(1) configure(HttpSecurity)

# whitelist

# httpBasic

# formlogin

(2) UserDetailsService userDetailsService()

how to retrive user information

username

password

role

authorization

others

eg:

# create a userDetails

UserDetails user1 = User.builder()

.username("a")

.password("password") // password must be encoded with bean "PasswordEncoder", spring security also used that for password comparison

.roles("STUDENT") // Spring saves this role as "ROLE\_STUDENT" internally

.build();

# return new InMemoryUserDetilasManager(user1);

(3) roles and permissions (synonymous to: authorities)



ADMIN //role

{

// permissions

[

STUDENT:READ,

STUDENT:WRITE,

COURSES:READ,

COURSES:WRITE

]

}

STUDENT //role

{

// permissions

[

STUDENT:READ,

STUDENT:WRITE,

COURSES:READ

]

}

common String, collectionUtils:

-> com.google.guava; guava; 28.1-jre

Sets.newSet()

(4) role based authentication

.antMatchers("/api/\*\*").hasRole("STUDENT")

role is also a form of authority that starts with "ROLE\_"

permission is the permission name as-is

(5) authority based authentication

.antMatchers("HttpMethod.DELETE, "/management/api/\*\*").hasAuthority("COURSES:WRITE")

.antMatchers("HttpMethod.POST, "/management/api/\*\*").hasAuthority("COURSES:WRITE")

.antMatchers("HttpMethod.PUT, "/management/api/\*\*").hasAuthority("COURSES:WRITE")

.antMatchers("HttpMethod.GET, "/management/api/\*\*").hasAnyRole("ADMIN", "ADMINTRAINEE")

User.builder().authorities(Collection(? extends GrantedAuthorities))

// it's a collection of "ROLE\_\*" + permission names as-is

In the user authority definition, the role is "ROLE\_ADMIN", but when used in hasRole, it's "ADMIN"

"ROLE\_" is the distinction between role and permission. Role and permissions are both bundled inside the collection of Collection<? extends GrantedAuthority>

(6) Java enum is just a class with a bunch of pre-defined instances. All other members are the same

(7) order of the antmatchers matters

first defined first served

(8) Authority based authentication through method level annotations

# in WebSecurityConfigurerAdapter

@EnableGlobalMethodSecurity(prePostEnabled = true)

# in restController

// hasRole('ROLE\_') hasAnyRole('ROLE\_', 'ROLE\_') hasAuthority('permission') hasAnyAuthority('permission', 'permission')

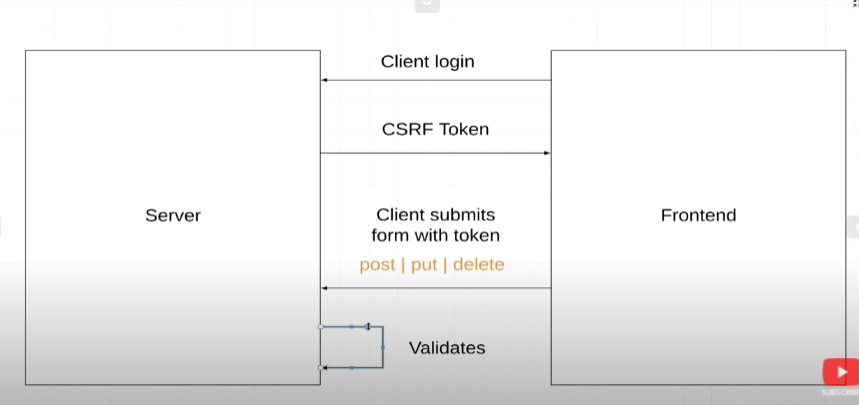
@PreAuthorize("hasAnyRole('ROLE\_ADMIN', 'ROLE\_ADMINTRAINEE')")

@PreAuthorize("hasAuthority('student:write')")

@PreAuthorize("hasAuthority('student:write')")

@PreAuthorize("hasAuthority('student:write')")

(9) CSRF: Cross site request forgery



# workflow:

CSRF stands for **Cross Site Request Forgery**

It is one kind of token that is sent with the request to prevent the attacks. In order to use the Spring Security CSRF protection, we'll first need to make sure we use the proper HTTP methods for anything that modifies the state (PATCH, POST, PUT, and DELETE – not GET).

