

Contents

Étape 1 : dockeriser TaskFlow	1
Prérequis	1
Ce que vous allez construire	2
Étape 1.0 : préparer la branche (2 min)	2
Étape 1.1 : créer le réseau et le volume (5 min)	2
Étape 1.2 : lancer PostgreSQL dans un conteneur (10 min)	3
Étape 1.3 : lancer l'API en local contre ce PostgreSQL (10 min)	4
Étape 1.4 : écrire api/Dockerfile et api/.dockerignore (20 min)	5
Étape 1.5 : lancer l'API dans un conteneur, sur le réseau (15 min)	7
Étape 1.6 : écrire le Dockerfile multi-stage du front et nginx.conf (25 min)	8
Étape 1.7 : lancer le front et tester l'application complète (10 min)	11
Étape 1.8 : vérifier la persistance des données (10 min)	11
Étape 1.9 : commiter et ouvrir la Pull Request (10 min)	12
Je valide avec le formateur	13
Questions de réflexion	13
Pour aller plus loin	13

Étape 1 : dockeriser TaskFlow

Quand : jour 2, après-midi | **Durée** : ~2h **Objectif** : faire tourner TaskFlow entièrement dans des conteneurs : PostgreSQL avec un volume, l'API dans une image que vous construisez (api/Dockerfile), le front dans une image multi-stage servie par Nginx (front/Dockerfile + front/nginx.conf), le tout relié par un réseau Docker. Puis vérifier que les données survivent à la suppression du conteneur de la base.

Prérequis

- Votre dépôt ~/taskflow créé depuis le starter, main protégée (voir [README.md](#)).
- Docker fonctionne : `docker run --rm hello-world` affiche Hello from Docker!.
- Slides du jour 2, blocs 3.2 à 3.4 ; antisèche [../ressources/cheatsheet-docker.md](#).

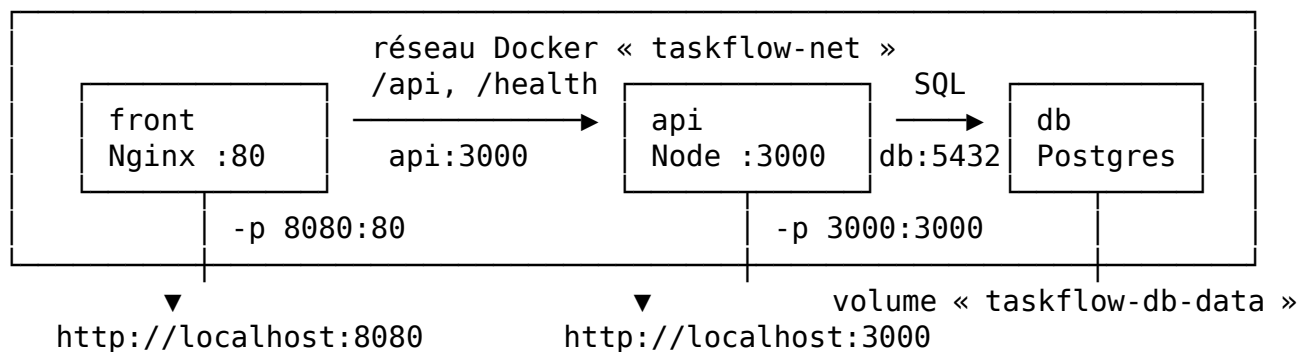
Vocabulaire

Terme	Signification
Image	Un modèle figé (système + dépendances + code) ; comme une classe en programmation
Conteneur	Une instance d'image en cours d'exécution ; comme un objet créé à partir de la classe
Dockerfile	La recette qui fabrique une image, instruction par instruction
Volume	Un espace de stockage géré par Docker, qui survit à la suppression des conteneurs

Terme	Signification
Réseau bridge	Un réseau privé entre conteneurs ; sur un réseau créé par vous, chaque conteneur est joignable par son nom
Multi-stage	Un Dockerfile en plusieurs étapes : on construit dans une image, on ne garde que le résultat dans une autre

Ce que vous allez construire

Votre poste



Toutes les commandes se lancent depuis la racine de votre dépôt (cd ~/taskflow), sauf mention contraire.

