



# Inception Tutorial



## Objectifs d’apprentissage

1. Comprendre **Docker** et **Compose**.
2. Découvrir les différents points du **sujet** d’Inception.
3. Mettre en place le container **NGINX**
4. Mettre en place le container **MariaDB**
5. Mettre en place le container **WordPress**
6. Relier les containers avec **Compose**
7. Comprendre les volumes de **Compose**
8. Finaliser le projet

## 1. Comprendre Docker :

L’avantage de Docker est clair, il **résout** l’un des plus **gros problèmes des développeurs**:

Se retrouver à créer un super programme sur son ordinateur, et se rendre compte qu’il ne fonctionne que sur son propre ordinateur. Pour l’utiliser ailleurs il va falloir installer les dépendances requises

Souvenez vous ce **super programme** que vous aviez trouvé sur **Github**, vous l’installez comme prévu par le tutoriel du ReadMe , mais l’installation **crash** indiquant “*You have missing dependencies*”, ou encore “*This version of this file is not compatible with your OS*” ☹

Alors oui, vous pouvez rester un **bon développeur** et proposer un super **script qui installera ces dépendances**, mais vous ne pouvez pas prévoir que l’utilisateur soit sur Mac, Linux ou encore qu’il ait une version d’OS si vieille que celle ci ne connait même pas vos dépendances !

► ☼ Les types de problèmes que corrige Docker

| Passer 4 heures sur du **débug** d’un logiciel qui n’est pas le notre et se rendre compte que nous n’y arriverons pas, a tendance à rendre fou

C’est ce qu’il a du arriver à **Solomon Hykes**, un **franco-américain** qui a finit par se demander s’il était possible de trouver une **solution** à ce genre de problème. En réponse à cela, il sort **Docker** le **20 mars 2013**.

Rappel du Wiki de Docker: **Docker est un outil qui peut emballer une application et ses dépendances dans un conteneur isolé.**

► ☼ L’Histoire de Docker

Tout ça semble tout beau mais en fait, si ça existe déjà avec les VM,

### Pourquoi les développeurs utilisent Docker ?

Le grand avantage de **Docker** est la possibilité de modéliser chaque conteneur sous la forme d’une image que l’on peut stocker localement.

Un **conteneur** est une machine virtuelle sans noyau.

Ce que j’appelle **noyau** est tout l’ensemble du système permettant à la machine virtuelle de fonctionner, l’OS, le coté graphique, réseau, etc...

En d’autres termes, un conteneur ne contient que l’application et les dépendances de l’application.

### Docker Hub:

**Docker** met a disposition une sorte d’App Store, contenant des images (conteneur) de milliers de personnes, simplifiant encore plus son usage

Nous aurions besoin du container Docker qui installe de lui même NGINX.

☺ Ca tombe bien, étant connu, l'image NGINX à été publié par **NGINX sur le Docker Hub!**

Regardons un exemple de ce à quoi pourrais ressembler une image NGINX:

```
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"]
```

Ce fichier est un **Dockerfile**, c'est le nom du fichier principal de vos images **Docker**.  
Qui dit **Dockerfile**, dit nouveau langage de programmation, mais ne fuyez pas, il s'agit de connaitre ces quelques mots-clefs.

Quelques mots-clefs Dockerfile:

- **FROM**
- **RUN**
- **COPY**
- **EXPOSE**
- **ENTRYPOINT**

Ici un site détaillant les différents types de mots-clefs utilisables dans Docker.

| △ Pour continuer vous devez avoir bien compris le principe de Docker.

Docker-Compose :

Maintenant que vous avez compris la réelle utilité de Docker, il s'agit de comprendre une fonctionnalité de **Docker** appelée Compose.

Voici ce qu'explique la doc de Docker sur Compose:

| **Docker Compose** est un **outil qui a été développé pour aider à définir et à partager des applications multi-conteneurs**.  
Avec **Compose**, nous pouvons créer un fichier **YAML** pour définir les services et, à l'aide d'une seule commande, tout mettre en **route** ou tout **démonter**.

**Compose** permettrait donc de gérer des applications qui utilisent plusieurs containers et de communiquer entre eux.

*Vous devez également rendre un Makefile qui doit se trouver à la racine de votre répertoire. Il doit permettre de mettre en place toute votre application (c'est-à-dire build les images Docker via docker-compose.yml)*

Un texte sur fond bleu comme celui ci est toujours tiré du sujet **d'Inception** de 42.

Ici le *sujet* demande de build les différentes images via un fichier **.yml**, cela tombe bien ce genre de fichier est appelé **YAML**, c'est le fameux format indiqué plus haut par la Doc de Docker pour utiliser **Docker-Compose**.

Ce projet consistera à vous faire mettre en place une mini-infrastructure de différents services en suivant des règles spécifiques.

Maintenant c'est certain, le projet consiste à relier plusieurs image **Docker**, et pouvoir les lancer ensemble, sans pour autant, qu'elles perdent leur indépendance.  
Tout ca grace à **Docker-Compose** qui est prévu pour ce genre d'utilisation.

Mais ca dans la vie de tous les jours, c'est utile ?

En fait, l'utilisation de **Docker-Compose** prend tout son sens dans une infrastructure informatique.

Imaginez, vous lancez votre startup, **Bananeo**.  
Vous auriez besoin de mettre en place un site internet, vous créez votre image NGINX (*ou vous récupérez celle de NGINX sur le **DockerHub***).  
Vous avez désormais un site internet fonctionnel pour votre entreprise.

| Évidemment cela risque de prendre plus de temps que 2 phrases à mettre en place

Maintenant que **Bananeo** compte une dizaine d'employés, il serait sympa de mettre en place une badgeuse à l'entrée pour éviter trop de travail à Manu ♂ de la sécurité.

Vous créez donc une nouvelle image **Docker**, spécialement conçue pour enregistrer vos salariés dans une **base de données** et étant reliée directement à votre badgeuse.  
Vous déployez également rapidement une autre image gérant un site internet intra.Bananeo.fr permettant à vos salariés de gérer leurs heures de travail.

Et Yop ! Tout fonctionne. Mais votre badgeuse ne communique jamais avec le site de l'intra, et il serait sympa de pouvoir préciser à vos employés leurs retards, surtout pour Éric qui trouve toujours la bonne excuse depuis 2 semaines.

C'est la qu'intervient Compose!

Vous allez mettre en place un fichier **.yml**, rappelez vous que c'est le format **YAML** qui permet de donner les instructions à **Compose** sur comment gérer ces différentes images.

```
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 meme nom que le service d'apres 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
```

☹ Non, ce docker-compose.yml ne fonctionne pas sur Inception.

Ce fichier **.yml** permet de donner les instructions à **Compose** pour gérer 3 images mariadb, NGINX et WordPress.  
Tel un **Dockerfile** qui doit obligatoirement commencer par **FROM** suivi de la version, votre fichier **YAML** doit commencer par la version de **Compose**, référez vous aux versions actuelles.

## 2. Comprendre le sujet:

► **SUJET À TÉLÉCHARGER**

Chaque service devra tourner dans un container dédié.

