Orchestration

Principes de base de l'Orchestration de conteneurs, application avec Docker Compose.

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

2026-11-01

Table des matières

1 Orchestration de Conteneurs 1
2 Exemples de base 2
3 Anatomie d'un fichier Docker Compose 2
3.1 Structure Générale de Docker Compose 3
3.2 Exemple Pratique 3
4 Construction d'images à la volée 6
5 Gestion des dépendances entre services 7
5.1 Un exemple avancé 7
5.2 Best practices 7
6 Passer à l'échelle : du Scale-up au Scale-out 8
6.1 Rôle d'un orchestrateur de conteneurs 9
6.2 Docker Swarm et Kubernetes 9
6.3 Pourquoi utiliser un orchestrateur en production ? 9

1 Orchestration de Conteneurs

L'orchestration consiste à coordonner plusieurs conteneurs au sein d'une même application. Si kubernetes de Google est l'outil de référence pour les déploiements complexes, docker compose offre une alternative plus simple pour les architectures modestes.

Docker compose, disponible comme plugin de Docker, permet de décrire une architecture multi-conteneurs dans un fichier YAML (`docker-compose.yml`). Il gère les services, volumes et réseaux avec la même syntaxe que Docker, simplifiant ainsi le déploiement d'applications conteneurisées.

- L'orchestration de conteneurs coordonne plusieurs conteneurs pour une application.
- Kubernetes est l'outil de référence pour les déploiements complexes.
- Docker Compose est une alternative plus simple pour les architectures modestes.
- Il utilise un fichier YAML (`compose.yml`) pour définir les services, réseaux et volumes.
- Parfait pour le développement local et les déploiements modestes, mais limité pour les environnements de production complexes.
- Permet de lancer, arrêter et gérer plusieurs conteneurs avec une seule commande.

2 Exemples de base

Les exemples suivants sont accessibles dans le dépôt :

: ebpro/sample-containers-java-restjpa

Détails

- **URL**: https://github.com/ebpro/sample-containers-java-restjpa
- **Branch**: develop
- **Local Path**: `/home/jovyan/work/github/ebpro/sample-containers-java-restjpa`
- **Latest Commit**: c195ffd - chore: add _quarto-utils submodule + .devcontainer (SOTA pattern)
- **Timestamp**: 2026-11-01T12:06:05Z

Clone Command:

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

Projet de demo : /home/jovyan/work/github/ebpro/sample-containers-java-restjpa

3 Anatomie d'un fichier Docker Compose

- Compose permet de définir et gérer des applications multi-conteneurs via un fichier YAML : `docker-compose.yml` ou `compose.yml`.
- Ce fichier offre une syntaxe déclarative pour configurer les services, réseaux et volumes correspondant aux commandes Docker habituelles.
- Un fichier `compose.yml` (ou `docker-compose.yml`) s'articule autour de quatre piliers principaux.
  - **`services`** : C'est le cœur du fichier. Chaque service définit comment instancier un conteneur (image, build, variables d'environnement, dépendances).
  - **`networks`** : Définit les ponts de communication. Par défaut, Compose crée un réseau pour le projet, mais vous pouvez isoler (ex: `frontend` vs `backend`).
  - **`volumes`** : Déclare les espaces de stockage qui survivent à la suppression des conteneurs (`docker compose down`).

- `configs` / `secrets` : Permet de monter des fichiers de configuration ou des données sensibles de manière sécurisée (souvent utilisé avec Swarm ou les versions récentes de Compose).

3.1 Structure Générale de Docker Compose

```
services:      # 1. Les conteneurs à lancer
  web:
    build: .
    ports: ["80:80"]
    networks: [frontend]
    volumes: [data:/var/www/html]

networks:      # 2. Les réseaux (isolations logiques)
  frontend:

volumes:       # 3. Le stockage persistant
  data:

configs/secrets: # 4. Données de config et mots de passe
```

Un service dans Docker Compose correspond à un conteneur Docker. Chaque service peut spécifier une image Docker à utiliser ou un contexte de construction pour créer l'image. Les services peuvent également définir des variables d'environnement, des ports exposés, des volumes montés, et des dépendances entre eux.

3.2 Exemple Pratique

Pour démarrer une application multi-conteneurs, utilisez la commande `docker compose up` dans le répertoire contenant le fichier `docker-compose.yml`. L'option `-d` (detach) permet d'exécuter les conteneurs en arrière-plan :

```
if [ -f compose.yml ]; then
  docker compose up --detach
else
  echo "(skipped: no compose.yml in $(pwd))"
fi
```

La commande `docker compose ls` fournit une vue d'ensemble de tous les projets Docker Compose en cours d'exécution, affichant pour chacun son nom, le répertoire source, l'état des services et le nombre de conteneurs actifs.

```
docker compose ls
```

```
NAME                                STATUS              CONFIG FILES
sample-containers-java-restjpa      running(2)          /home/jovyan/work/github/
ebpro/sample-containers-java-restjpa/compose.yml
```

