pmouhali / **ft_server**

An introduction project to webservers and docker containers.

#docker #docker-image #linux #nginx #42projects #mysql #42school #wordpress #phpmyadmin #tutorial #lemp-stack

40 commits | **1** branch | **0** packages | **0** releases | **1** contributor

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download


pmouhali Update README.md — Latest commit `890ecc2` 7 days ago

srcs	rename dir with subject name	2 months ago
Dockerfile	correct Dfile	2 months ago
README.md	Update README.md	7 days ago

README.md

ft_server

Usage :

Dans le repo :

```
docker build -t [nom-image] .
docker run -p 443:443 [nom-image]
```

Urls disponibles (navigateur, le certificat ssl n'est pas reconnu, le navigateur renverra une erreur de connexion non privée, poursuivre la navigation quand même) :

- localhost/index.html
- localhost/wordpress
- localhost/wordpress/wp-config
- localhost/phpMyAdmin

Pour pouvoir utiliser shell dans le container :

```
docker exec -it [pid ou nom du container] /bin/bash
```

Pour faire des tests de fonctionnement basiques :

```
cd /root/tests/
bash test.sh
```

- localhost/test.html (la configuration basique fonctionne)
- localhost/info.php (la configuration php fonctionne)
- localhost/todo_list.php (la configuration php/sql fonctionne)

tutoriel

Docker-machine mac 42

Docker utilise une techno native Linux donc on devra le faire tourner sur machine virtuelle. Pour pas se prendre la tête :

- installer VirtualBox depuis le Managed Software Center (dispo sur les macs de l'école)
- installer Docker toujours depuis le Managed Software Center
- run ce script : https://github.com/Millon15/php_piscine/blob/master/docker_init.sh

Pour faire tourner le serveur on le branche sur un ou plusieurs ports (80, 443). Le serveur écoute le port X du container. Mais le port X du container n'est pas forcément le port X de notre machine (sur laquelle tourne notre navigateur), ça va poser problème pour se connecter au serveur. Il faudra 'mapper' les ports : c'est à dire les faire correspondre.

Au moment de lancer un container, on pourra utiliser l'option '-p' pour que le port X du container corresponde au port X de la machine sur laquelle le container tourne.

Since we run our container on docker-machine, a virtual machine, the docker port mapping command will not map the port of the container to that of our mac, but to that of the virtual machine. Same thing for the VM, it will be necessary to map its port to ours.

So: port: mac <-> port: vm <-> port: container.

After having run the script without error, to map the port of the vm to ours (443 is the port number for my config, we can put another one):

```
docker-machine stop default
vboxmanage modifyvm default --natpf1 "localhost,tcp,,443,,443"
docker-machine start default
```

Dock worker:

A Docker image contains everything that we decide to install (Java, a database, a script that we are going to launch, etc.) for a container, but is in an inert state. Images are created from configuration files, called “Dockerfile”, which describe exactly what needs to be installed on the system. A container is the execution of an image: it has the file system copy of the image, as well as the ability to launch processes. In this container, we will be able to interact with the applications installed in the image, execute scripts, run a server, etc.

Once the Dockerfile fills with what we want, we can build the image (as compile) then run a container from this image (execute). In the Dockerfile folder:

```
docker build -t [nom-qu-on-veut-donner-a-l-image] .

docker run [nom-qu-on-a-donner-a-l-image]
```

Dockerfile line: 1: `FROM debian:buster-slim`

The FROM instruction allows you to specify an existing image on which you want to build. Perfect for us, we're going to boot from an image that contains the Debian Buster OS.

If we build then run with this single instruction, we get a container in which Debian Buster is installed. The -i option of the run command allows us to launch a container in interactive mode: we have access to the container terminal. If we run the ls command in this terminal, we can see that the files present are only those delivered with the installation of Debian.

Ideally, we want to obtain a container in which the server is already ready and does not require any manual configuration once launched.

To configure the server, we will have to install the packages we will need (php for phpMyAdmin for example), and copy the files and folders we will use there (a nginx configuration file for example).

To execute a command during the creation of the image, and therefore before launching the container, we use the RUN instruction of Dockerfile, which can execute a Shell command. We can directly specify the commands or ask it to execute a script that will contain all of our commands and operations (copying files for example).

Here we will use the RUN instruction only to install the packages.

Dockerfile line: 3-12: `RUN apt-get -y update && apt-get -y install mariadb-server \ wget \ php \ php-cli \ php-cgi \ php-mbstring \ php-fpm \ php-mysql \ nginx \ libnss3-tools`

To choose how to build your Dockerfile: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

Now if we build / run then ls, we can see that the files present are those delivered by Debian and those installed by libs (packages).

src / container_entrypoint.sh line: 3-12: `COPY srcs ./root/`

All the contents of the 'srcs' folder as is will be copied to the container's 'root' folder. The container will therefore have, among other things, in its 'root' folder our configuration files, and our wordpress and phpMyAdmin installation folders.

