

Spring Cloud Config - PADSA

Spring Cloud Config es una funcionalidad que nos permite crear un servicio que centralizara las configuraciones de nuestra infraestructura de microservicios y alojar todos estos archivos o configuraciones para hacer manejo y modificación de ellas desde un solo lugar, permitiendo un manejo de nuestra infraestructura mas simple y rápido una vez se haya montado.

Documentación Oficial de Spring Cloud Config:

Spring Cloud Config

This quick start walks through using both the server and the client of Spring Cloud Config Server. First, start the server, as follows: `$ cd spring-cloud-config-server $./mvnw spring-boot:run` The server is a Spring Boot application, so you can run it from your IDE if you prefer to do so (the main class is `ConfigServerApplication`).

https://docs.spring.io/spring-cloud-config/docs/current/reference/html/#_quick_start

Entorno de trabajo y herramientas utilizadas

- Windows 10 home
- Java openjdk 11.0.13
- IDE Spring Suit Tool 4
- Constructor de código Gradle
- MySQL Workbench 8
- Postman API tester

Configuración de Spring Cloud Config con un Backend GitHub y Uso de Refresh Scope

Este ejemplo se conecta a un repositorio en GitHub para obtener los datos de *properties* y poder suministrarlos a los correspondientes microservicios conectados al “*Config-Service*”, además se hace uso de “*actuator*” para poder hacer un “*refresh scope*” en los “*beans*” de los “*properties*” que lee y actualizarlos en el microservicio sin necesidad de detenerlo.

En este ejemplo únicamente se levantarán dos servicios con los siguientes nombres:

my-config-service: Este servicio se levanta como un servidor(para ser un medio para tener centralizadas las configuraciones de los microservicios en nuestra arquitectura de trabajo), se conecta al repositorio y se encarga de filtrar y traer los registros de *properties* correspondientes a cada servicio que este conectado a este.

refresh-scope: Se conecta al servicio de configuraciones para leer los valores de sus configuraciones correspondientes.

Estructura del servicio *my-config-service*

```
my-config-service [boot]
├── src/main/java
│   └── com.scloud
│       └── MyConfigServiceApplication.java
├── src/main/resources
│   ├── META-INF
│   │   └── bootstrap.yml
│   └── src/test/java
├── JRE System Library [JavaSE-11]
├── Project and External Dependencies
├── bin
├── gradle
├── src
│   ├── build.gradle
│   ├── gradlew
│   ├── gradlew.bat
│   ├── HELP.md
│   └── settings.gradle
```

1. Agregar las Dependencias en nuestro proyecto al Archivo *build.gradle*

- `spring-boot-starter-security` para la seguridad del acceso a este microservicio.
- `spring-cloud-config-server` para levantar el servidor de configuraciones centralizadas.
- `spring-cloud-starter-bootstrap` para hacer uso del archivo bootstrap.

```
plugins {
    id 'org.springframework.boot' version '2.6.3'
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'
    id 'java'
}

group = 'com.scloud'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

repositories {
    mavenCentral()
}

ext {
    set('springCloudVersion', "2021.0.1")
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-security'
    implementation 'org.springframework.cloud:spring-cloud-config-server'
    implementation group: 'org.springframework.cloud', name: 'spring-cloud-starter-bootstrap', version: '3.1.1'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    testImplementation 'org.springframework.security:spring-security-test'
}

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-dependencies:${springCloudVersion}"
    }
}

tasks.named('test') {
    useJUnitPlatform()
}
```

2. En la clase principal se agrega la anotación `@EnableConfigServer` para habilitar el servidor.

```
package com.scloud;

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

@EnableConfigServer
@SpringBootApplication
public class MyConfigServiceApplication {

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

3. Configuraciones: Se cambia el archivo default `application.properties` por un archivo `bootstrap.yml` pues este cargara antes que de contexto de la aplicación y se le declaran las siguientes configuraciones; puerto 8888 recomendado para el servicio config, la uri del repositorio a gitHub, el directorio donde se encuentran los archivos de configuración *config-props* para este ejemplo, user y password de GitHub y la rama de consulta.

Le damos un nombre de identificación al servicio en el environment.

Establecemos las credenciales de security, username y password para los servicios que se conecten al *config-service*.

