# GÉREZ VOS CONTENEURS AVEC LE DOCKER COMPOSE

## Introduction

Docker Compose est un outil permettant de **définir le comportement de vos conteneurs** et d'**exécuter des applications Docker à conteneurs multiples**. La config se fait à partir d'un fichier YAML, et ensuite, avec une seule commande, vous **créez et démarrez tous vos conteneurs de votre configuration**.

## Installation du docker-compose

Docker Compose n'est pas installé par défaut et s'appuie sur le moteur Docker pour fonctionner. Au jour d'aujourd'hui, la dernière version de Docker Compose est la 1.24.0.

Voici la procédure à suivre pour **télécharger Docker Compose sous un environnement Linux** :

```
sudo curl -L "https://github.com/docker/compose/releases/download/1.24.0/docker-compo
```

```
sudo chmod +x /usr/local/bin/docker-compose
```

```
sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose
```

Vérifiez ensuite votre installation :

```
docker-compose --version
```

```
docker-compose version 1.24.0, build 0aa59064
```

Si vous n'avez pas d'erreur, alors vous pouvez poursuivre la lecture de ce chapitre.

# Définition des besoins du Docker Compose et amélioration du Dockerfile

Le but de cet article est d'améliorer notre ancienne application LAMP. Par la suite nous allons séparer le conteneur de notre application web par rapport au conteneur de notre base de données.

Au préalable, commencez par télécharger les sources du projet en cliquant ici et désarchivez ensuite le projet.

## Amélioration du Dockerfile

Profitons de cet article pour améliorer le Dockerfile de notre stack LAMP en réduisant son nombre d'instructions. Pour cela, on se basera sur une nouvelle image.

> **Conseil**
>
> Si vous souhaitez conteneuriser une application assez connue, alors je vous conseille de toujours fouiller dans le Hub Docker, afin de savoir si une image officielle de l'application existe déjà.

En cherchant dans le Hub Docker, j'ai pu dénicher les images adéquates, notamment :

- Une image officielle php avec le tag 7-apache
- Une image officielle mysql

Une fois que j'ai trouvé les bonnes images, je peux alors m'attaquer à la modification du Dockerfile.

Pour le moment, nous utiliserons ce Dockerfile seulement pour construire une image avec une couche OS, Apache et Php sans implémenter aucun service de base de données. Cette image se basera sur l'image officielle php avec le tag 7-apache qui vient déjà avec un OS (distribution Debian). Concernant l'image mysql nous l'utiliserons plus tard dans notre docker-compose.yml.

Dans le même dossier que vous avez désarchivé, créez un fichier Dockerfile et mettez dedans le contenu suivant :

```
FROM php:7-apache

LABEL version="1.0" maintainer="AJDAINI Hatim "

# Activation des modules php
RUN docker-php-ext-install pdo pdo_mysql

WORKDIR  /var/www/html
```

Buildez ensuite votre image avec la commande suivante :

```
docker build -t myapp .
```

## Les besoins pour notre Docker Compose

Avant de créer notre fichier docker-compose.yml, il faut auparavant **définir les comportements de nos conteneurs**.

### Nos besoins pour le conteneur de la base de données

On va débuter par la récolte des besoins du conteneur de la base de données. Pour celle-ci, il nous faudra :

- Un fichier sql pour créer l'architecture de notre base de données.
- Un volume pour stocker les données.

Avant de foncer tête baissée dans la création/modification de notre fichier sql, il toujours important de vérifier avant ce que nous propose la page Docker Hub de l'image mysql. En lisant sa description, les informations qui m'ont le plus captivé sont ses variables d'environnements qu'on peut surcharger, notamment :

- `MYSQL_ROOT_PASSWORD` : spécifie le mot de passe qui sera défini pour le compte MySQL root (**c'est une variable obligatoire**).
- `MYSQL_DATABASE` : spécifie le nom de la base de données à créer au démarrage de l'image.
- `MYSQL_USER` et `MYSQL_PASSWORD` : utilisées conjointement pour créer un nouvel utilisateur avec son mot de passe. Cet utilisateur se verra accorder des autorisations de super-utilisateur pour la base de données `MYSQL_DATABASE`.

