

TP3- part 3:

Architecture micro service avec Spring Cloud

Application: Order-service

1. Cree le package **entities**.
2. Cree la classe **Order**.

```
@Entity
// Pour certains SGBD, on ne peut pas utiliser "Order"
// directement car c'est un mot-clé. Il vaut mieux
// utiliser l'annotation @Table.
@Table(name = "Orders")
@NoArgsConstructor @AllArgsConstructor @Builder
public class Order {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Date createdAt;
    private OrderStatus status;
    private Long customerId;
    @Transient //indique que le champ ne doit pas être persisté
    private Customer customer;
    @OneToMany(mappedBy = "order")
    private List<ProductItem> productItems;
}
```

@Transient: indique qu'un champ est non persistant.
Un attribut non persistant fait référence à un attribut d'une classe qui n'est pas stocké de manière permanente dans une base de données.

3. Cree le package **enums**
4. Cree la classe **OrderStatus**

```
public enum OrderStatus {
    no usages
    CREATED, PENDING, DELIVERED, CANCELED
}
```

5. Cree la classe **ProductItem** dans le package **entities**

```
@Entity
@Data @NoArgsConstructor @AllArgsConstructor @Builder
public class ProductItem {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Long productId;
    @Transient
    private Product product;
    private double price;
    private int quantity; //quantity in order
    private double discount;
    @ManyToOne
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private Order order;
}
```

Application: Order-service

- 6. Cree le package **model**.
- 7. Cree la classe **Product**.
- 8. Cree le package **enums**
- 9. Cree la classe **OrderStatus**

On utilise ces deux classe pour stoker les détails de Product et Customer

```
2 usages
@Data
public class Product {
    private Long id;
    private String name;
    private double price;
    private int quantity;
}
```

```
2 usages
@Data
public class Customer {
    private Long id;
    private String name;
    private String email;
}
```

- 10. Cree le package **repo**.
- 11. Cree l'interface **OrderRepository**
- 12. Cree l'interface **ProductItemRepository**

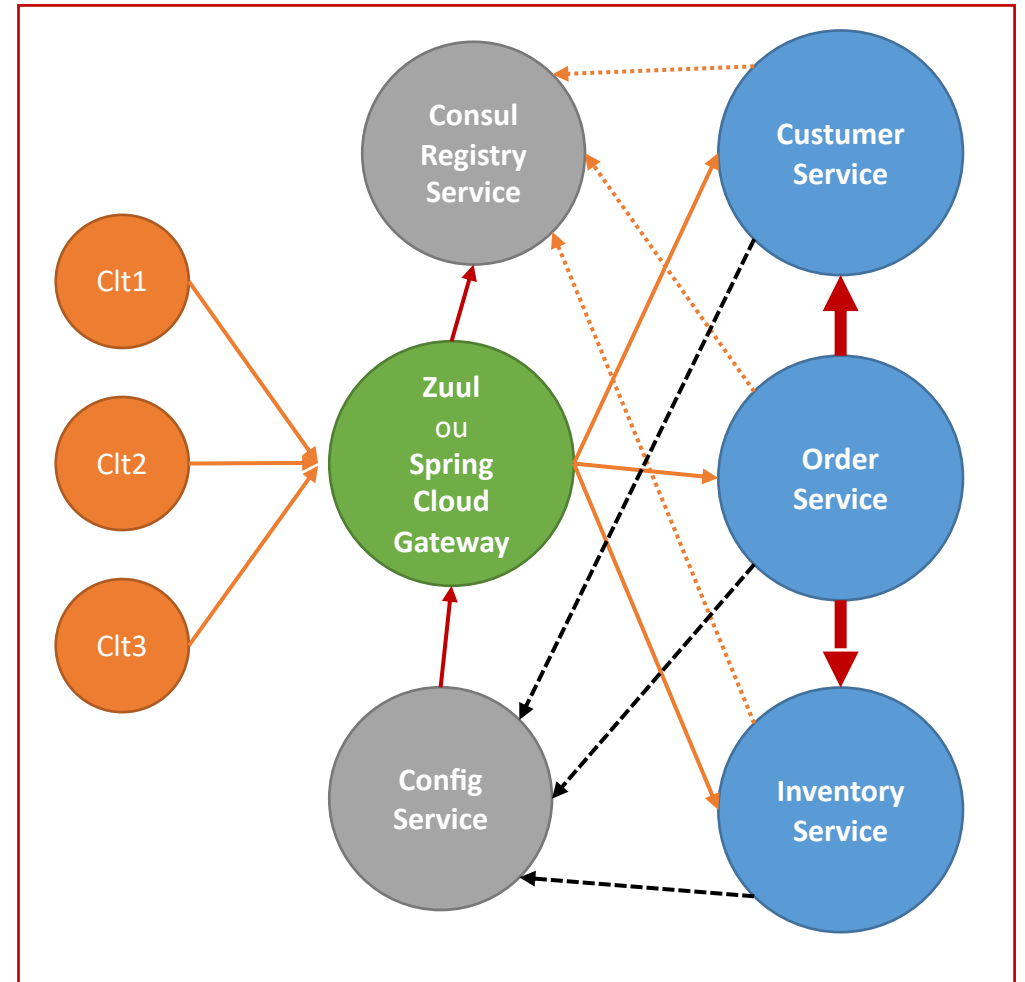
```
@RepositoryRestResource
public interface OrderRepository extends JpaRepository<Order, Long> {
}
```

```
@RepositoryRestResource
public interface ProductItemRepository extends JpaRepository<ProductItem, Long> {
}
```

Application: Order-service

Order-service a besoin de communiquer avec customer-service et inventory-service. On peut utiliser RestTemplate comme dans TP2 ou on peut utiliser **OpenFeign**.

OpenFeign est une bibliothèque Java qui simplifie le développement d'applications basées sur microservices en facilitant les appels de service à service. Elle fait partie du projet Spring Cloud et offre une abstraction déclarative des appels HTTP, permettant aux développeurs de définir des interfaces de service simples et d'utiliser des annotations pour décrire le comportement des requêtes.



