Spring Boot Microservices - Spring Cloud Config Server

author: Ramesh Fadatare

MICROSERVICES SPRING BOOT SPRING CLOUD

In this previous couple of tutorials, we have seen:

Spring Boot Microservices Communication Example using RestTemplate

Spring Boot Microservices Communication Example using WebClient

Spring Boot Microservices Communication Example using Spring Cloud Open Feign

Spring Boot Microservices - Spring Cloud Netflix Eureka-based Service Registry

In this tutorial, we will learn how to create a Spring cloud config server to centralize configurations of the Spring boot microservices.

YouTube Video

Problem and Solution

Problem

In the microservices project, there could be a large number of microservices and multiple instances of those microservices are running. Updating configuration properties and restarting all those instances manually or even with automated scripts may not be feasible.

Solution

Spring Cloud Config addresses this problem.

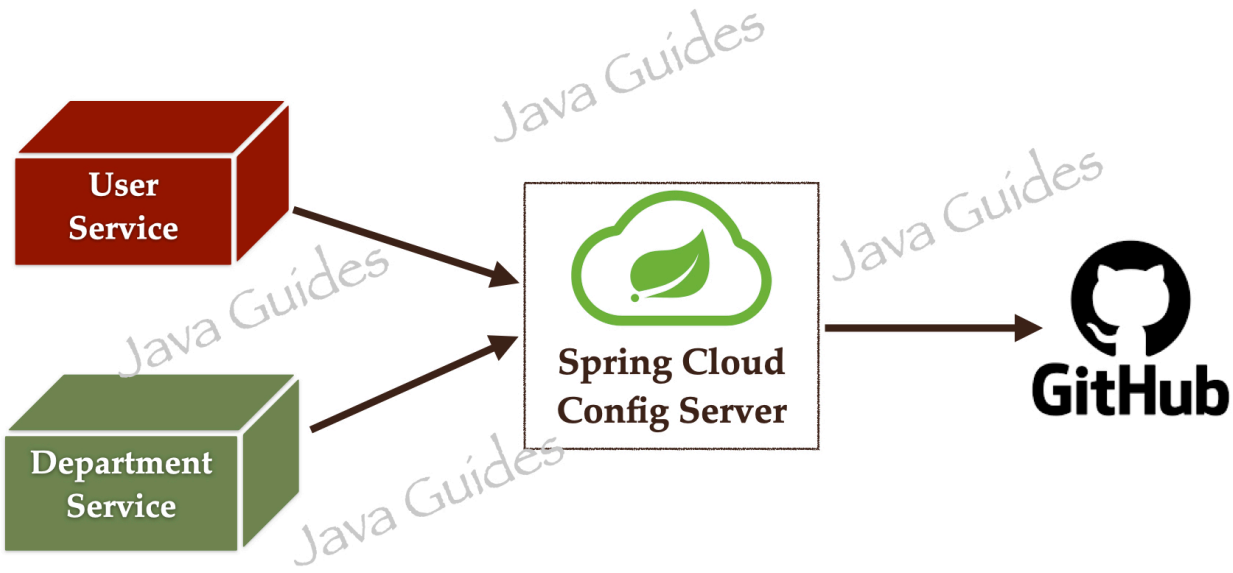
We can create a Spring Cloud Config Server which provides the configuration values for all of our microservices. We can use git, svn, database, or Consul as a backend to store the configuration parameters.

Next, we can configure the location of the Spring Cloud Config server in our microservice so that it will load all the properties when we start the application. In addition to that, whenever we update the properties we can invoke the `actuator/refresh` the REST endpoint in our microservice so that it will reload the configuration changes without requiring us to restart the application.

What we will build?

Let's create Spring Cloud Config Server using Git as a backend to store the configurations. Spring Cloud Config Server is nothing but a SpringBoot project.

