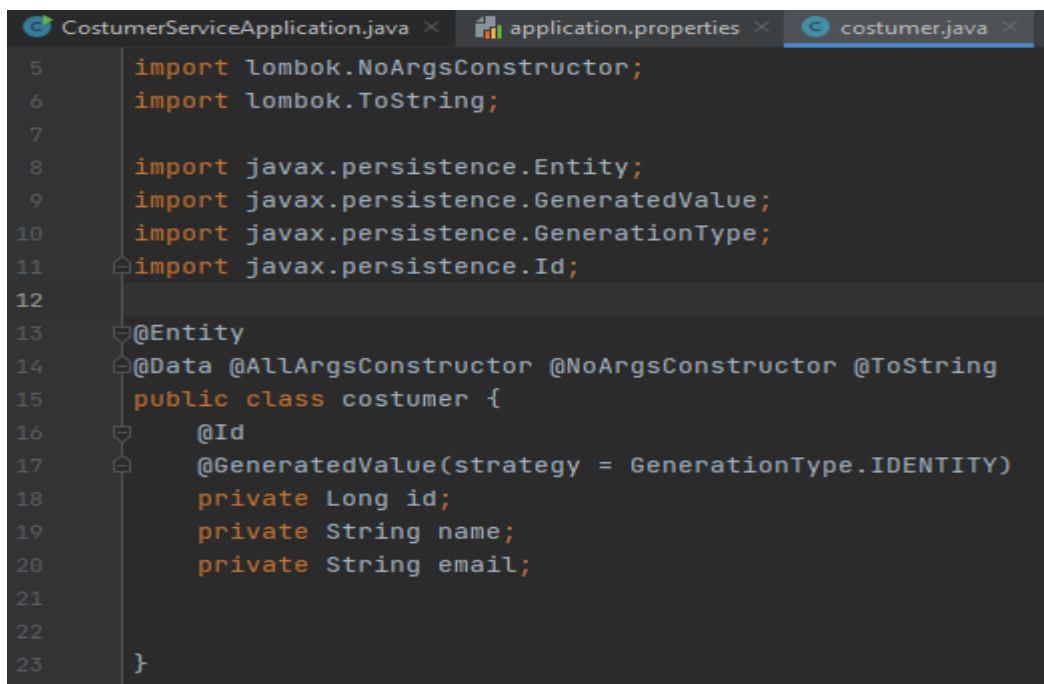


Rapport de projet d'architectures distribuées basées sur les micro-services

L'objectif est de réaliser une application basée sur les micro-services en premier nous allons créer un micro-service Customer-service qui permet de gérer des clients, ensuite on crée le micro service inventory-service , après le spring cloud Gateway pour la configuration statique , la configuration dynamique va être faite à l'aide Eureka-service , ensuite la création du deuxième micro-service qui permet de gérer les factures, la communication entre ces deux micro services va être à l'aide de openFeign pour une communication REST, cette application va être basée sur SPRING Framework en mettant en œuvre l'architecture de micro-service avec spring cloud

CUSTOMER-SERVICE :

Le micro service Customer se base en premier sur les dépendances suivante : springWeb, springDataJpa, h2-database, RestRepository , Lombok , devTools , EurekaDiscoveryClient, SpringBootActuator , après sa création par le springinitializr on cree notre entité jpa qui contient un id , le nom et l'adresse mail du client ,on ajoute donc l'annotation entity pour mentionné que c'est une entité ensuite le Lombok à l'aide l'entité data pour générer les getters et setter , un constructeur avec argument et un autre sans argument



```
CostumerServiceApplication.java × application.properties × costumer.java ×
5      import lombok.NoArgsConstructor;
6      import lombok.ToString;
7
8      import javax.persistence.Entity;
9      import javax.persistence.GeneratedValue;
10     import javax.persistence.GenerationType;
11     import javax.persistence.Id;
12
13     @Entity
14     @Data @AllArgsConstructor @NoArgsConstructor @ToString
15     public class costumer {
16         @Id
17         @GeneratedValue(strategy = GenerationType.IDENTITY)
18         private Long id;
19         private String name;
20         private String email;
21
22
23     }
```

