

TP 9 : Microservices avec Spring Boot et Spring Cloud



Objectifs du TP

- Création de microservices avec Spring Boot et Spring Cloud
- Déploiement d'un microservices sur plusieurs instances

Outils et Versions

- ♦ **Spring Boot** [<https://projects.spring.io/spring-boot/>] Version : **3.0.0** ou plus
- ♦ **Spring Cloud** [<http://projects.spring.io/spring-cloud/>] Version **2022.0.0** ou plus
- ♦ **Java** Version 17
- ♦ **Maven** [<https://maven.apache.org/>] Version 4.0.0
- ♦ **IntelliJ IDEA** (Ultimate Edition)

Architecture Microservices

Présentation

Une architecture Microservices (MSA) représente un moyen de concevoir les applications comme ensemble de microservices indépendamment déployables. Ces microservices doivent de préférence être organisés autour des compétences métier, de déploiement automatique, d'extrémités intelligentes et de contrôle décentralisé des technologies et des données.

Architecture Proposée

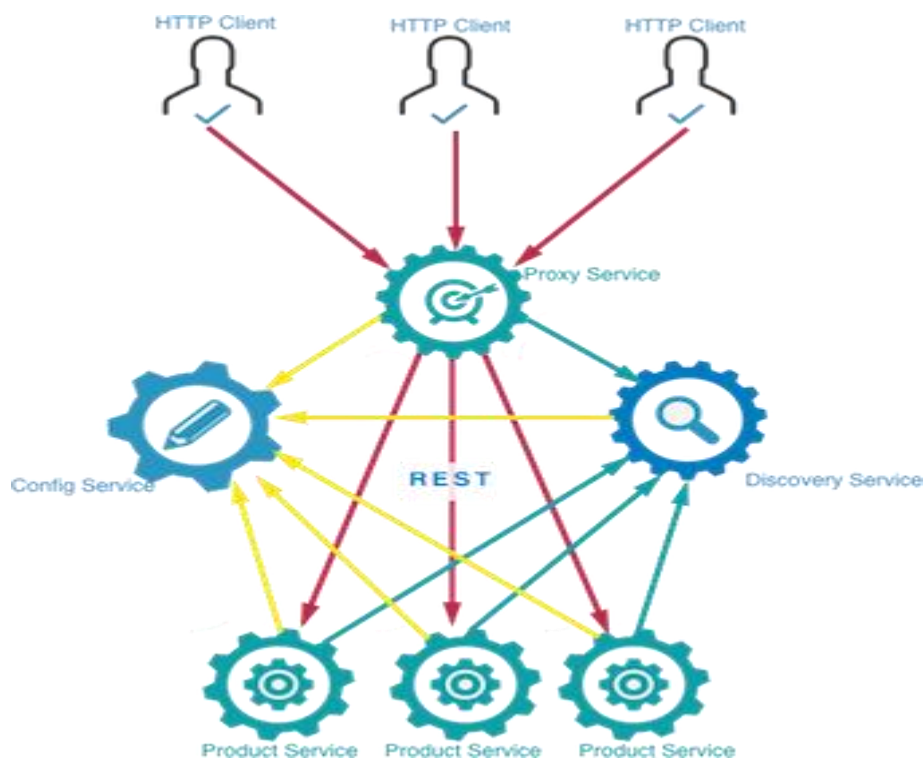
L'objectif de ce travail est de montrer comment créer plusieurs services indépendamment déployables qui communiquent entre eux, en utilisant les facilités offertes par Spring Cloud et Spring Boot. **Spring Cloud** [<http://projects.spring.io/spring-cloud/>] fournit des outils pour les développeurs pour construire rapidement et facilement des patrons communs de systèmes répartis (tel que des services de configuration, de découverte ou de routage intelligent...etc.).

Spring Boot [<https://projects.spring.io/spring-boot/>] permet de son côté de construire des applications Spring rapidement aussi rapidement que possible, en minimisant au maximum le temps de configuration, d'habitude pénible, des applications Spring.

Nous allons donc créer les microservices suivants :

1. **Product Service** : Service principal, qui offre une API REST pour lister une liste de produits.
2. **Config Service** : Service de configuration, dont le rôle est de centraliser les fichiers de configuration des différents microservices dans un endroit unique.
3. **Proxy Service** : Passerelle se chargeant du routage d'une requête vers l'une des instances d'un service, de manière à gérer automatiquement la distribution de charge.
4. **Discovery Service** : Service permettant l'enregistrement des instances des services en vue d'être découvertes par d'autres services.

L'architecture résultante aura l'allure suivante :



Création des Microservices

Microservice **ProductService**

Nous commençons par créer le service principal : *Product Service*.

Suivre les étapes suivantes pour créer le microservice *ProductService* :

- Sous le répertoire *src/main/java* et dans le package package **com.ecommerce.productservice**, créer la classe *Product* suivante:

```
package com.ecommerce.productservice;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import java.io.Serializable;

@Entity
public class Product implements Serializable {
    @Id
    @GeneratedValue
    private int id;
    private String name;
    public Product() {
    }
    public Product(String name) {
        this.name = name;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

Cette classe est annotée avec JPA, pour stocker ensuite les objets *Product* dans la base de données H2 grâce à Spring Data. Pour cela, créer l'interface ***ProductRepository*** dans le même package:

```
package com.ecommerce.productservice;
import org.springframework.data.jpa.repository.JpaRepository;

public interface ProductRepository extends JpaRepository<Product ,
Integer> {
}
```

Pour insérer les objets dans la base, nous utiliserons l'objet *Stream*. Pour cela, nous allons créer la classe *DummyDataCLR* :

```
package com.ecommerce.productservice;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;import
java.util.stream.Stream;

