

RAPPORT - MINI PROJET MICROSERVICES

Système de Gestion Hospitalière

Nom et Prénom : Aymen Idriss OUAHYB

Date : 21 Février 2026

Module : Architecture Microservices

Email : tarik.boudaa@gmail.com

GitHub : <https://github.com/ouahybaymenidriss-dotcom/systeme-gestion-hospitaliere>

1. Introduction

Ce projet met en œuvre une application de gestion hospitalière utilisant une architecture à base de microservices avec Spring Boot et Spring Cloud. L'objectif est de démontrer les concepts fondamentaux de cette architecture : découverte de services, configuration centralisée, passerelle API, communication inter-services, et tolérance aux pannes.

2. Architecture Technique

2.1 Technologies et Dépendances

Composant	Technologie	Version	Dépendance Principale
Framework	Spring Boot	4.0.3	spring-boot-starter-web
Cloud	Spring Cloud	2025.1.0	(Nouveau Stack 2025)
Gateway	Spring Cloud Gateway MVC	-	spring-cloud-starter-gateway
Discovery	Netflix Eureka	via Spring Cloud	spring-cloud-starter-netflix-eureka-client
Config Server	Spring Cloud Config	native	spring-cloud-config-server
Config Client	Spring Cloud Config	native	spring-cloud-starter-config
Communication	OpenFeign	via Spring Cloud	spring-cloud-starter-openfeign

Resilience	Resilience4j	2.3.0	<code>spring-cloud-starter-circuitbreaker-resilience4j</code>
Base de données	H2 (in-memory)	2.4.x	<code>com.h2database:h2</code>
ORM	Spring Data JPA	-	<code>spring-boot-starter-data-jpa</code>
Build	Maven	3.9+	-
Runtime	Java	21	-

2.2 Microservices

Le système est composé de 6 microservices :

1. **Config Server (port 8888)** — Fournit la configuration centralisée à tous les services via le profil `native` (fichiers YAML stockés localement).
2. **Eureka Server (port 8761)** — Annuaire de services permettant la découverte dynamique et l'enregistrement automatique de chaque microservice.
3. **API Gateway (port 8080)** — Point d'entrée unique de l'application. Route les requêtes vers les services appropriés en utilisant `GatewayRouterFunctions` et `HandlerFunctions.http()` de Spring Cloud Gateway MVC.
4. **Patient Service (port 8081)** — Gestion CRUD des patients (nom, prénom, date de naissance, contact). Exposé sous `/api/patients`.
5. **Appointment Service (port 8082)** — Gestion des rendez-vous. Utilise OpenFeign pour vérifier l'existence du patient avant la création d'un rendez-vous. Exposé sous `/api/appointments`.
6. **Medical Record Service (port 8083)** — Gestion des dossiers médicaux. Utilise OpenFeign pour vérifier l'existence du patient. Exposé sous `/api/medical-records`.

2.3 Diagramme d'architecture

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project View:** Shows the project structure under "systeme-de-gestion-hospitaliere-microservices-main". The "config-server" service is selected.
- Code Editor:** Displays the content of `PatientDTO.java` (shown below).
- Services View:** Shows the running services: Spring Boot (ApiGatewayApplication:8080), AppointmentServiceApplication:8082, ConfigServerApplication:8888, EurekaServerApplication:8761, MedicalRecordServiceApplication:8083, and PatientServiceApplication:8081.
- Console View:** Displays log entries from the ApiGatewayApplication, showing interactions with the Netflix Eureka discovery client.

```
package com.hospital.appointment_service.clients;
import java.time.LocalDate;

public class PatientDTO {
    private Long id;
    private String nom;
    private String prenom;
    private LocalDate dateNaissance;
    private String contact;

    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getNom() { return nom; }
    public void setNom(String nom) { this.nom = nom; }
    public String getPrenom() { return prenom; }
    public void setPrenom(String prenom) { this.prenom = prenom; }
    public LocalDate getDateNaissance() { return dateNaissance; }
    public void setDateNaissance(LocalDate dateNaissance) { this.dateNaissance = dateNaissance; }
    public String getContact() { return contact; }
    public void setContact(String contact) { this.contact = contact; }
}
```

Figure 1 : Structure du projet dans IntelliJ IDEA montrant les 6 microservices

3. Entités et Modèles de données

Patient

Champ	Type	Description
id	Long	Identifiant auto-généré
nom	String	Nom du patient
prenom	String	Prénom du patient
dateNaissance	LocalDate	Date de naissance
contact	String	Coordonnées

Appointment (Rendez-vous)

Champ	Type	Description
id	Long	Identifiant auto-généré
patientId	Long	Référence au patient (clé étrangère logique)
date	LocalDateTime	Date et heure du rendez-vous

MedicalRecord (Dossier médical)

Champ	Type	Description
id	Long	Identifiant auto-généré
patientId	Long	Référence au patient
diagnostics	String	Diagnostic médical
createdAt	LocalDateTime	Date de création

4. Communication Inter-services

4.1 OpenFeign

Les services `appointment-service` et `medical-record-service` utilisent OpenFeign pour appeler le `patient-service` via Eureka :

java

```
@FeignClient(name = "patient-service")
public interface PatientServiceClient {
    @GetMapping("/api/patients/{id}")
    PatientDTO getPatientById(@PathVariable("id") Long id);
}
```

Lors de la création d'un rendez-vous ou d'un dossier médical, le service vérifie l'existence du patient avant de persister la donnée.

