CENTRE ESPECÍFIC D´EDUCACIÓ A DISTÀNCIA DE LA COMUNITAT VALENCIANA

UT 011. DOCKER

Computer Systems
CFGS DAW

Álvaro Maceda
a.macedaarranz@edu.gva.es

2022/2023
Version:230227.0631

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

UT 011. Docker

1. What is Docker

When we deploy a web application, we need to prepare a computer with all the software we will need to run it: libraries, external packages, additional files, etc. We need to be able to recreate that computer for development and testing purposes. For example, if we are developing a web application on our personal computer we need to be sure that all the libraries are the same as the ones that we will use on the computer or computers where that application will be deployed later.

Maybe we could replicate the system in our computer for one application, but what happens if we are working on multiple applications? It will be very difficult and error-prone to keep our computers in sync.

We need a way to create a reproducible computer

1.1 Virtual machines

A way to create an isolated and reproducible environment is through the use of a virtual machine (VM). Essentially, a VM is a single computer residing within a host machine, where multiple VMs can coexist within one host machine. The creation of a VM involves the virtualisation of the host machine's underlying hardware ─processing, memory, and disk─ which are emulated using the real hardware of the host machine.

However, this virtualisation process incurs significant computational overhead. In addition, each VM is a separate machine and requires its own operating system, which typically requires tens of gigabytes of storage and time to install, a process that must be repeated every time a new VM is spun up.

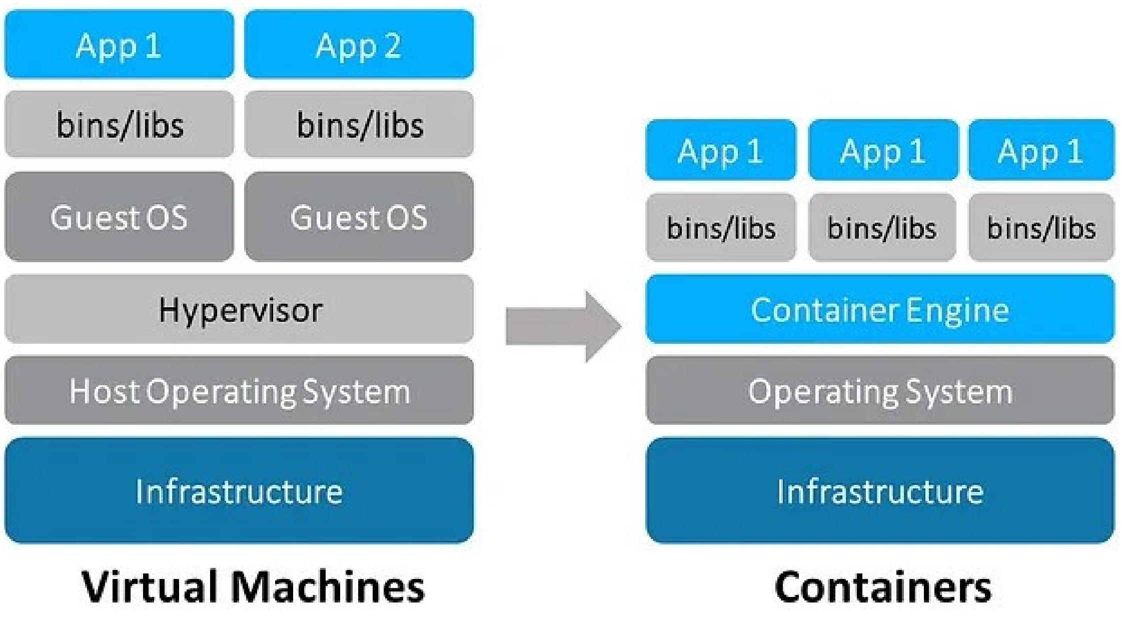
Although virtual machines (VMs) provide many benefits, such as the ability to create an isolated environment and run multiple instances of an operating system on a single host machine, if we use virtual machines for deploying applications they can generate some problems :

