Kubernetes

(présentation)

Table des matières

I - Kubernetes (vue d'ensemble)

1. Kubernetes (présentation)

Kubernetes (*k8s*)

Kubernetes signifie "timonier" ou "pilote" en langue grecque .

Kubernetes (k8s) est une **plate-forme logicielle** permettant d'automatiser le déploiement, la montée en charge et la mise en œuvre de **conteneurs** d'application sur des **clusters de serveurs** .

Kubernetes est très souvent utilisé avec des conteneurs "**docker**" mais il a été cependant conçu pour fonctionner avec d'autres sortes de conteneurs .

Kubernetes s'inscrit à fond dans la tendance architecturale "***DevOps , Cloud, micro-services***"

Historique , utilisation et évolution de Kubernetes

- Annoncée par **Google** en 2014 , la première version de kubernetes sort en **2015 .** Kubernetes est le résultat de la réécriture en langage "**GO**" d'un ancien système "borg" permettant de gérer certaines infrastructures de "Google"

- D'origine "Google" , kubernetes a été offert à la "***Cloud Native Computing Foundation***" qui le gère actuellement en tant que projet "***open source***".

- ***Kubernetes sert actuellement à arrêter et redémarrer des millions de containers chaque semaine***.

- Des services comme ***Gmail*** ou ***Map*** tournent dans des conteneurs gérés par Kubernetes

- En 2018, **Kubernetes** représentait **51% du marché des orchestrateurs de conteneurs** contre **11% pour Docker Swarm** et **4% pour Mesos** .

Besoins à l'origine de Kubernetes

Pour gérer un grand nombre de conteneurs "docker" en production de nombreuses questions se posent :

- Comment gérer les dysfonctionnements ?
- Comment gérer les déploiements et leurs emplacements ?
- Comment gérer l'élasticité (le "scaling")?
- Comment gérer les mises à jours ?
- Comment gérer la communication entre les conteneurs ?
- Comment gérer le stockage nécessaire à la persistance des données ?
- Comment gérer la sécurité (clefs secrètes) et la configuration ?
- etc.

Tout gérer de manière manuelle et sans surcouche au système de conteneurs (docker ou autre) n'est pas viable ni maintenable.
--> une automatisation est nécessaire .s

Terminologie de Kubernetes

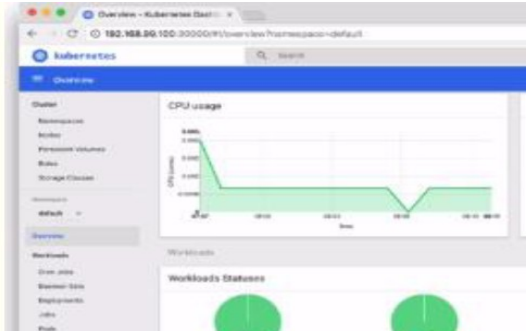
- **noeud** = **machine hôte** (virtuelle ou pas) **, cluster=ensemble de noeuds** (rendant les mêmes services techniques) supervisés comme un tout cohérent.
- **pod** = **ensemble de conteneurs co-localisés** sur un même noeud et ***partageant des accès à certaines ressources*** (réseau , stockage)
- **replicas** = **instances** d'un **pod** .
- **service** = paquet d'instances de "pods" rendant les mêmes services logiciels/applicatifs (souvent associé à une des couches logicielles : IHM/web ou API-REST ou database , ...) . pour "load-balancing" .
- **deployment** = configuration du déploiement d'une application (avec différents états)

Vue externe (contrôle) d'un cluster kubernetes

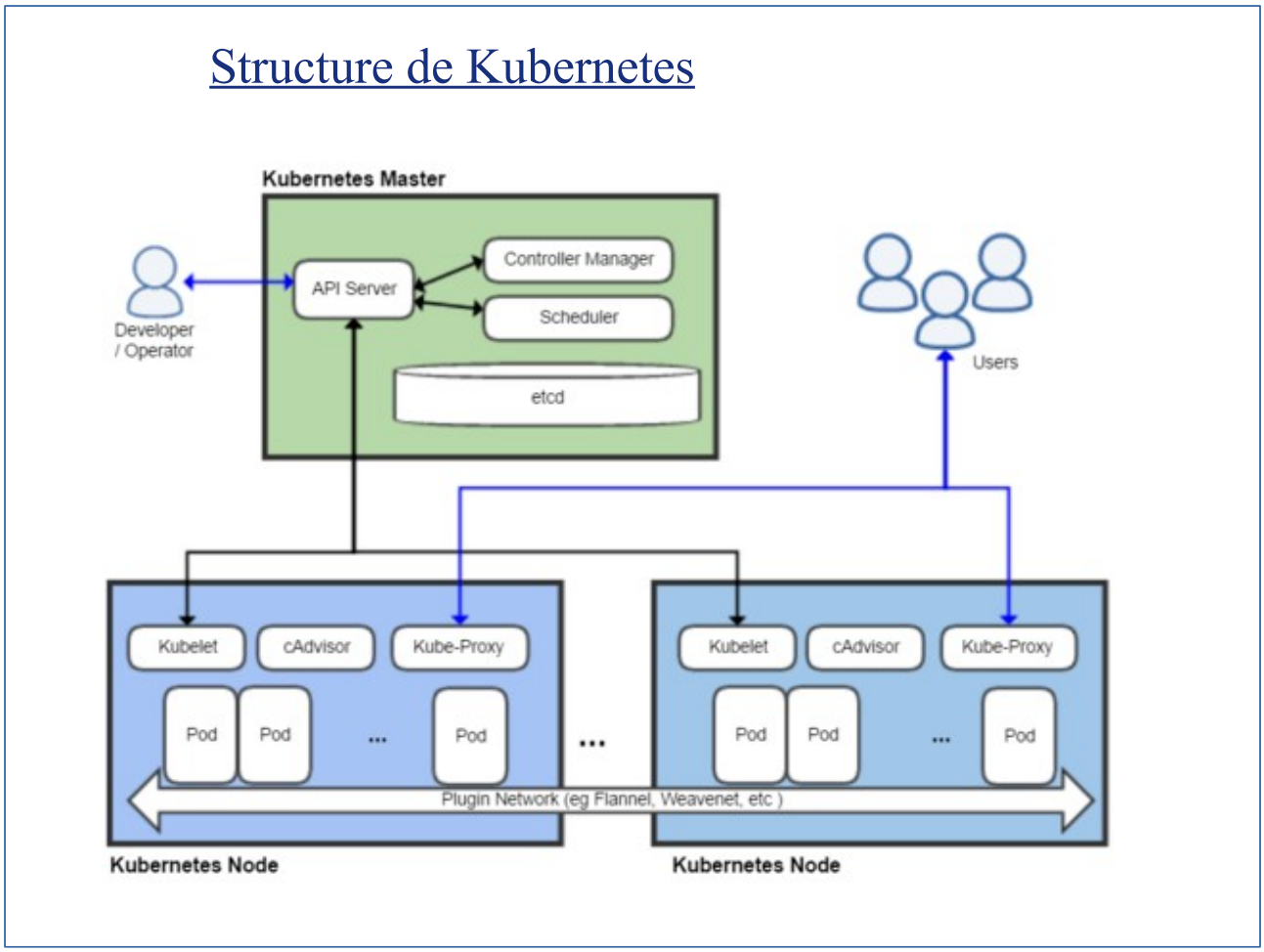
Un cluster "kubernetes" est géré/contrôlés via :

- une application "**kubectl**"
  (en ligne de commande) :
  exemples :
  - **kubectl** create -f specif.yml
  - **kubectl** get pods

