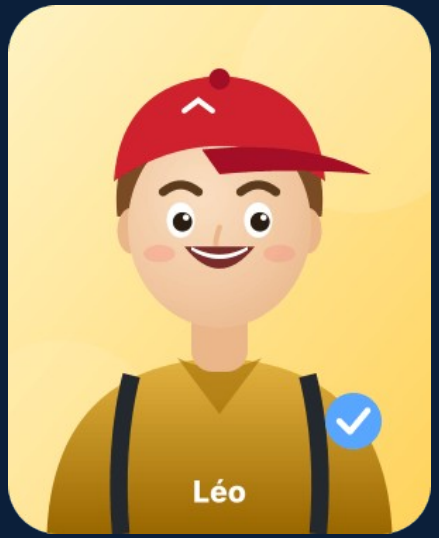
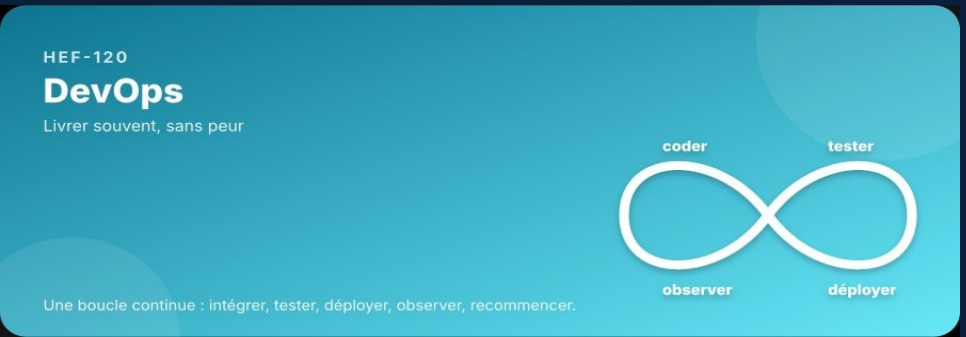
HEF-120 · MISSION 04 / 10

Conteneurs



Un conteneur emballe un programme avec tout ce dont il a besoin pour tourner : même boîte sur ton ordinateur, en test et en production.

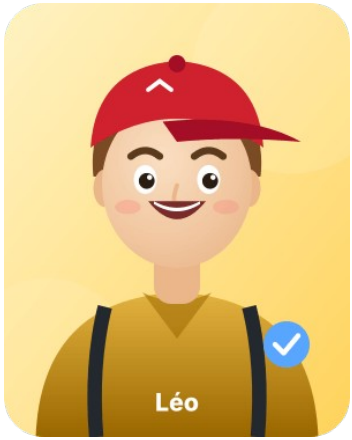


Badge : Capitaine de conteneurs

100 min · intermédiaire

L'histoire

« Ça marche chez moi. » Kodi l'a dit trop souvent : sa machine a Python 3.12 et une bibliothèque système particulière ; le serveur de test a une autre version. HashBox décide d'**empaqueter** l'application avec son environnement exact, dans une boîte standard que n'importe quel serveur sait exécuter.



> *« Un conteneur n'est pas une machine virtuelle, précise Sam. C'est un processus ordinaire, isolé par le noyau, avec son propre système de fichiers. Léger, rapide à démarrer, identique partout. Et on le décrit par un fichier texte : le Dockerfile. »*

Tu apprends ce qu'est vraiment un conteneur, comment une image est faite de couches, comment écrire un bon Dockerfile, et comment les conteneurs sont placés sur des machines.

