

Université Paris Descartes

# Projet Web / Distribué

Architecture micro-Service déployé sur google Cloud

Wacim Belahcel  
01/05/2020

Encadrant : Benoit Charroux



## Table des matières

1. Introduction .....	3
2. Instruction pour lancement sur VM / Cloud : .....	4
3. Développement web : .....	5
Description de l'application : .....	5
Architecture : .....	5
Back-end : .....	6
Micro-service RESTful : .....	6
Registry eureka : .....	8
Gateway Zuul : .....	10
Bases de données : .....	11
Frontend : .....	12
Menu : .....	12
Ajouter / Modifier : .....	13
Components liste: .....	14
Details : .....	14
4. Déploiement distribué : .....	14
Docker .....	14
Kubernetease : .....	16
Architecture : .....	16
Petit Obstacle : .....	19
5. Conclusion .....	20

# 1. Introduction

Ce rapport a pour but de présenter le travail que j'ai effectué pour les deux projets : web et distribué, ayant déjà fait beaucoup de web lors de ma formation (J2EE, PHP Laravel/Symfony, Nodejs) j'ai préféré m'attarder sur l'aspect distribué, je tacherai dans un premier temps de décrire les fonctionnalités de mon application web ainsi que son architecture.

Dans une seconde partie, je parlerai des du déploiement de l'application en utilisant docker et kubernetes sur deux environnements différents ( VM linux / Google Cloud Kuberenetes), le fichier yaml ainsi que certaines configuration sont un peu différentes selon les deux cas.

Je finirai par aborder un obstacle que je n'ai malheureusement pas eu le temps de régler (car je m'en suis rendu compte que récemment) qui n'a pas trop d'influence sur la notation mais qui a tout de même son importance lors du déploiement cloud.

Le travail présenté respecte la totalité des conditions imposées lors de la définition du projet.

## 2. Instruction pour lancement sur VM / Cloud :

**Pour le lancement sur vm**, il suffit de lancer :

```
kubectl apply -f deployment-mininventaire.yaml
```

il faut ensuite vérifier si une adresse ip a été attribué au contrôleur ingress avec :

```
kubectl get ingress
```

si c'est le cas, ajouter l'ip au fichier /etc/hosts avec les adresse : « frontend.localhost » et « backend.localhost », le site web sera accessible via « frontend.localhost ».

**Pour un lancement sur google cloud**, il faut d'abord créer une adresse ip global grace à la commande :

```
gcloud compute addresses create myglobalingress-ip --global
```

Ensuite utiliser le fichier yaml deployment-mininventaire-cloud.yaml, le site web sera accessible via l'adresse IP attribué (get ingress).

### 3. Développement web :

#### *Description de l'application :*

L'application web est une application de gestion d'inventaire (gestion de stock) simple, l'utilisateur peut Créer / Modifier / Supprimer / Ajouter les éléments suivants :

- Des produits.
- Des clients
- Des Vendeurs
- Des ordres (ordre d'achat).

#### *Architecture :*

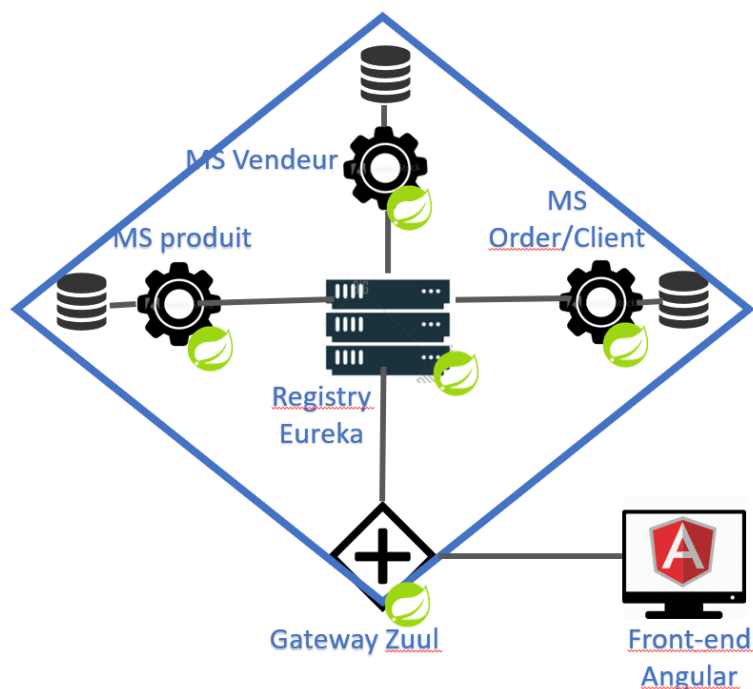


Figure 1 Architecture web / Micro Services

Comme on peut le voir sur l'image ci-dessus, l'architecture respecte une architecture micro-service (registre / gateway / loadBalancer), nous avons 3 micro services :

- Produit : contient les méthodes CRUD des produits.
- Vendeur : CRUD Vendeur.
- Client / Order : Crud des clients et des orders (car leurs models sont lié).

Chaque micro-service possède sa propre base de données SQL **persistante**, de plus chaque micro-service est un Eureka client, qui s'enregistre sur un (**ou plusieurs** car le déploiement et la configuration a été pensé pour pouvoir le faire si nécessaire).

Une gateway (ou plusieurs) a aussi été mis en place, cette dernière est un Eureka client, elle possède aussi la fonctionnalité eureka discovery afin de pouvoir accéder à l'ensemble des micro-service depuis le service Eureka.

Enfin, le front Angular communique avec el back via la gateway, de plus, pour le déploiement sur cloud, il fallu mettre en place un serveur proxy au niveau du serveur front, j'expliquerai cela plus en détails sur la partie déploiement.

## Back-end :

Comme cité précédemment il y'a en tout 5 composants au niveau du backend.

### Micro-service RESTful :

Les trois micro-service possède une architecture et une configuration similaire :

- Un modèle : qui constitue le point de liaison entre l'ORM JPA et nos micro-services
- Une classe repository qui possède les fonctions CRUD permettant de modifier la base de données à l'aide d'un entity manager.
- Une classe service : qui possède les fonctions qui effectue les différentes tâches CRUD.
- Une classe Controller qui représente les chemins de notre service REST.
- Une classe main permettant de lancer l'application

Voici les diagramme UML des différents micro-service, Les attributs des modèles représentent les colonnes des tableaux en base de données.

