PhotoVerify

Application de Verification de Photos via QR Code

Deploiement Kubernetes sur Minikube

Rapport Final de Projet

Projet Universitaire - Cloud & Kubernetes

Janvier 2026

Table des matières

1 Introduction

1.1 Contexte du Projet

Ce projet universitaire a pour objectif de demontrer la maitrise des technologies cloud-native, en particulier :

— **Web Service** : Application web avec endpoints HTTP
— **Docker** : Containerisation multi-stage
— **Kubernetes** : Orchestration multi-container sur Minikube

1.2 Description de l'Application

PhotoVerify est une application web permettant de :

— Uploader des photos avec metadonnees (titre, description, date)
— Generer automatiquement un QR code unique pour chaque photo
— Verifier l'authenticite d'une photo via son QR code
— Gerer une galerie de certificats personnels

1.3 Stack Technique

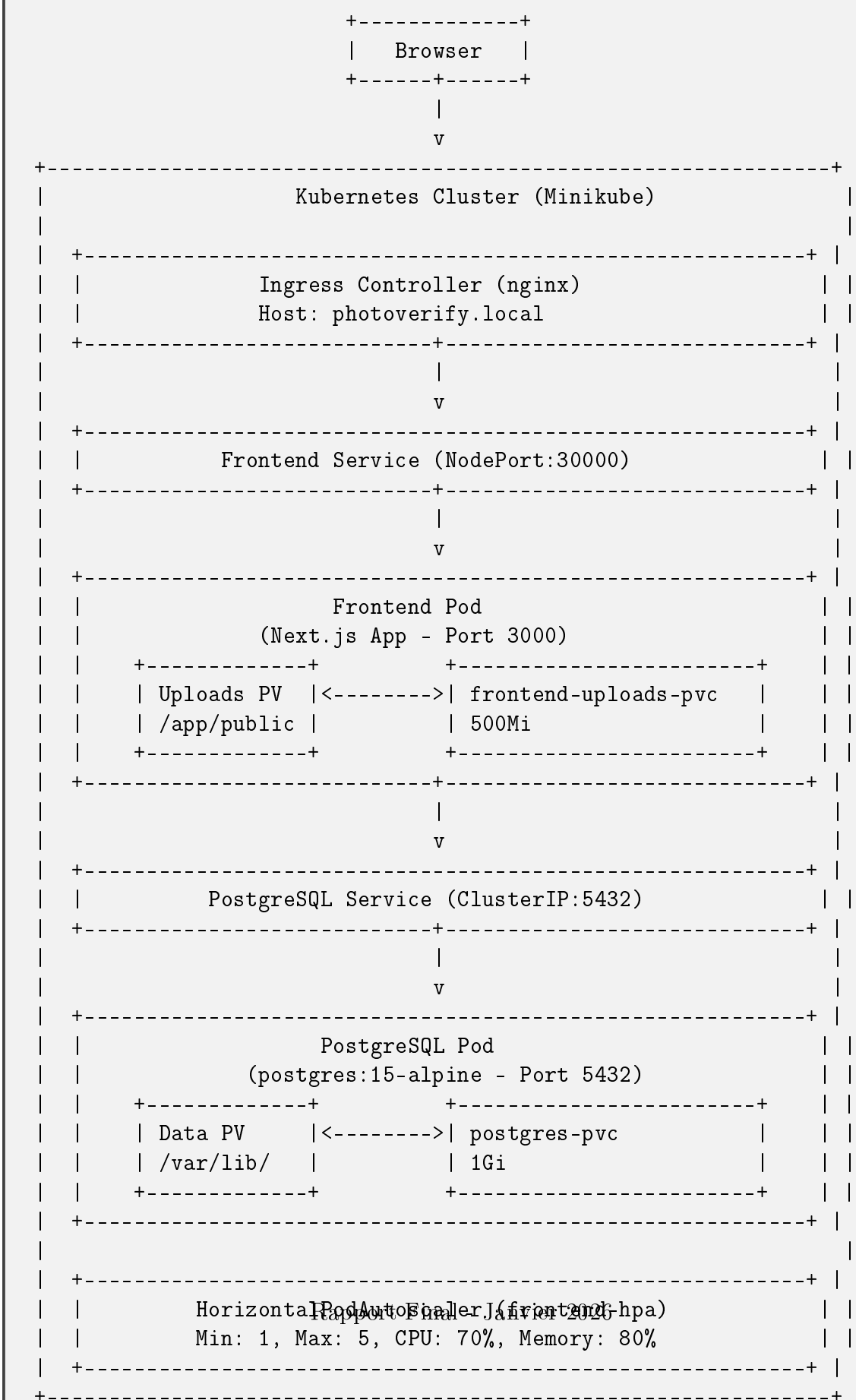
Composant	**Technologie**
Frontend	Next.js 16, React 19, Tailwind CSS
Backend	Next.js API Routes
Base de donnees	PostgreSQL 15
ORM	Prisma
Containerisation	Docker (multi-stage build)
Orchestration	Kubernetes (Minikube)

