

2. Orchestration avec Spring Cloud

2.1 Introduction à Spring Cloud et Eureka

Spring Cloud est un framework open-source conçu pour simplifier le développement d'applications distribuées. Il offre des outils permettant de créer des systèmes robustes et évolutifs, construits selon une architecture de microservices. Eureka, en tant que service de découverte, permet l'enregistrement des microservices et leur découverte au sein du réseau, facilitant ainsi leur interaction.

2.2 Caractéristiques Principales de Spring Cloud

Coordination des Microservices

Spring Cloud propose des mécanismes de coordination et de gestion des microservices, facilitant la découverte de services, la gestion de la configuration ainsi que la résilience des systèmes distribués. Ces fonctionnalités permettent de concevoir des architectures flexibles et adaptées aux évolutions de l'environnement.

Découverte de Services avec Eureka

Eureka, un composant central de Spring Cloud, permet à chaque microservice de s'enregistrer dans un registre. Les services clients interrogent ce registre pour identifier les instances disponibles d'un service donné. Cette approche optimise la communication entre les microservices dans une architecture où chaque composant évolue indépendamment.

Gestion de Configuration

La gestion des configurations est centralisée grâce à Spring Cloud Config. Les paramètres de configuration sont stockés dans des référentiels tels que Git, offrant une gestion versionnée et sécurisée. Une modification dans la configuration est appliquée dynamiquement sans nécessiter un redéploiement, ce qui améliore la souplesse dans les environnements de production.

Résilience et Tolérance aux Pannes

Spring Cloud intègre des outils comme Hystrix pour renforcer la résilience et assurer une tolérance aux pannes. En cas de défaillance d'un composant, Hystrix offre des solutions de repli (fallbacks), évitant ainsi les échecs en cascade. Cela garantit une disponibilité élevée et un comportement stable même dans des conditions adverses.

API Gateway

L'utilisation de Spring Cloud Gateway permet de centraliser les requêtes entrantes, d'assurer leur routage vers les microservices appropriés et d'appliquer des règles de sécurité. Ce composant joue un rôle crucial dans la gestion cohérente du trafic et la mise en œuvre des politiques de sécurité.

Intégration avec Spring Boot

Spring Cloud s'intègre avec Spring Boot, simplifiant ainsi le développement de microservices. Chaque microservice bénéficie des outils offerts par Spring Boot pour la configuration, tout en utilisant les fonctionnalités de gestion distribuée proposées par Spring Cloud.

2.3 Architecture de Microservices avec Spring Cloud et Eureka

L'architecture de microservices s'appuie sur des principes comme la répartition des responsabilités, l'autonomie des services et l'utilisation de protocoles légers pour la communication. La figure 2.1 illustre les interactions entre les différents composants dans un système typique.

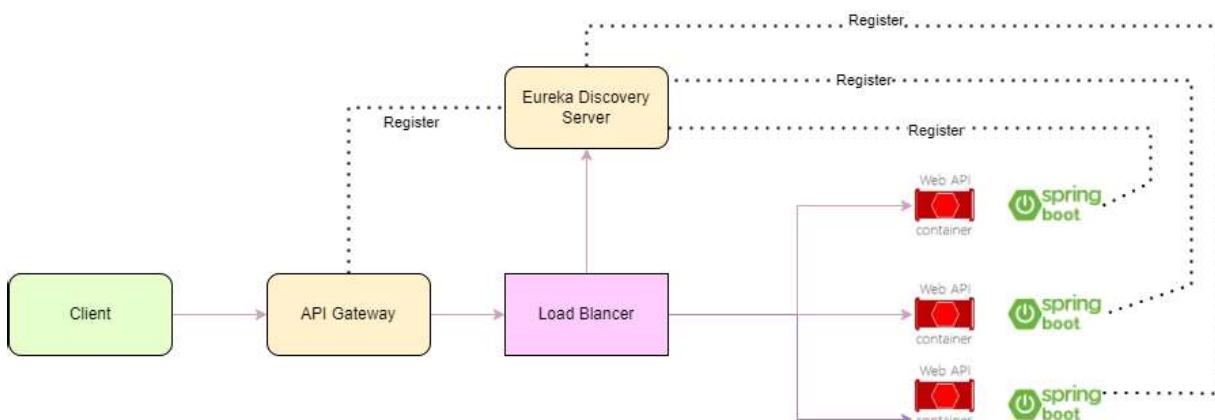


Figure 2.1: Architecture de microservices utilisant Spring Cloud, Eureka, un Load Balancer et une API Gateway.