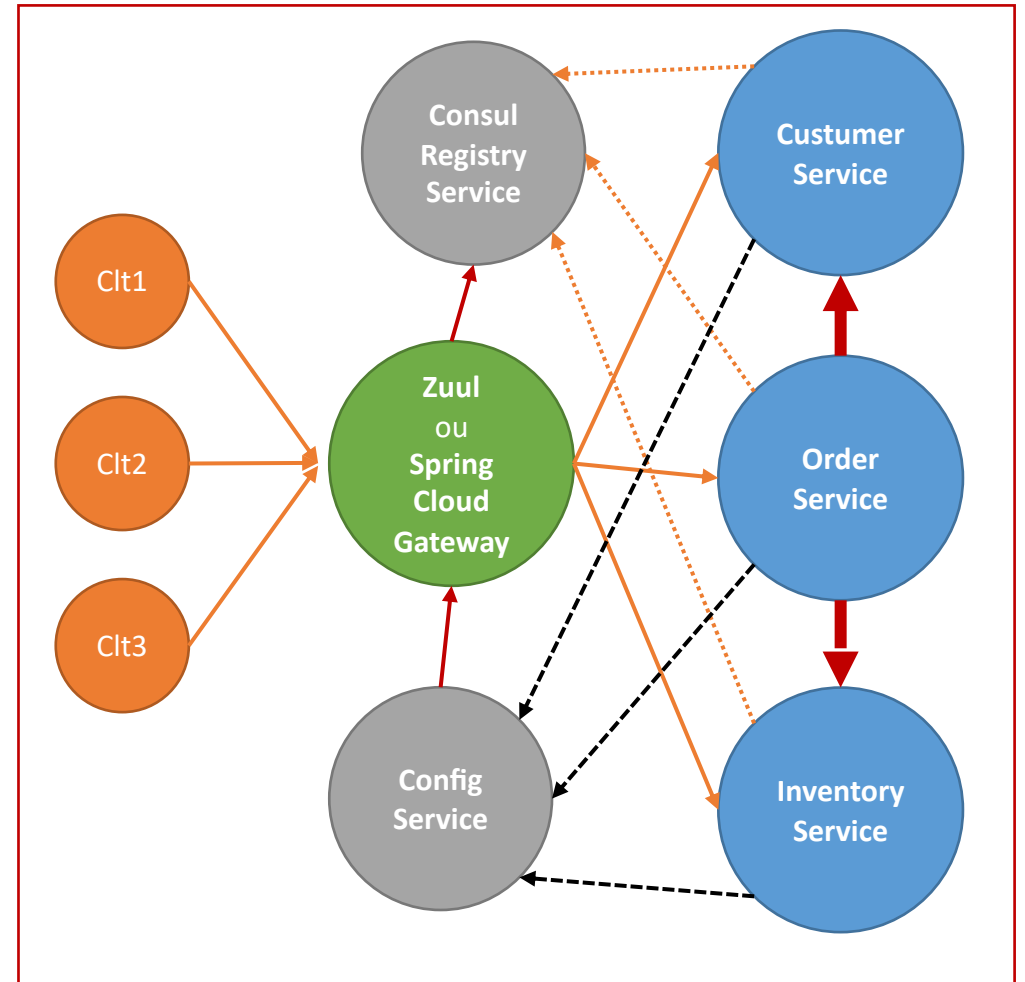
Application: Order-service

13. Pour utiliser OpenFeign il faut ajouter la dependance suivante a pom.xml

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

14. On aura aussi besoin de:

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



Application: Order-service

15. Cree le package **services**.

16. Cree les clients rest en créant deux interfaces: **CustomerRestClientService** et **ProductRestClientService**.

```
no usages
@FeignClient(name = "customer-service")
public interface CustomerRestClientService {
    no usages
    @GetMapping("/customers/{id}?projection=fullCustomer")
    public Customer customerById(@PathVariable Long id);
    no usages
    @GetMapping("/customers?projection=fullCustomer")
    public PagedModel<Customer> allCustomers();
    //springDataRest nous donne une page dans laquelle il list
    //les customers. C'est pourquoi on utilise PagedModel de hateoas
}
```

```
5 usages
@FeignClient(name = "inventory-service")
public interface InventoryRestClientService {
    no usages
    @GetMapping("/products/{id}?projection=fullProduct")
    public Product productById(@PathVariable Long id);
    1 usage
    @GetMapping("/products?projection=fullProduct")
    public PagedModel<Product> allProducts();
}
```

Lorsqu'on invoque cette méthode en lui demandant d'envoyer une requête vers le micro0service customer-service, elle récupère le client correspondant à cet identifiant.

On a utilisé la projection pour pouvoir récupérer également l'ID.

Application: Order-service

17. Configurer le micro-service

Application.properties de order-service

```
server.port=8083  
spring.application.name=order-service  
spring.config.import=optional:configserver:http://localhost:8888/
```

Cree le fichier order-service.properties dans le dossier le configuration Config-repo

```
spring.datasource.url=jdbc:h2:mem:orders-db
```

N'oubliez pas :

```
admin@Imanes-MacBook-Pro Config-repo % git add .  
admin@Imanes-MacBook-Pro Config-repo % git commit -m "V4"
```

Application: Order-service

18. Pour effectuer le test, nous devons remplir la base de données de manière aléatoire.

- Nous commençons par récupérer les produits et les clients.
- Ensuite, nous créons 20 commandes de manière aléatoire.
- Pour chaque commande, nous lui attribuons une liste de produits de façon aléatoire.

```
@SpringBootApplication
@EnableFeignClients
public class OrderServiceApplication {

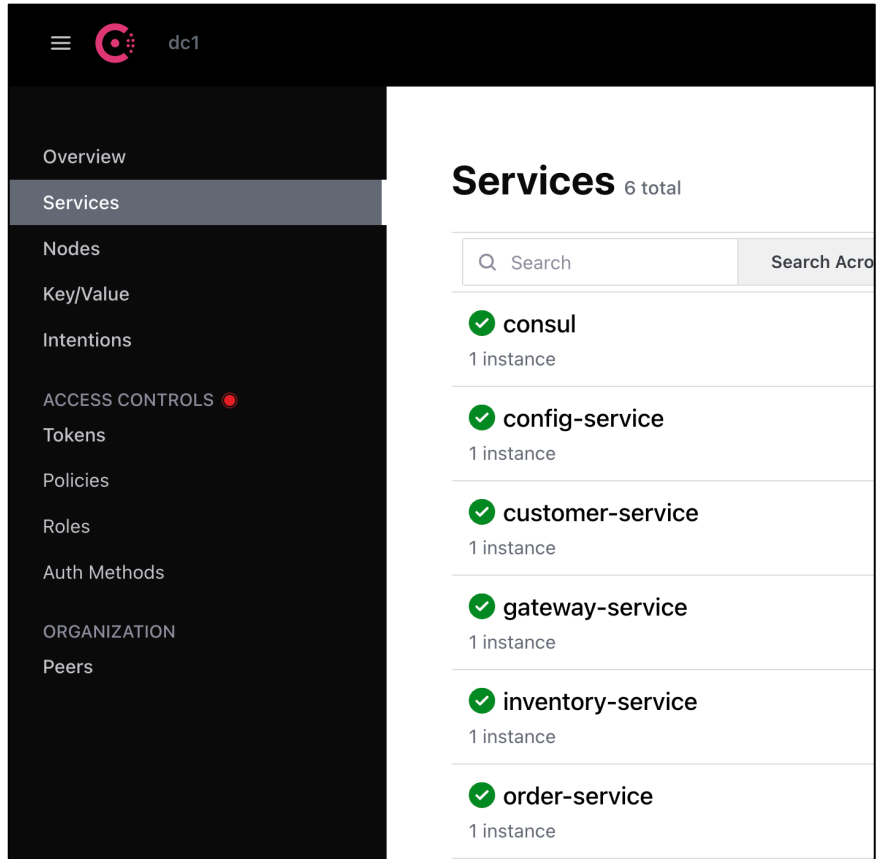
    public static void main(String[] args) { SpringApplication.run(OrderServiceApplication.class, args); }

    no usages
    @Bean
    CommandLineRunner start(OrderRepository orderRepository,
                            ProductItemRepository productItemRepository,
                            CustomerRestClientService customerRestClientService,
                            InventoryRestClientService inventoryRestClientService) {

        return args -> {
            List<Customer> customers = customerRestClientService.allCustomers().getContent().stream().toList();
            List<Product> products = inventoryRestClientService.allProducts().getContent().stream().toList();
            Long customerId = 1L;
            Random random = new Random();
            Customer customer = customerRestClientService.customerById(customerId);
            for (int i = 0; i < 20; i++) {
                Order order = Order.builder()
                    .customerId(customers.get(random.nextInt(customers.size())).getId())
                    .status(Math.random() > 0.5 ? OrderStatus.PENDING : OrderStatus.CREATED)
                    .createdAt(new Date())
                    .build();
                Order savedOrder = orderRepository.save(order);
                for (int j = 0; j < products.size(); j++) {
                    if (Math.random() > 0.75) {
                        ProductItem productItem = ProductItem.builder()
                            .order(savedOrder)
                            .productId(products.get(j).getId())
                            .price(products.get(j).getPrice())
                            .quantity(1 + random.nextInt(bound: 10))
                            .discount(Math.random())
                            .build();
                        productItemRepository.save(productItem);
                    }
                }
            }
        };
    }
}
```