- **Resource Overhead:** The process of virtualization involves the creation of a virtual machine that has its own operating system, applications, and resources. This requires a significant amount of computing resources, which can lead to high resource overhead and reduced performance.
- **Storage Overhead**: Each virtual machine requires its own operating system and applications, which can result in significant storage overhead. This can be especially problematic when deploying multiple VMs, as each one will require a separate set of resources.
- **Complexity**: Virtual machine environments can be complex and difficult to manage. They often require specialized expertise to set up and maintain, and can be difficult to troubleshoot in the event of an issue.
- **Security**: Virtual machines can introduce additional security risks to an environment. Each VM is essentially a separate machine with its own operating system, which means that

security must be maintained for each individual VM. Additionally, virtual machines can be more vulnerable to attacks that target the virtualization layer.

1.2 Containers

Containers try to solve some of those problems, creating a lightweight and portable way for creating isolated environments. Containers utilize a different method for achieving isolation: they operate similarly to virtual machines, residing on top of a host machine and using its resources, but instead of virtualizing the hardware, they virtualize the host operating system. As a result, containers are much more lightweight than virtual machines and can be spun up quickly, as they do not require their own operating system.



The Docker daemon serves as a counterpart to the hypervisor layer in container deployment (assuming Docker is being used), acting as an intermediary between the host operating system and the containers. This results in lower computational overhead compared to hypervisor software, as illustrated by the thinner box in the diagram above, further contributing to the lightweight nature of containers compared to virtual machines.

Containers do not provide the same level of isolation from the host system as virtual machines do. However, they do provide enough isolation to prevent the configuration of the host operating system from affecting the operation of the application (for the most part).

A container engine can run **only the same operating system as the host**. Nevertheless, Windows and Mac OS operating systems provides a way of running Linux containers. That’s because those operating systems runs a lightweight Linux kernel and, on top of them, the container engine.

Some of the advantages of the containers are:

- **Lightweight**: Containers are much more lightweight than VMs. Containers share the host operating system and do not require a separate guest operating system, which makes them more efficient and faster to start up.
- **Fast startup time**: Because containers do not need to boot an entire operating system, they can start up in just a few seconds. This makes it easy to spin up new containers quickly and scale applications as needed.
- **More efficient resource utilization**: Containers are more efficient in their use of resources than VMs. Since they share the host operating system, they require fewer resources to run, which means you can run more containers on a single host than you could VMs.
- **Greater flexibility**: Containers offer more flexibility than VMs. You can customize container images to include only the necessary components for your application, which reduces the attack surface and minimizes the risk of vulnerabilities.

In this course we will learn Docker, one of the most popular containerization platforms today, although there are other alternatives like Podman or LXD.

2. An example of Docker

Before we learn more about Docker, let's take a look at a quick example so you can have a sense of what we will be able to do with a container system. To run this example, you will need Docker installed in your system.

We will run a web server from our machine, serving the pages of a local directory. We will need to create an `index.html` file in the current directory with this content:

```
<html>
  <head>
    <title>Hello from Docker</title>
  </head>
  <body>
    <h1>Hello!</h1>
    <p>This is a web page served with nginx</p>
  </body>
</html>
```

After creating that file, we will serve it using nginx, a web server. But, instead of installing nginx in our machine, we will use a Docker container:

```
docker run --rm --publish 8080:80 \
           --mount type=bind,src=`pwd`,dst=/usr/share/nginx/html,readonly \
           --detach --name docker-example nginx
```

The exit of that command will be something like this:

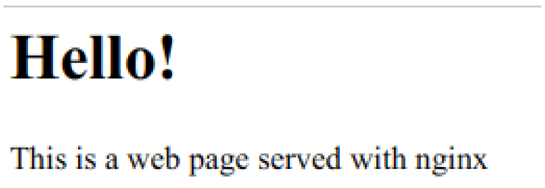
```
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
01b5b2efb836: Pull complete
db354f722736: Pull complete
abb02e674be5: Pull complete
214be53c3027: Pull complete
a69afcef752d: Pull complete
```

```
625184acb94e: Pull complete
Digest:
sha256:c54fb26749e49dc2df77c6155e8b5f0f78b781b7f0eadd96ecfabdcdfa5b1ec4
Status: Downloaded newer image for nginx:latest
db032f2b2e159d6ab2d47b07fd7f3789e30e65d979bfd10f23dc993551e49c48
```

