Développement des applications Web basées sur les microservices (Partie 1)

Responsable du cours:
Dr. Ing. Mariam LAHAMI
Dr. Ing. Sihem LOUKIL
Email: mariem.lahami@enis.tn
sihem.loukil@enis.tn

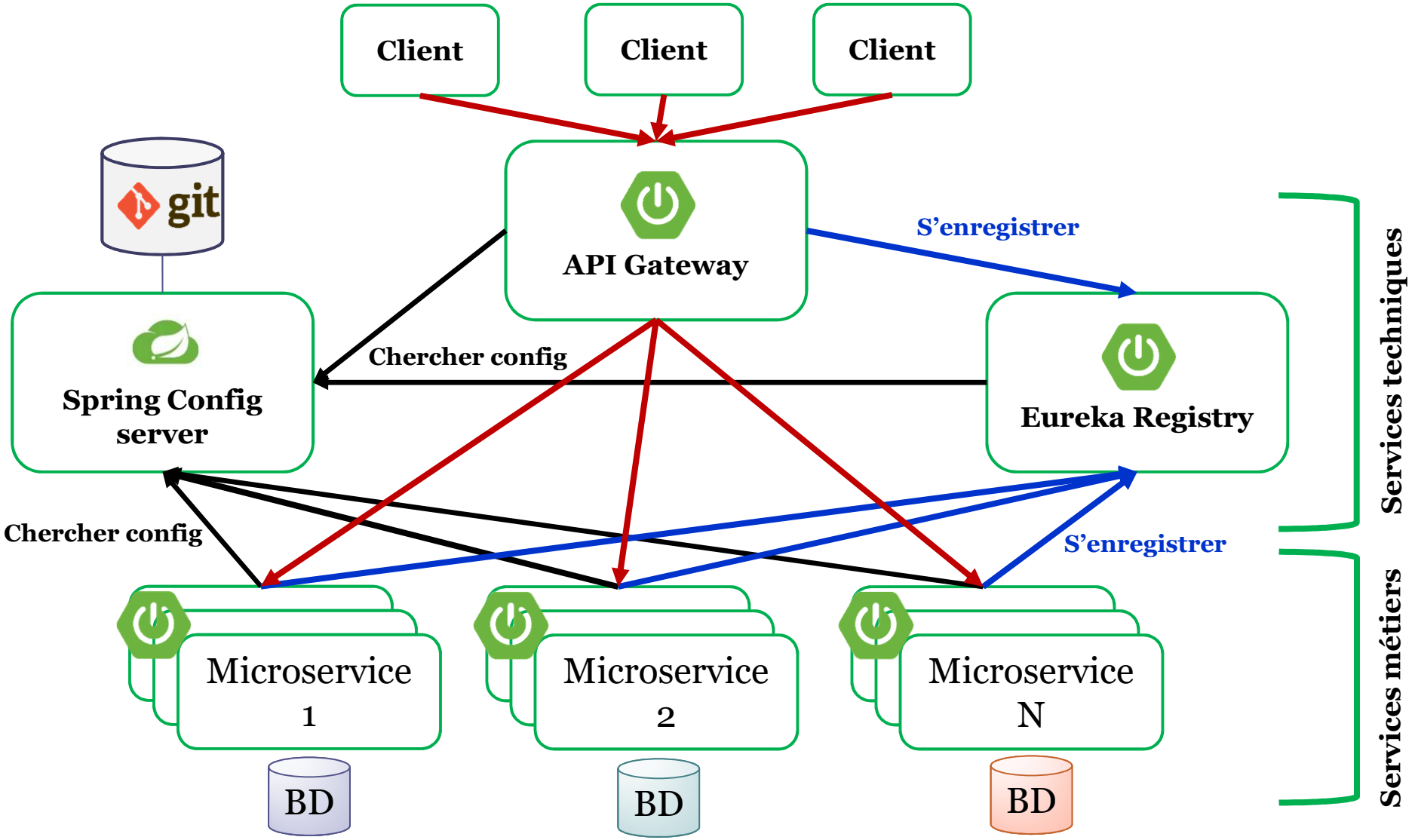
Mise en œuvre d'une architecture micro-services avec Spring Cloud

- Architecture
- Création du service de configuration
- Création du service d'enregistrement
- Création du service proxy

Objectif

- Concevoir des applications comme ensemble de microservices indépendants et déployables qui communiquent entre eux en utilisant :
  - Spring boot
  - Spring cloud

