ETAPE 1 CRÉER UNE DB POSTGRES:

Nous allons créer un dossier docker, dans ce dossier nous allons créer un dossier nommé db.

Ensuite nous allons créer un fichier nommé dockerfile et ajouter ce code:

```
FROM postgres

ENV POSTGRES_USER=emre
ENV POSTGRES_PASSWORD=mdp
ENV POSTGRES_DB=db_emre

COPY ./init.sql /docker-entrypoint-initdb.d/
```

Ici on copie notre fichier de création de table et d'insertions dans le répertoire docker

Nous allons créer ce fichier init.sql où nous allons créer notre table et y insérer une valeur

```
CREATE TABLE message(
    id SERIAL PRIMARY KEY,
    nom VARCHAR(200)
);

INSERT INTO message (nom)
VALUES ('Hello World');
```

après avoir creer les deux fichier nous allons créer l'image :

docker build -t emre .

nous allons run le conteneur:
docker run -d emre

un message comme ca devra s'afficher:

```
PS C:\docker\db> docker run -d emre
2494af5f9026cb47f3d578205799c0f9b2004dfe80f46dad55f39c2f633cdf6a
```

docker ps (pour voir l'id du contenair)
on va utiliser l'id sur la commande suivante

docker exec -it (id du conteneur) /bin/bash
/bin/bash pour aller sur le terminal de psql

psql -U emre -d db_emre -h localhost

pour ce connecter à notre db

faire \dt pour voir toute les tables disponible

```
db_emre=# \dt
        List of relations
 Schema |  Name   | Type  | Owner
--------+---------+-------+-------
 public | message | table | emre
(1 row)
```

faire un select * from message; pour voir le contenu de notre table

```
db_emre=# select * from message
;
 id |     nom
----+-------------
  2 | Hello World
(1 row)
```

ETAPE 2 CRÉER API FLASK:

Une fois tout cela est fait nous allons créer un dossier back qui contiendra un fichier app.py (notre fichier flask), un fichier requirements.txt pour installer toutes les extension nécessaire pour le bon fonctionnement de notre api flask et un fichier dockerfile pour lancer notre fichier app.py.
Notre api va nous permettre de récupérer le contenu de notre db

Ce Dockerfile créer un répertoire de travail , copie le fichier requirements.txt dans ce répertoire, il installe toutes les dépendances pour le bon fonctionnement de notre api, et pour finir il va exécuter l'application flask

```
FROM python:3.8-slim-buster

WORKDIR /python-docker

COPY requirements.txt requirements.txt
RUN pip3 install -r requirements.txt

COPY . .

CMD [ "python3", "-m" , "flask", "run", "--host=0.0.0.0"]
```

voici le fichier app.py qui va se connecter à notre db et va retourner une valeur au format json à la route /hello (localhost:5000/hello), la route par défaut nous renvoi vers la route /hello :

```python
from flask import Flask, jsonify, redirect, url_for
import psycopg2

app = Flask(__name__)

def get_db_connection():
    conn = psycopg2.connect(
        host="localhost",
        database="db_emre",
        user="emre",
        password="mdpEmre"
    )
    return conn

@app.route('/')
def index():
    return redirect(url_for('hello'))

@app.route('/hello')
def hello():
    conn = get_db_connection()
    cur = conn.cursor()

    cur.execute('SELECT nom FROM message LIMIT 1;')
    name = cur.fetchone()

    cur.close()
    conn.close()

    if conn is None:
        return jsonify({"message": "No data found"}), 404
    else:
        return jsonify({"message": conn})

if __name__ == '__main__':
    app.run()
```

voici notre fichier requirements.txt

```
flask
psycopg2-binary
```

ETAPE 2.2: sécurisé, créer un volume et faire un docker-composer:

à l'étape d'avant nous étions obligé de build , run chaque dockerfile , ce qui est assez relou et long surtout , d'autant plus que le mot de passe de notre db était visible par tous ce qui est problématique, nous allons aussi créer un volume pour que notre données de notre db soit enregistrer, sécurisé et gérer par docker.
Dans cette étape nous allons créer un docker-compose (format yaml) et faire quelque modification sur nos fichier

Dans un premier temps nous allons crée un fichier nommé postgres-passwd.txt où nous allons mettre notre mdp de notre db

voici à quoi va ressembler notre docker-compose:

```yaml
services:
  db:
    build: ./db
    volumes:
      - db_data:/var/lib/postgresql/data
    environment:
      POSTGRES_PASSWORD_FILE: /run/secrets/db_password
    secrets:
      - db_password

  back:
    build: ./back
    ports:
      - "5000:5000"
    depends_on:
      - db
    secrets:
      - db_password

secrets:
  db_password:
    file: ./secrets/postgres-passwd.txt

volumes:
  db_data:
```

ici nous avons créé deux service (db et back), dans la db nous avons fait un build pour que quand on run ce docker-compose ca nous build le contenair , ensuite nous avons un environnement qui donne le mdp de notre db qui se trouve dans la variable secrets et qui appel notre fichier postgres-passwd.txt que nous avons créé juste avant.
Dans notre service back avons mis un port (port par défaut de flask) et avons mis une dépendance qui permet au back d'être lancé uniquement si la db c'est correctement lancer

voici le dockerfile dans le document db:

```dockerfile
from postgres

ENV POSTGRES_USER=emre
ENV POSTGRES_DB=db_emre

COPY hello.sql /docker-entrypoint-initdb.d/
```

nous avons plus besoin de définir en gros le mdp car nous l'avons mis dans notre environnement de notre docker-compose

voici à quoi ressemble notre app.py:

```python
from flask import Flask, jsonify, redirect, url_for
import psycopg2

app = Flask(__name__)


def get_db_connection():

    with open('/run/secrets/db_password', 'r') as file:
        password = file.read().strip()

    conn = psycopg2.connect(
        host="db",
        database="db_emre",
        user="emre",
        password=password
    )
    return conn

@app.route('/')
def index():
    return redirect(url_for('hello'))

@app.route('/hello')
def hello():
    conn = get_db_connection()
    cur = conn.cursor()

    cur.execute('SELECT nom FROM message LIMIT 1;')
    name = cur.fetchone()

    cur.close()
    conn.close()

    if conn is None:
        return jsonify({"message": "No data found"}), 404
    else:
        return jsonify({"message": conn})

if __name__ == '__main__':
    app.run()
```

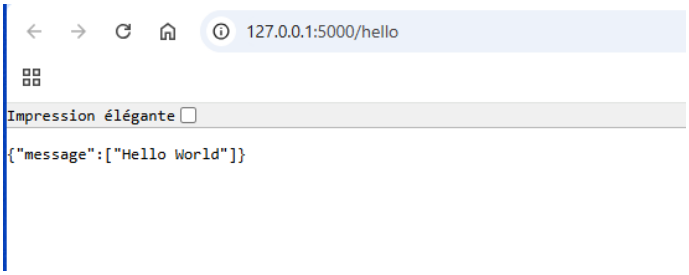
Grâce au secrets dans notre docker-compose nous avons pu récupérer le mdp sans qu'il soit visible et le host est maintenant db.

maintenant si on fait docker-compose up celà va nous lancer tous les contenair que nous avons paramétré

```
db-1    | 2024-10-30 12:48:43.023 UTC [29] LOG:  database system was shut down at 2024-10-30 12:48:30 UTC
db-1    | 2024-10-30 12:48:43.035 UTC [1] LOG:  database system is ready to accept connections
back-1  |  * Debug mode: off
back-1  | WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
back-1  |  * Running on all addresses (0.0.0.0)
back-1  |  * Running on http://127.0.0.1:5000
back-1  |  * Running on http://172.18.0.3:5000
back-1  | Press CTRL+C to quit
back-1  | 172.18.0.1 - - [30/Oct/2024 12:49:08] "GET / HTTP/1.1" 302 -
back-1  | 172.18.0.1 - - [30/Oct/2024 12:49:08] "GET /hello HTTP/1.1" 200 -
```