Application: Order-service

19. Démarrer le micro-service pour tester



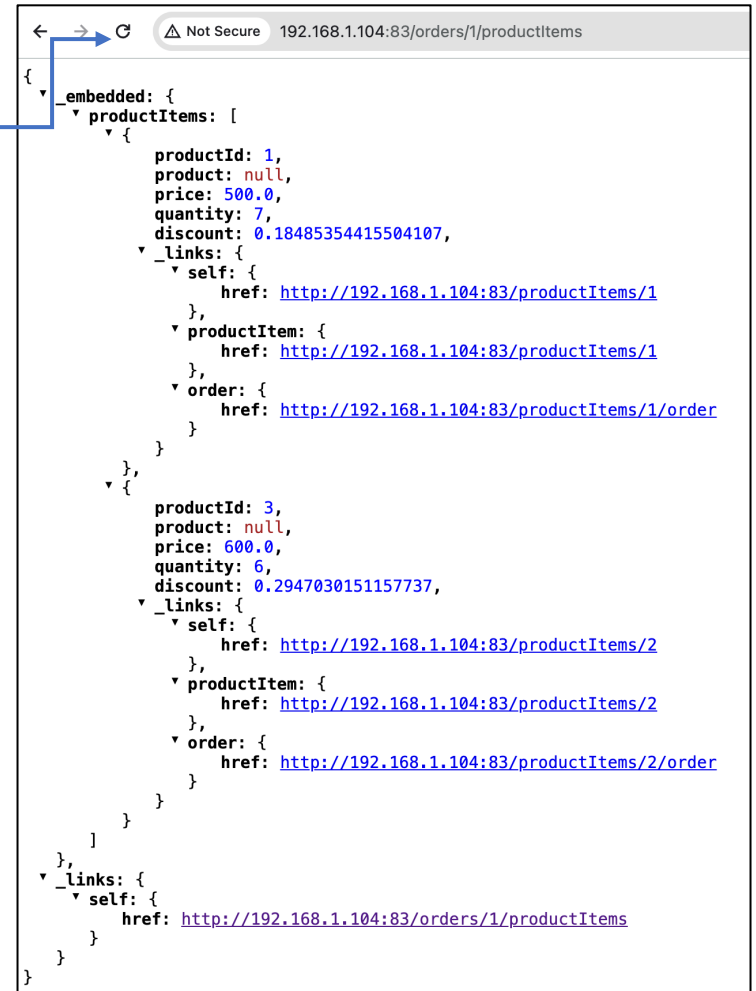
The screenshot shows the DC/OS (Docker Cloud Operating System) interface. On the left is a navigation menu with options: Overview, Services (selected), Nodes, Key/Value, Intentions, ACCESS CONTROLS, Tokens, Policies, Roles, Auth Methods, ORGANIZATION, and Peers. The main panel displays 'Services 6 total'. A search bar is present. Below it, a list of services is shown, each with a green checkmark icon and '1 instance' status:

- consul
- config-service
- customer-service
- gateway-service
- inventory-service
- order-service**

http://localhost:9999/order-service/orders

```
{
  "_embedded": {
    "orders": [
      {
        "_links": {
          "self": {
            href: "http://192.168.1.104:83/orders/1"
          }
        },
        "order": {
          href: "http://192.168.1.104:83/orders/1"
        },
        "productItems": {
          href: "http://192.168.1.104:83/orders/1/productItems"
        }
      },
      {
        "_links": {
          "self": {
            href: "http://192.168.1.104:83/orders/2"
          }
        },
        "order": {
          href: "http://192.168.1.104:83/orders/2"
        },
        "productItems": {
          href: "http://192.168.1.104:83/orders/2/productItems"
        }
      },
      {
        "_links": {
          "self": {
            href: "http://192.168.1.104:83/orders/3"
          }
        },
        "order": {
          href: "http://192.168.1.104:83/orders/3"
        },
        "productItems": {
          href: "http://192.168.1.104:83/orders/3/productItems"
        }
      },
      {
        "_links": {
          "self": {
            href: "http://192.168.1.104:83/orders/4"
          }
        },
        "order": {
          href: "http://192.168.1.104:83/orders/4"
        },
        "productItems": {
          href: "http://192.168.1.104:83/orders/4/productItems"
        }
      }
    ]
  },
  "_links": {
    "self": {
      href: "http://192.168.1.104:83/orders"
    }
  }
}
```