Voici une description des composants représentés dans l'architecture :

- **Client** : Entité envoyant des requêtes à l'application, comme un utilisateur final ou une autre application.
- **API Gateway** : Point d'entrée unique gérant les requêtes, leur routage et l'application des règles de sécurité.
- **Load Balancer** : Composant assurant la distribution des requêtes entre les instances des microservices.
- **Eureka Discovery Server** : Registre central où chaque microservice s'enregistre. Ce serveur facilite la découverte dynamique des services disponibles.
- **Microservices (Web API containers)** : Applications Spring Boot encapsulées dans des conteneurs. Chaque microservice est conçu pour gérer une tâche spécifique.

Le fonctionnement suit plusieurs étapes :

- Un microservice s'enregistre auprès du serveur Eureka avec ses informations de connexion.
- Le Load Balancer et les autres microservices interrogent le serveur pour obtenir la liste des instances disponibles.

- Une requête est d'abord dirigée vers l'API Gateway avant d'être routée par le Load Balancer.
- Le Load Balancer distribue la requête vers une instance de microservice.

2.4 Avantages de Spring Cloud avec Eureka

- **Simplicité de Développement** : Réduction de la complexité liée à la gestion des systèmes distribués.
- **Scalabilité et Évolutivité** : Chaque microservice peut évoluer indépendamment.
- **Gestion Centralisée** : Une configuration centralisée facilite l'administration des systèmes.
- **Intégration Transparente** : Compatibilité optimale avec l'écosystème Spring.



L'utilisation de Spring Cloud avec Eureka améliore considérablement le développement d'architectures distribuées. L'intégration des fonctionnalités de découverte, de gestion de la configuration et de résilience permet de concevoir des applications performantes, évolutives et facilement maintenables.

2.5 Activité pratique

2.5.1 Architecture Micro-services avec FeignClient

Objectif

Ce TP a pour objectif de favoriser une compréhension approfondie de l'architecture microservice. Les axes essentiels de cet apprentissage incluent la création et l'enregistrement de microservices, la connexion à une base de données In-memory H2, la mise en place d'un microservice Gateway et l'implémentation d'une communication synchrone entre les microservices à l'aide de l'outil OPENFEIGN.

Ce TP adopte une architecture basée sur les microservices, caractérisée par la décomposition d'une application en de petits services indépendants. Au cœur de cette structure se trouvent les microservices clients, des entités autonomes qui interagissent pour fournir une fonctionnalité complète. L'API Gateway joue le rôle de point d'entrée centralisé, simplifiant la gestion des requêtes en dirigeant le trafic vers les microservices appropriés. Le serveur de découverte Eureka joue un rôle essentiel en permettant à chaque microservice de s'enregistrer de manière dynamique, constituant ainsi un annuaire décentralisé des services disponibles.

A. Création du service Discovery Eureka

La mise en place d'un service discovery Eureka nécessite de suivre les étapes suivantes :

- 1. Crédit du projet**

Créer un nouveau projet sur Spring Initializr avec le nom Eureka Server.

