# Objectifs :

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

# 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 :

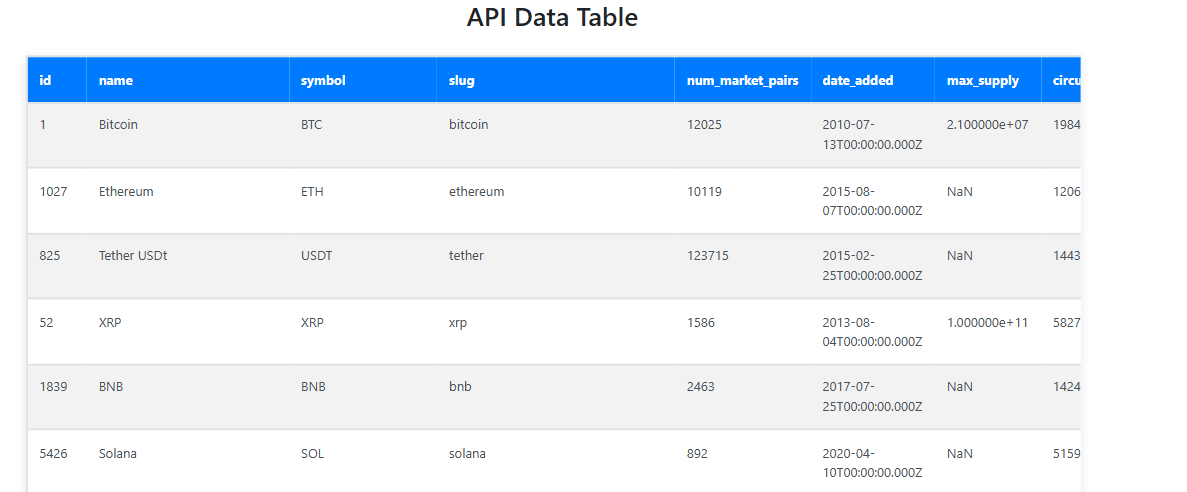


Figure : La table affichée par l'application

# 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.

1. **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.

1. **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)**

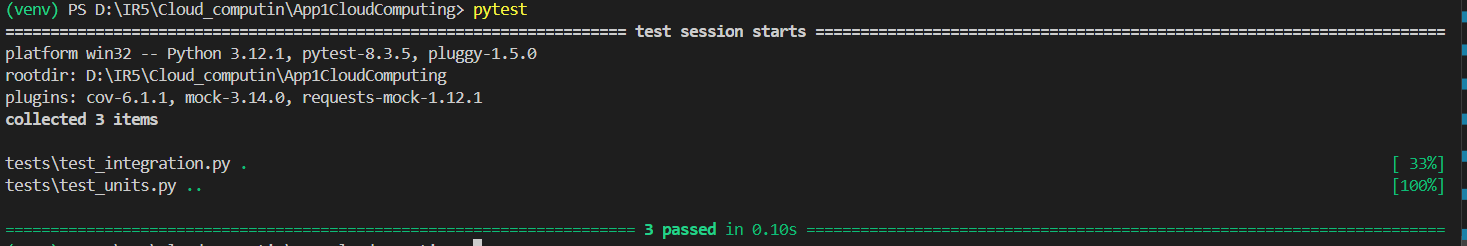
* Fournit un client HTTP de test pour envoyer des requêtes à l’application Flask.
* Utilisé dans les tests d’intégration pour simuler des requêtes de navigateur.

