**TP 1: Spring Cloud et Eureka Discovery**

## 1.1. Introduction à Spring Cloud et Eureka

Spring Cloud est un framework open-source développé par l’équipe Spring de Pivotal. Il simplifie le développement d’applications distribuées en fournissant des outils pour la création de systèmes robustes et évolutifs basés sur l’architecture de microservices. Eureka, un composant clé de Spring Cloud, est un service de découverte de services permettant aux microservices de s’enregistrer et de découvrir d’autres services dans le réseau.

## 1.2. Caractéristiques Principales de Spring Cloud

### Coordination des Microservices

Spring Cloud offre des mécanismes de coordination et de gestion des microservices, facilitant la découverte de services, la gestion de la configuration, et la résilience des applications distribuées.

### Découverte de Services avec Eureka

Eureka simplifie la découverte de services en permettant à chaque microservice de s’enregistrer auprès du registre Eureka. Les services clients peuvent ensuite interroger Eureka pour découvrir les instances disponibles d’un service donné.

### Gestion de Configuration

Avec Spring Cloud Config, la configuration des applications peut être externalisée et gérée de manière centralisée. Cela facilite la modification dynamique des configurations sans nécessiter le redéploiement des microservices.

### 

### Résilience et Tolérance aux Pannes

Spring Cloud intègre des mécanismes tels que Hystrix pour la gestion de la résilience et la tolérance aux pannes. Cela permet aux microservices de mieux gérer les défaillances et de maintenir une disponibilité élevée.

### API Gateway

Spring Cloud Gateway offre une solution robuste pour créer des API Gateway, facilitant ainsi la gestion des requêtes entrantes, l’acheminement vers les services appropriés et l’application de filtres.

### 

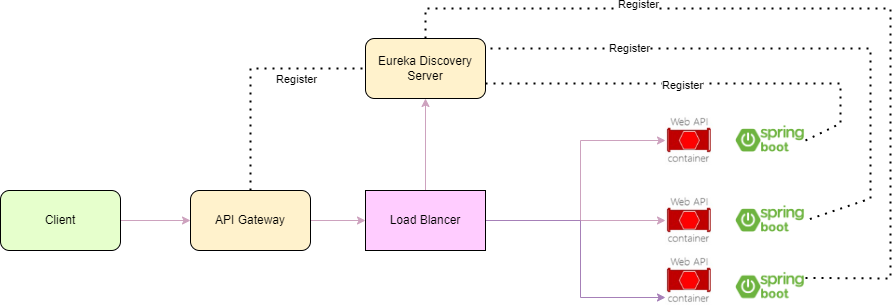
### Intégration avec Spring Boot

Spring Cloud s’intègre parfaitement avec Spring Boot, le rendant facile à utiliser dans des pro- jets basés sur Spring. Les microservices peuvent être développés rapidement en utilisant les fonctionnalités conventionnelles de Spring Boot.

## 

## 1.3. Architecture de Microservices avec Spring Cloud et Eureka

L’architecture de microservices avec Spring Cloud et Eureka repose sur des principes clés tels que la distribution des responsabilités, la gestion autonome des services, et la communication à travers des protocoles légers.



### Coordination des Services avec Eureka

* + - * **Découverte de Services** : Eureka permet aux services de s’enregistrer et de découvrir d’autres services.
      * **Équilibrage de Charge** : Eureka facilite l’équilibrage de charge en fournissant des informa- tions sur les instances disponibles d’un service.
      * **Résilience** : En cas de défaillance d’une instance, Eureka met à jour dynamiquement le registre pour refléter l’état actuel des services.

### Communication entre Microservices

* + - * **API Gateway** : Spring Cloud Gateway gère l’acheminement des requêtes entrantes.
      * **Protocole Léger** : Utilisation de protocoles légers tels que HTTP/JSON pour la communica- tion inter-microservices.

