
Christophe Pichaud
Lead Architect
Microsoft chez Infeeny
christophep@cpixxi.com |
www.windowscpp.com


Julie Lacognata
Infeeny
julie.lacognata@infeeny.com


Kevin Ansard
Infeeny
kevin.ansard@infeeny.com

Déploiement d'une application API REST sur Docker / Kubernetes sous Linux

Partie 2

Dans cet article, nous allons réaliser une application Web en .NET Core 3.1 avec un déploiement Kubernetes sous un environnement Linux. Le but de cette application est de vous montrer comment utiliser et déployer avec Kubernetes. Mais surtout de vous expliquer le concept même de Kubernetes et pourquoi ce sont, avec Docker, des technologies phares du moment.

Passons maintenant à Kubernetes via Microk8s.

Kubernetes, qu'est-ce que c'est ?

Pour rappel, Kubernetes n'est ni plus ni moins qu'un orchestrateur. C'est un ensemble de services réseaux qui permet de lancer des pods Docker en cluster avec une gestion de load-balancing. Pour résumer, il gère les conteneurs et les pods. On va utiliser Kubernetes qui lui-même utilise des objets pour décrire l'état souhaité de votre cluster (c'est là où s'exécute le code). Il va en outre, nous permettre de définir les applications souhaitées, les processus, le nombre de replicas, le volume et plein d'autres paramètres. Voici un petit lexique Kubernetes :

Objets

Une fois les états de vos objets déclarés, Kubernetes fait appel au Control Plan, qui lui-même va changer les états de vos objets de base en état souhaité. Plus simplement, le Control Plan est un ensemble de processus dans votre cluster. Comme nous l'avons dit précédemment, Kubernetes fonctionne avec des objets et va définir un ensemble de choses. Néanmoins, il y a des objets qui sont inclus de base :

Pods

Les pods correspondent au processus en cours d'exécution et encapsulent un ou des conteneurs applicatifs. Ce sont des instances uniques, cela signifie qu'ils possèdent :

- Une IP unique ;
- Un fichier qui indique comment le conteneur doit être exécuté ;
- Des ressources de stockage.

Ce qu'il faut retenir, si nous devions résumer un peu tout ça, c'est que les pods peuvent correspondre à une application ayant sa propre mémoire, sa propre IP. Néanmoins, un pod a une durée de vie définie et ne sera pas en mesure de se relancer automatiquement de lui-même.

Replica

Correspond au nombre de répliques d'un même pod. Il ne faut pas oublier que chaque pod a une IP unique, une mémoire unique, etc. Même si, ils partent de la même encapsulation de conteneurs.

Services / Micro-services

Ils vont définir la logique entre les pods (entre eux) et la politique d'accès à ceux-ci. À quoi ça sert ? Comme nous l'avons énoncé auparavant, les pods ont des IP uniques, mais imaginons une application avec plusieurs instances (replicas de pods) de backend et qui ont chacune une IP, cela peut poser problème. La principale question étant : comment peut-on y accéder si les IP changent ? C'est pour cette raison que les services sont existants, car ils vont simplifier l'accès entre les différents pods et leur politique d'accès.

Volume

Maintenant que nous avons expliqué ce que sont les pods et les services, il reste un problème majeur : nos pods peuvent mourir et donc perdre leurs données. Ou tout simplement, nous avons également besoin, si un pod possède plusieurs conteneurs de partager la mémoire entre eux. Donc, comment allons-nous partager cette mémoire sachant que chaque instance possède sa propre mémoire indépendante des autres ? C'est pourquoi le Volume est une sorte de mémoire partagée entre les conteneurs d'un même pod. Concrètement, le Volume, initialement, n'est qu'un dossier accessible aux différents conteneurs. Un Volume doit être défini, ainsi que sa politique d'accès.

Objets : Contrôleurs

Nous allons maintenant vous présenter les différents contrôleurs que l'on retrouve sur Kubernetes. Un contrôleur n'est ni plus ni moins qu'un objet Kubernetes, mais qui a la particularité d'ajouter une fonctionnalité. On peut résumer ça en disant que ce sont des « super objets » !

