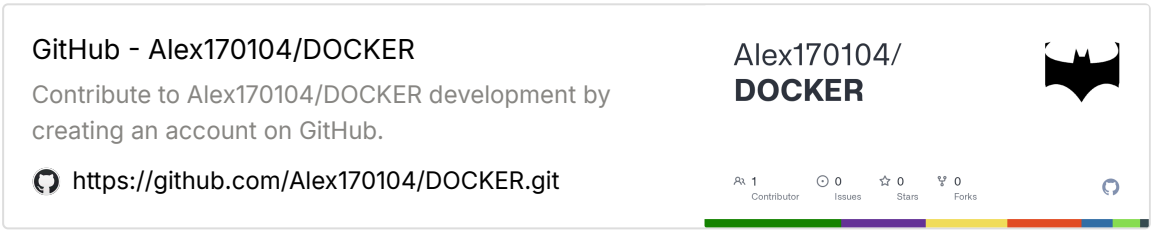
DOCKER

Prérequis

- Docker installé
- Docker Compose v2
- Accès a un terminal
- Pour la partie SWARM : 3 machines ou VM avec Docker
- Cloner le dépôt ou extraire l'archive `.zip`



DOCKER-main.zip

ETAPE 1 - APPROPRIATION DE L'APPLICATION

Ouvrir votre IDE et le projet voting-app.

Le projet contient 4 dossiers :

- vote/
- result/
- worker/
- run/

Analysons maintenant les fichiers contenus dans `run/`. Ce sont tous des fichiers `.bash`. On examine les étapes une par une pour comprendre leur fonction :

- Reset : supprime les dépendances NODE et les artefacts .NET
- Step 1 : l'application utilise REDIS sur le port `6379`
- Step 2 : l'application a un système de vote codé en Python. Pour lancer l'app Python, il faut exécuter `python app.py`
- Step 3 : l'application utilise POSTGRES sur le port `5432` avec plusieurs variables d'environnement
- Step 4 : l'application a un système de worker qui tourne sous .NET 7. Il utilise REDIS (step 1) et POSTGRES (step 3)
- Step 5 : l'application utilise Node.js pour un rendu visuel dynamique qui renvoie les résultats en temps réel

Dans la suite les scripts `.bash` ne sont plus utilisés. Leur logique a été entièrement reproduite via Dockerfiles et Docker Compose, ce qui permet un lancement unifié et reproductible de l'application.

ETAPE 2 - VERIFICATION DES PORTS

Vérifions maintenant si les ports des applications correspondent bien.

On cherche dans le fichier `vote/app.py` et on trouve la ligne :

```
app.run(host='0.0.0.0', port=8080, debug=True, threaded=True)
```

Le port utilisé est bien le port par défaut `8080`.

On cherche ensuite dans le fichier `result/` :

```
server.listen(port, function () {
    const port = server.address().port
    console.log("App running on port " + port)
})
```

La valeur du port est `8888`.

ETAPE 3 - dockerfile pour `vote/`

dans le sous répertoire `vote/` créer un fichier dockerfile

dans lequel on écrit le contenu suivant :

```
FROM python:alpine3.23

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 8080

CMD ["python", "app.py"]
```

la version `alpine3.23` est la version la plus récente de l'image python

> python - Official Image | Docker Hub
> Python is an interpreted, interactive, object-oriented, open-source programming language.
> https://hub.docker.com/_/python

et comme vu précédemment on utilise bien le port 8080

ETAPE 4 - dockerfile pour `result/`

dans le sous répertoire `result/` on créer un fichier dockerfile

dans lequel on écrit le contenu suivant :

```
FROM node:lts-alpine3.23
```

```
WORKDIR /app

COPY package*.json ./
RUN npm install

COPY . .

EXPOSE 8888

CMD ["npm", "start"]
```

la version `node:lts-alpine3.23` est le derrniere version pour node

> node - Official Image | Docker Hub
> Node.js is a JavaScript-based platform for server-side and networking applications.
> https://hub.docker.com/_/node

et comme vu précédemment on utilise bien le port 8888

ETAPE 5 - dockerfile pour `worker/`

dans le sous répertoire `worker/` on créer un fichier dockerfile

on lui ajoute le contenu suivant :

```
FROM mcr.microsoft.com/dotnet/sdk:7.0 AS build
WORKDIR /src

COPY . .
RUN dotnet restore
RUN dotnet publish -c Release --no-restore -o /app/publish

FROM mcr.microsoft.com/dotnet/runtime:7.0
WORKDIR /app
COPY --from=build /app/publish .
```