Architecture à mettre en place

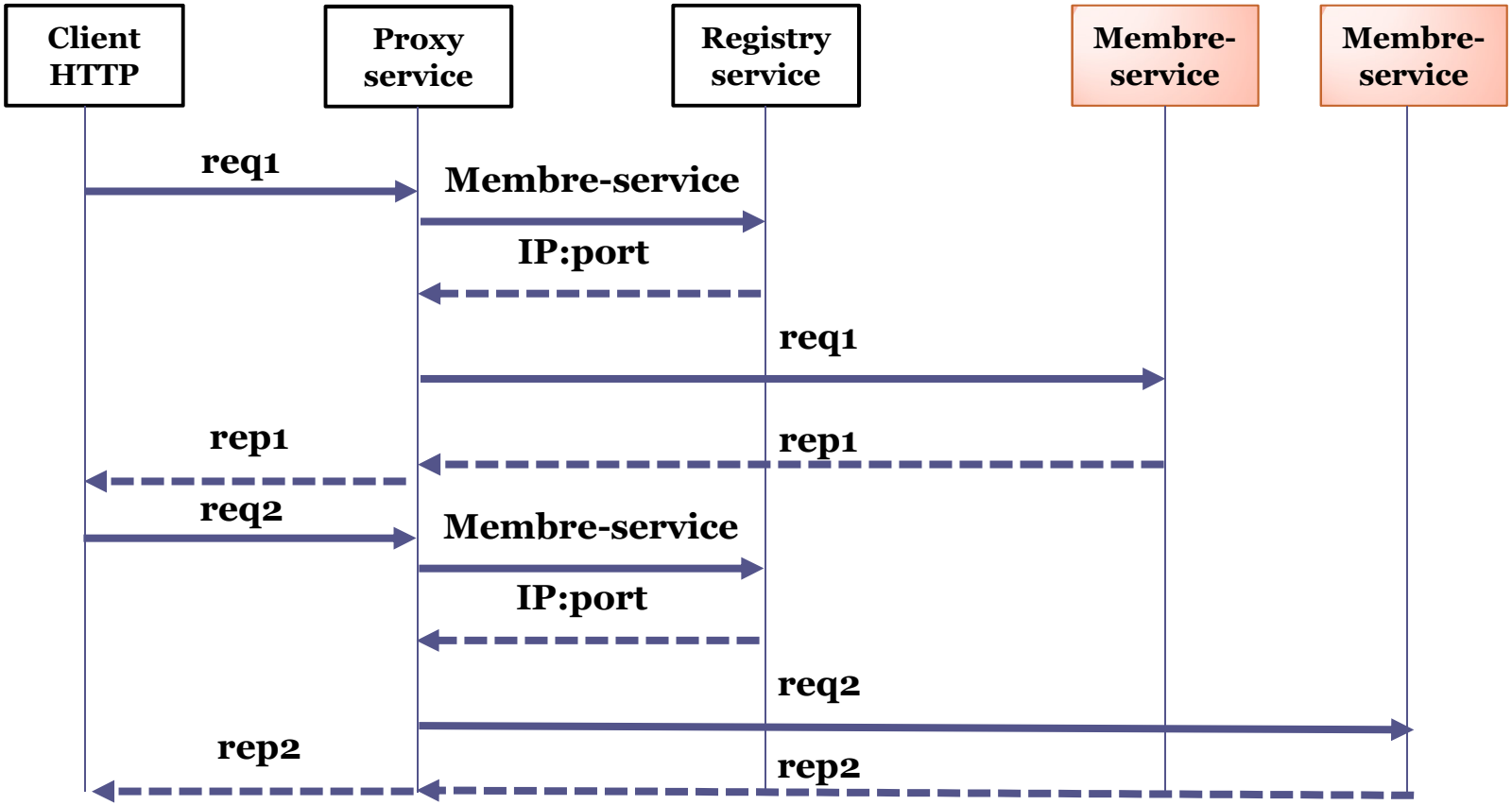


Architecture à mettre en place

- **Config service:** service de configuration ayant pour rôle de centraliser les fichiers de configurations des différents microservices dans un répertoire Git
- **Discovery/Registry service:** service permettant l'enregistrement des instances de services en vue d'être découvertes par d'autres services
- **API Gateway Service:** passerelle permettant de router les requêtes vers l'une des instances du service afin de gérer automatiquement la distribution de charge

Fonctionnement

- Soit la requête de type Get suivante : http://localhost:8080/membre-service/membres



Service de configuration

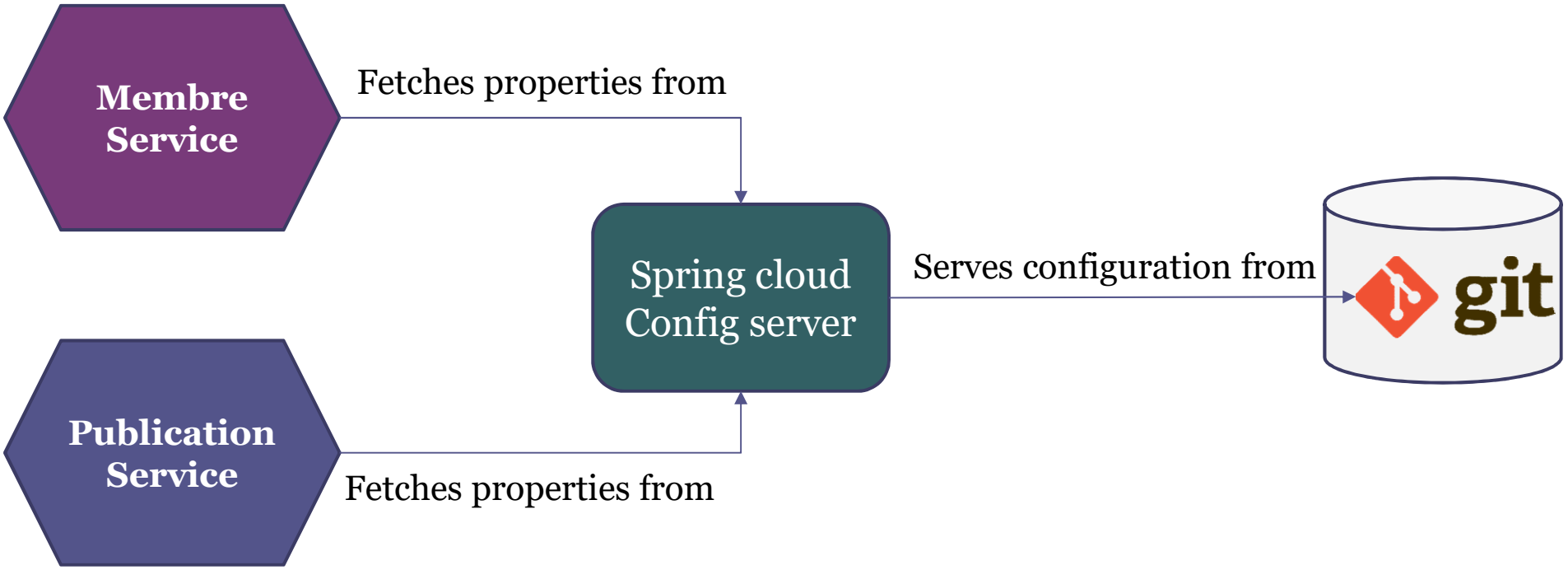
- Motivation : Plusieurs microservices s'exécutent en même temps avec des fichiers de configuration différents

⇒Externaliser ces fichiers de configuration dans un endroit centralisé afin de faciliter leur maintenance

- Outil : **Spring cloud config** offre un support côté serveur et côté client pour externaliser les fichiers de configuration de ces microservices