Ta mission

À la fin, je serai capable de…

1. distinguer conteneur, image, machine virtuelle et registre
2. expliquer l'isolation par le noyau (espaces de noms, groupes de contrôle)
3. comprendre les couches d'image et le cache de build, et écrire un Dockerfile efficace
4. appliquer les bonnes pratiques (non root, image minimale, multi-étapes, épinglage)
5. expliquer le placement de conteneurs avec requêtes de ressources

Défi d'échauffement

Défi d'échauffement

Un conteneur de transport maritime a des dimensions standard : des grues, des camions et des bateaux de n'importe quel pays savent le manipuler sans savoir ce qu'il contient. Quel avantage pour un logiciel d'avoir une « boîte standard » ? Qu'est-ce qui reste de l'extérieur et qui n'est pas dans la boîte (le bateau, la grue) ?

Le concept

Conteneur, image, registre

Terme	Définition
Image	modèle **immuable** : système de fichiers en couches + métadonnées (commande de démarrage…)
Conteneur	**instance en cours d'exécution** d'une image : un processus isolé
Registre	dépôt d'images (Docker Hub, GHCR, registre privé)
Dockerfile	fichier texte décrivant comment construire une image
Moteur / runtime	programme qui crée et exécute les conteneurs (Docker, containerd, Podman)

Analogie : **image = classe**, **conteneur = objet**.

Le concept

Conteneur ≠ machine virtuelle

	Machine virtuelle	Conteneur
Isolation	un système d'exploitation complet par VM (hyperviseur)	**processus** isolés qui partagent le **noyau** de l'hôte
Taille	gigaoctets	mégaoctets
Démarrage	minutes	millisecondes à secondes
Isolation de sécurité	plus forte	plus faible (noyau partagé) : bonnes pratiques indispensables

Comment le noyau Linux isole

- **Espaces de noms** (*namespaces*) : chaque conteneur voit ses propres processus (PID), réseau, systèmes de fichiers montés, noms d'hôte, utilisateurs.

Le concept

Comment le noyau Linux isole (suite)

- **Groupes de contrôle** (*cgroups*) : **limitent** les ressources (CPU, mémoire, E/S).
- **Capacités et filtres d'appels système** (seccomp) : réduisent ce que le processus peut faire.

Un conteneur est donc un **processus normal** dans un cadre de contraintes (cf. HEF-050).

Les images en couches

Chaque instruction du Dockerfile crée une **couche** immuable. Les couches sont **partagées** entre images et **mises en cache**.

```
FROM python:3.12-slim          # couche de base
COPY requirements.txt .        # couche
RUN pip install -r requirements.txt   # couche (lente)
COPY . .                       # couche (change souvent)
CMD ["python", "app.py"]       # métadonnée
```

Le concept

Les images en couches (suite)

Règle d'or du cache : si une couche change, **toutes les suivantes** sont reconstruites. Donc on place **d'abord** ce qui change rarement (dépendances) et **à la fin** ce qui change souvent (code). Copier `requirements.txt` **avant** `COPY . .` évite de réinstaller toutes les dépendances à chaque modification du code.

Un bon Dockerfile : les bonnes pratiques

Pratique	Raison
Image de base minimale (`-slim`, distroless)	moins de taille, moins de failles
Version épinglée (`python:3.12.4-slim`, voire un haché)	reproductibilité (mission 3) ; jamais `latest`
Construction multi-étapes	l'outillage de build reste hors de l'image finale
Utilisateur non root (`USER app`)	limite les dégâts en cas de compromission (HEF-100)
`.dockerignore`	n'envoie pas `.git`, secrets, caches dans le contexte

Le concept

Un bon Dockerfile : les bonnes pratiques (suite)

Pratique	Raison
Un processus par conteneur	supervision et mise à l'échelle simples
Pas de secrets dans l'image	ils persistent dans les couches
`HEALTHCHECK`	l'orchestrateur sait si le service est sain
Journaux sur la sortie standard	collectés par la plateforme

Multi-étapes : exemple

```
FROM python:3.12-slim AS build
WORKDIR /src
COPY requirements.txt .
RUN pip wheel --wheel-dir /wheels -r requirements.txt

FROM python:3.12-slim
```