This command is doing a couple of things:

- Downloading the container images to your computer (if you run it again, it won't pull the images again because you will already have it in your system)
- Starting the container. The container, in this case, is running nginx
- Sharing the current directory with the container
- Redirecting port 8080 of your machine to port 80 of the container

After starting the container, we can open the URL http://localhost:8080 and we will see the page served by the container:



Hello!

This is a web page served with nginx

To stop the container, we can run:

```
docker stop docker-example
```

Now, let's see what's happening under the hood.

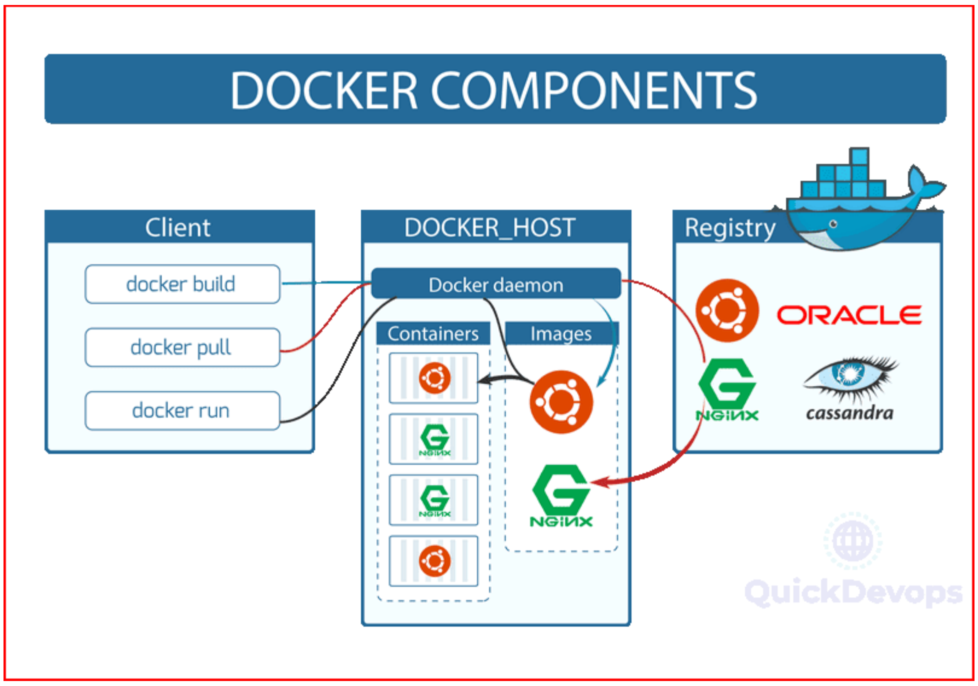
3. Docker architecture

Docker is composed of multiple components. We will be working mostly with the Docker client (docker) but it's important to know all its parts.

Docker's architecture is based on these components:

- **Images**: The images are like the "hard disk" for the virtual computer. Through Dockerfiles we can create those images that will generate virtual computers with the same files. In the above example, we are using an image called `nginx:latest` which generates a computer running a web server.
- **Containers**: From an image, we can create and run "virtual computers". Those "computers" are called containers. We can create multiple containers from the same image (it's like installing multiple computers with the same configuration) In the above example, we are calling the container docker-example. You can run the `docker run` command multiple times (using different names) to launch multiple identical containers.
- **Docker Client**: The docker client is the command line tool that allows the user to interact with the daemon. We will be running that command (`docker`) with different arguments to use docker.

- **Docker Daemon**: The docker client talks to a background service running on the host that does all the work: manages images and building, running and distributing containers.
- **Docker Hub**: It's an internet repository containing available Docker images. If we don't have an image in our computer, docker will search for it in a repository and download it to our machine.