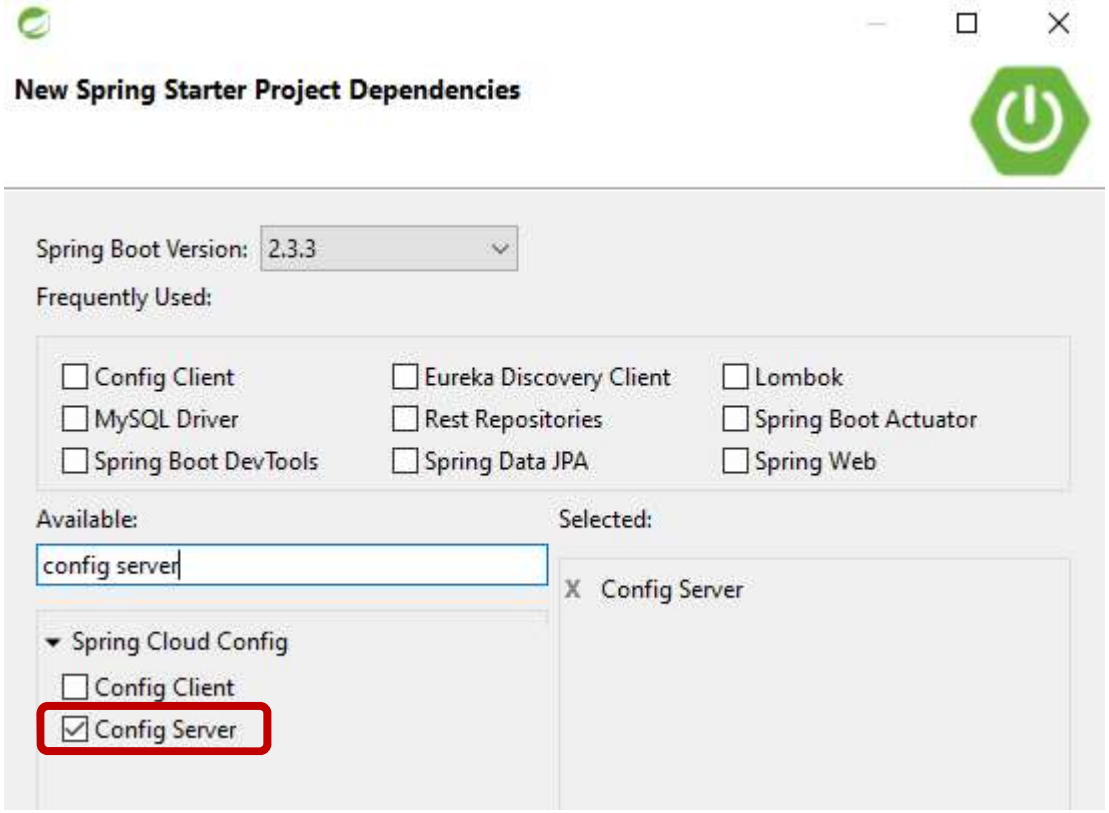
Service de configuration

- Rôle



Création du service de configuration

- Créer un nouveau projet Spring **ConfigService** en utilisant la dépendance



Création du service de configuration

- Pour activer et exposer ce service de configuration, utiliser l'annotation **@EnableConfigServer** dans la classe **ConfigServiceApplication**

```
@SpringBootApplication
@EnableConfigServer
public class ServiceConfig1Application {

public static void main(String[] args) {
SpringApplication.run(ServiceConfig1Application.class, args);
}
}
```

Paramétrage du service de configuration

- Ajouter dans son fichier **application.properties** les valeurs suivantes :

Spécifier le nom de la branche par défaut de GitHub

Représente le chemin absolu du répertoire contenant les fichiers de configuration des microservices de l'application

```
server.port=8888

spring.cloud.config.server.git.default-label=master

spring.cloud.config.server.git.uri=file:///${user.home}/myconfig

#spring.cloud.config.server.git.uri=file:./src/main/resources/myconfig
```

Indique que les fichiers de configuration se trouvent dans le répertoire src/main/resources/myconfig

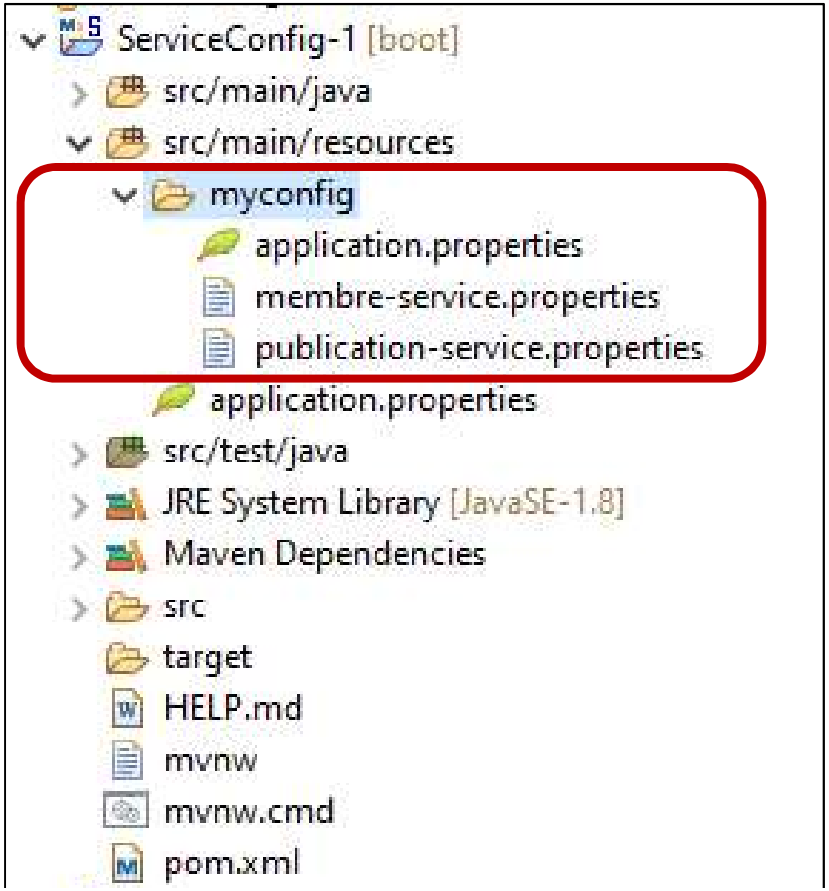
Paramétrage du service de configuration

- Le répertoire de configuration doit être un répertoire git :
  - Ouvrir le terminal et naviguer vers votre dossier myconfig
  - Taper la commande **git init** pour rendre le répertoire un répertoire git

