(http://baeldung.com)

Introducción a Spring Method Security

Última modificación: 20 de enero de 2018

por baeldung (http://www.baeldung.com/author/baeldung/)
(http://www.baeldung.com/author/baeldung/)

Seguridad (http://www.baeldung.com/category/security-2/)

Primavera (http://www.baeldung.com/category/spring/) +

1. Introducción

En pocas palabras, Spring Security admite la semántica de autorización en el nivel de método.

Por lo general, podemos asegurar nuestra capa de servicio, por ejemplo, restringiendo qué roles son capaces de ejecutar un método en particular, y probándolo usando soporte de prueba de seguridad a nivel de método.

En este artículo, vamos a revisar el uso de algunas anotaciones de seguridad primero. Luego, nos enfocaremos en probar la seguridad de nuestro método con diferentes estrategias.

2. Habilitación de la seguridad del método

En primer lugar, para usar Spring Method Security, necesitamos agregar la dependencia *spring-security-config* :

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

Podemos encontrar su última versión en Maven Central (https://search.maven.org/#search%7Cga%7C1%7C%22spring-security-config%22) .

Si queremos usar Spring Boot, podemos usar la dependencia *spring-boot-starter-security* que incluye *spring-security-config* :

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

Nuevamente, la última versión se puede encontrar en Maven Central (https://search.maven.org/#search%7Cga%7C1%7C%22spring-boot-starter-security%22) .

A continuación, debemos habilitar la Seguridad de método global:

```
@Configuration
@EnableGlobalMethodSecurity(
  prePostEnabled = true,
  securedEnabled = true,
  jsr250Enabled = true)
public class MethodSecurityConfig
  extends GlobalMethodSecurityConfiguration {
}
```

- La propiedad *prePostEnabled* habilita las anotaciones previas / posteriores de Spring Security
- La propiedad *securedEnabled* determina si la anotación *@Secured* debe estar habilitada
- La propiedad *jsr250Enabled* nos permite usar la anotación *@RoleAllowed*

Exploraremos más sobre estas anotaciones en la próxima sección.

3. Aplicación de la seguridad del método

3.1. Usando *@Acured* Annotation

La anotación *@Secured* se usa para especificar una lista de roles en un método. Por lo tanto, un usuario solo puede acceder a ese método si tiene al menos una de las funciones especificadas.

Vamos a definir un método *getUsername* :

```
@Secured("ROLE_VIEWER")
public String getUsername() {
    SecurityContext securityContext = SecurityContextHolder.getContext();
    return securityContext.getAuthentication().getName();
}
```

Aquí, la *anotación @Secured ("ROLE_VIEWER")* define que solo los usuarios que tienen el rol *ROLE_VIEWER* pueden ejecutar el método *getUsername* .

Además, podemos definir una lista de roles en una anotación *@Secured* :

```
@Secured({ "ROLE_VIEWER", "ROLE_EDITOR" })
public boolean isValidUsername(String username) {
    return userRoleRepository.isValidUsername(username);
}
```

En este caso, la configuración establece que si un usuario tiene *ROLE_VIEWER* o *ROLE_EDITOR* , ese usuario puede invocar el método *isValidUsername* .

La anotación *@Secured* no es compatible con Spring Expression Language (SpEL).

3.2. Usando *@RoleAllowed* Annotation

El *@RoleAllowed* anotación es equivalente a la anotación de JSR-250 de la *@Secured* anotación .

Básicamente, podemos usar la anotación *@RoleAllowed* de forma similar a *@Secured* . Por lo tanto, podríamos volver a definir los métodos *getUsername* y *isValidUsername* :

```
@RolesAllowed("ROLE_VIEWER")
public String getUsername2() {
    //...
}

@RolesAllowed({ "ROLE_VIEWER", "ROLE_EDITOR" })
public boolean isValidUsername2(String username) {
    //...
}
```

Del mismo modo, solo el usuario que tiene el rol *ROLE_VIEWER* puede ejecutar *getUsername2* .

Nuevamente, un usuario puede invocar *isValidUsername2* solo si tiene al menos una de las *funciones ROLE_VIEWER* o *ROLER_EDITOR* .

