Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

2022/2023

Systèmes d'information 2

Rapport 01

TP2 – Microservices et Classe inversée framework Frontend

Ecole	Haute école d'ingénierie et d'architecture de Fribourg (HEIA-FR)
Branche	Systèmes d'information 2
Etudiants	Barras Simon, Collaud Roman, Terreaux Nicolas
Classe	ISC-IL-3
Version/Date	V1 du 18.01.2023
GIT	https://gitlab.forge.hefr.ch/si-ii-tp-groupe6-22-23/si-ii-tp2-groupe6-22-23

Hes·so FRIBOURG/FREIBURG
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

1 Introduction

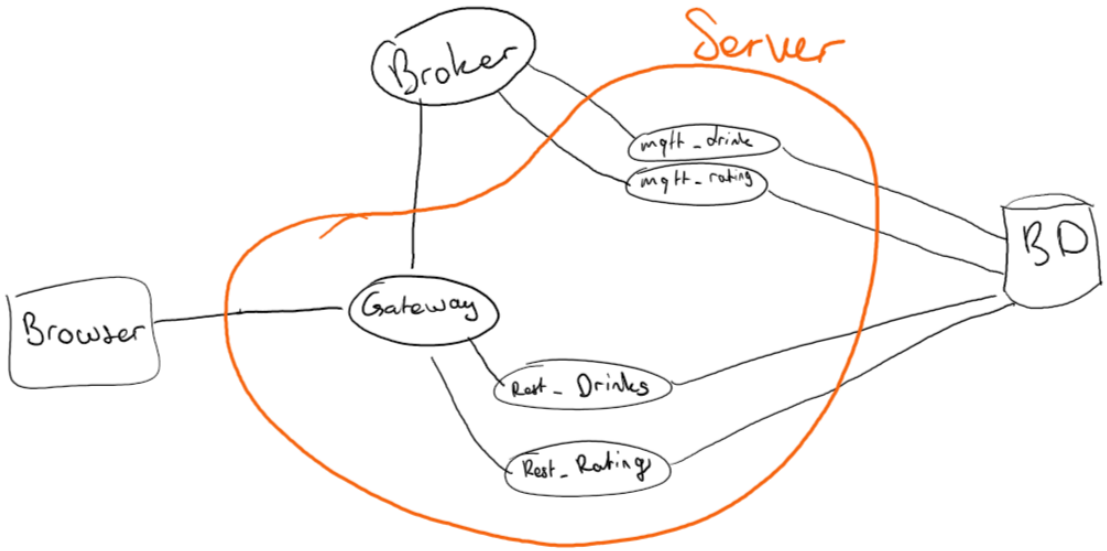
Dans ce rapport, nous allons décrire notre implémentation d'une application web avec une architecture de type micro-services. Le but de ce travail est de créer une application de notation de boissons. Nous allons implémenter une application avec deux moyens de communication, une communication synchrone via API REST et l'autre asynchrone via un broker. Nous allons ensuite comparer les deux moyens de communications de l'application selon des critères pertinents tels que la scalabilité, la latence et la fiabilité. Nous allons également prendre en main un Framework frontend pour l'application et le présenter de manière théorique. Nous allons décrire les étapes de l'implémentation de l'application, les technologies utilisées, les défis rencontrés et les résultats obtenus. Enfin, nous allons décrire comment nous avons déployé l'application sur un cluster Kubernetes sur Exoscale et comment nous avons utilisé une pipeline CI/CD pour automatiser les étapes d'implémentation.

2 Etapes de développements

Nous avons décidé d'implémenter 5 micro-services. Le premier, est un Gateway afin de simplifier les appels depuis le Browser. Toutes les requêtes du Browser vers le backend passent pas ce Gateway et ce Gateway redirigera ces requêtes vers le bon micro-service REST ou vers le Broker MQTT. Ensuite, des micro-services s'occuperont de revoir les requêtes http ou d'écouter les queues sur le broker afin de traiter les données et les enregistrer dans la base de données.

Nous avons utilisé MongoDB pour le stockage de nos éléments. La base de données est directement stockée chez eux.

Voici un schéma simplifié qui résume notre architecture.



La partie entourée est les micros-services que nous avons développés. Le broker est aussi déployé sur le cluster et la partie svelte est représentée ici par le Browser.

2.1 Backend

Pour le backend, nous avons 6 micro-services :

- gateway : réceptionne toutes les requêtes depuis l'UI et les redirige sur les bons micro-services
- drink-rest : qui s'occupe des boissons avec REST
- rating-rest : qui s'occupe des avis des boissons avec REST
- rabbitmq : notre broker MQTT
- drink-mqtt : qui s'occupe des boissons avec MQTT
- rating-mqtt : qui s'occupe des avis avec MQTT

Nous avons décidé d'utiliser Go pour la partie Backend. Pour l'API REST, nous avons utilisé Gin et pour MQTT, nous avons utilisé le broker RabbitMQ.

