

Conteneurs — Création d'images

Docker & OCI

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

2026-10-14

Table des matières

1 Objectifs	1	15 Multi-architecture (buildx)	9
2 Images de conteneurs (OCI)	1	16 À retenir	9
2.1 Exemple : Ubuntu	2		
2.2 Architecture en couches	3		
3 Création manuelle d'une image	3		
3.1 Exemple : installer Git manuellement	3		
4 Dockerfile (méthode recommandée)	4		
4.1 Dockerfile — Structure de base	4		
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	6		
6 Exercice 1 — Une Image Docker simple ...	6		
7 Publication sur un registry	6		
8 HEALTHCHECK	6		
9 ENTRYPOINT vs CMD	7		
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	8		
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).

- **registry** : hôte du registre (optionnel); si absent on entend en pratique le registre Docker Hub (docker.io).
- **namespace** : segment optionnel pour regrouper les dépôts (sur Docker Hub, l'espace **library** est implicite pour les images officielles).
- **tag** : référence optionnelle lisible (ex. **:latest**, **:1.2.3**); si aucun tag n'est fourni, **latest** est couramment utilisé comme valeur par défaut pour l'usage, mais l'absence de tag signifie que l'image peut aussi être désignée par un **digest**.
- **digest** : identifiant immuable optionnel (forme **sha256:<hex>**); lorsqu'il est présent (**@digest**) il prend la priorité sur le **tag** pour désigner de façon exacte une image. (Les parties entre crochets **[]** sont optionnelles selon la spécification de référence des images.)

➔ Une image **n'est pas** un conteneur mais le modèle de fichier utilisé pour en créer un ou plusieurs.

2.1 Exemple : Ubuntu

- **pull** commande pour récupérer une image depuis un registry
 - Les registries sont des dépôts d'images (Docker Hub, GitHub Container Registry, etc.)
- Image officielle Ubuntu : **ubuntu:jammy**
- Basée sur plusieurs couches empilées

```
docker pull ubuntu:jammy
```

```
jammy: Pulling from library/ubuntu
```

```
98c4455a9898: Pulling fs layer
```

```
98c4455a9898:
Downloading 310.7kB/29.75MB
```

```
98c4455a9898:
Downloading 6.537MB/29.75MB
```

```
98c4455a9898:
Downloading 14.32MB/29.75MB
```

```
98c4455a9898:
Downloading 23.35MB/29.75MB
```

```
98c4455a9898: Verifying Checksum
```

```
98c4455a9898: Download complete
```

```
98c4455a9898:
Extracting 327.7kB/29.75MB
```

```
98c4455a9898:
Extracting 3.277MB/29.75MB
```

```
98c4455a9898:
Extracting 5.898MB/29.75MB
```

```
98c4455a9898:
Extracting 7.864MB/29.75MB
```

```
98c4455a9898:
Extracting 9.175MB/29.75MB
```

```
98c4455a9898:
Extracting 10.81MB/29.75MB
```

```
98c4455a9898:
Extracting 14.09MB/29.75MB
```

```
98c4455a9898:
Extracting 17.69MB/29.75MB
```

```
98c4455a9898:
Extracting 20.97MB/29.75MB
```

```
98c4455a9898:
Extracting 23.59MB/29.75MB
```

```

98c4455a9898:
Extracting 25.56MB/29.75MB

98c4455a9898:
Extracting 25.89MB/29.75MB

98c4455a9898:
Extracting 28.18MB/29.75MB

98c4455a9898:
Extracting 28.84MB/29.75MB

98c4455a9898:
Extracting 29.49MB/29.75MB

98c4455a9898:
Extracting 29.75MB/29.75MB

98c4455a9898: Pull complete
Digest: sha256:5ec03bb3441e8b0bf3b4f9cd4629a1de765010dc3035b58da5a6cf026486401
Status: Downloaded newer image for
ubuntu:jammy
docker.io/library/ubuntu:jammy

```

- Liste des couches

```
docker history ubuntu:jammy
```

IMAGE SIZE	CREATED COMMENT	CREATED BY
e8df6c74f650	2 weeks ago	/
bin/sh -c #(nop) CMD ["/bin/bash"]		
0B		
<missing>	2 weeks ago	/bin/sh -
c #(nop) ADD file:b0bf3f64519bf10a5...		
78.1MB		
<missing>	2 weeks ago	/bin/sh -
c #(nop) LABEL org.opencontainers...		
0B		
<missing>	2 weeks ago	/bin/sh
-c #(nop) ARG LAUNCHPAD_BUILD_ARCH		

```

0B
<missing>      2 weeks ago    /
bin/sh -c #(nop) ARG RELEASE
0B

```

- dans docker history (même après un pull)
 - Docker ne peut pas afficher l'ID d'une couche.
 - l'historique stocké dans l'image ne contient pas toujours les IDs des couches,
 - ou Docker n'a pas conservé ces couches dans son cache local.

