TP 3 : Construction d'une Architecture Microservices avec Spring Cloud

Auteur : Saoudi Haythem
Matière : JEE2
Classe : 5ème Année Génie Logiciel

Table des matières

1 Introduction et Architecture Cible

Bienvenue dans ce travaux pratique dédié à la construction d'une architecture microservices moderne avec l'écosystème Spring. Avant de nous lancer dans l'implémentation, il est crucial de comprendre la philosophie derrière cette approche et les problèmes qu'elle cherche à résoudre.

1.1 Le Contexte : Des Monolithes aux Microservices

Historiquement, les applications étaient souvent développées comme un **monolithe** : une seule et unique unité de déploiement contenant l'ensemble de la logique métier (interface utilisateur, logique applicative, accès aux données). Si cette approche est simple au démarrage, elle révèle rapidement ses limites lorsque l'application grandit en complexité :

- **Scalabilité rigide** : Si une seule fonctionnalité de l'application nécessite plus de ressources, il faut redéployer et mettre à l'échelle l'application entière, ce qui est inefficace.
- **Fragilité** : Une erreur non gérée dans un module peut potentiellement faire tomber toute l'application (un seul point de défaillance).
- **Inertie technologique** : Il est très difficile de faire évoluer la pile technologique (langage, base de données) d'un monolithe. L'ensemble du projet est "verrouillé" dans les choix initiaux.
- **Complexité et ralentissement des développements** : Plus le code base grossit, plus il devient difficile à comprendre, à maintenir et à faire évoluer rapidement.

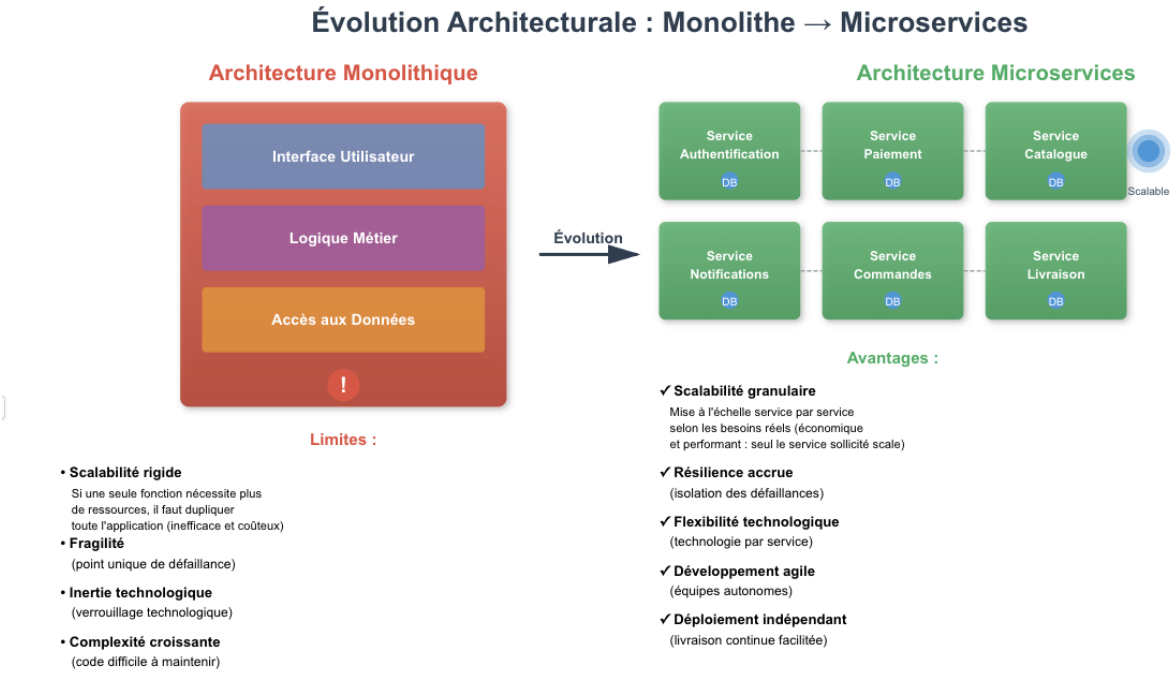

FIGURE 1 – Architecture monolithique vs. microservices.

L'architecture orientée microservices est une réponse directe à ces défis. L'idée fondamentale est de décomposer une application complexe en une collection de **services plus petits, autonomes et faiblement couplés**, chacun étant organisé autour d'une capacité métier spécifique. Cette approche offre une meilleure résilience, une plus grande scalabilité et une flexibilité technologique accrue.