- Créer dans ce dossier le fichier **application.properties** contenant la configuration en commun des différents microservices dans l'application

Paramétrage du service de configuration

- Pour chaque microservice dans l'application, il faut créer dans ce répertoire git son fichier de configuration



- Il faut aussi taper les commandes :
  - git add .
  - Pour avoir une première version de la configuration: git commit –m " first config"

Démarrage du Service de configuration

- Accéder à la configuration globale :

http://localhost:8888/application/master

- Accéder à la configuration d'un microservice bien déterminé

http://localhost:8888/membre-service/master

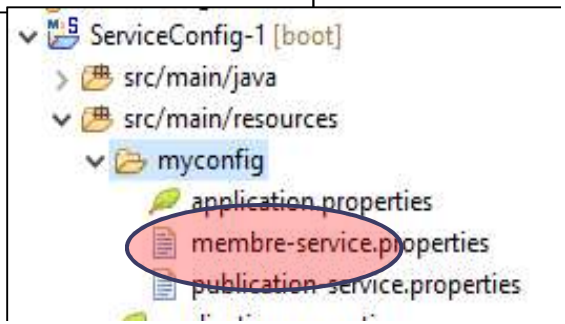
Centraliser la configuration du MS membre

- Ajouter la dépendance Config client au niveau du fichier POM.xml :

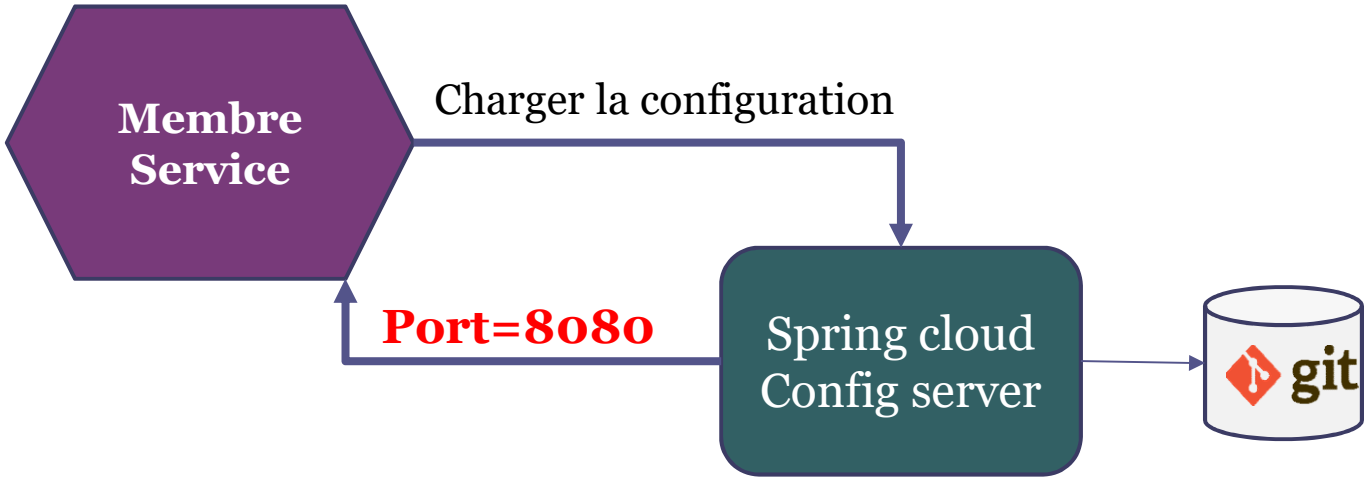
```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

- Dans le fichier application.properties du Microservice Membre, définir ces propriétes:

