

On souhaite maintenant poser l'application [TODO](#) sur un cluster Kubernetes.

Lancez d'abord un cluster Kubernetes comme décrit ci-dessous, cette opération prenant beaucoup de temps. Ensuite, familiarisez-vous avec les [notions principales de Kubernetes](#), notamment les notions de Pod, Deployment, Service, Volume, PersistentVolumeClaim, Ingress, Secret.

Exemple illustratif

Une configuration à appliquer sur un cluster Kubernetes est donnée par un fichier YAML. Il est possible de donner plusieurs descriptions à la suite dans le même fichier en les séparant par de lignes contenant ---. Étudions [un exemple](#) pour illustrer certaines des notions essentielles.

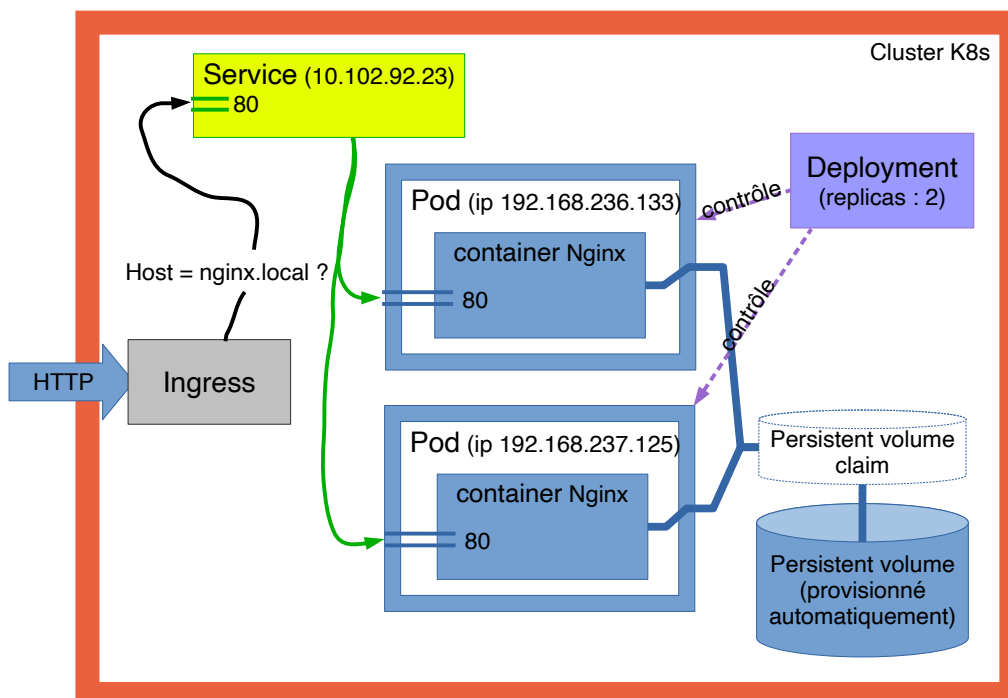
On remarque tout d'abord la création d'un volume ([PersistentVolumeClaim](#)). Un volume est un espace disque qui sera rendu disponible pour un Pod, un Pod étant un ensemble de conteneurs (souvent un seul) partageant une même interface réseau. Ici, le volume est créé automatiquement à l'aide du driver nfs-csi indiquée dans le descripteur.

En second, vient un [déploiement](#). Un déploiement maintient un [ensemble de pods similaires](#), retrouvés (*selector*) par leurs étiquettes (*label*), ici tous les pods possédant une étiquette *run* positionnée à *nginx* seront considérés comme faisant partie du déploiement. Le déploiement fera son possible pour maintenir 2 de ces pods (paramètre *replicas*). En cas de pod manquant, un *template* décrira la forme que devra prendre un nouveau pod. Tout d'abord, on indique ses métadonnées, qui contiennent nécessairement l'étiquette permettant d'intégrer un pod créé au déploiement. On indique ensuite que le pod aura à utiliser le volume créé par le *PersistentVolumeClaim* de tout à l'heure, retrouvé à l'aide de son nom (*test-pvc*). Enfin, vient la description des conteneurs à créer dans le pod (ici, un seul). Ce conteneur aura pour image [nginx](#), retrouvé sur Docker Hub ; il est bien sûr possible de donner n'importe quelle image Docker, y compris une image retrouvée sur un registry privé comme vu dans l'[exercice sur Docker SWARM](#). Sont indiqués pour le conteneur la nécessité de monter le volume prévu pour le pod sur le répertoire */usr/share/nginx/html* à l'intérieur du conteneur, et que le conteneur va écouter sur le port 80.

