**How To Implement Spring Cloud Config Server In Microservices**

In a Microservices architecture based Java application, we have multiple microservices in the form of multiple Spring Boot applications. Each Spring Boot application will have its separate configuration file, where we specify the values in the form of key – value pairs. If you used maven in your application, you must have guessed that we are talking about nothing but application.properties file.

You might have noticed that some of the entries in each application’s application.properties file are common, such as registering with Eureka server, Email, Security, JPA configurations etc. If we can keep these common entries in one central place and make them accessible by each application, it will make our development process easier. Therefore, How can we make a central repository to accommodate all these common entries is the topic of our discussion in this article. Our topic is ‘How To Implement Spring Cloud Config Server In Microservices’.

Generally, we create a space to accommodate our common configuration file in one of the most common repositories, such as Github, Gitlab, Bitbucket etc., provided by Git vendor. The location of this space is mapped into a central server called Config Server provided by Spring Cloud. Each microservice who will use the common configuration entries will have the Config Server location and act as a Config Client for that Config Server. We will discuss about all these basics in detail in this article before trying an example. Let’s start discussing ‘How To Implement Spring Cloud Config Server In Microservices’ and its related concepts.

**Software/Technologies Used in the Example**

Sometimes some version conflicts with other version. Hence, listing down the combinations that are proven to be working with each other. Below is the proven combination of software that are used to develop these examples.

1) GitHub Repository  
2) JDK 1.8 or later (Tested on JDK 11)  
3) Spring Boot 2.6.6  
4) Spring Framework 5.3.18  
5) Spring Cloud 2021.0.1  
6) Spring Cloud Config Server 3.1.1  
7) Maven 4.0.0  
8) IDE – STS 4.7.1.RELEASE  
9) Postman v9.0.9

**What is Spring Cloud Config Server in Microservices?**

Spring Cloud Config is a starter project to manage common configurations. It is integrated with Spring Boot and provided by Spring Cloud. When we create a Spring Boot Project and include the dependency of Spring Cloud Config in order to deal with configuration management, it acts as a dedicated Spring Cloud Config Server.

In other words, it acts as a central configuration server that provides configurations (properties) to each microservice connected to it. Moreover, it significantly simplifies the management of many microservices by centralizing their configuration in one location. Also, it provides the ability to refresh the configuration of a microservice without redeploying the configuration changes. We apply @EnableConfigServer annotation at the main class of the Application to recognize that this application will act as a Spring Cloud Config Server.

**What is Spring Cloud Config Client?**

Although, Spring Cloud Config Server is a popular term among microservices developers. On the other hand, we also have Spring Cloud Config Client which utilizes the services provided by Spring Cloud Config Server. Technically, these are microservices or Spring Boot Applications that take immediate advantage of the Spring Cloud Config Server. In order to connect with Config Server, we provide the entry of Config Server in the application.properties file of Config Client.

**Why do we need a Spring Cloud Config Server?**

Spring Boot is the most popular framework for developing microservices rapidly and smoothly. Using Spring Boot it’s easy to create even hundreds of microservices for a project. However, managing the configurations of all these services sometimes becomes a headache. As each service has its own configuration, it works smoothly for small, monolithic applications. But when we deal with hundreds of services and potentially thousands of configurations in microservices based applications, managing all of them can be a concern. Here Spring Cloud Config Server comes into the picture and makes a developer’s life easy by offering a common place for all common configurations.

**What are the Advantages of Using Spring Cloud Config Server in Microservices?**

1) It promotes code reusability and removes code repetition. By using Spring Cloud Config Server, we have a centralized configuration. Therefore, we don’t need to write the same set of properties in multiple microservices. Instead, write them at one place and access from any microservice.

2) On the other hand, we don’t need to redeploy applications in case we modify any configuration.

**What are the different ways to implement Config Server In Microservices?**

Generally, there are two ways to implement Config Server in Microservices.

1) External Config Server

2) Native Config Server

**External Config Server**

In this approach, we use remote repository to accommodate our common config file such as Github, Gitlab, Bitbucket etc. This approach is used in production environment.

**Native Config Server**

In Native Config Server or Internal Config Server, we use local system to accommodate our common config file such as installing Git in a local system & connecting it with Config Server. This approach is generally used in development or testing environment.

Furthermore, steps for implementing both of them are similar, except the location entry in the Config Server. Here, we will discuss about the External Config Server approach which can be useful in a production environment.

