

## TP4 Microservices (2<sup>ème</sup> partie)

**Objectif du TP :** Dans cette 2<sup>ème</sup> partie de TP on va compléter l'application microservice abordé au TP précédent en :

- Ajoutant le microservice `rdv-service` qui va gérer les RDVs et communiquer avec les 2 microservices `patient-service` et `medecin-service`.
- Configurait le Config Service qui centralisera les configurations des différents microservices.

### Création de `rdv-service` et de ses composants :

Ajouter un autre module de type Spring Initializr dans ce projet nommé `rdv-service` ayant les mêmes dépendances que `patient-service`. Copier les composants nécessaires dans ce module (l'entité, le Repository, les services et le contrôleur). Attribuer le port 8083 à ce service qui aura comme BD `bd-rdv`.

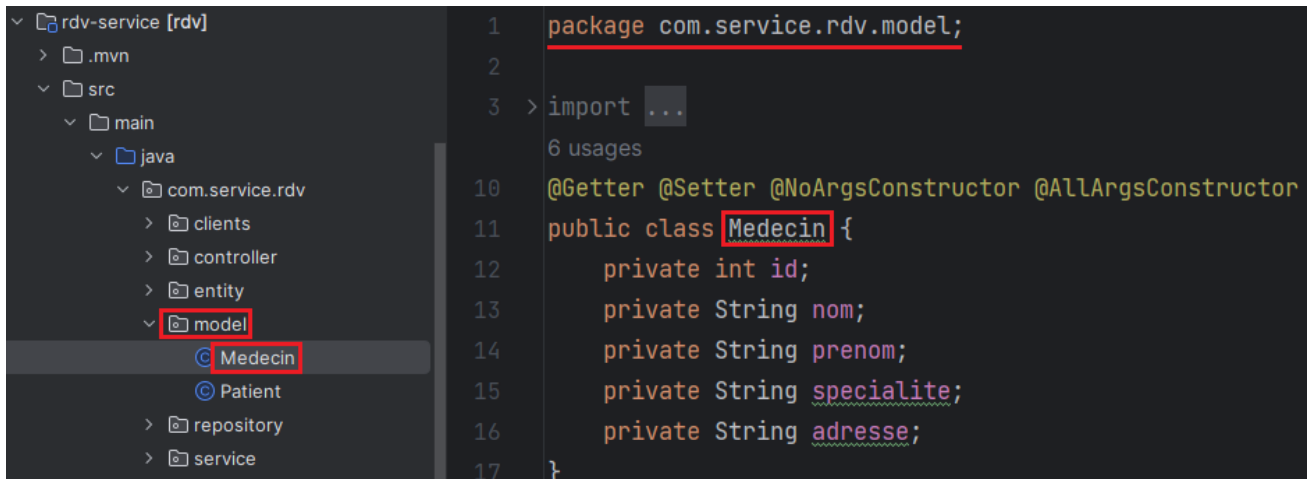
Dans l'entité `Rdv`, Remplacer les attributs de type entités `Patient` et `Medecin` par deux attributs de type `int`. Chacun des attributs ajoutés (`patientId` et `medecinId`) permettra de référencer l'entité correspondante :

```
@Getter @Setter @NoArgsConstructor @AllArgsConstructor
@Entity
public class Rdv {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    @JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss", shape = JsonFormat.Shape.STRING)
    private LocalDateTime dateRdv;
    private String etat;
    private int patientId;
    private int medecinId;
}
```

Dans la classe `ServiceRdv` apporter les changements suivants dans l'implémentation de la méthode `addRdv` :

```
public Rdv addRdv(Rdv rdv) {
    Rdv rdv1 = rdvRepository.findByPatientIdAndDateRdv(rdv.getPatientId(), rdv.getDateRdv());
    Rdv rdv2 = rdvRepository.findByMedecinIdAndDateRdv(rdv.getMedecinId(), rdv.getDateRdv());
    if(rdv1==null && rdv2==null)
        return rdvRepository.save(rdv);
    else
        return null;
}
```

