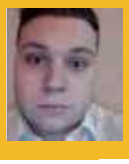

Christophe Pichaud
Lead Architect
Microsoft chez Infeeny
christophep@cpixxi.com I
www.windowscpp.com


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 1

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.

Comme nous le disions plus haut, nous entendons de plus en plus parler de Docker et de Kubernetes. Aujourd'hui nous allons nous concentrer sur l'explication des concepts de base de Kubernetes et son utilisation sous un environnement Linux.
À l'heure actuelle, vous entendez sûrement les termes AKS, Docker, etc. Néanmoins, il faut être vigilant, car sous Azure, d'une part la visibilité sur l'utilisation de Kubernetes à un aspect « magique » qui retire aux développeurs la compréhension de l'outil, mais il y a aussi l'aspect « coût » sous Azure qui n'est pas négligeable. Ce n'est pas notre propos dans cet article. C'est pour cette raison que nous allons nous tourner vers Linux. Pourquoi ce choix ? Tout simplement, car Kubernetes dans son plus simple appareil ne fonctionne pas sous Windows. Pour les utilisateurs ayant uniquement Windows, deux choix s'offrent à vous, mais dans les deux cas il vous faut la version 19.10 d'Ubuntu que vous pouvez facilement trouver et télécharger sur le site officiel. La première solution consiste à avoir une clef USB de 8Go maximum et depuis votre PC de télécharger et d'installer Rufus, qui est un petit utilitaire vous permettant de rendre la clef USB bootable avec l'ISO dessus ce qui va vous permettre de faire un dual boot. La deuxième solution quant à elle, consiste à créer et utiliser une machine virtuelle. Nous avons choisi, ici, d'utiliser la deuxième solution en utilisant Hyper-V. 1
Une fois votre système Linux mis en place, nous allons pouvoir créer un projet en .NET Core.

Mise en place d'un projet ASP .NET Core sous Linux

Pour réaliser ce tutoriel dans de bonnes conditions, voici la liste exhaustive des outils nécessaires :

- Visual Studio Code (IDE)
- SDK .NET Core 3.1
- Docker CE
- Helm 3.0.0
- Kubernetes via Microk8s

Ce qu'il faut savoir, c'est que sous Linux, les installations s'effectuent différemment que sous Windows.
Pour l'installation sous Ubuntu, il vous faut ouvrir votre Terminal, ainsi qu'avoir à taper ce qu'on appelle les commandes « super-droits utilisateurs » plus communément appelé ***sudo*** sous les OS Linux. Pour toutes installations de paquets (ou package en anglais), Ubuntu utilise la commande ***apt-get install*** ou ***snap install***. Nous allons d'abord vous expliquer comment installer les prérequis sur votre machine virtuelle.

1

Installation de Visual Studio Code

VS Code est un éditeur de code Open Source développé par Microsoft. Il est compatible avec Windows, macOS et Linux. L'installation de VS Code se fait par un simple :

```
sudo snap install code –classic
```

Nous utilisons le mot clef ***sudo*** pour faire valoir les super-droits utilisateurs, puis nous faisons appel à la commande ***snap install*** pour télécharger et installer VS Code, ici nommé ***code***.
Une fois que VS Code installé, nous allons ajouter l'extension C#. Pour se faire, dans VS Code cliquez sur « Extensions » puis dans la barre de recherche, tapez « C# » et installez l'extension. Nous vous conseillons fortement de faire ça pour vous simplifier le développement et l'agencement du code C#. Nous allons maintenant passer à l'installation du SDK .NET Core.

Installation du SDK .NET Core

Le .NET Core 3.1 est le framework préconisé par Microsoft pour faire les nouveaux projets NET. Nous allons donc installer ce SDK en faisant attention à bien sélectionner la version 3.1. Nous allons donc procéder de la manière suivante :

```
wget -q https://packages.microsoft.com/config/ubuntu/19.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
```

Le mot clef ***wget*** va nous permettre de télécharger des paquets depuis une URL précise. Suite à l'exécution de cette commande, nous allons ensuite taper la commande :

```
sudo dpkg -i packages-microsoft-prod.deb
```

Le mot clef dpkg va installer les paquets.
L'installation du SDK n'est toujours pas terminée, mais presque, il nous reste quelques commandes à lancer :

```
sudo apt-get update
sudo apt-get install apt-transport-https
```

```
sudo apt-get update
sudo apt-get install dotnet-sdk-3.1
```

La deuxième ligne permet d'installer le paquet ***transport-https***. Enfin, la quatrième ligne permet de télécharger et d'installer le framework .NET Core.