CSRF protection with Spring CookieCsrfTokenRepository works as follows:

* The client makes a GET request to Server (Spring Boot Backend), e.g. request for the main page
* Spring sends the response for *GET* request along with Set-cookie header which contains securely generated XSRF Token
* The browser sets the cookie with XSRF Token
* While sending a state-changing request (e.g. POST) the client (might be angular) copies the cookie value to the HTTP request header
* The request is sent with both header and cookie (browser attaches the cookie automatically)
* Spring compares the header and the cookie values, if they are the same the request is accepted, otherwise, 403 is returned to the client

The method withHttpOnlyFalse allows angular to read XSRF cookie. Make sure that Angular makes XHR request with withCreddentials flag set to true.

When frontend sends post/put/delete requests to the server, X-XSRF-TOKEN needs to be specified in the header.

X-XSRF-TOKEN -> header name

XSRF-TOKEN -> cookie name

can be retrieved from cookies after a successful get request

CSRF token validation is enabled by default for all post/put/delete requests unless

.csrf().disable()

General practice principle:

Use CSRF for any request that could be processed directly from a browser by end users.

Disable CSRF protection if those endpoints are only used by non-browser users, eg: backend services.

(10) CSRF token generation

.csrf().csrfTokenRepository(CookieCsrfTokenRepository.withHttpOlyFalse())

(11) basic auth

.anyRequest()

.authenticated()

.and()

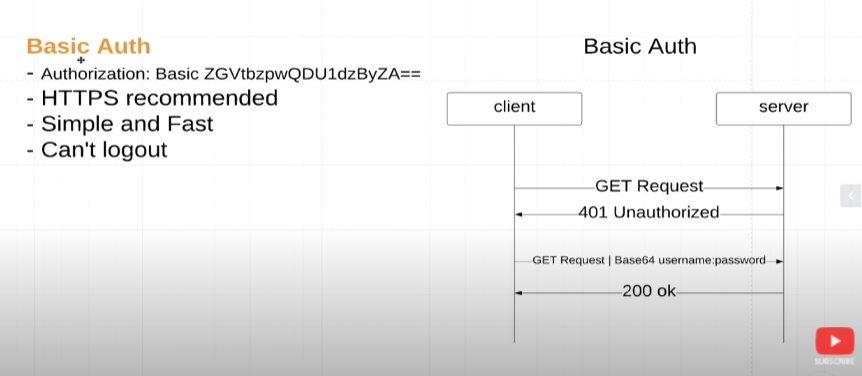
.httpBasic();

Header

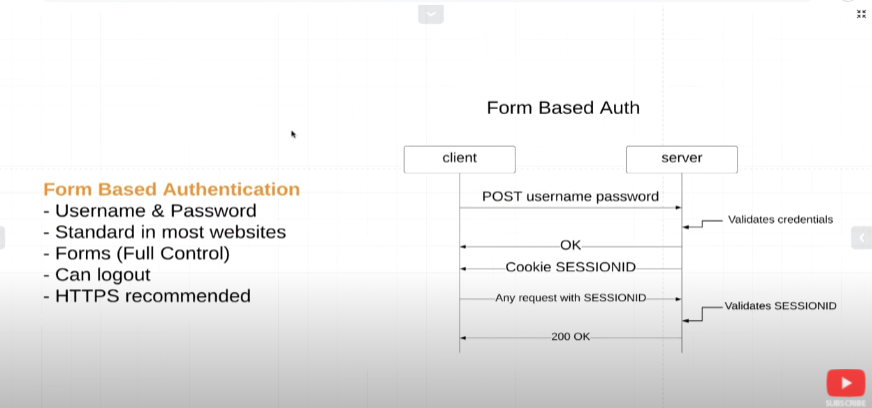
Authorization: Basic \*\*\*\*\*

Needs to be passed in each request

Can’t logout since no session was created



(12) form based auth (session based)