La commande `docker compose ps` affiche les conteneurs du projet en cours, avec leurs noms générés automatiquement selon le format `<projet>-<service>-<numéro>`. Cette convention de nommage permet d'exécuter plusieurs instances du même projet dans différents répertoires ou de dupliquer des services sans conflits, à condition d'éviter les liaisons de volumes (`bind mounts`) et les mappages de ports fixes vers l'hôte.

```
if [ -f compose.yml ] || [ -f compose.prod.yml ]; then
  docker compose ps
else
  echo "(skipped: no compose project in $(pwd))"
fi
```

```
NAME                                    IMAGE                               COMMAND
SERVICE   CREATED          STATUS                          PORTS
sample-containers-java-restjpa-app-1   docker.io/brunoe/restjpa:0.1.0   "java -
jar /app/app.…"    app        13 seconds ago    Up 1 second (health: starting)
0.0.0.0:8088→8080/tcp, [::]:8088→8080/tcp
sample-containers-java-restjpa-db-1    postgres:15.2-alpine            "docker-
entrypoint.s…"   db      13 seconds ago   Up 12 seconds (healthy)        5432/tcp
```

Docker Compose gère automatiquement les réseaux et volumes en les préfixant avec le nom du projet (par défaut, le nom du répertoire parent), permettant ainsi d'isoler les ressources entre différents projets et d'éviter les conflits de nommage, tout en facilitant leur identification et leur gestion avec des commandes comme `docker compose down -v` pour le nettoyage complet.

```
docker network ls
```

```
NETWORK ID     NAME                DRIVER    SCOPE
b8cb4053ad74   app-net             bridge    local
703b7cb7070f   bridge              bridge    local
e96917651d22   host                host      local
bba9adcbec6a   none                null      local
879ebccc2b3e   restjpa-backend     bridge    local
c4896216f86a   restjpa-frontend    bridge    local
```

```
docker volume ls
```

```
DRIVER    VOLUME NAME
local     20e3a163dc0997c6df71cb5546eaf55b0063462a62130f831abebe412f311da0
local     restjpa-pg-data
local     tp0-www
```

La commande `docker compose logs` permet de visualiser les journaux d'un ou plusieurs services, avec des options utiles comme `-f` pour suivre les logs en temps réel, `--tail` pour limiter le nombre de lignes affichées, et `--since` pour filtrer par date, facilitant ainsi le diagnostic et la surveillance des applications multi-conteneurs.

```
if [ -f compose.yml ]; then
  docker compose logs --tail 5
else
  echo "(skipped: no compose.yml in $(pwd))"
fi
```

```
app-1   | Picked up JAVA_TOOL_OPTIONS: -XX:+UseZGC -XX:+ZGenerational -Xmx512m
-Xms512m
app-1  | 2026-11-01 12:07:54.760 [main] INFO  com.zaxxer.hikari.HikariDataSource
- HikariPool-1 - Starting...
db-1  | 2026-11-01 12:07:44.974 UTC [1] LOG:  listening on IPv4 address "0.0.0.0",
port 5432
db-1   | 2026-11-01 12:07:44.974 UTC [1] LOG:  listening on IPv6 address "::",
port 5432
db-1   | 2026-11-01 12:07:45.001 UTC [1] LOG:  listening on Unix socket "/var/
run/postgresql/.s.PGSQL.5432"
db-1   | 2026-11-01 12:07:45.033 UTC [52] LOG:  database system was shut down at
2026-11-01 12:07:44 UTC
db-1    | 2026-11-01 12:07:45.049 UTC [1] LOG:  database system is ready to
accept connections
```

Docker Compose offre plusieurs commandes pour gérer le cycle de vie des services : `stop` pour arrêter les conteneurs, `rm` pour les supprimer, `restart` pour les redémarrer, et `down` pour tout nettoyer (conteneurs et réseaux) - avec l'option `-v` pour inclure également la suppression des volumes, qu'ils soient nommés ou anonymes.

> ⚠️ Attention
>
> La commande `docker compose down -v` supprime définitivement toutes les données persistées dans les volumes.

```
if [ -f compose.yml ]; then
  docker compose down -v
else
  echo "(skipped: no compose.yml in $(pwd))"
fi
```

4 Construction d'images à la volée

L'autre service `app` est une application JPA/REST Java dont l'image docker est produite par `sample-java/restjpa/Dockerfile`.

L'option `build` dans `docker-compose.yml` indique qu'il faut fabriquer l'image à partir du contexte courant (`image` sera alors son tag).

La sous-commande `build` de Docker Compose permet de construire les images à partir du contexte défini dans le fichier `docker-compose.yml`.

```
if [ -f compose.yml ]; then
  docker compose build
else
  echo "(skipped: no compose.yml in $(pwd))"
fi
```

Il est aussi une sous-commande `push` pour pousser les images vers un registre (Docker Hub, GitHub Container Registry, ...).

