Architecture micro-service

TP 5 - Premiers micro-services - Agrégation de service

Philippe Roussille

3iL INGENIEURS
</les sciences informatiques>

1 État des lieux

1.1 Présentation du cadre

Ginette est contente : son service de blague fonctionne. Roger aussi : il peut demander la météo à Rodez, Tombouctou ou Honolulu. Mais Hortense, qui doit gérer les affichages dans l'usine, en a assez de jongler entre deux services :

> *"Je veux une seule réponse, avec la météo ET une blague, et pas deux appels dans mon JavaScript !"*

Bonne idée. C'est le moment d'introduire un nouveau service, que l'on appellera **le service agrégateur**.

Il sera écrit lui aussi en Flask, et fera deux appels HTTP internes à vos autres microservices :

- un vers `/weather?city=...`
- un vers `/joke`

Puis il **fusionnera** les deux réponses dans un seul objet JSON.

1.2 Spécifications du service agrégateur

Le service doit exposer trois routes :

1.2.1 1. Route `/fullinfo?city=...`

Fait un appel aux deux microservices (blague et météo) et renvoie une réponse combinée.

Exemple :

```
{
  "city": "Rodez",
  "weather": {
    "temperature": 21.2,
    "condition": "nuageux"
```

```
  },
  "joke": "Où est caché le canard ? Dans le coin."
}
```

1.2.2 Route `/weather?city=...`

Fait uniquement un appel au service météo et renvoie sa réponse sans la modifier.

1.2.3 Route `/joke`

Fait uniquement un appel au service blague et renvoie sa réponse sans la modifier.

Ces deux dernières routes permettent de **repasser par le même point d'entrée**, sans accéder directement aux services sous-jacents.

1.3 Petits tuyeaux sympathiques

- Utiliser `requests.get(...)` en interne pour contacter les deux services
- S'assurer que le service météo et le service blague tournent déjà (en local ou via Compose)
- Le code du service agrégateur doit **gérer les erreurs** si l'un des deux services est en panne
- Documenter ce que fait le service (README, docstring ou Swagger si envie)

2 Quelques petites questions

1. Ce service fait deux appels HTTP internes. Est-ce efficace ? Pourquoi ?
2. Que se passe-t-il si l'un des deux services répond avec une erreur ou met trop de temps ?
3. Peut-on réutiliser ce service dans d'autres cas (email, impression, vocal) ?
4. Quelle est la différence entre un service "client" (ex : navigateur) et ce nouveau service ?

3 Dockerisation du service agrégateur

Une fois que votre service agrégateur fonctionne, il est temps de le **dockeriser** pour l'intégrer facilement aux autres services dans une stack complète.

Je vous laisse vous débrouiller : ça fait trois fois que vous faites ça aujourd'hui…

3.1 Petit points techniques sympathiques

3.1.1 Sur le `docker-compose.yml`

Assurez-vous qu'il utilise le même réseau que les autres, pour pouvoir les joindre par leur nom de service Docker :

```
aggregator:
  build: ./aggregator
```

```
    ports:
      - "5002:5000"
    networks:
      - canaduck
```

Et ajoutez au bas du fichier :

```
networks:
  canaduck:
    driver: bridge
```

3.1.2 Appels internes

Dans `app.py`, vous pourrez appeler les services comme :

```
requests.get("http://weather:5000/weather?city=Rodez")
requests.get("http://jokes:5000/joke")
```

(`weather` et `jokes` sont ici les noms des services dans le `docker-compose.yml`)

4 Limites de cette approche ?

Lucie et Matthieu, toujours curieux, posent une question importante à leurs parents (probablement un autre stage d'observation !) :

> *Mais… c'est bien ce service agrégateur, mais s'il faut reprogrammer un nouveau service à chaque fois qu'on veut en combiner deux, on n'a pas fini, non ?*

Roger hoche la tête. Ginette soupire. Effectivement, **plus les services se multiplient**, plus les combinaisons deviennent complexes à gérer. Chaque nouvelle entrée demande :

- Un nouveau port exposé
- Une mise à jour des appels
- Une configuration supplémentaire

Et surtout : **des doublons de code** et des redirections à maintenir manuellement.

> *"Il nous faut un point d'entrée unique, modulable, qui redirige vers le bon service automatiquement."*

C'est là qu'intervient **Traefik**, un outil appelé **reverse proxy dynamique**, qui va jouer le rôle de répartiteur :

- Il connaît l'ensemble des services déclarés
- Il les expose proprement sous une seule adresse (`/weather`, `/joke`, `/fullinfo`…)
- Il ne nécessite **aucune modification dans le code** de vos microservices

Et justement, vous allez l'ajouter **dans ce TP** !