.anyRequest()

.authenticated()

.and()

.formLogin()

(13) Customize form login configuration

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

Thymeleaf is a templating engine regarding to html files

.anyRequest()

.authenticated()

.and()

.formLogin()

.loginPage("/login")

.permitAll()

.defaultSuccessUrl("/courses", **true**)

.passwordParameter("password")

.usernameParameter("username")

.and()

.rememberMe()

.tokenValiditySeconds((**int**) TimeUnit.***DAYS***.toSeconds(21))

.key("somethingverysecured")

.rememberMeParameter("remember-me")

.and()

.logout()

.logoutUrl("/logout")

.logoutRequestMatcher(**new** AntPathRequestMatcher("/logout", "GET")) // https://docs.spring.io/spring-security/site/docs/4.2.12.RELEASE/apidocs/org/springframework/security/config/annotation/web/configurers/LogoutConfigurer.html

.clearAuthentication(**true**)

.invalidateHttpSession(**true**)

.deleteCookies("JSESSIONID", "remember-me")

.logoutSuccessUrl("/login");

(14) Dao authentication provider

UserDetails

(username, password, grantedAuthorities (role + permissions), booleans...)

<- userDetailsService

Public UserDetails loadUserByUsername(String username)

<- userDetailsServiceDao (repository interface)

<- DaoAuthenticationProvider

@Bean

**public** DaoAuthenticationProvider daoAuthenticationProvider() {

DaoAuthenticationProvider provider = **new** DaoAuthenticationProvider();

provider.setPasswordEncoder(passwordEncoder);

provider.setUserDetailsService(ImplementedUserDetailsService);

**return** provider;

}

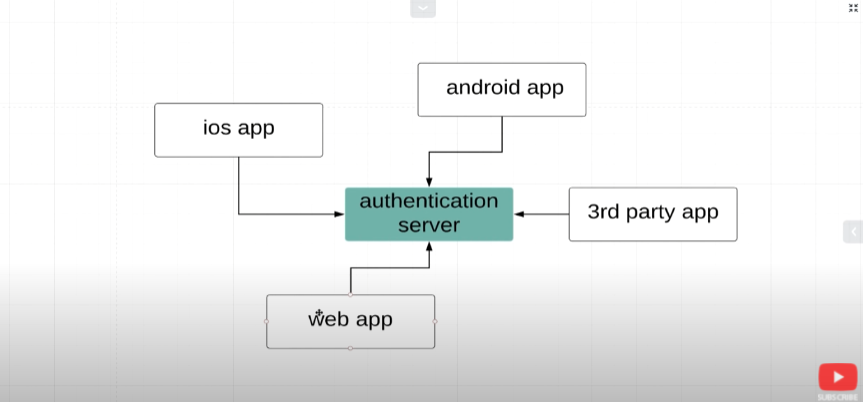
@Override

**protected** **void** configure(AuthenticationManagerBuilder auth) **throws** Exception {

auth.authenticationProvider(daoAuthenticationProvider());

}

(15) JWT (Json Web Token)



