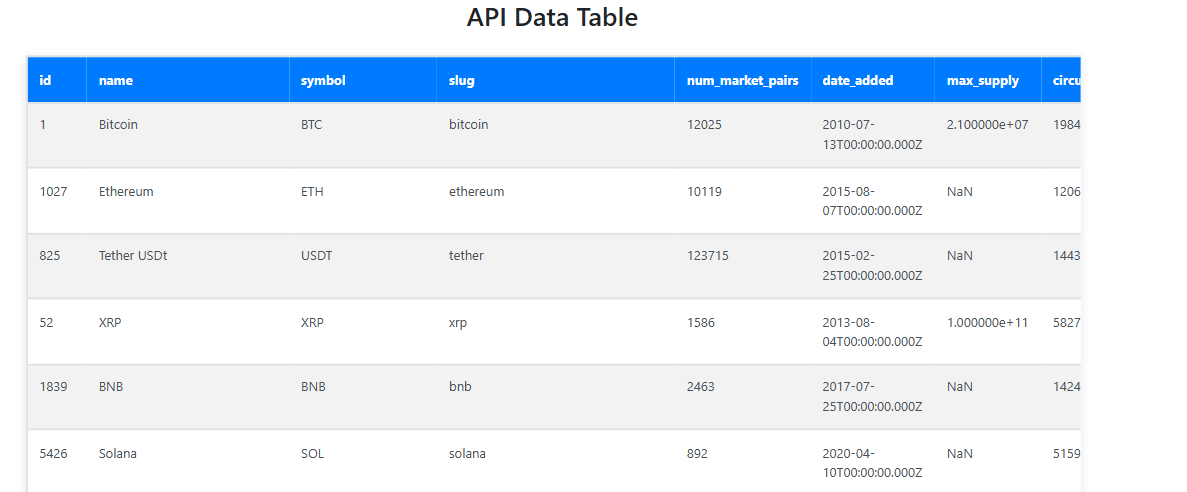
Objectifs :

1. Développer une application capable d’obtenir en temps réel des données d’information sur les crypto monnaie à l’aide d’une API
2. Effectuer des tests unitaires et généraux de l’application
3. Conteneuriser l’app en utilisant Docker
4. Stocker l’image Docker de l’appp 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 deployement 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 « « «

Pour cela le Framework flask 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’app.



# Test de l’application

Dans le but de s’assurer que toutes les fonctionnalités de l’application fonctionnent bien. Trois types 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)**

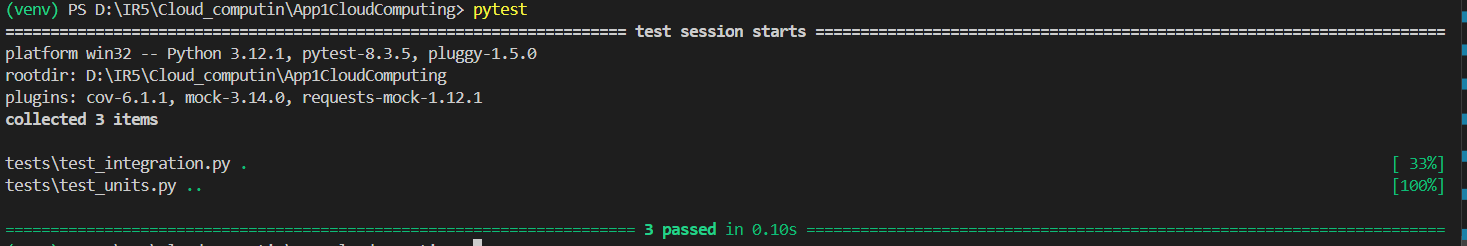
* Fournit un client HTTP de test pour envoyer des requêtes à l’application Flask.
* Utilisé dans les tests d’intégration pour simuler des requêtes de navigateur.