```
spring.application.name=membre-service
spring.config.import=configserver:http://localhost:8888
```

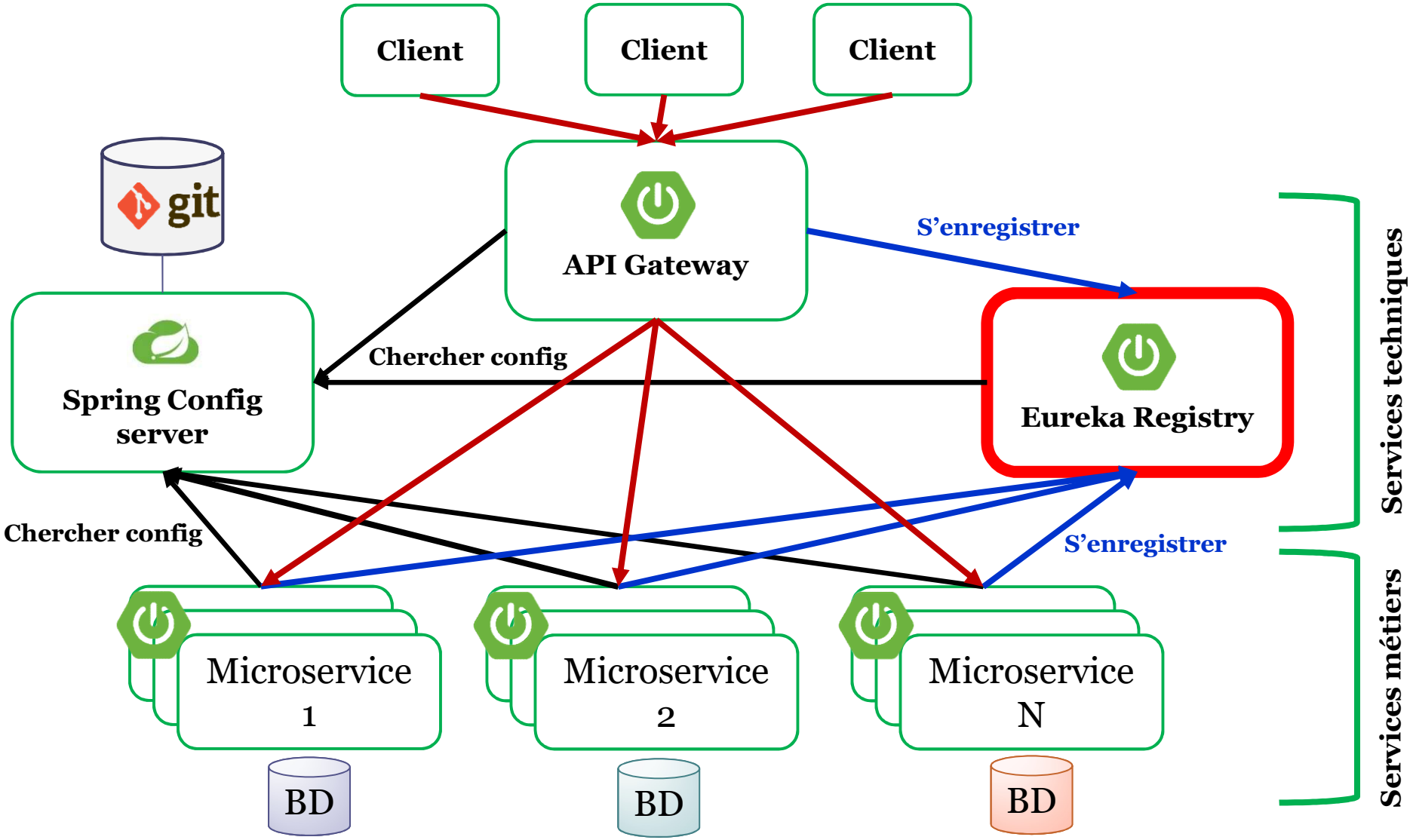


Centraliser la configuration du MS




```
Problems  Javadoc  Declaration  Console ×  Terminal
Membre-service2023 - MembreService2023Application [Spring Boot App]  [pid: 33512]
Hibernate: select e1_0.id,e1_0.cin,e1_0.cv,e1_0.date_naissance,e1_0.email,e1_0.nom,e1
Hibernate: select e1_0.id,e1_0.cin,e1_0.cv,e1_0.date_naissance,e1_0.email,e1_0.nom,e1
Hibernate: update membre set cin=?,cv=?,date_naissance=?,email=?,nom=?,password=?,pho
Hibernate: select m1_0.id,m1_0.type_mbr,m1_0.cin,m1_0.cv,m1_0.date_naissance,m1_0.ema
Hibernate: select e1_0.id,e1_0.cin,e1_0.cv,e1_0.date_naissance,e1_0.email,e1_0.nom,e1
Hibernate: select e1_0.id,e1_0.cin,e1_0.cv,e1_0.date_naissance,e1_0.email,e1_0.nom,e1
abid
Ayadi
```

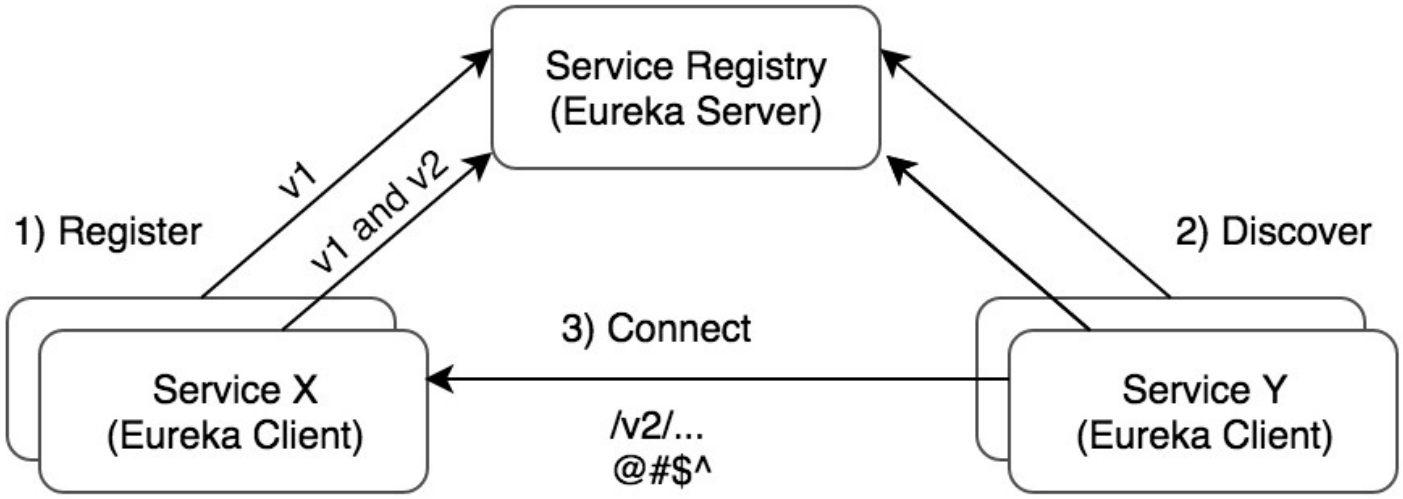
Architecture à mettre en place



Service d'enregistrement: Eureka

- Motivation: éviter le couplage fort entre microservices

- Principe :



Création du service d'enregistrement

- Créer un projet spring **registryService** avec les dépendances
  - Eureka server
  - Config client

- Spécifier les propriétés suivantes dans le fichier **application.properties**

```
spring.application.name=registry-service
spring.config.import=configserver:http://localhost:8888
```

Création du service d'enregistrement

- Dans le projet **service-config** et sous le répertoire **myconfig**, créer le fichier **registry-service.properties**

