Module 7 - Rapport projet

Yannick OREAL
16/01/2026

Livrables : https://github.com/aXee808/AI2-module7-project
Annexes : https://github.com/aXee808/AI2-module7-helm

1- Présentation

L'objectif de ce projet était d'industrialiser le déploiement d'une petite API. Afin de ne pas reprendre une api trop minimaliste, j'ai décidé d'implémenter un petit endpoint, et de m'appuyer sur un dataset (fichier csv) très connu de Kaggle : les données des passagers du Titanic.

Les étapes du projet se décline comme suit :
- construire l'API avec une librairie que je connaissais (FastAPI)
- construire un conteneur Docker avec l'api à l'intérieur (Dockerfile)
- construire plusieurs conteneurs qui communiquent (docker-compose)
- déployer un pod avec le conteneur sous Kubernetes
- déployer plusieurs pods, un service, configmap, secret, et un persistent volume sur Kubernetes
- construire un chart Helm avec tous les fichiers yaml des ressources kubernetes souhaitées
- déployer le chart Helm stocké sur un github, sur Kubernetes, avec l'aide de d'ArgoCD

2- API + Dockerfile

L'api contient deux endpoints :

- /health renvoie un json `{'status':'ok'}`
- /passengers renvoie un json `{'dead_female': integer,'dead_male': integer,'survived_female':integer,'survived_male':integer}`

Au lancement de l'api, celle-ci ouvre le fichier titanic.csv présent dans le sous dossier data, sous forme d'un dataframe. Deux arguments sont nécessaires pour interroger le endpoint /passengers, l'âge minimum des passagers, et l'âge maximum, et il retourne le nombre de passagers hommes et femmes, vivants et morts. L'api est très rudimentaire et ne contient ni vérifications des arguments, ni de gestion des erreurs (try/except).

Le Dockerfile est en multi staging, et permet de limiter un peu l'espace de l'image(-30Mo environ), ceci dit ce serait à approfondir (il faut bien connaître l'emplacement des librairies et surtout de uvicorn pour éviter de copier trop de chose sur l'image finale).

3- Docker compose

Le docker-compose m'a permis de tester l'implémentation de deux conteneurs : celui de l'api, et celui du service MinIO qui permet de simuler le service S3 d'AWS. L'objectif étant de stocker sur un bucket S3 le fichier titanic.csv afin que celui-ci ne soit pas copié dans l'image Docker, et que l'api charge le fichier à partir du bucket S3. C'est fonctionnel, toutefois j'ai rencontré deux contraintes :

- stockage des identifiants et paramètres pour se connecter au S3 dans les variables d'environnement
- création manuelle du bucket S3 sur MinIO, et ajout manuel du fichier titanic sur le bucket (il faut donc lancer le conteneur MinIO au préalable pour s'y connecter avant de pouvoir tester le fonctionnement du conteneur de l'api)

Il est possible également de passer par un volume créé dans le docker-compose, c'est fonctionnel.

Par ailleurs, on expose facilement le port des conteneurs vers l'extérieur dans le docker-compose, permettant l'accès à l'api.

4- Déploiement sur Kubernetes

Le déploiement sur Kubernetes se fait en plusieurs étapes, puisqu'il faut créer les ressources une par une (service, persistent volume claim, persistent volume, config map, secret, et deployment). Dans mon cas j'ai créé un namespace "titanic" afin de cloisonner mon api. L'intérêt principal de kubernetes étant de pouvoir dimensionner la capacité de l'api par du scaling horizontale (déploiement de X pods contenant l'api). J'ai choisi d'utiliser un Load Balancer (service) en guise d'interface, avec exposition du port 8000 pour avoir un seul point d'accès. Après quelques itérations de l'api, j'ai décidé également de stocker celle ci sur mon repository Docker Hub (`registry-1.docker.io/axee808/titanic-api:v3`), afin d'éviter les problèmes d'images non valides.

