

Rapport de TP 1 - De Local à Global : Déploiement d'un microservice conteneurisé en Serverless

Section 1 : Introduction

Dans ce premier TP, nous avons exploré le déploiement d'un microservice conteneurisé en utilisant une architecture serverless (minikube). L'objectif principal était de comprendre les concepts fondamentaux du serverless, ainsi que les avantages et les défis associés à cette approche.

Section 2 : Préparation de l'environnement

N'ayant pas de crédit GCP, nous avons décidé de monter un cluster mono node minikube en local sur nos machines. Nous avons donc installé Minikube et kubectl, puis nous avons démarré le cluster avec la commande `minikube start`.

Installation de docker :

```
sudo apt-get update
sudo apt-get install \
    ca-certificates \
    curl \
    gnupg \
    lsb-release
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o
/usr/share/keyrings/docker-archive-keyring.gpg
echo "\"deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-
keyring.gpg] https://download.docker.com/linux/ubuntu \
    $(lsb_release -cs) stable\" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Vérification de l'installation :

```
id001@id001-ThinkPad-T470:~/Nextcloud/BUT3/Cloud/rapport/tp1$ docker --version
Docker version 28.2.2, build 28.2.2-0ubuntu1~22.04.1
```

Installation de minikube :

```
curl -LO https://github.com/kubernetes/minikube/releases/latest/download/minikube-linux-
amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube && rm minikube-linux-amd64
```

Lancement du cluster :

```
minikube start --driver=docker
```

Section 3 : Développement du microservice

Nous avons suivis les instructions du TP pour créer un microservice simple en Python utilisant Flask. Le code source du microservice est le suivant :

package.json :

```
{
  "name": "pizza-api",
  "version": "1.0.0",
  "description": "API commande pizza serverless",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "dev": "node index.js"
  },
  "dependencies": {
    "express": "^4.18.2"
  },
  "engines": {
    "node": "18"
  }
}
```

Et l'index.js :

```
const express = require('express');
const app = express();

// Default 8080 si env vide (Cloud Run injecte PORT=8080)
const port = process.env.PORT || 8080;

// Middleware
app.use(express.json());
app.use((req, res, next) => {
  console.log(`[${new Date().toISOString()}] ${req.method} ${req.path}`);
  next();
});

// Route (healthcheck Cloud Run)
app.get('/', (req, res) => {
  res.json({
    status: 'ok',
    service: 'Pizza-Tech API v1.0',
    timestamp: new Date().toISOString()
  });
});

// Menu endpoint
app.get('/menu', (req, res) => {
  const menu = [
    { id: 1, name: 'Margherita', price: 12.99 },
```

```

    { id: 2, name: 'Pepperoni', price: 14.99 },
    { id: 3, name: 'Veggie', price: 13.99 },
    { id: 4, name: 'Quattro Formaggi', price: 15.99 }
  ];
  res.json(menu);
});

// Commande endpoint
app.post('/order', (req, res) => {
  const { pizzaId, quantity, address } = req.body;

  // Validation simple (Facteur VIII)
  if (!pizzaId || !quantity || !address) {
    return res.status(400).json({ error: 'Missing fields' });
  }

  // Simulation traitement
  const orderId = Math.random().toString(36).substring(7);
  res.json({
    orderId: orderId,
    status: 'confirmed',
    estimatedDelivery: '30 minutes'
  });
});

app.use((req, res) => {
  res.status(404).json({ error: 'Not found' });
});

app.listen(port, () => {
  console.log(`[STARTUP] Pizza-Tech API listening on port ${port}`);
  console.log(`[STARTUP] Health check: GET http://localhost:${port}/`);
});

// Gestion de la fermeture de l'application (Facteur IX 12-Factor)
process.on('SIGTERM', () => {
  console.log('[SHUTDOWN] SIGTERM received, closing gracefully...');
  process.exit(0);
});

```

Construction et installation des dépendances :

```
npm install
```

Et lancement de l'appli en local pour tester :

```
node start
```

Et on lance des requêtes curl pour tester les endpoints :

