Implementando AOP con Spring Boot y AspectJ

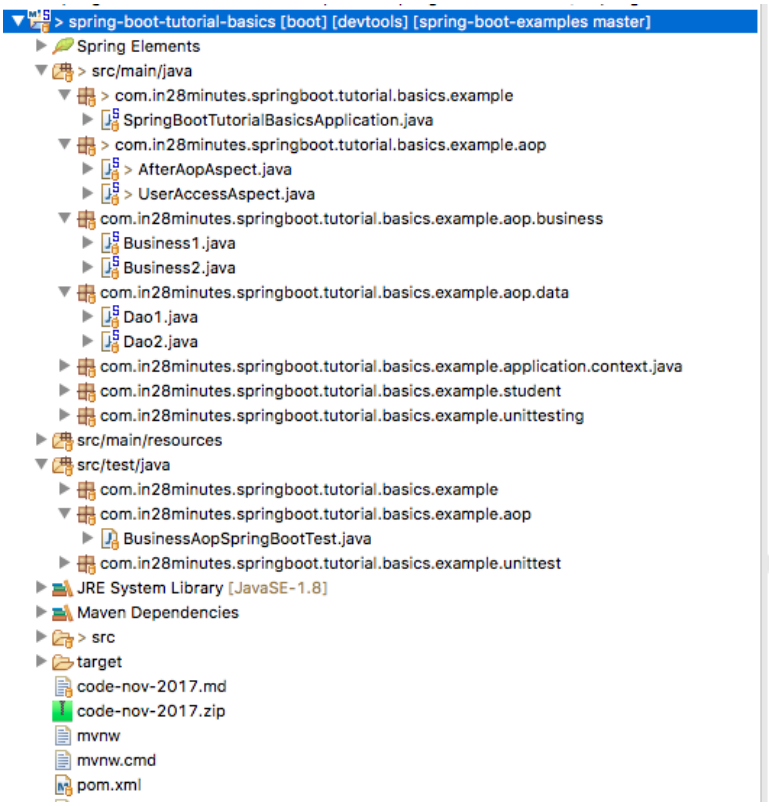
por Ranga Karanam MVB · 12 y 18 de febrero · Zona de Java

Esta guía lo ayudará a implementar el AOP con Spring Boot Starter AOP. Implementaremos cuatro consejos diferentes usando AspectJ y también crearemos una anotación personalizada para rastrear el tiempo de ejecución de un método.

Aprenderás

- ¿Cuáles son las preocupaciones transversales?
- ¿Cómo se implementan las preocupaciones transversales en una aplicación?
    - Si desea registrar cada solicitud en una aplicación web, ¿cuáles son las opciones que se le ocurren?
    - Si desea realizar un seguimiento del rendimiento de cada solicitud, ¿qué opciones puede pensar?
- ¿Qué son los Aspectos y los Pointcuts en AOP?

La siguiente captura de pantalla muestra la estructura del proyecto que crearemos.



Algunos detalles:

- SpringBootTutorialBasicsApplication.java: la clase de aplicación Spring Boot generada con Spring Initializer. Esta clase actúa como el punto de lanzamiento para la aplicación.
- `pom.xml` : Contiene todas las dependencias necesarias para construir este proyecto. Utilizaremos Spring Boot Starter AOP.
- `Business1.java` , `Business2.java` , `Dao1.java` , `Dao2.java` : Clases de negocio dependen de clases DAO. Escribimos aspectos para

Proyecto completo de Maven con ejemplos de código

Nuestro repositorio de GitHub tiene todos los ejemplos de código

Introducción al AOP

Las aplicaciones generalmente se desarrollan con múltiples capas. Una aplicación Java típica tiene:

- Capa Web: Exponer servicios al mundo exterior utilizando REST o una aplicación web
- Business Layer: lógica de negocios
- Capa de datos: lógica de persistencia

Si bien las responsabilidades de cada una de estas capas son diferentes, existen algunos aspectos comunes que se aplican a todas las capas

- Explotación florestal
- Seguridad

Estos aspectos comunes se llaman preocupaciones transversales.

Esto asegura que las preocupaciones transversales se definan en un componente de código cohesivo y se pueden aplicar según sea necesario.

Configurando Spring Boot AOP Project

La creación de un Spring AOP Project con Spring Initializr es una caminata.

Spring Initializr (http://start.spring.io/) es una gran herramienta para iniciar sus proyectos Spring Boot.

