Introduction

Dans le cadre de ce projet, nous avons déployé une application conteneurisée sur Azure en respectant les principes DevOps d'automatisation et de scalabilité. L'objectif principal était de configurer une infrastructure cloud dynamique capable d'héberger cinq composants principaux(API, MVC, Worker_Content, Worker_Image et Worker_DB) tout en assurant leur autoscaling en fonction de différents déclencheurs (CPU, Service Bus et Event Hub).
Pour y parvenir, on a utilisé Azure DevOps pour automatiser le déploiement via des pipelines CI/CD, ARM Templates pour la gestion de l'infrastructure et KEDA pour l'ajustement automatique des ressources. Tous les paramètres sensibles ont été externalisés dans Azure Key Vault et App Configuration évitant ainsi tout hardcoding dans le code.
Dans ce rapport, je détaille les étapes clés de mon implémentation, les défis rencontrés, les solutions apportées et les résultats obtenus.

De manière générale, j'ai mis mes informations dans les dockerfile, secret.env et rajouter les droits nécessaires. J ai aussi mis mon app config dans les differents fichiers json ainsi que mes initiales gali

```
# Pour le managed identity pour dev seulement.
ENV AZURE_CLIENT_ID="25c5f4cf-0897-442f-a215-f30193d1e87a"
ENV AZURE_TENANT_ID="a3367121-31c1-4410-860a-5703cb95a5b4"
ENV AZURE_CLIENT_SECRET="X5U8Q~VHq6JLfgKjJLOj8PiMko51aL27usZVaajs"
```

```
# Add Owner roles to pipeline service princi
# Add Global Administrator in AD to pipeline
# Add Key Vault Secrets Officer to pipeline
# Add AcrPull to pipeline service principale
```

2-DeployKube:

J'ai modifié ce fichier Cluster.yml de la manière suivante:

1. Déploiement de l'API

J'ai configuré un deployment pour l'API (api-deployment) en précisant :

- L'image conteneurisée stockée dans mon Azure Container Registry (galiregistry.azurecr.io/api:latest).
- L'ouverture du port 8080 pour exposer le service.
- L'utilisation de variables d'environnement (App Configuration et Key Vault) via un secret Kubernetes (my-secret), sans hardcoder

d'informations sensibles dans le code source.

3. Déploiement de l'application MVC

De la même manière, j'ai configuré le deployment du MVC (mvc-deployment) avec :

- Une image hébergée dans mon registre (galiregistry.azurecr.io/mvc:latest).
- Le port 8081 exposé.
- L'ajout d'un Horizontal Pod Autoscaler (HPA) (mvc-deployment-hpa) pour rendre le MVC autoscalable en fonction de la consommation CPU.

4. Déploiement des Workers

J'ai déployé trois workers essentiels :

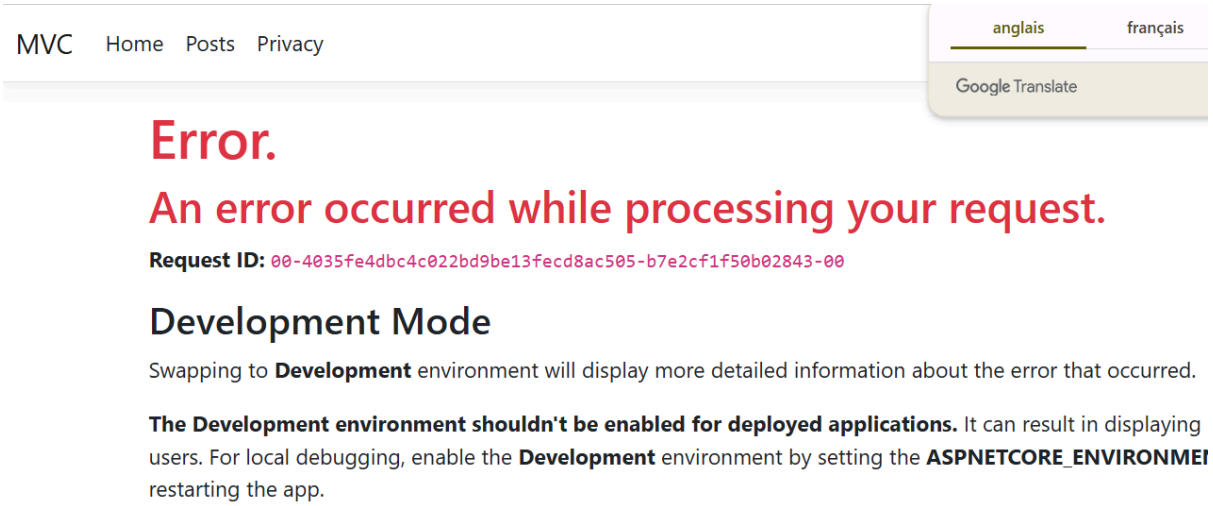
- Worker_Content
- Worker_DB
- Worker_Image Chacun avec son propre deployment, en configurant L'image correspondante dans Azure Container Registry.
- L'utilisation sécurisée de secrets pour récupérer toutes les informations de configuration nécessaires (comme les connexions Azure).