- 2. Ajout de la dépendance**

Ajouter la dépendance suivante, puis cliquer sur **Generate** pour générer le projet :



3. Cliquer sur `src/main/ressources` et ajouter les trois lignes suivantes :

```
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

- 4. Configuration des propriétés**

Accéder au dossier `src/main/resources` et modifier le fichier `application.properties` en ajoutant les lignes suivantes :

```
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

- 5. Activation du serveur Eureka**

Ajouter l'annotation `@EnableEurekaServer` dans la classe `EurekaServerApplication` :

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import
org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
```

```

@EnableEurekaServer
@SpringBootApplication
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }
}

```

6. Lancement et vérification du serveur Eureka

Exécuter le projet. Ouvrir un navigateur et accéder à l'URL suivante : <http://localhost:8761/>

Une page web s'affiche, indiquant que le serveur Eureka est opérationnel.

The screenshot shows the Spring Eureka dashboard. At the top, there's a header with the Spring logo and the word "Eureka". On the right, it says "HOME LAST 1000 SINCE STARTUP". Below the header, there's a section titled "System Status" with a table:

Environment	test	Current time	2023-12-08T11:28:37 +0100
Data center	default	Uptime	00:00
		Lease expiration enabled	false
		Renews threshold	1
		Renews (last min)	0

Below the status table, there's a section titled "DS Replicas" with a sub-section "Instances currently registered with Eureka". It shows a table with one row:

Application	AMIs	Availability Zones	Status
No instances available			

L'Eureka Server est désormais opérationnel, prêt à enregistrer et découvrir les microservices au sein du système.

B. Création du service Client (1er microservice)

La mise en place d'un service client repose sur plusieurs étapes essentielles, allant de la création du projet à la configuration du microservice et à la mise en place de la base de données.

1. Création du projet

Créer un nouveau projet à l'aide de Spring Initializr avec le nom Client.

2. Ajout des dépendances

Ajouter les dépendances nécessaires au projet, puis cliquer sur **Generate** pour générer le projet :

The screenshot shows a list of dependencies for a Spring Boot application:

- Spring Boot DevTools** [DEVELOPER TOOLS]: Provides fast application restarts, LiveReload, and configurations for enhanced development experience.
- Lombok** [DEVELOPER TOOLS]: Java annotation library which helps to reduce boilerplate code.
- Spring Boot Actuator** [OPS]: Supports built-in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc.
- Spring Data JPA** [SQL]: Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
- H2 Database** [SQL]: Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.
- Eureka Discovery Client** [SPRING CLOUD DISCOVERY]: A REST based service for locating services for the purpose of load balancing and failover of middle-tier servers.
- Spring Web** [WEB]: Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Les dépendances ajoutées sont les suivantes :

- **Spring Boot Actuator** : Prise en charge des points de terminaison permettant de surveiller et de gérer l'application (état de santé, métriques, etc.).
- **Eureka Discovery Client** : Permet d'enregistrer le microservice sur Eureka afin qu'il soit détectable par d'autres microservices.
- **H2** : Base de données en mémoire légère et rapide.
- **Spring Data JPA** : Facilite la manipulation des bases de données relationnelles via des entités.
- **Spring Web** : Fournit les fonctionnalités nécessaires pour créer des API REST.
- **Spring Boot Devtools** : Améliore l'expérience de développement en permettant des redémarrages rapides.
- **Rest Repositories** : Expose les dépôts JPA sous forme de services REST.
- **Lombok** : Simplifie la création des classes Java en générant automatiquement les getters, setters et constructeurs.

3. Configuration de la classe principale

Dans la classe ClientApplication, ajouter l'annotation @EnableDiscoveryClient pour permettre l'enregistrement du microservice auprès du serveur Eureka :

```

@EnableDiscoveryClient
@SpringBootApplication
public class ClientApplication {
    public static void main(String[] args) {
        SpringApplication.run(ClientApplication.class, args);
    }
}

```

Cette annotation permet au service de s'enregistrer auprès du serveur de découverte Eureka.

4. Configuration du fichier de propriétés

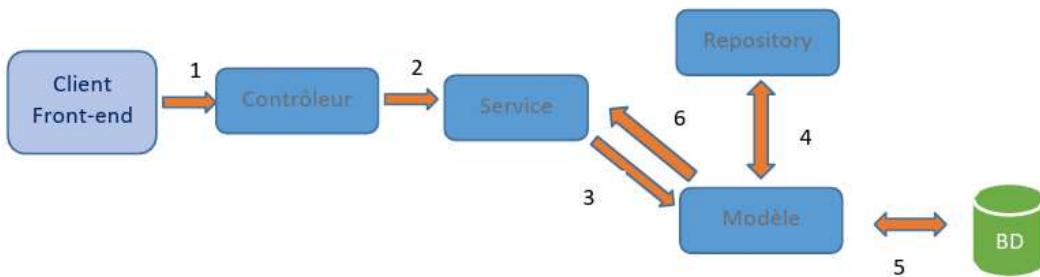
Modifier le fichier `application.properties` dans `src/main/resources` en y ajoutant les propriétés suivantes :