Le concept

Multi-étapes : exemple (suite)

```
RUN useradd -r app
COPY --from=build /wheels /wheels
RUN pip install --no-index --find-links=/wheels /wheels/*.whl && rm -rf /wheels
COPY app/ /app/
USER app
HEALTHCHECK CMD python -c "import urllib.request; urllib.request.urlopen('http://l
   ocalhost:8000/sante')"
CMD ["python", "/app/main.py"]
```

Orchestration

Pour exécuter beaucoup de conteneurs sur plusieurs machines, un **orchestrateur** (Kubernetes) :

- **place** les conteneurs selon leurs **requêtes de ressources** (CPU, mémoire) ;
- les **redémarre** s'ils tombent ;
- les **met à l'échelle** ;

Le concept

Orchestration (suite)

- gère le **réseau**, les **volumes**, la **configuration**, les **déploiements progressifs**.

Concepts : *pod* (un ou plusieurs conteneurs groupés), *deployment* (état voulu), *service* (adresse stable), *probe* de santé.

🧠 ***Astuce** : un conteneur est **jetable** : s'il se comporte mal, on le détruit et on en recrée un. Ne stocke rien d'important dedans — utilise des volumes ou des services externes.*

Exemple pas à pas

Problème : sans moteur Docker, **simuler** la construction en couches et le cache pour voir l'effet de l'ordre des instructions, puis auditer un Dockerfile.

```
import hashlib, re

def h(*morceaux):
    m = hashlib.sha256()
    for x in morceaux:
        m.update(str(x).encode())
    return m.hexdigest()[:10]

TEMPS = {"FROM": 1, "RUN pip": 60, "RUN": 5, "COPY": 1, "CMD": 0}        # seconde
   s simulées

def temps(instruction):
    if instruction.startswith("RUN pip"):
        return TEMPS["RUN pip"]
    return TEMPS.get(instruction.split()[0], 1)
```

Exemple pas à pas

Exemple pas à pas (suite)

```
def construire(dockerfile, fichiers, cache):
    # fichiers : nom -> contenu ; une couche COPY dépend du CONTENU des fichiers c
   opiés
    parent, journal, total = "", [], 0
    for ligne in dockerfile:
        if ligne.startswith("COPY"):
            _, source, _ = ligne.split()
            contenu = [fichiers[f] for f in sorted(fichiers) if source == "." or f
    == source]
            cle = h(parent, ligne, *contenu)
        else:
            cle = h(parent, ligne)
        if cle in cache:
            journal.append((ligne, "cache"))
        else:
            cache.add(cle); total += temps(ligne); journal.append((ligne, "constru
   it"))
        parent = cle
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    return total, journal

MAUVAIS = ["FROM python:3.12-slim", "COPY . /app", "RUN pip install -r /app/requir
   ements.txt", "CMD python /app/main.py"]
BON = ["FROM python:3.12-slim", "COPY requirements.txt /app/", "RUN pip install -r
    /app/requirements.txt", "COPY . /app", "CMD python /app/main.py"]

def scenario(dockerfile):
    cache, fichiers, durees = set(), {"requirements.txt": "flask==3.0", "main.py":
    "v1"}, []
    durees.append(construire(dockerfile, fichiers, cache)[0])          # 1re cons
   truction : tout
    fichiers["main.py"] = "v2"; durees.append(construire(dockerfile, fichiers, cac
   he)[0])      # on modifie le code
    fichiers["main.py"] = "v3"; durees.append(construire(dockerfile, fichiers, cac
   he)[0])      # encore
    fichiers["requirements.txt"] = "flask==3.1"; durees.append(construire(dockerfi
   le, fichiers, cache)[0])   # on change une dépendance
    return durees
```

