Installation infra Kapsule 2020

Avant de commencer

Quelques petites choses à réfléchir avant de se lancer. Quelle version de moteur network Kubernetes, quel OS etc.

En vrac :

- Choisir son moteur network : https://kubernetes.io/fr/docs/concepts/services-networking/service/
- Matrice de compatibilité rancher 2.4.2 https://rancher.com/support-maintenance-terms/all-supported-versions/rancher-v2.4.2/
- La dernière LTS ubuntu en ce mois de mai 2020 https://wiki.ubuntu.com/Releases et à ce jour c'est la version 20.04

Installation Kapsule

Dans **l'interface web scaleway** https://console.scaleway.com, aller dans la partie **kapsule** et créer son cluster kubernetes.

Les infos à saisir :

- Nom k8s-01 dans mon cas. Evidemment, mettez ce que vous voulez
- Les deux points suivant sont dans les options avancées du cluster kubernetes
- Pas de *kubernetes dashboard* car je prévois d'installer *rancher 2* qui fournit son propre dashboard
- Pas de `ingress` car je prévois d'installer le mien, à ma façon
- *A ce stade, on peut se poser la question de rattacher lecluster à un network privé, je choisis de ne pas le faire surtout que ce sera payant à un moment*
- Choisir la version de Kubernetes 1.18 dans mon cas
- Choix de 3 noeuds `DEV-M` pour limiter le cout tout en ayant quand même 3 noeuds
- Pour le moteur réseau, choix de flannel car plus performant (plus light en ressources pour mes petits DEV-M) et pas besoin de grosse sécurité

Installation serveur d'admin

Création du serveur

J'ai un PC Windows et je n'ai pas envie de devoir installer sur mon poste perso tout l'outillage pour piloter mon cluster Kubernetes. Ceci d'autant plus que je peux être amené à devoir intervenir sur le serveur sans avoir mon PC perso sous la main.

Je prévois donc de déployer en plus un simple **serveur linux** `DEV-S` chez scaleway avec tout l'outillage d'amdin installé dessus.

Pour cela, go sur la console web scaleway et ajout d'une instance avec les informations ci-dessous.

Les infos que j'utilise pour l'installation :

- Nom scw-k8s-adm-01 mais choisissez ce que vous voulez
- Ubuntu LTS 20.04

Un peu d'attente et ensuite il est possible de s'y connecter en SSH.

[INFO] On suppose ici que la partie génération de clé publique / privée pour l'accès SSH a déjà été fait et ajouté dans l'admin scaleway pour être associé au serveur déployé.

Première connection & Maj system

On se connecte en SSH avec par exemple `putty` sans oublier d'avoir préchargé sa clé dans `pagent`.

Puis, comme on est bien élevés, avant toute chose on commence par mettre à jour le système.

```
apt-get update
apt-get upgrade -y
```

Installation kubectl

A ce stade, il s'agit d'installer sur le serveur l'outil `kubectl`, qui nous permettra de piloter le cluster Kubernetes en ligne de commande.

La documentation de référence :
https://kubernetes.io/fr/docs/tasks/tools/install-kubectl/#installer-kubectl-sur-linux

```
apt-get update && apt-get install -y apt-transport-https
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add
-
echo "deb https://apt.kubernetes.io/ kubernetes-xenial main" | tee -a
/etc/apt/sources.list.d/kubernetes.list
apt-get update
apt-get install -y kubectl
```

Installation du fichier kubeconfig

Pour pouvoir piloter le cluster kubernetes, il faut le fichier `kubeconfig` qu'on va ensuite installer sur le serveur d'admin.

Rendez-vous dans l'interface d'admin **kapsule** scaleway pour le téléchargement du kubeconfig.

Puis installation sur le serveur après téléchargement :

```
mkdir ~/.kube
cp kubeconfig-k8s-01.yaml  ~/.kube/config
```

Pour vérifier que tout fonctionne on essaye d'obtenir les infos du cluster kapsule :

```
kubectl cluster-info
```

Installation helm

Après avoir regardé un peu, il m'a semblé que le plus simple d'installation et de mise à jour dans le temps consiste à utiliser `snap`.

```
snap install helm --classic
```

Ce qui est amusant et qui n'est pas dit de manière évidente c'est que le repo helm `stable` de base n'est pas dispo out of the box. Il faut donc installer le repo. J'ai trouvé les informations ici : https://github.com/helm/charts/issues/19279

```
helm repo add stable https://kubernetes-charts.storage.googleapis.com
```

Configuration Kapsule

[IMPORTANT] A ce stade vous devez savoir que, pour des raisons de cout notamment, j'ai choisi de ne déployer **qu'un seul ingress / loadbalancer**. J'y reviens un peu plus tard. Cela implique donc des configuration séparées, ou je déploie mes back-end d'un côté et met à jour mon ingress manuellement en central de l'autre.

Digression sur la récupération de l'ip client source

[INFO] Cette étape est clairement optionelle et ne fait pas partie de l'installation en tant que tel. J'ai préféré la noter car à un moment c'est ce qui m'a aidé pour déboguer et résoudre mes soucis d'ip client source qui n'allait pas jusqu'à mon backend wordpress.

