# Proposition d’architecture

Nous allons partir sur une architecture en micro-services.
Pour rappel, l’architecture en micro-services a l’avantage d’être plus robuste, puisque les fonctionnalités sont dépendantes de services indépendants. Par exemple, si le service gérant le shopping est down, cela n'empêchera pas les utilisateurs de pouvoir se connecter.

## Infrastructure

Afin de conserver la main sur la scalabilité horizontale de notre application microservice, nous utiliserons Kubernetes ou Nomad pour orchestrer nos conteneurs docker et nos services. Ceux-ci seront répliqués selon les besoins liés au trafic réseau. Un répartiteur de charge réseau sera également implémenté à l’aide de cette technologie.

## Architecture des services

Voici l’ensemble des services que nous prévoyons de mettre en place:

- Service d'authentification.
- Service de commandes.
- Service de paiement.
- Service d’inventaire.

- Le service d’authentification sera chargé de la création et de la gestion des comptes utilisateurs(inscription et authentification) ainsi que des requêtes CRUD sur les comptes.
- Le service de commandes sera chargé de gérer les commandes et les livraisons. La base de données contiendra donc les adresses des utilisateurs finaux ainsi que la liste des produits achetés. Les acheteurs pourront avoir accès à la liste de leurs commandes et les vendeurs pourront avoir accès à la liste de leurs ventes au travers de la plateforme.
- Le service de paiement sera exclusivement dédié aux paiements des transactions.
- Le service d’inventaire permettra aux utilisateurs finaux de voir l’intégralité des produits mis en vente et de les sélectionner pour les positionner dans le panier. Il permettra également aux vendeurs d’effectuer des requêtes CRUD sur les produits.

Chaque microservice sera doté de sa propre base de données. L'intérêt d'une architecture microservice est de disposer de services faiblement couplés (peu d'échanges d'informations entre ces services), évolutifs du fait de leur étanchéité et indépendants les uns des autres.
Chaque microservice doit donc posséder en théorie sa propre base de données dont les tables sont adaptées aux besoins du service. Les autres services ne doivent pas accéder aux données qu'ils ne possèdent pas ou dont ils n'ont pas besoin pour être fonctionnels.
Sur l'architecture que nous proposons, les modifications apportées à une seule base de données n'ont pas d'incidence sur les autres bases de données ou autres services. Ainsi, il n'y a pas un point de défaillance unique dans l'application qui serait susceptible de faire crasher l'intégralité de notre application. Celle-ci est donc plus résiliente. Nous pourrons également employer plusieurs technologies de base de données selon les besoins en fonction des exigences et fonctionnalités du service concerné.
En contrepartie, nous acceptons le fait que nous risquons de rencontrer des problématiques de communication inter service et de débogage complexe sur la gestion des données réparties sur les différents services (réplication de données entre les services scale de manière horizontale).
Pour palier à ce problème nous utiliserons un aggrégateur qui se chargera de respecter la logique de liaison des données partagées entre les différents services et leur base de donnée respective. L'agrégateur appellera chacun des services requis lors d'une requête effectuée par un acheteur ou un vendeur.

