

Département Télécommunications, Réseaux & Informatique

Ecole Nationale des Sciences Appliquées d’El Jadida

Université Chouaib Doukkali

Ingénierie Informatique et Technologies Emergentes (IITE)

**3eme année Cycle Ingénieur**



**Réalisé par** 

Rabab FAHSSI

Houda EL KORAINI

Saad BAKANZIZE

**Encadré par**

M.LACHGAR

Année Universitaire 2023/2024

**Table de matière**

[I-Création du service discovery Eureka 3](#_heading=h.gjdgxs)

[II-Création du service Client 4](#_heading=h.30j0zll)

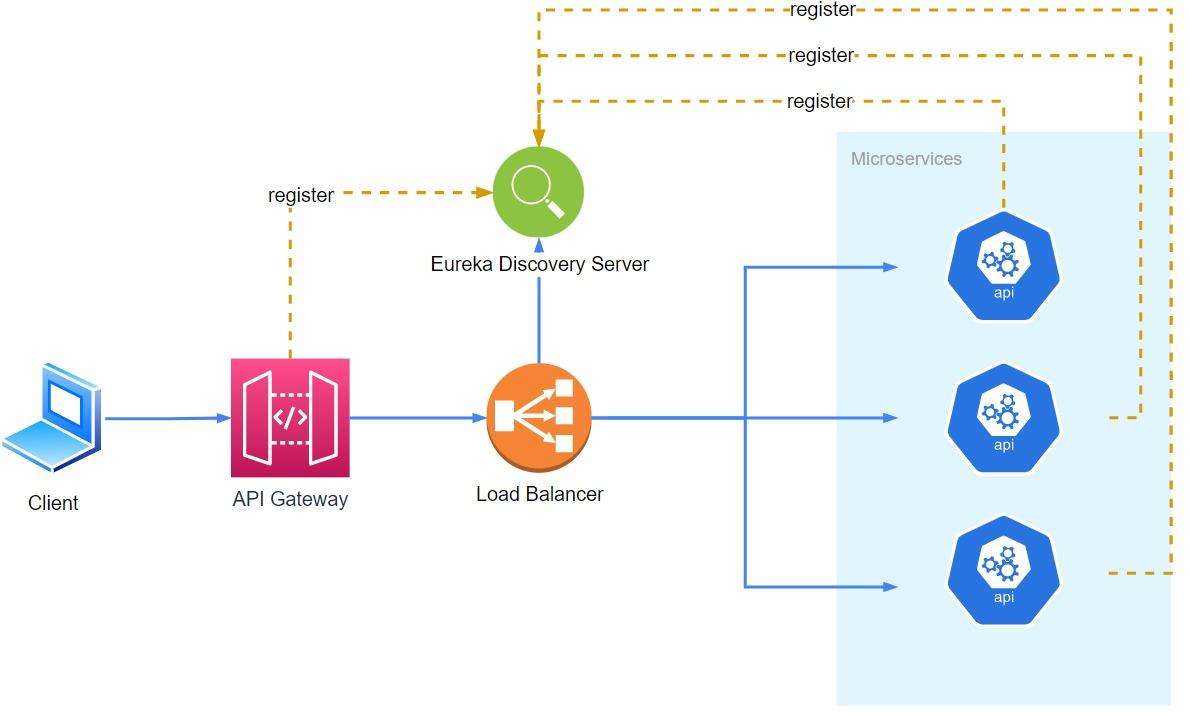
[III-Création d’un service Gateway 11](#_heading=h.1fob9te)

[IV- Architecture de l’application 16](#_heading=h.3znysh7)

# 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

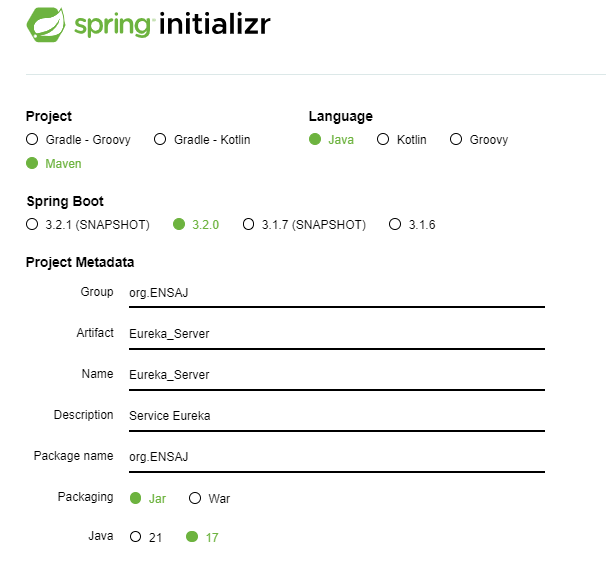
Dans ce TP, nous adopterons une architecture basée sur les microservices, caractérisée par la décomposition d'une application en de petits services indépendants. Au cœur de cette structure se situent les microservices clients, des entités autonomes qui interagissent pour fournir une fonctionnalité complète. 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.



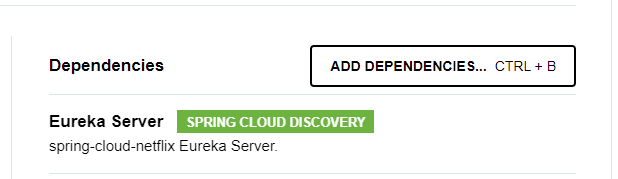
# I-Création du service discovery Eureka

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

1- Créer un nouveau projet sur Spring Initializr :



Ajoutez la dépendance suivante et cliquez sur Generate :



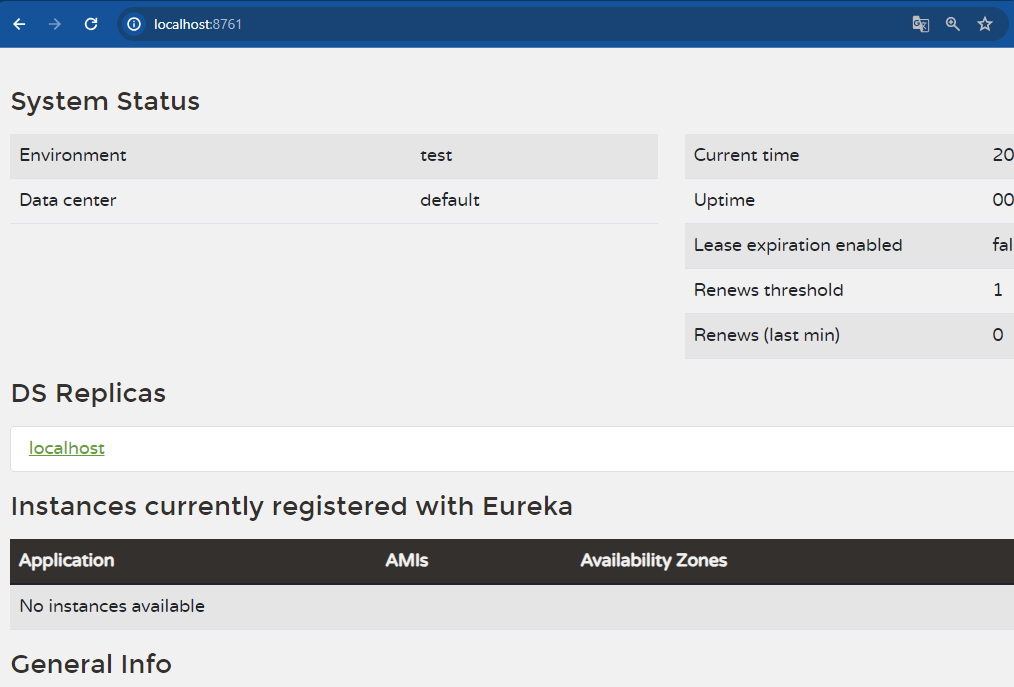
Cliquez sur src/main/ressources et ajoutez les trois lignes suivantes :

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

Cliquez sur src/main/ressources et ajoutez l’annotation @EnableEurekaServer comme suit :

package org.ENSAJ;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;  
  
@EnableEurekaServer  
@SpringBootApplication  
public class EurekaServerApplication {

Exécuter le projet et Lancez votre navigateur et tapez l’URL suivant : http://localhost:8761/ Une page web s’affichera comme suit :

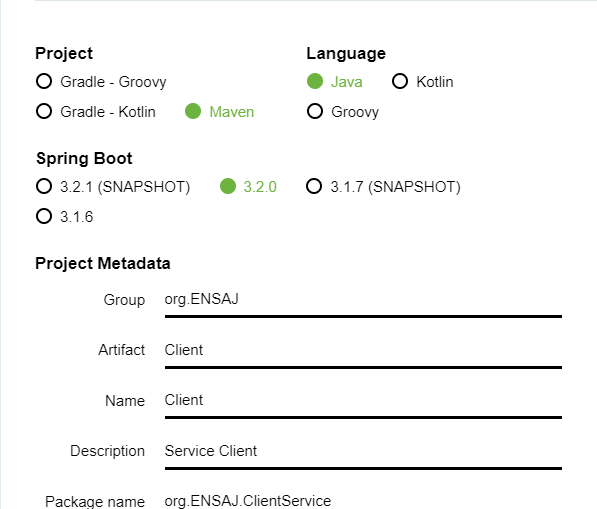


Félicitation vous avez créé votre premier Service Discovry !!