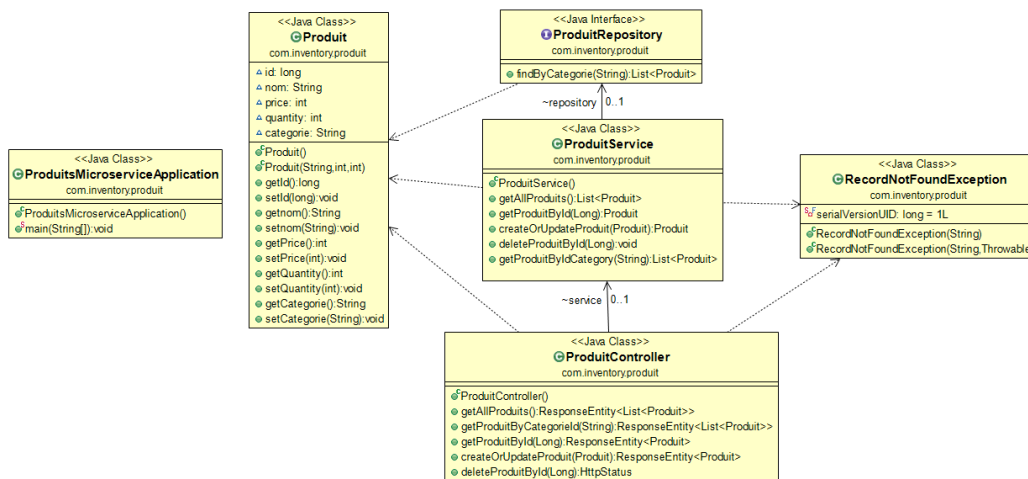


Figure 2 Diagramme de classe micro-service produit, model : Produit

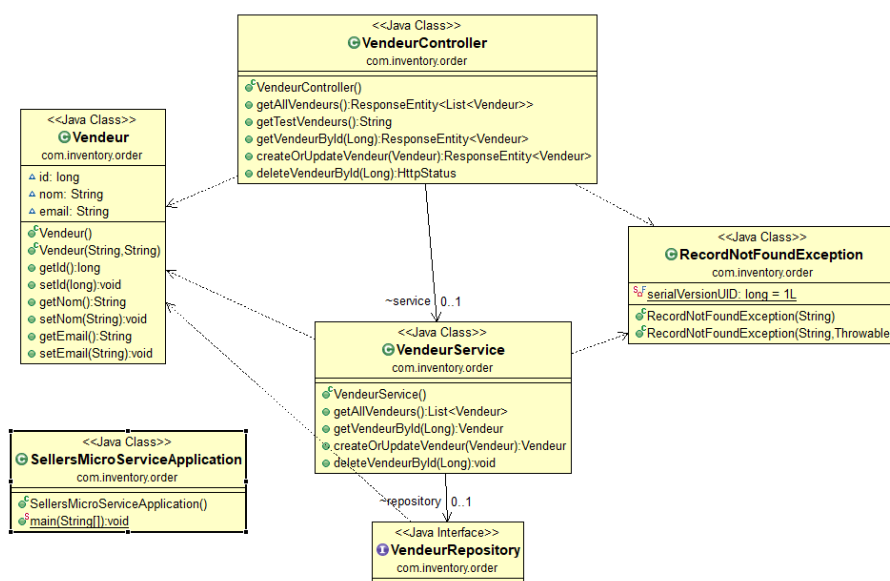


Figure 3 diagram de classe micro-service vendeur, modèle : vendeur

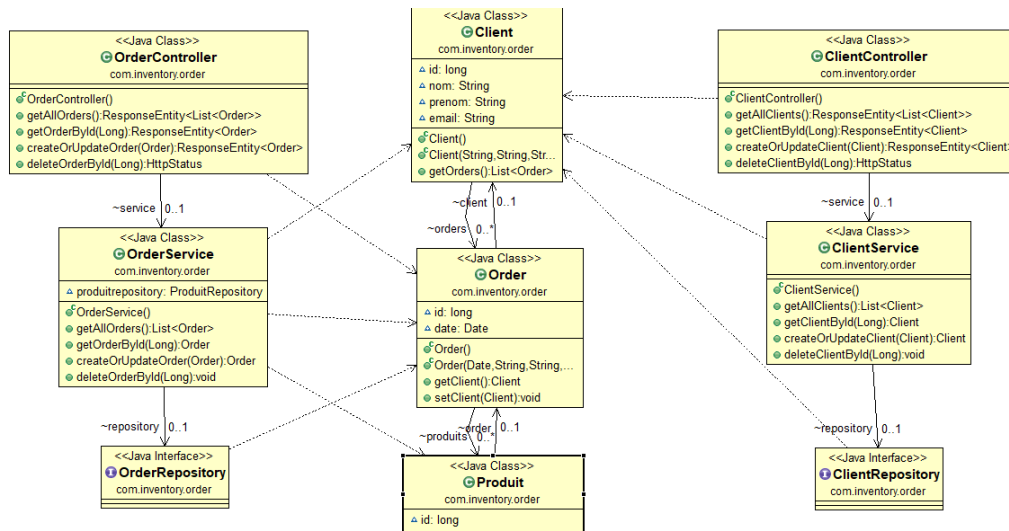


Figure 4 diagramme de classe du micro-service Order

Le diagramme le plus élaboré est celui du micro service « Order » car il possède 3 modèles : Order, client, produit. chaque client peut avoir plusieurs orders (one to many), chaque order peut avoir plusieurs produits (one to many ).

