Centralizing Configuration for Microservices with Spring Cloud Config

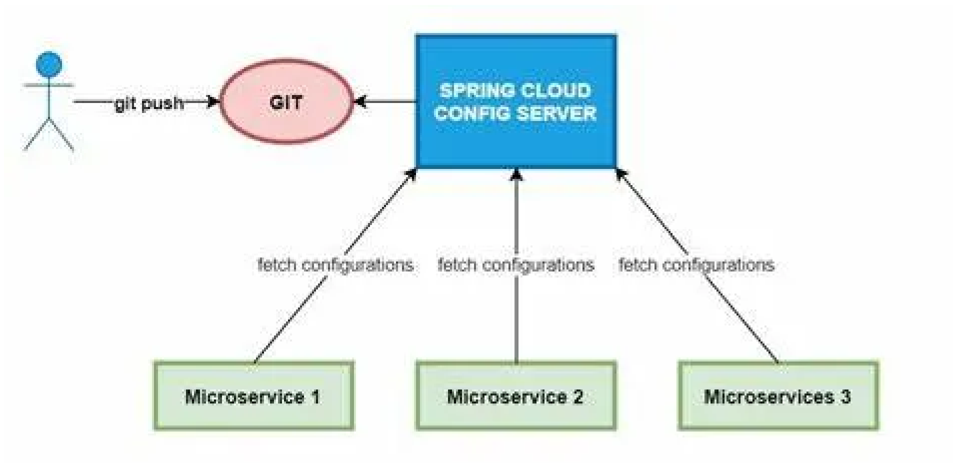
Kishore Karnatakapu · Follow
4 min read · Apr 24

79

Spring Cloud Config Server is a powerful tool for managing configuration in a microservices architecture. It allows to centralize configuration data and make it easily accessible to all microservices.

With Spring Cloud Config Server, we can store the configuration data in a Git repository or any other external configuration source. Microservices can access this data through REST endpoints or through client libraries provided by Spring Cloud.

One of the benefits of using Spring Cloud Config Server is that it allows us to manage configuration data dynamically. We can update configuration data without having to restart microservices, which can save time and reduce downtime.

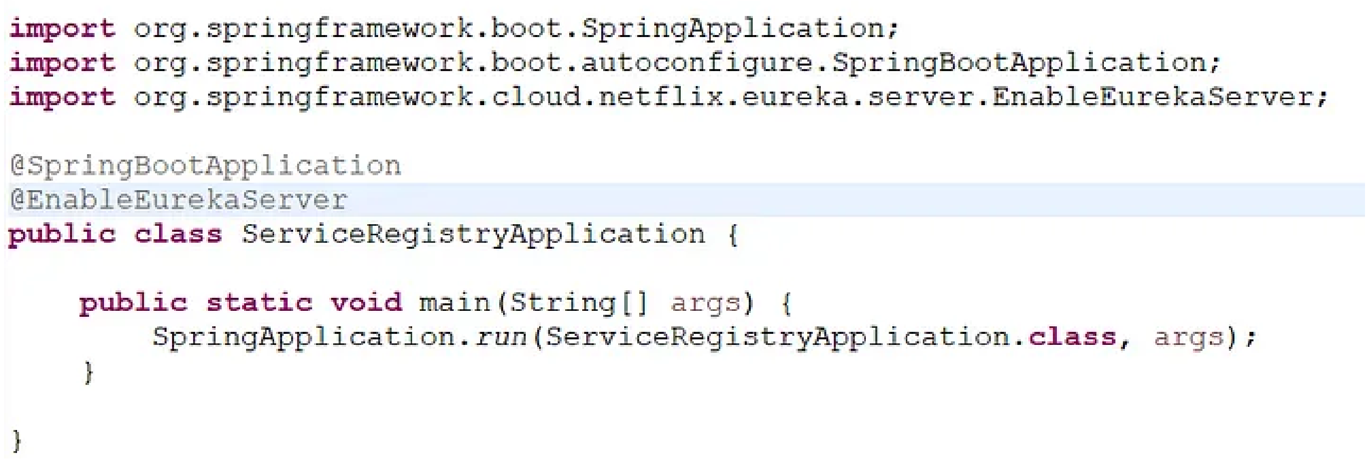


Implementation of Spring Cloud Config:

Create a Microservice (Eureka Server):