Ces variables d'environnements vont nous aider à créer une partie de l'architecture de notre base de données.

Dans la description de l'image mysql, il existe une autre information très utile. Lorsqu'un conteneur mysql est démarré, il exécutera des fichiers avec des extensions `.sh`, `.sql` et `.sql.gz` qui se trouvent dans `/docker-entrypoint-initdb.d`. Nous allons profiter de cette information pour déposer le fichier `articles.sql` (disponible dans les sources téléchargées) dans le dossier `/docker-entrypoint-initdb.d` afin de créer automatiquement notre table SQL.

## Nos besoins pour le conteneur de l'application web

Concernant le conteneur de l'application web, nous aurons besoin de :

- Une communication avec le conteneur de la base de données.
- Un volume pour stocker les sources de l'application web.

Me concernant la seule information utile dans la description de la page Docker Hub de l'image php, est qu'il est possible d'installer et d'activer les modules php dans le conteneur php avec la commande `docker-php-ext-install` (C'est la commande utilisée dans notre Dockerfile afin d'activer le module pdo et pdo_mysql).

## Lancer les conteneurs sans le docker-compose

Histoire de vous donner une idée sur la longueur de la commande `docker run` sans utiliser le fichier docker-compose.yml. Je vais alors l'utiliser pour démarrer les différents conteneurs de notre application.

Premièrement je vais vous dévoiler, deux nouvelles options de la commande `docker run` :

- `-e` : définit/surcharge des variables d'environnement
- `--link` : ajoute un lien à un autre conteneur afin de les faire communiquer entre eux.

Voici à quoi va ressembler la commande `docker run` pour la **création du conteneur de la base de données** :

```
docker run -d -e MYSQL_ROOT_PASSWORD='test' \
-e MYSQL_DATABASE='test' \
-e MYSQL_USER='test' \
```

```
-e MYSQL_PASSWORD='test' \
--volume db-volume:/var/lib/mysql \
--volume $PWD/articles.sql:/docker-entrypoint-initdb.d/articles.sql \
--name mysql_c mysql:5.7
```

Voici à quoi va ressembler la commande `docker run` pour la **création du conteneur de l'application web** :

```
docker run -d --volume $PWD/app:/var/www/html -p 8080:80 --link mysql_c --name myapp_
```

Dans cet exemple, on peut vite remarquer que les commandes `docker run` sont assez longues et par conséquent elles ne sont pas assez lisibles. De plus, vous aurez à lancer cette commande pour chaque nouveau démarrage de l'application. Mais vous aurez aussi à gérer vos différents conteneurs séparément. C'est pour ces raisons, que nous utiliserons le fichier docker-compose.yml afin de **centraliser la gestion de nos multiples conteneurs d'une application Docker depuis un seul fichier**. Dans notre cas il va nous permettre d' **exécuter et définir les services, les volumes et la mise en relation des différents conteneurs** de notre application.

# Création du docker-compose

## Contenu du docker-compose

Commencez d'abord par créer un fichier et nommez le **docker-compose.yml**, ensuite copiez collez le contenu ci-dessous. Par la suite, plus bas dans l'article, je vais vous fournir les explications des différentes lignes de ce fichier :

```
version: '3.7'

services:
    db:
        image: mysql:5.7
        container_name: mysql_c
```

```
        restart: always
        volumes:
            - db-volume:/var/lib/mysql
            - ./articles.sql:/docker-entrypoint-initdb.d/articles.sql
        environment:
            MYSQL_ROOT_PASSWORD: test
            MYSQL_DATABASE: test
            MYSQL_USER: test
            MYSQL_PASSWORD: test

    app:
        image: myapp
        container_name: myapp_c
        restart: always
        volumes:
            - ./app:/var/www/html
        ports:
            - 8080:80
        depends_on:
            - db

volumes:
    db-volume:
```

## Explication du fichier docker-compose.yml

```
version: '3.7'
```

Il existe plusieurs versions rétrocompatibles pour le format du fichier Compose (voici la liste des versions de Docker Compose selon la version moteur Docker). Dans mon cas je suis sous la version 18.09.7 du moteur Docker, donc j'utilise la version 3.7.

