## NB: gli esempio di seguito usano spring boot < 3 , e spring < 6. Nelle versioni >= spring boot 3.0 non si usa il web configurer adapter ma il filter chain

# Basic Authentication

Questo NON e’ il default per spring security.  
Per abilitarlo bisogna creare una classe @Configuration e @EnableWebSecurity che estende il WebSecurityConfigurerAdapter e fa override del configure sull’http (dove si puo’ scegliere quali url coprire etc).  
In questo modo si abilita la basic authentication, che NON usa sessione.  
Prevede che per ogni singola richiesta il cliente deve inviare username:password (messi sotto encoding base64).  
Tipicamente un pattern che si puo’ usare se si vuole customizzare la pagina di login (senza modificare e usare quella di spring) è abilitare la sessione sulla basic auth (così che si puo’ inviare una sola volta username:password per autenticarsi su tutte le richieste) ed esporre l’autenticazione su un solo url specifico (single point basic authentication with session).   
NB: per la basic auth username:password (in base 64) devono viaggiare come header parameter nella richiesta (in un header con nome **Authorization** e valore **Basic encodebase64(username:password)**).  
**Visto che nella basic auth devono viaggiare sempre user e password, conviene quindi usare https !!!**

NB: di default anche il basic auth usa il csfr token (quindi va disabilitato se non lo si vuole inviare altrimenti non ci autorizza)

# Form Based Authentication

Questo è il default per spring boot (quando si installano le dipendenze di spring security e non si configura nulla).  
Di default il form based copre tutti gli indirizzi, ed è con sessione.  
Quando l’utente non ha il cookie di sessione che indica che è loggato, se prova ad accedere ad una risorsa, viene fatto redirect sul form di login. Se logga correttamente viene salvato in sessione il cookie e quindi avrà accesso a tutte le risorse.  
NB: per la form based auth username e pw viaggiano nel body della richiesta POST (del form).  
NB2: di default il form based auth di spring boot utilizza un csrf-token per evitare i csrf-attack.  
Anche se è il default, comunque si puo’ specificare esplicitamente la form based Authentication, con le varie customizzazioni, sempre nel WebSecurityConfigurerAdapter, nel solito metodo configure (http).  
Il resto funziona uguale (gli antMatcher per le api, ruoli etc…vedi avanti).

Text

Description automatically generated  
La form based si attiva col metodo **formLogin()**.  
Si specifica dove andare a prendere la pagina di login (se la si vuole customizzare) indicando col metodo **loginPage(“/url”).permitAll()** dove url deve essere registrato su un classico **@Controller MVC** che ritorna una stringa che è il nome risolto poi dal view resolver (tipicamente si usa thymeleaf, che va messo nelle dipendenze, in maniera tale che vengano servite le pagine sotto /static/template).  
La pagina di login che si scrive, deve inviare la pw come parametro di type “password” e name “password” e type “text” per username e name “username”.  
Questo in un form classico.   
Se si vuole cambiare il nome dei parametri che viaggiano, nel configure si fa Text

Description automatically generated  
**NB: la loginPage inoltre deve essere scoperta dall’autenticazione, perche’ ovviamente deve essere accessibile (ecco perché permitAll() dopo).**E si indica poi la default landing page se successfull con **defaultSuccessfull(“/url”).  
REMEMBER-ME:** La form based authentication è di default con **SESSIONE.** Esiste un meccanismo che entra in gioco anche quando la sessione è scaduta , per mantenere l’utente collegato.  
Questo è il REMEMBER-ME.  
Sostanzialmente quando attivato , sulla form based auth di spring, con   
Logo, company name

Description automatically generated  
Se la pagina di login (customizzata) inviata un input che si chiama **remember-me** a true, questo fa si che spring stacchi un cookie aggiuntivo che invia al client, in cui viene salvato username, scadenza remember me (tipicamente 2 settimane) e codifica di questi 2 dati (fatta be con la uniqueAndSecret, che è una chiave).   
Per cambiare il nome del parametro remember-me rispetto al default, che deve viaggiare, sempre nel configure si puo’ fare  
  
Attenzione che il remember me crea seri problemi di sicurezza.   
**LOGOUT:** Tipicamente si configura come la login, su un url (che pero’ non deve essere mappato da un @Controller MVC né andare verso una pagina ovviamente) e si indica esplicitamente anche cosa ripulire (soprattutto JSESSION-ID che è per la sessione e remember-me)

Text

Description automatically generated

Attenzione: visto che il logout cambia lo stato, è buona norma usare non una richiesta get ma una post.  
Se non diversamente specificato, quando si registra il logout url con la logoutUrl(), viene registrato l’url su una chiamata GET (anche se non è una buona pratica).  
Ci sono casi in cui pero’, anche se registrato l’url con logoutUrl(“/..”) quindi con la GET, non funziona la logout. Questo è il caso in cui è abilitato il csfr-token.  
Di default quando è abilitato il csrf-token, il mapping per la logout deve essere fatta per forza in post.  
Per fare quindi un mapping sul request type esplicito si fa :  
  
in questo caso al posto di GET va messo post.

