ECOLE MAROCAINE DES SCIENCES DE L'INGENIEUR
Membre de HONORIS UNITED UNIVERSITIES

Rapport de projet

5ème année

Ingénieur Informatique & Réseaux

Platform e-commerce

REST avec Spring BOOT & REACT

Réalisé par : YASSINE Ech-chaoui
IMANE Chakrellah

Prof : Pr.Hasbi

Classe: 5IIR-11

2025-2026

Table des matières

(Cliquez sur le lien pour accéder au dépôt du projet)

https://github.com/CHAKRELLAH44/JEE.git

INTRODUCTION

Dans le cadre de ce projet, nous avons réalisé une architecture micro-services complète basée sur l'écosystème Spring Cloud. L'objectif principal est de concevoir un système distribué permettant de gérer des produits et des commandes, tout en intégrant un serveur de configuration centralisé, un registre de services Eureka, une API Gateway, ainsi que des mécanismes avancés tels que Hystrix, Load Balancing, Actuator et Swagger.

Ce projet met également en œuvre un front-end ReactJS qui communique avec les microservices via l'API Gateway.

1. Présentation du projet et des outils utilisés

Côté Back-End :

- **Spring Boot** : construction des microservices
- **Spring Cloud** : Eureka, Config Server, Gateway, Load Balancer, Hystrix
- **Spring Data JPA** : gestion des entités et accès BD
- **H2 Database** : base de données en mémoire
- **Maven** : gestion des dépendances
- **Actuator** : supervision / health checks
- **Swagger / OpenAPI** : documentation des APIs
- **Postman** : test des endpoints

Côté Front-End :

- **ReactJS** : création de l'interface
- **Axios** : communication avec la Gateway
- **Bootstrap / CSS** : mise en forme des pages

2. Objectifs du projet

Objectifs techniques

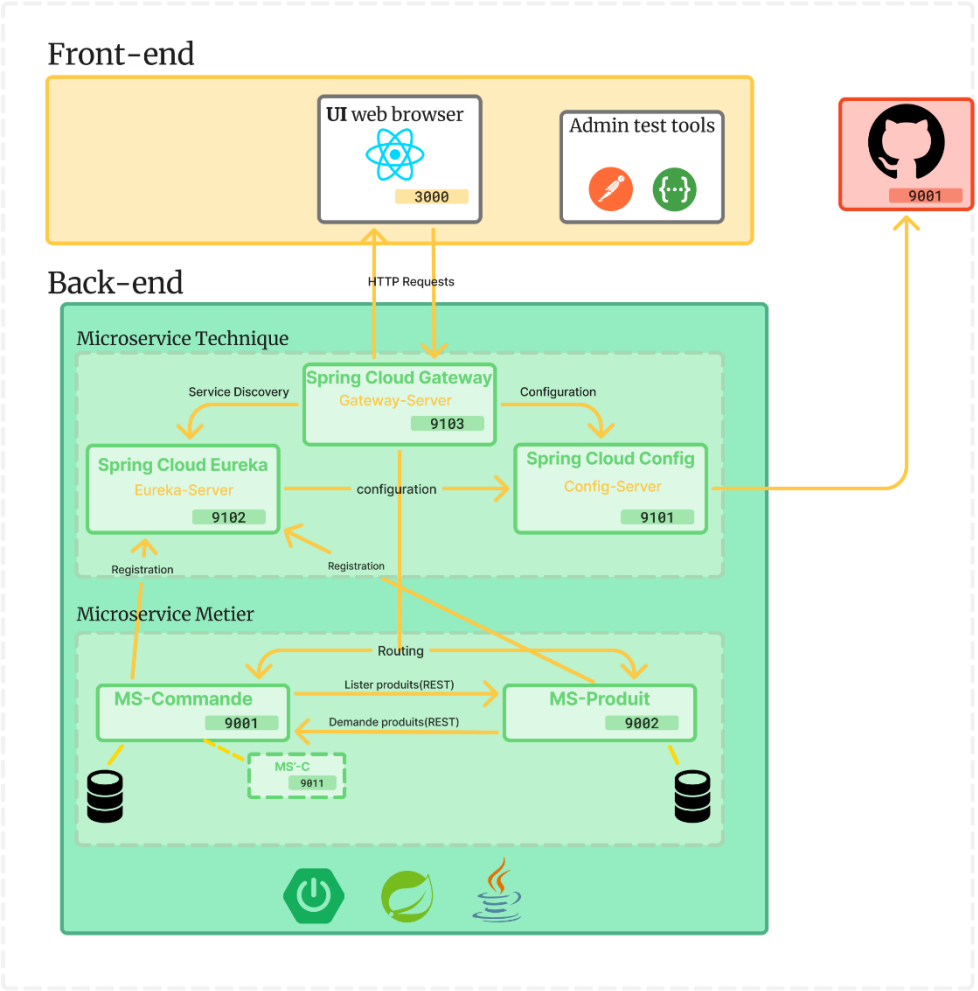
- Concevoir un **microservice Commandes** réalisant le **CRUD** via **Spring Data JPA** (aucun SQL manuel).
- Mettre en place une **configuration centralisée** et versionnée avec **Spring Cloud Config (GitHub)**.
- Ajouter des **propriétés personnalisées** et permettre leur **rechargement à chaud** grâce à **Actuator (/refresh)**.
- Implémenter un **health check personnalisé** (statut **UP/DOWN** selon l'état réel de la base).
- Faire évoluer le modèle de données vers la **version 2** en ajoutant la relation **Commande → Produit (idProduit)**.
- Centraliser l'accès à l'application via une **API Gateway** (routage + filtres).
- Assurer la **découverte automatique des services** grâce à **Eureka** et activer le **load balancing**.
- Tester la **résilience** du système : simuler un **timeout** et appliquer un **fallback** via **Hystrix**.
- Documenter l'ensemble des endpoints avec **Swagger / OpenAPI**.
- Développer un **front-end React** pour manipuler les produits et commandes (CRUD + upload si présent).

Objectifs pédagogiques

- Comprendre la **décomposition d'un monolithe en microservices** et ses avantages.
- Mettre en place une architecture **faiblement couplée**, évolutive et maintenable.
- Apprendre à **configurer, superviser et diagnostiquer** des microservices (Config Server + Actuator).
- Maîtriser le rôle d'une **Gateway** (point d'entrée unique, routage, filtrage).
- Introduire la **tolérance aux pannes** dans une architecture distribuée (timeout + fallback).
- Manipuler une base **H2 persistante** dans un contexte microservices.
- Consolider l'intégration **Back-end / Front-end** via une interface React complète.