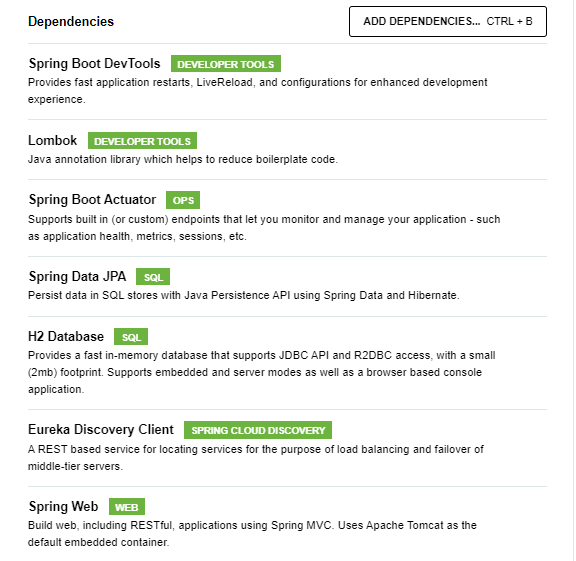
# II-Création du service Client

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

1. Créer un nouveau projet sur Spring Initializr :



Ajoutez les dépendances suivantes et cliquez sur Generate :



**Spring Boot Actuator** Prend en charge les points de terminaison intégrés (ou personnalisés) 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.

**Rest Repositories** Expose les JPA repositorie sur REST via Spring Data REST.

**Lombok** Bibliothèque d'annotation Java qui permet de réduire le code passe-partout.

4. Cliquez sur src/main/ressources et ajoutez l’annotation suivante : @EnableEurekaClient

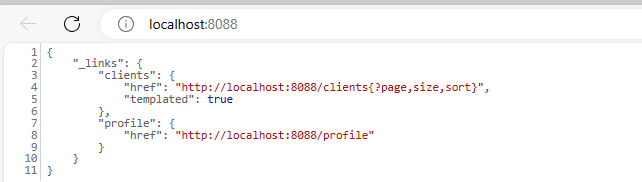
@EnableDiscoveryClient  
@SpringBootApplication  
public class ClientApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(ClientApplication.class, args);  
 }

Cette annotation permet de rendre Eureka discovry service active

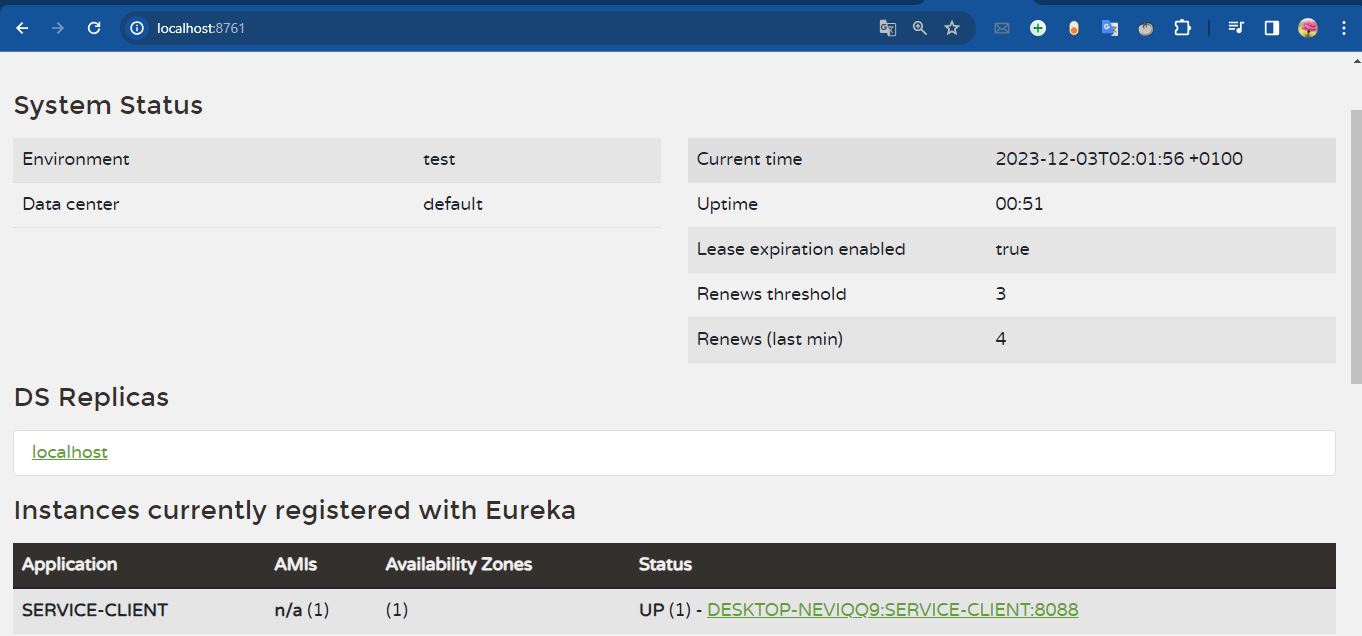
5. Cliquez sur src/main/ressources et ajoutez les trois lignes suivantes :