1.2 Notre Boîte à Outils : Spring Boot et Spring Cloud

Pour construire cette architecture, nous n'partons pas de zéro. Nous nous appuierons sur deux piliers de l'écosystème Java :

— **Spring Boot** : C'est le socle qui nous permettra de développer chaque microservice individuellement et rapidement. Il simplifie drastiquement la création d'applications Spring autonomes et prêtes pour la production, avec un serveur web embarqué et une configuration minimale.

— **Spring Cloud** : Si Spring Boot construit les maisons (nos services), Spring Cloud construit la ville (notre système distribué). Il fournit un ensemble d'outils pour résoudre les problèmes courants dans les architectures distribuées : la découverte de services, la gestion de la configuration, le routage intelligent, la tolérance aux pannes, etc.

1.3 L'Architecture Cible de notre TP

Dans ce TP, nous allons construire un système de e-commerce simple mais complet, composé de deux types de services.

1.3.1 Les Services Métier : Le Cœur de l'Application

Ce sont les services qui implémentent la logique fonctionnelle. Chaque service sera totalement indépendant, avec sa propre base de données (ici, une base MySQL), illustrant le principe de "décentralisation des données". Nous en construirons trois :

— `client-service` : Gère toutes les informations relatives aux clients.

— `produit-service` : Responsable du catalogue de produits.

— `commande-service` : Orchestre la logique de passage de commande en communiquant avec les deux autres services.

La structure interne des données pour chacun de ces services est détaillée dans la Figure 2.

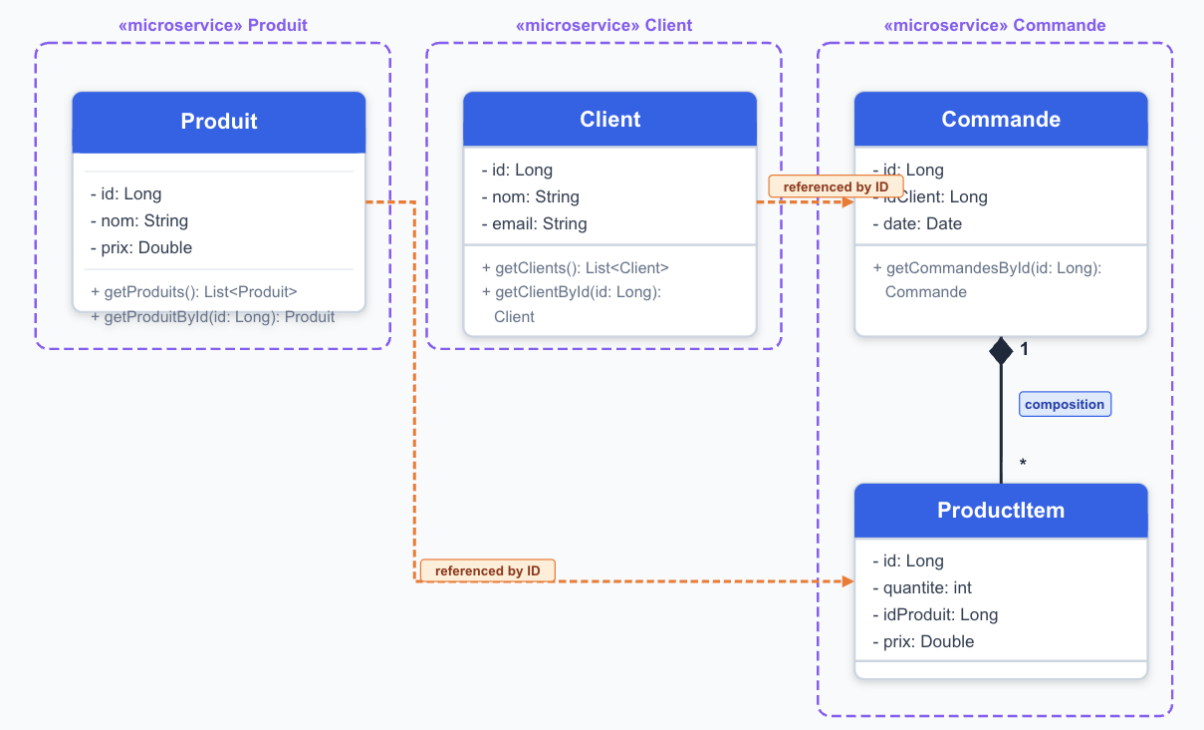

FIGURE 2 – Diagramme de classes détaillant les entités de chaque service métier.

1.3.2 Les Services d'Infrastructure : Le Ciment de l'Architecture

Passer à une architecture distribuée introduit de nouveaux défis. Comment les services communiquent-ils entre eux ? Comment les clients externes (ex : une application web ou mobile) accèdent-ils à notre système de manière sécurisée et cohérente ? C'est le rôle des services d'infrastructure.