Voici les différents containers à mettre en place d'après le sujet :

- ☐ NGINX *(avec TLS v1.2)*
- ☐ WordPress *(avec php-fpm configuré)*
- ☐ MARIADB *(sans NGINX)*

Voici les deux volumes à mettre en place d'après le sujet :

- ☐ Volume contenant votre **base de données WordPress**
- ☐ Volume contenant les **fichiers** de votre **site WordPress**

Ces volumes doivent être disponibles à partir du dossier `/home/<login>/data` de la machine hôte utilisant **Docker**.

Les volumes **Compose** présentent plusieurs avantages comme pouvoir être gérés à l'aide des commandes de la CLI de Docker ou de son API ou encore pouvoir être partagés de manière plus sûre entre plusieurs conteneurs.

+ d'info sur la doc de Docker ici

Nous devons également mettre en place :

- ☐ Un **docker-network** qui fera le lien entre vos containers.

Les utilisateurs à créer dans notre base de données WordPress:

- ☐ Un utilisateur **Admin** (ne doit pas s'appeler admin)
- ☐ Un utilisateur standard

Pour des questions de lisibilité, il faudra configurer notre nom de domaine afin qu'il pointe vers **notre adresse IP locale**. Ce nom de domaine sera <login>.42.fr

Afin de réaliser ces différentes tâches, le sujet apporte plusieurs autres précisions :

Le tag latest est **interdit**.

Lorsque vous précisez une dépendance à installer, vous devez préciser sa version à installer, l'avantage du tag *latest* est clair, installer la **dernière version** d'une dépendance.  
Cependant *latest* a également un gros désavantage, il peut apporter des **problèmes de compatibilité** sur le temps.

**Exemple:** Aujourd'hui vous pourriez tout configurer de manière a ce que les dernières versions fonctionnent correctement entre elles, mais imaginez que 2 ans après quelqu'un souhaite tester votre logiciel, il se pourrait que certaines dépendances aient évoluées et fonctionnent autrement entre elles, c'est pour cela qu'il est préférable d'indiquer la version à utiliser.

Aucun mot de passe ne doit être présent dans vos **Dockerfiles**.

Faut-il vraiment expliquer ce point la ? ☹

Il faut dire que c'est une des plus grosses erreurs des devs utilisant Github.

Oui, il arrive que des développeurs push dans leur repo certaines informations sensibles, telles que des clefs d'API ou encore des mots de passe en clair

Évidemment ça reste **l'erreur number 1** à ne pas faire.

Si cela vous arrive, ne faites pas la deuxième erreur de re-push par dessus en supprimant votre clef ou votre mot de passe, les commits de Github finiront par vous trahir ✕  
Pensez plutôt à rapidement supprimer votre commit.

L'utilisation des variables d'environnement est **obligatoire**.

Cette consigne va avec la précédente, vous devez donc utiliser l'environnement de votre machine.  
Cela pourrait sembler utile pour stocker les mots de passe ou autres

Nous savons désormais comment stocker notre environnement.

Dans le sujet, nous trouvons également un exemple d'a quoi pourrait ressembler ce fichier `.env` grace à un `cat srcs/.env` :

```
DOMAIN_NAME=w11.42.fr
# certificates
CERTS=./XXXXXXXXXX
# MYSQL SETUP
MYSQL_ROOT_PASSWORD=XXXXXXXXXX
MYSQL_USER=XXXXXXXXXX
MYSQL_PASSWORD=XXXXXXXXXX
[...]
```

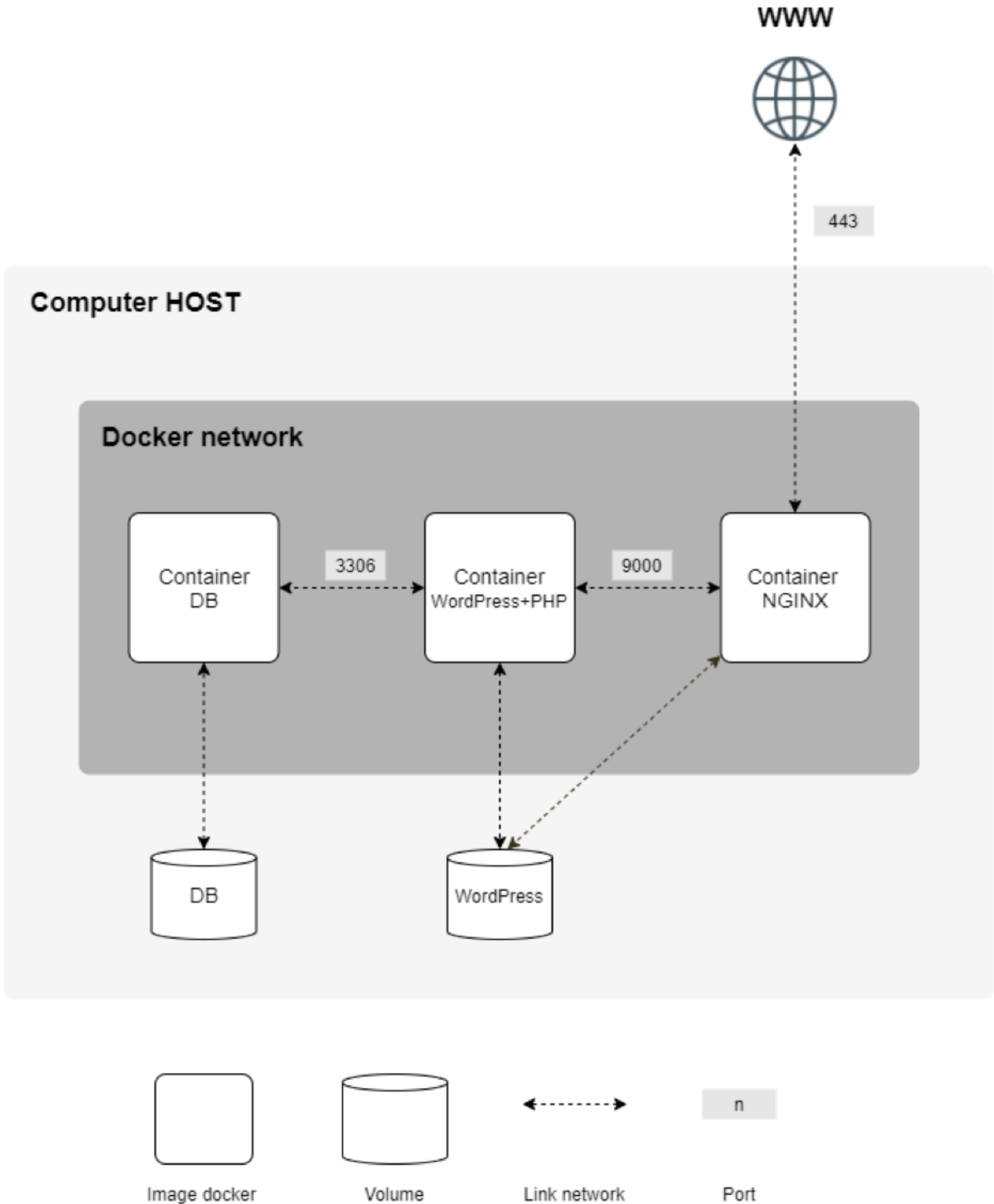
Grace à ce `cat` nous pouvons imaginer les infos dont nous aurons besoin pour mettre en place notre projet.

