

Conteneurs — Création d'images

Docker & OCI

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-09-30

Table des matières

1 Objectifs	1	15 Multi-architecture (buildx)	10
2 Images de conteneurs (OCI)	1	16 À retenir	10
2.1 Exemple : Ubuntu	2		
2.2 Architecture en couches	2		
3 Création manuelle d'une image	2		
3.1 Exemple : installer Git manuellement	3		
4 Dockerfile (méthode recommandée)	3		
4.1 Dockerfile — Structure de base	3		
4.2 Dockerfile : Elements de base	5		
4.3 Build d'une image et exécution	5		
5 Gestion des tags 1/2	5		
5.1 Gestion des tags 2/2	5		
5.2 Utilisation de l'image	5		
6 Exercice 1 — Une Image Docker simple ...	6		
7 Publication sur un registry	6		
8 HEALTHCHECK	6		
9 ENTRYPOINT vs CMD	6		
9.1 Formats pour ENTRYPOINT / CMD	7		
10 Exercice 2 — ARG et ENV	7		
11 Multi-stage build	7		
11.1 Exemple multi-étapes (C)	7		
11.2 Résultat	8		
12 Exposition des ports	8		
13 Exercice 3 — Multi-stage build	8		
14 Sécurité — Bonnes pratiques	9		
14.1 Utilisateur non-root	9		
14.2 Secrets avec BuildKit	9		

1 Objectifs

À la fin de ce cours, vous saurez :

- Comprendre la structure d'une image OCI
- Créer une image manuellement (et pourquoi éviter)
- Construire une image avec un Dockerfile
- Utiliser ENTRYPOINT et CMD
- Optimiser avec les builds multi-étapes
- Publier et analyser une image

2 Images de conteneurs (OCI)

Une image suit la **spécification OCI** (Open Container Initiative).

- Ensemble de **couches en lecture seule**
- Chaque couche = delta par rapport à la précédente
- Métadonnées séparées (CMD, ENV, labels...)
- Nom d'image formel — syntaxe générale : `[[registry/] [namespace/] repository[:tag] [@digest]]`.
 - ▶ **repository** : partie obligatoire qui identifie le dépôt (nom du répertoire).

3. Valider avec `docker commit`

- ✗ Non reproductible
- ✗ Historique opaque
- ✗ Mauvaise pratique en production

3.1 Exemple : installer Git manuellement

- Lancer un conteneur Ubuntu : `docker run --name my-ubuntu --interactive ubuntu:jammy bash`
- Installer Git en interactif : `apt-get update && apt-get install -y git`
- Valider l'image et supprimer le conteneur intermédiaire.

```
docker commit my-ubuntu mygit:latest
docker rm my-ubuntu
```

```
sha256:3262da9b45c86bf04a65e0981c6edd5985035db870294831f70d9fa656f25226
my-ubuntu
```

- Une nouvelle image `mygit:latest` est créée.

```
docker run --rm mygit git --version
```

```
git version 2.34.1
```

- Historique de l'image `mygit`.

```
docker history mygit
```

IMAGE SIZE	CREATED COMMENT	CREATED BY
3262da9b45c8 164MB	2 seconds ago	bash -
bf7f4568d957 0B	3 weeks ago	/bin/sh -c #(nop) CMD ["/bin/bash"]
<missing>	3 weeks ago	/bin/sh -c #(nop) ADD file:81c01921c5f642ac2...

```
78.1MB
<missing>      3 weeks ago    /bin/sh
-c #(nop) LABEL org.opencontainers...
0B
<missing>      3 weeks ago    /bin/sh
-c #(nop) ARG LAUNCHPAD_BUILD_ARCH
0B
<missing>      3 weeks ago    /
bin/sh -c #(nop) ARG RELEASE
0B
```

➔ Une seule couche opaque ajoutée

4 Dockerfile (méthode recommandée)

- Fichier texte avec instructions pour construire une image

Avantages :