Étape 1.0 : préparer la branche (2 min)

```

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

Si vous aviez lancé la base du README.md du starter (taskflow-db), supprimez-la pour libérer le port 5432 :

```
docker rm -f taskflow-db
```

Résultat attendu : git branch affiche * feature/docker. La commande docker rm affiche taskflow-db, ou Error: No such container si elle n'existait pas (ce n'est pas grave).

Étape 1.1 : créer le réseau et le volume (5 min)

```

docker network create taskflow-net
docker volume create taskflow-db-data
  
```

```
docker network ls
docker volume ls
```

Résultat attendu :

NETWORK ID	NAME	DRIVER	SCOPE
...			
3c9a1f0b2d4e	taskflow-net	bridge	local

DRIVER	VOLUME NAME
local	taskflow-db-data

Si ça ne marche pas - network with name taskflow-net already exists : il existe déjà, continuez. - Cannot connect to the Docker daemon : Docker n'est pas démarré (lancez Docker Desktop, ou `sudo systemctl start docker`).

Pourquoi ? Sur un réseau **que vous créez**, Docker fournit un DNS (annuaire) interne : le conteneur api pourra joindre la base avec le nom db. Le réseau bridge par défaut, lui, ne résout pas les noms. Le volume gardera les données de PostgreSQL en dehors du conteneur.

Étape 1.2 : lancer PostgreSQL dans un conteneur (10 min)

```
docker run -d --name db --network taskflow-net \
-e POSTGRES_USER=taskflow -e POSTGRES_PASSWORD=taskflow -e POSTGRES_DB=taskflow \
-v taskflow-db-data:/var/lib/postgresql/data \
-p 5432:5432 \
postgres:16-alpine
```

Option	Rôle
-d	En arrière-plan (<i>detached</i>) : le terminal vous est rendu
--name db	Le nom du conteneur, qui sera aussi son nom sur le réseau
--network taskflow-net	Le branche sur notre réseau
-e POSTGRES_...	Variables lues au premier démarrage pour créer l'utilisateur, son mot de passe et la base
-v taskflow-db-data:/var/lib/postgresql/data	Monte le volume là où PostgreSQL range ses fichiers
-p 5432:5432	Publie le port sur votre poste : temporaire , pour que l'API lancée hors Docker (étape 1.3) puisse s'y connecter

Vérifiez :

```
docker ps
docker logs db
docker exec db psql -U taskflow -d taskflow -c "SELECT version();"
```

Résultat attendu : docker ps montre db avec le statut Up. Les dernières lignes de docker logs db contiennent :

```
LOG: database system is ready to accept connections
```

et psql affiche PostgreSQL 16.x

Si ça ne marche pas - port is already allocated ou address already in use sur 5432 : un autre PostgreSQL tourne (un conteneur oublié, ou un PostgreSQL installé sur votre poste). docker ps -a pour le trouver et docker rm -f <nom>. Sinon, publiez un autre port (-p 5433:5432) et utilisez 5433 dans l'URL de l'étape 1.3. - Conflict. The container name "/db" is already in use : un ancien conteneur db existe. docker rm -f db et relancez. - Le conteneur s'arrête tout de suite : docker logs db. Le message superuser password is not specified signifie que -e POSTGRES_PASSWORD=... manque.

Pourquoi ? Plus besoin d'installer PostgreSQL : une commande, et la même version (16) tourne chez tout le monde. docker exec lance une commande **dans** un conteneur déjà démarré, ici le client psql.

Étape 1.3 : lancer l'API en local contre ce PostgreSQL (10 min)

Avant de mettre l'API dans un conteneur, on vérifie qu'elle fonctionne avec cette base, lancée directement par Node.

```
cd ~/taskflow/api
npm run dev
```

Résultat attendu (le terminal reste occupé par l'API) :

```
{"time":"...","level":"info","message":"API démarrée sur le port 3000","env":"local",
```

Dans un **second terminal** :

```
curl -s http://localhost:3000/health
```

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

Lancez aussi les tests d'intégration, qui étaient ignorés jusqu'ici (toujours dans le second terminal) :

