esprit Se former autrement	Année Universitaire : 2025-2026 **Applications Web Distribuées** **UP_WEB**

Atelier-Communication entre les Micro-services

Job et Candidat

Objectif

Fournir une compréhension approfondie de l'utilisation de **OpenFeign** pour assurer une **communication synchrone** entre les micro-services,

Introduction

Assurer la communication entre les services d'une manière efficace est une étape très importante dans une architecture à base de micro-services. Ce qui va permettre d'avoir une application performante, évolutive et maintenable.

La communication peut être classée en deux grandes catégories :

- **Communication synchrone** : le service qui effectue l'appel attend une réponse avant de continuer son exécution ;
  Parfait pour **les interactions directes**, par exemple, lorsqu'un micro-service doit interroger un autre service pour obtenir des informations en temps réel ou pour effectuer une opération qui nécessite une réponse immédiate.

- **Communication asynchrone** : le service envoie un message sans attendre de réponse immédiate.

  Utilisé pour **des traitements déconnectés** où les services peuvent fonctionner indépendamment et de manière parallèle. Généralement pour les tâches asynchrones, telles que les envois d'emails, les notifications, ou les tâches programmées

Le choix entre ces deux modes dépend de divers facteurs tels que la nature des interactions au niveau de l'application (simples et directes nécessitant une réponse immédiate ou bien complexes avec réponse à long terme) et la complexité des transactions.

Avec le **mode synchrone**, les micro-services communiquent souvent entre eux via des appels HTTP. Pour simplifier et abstraire ces appels, la bibliothèque Java **OpenFeign**, est utilisée pour créer des clients HTTP/REST.

1. Principe de communication avec OpenFeign :

Feign est un Client HTTP déclaratif pour **Java**, facilitant les appels entre micro-services.

Il permet d'appeler une ressource d'un service externe comme s'il s'agissait d'une méthode locale en :

- Permettant de déclarer des interfaces Java qui représentent les appels API.
- Utilisant des annotations pour spécifier les Endpoints et les paramètres des requêtes, ce qui rend le code plus propre et plus lisible.

Avec OpenFeign, les requêtes sont envoyées via des méthodes HTTP standard comme GET, POST, PUT, DELETE, et les réponses sont attendues de **manière synchrone**.

2. Création et Utilisation d'un Client Feign :

Scénario 1 : Le service Candidat appelle le service Job pour obtenir la liste des jobs disponibles et chercher un job par id

Le micro-service "Job" contiendra les informations sur les jobs dans sa base de données, tandis que le micro-service "Candidat" utilisera OpenFeign pour récupérer les données depuis le micro-service "Job".

Pour implémenter l'affichage de la liste des jobs à partir du micro-service candidat, chaque micro-service doit être configuré pour accéder à sa propre base de données.

1) Configuration de la communication entre les micro-services avec OpenFeign :

Ajouter la dépendance suivante dans le fichier pom.xml du micro-service Candidat :

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

2) **Activation de Feign Client (@EnableFeignClients)**

Activer Feign Clients dans l'application du micro-service "Candidat" pour appeler l'API du micro-service "Job":

```
@SpringBootApplication
@EnableFeignClients
public class CandidateApplication {
    unknown
    public static void main(String[] args) {
        SpringApplication.run(CandidateApplication.class,
    }
```

En trouvant cette annotation, Spring Boot va automatiquement scanner votre code pour trouver des interfaces annotées avec **@FeignClient** et les enregistrer.

3) **Création d'une casse DTO Job dans le service Candidat :**

Ajouter la classe Job dans le micro-service "Candidat". Cette classe représente un DTO (Data Transfer Object) qui sert uniquement à transférer les données entre les micro-services "Candidat" et "Job". Elle ne sera pas donc persistée dans la base de données du micro-service "Candidat" :

```
package tn.example.candidat;

public class Job {

    private  int id;
    private String service;
    private boolean etat;

    // Ajouter constructeurs, Getters et Setters
}
```

Le micro-service "Candidat" va utiliser la classe Job suivante pour recevoir les données du micro-service "Job". Puis, il utilisera simplement ces données à des fins d'affichage ou d'autres opérations sans les stocker localement dans sa base de données.

NB : Si vous souhaitez stocker les Jobs dans la DB du service Candidat, vous devez :

1. Créer une **entité similaire Job** qui sera stockée dans la base de données **H2**.
2. Définissez également un repository dans **CandidatService** pour l'entité Job.

4) Création d'un client Feign :

a. Créer l'interface java **JobClient** annotée avec « **@FeignClient** » pour simplifier et abstraire les appels HTTP entre les micro-services.
Cette interface déclare les Endpoints REST du micro-service Job que le micro-service Candidate souhaite appeler.

```
package tn.example.candidat;

@FeignClient(name = "job-s", url = "http://localhost:8081")
public interface JobClient {

    @RequestMapping("jobs")
    public List<Job> getAllJobs();

    @RequestMapping("jobs/{id}")
    public Job getJobById(@PathVariable int id);
}
```

L'annotation @FeignClient est utilisée pour indiquer que l'interface est un client Feign. Cela signifie que l'interface va être utilisée pour définir les appels HTTP vers un service distant.