**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**

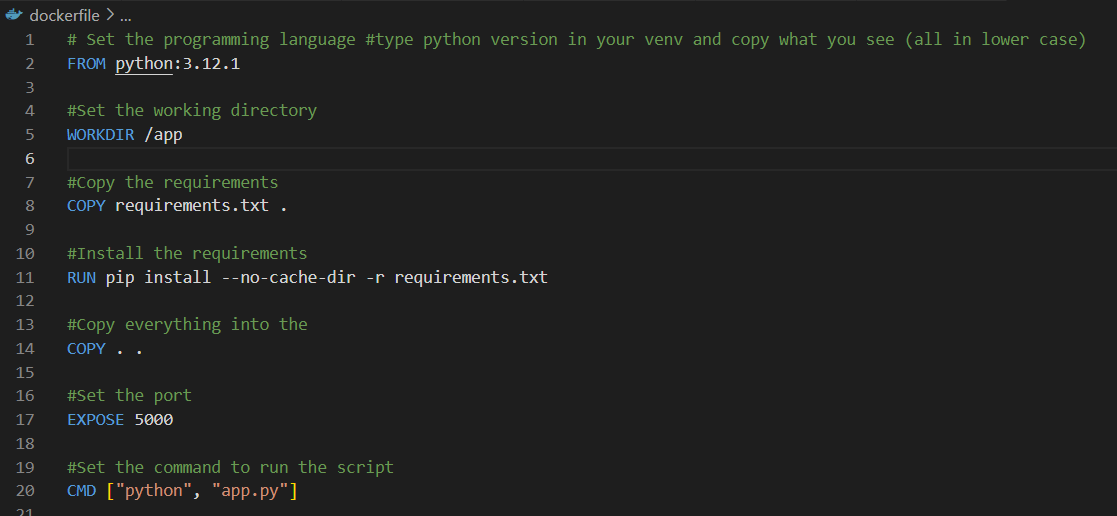
Comme vous pouvez le voir, tous les tests ont été validés.



# Conteneurisation avec Docker

Cette étape concise à créer une image docker contenant notre application et toutes ses dépendance pour que celle-ci puisse être exécuter depuis n’importe quel autre ordinateur.

Pour cela nous créons un Docker file qui permet de configurer la maniére avec laquelle l’image sera générée. Pour ce faire nous avons préciser des informations telles que la version de python compatible avec l’application (Python 3.12.1), le port sur lequel l’application est deployée (port 5000) ainsi que un paramètre intéressant ‘—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)

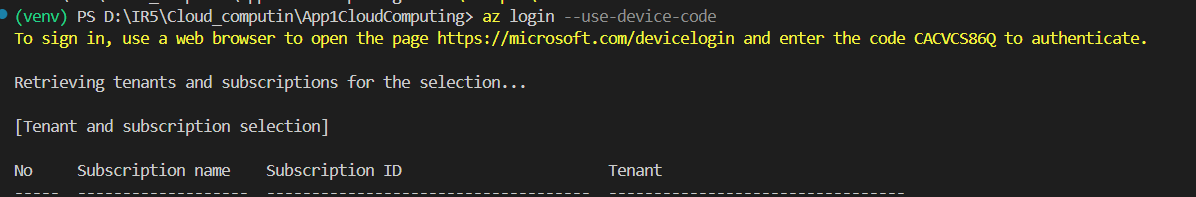
Une fois l’image Docker créer, l’étape suivante est de la déposer sur l’ACR afin que le service Kubernetes

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. (diff with blob storage ?)

Dans ce cas, puisque l’image Docker de l’application a été créée, nous la deposons dans l’ACR afin que celle-ci puisse être accédée par l’Azure Kubernetes en vue de son déploiement. Pour ce faire nous suivons les étapes suivantes :

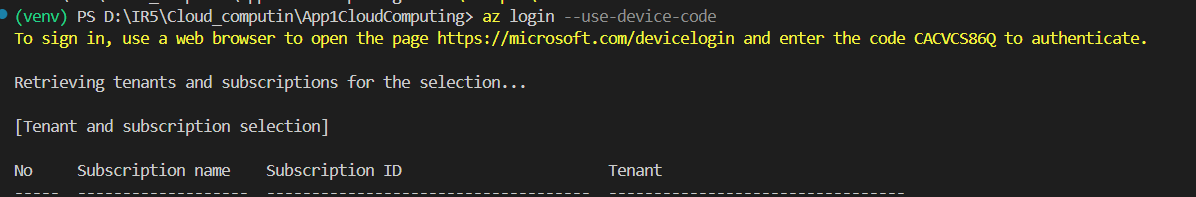
1. Connexion au compte Azure

Etant donné que nous avons déjà un compte Azure et une ressource dédie dans laquelle seront créer les different service Azure, il ne nous reste plus qu’a nous connecter à celui-ci



