Spring Security: Authentication and Authorization In-Depth

Last updated on August 21, 2020 - 41 comments

You can use this guide to understand what Spring Security is and how its core features like authentication, authorization or common exploit protection work. Also, a comprehensive FAQ.

(Editor's note: At ~6500 words, you probably don't want to try reading this on a mobile device. Bookmark it and come back later.)

Introduction

Sooner or later everyone needs to add security to his project and in the Spring ecosystem you do that with the help of the Spring Security library.

So you go along, add Spring Security to your Spring Boot (or plain Spring) project and suddenly...

- ...you have auto-generated login-pages.
- ...you cannot execute POST requests anymore.
- ...your whole application is on lockdown and prompts you to enter a username and password.

Having survived the subsequent mental breakdown, you might be interested in how all of this works.

What is Spring Security and how does it work?

The short answer:

At its core, Spring Security is really just a bunch of servlet filters that help you add authentication and authorization to your web application.

It also integrates well with frameworks like Spring Web MVC (or Spring Boot), as well as with standards like OAuth2 or SAML. And it auto-generates login/logout pages and protects against common exploits like CSRF.

Now, that doesn't really help, does it?

Luckily, there's also a long answer:

The remainder of this article.

Web Application Security: 101

Before you become a Spring Security Guru, you need to understand three important concepts:

1. Authentication
2. Authorization
3. Servlet Filters

Parental Advice: Don't skip this section, as it is the basis for *everything* that Spring Security does. Also, I'll make it as interesting as possible.

1. Authentication

First off, if you are running a typical (web) application, you need your users to *authenticate*. That means your application needs to verify if the user is *who* he claims to be, typically done with a username and password check.

User: "I'm the president of the United States. My `username` is: potus!"

Your webapp: "Sure sure, what's your `password` then, Mr. President?"

User: "My password is: th3don4ld".

Your webapp: "Correct. Welcome, Sir!"

2. Authorization

In simpler applications, authentication might be enough: As soon as a user authenticates, she can access every part of an application.

But most applications have the concept of permissions (or roles). Imagine: customers who have access to the public-facing frontend of your webshop, and administrators who have access to a separate admin area.

Both type of users need to login, but the mere fact of authentication doesn't say anything about what they are allowed to do in your system. Hence, you also need to check the permissions of an authenticated user, i.e. you need to *authorize* the user.

User: "Let me play with that nuclear football...."

Your webapp: "One second, I need to check your `permissions` first.....yes Mr. President, you have the right clearance level. Enjoy."

User: "What was that red button again...??"

3. Servlet Filters

Who the hell am I?

I'm @MarcoBehler and I share everything I know about making awesome software through my guides, screencasts, talks and courses.

Follow me on Twitter to find out what I'm currently working on.

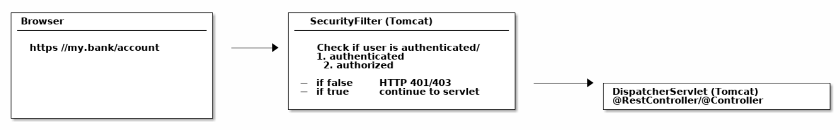
Last but not least, let’s have a look at Servlet Filters. What do they have to do with authentication and authorization? (If you are completely new to Java Servlets or Filters, I advise you to read the old, but still very valid Head First Servlets book.)

Why use Servlet Filters?

Think back to my other article, where we found out that basically any Spring web application is *just* one servlet: Spring’s good old DispatcherServlet, that redirects incoming HTTP requests (e.g. from a browser) to your @Controllers or @RestControllers.

The thing is: There is no security hardcoded into that DispatcherServlet and you also very likely don’t want to fumble around with a raw HTTP Basic Auth header in your @Controllers. Optimally, the authentication and authorization should be done *before* a request hits your @Controllers.

Luckily, there’s a way to do exactly this in the Java web world: you can put filters *in front* of servlets, which means you could think about writing a SecurityFilter and configure it in your Tomcat (servlet container/application server) to filter every incoming HTTP request before it hits your servlet.


Browser
https //my.bank/account
SecurityFilter (Tomcat)
Check if user is authenticated/
1. authenticated
2. authorized
– if false HTTP 401/403
– if true continue to servlet
DispatcherServlet (Tomcat)
@RestController/@Controller

A naive SecurityFilter

A SecurityFilter has roughly 4 tasks and a naive and overly-simplified implementation could look like this:

```
import javax.servlet.*;
import javax.servlet.http.HttpFilter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class SecurityServletFilter extends HttpFilter {

    @Override
    protected void doFilter(HttpServletRequest request,
HttpServletResponse response, FilterChain chain) throws
IOException, ServletException {

        UsernamePasswordToken token =
extractUsernameAndPasswordFrom(request);  // (1)

        if (notAuthenticated(token)) {  // (2)
            // either no or wrong username/password
            // unfortunately the HTTP status code is
called "unauthorized", instead of "unauthenticated"

response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
 // HTTP 401.
            return;
        }

        if (notAuthorized(token, request)) { // (3)
            // you are logged in, but don't have the
proper rights

response.setStatus(HttpServletResponse.SC_FORBIDDEN);
// HTTP 403
            return;
        }

        // allow the HttpRequest to go to Spring's
DispatcherServlet
        // and @RestControllers/@Controllers.
        chain.doFilter(request, response); // (4)
    }

    private UsernamePasswordToken
extractUsernameAndPasswordFrom(HttpServletRequest
request) {
        // Either try and read in a Basic Auth HTTP
Header, which comes in the form of user:password
        // Or try and find form login request
parameters or POST bodies, i.e. "username=me" &
"password="myPass"
        return checkVariousLoginOptions(request);
    }


    private boolean
notAuthenticated(UsernamePasswordToken token) {
        // compare the token with what you have in your
database...or in-memory...or in LDAP...
        return false;
    }

    private boolean notAuthorized(UsernamePasswordToken
token, HttpServletRequest request) {
       // check if currently authenticated user has the
permission/role to access this request's /URI
       // e.g. /admin needs a ROLE_ADMIN , /callcenter
needs ROLE_CALLCENTER, etc.
       return false;
    }
}
```

1. First, the filter needs to extract a username/password from the request. It could be via a Basic Auth HTTP Header, or form fields, or a cookie, etc.
2. Then the filter needs to validate that username/password combination against *something*, like a database.
3. The filter needs to check, after successful authentication, that the user is authorized to access the requested URI.
4. If the request *survives* all these checks, then the filter can let the request go through to your DispatcherServlet, i.e. your @Controllers.

