

Docker

Phase 1 : Docker - Veille Technique

Qu'est-ce qu'une machine virtuelle ?

Une **machine virtuelle** (VM) est un logiciel qui émule un ordinateur physique. Elle permet d'exécuter un système d'exploitation (OS) distinct, avec ses propres applications, sur un même serveur physique.

En pratique, une machine virtuelle se comporte comme un ordinateur indépendant :

- Elle possède son propre OS et ses applications.
- Elle partage les ressources matérielles du serveur (CPU, mémoire, disque, etc.) mais reste isolée des autres VMs et du système hôte.

Les machines virtuelles sont couramment utilisées pour **exécuter plusieurs OS sur un même serveur** sans conflit entre eux.

Quelle est la différence entre une machine virtuelle et un conteneur ?

Les machines virtuelles et les conteneurs servent tous deux à isoler des applications, mais leur fonctionnement est différent :

- **Machine virtuelle** : chaque VM inclut un OS complet, ses bibliothèques et ses dépendances. Cela consomme plus de ressources car chaque VM doit charger tout un système d'exploitation.
- **Conteneur** : un conteneur partage le même noyau de l'OS que le système hôte, ce qui le rend plus léger. Il ne contient que les applications et les dépendances nécessaires pour fonctionner. Les conteneurs démarrent et s'arrêtent plus rapidement que les VMs.

Résumé : Les conteneurs sont plus légers et rapides que les VMs, car ils partagent le noyau de l'OS du système hôte.

Qu'est-ce que Docker ? Qu'est-ce que la conteneurisation ?

Docker est une plateforme permettant de créer, déployer et gérer des applications dans des conteneurs.

- **Conteneurisation** : c'est le processus d'encapsuler une application et ses dépendances dans un conteneur.
- Un conteneur est une unité légère et isolée qui exécute une application indépendamment du système hôte.

Docker rend la conteneurisation accessible et pratique en simplifiant la gestion des conteneurs.

Quels sont les avantages de la conteneurisation ?

Voici quelques avantages principaux :

1. **Légereté** : les conteneurs partagent le noyau du système hôte, ce qui réduit leur taille et leur consommation de ressources.
2. **Portabilité** : les conteneurs fonctionnent de la même manière partout (localement, sur des serveurs ou dans le cloud), ce qui facilite le déploiement.
3. **Isolation** : chaque conteneur est isolé, ce qui garantit que les applications ne se perturbent pas entre elles.
4. **Rapidité** : les conteneurs démarrent et s'arrêtent rapidement, ce qui est idéal pour les environnements de développement et de test.

Qu'est-ce qu'une image Docker, et quelle est la différence avec un conteneur ?

- Une **image Docker** est un modèle statique contenant tout le nécessaire pour exécuter une application, y compris l'application elle-même, ses dépendances, ses bibliothèques et les configurations nécessaires.
- Un **conteneur Docker** est une instance en cours d'exécution d'une image. C'est l'image mise en action, exécutée dans un environnement isolé.

Exemple visuel

Penser à l'image comme à un moule de gâteau. On peut créer plusieurs gâteaux à partir du même moule. Chaque gâteau (conteneur) est indépendant des autres et peut être décoré ou modifié sans changer le moule d'origine.

Résumé : Une image est le modèle, et un conteneur est l'instance active de ce modèle.

Qu'est-ce qu'un Dockerfile ?

Un **Dockerfile** est un fichier texte contenant les instructions nécessaires pour créer une image Docker.

Dans un Dockerfile, on spécifie :

- Les **instructions** pour installer des logiciels, copier des fichiers, configurer l'application, etc.
- Le Dockerfile est ensuite utilisé pour construire une **image** Docker, qui pourra être exécutée sous forme de conteneur.

Phase 2 : Création et Gestion de Conteneurs

1. Lancer un conteneur Ubuntu en mode interactif

Pour lancer un conteneur **Ubuntu** en mode interactif, exécute la commande suivante :

```
docker run -it ubuntu
```

- `-it` permet d'ouvrir le conteneur en mode interactif, avec un terminal.
- `ubuntu` spécifie l'image Ubuntu.

Une fois le conteneur lancé, tu seras dans un shell **Linux** (comme si tu étais directement connecté à un terminal Ubuntu). Tu pourras alors tester des commandes Linux de base :

- **Lister les répertoires** : `ls`
- **Créer un fichier** : `touch monfichier.txt`
- **Créer un répertoire** : `mkdir mondossier`
- **Vérifier le contenu** : `ls -la`

Pour **quitter le conteneur** interactif, tape simplement `exit`.

