

PhotoVerify

Systeme d'Authentification de Photos par QR Code

Rapport Technique de Projet

Deploiement Kubernetes sur Minikube

Projet Universitaire

Cloud Computing & Orchestration de Conteneurs

Janvier 2026

Contents

1	Introduction	2
2	Architecture Technique	2
2.1	Vue d'ensemble	2
2.2	Choix technologiques	2
2.3	Le flux de donnees	3
3	Implementation Kubernetes	3
3.1	Les Deployments	3
3.2	Les Services	4
3.3	La persistance des donnees	4
3.4	L'Ingress Controller	4
3.5	L'auto-scaling horizontal	5
4	Defis Rencontres et Solutions	5
4.1	Le probleme des fichiers statiques Next.js	5
4.2	La sensibilite a la casse de PostgreSQL	5
4.3	Les timeouts au demarrage	6
5	Resultats et Verification	6
5.1	Etat des ressources Kubernetes	6
5.2	Tests fonctionnels	6
6	Conclusion	7

1 Introduction

Ce projet est né d'une réflexion simple : comment démontrer concrètement l'intégration de plusieurs technologies cloud-native dans un contexte réaliste ? Plutôt que de créer un ennuyeux tutoriel "Hello World", nous avons choisi de construire une application complète qui résout un problème réel — la vérification d'authenticité de photos.

L'idée centrale de PhotoVerify est de permettre à un utilisateur d'uploader une photo, d'y associer des métadonnées (titre, date, description), et de générer automatiquement un QR code unique. Ce QR code, une fois scanné, renvoie vers une page de vérification qui prouve l'authenticité du document. C'est un cas d'usage concret qu'on retrouve dans les certificats numériques, les diplômes, ou les documents officiels.

Mais au-delà de la fonctionnalité, ce qui rend ce projet intéressant d'un point de vue technique, c'est son architecture. Nous avons volontairement choisi de ne pas nous arrêter à une simple application Docker. L'objectif était d'aller jusqu'au déploiement Kubernetes complet, avec tous les défis que cela implique : gestion des volumes persistants, configuration des services, mise en place d'un Ingress, et même auto-scaling horizontal.

Ce rapport documente non seulement ce que nous avons construit, mais surtout comment et pourquoi nous avons fait certains choix architecturaux. Les erreurs rencontrées et les solutions trouvées sont tout aussi instructives que le résultat final.

2 Architecture Technique

2.1 Vue d'ensemble

L'architecture de PhotoVerify suit un modèle classique mais robuste : un frontend qui gère l'interface utilisateur et les API, couplé à une base de données PostgreSQL pour la persistance. Ce qui la distingue, c'est son déploiement sur Kubernetes avec une séparation claire des responsabilités.

Le cluster Kubernetes héberge deux Deployments principaux. Le premier gère le frontend Next.js, qui sert à la fois les pages web et les endpoints API. Le second gère PostgreSQL, notre base de données relationnelle. Ces deux composants communiquent via des Services Kubernetes, ce qui nous permet de les scaler et les mettre à jour indépendamment.

2.2 Choix technologiques

Pourquoi Next.js ? Parce qu'il nous permet d'avoir le frontend et le backend dans le même projet, ce qui simplifie considérablement le déploiement. Les API Routes de Next.js gèrent toutes les opérations : upload de fichiers, génération de QR codes, interactions avec la

base de donnees.

Pourquoi PostgreSQL plutot que SQLite (qu'on utilisait en developpement) ? La migration vers PostgreSQL n'etait pas triviale, mais elle etait necessaire pour un environnement de production. SQLite pose des problemes de concurrence en environnement containerise, et PostgreSQL offre une bien meilleure robustesse pour les volumes persistants Kubernetes.

2.3 Le flux de donnees

Quand un utilisateur uploade une photo, voici ce qui se passe :

Premierement, le navigateur envoie la requete au Service NodePort qui ecoute sur le port 30000. Ce service route le trafic vers le pod frontend disponible.

Deuxiemement, l'API Route `/api/upload` recoit le fichier, le sauvegarde sur le PersistentVolume monte dans `/app/public/uploads`, genere un UUID unique, et cree le QR code.