Ensuite, vient un [service](#). Un service peut être comparé à un répartiteur de charge entre pods. Les pods en question sont ici aussi repérés (*selector*) par leur étiquette. Ici, il s'agit de la même étiquette que celle utilisée par le déploiement ; ainsi, tous les pods créés par ledit déploiement pourront être regroupés au sein du même service. Une fois déployé, un service se voit attribué une adresse IP, accessible uniquement à l'intérieur du cluster Kubernetes. La clause ports indique que le service publiera un *port 80* qui sera redirigé sur les ports (*targetPort*) 80 des pods trouvés.

Pour exposer le service à l'extérieur du cluster, à condition qu'il soit HTTP, on utilise un [Ingress](#). Ici, toute requête HTTP dont le nom d'hôte (*host*) sera *nginx.local* et dont le chemin commencera par / sera transmise au service précédent (de nom *nginx-service*) sur son port 80. Notez que sur un cloud public, on se serait dirigé sur un [service type LoadBalancer](#) à la place d'un Ingress.

On se retrouve avec la configuration suivante (notez que les adresses IP données ne sont que des exemples, et ne peuvent être valides qu'à l'intérieur du cluster Kubernetes ; notez aussi qu'à aucun moment on ne s'est préoccupé de la localisation des pods ni de l'existence des machines qui forment le cluster) :



Lancement d'un cluster Kubernetes

Utilisez **une des** méthodes suivantes pour lancer un cluster Kubernetes. Ces méthodes sont données ci-dessous par ordre de préférence.

Docker et K3d

Une méthode légère et rapide est de lancer un cluster virtuel dans Docker : chaque nœud Kubernetes sera alors simulé par un conteneur. Ceci est permis par la technique dite "Docker-in-Docker" (des conteneurs sont lancés dans le nœud, lui-même étant déjà un conteneur). Il vous faudra alors [installer K3d](#) et [kubectl](#). Vous pourrez ensuite exécuter les commandes suivantes :

```

echo 'mirrors:
  docker.io:
    endpoint:
      - "https://nexus.cluster.ensisa.uha.fr" > reg.yml
k3d cluster create --registry-config reg.yml --agents=2 -p "80:80@server:0" -p "30500:30500@agent:0" # changez le premier 80
par un port disponible sur votre machine
kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/deploy/alternative.yaml
kubectl -n kubernetes-dashboard create serviceaccount admin-user
echo "apiVersion: v1
kind: Secret
metadata:
  namespace: kubernetes-dashboard
  name: admin-user-token
  annotations:
    kubernetes.io/service-account.name: admin-user
type: kubernetes.io/service-account-token" | kubectl apply -f -
kubectl -n kubernetes-dashboard create clusterrolebinding --clusterrole cluster-admin --serviceaccount kubernetes-
dashboard:admin-user admin-user
  
```

ATTENTION Sous Windows, il vous faudra être en mode conteneurs linux : le menu contextuel de la petite icône docker (en bas à droite) devra vous montrer "switch to windows containers" ; si ce n'est pas le case, cliquez sur "switch to linux containers".

ATTENTION Sous Windows, il est possible que *kubectl* ne parvienne pas à dialoguer avec le cluster créé. Il vous faudra remplacer dans le fichier `%HOME%.kube\config` `host.docker.internal` par `127.0.0.1` (en laissant bien le numéro de port).

On pourra alors lancer l'accès au dashboard par `kubectl proxy`, y accéder en utilisant l'url <http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/kubernetes-dashboard:80/proxy/#/login>. Vous trouverez le token d'authentification par la commande `kubectl -n kubernetes-dashboard get secret admin-user-token -o go-template="{{.data.token | base64decode}}"`.

Vous pouvez stopper le cluster par `k3d cluster stop`. On peut le relancer à l'aide de `k3d cluster start`. Il est alors souvent nécessaire de recréer la configuration pour kubectl : `k3d kubeconfig get -a > ~/.kube/config`. Il faudra alors à nouveau remplacer dans ce dernier fichier `host.docker.internal` par `127.0.0.1` sous Windows...

Vous pourrez supprimer votre cluster par un simple `k3d cluster delete`.

Partage du cluster de l'école (beta)

Sur demande, vous pouvez créer votre propre cluster virtuel hébergé dans celui de l'école. Ce cluster ne sera accessible que depuis le réseau de l'école (ou par VPN). Il sera nécessaire de stopper le cluster virtuel à la fin de chaque séance. On peut gérer son cluster sur [l'interface dédiée](#). Il sera aussi nécessaire d'[installer kubectl](#). Demandez à votre enseignant pour obtenir un compte.