```

curl http://localhost:8080/
# {"status": "ok", "service": "Pizza-Tech API v1.0", "timestamp": "2024-06-10T12:34:56.789Z"}

curl http://localhost:8080/menu
#[{"id":1,"name":"Margherita","price":12.99},{ "id":2,"name":"Pepperoni","price":14.99},
{"id":3,"name":"Veggie","price":13.99},{ "id":4,"name":"Quattro Formaggi","price":15.99}]

curl -X POST http://localhost:8080/order \
  -H "Content-Type: application/json" \
  -d '{"pizzaId":1,"quantity":2,"address":"123 Rue de la Paix"}'
#{ "orderId":"hfd84p", "status":"confirmed", "estimatedDelivery":"30 minutes"}

```

Et on voit les call api dans les logs :

```

id001@id001-ThinkPad-T470:~/Nextcloud/BUT3/Cloud/rapport/tp1/pizza-api$ npm start
> pizza-api@1.0.0 start
> node index.js

[STARTUP] Pizza-Tech API listening on port 8080
[STARTUP] Health check: GET http://localhost:8080/
[2026-01-12T08:17:13.874Z] GET /
[2026-01-12T08:17:27.017Z] GET /menu
[2026-01-12T08:17:39.057Z] POST /order
[2026-01-12T08:17:56.831Z] GET /
[2026-01-12T08:17:56.865Z] GET /menu
[2026-01-12T08:17:56.883Z] POST /order
[2026-01-12T08:19:47.007Z] GET /
[2026-01-12T08:20:00.305Z] GET /menu
[2026-01-12T08:20:26.216Z] POST /order

```

Section 4 : Conteneurisation avec Docker

On va créer une image Docker pour notre microservice. Voici le Dockerfile utilisé :

```

# Stage 1: Build & Audit
FROM node:18-slim AS builder
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm ci --only=production \
  && npm audit --audit-level=high --production
COPY . .

# Stage 2: Production
FROM node:18-slim AS production
WORKDIR /usr/src/app
COPY --from=builder /usr/src/app ./
RUN npm prune --production \
  && rm -rf /usr/src/app/tests /usr/src/app/.git /usr/src/app/*.md
EXPOSE 8080
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
  CMD node -e "require('http').get('http://localhost:8080', (r) => {if (r.statusCode !==
200) throw new Error(r.statusCode)})"
CMD [ "npm", "start" ]

```

On ajoute un `.dockerignore` pour éviter de copier les fichiers inutiles dans l'image Docker :

```
node_modules
npm-debug.log
.git
.gitignore
README.md
.env
.env.local
.env.example
.DS_Store
deployment/
DEPLOYMENT.md
*.sh
```

Construction de l'image Docker :

```
docker build -t pizza-api:latest .
```

Lancement du conteneur Docker en local pour tester :

```
id001@id001-ThinkPad-T470:~/Nextcloud/BUT3/Cloud/rapport/tp1/pizza-api$ docker build -t pizza-api:local .
[+] Building 4.4s (11/11) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 374B
=> [internal] load metadata for docker.io/library/node:18-slim
=> [auth] library/node:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> => transferring context: 114B
=> [1/5] FROM docker.io/library/node:18-slim@sha256:f9ab18e354e6855ae56ef2b290dd225c1e51a564f87584b9bd21dd651838830e
=> [internal] load build context
=> => transferring context: 508B
=> CACHED [2/5] WORKDIR /usr/src/app
=> CACHED [3/5] COPY package*.json ./
=> CACHED [4/5] RUN npm install --only=production
=> [5/5] COPY . .
=> exporting to image
=> => exporting layers
=> => writing image sha256:eb26efcd27287c27eccb45a47b3dfde9d62cf5b33241cc93c6b22fd07a14ec1a
=> => naming to docker.io/library/pizza-api:local
id001@id001-ThinkPad-T470:~/Nextcloud/BUT3/Cloud/rapport/tp1/pizza-api$ docker run -p 8080:8080 pizza-api:local
> pizza-api@1.0.0 start
> node index.js