FilterChains

Reality Check: While the above code ~~works~~ compiles, it would sooner or later lead to one monster filter with a ton of code for various authentication and authorization mechanisms.

In the real-world, however, you would split this one filter up into *multiple* filters, that you then *chain* together.

For example, an incoming HTTP request would…

1. First, go through a LoginMethodFilter…
2. Then, go through an AuthenticationFilter…
3. Then, go through an AuthorizationFilter…
4. Finally, hit your servlet.

This concept is called *FilterChain* and the last method call in your filter above is actually delegating to that very chain:

```
chain.doFilter(request, response);
```

With such a filter (chain) you can basically handle every authentication or authorization problem there is in your application, without needing to change your actual application implementation (think: your @RestControllers / @Controllers).

Armed with that knowledge, let’s find out how Spring Security makes use of this filter magic.

FilterChain & Security Configuration DSL

We’ll start covering Spring Security a bit unconventionally, by going in the reverse direction from the previous chapter, starting with Spring Security’s FilterChain.

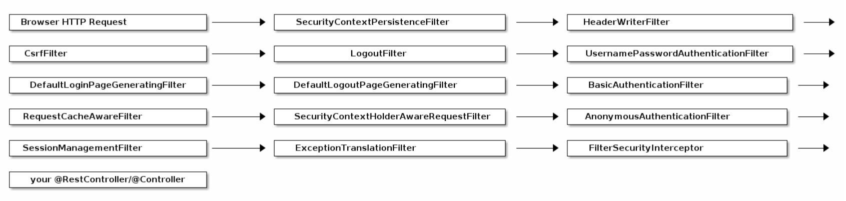
Spring’s DefaultSecurityFilterChain

Let’s assume you set up Spring Security correctly and then boot up your web application. You’ll see the following log message:

```
2020-02-25 10:24:27.875  INFO 11116 --- [
main] o.s.s.web.DefaultSecurityFilterChain     :
Creating filter chain: any request,
[org.springframework.security.web.context.request.async.We

org.springframework.security.web.context.SecurityContextPe

org.springframework.security.web.header.HeaderWriterFilter

org.springframework.security.web.csrf.CsrfFilter@51c65a43,

org.springframework.security.web.authentication.logout.Log

org.springframework.security.web.authentication.UsernamePa

org.springframework.security.web.authentication.ui.Default

org.springframework.security.web.authentication.ui.Default

org.springframework.security.web.authentication.www.BasicA

org.springframework.security.web.savedrequest.RequestCache

org.springframework.security.web.servletapi.SecurityContex

org.springframework.security.web.authentication.AnonymousA

org.springframework.security.web.session.SessionManagement

org.springframework.security.web.access.ExceptionTranslati

org.springframework.security.web.access.intercept.FilterSe
```

If you expand that one line into a list, it looks like Spring Security does not just install *one* filter, instead it installs a whole filter chain consisting of 15 (!) different filters.

So, when an HTTPRequest comes in, it will go through *all* these 15 filters, before your request finally hits your @RestControllers. The order is important, too, starting at the top of that list and going down to the bottom.


Browser HTTP Request
SecurityContextPersistenceFilter
HeaderWriterFilter
CsrfFilter
LogoutFilter
UsernamePasswordAuthenticationFilter
DefaultLoginPageGeneratingFilter
DefaultLogoutPageGeneratingFilter
BasicAuthenticationFilter
RequestCacheAwareFilter
SecurityContextHolderAwareRequestFilter
AnonymousAuthenticationFilter
SessionManagementFilter
ExceptionTranslationFilter
FilterSecurityInterceptor
your @RestController/@Controller

Analyzing Spring’s FilterChain

It would go too far to have a detailed look at every filter of this chain, but here’s the explanations for a few of those filters. Feel free to look at Spring Security’s source code to understand the other filters.

- BasicAuthenticationFilter: Tries to find a Basic Auth HTTP Header on the request and if found, tries to authenticate the user with the header’s username and password.

- UsernamePasswordAuthenticationFilter: Tries to find a username/password request parameter/POST body and if found, tries to authenticate the user with those values.
- DefaultLoginPageGeneratingFilter: Generates a login page for you, if you don’t explicitly disable that feature. THIS filter is why you get a default login page when enabling Spring Security.
- DefaultLogoutPageGeneratingFilter: Generates a logout page for you, if you don’t explicitly disable that feature.
- FilterSecurityInterceptor: Does your authorization.

So with these couple of filters, Spring Security provides you a login/logout page, as well as the ability to login with Basic Auth or Form Logins, as well as a couple of additional goodies like the CsrfFilter, that we are going to have a look at later.

Half-Time Break: Those filters, for a large part, *are* Spring Security. Not more, not less. They do all the work. What’s left for you is to *configure* how they do their work, i.e. which URLs to protect, which to ignore and what database tables to use for authentication.

Hence, we need to have a look at how to configure Spring Security, next.

How to configure Spring Security: WebSecurityConfigurerAdapter

With the latest Spring Security and/or Spring Boot versions, the way to configure Spring Security is by having a class that:

1. Is annotated with @EnableWebSecurity.
2. Extends WebSecurityConfigurer, which basically offers you a configuration DSL/methods. With those methods, you can specify what URIs in your application to protect or what exploit protections to enable/disable.

Here’s what a typical WebSecurityConfigurerAdapter looks like:

```
@Configuration
@EnableWebSecurity // (1)
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter { // (1)

  @Override
  protected void configure(HttpSecurity http) throws
Exception {  // (2)
      http
        .authorizeRequests()
          .antMatchers("/", "/home").permitAll() // (3)
          .anyRequest().authenticated() // (4)
          .and()
       .formLogin() // (5)
         .loginPage("/login") // (5)
         .permitAll()
         .and()
      .logout() // (6)
        .permitAll()
        .and()
      .httpBasic(); // (7)
  }
}
```

1. A normal Spring @Configuration with the @EnableWebSecurity annotation, extending from WebSecurityConfigurerAdapter.
2. By overriding the adapter's configure(HttpSecurity) method, you get a nice little DSL with which you can configure your FilterChain.
3. All requests going to `/` and `/home` are allowed (permitted) - the user does *not* have to authenticate. You are using an antMatcher, which means you could have also used wildcards (*, **, ?) in the string.
4. Any other request needs the user to be authenticated *first*, i.e. the user needs to login.
5. You are allowing form login (username/password in a form), with a custom loginPage (`/login`, i.e. not Spring Security's auto-generated one). Anyone should be able to access the login page, without having to log in first (permitAll; otherwise we would have a Catch-22!).
6. The same goes for the logout page
7. On top of that, you are also allowing Basic Auth, i.e. sending in an HTTP Basic Auth Header to authenticate.

