



None

TJO & MVT

Copyright © 2025 TomTalks

Table of contents

1. 🔍 Un cluster k8s aux petits oignons 🍷	4
1.1 C'est parti 🍷	4
1.2 Dernier check avant de démarrer	4
2. Créer son cluster avec Terraform	5
3. Un premier déploiement dans notre cluster	7
4. Contrôler nos Ingress	8
5. Gestion des URLs	11
5.1 Configuration de external-dns pour CloudFlare	11
5.2 Installation d'external-dns	11
5.3 External DNS en action	12
6. Sécurité avant tout	14
6.1 Installation	14
6.2 Configuration	14
6.3 Utilisation	15
7. Sécurisation des secrets	17
7.1 Installation	17
7.2 Configuration	17
7.3 Configuration d'un ExternalSecret	19
7.4 Configuration de external-dns pour utiliser les nouveaux secrets	20
8. Set up Kyverno	22
8.1 Installation	22
8.2 Ma première policy	23
8.3 Une 2ème policy	25
8.4 Suivi de nos policies	27
8.5 Pour aller plus loin	27
9. Repos du guerrier	29
9.1 Installation	29
9.2 Test	30
10. Notre recette d'un cluster kub aux petits oignons	32
10.1 La recette	32
10.2 Et ça sert à quoi !?	32
10.3 C'est parti 🍷	32
10.4 Avant de partir 🍷 (APRES avoir fait les parties GitLab env. et/ou Flux)	33
11. 🍷 Demos	34
11.1 Les GitLab environnements	34

11.2 GitOps tu connais ?	38
--------------------------	----

1. 🔍 Un cluster k8s aux petits oignons 🍷

Estimated time to read: 2 minutes

Bienvenue dans ce merveilleux cours de *cuisine* de Kubernetes.

L'objectif est de vous faire créer et paramétrer un cluster Kubernetes *from scratch* pour avoir des environnements de développements, de tests aux petits oignons pour vous et vos équipes/collègues.

1.1 C'est parti 🍷

Pour faire simple et sans polluer votre PC, ouvrir le workspace [Gitpod](#)

ou si vous préférez, vous pouvez cloner le repo en local dans votre répertoire préféré:

```
git clone https://gitlab.com/yodamad-workshops/2024/devvxx/kub-workshop.git
```

1.1.1 Prérequis 🛠️ (si vous faites en local)

Si vous avez choisi l'option [Gitpod](#), ils sont déjà installés 😊.

Pour ce workshop, vous aurez besoin des outils.

- git : [Installation](#)
- kubectl : [Installation](#)
- helm : [Installation](#)
- curl : [Installation](#)
- terraform : [Installation](#)

1.2 Dernier check avant de démarrer

Pour vérifier que tout est ok et initialiser les variables d'environnement qui vont bien, nous avons prévu un petit script (à faire aussi sur Gitpod)

Si tout se déroule comme prévu, vous devez avoir un résultat comme suit (au delta de la mise en forme suivant votre shell)

```
*****
* 🍷 Bienvenue à notre super workshop 🍷 *
* Quelques vérifications avant de commencer *
*****

🔧 Check local env
🍷 curl ... ✅
🍷 kubectl ... ✅
🍷 helm ... ✅
🍷 git ... ✅
🍷 terraform ... ✅

🛠️ Setup local env...
🍷 OVH connection setup ... ✅
🍷 Cloudflare setup ... ✅
🍷 GitLab setup ... ✅

*****
* 😊 All good !! *
* C'est parti, amusez vous bien 🍷 *
*****
```

🚀 Let's go ! Première étape : créer notre cluster ➡️

2. Créer son cluster avec Terraform

Estimated time to read: 2 minutes

D'abord, il faut initialiser l'environnement terraform

```
cd terraform # (1)
terraform init
```

1. On doit se mettre dans le répertoire `terraform` pour réaliser les étapes



Terraform initialisé correctement

La commande se termine par **Terraform has been successfully initialized!**

Il faut donner un nom à votre cluster:

```
export TF_VAR_OVH_CLOUD_PROJECT_KUBE_NAME=<votre_trigramme> # (1)
```

1. 🚀 Mettre un trigramme a minima, voire un chiffre également pour l'unicité des clusters

🚀 Attention à bien respecter les règles de nommage:

- Pas d'underscore, pas d'espace, pas de majuscule, pas d'accent, pas de caractères spéciaux
- RIEN !! que des minuscules et - 😊 (regex: `^[a-z0-9]([-a-z0-9]*[a-z0-9])?$`)



Variables nécessaires pour initialiser le cluster sur OVH

Ensuite, il faut configurer des variables d'environnements pour interagir avec notre cloud provider:

```
export TF_VAR_SERVICE_NAME=""
export TF_VAR_APPLICATION_KEY=""
export TF_VAR_APPLICATION_SECRET=""
export TF_VAR_CONSUMER_KEY=""
```

On est sympa, c'est déjà fait grâce au script exécuté au début du workshop. Si vous avez changer de terminal, il faut refaire la commande suivante:

D'autres variables décrites dans `variables.tf` peuvent être surchargées comme par exemple la taille des noeuds ou la localisation du cluster. Mais ...

👤 **Horacio González (@LostInBrittany)**

Le GRA : c'est la vie ! 🙌

donc par défaut nous utilisons le datacenter de *Gravelines*.



Pour les curieux `variables.tf`

On peut maintenant "planifier" notre déploiement:

```
terraform plan -out kub-workshop.plan
```



Terraform correctement planifié

Aucune erreur n'apparaît dans la console.

On peut désormais exécuter le déploiement

```
terraform apply kub-workshop.plan
```



À propos de `kub-workshop.plan`

On peut voir qu'un fichier `kub-workshop.plan` a été créé à la racine de notre repo. Il s'agit d'un fichier binaire contenant les informations nécessaires à Terraform pour interagir avec le provider.

L'exécution prend une dizaine de minutes, le temps de prendre un ☕ car après c'est parti !!



Cluster créé

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Maintenant, il faut récupérer notre fichier `kubeconfig` pour interagir avec notre nouveau cluster:

```
terraform output -raw kubeconfig > cluster-ovh-${TF_VAR_OVH_CLOUD_PROJECT_KUBE_NAME}.yaml
export KUBECONFIG=$(pwd)/cluster-ovh-${TF_VAR_OVH_CLOUD_PROJECT_KUBE_NAME}.yaml
```

Vérifier que la connexion est ok:

```
kubectl get nodes -o wide
```

Le résultat ressemble à: (1)

1. La version peut différer par rapport à l'exemple

NAME	STATUS	ROLES	AGE	VERSION	INTERNAL-IP	EXTERNAL-IP	OS-IMAGE	KERNEL-VERSION	CONTAINER-RUNTIME
mvt-snowcamp-pool-node-47dd14	Ready	<none>	20m	v1.27.8	57.128.56.219	<none>	Ubuntu 22.04.3 LTS	5.15.0-91-generic	containerd://1.6.25



Notre cluster est prêt, déployons notre première application ➡

3. Un premier déploiement dans notre cluster

Estimated time to read: 1 minute

Et si on essayait de déployer un *truc* dans notre cluster, par exemple un [cinéma en ascii](#) 🍿



Rebasculer à la racine du repo `cd ..`

```
kubectl apply -f demos/simple-deployment.yml
```



Pour les curieux, le contenu de `simple-deployment.yml`



Installation réussie

```
namespace/demos created
deployment.apps/simple-deployment created
service/simple-deployment-service created
```

On vérifie que tout est ok:

```
kubectl get deployments -n demos
```



Demo déployée

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
simple-deployment	1/1	1	1	39s

Maintenant, pour accéder à notre cinéma, il faut créer un *tunnel* vers notre cluster:

```
kubectl -n demos port-forward $(kubectl get pods -o=name -n demos) 8080:80
```

🍿 Notre application est disponible <http://localhost:8080>

Ok, ça marche mais c'est pas super pratique pour:

- travailler en équipe
- il faut faire le mapping des ports et des hosts pour chaque application
- ...

Pour accéder depuis l'extérieur, il va nous falloir un soupçon d'*Ingress* et un *Ingress Controller* pour gérer cela pour nous.



En route pour la découverte de comment Nginx controller va nous faciliter la vie ➡

4. Contrôler nos Ingress

Estimated time to read: 4 minutes

Pour nous aider à gérer nos Ingress, [ingress-nginx-controller](#) va nous aider pour automatiquement faire le routage depuis l'extérieur.

D'abord, il faut ajouter le repo Helm pour le chart:

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
helm repo update
```

Ensuite, on peut installer le chart: (1)

1. l'option `create-namespace` force la création d'un nouveau namespace

```
helm upgrade --install ingress-nginx ingress-nginx/ingress-nginx \
  --namespace nginx-ingress-controller \
  --create-namespace \
  --set controller.publishService.enabled=true
```

On peut vérifier que tout est installé correctement

```
helm ls -A
```

helm installé

NAME	NAMESPACE	REVISION	UPDATED	STATUS	CHART	APP VERSION
ingress-nginx	nginx-ingress-controller	1	2023-12-19 15:33:53.505518 +0100 CET	deployed	ingress-nginx-4.8.4	1.9.4 # (1)

1. La version 1.9.4 peut varier

On peut vérifier que le namespace et les composants sont bien créés :

```
kubectl get ns
```

Le namespace est créé

NAME	STATUS	AGE
default	Active	5h17m
kube-node-lease	Active	5h17m
kube-public	Active	5h17m
kube-system	Active	5h17m
nginx-ingress-controller	Active	10s

```
kubectl get all -n nginx-ingress-controller
```

Les composants sont instanciés et opérationnels

NAME	READY	STATUS	RESTARTS	AGE
pod/ingress-nginx-controller-75967d99c9-9vcwr	1/1	Running	0	2m45s

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/ingress-nginx-controller	LoadBalancer	10.3.28.249	57.128.120.49	80:31253/TCP,443:31582/TCP	2m46s
service/ingress-nginx-controller-admission	ClusterIP	10.3.10.52	<none>	443/TCP	2m46s

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/ingress-nginx-controller	1/1	1	1	2m46s

NAME	DESIRED	CURRENT	READY	AGE
replicaset.apps/ingress-nginx-controller-75967d99c9	1	1	1	2m46s



Cela peut prendre un peu de temps avant que l'IP externe soit disponible.

Une fois l'IP externe disponible, on peut facilement la récupérer:

```
export EXT_IP=$(kubectl get service ingress-nginx-controller -n nginx-ingress-controller -o jsonpath='{.status.loadBalancer.ingress[0].ip}')
echo ${EXT_IP:-NOT_SET_WAIT_AND_RETRY}
```

Désormais, notre `ingress-controller` va automatiquement gérer le routage pour nous lors de l'ajout d'`Ingress` dans le cluster.

Pour cela, il faut définir un `Ingress` avec le `Deployment` précisant l'URL sur laquelle exposer le service et spécifier dans l'attribut `host` l'URL sur laquelle on souhaite exposer notre deployment.



Exemple avec le premier déploiement

(pas besoin de le faire, on l'a préparé pour vous, cf. la suite 😊)

On pourrait ajouter cela à notre 1er déploiement, par exemple:

```
1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    name: simple-deployment-ingress
5    namespace: demos
6    annotations:
7      external-dns.alpha.kubernetes.io/cloudflare-proxied: "false" # (1)
8  spec:
9    ingressClassName: nginx
10   rules:
11     - host: simple-deployment.<votre_trigramme>.grunty.uk # (2)
12       http:
13         paths:
14           - backend:
15               service:
16                 name: simple-deployment-service
17                 port:
18                   number: 80
19             pathType: Prefix
20             path: /
21   ---
```

1. Spécificité lorsque l'on utilise cloudflare
2. URL sur laquelle on veut exposer le service

On va installer un nouveau `Deployment` avec son `Ingress` grâce au fichier `demos/deployment-with-ingress-grunty.yml`



Pensez à mettre votre trigramme

Pensez bien à modifier `demos/deployment-with-ingress-grunty.yml` avec votre trigramme avant de faire le `apply`

```
kubectl apply -f demos/deployment-with-ingress-grunty.yml
export my_host=new-deployment.<votre_trigramme>.grunty.uk
echo "Site URL http://"${my_host}
```



Pour les curieux, le contenu de `deployment-with-ingress-grunty.yml`

Demo déployée

Le pod est démarré :

```
kubectl get po -n demos
```

donne

NAME	READY	STATUS	RESTARTS	AGE
new-deployment-5998d8bcc-kdbrk	1/1	Running	0	9m21s
simple-deployment-5897799cb-94xzv	1/1	Running	0	58m

et l'ingress créé avec la bonne url:

```
kubectl get ingress -n demos
```

donne

NAME	CLASS	HOSTS	ADDRESS	PORTS	AGE
new-deployment-ingress	nginx	new-deployment.mvt.grunty.uk	57.128.120.31	80	9m32s

Grâce à l'ingress controller, nous n'avons plus besoin de faire le `port-forward` comme précédemment car le routage est fait sur l'URL. On peut donc tester d'accéder à nos pods via `curl`

2. méthodes possibles:

- En forçant un header `Host` avec l'URL voulue:

```
echo "Host: ${my_host}" "http://${EXT_IP}/"
curl --header "Host: ${my_host}" "http://${EXT_IP}/"
```

* ou, en surchargeant la résolution DNS dynamiquement:

```
curl --resolve ${my_host}:80:${EXT_IP} -H "Host: ${my_host}" -i "http://${EXT_IP}"
```

Les 2 commandes donnent le même résultat: (1)

1. La commande avec l'option `--resolve` est légèrement plus verbeuse

```
Server address: 10.2.1.6:80
Server name: new-deployment-5998d8bcc-kdbrk
Date: 19/Dec/2023:15:07:19 +0000
URI: /
Request ID: 4246cc4f7d0d56a49792cc73024dc779
```

On a changé l'image Docker entre temps

Ne vous étonnez pas si vous ne voyez plus le magnifique cinéma ASCII du départ.

On a changé l'image Docker pour que ce que l'on affiche soit lisible lorsque l'on fait un `cUrl`

Ok ça fonctionne, mais ça reste pas super pratique à utiliser 😞 car même si l'on a défini une URL dans notre manifest, celle-ci n'est pas référencée dans un DNS.

💡 Ca serait pratique, si automatiquement nos entrées DNS étaient créées et publiées pour que ce soit accessible depuis un navigateur comme tout autre site "classique".

Et si `external-dns` était la solution **!?**

 Alors ajoutons-en une pincée pour voir ➡

5. Gestion des URLs

Estimated time to read: 3 minutes

Avant d'installer `external-dns`, il est nécessaire de créer un `ServiceAccount` afin de lui donner accès aux différents namespaces pour détecter les nouveaux endpoints à gérer:

```
kubectl apply -f external-dns/external-dns-rbac.yml
```



ServiceAccount créé

```
namespace/external-dns created
serviceaccount/external-dns created
clusterrole.rbac.authorization.k8s.io/external-dns created
clusterrolebinding.rbac.authorization.k8s.io/external-dns-viewer created
```

5.1 Configuration de external-dns pour CloudFlare

Vous aurez besoin de 2 infos : la clé d'API et le user email référencé par Cloudflare. Ces 2 infos sont stockés dans des variables d'environnement `API_KEY` & `API_MAIL`



Elles sont déjà configurées

On est sympa, c'est déjà fait grâce au script exécuté au début du workshop. Si vous avez changer de terminal, il faut refaire la commande suivante:

5.1.1 Gestion du secret pour accéder à l'API Cloudflare

Il est nécessaire de créer un secret pour stocker l'API key d'accès à Cloudflare et un autre pour le compte de connexion

```
kubectl -n external-dns create secret generic cloudflare-api-token --from-literal=api-key=$API_KEY
kubectl -n external-dns create secret generic cloudflare-user-mail --from-literal=user-mail=$API_MAIL
```

On peut vérifier que les secrets sont bien créés et disponibles

```
kubectl get secrets -n external-dns
```



Secrets créés

NAME	TYPE	DATA	AGE
cloudflare-api-token	Opaque	1	15s
cloudflare-user-mail	Opaque	1	11s

5.2 Installation d'external-dns

Pour déployer `external-dns`, il suffit de créer un `Deployment` installant l'image officielle d' `external-dns`

```
kubectl apply -f external-dns/external-dns-cloudflare.yml
```

Pour comprendre le manifest:

1



Documentation officielle Cloudflare

La documentation officielle pour Cloudflare est [dispo](#)



external-dns is Running

```
kubectl get po -n external-dns
```

NAME	READY	STATUS	RESTARTS	AGE
external-dns-cloudflare-694f6f75-nf8n8	1/1	Running	0	2s



Vérifier les logs

On peut aussi vérifier que la connexion à Cloudflare est bien ok

```
kubectl logs $(kubectl get po -n external-dns | grep external-dns-cloudflare | cut -d' ' -f1) -n external-dns
```

```
...
time="2023-12-20T08:52:41Z" level=info msg="Instantiating new Kubernetes client"
time="2023-12-20T08:52:41Z" level=info msg="Using inCluster-config based on serviceaccount-token"
time="2023-12-20T08:52:41Z" level=info msg="Created Kubernetes client https://10.3.0.1:443"
...
```

5.3 External DNS en action

On peut voir dans les logs qu'external-dns a déjà automatiquement détecté notre Ingress précédent:

```
kubectl logs -f $(kubectl get po -n external-dns | grep external-dns-cloudflare | cut -d' ' -f1) -n external-dns
```

```
...
time="2023-12-20T08:52:44Z" level=info msg="Changing record." action=CREATE record=new-deployment.mvt.grunty.uk ttl=1 type=A zone=be73d3e4c087b970da9bb670130a11fc
time="2023-12-20T08:52:45Z" level=info msg="Changing record." action=CREATE record=new-deployment.mvt.grunty.uk ttl=1 type=TXT zone=be73d3e4c087b970da9bb670130a11fc
time="2023-12-20T08:52:45Z" level=info msg="Changing record." action=CREATE record=a-new-deployment.mvt.grunty.uk ttl=1 type=TXT zone=be73d3e4c087b970da9bb670130a11fc
...
```

On peut vérifier que notre nom de domaine est bien reconnu

```
dig new-deployment.$TF_VAR_OVH_CLOUD_PROJECT_KUBE_NAME.grunty.uk @ara.ns.cloudflare.com
```



DNS est configuré avec notre IP publique

```
;; ANSWER SECTION:
new-deployment.<votre_trigramme>.grunty.uk. 300 IN A 57.128.120.31 (1)
```

1. 📶 l'IP sera différente



On a changé l'image Docker entre temps

Ne vous étonnez pas si vous ne voyez plus le magnifique cinéma ASCII du départ.

On a changé l'image Docker pour que ce que l'on affiche soit lisible lorsque l'on fait un cUrl

On peut aussi valider que le navigateur reconnait notre URL en visitant http://new-deployment.<votre_trigramme>.grunty.uk/

ou via un cURL comme à l'étape précédente

```
curl new-deployment.<votre_trigramme>.grunty.uk
```

```
Server address: 10.2.1.6:80
Server name: new-deployment-5998d8dbcc-kdbrk
Date: 20/Dec/2023:09:45:13 +0000
URI: /
Request ID: 82d85003846eed6c62744103d7ac2bda
```

C'est beaucoup plus pratique !! 🥳

Mais ... ce n'est pas très sécurisé le HTTP, le chef de brigade de la sécurité nous rappelle à l'ordre 🙄

Si on ajoute quelques mL de sécurité avec des certificats pour notre HTTPS ➡

6. Sécurité avant tout

Estimated time to read: 4 minutes

Nous allons gérer les certificats avec [cert-manager](#)

6.1 Installation

`cert-manager` fournit un Helm chart, configurons le repo:

```
helm repo add jetstack https://charts.jetstack.io
helm repo update
```

Et maintenant nous pouvons installer `cert-manager` dans notre cluster dans un namespace dédié

```
helm install \
  cert-manager jetstack/cert-manager \
  --namespace cert-manager \
  --create-namespace \
  --version v1.13.3 \
  --set installCRDs=true
```



cert-manager est installé

```
cert-manager v1.13.3 has been deployed successfully! # (1)
```

1. La version peut être différente

Maintenant, vérifions que tout est ok:

```
kubectl get all -n cert-manager
```

NAME	READY	STATUS	RESTARTS	AGE
pod/cert-manager-6856dc897b-k6dks	1/1	Running	0	32s
pod/cert-manager-cainjector-c86f8699-cmc7t	1/1	Running	0	32s
pod/cert-manager-webhook-f8f64cb85-7rjjz	1/1	Running	0	32s

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/cert-manager	ClusterIP	10.3.242.74	<none>	9402/TCP	33s
service/cert-manager-webhook	ClusterIP	10.3.182.243	<none>	443/TCP	33s

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/cert-manager	1/1	1	1	33s
deployment.apps/cert-manager-cainjector	1/1	1	1	33s
deployment.apps/cert-manager-webhook	1/1	1	1	33s

NAME	DESIRED	CURRENT	READY	AGE
replicaset.apps/cert-manager-6856dc897b	1	1	1	33s
replicaset.apps/cert-manager-cainjector-c86f8699	1	1	1	33s
replicaset.apps/cert-manager-webhook-f8f64cb85	1	1	1	33s




6.2 Configuration

`cert-manager` permet de générer des certificats pour sécuriser nos endpoints. Il supporte plusieurs providers, ici nous allons utiliser le classique mais pratique [Let's Encrypt](#).

Pour cela, nous devons créer un `Issuer`, ici un `ClusterIssuer` pour accéder à l'ensemble du cluster

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-production
  namespace: cert-manager
spec:
  acme:
    server: https://acme-v02.api.letsencrypt.org/directory
    email: <your-email> # (1)
```

```
privateKeySecretRef:
  name: letsencrypt-production # (2)
solvers:
- http01:
    ingress:
      class: nginx # (3)
```

1.  Un email est nécessaire pour l'appartenance du domaine
2.  On utilise le serveur de production
3.  On précise que l'on répondra au challenge HTTP en utilisant un Ingress de type `nginx`

Editer le fichier `cert-manager/letsencrypt-cluster-issuer.yml` pour positionner un email valide.

