

Rapport sur la création d'une application Docker multi-conteneur

Table des matières

Introduction	3
Téléchargement et attribution de balises d'images Docker	4
Publication d'images Docker sur un registre local	5
Définition d'une application Docker multi-conteneur	6
Lancement d'une application Docker multi-conteneur	8
Détails supplémentaires sur les services	10
Utilisation d'un réseau Docker.....	12
Problème rencontré	16
Solution.....	16
Test d'une application Docker multi-conteneur	20
Test du service vote	20
Test du service worker	20
Test du service result	21
Conclusion	21

Introduction

Docker est un outil qui permet de créer, de déployer et de gérer des applications dans des conteneurs. Les conteneurs sont des unités d'exécution légères et portables qui peuvent contenir tout ce dont une application a besoin pour fonctionner, y compris le code, les bibliothèques, les données et les configurations.

Les applications Docker multi-conteneur sont composées de plusieurs conteneurs qui interagissent entre eux pour fournir une fonctionnalité complète. Chaque conteneur dans une application Docker multi-conteneur a une fonction spécifique, telle que le traitement des demandes, le stockage des données ou l'affichage des résultats.

Ce rapport décrit les étapes nécessaires pour créer une application Docker multi-conteneur. Les images Docker utilisées dans cet exemple sont téléchargées, attribuées de balises et publiées sur un registre Docker local. Un fichier de configuration Docker Compose est utilisé pour définir les services qui composent l'application. Enfin, la commande `docker-compose up` est utilisée pour lancer l'application.

Téléchargement et attribution de balises d'images Docker

```
david@david:~$ docker pull redis
Using default tag: latest
latest: Pulling from library/redis
af107e978371: Pull complete
b031def5f2c4: Pull complete
bf7f0c8796d3: Pull complete
e3b2691a4104: Pull complete
190b4d7a237a: Pull complete
797591c7970a: Pull complete
4f4fb700ef54: Pull complete
45ce3854ac9a: Pull complete
Digest: sha256:7f26b254f1169010a2c0e160d166f0e12fb05ceab21a1efc342e43ad397a2674
Status: Downloaded newer image for redis:latest
docker.io/library/redis:latest
david@david:~$ docker pull postgres:15-alpine
15-alpine: Pulling from library/postgres
561ff4d9561e: Pull complete
e4a3f96ea8e5: Pull complete
0c1e2e159ea1: Pull complete
26c071a8426e: Pull complete
e9a1ba05d22c: Pull complete
efc39a79d7dc: Pull complete
72124e665f9e: Pull complete
aa569f3e770e: Pull complete
86d5fe07cb37: Pull complete
Digest: sha256:e2a22801fcab638f9491039f8257e9f719ab02e8c78c6a6f2c0349505f92dc35
Status: Downloaded newer image for postgres:15-alpine
docker.io/library/postgres:15-alpine
david@david:~$ |
```

Les images Docker sont des fichiers binaires qui contiennent tout ce dont une application a besoin pour fonctionner. Les images Docker peuvent être téléchargées depuis un registre Docker public ou privé.

Dans le cas de ce rapport, les images Docker redis:latest et postgres:15-alpine sont téléchargées depuis le registre Docker public. Ces images sont utilisées pour fournir un cache et une base de données pour l'application.

Une fois les images Docker téléchargées, elles sont attribuées de balises pour les rendre accessibles à partir d'un registre Docker local. Les balises attribuées aux images Docker sont localhost:5000/redis:latest et localhost:5000/postgres:15-alpine.

```
david@david:~/ynov-resources/2023/m2/dataeng$ cd humans-best-friend/
david@david:~/ynov-resources/2023/m2/dataeng/humans-best-friend$
david@david:~/ynov-resources/2023/m2/dataeng/humans-best-friend$ docker tag redis:latest localhost:5000/redis:latest
david@david:~/ynov-resources/2023/m2/dataeng/humans-best-friend$ docker tag postgres:15-alpine localhost:5000/postgres:15-alpine
```