— **Discovery Service (avec Eureka)** : Imaginez un annuaire téléphonique pour nos microservices. Dans un environnement cloud, les adresses IP des services peuvent changer dynamiquement. Il est donc impossible de les coder en dur. Le Discovery Service résout ce problème : chaque service s'enregistre auprès de lui à son démarrage. Quand un service A a besoin de parler à un service B, il demande simplement à Eureka : "Quelle est l'adresse actuelle de B ?". Cela permet une communication dynamique et résiliente.

— **API Gateway (avec Spring Cloud Gateway)** : C'est le **point d'entrée unique** pour toutes les requêtes venant de l'extérieur. Au lieu d'exposer tous nos services sur Internet, nous n'exposons que la passerelle. Elle agit comme un portier intelligent qui redirige chaque requête vers le bon microservice interne. Cela offre de multiples avantages : sécurité renforcée (les services internes sont cachés), simplification pour les clients, et un endroit centralisé pour gérer des problématiques transverses comme l'authentification, le logging, la limitation de débit (rate limiting), etc.

L'interaction globale entre les services métier et les services d'infrastructure est illustrée dans la Figure 3.

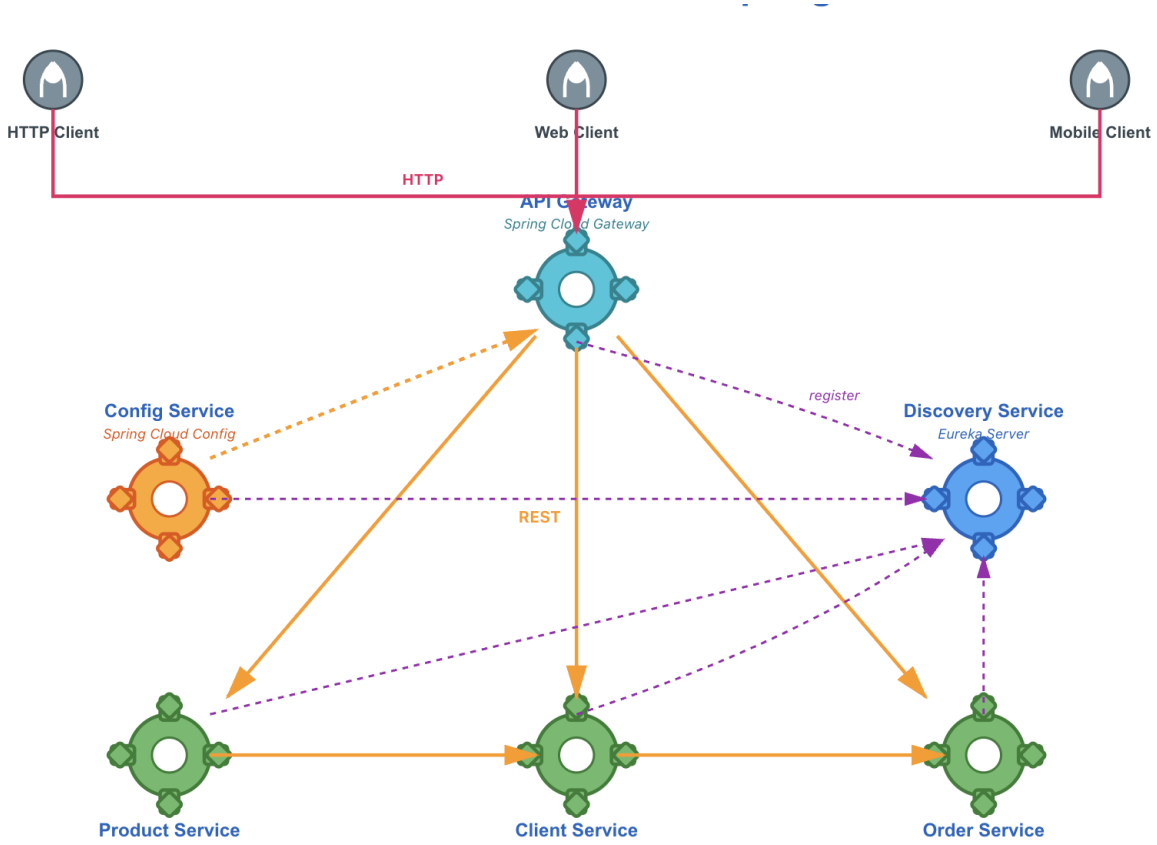

FIGURE 3 – Schéma de l'architecture microservices cible.

Ce TP vous guidera pas à pas dans la construction de cet écosystème, vous permettant de maîtriser les concepts fondamentaux qui animent les applications cloud-natives modernes.

