



## UT 011. DOCKER

Computer Systems  
CFGs DAW

Álvaro Maceda  
[a.macedaarranz@edu.gva.es](mailto:a.macedaarranz@edu.gva.es)

2022/2023

Version:230309.1335

## License



**Attribution - NonCommercial - ShareAlike (by-nc-sa):** No commercial use of the original work or any derivative works is permitted, distribution of which must be under a license equal to that governing the original work.

## Nomenclature

Throughout this unit different symbols will be used to distinguish important elements within the content. These symbols are:



Important



Attention



Interesting

## TABLE OF CONTENTS

<b>5. Working with images.....</b>	<b>4</b>
5.1 What is a Docker Image.....	4
5.2 Docker Hub.....	6
5.3 Creating images.....	8
<b>6. Networking.....</b>	<b>11</b>
6.1 Default bridge.....	11
6.2 Custom bridge networks.....	13
<b>7. Supplementary material.....</b>	<b>14</b>

## UT 011. DOCKER

### 5. WORKING WITH IMAGES

#### 5.1 What is a Docker Image

The Docker image is like the “hard disk” from which containers are created. It’s made of layers, each layer representing the changes in the file system from the previous layers. **We call an image each one of those layers**, so an image is “composed” of another images.

For example, when we download an image:

```
$ docker pull wernight/funbox  
  
Using default tag: latest  
latest: Pulling from wernight/funbox  
f2b6b4884fc8: Pull complete  
24876304c826: Pull complete  
dc2853569c8e: Pull complete  
f1feacc76ece: Pull complete  
47b0568134ef: Pull complete
```

You can see that multiple layers are downloaded independently. That’s because, when creating images, each time you modify something a new layer is added. For example, if we want an image with Apache, Git and some source code, the process would be more or less like this:



How images and layers are created

The blue boxes are the new information that we add on top of the previous layer. In the first step, we install Apache over the contents of Image 1.0: those changes will form the layer “Image 1.1”. After that, we create the layer “Image 1.2” by installing git, and so on.

When a container is running, the last layer is the only writable one. That layer is destroyed when we destroy the container.

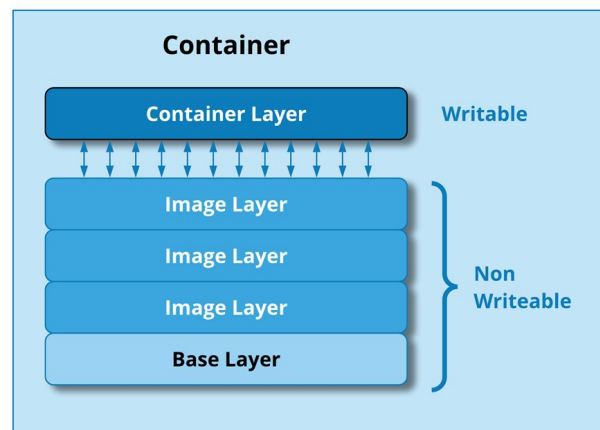


Image layers in a Docker container

### 5.1.1 Image labels and tags

The images are identifier with an ID (the hexadecimal number that you can see on the left of the docker pull example) We can always reference an image by its ID, but it's more convenient to reference an image by its label. Usually, only the top image is labeled.

A label is a string composed of a **name** and a **tag**. For example, if you list the images that you already have in your machine with `docker image ls`:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
alpine	latest	b2aa39c304c2	2 weeks ago	7.05MB
nginx	latest	9eee96112def	3 weeks ago	142MB
ubuntu	latest	58db3edaf2be	4 weeks ago	77.8MB
ubuntu	20.04	e40cf56b4be3	4 weeks ago	72.8MB
hello-world	latest	feb5d9fea6a5	17 months ago	13.3kB
wernight/funbox	latest	538c146646c3	4 years ago	1.12GB

The column REPOSITORY is the name, and the column TAG is the version. There is a special tag, `latest`, that is used to identify the last version available of an image. In the above example, you can see that we have two different images for ubuntu: the image labeled with version 20.04 and the latest version available.