5. Exposition des Services API et MVC

Pour permettre aux utilisateurs (et aux autres services) d'accéder à l'API et à l'application MVC, deux services Kubernetes sont configurés (api-service et mvc-service) de type LoadBalancer :

- api-service expose le port 8080.
- mvc-service expose le port 8081. Cela permet d'obtenir une IP publique pour accéder directement à ces deux applications après le déploiement.

6-Program.cs du projet MVC:



Le post ne fonctionnait pas sur le site de MVC J'ai corrigé l'injection de dépendance de EventHubController pour utiliser la bonne valeur (EventHubName) a la ligne 132. Cette correction a permis de rétablir la fonctionnalité de création de posts sur le site MVC.Elle respecte aussi la logique d'architecture event-driven où le MVC est un publisher et non un consumer.

7- program.cs de Worker_Content:

En travaillant sur le chargement dynamique des configurations via Azure App Configuration, j'ai rencontré un souci particulier avec la ligne suivante :

```
ConfigurationSetting EventHubHubName =
appConfigClient.GetConfigurationSetting("ApplicationConfiguration:EventHubN
ame");
```

À l'origine, cette ligne contenait un retour à la ligne parasite mal placé dans la clé utilisée, ce qui empêchait App Configuration de retourner la bonne valeur. Aucune donnée n'était récupérée pour le nom de l'Event Hub, ce qui bloquait le bon fonctionnement du traitement des événements dans le worker.

J'ai retiré le retour à la ligne pour que la clé "ApplicationConfiguration:EventHubName" soit lue en une seule ligne continue, ce qui a permis de corriger l'erreur immédiatement. L'événement pouvait ensuite être traité correctement depuis Event Hub.

Cette correction mineure mais critique a permis de restaurer la liaison avec le nom de l'Event Hub défini dans App Configuration, et de permettre au worker de fonctionner normalement dans l'architecture event-driven.

8- azure-pipelines1:

Dans ce pipeline, j'ai remplacé la valeur de la variable AzureSubscription par mon identifiant de souscription Azure personnel value: 'Azure subscription 2(564ce5bd-33c8-4554-ad6e-806f11d688e1)'. Cela permet à Azure DevOps de cibler précisément mon environnement cloud et d'effectuer toutes les opérations (authentification, déploiement, push vers ACR, ...) dans mon propre abonnement plutôt que celui fourni par vous.

Fonction du pipeline

Ce pipeline CI/CD a pour but de :

- Nettoyer les Dockerfile de toutes les variables sensibles (AZURE_CLIENT_ID, AZURE_TENANT_ID, AZURE_CLIENT_SECRET) pour garantir que rien n'est hardcodé dans les conteneurs.

- Compiler et publier les images Docker des projets API, MVC, Worker_Content, Worker_Image et Worker_DB vers mon Azure Container Registry (galiregistry) via la connexion Docker Registry Dynamic.

- Garantir la conformité avec les exigences de sécurité du devoir (aucune information sensible dans les Dockerfile).