2 Étape 1 : Initialisation de l'Environnement de Travail

Une bonne organisation est la clé. Nous utilisons un projet parent Maven multi-modules pour centraliser la gestion des dépendances et simplifier la construction du projet global. Ce parent contiendra les cinq modules Spring Boot : `client-service`, `produit-service`, `commande-service`, `discovery-service`, et `gateway-service`.

3 Étape 2 : Développement des Microservices Métier

Nous commençons par les services qui portent la logique applicative. Chaque service sera autonome, avec sa propre base de données.

3.1 Le service `client-service`

3.1.1 Dépendances

Le fichier `pom.xml` déclare les "outils" dont notre service a besoin.

```
<dependencies>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>


    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
<!-- Permet à ce service de s'enregistrer auprès d'Eureka -->
     <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    </dependency>
</dependencies>
```

Listing 1 – pom.xml pour client-service

3.1.2 Configuration

Le fichier `application.properties` est le panneau de contrôle de notre service.

```
# L'identifiant unique du service. C'est ce nom qui sera enregistré dans Eureka.
spring.application.name=client-service
server.port=8082
spring.datasource.url=jdbc:mysql://localhost:3306/db_service_client?
    createDatabaseIfNotExist=true
spring.datasource.username=root
```

```
spring.datasource.password=votremotdepasse
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true

# Indique au client Eureka où trouver le serveur (l'annuaire)
eureka.client.service-url.default-zone=http://localhost:8761/eureka/
```

Listing 2 – application.properties pour client-service

3.1.3 Implémentation des classes Java

```
@Entity @Data @NoArgsConstructor @AllArgsConstructor
public class Client {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nom;
    private String email;
}
```

Listing 3 – Entité Client.java

```
@RestController
@AllArgsConstructor
public class ClientController {
    @Autowired private ClientRepository clientRepository;

    @GetMapping("/clients")
    public List<Client> getAllClients() { return clientRepository.findAll(); }

    @GetMapping("/clients/{id}")
    public Client getClientById(@PathVariable Long id) {
        return clientRepository.findById(id).orElse(null);
    }
}
```

Listing 4 – Controller ClientController.java

3.2 Le service

Ce service suit la même architecture que `client-service`. Adaptez la configuration pour utiliser le port `8081` et la base de données `db_service_produit`. L'entité, le repository et le controller seront créés sur le modèle de ceux du client.

4 Étape 3 : Mise en Place de l'Infrastructure Cloud

Ces services sont les piliers de notre architecture distribuée.

4.1 Le service de découverte (`discovery-service`)

Note Pédagogique

Le rôle d'Eureka : Imaginez Eureka comme l'annuaire téléphonique de votre architecture. Chaque service, au démarrage, l'appelle pour s'y inscrire avec son nom (ex : `client-service`) et son adresse (IP :Port). Quand un service a besoin d'en appeler un autre, il demande à Eureka : "Donne-moi l'adresse de `client-service`", ce qui évite d'avoir à coder en dur les adresses.

— **Dépendances** : `spring-cloud-starter-netflix-eureka-server`.

— **Annotation** : `@EnableEurekaServer` sur la classe principale. Cette annotation transforme une application Spring Boot standard en un serveur de découverte.
— **Configuration** :

```
spring.application.name=discovery-service
server.port=8761

# Point Crucial : Un serveur Eureka est aussi un client par défaut.
# Ces deux lignes lui indiquent de ne PAS essayer
# de s'enregistrer auprès de lui-même.
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

Listing 5 – application.properties pour discovery-service

4.2 La passerelle API (`gateway-service`)

Note Pédagogique

Le rôle de l'API Gateway : La passerelle est le portier de notre système. Toutes les requêtes venant de l'extérieur (ex : une application web, une application mobile) doivent passer par elle. Elle se charge ensuite de router la requête vers le bon microservice interne. Cela offre une sécurité accrue (les services internes ne sont pas exposés directement) et un point central pour gérer des problématiques transverses (authentification, logging, etc.).

Nous optons ici pour le routage dynamique, l'approche la plus flexible.
— **Dépendances** : `spring-cloud-starter-gateway` et `spring-cloud-starter-netflix-eureka-client`.
— **Configuration** : Le fichier de configuration est minimaliste.

```
spring.application.name=gateway-service
server.port=8887
eureka.client.service-url.default-zone=http://localhost:8761/eureka/
#spring.cloud.gateway.server.webflux.discovery.locator.lower-case-service-id=
    true
```

Listing 6 – application.properties pour gateway-service

4.2.1 Découverte dynamique des routes

Ajoutez le bean suivant dans une classe de configuration (ou la classe principale) du `gateway-service`.