J'avoue que cette documentation m'a été utile :
https://docs.ovh.com/gb/en/kubernetes/getting-source-ip-behind-loadbalancer/

Aussi bien pour comprendre un peu mieux ce qu'il y avait à faire mais aussi pour avoir les moyens de tester sans trop me prendre la tête.

[IMPORTANT] A noter ici que si vous suivez tout ce qui est dans ce guide vous déploierez un autre ingress. Encore une fois cette partie n'avance pas la configuration en tant que tel mes ses conclusions sont intégrées dans les autres parties de ce guide.

Tester avec ce guide et des ingress à part, c'est une chose que j'ai d'ailleurs faite au début car je pensais que mon *impossibilité* d'avoir l'ip source réelle (oui c'était une vrai guerre) venait de ma façon de déployer l'ingress. Ce qui n'était pas tout à fait vrai. Oui il y en avait un peu à cet endroit mais aussi a d'autres...

Digression sur la configuration SSL de l'ingress

[IMPORTANT] Cette section est à vocation explicative principalement car elle a été intégrée à la configuration ingress nginx citée par ailleurs dans ce guide.

Chose intéressante, j'ai remarqué qu'avec internet explorer 11 *(oui je sais c'est mal mais parfois on a pas le choix)*, mon blog n'était pas accessible en https pour une raison obscure. Après enquête *(pas facile, j'ai du déduire un peu)*, il s'avère que c'est parceque les paramètres TLS et les **algorithmes fournis de base par ma configuration TLS serveur son trop récents** et non pris en compte par ce vieux ie 11 (et d'autres config de là même époque).

Pour y voir plus clair j'avoue que https://www.ssllabs.com a été d'une grande aide pour avoir des infos sur la compatibilité et ce que mon serveur sait faire.

Quelques liens utiles:

- https://www.linode.com/docs/web-servers/nginx/tls-deployment-best-practices-for-nginx/
- https://github.com/kubernetes/ingress-nginx/blob/master/docs/user-guide/nginx-configuration/annotations.md#ssl-ciphers
- https://www.ssllabs.com/ssltest/analyze.html?d=alban.montaigu.io

Donc pour améliorer la compatibilté il suffit d'ajouter 1 ou deux `ciphers` *weak* qui sont pris en charge par les vieux navigateurs :

```
ssl-protocols: "TLSv1.2 TLSv1.3"
ssl-ciphers: "EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH;"
```

A noter que si vous voulez éviter de forcer le tls partout sur vos déploiement web quand ya des soucis de conf / compat mieux vaut désactiver la redirection *(qui se fait par défaut je crois)* :

```
ssl-redirect: "false"
```

Sinon les suites cipher ci-dessous (weak) ne sont pas dispo et ne premettent plus de supporter les vieilles config comme justement notre ami ie11.

```
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 (0xc028)   ECDH x25519 (eq. 3072 bits
RSA)   FS   WEAK    256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)   ECDH x25519 (eq. 3072 bits
RSA)   FS   WEAK
```

Les commandes ci-dessous peuvent être utiles pour debug un peu le TLS *(a adapter en fonction de votre domaine bien sur)* :

```
curl -v --tlsv1.3 -s https://alban.montaigu.io/wp-json > /dev/null
curl -v -s http://alban.montaigu.io/wp-json > /dev/null
```

Installation & configuration de l'ingress

Après toutes ces **digressions** voici ce que cela donne.

La documentation que j'ai utilisée et qui m'a servi à un moment ou un autre :

- https://kubernetes.github.io/ingress-nginx/deploy/#using-helm
- https://www.digitalocean.com/community/tutorials/how-to-set-up-an-nginx-ingress-on-digitaloce

an-kubernetes-using-helm
- https://hub.helm.sh/charts/nginx/nginx-ingress
- https://github.com/kubernetes/ingress-nginx/blob/master/charts/ingress-nginx/values.yaml
- https://www.asykim.com/blog/deep-dive-into-kubernetes-external-traffic-policies
- https://www.ovh.com/blog/getting-external-traffic-into-kubernetes-clusterip-nodeport-loadbalancer-and-ingress/

[IMPORTANT] La configuration du paramètre `externalTrafficPolicy` doit être à `Local` pour éviter les hop dans le cluster kubernet et préserver l'ip client source. En effet, il ne faut pas oublier que les loadbalancer / ingress vont être en front de vos applications et si vous voulez que l'ip client passe jusqu'à votre application web il va falloir faire quelques efforts de configuration en commencant par celui là **(ce n'est pas le seul malheureusement)**.

