RAPPORT DE PROJET :

APPLICATION DE SUIVI DE COURS DE CRYPTO-MONNAIE

Projet Cloud Computing

Encadreur Académique:
Angela CIOCAN

Étudiants :
Chris Manuel ESSOMBA LENJI
Oumar SARR

2024-2025

Table des matières

Introduction

Dans le cadre du cours de *« Cloud Computing »,* il nous a été demandé de réaliser un projet visant à la création d'une application qui interagit avec une API, ensuite de la conteneuriser en utilisant Docker, et enfin la déployer sur le cloud grâce à Azure, le tout dans le bon respect des technique de travail collaboratif sur Github (CI/CD).
Dans le but de révéler ce défis, nous avons développé et déployé une application qui affiche sous forme de table, les données de crypto-monnaies qu'elle obtient automatiquement grâce à une API.
A continuation nous vous présenterons en détails l'ensemble des taches réalisés et des difficultés rencontrés lors de la réalisation de ce projet.

I. Objectifs du projet:

1. Développer une application capable d'obtenir en temps réel des données sur les crypto monnaie à l'aide d'une API
2. Effectuer des tests unitaires, de spécification et de configuration de l'application
3. Conteneuriser l'application en utilisant Docker
4. Stocker son image Docker dans le Azure container registy
5. Déployer l'application sur le Azure Kubernetes Service tout en conservant la sécurité de la clé API en utilisant le Secret.
6. Automatiser les processus de build et déploiement et de Monitoring de l'app en utilisant Github Action CI/CD

II. Développement de l'application.

Il s'agit d'une application qui se connecte à l'API du site web xxx afin d'obtenir les dernières informations sur les crypto monnaie comme leur noms, symboles, valeurs actuelles…
Pour cela le Framework Flaks a été utilisé pour pouvoir créer une table et afficher la table des données sur celle-ci. Le développement de d'autres fonctionnalités telles que l'analyse des données et la prédiction des prix de cryptos est envisagé pour les prochaines versions de l'application, pour l'heure celle-ci se contente d'afficher certaines informations basiques.
Ci-dessous, une image de l'application :

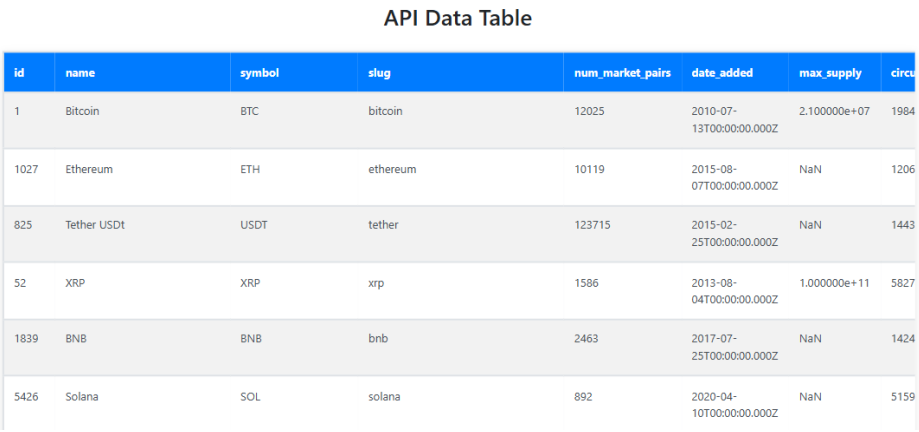


Figure 1: La table affichée par l'application

III. Test de l'application

Dans le but de s'assurer que toutes les fonctionnalités de l'application marchent correctement individuellement et collectivement, trois groupes de tests ont été développés. A continuation la présentation de chacun d'eux :

1. **Les tests unitaires**

Ils se concentrent sur le test de fonctions individuelles de manière isolée, en simulant les dépendances externes (comme les appels à une API) pour vérifier que la fonction se comporte comme prévu.