2. Utiliser `docker exec` pour exécuter une commande dans un conteneur en cours d'exécution

Si tu as un conteneur Ubuntu en cours d'exécution, tu peux exécuter une commande sans entrer dans le conteneur de façon interactive, grâce à `docker exec`.

1. **Trouve l'ID du conteneur** (exécute `docker ps` pour afficher la liste des conteneurs en cours d'exécution).
2. Utilise `docker exec` pour exécuter une commande dans le conteneur :

```
docker exec -it <id_du_conteneur> ls /
```

3. Utiliser `docker logs` pour récupérer les logs d'un conteneur

Pour récupérer les **logs d'un conteneur**, utilise la commande suivante :

```
docker logs <id_du_conteneur>
```

- Par exemple, si tu as un conteneur `nginx` en cours d'exécution, tu pourras voir les logs d'accès.
-

4. Utiliser `docker rm` pour supprimer un conteneur arrêté

Une fois qu'un conteneur est arrêté, tu peux le supprimer avec `docker rm`.

1. Arrête le conteneur (si nécessaire) :

```
docker stop <id_du_conteneur>
```

2. Supprime le conteneur :

```
docker rm <id_du_conteneur>
```

5. Monter un volume dans un conteneur et observer la persistance des données

Pour monter un **volume** et observer la persistance des données :

1. Crée et lance un conteneur avec un volume monté :

```
docker run -it -v mon_volume:/data ubuntu
```

```
- '-v mon_volume:/data' monte un volume nommé 'mon_volume' dans le répertoire '/data' du conteneur.
```

2. Dans le conteneur, crée un fichier dans le volume :

```
touch /data/monfichier.txt
```

3. Quitte et supprime le conteneur :

```
exit docker rm <id_du_conteneur>
```

4. Relance un nouveau conteneur avec le même volume et vérifie si le fichier est toujours présent :

```
docker run -it -v mon_volume:/data ubuntu ls /data
```

Si le fichier `monfichier.txt` est là, cela prouve que les données dans le volume sont persistantes, même après la suppression du conteneur.

6. Créer un réseau Docker et connecter deux conteneurs

Pour connecter deux conteneurs dans un réseau Docker :

1. Crée un réseau Docker nommé `mon_reseau` :

```
docker network create mon_reseau
```

2. Lance un conteneur serveur (par exemple, basé sur Ubuntu) et connecte-le au réseau :

```
docker run -it --name serveur --network mon_reseau ubuntu
```

3. Lance un autre conteneur client sur le même réseau :

```
docker run -it --name client --network mon_reseau ubuntu
```

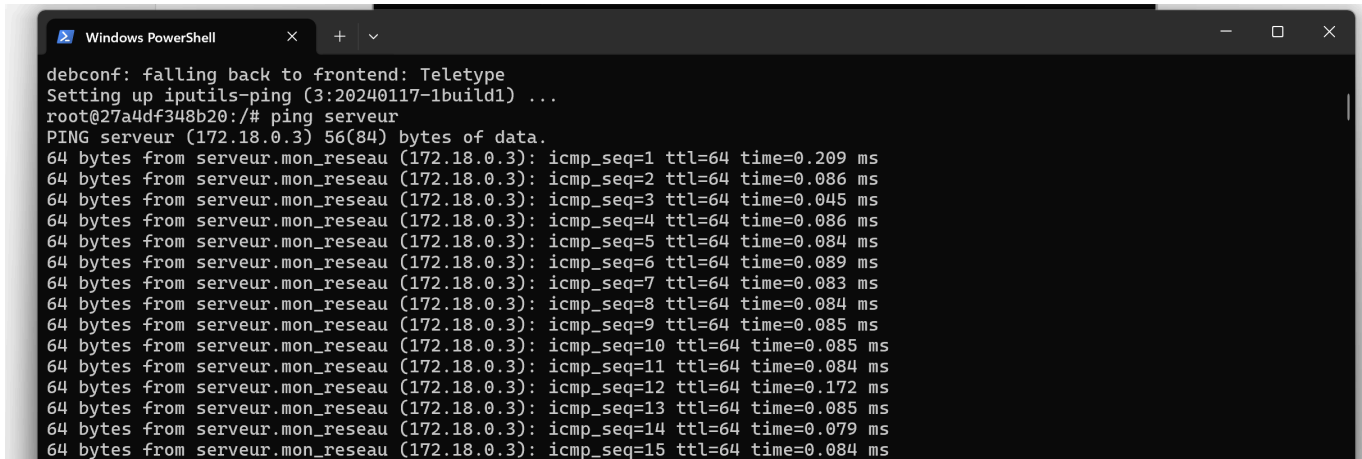
4. Depuis le conteneur **client**, teste la communication avec le serveur :

```
ping serveur
```

Cette commande enverra des paquets vers le conteneur serveur, ce qui prouve que les deux conteneurs peuvent communiquer sur le réseau Docker.

