Architecture micro-service

TP 5 - Premiers micro-services - Faites des blagues

Philippe Roussille

3iL INGENIEURS
</les sciences informatiques>

1 Contexte

2023. Roger et Ginette, responsables informatiques historiques de l'usine CanaDuck, préparent leur départ en retraite. Pour assurer la continuité de l'infrastructure numérique de l'entreprise et transmettre un peu de leur savoir à Lucie et Matthieu, leurs enfants bientôt en DUT (eh oui, ça passe vite), ils ont lancé un chantier de modernisation : tout passer en microservices.

Premier élément migré : le générateur de blagues internes. Cette fonction, jadis embarquée dans un vieux script console, doit devenir un **microservice indépendant** et réutilisable.

2 Objectifs pédagogiques

Dans ce TP, nous allons commencer à implémenter la vision microservice : **donner vie à de petites fonctionnalités**, en les rendant **autonomes**, **réutilisables**, et **indépendantes**. Plutôt qu'un gros service qui fait tout, chaque service est responsable d'une seule chose. Cela rend le système plus souple, plus modulaire, et plus facile à maintenir.

3 Questions de compréhension

Avant de parler microservices, posons-nous quelques questions simples :

3.1 Pourquoi faire des petits services ?

1. Pourquoi voudrait-on **diviser** un gros service web en plusieurs morceaux plus petits ?
2. Imaginez que vous travaillez sur un gros projet. Que se passe-t-il si vous devez modifier une seule fonctionnalité ? Est-ce facile ? Risqué ?
3. Peut-on confier un petit service à une autre équipe ou un autre développeur sans qu'il ait besoin de connaître tout le reste ?

4. Que se passe-t-il si une partie tombe en panne ? Peut-on réparer sans tout redémarrer ?
5. Avez-vous déjà vu ou utilisé un site ou une appli qui semblait “modulaire” ?

3.2 Comment découper un service ?

6. Sur quels critères peut-on séparer un gros service en plusieurs petits ?
7. Faut-il découper par fonctionnalité (ex : blague, météo, cantine…) ? Par type de donnée ? Par public cible ?
8. À partir de combien de lignes de code ou de routes HTTP faut-il envisager un découpage ?
9. Le découpage doit-il être figé ou peut-il évoluer ?

4 Le premier micro-service

4.1 Spécifications fonctionnelles

4.1.1 Endpoints à implémenter

`GET /joke`

— Renvoie une blague aléatoire
— Réponse attendue (JSON) :

```
{ "joke": "Où se cachent les canards ? Dans le coin coin coin." }
```

`POST /joke`

— Ajoute une nouvelle blague au service
— Corps de requête (JSON) :

```
{ "joke": "Que dit un canard qui s'est perdu? Coin !" }
```

— Le champ `joke` est obligatoire et doit faire au moins 10 caractères.
— Réponse en cas de succès : code HTTP `201 Created`
— Réponse en cas d’erreur : code HTTP `400 Bad Request` + message JSON

Données Voici quelques exemples de blagues que vous pouvez intégrer à votre liste initiale :

```
[
  { "joke": "Pourquoi un canard est toujours à l'heure ? Parce qu'il est
   ↪  dans l'étang." },
  { "joke": "Quel est le jeu de cartes préféré des canards ? La
   ↪  coin-che." },
  { "joke": "Qu'est-ce qui fait Nioc nioc? Un canard qui parle en
   ↪  verlan." },
  { "joke": "Comment appelle-t-on un canard qui fait du DevOps ? Un
   ↪  DuckOps." }
]
```

— Ces blagues peuvent être codées en dur dans une liste Python.

— Vous pouvez les enrichir librement (j'ai probablement raté une longue carrière d'humoriste).
— Blagues stockées en **mémoire** (liste Python)

4.2 Suggestions d'évolutions (facultatif)

— `GET /jokes` : lister toutes les blagues
— `GET /joke/<int:id>` : accéder à une blague précise
— Enregistrer les blagues dans un fichier JSON (persistance simplifiée)

4.3 Contraintes techniques

— Utiliser **Flask** uniquement (sans framework additionnel)
— Code Python 3.7 ou plus
— Retour JSON standardisé (avec en-tête `Content-Type: application/json`)

4.4 Pour tester votre service

— `curl http://localhost:5000/joke` (de base, Flask tourne sur le port 5000)
— `curl -X POST -H "Content-Type: application/json" -d '{"joke": "coin coin coin"}' http://localhost:5000/joke`

5 Documenter le microservice avec Flasgger

> *"Roger… t'as bien documenté ton service j'espère ? Parce que si je tombe malade, c'est pas toi qui vas me retrouver les routes à la main dans app.py…"* — Ginette, un matin de maintenance

Après avoir mis en place un microservice de blagues fonctionnel, Roger se rend compte qu'il manque quelque chose d'essentiel : **la documentation automatique de l'API**.