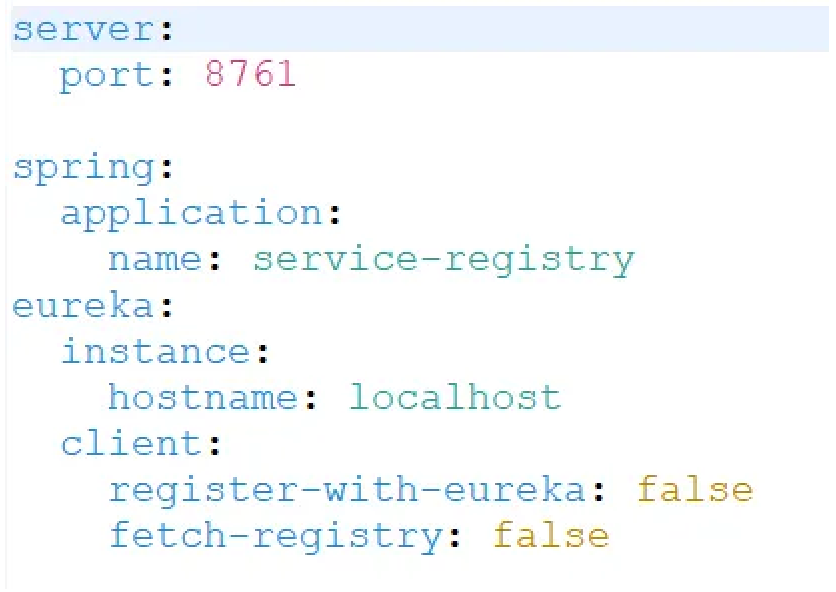
Eureka Server acts as a central registry for all the microservices in the system, allowing them to discover and communicate with each other.

In order to recognize that this is a Eureka Server, We need to add @EnableEurekaServer in the main class of application.



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

@SpringBootApplication
@EnableEurekaServer
public class ServiceRegistryApplication {

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

}
```

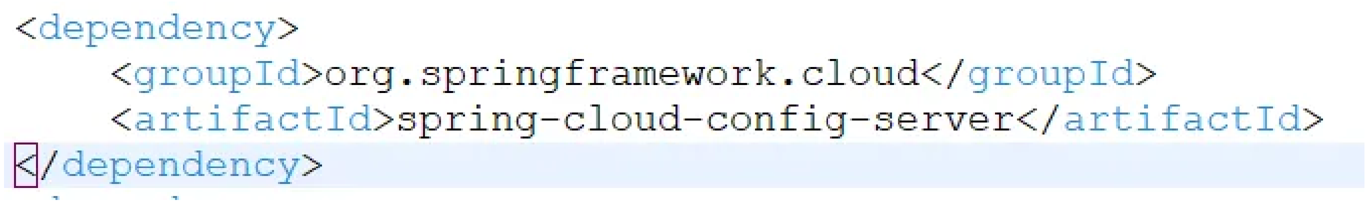
Modify application.yaml or application.properties file