```
server:
  port: 8888
```

```

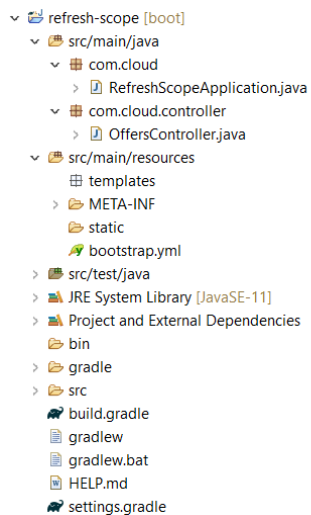
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/edgarSTM/spring-cloud.git
          searchpaths: config-props
          username: ${GIT_USER}
          password: ${GIT_PASSWORD}
          default-label: "master"

application:
  name: my-config-service

security:
  user:
    name: root
    password: s3cr3t

```

Creación del segundo microservicio



1. Se agregan las dependencias necesarias al archivo *build.gradle*

- `spring-boot-starter-actuator` para habilitar los endpoints de monitoreo del servicio.
- `spring-boot-starter-web` para poder exponer contenido en el navegador.
- `spring-cloud-starter-config` para conectarse al como cliente al *config-service*.
- `spring-cloud-starter-bootstrap` para hacer uso del archivo bootstrap.

```

plugins {
    id 'org.springframework.boot' version '2.5.10'
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'
    id 'java'
}

group = 'com.cloud'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

repositories {
    mavenCentral()
}

ext {
    set('springCloudVersion', "2020.0.5")
}

dependencies {

```

```

implementation 'org.springframework.boot:spring-boot-starter-actuator'
implementation 'org.springframework.boot:spring-boot-starter-web'
implementation group: 'de.codecentric', name: 'spring-boot-admin-starter-client', version: '2.6.2'
implementation group: 'org.springframework.cloud', name: 'spring-cloud-starter-bootstrap', version: '3.1.1'
implementation group: 'org.springframework.cloud', name: 'spring-cloud-starter-config', version: '3.1.1'
testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-dependencies:${springCloudVersion}"
    }
}

tasks.named('test') {
    useJUnitPlatform()
}

```

2. En la clase del controlador agregamos la anotaciones `@RestController` y `@RefreshScope` y con la anotación `@Value` spring se encargara de traer los valores de las propiedades y asignarlos a una variable.

```

package com.cloud.controller;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RefreshScope
public class OffersController {

    @Value("${offers.discount:35%}")
    private String discount;

    @Value("${mi.string: Hola Mundo}")
    private String hello;

    @GetMapping("/")
    public String viewDiscounts() {
        return "Descuento: " + discount + " " + hello;
    }
}

```

3. Se cambia el archivo por defecto `application.properties` por un archivo `bootstrap.yml` con la siguiente configuración para conectarse al servicio config y obtener los datos.

```

spring:
  application:
    name: refresh-scope

#Conección al Config Service
cloud:
  config:
    name: refresh-scope
    uri: http://localhost:8888
#Credenciales de Spring Security
    username: root
    password: s3cr3t

```

4. Creamos el archivo `refresh-scope.properties` con las configuraciones que pedirá este servicio al `config-service` y que se alojaran en el repositorio de GitHub.

