user: not specified, name: *alpine*, tag: *latest*

Exercise
What's the difference between the following image names?

1. alpine:latest

2. registry.hub.docker.com/library/alpine

::::: {.last-child}
3. alpine
:::::

Solution

There is no difference. They all point to the same image, that is the latest version of *alpine* in the DockerHub registry.

We now learn how to use the command `docker run` and some of its options. In the following exercises, we'll run containers from the image named *alpine* that is available on the DockerHub registry. This image provides a lightweight distribution (i.e., it doesn't contain many features) of Linux.

Exercise
You want to run the container from the latest version of
the image *alpine*.
Which command would you write in the terminal?

Solution

The goal of this exercise is to start playing with the `docker run` command. Since the question doesn't say anything about the options nor does it mention the command to run inside the container, we'd type:

```
docker run alpine
```

Exercise
Execute the command that you proposed in the previous exercise, observe the output in the terminal and explain the actions taken by Docker to run the container.

Solution

The output obtained from executing the command should look like the following:

```
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
aad63a933944: Pull complete
Digest: sha256:b276d875eeed9c7d3f1cfa7edb06b22ed22b14219a7d67c52c56612330348239
Status: Downloaded newer image for alpine:latest
```

Here is what happens under the hood:

1. Docker looks for an image named *alpine:latest* in the host computer and cannot find it.
2. Docker *pulls* (i.e., downloads) the image from the DockerHub registry.

For now, you can ignore the identifiers output by Docker, such as the digest.

Okay but **where's the result of running the container?**

First of all, let's see if the container is still running in the first place. In order to list all containers still running on the host, type the following command:

```
docker container ls
```

Your container shouldn't appear in the output of the command. This is because it's not running. In order to see all containers, including those that are not running, type the following command:

```
docker container ls -a
```

Exercise
What information is displayed for each container?

Solution

- The identifier of the container.
- The name of the image used to run the container (it should be *alpine* for your container).
- The command executed within the container (it should be `/bin/sh` for your container).
- When the container has been created.
- The container current status (it should be *exited (0) x seconds ago* for your container).
- The network ports used by the container (we'll study them later).
- The name of the container. If you don't specify any when you run the container (as is our case), Docker generates a random name by concatenating an adjective and a famous scientist's name (e.g., *agitated_newton*).

Exercise
By looking at the command executed within the container (``/bin/sh``), can you tell why the container stopped without giving any output?

Solution

The command is `/bin/sh`; the container runs a Linux terminal. But since we didn't specify what to do with that terminal (we didn't run any Linux command nor we tried to access the terminal), the container stopped.

We're now going to do something useful with the image *alpine*. But first, we start with some good practices that you should adopt while playing with images and containers.

1.1 Good practices

1. **Name your containers.** Although Docker assigns a default name to a new container, it's usually a good practice to give a container a name of your choice to make it easily distinguishable. You can do it by using the option `--name`. Try the following:

```
docker run --name my-alpine alpine
```

As before, the container stops immediately. If you list all your containers by typing again:

```
docker container ls -a
```

you should see a container named *my-alpine*.

2. **Remove a container when it stops.** Unless you want to reuse your container later, you can ask Docker to automatically remove it when it stops by using the option `--rm`. This will prevent unused containers from taking up too much disk space.

Try the following:

```
docker run --rm --name container-to-remove alpine
```

If you list all the containers you should see that there is no container named *container-to-remove*.

3. **Remove unused containers.** Stopped containers that have been run without using the option `--rm` are still stored in the host. If you want to remove a specific container (e.g., *my-alpine*), use the following command:

```
docker container rm my-alpine
```

If you want to remove all stopped containers, use the following command:

```
docker container prune
```

4. **Remove unused images.** Images can take up a lot of disk space. As a result, you should remember to remove those that you don't intend to use any longer. The commands to remove a specific image and prune unused ones are `docker image rm` and `docker image prune -a` respectively.

1.2 Execute a command within a container

Remember that the template of `docker run` is the following:

```
docker run [options] image-name [command] [arg]
```

The optional parameter *command* refers to a command to be executed within the container, possibly with some arguments (parameter *arg*). As we saw before, when we run a container from the image *alpine*, a Linux terminal `/bin/sh` is launched.

> **Notice**
> The Linux terminal `/bin/sh` is run within the container. Henceforth, we'll use the following terms:
> - **Host terminal.** The terminal that you use to interact with the operating system of your computer.
> - **Guest terminal.** The terminal that is run within the container.

By using the optional parameter *command*, we can run a command in the guest terminal.

Exercise
Run a container from the image *alpine* and execute the
Linux command `ls` that lists the content of the current directory.