Notas:

- Inicie Spring Initializr y elija lo siguiente
  - Elige `com.in28minutes.springboot.tutorial.basics.example` como Grupo
  - Elija `spring-boot-tutorial-basics` como Artefacto
  - Elija las siguientes dependencias
    - AOP
- Haga clic en Generar proyecto.
- Importe el proyecto en Eclipse.

```
    < artifactId > spring-aop </ artifactId >
    < version > 5.0.1.RELEASE </ version >
    < scope > compile </ scope >
</ dependency >
< dependencia >
    < groupId > org.aspectj </ groupId >
    < artifactId > aspectjweaver </ artifactId
    < version > 1.8.12 </ version >
    < scope > compile </ scope >
</ dependency >
```

Configurando AOP

Agreguemos un par de clases de negocios: Business1 y Business2. Estas clases de negocios dependen de un par de clases de datos: Data1 y Data2.

```
@Servicio
 clase  pública Business1 {

    privada  Logger  logger  =  LoggerFactory .

    @Autowired
    privado  Dao1  dao1 ;

    public  String  calculateSomething () {
         Valor de  cadena =  dao1 . retrieveSom
        registrador . información ( "En negocio
```

```
        devolver  dao2 . retrieveSomething ();
    }
}
```

```
@Repositorio
 clase  pública Dao1 {

    public  String  retrieveSomething () {
        devolver  "Dao1" ;
    }

}
```

```
@Repositorio
 clase  pública Dao2 {

    public  String  retrieveSomething () {
        devolver  "Dao2" ;
    }

}
```

Notas:

- `@Autowired private Dao1 dao1` : Los DAO se autoconectan como dependencias en las clases Business.
- `public String calculateSomething(){` : Cada una de las clases de negocios tiene un método de cálculo simple.

```
7       @Autowired
8       privado  Business1  business1 ;
9
10      @Autowired
11      privado  Business2  business2 ;
12
13      @Prueba
14      public  void  invokeAOPStuff () {
            registrador . información ( business1 .
15
            registrador . información ( business2 .
16
17      }
18  }
```

Notas:

- `@RunWith(SpringRunner.class) @SpringBootTest public class BusinessAopSpringBootTest` : Estamos lanzando la aplicación Spring Boot completa en la prueba unitaria.
- `@Autowired private Business1 business1` y `@Autowiredprivate Business2 business2` : Autocablee las clases de negocios en la prueba desde el Contexto de Primavera lanzado.
- `@Test public void invokeAOPStuff() {` : Invoque los métodos en los servicios comerciales.

En este punto, no tenemos implementada la lógica AOP. Entonces, el resultado serían los mensajes simples de las clases Dao y Business.

Una implementación se muestra a continuación:

```
@Aspecto
@Configuración
 clase  pública UserAccessAspect {

    privada  Logger  logger  =  LoggerFactory .

    // Qué tipo de método llamaría para interce
    // ejecución (* PAQUETE. *. * (..))
    // Weaving & Weaver
    @Before ( "execution (* com.in28minutes.spr
     vacío  público anterior ( JoinPoint  joinP
        //Consejo
        registrador . información ( "Verificar
        registrador . información ( "Ejecución
    }
}
```

Notas

- `@Aspect` : indica que este es un aspecto
- `@Configuration` : indica que este archivo contiene una configuración de Spring Bean para un Aspecto.
- `@Before` : Quisiéramos ejecutar el Aspecto antes de la ejecución del método
- ("execution(*

```
Ejecución permitida para la ejecución (String c
cistbeaBusinessAopSpringBootTest: In Business -
cistbeaBusinessAopSpringBootTest: Dao1

Verificar el acceso del usuario
Ejecución permitida para la ejecución (String c
cistbeaBusinessAopSpringBootTest: Dao2
```

Comprender la terminología de AOP: Pointcut, Advice, Aspect, Join Point

Dediquemos un tiempo a entender la terminología de AOP.

- Pointcut: la expresión utilizada para definir cuándo debe interceptarse una llamada a un método. En el ejemplo anterior, es el punto de corte. `execution(* com.in28minutes.springboot.tutorial.basics.example.aop.data.*.*(..))`
- Consejo: ¿Qué quieres hacer? Un consejo es la lógica que desea invocar cuando intercepta un método. En el ejemplo anterior, es el código dentro del `before(JoinPoint joinPoint)` método.
- Aspecto: una combinación de definir cuándo quiere interceptar una llamada a un método (Pointcut) y qué hacer (Asesoramiento) se denomina Aspecto.