[STARTUP] Pizza-Tech API listening on port 8080
[STARTUP] Health check: GET http://localhost:8080/
[2026-01-12T08:30:55.306Z] GET /
```

Section 5 : Déploiement sur Minikube à l'aide d'un déploiement Kubernetes

On crée un fichier de déploiement Kubernetes `deployment.yaml` avec un Service pour exposer l'API :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pizza-api-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: pizza-api
```

```

template:
  metadata:
    labels:
      app: pizza-api
  spec:
    containers:
      - name: pizza-api
        image: pizza-api:local
        ports:
          - containerPort: 8080
        readinessProbe:
          httpGet:
            path: /
            port: 8080
          initialDelaySeconds: 5
          periodSeconds: 10
---
apiVersion: v1
kind: Service
metadata:
  name: pizza-api-service
spec:
  selector:
    app: pizza-api
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
      nodePort: 30080
  type: NodePort

```

Vous pouvez ensuite tester l'API sur `http://192.168.49.2:30080`.

L'ip peut être obtenue avec la commande `minikube ip`.

Et le port est celui défini dans le service (30080).

On applique le déploiement sur Minikube :

```
kubectl apply -f deployment/pizza-deployment.yaml
```

On voit que le pod est bien créé et en cours d'exécution :

```

id001@id001-ThinkPad-T470:~/Nextcloud/BUT3/Cloud/rapport/tp1/pizza-api$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
pizza-api-deployment-6485b5594b-96l5j  1/1     Running   0           46s
id001@id001-ThinkPad-T470:~/Nextcloud/BUT3/Cloud/rapport/tp1/pizza-api$

```

A cette étape, nous avons donc une API exposée sur l'interface WAN du noeud minikube, l'api est accessible :

<http://192.168.49.2:30080>

Section 7 : Démonstration de l'élasticité et de la scalabilité

Afin de démontrer la scalabilité nous allons modifier le micro service de la manière suivante :

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: pizza-api-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: pizza-api
  template:
    metadata:
      labels:
        app: pizza-api
    spec:
      containers:
        - name: pizza-api
          image: pizza-api:local
          ports:
            - containerPort: 8080
          resources:
            requests:
              cpu: "100m"
              memory: "128Mi"
            limits:
              cpu: "500m"
              memory: "256Mi"
          readinessProbe:
            httpGet:
              path: /
              port: 8080
            initialDelaySeconds: 5
            periodSeconds: 10

```

```
---
```

```

apiVersion: v1
kind: Service
metadata:
  name: pizza-api-service
spec:
  selector:
    app: pizza-api
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
      nodePort: 30080
  type: NodePort

```

```
---
```

```

apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: pizza-api-hpa
spec:

```

```

scaleTargetRef:
  apiVersion: apps/v1
  kind: Deployment
  name: pizza-api-deployment
minReplicas: 1
maxReplicas: 5
metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50

```

- Nous avons ajouté un HorizontalPodAutoscaler (HPA) pour permettre au déploiement de scaler automatiquement entre 1 et 5 réplicas en fonction de l'utilisation du CPU.
- Nous avons défini une métrique basée sur l'utilisation du CPU avec un seuil de 50%.
On applique le nouveau déploiement avec l'HPA :

```
kubectl apply -f deployment/pizza-deployment.yaml
```

Afin de vérifier le bon fonctionnement du HPA, nous avons utilisés les méthodes suivantes :

Terminal 1 : Générateur de trafic simple avec bash

Pour générer du trafic en parallèle et tester la scalabilité, vous pouvez utiliser le script suivant qui lance plusieurs processus curl en arrière-plan :

```

#!/bin/bash
# Génère du trafic parallèle vers l'API /menu

PARALLEL=10 # Nombre de processus asynchrone

for i in $(seq 1 $PARALLEL); do
  while true; do
    curl -s http://192.168.49.2:30080/menu > /dev/null
    sleep 0.1
  done &
done

echo "$PARALLEL générateurs de trafic lancés en arrière-plan."
echo "Utilisez 'pkill curl' pour arrêter tous les processus."

```

Ce script lance 10 boucles curl en parallèle. Vous pouvez ajuster la variable `PARALLEL` selon vos besoins.

Terminal 2 : Alternative avec Apache Benchmark (ab)

Vous pouvez également utiliser l'outil Apache Benchmark (ab) pour générer du trafic de manière plus contrôlée. Voici un exemple de commande ab :