Créons notre issuer:

```
kubectl apply -n cert-manager -f cert-manager/letsencrypt-cluster-issuer.yml
```

Issuer créé

```
clusterissuer.cert-manager.io/letsencrypt-production created
```

Utilisation du serveur de staging

Par défaut, on utilise le serveur de production de Let's Encrypt, mais celui à la limitation que si vous avez un nombre de requêtes en échec trop importants, vous pouvez être banni temporairement.

Pour une phase de test, il est possible d'utiliser le serveur de staging de Let's Encrypt qui n'a pas cette limitation.

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-staging
  namespace: cert-manager
spec:
  acme:
    server: https://acme-staging-v02.api.letsencrypt.org/directory
    email: <your-email>
    privateKeySecretRef:
      name: letsencrypt-staging
  solvers:
  - http01:
      ingress:
        class: nginx
```

6.3 Utilisation

Maintenant, on peut sécuriser nos URLs. Il suffit d'updater notre Ingress pour lui configurer le TLS.

Pour cela, il faut mettre à jour le fichier `demo/demo-with-ingress-https-grunty.yml` avec votre trigramme.

Déployons ensuite notre nouveau composant:

```
kubectl apply -f demo/demo-with-ingress-https-grunty.yml
```

Pour les curieux, le contenu du manifest `deployment-with-ingress-https-grunty.yml`

Les lignes surlignées configurent le TLS

Deployment installé

```
ingress.networking.k8s.io/secured-deployment-ingress created
```

On peut voir que `cert-manager` instancie un `Ingress` pour gérer les échanges avec *Let's Encrypt* pour la génération du certificat:

```
kubectl get ingress -n demos
```

NAME	CLASS	HOSTS	ADDRESS	PORTS	AGE
cm-acme-http-solver-hmh8t	<none>	secured.<votre_trigramme>.grunty.uk		80	47s
new-deployment-ingress	nginx	new-deployment.<votre_trigramme>.grunty.uk	57.128.120.31	80	19h
secured-deployment-ingress	nginx	secured.<votre_trigramme>.grunty.uk		80, 443	50s

Une fois les challenges terminés avec *Let's Encrypt*, on voit aussi qu'un `Certificate` a été généré. Il est au statut `Ready` à `False`, au bout de quelques secondes, il devrait passer à `True`

```
kubectl get certificates -n demos
```

Certificat généré

NAME	READY	SECRET	AGE
secured.<votre_trigramme>.grunty.uk-tls	True	secured.<votre_trigramme>.grunty.uk-tls	3m44s

Behind the scene

On peut facilement voir les étapes en observant les events

```
kubectl get events --sort-by='.lastTimestamp' -n demos
```

Tout semble ok, on peut vérifier que notre URL est désormais sécurisée : https://secured.{votre_trigramme}.grunty.uk

Site accessible en HTTPS

Mais notre chef de brigade est vraiment pointilleux 😬, notre gestion des données sensibles tels que nos API keys ne lui convient pas.

Alors pimentons un peu tout cela avec une gestion de secrets plus propre ➡

7. Sécurisation des secrets

Estimated time to read: 5 minutes

Pour sécuriser nos secrets, nous allons utiliser `external-secrets` pour simplifier la délégation de la gestion des secrets à un outil tiers tel que *GitLab*, *HashiCorp Vault*...

7.1 Installation

`external-secrets` fournit un Helm chart pour une installation simple

On ajoute donc le repo pour `external-secrets` :

```
helm repo add external-secrets https://charts.external-secrets.io
helm repo update
```

et on installe dans un namespace dédié 🚀

```
helm install external-secrets external-secrets/external-secrets -n external-secrets --create-namespace --set installCRDs=true
```



external-secrets est installé

external-secrets has been deployed successfully!

`external-secrets` instancie plusieurs composants, on peut les lister:

```
kubectl get all -n external-secrets
```

NAME	READY	STATUS	RESTARTS	AGE
pod/external-secrets-5f45b6f844-27fmq	1/1	Running	0	2m9s
pod/external-secrets-cert-controller-9795887f6-8mssr	1/1	Running	0	2m9s
pod/external-secrets-webhook-6f4789ccf-qmpkn	1/1	Running	0	2m9s

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/external-secrets-webhook	ClusterIP	10.3.175.44	<none>	443/TCP	2m11s

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/external-secrets	1/1	1	1	2m10s
deployment.apps/external-secrets-cert-controller	1/1	1	1	2m10s
deployment.apps/external-secrets-webhook	1/1	1	1	2m10s

NAME	DESIRED	CURRENT	READY	AGE
replicaset.apps/external-secrets-5f45b6f844	1	1	1	2m9s
replicaset.apps/external-secrets-cert-controller-9795887f6	1	1	1	2m9s
replicaset.apps/external-secrets-webhook-6f4789ccf	1	1	1	2m9s

Tout est prêt !

7.2 Configuration







Pour permettre à `external-secrets` de gérer les secrets, il est nécessaire de créer un `SecretStore`. Celui-ci fera le lien entre le cluster et l'outil hébergeant les secrets.

Pour faire simple dans ce workshop, on va faire avec *GitLab* pour stocker nos données sensibles via l'utilisation des variables de CI/CD. C'est simple à mettre en oeuvre mais introduit une faiblesse coté sécurité car nécessite quand même d'avoir un secret coté Kubernetes, pour stocker la clé d'API pour se connecter à *GitLab*.

**D'autres solutions sont possibles...**

`external-secrets` fournit plusieurs [connecteurs](#) possibles. Le plus complet semble être celui d'HashiCorp Vault car il n'introduit pas de besoin de clé ou de credentials pour s'intégrer dans le cluster.

On a déjà préparé les variables dans le projet GitLab (via le menu `Settings -> CI/CD -> Variables`):

CI/CD Variables </> 2		
↑ Key	Value	Environments
CF_API_KEY  Masked Expanded	***** 	All (default) 
CF_USER_MAIL  Masked Expanded	***** 	All (default) 

Créons le secret de connexion:

```
kubectrl -n external-secrets create secret generic gitlab-token --from-literal=token=$GITLAB_TOKEN
```

**GITLAB_TOKEN est déjà configuré pour vous**

On est sympa, c'est déjà fait grâce au script exécuté au début du workshop. Si vous avez changer de terminal, il faut refaire la commande suivante:

Maintenant on peut créer un `ClusterSecretStore` pour faire le lien entre GitLab et le cluster.

```
apiVersion: external-secrets.io/v1beta1
kind: ClusterSecretStore
metadata:
  name: gitlab-cluster-secret-store
  namespace: external-secrets
spec:
  provider:
    # provider type: gitlab
    gitlab: # (1)
      url: https://gitlab.com/ # (2)
      auth:
        SecretRef:
          accessToken:
            name: gitlab-token
            key: token
            namespace: external-secrets
          projectID: "53147568" # (3)
```


1. 🧐 Le type du provider
2. 🔗 L'URL de votre GitLab, pas forcément gitlab.com
3. 📦 L'ID du projet qui héberge les données sensibles

**Alternative SecretStore**

Il est possible d'utiliser un `SecretStore` pour restreindre à un namespace, si vous voulez limiter l'accessibilité/utilisation de vos secrets.


On peut déployer notre `ClusterSecretStore` :

```
kubectl apply -f external-secrets/external-secrets-secret-store.yml -n external-secrets
```

 Pour les curieux, le fichier `external-secrets-secret-store.yml`

et vérifier que le connexion à GitLab est opérationnelle:

```
kubectl get clustersecretstores.external-secrets.io -n external-secrets
```

 **ClusterSecretStore est valide**

NAME	AGE	STATUS	CAPABILITIES	READY
gitlab-cluster-secret-store	2m50s	Valid	ReadOnly	True


7.3 Configuration d'un ExternalSecret

Nos secrets stockés dans GitLab, il faut créer désormais créer un `ExternalSecret` qui va rappatrier dans notre cluster le secret stocké dans GitLab.

On va faire le test sur `external-dns` pour externaliser notre clé d'API et username Cloudflare dans GitLab:

```
kubectl apply -n external-dns -f external-secrets/external-secrets-external-secret.yml
```


 Pour les curieux, le fichier `external-secrets-external-secret.yml`

 **Les secrets sont créés**

```
externalsecret.external-secrets.io/external-secret-cloudflare-key-credentials created
externalsecret.external-secrets.io/external-secret-cloudflare-mail-credentials created
```

On peut vérifier que nos secrets sont valides (ie *synchronisés à l'état* `SecretSynced`):

```
kubectl get externalsecrets.external-secrets.io -n external-dns
```

 **Success**

NAME	STORE	REFRESH INTERVAL	STATUS	READY
external-secret-cloudflare-key-credentials	gitlab-cluster-secret-store	1h	SecretSynced	True
external-secret-cloudflare-mail-credentials	gitlab-cluster-secret-store	1h	SecretSynced	True

Automatiquement `external-secrets` a généré des secrets pour nous:

```
kubectl get secrets -n external-dns
```

**Les nouveaux secrets sont générés**

NAME	TYPE	DATA	AGE
cloudflare-api-token	Opaque	1	4h53m
cloudflare-user-mail	Opaque	1	15h
external-cloudflare-api-token	Opaque	1	99s
external-cloudflare-user-mail	Opaque	1	98s

7.4 Configuration de external-dns pour utiliser les nouveaux secrets

Maintenant que l'on a nos nouveaux secrets, on peut mettre à jour notre `Deployment` d'external-dns pour qu'il les utilise plutôt que les anciens.

On change donc la référence des secrets dans le `Deployment` :

```
- name: CF_API_TOKEN
  valueFrom:
    secretKeyRef:
      name: external-cloudflare-api-token
      key: api-key
- name: CF_API_EMAIL
  valueFrom:
    secretKeyRef:
      name: external-cloudflare-user-mail
      key: user-mail
```

On aura plus besoin de nos anciens secrets, autant les supprimer:

```
kubectl delete secret cloudflare-api-token cloudflare-user-mail -n external-dns
```

Et on met à jour external-dns pour utiliser les nouveaux

```
kubectl apply -f external-secrets/external-dns-cloudflare.yml -n external-dns
```

**Pour les curieux, le fichier external-dns-cloudflare.yml**

Après quelques secondes, on voit qu'un nouveau pod a été créé et fonctionne:

```
kubectl get po -n external-dns
```

**external-dns est recréé et opérationnel**

NAME	READY	STATUS	RESTARTS	AGE
external-dns-XXX-7d4cf677f6-nmwn5	1/1	Running	0	25s

et on vérifie que la connexion avec Cloudflare est toujours ok:

```
kubectl logs -f $(kubectl get pods -l "app.kubernetes.io/name=external-dns" -n external-dns | grep external-dns-cloudflare | cut -d ' ' -f1) -n external-dns
```

**external-dns est bien connecté à Cloudflare**

```
time="2023-12-20T13:48:56Z" level=info msg="Instantiating new Kubernetes client"
time="2023-12-20T13:48:56Z" level=info msg="Using inCluster-config based on serviceaccount-token"
time="2023-12-20T13:48:56Z" level=info msg="Created Kubernetes client https://10.3.0.1:443"
```

Le chef de brigade sécurité est content, mais il sent qu'on a du répondant 🤪, il nous impose alors encore de nouvelles contraintes : il ne veut que des ingrédients contrôlés en amont et veut favoriser la filière locale plutôt que la grande distribution... 🛒

Ok, challenge accepted !! 💪 Montrons-lui comment faire ➡

8. Set up Kyverno

Estimated time to read: 8 minutes

Pour contrôler ce qu'il se passe dans notre cluster, nous allons utiliser [kyverno](#).

8.1 Installation

Kyverno fournit un Helm chart pour simplifier l'installation.

