

RESOLUTION DU TP DE 3DCT

1 - Lancement des conteneurs

a) Utiliser les commandes de l'interface de ligne de commande Docker pour démarrer, arrêter et gérer les conteneurs.

- **docker build -t nom_image** : cette commande permet de construire une image
- **docker rm nom_image** : Pour supprimer un conteneur arrêté
- **docker images** : permet de lister les différentes images existantes.
- **docker pull nom_image** : permet de télécharger une image depuis le docker hub.
- **docker-compose up** : permet de lancer un conteneur
- **docker-compose down** : permet d'arrêter un conteneur
- **docker ps** : permet de lister tous les conteneurs en marche
- **docker ls** : permet de lister tous les conteneurs
- **docker cp fichier :source destination** : permet de copier un fichier d'une source vers une destination
- **docker exec -it nom_image /bin/bash** : permet d'ouvrir un terminal interactif dans un conteneur
- **docker logs nom_image** : permet de voir les logs d'un conteneur
- **docker system prune -a** : permet de stopper tous les conteneurs
- **docker-compose prune** : permet de supprimer tous les conteneurs stoppés

b) Configurer et utiliser la liaison de ports pour permettre la communication vers/depuis les conteneurs.

- **docker run -d nom_image -p 8000 :80** : permet de démarrer un conteneur en mode détaché tout en spécifiant les ports sur lesquels notre conteneur doit écouter

2 - Compréhension de la conteneurisation en entreprise

a) Comparer la conteneurisation avec la virtualisation traditionnelle (VM).

La conteneurisation se distingue de la virtualisation traditionnelle des machines (VM) par sa légèreté et son efficacité. Alors que les VM nécessitent l'allocation de ressources matérielles dédiées, y compris le système d'exploitation complet pour chaque instance,

les conteneurs partagent le même noyau du système d'exploitation de l'hôte. Cela permet une utilisation plus efficace des ressources matérielles et une mise en place plus rapide des environnements.

De plus, les conteneurs offrent une isolation des processus plus légère que les VM, ce qui signifie qu'ils peuvent être démarrés et arrêtés plus rapidement, ce qui permet une montée en charge plus agile et des temps de déploiement plus courts.

b) Discuter de l'impact de Docker sur les cycles de développement, de test et de déploiement dans un contexte d'entreprise.

La conteneurisation est devenue une pratique courante dans les environnements d'entreprise en raison de ses nombreux avantages par rapport aux méthodes traditionnelles de déploiement logiciel, notamment la virtualisation des machines (VM). Dans un contexte d'entreprise, Docker et la conteneurisation en général ont un impact significatif sur les cycles de développement, de test et de déploiement.

- ❖ **Développement** : Les conteneurs fournissent un environnement de développement cohérent et portable. Les développeurs peuvent créer des images conteneurisées avec toutes les dépendances nécessaires à l'exécution de leur application, garantissant ainsi que l'application fonctionnera de la même manière sur n'importe quelle plateforme. Cela réduit les problèmes liés à la configuration de l'environnement de développement et facilite la collaboration entre les membres de l'équipe.
- ❖ **Test** : Les conteneurs permettent la mise en place rapide d'environnements de test isolés. Les équipes de test peuvent déployer plusieurs instances de l'application dans des conteneurs distincts pour effectuer des tests d'intégration, des tests de charge et des tests de régression de manière efficace. De plus, la portabilité des conteneurs facilite la reproduction des problèmes signalés par les utilisateurs dans des environnements de test, accélérant ainsi le processus de résolution des problèmes.
- ❖ **Déploiement** : Docker simplifie le déploiement d'applications dans des environnements de production en automatisant le processus de création, de distribution et de déploiement des conteneurs. Les orchestrateurs de conteneurs tels que Kubernetes fournissent des fonctionnalités avancées telles que l'équilibrage de charge, la mise à l'échelle automatique et la gestion des mises à jour, ce qui rend le déploiement et la gestion des applications à grande échelle plus faciles et plus fiables.