3.3. Usando *@PreAuthorize* y *@PostAuthorize* Anotaciones

Las anotaciones *@PreAuthorize* y *@PostAuthorize* proporcionan control de acceso basado en expresiones. Por lo tanto, los predicados se pueden escribir usando SpEL (Spring Expression Language) (http://www.baeldung.com/spring-expression-language) .

La anotación *@PreAuthorize* comprueba la expresión dada antes de ingresar el método , mientras que **la anotación *@PostAuthorize la* verifica después de la ejecución del método y puede alterar el resultado** .

Ahora, declaremos un método *getUsernameInUpperCase de la* siguiente manera:

```
@PreAuthorize("hasRole('ROLE_VIEWER')")
public String getUsernameInUpperCase() {
    return getUsername().toUpperCase();
}
```

El *@PreAuthorize ("hasRole ('ROLE_VIEWER')")* tiene el mismo significado que *@Secured ("ROLE_VIEWER")* que usamos en la sección anterior. Siéntase libre de descubrir más detalles de las expresiones de seguridad en artículos anteriores (http://www.baeldung.com/spring-security-expressions-basic) .

En consecuencia, la anotación *@Secured ({"ROLE_VIEWER", "ROLE_EDITOR"})* se puede reemplazar con *@PreAuthorize ("hasRole ('ROLE_VIEWER') o hasRole ('ROLE_EDITOR')")*:

```
@PreAuthorize("hasRole('ROLE_VIEWER') or hasRole('ROLE_EDITOR')")
public boolean isValidUsername3(String username) {
    //...
}
```

Además, **podemos usar el argumento método como parte de la expresión** :

```
@PreAuthorize("#username == authentication.principal.username")
public String getMyRoles(String username) {
    //...
}
```

Aquí, un usuario puede invocar el método *getMyRoles* solo si el valor del argumento *username* es el mismo que el del usuario actual del principal.

Vale la pena señalar que los *@PreAuthorize* expresiones pueden ser reemplazados por *@PostAuthorize* los .

Vamos a reescribir *getMyRoles* :

```
@PostAuthorize("#username == authentication.principal.username")
public String getMyRoles2(String username) {
    //...
}
```

En el ejemplo anterior, sin embargo, la autorización se retrasaría después de la ejecución del método de destino.

Además, **la anotación *@PostAuthorize* proporciona la capacidad de acceder al resultado del método** :

```
@PostAuthorize
  ("returnObject.username == authentication.principal.nickName")
public CustomUser loadUserDetail(String username) {
    return userRoleRepository.loadUserByUserName(username);
}
```

En este ejemplo, el método *loadUserDetail* solo se ejecutará correctamente si el *nombre* de *usuario* del *CustomUser* devuelto es igual al *apodo* del principal de autenticación actual .

En esta sección, usamos principalmente expresiones simples de Spring. Para escenarios más complejos, podríamos crear expresiones de seguridad personalizadas (http://www.baeldung.com/spring-security-create-new-custom-

security-expression) .

3.4. Usando *@PreFilter* y *@PostFilter* Anotaciones

Spring Security proporciona la anotación *@PreFilter* para filtrar un argumento de recopilación antes de ejecutar el método :

```
@PreFilter("filterObject != authentication.principal.username")
public String joinUsernames(List<String> usernames) {
    return usernames.stream().collect(Collectors.joining(";"));
}
```

En este ejemplo, unimos todos los nombres de usuario, excepto el que está autenticado.

Aquí, **nuestra expresión usa el nombre *filterObject* para representar el objeto actual en la colección** .

Sin embargo, si el método tiene más de un argumento que es un tipo de colección, necesitamos usar la propiedad *filterTarget* para especificar qué argumento queremos filtrar:

```
@PreFilter
  (value = "filterObject != authentication.principal.username",
  filterTarget = "usernames")
public String joinUsernamesAndRoles(
  List<String> usernames, List<String> roles) {

    return usernames.stream().collect(Collectors.joining(";"))
      + ":" + roles.stream().collect(Collectors.joining(";"));
}
```

Además, **también podemos filtrar la colección devuelta de un método utilizando la anotación *@PostFilter*** :

```
@PostFilter("filterObject != authentication.principal.username")
public List<String> getAllUsernamesExceptCurrent() {
    return userRoleRepository.getAllUsernames();
}
```

En este caso, el nombre *filterObject se* refiere al objeto actual en la colección devuelta.

