

# CI/CD gitops

Le but de cet exercice est de mettre en place un pipeline d'intégration et de déploiement continu. La partie *déploiement* sera basée sur l'approche *GitOps*: un processus tournant à l'intérieur du cluster sera chargé de détecter les modifications effectuées dans un repository Git et, en fonction de celles-ci, mettra l'application à jour.

Pour ce faire, vous allez effectuer les actions suivantes:

- coder un serveur web très simple
- créer un projet *GitLab* pour gérer les sources
- mettre en place un cluster Kubernetes
- mettre en place un pipeline d'intégration continu avec GitLab CI
- mettre en place un pipeline de déploiement continu en utilisant *ArgoCD*
- test de la chaîne complète

Le but de l'ensemble de ces actions étant qu'une modification envoyée dans le projet GitLab déclenche automatiquement les tests et le déploiement de la nouvelle version du code sur le cluster Kubernetes.

---

## 1. Création d'un serveur web simple

Créez un folder nommé *api* sur votre machine locale puis positionnez vous dans celui-ci.

En utilisant le langage de votre choix, développez un serveur web simple ayant les caractéristiques suivantes:

- écoute sur le port 8000
- expose le endpoint / en GET
- retourne la chaîne 'Hi!' pour chaque requête reçue

Créez également un Dockerfile pour packager le serveur dans une image Docker.

### Exemple de serveur en Python

- app.py

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hi!"

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=8000)
```

- requirements.txt

```
Flask==1.0.2
```

- Dockerfile

```
FROM python:3.4
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
EXPOSE 8000
CMD python /app/app.py
```

## Construction de l'image

Créez ensuite une image, nommée *api*, dans laquelle sera packagé le serveur mise en place dans l'étape précédente:

```
$ docker build -t api .
```

## Test

Une fois l'image créée, lancez un container avec la commande suivante:

```
$ docker run --name api -d -p 8000:8000 api
```

Puis vérifiez que le serveur fonctionne correctement:

```
$ curl http://localhost:8000
Hi!
```

Supprimez ensuite le container:

```
$ docker rm -f api
```

---

## 2. Gestion du projet dans GitLab

### 2.1. Création d'un repository

1. Créez un compte sur GitLab ou utilisez un compte existant.

1. Créez un projet sans utiliser de template

Nommez le *demo-api* Sélectionnez *Public* dans le champ *Visibility*.

## 1. Push du code dans GitLab

Suivez les instructions présentées dans la section "Push an existing folder" afin de pusher votre projet dans ce repository GitLab.

Il sera nécessaire de lancer les commandes suivantes depuis le folder *api* créé précédemment.

🔥 Assurez-vous de remplacer *GITLAB\_USER* par votre nom d'utilisateur

```
git init
git remote add origin git@gitlab.com:GITLAB_USER/demo-api.git
git add .
git commit -m "Initial commit"
git push -u origin master
```

## 2.2. Mise en place d'un pipeline d'intégration continue

Créez un fichier `.gitlab-ci.yml` à la racine de votre projet et assurez vous qu'il contienne les instructions suivantes:

```
stages:
  - package

push image docker:
  image: docker:stable
  stage: package
  services:
    - docker:18-dind
  script:
    - docker build -t $CI_REGISTRY_IMAGE:latest -t
    $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA .
    - docker login -u gitlab-ci-token -p $CI_BUILD_TOKEN $CI_REGISTRY
    - docker push $CI_REGISTRY_IMAGE:latest
    - docker push $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA
```

Ces instructions définissent un *stage* nommé *package* contenant les commandes servant à créer une image Docker et à envoyer celle-ci dans le registry GitLab. 2 tags seront apposés sur cette image:

- latest
- la version courte du commit de Git

Commitez l'ajout de ce fichier et envoyez ces changements sur GitLab:

```
git add .gitlab-ci.yml
git commit -m 'Add GitLab pipeline'
```

```
git push origin master
```

Depuis le menu *CI / CD* de l'interface de GitLab, vérifiez que la pipeline a été déclenchée.

Une fois que cette pipeline est terminée, allez dans le menu *Packages -> Container Registry* et vérifiez que cette première image est maintenant présente dans le registry.

## 2.3. Ajout d'un scanner de vulnérabilité

Dans le fichier `.gitlab-ci.yml`, ajoutez une nouvelle entrée *scan* sous la clé *stage*.