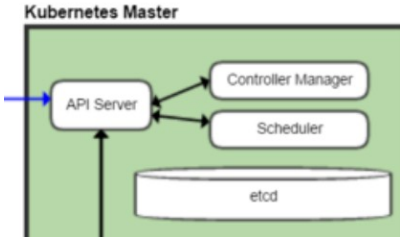
- une console web
  (facultative) :
"**dashboard**"



2. Structure de kubernetes

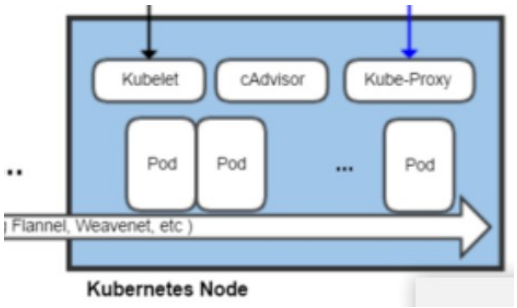


Composants internes de "kubernetes" sur une machine "maître"

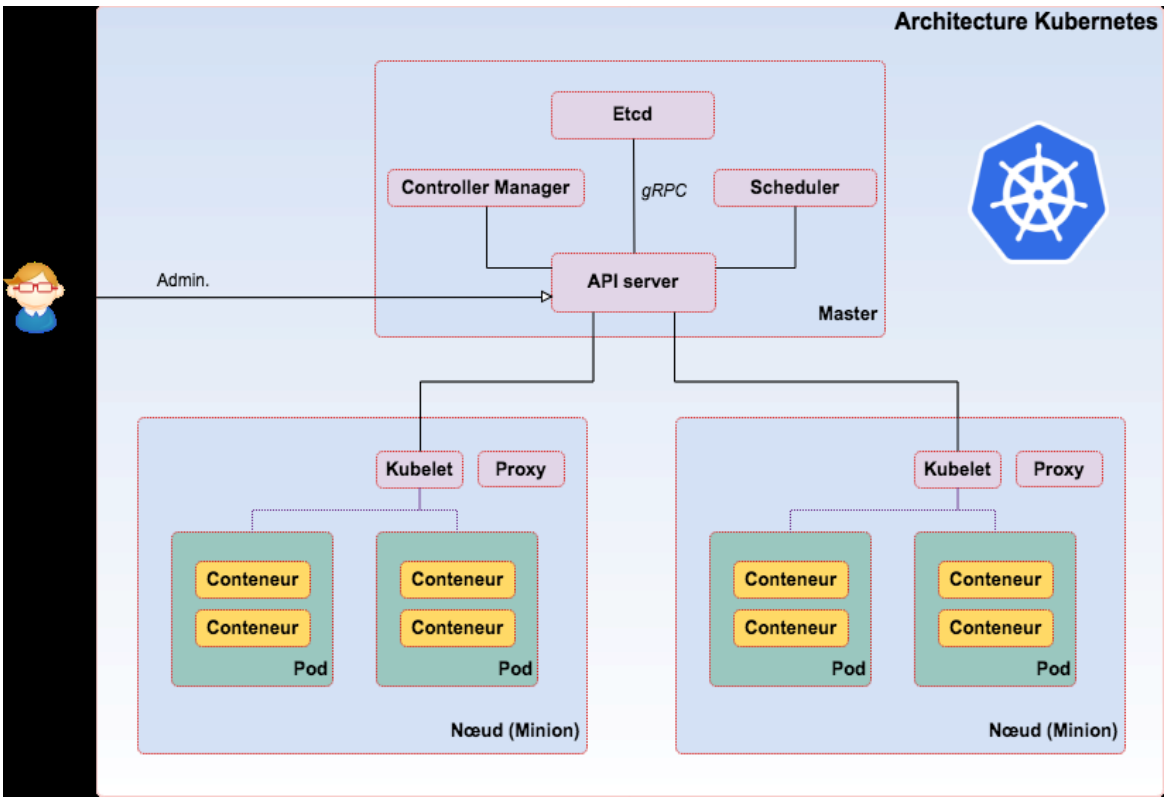


- **etcd** = **base de données** distribuée et légère (clefs/valeurs) utilisée pour mémoriser de façon persistante la **configuration du cluster** .
- **API-Server** = Server d'api REST/JSON servant à contrôler le cluster (principal client = kubectl) .
- **Controller Manager** = Gestionnaire de "contrôleurs/superviseurs" (de replicas, de pods, de ...)
- **Scheduler** (ordonnanceur) : contrôle l'aspect "auto-adaptatif" d'un cluster kubernetes (ordres de "lancement / arrêt" selon états des noeuds : charge limite saturation, ...), S'occupe d'assigner un noeud disponible à un pod.
- **Service DNS** : intégré à Kubernetes, les conteneurs utilisent ce service dès leur création pour la résolution de nom à l'intérieur du cluster.

Composants internes de "kubernetes" sur une machine "worker"



- **Kubelet** (agent principal) : Kubelet est responsable de l'état d'exécution de chaque nœud (c'est-à-dire, d'assurer que tous les conteneurs sur un nœud sont en bonne santé). Il prend en charge le démarrage, l'arrêt, et la maintenance des conteneurs d'applications (organisés en pods)
- **cAdvisor** : agent qui surveille et récupère les données de consommation des ressources et des performances comme le processeur, la mémoire, ainsi que l'utilisation disque et réseau des conteneurs de chaque node.
- Le **kube-proxy** est l'implémentation d'un proxy réseau et d'un répartiteur de charge, il gère le service d'abstraction ainsi que d'autres opérations réseaux



Overlay network (SDN) (ex : Flannel) in kubernetes

- **SDN** : **S**oftware **D**efined **N**etwork.
- **overlay network = réseau superposé** (réseau virtuel sur réseau réel)
- ***Flannel*** est un des "SDN/overlay network" utilisable avec Kubernetes.

Utilité :

Les "pods" sont considérés relativement éphémères (il peuvent être arrêtés ou déplacés sur un autre noeud) .

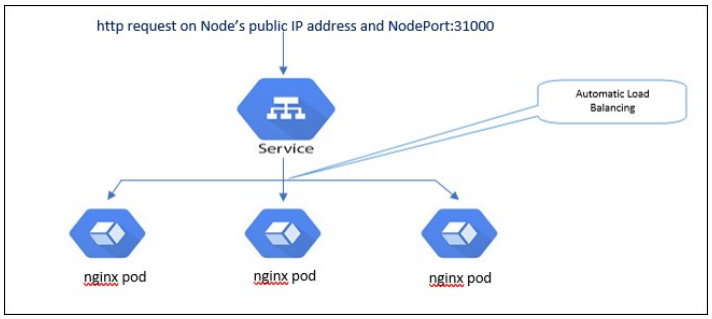
De manière à effectuer des communications inter-machines au sein d'un cluster en potentiellement permanente restructuration, on ne peut pas s'appuyer sur des adresses IP fixes mais il faut un système "overlay network" avec adresses IP virtuelles (liées au cluster) .