Con esa configuración, Spring Security repetirá la lista devuelta y eliminará cualquier valor que coincida con el nombre de usuario del principal.

Puede encontrar más detalles de *@PreFilter* y *@PostFilter* en el artículo Spring Security - @PreFilter y @PostFilter (http://www.baeldung.com/spring-security-prefilter-postfilter) .

3.5. Meta-anotación de seguridad de método

Normalmente nos encontramos en una situación en la que protegemos diferentes métodos utilizando la misma configuración de seguridad.

En este caso, podemos definir una meta-anotación de seguridad:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@PreAuthorize("hasRole('VIEWER')")
public @interface IsViewer {
}
```

A continuación, podemos usar directamente la anotación @IsViewer para asegurar nuestro método:

```
@IsViewer
public String getUsername4() {
    //...
}
```

Las meta-anotaciones de seguridad son una gran idea porque agregan más semántica y desacoplan nuestra lógica empresarial del marco de seguridad.

3.6. Anotación de seguridad en el nivel de la clase

Si nos encontramos usando la misma anotación de seguridad para cada método dentro de una clase, podemos considerar poner esa anotación a nivel de clase:

```
@Service
@PreAuthorize("hasRole('ROLE_ADMIN')")
public class SystemService {

    public String getSystemYear(){
        //...
    }

    public String getSystemDate(){
        //...
    }
}
```

En el ejemplo anterior, la regla de seguridad *hasRole ('ROLE_ADMIN')* se aplicará a los métodos *getSystemYear* y *getSystemDate* .

3.7. Anotaciones de seguridad múltiples en un método

También podemos usar múltiples anotaciones de seguridad en un método:

```
@PreAuthorize("#username == authentication.principal.username")
@PostAuthorize("returnObject.username == authentication.principal.nickName")
public CustomUser securedLoadUserDetail(String username) {
    return userRoleRepository.loadUserByUserName(username);
}
```

Por lo tanto, Spring verificará la autorización antes y después de la ejecución del método *securedLoadUserDetail* .

4. Consideraciones importantes

Hay dos puntos que nos gustaría recordar sobre la seguridad del método:

- **Por defecto, el proxy AOP de Spring se usa para aplicar la seguridad del método:** si un método seguro A es llamado por otro método dentro de la misma clase, la seguridad en A se ignora por completo. Esto significa que el método A se ejecutará sin ninguna comprobación de seguridad. Lo mismo se aplica a los métodos privados
- **Spring *SecurityContext* está enlazado a hilos:** de manera predeterminada, el contexto de seguridad no se propaga a hilos hijo. Para obtener más información, podemos consultar el artículo de **Propagación de contexto de**

seguridad de primavera (http://www.baeldung.com/spring-security-async-principal-propagation)

5. Seguridad del método de prueba

5.1. Configuración

Para probar Spring Security con JUnit, necesitamos la dependencia de *spring-security-test* :

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
</dependency>
```

No es necesario que especifiquemos la versión de dependencia porque estamos usando el complemento Spring Boot. Las últimas versiones de esta dependencia se pueden encontrar en Maven Central (https://search.maven.org/#search%7Cga%7C1%7C%22spring-security-test%22) .

A continuación, configuremos una simple prueba de integración de Spring especificando el corredor y la configuración de *ApplicationContext* :

```
@RunWith(SpringRunner.class)
@ContextConfiguration
public class TestMethodSecurity {
    // ...
}
```

5.2. Prueba de nombre de usuario y roles

Ahora que nuestra configuración está lista, intentemos probar nuestro método *getUsername* , que está protegido por la anotación *@Secured ("ROLE_VIEWER"):*

```
@Secured("ROLE_VIEWER")
public String getUsername() {
    SecurityContext securityContext = SecurityContextHolder.getContext();
    return securityContext.getAuthentication().getName();
}
```

Como usamos la anotación *@Secured* aquí, se requiere que un usuario sea autenticado para invocar el método. De lo contrario, obtendremos una *AuthenticationCredentialsNotFoundException.*