```
@Configuration
public class GatewayConfig {
    @Bean
    public DiscoveryClientRouteDefinitionLocator
    discoveryClientRouteDefinitionLocator(
            ReactiveDiscoveryClient rdc,
            DiscoveryLocatorProperties dlp) {
        return new DiscoveryClientRouteDefinitionLocator(rdc, dlp);
    }
}
```

Listing 7 – Activation du routage dynamique via un Bean

Ce bean est le pilote automatique de notre passerelle. Il fonctionne en trois temps :

1. **Il consulte Eureka** en permanence pour connaître la liste des services enregistrés.
2. **Il crée une route en mémoire** pour chaque service.
3. **Il applique une convention de nommage** : par défaut, un service enregistré avec l'ID `CLIENT-SERVICE` sera automatiquement accessible via le chemin `/client-service/**` sur la passerelle.

Cette approche est puissante car elle ne nécessite aucune modification de la passerelle lorsqu'un nouveau microservice est ajouté.

5 Étape 4 : Orchestration avec le Service de Commande

Ce service est le cœur de notre logique métier distribuée. Il a une double responsabilité :

1. **Être un service métier classique** : Il gère ses propres données et sa propre logique, à savoir les commandes et les lignes de produits qui les composent. Pour cela, il possède sa propre base de données avec deux entités liées : `Commande` et `ProductItem`.
2. **Être un orchestrateur** : Il ne se contente pas de ses données. Pour fournir une vue complète d'une commande, il doit communiquer avec `client-service` et `produit-service` pour enrichir ses données locales avec les informations complètes sur le client et les produits.

Pour réaliser cette communication, nous allons utiliser **OpenFeign**, un outil qui transforme des appels d'API REST en simples appels de méthodes Java.

Note Pédagogique

Le rôle d'OpenFeign : Un assistant pour la communication

Pensez à OpenFeign comme un **assistant personnel** qui permet à vos services de communiquer entre eux sans effort. Vous lui donnez une simple "liste de tâches" (une interface Java) et il s'occupe de tout le travail complexe en arrière-plan :

- **Trouver :** Il demande à l'annuaire (Eureka) où se trouve le bon service à contacter.
- **Répartir la charge :** Si un service a plusieurs copies, il distribue intelligemment les appels entre elles pour éviter les surcharges.
- **Communiquer :** Il envoie la requête sur le réseau de la bonne manière.
- **Traduire :** Il reçoit la réponse technique (souvent en JSON) et la transforme en un objet Java simple, prêt à être utilisé dans votre code.

En bref, vous déclarez **ce que vous voulez faire**, et OpenFeign se charge de **comment le faire**.

- **Dépendances** : Ajoutez `spring-cloud-starter-openfeign`, `spring-boot-starter-web`, `spring-boot-starter-data-jpa`, `spring-cloud-starter-netflix-eureka-client`, lombok et le connecteur MySQL à votre `pom.xml`.
- **Annotation** : Annotez votre classe principale avec `@EnableFeignClients` pour activer la magie de Feign.

5.1 Configuration

Le fichier `application.properties` configure l'identité du service, sa connexion à la base de données et son enregistrement auprès d'Eureka.

```
# 1. Identification du microservice
spring.application.name=commande-service
server.port=8083

# 2. Configuration de la base de donnees locale (MySQL)
spring.datasource.url=jdbc:mysql://localhost:3306/db_service_commande?
    createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=votremotdepasse

# 3. Configuration de JPA/Hibernate
spring.jpa.hibernate.ddl-auto=update
```

```
spring.jpa.show-sql=true

# 4. Configuration du client Eureka
# Indique au service ou se trouve l'annuaire pour s'y enregistrer
eureka.client.service-url.default-zone=http://localhost:8761/eureka/
```

Listing 8 – Fichier application.properties pour commande-service

5.2 Implémentation de la persistance locale

Nous créons d'abord les entités qui seront stockées dans la base de données `db_service_commande`.

5.2.1 Les Entités JPA

Notez l'utilisation de `@Transient`. Ces champs ne seront pas des colonnes dans la base de données. Ce sont des conteneurs temporaires que nous remplirons en appelant les autres micro-services.

