Déploiement de Wordpress avec docker-compose

Vous avez un nouveau projet de site avec WordPress. 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 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**.

Docker Compose permet d'orchestrer plusieurs conteneurs

En rappel

Vous connaissez maintenant les commandes principales pour utiliser une stack Docker Compose. Voici les commandes les plus importantes :

- `docker-compose up -d` vous permettra de **démarrer** l'ensemble des conteneurs en arrière-plan ;
- `docker-compose ps` vous permettra de voir le **statut** de l'ensemble de votre stack ;
- `docker-compose logs -f --tail 5` vous permettra d'afficher les **logs** de votre stack ;
- `docker-compose stop` vous permettra d'**arrêter** l'ensemble des services d'une stack ;
- `docker-compose down` vous permettra de **détruire** l'ensemble des ressources d'une stack ;
- `docker-compose config` vous permettra de **valider** la syntaxe de votre fichier `docker-compose.yml .`

Créez votre fichier docker-compose.yml

Structure de notre Docker Compose

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.

Décrivez votre premier service : db

Définissez la version de Docker Compose

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

```
version: '3'
```

L'argument `version` permet de spécifier à Docker Compose quelle **version** on souhaite utiliser, et donc d'utiliser ou pas certaines versions. Dans notre cas, nous utiliserons la version 3, qui est actuellement la version la plus utilisée.

Déclarez le premier service et son image

Nous allons maintenant déclarer notre premier service, et donc créer notre stack WordPress !

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 ; dans notre cas, notre premier conteneur se nommera `db` .

```
services:
  db:
    image: mysql:5.7
```

Puis, vous devez **décrire votre conteneur** ; dans notre cas, nous utilisons l'argument `image` qui nous permet de définir l'image Docker que nous souhaitons utiliser.

Nous aurions pu aussi utiliser l'argument `build` en lui spécifiant le chemin vers notre fichier Dockerfile ; ainsi, lors de l'exécution de Docker Compose, il aurait construit le conteneur via le Dockerfile avant de l'exécuter.

Définissez le volume pour faire persister vos données

```
services:
  db:
    image: mysql:5.7
    volumes:
      - db_data:/var/lib/mysql
```

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.

Cette description est présente grâce à la ligne `db_data:/var/lib/mysql` . `db_data` est un volume créé par Docker directement, qui permet d'écrire les données sur le disque hôte sans spécifier l'emplacement exact. Vous auriez pu aussi faire un `/data/mysql:/var/lib/mysql` qui serait aussi fonctionnel.

Définissez la politique de redémarrage du conteneur

```
services:
  db:
    image: mysql:5.7
    volumes:
      - db_data:/var/lib/mysql
    restart: always
```

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 .`

Définissez les variables d'environnement

```
services:
  db:
    image: mysql:5.7
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: password
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: password
```

L'image MySQL fournie dispose de **plusieurs variables d'environnement** que vous pouvez utiliser ; dans votre cas, nous allons donner au conteneur les valeurs des différents mots de passe et utilisateurs qui doivent exister sur cette base. Quand vous souhaitez donner des variables d'environnement à un conteneur, vous devez utiliser l'argument `environment` , comme nous l'avons utilisé dans le fichier `docker-compose.yml` ci-dessus.

Décrivez votre second service : WordPress

Dans le second service, nous créons un conteneur qui contiendra le nécessaire pour faire fonctionner votre site avec **WordPress**. Cela nous permet d'introduire deux arguments supplémentaires.

```
services:
  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    ports:
      - "8000:80"
    restart: always
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: password
      WORDPRESS_DB_NAME: wordpress
```

Le premier argument, `depends_on`, nous permet de créer une **dépendance** entre deux conteneurs. Ainsi, Docker démarrera le service db avant de démarrer le service WordPress.

Ce qui est un comportement souhaitable, car WordPress dépend de la base de données pour fonctionner correctement.

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.