Préparation de l'installation du ingress avec helm :

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
kubectl create namespace ingress-nginx
```

Ensuite avec le fichier de configuration suivant :

ingress_nginx_helm_values.yaml

```
## nginx configuration
## Ref: https://github.com/kubernetes/ingress/blob/master/controllers/nginx/configuration.md
##
controller:

  # Will add custom configuration options to Nginx https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuration/configmap/
  config: {
    proxy-connect-timeout: "30",
    proxy-read-timeout: "1800",
    proxy-send-timeout: "1800",
    proxy-body-size: "500m",
    use-proxy-protocol: "true",
    proxy-real-ip-cidr: "51.159.75.81/32",
    compute-full-forwarded-for: "true",
    use-forwarded-headers: "true",
    ssl-redirect: "true",
    ssl-protocols: "TLSv1.2 TLSv1.3",
    ssl-ciphers: "EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH;"
  }

  ## Allows customization of the source of the IP address or FQDN to report
  ## in the ingress status field. By default, it reads the information provided
  ## by the service. If disable, the status field reports the IP
```

```
address of the
  ## node or nodes where an ingress controller pod is running.
  publishService:
    enabled: true

  service:

    annotations: {
    #  service.beta.kubernetes.io/scw-loadbalancer-send-proxy-v2:
"true"
    }

    ## Set external traffic policy to: "Local" to preserve source IP on
    ## providers supporting it
    ## Ref:
https://kubernetes.io/docs/tutorials/services/source-ip/#source-ip-for-
services-with-typeloadbalancer
    externalTrafficPolicy: Local
```

[INFO] `51.159.75.81` correspond à l'ip du loadbalancer. Si elle n'est pas connue à l'avance, commencer par `0.0.0.0/32` puis la mettre à jour ensuite. C'est assez important car c'est ce paramètre qui va dire a nginx quel serveur continent la valeur ip client source a retenir dans les headers proxy.
[INFO] A noter que les paramètres `timeout` et `body-size` du proxy / ingress sont importants si vous avez des applications qui ont besoin d'uploader de gros fichiers *(utile par exemple pour l'import / export de wordpress)* d'une part et / ou faire de gros traitements d'autre part.
[IMPORTANT] A ce state vous pouvez choisir d'activer ou non le **proxy protocol** et tout ce qu'il va avec. Néanmoins, dans mes expériences c'est un choix qui prête à conséquence. En effet, l'activer permettra à vos applications d'avoir à terme l'**ip source client** *(par exemple les modules de stats wordpress)*. Le **problème** de ca c'est que de ce que j'ai pu constater, activer cela faisait à minima planter certaines résolutions réseau “locales”.

Je m'explique, si jamais vous donnez un domaine associé a l'ip de l'ingress / loadbalancer pour pointer sur vos applications, j'ai constaté que les `curl` vers ce domaine depuis n'importe quel conteneur du cluster ne fonctionnaient pas *(je m'en suis rendu compte car l'outil de **santé wordpress** utilise curl pour tester le loopback et ce dernier plantait lamentablement)*.

Je n'ai pas réellement trouvé de solution à vrai dire. La seule chose que j'ai constaté c'est que ce problème n'arrive pas quand on laisse le ingress / loadbalancer **en mode tcp de base sans proxy**.

On déploie l'ingress avec les bonnes options de configuration :

```
helm install ingress-nginx ingress-nginx/ingress-nginx -f
ingress_nginx_helm_values.yaml -n ingress-nginx
```

A noter qu'il est possible aussi d'appliquer des configurations **à postériori** comme décrit ci dessous.

Changer la configuration de l'ingress

Un peu de documentation utile *(intégrée aussi pour la définition du helm values)* :

- https://stackoverflow.com/questions/54884735/how-to-use-configmap-configuration-with-helm-nginx-ingress-controller-kubernet
- https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuration/annotations/#custom-max-body-size
- https://github.com/kubernetes/ingress-nginx/issues/2007
- https://github.com/kubernetes/ingress-nginx/issues/3529
- https://github.com/kubernetes/ingress-nginx/issues/1067

[IMPORTANT] Cette étape peut être **optionelle** grâce aux helm values du fichier `ingress_nginx_helm_values.yaml` mais c'est bon à savoir que vous pouvez l'utiliser s'il faut changer quelque chose dans la configuration de votre ingress.

Un exemple du fichier de configuration en question :

ingress_nginx_config.yaml

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: ingress-nginx-ingress-nginx-controller
  namespace: ingress-nginx
data:
  proxy-connect-timeout: "30"
  proxy-read-timeout: "1800"
  proxy-send-timeout: "1800"
  proxy-body-size: "500m"
  use-proxy-protocol: "true"
  proxy-real-ip-cidr: "51.159.75.81/32"
  compute-full-forwarded-for: "true"
  use-forwarded-headers: "true"
  ssl-redirect: "true"
  ssl-protocols: "TLSv1.2 TLSv1.3"
  ssl-ciphers: "EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH;"
```

Et la commande pour l'appliquer :

```
kubectl apply -f ingress_nginx_config.yaml
```

[IMPORTANT] Le déploiement d'ingress va créer automatiquement un **loadbalancer** associé chez scaleway. C'est assez important à savoir car de mémoire il en va d'environ 9 euros par mois il vaut donc mieux le savoir avant de déployer des ingress partout.