@After : ejecutado en dos situaciones: cuando un método se ejecuta correctamente o arroja una excepción.

- `@AfterReturning` : ejecutado solo cuando un método se ejecuta con éxito.
- `@AfterThrowing` : ejecutado solo cuando un método arroja una excepción.

Vamos a crear un aspecto simple con un par de estas variaciones.

```
@Aspecto
@Configuración
 clase  pública AfterAopAspect {

    privada  Logger  logger  =  LoggerFactory .

    @AfterReturning ( value  =  "execution (* c

        return  =  "resultado" )
     vacío  público afterReturning ( JoinPoint

        registrador . información ( "{} devuelt

    }

    @After ( valor  =  "ejecución (* com.in28mi

     vacío  público después ( JoinPoint  joinPc

        registrador . información ( "después de

    }
}
```

```
cistbeaBusinessAopSpringBootTest: Dao1
Verificar el acceso del usuario
Ejecución permitida para la ejecución (String c
después de la ejecución de la ejecución (String
ejecución (String com.in28minutes.springboot.tu
cistbeaBusinessAopSpringBootTest: Dao2
```

Como puede ver, justo antes de devolver los valores a los métodos comerciales de llamada, los `after` consejos se ejecutan.

Otras características de AOP: @Around y anotaciones

Una de las otras características que puede implementar utilizando AOP son las anotaciones personalizadas para interceptar llamadas a métodos.

El siguiente ejemplo muestra una `TrackTime` anotación simple :

```
@Target ( ElementType . MÉTODO )
@Retention ( RetentionPolicy . RUNTIME )
public  @interface  TrackTime {
```

Podemos agregar un Aspecto que define qué se debe hacer cuando `TrackTime` se usa la anotación.

```
        long  startTime  =  Sistema . currentTi
9
10
11      joinPoint . proceder ();
12
        long  timeTaken  =  Sistema . currentTi
13
        registrador . información ( "Tiempo tom
14
15    }
16  }
```

Notas:

- `@Around` usa un consejo alrededor Se intercepta la llamada al método y se utiliza `joinPoint.proceed()` para ejecutar el método.
- `@annotation(com.in28minutes.springboot.tutorial.basics.example.aop.TrackTime)` es el punto de corte para definir la intercepción basada en una anotación: @anotación seguida del nombre de tipo completo de la anotación.

Una vez que definimos la anotación y el consejo, podemos usar la anotación en los métodos que deseamos seguir, como se muestra a continuación:

```
1   @Servicio
2    clase  pública Business1 {
3
4      @TrackTime
       public  String  calculateSomething () {
5
```

```
    @Pointcut ( "ejecución (* com.in28minutes.s
    public  void  businessLayerExecution () {}

}
```

La definición común anterior se puede usar al definir puntos de corte en otros aspectos.

```
@Around ( "com.in28minutes.spring.aop.springaop
```

Ejemplos de código completo

pom.xml

```
<? xml  version = "1.0" encoding = "UTF-8"?>
< project  xmlns = "http://maven.apache.org/POM
    < modelVersion > 4.0.0 </ modelVersion >
    < ID de grupo > com.in28minutes.springboot.
    < artifactId > spring-boot-tutorial-basics
    < version > 0.0.1-SNAPSHOT </ version >
    < packaging > jar </ packaging >
    < nombre > spring-boot-tutorial-basics </ r
```

```
18
        < project.reporting.outputEncoding > UT
19
        < java.version > 1.8 </ java.version >
20
    </ propiedades >
21
    < dependencias >
22
        < dependencia >
23
            < groupId > org.springframework.boo
24
            < artifactId > spring-boot-starter-
25
        </ dependency >
26
        < dependencia >
27
            < groupId > org.springframework.boo
28
            < artifactId > spring-boot-starter-
29
        </ dependency >
30
        < dependencia >
31
            < groupId > org.springframework.boo
32
            < artifactId > spring-boot-devtools
33
            < scope > tiempo de ejecución </ sc
34
        </ dependency >
35
        < dependencia >
36
            < groupId > org.springframework.boo
37
            < artifactId > spring-boot-starter-
38
            < scope > prueba </ scope >
39
        </ dependency >
40
    </ dependencias >
41
    < construir >
42
```

```
53                                                                    
            < url > https://repo.spring.io/snap
54                                                                    
            < instantáneas >
55              < enabled > true </ enabled >
56                                                                    
            </ snapshots >
57          </ repository >
58          < repositorio >
59              < id > primavera-hitos </ id >
60              < nombre > Spring Milestones </ nam
61                                                                    
            < url > https://repo.spring.io/mile
62                                                                    
            < instantáneas >
63              < habilitado > falso </ enablec
64                                                                    
            </ snapshots >
65          </ repository >
66      </ repositories >
67      < pluginRepositories >
68          < pluginRepository >
69              < id > instantáneas de primavera </
70                                                                    
            < nombre > Spring Snapshots </ name
71                                                                    
            < url > https://repo.spring.io/snap
72                                                                    
            < instantáneas >
73              < enabled > true </ enabled >
74                                                                    
            </ snapshots >
75          </ pluginRepository >
76          < pluginRepository >
77              < id > primavera-hitos </ id >
```