A label is only an alias for the image ID, so you can have different labels pointing to the same image ID.

Not all the images are labeled. Intermediate images are often not labeled. You can see them with `docker image ls --all`:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
alpine	latest	b2aa39c304c2	2 weeks ago	7.05MB
ubuntu	20.04	e40cf56b4be3	4 weeks ago	72.8MB
ubuntu	latest	58db3edaf2be	4 weeks ago	77.8MB
<none>	<none>	86105d084ebb	3 months ago	84.2MB
<none>	<none>	9419c58a6e1e	3 months ago	84.2MB
<none>	<none>	116e02c4b147	3 months ago	76.1MB
<none>	<none>	61968c3a4100	3 months ago	73.9MB
mariadb	latest	6e0162b44a5f	11 months ago	414MB
hello-world	latest	feb5d9fea6a5	17 months ago	13.3kB
wernight/funbox	latest	538c146646c3	4 years ago	1.12GB

### 5.1.2 Removing images

When we don't need an image anymore, we can remove it with `docker image rm <tag or ID>`

Often, we want to remove the images that there are not needed to free space in our hard drive. To do that, we need to know the difference between unused and dangling images:

- **Unused images:** Images that are not used by any container. Take in account that only the last image (the last layer) is the one used by a container, but the images used by that one are considered as used, too.
- **Dangling images:** Images that has no label and are not needed by another image.

Dangling images are removed with `docker image prune`. Unused and dangling images are removed with `docker image prune --all`.

It's usually safe to remove images because you can always download or generate them again. We will see how to do that in a moment.

## 5.2 Docker Hub

The easiest way of getting an image to create a container is to use a Hub. A Hub is a place on the Internet with a directory of images. The most used public hub is Docker Hub (you don't need to register to use the site):

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

You can search for images there by a keyword. You will find usually more than one image that provides that service:

The screenshot shows the Docker Hub search results for the keyword 'nginx'. The interface includes a search bar at the top with 'nginx' entered, and navigation links for 'Explore', 'Pricing', 'Sign In', and 'Register'. On the left, there are filters for 'Products' (Images, Extensions, Plugins) and 'Trusted Content' (Docker Official Image, Verified Publisher, Sponsored OSS). The main results area shows two images:

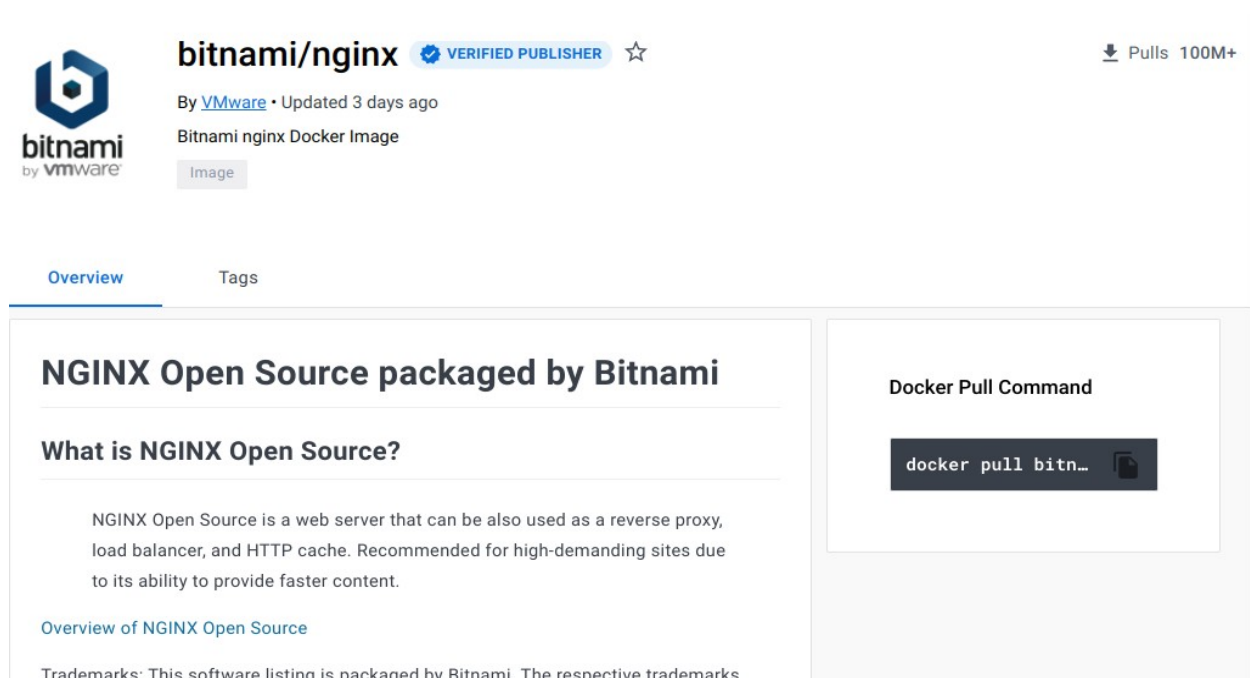
- nginx** (DOCKER OFFICIAL IMAGE): Updated an hour ago. Official build of Nginx. Supports Linux, ARM, ARM 64, 386, mips64le, PowerPC 64 LE, IBM Z, and x86-64. It has 1B+ downloads and 1 star. Pulls: 31,810,315 (Last week).
- bitnami/nginx** (VERIFIED PUBLISHER): By VMware, Updated 6 days ago. Bitnami nginx Docker Image. It has 100M+ downloads. Pulls: 1,078,701 (Last week).

Docker Hub search results

Anyone can publish an image to Docker Hub, but images can contain malware or backdoors, so they are more or less trustworthy depending on who published them. According to the publisher, the images can be classified on:

- **Official images:** Those images are reviewed by a Docker's dedicated team. They have good documentation and are well supported. They are the most trustworthy ones.
- **Verified publisher:** The publisher of the image has been verified by Docker, so you can trust those images
- **Rest of the images:** Images that hasn't been verified. They are often harmless, but you should use them with care in your production system.

Once you have selected an image, in its page you can find relevant information like the way in which you should use the image. If you don't know how to use an image, you will probably find that information in this page:



The screenshot shows the Docker Hub page for the `bitnami/nginx` image. At the top, the image name is displayed with a 'VERIFIED PUBLISHER' badge and a star icon. Below this, it says 'By VMware • Updated 3 days ago' and 'Bitnami nginx Docker Image'. The 'Overview' tab is selected, showing a title 'NGINX Open Source packaged by Bitnami' and a section 'What is NGINX Open Source?' which describes NGINX as a web server, reverse proxy, load balancer, and HTTP cache. A 'Docker Pull Command' box on the right shows the command `docker pull bitnami/nginx`.

You also have a “Tags” tab with the different tags for that image. To use an image you reference it by its name and its tag. For example, if you want to use the Debian version of nginx 1.22.1 you could create a container with that image with:

```
docker create --name nginx bitnami/nginx:1.22.1-debian-11-r46
```

Images are pulled automatically from Docker Hub if you don't have them in your computer, but you can pull an image before with `docker pull <image>`.

If you don't specify a tag when using an image, the `latest` image will be used. For example, `docker pull nginx` will be the same as `docker pull nginx:latest`.

Overview

Tags

Sort by

Newest

Filter Tags

TAG

[latest](#)

docker pull bitnami/nginx:la...

Last pushed 3 days ago by [bitnamibot](#)

DIGEST	OS/ARCH	SCANNED	COMPRESSED SIZE
<a href="#">56fd7ff483c8</a>	linux/amd64	---	36.33 MB
<a href="#">44e4e8bed0ec</a>	linux/arm64	---	34.84 MB

TAG

[1.22.1-debian-11-r45](#)

docker pull bitnami/nginx:1...

