Formation Kubernetes

-

Pratique

kubernetes

Atelier 6

-

Services et accès externes

Table des matières

1. Services 3
   1.1. Principes de la ressource Service 3
   1.2. Création d'un service mono-pod 4
   1.3. Service et commande ping 5
   1.4. Service multi-pods 6
   1.5. Fonctionnement de la découverte service<>pods 6
   1.6. Persistance de session 7
   1.7. Service multi-ports 8
   1.8. DNS et service discovery 10
2. Accès externe aux services 12
   2.1. Type NodePort 12
   2.2. Type Load-Balancer 14
   2.3. Ingress 17
   2.4. Ingress multi-services 19
   2.5. Sécurisation de l'Ingress en TLS 20

1. Services

1.1.Principes de la ressource Service

Le fonctionnement des pods s'accompagne de certaines contraintes qui limitent la gestion automatisée de vos micro-services :

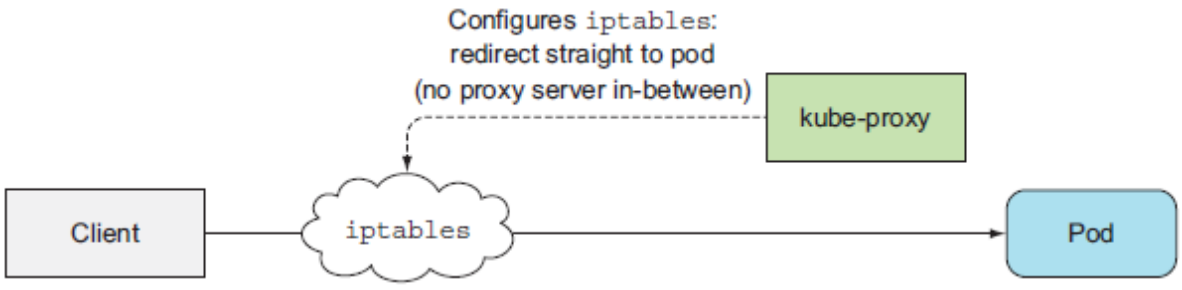
- Un pod est éphémère et peut être redémarré par **Kubernetes**
- Un pod peut changer d'IP après un redémarrage

Ce fonctionnement nécessiterait donc de devoir reconfigurer vos micro-services interconnectés dès qu'un pod redémarre. De plus, si vous souhaitez scaler horizontalement vos services, il faut un mécanisme de répartition de charge en frontal de vos pods.

La ressource **Service** de **Kubernetes** va permettre d'adresser ce besoin en fournissant un point d'entrée unique et constant pour accéder à vos pods via l'exposition d'une IP et d'un port.

À titre informatif, le fonctionnement des **Services** repose sur l'utilisation du composant kube-proxy présent sur chaque node :

- Kube-proxy manipule les règles d'iptables afin de récupérer le flux vers le service et le port.
- Puis pour chaque pod sous la gestion d'un service, appelé endpoints, il installe les règles d'iptables vers ce pod.
- Le mécanisme de redirection vers les pods se base sur un mode aléatoire



Nous allons créer un manifeste de **Service** pour décrire son fonctionnement.

1.2. Création d'un service mono-pod

Comme pour les **Replicaset** vu précédemment, la ressource **Service** va également se baser sur la notion de label et de selector pour permettre au **Service** d'identifier les pods qu'il doit définir comme endpoints.

1) Créez le manifeste suivant et exécutez-le via la commande « kubectl create -f [fichier] ».

```yaml
kind: Service
apiVersion: v1
metadata:
  name: webservice
spec:
  selector:
    app: generatorsvc
  ports:
  - port: 81
    targetPort: 80
```

Ce **Service** a pour but d'identifier les pods dont le label est « app=generatorsvc » et d'exposer le port 81 qu'il redirigera vers le port 80 du pod cible.

2) Affichez le service avec l'une des commandes suivantes :

```
kubectl get service
```

```
kubectl get svc
```

Le service créé expose le port 81 sur l'IP attribuée au service au sein du cluster. À noter que ce mode par défaut « Cluster IP » du service lui permet d'être joignable depuis tout pod du cluster et ainsi permettre à vos micro-services de communiquer au sein du cluster.

À ce stade, le service est créé, mais aucun pod n'est associé.

