Mini-Projet : Système de Gestion d'Hôpital en Microservices

Architecture distribuée avec Spring Boot, Spring Cloud, Eureka, Config Server, API Gateway, OpenFeign et Resilience4j

Étudiant : Mouad Chakir
Établissement : Faculté des Sciences, Université Ibn Tofaïl – Kénitra

1. Introduction & Objectif du projet

Ce mini-projet met en place un **système simplifié de gestion d'hôpital** en architecture microservices, en suivant des pratiques proches d'un environnement de production : découpage en services, découverte, routage centralisé, configuration distribuée, communication inter-services et mécanismes de résilience.

Le système est composé de **3 microservices métier** (Patient, Appointment, Medical Record) et de **4 services d'infrastructure** (Eureka Server, API Gateway, Config Server, Zipkin optionnel). Chaque microservice possède **sa propre base de données MySQL** afin de garantir l'isolation des données et le couplage faible.

Java 17 Spring Boot Spring Cloud 2023 MySQL

Objectif principal : démontrer une architecture microservices complète incluant :

- Découpage en services métier indépendants
- Découverte de services via Eureka
- Routage centralisé via Spring Cloud Gateway
- Configuration distribuée avec Spring Cloud Config
- Communication inter-services via OpenFeign
- Résilience avec Resilience4j (Circuit Breaker, Retry, Timeout, Fallback)

2. Architecture globale

2.1 Vue d'ensemble

L'architecture suit un schéma **client → API Gateway → Microservices métier**, avec un **Config Server** pour centraliser la configuration et un **Eureka Server** pour la découverte des services.

- **Patient Service** – gestion des patients (données administratives)
- **Appointment Service** – gestion des rendez-vous, dépend du Patient Service
- **Medical Record Service** – gestion du dossier médical, dépend du Patient Service
- **Eureka Server** – annuaire de services
- **API Gateway** – point d'entrée unique pour les clients
- **Config Server** – configuration centralisée (YAML) pour tous les services
- **Zipkin Server (optionnel)** – base pour le traçage distribué

2.2 Ports et services

Service	Rôle	Port
Eureka Server	Découverte de services	8761
Config Server	Configuration centralisée	8888
API Gateway	Routage & point d'entrée	8080
Patient Service	Gestion des patients	8081
Appointment Service	Gestion des rendez-vous	8082
Medical Record Service	Gestion des dossiers médicaux	8083
Zipkin (optionnel)	Traçage distribué	9411

2.3 Technologies principales

Le projet utilise les technologies et dépendances suivantes :

Back-end & Cloud

Spring Boot Spring Web / WebFlux

Spring Data JPA

Spring Cloud Netflix Eureka

Spring Cloud Gateway

Spring Cloud Config

Spring Cloud OpenFeign Resilience4j

Data & Infra

MySQL (1 DB par microservice)

Hibernate HikariCP Actuator

Build & Environnement

Java 17+ Maven multi-modules

YAML configuration

3. Microservices métier

3.1 Patient Service

Responsabilité : gérer les données administratives des patients.

- **Entity :** `Patient` (id, nom, prenom, dateNaissance, contact)
- **Couches :** Controller, Service, ServiceImpl, Repository, Entity
- **Base de données :** `patient_db` (MySQL)
- **Enregistrement Eureka :** `PATIENT-SERVICE`
- **Configuration :** fournie par Config Server via `patient-service.yml`

Endpoints exposés (via Gateway 8080) :

- `POST /patients` – créer un patient
- `GET /patients/{id}` – récupérer un patient par id
- `GET /patients` – lister tous les patients

3.2 Appointment Service

Responsabilité : gérer les rendez-vous médicaux.

- **Entity :** `Appointment` (id, date, reason, patientId)
- **Base de données :** `appointment_db` (MySQL)
- **Dépendance :** vérifie l'existence du patient via Patient Service

Endpoints (via Gateway 8080) :

- `POST /appointments` – créer un rendez-vous
- `GET /appointments/patient/{patientId}` – rendez-vous par patient

La validation du patient se fait via un client OpenFeign vers l'API Gateway :

```
@FeignClient(name = "api-gateway")
public interface PatientClient {
    @GetMapping("/patients/{id}")
```

```
    PatientDto getPatientById(@PathVariable("id") Long id);
}
```

3.3 Medical Record Service

Responsabilité : gérer le dossier médical (historique) d'un patient.

