

Librería compartida Spring-boot

En este documento se abordará como podemos generar una librería compartida entre microservicios spring-boot.

Librerías Compartidas

Para empezar, cabe aclarar que el uso de librerías compartidas entre microservicios es un tema relacionado con la arquitectura de cualquier proyecto, ya sea grande o pequeño. En la arquitectura de microservicios es útil que cada uno de ellos se mantenga lo más aislado e independiente posible, aunque en algunos casos como el que presentamos a continuación es conveniente usar una librería que gestione todo el código transversal al proyecto; aquel que está relacionado con la configuración del mismo, como: auditoría, internacionalización, seguridad, etc. Todo este código no crece de manera exponencial pero si se repite en cada microservicio.

En el ejemplo que trataremos en este documento usaremos proyectos de tipo spring-boot para generar librerías, de esta manera pueden ser usadas como dependencias, esto ocurre gracias a la ayuda de spring-boot que con uno de sus plugins nos ofrecen herramientas para generar un jar que pueda suplir esta función.



Las librerías se deben gestionar por medio de versiones, esto hace que sea fácil su mantenibilidad, sus pruebas y despliegue.

El primer problema enfrentado es intentar encontrar una forma viable de usar una librería compartida entre los diferentes microservicios, habían diferentes opciones que se estudiaron, pero se llegó a que la más conveniente es usar una herramienta propia del framework, que permite generar una librería a partir de un proyecto spring-boot, esto despejaba varias dudas, ya que no empaquetaba todas las dependencias en el jar, sino que la librería era empaquetada únicamente con el código necesario. Usando un plugin de maven, (spring-boot-maven-plugin) se podían generar dos jar, uno que actuaba como un ejecutable y el otro que era la dependencia, luego configurando este plugin se generaba solamente el jar de la dependencia que era lo que se necesitaba para este caso. El siguiente fragmento de código es sacado del archivo *pom.xml* del proyecto librería.

```
1  <build>
2      <plugins>
3          <plugin>
4              <groupId>org.springframework.boot</groupId>
5              <artifactId>spring-boot-maven-plugin</artifactId>
6              <executions>
7                  <execution>
8                      <id>repackage</id>
9                      <configuration>
```

```

10         <skip>true</skip>
11     </configuration>
12 </execution>
13 </executions>
14 </plugin>
15 </plugins>
16 </build>

```

Esta configuración puede cambiar según la versión de spring, pero para la versión usada actualmente esta es la configuración apropiada. Una vez se define esto dentro del proyecto librería, ya se puede empezar a trasladar código transversal a la librería.

Para poder importar la librería solo hace falta declarar la dependencia en el apartado de dependencies, del proyecto principal, de la siguiente forma:

```

1 <dependency>
2   <groupId>com.example</groupId>
3   <artifactId>library</artifactId>
4   <version>0.0.1-SNAPSHOT</version>
5 </dependency>

```



Como las pruebas fueron hechas sobre un repositorio local, maven detecta la dependencia alojada. Hay que revisar a futuro como se va gestionar el manejo de versiones y en que repositorio estará alojada la dependencia.

También es muy importante definir en la clase principal del proyecto (padre), que paquetes van a contener componentes, esto es para que el contenedor de spring los pueda administrar y así poder evitar muchos inconvenientes.

```

@SpringBootApplication(scanBasePackages = "co.edu.uis")
@EntityScan(basePackages = {"co.edu.uis", "com.example.library.model"})
@EnableJpaRepositories(basePackages = {"co.edu.uis", "com.example.library.repository"})
@ComponentScan(basePackages = {"co.edu.uis", "com.example.library.config", "com.example.library.util",
    "com.example.library.service", "com.example.library.exception"})
public class TrainingApplication extends SpringBootServletInitializer {

```

Lo primero que se pasó fue lo relacionado con internacionalización, se trasladó la clase

`CustomLocalResolver` y los `Bean` responsables de configurarla en la clase principal del proyecto a una clase llamada `PrincipalConfiguration` anotada como `@Configuration` en el paquete

config de la librería.

```

103 public MessageSource messageSource() {
102     var messageSource = new ReloadableResourceBundleMessageSource();
101     messageSource.setBasename("classpath:messages");
100     messageSource.setDefaultEncoding("UTF-8");
99     return messageSource;
98 }
97
96 /**
95  * Permite internacionalización y este se lanza para aquellas validaciones
94  * internas de anotaciones tomando el locale adecuado
93  *
92  *
91  */
90 @Bean
89 public LocalValidatorFactoryBean getValidator() {
88     var bean = new LocalValidatorFactoryBean();
87     bean.setValidationMessageSource(messageSource());
86     return bean;
85 }
84
83 /**
82  * Genera un Bean con un LocaleResolver personalizado, éste es encargado de la
81  * configuración regional para el archivo messages_XX.properties que va ser
80  * tenido en cuenta y también para los mensajes por defecto que tiene el
79  * proyecto
78  */
77 @Bean
76 public LocaleResolver localeResolver() { return new CustomLocaleResolver(); }
75

```

Se comprobó el correcto funcionamiento del apartado de internacionalización, y posterior a esto se empezaron a migrar otras cosas transversales como auditoría, excepciones, utilería, etc.