@Component
class DummyDataCLR implements CommandLineRunner {

    @Override
    public void run(String... strings) throws Exception {
        Stream.of("Pencil", "Book", "Eraser").forEach(s->productRepository.save(new
            Product(s)));
        productRepository.findAll().forEach(s->System.out.println(s.getName()));
    }

    @Autowired
    private ProductRepository productRepository;
}
```

Nous remarquons ici que le *productRepository* sera instancié automatiquement grâce au mécanisme d'injection de dépendances, utilisé par Spring.

Dans le pom.xml ajouter :

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

Lancer la classe principale. Une base de données H2 sera créée et le *CommandLineRunner* se chargera de lui injecter les données.

Le résultat sur la console devrait ressembler à ce qui suit :

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_151.jdk/Contents/Home/bin/java ...
obj[68702]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_151.jdk/Contents/Home/bin/java (0x10e9644c0) and /Library/Java/JavaVirtualMachines/jdk1.
2017-11-19 10:32:07.614 INFO 68702 --- [main] s.c.a.AnnotationConfigApplicationContext : Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@4099c
2017-11-19 10:32:07.837 INFO 68702 --- [main] f.a.AutowiredAnnotationBeanPostProcessor : JSR-330 'javax.inject.Inject' annotation found and supported for autowiring
2017-11-19 10:32:07.884 INFO 68702 --- [main] trationDelegatesBeanPostProcessorChecker : Bean 'configurationPropertiesRebinderAutoConfiguration' of type [org.springframework.clou

  ____ _
 / ___ \| | | |
/ /___ \| | | |
\___)___|_|_|_|
:: Spring Boot :: (v1.5.8.RELEASE)

2017-11-19 10:32:08.155 INFO 68702 --- [main] c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at: http://localhost:8888
2017-11-19 10:32:08.350 WARN 68702 --- [main] c.c.c.ConfigServicePropertySourceLocator : Could not locate PropertySource: I/O error on GET request for "http://localhost:8888/appl
2017-11-19 10:32:08.351 INFO 68702 --- [main] t.i.t.p.ProductServiceApplication : No active profile set, falling back to default profiles: default
2017-11-19 10:32:08.360 INFO 68702 --- [main] ationConfigEmbeddedWebApplicationContext : Refreshing org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicati
2017-11-19 10:32:08.967 INFO 68702 --- [main] o.s.b.f.s.DefaultListableBeanFactory : Overriding bean definition for bean 'httpRequestHandlerAdapter' with a different definiti
2017-11-19 10:32:09.015 INFO 68702 --- [main] o.s.b.f.s.DefaultListableBeanFactory : Overriding bean definition for bean 'managementServletContext' with a different definiti
2017-11-19 10:32:09.223 INFO 68702 --- [main] o.s.cloud.context.scope.GenericScope : BeanFactory id=6ec89c33-e335-31d1-834a-bf20bc24577b
2017-11-19 10:32:09.251 INFO 68702 --- [main] f.a.AutowiredAnnotationBeanPostProcessor : JSR-330 'javax.inject.Inject' annotation found and supported for autowiring
2017-11-19 10:32:09.376 INFO 68702 --- [main] trationDelegatesBeanPostProcessorChecker : Bean 'org.springframework.transaction.annotation.ProxyTransactionManagementConfiguration'
2017-11-19 10:32:09.438 INFO 68702 --- [main] trationDelegatesBeanPostProcessorChecker : Bean 'org.springframework.cloud.netflix.metrics.MetricsInterceptorConfiguration$MetricsRe
2017-11-19 10:32:09.458 INFO 68702 --- [main] trationDelegatesBeanPostProcessorChecker : Bean 'org.springframework.cloud.autoconfigure.ConfigurationPropertiesRebinderAutoConfigur
2017-11-19 10:32:09.715 INFO 68702 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat initialized with port(s): 8080 (http)
2017-11-19 10:32:09.724 INFO 68702 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2017-11-19 10:32:09.725 INFO 68702 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache Tomcat/8.5.23
2017-11-19 10:32:09.812 INFO 68702 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext

( ... )

2017-11-19 10:32:13.003 INFO 68702 --- [main] o.s.c.support.DefaultLifecycleProcessor : Starting beans in phase 0
2017-11-19 10:32:13.108 INFO 68702 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-11-19 10:32:13.172 INFO 68702 --- [main] o.h.h.i.QueryTranslatorFactoryInitiator : HH000397: Using ASTQueryTranslatorFactory

Pencil
Book
Eraser

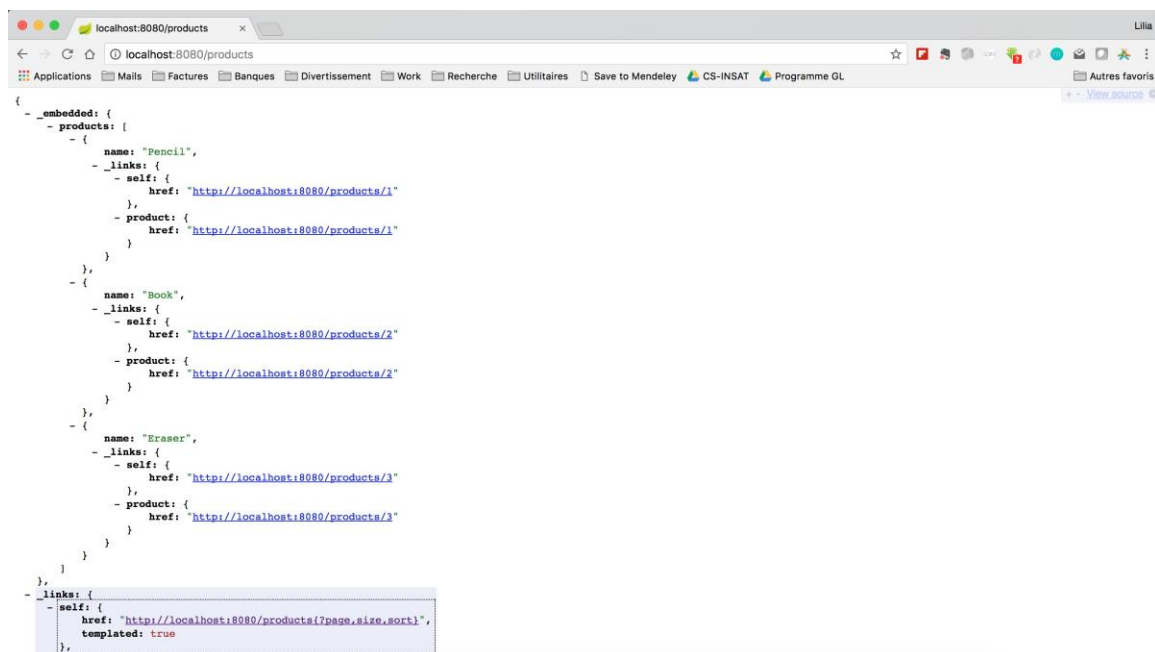
2017-11-19 10:32:13.237 INFO 68702 --- [main] t.i.t.p.ProductServiceApplication : Started ProductServiceApplication in 5.955 seconds (JVM running for 6.519)
2017-11-19 10:32:13.556 INFO 68702 --- [6]-192.168.1.46] c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at: http://localhost:8888
2017-11-19 10:32:13.662 WARN 68702 --- [6]-192.168.1.46] c.c.c.ConfigServicePropertySourceLocator : Could not locate PropertySource: I/O error on GET request for "http://localhost:8888/appl
```