```
server.port=8761

eureka.client.fetch-registry=false

eureka.client.register-with-eureka=false
```

Indiquer à Eureka server de ne pas enregistrer lui même

Indiquer à Eureka server de ne pas enregistrer lui-même comme étant client eureka

Création du service d'enregistrement

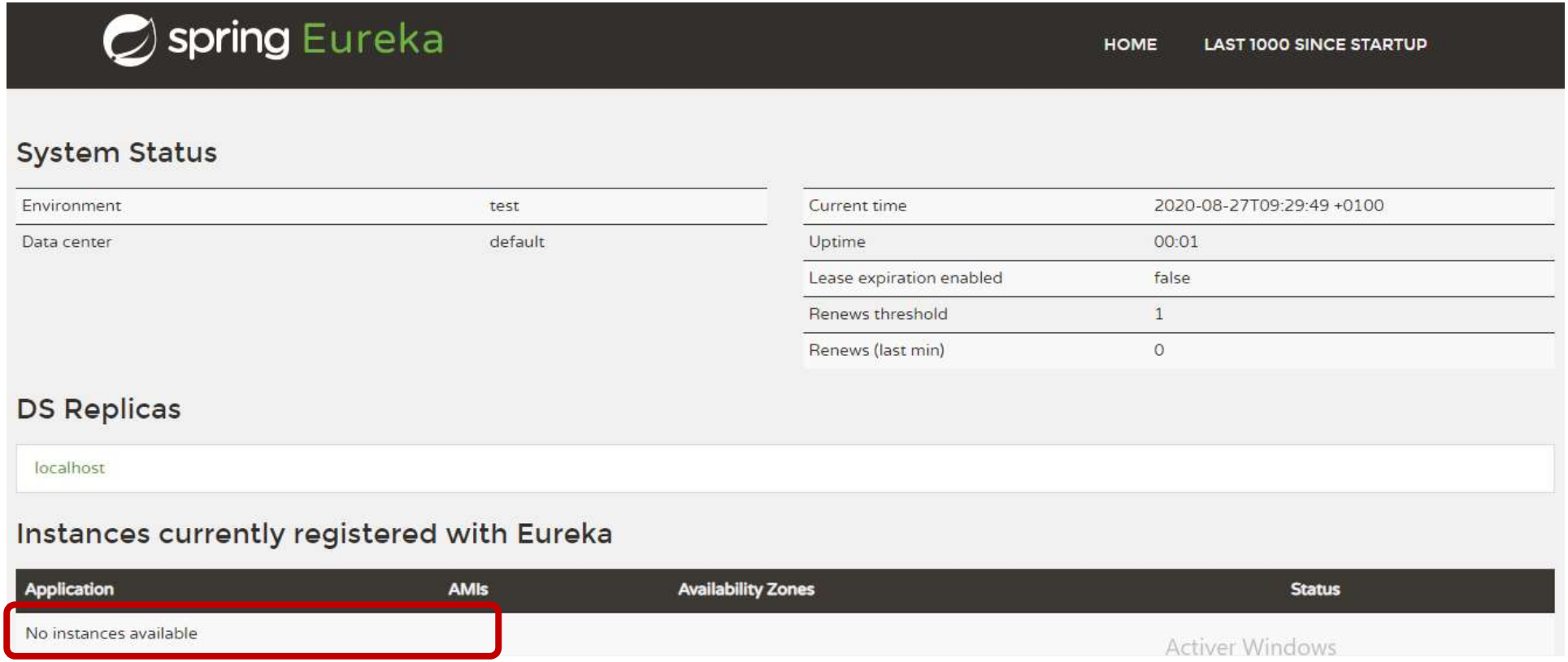
- Ajouter l'annotation **@EnableEurekaServer** dans la classe principale du service registryService

```
@SpringBootApplication
@EnableEurekaServer
public class RegistryService1Application {

public static void main(String[] args) {
SpringApplication.run(RegistryService1Application.c
lass, args);
}
}
```

Démarrage du service d'enregistrement

- Taper http://localhost:8761/



Enregistrer le MS Membre dans Eureka

- Dans le pom.xml du MS Membre, ajouter la dépendance suivante:

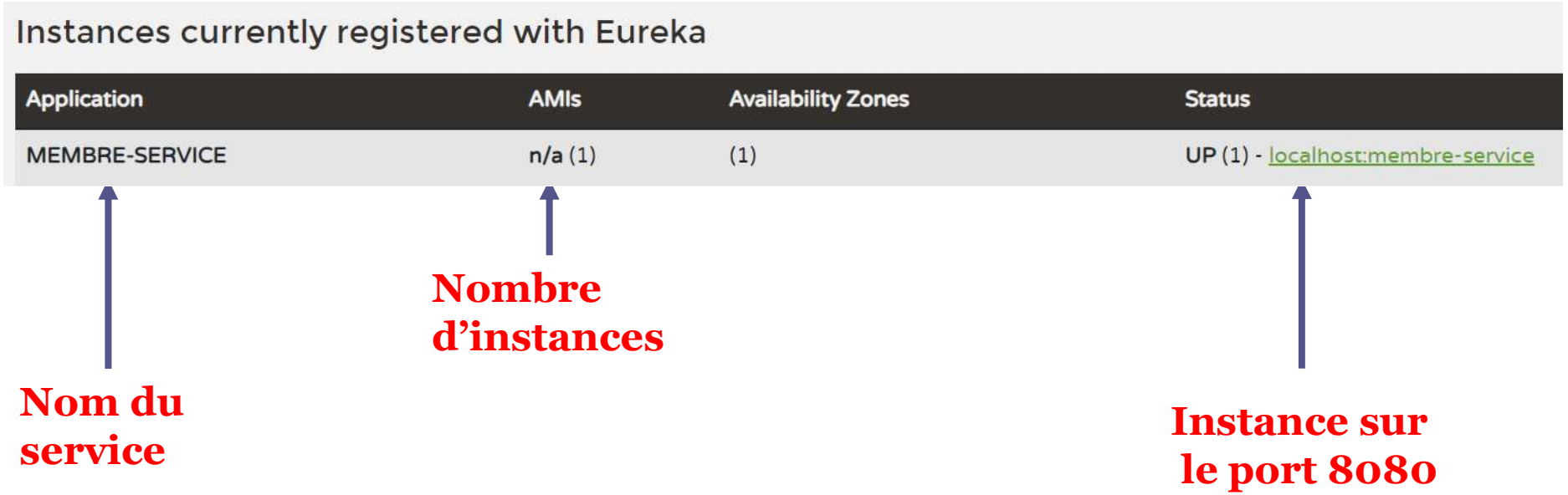
```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

