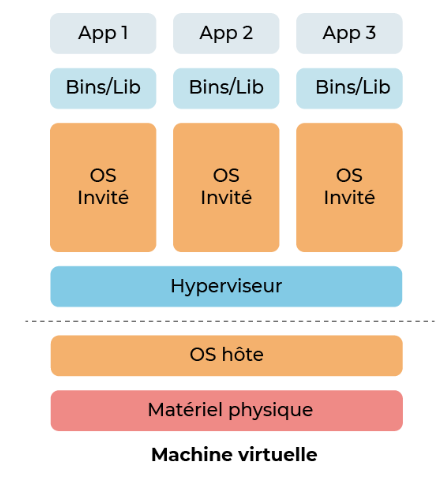
**Une machine virtuelle ?**



Lorsque vous utilisez une machine virtuelle (VM), vous faites ce qu’on appelle de la ***virtualisation lourde***.  En effet, vous recréez un système complet dans le système hôte, pour qu’il ait ses propres ressources.

**L'isolation avec le système hôte est donc totale**; cependant, cela apporte plusieurs contraintes :

* ❌Une machine virtuelle prend du **temps** à démarrer ;
* ❌Une machine virtuelle **réserve les ressources** (CPU/RAM) sur le système hôte.

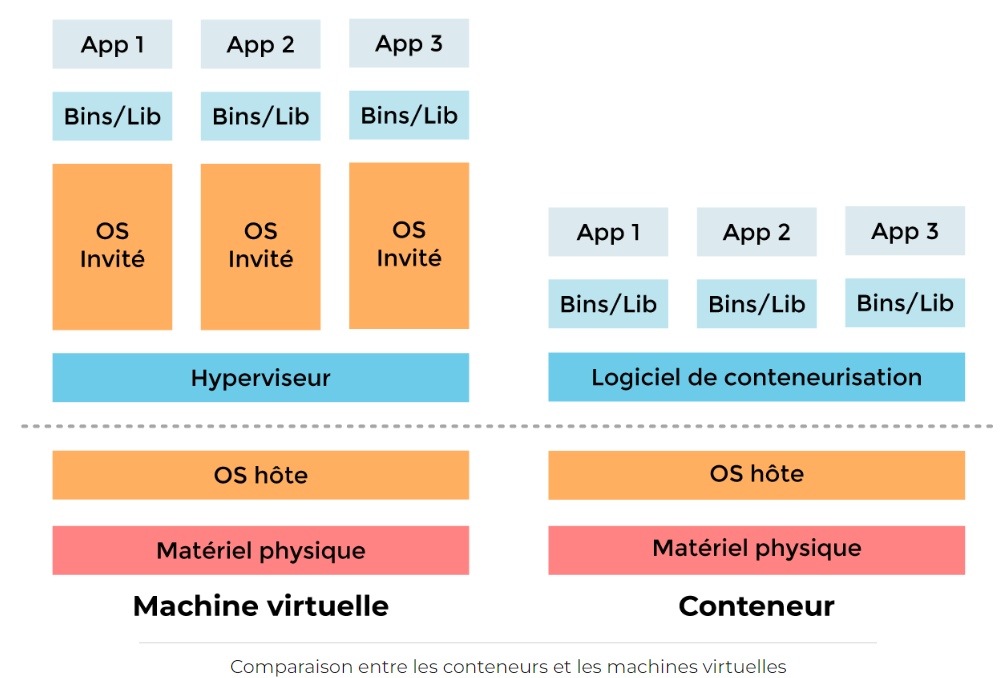
Mais cette solution présente aussi de nombreux avantages :

* ✅Une machine virtuelle est totalement **isolée** du système hôte ;
* ✅Les ressources attribuées à une machine virtuelle lui sont totalement **réservées**;
* ✅Vous pouvez installer **différents OS** (Linux, Windows, BSD, etc.).

Mais il arrive très souvent que l'application qu'elle fait tourner ne consomme pas l'ensemble des ressources disponibles sur la machine virtuelle. Alors est né un nouveau système de virtualisation plus léger : les **conteneurs**.

### Un conteneur ?

Un conteneur Linux est un **processus** ou un ensemble de processus isolés du reste du système, tout en étant **légers**.

