

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

2026-09-30

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

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.
 - ▶ Docker Compose est un outil d'orchestration léger intégré à Docker, idéal pour les applications multi-conteneurs simples.
 - ▶ 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/notebook-containers-intro-sample-java-restjpa](https://github.com/ebpro/notebook-containers-intro-sample-java-restjpa)

Détails

- **URL:** <https://github.com/ebpro/notebook-containers-intro-sample-java-restjpa>
- **Branch:** develop
- **Local Path:** `/home/jovyan/work/examples/github/ebpro/notebook-containers-intro-sample-java-restjpa`
- **Latest Commit:** 6cf36b4 - ci: use shared reusable SonarQube workflow (ebpro/ci-java v1.0.0)
- **Timestamp:** 2026-09-30T07:10:14Z

Clone Command:

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

```
docker compose up --detach
```

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
notebook-containers-intro-sample-java-restjpa	running(2)	/home/jovyan/work/examples/github/ebpro/notebook-containers-intro-sample-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.

```
docker compose ps
```

NAME	IMAGE	COMMAND	SERVICE	CREATED	STATUS	PORTS
notebook-containers-intro-sample-java-restjpa-app-1	docker.io/brunoe/restjpa:0.1.0	"java -jar /app/app..."	app	12 seconds ago	Up 1 second (health: starting)	0.0.0.0:8088->8080/tcp, [::]:8088->8080/tcp
notebook-containers-intro-sample-java-restjpa-db-1	postgres:15.2-alpine	"docker-entrypoint.s..."	db	13 seconds ago	Up 11 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		
7c3769659ab9	app-net	bridge
local		
8fef64541bc6	bridge	bridge
local		
cc35b78d4d3c	host	host
local		
aee8e3a815ab	none	null

```
local
430cbcl3cae9          restjpa-backend
bridge    local
76ded3c0f41f          restjpa-frontend
bridge    local
```

```
docker volume ls
```

```
DRIVER    VOLUME NAME
local     8d25f3749a19915301a5ad52c4b4c504829c5
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.

```
docker compose logs --tail 5
```

```
db-1 | 2026-09-30 07:11:24.266 UTC
[1] LOG:  listening on IPv4 address
"0.0.0.0", port 5432
db-1 | 2026-09-30 07:11:24.266 UTC
[1] LOG:  listening on IPv6 address
"::", port 5432
db-1 | 2026-09-30 07:11:24.286 UTC
[1] LOG:  listening on Unix socket "/
var/run/postgresql/.s.PGSQL.5432"
db-1 | 2026-09-30 07:11:24.309 UTC
[52] LOG:  database system was shut
down at 2026-09-30 07:11:24 UTC
db-1 | 2026-09-30 07:11:24.325 UTC
[1] LOG:  database system is ready to
accept connections
app-1 | Picked up JAVA_TOOL_OPTIONS:
```

```
-XX:+UseZGC -XX:+ZGenerational -
Xmx512m -Xms512m
app-1 | 2026-09-30
07:11:33.942 [main] INFO
com.zaxxer.hikari.HikariDataSource -
HikariPool-1 - Starting...
```

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.

```
docker compose down -v
```

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`.

```
docker compose build
```

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`.

```
docker compose up --detach
```

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

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/examples/github/ebpro/notebook-java-rest-sample-quarkus`
- **Latest Commit:** `069bbcb` - ci: use shared reusable SonarQube workflow (ebpro/ci-java v1.0.0)

- **Timestamp:** `2026-09-30T07:11:52Z`

Clone Command:

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

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 :

```
1  # =====
2  # Compose BASIQUE pour dev JPA/
3  #   PostgreSQL   +   Application
4  #   Quarkus
5  # =====
6  services:
7      # Base de données PostgreSQL
8      postgres:
9          # Image officielle PostgreSQL
10         image: postgres:16-alpine
11         # Les variables
12         container_name: postgres-db
13         environment:
14             POSTGRES_DB: ${DB_NAME:-
15             products}
16             POSTGRES_USER: ${DB_USER:-
17             tpuser}
18             POSTGRES_PASSWORD:
19             ${DB_PASSWORD:-Tp@2026}
20         ports:
21             # Exposer sur l'hôte pour
22             accès direct (DBever, pgAdmin, etc.)
```

