Conteneurisation et orchestration

Aurélien CHATILLON - Mohammad-Reza SHAKIBAEI - Arthur KOZIOR

Vincent LECLERC – Paris Ynov Campus

Table des matières

[Déploiement 3](#_Toc100664749)

[Sonde Livness et Readness 6](#_Toc100664750)

[Mise en place de Prometheus 7](#_Toc100664751)

[Déploiement automatisé 10](#_Toc100664752)

[Journaux 11](#_Toc100664753)

[Certificat 14](#_Toc100664754)

[Créer une demande de signature de certificat 14](#_Toc100664755)

[Créer un objet CertificateSigningRequest à envoyer à l'API Kubernetes 14](#_Toc100664756)

[Faire approuver la CertificateSigningRequest 15](#_Toc100664757)

[Signer le CertificateSigningRequest 15](#_Toc100664758)

[Créer une autorité de certification 15](#_Toc100664759)

[Délivrer un certificat 15](#_Toc100664760)

[Télécharger le certificat signé 16](#_Toc100664761)

[Téléchargement du certificat et utilisation 16](#_Toc100664762)

[Conclusion 18](#_Toc100664763)

# Introduction

Lors de notre cursus en M1 informatique spécialité Infrastructure & r&seaux, dans le cadre notre enseignement ils nous a été de demandé de réalisé un projet conteneurisation et orchestration d’une application web de type micro-service, en utilisant les technolgies de notre choix pour la partie conteneur et Kubernetes pour la partie orchestration.

L'objectif est le déploiement, la maintenance et la mise à jour de différents micro-services REST et de l’applications web dans un environnement conteneurisé, avec un orchestrateur Kubernetes afin d'assurer la haute disponibilité des apps.

# Déploiement

Pour ce projet nous devons déployer et maintenir des micro-services REST et applications web dans un environnement conteneurisé, avec un orchestrateur Kubernetes.

Les applications sont fournies avec leurs relations décrites dans un docker-compose.

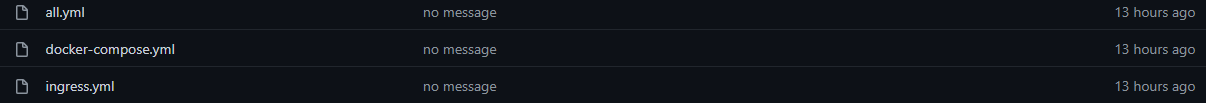
Donc pour commencer nous devons lire le docker-compose correctement afin de bien comprendre et ne pas partir dans tous les sens.

Une image contenant texte, écran, noir, capture d’écran

Description générée automatiquement

Dans ce dossier, notre premier jet pour déployer les applications.

Nous y étions presque quand vous avez mis à jour le github, et pour ne pas perdre plus de temps et attaquer les choses sérieuses, nous avons décidé de se lancer avec vos fichiers.



Grace a ces fichiers le déploiement c’est fait sans soucis.

Une image contenant texte

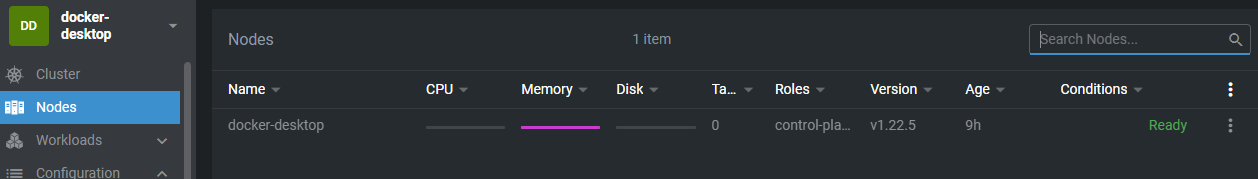
Description générée automatiquement

Une fois que le déploiement sur docker est fait, j’active Kubernetes sur Docker DesktopUne image contenant texte

Description générée automatiquement

Et j’installe aussi Lens pour avoir une interface graphique et bien comprendre ce que je fais.

* Swarm init pour créer le cluster
* kubectl config set current-context “name” pour choisir le cluster



Nous pouvons voir que je possède bien un conteneur actif, mon docker-desktop

Une image contenant texte

Description générée automatiquement

Ainsi que tous les services actifs.

Mais ce n’est pas terminé, il faut maintenant pouvoir accéder au service web, et pour cela nous avons besoin d’un ingress qui fasse la liaison avec une connexion externe, notre pc.

Pour cela je récupère un ingress nginx depuis un repo, et je l’installe grâce a helm.

Une image contenant texte

Description générée automatiquement

La connexion est assurée, ici l’erreur vient simplement du fait qu’il y a un conflit de nom de services. Ce qui n’impacte pas notre travail.