```
server.port=8088
spring.application.name=SERVICE-CLIENT
```

Cette configuration définit le port du service sur 8088 et le nom du service sur SERVICE-CLIENT, ce qui facilitera sa découverte par Eureka.

5. Architecture du microservice

L'architecture suit le modèle multi-couches, organisé de la manière suivante :



Cette structure comprend les couches suivantes :

- **Controller** : Gère les requêtes HTTP entrantes et déclenche les actions correspondantes.
- **Service** : Contient la logique métier.
- **Repository** : Assure la communication avec la base de données.
- **Entity** : Représente les données sous forme d'objets Java.

6. Création des packages

Créer les sous-packages suivants dans le package principal :

- `entities` : Contient les classes d'entités.
- `repositories` : Contient les interfaces JPARepository.
- `controllers` : Contient les classes contrôleurs.
- `services` : Contient la logique métier.

7. Création de l'entité Client

Créer la classe Client dans le package `entities` :

```

@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Client {
    @Id
    @GeneratedValue
    private Long id;
    private String nom;
    private Float age;
}
  
```

La classe est marquée par l'annotation `@Entity`, ce qui permet à JPA de la considérer comme une entité persistante. Les annotations `@Data`, `@AllArgsConstructor`, et `@NoArgsConstructor` proviennent de Lombok et génèrent automatiquement les méthodes getter, setter et les constructeurs.

8. Création du Repository

Créer l'interface ClientRepository dans le package repositories :

```
@Repository
public interface ClientRepository extends JpaRepository<Client, Long> {}
```

L'interface ClientRepository hérite de JpaRepository, ce qui permet de bénéficier de toutes les méthodes de base (findAll, findById, save, delete, etc.).

9. Création du contrôleur

Créer la classe ClientController dans le package controllers :

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

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

    @GetMapping("/client/{id}")
    public Client findById(@PathVariable Long id) throws Exception {
        return clientRepository.findById(id)
            .orElseThrow(() -> new Exception("Client non trouvé"));
    }
}
```

Ce contrôleur expose deux points de terminaison :

- /clients : Récupère la liste de tous les clients.
- /client/id : Récupère un client spécifique à partir de son identifiant.

10. Insertion de données de test

Ajouter des clients dans la base de données H2 à l'aide d'un CommandLineRunner :

```
@Bean
CommandLineRunner initialiserBaseH2(ClientRepository clientRepository) {
    return args -> {
        clientRepository.save(new Client(null, "Rabab SELIMANI", 23f));
        clientRepository.save(new Client(null, "Amal RAMI", 22f));
        clientRepository.save(new Client(null, "Samir SAFI", 22f));
    };
}
```

Cette méthode permet d'insérer automatiquement des données lors du démarrage de l'application.

11. Test des endpoints

Lancer l'application et accéder aux URLs suivantes :

- <http://localhost:8088/clients> : Affiche la liste des clients.
- <http://localhost:8088/client/1> : Affiche les détails du client avec l'ID 1.



12. Vérification de l'enregistrement auprès d'Eureka

Accéder au serveur Eureka via l'URL : <http://localhost:8761/> et vérifier que le service SERVICE-CLIENT est bien enregistré.

