Docker Documentation

Alejandro Leon Marin

February 24, 2024

Introduction

In this document we will cover the basics of Docker, a containerization platform that allows you to run applications in a controlled environment. Also it will help us to understand the basic concepts of Docker and how to use it. We'll cover the definition of containers, images, and the basic commands to run docker containers.

What is Docker?

Docker is a versatile platform designed for developing, packaging, and running applications within a controlled environment. This platform utilizes containerization, allowing applications to run efficiently on various systems while minimizing resource usage. Developers widely adopt Docker due to its ability to streamline the installation and execution of applications.

A container, in this context, serves as a packaged unit comprising an application and its dependencies, such as libraries and system tools. It functions as an isolated environment, ensuring that the application runs consistently across different systems with Docker installed. Consider a container as a virtual box containing your application's code, language-specific components (e.g., HTML and Node.js), and even environment variables within a designated directory like ".env."

Containers can be easily transported between systems, facilitating seamless sharing among developers and operations teams. Docker containers are stored in repositories, which are akin to GitHub but specifically for containers. There are two types of repositories:

1. Public repositories: Open to the public, allowing anyone to view the containers they contain, with Docker Hub being a prominent example. 2. Private repositories: Access to these repositories is restricted, and only individuals with authorized access can view the containers stored within.

Docker Hub provides a wide array of containers for databases, web servers, and other tools that developers can readily incorporate into their projects. When you run a container, you are essentially launching an instance of an image. This approach becomes particularly advantageous in collaborative projects, where team members may have different software versions installed. Docker simplifies this by enabling users to run an image, based on a specific Linux distribution, with the necessary dependencies—eliminating the need to install them individually.

In summary, Docker facilitates a more efficient and standardized development and deployment process by encapsulating applications and their dependencies into containers, which can be easily shared and executed across diverse environments.

What is an image?

So an image is a file that contains all the necessary files to run a container. It includes the code, the runtime, the libraries, the environment variables, and the configuration files and the container makes that all of this files run in a controlled environment. So the image is the file that you can share with other developers and with the operations team. So what's a container?

What is a container?

A container are layers trough layers of images. Where the lowest layer is most of the time a linux system distribution. The most used is called Alpine linux, this is because it's a very small distribution and it's very fast to download. So over that layer we will have many layers of images until we get to the layer of our application. So the magic of the containers are that they are very small talking in terms memory space if we compare them with actual virtual machines where the vm's can take up Gigabytes of memory and the container can take MegaBytes of memory. And why virtualization?

Why virtualization?

Docker is a way of virtualization. So let's talk about virtualization. Let's take VM as Virtual Machine. So VM is based on 3 layers

- The hardware: It is where the VM is running.
- The kernel: It is the layer that is between the hardware and the VM.It's the layer that allows the VM to run on the hardware.
- The apps: Not much to explain, this are the applications we use when we work with the VM

So when we talk about VM we virtualise The apps and the kernel. In this case the kernel could be a linux distribution, windows iso or mac os iso. And this makes that the images of the VM's size up to Gigabytes. In docker case we only virtualize the apps. So with the kernel docker uses the host kernel. So if you are executing a container in a linux host, the container will use the linux kernel of the host. Same happen with windows and MacOs.

Images commands

Let's start with the basic commands of docker. First to run the commands you need to run docker desktop running. Also to check if you have docker installed you can run the command

```
docker --version
```

Also docker recommend to run the hello-world image to check if the installation is correct. To do that you can run the command

```
docker run hello-world
```

This command will download the hello-world image and run it. If everything is correct you will see a message that says that the installation is correct.

Now our first command will be to list all the images that we have in our local machine. To do that we can run the command

```
docker images
```

Now to download our image we can run the command

```
docker pull <image-name>
```

In this case ¡image-name¿ could python for example. This will download the python image from the docker hub or Node, or mysql, or Postgres, etc. Another thing to have in consideration is that you can pull a image with a specific version. For example if you want to pull the python 3.8.5 you can run the command

```
docker pull python:3.8.5
```

But if you use the command docker pull python you will download the latest version of python.

And in the process of the pull you will see graphically how the layers of the image are being downloaded and this will help to understand that concepts.

Now in the list of images you will see the image that you just downloaded. This list have the name of repository where the image is, the tag, the image id, the created date and the size of the image. To know what images you can pull you can go to the docker hub and search for the image that you want to pull.

