Docker

2024-2025

Junia-ISEN - M1

Contents

Introduction

Imagine you are an young system admin and you want to install various tools to work on different workflows and different self-hosted websites for your company. You have one ticketing tool, one RSS flux reader, a drive, a calendar... Behind those tools, you need to install packages (PHP, postgresql, mariadb, apache, nginx...). Unfortunately, one tool requires the version 7 of PHP and one tool requires the version 8. How do you cope with this situation?

One would think that you install both version on the same system but it is far from trivial. You could separate the tools on different machines but for cost efficiency, you only have one machine. Virtualization is then the way to go.

Then you could install a virtual machine (VM) for the tool requiring PHP 8 and the rest is installed on the other. That would be a little bit overkill to install one VM for only one tool. Then, you look at what could be the alternatives. Fortunately, modern computer technologies based on new Linux kernel features enable to create containers. One of those tools is called Docker.

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allows you to run many containers simultaneously on a given host.

Containers are lightweight and contain everything needed to run the application, so you do not need to rely on what is currently installed on the host. Containers use the same kernel as the host and do not need an additional OS installed in a conventional way to run.

Think of containers being similar to a virtual machine but only for the applications.

They rely on the concepts of *namespaces* and *cgroups*.

1 Characteristics

A few key points in order to see why docker is an interesting virtualization tool compared to other systems.

Host processes

- Isolate address space
- No isolation for files or networks
- Lightweight

Virtualization Machines (VMWare, Virtual Box...)

- Isolate address space
- Isolate files and networks
- Heavyweight

Containers

- Isolate address space
- Isolate files and networks
- Lightweight

2 VM vs Docker

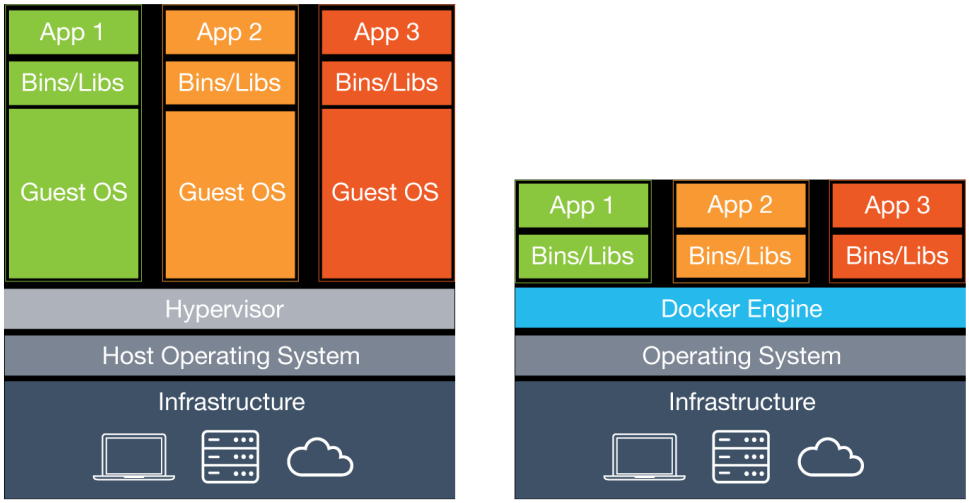

Figure 1: Differences between a Virtual Machine (VM) and the Docker engine

As you can see on this image. One of the main difference between the docker engine and a virtual machine is the presence of a second host (and hypervisor) in the case of VM. It means, that Docker will use the same kernel as the host. The virtualization in the case of docker is therefore minimal.

2.1 In a nutshell, Docker and containerization in general are:

- New way of designing systems
- Modern approach to thinking about network architectures
- Faster and simpler deployments
- Abstraction of virtualization components

3 How is it Working?

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers.

The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface. Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.

3.1 Features

3.1.1 The Docker daemon

The Docker daemon (dockerd) receives Docker API requests and oversees various Docker entities, including images, containers, networks, and volumes. It can also establish communication with other daemons to handle Docker services effectively.

