* Spring Security foloseste AOP pentru securrity metohod level
* Deci, el foloseste proxy, si daca vom apela o metoda din alta metoda ce are tot anotatii de securitate, este evident ca acele anotatii nu vor merge, asa cum metoda a fost apelata nu din proxy, ci direct din obiect

Pentru a permite folosirea anotatiilor la nivel de metoda, folosim:

@EnableMethodSecurity(  
 securedEnabled = true,  
 prePostEnabled = true,  
 jsr250Enabled = true)

**prePostEnabled este true in mod default!**

**securedEnbled –** activeaza anotatia @Secured

**prePostEnabled** – activeaza @PreAuthorize si @PostAuthorize, ele sunt activate default

**jsr250Enabled** – activeaza @RolesAllowed

**@Secured**

* Este folosita pentru a permite accestul la un role sau la o lista de roluri
* @GetMapping  
  @Secured({"ROLE\_Manager","ROLE\_Admin"})  
  public User user(){  
   return new User("Test","test");  
  }

Sau, putem pune si un singur role.

**The @Secured annotation doesn't support Spring Expression Language (SpEL).**

**@RolesAllowed**

@RestController  
@RequestMapping("/api")  
public class REST {  
 @GetMapping  
 @RolesAllowed({"Manager","Admin"})  
 public User user(){  
 return new User("Test","test");  
 }

}

@RollesAllowed e exact ca @Secured, dar sunt si ceva diferente:

- @RollesAllowed pune automat in fara la role “ROLE\_”, si daca punem noi, vom avea eroare, dar @Secured nu pune.

- RollesAllowed este de la Java, @Secured de la Spring Security

**@PreAuthorize si @PostAuthorize**

* Ambele suporta Spring Expression Language (SpEL)
* Nu sunt facute doar pentru a verifica rolul, de aceea folosesc expresii gen “hasRole(‘ROLE\_ADMIN’)”
* **@PreAuthorize** – verifica expresia inainte de a intra in metoda
* **@PostAuthorize** – verifica expresia dupa ce metoda e executata si poate influenta rezultatul
* **@PreAuthorize(“hasRole(‘VIEWER')”)** has the same meaning as **@Secured(“ROLE\_VIEWER”),**
* Putem folosi si oricare alte metode de validare, ca hasAnyRole(), isAuthenticated() etc. si operatori, ca ||,OR etc.
* **Aceste anotatii mai ofera un operator, si anume #Parametru.** Acest operator ne permite sa accesam parametrul trimis metodei, si sa facem verificari cu el:
* @PreAuthorize("hasRole('Manager') && #user.firstName.equals('Alex')")  
  @PostMapping  
  public String add(@RequestBody User user){  
   System.*out*.println(user);  
    
   return "Success";  
  }



**Atentie! Cand lucram cu @Post, @Delete, @Put, trebuie sa dezactivam CSRF la spring configuration, de altfel vom primi mereu acces interzis:**

@Bean  
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {  
 return http  
 .csrf(csrf -> csrf.disable())  
 .httpBasic(Customizer.*withDefaults*())  
 .build();  
}

* **@PostAuthorize** este exact ca @PreAuthorize, unica diferenta fiind ca @PostAuthorize va face verificarea dupa ce metoda va fi executata abea,si deci poate influenta asupra rezultatului returnat deja. De ex, daca avem:
* @PostAuthorize("hasRole('Admin') && #user.firstName.equals('Alex')")  
  @PostMapping  
  public String add(@RequestBody User user){  
   System.*out*.println(user);  
    
   return "Success";  
  }

si facem un post request asa:

{

    "firstName" : "Alex",

    "lastName" : "Goncear"

}

dar daca userul are rolul de Manager de ex, ei bine el va primi acces interzis, dar ideea e ca metoda add() se va executa, insa abea dupa executie el va primi acces interzis. Deci, in consola intai se va arata userul, dar rezultatul din return nu va mai ajunge la user.