3) Créez un premier pod « fortune » sur le modèle suivant :

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: fortune
  labels:
    app: generatorsvc
spec:
  containers:
  - name: webfront
    image: ilkiformation/webapache:v1
    ports:
    - containerPort: 80
    - containerPort: 22
  - name: generator
    image: ilkiformation/fortunegen:v1
```

4) Connectez-vous en SSH sur l'une des nodes du cluster depuis GCP et lancez un curl sur l'IP « cluster IP » du **Service** créé et sur le port 81 pour vérifier le bon fonctionnement.

```
$ curl http://IP_Service:81
```

1.3.Service et commande ping

1) Avant de poursuivre avec un second pod, testez le fonctionnement du ping sur le **Service** pour juger si cette commande sera pertinente pour la résolution de problème dans **Kubernetes**. En effet ,le ping est généralement la première commande utilisée pour résoudre un problème réseau.

```
$ kubectl exec -it nom-du-pod bash
root@nom-du-pod:/# apt-get install iputils-ping
root@nom-du-pod:/# ping IP-service
```

- Que constatez-vous ?

L'explication de ce comportement s'explique par le fonctionnement de la ressource **Service** dans **Kubernetes** qui est une IP virtuelle et qui n'autorise l'accès que si un port est explicitement précisé.

1.4.Service multi-pods

1) Ajoutez maintenant un second pod au service basé sur le manifeste suivant :

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: chucknorris
  labels:
    app: generatorsvc
spec:
  containers:
  - name: webfront
    image: ilkiformation/webapache:v1
    ports:
    - containerPort: 80
    - containerPort: 22
  - name: generator
    image: ilkiformation/chuckgen:v1
```

2) Connectez-vous à nouveau en ssh sur l'une des nodes du cluster depuis GCP et lancez un curl sur l'IP « cluster IP » du service créé et sur le port 81 pour vérifier le bon fonctionnement.

```
$ curl http://IP_Service:81
```

   - Réalisez plusieurs fois le curl.
   - Vous devez constater que vous tombez sur l'un ou l'autre des générateurs, à savoir « chucknorris » ou « fortune ».
   - Déterminez le mode de répartition de sessions utilisé par le service

3) Vous pouvez également utiliser la commande ***« kubectl describe service »*** pour constater la présence des 2 pods dans la section endpoints.

1.5.Fonctionnement de la découverte service<>pods

1) Vous allez maintenant supprimer votre **Service**, constater sa bonne suppression, puis recréer le **Service**.

- Vérifiez le fonctionnement du service
- Que constatez-vous ?
- Supprimez puis recréez vos 2 pods
- Vérifiez à nouveau le fonctionnement du service
- Que constatez-vous ?

Ce comportement s'explique par le fait qu'au démarrage d'un pod, **Kubernetes** paramètre un certain nombre de variables d'environnement au sein du pod pour lui indiquer les **Services** existants dans le cluster.

2) Exécutez la commande suivante pour afficher les variables d'environnement définies dans le pod et constater l'existence des informations associées au service :

```
$ kubectl exec nom-du-pod env
```

Vous devez retrouver les informations d'adresse IP et de port de votre service au niveau des variables suivantes :

- *NOM_SERVICE*_SERVICE_HOST=
- *NOM_SERVICE*_SERVICE_PORT=

1.6. Persistance de session

1) Lancez à nouveau la commande ***« kubectl describe service »***:
   - Quelle est la valeur du paramètre « Session Affinity » ?

2) Depuis l'une des nodes sur GCP, exécutez à nouveau la commande suivante plusieurs fois pour vous assurer que vos 2 pods sont bien fonctionnels.

```
$ curl http://IP_Service:81
```

3) Modifiez le manifeste de votre service comme suit en utilisant la spécification « sessionAffinity »

```yaml
kind: Service
apiVersion: v1
metadata:
  name: webservice
spec:
  sessionAffinity : ClientIP
  selector:
    app: generatorsvc
  ports:
  - port: 81
    targetPort: 80
```

4) Exécutez maintenant la commande ***« kubectl apply »*** pour mettre à jour votre service existant :

Note : en cas de warning, n'en tenez pas compte pour le moment et poursuivez.

5) Affichez les informations de votre service avec la commande « kubectl describe ».
   - Vérifier le bon paramétrage du paramètre « SessionAffinity »

6) Depuis l'une des nodes sur GCP, exécutez à nouveau la commande suivante plusieurs fois pour vous assurer que la persistance de session fonctionne désormais correctement.

```
$ curl http://IP_Service:81
```

7) Pour connaitre les valeurs possibles de "sessionAffinity", utiliser la commande ***« kubectl explain ».***

La raison pour laquelle si peu de valeurs sont possibles et notamment l'absence, par exemple, de persistance de session par cookie, s'explique par la nature des **Services**. En effet, les **Services** fonctionnent uniquement au niveau TCP/UDP et n'ont pas de visibilité sur les couches hautes et notamment HTTP.