Une image contenant texte

Description générée automatiquement

# Sonde Livness et Readness

Il existe plusieurs types de sonde afin de faire un bilan de santé de nos services sur kubernete, ici nous avons utilisé une sonde liveness afin de détecter quand redémarrer un conteneur. Elles permettent de rattraper les deadlocks dans les cas où l’application est en cours d'exécution, mais qu’il est incapable de traiter les requêtes. Mais nous avons aussi utilisé une sonde readiness pour savoir quand un conteneur est prêt à accepter le trafic.

Ci-dessous un exemple de sonde readness/livness mis en place pour l’api web.

Une image contenant texte

Description générée automatiquement

Pour le mettre en place il vous suffira d’exécuter la commande suivante : kubectl appli -f livness-exec.yml

# Mise en place de Prometheus

Qu’est-ce que Prometheus ?

Prometheus est un logiciel de monitoring open-source utilisée afin de surveiller des systemes informatique et de générer des alertes. Pour cela il enregistre des métriques en temps réel dans une base de données de séries temporelles en se basant sur le contenu de point d'entrée exposé à l'aide du protocole HTTP.

Comment mettre en place Prometheus ?

Option 1 : via l’IDE Lens

Pour mettre en place Prometheus nous pouvons directement passé par l’IDE lens qui propose via les paramètres de le mettre en place facilement pour un cluser en activant l’option.

Une image contenant texte

Description générée automatiquement

Option 2 : Via les fichiers de config yaml

Etape 1 : Mise en place d’un Name space

Pour commencer nous créons un Name space Kubernetes nos futurs composants afin d’effectuer la surveillance. Si cet espace dédié n’est pas ils sont déployés au Name space par défaut.

Une image contenant texte

Description générée automatiquement

Commande : kubectl create -f orch\_monitoring\_ns.yml

Etape 2 : Mise en place du cluster Rôle

Dans le fichier yml cluster rôle va être ajouté les autorisations get, list et watch aux nœuds, points de terminaison de services, pods et entrées. Le cluser rôle est lié au name space monitoring crée préalablement. Dans notre cas il va être utile afin de récupérer des métriques à partir de n'importe quel objet stipulé eu sein de celui-ci.

Une image contenant texte

Description générée automatiquement

Commande : kubectl create -f orch\_CluserRole.yml

Etape 3 : Mise en place de la config map

La Configmap est un objet Kubernetes qui va nous permettre le maintien d’une valeur clé qui facilement être utilisée par d'autres objets Kubernetes tels que pod, déploiements et services. Ci-dessous la config map réalisée pour prometheus.



Commande : kubectl create -f orch\_ConfigMap.yml

Etape 4 : Mise en place du fichier de déploiement + service

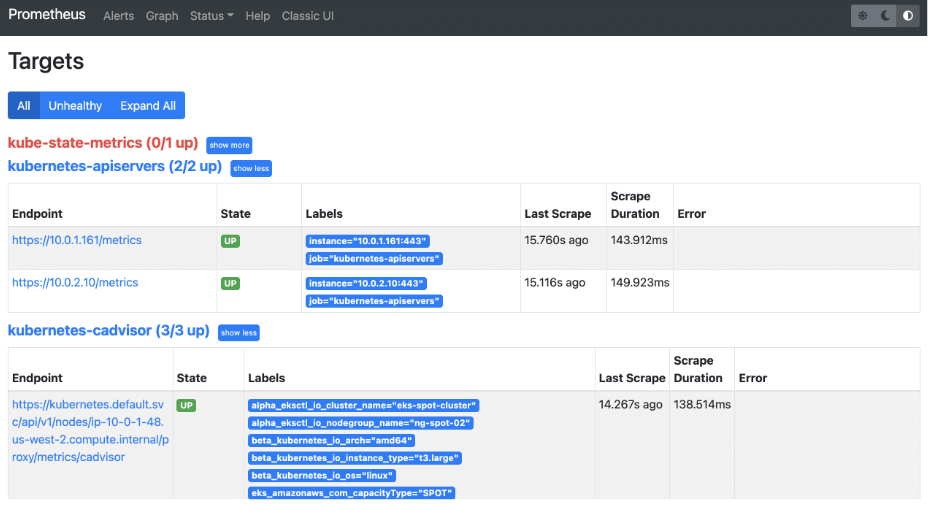
Créez un fichier nommé orch\_prometheus.yml afin de mettre en place l’outil ainsi que son service. Ci-dessous notre fichier yml.



Commande : kubectl create -f orch\_prometheus.yml

Conclusion :

Une fois ces étapes effectuées et le port de prometheus forwardé via Lens. Vous aurez directement accès via le localhost à l’outil et pourrez désormais surveiller les composantes de votre application.



# Déploiement automatisé

Pour le déploiement automatisé, nous allons utiliser une infrastructure générer avec helm create permettant de déployer un notre livrable sur un environnement cible. Pour cela il faut suivre les étapes suivantes.