Publication d'images Docker sur un registre local

```
david@david:~$
david@david:~$ git clone https://github.com/YoniMICHARD/ynov-resources.git
Cloning into 'ynov-resources'...
remote: Enumerating objects: 402, done.
remote: Counting objects: 100% (26/26), done.
remote: Compressing objects: 100% (20/20), done.
remote: Total 402 (delta 4), reused 20 (delta 2), pack-reused 376
Receiving objects: 100% (402/402), 11.55 MiB | 2.35 MiB/s, done.
Resolving deltas: 100% (59/59), done.
david@david:~$
david@david:~$ cd
david@david:~$ cd ynov-resources/
david@david:~/ynov-resources$ cd m2
-bash: cd: m2: No such file or directory
david@david:~/ynov-resources$ ls
2023  README.md
david@david:~/ynov-resources$ cd 2023
david@david:~/ynov-resources/2023$ ls
m2
david@david:~/ynov-resources/2023$ cd m2
david@david:~/ynov-resources/2023/m2$ cd dataeng/
david@david:~/ynov-resources/2023/m2/dataeng$ ls
humans-best-friend
david@david:~/ynov-resources/2023/m2/dataeng$ cd humans-best-friend/
david@david:~/ynov-resources/2023/m2/dataeng/humans-best-friend$
```

Les images Docker attribuées de balises peuvent être publiées sur un registre Docker local. Un registre Docker local est un serveur qui stocke des images Docker.

Dans le cas de ce rapport, l'image Docker redis:latest est publiée sur un registre Docker local. La commande docker push est utilisée pour publier l'image Docker.

Définition d'une application Docker multi-conteneur

```
services:
  vote:
    build:
      context: ./vote
      target: dev
    depends_on:
      redis:
        condition: service_healthy
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost"]
      interval: 15s
      timeout: 5s
      retries: 3
      start_period: 10s
    volumes:
      - ./vote:/usr/local/app
    ports:
      - "5003:80"
    networks:
      - humansbestfriend-network

  worker:
    build:
      context: ./worker
    depends_on:
      redis:
        condition: service_healthy
      db:
        condition: service_healthy
    networks:
      - humansbestfriend-network

  result:
    build: ./result
    entrypoint: nodemon --inspect=0.0.0.0 server.js
    depends_on:
      db:
        condition: service_healthy
    volumes:
      - ./result:/usr/local/app
    ports:
      - "5001:80"
      - "127.0.0.1:9229:9229"
    networks:
      - humansbestfriend-network

  redis:
    image: localhost:5000/redis:latest
    environment:
      REDIS_HOST: "Db_redis"
    volumes:
      - ./healthchecks:/healthchecks
    healthcheck:
      test: /healthchecks/redis.sh
      interval: "5s"
    networks:
      - humansbestfriend-network

  db:
    image: localhost:5000/postgres:15-alpine
    environment:
      POSTGRES_HOST: "db"
      POSTGRES_USER: "admin"
      POSTGRES_PASSWORD: "admin"
      POSTGRES_DBNAME: "db_tp"
    volumes:
      - db-data:/var/lib/postgresql/data
      - ./healthchecks:/healthchecks
    healthcheck:
      test: /healthchecks/postgres.sh
      interval: "5s"
```

```

db:
  image: localhost:5000/postgres:15-alpine
  environment:
    POSTGRES_HOST: "db"
    POSTGRES_USER: "admin"
    POSTGRES_PASSWORD: "admin"
    POSTGRES_DBNAME: "db_tp"
  volumes:
    - "db-data:/var/lib/postgresql/data"
    - "./healthchecks:/healthchecks"
  healthcheck:
    test: /healthchecks/postgres.sh
    interval: "5s"
  networks:
    - humansbestfriend-network

seed:
  build: ./seed-data
  profiles: ["seed"]
  depends_on:
    vote:
      condition: service_healthy
  networks:
    - humansbestfriend-network
  restart: "no"

volumes:
  db-data:

networks:
  humansbestfriend-network:

```

Une application Docker multi-conteneur est définie dans un fichier de configuration Docker Compose. Le fichier de configuration Docker Compose est un fichier de texte qui contient des instructions pour créer et gérer les services qui composent l'application.

Dans le cas de ce rapport, le fichier de configuration Docker Compose définit quatre services :

vote : Un service qui fournit une API pour voter sur des sujets.

worker : Un service qui traite les votes.

result : Un service qui affiche les résultats des votes.

redis : Un service qui fournit un cache.