1.7.Service multi-ports

Il peut être utile de disposer de services multi-ports pour un même ensemble de pods. Le cas le plus classique concerne notamment un service web qui écouterait en non-sécurisé sur HTTP et en sécurisé HTTPS.

Pour valider ce fonctionnement, nous allons nous baser sur les manifestes de pods existants qui exposent le port 80 et le port 22.

1) Avant de démarrer, supprimez le service et les pods existants.

2) Éditez votre manifeste de service et modifiez-le comme suit :

```yaml
kind: Service
apiVersion: v1
metadata:
  name: webservice
spec:
  selector:
    app: generatorsvc
  ports:
  - name: http
    port: 81
    targetPort: 80
  - name : ssh
    port: 22
    targetPort: 22
```

3) Créez ce nouveau service double ports puis les pods associés.
   - Validez le bon fonctionnement du service en vérifiant les endpoints avec ***« kubectl describe »***

4) Testez une connexion en SSH depuis l'une des nodes du cluster sur GCP sur votre nouveau service.
   - Voici les informations de connexion sur le pod en SSH :
     - Login : root
     - Mot de passe : azerty

1.8.DNS et service discovery

Lors des précédents ateliers, vous avez notez l'existence d'un namespace **kube-system** dans lequel est présent un service appelé **kube-dns**. Ce service exécute un serveur DNS au sein du cluster.

Tous les pods qui démarrent sur le cluster sont automatiquement gérés par ce service **kube-dns** qui va modifier le fichier */etc/resolv.conf* dans chaque container.

Note : par défaut, le DNS du service **kube-dns** est utilisé par les pods. Il est néanmoins possible de préciser l'usage d'un DNS externe au cluster via la spécification de pod « DnsPolicy ».

1) Connectez-vous dans un pod et affichez le contenu du fichier */etc/resolv.conf*.
   - Vérifier la valeur du DNS via « nameserver »
   - Afficher les services du namespace et vérifier que c'est bien l'IP de **kube-dns** qui est ici paramétrée

Vous constatez également que plusieurs suffixes et extensions sont présents dans ce fichier resolv.conf.

- Default.svc.cluster.local
- Svc.cluster.local
- Cluster.local
- + suffixes DNS google internal

2) Connectez-vous dans un de vos pods puis exécutez le curl de votre service web sur les URLs et hostname suivants :

```
$ curl http://nom-service:81
$ curl http://nom-service.formation:81
$ curl http://nom-service.formation.svc:81
$ curl http://nom-service.formation.svc.cluster.local:81
```

Voici comment sont composés les suffixes DNS au sein d'un cluster :

- « nom-service » : représente le nom du service
- « formation» : représente le namespace dans lequel le service s'exécute
- « svc.cluster.local » : représente le nom de domaine utilisé par le cluster

En définitif, vous pouvez grâce aux entrées DNS interconnecter plus facilement vos différents services sans nécessiter de connaitre les IPs utilisées.

Note :

Au sein d'un même namespace, il n'est pas nécessaire d'utiliser les suffixes DNS. Ainsi, si vous souhaitez connecter un service front-end à son back-end, l'usage du simple « hostname » du service backend fonctionnera.

2. Accès externe aux services

Dans la première partie de cet atelier, vous avez appris à créer des services et à les faire communiquer ensemble. Nous allons maintenant aborder l'accès aux services depuis l'extérieur du cluster afin, par exemple, d'exposer un service web à vos utilisateurs ou clients.

Il existe 3 options principales pour exposer un service vers l'extérieur :

- **NodePort**
    - La création d'un service avec le type **NodePort** implique que chaque node ouvre un port directement sur son IP local et redirige le trafic reçu sur ce port vers le service indiqué. A noter que le service, dans cette configuration, reste accessible par l'IP attribuée « Service Cluster IP »
- **LoadBalancer**
    - La création d'un service avec le type **LoadBalancer** repose sur une extension du modèle **Nodeport**. Il s'agit ici de provisionner un load-balancer extérieur au cluster **Kubernetes** pour répartir le trafic entrant vers les ports « Nodeport » exposés. Le client se connecte donc à l'IP virtuelle du load-balancer.
- **Ingress**
    - Le type **Ingress** permet d'exposer plusieurs services sur une seule et unique IP externe. La ressource **Ingress** repose sur un mécanisme de reverse-proxy HTTP. Le trafic est donc géré au niveau 7 dans cette configuration.