3. Composants à configurer et administrer

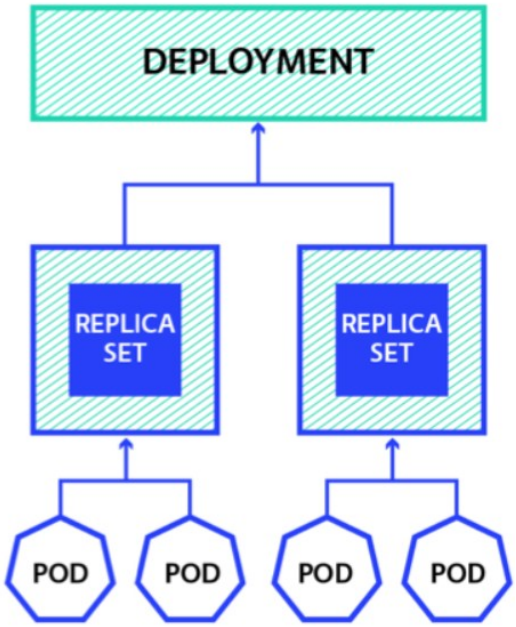
Service "kubernetes"

- abstraction correspondant à un ensemble logique de pods (rendant les mêmes services)
- assure un découplage entre :
  - le micro-service appelant (qui ne voit que l'adresse IP virtuelle fixe ou le nom logique du service)
  - les "réplicas"/instances des pods
- Prend en charge le "load-balancing" vers les différents "pods/noeuds" (en tenant compte de l'aspect "fail-over")

exemple :



Deployment kubernetes



xyz-deployment.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

kubectl apply -f https://k8s.io/examples/controllers/***nginx-deployment.yaml***

4. Minikube

minikube : cluster ultra-simplifié (sur un seul noeud)

- **Minikube** exécute un cluster Kubernetes à un seul nœud à l'intérieur d'une machine virtuelle sur votre ordinateur *(**linux** ou **windows** ou ...)*.
- *ça* convient pour une première prise en main Kubernetes ou effectuer quelques tests ou pour se familiariser avec les commandes de "kubectl" ou la console web "dashboard". Son installation nécessite au préalable un hyperviseur, kubectl et docker .
- Il démarre avec la commande: **minikube start**

ANNEXES

II - Annexe – DevOps

1. DevOps

1.1. Présentation de devops (origine, concepts, ...)

DevOps

Devops est la concaténation des trois premières lettres du mot anglais development (*développement*) et de l'abréviation usuelle ops du mot anglais operations (*exploitation*), deux fonctions de la gestion des systèmes informatiques qui ont souvent des objectifs contradictoires *(ex : nouvelles fonctionnalités apportant de l'instabilité / fiabilité exigée en exploitation)*

Le **devops** est un mouvement en ingénierie informatique et une pratique technique visant à l'**unification** du développement logiciel (***dev***) et de l'administration des infrastructures informatiques (***ops***), notamment l'administration système. *(source wikipedia)*

Origine et principes de DevOps

Apparu autour de 2007 en Belgique avec Patrick Debois, le mouvement Devops se caractérise principalement par la promotion de l'automation et du suivi (monitoring) de toutes les étapes de la création d'un logiciel, depuis le développement, l'intégration, les tests, la livraison jusqu'au déploiement, l'exploitation et la maintenance des infrastructures.

Conférences "DevOpsDays" (la première en 2009 à Gand / Belgique)

Autre terminologie: "***Agile Infrastructure***"

Les principes Devops soutiennent des cycles de développement plus courts, une augmentation de la fréquence des déploiements et des livraisons continues, pour une meilleure atteinte des objectifs économiques de l'entreprise. *(source wikipedia)*

Avant "DevOps"

Univers "Dev" et "Ops" traditionnellement séparés dans les années 1995-2010 pour les raisons suivantes :
- beaucoup de complexités à gérer → besoin de se spécialiser .
- solutions à bases de serveurs (ex : ServApp JEE) à administrer et dans lesquels on déploie des applications (à développer).
- peu d'automatisation , contextes différents (O.S. , ...)

Objectifs "dev"	Objectifs "ops"
. apporter les changements nécessaires au moindre coût et le plus vite possible souvent au détriment de la qualité lorsque des retards viennent mettre les délais du planning en péril	. garantir la stabilité du système. . se concentrer sur contrainte qualité, . besoin de temps et de moyens . contrôler sévèrement les changements apportés au système

Vers "DevOps"

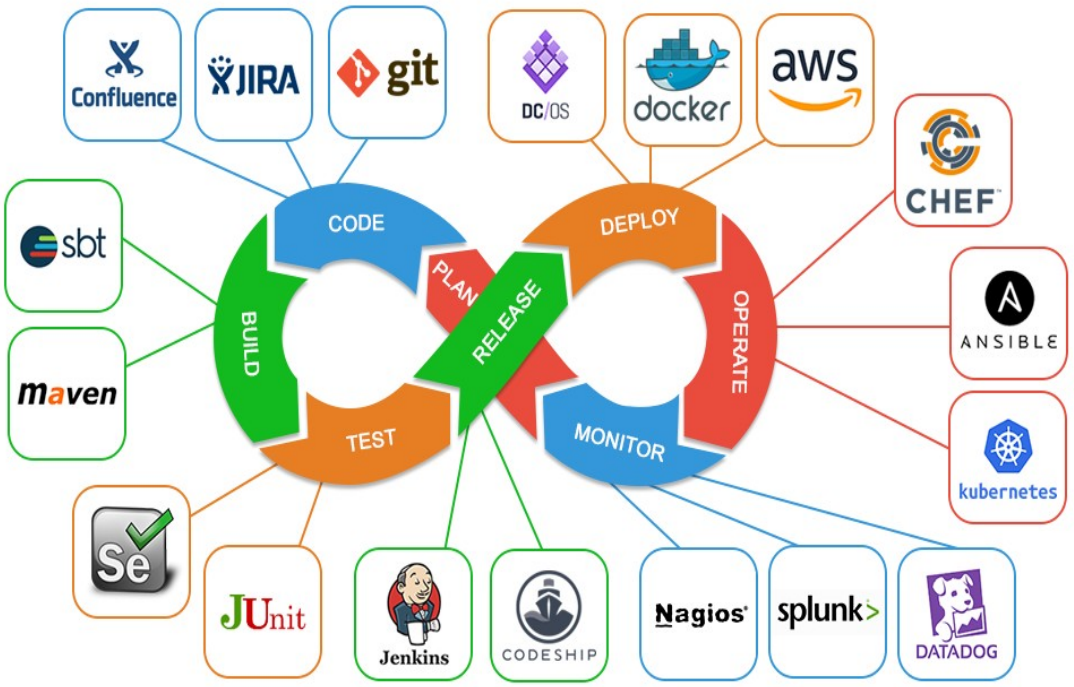
- changement de paradigme ("séparation" → "unification")
- les équipes "dev" et "ops" ne doivent pas se rejetter la faute en cas de problèmes mais doivent collaborer et s'entraîder de façon à ne forme qu'une seule "méga-équipe" .

- automatisations (intégration continue , déploiements, …)
- cycles courts (dev, tests, intégration , mise en prod, …)

- bonnes métriques / supervisions
- applications des méthodes agiles (tests , communications, …)
- alignement avec les technologies "cloud" et "web"

1.2. cycle de vie devops

Cycle de vie "DevOps"



Première phases du cycle "devops"

1. Planification (et modélisation , suivi)

2. Codage/implémentation (avec tests unitaires)

Phases suivantes du cycle "devops"

3. Build (de "dev" ou "release/prod")
(assemblage , pré-released)

maven

…

éventuels profils de 'dev' / 'prod'
avec "bases" et ...
plus ou moins simplifié(e)s

4. Tests d'intégration (intégration continue)

Exécution automatique des tests dans le but d'avoir un feedback concernant la qualité du code produit en cours de déploiement

Phases suivantes du cycle "devops"

5. Release (de ou vers "prod")

maven profile de 'prod'
avec vraies "bases" / "serveurs" / ...

Environnement au sein duquel l'identification
des anomalies doit idéalement pouvoir se faire aisément.

