### What is Spring Security?

Spring Security is one of the Spring Framework’s Security modules. It is a Java SE/Java EE Security Framework to provide Authentication, Authorization, SSO and other Security features for Web Applications or Enterprise Applications.

Some of the benefits of using Spring Security are:

1. Proven technology, it’s better to use this than reinvent the wheel. Security is something where we need to take extra care, otherwise our application will be vulnerable for attackers.
2. Prevents some of the common attacks such as CSRF, session fixation attacks.
3. Easy to integrate in any web application. We don’t need to modify web application configurations, spring automatically injects security filters to the web application.
4. Provides support for authentication by different ways – in-memory, DAO, JDBC, LDAP and many more.
5. Provides option to ignore specific URL patterns, good for serving static HTML, image files.
6. Support for groups and roles.

## UserDetailsService

If we want to use any DAO class for authentication, we need to implement UserDetailsService interface. Once the DAO is configured, it’s loadUserByUsername() is used to validate the user.

**AuthenticationManagerBuilder**

[SecurityBuilder](https://docs.spring.io/spring-security/site/docs/4.0.x/apidocs/org/springframework/security/config/annotation/SecurityBuilder.html) used to create an [AuthenticationManager](https://docs.spring.io/spring-security/site/docs/4.0.x/apidocs/org/springframework/security/authentication/AuthenticationManager.html" \o "interface in org.springframework.security.authentication). Allows for easily building in memory authentication, LDAP authentication, JDBC based authentication, adding [UserDetailsService](https://docs.spring.io/spring-security/site/docs/4.0.x/apidocs/org/springframework/security/core/userdetails/UserDetailsService.html" \o "interface in org.springframework.security.core.userdetails), and adding [AuthenticationProvider](https://docs.spring.io/spring-security/site/docs/4.0.x/apidocs/org/springframework/security/authentication/AuthenticationProvider.html" \o "interface in org.springframework.security.authentication)'s.

Spring 4 Security Module supports the following options to store and manage User Credentials:

1. In-Memory Store
2. Relations Databases(RDBMS)
3. No SQL Data Stores
4. LDAP
5. @EnableWebSecurity Annotation is used to enable web security in any web application.
6. should extend “WebSecurityConfigurerAdapter” class or implement related interface.
7. configureGlobal() method is used to store and mange User Credentials.
8. In configureGlobal() method, we can use authorities() method to define our application Roles like “ROLE\_USER”. We can also use roles() method for same purpose.
9. Difference between authorities() and roles() methods:
10. authorities() needs a complete role name like “ROLE\_USER”  
    roles() needs a role name like “USER”. It will automatically add “ROLE\_” value to this “USER” role name.

#### @PreAuthorize / @PostAuthorize

Spring’s @PreAuthorize/@PostAuthorize annotations are preferred way for applying method-level security, and supports Spring Expression Language out of the box, and provide expression-based access control.

**@PreAuthorize** is suitable for verifying authorization before entering into method. @PreAuthorize can take into account, the roles/permissions of logged-in User, argument passed to the method etc.

**@PostAuthorize** , not often used though, checks for authorization after method have been executed, so it is suitable for verifying authorization on returned values. Spring EL provides **returnObject** object that can be accessed in expression language and reflects the actual object returned from method.

Please refer to [Common Built-In Expressions](http://docs.spring.io/spring-security/site/docs/4.0.1.RELEASE/reference/htmlsingle/#el-common-built-in) to get the complete list of supported expressions.

## **Enable pre-post Annotations**

To use @PreAuthorize and @PostAuthorize annotations in our Spring Security application, we need to enable pre-post annotations.  
Find the code for XML configuration.Enable pre-post annotations using <global-method-security> namespace as given below.

@EnableGlobalMethodSecurity(prePostEnabled = true)  – this enables the annotations on the JPA model.

<global-method-security pre-post-annotations="enabled"/>

In case we are using Java configuration, then that class needs to be annotated with @EnableGlobalMethodSecurity as given below.

@Configuration

@EnableWebSecurity

@EnableGlobalMethodSecurity(prePostEnabled=true)

public class SecurityConfig extends WebSecurityConfigurerAdapter {

------

}

Prior to Spring Security 5.0, the default PasswordEncoder was NoOpPasswordEncoder which required plain text passwords but is insecure. Spring Security 5.x onwards, the default PasswordEncoder is DelegatingPasswordEncoder, which requires a **Password Storage Format**. You can find more details on migrating to Spring Security 5

Password Storage Format

The general format for a password is:

{id}encodedPassword

where:

**"id"**

is an identifier used to look up which PasswordEncoder should be used.

**"encodedPassword"**

is the original encoded password for the selected PasswordEncoder.

Most commonly used PasswordEncoders with their id’s are:

1. "noop" which uses plain text NoOpPasswordEncoder
2. "bcrypt" which uses `BCryptPasswordEncoder'
3. "scrypt" which uses SCryptPasswordEncoder
4. "pbkdf2" which uses 'Pbkdf2PasswordEncoder'
5. "sha256" which uses StandardPasswordEncoder

Example of a Password that is encoded using bcrypt is:

{bcrypt}$2a$10$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1tlRy.fqvM/BG

DelegatingPasswordEncoder solves many of the challenges:

1. It allows validating passwords in modern and legacy formats.
2. It allows for upgrading to a newer encoding in the future.
3. It ensures that passwords are encoded using the current password storage recommendations.