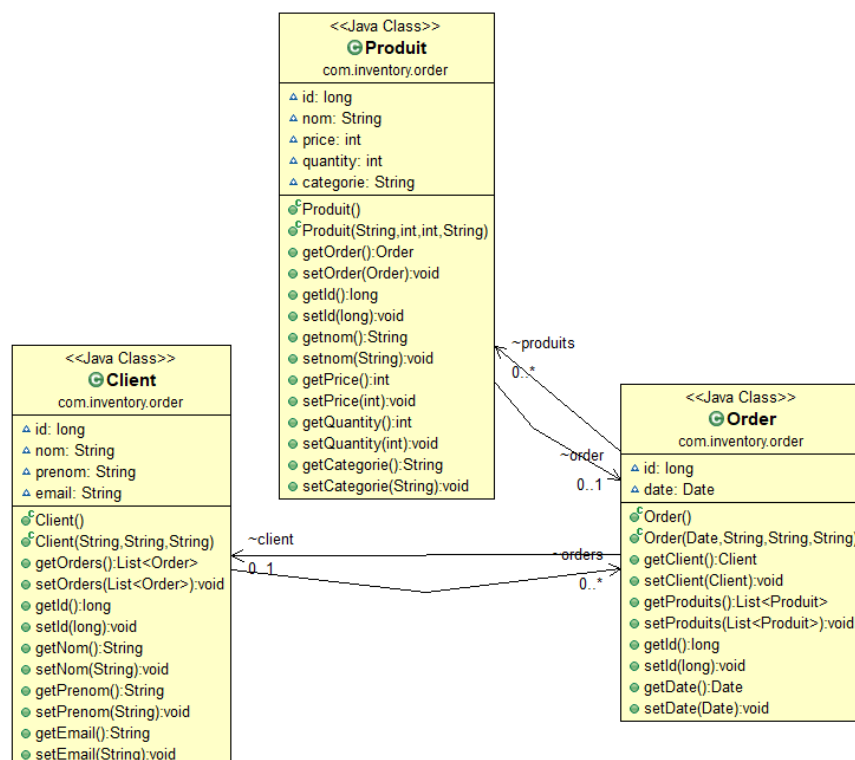


Figure 5 Classe modèle du micro-service Oder

Chaque micro-service possède la même configuration, tout d'abord la configuration de la base de données :



```

1
2 spring.h2.console.enabled=true
3 spring.h2.console.path=/h2_console
4 spring.datasource.url=jdbc:h2:file:~/h2/orderdb
5 spring.datasource.username=sa
6 spring.datasource.password=
7 spring.datasource.driverClassName=org.h2.Driver
8 spring.jpa.hibernate.ddl-auto = update
9 spring.jpa.show-sql=true

```

Figure 6 Configuration micro services BDD

Cette configuration permet de transformer la base de données en une base de donnée persistante, de plus le dossier « ~/h2 » sera monté sur un dossier local sur la machine qui lancera le script kubernetes à l'aide de PersistentVolumes.

A cela s'ajoute la configuration du serveur et du eureka client :

```

server:
  port: 9094

# Remember that the name of the application has to be the same name of the registered service in the service discovery tool
spring:
  application:
    name: orders-service

eureka:
  instance:
    # Leave prefer ip address to allow the gateway inside the kubernetes cluster to find this service by it's pod ip
    preferIpAddress: true
    # The hostname of the service, to register the pod and turn it easier for the gateway to find it
    hostname: orders-service
  client:
    registerWithEureka: true
    fetchRegistry: true
    serviceUrl:
      defaultZone: http://eureka-0.eureka.default.svc.cluster.local:8761/eureka

```

Figure 7 Configuration Micro service

Au niveau de « defaultZone », on peut remarquer, le lien du serveur eureka qui correspond au lien du serveur eureka sur une configuration kubernetes (lien du service eureka serveur), nous pouvons

ajouter plus de serveurs si

nécessaire en ajoutant le lien de ces derniers.

## Registry eureka :

Le serveur eureka ne possède qu'une classe main ainsi que les différentes annotations nécessaires afin de le définir comme étant Serveur registry, voici la configuration du fichier yaml :

```

1 server:
2   port: 9094
3
4 # Remember that the name of the application has to be the same name of the registered service in the service discovery tool
5 spring:
6   application:
7     name: orders-service
8
9 eureka:
10  instance:
11    # Leave prefer ip address to allow the gateway inside the kubernetes cluster to find this service by it's pod ip
12    preferIpAddress: true
13    # The hostname of the service, to register the pod and turn it easier for the gateway to find it
14    hostname: orders-service
15  client:
16    registerWithEureka: true
17    fetchRegistry: true
18    serviceUrl:
19      defaultZone: http://eureka-0.eureka.default.svc.cluster.local:8761/eureka
20

```

Figure 8 Configuration Serveur eureka

Ci-dessous les réponses de quelques requête GET / POST / PUT / DELETE (Delete n'affiche rien) , le principe est le même (bien que les fonctions sont gérer un peu différemment) pour tous les micro services.

