TP Module 2

Construction d'images Docker et compréhension des layers

Objectif général

Ce TP vous fait construire progressivement plusieurs images Docker afin de comprendre le fonctionnement des layers, l'usage des directives du Dockerfile et la différence entre ENTRYPOINT et CMD.

Chaque étape doit être testée avec `docker build`, `docker run`, puis inspectée avec `docker history` et `docker image inspect`.

Partie 1 — Première image simple : hello-world personnalisé

1.1 Créer l'arborescence du TP

Dans un dossier `tp-dockerfile/`, créer les éléments suivants :

```
tp-dockerfile/
  v1/
  v2/
  v3/
  app/
```

1.2 Créer un script simple

Dans `app/hello.sh` :

```
#!/bin/sh
echo "Hello from inside the container"
```

Rendre le fichier exécutable :

```
chmod +x app/hello.sh
```

1.3 Écrire un premier Dockerfile minimal

Dans `v1/Dockerfile` :

```
FROM alpine:3.20

COPY ../app/hello.sh /hello.sh

CMD ["/hello.sh"]
```

1.4 Construire et exécuter

Construire :

```
docker build -t tp:v1 v1/
```

Exécuter :

```
docker run tp:v1
```

1.5 Analyse des layers

Afficher l'historique :

```
docker history tp:v1
```

Questions :

1. Combien de layers sont visibles ?
2. Quelle instruction a créé la plus grosse layer ?
3. Pourquoi ?

Partie 2 — Ajouter un RUN et observer les layers

2.1 Modifier le Dockerfile

Dans `v2/Dockerfile` :

```
FROM alpine:3.20
RUN apk add --no-cache bash
COPY ../app/hello.sh /hello.sh
CMD ["/bin/bash", "/hello.sh"]
```

2.2 Construire et exécuter

```
docker build -t tp:v2 v2/
docker run tp:v2
```

2.3 Questions d'analyse

1. Quelle différence notable dans `docker history` entre v1 et v2 ?
2. L'installation de bash a-t-elle créé une nouvelle layer ? Pourquoi ?
3. Les layers de v1 ont-elles été réutilisées ?

Partie 3 — Réorganisation pour optimiser le cache

3.1 Ajouter un fichier requirements

Créer dans `app/requirements.txt` :

```
flask==3.0.0
```

3.2 Créer v3/Dockerfile

```
FROM python:3.12-slim
WORKDIR /app
COPY ../app/requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY ../app/ ./
CMD ["python3", "-c", "print('App directory loaded')"]
```

3.3 Construire et exécuter

```
docker build -t tp:v3 v3/
docker run tp:v3
```

3.4 Modifier un fichier et reconstruire

Modifier `hello.sh` dans `app/` en changeant le texte affiché.
Reconstruire :

```
docker build -t tp:v3 v3/
```

Observation :
Le pip install a-t-il été ré-exécuté ? Pourquoi ou pourquoi pas ?

Explication attendue :
Le cache dépend de la position du COPY. Seul le COPY final a changé.

Partie 4 — Comprendre ENTRYPOINT et CMD

4.1 Nouvelle image dédiée

Créer `v4/Dockerfile` :

```
FROM alpine:3.20
COPY ../app/hello.sh /hello.sh
ENTRYPOINT ["/hello.sh"]
CMD ["default-arg"]
```

Modifier `app/hello.sh` :

```
#!/bin/sh
echo "Argument reçu: $1"
```

4.2 Construire

```
docker build -t tp:v4 v4/
```

4.3 Tester sans argument

```
docker run tp:v4
```

Résultat attendu :
Argument reçu: default-arg

4.4 Tester avec un argument

```
docker run tp:v4 override
```

Résultat attendu :
Argument reçu: override
(ENTRYPOINT reste identique, CMD est remplacé.)

4.5 Analyse

1. Quelle est la différence de rôle entre ENTRYPOINT et CMD dans cette image ?
2. Pourquoi ENTRYPOINT n'est-il pas remplacé ?
3. À quoi servent ENTRYPOINT + CMD dans une image CLI ?

Partie 5 — Script wrapper avec ENTRYPOINT

5.1 Ajouter un script plus complexe

Dans `app/wrapper.sh` :

```
#!/bin/sh
echo "Préparation de l'environnement..."
sleep 1
exec "$@"

chmod +x app/wrapper.sh
```

5.2 Nouveau Dockerfile

Créer `v5/Dockerfile` :

```
FROM alpine:3.20
COPY ../app/wrapper.sh /usr/local/bin/wrapper.sh
ENTRYPOINT ["wrapper.sh"]
CMD ["echo", "Commande par défaut"]
```

5.3 Construire et tester

```
docker build -t tp:v5 v5/
docker run tp:v5
```

Ensuite :

```
docker run tp:v5 ls -l /
```

Observation :
ENTRYPOINT sert ici de bootstrap. Les arguments sont transmis grâce à `exec "$@"`.

Partie 6 — Multi-stage build (approche avancée)

6.1 Créer un petit programme

Dans `app/main.c` :

```
#include <stdio.h>
int main() {
    printf("Hello from a compiled binary\n");
    return 0;
}
```

6.2 Dockerfile multi-stage

Créer `v6/Dockerfile` :

```
FROM alpine:3.20 AS build
RUN apk add --no-cache gcc musl-dev
```

```
COPY ../app/main.c /src/main.c
RUN gcc /src/main.c -o /src/app

FROM alpine:3.20
COPY --from=build /src/app /usr/local/bin/app
ENTRYPOINT ["app"]
```

6.3 Construire et exécuter

```
docker build -t tp:v6 v6/
docker run tp:v6
```

6.4 Analyse

1. Combien de layers dans la deuxième étape ?
2. Pourquoi le binaire est-il beaucoup plus petit que dans un build non multi-stage ?
3. Qu'est-ce que cela change en termes de sécurité ?

Partie 7 — Bilan et checklist de validation

À ce stade, vous devez être capable de :

- Construire une image simple avec CMD
- Ajouter des RUN et analyser les layers
- Comprendre les effets du cache entre RUN, COPY et WORKDIR
- Distinguer clairement ENTRYPOINT et CMD
- Créer une image CLI configurable
- Utiliser un wrapper avec ENTRYPOINT
- Créer une image multi-stage pour réduire la taille finale