**Spring security**

Spring framework – կոմպոնենտ է նախատեսված անվտանգության համար: Օգտագործվում է ավտորիզացիոն գործողությունների համար և մոնոլիտ և միկրոսերվիսային համակարգերում:

Authentication – loggin / passwor այսինքն հետագա գործողությունները շարունակելու համար մուտք սերվիս

Authorization – Authentication – ից հետո որոշակի դեր օր՝ user, admin և այլն: Այսինքն դերերի բաժանումը դա authorization – ն է թե ինչ կարող ենք անել սերվիսում: Առանձին դերեր ավտորիզացված են մեկ իսկ այլ դերեր մեկ այլ գործողությունների համար (permission):

Security – ն աշխատում է ֆիլտրների միջոցով որոնք աշխատում են HTTP request – ի ու Spring application – ի միջև: Ֆիլտրը օբյեկտ է որը մշակում է requst – ները միջև դրանք կհասնեն controller – ին: Ֆիլտրները աշխատում են security գրադարանը ավելացնելուց հետո, boot – ում առանց configuration – ի անհրաժեշտության:

Authentication – ի ռեալիզացիա:

Կատարվում է AuthenticationProvider ինտերֆեյսի և Authentication օբյեկտի միջոցով: Պետք է ստեղծել ինտերֆեյսը ռեալիզացնող կլասս և դրա Authentication authenticate(Authentication credentials)

մեթոդի մեջ ռեալիզացնել անհրաժեշտ լոգիկան, որը կարող է ենթադրել օրինակ DB – ից տվյալների հետ համեմատում, authenticate – ի իրականացում այլ սերվերի վրա և այլն:

app – ի մեջ կարող է լինել 1 – ից ավելի AuthenticationProvider:

Մեթոդի վերադարձրած Authentication տիպի օբյեկտը իր մեջ պարունակում է principal` այն օբյեկտը որը ստանում ենք հաջող auth – ից հետո, սովորաբար user – ի օբյեկտը: Auth – ից հետո ամեն հաջորդ request – ի ժամանակ պետք չէ auth լինել քանի որ principal – ը պահվում է սեսսիայի մեջ: Սեսսիայից principal օբյեկտի load լինելու համար աշխատում է առանձին ֆիլտր: Սեսսիայի հետ աշխատանքը իրականացվում է cookie – ների միջոցով: Cookie – ն key:value ձևով պահվող տվյալներ են կոնկրետ ռեսուրսի համար, որոնք պահվում են բրաուզերում: Request – ի ժամանակ դրանք ուղարկվում են սերվերին: Սերվերը ավելացնում է բրաուզերում նոր cookie – ներ կամ փոփոխում է դրանք: ունեն նաև expires – ի աժեք և Domain, Path՝ Same Origin Policy – ի կոնցեպտում:

Spring Security – ն app – ին միացնելուց հետո մինչև request – ը կհասնի controller – ին բացվում է login page որում կարելի է մուտք գործել security – ի կողմից ավտոմատ գեներացված user – ով, default login – ը՝ user, password – ը գեներացվում է log – երում:

entity – ների հետ security – ում ընդունված է աշխատել միջնորդ կլասսի միջոցով օր` PersonDetails որը պետք է իմպլեմենտացնի security – ից ստացվող UserDetails ինտերֆեյսը: Ինտերֆեյսը ունի մեթոդներ՝ getUserName, getPssword և այլն որոնք պետք է ռեալիզացնել:

security – ն պահանջում է նաև config որը սահմանվում է համապատասխան կլասսում որտեղ կարգավորվում են auth – ը autorization - ը: Կլասսը պետք է ունենա ժառանգվի WebSecurityConfigurerAdapter կլասսից և ունենա @EnableWebSecurity անոտացիան:

custom config – ներ տալուց հետո այլևս չի գեներացվում password log – երում, այլ user – ին փորձում է վերցնել DB – ից:

կարելի է ստեղծել custom auth provider որը իմպլեմենտացնում է AuthenticationProvider – ին և authentication() - ում սահմանում նույնականացման լոգիկան, բայց քանի որ այդ լոգիկան մեծամասամբ ստանդարտ է՝ usernam passwod -ի ստուգում և Authentication – տիպի օբյեկտի վերաձարձում (իր մեջ պարունակում է principal) կարելի է աշխատել և առանց custom provider ունենալու: Լոգիկան կկատարի security – ն ավտոմատ հիմնվելով իմպլեմենտացված ինտերֆեյսների և դրանց մեթոդների անունների վրա:

Եթե auth – ը պարզ է ապա կարելի է չօգտագործել custom auth provider և լոգիկան թողնել security – ին, custom provider իմաստ ունի գրել եթե պահանջվում է ստանդարտից շեղվող բարդ auth լոգիկա:

Autorization – ը կազմակերպվում է configure (HttpSecurity http) մեթոդի միջոցով: http օբյեկտի վրա կարելի է chaining – ով սահմանել տարբեր կարգավորումներ ավտորիզացիայի կամ default login էջը փոխելու և այլն:

Օր՝ http.authorizeRequests()

.antMatchers(«path», «path»).permitAll() // հասանելի բոլորին

.anyRequest().authenticated() // auth մնացած հասցեների համար

**Password – ի ծածկագրում**

Ծածկագրման ժամանակ առանցքային հասկացությունը դա միակողմանի ֆունկցիան է one way function: Այսինքն ֆուկցիան արգումենտի համար դուրս է բերում արժեք բայց այդ արժեքի համար դուրս չի բերում արգումենտը: Կենցաղային օրիկան է միքսերը, որը արդյունքը հետ չի կարող վերադարձնել: Մաթեմատիկայում օրինակ 2 թվերի արտադրյալը հայտնի թիվ է բայց այդ արդյունքնով չի կարելի որոշել թե կոնկրետ որ թվեերի բազմապատկման արդյունքում ենք ստացել այդ թիվը, այսինքն բազմապատկումը միակողմանի ֆունկցիա է:

Օրինակ է նաև hashCode – ը string օբյեկտի համար որը ստացվում է դրա առանձին սիմվոլների հետ գործողությունների և վերջում դրանց գումարման արդյունքում: Այստեղ նույնպես արդյունքից ելելով չենք կարող ստանալ սկզբնական արժեքը:

Միակողմանի ֆունկցաիները պետք է լինեն դետեմինացված, այսինքն նույն արգումենտի համար միշտ պետք է դուրս բերեն նույն արժեքը:

Ծածկագրման տարածված ալգորիթմներից է bcrypt – ը:

Ծածկագրման համար կա առանձին Bean security config - ում`

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

Դրանից հետո injet եղած օբյեկտի վրա՝

String bCryptPasswordEncoder.encode(String password)

Որպեսզի auth – ի ժամանակ ստուգումը նույնպես տեղի ունենա ծածկագրված գաղտնաբառուվ configure մեթոդում պետք է ավելացնել passwordEncoder – ը: