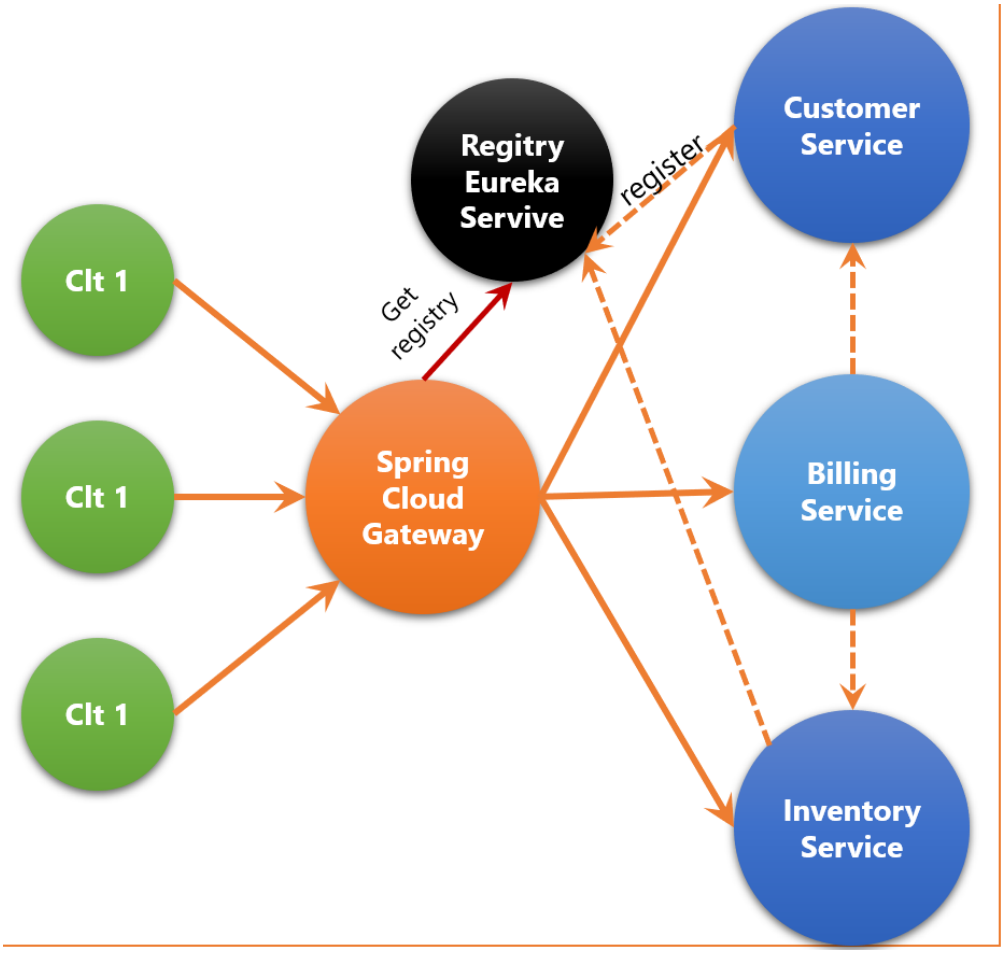
Prénom & nom : Fahd KORAICHE
Groupe : 5IIRG2C-C

Rapport Architectures micro-services

I – Introduction :

Le projet va consister a la réalisation de plusieurs micro-services ainsi qu'un service d'enregistrement et un service gateway.