Pour chaque order on peut voir la list ProductItems

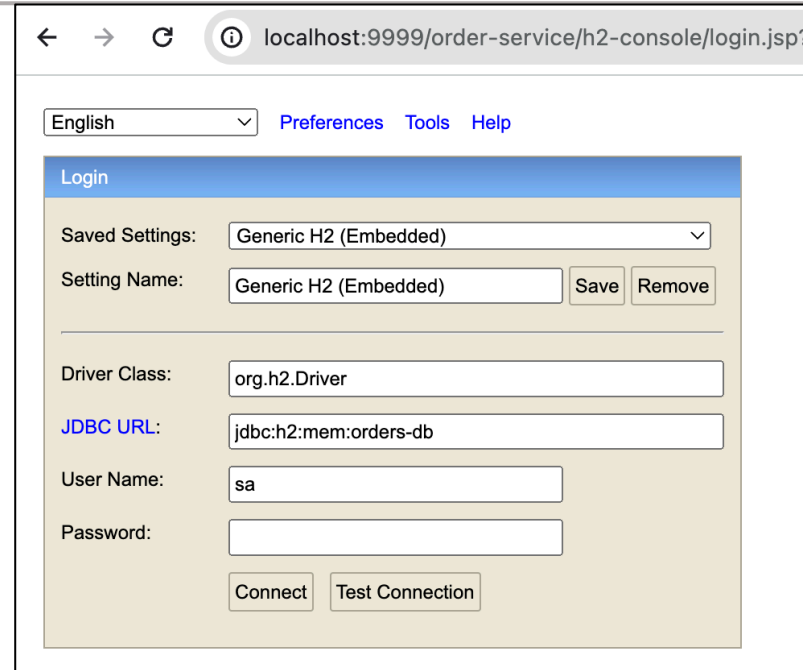


The screenshot shows a web browser window with the address bar displaying '192.168.1.104:83/orders/1/productItems'. The page content is a JSON array of product items:

```
{
  "_embedded": {
    "productItems": [
      {
        "productId": 1,
        "product": null,
        "price": 500.0,
        "quantity": 7,
        "discount": 0.18485354415504107,
        "_links": {
          "self": {
            href: "http://192.168.1.104:83/productItems/1"
          },
          "productItem": {
            href: "http://192.168.1.104:83/productItems/1"
          },
          "order": {
            href: "http://192.168.1.104:83/productItems/1/order"
          }
        }
      },
      {
        "productId": 3,
        "product": null,
        "price": 600.0,
        "quantity": 6,
        "discount": 0.2947030151157737,
        "_links": {
          "self": {
            href: "http://192.168.1.104:83/productItems/2"
          },
          "productItem": {
            href: "http://192.168.1.104:83/productItems/2"
          },
          "order": {
            href: "http://192.168.1.104:83/productItems/2/order"
          }
        }
      }
    ]
  },
  "_links": {
    "self": {
      href: "http://192.168.1.104:83/orders/1/productItems"
    }
  }
}
```

Application: Order-service

20. Vérifier la base de données



localhost:9999/order-service/h2-console/login.jsp?

English Preferences Tools Help

Login

Saved Settings: Generic H2 (Embedded)

Setting Name: Generic H2 (Embedded) Save Remove

Driver Class: org.h2.Driver

JDBC URL: jdbc:h2:mem:orders-db

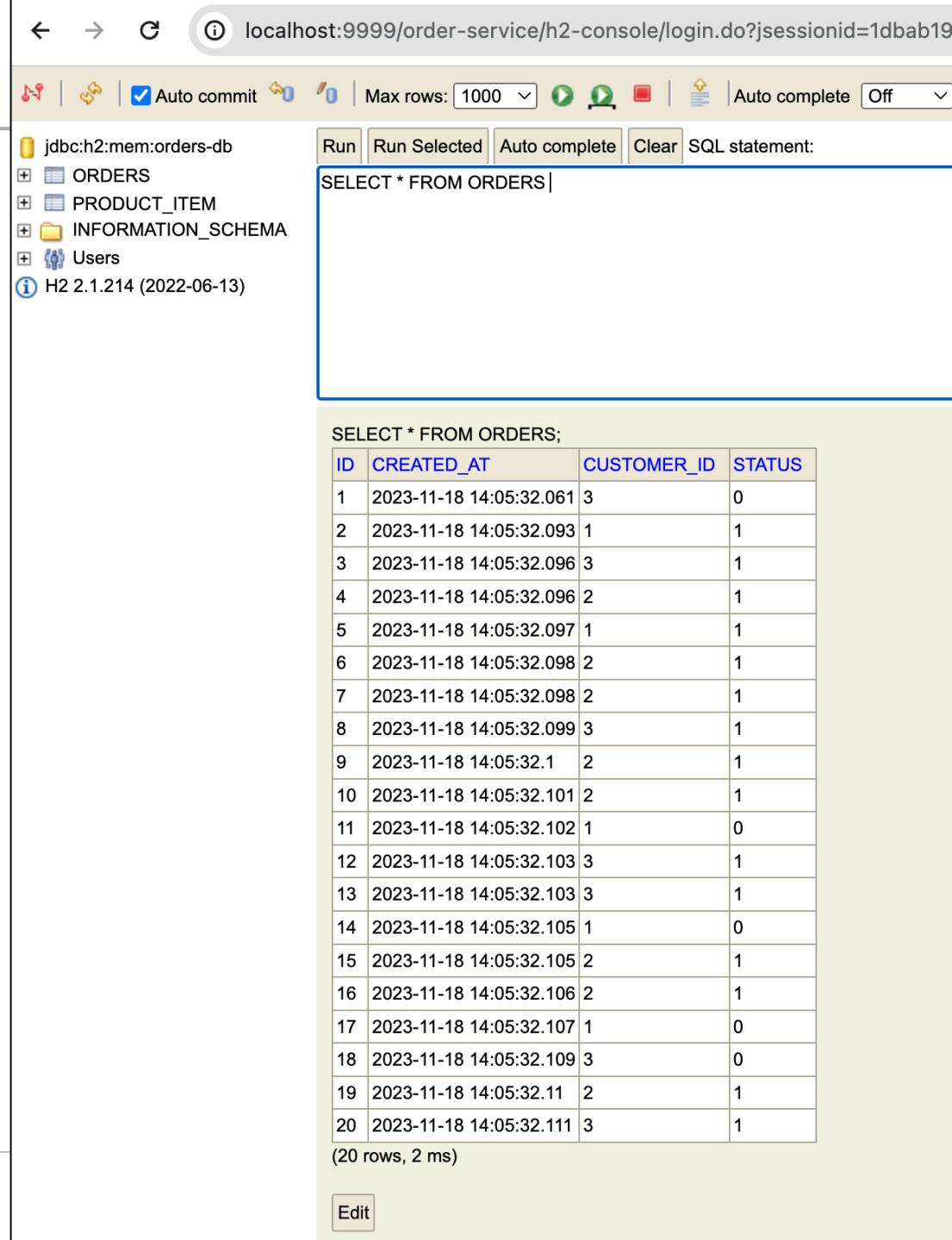
User Name: sa

Password:

Connect Test Connection

Si vous recevez l'erreur: H2 Console Sorry, remote connections ('webAllowOthers') are disabled on this server.