**How To Implement Spring Cloud Config Server In Microservices?**

Till now, we are clear with What is Config Server?, Why do we need this? and How does it work?. Now, it’s time to dig into our article header ‘How To Implement Spring Cloud Config Server In Microservices’ with an example and be more confident on implementing it by applying a step by process.

**Use Case Details**

In order to implement Config Server on a live project, let’s consider a use case where we will have a minimum of three microservices. First microservice will be Eureka Server to register other microservices. Second microservice will be Config Server just like a mediator between a third party repository(Github, Gitlab Bitbucket etc.) and other microservices in the application. Furthermore, third microservice will be our general purpose service who will use the configuration values at run time. It will also act as a Config Client for aforesaid Config Server.

**Create a Repository in GitHub**

If you don’t have your account in GitHub, create it to accommodate your configuration file and access it remotely with the help of Config Server. You can use any remote repository like GitHub, GitLab, Bitbucket etc. Here in this example, we will use GitHub repository. In order to create a new repository in GitHub, follow below steps.

1) Register and Login to [GitHub](https://github.com/" \t "_blank)

2) Create a new repository e.g. config\_repo

3) Create one file name e.g. application.properties. Enter some property in the form of key-value pair. For Example:  my.app.title=Sample App. Finally, commit the changes. Please note that we can create application.yml file also.

4) Copy URL: https://github.com/username/config\_repo.git, username and password. We will require URL, username & password when we create the config server at the later stage in this article.

**Create Microservice #1(Eureka Server)**

First of all, we will create a Eureka Server Service in order to discover and communicate Microservices with each other. Please note that Eureka server itself acts as a Microservice. Moreover, it is nothing but just a Spring Boot Project that incorporates Spring Cloud’s Eureka Server dependency. In order to recognize that this is a Eureka Server, our main class will have @EnableEurekaServer annotation. Additionally, we will have some specific properties in application.properties file that will clearly indicate that this application/microservice is a Eureka Server. In order to create Eureka Server follow the below steps.

**Step #1: Create a Spring Boot Project**

Here, we will use STS(Spring Tool Suite) to create our Spring Boot Project. If you are new to Spring Boot, visit [Internal Link](https://javatechonline.com/saving-data-into-database-using-spring-boot-data-jpa-step-by-step-tutorial/#Step_1_Creating_Starter_Project_using_STS) to create a sample project in spring boot. While creating a project in STS, add starter ‘Eureka Server’ in order to get features of it.

**Step #2: Apply Annotation @EnableEurekaServer at the main class**

In order to make your application/microservice acts as Eureka server, you need to apply @EnableEurekaServer at the main class of your application.

**import org.springframework.boot.SpringApplication;**

**import org.springframework.boot.autoconfigure.SpringBootApplication;**

**import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;**

**@SpringBootApplication**

**@EnableEurekaServer**

**public class SpringCloudEurekaServerApplication {**

**public static void main(String[] args) {**

**SpringApplication.run(SpringCloudEurekaServerApplication.class, args);**

**}**

**}**

**Step #3: Modify application.properties file**

Add below properties in your application.properties file.

**server.port=8761**

**eureka.client.register-with-eureka=false**

**eureka.client.fetch-registry=false**

**Create Microservice #2(Config Server)**

This is one of the important microservice of our article i.e. Config Server. We will provide dependency of Config Server as a starter project in our service. Let’s develop it step by step.

**Step #1: Create a Spring Boot Project**

We are using STS(Spring Tool Suite) to create our Spring Boot Project. While creating a project in STS, add starter ‘Config Server’ in order to get all required features of it.

**Step #2: Apply Annotation @EnableConfigServer at the main class**

**import org.springframework.boot.SpringApplication;**

**import org.springframework.boot.autoconfigure.SpringBootApplication;**

**import org.springframework.cloud.netflix.eureka.EnableEurekaClient;**

**@SpringBootApplication**

**@EnableConfigServer**

**public class SpringCloudConfigServerApplication {**

**public static void main(String[] args) {**

**SpringApplication.run(SpringCloudFeignBookServiceApplication.class, args);**

**}**

**}**

**Step #3: Modify application.properties file**

Add below properties in your application.properties file.

# Server port

**server.port=8888**

# Repository Location in Github

**spring.cloud.config.server.git.uri=https://github.com/username/config\_repo.git**

# Github username

**spring.cloud.config.server.git.username=yourUserName**

# Github Password

**spring.cloud.config.server.git.password=yourPassword**

# Github default branch

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

**Create Microservice #3(Microservice as a Config Client)**

This service will act as a Config Client for the aforementioned Config Server. We will include config client as a dependent starter project in order to use the features of a Config Client. Let’s develop it step by step.