Dockerfile line: 16: `WORKDIR /root/`

The WORKDIR instruction is used to specify in which folder to be placed when the container is launched, if an ENTRYPOINT or CMD (https://docs.docker.com/engine/reference/builder/#entrypoint) is specified, the commands launched will be from this folder. If the command is a script to be executed, it is necessary to launch the script in the directory where the script is located.

Dockerfile line: 16: `ENTRYPOINT ["bash", "container_entrypoint.sh"]`

The ENTRYPOINT instruction allows (very rough explanation) to specify an order to be executed when the container is launched. We will launch the script 'container_entrypoint.sh' which contains the operations which finalize the server config.

Notes:

The 'container_entrypoint.sh' script launches the command `service nginx start` . This command starts the nginx server.

If we executed this script via the RUN instruction of Dockerfile, during the construction of the image, when launching the container the server would not be started at all, and therefore will not function. An image cannot contain processes. It will only contain 'static' data, such as files, instructions for moving files, installing files (packages), etc.

All the instructions relating to processes must be launched once the container has started, either manually in the container, or via an ENTRYPOINT or CMD instruction for example.

LEMP Stack

Basically a LEMP stack, it's the linux / nginx / php / mysql combo to run a server. We need these four tools (in addition to wordpress and phpmyadmin). We can start by following this tutorial to make a basic installation and verify that everything works: https://www.digitalocean.com/community/tutorials/how-to-install-linux-nginx-mariadb-php-lemp-stack- on-debian-10

srcs / container_entrypoint.sh line: 3-6: (we can skip the creation user sql part of the tutorial for the moment)

```
mkdir /var/www/localhost
```

We create the folder in which the server will look for the web pages (src / localhost_index_on line: 5)

```
cp localhost_index_on /etc/nginx/sites-available/localhost
```

We copy the custom config file in the nginx directory (the file should just be called localhost)

```
ln -s /etc/nginx/sites-available/localhost /etc/nginx/sites-enabled/
```

We link the conf file of sites-available with that of sites-enabled

srcs / tests / test.sh line: 3-5:

```
cp info.php /var/www/localhost/info.php
```

We copy our test files, a basic html to see that the server redirects to the correct url, a php to see if the php config works (todo_list.php will be used later), we place them in the folder in which the server will go search web pages.

srcs / container_entrypoint.sh line: 27-30: `service nginx start`

We start the server.

```
/etc/init.d/php7.3-fpm start
```

We execute the initialization script of php-fpm (it's not in the tutorial, we check that we have it in the container: ls /etc/init.d/, if we don't have it we don't have installed the correct packages or an error has occurred.)

srcs / localhost_index_on line: 30: `fastcgi_pass unix:/var/run/php/php7.3-fpm.sock;`

Check on this line that the php version is the right one (otherwise the php processing will not work).

Then we can try to see if we can access our two urls: localhost / test.html and localhost / info.php

If the html returns a 404, relaunch by mapping the ports (80 if the configuration is that of the tutorial, 443 if the configuration is the same as mine ie using ssl):

```
  docker run -p 80:80 -ti [nom-qu-on-a-donner-a-l-image]
```

To verify that we can actually access our database from other services (which we will have to do for wordpress and phpmyadmin), we will create a database, and insert random values inside, then get them from a php script:

srcs / tests / test.sh line: 9-18:

```
echo "CREATE DATABASE testdb;" | mysql -u root
```

We can enter these queries manually in the sql console or via a pipe with echo (which allows us to execute them via a script).

```
echo "CREATE USER 'test'@'localhost';" | mysql -u root

echo "SET password FOR 'test'@'localhost' = password('password');" | mysql -u root

echo "GRANT ALL ON testdb.* TO 'test'@'localhost' IDENTIFIED BY 'password' WITH GRANT OPTION;" | mysql -u root
```

We create a new user mysql, different from the user root (us) with which we launch the commands. He is assigned a password. We give it all the rights on the database 'testdb'. Why not use root and why give all the rights?

Services such as wordpress and phpMyAdmin need to connect to a database to function. It will be necessary to be able to provide them with the name of the database, a user sql with which to connect to mysql, which will itself have access to this database (and therefore rights on it), and the password for this user sql. . The php test script 'todo_list' uses the same principle, it retrieves the user and his password to get information from the database.

If we decide to use 'root' and assign a password to it, when we want to execute other commands, we will no longer be able to do so without entering our password (we will see errors like this):

```
  ERROR 1045 (28000): Access denied for user 'root'@'localhost' (using password: NO)
```

It is therefore easier to use a dedicated user. (And it's highly recommended for security reasons.)

srcs / tests / todo_list.php:

`$user = "test";` We modify the tutorial script so that it works with our setup. In the script testdb.sh we called the user 'test' @ 'localhost'.

`$password = "password";` The password that we defined is password.

`$database = "testdb";` Our database is called testdb.