How to use Spring Security's configure DSL

It takes some time getting used to that DSL, but you'll find more examples in the FAQ section: AntMatchers: Common Examples.

What is important for now, is that *THIS* `configure` method is where you specify:

1. What URLs to protect (authenticated()) and which ones are allowed (permitAll()).

2. Which authentication methods are allowed (formLogin(), httpBasic()) and how they are configured.

3. In short: your application's complete security configuration.

Note: You wouldn't have needed to immediately override the adapter's configure method, because it comes with a pretty reasonable implementation - by default. This is what it looks like:

```
public abstract class WebSecurityConfigurerAdapter
implements
                WebSecurityConfigurer<WebSecurity> {

    protected void configure(HttpSecurity http) throws
Exception {
            http
                .authorizeRequests()
                    .anyRequest().authenticated()  //
(1)
                    .and()
                .formLogin().and()   // (2)
                .httpBasic();  // (3)
        }
}
```

1. To access *any* URI (*anyRequest()*) on your application, you need to authenticate (authenticated()).

2. Form Login (*formLogin()*) with default settings is enabled.

3. As is HTTP Basic authentication (*httpBasic()*).

This default configuration is why your application is on lock-down, as soon as you add Spring Security to it. Simple, isn't it?

Summary: WebSecurityConfigurerAdapter's DSL configuration

We learned that Spring Security consists of a couple of filters that you configure with a WebSecurityConfigurerAdapter @Configuration class.

But there's one crucial piece missing. Let's take Spring's BasicAuthFilter for example. It can extract a username/password from an HTTP Basic Auth header, but what does it *authenticate* these credentials against?

This naturally leads us to the question of how authentication works with Spring Security.

Authentication with Spring Security

When it comes to authentication and Spring Security you have roughly three scenarios:

1. The default: You *can* access the (hashed) password of the user, because you have his details (username, password) saved in e.g. a database table.

2. Less common: You *cannot* access the (hashed) password of the user. This is the case if your users and passwords are stored *somewhere* else, like in a 3rd party identity management product offering REST services for authentication. Think: Atlassian Crowd.

3. Also popular: You want to use OAuth2 or "Login with Google/Twitter/etc." (OpenID), likely in combination with JWT. Then none of the following applies and you should go straight to the OAuth2 chapter.

Note: Depending on your scenario, you need to specify different @Beans to get Spring Security working, otherwise you'll end up getting pretty confusing exceptions (like a NullPointerException if you forgot to specify the PasswordEncoder). Keep that in mind.

Let's have a look at the top two scenarios.

1. UserDetailsService: Having access to the user's password

Imagine you have a database table where you store your users. It has a couple of columns, but most importantly it has a username and password column, where you store the user's hashed(!) password.

```
create table users (id int auto_increment primary key,
username varchar(255), password varchar(255));
```

In this case Spring Security needs you to define two beans to get authentication up and running.

1. A UserDetailsService.

2. A PasswordEncoder.

Specifying a UserDetailsService is as simple as this:

```
@Bean
public UserDetailsService userDetailsService() {
    return new MyDatabaseUserDetailsService(); // (1)
}
```

1. MyDatabaseUserDetailsService implements UserDetailsService, a very simple interface, which consists of one method returning a UserDetails object:

```java
public class MyDatabaseUserDetailsService implements
UserDetailsService {

        UserDetails loadUserByUsername(String username)
throws UsernameNotFoundException { // (1)
         // 1. Load the user from the users table by
username. If not found, throw
UsernameNotFoundException.
         // 2. Convert/wrap the user to a UserDetails
object and return it.
        return someUserDetails;
    }
}

public interface UserDetails extends Serializable { //
(2)

    String getUsername();

    String getPassword();

    // <3> more methods:
    // isAccountNonExpired,isAccountNonLocked,
    // isCredentialsNonExpired,isEnabled
}
```

1. A UserDetailsService loads UserDetails via the user's username. Note that the method takes only one parameter: username (not the password).

2. The UserDetails interface has methods to get the (hashed!) password and one to get the username.

3. UserDetails has even more methods, like is the account active or blocked, have the credentials expired or what permissions the user has - but we won't cover them here.

So you can either implement these interfaces yourself, like we did above, or use existing ones that Spring Security provides.

Off-The-Shelf Implementations

Just a quick note: You can always implement the UserDetailsService and UserDetails interfaces yourself.

But, you'll also find off-the-shelf implementations by Spring Security that you can use/configure/extend/override instead.

1. JdbcUserDetailsManager, which is a JDBC(database)-based UserDetailsService. You can configure it to match your *user* table/column structure.

2. InMemoryUserDetailsManager, which keeps all userdetails in-memory and is great for testing.

3. org.springframework.security.core.userdetail.User, which is a sensible, default UserDetails implementation that you could use. That would mean potentially mapping/copying between your

entities/database tables and this user class. Alternatively, you could simply make your entities implement the UserDetails interface.

Full UserDetails Workflow: HTTP Basic Authentication

Now think back to your HTTP Basic Authentication, that means you are securing your application with Spring Security and Basic Auth. This is what happens when you specify a UserDetailsService and try to login:

1. Extract the username/password combination from the HTTP Basic Auth header in a filter. You don't have to do anything for that, it will happen under the hood.

2. Call *your* MyDatabaseUserDetailsService to load the corresponding user from the database, wrapped as a UserDetails object, which exposes the user's hashed password.

3. Take the extracted password from the HTTP Basic Auth header, hash it *automatically* and compare it with the hashed password from your UserDetails object. If both match, the user is successfully authenticated.

That's all there is to it. But hold on, *how* does Spring Security hash the password from the client (step 3)? With what algorithm?

PasswordEncoders

Spring Security cannot magically guess your preferred password hashing algorithm. That's why you need to specify another @Bean, a *PasswordEncoder*. If you want to, say, use the BCrypt password hashing function (Spring Security's default) for *all your passwords*, you would specify this @Bean in your SecurityConfig.