Prénom & nom : Fahd KORAICHE
Groupe : 5IIRG2C-C

II – Procédure :

1. Créer le micro service Customer-service
    - Créer l'entité Customer
    - Créer l'interface CustomerRepository basée sur Spring Data
    - Déployer l'API Restful du micro-service en utilisant Spring Data Rest
    - Tester le Micro service
2. Créer le micro service Inventory-service
    - Créer l'entité Product
    - Créer l'interface ProductRepository basée sur Spring Data
    - Déployer l'API Restful du micro-service en utilisant Spring Data Rest
    - Tester le Micro service
3. Créer la Gateway service en utilisant Spring Cloud Gateway
    - Tester la Service proxy en utilisant une configuration Statique basée sur le fichier application.yml
    - Tester la Service proxy en utilisant une configuration Statique basée une configuration Java
4. Créer l'annuaire Registry Service basé sur NetFlix Eureka Server

5. Tester le proxy en utilisant une configuration dynamique de Gestion des routes vers les micro services enregistrés dans l'annuaire Eureka Server
6. Créer Le service Billing-Service en utilisant Open Feign pour communiquer avec les services Customer-service et Inventory-service
7. Créer un client Angular qui permet d'afficher une facture

III – Customer service :

Ce micro-service prend en charge toutes les opérations relatives aux données client, il utilise sa propre base de données, ainsi qu'il s'enregistre dans le service d'enregistrement eurekaDiscovery-service :

```
spring.cloud.discovery.enabled = true
```

Classe :

```
    @Entity @Data
@NoArgsConstructor
@AllArgsConstructor @ToString
public class Customer {
    @Id @GeneratedValue(strategy =
GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String email;
}
```

Repository :

```
@RepositoryRestResource
public interface CustomerRepository
extends JpaRepository<Customer, Long>
{}
```

IV – Inventory service :

Ce micro-service prend en charge toutes les opérations relatives aux données des produits, il utilise sa propre base de données, ainsi qu'il s'enregistre dans le service d'enregistrement eurekaDiscovery-service :

```
spring.cloud.discovery.enabled = true
```

Classe :

```
@Entity @Data @NoArgsConstructor
@AllArgsConstructor @ToString
public class Product {
    @Id @GeneratedValue(strategy =
GenerationType.IDENTITY)
    private Long id;
    private String name;
    private double price;
    private int quantity;
}
```

Repository :

```
@RepositoryRestResource
public interface ProductRepository
extends JpaRepository<Product, Long>
{
}
```

V – Gateway service (spring cloud gateway) :

Ce service doit, comme les autres s'inscrire dans le service d'enregistrement.

```
spring.cloud.discovery.enabled = true
```

-Configuration statique :

1 - En utilisant application.yml :

```
spring:
  cloud:
    gateway:
      routes:
        - id: r1
          uri: http://localhost:8080/
          predicates:
            - Path= /customers/**
        - id: r2
          uri: http://localhost:8081/
```

```
            predicates:
              - Path= /products/**
```

2 – En utilisant RouteLocatorBuilder :

```
@Bean
RouteLocator
routeLocator(RouteLocatorBuilder builder)
{
    return builder.routes()
              .route((r) ->
r.path("/customers/**").uri("lb://CUSTOME
R-SERVICE"))
              .route((r) ->
r.path("/products/**").uri("lb://INVENTOR
Y-SERVICE"))
              .build();
}
```

-Configuration dynamique (DiscoveryClientRoute DefinitionLocator) :

ici on dit a spring cloud gateway que nous enfaite on ne connait pas les routes,
* mais a chaque fois que TU reçois une requête, regarde dans le corps de la requête et tu va trouver
* le nom du micro Service, en utilisant ce nom Eureka va te fournir l'@ et le port pour que tu y accède.

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

Cette method permet de recuperer les requete puis les associer avec les services correspondant selon le service mentionné dans la requete.

VI – Eureka discovery :

C'es le service d'enregistrement il sert a enregistrer les services pour facilliter le travail du gateway, en creant le projet on a que a configurer son fichier application.properties, mais coté dépendances ca differe car dans les service precedants on a installer eureka discovery client :

```
<dependency>

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

Tandis que dans note service d'enregistrement on va installer eureka discovery server :

```
<dependency>

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

On note bien que les microservices connaissent que le serveur d'enregistrement tourne par defaut sur le port : 8761, donc on a qu'a préciser au service qu'il n'a pas a chercher de s'enregistrer quelque part puisque c'est au autre service de le chercher.

```
server.port = 8761
eureka.client.fetch-registry=false
eureka.client.register-with-eureka=false
```

