

# Inception Tutorial



## Learning objectives

1. Understanding **Docker** and **Compose** .
2. Discover the different points of the **subject** from Inception.
3. Set up the container**NGINX**
4. Set up the container**MariaDB**
5. Set up the container**WordPress**
6. Connect the containers with**Compound**
7. Understanding the volumes of**Compound**
- 8.Finalizethe project

## 1. Understanding Docker :

📌 The advantage of Docker is clear, it's **solves** one of the biggest **problems for developers** :

👉 Finding yourself creating a great program on your computer, and realizing that it only works on your own computer. To use it elsewhere you will have to install the required dependencies 🎉

<input checked="" type="checkbox"/> Docker	<input type="checkbox"/> Subject	<input type="checkbox"/> NGINX	<input type="checkbox"/> MariaDB	<input type="checkbox"/> WordPress	<input type="checkbox"/> Compound	<input type="checkbox"/> Volumes	<input type="checkbox"/> Finalize the project	<input type="checkbox"/>
--	----------------------------------	--------------------------------	----------------------------------	------------------------------------	-----------------------------------	----------------------------------	---	--------------------------

👉 So yes, you can stay **a good developer** and come up with a great **script that will install those dependencies**, but you can't predict that the user is on Mac, Linux, or has an OS version so old that it doesn't even know about your dependencies!

#### ▼ ⓘ The types of problems that Docker fixes

- A dependency is not compatible with your software version 😬
- You already have the dependency but in a different version 😐
- Your dependency does not exist on your OS 🤯
- Your addiction crashes on launch 😵
- etc...

| Spend 4 hours on **debug** of software that is not ours and realizing that we will not succeed, tends to drive us crazy 😤

This is what must have happened to **Solomon Hykes**, a French-American who ended up wondering if it was possible to find a solution to this kind of problem.  
In response to this, he comes out Docker March 20, 2013.

⌚ Reminder from the Docker Wiki : **Docker is a tool that can package an application and its dependencies into an isolated container.**

#### ▼ ⓘ The History of Docker

- Docker was developed for an internal project of dotCloud, a French company 😊 offering a platform as a service .
- Docker is an evolution based on dotCloud's proprietary technologies, themselves built on open source projects 🔒
- Docker is distributed as an open source project ❤️ as of March 2013.
- It is currently the most widely used containerization engine 🚀
- In November 2019, Docker sold its *Docker Enterprise* product to Mirantis , raising nearly \$ 35 million in funding 😱

This all sounds great, but in fact, if it already exists with VMs ,

**Why do developers use Docker? 🤔**

The great advantage of **Docker** is the ability to model each container as an image that can be stored locally.

🔍 A container is a virtual machine without a kernel.

↗ What I call the kernel is the entire system that allows the virtual machine to function, the OS, the graphics side, network, etc.

🔍 In other words, a container contains only the application and the application's dependencies .

### Docker Hub:

**Docker** provides a kind of App Store, containing images (container) of thousands of people, simplifying its use even more 👍

Docker	Subject	NGINX	MariaDB	WordPress	Compound	Volumes	Finalize the project
the wrong dependencies:							

🤔 We would need the Docker container which installs NGINX by itself.

😊 That's good, since it is known that the NGINX image has been published by **NGINX on the Docker Hub!**



Let's look at an example of what an NGINX image might look like:

```
FROM alpine:3.12
RUN apk update && apk upgrade && apk add \
    openssl \
    nginx \
    curl \
    vim \
    sudo
RUN rm -f /etc/nginx/nginx.conf
COPY ./config/nginx.conf /etc/nginx/nginx.conf
COPY scripts/setup_nginx.sh /setup_nginx.sh
RUN chmod -R +x /setup_nginx.sh
EXPOSE 443
ENTRYPOINT ["sh", "setup_nginx.sh"]
```

👉 This file is a **Dockerfile**, it is the name of the main file of your images**Docker**. Who says **Dockerfile**, says new programming language, but don't run away, it's about knowing these few key words.

## Some Dockerfile keywords :

### ▼ FROM

Allows you to tell Docker which OS your virtual machine should run on.

This is the first keyword in your Dockerfile and is mandatory.

The most common are *debian:buster* for **Debian** or *alpine:x:xx* for **Linux**.

### ▼ RUN

Allows you to run a command on your virtual machine

💡 The equivalent of logging in via ssh, then typing a bash command, like: `echo "Hello World!"`, which will display....

In general, the first **RUN** provided in the Dockerfile consist of updating your VM's resources, such as apk, or adding basic utilities like vim , curl or sudo .

### ▼ COPY

There you have it! This indeed allows you to copy a file.



A docker image is a folder, it necessarily contains your **Dockerfile** at the root of the folder but can also contain a bunch of other files so you can then copy them directly into your VM.

## ▼ EXPOSED

Here it's a question of network

The instruction **EXPOSED** informs Docker that the container is listening on the specified network ports at runtime. **EXPOSED** does not make container ports accessible to the host.

Wait! What? The container is listening on the network port and is not accessible to the host?

What does this mean?

The instruction **EXPOSED** exposes the specified port and makes it available only for inter-container communication. Let's understand this with an example.

Let's say we have two containers, a wordpress application and a mariadb server. Our wordpress application needs to communicate with the mariadb server for several reasons.

In order for the WordPress application to talk to the MariaDB server, the WordPress container needs to expose the port. Take a look at the Dockerfile of the official wordpress image and you will see a line saying **EXPOSED 3306**. This is what helps the two containers communicate with each other.

So when your WordPress container tries to connect to the port **3306** from the MariaDB container, this is the instruction **EXPOSED** who makes this possible.

Note: For the WordPress server to communicate with the MariaDB container, it is important that both containers are running in the same docker network.

## ▼ ENTRYPOINT

Yay! Your container looks ready to go.

However, it would probably be more judicious to ask the container to launch a certain command when it is launched.

This is what the keyword allows you to do **ENTRYPOINT**!

Simply state your command, argument by argument, in the following format:

```
| ENTRYPOINT "bash", "-c", """$(curlhttps://grademe.fr)"""]
```

Here is a site detailing the different types of keywords that can be used in Docker.

| To continue you must have understood the principle of Docker.

## Docker-Compose:

Now that you understand the real usefulness of Docker, it is a matter of understanding a functionality of **Docker** called Compose.

| Here is what the Docker docs explain about Compose:

| **Docker Compose** is a tool that was developed to help define and share multi-container applications.

 **Compound** would therefore allow to manage applications which use several containers and to communicate between them.

 You also need to make a *Makefile* which should be located at the root of your directory. It should allow you to set up your entire application (i.e. build the Docker images via *docker-compose.yml*)

 A text on a blue background like this is always taken from the subject **of Inception** from 42.

Here the *subject* request to build the different images via a file **.yml**, that's good, this kind of file is called **YAML**, this is the famous format indicated above by the Docker Doc to use **Docker-Compose**.

