Spring security

Spring Sécurity est un framework permettant d'ajouter aux applications Spring authentification et contrôle d'accès.

Spring security ajoute au Dispatcher servlet de Spring MVC un ensemble de filtres (servlet filters) qualifié de **filter chain**.

Intégration

Ajouter les dépendances suivantes à **pom.xml** :

```
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-config</artifactId>
        </dependency>
```

Authentification par défaut

L'intégration de Spring security met en place une authentification basique par défaut, avec un utilisateur de non **user**, ayant le rôle **USER**,
dont le password s'affiche dans la console de démarrage de Spring, utilisable en mode développement :

```
Using generated security password: 90990698-6bb0-4953-8151-fe8010f2547a

This generated password is for development use only. Your security configuration must be updated before running your application in production.
```

Il est possible de modifier l'utilisateur par défaut dans **application.properties** :

```
spring.security.user.name=boris
spring.security.user.password=boris-password
spring.security.user.roles=manager
```

Configuration

La configuration se fait au travers d'une classe de configuration activant la sécurité :

```
@Configuration
@EnableWebSecurity
class WebSecurityConfig {
    @Bean
    @Throws(Exception::class)
    fun configure(http: HttpSecurity): SecurityFilterChain {
        http
            .csrf().disable().authorizeHttpRequests()
            .requestMatchers("/master/**").hasRole("manager")
            .anyRequest().authenticated()
            .and()
            .formLogin()
        return http.build()
    }

}
```

Authentification depuis une BDD

Entity

Créer une entity **User** pour gérer les utilisateur et leur rôle :

```
@Entity
open class User() {

    constructor(username:String):this(){
    this.username=username
    }

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    open var id:Int=0

    @Column(length = 65, nullable = false)
    open lateinit var username:String

    @Column(length = 256)
    open var password:String?=null

    @Column(unique = true, length = 115)
    open var email:String?=null

    @Column(nullable = false)
    open lateinit var role:String
}
```

Créer le repository associé et ajouter une méthode permettant de rechercher un User par son username ou son email :

```
interface UserRepository : JpaRepository<User, Int> {
    @Query("SELECT u FROM User u WHERE u.username=:usernameOrEmail OR
u.email=:usernameOrEmail")
    fun findByUsernameOrEmail(usernameOrEmail: String?): User?
}
```

UserDetailsService

Créer un Service implémentant **UserDetailsService** :

Ce service retourne depuis la base de données un utilisateur recherché par son nom ou son email et retourne une instance de **UserDetails** :

```
class DbUserService:UserDetailsService {
    @Autowired
    lateinit var userRepository: UserRepository

    @Autowired
    lateinit var passwordEncoder: PasswordEncoder

    override fun loadUserByUsername(username: String?): UserDetails {
        val user= userRepository.findByUsernameOrEmail(username!!) ?: throw
UsernameNotFoundException("User not found")
        return
```

```
org.springframework.security.core.userdetails.User(user.username,user.password,
getGrantedAuthorities(user))
    }

    private fun getGrantedAuthorities(user: User): List<GrantedAuthority>? {
        val authorities: MutableList<GrantedAuthority> = ArrayList()
        authorities.add(SimpleGrantedAuthority(user.role))
        return authorities
    }

    fun encodePassword(user: User) {
        user.password=passwordEncoder.encode(user.password)
    }
}
```

Password encoder

Modifier la configuration de spring security pour définir le hashage du mot de passe (Bcrypt) :

```
@Configuration
@EnableWebSecurity
class WebSecurityConfig {

    @Bean
    @Throws(Exception::class)
    fun configure(http: HttpSecurity): SecurityFilterChain {
        ...

    @Bean
    fun userDetailsService(): UserDetailsService? {
        return DbUserService()
    }

    @Bean
    fun bCryptPasswordEncoder(): BCryptPasswordEncoder? {
        return BCryptPasswordEncoder()
    }
}
```

Création d'utilisateur

Ne pas oublier de hasher le password User avant création :

```
@Controller
class InitController {
    @Autowired
    lateinit var dbUserService: UserDetailsService

    @Autowired
    lateinit var userRepository: UserRepository
```

```
    @Autowired
    lateinit var roleRepository: RoleRepository

    @RequestMapping("/init/{username}")
    fun createUser(@PathVariable username:String):String {
        val user=User()
        user.username=username
        user.email=username.lowercase()+"@gmail.com"
        user.role="MANAGER"
        user.password="1234"
        (dbUserService as DbUserService).encodePassword(user)
        userRepository.save(user)
        return "redirect:/"
    }
}
```