Google Cloud

Si vous le préférez vous pouvez aussi utiliser [Google Cloud](#) (GCP) Kubernetes Engine. Vous aurez alors accès à une console en utilisant le bouton "Activer Cloud Shell"  en haut à gauche lorsqu'on consulte la liste des clusters de Kubernetes Engine. Précisez l'identifiant de votre projet (que vous pourrez trouver en utilisant le menu de la barre bleue du haut ) avec la commande `gcloud config set core/project [id projet]`. En cas d'erreur "Unable to connect to the server: x509: certificate signed by unknown authority" lors de l'invocation de kubectl (comme kubectl get pods), invoquez la commande `gcloud container clusters get-credentials nom_cluster --zone nom_zone` (par exemple avec le `nom_cluster` cluster-1 et le `nom_zone` europe-west3-a).

OVH

Encore une autre solution est d'utiliser un "Managed Kubernetes Service" d'OVH. Il faut alors créer un nouveau cluster et lui ajouter 1 ou 2 nœuds. Pour travailler avec kubectl, il vous faudra [l'installer](#) sur votre machine locale (installez aussi les [bash-completions](#) si possible), télécharger le fichier [kubeconfig](#) correspondant à votre cluster Kubernetes créé chez OVH, et le [référer](#) par la variable d'environnement `KUBECONFIG`. Vous pourrez alors voir les nœuds ajoutés précédemment avec `kubectl get nodes`. Vous aurez également à installer [docker engine](#).

Vagrant et VirtualBox

On utilisera un cluster Kubernetes sur 2 machines virtuelles en local. Attention, la configuration qui suit crée des machines virtuelles utilisant chacune 2Go de mémoire (pour un total de 4Go). Avec moins de mémoire, les machines vont se mettre à "swapper" (utilisation excessive du [fichier d'échange](#)) et deviendront inutilisables ; augmentez ce nombre si votre configuration matérielle le permet. Vous aurez besoin de 1,5 Go sur votre disque pour stocker l'image de base et de 10Go supplémentaires pour le stockage des VMs elles-mêmes. Vous pouvez décider de l'endroit où ces dernières seront stockées en modifiant les paramètres de VirtualBox (Général / Dossier par défaut des machines). La mise en route du "cluster" peut se faire à l'aide des commandes suivantes (si vous êtes sur Windows, utilisez git bash - `C:\Program Files\Git\git-bash.exe` - ou un terminal Cygwin) :

```
# vboxmanage setproperty machinefolder C:\\Local\\vagrant\\VM # permettra de modifier le dossier dans lequel la VM sera créée
[ -d vagrant-kubernetes ] || git clone -b microk8s https://github.com/fondemen/vagrant-kubernetes.git
cd vagrant-kubernetes
git pull
# export NODES=1 # si vous êtes juste en ressources, mais vous n'aurez alors plus qu'une seule VM dans votre "cluster"
export K8S_IMAGE=1
export LOCAL_INSECURE_REGISTRIES=127.0.0.1:30500
vagrant up
# vagrant up --provision # à répéter en cas d'erreur
vagrant ssh
```

ATTENTION : sur les machines de l'école, il vous sera peut-être demandé le mot de passe administrateur. Demandez à votre enseignant. Si après cela une erreur type `VERR_INTNET_FLT_IF_NOT_FOUND` apparaît, redémarrez la machine.

En fin de séance, vous n'oublierez pas de supprimer votre cluster à l'aide de la commande `vagrant destroy -f` sur les machines de l'école, ou plus simplement avec `vagrant halt` sur vos machines personnelles, le cluster pouvant alors être relancé avec `vagrant up`. Une fois les exercices (et l'examen) passé, vous pourrez supprimer le cluster (avec `vagrant destroy -f`) et l'image (avec `vagrant box remove fondement/microk8s`).

Lorsque le cluster sera prêt, un dashboard sera alors disponible sur <http://192.168.60.100:8001>.

Vous pouvez tester votre cluster en [appliquant le descripteur exemple](#). Vous pourrez supprimer ensuite tous les services de l'exemple à l'aide de `kubectl delete -f fichier_descripteur.yml`.

