CarGoverload

V4: Cargo

Encadrant : Guilhem Molines

Étudiants :

Masia Sylvain

Montoya Damien

Peres Richard

Rigaut François

2020 – 2021

Sommaire

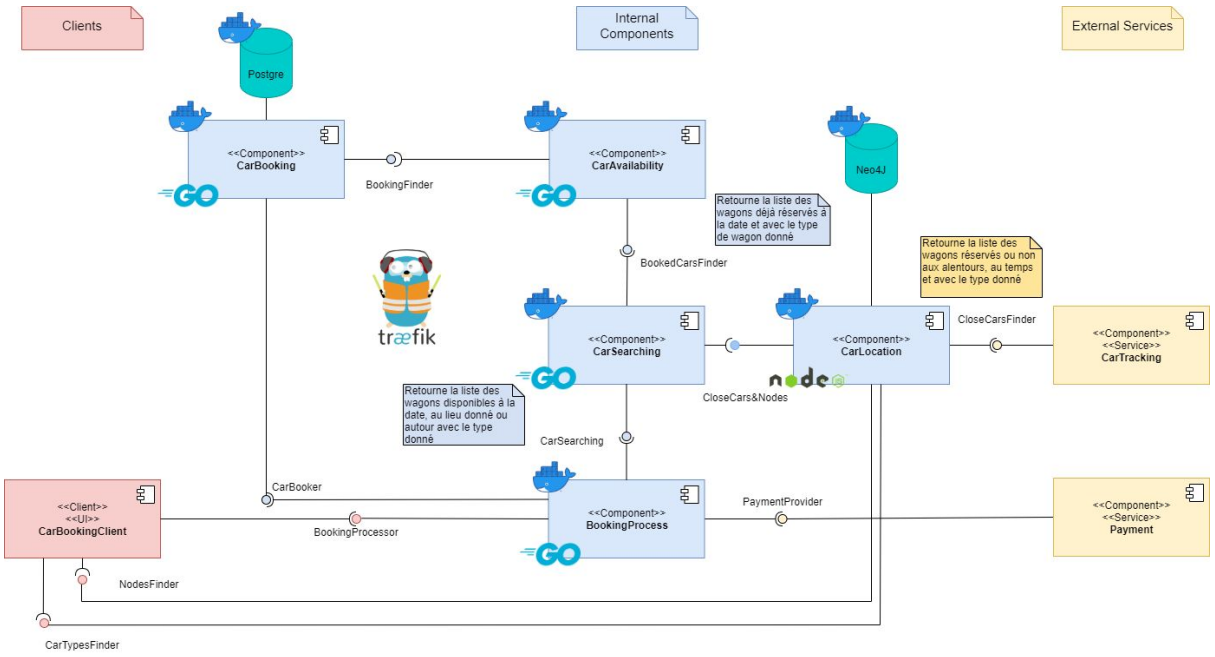
Architecture courante

Architecture globale

Nous avons fait évoluer notre architecture tout au long du projet pour en arriver au stade ci-dessous.

Nous sommes parties sur une architecture service/composants. Nous avons mis en place 5 services déterminés selon les fonctionnalités métiers afin de répartir au mieux les différentes étapes de la réservation de fret.

Ces services sont majoritairement faits en Golang, à l'exception de CarLocation qui est en NodeJS.



Infrastructure

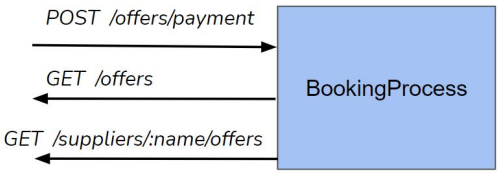
En terme d'infrastructure, tous nos composants sont Dockerisés et déployables via un Docker-Compose. Les services ont une image Docker qu'il leur est propre et est (re)générée lors de la mise à jour du service. Les bases de données Postgre et Neo4J sont également sous la forme de leur image Docker respective.

Nous avons mis en place Traefik en mode reverse-proxy pour simplifier le rooting entre chaque service. Chaque service tournant dans un conteneur Docker, ils exposent chacun un entrypoint HTTP afin de consommer leur API entre eux. Traefik se place devant ces services et permets de rooter l'ensemble des endpoints des services sans se soucier du port et du network.

