

Servidor de Configuración: gestiona configuraciones e integra con un repositorio Git.  
 Microservicios: Consumen configuraciones y escuchan actualizaciones a través de Spring Cloud Bus.  
 Spring Cloud Bus: Conecta el Servidor de Configuración con los microservicios, utilizando un intermediario de mensajes para difundir cambios.  
 Intermediario de Mensajes: Facilita la comunicación en tiempo real (por ejemplo, RabbitMQ o Kafka).  
 Repositorio: Almacena archivos de configuración en un sistema de control de versiones (por ejemplo, Git).

## INTRODUCCIÓN

El objetivo de este proyecto es conseguir que los diferentes microservicios que tengamos puedan actualizar ciertos atributos en tiempo de ejecución, sin la necesidad de reiniciarlos. Esto se conseguirá mediante otros cuatro elementos.

Primero necesitaremos un repositorio que contendrá todas las configuraciones de los distintos micros; estos se pueden organizar en carpetas y en distintas ramas. De esta manera, podemos tener configuraciones distintas para cada parte del desarrollo.

Después, tendremos que crear un microservicio que actuará de Config-Server (CS). Los servicios aplicativos, al iniciarse, se comunicarán con este CS para obtener su respectiva configuración. Cuando esto ocurra, el CS se comunicará con el repositorio que hemos creado anteriormente para obtener el archivo YAML correspondiente y devolverlo al microservicio.

Ahora, cuando creemos los microservicios que simularán a los servicios aplicativos, deberemos añadir distintas dependencias para que actúen como clientes del Config-Server. De esta manera, al iniciarse, irán a buscar su respectiva configuración.

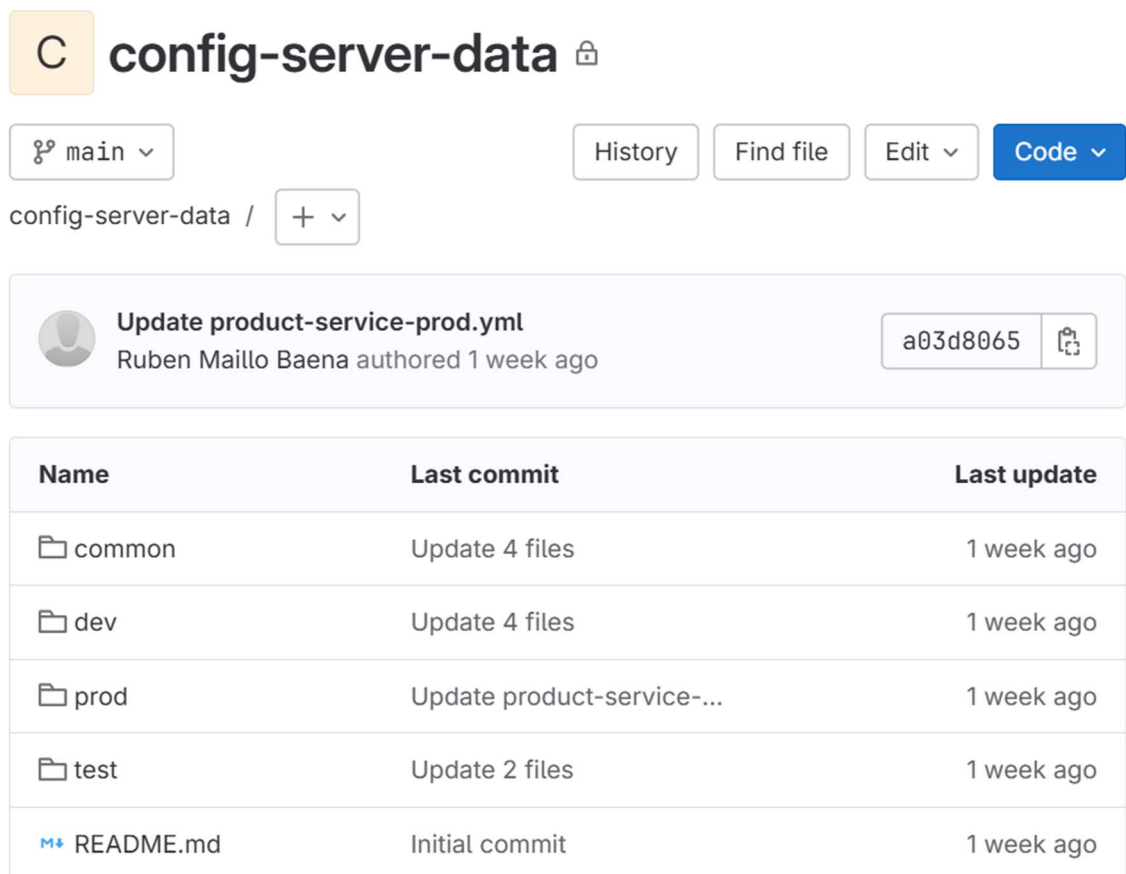
Finalmente, para conseguir la actualización en tiempo de ejecución, usaremos Spring Cloud Bus junto a un Message Broker de nuestra elección (RabbitMQ o Kafka). Para esta POC usaremos RabbitMQ, y este nos permitirá actualizar los diferentes atributos de los microservicios sin tener que reiniciarlos, añadiendo diferentes dependencias tanto en el config-server como en los microservicios.