## Avantages de Spring Cloud avec Eureka

* + - **Simplicité de Développement** : Spring Cloud avec Eureka simplifie le développement d’applications distribuées en fournissant des abstractions et des outils prêts à l’emploi.
    - **Scalabilité et Évolutivité** : L’architecture de microservices basée sur Spring Cloud et Eureka permet de construire des systèmes évolutifs et hautement performants.
    - **Gestion de la Complexité** : Spring Cloud avec Eureka offre des solutions pour gérer la complexité des architectures de microservices, y compris la découverte de services et la gestion de la configuration.
    - **Intégration Transparente** : S’intègre facilement avec l’écosystème Spring, y compris Spring Boot, pour une expérience de développement homogène.

## 1.4. Conclusion

En conclusion, Spring Cloud avec Eureka offre une solution complète et puissante pour le développe- ment d’architectures de microservices. En facilitant la coordination avec Eureka, la découverte de services, la gestion de configuration, la résilience, et la création d’API Gateway avec Spring Cloud, les développeurs peuvent créer des applications distribuées robustes, évolutives et faciles à maintenir. L’utilisation de Eureka simplifie la coordination des services, permettant ainsi aux microservices de s’adapter dynamiquement aux changements dans leur environnement.

## Travaux pratiques

**Architecture Micro-services avec FeignClient Objectif**

Ce TP vise à développer une compréhension approfondie de l’architecture micro-service. Les axes centraux de cet apprentissage englobent la création et l’enregistrement de micro-services, la connexion à une base de données In-memory H2, l’établissement d’un micro-service Gateway, et l’implémentation d’une communication synchrone entre les micro-services en utilisant l’outil OPENFEIGN.

Dans ce TP, nous adopterons une architecture basée sur les microservices, caractérisée par la décomposition d’une application en de petits services indépendants. Au cœur de cette structure se situent les microservices clients, des entités autonomes qui interagissent pour fournir une fonctionnalité complète. L’API Gateway agit en tant que point d’entrée centralisé, simplifiant la gestion des requêtes en dirigeant le trafic vers les microservices appropriés. Le serveur de découverte Eureka revêt un rôle crucial en permettant à chaque microservice de s’enregistrer de manière dynamique, formant ainsi un annuaire décentralisé des services disponibles.

* + - 1. **Création du service discovery Eureka**

Pour créer un service discovery, on doit procéder de la manière suivante :

* + - * 1. Créer un nouveau projet sur Spring Initializr nommé Eureka Server.
        2. Ajoutez la dépendance suivante et cliquer sur Generate :



* + - * 1. 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

* + - * 1. Dans la classe EurekaServerApplication ajouter l’annotation @EnableEurekaServer :