Votre container **NGINX** doit être le seul point d'entrée de votre infrastructure par le port **443** uniquement en utilisant le protocole **TLSv1.2** ou **TLSv1.3**

Le seul point d'entrée de notre Compose doit être par le container de NGINX, passant par le port **443**.

C'est à dire que le seul porte qu'ouvrira Docker-Compose sur votre machine sera le **443** (*tips: c'est le port qui permet l'accès par https:// , et le 80 par http://*)

Ce **schéma** fournit dans le sujet devrait éclaircir les choses:



Ici, on retrouve les ports qui permettront à **WordPress** de communiquer en interne avec la base de données (pour pouvoir stocker des nouvelles pages ou autres) et **NGINX** qui communique avec **WordPress** (qui lui dira quoi afficher sur le serveur web).

Le container **MariaDB** stocke ses informations (*base de donnée*) dans le volume correspondant et **WordPress** (*site wordpress*) dans le sien.

Il est indiqué que nous devons utiliser le protocole **TLSv1.2** ou **TLSv1.3**

### Mais qu'est ce que TLS ?

**Wikipédia** nous indique :

La **Transport Layer Security (TLS)** ou « Sécurité de la couche de transport » est un protocole de sécurisation des échanges par réseau informatique, notamment par Internet.

TLS permet :

- l'authentification du serveur
- la confidentialité des données échangées (*ou session chiffrée*)
- l'intégrité des données échangées
- de manière optionnelle, l'authentification du client (*mais dans la réalité celle-ci est souvent assurée par la couche applicative*)

Je ne sais pas vous mais **TLS** me semble ressembler de très près a du **SSL**, ce fameux protocole qui rajoute un **cadenas vert** quand vous accédez à un site **sécurisé**, comme la plupart des sites aujourd'hui.



Ma question est donc rapidement,

Quelle est la différence entre TLS et SSL ?

SSL et TLS sont deux protocoles qui permettent l'authentification, et le chiffrement des données qui transitent entre des serveurs.

En fait le SSL est le prédécesseur du TLS. Au fil du temps, de nouvelles versions de ces protocoles ont vu le jour pour faire face aux vulnérabilités et prendre en charge des suites et des algorithmes de chiffrement toujours plus forts, toujours plus sécurisés

L'histoire du SSL/TSL :

Initialement développé par Netscape, le SSL sort en 1995 dans sa version SSL 2.0 (le SSL 1.0 n'étant jamais sorti).

Mais après la découverte de plusieurs vulnérabilités en 1996, la version 2.0 est vite remplacée par le SSL 3.0.

Les versions 2.0 et 3.0 sont parfois libellées ainsi : SSLv2 et SSLv3.

Basé sur le SSL 3.0, le TLS est introduit en 1999 comme la nouvelle version du SSL.

Le protocole TSL est donc simplement le remplaçant du SSL. Il corrige certaines vulnérabilités de sécurité dans les anciens protocoles SSL.

Nous devons donc rendre notre système sécurisé, par un protocole semblable au SSL, le TSL.

Avant de commencer notre projet, regardons le dernier document fourni avec le sujet, la structure attendue de notre projet :

```
$> 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
drwxrwxr-x 5 wil wil 4096 avril 42 20:42 .
drwxrwxr-x 3 wil wil 4096 avril 42 20:42 ..
drwxrwxr-x 4 wil wil 4096 avril 42 20:42 bonus
drwxrwxr-x 4 wil wil 4096 avril 42 20:42 mariadb
drwxrwxr-x 4 wil wil 4096 avril 42 20:42 nginx
drwxrwxr-x 4 wil wil 4096 avril 42 20:42 tools
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
```

Évitons de négliger ce genre d'information fournie avec le sujet, elle peut déjà nous aider à démarrer le projet.

Cette structure nous montre clairement qu'il doit y avoir un dossier principal srcs contenant notre fameux docker-compose.yml que nous avons vu juste au dessus, notre fichier .env, ainsi qu'un dossier requirements qui contiendra nos différents containers.

Rappel des 3 containers à mettre en place :

- ☐ NGINX (avec TLS v1.2)
- ☐ WordPress (avec php-fpm configuré)
- ☐ MARIADB (sans NGINX)

Comme je l'expliquais au début, chaque container est représenté par un dossier portant son nom.

Dans chaque dossier de container doit obligatoirement se trouver son Dockerfile, sans quoi Docker serait perdu.

Rappelez vous, en + du Dockerfile nous pouvons rajouter différents fichiers/dossiers de config qu'il serait ensuite intéressant de copier dans notre container grâce au Dockerfile et son mot-clef COPY.

Le schéma fournit avec le sujet nous indique directement quels fichiers il serait intéressant de fournir avec nos containers, les voici:

- Un fichier conf qui pourrait contenir le fichier de configuration du container (la config de NGINX pour son container associé par exemple)
- Un fichier .dockerignore. Tout comme le fichier .gitignore, il précise à Docker des fichiers à ne pas regarder, car ils n'auront surement aucune utilité pour Docker.
- Un dossier tools permettant surement de stocker les autres outils dont nous pourrions avoir besoin.

Nous pouvons déjà reproduire la structure de fichiers demandée histoire d'y voir plus clair

Maintenant que nous comprenons ce que le sujet attend de nous, il serait interessant de se demander :

Par où commencer ?

Une fois que nous avons nos dossiers et nos fichiers requis, nous devrions commencer par créer un des trois containers requis.

Inutile de s'attaquer à la partie Compose pour l'instant, comment voudriez-vous relier 3 containers entre eux si vous n'en possédez aucun?

3. Container NGINX :

Commençons par notre container le moins flou, NGINX.

NGINX permet de mettre en place un serveur Web.

Nous allons commencer par étape afin de comprendre au mieux comment utiliser et se balader avec Docker.

La plupart des développeurs serait directement passée par là, mais ici c'est nous l'intéressant, afin de savoir la recréer nous même.

Si vous avez bien suivi, vous devriez savoir quelle ligne écrire en premier, du moins le premier mot clef du Dockerfile.

► SPOIL

Pour des raisons de performance, les containers devront être build au choix : soit sous Alpine Linux avec l'avant-dernière version stable, soit sous Debian Buster.

Pour l'OS, nous avons donc le choix entre *debian:buster* ou *alpine:X.XX* (vérifiez l'avant-dernière version stable en date, ici).

Quelle différence entre Alpine et Debian?

**Alpine Linux** est une distribution **Linux** légère, orientée sécurité, elle contient le moins de fichiers et outils possibles afin de laisser la possibilité au développeur de les installer par lui même si besoin.  
**Debian** est le système d'exploitation universel. Les systèmes Debian utilisent actuellement le noyau Linux ou le noyau **FreeBSD**.

