Projet Cloud & DevOps

Plateforme de diffusion de contenu statique cloud-native

Binôme – Azure for Students – AKS + GitHub Container Registry

1. Contexte industriel

Une équipe *Digital Platforms* doit concevoir et déployer une plateforme cloud-native permettant de diffuser dynamiquement du contenu statique (événements, actualités, FAQ) à destination :

- D'un site web public,
- D'applications clientes (mobile / frontend),
- De partenaires via API REST.

Le contenu est produit par une équipe éditoriale et stocké dans Azure Blob Storage sous forme de fichiers JSON/YAML.
La plateforme doit être déployée automatiquement, scalable, sécurisée, observable, et économique, dans un contexte Azure étudiant.

Vous êtes missionnés pour concevoir, implémenter, déployer et justifier cette solution.

À l'issue du projet, vous devrez démontrer votre capacité à :

- Concevoir une architecture cloud cohérente
- Conteneuriser une application (Docker)
- Déployer sur Azure Kubernetes Service (AKS)
- Automatiser via CI/CD (GitHub Actions)
- Gérer la sécurité, la scalabilité et l'observabilité
- Justifier vos choix techniques (coût, performance, contraintes)

2. Contraintes techniques imposées

- Compte Azure for Students
- Orchestration: Azure Kubernetes Service (AKS)
- Registry d'images : GitHub Container Registry (GHCR)
- Application backend : Flask
- Stockage des contenus : Azure Blob Storage
- CI/CD : GitHub Actions

3. Fonctionnalités attendues

Application Flask

L'application devra :

1. Lire des fichiers JSON et YAML depuis Azure Blob Storage
2. Exposer les endpoints REST suivants :

Endpoint	*Méthode*	*Description*
/api/events	GET	Retourne les événements
/api/news	GET	Retourne les actualités
/api/faq	GET	Retourne la FAQ
/healthz	GET	Vérification de vie
/readyz	GET	Vérification de disponibilité

3. Implémenter un cache mémoire avec TTL (ex : 60 s)
4. Fournir une interface web minimale pour visualiser les données

4. Gestion de version (Git)

- Dépôt GitHub structuré proprement
- README.md complet
- Fichier .gitignore adapté
- Stratégie Git justifiée :
  - Trunk-based ou
  - Git Flow

Questions à traiter

1. Pourquoi cette stratégie Git ?
2. Comment garantir un historique lisible et maintenable ?
3. Quels fichiers ne doivent jamais être versionnés ?

5. Conteneurisation (Docker)

- Dockerfile optimisé :
  - Image slim

- Utilisateur non-root
- Pas de dépendances inutiles

- Test local obligatoire

Questions à traiter

1. Comment réduire la taille de l'image ?
2. Pourquoi l'image Docker est-elle un *paquet binaire d'application* ?
3. Pourquoi le conteneur doit être stateless ?

6. Orchestration Kubernetes (AKS)

Manifests requis

- Namespace
- Deployment
- Service
- Ingress (NGINX recommandé)
- ConfigMap
- Secret (clé Blob ou équivalent)

Exigences

- requests / limits CPU et mémoire
- Readiness & liveness probes
- Rolling update sans interruption

Questions à traiter

1. Rôle de chaque ressource Kubernetes ?
2. Différence entre readiness et liveness ?
3. Impact des resources sur la scalabilité ?

7. CI/CD avec GitHub Actions

Pipeline automatisé déclenché à chaque push sur main :

1. Lint et tests
2. Build image Docker

3. Push image vers GHCR
4. Déploiement sur AKS
5. Smoke test post-déploiement

Questions à traiter

1. Pourquoi GHCR plutôt qu'Azure Container Registry ?
2. Comment gérer les secrets dans le pipeline ?
3. Quelle stratégie de rollback ?

8. Surveillance & journalisation (Azure Monitor – simple)

- Monitoring basique :
  - CPU / mémoire
  - Disponibilité du service
- Logs applicatifs niveau INFO
- 1 alerte simple (ex : erreurs 5xx)

Questions à traiter

1. Quelles métriques sont réellement utiles ?
2. Pourquoi éviter une journalisation excessive ?
3. Comment limiter les coûts Azure Monitor ?

9. Sécurité

- Accès Blob sécurisé :
  - Secret Kubernetes (minimum)
  - Bonus : Managed Identity
- Secrets non exposés dans le code
- Image Docker sécurisée (non-root)

Questions à traiter

1. Pourquoi ne pas stocker de secrets dans Git ?
2. Avantages Managed Identity vs clé statique ?
3. Risques de fuite dans les logs ?

10. Tests et assurance qualité

Démontrer que l'application :

- Fonctionne correctement de manière automatisée,
- Est reproductible (localement et dans la CI),
- Détecte immédiatement toute régression avant déploiement.

10.1 Exigences générales

- Écrire un jeu de tests automatisés pour l'application Flask ;
- Exécuter ces tests localement et dans le pipeline CI ;
- Démontrer le bon fonctionnement des tests lors de la soutenance.

Les tests doivent être :

- Rapides,
- Indépendants de l'environnement Azure,
- Reproductibles (sans dépendre d'un service externe actif).

10.2 Périmètre de test obligatoire

Les tests devront couvrir au minimum les éléments suivants :

a) Tests de santé (Health checks)

- Endpoint /healthz
- Endpoint /readyz

Critères de validation :

- Code HTTP 200,
- Réponse JSON valide,
- Présence d'un champ indiquant l'état du service.

b) Tests fonctionnels des endpoints API

Les endpoints suivants doivent être testés :

- /api/events
- /api/news
- /api/faq

Critè*res de validation :*

- Code HTTP 200,

- Réponse au format JSON,
- Structure stable (exemple : clé items contenant une liste).

11. Livrables attendus

Dépôt GitHub

- Code source complet.
- Manifests Kubernetes.
- Workflow CI/CD GitHub Actions.
- Fichier README.md documenté.

Rapport technique (PDF – 10 à 15 pages)

- Architecture globale (schémas).
- Justification des choix techniques.
- Réponses aux questions posées.
- Répartition du travail au sein du binôme.
- Retour d'expérience (difficultés rencontrées, limites, améliorations possibles).

Ordre imposé :

1. Architecture & repo
2. Flask local
3. Tests Flask
4. Docker
5. CI (sans AKS)
6. AKS
7. Monitoring & sécurité
8. Démo & rapport

On ne déploie jamais ce qui n'est pas testé…