---

```
services:
```

Dans une application Docker distribuée, différentes parties de l'application sont appelées **`services`**. Les services ne sont en réalité que des conteneurs. Dans notre cas nous aurons besoin d'un service pour notre base de données et un autre pour notre application web.

---

```
db:
    image: mysql:5.7
    container_name: mysql_c
    restart: always
    volumes:
        - db-volume:/var/lib/mysql
        - ./articles.sql:/docker-entrypoint-initdb.d/articles.sql
    environment:
        MYSQL_ROOT_PASSWORD: test
        MYSQL_DATABASE: test
        MYSQL_USER: test
        MYSQL_PASSWORD: test
```

Dans cette partie, on crée un service nommé `db`. Ce service indique au moteur Docker de procéder comme suit :

1. Se baser sur l'image `mysql:5.7`
2. Nommer le conteneur `mysql_c`
3. Le `restart: always` démarrera automatiquement le conteneur en cas de redémarrage du serveur
4. Définir les volumes à créer et utiliser (un volume pour exécuter automatiquement notre fichier sql et un autre pour sauvegarder les données de la base de données)
5. Surcharger les variables d'environnements à utiliser

---

```
app:
    image: myapp
    container_name: myapp_c
    restart: always
    volumes:
        - ./app:/var/www/html
    ports:
        - 8080:80
    depends_on:
        - db
```

Ici, on crée un service nommé `app`. Ce service indique au moteur Docker de procéder comme suit :

1. Se baser sur l'image nommée `myapp` qu'on avait construit depuis notre Dockerfile
2. Nommer le conteneur `myapp_c`
3. Le `restart: always` démarrera automatiquement le conteneur en cas de redémarrage du serveur
4. Définir les volumes à créer et à utiliser pour sauvegarder les sources de notre application
5. Mapper le port 8080 sur le port 80
6. Le `depends_on` indique les dépendances du service `app`. Ces dépendances vont provoquer les comportements suivants :
    - Les services démarrent en ordre de dépendance. Dans notre cas, le service `db` est démarré avant le service `app`
    - Les services s'arrêtent selon l'ordre de dépendance. Dans notre cas, le service `app` est arrêté avant le service `db`

---

```
volumes:
    db-volume:
```

Enfin, je demande au moteur Docker de me créer un volume nommé `db-volume`, c'est le volume pour stocker les données de notre base de données.

## Lancer l'application depuis docker-compose.yml

Pour être sur le même pied d'estale, voici à quoi doit ressembler votre arborescence :

```
|___ app
|   |___ db-config.php
|   |___ index.php
|
|___ validation.php
|___ articles.sql
|___ docker-compose.yml

|___ Dockerfile
```

Placez vous au niveau du dossier qui contient le fichier docker-compose.yml. Ensuite lancez la commande suivante pour **exécuter les services du docker-compose.yml** :

```
docker-compose up -d
```

Ici l'option `-d` permet d'**exécuter les conteneur du Docker compose en arrière-plan**.