- **name** : représente le nom logique du service auquel vous souhaitez vous connecter, tel qu'il est enregistré dans le **serveur de découverte Eureka** ou tel que défini dans la configuration de votre application.
Ici, job-s est le nom du service Job tel qu'il est enregistré dans Eureka, et Feign peut découvrir automatiquement l'URL de ce service via Eureka sans l'ajout de url.

- **url** : l'adresse d'accès au service Job. Il n'est **pas obligatoire** si vous utilisez un **service de découverte comme Eureka**, car Feign trouvera automatiquement l'URL basée sur le "name" du service. Doit être spécifié si pas de découverte de services

Les ressources `getAllJobs()` et `getJobById(@PathVariable int id)` sont déjà définies dans le contrôleur du service Job :

```
package com.esprit.jobms;
import ...

@RestController
@RequestMapping("/jobs")
public class JobRestAPI {


    2 usages
    @Autowired
    private JobRepository jobRepository;

    @RequestMapping("/")
    public List<Job> getAllJobs() { return jobRep

    @RequestMapping("/{id}")
    public Job getJobById(@PathVariable int id) {
}
```

Le path « jobs » ajouté dans l'`interface JobClient` est le même déjà utilisé pour identifier les ressources à consommer à partir du service Job externe.

5) Injection du Client Feign créé:

Dans le service **Candidat**, nous allons implémenter la logique pour appeler le service Job via OpenFeign :

```
@Service
public class CandidatService {
    @Autowired

    private CandidatRepository candidateRepository;
    @Autowired
    private JobClient jobServiceClient;

    public List<Job> getJobs() {
        return jobServiceClient.getAllJobs();
    }


    public Job getJobById(int id) {
        return jobServiceClient.getJobById(id);
    }
 // Le reste de la logique du service Candidat
}
```

Puis, dans le contrôleur du Candidat, on injectera les services définis :

```
@RestController
@RequestMapping("/candidats")
public class CandidatResAPI {
```

```
    @Autowired
    private CandidatService candidatService;


    @RequestMapping("/jobs")
    public List<Job> getAllJobs() {
        return candidatService.getJobs();
    }

    @RequestMapping("jobs/{id}")
    public Job getJobById(@PathVariable int id) {
        return candidatService.getJobById(id);
    }
 // Le reste des ressources liées à l'API Candidat
}
```

6) Tester l'API pour Afficher les Jobs :

- Démarrer le serveur de découverte Eureka.
- Démarrer les deux micro-services Candidat et Job.
- Tester l'affichage des jobs disponibles à partir du micro-service Job en envoyant les requête GET suivantes : http://localhost:8089/candidats/jobs puis http://localhost:8089/candidats/jobs/1

Scénario 2 : Le service Candidat appelle le service Job pour enregistrer les jobs dans la liste de favoris.

On souhaite maintenant ajouter un job aux favoris du candidat et récupérer les jobs favoris du candidat.

1. On commence par l'ajout de la variable favoriteJobs suivante dans l'entité Candidat :

```
@Entity
public class Candidat {

    @Id
    @GeneratedValue
    private  int id;
    private String nom, prenom, email;

    @ElementCollection
    private Set<Integer> favoriteJobs = new HashSet<>();

    // Generate Getters and Setters
```

```
}
```

- **@ElementCollection** : Cela indique à JPA que la collection **favoriteJobs** doit être persistée en tant qu'éléments simples (ici des **Integer**) dans une table séparée, associée à l'entité contenant cette collection.
  Lorsque tu utilises **@ElementCollection**, JPA va générer automatiquement une table pour stocker les éléments de la collection, distincte de la table principale de l'entité (celle de Candidat dans notre cas). Cette table supplémentaire contiendra les ID des jobs favoris.
- **Set<Integer>** : Une collection d'**Integer** utilisée pour stocker les identifiants des jobs favoris d'un candidat.

2. Dans la classe **CandidatService**, ajoutez la logique des deux fonctions permettant l'ajout d'un job aux favoris du candidat et la récupération des jobs favoris du candidat :

```
@Service
public class CandidatService {
------------------------------
------------------------------
public List<Job> getFavoriteJobs(int candidateId) {
    Candidat candidate = candidateRepository.findById(candidateId).get();
    return candidate.getFavoriteJobs().stream()
            .map(jobServiceClient::getJobById)
            .collect(Collectors.toList());
}

public void saveFavoriteJob(int candidateId, int jobId) {
    Candidat candidate = candidateRepository.findById(candidateId).get();
    candidate.getFavoriteJobs().add(jobId);
    candidateRepository.save(candidate);
}
```