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@EnableEurekaServer

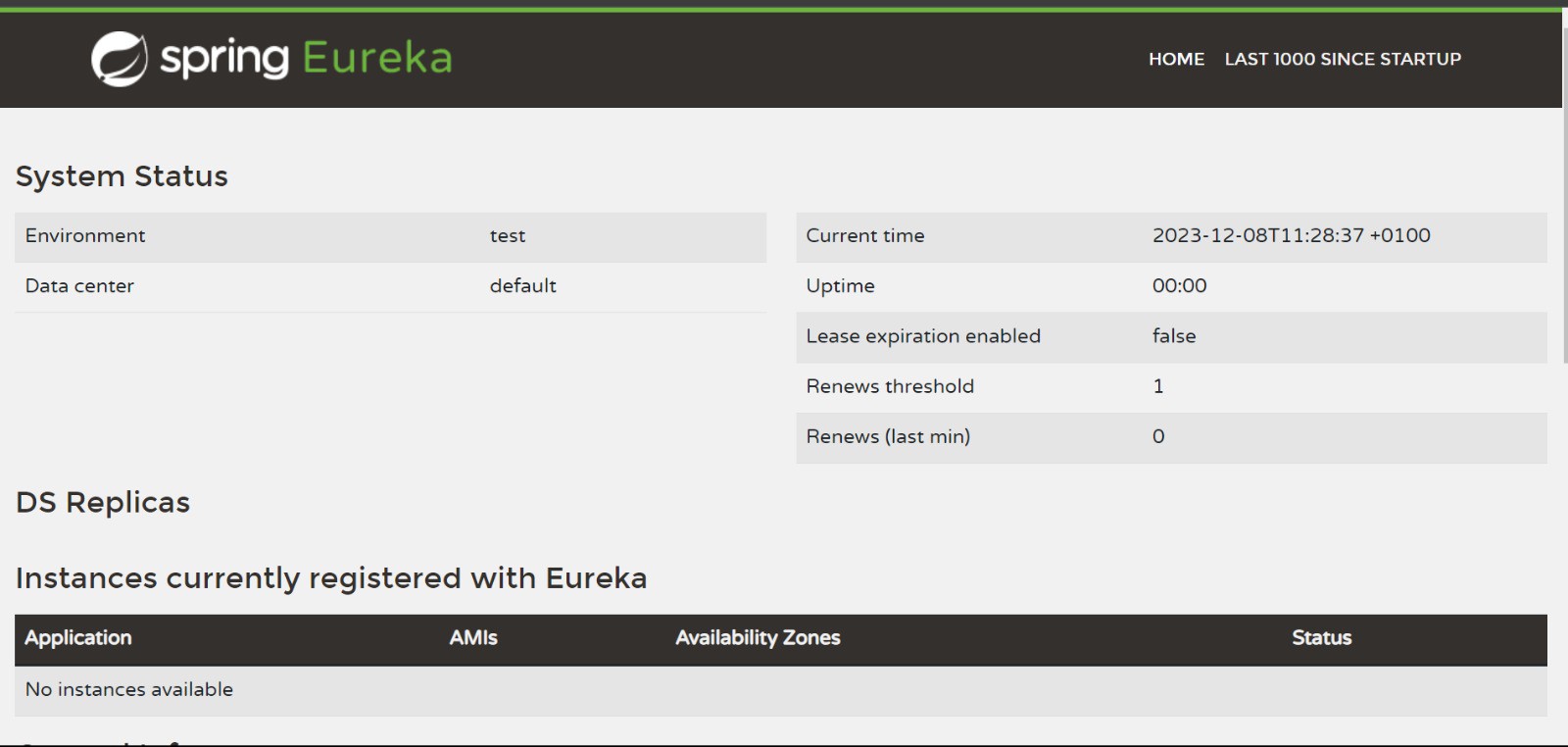
@SpringBootApplication

public class EurekaServerApplication {

...

}

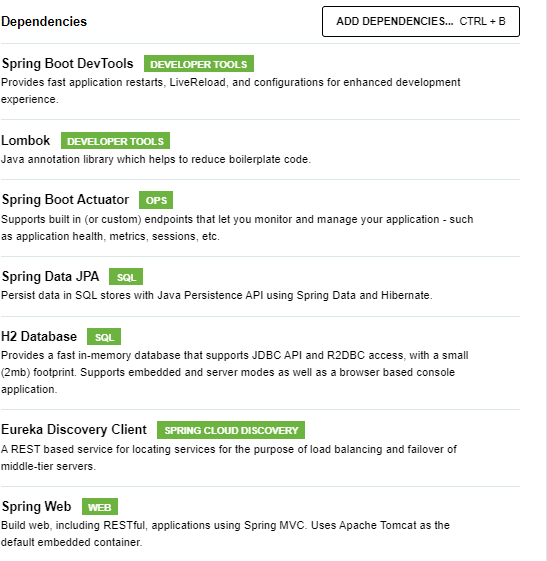
* + - * 1. Exécuter le projet et Lancer le navigateur et taper l’URL suivant : http://localhost:8761/ Une page web s’affichera comme suit :



Félicitation vous avez créé votre premier Service Discovery !!