Exemple pas à pas (suite)

```
d_mauvais, d_bon = scenario(MAUVAIS), scenario(BON)
print("Dockerfile mal ordonné :", d_mauvais, "s  → total", sum(d_mauvais))
print("Dockerfile bien ordonné:", d_bon, "s  → total", sum(d_bon))
assert d_mauvais == [62, 61, 61, 61] and d_bon == [63, 1, 1, 62]
assert sum(d_bon) < sum(d_mauvais) / 1.5

# ---------------- audit de Dockerfile ----------------
def auditer(dockerfile):
    problemes, a_user = [], False
    for n, ligne in enumerate(dockerfile.strip().splitlines(), 1):
        l = ligne.strip()
        if re.match(r"FROM\s+\S+(:latest)?\s*(AS\s+\w+)?$", l, re.I) and (":latest
   " in l or ":" not in l.split()[1]):
            problemes.append((n, "image non épinglée (latest ou sans étiquette)"))
        if l.upper().startswith("USER ") and l.split()[1] not in ("root", "0"):
            a_user = True
        if re.match(r"(ENV|ARG)\s+\w*(PASSWORD|SECRET|TOKEN|KEY)\w*\s*=", l, re.I)
   :
```

Exemple pas à pas (suite)

```
            problemes.append((n, "secret dans l'image"))
        if re.match(r"RUN\s+.*(curl|wget).*\|\s*(sh|bash)", l):
            problemes.append((n, "exécution d'un script téléchargé sans vérificati
   on"))
        if l.upper().startswith("ADD ") and re.search(r"https?://", l):
            problemes.append((n, "ADD depuis une URL"))
    if not a_user:
        problemes.append((0, "pas d'utilisateur non root"))
    if "HEALTHCHECK" not in dockerfile.upper():
        problemes.append((0, "pas de HEALTHCHECK"))
    return problemes

for numero, probleme in auditer("FROM python:latest\nENV API_TOKEN=abc123\nRUN cur
   l http://x.sh | sh\nCOPY . /app\nCMD python /app/main.py"):
    print(f"  ligne {numero} : {probleme}")
assert auditer("FROM python:3.12.4-slim\nUSER app\nHEALTHCHECK CMD true\nCMD pytho
   n x") == []
```

Exemple pas à pas (suite)

Lecture :

- Dans le Dockerfile **mal ordonné**, `COPY . /app` vient avant l'installation des dépendances : chaque modification du code invalide la couche suivante et relance `pip install` (60 s). Dans le **bien ordonné**, modifier le code ne reconstruit que la dernière couche (~1 s) ; seul le changement de `requirements.txt` relance l'installation.
- Sur 4 constructions, le gain est de l'ordre d'un facteur deux ou plus, et il grandit avec le nombre de modifications de code.
- L'**audit** repère les défauts classiques (`latest`, secret dans `ENV`, script téléchargé exécuté, absence d'utilisateur non root et de `HEALTHCHECK`).

Pour construire réellement : `docker build -t hashbox:1.0 .` puis `docker run --rm -p 8000:8000 hashbox:1.0`.

À toi, avec aide

Écris `suggerer_ordre(dockerfile)` qui détecte le défaut le plus courant : un `COPY . …` placé **avant** une instruction `RUN` d'installation de dépendances (`pip install`, `npm install`, `apt-get install`) alors qu'un fichier de dépendances existe. Elle renvoie le Dockerfile corrigé : insère `COPY <fichier de dépendances> …` avant l'installation, puis `COPY . …` après. Teste avec `requirements.txt`.

Indices progressifs

Indice 1

Cherche l'indice du premier `COPY .` et celui de `RUN pip install -r <fichier>`.

Indice 2

Extrais le nom du fichier de dépendances de la commande `pip install -r fichier`.

Indice 3

Insère les lignes dans le bon ordre.