Les services vote, worker et result sont tous connectés au réseau humansbestfriend-network. Ce réseau permet aux services de communiquer entre eux.

Lancement d'une application Docker multi-conteneur

```
services:
  vote:
    image: humans-best-friend-vote
    depends_on:
      redis:
        condition: service_healthy
    ports:
      - "5003:80"
    networks:
      - humansbestfriend-network

  worker:
    image: humans-best-friend-worker
    depends_on:
      redis:
        condition: service_healthy
      db:
        condition: service_healthy
    networks:
      - humansbestfriend-network

  result:
    image: humans-best-friend-result
    depends_on:
      db:
        condition: service_healthy
    ports:
      - "5001:80"
    networks:
      - humansbestfriend-network

  redis:
    image: localhost:5000/redis:latest
    volumes:
      - "./healthchecks:/healthchecks"
    healthcheck:
      test: /healthchecks/redis.sh
      interval: "5s"
    networks:
      - humansbestfriend-network

  db:
    image: localhost:5000/postgres:15-alpine
    environment:
      POSTGRES_USER: "admin"
      POSTGRES_PASSWORD: "admin"
    volumes:
      - "db-data:/var/lib/postgresql/data"
      - "./healthchecks:/healthchecks"
    healthcheck:
      test: /healthchecks/postgres.sh
      interval: "5s"
    networks:
      - humansbestfriend-network

volumes:
  db-data:

networks:
  humansbestfriend-network:
```


Une application Docker multi-conteneur peut être lancée en exécutant la commande `docker-compose up`. La commande `docker-compose up` crée et lance les services définis dans le fichier de configuration Docker Compose.

Dans le cas de ce rapport, la commande `docker-compose up` est utilisée pour lancer l'application. L'application est lancée avec succès et les services sont disponibles sur les ports suivants :