```yaml
server:
  port: 8761

spring:
  application:
    name: service-registry
eureka:
  instance:
    hostname: localhost
  client:
    register-with-eureka: false
    fetch-registry: false
```

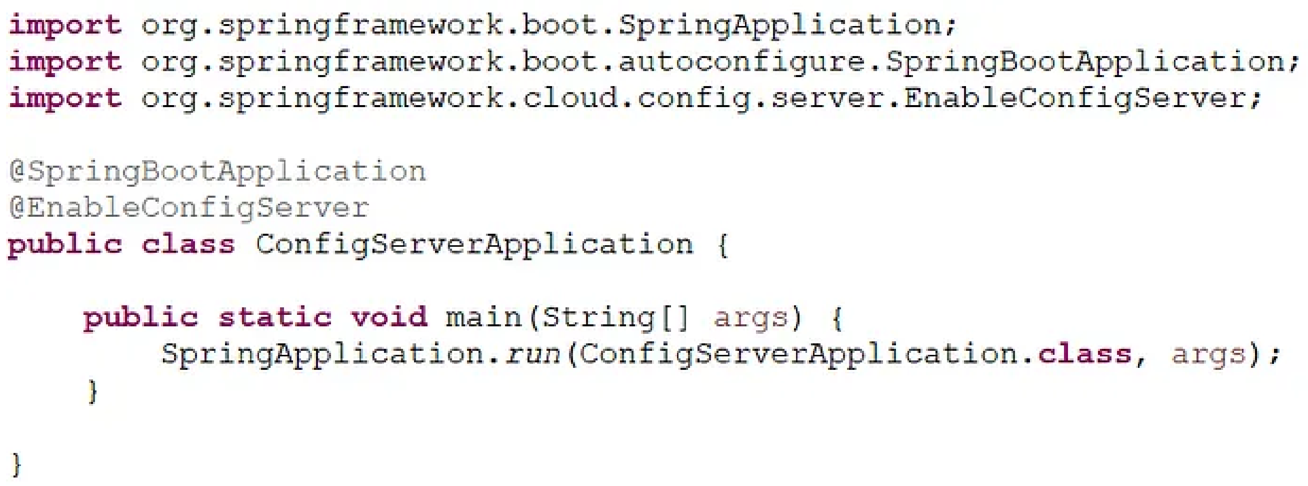
Create Config Server:

Add dependency in POM.XML



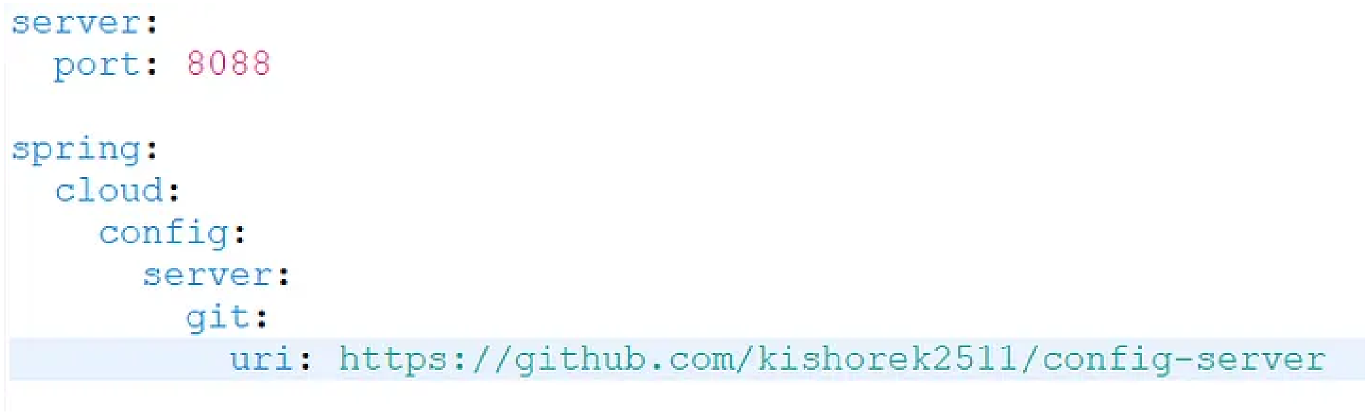
```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

Add annotation @EnableConfigServer in the main class



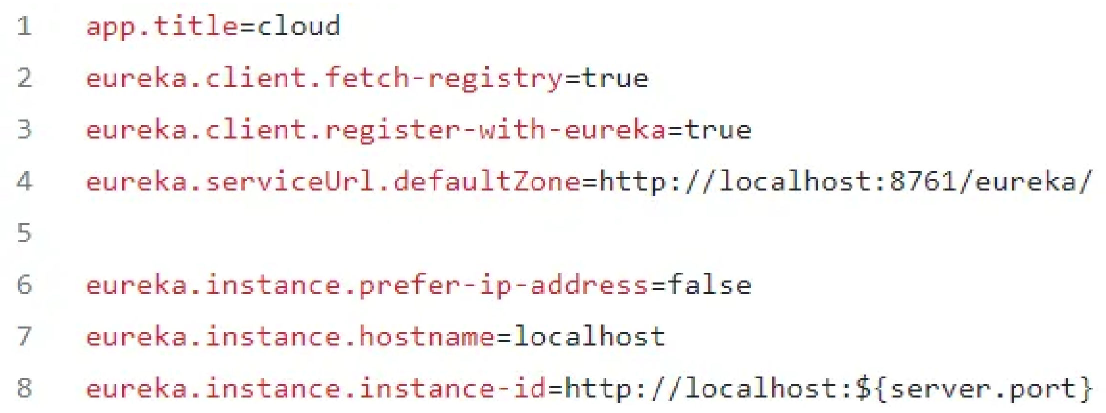
```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;

@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {

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

}
```

Modify application.yaml or application.properties file



```yaml
server:
  port: 8088

spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/kishorek2511/config-server
```

Create a new repository in git and add properties file, give the repository url in above yaml file.