Le conteneur permet de faire de la **virtualisation légère**, c'est-à-dire qu'il ne virtualise pas les ressources, il ne crée qu'une **isolation des processus**. Le conteneur partage donc les ressources avec le système hôte. 

Les conteneurs, au sens d'OpenVZ et LXC, apportent une **isolation importante des processus systèmes**; cependant, les ressources CPU, RAM et disque sont totalement partagées avec l'ensemble du système. Les conteneurs partagent entre eux le kernel Linux ; ainsi, il n'est pas possible de faire fonctionner un système Windows ou BSD dans celui-ci.

Voyons ensemble quelques avantages des conteneurs.

#### Ne réservez que les ressources nécessaires

#### Démarrez rapidement vos conteneurs

#### Donnez plus d'autonomie à vos développeur

## **Découvrez ce qu'est Docker**

Dans la vision Docker, un conteneur ne doit faire tourner qu'**un seul processus**. Ainsi, dans le cas d'une stack LAMP (Linux, Apache, MySQL, PHP), nous devons créer **3 conteneurs différents**, un pour Apache, un pour MySQL et un dernier pour PHP. Alors que dans un conteneur LXC ou OpenVZ, nous aurions fait tourner l'ensemble des 3 services dans un seul et unique conteneur.

Docker répond à une problématique forte dans le monde du développement.

Prenons un exemple : vous avez développé votre projet de Twitter Lite en local. Tout fonctionne bien, mais au moment de mettre en production, vous vous rendez compte que vous ne savez pas comment **déployer votre projet**. Un autre exemple : vous êtes dans une équipe de 10 personnes et chacun utilise un OS différent (Ubuntu, macOS, Windows, CentOS, etc.). Comment faire pour avoir un **environnement unifié et fonctionnel** chez l'ensemble des développeurs ?

Docker répond à ces problématiques en créant des conteneurs. Grâce à Docker, vous n'aurez plus de problème de différence d'environnement, et votre code marchera partout !

Les conteneurs Docker vous apportent aussi les notions de stateless et d'immutabilité.

#### Stateless et immutabilité

##### Stateless vs Stateful

Dans le monde de Docker, vous allez souvent entendre parler de **stateless et stateful**, deux catégories de conteneurs, et vous devez savoir à quoi correspond chaque catégorie.

Si nous prenons le cas d'une **base de donnée MySQL**, celle-ci est **stateful** car elle stocke un état. Ainsi, si vous éteignez et rallumez votre base de données, vous la retrouverez dans le même état de fonctionnement.

Stateless est donc l'inverse : l'application ne stocke pas d'état. Vous pouvez prendre le cas du **protocole HTTP**, celui-ci est **stateless**. À chaque nouvelle requête HTTP, les mêmes séries d'actions seront réalisées.

##### Un conteneur est immuable

**L'immutabilité** d'un conteneur est aussi importante. Un conteneur ne doit pas stocker de données qui doivent être pérennes, car il les perdra (à moins que vous les ayez pérennisées). Mais si vous souhaitez en local mettre une base de données dans un conteneur Docker, vous devez créer un volume pour que celui-ci puisse stocker les données de façon pérenne.

## Lancez votre premier conteneur en local

Nous allons commencer par découvrir l'**interface en ligne de commande**, qui nous permet de discuter avec le daemon Docker installé précédemment.

* Docker run hello-world 🡪 Pour démarrer votre premier conteneur du test, Quand vous utilisez cette commande, le **daemon Docker** va chercher si l'image hello-world est **disponible en local**. Dans le cas contraire, il va la **récupérer sur la registry Docker officielle**.
* docker run -d -p 8080:80 nginx 🡪 -d pour **détacher le conteneur** du processus principal de la console. Il vous permet de continuer à utiliser la console pendant que votre conteneur tourne sur un autre processus, -p pour définir **l'utilisation de ports**. Dans notre cas, nous lui avons demandé de transférer le trafic **du port 8080 vers le port 80 du conteneur**. Ainsi, en vous rendant sur l’adresse http://127.0.0.1:8080, vous aurez la page par défaut de Nginx.
* docker run -it node 🡪 permet de lancer un conteneur de NodeJs en mode exécutable node (-it interactif).
* docker ps 🡪 sert á savoir les conteneurs en état actifs.
* docker stop id 🡪 Celle-ci va détruire le conteneur et son contenu ; cependant, vous pouvez toujours recréer votre conteneur avec la commande docker run vue plus haut.
* docker pull node 🡪  **permet de récupérer des images sur le Docker Hub** sans pour autant lancer de conteneur.
* docker images 🡪 Vous pouvez aussi voir l'ensemble des images présentes en local sur votre ordinateur, avec la commande.