Troisiemement, les metadonnees sont inserees dans PostgreSQL via le Service ClusterIP interne, qui n'est accessible que depuis l'interieur du cluster.

Finalement, l'utilisateur recoit l'URL de verification et le QR code en base64.

Ce flux peut sembler simple, mais chaque etape a necessite une configuration precise dans Kubernetes pour fonctionner correctement.

3 Implementation Kubernetes

3.1 Les Deployments

Notre fichier `frontend-deployment.yaml` merite qu'on s'y attarde. Au-dela de la configuration basique, nous avons du resoudre plusieurs problemes concrets.

Le premier concernait les permissions de fichiers. Le conteneur Next.js tourne avec un utilisateur non-root (UID 1001) pour des raisons de securite. Mais quand on monte un PersistentVolume, il appartient par default a root. Solution : nous avons ajoute un initContainer qui s'execute avant le conteneur principal et ajuste les permissions avec `chown`.

Listing 1: InitContainer pour les permissions

```
1 initContainers:
2   - name: fix-permissions
3     image: busybox:1.36
4     command: ['sh', '-c', 'chown -R 1001:1001 /app/public/uploads']
5     volumeMounts:
```

```
6     - name: uploads-storage
7       mountPath: /app/public/uploads
8     securityContext:
9       runAsUser: 0
```

Le second probleme concernait les health checks. Next.js en mode production met quelques secondes a demarrer. Sans configuration appropriee, Kubernetes tuait le pod avant qu'il soit pret. Nous avons configure des probes avec des delais adaptes.

3.2 Les Services

Deux types de Services coexistent dans notre architecture.

Le **frontend-service** est de type NodePort. Il expose l'application sur le port 30000 du noeud, ce qui permet d'y acceder depuis l'exterieur du cluster. C'est la solution la plus simple pour Minikube, meme si en production on utiliserait plutot un LoadBalancer.

Le **postgres-service** est de type ClusterIP. Il n'est accessible que depuis l'interieur du cluster, ce qui est exactement ce qu'on veut pour une base de donnees. Aucune raison d'exposer PostgreSQL au monde exterieur.

3.3 La persistance des donnees

C'est probablement l'aspect le plus critique de notre deploiement. Sans PersistentVolumes, chaque redemarrage de pod perdrait toutes les donnees.

Nous avons configure deux volumes distincts :

Le premier, **postgres-pv**, avec 1Gi de stockage, heberge les donnees PostgreSQL. Sa politique de retention est "Retain", ce qui signifie que meme si on supprime le PVC, les donnees restent sur le disque.

Le second, **frontend-uploads-pv**, avec 500Mi, stocke les photos uploadées. Il est monte dans le pod frontend sur `/app/public/uploads`.

Cette separation nous permet de gerer independamment la base de donnees et les fichiers uploades, avec des strategies de backup differentes si necessaire.

3.4 L'Ingress Controller

Pour demontrer une configuration plus avancee, nous avons ajoute un Ingress NGINX. Au lieu d'accéder a l'application via `localhost:30000`, on peut maintenant utiliser le hostname `photoverify.local`.

L'Ingress agit comme un reverse proxy intelligent qui route les requetes vers le bon service en fonction du hostname ou du path. Dans notre cas, toutes les requetes vers `photoverify.local` sont dirigees vers le frontend.

3.5 L'auto-scaling horizontal

Le HorizontalPodAutoscaler (HPA) est configure pour surveiller l'utilisation CPU et memoire du deployment frontend. Si l'utilisation CPU depasse 70% ou la memoire 80%, Kubernetes cree automatiquement de nouveaux pods, jusqu'a un maximum de 5.

En pratique, avec notre charge de test, on reste a 1 replica. Mais la configuration est en place et fonctionnelle, prete a scaler si necessaire.