# AntMatchers

Gli antMatchers si usano per fare whitelist o altre operazioni su pattern di url.

# UserDetails

Di default spring security usa un db in memoria per gli utenti.  
Le info dell’utente sono : **username**, **password** (encoded), **roles**, **authorities** (opzionali).  
   
NB: visto che le pw devono essere encoded, bisogna definire un password encoder a livello di ambiente spring (un semplice @Configuration che spara un @Bean PasswordEncoder, tipicamente un BCryptPasswordEncoder)

Quello che si fa, quando si vuole usare un DB per mantenere users e pws, è creare tutta l’infrastruttura DB (entity, repos) oppure ci si aggancia ad un servizio esterno nostro tipo LDAP.  
A spring tutto questo non interessa.  
Tutto quello che interessa a Spring è avare un servizio che fa da anello di unione tra il mondo degli utenti e permessi/ruoli/autorizzazioni (leggi dopo che non c’e’ differenza) di spring , modellato usando l’interfaccia UserDetails (implementata ad es da User) che serve per definire gli utenti nel modello spring , e i loro permessi, modellate dalla classe SimpleGrantedAuthority, e tutto quello che è il mondo degli utenti/permessi nel modello di business (quindi ad es il nostro database, il nostro LDAP etc).  
Quello che è l’anello di congiunzione è proprio un nostro bean che implementa l’interfaccia UserDetailsService, e definisce il metodo per ottenere gli utenti (del modello Spring, quindi UserDetails/User) dato un username. Questi utenti dovranno essere decorati con la password (cryptata in BCrypt ad es se stiamo usando un BCrypt per criptare/decriptare) e con associata la lista di SimpleGrantedAuthority (che creeremo dai permission modellati al db ad es nella tabella NxM dei permessi per gli utenti)  
Una volta definita tale classe, bisogna specificare nel **WebSecurityConfigurerAdapter** che usiamo tale nostro servizio per l’anello di congiunzione.  
Nello stesso punto inoltre definiremo il nostro password encrypter.  
Questo si fa col metodo configureGlobal:  
Text

Description automatically generated  
  
  
E di seguito c’e’ l’esempio dell’implementazione del nostro UserDetailsService  
Text

Description automatically generated

# Roles and Authorities(permissions)

Attenzione alla confusione!  
Dal punto di vista di spring esiste solo il concetto di **permission** (ad es con la classe **ApplicationUserPermission**) e del suo wrapper che lo contiene , l’authority (classe ad es **SimpleGrantedAuthority** che implementa interfaccia **GrantedAuthority**). Non c’e’ una differenza tra ruolo e permission/authorization se non nel modo per cui alcuni metodi (es hasRole) mettono loro il prefisso ROLE\_ a quello che prendono in input, altri no.  
In spring si puo’ bloccare un’api, nel configure(http..) del WebSecurityConfigurerAdapter, col metodo hasRole(“NOMERUOLO), oppure sempre nello stesso metodo con il hasAuthority(ROLE\_NOMERUOLO).  
Text

Description automatically generated  
  
  
  
  
Oppure sul metodo del controller con @PreAuthorize(ROLE\_NOMERUOLO) (per fare questo bisogna attivare quest’annotazione con @@EnableGlobalMethodSecurity(prePostEnabled = true) //to enable @preAuthorize on Apis )Text

Description automatically generated

Il fatto che un ruolo sia un insieme di permessi, è semplicemente una cosa che tipicamente viene modellata a livello di business (creando una enum di ruolo che contiene enum di permessi/authorization) e siamo noi programmatori a scegliere la grana.  
Ma dal punto di vista di SpringSecurity non c’e’ questa differenza (se non nel modo in cui mette o non mette il prefisso ROLE\_).

# CSRF Token

Il CSRF Token serve per proteggersi ogni qual volta si usa un approccio STATEFULL (con i cookie di sessione).

Se non disabilitiamo il CSFR (token) su spring boot, nel configure del websecurity configurer adapter,  
spring security lo vuole.  
Tipicamente il giro è che il client deve mandare una richiesta GET al server. Spring security risponde  
al client inviandogli tra le varie cose un cookie chiamato XSRF-TOKEN.  
Questo è il token.  
Questo token deve essere rimandato con le successive request (POST/PUT/DELETE) del client, nell’attributo X-XSRF-TOKEN affinchè si possa supererare la validazione.