6. Deploy (packaging & déploiement)

Dernieres phases du cycle "devops"

7. Operate (fonctionnement en "production")

fonctionnement en cluster ,
...

8. Monitoring (surveillance & métriques)

surveillance/exploitation ,
remontées/interprétations des mesures , gestion des logs,
Redimensionnement éventuel (élasticité cloud, ...)

1.3. Devops , concrètement ...

Concrètement "DevOps" c'est avant tout :

- De l'agilité au niveau de l'infrastructure (et pas que sur le papier en théorie)
- Ça se traduit par des automatisations efficaces (intégration continue et déploiement de conteneurs "docker") de façon à ce que les développeurs puissent assez facilement construire des "briques" prêtes à être déployées sans trop de d'ajustements/paramétrages .
- Et inversement , certains informaticiens (très techniques) peuvent packager des environnements sophistiqués (avec sécurité , …) correspondant aux contraintes réelles de production sous forme d'images "docker" de façon à ce que le développeur puisse facilement y intégrer et tester de nouvelles fonctionnalités.

III - Annexe – Architecture micro-services

1. Architecture "micro-services" (essentiel)

1.1. Présentation architecture micro-services

Architecture "micro-service" (*MSA*)

L'**architecture "micro-services"** peut être vue comme une *évolution de l'architecture "SOA/orientée services" avec* :

- ***plus de légèreté dans l'infrastructure logicielle*** (moins ou pas de ESB, Orchestrateurs "BPM" et autres intermédiaires sophistiqués).

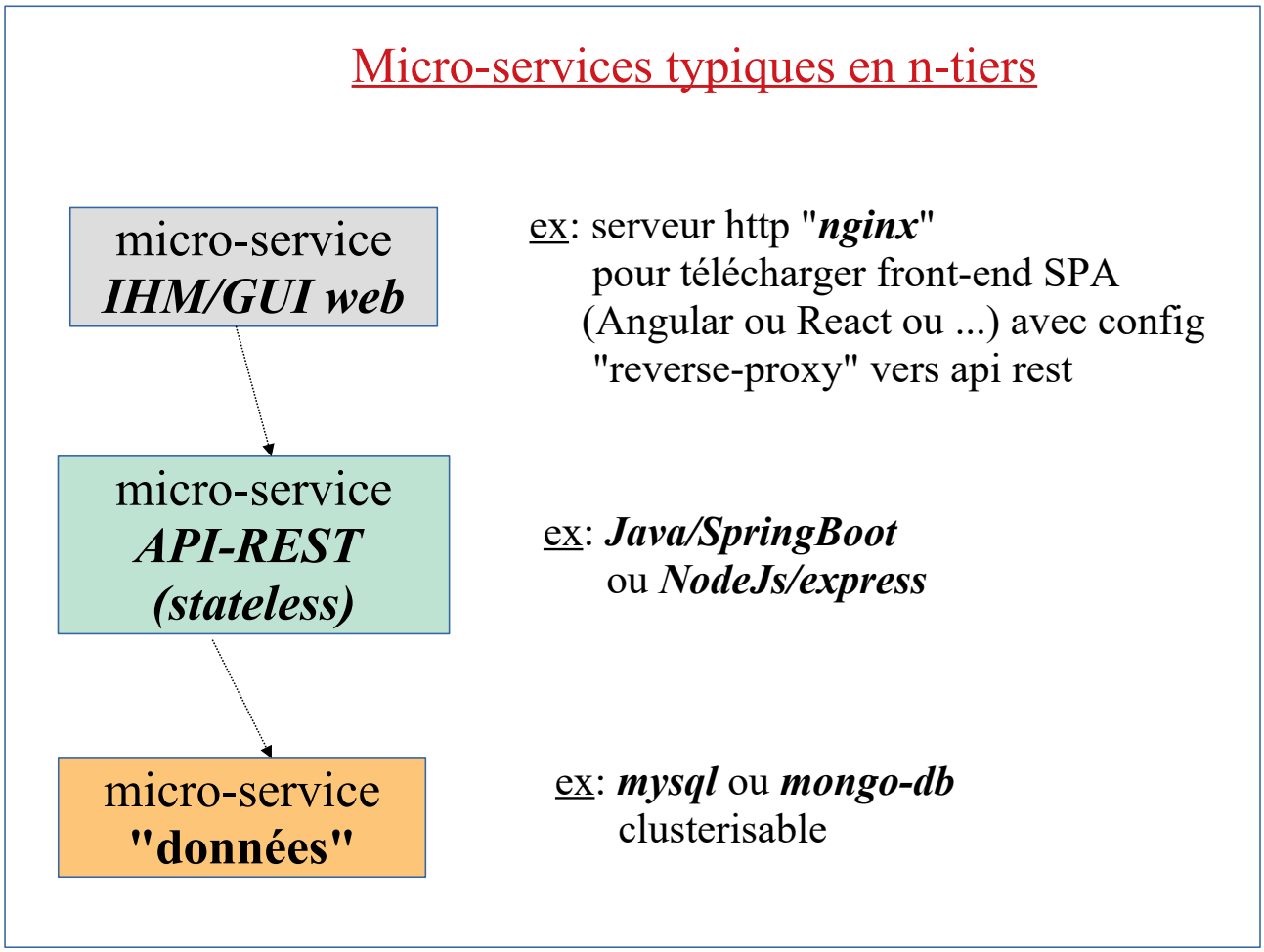
- **alignement sur les technologies "cloud" (Iaas,Paas) et "micro-conteneurs"** (infrastructure "scalable"/élastique (utilisation fréquente de *Docker* , *Kubernetes* , ...))

- ***tolérance vis à vis des services inaccessibles*** (pas de "point de panne simple" , cluster avec prise en compte de l'état des serveurs)