**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

1. **Résultat**

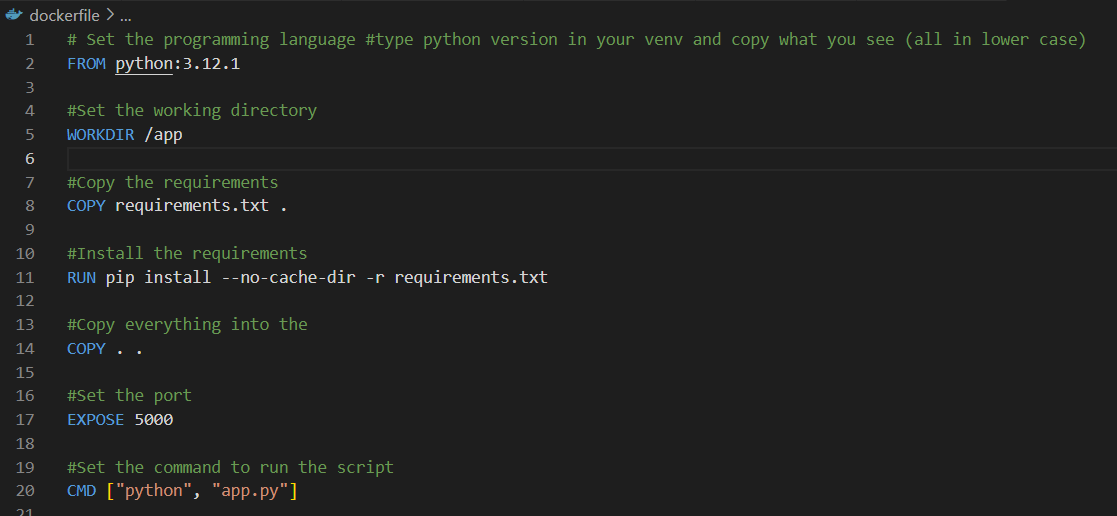
Comme vous pouvez le voir, tous les tests ont tous été validés, ce qui signifie que les fonctionnalités de l’application ont bien été implémentés.



# 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.



# Stocker l’image dans le Azure Container Registry (ACR)

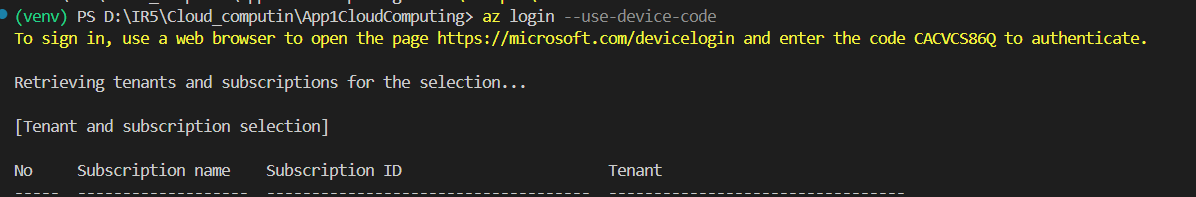
L’Azure Container Registry est un service Azure qui permet de stocker des ressources afin de les mettre à la disposition de d’autres services.

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:

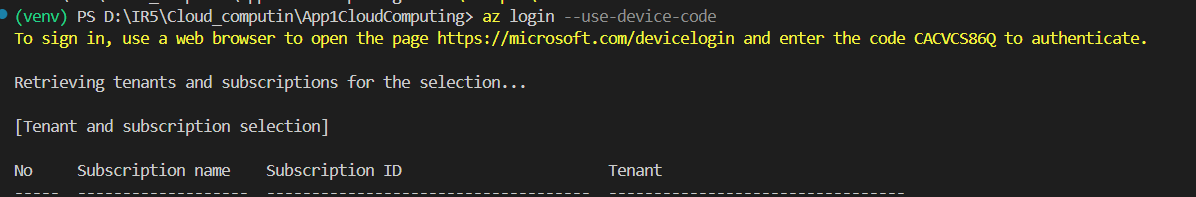
1. Connexion au compte Azure

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.



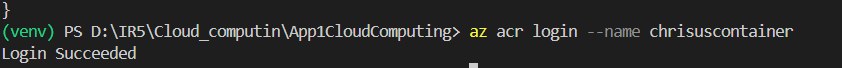
1. Création de l’Azure Container Registry (ACR)

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.



1. 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



1. 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.



1. 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.