ReplicaSet

Un ReplicaSet a pour but de définir et de maintenir un ensemble de pods. Pour ce faire, nous utilisons le sélecteur pour identifier les pods, que le contrôleur va posséder. Imaginons que nous demanderons à un pod de se répliquer 3 fois. Si un de ces pods meurt, le ReplicaSet va en recréer un. Si nous ne voulons que 2 répliques finalement, le contrôleur ReplicaSet va donc supprimer un des pods. Il a donc pour but de rendre stable cet ensemble de pods.

Deployments

Le Deployment est une sorte de gestionnaire qui va nous permettre de changer l'état de nos pods ou de nos ReplicaSet. Autrement dit, le nombre de replicas voulu, ou autre statut de ceux-ci (comme par exemple l'arrêt des ReplicaSet, le stop du Deployment, l'augmentation de la charge traitable, etc.). En outre, ce gestionnaire vous permet de modifier les statuts et parfois même de revenir à des versions antérieures de déploiement.

DaemonSet

Le DaemonSet permet de gérer les nœuds (ou n'odes), plus précisément les pods qui doivent être exécutés dans ces nœuds. Si des nœuds sont ajoutés, le contrôleur va alors ajouter les pods demandés. En opposition, s'ils sont supprimés (les nœuds), les pods qui étaient utilisés vont être récupérés par le DaemonSet. La suppression de ce contrôleur va nettoyer les pods qui avaient été ajoutés.

StatefulSet

Le contrôleur StatefulSet gère le déploiement et la mise à l'échelle d'un ensemble de pods. Le StatefulSet est assez similaire au Deployment (le contrôleur), car il y a cette notion de gestion des pods, mais contrairement au Deployment il y a le terme de « sticky identity » (en français « identité collante ») pour chacun de ses pods. Ce qui signifie que les pods sont créés à partir de la même spécification, mais qu'ils ne sont en aucun cas interchangeables : chacun d'entre eux possède un ID unique et permanent. Le StatefulSet est une sorte de Deployment statique et que par conséquent il possède des limites.

Jobs

Les Jobs permettent de définir un cycle de vie, on va donc leur définir un nombre de réussites voulues. Ce qu'on veut dire par là, c'est que le contrôleur va créer des pods et va « regarder » s'ils se terminent avec succès. Une fois le nombre de succès atteint, la tâche est terminée. Le Job peut donc être supprimé, ce qui nettoiera les pods créés.

Notions d'architectures

Nœuds

Les nœuds peuvent être assimilés à des machines virtuelles ou physiques. Chacun de ces nœuds possède un ou plusieurs services qui sont nécessaires à la gestion des pods. Les nœuds sont gérés par le composant **master**, plus précisément par le contrôleur Node (Node Controller en anglais, NDLR.). Ce contrôleur a plusieurs rôles, nous allons les lister ci-dessous :

- Il tient à jour la liste interne de ses nœuds ;
- La surveillance de la santé des nœuds (qui le surveille toutes les ***n*** secondes, ***n*** étant définit au préalable par nos soins) ;
- Affectation d'un bloc CITR (d'un **master**).

Mapping de port

Le mapping de port, comme son nom l'indique, est censé « mapper » les ports. Ce que l'on entend par « mapper » est la possibilité d'attribuer un port à un objet spécifique ainsi que son adresse IP. Dans le cas de Kubernetes, les Pods (objet) peuvent rechercher les services à l'aide de leur namespace. Chacun de ces services se charge quant à lui d'attribuer les IP et les ports à chaque pods. Un service peut avoir défini plusieurs ports.

```
spec:
    selector:
        app: MyApp
    ports:
        - name: http
          protocol; TCP
          port: 80
          targetPort: 9376
        - name: https
          protocol: TCP
          port: 443
          targetPort: 9377
```

Comme nous pouvons le voir ci-dessus, nous définissons plusieurs ports. Nous pouvons définir manuellement les IP de nos pods, en sachant qu'il vous faudra donc toujours définir un service (ou DNS) qui se chargera de l'attribution des ports et des IP. Pour ce faire, les pods auront besoin des namespaces pour accéder au service DNS (ou à d'autres services).

Mise en place de Kubernetes

(cf. Installation Kubernetes)

La commande :

```
sudo microk8s.enable dns dashboard registry
```

Permet de créer un service avec le nom donné en local. Plus précisément, un registry est une sorte de « repository » d'images. Grossièrement on peut dire que c'est un espace de stockage d'images.