3. Architecture générale du projet

Schéma d'architecture globale



Description du flux

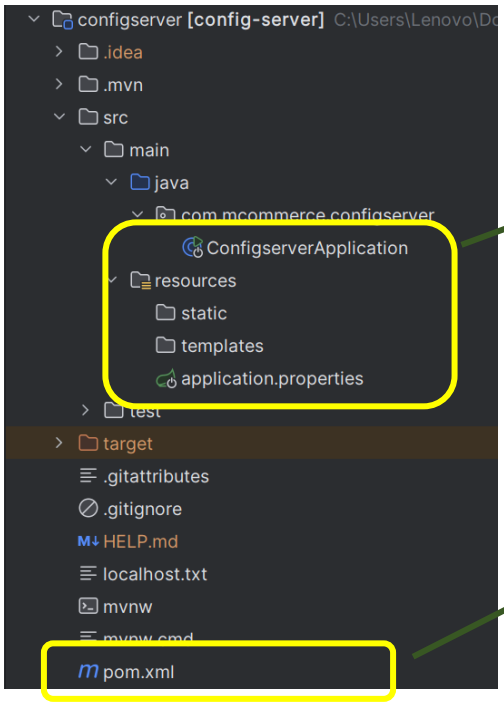
1. **Le front React** envoie toutes les requêtes vers **Gateway (port 9103)**.
2. **Gateway** redirige vers les microservices selon la route :
   - /PRODUITS/** → microservice-produits
   - /COMMANDES/** → microservice-commandes
3. **Eureka Server (9102)** enregistre les microservices et permet à la Gateway de les découvrir.
4. **Config Server (9101)** fournit les fichiers de configuration hébergés sur GitHub.
5. Les microservices communiquent avec la base H2 et entre eux lorsque nécessaire.

4. Architecture interne de chaque microservice

4.1. Config Server

But

- Centraliser la configuration de tous les microservices.
- Se connecter à un dépôt GitHub contenant application.yml pour chaque MS.
- Permet un **chargement à chaud** avec Actuator /refresh.



Dossiers importants

- ConfigserverApplication.java → démarre le serveur de configuration
- application.properties → configure le lien GitHub

Dépendances nécessaires

- **spring-cloud-config-server** : fournit le serveur de configuration Git.
- **spring-boot-starter-web** : expose les endpoints HTTP (/microservice-produits/default...).
- **spring-boot-starter-actuator** : supervision (health, etc.).

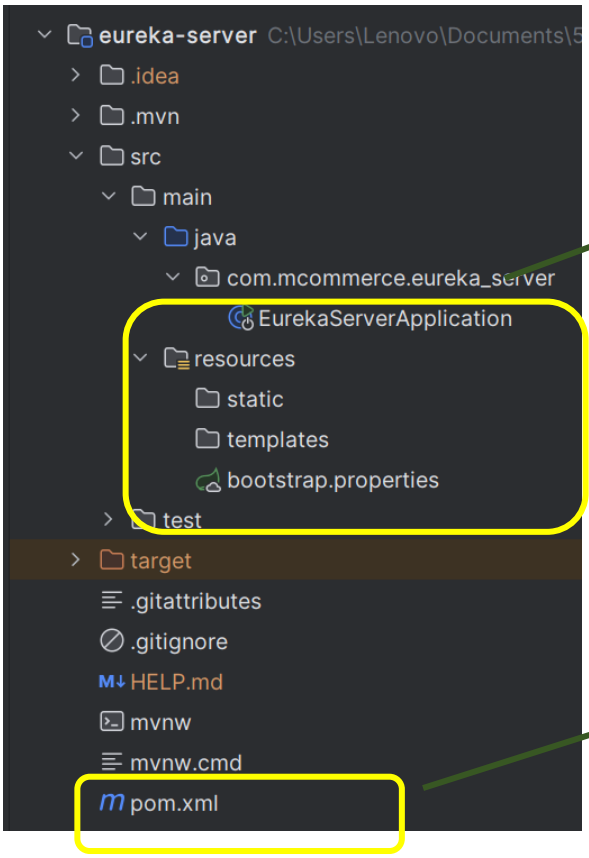
Notation essentielle

@EnableConfigServer → active le serveur de configuration.

4.2. Eureka Server

But

- Jouer le rôle d'annuaire des microservices.
- Permettre la découverte dynamique pour la Gateway.
- Fournit un registre de services pour permettre le **load balancing** et la **découverte dynamique**.



Dossiers importants

- bootstrap.properties configuration chargée **au démarrage** (avant application.properties)
- Classe principale contenant @EnableEurekaServer classe principale Spring Boot qui démarre le serveur Eureka

Dépendances

- **spring-cloud-starter-netflix-eureka-server** : active Eureka Server (dashboard + registre).
- **spring-cloud-starter-config** : récupère la configuration depuis le Config Server.
- **spring-boot-starter-actuator** : expose les endpoints de supervision (health, refresh si activé, etc.).

4.3. API Gateway

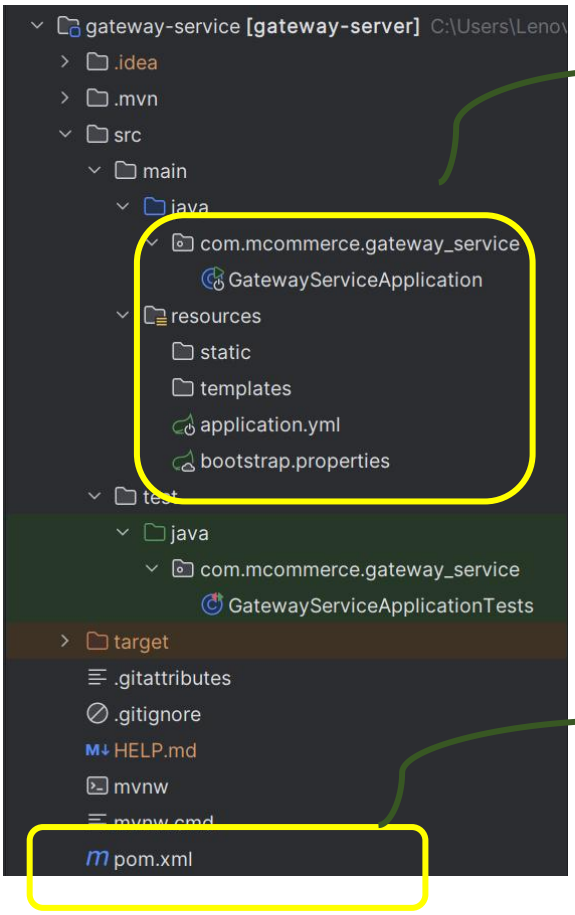
But

- Unique point d'accès entre le front et les microservices.
- Routing dynamique, CORS, Load Balancing, filtres, sécurité, etc.
- Redirige /PRODUITS/** et /COMMANDES/** vers les microservices concernés.