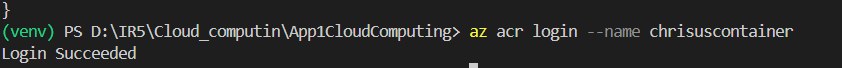
1. Creation de l’Azure Container Registry (ACR)

Nous utilisons une commande pour créer l’ACR, dans laquelle nous n’avons qu’a préciser le nom que nous souhaitons lui attribuer ainsi que le groupe ressource (déjà créé au préalable) dans lequel il sera créé. Nous avons également opté pour utilisé la configuration par défaut.



1. Connexion à l’ACR

Nous établissons ensuite la connexion à l’ACR afin d’une part de vérifié qu’il a bien été crée et d’autre part pourvoir y importer notre image Docker



1. Labélisation de l’image Docker

Cette étape est éssentielle car elle nous permet de préciser le noms et le label qui seront rattaché à notre image Docker une fois dans Azure. Il est recommandé pour ce faire de préciser le nom de l’ACR dans lequel sera conservé l’image et à en de 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’a transférer l’image dans son container. Puisque nous nous y sommes déjà connecté, cela ne posera aucun problème.



1. Visualisation de l’image dans l’ACR

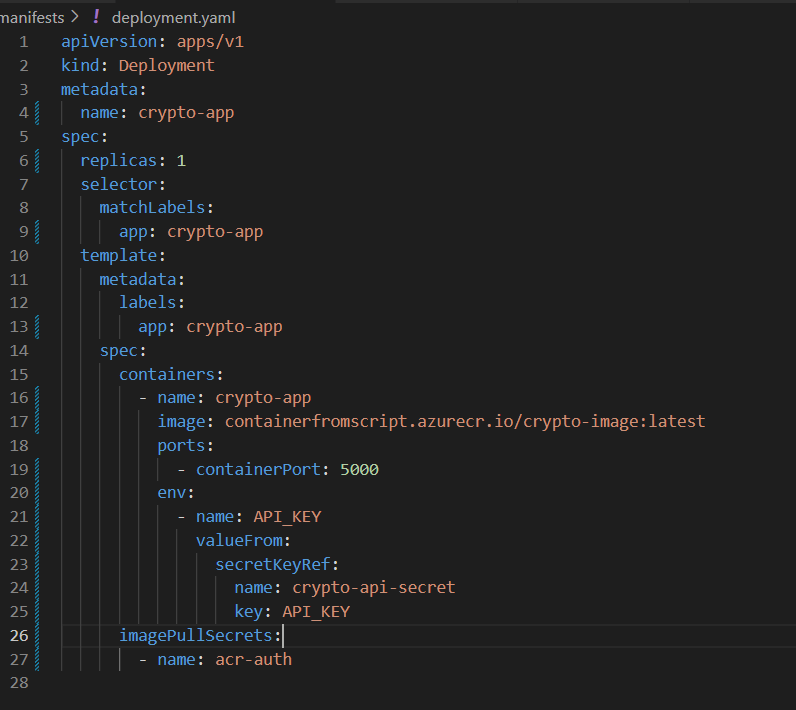
Déployer l’image dans l’Azure Kubernetes Service (AKS)

Une fois que l’image est accéssible dans le Clous Azure grâce à l’ACR. Il ne reste plus qu’a créer et configurer un Cluster de Kubernetes qui permettra de déployer l’application, surveiller son fonctionnement et orchestrer des mise à jour et bien d’autres encore.

Pour ce faire, nous creons trois fichiers qui jouent un chacun un rôle essentiel pour la réussite de cette operation.