Nous allons, à partir de maintenant configurer le portail Kubernetes. Celui-ci est utile notamment pour vérifier les « statuts » de vos services :

```
sudo microk8s.enable dns dashboard registry ingress
```

Cette commande ci-dessous, va nous permettre de créer une sorte de contrôler pour gérer les différents services.

```
sudo microk8s.kubectl proxy --accept-hosts=.* --adress=0.0.0.0 &
```

Cette commande nous permet d'accepter n'importe quel port et adresse.

```
sudo microk8s.kubectl -n kube-system edit deploy kubernetes-dashboard -o yaml
```

La dernière commande nous permet d'ouvrir VIM pour éditer un fichier de configuration du portail.

Dans la section spec / container / args nous allons ajouter la ligne : --enable-skip-login

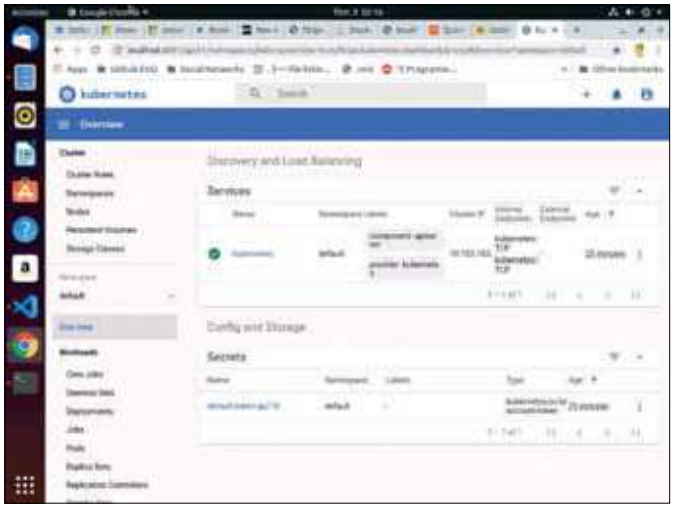
La séquence de touche est la suivante :

- ESC
- I
- --enable-skip-login
- ESC
- :w
- :q

Va vous permettre de générer ça :

```
spec:
    containers:
    - args:
      - --auto-generate-certificates
      - --namespace=kube-system
      - --enable-skip-login
```

Nous pouvons enfin accéder au portail via l'URL suivante :
http://localhost:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/ 6



6

Nous allons pouvoir utiliser Kubernetes à partir de là. Ce qu'il faut savoir c'est que pour déployer une application sur Kubernetes, nous allons utiliser Helm. Helm, pour rappel, est un gestionnaire de paquets pour Kubernetes comme annoncé plus haut lors de l'installation. Au préalable nous allons lancer la commande suivante :

```
sudo microk8s.kubectl config view --raw >~/.kube/config
```

La procédure pour faire tourner une application sous Kubernetes est la suivante : Kubernetes a besoin d'une image qu'il peut extraire lors de son utilisation. Il nous faudra donc :

- Créer notre image et l'envoyer

Nous allons d'abord builder en local sous Docker et tagger l'application (création de l'image) :

```
sudo docker build . -t localhost:32000/app1:latest
```

- Nous allons pousser l'application dans un registry locale :

```
sudo docker push localhost:32000/app1
```

- Nous allons déployer l'application via Helm (pour information, Helm va chercher une image, il est donc nécessaire de charger notre image au préalable, pour rappel) !

À la fin de toutes ces étapes, voilà l'architecture de vos fichiers que vous devez retrouver dans le dossier kubernetes_project : 7

7

Déploiement sous Helm

Pour réaliser le déploiement, nous avons besoin des 4 fichiers suivants dans le dossier **chart** :

- ./chart
  - charts
  - Chart.yaml
  - templates
    - deployment.yaml
    - service.yaml
  - values.yaml

Fichier Chart.yaml

```
name: aspnet3-demo
version: 1.0.0
```

Fichier values.yaml

```
environment: development

apphost: k8s

label:
 name: aspnet3core

container:
 name: aspnet3
 pullPolicy: IfNotPresent
 image: localhost:32000/aspnet3k8s
 tag: latest
 port: 80
replicas: 3

service:
 port: 8888
 #type: ClusterIP
 type: NodePort
```