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

What if you have *multiple* password hashing algorithms, because you have some legacy users whose passwords were stored with MD5 (don't do this), and newer ones with Bcrypt or even a third algorithm like SHA-256? Then you would use the following encoder:

```
@Bean
public PasswordEncoder passwordEncoder() {
    return
PasswordEncoderFactories.createDelegatingPasswordEncoder(

}
```

How does this delegating encoder work? It will look at the UserDetail's hashed password (coming from e.g. your database table), which now has to start with a `{prefix}`. That prefix, is your hashing method! Your database table would then look like this:

username	password
john@doe.com	{bcrypt}$2y$12$6t86Rpr3llMANhCUt26oUen2WhvXr/A89Xo9zJion8W7gWgZ/
my@user.com	{sha256}5ffa39f5757a0dad5dfada519d02c6b71b61ab1df51b4ed1f3beed6abe0ff5

Table 1. Users Table

Spring Security will:

1. Read in those passwords and strip off the prefix ({bcrypt} or {sha256}).
2. Depending on the prefix value, use the correct PasswordEncoder (i.e. a BCryptEncoder, or a SHA256Encoder)
3. Hash the incoming, raw password with that PasswordEncoder and compare it with the stored one.

That's all there is to PasswordEncoders.

Summary: Having access to the user's password

The takeaway for this section is: if you are using Spring Security and have access to the user's password, then:

1. Specify a UserDetailsService. Either a custom implementation or use and configure one that Spring Security offers.
2. Specify a PasswordEncoder.

That is Spring Security authentication in a nutshell.

2. AuthenticationProvider: Not having access to the user's password

Now, imagine that you are using Atlassian Crowd for centralized identity management. That means all your users and passwords for all your applications are stored in Atlassian Crowd and not in your database table anymore.

This has two implications:

1. You do *not have* the user passwords anymore in your application, as you cannot ask Crowd to just give you those passwords.
2. You do, however, have a REST API that you can login against, with your username and password. (A POST request to the `/rest/usermanagement/1/authentication` REST endpoint).

If that is the case, you cannot use a UserDetailsService anymore, instead you need to implement and provide an AuthenticationProvider @Bean.

```
    @Bean
    public AuthenticationProvider
authenticationProvider() {
        return new
AtlassianCrowdAuthenticationProvider();
    }
```

An AuthenticationProvider consists primarily of one method and a naive implementation could look like this:

```
public class AtlassianCrowdAuthenticationProvider
implements AuthenticationProvider {

        Authentication authenticate(Authentication
authentication)  // (1)
                throws AuthenticationException {
            String username =
authentication.getPrincipal().toString(); // (1)
            String password =
authentication.getCredentials().toString(); // (1)

            User user =
callAtlassianCrowdRestService(username, password); //
(2)
            if (user == null) {
// (3)
                throw new
AuthenticationException("could not login");
            }
            return new
UserNamePasswordAuthenticationToken(user.getUsername(),
user.getPassword(), user.getAuthorities()); // (4)
        }
            // other method ignored
}
```

1. Compared to the UserDetails load() method, where you only had access to the username, you now have access to the complete authentication attempt, *usually* containing a username and password.

2. You can do whatever you want to authenticate the user, e.g. call a REST-service.

3. If authentication failed, you need to throw an exception.

4. If authentication succeeded, you need to return a fully initialized UsernamePasswordAuthenticationToken. It is an implementation of the Authentication interface and needs to have the field authenticated be set to true (which the constructor used above will automatically set). We'll cover authorities in the next chapter.

Full AuthenticationProvider Workflow: HTTP Basic Authentication

Now think back to your HTTP Basic Authentication, that means you are securing your application with Spring Security and Basic Auth. This is what happens when you specify an AuthenticationProvider and try to login:

1. Extract the username/password combination from the HTTP Basic Auth header in a filter. You don't have to do anything for that, it will happen under the hood.

2. Call *your* AuthenticationProvider (e.g. AtlassianCrowdAuthenticationProvider) with that username and password for you to do the authentication (e.g. REST call) yourself.

There is no password hashing or similar going on, as you are essentially delegating to a third-party to do the actual username/password check. That's AuthenticationProvider authentication in a nutshell!

Summary: AuthenticationProvider

The takeaway for this section is: if you are using Spring Security and *do not* have access to the user's password, then *implement and provide an AuthenticationProvider @Bean*.

Authorization with Spring Security

So far, we have only talked about authentication, e.g. username and password checks.

Let's now have a look at permissions, or rather *roles* and *authorities* in Spring Security speak.

What is Authorization?

Take your typical e-commerce web-shop. It likely consists of the following pieces:

- The web-shop itself. Let's assume its URL is `www.youramazinshop.com`.

- Maybe an area for callcenter agents, where they can login and see what a customer recently bought or where their parcel is. Its URL could be `www.youramazinshop.com/callcenter`.

- A separate admin area, where administrators can login and manage callcenter agents or other technical aspects (like themes, performance, etc.) of the web-shop. Its URL could be `www.youramazinshop.com/admin`.

This has the following implications, as simply authenticating users is not enough anymore:

- A customer obviously shouldn't be able to access the callcenter *or* admin area. He is only allowed to shop in the website.

- A callcenter agent shouldn't be able to access the admin area.
- Whereas an admin can access the web-shop, the callcenter area and the admin area.

Simply put, you want to allow different access for different users, depending on their *authorities* or *roles*.

What are Authorities? What are Roles?

Simple:

- An authority (in its simplest form) is just a string, it can be anything like: user, ADMIN, ROLE_ADMIN or 53cr37_r0l3.
- A role is an authority with a `ROLE_` prefix. So a role called `ADMIN` is the same as an authority called `ROLE_ADMIN`.

The distinction between roles and authorities is purely conceptual and something that often bewilders people new to Spring Security.

Why is there a distinction between roles and authorities?

Quite honestly, I've read the Spring Security documentation as well as a couple of related StackOverflow threads on this very question and I can't give you a definitive, *good* answer.

What are GrantedAuthorities? What are SimpleGrantedAuthorities?

Of course, Spring Security doesn't let you get away with *just* using Strings. There's a Java class representing your authority String, a popular one being SimpleGrantedAuthority.

```
public final class SimpleGrantedAuthority implements
GrantedAuthority {

        private final String role;

    @Override
        public String getAuthority() {
                return role;
        }
}
```

(Note: There's other authority classes as well, that let you store additional objects (e.g. the principal) alongside your string, I won't cover them here. For now, we will go with SimpleGrantedAuthority, only.)