```
package com.poly.servicecommande.entities;
//... imports
@Entity @Data @NoArgsConstructor @AllArgsConstructor
public class Commande {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Date date;
    private Long idClient; // L'ID du client, stocke localement

    @Transient // Ne sera pas persiste en base
    private Client client; // L'objet Client complet, qui sera recupere du
    client-service

    @OneToMany(mappedBy = "commande")
    private List<ProductItem> productItems;
}
```

Listing 9 – Entité Commande.java

```
package com.poly.servicecommande.entities;
//... imports
@Entity @Data @NoArgsConstructor @AllArgsConstructor
public class ProductItem {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Long idProduit; // L'ID du produit, stocke localement
    private int quantite;
    private double prix;

    @Transient // Ne sera pas persiste en base
    private Produit produit; // L'objet Produit complet, recupere du produit-
    service

    @ManyToOne
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY) // Evite les boucles
    infinies
    private Commande commande;
}
```

Listing 10 – Entité ProductItem.java

5.2.2 Les "Models" pour les données distantes

Ce sont de simples classes Java qui représentent la structure des données que nous allons recevoir des autres services. Elles doivent être placées dans un package, par exemple `com.poly.servicecommande.model`.

```
package com.poly.servicecommande.model;
//... imports
@Data
public class Produit {
    private Long id;
    private String nom;
    private double prix;

}
```

Listing 11 – Model model/Produit.java

```
package com.poly.servicecommande.model;
//... imports
@Data
public class Client {
    private Long id;
    private String nom;
    private String email;
}
```

Listing 12 – Model model/Client.java

5.2.3 Les Repositories JPA

Créez les interfaces standards pour l'accès aux données dans le package `com.poly.servicecommande.repository`.

```
@Repository
public interface CommandeRepository extends JpaRepository<Commande, Long> { }

@Repository
public interface ProductItemRepository extends JpaRepository<ProductItem, Long>
    { }
```

Listing 13 – Repository repository/CommandeRepository.java et repository/ProductItemRepository.java

5.3 Implémentation de la communication inter-services

Nous définissons maintenant les "clients Feign". C'est une bonne pratique de les isoler dans leur propre package, par exemple `com.poly.servicecommande.feign`, afin de bien séparer le code de communication externe.

Pour que notre code d'initialisation fonctionne, nous devons également ajouter des méthodes pour récupérer la liste complète des clients et des produits.

```
package com.poly.servicecommande.feign;

import com.poly.servicecommande.model.Client;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import java.util.List;

// Le nom "CLIENT-SERVICE" doit correspondre au ‘spring.application.name‘ du
    service cible.
```

```java
@FeignClient(name = "CLIENT-SERVICE")
public interface ClientRestClient {
    // La signature doit correspondre a celle du Controller du service client.
    @GetMapping("/clients/{id}")
    Client findClientById(@PathVariable Long id);

    @GetMapping("/clients")
    List<Client> getClients();
}
```

Listing 14 – Interface feign/ClientRestClient.java

```java
package com.poly.servicecommande.feign;

import com.poly.servicecommande.model.Produit;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import java.util.List;

@FeignClient(name = "PRODUIT-SERVICE")
public interface ProduitRestClient {
    @GetMapping("/produits/{id}")
    Produit findProduitById(@PathVariable Long id);

    @GetMapping("/produits")
    List<Produit> getProduits();
}
```

Listing 15 – Interface feign/ProduitRestClient.java

5.4 L'Orchestration dans le Controller

C'est ici que tout se met en place. Le controller, situé dans `com.poly.servicecommande.web`, va utiliser les repositories pour accéder à ses données locales et les clients Feign pour aller chercher les données distantes et assembler le tout.

```java
package com.poly.servicecommande.web;
//... imports
@RestController
@AllArgsConstructor
public class CommandeController {
    private CommandeRepository commandeRepository;
    private ProductItemRepository productItemRepository;
    private ClientRestClient clientRestClient;
    private ProduitRestClient produitRestClient;

    @GetMapping("/commandes/{id}")
    public Commande getCommandeDetails(@PathVariable Long id) {
        // 1. Recuperer les donnees brutes de la commande depuis la base locale
        Commande commande = commandeRepository.findById(id).orElse(null);
        if (commande == null) return null;

        // 2. Enrichir la commande en appelant le client-service via Feign
        Client client = clientRestClient.findClientById(commande.getIdClient());
        commande.setClient(client);

        // 3. Enrichir chaque article en appelant le produit-service
        commande.getProductItems().forEach(pi -> {
            Produit produit = produitRestClient.findProduitById(pi.getIdProduit
    ());
            pi.setProduit(produit);
        });
```

```
26
27          // 4. Retourner l'objet de donnees completes et aggregees
28          return commande;
29      }
30  }
```

Listing 16 – Controller web/CommandeController.java - L'orchestrateur

5.5 Initialisation des données de test au démarrage

Pour pouvoir tester notre endpoint `/commandes/{id}` sans avoir une base de données vide, nous pouvons utiliser un `CommandLineRunner`. Ce composant Spring s'exécute une seule fois, juste après le démarrage de l'application.

Notre objectif est de créer un jeu de données de test de manière dynamique :

— Il va d'abord appeler `client-service` et `produit-service` pour récupérer la liste de tous les clients et produits existants.

— Ensuite, pour chaque client trouvé, il créera une nouvelle `Commande` dans sa propre base de données.

— Finalement, il ajoutera chaque produit comme un `ProductItem` à la commande de ce client.

C'est une démonstration parfaite du rôle d'orchestrateur, utilisant des données distantes pour créer et persister ses propres données locales.