2.1. Type NodePort

Dans cet exercice, nous allons exposer un service via le type **NodePort** en réservant un port sur chacune des nodes du cluster.

1) Pour cela, vous allez réutiliser votre service « service-generator » pour atteindre vos pods. Vous configurerez le port 30080 pour accéder au port http 80 de notre service web.

Note : si vous indiquez le type **Nodeport** sans préciser explicitement le port via « nodePort », **Kubernetes** attribuera automatiquement un port aléatoire à votre service.

2) Créez un nouveau manifeste « web-nodeport » en vous basant sur celui-ci et démarrez votre service et vos pods :

```yaml
kind: Service
apiVersion: v1
metadata:
  name: web-nodeport
spec:
  type: NodePort
  selector:
    app: generatorsvc
  ports:
  - port: 81
    targetPort: 80
    nodePort: 30080
```

3) Exécutez la commande ***« kubectl describe »*** pour analyser les spécifications du service :

   - Vérifiez que le Type est correctement configuré
   - Vérifiez que la propriété « nodePort » est paramétrée sur le bon port

4) Exécutez les curls suivants en vous connectant sur l'une des nodes du cluster :

```
$ curl http://IP_Service:81
$ curl http://IP_node:30080
$ curl http://IP_Service:30080
$ curl http://IP_node:81
```

   - Que remarquez-vous?

5) Pour paramétrer l'ouverture du port TCP/30080 sur chacune des nodes du cluster, exécutez la commande suivante :

```
gcloud compute firewall-rules create service-generator-rule1 --allow=tcp:30080
```

6) Une fois la règle activée, vérifiez le bon fonctionnement en exécutant les curls suivants depuis l'une des nodes :

```
$ curl http://IP_node1:30080
$ curl http://IP_node2:30080
$ curl http://IP_node3:30080
```

7) Ouvrez un navigateur sur votre poste de travail et ouvrez l'URL http://IP_node:30080

Votre service est désormais accessible depuis Internet sur les IPs publiques de chacune des nodes.

Dans cette situation, il est possible de joindre votre service depuis l'IP de n'importe laquelle des nodes du cluster. Néanmoins, si un client se connecte toujours sur la même node et que celle-ci rencontre un problème, il ne saura pas automatiquement se connecter sur l'IP d'une autre node. Pour cette raison, nous allons placer un load-balancer en frontal des IPs des différentes nodes dans la partie suivante.

2.2.Type Load-Balancer

En fonction de la distribution **Kubernetes** que vous utilisez, il est possible d'automatiser le provisionning d'un load-balancer dans votre cluster en utilisant le type **LoadBalancer**.

A noter que les clusters **Kubernetes** proposés par la majorité des cloud providers disposent nativement de ce niveau d'automatisation.

1) Créez un nouveau manifeste « web-loadbalancer » en vous basant sur celui-ci et démarrez votre service et vos pods :

```
kind: Service
apiVersion: v1
metadata:
  name: web-loadbalancer
spec:
  type: LoadBalancer
  selector:
    app: generatorsvc
  ports:
  - port: 81
    targetPort: 80
```

Note : dans le manifeste ci-dessus, nous n'avons pas créé de port **Nodeport** même si cela est possible. **Kubernetes** va alors automatiquement attribuer un port aléatoire.

2) Exécutez la commande ***« kubectl get svc »*** pour analyser le service :

- Analysez le champ « EXTERNAL_IP »

- Exécutez plusieurs fois la commande jusqu'à l'obtention d'une IP dans cette colonne pour votre service

3) Exécutez la commande ***« kubectl describe »*** pour analyser les spécifications du service
   - Vérifiez que le Type est correctement configuré
   - Vérifiez le port attribué automatiquement par **Kubernetes** dans le champ « nodePort »
   - Vérifiez les events
   - Vérifiez l'IP attribuée au **LoadBalancer** et notez là

