ESIEE PARIS

Mini Rapport

Cloud Car Rental

Réalisateurs :

Hocine BOUROUIH
Mohamed AIDAOUI

Table des matières

1 Introduction

1.1 Contexte

Ce projet s'inscrit dans le cadre du cours **Cloud Integration** et vise à concevoir une application de location de voitures basée sur une architecture microservices. L'objectif principal est de mettre en pratique les technologies **Docker**, **Kubernetes**, les **API REST**, les bases de données **PostgreSQL** et les **gateways** (Ingress NGINX) dans un environnement local utilisant **minikube**.

L'application développée, **Cloud Car Rental**, permet de gérer un parc de voitures et les locations associées, en démontrant les principes fondamentaux des architectures distribuées modernes.

1.2 Objectifs pédagogiques

Les objectifs de ce projet sont multiples :

— **Containeriser** des services Node.js avec Docker
— **Déployer** une architecture multi-conteneurs avec Kubernetes
— **Exposer** les services via une API Gateway (Ingress NGINX)
— **Persister** les données avec PostgreSQL
— **Implémenter** une collaboration inter-services (appels HTTP entre microservices)
— **Ajouter** une interface web moderne (bonus front-end)
— **Sécuriser** le cluster avec NetworkPolicies (bonus sécurité)

2 Architecture générale

2.1 Schéma d'architecture

L'architecture de **Cloud Car Rental** repose sur trois microservices principaux interconnectés via un Ingress NGINX, le tout orchestré par Kubernetes.