* + - 1. **Création du service Client (1er microservice)**

Pour créer un service client, on doit procéder de la manière suivante :

* + - * 1. Créer un nouveau projet sur Spring Initializr nommé Client.
        2. Ajoutez les dépendances suivantes et cliquer sur Generate :
    - **Spring Boot Actuator** Prend en charge les points de terminaison intégrés (ou person- nalisés) qui vous permettent de surveiller et de gérer votre application - comme la santé de l’application, les mesures, les sessions, etc;
    - **Eureka Discovry Client** il se base sur REST pour localiser des services dans le but d’équilibrer la charge et le basculement des serveurs intermédiaires;
    - **H2** Fournit une base de données rapide en mémoire qui prend en charge l’API JDBC, avec un faible encombrement (2 mb);
    - **Spring Data JPA** Persistance des données SQL avec l’API qui permet aux développeurs d’organiser des données relationnelles dans des applications utilisant la plateforme Java en se basant sur Spring Data et Hibernate;
    - **Spring Web** pour créer des applications web en utilisant Spring MVC. Il utilise Apache Tomcat comme conteneur intégré par défaut;
    - **Spring Boot Devtools** Offre des redémarrages rapides des applications, LiveReload, et des configurations pour une expérience de développement améliorée;
    - **Lombok** Bibliothèque d’annotation Java qui permet de réduire le code passe-partout.
      * 1. Dans la classe ClientApplication , ajouter l’annotation @EnableDiscoveryClient

suivante :

@EnableDiscoveryClient

@SpringBootApplication

public class ClientApplication {

public static void main(String[] args) {

SpringApplication.run(ClientApplication.class, args);

}

}

Cette annotation permet de rendre Eureka discovry service active.

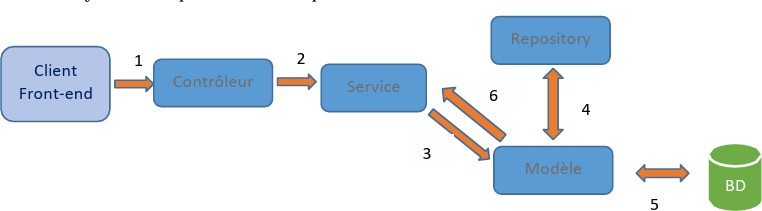
* + - * 1. Cliquer sur src/main/ressources et ajouter les trois lignes suivantes :

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

* + - * 1. Architecture de microservice :

Pour créer un microservice sur Spring Boot capable de se connecter à une base de données IN-memory H2, il est préférable de respecter l’architecture multi-couches suivante : Client

:



Dans cette architecture, la couche contrôleur analyse d’abord le type de la requête HTTP, puis appelle la méthode correspondante de la classe de service. Cette méthode, à son tour, fait appel à la classe modèle pour communiquer avec la base de données. Ceci est réalisé grâce à une interface qui hérite de l’interface JPARepository de la couche repository. Cette interface est implémentée par la classe modèle. Une fois que la classe modèle a récupéré les données, elle les envoie à la classe service pour les exploiter.

Pour implémenter cette architecture, il est nécessaire de suivre les étapes suivantes :

Sélectionner le package principal et créer les sous-packages suivants : entities, con- trollers, repositories et services.

Dans le package entities :

Créer la classe Client avec les attributs (id Long, nom String, prenom String, age Float) dans le package Model. Cette classe, de type entité (Entity), représente la couche de persistance. C’est pour cette raison qu’il faut ajouter au-dessus de la classe l’annotation JPA @Entity;

Ajouter les annotations Lombok : @Data, @AllArgsConstructor et @NoArgsConstructor pour générer les setters, getters, ainsi que les constructeurs avec et sans arguments;

Ajouter au-dessus de l’attribut Id l’annotation JPA @Id afin d’indiquer à Spring Boot que ce champ est une clé;

Ajouter au-dessus de l’attribut Id l’annotation JPA @GeneratedValue afin d’indiquer à Spring Boot que la valeur de ce champ est générée automatiquement.

@Entity

@Data

@AllArgsConstructor

@NoArgsConstructor

public class Client {

// annotation pour dire que l’attribut Id est une clé de la classe Client @Id

// pour générer les valeurs d’Id automatiquement

@Id

@GeneratedValue

private Long id;

private String nom;

private Float age;

}

Dans le package repositories :

Créer une interface ClientRepository;

Faire hériter cette interface de l’interface JpaRepository. Qui est de type Client. Le type de la clé est Long;

Ajouter au-dessus de la classe l’annotation @Repository pour indiquer que c’est un repository.