Ahora entraremos más en detalle.

## IMPLEMENTACIÓN CONFIG-SERVER

### REPOSITORIO:

Para este ejemplo, he usado GitLab y he creado un repositorio para las configuraciones con la siguiente organización:



The screenshot shows the GitLab interface for the repository 'config-server-data'. At the top, there's a navigation bar with 'main' selected, and buttons for 'History', 'Find file', 'Edit', and 'Code'. Below this, the repository path 'config-server-data /' is shown with a '+' button. The main content area displays a commit titled 'Update product-service-prod.yml' by 'Ruben Maillo Baena' from 1 week ago, with the commit hash 'a03d8065'. Below the commit information is a table listing the repository's structure.

Name	Last commit	Last update
common	Update 4 files	1 week ago
dev	Update 4 files	1 week ago
prod	Update product-service-...	1 week ago
test	Update 2 files	1 week ago
README.md	Initial commit	1 week ago

Podemos observar que contamos con distintas carpetas, cada una de las cuales representa un perfil que los microservicios podrán escoger para obtener una configuración u otra.

Estos perfiles pueden servir, por ejemplo, para representar las diferentes partes del desarrollo. Además, hay una carpeta llamada "COMMON", que contiene una configuración general para todos los microservicios en caso de que no tengan ningún perfil especificado.

main config-server-data / common / +



Update 4 files

Ruben Maillo Baena authored 1 week ago

Name	Last commit
..	
.gitkeep	Add new directory
application.yml	Update 4 files

Esta es la carpeta COMMON, que solo contiene un application.yml, el cual, como ya he comentado, es una configuración por defecto para los microservicios.

main config-server-data / common / application.yml



Update 4 files

Ruben Maillo Baena authored 1 week ago

application.yml	72 B
1	custom:
2	timeout: 999
3	max-retries: 99
4	profile: common
5	label: main

Ahora volvamos atrás, al inicio del repositorio y entraremos en la carpeta “dev”:

main config-server-data / dev / +



Update 4 files

Ruben Maillo Baena authored 1 week ago

Name	Last commit
..	
 .gitkeep	Add new directory
 product-service-dev.yml	Update 4 files
 user-service-dev.yml	Update 4 files

Aquí podemos observar que tenemos un archivo de configuración para cada microservicio que lo requiera.