```
ab -n 1000 -c 50 http://192.168.49.2:30080/menu
```


Toutes les 1,0s: `kubectl get pods`

NAME	READY	STATUS	RESTARTS	AGE
pizza-api-deployment-56f6fd85b8-djrx8	1/1	Running	0	6m14s
pizza-api-deployment-56f6fd85b8-v9zw9	1/1	Running	0	88s
pizza-api-deployment-56f6fd85b8-xtlgt	1/1	Running	0	3m28s
pizza-api-deployment-56f6fd85b8-zkl74	1/1	Running	0	88s

Section 8 : Amélioration de la sécurité et de la configuration

8.1 Externalisation complète de la configuration (12-Factor App - Facteur III)

Pour atteindre le niveau max de la notation, nous avons complètement externalisé la configuration de l'application. Toutes les valeurs configurables sont maintenant gérées via des variables d'environnement.

Fichier `.env` contenant les variables d'environnements

```

1  # Configuration de l'API Pizza-Tech
2  # Ce fichier est pour le développement local uniquement
3
4  # Configuration générale
5  PORT=8080
6  SERVICE_NAME=Pizza-Tech API
7  SERVICE_VERSION=2.0
8  NODE_ENV=development
9  DEFAULT_DELIVERY_TIME=30 minutes
10
11 # Secrets (valeurs de développement)
12 API_KEY=dev-api-key-local
13 DB_CONNECTION_STRING=mongodb://localhost:27017/pizzadb-dev
14

```

Code mis à jour (index.js)

```

const express = require('express');
const app = express();

// ===== Configuration via Variables d'Environnement (Facteur III 12-Factor) =====
const config = {
  port: process.env.PORT || 8080,
  serviceName: process.env.SERVICE_NAME || 'Pizza-Tech API',
  serviceVersion: process.env.SERVICE_VERSION || '1.0',
  nodeEnv: process.env.NODE_ENV || 'production',
  deliveryTime: process.env.DEFAULT_DELIVERY_TIME || '30 minutes',
  // Secrets (ne jamais hardcoder en production)
  apiKey: process.env.API_KEY || '',
  dbConnectionString: process.env.DB_CONNECTION_STRING || ''

```

```
};
```

ConfigMap Kubernetes (configmap.yaml)

Création d'un ConfigMap pour gérer les variables de configuration non-sensibles :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: pizza-api-config
data:
  SERVICE_NAME: "Pizza-Tech API"
  SERVICE_VERSION: "2.0"
  NODE_ENV: "production"
  DEFAULT_DELIVERY_TIME: "30 minutes"
```

Secret Kubernetes (secret.yaml)

Création d'un Secret pour les données sensibles (API keys, credentials DB, etc.) :

```
apiVersion: v1
kind: Secret
metadata:
  name: pizza-api-secret
type: Opaque
stringData:
  # ATTENTION : En production, utiliser un Secrets Manager externe (Vault, GCP Secret
  Manager)
  API_KEY: "demo-api-key-12345"
  DB_CONNECTION_STRING: "mongodb://user:pass@localhost:27017/pizzadb"
```

8.2 Mise en place du HTTPS obligatoire

Pour sécuriser les communications, nous avons configuré un Ingress Kubernetes avec TLS.

Modification du Service

Le Service est maintenant de type `ClusterIP` au lieu de `NodePort`, car le trafic passera par l'Ingress (reverse proxy).

```

apiVersion: v1
kind: Service
metadata:
  name: pizza-api-service
spec:
  selector:
    app: pizza-api
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  type: ClusterIP

```

Configuration de l'Ingress avec TLS (ingress.yaml)

```

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: pizza-api-ingress
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
    nginx.ingress.kubernetes.io/force-ssl-redirect: "true"
spec:
  ingressClassName: nginx
  tls:
    - hosts:
        - pizza-api.local
      secretName: pizza-tls-secret
  rules:
    - host: pizza-api.local
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: pizza-api-service
                port:
                  number: 80

```

Fonctionnalités :

HTTPS obligatoire (redirection automatique HTTP → HTTPS | Certificat TLS stocké dans un Secret Kubernetes |
 solution réseau via ClusterIP | Gestion centralisée du trafic via Ingress

Génération du certificat TLS

Pour le développement local, un certificat auto-signé est généré via OPENSSL :