Dans application.properties de Config-repo ajouter
`spring.h2.console.settings.web-allow-others=true`
Faites le commit et redémarrez order-service



localhost:9999/order-service/h2-console/login.do?jsessionid=1dbab19

Auto commit Max rows: 1000 Auto complete Off

jdbc:h2:mem:orders-db

- ORDERS
- PRODUCT_ITEM
- INFORMATION_SCHEMA
- Users
- H2 2.1.214 (2022-06-13)

Run Run Selected Auto complete Clear SQL statement:

SELECT * FROM ORDERS;

SELECT * FROM ORDERS;

ID	CREATED_AT	CUSTOMER_ID	STATUS
1	2023-11-18 14:05:32.061	3	0
2	2023-11-18 14:05:32.093	1	1
3	2023-11-18 14:05:32.096	3	1
4	2023-11-18 14:05:32.096	2	1
5	2023-11-18 14:05:32.097	1	1
6	2023-11-18 14:05:32.098	2	1
7	2023-11-18 14:05:32.098	2	1
8	2023-11-18 14:05:32.099	3	1
9	2023-11-18 14:05:32.1	2	1
10	2023-11-18 14:05:32.101	2	1
11	2023-11-18 14:05:32.102	1	0
12	2023-11-18 14:05:32.103	3	1
13	2023-11-18 14:05:32.103	3	1
14	2023-11-18 14:05:32.105	1	0
15	2023-11-18 14:05:32.105	2	1
16	2023-11-18 14:05:32.106	2	1
17	2023-11-18 14:05:32.107	1	0
18	2023-11-18 14:05:32.109	3	0
19	2023-11-18 14:05:32.11	2	1
20	2023-11-18 14:05:32.111	3	1

(20 rows, 2 ms)

Edit

Application: Order-service

20. Cree un RestController qui nous permet de consulter une commande complète:

- Cree le package web
- Cree la classe OrderRestController

N'oubliez pas de faire l'injection en ajoutant des constructeurs avec paramètres ou en utilisant @Autowired

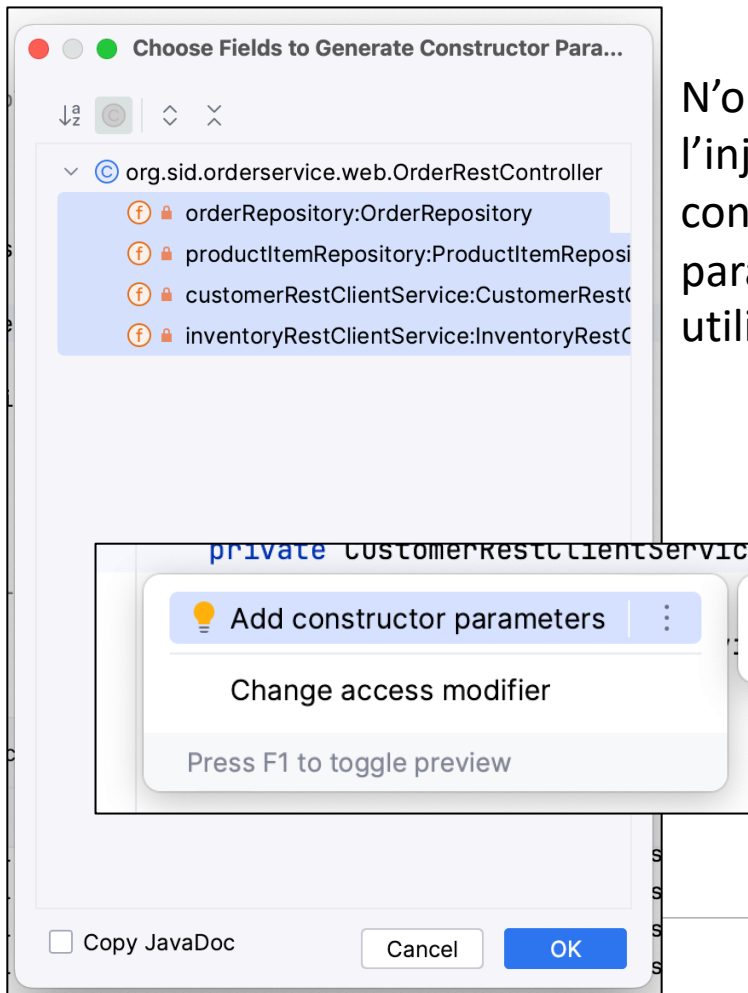
```
@RestController
public class OrderRestController {

    2 usages
    private OrderRepository orderRepository;
    1 usage
    private ProductItemRepository productItemRepository;
    2 usages
    private CustomerRestClientService customerRestClientService;
    2 usages
    private InventoryRestClientService inventoryRestClientService;

    no usages
    public OrderRestController(OrderRepository orderRepository,
                               ProductItemRepository productItemRepository,
                               CustomerRestClientService customerRestClientService,
                               InventoryRestClientService inventoryRestClientService) {

        this.orderRepository = orderRepository;
        this.productItemRepository = productItemRepository;
        this.customerRestClientService = customerRestClientService;
        this.inventoryRestClientService = inventoryRestClientService;
    }

    no usages
    @GetMapping("/fullOrder/{id}")
    public Order getOrder(@PathVariable Long id){
        Order order=orderRepository.findById(id).get();
        //on veut recuperer le detail du client
        Customer customer=customerRestClientService.customerById(order.getCustomerId());
        order.setCustomer(customer);
        order.getProductItems().forEach(pi->{
            Product product = inventoryRestClientService.productById(pi.getProductId());
            pi.setProduct(product);
        });
        return order;
    }
}
```



Application: Order-service

20. Testez si on arrive a consulter une commande complète.

```
localhost:9999/order-service/fullOrder/1

{
  id: 1,
  createdAt: "2023-11-19T14:38:23.292+00:00",
  status: "PENDING",
  customerId: 1,
  customer: {
    id: 1,
    name: "aaaaaa",
    email: "aaaa@gmail.com"
  },
  productItems: [
    {
      id: 1,
      productId: 1,
      product: {
        id: 1,
        name: "screen",
        price: 500.0,
        quantity: 12
      },
      price: 500.0,
      quantity: 6,
      discount: 0.918221258523199
    }
  ]
}
```