::::: {.last-child}
* Where are the listed files stored?
In the host or in the container?
:::::

Solution

```
docker run --rm --name ls-test alpine ls
```

- The command `ls` is run in the guest terminal, therefore what we see in the output is a list of files stored in the container.

Notice
In Exercise @ref(exr:command-in-guest) the command `ls` is executed in the guest terminal, but its output is redirected to the host terminal.
In other words, when we run the container, we don't interact directly with the guest terminal; we just send a command and the output is redirected to the host terminal.

Now let's see how to execute a command in the guest terminal that also requires an argument.

Exercise
By using the Linux utility ``ping``, check
whether the Web site www.centralesupelec.fr is reachable.

Solution

```
docker run --rm --name ping-test alpine ping www.centralesupelec.fr
```

In order to interrupt `ping` just type the key combination that you's use to interrupt any other command in your terminal. (typically Ctrl-C on Windows and Cmd-C in MacOs).

1.3 Interacting with a container

An application running in a container might need to interact with the user. For instance, the Linux command `rev` reverses whatever the user types on the keyboard. In order to interact with a container, you should use the option `-it` of `docker run`.

xercisebox
Exercise
Run a container from the image *alpine* to execute the
Linux command `rev` and interact with it.

Solution

```
docker run --rm --name rev -it alpine rev
```

After typing the command, type a word on your keyboard (e.g., *deeps*), press *Return* and you should see the same word reversed (e.g., *speed*).

The option `-t` opens a guest terminal (so we can see its output); the option `-i` allows you to write directly into the guest terminal.

In order to stop using the guest terminal, you'll need to press Ctrl+D (both in Windows and MacOs).

Now run the following command:

```
docker run  --name my-alpine -it alpine
```

Note: we didn't use the option `--rm` (the container will not be removed when we stop it, we're going to use it again). Moreover, we didn't specify any command to run in the guest terminal.

> **Exercise**
> What do you obtain?

Solution

When we run a container from the image *alpine*, the command `/bin/sh` is executed within the container. Since we specified the option `-it`, what we obtain is an access to the Linux terminal running in the container.

1.4 Starting and stopping containers.

`docker run` is a shorthand for two Docker commands, namely `docker create`, that creates a container from an image, and `docker start`, that starts the container after its creation.

Suppose now that you want to download a Web page by using Linux Alpine. You can use the Linux command `wget` followed by the URL of the page that you want to download.

> **Exercise**
> By using the guest terminal
> in the container *my-alpine*,
> download
> [this Web page](https://www.centralesupelec.fr/fr/presentation){target="_blank"}.
>
> ::::: {.last-child}
> * Where will the Web page be saved? The host computer or the container?
> :::::

Solution

Just type in *my-alpine* guest terminal the following command:

```
wget https://www.centralesupelec.fr/fr/presentation
```

- The Web page will be saved in the current directory of the container. You can verify that the file is there by typing `ls` in the guest terminal.

In *my-alpine* guest terminal type `exit`. This closes the guest terminal and, as a result, stops the container.

> **NOTICE**
> Stopping the container will not erase any of the files stored in the container. Removing the container will.

If you want to start the container *my-alpine* again, you can use the following command:

```
docker container start -ai my-alpine
```

This will open the guest terminal of the container again; type `ls` to verify that the Web page that you downloaded before is still there.

1.5 A simple use case

Suppose that you need to download all the figures of this Web page. The Linux utility `wget` comes in handy. However, you don't have Linux and you'd like to avoid the hassle of installing it on your computer, or in a virtual machine, just for this task.

A great alternative is to run Linux in a Docker container. Unfortunately, the Alpine distribution that we've been playing with doesn't provide an implementation of `wget` with all the options that we need.

We turn to another Linux distribution, **Ubuntu**, for which DockerHub has several images.

> **Exercise**
> Run a container with Ubuntu 19.10 and open a guest terminal.
> Call the container *dl-figures*, and avoid the option
> ``--rm``, we'll use this container later.

Solution

```
docker run --name dl-figures -it ubuntu:19.10
```

If you look at the DockerHub registry Web page describing Ubuntu, you'll see that the version 19.10 has many tags, including *19.10*, *eoan-20200313*, *eoan*, *rolling*. You can use any of these tags to download the image. Another way to write the previous command is:

```
docker run --name dl-figures -it ubuntu:eoan
```

From now on, we'll be interacting with the guest Ubuntu terminal. If you type the command `wget`, you'll get an error (`bash: wget: command not found`).

> **Notice**
> The image *Ubuntu* doesn't include all the commands that you'd find in a full-blown Ubuntu distribution; the reason is to keep the size of the image small, a necessary constraint given that images are transferred over the Internet.