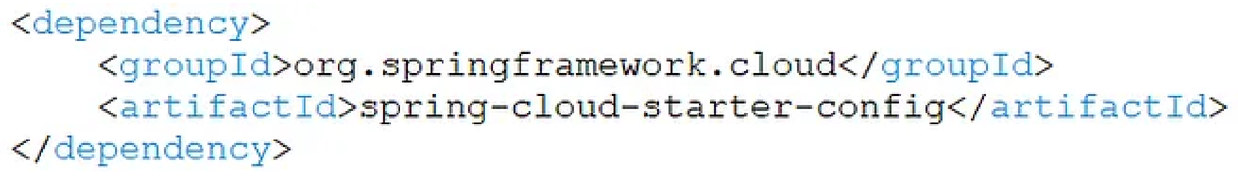


```properties
app.title=cloud
eureka.client.fetch-registry=true
eureka.client.register-with-eureka=true
eureka.serviceUrl.defaultZone=http://localhost:8761/eureka/

eureka.instance.prefer-ip-address=false
eureka.instance.hostname=localhost
eureka.instance.instance-id=http://localhost:${server.port}
```

application.properties file in git repository

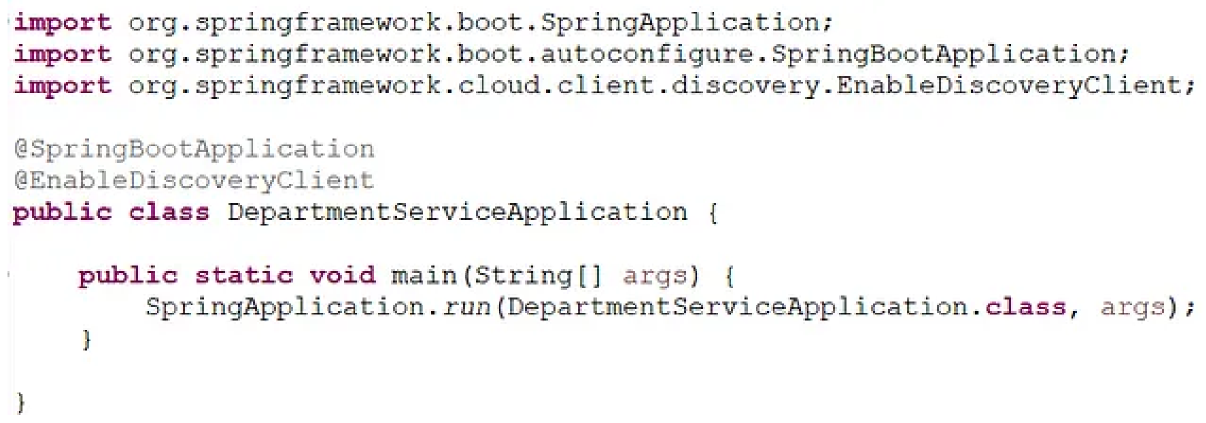
Create Config Client Microservice:

Add dependecny in POM.XML



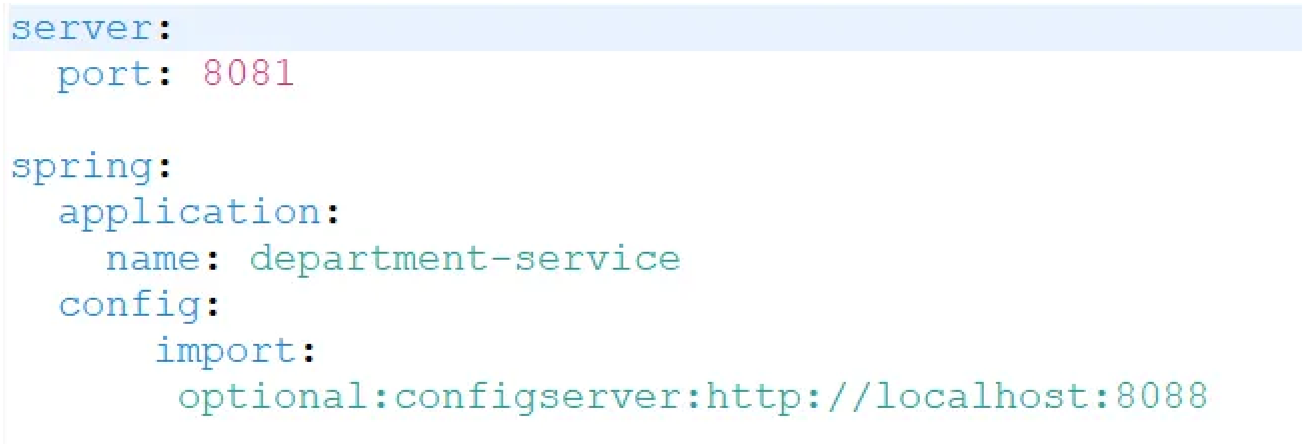
```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

Add annotation @EnableDiscoveryClient in the main class



```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication
@EnableDiscoveryClient
public class DepartmentServiceApplication {