Installation de Docker CE

Docker est une plateforme de conteneurisation. Quand on parle de cette notion, on parle évidemment de conteneur. Les conteneurs ressemblent à des machines virtuelles ou seul le système d'exploitation ou OS aura été virtualisé et ils regroupent tout le nécessaire de notre application (peu importe qu'elle soit sous Linux ou sous Windows). En opposition aux machines virtuelles, qui elles, vont virtualiser le hardware dans son intégralité. Docker est donc une solution qualifiée de compacte, puissante, innovante et qui permet une scalabilité importante. Passons maintenant à son installation :

```
sudo apt install docker.io
```

Cette simplement comme vous permet l'installation de Docker CE (Community Edition, ndlr.). Maintenant, regardons comment démarrer une instance de Docker à l'aide des lignes suivantes :

```
sudo systemctl start docker
sudo systemctl enable docker
```

Installation de Helm

Maintenant que toutes les étapes précédentes sont bien mises en place, nous pouvons télécharger et installer Helm. Helm est un outil qui va vous permettre d'installer et de gérer le cycle de vie de vos applications Kubernetes. C'est un gestionnaire de paquets à déployer pour Kubernetes.

```
sudo snap install helm –classic
```

Installation de Kubernetes

Pour son installation sous Ubuntu, nous allons passer par Microk8s, qui est une version si l'on peut dire « light », mais aussi une version pour les développeurs de Kubernetes et qui va nous servir pour notre déploiement en local. Pourquoi une version « light » ? Tout simplement, car nos machines ne sont pas extensibles à l'infini, et c'est la même raison qui nous a poussés à vous faire installer VS Code qui est beaucoup plus léger que son camarade Visual Studio. Bref, revenons-en au sujet principal : Kubernetes ! C'est une plateforme Open Source qui a pour but de gérer la charge de travail des services conteneurisés. Autrement dit, on peut le visualiser comme un chef d'orchestre ou tout simplement comme un orchestrateur (c'est le terme le plus récurrent lorsque l'on parle de Kubernetes, NDLR.). Pour l'installer, rien de plus simple, nous allons exécuter la commande :

```
sudo snap install microk8s --classic --channel=1.17/stable
```

Par la suite, nous allons donner les droits utilisateurs sur Microk8s grâce à :

```
sudo usermod -a -G microk8s $USER
```

Enfin, pour vérifier les différents statuts de Microk8s, nous allons lancer :

```
microk8s.status --wait-ready
```

Création d'un projet REST API C#

Ouvrez votre Terminal et créer un dossier directement sur votre Bureau avec la commande suivante :

2

```
mkdir kubernetes_project
```

Ensuite, placez-vous dans le dossier récemment créé :

```
cd kubernetes_project
```

Maintenant, nous allons créer un projet .NET Core Web API par défaut, pour se faire, nous allons utiliser l'assistant CLI dotnet et plus précisément la commande suivante :

```
dotnet new webapi -o app
```

Une fois que votre projet est initialisé, il faut vous placer dans le dossier app :

```
cd app
```

Nous allons vérifier que notre application compile et se lance sans problème :
Petite astuce : Si vous voulez vérifier où vous vous situez en termes de path dans votre terminal, tapez ***pwd***

```
dotnet run watch
```

Votre terminal va vous indiquer l'adresse et le port sur lequel il « écoute », vous devez vous rendre sur cette URL :
https://localhost:5001/WeatherForecast/ . 2
On constate que l'API générée par défaut, par le client .NET, nous renvoie un JSON avec les informations présentes dans la solution WeatherForecast. Nous pouvons dire qu'à ce stage, nous avons une Web API .NET Core sous Linux qui fonctionne correctement. Nous allons, maintenant, modifier le code source initial, pour afficher deux variables d'environnement. Variables qui vont remplacer les températures aléatoires que nous avions juste avant.
Voici les modifications à apporter à votre classe WeatherForecast :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;

namespace app.Controllers
{
  [ApiController]
  [Route("[controller]")]
```

```
public class WeatherForecastController : ControllerBase
{
  private readonly ILogger<WeatherForecastController> _logger;

  public WeatherForecastController(ILogger<WeatherForecastController> logger)
  {
    _logger = logger;
  }