3.1.2 The Docker client

The Docker client (docker) serves as the main interaction tool for numerous Docker users. Interactions like utilizing "docker run" commands result in the client transmitting these instructions to dockerd, which executes them. The docker command operates using the Docker API and is capable of interfacing with multiple daemons.

3.1.3 Docker Desktop

Docker Desktop, an easily installable application for Mac and Windows, facilitates the creation and sharing of containerized applications and microservices. It encompasses components such as the Docker daemon (dockerd), Docker client (docker), Docker Compose, Docker Content Trust, Kubernetes... Despite Windows users having access to the graphical interface, we'll focus on the terminal for a more comprehensive understanding.

3.1.4 Docker registries

A Docker registry serves as a repository for Docker images. Docker Hub, a public registry, is the default location for image searches. Private registries can also be employed. The `docker pull` and `docker run` commands retrieve images from the configured registry, while `docker push` uploads images to the chosen registry.

3.2 Docker objects

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief overview of some of those objects.

3.2.1 Images

An image functions as a read-only template with instructions for generating a Docker container. Typically, images are based on existing ones with additional customizations (refer to Dockerfile). For instance, you can create an image derived from the `ubuntu` image that includes Apache web server, your application, and configuration specifics. You may produce your images or utilize those crafted by others available in a registry. Crafting an image involves generating a Dockerfile that outlines the steps for creating and executing the image. Each Dockerfile directive forms a layer within the image. When modifying the Dockerfile and reconstructing the image, only altered layers are rebuilt. This contributes to the lightweight, small, and speedy nature of Docker images, differentiating them from other virtualization methods.

3.2.2 Containers

A container is a runnable iteration of an image. Using the Docker API or CLI, you can generate, initiate, halt, shift, or eliminate a container. Networking can link a container to one or more networks, while storage can be attached, and even a new image can be derived from its present state. Ordinarily, a container exhibits a degree of isolation from other containers and the host machine. The extent of isolation, whether concerning network, storage, or other subsystems, can be regulated to vary from the host machine or other containers. The image and any configuration particulars supplied upon container creation or launch define its attributes. Once a container is deleted, any non-persistently stored alterations to its state vanish.

3.2.3 Networks

Docker's networking system is modular and supports various drivers. Several built-in drivers offer core networking capabilities:

`bridge`: This is the default driver for networks. If you don't specify a driver, it's the one used. Bridge networks are useful for standalone container applications that require communication.

`host`: For standalone containers, this eliminates network separation between the container and the Docker host, utilizing the host's networking directly.

`overlay`: Overlay networks link multiple Docker daemons and enable swarm services to communicate. They facilitate interaction between swarm services and standalone containers, removing the need for OS-level routing.

`ipvlan`: IPvlan networks provide full control over IPv4 and IPv6 addressing. The VLAN driver extends this by offering complete control of layer 2 VLAN tagging and IPvlan L3 routing for underlay network integration.

`macvlan`: Macvlan networks assign a MAC address to a container, simulating a physical device on the network. Traffic is routed to containers based on their MAC addresses. This

driver is suitable for legacy applications expecting direct connection to the physical network, bypassing the Docker host's network stack.

`none`: For this container, disable all networking. Usually used in conjunction with a custom network driver. none is not available for swarm services.

`Network plugins`: You can install and use third-party network plugins with Docker. These plugins are available from Docker Hub or from third-party vendors.

3.2.4 Volumes

Volumes are the recommended approach for preserving data produced and utilized by Docker containers. Unlike bind mounts, which rely on the host machine's directory structure and operating system, volumes are entirely overseen by Docker. Furthermore, volumes often present a superior alternative to persisting data within a container's writable layer. Unlike such an approach, a volume doesn't augment the size of containers leveraging it, and the volume's content remains unaffected by the lifecycle of any particular container.

Volumes are logical type of storage that are a small and extensible part of the physical hard-drives directly managed by Docker. You should be aware that volumes and bind mounts are two different approach for the same thing.