Pourquoi un fichier YAML ici ?

- la Gateway a besoin :
  - de déclarer les routes,
  - de dire comment router vers Eureka,
  - de configurer le multipart upload,
  - d'ajouter des filtres globaux,
  - d'activer les services Spring Cloud.

Ce fichier est le **cœur logique** de l'infrastructure.



Dossiers importants

- application.yml → configuration des routes Gateway (FICHIER LE PLUS IMPORTANT)
- Aucun controller → tout passe via le proxy
- GatewayServiceApplication : lance la Gateway.
- application.yml : **fichier le plus important ici**, contient tout le routage (routes, predicates, filters) + la config multipart si upload.
- bootstrap.properties : connexion au Config Server + activation Actuator.
- resources/ : config de démarrage.

Dépendances

- **spring-cloud-starter-gateway** : routage dynamique + filtres.
- **spring-cloud-starter-netflix-eureka-client** : enregistrement dans Eureka + découverte.
- **spring-cloud-starter-config** : config centralisée.
- **spring-boot-starter-actuator** : endpoints de monitoring.
- spring-cloud-starter-loadbalancer

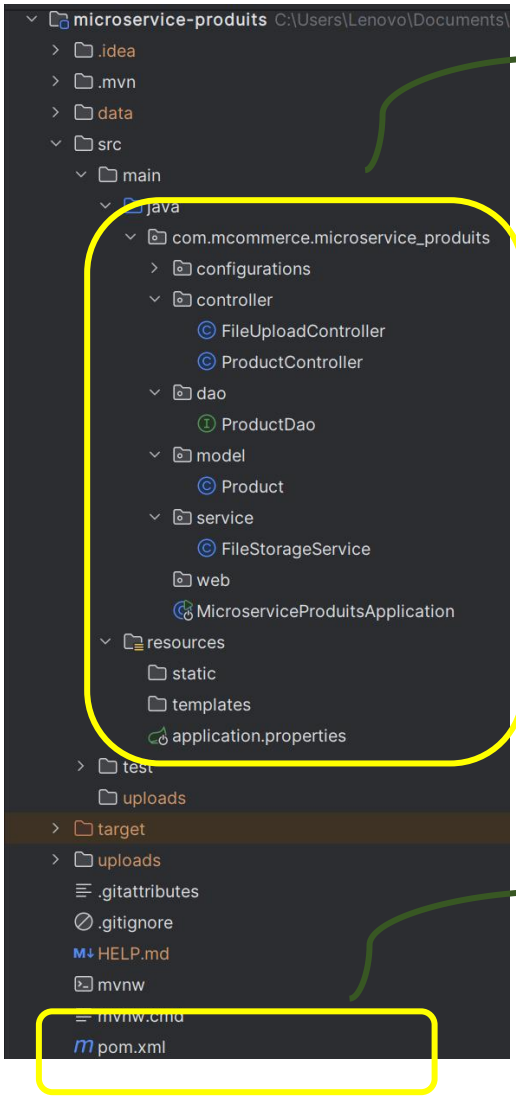
Notation

StripPrefix=1 → retire le premier fragment d'URL pour accéder au microservice.

4.4. Microservice-Produits

But

Gérer le CRUD des produits via REST + persistance H2/JPA.
Le microservice permet aussi l'upload d'image produit, stockée sur disque, et retourne un chemin (/uploads/...) exploitable par le front..



Dossiers importants

- controller/ : endpoints REST (CRUD + upload).
- dao/ : repository Spring Data JPA (CRUD sans SQL).
- model/ : entités JPA (Product) = structure de la table.
- service/ : logique métier (ex: FileStorageService pour sauvegarder les fichiers).
- configurations/ : config personnalisée (ex: GlobalConfig pour limit).
- application.properties : port, H2, Eureka, Actuator, multipart…

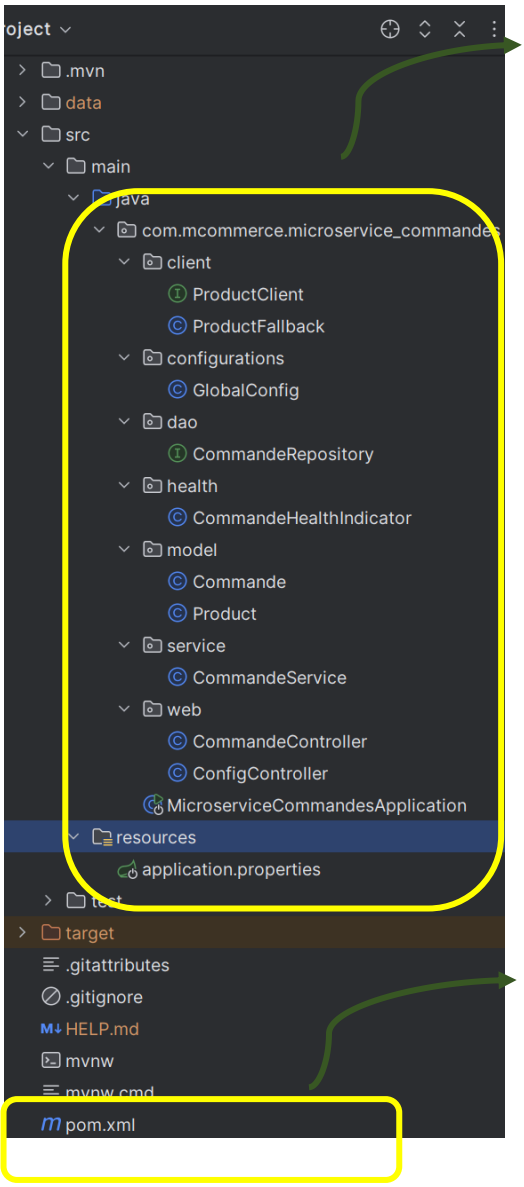
Dépendances

- **spring-boot-starter-web** : API REST.
- **spring-boot-starter-data-jpa** : CRUD JPA sans SQL.
- **h2** : base locale (test + persistance).
- **spring-cloud-starter-netflix-eureka-client** : découverte via Eureka.
- **spring-cloud-starter-config** : récupération config distante.
- **spring-boot-starter-actuator** : health, refresh, monitoring.
- **springdoc-openapi / swagger**: documentation API.

