

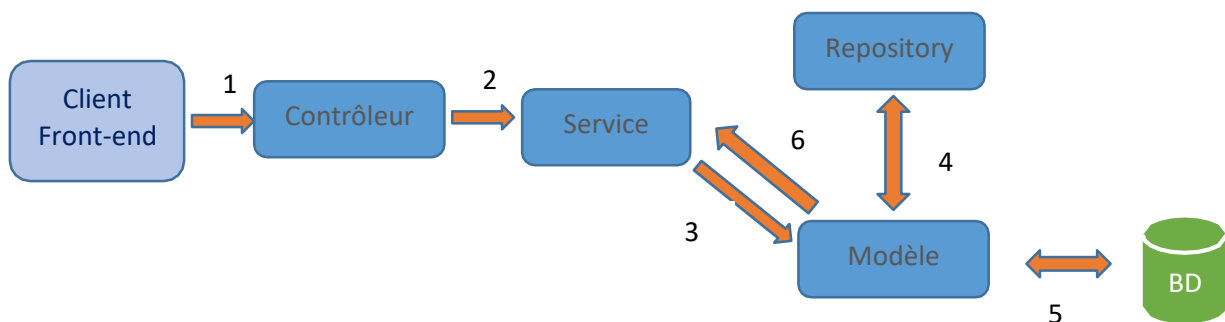
TP2

Objectifs du TP :

- Connecter un micro-service à une base de données In-memory H2.
- Création du micro-service **GateWay**

I- 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 :

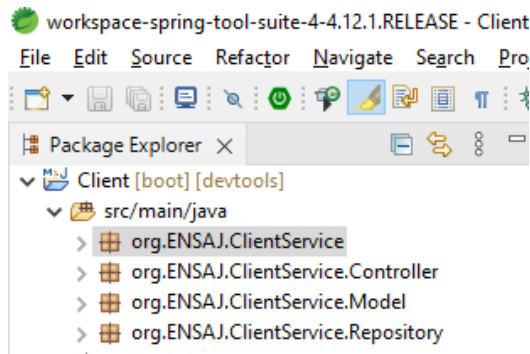


Dans cette architecture, la couche contrôleur, d'abord analyse le type de la requête http et appelle ensuite la méthode de la classe service correspondante. Cette méthode fait appel à son tour à la classe modèle afin de communiquer avec la BD. Ceci est réalisé grâce à une interface qui va hériter de l'interface *JpaRepository<T, classtype>* 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.

Ce TP est la continuité du TP1 du service client. On va supprimer la classe contrôleur crée précédemment et on va la remplacer par une autre classe qui va jouer le rôle du contrôleur.

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 :



- Créer la classe *Client* avec les attributs (**Id** Long, **Nom** String, **Prénom** String) 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` `@NoArgsConstructor` pour générer les setters, getters 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 ces champs est générée automatiquement.

```

1 package org.ENSAT.ClientService.Model;
2
3 import javax.persistence.Entity;
4 import javax.persistence.GeneratedValue;
5 import javax.persistence.Id;
6 import lombok.AllArgsConstructor;
7 import lombok.Data;
8 import lombok.NoArgsConstructor;
9
10 // pour la persistance de la classe via JPA
11 @Entity
12
13 // annotations lombok pour générer les setters, getters les constructeurs avec et sans arguments
14 @Data @AllArgsConstructor @NoArgsConstructor
15 public class Client {
16     // annotation pour dire que l'attribut Id est une clés de la classe Client
17     @Id
18     // pour générer les valeur d'Id automatiquement
19     @GeneratedValue
20     private Long Id;
21     private String Nom;
22     private Float Age;

```

3. Dans le package `org.ENSAT.ClientService.Repostory` on doit :

- Créer une interface *ClientRepository*
- Faire hériter cette interface de l'interface *Repository<Client, Long>*. 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 une repository

```

1 package org.ENSAT.ClientService.Repository;
2
3
4 import org.ENSAT.ClientService.Model.Client;
5 import org.springframework.data.jpa.repository.JpaRepository;
6 import org.springframework.stereotype.Repository;
7
8 @Repository
9 public interface ClientRepository extends JpaRepository<Client, Long> {
10
11 }

```

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

- Créer une classe `ClientController` avec au-dessus l'annotation `@RestController` pour indiquer que c'est un contrôleur.
- Créer un attribut `clientRepository` de type `ClientRepository`
- 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 :