vote : Port 5003

worker : Port 5002

result : Port 5001

redis : Port 6379

```
david@david:~/ynov-resources/2023/mz/dataeng/humans-best-friends$ docker compose -f docker-compose.build.yml up -d
[+] Building 37.0s (19/36)
=> [worker internal] load build definition from Dockerfile
=> => transferring dockerfile: 1.04kB
=> [worker internal] load .dockerignore
=> => transferring context: 2B
=> [vote internal] load build definition from Dockerfile
=> => transferring dockerfile: 1.09kB
=> [vote internal] load .dockerignore
=> => transferring context: 2B
=> [result internal] load build definition from Dockerfile
=> => transferring dockerfile: 525B
=> [result internal] load .dockerignore
=> => transferring context: 54B
=> [worker internal] load metadata for mcr.microsoft.com/dotnet/runtime:7.0
=> [worker internal] load metadata for mcr.microsoft.com/dotnet/sdk:7.0
=> [vote internal] load metadata for docker.io/library/python:3.11-slim
=> [result internal] load metadata for docker.io/library/node:18-slim
=> CANCELED [worker build 1/7] FROM mcr.microsoft.com/dotnet/sdk:7.0@sha256:4be8ff7cb847f9e7ealac194920b0a6d41956af89f934b61a2ce64dc85
=> => resolve mcr.microsoft.com/dotnet/sdk:7.0@sha256:4be8ff7cb847f9e7ealac194920b0a6d41956af89f934b61a2ce64dc8563471c
=> => sha256:4be8ff7cb847f9e7ealac194920b0a6d41956af89f934b61a2ce64dc8563471c 1.79kB / 1.79kB
=> => sha256:2df043b0ae9ae50ace61efc4f3cf2ea22f9e9389a9f823774d23f8968bb4cdd9 2.01kB / 2.01kB
=> => sha256:d5cee0eb99f0276461c1016534119d3df7044bfe23d002340c7313712bfc83cd 5.31kB / 5.31kB
=> => sha256:b5a0d5c14ba9ece1eedc5137c468d9a123372b0af2ed2c8c4446137730c90e5b 31.42MB / 31.42MB
=> => sha256:ccb4ba5bb726748e965e7730afcb6ab92eb14745c7bcad9861d77d81b2372cfe 32.46MB / 32.46MB
=> => sha256:4ece0626219de44070331daf1eff6932a03a31333a6f7f2d7b8b592a2e80d5b0 14.97MB / 14.97MB
=> => sha256:bdf2c62d9548601f6df118ad7ddf984e15a0aa258cc56b4b77945fa07a6978e7 155B / 155B
=> => sha256:d2e7695b08ad6a58b48f965619b84b2a3d47cbc0b15bb638aa29572bd097274 10.12MB / 10.12MB
=> => extracting sha256:b5a0d5c14ba9ece1eedc5137c468d9a123372b0af2ed2c8c4446137730c90e5b
=> => sha256:d41336b5e4674fbb04b625b2794c9f38eee0c549c63c3147df863043af106f9b 25.38MB / 25.38MB
=> => sha256:0f0fbcdde3825e1d159fbcc6d9b3910e65deadeaa651a2f6a4108c3d8234568b 180.94MB / 180.94MB
=> => sha256:449287723c1b5f4cc37c87dc59d8270f8a81bd5976cd1af3a454695a14e12bc 13.97MB / 13.97MB
=> => extracting sha256:4ece0626219de44070331daf1eff6932a03a31333a6f7f2d7b8b592a2e80d5b0
=> => extracting sha256:ccb4ba5bb726748e965e7730afcb6ab92eb14745c7bcad9861d77d81b2372cfe
=> CANCELED [worker stage-1 1/3] FROM mcr.microsoft.com/dotnet/runtime:7.0@sha256:b41a241da8624e65544dd83cbcc642152f10a751082d1ea1a912
=> => resolve mcr.microsoft.com/dotnet/runtime:7.0@sha256:b41a241da8624e65544dd83cbcc642152f10a751082d1ea1a912e238b8259533
=> => sha256:b41a241da8624e65544dd83cbcc642152f10a751082d1ea1a912e238b8259533 1.79kB / 1.79kB
=> => sha256:198cbf45efc0a2a5d64766ec547cb7bfa859b6e3a178543558a1beb358eb09bf 1.16kB / 1.16kB
=> => sha256:337945a71cfdb1635ab48144281b3575dd6726b6568343e01d9f711ab07dda5e 1.92kB / 1.92kB
=> => sha256:b5a0d5c14ba9ece1eedc5137c468d9a123372b0af2ed2c8c4446137730c90e5b 31.42MB / 31.42MB
=> => sha256:ccb4ba5bb726748e965e7730afcb6ab92eb14745c7bcad9861d77d81b2372cfe 32.46MB / 32.46MB
=> => sha256:4ece0626219de44070331daf1eff6932a03a31333a6f7f2d7b8b592a2e80d5b0 14.97MB / 14.97MB
=> => sha256:bdf2c62d9548601f6df118ad7ddf984e15a0aa258cc56b4b77945fa07a6978e7 155B / 155B
=> => extracting sha256:b5a0d5c14ba9ece1eedc5137c468d9a123372b0af2ed2c8c4446137730c90e5b
=> => extracting sha256:4ece0626219de44070331daf1eff6932a03a31333a6f7f2d7b8b592a2e80d5b0
=> => extracting sha256:ccb4ba5bb726748e965e7730afcb6ab92eb14745c7bcad9861d77d81b2372cfe
=> [worker internal] load build context
=> => transferring context: 7.07kB
=> [result 1/7] FROM docker.io/library/node:18-slim@sha256:fe687021c06383a2bc5eafa6db29b627ed28a55f6bdfbcea108f0c624b783c37
=> => resolve docker.io/library/node:18-slim@sha256:fe687021c06383a2bc5eafa6db29b627ed28a55f6bdfbcea108f0c624b783c37
=> => sha256:fe687021c06383a2bc5eafa6db29b627ed28a55f6bdfbcea108f0c624b783c37 1.21kB / 1.21kB
=> => sha256:8e04602828dd8c394c701f265c048f2a7cf9cbf112635ba26cec2d06936f17b 1.37kB / 1.37kB
=> => sha256:d3ccee7487840f2783532a37fca9e79c4d55fea3d5c3caf7c145596fcc457f8e 7.62kB / 7.62kB
=> => sha256:ebd7ac832a7e4bd07f5ad6f4bbcd6b1f5b02de7a8e07b40627142ef61a9b1b9d 3.36kB / 3.36kB
=> => sha256:9d5fc5b38df6ebdd70895cf78cd4e3de9aef9dc55ed5c79c2945066661a0e8e4 38.57MB / 38.57MB
=> => extracting sha256:ebd7ac832a7e4bd07f5ad6f4bbcd6b1f5b02de7a8e07b40627142ef61a9b1b9d
=> => sha256:e4cb19787d8cf84786456582936c59a7e23c541a29061d728df3f6c8923dd739 2.67MB / 2.67MB
=> => sha256:6f59eaf1fe6ede7c1fe4ad178ff96c351c02b84f6a0a7d4f38dcba13d4642298b 452B / 452B
=> => extracting sha256:9d5fc5b38df6ebdd70895cf78cd4e3de9aef9dc55ed5c79c2945066661a0e8e4
=> => extracting sha256:e4cb19787d8cf84786456582936c59a7e23c541a29061d728df3f6c8923dd739
=> => extracting sha256:6f59eaf1fe6ede7c1fe4ad178ff96c351c02b84f6a0a7d4f38dcba13d4642298b
=> [result internal] load build context
=> => transferring context: 278.07kB
=> [vote internal] load build context
```