2.2 Architecture en couches

Avantages :

- Mutualisation des couches communes
- Réduction de l'espace disque
- Téléchargement incrémental
- Cache efficace lors des builds

3 Création manuelle d'une image

Méthode possible mais **déconseillée** :

1. Lancer un conteneur
2. Modifier le système de fichiers
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
```

```
sha256:881ee4f5ca0f1d1d6057ce01fe5f971ae7125e32e2e3f7e015d9e8424a4b99b6
my-ubuntu
```

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

```
docker run --label ebpro-render=true
--rm mygit git --version
```

```
git version 2.34.1
```

- Historique de l'image `mygit`.

```
docker history mygit
```

IMAGE SIZE	CREATED COMMENT	CREATED BY
881ee4f5ca0f	4 seconds ago	bash - 157MB
e8df6c74f650	2 weeks ago	/bin/sh -c #(nop) CMD ["/bin/bash"] 0B
<missing>	2 weeks ago	/bin/sh -c #(nop) ADD file:b0bf3f64519bf10a5... 78.1MB
<missing>	2 weeks ago	/bin/sh -c #(nop) LABEL org.opencontainers... 0B
<missing>	2 weeks ago	/bin/sh -c #(nop) ARG LAUNCHPAD_BUILD_ARCH 0B
<missing>	2 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/sample-containers-python-hello-world`

Détails

- **URL:** <https://github.com/ebpro/sample-containers-python-helloworld>
- **Branch:** `develop`
- **Local Path:** `/home/jovyan/work/github/ebpro/sample-containers-python-helloworld`
- **Latest Commit:** `9f358d3 - chore: add _quarto-utils submodule + .devcontainer (SOTA pattern)`
- **Timestamp:** `2026-10-14T14:50:27Z`

Clone Command:

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

```

/home/jovyan/work/github/ebpro/
sample-containers-python-helloworld
├── Dockerfile
├── Dockerfile.stage
├── README.md
├── _quarto-utils
├── hello.py
└── requirements.txt

```

2 directories, 5 files

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 --label ebpro-render=true
--rm mon-app:0.0.1
```

```

2026-10-14 14:50:56,128 INFO Hello
John Doe, I'm Python running inside
a container!
2026-10-14 14:50:56,128 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 partager :
 - `${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 --label ebpro-
render=true --rm docker.io/
${DOCKERHUB_USERNAME}/helloworld
```

```
2026-10-14 14:51:00,720 INFO Hello
John Doe, I'm Python running inside
a container!
```

```
2026-10-14 14:51:00,720 INFO Here is
your DataFrame:
```

	Name	Age
0	Pierre	22
1	Paul	35
2	Marie	58

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

```
2026-10-14 14:51:02,652 INFO Hello
Pierre, I'm Python running inside a
container!
```

```
2026-10-14 14:51:02,652 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 défaillants.

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
<code>ENTRYPOINT</code>	Comportement principal
<code>CMD</code>	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/sample-containers-c`

Détails

- **URL**: `https://github.com/ebpro/sample-containers-c`
- **Branch**: `develop`
- **Local Path**: `/home/jovyan/work/github/ebpro/sample-containers-c`
- **Latest Commit**: `f81322d - chore: add _quarto-utils submodule + .devcontainer (SOTA pattern)`
- **Timestamp**: `2026-10-14T14:51:04Z`

Clone Command:

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

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

docker run --label ebpro-render=true
--rm      ${DOCKERHUB_USERNAME}/
helloworld_c:0.0.1 Paul
```

```
Hostname : 74a29cba6436
PID       : 1
UID       : 999
Hello World!
```

11.2 Résultat

- Chaîne de build est versionnable, reproductible (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 mapper 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