Université Sorbonne Paris Nord
École d'ingénieur Sup Galilée

Séance2 / Du développement au déploiement d'applications web

Compte rendu

Réalisé par :

Célian WIRTZ (celian.wirtz@edu.univ-paris13.fr)

3ème année
Année universitaire : 2025/2026

1. Concepts clés

Image vs Container

- **Image** : snapshot immuable (template) composé de couches (layers). Contient le système de fichiers, les dépendances, la configuration. Exemple : alpine:latest, python:3.11.
- **Container** : instance en cours d'exécution d'une image (runtime). On peut en créer/stopper/supprimer autant qu'on veut à partir d'une même image.
  Intérêt de la distinction : l'image permet la reproductibilité ; les containers permettent l'échelle (lancer N instances identiques rapidement).

Isolation

- Les containers partagent le **même noyau** de l'hôte (kernel) — d'où **isolation moins forte** que VM (qui ont leur propre noyau).
- Isolation fournie par **namespaces**, **cgroups**, et autres mécanismes (seccomp, AppArmor, capabilities).
- Implication pratique : plus léger que VM (démarrage + taille) mais attention à la sécurité et aux privilèges.

Pourquoi les images sont petites (Alpine, etc.)

- Images minimalistes comme **Alpine** réduisent la taille parce qu'elles contiennent seulement l'essentiel.
- Dans ces images il manque souvent des outils (vim, curl, bash...) — il faut les installer si nécessaire.

2. Quand et quoi utiliser

- **Docker Compose** : orchestration sur une **même machine** — idéal pour une app multi-conteneurs en dev ou prod simple.
- **Docker Swarm** : orchestrateur natif Docker, simple à prendre en main pour créer un petit cluster.
- **Kubernetes** : standard industriel pour l'orchestration à grande échelle (prod), plus d'outils et complexité.

3. Commandes de base

Exécution et interaction

créer & lancer un container interactif à partir d'une image

docker run -it --name moncontainer alpine sh

-i : stdin ouvert ; -t : alloue pseudo-tty (on écrit souvent -it ou -ti)

- docker run : crée (si nécessaire) et démarre un container à partir d'une image.
- docker exec -it <container> sh : exécute un shell **dans un container déjà en cours d'exécution**.
- docker start <container> : démarre un container arrêté (ne crée pas).

Détaché / foreground

- -d ou --detach : démarre le container en arrière-plan (detached).

docker run -d --name nginx1 -p 8080:80 nginx

mappe le port 8080 de l'hôte sur le port 80 du container

Lister / voir

docker ps # containers en cours d'exécution

docker ps -a # tous les containers (y compris stoppés)

docker images # liste des images locales

Supprimer / nettoyer

docker rm <container> # supprime un container (doit être arrêté)

docker rm -f <container> # force la suppression (arrête puis supprime)

docker rm -f $(docker ps -aq) # supprime tous les containers

docker rmi <image> # supprimer une image

docker image prune # supprime images “dangling” (non référencées)

docker image prune -a # supprime images non utilisées

docker system prune -a --volumes # nettoyage complet (containers, images, réseaux, volumes)

docker volume prune # supprime volumes non utilisés

4. Installer des paquets dans une image minimale

Dans Alpine (package manager apk) :

depuis un shell dans un container Alpine

apk add --no-cache vim

- --no-cache évite de conserver l'index des paquets dans l'image (réduit la taille).

Dans Debian/Ubuntu :

apt-get update && apt-get install -y vim

Attention : installer quelque chose **dans un container en cours d’exécution** modifie l’instance en RAM / couche writable. Pour que l’installation devienne réutilisable comme image, il faut soit **commit** (ou mieux : créer un Dockerfile et rebuild).

5. Construire une image (Dockerfile)

Un Dockerfile minimal :

```
# Dockerfile

FROM debian:bullseye-slim

LABEL Cel="Cel@example.com"

RUN apt-get update

COPY . /app

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

```
docker build -t monimage:latest .

docker run --name moncontainer -it monimage
```

6. Créer une image à partir d'un container (sauvegarder l'état)

- **docker commit** : transforme l'état d'un container en image :

```
docker commit <container-id> monimage:tag
```

- **docker export / import** :
  - docker export exporte le **filesystem** d'un container (perd l'historique des couches, pas de métadonnées).
  - docker import crée une image à partir d'une archive tar issue de export.
  - Utile pour migrations rapides, mais **préfère docker save/load** pour les images.
- **docker save / docker load** (**recommandé** pour transférer des images entre machines sans Docker Hub) :

```
docker save -o monimage.tar monimage:tag

# transférer monimage.tar

docker load -i monimage.tar
```

Comparaison :
save/load = sauvegarde **d'une image** (garde couches, tags, métadonnées).
export/import = export du **filesystem d'un container** (perd infos de couche, ports exposés, history).

7. Partager une image sur Docker Hub

- **Docker Hub** : workflow typique

```
docker login

docker tag monimage:tag mondockerhubuser/monimage:tag

docker push mondockerhubuser/monimage:tag
```

8. Réseaux, ports & volumes

- **Ports** : -p <host>:<container> (host:container). Exemple :

docker run -d -p 8080:80 nginx

nginx écoute sur 80 dans le container, accessible sur localhost:8080

- **Volumes** (persistance des données) :

docker run -v /host/path:/container/path ...

ou avec volumes nommés : -v monvolume:/data

- Sans volumes, tout ce qui est créé dans le container est éphémère (perdu si container supprimé).

9. Différences pratiques run vs exec vs start

- docker run : crée + démarre + (optionnellement) exécute une commande.
- docker start : démarre un container existant (arrêté).
- docker exec : exécute une commande **dans** un container qui est déjà en cours d'exécution.

Exemple :

docker run -it --name test alpine sh # crée + démarre + shell

ctrl-d ou exit → container s'arrête

docker start test # redémarre le container (sans ouvrir de shell)

docker exec -it test sh # ouvre un shell dans le container en cours

10. Sécurité — bonnes pratiques rapides

- Évite de lancer des containers en **privilégié** (--privileged) sauf si nécessaire.
- Limite les capacités (--cap-drop, --cap-add), utilise **user namespaces** / **run as non-root** dans Dockerfile (USER).
- Met à jour images de base et évite d'installer des paquets inutiles.

11. Bonnes pratiques de construction d'images

- Utiliser des images officielles quand disponibles (python:, node:, nginx:).
- Minimiser la taille : supprimer caches, utiliser --no-cache (apk) ou --no-cache-dir (pip), nettoyer apt lists.
- Faire des builds reproductibles (pin de version), éviter d'exécuter des commandes manuelles dans containers productifs.
- Gérer les secrets via mécanismes sécurisés (ne pas COPY de fichiers secrets dans l'image)