@Repository

public interface ClientRepository extends JpaRepository<Client,Long> {

}

Dans le package controllers

Créer une classe **ClientController** avec au-dessus l’annotation **@RestController**

pour indiquer que c’est un contrôleur;

Créer un attribut **clientRepository** de type **ClientRepository**. Ensuite, ajouter l’annotation **@Autowired** au-dessus de l’attribut clientRepository.

Afin de tester l’accès à la base de données avec succès, il est nécessaire de créer des méthodes dans lesquelles on fait appel directement à la couche Repository sans passer par la couche de présentation. Pour cela, les étapes suivantes sont requises :

Créer la méthode **List findAll()** avec l’annotation **@GetMapping("/clients")** au-dessus. Dans cette méthode, on fait appel à la méthode **findAll()** de l’attribut **clientRepository** déjà implémentée par Spring Boot. Cette méthode renvoie la liste des objets **ClientRepository** dans la base de données.

Créer la méthode **List findById(Long id)** avec l’annotation **@GetMap- ping("/client/id")** au-dessus. Dans cette méthode, on fait appel à la méthode **findById(id)**. Cette méthode reçoit en paramètres un **id** de type **Long** et renvoie un objet **ClientRepository** avec le même **id** depuis la base de données.

@RestController

public class ClientController {

@Autowired

ClientRepository clientRepository ;

@GetMapping("/clients")

public List findAll(){

return clientRepository.findAll();

}

@GetMapping("/client/{id}")

public Client findById (@PathVariable Long id) throws Exception{

return this.clientRepository.findById(id).orElseThrow(()-> new Exception("Client inexistant"));

}

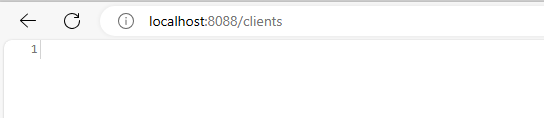
}

Notez qu’il faut ajouter l’annotation @PathVariable pour indiquer que le paramètre