Pour tester votre application, ouvrir la page <http://localhost:8080> sur le navigateur. Vous obtiendrez (si tout se passe bien)le résultat suivant:



```
{
  "_links": {
    "products": {
      href: "http://localhost:8080/products{?page,size,sort}",
      templated: true
    },
    "profile": {
      href: "http://localhost:8080/profile"
    }
  }
}
```

Si vous naviguez vers la page <http://localhost:8080/products> , vous verrez la liste des produits, injectés par leCLR, comme suit:



```
{
  "_embedded": {
    "products": [
      {
        name: "Pencil",
        "_links": {
          "self": {
            href: "http://localhost:8080/products/1"
          },
          "product": {
            href: "http://localhost:8080/products/1"
          }
        }
      },
      {
        name: "Book",
        "_links": {
          "self": {
            href: "http://localhost:8080/products/2"
          },
          "product": {
            href: "http://localhost:8080/products/2"
          }
        }
      },
      {
        name: "Eraser",
        "_links": {
          "self": {
            href: "http://localhost:8080/products/3"
          },
          "product": {
            href: "http://localhost:8080/products/3"
          }
        }
      }
    ]
  },
  "_links": {
    "self": {
      href: "http://localhost:8080/products{?page,size,sort}",
      templated: true
    }
  }
}
```

Pour voir les informations relatives à un seul produit, il suffit de connaître son ID: <http://localhost:8080/products/1> par exemple.

Pour rajouter une fonctionnalité de recherche par nom, par exemple, modifier l'interface *ProductRepository*, comme suit:


```

package com.ecommerce.productservice;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import
org.springframework.data.rest.core.annotation.RepositoryRestResource;

@RepositoryRestResource
public interface ProductRepository extends JpaRepository<Product ,
Integer> {

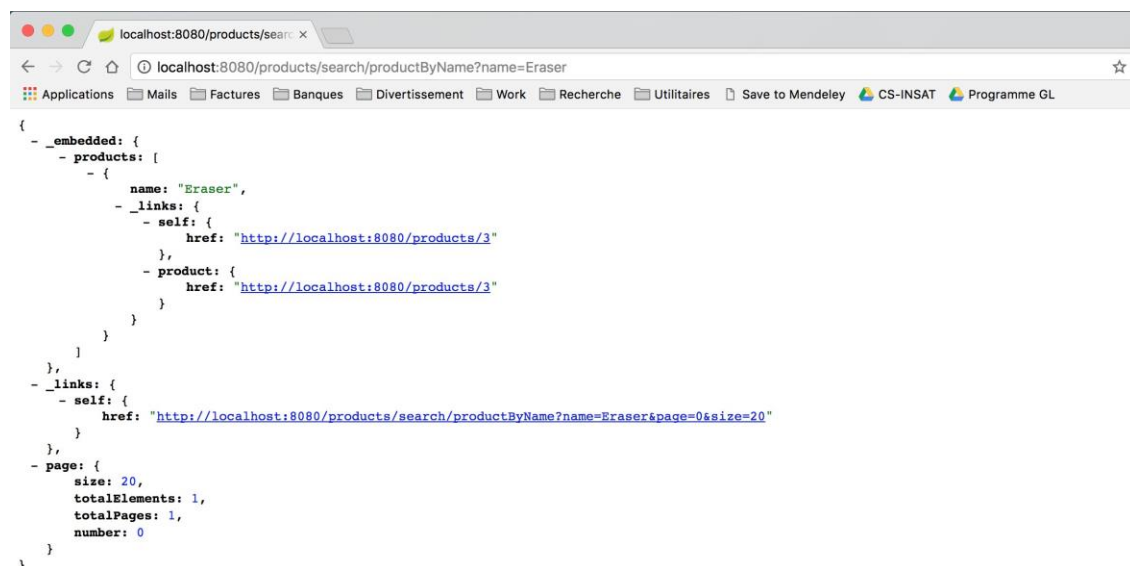
    @Query("select p from Product p where p.name like :name")
    public Page<Product> productByName(@Param("name") String mc
        , Pageable pageable);
}