Heureusement, il existe une solution simple et efficace : **Flasgger**, une extension de Flask qui permet de documenter l'API REST directement dans le code, et d'afficher une interface Swagger à l'URL `/apidocs`.

5.1 Installation

Ajoutez à votre `requirements.txt` ou installez via pip :

```
pip install flasgger
```

5.2 Rédaction d'une documentation Swagger

La documentation Swagger (ou OpenAPI) s'écrit généralement en **YAML** à l'intérieur des docstrings Python. Voici la structure typique :

```
"""
Résumé de la route
---
parameters:
```

```
  - name: nom_du_champ
    in: body / query / path
    required: true / false
    schema:
      type: string / integer / object
      example: "Exemple de valeur"
responses:
  200:
    description: Succès
    examples:
      application/json: {"clé": "valeur"}
  400:
    description: Erreur de requête
"""
```

- `parameters` : décrit les champs attendus (dans le corps, l'URL, etc.)
- `responses` : décrit les codes HTTP possibles et les structures de réponse
- `examples` : affiche un exemple concret dans Swagger UI

Cette structure est interprétée automatiquement par Flasgger.

5.3 Étapes à suivre

1. **Importer Flasgger dans `app.py`**

```python
from flasgger import Swagger
```

2. **Initialiser Swagger après la création de l'app Flask**

```python
app = Flask(__name__)
swagger = Swagger(app)
```

3. **Documenter chaque route à l'aide de docstrings au format YAML**

Exemple pour `GET /joke` :

```python
@app.route("/joke", methods=["GET"])
def get_joke():
    """
    Renvoie une blague aléatoire
    ---
    responses:
      200:
        description: Une blague bien formulée
        examples:
          application/json: {"joke": "Pourquoi les canards n'ont pas
↪  d'ordinateur ?"}
    """
    ...
```

Exemple pour `POST /joke` :

```python
@app.route("/joke", methods=["POST"])
def post_joke():
```

```
    """
    Ajoute une nouvelle blague
    ---
    parameters:
      - name: joke
        in: body
        required: true
        schema:
          type: object
          properties:
            joke:
              type: string
              example: "coin coin coin"
    responses:
      201:
        description: Blague enregistrée avec succès
      400:
        description: Erreur dans le format de la requête
    """
    ...
```

4. **Lancer le serveur et tester** Accédez à la documentation interactive sur :

```
http://localhost:5000/apidocs
```

Vous pouvez **tester les routes directement dans l'interface Swagger** : envoyer des `POST`, modifier les valeurs, voir les réponses en direct.

6 Dockeriser le microservice

> *"Moi, je réinstalle pas Flask et Flasgger à chaque fois qu'on bouge de machine, hein."* — Patrice, nouvel arrivant chez CanaDuck *"La prod', c'est pas un environnement de test. On veut du propre, du portable, du conteneurisé !"* — Paul, DevOps fraîchement recruté

Après avoir mis en place un microservice de blagues bien documenté avec Flasgger, il est temps de rendre son déploiement **portable et reproductible**. Pour cela, on va **dockeriser** l'application.

6.1 Pré-requis

— Avoir un fichier `app.py` fonctionnel
— Avoir un fichier `requirements.txt` contenant au minimum `flask` et `flasgger`

6.2 Étapes à suivre

6.2.1 Créer un fichier `Dockerfile`

```
# Utilise une image de base Python
FROM python:3.9-slim
```

```dockerfile
# Crée un répertoire de travail
WORKDIR /app

# Copie le code et les dépendances
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

# Expose le port du service
EXPOSE 5000

# Lance le serveur Flask
CMD ["python", "app.py"]
```

6.2.2 Créer un fichier `.dockerignore`

```
__pycache__
*.pyc
*.pyo
*.pyd
*.db
.env
```

Vous connaissiez déjà le `.gitignore`, je vous laisse deviner l'utilité de ce dernier…

6.2.3 Construire l'image Docker

```
docker build -t canaduck/blague-service .
```

6.2.4 Lancer le conteneur

```
docker run -p 5000:5000 canaduck/blague-service
```

Vous pouvez maintenant accéder à :

- `http://localhost:5000/joke`
- `http://localhost:5000/apidocs`

6.3 Bonus : Docker Compose (optionnel)

Créer un fichier `docker-compose.yml` pour un lancement encore plus simple :

```yaml
version: '3.8'
services:
  blagues:
    build: .
    ports:
      - "5000:5000"
```

Lancez avec :

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

> *"C'est la première fois que je déploie un canard dans un conteneur. Je crois qu'on a franchi un cap."* — Ginette