```
cd ~/taskflow/api
DATABASE_URL=postgres://taskflow:taskflow@localhost:5432/taskflow npm run test:int
```

Résultat attendu : Tests 10 passed (10).

Attention : ces tests **vident la table tasks** avant chaque test (TRUNCATE). Ici la base est neuve, aucun risque. Ne les lancez jamais contre une base qui contient de vraies données (et surtout pas la production du jour 4).

Arrêtez ensuite l'API du premier terminal avec Ctrl+C (le port 3000 doit être libre pour la suite) et revenez à la racine : `cd ~/taskflow`.

Si ça ne marche pas - `{"level":"fatal","message":"Démarrage impossible : ..."} : la base n'est pas joignable sur localhost:5432. Vérifiez que db tourne (docker ps) et que vous avez bien mis -p 5432:5432. - password authentication failed for user "taskflow" : le volume a été créé lors d'un essai précédent avec un autre mot de passe. Repartez de zéro : docker rm -f db && docker volume rm taskflow-db-data, puis étapes 1.1 (volume) et 1.2. - npm run dev : Cannot find module 'express' : faites npm ci dans api/.`

Pourquoi ? On change **une chose à la fois** : d'abord la base dans Docker (l'API reste « normale »), puis l'API. Si quelque chose casse ensuite, vous saurez que c'est le Dockerfile, pas le code.

Étape 1.4 : écrire api/Dockerfile et api/.dockerignore (20 min)

Créez le fichier api/Dockerfile :

```
# Image de base : Node 20 sur Alpine Linux (une distribution minuscule).
# On fixe la version majeure pour que le build reste reproductible.
FROM node:20-alpine

# Dossier de travail dans l'image : les commandes suivantes s'exécutent ici.
WORKDIR /app

ENV NODE_ENV=production

# Astuce de cache : on copie D'ABORD les fichiers de dépendances seulement.
# Tant qu'ils ne changent pas, Docker réutilise la couche « npm ci » ci-dessous.
COPY package.json package-lock.json ./

# Installation stricte d'après le package-lock.json, sans les outils de dev (ESLint,
RUN npm ci --omit=dev

# Puis le code source, qui change souvent.
COPY src ./src

# Version affichée par /health. Le pipeline la fournira : --build-arg APP_VERSION=<sh
ARG APP_VERSION=dev
ENV APP_VERSION=$APP_VERSION

# Sécurité : ne pas tourner en root. L'utilisateur « node » existe déjà dans l'image.
USER node

# Documentation : l'API écoute sur 3000 (publier le port, c'est -p au lancement).
EXPOSE 3000

# Docker appelle /health toutes les 10 s. Trois échecs de suite : le conteneur devien
HEALTHCHECK --interval=10s --timeout=3s --start-period=10s --retries=3 \
```

```
CMD wget -q -O /dev/null http://127.0.0.1:3000/health || exit 1
```

Commande de démarrage (forme « exec » : le signal d'arrêt de docker stop arrive bien)

```
CMD ["node", "src/server.js"]
```

Créez le fichier api/.dockerignore :

```
# Fichiers exclus du « contexte de build » envoyé à Docker.
```

```
# Les dépendances sont réinstallées DANS l'image (npm ci) : ne pas copier celles du poste.
```

```
node_modules
```

```
# Secrets locaux : jamais dans une image
```

```
.env
```

```
.env.*
```

```
# Inutile pour faire tourner l'API
```

```
tests
```

```
reports
```

```
coverage
```

```
eslint.config.js
```

```
.prettierrc.json
```

```
.prettierignore
```

```
Dockerfile
```

```
.dockerignore
```

```
*.log
```

Construisez l'image (depuis ~/taskflow) :