Es importante destacar que el nombre del archivo YAML debe seguir el siguiente formato: {nombre\_de\_la\_aplicacion}-{profile}.yml, de esta manera el config-server podrá distinguir entre todas las configuraciones.

Veamos el user-service-dev.yml como ejemplo:

main config-server-data / dev / user-service-dev.yml



Update 4 files

Ruben Maillo Baena authored 1 week ago



user-service-dev.yml 91 B

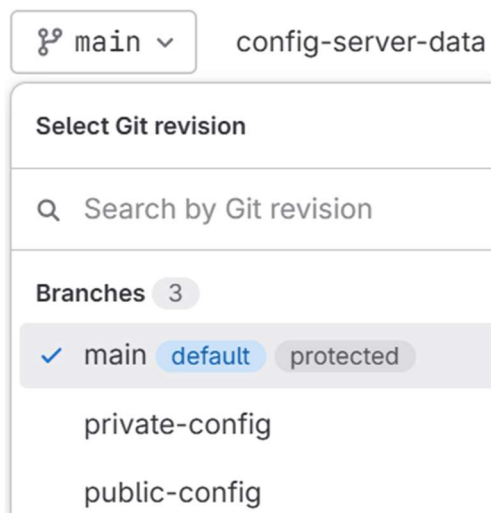
```
1 server:
2   port: 8001
3
4 custom:
5   timeout: 8000
6   max-retries: 2
7   profile: dev
8   label: main
```

Colocamos los atributos que consideremos para esta configuración y el puerto que queramos.

Para las demás carpetas o perfiles que hemos visto al inicio, siguen todos el mismo formato: {nombre\_de\_la\_aplicacion}-{profile}.yml.

Por último, también tenemos la opción de usar las Labels que ofrecen los repositorios. Podemos crear una label directamente o crear distintas ramas, que es lo que he hecho en este ejemplo. A partir del main, creamos dos ramas nuevas: public-config y private-config.

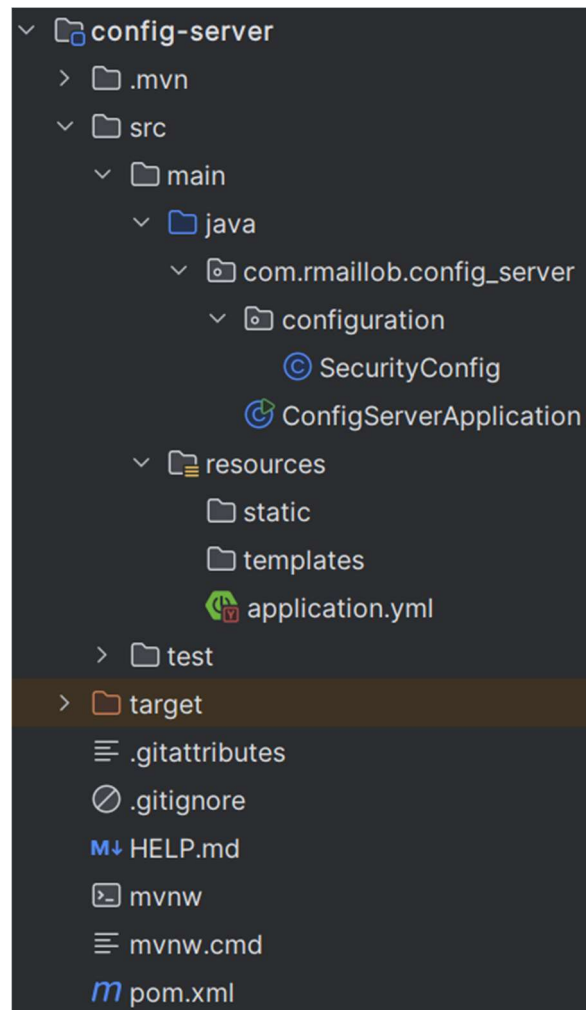
Esto nos permitirá tener muchas más configuraciones; podemos indicarle a un microservicio que tenga la configuración privada o la pública. Dentro de estas ramas, el formato de perfiles será el mismo. Si no escogemos ninguna label, la main será la que se abra por defecto.