On a récupérer les informations du client depuis customer-service

On a récupérer les détails des produit depuis inventory-service

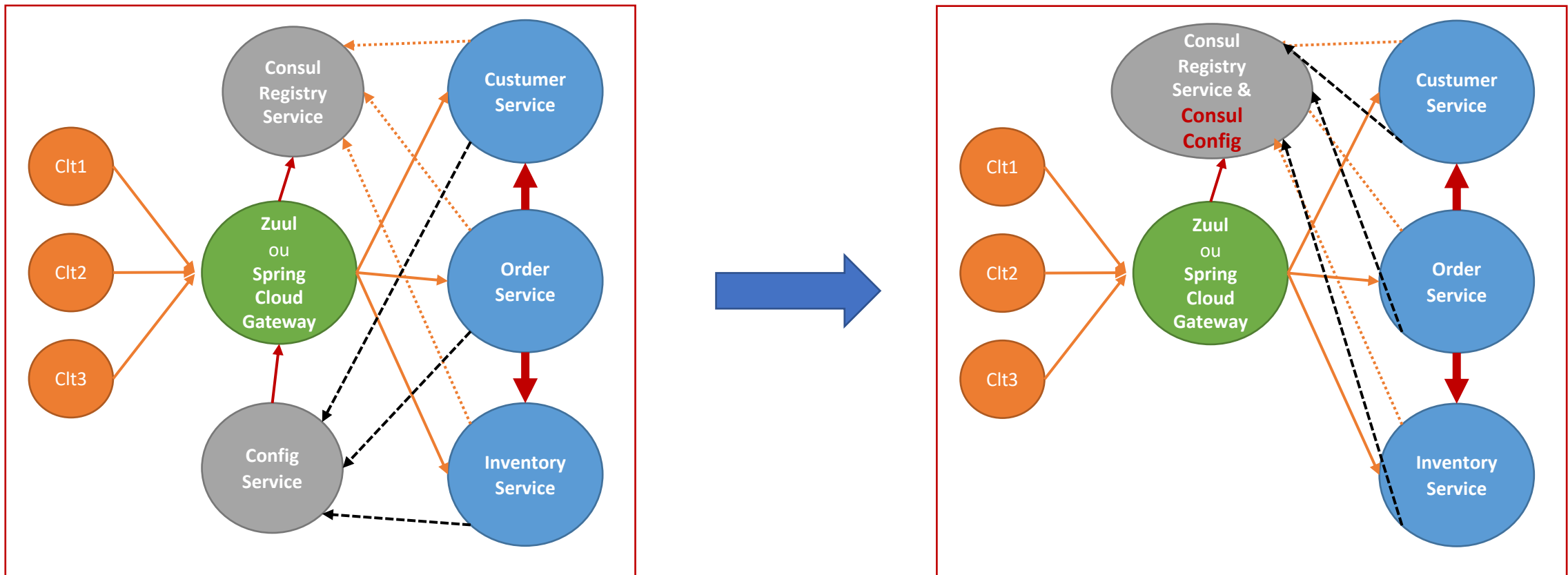
```
localhost:9999/order-service/fullOrder/6

{
  id: 6,
  createdAt: "2023-11-19T14:38:23.328+00:00",
  status: "CREATED",
  customerId: 1,
  customer: {
    id: 1,
    name: "aaaaaa",
    email: "aaaa@gmail.com"
  },
  productItems: [
    {
      id: 5,
      productId: 1,
      product: {
        id: 1,
        name: "screen",
        price: 500.0,
        quantity: 12
      },
      price: 500.0,
      quantity: 9,
      discount: 0.5929991802491204
    },
    {
      id: 6,
      productId: 3,
      product: {
        id: 3,
        name: "battery",
        price: 600.0,
        quantity: 15
      },
      price: 600.0,
      quantity: 6,
      discount: 0.1996214552103931
    }
  ]
}
```

Application:

Utiliser **Consul Config** au lieu de **Spring cloud config**

Consul Config est un outil puissant de configuration fourni par HashiCorp, conçu pour simplifier et automatiser la gestion des configurations dans les environnements informatiques modernes. Il fait partie de l'écosystème Consul,



Application: Consul Config vs Spring cloud config

Consul Config et Spring Cloud Config sont deux outils conçus pour gérer la configuration dans des systèmes distribués, mais ils sont associés à des piles technologiques différentes et présentent des différences en termes d'architecture et de fonctionnalités.

Pile technologique :

- Consul Config : Il fait partie de l'écosystème Consul fourni par HashiCorp, qui comprend des outils pour la découverte de services, la vérification de l'état de santé et le stockage de clés-valeurs. Consul Config est étroitement intégré à Consul, qui est une solution de système distribué et de maillage de services.
- Spring Cloud Config : Il fait partie du framework Spring Cloud, un ensemble d'outils et de bibliothèques pour la création de microservices dans l'écosystème Java. Spring Cloud Config peut être utilisé avec divers backends, notamment Git, Subversion ou un système de fichiers local.

Intégration :

- Consul Config : S'intègre de manière transparente à Consul pour la découverte de services et d'autres fonctionnalités. Il est bien adapté aux environnements où Consul est déjà utilisé pour l'orchestration et la découverte de services.
- Spring Cloud Config : S'intègre à l'écosystème Spring, fournissant une gestion de la configuration pour les applications Spring Boot. Il fait partie de la suite Spring Cloud, qui comprend des outils pour la découverte de services, l'équilibrage de charge, et plus encore.

Application: Consul Config vs Spring cloud config

Backend :

- Consul Config : Utilise généralement Consul comme backend pour le stockage de la configuration. Consul fournit un stockage distribué de clés-valeurs qui peut être utilisé pour stocker des données de configuration.
- Spring Cloud Config : Prend en charge divers backends, notamment des systèmes de contrôle de version comme Git et des systèmes de fichiers. Cette flexibilité permet aux équipes de choisir un backend qui correspond à leur infrastructure et à leurs flux de travail existants.

Prise en charge des langages :

- Consul Config : Bien que Consul lui-même ne soit pas limité à un langage spécifique, il peut être utilisé avec des applications écrites dans divers langages de programmation.
- Spring Cloud Config : Principalement conçu pour les applications Java, en particulier celles construites avec le framework Spring.

Communauté et écosystème :

- Consul Config : Bénéficie d'une communauté forte et fait partie de l'écosystème Consul plus vaste, qui comprend des outils supplémentaires pour la gestion des réseaux et de l'infrastructure.
- Spring Cloud Config : Profite de l'importante communauté et de l'écosystème Spring. Spring Cloud offre un ensemble complet d'outils pour la création et le déploiement d'applications cloud-native.

Application:

Dans cette partie du TP, nous allons tester Consul Config. Nous allons créer un nouveau micro-service appelé «billing-service». Au lieu d'utiliser Spring Cloud Config comme pour les autres, nous allons utiliser Consul Config.

Project		Language	
<input type="radio"/> Gradle - Groovy <input type="radio"/> Gradle - Kotlin <input checked="" type="radio"/> Java <input type="radio"/> Kotlin <input type="radio"/> Groovy			
<input checked="" type="radio"/> Maven			
Spring Boot			
<input type="radio"/> 3.2.0 (SNAPSHOT) <input type="radio"/> 3.2.0 (RC2) <input type="radio"/> 3.1.6 (SNAPSHOT) <input type="radio"/> 3.1.5			
<input type="radio"/> 3.0.13 (SNAPSHOT) <input type="radio"/> 3.0.12 <input type="radio"/> 2.7.18 (SNAPSHOT) <input checked="" type="radio"/> 2.7.17			
Project Metadata			
Group	org.sid		
Artifact	billing-service		
Name	billing-service		
Description	Demo project for Spring Boot		
Package name	org.sid.biling-service		
Packaging	<input checked="" type="radio"/> Jar <input type="radio"/> War		
Java	<input type="radio"/> 21 <input checked="" type="radio"/> 17 <input type="radio"/> 11 <input type="radio"/> 8		

Selected dependencies:







- **Spring Web**: Build web, including RESTful, applications using Spring MC. Uses Apache Tomcat as the default embedded container.
- **Spring Data JPA**: Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
- **H2 Database** : Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.
- **Rest Repositories**: Exposing Spring Data repositories over REST via Spring Data REST.
- **Lombok**: Java annotation library which helps to reduce boilerplate code.
- **Spring Boot DevTools**: Provides fast application restarts, LiveReload, and configurations for enhanced development experience.
- **Spring Boot Actuator**: Supports built in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc.
- **Consul Discovery** : Service discovery with Hashicorp Consul.
- **Consul Configuration** : Enable and configure the common patterns inside your application and build large distributed systems with Hashicorp's Consul. The patterns provided include Service Discovery, Distributed Configuration and Control Bus.