À toi, avec aide · réponse

Réponse

```
import re

def suggerer_ordre(lignes):
    i_copy = next((i for i, l in enumerate(lignes) if re.match(r"COPY\s+\.\s", l))
   , None)
    for i_run, l in enumerate(lignes):
        m = re.match(r"RUN\s+pip install .*-r\s+(\S+)", l)
        if m and i_copy is not None and i_copy < i_run:
            fichier = m.group(1).split("/")[-1]
            destination = re.match(r"COPY\s+\.\s+(\S+)", lignes[i_copy]).group(1)
            resultat = lignes[:i_copy]
            resultat += [f"COPY {fichier} {destination.rstrip('/')}/", l.replace(f
   "/app/{fichier}", f"{destination.rstrip('/')}/{fichier}")]
            resultat += [lignes[i_copy]] + [x for k, x in enumerate(lignes) if k >
    i_copy and k != i_run]
            return resultat
    return lignes

avant = ["FROM python:3.12-slim", "COPY . /app", "RUN pip install -r /app/requirem
```

À toi, avec aide · réponse

Réponse (suite)

```
   ents.txt", "CMD python /app/main.py"]
apres = suggerer_ordre(avant)
assert apres == ["FROM python:3.12-slim", "COPY requirements.txt /app/", "RUN pip
   install -r /app/requirements.txt", "COPY . /app", "CMD python /app/main.py"]
assert suggerer_ordre(apres) == apres                          # déjà bon : inchang
   é
print("\n".join(apres))
```

Un outil de lint de Dockerfile (hadolint) détecte ce genre de défaut automatiquement dans la CI.

À toi, en solo

Placement de conteneurs. Écris un ordonnanceur simplifié :

1. des **nœuds** avec CPU (en cœurs) et mémoire (en Go) disponibles ;
2. des **conteneurs** avec une **requête** (CPU, mémoire) ;
3. algorithme *first-fit decreasing* : trier les conteneurs par taille décroissante (mémoire puis CPU) et les placer sur le premier nœud où ils tiennent ;
4. détecte ceux qui ne tiennent nulle part (« en attente ») ;
5. calcule le taux d'utilisation de chaque nœud ;
6. simule la **panne d'un nœud** : replace ses conteneurs sur les autres ; indique ceux qui ne trouvent plus de place.

À toi, en solo · réponse

Pistes de correction

```
def placer(noeuds, conteneurs):
    libres = {n: dict(r) for n, r in noeuds.items()}
    placement, attente = {}, []
    for nom, req in sorted(conteneurs.items(), key=lambda kv: (-kv[1]["mem"], -kv[
   1]["cpu"], kv[0])):
        for n in sorted(libres):
            if libres[n]["cpu"] >= req["cpu"] and libres[n]["mem"] >= req["mem"]:
                libres[n]["cpu"] -= req["cpu"]; libres[n]["mem"] -= req["mem"]
                placement[nom] = n
                break
        else:
            attente.append(nom)
    return placement, attente, libres

NOEUDS = {"n1": {"cpu": 4, "mem": 8}, "n2": {"cpu": 4, "mem": 8}, "n3": {"cpu": 2,
    "mem": 4}}
CONTENEURS = {"api": {"cpu": 2, "mem": 3}, "db": {"cpu": 2, "mem": 6}, "cache": {"
   cpu": 1, "mem": 4}, "worker1": {"cpu": 1, "mem": 2},
              "worker2": {"cpu": 1, "mem": 2}, "front": {"cpu": 1, "mem": 1}, "bat
```