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

}
```

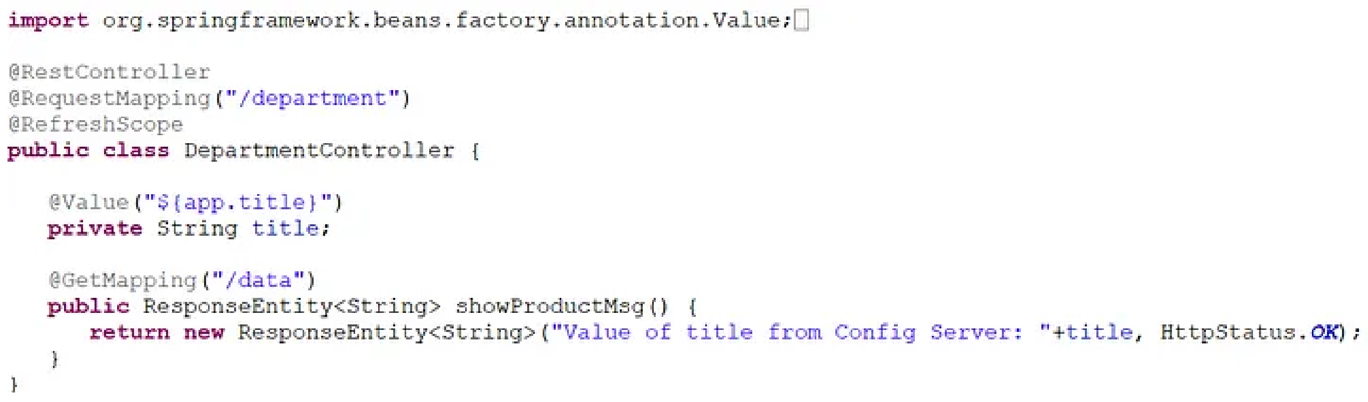
Modify application.yaml or application.properties file



```yaml
server:
  port: 8081

spring:
  application:
    name: department-service
  config:
      import:
       optional:configserver:http://localhost:8088
```

In above yaml file, we are importing the config server microservice properties which will redirect to cloud config where we added common properties.

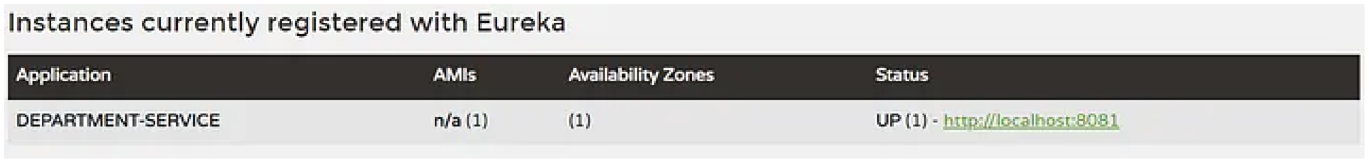
Create a controller in department service microservice(config client):



```java
import org.springframework.beans.factory.annotation.Value;

@RestController
@RequestMapping("/department")
@RefreshScope
public class DepartmentController {

   @Value("${app.title}")
   private String title;