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

7. Lancez votre navigateur et tapez l’URL suivant : http://localhost:8088/ Une page web s’affichera comme suit :



Ceci indique que votre service marche bien. Il faut vérifier maintenant s’il est bien enregistré par le service Eurika. Pour cela il faut taper l’URL : http://localhost:8761/ est actualiser le navigateur. On obtient alors :

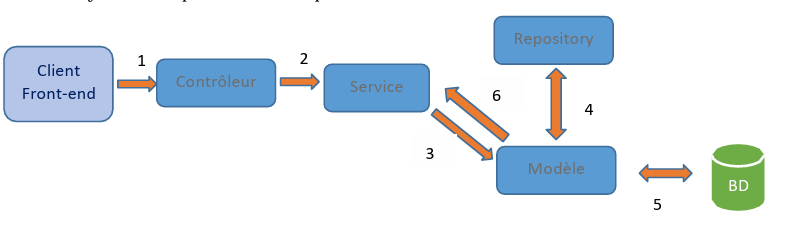


On voit bien que le service « SERVICE-CLIENT» est bien enregistré.

Connecter un micro-service à une base de données In-memory H2.

1. Connection du M.S client à une base de données H2

Pour créer une micro-service 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 :



Dans cette architecture, la couche contrôleur, d’abord analyse le type de la requête http et appelle ensuite la méthode de la classe service correspondante. Cette méthode fait appel à son tour à la classe modèle afin de communiquer avec la BD. Ceci est réalisé grâce à une interface qui va hériter de l’interface JPARepository de la couche repository. Cette interface est implémentée par la classe modèle. Une fois la classe modèle récupère les données, elle les envoie à la classe service afin d’être exploiter.

Pour implémenter cette architecture, on doit suivre les étapes qui suivent :

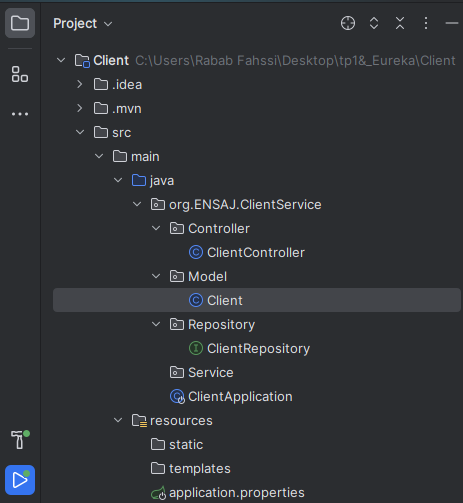
1. Sélectionner le package org.ENSAJ.ClientService et créer les sous-packages suivants :

a) org.ENSAJ.ClientService.Model

b) org.ENSAJ.ClientService.Repository

c) org.ENSAJ.ClientService.Controller

d) org.ENSAJ.ClientService.Service



2. Dans le package org.ENSAJ.ClientService.Model vous devez :

a) Créer la classe Client avec les attributs (Id Long, Nom String, Prénom String,Float age) 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

b) Ajouter les annotations Lombok :

@Data @AllArgsConstructor @NoArgsConstructor pour générer les setters, getters les constructeurs avec et sans arguments.