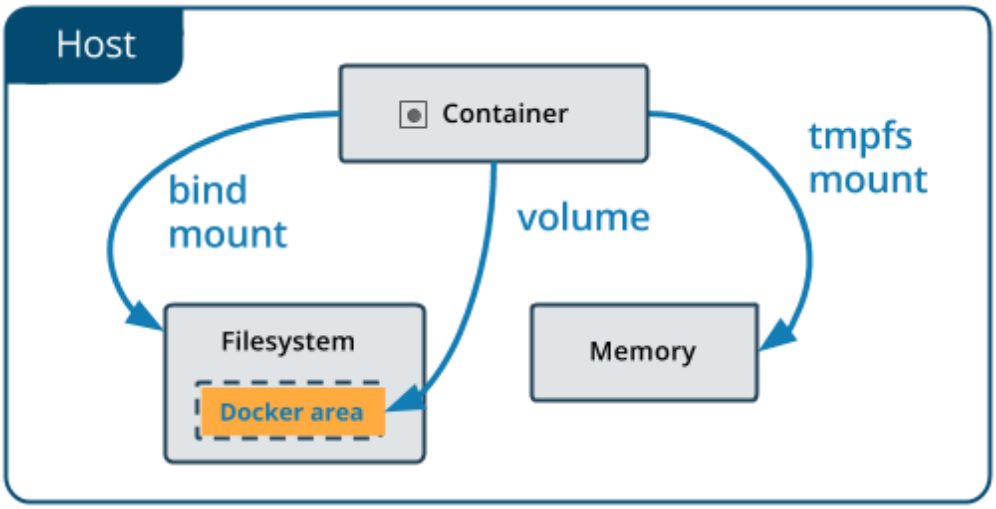

Figure 2: Types of volumes and difference between Volumes and Bind mount Volumes

Info. Try to understand the different between *Volumes* and *Bind mounts*. We will need that later in the labs.

4 Installing Docker

4.1 If you are using Windows as a main OS

As Docker needs the features of a Linux kernel to run (*namespaces* and *cgroups*), then we will install a real Linux distribution first. For that, we will either install a dual boot (your job to do it, you can ask the C2I), or install it through a virtualization software like VirtualBox or VMWare Player

Once the virtualization tool is installed, you can download an ISO file of your Linux distribution of your choice (ubuntu is usually a safe choice). Ask the teacher if you are unsure what to do.

Depending on the distribution you are on, please follow the steps below to install Docker.

4.2 If you are using Linux as a main OS

If you are on Ubuntu

1. Update your system and install required dependencies:

```
sudo apt-get update
sudo apt-get install \
  ca-certificates \
  curl \
  gnupg \
  lsb-release
```

2. Add Docker's official GPG key:

```
sudo mkdir -p /etc/apt/keyrings

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
```

3. Use the following command to set up the repository:

```
echo \
  "deb [arch=$(dpkg --print-architecture) \
  signed-by=/etc/apt/keyrings/docker.gpg] \
  https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

4. Install the docker engine:

```
sudo apt-get update

```

```
    sudo apt-get install docker-ce docker-ce-cli containerd.io \
    docker-compose-plugin
```

Info Receiving a GPG error when running apt-get update?. Your default umask may not be set correctly, causing the public key file for the repo to not be detected. Run the following command and then try to update your repo again: `sudo chmod a+r /etc/apt/keyrings/docker.gpg`

5. Verify that Docker Engine is installed correctly by running the hello-world image. `sudo docker run hello-world`

This command downloads a test image and runs it in a container. When the container runs, it prints a message and exits.

If you are using Fedora

Install the dnf-plugins-core package (which provides the commands to manage your DNF repositories) and set up the repository.

```
    sudo dnf -y install dnf-plugins-core
    sudo dnf config-manager --add-repo \
    https://download.docker.com/linux/fedora/docker-ce.repo
```

Install the latest version of Docker Engine, containerd, and Docker Compose.