```
docker build -t taskflow-api:dev api
```

```
docker images taskflow-api
```

Résultat attendu : le build se termine sans erreur, par une ligne du type naming to docker.io/library/taskflow-api:dev, puis :

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
taskflow-api	dev	7d1e2f3a4b5c	10 seconds ago	~140MB

Relancez la même commande docker build : elle prend une seconde, toutes les étapes affichent CACHED. Modifiez un commentaire dans api/src/server.js, relancez : seules les étapes à partir de COPY src ./src sont refaites, **pas** npm ci. Annulez ensuite votre modification (git checkout api/src/server.js).

Si ça ne marche pas - failed to read dockerfile: open Dockerfile: no such file or directory : le fichier doit s'appeler exactement Dockerfile (D majuscule, sans extension), dans api/. Le dernier argument de docker build est le **dossier** api. - npm ci échoue : package-lock.json n'est pas dans le contexte. Vérifiez la ligne COPY et que .dockerignore n'exclut pas ce fichier. - COPY failed: file not found : vous lancez docker build depuis le mauvais dossier. pwd doit afficher ~/taskflow.

Pourquoi ? L'ordre des instructions compte : Docker met chaque couche en cache et ne refait que ce qui suit un changement. Dépendances d'abord, code ensuite : les builds suivants prennent quelques secondes au lieu d'une minute.

.dockerignore rend le build plus rapide et empêche un .env de finir dans une image partagée.

Étape 1.5 : lancer l'API dans un conteneur, sur le réseau (15 min)

Dans un conteneur, localhost désigne **le conteneur lui-même** : l'API doit joindre la base par son nom, db. Créez le fichier api/.env (il est ignoré par Git et par Docker, vérifiez-le à la fin de l'étape) :

```
cat > api/.env <<'EOF'
DATABASE_URL=postgres://taskflow:taskflow@db:5432/taskflow
APP_ENV=local
EOF
```

Lancez le conteneur :

```
docker run -d --name api --network taskflow-net -p 3000:3000 --env-file api/.env taskflow
docker logs api
```

Résultat attendu :

```
{"time":"...","level":"info","message":"API démarrée sur le port 3000","env":"local",
```

Testez l'API :

```
curl -s http://localhost:3000/health
curl -s -X POST http://localhost:3000/api/tasks -H "Content-Type: application/json" -d {}
curl -s http://localhost:3000/api/tasks
```

Résultat attendu :

```
{"status":"ok","db":"up","version":"dev","env":"local"}
{"id":1,"title":"Dockeriser TaskFlow","done":false,"created_at":"2026-10-20T14:58:12.345Z"}
[{"id":1,"title":"Dockeriser TaskFlow","done":false,"created_at":"2026-10-20T14:58:12.345Z"}]
```

(Les tests d'intégration de l'étape 1.3 ont utilisé la même base : l'id peut valoir plus que 1, et une tâche « en base » laissée par ces tests peut apparaître dans la liste. Ce n'est pas grave.)

Attendez une quinzaine de secondes, puis :

```
docker ps --format 'table {{.Names}}\t{{.Image}}\t{{.Status}}'
docker logs api
git status
```

Résultat attendu : api est Up ... (healthy) ; les logs contiennent une ligne JSON par requête ("method":"GET","path":"/health","status":200,"ms":...) ; git status **ne liste pas** api/.env.

Si ça ne marche pas - docker logs api affiche Démarrage impossible : connect ECONNREFUSED 127.0.0.1:5432 (ou un message vide après « Démarrage impossible : ») : l'API cherche la base sur localhost. Vérifiez que api/.env contient bien l'hôte db et que vous avez passé --env-file api/.env. - getad-drinfo ENOTFOUND db : api et db ne sont pas sur le même réseau. Il faut --

network taskflow-net aux deux. - port is already allocated sur 3000 : l'API de l'étape 1.3 tourne encore (faites Ctrl+C dans son terminal). - Pour corriger et relancer : `docker rm -f api`, puis à nouveau la commande `docker run`. - Le statut reste (health: starting) : attendez 10 à 20 secondes. S'il passe à (unhealthy), regardez `docker inspect --format '{{json .State.Health}}' api`.

Pourquoi ? La **même image** tournera en développement, en test et en production : seule la configuration change, et elle est passée **au lancement** (`--env-file`), jamais écrite dans l'image. Un secret dans une image est lisible par toute personne qui la télécharge (`docker history`).

Étape 1.6 : écrire le Dockerfile multi-stage du front et nginx.conf (25 min)

Le front est un site **statique** une fois compilé par Vite (`npm run build` produit `dist/`). En production, il n'y a plus besoin de Node : Nginx sert les fichiers, et relaie les appels `/api/...` et `/health` vers l'API. C'est ce que faisait le proxy de Vite en développement (voir `front/vite.config.js`).

Créez le fichier `front/Dockerfile` :

```
# ----- Étape 1 : construire le site -----
# « AS build » nomme cette étape pour pouvoir y faire référence plus bas.
FROM node:20-alpine AS build
WORKDIR /app

# Même astuce de cache que pour l'API : les dépendances d'abord, le code ensuite.
COPY package.json package-lock.json ./
RUN npm ci

