ACERCA DE MICHAEL GOOD

Michael es un ingeniero de software ubicado en el área de Washington DC que está interesado en Java, la ciberseguridad y las tecnologías de código abierto. Sigue su blog personal para leer más de Michael.

Introducción al servidor de configuración de Spring Cloud

Publicado por: Michael Good en Enterprise Java 20 de diciembre de 2017

1. Información general

En este tutorial, repasaremos los conceptos básicos del **servidor de configuración** de **Spring Cloud** . Configuraremos un **servidor de configuración** y luego crearemos una **aplicación cliente** que **consumirá la configuración** al inicio y luego actualizará la configuración sin reiniciar. La aplicación que estamos creando es la misma aplicación "Hello World" que se analiza en la Guía de introducción a la configuración centralizada , pero profundizamos en los conceptos de Spring Cloud Config Server en este artículo.

El código fuente completo para el tutorial está en Github .

2. ¿Qué es Spring Cloud Config Server?

Como se indica sucintamente en la documentación , "Spring Cloud Config brinda soporte al servidor y al lado del cliente para la configuración externalizada en un sistema distribuido". La implementación predeterminada del backend de almacenamiento del servidor usa **git** , por lo que es compatible con versiones etiquetadas de entornos de configuración. a muchas herramientas para administrar el contenido.

Spring Cloud Config se **adapta muy bien a las aplicaciones de Spring** porque sus conceptos de cliente y servidor se corresponden con las abstracciones Spring *Environment* y *PropertySource* . Sin embargo, Spring Cloud Config se puede usar con cualquier aplicación que se ejecute en cualquier idioma.

3. Crea un Proyecto de Múltiples Módulos

La aplicación que estamos creando tendrá dos módulos: uno para el servicio de configuración y otro para el cliente de configuración. Debido a esto, necesitamos crear un *pom* padre .

3.1 Padres

En nuestro IDE, creemos un nuevo proyecto. Estoy usando Spring Tool Suite, pero eso es solo una preferencia personal.

En nuestro *pom.xml* , especifiquemos nuestros dos módulos:

```
<?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 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.michaelcgood</groupId>
    <artifactId>com.michaelcgood</artifactId>
    <version>0.0.1</version>
    <packaging>pom</packaging>

    <name>michaelcgood-spring-cloud-config-server</name>
    <description>Intro to Spring Cloud Config Server</description>


    <modules>
        <module>mcg-configuration-client</module>
        <module>mcg-configuration-service</module>
```

3.2 Servicio de configuración

En nuestro IDE, creemos un nuevo módulo Maven para nuestro servicio de configuración e inserte esto en nuestro *pom* :

```
<?xml version="1.0"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd" xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.michaelcgood</groupId>
  <artifactId>mcg-configuration-service</artifactId>
  <version>0.0.1</version>
  <packaging>jar</packaging>
  <name>mcg-configuration-service</name>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.9.RELEASE</version>
        <relativePath /> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <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>Edgware.RELEASE</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>
```

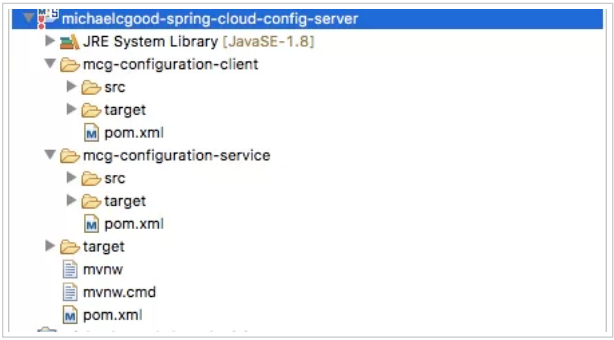
3.3 Cliente de configuración

Ahora solo tenemos que hacer un módulo para nuestro cliente de configuración. Entonces, hagamos otro módulo Maven e inserte esto en nuestro *pom* :

```
<?xml version="1.0"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd" xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.michaelcgood</groupId>
  <artifactId>mcg-configuration-client</artifactId>
  <version>0.0.1</version>
  <packaging>jar</packaging>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.9.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <java.version>1.8</java.version>
    </properties>

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

```
    </dependencies>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Edgware.RELEASE</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>
```

Nuestra estructura de proyecto se ve así ahora:



4. Servidor de configuración

Ahora crearemos un **Servicio de configuración** e para actuar como intermediario entre nuestro cliente y un **repositorio de git** .

4.1 Habilitar servidor de configuración

Usamos *@EnableConfigServer de* Spring Cloud para crear un servidor de configuración con el que se pueda comunicar. Por lo tanto, esto es solo una **aplicación Spring Boot** normal **con una anotación añadida para habilitar el Servidor de configuración** .