Spring Cloud Config Server



Prerequisites

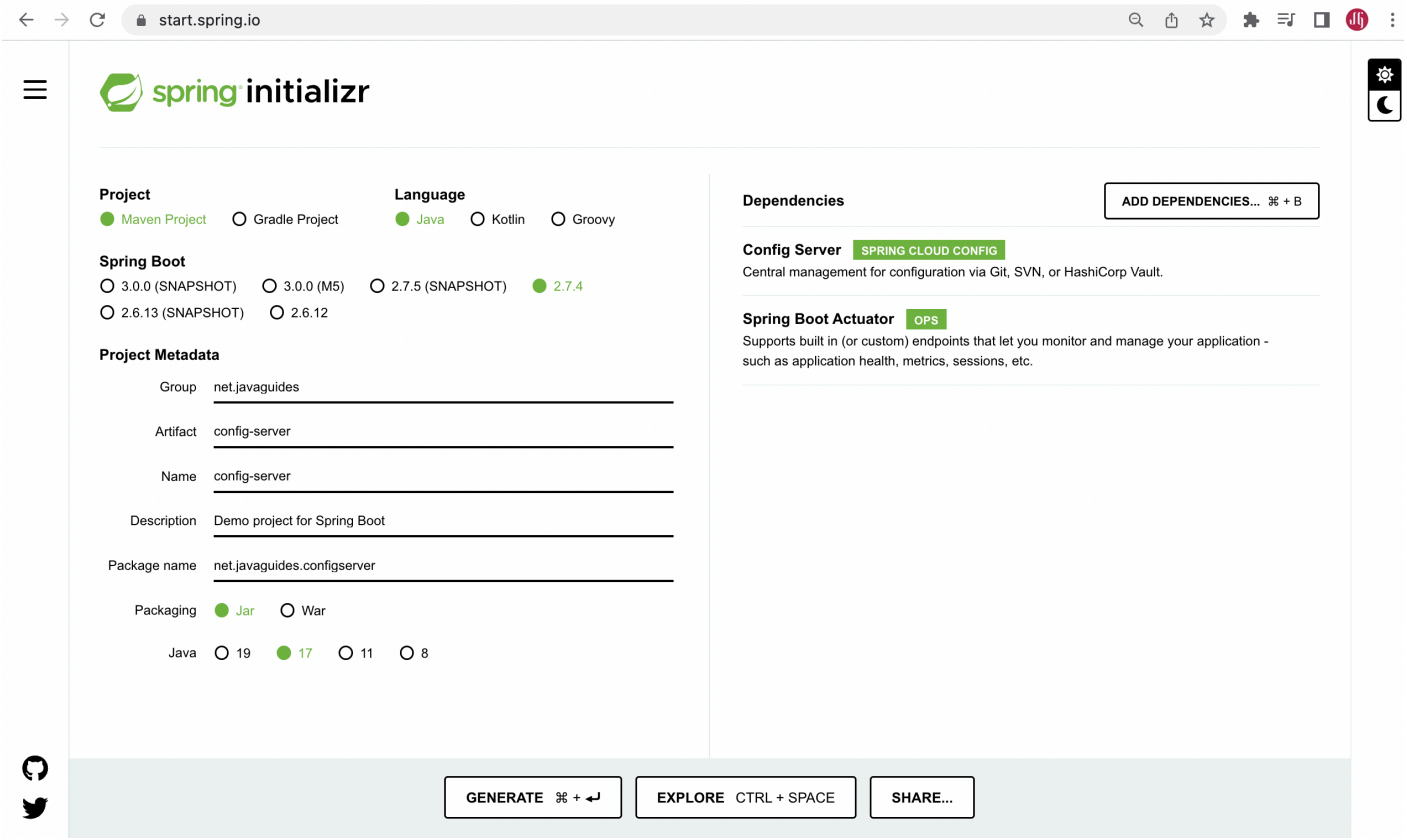
Refer to the below tutorial to create `department-service` and `user-service` microservices:

Spring Boot Microservices Communication Example using RestTemplate.

1. Create and Setup Spring Boot Project in IntelliJ IDEA

Let's create a Spring boot project using the **spring initializr**.

Refer to the below screenshot to enter details while creating the spring boot application using the **spring initializr**:



Click on Generate button to download the Spring boot project as a zip file. Unzip the zip file and import the Spring boot project in IntelliJ IDEA.

Here is the `pom.xml` file for your reference:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>2.7.4</version>
                <relativePath/> <!-- lookup parent from repository -->
        </parent>
        <groupId>net.javaguides</groupId>
        <artifactId>config-server</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <name>config-server</name>
        <description>Demo project for Spring Boot</description>
        <properties>
                <java.version>17</java.version>
                <spring-cloud.version>2021.0.4</spring-cloud.version>
        </properties>
        <dependencies>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-actuator</artifactId>
                </dependency>
                <dependency>
                        <groupId>org.springframework.cloud</groupId>
                        <artifactId>spring-cloud-config-server</artifactId>
                </dependency>

                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-test</artifactId>
                        <scope>test</scope>
                </dependency>
        </dependencies>
        <dependencyManagement>
```

```
		<dependencies>
			<dependency>
				<groupId>org.springframework.cloud</groupId>
				<artifactId>spring-cloud-dependencies</artifactId>
				<version>${spring-cloud.version}</version>
				<type>pom</type>
				<scope>import</scope>
			</dependency>
		</dependencies>
	</dependencyManagement>

	<build>
		<plugins>
			<plugin>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-maven-plugin</artifactId>
			</plugin>
		</plugins>
	</build>

</project>
```

2. Enable Config Server using @EnableConfigServer Annotation

To make our Spring Boot application as a Spring Cloud Config Server, we just need to add the `@EnableConfigServer` annotation to the main entry point class:

```
package net.javaguides.configserver;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;
import org.springframework.web.bind.annotation.GetMapping;

@EnableConfigServer
@SpringBootApplication
public class ConfigServerApplication {

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

}
```

3. Configure the Location of the Git repository

On Github, create a Git repository named "microservices-config-repo".

Now, let's configure the location of the git repository where we are going to store all our configuration files in the `application.properties` file.

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

spring.cloud.config.server.git.uri=https://github.com/RameshMF/microservices-config-repo.git
spring.cloud.config.server.git.skipSslValidation=true
spring.cloud.config.server.git.clone-on-start=true
management.endpoints.web.exposure.include=*
```

That's it. This is all you need to do to create Spring Cloud Config Server and you just need to add application-specific config files in the git repository.