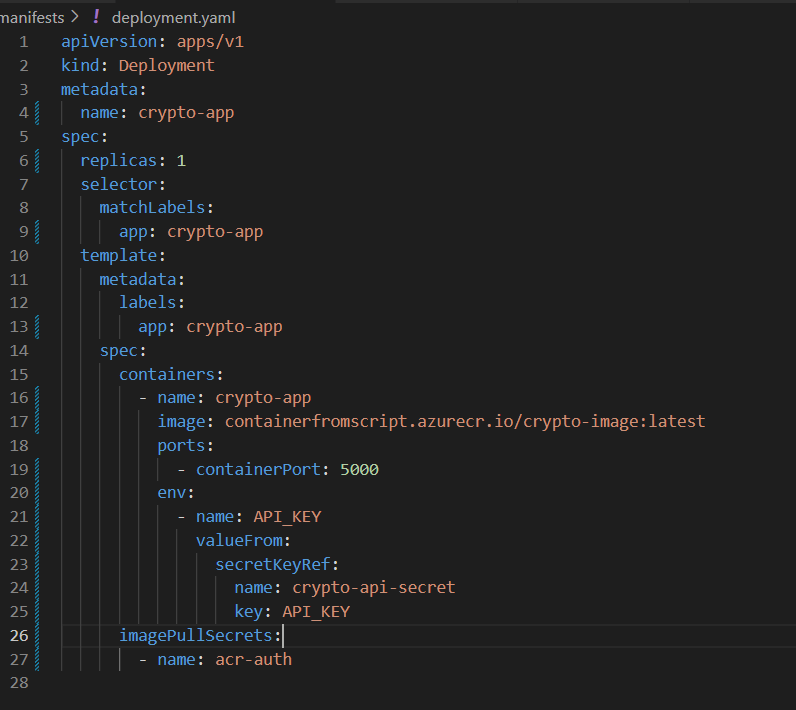
# Déployer l’image dans l’Azure Kubernetes Service (AKS)

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.

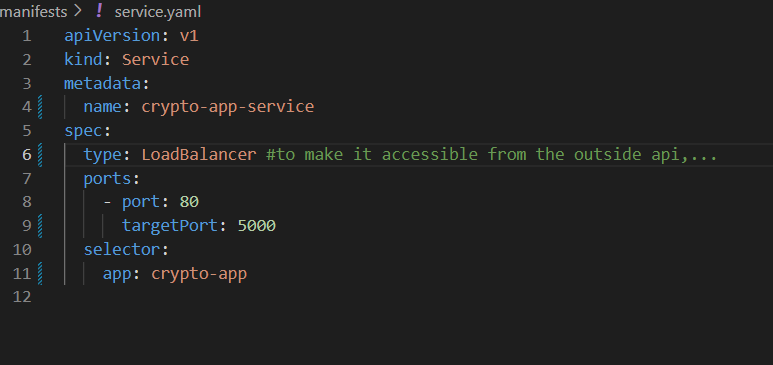
1. **Configuration des trois fichiers de configuration du cluster**
2. Création du fichier Deployment.yaml

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.



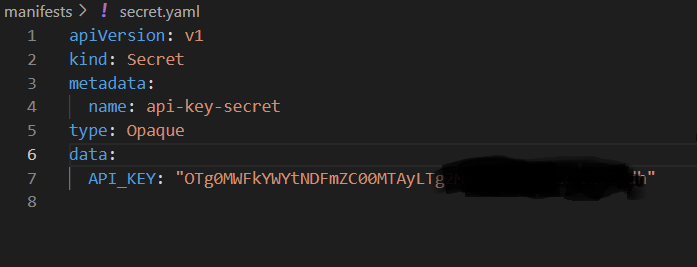
1. Création du fichier Service.yaml

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*.



1. Création du fichier Secret.yaml

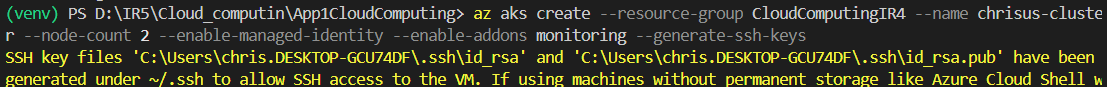
Il s’agit du fichier qui contient toutes les données sensibles utilisées dans l’application et qui les rend accessible à l’exécution. Dans ce cas, nous y gardons la clé API permettant d’obtenir les données de cryptos.