Quoi sert cet ensemble de tests ?

- Il teste la fonction get_data() lorsque l'appel à l'API réussit.
- Il simule la réponse de l'API pour qu'elle retourne une structure JSON prédéfinie.
- Il vérifie si :
    - La fonction retourne un pandas.DataFrame.
    - Le DataFrame contient le bon nombre de lignes (2 dans ce cas).
    - La cryptomonnaie attendue (Bitcoin) est présente.

Pourquoi est-ce important ?

- Cela garantit que lorsque l'API renvoie des données valides, la fonction get_data() les traite correctement.
- Cela permet d'éviter des régressions si la structure de la réponse de l'API venait à changer.

2. **Les tests d'intégration**

Ils vérifient comment les différentes parties de l'application fonctionnent ensemble (par exemple, les routes Flask + les fonctions + les templates).

Quoi sert cet ensemble de tests?

- Il teste la fonction get_data() lorsque l'appel à l'API échoue (simulation d'une ConnectionError).
- Il vérifie si :
    - La fonction retourne tout de même un pandas.DataFrame (même vide).
    - Le DataFrame est vide (puisque l'API a échoué).

Pourquoi est-ce important ?

- Cela garantit que l'application gère correctement les échecs de l'API au lieu de planter.

- Cela vérifie que la fonction retourne toujours un type cohérent (DataFrame), même en cas d'erreur.

3. Les tests de configuration

Ils configurent des *fixtures* (composants de test réutilisables) pour **pytest**.

Fixture 1 : app()

- Fournit une instance de l'application Flask en mode test (TESTING=True).
- Garantit que l'application est correctement configurée pour les tests (par exemple, sans interférences liées au mode debug).

Fixture 2 : client(app)

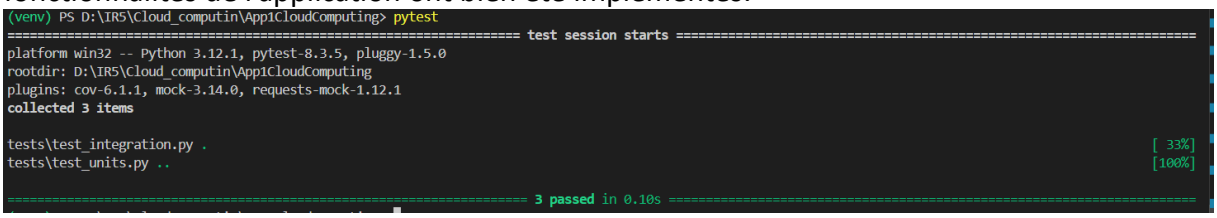
- Fournit un client HTTP de test pour envoyer des requêtes à l'application Flask.
- Utilisé dans les tests d'intégration pour simuler des requêtes de navigateur.

Pourquoi est-ce important ?

- Cela évite la duplication de code en centralisant la configuration des tests.
- Cela garantit que tous les tests utilisent la même configuration de l'application

4. Résultat

Comme vous pouvez le voir, tous les tests ont tous été validés, ce qui signifie que les fonctionnalités de l'application ont bien été implémentés.



IV. Conteneurisation avec Docker

Cette étape concise à créer une image docker contenant l'application et toutes ses dépendances pour que celle-ci puisse être exécutée sur n'importe quel autre environnement.