docker system prune :  permet de nettoyer le système en supprimant les données suivantes :

* L’ensemble des **conteneurs** Docker qui ne sont pas en status *running*;
* L’ensemble des **réseaux** créés par Docker qui ne sont pas utilisés par au moins un conteneur ;
* L’ensemble des **images** Docker non utilisées ;
* L’ensemble des **caches** utilisés pour la création d'images Docker.

## Créez votre premier Dockerfile

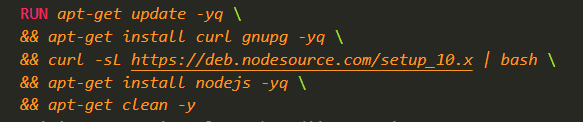
**Dans ce chapitre**, nous allons créer ensemble une image Docker, dans laquelle nous allons installer **Node.js**, ainsi que les différentes **dépendances de notre projet**.

Pour cela, nous allons créer **un fichier nommé Dockerfile .** Le Dockerfile permet de connaître notre recette pour faire **une pièce montée**. Alors, chaque argument de celle-ci crée un **nouvel étage sur la pièce montée**, nommé layer. Notre but étant de **limiter le nombre d'étages**, pour que notre pièce montée soit la plus **légère** et **performante** possible.

La première chose que vous devez faire est de créer un fichier nommé "Dockerfile", puis de définir dans celui-ci **l'image que vous allez utiliser comme base**, grâce à l'instruction FROM. Dans notre cas, nous allons utiliser une image de base Debian 9.



Ensuite, utilisez l'instruction RUN pour exécuter une commande dans votre conteneur.

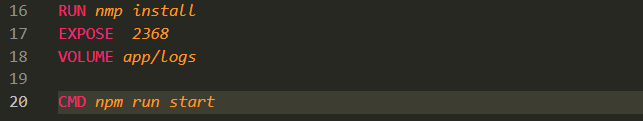


Limitez au maximum le nombre d'instructions RUN , afin de limiter le nombre de layers créées, et donc de réduire la taille de notre image Docker.

Puis, utilisez l'instruction  ADD  afin de copier ou de télécharger des fichiers dans l'image. Dans notre cas, nous l'utilisons pour ajouter les **sources de notre application** locale dans le dossier /app/ de l'image. Après utilisier WORKDIR qui permet de modifier le répertoire courant. La commande est équivalente à une commande cd en ligne de commande.



Puis, l'instruction RUN suivante permet d'installer le package du projet Node.js. Ensuite L'instruction EXPOSE permet d'indiquer le port sur lequel votre application écoute, et finalement L'instruction VOLUME permet d'indiquer quel répertoire vous voulez partager avec votre host. Pour conclure en utilise á la dernière ligne la commande : CMD qui permet à notre conteneur de savoir quelle commande il doit exécuter lors de son démarrage.



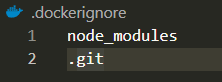
**En résumé, voici notre Dockerfile une fois terminé :**



Notre Dockerfile est maintenant prêt à fonctionner ! Cependant, il nous reste encore quelques petites modifications à faire.

**Fichier .dockerignore :** permet de ne pas copier certains fichiers et/ou dossiers dans notre conteneur lors de l’exécution de l'instruction ADD.

À la racine de votre projet, vous devez créer un fichier .dockerignore qui contiendra les lignes suivantes :



Vous pouvez maintenant créer votre première image Docker !