If you have a problem with the image that you just downloaded you can try fixing it with this command

```
docker pull --platform linux/x86_64 <image-name>
```

Now we see the command we need to run to delete the images that we don't need anymore. To do that we can run the command

```
docker image rm <image-id>
```

or

```
docker image rm <image-name>:<tag>
```

Containers commands

First, to mount a container we need a image. So you can download the image you want. Then to create a container you can run the command

```
docker create <image-name>
```

and this will return a container id, save it because you will need it to run the container.

Now to run the container that you create run the command

```
docker start <container-id>
```

and this will return the container id again. And how do i do to check if the container is running? You can run the command

```
docker ps
```

now this will return a table with valuable information. The first column is the container id, the second column is the image name, the third column is the command that the container is running, the fourth column is the creation date, the fifth column is the status of the container, the sixth column is the ports that the container is using and the last column is the name of the container.

Now to stop the container you can run the command

```
docker stop <container-id>
```

this will return the container id again and you can check if the container is stopped with the command

```
docker ps
```

To show all the containers that you have in your local machine you can run the command

```
docker ps -a
```

to delete a container you can run the command

```
docker rm <container-id>
```

Now if you want to assign a name to the container you can run the command

```
docker run --name <container-name> <image-name>
```

and with this you can run, stop and remove the container by the name that you assigned. You can do the 3 actions with the same commands that we saw before just thata change de ¡container-id¿ for the ¡container-name¿

And now we can't use this container because we don't have any port open.So we need to indicate to docker that we want to open a port but this port we need to map it to a port of our container. Because we can access to the container's port. But first let's explain what is Port Mapping

Port Mapping

The concept of port mapping is essential to understand when working with Docker. Basically is the process of forwarding network traffic from one network port to another. Imagine that we have and application that is running in the port 3000 and we have a container that is running in the port 27017. We can map the port 3000 of our local machine to the port 27017 of the container. So when we access to the port 3000 of our local machine we are accessing to the port 27017 and now we implemented the port mapping.

So to map a port you can run the command

```
docker run -p<local-port>:<container-port> <image-name>
```

we can also map the port to a specific ip address. To do that you can run the command

```
docker run -p<ip-address>:<local-port>:<container-port> <image-name>
```

also we can let docker decide the port that we are going to use. To do that you can run the command

```
docker run -p<container-port> <image-name>
```

And now we need to now if the container is being executed in the right way. To do that you can run the command

```
docker logs <container-id>
```

ando to keep tracking the logs we can run the command

```
docker logs --follow <container-id>
```

and with this command we can see the logs in real time. To exit of those logs you can press ctrl + c.

Now we are going to learn a new comman, that will be a combination of pull, create and start. This command is

```
docker run <image-name>
```

This command will pull the image if it's not in your local machine, then it will create the container and then it will start the container. And this command will show the logs of the container in real time and you can detained with the command ctrl + c but this will detained all the container. But if you want to run the container in the background you can run the command

```
docker run -d <image-name>
```

and this will return the container id. Also we can merge all the commands that we saw before in one command.For example

```
docker run -d -p<local-port>:<container-port>
--name <container-name> <image-name>
```

So as you can see we can merge all the commands in one command. And this will make the process of creating and running a container It's important to know that every time you run a container you are creating a new container.

Conecting to a container

And now we are going to learn how to connect to a container. So first we need to go Docker hub and search for the image that we want to run. So why go to docker hub and search for the image? Because in the documentation of the image we have some paremeters that we need to configure to run the container and access to it. So in this case and many of the times when we put in a container a Database we need to configure a user and a password. So we go to the documentation of the image and we search for the paremeters that we need to configure. All of the containers need a different configuration so it's important to go to the documentation of the image. So let's see an example. Let's say that we want to run mongo in a container. So we go to the documentation of the image and we see that we need to configure a user and a password. Then we can run the command

```
docker run -d -p27017:27017 -e
    MONGO_INITDB_ROOT_USERNAME=admin -e
    MONGO_INITDB_ROOT_PASSWORD=admin --name mongo
```

All in same line just that i put the command in different lines to make it more readable. So in this case we have configure our docker, now on our code we can connect to the database with the user and the password that we configure. As the language that your are using required. Because the configuration of the database is in the container and that container that we configure is up and running as an application of the database in our local machine. Now we cant put our app into a container so let's see how we can do that

