

Contents

Étape 2 : environnement complet avec Docker Compose	1
Prérequis	1
Étape 2.0 : faire place nette et créer la branche (5 min)	2
Étape 2.1 : .env.example et .env (10 min)	2
Étape 2.2 : écrire compose.yaml (30 min)	3
2.2.a : le service db	3
2.2.b : le service api	4
2.2.c : le service front	5
Étape 2.3 : tout lancer en une commande (15 min)	5
Étape 2.4 : down, down -v et le volume (10 min)	6
Étape 2.5 : documenter le démarrage dans le README.md (10 min)	7
Étape 2.6 : commiter et ouvrir la PR (10 min)	7
Étape 2.7 : le test du nouvel arrivant, en binôme (15 min)	7
Étape 2.8 : la revue de code par le binôme, puis le merge (10 min)	8
Je valide avec le formateur	9
Questions de réflexion	9
Pour aller plus loin	9

Étape 2 : environnement complet avec Docker Compose

Quand : jour 3, matin | **Durée** : ~1h40 **Objectif** : remplacer les trois docker run d'hier par **un seul fichier**, compose.yaml, qui décrit la base, l'API et le front, avec leurs healthchecks, leur ordre de démarrage, un volume et un fichier .env. Un nouvel arrivant doit pouvoir lancer TaskFlow en **une commande**, ce que vous vérifierez en binôme.

Prérequis

- L'étape 1 est fusionnée dans main : api/Dockerfile, front/Dockerfile et front/nginx.conf existent.
- Slides du jour 3, blocs 3.5 et 3.6 ; antisèche [../ressources/cheatsheet-docker.md](#).
- Un binôme (pour l'étape 2.7).

Vocabulaire

Terme	Signification
Service	Un conteneur décrit dans compose.yaml (ici db, api, front) ; son nom sert aussi de nom sur le réseau
Healthcheck	Une commande que Docker lance régulièrement pour savoir si le conteneur est prêt (pas seulement démarré)
depends_on + condition: service_healthy	« Ne démarre ce service que lorsque l'autre est <i>healthy</i> »

Terme	Signification
Projet Compose	L'ensemble des conteneurs, du réseau et des volumes créés par un <code>compose.yaml</code> ; son nom est par défaut celui du dossier

Étape 2.0 : faire place nette et créer la branche (5 min)

Supprimez les conteneurs, le réseau et le volume de l'étape 1 (les ports 3000, 5432 et 8080 doivent être libres) :

```
docker rm -f front api db
docker network rm taskflow-net
docker volume rm taskflow-db-data
```

Puis :

```
cd ~/taskflow
git switch main
git pull
git switch -c feature/compose
```

Résultat attendu : `docker ps` ne montre plus `front`, `api` ni `db` ; `git branch` affiche `* feature/compose`.

Si ça ne marche pas - `No such container` ou `network not found` : déjà supprimé, continuez. - `volume is in use` : un conteneur utilise encore le volume ; `docker ps -a` pour le trouver, `docker rm -f <nom>`.

Pourquoi ? Compose va créer **son propre** réseau et **son propre** volume (préfixés par le nom du projet, `taskflow`). On repart de zéro pour ne pas confondre les deux.

Étape 2.1 : .env.example et .env (10 min)

Docker Compose lit **automatiquement** un fichier `.env` placé à côté de `compose.yaml` et remplace les `${VARIABLE}` du fichier par ses valeurs. On versionne un **exemple** (`.env.example`), jamais le vrai `.env`.

Créez le fichier `.env.example` à la racine du dépôt :

```
# Variables lues par compose.yaml (Docker Compose lit automatiquement le fichier .env).
# Pour démarrer : cp .env.example .env
# Le fichier .env est ignoré par Git : ne le committez jamais.

# Mot de passe de la base PostgreSQL de DÉVELOPPEMENT.
# Attention : il n'est lu qu'à la création du volume. Pour le changer ensuite : docker com
v
POSTGRES_PASSWORD=taskflow
```

Puis :

```
cp .env.example .env
git check-ignore -v .env
git status
```

Résultat attendu :

```
.gitignore:10:.env .env
```

et `git status` liste `.env.example`, mais **pas** `.env`.

Si ça ne marche pas - `git check-ignore` n'affiche rien : `.env` n'est pas ignoré. Vérifiez que le `.gitignore` du starter est bien à la racine du dépôt (`ls -a`) et contient la ligne `.env`.