Info : j'ai du installer le paquet `iputils-ping` dans le container client pour pouvoir effectuer un ping :

```
apt update && apt install -y iputils-ping
```



```
Windows PowerShell
debconf: falling back to frontend: Teletype
Setting up iputils-ping (3:20240117-1build1) ...
root@27a4df348b20:/# ping serveur
PING serveur (172.18.0.3) 56(84) bytes of data.
64 bytes from serveur.mon_reseau (172.18.0.3): icmp_seq=1 ttl=64 time=0.209 ms
64 bytes from serveur.mon_reseau (172.18.0.3): icmp_seq=2 ttl=64 time=0.086 ms
64 bytes from serveur.mon_reseau (172.18.0.3): icmp_seq=3 ttl=64 time=0.045 ms
64 bytes from serveur.mon_reseau (172.18.0.3): icmp_seq=4 ttl=64 time=0.086 ms
64 bytes from serveur.mon_reseau (172.18.0.3): icmp_seq=5 ttl=64 time=0.084 ms
64 bytes from serveur.mon_reseau (172.18.0.3): icmp_seq=6 ttl=64 time=0.089 ms
64 bytes from serveur.mon_reseau (172.18.0.3): icmp_seq=7 ttl=64 time=0.083 ms
64 bytes from serveur.mon_reseau (172.18.0.3): icmp_seq=8 ttl=64 time=0.084 ms
64 bytes from serveur.mon_reseau (172.18.0.3): icmp_seq=9 ttl=64 time=0.085 ms
64 bytes from serveur.mon_reseau (172.18.0.3): icmp_seq=10 ttl=64 time=0.085 ms
64 bytes from serveur.mon_reseau (172.18.0.3): icmp_seq=11 ttl=64 time=0.084 ms
64 bytes from serveur.mon_reseau (172.18.0.3): icmp_seq=12 ttl=64 time=0.172 ms
64 bytes from serveur.mon_reseau (172.18.0.3): icmp_seq=13 ttl=64 time=0.085 ms
64 bytes from serveur.mon_reseau (172.18.0.3): icmp_seq=14 ttl=64 time=0.079 ms
64 bytes from serveur.mon_reseau (172.18.0.3): icmp_seq=15 ttl=64 time=0.084 ms
```

Phase 3 : Création d'Images et Déploiement d'Application

1. Créer un Dockerfile pour une image de base Python avec des bibliothèques spécifiques

1. Dans le répertoire de votre projet, crée un fichier nommé `Dockerfile` .
2. Écris le `Dockerfile` pour créer une image basée sur Python avec la bibliothèque `pandas` .
Voici un exemple de contenu pour le `Dockerfile` :

```
# Utiliser Python 3.13-slim (ou la version la plus proche disponible)

FROM python:3.13-slim

# Définir le répertoire de travail dans le conteneur

WORKDIR /app

# Copier les fichiers du projet dans le conteneur

COPY . /app
```

```
# Installer les dépendances à partir de requirements.txt
```

```
RUN pip install --no-cache-dir -r requirements.txt
```

```
# Exposer le port 5000 pour l'application Flask
```

```
EXPOSE 5000
```

```
# Commande pour démarrer l'application Flask
```

```
CMD ["python3", "app.py"]  
```\
```

Ce Dockerfile utilise **l'image Python 3.13** comme base et installe **pandas**. Il copie également les fichiers du projet dans le répertoire `/app` du conteneur.

**Attention de bien spécifier le host et le port dans app.py pour les applications Flask :**

```
```python  
if __name__ == '__main__':  
  
    app.run(host='0.0.0.0', port=5000)
```