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

d) Ajouter au-dessus de l’attribut Id l’annotation JPA @GeneratedValue afin d’indiquer à Spring Boot que la valeur de ces champs 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;  
}

3. Dans le package org.ENSAJ.ClientService.Repostory

On doit :

a) Créer une interface ClientRepository

b) Faire hériter cette interface de l’interface Repository. Qui est de type Client. Le type de la clé est Long

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

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

4. Dans le package org.ENSAJ.ClientService.Controller on doit :

a) Créer une classe ClientController avec au-dessus l’annotation @RestController pour indiquer que c’est un contrôleur.

b) Créer un attribut clientRepository de type ClientRepository

c) Ajouter l’annotation @Autowired au-dessus de l’attribut Afin de tester que l’accès à la base de données se fait avec succès, on va créer des méthodes dans lesquelles on fait appel directement à la couche Repository sans passer par la couche présentation. Pour cela on doit :

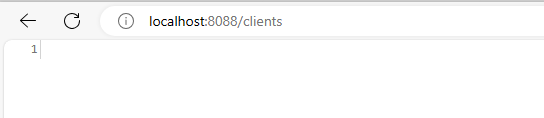
d) Créer la méthode List chercherClients() avec l’annotation @GetMapping("/clients") au-dessus. Dans cette méthode on fait appel à la méthode findAll() de l’attribut clientRepository déjà implémentée par Spring Boot. Cette méthode renvoie la liste des objets ClientRepository dans la base de données.

e) Créer la méthode List chercherUnClient(Long id) avec l’annotation @GetMapping("/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 chercherClients(){  
 return clientRepository.findAll();  
 }  
 @GetMapping("/client/{id}")  
 public Client chercherUnClients (@PathVariable Long id) throws Exception{  
  
 return this.clientRepository.findById(id).orElseThrow(()-> new Exception("Client inexistnt"));}

Notez qu’il faut ajouter l’annotation @PathVariable pour indiquer que le paramètre id de la fonction chercherUnClient 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 trouver dans la base l’objet correspondant.

5. Aller sur le navigateur est taper http://localhost:8088/clients :



Pour enregistrer des clients dans la base de données, il est nécessaire d'utiliser des commandes LineRunner, qui sont des commandes s'exécutant au lancement du projet. Ces commandes doivent être intégrées aux beans dans la fonction principale du programme.

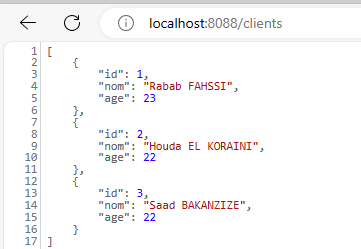
6. Créer une fonction nommée initialiserBaseH2 () ClientRepository clientRepository qui reçoit en paramètres un objet et qui retourne CommandLineRunner.

7. Ajouter au-dessus de cette méthode l’annotation @Bean

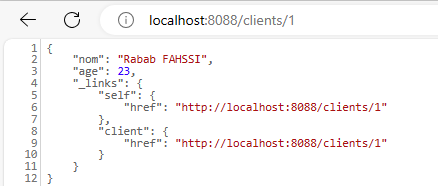
8. Utiliser le paramètres args qui est un tableau de String de la fonction main (String[] args) pour sauvegarder des objets client dans la base H2 via la méthode save(Client client) du clientRepository. Ce dernier objet est passé en argument à la fonction initialiserBaseH2()

@Bean  
CommandLineRunner initialiserBaseH2(ClientRepository clientRepository) {  
 return args -> {  
 clientRepository.save(new Client(Long.*parseLong*("1"), "Rabab FAHSSI", Float.*parseFloat*("23")));  
 clientRepository.save(new Client(Long.*parseLong*("2"), "Houda EL KORAINI", Float.*parseFloat*("22")));  
 clientRepository.save(new Client(Long.*parseLong*("3"), "Saad BAKANZIZE", Float.*parseFloat*("22")));  
  
 };  
}

9. Aller sur le navigateur est taper http://localhost:8088/clients :



On peut constater qu’on a pu récupérer les données sauvegardées dans la base H2 10. Aller sur le navigateur est taper http://localhost:8088/client/1 :