Ensuite on crée notre package repositories qui contiendra la repository CustomerRepository, en mentionnant dans cette dernière l'entité qu'on va gérer qui est Customer et un id de type Long pour exposer ceci sous forme application Restfull on ajoute l'annotation RepositoryRestResource

```
CustomerServiceApplication.java x application.properties x costumer.java x CustomerRepository.java x
package org.sid.costumerservice.repository;

import org.sid.costumerservice.entities.costumer;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

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

Pour le test de ce micro service on ajoute quelques données dans la base de données, on crée l'objet CommandLineRunner qu'on va nommer la fonction Start qui retourne une expression lambda qui va s'exécuter au démarrage on ajoute la notation Bean, on injecte le customerRepository pour insérer des enregistrements

```
import ...

@SpringBootApplication
public class CustomerServiceApplication {

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

    @Bean
    CommandLineRunner start(CustomerRepository c, RepositoryRestConfiguration Rrc) {
        Rrc.exposeIdsFor(costumer.class);
        return args -> {
            c.save(new costumer( id: null, name: "Mohamed", email: "med@gmail.com"));
            c.save(new costumer( id: null, name: "Ikram", email: "ikram@gmail.com"));
            c.save(new costumer( id: null, name: "Mourad", email: "mourad@gmail.com"));
            c.findAll().forEach(a->{
                System.out.println(a.toString());
            });
        };
    }
}
```

Ensuite on indique le port à la valeur 8081, le nom de l'application, le nom de la base de données et on active le spring cloud discovery pour s'enregistrer au démarrage

```
server.port=8081
spring.application.name=costmuer-service
spring.datasource.url=jdbc:h2:mem:costumer-db
spring.cloud.discovery.enabled=true
management.endpoints.web.exposure.include=*
```

INVENTORY-SERVICE :

Ce service contient les mêmes dépendances que Customer-service qui sont springWeb, springDataJpa, h2-database, RestRepository, Lombok, devTools, EurekaDiscoveryClient, SpringBootActuator, après sa création par le springinitializr on cree notre entité jpa Product qui contient l id, le nom, le prix, la quantité du produit, ,on ajoute donc l'annotation entity pour mentionné que c'est une entité ensuite le Lombok à l'aide l'entité data pour générer les getters et setter , un constructeur avec argument et un autre sans argument ,ensuite on ajoute la repository ProductRepository, en mentionnant dans cette dernière l'entité qu'on va gérer qui est Product et un id de type Long pour expose ceci sous forme application Restfull on ajoute l'annotation RepositoryRestResource

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

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

Pour le test de ce micro service on ajoute quelque données dans la base de données, on cree l'objet CommandLineRunner qu'on va nommer la fonction Start qui retourne une expression lambda qui va s'exécuter au démarrage on ajoute la notation Bean, on injecte le productRepository pour insérer des enregistrements

```
@SpringBootApplication
public class InventoryServiceApplication {

    public static void main(String[] args) {

        SpringApplication.run(InventoryServiceApplication.class, args);
    }

    @Bean
    CommandLineRunner start(ProductRepository p, RepositoryRestConfiguration Rrc){
        Rrc.exposeIdsFor(Product.class);

        return args ->{

            p.save(new Product(null,"ordi",788,12));
            p.save(new Product(null,"SMARTPHONE",1788,120));
            p.save(new Product(null,"IMPRIMENTE",18,129));
            p.findAll().forEach(s->{
                System.out.println(s.toString());
            });
        };
    }
};
```

Ensuite on indique le port a la valeur 8082, le nom de l'application product-service, le nom de la base de données product-db et on active le spring cloud discovery pour s'enregistrer au démarrage