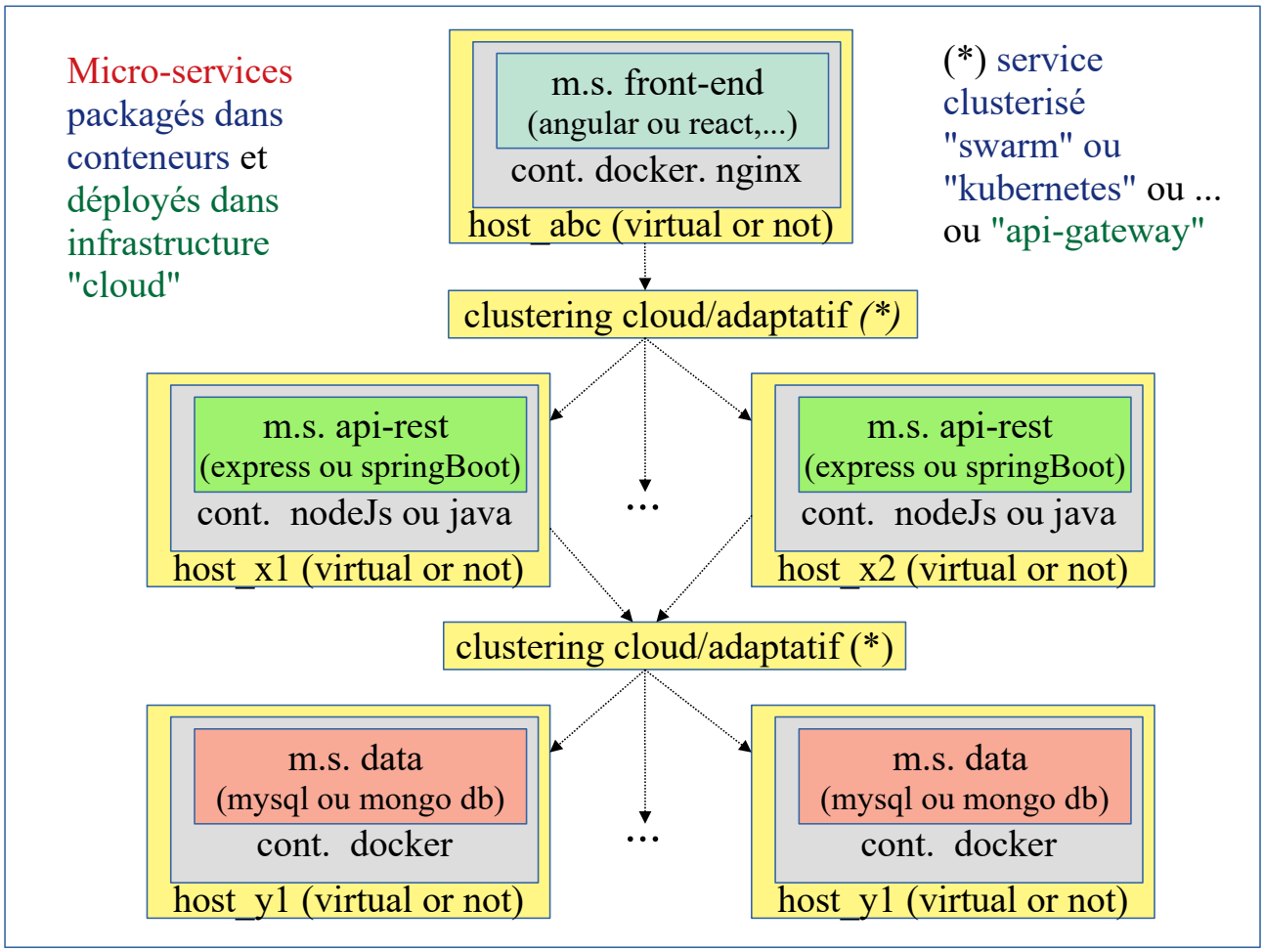
- alignement avec "WOA" (xml/soap --> ***json/REST/HTTP***)

- ***très peu d'inter-dépendances entres les micro-services*** (idéalement "stateless" et ***programmables avec technologies très variables*** : Java/SpringBoot , NodeJs/Express , python, ...)

1.2. illustration (exemples typiques)



1.3. illustration (packaging / fonctionnement "cloud")



1.4. Caractéristiques clefs des micro-services

Caractéristiques structurelles d'un micro-service

- ***forte cohésion interne , périmètre fonctionnel très réduit***
(faire une seule chose et le faire bien/efficacement)

- ***idéale indépendance*** (conçus et développés indépendamment , testés et déployés ***indépendamment*** des autres (dans micro-conteneurs)
--> agilité / souplesse technologique

- ***beaucoup de délégation*** (très grande sollicitation des réseaux informatiques)

- "***brique de code simple***" packagée et déployée dans topologie "compatible cloud" sophistiquée (ex : réseaux superposés virtuels sur réels)

1.5. Souplesse dans le choix des technologies

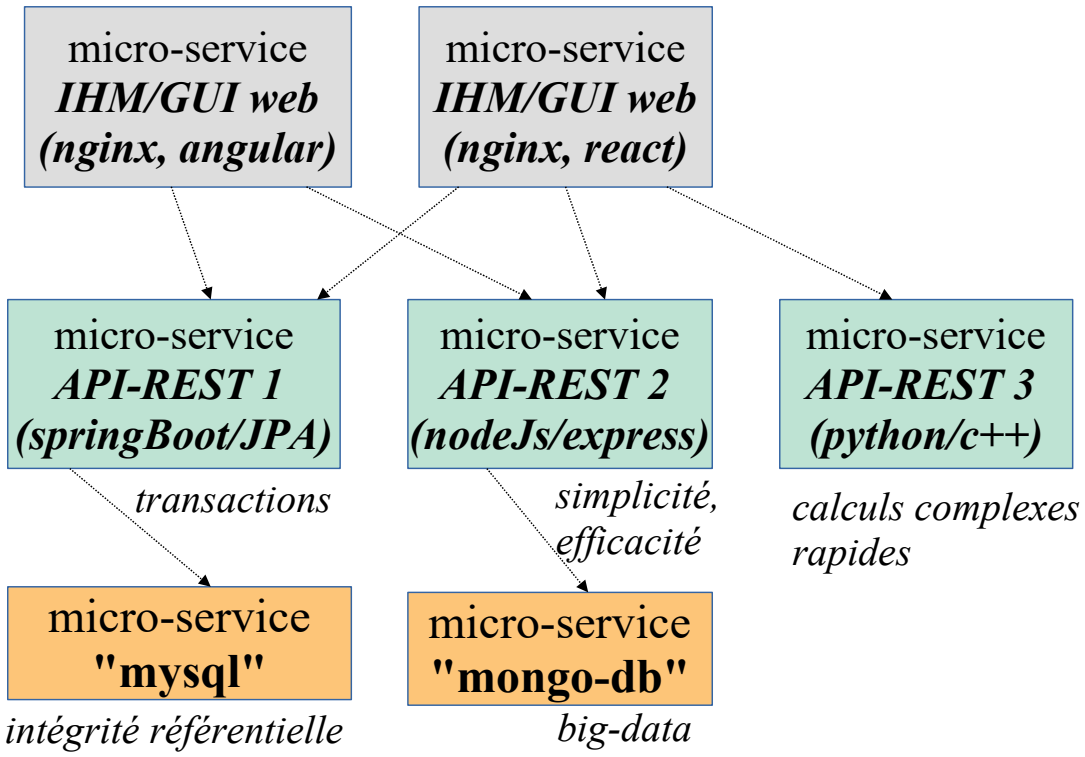
Agilité/souplesse technologique

- *Mis à part quelques petits éléments protocolaires généralement imposés (tcp/ip , ...)* , **un micro-service peut être programmé dans une technologie quelconque** *(java , javascript, python, ...)* car toutes les librairies et configurations spécifiques seront cachées/isolées au sein de micro-conteneurs opaques .

- ***chaque équipe de développement peut donc choisir librement la technologie la plus appropriée pour développer son micro-service***

(ex : *java/jee/spring-boot* pour *transactions courtes*
nodeJs/express pour *simplicité/efficacité*,
base relationnelle (mysql, ...) pour *données référentielles*
mongoDB/... pour données en grand volume (*big data*)
elastic-search pour *gestion historique et recherches*
python et c/c++ pour *service scientifique (calculs , ...)*
...

Variétés technologiques des micro-services



1.6. Impacts de l'architecture micro-services

Quelques impacts de l'architecture micro-service

- "*périmètre réduit*" + "*liberté dans technologie de mise en oeuvre*" font que :
  - **les risques sont plus limités/circoncis** .
    (*moins de problème d'incompatibilité entre technologies*, si un sous projet échoue on change de technologie et on recode mieux le micro-service à problème . ce n'est pas l'ensemble qui est compromis)

  - beaucoup de liberté dans le choix des frameworks , langages et serveurs :
    ---> ***marché des technos très ouvert*** (ça bouillonne , ça part dans tous les sens , orientation "open source", les grands éditeurs "Microsoft, IBM, ..." nous imposent moins leurs "solutions coûteuses").
    ---> ***développeurs devant de plus en plus être polyglottes !!!***
    (*plein de langages/technos à apprendre/maîtriser* ***!!!***).