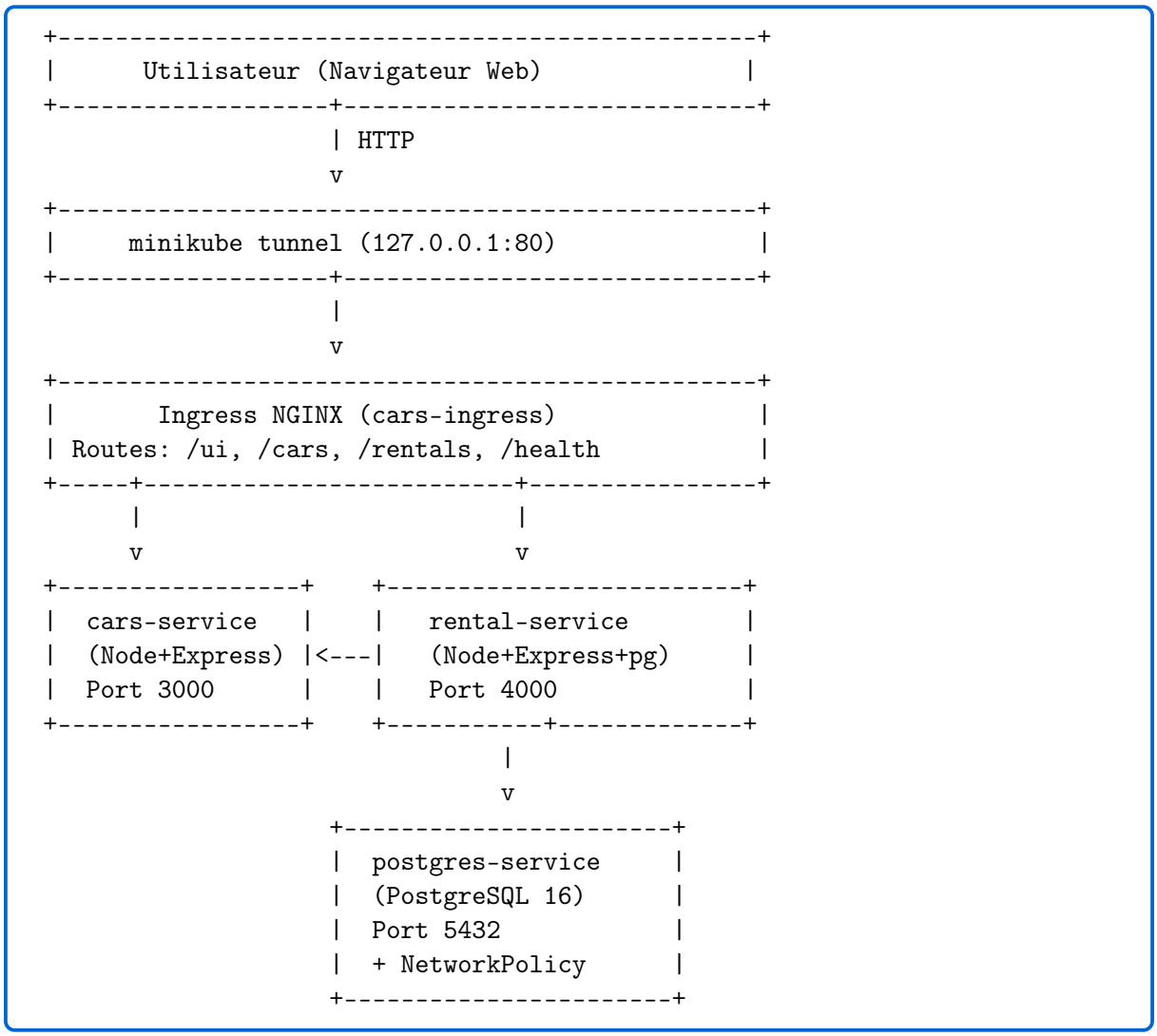

FIGURE 1 – Architecture microservices de Cloud Car Rental

2.2 Description des composants

cars-service Gère le parc de voitures avec opérations CRUD en mémoire. Expose les routes `/cars`, `/cars/:id/rent`, `/cars/:id/return` et sert le front-end sur `/ui`.

rental-service Gère les locations avec persistance dans PostgreSQL. Implémente la validation inter-services avec `cars-service`, la détection de conflits de dates et la synchronisation bidirectionnelle. Expose `/rentals`, `POST /rentals`, `DELETE /rentals/:id`.

postgres-service
: Base de données relationnelle hébergeant la table `rentals`. Initialisée automatiquement via ConfigMap avec données de test (Alice, Bob).

Ingress NGINX
: Point d'entrée unique de l'application, routant les requêtes HTTP vers les services appropriés. Exposé sur `127.0.0.1` via `minikube tunnel`.

NetworkPolicy
: Sécurise PostgreSQL en n'autorisant que le trafic provenant des pods `rental-service`.

Front-end
: Interface web moderne (HTML/CSS/JavaScript) servie par `cars-service` sur la route `/ui`.

3 Technologies utilisées

Le projet s'appuie sur un ensemble de technologies cloud-native et de frameworks web modernes.

Technologie	Rôle	Version
Node.js	Runtime JavaScript pour microservices	20-alpine
Express.js	Framework web REST API	latest
PostgreSQL	Base de données relationnelle	16-alpine
pg	Client PostgreSQL pour Node.js	latest
node-fetch	Appels HTTP inter-services	2.x
Docker	Containerisation des services	Desktop
Kubernetes	Orchestration (Deployments, Services, etc.)	minikube
Ingress NGINX	API Gateway / reverse proxy	minikube addon
Docker Hub	Registry public pour images	hocinebour/*

TABLE 1 – Stack technologique du projet

3.1 Justification des choix technologiques

— **Node.js + Express** : légèreté, facilité de développement REST, large écosystème npm.

— **PostgreSQL** : robustesse, conformité ACID, support natif des types de données complexes.

— **Kubernetes** : standard de facto pour l'orchestration de conteneurs, portabilité cloud.

— **Ingress NGINX** : solution mature et performante pour l'exposition de services HTTP(S).

— **Docker** : isolation des dépendances, reproductibilité des environnements.

4 Implémentation par palier

4.1 Palier 10/20 – Un seul service en local

4.1.1 Objectif

Créer `cars-service`, le dockeriser, le publier sur Docker Hub et le déployer dans Kubernetes.

4.1.2 Étapes réalisées

1. Développement de `cars-service` (Node.js + Express) avec routes REST :
   - `GET /cars` : liste de toutes les voitures
   - `GET /cars/:id` : détails d'une voiture
   - `GET /health` : healthcheck du service
2. Création du `Dockerfile` :

```
FROM node:20-alpine
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install --only=production
COPY app.js ./
ENV PORT=3000
EXPOSE 3000
CMD ["node", "app.js"]
```

3. Build et publication de l'image :

```
docker build -t hocinebour/cars-service:v1 .
docker push hocinebour/cars-service:v1
```

4. Création du Deployment Kubernetes (`cars-deployment.yaml`) avec 1 réplica
5. Création du Service Kubernetes (`cars-service.yaml`, type NodePort sur port 30080)
6. Tests réalisés :

```
kubectl port-forward svc/cars-service 3000:3000
curl http://localhost:3000/cars
```

4.1.3 Résultats

- Image `hocinebour/cars-service:v1` disponible sur Docker Hub
- Pod `cars-deployment` en état `Running`
- Service `cars-service` accessible via port-forward
- Réponse JSON correcte pour toutes les routes testées

4.2 Palier 12/20 – Gateway en local

4.2.1 Objectif

Exposer `cars-service` via Ingress NGINX pour fournir un point d'entrée unique.