Application: Biling-service

Ajouter biling-service a votre workspace



A screenshot of a workspace sidebar. The sidebar is a vertical list of items, each preceded by a right-pointing chevron icon. The items are: a folder icon followed by 'billing-service ~/Documents/EMSI/2023-2024 se', a folder icon followed by 'config-service ~/Documents/EMSI/2023-2024 se', a folder icon followed by 'customer-service ~/Documents/EMSI/2023-2024', a folder icon followed by 'ecom-ems ~/Documents/EMSI/2023-2024 seme', a folder icon followed by 'gateway-service ~/Documents/EMSI/2023-2024', a folder icon followed by 'inventory-service ~/Documents/EMSI/2023-2024', a folder icon followed by 'order-service ~/Documents/EMSI/2023-2024 ser', a book icon followed by 'External Libraries', and a menu icon followed by 'Scratches and Consoles'. The 'billing-service' item is highlighted with a light blue background.

- >  **billing-service** ~/Documents/EMSI/2023-2024 se
- >  **config-service** ~/Documents/EMSI/2023-2024 se
- >  **customer-service** ~/Documents/EMSI/2023-2024
- >  **ecom-ems** ~/Documents/EMSI/2023-2024 seme
- >  **gateway-service** ~/Documents/EMSI/2023-2024
- >  **inventory-service** ~/Documents/EMSI/2023-2024
- >  **order-service** ~/Documents/EMSI/2023-2024 ser
- >  External Libraries
- >  Scratches and Consoles

Application: Billing-service

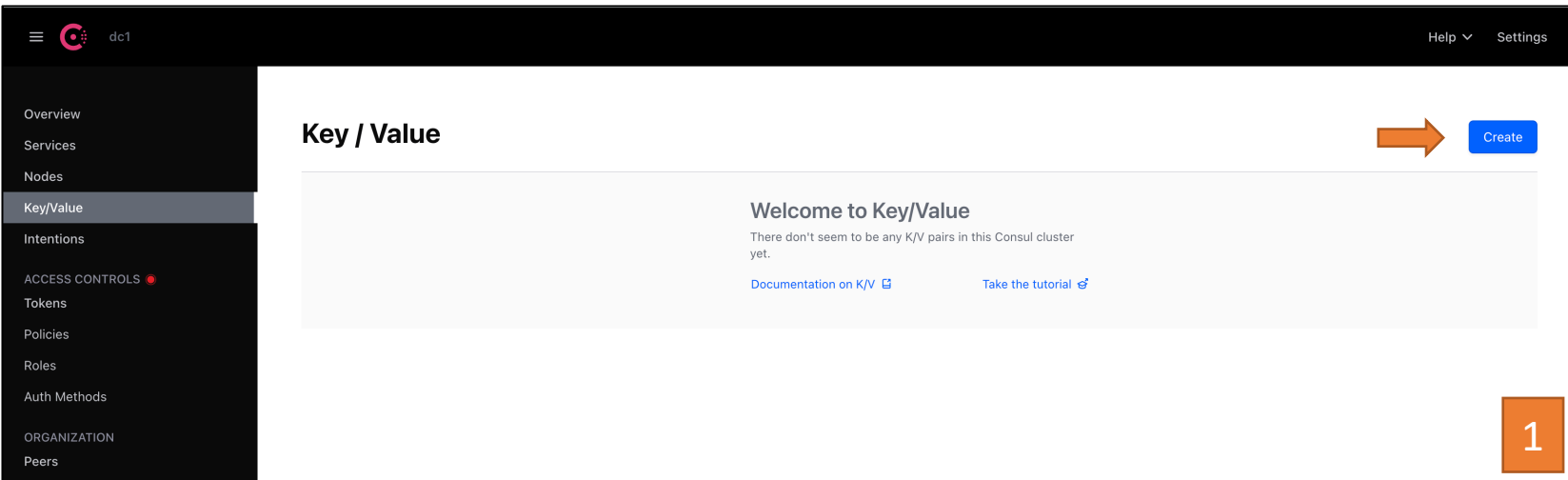
L'interface de Consul

The screenshot shows the Consul web interface. The left sidebar contains the following menu items: Overview, Services, Nodes, Key/Value, Intentions, ACCESS CONTROLS (with a red dot), Tokens, Policies, Roles, Auth Methods, ORGANIZATION, and Peers. The 'Services' and 'Key/Value' items are highlighted with red boxes. Red arrows point from these boxes to 'Discovery service' and 'Configuration service' labels on the right. The main area displays the 'Services' page with a search bar and a list of six services, each with a green checkmark and '1 instance'.

Service	Status	Instances
consul	✓	1 instance
config-service	✓	1 instance
customer-service	✓	1 instance
gateway-service	✓	1 instance
inventory-service	✓	1 instance
order-service	✓	1 instance

Application: Billing-service

L'interface de Consul -> Ajouter un dossier billing-service



New Key / Value

Key or folder

To create a folder, end a key with `/`

2 - ajouter / pour indiquer que c'est un dossier

< Key / Values

New Key / Value

Key or folder

To create a folder, end a key with `/`

3- cree un autre dossier dans config

config

Type ▾

Name

4

Le nom du dossier doit avoir le même nom que le micro-service.

Application: Billing-service

config

Search Type ▾

Name

billing-service 4



< Key / Values

billing-service

Welcome to Key/Value

There don't seem to be any K/V pairs in this Consul cluster yet.