Si vous faite des modification et que vous voulez que ca le prenne en compte il faut faire un docker-compose up --build

Voici le résultat (localhost:5000) :



ETAPE 3 faire le front:

Nous allons créer un dossier front nous allons faire un dockerfile, un fichier pour la configuration du proxy et un dossier src où il y aura un fichier html et un dossier js qui contiendra notre javascript

voici le dockerfile:

```
FROM httpd:2.4

COPY ./src /usr/local/apache2/htdocs/
COPY ./myapp.conf /usr/local/apache2/conf/extra/myapp.conf

RUN sed -i '/LoadModule proxy_module/s/^#//g' /usr/local/apache2/conf/httpd.conf \
    && sed -i '/LoadModule proxy_http_module/s/^#//g' /usr/local/apache2/conf/httpd.conf

RUN echo "Include conf/extra/myapp.conf" >> /usr/local/apache2/conf/httpd.conf
```

on va utiliser l'image apache et y mettre le fichier myapp.conf qu'on à créé.

voici le fichier myapp.conf :

```
<VirtualHost *:80>
    ServerName localhost

    Alias "/hello" "/usr/local/apache2/htdocs/index.html"

    ProxyPass "/api/hello" "http://back:5000/hello"
    ProxyPassReverse "/api/hello" "http://back:5000/hello"

    <Directory "/usr/local/apache2/htdocs/">
        Options -Indexes +FollowSymLinks
        AllowOverride All
        Require all granted
    </Directory>

    ErrorLog /usr/local/apache2/logs/error.log
    CustomLog /usr/local/apache2/logs/access.log combined
</VirtualHost>
```

voici le fichier html qui va attendre une réponse du javascript:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Hello Page</title>
    <script defer src="/js/script.js"></script>
</head>
<body>
    <h1>Message du back :</h1>
    <p id="Message"></p>
</body>
</html>
```

voici le fichier javascript:

```
document.addEventListener("DOMContentLoaded", async function() {
    try {
        const response = await fetch('/api/hello');

        if (!response.ok) {
            throw new Error('Network response was not ok');
        }

        const data = await response.json();
        document.getElementById('Message').innerText = data.message;

    } catch (error) {
        console.error('Error:', error);
        document.getElementById('Message').innerText = 'Error: Could not fetch data';
    }
});
```

Si nous avions pas de proxy nous serions obligé d'importer CORS (cross-origin resource sharing) sur notre api flask pour que ca fonctionne correctement avec notre front

Une fois tout cela est fait nous allons ajouter cela à notre docker-compose

```yaml
services:
  db:
    build: ./db
    volumes:
      - db_data:/var/lib/postgresql/data
    environment:
      POSTGRES_PASSWORD_FILE: /run/secrets/db_password
    secrets:
      - db_password
    networks:
      - db_network

  back:
    build: ./back
    ports:
      - "5000:5000"
    depends_on:
      - db
    secrets:
      - db_password
    networks:
      - db_network
      - frontend_network

  front:
    build: ./front
    ports:
      - "8080:80"
    depends_on:
      - back
    networks:
      - frontend_network

networks:
  frontend_network:
    driver: bridge
  db_network:
    driver: bridge

secrets:
  db_password:
    file: ./secrets/postgres-passwd.txt

volumes:
  db_data:
```

nous avons ajouté notre service front, ajouter la dépendance sur le back pour que le front ne se lance si seulement le back c'est lancer correctement et nous avons fait un networks pour que le front ne communique uniquement avec le back et le back avec les deux, cela permet une meilleur sécurité

Voici le résultat final (localhost:8080) :