Dockerisation d'une Application python

Dans ce document on va créer une application python très simple et voir comment on peut la dockeriser.

Il s'agit d'une API qui affiche un message. On utilise `Flask`.

Nous verrons ensuite

- comment limiter la taille de l'image avec `.dockerignore`
- lier le contenu du container au repertoire courant lorsque l'on développe en continu.
- l'utilisation de VOLUME pour persister les données au delà de la vie du container.

Step 1: Créer l'application python

créez un repertoire `flask-api`

```bash
mkdir flask-api
cd flask-api
```

Copier le code suivant dans un fichier `app.py`.

```python
# app.py
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return "Hello, World!"

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Il s'agit d'une simple API Flask qui affiche un message sur la route `/`.

Step 2: Create a Requirements File

Il nous faut un fichier de `requirements.txt` pour installer automatiquement la librairie Flask.

Donc dans un fichier intitulé `requirements.txt`, ajoutez la ligne:

```
Flask==3.0.3
```

Step 3: Ecrire le Dockerfile

On part cette fois-ci d'une image `python:3.9-slim` qui contient déjà python en version 3.9.

Le Dockerfile va implémenter les étapes suivantes

- image de base
- créer un repertoire de travail `/app`
- copier tous les fichiers locaux dans `/app`
- installer les librairies python avec `pip install`
- Exposer le port 5000 du container
- Définir une variable d'environnement `FLASK_APP=app.py`
- Enfin, executer le fichier `app.py` avec `python`

Voici le `Dockerfile`:

```
# Dockerfile

# Step 1: Use an official Python runtime as a parent image
FROM python:3.9-slim

# Step 2: Set the working directory in the container
WORKDIR /app

# Step 3: Copy the current directory contents into the container at /app
COPY . /app

# Step 4: Install any dependencies specified in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Step 5: Make port 5000 available to the world outside this container
EXPOSE 5000

# Step 6: Define environment variable
ENV FLASK_APP=app.py

# Step 7: Run the application
CMD ["python", "app.py"]
```

Explications

1. **Image de base (`FROM python:3.9-slim`)** :
   - On utilise une image Python légère (slim) basée sur la version 3.9.
2. **Répertoire de travail (`WORKDIR /app`)** :
   - Le répertoire de travail à l'intérieur du container est `/app`
3. **Copier les fichiers (`COPY . /app`)** :
   - On copie tous les fichiers du répertoire de l'application dans le répertoire `/app` à l'intérieur du container.
4. **Installer les dépendances (`RUN pip install`)** :
   - La commande `RUN` installe les dépendances listées dans le fichier `requirements.txt` à l'intérieur du conteneur.
5. **Exposer le port (`EXPOSE 5000`)** :
   - Le conteneur écoutera sur le port 5000, où Flask s'exécute par défaut.

6. **Variable d'environnement (`ENV FLASK_APP`)** :
    - On définit une variable d'environnement pour que Flask sache quel fichier utiliser comme application principale.
7. **Lancer l'application (`CMD`)** :
    - La commande `CMD` spécifie la commande à exécuter lorsque le conteneur démarre. Dans ce cas, elle exécute le script Python pour lancer l'application Flask.

Step 4: Build and Run the Docker Image

Pour builder l'image et lui donner le nom `flask-api` :

```bash
docker build -t flask-api .
```

On crée et on lance le container avec

```bash
docker run  -d -p 5001:5000 flask-api
```

Notez le mode détaché avec le flag `-d` . qui vous redonne la main dans le terminal. Et le binding entre le port interne 5000 et le port externe 5001.

L'application Flask est maintenant accessible sur `http://localhost:5001` et vous y verrez un tres beau "Hello World"

.gitignore

Il est de bonne pratique de limiter la taille de l'image en n'y copiant que les fichiers strictement nécessaires à son fonctionnement.

Pour éviter de devoir les spécifier un à un ou répertoire par répertoire, on utilise l'instruction `COPY . /app` pour copier tous les fichiers du repertoire courant dans l'image.

On peut exclure certains fichiers en les listant dans un fichier appelé `.dockerignore` . On y retrouvera une bonne partie des fichiers mentionnés dans `.gitignore` .

```
passphrase.txt
.env
logs/
.git
Dockerfile
```

Notez que l'on peut aussi ajouter `Dockerfile` dans `.dockerignore`.

Evolution du code

L'application est vraiment basique.

On veut la modifier pour que l'on puisse passer une variable `text` dans l'url et afficher sa valeur.

On remplace dans le fichier `app.py`, la function `hello()` par celle ci:

```python
def hello():
    # Get the 'text' query parameter from the URL, default to "Hello Wo
    message = request.args.get('text', 'Hello World')
    return message
```

et il faut aussi importer `request`.

