Introduction to Spring Method Security

Last modified: April 30, 2020

by
baeldung

Spring Security

Authorization

1. Introduction

Simply put, Spring Security supports authorization semantics at the method level.

Typically, we could secure our service layer by, for example, restricting which roles are able to execute a

In this article, we're going to review the use of some security annotations first. Then, we'll focus on testing our method security with different strategies.

Further reading:

Spring Expression Language Guide

This article explores Spring Expression Language (SpEL), a powerful expression language that supports querying and manipulating object graphs at runtime.

Read more →

Spring Security Expressions – hasRole Example

Intro to the hasRole Spring Security Expression: set up Web Authorization by URL and in page, as well as Method Security with @PreAuthorize.

Read more →

A Custom Security Expression with Spring Security

A guide to creating a new, custom security expression with Spring Security, and then using the new expression with the Pre and Post authorize annotations.

Read more →

2. Enabling Method Security

First of all, to use Spring Method Security, we need to add the *spring-security-config* dependency:

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

We can find its latest version on Maven Central.

If we want to use Spring Boot, we can use the *spring-boot-starter-security* dependency which includes *spring-security-config*:

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

Again, the latest version can be found on Maven Central.

Next, we need to enable global Method Security:

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

- The *prePostEnabled* property enables Spring Security pre/post annotations
- The *securedEnabled* property determines if the *@Secured* annotation should be enabled
- The *jsr250Enabled* property allows us to use the *@RoleAllowed* annotation

We'll explore more about these annotations in the next section.

3. Applying Method Security

3.1. Using *@Secured* Annotation

The *@Secured* annotation is used to specify a list of roles on a method. Hence, a user only can access that

method if she has at least one of the specified roles.

Let's define a *getUsername* method:

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

Here, the *@Secured("ROLE_VIEWER")* annotation defines that only users who have the role *ROLE_VIEWER* are able to execute the *getUsername* method.

Besides, we can define a list of roles in a *@Secured* annotation:

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

In this case, the configuration states that if a user has either *ROLE_VIEWER* or *ROLE_EDITOR*, that user can invoke the *isValidUsername* method.

The *@Secured* annotation doesn't support Spring Expression Language (SpEL).

3.2. Using *@RoleAllowed* Annotation

The *@RoleAllowed* annotation is the JSR-250's equivalent annotation of the *@Secured* annotation.

Basically, we can use the *@RoleAllowed* annotation in a similar way as *@Secured*. Thus, we could re-define *getUsername* and *isValidUsername* methods:

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

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

Similarly, only the user who has role *ROLE_VIEWER* can execute *getUsername2*.

Again, a user is able to invoke *isValidUsername2* only if she has at least one of *ROLE_VIEWER* or *ROLER_EDITOR* roles.

3.3. Using *@PreAuthorize* and *@PostAuthorize* Annotations

Both *@PreAuthorize* and *@PostAuthorize* annotations provide expression-based access control. Hence,

predicates can be written using SpEL (Spring Expression Language).

The *@PreAuthorize* annotation checks the given expression before entering the method, whereas, **the *@PostAuthorize* annotation verifies it after the execution of the method and could alter the result**.

Now, let's declare a *getUsernameInUpperCase* method as below:

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

The *@PreAuthorize("hasRole('ROLE_VIEWER')")* has the same meaning as *@Secured("ROLE_VIEWER")* which we used in the previous section. Feel free to discover more security expressions details in previous articles.

Consequently, the annotation *@Secured({"ROLE_VIEWER","ROLE_EDITOR"})* can be replaced with *@PreAuthorize("hasRole('ROLE_VIEWER') or hasRole('ROLE_EDITOR')"):*

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

Moreover, **we can actually use the method argument as part of the expression**:

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

Here, a user can invoke the *getMyRoles* method only if the value of the argument *username* is the same as current principal's username.

It's worth to note that *@PreAuthorize* expressions can be replaced by *@PostAuthorize* ones.

Let's rewrite *getMyRoles*:

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

In the previous example, however, the authorization would get delayed after the execution of the target method.

Additionally, **the *@PostAuthorize* annotation provides the ability to access the method result**:

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

In this example, the *loadUserDetail* method would only execute successfully if the *username* of the returned *CustomUser* is equal to the current authentication principal's *nickname*.

In this section, we mostly use simple Spring expressions. For more complex scenarios, we could create custom security expressions.

3.4. Using *@PreFilter* and *@PostFilter* Annotations

Spring Security provides the *@PreFilter* annotation to filter a collection argument before executing the method:

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

In this example, we're joining all usernames except for the one who is authenticated.

Here, **our expression uses the name *filterObject* to represent the current object in the collection**.

However, if the method has more than one argument which is a collection type, we need to use the *filterTarget* property to specify which argument we want to filter:

```
@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(";"));
}
```

Additionally, **we can also filter the returned collection of a method by using *@PostFilter* annotation**:

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

In this case, the name *filterObject* refers to the current object in the returned collection.

With that configuration, Spring Security will iterate through the returned list and remove any value which matches with the principal's username.

More detail of *@PreFilter* and *@PostFilter* can be found in the Spring Security – @PreFilter and @PostFilter article.

3.5. Method Security Meta-Annotation

We typically find ourselves in a situation where we protect different methods using the same security configuration.

In this case, we can define a security meta-annotation:

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

Next, we can directly use the @IsViewer annotation to secure our method:

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

Security meta-annotations are a great idea because they add more semantics and decouple our business logic from the security framework.