Application in a container

So first we need to create a file into our project that is called Dockerfile. This file will have the configuration of our container.So the dockerfile is a file that contains the instructions to build a container. So let's see an example of a dockerfile

```
FROM node:14 // Here we indicate the image that we
    are going to use and with the version that we are
    going to use


RUN mkdir -p /home/app  // Here we create a
    directory in the container where we are going to
    put our source code of the app


COPY . /home/app // Here we copy all the files of
    our project to the directory that we create in
    the container

EXPOSE 3000 // Here we indicate the port that we are
    going to use to run our app and we expose it to
    the container

CMD ["node", "/home/app/app.js"] // Here we indicate
    the command that we are going to use to run our
    app
```

So now we have our dockerfile bur we can't run the container yet because containers can't have communication with each other. Of course they can have communication with the exterior by mapping ports. But we now have encapsulated our app into a container and we have our database in another container. So how we can make that our app can communicate with the database? We can usea internal network of docker. So between containers in the same network

they can communicate with each other. Also we can have as many networks as we want. So let's se how we can create a network and how we can put our containers in that network. we are going to use the dockerfile that we create before. So we save the dockerfile and we close it and now we run the next command

```
docker network ls
```

This will retun a table with the networks that docker has configured. Now we are going to create a network. To do that we can run the command

```
docker network create <network-name>
```

of course this will return the network id. In case we need to delete the network we can run the command

```
docker network rm <network-name>
```

So now with the network created we can put our containers in that network. But first we need to edit our source code an change the connection to the database. So we need to change the connection to the database to the name of the container. For example

```
//before
mongoose.connect('mongodb://admin:admin@localhost:27017/miapp');
//after
mongoose.connect('mongodb://admin:admin@mongo:27017/miapp');
```

So now we can run the command

```
docker build -t <name:tag> <app-route>
```

"docker build" is use to build a image based on a dockerfile. So as you see docker builde requires two parameters. First is the name and the tag that we are want to put it and the second is the route of the app. That route we can indicated with a dot if you are in the route of the app. You can check if the image was created with the command

```
docker images
```

and now we can give the container our network. To do that we can run the command

```
docker create -p3000:3000 --name <container-name>
    --network <network-name> <image-name> -e
    <Environment variables>
```

and to connect the container with another in same network we can run the command

```
docker create -p27017:27017 --name <container-name>
    --network <network-name> <image-name:tag>
```

This section is more difficult to understand but it's very important to understand it. So if you didn't understand it you can go to this link and see this video `https://www.youtube.com/watch?v=4Dko5W96WHg&t=3831s`. And now let's see the Docker Compose

Docker Compose

Docker compose is a tool that allows you to optimize the process of creating and running containers. Because as you can see all the commands that we saw before are very long and we need to run a lot of commands to run a container. So with docker compose we can run all the commands in one command. . First we need to create the file that need to be call docker-compose.yml. And then we can put the next code in the file. So let's see an example of a docker-compose file

```
version: '3.1' // Here we indicate the version of
    the docker-compose file

services: // Here we indicate the services that we
    are going to use
     mongo: // Here we indicate the name of the
         service
          image: mongo // Here we indicate the image
              that we are going to use
          ports: // Here we indicate the ports that we
              are going to use
              - "27017:27017"
          environment: // Here we indicate the
              environment variables that we are going
              to use
               MONGO_INITDB_ROOT_USERNAME: admin
               MONGO_INITDB_ROOT_PASSWORD: admin
     app: // Here we indicate the name of the service
          build: . // Here we indicate the route of
              the app

          ports: // Here we indicate the ports that we
              are going to use
              - "3000:3000"

          links:
           - mongo
```

and now we save the file and we can run the command

```
docker-compose up
```

This will return the logs of the containers in real time. And to stop the containers you can press ctrl + c. And now let's see if this create us a new image. To do that you can run the command

```
docker images
```

and to see if this create us a new container. And now we can run the command

```
docker ps -a
```

and see that this create us a new containers and a new image. And both of this containers were created in the command line that we run before. And if we want to re-run the containers we can run the command

```
docker-compose up
```

and this will use the same containers that it created before. But what happen if we want to delete this containers and the image in a fast way. Well we can run the command

```
docker-compose down
```

Important thing that docker do is that when you run the command docker-compose up it will create a new image, a new container and a new network because if you code two services docker thinks that you want to communicate them so it creates the network, and with docker compose down it will delete the containers, the image and the network. So now let's continue with the volumes.