```
    sudo dnf install docker-ce docker-ce-cli containerd.io \
    docker-compose-plugin
```

If prompted to accept the GPG key, verify that the fingerprint matches **060A 61C5 1B55 8A7F 742B 77AA C52F EB6B 621E 9F35**, and if so, accept it.

This command installs Docker, but it doesn't start Docker. It also creates a docker group, however, it doesn't add any users to the group by default. Follow the steps below to add users to the docker group.

Start Docker: `sudo systemctl start docker`

Verify that Docker Engine is installed correctly by running the hello-world image.

`sudo docker run hello-world`

This command downloads a test image and runs it in a container. When the container runs, it prints a message and exits.

If you are using Arch

First update your system : `sudo pacman -Syu`

Then the following command should install all the necessary files: `sudo pacman -S docker`

Follow the additional steps below to add your user to the docker group.

Verify that Docker Engine is installed correctly by running the hello-world image: `sudo docker run hello-world`

4.2.1 Additional steps post-install on Linux

To create the docker group and add your user

If you receive the error:

`Got permission denied while trying to connect to the Docker daemon socket`.

This means that your user is not allowed to run the docker daemon. You will need to add the current user to the docker group.

- First add the docker group if it does not exist: `sudo groupadd docker`
- Then add your user to the docker group: `sudo usermod -aG docker $USER`

Log out and log back in so that your group membership is re-evaluated.

Configure Docker to start on boot

In order to start the docker daemon on boot, use the following commands:

`sudo systemctl enable docker` and `sudo systemctl enable containerd`

5 Labs

The labs are exercises to get familiar with the Docker concept. You will try to understand how to run, create and operate images and containers. Questions are going to be asked in the labs. You will create a report with answers to the questions. This report will serve you as a learning document for the exam. You have an additional assignment (project) where you have to build, create and run a set of containers using a dockerfile and docker-compose.

You will work with groups of 2 or 3 students.

Info. If you ever feel lost with the commands, don't forget that you can use `docker --help` or `docker OPTION --help`, where OPTION has to be replaced by the option you want.

5.1 Starting our first container

To start a container, run the command `docker run hello-world`

`hello-world` is the name of the image. It should print a message and exits.

Now, let's run a more sophisticated container :

`docker pull ubuntu` and then `docker run ubuntu`

Question 5.1.
- What is happening? Do you see something?
- Run the command `docker images`. What do you see?
- With the command `docker ps -a`, what does the command do?

Let's dig a bit in a ubuntu container. Run the command:

`docker run -it ubuntu /bin/bash`

Using this command, we have created a second `ubuntu` container. The first one still exist and we have entered the second one (see figure 3).

Question 5.2. What's the `-it` and `/bin/bash` doing?

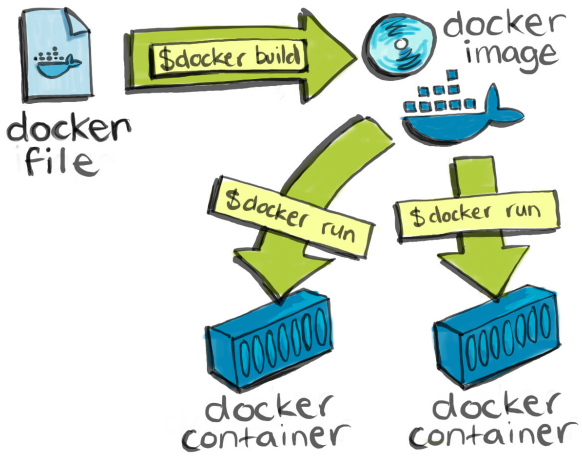

Figure 3: A dockerfile builds an image and a container is created by running an image

Question 5.3.

- Review the folders in the container. What can you say about the folder structure? Compare it with a standard Linux distribution.
- Go to the `/media` and create a folder `mkdir folder`. Then exit the container and go back inside the same container. Do you find the folder `/media/folder` you have created?
- Now, create a new container based on the same image, what happened to the `/media/folder`?
- As we won't need it anymore. Learn how to delete an image : remove the `hello-world` images from your computer

Before going into more details with the features of Docker. Let's stop and think a little bit on this image.

Question 5.4.

- From this image, if we run twice the image, is the two containers identical?
- What if we modify something inside a container, exit the container and run again the image. Do you think the modification will transfer to the newly created container?
- What do you think is missing at this point?

5.2 Volumes

With the previous exercise, we saw that each container starts from the image definition each time it starts. While containers can create, update, and delete files, those changes are lost when the container is removed and all changes are isolated to that container. With volumes, we can change all of this. Imagine that using your computer, each time you boot, you lost all

the computer program that you wrote the day before. That would be unpractical! Therefore we need a mean to store data. On a typical computer, we have physical hard drives that can be segmented into logical ones. For instance, one hard drive appears as 2 (or more) drives on your operating system. Here the concept is similar, we will allow an amount of physical space to a *volume* and this volume will be attached to a container.

Question 5.5. Type the command `docker volume create ubuntu-folder`

Now that the volume is created, we need to attach the volume to the container.

Question 5.6. Using the `docker` command and the manual, try to attach the volume you just created to an ubuntu container and run it interactively.

