

2.6 Activité pratique : Architecture Micro-services avec RestTemplate

Cette activité consiste à développer une application basée sur une architecture microservices pour la gestion des clients et des voitures. La communication entre les microservices est assurée à l'aide de `RestTemplate`, et les fichiers de configuration au format YAML remplacent les fichiers de type `properties`.

A - Mise en place du service discovery Eureka

La mise en place du service discovery Eureka avec une configuration YAML implique les étapes suivantes :

Étape 1 : Création d'un projet Spring Boot

Utiliser Spring Initializr pour créer un nouveau projet Spring Boot. Sélectionner les dépendances essentielles, y compris Eureka Server.

Étape 2 : Configuration du fichier `application.yml`

Intégrer le contenu suivant dans le fichier `src/main/resources/application.yml` du projet. Ce fichier doit inclure les propriétés suivantes :

```
server:
  port: 8761

eureka:
  client:
    register-with-eureka: false
    fetch-registry: false

logging:
  level:
    com.netflix.eureka: OFF
    com.netflix.discovery: OFF
```

Ces paramètres configurent le port du serveur, désactivent l'enregistrement automatique du serveur Eureka en tant que client et réduisent le niveau de journalisation pour certaines classes spécifiques.

Étape 3 : Configuration de la classe principale

Il est nécessaire de vérifier que la classe principale de l'application (généralement annotée avec `@SpringBootApplication`) inclut l'annotation `@EnableEurekaServer`. Cette annotation permet de configurer l'application en tant que serveur Eureka.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@EnableEurekaServer
@SpringBootApplication
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }
}
```

Cette classe définit le point d'entrée de l'application Spring Boot et active le serveur Eureka à l'aide de l'annotation `@EnableEurekaServer`.

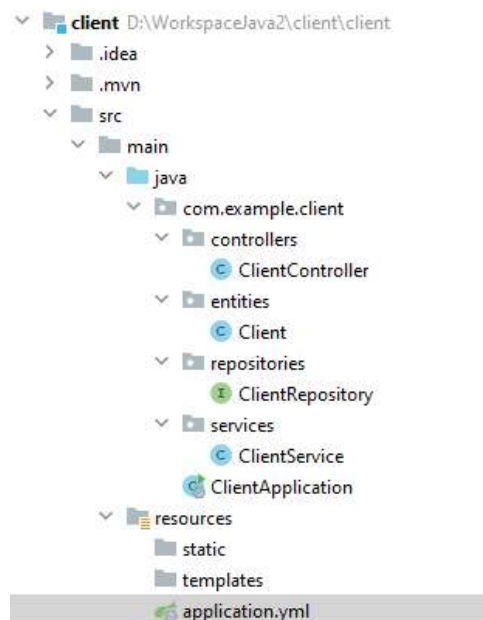
Étape 4 : Exécution de l'application

Lancer l'application. Après le démarrage, accéder à l'interface utilisateur Eureka en ouvrant un navigateur web et en naviguant à l'adresse `http://localhost:8761`.

Cette configuration met en place un serveur Eureka, conforme aux paramètres du fichier YAML. L'application agit comme un serveur dédié à la gestion des services et n'est pas enregistrée en tant que client Eureka. Cette approche permet d'éviter l'enregistrement automatique du serveur lui-même au sein du registre. Il est possible de personnaliser davantage la configuration selon les besoins du projet.

B - Microservice Client

Le microservice Client est une application Spring Boot qui gère les opérations associées à la gestion des clients. Ce service repose sur une base de données MySQL pour le stockage des informations essentielles des clients, telles que le nom et l'âge. Des points de terminaison RESTful sont exposés afin de permettre la récupération de la liste des clients, la recherche d'un client par identifiant, ainsi que l'ajout de nouveaux clients. L'intégration au service de découverte Eureka garantit la découverte automatique des services dans un environnement de type microservices.



Étape 1 : Configurer le service Spring Boot

1. Création d'un projet Spring Boot à l'aide de Spring Initializer en sélectionnant les dépendances requises, notamment Spring Web et Spring Data JPA .
2. Configuration du fichier `application.yml` de la manière suivante :

```

server:
  port: 8081

spring:
  application:
    name: SERVICE-CLIENT
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url:
      jdbc:mysql://localhost:3306/client servicedb?createDatabaseIfNotExist=true
    username: "root"
    password: ""
  jpa:

```

```
hibernate:
  ddl-auto: update
  show-sql: true
```

Étape 2 : Créer l'entité Client

1. Création d'une classe Java nommée Client dans le package `com.example.client.entities`.
2. Annotation de la classe avec JPA pour définir l'entité, y compris l'identifiant auto-généré.
3. Déclaration des propriétés `id`, `nom` et `age` avec les annotations JPA nécessaires.