Commandes importantes

La commande `kubectl` (abrégée `k` dans le déploiement donné) permet de piloter le cluster kubernetes. [Bash-completion](#) y est installé et vous aidera à savoir à tout moment quelles sont les possibilités en utilisant la touche de tabulation. On trouvera notamment :

- `apply -f [un fichier yaml]` : pour appliquer la configuration contenue dans un fichier yaml
- `delete -f [un fichier yaml]` : pour supprimer les ressources décrites par un fichier yaml
- `get pod,svc,deploy,secrets,pv,pvc,ing` (ou tout autre [type de ressource](#)) : pour avoir des détails sur les ressources données ; l'option `-o wide` peut être utilisée par exemple pour obtenir plus de détails, comme par exemple l'IP ou la localisation d'un pod
- `delete [pod|svc|deploy|secrets|ing|...] [nom de la ressource]` : pour supprimer une ressource en particulier
- `describe [pod|svc|deploy|secrets|ing|...] [nom de la ressource]` : pour avoir des résultats détaillés d'une ressource en particulier
- `logs [nom pod]` : pour savoir ce que fait un pod (sa sortie standard)

- `exec -ti [nom pod] -- [commande à exécuter dans le pod - par exemple /bin/bash]` : pour explorer l'environnement d'exécution d'un pod en particulier
- `rollout restart [deployment|...] [nom déploiement]` : pour redémarrer un déploiement
- `get [pod|svc|deployment|secrets|ing|...] [nom] -o yaml` : pour obtenir un yaml décrivant la ressource, dont sa définition, son statut et les derniers événements le concernant

Pour explorer les ressources déployées sur le cluster, vous pouvez aussi utiliser le dashboard. Pour modifier ou ajouter des ressources, préférez tout de même le travail avec les fichiers yaml pour toujours avoir à un endroit bien identifié vos descripteurs.

Base de données

Vous n'utiliserez pas cette fois Galera mais un simple service [MySQL](#) (ou [MariaDB](#), à votre convenance). Vous créerez pour les besoins de l'exercice une base dédiée à l'application TODO. Créez donc dans un premier temps un déploiement pour la base, ainsi qu'un service pour la base de données. On trouve [un exemple](#) d'un tel déploiement dans la documentation de Kubernetes. Afin de ne pas perdre les données en cas d'arrêt du pod MySQL, pensez à utiliser un [PersistentVolumeClaim](#) d'une [classe](#) disponible sur le cluster (qu'on retrouvera sur le dashboard ou par la commande `kubectl get storageclass`). On verra apparaître le volume et son utilisation avec la commande `kubectl get pv,pvc`. Vous pourrez constater l'apparition d'un volume depuis le dashboard. Sous GCP, on les verra dans le dashboard de Kubernetes Engine (Stockage), et sur OVH comme "Block Storage".

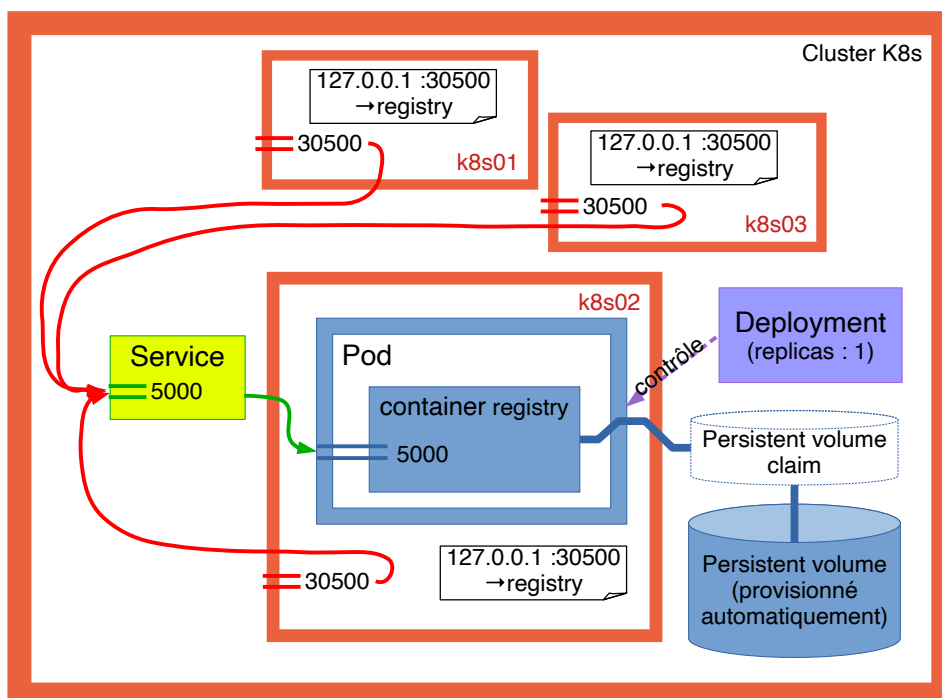
Essayez également de mettre en place des [sondes](#) de démarrage et de disponibilité (healthcheck) à l'aide de la commande `sh -c 'mysqladmin ping -h localhost --protocol=tcp -u root -p${MYSQL_ROOT_PASSWORD} | grep alive'`. Créez une base et un couple utilisateur / mot de passe dédié à l'application TODO. Si vous souhaitez redémarrer la base à partir de l'état initial, vous pourrez tout supprimer à l'aide de la commande `kubectl delete -f [fichier descripteur yaml]`. Attention, au premier démarrage, MySQL a besoin de temps pour s'initialiser, une initialisation mal faite empêchant la base de démarrer. Prévoyez bien des tolérances suffisantes au démarrage pour que le scheduler Kubernetes ne décide pas de redémarrer le pod MySQL en pleine initialisation.