- Créer la méthode `List<Client> 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.

```
1 package org. ENSAJ. ClientService. Controller;
2
3 import java.util. List;
4
5 import org. ENSAJ. ClientService. Model. Client;
6 import org. ENSAJ. ClientService. Repository. ClientRepository;
7 import org. springframework. beans. factory. annotation. Autowired;
8 import org. springframework. web. bind. annotation. GetMapping;
9 import org. springframework. web. bind. annotation. RestController;
10
11 @RestController
12 public class ClientController {
13
14     @Autowired
15     ClientRepository clientRepository ;
16
17     @GetMapping("/clients")
18     public List<Client> chercherClients() {
19         return clientRepository.findAll();
20     }
21 }
```

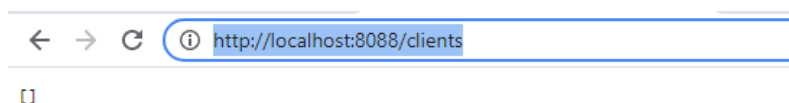
- Créer la méthode `List<Client> chercherUnClient(Long id)` avec l'annotation `@GetMapping("/client/{id}")` au-dessus. Dans cette méthode on fait appel à la méthode `findById(id)`. Cette méthode reçoit en paramètres un id de type Long et renvoie un objet `ClientRepository` avec le même Id depuis la base de données.

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

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 et taper <http://localhost:8088/clients> :



Le résultat est vide. C'est normale car la base de données est vide et il faut la remplir. Pour ceci, vous devez utiliser des commandes LineRunner (des commandes qui s'exécutent lors du lancement du projet) via des Beans dans la fonction main principale (figure suivante).

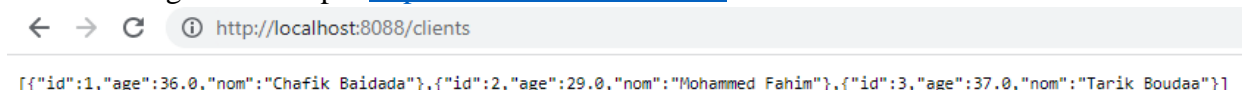
6. Créer une fonction nommée `initialiserBaseH2 ()` qui reçoit en paramètres un objet `ClientRepository clientRepository` et qui retourne `CommandLineRunner`.
7. Ajouter au-dessus de cette méthode l'annotation `@Bean`
8. Utiliser le paramètres `args` qui est un tableau de String de la fonction `main (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()`

```

2 import org.ENSAB.ClientService.Model.Client;
3 import org.ENSAB.ClientService.Repository.ClientRepository;
4 import org.springframework.boot.CommandLineRunner;
5 import org.springframework.boot.SpringApplication;
6 import org.springframework.boot.autoconfigure.SpringBootApplication;
7 import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
8 import org.springframework.context.annotation.Bean;
9
10
11
12 @EnableEurekaClient
13 @SpringBootApplication
14 public class ClientApplication {
15
16
17     public static void main(String[] args) {
18         SpringApplication.run(ClientApplication.class, args);
19     }
20
21     @Bean
22     CommandLineRunner initialiserBaseH2(ClientRepository clientRepository) {
23         return args->{
24             clientRepository.save( new Client(Long.parseLong("1"), "Chafik Baidada", Float.parseFloat("36")));
25             clientRepository.save( new Client(Long.parseLong("2"), "Mohammed Fahim", Float.parseFloat("29")));
26             clientRepository.save( new Client(Long.parseLong("3"), "Tarik Boudaa", Float.parseFloat("37")));
27         };
28     }

```

9. Aller sur le navigateur et taper <http://localhost:8088/clients> :

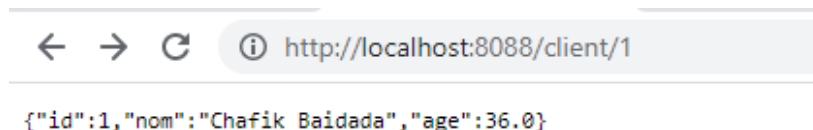


http://localhost:8088/clients

```
[{"id":1,"age":36.0,"nom":"Chafik Baidada"}, {"id":2,"age":29.0,"nom":"Mohammed Fahim"}, {"id":3,"age":37.0,"nom":"Tarik Boudaa"}]
```

On peut constater qu'on a pu récupérer les données sauvegardées dans la base H2

10. Aller sur le navigateur et taper <http://localhost:8088/client/1> :



http://localhost:8088/client/1

```
{"id":1,"nom":"Chafik Baidada","age":36.0}
```

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

II- Création de micro-service GateWay

Lorsqu'une application basée sur une architecture micro-service utilise un nombre réduit de services, on peut faire communiquer les clients avec chaque micro-service correspondant. Cependant, dès que le nombre augmente significativement, il devient compliqué de savoir quel micro-service devra être appelé ; et ce d'autant plus que, souvent, un client devra appeler plusieurs micro-services pour collecter l'ensemble des données nécessaire au fonctionnement de l'application. De plus, le nombre de requêtes envoyées au back-end accroît de manière à causer des problèmes au niveau de la performance de l'application.