**$ docker build -t ocr-docker-build . 🡪** permet de créer votre image personnelle de docker.

L'argument -t permet de **donner un nom à votre image** Docker. Cela permet de retrouver plus facilement votre image par la suite. Le . est le répertoire où se trouve le Dockerfile ; dans notre cas, à la racine de notre projet.

Maintenant, vous pouvez lancer votre conteneur avec la commande docker run :

**$ docker run -d -p 2368:2368 ocr-docker-build 🡪** Vous retrouvez, dans le dossier logs  , les logs de votre application, et vous pourrez y accéder sur le port 2368  , soit via l'URL [http://127.0.0.1:2368](http://127.0.0.1:2368/).

## Utilisez des images grâce au partage sur le Docker Hub

Vous avez maintenant créé votre propre image via un Dockerfile, et vous souhaitez partager cette image avec vos collègues. Pour cela, il existe deux solutions :

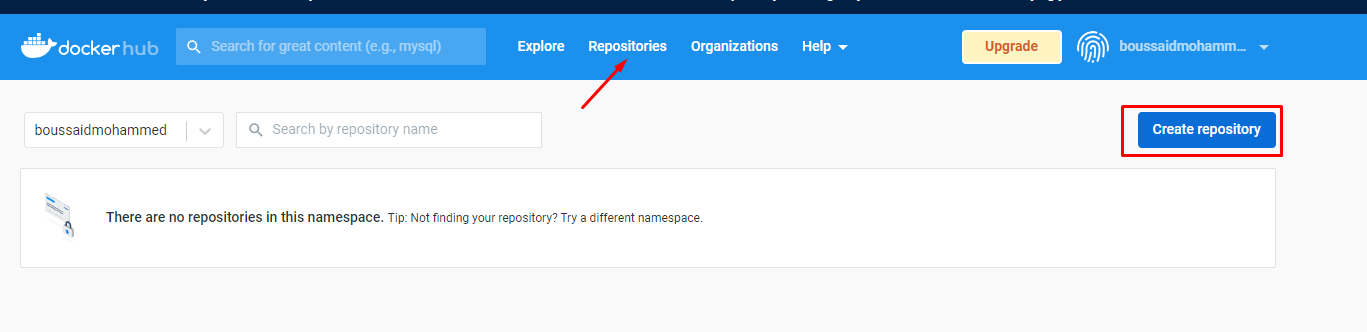
* soit vous partagez votre fichier Dockerfile à chacun de vos collègues, et vous leur demandez de créer eux-mêmes leur propre image avec un docker build  ;
* soit vous envoyez votre image sur votre propre **registry**.

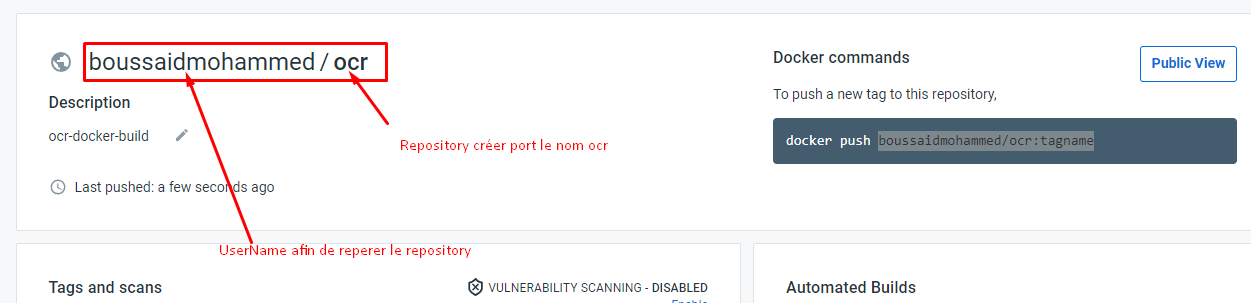
La première solution, bien qu'étant totalement fonctionnelle, force vos collègues à **recréer eux-mêmes chacune des images**, et cela peut prendre beaucoup de temps.

Pour améliorer cela, vous pouvez utiliser une **registry Docker** qui vous permet de partager directement votre image.

Pour cela, vous devez vous rendre sur le site <https://hub.docker.com/> et vous connecter avec le compte que vous avez créé lors de l'installation de Docker.