Basic authentication or form based authentication can’t do this

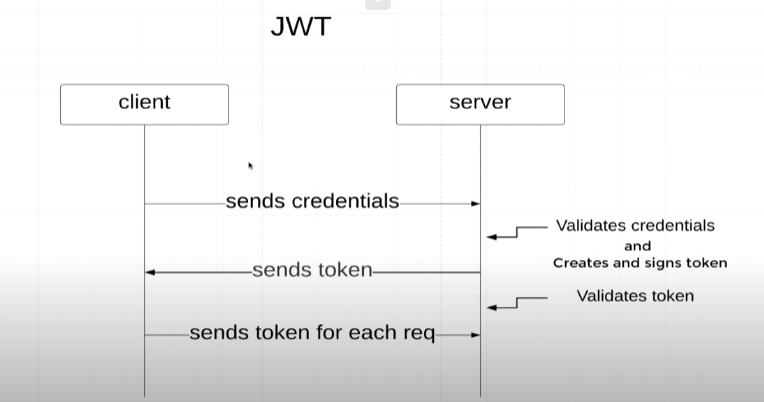
Features:

Pro:

1. Stateless (no session needed)
2. Multiple services access your application. (ldap also works)

Con:

1. If the secret key is compromised
2. No visibility to logged in users (if user is currently logged in or out)
3. Token can be stolen



One implementation library:

jwtk/jjwt

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-api</artifactId>

<version>0.10.7</version>

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-impl</artifactId>

<version>0.10.7</version>

<scope>runtime</scope>

</dependency>

<dependency>

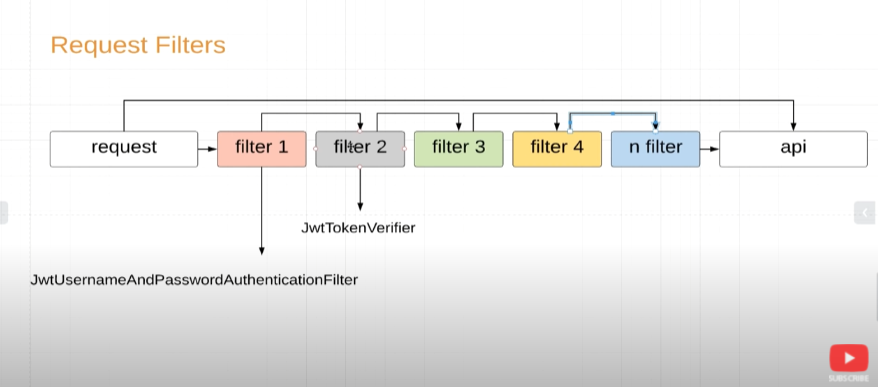
<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-jackson</artifactId>

<version>0.10.7</version>

<scope>runtime</scope>

</dependency>



Order of this filters can be specified through addFilterAfter

In each filter:

Check from request

Do something

Set response

filterChain.doFilter(request, response)

UsernamePasswordAuthenticationFilter only responds to filterProcessing Url by Post method and successfulAuthentication is the end of the request.

**Overall architecture**:

User details provider:

(passwordEncoder, UserDetailsService) -> authenticationProvider -> AuthenticationManagerBuilder -> AuthenticationManager [authenticationManager()]

3 means of authentication:

# form based (session)

# http basic

# JWT

http.csrf().disable()

.sessionManagement()

.sessionCreationPolicy(SessionCreationPolicy.***STATELESS***)

.and()

// this filter processes post /login and return jwt token

.addFilter(**new** JwtUsernameAndPasswordAuthenticationFilter(authenticationManager(), jwtConfig, secretKey))

// this filter extends OncePerRequestFilter and set security context (SecurityContextHolder.*getContext*().setAuthentication(authentication))

.addFilterAfter(**new** JwtTokenVerifier(secretKey, jwtConfig),JwtUsernameAndPasswordAuthenticationFilter.**class**)

.authorizeRequests()

.antMatchers("/", "index", "/css/\*", "/js/\*").permitAll()

.antMatchers("/api/\*\*")

// This is checked by authentication in security context, same apply to @PreAuthorize

.hasRole(***STUDENT***.name())

.anyRequest()

.authenticated();

## Authentication provider (UserDetailsService) is for authentication which returns authentication(UserDetails).

## Security context which holds authentication is for authorization (has role, has authority)