```

Pour tester cette fonctionnalité de recherche, aller au lien

<http://localhost:8080/products/search/productByName?name=Eraser>

Le résultat obtenu sera le suivant : (n'oublier pas de relancer l'exécution)

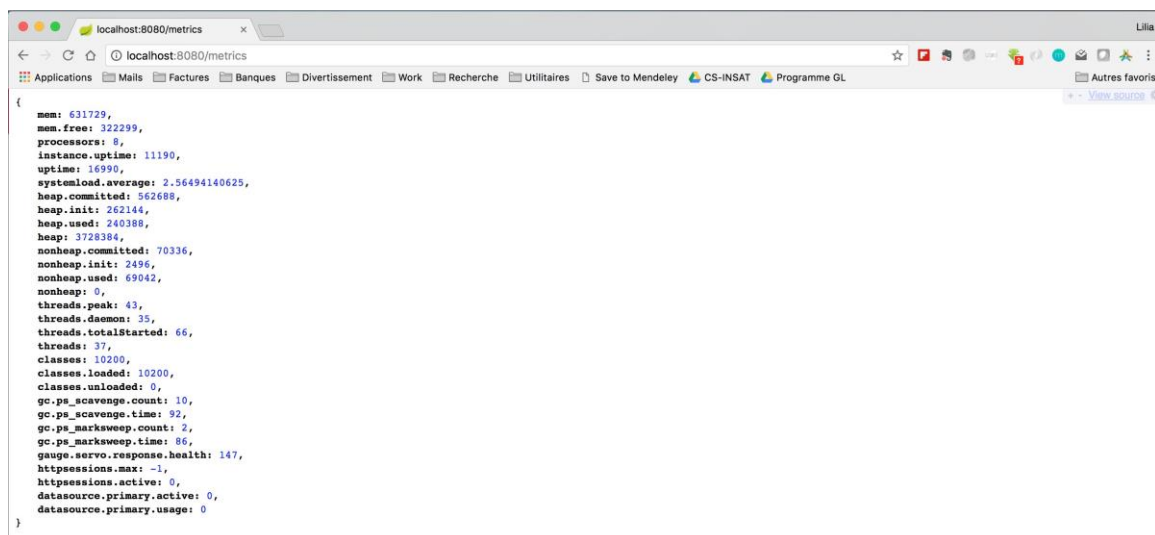


La dépendance *Actuator* qui a été rajoutée au projet permet d'afficher des informations sur votre API REST sans avoir à implémenter explicitement la

fonctionnalité. Par exemple, si vous allez vers <http://localhost:8080/actuator/metrics> vous pourrez avoir plusieurs informations sur le microservice, tel que le nombre de threads, la capacité mémoire, la classe chargée en mémoire, etc. Mais d'abord, rajouter les deux lignes suivantes au fichier `src/main/resources/application.properties` pour afficher des informations plus détaillées sur l'état du service

```
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=always
```

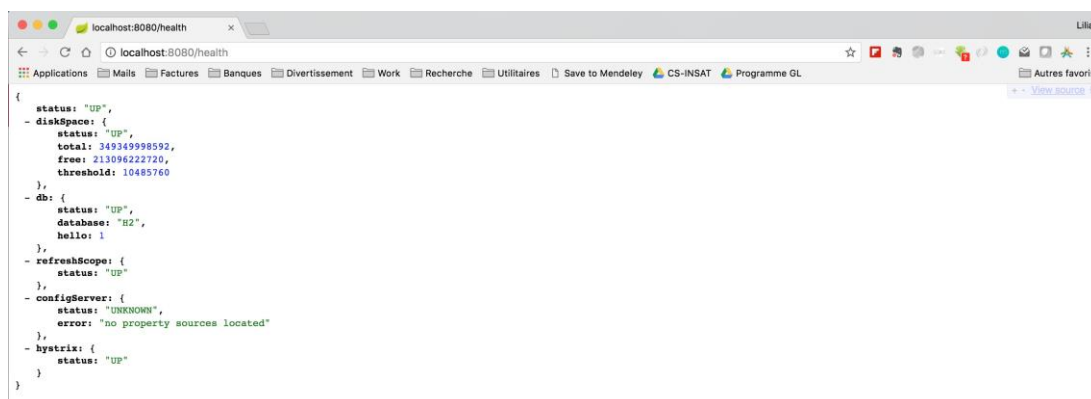
Relancer le projet. Le résultat obtenu en exécutant <http://localhost:8080/actuator/metrics> sera comme suit:



The screenshot shows a web browser window with the address bar set to `localhost:8080/metrics`. The page displays a JSON object containing various system and application metrics. The metrics include memory usage (mem, mem.free), processor count (processors), instance uptime (instance.uptime), system uptime (uptime), system load average (systemload.average), heap memory (heap.committed, heap.init, heap.used, heap.max), non-heap memory (nonheap.committed, nonheap.init, nonheap.used, nonheap.max), thread statistics (threads, threads.daemon, threads.totalStarted), class statistics (classes, classes.loaded, classes.unloaded), garbage collection statistics (gc.ps.scavenge, gc.ps.markweep), gauge response (gauge.servo.response.health), HTTP sessions (httpsessions.max, httpsessions.active), and data source statistics (datasource.primary.active, datasource.primary.usage).

```
{
  mem: 631729,
  mem.free: 322299,
  processors: 8,
  instance.uptime: 11190,
  uptime: 16990,
  systemload.average: 2.56494140625,
  heap.committed: 562688,
  heap.init: 262144,
  heap.used: 240388,
  heap.max: 3728384,
  nonheap.committed: 70336,
  nonheap.init: 2496,
  nonheap.used: 69042,
  nonheap.max: 0,
  threads: 37,
  threads.daemon: 35,
  threads.totalStarted: 66,
  classes: 10200,
  classes.loaded: 10200,
  classes.unloaded: 0,
  gc.ps.scavenge.count: 10,
  gc.ps.scavenge.time: 92,
  gc.ps.markweep.count: 2,
  gc.ps.markweep.time: 86,
  gauge.servo.response.health: 147,
  httpsessions.max: -1,
  httpsessions.active: 0,
  datasource.primary.active: 0,
  datasource.primary.usage: 0
}
```