```
helm repo add kyverno https://kyverno.github.io/kyverno/
helm repo update
```

Une fois configuré, on peut installer le Helm chart:

```
helm install kyverno kyverno/kyverno -n kyverno --create-namespace
```



Thank you for installing kyverno! Your release is named kyverno.

The following components have been installed in your cluster:

- CRDs
- Admission controller
- Reports controller
- Cleanup controller
- Background controller

On vérifie que tout est ok coté namespace `kyverno`

```
kubectl get all -n kyverno
```

Les composants installés par kyverno

NAME	READY	STATUS	RESTARTS	AGE
pod/kyverno-admission-controller-547894dbc9-srg54	1/1	Running	0	2m43s
pod/kyverno-background-controller-696f7765d-d8knf	1/1	Running	0	2m43s
pod/kyverno-cleanup-controller-567bb6695c-mxf7s	1/1	Running	0	2m43s
pod/kyverno-reports-controller-5799587486-x2gjp	1/1	Running	0	2m43s

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/kyverno-background-controller-metrics	ClusterIP	10.3.236.41	<none>	8000/TCP	2m45s
service/kyverno-cleanup-controller	ClusterIP	10.3.230.129	<none>	443/TCP	2m45s
service/kyverno-cleanup-controller-metrics	ClusterIP	10.3.189.211	<none>	8000/TCP	2m45s
service/kyverno-reports-controller-metrics	ClusterIP	10.3.153.42	<none>	8000/TCP	2m45s
service/kyverno-svc	ClusterIP	10.3.165.95	<none>	443/TCP	2m45s
service/kyverno-svc-metrics	ClusterIP	10.3.106.166	<none>	8000/TCP	2m45s

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/kyverno-admission-controller	1/1	1	1	2m44s
deployment.apps/kyverno-background-controller	1/1	1	1	2m44s
deployment.apps/kyverno-cleanup-controller	1/1	1	1	2m44s
deployment.apps/kyverno-reports-controller	1/1	1	1	2m44s

NAME	DESIRED	CURRENT	READY	AGE
replicaset.apps/kyverno-admission-controller-547894dbc9	1	1	1	2m44s
replicaset.apps/kyverno-background-controller-696f7765d	1	1	1	2m44s
replicaset.apps/kyverno-cleanup-controller-567bb6695c	1	1	1	2m44s
replicaset.apps/kyverno-reports-controller-5799587486	1	1	1	2m44s

NAME	SCHEDULE	SUSPEND	ACTIVE	LAST SCHEDULE	AGE
cronjob.batch/kyverno-cleanup-admission-reports	*/* * * * *	False	0	<none>	2m44s
cronjob.batch/kyverno-cleanup-cluster-admission-reports	*/* * * * *	False	0	<none>	2m44s

8.2 Ma première policy

Comme nous l'a demandé le chef de brigade, on veut contrôler nos ingrédients : mettons en place une policy pour limiter l'usage d'images Docker provenant uniquement d'une registry privée en laquelle on a confiance, dans notre demo, le registry de gitlab.com de notre projet.

Définissons notre policy `ClusterPolicy` qui sera à l'échelle du cluster complet (les **+** vous donnent des indications pour mieux comprendre)

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  annotations:
    policies.kyverno.io/title: Restrict Image Registries
    policies.kyverno.io/description: >~
      Only images from specific gitlab.com registry are allowed.
spec:
  validationFailureAction: Enforce # (1)
  background: true
  rules:
    - name: validate-registries
      match:
        any: # (2)
        - resources:
            kinds:
              - Pod
      exclude: # (3)
        any:
          - resources:
              namespaces:
                - kube-system
                - external-dns
                - external-secrets
                - nginx-ingress-controller
                - kyverno
                - cert-manager
      validate: # (4)
        message: "Unauthorized registry."
        pattern:
          spec: # (5)
            =(ephemeralContainers):
              - image: "registry.gitlab.com/*"
            =(initContainers):
              - image: "registry.gitlab.com/*"
            containers:
              - image: "registry.gitlab.com/*"
```

1. 🗨️ On utilise une action de type `Enforce` qui est bloquante. Il existe aussi le type `Audit` qui est un warning mais non bloquant
2. 🔍 On applique à tous les Pod
3. 🚫 On filtre les namespaces que l'on a déjà instancié et qui sont des namespaces d'administration
4. 🧠 Le type de règle, ici `validate`
5. 📝 On spécifie pour chaque type, les origines autorisées

On peut déployer notre policy:

```
kubectl apply -f kyverno/kyverno-registry-policy.yml
```

et on vérifie qu'elle est bien opérationnelle

```
kubectl get clusterpolicies.kyverno.io -n kyverno
```

Policy au statut Ready

NAME	ADMISSION	BACKGROUND	VALIDATE	ACTION	READY	AGE	MESSAGE
restrict-image-registry	true	true	Enforce		True	18m	Ready

8.2.1 Vérification

Essayons de démarrer un pod avec une image issue de Docker hub:

```
kubectl run demo-nginx --image=nginx:latest -n demos
```

Impossible de démarrer le pod

```
Error from server: admission webhook "validate.kyverno.svc-fail" denied the request:

resource Pod/demos/demo-nginx was blocked due to the following policies

restrict-image-registry:
  validate-registries: 'validation error: Unauthorized registry. rule validate-registries
    failed at path /spec/containers/0/image/'
```

On met à jour notre `Deployment` pour utiliser une image issue de la registry GitLab:

```
apiVersion: apps/v1
kind: Deployment
# ...
spec:
  selector:
    matchLabels:
      app: nginx-hardened
  template:
    metadata:
      labels:
        app: nginx-hardened
    spec:
      containers:
        - image: registry.gitlab.com/yodamad-workshops/kub-workshop/asciinema:latest # (1)
          name: asciinema-hardened
          ports:
            - containerPort: 80
      imagePullSecrets:
        - name: gitlabcred # (2)
```

1. 🐳 On utilise une image de notre registry privée
2. 🗝️ C'est une registry privée, il faut s'authentifier...

On voit qu'il faut ajout un secret pour être capable de `pull` une image depuis une registry privée, normal... Mais du coup, il faut créer un secret pour cela. Alors on va en créer un

```
kubectl create secret gitlabcred regcred --docker-server=<your-registry-server> --docker-username=<your-name> --docker-password=<your-pword> --docker-email=<your-email>
```

Le chef de la brigade de la sécurité

Pas de secret en dur dans mon cluster !! 🚫

Oups, il a pas tort 😊 On devrait plutôt utiliser `external-secrets`

On crée un `ExternalSecret`

```
---
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
  name: external-secret-gitlabcred
  namespace: kyverno
spec:
  refreshInterval: 1m
  secretStoreRef:
    name: gitlab-cluster-secret-store
    kind: ClusterSecretStore
  target:
    name: gitlabcred
    creationPolicy: Owner
  template:
    engineVersion: v2
    type: kubernetes.io/dockerconfigjson # (1)
    data: # (2)
      .dockerconfigjson: "{\"auths\":{\"registry.gitlab.com\":{\"username\":\"{{ .username }}\",\"password\":\"{{ .password }}\",\"auth\":\"{{(printf \"%s:%s\" .username .password) | b64enc }}\"}}}"
    data: # (3)
      - secretKey: username
        remoteRef:
          key: gl_cr_username
      - secretKey: password
```



```
remoteRef:
  key: gl_cr_password
```

1. 🐙 On définit le type de template que l'on utilise pour créer le secret
2. 🐳 On crée le dockerconfigson à partir de 2 variables
3. 🦊 On récupère le username & le mot de passe depuis GitLab

que l'on déploie

```
kubectl apply -f kyverno/kyverno-gitlabcred-external-secret.yml
```

✔ Secret correctement créé

L'ExternalSecret est créé et synchronisé

```
kubectl get externalsecrets.external-secrets.io -n kyverno
```

NAME	STORE	REFRESH INTERVAL	STATUS	READY
external-secret-gitlabcred	gitlab-cluster-secret-store	1m	SecretSynced	True

et le secret aussi

```
kubectl describe secret gitlabcred -n kyverno
```

```
Name:      gitlabcred
Namespace: kyverno
Labels:    reconcile.external-secrets.io/created-by=25390d6b8b839ba8a1d72cfcfe6f6319
Annotations: reconcile.external-secrets.io/data-hash: 9dfdd28d70dbacf25d05ab5d782ae9a5

Type: kubernetes.io/dockerconfigjson

Data
====
.dockerconfigjson: 154 bytes
```

On doit pouvoir déployer notre nouvelle image

```
kubectl apply -f kyverno/kyverno-asciinemantic-hardened.yml
```

📄 Pour les curieux, le fichier kyverno-asciinemantic-hardened.yml

✔ Image déployée

```
kubectl get po -n demos
```

NAME	READY	STATUS	RESTARTS	AGE
asciinemantic-hardened-95b5f9d76-b87j9	1/1	Running	0	64s

8.3 Une 2ème policy

Du fait que l'on a forcé sur l'ensemble du cluster que les images proviennent de GitLab, il serait pratique qu'automatiquement lorsque l'on crée un namespace, automatiquement le secret pour se connecter à GitLab via `external-secrets` se crée.

