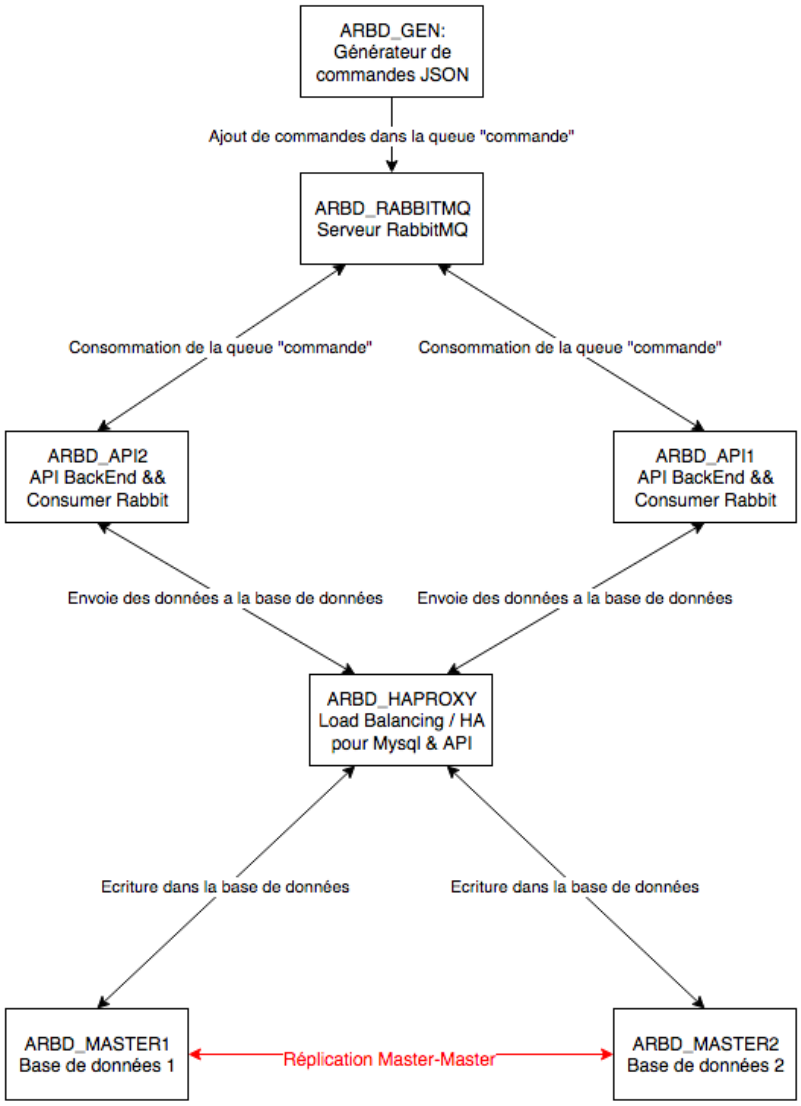
Compte Rendu Projet Speed Bouffe

Architecture globale du projet :

Le projet à été intégralement conçu avec Docker. A l'initialisation du projet, 8 containers Docker sont créés suivant le schéma ci-dessous :



- ARBD_GEN
Générateur de commandes en JSON

- ARBD_RABBITMQ
Serveur RabbitMQ pour garder dans une queue les commandes envoyées avant traitement par l'API.

- ARBD_API1
- ARBD_API2
Deux API et consumers de la queue RabbitMQ des commandes

- ARBD_HAPROXY
HAProxy qui sert à faire du Load Balancing et de la Haute Disponibilité sur les API mais aussi sur les base de données.