Exit the container and run the command `docker volume inspect ubuntu-folder`. The `Mountpoint` is the actual location on the disk where the data is stored. Note that on most machines, you will need to have root access to access this directory from the host.

Question 5.7.

- Try to bind-mount **an existing folder** on your computer to the container.
- Try to modify a file from inside the container and try to modify the same file outside the container. Do you see the changes on either sides?
- Try to import a file from a bind mount and upload it to the *volume* to make it persistent.
- To your knowledge, which type of storage is the safest option in term of cyber-security here and why?

5.3 Networks

Containers are isolated environments, but by default, a bridge connection is created by Docker between the host and the container. Therefore, we have access to the internet inside of our container. However, the interface is created "randomly" but we can still have access afterward. The idea here is to understand how to create a network and have more control over it.

Question 5.8.

- Run interactively once again the `ubuntu` image.
- Install `net-tools` and `iputils-ping` inside the container (you need to use the command `apt-get update` first, and then `apt-get install net-tools iputils-ping`).
- Run the command `ifconfig -a`. What can you say on the interfaces?
- In a similar fashion you created a volume, try to create a network and change the default network of the ubuntu container (hint: use the `subnets` option).
- Specify an IP address for the container that you connect to the network.
- Run a second ubuntu container within the same network as the first one. How can you check that the two containers can communicate (show an example in your report)?
- Let us say that we have a park of containers with multiple networks available. Draw a diagram and assign all the containers to a network. Put the details about the network's name, IP range, container's name and IP, if they can communicate between each other etc...

5.4 namespaces

Now, let's take our head out of the containers and docker. The *Volumes* and *Networks* sections were good introductions to understand the concept of *namespaces*. *namespaces* as a Linux kernel feature, provide a powerful mechanism for process isolation, resource management, and the creation of distinct environments within a single operating system instance. These resources include **processes**, **network interfaces**, **filesystem mounts**, **devices**, and more. Think of namespaces as a tool that empowers us to create separate "worlds" within the same operating system, each with its own set of resources and characteristics.

5.4.1 The Why and How of Linux Kernel Namespaces

- **Process Isolation and Resource Separation**: Imagine running multiple applications on a single machine. Without namespaces, these applications could inadvertently interfere with each other, leading to instability and security risks. Namespaces, however, provide a way to isolate processes and allocate resources as if they were running on separate systems altogether.
- **Containers and Virtualization**: Namespaces form the foundation for containerization technologies like Docker and Kubernetes. Containers encapsulate applications and their dependencies, providing a lightweight and consistent environment. Namespaces ensure that each container has its own isolated view of system resources.
- **Namespace Types**: The Linux kernel supports various types of namespaces, each focusing on a specific set of resources. These include process namespaces (PID), network namespaces (NET), mount namespaces (MNT), and more. Each type of namespace allows processes to have their own isolated instances of the corresponding resource.

- **Usage in System Calls**: Linux system calls like clone() and unshare() enable the creation and manipulation of namespaces. The clone() call, for example, can be used to create a new process with its own namespace, allowing the process to operate within a distinct environment.

Question 5.9.
- Using the terminal, type the command `sudo unshare --pid --fork --mount-proc /bin/bash`. Try to understand what this command is doing (type of namespace for example)?
- How can you identify if you are in an isolated environment or not?
- Considering that a Linux operating system **NEEDS** a PID starting at **1** (this is where the kernel start the initialization of the system), explain the role of this namespace in the creation of containers?
- Use a diagram to draw a PID tree of the host system and the new created isolated tree of the process namespace.

Question 5.10.
- Using the terminal, type the command `sudo unshare --net /bin/bash`. Try to understand what this command is doing (type of namespace)?
- How can you identify interfaces available to you?
- Is it possible to communicate to the outside from where you are? Explain.
- Use a diagram to draw what you understand of this namespace.

Question 5.11. With your knowledge of namespace now, list all the namespaces that exist inside a container (try to think of all the possible abstractions that exist, we saw process and network... What else?) and give a definition for all of them.

5.5 Dockerfile

The dockerfile section explains how to create yourself you own images. Remember that each image creates a container and each image is created using a dockerfile.

- Let's create the index.html file