```
server.port=8082
spring.application.name=product-service
spring.datasource.url=jdbc:h2:mem:product-db
spring.cloud.discovery.enabled=true
#management.endpoints.web.exposure.include=*
```

GATEWAY-SERVICE :

Pour accéder à l'ensemble des micro service on a besoin d'un service GATEWAY et la configurer pour qu'on puisse dispatcher les requêtes au micro service convenable, pour cree ce service on cree un nouveau projet avec les dépendances suivante : EurekaDiscoveryClient, SpringBootActuator, Gateway. Après la création de GATEWAY nous allons cree un fichier de configuration nommer app.yml qu'on peut l'utiliser en parallèle avec le fichier application.properties, ce dernier va contenir le port de serveur 8888, le nom de micro service Gateway-service

```
GatewayApplication.java x app.yml x application.properties x
1 server.port=8888
2 spring.application.name=gateway-service
3 spring.cloud.discovery.enable=true
4
```

Pour faire la configuration des routes nous allons utiliser le fichier de configuration app.yml, on spécifie les routes en premier par un id, Uri et les prédicats qui contiennent un path ou un chemin, on spécifie les routes pour les micros service d'une manière statique

```
GatewayApplication.java x app.yml x application.properties x
1 spring:
2   cloud:
3     gateway:
4       routes:
5         - id: r1
6           uri: http://localhost:8081/
7           predicates:
8             - Path=/costumers/**
9         - id: r2
10          uri: http://localhost:8082/
11          predicates:
12            - Path= /products/**
13       discovery:
14         enabled: false
15 server:
16   port: 8888
```

On cree un Object nomme discoveryClientRouteDefinitionLocator qui prend en paramètre un reactiveDiscoveryClient et un discoverylocatorProperties qui va retourner un Object de type discoveryClientRouteDefinitionLocator ce dernier

prendra le nom du micro service à partir de l'URL et il va router la requête vers le bon service

```
12  @SpringBootApplication
13  ▶ public class GatewayApplication {
14
15  ▶     public static void main(String[] args) {
16
17         SpringApplication.run(GatewayApplication.class, args);
18     }
19
20     // @Bean
21     // RouteLocator gatewayRoutes(RouteLocatorBuilder builder){
22         //return builder.routes()
23         //    .route(r->r.path("/costumers/**").uri("lb://COSTMUE-SERVICE"))
24         //    .route(r->r.path("/products/**").uri("lb://PRODUCT-SERVICE"))
25         //    .build();
26
27     //}
28     @Bean
29     DiscoveryClientRouteDefinitionLocator dynamicRoutes(ReactiveDiscoveryClient rdc,
30                                                         DiscoveryLocatorProperties dlp){
31         return new DiscoveryClientRouteDefinitionLocator(rdc, dlp);
32     }
33 }
```

EUREKA -DISCOVERY:

Ce service contient la dépendance EurekaDiscoveryClient, on commence par la notation EnableEurekaServer pour activer le serveur eureka, ce dernier nous donne tout les instances ou les micro service enregistrer, en prenant l'adresse IP de chaque service