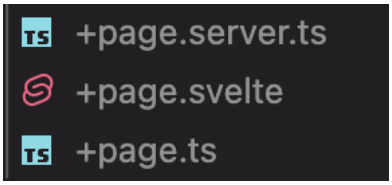
Pour ce qui est du déploiement, nous avons suivi les étapes du TD5.

2.2 UI Svelte

La partie frontend a été créée avec le framework svelte (sveltekit).

Dans un premier temps, il a été question de pouvoir requêter sur le backend pour récupérer les produits afin de les afficher sur l'interface.

Svelte a une structure de projet un peu différente d'autres frameworks frontend que l'on a pu prendre en main jusqu'ici.



Le routing dans svelte est déterminé par une structure hiérarchique dans un dossier nommé « routes » où chaque sous-dossier représente une nouvelle route et contient généralement trois fichiers qui servent à construire une page qu'on peut voir comme une vue de l'application.

Le fichier +page.svelte va contenir tout le code client exécuté du côté du navigateur.

Le fichier +page.ts contient toute la logique qui permet de traiter et requêter les données qui peuvent ensuite être récupérées pour être afficher de manière réactive sur les vues.

Le fichier +page.server.ts servent à définir le code côté serveur pour les pages spécifiques.

Dans un deuxième temps, nous avons développé le formulaire et la logique qui permet de soit ajouter ou modifier un produit du shop. Ensuite la fonctionnalité de suppression d'un produit et la possibilité de pouvoir ajouter une évaluation pour un produit spécifique.

Malheureusement, nous n'avons pas réussi à récupérer correctement les évaluations existantes d'un produit et pouvoir les afficher dynamiquement sur l'interface sous forme de timeline comme demandé. En effet, nous avons rencontré un problème avec le traitement des données reçus sous forme de *Promise* de la requête asynchrone qui requête les données sur le backend. Nous recevons bien les données côté client mais nous n'avons trouvé le moyen d'assigner celles-ci dans une variable réactive qui permet de les afficher dynamiquement sur l'interface.

En ayant perdu beaucoup de temps sur ce problème, nous n'avons pas eu le temps d'implémenter la création d'un pdf et le switch qui nous aurait permis de choisir depuis le frontend quel microservice

(mqtt ou rest) est utilisé côté backend. Néanmoins c'est une fonctionnalité que nous avions déjà implémenté lors du travail pratique précédent et qui était fonctionnelle.

2.3 BD

Comme expliquer plus haut, la base de données est stockée chez MongoDB. Nous avons 2 tables : drink et rating.

Voici un exemple de boisson :

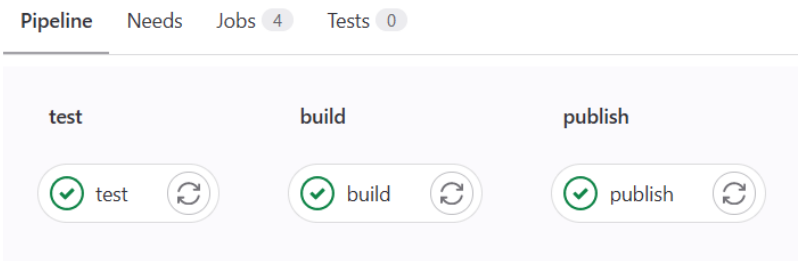
```
_id: 4039455774
name: "Beer1"
alcohol: 4.199999809265137
producer: "Heineken"
price: 4.800000190734863
country: "Holland"
capacity: 5
```

Et un exemple de rating :

```
_id: 1800391757
drinkId: 0
date: "2023-01-12 10:11:12.489811 +0000 UTC"
author: "string"
title: "string"
comment: "string"
rating: 0
```

2.4 Pipeline

Nous avons utilisé la pipeline CI/CD de gitlab qui contient 3 étapes.



La première étape est des tests unitaires sur le backend, la seconde est le build et push automatique des 5 images nécessaires pour le backend. La dernière étape, est la publication ou mise à jour des micro-services sur le cluster kubernetes. Cette étape n'est exécutée dans la pipeline que si nous faisons un commit sur la branche main.

3 Analyse et comparaison

Voici les 3 critères que nous avons choisi :

- Performance
- Facilité d'implémentation

- Possibilités d'extension à des nouvelles fonctionnalités

Nous avons choisi ces critères parmi la liste données car analyser l'occupation de la mémoire, la charge CPU ainsi que le coût de développement ne nous semblait pas utile dans notre cas. Nous avons aussi pris la facilité d'implémentation et pas la facilité de découplage car selon nous, les deux critères sont assez liés et nous avons donc choisi un des deux.