 This project will consist of you setting up a mini-infrastructure of different services following specific rules.

Now it is certain, the project consists of linking several images **Docker**, and to be able to launch them together, without them losing their independence.

All this thanks to **Docker-Compose** which is intended for this type of use.

## But is this useful in everyday life?

In fact, the use of **Docker-Compose** takes on its full meaning in an IT infrastructure.

Imagine, you launch your startup, **Bananeo**.

You would need to set up a website, you create your NGINX image (or you get the one from NGINX on the **DockerHub**).

You now have a functional website for your business.

Obviously this may take more time than 2 sentences to put in place 😅

Now that **Bananeo** has about ten employees, it would be nice to set up a badge reader  at the entrance to avoid too much work for Manu  from security.

So you create a new image **Docker**, specially designed to register your employees in a **database** and being directly connected to your time clock.

You also quickly deploy another image managing an [intra.Bananeo.fr](http://intra.Bananeo.fr) website allowing your employees to manage their working hours.

And Yop! Everything works. But your time clock never communicates with the intra site, and it would be nice to be able to tell your employees when they are late, especially for Eric who has always found the right excuse for the past 2 weeks.

## That's where Compose comes in !

You are going to set up a file **.yml**, remember this is the format **YAML** which allows instructions to be given to **Compound** on how to manage these different images.

Docker	Subject	NGINX	MariaDB	WordPress	Compound	Volumes	Finalize the project
--------	---------	-------	---------	-----------	----------	---------	----------------------

```

version: "3"

services:      # précise les différents services (images) à utiliser
nginx:
  build: requirements/website/
  env_file: .env          # indique le fichier optionnel contenant l'environnement
  container_name: website # Le nom du container ( doit porter le même nom que le service d'après le sujet )
  ports:
    - "80:80"             # le port, détaillé juste en dessous
    restart: always        # Permet de redémarrer automatiquement le container en cas de crash
nginx:
  build: requirements/intra/
  env_file: .env
  container_name: intra
  ports:
    - "80:80"
    restart: always
mariadb:
  container_name: badgeuse
  build: mariadb
  env_file: .env
  restart: always

```

😢 No, this docker-compose.yml does not work on Inception.

This file **.yml** allows to give instructions to **Compound** to manage 3 mariadb images, NGINX and WordPress. Like a **Dockerfile** which must necessarily begin with **FROM** version tracking, your file **YAML** must start with the version of **Compound**, refer to current versions .

## 2. Understand the topic:

### ► SUBJECT TO DOWNLOAD

📘 Each service will have to run in a dedicated container.

Here are the different containers to set up according to the subject:

NGINX(*with TLS v1.2*)

WordPress(*with php-fpm configured*)

MARIADB(*without NGINX*)

Here are the two volumes to be set up according to the subject:

Volume containing your **WordPress database**

Volume containing your **WordPress site files**

These volumes must be available from the **/home/ <login> /data** host machine folder using **Docker**.

	Docker	Subject	NGINX	MariaDB	WordPress	Compound	Volumes	Finalize the project	
--	--------	---------	-------	---------	-----------	----------	---------	----------------------	--

We will also need to put in place:

A docker-network that will link your containers.

Users to create in our WordPress database:

An **Admin** user (must not be called admin)

A standard user

For readability reasons, we will need to configure our domain name to point to **our local IP address**. This domain name will be <login>.42.fr

**In order to carry out these different tasks, the subject provides several other details:**

The latest tag is forbidden.

When you specify a dependency to install, you must specify its version to install, the advantage of the tag **latest** is clear, install the **latest version** of a dependency.

However **latest** also has a big disadvantage, it can bring **compatibility problems** over time.

**Example:** Today you could configure everything so that the latest versions work correctly with each other, but imagine that 2 years later someone wants to test your software, it could be that some dependencies have evolved and work differently with each other, that's why it is better to indicate the version to use.

No password should be present in your **Dockerfiles**.

Do we really need to explain this point? 😅

It must be said that this is one of the biggest mistakes made by devs using Github.

Yes, it happens that developers push certain sensitive information into their repo, such as API keys or even clear passwords 😱

Obviously this remains the number 1 mistake to avoid.

If this happens to you, don't make the second mistake of re-pushing on top by deleting your key or password, Github commits will eventually betray you ✘

Instead, consider quickly deleting your commit .

The use of environment variables is mandatory.

This instruction goes with the previous one, so you must use your machine's environment.

This might seem useful for storing passwords or the like.

Docker	Subject	NGINX	MariaDB	WordPress	Compound	Volumes	Finalize the project
--------	---------	-------	---------	-----------	----------	---------	----------------------

We now know how to store our environment.

In the topic we also find an example of what this .env file could look like thanks to a `cat srcs/.env` :

```
DOMAIN_NAME=wil.42.fr
# certificates
CERTS_=./XXXXXXXXXX
# MYSQL SETUP
MYSQL_ROOT_PASSWORD=XXXXXXXXXX
MYSQL_USER=XXXXXXXXXX
MYSQL_PASSWORD=XXXXXXXXXX
[...]
```

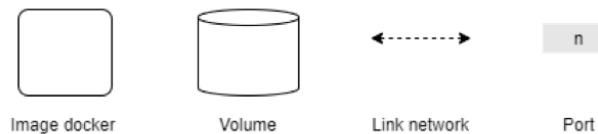
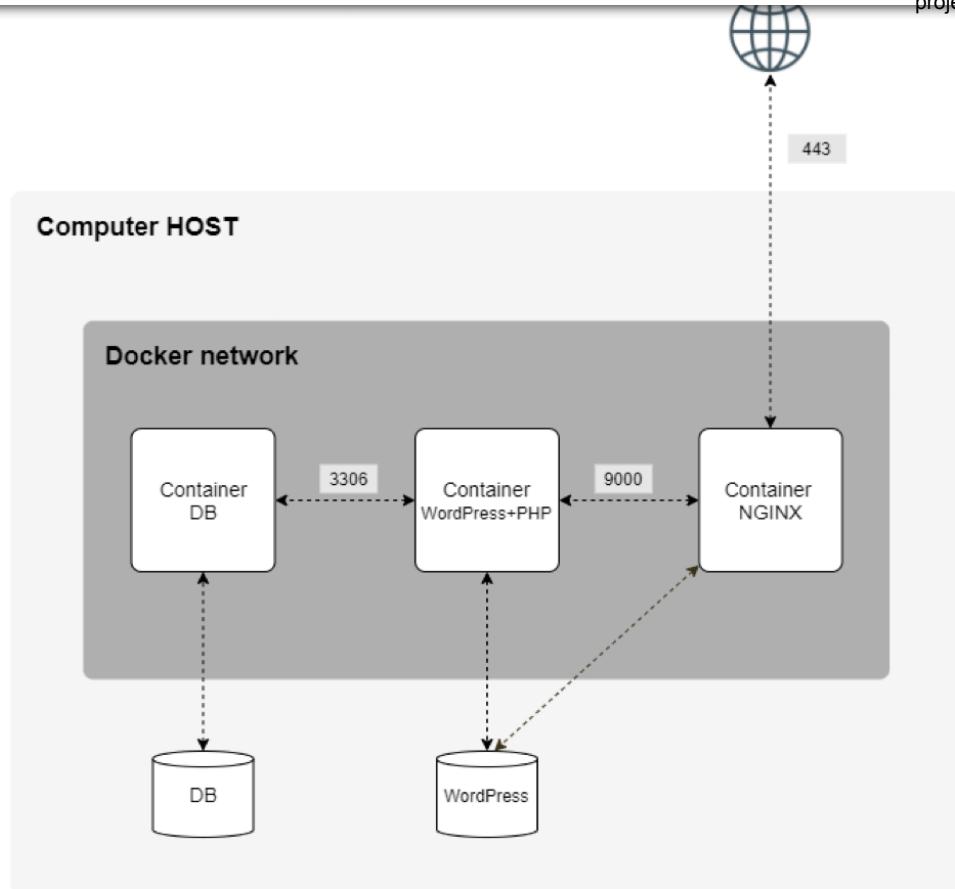
 Thanks to this `cat` we can imagine the information we will need to set up our project.