Détails supplémentaires sur les services

Le service vote fournit une API pour voter sur des sujets. L'API accepte deux paramètres : le sujet sur lequel voter et le vote (oui ou non).

Le service worker traite les votes. Le service utilise la base de données pour stocker les votes.

Le service result affiche les résultats des votes. Le service utilise la base de données pour récupérer les résultats des votes.

Le service redis fournit un cache. Le service est utilisé pour stocker les résultats des votes afin d'accélérer les requêtes.

Utilisation d'un réseau Docker

```

services:
  vote:
    build:
      context: ./vote
      target: dev
    depends_on:
      redis:
        condition: service_healthy
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost"]
      interval: 15s
      timeout: 5s
      retries: 3
      start_period: 10s
    volumes:
      - ./vote:/usr/local/app
    ports:
      - "5003:80"
    networks:
      - humansbestfriend-network

  worker:
    build:
      context: ./worker
    depends_on:
      redis:
        condition: service_healthy
      db:
        condition: service_healthy
    networks:
      - humansbestfriend-network

  result:
    build: ./result
    entrypoint: nodemon --inspect=0.0.0.0 server.js
    depends_on:
      db:
        condition: service_healthy
    volumes:
      - ./result:/usr/local/app
    ports:
      - "5001:80"
      - "127.0.0.1:9229:9229"
    networks:
      - humansbestfriend-network

  redis:
    image: localhost:5000/redis:latest
    environment:
      REDIS_HOST: "db_redis"
    volumes:
      - "./healthchecks:/healthchecks"
    healthcheck:
      test: /healthchecks/redis.sh
      interval: "5s"
    networks:
      - humansbestfriend-network

  db:
    image: localhost:5000/postgres:15-alpine
    environment:
      POSTGRES_HOST: "db"
      POSTGRES_USER: "admin"
      POSTGRES_PASSWORD: "admin"
      POSTGRES_DBNAME: "db_tp"
    volumes:
      - "db-data:/var/lib/postgresql/data"
      - "./healthchecks:/healthchecks"
    healthcheck:
      test: /healthchecks/postgres.sh
      interval: "5s"

```

```

db:
  image: localhost:5000/postgres:15-alpine
  environment:
    POSTGRES_HOST: "db"
    POSTGRES_USER: "admin"
    POSTGRES_PASSWORD: "admin"
    POSTGRES_DBNAME: "db_tp"
  volumes:
    - "db-data:/var/lib/postgresql/data"
    - "./healthchecks:/healthchecks"
  healthcheck:
    test: /healthchecks/postgres.sh
    interval: "5s"
  networks:
    - humansbestfriend-network

seed:
  build: ./seed-data
  profiles: ["seed"]
  depends_on:
    vote:
      condition: service_healthy
  networks:
    - humansbestfriend-network
  restart: "no"

volumes:
  db-data:

networks:
  humansbestfriend-network:

```

L'utilisation d'un réseau Docker permet de connecter les services d'une application Docker multi-conteneur. Cela permet aux services de communiquer entre eux sans avoir besoin de connaître l'adresse IP ou le port de chaque service.