À toi, en solo · réponse

Pistes de correction (suite)

```
   ch": {"cpu": 4, "mem": 7}}
placement, attente, libres = placer(NOEUDS, CONTENEURS)
for n in sorted(NOEUDS):
    util = 1 - libres[n]["mem"] / NOEUDS[n]["mem"]
    print(n, "→", sorted(c for c, x in placement.items() if x == n), f"mémoire uti
   lisée {util:.0%}")
print("en attente :", attente)
assert set(placement) | set(attente) == set(CONTENEURS)

# panne du nœud n2 : on replace ses conteneurs sur les nœuds restants
survivants = {n: r for n, r in NOEUDS.items() if n != "n2"}
a_replacer = {c: CONTENEURS[c] for c, n in placement.items() if n == "n2"}
deja = {c: n for c, n in placement.items() if n != "n2"}
libres2 = {n: dict(r) for n, r in survivants.items()}
for c, n in deja.items():
    libres2[n]["cpu"] -= CONTENEURS[c]["cpu"]; libres2[n]["mem"] -= CONTENEURS[c][
   "mem"]
sans_place = []
for c, req in a_replacer.items():
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    for n in sorted(libres2):
        if libres2[n]["cpu"] >= req["cpu"] and libres2[n]["mem"] >= req["mem"]:
            libres2[n]["cpu"] -= req["cpu"]; libres2[n]["mem"] -= req["mem"]; brea
  k
    else:
        sans_place.append(c)
print("après panne de n2, sans place :", sans_place)
```

La capacité doit prévoir la **perte d'un nœud** (N+1) : sinon une panne matérielle devient une panne de service. Kubernetes fait exactement ce calcul en continu.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Utiliser `latest`	Image différente à chaque construction	Version (ou haché) épinglé
Exécuter en root	Compromission = pouvoirs étendus	USER non privilégié
Secrets dans l'image	Ils restent dans les couches	Injection à l'exécution (mission 5)
Stocker des données dans le conteneur	Perdues à chaque redémarrage	Volumes ou services externes

Mini-boss

Mini-boss

Dossier conteneur de HashBox. Pour une application web de trois services (api, worker, base) :

1. écris les trois Dockerfiles (multi-étapes, non root, épinglés, `HEALTHCHECK`) ;
2. écris un `.dockerignore` ;
3. écris un descriptif « composition » (dictionnaire) : services, dépendances, ports, variables, ressources, volumes ;
4. écris un valideur : détecte les cycles de dépendances, les ports en conflit, les services sans `healthcheck`, les images non épinglées ; calcule l'**ordre de démarrage** (tri topologique) ;
5. simule la construction avec cache sur 6 commits de code et mesure le temps avec et sans bon ordre des couches ;
6. place les conteneurs sur 3 nœuds et simule une panne.

Mini-boss · réponse

Pistes de correction

Un bon travail : Dockerfiles avec une étape `build` et une étape finale, `USER app`, `HEALTHCHECK`, versions épinglées ; un valideur avec tri topologique (`graphlib.TopologicalSorter`) qui lève `CycleError` ; contrôle des ports hôtes ; simulation de cache réutilisant la fonction `construire` de l'exemple ; ordonnanceur FFD avec test de panne. L'ordre de démarrage doit placer la base avant l'API et l'API avant le worker si le worker en dépend.

Quiz éclair

Quiz éclair

1. Un conteneur est :

A. un processus isolé par le noyau, instance d'une image
B. une machine virtuelle complète
C. un fichier de configuration

2. Pour tirer parti du cache de build, on place d'abord :

A. ce qui change souvent (code)
B. ce qui change rarement (dépendances)
C. les secrets

3. Une bonne pratique de sécurité d'un Dockerfile est :

A. utiliser `latest`

Quiz éclair

Quiz éclair (suite)

B. copier les secrets dans l'image
C. exécuter le processus avec un utilisateur non root

4. 🔁 *Révision mission 03* : Un build reproductible produit :

A. le même artefact à partir des mêmes sources et configuration
B. un artefact différent à chaque fois
C. uniquement des logs

5. 🔁 *Révision mission 03* : Un fichier de verrouillage avec empreintes sert à :

A. augmenter la vitesse du build
B. figer les versions exactes et refuser un paquet altéré
C. remplacer les tests

Quiz éclair · réponse

Réponses

1. **A**.
2. **B**.
3. **C**.
4. **A**.
5. **B**.