On a utilisé un type `validate` lors de notre première policy, dans ce second cas, on va utiliser le type `generate` qui va automatiquement générer des éléments.

```
# ...
policies.kyverno.io/description: >-
  Secrets like registry credentials often need to exist in multiple
  Namespaces so Pods there have access. Manually duplicating those Secrets
```

```

is time consuming and error prone. This policy will copy a
Secret called 'gitlabcred' which exists in the 'kyverno' Namespace to
new Namespaces when they are created. It will also push updates to
the copied Secrets should the source Secret be changed.
spec:
  rules:
  - name: sync-image-pull-secret
    # ...
    generate: # (1)
      apiVersion: v1
      kind: Secret
      name: regcred
      namespace: "{{request.object.metadata.name}}"
      synchronize: true
    clone: # (2)
      namespace: kyverno
      name: gitlabcred

```

1. 🧐 Type `generate`
2. 🧐 Action de clone d'un objet existant

Déployons cette nouvelle policy

```
kubectl apply -f kyverno/kyverno-sync-regcred.yml
```

 Pour les curieux, le fichier `kyverno-sync-regcred.yml`

On vérifie que la policy est créée et opérationnelle

```
kubectl get clusterpolicies.kyverno.io -n kyverno
```

✅ **Policy créée et opérationnelle**

NAME	ADMISSION	BACKGROUND	VALIDATE	ACTION	READY	AGE	MESSAGE
restrict-image-registry	true	true	Enforce		True	66m	Ready # (1)
sync-secrets	true	true	Audit		True	26s	Ready

1. 🚚 On retrouve bien notre première policy

8.3.1 Vérification

Vérifions que notre nouvelle policy fonctionne bien en créant un nouveau namespace

```
kubectl create ns demo-kyverno
```

Normalement, on devrait avoir un secret dans notre nouveau namespace

```
kubectl get secret -n demo-kyverno
```

✅ **Secret créé**

NAME	TYPE	DATA	AGE
regcred	kubernetes.io/dockerconfigjson	1	112s

Voyons si l'on essaie de déployer une image provenant de notre registry privée dans notre nouveau namespace

```
kubectl run demo-nginx --image=registry.gitlab.com/yodamad-workshops/kub-workshop/nginx:hardened -n demo-kyverno
```

Image déployée

```
kubectl get po -n demo-kyverno
```

NAME	READY	STATUS	RESTARTS	AGE
demo-nginx	1/1	Running	0	15s

Et vérifions que notre première policy est bien valable dans notre nouveau namespace

```
kubectl run demo-nginx --image=nginx:latest -n demo-kyverno
```

Impossible de déployer

Error from server: admission webhook "validate.kyverno.svc-fail" denied the request:

resource Pod/demo-kyverno/demo-nginx was blocked due to the following policies

restrict-image-registry:
 validate-registries: 'validation error: Unauthorized registry. rule validate-registries failed at path /spec/containers/0/image/'

8.4 Suivi de nos policies

Kyverno permet de facilement voir les policies qui ont été exécutées

```
kubectl get policyreports -n demo-kyverno -o wide
```

La policy a été exécutée avec succès

NAME	KIND	NAME	PASS	FAIL	WARN	ERROR	SKIP	AGE
3e3ad989-01be-46fc-afc9-7eea0f78a74c	Pod	demo-nginx	1	0	0	0	0	5m9s

On peut aussi vérifier à l'échelle du cluster

```
kubectl get policyreports -A
```

L'ensemble des reports du cluster

NAMESPACE	NAME	PASS	FAIL	WARN	ERROR	SKIP	AGE
demo-kyverno	3e3ad989-01be-46fc-afc9-7eea0f78a74c	1	0	0	0	0	14m
demoss	4649657f-4994-47eb-a15b-a3e2b14e82db	0	1	0	0	0	88m
demoss	718a97de-06af-4df8-b7d0-17f7ca9b5d02	1	0	0	0	0	34m
demoss	81d369ad-0b5a-46ba-85b6-0e73e7572afe	0	1	0	0	0	88m

On voit que les policies se sont exécutées sur les composants installés avant la policy et que certains sont aussi status **FAIL**

8.5 Pour aller plus loin

Kyverno propose d'autres types de [policies](#).

Article sur le sujet #autopromo

Un article décrivant plus en détails cela est dispo sur le [blog](#)

Notre cocotte est sécurisée et avec des bons produits issus de la filière locale 😊

Il est temps de se reposer un peu ➡

9. Repos du guerrier

Estimated time to read: 3 minutes

Vu qu'on a bien travaillé, on peut se reposer un peu et nos pods aussi, on va donc mettre en place [kube-downscaler](#) pour éteindre nos workloads lors des temps de pause.

9.1 Installation

`kube-downscaler` est un projet plus récent, il n'est pas encore très industrialisé donc il est nécessaire de faire quelques adaptations pour le déployer:

- changer le namespace `default` en `kube-downscaler` dans `deploy/rbac.yml`
- désactiver le mode `dry-run` dans `deploy/deployment.yml`
- (pour la démo) réduire la grace period de prise en compte des composants à 0min (plutôt que 15 par défaut) dans `deploy/deployment.yml`
- (pour la démo) appliquer uniquement les politiques sur le namespace `demos` dans `deploy/deployment.yml`

```
--grace-period=0
--namespace=demos
```

 Pour les curieux, les fichiers `rbac.yml` et `deployment.yml`

`rbac.yml` :

`deployment.yml` :

Les fichiers sont dans le répertoire `kube-downscaler/deploy`, on utilise [kustomize](#) pour déployer (contrairement à Helm les autres fois).

```
kubectrl apply -k kube-downscaler/deploy/ # (1)
```

1. On utilise `-k` au lieu du `-f` habituel

 Tout est créé

```
namespace/kube-downscaler created
serviceaccount/kube-downscaler created
clusterrole.rbac.authorization.k8s.io/kube-downscaler created
clusterrolebinding.rbac.authorization.k8s.io/kube-downscaler created
configmap/kube-downscaler created
deployment.apps/kube-downscaler created
```

On vérifie que tout est ok

```
kubectrl get all -n kube-downscaler
```

**Tout est opérationnel**

NAME	READY	STATUS	RESTARTS	AGE
pod/kube-downscaler-f5cbb6cfc-f2j9w	1/1	Running	0	39s

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/kube-downscaler	1/1	1	1	39s

NAME	DESIRED	CURRENT	READY	AGE
replicaset.apps/kube-downscaler-f5cbb6cfc	1	1	1	39s

9.2 Test

Vu qu'on aime pas trop bosser, on va configurer `kube-downscaler` pour que nos pods ne soient démarrés que le lundi de 7h30 à 18h 😊

```
annotations:
  downscaler/uptime: Mon-Mon 07:30-18:00 CET
  downscaler/force-downtime: "true"
```

Pour cela, on annote notre namespace `demons` pour qu'il ne soit up que sur cette période.

NB: `kube-downscaler` peut être configuré à différents niveaux, on aurait pu annoter les deployments directement plutôt que le namespace.

```
kubectl annotate ns demons 'downscaler/uptime=Mon-Mon 07:30-18:00 CET'
```

**Le namespace est annoté pour s'éteindre**

```
kubectl describe ns demons
```

```
Name:      demons
Labels:    kubernetes.io/metadata.name=demons
Annotations: downscaler/uptime: Mon-Fri 21:30-23:30 CET
Status:    Active
```

No resource quota.

No LimitRange resource.

On peut voir immédiatement que nos pods sont éteints et que nos déploiements sont à 0

**Tout est éteint**

```
kubectl get po -n demons
```

No resources found in demons namespace

et les déploiements

```
kubectl get deploy -n demons
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
asciinemetic-hardened	0/0	0	0	156m
new-deployment	0/0	0	0	28h
simple-deployment	0/0	0	0	28h

On peut voir dans les logs de `kube-downscaler` qu'il a traité notre configuration

```
kubectl logs -f $(kubectl get po -n kube-downscaler | grep kube-downscaler | cut -d' ' -f1) -n kube-downscaler
```



```
2023-12-20 18:28:55,501 INFO: Downscaler v23.2.0 started with debug=False, default_downtime=never, default_uptime=Mon-Fri 07:30-20:30 CET, deployment_time_annotation=None, downscale_period=never,
downtime_replicas=0, dry_run=False, enable_events=False, exclude_deployments=kube-downscaler,downscaler, exclude_namespaces=kube-system, grace_period=0, include_resources=deployments, interval=2,
namespace=demos, once=False, upscale_period=never
2023-12-20 18:47:29,649 INFO: Scaling down Deployment demos/asciinema-hardened from 1 to 0 replicas (uptime: Mon-Fri 21:30-23:30 CET, downtime: never)
2023-12-20 18:47:29,675 INFO: Scaling down Deployment demos/new-deployment from 1 to 0 replicas (uptime: Mon-Fri 21:30-23:30 CET, downtime: never)
2023-12-20 18:47:29,700 INFO: Scaling down Deployment demos/simple-deployment from 1 to 0 replicas (uptime: Mon-Fri 21:30-23:30 CET, downtime: never)
```

Voilà, on a une recette bien complète, faisons un petit récapitulatif ➡

10. Notre recette d'un cluster kub aux petits oignons

Estimated time to read: 2 minutes

10.1 La recette

Pour avoir un bon cluster:

- on l'instancie avec [terraform](#)
- on gère l'accès à nos endpoints depuis l'extérieur avec un **ingress-controller** tel que [nginx](#)
- pour que ça soit lisible, on crée automatique des entrées DNS pour chaque nouveau endpoint avec [external-dns](#)
- on sécurise tout ça avec des jolis certificats *Let's Encrypt* et à l'aide de [cert-manager](#)
- on externalise nos secrets dans GitLab avec [external-secrets](#)
- on contrôle un peu tout ce qu'il se passe dans la cocotte avec [kyverno](#)
- une fois qu'on a bien travaillé, on autorise tout le monde à se reposer avec [kube-downscaler](#)

Miam 😊

10.2 Et ça sert à quoi !?

Dans la vraie vie, ça nous sert à quoi.

Un premier exemple via l'utilisation des [environnements GitLab](#) pour simplement visualiser les différentes environnements d'un projet, quelle version/commit est déployé sur chacun, et faciliter leur listing et leur accès.

Aussi, on peut voir qu'avec une approche GitOps et [Flux](#) par exemple, on peut facilement automatiser l'installation et la configuration de tout cela et de simplement mettre à jour tous nos clusters.

Essayons tout cela...


10.3 C'est parti 🍷

Pour ces 2 exemples, il est nécessaire de **forker** le projet car vous aurez besoin d'être **Owner** du projet pour faire les manipulations.

Les 2 exercices sont indépendants, vous pouvez les faire dans l'ordre que vous souhaitez.

Pour cela, il suffit de cliquer sur [Fork](#) sur la page du projet et réaliser le fork de ce projet dans votre espace personnel.

yodamad-workshops / kub-workshop-snowcamp-2024 / Fork project



Fork project

A fork is a copy of a project. Forking a repository allows you to make changes without affecting the original project.

Project name

Must start with a lowercase or uppercase letter, digit, emoji, or underscore. Can also contain dots, pluses, dashes, or spaces.

Project URL

Project slug

Want to organize several dependent projects under the same namespace? [Create a group](#)

Project description (optional)

Branches to include

☐ All branches

☒ Only the default branch main

Visibility level ⓘ

☒ **Private**
Project access must be granted explicitly to each user. If this project is part of a group, access will be granted to members of the group.

☐ **Internal**
The project can be accessed by any logged in user.

☐ **Public**
The project can be accessed without any authentication.

Il en reste plus qu'à cloner votre fork en local !

- Pour la recette des environnements Gitlab ➡.
- Pour la recette avec FluxCD ➡.

10.4 Avant de partir 🍷 (APRES avoir fait les parties GitLab env. et/ou Flux)

Pensez à faire du ménage 🧹

- 1 Supprimer les ingress **avant** de supprimer le cluster (pour purger les records DNS):