We can now check that the web page displays the info of the db on localhost / todo_list.php.

Wordpress

(Excluding container)

We start by downloading Wordpress (we will recover a compressed archive from the official site):

```
wget http://fr.wordpress.org/latest-fr_FR.tar.gz
```

On décompresse et déplace le dossier 'wordpress' dans le dossier de sources.

```
tar -xzvf latest-fr_FR.tar.gz
```

Dans le container, on devra placer ce dossier 'wordpress' dans le dossier ou le serveur va choper les pages web, donc on ajoute une instruction au script 'install.sh'.

src/container_entrypoint.sh line:8 : `cp -r wordpress /var/www/localhost/wordpress`

Puis on peut re build l'image et lancer le container.

src/container_entrypoint.sh line:13-18 :

```
echo "CREATE DATABASE wordpress;" | mysql -u root
```

On crée une bdd wordpress, un user dédié avec son mot de passe comme pour le test todo_list.

On peut ensuite aller à localhost/wordpress qui normalement amène à l'install menu de wordpress. On nous demandera le nom de la db, le nom de l'user sql, le password, le nom du serveur etc : 'wordpress', 'wordpress', 'password', 'localhost', etc.

Wordpress va soit créer, soit nous demander de créer (et copier le contenu dedans) le fichier 'wp-config.php'. Il devra se trouver dans le dossier 'wordpress/'. Si on est pas redirigé automatiquement, on peut maintenant aller à :

- localhost/wordpress : la page d'acceuil de notre site wordpress
- localhost/wordpress/wp-admin : le tableau de bord de notre site wordpress

Wordpress est maintenant installé grâce au fichier wp-config. Mais si on sort du container, ce fichier est perdu et il faudra refaire l'installation de wordpress à chaque fois qu'on relance un container. Il faut donc copier le fichier wp-config obtenu (copier/coller par exemple) et le placer dans le dossier wordpress du repo pour que Wordpress soit toujours installé après un rebuild/rerun.

Par contre la structure de bdd créee par et pour Wordpress dans la base 'wordpress' sera perdue. La solution est de faire un fichier de sauvegarde de l'état de la bdd et l'importer à chaque rerun. La base sera donc recréee et réimportée à chaque rerun (cela prends plus ou moins de temps en fonction du poids de la base) ce qui permettra de conserver l'installation complète. Pour ça on va utiliser phpMyAdmin.

phpMyAdmin

(Hors container)

On télécharge phpMyadmin (on va également récupérer une archive compressée depuis le site officiel) :

```
wget https://files.phpmyadmin.net/phpMyAdmin/4.9.0.1/phpMyAdmin-4.9.0.1-all-languages.tar.gz
```

On peut la décompresser, renommer le dossier pour que le nom soit plus court (juste phpMyAdmin) et le placer dans les sources.

src/container_entrypoint.sh line:10 : `cp -r phpMyAdmin /var/www/localhost/phpMyAdmin`

On peut rebuild et run.

On peut aller à 'localhost/phpMyAdmin' pour accéder à l'écran de connexion PMA, il suffit de se connecter avec l'user mysql wordpress : 'wordpress', 'password'.

On peut visualiser nos bases de données dans l'onglet base de donnée. On peut visualiser toutes les tables de la bdd wordpress ainsi que leur contenu. Si on poste un commentaire sur wordpress, on pourra aussi le voir sur phpMyAdmin.

On va pouvoir récupérer le fichier de sauvegarde de l'état de la base de donnée wordpress. Il suffit de cliquer sur l'onglet 'Exporter', et récupérer le fichier 'wordpress.sql' (le navigateur le télécharge).

On peut maintenant placer ce fichier dans nos sources puis ajouter une instruction au run script pour importer la base.

src/container_entrypoint.sh line:19 : `mysql wordpress -u root < wordpress.sql`

SSL

https://youtu.be/NXyE3mayrtg

https://youtu.be/_UpuZ0Y3k-c

https://lehollandaisvolant.net/?d=2019/01/07/22/57/47-localhost-et-https

https://letsencrypt.org/fr/docs/certificates-for-localhost/

https://r.je/guide-lets-encrypt-certificate-for-local-development

Si on a des 'erreurs' de type connexion non privée en utilisant ssl, c'est ok. On n'est pas censés utiliser ssl pour du local, le but du sujet n'est pas de réussir à faire accepter notre certificat self-signed ou local au navigateur.

Autres

Pour afficher les logs du serveur en temps réel :

```
tail -f /var/log/nginx/access.log /var/log/nginx/error.log
```

Sources

https://github.com/matteoolefloch/ft_server

https://www.digitalocean.com/community/tutorials/how-to-install-linux-nginx-mariadb-php-lemp-stack-on-debian-10

https://howto.wared.fr/installation-wordpress-ubuntu-nginx/

https://www.itzgeek.com/how-tos/linux/debian/how-to-install-phpmyadmin-with-nginx-on-debian-10.html