VII – Billing service :

Ce microservice prend en charge la gestion des factures, ill dispose aussi de sa propre base de données :

```
spring.datasource.url = jdbc:h2:mem:billingDB
spring.cloud.discovery.enabled = true
```

Prénom & nom : Fahd KORAICHE
Groupe : 5IIRG2C-C

Entités :

```
@Entity @Data @NoArgsConstructor @AllArgsConstructor
@ToString
public class Bill {
    @Id @GeneratedValue(strategy =
GenerationType.IDENTITY)
    private Long id;
    private Date billingDate;
    @OneToMany(mappedBy="bill")
    private Collection<ProductItem> productItems;
    private Long CustomerID;
    @Transient
    private Customer customer;
}
```

```
@Entity @Data @NoArgsConstructor @AllArgsConstructor @ToString
public class ProductItem {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private double quantity;
    private double price;
    private long productID;
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    @ManyToOne
    private Bill bill;
    @Transient
    private Product product;
    @Transient
    private String productName;
}
```

Repositories :

```
@RepositoryRestResource
public interface ProductItemRepository
extends JpaRepository<ProductItem,Long> {
    public Collection<ProductItem>
findByBillId(Long id);
}
```

```
@RepositoryRestResource
public interface BillRepository extends
JpaRepository<Bill,Long> {
}
```

On note bien que la particularité de ce microservice c'est qu'il utilize les methods/objets d'autres microservice a distance en utilisant OpenFeign client :

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

Pour utiliser cette dependance on va poursuivre en créant des interface restClient pour faire une abstraction du code distant dans notre service en local :

```
@FeignClient(name="CUSTOMER-SERVICE")
public interface CustomerRestClient {
    @GetMapping(path ="/customers/{id}")
    Customer getCustomerById(@PathVariable(name="id")
Long id);

    @GetMapping(path = "/customers")
    Collection<Customer> getCustomers();
}
@FeignClient(name="INVENTORY-SERVICE")
public interface ProductItemRestClient {
    @GetMapping(path ="/products")
    PagedModel<Product>
pageProducts(@RequestParam(value="page") int page,
@RequestParam(value="size") int size);
    @GetMapping(path ="/products/{id}")
    Product getProductById(@PathVariable(name="id")
Long id);
}
```

Mais puisque on a pas concrètement les classes distantes chez nous on va créer des modèles qui leurs ressemblent pour pouvoir récupérer leurs objets et pour faire la correspondance ente eux on va ajouter une nouvelle dépendance qui se base sur JSON :

```
<dependency>
     <groupId>
          org.springframework.boot
      </groupId>
     <artifactId>
          spring-boot-starter-hateoas
      </artifactId>
</dependency>
```

Mise en situation :

```
@Bean
CommandLineRunner start(BillRepository billRepository,
                        ProductItemRepository productItemRepository,
                        CustomerRestClient customerRestClient,
                        ProductItemRestClient productItemRestClient){
    return args -> {
        Customer customer = customerRestClient.getCustomerById(11);
        Bill bill1 = billRepository.save(new Bill(null, new Date(), null,
customer.getId(), null));
        PagedModel<Product> productPagedModel =
productItemRestClient.pageProducts(0,3);
        productPagedModel.forEach(p->{
            ProductItem productItem = new ProductItem();
            productItem.setQuantity(1 + new Random().nextInt(100));
            productItem.setPrice(p.getPrice());
            productItem.setBill(bill1);
            productItem.setProductID(p.getId());
            productItemRepository.save(productItem);

        });
    };
}
```

Dans l’exemple au-dessous on démontre l’utilisation combine entre les repository qu’on a et les restClient qu’on import en utilisant Openfeign.

Prénom & nom : Fahd KORAICHE
Groupe : 5IIRG2C-C

VII – Frontend - AngularJS :

Pour la derniere etape de notre procedure on va implementer une petite application angular qui va interroger notre gateway pour accedez au different API, dans cette exemple en va se contenter de recuperer une facture d'un seule client puis l'afficher dans notre page web.