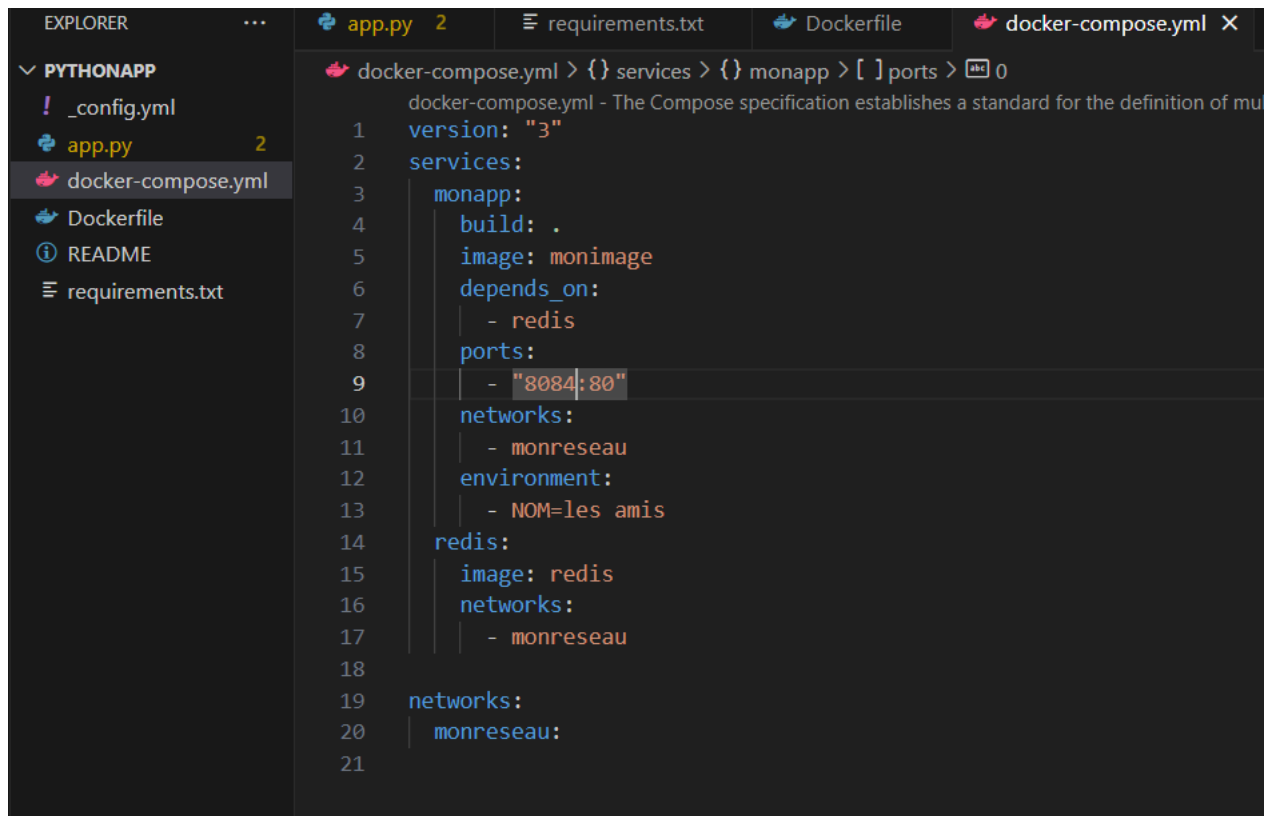
3 - Création d'images Docker

❖ Détails de l'Application :

J'ai déployé une application Python Flask en partant d'une image **python:2.7-slim**. J'ai spécifiquement choisi cette image car elle garantit la possibilité d'exécuter des commandes Linux, offrant ainsi une flexibilité et une légèreté optimales pour notre environnement de développement.

❖ Configuration Multiservice :

En plus de l'application Flask, mon déploiement inclut également un service de base de données Redis pour gérer les données de l'application de manière efficace et performante.



```
1 version: "3"
2 services:
3   monapp:
4     build: .
5     image: monimage
6     depends_on:
7       - redis
8     ports:
9       - "8084:80"
10    networks:
11      - monreseau
12    environment:
13      - NOM=les amis
14  redis:
15    image: redis
16    networks:
17      - monreseau
18
19  networks:
20    monreseau:
21
```