```
@EnableConfigServer
@SpringBootApplication
public class ConfigServiceApplication {

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

4.2 application.properties

Para garantizar que no haya conflicto entre los puertos para nuestro servicio de configuración y cliente, especificamos un puerto diferente para el servicio de configuración:

```
server.port=8888

spring.cloud.config.server.git.uri=${HOME}/Desktop/mcg-config
```

La segunda línea, *spring.cloud.config.server.git.uri = $ {HOME} / Desktop / mcg-config,* apunta a un repositorio de git, que crearemos a continuación.

4.3 Git

En un sistema * nix, podemos hacer todo en la línea de comando.
Hacemos una carpeta en nuestro escritorio:

```
mkdir mcg-config
```

Creamos un archivo llamado *a-bootiful-client.properties* usando vim :

```
vim a-bootiful-client.properties
```

Agregamos el mensaje, "Hola mundo", pero esto podría ser lo que nos gustaría. Después de escribir (: w) salimos (: q) vim.

Ahora vamos a crear un nuevo repositorio:

```
git init
```

Comprometámonos:

```
git commit
```

5. Cliente de configuración

Ahora vamos a crear una nueva aplicación Spring Boot que use el Servidor de configuración para cargar su propia configuración y que actualice su configuración para reflejar los cambios en el Servidor de configuración bajo demanda, sin reiniciar la JVM.

Spring verá los archivos de propiedades de configuración de la misma forma que cualquier archivo de propiedades cargado desde *application.properties* , *application.yml* o cualquier otra *PropertySource* .

5.1 Cambios que reflejan

El cliente puede acceder a cualquier valor en el Servidor de configuración utilizando las formas Spring estándar, como *@ConfigurationProperties* o *@Value ("$ {...}")* .

Con esto en mente, creamos un controlador REST que devuelve el valor de la propiedad del mensaje resuelto:

```
@SpringBootApplication
public class ConfigClientApplication {

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

@RefreshScope
@RestController
class MessageRestController {

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

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

La configuración predeterminada solo permite leer los valores en el inicio del cliente y no de nuevo. Entonces, usando *@RefreshScope forzamos* al bean a refrescar su configuración, lo que significa que *extraerá* valores actualizados del Servidor de Configuración y luego activará un evento de actualización.

5.2 bootstrap.properties

Las propiedades para configurar el Cliente de configuración deben leerse antes de que el resto de la configuración de la aplicación se lea desde el Servidor de configuración, durante la fase de arranque.

Especificamos *spring.application.name* del cliente y la ubicación del servidor de configuración *spring.cloud.config.uri* :

```
spring.application.name=a-bootiful-client
spring.cloud.config.uri=http://localhost:8888
management.security.enabled=false
```

Aviso:
desactivamos la seguridad con nuestra configuración *management.security.enabled = false* para facilitar las pruebas y los ajustes.

6. Demo

Primero debemos cambiar el directorio a nuestro servicio de configuración e iniciarlo:

```
mcg-configuration-service mike$  mvn spring-boot:run
```

Y luego haz lo mismo con nuestro cliente:

```
mcg-configuration-client mike$  mvn spring-boot:run
```

Podemos ver en nuestro terminal el servicio de configuración cuando se agrega *a-bootiful-client.properties* :

```
INFO 5921 --- [nio-8888-exec-1] o.s.c.c.s.e.NativeEnvironmentRepository  : Adding property source:
file:/var/folders/dk/4819cm2x3vnfl5ymh6dtxpwc0000gn/T/config-repo-7195892194658362240/a-bootiful-
client.properties
```

Abramos nuestro navegador y visite *http: // localhost: 8080 / message* . Vemos "Hola mundo".

Ahora cambiemos el mensaje en *a-bootiful-client.properties* nuevamente y esta vez pongamos, "¡Hola! :-) ".

Después de guardar y hacer una confirmación, visitamos *http: // localhost: 8888 / a-bootiful-client / default* para confirmar nuestro cambio.

Ahora invocamos el punto final de reinicio de Spring Boot Actuator para actualizar nuestro cliente:

```
curl -X POST http://localhost:8080/refresh
```

7. Conclusión

Acabamos de completar la configuración centralizada de nuestros servicios en Spring. Logramos esto montando un servidor de configuración de Spring Cloud y creando un cliente para consumir la configuración al inicio y luego actualizar la configuración sin reiniciar.

Se pueden hacer muchas otras cosas con Spring Cloud Config Server que no hemos tocado, como por ejemplo:

- Haga que el servidor de configuración se registre con el servicio Discovery para Spring Cloud Netflix, Eureka Service Discovery o Spring Cloud Consul
- Sirve la configuración en formato YAML o Propiedades
- Sirve archivos de configuración de texto plano
- Incrustar el servidor de configuración en una aplicación

El código fuente completo se puede encontrar en Github .

Publicado en Java Code Geeks con el permiso de Michael Good, socio de nuestro programa JCG . Vea el artículo original aquí: Introducción al servidor de configuración de Spring Cloud

Las opiniones expresadas por los colaboradores de Java Code Geeks son las suyas.

Etiquetado con: SPRING CLOUD

Deja una respuesta

¡Sé el primero en comentar!

Notificar de nuevos comentarios de seguimiento Email >

Start the discussion