Service BookingProcess

Ce service est responsable de la vente de réservation, c'est le premier service appelé par un client pour ce processus. Il se charge de transmettre au service carSearching les caractéristiques du wagon souhaités et de réaliser le paiement d'un résultat disponible. Ce service est stateful, les offres recherchées disponibles par un fournisseur sont stockées en mémoire pour lui donner la possibilité de réfléchir sur une proposition et d'en payer la réservation ultérieurement.

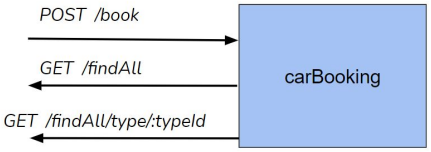
Interfaces



Service CarBooking

Ce service est responsable de la lecture et de l'écriture des réservations en base de données. Ce service est stateless, concernant le stockage des données de ce service, nous avons fait le choix d'utiliser Postgre pour stocker les réservations.

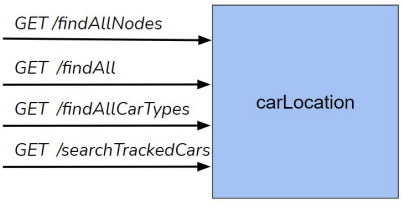
Interfaces



Service CarLocation

Ce service est responsable de la lecture et de l'écriture de la carte des nœuds en base de données. Il a également pour charge de trouver les wagons proches d'un nœud via l'utilisation du service externe CarTracking qui permet de connaître l'emplacement des wagons. Neo4j est utilisé pour stocker en base de données la carte des nœuds sous forme de graphe.

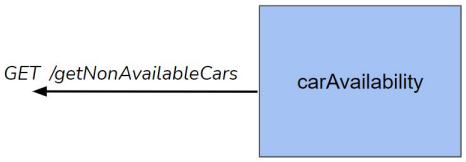
Interfaces



Service CarAvailability

Le service CarAvailability a pour but de faire le lien entre les réservations et la recherche de wagon. Il permet de renvoyer à CarSearching une liste des wagons déjà réservés selon un type et une date. Grâce à cela, CarSearching sera en mesure de savoir quel wagon proche d'un nœud est disponible ou non.

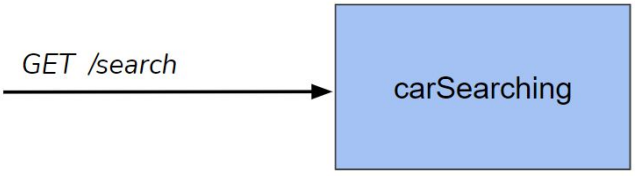
Interfaces



Service CarSearching

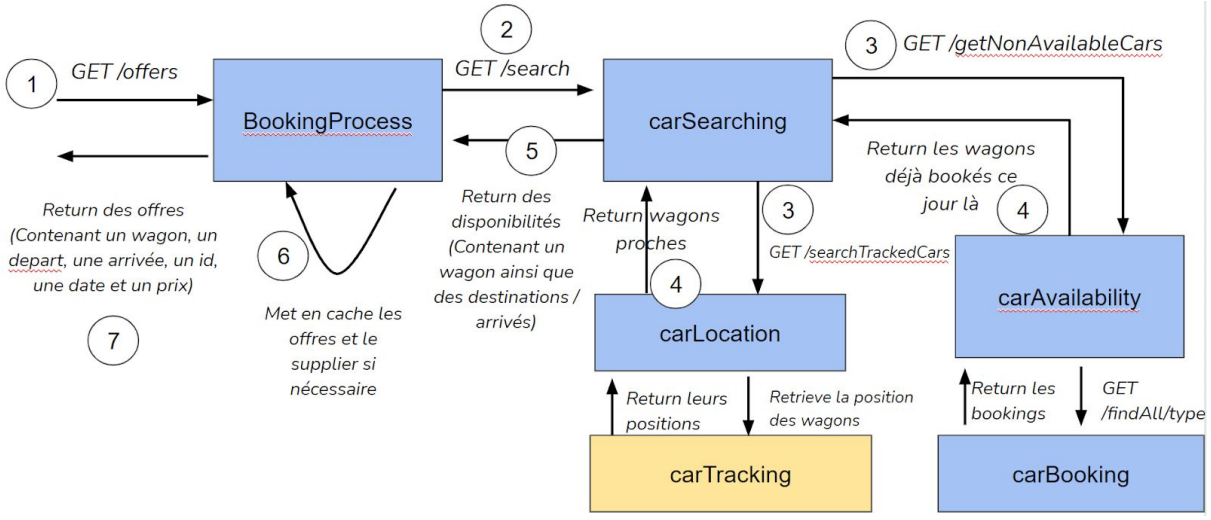
Ce dernier service à pour but de retourner une liste de wagons disponibles pour permettre à BookingProcess de proposer des offres de billets. Le service contactera CarLocation pour récupérer les wagons à proximité d'un nœud, et grâce à CarAvailability il saura retirer les wagons déjà réservés.

