Intro to Spring Security Expressions

Last modified: December 3, 2022

Written by: baeldung (https://www.baeldung.com/author/baeldung)

Spring Security (https://www.baeldung.com/category/spring/spring-security)

Authorization (https://www.baeld

Guru99.com

1. Introduction

In this tutorial, we'll focus on Spring Security Expressions and practical examples using these expressions.

Before looking at more complex implementations, such as ACL, it's important to have a solid grasp on security expressions, as they can be quite flexible and powerful if used correctly.

2. Maven Dependencies

In order to use Spring Security, we need to include the following section in our *pom.xml* file:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-web</artifactId>
        <version>5.6.0</version>
   </dependency>
</dependencies>
```

The latest version can be found here (https://search.maven.org/classic/#search%7Cga%7C1%7Ca%3A%22spring-security-web%22).

Please note that this dependency only covers Spring Security; we'll need to add s*pring-core* and *spring-context* for a full web application.

3. Configuration

First, let's take a look at a Java configuration:

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
@ComponentScan("com.baeldung.security")
public class SecurityJavaConfig {
    ...
}
```

We can, of course, do an XML configuration as well:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans ...>
    <global-method-security pre-post-annotations="enabled"/>
</beans:beans>
```

4. Web Security Expressions

Now let's explore the security expressions:

- *hasRole*, *hasAnyRole*
- *hasAuthority*, *hasAnyAuthority*
- *permitAll*, *denyAll*
- *isAnonymous*, *isRememberMe*, *isAuthenticated*, *isFullyAuthenticated*
- *principal*, *authentication*
- *hasPermission*

4.1. *hasRole, hasAnyRole*

These expressions are responsible for defining the access control or authorization to specific URLs and methods in our application:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws
Exception {
    ...
    .antMatchers("/auth/admin/*").hasRole("ADMIN")
    .antMatchers("/auth/*").hasAnyRole("ADMIN","USER")
    ...
}
```

In the above example, we specified access to all the links starting with */auth/*, restricting them to users that log in with role *USER* or role *ADMIN.* Moreover, to access links starting with */auth/admin/*, we need to have an *ADMIN* role in the system.

We can achieve the same configuration in an XML file by writing:

```
<http>
    <intercept-url pattern="/auth/admin/*" access="hasRole('ADMIN')"/>
    <intercept-url pattern="/auth/*"
access="hasAnyRole('ADMIN','USER')"/>
</http>
```

4.2. *hasAuthority, hasAnyAuthority*

Roles and authorities are similar in Spring.

The main difference is that roles have special semantics. Starting with Spring Security 4, the '*ROLE_*' prefix is automatically added (if it's not already there) by any role-related method.

So *hasAuthority('ROLE_ADMIN')* is similar to *hasRole('ADMIN')* because the '*ROLE_*' prefix gets added automatically.

The benefit to using authorities is that we don't have to use the *ROLE_* prefix at all.

Here's a quick example defining users with specific authorities:

```
@Bean
public InMemoryUserDetailsManager userDetailsService() {
    UserDetails admin = User.withUsername("admin")
        .password(encoder().encode("adminPass"))
        .roles("ADMIN")
        .build();
    UserDetails user = User.withUsername("user")
        .password(encoder().encode("userPass"))
        .roles("USER")
        .build();
    return new InMemoryUserDetailsManager(admin, user);
}
```

We can then use these authorities expressions:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws
Exception {
    ...
    .antMatchers("/auth/admin/*").hasAuthority("ADMIN")
    .antMatchers("/auth/*").hasAnyAuthority("ADMIN", "USER")
    ...
}
```

As we can see, we don't mention roles here at all.

Additionally, starting with Spring 5, we need a *PasswordEncoder* (/spring-security-5-default-password-encoder) bean:

```
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

Finally, we have the option to achieve the same functionality using XML configuration as well:

```
<authentication-manager>
    <authentication-provider>
        <user-service>
            <user name="user1" password="user1Pass"
authorities="ROLE_USER"/>
            <user name="admin" password="adminPass"
authorities="ROLE_ADMIN"/>
        </user-service>
    </authentication-provider>
</authentication-manager>
<bean name="passwordEncoder"

class="org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder
"/>
```

And:

```
<http>
    <intercept-url pattern="/auth/admin/*"
access="hasAuthority('ADMIN')"/>
    <intercept-url pattern="/auth/*"
access="hasAnyAuthority('ADMIN','USER')"/>
</http>
```

4.3. *permitAll, denyAll*

These two annotations are also quite straightforward. We may either permit or deny access to some URL in our service.