Luckily, there's a way to install `wget` in our Ubuntu distribution. Ubuntu provides a powerful command-line package manager called **Advanced Package Tool** (APT). First, you need to run the following command:

```
apt-get update
```

which fetches the available packages from a list of sources available in file `/etc/apt/sources.list`.

Then, you can install `wget` by running the following command:

```
apt-get install -y wget
```

In order to obtain all the figures from a Web page, type the following command:

```
wget -nd -H -p -P /my-figures -A jpg,jpeg,png,gif -e robots=off -w 0.5 https://www.centralesupelec.fr/fi
```

You should see in the current directory a new folder named *my-figures* containing the downloaded figures; verify it by typing `ls my-figures`.

Before terminating, don't forget to read your fortune cookie. In the shell, run the following command:

```
apt-get install -y fortune
```

and then:

```
/usr/games/fortune -s
```

When you're done, you can simply type the command `exit` to quit the guest terminal and stop the container.

> **Notice**
> You might wonder how you can transfer the downloaded figures from the container to the host computer. We'll see that later when we introduce the notion of **volumes**.

2 Creating Images

A Docker image can be thought of as a template to create and run a container. An image is a file that contains a **layered filesystem** with each layer being **immutable**; this means that the files that belong to a layer cannot be modified or deleted, nor can files be added to a layer.

When a container is created from an image, it will be composed of all the image read-only layers and, on top of them, a writable layer (termed the **container layer**), where all the new files created in the container will be written. For example, the figures that we downloaded in the container *dl-figures* were stored in the writable layer of that container.

2.1 Interactive image creation

When we run the container *dl-figures* in Section @ref(simple-use-case), we modified the container to install the command `wget`. You can see the modifications by typing the following command:

```
docker diff dl-figures
```

The output consists of a list of files tagged with the letter A, C or D, indicating respectively that the file has been added (A), changed (C) or deleted (D). In this list you'll find the downloaded figures, as well as other files that have been added or modified or deleted when we installed `wget`.

> **Exercise**
> `If layers, except the top one, are immutable,`
> `how can files that belong to the lower layers be modified or deleted?`

Solution

All files marked with A are new and therefore are added to the writable layer of the container.

As for the existing files, they live in the immutable layers of the image, and therefore cannot be touched directly. Instead, they are copied from the bottom layers to the writable layer where they are modified. This strategy is called **copy-on-write**.

The structure of layers generates a **layered filesystem** in the image; if different copies of the same file exist in different layers, the copy in the uppermost layer overwrites the others.

We can create a new image from the container *dl-figures*, one that provides a Ubuntu distribution with the command `wget` already installed, with the following command:

```
docker commit dl-figures ubuntu-with-wget
```

The command creates a new image called *ubuntu-with-wget*.

Exercise
Run a container from the image *ubuntu-with-wget* and verify that the command *wget* is actually installed.

Solution

Just type the following command:

```
docker run --rm -it ubuntu-with-wget
```

In the guest terminal type `wget`: you should see the following output:

```
wget: missing URL
Usage: wget [OPTION]... [URL]...

Try `wget --help` for more options.
```

2.2 Dockerfiles

The most common way to create an image is to use a **Dockerfile**, a text file that contains all the instructions necessary to build the image. The advantage of the Dockerfile is that it can be interpreted by the Docker engine, which makes the creation of images an automated and repeatable task, rather than manual like the interactive method.

Inspired by the previous example, suppose that we want to create a containerized application to download figures from a Web page. As a template for this application, we need to build a new image, that we'll call *fig-downloader*.

The Dockerfile containing the instructions to build the image *fig-downloader* is as follows:

```
FROM ubuntu:eoan
RUN apt-get update
RUN apt-get install -y wget
RUN mkdir -p /my-figures
WORKDIR /my-figures
ENTRYPOINT ["wget", "-nd", "-r", "-A", "jpg,jpeg,bmp,png,gif"]
CMD ["https://www.centralesupelec.fr/fr/presentation"]
```

Here's the explanation:

1. We use the image *ubuntu:eoan* as the **base image**. This corresponds to the instruction `FROM ubuntu:eoan`.
2. We install the utility `wget` in the base image. This corresponds to the instructions `RUN apt-get update` and `RUN apt-get install -y wget`.
3. We create a directory *my-figures* under the root directory of the image. This corresponds to the instruction `RUN mkdir -p /my-figures`.
4. We set the newly created directory */my-figures* as the **working directory** of the image. This corresponds to the instruction `WORKDIR /my-figures`.
5. We specify the command to be executed when a container is run from this image. This corresponds to the instruction `ENTRYPOINT ["wget", "-nd", "-r", "-A", "jpg,jpeg,bmp,png,gif"]`. This instruction means: execute `wget` with the options `-nd`, `-r`, `-A`; the last option takes a list of file extensions (`jpg,jpeg,bmp,png,gif`) as its argument.
6. Remember that the utility `wget` takes the URL of the Web page as an argument. The URL will be specified when we run the container from the image *fig-downloader*. Optionally, we can

specify a default argument by using the keyword CMD. The meaning of the instruction `CMD ["https://www.centralesupelec.fr/fr/presentation"]` is: if we don't give any URL when we run the container, the figures will be downloaded from https://www.centralesupelec.fr/fr/presentation.

Exercise
What is the relation between the Dockerfile lines and the image layers?

Solution

Each line corresponds to a new layer. The first line corresponds to the bottom layer; the last line to the top layer.

Exercise
Could you identify a problem in this Dockerfile?
Modify the Dockerfile accordingly.

Solution

When creating an image, we should keep the number of layers relatively small; in fact, the more the layers, the longer the image will take to build. Here we create three separate layers with two RUN commands; we can just merge the two layers. The resulting Dockerfile will be:

```
FROM ubuntu:eoan
RUN apt-get update &&
    apt-get install -y wget &&
    mkdir -p /my-figures
WORKDIR /my-figures
ENTRYPOINT ["wget", "-nd", "-r", "-A", "jpg,jpeg,bmp,png,gif"]
CMD ["https://www.centralesupelec.fr/fr/presentation"]
```

2.3 Building an image

We're now going to build an image from a Dockerfile.

1. Create a directory named *fig-downloader* in your computer with a file named *Dockerfile* inside.
2. In the *Dockerfile* write the set of instructions that you proposed in Exercise @ref(exr:dockerfile-creation).
3. In the terminal, set the working directory to *fig-downloader*.
4. Build an image called *fig-downloader* by executing the following command:

```
docker build -t fig-downloader .
```

The `.` at the end of the command means that the Docker engine will look for a file named *Dockerfile* in the working directory.

Exercise
Once the image is built, type the command ``docker image ls -a``.
What are the images with repository and tag ``<none>``?
Why are there three of such images?

Solution

These are the **intermediate images**. Once a layer is compiled, an intermediate image is created that contains that layer and all the layers underneath. In other words, the intermediate image corresponding to the layer i contains all files up to the layer i, including layers 1 through $i-1$.

The intermediate layers are used by the **build cache**, of which we'll see an example later.

Although there are five layers in the new image, there are only three intermediate images because:

- the base image is *ubuntu:eoan*.
- the image corresponding to the top layer is the final image *fig-downloader*.

Good to know
If you give the Dockerfile a different name (say, *Dockerfile-fig-downloader*), the command to build the image will be:
`docker build -t fig-downloader -f Dockerfile-fig-downloader .`
The option `-f` is used to specify the name of the Dockerfile.

Let's dive deeper into the anatomy of an image.

Exercise
Run the following command:

```
docker history fig-downloader
```

and analyze the layers of the new image.

* Why do some layers have an ID, while other
are marked as missing?

::::: {.last-child}
* Can you find the identifiers of the intermediate images?
:::::

Solution

The layers that have an ID correspond to the layers of the new image, including the top layer and the base image. The layers marked as *missing* are those that compose the base image. Those layers are not stored in your computer, simply because they belong to an image that hasn't been build on your computer but you downloaded from the DockerHub registry.

By looking at the output of `docker image ls -a` and the output of this command, we see that the layers between the base image and the top layer have the same identifiers as the intermediate images.

xercisebox
Exercise
Run the following command:

``
docker run --name dl-1 fig-downloader
``

What does it do? Where are the downloaded pictures?

Solution

We downloaded the figures of the page https://www.centralesupelec.fr/fr/presentation. The downloaded pictures are in the folder */my-figures* of the container *dl-1*. For now, don't worry about accessing them.

Exercise
Run the following command:

```
docker run --name dl-2 fig-downloader https://www.centralesupelec.fr/fr/nos-c
```

What does it do? Where are the downloaded pictures?

Solution

We downloaded the figures of the page https://www.centralesupelec.fr/fr/nos-campus. We basically overwrote the URL specified by the CMD keyword with a new one. The downloader pictures are in the folder */my-figures* of the container *dl-2*. For now, don't worry about accessing these figures.