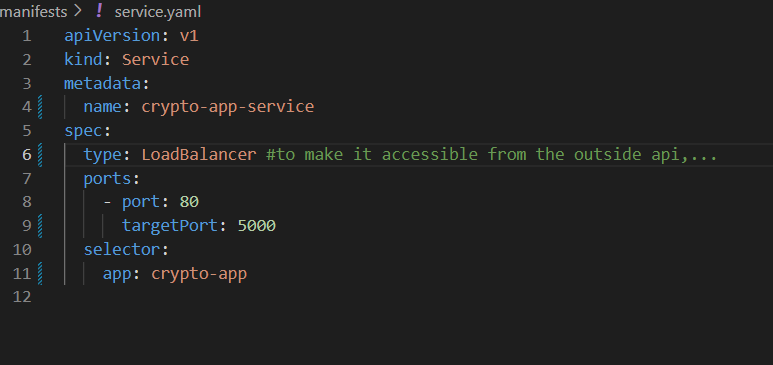
1. Configuration des trois fichiers
2. Creation du fichier Deployment.yaml

Ce fichier contient la feuille de route que suivra l’AKS pour le deployment de notre application. De fait, celui-ci contient des informations essentielles telle que le nombre d’instance de notre application à créer (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.



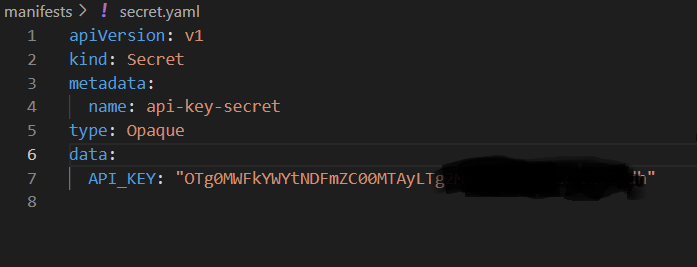
1. Création du fichier Service.yaml

Ce service est important parce qu’il permet au conteneur de l’application de communiquer avec d’autres conteneur de l’environnement et permet aussi à l’application d’etre accessible depuis l’exterieur grâce à son option Load balancer.



1. Création du fichier Secret.yaml

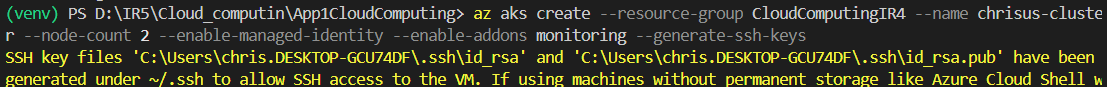
Il s’agit du fichier qui contient toutes les données sensibles utilisées dans l’application et qui les rend accessible à l’exécution. Dans notre cas, nous y gardons la clé API permettant d’obtenir les données de cryptos.



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

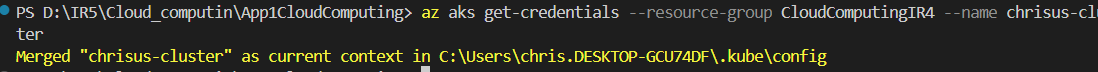
Il s’agit du service qui orchestre tout le processus. Nous y précisons des informations comme le nombre de node. Un node est un machine virtuelle dans lequel une ou plusieurs conteneur d’applications sont exécuter.

Dans le cluster nous précisons des options permettent à l’AKS d’interragir avec d’autres services de manière sécuriser sans nécessiter de nouveau indentifiants, une option permettant d’incoporer des outils de surveillance dont nous aurons besoin plus tard pour nos analyse et enfin générer une clé ssh au cas où elle n’aurait djà été générée.



1. Obtenir les indentifiant Kubernetes

Cette etape permet d’aquerir les autorisation nécessaire 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’associer nos fichiers que nous avions créés précédemment afin que ceux-ci puissent jouer leur rôles. A partir de là, l’application est créer et si nous allons sur notre kubernernetes nous pouvons y accéder.

1. Gestion des CI/CD
2. Monitoring