4.2 API Gateway - Routage

L'API Gateway utilise la DSL Java de Spring Cloud Gateway MVC :

java

```
@Bean
public RouterFunction<ServerResponse> patientRoute() {
    return route("patient-service")
```

```

        .route(path("/api/patients/**"), HandlerFunctions.http())
        .before(BeforeFilterFunctions.uri("http://localhost:8081"))
            .build();
    }
}

```

5. Tolérance aux pannes (Resilience4j)

5.1 Configuration

yaml

```

@Bean
public RouterFunction<ServerResponse> patientRoute() {
    return route("patient-service")
        .route(path("/api/patients/**"), HandlerFunctions.http())
        .before(BeforeFilterFunctions.uri("http://localhost:8081"))
            .build();
}

```

5.2 Annotations

java

```

@PostMapping
@CircuitBreaker(name = "patientService", fallbackMethod =
    "fallbackCreateAppointment")
@Retry(name = "patientService")
public Appointment createAppointment(@RequestBody Appointment appointment)
    { ... }

public Appointment fallbackCreateAppointment(Appointment appointment, Throwable
    throwable) {
    throw new RuntimeException("Service patient temporairement indisponible...");
}

```

8

5.3 Résultat du test

Scénario	Patient Service	Résultat
Création RDV	UP	200 OK - RDV créé avec succès

Création RDV	DOWN	500 - Fallback : « Service patient indisponible »
Création dossier	UP	200 OK - Dossier créé
Création dossier	DOWN	500 - Fallback : « Service patient indisponible »

6. Tests effectués

6.1 Vérification des services

- Tous les 6 services démarrent correctement.
- Tous les services s'enregistrent sur Eureka.
- Le Config Server distribue les configurations.

The screenshot shows the Spring Eureka dashboard at localhost:8761. The top navigation bar includes links for Home, Documentation, GitHub, and Issues. The main content area is divided into sections:

- System Status:** Displays environment (test), data center (default), current time (2026-02-21T22:24:48+0000), uptime (00:00), lease expiration enabled (false), renew threshold (1), and renew count (0).
- DS Replicas:** Shows a single instance registered under the host 'localhost'.
- Instances currently registered with Eureka:** A table with columns Application, AMIs, Availability Zones, and Status. It shows 'No instances available'.
- General Info:** A table with columns Name and Value. It includes metrics like total-available-memory (132mb), number of cpus (6), current-memory-usage (74mb (56%)), server uptime (00:00), and registered-replicas (HTTP://localhost:8761/eureka/). It also lists URLs for configuration and bootstrap files.

Figure 2 : Dashboard Eureka confirmant l'enregistrement des services

6.2 Tests fonctionnels via API Gateway

- **POST /api/patients** → Création patient (200 OK)
- **GET /api/patients** → Liste des patients (200 OK)
- **POST /api/appointments** → Création RDV avec vérification patient (200 OK)

- `POST /api/medical-records` → Création dossier avec vérification patient (200 OK)

(Voir vidéo de démonstration pour les tests Postman en direct)

6.3 Tests de résilience

- Arrêt du `patient-service` → Fallback Resilience4j activé.
- Circuit Breaker + Retry fonctionnels.
- Message d'erreur contrôlé retourné au client.

(Voir vidéo de démonstration minute 2:00 pour le test de panne)

7. Difficultés rencontrées

1. **Spring Cloud Gateway MVC vs WebFlux** : Spring Boot 4.x utilise par défaut Gateway MVC (Tomcat) au lieu de WebFlux (Netty). La configuration YAML des routes ne fonctionnait pas avec la version MVC.
 - a. **Solution** : Remplacement de la configuration YAML par du code Java utilisant `GatewayRouterFunctions.route()` et `HandlerFunctions.http()`.
2. **Dépendance Config Server manquante** : L'API Gateway ne récupérait pas sa configuration du Config Server car la dépendance `spring-cloud-starter-config` était absente dans le POM initial.
 - a. **Solution** : Ajout manuel de la dépendance dans le `pom.xml` et vérification des profiles actifs.
3. **Conflits de version Java** : Des erreurs de compilation sont apparues lors du build Maven local dues à une incompatibilité de version JDK.
 - a. **Solution** : Uniformisation de tous les projets sur Java 21 et utilisation des outils de build intégrés à IntelliJ.

8. Conclusion

Ce projet démontre la mise en œuvre complète d'une architecture microservices avec Spring Cloud, incluant la configuration centralisée, la découverte de services, le routage API, la communication inter-services via OpenFeign, et la tolérance aux pannes avec Resilience4j. L'ensemble fonctionne de manière cohérente et les mécanismes de résilience protègent efficacement le système en cas de défaillance d'un service.

Lien GitHub : <https://github.com/ouahybaymenidriss-dotcom/systeme-gestion-hospitaliere>