Etape 1 :

Via votre terminal exécuter la commande « helm create autoDeploy » , cela va permettre de crée un Template de déploiement helm. Ci-dessous l’arborescence du fichier généré.

Une image contenant texte

Description générée automatiquement

Une fois le fichier obtenu il nous suffit d’y effectuer les remplacements nécessaires dans le fichier Template, en mettant à jour les yml qui nous intéresse.

Une image contenant texte

Description générée automatiquement

Ci-dessous un exemple pour l’api job uniquement.



Pour lancer le déploiement il vous suffira d’exécuter la commande suivante :

helm install --dry-run --debug ./autoDeploy --set service.internalPort=8286

# Journaux

**Pour pouvoir exploiter les logs correctement il faudra mettre en place une pile complète (EFK ou ELK) afin de fournir une gestion de logs avancée.**

Choix entre ELK ou EFK

ELK et EFK utilise 2 meme logiciel

E = Elastishearch :

* Elastishearch est un moteur de recherche et d'analyse.

K = Kibana :

* Kibana permet aux utilisateurs de visualiser des données avec des tableaux et des graphes dans Elasticsearch.

L = Logstash :

* Logstach est un pipeline côté serveur, destiné au traitement des données. Il fait partie de la suite Elastic

F = Fluentd

* Fluentd est, comme Logstash dans la pile ELK, est également un collecteur de données open source, qui vous permet d'unifier la collecte et la consommation de données.

J’ai choisi de faire tous l’exploitation des logs avec la suite Elastic( ELK).

Dans un premier temps on met en place Elasticsearch :

On crée un fichier .yml pour la mise en place de elasticsearch

Une image contenant texte

Description générée automatiquement

SI tous est bon

On peut allez sur le localhost :9200 ce qui nous montre cet page avec les différents paramètre.

Une image contenant texte

Description générée automatiquement

Dans un second temps on met en place Kibana :

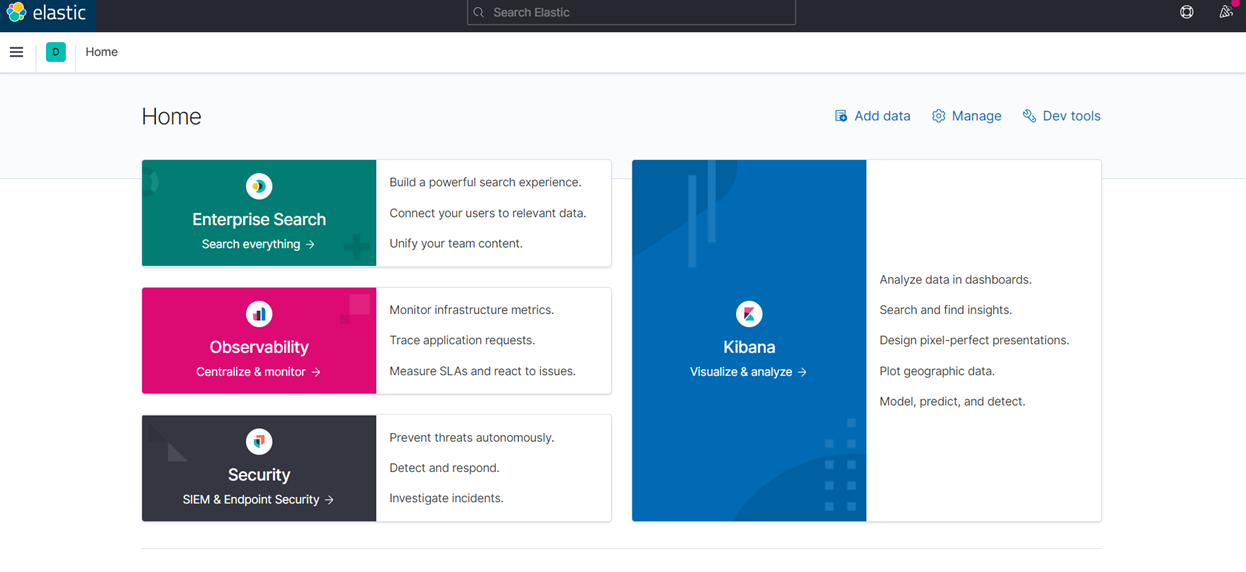
Pareil que pour eleasticsearch. On crée un fichier .YML

Une image contenant texte

Description générée automatiquement

SI tous est bon

On peut allez sur le localhost :5601 ce qui nous montre cet page avec les différents paramètre graphe de kibana.



Dans un troisième temps on met en place Logstash :

Pareil que pour les 2 dernier. Un fichier .yml pour la mise en place

Une image contenant texte

Description générée automatiquement

En fin on configure logstrash pour la récupération des logs :

On crée une configuration UDP(composant de suite de protocole internet utiliser pour envoyer des message courts) de terminaison GELF( pilote de journalisation) sur le port 12201 (par defaut) .