1. UserDetailsService: Where to store and get authorities?

Assuming you are storing the users in your own application (think: UserDetailsService), you are going to have a Users table.

Now, you would simply add a column called "authorities" to it. For this article I chose a simple string column here, though it could contain multiple, comma-separated values. Alternatively I could also have a completely separate table AUTHORITIES, but for the scope of this article this will do.

Note: Referring back to What are Authorities? What are Roles?: You save *authorities*, i.e. Strings, to the database. It so happens that these authorities start with the ROLE_ prefix, so, in terms of Spring Security these authorities are *also* roles.

username	password	authorities
john@doe.com	{bcrypt}...	ROLE_ADMIN
my@callcenter.com	{sha256}...	ROLE_CALLCENTER

Table 2. Users Table With Permissions

The only thing that's left to do is to adjust your UserDetailsService to read in that authorities column.

```
public class MyDatabaseUserDetailsService implements
UserDetailsService {

  UserDetails loadUserByUsername(String username)
throws UsernameNotFoundException {
     User user = userDao.findByUsername(username);
     List<SimpleGrantedAuthority> grantedAuthorities =
user.getAuthorities().map(authority -> new
SimpleGrantedAuthority(authority)).collect(Collectors.toLi
 // (1)
     return new
org.springframework.security.core.userdetails.User(user.ge
 user.getPassword(), grantedAuthorities); // (2)
  }

}
```

1. You simply map whatever is inside your database column to a list of SimpleGrantedAuthorities. Done.
2. Again, we're using Spring Security's base implementation of UserDetails here. You could also use your own class implementing UserDetails here and might not even have to map then.

2. AuthenticationManager: Where to store and get authorities?

When the users comes from a third-party application, like Atlassian Cloud, you'll need to find out what concept they are using to support authorities. Atlassian Crowd had the concepts of "roles", but deprecated it in favour of "groups".

So, depending on the actual product you are using, you need to map this to a Spring Security authority, in your AuthenticationProvider.

```
public class AtlassianCrowdAuthenticationProvider
implements AuthenticationProvider {

    Authentication authenticate(Authentication
authentication)
            throws AuthenticationException {
        String username =
authentication.getPrincipal().toString();
        String password =
authentication.getCredentials().toString();

        atlassian.crowd.User user =
callAtlassianCrowdRestService(username, password); //
(1)
        if (user == null) {
            throw new AuthenticationException("could
not login");
        }
        return new
UserNamePasswordAuthenticationToken(user.getUsername(),
user.getPassword(),
mapToAuthorities(user.getGroups())); // (2)
    }
            // other method ignored
}
```

1. Note: This is not *actual* Atlassian Crowd code, but serves its purpose. You authenticate against a REST service and get back a JSON User object, which then gets converted to an atlassian.crowd.User object.

2. That user can be a member of one or more groups, which are assumed to be just strings here. You can then simply map these groups to Spring's "SimpleGrantedAuthority".

Revisiting WebSecurityConfigurerAdapter for Authorities

So far, we talked a lot about storing and retrieving authorities for authenticated users in Spring Security. But how do you *protect* URLs with different authorities with Spring Security's DSL? Simple:

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http)
throws Exception {
        http
          .authorizeRequests()

.antMatchers("/admin").hasAuthority("ROLE_ADMIN") //
(1)

.antMatchers("/callcenter").hasAnyAuthority("ROLE_ADMIN",
 "ROLE_CALLCENTER") // (2)
            .anyRequest().authenticated() // (3)
            .and()
         .formLogin()
           .and()
         .httpBasic();
        }
}
```

1. To access the */admin* area you (i.e. the user) need to be authenticated *AND* have the authority (a simple string) ROLE_ADMIN.
2. To access the */callcenter* area you need to be authenticated *AND* have either the authority ROLE_ADMIN *OR* ROLE_CALLCENTER.
3. For any other request, you do not need a specific role but still need to be authenticated.

Note, that the above code (1,2) is *equivalent* to the following:

```
  http
    .authorizeRequests()
      .antMatchers("/admin").hasRole("ADMIN") // (1)
      .antMatchers("/callcenter").hasAnyRole("ADMIN",
"CALLCENTER") // (2)
```

1. Instead of calling "hasAuthority", you now call "hasRole". Note: Spring Security will look for an authority called *ROLE_ADMIN* on the authenticated user.
2. Instead of calling "hasAnyAuthority", you now call "hasAnyRole". Note: Spring Security will look for an authority called *ROLE_ADMIN* or *ROLE_CALLCENTER* on the authenticated user.

hasAccess and SpEL

Last, but not least, the most powerful way to configure authorizations, is with the *access* method. It lets you specify pretty much any valid SpEL expressions.

```
  http
    .authorizeRequests()
      .antMatchers("/admin").access("hasRole('admin')
and hasIpAddress('192.168.1.0/24') and
@myCustomBean.checkAccess(authentication,request)") //
(1)
```

1. You are checking that the user has ROLE_ADMIN, with a specific IP address as well as a custom bean check.

To get a full overview of what's possible with Spring's Expression-Based Access Control, have a look at the official documentation.

Common Exploit Protections

There is a variety of common attacks that Spring Security helps you to protect against. It starts with timing attacks (i.e. Spring Security will always hash the supplied password on login, even if the user does not exist) and ends up with protections against cache control attacks, content sniffing, click jacking, cross-site scripting and more.

It is impossible to go into the details of each of these attacks in the scope of this guide. Hence, we will only look at the one protection that throws most Spring Security newbies off the most: Cross-Site-Request-Forgery.

Cross-Site-Request-Forgery: CSRF

If you are completely new to CSRF, you might want to watch this YouTube video to get up to speed with it. However, the quick takeaway is, that by default Spring Security protects any incoming POST (or PUT/DELETE/PATCH) request with a valid CSRF token.

What does that mean?

CSRF & Server-Side Rendered HTML

Imagine a bank transfer form or any form (like a login form) for that matter, that gets rendered by your @Controllers with the help of a templating technology like Thymeleaf or Freemarker.

```
<form action="/transfer" method="post">  <!-- 1 -->
  <input type="text" name="amount"/>
  <input type="text" name="routingNumber"/>
  <input type="text" name="account"/>
  <input type="submit" value="Transfer"/>
</form>
```

With Spring Security enabled, you won't be able to submit that form anymore. Because Spring Security's CSRFFilter is looking for an *additional hidden parameter* on any POST (PUT/DELETE) request: a so-called CSRF token.

It generates such a token, by default, *per HTTP session* and stores it there. And you need to make sure to inject it into any of your HTML forms.

CSRF Tokens & Thymeleaf

As Thymeleaf has good integration with Spring Security (when used together with Spring Boot), you can simply add the following snippet to any form and you'll get the token injected automatically, from the session, into your form. Even better, if you are using "th:action" for your form, Thymeleaf will *automatically* inject that hidden field for you, without having to do it manually.