4.5. Microservice-Commandes

But

Gérer les commandes en CRUD (JPA) et relier une commande à un produit via idProduit.
Le service consomme aussi le microservice Produits via un client (Feign/Rest) et protège l'appel avec Hystrix (fallback en cas de timeout).



Dossiers importants

- eb/ : controllers REST (CRUD + endpoints config).
- dao/ : repository JPA (CommandeRepository).
- model/ : entités (Commande) + DTO modèle Product (si besoin).
- service/ : logique métier (ex: calcul, traitement, appels).
- client/ : client vers MS Produits + Fallback Hystrix.
- health/ : health indicator personnalisé (UP/DOWN selon présence de commandes).
- configurations/ : propriétés personnalisées (ex: commandes-last).
- application.properties : port, H2, Eureka, Actuator, config server.

Dépendances

- **spring-boot-starter-web** : API REST.
- **spring-boot-starter-data-jpa** : CRUD JPA.
- **h2** : base locale persistante.
- **spring-cloud-starter-netflix-eureka-client** : discovery.
- **spring-cloud-starter-config** : config centralisée.
- **spring-boot-starter-actuator** : health + refresh.
- **Hystrix** (ou Resilience selon version) : fallback en cas timeout.
- **OpenAPI/Swagger** : documentation.

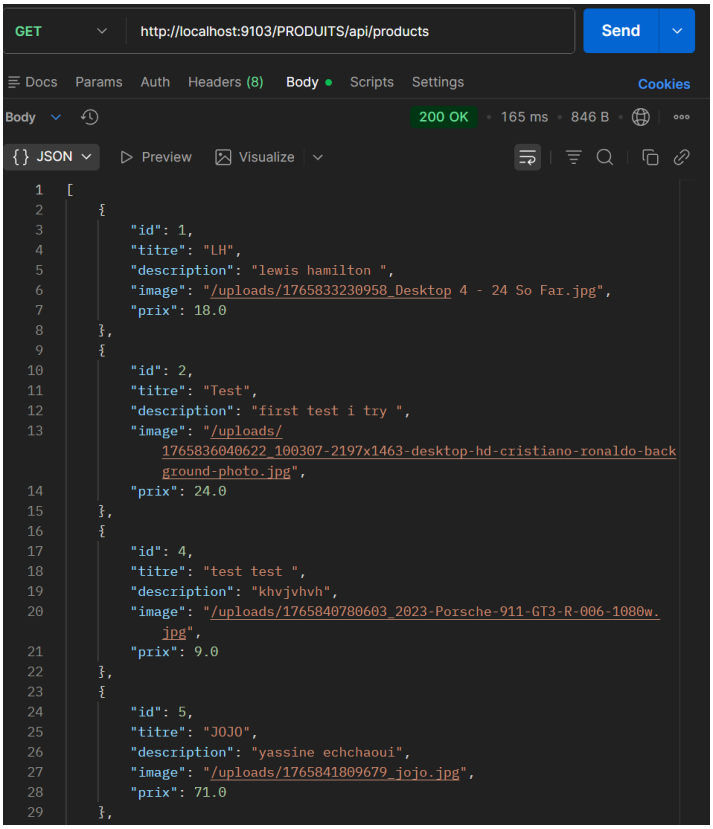
Notations importantes

- @RefreshScope → recharger la config à chaud
- @Value("${mes-config-ms.commandes-last}")
- HealthIndicator pour personnaliser l'état UP/DOWN

5. Tests techniques

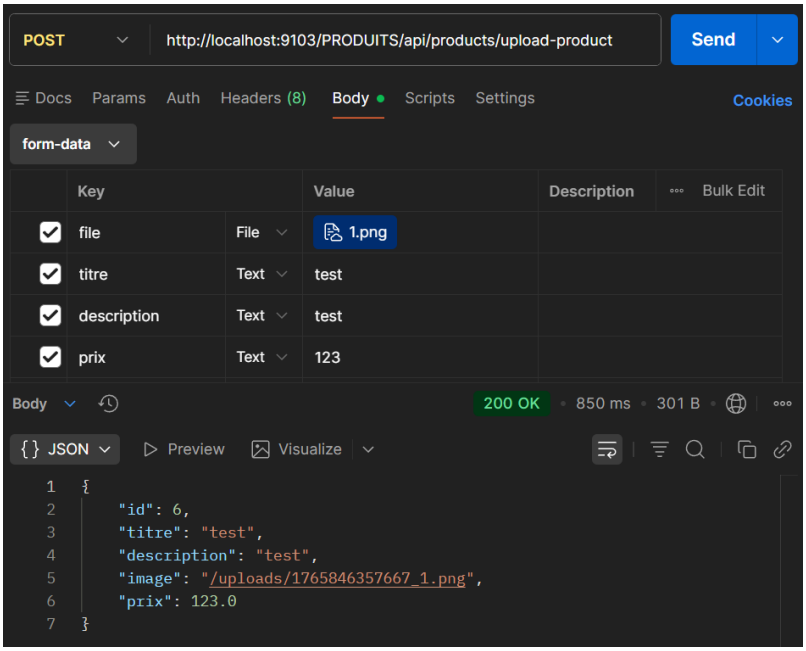
5.1 Tests microservice-produits a partir du GateWay

GET all produits



Commentaire :
Cette requête valide que la Gateway route correctement /PRODUITS/** vers microservice-produits. Le service retourne la liste des produits au format JSON, ce qui confirme le bon fonctionnement du routage + discovery Eureka.