- Reproductible
- Versionnable (Git)
- Automatisable
- Lisible

```
# Exemple simple
FROM ubuntu:jammy
RUN apt-get update && apt-get install
-y git
ENTRYPOINT ["git"]
CMD ["--version"]
```

4.1 Dockerfile — Structure de base

Les exemples suivants sont accessibles dans le dépôt :

☒ : `ebpro/notebook-containers-intro-sample-python-helloworld`

Détails

- **URL:** <https://github.com/ebpro/notebook-containers-intro-sample-python-helloworld>
- **Branch:** develop

- **Local Path:** /home/jovyan/work/examples/github/ebpro/notebook-containers-intro-sample-python-helloworld
- **Latest Commit:** 2fcfd54 - feat(stage): sota 2026 uv+stage
- **Timestamp:** 2026-09-30T06:58:11Z

Clone Command:

```
git clone -b develop https://github.com/ebpro/notebook-containers-intro-sample-python-helloworld
```

```
/home/jovyan/work/examples/github/ebpro/notebook-containers-intro-sample-python-helloworld
├─ Dockerfile
├─ Dockerfile.stage
├─ README.md
├─ hello.py
└─ requirements.txt
```

1 directory, 5 files

```
1  # -----
2  # Base image
3  # -----
4  FROM python:3.13-slim
5
6  # -----
7  # Build arguments (available only during image build)
8  # -----
9  ARG
BUILD_DATE="1970-01-01T00:00:00Z"
10
11 # -----
12 # OCI image metadata
13
# https://github.com/opencontainers/image-spec/blob/main/annotations.md
```

```
14 # -----
15 LABEL org.opencontainers.image.authors="emmanuel.b
tln.fr" \
16     org.opencontainers.image.created="${BUILD_DATE}"
17
18 # -----
19 # Environment variables
20 # -----
21 ENV NAME="John Doe"
22
23 # -----
24 # Application directory
25 # -----
26 WORKDIR /app
27
28 # -----
29 # Install Python dependencies
30 # Copy requirements first to maximize Docker cache reuse.
31 # -----
32 COPY requirements.txt .
33
34 RUN pip install --no-cache-dir -r requirements.txt
35
36 # -----
37 # Copy application source
38 # -----
39 COPY hello.py .
40
41 # -----
42 # Run as a non-root user (container security best practice)
43 # -----
44 RUN useradd --create-home appuser
45 USER appuser
46
47 # -----
48 # Default command
49 # -----
```

50 ENTRYPOINT

```
["python", "hello.py"]
```

4.2 Dockerfile : Elements de base

- **FROM** : Image de base (ex: python:3.9).
- **WORKDIR** : Définit le dossier de travail (ex: /app).
- **COPY** : Copie les fichiers locaux dans l'image.
- **RUN** : Exécute des commandes : (ex. Installe les dépendances, `pip install`).
- **ENTRYPOINT** et **CMD** : Définit la commande par défaut.

4.3 Build d'une image et exécution

- Commande `docker image build`
- Contexte de build : répertoire avec Dockerfile ici .
 - `.dockerignore` exclure des fichiers du contexte de build ce qui accélère et sécurise la construction,

```
docker image build -t mon-app:0.0.1 .
```

```
docker run --rm mon-app:0.0.1
```

```
2026-09-30 06:58:38,745 INFO Hello
John Doe, I'm Python running inside
a container!
```

```
2026-09-30 06:58:38,746 INFO Here is
your DataFrame:
```

	Name	Age
0	Pierre	22
1	Paul	35
2	Marie	58

5 Gestion des tags 1/2

Bonnes pratiques :

- Option `--tag / -t` pour nommer l'image
- Tag multiples pour une même image avec versionnage sémantique (semver) :