Pourquoi ? Le `.env.example` documente **quelles** variables existent ; le `.env` contient **les valeurs** de votre poste. Le jour 4, le `.env` de production contiendra de vrais secrets : le réflexe doit être pris dès maintenant.

Étape 2.2 : écrire `compose.yaml` (30 min)

On construit le fichier service par service, en testant à chaque fois.

2.2.a : le service db

Créez le fichier `compose.yaml` à la racine du dépôt :

```
# compose.yaml : environnement de DÉVELOPPEMENT complet en une commande
# docker compose up -d --build      # démarre db + api + front
# docker compose ps                 # état des services (healthy ?)
# docker compose logs -f api        # suivre les logs de l'API
# docker compose down               # arrête et supprime les conteneurs (garde les d
# docker compose down -v            # idem + supprime le volume (efface les données
# Application : http://localhost:8080
```

services:

```
# ----- Base de données PostgreSQL -----
db:
  image: postgres:16-alpine
  # Lues au PREMIER démarrage pour créer l'utilisateur et la base.
  # ${VAR:-défaut} = la valeur de VAR (fichier .env), sinon la valeur après « :- ».
  environment:
    POSTGRES_USER: taskflow
    POSTGRES_PASSWORD: ${POSTGRES_PASSWORD:-taskflow}
    POSTGRES_DB: taskflow
  # Volume NOMMÉ : les données survivent à la suppression du conteneur.
  volumes:
    - pgdata:/var/lib/postgresql/data
  # « La base est-elle prête à accepter des connexions ? »
  # (un conteneur démarré n'est pas forcément prêt)
```

```
healthcheck:
  test: ["CMD-SHELL", "pg_isready -U taskflow -d taskflow"]
  interval: 5s
  timeout: 3s
  retries: 10
  restart: unless-stopped
```

Déclaration des volumes nommés utilisés ci-dessus

```
volumes:
  pgdata:
```

Testez ce premier service seul :

```
docker compose config --quiet && echo "YAML valide"
docker compose up -d db
docker compose ps
```

Résultat attendu : YAML valide, puis au bout de quelques secondes :

NAME	IMAGE	COMMAND	SERVICE	CREATED
taskflow-db-1	postgres:16-alpine	"docker-entrypoint.s..."	db	10 seconds ago

Si ça ne marche pas - yml: line X: did not find expected key ou mapping values are not allowed here : erreur d'indentation. Le YAML n'accepte **que des espaces** (pas de tabulation), et chaque niveau est décalé de 2 espaces. - Le statut reste (health: starting) : attendez 10 secondes ; s'il devient unhealthy, docker compose logs db.

2.2.b : le service api

Ajoutez ce bloc **sous** le service db (au même niveau d'indentation que db: , donc 2 espaces) et **avant** la ligne volumes: du bas :

```
# ----- API Node -----
api:
  # Compose construit l'image à partir de api/Dockerfile
  build:
    context: ./api
  environment:
    PORT: "3000"
    # L'hôte est « db » (le nom du service), et non « localhost » !
    DATABASE_URL: postgres://taskflow:${POSTGRES_PASSWORD:-taskflow}@db:5432/taskflow
    APP_ENV: local
    APP_VERSION: dev
  # L'API ne démarre qu'une fois la base « healthy »
  depends_on:
    db:
      condition: service_healthy
  # Le HEALTHCHECK est déjà dans api/Dockerfile : inutile de le répéter ici.
  restart: unless-stopped
```

```
docker compose up -d --build api
docker compose ps
docker compose logs api
```

Résultat attendu : api passe à Up ... (healthy) au bout de 10 à 20 secondes, et ses logs contiennent "message": "API démarrée sur le port 3000". Notez que api n'a **pas** de port publié : on ne peut plus faire curl localhost:3000. C'est voulu.

Si ça ne marche pas - ECONNREFUSED dans les logs de l'API : DATABASE_URL contient localhost au lieu de db. - password authentication failed : le volume a été créé avec un autre mot de passe. docker compose down -v puis docker compose up -d --build api.

2.2.c : le service front

Ajoutez ce dernier service, toujours avant le volumes : du bas :

```
# ----- Front (Nginx) -----
front:
  build:
    context: ./front
  # "port du poste:port du conteneur", toujours entre guillemets
  ports:
    - "8080:80"
  depends_on:
    api:
      condition: service_healthy
  restart: unless-stopped
```

Vérifiez votre fichier complet : il doit avoir, dans l'ordre, services : avec db, api et front, puis volumes : avec pgdata :. En cas de doute, comparez avec [solution/compose.yaml](#).