3.6. Security Annotation at the Class Level

If we find ourselves using the same security annotation for every method within one class, we can consider putting that annotation at class level:

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

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

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

In above example, the security rule *hasRole('ROLE_ADMIN')* will be applied to both *getSystemYear* and *getSystemDate* methods.

3.7. Multiple Security Annotations on a Method

We can also use multiple security annotations on one method:

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

Hence, Spring will verify authorization both before and after the execution of the *securedLoadUserDetail* method.

4. Important Considerations

There are two points we'd like to remind regarding method security:

- **By default, Spring AOP proxying is used to apply method security –** if a secured method A is called by another method within the same class, security in A is ignored altogether. This means method A will execute without any security checking. The same applies to private methods
- **Spring *SecurityContext* is thread-bound –** by default, the security context isn't propagated to child-threads. For more information, we can refer to **Spring Security Context Propagation** article

5. Testing Method Security

5.1. Configuration

To test Spring Security with JUnit, we need the *spring-security-test* dependency:

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

We don't need to specify the dependency version because we're using the Spring Boot plugin. Latest versions of this dependency can be found on Maven Central.

Next, let's configure a simple Spring Integration test by specifying the runner and the *ApplicationContext* configuration:

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

5.2. Testing Username and Roles

Now that our configuration is ready, let's try to test our *getUsername* method which is secured by the annotation *@Secured(“ROLE_VIEWER”):*

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

Since we use the *@Secured* annotation here, it requires a user to be authenticated to invoke the method. Otherwise, we'll get an *AuthenticationCredentialsNotFoundException.*

Hence, **we need to provide a user to test our secured method. To achieve this, we decorate the test method with *@WithMockUser* and provide a user and roles:**

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

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

```
}
```

We've provided an authenticated user whose username is *john* and whose role is *ROLE_VIEWER*. If we don't specify the *username* or *role*, the default *username* is *user* and default *role* is *ROLE_USER*.

Note that it isn't necessary to add the *ROLE_* prefix here, Spring Security will add that prefix automatically.

If we don't want to have that prefix, we can consider using *authority* instead of *role*.

For example, let's declare a *getUsernameInLowerCase* method:

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

We could test that using authorities:

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

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

Conveniently, **if we want to use the same user for many test cases, we can declare the *@WithMockUser* annotation at test class**:

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

If we wanted to run our test as an anonymous user, we could use the *@WithAnonymousUser* annotation:

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

In the example above, we expect an *AccessDeniedException* because the anonymous user isn't granted the role *ROLE_VIEWER* or the authority *SYS_ADMIN*.

5.3. Testing With a Custom *UserDetailsService*

For most applications, it's common to use a custom class as authentication principal. In this case, the custom class needs to implement the *org.springframework.security.core.userdetails.UserDetails* interface.

In this article, we declare a *CustomUser* class which extends the existing implementation of *UserDetails*, which is *org.springframework.security.core.userdetails.User:*

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

Let's take back the example with the *@PostAuthorize* annotation in section 3:

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

In this case, the method would only execute successfully if the *username* of the returned *CustomUser* is equal to the current authentication principal's *nickname*.

If we wanted to test that method**, we could provide an implementation of *UserDetailsService* which could load our *CustomUser* based on the username**:

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

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

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

Here, the *@WithUserDetails* annotation states that we'll use a *UserDetailsService* to initialize our authenticated user. The service is referred by the *userDetailsServiceBeanName* property. This *UserDetailsService* might be a real implementation or a fake for testing purposes.

Additionally, the service will use the value of the property *value* as the username to load *UserDetails*.

Conveniently, we can also decorate with a *@WithUserDetails* annotation at the class level, similarly to what we did with the *@WithMockUser* annotation.

5.4. Testing With Meta Annotations

We often find ourselves reusing the same user/roles over and over again in various tests.

For these situations, it's convenient to create a *meta-annotation.*

Taking back the previous example *@WithMockUser(username=”john”, roles={“VIEWER”})*, we can declare a meta-annotation as:

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

Then we can simply use *@WithMockJohnViewer* in our test:

```
@Test
@WithMockJohnViewer
public void
givenMockedJohnViewer_whenCallGetUsername_thenReturnUsername() {
```

```
    String userName = userRoleService.getUsername();

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

Likewise, we can use meta-annotations to create domain-specific users using *@WithUserDetails*.

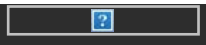
6. Conclusion

In this tutorial, we've explored various options for using Method Security in Spring Security.

We also have gone through a few techniques to easily test method security and learned how to reuse mocked users in different tests.

All examples of this tutorial can be found over on Github.

Comments are closed on this article!



CATEGORIES

SPRING
REST
JAVA
SECURITY
PERSISTENCE
JACKSON
HTTP CLIENT-SIDE
KOTLIN

SERIES

JAVA “BACK TO BASICS” TUTORIAL
JACKSON JSON TUTORIAL
HTTPCLIENT 4 TUTORIAL
REST WITH SPRING TUTORIAL
SPRING PERSISTENCE TUTORIAL
SECURITY WITH SPRING

ABOUT

ABOUT BAELDUNG
THE COURSES
JOBS
META BAELDUNG
THE FULL ARCHIVE
WRITE FOR BAELDUNG
EDITORS
OUR PARTNERS
ADVERTISE ON BAELDUNG

TERMS OF SERVICE | PRIVACY POLICY | COMPANY INFO | CONTACT