```
stages:
  - package
  - scan
```

A la fin du fichier, ajoutez le step *scanning* suivant:

```
scanning:
  image: alpine:3.13
  stage: scan
  script:
    - apk add -u curl
    - curl -sL
      https://raw.githubusercontent.com/aquasecurity/trivy/master/contrib/install
      .sh | sh -s -- -b /usr/local/bin
    - trivy $CI_REGISTRY_IMAGE:latest > cve.txt
  artifacts:
    paths: [cve.txt]
```

Ce step met en place le scanner de vulnérabilité *Trivy* et lance un scan de l'image créée dans le step précédent.

Commitez l'ajout de ce fichier et envoyez ces changements sur GitLab:

```
git add .gitlab-ci.yml
git commit -m 'Add CVE scanner'
git push origin master
```

Un nouveau pipeline sera déclenché, le résultat du scan sera disponible dans les artefacts depuis l'interface Gitlab.

🔥 si votre image contient de nombreuses CVE, essayer d'utiliser une image de base plus légère, par exemple une image basée sur la distribution *Alpine*.

## 2.4. Ajout de tests d'intégration

Dans le fichier `.gitlab-ci.yml`, ajoutez une nouvelle entrée *integration* sous la clé *stage*.

```
stages:
  - package
  - scan
  - integration
```

A la fin du fichier, ajoutez le step *integration test* suivant:

```
integration test:
  image: docker:stable
  stage: integration
  services:
    - docker:18-dind
  script:
    - docker run -d --name myapp $CI_REGISTRY_IMAGE:latest
    - sleep 10s
    - TEST_RESULT=$(docker run --link myapp lucj/curl -s http://myapp:8000)
    - echo $TEST_RESULT
    - $([ "$TEST_RESULT" == "Hello World!" ])
```

Ce step définit un test de l'image créée. Il vérifie que la chaîne "Hello World!" est retournée.

Envoyez ces modifications dans le repository GitLab:

```
git add .gitlab-ci.yml
git commit -m 'Add integration step'
git push origin master
```

Depuis l'interface GitLab, vérifiez que la pipeline est déclenchée.

Vous devriez voir que l'étape *integration* a terminé en erreur.

Regardez dans les logs du job, corrigez le code et envoyez les changements dans GitLab. Vérifiez ensuite que le job passe correctement cette fois ci.

---

## 3. Mise en place d'un cluster Kubernetes

🔥 si vous avez déjà accès à un cluster, vous pouvez passer directement à la section 4 qui suit.

Si vous n'avez pas encore de cluster, la procédure ci dessous vous permettra de mettre en place un cluster [k3s](#) très facilement sur une VM créée en utilisant *Multipass*:

- *K3s* est une distribution Kubernetes très légère créée par [Rancher](#) et donnée à la [CNCF](#) en Août 2020.
- *Multipass* est un utilitaire développé par Canonical qui permet de créer très facilement des machines virtuelles Ubuntu en local. *Multipass* s'intègre de manière native à différents hyperviseurs:
  - Hyper-V (Windows)
  - HyperKit (masOS)
  - KVM (Linux)
  - VirtualBox (Windows, macOS, Linux)

Installez tout d'abord *Multipass* en suivant les instructions depuis <https://multipass.run>, puis utilisez la commande suivante pour créer une machine virtuelle:

```
$ multipass launch -c 2 -m 2G -n k3s -d 10G
```

Ensuite, installez [k3s.io](#) dans cette nouvelle VM, l'installation ne devrait prendre que quelques dizaines de secondes:

```
$ multipass exec k3s -- bash -c "curl -sL https://get.k3s.io | sh -"
```

Les commandes suivantes permettent de récupérer le fichier de configuration du cluster et de configurer le binaire *kubectl* présent sur la machine locale:

🔥 assurez-vous de bien avoir installé *kubectl* sur votre machine au préalable

```
IP=$(multipass info k3s | grep IPv4 | awk '{print $2}')
multipass exec k3s -- sudo cat /etc/rancher/k3s/k3s.yaml > k3s.yaml
sed -i 's/127.0.0.1/$IP/' k3s.yaml
export KUBECONFIG=$PWD/k3s.yaml
```

Vous pouvez à présent interagir avec votre cluster depuis votre machine locale. Commencez par lister les nodes présent (un seul dans cet exemple):

```
$ kubectl get nodes
NAME      STATUS    ROLES                  AGE      VERSION
k3s       Ready     control-plane,master   2m9s     v1.20.4+k3s1
```

---

## 4. Manifests de déploiement de l'application

A la racine de votre projet, créez le répertoire *manifests* et copiez les 2 spécifications suivantes dans celui-ci.

Note: remplacez *GITLAB\_USER* avec votre nom d'utilisateur GitLab et *GITLAB\_PROJECT* avec le nom de votre projet sur GitLab

- Fichier *deploy.yaml*:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api
  labels:
    app: api
spec:
  replicas: 1
  selector:
    matchLabels:
      app: api
  template:
    metadata:
      labels:
        app: api
    spec:
      containers:
      - name: api
        image: registry.gitlab.com/GITLAB_USER/GITLAB_PROJECT:latest
        imagePullPolicy: Always
```

- fichier *service.yaml*:

```
apiVersion: v1
kind: Service
metadata:
  name: api
spec:
  type: NodePort
  ports:
    - name: api
      nodePort: 31000
      port: 80
      targetPort: 8000
      protocol: TCP
  selector:
    app: api
```

Commitez ces changements et envoyez les dans le repository GitLab:

```
git add manifests
git commit -m 'Add manifests files'
git push origin master
```

Vous utiliserez ces spécifications dans la suite de l'exercice afin de déployer votre api dans Kubernetes.

## 5. ArgoCD

[ArgoCD](#) est un projet de la [CNCF](#), actuellement dans l'état *Incubating*. ArgoCD est utilisé pour effectuer du Continuous Delivery dans un cluster Kubernetes, il met en avant une approche *GitOps* que nous allons illustrer dans la suite.

### Installation

Tout d'abord, installez ArgoCD dans votre cluster en utilisant les commandes suivantes:

```
kubectl create namespace argocd
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
```

Plusieurs Pods sont créés dans le namespace *argocd*:

```
$ kubectl get pod -n argocd
```

NAME	READY	STATUS	RESTARTS	AGE
argocd-redis-6fb68d9df5-gc27h	1/1	Running	0	2m47s
argocd-repo-server-6dcdb9cb-4pg4w	1/1	Running	0	2m47s
argocd-dex-server-567b48df49-frx5h	1/1	Running	0	2m48s
argocd-application-controller-0	1/1	Running	0	2m47s
argocd-server-69678b4f65-7c9sm	1/1	Running	0	2m47s

En utilisant la commande suivante (port-forward), accédez à l'interface web de ArgoCD:

```
kubectl port-forward svc/argocd-server -n argocd 8080:443
```

Le login est *admin*, le mot de passe peut-être retrouvé grâce à la commande suivante:

```
kubectl get pods -n argocd -l app.kubernetes.io/name=argocd-server -o name
| cut -d '/' -f 2
```

Exemple de mot de passe:

```
argocd-server-69678b4f65-7c9sm
```

## Définition de l'application

Afin de pouvoir utiliser ArgoCD pour gérer le déploiement de votre application, vous allez tout d'abord ajouter celle-ci via l'interface web. Pour cela, il suffit de cliquer sur *Create Application* et de remplir les champs du formulaire:

GENERAL:

- application name: *api* mais vous pouvez indiquer ce que vous souhaitez
- project: *default*
- Sync policy: laissez *Manual* pour le moment

SOURCE:

- Repository URL: indiquez l'URL de votre projet sous GitLab dans le format [https://gitlab.com/GITLAB\\_USER/demo-api.git](https://gitlab.com/GITLAB_USER/demo-api.git)
- Revision: laissez la valeur *HEAD*



- Path: indiquez le sous répertoire dans lequel se trouvent les spécifications yaml, ici *manifests*

DESTINATION:

- Cluster URL: indiquez l'URL <https://kubernetes.default.svc> car l'application sera déployée dans le cluster dans lequel Argo est installé
- Namespace: indiquez *default* le namespace dans lequel l'application sera lancée

Cliquez ensuite sur *Create* pour ajouter l'application. Après quelques secondes celle-ci apparaîtra en *OutOfSync* car les fichiers manifests présents dans Git (considéré comme la *Source Of Trust*) n'ont pas encore été créés dans le cluster.

## Synchronisation manuelle

Lancez ensuite une synchronisation manuelle, celle-ci créera le Service et le Deployment dans le cluster.

L'application aura rapidement le statut *Synced*

Assurez-vous alors que les ressources ont été créées correctement dans le cluster

```
$ kubectl get deploy,pod,svc
```

Vous devriez obtenir un résultat similaire à celui ci-dessous:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/api	2/2	2	2	2m14s

  

NAME	READY	STATUS	RESTARTS	AGE
pod/api-589bbfcd68-swb2s	1/1	Running	0	2m14s
pod/api-589bbfcd68-5xsrr	1/1	Running	0	2m14s

  

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
service/kubernetes	ClusterIP	10.43.0.1	<none>	443/TCP
14m				
service/api	NodePort	10.43.242.188	<none>	80:31000/TCP
2m15s				

En utilisant l'adresse IP de la VM dans laquelle tourne Kubernetes, vérifiez que le serveur web est disponible sur le port 31000.

```
$ curl http://$IP:31000  
Hi!
```

## Désynchronisation

Depuis la ligne de commande, changez le nombre de réplicas de l'application:

```
kubectl scale deployment api --replicas=3
```

Etant donné que le nouveau nombre de réplica ne correspond pas à celui spécifié dans Git, l'application apparaît maintenant *OutOfSync*: elle n'est plus alignée avec la définition présente dans le répertoire *manifests*.

Lorsque vous cliquez sur l'application, vous obtenez une vue détaillée de celle-ci:

Le bouton *App Diff* permet notamment de voir la différence entre la spécification qui tourne dans la cluster et celle présente dans le répertoire Git. Ici, seul le nombre de réplicas diffère:

L'interface de ArgoCD permet d'avoir des informations détaillées sur la façon dont votre application est synchronisée, l'historique de synchronisation, ... N'hésitez pas à naviguer entre les différents menus afin de mieux prendre en main ce produit.

## Synchronisation automatique

Vous allez à présent activer la synchronisation automatique et façon à ce que ArgoCD déploie automatiquement les changements qui sont détectés dans le repository GitLab.

Il suffit pour cela d'aller dans l'onglet *App Details* et de descendre jusqu'à la rubrique *Sync Policy*:

Changez ensuite les options de façon à avoir la configuration suivante:

Si vous modifiez une nouvelle fois le nombre de réplicas en ligne de commande, vous verrez que la synchronisation se fera automatiquement afin de ramener ce nombre à 2 (valeur précisée dans la spécification existante dans GitLab).

```
kubectl scale deployment api --replicas=1
```

---

## 6. Mise en place du deployment automatique

Dans l'étape *package* du pipeline d'intégration continu, vous avez créé une image ayant les tags *latest* et *\$CI\_COMMIT\_SHORT\_SHA*. Vous allez maintenant mettre en place une nouvelle étape dans laquelle vous mettrez à jour le fichier *manifests/deploy.yaml* en utilisant le tag *\$CI\_COMMIT\_SHORT\_SHA*.