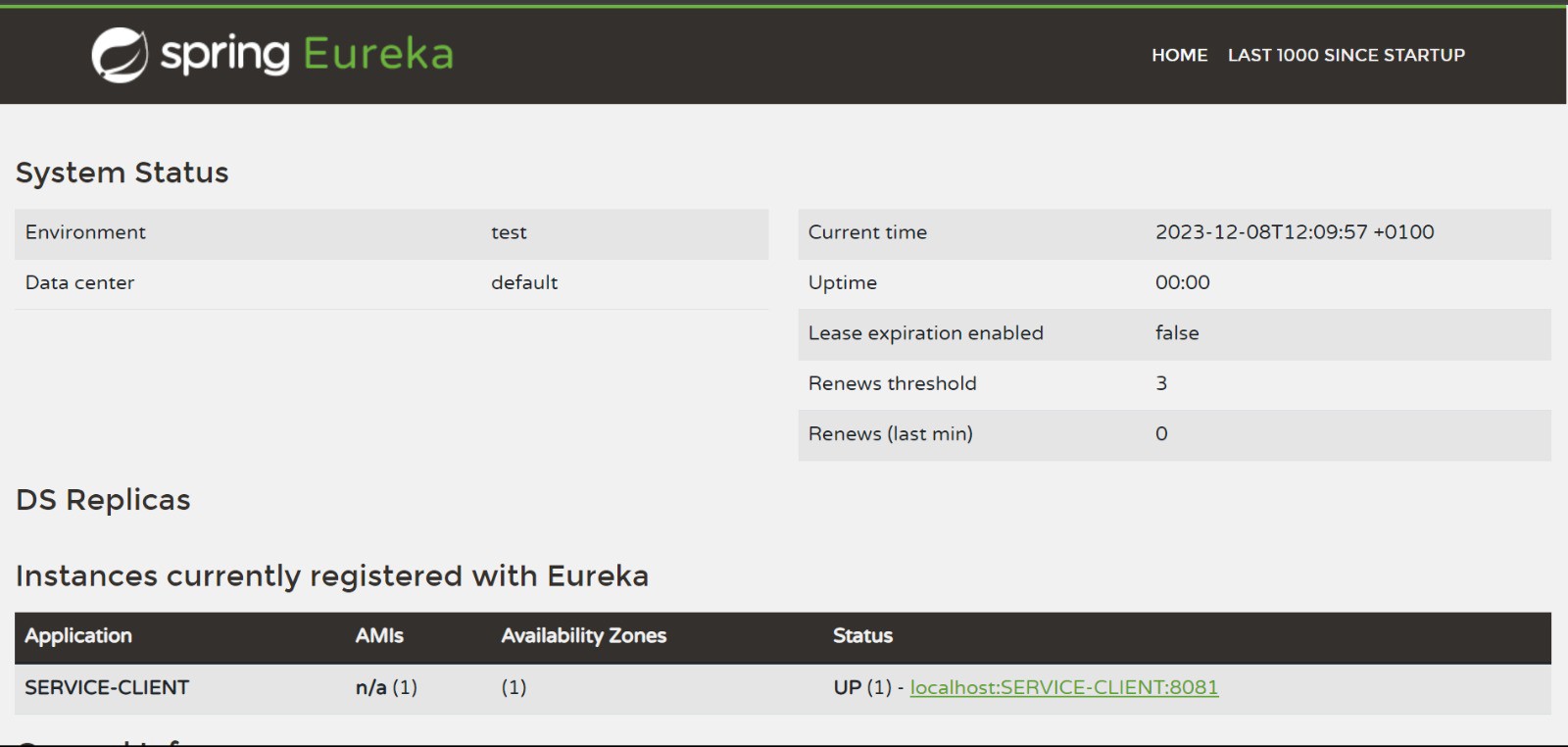
id de la fonction findById est le même id récupéré depuis l’URL ("/client/id"). L’utilisation de la fonction orElseThrow(() -> new Exception("Client inexistant")) a pour objectif de lever une exception si la méthode findById(id) n’arrive pas à trou-

ver dans la base l’objet correspondant.

* + - * 1. Aller sur le navigateur est taper http://localhost:8088/clients :



Ceci indique que votre service fonctionne correctement. Il faut maintenant vérifier s’il est bien enregistré par le service Eureka. Pour cela, tapez l’URL : http://localhost:8761/ et actualisez le navigateur. Vous devriez alors obtenir :



On voit bien que le service « SERVICE-CLIENT» est bien enregistré.

* + - * 1. Pour enregistrer des clients dans la base de données, il est nécessaire d’utiliser des commandes CommandLineRunner, qui sont des commandes s’exécutant au lancement du projet. Ces commandes doivent être intégrées aux beans dans la fonction principale du programme.
* Créer une fonction nommée initialiserBaseH2() dans ClientRepository qui reçoit en paramètres un objet ClientRepository et qui retourne CommandLineRunner.
  + - Ajouter au-dessus de cette méthode l’annotation @Bean.
    - Utiliser le paramètre args qui est un tableau de String de la fonction main(String[] args) pour sauvegarder des objets client dans la base H2 via la méthode save(Client client) du clientRepository. Ce dernier objet est passé en argument à la fonction initialiserBaseH2().

@Bean

CommandLineRunner initialiserBaseH2(ClientRepository clientRepository) {

return args -> {

clientRepository.save(new Client(Long.parseLong("1"), "Rabab SELIMANI", Float.parseFloat("23")));

clientRepository.save(new Client(Long.parseLong("2"), "Amal RAMI", Float.parseFloat("22")));

clientRepository.save(new Client(Long.parseLong("3"), "Samir SAFI", Float.parseFloat("22")));

