RAPPORT AL

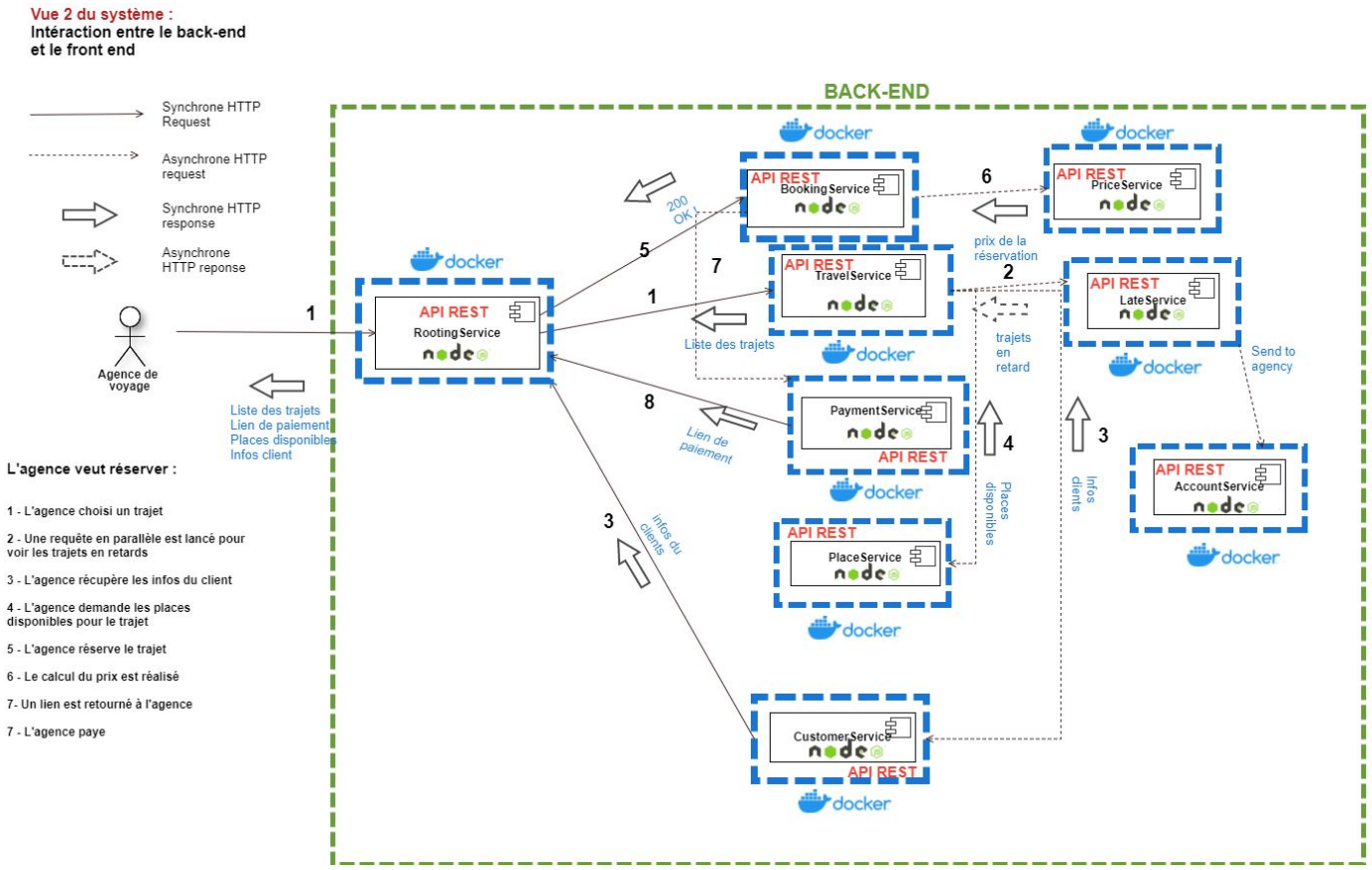
Team H

Sommaire

Rappel de l'architecture

Notre sujet était **Travel agent API and Group booking**. Pour cela nous avons choisi une architecture en microservices pour différentes raisons (découplage, flexibilité des technologies, meilleure scalabilité, meilleure maintenance…).
Nous avons actuellement 9 microservices dans notre application :