- Dans son fichier bootstrap.properties , spécifier le port

- Dans la classe principale, ajouter l'annotation `@EnableDiscoveryClient`

Enregistrer le MS Membre dans Eureka

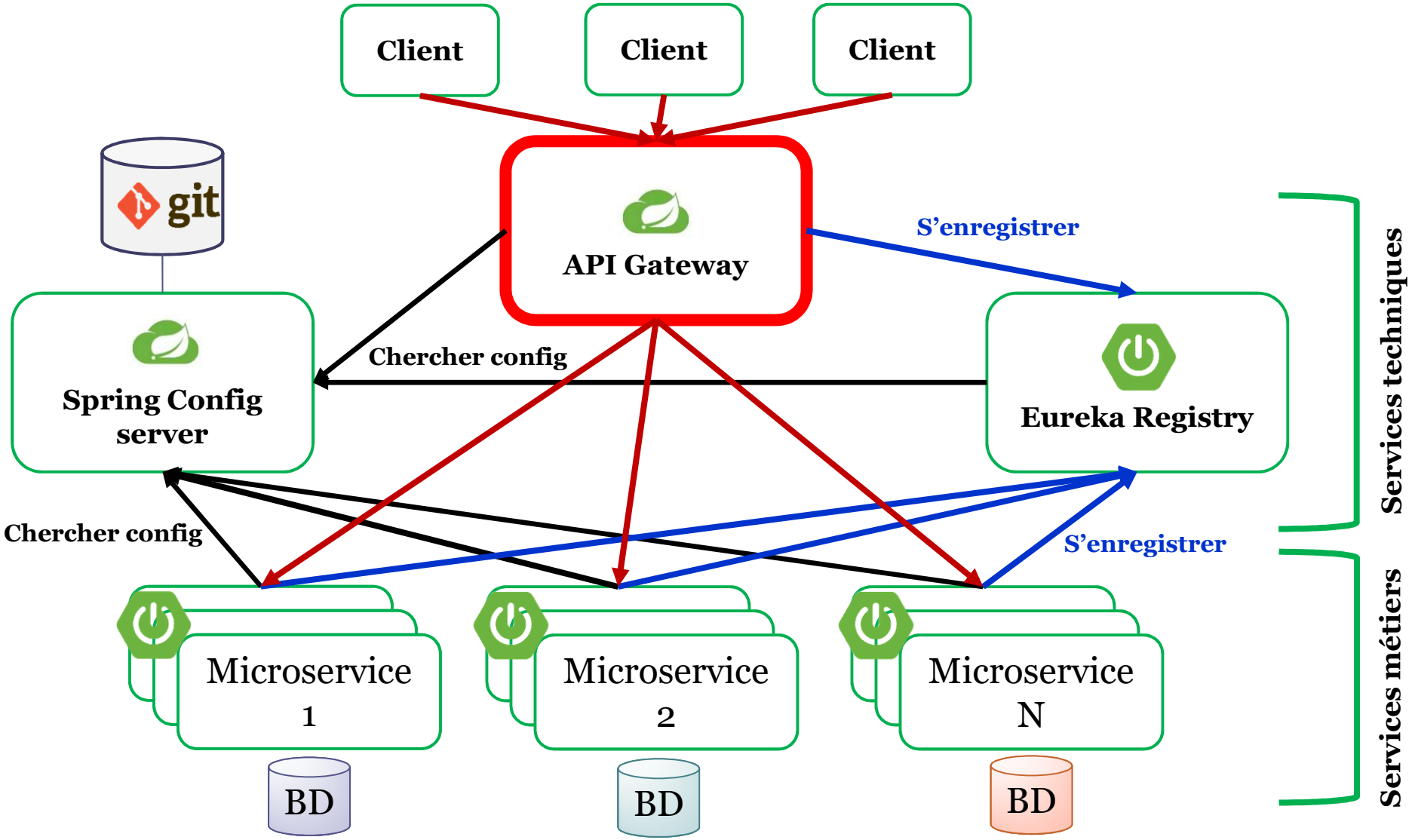
- Démarrer une instance



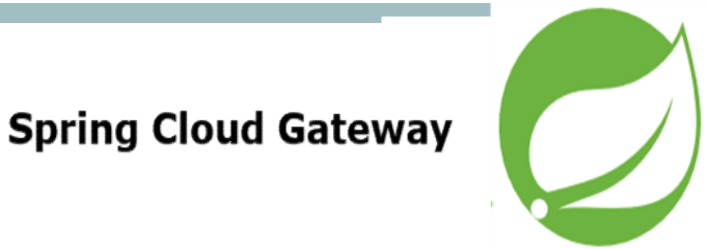
Enregistrer le MS Membre dans Eureka

- Démarrer plusieurs instances avec différents ports

Architecture à mettre en place



API Gateway



- C'est un service offrant des fonctionnalités avancées y compris l'orchestration, la surveillance et le monitoring
  - Exemples d'API Gateway : Netflix Zuul, Amazon Gateway API, Spring cloud gateway, etc.
- Spring cloud gateway est un proxy utilisant une API non bloquante
  - Un thread est toujours disponible pour traiter la requête entrante
  - Les requêtes sont traitées d'une manière asynchrone en arrière plan et une fois terminée la réponse est envoyée
  - Aucune requête n'est bloquée sauf si les ressources CPU et RAM sont saturées