```
<h1>hello world</h1>
```

- Create file named Dockerfile and put it in the same folder as index.html

```
#Base image
FROM nginx:1.21-alpine
COPY . /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Question 5.12. Explain using simple words what this file is doing? (Explain each command)

- Create docker image

To build the image with the dockerfile, use the command `docker build -t html-hello-world:v1 .`

Question 5.13. Explain what the `-t` and the `.` means

- Run docker image

You can run your image using : `docker run -p 8080:80 html-hello-world:v1`

Question 5.14. `-p` is used with port numbers. Explain what this is doing?

- Open application on the web browser

Open your web browser and open http://localhost:8080 this will open up the HTML page you have created.

Question 5.15.

- Run a second instance of the html-hello-world. What error do you get? How to correct it?
- Replace, the `COPY` by `ADD` and rebuild the image (call it `html-hello-world:v2`). Do you see any differences?
- Now try instead of `ADD . /usr/share/nginx/html`, try the command:
  - `ADD https://cloud.capiod.fr/s/2LA9GpxjWkz9NGm/download/index.html /usr/share/nginx/html`
  - `RUN chmod 755 /usr/share/nginx/html/index.html`

  What do you see? Try the same with `COPY`, is it working? Try to explain why!

5.5.1 A more complex dockerfile

Here is an example of an Ubuntu dockerfile :

```
1  # Ubuntu Dockerfile
2  #
3  # https://github.com/dockerfile/ubuntu
4  #
5
6  # Pull base image.
7  FROM ubuntu:14.04
8
```

```
# Install.
RUN \
  sed -i 's/# \(.*multiverse$\)/\1/g' /etc/apt/sources.list && \
  apt-get update && \
  apt-get -y upgrade && \
  apt-get install -y build-essential && \
  apt-get install -y software-properties-common && \
  apt-get install -y byobu curl git htop man unzip vim wget && \
  rm -rf /var/lib/apt/lists/*

# Add files.
ADD root/.bashrc /root/.bashrc
ADD root/.gitconfig /root/.gitconfig
ADD root/.scripts /root/.scripts

???
ENV HOME /root

???
WORKDIR /root

# Define default command.
CMD ["bash"]
```

Info. We will use already made files (Dockerfile + configuration files of the above dockerfile) that you will download at the following link https://cloud.capiod.fr/s/H4cSE3TcZ3q9xWM.

Question 5.16.

- Explain the steps that the docker is doing. Some additional command are marked by `???` which, we did not use above. With your knowledge, try to explain those commands.
- It is possible to "enforce" the execution of commands or scripts with the command `ENTRYPOINT`. Create a small script (ask for help if don't know how to) that mimic the line `CMD ["bash"]`
- Using the link above and all the files in the archive, build the dockerfile to create an image `ubuntu-junia:v1`.

5.6 Docker layers and history

A Docker image consists of several layers. Each layer corresponds to certain instructions in your Dockerfile. The following instructions create a layer: `RUN`, `COPY`, `ADD`

In order to review the layers, you need to know the image ID. You can do `docker images`.

Question 5.17.

- Find the image ID of your freshly created `html-hello-world:v1` and `ubuntu-junia:v1` and review the history of it using `docker history IMAGEID`.
- What can you see? What are the differences between the history and the dockerfile you used?
- Edit the docker file (of `html-hello-world:v1`) and add an instruction to update the alpine system using `apk update`. Be aware that you need to write it using the dockerfile syntax.
- Build the image and call it `html-hello-world:v2`. Review the history. What are the differences?