Pour ma part, étant plus à l'aise avec ce système, je vais utiliser **Debian**.

La suite du tutoriel est représenté pour **Debian**. Mais vous devriez facilement pouvoir adapter cela sur **Alpine**.

Nous pouvons donc commencer par écrire : `FROM debian:buster` dans notre **Dockerfile**.

Maintenant j'ai envie de vous dire, si cette ligne est la seule ligne obligatoire, notre container pourrait-il démarrer ?

Nous avons indiqué un OS à installer, nous devrions pouvoir le lancer.

Le plus sympa, c'est que **Docker** peut nous permettre de lancer une "machine virtuelle", mais en lui précisant, on peut également accéder à son terminal ! Utile pour voir ce qu'il y'a dedans ou effectuer nos propres tests.

Pour cela, il faut connaitre les commandes indispensables aux containers de Docker.

Les commandes essentielles d'un container Docker :

Un container **Docker** doit être **build** avant d'être lancé.

C'est pendant le **build** que vous pourrez obtenir des infos sur des potentielles erreurs que vous auriez fait dans le **Dockerfile**.

Chaque commande Docker commence par le mot clef `docker`

Build un container Docker : `docker build`

☺ J'espère que vous n'oseriez pas tenter cette commande sans l'avoir parfaitement comprise !

De toute manière, la commande n'est pas fonctionnelle, docker demande obligatoirement un chemin ou se trouve le Dockerfile de l'image à build.

Dans notre cas, ce serait `docker build srcs/requirements/nginx/`

Ou encore plus simple, `docker build .` en vous trouvant directement dans le dossier NGINX.

Vous pouvez également préciser un nom à votre build, avec le flag `-t`

Exemple : `docker build -t nginx .`

Si vous obtenez une erreur de type `Cannot connect to the Docker daemon`, vérifiez que Docker est bien ouvert et qu'il fonctionne.

Connaitre les images actuelles (après un build réussi) : `docker image ls`

Vous devriez visualiser votre première image !

☺ Cependant celle ci n'a pas de nom en dessous de **REPOSITORY**, il est indiqué *<none>*, meme si on peut relier cette image à son ID présent dans la quatrième colonne, le sujet nous demande que le nom de l'image porte le nom du container associé, ici **NGINX**.

Et on peut faire cela ! Il suffit simplement de préciser quand on build l'image, son nom, grace au flag `-t`

Démarrer une image (run) : `docker run <image_name>`

Vous devrez indiquer le nom de votre image à run.

En précisant `-it` avant le nom de votre image, vous accéderez directement au terminal de votre container à son lancement.

Connaitre les containers actuellement lancés : `docker ps`

Vous pouvez même obtenir les containers Docker stoppés en rajoutant le flag `-a`

J'ai créé un **mini script bash** qui permet de build un container en faisant `docker build -t inception .`

Après avoir réussi, il propose de le lancer avec `-it`, ce qui permet d'accéder directement à la console du container.

Mettez simplement ce bout de texte dans un fichier `execute.sh` et exécutez le avec bash.

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

Nous sommes maintenant prêt pour tenter de lancer notre mini **Dockerfile**.

Commençons par **build** notre container.

Dans le dossier contenant le **Dockerfile NGINX** : `docker build -t nginx .`

Maintenant lançons le **container** que nous venons de **build** : `docker run -it nginx`

► `-it` permet d'ouvrir le terminal du container à son lancement.

**Bienvenue** dans votre container !

Pour quitter le terminal d'un **container**, c'est un classique, il faut taper `'exit'`

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

C'est le fameux noyau de **Debian**!

En soit, les fichiers permettant au système d'exploitation de fonctionner correctement.

Maintenant vous êtes dans votre container, vous pouvez installer ce que vous voulez.

Commençons par mettre à jour ***apt** dans le terminal du container*

**APT** est un utilitaire de ligne de commande permettant d'installer, de mettre à jour, de supprimer et de gérer les paquets deb sur Ubuntu, Debian et les distributions Linux associées.

**APT** est donc un gestionnaire de paquet pour Debian, pour être certain que la version d'**APT** installée sur ce container connaît les dernières versions des paquets, nous allons lui demander de vérifier les mises à jour en utilisant : `apt update`, si tout va bien APT devra répondre : **All packages are up to date.**

Nous pouvons rajouter `apt upgrade` pour installer des éventuelles mises à jour de paquets trouvés.

Maintenant que **APT** connaît les dernières versions des paquets, nous allons lui demander de directement installer celui qui nous intéresse, **NGINX**.

Il suffit de taper : `apt install nginx`

Top ! Nous voici désormais avec un container qui possède **NGINX**.

Mais petit problème: si nous fermons notre container, et que nous l'ouvrons à nouveau, **NGINX** a disparu ! ☹

Cela voudrait dire qu'il va falloir entrer à chaque ouverture du container dans le terminal pour l'installer à nouveau ?

Heureusement non, et c'est là qu'intervient notre **Dockerfile**

Rappelez vous, il existe un mot clef **RUN** qui permet d'indiquer à Docker une commande à réaliser à la création du container.

Nous pourrions donc tout simplement rajouter dans notre **Dockerfile** : `RUN apt install nginx` ➤ installer **NGINX**

► **SPOIL**

Maintenant notre build va plus loin ! Mais il y'a encore une **erreur**...

Vous pouvez remarquer que l'avant dernière ligne indique : `Do you want to continue? [Y/n] Abort.`

Hein ? Mais oui je veux continuer ! Cependant quand **Docker build** un container, il n'y a aucun prompt permettant de répondre **YES** à cette question...

De toute façon l'idée d'un **Dockerfile** c'est de pouvoir créer votre projet sans vous demander votre avis en cours d'installation.

Comment faire ?

**APT** propose une option `-y` qui permet de répondre automatiquement **YES** à ce genre de question lors de l'installation d'un package.

Il serait donc une bonne idée de prendre l'habitude de le rajouter lors de l'ajout de ligne utilisant **APT**, sinon votre container ne pourra pas se créer.

Il suffit donc de rajouter un `-y` à la fin de la ligne qui concerne l'installation de **NGINX** par **APT**.

Cela donne : `RUN apt install nginx -y`

Tentons de build à nouveau... Ça fonctionne !

Vous pouvez désormais rentrer dans votre container avec `docker run -it nginx` et vous retrouver à nouveau dans le container, mais cette fois l'installation de **Nginx** est déjà effectuée !

Histoire de pouvoir être à l'aise en accédant au terminal du container, je vais installer **vim** et **curl**. Libre à vous d'installer les packages qui vous semblent utiles :

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

OK maintenant place au **TSL** !

Nous allons créer un dossier, qui permettra de stocker le certificat et la clef pour une **connexion sécurisée**.

Rajoutons : `RUN mkdir -p /etc/nginx/ssl`

Il serait également important de télécharger l'outil principal pour la gestion/création de certificat SSL, **OpenSSL**

Pour cela, comme les autres packages, nous l'installons avec **apt**.