```

server.port=3000
offers.discount=50%
mi.string=descuento para el dev
management.endpoints.web.exposure.include=*

```

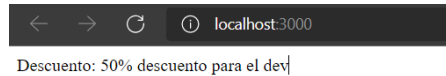
Utilizando postman y los endpoints que se habilitan gracias a *actuator* podremos traer la data de propiedades de cada servicio registrado y que tenga su correspondiente archivo de propiedades en el repositorio, se puede comprobar con un endpoint como el siguiente haciendo referencia al servicio que se necesite consultar.

<http://root:s3cr3t@localhost:8888/refresh-scope/default>

- Al correr este servicio podemos ver en los logs de consola como se conecta al servicio config y aplica el perfil y label definidos. Ahora podemos levantar nuestro servicio y si la conexión fue correcta el *config-service* le proveerá sus correspondientes configuraciones, por ejemplo que se levantara en el puerto 3000 como lo declaramos en el archivo en GitHub.

El servicio Rest muestra los valores en el siguiente endpoint:

<http://localhost:3000/>



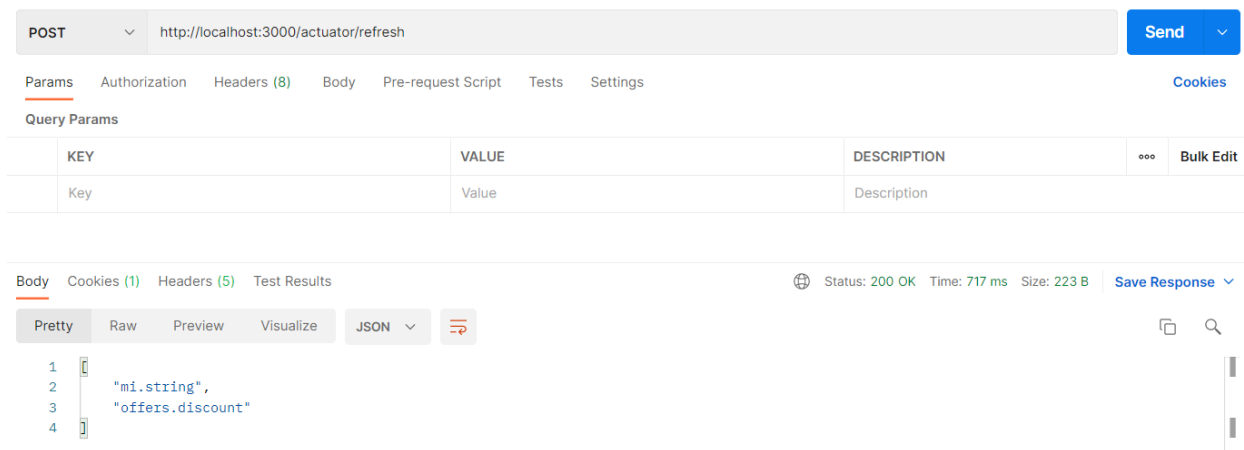
Descuento: 50% descuento para el dev

- Ahora podemos ir al archivo *refresh-scope.properties* del repositorio y modificar los valores de `offers.discount` y/o `mi.string`.

El *config-service* obtiene este archivo identificándolo por tener el mismo nombre que se le asignó al servicio como identificador en el environment.

```
server.port=3000
offers.discount=25%
mi.string=descuento para el dev con refresh scope
management.endpoints.web.exposure.include=*
```

- Luego se hace un post al endpoint de actuator <http://localhost:3000/actuator/refresh> con postman, además se pueden verificar los keys que cambiaron.



POST <http://localhost:3000/actuator/refresh> Send

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

Query Params

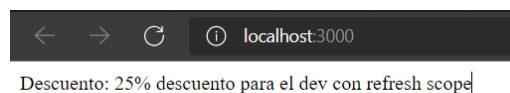
KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

Body Cookies (1) Headers (5) Test Results Status: 200 OK Time: 717 ms Size: 223 B Save Response

Pretty Raw Preview Visualize JSON

