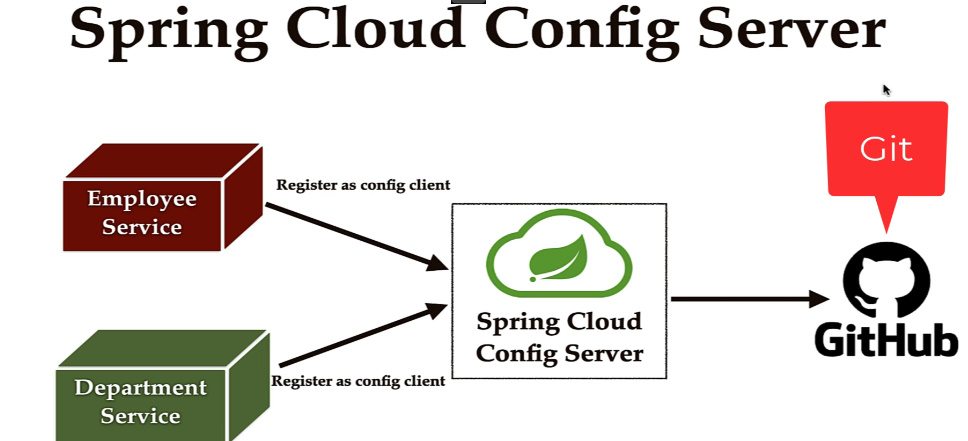


* Fie microservice1,2 si 3. Microservice1 are inca 2 instante care ruleaza. El, ca si celelalte microservice, are un config file(properties), si logic si ca instatele sale tot il au.
* Acum, daca vom modifica acel config file, va trebui sa dam restart la acest Microservice, si instantele sale
* Asta nu e o idee buna.
* Cu **Spring Config Server** nu trebuie sa dam restart la microservice si la toate instantele sale
* Totodata, cu **Spring Config Server** putem externaliza config file(properties), la fiecare microservice intr-un central repository
* Odata ce toate config files(properties), sunt pe un repository central, daca facem vreo modificare la microservicii, nu trebuie sa mergem la fiecare dintre ele ca sa modificam config file separat,ci o facem doar in respository
* **Deci, Spring Config Server rezolva 2 probleme:**

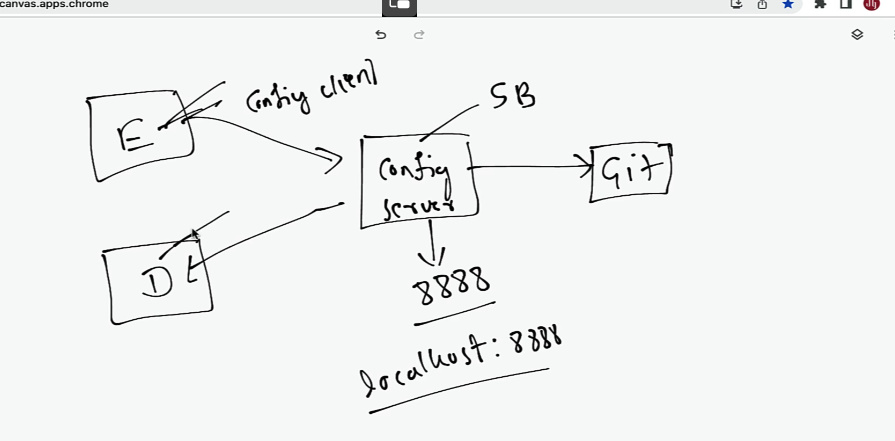
1. Nu e necesar sa dam restart la fiecare microservice si instantele sale la modificarea la config file
2. Config files(properties), la microservicii sunt puse intr-un repository

**Spring Cloud Config Server**



Deci, services vor lua config file(properties), deja din Spring Cloud Config Server, care are grija ca ele sa fie luate si pastrate in git repository

* Spring Cloud Config Server este o Spring Boot application tot, adica un project separat si un Eureka client
* El doar ia config file din repository si le ofera la services



* Apare intrebarea cum microserviciile vor folosi acum host si port de la config server
* Asta se va face prin **config client dependency,** care, cand un service ruleaza, are grija sa preia si injecteze config file pentru microservice din config server