   @GetMapping("/data")
   public ResponseEntity<String> showProductMsg() {
      return new ResponseEntity<String>("Value of title from Config Server: "+title, HttpStatus.OK);
   }
}
```

Testing the Implementation:

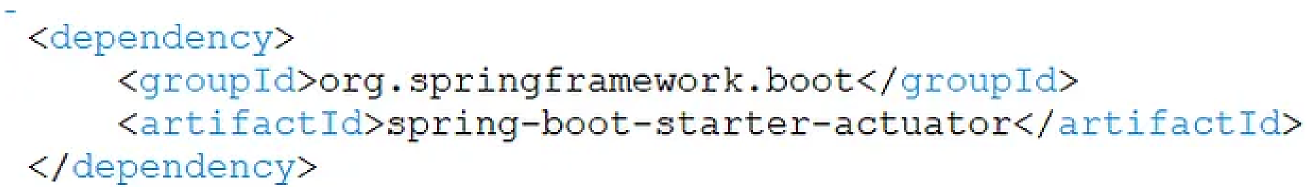
Start the eureka server, config server, config client service. Config client service should display in eureka dashboard.



Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
DEPARTMENT-SERVICE	n/a (1)	(1)	UP (1) - http://localhost:8081

Inorder to get updated values from config server to config client microservice without restarting the application we have to add actuators to config client service.



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

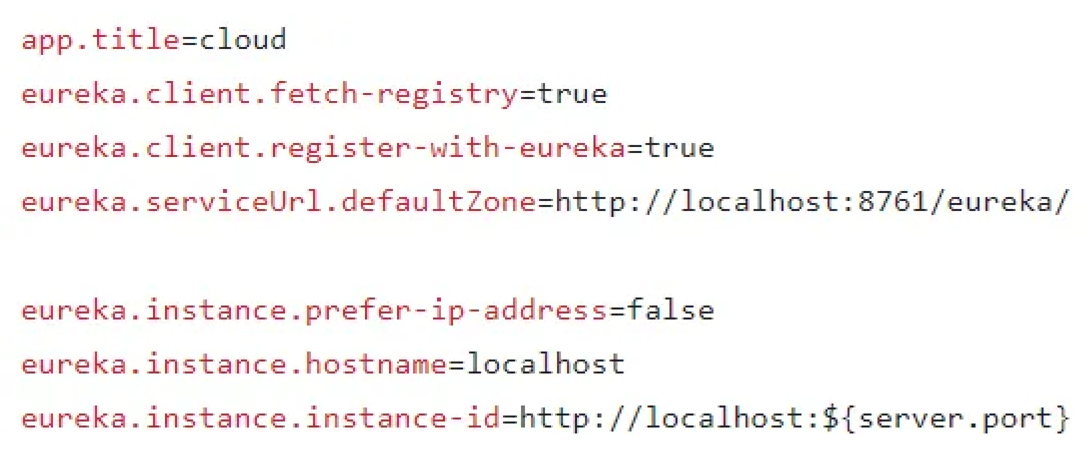
Actuator dependency

Add @RefreshScope annotation in controller, @RefreshScope is a Spring Cloud annotation that can be used in microservices to refresh the configuration without restarting the microservice.

To test this open Postman tool.

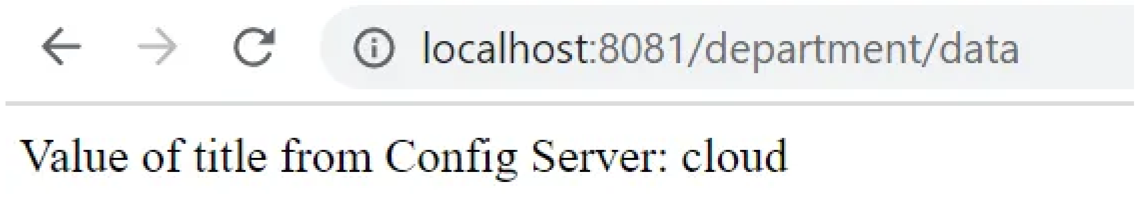
Change the value of property in GitHub and commit the changes. In our example its ‘app.title’.

Before changing the app.title in application.properties file



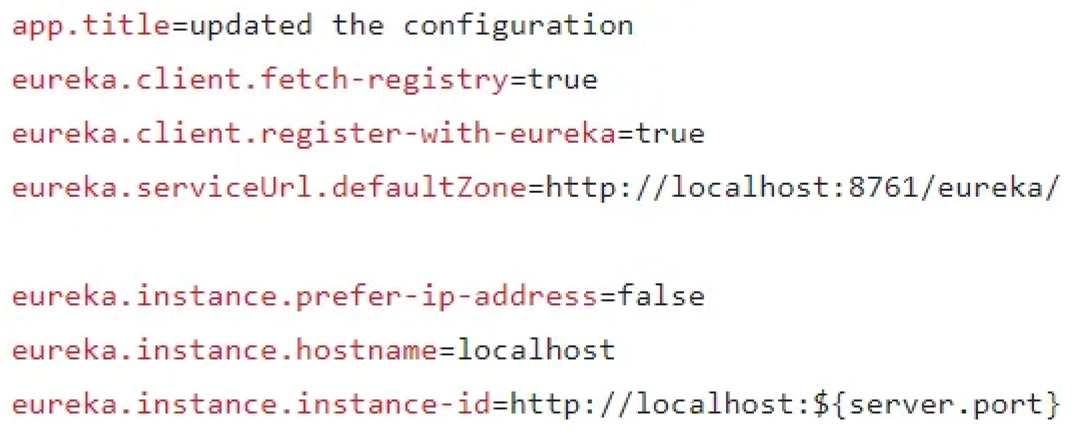
```
app.title=cloud
eureka.client.fetch-registry=true
eureka.client.register-with-eureka=true
eureka.serviceUrl.defaultZone=http://localhost:8761/eureka/