Let's have a look at the example:

```
...
.antMatchers("/*").permitAll()
...
```

With this config, we'll authorize all users (both anonymous and logged in) to access the page starting with '/' (for example, our homepage).

We can also deny access to our entire URL space:

```
...
.antMatchers("/*").denyAll()
...
```

And again, we can do the same configuration with XML as well:

```
<http auto-config="true" use-expressions="true">
    <intercept-url access="permitAll" pattern="/*" /> <!-- Choose only
one -->
    <intercept-url access="denyAll" pattern="/*" /> <!-- Choose only
one -->
</http>
```

4.4. *isAnonymous, isRememberMe, isAuthenticated, isFullyAuthenticated*

In this subsection, we'll focus on expressions related to the login status of the user. Let's start with a user that didn't log in to our page. By specifying the following in the Java config, we'll enable all unauthorized users to access our main page:

```
...
.antMatchers("/*").anonymous()
...
```

Here's the same in XML config:

```
<http>
    <intercept-url pattern="/*" access="isAnonymous()"/>
</http>
```

If we want to secure the website so that everyone who uses it needs to log in, we'll need to use the *isAuthenticated()* method:

```
...
.antMatchers("/*").authenticated()
...
```

Or we can use the XML version:

```
<http>
    <intercept-url pattern="/*" access="isAuthenticated()"/>
</http>
```

We also have two additional expressions, *isRememberMe()* and *isFullyAuthenticated()*. Through the use of cookies, Spring enables remember-me capabilities, so there's no need to log into the system each time. We can read more about *Remember Me* here (/spring-security-remember-me).

In order to give access to users that were logged in by the remember me function, we can use:

```
...
.antMatchers("/*").rememberMe()
...
```

We can also use the XML version:

```
<http>
    <intercept-url pattern="*" access="isRememberMe()"/>
</http>
```

Finally, some parts of our services require the user to be authenticated again, even if the user is already logged in. For example, let's say a user wants to change the settings or payment information; it's good practice to ask for manual authentication in the more sensitive areas of the system.

In order to do this, we can specify *isFullyAuthenticated()*, which returns *true* if the user isn't an anonymous or remember-me user:

```
...
.antMatchers("/*").fullyAuthenticated()
...
```

Here's the XML version:

```
<http>
    <intercept-url pattern="*" access="isFullyAuthenticated()"/>
</http>
```

4.5. *principal, authentication*

These expressions allow us to access the *principal* object representing the current authorized (or anonymous) user and the current *Authentication* object from the *SecurityContext*, respectively.

We can, for example, use *principal* to load a user's email, avatar, or any other data that's accessible from the logged-in user.

And *authentication* provides information about the full *Authentication* object, along with its granted authorities.

Both of these expressions are described in further detail in the article Retrieve User Information in Spring Security (/get-user-in-spring-security).

4.6. *hasPermission* APIs

This expression is documented (https://docs.spring.io/spring-security/site/docs/5.1.5.RELEASE/reference/html/authorization.html), and intended to be a bridge between the expression system and Spring Security's ACL system, allowing us to specify authorization constraints on individual domain objects based on abstract permissions.

Let's look at an example. Imagine that we have a service that allows cooperative article writing, with a main editor who decides which articles proposed by the authors should be published.

In order to allow the use of such a service, we can create the following methods with access control methods:

```
@PreAuthorize("hasPermission(#article, 'isEditor')")
public void acceptArticle(Article article) {
   …
}
```

Only the authorized user can call this method, and they need to have *isEditor* permission in the service.