▸ Versions complètes : 1.2.3

▸ Versions majeures : 1, 1.2

▸ Alias : latest

- Format : `[registry/] [namespace/] repository:tag`

5.1 Gestion des tags 2/2

Tag à la construction (`--tag / -t`) :

```
docker image build \
  --tag helloworld:0.0.1 \
  .
```

Tag additionnels (`docker tag`) :

- Gestion des versions plus "large"
- Ajout d'un compte sur un registry pour partage :
 - `${DOCKERHUB_USERNAME}` = votre nom d'utilisateur Docker Hub (ou autre registry)

```
docker image tag helloworld:0.0.1 \
  docker.io/${DOCKERHUB_USERNAME}/
helloworld:0
```

```
docker image tag helloworld:0.0.1 \
  docker.io/${DOCKERHUB_USERNAME}/
helloworld:0.1
```

```
docker image tag helloworld:0.0.1 \
  docker.io/${DOCKERHUB_USERNAME}/
helloworld:0.0.1
```

```
docker image tag helloworld:0.0.1 \
  docker.io/${DOCKERHUB_USERNAME}/
helloworld:latest
```

5.2 Utilisation de l'image

- Commande `docker run`
- `-e / --env` pour variables d'environnement dans le conteneur

```
docker run --rm docker.io/
${DOCKERHUB_USERNAME}/helloworld
```

```
2026-09-30 06:58:43,216 INFO Hello
John Doe, I'm Python running inside
a container!
2026-09-30 06:58:43,216 INFO Here is
your DataFrame:
      Name  Age
0  Pierre   22
1    Paul   35
2   Marie   58
```

```
docker run --rm -e NAME=Pierre
docker.io/${DOCKERHUB_USERNAME}/
helloworld:0.0.1
```

```
2026-09-30 06:58:44,993 INFO Hello
Pierre, I'm Python running inside a
container!
2026-09-30 06:58:44,993 INFO Here is
your DataFrame:
      Name  Age
0  Pierre   22
1    Paul   35
2   Marie   58
```

6 Exercice 1 — Une Image Docker simple

Faire l'exercice :

Pratique Container Image — Exercice 1

7 Publication sur un registry

Étapes :

1. (Une fois) :
 - création d'un compte (docker hub ou autres)
 - connexion : `docker login`

1. Build + Tag correct (registry + namespace (nom d'utilisateur)).
2. Poussez les tags (`docker push`)

```
docker push docker.io/
${DOCKERHUB_USERNAME}/app:latest
```

8 HEALTHCHECK

L'instruction `HEALTHCHECK` permet de définir une commande pour vérifier la santé d'un conteneur en cours d'exécution. Docker exécute périodiquement cette commande et utilise son code de retour pour déterminer l'état du conteneur : 0 (sain), 1 (non sain) ou 2 (réservé).

Exemple :

```
HEALTHCHECK --interval=30s --
timeout=3s --start-period=5s --
retries=3 \
  CMD curl --max-time 30 -f http://
localhost/ || exit 1
```

Options principales :

- `--interval`: fréquence de vérification (ex. 30s).
- `--timeout`: délai maximum d'exécution de la commande (ex. 3s).
- `--start-period`: délai initial avant la première vérification (ex. 5s).
- `--retries`: nombre d'échecs consécutifs avant de considérer le conteneur comme non sain.

Les `HEALTHCHECK` sont utiles pour l'orchestration (restart policies, load balancers) et pour détecter des services défectueux.

