API Service Lab

Equipe A :

- Cancela Vaz Joël, joel.cancela-vaz@etu.unice.fr
- Deslandes Alexis, alexis.deslandes@etu.unice.fr
- Rainero Pierre, pierre.rainero@etu.unice.fr
- Costa Renaud, renaud.costa@etu.unice.fr

Architecture choisie

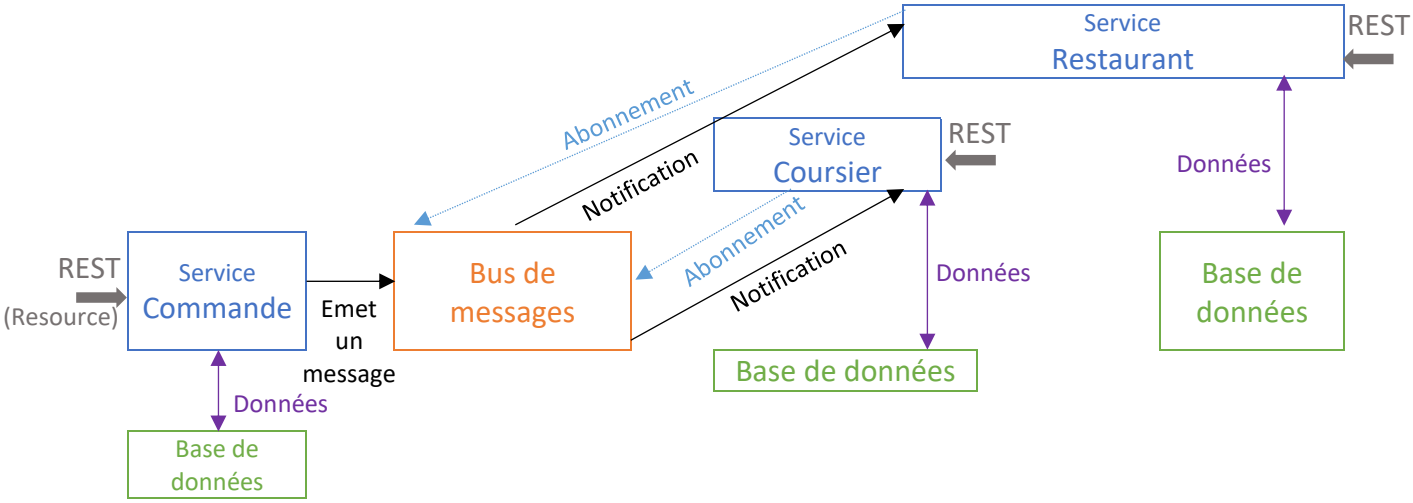

Figure 1 - Architecture mise en place

Comme l'illustre la Figure 1, nous avons opté pour une architecture mêlant les styles de service *Document* et *Resource*, les raisons de ce choix sont les suivantes :

- Si le service du restaurant ne répond plus, sans le style de service *Document* représenté ici par l'utilisation du bus de messages, la commande serait perdue et le restaurant n'aurait jamais l'information d'une nouvelle commande à préparer. Avec le bus de messages, la commande sera prise en compte une fois que le service sera de nouveau disponible.
- Les services « restaurant » et « coursier » se basent sur un système de « notifications », on peut imaginer qu'un écran affiche les commandes à préparer/livrer. Dans ce cas un modèle *RPC* ou *Resource* n'aurait pas été pertinent car il aurait fallu rappeler à intervalle régulier ces services.
- Les interactions utilisateurs (commande et validation d'une livraison) ne nécessitent pas d'abonnements. De plus, le métier se base sur des échanges de données (des créations de commandes, des recherches de commandes, des recherches de plats), une représentation *Resource* est pertinente (et facilement identifiable) pour ces services.

Forces de cette architecture

L'avantage de cette architecture, est de bénéficier des forces des styles de service *Document* et *Resource*. Le bus de messages permet une scalabilité souple de l'architecture. On bénéficie de la conservation de l'informations même en cas d'interruption des services appelés (sous condition que le bus de message ne tombe pas). Ceci n'est pas possible avec un style de service *Resource* ou *RPC*. En ce qui concerne le style de service Resource, il nous permet des traitements de données simplifiés tant pour les recherches que pour les ajouts, il est également agréable pour interagir avec « l'extérieur » comme des interfaces graphiques qui pourront mettre en formes les informations exposées par les API.

Faiblesses de cette architecture

Chaque service étant séparé, on a une certaine duplication de plusieurs éléments présents dans les services : les bases de données que l'on retrouve dans chaque service et qui ne sont pas partagées entre les différents services, les entités qui peuvent être communes également. On obtient alors de la duplication de code entre les différents services. Pour de nombreux appels à la base de données simultanés, on observe un problème. En effet, les bases de données sont à accès unique, seulement un utilisateur peut y accéder simultanément. Même si on a une base par service, on observe quand même des soucis d'étranglement autour de l'accès aux bases de données.

Modèle du domaine et APIs

Type de messages

- **NEWORDER** : émit par *orderService*, consommé par *coursierService* et *restaurantService*

  Exemple :

  {"address":"410 ch de chez moi","food":["plat1","plat2"],"type":"NEW_ORDER"}

APIs exposées

Service de livraison (coursierService) :

- /deliveries

  GET() : Retourne l'ensemble des livraisons à effectuer.

  PUT("/idDelivery") : Met à jour la livraison avec l'identifiant "idDelivery" pour la mettre dans l'état livré.

  POST() RequestBody : OrderDTO : Enregistre une commande dans le système pour les redistribuer aux coursiers par la suite.

Service de restauration (restaurantService) :

- /restaurants/orders

  GET() : Retourne l'ensemble des livraisons des resturants.

- /restaurants/new_order

  POST() : RequestBody : NewOrder : Réceptionne une nouvelle commande

Service de commande (orderService) :

- /orders
  - POST (content-type: JSON, encoding : UTF-8), body : OrderDTO
    - Permet d'ajouter une nouvelle commande
  - PUT (content-type: JSON, encoding : UTF-8), body : OrderDTO
    - Permet de valider ou non une commande en changeant son statut ("state")
- /meals
  - GET (content-type : JSON, encoding : UTF-8), queryParam : tag=AFoodTag
    - Permet de trouver tous les plats correspondant à un tag ("Asian" par exemple)

Architectures alternatives avec d'autres styles de service

Le service de commandes pourrait être fait avec un service de type RPC, cela imposerait un contrat fort mais un service de commandes pouvant être considéré comme ayant « peu » d'évolutions s'il est bien formalisé, pourrait être utilisé plus facilement que REST (grâce au WSDL notamment).