- ARBD_MASTER1
- ARBD_MASTER2
Bases de données répliquées en Master-Master

Schéma de l'écriture des données dans la base

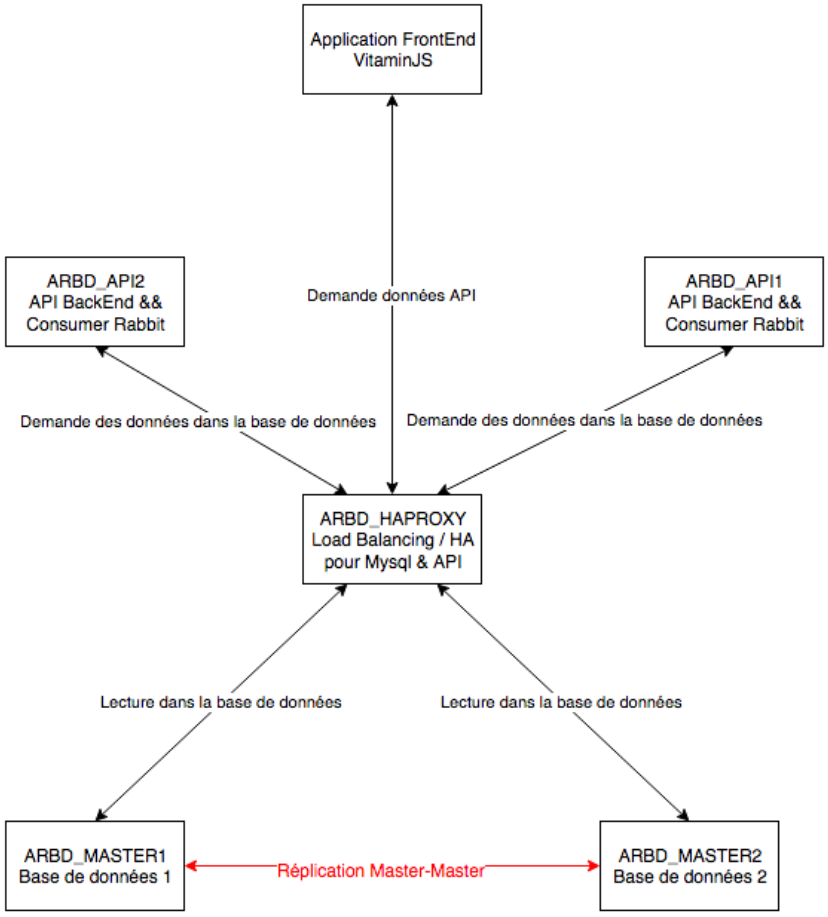


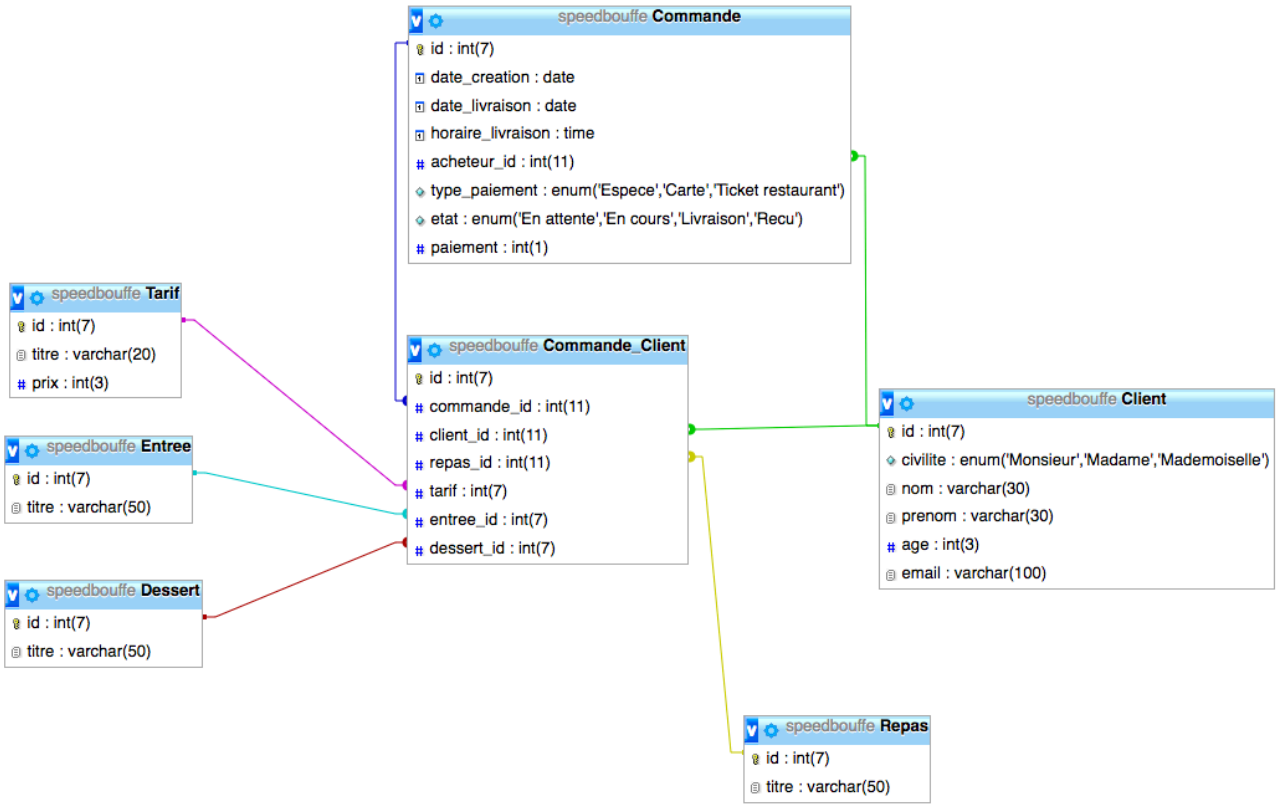
Schéma de lecture des données dans la base