POST création de produit

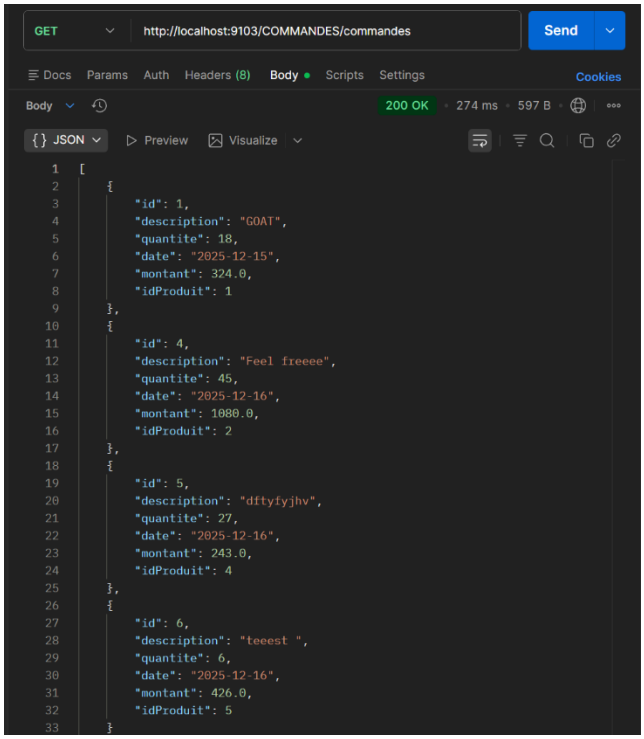


Commentaire :
Cette requête valide que la Gateway route correctement /PRODUITS/** vers microservice-produits. Le service retourne la liste des produits au format JSON, ce qui confirme le bon fonctionnement du routage + discovery Eureka.

5.2 Tests microservice-commandes a partir du GateWay

GET commandes

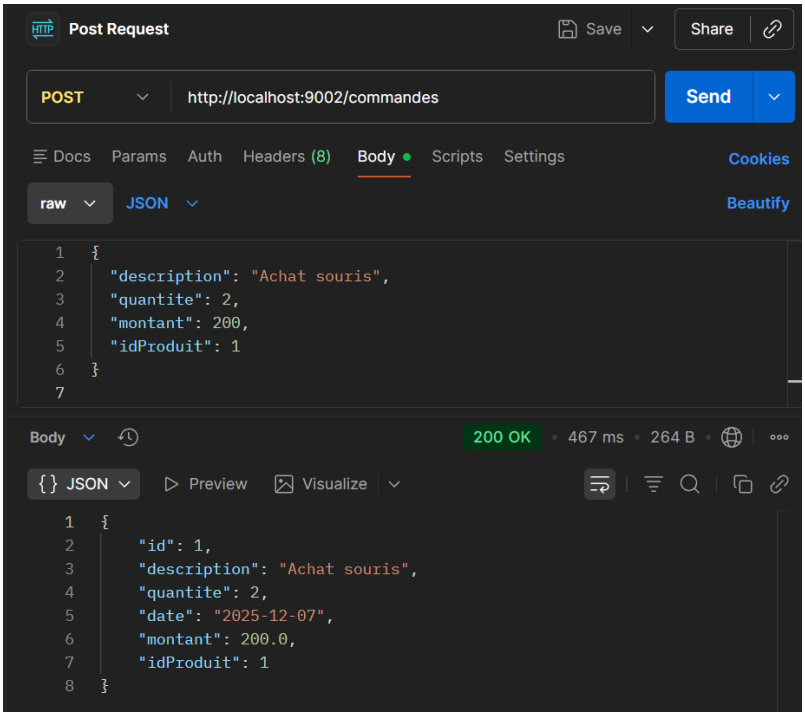
⇨ http://localhost:9103/COMMANDES/commandes



Commentaire :
Ce test confirme la création d'un produit via la Gateway. La réponse contient l'objet produit enregistré (avec son id généré), prouvant que l'API REST et la persistance via JPA/H2 fonctionnent sans SQL manuel.

POST création commande

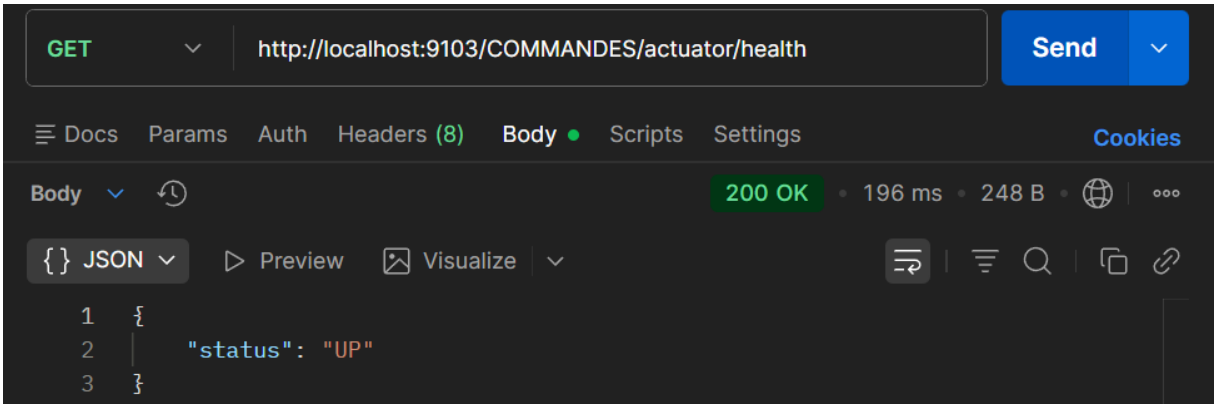
⇨ http://localhost:9103/COMMANDES/api/commandes



Commentaire :
Cette requête vérifie que la Gateway redirige /COMMANDES/** vers microservice-commandes. La liste retournée confirme l'accès CRUD et la lecture des données stockées en base H2.

Health Check

⇨ GET http://localhost:9103/COMMANDES/actuator/health



Commentaire :
La capture montre l'endpoint /actuator/health qui indique l'état du microservice. Le statut **UP** confirme que le service est disponible et que la supervision via Actuator est correctement activée.