```
package com.example.client.entities;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Client {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nom;
    private Float age;
}
```

Étape 3 : Créer le repository JPA

1. Création d'une interface Java nommée ClientRepository dans le package `com.example.client.repositories`.
2. Extension de l'interface JpaRepository en spécifiant les types Client et Long.

```
package com.example.client.repositories;

import com.example.client.entities.Client;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface ClientRepository extends JpaRepository<Client, Long> {}
```

Étape 4 : Créer le service

1. Création d'une classe Java nommée ClientService dans le package `com.example.client.services`.
2. Annotation de la classe avec `@Service` pour indiquer qu'il s'agit d'un service Spring.
3. Injection de ClientRepository dans la classe et mise en œuvre des méthodes nécessaires.

```
package com.example.client.services;

import com.example.client.entities.Client;
import com.example.client.repositories.ClientRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class ClientService {

    @Autowired
    private ClientRepository clientRepository;

    public List<Client> findAll() {
        return clientRepository.findAll();
    }

    public Client findById(Long id) throws Exception {
        return clientRepository.findById(id).orElseThrow(() -> new
            Exception("Invalid Client ID"));
    }

    public void addClient(Client client) {
        clientRepository.save(client);
    }
}
```

Étape 5 : Créer le contrôleur REST

1. Création d'une classe Java nommée ClientController dans le package com.example.client.controllers.
2. Annotation de la classe avec @RestController pour indiquer qu'il s'agit d'un contrôleur REST.
3. Définition des points de terminaison (/api/client) à l'aide des annotations @GetMapping, @PostMapping, etc.

```
package com.example.client.controllers;

import com.example.client.entities.Client;
import com.example.client.services.ClientService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("api/client")
public class ClientController {

    @Autowired
    private ClientService service;
```

```

@GetMapping
public List<Client> findAll() {
    return service.findAll();
}

@GetMapping("/{id}")
public Client findById(@PathVariable Long id) throws Exception {
    return service.findById(id);
}

@PostMapping
public void save(@RequestBody Client client) {
    service.addClient(client);
}
}

```

C - Service Gateway

Le service Gateway assure la gestion centralisée des requêtes en utilisant Spring Cloud Gateway. Ce service permet la redirection des requêtes vers les microservices enregistrés dans le registre Eureka.

Étape 1 : Ajouter les dépendances

Pour garantir la présence des dépendances nécessaires, il convient d'ajouter les dépendances suivantes au fichier build.gradle (pour Gradle) ou pom.xml (pour Maven) :

Gradle:

```

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'org.springframework.cloud:spring-cloud-starter-gateway'
    implementation
        'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client'
}

```

Maven:

```

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <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>
</dependencies>

```

Étape 2 : Configurer les propriétés de l'application

Le fichier application.properties ou application.yml doit être configuré avec les paramètres Eureka et Gateway comme suit :

```

server:
  port: 8888

spring:
  application:
    name: Gateway
  cloud:
    discovery:
      enabled: true

eureka:
  instance:
    hostname: localhost

```

- R** La configuration suivante permet de paramétrer le service Gateway de manière à assurer son interaction avec le serveur Eureka et à activer la découverte des services.
- `server.port: 8888` : Ce paramètre définit le port sur lequel l'application Gateway est accessible. Ici, le port 8888 est utilisé.
 - `spring.application.name: Gateway` : Ce paramètre spécifie le nom de l'application. Ce nom est essentiel pour identifier le service Gateway au sein du registre Eureka.
 - `spring.cloud.discovery.enabled: true` : Ce paramètre active la fonctionnalité de découverte des services. Il permet au Gateway de détecter automatiquement les services enregistrés dans le serveur Eureka.
 - `eureka.instance.hostname: localhost` : Ce paramètre indique l'hôte où se trouve le serveur Eureka. Ici, le serveur Eureka est hébergé localement sur localhost.
- Cette configuration assure que le service Gateway peut interagir avec Eureka pour obtenir les informations sur les services disponibles, ce qui facilite la redirection dynamique des requêtes vers les microservices.

Étape 3 : Créer la classe d'application Gateway

```

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.ReactiveDiscoveryClient;
import
    org.springframework.cloud.gateway.discovery.DiscoveryClientRouteDefinitionLocator;
import org.springframework.cloud.gateway.discovery.DiscoveryLocatorProperties;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class GatewayApplication {