COPY . .
# Produit le dossier /app/dist (le site compilé)
RUN npm run build

# ----- Étape 2 : servir le site avec Nginx -----
FROM nginx:1.27-alpine

# L'image Nginx transforme les fichiers /etc/nginx/templates/*.template en vraie
# configuration au démarrage, en y remplaçant les ${VARIABLES}.
# Cette option lui fait calculer l'adresse du DNS du réseau Docker (voir nginx.conf).
ENV NGINX_ENTRYPOINT_LOCAL_RESOLVERS=1
COPY nginx.conf /etc/nginx/templates/default.conf.template

# On ne copie QUE le résultat du build de l'étape 1 : ni Node, ni node_modules, ni le
COPY --from=build /app/dist /usr/share/nginx/html

EXPOSE 80

# Nginx répond-il ? (la page d'accueil, sans dépendre de l'API)
```

```
HEALTHCHECK --interval=10s --timeout=3s --start-period=5s --retries=3 \
  CMD wget -q -O /dev/null http://127.0.0.1/ || exit 1
```

Pas de CMD : l'image Nginx a déjà le bon.

Créez le fichier front/nginx.conf :

```
# Configuration Nginx du conteneur « front » :
# (1) servir le site compilé, (2) transmettre /api et /health à l'API.
# Ce fichier est un modèle : ${NGINX_LOCAL_RESOLVERS} est remplacé au démarrage
# du conteneur par l'adresse du DNS du réseau Docker.

server {
    listen 80;
    server_name _;

    root /usr/share/nginx/html;
    index index.html;

    # DNS utilisé pour retrouver l'adresse du conteneur « api ». valid=10s : on redem
    # l'adresse toutes les 10 s. Sans cela, Nginx garderait l'adresse IP de l'API tro
    # au démarrage : après un redémarrage de l'API (nouvelle IP), erreur 502.
    resolver ${NGINX_LOCAL_RESOLVERS} valid=10s ipv6=off;

    # « api » est le nom du conteneur (puis du service Compose). Le mettre dans une
    # variable oblige Nginx à le résoudre à chaque requête, et pas une seule fois au
    set $api_upstream http://api:3000;

    # Les appels /api/... partent vers l'API (le chemin est transmis tel quel).
    location /api/ {
        proxy_pass $api_upstream;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }

    # /health aussi : utile pour le smoke test et la supervision depuis l'extérieur.
    location = /health {
        proxy_pass $api_upstream;
        proxy_set_header Host $host;
    }

    # Tout le reste : les fichiers du site ; si le fichier n'existe pas, index.html.
    location / {
        try_files $uri $uri/ /index.html;
    }

    # Les fichiers générés par Vite ont un nom qui change à chaque build : cache long
    location /assets/ {
```

```

        expires 1y;
        add_header Cache-Control "public, immutable";
    }
}

```

Créez le fichier front/.dockerignore :

```

node_modules
dist
.env
.env.*
tests
reports
coverage
eslint.config.js
.prettierrc.json
.prettierignore
Dockerfile
.dockerignore
*.log

```

Construisez et comparez la taille :

```

docker build -t taskflow-front:dev front
docker images --format 'table {{.Repository}}\t{{.Tag}}\t{{.Size}}' | grep -E 'REPOS

```

Résultat attendu : taskflow-front:dev pèse **moins de 60 Mo**, alors que l'image node:20-alpine utilisée pour le build en pèse plus du double.

Le modèle et le resolver, en clair - Au démarrage, l'image Nginx lit /etc/nginx/templates/default.conf.template, remplace \${NGINX_LOCAL_RESOLVERS} par l'adresse du DNS du réseau (sur Docker, souvent 127.0.0.11) et écrit le résultat dans /etc/nginx/conf.d/default.conf. Seules les variables d'environnement existantes sont remplacées : \$host, \$uri ou \$api_upstream restent intactes. - Grâce à la variable \$api_upstream et au resolver, Nginx cherche l'adresse de api **au moment de la requête**. Conséquences : le front démarre même si l'API n'est pas encore là, et il retrouve l'API après son redémarrage. La version « simple » des slides (proxy_pass http://api:3000;) fonctionne aussi, mais refuse de démarrer si api n'existe pas encore, et garde l'ancienne adresse après un redémarrage de l'API.

Si ça ne marche pas - Le build échoue à RUN npm run build : lisez le message de Vite ; vérifiez que .dockerignore n'exclut pas src ou index.html. - COPY nginx.conf ...: not found : le fichier doit être front/nginx.conf (à côté du Dockerfile du front). - L'image pèse plus de 300 Mo : la deuxième étape FROM nginx:1.27-alpine manque, ou vous avez copié /app au lieu de /app/dist.

Pourquoi ? Le multi-stage sépare la cuisine (Node, node_modules, les sources) de l'assiette servie au client (Nginx + dist/). Image plus petite : téléchargement plus rapide sur le serveur, moins d'outils pour un attaquant.

Étape 1.7 : lancer le front et tester l'application complète (10 min)