eureka.instance.prefer-ip-address=false
eureka.instance.hostname=localhost
eureka.instance.instance-id=http://localhost:${server.port}
```

Open your browser and hit the actual URL(http://localhost:8081/department/data)You will see the updated value.



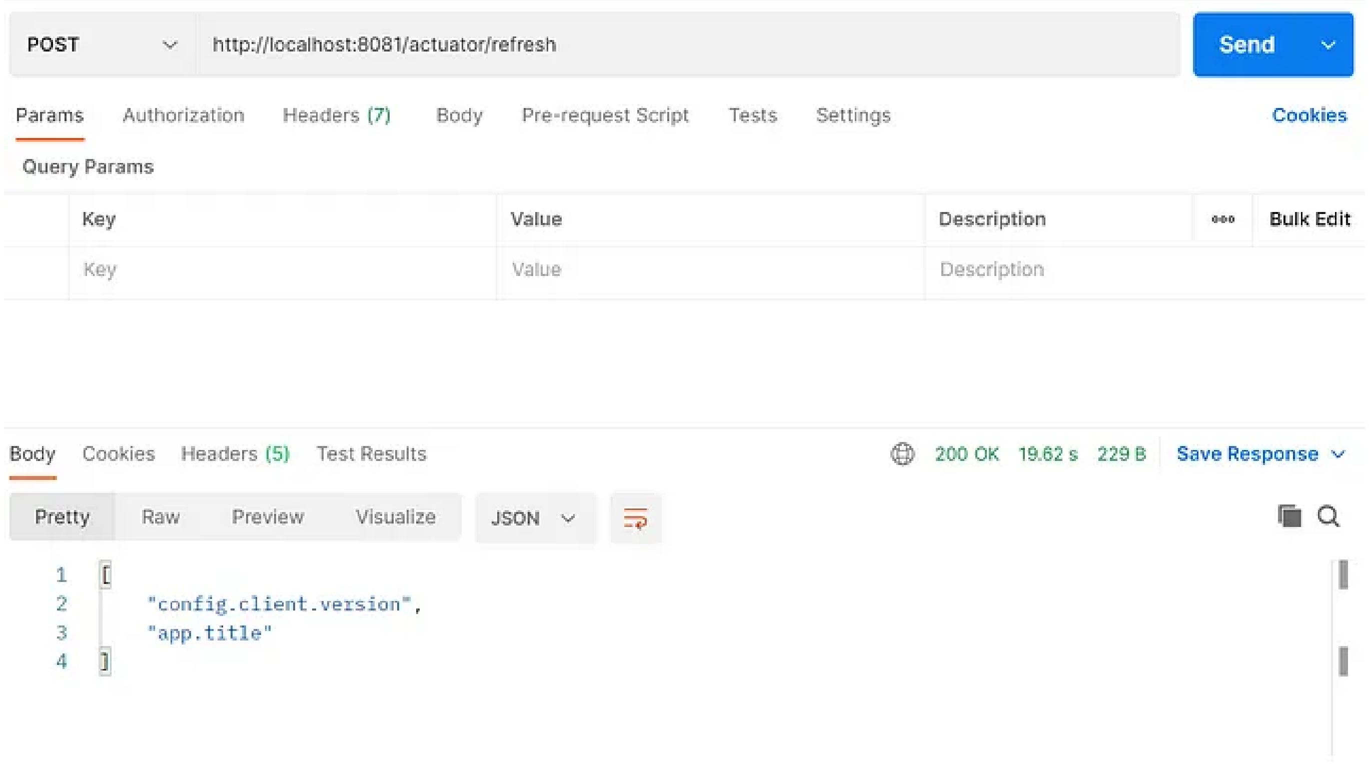
Value of title from Config Server: cloud

After Changing the app.title in application.properties file

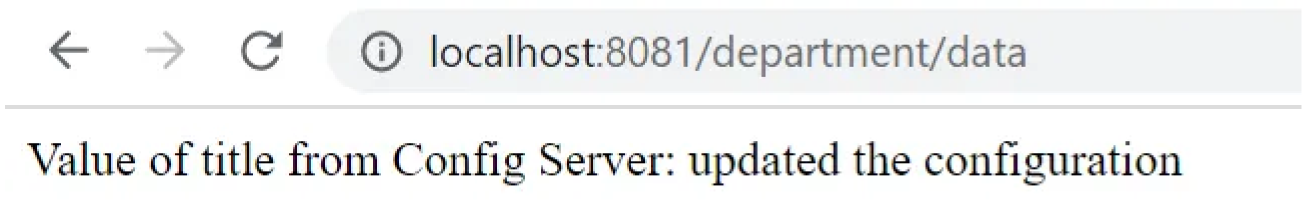


```
app.title=updated the configuration
eureka.client.fetch-registry=true
eureka.client.register-with-eureka=true
eureka.serviceUrl.defaultZone=http://localhost:8761/eureka/

eureka.instance.prefer-ip-address=false
eureka.instance.hostname=localhost
eureka.instance.instance-id=http://localhost:${server.port}
```

Go to Postman tool, select method 'POST', enter URL 'http://localhost:8081/actuator/refresh' and click on send button. You will receive 200 status code with some result as shown in the screen below.