* @PostAuthorize ne ofera posibilitatea de a accesa obiectul returnat prin **returnObject**:
* @PostAuthorize("returnObject.equals('Success')")  
  @PostMapping  
  public String add(@RequestBody User user){  
   System.*out*.println(user);  
    
   return "ERROR";  
  }

**@PreFilter and @PostFilter**

* @Prefilter si @PostFilter ne permit sa filtrtram datele.
* @PreFilter permite filtrarea datelor dintr-o colectie ce este argument in metoda:
* @PreFilter("filterObject.lastName.equals('Mititiuc')")  
  @PostMapping  
  public String add(@RequestBody List<User> user){  
   System.*out*.println(user);  
    
   return "Success";  
  }



Si daca vom trimtie un asa post:

[{

    "firstName" : "Eduard",

    "lastName" : "Mititiuc"

},

{

    "firstName" : "Luminita",

    "lastName" : "Mititiuc"

},

{

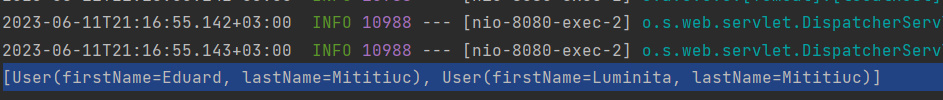
    "firstName" : "Alex",

    "lastName" : "Goncear"

}

]

Vedem rezultatul filtrat:



Atentie! Pentru a face referinta la colectia ca argument, folosim filterObject

* Daca se intampla ca sunt mai multe argumente colectii, folosim argumentul filterTarget = “argument” pentru a ne referi la cel dorit:

@PreFilter (value = "filterObject != authentication.principal.username", filterTarget = "usernames")



**public** String **joinUsernamesAndRoles**( List<String> usernames, List<String> roles) {

**return** usernames.stream().collect(Collectors.joining(";")) + ":" + roles.stream().collect(Collectors.joining(";")); }

* @PostFilter se ocupa de filtrarea la colectia din return:
* @PostFilter("filterObject.lastName.equals('MITITIUC')")  
  @PostMapping  
  public List<User> add(@RequestBody List<User> user){  
   user.forEach(u -> {  
   u.setFirstName(u.getFirstName().toUpperCase());  
   u.setLastName(u.getLastName().toUpperCase());  
   });  
   System.*out*.println(user);  
    
   return user;  
  }

Si dam post:

[{

    "firstName" : "Eduard",

    "lastName" : "Mititiuc"

},

{

    "firstName" : "Luminita",

    "lastName" : "Mititiuc"

},

{

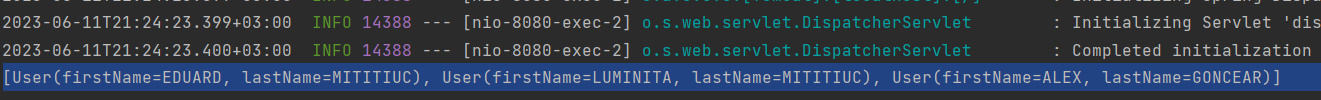
    "firstName" : "Alex",

    "lastName" : "Goncear"

}

]

Observam ca in metoda datele exact asa au venit ca din JSON:



Deci, s-a returnat o lista filtrata:

[

    {

        "firstName": "EDUARD",

        "lastName": "MITITIUC"

    },

    {

        "firstName": "LUMINITA",

        "lastName": "MITITIUC"

    }

]

**Custom Annotation**

* Putem pune aceste anotatii si la nivel de anotatie chiar, si asa sa cream o anotatie proprie. De ex, daca vrem sa oferim acces la mai multe resurse la Manager si Member, va trebui sa repetam mult code. E mai usor sa cream o anotatie ce deodata face asta:
* @Target(ElementType.*METHOD*)  
  @Retention(RetentionPolicy.*RUNTIME*)  
  @PreAuthorize("hasAnyRole('Manager','Admin')")  
  public @interface ManagerOrAdmin {  
  }

Si

@ManagerOrAdmin  
@GetMapping  
public User user(){  
 return new User("Test","test");  
}

**Annotation class Level**

Daca folosim aceeasi anotatie la fiecare metoda din clasa, putem pur si simplu sa o punem la nivel de clasa si ea automat va fi pusa la fiecare metoda din clasa:

@Service

@PreAuthorize("hasRole('ROLE\_ADMIN')")

**public** **class** **SystemService** {

**public** String **getSystemYear**(){ //...

}

**public** String **getSystemDate**(){

//...

}

}

**Atentie la hasRole() si hasAuthority()**

* hasRole(“ADMIN”) – mereu va pune in fata “ROLE\_”, deci daca un enum avem ADMIN si in baza de date ADMIN, hasRole(“ADMIN”) nu va functiona! El va cauta ROLE\_ADMIN!!!
* hasAuthority(“ADMIN”) – nu va adauga nimic in fata, nici un prefix!!!