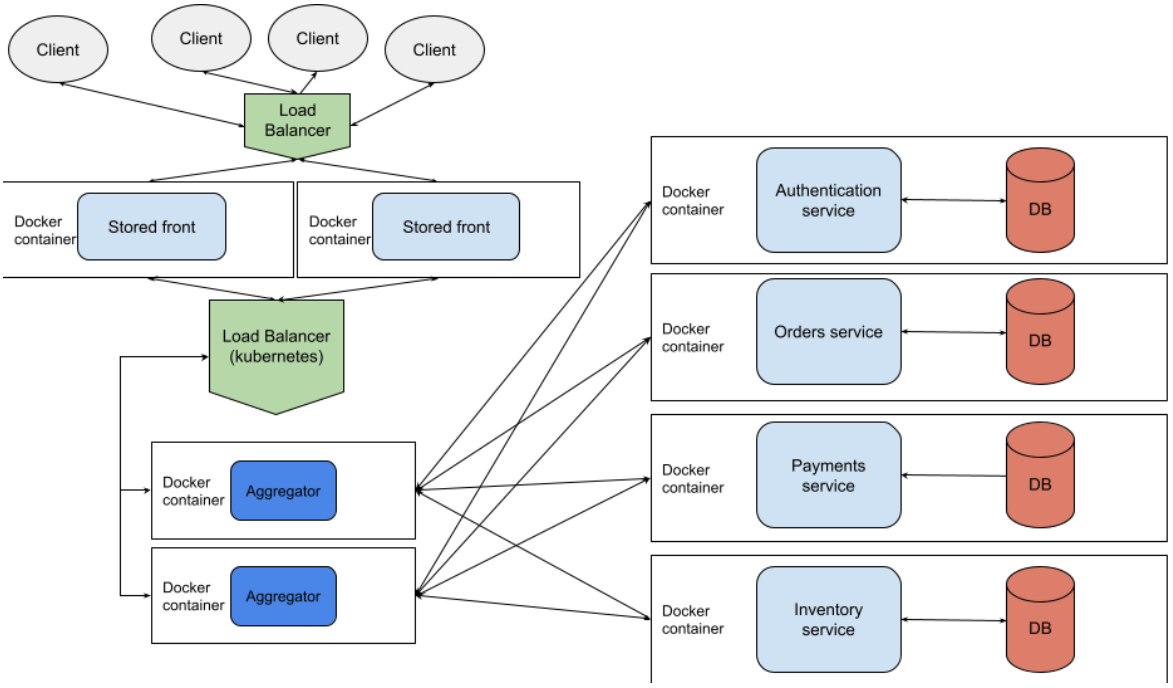


*Schéma représentatif de notre architecture microservice pour Mylittleshopping.*

## Mailing:

Notre architecture comportera des outils de mailing afin de contacter les clients qui font usage de nos services. Nous utiliserons plus spécifiquement le service proposé par mailjet avec sa propre API de mailing. Nous n'auront donc pas de service de mailing dédié à l'intérieure de notre architecture mais nous ferons des appel api vers le service de mailjet au sein du service d'authentification (lorsqu'un utilisateur s'inscrit et vérifie son compte) et au sein du service de livraisons (afin que l'utilisateur soit en mesure de suivre sa commande.

## Protocole de communication :

Pour ce qui est de la transmission de donnée nous partirons sur des API Rest ( protocole HTTP )

## Format de communication :

Concernant le format de communication , nous privilégions le format JSON car le format est léger, flexible et pris en charge par quasiment tous les langages.

## Gestion de logs :

La gestion de logs pour les microservices est beaucoup plus complexe que dans une architecture monolithique vu que chaque microservice conserve son propre lot de logs. Pour être efficace , il faut que chaque microservice puisse précisément  centraliser les logs dans un seul dépôt.

## Documentation des interfaces :

Nous allons utiliser OpenAPI pour définir les spécifications de nos différents microservices, pour ensuite pouvoir de manière simple appeler les différentes fonctionnalités (génération de SDK à partir de fichier spec pour utiliser des clients au lieu de faire des requêtes HTTP brutes). Cela nous permettra aussi de pouvoir avoir une documentation pour les développeurs.

## Backups :

Au sujet des backups , on retrouve généralement deux niveaux de backup.
Le premier niveau consiste à sauvegarder l'état de tout le conteneur du service. C'est la solution la plus directe.

Le second consiste, à l'aide des outils proposés par les clients de base de données à générer un script sql de création des tables et des données existantes à n'importe quel moment. Le script peut ensuite être exécuté dans la même instance (ou une autre) pour revenir à l'état sauvegardé. L'opération est cependant coûteuse en temps surtout dans le cas des full backup.

Nous avons l'intention d'implémenter une système de back-up incrémental à fréquence quotidienne, ainsi l'état de la database est sauvegardé et ajouté à l'état précédent tout cela en partant de la dernière full back-up comme on peut le voir sur le pattern suivant:

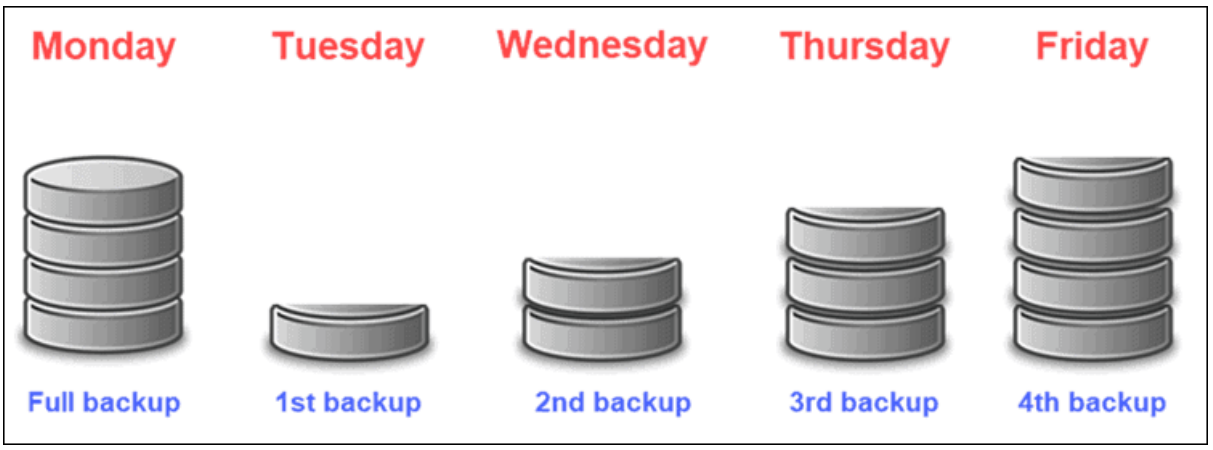


*Schéma représentatif d'un système de back-up incrémental de base de données.*

En cas de panne chez un hébergeur, l'infrastructure peut être remontée en quelque minutes en utilisant un outil comme Terraform où il suffira de changer de provider

## Répartition des tâches:

- Alex

Développement des microservices d'authentification et d'inventaire.
Développement de la logique applicative liées à ces deux services dans l'agrégateur.

- Rayane

Développement des microservices de commandes et de paiement..
Développement de la logique applicative liées à ces deux services dans l'agrégateur.

- Thomas

Dockerisation des services et montage en volumes.
Mise en place des répartiteurs de charge réseau et de l'orchestrateur de conteneurs pour maintenir la scalabilité de l'application.

- Tâches communes:

Maintenir à jour la documentation swagger.
Maintenir une collecte propre et stable des logs pour chaque service (outil logstash).