```
docker run -d --name front --network taskflow-net -p 8080:80 taskflow-front:dev
docker logs front
```

Résultat attendu : parmi les lignes de démarrage de Nginx, vous voyez la génération de la configuration :

```
20-envsubst-on-templates.sh: Running envsubst on /etc/nginx/templates/default.conf.template
```

Vérifiez la configuration réellement utilisée, puis testez :

```
docker exec front grep resolver /etc/nginx/conf.d/default.conf
curl -s http://localhost:8080/health
```

Résultat attendu :

```
resolver 127.0.0.11 valid=10s ipv6=off;
{"status":"ok","db":"up","version":"dev","env":"local"}
```

(L'adresse du resolver peut être différente selon votre installation : l'important est qu'il y ait une adresse.)

Ouvrez <http://localhost:8080> : la tâche « Dockeriser TaskFlow » apparaît. Ajoutez deux tâches, cochez-en une. Le pied de page affiche API : ok · base : up · version dev (local).

Si ça ne marche pas - La page s'affiche mais « Erreur de l'API (HTTP 502) » : Nginx ne joint pas l'API. Le conteneur doit s'appeler **exactement** api et être sur taskflow-net (docker network inspect taskflow-net). Regardez docker logs front. - Le conteneur front s'arrête aussitôt et docker logs front affiche host not found in resolver "\${NGINX_LOCAL_RESOLVERS}" : la ligne ENV NGINX_ENTRYPOINT_LOCAL_RESOLVERS=1 manque dans le Dockerfile, ou nginx.conf a été copié ailleurs que dans /etc/nginx/templates/. Corrigez, reconstruisez l'image, puis docker rm -f front et relancez. - docker logs front affiche api could not be resolved (3: Host not found) : le conteneur api est arrêté, mal nommé ou sur un autre réseau. - port is already allocated sur 8080 : changez le port de gauche (-p 8081:80) et ouvrez <http://localhost:8081>.

Pourquoi ? Le navigateur ne parle qu'au front (port 8080). C'est Nginx qui relaie vers l'API : pas de problème de CORS (le blocage par le navigateur des appels vers une autre adresse), et l'API n'aurait même pas besoin d'être publiée.

Étape 1.8 : vérifier la persistance des données (10 min)

On simule la perte du conteneur de la base.

```
docker rm -f db
curl -s http://localhost:3000/health
```

Résultat attendu : {"status":"error","db":"down",...} : l'API tourne toujours mais signale que la base ne répond plus (code HTTP 503). Dans le navigateur, le pied de page affiche base : down après un rechargement.

Recréez la base avec **le même volume**, et cette fois **sans** publier le port 5432 (seule l'API a besoin de la joindre, par le réseau Docker) :

```
docker run -d --name db --network taskflow-net \
  -e POSTGRES_USER=taskflow -e POSTGRES_PASSWORD=taskflow -e POSTGRES_DB=taskflow \
  -v taskflow-db-data:/var/lib/postgresql/data \
  postgres:16-alpine
```

Attendez 5 secondes, puis :