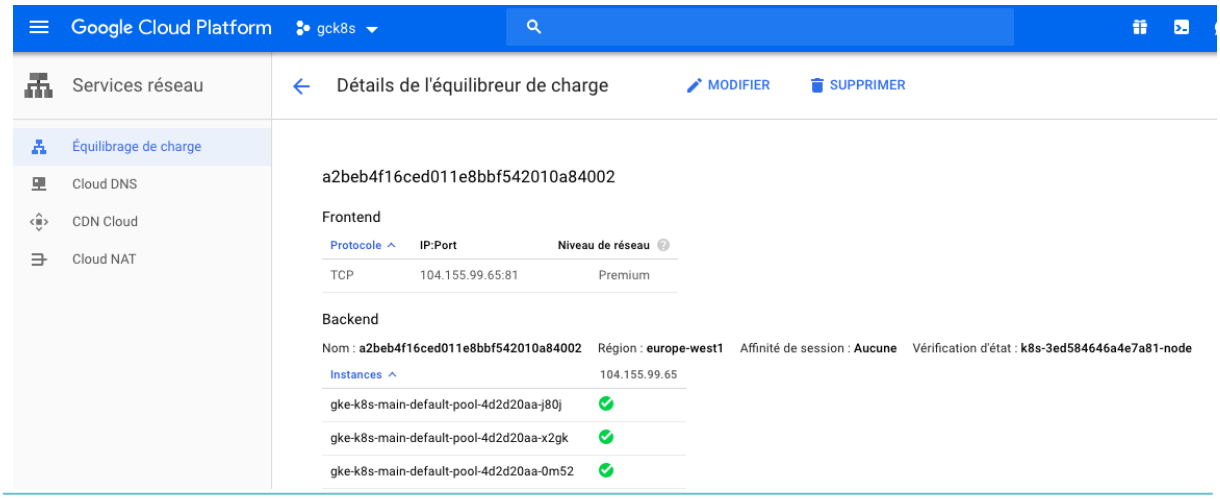
4) Exécutez les curls suivants en vous connectant sur l'une des nodes du cluster :

```
$ curl http://IP_Service:81
$ curl http://IP_node:
$ curl http://IP_EXTERNAL:81
```

5) Vous pouvez maintenant ouvrir un navigateur sur l'URL du load-balancer créée automatiquement sur GCP à l'adresse suivante http://EXTERNAL_IP:81.

Par rapport au type **NodePort**, nous notez que vous n'avez pas eu besoin d'ouvrir le port attribué au **NodePort** car la ressource **LoadBalancer** créée automatiquement par GCP a configuré l'ouverture de port, dans notre cas TCP/81, dès sa création.

Vérifiez cela depuis la console GCP comme suit :



6) Supprimez vos pods et votre service, puis faites un rafraichissement de la page ci-dessus.
   - Que constatez-vous ?

Maintenant que vous utilisez un load-balancer en frontal de votre service via l'utilisation des IP des nodes de votre cluster, il est important de comprendre que le comportement suivant peut se produire :

- Le loadbalancer GCP vous redirige vers une première node, par exemple node1
- Le service visé réparti alors aléatoirement le trafic vers l'un des pods de sa liste de endpoints
- Le pod finalement utilisé peut donc se trouver sur une autre node que le point d'entrée obtenu par le load-balancer frontal

Pour éviter ce fonctionnement en double-hop, il est possible de forcer le service à utiliser le pod présent localement sur la node attribuée par le loadbalancer GCP avec la spécification « externalTrafficPolicy » sur « local ».

7) Créez un nouveau manifeste « web-lb-local » en vous basant sur celui-ci et démarrez votre service et vos pods :

```yaml
kind: Service
apiVersion: v1
metadata:
  name: web-lb-local
spec:
  type: LoadBalancer
  externalTrafficPolicy: Local
  selector:
    app: generatorsvc
  ports:
  - port: 81
    targetPort: 80
```

Attention, le fonctionnement du paramètre « externalTrafficPolicy = Local » implique que vous devez vous assurez que chaque node du cluster géré par le load-balancer héberge un pod du service. Le cas échéant, la connexion n'aboutira pas.

De plus, si plusieurs pods se retrouve sur une seule node, ils accepteront moins de trafic qu'un pod unique sur une autre node.

En définitif, le paramètre « externalTrafficPolicy = Local » permet d'optimiser les communications réseaux afin d'éviter le double-hop, mais s'accompagne de certaines contraintes d'architecture de vos applications.

2.3.Ingress

La dernière option pour exposer ses services à l'extérieur du cluster se base sur la ressource **Ingress**. **L'Ingress** a pour principal avantage de ne nécessiter qu'une seule et unique IP externe pour adresser plusieurs services. Ce point est avantageux notamment lorsqu'on expose ses services sur Internet afin de limiter la prolifération d'IP publiques.