9 ENTRYPOINT vs CMD

- Pour définir le comportement par défaut d'une image
 - Une image peut définir un `ENTRYPOINT` et un `CMD`

- **ENTRYPOINT** + **CMD** = commande complète
- Si **CMD** absent, arguments passés à **docker run** sont ajoutés à **ENTRYPOINT**
- Si **ENTRYPOINT** absent, **docker run** utilise **CMD** ou les arguments passés
- **ENTRYPOINT** : commande principale
- **CMD** : arguments par défaut

Directive	Rôle
ENTRYPOINT	Comportement principal
CMD	Paramètres remplaçables

```
ENTRYPOINT ["git"]
CMD ["--version"]
```

```
docker run mygit
docker run mygit status
```

➔ Image utilisable comme un programme

9.1 Formats pour **ENTRYPOINT** / **CMD**

- Exec form : `["executable", "param1"]` (Recommandé : transmet les signaux OS comme SIGTERM).
- Shell form : `executable param1` :
 - À éviter (l'exécutable devient un enfant de `/bin/sh -c` et ne reçoit pas les signaux d'arrêt, ce qui rend le docker stop lent).
 - Mais permet le globbing (*) et l'interpolation des variables SHELL
 - Solution : utiliser un script `entrypoint.sh`

10 Exercice 2 — ARG et ENV

Faire l'exercice :

Pratique Container Image — Exercice 1

11 Multi-stage build

Objectifs :

- Séparer build et runtime
- Réduire la taille finale

- Supprimer toolchains et sources

Fonctionnement :

- Plusieurs étapes **FROM** dans un Dockerfile
- Chaque étape peut copier des artefacts de l'étape précédente (`COPY --from=...`)
- L'étape finale est l'image résultante

11.1 Exemple multi-étapes (C)

Les exemples suivants sont accessibles dans le dépôt :

☒ : [ebpro/notebook-containers-intro-sample-c](https://github.com/ebpro/notebook-containers-intro-sample-c)

Détails

- **URL**: <https://github.com/ebpro/notebook-containers-intro-sample-c>
- **Branch**: develop
- **Local Path**: `/home/jovyan/work/examples/github/ebpro/notebook-containers-intro-sample-c`
- **Latest Commit**: ede5715 - feat(2026): 2026 fix
- **Timestamp**: 2026-09-30T06:58:46Z

Clone Command:

```
git clone -b develop https://github.com/ebpro/notebook-containers-intro-sample-c
```

```
1  # Build stage
2  FROM gcc:14-bookworm AS builder
3
4  # An argument sets at build
   command with --build-arg
5  # It as a default value
6
   ARG BUILD_DATE=1970-01-01T00:00:00Z
7
8  # key-value pair as image
   metadata
9  LABEL
```

```

maintainer="emmanuel.bruno@univ-
tln.fr"
10 # See http://label-schema.org/rc
1/ for a list of usefull labels
11 LABEL org.label-schema.build-
date=$BUILD_DATE
12
13 WORKDIR /src/app
14 COPY helloworld.c .
15 RUN gcc -Wall -Wextra -Werror
-02 -fPIE -pie -D_FORTIFY_SOURCE=2
-static-libgcc helloworld.c -o
helloworld
16
17 # Runtime stage
18 FROM debian:bookworm-slim
19
20 # Create non-root user
21 RUN groupadd -r appuser &&
useradd -r -g appuser appuser
22 WORKDIR /app
23
24 # Copy only the compiled binary
25 COPY --from=builder --
chown=appuser:appuser /src/app/
helloworld /app/helloworld
26
27 # Switch to non-root user
28 USER appuser
29
30 # Set executable permissions
31 RUN chmod 550 /app/helloworld
32
33 ENTRYPOINT ["/app/helloworld"]

```

```

docker image build \
    --tag ${DOCKERHUB_USERNAME}/
helloworld_c:0.0.1 \
    .

docker

```

```

run --rm ${DOCKERHUB_USERNAME}/
helloworld_c:0.0.1 Paul

```

```

Hostname : 39f10ed86297
PID      : 1
UID      : 999
Hello World!