2.4 Containerized Python application

Download this archive file and unzip it into your working directory. In this archive you'll find:

- A Dockerfile.
- A Python script *main.py* that asks the user to enter the URL and the language of a Web page, and prints the 10 most frequent words occurring in that page.
- A file *requirements.txt* with the list of the Python packages needed to run the given script.

The content of the Dockerfile is as follows:

```
FROM python:3.7-slim
RUN mkdir -p /app
WORKDIR /app
COPY ./main.py ./requirements.txt /app/
RUN pip install -r requirements.txt
ENTRYPOINT ["python", "main.py"]
```

Exercise
Describe what this Dockerfile does.

Solution

- Takes *python:3.7-slim* as the base image.
- Creates a new folder *app* in the image under the root directory.
- Changes the working directory to */app*.

- Copies the files *main.py* and *requirements.txt* from the local computer to the directory */app* in the image.
- Runs the command `pip install` to install the Python libraries specified in the file *requirements.txt*.
- Execute the command `python main.py`.

Exercise

`Build an image called *wordfreq* from this Dockerfile.`

Solution

```
docker build -t wordfreq .
```

> **Exercise**
> Without changing the Dockerfile, rebuild the same image.
> What do you notice?

Solution

The build is very fast. Since we didn't change the Dockerfile, the image is rebuilt by using the image layers created previously. This is clearly indicated by the phrase **using cache** written at each layer. Using the already stored layers is called **build cache**.

> **Exercise**
> What happens if you modify a line in the Python script and
> you rebuild the image?
> Is the build cache still used?

Solution

Add any instruction at the end of *main.py*, such as:

```
print("I added this line")
```

then rebuild the image. The three bottom layers are not affected by the modification, therefore they benefit from the build cache. Layer 4 is the first layer affected by the modification. This layer, and all the upper layers, need therefore to be rebuilt.

> **Exercise**
> Considering how the build cache is used in Docker,
> can you tell what's wrong with this Dockerfile?
> Modify the Dockerfile accordingly.

Solution

Each time we modify *main.py* and we rebuild the image, the layer 4 and 5 are recreated, meaning that all the Python packages are downloaded and installed. Depending on the size and number of the packages, this might take some while. A better way to structure the Dockerfile is to install the packages before copying the Python script to the image. Here is how we should modify the Dockerfile:

```
FROM python:3.7-slim
RUN mkdir -p /app
WORKDIR /app
COPY ./requirements.txt /app/
RUN pip install -r requirements.txt
```

```
COPY ./main.py /app/
ENTRYPOINT ["python", "main.py"]
```

Exercise
Modify *main.py* and rebuild the image.
What changed?

Solution The Python packages are not reinstalled, as a result rebuilding the image takes much less time.

3 Data Volumes

In Exercise @ref(exr:dl-1-container) you'be been asked to run a container named *dl-1* to download some figures from a Web page. The figures were downloaded into the directory */my-figures* of the container. But we left a question unanswered.

How do we transfer those figures from the container to the host computer?

One way to go about that is to run the following command in the host terminal:

```
docker cp dl-1:/my-figures .
```

This will copy the directory */my-figures* from the container *dl-1* to the host computer working directory. You can verify it by yourself.

Exercise
Can you tell why this solution is less than ideal?

Solution

1. After running the container we need to do an additional action to copy the figures from the container to the host computer.
2. The container is created and run only to download some figures. We'd like to remove it automatically (with the option `--rm`) when its execution is over. However, if we do so, the pictures will be lost before we can copy them to the host computer.

3.1 Using a host volume

A better solution is to **mount** (i.e., attach) a directory of the host computer at the container's directory */my-figures* when we run it. Let's see how it works.

Step 1. Create a directory named *figs-volume* in your working directory.

Step 2. Type and execute the following command:

```
docker run --rm -v $(pwd)/figs-volume:/my-figures fig-downloader
```

This command runs a container from the image *fig-downloader*.

- With the option `-v` we specify that we want to mount the directory *$(pwd)/figs-volume* (*$(pwd)* indicates the host working directory) at the directory *figs-volume* in the container;
- The option `--rm` indicates that we want the container to be removed when its execution is over.

Step 3. Verify that the pictures are in the folder *figs-volume*.

In this example, we've used the directory *figs-volume* as a **volume** (essentially, an external storage area) of the container; when the container is destroyed, the volume remains with all its data.

3.2 Docker volumes

In the example that we've just described, we've used a host directory as a volume. This is useful when we, or an application running on the host, need to access the files produced by a container. In all the other cases, a container should use a **Docker volume**, which is managed directly by the Docker engine.