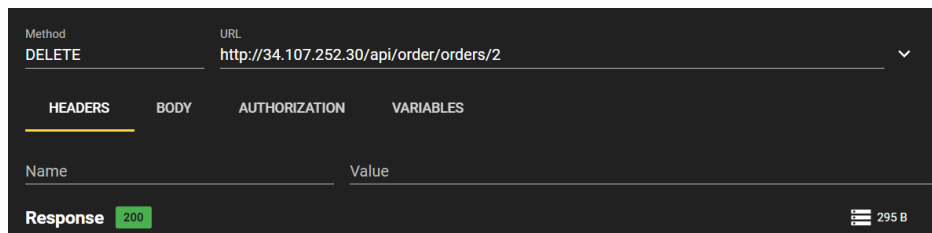


Figure 9 Delete d'un order reponse 200 OK

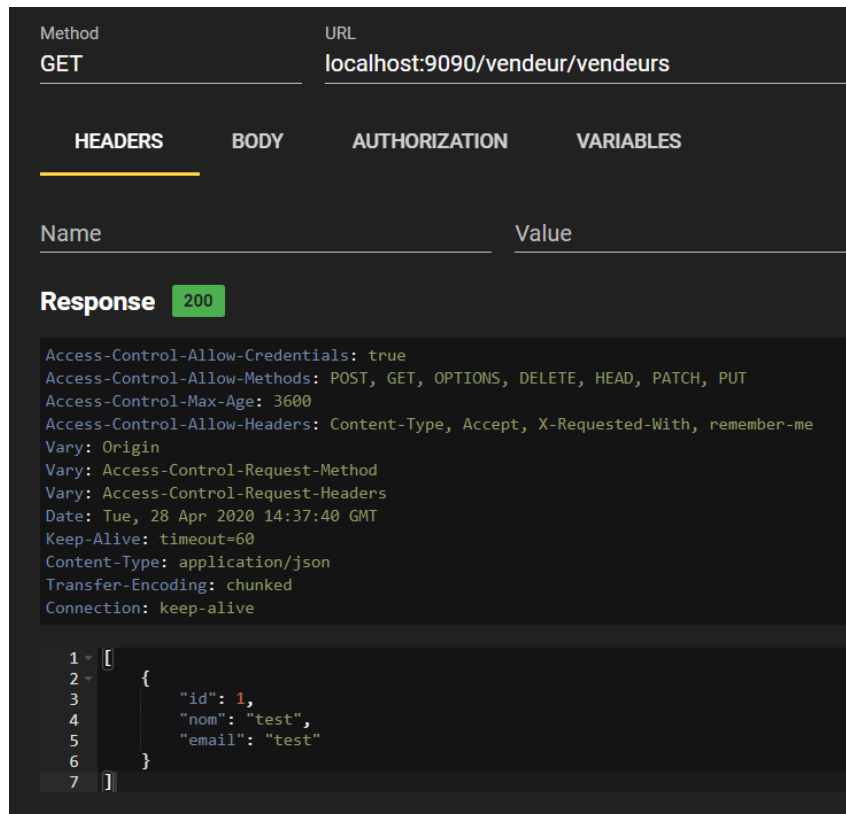


Figure 10 GET d'un vendeur

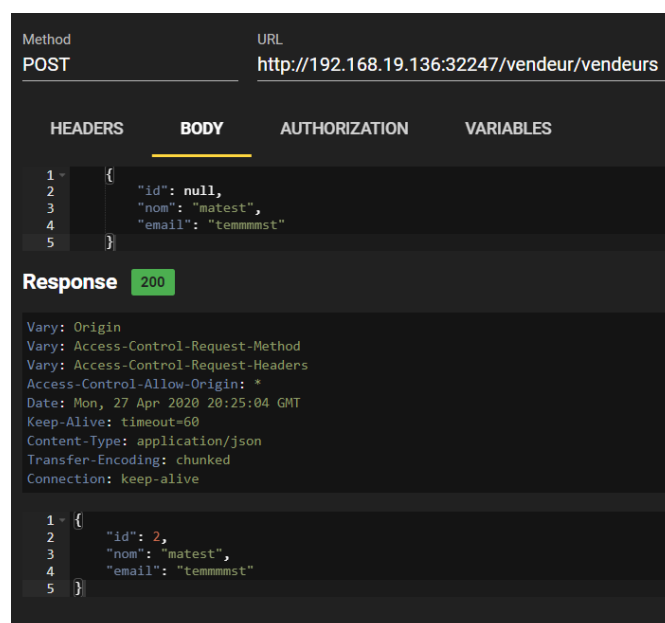


Figure 11 POST d'un vendeur

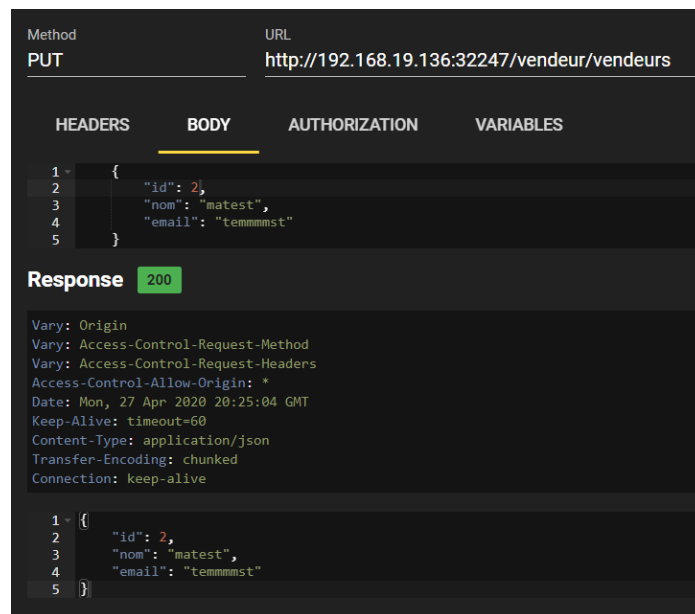


Figure 12 PUT d'un vendeur

### Gateway Zuul :

Le gateway est lui aussi très simple, on ajoute les annotations ainsi que la configuration nécessaire pour les CORS policies (CrossOrigin, DiscoveryClient, EurekaClient), et la configuration des différents chemins des micro services :

```

1 server:
2   port: 9090
3
4 spring:
5   profiles:
6     active: "dev"
7   application:
8     name: zuul-server
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

Figure 13 configuration zuul

## ***Bases de données :***

Les bases de données des micro-service sont des bases de données persistante H2 Base, les dossier ou sont contenu les données (~/.h2/..) sont monté sur disque local grâce à un PersistentVolume.

Il existe une base de données différente pour chaque micro service :

### ***BDD Produit :***

Possède une seul table Produit ayant comme colonne :

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
long id;

@Column(name="nom")
String nom;
@Column(name="price")
int price;
@Column(name="quantity")
int quantity;
@Column(name="categorie")
String categorie;
```

### ***BDD Vendeur :***

Possède une seul table Vendeur ayant comme colonne :

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
long id;
@Column(name="nom")
String nom;
@Column(name="email")
String email;
```

### ***BDD Order :***

Ayant 3 table, Client :

```
@GeneratedValue(strategy = GenerationType.IDENTITY)
long id;
@Column(name="nom")
String nom;
@Column(name="prenom")
String prenom;
@Column(name="email")
String email;
@OneToMany(mappedBy = "client")
@JsonIgnore
List<Order> orders;
```

Order :

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
long id;
@Temporal(TemporalType.DATE)
Date date;
@ManyToOne
@JoinColumn(name = "CLIENT_ID")
Client client;
```

Produit :

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
long id;

