Spring Cloud Config Example Tutorial

author: Ramesh Fadatare

SPRING BOOT SPRING CLOUD

Managing configuration in a microservices architecture can be challenging. Hardcoding configurations or maintaining separate configuration files for each service leads to inconsistencies and potential errors. Enter **Spring Cloud Config**, a powerful tool that provides server-side and client-side support for externalized configuration in a distributed system.

In this blog post, we'll explore how to set up a Spring Cloud Config Server and two client applications using the latest Spring Cloud Config and Spring Boot versions.

What is Spring Cloud Config?

Spring Cloud Config provides a centralized way to manage external properties for applications across all environments. The Config Server serves these configurations, which are typically stored in a version-controlled repository like Git. Client applications fetch their configurations from this central server, ensuring consistency and ease of updates.

Key Benefits:

1. **Centralized Management**: Store all configuration files in a single location.
2. **Version Control**: Use Git for version control, enabling easy rollback to previous configurations.
3. **Dynamic Refresh**: Change configurations without redeploying applications.

Step 1: Setting Up the Config Server

1.1 Create a Spring Boot Application for the Config Server

Using Spring Initializr to create the project:

1. Go to **Spring Initializr**.
2. Set the project metadata:
    - **Group**: `com.example`
    - **Artifact**: `config-server`
    - **Name**: `Config Server`
    - **Description**: `Demo Config Server for Spring Cloud Config`
    - **Package Name**: `com.example.configserver`
    - **Packaging**: `Jar`
    - **Java**: `11` (or higher)
3. Add dependencies:
    - **Config Server**
    - **Spring Boot Actuator**
4. Click on **Generate** to download the project.

5. Unzip the downloaded project and open it in your IDE.

Adding dependencies manually (if needed):

For Maven:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-config-server</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
</dependencies>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>2021.0.6</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

For Gradle:

```
dependencies {
    implementation 'org.springframework.cloud:spring-cloud-config-server'
    implementation 'org.springframework.boot:spring-boot-starter-actuator'
}

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-dependencies:2021.0.6"
    }
}
```

1.2 Enable the Config Server

Add the `@EnableConfigServer` annotation to the main application class. This enables the Spring Cloud Config Server functionality.

```
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);
    }
}
```

1.3 Configure the Application Properties

Create an `application.yml` file in the `src/main/resources` directory and define the necessary properties. This configuration points to a Git repository where the configuration files are stored.

```yaml
server:
  port: 8888

spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/your-username/config-repo
          clone-on-start: true
```

Explanation:

- `server.port`: The port on which the Config Server will run.
- `spring.cloud.config.server.git.uri`: The URL of the Git repository containing the configuration files.
- `spring.cloud.config.server.git.clone-on-start`: Clone the repository when the server starts.

1.4 Create a Configuration Repository

Create a Git repository (e.g., on GitHub) to store your configuration files. Add service-specific configuration files for the Payment Service and Order Service.

Example structure in the Git repository:

```
config-repo/
  payment-service.yml
  order-service.yml
```

payment-service.yml:

```yaml
payment-service:
  message: "Payment Service: Your payment is being processed."
```

order-service.yml:

```yaml
order-service:
  message: "Order Service: Your order is being processed."
```

Explanation:

- `payment-service.yml`: Contains configuration properties specific to the Payment Service.
- `order-service.yml`: Contains configuration properties specific to the Order Service.

1.5 Run the Config Server

Run the Config Server application. Verify it's running by accessing the following URLs in your browser or using a tool like Postman:

- `http://localhost:8888/payment-service/default`
- `http://localhost:8888/order-service/default`

These URLs should return the configurations from the Git repository.

Step 2: Setting Up the Config Clients

2.1 Create Spring Boot Applications for the Config Clients

Using Spring Initializr to create the projects:

1. Go to **Spring Initializr**.
2. Set the project metadata for both services:
    - **Group**: `com.example`
    - **Artifact**: `payment-service` and `order-service`
    - **Name**: `Payment Service` and `Order Service`
    - **Description**: `Demo Payment Service for Spring Cloud Config` and `Demo Order Service for Spring Cloud Config`
    - **Package Name**: `com.example.paymentservice` and `com.example.orderservice`
    - **Packaging**: `Jar`
    - **Java**: `11` (or higher)
3. Add dependencies:
    - **Config Client**
    - **Spring Web**
4. Click on **Generate** to download the projects.
5. Unzip the downloaded projects and open them in your IDE.

Adding dependencies manually (if needed):

For Maven:

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

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>2021.0.6</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

For Gradle:

```
dependencies {
    implementation 'org.springframework.cloud:spring-cloud-starter-config'
    implementation 'org.springframework.boot:spring-boot-starter-web'
}

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-dependencies:2021.0.6"
    }
}
```

2.2 Configure the Application Properties

For both the Payment Service and Order Service, create an `application.yml` file in the `src/main/resources` directory and define the properties to point to the Config Server.

payment-service/src/main/resources/application.yml:

```
spring:
  application:
    name: payment-service

  cloud:
    config:
      uri: http://localhost:8888
      fail-fast: true
```

order-service/src/main/resources/application.yml:

```
spring:
  application:
    name: order-service

  cloud:
    config:
```

```
      uri: http://localhost:8888
      fail-fast: true
```

Explanation:

- `spring.application.name`: The name of the application, which should match the configuration file name in the Git repository.
- `spring.cloud.config.uri`: The URI of the Config Server.
- `spring.cloud.config.fail-fast`: If true, the application will fail to start if it cannot connect to the Config Server.

2.3 Create REST Controllers

Create REST controllers for both services to read and display the configuration properties.

Payment Service:

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class PaymentController {

    @Value("${payment-service.message}")
    private String message;

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

Order Service:

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class OrderController {

    @Value("${order-service.message}")
    private String message;

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

Explanation:

- `@Value("${payment-service.message}")` and `@Value("${order-service.message}")`: Inject the configuration property values from the Config Server.
- `@GetMapping("/message")`: Define an endpoint to return the message.

2.4 Run the Config Clients

Run both the Payment

Service and Order Service applications. Verify they are reading the configuration from the Config Server by accessing the following URLs in your browser or using a tool like Postman:

- Payment Service URL: `http://localhost:8080/message`
- Order Service URL: `http://localhost:8081/message` (Ensure the port is different if running on the same machine)

These URLs should return the respective messages from the configuration files stored in the Git repository.

Conclusion

With Spring Cloud Config, managing configurations for microservices becomes a breeze. You can centralize your configuration, leverage version control, and ensure consistency across your services. This setup not only streamlines the development process but also enhances the maintainability and scalability of your applications.

By following the steps in this blog post, you've set up a Spring Cloud Config Server and two client applications, demonstrating how easy it is to externalize and manage your configurations. Embrace the power of Spring Cloud Config and simplify your microservices architecture today!