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

    @Bean
    public DiscoveryClientRouteDefinitionLocator routesDynamic(
        ReactiveDiscoveryClient reactiveDiscoveryClient,
        DiscoveryLocatorProperties discoveryLocatorProperties) {
        return new DiscoveryClientRouteDefinitionLocator(reactiveDiscoveryClient,
            discoveryLocatorProperties);
    }
}

```

R La classe `GatewayApplication` constitue le point d'entrée principal de l'application Spring Boot. Voici une explication des éléments essentiels de cette classe :

- `@SpringBootApplication` : Cette annotation déclare la classe comme une application Spring Boot. Elle active la configuration automatique, le balayage des composants (component scan) et d'autres fonctionnalités.
- `public static void main(String[] args)` : Cette méthode principale (main) lance l'application en utilisant la méthode `SpringApplication.run()`, qui initialise le contexte Spring Boot.
- `@Bean public DiscoveryClientRouteDefinitionLocator routesDynamic` : Cette méthode déclare un composant de type `DiscoveryClientRouteDefinitionLocator`. Ce composant permet de créer dynamiquement des routes Gateway à partir des services découverts via le `ReactiveDiscoveryClient`.
- **Paramètres de la méthode `routesDynamic`** :
 - `ReactiveDiscoveryClient reactiveDiscoveryClient` : Ce client réactif interagit avec Eureka pour obtenir la liste des services disponibles.
 - `DiscoveryLocatorProperties discoveryLocatorProperties` : Ce composant contient les paramètres de configuration permettant de définir la manière dont les routes doivent être découvertes.
- **Retour de la méthode `routesDynamic`** : La méthode renvoie une instance de `DiscoveryClientRouteDefinitionLocator`, ce qui permet de définir des routes dynamiques basées sur les services enregistrés dans le registre Eureka.

Cette classe permet de démarrer l'application Gateway et de créer des routes dynamiques pour rediriger les requêtes entrantes vers les services découverts dans l'environnement microservices.

Étape 4 : Exécuter l'application

L'exécution de l'application Spring Cloud Gateway permet la création d'un localisateur de routes dynamique. La méthode `@Bean routesDynamic` découvre automatiquement les routes depuis le serveur Eureka.

Étape 5 : Tester la découverte dynamique des routes

1. **Lancement des services** :
 - Lancer le serveur Eureka.
 - Démarrer le service `SERVICE-CLIENT`.
2. **Vérification du tableau de bord Eureka** :
 - Accéder au tableau de bord Eureka à l'adresse `http://localhost:8761`.
 - Vérifier que le service `SERVICE-CLIENT` est enregistré avec succès.
3. **Lancement de la passerelle** :
 - Démarrer l'application Spring Cloud Gateway.
4. **Test de l'accès au service `SERVICE-CLIENT`** :
 - Utiliser un client HTTP ou un navigateur pour accéder au service `SERVICE-CLIENT` via la passerelle : `http://localhost:8888/SERVICE-CLIENT/api/client`.



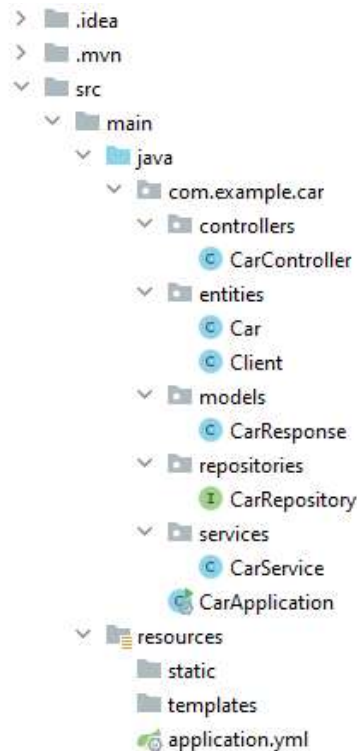
```

1 [
2   {
3     "id": 1,
4     "nom": "Amine SAFI",
5     "age": 23
6   },
7   {
8     "id": 2,
9     "nom": "Amal ALAOUI",
10    "age": 22
11  },
12  {
13    "id": 3,
14    "nom": "Samir RAMI",
15    "age": 22
16  }
17 ]

```

D - Service Voiture

Le service Voiture assure la gestion des informations sur les voitures. Les principales opérations incluent la récupération de toutes les voitures, la recherche d'une voiture par identifiant et l'affichage des détails de la voiture ainsi que ceux du client associé. La structure du projet est présentée ci-dessous :



Ètape 1 : Configuration de la base de données

Une instance MySQL doit être exécutée localement sur le port 3306. La base de données carservicedb est créée. La configuration des paramètres de connexion est définie dans le fichier application.yml.