2. Construire l'image avec la commande `docker build`

1. Pour construire l'image, exécute la commande suivante depuis le répertoire où se trouve le Dockerfile :

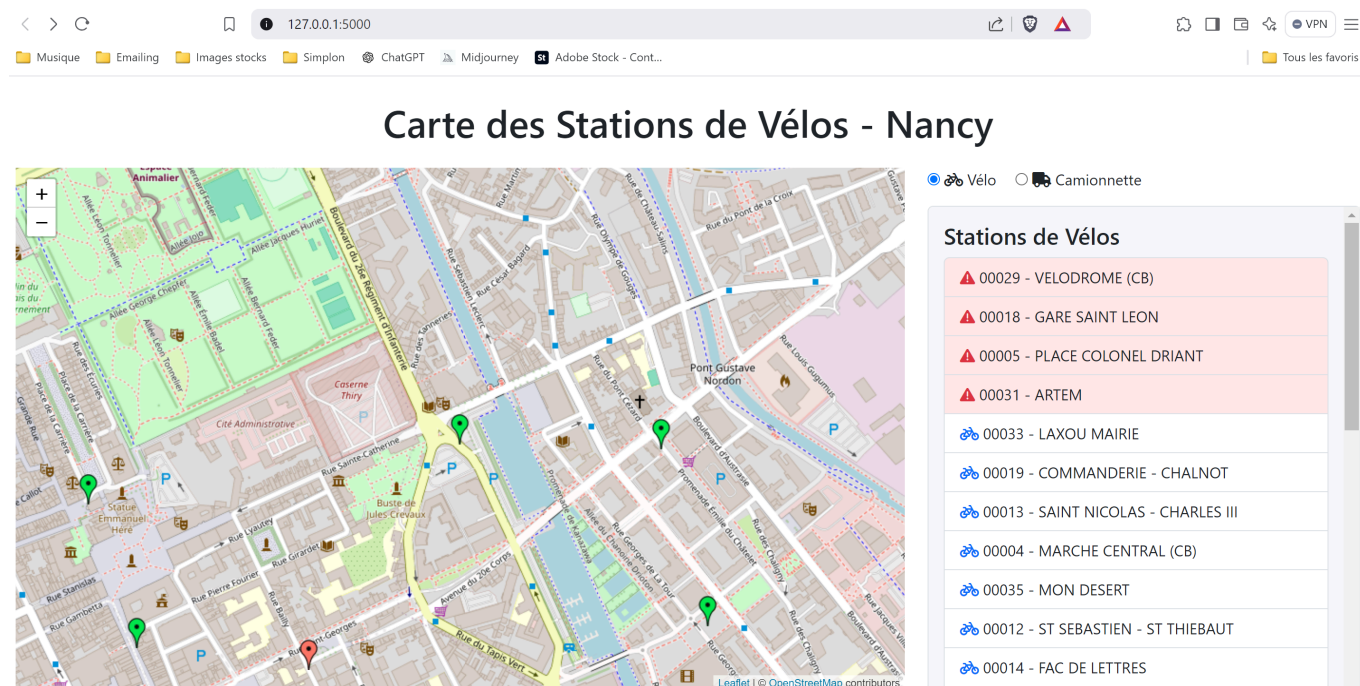
```
docker build -t mon_image_python .
```

Docker télécharge l'image de base, installe pandas et crée une nouvelle image `mon_image_python`.

3. Intégrer une application Flask dans le conteneur

Pour cette partie, j'ai choisi d'intégrer l'application Flask Velostan qui permet de voir quelles les stations sous-chargées et surchargées.

4. Test de l'application



Phase 4 : Projet de Déploiement en Conditions Réelles

Pour la Phase 4, j'ai choisi le projet Nutriscore. L'architecture Docker du projet est composée de deux services principaux :

1. Service WebApp (Frontend/API)

```
webapp:
  build:
    context: .
    dockerfile: Dockerfile.webapp
  ports:
    - "5000:5000"
  environment:
    - FLASK_APP=run.py
    - MODEL_SERVICE_URL=http://modelservice:5001
  depends_on:
    - modelservice
  networks:
```


Ce service :

- Utilise Python 3.13.0-slim comme image de base
- Expose le port 5000
- Se connecte au service de modèle via l'URL `http://modelservice:5001`
- Dépend du service `modelservice`

2. Service ModelService (Service de prédiction)

```
modelservice:
  build:
    context: .
    dockerfile: Dockerfile.model
  ports:
    - "5001:5001"
  networks:
```

Ce service :

- Utilise également Python 3.13.0-slim
- Expose le port 5001
- Contient le modèle de prédiction

Les Dockerfiles sont structurés comme suit :

- **Dockerfile.webapp :**
 - Crée l'environnement pour l'application web Flask
 - Copie les fichiers de l'application et les dépendances
 - Configure le point d'entrée sur `run.py`
- **Dockerfile.model :**
 - Configure l'environnement pour le service de prédiction
 - Copie les fichiers nécessaires (modèle, scaler, données)
 - Lance le service via `model_service.py`

Points importants à noter :

1. Réseau Docker :

```
networks:
  nutriscore-net:
```

```
driver: bridge
```

Un réseau bridge est créé pour permettre la communication entre les services.

2. Bonnes pratiques :

- Utilisation de **.dockerignore** pour exclure les fichiers inutiles
- Images légères avec Python slim
- Séparation claire des responsabilités entre services
- Variables d'environnement pour la configuration

Pour installer et démarrer l'application :

```
git clone https://github.com/seb54/docker_nutriscore.git  
cd docker_nutriscore
```

Pour démarrer l'application :

```
docker-compose up --build
```

Pour l'arrêter :

```
docker-compose down
```