Fichier templates/services.yaml

```
apiVersion: v1
kind: Service
metadata:
 name: {{ .Release.Name }}-service
 labels:
  app: {{ .Values.label.name }}
spec:
 ports:
 - port: {{ .Values.service.port}}
  protocol: TCP
  targetPort: {{ .Values.container.port }}
 selector:
  app: {{ .Values.label.name }}
 type: {{ .Values.service.type }}
```

Fichier templates/deployment.yaml

Code complet sur programmez.com & github

Comme vous pouvez le constater, c'est le fichier values.yaml qui compte le plus :

- Champs container / image : nom de l'image docker dans la registry locale ;
- Champs replicas : c'est le nombre de pods Docker à déployer ;
- Champs service / type : NodePort indique que le service est accessible depuis l'extérieur du Cluster.

Une fois que les fichiers sont prêts, on peut déployer à l'aide de la commande :

```
helm install aspnet3release ./chart
```

La commande ***install*** permet de déployer l'image et a3 correspond au nom donné au déploiement. Maintenant, afin de vérifier que les services se sont correctement déployés, nous pouvons exécuter la commande suivante :

```
sudo microk8s.kubectl get all
```

La réponse à cette commande est la suivante :

```
NAME                                        READY   STATUS    RESTARTS   AGE
pod/aspnet3release-deployment-9b94ff7c8-f7xfh   1/1     Running   17         38d
pod/aspnet3release-deployment-9b94ff7c8-fjwsq   1/1     Running   18         38d
pod/aspnet3release-deployment-9b94ff7c8-xn6sx   1/1     Running   20         38d
pod/kubernetes-bootcamp-69fbc6f4cf-zzc77        1/1     Running   22         39d

NAME                        TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
service/aspnet3release-service   NodePort    10.152.183.30    <none>        8888:31404/TCP   38d
service/kubernetes-bootcamp      NodePort    10.152.183.26    <none>        8080:32667/TCP   39d

NAME                                   READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/aspnet3release-deployment   3/3     3            3           38d
deployment.apps/kubernetes-bootcamp         1/1     1            1           39d

NAME                                              DESIRED   CURRENT   READY   AGE
```

```
replicaset.apps/aspnet3release-deployment-9b94ff7c8   3     3     3     38d
replicaset.apps/kubernetes-bootcamp-69fbc6f4cf        1     1     1     39d
```

Si vous souhaitez affiner votre recherche, vous pouvez modifier la commande ci-dessus en indiquant le sélecteur (sudo microk8s.kubectl get all –selector app=aspnet3release). Avec la commande « ip a », on récupère l'adresse IP de la machine Linux : 172.23.58.135.

Pour vérifier que vos pods sont bien exécutés (état RUNNING), tapez la commande :

```
sudo microk8s.kubectl get pods
```

La réponse à cette commande est la suivante :

```
NAME                                   READY   STATUS    RESTARTS   AGE
aspnet3release-deployment-9b94ff7c8-f7xfh   1/1     Running   17         38d
aspnet3release-deployment-9b94ff7c8-fjwsq   1/1     Running   18         38d
aspnet3release-deployment-9b94ff7c8-xn6sx   1/1     Running   20         38d
kubernetes-bootcamp-69fbc6f4cf-zzc77        1/1     Running   22         39d
```

Il est important que vos pods soient en état RUNNING comme les 3 premières lignes, qui correspondent à nos trois réplicas. Nous allons maintenant vérifier que nos services sont correctement exécutés :

```
sudo microk8s.kubectl get services
```

La réponse à cette commande est la suivante :

```
NAME                  TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
aspnet3release-service   NodePort    10.152.183.30    <none>        8888:31404/TCP   38d
kubernetes-bootcamp   NodePort    10.152.183.26    <none>        8080:32667/TCP   39d
```

8

En ce qui concerne le mapping de port, nous vous invitons à retourner voir la section Kubernetes / Notions d'architectures / Mapping de port.

On consulte le portail sur une autre machine :

http://172.23.58.135:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/#/workloads?namespace=default

Le portail permet de visualiser les services, les pods et les autres ressources Kubernetes.