 Your **NGINX** container must be the only entry point to your infrastructure through port **443 only using TLSv1.2 or TLSv1.3 protocol**

The only entry point of our Compose must be through the NGINX container, going through port **443**.

That is to say that the only port that Docker-Compose will open on your machine will be **443** (*tips: this is the port that allows access via https://, and 80 via http://*)

This **diagram** provided in the topic should clear things up:



Here we find the ports that will allow **WordPress** to communicate internally with the database(to be able to store new pages or others)and **NGINX** which communicates with **WordPress** (which will tell it what to display on the web server).

The **MariaDB** container stores its information(*database*)in the corresponding volume and **WordPress** (*wordpress site*)in his.

It is stated that we should use **TLSv1.2** or **TLSv1.3** protocol

But what is **TLS**?

[Wikipedia](#) tells us:

Transport **Layer Security ( TLS )** is a protocol securing exchanges via computer networks , particularly via the Internet .

	Docker	Subject	NGINX	MariaDB	WordPress	Compound	Volumes	Finalize the project	<input type="checkbox"/>
--	--------	---------	-------	---------	-----------	----------	---------	----------------------	--------------------------

- the integrity of the data exchanged
- optionally, client authentication (*but in reality this is often ensured by the application layer*)

I don't know about you, but **TLS** seems to me to be very similar to SSL , the famous protocol that adds a **green padlock** when you access a site **secure**, like most sites today.

This is the protocol that changes your **URLs** from *http* to *https* .

So my question is quickly,

### What is the difference between **TLS** And **SSL**?

**SSL** and **TLS** are two protocols that allow authentication and encryption of data that passes between servers.

In fact, **SSL** is the predecessor of **TLS** . Over time, new versions of these protocols have emerged to address vulnerabilities and support ever stronger, ever more secure encryption suites and algorithms

### The History of **SSL/TSL**:

Initially developed by **Netscape** , SSL was released in **1995** in its SSL 2.0 version.(SSL 1.0 was never released).

But after the discovery of several vulnerabilities in **1996** , version 2.0 was quickly replaced by SSL 3.0.

Versions 2.0 and 3.0 are sometimes labeled as **SSLv2** and **SSLv3**.

Based on SSL 3.0, TLS was introduced in **1999** as **the new version of SSL**.

The **TSL** protocol is therefore simply the **replacement for SSL** . It corrects certain **security vulnerabilities** in older **SSL** protocols .

We will therefore have to make our system secure, by a protocol similar to SSL, the**TSL**.

---

Before starting our project, let's look at the last document provided with the topic, the **expected structure of our project** :

```
$> ls -alR
total XX
drwxrwxr-x 3 wil wil 4096 avril 42 20:42 .
drwxrwxrwt 17 wil wil 4096 avril 42 20:42 ..
-rw-rw-r-- 1 wil wil XXXX avril 42 20:42 Makefile
drwxrwxr-x 3 wil wil 4096 avril 42 20:42 srcs

./srcs:
total XX
drwxrwxr-x 3 wil wil 4096 avril 42 20:42 .
drwxrwxr-x 3 wil wil 4096 avril 42 20:42 ..
-rw-rw-r-- 1 wil wil XXXX avril 42 20:42 docker-compose.yml
-rw-rw-r-- 1 wil wil XXXX avril 42 20:42 .env
drwxrwxr-x 5 wil wil 4096 avril 42 20:42 requirements

./srcs/requirements:
total XX
```

Docker	Subject	NGINX	MariaDB	WordPress	Compound	Volumes	Finalize the project
<pre>drwxrwxr-x 4 wil wil 4096 avril 42 20:42 wordpress ./srcs/requirements/mariadb: total XX drwxrwxr-x 4 wil wil 4096 avril 42 20:45 . drwxrwxr-x 5 wil wil 4096 avril 42 20:42 .. drwxrwxr-x 2 wil wil 4096 avril 42 20:42 conf -rw-rw-r-- 1 wil wil XXXX avril 42 20:42 Dockerfile -rw-rw-r-- 1 wil wil XXXX avril 42 20:42 .dockerignore drwxrwxr-x 2 wil wil 4096 avril 42 20:42 tools [...]  ./srcs/requirements/nginx: total XX drwxrwxr-x 4 wil wil 4096 avril 42 20:42 . drwxrwxr-x 5 wil wil 4096 avril 42 20:42 .. drwxrwxr-x 2 wil wil 4096 avril 42 20:42 conf -rw-rw-r-- 1 wil wil XXXX avril 42 20:42 Dockerfile -rw-rw-r-- 1 wil wil XXXX avril 42 20:42 .dockerignore drwxrwxr-x 2 wil wil 4096 avril 42 20:42 tools</pre>							

 Let's not neglect this kind of information provided with the subject, it can already help us start the project.

This structure clearly shows us that there must be a main folder `srcs` containing our famous **docker-compose.yml** that we saw just above, our file `.env`, as well as a folder `requirements` that will contain our different containers.

Reminder of the **3 containers** to put in place:

**NGINX** (with TLS v1.2)

**WordPress** (with php-fpm configured)

**MARIADB** (without NGINX)

As I explained at the beginning, each container is represented by a symbol `dossier` bearing its name.

In each container folder there must be its **Dockerfile**, without what **Docker** would be lost.

Remember, in addition to the **Dockerfile** we can add different configuration files/folders that it would then be interesting to copy into our container using the **Dockerfile** and its keyword **COPY**.

The diagram provided with the subject directly tells us which files it would be interesting to provide with our containers, here they are:

- A **conf** file that could contain the container configuration file (*the NGINX config for its associated container for example*)
- A `.dockerignore` file . Just like the `.gitignore` file , it tells **Docker** files that should not be looked at, because they will probably have no use for you **Docker**.
- A folder `tools` that will probably store other tools that we might need.

