TEAM D
Nassim Bounouas
Gaetan Duminy
Rudy Meersman
Nikita Rousseau

Service API

Document Service - 07/10/2018

Introduction

Dans le cadre du cours de *Service Oriented Architecture*, nous sommes amenés à réaliser une solution technologique pour le client *Uberoo*. Au travers du problème que propose cette entreprise factice, nous allons être confrontés au problème d'**intégration** de services.

Uberoo est une plateforme web qui permet à des restaurateurs d'afficher leur carte, permettant à des consommateurs de les commander et de les faire livrer à domicile. Dès lors, plusieurs acteurs de systèmes tiers sont mis en relation pour permettre ce service. Il faut donc permettre aux différents systèmes de collaborer entre eux.

Nous avons étudié en cours plusieurs paradigmes permettant à différents service de communiquer entre eux, tout en maximisant un couplage lâche entre eux.

Nous allons décrire dans ce rapport notre choix d'implémentation pour l'intégration des services, tout en proposant les alternatives connues.

Scénario

Dans le cadre de ce projet, le scénario à considérer est le MVP suivant:

> Bob est un étudiant qui souhaite commander un plat de ramen à l'aide de *Uberoo*.

1) Bob demande la liste des catégories, la récupère puis demande la liste des plats en donnant une catégorie. Il récupère la liste des plats et en choisit un celle-ci. Il interroge un service "menu"
2) *Uberoo* est interrogé pour calculer le temps estimée pour la livraison de la commande à Bob.
3) Bob récupère et accepte le temps de livraison.
4) *Uberoo* envoi la commande au restaurant et prévient un livreur.
5) Le livreur récupère la commande au restaurant et la livre à Bob.

Dans un premier temps, nous proposons de construire le squelette complet de l'application, sans interface homme machine. Les communications sont dès lors orientées machine 2 machine. Il faut démontrer que tous les sous-systèmes d'*Uberoo* sont capables de coopérer. Nous avons créé un script docker-compose pour l'orchestration du déploiement de l'application, et un script qui permet d'interroger les containers pour démontrer la communication de ceux-ci.

Acteurs et Systèmes

Les acteurs sont **Bob** (le client), le **restaurateur**, le **livreur** et le calculateur **ETA** d'*Uberoo*

Le système *Uberoo* est décomposé en plusieurs services:

- Le **Menu Service** qui permet au client de choisir une catégorie de plats puis permet au client de choisir un plat dans cette catégorie.

Ce service accepte les payloads suivants :

```
{
      "Action": "READ_CATEGORIES",
      "Message": {}
}
```

```
{
      "Action": "READ_MEALS_BY_CATEGORY",
      "Message": {
            "Category": "Japonais"
      }
}
```

- Le **Ordering Service** qui va recevoir le plat choisi par le client et va sélectionner un restaurant proposant ce plat.

Ce service accepte les payloads suivants :

```
{
    "Action" : "order_meal",
    "Message" :
    {
        "Meal" : "Ramen"
    }
}
```

- Le **ETA Computer** qui va calculer le délai estimée pour que le livreur vienne chercher la commande en restaurant ainsi que le délai pour que le client soit livré.

Ce service accepte les payloads suivants :

```
{
      "Action": "compute_eta",
      "Message": {
      "Meal": "Ramen",
      "Restaurant": "Lyianhg Restaurant",
      "Delivery_Address": "Campus Templier"
      }
}
```

- Le **Ordering Service** est à nouveau appelé pour générer un bon de commande avec un id unique. Le bon de commande contient le restaurant, le plat, la date de livraison, l'adresse de livraison et le prix.

Ce service accepte les payloads suivants :

```
{
    "Action" : "validate_order",
    "Message" :
    {
        "Meal": "Ramen",
        "Restaurant": "Lyianhg Restaurant",
        "Delivery_Address": "Templier",
        "Delivery_Date": "date",
        "Price": 5
    }
}
```

- Le **Restaurant Service** qui envoie au restaurant les informations de la commande : le plat et le délai pour que le livreur vienne chercher le plat.

Ce service accepte les payloads suivants :

```
{
    "Action":"Receive_order",
     "Message":
"{"Meal":"ORDERED_MEAL","Client":"CLIENT_NAME","PickUpDate":"PICKUP_DATE
"}"
}
```

- Le **Delivery Service** qui envoie au livreur les informations de la commande : le plat, le restaurant, le délai pour venir chercher le plat, le délai pour livrer le plat au client ainsi que son adresse.