Hay que recordar que si el repositorio es privado, para poder acceder fácilmente a través de Spring Boot, podemos crear un token personal que nos permita acceder a repositorios, solo de lectura. De esta manera podremos usarlo de contraseña en nuestro Config-Server.

## CONFIG-SERVER

Para el config-server tendremos la siguiente estructura:



En el pom.xml añadiremos las siguientes dependencias:

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

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

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

Y en el Application añadiremos la siguiente anotación:

```
@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {

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

}
```

Por ultimo en el application.yml añadiremos la siguiente configuración:

```
server.port: 8888
spring.application.name: config-server

### NTT DATA - GITLAB REPOSITORY
spring.cloud.config.server:
  accept-empty: true
  git:
    uri: https://umane.emeal.nttdata.com/git/BSISPROTBSISARQUITEC/config-server-data
    search-paths:
      - common
      - test
      - dev
      - prod
    username: rmaillob
    password: -h6dCWNfG1B4NN8cVHbp

spring.security.user:
  name: root
  password: root
```

Los search-path se encarga de saber los distintos perfiles que existen. Se carga de arriba hacia abajo. Es decir, si ponemos primero el common este será el que haya por defecto y viceversa.

Esta configuración se conecta al repositorio, usa el token como contraseña ya que es privado ya demás añade seguridad básica de spring. Los micros deberán usar el nombre y contraseña para acceder a las configuraciones.

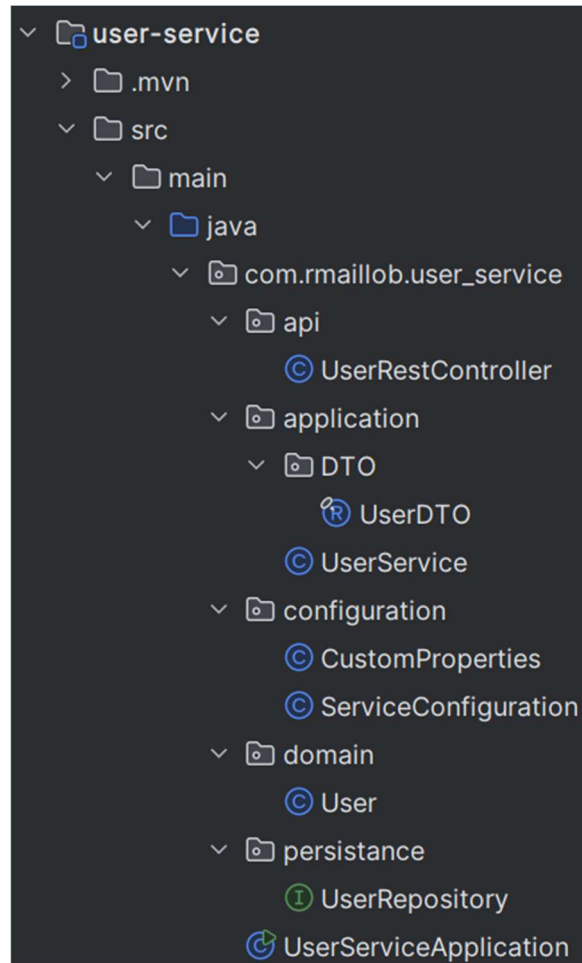
## MICRO-SERVICIOS

Para esta POC, tendremos dos microservicios de ejemplo:

- User-service
- Product-service

Estos contienen lo básico para poder crear Usuarios y Productos a través del API Rest y que se guarden en una base de datos local como puede ser H2. También

contienen un application.yml donde deberemos añadir diferentes configuraciones que explicare mas adelante.