In the `nginx` example, this is what happened:

1. We used the docker client (`docker run...`) to tell Docker that we wanted to create and start a container from an image called `nginx`. The docker client is the executable named `docker` that we launch from the command line
2. The docker client talked to the docker service that was running in our machine, translating our instructions to its own protocol
3. The docker service saw that it did not have the image `nginx` available
4. The docker service sends an Internet request to the Docker hub (https://hub.docker.com/) seeking that image. The hub answer with the information about that image.
5. The docker service downloaded the image to our machine. An image, as we will see later, is composed of multiple layers that can be downloaded independently
6. Once it had the image, the docker service created a container with the configuration we provided and ran it.

Let's see a little bit more about those components.

3.1 Docker client and service

3.1.1 Client

The Docker client is the executable that we will use to perform operations with Docker. It's and executable named `docker`.

It has multiple sub-commands, each one used to perform different actions. You can view all the different possibilities in Docker's documentation:

https://docs.docker.com/engine/reference/commandline/cli/

Each sub-command is like a different command, and they have multiple command line options. Most of them are not used often. Anywhere, you should check the docker documentation for each command and take a look at the different options to be aware of the variety of things that you can achieve with each command.

Don't be overwhelmed by all the possibilities: we will be using only a tiny subset of them in this course.

3.1.2 Server

The Docker service is in charge of performing the operations that we send through the docker client. The Docker client does nothing: only talks to the server to perform operations.

As the client and the server are different components, we could configure our client to talk to a docker server on another machine. We will not see that kind of configuration in this course, but you must be aware of the possibility.

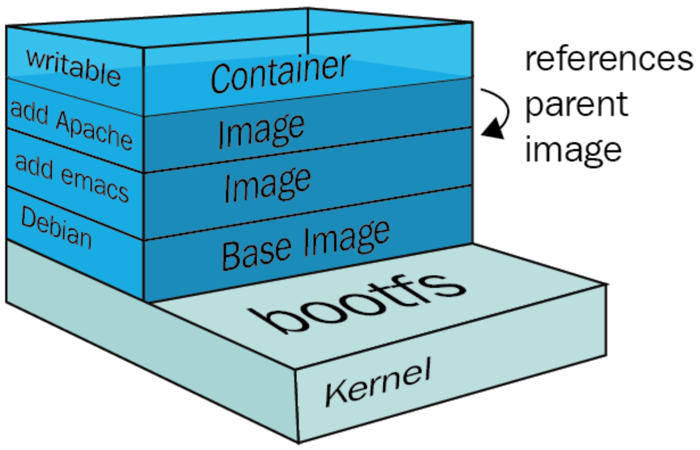
You can manage the docker service like any other service using the `systemctl` and `service` commands. For example, you can run `systemctl status docker` to see the status of the service. Of course, if the service is stopped you won't be able to work with docker on your machine.

3.2 Images and hubs

A Docker image is something similar to a disk image for a virtual machine. We create an image "installing" things in the base computer, and once we are done we have a hard drive from which we can create multiple identical computers.

Those hard drives, or images, are structured in layers. From a base layer (usually provided by Docker) we create new layers by adding, removing or modifying files. This allows us to share layers between different containers.

Once we start a container using an image, we add a new layer on top of it. The only writable layer is the last one, so the other layers can be shared by the other containers and images:



From the container's viewpoint, it has a "normal" hard drive. The layered architecture is something internal to Docker and most of the time you won't need to care about it.

With that in mind, we can classify the images into base and child images:

- **Base images** are images that have no parent image: they don't build on or derive from another image. They are usually images that represent an operating system (e.g. Ubuntu, busybox...). You won't create containers using those images, because they are useless by themselves.
- **Child images** are images that build on base images and add additional functionality, most images you're likely to make will be child images.