Dans l'entité `Rdv`, par exemple, la valeur de `medecinId` permettra de récupérer les informations sur le médecin concerné par un RDV en envoyant vers le microservice `medecin-service` **une requête de recherche par id**. Les informations récupérées seront placées dans une instance d'une classe (**non entité**) nommée `Medecin` définies dans un package nommé `model` de `rdv-service`. Cette classe contiendra les attributs à récupérer sur le médecin :



Puis on va ajouter dans l'entité `Rdv` un autre attribut qui contiendra les informations sur le médecin mais cet attribut ne sera pas représenté dans la table `rdv` (en l'annotant par `@Transient`). Cette annotation indique à Spring Data JPA qu'il faut ignorer la représentation de cet attribut dans la table `rdv` :

```
public class Rdv {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    @JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss", shape = JsonFormat.Shape.STRING)
    private LocalDateTime dateRdv;
    private String etat;
    private int patientId;
    @Transient
    private Medecin medecin;
    private int medecinId;
}
```

Ajouter aussi la définition d'une autre classe nommé `Patient` dans le package `model`, dont une référence sera ajoutée dans l'entité `Rdv` et annotée `@Transient`.

Dans l'interface `IServiceRdv`, ajouter la signature d'une méthode qui retourne un RDV par son id :

```
Rdv getRdvById(int id);
```

Implémenter cette méthode dans la classe `ServiceRdv` :

```
@Override
public Rdv getRdvById(int id) { return rdvRepository.findById(id).get(); }
```

Dans le microservice `patient-service`, ajouter dans l'interface `IServicePatient` la signature d'une méthode qui recherche par id et retourne un `Optional<Patient>`. Implémenter cette méthode dans la classe `ServicePatient` et ajouter dans le contrôleur `PatientRestController` une action qui traite la recherche par la méthode GET un patient par son id :

```
@GetMapping("/{id}")
public Optional<Patient> getById(@PathVariable int id){
    return iServicePatient.getPatientById(id);
}
```

Faite de même dans le microservice `medecin-service`.

## Communication synchrone entre les microservices

Lorsque nous avons définie l'entité `Rdv` dans `rdv-service`, nous avons ajouté un attribut `medecinId` (comme clé étrangère) pour référencer le médecin traitant ce patient et un attribut (Transient) `medecin` prévu pour contenir le détail de ce médecin lorsqu'on va envoyer une requête vers `medecin-service` pour récupérer les données d'un médecin à partir de son id. Ce médecin sera affiché comme détail du `rdv`. Cette explication est aussi valable pour l'attribut `patientId`.

Pour pouvoir récupérer un médecin depuis `medecin-service`, (et un patient de `patient-service`) on va utiliser un Framework Rest nommé Open Feign.

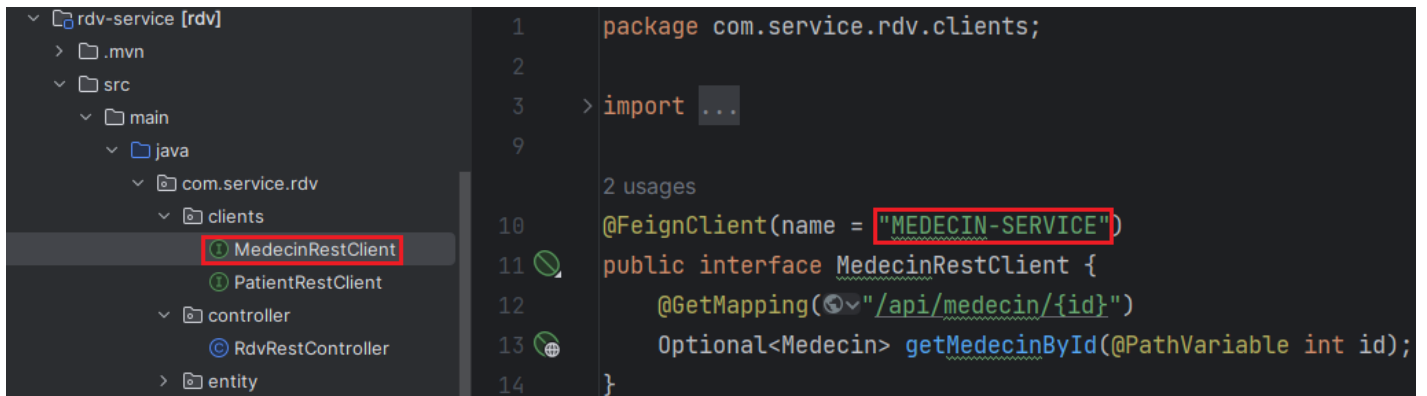
Ce Framework est déclaratif c'est-à-dire qu'à partir d'une interface on peut déclarer les méthodes qui réalisent la communication entre les microservices (identique aux interfaces Repository de Spring Data JPA).