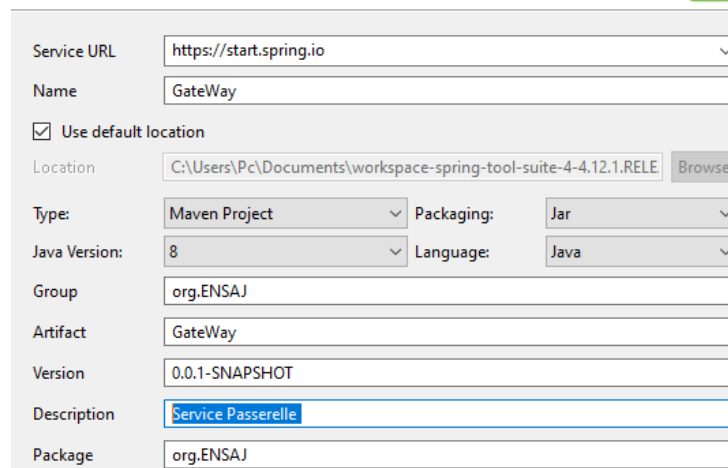
Pour gérer ces différentes problématiques on peut faire usage d'une **passerelle** d'API ou API Gateway. Une API Gateway est une sorte de proxy entre les API exposées par des micro-services (ou une application) et un front consommant ces API.

Pour mettre en place une passerelle Gateway sur Spring Boot, on doit d'abord créer un service et ensuite le configurer.

1. Création d'un service Gateway

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

- 1- Cliquez : **File->New -> Spring Starter Project** et remplissez la description du projet comme suit :

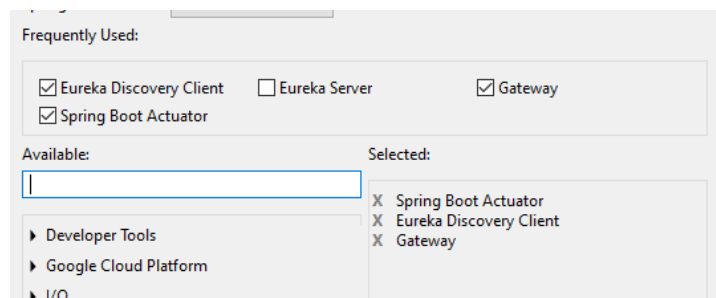


The screenshot shows the 'Spring Starter Project' configuration window. The fields are filled as follows:

- Service URL: <https://start.spring.io>
- Name: GateWay
- ☒ Use default location
- Location: C:\Users\Pc\Documents\workspace-spring-tool-suite-4-4.12.1.RELE
- Type: Maven Project
- Packaging: Jar
- Java Version: 8
- Language: Java
- Group: org.ENSJA
- Artifact: GateWay
- Version: 0.0.1-SNAPSHOT
- Description: Service Passerelle
- Package: org.ENSJA

- 2- Ajoutez les dépendances suivantes et cliquez sur *Finish* :

- Spring Cloud Routing Gateway
- Spring Boot Actuator
- Eureka Discovery Client



The screenshot shows the 'Frequently Used' dependencies section of the Spring Starter Project wizard. The 'Available' list is empty. The 'Selected' list contains:

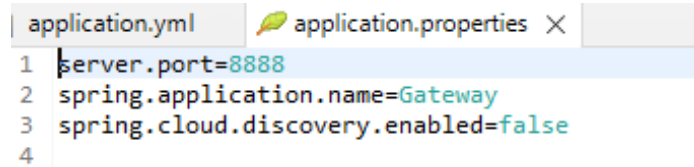
- X Spring Boot Actuator
- X Eureka Discovery Client
- X Gateway

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

- Statique via des fichiers yaml et proprietes ou bien via du code Java
- Dynamique avec du code Java seulement

Configuration statique

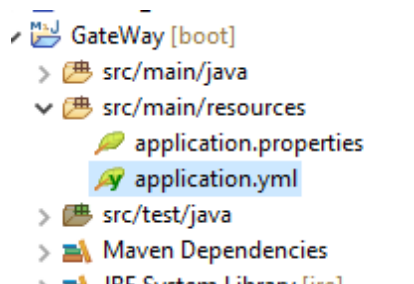
1- Ouvrez le fichier *application.properties* et remplissez le comme suit :



```
application.yml application.properties X
1 server.port=8888
2 spring.application.name=Gateway
3 spring.cloud.discovery.enabled=false
4
```