We can already reproduce the requested file structure to see things more clearly 😊

Now that we understand what the subject expects from us, it would be interesting to ask ourselves:

## Where to start ?

Once we have our `dossiers` required files, we should start by creating one of the three required containers.

### 3. NGINX Container:

Let's start with our least fuzzy container, **NGINX**.

**NGINX** allows you to set up a web server.

We will start step by step in order to better understand how to use and walk around with it.**Docker**.

Let's first open the essential Docker file, the**Dockerfile NGINX**.

 Remember that there is a ready-made NGINX image, made available by **NGINX** on the **Docker Hub**.  
Most developers would have gone straight through this, but here **42** forbids us from doing so, so we can recreate it ourselves.

If you followed correctly, you should know which line to write first, at least the first keyword in the Dockerfile.

#### ▼ SPOILER 🔍

It is**FROM**! As stated at the beginning of the course,**FROM**allows you to tell Docker which OS we want to run our machine on. This is a mandatory keyword and must always be first in a**Dockerfile**.

 For performance reasons, the containers will have to be built either under Alpine Linux with the penultimate stable version, or under Debian Buster.

So for the OS we have the choice between**debian:buster**Or**alpine:X.XX**(check the penultimate stable version [here](#)).

#### What is the difference between Alpine and Debian ?

**Alpine Linux**is a **lightweight, security-oriented Linux** distribution , it contains as few files and tools as possible in order to allow the developer to install them themselves if necessary.

**Debian**is the universal operating system. Debian systems currently use the Linux kernel or the **FreeBSD** kernel .

For my part, being more comfortable with this system, I will use**Debian**.

 The rest of the tutorial is shown for**Debian**. But you should easily be able to adapt this to**Alpine**.

So we can start by writing: **FROM debian:buster** in our**Dockerfile**.

Now I want to tell you, if this line is the only mandatory line, could our container start?

We have specified an OS to install, we should be able to launch it.

The coolest thing is that**Docker**can allow us to launch a "*virtual machine*", but by specifying it, we can also access its terminal! Useful for seeing what's inside or performing our own tests.

## The essential commands of a Docker container:

A container **Docker** must be **build** before launching.

It is during the **build** that you will be able to obtain information on potential errors that you may have made in the **Dockerfile**.

 Every Docker command starts with the keyword `docker`

Build a Docker container: `docker build`

 I hope you wouldn't dare try this command without having fully understood it!

 In any case, the command is not functional, docker necessarily asks for a path where the Dockerfile of the image to be built is located.

In our case, it would be `docker build srcs/requirements/nginx/`

Or even simpler, `docker build .` by finding yourself directly in the NGINX folder.

You can also specify a name for your build, with the flag `-t`

Example : `docker build -t nginx .`

 If you get an error like this `Cannot connect to the Docker daemon`, check that Docker is open and running .

Know the current images(after a successful build): `docker image ls`

 You should see your first image!

 However this one does not have a name below **REPOSITORY**, it is indicated `<none>`, even if we can link this image to its ID present in the fourth column, the subject asks us that the name of the image bears the name of the associated container, here **NGINX** .

And we can do that! We simply need to specify when we build the image, its name, using the flag `-t`

Start an image(run) : `docker run <image_name>`

 You will need to provide the name of your image to run.

 By specifying `-it` the name of your image before, you will directly access the terminal of your container when it is launched.

Know the containers currently launched: `docker ps`

 You can even get the Docker containers stopped by adding the flag `-a`

	Docker	Subject	NGINX	MariaDB	WordPress	Compound	Volumes	Finalize the project	<input type="checkbox"/>
--	--------	---------	-------	---------	-----------	----------	---------	----------------------	--------------------------

Just put this piece of text in a file `execute.sh` and run it with bash.

```
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/d7c6c85d-8a9e-46e3-8b31-2898d714aa14/use.txt
```

We are now ready to attempt to launch our mini**Dockerfile**.

Let's start by **building** our container.

In the folder containing the**DockerfileNGINX** : `docker build -t nginx .`

Now let's launch the **container** we just **built** : `docker run -it nginx`

➤ `-it` allows you to open the container terminal when it is launched.

**Welcome** to your container!

To exit the terminal of a **container**, it's a classic, you have to type '`exit`'

If you do a `ls`, you will immediately see:

```
bin  dev  home  lib64  mnt  proc  run  srv  tmp  var
boot etc  lib  media  opt  root  sbin  sys  usr
```

This is the famous Debian kernel !

In itself, the files that allow the operating system to function properly.

Now you are in your container, you can install whatever you want.

Let's start by updating **apt** *in the container terminal*

**APT** is a command-line utility for installing, updating, removing, and managing deb packages on Ubuntu, Debian, and related Linux distributions.

**So APT** is a package manager for Debian, to be sure that the version of **APT** installed on this container knows the latest versions of the packages, we will ask it to check for updates using: `apt update` , if all goes well APT should respond: All packages are up to date.

We can add `apt upgrade` to install any updates of packages found.

Now that **APT** knows the latest versions of the packages, we will ask it to directly install the one we are interested in, **NGINX** .

Just type: `apt install nginx`

Great! Now we have a container that has **NGINX** .

But small problem: if we close our container, and we open it again, **NGINX** is gone! 😞

This would mean that you will have to enter the terminal each time you open the container to install it again? 😕

So we could simply add to our **Dockerfile**: `RUN apt install nginx` ➤ **install NGINX**

▼ **SPOILER** 🔎

If you try to build your container again with this added, **Docker** indicates this error: `E: Unable to locate package nginx`

Yes! We forgot to update APT just before with the command `apt update`!

Let's add it just before, `RUN apt update`

Now our build goes further! But there is still one **error**...

You may notice that the penultimate line says: `Do you want to continue? [Y/n] Abort.`

Huh? But yes I want to continue! However when **Docker** build a container, there is no prompt to respond **YES** to this question...

Anyway the idea of a **Dockerfile** is to be able to create your project without asking your opinion during installation.

How to do ?

**APT** provides an option `-y` to automatically respond **YES** to this kind of question when installing a package.

So it would be a good idea to get into the habit of adding it when adding a line using **APT**, otherwise your container will not be able to create itself.

So you just have to add a `-y` at the end of the line which concerns the installation of **NGINX** by **APT**.

This gives : `RUN apt install nginx -y`

Let's try to build again... It works!

You can now re-enter your container with `docker run -it nginx` and find yourself in the container again, but this time the **Nginx** installation is already done!

In order to be able to be comfortable accessing the container terminal, I will install **vim** and **curl**. Feel free to install the packages that seem useful to you:

```
RUN apt install vim -y
RUN apt install curl -y
```

Ok now on to the **TSL** !

We will create a folder, which will store the certificate and the key for a **secure connection**.