```

11.2 Résultat

- Chaîne de build est versionnable, reproduc-
tible (gcc, sources, etc. dans le Dockerfile)
- Image finale très légère (uniquement
l'exécutable + libs)
 - Aucun compilateur présent
 - Sécurité accrue
 - Démarrage rapide

12 Exposition des ports

- **EXPOSE** dans le Dockerfile pour documenter
les ports utilisés
- **-p / --publish** dans **docker run** pour map-
per les ports
- Exemple : application web sur le port 80

Dans le Dockerfile :

```
EXPOSE 80
```

```
docker run -p 8080:80 myapp
```

EXPOSE ne publie pas le port, il documente
seulement.

13 Exercice 3 — Multi-stage build

Faire l'exercice :

Pratique Container Image — Exercice 3

14 Sécurité — Bonnes pratiques

Principes essentiels pour réduire la surface d'attaque et rendre les images sûres :

- **Images minimalistes** : choisir des bases petites (Alpine, distroless) ou builder multi-étapes pour ne garder que l'exécutable final.
- **Éviter les secrets dans l'image** : ne jamais écrire de mots de passe/jetons dans les Dockerfile. Utiliser BuildKit secrets (`--mount=type=secret`) ou des volumes/variables d'environnement au runtime.
- **Ne pas exécuter en root** : définir un utilisateur non-root dans le Dockerfile.

```
RUN addgroup -S appgroup && adduser -S appuser -G appgroup
USER appuser
```

- **Pinning & immutabilité** : référencer des images par digest (`@sha256:...`) ou pinner les versions de paquets pour éviter des mises à jour non contrôlées.
- **Nettoyage des artefacts** : supprimer les caches d'installateurs et réduire le nombre de layers (ex. `apt-get clean && rm -rf /var/lib/apt/lists/*`).
- **Scanner et vérifier** : utiliser des scanners (ex. `docker scan`, `trivy`) et générer un SBOM. Ex : `trivy image --severity HIGH,CRITICAL myimage:latest`.
- **Limiter les privilèges au runtime** : appliquer des profils seccomp/AppArmor, retirer les capacités inutiles (`--cap-drop`), monter le FS en lecture seule quand possible (`--read-only`).
- **Ressources et isolation** : définir limites mémoire/CPU et pids pour limiter l'impact d'un processus compromis.

14.1 Utilisateur non-root

```
RUN adduser -D appuser
USER appuser
```

14.2 Secrets avec BuildKit

- Attention : ne pas inclure de secrets dans l'image finale
 - **JAMAIS** dans `ENV` ni non plus dans `ARG` (persistants dans l'image)
 - Utiliser les secrets temporaires de BuildKit

```
FROM alpine:3.18

# Dépendance nécessaire
RUN apk add --no-cache curl

# ARG pour une valeur non sensible
# (ex : URL ou nom d'environnement)
ARG     API_URL=https://example.com/secure-data

# Utilisation du secret via BuildKit
RUN --mount=type=secret,id=api_key \
    curl --max-time 30 -H
    "Authorization: Bearer $(cat /run/
    secrets/api_key)" \
    ${API_URL} -o /tmp/data.json

# On peut ajouter un step pour
# vérifier que le fichier existe
RUN test -f /tmp/data.json

CMD ["cat", "/tmp/data.json"]
```

```
docker build \
    --secret id=api_key,src=./
    api_key.txt \
    -t myapp:latest .
```

15 Multi-architecture (buildx)

- Construire pour plusieurs architectures (amd64, arm64, etc.)
- Utiliser `docker buildx` ou `docker build --platform`

```
#| echo: true
#| output: true
docker build \
  --platform linux/amd64,linux/arm64
\
-t docker.io/${DOCKERHUB_USERNAME}/
helloworld_c:0.0.1 \
.
```

- i Pour publier une image multi-architecture sur un registry, utiliser `docker buildx build --push`.

16 À retenir

- Une image ≠ un conteneur
- `Dockerfile` pour créer des images
- `ENTRYPOINT` définit le comportement
- `CMD` fournit des arguments par défaut
- Multi-stage sépare build et runtime
- `push` Publier sur un registry
- Toujours penser sécurité & reproductibilité