“

Para el tema de auditoría es importante definir en el archivo orm.xml del proyecto principal en donde se encuentra el `EntityHiberanteListener`

```

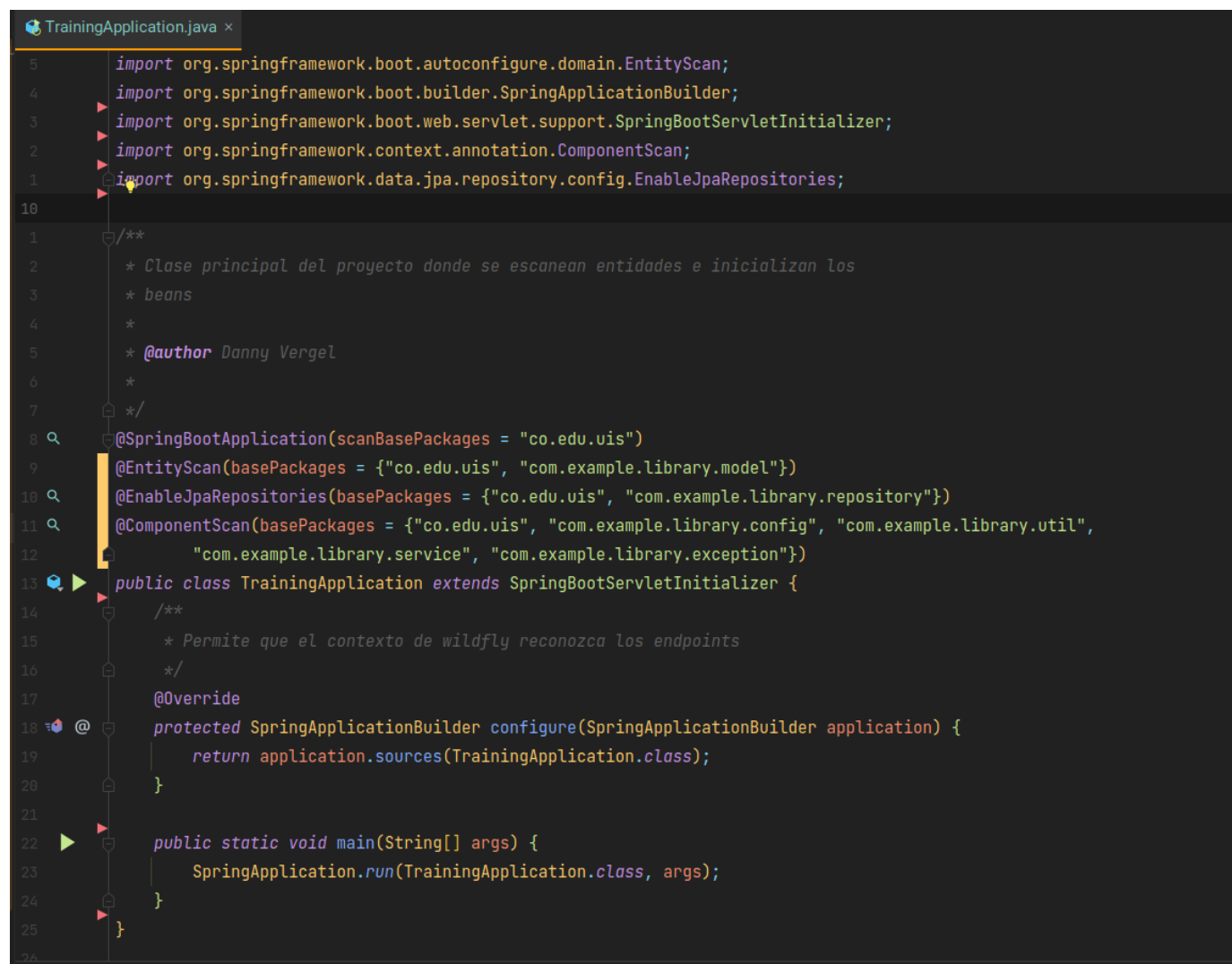
1 <entity-mappings>
2     <persistence-unit-metadata>
3         <persistence-unit-defaults>
4             <entity-listeners>
5                 <entity-listener class="com.example.library.config.EntityHiberanteListener"/>
6             </entity-listeners>
7         </persistence-unit-defaults>
8     </persistence-unit-metadata>
9 </entity-mappings>