The screenshot shows the Spring Eureka dashboard. At the top, it says "spring Eureka" and "HOME LAST 1000 SINCE STARTUP". Below that is a "System Status" table with the following data:

Environment	test	Current time	2023-12-08T12:09:57 +0100
Data center	default	Uptime	00:00
		Lease expiration enabled	false
		Renews threshold	3
		Renews (last min)	0

Below the status table is a section titled "DS Replicas" which lists "Instances currently registered with Eureka". A table shows the following data:

Application	AMIs	Availability Zones	Status
SERVICE-CLIENT	n/a (1)	(1)	UP (1) - localhost:SERVICE-CLIENT:8081

La présence de SERVICE-CLIENT dans la liste des services enregistrés confirme le bon fonctionnement de la communication entre le service et le serveur Eureka.

C. Création d'un service Gateway

Pour créer un service Gateway, il convient de suivre la procédure suivante :

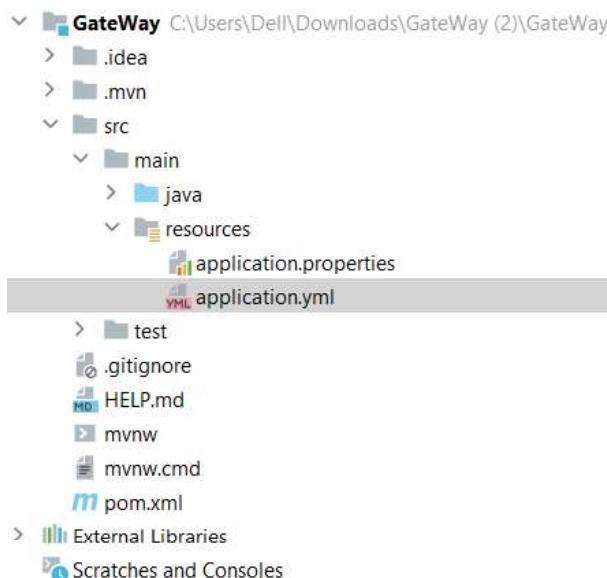
1. Créez un nouveau projet sur Spring Initializr nommé GateWay.
2. Ajoutez les dépendances suivantes et cliquez sur Generate :

The screenshot shows the Spring Initializr interface. On the left, under "Dependencies", there is a button labeled "ADD DEPENDENCIES... CTRL + B". Below it, the "Spring Boot Actuator" dependency is selected, indicated by a green bar. Under "Gateway", the "SPRING CLOUD ROUTING" dependency is selected. Under "Eureka Discovery Client", the "SPRING CLOUD DISCOVERY" dependency is selected. The descriptions for each selected dependency are visible:

- Spring Boot Actuator** [OPS]: Supports built in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc.
- Gateway** [SPRING CLOUD ROUTING]: Provides a simple, yet effective way to route to APIs and provide cross cutting concerns to them such as security, monitoring/metrics, and resiliency.
- Eureka Discovery Client** [SPRING CLOUD DISCOVERY]: A REST based service for locating services for the purpose of load balancing and failover of middle-tier servers.

La configuration d'une GateWay peut se faire avec deux manières :

- Statique via des fichiers yaml et propriétés ou bien via du code Java
 - Dynamique avec du code Java seulement
- (a) Configuration statique :
- Ouvrir le fichier application.properties et ajouter les propriétés suivantes :
- ```
server.port=8888
spring.application.name=Gateway
spring.cloud.discovery.enabled=false
```
- Cette configuration attribue le nom "Gateway" à notre passerelle et définit son port à 8888. Ensuite, désactivez l'enregistrement du service dans le service Discovery (ce service n'est pas nécessaire pour le moment).
- Dans le dossier src/main/resources, créer un fichier YAML nommé application.yml :



YAML (Yet Another Markup Language) est un langage de représentation de données. Il est généralement utilisé par Spring Boot à des fins de configuration. Nous allons l'utiliser ici pour configurer notre passerelle Gateway pour le routage entre les microservices.

- Configurer le fichier application.yml comme suit :

```
spring:
 cloud:
 gateway:
 mvc:
 routes:
 - id: r1
 uri: http://localhost:8088/
 predicates:
 - Path=/clients/**
```



En cas de problème sur la route, changer la dépendance vers :

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

Cette configuration indique au micro-service Gateway de router les requêtes HTTP ayant l'URL suivante : `http://localhost:8888/clients` vers le micro-service `http://localhost:8088/clients`.