- **Entity :** `MedicalRecord` (id, patientId, diagnosis, description, date)
- **Base de données :** `medical_record_db` (MySQL)
- **Dépendance :** vérifie le patient via Patient Service (Feign + Gateway)

Endpoints (via Gateway 8080) :

- `POST /records` – créer un dossier médical
- `GET /records/patient/{patientId}` – dossiers médicaux d'un patient

4. Services d'infrastructure

4.1 Eureka Server

Eureka joue le rôle d'**annuaire de services**. Tous les microservices métier, ainsi que l'API Gateway, s'y enregistrent au démarrage. Le dashboard est accessible sur `http://localhost:8761` .

4.2 Config Server

Le Config Server fournit une **configuration centralisée** en mode *native*, avec des fichiers YAML stockés dans `classpath:/config` . Chaque microservice possède un fichier dédié, par exemple :

- `patient-service.yml`
- `appointment-service.yml`
- `medical-record-service.yml`
- `api-gateway.yml`

Les microservices consomment cette configuration via `bootstrap.yml` et la propriété `spring.config.import: "configserver:"` .

4.3 API Gateway

L'API Gateway (Spring Cloud Gateway) est le **point d'entrée unique** pour tous les clients. Il route les requêtes vers les microservices cibles en se basant sur Eureka.

Routage principal :

- `/patients/** → PATIENT-SERVICE`
- `/appointments/** → APPOINTMENT-SERVICE`
- `/records/** → MEDICAL-RECORD-SERVICE`

4.4 Zipkin (optionnel)

Un module Zipkin est prévu pour héberger un serveur de traçage distribué sur le port 9411. Dans ce mini-projet, il sert de base pour une intégration future de tracing (span, traces entre services).

5. Résilience : Circuit Breaker, Retry, Timeout, Fallback

Les microservices **Appointment** et **Medical Record** dépendent du Patient Service. Pour éviter qu'une panne ou une lenteur du Patient Service ne bloque tout le système, on utilise **Spring Cloud Circuit Breaker avec Resilience4j**.

5.1 Configuration Resilience4j (YAML)

Les configurations sont centralisées dans le Config Server :

Appointment Service – `appointment-service.yml`

```
resilience4j:
  circuitbreaker:
    instances:
      patientService:
        slidingWindowSize: 5
        minimumNumberOfCalls: 3
        failureRateThreshold: 50
        waitDurationInOpenState: 5s
  retry:
    instances:
      patientService:
        maxAttempts: 3  # => Retry (3 fois)
        waitDuration: 500ms
  timelimiter:
    instances:
      patientService:
        timeoutDuration: 2s  # => Timeout 2 secondes
```

Medical Record Service – `medical-record-service.yml`

```
resilience4j:
  circuitbreaker:
    instances:
      patientService:
        slidingWindowSize: 5
        minimumNumberOfCalls: 3
        failureRateThreshold: 50
        waitDurationInOpenState: 5s
  retry:
    instances:
      patientService:
        maxAttempts: 3  # => Retry (3 fois)
        waitDuration: 500ms
  timelimiter:
    instances:
      patientService:
        timeoutDuration: 2s  # => Timeout 2 secondes
```

5.2 Implémentation dans Appointment Service

Dans `AppointmentServiceImpl` , la méthode `validatePatient` est protégée par un Circuit Breaker, un Retry et un TimeLimiter. Elle utilise le client Feign pour appeler Patient Service via l'API Gateway.

```
 //Circuit Breaker
@CircuitBreaker(name = "patientService", fallbackMethod = "patientServiceFallba
@Retry(name = "patientService")
@TimeLimiter(name = "patientService")
public CompletableFuture<Void> validatePatient(Long patientId) {
    return CompletableFuture.runAsync(() -> patientClient.getPatientById(patien
}

//Fallback
public CompletableFuture<Void> patientServiceFallback(Long patientId, Throwable
    return CompletableFuture.failedFuture(
        new IllegalStateException("Patient service unavailable, please try late
    );
}
```