- **Rooting Service** : API gateway qui agit comme le point d'entrée entre le client et le backend de l'application. Pour l'instant elle sert juste à rediriger les requêtes vers le backend, mais il y a beaucoup d'avantages dont nous pourrions nous servir (sécurité, load balancing, différenciation des types de clients).
- **Booking Service** : Service "central" de l'application, qui va s'occuper de créer les réservations de trains. Pour cela il va communiquer avec la plupart des autres services.
- **Travel Service** : Renvoie les trajets disponibles pour différentes destinations et pour une date précise.
- **Place Service** : Gère les différentes places dans les trains, par exemple si le groupe de clients veut réserver une place PMR, ou veut un emplacement pour vélo… A terme, il servira également à choisir sa place précisément dans le train.
- **Account Service** : Gère les comptes personnels des clients qui s'enregistrent. Si une agence de voyage s'enregistre, elle recevra un numéro de client qui servira pour pouvoir réserver des voyages..
- **Payment Service** : Gère la facturation des voyages aux clients. Quand une réservation est effectuée, le service envoie un lien de paiement grâce auquel le client pourra payer et conclure la réservation.
- **Late Service** : Service qui va notifier le client si un train qu'il a réservé est en retard. Il n'est actuellement pas terminé, mais à terme il enverra un mail au client si un de ses trains a du retard. Il permet aussi de visualiser les trains avec du retard lors de la recherche de trajets avec Travel Service.
- **Price Service** : C'est ce service qui va calculer le prix des réservations en fonction de différents paramètres : l'âge des voyageurs, s'ils ont un abonnement, l'emplacement et le type de places…
- **Customer Service** : Contient les informations des voyageurs lorsqu'une agence réserve un voyage.



Pour l'instant toutes les données du système sont contenues directement dans la mémoire des services, mais nous aimerions implémenter des bases de données.
Tous nos services sont contenus dans des containers Docker et communiquent entre eux par des appels REST.

Choix architecturaux et techniques

Nous allons analyser dans cette partie les forces et les faiblesses de notre architecture afin d'avoir un recul sur notre projet, cela nous permettra de mieux savoir par la suite ce qu'il faudra améliorer pour le deuxième bimestre.

Forces :

Extensibilité et intégration de nouveaux services

Notre architecture en microservice permet d'intégrer de nouveaux services plus facilement, car chaque service peut être écrit dans un langage différent et il n'y aura pas de modification

de code sur les services existants du fait du faible couplage (chaque service à son propre domaine métier)

Pratiques Devops

- Nous avons utilisé Travis afin de tester automatiquement les services, cela accélère le “workflow” général du cycle de vie de notre application

- Docker : l’utilisation de Docker facilite la mise en production des microservices lorsque notre application sera distribuée, ils pourront être portables, cela facilite aussi le développement en uniformisant l’environnement de travail de chaque développeur.

- Les adresses de nos microservices sont gérées par des fichiers de configuration (.env), elles ne sont donc pas mélangées avec du code. Cela permet de modifier très facilement et rapidement l’environnement sur lequel tournent les services (environnement de développement ou environnement de production) et donc de faciliter le déploiement de ces derniers.

Microservices

- Chaque microservice a son propre domaine métier qui est entièrement dédié : par exemple, la création d’une réservation est uniquement gérée par le micro-service BookingService.
  Cela à plusieurs avantages tels que : une meilleure maintenabilité, une facilité à l'élasticité...

- L’utilisation du pattern API Gateway permet de faciliter l’utilisation de notre application en créant un unique point d’entrée : l’utilisateur n’a pas besoin de savoir comment est architecturée notre application et de connaître les adresses de tous les microservices pour pouvoir faire une réservation.

- Nous avons de la persistance au niveau des données grâce à nos bases de données, cela nous permet de ne pas perdre d’information en cas de panne

- Les communications de nos microservices sont en asynchrones (http asynchrone), cela permet de moins impacter la performance (en terme de temps) lorsque beaucoup de requêtes utilisateurs sont émises car lorsque plusieurs utilisateurs réservent, une chaîne de transactions est mise en place. Si cette chaîne était synchrone, alors chaque client devrait attendre la transaction d’un autre client, ce qui n’est pas acceptable car une architecture microservices doit être “scalable”

- En utilisant NodeJS dans tous nos services, cela nous permet de n’avoir qu’une technologie de développement à maîtriser et donc souvent une plus grande maîtrise

de cette dernière, en conséquence il y aura une meilleure qualité de code et une plus grande rapidité de développement

- Par nos choix technologiques, nos microservices sont très légers, cela nous permet de nombreux avantages tel qu'une baisse des coûts de stockage si l'on souhaite héberger nos microservices dans un serveur.

Faiblesses :

- Les bases de données utilisées ne sont pas accessibles si les microservices ne sont pas lancés dans le même environnement, cela peut être problématique si par la suite des données doivent être partagées entre différentes bases de données.

- Notre projet n'a aucune tolérance à la panne pour le moment: si un service ne fonctionne plus, alors, une grande partie de nos services ne fonctionneront plus et nous n'aurons pas un message d'erreur clair pour l'utilisateur.
  Lorsqu'il y a une transaction qui échoue lors de la réservation (par exemple, le service de paiement échoue), notre système ne pourra pas revenir en arrière et en informer l'utilisateur.

