esprit Se former autrement HONORIS UNITED UNIVERSITIES	Année Universitaire : 2024-2025 **Applications Web Distribuées**

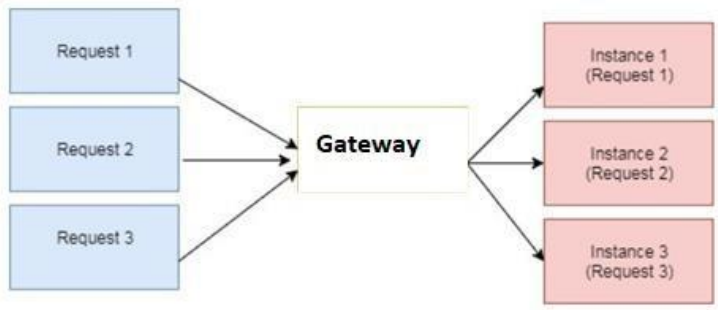
Implémentation « API Gateway»

Objectif

Une API Gateway agit comme un point d'entrée unique pour l'accès aux microservices. Lorsqu'elle reçoit une requête, elle sélectionne une instance de service disponible et lui transfère la requête. L'objectif de cet atelier est d'organiser l'appel vers les microservices en utilisant une API Gateway.

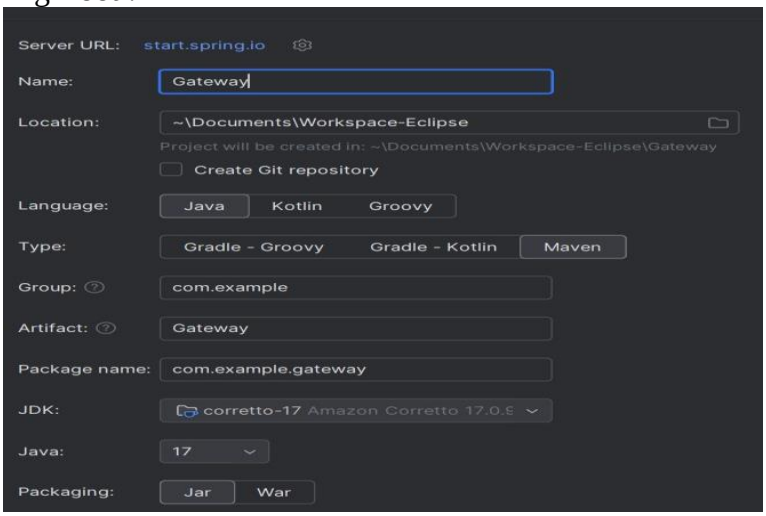
Contenu du Workshop

Partie 1 : Création d'un projet API Gateway
Partie 2 : Configuration statique de l'API Gateway
Partie 3 : Configuration dynamique de l'API Gateway avec Eureka
Rendu : Répondre aux questions de la Partie 3 (2.b et 3).