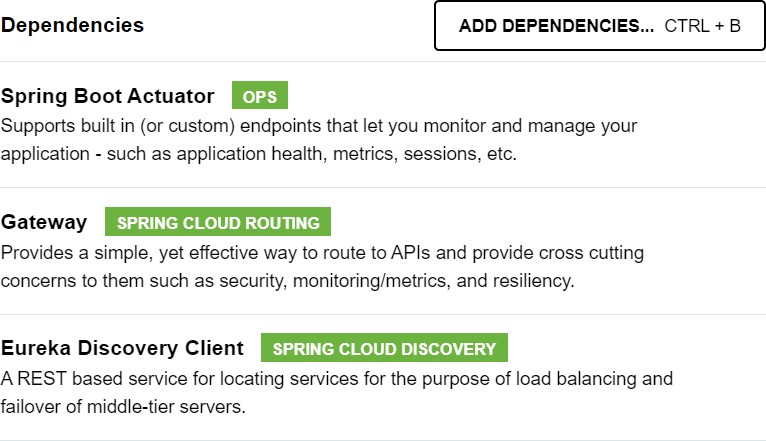
};

}

* + - * 1. Aller sur le navigateur et taper http://localhost:8088/clients. Constater que les données sauvegardées dans la base H2 sont bien récupérées.
        2. Aller sur le navigateur est taper http://localhost:8088/client/1. Vérifier que le contrôleur parvient à orienter la requête HTTP GET vers la méthode appropriée et que les données sont bien récupérées.
      1. **Création d’un service Gateway**

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

* + - * 1. Créer un nouveau projet sur Spring Initializr nommé GateWay.
        2. Ajoutez les dépendances suivantes et cliquer sur Generate :



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

* Statique via des fichiers yaml et proprieties ou bien via du code Java
* Dynamique avec du code Java seulement

Configuration statique :

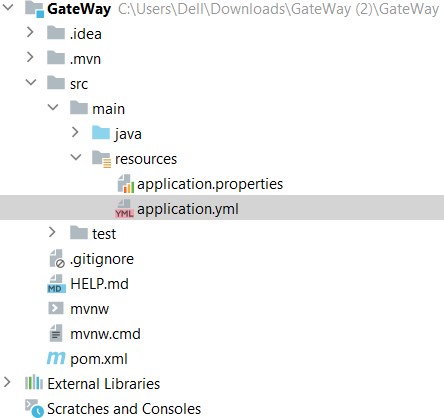
* Ouvrir le fichier application.properties et ajouter les propriétés suivantes :

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/\*\*

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 :

1. 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.
2. Désactiver la configuration statique de la Gateway en renommant le fichier application.yml à app.yml.
3. 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

1. Ouvrir la main classe de la Gateway et ajoutez le Bean suivant :

@Bean

RouteLocator routes(RouteLocatorBuilder builder) {

return builder.routes()

.route(r ->

r.path("/clients/\*\*").uri("lb://SERVICE-CLIENT"))

.build();

}

1. Exécuter tous les microservices
2. Ouvrir le navigateur et taper : http://localhost:8888/clients. La liste des clients doit apparaître.

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);

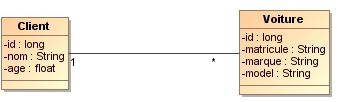
}

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.

1. 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.
2. Aller sur Maven Repository pour récupérer les dépendances liées à :
   * OPENFEIGN
   * hateoas
3. Ajouter ces dépendances au projet.
4. 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;

}

1. 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;

}

1. Ajouter la configuration suivante :

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

1. Créer un Bean avec des CommandLines Runner pour insérer des voitures dans la base de données H2.
2. Tester sur un navigateur que tout se déroule correctement.
3. 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);

}

1. 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.
2. Pour pouvoir récupérer l’id du Micro-service nommé service-client, ajoutez la config- uration 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.
3. 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("Nom 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));

};

}

1. Exécuter tous les micro-service : Eureka , Client et voiture
2. Lancer le navigateur et tapez : http://localhost:8089/voitures :



**Controlleur 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()); | | | |
|  |  | } | | | |
|  | } |  | | | |
|  |  |  | | | |
|  |  |  | | | |
|  |  |  | | | |
| } |  |  | | | |