```
1 [
2   "mi.string",
3   "offers.discount"
4 ]
```

El recargar o hacer de nuevo la petición get los valores se habrán actualizado, esto sin haber detenido al servicio gracias al refresh scope y actuator.



Descuento: 25% descuento para el dev con refresh scope

Configuración de Spring Cloud Config con un Backend JDBC y Uso de Refresh Scope

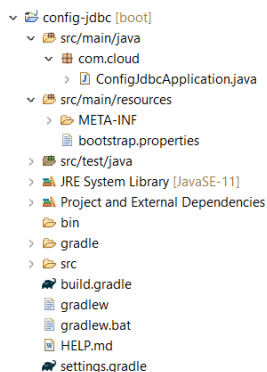
Este ejemplo se conecta a una base de datos de MySQL para obtener los datos de *properties* y poder suministrarlos a los correspondientes microservicios conectados al “Config-Service”, además se hace uso de “actuator” para poder hacer un “refresh scope” en los “beans” de los “properties” que lee y actualizarlos en el microservicio sin necesidad de detenerlo.

En este ejemplo únicamente se levantarán dos servicios con los siguientes nombres:

config-jdbc: Este servicio se levanta como un servidor(para ser un medio para tener centralizadas las configuraciones de los microservicios en nuestra arquitectura de trabajo), se conecta a la base de datos y se encarga de filtrar y traer los registros de *properties* correspondientes a cada servicio que este conectado a este.

refresh-scope: Se conecta al servicio de configuraciones para leer los valores de sus configuraciones correspondientes.

Estructura del servicio de configuraciones *config-jdbc*



1. Agregar las Dependencias en nuestro proyecto al Archivo *build.gradle*

- `Spring Security` para la seguridad del acceso a este microservicio.
- `Spring Cloud Config Server` para levantar el servidor de configuraciones centralizadas.
- `spring-boot-starter-jdbc` para habilitar el uso del profile jdbc.
- `mysql:mysql-connector-java` para obtener el controlador de MySQL y hacer la conexión a base de datos.
- `spring-cloud-starter-bootstrap` para hacer uso del archivo bootstrap.

```
plugins {
    id 'org.springframework.boot' version '2.5.10'
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'
    id 'java'
}

group = 'com.cloud'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

repositories {
    mavenCentral()
}

ext {
    set('springCloudVersion', "2020.0.5")
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-security'
    implementation 'org.springframework.cloud:spring-cloud-config-server'
    implementation 'org.springframework.cloud:spring-cloud-starter'
```

```

implementation group: 'org.springframework.boot', name: 'spring-boot-starter-jdbc', version: '2.6.3'
implementation group: 'org.springframework.cloud', name: 'spring-cloud-starter-bootstrap', version: '3.1.1'
runtimeOnly 'mysql:mysql-connector-java'
testImplementation 'org.springframework.boot:spring-boot-starter-test'
testImplementation 'org.springframework.security:spring-security-test'
}

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-dependencies:${springCloudVersion}"
    }
}

tasks.named('test') {
    useJUnitPlatform()
}

```

2. En nuestra base de datos debemos crear una tabla con nombre `PROPERTIES` y con las columnas `APPLICATION`, `PROFILE`, `LABEL`, `KEY` y `VALUE`. Para este ejemplo se creo el siguiente esquema añadiendo las columnas `id`, `CREATED_DATE`, `UPDATED_AT` para la identificación de los registros.

```

create table PROPERTIES (
id serial primary key,
CREATED_DATE TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
UPDATED_DATE TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
APPLICATION varchar(200),
PROFILE varchar(200),
LABEL varchar(200),
PROP_KEY varchar(200),
VALUE text,
KEY `PROPERTIES_APPLICATION_IDX` (`APPLICATION`),
KEY `PROPERTIES_PROFILE_IDX` (`PROFILE`),
KEY `PROPERTIES_LABEL_IDX` (`LABEL`)
);

```

3. Agregamos datos a nuestra tabla para la configuración de nuestro microservicios que conectaran al `config-service`.

Se pueden agregar las los valores que sean necesarios, para este ejemplo se agregan configuraciones para el puerto de salida, la habilitación de los endpoints de actuador y los valores que se leerán en el bean.