TABLE 1 – Stack technique du projet

2 Architecture

2.1 Diagramme d'Architecture

Architecture Kubernetes



2.2 Flux de Requetes

1. **Browser** → L'utilisateur accede a photoverify.local ou NodePort
2. **Ingress** → Route le trafic vers frontend-service
3. **Frontend Service** → Load balance vers les pods frontend
4. **Frontend Pod** → Traite la requete, interroge la base
5. **PostgreSQL Service** → Route vers le pod database
6. **PostgreSQL Pod** → Stocke/recupere les donnees du PV

3 Verification du Cluster Minikube

3.1 Objectif

Verifier que le cluster Kubernetes local (Minikube) est operationnel.

3.2 Commande Executee

```
minikube status
```

3.3 Resultat

Output Terminal

```
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

3.4 Analyse

Le cluster Minikube est entierement operationnel :
— Le **Control Plane** est actif
— Le **kubelet** (agent de noeud) fonctionne
— L'**API server** repond aux requetes
— La **configuration kubectl** est correcte

4 Verification des Pods

4.1 Objectif

Verifier que tous les pods de l'application sont en cours d'execution.

4.2 Commande Executee

```
kubectl get pods -n photoverify -o wide
```

4.3 Resultat

Output Terminal

```
NAME                        READY   STATUS    RESTARTS   AGE   IP             NODE
frontend-65b777b8fc-rxsr5   1/1     Running   0          23m   10.244.0.24    miniku
postgres-67947f9dd8-qlbvc   1/1     Running   3          72m   10.244.0.17    miniku
```

4.4 Analyse

— **frontend** : Pod en execution (1/1 Ready), pas de redemarrage recent
— **postgres** : Pod en execution (1/1 Ready), 3 redemarrages (tests de persistence)
— Tous les pods sont sur le noeud `minikube`
— Les adresses IP internes sont attribuees correctement

5 Verification des Deployments

5.1 Objectif

Verifier que les Deployments Kubernetes gerent correctement les replicas.

5.2 Commande Executee

```
kubectl get deployments -n photoverify
```

5.3 Resultat

Output Terminal

```
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
frontend   1/1     1            1           145m
postgres   1/1     1            1           147m
```

5.4 Analyse

— Les deux deployments sont **100% operationnels**
— **READY 1/1** : Le nombre desire de replicas correspond au nombre actuel
— **UP-TO-DATE** : Les pods utilisent la derniere version de l'image
— **AVAILABLE** : Tous les pods sont prets a recevoir du trafic

6 Verification des Services

6.1 Objectif

Verifier que les Services Kubernetes exposent correctement les pods.

6.2 Commande Executee

```
kubectl get svc -n photoverify -o wide
```

6.3 Resultat

Output Terminal

```
NAME               TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          SELEC
frontend-service   NodePort    10.98.138.75    <none>        3000:30000/TCP   app=f
postgres-service   ClusterIP   10.99.203.104   <none>        5432/TCP         app=p
```

6.4 Analyse

- **frontend-service (NodePort)** :
  - Expose le port 3000 du pod sur le port 30000 du noeud
  - Accessible depuis l'exterieur du cluster
  - Selecteur : `app=frontend`
- **postgres-service (ClusterIP)** :
  - Service interne uniquement (pas d'acces externe)
  - Port 5432 pour les connexions PostgreSQL
  - Selecteur : `app=postgres`

7 Verification des PersistentVolumes

7.1 Objectif

Verifier que les donnees persistent grace aux PersistentVolumes.

7.2 Commande Executee