```
<form action="/transfer" method="post">  <!-- 1 -->
  <input type="text" name="amount"/>
  <input type="text" name="routingNumber"/>
  <input type="text" name="account"/>
  <input type="submit" value="Transfer"/>
  <input type="hidden" name="${_csrf.parameterName}"
value="${_csrf.token}"/>
</form>

<!-- OR -->

<form th:action="/transfer" method="post">  <!-- 2 -->
  <input type="text" name="amount"/>
  <input type="text" name="routingNumber"/>
  <input type="text" name="account"/>
  <input type="submit" value="Transfer"/>
</form>
```

1. Here, we are adding the CSRF parameter manually.
2. Here, we are using Thymeleaf's form support.

Note: For more information on Thymeleaf's CSRF support, see the official documentation.

CSRF & Other Templating Libraries

I cannot cover all templating libraries in this section, but as a last resort, you can always inject the CSRFToken into any of your @Controller methods and simply add it to the model to render it in a view or access it directly as HttpServletRequest request attribute.

```
@Controller
public class MyController {
    @GetMaping("/login")
    public String login(Model model, CsrfToken token) {
        // the token will be injected automatically
        return "/templates/login";
    }
}
```

CSRF & React or Angular

Things are a bit different for a Javascript app, like a React or Angular single page app. Here's what you need to do:

1. Configure Spring Security to use a CookieCsrfTokenRepository, which will put the CSRFToken into a cookie "XSRF-TOKEN" (and send that to the browser).

2. Make your Javascript app take that cookie value, and send it as an "X-XSRF-TOKEN" *header* with every POST(/PUT/PATCH/DELETE) request.

For a full copy-and-paste React example, have a look at this great blog post: https://developer.okta.com/blog/2018/07/19/simple-crud-react-and-spring-boot.

Disabling CSRF

If you are only providing a stateless REST API where CSRF protection does not make any sense, you would completely disable CSRF protection. This is how you would do it:

```
@EnableWebSecurity
@Configuration
public class WebSecurityConfig extends
   WebSecurityConfigurerAdapter {

  @Override
  protected void configure(HttpSecurity http) throws
Exception {
    http
      .csrf().disable();
  }
}
```

OAuth2

Spring Security’s OAuth2 integration is a *complex* topic and enough for another 7,000 words, which do not fit into the scope of this article.

Update August 21st, 2020: I just published the Spring Security & OAuth2 article. Check it out!

Spring Integrations

Spring Security & Spring Framework

For most of this article, you only specified security configurations on the *web tier* of your application. You protected certain URLs with antMatcher or regexMatchers with the WebSecurityConfigurerAdapter’s DSL. That is a perfectly fine and standard approach to security.

In addition to protecting your web tier, there’s also the idea of "defense in depth". That means in addition to protecting URLs, you might want to protect your business logic itself. Think: your @Controllers, @Components, @Services or even @Repositories. In short, your Spring beans.

Method Security

That approach is called *method security* and works through annotations that you can basically put on any public method of your Spring beans. You also need to explicitly enable method security by putting the @EnableGlobalMethodSecurity annotation on your ApplicationContextConfiguration.

```
@Configuration
@EnableGlobalMethodSecurity(
  prePostEnabled = true, // (1)
  securedEnabled = true, // (2)
  jsr250Enabled = true) // (3)
public class YourSecurityConfig extends
WebSecurityConfigurerAdapter{
}
```

1. The prePostEnabled property enables support for Spring's `@PreAuthorize` and `@PostAuthorize` annotations. Support means, that Spring will ignore this annotation unless you set the flag to true.
2. The securedEnabled property enables support for the `@Secured` annotation. Support means, that Spring will ignore this annotation unless you set the flag to true.
3. The jsr250Enabled property enables support for the `@RolesAllowed` annotation. Support means, that Spring will ignore this annotation unless you set the flag to true.

What is the difference between @PreAuthorize, @Secured and @RolesAllowed?

@Secured and @RolesAllowed are basically the same, though @Secured is a Spring-specific annotation coming with the spring-security-core dependency and @RolesAllowed is a standardised annotation, living in the javax.annotation-api dependency. Both annotations take in an authority/role string as value.

@PreAuthorize/@PostAuthorize are also (newer) Spring specific annotations and more powerful than the above annotations, as they can contain not only authorities/roles, but also *any* valid SpEL expression.

Lastly, all these annotations will raise an `AccessDeniedException` if you try and access a protected method with an insufficient authority/role.

So, let's finally see these annotations in action.

```
@Service
public class SomeService {

    @Secured("ROLE_CALLCENTER") // (1)
    // == @RolesAllowed("ADMIN")
    public BankAccountInfo get(...) {

    }

    @PreAuthorize("isAnonymous()") // (2)
    // @PreAuthorize("#contact.name == principal.name")
    // @PreAuthorize("ROLE_ADMIN")
    public void trackVisit(Long id);

    }
}
```

1. As mentioned, @Secured takes an authority/role as parameter. @RolesAllowed, likewise. Note: Remember that `@RolesAllowed("ADMIN")` will check for a granted authority `ROLE_ADMIN`.

2. As mentioned, @PreAuthorize takes in authorities, but also any valid SpEL expression. For a list of common built-in security expressions like `isAnonymous()` above, as opposed to writing your own SpEL expressions, check out the official documentation.

Which annotation should I use?

This is mainly a matter of homogenity, not so much of tying yourself too much to Spring-specific APIs (an argument, that is often brought forward).

If using @Secured, stick to it and don't hop on the @RolesAllowed annotation in 28% of your other beans in an effort to *standardise*, but never fully pull through.

To start off, you can always use @Secured and switch to @PreAuthorize as soon as the need arises.

Spring Security & Spring Web MVC

As for the integration with Spring WebMVC, Spring Security allows you to do a couple of things:

1. In addition to antMatchers and regexMatchers, you can also use mvcMatchers. The difference is, that while antMatchers and regexMatchers basically match URI strings with wildcards, mvcMatchers behave *exactly* like @RequestMappings.

2. Injection of your currently authenticated principal into a @Controller/@RestController method.

3. Injection of your current session CSRFToken into a @Controller/@RestController method.

4. Correct handling of security for async request processing.