Por **lo** tanto, **debemos proporcionar un usuario para probar nuestro método seguro. Para lograr esto, decoramos el método de prueba con *@WithMockUser* y proporcionamos un usuario y roles:**

```
@Test
@WithMockUser(username = "john", roles = { "VIEWER" })
public void givenRoleViewer_whenCallGetUsername_thenReturnUsername() {
    String userName = userRoleService.getUsername();

    assertEquals("john", userName);
}
```

Proporcionamos un usuario autenticado cuyo nombre de usuario es *john* y cuyo rol es *ROLE_VIEWER* . Si no especificamos el *nombre de usuario* o la *función* , el *nombre de usuario* predeterminado es *usuario* y la *función* predeterminada es *ROLE_USER* .

Tenga en cuenta que no es necesario agregar el prefijo *ROLE_* aquí, Spring Security agregará ese prefijo automáticamente.

Si no queremos tener ese prefijo, podemos considerar el uso de la *autoridad en* lugar de la *función.*

Por ejemplo, declaremos un método *getUsernameInLowerCase* :

```
@PreAuthorize("hasAuthority('SYS_ADMIN')")
public String getUsernameLC(){
    return getUsername().toLowerCase();
}
```

Podríamos probar que usando autoridades:

```
@Test
@WithMockUser(username = "JOHN", authorities = { "SYS_ADMIN" })
public void givenAuthoritySysAdmin_whenCallGetUsernameLC_thenReturnUsername(
    String username = userRoleService.getUsernameInLowerCase();

    assertEquals("john", username);
}
```

Convenientemente, **si queremos usar el mismo usuario para muchos casos de prueba, podemos declarar la anotación *@WithMockUser* en la clase de prueba** :

```
@RunWith(SpringRunner.class)
@ContextConfiguration
@WithMockUser(username = "john", roles = { "VIEWER" })
public class TestWithMockUserAtClassLevel {
    //...
}
```

Si quisiéramos ejecutar nuestra prueba como usuario anónimo, podríamos usar la anotación *@WithAnonymousUser*:

```
@Test(expected = AccessDeniedException.class)
@WithAnonymousUser
public void givenAnomynousUser_whenCallGetUsername_thenAccessDenied() {
    userRoleService.getUsername();
}
```

En el ejemplo anterior, esperamos una *AccessDeniedException* porque al usuario anónimo no se le otorga el rol *ROLE_VIEWER* o la autoridad *SYS_ADMIN*.

5.3. Prueba con un Custom *UserDetailsService*

Para la mayoría de las aplicaciones, es común usar una clase personalizada como principal de autenticación. En este caso, la clase personalizada necesita implementar *org.springframework.security.core.userdetails.* Interfaz *UserDetails*.

En este artículo, declaramos una clase *CustomUser* que amplía la implementación existente de *UserDetails*, que es *org.springframework.security.core.userdetails. Usuario:*

```
public class CustomUser extends User {
    private String nickName;
    // getter and setter
}
```

Retomemos el ejemplo con la anotación *@PostAuthorize* en la sección 3:

```
@PostAuthorize("returnObject.username == authentication.principal.nickName")
public CustomUser loadUserDetail(String username) {
    return userRoleRepository.loadUserByUserName(username);
}
```

En este caso, el método solo se ejecutará correctamente si el *nombre* de *usuario* del *CustomUser* devuelto es igual al *apodo* del principal de autenticación actual.

Si quisiéramos probar ese método **, podríamos proporcionar una implementación de *UserDetailsService* que podría cargar nuestro *CustomUser* basado en el nombre de usuario** :

```
@Test
@WithUserDetails(
  value = "john",
  userDetailsServiceBeanName = "userDetailService")
public void whenJohn_callLoadUserDetail_thenOK() {

    CustomUser user = userService.loadUserDetail("jane");

    assertEquals("jane", user.getNickName());
}
```

Aquí, la anotación *@WithUserDetails* indica que usaremos un *UserDetailsService* para inicializar a nuestro usuario autenticado. El servicio se refiere a la propiedad *userDetailsServiceBeanName* . Este *UserDetailsService* puede ser una implementación real o una falsificación para fines de prueba.