Étape 2.3 : tout lancer en une commande (15 min)

```
docker compose down
docker compose up -d --build
docker compose ps
```

Résultat attendu : Compose crée le réseau taskflow_default, démarre db, **attend** qu'elle soit *healthy*, démarre api, attend, puis démarre front. Au bout de 30 secondes environ :

NAME	IMAGE	COMMAND	SERVICE	CREATED
taskflow-api-1	taskflow-api	"docker-entrypoint.s..."	api	35 seconds
taskflow-db-1	postgres:16-alpine	"docker-entrypoint.s..."	db	35 seconds
taskflow-front-1	taskflow-front	"/docker-entrypoint...."	front	35 seconds

Testez :

```
curl -s http://localhost:8080/health
```

```
{"status":"ok","db":"up","version":"dev","env":"local"}
```

Ouvrez <http://localhost:8080>, créez **deux tâches**, cochez-en une. Vérifiez qu’elles sont bien en base :

```
docker compose exec db psql -U taskflow -d taskflow -c "SELECT id, title, done FROM t
```

Résultat attendu : vos deux tâches, dont une avec done à t.

Suivez les logs de l’API en direct pendant que vous cliquez dans l’application (quittez avec Ctrl+C) :

```
docker compose logs -f api
```

Si ça ne marche pas - dependency failed to start: container taskflow-api-1 is unhealthy : l’API n’arrive pas à devenir *healthy*. docker compose logs api donne la raison (souvent DATABASE_URL). - Bind for 0.0.0.0:8080 failed: port is already allocated : un conteneur de l’étape 1 tourne encore (docker ps), ou un autre programme utilise 8080. Changez le port de gauche ("8081:80") si besoin. - La page affiche « Erreur de l’API (HTTP 502) » quelques secondes après le démarrage : attendez que api soit *healthy*, puis rechargez.

Pourquoi ? depends_on seul attend que le conteneur soit **démarré** ; avec condition: service_healthy, il attend qu’il soit **prêt**. Sans cela, l’API démarrerait avant que PostgreSQL accepte les connexions, et planterait.

Étape 2.4 : down, down -v et le volume (10 min)

Faites l’expérience dans l’ordre, et notez ce que vous observez à chaque fois :

```
docker compose down
docker compose ps                # plus rien
docker volume ls | grep pgdata   # le volume taskflow_pgdata existe toujours
docker compose up -d
```

Ouvrez <http://localhost:8080> : vos deux tâches sont **toujours là**.

```
docker compose down -v
docker volume ls | grep pgdata   # plus rien
docker compose up -d
```

Ouvrez <http://localhost:8080> : la liste est **vide**.

Résultat attendu : vous savez expliquer, en une phrase chacune, la différence entre down et down -v.

Pourquoi ? down supprime les conteneurs et le réseau, mais garde les volumes. down -v supprime **aussi** les volumes : c’est le bouton « remise à zéro » du développement. En production, on ne le tape **jamais**.

Étape 2.5 : documenter le démarrage dans le README.md (10 min)

Le test de l'étape 2.7 consiste à suivre **uniquement** le README. Ajoutez une section en haut du README.md (après la présentation) :

Démarrer avec Docker (recommandé)

Prérequis : Docker et Docker Compose v2 (*`docker compose version`*).

```
```bash
cp .env.example .env
docker compose up -d --build
docker compose ps # db, api et front doivent être "healthy" (environ 30 secondes)
```

- Application : <http://localhost:8080>
- Santé de l'API : <http://localhost:8080/health>
- Logs : `docker compose logs -f api`
- Arrêter : `docker compose down` (garde les données) ; tout effacer : `docker compose rm -f`
```

Étape 2.6 : commiter et ouvrir la PR (10 min)

```
git add compose.yaml .env.example README.md
git status
git commit -m "build: ajoute compose.yaml pour lancer TaskFlow en une commande"
git push -u origin feature/compose
```

Résultat attendu : `git status` (avant le commit) liste trois fichiers et **pas** `.env`.

Sur GitHub :

1. Ajoutez votre binôme comme collaborateur : **Settings > Collaborators > Add people** (il doit accepter l'invitation). Sans cela, son approbation ne compte pas.
2. Ouvrez la PR `build: ajoute compose.yaml...` et choisissez votre binôme dans **Reviewers**. **Ne fusionnez pas encore**.
3. Facultatif mais recommandé : dans la protection de main, cochez maintenant *Require approvals* (1).