```
1  package com.poly.servicecommande;
2
3  import com.poly.servicecommande.entities.Commande;
4  import com.poly.servicecommande.entities.ProductItem;
5  import com.poly.servicecommande.feign.ClientRestClient;
6  import com.poly.servicecommande.feign.ProductRestClient;
7  import com.poly.servicecommande.model.Client;
8  import com.poly.servicecommande.model.Produit;
9  import com.poly.servicecommande.repository.CommandeRepository;
10 import com.poly.servicecommande.repository.ProductItemRepository;
11 import org.springframework.boot.CommandLineRunner;
12 import org.springframework.boot.SpringApplication;
13 import org.springframework.boot.autoconfigure.SpringBootApplication;
14 import org.springframework.cloud.openfeign.EnableFeignClients;
15 import org.springframework.context.annotation.Bean;
16
17 import java.util.Date;
18 import java.util.List;
19
20 @EnableFeignClients // Active la détection et la création automatique des
       clients Feign
21 @SpringBootApplication
22 public class ServiceCommandeApplication {
23
24     public static void main(String[] args) {
25         SpringApplication.run(ServiceCommandeApplication.class, args);
26     }
27
28     @Bean
29     CommandLineRunner commandLineRunner(
30             CommandeRepository commandeRepository,
31             ProductItemRepository productItemRepository,
32             ProductRestClient productRestClient,
33             ClientRestClient clientRestClient
34     ) {
35         return args -> {
36             // 1. Récupérer les données des autres services
```

```
37              List<Client> clients = clientRestClient.getClients();
38              List<Produit> produits = productRestClient.getProduits();
39
40              if (clients.isEmpty() || produits.isEmpty()) {
41                  System.out.println("Aucun client ou produit trouvé. Impossible d
    'initialiser les commandes.");
42                  return;
43              }
44
45              // 2. Pour chaque client, créer une commande contenant tous les
    produits
46              clients.forEach(client -> {
47                  Commande commande = new Commande();
48                  commande.setIdClient(client.getId());
49                  commande.setDate(new Date());
50                  Commande savedCommande = commandeRepository.save(commande);
51
52                  produits.forEach(produit -> {
53                      ProductItem productItem = new ProductItem();
54                      productItem.setIdProduit(produit.getId());
55                      productItem.setCommande(savedCommande);
56                      productItem.setPrix(produit.getPrix()); // On utilise le
    prix actuel du produit
57                      productItem.setQuantite(1 + (int)(Math.random() * 10)); //
    Quantité aléatoire entre 1 et 10
58                      productItemRepository.save(productItem);
59                  });
60              });
61
62              System.out.println("Jeu de données de commandes créé avec succès !")
    ;
63          };
64      }
65  }
```

Listing 17 – Classe principale ServiceCommandeApplication.java

6 Étape 5 : Tolérance aux Pannes et Résilience

Dans une architecture distribuée, les services communiquent via le réseau. Si le `client-service` tombe en panne, le `commande-service` risque de bloquer en attendant une réponse.

Pour gérer cela, nous allons utiliser le pattern **Circuit Breaker** directement dans notre interface OpenFeign grâce à l'annotation `@CircuitBreaker` et aux **Default Methods** de Java 8.

6.1 Configuration

Ajoutez la dépendance Resilience4j dans le `pom.xml` du `commande-service` :

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
</dependency>
```

Listing 18 – Dépendances Resilience4j

6.2 Implémentation dans l'interface OpenFeign

C'est ici que réside toute l'astuce. Nous allons :

1. Annoter la méthode de requête avec `@CircuitBreaker`.

2. Définir le comportement de secours via une **default method** directement dans l'interface.

Cette approche centralise la définition de l'API et sa stratégie de résilience au même endroit.

```java
package com.poly.servicecommande.feign;

import com.poly.servicecommande.model.Client;
import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import java.util.List;
import java.util.ArrayList;

@FeignClient(name = "CLIENT-SERVICE")
public interface ClientRestClient {

    @GetMapping("/clients/{id}")
    @CircuitBreaker(name = "client-service-cb", fallbackMethod = "
    getDefaultClient")
    Client findClientById(@PathVariable Long id);

    ...

    // --- METHODES PAR DEFAUT (FALLBACK) ---
    // Elles doivent etre definies comme "default" car nous sommes dans une
    interface.
    // La signature doit etre identique a la methode originale + le parametre
    Exception.

    default Client getDefaultClient(Long id, Exception exception) {
        Client client = new Client();
        client.setId(id);
        client.setNom("Client Non Disponible");
        client.setEmail("defaut@error.com");
        return client;
    }


}
```

Listing 19 – Interface feign/ClientRestClient.java avec Default Method

7 Étape 6 : Lancement et Validation

7.1 Ordre de Démarrage

Note Pédagogique

Pourquoi cet ordre est-il crucial ?