/src/main/java/com/in28minutes/springboot/tutorial/basics/example/aop/AfterAopAspect.java

```
1  paquete  com . in28minutes . Springboot . tutor
2  import  org . aspectj . lang . JoinPoint ;
3  import  org . aspectj . lang . anotación . Desp
4  import  org . aspectj . lang . anotación . Afte
5  import  org . aspectj . lang . anotación . aspe
6  import  org . slf4j . Logger ;
7  import  org . slf4j . LoggerFactory ;
8  import  org . springframework . contexto . anot
9
10 // AOP
11 //Configuración
12 @Aspecto
13 @Configuración
14  clase  pública AfterAopAspect {
15
16     privada  Logger  logger  =  LoggerFactory .
17
18     @AfterReturning ( value  =  "execution (* c
19         return  =  "resultado" )
20      vacío  público afterReturning ( JoinPoint
21         registrador . información ( "{} devuelt
```

/springboot/tutorial/basics/example/aop/business/Business1.java

```
paquete  com . in28minutes . Springboot . tutor
import  org . slf4j . Logger ;
import  org . slf4j . LoggerFactory ;
import  org . springframework . habas . fábrica
import  org . springframework . estereotipo . S

importación  com . in28minutes . Springboot . t

@Servicio
 clase  pública Business1 {

    privada  Logger  logger  =  LoggerFactory .

    @Autowired
    privado  Dao1  dao1 ;

    public  String  calculateSomething () {
        //Lógica de negocios
         Valor de  cadena =  dao1 . retrieveSom
        registrador . información ( "En negocio
         valor de retorno ;
    }
```

```
import  org . springframework . estereotipo . S

importación  com . in28minutes . Springboot . t

@Servicio
 clase  pública Business2 {

    @Autowired
     Dao2  dao2 privado ;

    public  String  calculateSomething () {
        //Lógica de negocios
        devolver  dao2 . retrieveSomething ();
    }
}
```

/src/main/java/com/in28minutes/springboot/tutorial/basics/example/aop/data/Dao1.java

```
paquete  com . in28minutes . Springboot . tutor

import  org . springframework . estereotipo . r

@Repositorio
 clase  pública Dao1 {
```

```
1
2
import  org . springframework . estereotipo . r
3
4
5   @Repositorio
6    clase  pública Dao2 {
7
8       public  String  retrieveSomething () {
9           devolver  "Dao2" ;
10      }
11
12  }
```

/src/main/java/com/in28minutes/springboot/tutorial/basics/example/aop/UserAccessAspect.java

```
    paquete  com . in28minutes . Springboot . tutor
1
2   import  org . aspectj . lang . JoinPoint ;
    import  org . aspectj . lang . anotación . aspe
3
    import  org . aspectj . lang . anotación . Ante
4
5   import  org . slf4j . Logger ;
6   import  org . slf4j . LoggerFactory ;
    import  org . springframework . contexto . anot
7
8
```

```
20
     vacío  público anterior ( JoinPoint  joinP
21
22         //Consejo
           registrador . información ( "Verificar
23
           registrador . información ( "Ejecución
24
25     }
26 }
```

