Documentation du Système de Surveillance Environnementale pour Jardins Urbains

Introduction

Le projet vise à fournir une plateforme API basée sur le cloud pour la surveillance environnementale des jardins urbains. Grâce à cette plateforme, nous pouvons surveiller et analyser les conditions environnementales pour prendre des décisions éclairées sur l'entretien des jardins urbains.

Outils Azure Utilisés

1. Azure App Service

- **Pourquoi** : Azure App Service offre une plateforme hautement évolutive pour l'hébergement d'applications web et mobiles. Dans notre cas, nous l'avons utilisé pour héberger notre application Flask qui constitue notre API.
- **Configuration** :
    - **Plan de Service**: Nous avons opté pour un plan de tarification spécifique adapté à nos besoins en termes de performance et de coûts.
    - **Déploiement**: Une intégration Git a été mise en place pour faciliter les déploiements continus. Ainsi, chaque fois qu'un changement est poussé vers la branche principale, Azure App Service récupère automatiquement les changements et déploie la nouvelle version.
    - **Variables d'Environnement**: Des variables d'environnement ont été définies pour sécuriser certains aspects de l'application, tels que les clés API, les chaînes de connexion, etc.
    - **Réglages**: La plateforme a été configurée pour être compatible avec les applications Python, notamment Flask.
- **Pourquoi cette configuration** : Ces configurations garantissent que l'API est non seulement sécurisée mais aussi hautement disponible. L'intégration Git facilite les mises à jour, et les variables d'environnement sécurisent les informations sensibles.

2. Azure Functions

- **Pourquoi**: Azure Functions nous a permis de créer des solutions serverless pour exécuter des scripts ou des fonctions en réponse à divers événements. Dans notre projet, cela a été utilisé pour des tâches spécifiques qui devaient être automatisées sans avoir besoin d'un serveur dédié.
- **Configuration**:
    - **Déclencheurs**: Nous avons utilisé des déclencheurs spécifiques pour exécuter notre fonction en fonction de certains événements ou à des moments précis.
    - **Langage et environnement**: Python a été utilisé comme langage de programmation, et nous avons défini les packages nécessaires dans le fichier `requirements.txt`.
    - **Paramètres d'application**: De même que pour l'App Service, des variables d'environnement ont été définies pour sécuriser les informations sensibles utilisées dans les fonctions.
- **Pourquoi cette configuration**: La nature serverless d'Azure Functions nous a permis d'économiser des coûts et d'automatiser certaines tâches sans avoir besoin d'infrastructures supplémentaires. Sa flexibilité en termes de déclencheurs et sa capacité à intégrer différents services Azure le rendent idéal pour notre projet.

Étapes d'Implémentation Détaillées

1. Configuration initiale

a. Environnement Azure : Avant tout développement, un environnement Azure a été configuré pour permettre le déploiement et la mise en œuvre des services.

b. Création de l'App Service : Dans Azure, un App Service a été créé pour héberger notre application Flask. Cela nous a permis de déployer facilement notre API et de la rendre accessible via le web.

2. Développement de l'API

a. Choix du Framework : Flask a été choisi pour son caractère léger et sa flexibilité. Cela nous a permis de développer rapidement une API répondant à nos besoins.

b. Points d'accès API : Un endpoint a été crée afin de pouvoir consulter les données stockées.

c. Modèle de Données : Un modèle de données a été conçu pour stocker et organiser les informations environnementales. Cela a facilité le traitement et l'extraction des données.

d. Extraction des Données : Des méthodes ont été mises en place pour extraire des données de diverses sources environnementales, les traiter et les rendre disponibles via l'API.

3. Tests de Performance avec Locust

a. Installation et Configuration de Locust : Locust, un outil de test de performance, a été installé et configuré pour simuler différents scénarios de trafic.

b. Création du locustfile.py : Un fichier de configuration spécifique à Locust (`locustfile.py`) a été créé. Il définit le comportement des utilisateurs simulés et les requêtes qu'ils effectuent sur l'API.

c. Lancement des Tests : Des tests ont été réalisés pour simuler différents volumes de trafic et évaluer la capacité de l'API à gérer de nombreuses requêtes.

d. Analyse des Résultats : Après avoir exécuté les tests, les résultats ont été analysés pour déterminer les limites de performance de l'API et pour identifier les éventuels goulets d'étranglement.

4. Déploiement sur Azure

a. Configuration de l'Environnement : Les variables d'environnement et les paramètres nécessaires au déploiement ont été définis dans Azure.

b. Mise en ligne de l'API : L'API a été déployée sur l'App Service d'Azure à l'aide des outils de développement Azure.

c. Vérification de la mise en ligne : Après le déploiement, des tests ont été effectués pour s'assurer que l'API était fonctionnelle et accessible en ligne.

Conclusion

Ce projet a combiné plusieurs services Azure pour créer une solution complète de surveillance environnementale pour les jardins urbains. De l'API hébergée sur Azure App Service aux automatisations serverless avec Azure Functions, chaque composant a été soigneusement choisi et configuré pour offrir la meilleure solution possible.