```

INSERT INTO `mycrud`.`properties` (`APPLICATION`, `PROFILE`, `LABEL`, `PROP_KEY`, `VALUE`) VALUES ('refresh-scope', 'dev', 'latest', 'serve
INSERT INTO `mycrud`.`properties` (`APPLICATION`, `PROFILE`, `LABEL`, `PROP_KEY`, `VALUE`) VALUES ('refresh-scope', 'dev', 'latest', 'manag
INSERT INTO `mycrud`.`properties` (`APPLICATION`, `PROFILE`, `LABEL`, `PROP_KEY`, `VALUE`) VALUES ('refresh-scope', 'dev', 'latest', 'offer
INSERT INTO `mycrud`.`properties` (`APPLICATION`, `PROFILE`, `LABEL`, `PROP_KEY`, `VALUE`) VALUES ('refresh-scope', 'dev', 'latest', 'mi.st
INSERT INTO `mycrud`.`properties` (`APPLICATION`, `PROFILE`, `LABEL`, `PROP_KEY`, `VALUE`) VALUES ('refresh-scope', 'admin', 'latest', 'ser
INSERT INTO `mycrud`.`properties` (`APPLICATION`, `PROFILE`, `LABEL`, `PROP_KEY`, `VALUE`) VALUES ('refresh-scope', 'admin', 'latest', 'man
INSERT INTO `mycrud`.`properties` (`APPLICATION`, `PROFILE`, `LABEL`, `PROP_KEY`, `VALUE`) VALUES ('refresh-scope', 'admin', 'latest', 'off
INSERT INTO `mycrud`.`properties` (`APPLICATION`, `PROFILE`, `LABEL`, `PROP_KEY`, `VALUE`) VALUES ('refresh-scope', 'admin', 'latest', 'mi.

```

4. En la clase principal se agrega la anotación `@EnableConfigServer` para habilitar el servidor.

```

package com.cloud;

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

@SpringBootApplication
@EnableConfigServer
public class ConfigJdbcApplication {

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

}

```

5. Configuraciones: Se cambia el archivo default `application.properties` por un archivo `bootstrap.properties` pues este cargara antes que de contexto de la aplicación y se le declaran las siguientes configuraciones de puerto, perfil JDBC, conexión a

MySQL con user y password y la consulta de las propiedades en la base de datos por el cloud config service

```
{spring.cloud.config.server.jdbc.sql}.
```

```
#El Puerto 8888 es el estandar para correr este servicio
server.port=8888

#Se activa el Perfil JDBC
spring.application.name=config-jdbc
spring.profiles.active=jdbc

##DB connection
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/mycrud?serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=root.01

#Consulta de los configuraciones
spring.cloud.config.server.jdbc.sql=SELECT PROP_KEY, VALUE from PROPERTIES where APPLICATION=? and PROFILE=? and LABEL=?

#Asignación de credenciales de spring security para acceder a este servicio
spring.security.user.name=root
spring.security.user.password=s3cr3t
```

6. Utilizando postman y los endpoints que se habilitan gracias a *actuator* podremos traer la data de propiedades de cada servicio registrado en nuestra base de datos.

<http://root:s3cr3t@localhost:8888/refresh-scope/dev/latest>

<http://root:s3cr3t@localhost:8888/refresh-scope/admin/latest>

GET <http://root:s3cr3t@localhost:8888/refresh-scope/dev/latest> Send

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies

Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
-----	-------	-------------	-----	-----------

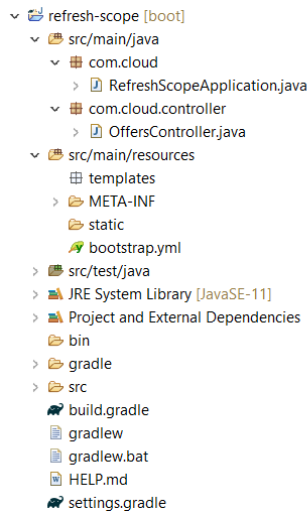
Body Cookies (1) Headers (12) Test Results Status: 200 OK Time: 1424 ms Size: 698 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "name": "refresh-scope",
3   "profiles": [
4     "dev"
5   ],
6   "label": "latest",
7   "version": null,
8   "state": null,
9   "propertySources": [
10    {
11      "name": "refresh-scope-dev",
12      "source": {
13        "server.port": "3000",
14        "management.endpoints.web.exposure.include": "*",
15        "offers.discount": "50%",
16        "mi.string": "descuento para el dev"
17      }
18    }
19  ]
20 }
```