La création de rendez-vous appelle cette méthode avant la sauvegarde :

```
public Appointment createAppointment(Appointment appointment) {
    try {
        validatePatient(appointment.getPatientId()).join();
    } catch (CompletionException ex) {
        // gestion de l'exception remontée par le fallback ou par Resilience4j
    }
    return appointmentRepository.save(appointment);
}
```

5.3 Implémentation dans Medical Record Service

Le même schéma est appliqué dans `MedicalRecordServiceImpl` :

```
//Circuit Breaker
@CircuitBreaker(name = "patientService", fallbackMethod = "patientServiceFallba
@Retry(name = "patientService")
@TimeLimiter(name = "patientService")
public CompletableFuture<Void> validatePatient(Long patientId) {
    return CompletableFuture.runAsync(() -> patientClient.getPatientById(patien
}

//Fallback
public CompletableFuture<Void> patientServiceFallback(Long patientId, Throwable
    return CompletableFuture.failedFuture(
        new IllegalStateException("Patient service unavailable, please try late
    );
}
```

5.4 Fallback fonctionnel

Quand le Patient Service est indisponible (panne ou lenteur) :

- Le TimeLimiter déclenche un timeout si l'appel dépasse 2s.
- Le Retry réessaie jusqu'à 3 fois l'appel au service.
- Si les erreurs persistent, le Circuit Breaker s'ouvre et court-circuite les appels suivants.
- La méthode de fallback renvoie un message clair : *"Patient service unavailable, please try later"*.

Ainsi, le système reste fonctionnel et renvoie un message explicite au lieu de planter silencieusement.

6. Scénarios de test

6.1 Tests fonctionnels de base

1. **Créer un patient**
   `POST http://localhost:8080/patients`
   Body JSON exemple :

```
{
  "nom": "Dupont",
  "prenom": "Jean",
  "dateNaissance": "1990-05-10",
  "contact": "0600000000"
}
```

2. **Récupérer un patient**
   `GET http://localhost:8080/patients/1`
3. **Créer un rendez-vous pour un patient existant**
   `POST http://localhost:8080/appointments`

```
{
  "date": "2026-01-28T10:00:00",
  "reason": "Consultation",
  "patientId": 1
}
```

4. **Lister les rendez-vous d'un patient**
   `GET http://localhost:8080/appointments/patient/1`
5. **Créer un dossier médical**
   `POST http://localhost:8080/records`

```
{
  "patientId": 1,
  "diagnosis": "Grippe",
  "description": "Fièvre, toux",
  "date": "2026-01-27"
}
```

6. **Lister les dossiers médicaux d'un patient**
   `GET http://localhost:8080/records/patient/1`

6.2 Tests de résilience

1. **Patient Service arrêté**
    - Arrêter le **patient-service**.
    - Appeler `POST /appointments` ou `POST /records` .
    - Observer : les appels vers le Patient Service échouent, le fallback renvoie le message *"Patient service unavailable, please try later"*.
2. **Timeout & Retry**
    - Simuler un délai important dans le Patient Service (par ex. `Thread.sleep`).
    - Observer que les appels sont retentés (retry) pendant 2 secondes maximum
    - Au-delà, le TimeLimiter déclenche un timeout et le fallback est exécuté.
3. **Circuit Breaker ouvert**
    - Provoquer plusieurs erreurs consécutives (patient-service down).
    - Après un certain nombre d'échecs (fenêtre glissante), le circuit passe en état *OPEN*.
    - Les appels suivants sont court-circuités immédiatement et le fallback est appelé directement.

7. Conclusion

Ce mini-projet illustre une **architecture microservices cohérente et complète** pour un système de gestion d'hôpital simplifié, avec séparation claire des responsabilités, découverte de services, configuration centralisée, routage via API Gateway et mécanismes de résilience.

Les principes mis en œuvre (bases de données séparées, API REST, Feign, Resilience4j, Eureka, Config Server) sont directement transposables à des architectures de production plus complexes.

Rapport généré pour le mini-projet "Microservices Hospital Management System".