# JWT Based Auth

**NB: la demo a cui si fa riferimento usa Spring Boot 3.0 (quindi invece del WebConfigurerAdapter useremo la nuova gestione col FilterChain ma cambia poco)**Tipicamente quando si usa JWT si usa approccio STATELESS (senza sessione quindi).  
Visto che non si usa un cookie di sessione, non c’e’ bisogno di usare CSRF-Token per proteggersi .  
Il JWT tipicamente viene staccato dal server quando l’utente è loggato con successo, viene inviato al client, e il client lo usa per decorare nell’header (AUTHORIZATION header, di tipo Bearer Token) le successive chiamate.  
Un JWT è fatto di 3 parti:  
  
**-Header**(JWT) **: BASE\_64**(tipo del token (JWT) e Algoritmo usato per la signature)-> NB: è in chiaro, anche se in base64 ma NON CRIPTATA  
**-Body**: **BASE\_64**(**claims: i claims sono una lista di campi in una mappa chiave valore, tipicamente username, nome e cognome, e data scadenza e data issue del token**) ->NB: è in chiaro, in base 64 si ma è decodificabile da tutti e quindi non deve contenere dati sensibili. I claims servono soltanto al server per ritrovare i dati dell’utente (se il JWT che gli sta mandando è valido)  
**-Signature: Viene applicato algoritmo di encript (usando il BASE64 della chiave segreta del server) alla concatenazione dei Base64 dell’HEADER JWT e del BODY (usando il . come concatenazione)**

Quindi ad esempio se si usa HMACSHA256 come algoritmo di encription, e l’utente che si è loggato correttamente (al primo login bisogna sempre usare username e pw classici, che sia tramite form based o based auth o come si vuole) ha   
  
nome utente: sitodskij  
nome : davide  
cognome : sito  
  
il server crea **HEADER JWT : fa BASE64 encoding di un oggetto contenete type e algorithm usati.**il server crea un payload con i claims a scelta del programmatore, es

{username: sitodskij, name: davide, surname: sito, subject:sitodskij} (il subject tipicamente è l’username) e **fa BASE54 encoding di questo.**Il server concatena BASE64(HEADER).BASE64(BODY)  
  
Il server prende la sua chiave privata, ne fa BASE64 encoding, e la usa per criptare, usando HMACSHA256 questo -> BASE64(HEADER).BASE64(BODY) ottenendo la **SIGNATURE DEL JWT**  
  
Alla fine il jwt sarà composto da :  
  
JWT\_TOKEN = BASE64(HEADER).BASE64(BODY).SIGNATURE-JWT  
  
E lo manda al client.  
  
Il client alle successive chiamate, per essere autenticato dovrà mandare nell’header http con nome **AUTHORIZATION (di tipo Bearer Token) questo:  
  
Bearer JWT\_TOKEN  
  
Il Backend lo prende, usa il Base64 della sua chiave privata per verificare se la signature corrisponde alla concatenazione di Header e Payload, e se è così allora vuol dire che quel token l’ha realmente staccato lui.  
Controlla inoltre che non sia scaduto.  
Se è così la chiamata dell’utente viene considerata autorizzata, altrimenti ritorna 401.  
  
Come viene implementato tutto questo in spring security ?  
Uno degli approcci (demo repository) consiste nel   
  
-Disabilitare CSRF-Token (non serve visto che siamo stateless)  
-Settare stateless  
-Tenere protetti tutti gli endpoint ad eccezione di /login e /register.  
-Sotto /register l’utente manda (in POST) in https i dati sensibili username, pw, nome , cognome etc, e il BE crea utente.  
-Sotto /login, il client manda username e pw in HTTPS, il server controlla che quell’utente esiste e abbia quella pw, e in caso sia così stacca il jwt e lo manda nel dto al client.  
-Si crea quindi un filtro che parte come prima cosa nelle chiamate http del client.  
-Questo filtro controlla se esiste un header di tipo AUTHORIZATION e se è composto di 2 parti, la cui prima deve essere Bearer (la seconda sarà quindi il JWT).  
In tal caso prende il JWT e verifica se è corretto (usando il base 64 della sua chiave privata).  
Se è così allora setta esplicitamente nell SecurityContextHolder context l’authentication.  
Settare l’authentication nel security Context holder context vuol dire settare un wrapper (**UsernamePasswordAuthenticationToken)  **che contiene l’user dto estratto usando i dati del claim una volta che il jwt risulta valido ed è stato decriptato.  
-In tutti gli altri casi il filtro continua.  
  
Si veda la demo nel repository per ulteriori dettagli.**