4.2.2 Étapes réalisées

1. Activation de l'addon Ingress dans minikube :

```
minikube addons enable ingress
```

2. Création de la ressource `cars-ingress.yaml` :

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: cars-ingress
spec:
  ingressClassName: nginx
  rules:
    - http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: cars-service
                port:
                  number: 3000
```

3. Lancement du tunnel minikube :

```
minikube tunnel
```

4. Tests d'accès via l'Ingress :

```
curl http://127.0.0.1/cars
curl http://127.0.0.1/health
```

4.2.3 Résultats

— Ingress `cars-ingress` créé et actif
— Contrôleur Ingress NGINX en état `Running`
— Accès réussi à `cars-service` via `127.0.0.1`
— Tunnel minikube stable et fonctionnel

4.3 Palier 14/20 – Deuxième service

4.3.1 Objectif

Ajouter `rental-service`, le relier à `cars-service`, et l'exposer via Ingress.

4.3.2 Étapes réalisées

1. Développement de `rental-service` avec les routes :
   - `GET /rentals` : liste des locations
   - `GET /rentals/:id` : détails d'une location
   - `POST /rentals` : création de location
   - `DELETE /rentals/:id` : suppression de location
   - `GET /health` : healthcheck avec test de connexion DB
2. Implémentation de l'appel inter-services dans `POST /rentals` :

```
// Validation du carId via cars-service
const carResponse = await fetch(
  `http://cars-service:3000/cars/${carId}`
);
if (!carResponse.ok) {
  return res.status(400).json({
    error: `Car with id ${carId} does not exist`
  });
}

// Synchronisation du statut
await fetch(
  `http://cars-service:3000/cars/${carId}/rent`,
  { method: 'PUT' }
);
```

3. Build et publication des images (versions v1 à v5) :

```
docker build -t hocinebour/rental-service:v5 .
docker push hocinebour/rental-service:v5
```

4. Déploiement Kubernetes avec `rental-deployment.yaml` et `rental-service.yaml` (ClusterIP)
5. Mise à jour de l'Ingress pour router `/rentals` vers `rental-service:4000`
6. Tests d'intégration :

```
curl http://127.0.0.1/rentals
curl -X POST http://127.0.0.1/rentals \
  -H "Content-Type: application/json" \
  -d '{"customer":"Test","carId":1,...}'
```

4.3.3 Résultats

— Deux microservices opérationnels dans le cluster

— Communication inter-services fonctionnelle via DNS Kubernetes
— Validation de l'existence des voitures avant création de location
— Logs montrant les appels réussis : `"Car validated from cars-service: ..."`
— Routes `/cars` et `/rentals` accessibles via Ingress

4.4 Palier 16/20 – Base de données

4.4.1 Objectif

Ajouter PostgreSQL et connecter `rental-service` pour persister les locations.

4.4.2 Étapes réalisées

1. Création d'un Secret Kubernetes pour les identifiants PostgreSQL :

```
apiVersion: v1
kind: Secret
metadata:
  name: postgres-secret
type: Opaque
stringData:
  POSTGRES_PASSWORD: "carrentalpass"
```

2. Déploiement de PostgreSQL avec :
   - Image `postgres:16-alpine`
   - Variables d'environnement : `POSTGRES_DB`, `POSTGRES_USER`
   - Volume `emptyDir` pour stockage temporaire
   - Service ClusterIP sur port 5432
3. Création d'un ConfigMap pour initialisation automatique :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: postgres-init-sql
data:
  init.sql: |
    CREATE TABLE IF NOT EXISTS rentals (
      id SERIAL PRIMARY KEY,
      customer VARCHAR(100) NOT NULL,
      car_id INTEGER NOT NULL,
      start_date DATE NOT NULL,
      end_date DATE NOT NULL
    );
    INSERT INTO rentals (...) VALUES (...);
```

4. Montage du ConfigMap dans `/docker-entrypoint-initdb.d`
5. Modification de `rental-service` pour utiliser `pg` :

```
const { Pool } = require('pg');
const pool = new Pool({
  host: 'postgres-service',
  port: 5432,
  database: 'carrental',
  user: 'carrental',
  password: process.env.DB_PASSWORD
});
```

6. Tests de persistance :

```
kubectl exec -it postgres-pod -- \
  psql -U carrental -d carrental \
  -c "SELECT * FROM rentals;"
```

4.4.3 Résultats

— Pod PostgreSQL opérationnel

— Table `rentals` créée automatiquement au démarrage

— Données de test (Alice, Bob) insérées par le script d'init

— `rental-service` connecté à la base avec succès

— Toutes les opérations CRUD fonctionnelles avec persistance

— Healthcheck `/health` retourne `"db":"OK"`

5 Fonctionnalités bonus

5.1 Bonus sécurité – NetworkPolicy

5.1.1 Objectif

Restreindre l'accès réseau à PostgreSQL selon le principe du moindre privilège.