C'est d'ailleurs un problème de fond que je n'ai jamais vu sur le net. En effet, nombre de déploiements helm comportent un ingress intégré. Quand on est innocent, on déploie sans trop se poser de question. Sauf qu'à la fin on se rend compte qu'**on a déployé une 10aine de load balancer à 9 euros par mois** . Je ne sais pas pour vous mais pour moi, c'est vraiment trop cher !

Installation rancher

A ce stade, on doit avoir un cluster kubernetes de base configuré et bien sur le serveur d'admin qui nous a permis d'y arriver. Passons maintenant a l'installation de **Rancher** qui permettra de gérer les déploiements d'application sur kubernetes. Cela permet aussi d'avoir un outil d'admin potable.

Les documentations de référence :

- https://rancher.com/docs/rancher/v2.x/en/installation/
- https://rancher.com/docs/rancher/v2.x/en/installation/k8s-install/helm-rancher/

On prépare les repo helm :

```
helm repo add rancher-stable https://releases.rancher.com/server-
charts/stable
helm repo add jetstack https://charts.jetstack.io
helm repo update
```

On crée les namespaces qui seront utiles plus tard :

```
kubectl create namespace cattle-system
kubectl create namespace cert-manager
```

[IMPORTANT] A ce stade, je fais une déviation par rapport à l'installation officielle rancher. En effet, pour l'installation de `cert-manager` j'ai préféré me référer à https://cert-manager.io/docs/installation/kubernetes/

Donc ne pas appliquer les comdandes suivantes :

```
# kubectl apply -f
https://raw.githubusercontent.com/jetstack/cert-manager/release-0.12/deploy/
manifests/00-crds.yaml
# helm install cert-manager jetstack/cert-manager --namespace cert-manager -
-version v0.12.0
```

On utilise plutot la méthode d'installation proposée par `cert-manager` *(la version peut être à mettre à jour)* :

```
helm install cert-manager jetstack/cert-manager --namespace cert-manager --
version v0.15.0 --set installCRDs=true
```

Et on vérifie que tout fonctionne bien comme cela par exemple :

```
kubectl get pods --namespace cert-manager
```

On continue avec l'install de rancher *(ici avec option letsEncrypt, à adapter avec vos infos à vous)* :

```
helm install rancher rancher-stable/rancher --namespace cattle-system --set
```

```
hostname=k8s.montaigu.io --set ingress.tls.source=letsEncrypt --set
letsEncrypt.email=alban.montaigu@gmail.com
```

Et on vérifie l'avancement du déploiement / que tout se passe bien avant de passer à la suite :

```
kubectl -n cattle-system rollout status deploy/rancher
kubectl -n cattle-system get deploy rancher
```

Il n'est pas inutile non plus de vérifier les certificats `letsEncrypt` :

```
kubectl -n cattle-system describe certificate
kubectl -n cattle-system describe issuer
```

[INFO] Au cas ou vous auriez besoin de reset le password rancher (parce que mal noté ou autre), utilisez la commande ci-dessous pour reset le mot de passe.

La commande en question pour reset le password :

```
kubectl  -n cattle-system exec $(kubectl  -n cattle-system get pods -l
app=rancher | grep '1/1' | head -1 | awk '{ print $1 }') -- reset-password
```

Installation de wordpress

Installation du backend applicatif

La documentation / le contenu de référence :

- https://docs.bitnami.com/tutorials/deploy-custom-wordpress-production-helm/#step-1-define-configuration-values-for-the-bitnami-wordpress-helm-chart
- https://github.com/bitnami/charts/tree/master/bitnami/wordpress
- https://bitnami.com/stack/wordpress/helm
- https://github.com/bitnami/charts/tree/master/bitnami/wordpress/#installing-the-chart

J'ai retenu le **chart bitnami** plutôt que celui proposé par rancher car il est plus récent. Et de toutes façons rancher s'inspire du chart bitnami.

Donc a ce stade go dans la gestion des catalogues de rancher, ajouter la référence vers les charts bitnami et utiliser à la fin la fonction de launch app en choisissant `wordpress`.

Ma configuration `helm` wordpress se base sur https://github.com/bitnami/charts/blob/master/bitnami/wordpress/values.yaml

Ajout catalogue chart bitnami dans rancher :

- Add catalog
- Global catalog
- Utiliser cette adresse https://charts.bitnami.com/bitnami

A noter qu'il faudra passer les bonnes valeurs de help dans le web *(soit en updloadant le fichier yaml,*

soit en le copian collant). D'ailleurs pourquoi via le web rancher plutôt qu'une simple commande helm ? Pour ma part parceque les applications déployées via rancher on des choses faites automatiquement *(un namespace, des éléments regroupement dans le web, des options d'upgrade, etc)*. Maintenant à chacun ses goûts :)