Let's add: `RUN mkdir -p /etc/nginx/ssl`

It would also be important to download the main tool for SSL certificate management/creation, **OpenSSL**

For this, like other packages, we install it with **apt**.

```
RUN apt install OpenSSL -y
```

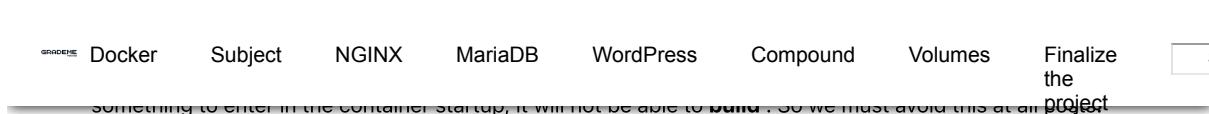
Next, we will need to generate an **SSL certificate and yes we use SSL** tools to create a **TSL/SSL** certificate .

**The command is a bit long, let's start with the openssl keyword then req .**

```
openssl req
```

 **The req** command primarily creates and processes certificate requests in PKCS#10 format. It can also create self-signed certificates.

We will then add the keyword **-x509** to specify the type of certificate.



Fortunately, **OpenSSL** has anticipated this, with the **-nodes** option , our private key will simply be left without a password.

```
openssl req -x509 -nodes
```

You then need to tell OpenSSL where you want to store the certificate and key for our SSL by adding the **-out** and **-keyout** options.

```
openssl req -x509 -nodes -out /etc/nginx/ssl/inception.crt -keyout /etc/nginx/ssl/inception.key
```

If we launch this command we risk having a**prompt**which requires certain information for the certificate. Fortunately, OpenSSL has planned for this again and allows us to pre-fill them by adding an option called **-subj**

```
openssl req -x509 -nodes -out /etc/nginx/ssl/inception.crt -keyout /etc/nginx/ssl/inception.key -subj "/C=FR/ST=IDF/L=Paris/O=42/OU=42/CN=login.42.fr/UID=login"
```

All that remains is to enter this preceded by a **RUN** in our**Dockerfile**.

**Well done !**Your TSL key and certificate are now automatically created when you start your **container** !

---

We will then create a folder that will allow us to store the NGINX config files.

```
RUN mkdir -p /var/run/nginx
```

Now we should modify the **NGINX** configuration *file* as we wish. To do this, we could redirect **certain** sentences in the **config** file , but we will instead use the keyword**COPY**of**Dockerfile**, which seems made for that.

So we're going to take the **base NGINX** config file and modify it afterwards. It's located **in** `/etc/nginx/nginx.conf`

```
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/a97aeef6-ffc4-4d2e-835d-36986aacdff  
5/nginx.conf
```

We will put it in a folder **conf** , itself in the folder **nginx** (*to respect the structure indicated in the subject*).

By adding the line:

```
COPY conf/nginx.conf /etc/nginx/nginx.conf
```

**The NGINX** configuration file should be replaced by ours when the container starts, let's do a test by adding the sentence"yes we can"commented out at the beginning of our `nginx.conf` file in the **conf** .

If all goes well, by **building** the container, and **launching it** , you should see your NGINX conf file modified by accessing it with vim for example.

Before launching **NGINX** , you should tell **NGINX** which configuration you want to use within Inception .

---

## The NGINX configuration file:

This page helps us configure **NGINX** for **ssl** .

Here is the basic configuration that can be found:

Grade	Docker	Subject	NGINX	MariaDB	WordPress	Compound	Volumes	Finalize the project	

```

ssl_protocols TLSv1 TLSv1.1 TLSv1.2;           > Gardons TLSv1.2 & TLSv1.3
ssl_certificate /etc/nginx/ssl/bundle.crt;        > Indiquons notre certificat
ssl_certificate_key /etc/nginx/ssl/private.key;    > et notre clef
...
}

```

The config file therefore becomes:

```

server {
    listen 443 ssl;
    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_certificate /etc/nginx/ssl/inception.crt;
    ssl_certificate_key /etc/nginx/ssl/inception.key;
    ...
}

```

Now we have the **SSL/TSL** part working with **NGINX**.

I will clarify the home folder by adding `root /var/www/wordpress;`

This is the folder where WordPress will be located and therefore its first page to display.

I will also specify which page to display first, in WordPress **you** have to indicate **index.php**, but I will also add others that seem important to me.

```
index index.php index.html index.htm;
```

Since the connection will be made from localhost, I indicate it in **server\_name**: `server_name localhost;`

 It will be necessary to change localhost by IP `login.42.fr`

The config file now looks like this:

```

server {
    #SSL/TLS Configuration
    listen 443 ssl;
    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_certificate /etc/nginx/ssl/inception.crt;
    ssl_certificate_key /etc/nginx/ssl/inception.key;

    #root and index and server_name
    root /var/www/html;
    server_name localhost;
    index index.php index.html index.htm;
}

```

Next, you need to add rental rules for **WordPress**.

First we need to tell NGINX to return any request we don't know about with a **404 error**.

This [topic on stackoverflow](#) explains this well. So we add the rule:

```

location / {
    try_files $uri $uri/ =404;
}

```

We still need to install **PHP** to be able to handle WordPress **PHP** requests .

 The topic indicates that PHP must be installed on the **WordPress** container and not NGINX.

No worries! So we'll just tell **NGINX** how to handle **PHP** and tell it where to return our **php** code .

```
location ~ \.php$ { # Pour toutes les requetes php
    include snippets/fastcgi-php.conf;
    fastcgi_pass wordpress:9000; # Comment renvoyer les requetes php sur le port 9000
}
```

I will now simply add in the **Dockerfile** an order `chmod` to ensure that we will have the rights to what we need.

```
RUN chmod 755 /var/www/html (our main root)
RUN chown -R www-data:www-data /var/www/html (the main user)
```

Now all that remains is to launch NGINX, we will use the keyword `CMD : CMD [ "nginx", "-g", "daemon off;" ]`

This will start **NGINX** in the foreground so the container does not stop.

You could see the NGINX homepage by going to localhost , using **https** (because you have not opened the **http**). You should also change the **root** of the configuration file and remove it `wordpress:9000` for php, because you don't have the PHP container yet(WordPress)

### A container completed!

Which container should I continue with?

We will need **MariaDB** to install **WordPress** .

When installing **WordPress** , it requires a web server(NGINX), and a database, here **MariaDB** .

So it would be wiser to understand everything about databases and start by creating this container.

## 4. MARIADB Container:

**MariaDB** is a database management system released under the GPL license. It is a community fork of **MySQL** : the project is governed by the **MariaDB Foundation**.

Basically, **MariaDB** is almost a copy of **MySQL** , but why?

**MySQL** was initially completely open-source , then it was acquired by **Oracle** .

Since then, several organizations have been concerned about the possibility that **Oracle** could make **paying** his software.

To prevent this, the **MariaDB** Foundation created a version that was almost identical to MySQL, but completely open-source.