```
[
    "config.client.version",
    "app.title"
]
```



Value of title from Config Server: updated the configuration

Overall, Spring Cloud Config Server is an essential tool for managing configuration in a microservices architecture. It simplifies the management of configuration data and improves the flexibility and security of your microservices.

Spring Cloud Spring Boot Java Programming Software Engineering

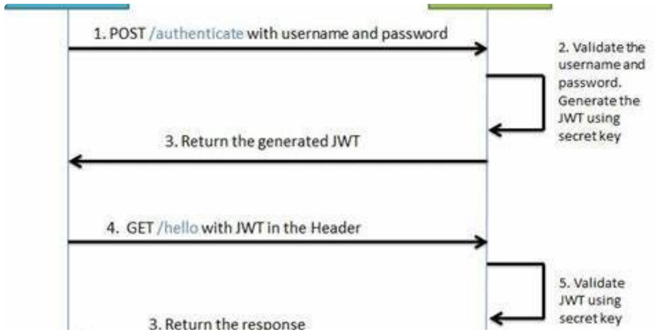
Written by Kishore Karnatakapu

Follow

227 Followers

Software Engineer | Java | Spring Boot| Microservices

More from Kishore Karnatakapu


1. POST /authenticate with username and password
2. Validate the username and password. Generate the JWT using secret key
3. Return the generated JWT
4. GET /hello with JWT in the Header
5. Validate JWT using secret key
3. Return the response

Kishore Karnatakapu in Javarevisited

How to secure Spring Boot with JWT Authentication and…

Hello learners, here we are going to learn about spring security implementation with…

4 min read · Apr 19

96 6

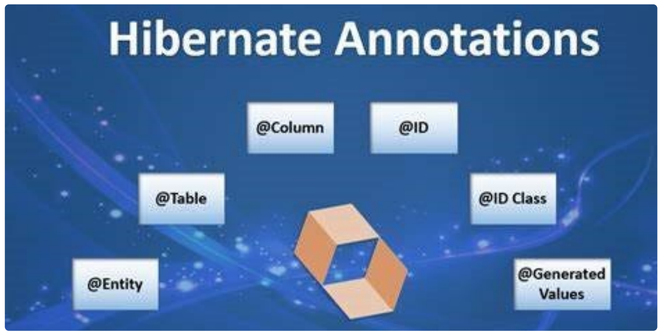
Kishore Karnatakapu in Level Up Coding

How to follow good coding standards in Spring Boot

Spring Boot is a widely used and very popular enterprise-level high-performance…

7 min read · Jun 12

187 2



How to Make Concurrent REST API'S
How to call Spring REST APIs concurrently using Java CompletableFuture

Kishore Karnatakapu in Javarevisited

Kishore Karnatakapu in Level Up Coding

10 Most Commonly used Hibernate Annotations

Hibernate is an open-source Object-Relational Mapping (ORM) framework for Ja...

7 min read · May 5

135

How to call Spring Boot Rest API's Concurrently

How to call Spring REST APIs concurrently using Java CompletableFuture.

5 min read · May 11

102

See all from Kishore Karnatakapu

Recommended from Medium

Chathura Hansika

Key Features and Benefits of Spring Boot Quartz

From periodic job execution to intricate workflow management, task scheduling play...

2 min read · Jun 9

2

Hakan Yılmaz

Entity Auditing in Spring Boot with Hibernate Envers

What is Hibernate Envers ?

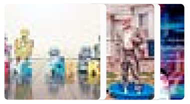
2 min read · Jun 11

5

Lists

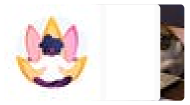
General Coding Knowledge

20 stories · 4 saves



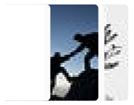
It's never too late or early to start something

10 stories · 4 saves



Stories to Help You Grow as a Software Developer

19 stories · 138 saves



Leadership

31 stories · 63 saves

Moiz Husain Bohra

Java 8 Stream API Interview Questions

The Stream API is used for dealing with object collections. A stream is a collection of items…

11 min read · Jun 9

17

Kishore Karnatakapu in Javarevisited

How to create custom validations in Spring Boot Application

In Spring Boot, validation refers to the process of checking whether the input data…

4 min read · May 1

245 2

Sam Cao

Kunal Nalawade in Level Up Coding

Spring Boot 2.7 to 3.1 — The Migration Story

Spring Boot is a popular framework for building Java-based web applications. It...

3 min read · Jun 6

12

5 Design Patterns in Java that Solve Major Problems!

In a previous post, I showed you how the OOP world looks like. OOP is a pretty convenient...

11 min read · Jan 19

301 6

See more recommendations