Interfaces

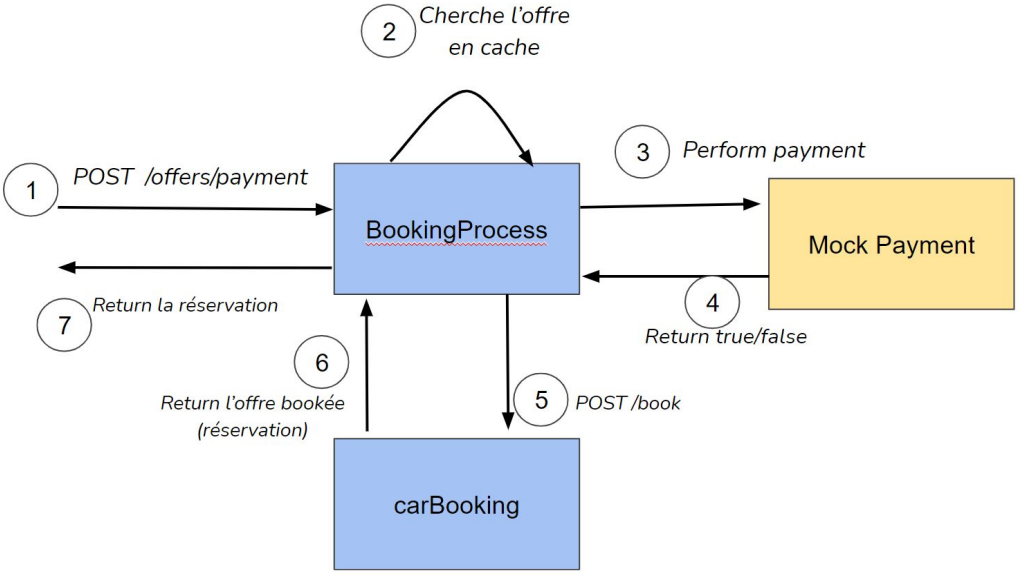


Workflows

Recherche des offres



Payer une offre



Réflexion

Vers quel type d'architecture se tourner ?

Dans un premier temps, nous pouvons nous poser la question suivante: Une architecture en service comme nous l'avons faite est-elle pertinente? En effet, nous sommes partis assez rapidement sur ce type d'architecture que nous maîtrisons bien et qui permet de séparer les responsabilités au sein du système. Cela s'est fait assez naturellement après une phase de conception initiale, qui nous a permis d'identifier nos composants.

L'architecture en services est notamment intéressante pour pouvoir utiliser plusieurs langages, étant donné que nos composants ont des rôles et des technologies très différentes (cela s'est d'autant plus avéré pour CarLocation, qui a nécessité une recherche technique des langages adaptés à Neo4j).

De plus, elle nous permet, sur le long terme, de faire passer à l'échelle les services indépendamment. Cela est intéressant dans notre cadre puisque certains services comme CarBooking seront moins sollicités que CarSearching par exemple, surtout si nous mettons en place un système de cache avancé.

Cependant, nous ne tirons pas pleinement parti de ce type d'architecture, étant donné que tous nos services sont liés "un à un" (nous n'avons pas de bus de messages au sein de notre architecture). Cela crée des dépendances assez fortes entre chaque service, mais ne nous empêche pas de faire des traitements asynchrones des données (notamment dans CarSearching) ni même de mettre en place de la compensation en cas d'erreur dans les services.