```
curl -s http://localhost:3000/health
docker exec db psql -U taskflow -d taskflow -c "SELECT id, title, done FROM tasks;"
docker ps --format 'table {{.Names}}\t{{.Image}}\t{{.Status}}\t{{.Ports}}'
```

Résultat attendu : /health répond de nouveau "db": "up", et vos tâches sont **toujours** là :

id	title	done
1	Dockeriser TaskFlow	f
2	Écrire le Dockerfile	t
...		

NAMES	IMAGE	STATUS	PORTS
db	postgres:16-alpine	Up 20 seconds	5432/tcp
front	taskflow-front:dev	Up 8 minutes (healthy)	0.0.0.0:8080->80/tcp
api	taskflow-api:dev	Up 25 minutes (healthy)	0.0.0.0:3000->3000/tcp

Notez la colonne PORTS de db : 5432/tcp sans flèche signifie « port ouvert dans le réseau Docker, **pas** publié sur votre poste ».

Si ça ne marche pas - Les tâches ont disparu : le volume n'est pas le même. Vérifiez `docker volume ls` et que le `-v` est identique aux deux lancements (une faute de frappe crée un **nouveau** volume vide). - /health reste à "db": "down" plus de 30 secondes : vérifiez que la nouvelle base s'appelle bien db et est sur taskflow-net, puis regardez `docker logs api`.

Pourquoi ? Un conteneur est **jetable** ; les données ne doivent pas l'être. Le volume vit en dehors du conteneur : on peut supprimer, mettre à jour ou recréer la base sans rien perdre. C'est exactement ce qui se passera à chaque déploiement le jour 4.

Étape 1.9 : commiter et ouvrir la Pull Request (10 min)

```
git status
git add api/Dockerfile api/.dockerignore front/Dockerfile front/.dockerignore front/r
git status
git commit -m "build: dockerise l'API et le front"
git push -u origin feature/docker
```

Résultat attendu : le second `git status` liste **cinq** fichiers à commiter, et **pas** `api/.env`.

Ouvrez la PR sur GitHub (*Compare & pull request*), titre build: dockerise l'API et le front. Dans la description, indiquez comment tester (les commandes `docker build` et `docker run`). Relisez l'onglet *Files changed*, puis fusionnez (**Merge pull request**) et supprimez la branche.

Mettez à jour votre main locale :

```
git switch main
git pull
```

Gardez vos conteneurs pour l'instant : l'étape 2 commencera par les supprimer.

Si ça ne marche pas - `api/.env` apparaît dans `git status` : le `.gitignore` du starter n'est pas à la racine du dépôt. Ne l'ajoutez surtout pas : `git restore --staged api/.env` s'il était déjà ajouté, puis vérifiez le `.gitignore`.

Je valide avec le formateur

- ☐ `docker ps` montre db, api (healthy) et front (healthy), tous sur le réseau taskflow-net.
- ☐ `curl -s http://localhost:8080/health` répond "db":"up" et je peux créer une tâche dans le navigateur.
- ☐ L'image taskflow-front:dev pèse moins de 60 Mo ; je sais expliquer pourquoi.
- ☐ Mes tâches ont survécu à `docker rm -f db` grâce au volume.
- ☐ `api/.env` n'est ni dans Git ni dans l'image.
- ☐ La PR build: dockerise l'API et le front est fusionnée dans main.

Questions de réflexion

À traiter **après** la validation, par écrit, en deux ou trois phrases chacune :

1. Pourquoi le port 5432 n'est-il plus publié à la fin ? Qui a encore besoin de joindre la base, et par quel chemin ?
2. Que se serait-il passé à l'étape 1.8 sans l'option `-v` ?
3. Pourquoi copier `package.json` et `package-lock.json` **avant** `src/` dans `api/Dockerfile` ?
4. Pourquoi USER `node` ? Qu'est-ce qu'un attaquant pourrait faire de plus si l'API tournait en `root` ?
5. Dans `api/.env`, l'hôte est `db` ; dans l'étape 1.3, c'était `localhost`. Expliquez la différence.
6. Comptez les commandes `docker run` tapées aujourd'hui, avec toutes leurs options. Que se passe-t-il si un collègue en oublie une ? (C'est le problème que résout Docker Compose, demain matin.)

Pour aller plus loin

- `docker history taskflow-api:dev` : retrouvez chaque instruction du Dockerfile et la taille de chaque couche.
- `docker exec -it api sh`, puis `whoami` et `ls -la` : vérifiez que l'API tourne avec l'utilisateur `node` et que ni `tests/` ni `.env` ne sont dans l'image.

- Exercices : [../..exercices/jour2/](#) (Dockerfile à corriger, volumes et persistance).