Puis, cliquez sur le lien *Create Repository*. Vous arrivez alors sur une page où vous devez saisir le **nom de votre image**, ainsi qu'une description.





Nous allons maintenant la **publier sur le Docker Hub**. Pour cela, nous devons utiliser plusieurs commandes.

tag ocr-docker-build:latest YOUR\_USERNAME/ocr-docker-build:latest  . Celle-ci va créer un **lien** entre notre image ocr-docker-build:latest créée précédemment et l'image que nous voulons envoyer sur le Docker Hub.



pour **envoyer votre image** vers le Docker Hub, vous allez exécuter la commande :



Si vous retournez maintenant sur la page Docker Hub de votre image, vous pourrez voir qu'il existe une première version de celle-ci.

Vous pouvez créer d'autres versions de votre image en remplaçant le :latest par une autre chaîne de caractères. Cependant, il faut bien faire attention, car l'image utilisée par défaut sera toujours l'image :latest

$ docker search node 🡪 cette commande permet de chercher une image sur internet. Docker pull après pour télécharger l’image voulue.

## Découvrez et installez Docker Compose

**Docker Compose va vous permettre d'orchestrer vos conteneurs, et ainsi de simplifier vos déploiements sur de multiples environnements. Docker Compose est un outil écrit en Python qui permet de décrire, dans un fichier YAML, plusieurs conteneurs comme un ensemble de services.**

Vous avez un nouveau projet de site avec [WordPress](https://fr.wordpress.org/). Pour **simplifier la gestion de l'infrastructure**, vous souhaitez déployer l'ensemble des composants dans des conteneurs Docker. Pour cela, nous allons avoir besoin de deux conteneurs :

* un conteneur **MySQL** ;
* un conteneur **WordPress**

$ docker-compose –version : **permet de savoir la version de docker compose installer sur votre machine.**

**Le CLI de Docker Compose et celui de Docker sont très proches**. Par exemple, si vous souhaitez récupérer l'ensemble des images décrites dans votre fichier docker-compose.yml et les télécharger depuis le Docker Hub, vous devez faire un docker-compose pull  . Du côté de Docker, la commande serait un docker pull  .

#### Démarrer une stack Docker Compose

Si vous souhaitez lancer la création de l'ensemble des conteneurs, vous devez lancer la commande docker-compose up (pour rappel, vous faites un docker run pour lancer un seul conteneur). Vous pouvez ajouter l’argument -d pour faire **tourner les conteneurs en tâche de fond**.

docker-compose ps 🡪 permet de savoir l’ensemble des conteneurs qui son en état fonctionnel.

docker-compose logs -f --tail 5 🡪 Celle-ci permet de voir l'ensemble des logs sur les différents conteneurs de façon continue, tout en limitant l'affichage aux 5 premières lignes.

docker-compose stop 🡪 pour **arrêter** une stack Docker Compose.

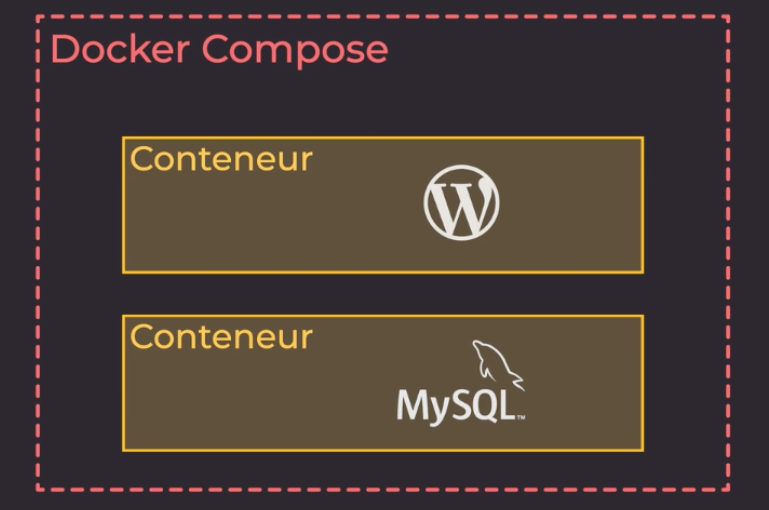
docker-compose up -d 🡪 l’ensemble de votre stack sera tout de suite à nouveau fonctionnel.

docker-compose down 🡪 afin de supprimer  l'ensemble de la stack Docker Compose, **supprimera également l’ensemble des ressources** créées.

docker-compose config 🡪 qui vous permettra de **valider la syntaxe de votre fichier docker-compose**.