How a database like MySQL works:

nom	Docker	Subject	NGINX	MariaDB	WordPress	Compound	Volumes	Finalize the project
oeuvre_id	auteurs_id	titre			Date de publication			
1	1	Le seigneur des anneaux				1954		
2	1	Le Hobbit				1937		
3	1	Le Silmarillion				1977		
4	2	Le guide du voyageur galactique				1979		
5	2	Le dernier restaurant avant la fin du monde				1980		
6	2	La vie, l'univers et le reste				1984		

*This tutorial* perfectly shows how to create a database with **MariaDB** on [Debian 10](#).

*UPDATE:* This tutorial perfectly shows how to create a database with **MariaDB** on Debian 11.

# Tutorial for using MySQL/MariaDB:

To start and be able to do your tests on your side too, we will create a **Dockerfile** minimum for **MariaDB**. If you remember what we did for the beginning of **NGINX**, we simply need to start by writing **FROM debian:buster**.

From there, you can **build** and launch the container by entering your terminal(go back up to find out how if necessary)

Now we find ourselves in our container and we can carry out our test orders there.

Let's start with the famous, essential, `apt update -y`.

Let's also add the `apt upgrade -y`.

**APT** now contains up-to-date dependencies, we can install **MariaDB**.

Since MariaDB version 10.3 is the most common, we can install it using [APT](#).

```
apt-get install mariadb-server -y
```



Think about the -y, otherwise docker will have trouble **build** your container because it will wait for confirmation from **y/n** ...

Next, let's visit the MySQL configuration file, called `50-server.cnf` and located at `/etc/mysql/mariadb.conf.d/50-server.cnf`

This is what the base file looks like:

```
# The MariaDB configuration file
#
# The MariaDB/MySQL tools read configuration files in the following order:
# 1. "/etc/mysql/mariadb.cnf" (this file) to set global defaults,
# 2. "/etc/mysql/conf.d/*.cnf" to set global options.
# 3. "/etc/mysql/mariadb.conf.d/*.cnf" to set MariaDB-only options.
# 4. "~/.my.cnf" to set user-specific options.
#
# If the same option is defined multiple times, the last one will apply.
#
# One can use all long options that the program supports.
# Run program with --help to get a list of available options and with
# --print-defaults to see which it would actually understand and use.
#
# This group is read both both by the client and the server
# use it for options that affect everything
#
[client-server]

# Import all .cnf files from configuration directory
!includedir /etc/mysql/conf.d/
!includedir /etc/mysql/mariadb.conf.d/
```

	Docker	Subject	NGINX	MariaDB	WordPress	Compound	Volumes	Finalize the project	<input type="checkbox"/>
--	--------	---------	-------	---------	-----------	----------	---------	----------------------	--------------------------

Ok, what if we *remove* the comments?

```
[client-server]
!includedir /etc/mysql/conf.d/
!includedir /etc/mysql/mariadb.conf.d/
```

### The MariaDB configuration file :

So that's the gist of the basic config file , except we don't need that part that concerns the client side.

We start with the brackets `[mysqld]` in the file to indicate which category the next configuration will follow (this is how the MySQL configuration file **works** )

This is where we can tell **MySQL** which port to communicate on, indicated by the subject, it is **3306** with `port = 3306`

Afterwards, we also specify that all IPs on the network can connect, for this it is the line `bind_address=*` , which roughly means: `bind_address=XXX.XXX.XX.XX`

We can also specify our folder which will store our database with `datadir=/var/lib/mysql`

I add the user with `user=mysql`

I'm not sure if this is required since it's the base path, but I also specify where MySQL can find the **socket** to communicate with: `socket = /run/mysqld/mysqld.sock`

### Ok good!

So our configuration file should be:

```
[mysqld]
datadir = /var/lib/mysql
socket = /run/mysqld/mysqld.sock
bind_address=*
port = 3306
user = mysql
```

Now we need to save this file in a folder `conf` as requested by the project structure, I give it the same name, `50-server.cnf` .

Then all that remains is to ask the **Dockerfile** to copy it to the right place to replace it.

```
COPY conf/50-server.cnf /etc/mysql/mariadb.conf.d/50-server.cnf
```

Now that **MySQL** is correctly **installed**, you need to create a **database** and an associated **user** .

For this I will go through a script that I will ask the **Dockerfile** to execute.

This script must create with MySQL the table system, thanks to the command `mysql -e`

In this script we can first **to start up** MySQL, otherwise it will be difficult to configure.

```
service mysql start;
```

	Docker	Subject	NGINX	MariaDB	WordPress	Compound	Volumes	Finalize the project	
--	--------	---------	-------	---------	-----------	----------	---------	----------------------	--

```
mysql -e "CREATE DATABASE IF NOT EXISTS ` ${SQL_DATABASE} `;"
```

➤ I ask to create a table if it does not already exist, named after the environment variable **SQL\_DATABASE**, indicated in my .env file which will be sent by the [docker-compose.yml](#).

The table is created! I will then create a user who will be able to manipulate it.

```
mysql -e "CREATE USER IF NOT EXISTS ` ${SQL_USER} `@'localhost' IDENTIFIED BY ' ${SQL_PASSWORD} ';"
```

➤ I create the user **SQL\_USER** if it does not exist, with the password **SQL\_PASSWORD**, always to be indicated in the .env

I give all rights to this user.

```
mysql -e "GRANT ALL PRIVILEGES ON ` ${SQL_DATABASE} `.* TO ` ${SQL_USER} `@'%' IDENTIFIED BY ' ${SQL_PASSWORD} ';"
```

➤ I give the rights to the user **SQL\_USER** with password **SQL\_PASSWORD** for the table **SQL\_DATABASE**

I will then change my **root user with localhost rights** with this command:

```
mysql -e "ALTER USER 'root'@'localhost' IDENTIFIED BY ' ${SQL_ROOT_PASSWORD} ';"
```

➤ I change the root rights by **localhost**, with the root password **SQL\_ROOT\_PASSWORD**

All that remains is to refresh all of this so that MySQL takes it into account.

```
mysql -e "FLUSH PRIVILEGES;"
```

➤ Quite simply.

Now we just need to restart MySQL for all this to take effect!

```
mysqladmin -u root -p$SQL_ROOT_PASSWORD shutdown
```

➤ I start by turning off MySQL.

```
exec mysqld_safe
```

➤ Here I run the famous command that MySQL constantly recommends when it starts.

My script that configures our database is now functional and should launch the **mariadb** container without any problems.

Let's try to launch the container using a:  `docker build -t mariadb .`

Then from a  `docker run -it mariadb`

If all goes well, you should have a little [OK] which indicates that MySQL was launched without problems, and if all goes well, you should also have a **error**!

Nothing to do, if **MySQL** starts and indicates at least [OK] is that you have completed this part.

## 5. WordPress Container:

Let's set up **WordPress** !

Luckily, **WordPress** is used by over **43%** of websites worldwide, as their homepage says. Therefore, we should find a lot of documentation about it.

Let's start as usual with aDockerfileunder **debian:buster** .