Table 1: Configuration du HPA

Parametre	Valeur
Replicas minimum	1
Replicas maximum	5
Seuil CPU	70%
Seuil Memoire	80%

4 Defis Rencontres et Solutions

4.1 Le probleme des fichiers statiques Next.js

Un des bugs les plus frustrants a ete la non-apparition des images uploadées. Tout fonctionnait : l'upload reussissait, le fichier etait bien sur le volume, mais l'image ne s'affichait pas dans le navigateur.

Après investigation, nous avons compris le probleme. Next.js en mode "standalone" (necessaire pour Docker) ne sert pas automatiquement les fichiers statiques depuis les volumes montes. Le repertoire `/app/public/uploads` existe bien, mais Next.js ne le connait pas au runtime.

La solution a ete de creer une API Route dediee `/api/uploads/[filename]` qui lit manuellement les fichiers depuis le volume et les renvoie avec les bons headers MIME. Pas elegant, mais efficace.

4.2 La sensibilite a la casse de PostgreSQL

En developpement avec SQLite, les noms de colonnes n'etaient pas sensibles a la casse. En production avec PostgreSQL, "createdAt" et "createdat" sont deux colonnes differentes.

Prisma genere des noms de colonnes en camelCase, mais PostgreSQL les convertit en minuscules sauf si on les quote. Nous avons du modifier toutes nos requetes SQL brutes pour utiliser des guillemets doubles autour des identifiants mixtes.

4.3 Les timeouts au demarrage

Le premier deploiement echouait systematiquement. Kubernetes marquait les pods comme "unhealthy" et les redemarrant en boucle.

Le probleme : nos liveness probes etaient trop agressives. Next.js en production met environ 10-15 secondes a demarrer, mais notre probe commencait a verifier apres 5 secondes seulement.

Solution : augmenter `initialDelaySeconds` a 30 secondes et `periodSeconds` a 10 secondes.

5 Resultats et Verification

Pour valider notre implementation, nous avons execute une serie de tests systematiques.

5.1 Etat des ressources Kubernetes

Tous les composants sont operationnels :

Table 2: Etat des ressources Kubernetes

Ressource	Nom	Statut
Pod	frontend-xxx	Running (1/1)
Pod	postgres-xxx	Running (1/1)
Deployment	frontend	1/1 Ready
Deployment	postgres	1/1 Ready
Service	frontend-service	NodePort:30000
Service	postgres-service	ClusterIP:5432
PVC	frontend-uploads-pvc	Bound (500Mi)
PVC	postgres-pvc	Bound (1Gi)
Ingress	photoverify-ingress	Active
HPA	frontend-hpa	CPU: 1%/70%

5.2 Tests fonctionnels

Nous avons verifie chaque endpoint :

La page d'accueil repond correctement avec un code 200. L'API `/api/photos` retourne la liste des photos au format JSON. L'upload via `/api/upload` cree bien un nouvel enregistrement avec son QR code. La verification via `/verify/[id]` affiche correctement les details de la photo.

Le test le plus important : supprimer le pod frontend et verifier que les fichiers uploades sont toujours presents apres le redemarrage. Ce test confirme que notre PersistentVolume fonctionne comme prevu.

6 Conclusion

Ce projet nous a permis d'explorer en profondeur l'écosystème Kubernetes, bien au-delà de ce qu'un simple tutoriel pourrait offrir. Les problèmes rencontrés — permissions de fichiers, service de fichiers statiques, sensibilité à la casse SQL — sont exactement le type de défis qu'on rencontre en production.

L'architecture finale est robuste : les données persistent, l'application peut scaler, et le déploiement est reproductible via les manifests YAML. Le code source complet est disponible sur GitHub à l'adresse suivante :

<https://github.com/BamlakT/PhotoVerify-K8s>

Plusieurs améliorations seraient envisageables pour aller plus loin : intégration d'un service mesh comme Istio pour le monitoring, mise en place de CI/CD avec GitHub Actions, ou déploiement sur un cluster cloud (GKE, EKS, AKS) pour tester le comportement en environnement réel.

Mais tel quel, ce projet démontre une compréhension solide des concepts fondamentaux : Deployments, Services, PersistentVolumes, Ingress, et HPA. C'est une base sur laquelle construire des applications cloud-native plus complexes.