Voici le fichier **docker-compose.yml** dans sa version finale :

```
version: '3'
services:
  db:
    image: mysql:5.7
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: somewordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress

  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    ports:
      - "8000:80"
    restart: always
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
      WORDPRESS_DB_NAME: wordpress

volumes:
  db_data: {}
```

Lancez votre stack Docker Compose

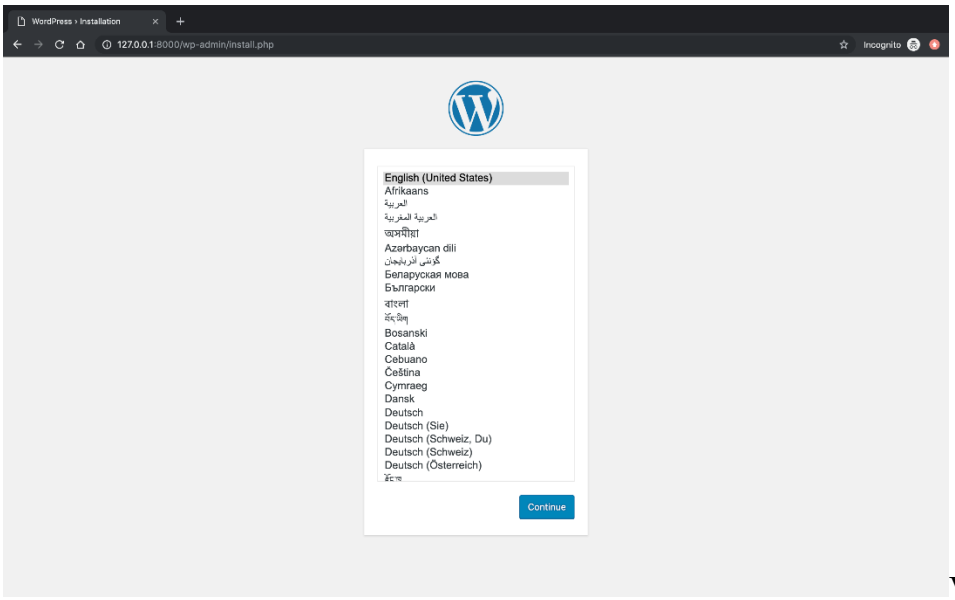
Quand vous lancerez vos conteneurs avec la commande `docker-compose up -d`, vous devriez avoir le résultat suivant :

```
$ docker-compose up -d
Creating network "my_wordpress_default" with the default driver
Pulling db (mysql:5.7)...
5.7: Pulling from library/mysql
efd26ecc9548: Pull complete
a3ed95caeb02: Pull complete
...
Digest:
sha256:34a0aca88e85f2efa5edff1cea77cf5d3147ad93545dbec99cfe705b03c520de
Status: Downloaded newer image for mysql:5.7
Pulling wordpress (wordpress:latest)...
latest: Pulling from library/wordpress
efd26ecc9548: Already exists
a3ed95caeb02: Pull complete
589a9d9a7c64: Pull complete
...
```

```
Digest:
sha256:ed28506ae44d5def89075fd5c01456610cd6c64006addfe5210b8c675881aff6
Status: Downloaded newer image for wordpress:latest
Creating my_wordpress_db_1
Creating my_wordpress_wordpress_1
```

Lors de l'exécution de cette commande, Docker Compose commence par vérifier si nous disposons bien en local des images nécessaires au lancement des stacks. Dans le cas contraire, il les télécharge depuis une registry, ou les build via un `docker build`.

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`.



Votre site WordPress fonctionnel !

En résumé

Vous savez maintenant utiliser les commandes de base de Docker Compose, et créer un fichier `docker-compose.yml` pour orchestrer vos conteneurs Docker.

Pour rappel, voici les arguments que nous avons pu voir dans ce chapitre :

- `image` qui permet de spécifier l'**image source** pour le conteneur ;
- `build` qui permet de spécifier le **Dockerfile source** pour créer l'image du conteneur ;
- `volume` qui permet de spécifier les **points de montage** entre le système hôte et les conteneurs ;
- `restart` qui permet de définir le comportement du conteneur **en cas d'arrêt** du processus ;
- `environment` qui permet de définir les **variables d'environnement** ;
- `depends_on` qui permet de dire que le conteneur **dépend** d'un autre conteneur ;
- `ports` qui permet de définir les **ports** disponibles entre la machine host et le conteneur.