**Step #1: Create a Spring Boot Project**

As we are using STS(Spring Tool Suite) to create our Spring Boot Project. While creating a project in STS, add starter ‘Config Client’, ‘Spring web’, and ‘Eureka Discovery Client’ in order to get all required features of it.

**Step #2: Apply Annotation @EnableEurekaClient at the main class**

**import org.springframework.boot.SpringApplication;**

**import org.springframework.boot.autoconfigure.SpringBootApplication;**

**import org.springframework.cloud.netflix.eureka.EnableEurekaClient;**

**@SpringBootApplication**

**@EnableEurekaClient**

**public class SpringCloudProductServiceApplication {**

**public static void main(String[] args) {**

**SpringApplication.run(SpringCloudFeignBookServiceApplication.class, args);**

**}**

**}**

**Step #3: Modify application.properties file**

Add below properties in your application.properties file.

# port

**server.port=9940**

# serviceId (application-name)

**spring.application.name=PRODUCT-SERVICE**

# Eureka Location

**eureka.client.service-url.defaultZone=http://localhost:8761/eureka**

#Config Server location

**spring.config.import=optional:configserver:http://localhost:8888**

Note: In real time scenario, localhost will be replaced by actual server name or ip address as applicable.

**Step #4: Create a RestContoller class as ProductRestController.java**

In order to test if our Product Service is getting values from the remore repository or not, we will create a ProductRestController and define the endpoints as below.

**import org.springframework.beans.factory.annotation.Value;**

**import org.springframework.http.ResponseEntity;**

**import org.springframework.web.bind.annotation.GetMapping;**

**import org.springframework.web.bind.annotation.RequestMapping;**

**import org.springframework.web.bind.annotation.RestController;**

**@RestController**

**@RequestMapping("/product")**

**public class ProductRestController {**

**@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*);**

**}**

**}**

**How to test Config Server Enabled Application?**

It’s time to test our implemented concept of Config Server. Please follow below steps:

1) Start Service/Application containing Eureka Server

2) Start Application containing Config Server

3) Start Application containing Product Service

4) Open a browser window, hit the below URL and observe the results.

**http://localhost:9940/product/data**

If you are getting configuration properties values from Github repository, it means you are getting correct values and your implementation is also correct.

**What if same key is present at microservice configuration file and Config Server with different values?**

Sometime you may face a situation that by mistake you have some configuration properties with the same keys in both the places: in your microservices’ application.properties file and in Github repository which is connected to your Config server. In that case, Config Server takes the higher priority.

**What is Spring Boot Actuator?**

In a nutshell, Spring Boot Actuator is a starter project(sub-project) of the Spring Boot Framework. We also use this to monitor & get health checks of a production-ready application. Moreover, it offers various services as part of monitoring such as understanding the traffic, knowing the state of the database, knowing the application environment, gathering of metrices and many more. It uses HTTP endpoints to expose these operational information about any running application.

In order to enable Spring Boot actuator endpoints to our Spring Boot application, we need to add the Spring Boot Starter actuator dependency in our build configuration file i,e pom.xml(Maven Users). If you are using STS as an IDE to develop your project, you can select ‘Spring Boot Actuator’ as a starter dependency while creating your project. Even you can go to Edit Starters section of STS and add it if you have not added it while creating the project. Below is the dependency for pom.xml.

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-actuator</artifactId>**

**</dependency>**

**What is RefreshScope?**

Spring Cloud Configuration Server lets developers to load the new configuration properties without restarting the application and without any downtime.

RefreshScope is a class under Spring Cloud that allows for beans to be refreshed dynamically at runtime (see [refresh(String)](https://www.javadoc.io/static/org.springframework.cloud/spring-cloud-commons-parent/1.1.4.RELEASE/org/springframework/cloud/context/scope/refresh/RefreshScope.html#refresh-java.lang.String-) and [refreshAll()](https://www.javadoc.io/static/org.springframework.cloud/spring-cloud-commons-parent/1.1.4.RELEASE/org/springframework/cloud/context/scope/refresh/RefreshScope.html" \l "refreshAll--" \t "_blank)). If a bean is refreshed then the next time the bean is accessed (i.e. a method is executed) a new instance is created. Spring Cloud offers **@RefreshScope** annotation to implement it in the project. It is used to load the configuration properties value from the Config server.

If we talk about implementation, it is nothing but an endpoint(/refresh) of the Spring Boot Actuator which helps to load the configuration properties value from the Config server. It is provided by Spring Cloud. Additionally, we need to add the @RefreshScope annotation to our main class of Spring Boot application.