Enfin, stockez les mots de passes dans un [secret](#) qui sera ensuite [injecté](#) dans les variables d'environnement nécessaires. Notez que vous pourrez encoder vos données en base 64 avec la commande `echo -n 'm0N_S3kReT' | base64`.

Registre d'images

Le but de l'exercice est de déployer l'application TODO. Cette application n'étant pas une image téléchargeable depuis un dépôt public (comme Docker Hub), il vous faudra au préalable déployer un service de dépôt d'image docker privé. On se propose ici d'utiliser le dépôt qu'on trouvera sous l'image [registry](#) dans Docker Hub.

Sur le cluster créé par Vagrant ou K3d, vous veillerez à réaliser un service Kubernetes "[NodePort](#)" afin que chaque machine du cluster publie le port du dépôt pour pouvoir y pusher une image docker. Idéalement, vous stockerez les images à l'aide d'un PersistentVolumeClaim monté dans le répertoire `/var/lib/registry` du conteneur. On devra alors se retrouver avec une architecture sur laquelle chaque membre du cluster Kubernetes aura à disposition le (même) registry sur `127.0.0.1:port_choisi` (dans l'exemple 30500, **valeur à utiliser si vous avez lancé votre cluster avec k3d ou vagrant**) :



Vous pourrez tester le bon fonctionnement du registry à l'aide de son [api](#) en utilisant curl `ip:port/v2/`, le résultat attendu étant un simple `{}` :

- sur l'ip du pod (`kubectl get pod -o wide`) port 5000
- sur l'ip du service (`kubectl get svc`) port du service (5000 par défaut - le même que le pod)
- avec 127.0.0.1 comme ip et le port que vous aurez défini comme nodePort

Vous aurez donc à construire l'image de l'application TODO et à la déployer en utilisant l'IP 127.0.0.1 et le port que vous aurez défini (entre 30000 et 32767). Vous pourrez transférer le WAR de l'application (ou tout autre fichier) depuis la machine locale vers la machine virtuelle à l'aide des commandes `vagrant scp [fichier] k3s01:~`. Pour envoyer votre image Docker sur le registry, tagguez votre image avec le nom du service registry :

```
docker build -t 127.0.0.1:30500/todo .
docker push --tls-verify=false 127.0.0.1:30500/todo
```

On devrait alors voir apparaître l'image dans la liste des images connues du registre : `curl http://127.0.0.1:30500/v2/_catalog`.

Si vous travaillez sur un cluster virtuel hébergé à l'école, vous disposez déjà d'un registre.

Sous GCP, vous pouvez essayer d'utiliser [Artifact Registry](#).

Sous OVH, vous aurez à créer un "Managed Private Repository", générer les informations d'identification, faire un `docker login` avec ces dernières, et utiliser comme nom d'image `url_managed_repository/library/nom_image`.