Autres impacts de l'architecture micro-service

- **Tests d'intégration facilités** :
  - certains micro-services sont dépendants d'autres micro-servives en arrière plan (ex : api-rest nécessitant accès database)

  - une fois packagé sous forme de conteneur "docker" , un micro-service d'arrière plan pourra:
    - être complexe/sophistiqué en interne (ex : sécurité , ...).
    - être vu comme une "boîte opaque" rendant un certain service utile (un minimum documenté)
    - être utilisé au sein de test d'intégration (alias "end-to-end") pour vérifier la bonne communication inter-service dès la phase de "pré-released" / "pré-production" du micro-service appelant .

  - prévoir peut être variantes/profils "avec ou sans cluster" durant les phases de tests (avec complexité progressive).

1.7. Factorisation / réutilisation des micro-services

Factorisation et ré-utilisation des micro-services

Trouver le bon niveau de découpage / décomposition sachant que :

-un service simple/réduit peu plus facilement être réutilisé

-trop de délégations peuvent induire un *très grand trafic réseau* et un *ralentissement* dû aux sérialisations/dé-sérialisations des requêtes / réponses

- certaines données sont plus simples à relier/corréler localement qu'en mode "dispersé" : *il faut idéalement modéliser (avec UML ou autre) pour étudier les aspects "cohésion non décomposable" et "indépendances possibles"* .

- *ne pas hésiter à restructurer d'une version à l'autre* (le "refactoring" n'est jamais un objectif mais c'est un "moyen souvent nécessaire à mettre en oeuvre").

- *orchestration simple = "bonne idée de SOA à reprendre"*

1.8. Evolutivité des micro-services

Evolutivité d'une architecture micro-service

Aspects favorisant l'évolutivité de l'architecture :

- "périmètre réduit" , "faible complexité" et "choix technologiques" favorisent les évolutions unitaires
(ex : meilleurs performances après ré-écriture ,
ajouts de fonctionnalités opaques (meilleurs heuristiques , ...),
ajouts de traitements annexes)
- "déploiements indépendants" (selon infrastructure hôte telle que "kubernetes") favorisent également l'évolutivité de l'architecture

Contraintes à garder à l'esprit :

- dans l'idéal : ne pas trop changer les structures de données échangées d'un micro-services à l'autre pour ***ne pas remettre en cause les "contrats d'invocation"*** . il doit y avoir une idéale *compatibilité ascendante . [NB : JSON un peu plus souple que SOAP/XML et certains ajustement sont plus simples/rapides]*
- migration de données en cas de changement de format (mysql / mongo) ?

1.9. Résilience des micro-services

Résilience (tolérance panne partielle)

Au sein d'une architecture micro-service , l'**inaccessibilité ponctuelle et les pannes partielles sont considérées comme des phénomènes normaux / courants** .

Pour qu'un micro-service puisse continuer à fonctionner (éventuellement en mode "service légèrement dégradé") dans le cas ou un élément dont il dépend ne fonctionne plus bien , il faudra anticiper des "plans B".

--> ***Choisir des technologies "clusterisables"*** (avec "load-balancing" , "rétablissement de connexions" , ...)

--> ***Bien gérer les exceptions*** (des 2 cotés : client et serveur)

--> ***Bien paramétrer l'infrastructure hôte*** (ex : "*Kubernetes*") qui dit coopérer avec la technologie "clusterisable" (ex : mysql, mongo)

1.10. Bonne exploitation des ressources matérielles

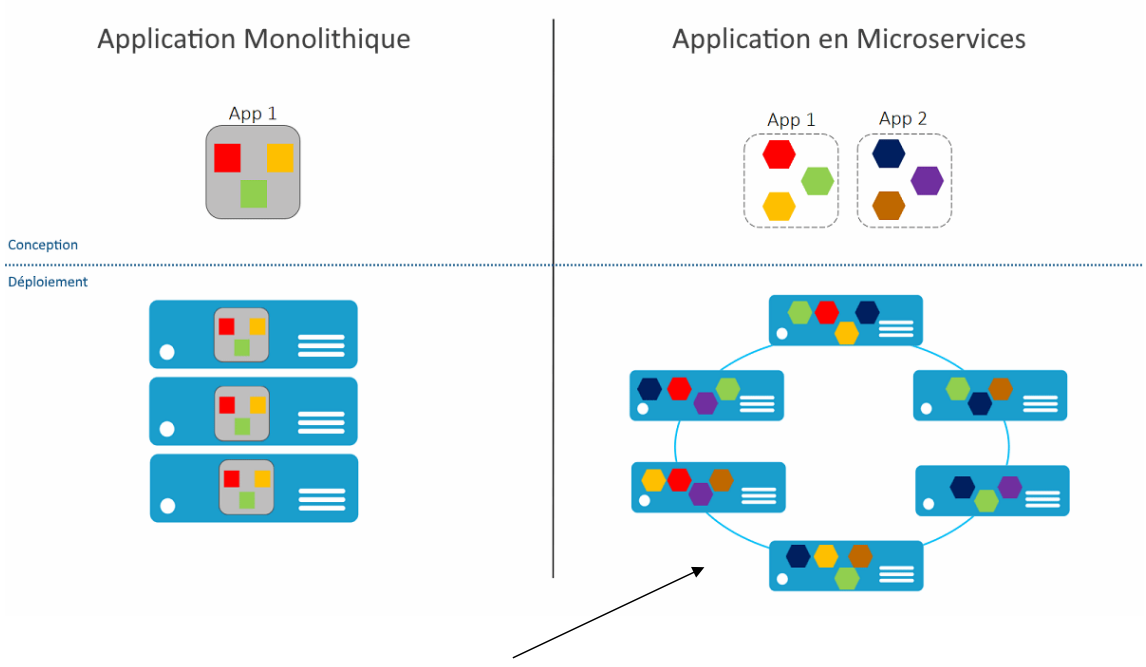
Bonne exploitation des ressources matérielles

L'architecture "micro-service dans micro-conteneur" permet (avec de bons réglages) de généralement mieux exploiter les ressources matérielles (CPU, RAM, ...) qu'une architecture "application mono-bloc" .

- une application mono-bloc (avec plusieurs couches logicielles "GUI" + "Tx" + "Accès données") comporte généralement quelques "goulets d'étranglement" difficiles à gérer/optimiser .
- il faut souvent surdimensionner la puissance de la machine hôte pour être certain d'anticiper certains pics de charge.

- un micro service correspond généralement à un seul "tiers" (soit "GUI" , soit "Api REST" , soit ....) et la consommation en ressource matérielle est plus régulière/homogène . En multipliant le déploiement de conteneurs identiques sur la même machine on peut ainsi assez bien exploiter la puissance d'une machine sans trop la sous-exploiter .

Exploitation optimisée des ressources matérielles



+ nb instances de "micro-conteneurs" selon capacité machine hôte

1.11. Elasticité / scalabilité

Extensibilité/élasticité/scalabilité selon orientation "stateless"

D'emblée prévus pour l'infrastructure "cloud" , les micro-services doivent pouvoir s'adapter à des redimensionnement de voilure (changement de la taille d'un cluster).

Ceci ne pose aucun problème pour les micro-services de type "API stateless" (ex : Web Service de calcul , ...) mais pose de très nombreux problèmes sur l'aspect "cohérence des données réparties/dispersées" :

- théorème "CAP" spécifiant que l'on ne peut pas avoir en même temps les aspects "distribué / tolérance de panne" , "disponibilité / réponses quasi immédiates" et "transactions parfaites ACID" . il faudra privilégier un aspect prioritaire par rapport aux autres .

- intégrité référentielle en mode distribué/dispersé (cohérence entre données fonctionnelles/métiers réparties dans plein de petites bases?)