**How can we get updated values from Config Server in microservices without restarting the application?**

If you are going to appear in an interview where microservices knowledge is expected, this question may arise. Furthermore, you need to understand what the interviewer wants you to explain. Here, the interviewer wants to check if you have knowledge of refresh scope or not.

Let’s understand it programmatically how we can achieve this by following step by step process as below:

**Step#1: Add Spring Boot Actuator dependency in your service (Spring Boot Application)**

In our case, we will use it in our Product Service. We are using STS(Spring Tool Suite) to develop our project.

Go to Spring Boot Project → Right Click → Spring → Add Starters → search for ‘Actuator’ → Select ‘Spring Boot Actuator’ → Click on ‘OK’

**Step#2: Update application.properties of your Application**

Add below entry in your application.properties

  #Activate Spring Boot Actuator

**#management.endpoints.web.exposure.include=refresh**

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

Sometimes refresh doesn’t work, hence include ‘\*’ for safer side. If it works, no need to include ‘\*’.

**Step#3: Add @RefreshScope at the RestController of your microservice**

In our case, we will add the @RefresgScope at ProductRestController class as below.

**import org.springframework.beans.factory.annotation.Value;**

**import org.springframework.http.ResponseEntity;**

**import org.springframework.web.bind.annotation.GetMapping;**

**import org.springframework.web.bind.annotation.RequestMapping;**

**import org.springframework.web.bind.annotation.RestController;**

**@RestController**

**@RequestMapping("/product")**

**@RefreshScope**

**public class ProductRestController {**

**@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*);**

**}**

**}**

**How to test Config Server with Refresh Scope Enabled Application?**

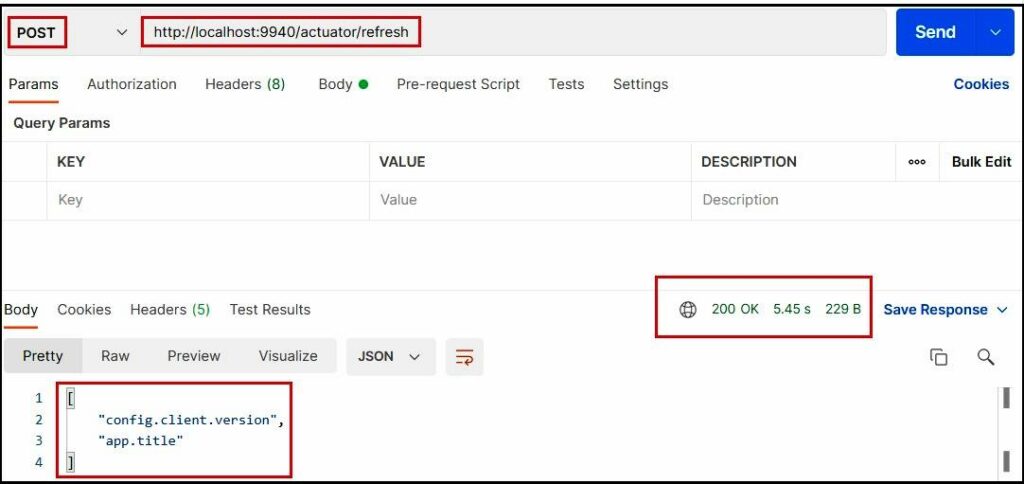
In order to test Refresh Scope, we will use POSTMAN tool. Let’s follow below process step by step.

**Step#1:** Open Postman tool

**Step#2:** Change the value of property in GitHub and commit the changes. In our case its ‘app.title’.

**Step#3:** In the Postman tool, select method as ‘POST’, enter URL ‘http://localhost:9940/actuator/refresh‘ and click on send button. You will receive 200 status code with some result as shown in the screen below.

**Step#4:** Open your browser and hit the actual URL(http://localhost:9940/product/data). If the browser with actual URL id already opened, just refresh the browser. You will see the updated value.



**Also read:**Other [**Implementations & Tutorials on Java Microservices**](https://javatechonline.com/microservices-tutorial/)

**Conclusion**

After going through all the theoretical & example part of ‘How To Implement Spring Cloud Config Server In Microservices’, finally, we should be able to implement Spring Cloud Config Server in Microservices. Similarly, we expect from you to further extend these examples and implement them in your project accordingly. In addition, If there is any update in the future, we will also update the article accordingly. Moreover, Feel free to provide your comments in the comments section below.