Je réfléchis

1. Quels éléments de ton environnement de développement ne sont pas décrits dans un fichier versionné ?
2. Que se passerait-il si le conteneur de ton service était détruit et recréé à l'instant ?

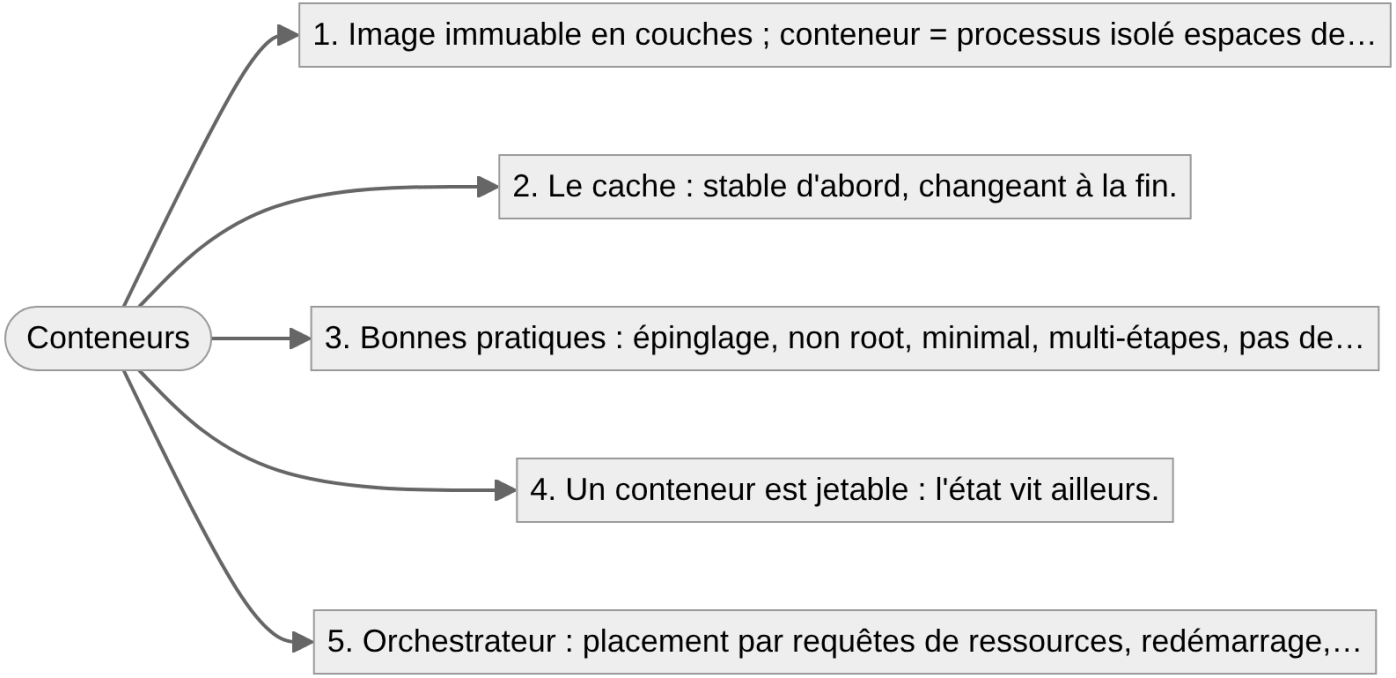
À retenir

À retenir

1. Image immuable en couches ; conteneur = processus isolé (espaces de noms, cgroups).
2. Le cache : stable d'abord, changeant à la fin.
3. Bonnes pratiques : épinglage, non root, minimal, multi-étapes, pas de secret, healthcheck.
4. Un conteneur est jetable : l'état vit ailleurs.
5. Orchestrateur : placement par requêtes de ressources, redémarrage, mise à l'échelle.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Capitaine de conteneurs

Prochaine mission : Configuration et secrets