```
EurekaDiscoveryApplication.java x
1  package com.sid.Eurekadiscovery;
2
3  import ...
4
5
6
7  @SpringBootApplication
8  @EnableEurekaServer
9
10 ▶ public class EurekaDiscoveryApplication {
11
12 ▶     public static void main(String[] args) { SpringApplication.
13
14
15
16     }
17 }
```

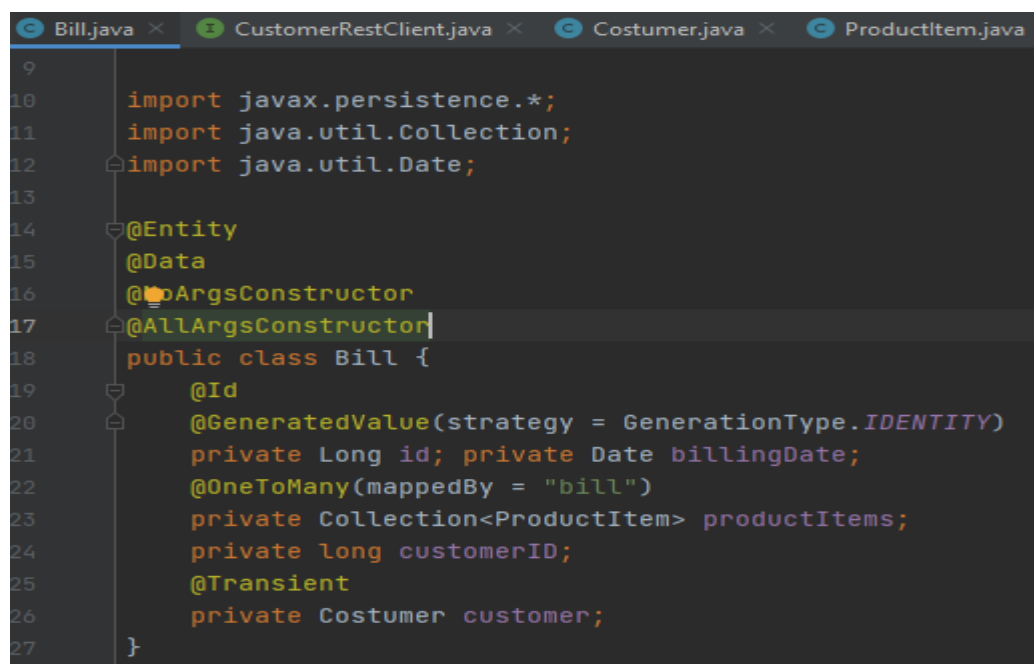
Ensuite on ajoute dans le fichier de configuration le port de serveur 8761, on désactive le fetch-registry et le register-with-eureka, on n'a pas besoin d'exposer les services dans l'annuaire

```
ekaDiscoveryApplication.java x application.properties x
server.port=8761
# dont register server itself as a client.
eureka.client.fetch-registry=false
# Does not register itself in the service registry.
eureka.client.register-with-eureka=false
|
```

BILLING-SERVICE :

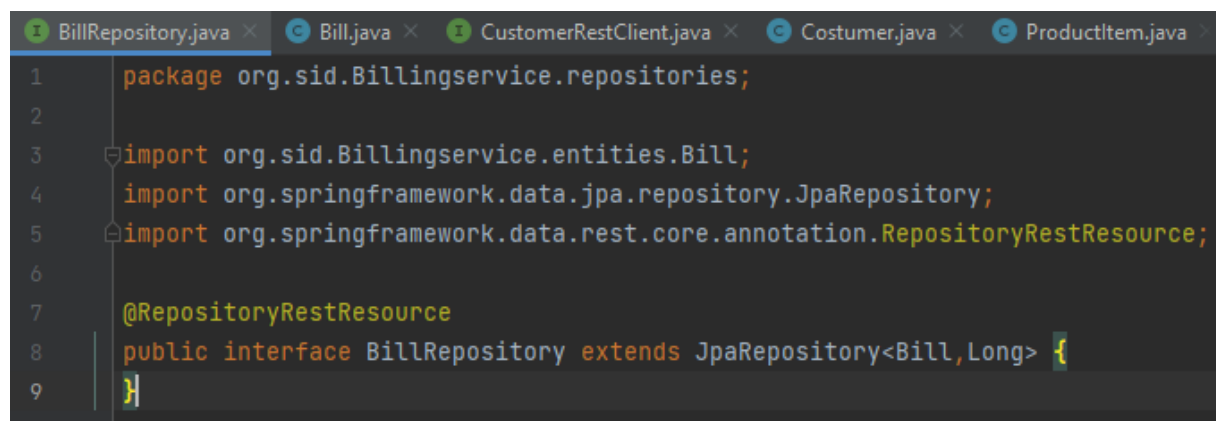
Ce service gère la facturation qui va communiquer avec Customer-service et inventory-service pour avoir la liste des produits et des informations sur le client, dans ce service on a besoin d'utiliser le Feign Framework qui permet de créer une communication REST, il contient comme dépendances springWeb, springDataJpa, h2-database, RestRepository, Lombok, devTools, EurekaDiscoveryClient, OpenFeign, SpringHATEOAS POUR UN FORMAT JSON

Après sa création par le springinitializr on crée notre entité jpa Bill qui contient l'id, la date de facturation, l'id du client, on ajoute donc l'annotation entity pour mentionner que c'est une entité ensuite le Lombok à l'aide de l'entité data pour générer les getters et setters, un constructeur avec argument et un autre sans argument