Ajoutez un nouveau *stage* nommé *tag* dans le fichier *.gitlab-ci.yaml*:

```
stages:
  - package
  - scan
  - integration
  - tag
```

Pour ce stage, ajouter les instructions suivantes à la fin du fichier:

🔥 remplacez *GITLAB\_USER* par votre nom d'utilisateur

```
update_manifest:
  stage: tag
  image: alpine:3.13
  variables:
    USER: ***GITLAB_USER***
    PROJECT: demo-api
    REGISTRY: registry.gitlab.com
    GIT_REPOSITORY: git@gitlab.com:$USER/$PROJECT.git
  before_script:
    - apk add --no-cache git
    - git remote set-url origin
      https://$USER:${CI_PUSH_TOKEN}@gitlab.com/$USER/$PROJECT.git
    - git config --global user.email "gitlab@gitlab.com"
    - git config --global user.name "GitLab CI/CD"
    - git checkout -B master
  script:
    - cd manifests
    - sed -i "s/image:\ $REGISTRY.*/image:\
      $REGISTRY\/$USER\/$PROJECT:$CI_COMMIT_SHORT_SHA/" deploy.yaml
    - git commit -am '[skip ci] update image tag'
    - git push origin master
  only:
    - master
```

Ce stage met à jour le fichier *manifests/deploy.yaml* et envoie les modifications dans GitLab. L'utilisation de la chaîne *[skip ci]* dans le message du commit assure qu'un nouveau pipeline d'intégration ne sera pas déclenché sur ce commit.

### Access Token

Étant donné qu'une mise à jour du repository sera effectuée depuis le pipeline d'intégration, il faut tout d'abord créer un Access Token. Celui-ci peut être créé à partir de votre profil GitLab.

Il faut ensuite l'ajouter en tant que valeur de la variable d'environnement `CI_PUSH_TOKEN`. Cela peut être effectué depuis le menu *Settings* -> *CI/CD* -> *Variable* du projet GitLab.

Envoyez les modifications dans le repository GitLab:

```
git add manifests
git commit -m 'Add tag step'
git push origin master
```

---

## 7. Tests

Pour vérifier qu'un changement dans le code du serveur déclenche une mise à jour de l'application, effectuez les modifications suivantes:

- changez la chaîne de caractères retournée par le serveur en "Hello!"
- modifiez le test présent dans `.gitlab-ci.yml`

Publiez ensuite les changements:

```
git add manifests
git commit -m 'Change returned string'
git push origin master
```

Un nouveau pipeline d'intégration est lancé. La dernière étape de celui-ci (*update manifest*) modifie le fichier `manifests/deploy.yaml` en mettant à jour le tag de l'image. ArgoCD va donc détecter une divergence: l'application qui tourne dans le cluster ne correspond plus à la spécification présente dans GitLab, et va automatiquement synchroniser l'application en utilisant la nouvelle version de la spécification.

Une fois le pipeline terminé, attendez quelques instants puis testez une nouvelle fois l'application. Vous devriez obtenir la nouvelle chaîne de caractères.

```
$ curl http://$IP:31000
Hello!
```

---

## En résumé

Cet exercice met en évidence l'approche *GitOps* qui considère Git comme le référentiel (Source of Truth): une application tournant dans un cluster doit constamment refléter les spécifications présentes dans Git.

Nous avons vu un aperçu de [ArgoCD](#), l'un des outils phares utilisés dans cette approche GitOps. D'autres outils existent, comme [FluxCD](#) un projet Incubating de la CNCF. N'hésitez notamment à refaire cet exercice à l'aide de FluxCD si vous le souhaitez.

L'approche *GitOps* est une grande tendance qui s'est dessinée ces quelques dernières années. Je vous encourage à regarder l'écosystème *GitOps* de plus près, vous trouverez plus d'information sur <https://www.gitops.tech/>.