Dans le cas de l'application Docker multi-conteneur décrite dans ce rapport, les services vote, worker et result sont tous connectés au réseau humansbestfriend-network.

Pour créer le réseau, j'ai utilisé la commande suivante :

```
docker network create humansbestfriend-network
```

J'ai ensuite mis à jour le fichier de configuration Docker Compose pour connecter les services au réseau. La section networks du fichier de configuration a été mise à jour pour ajouter le nom du réseau :

networks:

humansbestfriend-network:

Les services vote, worker et result ont ensuite été connectés au réseau en spécifiant le nom du réseau dans la configuration Docker Compose. La section services du fichier de configuration a été mise à jour pour ajouter la propriété networks à chaque service :

vote:

image: localhost:5000/vote:latest

ports:

- "5003:80"

networks:

- humansbestfriend-network

worker:

image: localhost:5000/worker:latest

ports:

- "5002:80"

networks:

- humansbestfriend-network

result:

image: localhost:5000/result:latest

ports:

- "5001:80"

networks:

- humansbestfriend-network

Une fois le fichier de configuration Docker Compose mis à jour, j'ai lancé l'application Docker multi-conteneur à l'aide de la commande `docker-compose up`. L'application a été lancée et les services ont été connectés au réseau `humansbestfriend-network`.

L'utilisation d'un réseau Docker simplifie la communication entre les services d'une application Docker multi-conteneur. Cela permet aux développeurs de se concentrer sur le développement de l'application sans avoir à se soucier de la configuration de la communication entre les services.

Problème rencontré

```
> [worker stage-1 2/3] WORKDIR /app
> [vote dev 1/1] RUN pip install watchdog
> [vote] exporting to image
> => exporting layers
> => writing image sha256:ae36529f2a9db28d412009b1c4b70c98d864bcaa511729bf8525d101088070c9
> => naming to docker.io/library/humans-best-friend-vote
> [result 7/7] COPY . .
> [result] exporting to image
> => exporting layers
> => writing image sha256:c6eafb03d08caad5bbe2f7288d91e90130f2cc4e9bd591d798529cb42af7fe9d
> => naming to docker.io/library/humans-best-friend-result
-----
> [worker build 1/7] FROM mcr.microsoft.com/dotnet/sdk:7.0@sha256:4be8ff7cb847f9e7ea1ac194920b0a6d41956af89f934b61a2ce64dc8563471c:
-----
failed to solve: failed to register layer: write /usr/share/dotnet/sdk/7.0.404/Sdks/NuGet.Build.Tasks.Pack/Desktop/NuGet.Build.Tasks.Pack.dll: no space left on device
david@david:~/ynov-resources/2023/m2/dataeng/humans-best-friend$ ls
architecture.png  docker-compose.build.yml  docker-compose.yml  healthchecks  README.md  result  seed-data  vote  worker
david@david:~/ynov-resources/2023/m2/dataeng/humans-best-friend$
david@david:~/ynov-resources/2023/m2/dataeng/humans-best-friend$
david@david:~/ynov-resources/2023/m2/dataeng/humans-best-friend$
```

Lors de la création du réseau `humansbestfriend-network`, j'ai reçu l'erreur suivante :

Error: failed to register layer: write
/usr/share/dotnet/sdk/7.0.404/Sdks/NuGet.Build.Tasks.Pack/Desktop/NuGet.Build.Tasks.Pack.
dll: no space left on device

Solution

Le problème est dû au fait que le disque dur de l'hôte était plein. Pour résoudre le problème, j'ai libéré de l'espace sur le disque dur de l'hôte.

J'ai pu libérer de l'espace en supprimant des fichiers inutiles, tels que des fichiers temporaires ou des fichiers de sauvegarde. J'ai également pu libérer de l'espace en compressant des fichiers volumineux.