5 Mise en place de Traefik dans la stack

5.1 Créer un réseau commun `traefik_net` :

```
docker network create traefik_net
```

5.2 Ajouter Traefik au `docker-compose.yml`

```yaml
traefik:
  image: traefik:v2.11
  command:
    - "--api.insecure=true"
    - "--providers.docker=true"
    - "--entrypoints.web.address=:80"
  ports:
    - "80:80"
    - "8080:8080"  # interface de contrôle
  volumes:
    - "/var/run/docker.sock:/var/run/docker.sock:ro"
  networks:
    - traefik_net
```

5.3 Modifier les services existants pour qu'ils utilisent ce réseau et déclarent leurs routes

Exemple pour le service météo :

```yaml
weather:
  build: ./weather
  labels:
    - "traefik.enable=true"
    - "traefik.http.routers.weather.rule=PathPrefix(`/weather`)"
    - "traefik.http.services.weather.loadbalancer.server.port=5000"
  networks:
    - traefik_net
```

Même chose pour les services `jokes` et `aggregator`, avec leurs propres `PathPrefix` (`/joke`, `/fullinfo`).

Une fois lancé, accédez à `http://localhost/weather` ou `http://localhost/fullinfo` sans vous soucier des ports.

> *"C'est comme si on mettait une pancarte à l'entrée de chaque service."*
> — Paul

6 D'autres reverse proxies que Traefik ?

Traefik est un excellent choix pour une stack Docker dynamique, mais ce n'est **pas le seul**. Voici deux autres alternatives très répandues :

— **Nginx (reverse proxy)**

- Configuré via des fichiers `nginx.conf` ou des blocs `location /` dans un serveur virtuel.
- Très robuste, utilisé dans de nombreuses infrastructures en production.
- Nécessite une configuration manuelle ou des outils externes pour suivre l'état de vos conteneurs.

— **Caddy**

- Configuration très concise avec un `Caddyfile`, facile à apprendre.
- Gère automatiquement les certificats HTTPS avec Let's Encrypt.
- Moins connu que Nginx, mais très apprécié pour les déploiements simples.

6.1 Questions d'exploration

1. Quelles sont les principales différences entre la configuration manuelle d'un `nginx.conf` et l'approche déclarative de Traefik ?
2. Dans quel type de projet Caddy pourrait-il être plus avantageux que Traefik ?
3. Que se passe-t-il si un service change de nom ou de port dans votre environnement Docker ? Lequel de ces outils gère cela automatiquement ?
4. Lequel de ces outils vous semble le plus adapté à un usage en production ? Pourquoi ?
5. Comment chacun de ces reverse proxies gère-t-il la génération et le renouvellement des certificats SSL ?
6. Est-il raisonnable d'exposer le `docker.sock` à un reverse proxy ? Quels avantages cela apporte-t-il ? Quels risques cela peut-il poser ?
7. Pourriez-vous imaginer un scénario où l'on utiliserait **Traefik pour les services internes** et **Nginx pour l'exposition publique** ? Pourquoi (ou pourquoi pas) ?

À vous de voir, en fonction de vos besoins : priorité à l'automatisation, à la clarté de configuration ou à la stabilité éprouvée ?

7 Le teaser

In fine, on a vu un peu tout ce qu'il y avait à voir sur les microservices avec ce TP :

— communication entre services via HTTP,
— appels croisés avec `requests`,
— agrégation de réponses,
— conteneurisation avec Docker,
— et mise en place d'un **reverse proxy dynamique** avec Traefik.

Ce TP vous a permis de comprendre **comment organiser, exposer et faire dialoguer** plusieurs microservices dans une architecture claire et évolutive.

Mais surtout, vous êtes désormais ***prêts pour la suite***.

7.1 Prochain TP (projet de fin de cours !) : IRC en architecture micro-services

Le prochain TP consistera à **découper et reconstruire un vrai projet** (un mini-IRC) en microservices :

- un service pour la gestion des utilisateurs,
- un service pour les messages,
- un point d'entrée unifié,
- persistance, scalabilité, sécurité, tests…

Vous mettrez ainsi en pratique **tout ce que vous avez appris** sur la composition, le découpage, la communication inter-service, la persistance, le déploiement, et la documentation.

En bref, vous allez construire **un vrai système distribué**, complet, à plusieurs. Soyez prêts à canarder…