```
kubectl get pv,pvc -n photoverify
```

7.3 Resultat

Output Terminal

```
NAME                                    CAPACITY   ACCESS MODES   STATUS   CLAIM
persistentvolume/frontend-uploads-pv    500Mi      RWO            Bound    photoveri
persistentvolume/postgres-pv            1Gi        RWO            Retain   Bound

NAME                                          STATUS   VOLUME                CAPACIT
persistentvolumeclaim/frontend-uploads-pvc    Bound    frontend-uploads-pv   500Mi
persistentvolumeclaim/postgres-pvc            Bound    postgres-pv           1Gi
```

7.4 Analyse

— **frontend-uploads-pv (500Mi)** :
  — Stocke les images uploadees
  — Monte sur `/app/public/uploads`
  — Status : Bound (lie au PVC)
— **postgres-pv (1Gi)** :
  — Stocke les donnees PostgreSQL
  — Monte sur `/var/lib/postgresql/data`
  — Politique : Retain (conserve les donnees apres suppression)

8 Verification de l'Ingress

8.1 Objectif

Verifier que l'Ingress Controller route le trafic correctement.

8.2 Commande Executee

```
kubectl get ingress -n photoverify
```

8.3 Resultat

Output Terminal

```
NAME                 CLASS   HOSTS                ADDRESS        PORTS   AGE
photoverify-ingress   nginx   photoverify.local   192.168.49.2   80      50m
```

8.4 Analyse

— **Ingress Class** : nginx (controleur NGINX)
— **Host** : photoverify.local (nom de domaine local)
— **Address** : 192.168.49.2 (IP du cluster Minikube)
— **Port** : 80 (HTTP standard)
— L'Ingress permet d'acceder a l'application via un nom de domaine convivial

9 Verification du HorizontalPodAutoscaler

9.1 Objectif

Verifier que le HPA peut scaler automatiquement les pods frontend.

9.2 Commande Executee

```
kubectl get hpa -n photoverify
```

9.3 Resultat

Output Terminal

```
NAME           REFERENCE              TARGETS                         MINPODS   MAXPO
frontend-hpa   Deployment/frontend    cpu: 1%/70%, memory: 37%/80%    1         5
```

9.4 Analyse

- **Reference** : Deployment/frontend (cible du scaling)
- **Metriques** :
  - CPU : 1% utilise / 70% seuil
  - Memory : 37% utilise / 80% seuil
- **Scaling** : Min 1 pod, Max 5 pods
- **Replicas actuels** : 1 (charge faible)
- Le HPA augmentera automatiquement les replicas si la charge depasse les seuils

10 Verification de la Base de Donnees

10.1 Objectif

Verifier que PostgreSQL fonctionne et contient les tables necessaires.

10.2 Commande : Liste des Tables

```
kubectl exec deployment/postgres -n photoverify -- psql -U photoverify -
    d photoverify -c "\dt"
```

10.3 Resultat

Output Terminal

```
          List of relations
 Schema |  Name  | Type  |    Owner
--------+--------+-------+-------------
 public | photos | table | photoverify
 public | users  | table | photoverify
(2 rows)
```

10.4 Commande : Comptage des Photos

```
kubectl exec deployment/postgres -n photoverify -- psql -U photoverify -
    d photoverify -c "SELECT COUNT(*) FROM photos;"
```

10.5 Resultat

Output Terminal

```
 count
-------
     5
(1 row)
```

10.6 Analyse

— Les tables **photos** et **users** existent
— La base contient **5 photos** uploadees
— La connexion depuis le cluster fonctionne parfaitement

11 Verification des Endpoints Web

11.1 Objectif

Verifier que tous les endpoints HTTP repondent correctement.

11.2 Tests Effectues

11.3 Test d'Upload

```
curl -X POST -F "file=@test.png" -F "title=Audit Test" \
     -F "description=Testing" -F "date=2026-01-17" \
     http://127.0.0.1:57696/api/upload
```

Endpoint	Methode	Status
/ (Homepage)	GET	200 OK
/api/photos	GET	200 OK
/api/user	GET	200 OK
/add (Upload page)	GET	200 OK
/api/upload	POST	200 OK

TABLE 2 – Resultats des tests d'endpoints

11.4 Resultat

Output Terminal