Si vous le souhaitez, vous pouvez **vérifier le démarrage des conteneurs issus du docker-compose.yml** :

```
docker ps
```

**Résultat :**

```
CONTAINER ID        IMAGE               COMMAND                  CREATED
26bb6e0dd252        myapp               "docker-php-entrypoi…"   34 seconds ago
b5ee22310ebc        mysql:5.7           "docker-entrypoint.s…"   35 seconds ago
```

Pour seulement **lister les conteneurs du docker-compose.yml**, il suffit d'exécuter la commande suivante :

```
docker-compose ps
```

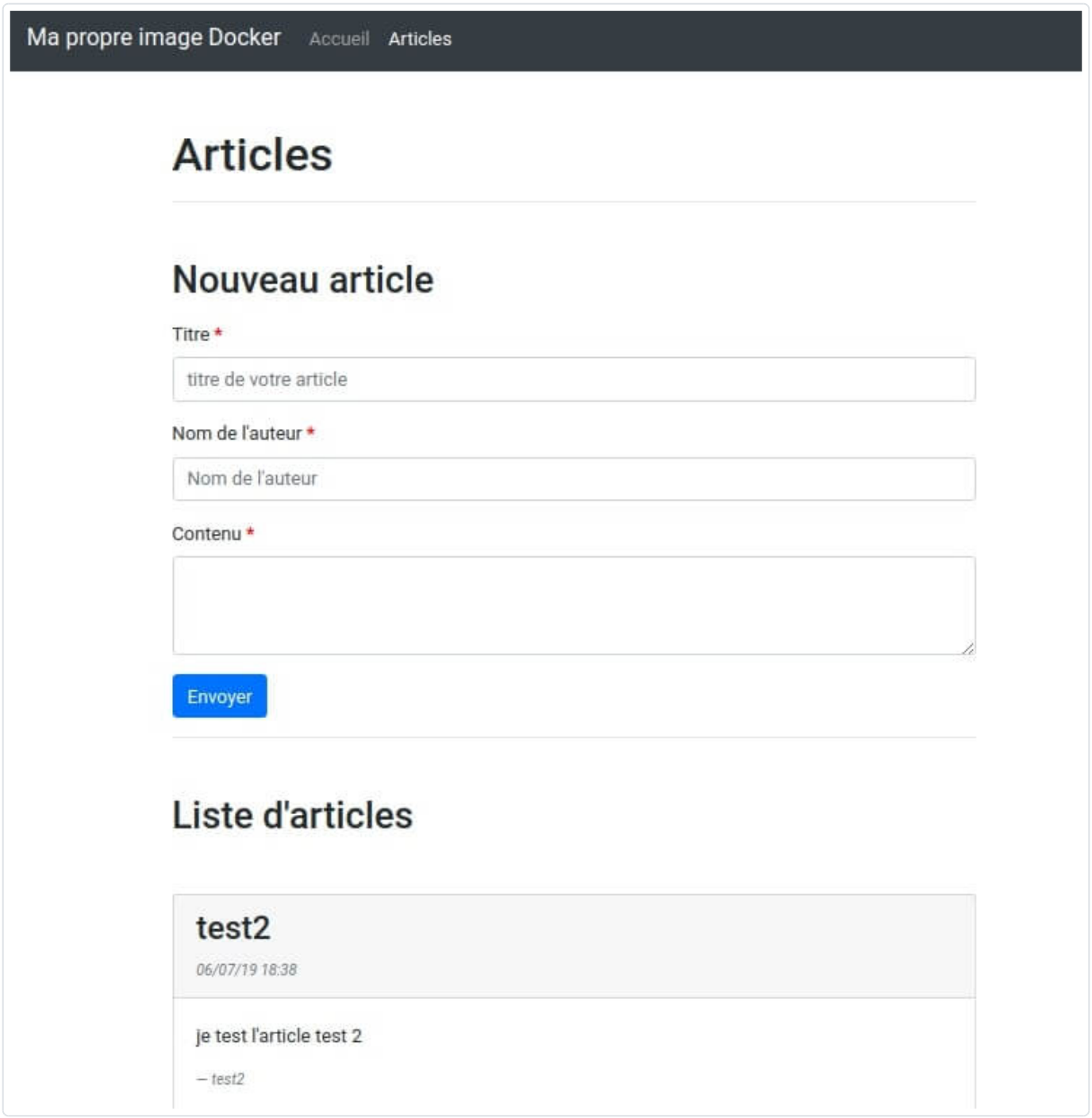
**Résultat :**

```
Name                Command               State          Ports
------------------------------------------------------------------------
myapp_c   docker-php-entrypoint apac ...   Up      0.0.0.0:8080->80/tcp
mysql_c   docker-entrypoint.sh mysqld      Up      3306/tcp, 33060/tcp
```