```python
from flask import Flask, request
```

Pour que les **changements de code soient pris en compte** il faut builder l'image à nouveau et relancer le container.

MAIS: A chaque fois cela crée une nouvelle image et un nouveau container.

Ce n'est pas pratique!

Mounts

> Pour éviter de reconstruire l'image Docker à chaque modification du code, il faut monter le répertoire de code local dans le conteneur Docker en cours d'exécution.

De cette manière, les modifications de code sont immédiatement reflétées à l'intérieur du conteneur, sans avoir besoin de reconstruire l'image.

Pour lier un repertoire sur le host à un repertoire dans le container, on parle de **Bind**

Mounts.

On peut associer les répertoires du host au container au moment de lancer le container avec le flag `-v`

```Bash
docker run -d -v /path/on/host:/folder_in_container image_name
```

Dans notre cas, on va associer le répertoire courant `$(pwd)` où se trouve le code avec le répertoire `/app` dans le container

```Bash
docker run -d -v  $(pwd):/app -p 5001:5000 flask-api
```

`$(pwd)` correspond au répertoire courant sur le host, c'est équivalent à `./` mais avec le full path.

Volume

La commande VOLUME dans un Dockerfile permet elle de déclarer un espace de stockage persistent au dela du cycle de vie du container. Attention: **l'espace défini par `VOLUME` réside dans Docker et non pas sur le host**.

L'idée est de permettre de stocker des données meme quand le container n'existe plus ou est stoppé.

et de partager les données entre plusieurs containers.

- un container écrit les données, un autre les lit: cache, microservices, partage de logs, ...

```
VOLUME /shared-data
```

- persistence des données dans une base de données

```
VOLUME /var/lib/postgresql/data
```

- logs, backup, cache, ...

```
VOLUME /app/logs
VOLUME /app/data
VOLUME /app/cache
```

Accéder au contenu du Volume

Pour accéder au données contenues dans le VOLUME,

Lorsque que l'on RUN le container on peut monter le VOLUME sur un repertoire host en utilisant le flag `-v`

par exemple

```Bash
docker run -d --name my_container -v my_logs:/var/log/app my_image
```

- `my_logs` : This is the named volume that will persist data.
- `/var/log/app` : Le repertoire où sont stockés les logs a l'interieur du container

Meme quand le container n'existe plus

Si le container est stoppé ou n'existe plus on peut quand meme accéder au données déclarées dans le VOLUME avec un container temporaire créé juste pour accéder au VOLUME

```Bash
docker run --rm -v my_logs:/logs busybox ls /logs
```

This command does the following:

- `--rm` : Automatically removes the container after execution.
- `-v my_logs:/logs` : Mounts the `my_logs` volume to `/logs` in the container.
- `busybox ls /logs` : Runs the `ls` command in a temporary `busybox` container to list the contents of the volume.

You can also explore and copy data by starting an interactive session:

```Bash
docker run --rm -it -v my_logs:/logs busybox sh
```

Inside the `sh` shell, you can explore and copy files from the `/logs` directory.

Les volumes ainsi déclarée sont visibles et manageable par la commande

```Bash
docker volume ls
```

VOLUME vs `-v`

Pour des mise à jour en continu pendant le développement, il faut utiliser `-v` quand on lance le container.

Il ne faut utiliser `VOlUME` dans le Dockerfile que si l'on veut déclarer un espace de stockage persistent qui est géré par Docker. C'est utile pour les données persistantes comme pour les bases de données.

Final

- supprimer le container après l'avoir stoppé
- ajouter un fichier `.dockerignore` avec au moins Dockerfile
- relancer le container en montant le rep local au rep container /app
- Modifiez la fonction `hello()` dans `app.py` pour admettre un message comme variable dans l'url
- vérifiez que ça marche en accédant a `localhost:5001?text="Bonjour"`
- le message `Bonjour` doit apparaitre sur la page

ensuite:

- taggez votre image avec votre repo docker hub et le tag `flask-api-01`
- publiez (push) l'image `flask-api-01` sur votre repo Docker Hub
- et renseignez la bonne case dans le formulaire https://docs.google.com/spreadsheets/d/17jjXXb-bJhaosVoTLk1c_JKSaDfPklLyXYP4lUPErQk/edit?gid=0#gid=0

Lectures

- https://blog.logrocket.com/docker-volumes-vs-bind-mounts/