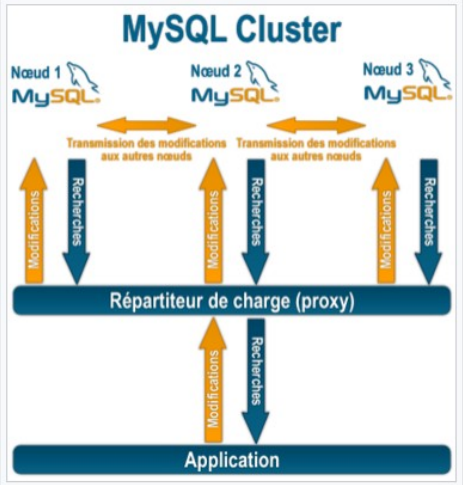
1.12. Cluster de données

Cluster de données avec priorité aux écritures cohérentes

Lorsque l'on fait fonctionner "mysql" en cluster sur différentes machines (avec "load-balancing" et "fail-over") ,

- les *lectures simultanées* peuvent alors être *déclenchées en grand nombre* (relativement rapidement)

- les *écritures sont par contre ralenties* (car *le système relationnel privilégie la cohérence des données écrites/modifiées* via *une réplication immédiate synchronisée vers les autres membres actifs du cluster*)

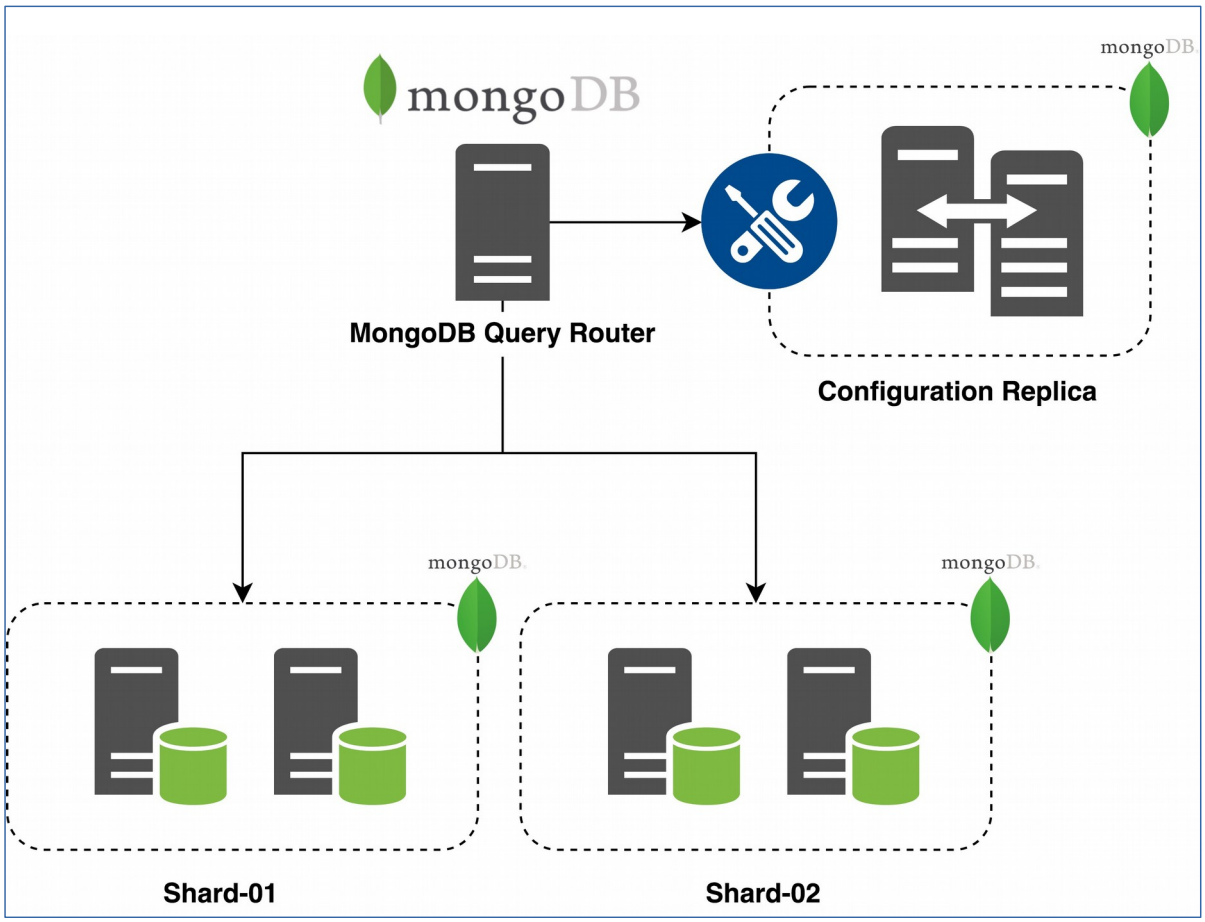


Cluster de données avec priorité aux écritures rapides et lectures immédiates (avec consolidation différée)

Lorsque l'on fait fonctionner certaines technologie "noSQL" (ex : "mongo db") en cluster sur différentes machines (avec "load-balancing" et "fail-over") ,

- *les écritures sont quelquefois* (selon le niveau transactionnel paramétré) *effectuées relativement rapidement et répliquées de façon différée vers les autres membres actifs du cluster*

- *des lectures simultanées effectuées/redirigées vers différents membres du cluster peuvent quelquefois retourner des valeurs différentes* (lorsque la réplication différée n'est pas terminée)
NB : ce manque de précision/actualisation des valeurs lues est quelquefois "pas gênante" dans certains contexte (ex : données statistiques , big-data , ...)



Basic NoSQL (key, value, *timeStamp*)

1. Ordre mise à jour

Gestionnaire de base NoSQL (distribuée)

2. mise à jour initiale (ex : 14:03:04)

2. réplication (en léger différé) de la mise à jour (ex : 14:03:05 et 14:03:06)

A

E1,"nom1_adresse1",*12mai_14:03:04*
E2,"nom2_adresse2",12mai_15:13:04
…
E8,"nom8_adresse8",12mai_17:03:04

B

E1,"nom1_old_adr1",12mai_08:33:09
E2,"nom2_adresse2",12mai_15:13:05
…
E8,"nom8_adresse8",12mai_17:03:05

C

Les synchronisations/réplications peuvent s'effectuer dans tous les sens (A-->B,C ou B-->A,C , …) . L'information technique/cachée "timeStamp" (ex : *12mai_14:03:04*) permet de connaître quelle est la version la plus à jour (la plus récente).