- Ouvrir le navigateur et saisir `http://localhost:8888/clients`. La page web qui liste les client doit s'afficher.
  - Inclure une seconde voie d'accès vers `/client/*`.
  - Ouvrir le navigateur et saisir l'URL suivante : `http://localhost:8888/client/`
1. La page web affichant les détails du client avec l'identifiant 1 devrait apparaître.

On constate que le Micro-service Gateway fonctionne correctement !!

Il est également possible de configurer cela avec du code Java. Nous souhaitons en outre ajouter une option permettant d'appeler le service en question par son nom d'hôte dans l'URL plutôt que par son adresse IP.

Pour ce faire, suivez ces étapes :

- i. Avant de commencer, il faut d'abord reconfigurer les micro-services client et Gateway pour leur autoriser de s'auto-enregistrer sur le service Discovery Eureka.
- ii. Désactiver la configuration statique de la Gateway en renommant le fichier `application.yml` à `app.yml`.
- iii. Ajouter la ligne `eureka.instance.hostname=localhost` sur le fichier `application.properties`.

```
server.port=8888
spring.application.name=Gateway
spring.cloud.discovery.enabled=true
eureka.instance.hostname=localhost
```

- iv. Ouvrir la main classe de la Gateway et ajouter le Bean suivant :

---

```
@Bean
RouteLocator routes(RouteLocatorBuilder builder) {
 return builder.routes()
 .route(r -> r.path("/clients/**").uri("lb://SERVICE-CLIENT"))
 .build();
}
```

---

- v. Exécuter tous les microservices
- vi. Ouvrir le navigateur et taper : `http://localhost:8888/clients`. La liste des clients doit apparaître.

(b) Configuration dynamique :

Pour la configuration dynamique, c'est plus simple. On conserve la même configuration que précédemment. Il suffit simplement de commenter ou de supprimer le bean précédent, puis de le remplacer par un nouveau bean comme suit :

---

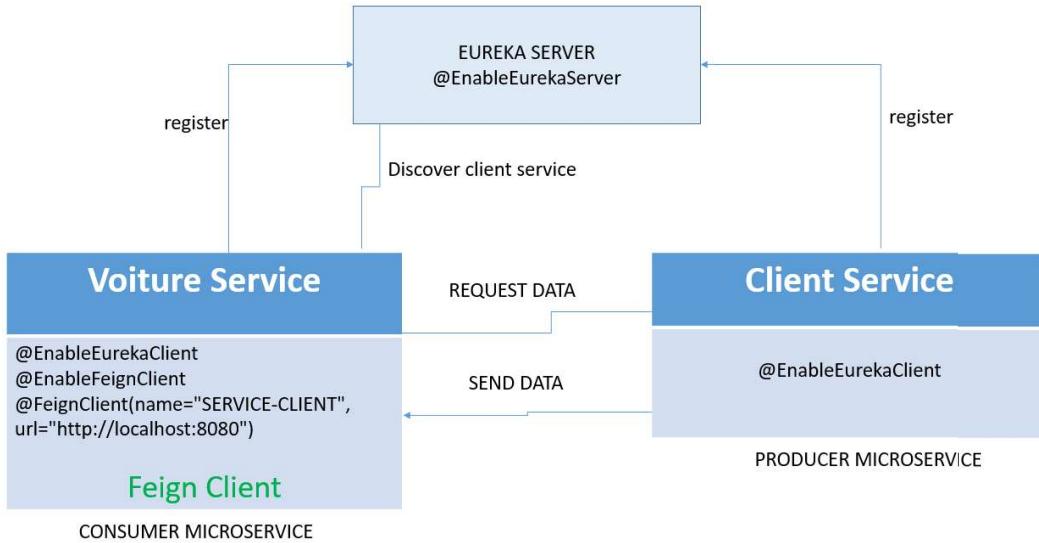
```
@Bean
DiscoveryClientRouteDefinitionLocator
 routesDynamique(ReactiveDiscoveryClient rdc,
 DiscoveryLocatorProperties dlp){
 return new DiscoveryClientRouteDefinitionLocator(rdc, dlp);
}
```

---

Dans la configuration dynamique pour accéder au service souhaité, il suffit de taper son nom dans l'URL (par exemple : `http://localhost:8888/SERVICE-CLIENT/clients`).