[Documentation on K/V](#) [Take the tutorial](#)

Create 5

Dans billing-service on peut crée nos paramètre

< Key / Values / billing-service

New Key / Value

Nom 6

Key or folder

token.accessTokenTimeout

To create a folder, end a key with /

Value valeur

1 56003

HCL

Code

Save Cancel

< Key / Values

billing-service

Search Type ▾

Name

token.accessTokenTimeout 7

On a cree un parametre :
token.accessTokenTimeout et on lui
a donne la valeur 56003

Application: Billing-service

Refaire les etapes 4 et 5 pour cree un parametre : token.refreshTokenTimeout et on lui a donne la valeur 540000

[<](#) Key / Values

billing-service

Type ▾

Name	
token.accessTokenTimeout	
token.refreshTokenTimeout	8

Application: Billing-service

Dans application.properties de billing-service

```
server.port=8084  
spring.application.name=billing-service  
spring.config.import=optional:consul:
```

Automatiquement, il se connectera au port par défaut 8500. Si l'on souhaite préciser le port explicitement, on doit utiliser :

```
spring.config.import=optional:consul:http://localhost:numero-du-port/
```

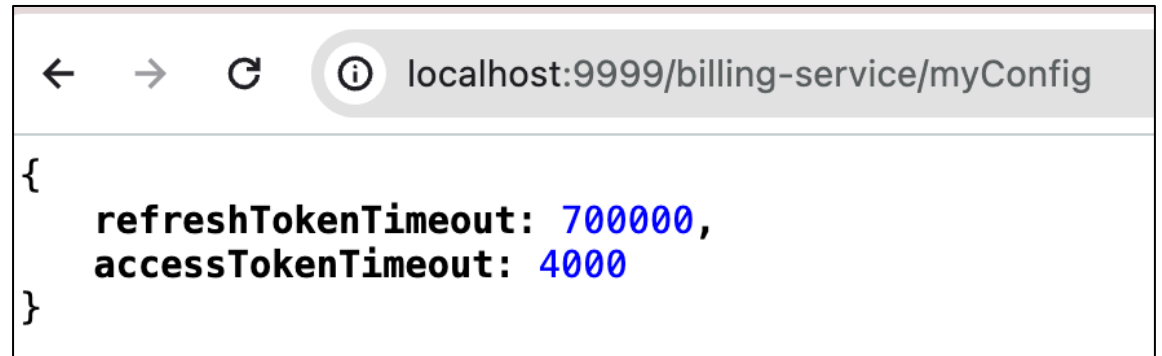
Application: Billing-service

Maintenant on va récupérer rapidement la configuration depuis Consul Config:

1. Crée la classe **ConsulConfigRestController** dans billing-service

```
@RestController
@RefreshScope
public class ConsulConfigRestController {
    1 usage
    @Value("${token.accessTokenTimeout}")
    private Long accessTokenTimeout;
    1 usage
    @Value("${token.refreshTokenTimeout}")
    private Long refreshTokenTimeout;
    no usages
    @GetMapping("/myConfig")
    public Map<String, Object> myConfig(){
        return Map.of( k1: "accessTokenTimeout", accessTokenTimeout,
                       k2: "refreshTokenTimeout", refreshTokenTimeout);
    }
}
```

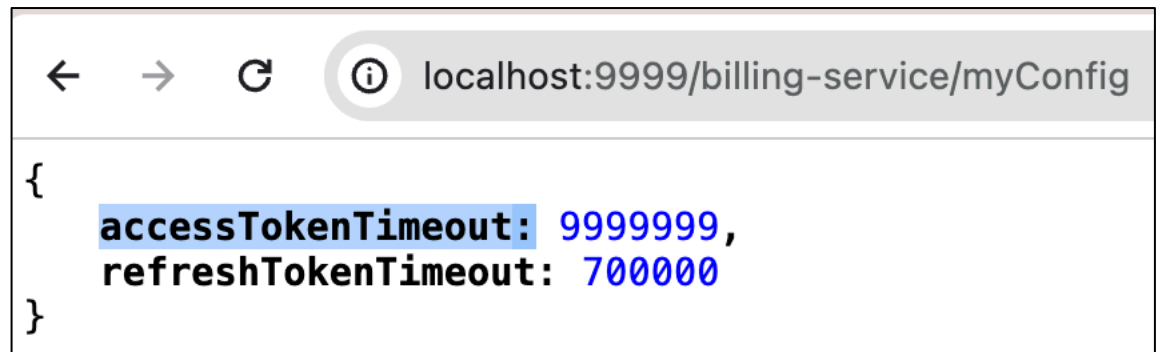
2. On teste



```
localhost:9999/billing-service/myConfig

{
  refreshTokenTimeout: 700000,
  accessTokenTimeout: 4000
}
```

3. Changer la valeur de **accessTokenTimeout** dans consul.
La nouvelle configuration est automatiquement prise en charge



```
localhost:9999/billing-service/myConfig

{
  accessTokenTimeout: 9999999,
  refreshTokenTimeout: 700000
}
```

Application: Billing-service

Maintenant on va recupere rapidement la configuration depuis Consul Config:

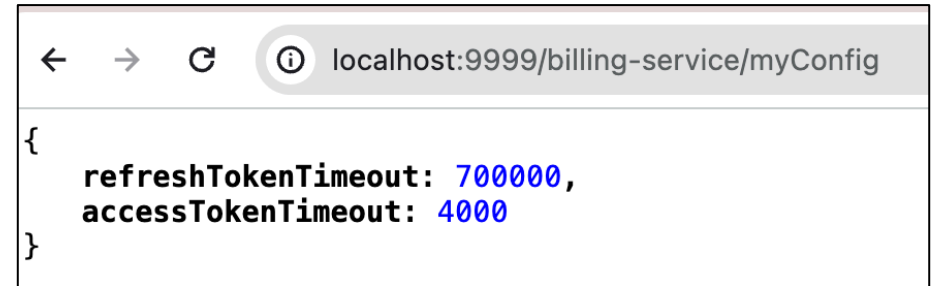
1. Cree la classe **MyConsulConfig** dans billing-service

```
@Component
@ConfigurationProperties(prefix = "token")
@Data
public class MyConsulConfig {
    private long accessTokenTimeout;
    private long refreshTokenTimeout;
}
```

2. Cree la classe **ConsulConfigRestController** dans billing-service

```
@RestController
public class ConsulConfigRestController {
    1 usage
    @Autowired
    private MyConsulConfig myConsulConfig;
    no usages
    @GetMapping("/myConfig")
    public MyConsulConfig myConfig(){
        return myConsulConfig;
    }
}
```

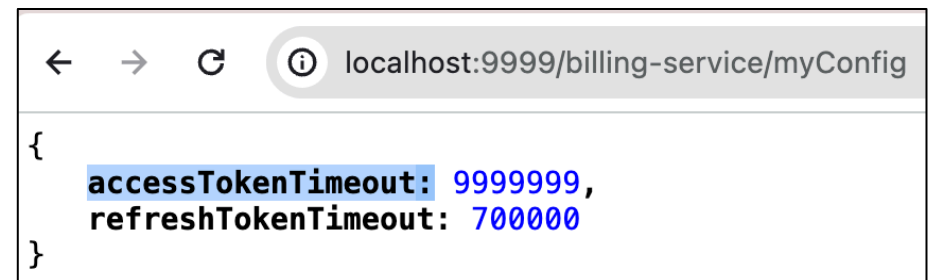
3. On teste



```
localhost:9999/billing-service/myConfig
{
  refreshTokenTimeout: 700000,
  accessTokenTimeout: 4000
}
```

4. Changer la valeur de **accessTokenTimeout** dans consul.

La nouvelle configuration est automatiquement prise en charge



```
localhost:9999/billing-service/myConfig
{
  accessTokenTimeout: 9999999,
  refreshTokenTimeout: 700000
}
```