```
@Controller
public class MyController {

    @RequestMapping("/messages/inbox")
    public ModelAndView
findMessagesForUser(@AuthenticationPrincipal CustomUser
customUser, CsrfToken token) {  // (1) (2)

    // .. find messages for this user and return them
...
    }
}
```

1. @AuthenticationPrincipal will inject a principal if a user is authenticated, or null if no user is authenticated. This principal is the object coming from your UserDetailsService/AuthenticationManager!
2. Or you could inject the current session CSRFToken into each method.

If you are not using the @AuthenticationPrincipal annotation, you would have to fetch the principal yourself, through the SecurityContextHolder. A technique often seen in legacy Spring Security applications.

```
@Controller
public class MyController {

    @RequestMapping("/messages/inbox")
    public ModelAndView findMessagesForUser(CsrfToken
token) {
         SecurityContext context =
SecurityContextHolder.getContext();
         Authentication authentication =
context.getAuthentication();

         if (authentication != null &&
authentication.getPrincipal() instanceof UserDetails) {
             CustomUser customUser = (CustomUser)
authentication.getPrincipal();
             // .. find messages for this user and
return them ...
         }

         // todo
    }
}
```

Spring Security & Spring Boot

Spring Boot really only pre-configures Spring Security for you, whenever you add the *spring-boot-starter-security* dependency to your Spring Boot project.

Other than that, all security configuration is done with plain Spring Security concepts (think: WebSecurityConfigurerAdapter, authentication & authorization rules), which have nothing to do with Spring Boot, per se.

So, everything you read in this guide applies 1:1 to using Spring Security with Spring Boot. And if you do not understand plain Security, don't expect to properly understand how both technologies work together.

Spring Security & Thymeleaf

Spring Security integrates well with Thymeleaf. It offers a special Spring Security Thymeleaf dialect, which allows you to put security expressions directly into your Thymeleaf HTML templates.

```
<div sec:authorize="isAuthenticated()">
  This content is only shown to authenticated users.
</div>
<div sec:authorize="hasRole('ROLE_ADMIN')">
  This content is only shown to administrators.
</div>
<div sec:authorize="hasRole('ROLE_USER')">
  This content is only shown to users.
</div>
```

For a full and more detailed overview of how both technologies work together, have a look at the official documentation.

FAQ

What is the latest Spring Security version?

As of May 2020, that is 5.3.2.RELEASE.

Note that if you are using the Spring Security dependencies defined by Spring Boot, you might be on a slightly older Spring Security version, like 5.2.1.

Are older Spring Security versions compatible with the latest version?

Spring Security has been undergoing quite some heavy changes recently. You'll therefore need to find the migration guides for your targeted versions and work through them:

- Spring Security 3.x to 4.x → https://docs.spring.io/spring-security/site/migrate/current/3-to-4/html5/migrate-3-to-4-jc.html
- Spring Security 4.x to 5.x(< 5.3) → https://docs.spring.io/spring-security/site/docs/5.0.15.RELEASE/reference/htmlsingle/#new (not a real guide, but a what's new)
- Spring Security 5.x to 5.3 → https://docs.spring.io/spring-security/site/docs/5.3.1.RELEASE/reference/html5/#new (not a real guide, but a what's new)

What dependencies do I need to add for Spring Security to work?

Plain Spring Project

If you are working with a plain Spring project (*not* Spring Boot), you need to add the following two Maven/Gradle dependencies to your project:

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>5.2.2.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>5.2.2.RELEASE</version>
</dependency>
```

You'll also need to configure the SecurityFilterChain in your web.xml or Java config. See how to do it here.

Spring Boot Project

If you are working with a Spring Boot project, you need to add the following Maven/Gradle dependency to your project:

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

Everything else will automatically be configured for you and you can immediately start writing your WebSecurityConfigurerAdapter.

How do I programmatically access the currently authenticated user in Spring Security?

As mentioned in the article, Spring Security stores the currently authenticated user (or rather a SecurityContext) in a thread-local variable inside the SecurityContextHolder. You can access it like so:

```
SecurityContext context =
SecurityContextHolder.getContext();
Authentication authentication =
context.getAuthentication();
String username = authentication.getName();
Object principal = authentication.getPrincipal();
Collection<? extends GrantedAuthority> authorities =
authentication.getAuthorities();
```

Note, that Spring Security *by default* will set an `AnonymousAuthenticationToken` as authentication on the SecurityContextHolder, if you are not logged in. This leads to some confusion, as people would naturally expect a null value there.

AntMatchers: Common Examples

A non-sensical example displaying the most useful antMatchers (and regexMatcher/mvcMatcher) possibilities:

```
@Override
protected void configure(HttpSecurity http) throws
Exception {
    http
      .authorizeRequests()
      .antMatchers("/api/user/**", "/api/ticket/**",
"/index").hasAuthority("ROLE_USER")
      .antMatchers(HttpMethod.POST,
"/forms/**").hasAnyRole("ADMIN", "CALLCENTER")

.antMatchers("/user/**").access("@webSecurity.check(auther

}
```

How to use a custom login page with Spring Security?

```
@Override
protected void configure(HttpSecurity http) throws
Exception {
  http
      .authorizeRequests()
          .anyRequest().authenticated()
          .and()
      .formLogin()
          .loginPage("/login") // (1)
          .permitAll();
}
```

1. The URL for your custom login page. As soon as you specify this, the automatically generated login page will disappear.

How to do a programmatic login with Spring Security?

```
UserDetails principal =
userDetailsService.loadUserByUsername(username);
Authentication authentication = new
UsernamePasswordAuthenticationToken(principal,
principal.getPassword(), principal.getAuthorities());
SecurityContext context =
SecurityContextHolder.createEmptyContext();
context.setAuthentication(authentication);
```

How to disable CSRF just for certain paths?

```
@Override
    protected void configure(HttpSecurity http) throws
Exception {
        http
         .csrf().ignoringAntMatchers("/api/**");
    }
```

Fin

If you have read this far, you should now have a pretty good understanding of the complexity of the Spring Security ecosystem, even without OAuth2. To sum things up:

1. It helps if you have a basic understanding of how Spring Security's FilterChain works and what its default exploit protections are (think: CSRF).
2. Make sure to understand the difference between authentication and authorization. Also what @Beans you need to specify for specific authentication workflows.
3. Make sure you understand Spring Security's WebSecurityConfigurerAdapter's DSL as well as the annotation-based method-security.
4. Last but not least, it helps to double-check the integration Spring Security has with other frameworks and libraries, like Spring MVC or Thymeleaf.