Let's create a new Docker volume called *data-volume*:

```
docker volume create data-volume
```

Good to know (advanced notion)
Where the data will be actually stored?
You can inspect the new volume by typing the following command (or, you can click on the volume in the GUI):
`docker volume inspect data-volume`
A *mount path* is indicated; that's the folder where the data will be actually stored. If your computer runs Linux, that folder will be available on the host; if your computer runs Windows or MacOS, you'll not find that folder on your computer. Instead, it will be available in the hidden virtual machine that Docker for Windows and Docker for Mac use.
Do you want to see the directory?
One way to look into the hidden VM is to run the following containerized application:
`docker run -it --rm --privileged --pid=host justincormack/nsenter1`
This application will open a guest terminal into the VM. You can then use the commands `cd` and `ls` to browse to the directory indicated as the mount path of the new volume.

3.2.1 Sharing data

A Docker volume can be used to share data between containers.

Exercise

```
Run a container from the image ``ubuntu:eoan``,
specifying the options to:

* Remove the container once its execution is over.

* Interact with the terminal in the container.

::::: {.last-child}
* Mount the volume *data-volume* at the container's directory */data*.
:::::
```

Solution

```
docker run --rm -it -v data-volume:/data ubuntu:eoan
```

Exercise

```
Type a command in the guest Linux terminal to create a file
*test-file.txt* in the directory */data*.
Verify that the file is created.
```

Solution

The following command:

```
echo "This is a new file" > /data/test-file.txt
```

Creates a file *test-file.txt* with the line of text "This is a new file".

In order to verify that the file is created, we can type the following command:

```
ls /data/test-file.txt
```

To see the content of the file, we can type:

```
cat /data/test-file.txt
```

Exercise
Run a container from the image *alpine:latest*,
specifying the options to:

* Remove the container once its execution is over.

* Interact with the terminal in the container.

::::: {.last-child}
* Mount the volume *data-volume* to the directory */my-data*
of the container.
:::::

Solution

```
docker container run --rm -it -v data-volume:/my-data alpine
```

Exercise
Verify that you can read the file *test-file.txt*.
Which folder would you look in?

Solution

We need to look in the folder */my-data* because this is where we mounted *data-volume*.

```
cat /my-data/test-file.txt
```

In the guest terminals of both containers type `exit`. This will terminate and destroy (since we used the option `--rm`) the containers.

Exercise
Will the file *test-file.txt* be removed?
Why?

Solution

No. The file that we created before has been saved in the volume *data-volume*. Volumes are a way to persist data beyond the life span of a container.

4 Single-Host Networking

In order to let containers communicate and, therefore, co-operate, Docker defines a simple networking model known as the **container network model**

Exercise
Describe the output of the following command:

```
docker network ls
```

Solution The command lists all the networks created by Docker on your computer. For each network, the values of four attributes are shown:

- The identifier.
- The name.
- The driver used by the network.
- The scope of the network (local or global). A local scope means that the network connects containers running on the same host, as opposed to a global scope that allow the communication between containers on different hosts.

Depending on the containers that you used in the past, you might see different networks. However, three networks are worth noting:

- The network named **bridge**, that uses the driver **bridge** and a local scope. By default, any new container is attached to this network.
- The network named **host**, that uses the driver **host** and a local scope. It is used when we want a container to use the network interface of the host directly. It is important to remember that this network should only be used when analyzing the host's network traffic. In the other cases, using this network exposes the container to all sorts of security risks.
- The network named **none**, that uses the driver **null** and a local scope. Attaching a container to this network means that the container cannot communicate with any other container or the outside world.

Exercise
The following command:

```
docker network inspect bridge
```

outputs the configuration of the network **bridge**.
By looking at this configuration, can you tell
what IP addresses will be given to the containers attached to this
network? What's the IP address of the router of this network?

Solution

The information is specified in the field named **IPAM**, more specifically:

- **Subnet** indicates the range of IP addresses used by the network. The value of this field should be 172.17.0.0/16; the addresses range from 172.17.0.1 to 172.17.255.255.
- **Gateway** indicates the IP address of the router of the network. The value should be 172.17.0.1

4.1 Creating networks

By default, any new container is attached to the network named *bridge*. As a result, all new containers will be able to communicate over this network. This is not a good idea. If a hacker can compromise any of these containers, s/he might be able to attack the other containers as well. As a rule of thumb, we should attach two containers to the same network **only** on a need-to-communicate basis.

```
What if a container doesn't need to use the network at all?
Try to run a container disconnected from any network and
verify that you cannot ping the URL
[www.google.com](http://www.google.com){target="_blank"}.
```

Solution

We should attach the container to the network **none**. As an example, we run the following command:

```
docker run --rm -it --network none alpine /bin/sh
```

Then we try to ping www.google.com as follows:

```
ping www.google.com
```

We should obtain the following message:

```
bad address 'www.google.com'
```

Type the command `exit` to quit the container.