Choix de l'architecture et optimisation

Nous sommes partis sur une architecture intégrée entièrement sur Docker afin que chacun de nous puisse travailler sur sa partie depuis son poste. Docker permet aussi un redémarrage du service si celui-ci tombe. En créant un réseau virtuel privé Docker, les services se reconnaissent entre eux et, toute la configuration de l'architecture tiens en un fichier. Le paramétrage de la solution quand à elle a été scriptée pour que lors du démarrage du projet, nous ayons un environnement fonctionnel. Toute la réplication, lancement des scripts de démarrage des services sont dans un script de démarrage mettant en moins de 3 minutes le projet en route. Seule l'application frontend doit être lancée à la main.

En cours de projet, de nombreuses améliorations ont été effectuées sur l'architecture tel que la mise en place de la queue RabbitMQ qui, si une donnée n'est pas écrite dans la base de données permet de redistribuer cette donnée dans la queue pour que la donnée ne soit pas perdue. A l'origine du projet, une seule base de donnée était disponible, une seconde avec une réplication Master-Master a été configurée et accessible sur le HAProxy pour faire en sorte que si une base tombe, l'autre peut prendre le relai rapidement.

Structure de la base de données



Structure de la base de données

Nous avons fait en sorte d'avoir une structure de base de données la plus simple possible afin de faciliter l'enregistrement des commandes. Nous avons également apportés quelques modifications au cours du projet à la suite des différents comptes-rendus, notamment pour le tarif des commandes. A partir des données envoyées par le générateur nous avons déterminés que les commandes contenaient plusieurs lignes de commandes représentant un repas pour un client. Nous avons donc créer une table Commande qui stocke les infos de base de la commande avec un lien vers un Client qui est l'acheteur de la commande, et des liens vers plusieurs « ligne de commande » représenté par la table Commande_client. Une ligne de commande est rattachée à un client (qui est le client à livrer), à un Repas, une Entrée et un Dessert (qui sont les choix du client). Pour ce qui concerne le tarif, nous avions imaginé au départ que chaque plat aurait son tarif propre mais finalement d'après le compte rendu nous avons modifié la structure afin de rattaché le tarif de la commande à la ligne de commande. La table Tarif contient donc les différents tarif applicable ainsi que leur prix.

Une fois notre structure validée et le MCD réalisé, nous avons extrait le SQL généré pour créer notre base de données SQL. Pour l'exploitation de ces données et la réalisation des statistiques nous avons choisi de créer une API en Silex avec l'utilisation de doctrine. Pour se faire nous avons utilisés le principe de reverse engineering qui nous a permis de créer les différentes entités doctrine. Notre API est très complète en terme de statistiques mais permet également si l'on souhaite de créer de nouveau repas, tarif etc..

Gestion de situation critique et scalabilité

Si une base de données tombe, le HAProxy va alors le voir directement et ne va envoyer tout ses appels que vers la base de données qui est encore fonctionnelle. Si jamais les deux bases de données venaient à tomber, les données ne seraient pas perdues car, elles restent dans la queue RabbitMQ et sont donc stockées.

L'architecture est scalable car, si on aperçoit que de plus en plus de données arrivent sur la queue de RabbitMQ, il est encore possible de déployer plus de consumer sur la queue soit dans les API qui sont déjà initialisées ou en rajoutant des containers Dockers API avec la même configuration que ceux qui sont lancés au lancement du script d'initialisation.
Pour ce qui est de la lecture et écriture dans les bases de données, la contrainte de grosse volumétrie de donnée a été palliée par la mise en place d'un HAProxy qui permet de dispatcher les requêtes sur deux bases et sur deux API différentes. La mise en place de RabbitMQ permet aussi de pouvoir traiter en mode asynchrone les informations qui arrivent en écriture sur les bases de données et donc de laisser les informations s'écrire correctement dans les bases de données.