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

Ensuite, il va falloir générer un certificat **SSL** et oui nous utilisons les outils **SSL** afin de créer un certificat **TSL/SSL**.

La commande est un peu longue, commençons par le mot clef **openssl** puis **req**.

```
openssl req
```

La commande **req** crée et traite principalement des demandes de certificats au format PKCS#10. Elle peut en outre créer des certificats auto-signés.

Nous rajouterons ensuite le mot clef **-x509** pour préciser le type du certificat.

```
openssl req -x509
```

Maintenant si nous créons notre certificat, OpenSSL nous demandera un mot de passe, et souvenez vous, si on demande quelque chose à saisir dans le démarrage du container, celui-ci ne va pas pouvoir se **build**. Il faut donc éviter cela à tout prix !

Heureusement, **OpenSSL** a prévu le coup, avec l'option **-nodes**, notre clef privée se retrouvera simplement sans mot de passe.

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

Il faut ensuite indiquer à OpenSSL où l'on souhaite stocker le certificat et la clef de notre SSL en rajoutant les options **-out** et **-keyout**

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

Si nous lançons cette commande nous risquons d'avoir un **prompt** qui requiert certaines informations pour le certificat.

Heureusement, OpenSSL a encore prévu le coup et nous permet de les préremplir en rajoutant une option appelé `-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"
```

Plus qu'à rentrer cela précédé d'un `RUN` dans notre **Dockerfile**.

**Bravo !** Votre clef et votre certificat TSL sont désormais automatiquement créés au démarrage de votre container !

Nous allons ensuite créer un dossier qui nous permettra de stocker les fichiers de config de NGINX.

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

Désormais il faudrait modifier le fichier de *configuration* de **NGINX** comme on le souhaite. Pour cela, nous pourrions faire des **redirections** de certaines phrases dans le fichier de **config**, mais nous allons plutôt utiliser le mot clef **COPY** du **Dockerfile**, qui a l'air fait pour ça.

Nous allons donc prendre le fichier de **config** de **NGINX** de base et nous le modifierons après. Il se trouve dans `/etc/nginx/nginx.conf`



Nous allons le mettre dans un dossier `conf` , lui même dans le dossier `nginx` (*pour respecter la structure indiqué dans le sujet*).

En rajoutant la ligne :

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

Le fichier de configuration de **NGINX** devrait être remplacé par le notre au démarrage du container, faisons un test en rajoutant la phrase “*yes we can*” en commentaire au début de notre fichier `nginx.conf` dans le dossier `conf`.

Si tout va bien, en ***buildant*** le container, et en le **lancant**, vous devriez voir votre fichier `conf` de **NGINX** modifié en y accédant avec vim par exemple.

Avant de lancer **NGINX**, il faudrait indiquer à **NGINX** la configuration qu’on souhaite utiliser dans le cadre d'Inception.

## Le fichier de configuration de NGINX:

Cette page nous aide à paramétrer **NGINX** pour le **ssl**.

Voici la configuration de base que l’on peut trouver :

```
server {
    listen 443;
    ssl on;
    ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
    ssl_certificate /etc/nginx/ssl/bundle.crt;
    ssl_certificate_key /etc/nginx/ssl/private.key;
    ...
}

# Depuis la version 1.12 de NGINX, il faut préciser ssl.
# Inutile depuis la version 1.12
# Gardons TLSv1.2 & TLSv1.3
# Indiquons notre certificat
# et notre clef
```

Le fichier de **config** devient donc :

```
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;
    ...
}
```

Nous avons maintenant la partie **SSL/TSL** qui fonctionne avec **NGINX**.

Je vais préciser le dossier d'accueil en ajoutant `root /var/www/wordpress;`

C'est le dossier où se trouvera WordPress et donc sa première page à afficher.

Je vais également préciser quelle page afficher en premier, dans le cadre de **WordPress** il faut indiquer **index.php**, mais je vais également en ajouter d'autres qui me semblent importantes.

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

Étant donné que la connexion se fera depuis localhost, je l'indique en **server\_name** : `server_name localhost;`

⚠ Il faudra changer `localhost` par l'IP `login.42.fr`

Le fichier de config ressemble maintenant à cela:

```
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;
}
```

Il faut ensuite ajouter des règles sur les locations pour **WordPress**.

Nous devons d'abord demander à NGINX de renvoyer n'importe quelle requête que nous ne connaissons pas sur un **404 error**.

Ce *topic* sur *stackoverflow* explique bien cela. Nous donc ajoutons la règle :

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

Pour tous les fichiers, nous essayons d'ouvrir le fichier renseigné, si c'est un échec nous renverrons **404**.

Il nous reste à installer **PHP** pour pouvoir gérer les requêtes **PHP** de **WordPress**.

Le sujet indique que le PHP doit être installé sur le container de **WordPress** et non de NGINX.

Pas de soucis ! Nous allons donc simplement préciser comment gérer le **PHP** à **NGINX** et lui indiquer où il faut qu'il renvoie notre code **php**.

```
location ~ \.php$ {
    include snippets/fastcgi-php.conf;
    fastcgi_pass wordpress:9000;
}

# Pour toutes les requetes php
# Comment renvoyer les requetes php sur le port 9000
```

Je vais à présent simplement ajouter dans le **Dockerfile** une commande `chmod` afin de s'assurer que nous aurons les droits sur ce dont nous avons besoin.