Étape 2.7 : le test du nouvel arrivant, en binôme (15 min)

Votre binôme joue le **nouveau développeur** qui arrive dans l'équipe, sur **son** poste, puis vous inversez les rôles.

D'abord, il arrête sa propre stack (le port 8080 doit être libre) :

```
cd ~/taskflow && docker compose down
```

Puis il clone **votre** dépôt dans un dossier à part et suit **uniquement** votre README :

```
git clone git@github.com:<vous>/taskflow.git ~/taskflow-<vous>
cd ~/taskflow-<vous>
git switch feature/compose
cp .env.example .env
docker compose up -d --build
docker compose ps
```

Résultat attendu : sans **aucune autre** explication que le README, TaskFlow tourne sur <http://localhost:8080> chez votre binôme, et il peut créer une tâche.

Si ça bloque, **ce n'est pas votre binôme qu'il faut corriger, c'est le README ou le compose.yaml** : notez le problème en commentaire sur la PR, corrigez sur la branche feature/compose, poussez, et recommencez le test.

Une fois le test réussi, votre binôme nettoie :

```
docker compose down -v
cd ~ && rm -rf ~/taskflow-<vous>
```

Si ça ne marche pas - Le binôme a oublié `cp .env.example .env` : TaskFlow démarre quand même, car `compose.yaml` prévoit une valeur par défaut (`${POSTGRES_PASSWORD:-taskflow}`). Vérifiez seulement que l'étape est bien écrite dans le README. - Les deux projets se mélangent : chaque dossier est un projet Compose différent (son nom est celui du dossier), mais le port 8080 ne peut servir qu'à un seul à la fois. Arrêtez l'autre stack avec `docker compose down` dans son dossier.

Pourquoi ? C'est le test qui prouve que « ça marche sur ma machine » est réglé : si un collègue peut lancer le projet en une commande, un serveur aussi. Une documentation n'est bonne que si quelqu'un d'autre a réussi à la suivre.

Étape 2.8 : la revue de code par le binôme, puis le merge (10 min)

Votre binôme relit la PR, onglet **Files changed**, ligne à ligne :

- les noms de services (db, api, front) correspondent-ils à ceux utilisés dans DATABASE_URL et dans front/nginx.conf ?
- seul front publie-t-il un port ?
- le volume est-il déclaré et utilisé ? les healthchecks et depends_on sont-ils cohérents ?
- aucun mot de passe réel n'est-il commité (seul .env.example, avec une valeur de développement) ?

Il laisse **au moins deux commentaires** (une question et une suggestion, par exemple), puis **Approve**. Vous répondez aux commentaires, corrigez si besoin, puis **Merge pull request** et **Delete branch**.

```
git switch main
git pull
```

Je valide avec le formateur

- ☐ `docker compose ps` : db, api et front sont healthy.
- ☐ Une tâche créée dans le navigateur apparaît dans `psql` (`docker compose exec db psql ...`).
- ☐ Je sais expliquer la différence entre `down` et `down -v`, et à quoi sert `condition: service_healthy`.
- ☐ `.env.example` est commité, `.env` ne l'est pas.
- ☐ Mon binôme a lancé TaskFlow en suivant **uniquement** mon README.
- ☐ La PR a été relue (au moins deux commentaires), approuvée et fusionnée.

Questions de réflexion

1. Comparez avec hier : combien de commandes faut-il maintenant pour lancer TaskFlow ? Qu'est-ce qui a disparu ?
2. Pourquoi api n'a-t-il pas de ports : ? Qui l'appelle, et par quel nom ?
3. Retirez (en pensée) les `condition: service_healthy`. Que se passerait-il au premier démarrage ?
4. Vous changez `POSTGRES_PASSWORD` dans `.env` puis faites `docker compose up -d`. L'API n'arrive plus à se connecter. Pourquoi ? Comment s'en sortir en développement ? Et en production ?
5. Le nom des conteneurs commence par `taskflow-`. D'où vient ce préfixe ? Pourquoi est-ce utile le jour 4, quand une deuxième stack (`taskflow-prod`) tournera sur la même machine ?

Pour aller plus loin

- `docker compose config` : affichez le fichier tel que Compose le comprend, variables remplacées.
- `docker compose exec api sh` puis `wget -q0- http://db:5432` : que se passe-t-il ? (PostgreSQL ne parle pas HTTP.)
- Exercice : [../..exercices/jour3/02-compose-a-completer.md](https://github.com/leodotcloud/terraform-k8s/blob/master/exercices/jour3/02-compose-a-completer.md).