Refresh config

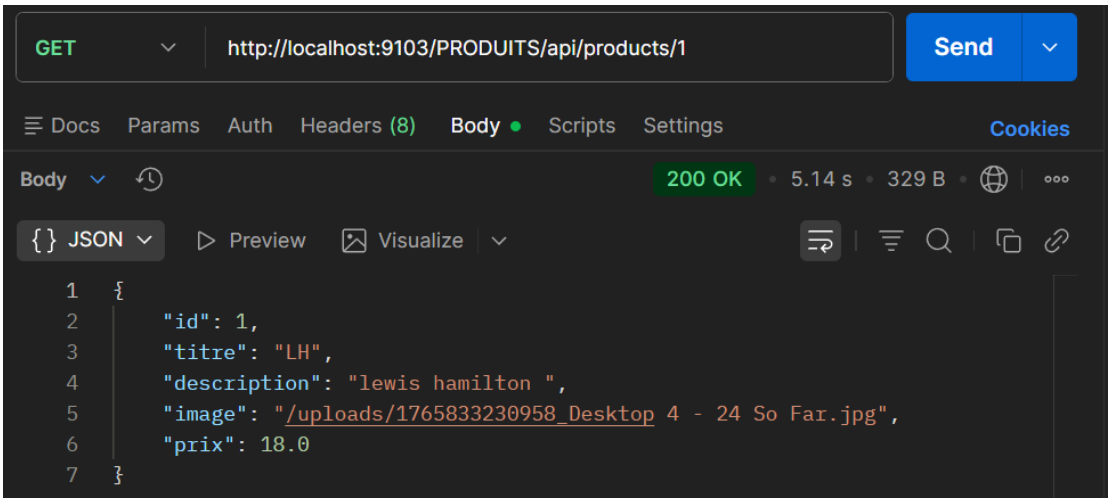
POST http://localhost:9103/COMMANDES/actuator/refresh

Commentaire :
Ce test illustre le rechargement à chaud de la configuration depuis Spring Cloud Config grâce à /actuator/refresh. Après modification de la propriété personnalisée (ex: commandes-last), le microservice applique la nouvelle valeur sans redémarrage.

5.3 Test Timeout + Hystrix

- Simuler un Thread.sleep(5000) dans MS Produits.
- Appeler via Gateway → Hystrix renvoie le fallback.

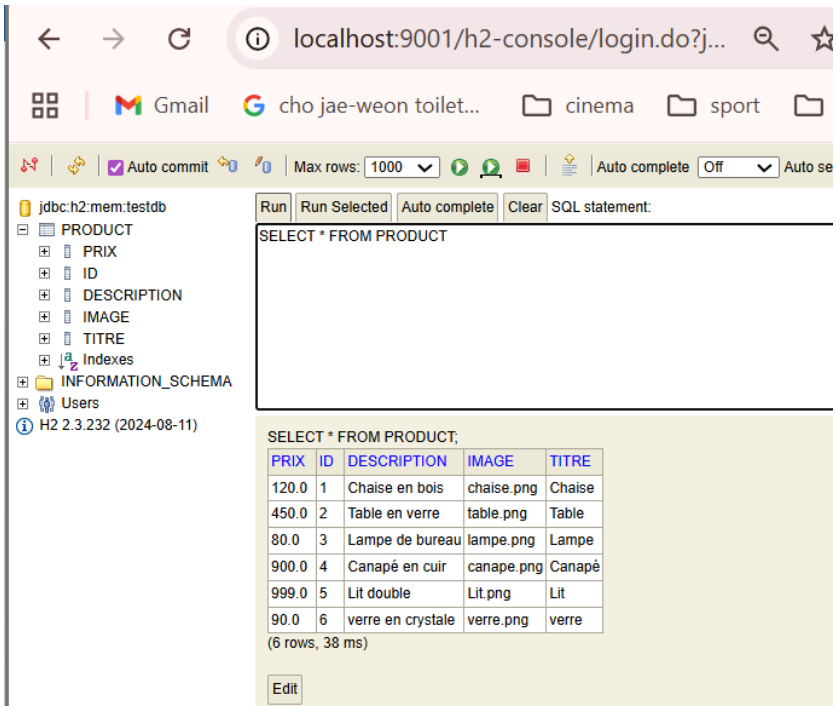
Celui-ci attend 5 secondes et renvoie le produit réel.



5.4 Console H2

=> http://localhost:9001/h2-console

JDBC URL : jdbc:h2:mem:testdb

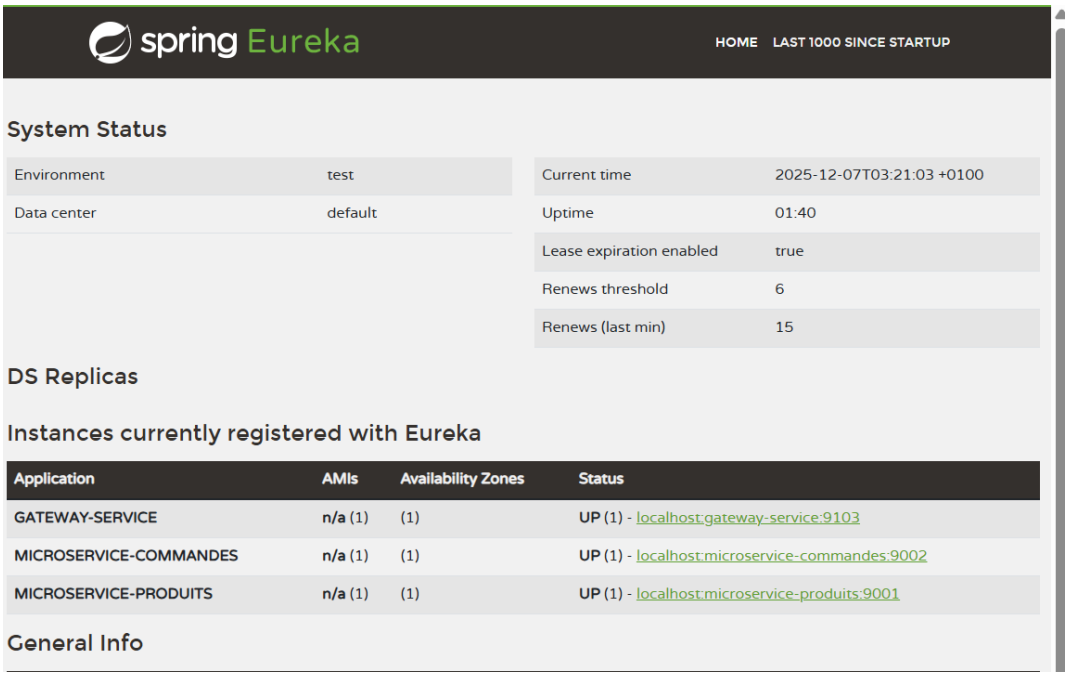


Commentaire :
La requête SQL exécutée dans la console H2 permet de vérifier directement la persistance des produits. Les lignes affichées confirment que les opérations CRUD réalisées via l'API sont bien reflétées dans la base.