Cela gênera les journaux sur l’hôte elastisearch

Une image contenant texte

Description générée automatiquement

Il faut modifier le pilote de journalisation par défaut de Docker pour que chaque conteneur créé envoie automatiquement les journaux vers le conteneur logstash. Cela se fait en éditant le /etc/docker/daemon.jsonfichier de configuration

Une image contenant texte

Description générée automatiquement

# Certificat

**Pour sécuriser on a besoin d’utiliser un certificat, ce qui demande plusieurs étapes que je vais décrire ici :**

## Créer une demande de signature de certificat

Je génère une clé privée et une demande de signature de certificat (ou CSR) en exécutant la commande suivante :

Une image contenant texte

Description générée automatiquement

Cette commande génère deux fichiers ; il génère server.csr contenant la demande de certification server-key.pem encodée PEM et contenant la clé encodée PEM du certificat qui reste à créer.

## Créer un objet CertificateSigningRequest à envoyer à l'API Kubernetes

Je génère un manifeste CSR (en YAML) et je l’envoie au serveur d'API :

Une image contenant texte

Description générée automatiquement

Le CSR devrait maintenant être visible depuis l'API dans un état En attente. Avec la commande :

kubectl describe csr my-svc.my-namespace

## Faire approuver la CertificateSigningRequest

L'approbation de la demande de signature de certificat est effectuée un administrateur de cluster. Je le fais manuellement :

kubectl certificate approve my-svc.my-namespace



Cela signifie que la demande de certificat a été approuvée et attend que le signataire demandé la signe.

## Signer le CertificateSigningRequest

Ensuite, je joue le rôle d'un signataire de certificat, j’émets le certificat et je le télécharge sur l'API.

## Créer une autorité de certification

J’ai besoin d'une autorité pour fournir la signature numérique sur le nouveau certificat :

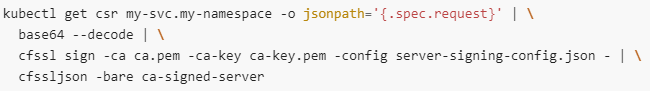
Une image contenant texte

Description générée automatiquement

Cela produit un fichier de clé d'autorité de certification ( ca-key.pem) et un certificat ( ca.pem).

## Délivrer un certificat

J’utilise un server-signing-config.jsonconfiguration de signature et le fichier de clé et le certificat de l'autorité de certification pour signer la demande de certificat :



Cela produit un fichier de certificat de service signé, ca-signed-server.pem

## Télécharger le certificat signé

Enfin, je remplis le certificat signé dans le statut de l'objet API :



Avec la commande kubectl get csr



## Téléchargement du certificat et utilisation

je télécharge le certificat émis et l'enregistre dans un server.crt en exécutant ce qui suit :



Maintenant, je peux remplir server.crt et server-key.pem dans un Secret que mets ensuite dans un pod pour https.



Au final je rajoute quelques lignes dans l’ingress pour la cohérence des clés, et surtout je change les ports, de 80 (http) vers 443 (https).

Commande pour créer le secret avec le certificat et la clé ainsi que le nom de l’ingress

kubectl create secret tls nom\_édition-ingress-tls-secret --cert=./certificate.pem --key=./key.pem [--namespace espace\_nom]

Je sécurise l’Ingress en définissant un secret qui contient une clé privée et un certificat TLS. Le secret de TLS doit contenir les clés tls.crt et tls.key contenant le certificat et clé privée à utiliser pour TLS :

Une image contenant texte

Description générée automatiquement

Il me reste plus qu’a référencer ce secret dans l’ingress.

Référencer ce secret dans un Ingress indiquera au contrôleur d'Ingress de sécuriser le canal du client au load-balancer à l'aide de TLS.

Une image contenant texte

Description générée automatiquement

On n’oublie pas de changer le port

// ! \\ dans le rendu il n’y a pas le secret ni l’ingress modifié, car on a déjà fait la présentation, ca serait de la triche si nous avions rajouté après, mais nous avons compris le principe et nous savons le faire // ! \\

# Conclusion

Avec ce projet nous avons tous les trois beaucoup appris, ce sont des notions nouvelles et que certains d’entre nous utilise déjà dans leurs alternances. Malgré une prise en main un peu tardive du projet nous avons compris le principe de conteneurisation et d’orchestration, qui, nous pensons est le plus important.

Après la présentation de vendredi nous avons choisi de ne pas intégrer plus de contenus pour ne pas « tricher » car le temps était imparti, malgré le fait que nous avons bien compris les étapes et concept du cours et du projet.

Nous avons néanmoins rajouté un maximum d’informations avec des screens précis et parlant pour bien expliquer le projet.