9- azure-pipeline3:

Dans le pipeline j'ai modifié la tâche AzureKeyVault de lecture des secrets en ajoutant 'ClientID,TenantID,ClientSecret,ConnectionStringSB,ConnectionStringEventHub,ConnectionStringBlob'

Avant seuls ClientID, TenantID et ClientSecret étaient récupérés depuis Azure Key Vault. J'ai rajouté ConnectionStringSB, ConnectionStringEventHub, et ConnectionStringBlob pour pouvoir :

- Récupérer la connexion au Service Bus (pour les workers Worker_Content et Worker_Image qui écoutent des messages),

- Récupérer la connexion à l'Event Hub (pour le worker Worker_DB qui écoute les événements),

- Récupérer la connexion au Blob Storage (nécessaire pour les workers et pour stocker/traiter les fichiers).

Sans ces connexions, les workers et les applications ne pourraient pas accéder aux services Azure (Service Bus, Event Hub, Blob) ce qui empêcherait le fonctionnement normal du projet et du cluster.
Cela respecte aussi l'exigence du devoir à savoir qu' aucune information sensible ne doit être hardcodée.

Ce pipeline a pour rôle principal de :

- Créer la clé SSH automatiquement pour Kubernetes AKS si elle n'existe pas encore.

- Déployer l'infrastructure cloud via un ARM template (azuredeploy.json) pour créer mon cluster AKS.

- Lire les secrets nécessaires depuis Azure Key Vault pour injecter dynamiquement toutes les connexions nécessaires à l'application.

- Créer les secrets Kubernetes (my-secret) contenant les informations sensibles pour que les conteneurs puissent les utiliser sans les exposer dans le code.

Il prépare l'environnement sécurisé et automatisé pour le déploiement de tous mes conteneurs dans le cluster AKS.

10-azure-pipeline:

Définition des variables :
J'ai configuré la variable AzureSubscription à azPipelines pour pointer vers ma connexion de service Azure DevOps qui donne accès à mon abonnement Azure.
J'ai aussi ajouté ma AzureDevOpsPAT (le Personal Access Token) pour que les commandes Azure CLI puissent manipuler les ressources DevOps.
Cela permet de lier Azure DevOps à Azure de façon sécurisée et de pouvoir

créer dynamiquement des connexions sans intervention manuelle, en respectant les exigences du devoir d'automatisation.

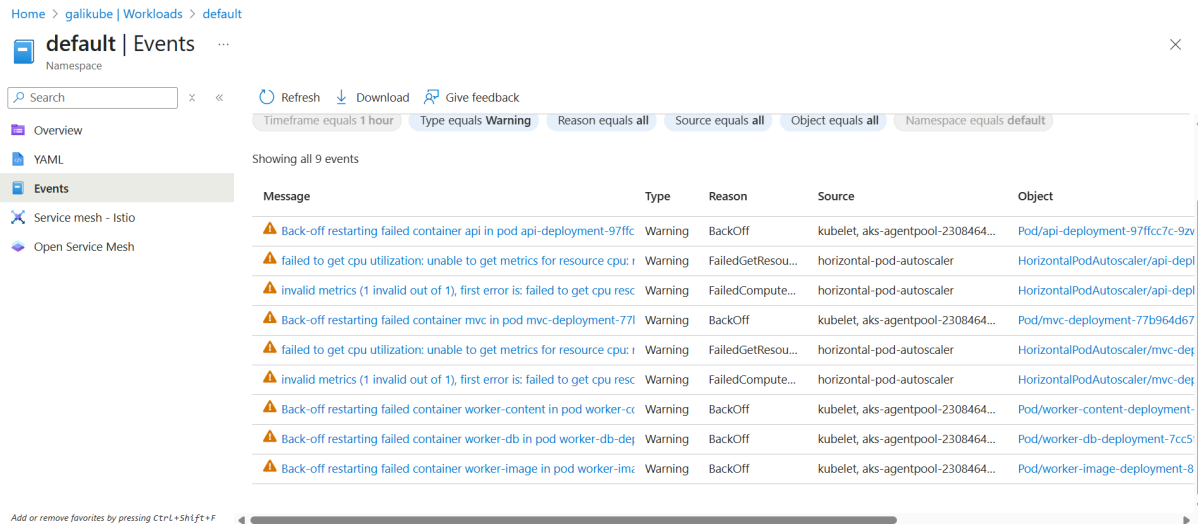
Grâce à ce pipeline, **on automatise** :