```
RUN chmod 755 /var/www/html (notre root principal)
```

```
RUN chown -R www-data:www-data /var/www/html (l'utilisateur principal)
```

Maintenant il ne reste plus qu'à lancer NGINX, nous utiliserons le mot clef `CMD` : `CMD [ "nginx", "-g", "daemon off;" ]`

Cela lancera **NGINX** en premier plan pour que le container ne se stop pas.

Vous pourriez voir la page d'accueil de NGINX en allant sur localhost, en utilisant **https** (car vous n'avez pas ouvert le **http**). Il faudrait aussi changer le **root** du fichier de configuration et enlever le `wordpress:9000` pour le php, car vous n'avez pas encore le container du PHP (WordPress)



Nous aurons besoin de **MariaDB** pour installer **WordPress**.

Lors de l'installation de **WordPress**, celui ci requiert un serveur web (NGINX), et une base de données, ici **MariaDB**.

Il serait donc plus judicieux de tout comprendre sur les bases de données et commencer par créer ce container.

## 4. Container MARIADB :

**MariaDB** est un système de gestion de base de données édité sous licence GPL. Il s'agit d'un embranchement communautaire de **MySQL** : la gouvernance du projet est assurée par la fondation **MariaDB**.

En gros, **MariaDB** est quasiment une copie de **MySQL**, mais pourquoi ?

**MySQL** était au début totalement open-source, puis il a été racheté par **Oracle**.

Depuis, plusieurs organismes s'inquiètent de la possibilité qu'a **Oracle** de rendre **payant** son logiciel.

Pour empêcher cela, la fondation **MariaDB** crée une version quasi-identique à MySQL, mais totalement **open-source**.

Le fonctionnement d'une base de données comme MySQL :

La gestion des données est basée sur un modèle de tableaux; toutes les données traitées sur **MySQL** sont stockées dans des tableaux pouvant être reliés les uns aux autres via des clés. En voici un exemple :

oeuvres			
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

Ce *tutoriel* indique parfaitement comment créer une base de données avec **MariaDB** sur Debian 10.

MAJ : Ce *tutoriel* indique parfaitement comment créer une base de données avec **MariaDB** sur Debian 11.

## Tutoriel pour utiliser MySQL/MariaDB:

Pour commencer et pouvoir faire vos tests aussi de votre côté, nous allons créer un **Dockerfile** minimum pour **MariaDB**. Si vous vous souvenez de ce qu'on a fait pour le début de **NGINX**, on a simplement besoin de commencer par d'écrire `FROM debian:buster`.

À partir de là, on peut **build** et lancer le container en rentrant dans son terminal (remontez plus haut pour retrouver comment si besoin)

Maintenant, on se retrouve dans notre container et nous pouvons y effectuer nos commandes tests.

Commençons par le fameux, essentiel, `apt update -y`.

Ajoutons aussi le `apt upgrade -y`.

**APT** contient à présent des dépendances à jour, nous pouvons installer **MariaDB**.

La version 10.3 de **MariaDB** étant la plus commune, nous pouvons l'installer en utiliser **APT**.

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

Pensez au -y, sinon docker aura du mal à **build** votre container car il attendra la confirmation du **y/n** ...

Rendons ensuite visite au fichier de configuration de MySQL, appelé `50-server.cnf` et se trouvant dans `etc/mysql/mariadb.conf.d/50-server.cnf`

Voici à quoi ressemble le fichier de base :

```
# 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/
```

Ok et si on *enlève* les commentaires ?

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

### Le fichier de configuration de MariaDB:

C'est donc l'essentiel du fichier de config de base, sauf que nous, nous n'avons pas besoin de cette partie qui concerne le côté client.

On commence par les crochets `[mysqld]` dans le fichier pour indiquer à quelle catégorie va suivre la configuration suivante (c'est comme cela que fonctionne le fichier de configuration de **MySQL**)

C'est ici que nous pouvons préciser à **MySQL** sur quel port communiquer, indiqué par le sujet, c'est le **3306** avec `port = 3306`

Après, on précise également que toutes les IP du réseau peuvent se connecter, pour cela c'est la ligne `bind_address=*`, qui en gros veut dire: `bind_address=XXX.XXX.XX.XX`

On peut également lui préciser notre dossier qui stockera notre base de données avec `datadir=/var/lib/mysql`

J'ajoute l'utilisateur avec `user=mysql`

Je ne suis pas sûr que ce soit obligatoire car c'est le chemin de base, mais je précise également où MySQL peut trouver la **socket** pour communiquer avec : `socket = /run/mysqld/mysqld.sock`

**Ok good !**

Notre fichier de configuration devrait donc être :

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

Puis reste à demander au [Dockerfile](#) de le copier au bon endroit pour le remplacer.

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

Maintenant que **MySQL** est correctement **installé**, il faut créer une **database** et un **utilisateur** associé.

Pour cela je vais passer par un script que je vais demander au [Dockerfile](#) d'exécuter.

Ce [script](#) doit créer avec MySQL le système de table, grâce à la commande `mysql -e`

Dans ce script nous pouvons d'abord **démarrer** MySQL, sans quoi il sera difficile de le configurer.

```
service mysql start;
```

› La commande [service](#) permet de démarrer MySQL avec la commande associée.

Ensuite il faut créer notre table !

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

› Je demande de créer une table si elle n'existe pas déjà, du nom de la variable d'environnement **SQL\_DATABASE**, indiqué dans mon fichier [.env](#) qui sera envoyé par le [docker-compose.yml](#).

La table est créée! Je vais ensuite créer un utilisateur qui pourra la manipuler.

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

› Je crée l'utilisateur **SQL\_USER** s'il n'existe pas, avec le mot de passe **SQL\_PASSWORD** , toujours à indiquer dans le [.env](#)

Je donne tous les droits à cet utilisateur.

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

› Je donne les droits à l'utilisateur **SQL\_USER** avec le mot de passe **SQL\_PASSWORD** pour la table **SQL\_DATABASE**

Je vais ensuite modifier mon utilisateur **root** avec les droits **localhost** avec cette commande:

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

› Je change les droits root par localhost, avec le mot de passe root **SQL\_ROOT\_PASSWORD**

Plus qu'à rafraichir tout cela pour que MySQL le prenne en compte.

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

› Tout simplement.

Il ne nous reste plus qu'à redémarrer MySQL pour que tout cela soit effectif !

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

› Je commence déjà par éteindre MySQL.

```
exec mysqld_safe
```

› Ici je lance la fameuse commande que MySQL recommande sans arrêt à son démarrage.

Mon script qui configure notre base de données est désormais fonctionnel et devrait lancer le container **mariadb** sans soucis.

Essayons de lancer le container à l'aide d'un : `docker build -t mariadb .`

Puis d'un `docker run -it mariadb`

Si tout va bien, vous devriez avoir un petit [OK] qui indique que MySQL à été lancé sans problème, et si tout va bien, vous devriez également avoir une **erreur!**

Celle-ci devrait indiquer que vous n'avez indiqué aucun mot de passe, ni d'utilisateur lors de la configuration, normal ! Vous avez indiqué des variables d'environnement et celle-ci seront envoyées par docker-compose. Vu que vous démarrez le container seul, il ne trouve pas ces variables.

Rien à faire, si **MySQL** démarre et indique au moins [OK] c'est que vous avez terminé cette partie.

## 5. Container WordPress :

Mettons en place [WordPress](#)!

On a de la chance, **WordPress** est utilisé par plus de **43%** des sites [dans le monde](#), c'est leur page d'accueil qui l'indique. Par conséquent, nous devrions trouver un grand nombre de documentation à son sujet.

Commençons comme d'habitude par un [Dockerfile](#) sous **debian:buster**.

Ensuite viens le traditionnel `apt-get update` et `apt-get upgrade`

Pour prévoir la suite (nous aurons besoin d'installer WordPress avec son lien de téléchargement) nous devons installer [wget](#)

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

Le sujet nous l'indique, nous devons également installer **PHP** avec **WordPress**.

Rappelez vous, il communiquera sur le port **9000** avec **NGINX**.

Pour cela, j'utilise également **APT** pour installer **php7.3** et ses dépendances comme *php-fpm* et *php-mysql*.

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

Il est enfin temps d'installer le fameux **WordPress** dans notre container !  
Nous utilisons donc wget en indiquant le lien d'installation. J'ai pris la version **FR 6.0**.  
À vous de choisir, vous trouverez les différentes versions ici.

wget possède plusieurs options, dont une qui va nous permettre d'indiquer dans quel dossier on veut télécharger le fichier en utilisant -P.  
Dans quel dossier ? le **/var/www** évidemment ! C'est ici que nous avons indiqué notre dossier principal à afficher dans le container **NGINX**.

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

› Très bien, top, mais maintenant il faudrait le dé-tar, enfin le décompresser quoi !

Allez je vous aide, allons dans le dossier **/var/www** et utilisons `tar -xvf` suivi du fichier pour le décompresser et en obtenir le fameux dossier `wordpress` !

Ensuite on peut supprimer le `.tar` qui ne sert plus à rien.