- Nos services ne sont pas déployés automatiquement par une CI, cela oblige les développeurs à le faire manuellement, cela en résulte une perte de temps et des risques d'erreurs

- Nos microservices ne sont pas encore déployés en ligne mais seulement en local, cela en résulte qu'il n'y a aucune accessibilité pour un client distant

- Nous n'avons pas de load-balancer permettant de gérer la charge, nous n'avons donc pas d'élasticité, cela peut s'avérer problématique si un grand nombre d'utilisateurs souhaitent accéder à notre API

- Il n'y a pas une documentation claire sur l'utilisation de notre API, cela est problématique si une personne extérieure souhaite s'appuyer sur l'API pour réserver des voyages, et cela peut être aussi problématique si un développeur externe souhaite travailler avec nous car il mettra plus de temps pour comprendre les fonctionnalités de chaque service, et cela impactera le temps de développement pour de futures fonctionnalités.

- Notre projet n'a comme tests que des tests unitaires, il faudrait intégrer d'autres types de tests afin de pouvoir mieux surveiller le bon fonctionnement et la qualité de notre code, cela pourrait être des tests d'intégration, des tests de montée en charge, des tests de components…

- Comme nous n'avons pas de tests de montée en charge nous avons des difficultés pour trouver des faiblesses architecturales liées à la montée en charge

Analyse pour la suite

Nous avons des scénarios qui ne sont pas encore gérés par notre API, il nous reste encore à gérer quatre scénarios que nous n'avons pas fait pour le POC. En dessous nous avons écrit les scénarios manquants avec les possibles modifications à faire pour que ça soit pris en compte. Mais une analyse encore plus poussée par la suite devra être réalisée afin de voir si on peut réaliser tous ces scénarios.

- Scénarios non gérés actuellement :
  - Une agence souhaite réserver un voyage mais retard d'un train :
    Créer un service pour l'envoie de mail
    Fait le lien avec une annulation gratuite de voyage (BookingService) si correspondance impossible
  - Une agence souhaite annuler une réservation avec/sans remboursement
    Il faudra inclure de nouvelles fonctionnalités dans différents services pour prendre en compte ce cas

  - Une agence souhaite modifier une réservation
    Gérer le fait que la réservation peut être plus chère
  - Une agence souhaite réserver un voyage avec des clients possédant des cartes de réductions

Il y a des modifications prioritaires qui devront être réalisées, nous avons cité plusieurs faiblesses. On souhaiterait améliorer des faiblesses qui pourraient être critiques par la suite du projet, nous avons ainsi fait la liste ci-dessous des modifications prioritaires qui devront être réalisées en plus des nouveaux scénarios.

Choix (état de l'art) et changement des bases de données afin d'avoir une meilleure structure dans la base de données, cela permet une plus grande fiabilité et de meilleures performances afin de permettre l'utilisation de notre API par un plus grand nombre d'utilisateurs, et une meilleure maintenabilité/lisibilité pour les développeurs

Séparation architecturale permettant de séparer l'écriture et la lecture des bases de données, meilleure maintenabilité/lisibilité pour les développeurs ainsi que meilleures performances avec l'ajout d'élasticité

Séparation de l'infrastructure des bases de données de celle des services afin que chacun puisse fonctionner sans l'autre.

Gérer la tolérance aux pannes en mettant en place un système permettant de gérer les erreurs et éviter l'effet domino. Lorsque l'on réserve, une chaîne de transactions est mise en

place entre les différents services qu'on a. Il faut éviter qu'une erreur n'entraîne par la suite des erreurs dans les autres services jusqu'à arriver à une erreur incompréhensible (ce qui est déjà arrivé). On pense à appliquer le pattern "Circuit breaker" afin d'éviter de se retrouver dans ce cas-ci et de mieux fiabiliser notre API. Il faudra s'assurer également que les services retournent des message d'erreurs cohérent afin que l'utilisateur puisse mieux comprendre le problème en cas de panne.

Création d'un service d'envoi de mail avec comme entrée un bus d'évènement : l'envoi de mail n'ayant pas besoin d'être synchrone, une queue semble être la meilleure solution pour gérer cette partie.

Création d'une documentation utilisateur pour l'API : étude de l'art des différents outils de création de documentation comme par exemple "Swagger"

Mise en place du pattern "SAGA" afin de s'assurer de l'atomicité des opérations lors d'une suite d'actions contenant par exemple une transaction financière.

Rétrospective des choix du premier bimestre

Les dernières semaines, beaucoup de fonctionnalités plus complexes ont dû être développées. Comme nous avions du retard sur d'autres fonctionnalités, la charge de travail a été grande la toute dernière semaine, ce qui nous a empêché de travailler pleinement sur le rapport final du projet et sur la soutenance.
Il aurait donc fallu que nous estimons plus fidèlement la charge de travail nécessaire pour le développement des fonctionnalités complexes et mieux distribuer le travail au sein du groupe.
Nous aurions dû également créer des tests automatiques des différents scénarios, car à chaque nouvelle fonctionnalité, nous testions à la main le bon déroulé du scénario. Des tests d'intégration auraient permis de ne pas perdre du temps.