  [HttpGet]
  public InfoModel GetInfo()
  {
    return new InfoModel {
      AppEnvironment = GetEnvironmentVariable("APPENVIRONMENT"),
      AppHost = GetEnvironmentVariable("APPHOST") };
  }

  private string GetEnvironmentVariable(string name)
  {
    _logger.LogInformation($"Getting environment variable '{name}'.");
    return Environment.GetEnvironmentVariable(name.ToLower()) ??
Environment.GetEnvironmentVariable(name.ToUpper());
  }
 }
}
```

Et voici le Model, InfoModel :

```
using System;

namespace app.Models
{
  public class InfoMode
  {
    public string AppEnvironment {get; set;}
    public string AppHost {get; set;}
  }
}
```

Nous allons ensuite, déclarer deux variables d'environnement en copiant les deux lignes ci-dessous directement dans le terminal :

```
export APPENVIRONMENT="development"
export APPHOST = "local"
```

Ces commandes ne vous afficheront rien lors de l'exécution. Par contre, si vous souhaitez vérifier qu'elles ont bien été créées, vous pouvez écrire la commande suivante :

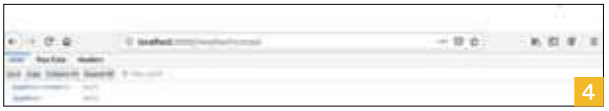
```
env | grep -e APPENVIRONMENT
```

Nous allons maintenant relancer notre application et nous pouvons constater dorénavant à l'affichage les éléments qui suivent : 3
Nous allons donc pouvoir mettre en place notre conteneur Docker avec notre API.

Mise en place de notre container Docker avec notre API

Nous allons pour cela créer un fichier se nommant Dockerfile à la racine du dossier kubernetes_project.
Ce fichier devra ressembler à ça :


3


4


5

```
FROM mcr.microsoft.com/dotnet/core/sdk:3.1-alpine as build
WORKDIR /app

# copy csproj and restore
COPY app/*.csproj ./aspnetapp/
RUN cd ./aspnetapp/ && dotnet restore

# copy all files and build
COPY app/. ./aspnetapp/
WORKDIR /app/aspnetapp
RUN dotnet publish -c Release -o out

FROM mcr.microsoft.com/dotnet/core/aspnet:3.1-alpine as runtime
WORKDIR /app
COPY --from=build /app/aspnetapp/out ./
ENTRYPOINT [ "dotnet", "app.dll" ]
```

On peut observer que le Dockerfile fait appel à l'OS Alpine avec .NET Core 3.1 d'installé. Maintenant que le Dockerfile est prêt, nous allons pouvoir créer notre image. Mais tout d'abord qu'est-ce qu'une image ? Une image Docker est un fichier qui renferme un code dit exécutable ou est inclus tout le nécessaire pour le bon fonctionnement du container, les outils systèmes, les bibliothèques, les OS, etc. Dans votre terminal et dans le dossier kubernetes_project, nous allons donc commencer à utiliser l'application ***docker***.
Tout d'abord, nous allons créer notre image :

```
sudo docker image build --pull -t aspnet3k8s:v1
```

Ensuite nous allons créer notre conteneur et l'exécuter en lui accordant le port 9000.

```
docker run --rm -it -p 9000:80 aspnet3k8s:v1
```

Nous allons donc relancer notre application, et nous pouvons constater en nous rendant sur le localhost de toute à l'heure, que les variables d'environnement sont maintenant dans un état différent : 4
Vous vous demandez sûrement pourquoi les variables d'environnement sont à la valeur null. On peut expliquer ça par le fait que nous nous trouvons sur notre conteneur et que celui-ci est isolé de notre machine. Pour résumer, le container est une machine à part entière. Si nous voulons avoir un résultat identique de celui présent initialement, il nous faudrait redéfinir les variables d'environnement dans notre conteneur avec la commande suivante :

```
docker run --env APPENVIRONMENT=production --env APPHOST=docker --rm -it -p 9000:80 aspnet3k8s:v1
```

Cette commande va donc nous créer les variables d'environnement suivantes dans notre container et exécuter notre conteneur.
Nous aurons donc ce résultat présent lorsque nous relançons l'application : 5

La suite de cet article dans le numéro 241