```

server:
  port: 8082

spring:
  application:
    name: SERVICE-CAR
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/carservicedb?createDatabaseIfNotExist=true
    username: "root"
    password: ""
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true

```

Ètape 2 : Création des entités et des repositories

Les entités Car et Client ainsi que le repository CarRepository sont créés dans les packages correspondants.

@Entity


```

@Data
@AllArgsConstructor
@NoArgsConstructor
public class Car {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String brand;
    private String model;
    private String matricule;
    private Long client_id;
}

```

```

@Data
@AllArgsConstructor
@NoArgsConstructor
public class Client {
    private Long id;
    private String name;
    private Integer age;
}

```

```

@Repository
public interface CarRepository extends JpaRepository<Car, Long> {}

```

Étape 3 : Création des modèles

Le modèle CarResponse est créé dans le package `com.example.car.models`.

```

@Builder
@Data
@AllArgsConstructor
@NoArgsConstructor
public class CarResponse {
    private Long id;
    private String brand;
    private String model;
    private String matricule;
    private Client client;
}

```

Étape 4 : Configuration du Service REST

Dans la classe `CarService`, la communication avec le service client est configurée via `RestTemplate`.

```

@SpringBootApplication
public class CarApplication {
    public static void main(String[] args) {
        SpringApplication.run(CarApplication.class, args);
    }

    @Bean
    public RestTemplate restTemplate() {

```

```

    RestTemplate restTemplate = new RestTemplate();
    SimpleClientHttpRequestFactory requestFactory = new
        SimpleClientHttpRequestFactory();
    requestFactory.setConnectTimeout(5000);
    requestFactory.setReadTimeout(5000);
    restTemplate.setRequestFactory(requestFactory);
    return restTemplate;
}
}

```

Ètape 5 : Création des contrôleurs

Les contrôleurs pour récupérer toutes les voitures (findAll) et récupérer une voiture par identifiant (findById) sont définis dans la classe CarController.

```

@RestController
@RequestMapping("api/car")
public class CarController {
    @Autowired
    private CarService carService;

    @GetMapping
    public List<CarResponse> findAll() {
        return carService.findAll();
    }

    @GetMapping("/{id}")
    public CarResponse findById(@PathVariable Long id) throws Exception {
        return carService.findById(id);
    }
}

```

Ètape 6 : Lancement de l'application

L'application est exécutée en utilisant la classe CarApplication et sa méthode main. Aucune erreur ne doit être présente au démarrage.

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
GATEWAY	n/a (1)	(1)	UP (1) - DESKTOP-AT134HQ:Gateway:8888
SERVICE-CAR	n/a (1)	(1)	UP (1) - DESKTOP-AT134HQ:SERVICE-CAR:8082
SERVICE-CLIENT	n/a (1)	(1)	UP (1) - DESKTOP-AT134HQ:SERVICE-CLIENT:8081

Ètape 7 : Test des endpoints

Des outils comme Advanced Rest Client ou curl permettent de tester les endpoints du microservice. Les requêtes GET permettent de récupérer toutes les voitures et une voiture par identifiant.




```

{
  "id": 1,
  "brand": "Ford",
  "model": "2022",
  "matricule": "12 A 2345",
  "client": {
    "id": 1,
    "name": null,
    "age": 23
  }
}

```

La liste des voitures peut être affichée comme suit :



```

[
  {
    "id": 1,
    "brand": "Ford",
    "model": "2022",
    "matricule": "12 A 2345",
    "client": {
      "id": 1,
      "nom": "Amine SAFI",
      "age": 23
    }
  },
  {
    "id": 2,
    "brand": "Renaut ",
    "model": "2000",
    "matricule": "14 R 5245",
    "client": {
      "id": 2,
      "nom": "Amal ALAOUI",
      "age": 22
    }
  },
  {
    "id": 3,
    "brand": "Toyota",
    "model": "1990",
    "matricule": "34 T 6755",
    "client": {
      "id": 3,
      "nom": "Samir RAMI",
      "age": 22
    }
  },
  {
    "id": 4,
    "brand": "Ford",
    "model": "2021",
    "matricule": "44 R 6756",
    "client": {
      "id": 2,
      "nom": "Amal ALAOUI",
      "age": 22
    }
  }
]

```