Volumes

So as you have seen when we run a container and we delete it we lose all the data that we have in the container. This is because when we create a container it has his own operating system and his own file system and into this file system is where all the data is saved. So if we want to mantein all of our data we need the tool of volumes. So volumes part from the file system of the container, so what this tool do is to mount that directory where de data is saved in the container moved to a directory in our local machine. So when you delete a container you don't lose the data. So there are three types of volumes

- Anonimous volumes: Thiis volume is where you only specify the route where you wanto to save the data in your local machine
- Host volumes: This the volume where you decide in which and where directory you want to save the data in your local machine
- Named volumes: This volume is like the anonimous volume but you can reference this volume when you create a new container

To star working with volumes we need to acces to the docker-compose file and add the next code

```
version: '3.1'

services:
    mongo:
        image: mongo
        ports:
            - "27017:27017"
```

```
        environment:
            MONGO_INITDB_ROOT_USERNAME: admin
            MONGO_INITDB_ROOT_PASSWORD: admin
        volumes: // Here we indicate the volumes
            that we are going to use
            -mongo-data:/data/db //Here we indicate
                the volume that we are going to use
                and indicate where the data is save
                in the container

    app:
        build: .

        ports:
            - "3000:3000"

        links:
         - mongo
volumes: // Here we indicate the volumes
    mongo-data: // Here we indicate the name of the
        volume
```

now we saved the file and we can run the command

```
docker-compose up
```

and check if the app is working properly. And that's how we can delete containers and create new containers and don't lose the data.

Environments and Hot Reload

Now we are going to learn how to configure differents environment with docker. Because we don't have the same environment in development than in production. So we need to create a docker file for development and another for production. So let's see how we can do that.

So you for production you can use the same docker file that we saw before. But for development you can use the next docker file

```
Dockerfile.dev
```

so now we can star editing this docker file. So first we are goint to recycle the code that we have in our dockerfile of production. So let's see an example of a docker file for development

```
FROM node:14

RUN npm i -g nodemon // Here we install this tool to
    detected the changes in the code
```

```
RUN mkdir -p /home/app

WORKDIR /home/app // Here we indicate the directory
    where we are going to work


EXPOSE 3000

CMD ["nodemon", "app.js"]
```

So the changes that we made were install nodemon to detected the changes in the code, specify the directory where we are going to work on and delete the copy of the files of the project because we will work with volumes. After all that we change the command to run the app and delete the route of where the files are, because we specify the directory where we are going to work on with the "WORKDIR" instruction. After that we save the file and create a new docker-compose file. This file will be called

```
docker-compose.dev.yml
```

and now we start editing the code. So we copy the code of the docker-compose file of production and paste it in the docker-compose file of development. That because we are going to use it as a template. So let's see an example of our docker-compose file of development

```
version: '3.1'

services:
    mongo:
        image: mongo
        ports:
            - "27017:27017"
        environment:
            MONGO_INITDB_ROOT_USERNAME: admin
            MONGO_INITDB_ROOT_PASSWORD: admin
        volumes:
            -mongo-data:/data/db

    app:
        build:
            context: . // Here we indicate the
                context of the build or where is the
                app
            dockerfile: Dockerfile.dev // Here we
                indicate the docker file that we are
                going to use
        ports:
```

```
            - "3000:3000"
        volumes:
            - .:/home/app // Here we indicate the
                volume that we are going to use, this
                volume is anonimous
        links:
         - mongo
volumes:
    mongo-data:
```

So the changes that we made were to indicate the context of the build or where is the app and indicate the docker file that we are going to use. Also indicate the volumen that we are going to use and in this case we use an anonimous volume. After that we save the file and can run the command

```
docker-compose -f docker-compose.dev.yml up
```

so as you can see we use the flag -f to indicate the file that we are going to use. And now we can check if the app is working properly.

Conclusion

So we finally finish the docker documentation. I hope that you understand all the concepts that we saw in this documentation. If some of the concepts you didn't understand you can go to this link and see this video `https://www.youtube.com/watch?v=4Dko5W96WHg&t=3831s`. And now you can start working with docker

- **Email:** `aleleonmarin01@gmail.com`
- **LinkedIn:** `Alejandro Leon Marin`
- **Discord:** `LionKing#6730`
- **Twitter:** `@aleleonmarin`
- **Instagram:** `@_ale.leon.marin_`
- **Facebook:** `Alejandro Leon Marin`
- **Github:** `aleleonmarin`