Enough for today, as that was quite a ride, wasn't it? Thanks for reading!

Acknowledgments

A big "thank you" goes out to Patricio "Pato" Moschcovich, who not only did the proofreading for this article but also provided invaluable feedback!

Comments

Login

Add a comment

M↓ MARKDOWN COMMENT ANONYMOUSLY ADD COMMENT

Upvotes Newest Oldest

Anonymous
1 point · 4 months ago

Fantastic. The best explanation I've seen about spring security. Congratulations

Anonymous
0 points · 5 months ago

Great Article!

lochschmidt
0 points · 5 months ago

Another good introduction. Thanks!
Method security is an easy source for security issues because the security rules from the annotations are **only applied, when the object is invoked from the outside**. Maybe you want to add a hint regarding that. So it never works when annotating private methods or when calling a @Secured... method from within the same class.

Anbu Sampath
0 points · 5 months ago

As usual Amazing article with excellent under the hood working details. Looking forward for OAuth2 article.

Anonymous
0 points · 5 months ago

Gread Article. Hoping for reactive part of spring like internal working :)

Anonymous
0 points · 5 months ago

Thanks lot ! Like always very awesome explanation ! Would have been even more awesome to explain in a microservice architecture using a open connect id provider !

Cheers !

Anonymous
0 points · 5 months ago

Thank you for your article, i believe it is one of the best introductions to Spring Security out there. One important note: You write passwords should be ENCRYPTED. What you mean, I think, is that passwords should be HASHED. Encrypted passwords would be huge security fault. Thank you again.

> **Marco Behler**
> **0 points** · 5 months ago
>
> You are absolutely right, will fix this in the next revision.
>
> > **andrei**
> > **0 points** · 4 months ago
> >
> > And they should also be salted.

Anonymous
0 points · 4 months ago

This was a great guide! I am just beginning to wrap my head around spring security, but having a hard time finding current stuff!

Anonymous
0 points · 4 months ago

Fantastic. I wish I had had this explanation when I first started with spring security!

andrei
0 points · 4 months ago

Great article! I like that you also give more information (for e.g. list of filters).

From https://github.com/spring-projects/spring-security/issues/2984

> > Can someone point out why a "ROLE_" prefix is appended at all?
>
> The reason is because there is more to authorities than just roles. For example, you can have a group based authority. The ROLE_ is a way to distinguish between the type of the authority.

Anonymous
0 points · 4 months ago

Amazing article! Thanks very much for have written it.

Anonymous
0 points · 4 months ago

one of the best explanation of spring security. easy to read, understand and to the point.

Anonymous
0 points · 3 months ago

Great article. Got more than what I came here for.

Anonymous
0 points · 4 months ago

Amazing article. As other posts on this site, it's well written and full of good info. I'm excited for the OAuth2 section!

Anonymous
0 points · 3 months ago

This is one of the best articles on Spring Security. The use case driven approach helps readers to understand and use spring easily

Anonymous
0 points · 2 months ago

Thanks Marco

Anonymous
0 points · 3 months ago

Thanks Marco, this article helped me a lot to understand how Spring Security works. The concepts you introduced and the order of them make this easy to follow. Again, thank you so much.

Anonymous
0 points · 3 months ago

Best article on Spring Security. Can't wait for the OAuth one.

Anonymous
0 points · 3 months ago

I think this is the best and the clearest explanation of Spring Security I have seen. Do an article regarding OAuth2 and JWT.

Anonymous

0 points · 3 months ago

Excellent article. Very much needed. Thank you Marco!

Can't wait for OAuth2. How to use spring security in case of spring-boot REST API whose only client is front end (angular) application that doesn't need login? The security needed that no third party applications should be able to access the REST API. Thanks!

Anonymous
0 points · 38 days ago

This is just great! I'm new to spring security, and a lot of tutorials weren't clear for me clear. Mostly those are going like "put that, put that..." with no deeper/ clear explanations. Thank you Marco!

Anonymous
0 points · 2 months ago

Wonderful write up. Looking forward to OAuth2 Security

Anonymous
0 points · 59 days ago

I have never seen such a clear and simple tutorial of Spring Security, though I have read a lot about it. Most of tutorials left a mess in my mind. Thank you very much!

Anonymous
0 points · 2 months ago

Excellent Article, Looking forward for OAuth2 article!

Anonymous
0 points · 53 days ago

Best Spring Security in-depth i read, thanx! Looking forward to OAuth2 Security!

Anonymous
0 points · 49 days ago

Great article! However I added spring security dependencies into my maven project and spring filters haven't been invoked. I missed some configuration when using spring-mvc (not spring boot). It would be great if you could add this information to your post for future reference. The configuration was missing an empty class that extends from AbstractSecurityWebApplicationInitializer public class SecurityWebAppInitializer extends AbstractSecurityWebApplicationInitializer {}

and security config class should be registered in the config as well.

public void onStartup(javax.servlet.ServletContext container) { AnnotationConfigWebApplicationContext rootContext = new AnnotationConfigWebApplicationContext(); rootContext.register(BeanConfig.class, SecurityConfig.class);

```
    container.addListener(new ContextLoaderListener(rootContext));
```

... }

ddjong67
0 points · 34 days ago

Dear Marco, is there any best practice for row based authorization (if you have 1000 rows in a table / view / ... and that contains users of multiple departments but my role only allows to see users in my department .. or set of departments ... is there best practice for such? (e.g. if you have www.strava.com and you want to see only rides of friends ... how to filter such in backend ...

Anonymous
0 points · 25 days ago

Agreed, this article is incredibly useful and detailed, thank you so much!

Anonymous
0 points · 29 days ago

Thanks and very good.

Twitter follower, now.

Shared on FB

pranjalchakrabortybabu
0 points · 24 days ago

A big fan, Marco. Watched the session that you took about 7-8 months ago for Java User Group Bangladesh on Spring Boot.

I think someone who has already worked with some legacy security codes, will appreciate Spring Security and this blog. Amazing. Learned a lot, learned what to learn next.

Anonymous
0 points · 11 days ago

This is one of the best explanation of Spring Security I have had. Just what I was looking.

Anonymous
0 points · 11 days ago

What is DSL?

Powered by **Commento**