/src/main/java/com/in28minutes/springboot/tutorial/basics/example/SpringBootTutorialBasicsApplication.java

```
   paquete  com . in28minutes . Springboot . tutor
1
2
   import  org . springframework . arranque . Spri
3
   import  org . springframework . arranque . autc
4
   import  org . springframework . contexto . Appl
5
6
7  @SpringBootApplication
    clase  pública SpringBootTutorialBasicsApplica
8
9
     public  static  void  main ( String [] args
10
         ApplicationContext  applicationContext
11
```

ple/aop/BusinessAopSpringBootTest.java

```
paquete  com . in28minutes . Springboot . tutor
1
2
3
import  org . junit . Prueba ;
4
import  org . junit . corredor . RunWith ;
5
import  org . slf4j . Logger ;
6
import  org . slf4j . LoggerFactory ;
7
import  org . springframework . habas . fábrica
8
import  org . springframework . arranque . prue
9
import  org . springframework . prueba . contex
10
11
importación  com . in28minutes . Springboot . t
12
importación  com . in28minutes . Springboot . t
13
14
@RunWith ( clase SpringRunner . )
15
@SpringBootTest
16
 clase  pública BusinessAopSpringBootTest {
17
18
    privada  Logger  logger  =  LoggerFactory .
19
20
    @Autowired
21
    privado  Business1  business1 ;
22
23
    @Autowired
```

/src/test/java/com/in28minutes/springboot/tutorial/basics/example/SpringBootTutorialBasicsApplicationTests.java

```
paquete  com . in28minutes . Springboot . tutor

import  org . junit . Prueba ;
import  org . junit . corredor . RunWith ;
import  org . springframework . arranque . prue
import  org . springframework . prueba . contex

@RunWith ( clase SpringRunner . )
@SpringBootTest
 clase  pública SpringBootTutorialBasicsApplica

    @Prueba
    public  void  contextLoads () {}

}
```

Construir vs Comprar una solución de calidad de datos: ¿Cuál es el mejor para usted? Mantener los datos de alta calidad es esencial para la eficiencia operativa, el análisis significativo y las buenas relaciones con los clientes a largo plazo. Pero, cuando se trata de múltiples fuentes de datos, la calidad de los datos se vuelve compleja, por lo que debe saber cuándo debe crear un esfuerzo de

DZone son suyas.

Recursos para socios de **Java**

Pruebas continuas para DevOps: evolución más allá de la automatización
Parasoft

Play Framework: La ruta del arquitecto de JVM hacia aplicaciones web súper rápidas
Lightbend

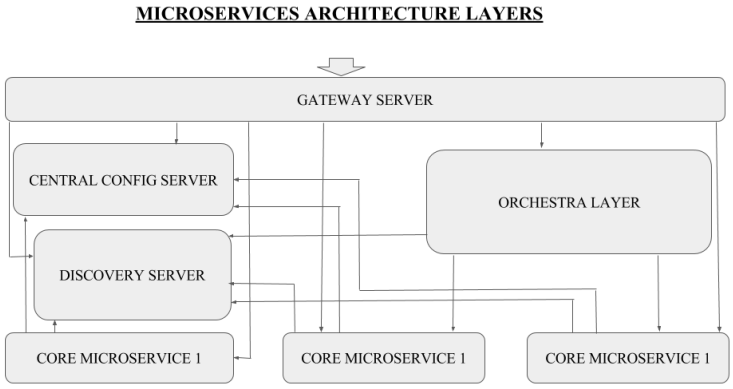
Reactive Microsystems - The Evolution of Microservices at Scale
Lightbend

Microservices for Java Developers: A Hands-On Introduction to Frameworks & Containers
Red Hat Developer Program

Basics for Setting Up a Microservices Architecture in a Project for Spring Boot and Gradle

by Akash Bhingole · Feb 12, 18 · Microservices Zone

simple web application built using microservices architecture.



1. Spring Boot

Spring Boot makes it easy to create stand-alone applications with tomcat installed, which you can run by starting the jar file. A Spring Boot app does not require any kind of XML configurations; everything is done using just annotations. It is very simple to create a web app using Spring Boot. Below, you can see an example of a Spring Boot controller, which makes it so simple to create a web app with a REST service:

```
@Controller
@EnableAutoConfiguration
public class SampleController {

    @RequestMapping("/")
    @ResponseBody
    String home() {
        return "Hello World!";
```

require any XML file, as it has its own DSL based in Groovy. Gradle is much simple and cleaner than Maven or Ant. We have the build.gradle file in it, which includes all the dependencies required for a web app. It also includes the jar name to be generated along with Java, Hibernate, and Database versions. Below is a code snippet from the build.gradle file:

```
apply plugin: 'java'
apply plugin: 'checkstyle'
apply plugin: 'findbugs'
apply plugin: 'pmd'

version = '1.0'

repositories {
    mavenCentral()
}

dependencies {
    testCompile group: 'junit', name: 'junit',
    testCompile group: 'org.hamcrest', name: 'h
}
```