```

19 -
20 "${DB_HOST_PORT:-15432}:5432"
21 volumes:
22     # Persistence des données
    dans un volume nommé
23     - postgres_data:/var/lib/
    postgresql/data
24 networks:
25     # Connexion au réseau commun
26     - product-network
27     # Vérification de l'état de
    santé de PostgreSQL
28 healthcheck:
29     test:
30         [
31             "CMD-SHELL",
32             "pg_isready -
33             U ${DB_USER:-tpuser} -d ${DB_NAME:-
34             products}",
35         ]
36     interval: 10s
37     timeout: 5s
38     retries: 5
39     start_period: 10s
40
41 # Application Quarkus JPA/REST
42 product-catalog:
43     # Image de l'application,
    configurable via des variables
    d'environnement
44     image: ${IMAGE_GROUP:-
    brunoe}/${IMAGE_NAME:-product-
    catalog}:${IMAGE_TAG:-1.0.0-jib}
45     # Dépendance sur PostgreSQL
    avec vérification de santé
46     depends_on:
47         postgres:
48             condition: service_healthy
49             environment:
50                 # Configuration base de
    données pour Quarkus
51                 DB_HOST: postgres

```

```

49     DB_PORT: 5432
50     DB_USER: ${DB_USER:-tpuser}
51     DB_PASSWORD:
52     ${DB_PASSWORD:-Tp@2026}
53     # Stratégie Hibernate
    (update pour dev, valide pour prod)
54     HIBERNATE_GENERATION:
55     ${HIBERNATE_GENERATION:-update}
56     # Logs
57     LOG_LEVEL: ${LOG_LEVEL:-
58     INFO}
59     LOG_SQL: ${LOG_SQL:-false}
60     # JVM Options
61     JAVA_OPTS: >-
62         -Xms256m
63         -Xmx512m
64         -XX:+UseContainerSupport
65         -XX:MaxRAMPercentage=75.0
66     ports:
67         # HTTP direct (pour
    developpement)
68 -
69     "${APP_HTTP_PORT:-8080}:8080"
70     networks:
71     - product-network
72     # Vérification de l'état de
    santé de l'application Quarkus
73     healthcheck:
74     test:
75         ["CMD-SHELL", "curl -
76         f http://localhost:8080/health/ready
77         || exit 1"]
78     interval: 30s
79     timeout: 5s
80     retries: 5
81     start_period: 30s
82
83 volumes:
84     postgres_data:

```

```

82     driver: local
83
84 networks:
85     product-network:
86     driver: bridge

```

```

>80/tcp,      0.0.0.0:5443->443/tcp,
[::]:5443->443/tcp

```

```

docker      compose      --file
compose.prod.yml down -v

```

5.2 Best practices

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

```

docker      compose      --file
compose.prod.yml up --detach

```

```

docker compose ps

```

NAME	IMAGE	COMMAND	SERVICE	CREATED	STATUS	PORTS
notebook-java-rest-sample-quarkus-postgres-1	postgres:16-alpine	"docker-entrypoint.s..."	postgres	33 seconds ago	Up 26 seconds (healthy)	5432/tcp
notebook-java-rest-sample-quarkus-product-catalog-1	brunoe/product-catalog:1.0.0-jib	"/opt/jboss/containe..."	product-catalog	27 seconds ago	Up 2 seconds (health: starting)	8080/tcp, 8443/tcp
notebook-java-rest-sample-quarkus-traefik-1	traefik:v3.0	"/entrypoint.sh --pi..."	traefik	33 seconds ago	Up 26 seconds (healthy)	0.0.0.0:8080->8080/tcp, [::]:8080->8080/tcp, 0.0.0.0:8888->80/tcp, [::]:8888-

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.

Cri- tère	Docker Swarm	Kubernetes (K8s)
Phi- loso- phie	Extension na- turelle de Do- cker	Plateforme complète d'orchestration
Com- plexi- té	Simple et rapide à prendre en main	Plus complexe mais extrême- ment puissant
Unité de dé- ploie- ment	Service	Pod (groupe de conteneurs)
Instal- lation	<code>docker swarm init</code>	Installation plus lourde en produc- tion ou Cloud ma- nagé
Cas d'usage typique	PME, projets simples	Standard indus- triel, Cloud natif, Multi-cloud
Éco- sys- tème	Limité	Très vaste et ex- tensible

6.3 Pourquoi utiliser un orchestrateur en production ?

- Auto-healing : Le cluster détecte les pannes et redémarre automatiquement les conteneurs défectueux.
- 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**.