We also need to remember to explicitly configure a *PermissionEvaluator* in our application context, where *customInterfaceImplementation* will be the class that implements *PermissionEvaluator*.

```
<global-method-security pre-post-annotations="enabled">
    <expression-handler ref="expressionHandler"/>
</global-method-security>

<bean id="expressionHandler"
    class="org.springframework.security.access.expression
      .method.DefaultMethodSecurityExpressionHandler">
    <property name="permissionEvaluator"
ref="customInterfaceImplementation"/>
</bean>
```

Of course, we can also do this with Java configuration as well:

```
@Override
protected MethodSecurityExpressionHandler expressionHandler() {
    DefaultMethodSecurityExpressionHandler expressionHandler =
      new DefaultMethodSecurityExpressionHandler();
    expressionHandler.setPermissionEvaluator(new
CustomInterfaceImplementation());
    return expressionHandler;
}
```

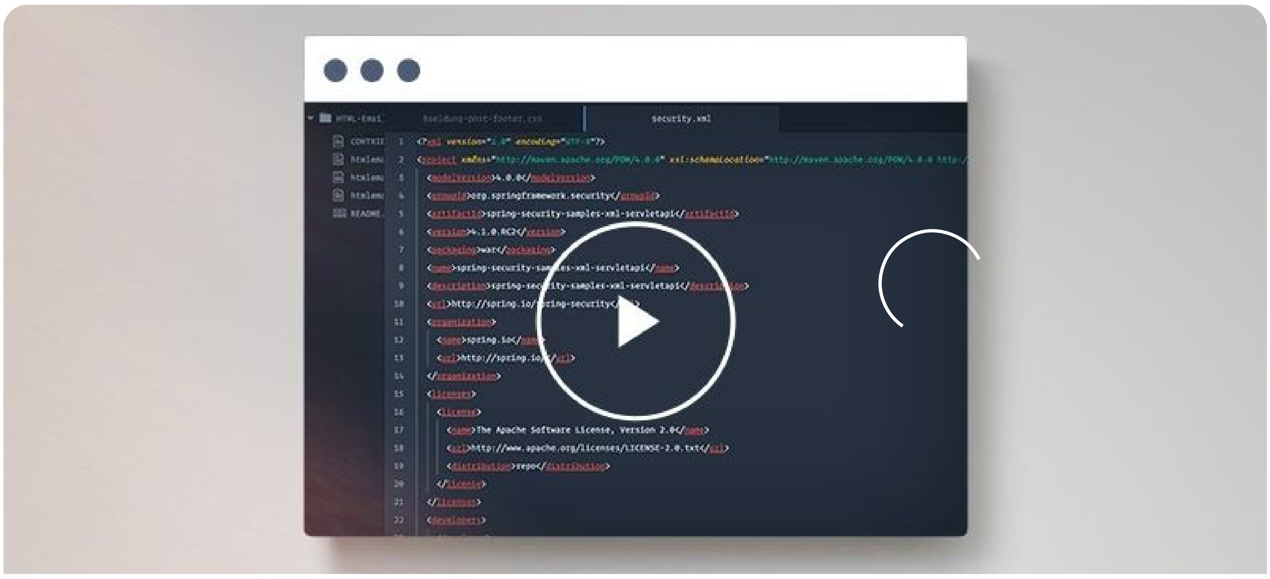
5. Conclusion

This article is a comprehensive introduction and guide to Spring Security Expressions.

All of the examples discussed here are available on the GitHub project (https://github.com/eugenp/tutorials/tree/master/spring-security-modules/spring-security-web-rest).

I just announced the new *Learn Spring Security* course, including the full material focused on the new OAuth2 stack in Spring Security 5:

>> CHECK OUT THE COURSE (/learn-spring-security-course#table)



```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http:/
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-samples-xml-servletapi</artifactId>
  <version>4.1.0.RC2</version>
  <packaging>war</packaging>
  <name>spring-security-samples-xml-servletapi</name>
  <description>spring-security-samples-xml-servletapi</description>
  <url>http://spring.io/spring-security</url>
  <organization>
    <name>spring.io</name>
    <url>http://spring.io/</url>
  </organization>
  <licenses>
    <license>
      <name>The Apache Software License, Version 2.0</name>
      <url>http://www.apache.org/licenses/LICENSE-2.0.txt</url>
      <distribution>repo</distribution>
    </license>
  </licenses>
  <developers>
```

Comments are closed on this article!

APACHE HTTPCLIENT TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)
SPRING REACTIVE TUTORIALS (/SPRING-REACTIVE-GUIDE)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE FULL ARCHIVE (/FULL_ARCHIVE)
EDITORS (/EDITORS)
JOBS (/TAG/ACTIVE-JOB/)
OUR PARTNERS (/PARTNERS)
PARTNER WITH BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)