Ma configuration :

wordpress_helm_values.yaml

```
## Global Docker image parameters
## Please, note that this will override the image parameters, including
dependencies, configured to use the global value
## Current available global Docker image parameters: imageRegistry and
imagePullSecrets
##

## User of the application
## ref:
https://github.com/bitnami/bitnami-docker-wordpress#environment-variabl
es
##
wordpressUsername: MON_USER

## Application password
## Defaults to a random 10-character alphanumeric string if not set
## ref:
https://github.com/bitnami/bitnami-docker-wordpress#environment-variabl
es
##
wordpressPassword: MON_PASSWORD

## Admin email
## ref:
https://github.com/bitnami/bitnami-docker-wordpress#environment-variabl
es
##
wordpressEmail: MON_EMAIL

## First name
## ref:
https://github.com/bitnami/bitnami-docker-wordpress#environment-variabl
es
##
wordpressFirstName: Alban

## Last name
## ref:
https://github.com/bitnami/bitnami-docker-wordpress#environment-variabl
es
##
wordpressLastName: Montaigu
```

```
## Blog name
## ref:
https://github.com/bitnami/bitnami-docker-wordpress#environment-variabl
es
##
wordpressBlogName: Alban's Home

## Set up update strategy for wordpress installation. Set to Recreate
if you use persistent volume that cannot be mounted by more than one
pods to makesure the pods is destroyed first.
## ref:
https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#s
trategy
## Example:
## updateStrategy:
##  type: RollingUpdate
##  rollingUpdate:
##    maxSurge: 25%
##    maxUnavailable: 25%
updateStrategy:
  type: Recreate

## Kubernetes configuration
## For minikube, set this to NodePort, elsewhere use LoadBalancer or
ClusterIP
##
service:
  type: ClusterIP
  ## HTTP Port
  ##

  ## Enable client source IP preservation
  ## ref
http://kubernetes.io/docs/tasks/access-application-cluster/create-exter
nal-load-balancer/#preserving-the-client-source-ip
  ##
  externalTrafficPolicy: Local

## Configure the ingress resource that allows you to access the
## WordPress installation. Set up the URL
## ref: http://kubernetes.io/docs/user-guide/ingress/
##
ingress:
  ## Set to true to enable ingress record generation
  ##
  enabled: false

  ## Set this to true in order to add the corresponding annotations for
cert-manager
  ##
```

```
  certManager: false

## Enable persistence using Persistent Volume Claims
## ref: http://kubernetes.io/docs/user-guide/persistent-volumes/
##
persistence:
  enabled: true
  ## wordpress data Persistent Volume Storage Class
  ## If defined, storageClassName: <storageClass>
  ## If set to "-", storageClassName: "", which disables dynamic
provisioning
  ## If undefined (the default) or set to null, no storageClassName
spec is
  ##   set, choosing the default provisioner.  (gp2 on AWS, standard on
  ##   GKE, AWS & OpenStack)
  ##
  storageClass: "scw-bssd-retain"
  ##
  ## If you want to reuse an existing claim, you can pass the name of
the PVC using
  ## the existingClaim variable
  # existingClaim: your-claim
  accessMode: ReadWriteOnce
  size: 10Gi

##
## MariaDB chart configuration
##
##
https://github.com/bitnami/charts/blob/master/bitnami/mariadb/values.ya
ml
##
mariadb:
  ## Whether to deploy a mariadb server to satisfy the applications
database requirements. To use an external database set this to false
and configure the externalDatabase parameters
  enabled: true

  ## Enable persistence using Persistent Volume Claims
  ## ref: http://kubernetes.io/docs/user-guide/persistent-volumes/
  ##
  master:
    persistence:
      enabled: true
      ## mariadb data Persistent Volume Storage Class
      ## If defined, storageClassName: <storageClass>
      ## If set to "-", storageClassName: "", which disables dynamic
provisioning
      ## If undefined (the default) or set to null, no storageClassName
spec is
```

```
      ##   set, choosing the default provisioner.  (gp2 on AWS,
standard on
      ##   GKE, AWS & OpenStack)
      ##
      storageClass: "scw-bssd-retain"
      accessModes:
        - ReadWriteOnce
      size: 8Gi
```

[IMPORTANT] `clusterIP` est important ici. Cela permet de déployer le backend accessible via ip interne dans le cluster kubernetes. C'est à cet endroit qu'on précise qu'on ne veut pas d'autre `ingress` ou de `loadbalancer` ou de `nodeport` car on gère cela d'un autre côté.

Configurer le (seul et unique) ingress pour causer avec le backend wordpress

Cette documentation comporte des informations qui peuvent être utiles : https://www.digitalocean.com/community/tutorials/how-to-set-up-an-nginx-ingress-with-cert-manager-on-digitalocean-kubernetes

Et pour ceux qui se posent la question **oui il est possible d'avoir un seul ingress qui envoie vers plusieurs backend dans plusieurs namespaces** : https://itnext.io/save-on-your-aws-bill-with-kubernetes-ingress-148214a79dcb

On commence déjà par préparer la configuration nécessaire pour la partie `letsEncrypt` *(les issuers de certificats, 1 pour le test et 1 pour la prod)*.