5.5 Eureka

=> http://localhost:9102

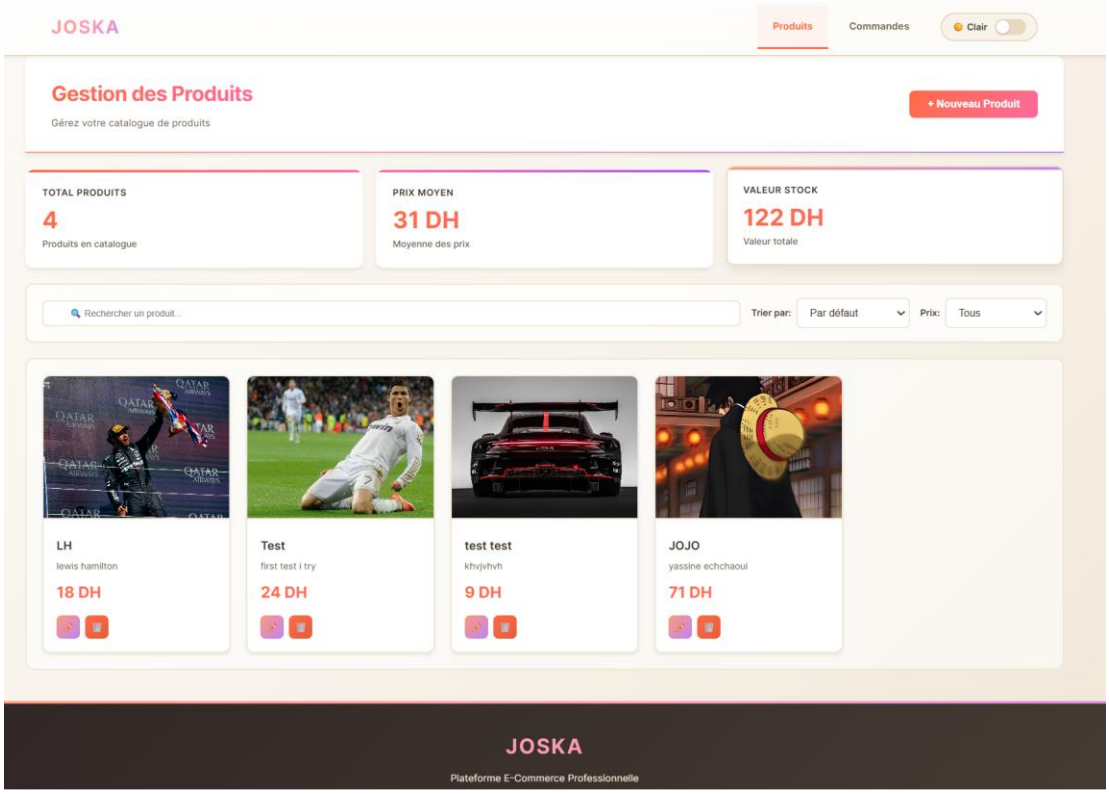
Tu dois voir le **Eureka Dashboard**



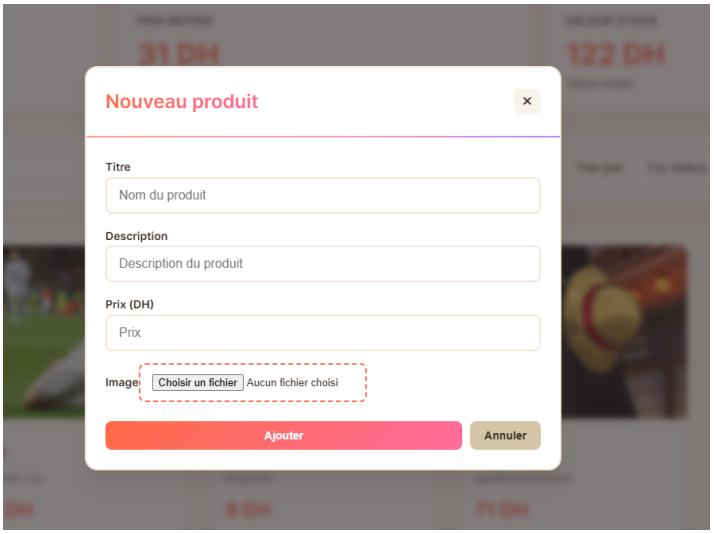
Commentaire :
Le dashboard Eureka affiche les services enregistrés (microservice-produits, microservice-commandes, gateway-service). Cela confirme que l'architecture microservices utilise correctement le Service Discovery pour la communication et le load balancing.