Además, el servicio utilizará el valor de la propiedad *de valor* como el nombre de usuario para cargar *DetallesUsuario* .

Convenientemente, también podemos decorar con una anotación *@WithUserDetails* en el nivel de clase, de forma similar a lo que hicimos con la anotación *@WithMockUser* .

5.4. Prueba con Meta Anotaciones

A menudo nos encontramos reutilizando el mismo usuario / roles una y otra vez en varias pruebas.

Para estas situaciones, es conveniente crear una *meta-anotación* .

Retomando el ejemplo anterior *@WithMockUser (username = "john", roles = {"VIEWER"})* , podemos declarar una meta-anotación como:

```
@Retention(RetentionPolicy.RUNTIME)
@WithMockUser(value = "john", roles = "VIEWER")
public @interface WithMockJohnViewer { }
```

Entonces simplemente podemos usar *@WithMockJohnViewer* en nuestra prueba:

```
@Test
@WithMockJohnViewer
public void givenMockedJohnViewer_whenCallGetUsername_thenReturnUsername() {
    String userName = userRoleService.getUsername();

    assertEquals("john", userName);
}
```

Del mismo modo, podemos usar meta-anotaciones para crear usuarios específicos del dominio usando *@WithUserDetails* .

6. Conclusión

En este tutorial, hemos explorado varias opciones para usar Method Security en Spring Security.

También hemos pasado por algunas técnicas para probar fácilmente la seguridad de los métodos y hemos aprendido a reutilizar usuarios burlados en diferentes pruebas.

Todos los ejemplos de este tutorial se pueden encontrar en Github (https://github.com/eugenp/tutorials/tree/master/spring-security-core) .

CATEGORÍAS

PRIMAVERA (HTTP://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
DESCANSO (HTTP://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTP://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SEGURIDAD (HTTP://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCIA (HTTP://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTP://WWW.BAELDUNG.COM/CATEGORY/JACKSON/)
HTTPCLIENT (HTTP://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTP://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIE

TUTORIAL "VOLVER A LO BÁSICO" DE JAVA (HTTP://WWW.BAELDUNG.COM/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (HTTP://WWW.BAELDUNG.COM/JACKSON)
TUTORIAL DE HTTPCLIENT 4 (HTTP://WWW.BAELDUNG.COM/HTTPCLIENT-GUIDE)
REST CON SPRING TUTORIAL (HTTP://WWW.BAELDUNG.COM/REST-WITH-SPRING-SERIES/)
TUTORIAL DE SPRING PERSISTENCE (HTTP://WWW.BAELDUNG.COM/PERSISTENCE-WITH-SPRING-SERIES/)
SEGURIDAD CON SPRING (HTTP://WWW.BAELDUNG.COM/SECURITY-SPRING)

ACERCA DE

ACERCA DE BAELDUNG (HTTP://WWW.BAELDUNG.COM/ABOUT/)
LOS CURSOS (HTTP://COURSES.BAELDUNG.COM)
TRABAJO DE CONSULTORÍA (HTTP://WWW.BAELDUNG.COM/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
EL ARCHIVO COMPLETO (HTTP://WWW.BAELDUNG.COM/FULL_ARCHIVE)
ESCRIBIR PARA BAELDUNG (HTTP://WWW.BAELDUNG.COM/CONTRIBUTION-GUIDELINES)
CONTACTO (HTTP://WWW.BAELDUNG.COM/CONTACT)
INFORMACIÓN DE LA COMPAÑÍA (HTTP://WWW.BAELDUNG.COM/BAELDUNG-COMPANY-INFO)
TÉRMINOS DE SERVICIO (HTTP://WWW.BAELDUNG.COM/TERMS-OF-SERVICE)
POLÍTICA DE PRIVACIDAD (HTTP://WWW.BAELDUNG.COM/PRIVACY-POLICY)
EDITORES (HTTP://WWW.BAELDUNG.COM/EDITORS)
KIT DE MEDIOS (PDF) (HTTPS://S3.AMAZONAWS.COM/BAELDUNG.COM/BAELDUNG+-+MEDIA+KIT.PDF)