Capture requests and cookies Bootcamp Runner Trash

Creación del segundo microservicio



1. Se agregan las dependencias necesarias al archivo build.gradle

- `spring-boot-starter-actuator` para habilitar los endpoints de monitoreo del servicio.
- `spring-boot-starter-web` para poder exponer contenido en el navegador.
- `spring-cloud-starter-config` para conectarse al como cliente al config service.
- `spring-cloud-starter-bootstrap` para hacer uso del archivo bootstrap.

```
plugins {
    id 'org.springframework.boot' version '2.5.10'
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'
    id 'java'
}

group = 'com.cloud'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

repositories {
    mavenCentral()
}

ext {
    set('springCloudVersion', "2020.0.5")
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-actuator'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation group: 'de.codecentric', name: 'spring-boot-admin-starter-client', version: '2.6.2'
    implementation group: 'org.springframework.cloud', name: 'spring-cloud-starter-bootstrap', version: '3.1.1'
    implementation group: 'org.springframework.cloud', name: 'spring-cloud-starter-config', version: '3.1.1'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-dependencies:${springCloudVersion}"
    }
}

tasks.named('test') {
    useJUnitPlatform()
}
```

2. Se cambia el archivo por defecto *application.properties* por un archivo *bootstrap.yml* con la siguiente configuración para conectarse al servicio config y obtener los datos.

```
spring:
  application:
    name: refresh-scope

#Conección al Config Service
cloud:
  config:
    name: refresh-scope
    profile: dev
    label: latest
    uri: http://localhost:8888
#Credenciales de Spring Security
username: root
password: s3cr3t
```

3. En la clase del controlador agregamos la anotaciones `@RestController` y `@RefreshScope` y con la anotación `@Value` spring se encargara de traer los valores de las propiedades y asignarlos a una variable.

```
package com.cloud.controller;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RefreshScope
public class OffersController {

    @Value("${offers.discount:35%}")
    private String discount;

    @Value("${mi.string: Hola Mundo}")
    private String hello;

    @GetMapping("/")
    public String viewDiscounts() {
        return "Descuento: " + discount + " " + hello;
    }
}
```

4. Al correr este servicio podemos ver en los logs de consola como se conecta al servicio config y aplica el perfil y label definidos.

```
refresh-scope - RefreshScopeApplication [Spring Boot App]

:: Spring Boot :: (v2.5.10)

2022-03-08 23:47:57.700 INFO 19572 --- [main] c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at : http://localhost:8888
2022-03-08 23:47:57.965 INFO 19572 --- [main] c.c.c.ConfigServicePropertySourceLocator : Located environment: name=refresh-scope, profiles=[dev], label=latest, version=null, state=null
2022-03-08 23:47:57.966 INFO 19572 --- [main] b.c.PropertySourceBootstrapConfiguration : Located property source: [BootstrapPropertySource {name='bootstrapProperties-configClient'}, BootstrapPropertySource {name='bootstrapProperties-configClient'}, BootstrapPropertySource {name='bootstrapProperties-configClient'}]
2022-03-08 23:47:57.973 INFO 19572 --- [main] com.cloud.RefreshScopeApplication : No active profile set, falling back to 1 default profile: "default"
2022-03-08 23:47:58.957 INFO 19572 --- [main] o.s.c.cloud.context.scope.GenericScope : BeanFactory id=361a5d3f-a48e-3523-b50d-b595acaf1a18
2022-03-08 23:47:59.334 INFO 19572 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 3000 (http)
2022-03-08 23:47:59.354 INFO 19572 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2022-03-08 23:47:59.354 INFO 19572 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.58]
2022-03-08 23:47:59.515 INFO 19572 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2022-03-08 23:47:59.516 INFO 19572 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1521 ms
2022-03-08 23:48:00.836 INFO 19572 --- [main] o.s.b.a.e.web.EndpointLinksResolver : Exposing 15 endpoint(s) beneath base path '/actuator'
2022-03-08 23:48:00.980 INFO 19572 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 3000 (http) with context path ''
2022-03-08 23:48:01.164 INFO 19572 --- [main] com.cloud.RefreshScopeApplication : Started RefreshScopeApplication in 4.781 seconds (JVM running for 5.837)
2022-03-08 23:48:01.261 WARN 19572 --- [gistrationTask1] d.c.b.a.c.r.ApplicationRegistrar : Failed to register application as Application(name=refresh-scope, managementUrl=http://LAPTOP-AHWS116.ms)
2022-03-08 23:48:02.527 INFO 19572 --- [on(1)-127.0.0.1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2022-03-08 23:48:02.527 INFO 19572 --- [on(1)-127.0.0.1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2022-03-08 23:48:02.528 INFO 19572 --- [on(1)-127.0.0.1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
```