Primero deberemos añadir las siguientes dependencias:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
  <version>4.2.2</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```



Y en el application.yml añadiremos lo siguiente:

```
spring:
  application:
    name: user-service
  profiles:
    active:
      - test
      - dev
      - prod
  config:
    import: optional:configserver:http://root:root@localhost:8888
    cloud.config.label: private-config
```

Es importante que el nombre coincida con los archivos de configuración. Vemos que podemos establecer los perfiles de los que queremos la configuración. Al igual que en el config.server, estos se cargan de arriba abajo. Por lo tanto, primero cargara el test, luego el dev y por ultimo se quedara con la configuración de prod. Se van sobrescribiendo.

Añadimos la URL al config-server y ponemos la contraseña y nombre (root:root). Finalmente podemos especificarle que label queremos usar. (En nuestro caso, la de la main, private o public).

## IMPLEMENTACIÓN SPRING CLOUD BUS & RABBIT-MQ

Ahora solo queda la parte de actualizar los micros con la base que ya tenemos creada.

### CLIENT ACTUATOR:

Para esto añadiremos la siguiente dependencia en los MICROSERVICIOS:

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

Ahora en el archivo de “Custom Properties” que tenemos en nuestros microservicios, podemos añadir todos los atributos que queramos tener. Además añadiremos la anotación de @RefreshScope.

```
@Component 2 usages
@Getter
@Setter
@RefreshScope
@ConfigurationProperties(prefix = "custom")
public class CustomProperties {

    private int timeout;
    private int maxRetries;
    private String profile;
    private String label;
}
```

Con el @ConfigurationProperties y el prefijo, le estamos indicando donde estarán situados estos atributos en los archivos de configuración. Es decir, nosotros tenemos “custom” por lo tanto cuando se inicie el micro y obtenga la configuración, buscare el prefijo “custom” y asignara los valores respectivamente en esta clase.

```
user-service-dev.yml 91 E
1  server:
2    port: 8001
3
4  custom:
5    timeout: 8000
6    max-retries: 2
7    profile: dev
8    label: main
```

Cabe destacar que en el archivo yaml, los nombre tiene que estar separados con guiones, pero en la clase de atributos del micro tiene que estar en camelCase. Para que Spring Cloud los identifique por si mismo.

Ahora el actuador que hemos añadido con las dependencias nos proporciona diferentes endpoints que podemos utilizar para actualizar el microservicio. Es posible que por defecto estos estén bloqueados, es por esto que podemos añadir la siguiente configuración en el application.yml del microservicio:

```
#spring actuator: endpoint --> /actuator/refresh (actualiza solo este microservicio)
management:
  endpoints:
    web:
      exposure:
        include:
          - "*"

```

De esta manera exponemos todos los endpoints. Ahora mismo si iniciamos todo, y llamamos a este endpoint de este microservicio :

`http://localhost:{puerto}/actuator/refresh`

Volverá a buscar su configuración y actualizará sus atributos si han cambiado. Sin la necesidad de reiniciar el micro.

## SPRING CLOUD BUS & RABBIT\_MQ:

Priemero vamos a configurar rabbitMQ. Una vez lo tengamos instalado si nos da problemas, podemos parar el servicio para volver a iniciarlo: (Se supone que con la instalación viene una terminal de rabbit que puedes ejecutar, pero en mi caso no funcionaba, así que use la terminal de windows.)

```
C:\Program Files\RabbitMQ Server\rabbitmq_server-4.1.0\sbin>rabbitmq-service.bat stop
El servicio de RabbitMQ está deteniéndose.
El servicio de RabbitMQ se detuvo correctamente.

C:\Program Files\RabbitMQ Server\rabbitmq_server-4.1.0\sbin>rabbitmq-service.bat start
El servicio de RabbitMQ está iniciándose.
El servicio de RabbitMQ se ha iniciado correctamente.

C:\Program Files\RabbitMQ Server\rabbitmq_server-4.1.0\sbin>