## 2ème micorservice

### 1. Architecture de l'application



Maintenant, nous allons ajouter un autre service et connecter les deux micro-services à la base de données H2. Les deux micro-services doivent communiquer pour maintenir la cohérence des données. Voici le diagramme de classe de notre application :



Transformer le diagramme de classe en code Java en veillant à respecter les règles de transformation, en particulier la conversion de l'association bidirectionnelle.

2. Créer un projet pour réaliser le M.S service-voiture en respectant les mêmes étapes de la création du M.S service-client lors du TP précédent.
3. Aller sur Maven Repository pour récupérer les dépendances liées à :
  - OPENFEIGN
  - hateoas
4. Ajouter ces dépendances au projet.
5. Une fois la classe Voiture est créée, il faut créer la classe Client dans le package de l'application Voiture comme ceci :

---

```

@Data
@AllArgsConstructor
@NoArgsConstructor
public class Client {
 private Long id;
 private String nom;
 private Float age;
}

```

---

6. Dans la classe Voiture, ajoutez l'annotation `@Transient` avant l'attribut `client` de la classe Client. Ceci vise à indiquer à Spring Boot que ce champ ne doit pas être persisté. L'annotation `@ManyToOne` signale qu'il s'agit d'une association plusieurs à un. Ainsi, la classe Voiture de la couche modèle devrait avoir l'aspect suivant :

---

```

@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Voiture {

 @Id
 @GeneratedValue
 private Long id;
 private String marque;
 private String matricule;
 private String model;
 private Long id_client;
 @Transient
 @ManyToOne
 private Client client;

}

```

---

7. Ajouter la configuration suivante :

```

server.port= 8089
spring.application.name=SERVICE-VOITURE
spring.cloud.discovery.enabled=true
eureka.instance.hostname=localhost

```

8. Créer un Bean avec des `CommandLineRunner` pour insérer des voitures dans la base de données H2.
9. Tester sur un navigateur que tout se déroule correctement.
10. Pour permettre à ce Micro-service de se connecter au micro-service service-client, créez une interface en-dessous de la classe VoitureApplication que vous nommerez `ClientService`. Cette interface doit être précédée de l'annotation `@FeignClient(name="service-client")`. Cette annotation indique que notre classe peut se connecter via le protocole REST au micro-service service-client.

---

```

@FeignClient(name="SERVICE-CLIENT")
public interface ClientService{
 @GetMapping(path="/clients/{id}")
 public Client clientById(@PathVariable Long id);
}

```

---

11. Créer la méthode `clientById` comme suit : Cette méthode reçoit en paramètre l'id du client récupéré en URI et renvoie l'objet client obtenu du Micro-service service-client.
12. Pour pouvoir récupérer l'id du Micro-service nommé service-client, ajoutez la configuration suivante à son fichier `properties` : `spring.cloud.discovery.enabled=true`. Ceci permet d'exposer les ID des enregistrements clients de la base de données H2.
13. Afin de tester votre Micro-service nommé service-voiture, modifiez le Bean comme suit :

---

```

@Bean

```

```

CommandLineRunner initialiserBaseH2(VoitureRepository voitureRepository,
ClientService clientService){

 return args -> {
 Client c1 = clientService.clientById(2L);
 Client c2 = clientService.clientById(1L);
 System.out.println("*****");
 System.out.println("Id est :" + c2.getId());
 System.out.println("Nom est :" + c2.getNom());
 System.out.println("*****");
 System.out.println("*****");
 System.out.println("Id est :" + c1.getId());
 System.out.println("Nom est :" + c1.getNom());
 System.out.println("Age est :" + c1.getAge());
 System.out.println("*****");
 voitureRepository.save(new Voiture(Long.parseLong("1"), "Toyota",
 "A 25 333", "Corolla", 1L, c2));
 voitureRepository.save(new Voiture(Long.parseLong("2"), "Renault",
 "B 6 3456", "Megane", 1L, c2));
 voitureRepository.save(new Voiture(Long.parseLong("3"), "Peugeot",
 "A 55 4444", "301", 2L, c1));
 };
}