It's also important to know that there are some images (official images) which are more reliable than others:

- **Official images** are images that are officially maintained and supported by the people at Docker. These are typically one word long. Examples include `nginx`, `ubuntu`, and `hello-world`.
- **User images** are images created and shared by people who use Docker. They usually build on base images and add functionality. Typically these are formatted as `user/image-name`.

Sometimes you will need to create your own images, but most often you will be using already created ones. A Docker Hub is a place to look for useful images, and the most used one is the Docker Hub:

https://hub.docker.com/

You can find there images ready for use with some of the most useful applications and services. We will talk more later about using a hub.

3.3 Containers

A container is like a virtual computer. Using an image (the "hard drive") to create a container is like configuring the computer: assigning shared resources, networking, shared ports...

Once we have the container configured, we can start and stop it as many times as we want, and destroy it when we won't need that "computer" anymore.

Containers can be in one of those states:

- **created**: Docker assigns the created state to the containers that were never started ever since they were created. No CPU or memory is used by the containers in this state: it's like a stopped virtual machine.
- **running**: The container is running: some processes are running in the isolated environment inside the container, using memory and CPU.
- **exited**: Once the container finishes executing its main command, it exits. It's like it shutdowns once the task has finished. Most of the containers don't exit, because they run a service that keeps running.

  For example, if we run a ngnix container it won't end until we manually stop the container. However, others will just execute a command and finish. If we run a `hello-world` container, it will stop when it finishes.
- **paused**: Containers can also be paused, consuming memory but not CPU.
- **dead**: if the container has an error it will be in a dead state.

You will understand more about these states when we work with containers in a later section

4. Working with containers

4.1 Creating containers

The first thing you need to work with a container is to create one. The simplest way to create a container is to use docker create:

https://docs.docker.com/engine/reference/commandline/create/

The usage of the command is:

```
docker create [OPTIONS] IMAGE [COMMAND] [ARG...]
```

The most important part here is the IMAGE. We always create a container from an image, so it's a required argument for docker create. For example, to create a container from the nginx image we can execute:

```
docker create nginx
```

It will output the ID of the newly created container (we can list all the containers with `docker container ls --all`)

The second most relevant part is the OPTIONS. We have many options here; with those options, we can configure the "virtual machine" by adding memory limits, IP addresses, CPU quotas... For example, for creating a container named `ubuntu_container` with a memory limit of 1GB we can run:

```
docker create --name ubuntu_container --memory 1g -ti ubuntu
```

The third part is the command that will be executed when starting the container (remember that the containers are not running yet) The images usually ship with a useful initial command, so we are not modifying it with `docker create` in this course.

4.2 Starting, pausing and stopping containers

4.2.1 Starting containers

Once we have the container created, we need to start it so it begins to execute:

```
docker start <container name>
```

That command will output the container name and return to the console. If we examine the running containers, we can see that the container is running:

```
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS          PORTS     NAMES
f36dc40fad64   nginx     "/docker-entrypoint.…"   2 minutes ago   Up 31 seconds   80/tcp    mycontainer
```

A container will run until its main process finishes. Containers usually provides a service like a database server, a web server... so often they will keep running until we stop them. Nevertheless, in some cases, the container will execute something and finish, so we won't be able to see it running.

4.2.2 Attaching to a container

When you start a container you can't see the output it generates, nor send commands to it. That's because, usually, containers provide services and we don't need to interact with them.

To see the output of the container, we can use the following command:

```
docker logs mycontainer
```

It will output the `stdout` of the container to our console. We can use the `--follow` parameter to keep watching for new outputs from the container. For example, if we create and start an nginx container, the output of `docker logs` could be something like this:

```
...
...
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2023/02/21 12:13:43 [notice] 1#1: using the "epoll" event method
2023/02/21 12:13:43 [notice] 1#1: nginx/1.23.3
2023/02/21 12:13:43 [notice] 1#1: built by gcc 10.2.1 20210110 (Debian 10.2.1-6)
2023/02/21 12:13:43 [notice] 1#1: OS: Linux 5.15.0-60-generic
2023/02/21 12:13:43 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2023/02/21 12:13:43 [notice] 1#1: start worker processes
2023/02/21 12:13:43 [notice] 1#1: start worker process 22
2023/02/21 12:13:43 [notice] 1#1: start worker process 23
2023/02/21 12:13:43 [notice] 1#1: start worker process 24
2023/02/21 12:13:43 [notice] 1#1: start worker process 25
```