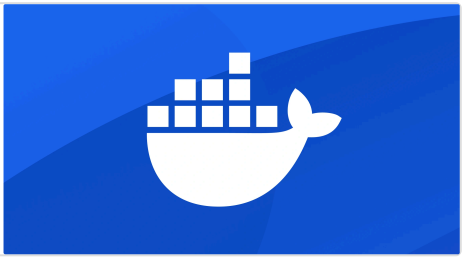
```
CMD ["dotnet", "Worker.dll"]
```

on recupere une image “builder” de notre worker puis on la met dans une image “runtime” qui est plus legere. C’est une bonne pratique de .NET et docker

Best practices

Hints, tips and guidelines for writing clean, reliable Dockerfiles

https://docs.docker.com/build/building/best-practices/#use-multi-stage-builds



ETAPE 6 - Docker Compose

à la racine du projet on créer un fichier `docker-compose.yml`

on s’occupe de configurer chaque service : `redis` , `db` , `vote` , `worker` , `result` .

- `networks`

```
networks:
    frontend:
    backend:
```

- `volumes`

```
volumes:
  postgres-data:
  redis-data:
```

Tout ce qui suis est dans la section :

```
services:
```

- `redis`

```
redis:
 image: redis:8.4.0-alpine
 volumes:
  - redis-data:/data
 networks:
  - backend
 healthcheck:
  test: [ "CMD", "redis-cli", "ping" ]
  interval: 5s
  timeout: 3s
  retries: 5
 restart: always
 ports:
  - "6379:6379"
```

- `db`

```
db:
image: postgres:18.1-alpine
environment:
  POSTGRES_USER: postgres
  POSTGRES_PASSWORD: postgres
  POSTGRES_DB: postgres
ports:
  - "5432:5432"
volumes:
  - postgres-data:/var/lib/postgresql/data
networks:
  - backend
healthcheck:
    test: ["CMD-SHELL", "pg_isready -U postgres"]
    interval: 10s
    timeout: 5s
    retries: 5
restart: always
```

- `vote`

```
vote:
  build: ./vote
  ports:
    - "8080:8080"
  depends_on:
    redis:
      condition: service_healthy
  networks:
    - frontend
    - backend
  environment:
    REDIS_HOST: redis
    REDIS_PORT: 6379
  healthcheck:
    test: [ "CMD", "curl", "-f", "http://localhost:8080" ]
    interval: 10s
    timeout: 3s
    retries: 5
  restart: always
```

- `worker`

```
worker:
  build: ./worker
  depends_on:
    redis:
      condition: service_healthy
    db:
      condition: service_healthy
  networks:
    - backend
  restart: always
  environment:
    REDIS_HOST: redis
    REDIS_PORT: 6379
    POSTGRES_HOST: db
    POSTGRES_USER: postgres
```

```
    POSTGRES_PASSWORD: postgres
    POSTGRES_DB: postgres
```

- result

```
result:
 build: ./result
 ports:
  - "8888:8888"
 depends_on:
  db:
   condition: service_healthy
 networks:
  - frontend
  - backend
 environment:
  POSTGRES_HOST: db
  POSTGRES_PORT: 5432
  POSTGRES_USER: postgres
  POSTGRES_PASSWORD: postgres
  POSTGRES_DB: postgres
 healthcheck:
  test: ["CMD", "curl", "-f", "http://localhost:8888"]
  interval: 10s
  timeout: 3s
  retries: 5
 restart: always
```

fichier complet :

docker-compose.yml

Explications

chaque service est composé de propriétés. Chaque propriétés ont des specificités.