@Column(name="nom")
String nom;
@Column(name="price")
int price;
@Column(name="quantity")
int quantity;
@Column(name="categorie")
String categorie;
@ManyToOne
@JoinColumn(name="ORDER_ID")
@JsonIgnore
Order order;
```

## Frontend :

```
const routes: Routes = [
  { path: '', redirectTo: 'menu', pathMatch: 'full' },
  { path: 'clients', component: ClientsComponent },
  { path: 'addClients', component: AddClientsComponent },
  { path: 'editClients/:id', component: EditClientsComponent },
  { path: 'detailsClients/:id', component: DetailsClientsComponent },
  { path: 'produits', component: ProduitsComponent },
  { path: 'addProduits', component: AddProduitsComponent },
  { path: 'editProduits/:id', component: EditProduitsComponent },
  { path: 'orders', component: OrdersComponent },
  { path: 'detailsOrders/:id', component: DetailsOrdersComponent },
  { path: 'addOrders', component: AddOrdersComponent },
  { path: 'addSellers', component: AddSellersComponent },
  { path: 'editSellers/:id', component: EditSellersComponent },
  { path: 'sellers', component: SellersComponent },
  { path: 'menu', component: MenuComponent },
];
```

Figure 14 routes Front

Le front end Angular est reparties en 5 parties : Menu, Client, Vendeur, Order, Produit, l'accès à l'ensemble des fonctionnalité se fait via les routes défini sur module.routing et sont ajouté à la barre du menu.

## Menu :

Il contient un seul component, le menu de l'application :

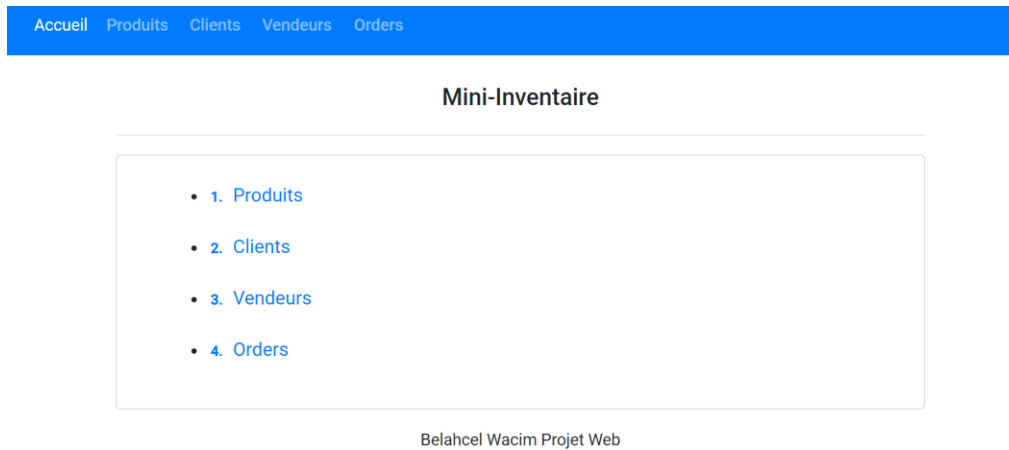


Figure 15 menu

### Ajouter / Modifier :

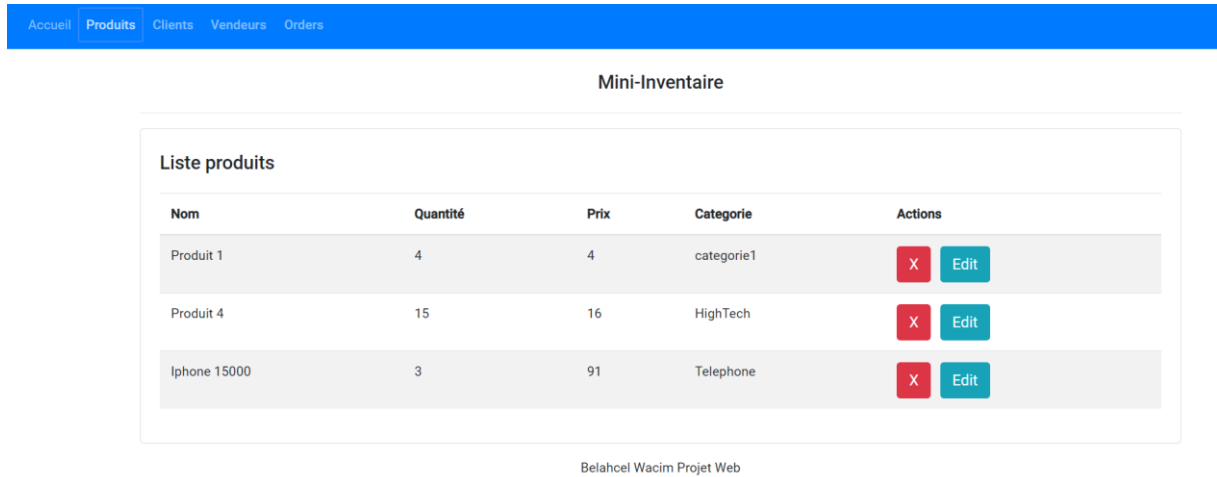
Il est possible d'ajouter et de modifier l'ensemble des modèles (produits / clients / vendeurs / orders), mis à part le composant order qui ne devrait pas être modifiable, et cela avec component formulaire (différent pour chaque composant selon ces attributs et ces besoins) :

Exemple avec le composant Order il est possible d'ajouter plusieurs produits à un order directement via l'interface d'ajout d'ordre de manière dynamique (formulaire dynamique):

Figure 16 Formulaire

### Components liste:

Il est possible d'afficher la liste de chacun des éléments de l'application grâce à des composants de type liste (des tableaux), exemple avec le tableau des produits, cette page permet aussi de supprimer un élément de la liste (mis à part le composant Order pour des raisons de sécurité, un chemin pour la suppression existe cependant au niveau de l'api Order DELETE : /order/orders/[id]) :



Accueil Produits Clients Vendeurs Orders

Mini-Inventaire

Liste produits

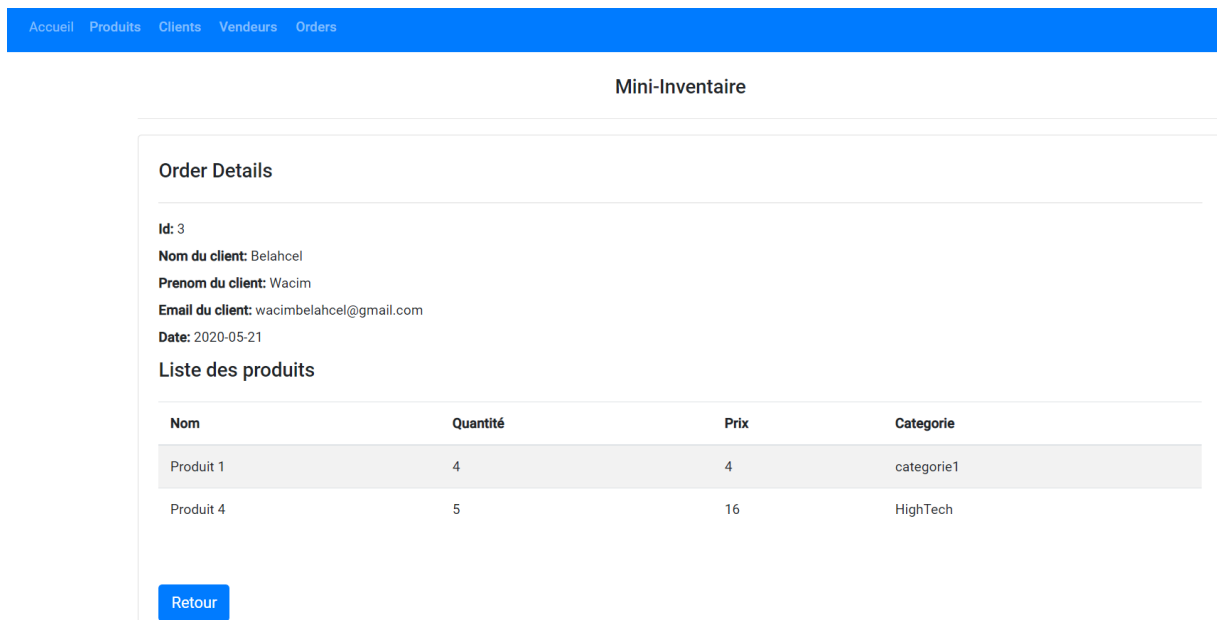
Nom	Quantité	Prix	Catégorie	Actions
Produit 1	4	4	categorie1	<span>X</span> <span>Edit</span>
Produit 4	15	16	HighTech	<span>X</span> <span>Edit</span>
Iphone 15000	3	91	Telephone	<span>X</span> <span>Edit</span>

Belahcel Wacim Projet Web

Figure 17 listes

### Details :

Enfin pour certains éléments il est aussi d'afficher les détails si nécessaire grâce à des composants « détails » :



Accueil Produits Clients Vendeurs Orders

Mini-Inventaire

Order Details

**Id:** 3  
**Nom du client:** Belahcel  
**Prenom du client:** Wacim  
**Email du client:** wacimbelahcel@gmail.com  
**Date:** 2020-05-21

Liste des produits

Nom	Quantité	Prix	Catégorie
Produit 1	4	4	categorie1
Produit 4	5	16	HighTech

Retour

Figure 5 details

## 4. Déploiement distribué :

### Docker

La création des images docker pour les services backend est assez simple, il suffit de build les projets et d'utiliser le Dockerfile suivant :

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
EXPOSE PORT DU MICRO SERVICE
ADD [CHEMIN DU JAR ] app.jar
ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/./urandom", "-jar", "/app.jar"]
```