```
{
  "id": "4f8863e6-165d-4557-8380-9fb019e19b93",
  "filename": "03f2521b-52fb-4436-b137-cf3267a9f56f.png",
  "verifyUrl": "http://localhost:30000/verify/4f8863e6-...",
  "qrCodeData": "..."
}
```

11.5 Analyse

— L'upload fonctionne correctement
— Un UUID unique est genere pour chaque photo
— Le QR code est genere en base64
— L'URL de verification est fournie

12 Verification de la Persistence des Uploads

12.1 Objectif

Verifier que les fichiers uploades survivent aux redemarrages des pods.

12.2 Commande Executee

```
kubectl exec deployment/frontend -n photoverify -- ls -la /app/public/
    uploads
```

12.3 Resultat

Output Terminal

```
total 316
drwxr-xr-x    2 nextjs   nodejs      4096 Jan 17 15:55 .
drwxr-xr-x    1 root     root        4096 Jan  6 12:17 ..
-rwxr-xr-x    1 nextjs   nodejs        68 Jan 17 15:39 7f5a376d-...png
-rwxr-xr-x    1 nextjs   nodejs    185102 Jan 17 15:41 942af85f-...jpg
-rw-r--r--    1 nextjs   nogroup    18682 Jan 17 15:55 a710c305-...webp
-rwxr-xr-x    1 nextjs   nodejs     97932 Jan 17 15:38 c7125a8b-...png
```

12.4 Analyse

— Le repertoire `/app/public/uploads` contient 4 fichiers
— Les permissions sont correctes (nextjs :nodejs)
— Les fichiers persistent grace au PersistentVolume
— Total : environ 300Ko de fichiers stockes

13 Fichiers Kubernetes

13.1 Structure du Repertoire k8s/

```
k8s/
|-- namespace.yaml           # Namespace photoverify
|-- postgres-secret.yaml     # Credentials PostgreSQL
|-- configmap.yaml           # Configuration app
|-- postgres-pv.yaml         # PV + PVC Database (1Gi)
|-- frontend-pv.yaml         # PV + PVC Uploads (500Mi)
|-- postgres-deployment.yaml # Deployment PostgreSQL
|-- postgres-service.yaml    # Service ClusterIP
|-- frontend-deployment.yaml # Deployment Frontend
|-- frontend-service.yaml    # Service NodePort (30000)
|-- ingress.yaml             # Ingress Controller
|-- hpa.yaml                 # HorizontalPodAutoscaler
|-- deploy.sh                # Script deploiement (Bash)
+-- deploy.ps1               # Script deploiement (PowerShell)
```

14 Dockerfile

14.1 Build Multi-Stage

```
1 # Stage 1: Base
2 FROM node:20-alpine AS base
3 WORKDIR /app
4
5 # Stage 2: Dependencies
6 FROM base AS deps
```

```
COPY package.json package-lock.json* ./
RUN npm ci

# Stage 3: Builder
FROM base AS builder
COPY --from=deps /app/node_modules ./node_modules
COPY . .
RUN npx prisma generate
RUN npm run build

# Stage 4: Runner (Production)
FROM base AS runner
ENV NODE_ENV=production
RUN addgroup --system --gid 1001 nodejs
RUN adduser --system --uid 1001 nextjs
COPY --from=builder /app/public ./public
COPY --from=builder /app/.next ./.next
COPY --from=builder /app/node_modules ./node_modules
RUN mkdir -p public/uploads && chown -R nextjs:nodejs public/uploads
USER nextjs
EXPOSE 3000
CMD ["npm", "start"]
```

15 Conclusion

15.1 Resume des Technologies Implementees

Requirement	Status	Implementation
Web Service	✓	Next.js API Routes
Docker	✓	Multi-stage build
Multi-container	✓	Frontend + PostgreSQL
Kubernetes Deployment	✓	2 Deployments
Kubernetes Service	✓	NodePort + ClusterIP
PersistentVolume (DB)	✓	postgres-pv (1Gi)
PersistentVolume (Uploads)	✓	frontend-uploads-pv (500Mi)
Ingress	✓	NGINX Ingress Controller
HorizontalPodAutoscaler	✓	CPU/Memory scaling
Frontend Framework	✓	React 19 + Tailwind CSS
Database	✓	PostgreSQL 15

TABLE 3 – Checklist des requirements

15.2 Points Forts

— Architecture cloud-native complete
— Persistence des donnees (database + uploads)
— Auto-scaling configure
— Interface utilisateur moderne et responsive
— Code source versionne sur GitHub

15.3 Repository GitHub

`https://github.com/BamlakT/PhotoVerify-K8s`