Last pushed 3 days ago by [bitnamibot](#)

DIGEST	OS/ARCH	SCANNED	COMPRESSED SIZE
<a href="#">8a2621aef557</a>	linux/amd64	---	36.33 MB
<a href="#">8e68f674a0cd</a>	linux/arm64	---	34.84 MB

TAG

[1.22.1](#)

docker pull bitnami/nginx:1...

Last pushed 3 days ago by [bitnamibot](#)

DIGEST	OS/ARCH	SCANNED	COMPRESSED SIZE
--------	---------	---------	-----------------

List of tags of a Docker image

## 5.3 Creating images

Sometimes you will suffice with published images but, when they don't, you can create your own image. Docker provides a way for doing that, the `Dockerfile` and the `docker build` command.

### 5.3.1 Dockerfile and building

A Dockerfile is a plain text file with a set of instructions to build the image. An example of a simple Dockerfile could be:

```
FROM python:3.8
RUN mkdir /scripts
COPY ./hello.py /scripts
CMD ["python", "/scripts/hello.py"]
```

We will see the meaning of those lines in the following section. With the Dockerfile we can build an image with this command:

```
docker build --tag <image name> <directory of the Dockerfile>
```

The directory we pass to `docker build` is called the context. We can reference any files in that directory in the build instructions. With the `--tag` option we will assign a name for the generated image.



### 5.3.2 Dockerfile commands

In this course, we will see only a subset of all available options for a Dockerfile. You can check all the available options in the documentation: <https://docs.docker.com/engine/reference/builder/>

#### FROM

This instruction is used for setting the base image for creating the new one. For example, if we want to build an image based on the `ubuntu` image we will use:

```
FROM ubuntu
```

As the first instruction in our Dockerfile. This instruction may only exist once per build stage and must appear as the first instruction (except for the `ARG` instruction, which may appear before)

#### RUN

When we utilize the `RUN` directive in Docker, we execute a command line instruction inside the “container” while constructing the image. The resulting changes are then placed on top of the existing image as a fresh layer.

For example, if we want to install `cowsay` in a base ubuntu container:

```
FROM ubuntu
RUN apt update
RUN apt install -y cowsay
```

That is to say, we run the same commands we would run in the terminal to install that application. After building the image, we will have the command `cowsay` available in all the containers created from that image.

The above example uses the shell notation: the commands are sent to a shell, which will execute them using substitutions, etc. There is another notation, the exec notation. This is the same example using the exec notation (the `#` at the start of the line means that the line is a commentary and won't be processed):

```
FROM ubuntu
# RUN apt update
# RUN apt install -y cowsay
# This is almost equivalent:
RUN ["apt","update"]
RUN ["apt","install","-y","cowsay"]
```

When building images, Docker will use the cache and won't regenerate a layer if it has been generated before. So, for example, if we build the above Dockerfile twice, the second build will be much faster than the first.

#### COPY

The `COPY` instruction adds files and directories to the image. The source will be the build context (the directory provided to the build command) For example, if we want that the file `my_script.sh` and the directory `./data/files` to be in the image we will do:

```
COPY my_script.sh /bin/my_script.sh
```

```
COPY data/files /destination
```

Take in account that you can't copy files from outside the context of the build. You will also need to assign the corresponding permissions to the files in the container.

## CMD

**CMD** (short for "Command") is an instruction in a Dockerfile that specifies the default command to be executed when a container based on the image is started. If the user does not provide any command-line arguments, the default command specified in the **CMD** instruction will be executed.