A partir du site [Maven Repository](#), rechercher la description XML de la dépendance de OpenFeign puis l'ajouter dans `pom.xml` de `rdv-service` et recharger le.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-openfeign</artifactId>
  <version>5.0.0</version>
</dependency>
```

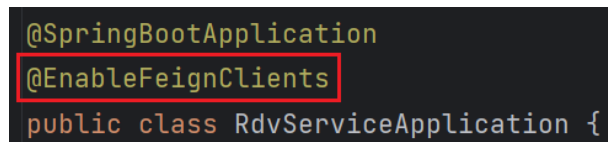
Dans `rdv-service`, créer un nouveau package nommé `clients` dans lequel on va créer une interface nommée `MedecinRestClient` qui contiendra la signature de la méthode qui va récupérer un médecin par son id. Cette interface doit être annotée par `@FeignClient` pour indiquer le nom du service dans lequel sera exécutées les méthodes déclarées dans l'interface. Finalement, la méthode est mappée avec une requête HTTP de type

GET qui contient dans le path le id du médecin (identique que dans l'action de MedecinRestController):

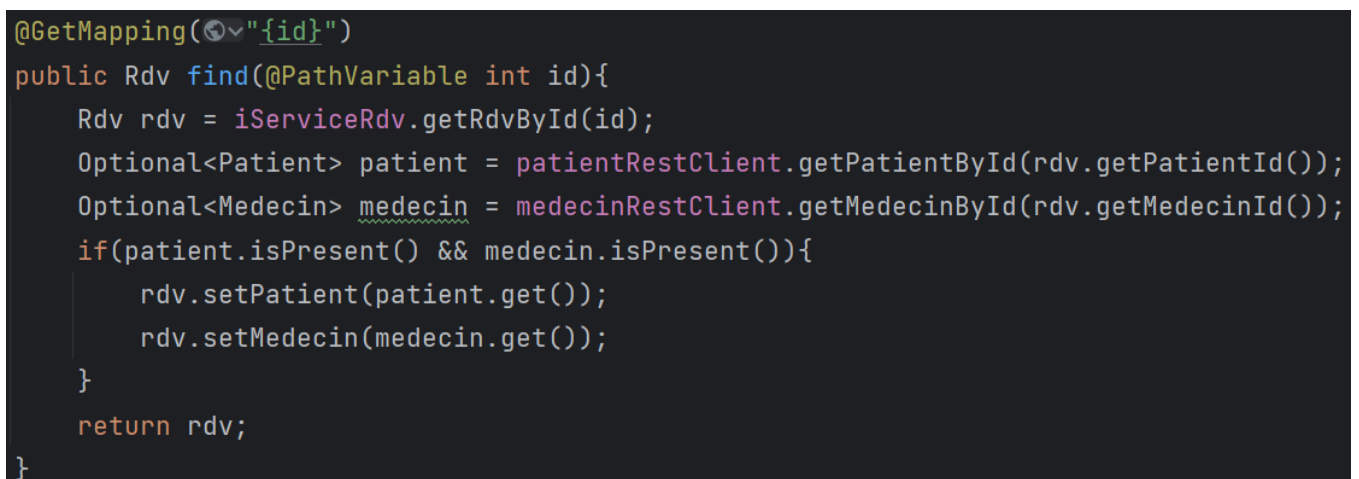


Dans le package clients, créer de la même manière, l'interface PatientRestClient.

Pour activer Open Feign, ajouter dans la classe de démarrage de rdv-service l'annotation @EnableFeignClients :



Dans le RdvRestController, ajouter une injection des dépendances sur les interfaces MedecinRestClient et PatientRestClient puis implémenter l'action de recherche d'un RDV par son id pour inclure la recherche et l'affectation du médecin et du patient par leurs id respectifs :



Toujours dans le RdvRestController, modifier l'action add d'un rdv pour vérifier l'existence des ids du patient et medecin avant de faire l'ajout d'un nouveau RDV sinon affiche un message adéquat :