Instead, if we run Linux Alpine without specifying the network (meaning that the container will be attached to the network *bridge*):

```
docker run --rm -it  alpine /bin/sh
```

and we try to ping www.google.com, we should get an answer. In some cases, the command ping would just hang and show no output; this is usually fixed by restarting Docker.

In order to create a new network, you can use the following command:

```
docker network create network_name
```

```
Exercise
Create two networks named *buckingham* and *rochefort* that
use the driver *bridge*.
By using the ``docker network inspect`` command,
look at the IP addresses of the new networks and write them down.
```

Solution

Just run the following commands:

```
docker network create buckingham
docker network create rochefort
```

The IP addresses for the network *buckingham* are 172.18.0.0/16 (addresses from 172.18.0.1 to 172.18.255.255); The IP addresses for the network *rochefort* are: 172.19.0.0/16 (assuming that you create *buckingham* before *rochefort*).

Exercise

```
Create three containers *athos*, *porthos* and *aramis* and attach them
to the two networks *buckingham* and *rochefort* as displayed
[in this figure](/courses/cloud-computing/docker-primer#fig:cnm){target="_blank"}
**The three containers will open a Linux Alpine shell**.
You'll need to launch the commands in three separate tabs of your terminal window

* What will the IP addresses of the three containers be in the two networks?
Remember that *porthos* is attached to two networks, therefore it'll have two
network interfaces (endpoints) and, as a result, two IP addresses.

:::: {.last-child}
* Verify your answers by inspecting the two networks (use the
command ``docker network inspect``).
:::::
```

Solution

Here are the commands to create *athos* and *aramis* that connect to just one network.

```
docker run --rm -it --name athos --network buckingham  alpine /bin/sh
docker run --rm -it --name aramis --network rochefort   alpine /bin/sh
```

Since we cannot attach a container to two networks in one go, we need two separate commands. The first command attaches the container to either network (here we choose *buckingham*):

```
docker run --rm -it --name porthos --network buckingham   alpine /bin/sh
```

The second command attaches the container to the second network:

```
docker network connect rochefort porthos
```

As for the IP addresses, each network has IP addresses in the range 172.x.0.0/16, where x is 18 in the network *buckingham* and 19 in the network *rochefort*. The address 172.x.0.1 is reserved for the router. Therefore, the containers will be assigned IP addresses from 172.x.0.2. In this solution, we created *athos*, *aramis* and *portos* in this order. Therefore, the IP addresses will be:

- In network *buckingham*:
  - *athos*: 172.18.0.2
  - *porthos*: 172.18.0.3
- In network *rochefort*:
  - *aramis*: 172.19.0.2
  - *porthos*: 172.19.0.3

You can actually verify this configuration by inspecting the two networks with the following commands:

```
docker network inspect buckingham
docker network inspect rochefort
```

4.2 Communication between containers

Let's see if and when the three containers can communicate.

Exercise

```
Which containers are able to communicate?
Justify your answer.
```

Solution

The only containers that cannot communicate are *athos* and *aramis*, because they're not connected to the same network.

Exercise
Try to ping *porthos* from *athos* by using its IP address.

::::: {.last-child}
* Which IP address of *porthos* would you use?
:::::

Solution

We need to use the IP address assigned to the endpoint linking *porthos* to the network *buckingham*, to which *athos* is connected. In our case, this is 172.18.0.3.

Exercise
Try to ping *porthos* from *athos* by using its name.
Do you succeed? Are you surprised?

Solution We succeed. Indeed, the network *buckingham* provides a DNS server, that can translate names into IP addresses.

4.3 A containerized chat room

We developed a simple chat room in Python that you can download here.

Participants use a *client* program to connect to the chat room; the chat room is managed by a *server* application that receives the client connections and forwards the messages between the users. The archive contains the following files:

- *client.py*. Implementation of the chat room client-side.
- *server.py*. Implementation of the chat room server-side.
- *utils.py*. Library with utility functions used in both *client.py* and *server.py*.

Exercise
By using Dockerfiles, create two images ``chat-client`` and ``chat-server``
that will be used to run the client and the server in Docker.