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
  -keyout /tmp/tls.key -out /tmp/tls.crt \
  -subj "/CN=pizza-api.local/O=pizza-api" \
  -addext "subjectAltName = DNS:pizza-api.local"

kubectl create secret tls pizza-tls-secret \
  --key /tmp/tls.key \
  --cert /tmp/tls.crt
```

Scan de sécurité

Le build inclut automatiquement un audit de sécurité, on peut également utiliser Trivy mais dans le projet il n'est pas déployé :

```
npm audit --audit-level=high --production
```

8.4 Mise à jour du déploiement Kubernetes

Le fichier `pizza-deployment.yaml` a été mis à jour pour intégrer les ConfigMaps et Secrets :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pizza-api-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: pizza-api
  template:
    metadata:
      labels:
        app: pizza-api
    spec:
      containers:
        - name: pizza-api
          image: pizza-api:local
          ports:
            - containerPort: 8080
          # Injection des variables d'environnement depuis ConfigMap
          envFrom:
            - configMapRef:
                name: pizza-api-config
          # Injection des secrets
          env:
            - name: API_KEY
              valueFrom:
                secretKeyRef:
                  name: pizza-api-secret
                  key: API_KEY
            - name: DB_CONNECTION_STRING
```

```

    valueFrom:
      secretKeyRef:
        name: pizza-api-secret
        key: DB_CONNECTION_STRING
  resources:
    requests:
      cpu: "100m"
      memory: "128Mi"
    limits:
      cpu: "500m"
      memory: "256Mi"
  readinessProbe:
    httpGet:
      path: /
      port: 8080
    initialDelaySeconds: 5
    periodSeconds: 10
  livenessProbe:
    httpGet:
      path: /
      port: 8080
    initialDelaySeconds: 15
    periodSeconds: 20

```

8.6 Procédure de redéploiement

Option 1 : Déploiement manuel (étape par étape)

```

# 1. Builder l'image
eval $(minikube docker-env)
docker build -t pizza-api:v2.0 .

# 2. Activer les addons
minikube addons enable ingress
minikube addons enable metrics-server

# 3. Appliquer les ressources
kubectl apply -f deployment/configmap.yaml
kubectl apply -f deployment/secret.yaml
kubectl apply -f deployment/pizza-deployment.yaml

# 4. Générer le certificat TLS
openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
  -keyout /tmp/tls.key -out /tmp/tls.crt \
  -subj "/CN=pizza-api.local/O=pizza-api" \
  -addext "subjectAltName = DNS:pizza-api.local"

kubectl create secret tls pizza-tls-secret \
  --key /tmp/tls.key --cert /tmp/tls.crt

rm -f /tmp/tls.key /tmp/tls.crt

```

```
kubectl apply -f deployment/ingress.yaml

# 5. Ajouter l'entrée DNS locale
echo "$(minikube ip) pizza-api.local" | sudo tee -a /etc/hosts
```

8.7 Tests de l'API en HTTPS

Après avoir ajouté l'entrée DNS dans `/etc/hosts`, nous pouvons tester l'API :

```
# Healthcheck
curl -k https://pizza-api.local/

# Menu
curl -k https://pizza-api.local/menu

# Commande
curl -k -X POST https://pizza-api.local/order \
  -H "Content-Type: application/json" \
  -d '{"pizzaId":1,"quantity":2,"address":"123 Rue de la Paix"}'
```

```
id001@id001-ThinkPad-T470:~/Nextcloud/BUT3/Cloud/rapport/tp1/pizza-api$ # Healthcheck
curl -k https://pizza-api.local/

# Menu
curl -k https://pizza-api.local/menu

# Commande
curl -k -X POST https://pizza-api.local/order \
  -H "Content-Type: application/json" \
  -d '{"pizzaId":1,"quantity":2,"address":"123 Rue de la Paix"}'
{"status":"ok","service":"Pizza-Tech API v1.0","timestamp":"2026-01-12T09:52:31.860Z"}[{"id":1,"name":"Margherita","price":12.99}, {"id":2,"name":"Pep
peroni","price":14.99}, {"id":3,"name":"Veggie","price":13.99}, {"id":4,"name":"Quattro Formaggi","price":15.99}]{"orderId":"zfwpv3p","status":"confirm
id001@id001-ThinkPad-T470:~/Nextcloud/BUT3/Cloud/rapport/tp1/pizza-api$ cd ~/Nextcloud/BUT3/Cloud/rapport/tp1/pizza-api && kubectl get pods -l app=pi
```

Note : Le flag `-k` permet d'ignorer la vérification du certificat auto-signé. En prod, avec un certificat valide (Let's Encrypt), ce flag n'est pas nécessaire.