```

Antes o después de iniciar el servicio, no lo recuerdo bien. Debemos añadir el siguiente plug in de rabbit para poder acceder a la consola:

```
C:\Program Files\RabbitMQ Server\rabbitmq_server-4.1.0\sbin> rabbitmq-
plugins.bat enable rabbitmq_management

```

Ahora podemos acceder a la consola de rabbitMQ para comprobar que funciona:  
<http://localhost:15672>

Accederemos a la consola de rabbit para poder crear los exchanges y las colas necesarias para nuestros micros:

OverviewConnectionsChannelsExchangesQueues and StreamsAdmin

Queues

All queues (3)

Page 1 of 1 - Filter:  ☐ Regex ?

Displaying 3 items , page size up to: 100

Overview					Messages			Message rates				+/-
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack		
/	Config-Server-Queue	classic	D Args	Idle	0	0	0					
/	Product-Service-Queue	classic	D Args	Idle	0	0	0					
/	User-Service-Queue	classic	D Args	Idle	0	0	0					

Add a new queue

Una cola para cada microservicio que tengamos.

OverviewConnectionsChannelsExchangesQueues and StreamsAdmin

Exchange: springCloudBus

Overview

Message rates last ten minutes ?

Currently idle

Details

Type	topic
Features	durable: true
Policy	

Bindings

This exchange

⇓

To	Routing key	Arguments	
Config-Server-Queue	#		Unbind
Product-Service-Queue	#		Unbind
User-Service-Queue	#		Unbind

Ademas crearemos el siguiente Exchange y añadiremos los binding a las colas que hemos creado anteriormente, no tenemos que poner argumentos y de routing key debemos poner un #.

Ahora una vez tenemos rabbit funcionando. En los microservicios deberemos añadir la siguiente dependencia:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-bus-amqp</artifactId>
  <version>4.2.1</version>
  <exclusions>
    <!-- Para evitar conflicto entre commons logging y spring-jcl -->
    <exclusion>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Ademas de la dependecia tuve que añadir una exclusion de commons-logging, ya que me generaba un conflicto con spring-jcl. (No estoy seguro de si es generad por otras dependencias, pero excluirlo aquí lo solucionaba).

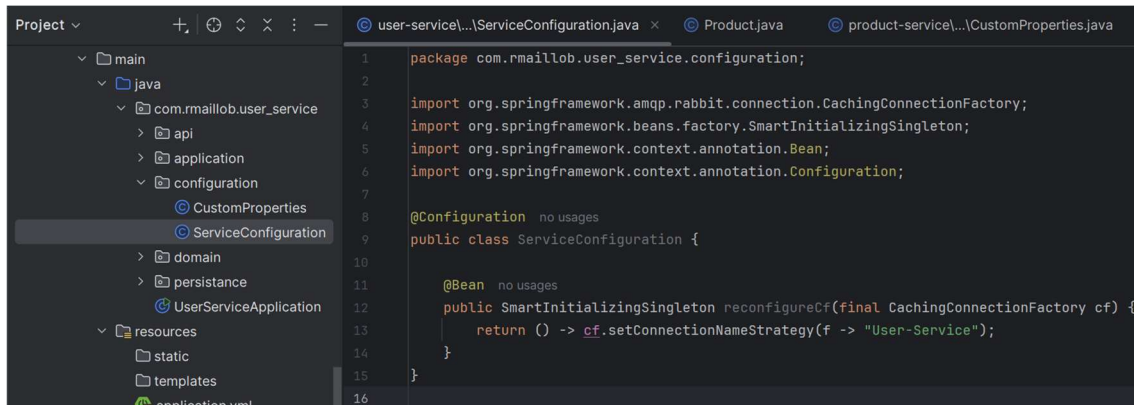
Ahora añadiremos la siguiente configuracion en su application.yml:

```
#Spring bus & RabbitMQ: endpoint --> /actuator/busrefresh (actualiza todos los microservicios)
---
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: user-service
    password: guest
  cloud:
    bus:
      enabled: true
      refresh:
        enabled: true
    stream:
      bindings:
        springCloudBusInput:
          destination: springCloudBus
          group: User-Service-Queue
      rabbit:
        bindings:
          springCloudBusInput:
            consumer:
              queue-name-group-only: true
```

En destination ponemos el nombre del exchange y en el group la cola que usaran. El nombre debe coincidir.