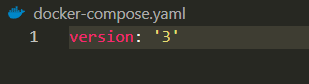
Revenant á notre projet WordPress, Pour cela, vous allez avoir besoin de plusieurs composants :

* Une base de données MySQL ;
* Le système de fichiers WordPress.



Vous devez commencer par créer un fichier docker-compose.yml à la racine de votre projet. Dans celui-ci, nous allons décrire l'ensemble des ressources et services nécessaires à la réalisation de votre POC.

Un fichier docker-compose.yml commence toujours par les informations suivantes :



L'ensemble des conteneurs qui doivent être créés doivent être définis sous l'argument services, Chaque conteneur commence avec un nom qui lui est propre (ici db), nous utilisons l’argument image qui nous permet de définir l'image Docker que nous souhaitons utiliser.



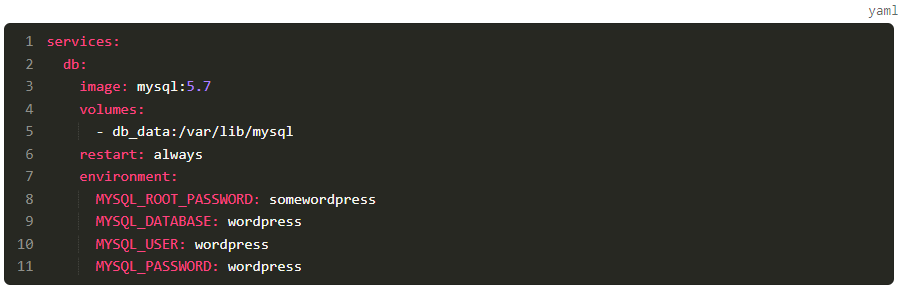
Pour rappel, nous avons vu précédemment que **les conteneurs Docker ne sont pas faits pour faire fonctionner des services stateful**, et une base de données est par définition un service stateful. Cependant, vous pouvez utiliser l'argument volumes qui vous permet de stocker l'ensemble du contenu du dossier /var/lib/mysql dans un disque persistant. Et donc, de pouvoir garder les données en local sur notre host.



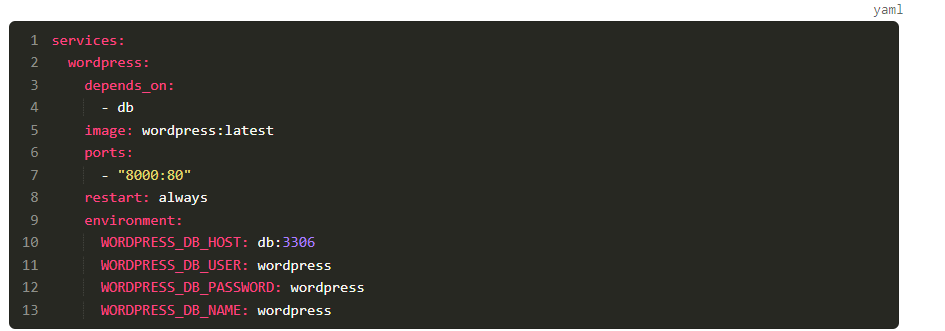
Un conteneur étant par définition monoprocessus, s'il rencontre une erreur fatale, il peut être amené à s'arrêter. Dans notre cas, si le serveur MySQL s'arrête, celui-ci redémarrera automatiquement grâce à l'argument restart: always.