Then comes the traditional `apt-get update` and `apt-get upgrade`

To plan for the future(we will need to install WordPress with its download link)we need to install **wget**

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

As the topic tells us, we also need to install **PHP** with **WordPress** .

Remember, it will communicate on port **9000** with **NGINX** .

For this I also use **APT** to install **php7.3** and its dependencies like *php-fpm* and *php-mysql*.

```
RUN apt-get install -y php7.3  
php-fpm\  
php-mysql\  
mariadb-client
```

It's finally time to install the famous **WordPress** in our container!

So we use **wget** indicating the installation link. I took the **FR 6.0 version**.

It's up to you to choose, you will find the different versions here .

`wget` has several options, including one that will allow us to indicate in which folder we want to download the file using `-P`.

In which folder? **/var/www** obviously! This is where we indicated our main folder to display in the **NGINX** container .

```
RUN wget https://fr.wordpress.org/wordpress-6.0-fr_FR.tar.gz -P /var/www
```

➤ Very good, great, but now we need to untar it, or rather uncompress it!

Come on, I'll help you, let's go to the **/var/www** folder and use `tar -xvf` the filename to unzip it and get the famous folder `wordpress` !

Then we can delete the `.tar` which is no longer useful.

 Docker	Subject	NGINX	MariaDB	WordPress	Compound	Volumes	Finalize the project	<input type="checkbox"/>
--	---------	-------	---------	-----------	----------	---------	----------------------	--------------------------

### › Here we are **wordpress** in the form of a file.

Now let's make sure to give root permissions to write to this folder.

```
RUN chown -R root:root /var/www/wordpress
```

Now that **WordPress** is installed, but not configured, let's take care of **PHP**.

**PHP** is used for **Wordpress**, most of the files in WordPress are .php.

 PHP is a free programming language, primarily used to produce dynamic web pages via an HTTP server, but can also function as any locally interpreted language.

Once it is installed with **APT**, we can modify its configuration file.

Here I took the base configuration file and changed only 2 small things.

I added a line `clear_env = no`, I think this line is clear enough not to have to explain it, it's for the environment.

But I also changed the line `listen`. I indicated that it should listen at the WordPress port, **9000**.

Roughly speaking, this gives: `listen=wordpress:9000`

**PHP** is ready, and yes, finally after that you obviously have to ask the **Dockerfile** to copy the configuration file to the right place in the container, but there is no point in explaining it to you, you should already be experts in the field.

Now let's configure our friend **WordPress**.

What do I need to configure it for? ☺

**WordPress** needs a database to work, at least to know its **password**, **name** and **host**.

All this is configured in the **wp-config.php** file (*something like that*)

If you don't do this, you will arrive directly at the site configuration page, which is not bad, but the subject requires automatic configuration.

But I have good news for you! A developer has simply created a **CLI** that allows you to configure this kind of information almost automatically for you.

Take a look at *this great tutorial*.

 A CLI is a text-based interface that processes commands to a **computer** program. There are different command line interfaces, such as DOL and Bash Shell.

I'm going for this idea which seems really nice to me, you already have to reuse **wget** to install the CLI.

```
RUN wget https://raw.githubusercontent.com/wp-cli/builds/gh-pages/phar/wp-cli.phar
```

Then you need to give it the correct rights and place it in the binaries.

```
RUN chmod +x wp-cli.phar
RUN mv wp-cli.phar /usr/local/bin/wp
```

Ok great! We installed **the WordPress CLI** 

Now we should use it, for that we will do as with **MariaDB**, create a small bash script that we will copy and which will carry out the commands for us when launching the container.

So I create one `auto_config.sh` that I place in the **WordPress conf** folder .

In this one, I will, as a precaution, put a `sleep 10` in order to be certain that the **MariaDB** database has had time to launch correctly.

Then, and only if our file `wp-config.php` does not exist (we wouldn't want to reconfigure WordPress at every launch right?), we use the CLI command `wp config create` to specify the information that **WordPress** needs.

We could fill this information manually directly by accessing localhost while launching our container.

It would look like this:



The screenshot shows the WordPress configuration page. At the top, there is a large blue 'W' logo. Below it, a message says: "Below you should enter your database connection details. If you're not sure about these, contact your host." There are five input fields with descriptions:

- Database Name:** `wordpress` (The name of the database you want to use with WordPress.)
- Username:** `username` (Your database username.)
- Password:** `password` (Your database password.)
- Database Host:** `localhost` (You should be able to get this info from your web host, if `localhost` doesn't work.)
- Table Prefix:** `myprefix_` (If you want to run multiple WordPress installations in a single database, change this.)

At the bottom left is a `Submit` button.

› This is the first page WordPress presents when it is launched.

Except that we would like this to be automatically configured already.

So we can indicate this information with the help `wp config create` of CLI.

It would look something like this:

```
wp config create --allow-root \
```

➤ To be indicated directly after our sleep in our `auto_config.sh` file

Now if we launch the **WordPress** container , it should no longer be the page that I showed you just above but the second one, this one is much nicer.

Welcome

Welcome to the famous five-minute WordPress installation process! Just fill in the information below and you'll be on your way to using the most extendable and powerful personal publishing platform in the world.

**Information needed**

Please provide the following information. Don't worry, you can always change these settings later.

<b>Site Title</b>	<input type="text"/>
<b>Username</b>	admin
Usernames can have only alphanumeric characters, spaces, underscores, hyphens, periods, and the @ symbol.	
<b>Password</b>	<input type="password" value="W3hqrO8HCLtk#a9)\$C"/> <span style="margin-left: 10px;">Hide</span>
<b>Important:</b> You will need this password to log in. Please store it in a secure location.	
<b>Your Email</b>	<input type="text"/>
Double-check your email address before continuing.	
<b>Search Engine Visibility</b>	<input type="checkbox"/> Discourage search engines from indexing this site It is up to search engines to honor this request.

**Install WordPress**

This is the page that asks you to choose a **title** for your site, as well as a **username** and **password** , but it's impossible to get it wrong because it's up to you to choose it 😊

For me it would be normal to leave this configuration page to be filled, but the subject asks that it also be automatically configured, because it asks that there are 2 **WordPress** users , and it is on this page that we can configure the first one.

I'll let you look at the **CLI** documentation , but it offers to configure this second page automatically with the command `wp core install` and even to add another user with the command `wp user create` . Frankly, I'll leave you the simplest part.

All you have to do is copy the file `auto_config.sh` into your container and run it with the keyword **ENTRYPOINT**.

Docker	Subject	NGINX	MariaDB	WordPress	Compound	Volumes	Finalize the project
Finally, launch <code>php-fpm</code> with the command:							

```
/usr/sbin/php-fpm7.3 -F
```

You have completed configuring your **WordPress** container !

---

## 6. Connect containers with Compose:

### Make way for the famous docker-compose.yml !

Well, first learn how to write a **docker-compose** file .

This always starts with the version of **docker-compose**, the last one, the 3rd one.