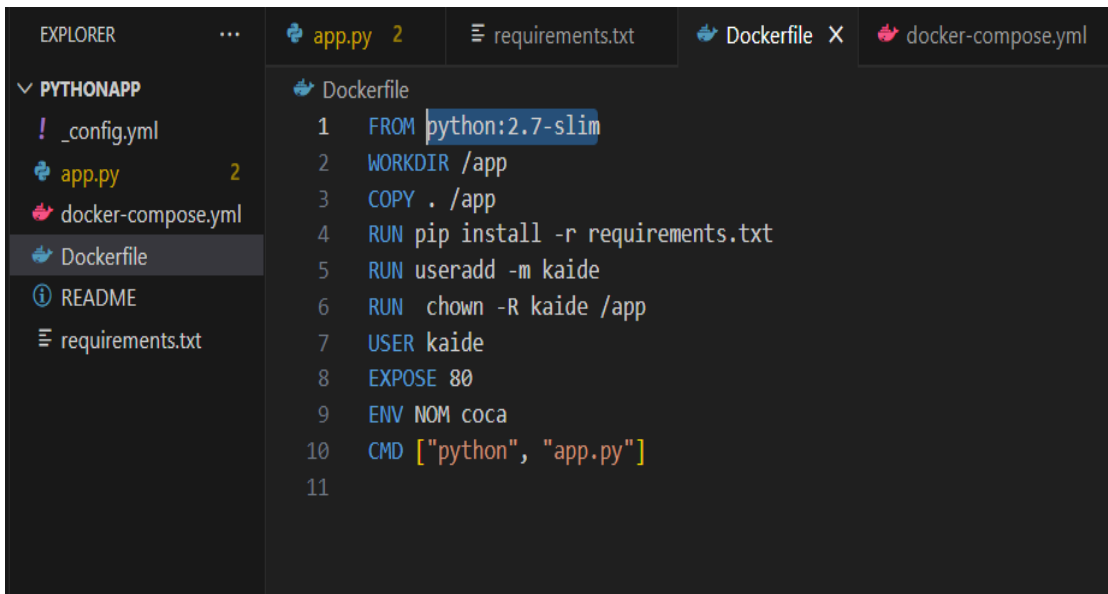
❖ Étapes Clés :

1 - Création d'un Nouvel Utilisateur

J'ai créé un nouvel utilisateur nommé kaide pour garantir que les commandes soient exécutées par cet utilisateur lors du démarrage du conteneur, améliorant ainsi la sécurité en évitant l'exécution en tant que root.

2 - Configuration Docker

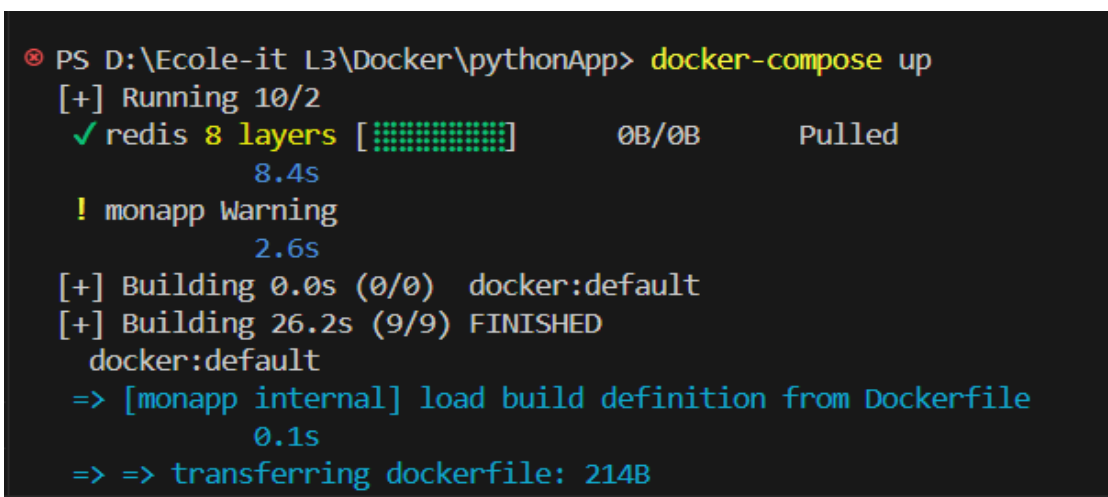
J'ai créé un nouvel utilisateur nommé kaide pour garantir que les commandes soient exécutées par cet utilisateur lors du démarrage du conteneur, améliorant ainsi la sécurité en évitant l'exécution en tant que root.