Application TODO

Désormais, chaque nœud du cluster a accès à l'image de l'[application TODO](#) à travers le dépôt d'image que vous venez de déployer. Décrivez un nouveau couple service/déploiement pour TODO, en la connectant au service de la base MySQL préalablement créé. Pour cela, utilisez les variables d'environnement que [l'application utilise à cet effet](#) : `RDS_HOSTNAME` pour le nom d'hôte de la base de données (donc le [nom DNS](#) du service que vous avez créé), `RDS_PORT` pour le port (qui aura été publié par le service), `RDS_DB_NAME` pour le nom de la base à utiliser `RDS_USERNAME` pour le nom d'utilisateur et `RDS_PASSWORD` pour son mot de passe tel qu'enregistré dans le [secret](#) que vous aviez créé à la précédente étape, et enfin `RDS_OPTIONS` sera positionné à `"useSSL=false&allowPublicKeyRetrieval=true"`. Enfin, positionnez `AUTH` à `"false"`. Notez que toutes ces variables d'environnement sont spécifiques à l'application, qui a été explicitement codée pour lire ces variables d'environnement là, et adapter son comportement en fonction. On trouve [un exemple](#) dont vous pourrez vous inspirer (utilisant lui des secrets générés automatiquement) de déploiement d'un service web (ayant lui une image publique) utilisant une base dans la documentation de Kubernetes. Vous constaterez la bonne connection à MySQL en regardant les logs du pod todo :

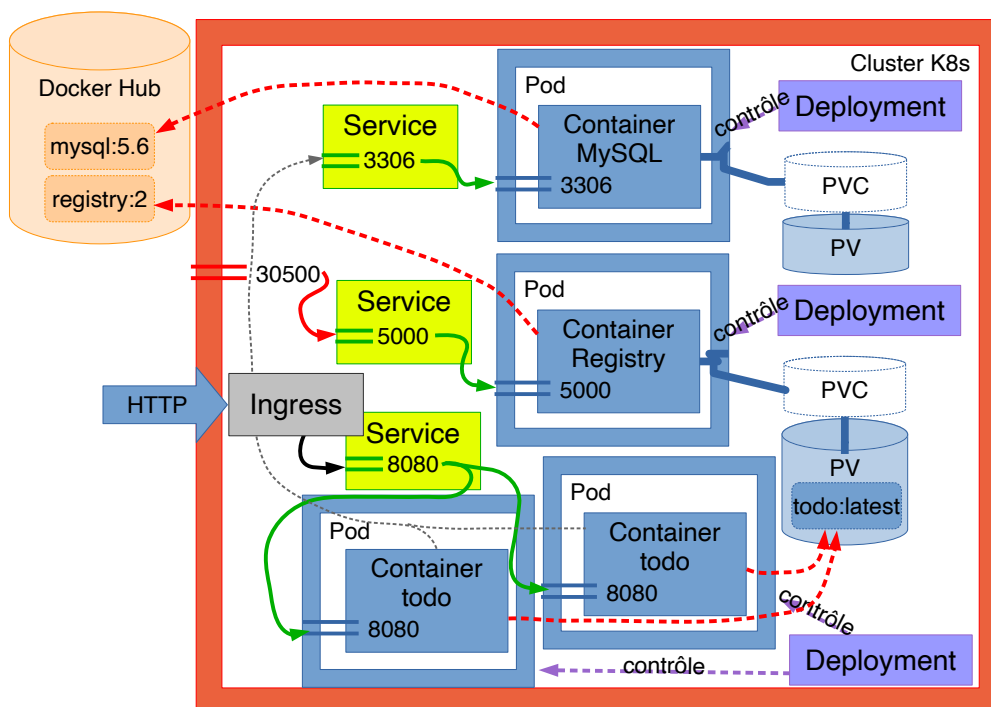
```
Getting remote connection with connection string from RDS_* environment variables. jdbc:mysql://mysql_host:port/dbname
...
Remote connection to mysql successful.
```

En cas d'indisponibilité de la base, l'application TODO stockera ses données en mémoire ; ainsi, si l'application démarre plus vite que la base (ce qui sera le cas par exemple si la base doit s'initialiser), elle n'utilisera pas la base que vous lui avez indiquée. Pour s'assurer que les pods de l'application TODO ne démarrent pas avant que la base de données ne soit disponible, vous pourrez utiliser un [conteneur d'initialisation](#) (qui devra avoir fini sans code d'erreur pour que le conteneur de l'application soit autorisé à démarrer). Pour ce faire, je vous conseille l'image [toschneck/wait-for-it](#) ; il vous faudra alors donner la commande `/wait-for-it.sh` suivie de l'*adresse:port* d'accès à la base de données (l'adresse étant le nom du service de la base de données, le port étant le port par défaut de la base de données - 3306 pour MySQL ou MariaDB). Le conteneur d'initialisation attendra alors que ce port derrière cette adresse accepte une connection (indiquant ainsi que la base de données est prête) ou terminera en échec au bout d'un certain temps. Dans ce dernier cas, le pod terminera en échec, et Kubernetes le replanifiera son démarrage pour plus tard.