El servicio rest muestra los valores en el siguiente endpoint:

<http://localhost:3000/>

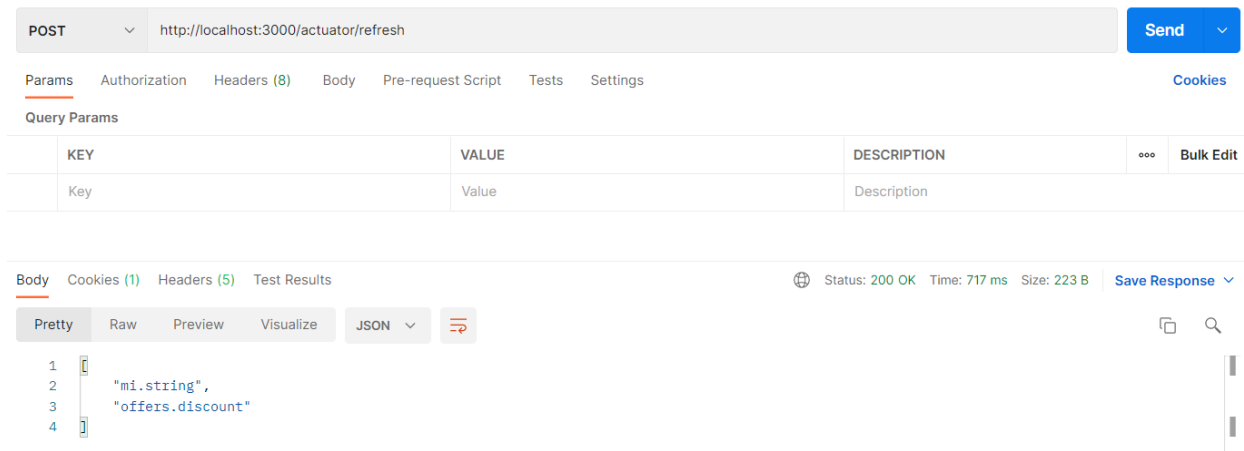
```
← → ↺ ⓘ localhost:3000

Descuento: 50% descuento para el dev
```

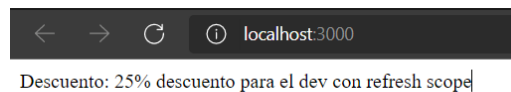
5. Ahora podemos ir a la base de datos y modificar los valores de `offers.discount` o `mi.string`

```
UPDATE `mycrud`.`properties` SET `VALUE` = '25%' WHERE (`id` = '3');  
UPDATE `mycrud`.`properties` SET `VALUE` = 'descuento para el dev con refresh scope' WHERE (`id` = '4');
```

6. Luego hacer un post al endpoint de actuador <http://localhost:3000/actuator/refresh> con postman, ademas se pueden verificar los keys que cambiaron.



El recargar o hacer denuevo la petición get los valores se habrán actualizado, esto sin haber detenido al servicio gracias al refresh scope y actuador.



Documentación para las anotaciones Value y Properties

<https://www.baeldung.com/spring-value-annotation>

<https://www.baeldung.com/properties-with-spring>

Documentación de Commons, Endpoints, Environment y otros.

Cloud Native Applications

Cloud Native is a style of application development that encourages easy adoption of best practices in the areas of continuous delivery and value-driven development. A related discipline is that of building 12-factor Applications, in which development practices are aligned with delivery and operations goals - for instance, by using declarative programming and management and monitoring.

<https://docs.spring.io/spring-cloud-commons/docs/current/reference/html/#endpoints>