The screenshot shows the Visual Studio Code interface with the Explorer sidebar on the left and the Dockerfile editor in the center. The Explorer sidebar shows a project named 'PYTHONAPP' with files: '_config.yml', 'app.py' (2 lines), 'docker-compose.yml', 'Dockerfile', 'README', and 'requirements.txt'. The Dockerfile editor shows the following content:

```
1 FROM python:2.7-slim
2 WORKDIR /app
3 COPY . /app
4 RUN pip install -r requirements.txt
5 RUN useradd -m kaide
6 RUN chown -R kaide /app
7 USER kaide
8 EXPOSE 80
9 ENV NOM coca
10 CMD ["python", "app.py"]
11
```

Démarrage du conteneur



The screenshot shows a terminal window with the following output:

```
PS D:\Ecole-it L3\Docker\pythonApp> docker-compose up
[+] Running 10/2
✓ redis 8 layers [██████████] 0B/0B Pulled
8.4s
! monapp Warning
2.6s
[+] Building 0.0s (0/0) docker:default
[+] Building 26.2s (9/9) FINISHED
docker:default
=> [monapp internal] load build definition from Dockerfile
0.1s
=> => transferring dockerfile: 214B
```

```
[+] Running 3/3
✓ Network pythonapp_monreseau Created
0.1s
✓ Container pythonapp-redis-1 Created
0.1s
✓ Container pythonapp-monapp-1 Created
0.1s
Attaching to monapp-1, redis-1
redis-1 | 1:C 18 May 2024 11:59:20.031 * oO00oO00oO00o Redis is starting o
redis-1 | 1:C 18 May 2024 11:59:20.031 * Redis version=7.2.4, bits=64, com
redis-1 | 1:C 18 May 2024 11:59:20.031 # Warning: no config file specified
/to/redis.conf
```

Listing des images

```
PS D:\Ecole-it L3\Docker\pythonApp> docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED
monimage	latest	99f1c11cea55	15 minutes ago

Listing des conteneurs en marche

```
PS D:\Ecole-it L3\Docker\pythonApp> docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
deb9f7c1118a	monimage	"python app.py"	About a minute ago	Up About a minute	0.0.0.0:8084->80/tcp

Exécution du conteneur

```
PS D:\Ecole-it L3\Docker\pythonApp> docker exec -ti deb9f7c1118a bash
kaide@deb9f7c1118a:/app$ ls
Dockerfile README _config.yml app.py docker-compose.yml requirements.txt
kaide@deb9f7c1118a:/app$ cd Dockerfile
bash: cd: Dockerfile: Not a directory
kaide@deb9f7c1118a:/app$ cat docker-compose.yml
version: "3"
services:
  monapp:
    build: .
    image: monimage
    depends_on:
      - redis
    ports:
      - "80:80"
    networks:
      - monreseau
    environment:
      - NOM=les amis
  redis:
    image: redis
    networks:
      - monreseau
```

4 - Sécurité et qualité des conteneurs

1) Identification des vulnérabilités

J'ai utilisé Docker Scout afin de pouvoir scanner les vulnérabilités que peuvent contenir mon application.

Il est important de noter que cet outil est disponible sur les versions à jour de Docker Desktop. Pour l'utiliser sur des versions plus anciennes de Docker Desktop, il faut l'installer via le CLI.

Pour l'utiliser, il faut créer un compte Docker Hub et se connecter via le CLI.

```
Username: lekaide
Password:
Login Succeeded
PS D:\Ecole-it L3\Docker\pythonApp> docker scout cves monimage
  v SBOM of image already cached, 142 packages indexed
  x Detected 34 vulnerable packages with a total of 124 vulnerabilities

## Overview
```

	Analyzed Image
Target	monimage:latest
digest	99f1c11cea55
platform	linux/amd64
vulnerabilities	14C 37H 34M 38L 2?
size	62 MB
packages	142

```
## Packages and Vulnerabilities

4C 4H 3M 4L glibc 2.28-10
pkg:deb/debian/glibc@2.28-10?os_distro=buster&os_name=debian&os_version=10

x CRITICAL CVE-2022-23219
https://scout.docker.com/v/CVE-2022-23219
Affected range : <2.28-10+deb10u2
Fixed version  : 2.28-10+deb10u2
```

Après avoir effectué le scan de mon image, je me suis rendu compte qu'elle était très vulnérable. J'ai ainsi obtenu un rapport de vulnérabilité comprenant

- 14 vulnérabilités critiques (C).
- 37 vulnérabilités de haute sévérité (H).
- 34 vulnérabilités de sévérité moyenne (M).
- 38 vulnérabilités de faible sévérité (L).
- 2 vulnérabilités de sévérité inconnue (?).