Une fois que j'ai libéré suffisamment d'espace sur le disque dur de l'hôte, j'ai pu créer le réseau humansbestfriend-network sans erreur.

```
db:
  image: localhost:5000/postgres:15-alpine
  environment:
    POSTGRES_HOST: "db"
    POSTGRES_USER: "admin"
    POSTGRES_PASSWORD: "admin"
    POSTGRES_DBNAME: "db_tp"
  volumes:
    - "db-data:/var/lib/postgresql/data"
    - "./healthchecks:/healthchecks"
  healthcheck:
    test: /healthchecks/postgres.sh
    interval: "5s"
  networks:
    - humansbestfriend-network

seed:
  build: ./seed-data
  profiles: ["seed"]
  depends_on:
    vote:
      condition: service_healthy
  networks:
    - humansbestfriend-network
  restart: "no"

volumes:
  db-data:

networks:
  humansbestfriend-network:
```

```
services:
  vote:
    build:
      context: ./vote
      target: dev
    depends_on:
      redis:
        condition: service_healthy
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost"]
      interval: 15s
      timeout: 5s
      retries: 3
      start_period: 10s
    volumes:
      - ./vote:/usr/local/app
    ports:
      - "5003:80"
    networks:
      - humansbestfriend-network

  worker:
    build:
      context: ./worker
    depends_on:
      redis:
        condition: service_healthy
      db:
        condition: service_healthy
    networks:
      - humansbestfriend-network

  result:
    build: ./result
    entrypoint: nodemon --inspect=0.0.0.0 server.js
    depends_on:
      db:
        condition: service_healthy
    volumes:
      - ./result:/usr/local/app
    ports:
      - "5001:80"
      - "127.0.0.1:9229:9229"
    networks:
      - humansbestfriend-network

  redis:
    image: localhost:5000/redis:latest
    environment:
      REDIS_HOST: "Db_redis"
    volumes:
      - ./healthchecks:/healthchecks
    healthcheck:
      test: /healthchecks/redis.sh
      interval: "5s"
    networks:
      - humansbestfriend-network

  db:
    image: localhost:5000/postgres:15-alpine
    environment:
      POSTGRES_HOST: "db"
      POSTGRES_USER: "admin"
      POSTGRES_PASSWORD: "admin"
      POSTGRES_DBNAME: "db_tp"
    volumes:
      - db-data:/var/lib/postgresql/data
      - ./healthchecks:/healthchecks
    healthcheck:
      test: /healthchecks/postgres.sh
      interval: "5s"
```

Test d'une application Docker multi-conteneur

```
david@david:~/ynov-resources/2023/m2/dataeng/humans-best-friend$ docker compose up -d
[+] Running 4/4
 ✓ Network humans-best-friend_humansbestfriend-network Created
 ✓ Volume "humans-best-friend_db-data" Created
 ✓ Container humans-best-friend-redis-1 Started
 ✓ Container humans-best-friend-db-1 Started
david@david:~/ynov-resources/2023/m2/dataeng/humans-best-friend$
```

Une fois l'application Docker multi-conteneur lancée, il est important de la tester pour s'assurer qu'elle fonctionne correctement.

Dans le cas de l'application Docker multi-conteneur décrite dans ce rapport, les services vote, worker et result sont tous accessibles via des URL distinctes.

Test du service vote

Pour tester le service vote, vous pouvez ouvrir un navigateur web et accéder à l'URL suivante :

<http://localhost:5003>

Vous devriez voir une page web qui vous permet de voter sur des sujets.

Test du service worker

Pour tester le service worker, vous pouvez utiliser la commande suivante :

```
curl -X POST http://localhost:5002 -d '{"subject": "Le meilleur ami de l'homme est-il le chien ?",
"vote": "oui"}
```

Cette commande envoie une requête POST au service worker avec un sujet et un vote.

Vous devriez recevoir une réponse JSON avec le résultat du vote.

Test du service result

Pour tester le service result, vous pouvez ouvrir un navigateur web et accéder à l'URL suivante :

`http://localhost:5001`

Vous devriez voir une page web qui affiche les résultats des votes.

Conclusion

Cette application Docker multi-conteneur peut être utilisée pour créer un système de vote simple. Les utilisateurs peuvent voter sur des sujets en appelant l'API du service vote. Les résultats des votes sont stockés dans la base de données et peuvent être affichés par le service result.

Voici quelques exemples de détails supplémentaires que vous pouvez ajouter à chacune des parties du rapport :