Création du service Gateway

- Créer un projet spring avec ces dépendances
  - Gateway
  - Eureka client

- Ajouter l' annotation dans la classe principale

```
@SpringBootApplication
@EnableDiscoveryClient
public class GatewayApplication {

public static void main(String[] args) {
SpringApplication.run(GatewayApplication.class, args);
}
```

Configuration et démarrage du service Gateway

- Dans le fichier application.properties

```
server.port=9000
spring.application.name=gateway-service
spring.cloud.discovery.enabled=true
```

- Après démarrage du service Gateway

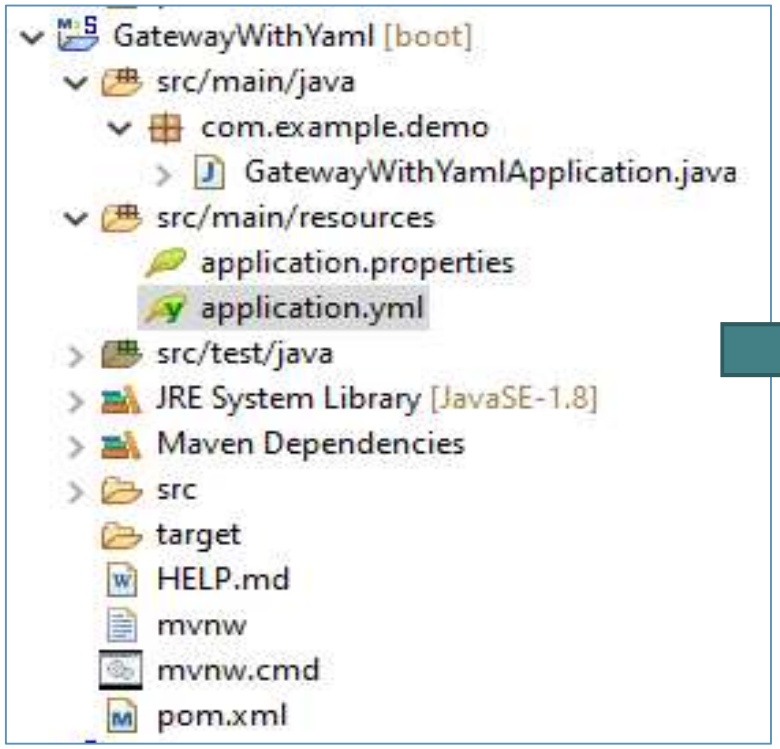
```
d.netty.NettyWebServer  : Netty started on port 9000
AutoServiceRegistration : Updating port to 9000
GatewayApplication      : Started GatewayApplication
```

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
GATEWAY-SERVICE	n/a (1)	(1)	UP (1) - 192.168.1.8:gateway-service:9000
MEMBRE-SERVICE	n/a (2)	(2)	UP (2) - 192.168.1.8:membre-service:8082 , 192.168.1.8:membre-service:8081

Routage des requêtes avec Yaml file

- Dans le dossier resources, créer un fichier application.yml contenant la configuration des routes



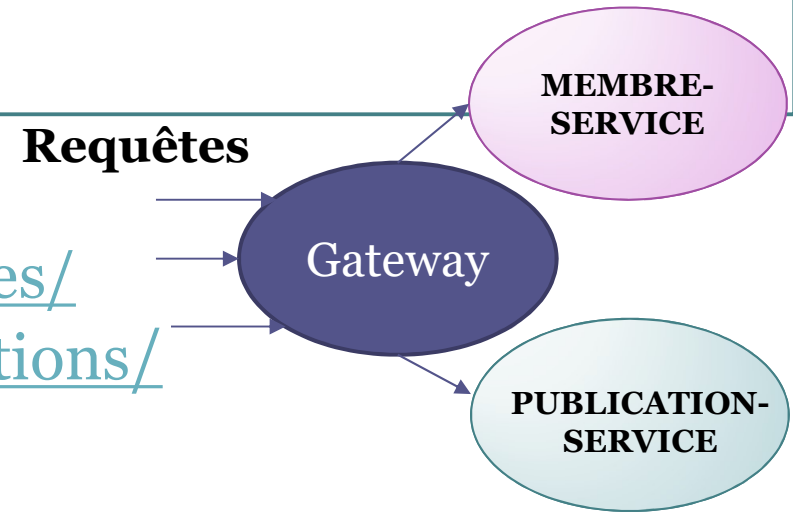

```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: r1
        uri: http://localhost:8081
        predicates:
        - Path=/membres/**
      - id: r2
        uri: http://localhost:8082
        predicates:
        - Path=/publications/**
```

Routage static des requêtes avec code Java

- Routage statique via une classe de configuration **sans le service Registry**

```
@Bean
RouteLocator routeLocator(RouteLocatorBuilder builder)
{
return builder.routes()
.route(r->r.path("/membres/**").uri("http://localhost:8081"))
.route(r->r.path("/publications/**").uri("http://localhost:8082"))
.build();
}
```

- Il suffit de taper ces URLs
  - http://localhost:9000/membres/
  - http://localhost:9000/publications/



Routage static des requêtes avec code Java

- Routage statique via une classe de configuration **avec le service Registry**

```
@Bean
RouteLocator routeLocator(RouteLocatorBuilder builder)
{
return builder.routes()
.route(r->r.path("/membres/**").uri("lb://MEMBRE-SERVICE"))
.route(r->r.path("/publications/**").uri("lb://PUBLICATION-SERVICE"))
.build();
}
```

- Il suffit de taper ces URLs
  - http://localhost:9000/membres/
  - http://localhost:9000/publications/

Routage des requêtes avec code Java

- Routage dynamique

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

- Il suffit de taper les URLs
  - http://localhost:9000/MEMBRE-SERVICE/membres
  - http://localhost:9000/PUBLICATION-SERVICE/publications

Load Balancing

- Suite à la création des 2 instances de Publication-service, nous pouvons visualiser le passage d'une instance à une autre

Req 1: http://localhost:9000/PUBLICATION-SERVICE/publications

Req 2: http://localhost:9000/PUBLICATION-SERVICE/publications