Sometimes we will need to interact with the container. We can attach the stdin, stdout and stderr of our console to the container so, in practice, it will be as if we were working in the container console. We can do that with:

```
docker attach mycontainer
```

We can also start the container in interactive mode using the `--interactive` or `--attach` parameters in the `docker start` command.

4.2.3 Stopping containers

If the container hasn't finished itself, we can stop it with:

```
docker stop mycontainer
```

It will not use CPU or memory until we start it again, but it will use some of the space in our hard disk.

You can also pause a container with `docker pause mycontainer`. It will not use CPU but it will still use memory from your host.

4.2.4 Removing containers

Removing a container will destroy all its configuration and its internal data. It won't use hard disk space anymore:

```
docker rm mycontainer
```

We can use the order `docker container prune` to remove all stopped containers. Use it with care.

4.3 Docker exec

We can run additional commands in an already-running container. One of the most common uses of that utility is to open a shell in the container to run some administrative commands, check the content of files, etc.:

```
docker exec -ti mycontainer /bin/sh
```

Once we have finished with the console, we can terminate it with `exit` or Control+D to return to the host computer.

To be able to interact with the command, we can use the `--tty` and `--interactive` options (or its abbreviation `-ti`) If we want to run an unattended command we can use the `--detach` option: it will launch the command in the container and return to our console directly.

The executable must exist in the container. Containers are an oversimplified version of Linux computers, so often we won't have some of the most used commands available, like `ps`, network tools, or the `bash` shell. We can install whatever we want into the container, but everything we do will be lost when we destroy the container.

That kind of operation should be used only while developing the container: the philosophy of docker is that containers could be created and discharged easily from the image so, once we have things clear, we should modify our image to reflect the changes we want for the container. We will talk about images later.

4.4 Docker run

Instead of creating a container and starting it later, we can do both operations in a single command with docker run. This option is more commonly used than the combination of `docker create` + `docker start`:

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

As you can see, the command is similar to docker create but it will start the container, too. The options are the same as docker create. It will also attach to the container. If we want to run the container in the background, we can use the `--detach` option.

So, for example, to create and start a nginx container with a single command you could use.

```
docker run nginx
```

You must be aware that `docker run` **won't destroy the container once it has stopped**. If you haven't specified a name when executing the command, Docker will create a random one. So if, for example, you run the command `docker run nginx` and then stop the container with Ctrl+C three times, you can end with three stopped containers like these, and you would need to remove them manually:

```
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS                       PORTS     NAMES
a0dba86b4936   nginx     "/docker-entrypoint.…"   7 seconds ago    Exited (0) 5 seconds ago               focused_faraday
17113cd4cd60   nginx     "/docker-entrypoint.…"   9 seconds ago    Exited (0) 7 seconds ago               wonderful_jang
6b4485de9248   nginx     "/docker-entrypoint.…"   21 seconds ago   Exited (0) 18 seconds ago              nostalgic_euclid
```

But if you use the command line option `--rm` the container will be automatically destroyed once it has finished its execution.

We can also run a command in the container. For example, we can open a shell in a nginx container with this command. The nginx server won't be running yet:

```
docker run -ti --rm nginx /bin/sh
```

Then we can modify files, start the service with `service start nginx`, etc, but the container will stop (and it will be destroyed, because we have used the `--rm` option) when we exit the shell.

We can also set a name for the container with the `--name` option (it does make sense to use that option along with the `--rm` option if we detach the container and we want to interact with it later)

4.5 Port mapping

Often, we use containers that provide network services. We will see Docker networking later, but Docker provides a quick way of making those services available in our host: the `--publish` or `-p` flag. By using that flag we *redirect* a port in our machine to a port in the container. So, for example, when we connect to the port 8080 in our machine we would actually be connecting to the port 80 in a container.