**Spring Cloud Config Server configuration**

1. **Cream un nou microservice ce va contine dependentele:**

- eureka client

- spring cloud config server

**- actuator(cu web exposure la refresh)**

**2. Activam spring cloud config server**

@EnableConfigServer

**3, Il facem eureka client prin a ne conecta la eureka server**

server.port=8888  
  
spring.application.name=CONFIG-SERVER  
  
eureka.client.service-url.defaultZone = <http://localhost:8761/eureka/>

**4, Cream Git repository pentru config server**

spring.cloud.config.server.git.uri=https://github.com/kerbecs/config-server-repo  
spring.cloud.config.server.git.clone-on-start=true

spring.cloud.config.server.git.default-label=main

- spring.cloud.config.server.git.clone-on-start – va clona in project repositoriul dat la startare de fiecare data, deci daca e un nou commit, il va pune mereu la startare

- spring.cloud.config.server.git.default-label – numele la branch spre care sa pointeze

**5, Modificam service ca sa fie client pentru config server**

- trebuie dependenta ca sa fie client config

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

si in noile versiuni de spring cloud:

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

- acum mutam fisierele .properties in git repository, si le oferim numele sub forma:

NUME-MICROSERVICE.properties sau

nume-microservice.properties

- nu stocam proprietatea

spring.application.name=DEPARTMENT-SERVICE

Nu e nevoie de ea in remote .properties, caci deja este in local properties si se va repeta. Acum stergem tot ce e in application.properties local, cu exceptia la

spring.application.name=DEPARTMENT-SERVICE

Cand microservice va rula, el va cauta un fisier in repository cu numele la aplicatie .properties, numele fiind luat din spring.application.name pus local, si pe baza lui, va prelua fisierul .properties din repository si va adauga proprietatile de acolo, adica va face un injection. Lower sau uppercase, nu conteaza.

**6. Conectam acum service cu config server:**

spring.config.import=optional:configserver:http://localhost:8080

**MAI BINE FOLOSESTE IN LOC DE spring.config.import in bootstrap.properties:**

spring.cloud.config.uri=http://localhost:8888

**ATENTIE!!! spring.cloud.config.uri va merge in bootstrap.properties doa!!!**

asta o punem in application.properties local. Anume va folosi acest config server pentru a cauta fisierul .properties

ATENTIE! Uneori nu se ia Ip, ci un mac, de aceea ne uitam in consola la config server ce host si port ne da, caci poate fi necesar sa inlocuim asa host si port pentur config server

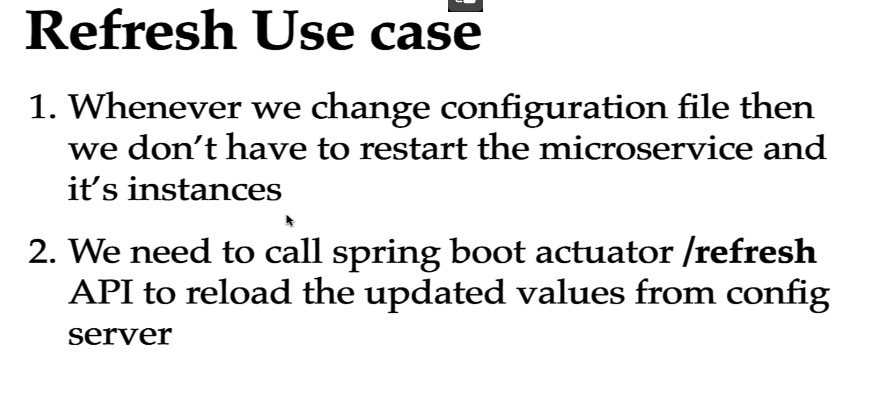
spring.config.import=optional:configserver:WIN-L14IC1A9JH7:8888

**MAI BINE FOLOSESTE IN LOC DE spring.config.import:**

spring.cloud.config.uri=http://localhost:8888

**ATENTIE!!! spring.cloud.config.uri va merge in bootstrap.properties doa!!!**

**Refresh UseCase**



Deci,iata cam ce va trebui sa facem:

1. Adaugam dependenta pentru actuators, ca sa avem acces la /actuator/refresh
2. Spring Cloud Config Server foloseste deja si bootstrap. Deci, scoatem asta din application.properties

~~spring.application.name=DEPARTMENT-SERVICE  
spring.cloud.config.uri=http://localhost:8888~~

si lasam cam doar asta

management.endpoints.web.exposure.include=\*

Si cream in resources un fisier bootstrap.properties, si mutam in ele numele la service actual si url la config server

spring.application.name=DEPARTMENT-SERVICE  
spring.cloud.config.uri=http://localhost:8888

bootstrap.properties e parintele lui application.properties, si el se incarca mereu inaintea lui. El e folosit cand stocam fisierele de configurare undeva remote. Putem face asta si in application.properties cum am facut, dar nu vom putea avea refresh cat aplicatia inca ruleaza!!!

1. Asta e tot, acum mai folosim doar anotatia **@RefreshScope**

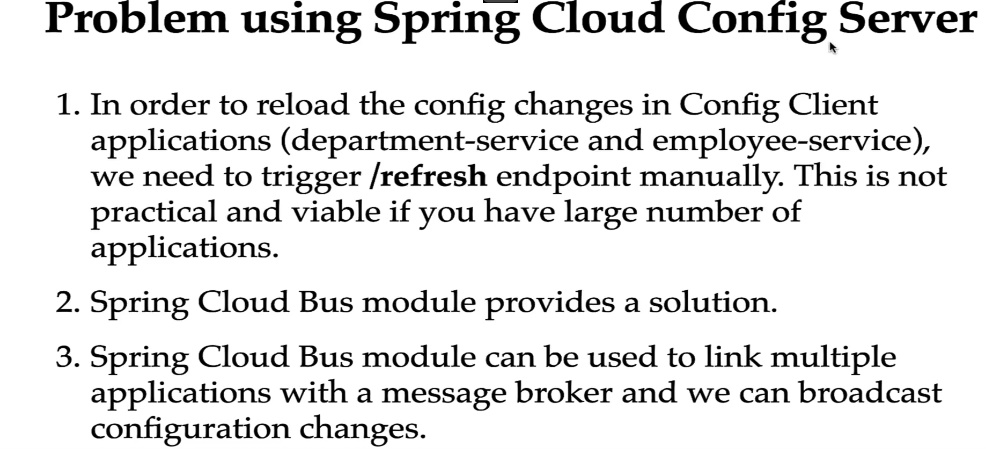
@RefreshScope  
@RestController  
@Getter  
@Setter  
public class MessageController {  
  
 @Value("${message.test}")  
 private String test;  
  
 @GetMapping("/message")  
 public String getMessage(){  
 return test;  
 }  
}

1. Acum, daca modificam application.properties remote, vom face

POST : localhost:8080/actuator/refresh, si apoi chemam /message inapoi si gata, avem noul messaj fara a da restart la aplicatie.

Ideea e ca mereu se stocheaza local un commit din remote, si cu el config server lucreaza. Odata ce dam /refresh, se adauga noul commit, dar microservice inca are referintal la cel vechi, desi stie ca este si unul nou. Anum @RefreshScope spune ca acel bean va lucra mereu cu cel mai recent commit local, nu cu cel la care are referinta in general microservice