Para poder identificarlos mejor, les añadi perfiles dentro de rabbitMQ. Es por eso que el username es diferente a guest.

Ademas quise que tuvieran un nombre de conexión específico, así que podemos añadir lo siguiente en una clase de configuración:



```
1 package com.rmaillob.user_service.configuration;
2
3 import org.springframework.amqp.rabbit.connection.CachingConnectionFactory;
4 import org.springframework.beans.factory.annotation.Bean;
5 import org.springframework.context.annotation.Configuration;
6
7 @Configuration
8 public class ServiceConfiguration {
9
10     @Bean
11     public SmartInitializingSingleton reconfigureCf(final CachingConnectionFactory cf) {
12         return () -> cf.setConnectionNameStrategy(f -> "User-Service");
13     }
14 }
15
16
```

Ahora si iniciamos todo y llamamos al endpoint del actuador del microservicio:

/actuador/busrefresh

Se actualizarán todos los microservicios que estén suscritos. Sin tener que ir uno por uno.

Para finalizar, nos falta añadir un par de cosas en el config-server. Primero, añadiremos las siguientes dependencias:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-monitor</artifactId>
  <version>4.2.2</version>
  <scope>runtime</scope>
</dependency>

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
  <version>4.2.1</version>
</dependency>
```

Y ahora deberemos añadir la siguiente configuración en su application.yml:

```
#RabbitMQ
---
spring.cloud:
  bus.enabled: true
  stream:
    bindings:
      springCloudBusInput:
        destination: springCloudBus
        group: Config-Server-Queue
    rabbit:
      bindings:
        springCloudBusInput:
          consumer:
            queue-name-group-only: true

spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: config-server
    password: guest
```

Ahora, con la dependencia de monitor que hemos añadido, nos proporciona un endpoint que podemos llamar desde un webHook de nuestro repositorio. Crearemos un WEBHOOK que se active cuando hagamos un push en cualquiera de los archivos de configuración. Cuando se active, llamará al siguiente endpoint:

<http://localhost:8888/monitor>

Esto hará que el config-server busque que archivos se han actualizado y si afectan a los microservicios activos. Si es así, actualizará solo los microservicios que tengan algo distinto de manera automática, los demás no.

De esta manera solo tenemos que hacer un cambio en el repo para que se actualice todo solo.

Por ultimo, el monitor me daba problema de autorizacion cuando lo llamaba, por eso tuve que añadir la siguiente configuracion:

```
@Configuration no usages
@EnableWebSecurity
public class SecurityConfig {

    //TODO: la solicitud http://root:root@localhost:8888/monitor da error 401, unauthorized.
    //TODO: Desactivamos el CSRF para que pueda processar la solicitud de /monitor
    @Bean no usages
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .csrf(csrf -> csrf
                .ignoringRequestMatchers( ...patterns: "/monitor")
            );
        return http.build();
    }

    @Bean no usages
    public SmartInitializingSingleton reconfigureCf(final CachingConnectionFactory cf) {
        return () -> cf.setConnectionNameStrategy(f -> "Config-Server");
    }
}
```

De esta manera lo ignoraba para el endpoint de monitor.

## EXTRA

Los webhooks no permiten llamar a localhost, por lo tanto deberiamos tener una herramienta de tunneling para exponer nuestro puerto local. O bien desplegarlo, solo si lo estamos probando en un entorno local como es mi caso.