1.13. Eventuel mode message pour les micro-services

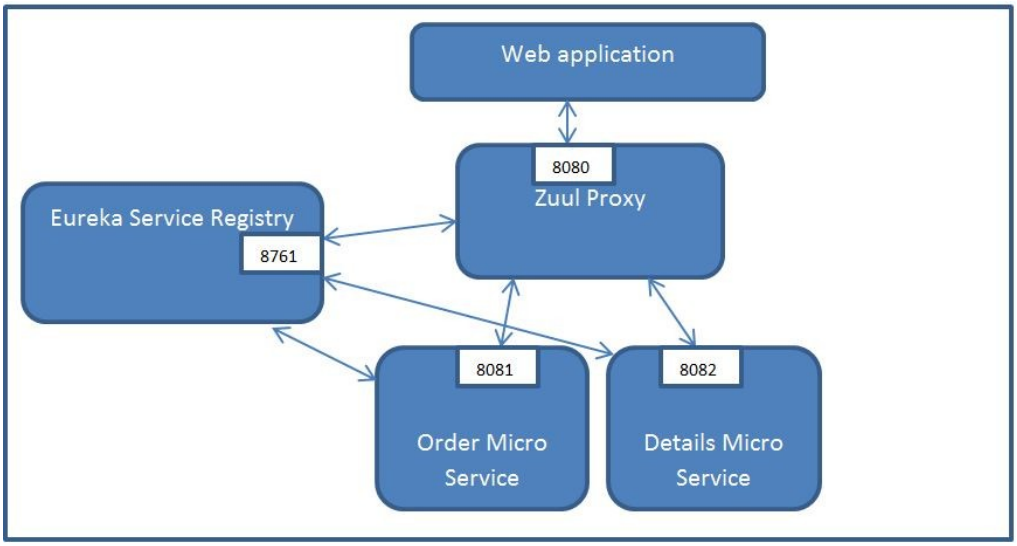
Micro service en mode "message asynchrone / événement"

Bien qu'une api "REST/JSON" soit le type de micro-service le plus répandu , il existe certaines technologies orientées "transports asynchrones de messages" qui sont compatibles avec l'architecture micro-services :

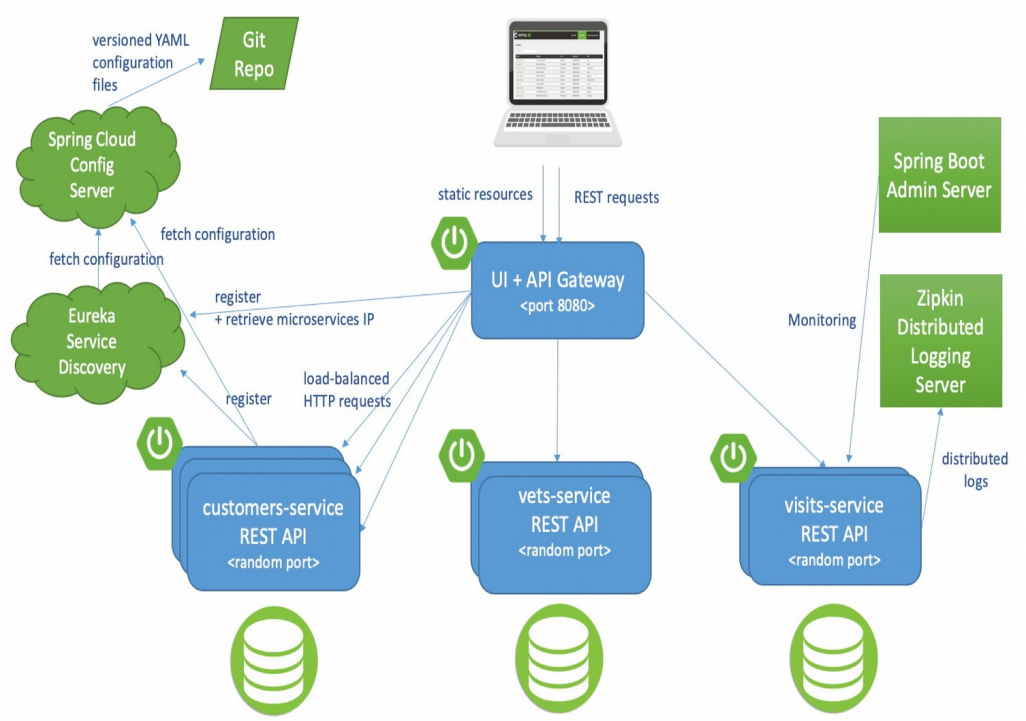
Exemple : Spring-cloud
avec services "Netflix OSS (Eureka, zuul)"
+ RabbitMQ

1.14. Diversité des solutions "cloud/micro-services"

Diversité des solutions "cloud" : Netflix-OSS

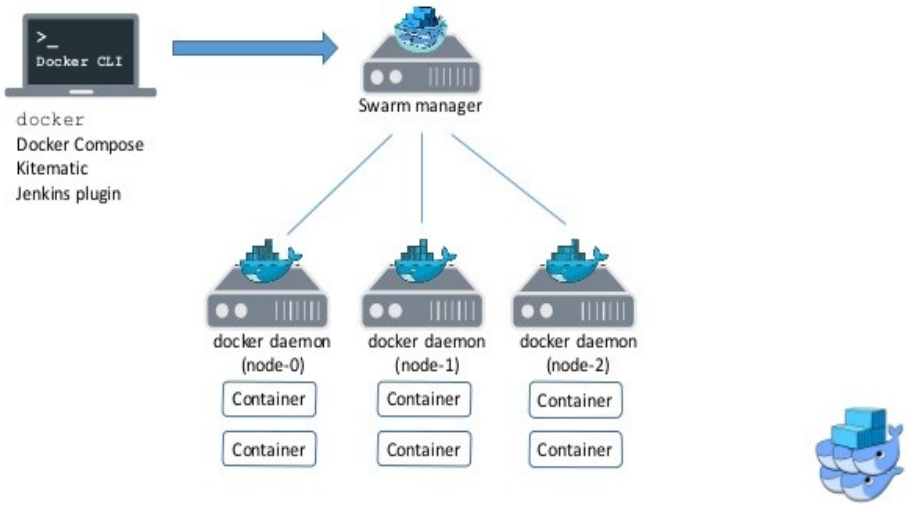


Diversité des solutions "cloud" : Spring-cloud

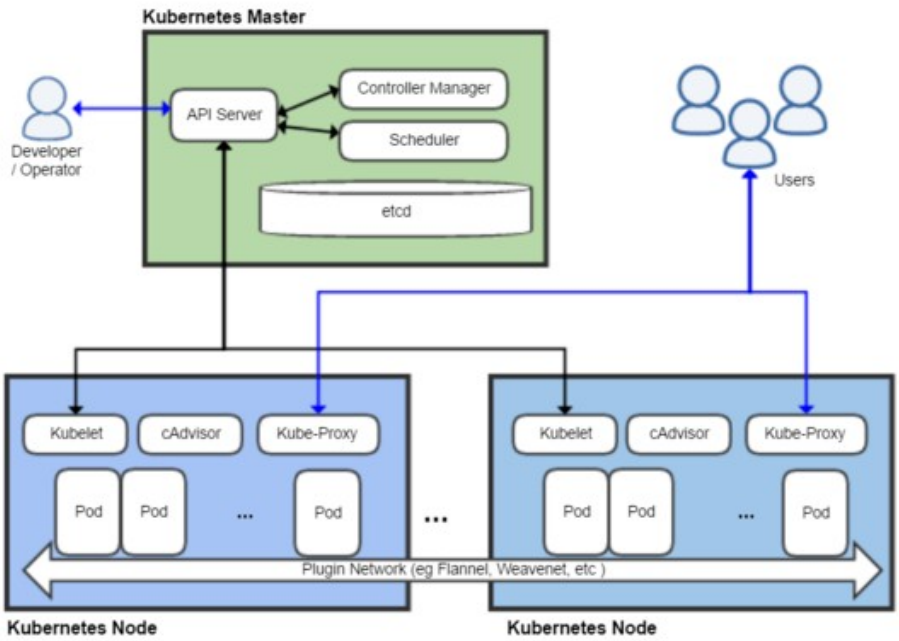


L'extension facultative "spring-cloud" s'appuie sur "netflix-oss"

Diversité des solutions "cloud" : "docker swarm"



Diversité des solutions "cloud" : Kubernetes



NB : Un "pod kubernetes" est un ensemble de "conteneurs co-localisés" (docker ou autres)

IV - Annexe – Bibliographie, Liens WEB + TP

1. Bibliographie et liens vers sites "internet"

2. TP