Pour cela, un fichier Docker est créé dans le but d'établir la configuration à respecter lors de la création de l'image Docker de l'application. Pour ce faire, certaines information indispensables sont précisées comme : la version de python compatible avec l'application (Python 3.12.1), le port sur lequel l'application sera déployée (port 5000) ainsi qu'un paramètre utile *« —no-cache-dir »* qui permet de supprimer les fichiers de téléchargement des librairies après leur installation, cela permet d'économiser de l'espace.

```dockerfile
# Set the programming language #type python version in your venv and copy what you see (all in lower case)
FROM python:3.12.1

#Set the working directory
WORKDIR /app

#Copy the requirements
COPY requirements.txt .

#Install the requirements
RUN pip install --no-cache-dir -r requirements.txt

#Copy everything into the
COPY . .

#Set the port
EXPOSE 5000

#Set the command to run the script
CMD ["python", "app.py"]
```

Figure 2: Configuration du fichier docker

V. Stocker l'image dans le Azure Container Registry (ACR)

L'Azure Container Registry est un service Azure qui permet de stocker des ressources afin de les mettre à la disposition de d'autres services.
Dans ce cas, suite la création de l'image Docker, celle-ci est déposée dans l'ACR afin que d'être chargée dans l'Azure Kubernetes en vue de son déploiement.
Pour ce faire, les étapes suivantes ont été réalisées:

1. Connexion au compte Azure

Elle se fait grâce à la commande « az login » sur l'invite de commande. Ainsi, l'utilisateur peut se connecter au compte Azure dans lequel il voudrait exécuter le projet.

```
(venv) PS D:\IR5\Cloud_computin\App1CloudComputing> az login --use-device-code
To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter the code CACVCS86Q to authenticate.

Retrieving tenants and subscriptions for the selection...

[Tenant and subscription selection]

No     Subscription name    Subscription ID                       Tenant
-----  -------------------  ------------------------------------  ---------------------------------
```

2. Création de l'Azure Container Registry (ACR)

Grace à une nouvelle commande, l'ACR est créé. Dans ce cas, les configurations par défauts ont été choisies et uniquement le groupe de ressource dans lequel l'ACR devait être créé a été précisé.

```
(venv) PS D:\IR5\Cloud_computin\App1CloudComputing> az login --use-device-code
To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter the code CACVCS86Q to authenticate.

Retrieving tenants and subscriptions for the selection...

[Tenant and subscription selection]

No     Subscription name    Subscription ID                       Tenant
-----  -------------------  ------------------------------------  ---------------------------------
```

3. Connexion à l'ACR

Une connexion à l'ACR est établit afin d'une part de vérifier qu'il a bien été créé et d'autre part pourvoir y importer notre image Docker

```
}
(venv) PS D:\IR5\Cloud_computin\App1CloudComputing> az acr login --name chrisuscontainer
Login Succeeded
```

4. Labélisation de l'image Docker

Cette étape est essentielle car elle permet de préciser le noms et le label qui seront rattaché à l'image Docker de l'application une fois dans Azure. Il est recommandé pour ce faire de préciser le nom de l'ACR dans lequel sera conservée l'image et ajouté à son nom, le tag *« latest »* pour préciser qu'il s'agit bien de la dernière version de l'application.

```
docker tag new-crypto-image:latest containerfromscript.azurecr.io/crypto-image:latest
```

5. Transfert de l'image dans le ACR

Il ne reste plus qu'à transférer l'image dans son container. Etant déjà connecté, aucune nouvelle authentification n'est requise.

```
> docker push  containerfromscript.azurecr.io/crypto-image:latest
```

VI. Déployer l'image dans l'Azure Kubernetes Service (AKS)

Une fois que l'image est accessible dans le Cloud Azure grâce à l'ACR. Il ne reste plus qu'à créer et configurer un Cluster de Kubernetes qui permettra de déployer l'application, surveiller son fonctionnement et orchestrer des mises à jour etc.
Pour ce faire, Trois fichiers sont créés et configurés. Ceux-ci qui jouent un chacun un rôle essentiel pour la réussite de cette opération.