Les informations sur l'état du service sont affichées grâce à <http://localhost:8080/actuator/health>

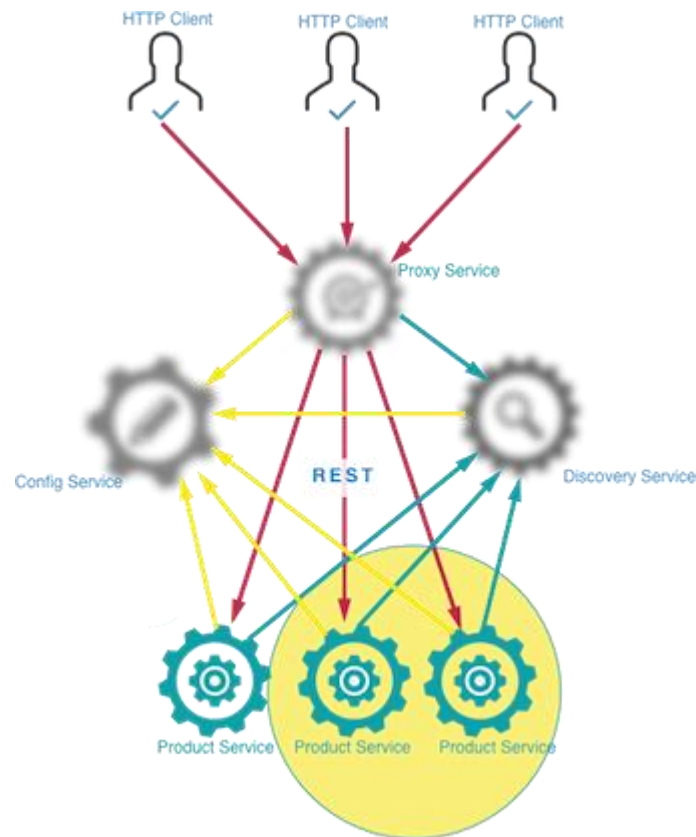


The screenshot shows a web browser window with the address bar set to `localhost:8080/health`. The page displays a JSON object representing the health status of the service. The status is "UP". The response includes details for disk space (status: "UP", total: 349349998592, free: 213096222720, threshold: 10485760), database (status: "UP", database: "H2", hello: 1), refresh scope (status: "UP"), config server (status: "UNKNOWN", error: "no property sources located"), and hystrix (status: "UP").


```
{
  status: "UP",
  - diskSpace: {
    status: "UP",
    total: 349349998592,
    free: 213096222720,
    threshold: 10485760
  },
  - db: {
    status: "UP",
    database: "H2",
    hello: 1
  },
  - refreshScope: {
    status: "UP"
  },
  - configServer: {
    status: "UNKNOWN",
    error: "no property sources located"
  },
  - hystrix: {
    status: "UP"
  }
}
```

Plusieurs Instances du Microservice **ProductService**

Nous allons maintenant créer d'autres instances du même service et les déployer sur des ports différents.



Pour lancer plusieurs instances du service *ProductService*, nous allons définir plusieurs configurations avec des numéros de port différents. Pour cela:

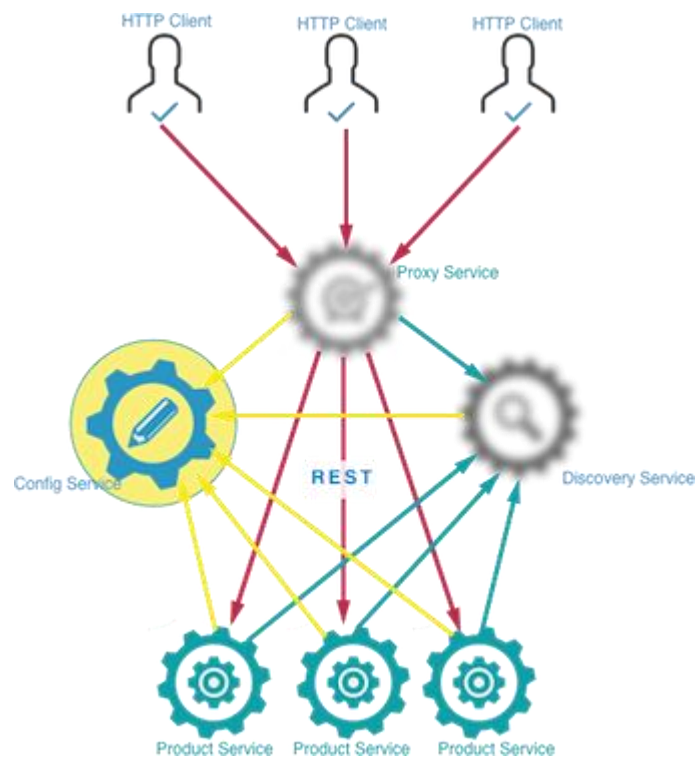
- Aller à *Run->Edit Configurations*, et copier la configuration *ProductServiceApplication* en la sélectionnant dans la barre latérale, et en cliquant sur l'icône . Une nouvelle configuration sera créée.
- Changer son nom : *ProductServiceApplication:8081*
- Ajouter dans la case *Program Arguments* l'argument suivant :

```
--server.port=8081
```

- Lancer la configuration. Un nouveau service sera disponible à l'adresse : <http://localhost:8081>
- Refaire les mêmes étapes pour créer une instance du service tournant sur le port 8082.

