# 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)**)

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.

# 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\_).