The first line will therefore be: `version: '3'`

 Be careful with the indentation of the **docker-compose.yml** ! As with Python, it's the indentation that matters.

Then we indicate the line `services:`

And there, with an indentation of **1 tab** , we can list our different services.

I start with **MariaDB**:

```
mariadb:
  container_name: mariadb      # Le nom du container, oui vraiment.
  networks:
    - inception               # à quel network il appartient
  build:
    context: requirements/mariadb # où se trouve son Dockerfile
    dockerfile: Dockerfile      # le nom du Dockerfile ?
  env_file: .env                # le fichier d'environnement pour transmettre les variables
  volumes:
    - mariadb:/var/lib/mysql   # Voir plus bas
  restart: unless-stopped       # redémarre tant qu'il n'est pas stoppé
  expose:                      # le port à exposer
    - "3306"
```

**Let's move on to the NGINX service** . Here it's the same principle, except for two things that change and the **names/paths** obviously.

```
nginx:
  container_name: nginx
  volumes:
    - wordpress:/var/www/wordpress
  networks:
    - inception
    depends_on:           # Nouvelle ligne, indiquant de ne pas démarrer NGINX tant que WordPress n'a pas démar
    - wordpress
  build:
    context: requirements/nginx
    dockerfile: Dockerfile
    env_file: .env
```

Alright, you got it? Let's do the same for **WordPress**.

```
wordpress:
  container_name: wordpress
  env_file: .env
  volumes:
    - wordpress:/var/www/wordpress
  networks:
    - inception
  build:
    context: requirements/wordpress
    dockerfile: Dockerfile
  depends_on:           # WordPress démarera uniquement après MariaDB (sinon il ne pourra pas configurer la b
    - mariadb
  restart: on-failure
  expose:
    - "9000"
```

› And there you have it! (I wish someone had done the work of reading the documentation for me)

---

## 7. Volumes to configure:

Let's quickly talk about the Volumes line :

We assign a volume to **MariaDB**, as we will do with **WordPress**.

This is a prerequisite of the topic. We need to enable data persistence, for this we will store some folders directly on our computer locally.

And yes, imagine that your **WordPress container crash or shut down**, and that by pure chance, it has lost all your files, it would be very annoying, that is why we will prefer to store them directly locally, and **docker-compose** allows us to do so.

In the case of Inception, we will store the files of **MySQL** located in `/var/lib/mysql` and **WordPress** in `var/www/wordpress`. These are the **paths** that I indicate after the line **volume**. Once again it is the subject that asks us.

Here in the volume line we indicate **MariaDB** (*this is the name of the volume, we could have called it vol\_maria or something else, obviously it will involve a line concerning the volumes in the docker-compose.yml which will indicate where to store it locally*) followed by `:` with the location we want to copy from the container.

All that remains is to specify the volumes that we have indicated:

```
volumes:
  wordpress:
    driver: local # ici nous stockons le volume en local
    driver_opts:
      type: 'none'          # aucun type spécifique
      o: 'bind'
      device: '/Users/login/data/wordpress'    #On stocker le dossier sur votre ordinateur en local
  mariadb:
    driver: local
    driver_opts:
      type: 'none'
      o: 'bind'            # Les Bind Mounts sont des volumes qui se montent sur un chemin d'accès à l'hôte
      device: '/Users/login/data/mariadb'      #On stocker le dossier sur votre ordinateur en local
```

 Your volumes will be available in the /home/login/data folder of the host machine running Docker.

► So remember to modify the path of the device: line accordingly.

All we have to do now is create the network part, be careful, it's very simple.

```
networks:  
  inception:  
    driver: bridge
```

► Yes that's all.

 Here `bridge` tells Docker to automatically install rules that will allow the 3 containers to communicate in bridge.

Your `docker-compose.yml` file is ready! I hope you didn't go as far as copying the **paths** without modifying them, that would be a shame 😊

Ah, I almost forgot! You can **build** your file.`yml`with the order

```
docker-compose -f <path_docker_compose> -d --build
```

To **stop** it : `docker-compose -f <path_docker_compose> stop`

To **remove** the build: `docker-compose -f <path_docker_compose> down -v`

If you have problems with docker you can use the command:

```
docker system prune -af
```

Be careful, this deletes all containers, images, etc.

---

## 8. Finalize the project:

### Small corrections:

Since testing all this, I've noticed a few small bugs, so I've added a few small clarifications to the **NGINX** configuration file .

I added `nginx.conf` the keyword in the first line of the file `events {}` because **NGINX** was asking for it with an error of the type `missing events{}` .

In this same file, I also added the line just below the `http include /etc/nginx/mime.types;` keyword .

Why? The CSS was not loading, and after a long investigation I noticed that these were indicated with a Content-Type of `html` ?!

Docker	Subject	NGINX	MariaDB	WordPress	Compound	Volumes	Finalize the project	<input type="checkbox"/>
Basically, all you have to do is create a <code>Makefile</code> and <b><u>you're good to go!</u></b>								

Oh yes, I chose to add a mini bash script that automatically configures the right paths in docker-compose.yml, always nice when a friend wants to test the project on his computer, he doesn't have to search for each path in all the files to change them.

If you try to connect to localhost or 127.0.0.1 from your browser (*after launching your container obviously*) you probably won't see anything appear. Normal we only opened port **443** as a listening port.

Port **443** corresponds to the **SSL** port so we must connect to it using `https://` and not `http://`, because the latter would take us to the standard port **80**, which we would surely have configured on a classic site (this would have redirected to 443). But here the subject **forbidden** the opening of a port other than **443**.

Another concern, depending on your browser, it should display an **alert** message indicating that this site is trying to **steal sensitive information**.

In **dev** language, this means that the **SSL** certificate was not signed by **Trusted Authority**. We can't do anything about it when it comes to a local project and even less with a certificate generated by **OpenSSL**.

However, we can still enter the site. For Safari, there is a button that still asks for access to the site. On Chrome, there is a little secret, just type the word `thisisunsafe` when you are on the alert page.

And there you have it, you should arrive at your site.



For readability reasons, you will need to configure your domain name to point to your local IP address.

This domain name will be `login.42.fr`. Once again, you will use your login.

#### ► This is only a configuration to be carried out on your machine locally.

Basically you can already access the site from **localhost**, which (*this word*) actually redirects to IP **127.0.0.1**, without telling you, but all of this is actually written to a file on your computer.

This file is very sensitive and you will need to edit it with a `sudo`. Indeed, it is a file very targeted by hackers, it would allow you to easily redirect to a fake Google when you type `google.fr`, for example.

By editing this file which is usually found here: `/etc/hosts` you can easily request the redirection of **127.0.0.1** to the IP you want, like `login.42.fr`



Don't forget to modify this IP in the NGINX conf file in the box `server_name`, the best would be to do it also in the generation of the **SSL** certificate, but hey, this one is not authenticated...

You now seem to be following all the rules of the topic.

Good luck with the correction! ☺