Sur la ligne service on repère que le port est mappé comme suit : 8888 :31404 : c'est l'adresse externe du cluster :

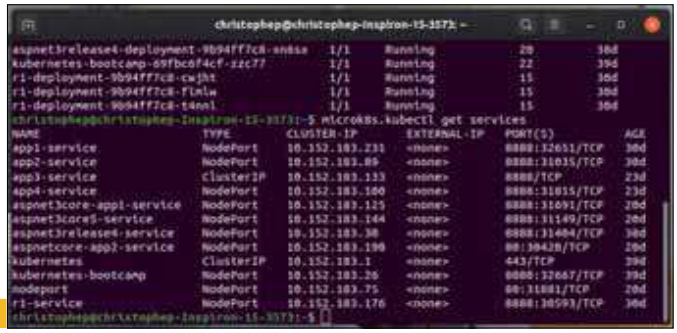
Lançons un browser depuis une autre machine allons sur

http://christophep-inspiron-15-3573:31404/WeatherForecast 9

Petites commandes bien utiles

Si vous souhaitez savoir à quels nœuds appartiennent vos différents services ou vos pods, nous vous conseillons d'exécuter la commande :

```
sudo microk8s.kubectl get nodes
```



8

La réponse à cette commande est la suivante :

```
NAME                 STATUS   ROLES    AGE   VERSION
christophep-inspiron-15-3573   Ready    <none>   39d   v1.17.3
```

Ici nous pouvons voir que nous n'avons qu'un seul node et qu'il correspond à notre machine virtuelle.

Utilisation du Linux depuis un PC Windows

Avec SSH, on se connecte à distance au PC Linux via Windows Subsystem for Linux v2 (WSL v2) : c'est une solution. Le confort ultime, c'est d'utiliser VS Code en mode SSH : on se connecte au PC Linux en donnant un user/password et on dispose d'un terminal en bas de fenêtre. Voilà à quoi cela ressemble : 10

Ce n'est pas forcément trivial de prime abord pour un développeur habitué à l'environnement Windows. Si vous souhaitez continuer de coder sous Linux, nous vous conseillons d'utiliser les méthodes indiquées plus haut.

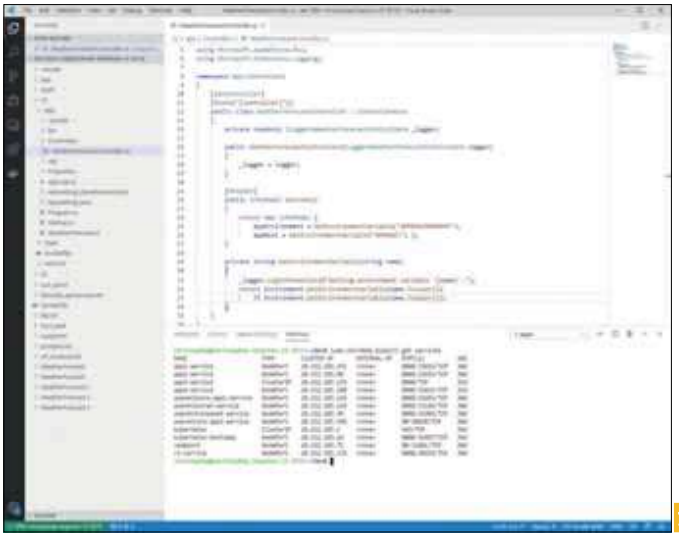
Trucs et astuces pas chers

Un PC pour Linux pas cher ? Allez chez Dell, Cdiscount ou Amazon… Cherchez une machine Pentium avec 8 Go de RAM et un disque SATA de 1 To avec une distribution Linux. Vous économisez la licence Windows. Vous en aurez pour 350 € pour une belle machine. Vous achetez un disque dur SSD 250 Go à 40€ et vous l'insérez dans le portable. C'est ce que j'ai fait. Je dispose d'une machine de course pour faire tous mes développements. 8 Go sous Linux, c'est la fête ! Les processeurs Pentium sont puissants et bien moins cher que les Core iX.

Conclusion

Si on résume le concept de Kubernetes et de Docker : imaginez que Kubernetes soit le chef d'orchestre et un container Docker, le musicien. En outre, Kubernetes ne sert à gérer que l'organisation de vos micro-services / containers Docker. Et Docker quant à lui, vous permet de les créer. Kubernetes n'est pas très compliqué. Le problème c'est que les opérations se font via la ligne de commande, comme Docker. Il y a bien le portail qui permet de visualiser les informations, mais le tooling est faible en quantité.

9



10