Ce service accepte les payloads suivants :

```
{
    "Action":"Delivery_request",
    "Message":
"{\"Meal\":\"ORDERED_MEAL\",\"PickupAddress\":\"RESTAURANT_ADRESS\",\"Pi
ckUpDate\":\"PICKUP_DATE\",\"Client\":\"CLIENT_NAME\",\"DeliveryAdress\"
:\"DELIVERY_ADRESS\"}"
}
```

Description de la solution d'intégration

Face au problème donné, nous avons remarqué que le métier proposé par *Uberoo* était basé sur des événements. Nous allons envoyer des *messages* pour notifier des événements qui se produisent sur les composants du système *Uberoo*. Les acteurs vont alors *consommer* les messages et effectuer le traitement nécessaire, avec la possibilité d'émettre à leur tour des messages.

Ce constat est motivé par le fait que l'on peut décomposer le *workflow* du scénario de base par des événements.

Pourquoi utiliser l'implémentation **Document / Event** :

Points positifs:

- Uberoo peut recevoir des modifications de fonctionnalités et d'interface lourdes. Le développement pour est en mode "bac à sable". L'utilisation d'un "broker" permet de découpler la communication entre les systèmes de l'application sans altérer le comportement interne des services.
- Les logiciels orientés RPC et Ressource sont vulnérables aux changements d'interface. Nous ne connaissons pas exactement les changements futurs du projet. C'est pour cela que nous les avons pas adoptés.
- Il est possible d'écrire des nouveaux Documents (événements) suivant les nouvelles exigences du client. C'est un bon pour pour l'évolutivité.

Points négatifs:

- Chaque sous-système doit pouvoir consommer nos messages. Certaines organisations peuvent ne pas être en mesure de les consommer, comme les banques.
- La multiplication des messages peut rendre la communication inter-service opaque : le système peut ne plus devenir maintenable.

Implémentation

Pour cette version préliminaire du système *Uberoo*, nous n'avons pas utilisé d'**Entreprise Service Bus** pour le passage des documents (messages événementiels). Après discussion avec nos encadrants, il était trop anticipé d'appliquer les solutions technologiques comme **Kafka**. Nous avons ainsi simulé la communication par Documents au travers d'une REST API extrêmement simplifiée.

Chaque service offre un point d'entrée unique sur une route HTTP :

```
{POST} /receive_event
```

C'est sur cette route que le service va écouter les documents que les autres systèmes vont envoyer. Nous avons aussi choisi une structure de données simplifiée pour décrire l'événement porté par le document :

```
{
	"Action": "NOM_DE_L_ACTION",
	"Message": {
		"Paramètre 1": "Nom du paramètre 1"
		"Paramètre 2": "Nom du paramètre 2"
	}
}
```

Ce format est standardisé pour l'ensemble des services d'*Uberoo.*

Chaque service va ainsi exposer sa route et éventuellement contacter d'autres services durant le traitement d'une action. Ainsi, nous pouvons faire interagir les systèmes entre eux. L'inconvénient est que nous sommes sur un workflow d'opérations synchrones : on ne peut pas implémenter un véritable BUS qui nous permettrai par exemple d'avoir des callbacks, avec un principe *Hollywood* (*don't call us, we will call you*). Cela est handicapant pour des traitements lourds, qui peut amener à la perte de messages et ainsi altérer le workflow d'opérations.

Script d'intégration

Nous utilisons **Docker** afin de créer des images pour chacun de nos services et pour les bases de données associés. De cette manière, grâce à **Docker Compose** il nous est possible d'initialiser et de démarrer les différents services, tout en les ciblant sur les bases de données respectives.

Chaque service relié à une base de donnée gère deux configurations : une configuration de développement et une configuration de production. Le choix entre ses deux configurations se fait grâce à une variable d'environnement spécifiant le contexte. Cette méthode nous permet de passer automatiquement en production lors du déploiement par le **Docker Compose**, nous profitons donc de la résolution DNS interne à docker pour cibler la base de donnée correspondante.

Il est possible de tester le bon fonctionnement du système à l'aide du script bash **RUN.SH**.

Intégrations Alternatives

Approche Ressource

Cette approche est plus claire grâce à l'utilisation des verbes HTTP, on peut s'en servir pour la sémantique afin de nous aider à comprendre l'utilité d'une méthode si l'API est bien conçue. De plus, cette méthode retourne le status de la requête effectuée pour savoir si elle a réussi ou non. Elle permet aussi la mise en cache pour augmenter les performances du service.
Les requêtes étant sans état (*stateless*), il est possible d'augmenter la scalabilité d'un service en augmentant le nombre de noeuds (*élasticité horizontale*) qui traitent les requêtes sur l'interface, la distribution de celles-ci reposant sur un *load-balancer*.

Cette approche possède aussi un certain nombre de défauts que l'on doit prendre en compte. Elle est forcément synchrone par définition donc si un service n'est pas disponible, alors la chaîne d'opérations est rompue.
Enfin, l'approche ressource repose sur un *contrat faible* entre l'interface exposée et les clients. Cela amène plus de flexibilité. Cependant, il requiert une documentation détaillée pour être automatisé.