```
for i in $(kubectl get ingress -A --no-headers -o=name); do
  kubectl delete $i -n demos
done
```

- 2 Supprimer le cluster:

```
cd terraform
terraform destroy
```

11. 🏠 Demos

11.1 Les GitLab environnements

Estimated time to read: 5 minutes

11.1.1 Pré-requis

Avant de démarrer cette partie, il est nécessaire de **forker** le projet car vous aurez besoin d'être **Owner** du projet pour faire les manipulations. Pour cela, il suffit de [forker le projet](#).

Pas besoin de forker 2 fois

Si vous avez fait la partie [flux](#) avant celle-ci et que vous avez déjà forké le projet, pas besoin de faire un nouveau fork. Vous pouvez réutiliser le 1er.

Gitpod

Si vous utilisez Gitpod n'oubliez pas de télécharger votre `kubeconfig` alias `cluster-ovh-${TF_VAR_OVH_CLOUD_PROJECT_KUBE_NAME}.yaml`. Avec le fork vous allez démarrer avec un nouveau pod et par conséquent vos fichiers locaux ne seront plus accessibles.

11.1.2 Installation

Pour installer l'agent, il faut suivre la [doc GitLab](#) qui consiste à :



- ajouter le repo Helm pour GitLab

```
helm repo add gitlab https://charts.gitlab.io
helm repo update
```

- Et installer l'agent

Un exemple mais à récupérer depuis l'IHM GitLab à cause du token

```
helm upgrade --install demo-gitlab-env gitlab/gitlab-agent \
--namespace gitlab-agent-demo-gitlab-env \
--create-namespace \
--set image.tag=v16.8.0-rc2 \ # (1)
--set config.token=glagent-v... \ # (2)
--set config.kasAddress=wss://kas.gitlab.com
```

1.  La version peut être différente
2.  Ce token est généré par GitLab au moment d'enregistrer l'agent.

L'agent est installé correctement

```
Thank you for installing gitlab-agent.

Your release is named demo-gitlab-env.
```

On vérifie que tout est ok

```
kubectl get po -n <namespace> # (1)
```

1. Namespace can be different



Les pods gitlab-agent sont démarrés

NAME	READY	STATUS	RESTARTS	AGE
demo-gitlab-env-gitlab-agent-v1-8b8bc9c85-2tq2b	1/1	Running	0	2m48s
demo-gitlab-env-gitlab-agent-v1-8b8bc9c85-w8wm7	1/1	Running	0	2m48s

11.1.3 Utilisation

Grâce à l'agent on va pouvoir interagir depuis GitLab-CI sans avoir besoin d'un `kubeconfig`

Par exemple, on peut déployer le helm custom disponible dans le répertoire `demo-gitlab`

On crée un job dans le fichier `.gitlab-ci.yml` à la racine du projet.



Supprimer le job existant

Vous devez supprimer le contenu du `.gitlab-ci.yml` existant avant d'ajouter les éléments

```
stages:
  - 🛠️ # (1)

🚢_deploy_env:
  stage: 🛠️
  image:
    name: dtzar/helm-kubectl
  script:
    - kubectl config use-context yodamad-workshops/2024/devoxx/kube-workshop:demo-gitlab-env # (2)
    - helm upgrade ${CI_COMMIT_REF_SLUG}-env ./demo-gitlab/ --set labName=${CI_PROJECT_NAME}-${CI_COMMIT_REF_SLUG} --install --create-namespace -n ${CI_PROJECT_NAME}-${CI_COMMIT_REF_SLUG}
  environment:
    name: ${CI_COMMIT_REF_SLUG}
    url : https://${CI_PROJECT_NAME}-${CI_COMMIT_REF_SLUG}.<votre_trigramme>.grunty.uk # (3)
  needs: []
```

1. 🚫 Ne pas oublier d'ajouter le stage du job dans la liste des stages 2. 🚢 Ne pas oublier de changer le path 3. ¹₂³₄ Ne pas oublier de mettre votre trigramme



Comprendre l'exemple

Dans cet exemple, on a plusieurs choses:

- `kubectl config use-context` pour connecter la CI au cluster
- `yodamad-workshops/2024/devoxx/kube-workshop` correspond au path vers votre projet forké sur gitlab.com. A adapter en fonction de vos données
- `demo-gitlab-env` correspond au nom du projet. A adapter en fonction de vos données
- `environment` : décrit l'environnement déployé en lien avec ce job
- `url` : une URL dynamique qui est construit en fonction du nom du projet et de la branche
- `on_stop` : l'action à déclencher quand l'environnement est arrêté, par exemple quand la branche associée est supprimée



Quel context utiliser

Si vous peinez à trouver quel est le bon nom du context à utiliser pour la partie `kubectl config use-context`, vous pouvez ajouter dans le job l'instruction suivante : `kubectl config get-contexts` qui vous listera les contextes disponibles.

```
🚢_deploy_env:
  [...]
  script:
    - kubectl config get-contexts
```

Grâce à ce que l'on a mis en place précédemment, on pourra simplement avoir un nouvel environnement avec une URL reconnue (merci `external-dns`), sécurisée (merci `cert-manager`) pour que les devs et les testeurs du projet puissent accéder à la version souhaitée.

Mettre votre trigramme dans l'ingress

Dans le fichier `demo-gitlab/values.yaml`, il faut changer `<votre_trigramme>` par votre trigramme et commiter

On peut commiter le fichier `.gitlab-ci.yml`, et une fois que le pipeline est terminé (avec succès), on peut voir que tout est bien créé

```
kubectl get po -n gitlab-agent-demo-gitlab-env
```

Tout est déployé

NAME	READY	STATUS	RESTARTS	AGE
pod/demo-gitlab-env-gitlab-agent-v1-8b8bc9c85-2tq2b	1/1	Running	0	11h
pod/demo-gitlab-env-gitlab-agent-v1-8b8bc9c85-w8wm7	1/1	Running	0	11h

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/demo-gitlab-env-gitlab-agent-v1	2/2	2	2	11h

NAME	DESIRED	CURRENT	READY	AGE
replicaset.apps/demo-gitlab-env-gitlab-agent-v1-8b8bc9c85	2	2	2	11h

Pour continuer, créer une *nouvelle branche* et changer l'image par défaut dans `demo-gitlab/values.yaml` :

Avant:

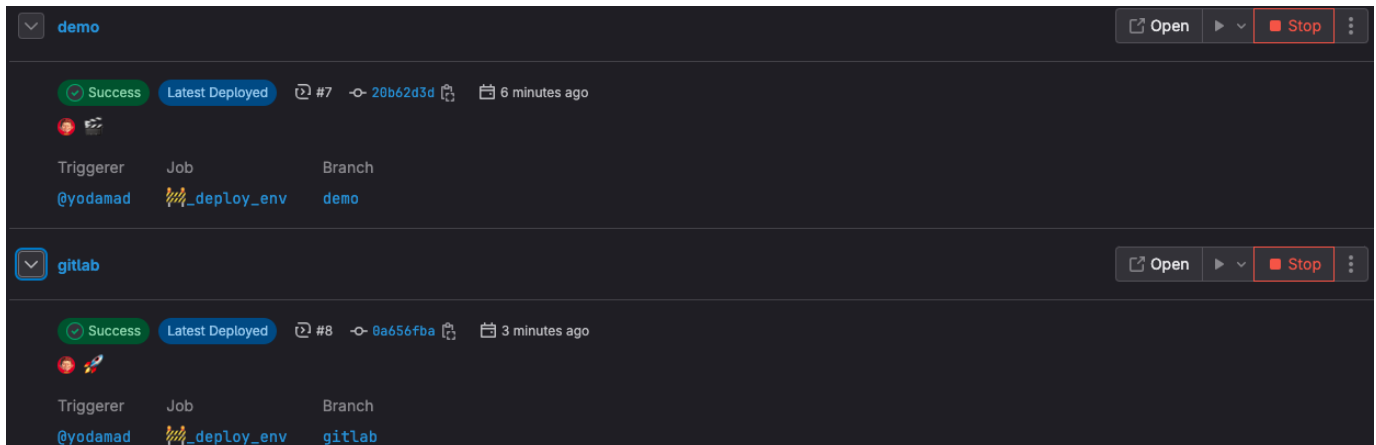
```
image: registry.gitlab.com/yodamad-workshops/kub-workshop/nginxdemos
version: plain-text
```

Après:

```
image: registry.gitlab.com/yodamad-workshops/kub-workshop/asciinema
version: latest
```

Commitez sur la *nouvelle branche* et attendez que le pipeline se termine.

Dans le menu `Operate > Environments`, il y a 2 environnements disponibles:



The screenshot shows the GitLab Operate interface for environments. At the top, there's a dropdown menu set to 'demo'. Below it, the 'demo' environment is shown with a green 'Success' status, 'Latest Deployed' badge, and a deployment ID of '20b62d3d' from 6 minutes ago. It lists the triggerer as '@yodamad', the job as '_deploy_env', and the branch as 'demo'. Below this, the 'gitlab' environment is shown with a green 'Success' status, 'Latest Deployed' badge, and a deployment ID of '0a656fba' from 3 minutes ago. It lists the triggerer as '@yodamad', the job as '_deploy_env', and the branch as 'gitlab'. Both environment cards have 'Open', 'Stop', and a menu icon at the top right.

En ouvrant les 2 environnements (via `Open`), on a bien :

- une IHM simple sur `main`
- un cinema ascii sur la nouvelle branche

11.1.4 Nettoyage

Il ne faut pas oublier de nettoyer ses environnements lorsqu'ils ne sont plus nécessaires (🌱 la planète vous dira merci).

Pour cela, on implémente le job qui est référencé dans le `on_stop`

```
stages:
  [...]
  - 🛠️
  - 🧹 # (1)