```
9
10 import javax.persistence.*;
11 import java.util.Collection;
12 import java.util.Date;
13
14 @Entity
15 @Data
16 @AllArgsConstructor
17 @AllArgsConstructor
18 public class Bill {
19     @Id
20     @GeneratedValue(strategy = GenerationType.IDENTITY)
21     private Long id; private Date billingDate;
22     @OneToMany(mappedBy = "bill")
23     private Collection<ProductItem> productItems;
24     private long customerID;
25     @Transient
26     private Customer customer;
27 }
```

Ensuite on ajoute la repository BillRepository, en mentionnant dans cette dernière l'entité qu'on va gérer qui est Bill et un id de type Long pour exposer ceci sous forme d'application Restfull on ajoute l'annotation RepositoryRestResource



```
1 package org.sid.Billingservice.repositories;
2
3 import org.sid.Billingservice.entities.Bill;
4 import org.springframework.data.jpa.repository.JpaRepository;
5 import org.springframework.data.rest.core.annotation.RepositoryRestResource;
6
7 @RepositoryRestResource
8 public interface BillRepository extends JpaRepository<Bill, Long> {
9
10 }
```

On crée notre entité jpa ProductItem qui contient l'id, un productId, le prix, la quantité, le nom de produit. On ajoute donc l'annotation entity pour mentionner que c'est une entité ensuite le Lombok à l'aide de l'entité data pour générer les getters et setters, un constructeur avec argument et un autre sans argument.

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

Ensuite on ajoute la repository ProductItemRepository, en mentionnant dans cette dernière l'entité qu'on va gérer qui est Bill et un id de type Long pour exposer ceci sous forme d'application Restfull on ajoute l'annotation RepositoryRestResource, dans cette dernière on ajoute la fonctionnalité findByBillId la recherche par l'id d'une facture.

```
BillRepository.java x ProductItemRepository.java x Bill.java x Costumer.java x ProductItem.java x
1 package org.sid.Billingservice.repositories;
2
3 import org.sid.Billingservice.entities.Bill;
4 import org.sid.Billingservice.entities.ProductItem;
5 import org.springframework.data.jpa.repository.JpaRepository;
6 import org.springframework.data.rest.core.annotation.RepositoryRestResource;
7
8 import java.util.Collection;
9
10 @RepositoryRestResource
11 public interface ProductItemRepository extends JpaRepository<ProductItem, Long> {
12     public Collection<ProductItem> findByBillId(long id);
13
14 }
```

Après on va créer un modèle qui contient deux classes dont la persistance est gérée dans les deux autres microservices, en premier on crée la classe `Costumer` qui contient comme attribut l'id, le nom et l'email du client

```
1 package org.sid.Billingservice.model;
2
3 import lombok.Data;
4
5 @Data
6 public class Costumer {
7     private Long id;
8     private String name;
9     private String email;
10 }
```

La classe `Product` contient comme attribut l'id, le nom et le prix du produit

```
package org.sid.Billingservice.model;
import lombok.Data;
@Data
public class Product {
    private Long id;
    private String name;
    private double price;
}
```

Ensuite en utilisant le `Openfeign` qui va nous permettre de communiquer avec d'autres microservices d'une manière déclarative via le REST, on crée une interface nommée `CustomerRestClient` qui va nous permettre d'accéder au client à chaque fois qu'on a besoin de quelque information à propos du client. L'annotation `FeignClient` prendra le nom du microservice, ensuite on déclare la méthode `getCustomerById` permet d'envoyer une requête vers `Customer service`