On crée le fichier de configuration :

```
nano staging_issuer.yaml
```

Avec son contenu :

staging_issuer.yaml

```
apiVersion: cert-manager.io/v1alpha2
kind: ClusterIssuer
metadata:
 name: letsencrypt-staging
 namespace: cert-manager
spec:
 acme:
   # The ACME server URL
   server: https://acme-staging-v02.api.letsencrypt.org/directory
   # Email address used for ACME registration
   email: MON_MAIL
   # Name of a secret used to store the ACME account private key
   privateKeySecretRef:
     name: letsencrypt-staging
   # Enable the HTTP-01 challenge provider
   solvers:
```

```
    - http01:
        ingress:
          class:  nginx
```

On déploie l'issuer pour les tests :

```
kubectl create -f staging_issuer.yaml
```

On crée ensuite la configuration pour la production :

```
nano prod_issuer.yaml
```

Le contenu du fichier :

prod_issuer.yaml

```
apiVersion: cert-manager.io/v1alpha2
kind: ClusterIssuer
metadata:
  name: letsencrypt-prod
  namespace: cert-manager
spec:
  acme:
    # The ACME server URL
    server: https://acme-v02.api.letsencrypt.org/directory
    # Email address used for ACME registration
    email: MON_MAIL
    # Name of a secret used to store the ACME account private key
    privateKeySecretRef:
      name: letsencrypt-prod
    # Enable the HTTP-01 challenge provider
    solvers:
    - http01:
        ingress:
          class: nginx
```

Et enfin, une fois la configuration de base let'sencrypt faite, on passe à la configuration ingress pour notre wordpress :

```
nano wordpress_ingress.yaml
```

Le contenu du fichier :

wordpress_ingress.yaml

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
```

```
  name: wordpress-ingress
  namespace: wordpress
  annotations:
    kubernetes.io/ingress.class: "nginx"
    cert-manager.io/cluster-issuer: "letsencrypt-prod"
spec:
  tls:
  - hosts:
    - alban.montaigu.io
    secretName: wordpress-tls
  rules:
  - host: alban.montaigu.io
    http:
      paths:
      - backend:
          serviceName: wordpress
          servicePort: 80
```

Maintenant que tous les fichiers de configuration sont prets, on passe au **deploiement**.

Le letsEncrypt de test *(même si on ne l'utilise pas précisément là, il peut être bon pour tester si besoin)* :

```
kubectl create -f staging_issuer.yaml
```

Le déploiement du `letsEncrypt` de prod et de la configuration ingress de wordpress *(avec certificat letsEncrypt)* :

```
kubectl create -f prod_issuer.yaml
kubectl apply -f  wordpress_ingress.yaml
```

Pour vérifier que la génération de certificat fonctionne :

```
kubectl describe certificate wordpress-tls
```

Les commandes ci-dessous peuvent être utiles pour des fins de début. Elles ne sont pas nécessaires à l'installation.

Avoir des infos sur le ingress wordpress :

```
kubectl get ing -n wordpress
```

Des infos de manière plus détaillée :

```
kubectl describe ing wordpress-ingress -n wordpress
```

Configuration wordpress

[IMPORTANT] Je me suis rendu compte que les fonctions *import / export* de wordpress ne sont pas

vraiment dédiées au backup. En effet, pour les médias *(images and co)* les données sont conservées sur le site d'origine, et au mieux c'est lors de l'import que les images d'origine sont téléchargées.

Si vous avez eu le malheur de démonter un peu vite votre ancien site, l'import sera en erreur sans aucun message aidant à comprendre. Oui j'ai compris après moultes tests et échecs...

[IMPORTANT] Dernier point important concernant l'import sur les médias, images par exemple. Il est possible que lors de l'import wordpress les range dans des sous dossiers organisés par date *(alors que ce n'était pas le cas dans la configuration d'origine)*. Autrement dit toutes les inclusions d'images dans vos articles seront cassées.

Dans mon cas je m'en suis sorti en re téléchargeant les images d'origine dans `wp-content/uploads` quitte à dupliquer les images *(ces dernières n'apparaissent pas dans le gestionnaire de média)*. Il y a sans doute quelque chose de mieux à faire mais dans mon cas c'est suffisant.

Sinon, en vrac, les quelques choses que j'ai du faire *(à remettre en forme un jour)* :

- `Export` dans le site d'origine wordpress (via les fonction admin) des articles & média
- `Import` dans le nouveau site des media puis articles séparés en associant l'ancien auteur au nouveau
- Rechercher remplacer les url ancien site nouveau site dans les articles et le contenu avec ce plugin https://fr.wordpress.org/plugins/better-search-replace/
- Changer la langue en Français
- Changer le slogan du blog
- Configuration ecriture URL à renregistrer
- Configuration du profil alban montaigu *(nom prénom, bio si nécessaire)*
- Changer les widgets : nuage de tag
- Widgets sites que j'aime bien et mes sites
- Page Piracy policy à configurer
- Régler les commentaires *(autoriser ou non selon l'envie du moment)*
- Plus généralement faire un tour général des configurations

[IMPORTANT] Dans l'url du site wordpress, il faudrait corriger le http en https. En effet, notre configuration est en https mais les liens wordpress ne le semblent pas toujours et cela génère des bugs. Néanmoins dans l'image wordpress bitnami, cette choses et plus ou moins auto détectée / configurée en dur donc non changeable dans l'administration.

Dans tous les cas j'ai essayé de le corriger mais cela à complètement fait planter mon site. Donc je me suis arrêté là.

[IMPORTANT] Attention en bidoullant la conf wordpress. A la moindre erreur de config, si PHP remonte une erreur, cela va générer des **erreurs dans le monitoring kubernetes** qui va commencer **rebooter non stop votre wordpress**. Cela m'à souvent obligé à réinstaller car impossible de récupérer le conteneur. En effet, modifier les fichiers dans un conteneur qui reboot est un challenge et encore plus dans kubernetes.

Au mieux, vous pouvez essayer de mettre l'orchestration kubernetes en pause *(pour éviter le reboot en boucle)* et avec un peu de chance le conteneur continuera à vivre suffisamment longtemps pour que vous puissiez corriger l'erreur.

Configuration du theme

En vrac pour le thème :

- Choix du thème `twenty seventeen`
- Suppression des autres themes
- Jeu de couleurs foncées
- Image personnalisée *(mon cher petit robot)*

Avec quelques modification CSS pour une largeur page adequate *(j'aime bien les sites qui prennent toute la page)*. A mettre dans le **custom css**.

Le contenu du custom CSS à mettre dans la personnalisation du thème :