3.1 Performance

Requête REST : http://app.159.100.248.27.nip.io/api/v1/ MQTT : Ajout du ?mqtt=true après la requête	Temps REST	Temps MQTT

POST /drink
{
"alcohol": 999,
"capacity": 44,
"country": "string",
"id": 44,
"name": "string",
"price": 44,
"producer": "string"
}
	63ms	67ms

GET /drink/{id}	35ms	82ms
PUT /drink/{id}	42ms	47ms
DELETE /drink/{id}	37ms	47ms
GET /drinks	73ms	216ms

POST /rating
{
"author": "string",
"comment": "string",
"date": "string",
"drinkId": 0,
"id": 0,
"rating": 0,
"title": "string"
}
	88ms	52ms

GET /ratings/info/{id}	34ms	106ms
GET /ratings/{id}	47ms	131ms

Comme on peut le voir dans le tableau ci-dessus, les résultats sont pratiquement les mêmes C'est certainement dû au fait que ce sont des « petites applications » avec peu de données. La méthode synchrone a tout de même un petit avantage.

Par conséquent, pour cette application, le choix entre REST et MQTT n'a pas d'importance sur les performances.

Ça n'aurait pas été la même chose pour une application avec beaucoup plus de données.

3.2 Facilité d'implémentation

Dans notre cas, il a été beaucoup plus facile d'implémenter l'application synchrone. En effet, nous avons déjà passablement fait d'API REST. Le langage de programmation go nous propose la librairie Gin qui rend le développement de l'API relativement simple. Nous avons même mis en place un Swagger qui nous a permis de tester rapidement nos requêtes sans avoir besoin d'utiliser un autre logiciel et donc de faire nos requêtes à la main.

Pour ce qui est de la partie asynchrone, c'est déjà plus compliqué à se représenter comment faire fonctionner l'application. Ce n'est pas un « simple » client-server comme REST. Il y a un broker, des queues et des channels, il faut comprendre les concepts de publisher/subscribers, …

Pour résumé, pour nous, l'application synchrone est beaucoup plus facile à implémenter que la partie asynchrone.

3.3 Possibilités d'extension à des nouvelles fonctionnalités

REST permet d'ajouter très facilement et rapidement une nouvelle route (GET, POST, POST ou DELETE) ce qui rend l'API extensible mais tout de même limité. Le fait d'utiliser sous forme de micros-services rend l'application bien plus extensible.

Pour ce qui est de MQTT, elle est aussi très extensible. Nous avons vu que MQTT prend en charge des fonctionnalités de qualité de service pour garantir la fiabilité de la transmission des messages contrairement à REST. Il prend aussi en charge le temps réel et est adapté pour l'IOT ce qui offre énormément de possibilité d'extension à de nouvelles fonctionnalités.

En résumé, les deux applications sont très extensibles. Cela dépendra du cas d'utilisation.

4 Problèmes rencontrés

4.1 Timeout

Lorsque nous avons déployé nos différents micro-services, nous obtenions souvent des timeouts comme on peut le voir sur l'image suivante :

Nous avons passé énormément de temps sur cette erreur. Voici ce que nous avons essayé.

- Refaire les images, les tester en local (tout fonctionnait correctement) et tout redéployer
- Ajouter des ressources à notre cluster
- Réinstallation de nginx
- Modifications des fichiers de déploiements (utilisation du fichier du TD5)
- …

La solution était que nous avions commis une erreur dans nos security group. En effet, le port 4789 (utilisé par Calico, le système de réseau entre-noeuds) doit être en UDP et pas TCP comme nous l'avions mis. Merci à Vincent Jaquet d'avoir trouvé notre erreur.

5 Synthèse et conclusion

Le TP était intéressant. Nous aurions aimé faire plus mais le temps nous a manqué. En effet la période était la plus chargée de l'année. Nous avons cependant eu le problème de timeout indiqué dans le chapitre des problèmes rencontrés qui nous a fait perdre énormément de temps.

Nous trouvons que la charge de travail était beaucoup mieux adaptée. De plus, le fait de faire le TP avec le framework que nous devons présenter est très intéressant car nous avons un plus grand recul dessus.

6 Synthèse Exoscale

TP1 :

Pour le premier TP, la mise en place des machines est relativement simple et intuitive. On y a accès avec un outil comme PuTTY facilement et nous n'avons rencontré aucun problème à ce sujet.

TP2 :

La mis en place d'un Cluster est relativement simple. Nous avons pris un moment avant de trouver l'endroit où nous devions créer le cluster C'est peut-être que le nom « SKS » n'est pas très intuitif pour les gens qui débutent dans le domaine du Cloud. Pour le reste, NGINX, le load balancer, etc. étaient très facile à mettre en place. Dès lors que kubernetes était entièrement fonctionnel, les déploiements se sont passés sans problème.

En conclusion, nous trouvons qu'Exoscale est une excellente alternative aux autres géants comme Azure ou encore Amazon.