Port mapping is done when creating the container. The flag has an argument of multiple values separated by colon. The first value is the port in our machine, and the second the port in the container. With this port mapping:

```
docker run --rm --publish 8080:80 \
         --detach --name docker-example nginx
```

When we connect to port 8080 in our machine (by navigating to http://localhost:8080, for example) we will be connecting to the port 80 in the container.

We can only use one port on our machine for a redirection. If we want to redirect to several containers, we will have to use different ports. For example, we can redirect the port 80 to the port 80 of one container and the port 81 to the port 80 of another container.

You can also specify the IP of your host that you want to redirect, and the protocol (TCP, by default or UDP) with this syntax: `-p 192.168.1.100:8080:80/udp`

4.6 Getting information about containers

There are two commands to get information about the containers:

`docker ps` will show information about the running containers. For example:

```
➜  ~ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED           STATUS                  PORTS     NAMES
2e4fb80e36da   nginx     "/docker-entrypoint.…"   22 seconds ago    Up 3 seconds            80/tcp    another_container
f36dc40fad64   nginx     "/docker-entrypoint.…"   16 minutes ago    Up 6 minutes (Paused)   80/tcp    mycontaine
```

It will show only running and paused containers. To see all the container we can use the `--all` parameter:

```
➜  ~ docker ps --all
CONTAINER ID   IMAGE     COMMAND                  CREATED           STATUS                  PORTS     NAMES
aeec1ee55bd7   wordpress "docker-entrypoint.s…"   7 seconds ago     Created                           one_more_container
2e4fb80e36da   nginx     "/docker-entrypoint.…"   2 minutes ago     Up 2 minutes               80/tcp    another_container
f36dc40fad64   nginx     "/docker-entrypoint.…"   18 minutes ago    Up 8 minutes (Paused)      80/tcp    mycontainer
```

This show us basic information about the containers. We can get all the information of a container, including the state, with `docker inspect`. It will return a big object in JSON format with all the information about the container:

```
➜  ~ docker inspect mycontainer
[
    {
        "Id":
"f36dc40fad64887d0b50148508ce2a7be83e8ed098d2d8fa24e6ebc7e0d916e5",
        "Created": "2023-02-17T11:32:06.105645545Z",
        "Path": "/docker-entrypoint.sh",
        "Args": [
            "nginx",
            "-g",
            "daemon off;"
        ],
...
...
...
                    "IPAddress": "172.17.0.2",
                    "IPPrefixLen": 16,
                    "IPv6Gateway": "",
                    "GlobalIPv6Address": "",
                    "GlobalIPv6PrefixLen": 0,
                    "MacAddress": "02:42:ac:11:00:02",
                    "DriverOpts": null
                }
            }
        }
    }
]
```

We can filter the information that we want to see using the `--format` parameter. For example, `docker inspect -f '{{.State.Status}}' mycontainer` will return only the state of the container.

4.7 Persisting data

Let's see what happens with the data that we have inside the containers. First, let's run a new Ubuntu container:

```
docker run -ti --name mycontainer ubuntu /bin/bash
```

This will open a bash shell inside our new container. We can create a new file inside the root directory:

```
root@3153657f9594:/# echo "Some content" > /a_new_file.txt
root@3153657f9594:/# cat a_new_file.txt
Some content
```

Then, we can type "exit": that will terminate the shell. If we start the container again:

```
docker start --interactive mycontainer
```

We can see that the file is still there, with the same contents. So one way of persisting data will be to keep the container until we don't need the data anymore. However, Docker's philosophy is that containers are disposable, so we need some kind of data persistence system that stores data somewhere outside of containers. Docker provides two ways of doing this data persistence: bind mounts and volumes. Both solutions work by mounting a data storage space in a directory in the container, similar to how a file system is mounted in Linux.

You must configure the bind mounts (and the volumes) when creating the container with `docker create` or `docker run`. **Once the container is created, we won't be able to add bind mount or volumes to it**.