**Problema cu Spring Cloud Server Config**

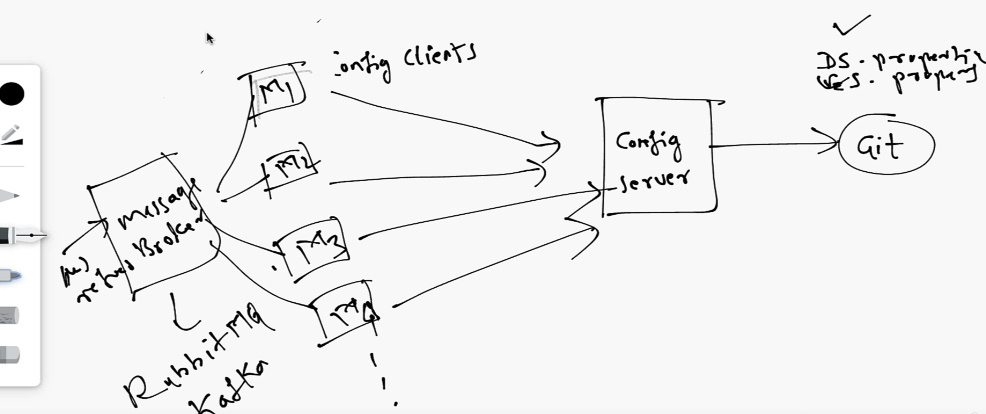


Deci cea mai mare problema ramane ca trebuie mereu sa apelam /refresh pentru fiecare microservice, si asta nu e prea practic.

* O solutie este Spring Cloud Bus

**Spring Cloud Bus**

* Sprinb Cloud Bus poate comunica cu un message broker(kafka,rabbitmq), si fiecare service este un Observer al acestui message brokers. Asa el leaga mai multe aplicatii cu un message broker
* Deci, cand apare vreo modificare, spring cloud bus va avea grija sa apeleze message broker, care va avea si el grija sa anunte observatorii, adica services ,de asta si ele vor da singure /refresh
* Trebuie doar sa apelam /busrefresh actuator si gata, message broker va anunta toate microservices ca e timpul de /refresh



Pasi de dezvoltare:

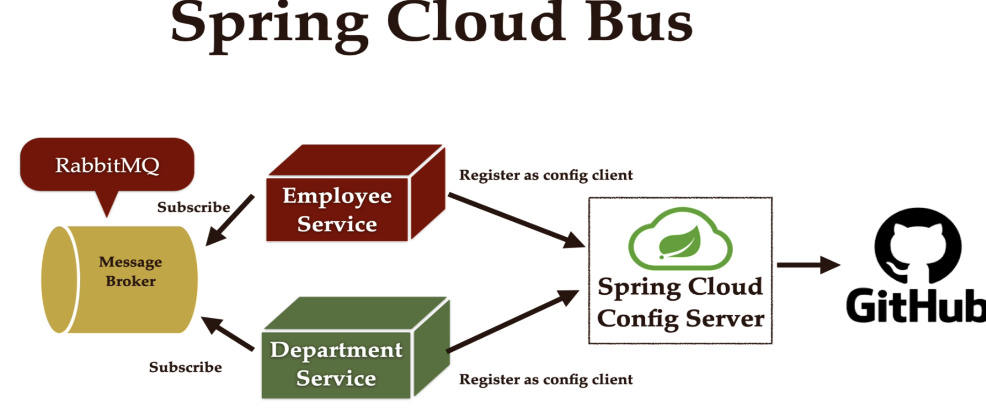
1. **Punem dependenta pentru spring cloud bus in fiecre microservice(Department si Employee)**

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-bus-amqp</artifactId>  
</dependency>

1. Adaugam asta:

spring.cloud.config.name=GATEWAY-SERVICE

1. **Instalam RabitMQ cu docker**



username si parola vor fi guest

docker run -it -p 5672:5672 -p 15672:15672 -e RABBITMQ\_DEFAULT\_USER=guest -e RABBITMQ\_DEFAULT\_PASS=guest rabbitmq:3-management

* ne conectam la interfata grafica cu loalhost:15672, dar in application folosim 5672

1. **Adaugam in department si employee service conexiune la rabbitmq:**

spring.rabbitmq.host=localhost  
spring.rabbitmq.port=5672  
spring.rabbitmq.username=guest  
spring.rabbitmq.password=guest

1. Acum, cand vom face vreo modificare in .properties remote la oricate services, vom apela doar /**actuator/busrefresh** folosind host si port la oricare microservice, ca e department, sau employee, oricum se va da refresh la toate prin message broker. Deci, busrefresh va trimite un event la messagebroker, care va trimite la toate observer sa dea /refresh. Nu uitam ca e valabil doar pentru beanuri cu anotatia @RefreshScope

**Spring Cloud Bus automat foloseste RabbitMQ si are deja o dependenta pusa pentru el.**