```

@PostMapping("add")
public ResponseEntity<Object> add(@RequestBody Rdv rdv){
    Rdv rdv1=null;
    Optional<Patient> patient = patientRestClient.getPatientById(rdv.getPatientId());
    Optional<Medecin> medecin = medecinRestClient.getMedecinById(rdv.getMedecinId());
    if(patient.isPresent() && medecin.isPresent())
        rdv1 = iServiceRdv.addRdv(rdv);
    if(rdv1!=null)
        return new ResponseEntity<>(rdv, HttpStatus.CREATED);
    else
        return new ResponseEntity<>({ body: "Le Rdv ne peut pas être crée, merci de vérifier vos données", HttpStatus.C
}

```

On démarre les 5 services en commençant par le discovery-service puis on effectue une recherche d'un rdv par son id :

localhost:8888/rdv-service/api/rdv/1

```

{
  "id": 1,
  "dateRdv": "2024-11-30 10:30:00",
  "etat": "en cours",
  "patient": {
    "id": 2,
    "nom": "Helal",
    "prenom": "Chadi",
    "age": 45,
    "tel": 73222555
  },
  "patientId": 2,
  "medecin": {
    "id": 1,
    "nom": "Touhami",
    "prenom": "Salem",
    "specialite": "Généraliste",
    "adresse": "Sousse"
  },
  "medecinId": 1
}

```

Les informations sur le patient et le médecin proviennent respectivement de la BD de patient-service, tandis que les informations sur le médecin proviennent de la BD de medecin-service. La date et l'état du RDV proviennent de la BD de rdv-service.

Dans le contrôleur RdvRestController, apporter les modifications nécessaires dans l'action allRdv pour afficher dans chaque Rdv de la liste, les informations sur le médecin et patient.

## Spring Cloud Config Server

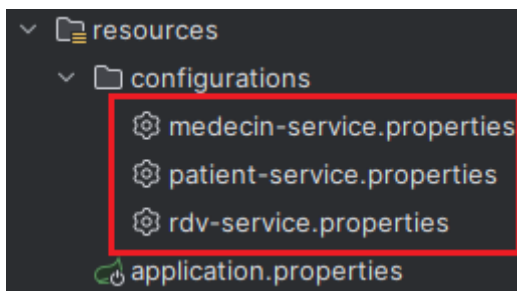
Spring Cloud Config Server a pour rôle principale de centraliser la configuration des différents microservices d'une même application dans un seul endroit : config-service. Pour démarrer, chaque microservice va importer sa propre configuration de config-service. On commence par activer config-service comme un serveur de configuration. Pour cela, ajouter dans la classe de démarrage de config-service l'annotation :

```
@SpringBootApplication
@EnableConfigServer
public class ConfigServiceApplication {
```

Ajouter la configuration suivante dans application.properties de config-service :

```
server.port= 9999
spring.application.name= config-service
spring.profiles.active=native
spring.cloud.config.server.native.search-locations= classpath:/configurations
```

Le dossier configurations (créé sous ressources) contiendra les différentes configurations des microservices. Par exemple, pour patient-service, créer un fichier patient-service.properties (ou yml) sous configurations :



Copier et coller le contenu de application.properties de patient-service dans le fichier patient-service.properties de config-service. N'oublier pas de modifier la propriété spring.cloud.config.enabled à true.

Changer le contenu de application.properties de patient-service par le suivant :

```
spring.application.name=patient-service
spring.config.import=optional:configserver:http://localhost:9999
```

La 2ème propriété indique au microservice patient-service d'où importer sa configuration au moment du démarrage.

**Faites les mêmes modifications pour medecin-service et rdv-service.**

Redémarrer les 6 microservices dans l'ordre suivant : config-service, discovery-service, gateway-service, medecin-service, patient-service et rdv-service puis interagir avec les 3 derniers microservices à travers la gateway.