3. Discovery Server

Discovery Server is mainly used to have all the microservices' clients connected at a central place so that they can easily communicate. Eureka Discovery receives heartbeat messages from each instance belonging to a service. If the heartbeat fails over a configurable timetable, the instance is normally

```
}
```

For all the discovery clients, you need to add the configuration below into the application.yml of each client module to locate its discovery server:

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka
```

4. Central Config Server

The main functionality of having central-config-server is for storing all kind of configuration properties at a central place so that we do not need to go to each core module explicitly to change properties. It is actually connected to the discovery server, which makes it easy for each core's microservices to get its properties files. Whenever a change is made to a property file, we can just restart this server and the core module whose property file is changed; you will not even require any kind of build for the core module to get the updated properties. You place properties files at any specific location (Git, etc) and just specify the path of properties in the application.yml file. The code snippet below will give you an overview of the Central Config server:

```
@SpringBootApplication
@EnableConfigServer
@EnableDiscoveryClient
public class CentralConfigServerApplication {
```

```
config:
    server:
        native:
            searchLocations: file:./prc
```

5. Gateway Server

Gateway/Zuul is an edge service that provides dynamic routing, monitoring, resiliency, security, and more. The main purpose of this is to provide security and routing for the core microservices. We can have different types of filters in the Gateway server so that we can manage security for any type of API call to the core microservices. It acts as a proxy between core microservices and the outside applications.

```
package com.example.EmployeeZuulService;
import org.springframework.boot.SpringApplicati
import org.springframework.boot.autoconfigure.S
import org.springframework.cloud.client.discove
import org.springframework.cloud.netflix.zuul.E
@EnableZuulProxy
@EnableDiscoveryClient
@SpringBootApplication
public class EmployeeZuulServiceApplication {
   public static void main(String[] args) {
      SpringApplication.run(EmployeeZuulService
   }
```

The use of this layer in microservice architecture is to combine different kinds of responses from multiple core services and do more processing on the data, then publish them in the response. The main need for this layer is less as compared to all other layers. It is just a Spring Boot app which is communicating to discovery, gateway, and microservices but does not have any kind of interactions with the database part.

```
@RestController
@RequestMapping("orchestra")
public class OrchestraController {

  @Autowired
  @LoadBalanced
  protected RestTemplate restTemplate;



  protected Logger LOGGER = LoggerFactory.getLo

  @ApiOperation(value = "Retrieve combined list
@RequestMapping(value="/combinedlists", methc
public ResponseEntity<List<Object>> getCombir
    LOGGER.info("Inside getCombinedList method"
    List<Object> combinedList = new ArrayList<>
    try {
      String url1 = "http://"+"<core_microservi
      String url2 = "http://"+"<core_microservi
```

```
28
      return new ResponseEntity<List<Object>>(c
29
    }
30
    LOGGER.info("Exit from getCombinedList meth
31
    return new ResponseEntity<List<Object>>(com
32
  }
33
}
34
```

7. Core Microservices Layer

This is the lowest layer in the microservices architecture, which actually performs a lot of operations on the database and process the data as per the need. The actual REST services are written in the core layer. This part does each operation of different transactions.

It has connections with the discovery through the `@EnableDiscoveryClient` annotations. As we already added environment level configurations in the central config server, we can still have application-level configuration settings/messages in the application.properties in the core module itself.

SampleServiceApplication.java

```
1  @SpringBootApplication
2  @EnableDiscoveryClient
3  public class SampleServiceApplication {
```

```
5
6       @Autowired
        private SetProcTimeBean setProcTimeBean;
7
8
9       @Autowired
        private LoadBalancerClient loadBalancer;
10  private RestTemplate restTemplate = new RestTem
11
12
13      @RequestMapping("/recommendation")
        public List<Recommendation> getRecommendati
14          List<Recommendation> list = new ArrayLi
15          list.add(new Recommendation(productId,
16          list.add(new Recommendation(productId,
17          list.add(new Recommendation(productId,
18          return list;
19      }
20  }
21
```

application.yml

```
1   server:
2     port: 7878
3
4   eureka:
5     instance:
6       leaseRenewalIntervalInSeconds: 10
7       metadataMap:
          instanceId: ${vcap.application.instance_i
```

```
9           serviceId: central-config-server
```

References

https://spring.io/guides/gs/centralized-configuration/ x

https://spring.io/guides/gs/service-registration-and-discovery/ x

Topics: MICROSERVICES, SOFTWARE ARCHITECTURE, CI/CD, TUTORIAL, SPRING BOOT, GRADLE