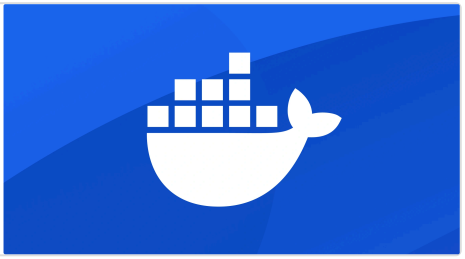
`depends_on` : permet de ne pas lancer le conteneur si et seulement si le service dépendant est sain et opérationnel

`healthcheck` : permet de vérifier si l'app est réellement UP

Services

Explore all the attributes the services top-level element can have.

https://docs.docker.com/reference/compose-file/services/#healthcheck



`environment` : permet de ne pas avoir de valeur en dur dans le code et donc de toutes les externalisées

`volumes` : permet la persistance (les votes ne seront pas perdus au restart de l'app)

`networks` : permet l'isolation par réseau

Networking

How Docker Compose sets up networking between containers

https://docs.docker.com/compose/how-tos/networking/

Ajout dans le code

Le code doit être adapté afin d'utiliser les variables d'environnement définies dans le fichier `docker-compose`, plutôt que des valeurs configurées en dur.

```
// /result/server.js

// à remplacer dans le code
connectionString: process.env.DATABASE_URL ||
      `postgres://${process.env.POSTGRES_USER}:${process.env.POSTGRE
```

```
S_PASSWORD}@${process.env.POSTGRES_HOST}:${process.env.POSTGRES_PORT}/${process.env.POSTGRES_DB}`,
```

```
#/vote/app.py

# remplacer cette ligne
Flask.redis = Redis(host="localhost", db=0, socket_timeout=5)

# par celles la
redis_host = os.getenv('REDIS_HOST', 'localhost')
redis_port = int(os.getenv('REDIS_PORT', 6379))
Flask.redis = Redis(host=redis_host, port=redis_port, db=0, socket_timeout=5)
```

```
// /worker/Program.cs

// replacer ces lignes la
var pgsql = OpenDbConnection("Server=localhost;Username=postgres;Password=postgres;");
var redisConn = OpenRedisConnection("localhost");
//...
redisConn = OpenRedisConnection("localhost");
//...
var pgsql = OpenDbConnection("Server=localhost;Username=postgres;Password=postgres;");


// par celles la
var pgHost = Environment.GetEnvironmentVariable("POSTGRES_HOST");
var pgUser = Environment.GetEnvironmentVariable("POSTGRES_USER");
var pgPassword = Environment.GetEnvironmentVariable("POSTGRES_PASSWORD");
var pgPort = Environment.GetEnvironmentVariable("POSTGRES_PORT");
var pgDatabase = Environment.GetEnvironmentVariable("POSTGRES_DB");
var redisHost = Environment.GetEnvironmentVariable("REDIS_HOST");
var pgConnStr = $"Host={pgHost};Port={pgPort};Username={pgUser};Password={pgPassword};Database={pgDatabase};";
```

```
var pgsql = OpenDbConnection(pgConnStr);
var redisConn = OpenRedisConnection(redisHost);
//...
redisConn = OpenRedisConnection(redisHost);
//...
pgsql = OpenDbConnection(pgConnStr);
```

Déploiement

pour UP le serveur :

```
docker compose up --build
```

pour DOWN le server :

```
docker compose down
```

pour accéder a la partie “vote” :

```
http://localhost:8080
```

pour accéder a la partie “result” :

```
http://localhost:8888
```

Le fichier Docker Compose respecte les bonnes pratiques présentées en cours : gestion explicite des dépendances via `depends_on` et `healthcheck`, isolation réseau, persistance des données par volumes, redémarrage automatique des conteneurs et configuration par variables d’environnement.

La table `votes` est créée automatiquement par le service worker lors de sa première connexion à PostgreSQL.