```

- 
14. Exécuter tous les micro-service : Eureka , Client et voiture
  15. Lancer le navigateur et tapez : <http://localhost:8089/voitures> :



**Contrôleur Voiture**

```
@RestController
public class VoitureController {

 @Autowired
 VoitureRepository voitureRepository;

 @Autowired
 VoitureService voitureService;

 @Autowired
 VoitureApplication.ClientService clientService ;

 @GetMapping(value = "/voitures", produces = {"application/json"})
 public ResponseEntity<Object> findAll() {
 try {
 List<Voiture> voitures = voitureRepository.findAll();
 return ResponseEntity.ok(voitures);
 } catch (Exception e) {
 return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
 .body("Error fetching voitures: " + e.getMessage());
 }
 }

 @GetMapping("/voitures/{id}")
 public ResponseEntity<Object> findById(@PathVariable Long id) {
 try {
 Voiture voiture = voitureRepository.findById(id)
 .orElseThrow(() -> new Exception("Voiture Introuvable"));

 // Fetch the client details using the clientService
 voiture.setClient(clientService.clientById(voiture.getClientId()));

 return ResponseEntity.ok(voiture);
 } catch (Exception e) {
 return ResponseEntity.status(HttpStatus.NOT_FOUND)
 .body("Voiture not found with ID: " + id);
 }
 }

 @GetMapping("/voitures/client/{id}")
 public ResponseEntity<List<Voiture>> findByClient(@PathVariable Long id) {
 try {
 Client client = clientService.clientById(id);
 if (client != null) {
 List<Voiture> voitures = voitureRepository.findByClientId(id);
 return ResponseEntity.ok(voitures);
 } else {
 return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
 }
 } catch (Exception e) {
 return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
 }
 }
}
```

```

 @PostMapping("/voitures/{clientId}")
 public ResponseEntity<Object> save(@PathVariable Long clientId, @RequestBody
 Voiture voiture) {
 try {
 // Fetch the client details using the clientService
 Client client = clientService.clientById(clientId);

 if (client != null) {
 // Set the fetched client in the voiture object
 voiture.setClient(client);

 // Save the Voiture with the associated Client
 voiture.setClientId(clientId);
 voiture.setClient(client);
 Voiture savedVoiture = voitureService.enregistrerVoiture(voiture);

 return ResponseEntity.ok(savedVoiture);
 } else {
 return ResponseEntity.status(HttpStatus.NOT_FOUND)
 .body("Client not found with ID: " + clientId);
 }
 } catch (Exception e) {
 return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
 .body("Error saving voiture: " + e.getMessage());
 }
 }

 @PutMapping("/voitures/{id}")
 public ResponseEntity<Object> update(@PathVariable Long id, @RequestBody
 Voiture updatedVoiture) {
 try {
 Voiture existingVoiture = voitureRepository.findById(id)
 .orElseThrow(() -> new Exception("Voiture not found with ID: " + id));

 // Update only the non-null fields from the request body
 if (updatedVoiture.getMatricule() != null &&
 !updatedVoiture.getMatricule().isEmpty()) {
 existingVoiture.setMatricule(updatedVoiture.getMatricule());
 }

 if (updatedVoiture.getMarque() != null &&
 !updatedVoiture.getMarque().isEmpty()) {
 existingVoiture.setMarque(updatedVoiture.getMarque());
 }

 if (updatedVoiture.getModel() != null &&
 !updatedVoiture.getModel().isEmpty()) {
 existingVoiture.setModel(updatedVoiture.getModel());
 }

 // Save the updated Voiture
 Voiture savedVoiture = voitureRepository.save(existingVoiture);

 return ResponseEntity.ok(savedVoiture);
 } catch (Exception e) {

```

```
 return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
 .body("Error updating voiture: " + e.getMessage());
 }
}
```

---