For example, let's say you want to create a Docker image that runs a Python script that prints "Hello, World!". You could create a Dockerfile with the following content:

```
FROM python:3
CMD python -c "print('Hello, World!')"
```

When you build this Docker image and start a container based on it, the python command will be executed by default:

```
docker build -t example-cmd .
docker run --rm example-cmd

Hello, World!
```

The **CMD** can be overridden by command line parameters. So, for example, we can run:

```
docker run --rm example-cmd echo "Potato"
Potato
```

The **echo "Potato"** command will be run instead of the **CMD** in the Dockerfile.

## ENTRYPOINT

Using **ENTRYPOINT** you can define an order that will be executed when the container starts, like **CMD**. However, **ENTRYPOINT** can't be overridden and will execute always. This is the command used by containers that run a service, like web servers or database server.

For example, if we create

```
FROM python:3
ENTRYPOINT python -c "print('I will survive')"
```

We won't be able to

```
docker build -t example-entrypoint .
docker run --rm example-entrypoint echo "Potato"
I will survive
```

If you use the exec notation for **ENTRYPOINT**, the parameters in the command line will be append as parameters to the entry point. For example:

```
FROM python:3
ENTRYPOINT ["python", "-c"]
```

It will add the command line parameters to the **ENTRYPOINT**, executing that code (the instructions after **-c** in python are interpreted and executed):

```
docker run --rm example-entrypoint print('From the command line')
From the command line
```

Here you have a comparison table between **CMD** and **ENTRYPOINT**:

CMD	ENTRYPOINT
Will execute if no cmd specified when running the container	Will execute always
Can be overridden by command-line parameters	Can't be overridden. Parameters are append to the ENTRYPOINT instruction if defined in exec format.

## 6.NETWORKING

Docker networking is a complex topic. There are multiple network models available, and you can also install third-party plugins to get additional options. Nevertheless, in this course we will work only with the **bridge** network driver. If you want to learn more about Docker networking you can check the official documentation: <https://docs.docker.com/network/>

### 6.1 Default bridge

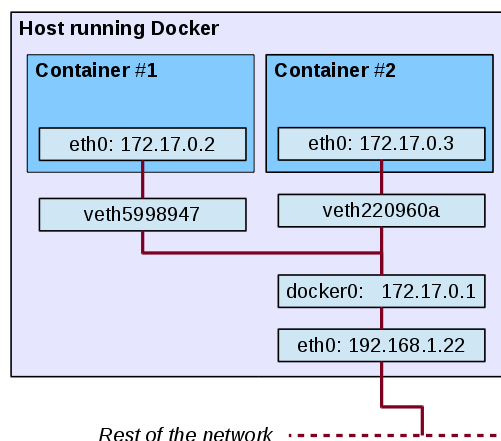
When you install Docker in your machine, a new interface **docker0** is created. You can show it running **ifconfig**:

```
docker0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    inet 172.17.0.1 netmask 255.255.0.0 broadcast 172.17.255.255
    ether 02:42:90:9b:6f:51 txqueuelen 0 (Ethernet)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

All containers will connect to that interface unless otherwise specified. It works as follows:

The docker0 default bridge

- The containers will have an IP in the **docker0** network



- A container can talk with another containers, using the container's internal IP address. You can get that address running a command inside the container or using docker inspect.
- The containers can connect to external sites. They have `docker0` as the gateway: all connections goes through it. It implements NAT, like the router in a home network installation. The interface `docker0` will be like the router in your house, and the containers like the devices you connect to your home's router.
- Your container can't be reachable from the outside.
- If you want to make your container available from the outside, you need to use port mapping to make it available in one of your host's ports.

### 6.1.1 Example:

In our host, we have the `docker0` interface:

```
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UP group default
    link/ether 02:42:90:9b:6f:51 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:90ff:fe9b:6f51/64 scope link
        valid_lft forever preferred_lft forever
```

If we create a container using the `busybox` image (that image has network utilities installed and opens a shell by default):

```
docker run --rm -ti --name containerA busybox
```

We can see that it has an `eth0` interface with the IP `172.17.0.2`, in the same network as our host's `docker0` interface:

```
/ # ip a
...
28: eth0@if29: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
noqueue
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
        valid_lft forever preferred_lft forever
```

And the default route is to the `docker0` IP in our host:

```
/ # route -n
Kernel IP routing table
Destination      Gateway          Genmask          Flags Metric Ref    Use Iface
0.0.0.0          172.17.0.1      0.0.0.0          UG    0      0      0 eth0
172.17.0.0       0.0.0.0         255.255.0.0      U      0      0      0 eth0
```

We can reach the outside from our container, try for example `ping portal.edu.gva.es`.

If we create another container:

```
docker run --rm --name containerB busybox
```

That container will have an address in the same network as the previous one:

```
14: eth0@if15: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
noqueue
```

```
link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
inet 172.17.0.3/16 brd 172.17.255.255 scope global eth0
valid_lft forever preferred_lft forever
```

And they can communicate:

```
ping 172.17.0.2
PING 172.17.0.2 (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.299 ms
64 bytes from 172.17.0.2: seq=1 ttl=64 time=0.142 ms
64 bytes from 172.17.0.2: seq=2 ttl=64 time=0.142 ms
```

The network `172.17.0.0/16` is a private network, so it can't be reached from the outside.

## 6.2 Custom bridge networks

Custom networks are used to isolate a group of containers that should communicate only between them. They work the same way as the default network, with these differences:

- They can reach only containers in the same network
- They can reach the containers in the same network by name
- You can connect and disconnect a container from a custom network at run time.

### 6.2.1 Example

Lets create a custom network in docker:

```
docker network create network-one
```

After creating the network, we will have another interface in our machine. Its name will be something like `br-XXXX`:

```
18: br-61ce1d3b85de: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc
noqueue state DOWN group default
link/ether 02:42:2b:c5:99:78 brd ff:ff:ff:ff:ff:ff
inet 172.20.0.1/16 brd 172.20.255.255 scope global br-61ce1d3b85de
valid_lft forever preferred_lft forever
```

We can launch a container in that network with the `--network` parameter:

```
docker run --rm -ti --name container-A --network network-one busybox
```

The container will have an IP of that new network:

```
23: eth0@if24: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
noqueue
link/ether 02:42:ac:14:00:02 brd ff:ff:ff:ff:ff:ff
inet 172.20.0.2/16 brd 172.20.255.255 scope global eth0
valid_lft forever preferred_lft forever
```

If we launch another container in the same network:

```
docker run --rm -ti --name container-B --network network-one busybox
```

It will be able to communicate to the others in the same network by IP **and by name**:

```
/ # ping 172.20.0.2
```

```
PING 172.20.0.2 (172.20.0.2): 56 data bytes
64 bytes from 172.20.0.2: seq=0 ttl=64 time=0.249 ms
64 bytes from 172.20.0.2: seq=1 ttl=64 time=0.144 ms

/ # ping container-A
PING container-A (172.20.0.2): 56 data bytes
64 bytes from 172.20.0.2: seq=0 ttl=64 time=0.303 ms
64 bytes from 172.20.0.2: seq=1 ttl=64 time=0.145 ms
```

If we launch a container in another network (the default one, for example):

```
docker run --rm -ti --name container-C busybox
```

It won't be able to communicate with the containers in the custom network:

```
# ping 172.20.0.2
PING 172.20.0.2 (172.20.0.2): 56 data bytes
^C
--- 172.20.0.2 ping statistics ---
3 packets transmitted, 0 packets received, 100% packet loss
```

### 6.2.2 Removing networks

You can remove a network when you don't need it with: `docker network rm <network name>`

If you want to remove all unused networks, you can run `docker network prune`. This will remove all the networks that are not currently used by a container.

## 7.SUPPLEMENTARY MATERIAL

Custom bridge networks:

<https://blog.devgenius.io/custom-docker-bridge-networks-how-to-run-containers-b8d40c51bab2>

DockerFile

<https://www.ionos.com/digitalguide/server/know-how/dockerfile/>