Pour vérifier que l'application TODO est bien active, vous pourrez demander l'IP du service à l'aide de `kubectl get svc` et charger la page à l'aide de `curl -L` (-L pour suivre les [redirections 30X](#)) depuis une des machines du cluster. Afin de voir l'application à l'extérieur, créez un ingress qui vous permettra d'avoir un résultat sur [http://\[ip du cluster\]](#). Pour simplifier les tests, évitez d'utiliser un nom d'hôte comme règle de routage, et privilégiez un path positionné sur `/`.

Si vous avez opté pour GCP (Google) ou OVH, utilisez un service de type [LoadBalancer](#) à la place. Vous pourrez alors voir son IP avec `kubectl get svc`. Vous verrez alors le répartiteur de charge apparaître dans le dashboard dans les "Services Réseau" / "Équilibrage de charge" sur GCP.

On devrait finalement obtenir l'architecture suivante :



Passage à l'échelle de l'application

Il serait maintenant intéressant de multiplier le nombre d'instances de l'application. Ceci peut paraître simple, car il suffit d'ajouter la directive `replicas` dans le [descripteur de déploiement](#), ou d'utiliser la commande `kubectl scale deploy [nom déploiement] --replicas=[nombre de répliques]` pour multiplier le nombre de répliques. Cependant, cette application a la mauvaise idée d'avoir un état, à savoir le nom de la personne connectée ainsi que le token de sécurité qui lui est associé.

Une première idée est de configurer une [affinité de session](#) pour tenter de rediriger au mieux un même client vers le même pod. Attention, l'affinité de session ne sera effective que lorsque vous aurez défini le nom du cookie destiné à retenir le backend à utiliser. Cependant, cela ne résistera pas au redémarrage d'un pod.

Si vous avez réalisé l'exercice permettant de [partager les informations de session à l'aide d'Hazelcast](#), vous savez comment résoudre ce problème. Vous pourrez donc réutiliser le résultat de cet exercice (en positionnant `AUTH` à `"true"`). Cependant, la manière dont Hazelcast retrouve ses pairs (les autres instances auxquelles il faut se connecter pour former le cluster Hazelcast) ne fonctionne pas lorsque l'instance est encapsulée dans un conteneur dans un pod. Adaptez la configuration d'Hazelcast de manière à ce que ce dernier retrouve ses pairs [à l'aide du nom DNS du service](#). Notez qu'il faudra un [service headless](#) pour l'application TODO et utiliser son [nom DNS complet](#) (non abrégé).

L'application pourra déterminer si elle est à l'intérieur d'un pod si elle trouve le fichier

`/var/run/secrets/kubernetes.io/serviceaccount/namespace` contenant le nom du namespace Kubernetes dans lequel elle s'exécute (par défaut `default`). Il vous faudra sans doute tout de même ajouter une variable d'environnement pour indiquer à l'application le nom du service Kubernetes qui lui a été attribué. Si vous modifiez le code de votre service, il faudra le recompiler (`mvn package` dans `app`), le copier dans une VM (`vagrant scp ...`), construire l'image (`docker build ...`), la publier dans le dépôt (`docker push ...`) et relancer le déploiement (`kubectl rollout restart deployment nom_déploiement`).