🧹_clean:
  stage: 🧹
  image:
    name: dtzar/helm-kubectl
  script:
    - kubectl config use-context yodamad-workshops/kub-workshop:demo-gitlab-env
    - helm uninstall ${CI_COMMIT_REF_SLUG}-env -n ${CI_PROJECT_NAME}-${CI_COMMIT_REF_SLUG}
    - kubectl delete ns ${CI_PROJECT_NAME}-${CI_COMMIT_REF_SLUG}
  when: manual
  environment:
    name: ${CI_COMMIT_REF_SLUG}
    action: stop
```

Il faut également ajouter l'instruction `on_stop` au job de déploiement :

```
🚀_deploy_env:
  [...]
  environment:
    [...]
  on_stop: 🧹_clean
```

1. ☢️ Ne pas oublier d'ajouter le stage du job dans la liste des stages

Arrêter l'environnement depuis l'IHM GitLab (via le menu `Operate > Environments`)

✅ **L'environnement est bien supprimé**

- Le job `on_stop` est bien déclenché

🟢 - 🟡

Stage: 🧹

🟡
🧹_clean
🚫

- Dans `Operate > Environments`, il n'y a plus qu'un environnement
- Dans le cluster, les éléments ont bien été supprimés

```
kubectl get ns kub-workshop
```

```
Error from server (NotFound): namespaces "kub-workshop-snowcamp-2024-demo" not found
```

Un premier usage plutôt pratique !

Retournons à la recette pour encore plus de découvertes ➡️

11.2 GitOps tu connais ?

Estimated time to read: 9 minutes

Ok, on a réussi notre petite recette. Tout est écrit mais ça serait encore mieux si on pouvait automatiser tout ça en utilisant [Flux](#) par exemple.

11.2.1 Pré-requis

Pour bootstraper Flux (en gros l'installer dans votre cluster), il faut que la personne qui lance la commande ait les droits **cluster admin** sur le cluster Kubernetes cible. Il est aussi nécessaire que la personne qui lance la commande soit le propriétaire du projet GitLab, ou ait les droits admin d'un groupe GitLab.

N'oubliez pas de [forker le projet](#) pour pouvoir le modifier **ET** de basculer sur votre nouveau repo 🤪



Si vous avez fait la partie [environnements gitlab](#) avant celle-ci et que vous avez déjà forké le projet, pas besoin de faire un nouveau fork. Vous pouvez réutiliser le 1er.



Si vous utilisez Gitpod n'oubliez pas de télécharger votre `kubeconfig` alias `cluster-ovh-${TF_VAR_OVH_CLOUD_PROJECT_KUBE_NAME}.yaml`. Avec le fork vous allez démarrer avec un nouveau pod et par conséquent vos fichiers locaux ne seront plus accessibles.

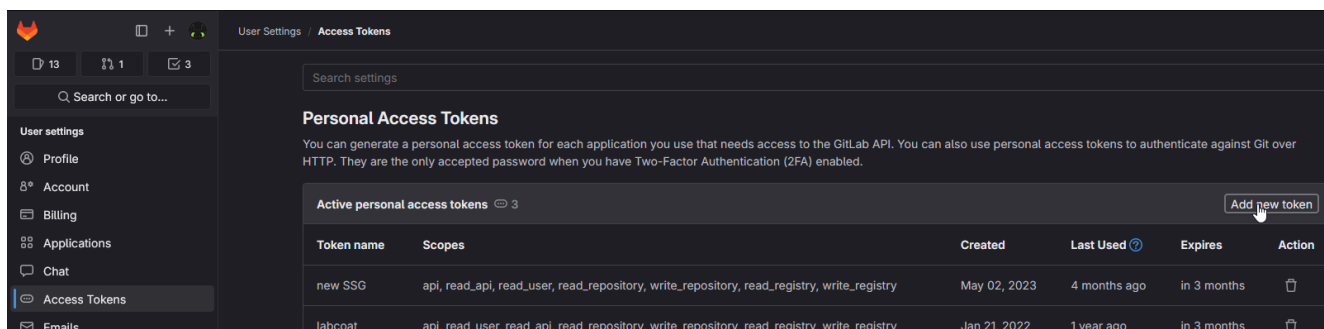
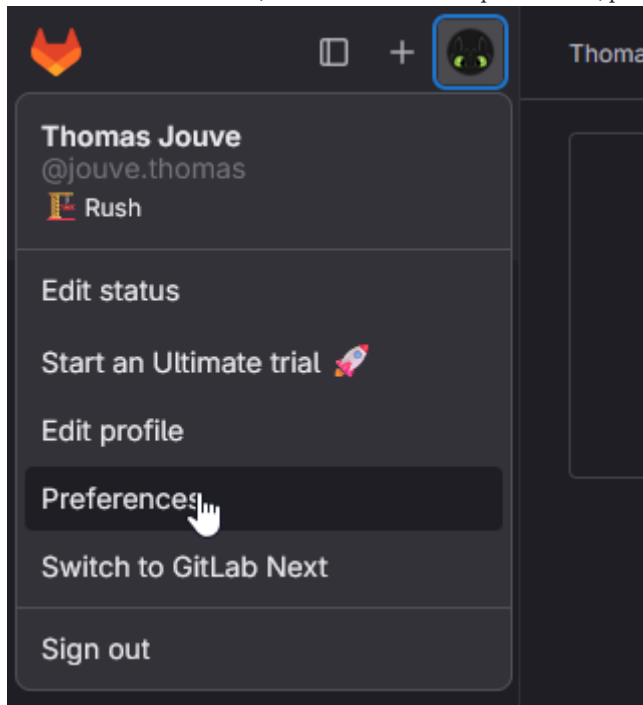
Ensuite, il va nous falloir un token GitLab pour que Flux puisse se connecter à notre repo GitLab.

GitLab PAT (Personal Access Token)

On va récupérer le token et le stocker dans une variable d'environnement.

Comment récupérer un token GitLab

Pour créer un token GitLab, il faut aller dans votre profil GitLab, puis dans `Preferences > Access Tokens`



Et créer un token avec les bons scopes:

```
export GITLAB_TOKEN=<THE TOKEN>
```

11.2.2 Installation de Flux

La première étape est d'installer Flux CLI.

```
curl -s https://fluxcd.io/install.sh | sudo bash
```

11.2.3 Cette fois c'est la bonne : on configure Flux

On va demander à la CLI d'initialiser Flux sur notre cluster et de se connecter à notre repo GitLab.

Tout est décrit [dans le tuto Flux](#).



Si la variable d'environnement `GITLAB_TOKEN` n'est pas renseignée, le bootstrap va demander de saisir le token.

Il est possible de fournir le token avec une commande du type: `echo "<gl-token>" | flux bootstrap gitlab`.

On lance le bootstrap sur le projet avec notre compte personnel:

```
flux bootstrap gitlab \
  --deploy-token-auth \
  --owner=<NAMESPACE_NAME> \
  --repository=<PROJECT_STUG> \
  --branch=main \
  --path=flux/devoxx-cluster/ \
  --personal
```


Il y a trois paramètres à remplacer :

- **NAMESPACE_NAME** : le groupe ou sous-groupe dans lequel vous voulez initialiser Flux
- **PROJECT_STUG** : le project dans lequel Flux va stocker les informations dont il a besoin
- **path** : le chemin où sont stocké les fichiers de configuration de Flux dans le repo

NAMESPACE_NAME + **PROJECT_STUG** doivent correspondre au chemin du repository dans lequel vous avez forké ce workshop.

Lorsque l'on utilise `--deploy-token-auth`, la CLI génère un token GL et le stock dans le cluster sous la forme d'une `Secret` qui s'appelle **flux-system** dans le Namespace **flux-system**.

Flux bootstrap output

```

▶ connecting to https://gitlab.com
▶ cloning branch "main" from Git repository "https://gitlab.com/yodamad-workshops/kub-workshop.git"
✓ cloned repository
▶ generating component manifests
✓ generated component manifests
✓ committed sync manifests to "main" ("4271d8d7adef5572f1031f0f21767d449d0ccbb4")
▶ pushing component manifests to "https://gitlab.com/yodamad-workshops/kub-workshop.git"
▶ installing components in "flux-system" namespace
✓ installed components
✓ reconciled components
▶ checking to reconcile deploy token for source secret
✓ configured deploy token "flux-system-main-flux-system-./flux/devoxx-cluster" for "https://gitlab.com/yodamad-workshops/kub-workshop"
▶ determining if source secret "flux-system/flux-system" exists
▶ generating source secret
▶ applying source secret "flux-system/flux-system"
✓ reconciled source secret
▶ generating sync manifests
✓ generated sync manifests
✓ committed sync manifests to "main" ("05dfd597d5959fda3a783f2336b65d1f1d7b121d")
▶ pushing sync manifests to "https://gitlab.com/yodamad-workshops/kub-workshop.git"
▶ applying sync manifests
✓ reconciled sync configuration
© waiting for Kustomization "flux-system/flux-system" to be reconciled
✓ Kustomization reconciled successfully
▶ confirming components are healthy
✓ helm-controller: deployment ready
✓ kustomize-controller: deployment ready
✓ notification-controller: deployment ready
✓ source-controller: deployment ready
✓ all components are healthy

```

Bien joué !

L'agent Flux va maintenant surveiller notre repo GitLab et appliquer les changements automatiquement.

11.2.4 Petite visite de notre nouvelle cuisine

Le cellier

Si on pull le repo GitLab, on peut voir que Flux a créé un nouveau répertoire `flux/devoxx-cluster/flux-system` qui contient la configuration de Flux :

- `kustomization.yaml` est un index, on va lister ici les manifests qui doivent être pris en compte dans ce répertoire.
- `gotk-components.yaml` contient la définition des RBAC et des CRDs (Custom Resource Definition) utilisées par Flux.
- `gotk-sync.yaml` définit la manière dont l'opérateur se connecte au repo au travers du **Kind** `GitRepository` et le type **Kustomization** permet de configurer quels sont les manifest / configuration à scruter. Pour nous, tout ce qui se trouve dans `./flux/devoxx-cluster` sera utilisé comme configuration.

Ici le fichier `./flux/devoxx-cluster/flux-system/gotk-sync.yaml` contient la configuration de Flux pour se connecter à notre repo GitLab.