2) Résolution des vulnérabilités

Utilisation de python:3.9.6-slim :

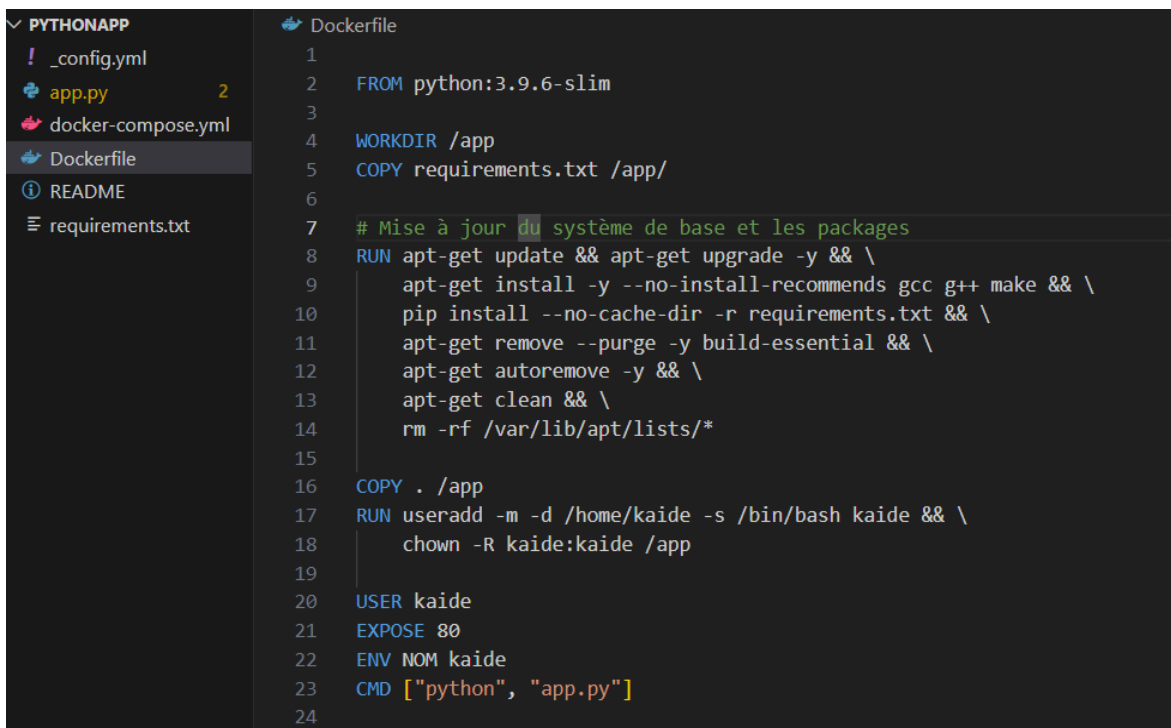
J'ai utilisé une version spécifique de l'image de base pour éviter les problèmes liés à des modifications non attendues dans les mises à jour futures.

Minimiser l'image de base :

slim est déjà une bonne base légère, mais vous pourriez envisager d'utiliser python:3.9-alpine pour une image encore plus légère et sécurisée. Cependant, cela peut nécessiter des modifications supplémentaires dans la gestion des paquets.

Mise à jour des paquets :

J'ai effectué les mises à jour de manière sécurisée et propre pour éviter les failles potentielles, tout combinant les commandes apt-get update et apt-get install dans une seule couche pour réduire la taille de l'image et éviter les problèmes de mise en cache.



```
✓ PYTHONAPP
  ! _config.yml
  📄 app.py 2
  📄 docker-compose.yml
  📄 Dockerfile
  ⓘ README
  ☰ requirements.txt

📄 Dockerfile
1
2 FROM python:3.9.6-slim
3
4 WORKDIR /app
5 COPY requirements.txt /app/
6
7 # Mise à jour du système de base et les packages
8 RUN apt-get update && apt-get upgrade -y && \
9     apt-get install -y --no-install-recommends gcc g++ make && \
10    pip install --no-cache-dir -r requirements.txt && \
11    apt-get remove --purge -y build-essential && \
12    apt-get autoremove -y && \
13    apt-get clean && \
14    rm -rf /var/lib/apt/lists/*
15
16 COPY . /app
17 RUN useradd -m -d /home/kaide -s /bin/bash kaide && \
18     chown -R kaide:kaide /app
19
20 USER kaide
21 EXPOSE 80
22 ENV NOM kaide
23 CMD ["python", "app.py"]
24
```