1. **Configuration des trois fichiers de configuration du cluster**

    a. Création du fichier Deployment.yaml

Ce fichier contient la feuille de route que suivra l'AKS pour le déploiement de l'application. De fait, celui-ci contient des informations essentielles telles que : le nombre d'instances de l'application à créer appelés encore *replicas*. Cette fonctionnalité est utile lorsque une instance de l'application rencontre des soucis de fonctionnement, une autre peut prendre le relai. Nous y précisons également le nom de l'image tel que nous l'avons défini dans le tag. Ainsi que le port sur lequel l'application sera déployer sans oublier la clé API essentiel à l'obtention des données. Dans ce cas, sa valeur n'est pas accédée directement mais plutôt la variable d'environnement qui la contient. Cette variable est définit dans le fichier Secret que nous verrons à continuation.

manifests > ! deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: crypto-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: crypto-app
  template:
    metadata:
      labels:
        app: crypto-app
    spec:
      containers:
        - name: crypto-app
          image: containerfromscript.azurecr.io/crypto-image:latest
          ports:
            - containerPort: 5000
          env:
            - name: API_KEY
              valueFrom:
                secretKeyRef:
                  name: crypto-api-secret
                  key: API_KEY
      imagePullSecrets:
        - name: acr-auth

```

Figure 3: Configuration du fichier de déploiement

b. Création du fichier Service.yaml

Ce fichier est important parce qu'il permet au conteneur de l'application de communiquer avec d'autres conteneur de l'environnement (*Node*) et permet aussi à l'application d'être accessible depuis l'extérieur grâce à l'activation de sa fonction *Load balancer*.

manifests > ! service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: crypto-app-service
spec:
  type: LoadBalancer #to make it accessible from the outside api,...
  ports:
    - port: 80
      targetPort: 5000
  selector:
    app: crypto-app

```

Figure 4: Configuration du fichier de Service

c. Création du fichier Secret.yaml

Il s'agit du fichier qui contient toutes les données sensibles utilisées dans l'application et qui les rend accessible à l'exécution. Dans ce cas, nous y gardons la clé API permettant d'obtenir les données de cryptos.

```
manifests > ! secret.yaml
1   apiVersion: v1
2   kind: Secret
3   metadata:
4     name: api-key-secret
5   type: Opaque
6   data:
7     API_KEY: "OTg0MWFkYWYtNDFmZC00MTAyLTg[redacted]h"
8
```

Figure 5 : Configuration du fichier de Secret

2. Création et configuration de l'AKS

a. Création du cluster dans AKS

Il s'agit du service qui orchestre tout le processus. Pour sa création, il est impératif de préciser le nombre de Nodes. Un Node est une machine virtuelle dans laquelle un ou plusieurs conteneur d'applications sont exécutés.
Dans le cluster nous précisons des options permettent à l'AKS d'interagir avec d'autres services de manière sécurisée sans nécessiter une nouvelle authentification, aussi, une option permettant d'incorporer des outils de surveillance dont nous aurons besoin plus tard pour nos analyses et enfin générer une clé ssh au cas où elle n'aurait déjà été générée.

```
(venv) PS D:\IR5\Cloud_computin\App1CloudComputing> az aks create --resource-group CloudComputingIR4 --name chrisus-cluste
r --node-count 2 --enable-managed-identity --enable-addons monitoring --generate-ssh-keys
SSH key files 'C:\Users\chris.DESKTOP-GCU74DF\.ssh\id_rsa' and 'C:\Users\chris.DESKTOP-GCU74DF\.ssh\id_rsa.pub' have been
generated under ~/.ssh to allow SSH access to the VM. If using machines without permanent storage like Azure Cloud Shell w
```

b. Obténir les identifiants Kubernetes

Cette étape permet d'acquérir les autorisations nécessaires pour pouvoir interagir avec kubernetes depuis l'invite de commande en utilisant l'outil « *Kuberctl* ».

```
PS D:\IR5\Cloud_computin\App1CloudComputing> az aks get-credentials --resource-group CloudComputingIR4 --name chrisus-cl
ter
Merged "chrisus-cluster" as current context in C:\Users\chris.DESKTOP-GCU74DF\.kube\config
```

c. Appliquer les fichiers aux kubernetes

Cette étape permet d'exécuter nos fichiers que nous avions créés précédemment afin que le AKS puisse être convenablement configuré. A partir de là, l'application est créer et si nous allons sur notre kubernetes nous pouvons y accéder.