```

1  apiVersion: source.toolkit.fluxcd.io/v1
2  kind: GitRepository
3  metadata:
4    # ICI on trouve le nom de notre Objet GitRepository
5    name: flux-system
6    namespace: flux-system
7  spec:
8    interval: 1m0s
9    ref:
10   # ICI la branche à utiliser
11   branch: main
12   secretRef:
13     name: flux-system
14   # ICI vous retrouverez l'adresse de votre repo GitLab
15   url: https://gitlab.com/jouve.thomas/kub-workshop-snowcamp-2024.git

```

Notre première recette

On va pouvoir lui dire d'appliquer la configuration grace aux objets **Kustomization**.

Si on regarde par exemple le fichier `flux/repo.yaml`

```

1

```

On peut voir que l'on décrit en langage **Flux** un nouveau répertoire à surveiller `./flux/repository` dans notre `GitRepository:flux-system`.

DÉFINITION DES HELMREPOSITORY

Dans ce répertoire `./flux/repository` on retrouve par exemple le fichier `nginx.yaml` :

```

1

```

Cette fois-ci on configure un **HelmRepository** qui va permettre à **Flux** de récupérer les informations sur les charts disponibles dans le repo **Helm**.

On pourra faire référence à ce chart sous le nom `nginx-ingress-controller`.

Nouveaux commits

On va maintenant déplacer ce fichier `repo.yaml` dans le répertoire `flux/devoxx-cluster/` qui est le seul, pour le moment, que connaît **Flux**.

Effectivement le fichier `gotk-sync.yaml` indique que seul le répertoire `./flux/devoxx-cluster` est scrupé par **Flux** :

```

1  apiVersion: kustomize.toolkit.fluxcd.io/v1
2  kind: Kustomization
3  metadata:
4    name: flux-system
5    namespace: flux-system
6  spec:
7    interval: 10m0s
8    path: ./flux/devoxx-cluster
9    prune: true
10   sourceRef:
11     kind: GitRepository
12     name: flux-system

```

11.2.5 On envoie les commandes en cuisine

Et on `push commit`, car maintenant c'est **Flux** qui se charge de faire la synchronisation sur le cluster depuis notre repo.

```

cp flux/repo.yaml flux/devoxx-cluster/repo.yaml
git add flux/devoxx-cluster/repo.yaml
git commit -am ":satellite_orbital: Setup Helm repos" && git push

```

On peut observer la réconciliation avec la commande suivante :

```
flux get kustomizations --watch
```

On a quelque chose comme ça :

- il y a 2 répertoires à surveiller (2 `Kustomization`)
- la synchronisation est active (`SUSPENDED` : `False`)
- et à jour (`READY` : `True`)

Il indique aussi quelle est la révision utilisée pour la synchronisation (ici : `main@sha1:c80d7d4c`).

NAME	REVISION	SUSPENDED	READY	MESSAGE
flux-system	main@sha1:c80d7d4c	False	True	Applied revision: main@sha1:c80d7d4c
repos	main@sha1:c80d7d4c	False	True	Applied revision: main@sha1:c80d7d4c

On peut vérifier en regardant si il a bien créé nos ressources `HelmRepository` :



```
kubectl get HelmRepository -A
```

NAMESPACE	NAME	URL	AGE	READY	STATUS
flux-system	cert-manager	https://charts.jetstack.io	112s	True	stored artifact: revision
'sha256:c930db5052b76d7be3026686612fa09f89a23f8547a8ecad7496d788e34964e5'					
flux-system	external-secrets	https://charts.external-secrets.io	112s	True	stored artifact: revision
'sha256:35fa1d6332232e3c6d032627547ffc74c7e61c4729ed1daa680b2202c61a78da'					
flux-system	nginx-ingress-controller	https://kubernetes.github.io/ingress-nginx	112s	True	stored artifact: revision
'sha256:e6a6c9e8f3682deea82b3bc22506d4fdabd667ce37cb1d0f7509459ca92c3426'					

Ingress Controller

Tout est prêt dans le répertoire `./nginx-ingress-controller/flux` pour déployer notre Ingress Controller.

LE DESCRIPTIF DE NOTRE RECETTE EST UNE `KUSTOMIZATION`

 Ici ce n'est pas une `Kustomization` mais une `Kustomization` 

Il faut lire :

- `Kustomization@kustomize.toolkit.fluxcd.io/v1` est la `CRD de Flux` pour les objets Flux (le liens vers un repo / repertoire / interval de scrapping)
- `Kustomization@kustomize.config.k8s.io/v1beta1` est l'objet `Kustomize` de Kubernetes

Ici on déclare une recette avec le nom `flux-nginx-ingress-controller` et qu'il est nécessaire d'utiliser les fichiers `nginx-ingress-controller.yaml` et `ns.yaml` pour déployer notre Ingress Controller.

```
1
```

LA DEFINITION DE NOTRE NAMESPACE

Avec un simple fichier manifest vanilla :


```
1
```

ET LA DEFINITION DE NOTRE HELMRELEASE

```
1
```

CHAUD DEVANT !

Avant de lancer la commande en cuisine, soyons fou et supprimons notre Ingress Controller précédemment installé pour laisser faire **Flux**.

 Ce n'est pas obligatoire, mais c'est pour voir la magie de Flux.

En fait Flux va juste réappliquer la configuration, donc si vous ne supprimez pas l'Ingress Controller, il va juste le mettre à jour. Il utilisera `install` ou `upgrade` en fonction de l'état de l'objet HelmRelease.

```
helm list -A
helm uninstall ingress-nginx -n nginx-ingress-controller
helm list -A
kubectl get all -n nginx-ingress-controller
kubectl delete ns nginx-ingress-controller
kubectl get ns
```

ON ENVOIE LA SAUCE

On va maintenant déplacer le fichier `flux/nginx-ingress-controller.yaml` dans le répertoire `flux/devoxx-cluster/` comme pour les repos Helm.

1

```
cp flux/nginx-ingress-controller.yaml flux/devoxx-cluster/nginx-ingress-controller.yaml
git add flux/devoxx-cluster/nginx-ingress-controller.yaml
git commit -am ":satellite_orbital: Setup Ingress Controller" && git push
```

On observe la synchro avec la commande suivante :

```
flux get kustomizations --watch
```

```
helm list -A
echo "Helm list ne nous retourne rien car c'est Flux qui gère maintenant"
sleep 20
echo "On utilise : "
kubectl get HelmRepository -A
kubectl get HelmChart -A
kubectl get HelmRelease -A
kubectl get ns
kubectl get all -n nginx-ingress-controller
```

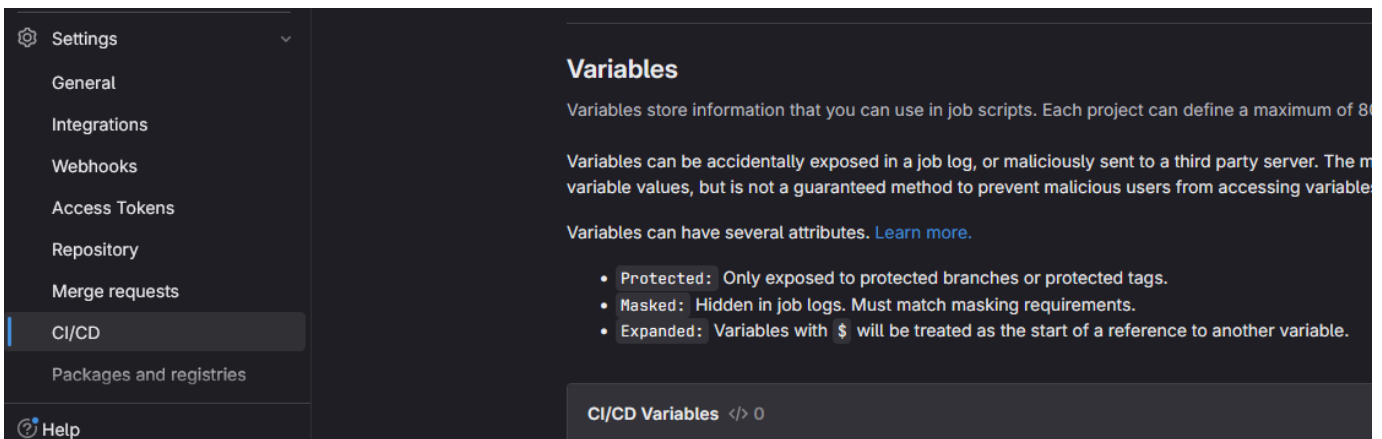
External Secrets

On va maintenant déployer notre External Secrets. Comme ça pas besoin de stocker les secrets à la main.

SETUP DES SECRETS

Commençons par créer les variables d'environnement dans notre repo GitLab. (Ils ne sont pas repris lors du fork et heureusement ...).

- Soit on passe par l'UI de Gitlab



- Soit on utilise l'API de Gitlab

Il vous faut l'ID du projet, vous pouvez le trouver ici :

Project
kub-workshop-snowcamp-2024

Pinned

Issues 0

Merge requests 0

Manage

Plan

Code

Build

Deploy

Operate

Monitor

Settings

General

Integrations

GitLab Pages has moved
To go to GitLab Pages, on the left sidebar, select [Deploy > Pages](#).

Naming, topics, avatar

Update your project name, topics, description, and avatar.

Project name
kub-workshop-snowcamp-2024

Project ID
54075302

Topics
Search for topic

Topics are publicly visible even on private projects. Do not include sensitive information in topic names. [Learn more](#).

Project description (optional)

Project avatar
Choose file... No file chosen.

```
PROJECT_ID= <YOUR_PROJECT_ID>
```

```
curl --request POST --header "PRIVATE-TOKEN: $GITLAB_TOKEN" \
  "https://gitlab.com/api/v4/projects/${PROJECT_ID}/variables" \
  --form "key=API_MAIL" --form "value=${API_MAIL}"

curl --request POST --header "PRIVATE-TOKEN: $GITLAB_TOKEN" \
  "https://gitlab.com/api/v4/projects/${PROJECT_ID}/variables" \
  --form "key=API_KEY" --form "value=${API_KEY}"
```

Il ne nous reste plus qu'à :

- créer notre `Secret` dans notre cluster Kubernetes [comme vu précédemment](#), mais cette fois avec votre compte.
- modifier le fichier `./external-secrets/flux/external-secrets-secret-store.yml` pour y mettre votre ID project.

```
1
```

Par exemple avec :

```
sed -i "s/<PROJECT_ID>/$PROJECT_ID/" ./external-secrets/flux/external-secrets-secret-store.yml
```

PRÉSENTATION DE NOTRE KUSTOMIZATION

```
1
```

- Création du NS
- Déploiement des CRDS (Custom Resource Definition)
- Déploiement de l'opérateur
- Déploiement des CRS (Custom Resource) [les différents providers](#) (ici GitLab)
- La définition de notre secret store

CHAUD DEVANT !

Fin prêt pour lancer la commande :

```
cp flux/external-secrets.yaml flux/devoxx-cluster/external-secrets.yaml
git add flux/devoxx-cluster/external-secrets.yaml
git commit -am ":satellite_orbital: Setup External Secret" && git push
```

External DNS && Cert-manager

Un dernier petit tips pour la route :

1

La commande `dependsOn` permet de définir une dépendance entre les différents objets Flux.

```
flowchart TD
  A[Cert Manager] -->|dependsOn| B[external-dns];
  A[Cert Manager] -->|dependsOn| C[nginx-ingress-controller];
  B -->|dependsOn| D[external-secrets];
```

On s'assure ainsi que les différentes recettes sont appliquées dans l'ordre.

Retournons à la recette pour encore plus de découvertes ➡