Microservice **ConfigService**

Dans une architecture microservices, plusieurs services s'exécutent en même temps, sur des processus différents, avec chacun sa propre configuration et ses propres paramètres. Spring Cloud Config fournit un support côté serveur et côté client pour externaliser les configurations dans un système distribué. Grâce au service de configuration, il est possible d'avoir un endroit centralisé pour gérer les propriétés de chacun de ces microservices.



Pour cela, aller au site start.spring.io [<http://start.spring.io>], et créer un projet avec les caractéristiques suivantes :

- Projet Maven avec Java et Spring Boot version 3.0.0
- Group :com.ecommerce
- Artifact : config-service

Ajouter les dépendances appropriées, comme indiqué sur la figure suivante :

Project <input type="radio"/> Gradle - Groovy <input type="radio"/> Gradle - Kotlin <input checked="" type="radio"/> Maven	Language <input checked="" type="radio"/> Java <input type="radio"/> Kotlin <input type="radio"/> Groovy	Dependencies ADD DEPENDENCIES... CTRL + B <input checked="" type="checkbox"/> Config Server <input checked="" type="checkbox"/> SPRING CLOUD CONFIG Central management for configuration via Git, SVN, or HashiCorp Vault.
Spring Boot <input type="radio"/> 3.0.1 (SNAPSHOT) <input checked="" type="radio"/> 3.0.0 <input type="radio"/> 2.7.8 (SNAPSHOT) <input type="radio"/> 2.7.7		

- Ouvrir le projet dans une autre instance d'IntelliJ IDEA.
- Pour exposer un service de configuration, utiliser l'annotation `@EnableConfigServer` pour la classe `ConfigServiceApplication`, comme suit :

```
package com.ecommerce.configservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;

@EnableConfigServer
@SpringBootApplication
public class ConfigServiceApplication { public

    static void main(String[] args) {

SpringApplication.run(ConfigServiceApplication.class, args);
    }
}
```

Pour paramétrer ce service de configuration :

- Créer le répertoire *myConfig* à l'arborescence *src/main/resources*
- Ajouter dans son fichier *application.properties* les valeurs suivantes:

```
server.port=8888
spring.cloud.config.server.git.uri=chemin vers le dossier myConfig
```

Ceci indique que le service de configuration sera lancé sur le port 8888 et que le répertoire contenant les fichiers de configuration se trouve dans le répertoire *myConfig*.

Créer dans le répertoire « myconfig » le fichier *application.properties* dans lequel vous insérez l'instruction suivant

```
global=xxxxxx
```

Ce fichier sera partagé entre tous les microservices utilisant ce service de configuration.

Le répertoire de configuration doit être un répertoire git. Pour cela :

- Ouvrir le terminal avec IntelliJ et naviguer vers ce répertoire.
- Initialiser votre répertoire: `git init`
- Créer une entrée racine dans le repository: `git add .`
- Faire un commit: `git commit -m "add ."`

Revenir vers le projet *ProductService* et ajouter dans le fichier de configuration *application.properties*:

```
spring.application.name = product-service
spring.cloud.config.uri = http://localhost:8888
```

Comme le fichier *application.properties* dedans *myConfig* contient toutes les propriétés partagées des différents microservices, nous aurons besoins d'autres fichiers pour les propriétés spécifiques à un microservice. Pour cela:

- Créer dans le répertoire *myConfig* un fichier *product-service.yml* pour le service *ProductService*.



Attention

Le nom du fichier doit correspondre à la propriété *spring.application.name* que vous avez saisi dans le fichier *application.properties* de votre microservice!

- Ajouter les propriétés de votre service

```
spring:
  application:
    name: product-service
  eureka:
    instance:
      hostname: localhost
me: amine.Bahij@ecommerce.com
```

- Renommer le fichier *application.properties* de votre projet « *product-service* » en *bootstrap.properties*
- Lancer le microservice de configuration.
- En consultant l'url <http://localhost:8888/product-service/master>, nous remarquons l'ajout de la nouvelle propriété.

```
{
  name: "product-
service", profiles: [
    "master"
  ],
  label: null,
  version:
    "6e1ea61d706133e2d8b62f40c6b784192fb5
8e8a", state: null,
  propertySources: [
    {
      name:
        "file:...../src/main/resources/myConfig/product-
service.properties",
      source: {
        me: "amine.Bahij@ecommerce.com"
      }
    },
    {
      name:
        "file:.. ...../src/main/resources/myConfig/applica
tion.properties", source: {
        global: "xxxxx"
      }
    }
  ]
}
```

Nous allons maintenant définir un appel REST à cette propriété. Pour cela :

- Créer la classe *ProductRestService* dans le projet *product-service*. Son code ressemblera à ce qui suit:

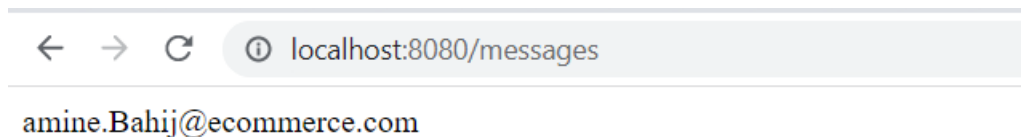
```
package com.ecommerce.productservice;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class ProductRestService {

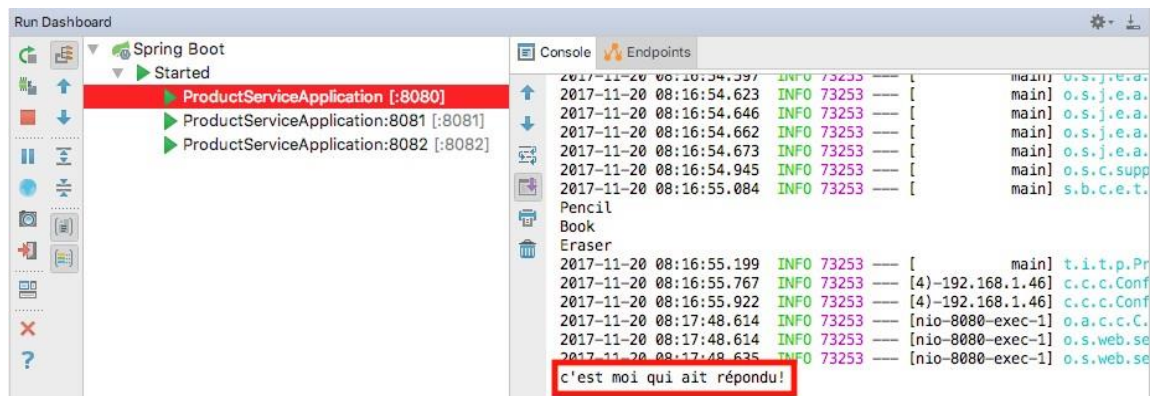
    @Value("${me}")
    private String me;

    @RequestMapping("/messages")
    public String tellMe(){
        System.out.println("c'est moi qui ai répondu!");
        return me;
    }
}
```

- Redémarrer les trois instances du service, puis appeler dans votre navigateur le service en tapant : <http://localhost:8080/messages> Vous verrez le résultat suivant sur le navigateur :

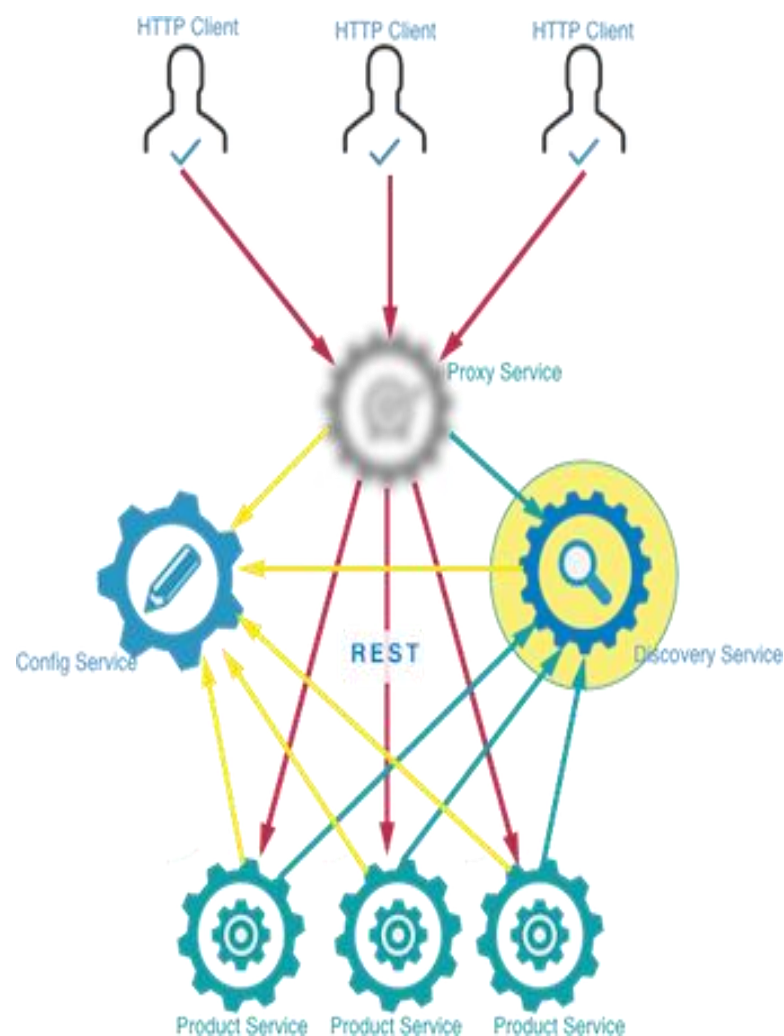


- Vous verrez le message suivant dans la console de l'instance du service lancée sur le port 8080:



Microservice **DiscoveryService**

Pour éviter un couplage fort entre microservices, il est fortement recommandé d'utiliser un service de découverte qui permet d'enregistrer les propriétés des différents services et d'éviter ainsi d'avoir à appeler un service directement. Au lieu de cela, le service de découverte fournira dynamiquement les informations nécessaires, ce qui permet d'assurer l'élasticité et la dynamique propres à une architecture microservices.



Pour réaliser cela, Netflix offre le service **Eureka Service Registration and Discovery** [<https://spring.io/guides/gs/service-registration-and-discovery/>], que nous allons utiliser dans notre application.

-
- Revenir à *Spring Initializr* et créer un nouveau projet Spring Boot intitulé *discovery-service* avec les dépendances *Eureka Server* et *Config Client*.
 - Lancer le projet avec IntelliJ.
 - Dans la classe *DiscoveryServiceApplication*, ajouter l'annotation *EnableEurekaServer* comme suit :

```
package com.ecommerce.discovery-service;

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

@EnableEurekaServer
@SpringBootApplication
public class DiscoveryServiceApplication {

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

- Renommer le fichier *application.properties* de votre projet « *discovery-service* » en *bootstrap.properties*
- Dans le *pom.xml* ajouter :

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

- Ajouter les propriétés suivantes dans son fichier *bootstrap.properties*.