Figure 19 Dockerfile : Spring boot

Grace à la commande :

```
docker build -t [pseudo-dockerhub]/[nom-image] . -f Dockerfile
```

```
docker push [pseudo-dockerhub]/[nom-image]
```

Pour le front end, deux images différentes ont été créées selon si le déploiement se fait sur le cloud ou sur machine virtuelle, car le déploiement sur le cloud nécessite de mettre en place un proxy au niveau du front afin de rediriger les requêtes du client vers le gateway, il suffit de décommenter la partie commentée du Dockerfile suivant afin de mettre en place le proxy :

```
FROM nginx:1.17.1-alpine
# COPY nginx.conf /etc/nginx/nginx.conf
COPY /dist/inventoryfrt /usr/share/nginx/html
```

Figure 20 Dockerfile Angular

Et voici-ci-dessous une partie du fichier qui configure le proxy nginx au niveau du front-end:

```
server {
    listen 80;
    listen [::]:80;
    location /api/vendeur/ {
        proxy_pass http://zuul-server.default.svc.cluster.local:9090/vendeur/;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
    location /api/order/ {
        proxy_pass http://zuul-server.default.svc.cluster.local:9090/order/;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
    location /api/produit/ {
        proxy_pass http://zuul-server.default.svc.cluster.local:9090/produit/;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
    location / {
        root /usr/share/nginx/html;
        index index.html index.htm;
    }
}
```

Figure 21 proxy

Le proxy est nécessaire car lors d'un déploiement sur machine distante, le client qui recevra les pages Angular reçoit aussi les liens du gateway backend (par exemple `zuul.default.svc.cluster.local`), cependant ce lien n'a plus aucun sens du point de vue de l'utilisateur, ainsi en utilisant un proxy, je suis capable de modifier le code afin de rediriger toutes les requêtes sur des chemins `/api` vers ma gateway, c'est la solution la plus simple à mettre en place.

Bien sur Google cloud propose d'autres solutions plus compliquées qui aurait pris du temps à prendre en main dans le cadre de ce projet.