5.1.2 Implémentation

Création d'une NetworkPolicy autorisant uniquement les pods `rental-service` à communiquer avec PostgreSQL :

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: postgres-allow-from-rental
spec:
  podSelector:
    matchLabels:
      app: postgres
  policyTypes:
    - Ingress
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: rental-service
      ports:
        - protocol: TCP
          port: 5432
```

Application :

```
kubectl apply -f postgres-networkpolicy.yaml
kubectl get networkpolicy
```

5.1.3 Résultats

— NetworkPolicy active dans le cluster
— Sélection correcte des pods `app=postgres`
— Trafic restreint aux seuls pods `app=rental-service`
— Amélioration de la posture de sécurité du cluster

5.2 Bonus front-end – Interface web moderne

5.2.1 Objectif

Fournir une interface graphique intuitive pour visualiser et gérer les voitures et locations.

5.2.2 Implémentation

Interface HTML/CSS/JavaScript servie par `cars-service` sur la route `/ui`, avec :

- **Design moderne** : fond sombre, cartes avec ombres portées, badges colorés, typographie soignée
- **Section Parc de Voitures** : affichage dynamique des 3 voitures avec statut (Disponible/Louée)
- **Section Locations** : liste scrollable des locations avec détails (client, voiture, dates)
- **Formulaire de création** : champs validés côté client et serveur
- **Boutons de suppression** : icône × pour chaque location
- **Rechargement automatique** : mise à jour en temps réel après création/suppression
- **Gestion des erreurs** : messages d'erreur contextuels (voiture inexistante, conflit de dates)

Code JavaScript pour les appels API :

```
async function loadCars() {
  const res = await fetch('/cars');
  const cars = await res.json();
  // Affichage dynamique
}

async function loadRentals() {
  const res = await fetch('/rentals');
  const rentals = await res.json();
  // Affichage dynamique + boutons suppression
}

// Formulaire POST
rentalForm.addEventListener('submit', async (e) => {
  const res = await fetch('/rentals', {
    method: 'POST',
    body: JSON.stringify(body)
  });
  await loadRentals();
  await loadCars(); // Rafraichissement automatique
});
```

5.2.3 Résultats

- Interface responsive et moderne accessible sur `http://127.0.0.1/ui`

— Expérience utilisateur fluide sans rechargement de page
— Validation en temps réel des formulaires
— Messages d'erreur clairs et contextuels
— Satisfaction du critère "présentation (front office - css)" de l'énoncé

5.3 Bonus fonctionnel – Gestion avancée des locations

5.3.1 Fonctionnalités implémentées

1. **Validation de l'existence de la voiture**

   `rental-service` appelle `cars-service` avant toute insertion :

```
const carResponse = await fetch(
  `http://cars-service:3000/cars/${carId}`
);
if (!carResponse.ok) {
  return res.status(400).json({
    error: `Car with id ${carId} does not exist`
  });
}
```

2. **Gestion des conflits de dates**

   Vérification SQL pour empêcher deux locations simultanées :

```
SELECT id FROM rentals
WHERE car_id = $1
  AND start_date <= $3
  AND end_date >= $2
```

3. **Synchronisation bidirectionnelle**

   Après création de location :

```
await fetch(
  `http://cars-service:3000/cars/${carId}/rent`,
  { method: 'PUT' }
);
```

   Après suppression de location :

```
await fetch(
  `http://cars-service:3000/cars/${carId}/return`,
  { method: 'PUT' }
);
```

4. **CRUD complet**

   — Create : `POST /rentals`
   — Read : `GET /rentals` et `GET /rentals/:id`
   — Delete : `DELETE /rentals/:id`

5.3.2 Résultats

— Cohérence garantie entre les deux microservices
— Prévention des erreurs métier (location de voiture inexistante, chevauchement)
— Statut "Louée"/"Disponible" synchronisé en temps réel
— Expérience utilisateur cohérente et fiable

6 Conclusion

Ce projet **Cloud Car Rental** a permis de mettre en œuvre une architecture microservices complète et fonctionnelle, répondant à tous les objectifs pédagogiques du cours Cloud Integration.

6.1 Bilan des réalisations

Paliers obligatoires atteints :

- **10/20** : Service `cars-service` dockerisé, publié et déployé sur Kubernetes
- **12/20** : Gateway Ingress NGINX configurée avec `minikube tunnel`
- **14/20** : Deuxième service `rental-service` avec communication inter-services
- **16/20** : Base de données PostgreSQL intégrée avec persistance des locations

Bonus implémentés :

- Interface web moderne et responsive
- Sécurisation réseau avec NetworkPolicy
- Gestion avancée des locations (validation, conflits de dates, synchronisation bidirectionnelle)
- CRUD complet sur les locations
- Initialisation automatique de la base via ConfigMap