Vérification de l'état du déploiement

```
# Vérifier les pods
kubectl get pods -l app=pizza-api
kubectl get ingress pizza-api-ingress
```

```
id001@id001-ThinkPad-T470:~/Nextcloud/BUT3/Cloud/rapport/tp1/pizza-api$ kubectl get pods -l app=pizza-api
NAME                                READY   STATUS    RESTARTS   AGE
pizza-api-deployment-747bb6876f-hgxr 1/1     Running   0           43m
id001@id001-ThinkPad-T470:~/Nextcloud/BUT3/Cloud/rapport/tp1/pizza-api$ kubectl get ingress pizza-api-ingress
NAME           CLASS    HOSTS                ADDRESS      PORTS      AGE
pizza-api-ingress nginx    pizza-api.local      192.168.49.2 80, 443    44m
id001@id001-ThinkPad-T470:~/Nextcloud/BUT3/Cloud/rapport/tp1/pizza-api$
```

8.8 Vérification de la sécurité

Vérification que les secrets ne sont pas en clair

```
kubectl get secret pizza-api-secret -o yaml
```

```

id001@id001-ThinkPad-T470:~/Nextcloud/BUT3/Cloud/rapport/tp1/pizza-api$ kubectl get secret pizza-api-secret -o yaml
apiVersion: v1
data:
  API_KEY: ZGVtb3VhcGkta2V5LTYeMzQ1
  DB_CONNECTION_STRING: bW9uZ29kYjovL3VzZXI6GfZc0Bsb2NhbgHvc3Q6MjcwMTcvcG16emFKYg==
kind: Secret
metadata:
  annotations:
    kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","kind":"Secret","metadata":{"annotations":{},"name":"pizza-api-secret","namespace":"default"},"stringData":{"API_KEY":"demo-api-key-12345","DB_CONNECTION_STRING":"m
ngodb://user:pass@localhost:27017/pizzadb"},"type":"Opaque"}
  creationTimestamp: "2026-01-12T09:45:25Z"
  name: pizza-api-secret
  namespace: default
  resourceVersion: "5787"
  uid: 7f3b58e5-0d4b-453e-960c-396a32276cc2
type: Opaque
id001@id001-ThinkPad-T470:~/Nextcloud/BUT3/Cloud/rapport/tp1/pizza-api$

```

Vérification du scan de vulnérabilités

```

# Scan avec npm audit
npm audit --production

```

```

#
# npm audit --production
npm warn config production Use `--omit=dev` instead.
found 0 vulnerabilities
#

```

8.9 Test de charge avec HTTPS

```

# Avec Apache Benchmark
ab -n 5000 -c 100 https://pizza-api.local/menu

# Observer le scaling
kubectl get hpa pizza-api-hpa -w

```

Toutes les 1,0s: kubectl get pods

NAME	READY	STATUS	RESTARTS	AGE
pizza-api-deployment-747bb6876f-6gmws	1/1	Running	0	21s
pizza-api-deployment-747bb6876f-hgxrr	1/1	Running	0	19m

8.10 Récapitulatif des améliorations (Niveau Expert)

Critère	Avant	Après (Expert)
Dockerfile	Image slim, build multi-stage basique	Image alpine, utilisateur non-root, scan vulnérabilités, healthcheck intégré
12-Factor	PORT via env uniquement	Configuration complète externalisée, ConfigMap/Secret K8s, logs stdout
Déploiement	Manuel via kubectl	Script automatisé (IaC), génération certificats TLS, activation addons
Élasticité	HPA configuré	HPA + metrics observées + optimisations
Sécurité	HTTP, secrets hardcodés	HTTPS obligatoire, Secrets Manager K8s, certificat TLS, utilisateur non-root

Lien du github : <https://github.com/LOIC-only-one/Cloud-TP-BUT3/tree/main/rapport/tp1>