Et voici ci-dessous les images Docker sur [Docker-hub](https://hub.docker.com/) :



warra29	Search by repository name...	Create Repository
warra29 / <b>front-cloud</b> Updated 10 hours ago	☆ 0	↓ 143 PUBLIC
warra29 / <b>front-end</b> Updated 2 days ago	☆ 0	↓ 91 PUBLIC
warra29 / <b>sellervservice</b> Updated 2 days ago	☆ 0	↓ 195 PUBLIC
warra29 / <b>orderservice</b> Updated 2 days ago	☆ 0	↓ 196 PUBLIC
warra29 / <b>produitservice</b> Updated 2 days ago	☆ 0	↓ 201 PUBLIC
warra29 / <b>servergateway</b> Updated 2 days ago	☆ 0	↓ 208 PUBLIC
warra29 / <b>eurekaserver</b> Updated 4 days ago	☆ 0	↓ 220 PUBLIC

Figure 22 images docker

## Kubernetes :

Le déploiement sur kubernetes a été fait sur deux environnement différent, VM et CLOUD Google, il y'a une différence minime entre les deux principalement lié à l'accès (utilisation d'un proxy, ingress controller gce sur cloud et Traeffik sur vm), de se fait, sur ce rapport je ne présenterai que le déploiement google cloud, cependant les deux fichier Yaml sont disponibles sur Github.

Cependant une petite erreur subsiste sur google cloud, le ingress controller « gce » ne permet pas de rediriger les routes de mon front, ainsi l'application déployer sur google cloud fonctionne, cependant elle n'est accessible que depuis le lien root, il s'avère que la gestion des routes sur google cloud doit se faire en utilisant en gestionnaire de Traffic http.

Les routes fonctionne cependant une fois qu'on accèdes au root de l'application, de plus **je n'ai pas cette erreur** lors du déploiement sur la VM fournit.

## Architecture :

L'image ci-dessous (et la définition des icones plus bas) représente l'architecture Kubernetes Déployé sur google cloud, une architecture plus ou moins similaire est utilisé pour le déploiement sur machine virtuel à la différence que la communication entre le service Front-end et le gateway qui est géré différemment (via des liens ingress) pour le déploiement en sur VM.

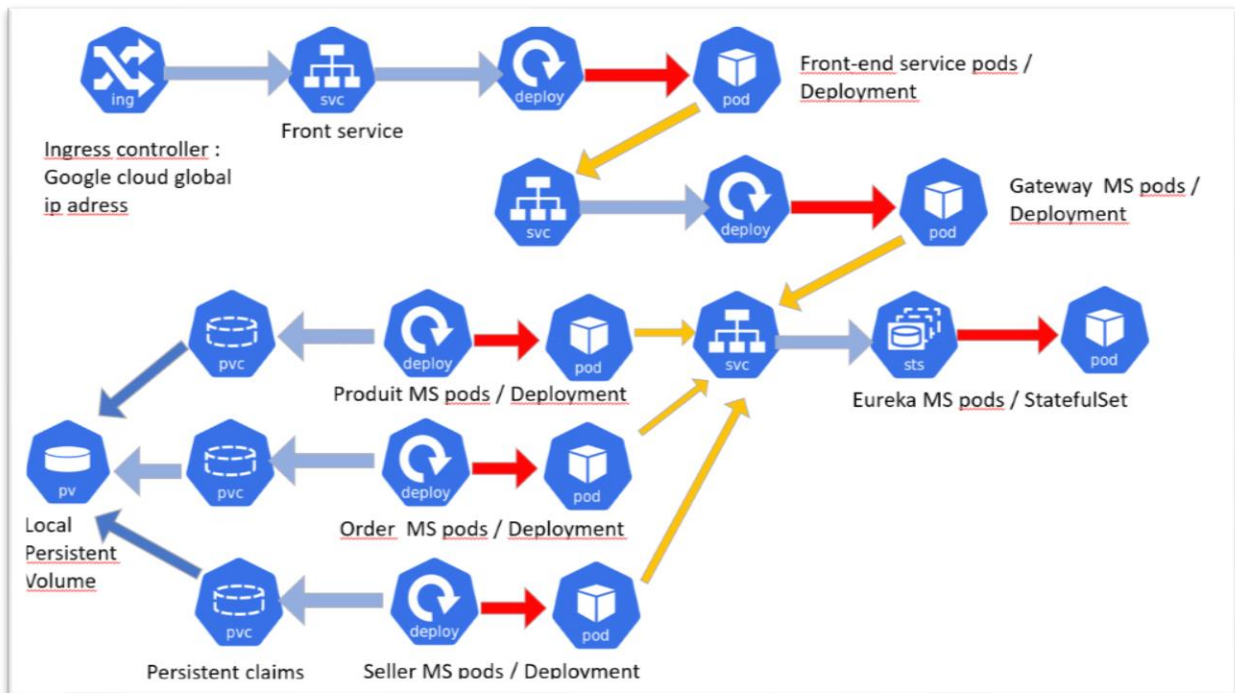


Figure 23 architecture kubernetes



**Ingress:** Ingress is a collection of rules that allow inbound connections to reach the endpoints defined by a backend. An Ingress can be configured to give services externally-reachable urls, load balance traffic, terminate SSL, offer name based virtual hosting etc.



**Service:** Service is a named abstraction of software service (for example, mysql) consisting of local port (for example 3306) that the proxy listens on, and the selector that determines which pods will answer requests sent through the proxy.



**PersistentVolume:** is a storage resource provisioned by an administrator.



**PersistentVolumeClaim:** PersistentVolumeClaim is a user's request for and claim to a persistent volume.



**Pod:** Pod is a collection of containers that can run on a host. This resource is created by clients and scheduled onto hosts.



**StatefulSet:** StatefulSet represents a set of pods with consistent identities. Identities are defined as: network, storage.



**Reference:** Use to represent a reference between components, reference can be through various selector (label, name ...)



**Creation:** Use to represent a generation, resource generate other resource



**Communication:** is used to represent a communication / use of a resource by another

Cette architecture a été réfléchi afin de permettre une bonne scalabilité en respectant le principe des architectures micro services, ainsi, chaque micro service (Produit, Order, Vendeur, Zuul, Front-end) est créer en utilisant un « Deployment » propre à lui, de se fait il est facile d'ajouter des répliques, dans mon cas, J'utilise 2 réplique Gateway afin de répartir efficacement la charge (loadbalancing).

L'ensemble de ces deployments crée des pods de nos différents services, à leurs création, ces derniers s'enregistrent au près d'un registre Eureka.