We can easily create an instance of DelegatingPasswordEncoder using the below code:

PasswordEncoder passwordEncoder =

PasswordEncoderFactories.createDelegatingPasswordEncoder();

or infact, you can create a bean for the same:

@Bean

public PasswordEncoder passwordEncoder() {

return PasswordEncoderFactories.createDelegatingPasswordEncoder();

}

Solution 1. Add password storage format

We can add password storage format, whoch is {noop} for plain text passwords.

### What is NoOpPasswordEncoder

NoOpPasswordEncoder is a password encoder that does nothing.

## Password Storage Format

The general format for a password is:

{id}encodedPassword

**Code Explanation**

* 1. In configureGlobal() method, we have added two users: One user with “ROLE\_USER” role and another user with both “ROLE\_USER” and “ROLE\_ADMIN” roles. That means this second user will act as a Admin User. Like this we can configure any number of users and roles.
  2. We can use either authorities(ROLE) or roles(ROLE) methods to configure Roles in our application.
  3. Difference between authorities() and roles() methods:
     + authorities() needs complete role name like “ROLE\_USER”
     + roles() needs role name like “USER”. It will automatically adds “ROLE\_” value to this “USER” role name.
  4. In configure() method, we have defined different URLs with required Access Roles.

antMatchers("/homePage")

.access("hasRole('ROLE\_USER') or hasRole('ROLE\_ADMIN')")

This code snippet configures that “/homePage” is available for both USER and ADMIN Roles.

.antMatchers("/userPage").access("hasRole('ROLE\_USER')")

.antMatchers("/adminPage").access("hasRole('ROLE\_ADMIN')")

This code snippet configures that “/userPage” is accessible by “USER” role only and /adminPage” is accessible by “ADMIN” role only.

If other roles access these pages, we will get access “403 Access is Denied” Error message.

Refer Links:

1. <https://www.journaldev.com/8748/spring-security-role-based-access-authorization-example>

2. <https://www.concretepage.com/spring/spring-security/preauthorize-postauthorize-in-spring-security>

3. <https://www.dineshonjava.com/spring-security-java-based-configuration-with-example/>

## 13.1 CSRF Attacks

Before we discuss how Spring Security can protect applications from CSRF attacks, we will explain what a CSRF attack is. Let's take a look at a concrete example to get a better understanding.

Assume that your bank's website provides a form that allows transferring money from the currently logged in user to another bank account. For example, the HTTP request might look like:

POST /transfer HTTP/1.1

Host: bank.example.com

Cookie: JSESSIONID=randomid; Domain=bank.example.com; Secure; HttpOnly

Content-Type: application/x-www-form-urlencoded

amount=100.00&routingNumber=1234&account=9876

Now pretend you authenticate to your bank's website and then, without logging out, visit an evil website. The evil website contains an HTML page with the following form:

<form action="https://bank.example.com/transfer" method="post">

<input type="hidden"

name="amount"

value="100.00"/>

<input type="hidden"

name="routingNumber"

value="evilsRoutingNumber"/>

<input type="hidden"

name="account"

value="evilsAccountNumber"/>

<input type="submit"

value="Win Money!"/>

</form>

You like to win money, so you click on the submit button. In the process, you have unintentionally transferred $100 to a malicious user. This happens because, while the evil website cannot see your cookies, the cookies associated with your bank are still sent along with the request.

Worst yet, this whole process could have been automated using JavaScript. This means you didn't even need to click on the button. So how do we protect ourselves from such attacks?

## 13.2 Synchronizer Token Pattern

The issue is that the HTTP request from the bank's website and the request from the evil website are exactly the same. This means there is no way to reject requests coming from the evil website and allow requests coming from the bank's website. To protect against CSRF attacks we need to ensure there is something in the request that the evil site is unable to provide.

One solution is to use the [Synchronizer Token Pattern](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet#General_Recommendation:_Synchronizer_Token_Pattern). This solution is to ensure that each request requires, in addition to our session cookie, a randomly generated token as an HTTP parameter. When a request is submitted, the server must look up the expected value for the parameter and compare it against the actual value in the request. If the values do not match, the request should fail.

We can relax the expectations to only require the token for each HTTP request that updates state. This can be safely done since the same origin policy ensures the evil site cannot read the response. Additionally, we do not want to include the random token in HTTP GET as this can cause the tokens to be leaked.

Let's take a look at how our example would change. Assume the randomly generated token is present in an HTTP parameter named \_csrf. For example, the request to transfer money would look like this:

POST /transfer HTTP/1.1

Host: bank.example.com

Cookie: JSESSIONID=randomid; Domain=bank.example.com; Secure; HttpOnly

Content-Type: application/x-www-form-urlencoded

amount=100.00&routingNumber=1234&account=9876&\_csrf=<secure-random>

You will notice that we added the \_csrf parameter with a random value. Now the evil website will not be able to guess the correct value for the \_csrf parameter (which must be explicitly provided on the evil website) and the transfer will fail when the server compares the actual token to the expected token.