Solution

The Dockerfile for the client (let's call it *Dockerfile_client*) is as follows.

```
FROM python:3.7-slim
RUN mkdir -p /app
WORKDIR /app
COPY ./client.py ./utils.py /app/
ENTRYPOINT ["python", "client.py"]
```

We build the image with the following command:

```
docker build -t chat-client -f Dockerfile-client .
```

The Dockerfile for the server (let's call it *Dockerfile_server*) is as follows.

```
FROM python:3.7-slim
RUN mkdir -p /app
WORKDIR /app
COPY ./server.py ./utils.py /app/
ENTRYPOINT ["python", "server.py"]
```

We build the image with the following command:

```
docker build -t chat-server -f Dockerfile-server .
```

> **Good to know**
> The first three layers in both images are identical. Therefore, when building the second image *chat-server* the Docker engine reuses the cached layers created for the first image. This is indicated in the output of the `docker build` command with the phrase *using cache*.

We'll now run both containers. Since they need to communicate, they need to be attached to the same network (e.g., *buckingham*).

> **Exercise**
> Run a container from the image *server-chat*.
> Set the options to:
>
> * Automatically remove the container once its execution is over.
>
> * Give a name to the container (e.g., *server-chat*).
>
> * The server will print messages on the screen.
> In order to see them, you must use the option `-t`.
>
> Also, keep in mind that *server.py* takes an argument
> that is the **port number** where the server will listen to incoming connections.
>
>
> **What is the IP address of the server?**

Solution

We execute the following command:

```
docker container run --rm -t --name chat-server --network buckingham chat-server 64903
```

In my case, the IP address is 172.18.0.2 and port number is 64903

Exercise

```
Run a container from the image *client-chat*.
Set the options to:

* Automatically remove the container once its execution is over.

* Give a name to the container (e.g., *client-chat*).

* Since you'll use the the client to write messages in the chat room,
remember to set the option ``-it``.

The client takes two arguments: the host where the server is running
and the port which the server is listening to.
```

Solution

```
docker run --rm -it --name chat-client --network buckingham chat-client 172.18.0.2 64903
```

Instead of the server host IP address, we can use the server container name (the network *buckingham* has a DNS server).

As a result, we can run the client as follows:

```
docker run --rm -it --name chat-client --network buckingham chat-client chat-server 64903
```

Once the client is started, you'll be prompted to enter your name. Then you can start writing messages.

Notice

- You can type *#quit* at any moment to exit the chat room (client-side).
- Type Ctrl-C to stop the server.

Now, suppose that one of your classmates wants to join the chat room, but s/he's on another computer.

Exercise

```
Do you think your classmate can connect to the containerized server running in
your machine? Justify your answer.
```

Solution

No, s/he can't. Indeed, two containers can communicate only if they're connected to the same network.

What we need to do here is to expose our server to the outside world. The server runs in a container c that, in turns, runs on the host machine h. The server listens to port p_c that is opened **inside the container**. We need to map port p_c to a port p_h in the host computer. This way, the classmate client will connect to the server by specifying the **IP address of the host** h (not c) and p_h as the port number.

We implement this solution step by step.

1. Stop both the server and the client running on your machine.
2. Declare the port number to expose.

Exercise
How would you add this declaration?
Modify the Dockerfile accordingly?

Solution In the Dockerfile of the server, use the keyword EXPOSE. The Dockerfile will look like as follows:

```
FROM python:3.7-slim
RUN mkdir -p /app
WORKDIR /app
EXPOSE 64903
COPY ./server.py ./utils.py /app/
ENTRYPOINT ["python", "server.py"]
```

Note that it's better to put the declaration before copying the source files to the directory */app*, so the instruction EXPOSE will not be re-executed every time we rebuild the image after changing the source code *server.py*.

3. Rebuild the image *server-chat*.
4. Run the server container so that that the exposed port is mapped to a random port in the host computer.

Exercise
Which option of the command ``docker container run`` would you use?
Write the command to run the container and execute it.

Solution

We need to use the option `-P`. The command to run the container is:

```
docker run --rm -it --name chat-server  --network buckingham -P  chat-server 64903
```

5. Verify that the exposed port p_c is correctly mapped to a port p_h in the host computer by running the following command.

```
docker container port chat-server
```

Exercise
How do you read the output of this command?

Solution The output looks like as follows:

```
64903/tcp -> 0.0.0.0:32771
```

This means that the port 64903 (p_c) in the container is mapped to the port 32771 (p_h) of the host computer. When a remote client wants to connect to the server, it'll use port 32771. The IP address 0.0.0.0 means that a client can connect to the host computer by using any of its IP addresses.

6. Run a client from your host computer (same as before).
7. Ask any of your classmates to connect to your server. For this, you'll need to tell your classmate the IP address of your machine and the port number p_h.

Solution Assuming that the IP address of your machine is 192.168.1.8, the command will be the following:

```
docker run --rm -it --name chat-client chat-client 192.168.1.8 32771
```