```
RUN cd /var/www && tar -xzf wordpress-6.0-fr_FR.tar.gz && rm wordpress-6.0-fr_FR.tar.gz
```

› Nous voilà avec `wordpress` sous forme de dossier.

Maintenant veillons à bien donner les droits à root d'écrire dans ce dossier.

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

Maintenant que **WordPress** est installé, mais pas configuré, occupons nous de **PHP**.  
**PHP** sert à **WordPress**, la plupart des fichiers de WordPress sont des `.php`.

PHP est un langage de programmation libre, principalement utilisé pour produire des pages Web dynamiques via un serveur HTTP, mais pouvant également fonctionner comme n'importe quel langage interprété de façon locale.

Une fois qu'il est installé avec **APT**, nous pouvons modifier son fichier de configuration.

Ici, j'ai pris le fichier de configuration de base et j'ai modifié seulement 2 petites choses.

J'ai ajouté une ligne `clear_env = no`, je pense que cette ligne est assez claire pour ne pas avoir à l'expliquer, c'est pour l'environnement.  
Mais j'ai également modifié la ligne `listen`. J'ai indiqué qu'il devait écouter au port de WordPress, le **9000**.  
Grosso modo ca donne : `listen=wordpress:9000`

**PHP** est prêt, et oui, enfin après il faut évidemment demander au **Dockerfile** de copier le fichier de configuration au bon endroit dans le container, mais ça inutile de vous l'expliquer vous devriez déjà être des experts dans le domaine.

Place à la configuration de notre ami **WordPress**.

Quoi il faut le configurer ? ☹

**WordPress** a besoin d'une base de données pour fonctionner, du moins connaitre son mot de passe, son nom et son host.

Tout cela se configure dans le fichier **wp-config.php** (*un truc comme ça*)

Si vous ne le faites pas, vous arriverez directement sur la page de configuration du site, c'est pas plus mal, mais le sujet requiert une configuration automatique.

Mais j'ai une bonne nouvelle pour vous ! Un développeur a simplement créé une **CLI** qui permet de configurer presque automatiquement ce genre d'informations pour vous.  
Jetez un oeil à *ce super tuto*.

Une CLI c'est une interface textuelle qui traite les commandes vers un programme **informatique**. Il existe différentes interfaces de ligne de commande, telles que DOL et Shell Bash.

Je fonce sur cette idée qui m'a l'air bien sympa, il faut déjà réutiliser **wget** pour installer le CLI.

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

Une autre possibilité aurait été de modifier directement le fichier `wp-config.php` à coup de `sed` pour rentrer les infos directement dans le fichier.

Il faut ensuite lui donner les bons droits ainsi que le placer dans les binaires.

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


Ok top ! On a installé le **CLI de WordPress**

Maintenant il faudrait l'utiliser, pour cela on va faire comme avec **MariaDB**, créer un petit script bash qu'on copiera et qui effectuera les commandes à notre place au lancement du container.

Je crée donc un `auto_config.sh` que je place dans le dossier `conf` de **WordPress**.

Dans celui ci, je vais, par précaution mettre un `sleep 10` afin d'être certain que la base de données **MariaDB** a bien eu le temps de se lancer correctement.  
Ensuite, et uniquement si notre fichier `wp-config.php` n'existe pas (nous ne voudrions pas *reconfigurer WordPress* à chaque lancement non ?) , nous utilisons la commande `wp config create` du **CLI** pour indiquer les informations dont **WordPress** a besoin.

Nous pourrions remplir ces informations à la main directement en accédant à localhost en lançant notre container.  
Cela ressemblerait à cela :



Below you should enter your database connection details. If you're not sure about these, contact your host.

Database Name	<input type="text" value="wordpress"/>	The name of the database you want to use with WordPress.
Username	<input type="text" value="username"/>	Your database username.
Password	<input type="text" value="password"/>	Your database password.
Database Host	<input type="text" value="localhost"/>	You should be able to get this info from your web host, if localhost doesn't work.
Table Prefix	<input type="text" value="myprefix_"/>	If you want to run multiple WordPress installations in a single database, change this.

Submit

➤ Ceci est la première page que présente WordPress à son lancement.


Sauf que nous, nous voudrions que cela soit déjà automatiquement configuré.  
Nous pouvons donc indiquer ces infos avec l'aide de `wp config create` du [CLI](#).

Cela donnerait quelque chose comme ça :

```
wp config create --allow-root \  
--dbname=$SQL_DATABASE \  
--dbuser=$SQL_USER \  
--dbpass=$SQL_PASSWORD \  
--dbhost=mariadb:3306 --path='/var/www/wordpress'
```

➤ À indiquer directement après notre sleep dans notre fichier `auto_config.sh`

Maintenant si nous lançons le container **WordPress**, ça ne devrait plus être la page que je vous ai montré juste au dessus mais la deuxième, celle-ci est bien plus sympa.



## 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.

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

Install WordPress

C'est la page qui demande de choisir un **titre** pour votre site, ainsi qu'un **nom d'utilisateur** et un **mot de passe**, mais la impossible d'avoir faux car c'est à vous de le choisir 😊  
Pour moi, il serait normal de laisser cette page de configuration à remplir, mais le sujet demande qu'elle soit également automatiquement configurée, car celui-ci demande qu'il y ait 2 utilisateurs **WordPress**, et c'est sur cette page qu'on peut configurer le premier.

Je vous laisse regarder la doc du **CLI** mais celui ci propose de configurer cette deuxième page automatiquement avec la commande `wp core install` et même d'ajouter un autre utilisateur avec la commande `wp user create`. Franchement je vous laisse la partie la plus simple.

Plus qu'à copier le fichier `auto_config.sh` dans votre container et l'exécuter avec le mot clef **ENTRYPOINT**.

Ensuite pour éviter une erreur qui concerne le **PHP**, j'ai rajouté une condition qui crée le dossier `/run/php` s'il n'existe pas.

Enfin, je lance **php-fpm** avec la commande :

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

Vous avez terminé la configuration de votre container **WordPress**!

## 6. Relier les containers avec Compose:

### Place au fameux `docker-compose.yml` !

Bon, d'abord renseignez vous sur comment s'écrit un fichier **docker-compose**.  
Celui-ci commence toujours par la version de **docker-compose**, la dernière, la 3.



Attention à l'indentation du **docker-compose.yml** ! Comme pour Python, c'est l'indentation qui compte.

Ensuite on indique la ligne `services:`

Et là, avec une indentation de **1 tab**, on peut énumérer nos différents services.

Je commence par **MariaDB** :

```
mariadb:
  container_name: mariadb      # Le nom du container, oui vraiment.
  networks:
    - inception                # à quel network il appartient
  build:
    context: requirements/mariadb # ou 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:
    - "3306"                   # le port à exposer
```

Place au service **NGINX**. Ici c'est le même principe, sauf deux choses qui changent et les **noms/paths** évidemment.