```
spring.application.name=discovery-service
spring.cloud.config.uri=http://localhost:8888
```

- Dans le projet *config-service*, créer un fichier *discovery-service.yml* sous le répertoire *myConfig*.
- Ajouter les propriétés suivantes pour (1) définir le port par défaut du service de

découverte et (2) empêcher un auto-enregistrement du service Eureka.

```
server:
  port: 8761
  eureka:
    client:
      register-with-eureka: false
      fetch-registry: false
    server:
      waitTimeInMsWhenSyncEmpty: 0
  instance:
    hostname: localhost
```

- Ajouter à la classe `ProductServiceApplication` l'annotation `@EnableDiscoveryClient` pour que le microservice `ProductService` soit également enregistré dans le service de découverte.
- Redémarrer les trois instances de services *ProductService* et actualiser la fenêtre de *Eureka*, vous verrez qu'un seul service est déclaré, avec trois adresses différentes.

Pour consulter le service Eureka, aller à <http://localhost:8761> l'interface suivante s'affiche :

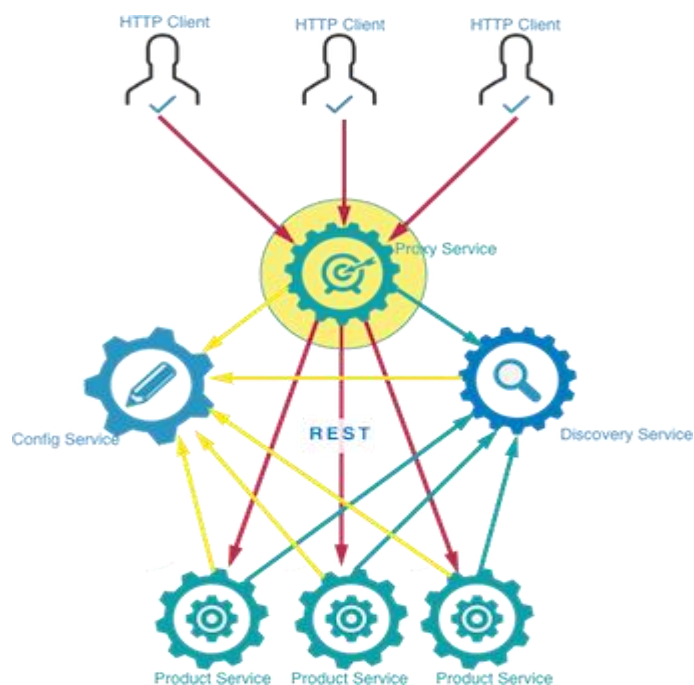


The screenshot shows the Eureka web interface. At the top, it says "Instances currently registered with Eureka". Below this is a table with the following data:

Application	AMIs	Availability Zones	Status
PRODUCT-SERVICE	n/a (3)	(3)	UP (3) - mdp-de-illia.homeproduct-service, mdp-de-illia.homeproduct-service:8082, mdp-de-illia.homeproduct-service:8081

Microservice **ProxyService**

L'architecture microservices, en fournissant un ensemble de services indépendants et faiblement couplés, se trouve confrontée au challenge de fournir une interface unifiée pour les consommateurs, de manière à ce qu'ils ne voient pas la décomposition à faible granularité de vos services. C'est pour cela que l'utilisation d'un service proxy, responsable du routage des requêtes et de la répartition de charge, est important.



Netflix offre le service **Gateway** pour réaliser cela. Pour créer votre microservice Proxy :

- Aller à *Spring Initializr*.
- Créer le projet *proxy-service* avec les dépendances suivantes : Gateway, Config Client et Eureka Discovery.
- Ouvrir le service avec IntelliJ IDEA.
- Ajouter à la classe *ProxyServiceApplication* l'annotation `@EnableDiscoveryClient` pour que le proxy soit également enregistré dans le service de découverte.

-
- Ajouter les propriétés `spring.application.name` et `spring.cloud.config.uri` dans le fichier *application.properties* du service proxy comme suit :

```
spring.application.name=proxy-service
spring.cloud.config.uri=http://localhost:8888
```

- Renommer le fichier *application.properties* de votre projet « proxy-service » en *bootstrap.properties*
- Créer le fichier *proxy-service.yml* dans le répertoire *myConfig* du service de configuration, dans lequel vous allez fixer les propriétés suivantes :

```
server:
  port: 9999

eureka:
  instance:
    hostname: localhost

spring:
  application:
    name: proxy-service
  cloud:
    gateway:
      routes:
        - id: product-service
          uri: lb://product-service/
          predicates:
            - Path=/product-service/**
```

- lançant le service Proxy, vous remarquerez qu'il est rajouté dans Eureka.

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
PRODUCT-SERVICE	n/a (3)	(3)	UP (3) - mbp-de-lilia.home:product-service , mbp-de-lilia.home:product-service:8082 , mbp-de-lilia.home:product-service:8081
PROXY-SERVICE	n/a (1)	(1)	UP (1) - mbp-de-lilia.home:proxy-service:9999

- Dans la classe *ProductRestService* du projet *ProductService* ajouter l'annotation `@RequestMapping("/product-service")` juste après l'annotation `@RestController`
- Redémarrer les trois instances du microservice « product-service »
- Si vous exécutez la requête <http://localhost:9999/product-service/messages> plusieurs fois, vous remarquerez que l'affichage *c'est moi qui ai répondu !* s'affichera sur les consoles des trois instances respectivement, à tour de rôle (équilibrage de charge).