```

Se testeó el correcto funcionamiento de auditoría, tanto en servidor tomcat (local) como en servidor wildfly,

también se hizo el traslado de todo el código relacionado con utilería, en este se encontró a priori un pequeño problema, ya que la clase `Mensajes` está anotada como `@Component`, y el contenedor de spring no la detectaba y por lo tanto los métodos de esta clase no estaban funcionando correctamente, a esto se le dio solución poniendo el paquete de util de la librería como un paquete que podría tener componentes (componentScan).

También se hizo el traslado de todos los `Bean` que estaban en la clase principal del proyecto padre, esto hizo que se tuviera una clase principal mucho más limpia y que todo ese código quedara solo en la librería.



```

5      import org.springframework.boot.autoconfigure.domain.EntityScan;
6      import org.springframework.boot.builder.SpringApplicationBuilder;
7      import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;
8      import org.springframework.context.annotation.ComponentScan;
9      import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
10
11  /**
12   * Clase principal del proyecto donde se escanean entidades e inicializan los
13   * beans
14   *
15   * @author Danny Vergel
16   */
17
18  @SpringBootApplication(scanBasePackages = "co.edu.uis")
19  @EntityScan(basePackages = {"co.edu.uis", "com.example.library.model"})
20  @EnableJpaRepositories(basePackages = {"co.edu.uis", "com.example.library.repository"})
21  @ComponentScan(basePackages = {"co.edu.uis", "com.example.library.config", "com.example.library.util",
22    "com.example.library.service", "com.example.library.exception"})
23  public class TrainingApplication extends SpringBootServletInitializer {
24
25      /**
26       * Permite que el contexto de wildfly reconozca los endpoints
27       */
28
29      @Override
30      protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
31          return application.sources(TrainingApplication.class);
32      }
33
34      public static void main(String[] args) {
35          SpringApplication.run(TrainingApplication.class, args);
36      }
37  }

```

“

Algo importante que aclarar, es que donde se use el `@Value` el va tomar estas variables del proyecto principal y no de la librería.

De esta manera todos estos Bean de configuración transversal quedaron en una clase de la librería llamada `PrincipalConfiguration` en el paquete config.

```

library > src > main > java > com > example > library > config > PrincipalConfiguration > messageSource >
Project
> .idea
> .mvn
> src
  > main
    > java
      > com.example.library
        > config
          CustomLocaleResolver
          EntityHibernateListener
          Evento
          PrincipalConfiguration
          RequestInterceptor
          TransactionalListener
        > dto.general
        > exception
        > model
        > repository
        > service.seguridad
        > util
          HolaMundo
          LibraryApplication
        > resources
          application.properties
        > test
        > target
        .gitignore
        HELP.md
        library.iml
        mvnw
        mvnw.cmd
Structure
Sources
20
21 @Configuration
22 public class PrincipalConfiguration {
23     private static IControlAccesoRepository controlAccesoRepository;
24
25     @Value("*.")
26     String clientURL;
27
28     @Value("${JWT_SECRET:rsi}")
29     String jwtSecret;
30
31     @Value("${ENABLE_PREFLIGHT:false}")
32     boolean enablePreflight;
33
34     private static final String HEADER_TOKEN = "Authorization";
35     private static final String ID_PRINCIPAL_TOKEN = "ID";
36     private static final String DEFAULT_SECRET = "rsi";
37     private static final String HEADER_USUARIO = "idUsuario";
38     private static final String PREFIX = "Bearer ";
39     private static final String MESSAGE_ERROR = "Forbidden";
40     private static final String API_URI_PUBLIC_PATTERN = "/public/api/**";
41
42     /**
43      * Permite modificar la propiedades del archivo de mensajes.
44      */
45     @Bean
46     public MessageSource messageSource() {
47         var messageSource = new ReloadableResourceBundleMessageSource();
48         messageSource.setBasename("classpath:messages");
49         messageSource.setDefaultEncoding("UTF-8");
50     }
51

```

También se testeó el manejo de la excepciones, la inserción de logs en base de datos por parte del [RSIControllerException](#), etc.

“

Se puede concluir que esta es una manera bastante viable de abordar el problema de código exponencial compartido entre los diferentes microservicios, y así tener un control de todo esta parte de configuración.