Une des possibilités serait d'implémenter un bus de messages dans notre architecture. Cependant, nous n'avons pas trouvé d'utilisation pertinente de ce genre de technologies pour ce projet. La piste la plus intéressante est de l'utiliser pour la recherche (CarSearching enverrait un message qui serait capté par CarLocation et CarAvailability), pour avoir un découplage plus fort, mais, dans ce cas, CarSearching devrait se souvenir des recherches en cours pour savoir quelle réponse de ces deux services correspond à quelle recherche effectuée. On pourrait aussi, de manière plus réaliste, l'utiliser en tant que file d'attente pour lancer des recherches, mais un bus de messages n'est peut-être pas adapté pour une simple queue d'événements entre deux services.

Une autre possibilité est d'évoluer vers une architecture cloud complètement découplée. Chaque recherche serait alors placée dans une file d'attente, et seul BookingProcess devrait attendre des réponses et envoyer des messages dans le cloud. Cela serait cependant beaucoup plus coûteux, en termes de temps et d'argent (Dans le cadre de notre projet étudiant où faire tourner un système sur nos machines ne coûte rien. En pratique, on réduirait les coûts de maintenance du système.), mais aurait des avantages certains, notamment au niveau de la scalabilité et de la résilience.

Quels moyens de communication pour nos services ?

En l'état actuel, nos services communiquent tous en HTTP, ce qui implique un fort couplage entre eux et complique l'extensibilité de notre architecture notamment si l'on souhaite rajouter une étape à notre processus de recherche d'offres.

Pour certaines étapes du système, il serait intéressant d'utiliser un autre protocole de communication. En effet, nous pouvons penser à l'utilisation du RPC lors du paiement de la réservation. Le RPC a la particularité d'ajouter un fort couplage dû au fait qu'il n'expose non pas des ressources mais des méthodes. Le fait d'utiliser ce protocole permettrait alors de lancer la méthode de paiement avec un contrat fort en respectant les différents types des paramètres et ainsi éviter tout problème de typage des données.

Il serait envisageable avec l'implémentation d'un bus de message, de faire de l'event sourcing et donc d'utiliser les évènements pour faire communiquer nos services entre eux. Cela pourrait nous permettre notamment de réduire le couplage et d'accueillir plus facilement les futures extensions de l'architecture.

Comment stocker nos données ?

Lors de la conception initiale du projet nous avons décidé d'utiliser mongoDB pour stocker nos réservations, les différents nœuds et les informations les concernant. Cependant, nous nous sommes vite rendu compte que cette décision était bancale car notre structure de données pour chacun des objets métier ne changerait pas ainsi que pour le fait que nos données sont plutôt liées entre elles, et qu'un système relationnel serait plus pratique grâce au respect des propriétés acides. Nous avons donc mis en place deux bases de données, une relationnelle sous Postgre permettant de stocker les réservations et les clients, ainsi qu'une base de données Neo4j orientée graphe pour gérer la localisation des différents nœuds.

L'utilisation de Neo4j s'avère très pratique car elle nous permet facilement et de manière performante de récupérer des nœuds proches de la localisation des wagons ou d'un autre nœud. Cependant, nous nous demandons si le choix de Postgre s'avère réellement judicieux. En effet, bien que Postgre puisse supporter une certaine charge, si le nombre d'utilisateur augmente trop Postgre ne sera pas en mesure de suivre en terme de performances. Cela s'avère problématique car le fait de distribuer une base de données Postgre sur plusieurs serveurs est assez compliqué.

Pour pallier ces désavantages, il serait intéressant d'utiliser une base de données distribuée comme Cassandra. Bien sûr, cela dépendrait du volume de données que l'on devrait stocker (selon les SLA, combien de temps on stocke l'historique des réservations, et combien de réservations quotidiennes nous avons). Cela permettrait d'accroître grandement la résilience du système, car, actuellement, la base de données Postgre représente un point critique dans notre architecture.

Quels langages utiliser pour notre système ?

Pour s'adapter à notre architecture orientée services et pour mieux effectuer les appels asynchrones, nous avons choisi d'utiliser Golang pour la majorité des services. En effet, ce langage est adapté dans ce contexte, et est facile d'utilisation, concis et performant.
Pour CarLocation, nous avons dû utiliser Javascript pour des soucis de compatibilité.