1. **Création et configuration de l’AKS**
2. Création du cluster dans AKS

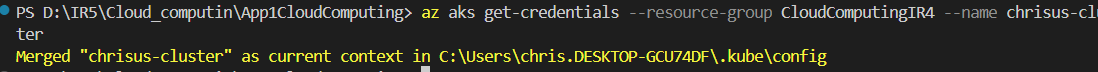
Il s’agit du service qui orchestre tout le processus. Pour sa création, il est impératif de préciser le nombre de Nodes. Un Node est une machine virtuelle dans laquelle un ou plusieurs conteneur d’applications sont exécutés.

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.



1. Obténir les identifiants Kubernetes

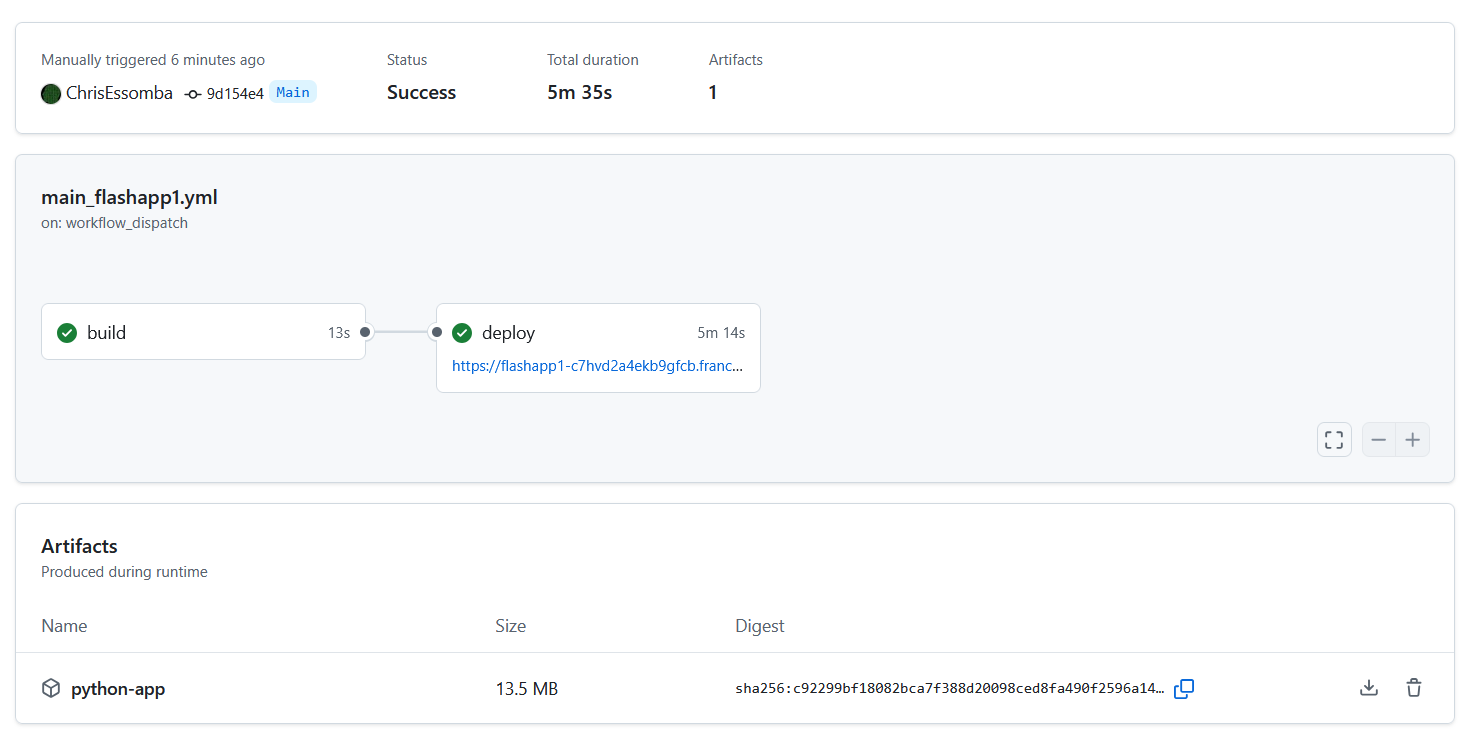
Cette étape permet d’acquérir les autorisations nécessaires pour pouvoir interagir avec kubernetes depuis l’invite de commande en utilisant l’outil « *Kuberctl* ».



1. 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.

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

****

La figure ci-dessus 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.