You can refer to my GitHub repository: **https://github.com/RameshMF/microservices-config-repo**

4. Refactor department-service to use Config Server

Our `department-service` will become a client for Config Server. So, let us add Config Client starter dependency to the `department-service` :

In `department-service` , add the below dependencies to pom.xml:

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

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

5. Push department-service.properties File to the GitHub

Now we need to add all the properties of our `department-service` in `department-service.properties` and commit/push it to our git repo **microservices-config-repo**.

Create a new file `department-service.properties` on the GitHub repository, add the below content and commit it:

```
spring.datasource.url=jdbc:mysql://localhost:3306/department_db
spring.datasource.username=root
spring.datasource.password=Mysql@123

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.ddl-auto=update
```

6. Configure Config Server in department-service

Next, change `resources/application.properties` with the config server

```
spring.application.name=department-service
spring.config.import=optional:configserver:http://localhost:8888
management.endpoints.web.exposure.include=*
```

Next, start the Config Server application and then the department-service application. This should work fine. You can check the console logs that department-service is fetching the properties from config server **http://localhost:8888/** at startup.

7. Refactor the user-service to use Config Server

Our `user-service` will become a client for Config Server. So, let us add Config Client starter to `user-service` which will add the following dependency.

In user-service, add below dependencies to pom.xml:

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

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

8. Push user-service.properties File to the GitHub

Now we need to add all the properties of our `user-service` in `user-service.properties` and commit/push it to our git repo **microservices-config-repo**.

Create a new file `user-service.properties` on the GitHub repository, add the below content, and commit it:

```
spring.datasource.url=jdbc:mysql://localhost:3306/employee_db
spring.datasource.username=root
spring.datasource.password=Mysql@123

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.ddl-auto=update

server.port = 8081
```

9. Configure Config Server in user-service

Next, change `resources/application.properties` with the config server

```
spring.application.name=USER-SERVICE
spring.config.import=optional:configserver:http://localhost:8888
management.endpoints.web.exposure.include=*
```

Now first start the Config Server application and then the `user-service` application. This should work fine. You can check the console logs that the `user-service` is fetching the properties from config server **http://localhost:8888/** at startup.

10. Refresh Use case

We also want to enable the `/refresh` endpoint, to demonstrate dynamic configuration changes.

The client can access any value in the Config Server by using traditional mechanisms (such as @ConfigurationProperties or @Value("${... }") or through the Environment abstraction). Now you need to create a Spring MVC REST controller that returns the resolved message property's value.

In department-service, create below REST API:

```
package net.javaguides.departmentservice.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;

@RefreshScope
@RestController
class MessageRestController {

    @Value("${message:Hello default}")
    private String message;

    @GetMapping("/message")
    String getMessage() {
        return this.message;
    }
}
```

By default, the configuration values are read on the client's startup and not again. You can force a bean to refresh its configuration (that is, to pull updated values from the Config Server) by annotating the MessageRestController with the Spring Cloud Config `@RefreshScope` and then triggering a refresh event.

Test Refresh Use Case

You can test the end-to-end result by starting the Config Service first and then, once it is running, starting the client. Visit the client app in the browser at **http://localhost:8080/message**. There, you should see Hello world in the response.

Change the message key in the `department-service.properties` file in the Git repository to something different (Hello, Ramesh!).

You need to invoke the `/refresh` Spring Boot Actuator endpoint in order to force the client to refresh itself and draw in the new value. Spring Boot's Actuator exposes operational endpoints (such as health checks and environment information) about an application.

You can invoke the refresh Actuator endpoint by sending an empty HTTP POST to the client's refresh endpoint: **http://localhost:8080/actuator/refresh**. Then you can confirm it worked by visiting the **http://localhost:8080/message** endpoint.

11. Testing department-service and user-service

First, start the config-server, and next, department-service, and user-service.

If your `department-service` able to connect to the MySQL database then you have successfully configured the Config server.

If your `user-service` able to connect to the MySQL database then you have successfully configured the Config server.

Next, you can test the REST endpoints of `department-service` and `user-service` microservices.

Refer to **Spring Boot Microservices Communication Example using RestTemplate** for testing REST endpoints.

Conclusion

In this tutorial, we learned how to create a Spring cloud config server with Git as a backend to centralize configurations of the Spring boot microservices.