```
@media screen and (min-width: 48em) {
  .wrap {
    max-width: 1500px;
  }
  .has-sidebar:not(.error404) #primary {
    width: 70%;
  }
  .has-sidebar #secondary {
    width: 24%;
  }
}
```

Ma sélection d'extension sur cette base d'image bitnami :

- Activer Askimet
- **NE PAS** Installer Apocalyspe Meow *(sauf si mode proxy mais il fait planter d'autres truc comme site health)*
- Désinstaller All-in-One WP Migration
- Désinstaller AMP
- Désinstaller Bitnami Production Console Helper
- Désinstaller Hello Dolly
- Désinstaller Jetpack par WordPress.com
- Activer All in One SEO Pack *(ou pas, selon l'envie du moment)*
- Activer Simple Tags *(utile pour avoir un nuage de tag avec des tailles différentes par fréquence d'utilisation)*
- Désinstaller WP Mail SMTP
- Desinstaller wordpress importer *(Plus besoin une fois l'import fait)*<note important> **[IMPORTANT]** Les permaliens semblent sauter a chaque reboot. Visiblement il suffit de retourer dans l'admin wordpress et enregistrer à nouveau configuration permalien. Me demandez pas pourquoi, je n'ai pas cherché plus avant. </note>

La guerre pour faire fonctionner apocalype meow

[IMPORTANT] Cette partie est expérimentale et n'a pas complètement été menée à son terme.

Quelques références :

- https://elwpin.com/2019/03/14/quick-fix-for-cloudflare-and-php-remote_addr-ip-detector/
- https://github.com/bitnami/bitnami-docker-wordpress/issues/178

C'est un des trucs qui a contribué à la fin à me faire changer complètement d'infra de serveur.

Il se trouve que pour bien fonctionner, ce cher plugin à besoin de **l'ip réelle source du client**. Logique, sauf que comme évoqué ce n'est pas si simple à configurer.

En effet, comme le plugin a des fonctions de ban d'ip source si tentative de bruteforce sur l'admin, il lui faut bien l'information quelque part.

Sauf que voilà, en admetant que vous avez configuré votre cluster kubernetes correctement, il se trouve que dans les options *(en tout cas pour moi à ce moment)*, le plugin **ne permet pas de sélectionner le header http qui continent l'ip source**. Plus exactement la configuration existe, mais à chaque enregistrement on revient à la valeur de base, la mauvaise évidemment.

Les commandes suivantes sont donc une tentative de hack pour mettre les bonnes valeurs dans le bon champ puisque le plugin ne permet pas le choix.

L'option de modifier directement la configuration en base peut exister mais bizarrement cela n'a pas fonctionné. Comme si les champs étaient en **lecture seule**. C'est peut être la root cause du problème mais je n'ai pas mené l'investigation jusqu'au bout.

On se connecte au conteneur et on modifie les fichiers *(le nom du conteneur va forcément changer pour vous)* :

```
kubectl exec -ti -n wordpress wordpress-5498fdfdd6-b5nxw -- bash
cp wp-config.php wp-config.php.back
chmod +w wp-config.php.back
echo "">> wp-config.php.back
echo "// Fix the REMOTE_ADDR value (since not possible to simply fix it in
httpd with docker image)">> wp-config.php.back
echo "if (isset(\$_SERVER[\"HTTP_X_REAL_IP\"])) {">> wp-config.php.back
echo "\$_SERVER[\"REMOTE_ADDR\"] = \$_SERVER[\"HTTP_X_REAL_IP\"];">> wp-
config.php.back3
echo "}">> wp-config.php.back
chmod -w wp-config.php.back
```

Une fois que tout est bon *(après vérification donc)* on peut déployer la nouvelle configuration :