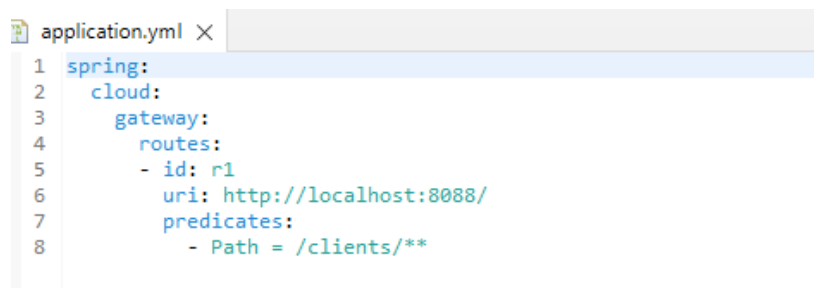
Cette configuration donne le nom Gateway à notre passerelle et définit son port à 8888. Après, désactivez l'enregistrement du service au service Discovery (On n'a pas besoin de ce service pour le moment).

2- 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.

3- Configurez le fichier *application.yml* comme suit :



```
application.yml X
1 spring:
2   cloud:
3     gateway:
4       routes:
5         - id: r1
6           uri: http://localhost:8088/
7           predicates:
8             - Path = /clients/**
```

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>

4- Ouvrez votre navigateur et tapez <http://localhost:8888/clients> . La page web suivante doit s'afficher

← → ↻ ⓘ http://localhost:8888/clients

```
[{"id":1,"age":36.0,"nom":"Chafik Baidada"}, {"id":2,"age":29.0,"nom":"Mohammed Fahim"}, {"id":3,"age":37.0,"nom":"Tarik Boudaa"}]
```

- 5- Ouvrez votre navigateur et tapez <http://localhost:8888/clients/1> .La page web suivante doit s'afficher

← → ↻ ⓘ http://localhost:8888/clients/1

Afficher des informations à propos du

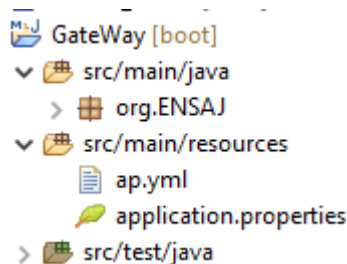
```
{
  "id" : 1,
  "age" : 36.0,
  "nom" : "Chafik Baidada",
  "_links" : {
    "self" : {
      "href" : "http://localhost:8088/clients/1"
    },
    "client" : {
      "href" : "http://localhost:8088/clients/1"
    }
  }
}
```

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 :

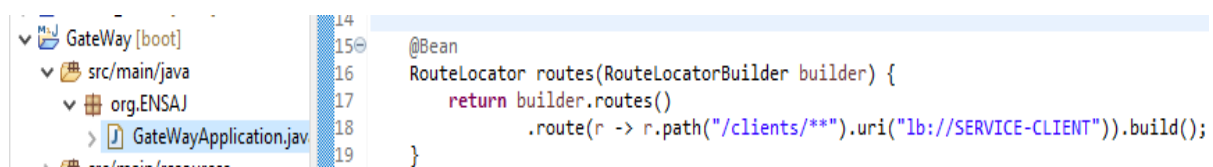
1. Avant de commencer, il faut d'abord reconfigurer les micro-services client et Gateway pour leur autoriser de s'auto-enregistrer sur le service Discovery Eureka.
2. Désactivez la configuration statique de la Gateway en renommant le fichier *application.yml* à *ap.yml* :



 GateWay [boot]

- src/main/java
 - org.ENSASJ
- src/main/resources
 - ap.yml
 - application.properties
- src/test/java

3. Ajouter la ligne `eureka.instance.hostname=localhost` sur le fichier *application.properties*
4. Ouvrez la main classe de la Gateway et ajoutez le Bean suivant :



 14

 15 @Bean

 16 RouteLocator routes(RouteLocatorBuilder builder) {

 17 return builder.routes()

 18 .route(r -> r.path("/clients/**").uri("lb://SERVICE-CLIENT")).build();

 19 }

5. Exécuter tous les M.S
6. Lancez votre navigateur et Tapez : <http://localhost:8888/client/1>

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 :

```
33  
34  
35 @Bean  
36 DiscoveryClientRouteDefinitionLocator routesDynamique(ReactiveDiscoveryClient rdc, DiscoveryLocatorProperties dlp) {  
37     return new DiscoveryClientRouteDefinitionLocator(rdc, dlp);  
38 }  
39  
40  
41 }
```

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