```
nginx:
  container_name: nginx
  volumes:
    - wordpress:/var/www/wordpress
  networks:
    - inception
  depends_on:
    - wordpress                # Nouvelle ligne, indiquant de ne pas démarrer NGINX tant que WordPress n'a pas démarré.
  build:
    context: requirements/nginx
  dockerfile: Dockerfile
  env_file: .env
  ports:
    - "443:443"                # on indique le port qui sera exposé a la machine locale
  restart: on-failure           # Ici nous changeons, le container redémarrera uniquement en cas de crash.
```

C'est bon, vous l'avez ? Faisons de même pour **WordPress**.

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

› Et voilà ! (J'aurais aimé qu'on fasse le taff de lire la doc à ma place moi)

## 7. Les volumes à configurer:

### Parlons rapidement de la ligne Volumes:

Nous assignons un volume à **MariaDB**, comme nous allons le faire avec **WordPress**.

C'est un *prérequis* du sujet. Nous devons permettre la persistance des données, pour cela nous allons stocker certains dossiers directement sur notre ordinateur en local.

Et oui, imaginez que votre container **WordPress** **crash** ou **s'éteint**, et que par pur hasard, celui-ci a perdu tous vos fichiers, ce serait bien embêtant, c'est pour cela que nous préférons les stocker directement en local, et **docker-compose** nous le permet.

Dans le cas d'Inception, nous allons stocker les dossiers de **MySQL** se trouvant dans `/var/lib/mysql` et WordPress dans `var/www/wordpress`. Ceux sont les **paths** que j'indique après la ligne *volume*. Encore une fois c'est le sujet qui nous le demande.

Ici à la ligne volume nous indiquons **MariaDB** (c'est le nom du volume, nous aurions pu l'appeler *vo\_maria* ou autre, évidemment il fera intervenir une ligne concernant les volumes dans le **docker-compose.yml** qui permettra d'indiquer où le stocker en local) suivi de `:` avec l'endroit que nous souhaitons copier du container.

Il ne manque plus qu'à préciser les volumes que nous avons indiqués :

```
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' #Ou 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, et ils peuvent être modifiés par d'autres processus en dehors de docker.
      device: '/Users/login/data/mariadb' #Ou stocker le dossier sur votre ordinateur en local
```

› Cette doc de docker explique bien comment gérer les volumes dans un docker-compose.

Vos volumes seront disponibles dans le dossier `/home/login/data` de la machine hôte utilisant Docker.

› Pensez donc à modifier le path de la ligne device: en conséquence.

Il ne nous reste plus qu'à créer la partie network, attention, c'est très simple.

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

› Oui, c'est tout.

Ici `bridge` indique a Docker d'installer automatiquement des règles qui permettront aux 3 containers de communiquer en bridge.

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

Pour l'arrêter : `docker-compose -f <path_docker_compose> stop`

Pour supprimer le build : `docker-compose -f <path_docker_compose> down -v`

Si vous rencontrez des problèmes avec docker vous pouvez utiliser la commande :

`docker system prune -af`

Attention, ça supprime tous les container, images, etc.

## 8. Finaliser le projet :

### Petites rectifications:

Depuis que j'ai testé tout cela, j'ai remarqué quelques petits bugs, j'ai donc ajouté quelques petites précisions au fichier de configuration de **NGINX**.

J'ai ajouté en première ligne du fichier `nginx.conf` le mot clef `events {}` car **NGINX** le demandait avec une erreur du type `missing events{}`.

Dans ce même fichier, j'ai également ajouté la ligne `include /etc/nginx/mime.types;` juste en dessous du mot clef **http**.

Pourquoi ? Le CSS ne chargeait pas, et après une longue enquête j'ai remarqué que ceux-ci était indiqué avec un Content-Type **html** ?!

Après de longues visites sur le site de [StackOverflow](#) j'ai finalement ajouté cette ligne qui précise les **content-types**, et tout est rentré dans l'ordre !

À priori, il ne vous reste plus qu'à faire un Makefile et **vous êtes bon**

Ah oui, j'ai choisi d'ajouter un mini script bash qui configure automatiquement les bon path dans docker-compose.yml, toujours sympa quand un ami veut tester le projet sur son ordinateur, il n'a pas à chercher chaque path dans tous les fichiers pour les changer.

Si vous tentez de vous connecter à localhost ou 127.0.0.1 depuis votre navigateur (*après avoir lancé votre container évidemment*) vous ne verrez sûrement rien apparaître. Normal nous n'avons ouvert que le port **443** comme port d'écoute.

Le port **443** correspondant au port **SSL** nous devons donc nous y connecter en utilisant `https://` et non `http://`, car ce dernier nous emmènerait sur le port standard **80**, que nous aurions sûrement configuré sur un site classique (celui-ci aurait redirigé vers le 443). Mais ici le sujet **interdit** l'ouverture d'un autre port que le **443**.

Autre souci, selon votre navigateur, celui-ci devrait afficher un message **d'alerte** indiquant que ce site **tente sûrement de vous voler des informations sensibles**.

Dans le langage d'un **dev**, cela veut dire que le certificat **SSL** n'a pas été signé par **Trusted Authority**. Nous ne pouvons rien y faire quand il s'agit d'un projet en local et encore moins avec un certificat généré par **OpenSSL**.

Cependant, nous pouvons tout de même entrer sur le site. Pour Safari, il existe un bouton qui demande tout de même l'accès au site. Sur Chrome, il existe un petit secret, il suffit de taper le mot `thisisunsafe` quand vous êtes sur la page d'alerte.

Et voilà, vous devriez arriver sur votre site.

Pour des questions de lisibilité, il faudra configurer votre nom de domaine afin qu'il pointe vers votre adresse IP locale.  
Ce nom de domaine sera login.42.fr. Une fois de plus, vous utiliserez votre login.

› Ici il s'agit uniquement d'une configuration à effectuer sur votre machine en local.

En gros vous pouvez déjà accéder au site depuis **localhost**, qui (*ce mot*) redirige en fait sur l'IP **127.0.0.1**, sans vous le dire, mais tout cela est en fait écrit dans un fichier de votre ordinateur.

Ce fichier est très sensible et il vous faudra l'éditer avec un `sudo`. En effet, c'est un fichier très visé par les hackers, il permettrait de vous rediriger facilement sur un faux google quand vous tapez google.fr, par exemple.

En éditant ce fichier qui se trouve généralement ici : `/etc/hosts` vous pourrez facilement demander la redirection de **127.0.0.1** sur l'IP que vous voudrez, comme **login.42.fr**

N'oubliez pas de modifier cet IP dans le fichier de conf de NGINX dans la case `server_name`, le mieux serait de le faire aussi dans la génération du certificat **SSL**, mais bon, celui-ci n'est pas authentifié...

À priori vous respectez maintenant toutes les règles du sujet.

Bon courage pour la correction !