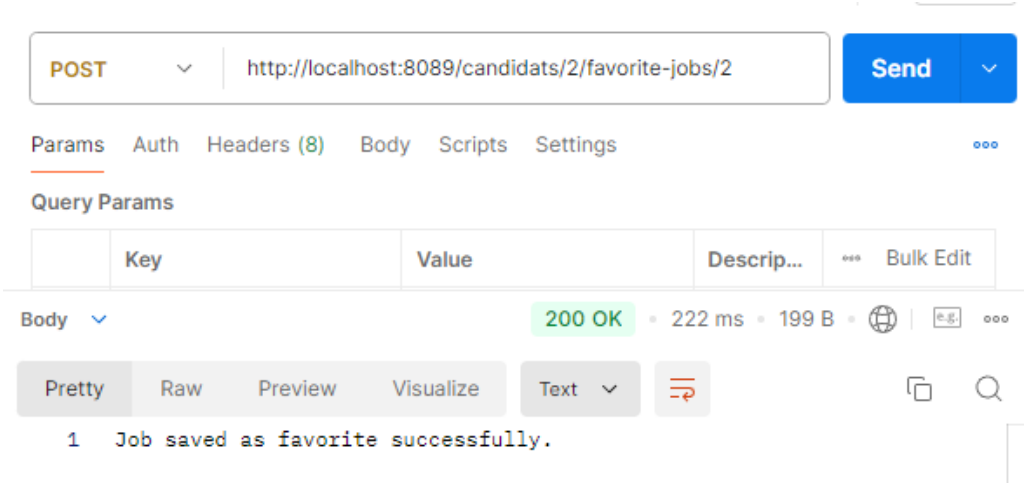
Sachant que `jobServiceClient` représente le **client Feign** injecté à partir duquel on consomme les méthodes du service Job.

3. On expose les 2 fonctions précédentes à partir du controlleur :

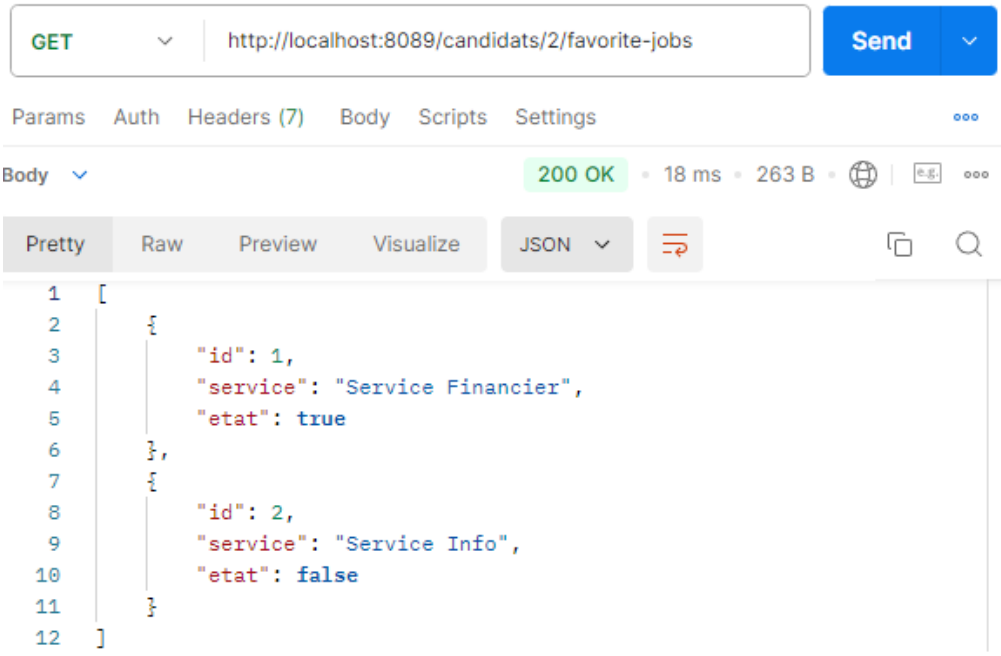
```
@RestController
@RequestMapping("/candidats")

------------------------------
------------------------------


@GetMapping("/{id}/favorite-jobs")
public List<Job> getFavoriteJobs(@PathVariable int id) {
    return candidatService.getFavoriteJobs(id);
}
```

```java
@PostMapping("/{id}/favorite-jobs/{jobId}")
public ResponseEntity<String> saveFavoriteJob(@PathVariable int id, @PathVariable
int jobId) {
    Job job = candidatService.getJobById(jobId);
    if (job != null) {
        candidatService.saveFavoriteJob(id, jobId);
        return ResponseEntity.status(HttpStatus.OK).body("Job saved as favorite
successfully.");

    } else {
        // Gérer le cas où le job n'existe pas
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
                .body("Job not found with ID: " + jobId);
    }
}
```

4. Tester l'API pour enregistrer un Job dans la liste de favoris d'un Candidat :



Puis, pour afficher la liste de favoris d'un Candidat. Ici, on suppose que le Candidat a envoyé la requête précédentes 2 fois afin d'ajouter deux Jobs dans sa liste de Favoris :


GET
http://localhost:8089/candidats/2/favorite-jobs
Send
Params
Auth
Headers (7)
Body
Scripts
Settings
Body
200 OK
18 ms
263 B
Pretty
Raw
Preview
Visualize
JSON
[
{
"id": 1,
"service": "Service Financier",
"etat": true
},
{
"id": 2,
"service": "Service Info",
"etat": false
}
]