5.7 Docker-compose

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.

Create a file named `docker-compose.yml` and copy/paste the following content (don't forget to download the zip package below):

```
version: "3.7"      # version of the docker-compose file (it should be
    omitted with the new versions)
services:
    random:       # name of the container
        build: app    # ???
        labels:      # ???
            role: backend
        ports:      # specify used ports
        - 3000
        networks:     # specify the network for this container
        - lab
networks:      # ???
    lab:
```

Unzip the following archive at https://cloud.capiod.fr/s/xrx7Z2G4DHELLLo to the same folder as the `docker-compose.yml` Then you run the docker-compose using : `docker-compose -p dcomp1 up -d`. The option `-d` is to detach the containers, so they can run in the background. Additionally we named the container `dcomp1` with the `-p` option.

Question 5.18.

- In your report, copy the `docker-compose.yml` file content and fill the `???`
- Review the `app` folder, what's inside and what the dockerfile do.

Now let's run a docker-compose with multiple containers at once:

```yaml
version: "3.7"
services:
    red:     # name of the container 1
        build: app
        environment:    # environment variable to pass to the container
            when created
            COLOR: "red"
        labels:
            role: backend
        ports:
        - 3000
        networks:
        - lab
    green:     # name of the container 2
        build: app
        environment:    # environment variable to pass to the container
            when created
            COLOR: "green"
        labels:
            role: backend
        ports:
        - 3000
        networks:
        - lab
    white:     # name of the container 3
        build: app
        environment:    # environment variable to pass to the container
            when created
            COLOR: "white"
        labels:
            role: backend
        ports:
        - 3000
        networks:
        - lab
networks:
    lab:
```

Run the new docker-compose file using `docker-compose -p dcomp2 up -d`. Using the command `docker-compose -p dcomp2 ps` you can view all the running container of the `dcomp2` project.

Question 5.19.

- Use the command `curl 0.0.0.0:PORT/text` (`PORT` has to be changed). What does the command do? What information do you have?
- What option could you add in the docker-compose file to set the desired port number outside and inside the container?

5.8 Let's go a little further (if you have time)

Let's add a load-balancer service to our multicolor applications in a new docker-compose.yml:

```
version: "3.7"
services:
    loadbalancer:           # Here is the new container that we will run
        in addition to the three we did before
        build: lb
        environment:
            APPLICATION_PORT: 3000
        ports:            # We wire the port 8080 outside the container to
            the port 80 inside the container
        - 8080:80
        networks:
        - lab
    red:
        build: app
        environment:
            COLOR: "red"
        labels:
            role: backend
        ports:
        - 3000
        networks:
        - lab
    green:
        build: app
        environment:
            COLOR: "green"
        labels:
            role: backend
        ports:
        - 3000
        networks:
        - lab
    white:
        build: app
        environment:
            COLOR: "white"
        labels:
```

```
36                role: backend
37            ports:
38            - 3000
39            networks:
40            - lab
41    networks:
42        lab:
```

Question 5.20.

- Run this docker-compose file, name the project `dcomp3`
- Try to understand what a loadbalancer do
- Review the `nginx.conf` file in the `lb` folder. What can you see?
- What does the command `curl -H "Host: white" 0.0.0.0:8080/text` do? Try to review each element of the command
- Rewrite the previous command with the other hosts (green and red). Is the loadbalancer doing what it is supposed to do?
- Run the command `docker-compose -p dcomp3 up -d --scale green=4`. We will run the green container 4 times
- `curl` the green host several times. What do you see?
- Try to think of a server application where a loadbalancer is useful.

6 Assignments

For the assignment, you will be actively creating it. Try to imagine a real scenario where you need to build a computer system based on services. This can be whatever you want (Web server, development framework, a complete virtual machine based on containers...). The evaluation will be on the complexity and the realization of the system. Of course, try to use what you learned during the labs! For instance, for a complete system with multiple containers, try to use `docker-compose`. If an image does not exist for what you want to use, create it using a dockerfile... Last but not least, provide a drawing of the system you want to build with as much details as possible.