La problématique principale rencontrée lors de cette étape est l'indentation et l'imbrication correcte des attributs dans le fichier yaml ^^ !

Par ailleurs, j'ai testé avec succès le secret et configmap (en passant les mêmes paramètres que sur le fichier .env,)

- `AWS_ACCESS_KEY_ID et AWS_SECRET_ACCESS_KEY dans le secret`
- `AWS_REGION_NAME, AWS_ENDPOINT_URL et AWS_BUCKET_MEDIA dans le configmap`

Cependant, l'api n'exploite pas les variables d'environnement faute d'avoir accès à un conteneur MinIO dans Kubernetes. J'ai noté qu'il était assez simple d'ajouter (via Helm) MinIO dans le même namespace que l'api, c'est donc un test à effectuer de mon côté.

La seule fonctionnalité que je n'ai pas réussi à faire fonctionner, c'est le système de persistent volume. Mon pv et pvc était opérationnel, mais je n'ai pas réussi à copier mon

fichier csv car il semble que le stockage local sur mon pc n'était pas vu directement par Kubernetes, à creuser donc.

5- Déploiement via Helm (puis avec ArgoCD)

Le déploiement par Helm permet de grandement simplifier le déploiement de toutes les ressources yaml déclarées sur la partie 4, et également de les supprimer (helm install et helm uninstall). Cela permet par ailleurs de déployer des ressources dont les charts sont disponibles sur le net (dont ArgoCD…), et donc de disposer d'infrastructures clé en main facilement déployable en dev ou en production.

Il reste toujours l'aspect un peu roots de devoir utiliser les commandes kubectl pour vérifier tous les problèmes de déploiement. C'est là qu'intervient ArgoCD, qui rend l'étape de déploiement et de supervision beaucoup plus simple, puisque l'interface graphique permet de voir tout de suite ce qui ne tourne pas, et d'avoir accès rapidement au code, aux logs, aux évènements. Autre avantage, toute modification sur le github est prise en compte automatiquement (synchronisation automatique) et déployé, et on peut également modifier les "values" directement sur ArgoCD si on le souhaite.

```
PS C:\Users\Raging\Desktop\DATASCIENCE\AI2\Module7\AI2-module7-project\helm> helm list -n titanic
NAME            NAMESPACE       REVISION        UPDATED                                 STATUS          CHART                   APP VERSION
titanic-app     titanic         1               2026-01-15 17:09:45.4014755 +0100 CET   deployed        titanic-app-0.1.0       1.00.0
PS C:\Users\Raging\Desktop\DATASCIENCE\AI2\Module7\AI2-module7-project\helm> kubectl get pods -n titanic
NAME                                READY   STATUS    RESTARTS   AGE
titanic-app-584cd5bcf8-6tdgf   1/1     Running   0          104s
titanic-app-584cd5bcf8-c7b7r   1/1     Running   0          104s
titanic-app-584cd5bcf8-qxvtw   1/1     Running   0          104s
titanic-app-584cd5bcf8-s24x6   1/1     Running   0          104s
titanic-app-584cd5bcf8-s9jv7   1/1     Running   0          104s
PS C:\Users\Raging\Desktop\DATASCIENCE\AI2\Module7\AI2-module7-project\helm>
```

Je mets en annexes le github avec ma version v2 qui fonctionne bien lorsqu'on le lance sur ArgoCD. A priori, la v3 doit fonctionner également.

6- Conclusion (ce qu'il manque pour que l'on parle d'un vrai CI/CD)

Il manque des choses pour que cela devienne un véritable CI/CD :

- implémenter un script python de tests unitaires pour l'api (test des deux endpoints)
- activer un github action pour déclencher les tests unitaires, le build automatique du Docker et export vers le repository Docker Hub, puis modification du helm chart avec le tag de la dernière version d'image
- déploiement automatique des pods via Argo CD (mais je ne sais pas comment interfacer Argo CD et le github Action)