3. Intégration de GitHub Actions pour l'automatisation CI/CD

La figure ci-dessous montre l'exécution réussie d'un workflow GitHub Actions, défini dans le fichier main_flashapp1.yml. Ce workflow est déclenché manuellement (workflow_dispatch) et comporte deux étapes principales :

- **build** : Cette étape permet de préparer l'environnement (installation des dépendances, tests éventuels, vérifications).
- **deploy** : Cette étape assure le déploiement automatique de l'application, par exemple sur un serveur ou une plateforme cloud

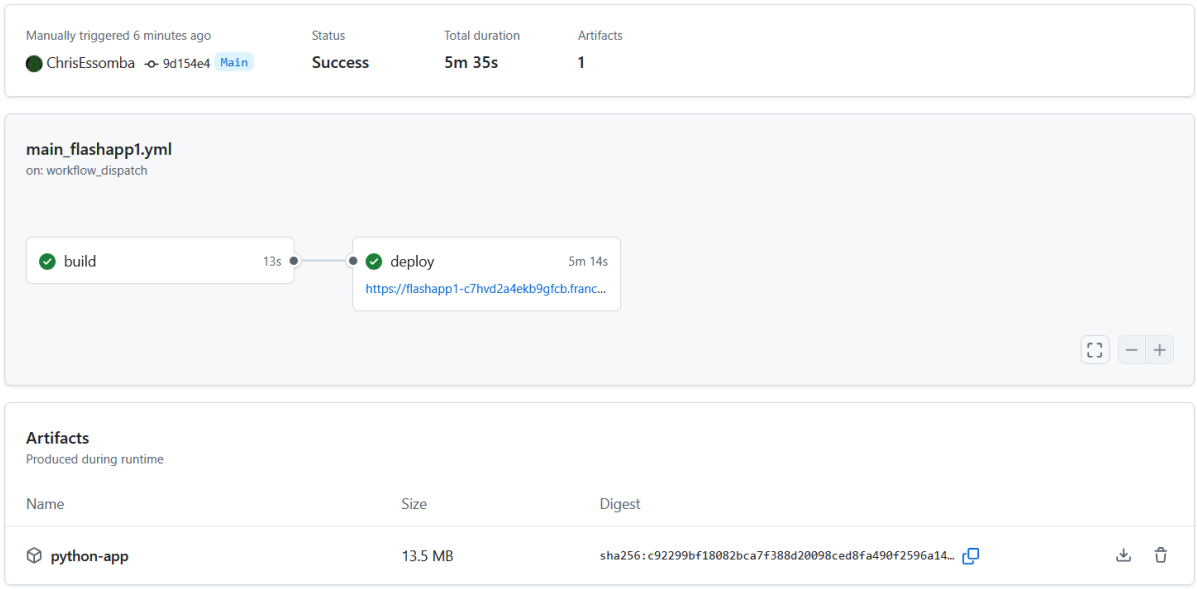


Figure 6 : Image du Github Actions

Conclusion

Ce projet a été une expérience enrichissante, car pour réussir à le mener à bien, nous avons dû développer énormément de nouvelles compétences : de la gestion des APIs au déploiement d'applications dans le cloud en passant par le processus de conteneurisation. Nous avons également rencontrés pleins de challenges, qui nous ont poussés à approfondir encore plus les notions que nous maitrisions mal ou considérions à tort comme acquises. A titre d'exemple, il a été challengeant pour nous de réussir à déployer notre application en tenant compte de la variable d'environnement (qui contient ici la clé API) à laquelle elle était rattachée. Pour résoudre ce problème nous avons dû créer le fichier « Secret » pour pouvoir contenir cette valeur et la rendre accessible à notre application de manière sécurisée.