ETAPE 7 - Docker Swarm

Mise en place du cluster

on va créer 1 nœud **manager** et 2 nœuds **worker**

(3 machines/VM comme dit précédemment dans Prérequis)

manager :

sur la machine **manager** lancer la commande :

```
docker swarm init
```

le nœud devient **manager** et docker donne une commande `docker swarm join`

ajouter des workers :

sur chacune des deux machines **workers** lancer la commande :

```
docker swarm join --token <TOKEN> <IP_MANAGER>:2377
```

vérifier sur la machine **manager** que les nœuds ont bien été ajoutés :

```
docker node ls
```

vous devez voir :

- 1 **manager** (Leader)
- 2 **workers** (Ready)

On peut aussi tester avec docker desktop c'est simple et efficace mais la finalité est de mettre en place un cluster sur plusieurs machine !

sur docker desktop :

```
docker swarm init
docker build -t vote:latest ./vote
docker build -t worker:latest ./worker
docker build -t result:latest ./result
```

```
docker stack deploy -c docker-compose.swarm.yml voting-app

#pour verifier
docker stack services voting-app
```

Adapter le `docker-compose.yml` pour Docker Swarm

il y a quelques points qu'il faut modifiés dans le `docker-compose.yml` car il y a des differences entre compose et swarm :

- ajout de la clé `deploy`
- on ne peut plus build directement

En Swarm on ne déploie plus du code mais bien des images directement.

on prépare donc les images en amont sur le **manager** :

```
docker build -t vote:latest ./vote
docker build -t result:latest ./result
docker build -t worker:latest ./worker
```

et on réadapte le `docker-compose.yml`

(tout est déjà fait dans le `docker-compose.swarm.yml`)

- `networks`

```
networks:
    frontend:
    backend:
```

- `volumes`

```
volumes:
  postgres-data:
```

```
  redis-data:
```

Tout ce qui suis est dans la section :

```
services:
```

- `redis`

```
 redis:
  image: redis:8.4.0-alpine
  volumes:
    - redis-data:/data
  networks:
    - backend
  healthcheck:
      test: [ "CMD", "redis-cli", "ping" ]
      interval: 5s
      timeout: 3s
      retries: 5
    deploy:
       restart_policy:
         condition: on-failure
  ports:
    - "6379:6379"
```

- `db`

```
  db:
  image: postgres:18.1-alpine
  environment:
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: postgres
    POSTGRES_DB: postgres
  ports:
    - "5432:5432"
  volumes:
    - postgres-data:/var/lib/postgresql/data
```

```
    networks:
      - backend
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 10s
      timeout: 5s
      retries: 5
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure
```

- `vote`

```
  vote:
    image: vote:latest
    ports:
      - "8080:8080"
    networks:
      - frontend
      - backend
    environment:
      REDIS_HOST: redis
      REDIS_PORT: 6379
    deploy:
      replicas: 2
      restart_policy:
        condition: on-failure
```

- `worker`

```
  worker:
    image: worker:latest
    depends_on:
      redis:
        condition: service_healthy
      db:
```

```
      condition: service_healthy
  networks:
    - backend
  deploy:
    replicas: 1
    restart_policy:
      condition: on-failure
  environment:
    REDIS_HOST: redis
    REDIS_PORT: 6379
    POSTGRES_HOST: db
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: postgres
    POSTGRES_DB: postgres
```

- `result`

```
  result:
    image: result:latest
    ports:
      - "8888:8888"
    depends_on:
      db:
        condition: service_healthy
    networks:
      - frontend
      - backend
    healthcheck:
      test: [ "CMD", "curl", "-f", "http://localhost:8888" ]
      interval: 10s
      timeout: 3s
      retries: 5
    deploy:
      replicas: 2
      restart_policy:
        condition: on-failure
```

fichier complet :

docker-compose.swarm.yml

on remarque que `restart: always` est remplacé par

```
deploy:
        restart_policy:
          condition: on-failure
```

En cas de panne d'un nœud **worker**, Docker Swarm redistribue automatiquement les réplicas des services `vote` et `result` sur les nœuds disponibles. Grâce à la configuration `replicas: 2`, les interfaces web restent accessibles même si un nœud quitte le cluster.