La fabrication de l'image sera automatique au démarrage au besoin avec `up` **uniquement** si l'image n'existe pas déjà localement ou si le Dockerfile a été modifié depuis la dernière construction. Pour forcer la reconstruction de l'image à chaque démarrage, utilisez l'option `--build` avec `docker compose up`.

```
if [ -f compose.yml ]; then
  docker compose up --detach
```

```
else
  echo "(skipped: no compose.yml in $(pwd))"
fi
```

5 Gestion des dépendances entre services

La directive `depends_on` dans Docker Compose permet de gérer les dépendances entre services, mais va au-delà du simple ordre de démarrage grâce à la condition `service_healthy` qui, combinée avec des `HEALTHCHECK` (cf. Dockerfile), assure qu'un service est réellement opérationnel avant le démarrage des services qui en dépendent - par exemple, vérifier qu'une base de données accepte effectivement les connexions avant de lancer l'application qui l'utilise.

5.1 Un exemple avancé

[1;33m[WARNING][0m Git operation failed with status code: 128 Les exemples suivants sont accessibles dans le dépôt :

✕ : ebpro/notebook-java-rest-sample-quarkus

Détails

- **URL**: https://github.com/ebpro/notebook-java-rest-sample-quarkus
- **Branch**: develop
- **Local Path**: `/home/jovyan/work/github/ebpro/notebook-java-rest-sample-quarkus`
- **Latest Commit**: No commit information available
- **Timestamp**: 2026-11-01T12:10:49Z

Clone Command:

```
git clone -b develop https://github.com/ebpro/notebook-java-rest-sample-quarkus
```

(gitpull: clone indisponible - la demo prod sera sautee) (SRC_DIR indisponible : la demo prod sera sautee)

L'exemple ci-dessous présente un exemple avancé en ajoutant un reverse proxy (https://traefik.io) et des outils de monitoring pour gérer les points d'entrées de l'application (sécurité, répartition de charges, ...).

Commencez par le `compose.yml` qui est plus simple :

```
```

5.2 Best practices

LISEZ LE README (le début sur les conteneurs) très attentivement pour comprendre la configuration pour découvrir les meilleures pratiques utilisées dans cet exemple :

```
if [ -f compose.prod.yml ]; then
  docker compose --file compose.prod.yml up --detach
else
  echo "(skipped: no compose.prod.yml in $(pwd))"
fi
```

```
(skipped: no compose.prod.yml in /home/jovyan/work/github/ebpro/sample-containers-java-restjpa)
```

```
if [ -f compose.yml ] || [ -f compose.prod.yml ]; then
  docker compose ps
else
  echo "(skipped: no compose project in $(pwd))"
fi
```

```
NAME                                    IMAGE                              COMMAND
SERVICE   CREATED         STATUS                    PORTS
sample-containers-java-restjpa-app-1    docker.io/brunoe/restjpa:0.1.0     "java
-jar /app/app.…"   app      3 minutes ago   Up 2 minutes (healthy)   0.0.0.0:8088-
>8080/tcp, [::]:8088→8080/tcp
sample-containers-java-restjpa-db-1     postgres:15.2-alpine               "docker-
entrypoint.s…"   db        3 minutes ago   Up 2 minutes (healthy)   5432/tcp
```

```
if [ -f compose.prod.yml ]; then
  docker compose --file compose.prod.yml down -v
else
  echo "(skipped: no compose.prod.yml in $(pwd))"
fi
```

```
(skipped: no compose.prod.yml in /home/jovyan/work/github/ebpro/sample-containers-java-restjpa)
```

6 Passer à l'échelle : du Scale-up au Scale-out

Lorsque l'application grandit, **un seul serveur devient vite une limite** :

- saturation CPU / RAM
- point unique de panne
- impossibilité de gérer la charge variable
- On passe alors d'un outil pensé pour le **développement local**

- à des outils conçus pour **exploiter plusieurs machines**

6.1 Rôle d'un orchestrateur de conteneurs

Un orchestrateur permet de :

- Distribuer automatiquement les conteneurs sur plusieurs serveurs
- Surveiller l'état des applications
- Redémarrer les services en cas de panne
- Déployer de nouvelles versions sans interruption
- Adapter dynamiquement la capacité selon la charge

6.2 Docker Swarm et Kubernetes

Deux approches outils :

- Docker Swarm : solution d'orchestration native de Docker, simple à configurer et à utiliser.
- Kubernetes : plateforme d'orchestration complète, plus complexe mais extrêmement puissante.

Critère	**Docker Swarm**	**Kubernetes (K8s)**
Philosophie	Extension naturelle de Docker	Plateforme complète d'orchestration
Complexité	Simple et rapide à prendre en main	Plus complexe mais extrêmement puissant
Unité de déploiement	Service	Pod (groupe de conteneurs)
Installation	`docker swarm init`	Installation plus lourde en production ou Cloud managé
Cas d'usage typique	PME, projets simples	Standard industriel, Cloud natif, Multi-cloud
Écosystème	Limité	Très vaste et extensible

6.3 Pourquoi utiliser un orchestrateur en production ?

- Auto-healing : Le cluster détecte les pannes et redémarre automatiquement les conteneurs défaillants.
- Déploiement sans interruption : Les nouvelles versions sont déployées progressivement (rolling updates).
- Auto-scaling : Le nombre d'instances s'adapte automatiquement à la charge.
- Abstraction de l'infrastructure : On ne déploie plus sur un serveur précis, mais sur un **cluster entier**.