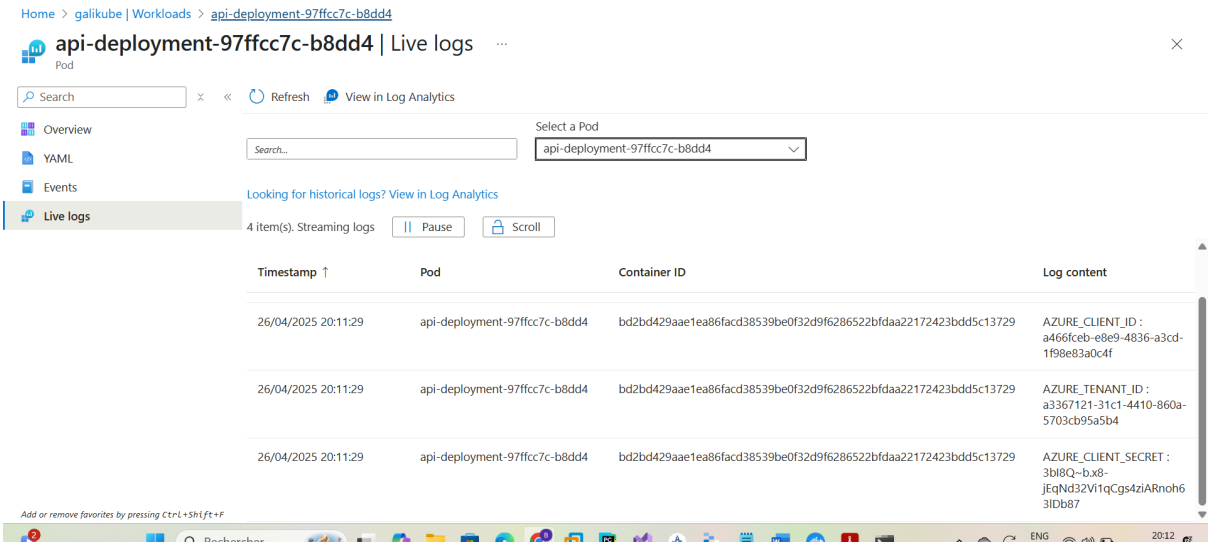
- La **création de l'identité Azure AD** (MVC_Dynamic),
- La **création des secrets** dans Key Vault,
- La **création des ressources cloud**,
- La **création dynamique d'une Service Connection** pour Docker Registry.

Cela rend l'infrastructure **prête pour déployer nos conteneurs de manière sécurisée et scalable**.

Problèmes rencontrés:

Dans la réalisation de ce tp, j'ai rencontré plusieurs crash et bugs notamment au niveau de lancer les 3 pipelines et le cluster.yml dans mon kubernetes(via apply a yaml) ainsi que d'autres. Pour les contourner, j'ai fait des recherches sur gpt et beaucoup de forum comme stackoverflow et le site officiel de microsoft education learn.
Pour le déploiement du cluster au début j'avais des erreurs de ce genre





et plein d'autres que j'ai reussi a resoudre

Conclusion

À travers ce projet, j'ai pu appliquer de manière concrète les principes de déploiement automatisé, de sécurisation de la configuration, et de scalabilité dynamique dans un environnement cloud Azure.
J ai réussi à :

- Construire une infrastructure complète à l'aide de ARM templates.
- Publier et déployer nos images conteneurisées sur Azure Kubernetes Service (AKS) via Azure Container Registry.
- Mettre en place un autoscaling grâce à Horizontal Pod Autoscaler pour les composants API et MVC, ainsi qu'à KEDA pour les workers basés sur Service Bus et Event Hub.
- Sécuriser intégralement les secrets et points de configuration via Azure Key Vault et App Configuration.
- Automatiser les étapes du processus grâce à Azure DevOps Pipelines.

Ce projet m'a permis de renforcer mes compétences en infrastructure as code, en orchestration de conteneurs et en pratiques DevOps.

Note: j'ai mis un dossier images(pour les captures).

A côté de tout ce que j’ai expliqué, j’ai créé aussi un agent (gaAgent) pour que je puisse rouler les pipelines