Architecture micro-service

TP 5 - Premiers micro-services - La météo, ça coûte cher en bande passante !

Philippe Roussille

3iL INGENIEURS
</les sciences informatiques>

1 État des lieux

1.1 Présentation du cadre

> *"C'est quoi ce flood sur open-meteo ?!" - Ginette "Désolé, j'ai fait une boucle infinie pour voir s'il faisait toujours beau à Rodez… et puis mon cactus a éternué, et donc je vais devoir m'absenter du bureau, je vous laisse gérer." - Philibert Roquart, stagiaire à temps partiel "efficace"*

Après plusieurs jours d'utilisation, certains ont remarqué que le microservice météo était interrogé **en boucle** par des scripts naïfs. Résultat : des appels redondants, une saturation du service externe… et une menace de ban de la part d'Open-Meteo.

1.2 Le problème

Le service météo actuel interroge *systématiquement* l'API distante à chaque requête, sans rien retenir. Pas malin, surtout pour Rodez, dont le climat change peu. Il est temps d'ajouter une **forme de persistance** : retenir les dernières infos pour ne pas toujours les redemander.

1.3 Ben c'est simple, on part sur des fichiers !

On pourrait, bien sûr, stocker les réponses dans un fichier JSON. Mais cela pose vite des problèmes :

- Fichiers à ouvrir/fermer proprement
- Accès concurrents risqués
- Difficile de filtrer ou d'organiser les données efficacement

Or, vous avez déjà vu un **outil bien plus adapté** : une **base de données**.

1.4 Deux outils outils : MySQL + SQLAlchemy

Grâce à la dockerisation du TP précédent, on peut maintenant facilement ajouter une **base MySQL via docker-compose**. Et pour éviter de manipuler des requêtes SQL à la main, on utilisera un **ORM** : un outil qui permet de manipuler les objets Python comme des enregistrements de base de données.

Ce n'est pas toujours nécessaire (ici, c'est même très artificiel) mais c'est **un passage obligé** pour comprendre les architectures modernes.

2 Le matériel de base pour flanquer une BD

2.1 Fichier `docker-compose.yml` + `.env`

Plutôt que de coder en dur les identifiants dans le `docker-compose.yml`, nous allons utiliser un fichier `.env` pour centraliser la configuration.

2.1.1 Contenu de `.env`

```
DB_HOST=db
DB_PORT=3306
DB_USER=weather
DB_PASSWORD=weatherpass
DB_NAME=weather_db
MYSQL_ROOT_PASSWORD=rootpass
```

2.1.2 Fichier `docker-compose.yml`

```yaml
version: '3.8'

services:
  weather:
    build: .
    ports:
      - "5000:5000"
    env_file:
      - .env
    depends_on:
      - db

  db:
    image: mysql:8.0
    restart: always
    env_file:
      - .env
    environment:
      MYSQL_DATABASE: ${DB_NAME}
      MYSQL_USER: ${DB_USER}
      MYSQL_PASSWORD: ${DB_PASSWORD}
      MYSQL_ROOT_PASSWORD: ${MYSQL_ROOT_PASSWORD}
    volumes:
```

```
      - dbdata:/var/lib/mysql
    ports:
      - "3306:3306"

volumes:
  dbdata:
```

(cela doit ressembler à votre production du dernier TP)

3 Questions de réflexion

Avant de coder, prenez le temps de réfléchir à ces questions. Elles vous aideront à comprendre le **pourquoi** avant de voir le **comment**.

3.1 Sur la base de données

1. Pourquoi ajouter une base de données à un service météo aussi simple ? Est-ce justifié ?
2. Est-ce que chaque microservice devrait avoir sa propre base, ou peut-on les partager ?
3. Que gagne-t-on (et que perd-on) en utilisant une base relationnelle plutôt qu'un fichier ou un dictionnaire Python ?
4. Que permet une base comme MySQL que ne permet pas un fichier JSON ?
5. Si on voulait partager cette météo avec d'autres services, la base est-elle une bonne interface ?
6. Peut-on facilement sauvegarder/exporter les données ? Et les restaurer ?

3.2 Sur les performances et la scalabilité

7. Est-ce que l'ajout d'une BDD rend le service plus rapide ? Plus lent ?
8. Que se passe-t-il si plusieurs clients envoient des requêtes simultanément ?
9. Peut-on mettre à jour une donnée météo sans recontacter l'API externe ?
10. Est-ce qu'on peut interroger la météo d'hier ou de demain avec cette architecture ?