```
package org.sid.Billingservice.feign;
import org.sid.Billingservice.model.Costumer;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
@FeignClient(name="COSTMUER-SERVICE")
public interface CustomerRestClient {
    @GetMapping("/costumers/{id}")
    Costumer getCustomerById(@PathVariable(name="id") Long id);
}
```

On crée par la suite une interface nommée `ProductItemRestClient` qui contient l'annotation `GetMapping` pour recevoir les produits après une méthode de type `PageModel` qui retourne des valeurs en format JSON, ensuite le deuxième `GetMapping` qui retourne produit recherché par son id


```

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.hateoas.PagedModel;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestParam;

import javax.ws.rs.QueryParam;

@FeignClient(name="PRODUCT-SERVICE")
public interface ProductItemRestClient {
    @GetMapping(path="/products")
    PagedModel<Product> pageProduct();
    @GetMapping(path="/products/{id}")
    Product getProductById(@PathVariable Long id);
}

```

On cree un RestController nommé billingRestController qui permet de consulter une facture complète ce qui va contenir les informations du produit et du client

En premier il va prendre l id de la facture en la cherchant dans billRepository ensuite on cree un customer qui va recevoir le client par le CustomerId qui est dans la classe Bill, ce dernier va être ajouter dans le customer qui est dans la classe bill de type transient, on utilise un forEach pour recevoir les productItems qui existe dans bill selon le productId dans la classe productItems pour enfin de compte avoir tout les product item de bill

```

@RestController
public class BillingRestController {
    @Autowired
    private BillRepository billRepository;
    @Autowired private ProductItemRepository productItemRepository;
    private CustomerRestClient customerRestClient;
    private ProductItemRestClient productItemRestClient;

    public BillingRestController(BillRepository billRepository,
                               ProductItemRepository productItemRepository, CustomerRestClient customerRestClient, ProductItemRestClient productItemRestClient) {
        this.billRepository = billRepository;
        this.productItemRepository = productItemRepository;
        this.customerRestClient = customerRestClient;
        this.productItemRestClient = productItemRestClient;
    }

    @GetMapping(path = "/fullBill/{id}")
    public Bill getBill(@PathVariable(name="id") Long id) {
        Bill bill = billRepository.findById(id).get();
        Customer customer = customerRestClient.getCustomerById(bill.getCustomerID());
        bill.setCustomer(customer);
        bill.getProductItems().forEach(pi -> {
            Product product = productItemRestClient.getProductById(pi.getProductID());
            //pi.setProduct(product);
            pi.setProductName(product.getName());
        });
        return bill;
    }
}

```

on cree l'objet CommandLineRunner qu'on va nommer la fonction Start qui retourne une expression lambda qui va s'exécuter au démarrage on ajoute la notation Bean

```
@EnableFeignClients
@SpringBootApplication
public class BillingServiceApplication {

    public static void main(String[] args) {

        SpringApplication.run(BillingServiceApplication.class, args);
    }

    @Bean
    CommandLineRunner start(BillRepository billRepo,
                            ProductItemRepository ProductItemRepository,
                            CustomerRestClient CustomerRestClient,
                            ProductItemRestClient ProductItemRestClient){

        return args -> {
            Costumer cu=CustomerRestClient.getCustomerById(1L);
            Bill bill1=billRepo.save(new Bill( id: null,new Date(), productItems: null,cu.getId(), customer: null));

            //Bill bill=billRepo.save(new Bill(null,new Date(),null,cu.getId(),null));
            PagedModel<Product> productPagedModel=ProductItemRestClient.pageProduct();
            productPagedModel.forEach(p->{
                ProductItem pri=new ProductItem();
                pri.setPrice(p.getPrice());
                pri.setQuantity(1+new Random().nextInt( bound: 100));
                pri.setBill(bill1);
                pri.setProductID(p.getId());
                ProductItemRepository.save(pri);
            });
        };
    }
}
```