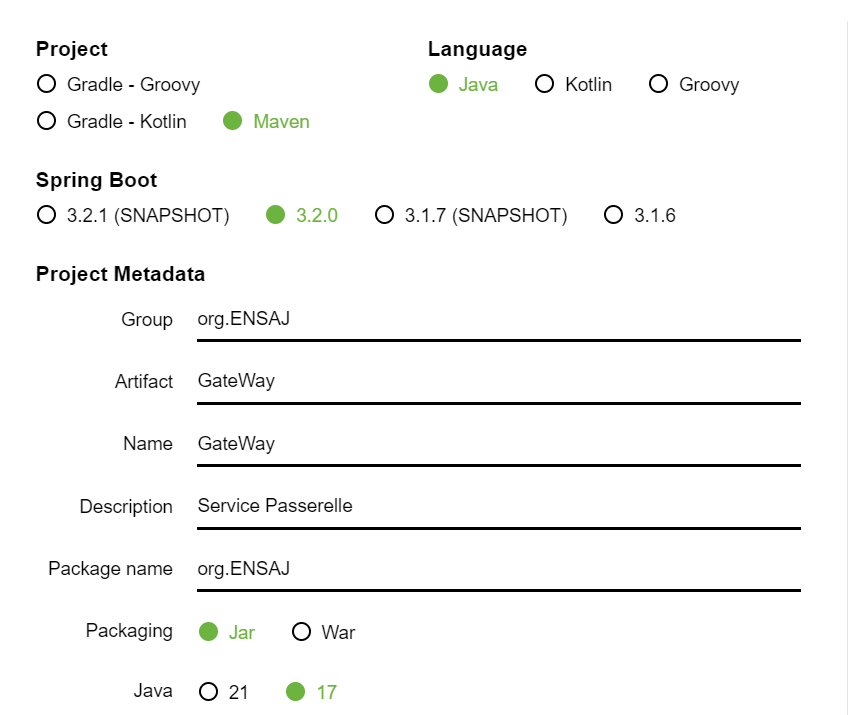


On voit bien que le contrôleur arrive à orienter la requêtes http Get à la méthode appropriée.

# III-Création d’un service Gateway

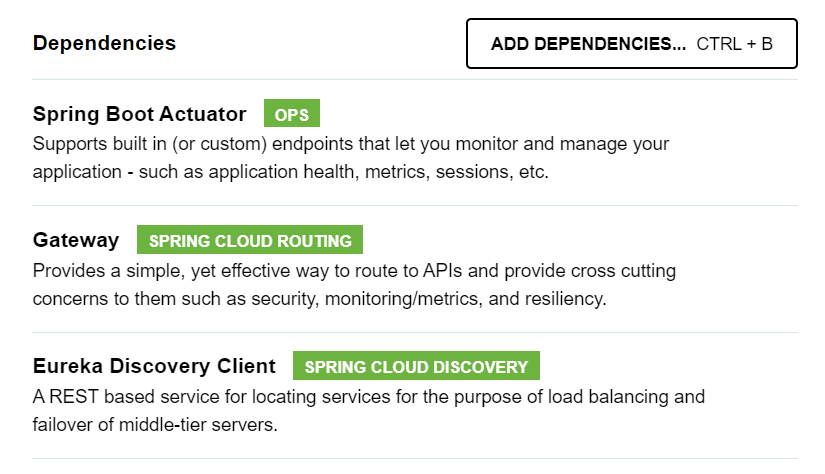
Pour créer un service Gateway, on doit procéder de la manière suivante :

1- Créer un nouveau projet sur Spring Initializr :



2- Ajoutez les dépendances suivantes et cliquez sur Generate :

* Spring Cloud Routing Gatway
* Spring Boot Actuator
* Eureka Discovry Client

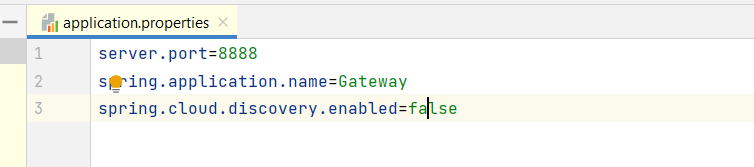


La configuration d’une GateWay peut se faire avec deux manières :