6. Interfaces finales

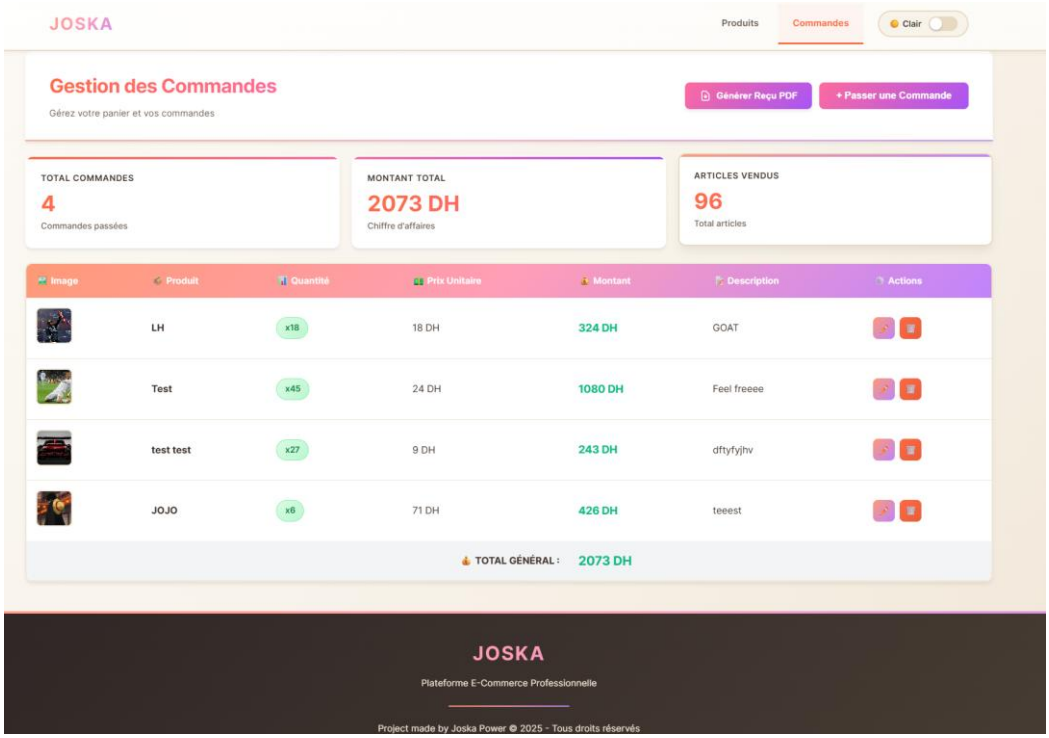
- Capture de la page React Produits



- Capture de l'ajout de produit (avec upload image)

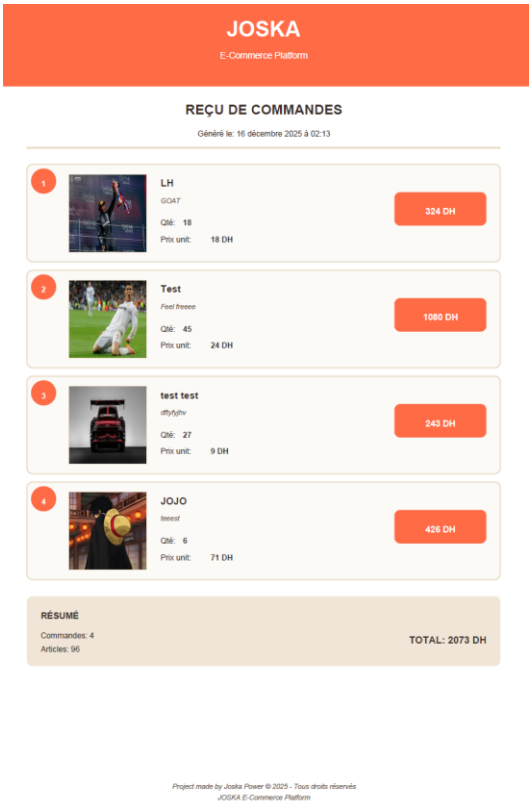


- Liste des commandes

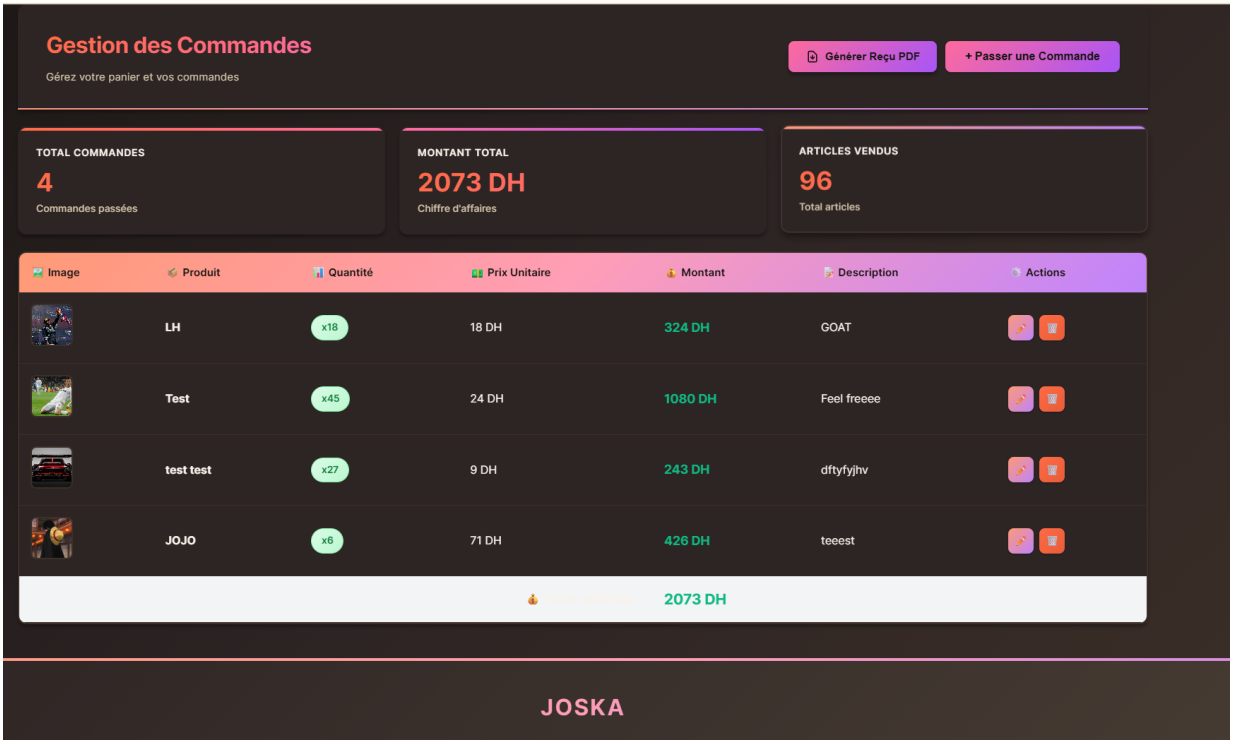


- Creation d une commande

- Recu des Commande realiser en PDF



- Mode Sombre



CONCLUSION

Ce projet a permis de mettre en place une architecture complète et moderne basée sur les microservices. L'utilisation de Spring Cloud (Eureka, Config Server, Gateway) a offert une structure scalable et professionnelle. Les fonctionnalités avancées (upload fichiers, Hystrix fallback, Actuator, H2 persistante) ont permis de couvrir l'ensemble des notions essentielles à la construction d'une application distribuée réelle.

Grâce à cette architecture modulaire, chaque service peut évoluer indépendamment, facilitant la maintenance, le déploiement et la montée en charge.