Nous pouvons cependant nous demander si l'utilisation de ce langage est pertinente dans tous les services. La question est d'autant plus intéressante dans le cas de CarBooking, qui doit interagir avec une base de données PostgreSQL et qui ne gère pas beaucoup d'asynchronicité. Nous avons utilisé Golang avec l'ORM go-pg, cependant cette technologie a posé quelques problèmes d'optimisation et de fonctionnalités. Au regard de ce point-ci et de la possible non-pertinence du langage Go dans le cadre de ce service, il pourrait en effet être intéressant d'en choisir un autre, comme par exemple Javascript avec Node pour la facilité d'implémentation ou même Java avec JDBC pour l'ancienneté et la fiabilité.

Comment déployer notre application ?

Dans ce projet, il nous a paru évident d'utiliser Docker pour déployer nos services. Premièrement, on n'a pas à se soucier de changements éventuels de ports ou autres variables, car on peut utiliser les variables d'environnement. Ensuite, pour lancer l'application, un docker-compose s'avère indispensable, d'autant plus lorsqu'on a des services. Enfin, le networking est facilité, notamment grâce à Traefik que nous utilisons en ce sens (il est vrai que cette technologie peut faire bien plus que du networking cependant).
La concurrence à Docker est très minoritaire, donc la piste de réflexion côté déploiement est plutôt bloquée sur cette technologie. Dans tous les cas, nous ne pensons pas remettre en cause ces choix.

Description des chantiers techniques

- Il faudrait voir pour remplacer le stockage des CarBooking dans une base de données Postgres avec une autre techno/technique (cache, S3, Cassandra, etc.)

- Plus globalement revoir notre manière de stocker (quoi stocker, où, pourquoi, comment) et plus important encore : revoir notre manière d'accéder au stockage (ORM, librairie, qui accède à quoi et où, etc.)

- Ajouter des caches pour limiter les appels au sein de notre système (tous les services sont concernés).

- Tenir compte de la concurrence de réservation qui changerait l'architecture de CarBooking et BookingProcess et de leurs communication

- Faire un état des lieux approfondi de la communication inter service/component (protocoles utilisés et descriptions interfaces)

- Conception, description et réalisation du déploiement (avec terraform par exemple) de tous les services sur une infra cloud par exemple

- On pourrait aussi en complément réfléchir à une traduction de l'architecture et de ses services sur un environnement distribué, notamment Kubernetes, pour simuler un déploiement "concret"

- Cela permettrait aussi de mettre en pratique notre architecture designée pour être parallélisable en mettant en place et testant la scalabilité de nos services (avec en plus l'ajout de load balancing plus ou moins intelligent, la conversion de nos services en full stateless). Il faut notamment réfléchir sur la parallélisation du point critique du projet : BookingProcess (qui n'est pas scalable/parallélisable)

- On a également comme piste technique d'amélioration de passer nos services en microservices en redécoupant de manière plus fine et surtout en ajoutant une couche non négligeable de résilience et de tolérance aux erreurs. À voir si cela est possible/envisageable en fonction des nouvelles US et fonctionnalités ajoutées au 2eme bimestre

- On devra retravailler CarSearching pour qu'il puisse effectuer des appels asynchrone vu qu'il en est “architecturalement” et “programmatiquement” capable de par le choix du langage et du workflow conçu

Si on avait su

- qu'on aurait eu quelques problèmes au niveau de la définition des objets entre chaque service, on aurait pris plus de temps à définir les DTO's entre nos services ensemble dès le début du projet.

- On aurait plus réfléchi et conçu pour avoir un système moins couplé qui nous aurait grandement facilité le développement en plus d'améliorer la qualité de la conception.

- On aurait donné plus d'importance au stockage qui est une part intégrante du projet et à la façon de le mettre en place

- que Golang ne propose pas d'ORM fonctionnel et complet pour Neo4j, on aurait implémenté directement CarLocation en Javascript.

- que mongoDB était un mauvais choix (pour des raisons de structuration des données), on aurait utilisé directement Postgre / une autre technologie comme nous en avons parlé précédemment plutôt que de faire du code pour mongoDB et ensuite switcher vers Postgre.