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

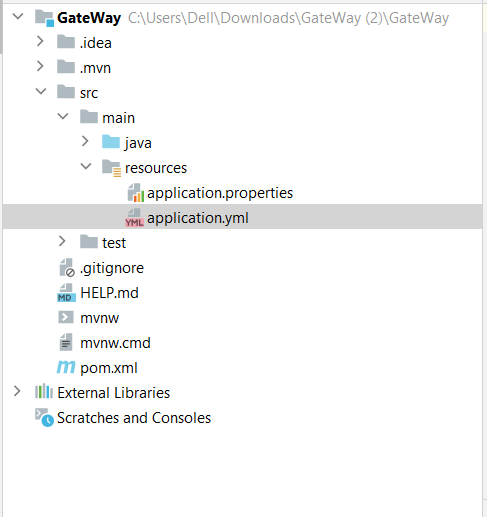
**Configuration statique**

1. Ouvrez le fichier application.properties est remplissez le comme suit :



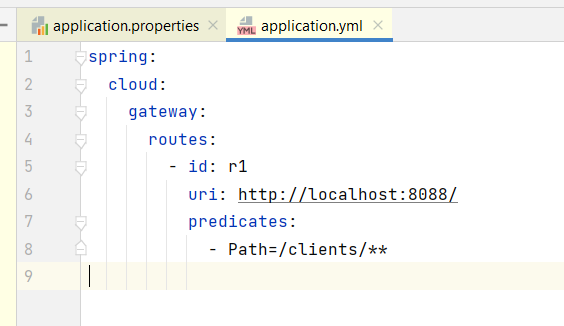
Cette configuration donne le nom Gateway à notre passerelle et définit son port à 8888. Après, désactivez l’enregistrement du service au service Discovery (On n’a pas besoin de ce service pour le moment).

1. Sur le dossier src/main/resources, créez un fichier yaml nommé application.yml :



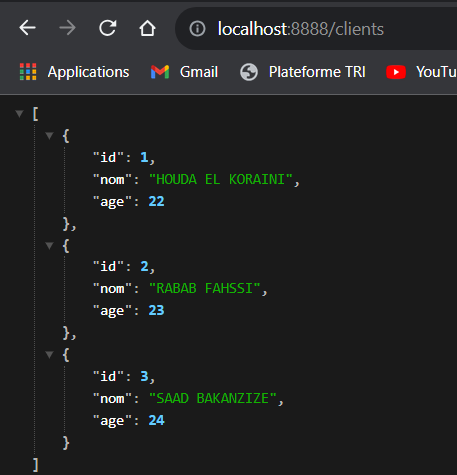
YAML ( Yet Another Markup Language) est langage de représentation de données. Il est utilisé généralement par Spring Boot pour des fins de configuration. On va l’utiliser ici afin de configurer notre passerelle Gateway pour le routage entre Micro-service.

1. Configurez le fichier application.yml comme suit :

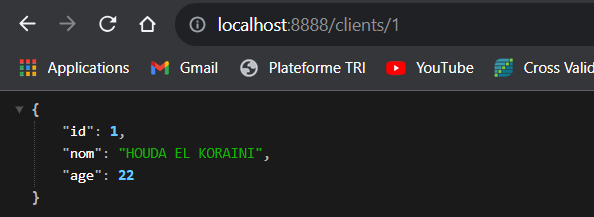


Cette configuration indique au micro-service Gateway de router les requêtes http ayant l‘url suivant : http:/localhost :8888/clients vers le Micro-service http:/localhost :8088/clients

1. Ouvrez votre navigateur et tapez http://localhost:8888/clients . La page web suivante doit s’afficher



1. Ouvrez votre navigateur et tapez http://localhost:8888/clients/1. La page web suivante doit s’afficher



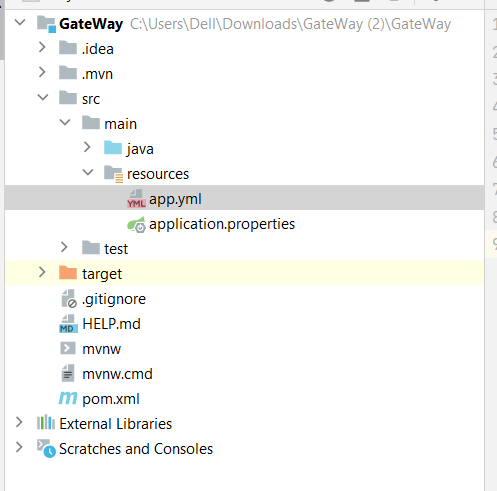
On voit bien que notre Micro-service Gateway marche bien !!

On peut aussi faire cette configuration avec du code Java. On veut ajouter en outre une option qui permet d’appeler le service concerné par son nom sur l’URL et non par son adresse IP.