Les registres Eureka sont quant à eux déployés en utilisant un « StatefulSet » afin de préserver un nom unique lors de leur création (Eureka-0), il est ainsi possible pour les différents services de s'enregistrer auprès du registre en utilisant le nom de ces pods, il est aussi possible d'ajouter plus de répliques de notre serveur Eureka.

Les micro-Services Produit Order Client utilisent aussi un **persistent volume** local via une **persistent claim** afin de monter le dossier de leurs bases de données sur un disque local et ainsi sauvegarder les données de manière **persistante**.

Finalement, le frontend est référencé en utilisant un service de type NodePort, ce service est ensuite référencé par Ingress en spécifiant que tous les chemins doivent être redirigés vers ce service.

Les micro-services front end, qui font aussi office de serveur proxy, peuvent rediriger les requêtes vers le gateway via le service gateway.

Ingress lui-même utilise comme host une IP statique globale générée par Google Cloud que je nomme « myglobalingress-ip » que j'utilise lors de la configuration du fichier yaml :

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: proxed-ingress
  annotations:
    kubernetes.io/ingress.class: "gce"
    kubernetes.io/ingress.global-static-ip-name: myglobalingress-ip
spec:
```

Et que je génère en utilisant la commande :

```
gcloud compute addresses create myglobalingress-ip --global
```

je peux ensuite récupérer cette adresse grâce à un `get ingress` par exemple :

```
wacimbelahcel@cloudshell:~$ kubectl get ingress
```

NAME	HOSTS	ADDRESS	PORTS	AGE
front-end-ingress	backend.localhost,frontend.localhost,registry.localhost		80	26m
proxed-ingress	*	34.107.252.30	80	26m

```
wacimbelahcel@cloudshell:~$
```

ci-dessous, l'ensemble des services / pods / pv-claim générés lors du déploiement sur Google Cloud (résultat similaire sur VM), certains services ont été créés seulement à but éducatif et permettent de tester la disponibilité efficacement.

SERVICES KUBERNETES

ENTRÉE

Les services sont des ensembles de pods avec un point de terminaison de réseau pouvant servir pour la détection et l'équilibrage de charge. Les entrées sont des ensembles de règles qui permettent de router le trafic HTTP(S) externe vers les services.

Est un objet système : Faux

Filtrer les secrets et les mappages de configuration

<input type="checkbox"/>	Nom ↑	État	Type	Points de terminaison	Pods	Espace de noms	Cluster
<input type="checkbox"/>	eureka	OK	Adresse IP de cluster	None	1/1	default	webdist
<input type="checkbox"/>	eureka-lb	OK	Équilibreur de charge	34.70.250.74:8761	1/1	default	webdist
<input type="checkbox"/>	front-end-ingress	OK	Ingress	backend.localhost/	0/0	default	webdist
<input type="checkbox"/>	frontend	OK	Port du nœud	10.0.9.138:80 TCP	1/1	default	webdist
<input type="checkbox"/>	frontend-lb	OK	Équilibreur de charge	35.226.110.72:80	1/1	default	webdist
<input type="checkbox"/>	orders-service	OK	Port du nœud	10.0.8.111:80 TCP	1/1	default	webdist
<input type="checkbox"/>	produits-service	OK	Port du nœud	10.0.6.43:80 TCP	1/1	default	webdist
<input type="checkbox"/>	proxed-ingress	OK	Ingress	34.107.252.30/*	0/0	default	webdist
<input type="checkbox"/>	sellers-service	OK	Port du nœud	10.0.7.139:80 TCP	1/1	default	webdist
<input type="checkbox"/>	zuul-server	OK	Adresse IP de cluster	None	2/2	default	webdist

Figure 24 services

Les demandes de volume persistant sont des requêtes de stockage avec une taille et un mode d'accès spécifiques. <a href="#">En savoir plus</a>						
Filtrer les demandes de volume persistant						
<input type="checkbox"/>	Nom ↑	Phase	Volume	Classe de stockage	Espace de noms	Cluster
<input type="checkbox"/>	website-pv-claim	Bound	website-pv-volume	manual	default	webdist

Figure 25 PV Claim

Les charges de travail sont des unités de calcul déployables qui peuvent être créées et gérées dans un cluster.									
<div>Est un objet système : Faux</div> <div>Filtrer les charges de travail</div>									
<input type="checkbox"/>	Nom ↑	État	Type	Pods	Espace de noms	Cluster	Pods en cours d'exécution	Pods souhaités	Est un objet système
<input type="checkbox"/>	eureka	OK	Stateful Set	1/1	default	webdist	1	1	false
<input type="checkbox"/>	frontend	OK	Deployment	1/1	default	webdist	1	1	false
<input type="checkbox"/>	orders-service	OK	Deployment	1/1	default	webdist	1	1	false
<input type="checkbox"/>	produits-service	OK	Deployment	1/1	default	webdist	1	1	false
<input type="checkbox"/>	sellers-service	OK	Deployment	1/1	default	webdist	1	1	false
<input type="checkbox"/>	zuul-server	OK	Deployment	2/2	default	webdist	2	2	false

Figure 26 Pods

Pour l'exemple, deux serveurs Zuul sont déployé ici, ces deux serveurs se répartissent la charge en RoundRobin.

### Petit Obstacle :

Tel que cité précédemment, la configuration du Controller ingress « gce », qui permet de posséder une adresse IP global, ne me permet malheureusement pas de rediriger toutes mes routes front end vers angular tel que je le souhaite, ainsi le site web est accessible sur le lien root « / », à partir de la on peut naviguer sur l'ensemble de l'application, cependant tout autre chemin n'est pas accessible depuis l'url du navigateur car elle ne se fait pas router correctement par Ingress.

Voici un [lien](#) vers une issue liée à ce sujet, une solution a été proposé mais qui prendrait plus de temps à mettre en place.

Cependant cette erreur n'apparait pas lors d'un déploiement sur VM avec un contrôleur ingress Traeffik.

## **5. Conclusion**

En conclusion, je tiens à préciser que malgré la situation actuelle, et bien qu'ayant déjà fait de web lors de mes précédents projets, ce projet m'en a appris énormément sur les technologies micro services et les déploiement en production, je me sens maintenant beaucoup plus apte à appliquer cela sur lors de déploiement d'envergure et je compte bien en apprendre plus sur le sujet.