Si jamais vos conteneurs ne sont pas dans l'état `UP`, alors **vérifiez les logs des services de votre Docker Compose** en tapant la commande suivante :

```
docker-compose logs
```

Si tout c'est bien passé, alors visitez la page suivante http://localhost:8080/, et vous obtiendrez le résultat suivant :




Remplissez le formulaire de l'application, et **tuez les conteneurs du docker-compose.yml**, avec la commande suivante :

```
docker-compose kill
```

Relancez ensuite vos services, et vous verrez que vos données sont bel et bien sauvegardées.

## Détails de la communication inter-conteneurs dans les sources de l'application

Je ne vais pas trop rentrer dans les détails sur la partie réseau, car je vais rédiger un article qui sera dédié à cette partie. Mais sachez juste qu'un réseau bridge est créé par défaut, plus précisément c'est l'interface docker0 (`ip addr show docker0`), c'est un **réseau qui permet une communication entre les différents conteneurs**.

Donc **les conteneurs possèdent par défaut une adresse ip**. Vous pouvez récolter cette information grâce à la commande suivante :

```
docker inspect -f '{{.Name}} - {{range .NetworkSettings.Networks}}{{.IPAddress}}{{en
```

**Résultat :**

```
/myapp_c - 172.18.0.2
/mysql_c - 172.18.0.3
```

Pour faire communiquer notre application web avec la base de données, on peux utiliser dans le conteneur de l'app web soit l'ip, le nom du service (ici `db`) ou le nom du conteneur (ici `mysql_c`) de la base de données.

Si vous ouvrez le fichier **db-config.php** dans le dossier **app**, alors vous verrez la ligne suivante :

```
const DB_DSN = 'mysql:host=mysql_c;dbname=test';
```

Dans ces cas, j'ai utilisé le nom du conteneur de la base de données pour communiquer avec ce dernier.

# Conclusion

Je pense que vous l'aurez compris, le Docker Compose est un outil permettant de faciliter la gestion des applications Docker à conteneurs multiples, comme :

- Démarrer, arrêter et reconstruire des services
- Afficher le statut des services en cours d'exécution
- Diffuser la sortie des logs des services en cours d'exécution
- Exécuter une commande unique sur un service
- etc ...

Comme pour chaque fin de chapitre, je vous liste ci-dessous un récapitulatif de quelques commandes intéressantes du Docker Compose:

```
## Exécuter les services du docker-compose.yml
docker-compose up
    -d : Exécuter les conteneurs en arrière-plan

## Lister des conteneurs du Docker Compose
docker-compose ls
    -a ou --all : afficher aussi les conteneurs stoppés

## Sorties/erreurs des conteneurs du Docker Compose
docker-compose logs
    -f : suivre en permanence les logs du conteneur
    -t : afficher la date et l'heure de la réception de la ligne de log
    --tail=<NOMBRE DE LIGNE> = nombre de lignes à afficher à partir de la fin pour ch

## Tuer les conteneurs du Docker Compose
docker-compose kill

## Stopper les conteneurs du Docker Compose
docker-compose stop
    -t ou --timeout : spécifier un timeout en seconde avant le stop (par défaut : 10s

## Démarrer les conteneurs du Docker Compose
docker-compose start

## Arrêtez les conteneurs et supprimer les conteneurs, réseaux, volumes, et les image
docker-compose down
    -t ou --timeout : spécifier un timeout en seconde avant la suppression (par défau

## Supprimer des conteneurs stoppés du Docker Compose
docker-compose rm
```

```
    -f ou --force : forcer la suppression

## Lister les images utilisées dans le docker-compose.yml
docker-compose images
```