Afin de réaliser ceci, on procède de la manière suivante :

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. Désactivez la configuration statique de la Gateway en renommant le fichier application.yml à app.yml :



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

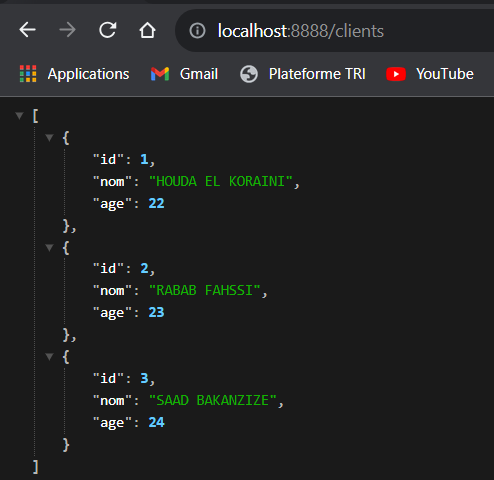
4. Ouvrez la main classe de la Gateway et ajoutez le Bean suivant :

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

5. Exécuter tous les Micro-services

6. Lancez votre navigateur et Tapez : <http://localhost:8888/client>

Bien sûr, on aura les mêmes résultats



**Configuration dynamique**

Pour la configuration dynamique c’est plus simple. On garde la même configuration que précédemment. Il faut juste commenter ou bien supprimer le Bean précédent et le remplacer par un nouveau Bean comme suit :

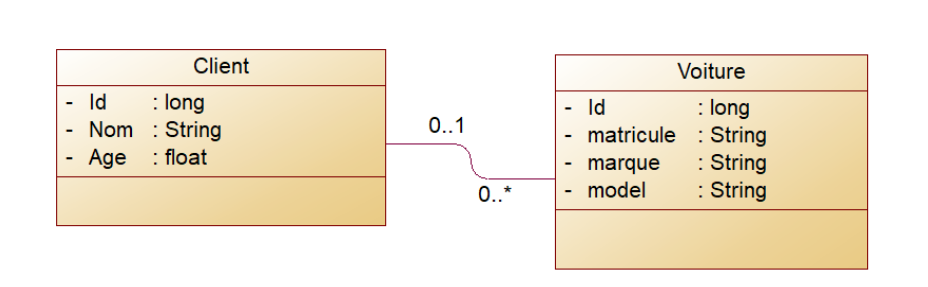
@Bean  
DiscoveryClientRouteDefinitionLocator routesDynamique(ReactiveDiscoveryClient rdc, DiscoveryLocatorProperties dlp){  
 return new DiscoveryClientRouteDefinitionLocator(rdc, dlp);  
}

Dans la configuration dynamique pour accéder au service voulu, il faut taper son nom dans l’URL (ex : <http://localhost:8888/SERVICE-CLIENT/clients>)

# IV- Architecture de l’application

Maintenant on va ajouter un autre service et on va connecter les 2 micro-services a la base de données H2 :

Les deux micro-services ont besoin de communiquer afin de garder une cohérence des données. Voici le diagramme de classe de notre application :



Transformez le diagramme de classe en code Java. Notez qu’on doit respecter les règles de transformation, notamment la transformation 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. Allez sur Maven Repository pour récupérer les dépendances liées à :

* OPENFEIGN
* hateoas

4. Ajoutez ces dépendances à votre projet.

5. Une fois la classe de la couche Model 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 pour indiquer à Spring boot que ce champ ne doit pas persister. L’annotation @ManyToOne indique que c’est une association plusieurs à un. La classe Voiture de la couche Model doit ressembler à ceci :

@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. Ajoutez la configuration suivante :

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

8. Créez un Bean avec des Commandes Lines Runner pour insérer des voitures dans la base de données H2

9. Tester sur un navigateur que tout va bien

Pour que ce M.S puisse se connecter au micro-service service-client, on doit créer une interface au-dessous de la classe VoitureApplication qu’on nommera 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

10. Créer la méthode clientById comme ceci :

Cette méthode reçoit en paramètre l’id du client récupéré en URI est renvoie l’objet client obtenu du Micro-service service-client.

Pour pouvoir récupérer l’id du M.S nommé service-client, on doit ajouter la configuration suivante son fichier properties : spring.cloud.discovery.enabled=true

Ceci permet d’exposer les Id des enregistrements client de la BD H2

11. Afin de tester votre Micro-service nommé service-voiture, modifier 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));  
 };  
}

12. Exécutez tous les micro-service : Eureka , Client et voiture

13. Lancez votre navigateur et tapez : http://localhost:8089/voitures. Sur la console vos aurez :