Le principe de fonctionnement se base sur un modèle reverse-proxy ; la redirection de trafic vers un service se base sur le nom d'hôte et le chemin de l'URL reçu par **l'Ingress**. Du fait que **l'Ingress** agit au niveau de la couche applicative, il est possible de configurer la persistance de session par cookie.

1) Créez le manifeste suivant pour créer la ressource **Ingress** et la rediriger vers le service précédent basé sur la configuration **NodePort**.

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-gen
spec:
  rules:
    - host: nom-du-service.ilkilab.io
      http:
        paths:
          - path: /*
            backend:
              serviceName: nom-du-service
              servicePort: 80
```

2) Exécutez le manifeste ci-dessus, puis créer le service et les pods associés.
   - Vérifier le bon fonctionnement des pods, du service

3) Vérifiez la bonne création de la ressource **Ingress** avec la commande suivante :

```
$ kubectl get ingresses
```

- Notez l'adresse IP publique attribuée à votre ressource **Ingress**

4) Vérifiez la bonne configuration de votre ressource **Ingress** sur le service souhaité avec la commande suivante :

```
kubectl get ingress
```

5) Ouvrez un navigateur depuis votre poste local et entrez l'adresse IP attribué à votre ressource **Ingress**.
   - Comment expliquer ce message d'erreur ?

6) Sur votre poste, modifiez le fichier host situé dans C:\Windows\System32\Drivers\etc en ajoutant l'entrée suivante :

```
IP-ressource-ingress     nom-de-service.ilkilab.io
```

7) Ouvrez à nouveau votre navigateur depuis votre poste local et entrez l'URL que vous avez renseignée dans votre ressource **Ingress** en finissant par « / ».

2.4.Ingress multi-services

Comme évoqué, la nature d'une ressource **Ingress** consiste à analyser l'URL demandée par le client pour rediriger le trafic vers le bon service.

Voici un premier exemple de manifeste présentant ce type de configuration basé sur le chemin de l'URL :

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-name
spec:
  rules:
  - host: nom-de-service.ilkilab.io
    http:
      paths:
      - path: /production
        backend :
          serviceName: service-production
          servicePort: 80
      - path: /test
        Backend :
          serviceName: service-test
          servicePort: 80
      - path: /recette
        backend:
          serviceName: service-recette
          servicePort: 80
```

Voici un second exemple de manifeste présentant ce type de configuration, mais basé sur des requêtes sur différents FQDN :

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-name
spec:
  rules:
  - host: service-production.ilkilab.io
    http:
      paths:
      - path: /*
        backend:
          serviceName: service-production
          servicePort: 80
  - host: service-test.ilkilab.io
    http:
      paths:
      - path: /*
        backend:
          serviceName: service-test
          servicePort: 80
  - host: service-recette.ilkilab.io
    http:
      paths:
      - path: /*
        backend:
          serviceName: service-recette
          servicePort: 80
```

En définitive, vous constatez que la ressource **Ingress** constitue un reverse-proxy / load-balancer de niveau 7.

2.5.Sécurisation de l'Ingress en TLS

Pour terminer ces manipulations autour des services et des accès externes, vous allez configurer un certificat TLS pour sécuriser les connexions à un service.

1) Tout d'abord, créez une paire de certificats clef publique / clef privée comme suit :

```
$ openssl genrsa -out tls.key 2048
$ openssl req -new -x509 -key tls.key -out tls.cert -days 360 -subj /CN=nom-de-service.ilkilab.io
```

- Vérifiez la présence des 2 fichiers tls.key et tls.cert dans votre dossier

2) Créer un **Secret** depuis ces 2 fichiers pour ensuite les utiliser dans votre ressource Ingress :

```
kubectl create secret tls tls-secret --cert=tls.cert --key=tls.key
```

Note : Vous en apprendrez plus sur les ressources de type **Secret** dans un prochain atelier

3) Vous pouvez maintenant modifier votre manifeste **Ingress** comme suit :

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingr-generator
spec:
  tls:
  - hosts:
    - nom-de-service.ilkilab.io
    secretName: tls-secret
  rules:
  - host: nom-de-service.ilkilab.io
    http:
      paths:
      - path: /*
        backend :
          serviceName: service-generator
          servicePort : 80
```

4) Déployez vos modifications en utilisant la commande ***« kubectl apply »*** afin d'éviter de supprimer et recréer à chaque fois votre ressource **Ingress**.

- Valider le bon fonctionnement de votre en HTTPS depuis un navigateur.