1. **Eureka en premier** : L'annuaire doit être ouvert avant que les services ne tentent de s'y inscrire.
2. **Services métier ensuite** : Ils doivent s'enregistrer auprès d'Eureka pour être "découvrables".
3. **Gateway en dernier** : La passerelle doit démarrer en dernier pour qu'au moment où elle consulte Eureka, elle trouve une liste complète des services prêts à recevoir du trafic.

7.2 Vérification et Tests

1. **Tableau de bord Eureka** : Accédez à `http://localhost:8761`. La page doit lister tous vos services (GATEWAY-SERVICE, CLIENT-SERVICE, PRODUIT-SERVICE, COMMANDE-SERVICE) avec le statut `UP`.

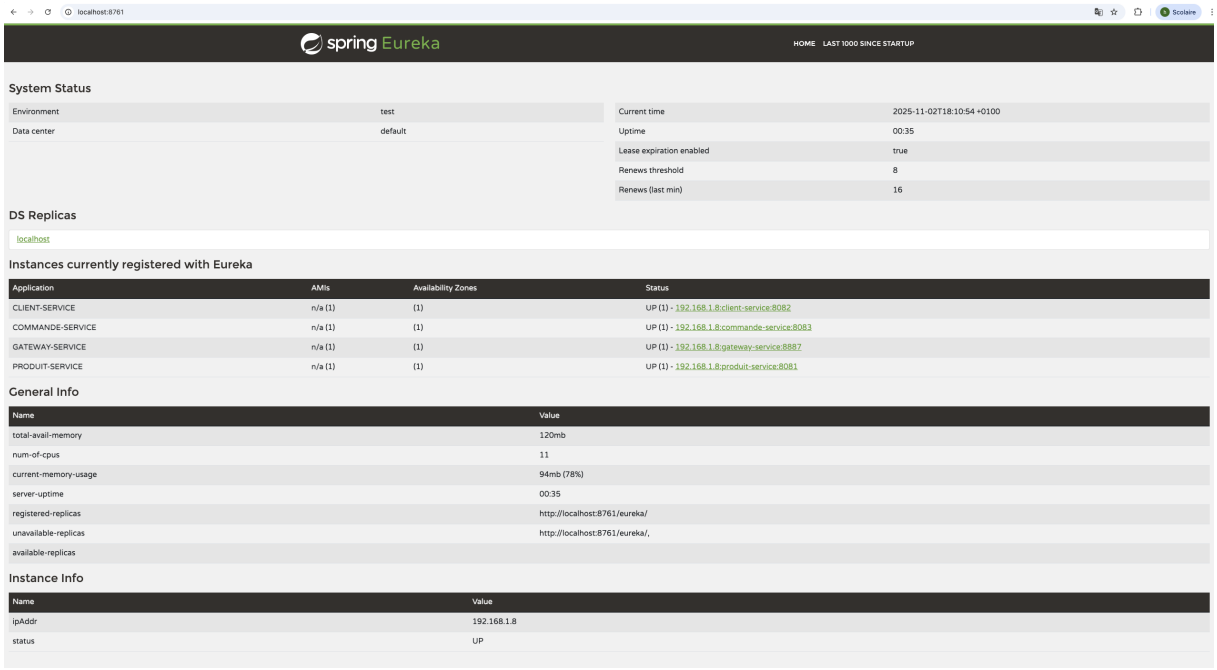

FIGURE 4 – Tableau de bord Eureka affichant les services enregistrés

2. **Tests via la passerelle** : Toutes les requêtes externes doivent passer par la passerelle sur le port `8887`. La structure de l'URL suit la convention du routage dynamique : `http://<gateway>/<service-id>/<endpoint>`.
   — Lister les clients :
   ```
   curl http://localhost:8887/CLIENT-SERVICE/clients
   ```
   — Lister les produits :
   ```
   curl http://localhost:8887/PRODUIT-SERVICE/produits
   ```
   — Récupérer une commande complète (agrégée) :
   ```
   curl http://localhost:8887/COMMANDE-SERVICE/commandes/1
   ```