4 Le vif du sujet : comprendre et utiliser SQLAlchemy (ORM)

4.1 Qu'est-ce qu'un ORM ?

Un **ORM** (Object-Relational Mapper) est une bibliothèque qui permet de manipuler une **base de données relationnelle** à l'aide d'objets Python, **sans écrire de SQL directement**.

— Vous écrivez des **classes Python** → il crée des **tables SQL**
— Vous instanciez des objets Python → il insère des **lignes en base**
— Vous modifiez des attributs → il exécute des **requêtes UPDATE**

L'ORM que nous utilisons ici est **SQLAlchemy**, avec son intégration Flask.

4.2 Installer SQLAlchemy

Une petite ligne de commande et hop : `pip install flask flask_sqlalchemy pymysql`

4.3 Déclarer le modèle

Dans `models.py`, on définit une classe qui hérite de `db.Model` :

```python
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

class WeatherData(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    city = db.Column(db.String(100), nullable=False)
    temperature = db.Column(db.Float)
    windspeed = db.Column(db.Float)
    condition = db.Column(db.String(100))
    timestamp = db.Column(db.DateTime)
```

Cela crée une table `weather_data` avec les colonnes suivantes : `id`, `city`, `temperature`, `windspeed`, `condition`, `timestamp`.

4.4 Initialiser SQLAlchemy dans `app.py`

```python
from flask import Flask
from models import db
from datetime import datetime

app = Flask(__name__)

# Config depuis les variables d'environnement
app.config["SQLALCHEMY_DATABASE_URI"] = (
    ↪ f"mysql+pymysql://{DB_USER}:{DB_PASSWORD}@{DB_HOST}:{DB_PORT}/{DB_NAME}"
)
app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False

# Initialisation
db.init_app(app)

# Création des tables (à faire une fois)
with app.app_context():
    db.create_all()
```

Pensez à installer le driver `pymysql` si besoin :

```
pip install pymysql
```

4.5 Créer et récupérer des données

4.5.1 Ajouter une entrée

```python
from models import WeatherData
from datetime import datetime

entry = WeatherData(
    city="Rodez",
    temperature=22.5,
    windspeed=10.4,
    condition="nuageux",
    timestamp=datetime.utcnow()
)
db.session.add(entry)
db.session.commit()
```

4.5.2 Chercher une donnée en BDD

```python
result = WeatherData.query.filter_by(city="Rodez").first()
```

On peut ensuite comparer le `timestamp` pour décider si la donnée est "fraîche" ou non.

5 Réparer le service pour éviter la magouille de Philibert (et tous les autres)

> *"Mais moi je m'en fiche, j'avais fait un code simple qui retient la météo et qui l'affiche. Je veux que ça marche !"* - Philibert Roquart, toujours pas convaincu

Avant même de penser à quoi que ce soit (trop fatiguant), Philibert a écrit un petit script personnel. C'est moche, ça spamme, mais… ça *marche*. Voici un exemple (à **ne pas reproduire tel quel**, sauf pour le corriger !) :

```python
# philibert.py
import requests
import time

while True:
    try:
        r = requests.get("http://localhost:5000/weather?city=Rodez")
        print(r.json())
    except Exception as e:
        print("Erreur:", e)
    time.sleep(1)
```

Ce script :

- interroge le microservice météo **chaque seconde**, sans jamais mémoriser la réponse,
- utilise `print()` pour tout afficher,

— n'a aucune tolérance aux erreurs réseau,
— fonctionne en local uniquement.

Votre mission, si vous l'acceptez :

— Éviter que ce genre de script n'explose les quotas de l'API ou ne fasse tourner le serveur dans le vide.
— Ajouter une couche de cache ou de persistance.
— Bonus : aider Philibert à améliorer son script pour qu'il interroge **intelligemment** le service météo.

6 Questions de réflexion

1. Pourquoi voudrait-on éviter d'écrire directement des requêtes SQL à la main ?
2. Que gagne-t-on en utilisant un ORM comme SQLAlchemy ?
3. Est-ce que l'ORM vous empêche complètement d'accéder au SQL si besoin ?
4. Est-ce que le code Python devient plus clair ou plus opaque avec un ORM ?
5. À quel moment l'ORM peut devenir un inconvénient ? (performances, complexité, etc.)

6.1 Quelques pistes pour vous aider...

— Un ORM est très pratique pour prototyper vite, mais **introduit une abstraction** : il faut comprendre ce qu'il génère en SQL !
— SQLAlchemy est un ORM très complet, mais aussi complexe. Ici, on reste dans un usage basique.
— Pensez à gérer les erreurs de connexion ou d'intégrité (`try/except`).