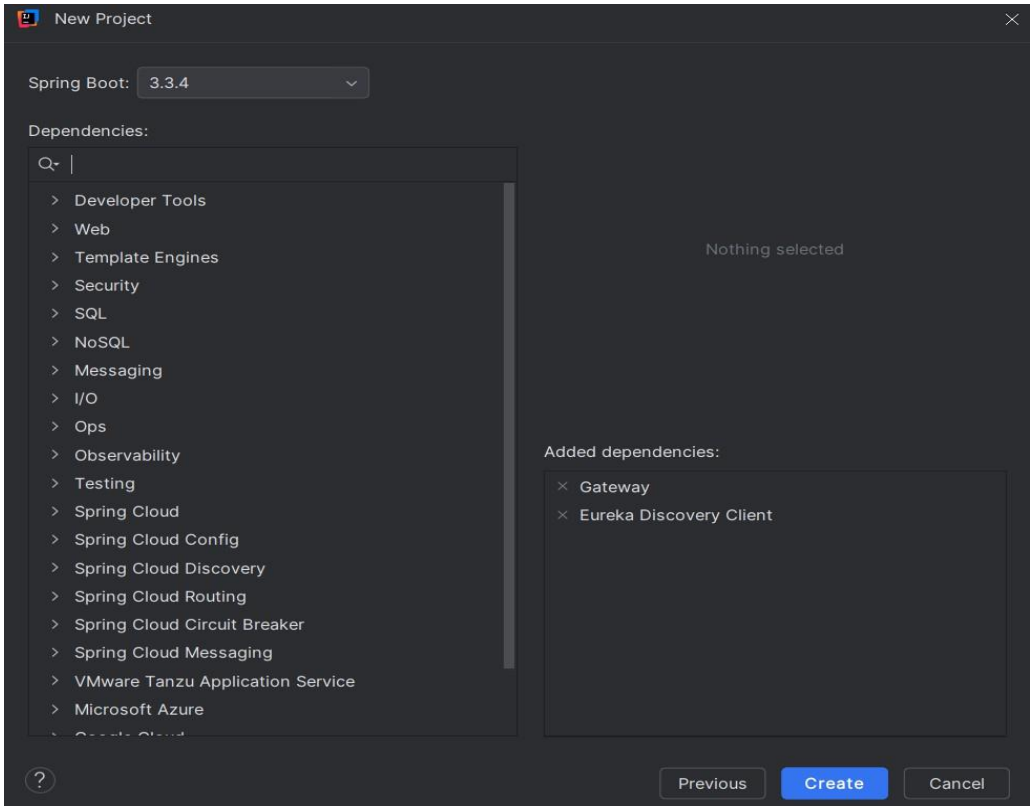


A.Partie 1 : Créer un Projet

1. Créez un projet Spring Boot :



2. Ajoutez ces deux starters : Eureka Discovery client et Gateway



Vous obtenez ces deux dépendances dans votre pom.xml :

```
 <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
<dependency>
        <groupId>org.springframework.cloud</groupId>
       <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

3. Faites un maven update project et clean install
4. Au niveau de la classe main spring Boot de votre projet, ajouter l'annotation : **@EnableEurekaClient** ou bien **@EnableDiscoveryClient**

Accédez à l'interface de votre Eureka server : http://localhost:8761/

B. Partie 2 : Utiliser une Configuration Statique

En utilisant une configuration statique, vous n'avez pas besoin d'utiliser l'Eureka Server comme intermédiaire. L'API Gateway pointera directement vers le microservice.
Pour définir une API Gateway, il y'a deux méthodes de travail :

- **Méthode 1 : Ajouter la relation entre Gateway et MS à travers une classe de configuration**

Dans la classe main, nous allons créer des routes en ajoutant le code suivant :

```
@Bean
public RouteLocator gatewayRoutes(RouteLocatorBuilder builder){
    return builder.routes()
            .route("Candidat",r->r.path("/candidats/**")
                    .uri("http://localhost:8080") )
            .route("Job", r->r.path("/jobs/**")
                    .uri("http://localhost:8081") )
            .build();
}
```

- **Path** : c'est le path à utiliser pour accéder au microservice à travers le Gateway
- **uri** : c'est l'emplacement de microservice
- **id** : c'est l'id de MS-candidat

NB : dans le fichier **application.properties**, vous ajouter juste le port et le nom de l'application

- **Méthode 2 : Ajouter la relation entre Gateway et MS à travers le fichier application.properties**

Ajoutez ces lignes dans votre fichier **application.properties** :

```
spring.cloud.gateway.routes[0].id=CANDIDAT
spring.cloud.gateway.routes[0].uri=http://localhost:8082
spring.cloud.gateway.routes[0].predicates[0]=Path=/candidat/**
```

B. Partie 3 : Utiliser une configuration dynamique (avec Eureka Server)

Pour une configuration dynamique des microservices, nous devons utiliser **Eureka**. Ce service est particulièrement utile dans les cas suivants :

- Le microservice peut changer de port ou d'adresse IP (par exemple, dans un environnement cloud ou un cluster Kubernetes).
- Mise en place d'un **load balancing** automatique.
- Scalabilité facilitée en ajoutant plusieurs instances d'un microservice de manière transparente.

1. Pour définir un API Gateway dynamique, il y'a deux méthodes de travail :
Méthode 1 : travers une classe de configuration

```
@SpringBootApplication
@EnableDiscoveryClient
public class GatewayApplication {

    public static void main(String[] args) { SpringApplication.run(GatewayApplication.class, args)
    @Bean
    public RouteLocator gatewayRoutes(RouteLocatorBuilder builder){

        return builder.routes()
                .route( id: "Candidat", r->r.path("/candidat/**")
                        .uri("lb://candidat"))
                .build();
    }

}
```

Méthode 2 : travers le fichier application.properties

```
spring.application.name=gateway
server.port=8085
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
eureka.client.register-with-eureka=true
eureka.client.fetch-registry=true

# Configuration des routes via Eureka
 spring.cloud.gateway.routes[0].id=CANDIDAT
spring.cloud.gateway.routes[0].uri=lb://candidat # Utilisation de Eureka pour résoudre l'URL du service
 spring.cloud.gateway.routes[0].predicates[0]=Path=/candidat/**  # Rediriger les requêtes /candidats/** vers ce micro
```

2. **Visualiser le routage de l'API-Gateway : Activer les logs de routage dans API Gateway**
a . Ajouter ces lignes dans **application.properties** pour afficher les détails des requêtes redirigées par la Gateway :

```
logging.level.org.springframework.cloud.gateway=DEBUG
logging.level.reactor.netty.http.client=DEBUG
```

Effet : À chaque requête http envoyée, on verra dans la console **quelle instance Eureka a été sélectionnée**.

b. Consulter la partie console et identifier l'algorithme utilisé pour faire le load balancer.

3. Chercher les autres types de load balancing existants puis ajouter la configuration nécessaire pour tester l'algorithme **Random** en l'appliquant uniquement au service candidat

Quelle approche choisir : statique ou dynamique ?

Approche	Utilise Eureka ?	Avantages	Inconvénients
Dynamique Avec lb://candidat	✓ Oui	Pas besoin de connaître l'URL exacte du microservice (utile en production avec plusieurs instances). Facilite le **load balancing**.	Dépendance à Eureka. Temps de découverte initial.
Statique Avec RouteLocator uri("http://localhost:8082")	✗ Non	Simple à configurer et rapide. Fonctionne même sans Eureka.	Si l'IP/port du microservice change, la Gateway ne pourra plus le trouver.