Approche RPC

L'un des avantages majeur de l'approche RPC vient de ses messages proche du code permettant une facilité d'écriture du message et dire directement ce que l'on souhaite. De plus, nous pouvons mettre en cache les messages permettant, tout comme l'approche Ressource, une meilleur persistance du service. Un autre de ses point fort est que le RPC est encapsulé permettant ainsi une facilité l'envoie de donnée.
Cependant, cette approche est vulnérable aux changement d'interfaces du fait que les services sont fortement couplé. Si l'un des services doit être modifié, tout le reste doit l'être aussi rendant difficile la mise à jour de ses derniers.

Une approche hybride ?

Nous venons de voir trois solutions, utilisant un paradigme différent. Cependant, aucune des trois approches n'est pleinement satisfaisante… Pourquoi ne pas prendre le meilleur des trois mondes ?

Il semble raisonnable de penser que la première partie du scénario, c'est à dire l'exploration des menus ainsi que le choix de l'adresse de livraison ne nécessite pas d'approche événementielle. En effet, c'est le même système qui traite les requêtes de l'utilisateur sur la première phase du scénario (1-2). Au moment du calcul de l'ETA ou de l'ordre de paiement que l'on diffuse des événements à des systèmes tiers.

Pour gagner en performance, il semble convenable de choisir un autre paradigme, comme une approche orientée *ressource*. L'interface utilisateur d'une application web sera de ce fait facilement intégrable.

Pour la partie paiement, nous pouvons envisager une approche *RPC*. L'organisme de paiement va offrir une interface avec laquelle on peut interagir.
Il est raisonnable d'imaginer qu'un organisme de paiement ne va pas consommer des messages dont le format est spécifique à notre système d'information. Une approche événementielle / document est donc écartée.
D'autre part, une approche orientée ressource (REST) ne propose de mécanisme de call-back. La réponse à notre requête pour la banque doit être synchrone. Si le traitement est trop long (timeout), on ne peut plus garantir l'intégrité de la transaction.

Ainsi, afin d'assurer que le paiement soit autorisé et que la carte de paiement ne soit pas frauduleuse, on peut mettre l'ordre de paiement en attente sur notre système et proposer une médiation avec la banque : on effectue un ordre de paiement sur leur interface en mode RPC, puis on se met en attente de notification de leur part sur l'état de la transaction (autorisée, refusée).

Synthèse

Face aux éléments traités dans les sections précédentes, on peut synthétiser les avantages et inconvénients des paradigmes de la façon suivante :

	Points positifs	Points négatifs
Remote Procedure Call	• Idéal pour les actions • Proche d'un langage de programmation	• De nombreuses commandes sont nécessaires à l'ensemble d'un processus • Payload souvent énorme par rapport à l'objectif réel (JSON-RPC résout ce problème) • Couplé aux procédures
Document (Event)	• Asynchrone • L'équilibrage de charge très simple • Peut avoir n'importe quel nombre d'instances d'un récepteur • Pas besoin de découvrir le service (fire and forget) • Le système de messagerie garantit la livraison • Enregistre les messages • Accusé de réception • Peut avoir des messages en double • Envoyer une requête / Attendre une réponse -> file d'attente temporaire • L'échec ne fait qu'augmenter la latence • Possibilité d'ajouter	• Mise en œuvre fastidieuse • S'appuyer sur la retransmission des messages entrants (L'idempotence doit être prise en compte). • Si le Bus est cassé, alors l'APP est cassée. • Difficulté de surveillance et de dépannage

	un nouveau event listener avec une logique métier supplémentaire	
Resources (REST)	• La sémantique du verbe HTTP aide à comprendre le comportement d'un endpoint si l'API est bien conçue. • Peut être mis en cache • Permet la combinaison entre les endpoints génériques et les paramètres de la requête • Idéal pour le domaine de modélisation (CRUD) • REST n'est pas CRUD, il peut traiter les actions • Peut être chargé et équilibré par l'infrastructure ou le logiciel • Codes d'état pour communiquer les problèmes sémantiques • Modèle naturel	• REST doit être stateless • Synchrone par défaut • "Fire and forget" ne convient pas naturellement • Si le système n'est pas disponible, il est compliqué de réessayer car il est pas idempotent • L'interrogation est un gaspillage (Exemple : REST Hooks) • Toujours faire attention aux timeouts et à la persistance

Conclusion

Au travers de ce projet, nous avons remarqué qu'il n'existe pas de paradigme parfait pour répondre à tous les besoins. Il faut choisir la bonne stratégie d'intégration en fonction des facteurs limitants et des degrés de flexibilité des services.