There are two different parameters for data persistence: `-v` or `--volume`, and `--mount`. We will be using the more explicit `--mount` syntax through the examples. Both options work almost the same way.

There are some caveats with Linux and Mac when using volumes, please take a look to the documentation if you have any problem with them.

4.7.1 Bind mounts

With a bind mount, we mount a host folder inside the container's folder. This is the simplest solution to persist container data, and it's used often when we use containers for developing applications.

We used a bind mount for the example at the start of the unit:

```
docker run --rm
           --mount type=bind,src=`pwd`,dst=/usr/share/nginx/html,readonly \
           --detach --name docker-example nginx
```

The `--mount` parameter consists of a series of options separated by a comma. The order of the options is not relevant:

- For bind mount, we will need to include the `type=bind` option
- The `source` option is the path to the file or directory in the host machine. You can mount both single files and directories (Docker will detect it, depending on the path, and will act accordingly) The source option **must be an absolute path**.

  In the above example, we use `` `pwd` `` because that expression will execute `pwd` and substitute the expression with the results of that command before executing docker run.
- The target options is the path inside the container. It can be written also as `destination` or `dst`, it's the same in all cases.
- The `readonly` option is optional. We use it when we don't want the container to have write permissions in that folder.

You can bind mount the same host's folder to more than one container. This will enable them to share data.

4.7.2 Volumes

Volumes are a more advanced solution for persisting data. They are stored in a part of the host file system managed by Docker.

To create a volume, use the following command:

```
docker volume create my_volume
```

That will create a new volume named my_volume. To use that volume in a new container:

```
docker run --rm -d -ti --name ubuntu_container4 \
           --mount type=volume,src=my_volume,dst=/mountpoint \
             ubuntu /bin/bash
```

We can list the current volumes with `docker volume ls`, and get detailed information about the volume with `docker volume inspect my_volume`.

Containers are not deleted when we stop or destroy the container. To remove a container we can execute:

```
docker volume rm my_volume
```

We can also remove all unused volumes with `docker volume prune`.

4.7.3 tmpfs mounts

In Linux hosts, we can also create `tmpfs` mounts. The data in `tmpfs` mounts is temporary (will be deleted when we stop the container) and will be stored in the host's memory.

Unlike volumes and bind mounts, you can't share `tmpfs` mounts between containers.

4.8 Permissions

The default user within a container is root (id=0). So, for example, if we create a container with a bind mount:

```
docker run --rm -ti --name ubuntu_container \
           --mount type=bind,src=/tmp/docker,target=/docker \
           ubuntu /bin/bash
```

The user in the container is root. If we create a file:

```
root@8d2cd626b49c:/# cd /docker/
root@8d2cd626b49c:/docker# whoami
root
root@8d2cd626b49c:/docker# touch a_file
```

The file will be created in the host as if it were created by root. Whe can change the user running the container with the --user option:

```
docker run --rm -ti --name ubuntu_container \
           --user `id -u`:`id -g`
           --mount type=bind,src=/tmp/docker,target=/docker \
           ubuntu /bin/bash
```

`` `id -u`:`id -g` `` is replaced by the current user and group in the host.

We can use numbers for the `--user` parameters and those users don’t need to exist in the container. If we use names, they should exist in the container (the image developer can create additional users)

5. Supplementary material

Docker Introduction — What You Need To Know To Start Creating Containers

https://medium.com/zero-equals-false/docker-introduction-what-you-need-to-know-to-start-creating-containers-8ffaf064930a

Introduction to Containers and Docker

https://endjin.com/blog/2022/01/introduction-to-containers-and-docker

A Docker tutorial for beginners

https://docker-curriculum.com/

Containers:

https://www.digitalocean.com/community/tutorials/working-with-docker-containers

https://www.baeldung.com/ops/docker-container-states

Cheatsheets:

https://phoenixnap.com/kb/docker-commands-cheat-sheet

https://dockerlabs.collabnix.com/docker/cheatsheet/