Quand vous souhaitez donner des variables d'environnement à un conteneur, vous devez utiliser l'argument environment,

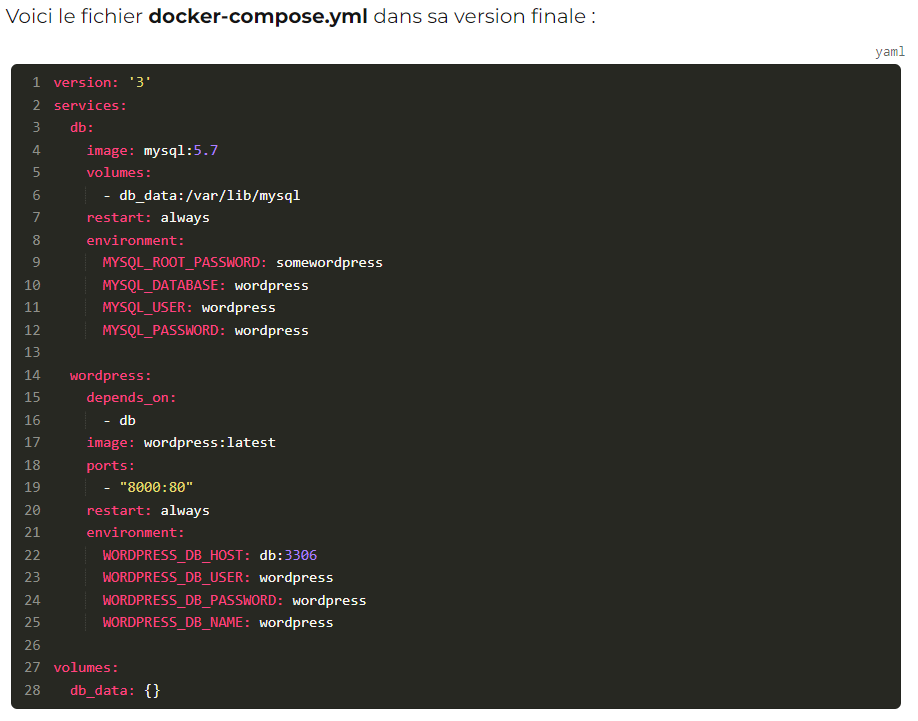
dans notre cas MySQL, nous allons donner au conteneur les valeurs des différents mots de passe et utilisateurs qui doivent exister sur cette base.



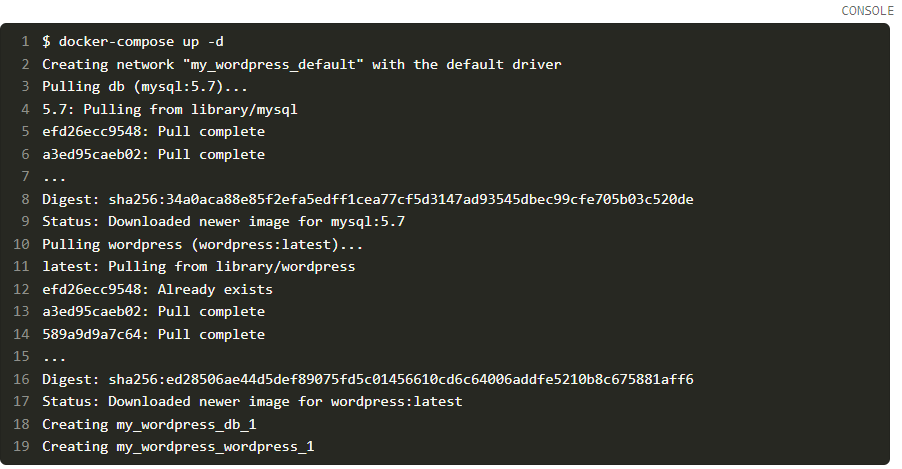
**Le second service WordPress :**



depends\_on  , nous permet de créer une **dépendance** entre deux conteneurs. Ainsi, Docker démarrera le service db avant de démarrer le service WordPress. Le second argument, ports  , permet de dire à Docker Compose qu'on veut exposer un **port** de notre machine hôte vers notre conteneur, et ainsi le rendre accessible depuis l'extérieur.



Quand vous lancerez vos conteneurs avec la commande docker-compose up -d  , vous devriez avoir le résultat suivant :



Puis celui-ci lance les deux conteneurs sur votre système ; dans notre cas, vous pourrez voir le résultat en vous ouvrant l'URL suivante dans votre navigateur :  http://127.0.0.1:8000