Vous constatez le bon fonctionnement du cluster Hazelcast en vous identifiant sur l'application. Vous pourrez voir dans les logs d'un pod pour l'application le démarrage du cluster Hazelcast:

```
INFO [localhost-startStop-1] com.hazelcast.system.null [192.168.236.130]:5701 [dev] [3.12.6] Hazelcast 3.12.6 (20200130 - be02cc5) starting at [192.168.236.130]:5701
...
INFO [localhost-startStop-1] com.hazelcast.spi.discovery.integration.DiscoveryService.null [192.168.236.130]:5701 [dev] [3.12.6] Kubernetes Discovery properties: { service-dns: nom_DNS_du_service_todo, ... }
INFO [localhost-startStop-1] com.hazelcast.spi.discovery.integration.DiscoveryService.null [192.168.236.130]:5701 [dev] [3.12.6] Kubernetes Discovery activated resolver: DnsEndpointResolver
INFO [localhost-startStop-1] com.hazelcast.instance.Node.null [192.168.236.130]:5701 [dev] [3.12.6] Activating Discovery SPI Joiner
...
INFO [hz.todo.cached.thread-3] com.hazelcast.nio.tcp.TcpIpConnector.null [192.168.236.130]:5701 [dev] [3.12.6] Connecting to /IP_d'un_autre_pod_du_déploiement:5701, timeout: 10000, bind-any: true
INFO [hz.todo.IO.thread-in-0] com.hazelcast.nio.tcp.TcpIpConnection.null [192.168.236.130]:5701 [dev] [3.12.6] Initialized new cluster connection between /192.168.236.130:5701 and /192.168.236.131:53859
INFO [hz.todo.IO.thread-in-1] com.hazelcast.nio.tcp.TcpIpConnection.null [192.168.236.130]:5701 [dev] [3.12.6] Initialized new cluster connection between /192.168.236.130:57185 and /192.168.235.130:5701
INFO [hz.todo.cached.thread-3] com.hazelcast.nio.tcp.TcpIpConnector.null [192.168.236.130]:5701 [dev] [3.12.6] Connecting to /IP_d'encore_un_autre_pod_du_déploiement:5701, timeout: 10000, bind-any: true
INFO [hz.todo.IO.thread-in-2] com.hazelcast.nio.tcp.TcpIpConnection.null [192.168.236.130]:5701 [dev] [3.12.6] Initialized new cluster connection between /192.168.236.130:53991 and /192.168.236.131:5701
INFO [hz.todo.generic-operation.thread-0] com.hazelcast.internal.cluster.ClusterService.null [192.168.236.130]:5701 [dev] [3.12.6]

Members {size:3, ver:3} [
  Member [192.168.235.130]:5701 - 5179b4a9-7ddc-4a41-a395-ca5cda5dfb96
  Member [192.168.236.131]:5701 - a8610144-c3cb-41d9-9020-d6cda1baaa3a
  Member [192.168.236.130]:5701 - a3ca07cc-30ae-4aab-bd28-cf5887f6be44 this
]

INFO [localhost-startStop-1] com.hazelcast.core.LifecycleService.null [192.168.236.130]:5701 [dev] [3.12.6] [192.168.236.130]:5701 is STARTED
```

Sous OVH, vous aurez aussi à installer un [ingress controller](#) à l'aide de Helm. En effet, le service pour todo ne pouvant être à la fois headless et LoadBalancer, vous devrez passer par un Ingress. Il faudra donc [installer Helm](#) en premier, puis on pourra installer le contrôleur d'ingress Nginx de la manière suivante :

```
helm repo add nginx-stable https://helm.nginx.com/stable
helm repo update
helm install ni nginx-stable/nginx-ingress
kubectl get svc ni-nginx-ingress --watch # Ctrl+c dès que EXTERNAL-IP est présent - peut prendre quelques minutes
```

Notez que vous aurez systématiquement à donner un nom d'hôte à vos Ingress. Il faudra donc utiliser l'adresse IP de l'external ip du load balancer (qu'on peut trouver par `nslookup $(kubectl get svc ni-nginx-ingress -o jsonpath='{.status.loadBalancer.ingress[0].hostname}')`): `curl -L ip_load_balancer -H 'Host: host_ingress'`. Vous pouvez aussi ajouter une ligne `ip_load_balancer host_ingress` dans le fichier `/etc/hosts` (ou `C:\windows\system32\drivers\etc\hosts` sous Windows) pour le voir sur un navigateur.

Sécurisation des échanges réseau

Seule l'application TODO est supposée accéder à la base de donnée MySQL créée plus haut. Autrement dit, seuls les pods correspondant à l'application TODO ont le droit de requêter le pod correspondant à MySQL sur son port 3306, et inversement, les seuls pods auxquels le pod MySQL a le droit de répondre sont les pods de l'application TODO.

De même, seuls les pods de l'application TODO doivent avoir le droit de former un cluster Hazelcast entre eux. Autrement dit, seuls les pods TODO on le droit de requêter le port 5701 sur un pod TODO. Cependant, leur port 8080 doit rester ouvert à toute autre communication.

Rédigez un ensemble de [NetworkPolicies](#) résumant ces règles. Vous pourrez tester la bonne application de ces règles en vous trompant volontairement de port (3307 au lieu de 3306, ...). Notez qu'il sera peut-être nécessaire de relancer les déploiements todo et/ou mysql après l'application d'une règle réseau, les connections établies (à durée de session longue) échappant aux nouvelles règles. Vous pourrez relancer un déploiement avec la commande `kubectl rollout restart deployment nom_déploiement`, et suivre l'avancée de son application avec la commande `kubectl get pod --watch` (Ctrl+c pour reprendre la main).

Modifié le: mercredi 25 juin 2025, 17:03

[Retour au cours](#)