```
cp cp wp-config.php.back wp-config.php
```

La guerre pour faire fonctionner W3 Total Cache

Visiblement notre ami rale après installation. De ce que j'ai trouvé c'est lié à un problème de droit :

```
chmod 775 wp-config.php
```

Une référence qui m'a aidé : https://github.com/bitnami/bitnami-docker-wordpress/issues/203

Changement DNS vivihome.net

Comme il s'agit d'une migration, penser à mettre à jour les DNS de l'ancien site :

- Supprimer les anciennes défiitions
- Ajouter une redirection visible vers une adresse web *(pas une ip donc)* avec un code http **redirection permanente**

Commandes utiles en vrac

Excecuter un petit shell dans son conteneur wordpress à des fin de debug par exemple :

```
kubectl exec -ti -n wordpress wordpress-5498fdfdd6-hrhfp /bin/bash
```

Exectuer un conteneur oneshot pour avoir shell :

```
kubectl run my-shell --rm -i --tty --image ubuntu -- bash
```

Installation Jira & Confluence

On le fait toujours dans rancher via un helm chart bien choisi qu'on ajoute dans les catalogues rancher. Cela permet ensuite de lancer une application pour installation *(comme pour wordpress)*.

Ici on ajoute le catalogue helm mox dans rancher *(via gestion des catalogues, ajout d'un catalogue helm 3)* en utilisant cette URL : https://helm.mox.sh.

Quelques documents / liens de référence utilisés :

- https://hub.helm.sh/charts/mox/jira-software
- https://hub.helm.sh/charts/mox/confluence-server
- https://github.com/javimox/helm-charts
- https://github.com/javimox/helm-charts/releases
- https://github.com/javimox/helm-charts/tree/master/charts

Comme d'habitude pour avoir l'ip source du client qui redescend jusqu'à l'application *(attention ce n'est pas le seul critère hélas)*, ne pas oublier de mettre le `externalTraficPolicy` à `Local`.

Le drame avec les ressources :

Ne pas oublier que les déploiements sont fait à l'origine sur des nodes `DEV-M` avec de mémoire 4Go de RAM. Sachant que JIRA dans la config helm en requiert 2G de réservés. Ces besoins assez importants qui devront être partagé avec les éventuels autres conteneurs déployés sur le node qui sera élu. Or, la litérature sur le net doit normalement le confirmer, java a du mal (moins avec les versions récentes) avec les conteneurs notamment sur la gestion de la mémoire. Et la le police kubernetes arrive avec du `OOMkill` ou autre.

C'est un peu ce qui a du m'arriver à ce stade, j'ai eu la joie d'avoir des

https://sysdig.com/blog/debug-kubernetes-crashloopbackoff/ ou autres joyeusetées.

Autre problème à corriger le `liveness` et le `readiness` qu'il faut adapter car au premier démarrage jira et confluence mettent un temps infini a démarrer et se préparer (phase d'install). Donc par défaut kubernetes perds patience et des timeout se déclenchent pour voir finalement son conteneur redémarré par kubernetes sans que le démarrage soit terminé.

J'ai pu m'en sortir avec des adaptations de readiness et liveness plus **ajout d'un nouveau pool de nodes avec des conf à 8Go de RAM* sans oublier** des regles d'affinité **pour que les conteneurs jira et confluence aillent sur les nodes les plus généreux en RAM d'une part et d'autre part qu'ils ne soient pas déployés ensemble sur le même noeud (et ainsi se faire la guerre sur les ressources). C'est là que je me suis arrêté et que j'ai renoncé pour être franc. Cela tournait mais pas de manière si rapide d'une part et d'autre part mon** porte monaie commencait à pleurer** car j'avais ajouté 2 noeuds à 8Go de RAM qui valent bien plus cher que mes noeuds de départ (que je devais conserver pour mes autres déploiements). Bref, trop cher, trop compliqué toujours pas fini, tant pis...

Conclusion

Alors heureux ? Non pour être clair. J'ai passé un temps bien trop long à tatonner pour un résultat complexe, difficile à maintenir et en plus cher en hébergement.

J'ai du tout recommencer un nombre incalculable de fois parfois pour une erreur idiote. Donc oui c'est sexy, oui j'ai écrit ce guide pour ne pas perdre les connaissances acquises. Oui j'espère que cela pourra être utile à d'autres. Mais en vrai moi, je vais trouver une autre solution.

Mes feedbacks plus détaillés sur mon blog :

- https://alban.montaigu.io/2020/05/26/premiers-retours-sur-la-nouvelle-infra-alban-montaigu-io/
- https://alban.montaigu.io/2020/05/26/second-retour-sur-la-nouvelle-infra-alban-montaigu-io/
- https://alban.montaigu.io/2020/05/27/retour-en-2013-2014-nouvelle-infra-montaigu-io/