Login form personnalisée

Créer un template pour le formulaire de connexion :

```
<form method="post" action="./login">
    <input type="text" name="username" placeholder="E-mail address or username">
    <input type="password" name="password" placeholder="Password">
    <button type="submit">Login</button>
</form>
```

Ajouter une route pour le **login**

```
class AppConfig  : WebMvcConfigurer {
    override fun addViewControllers(registry: ViewControllerRegistry) {
        registry.addViewController("/login").setViewName("loginForm")
    }
}
```

Modifier la configuration dans **WebSecurityConfig** :

```
@Configuration
@EnableWebSecurity
class WebSecurityConfig {

    @Bean
    @Throws(Exception::class)
    fun configure(http: HttpSecurity): SecurityFilterChain {
        ...
        http.formLogin().loginPage("/login")
        return http.build()
    }
```

Connexion à la console H2

Pour autoriser l'accès à la console h2 :

1. Autoriser la route vers la console **/h2-console**
2. Autoriser l'utilisation des iframes pour l'adresse locale
3. Désactiver la protection csrf

```
@Configuration
@EnableWebSecurity
class WebSecurityConfig {

    @Bean
    @Throws(Exception::class)
    fun configure(http: HttpSecurity): SecurityFilterChain {
        http
            ....
            .authorizeHttpRequests()
            .requestMatchers(PathRequest.toH2Console()).permitAll() // (1)
            .and()
            .headers().frameOptions().sameOrigin() // (2)
            .and()
            .csrf().disable() // (3)
        return http.build()
    }
}
```

Rôles et authorities

Spring Security distingue les authorities, accessible par une simple chaîne (encapsulée ensuite dans un objet de type **GrantedAuthority**) : **USER, ADMIN…**
et le rôle équivalent, authority préfixée par ROLE_ : **ROLE_USER, ROLE_ADMIN…**.

Il est possible de faire référence à l'un ou à l'autre indifféremment.

Le rôle des utilisateurs pourra dons être stocké en base de données, sous la forme d'une chaîne, ou d'une liste de chaînes, et il appartient ensuite à UserDetailsService de retourner un User avec une liste de **GrantedAuthority** :

```
    private fun getGrantedAuthorities(user: User): List<GrantedAuthority>? {
        val authorities: MutableList<GrantedAuthority> = ArrayList()
        val role: Role = user.role
        authorities.add(SimpleGrantedAuthority(role.name))
        return authorities
    }
```

Hiérarchie des authorities

Il est possible de définir les authorities de manière hiérarchique :

```
    @Bean
    fun roleHierarchy(): RoleHierarchyImpl? {
        val roleHierarchy = RoleHierarchyImpl()
        roleHierarchy.setHierarchy("ROLE_ADMIN > ROLE_STAFF > ROLE_USER >
ROLE_GUEST")
        return roleHierarchy
    }

    @Bean
    fun expressionHandler(): DefaultWebSecurityExpressionHandler? {
        val expressionHandler = DefaultWebSecurityExpressionHandler()
        expressionHandler.setRoleHierarchy(roleHierarchy())
        return expressionHandler
    }
```

Les utilisateurs peuvent dans ce cas n'avoir qu'une seule authority, qui leur en confère plusieurs avec la hiérarchie des rôles.

Inutilisable pour l'instant avec la version Spring security 6.0.1 (voir bug existant)

Configuration des rôles

```
@Configuration
@EnableWebSecurity
class WebSecurityConfig {

    @Bean
    @Throws(Exception::class)
    fun configure(http: HttpSecurity): SecurityFilterChain {
        http.authorizeHttpRequests { authorizeHttpRequests ->
            authorizeHttpRequests.requestMatchers("/css/**", "/assets/**",
"/login/**").permitAll() // (1)
authorizeHttpRequests.requestMatchers("/admin/**").hasRole("ROLE_ADMIN") // (2)
            authorizeHttpRequests.requestMatchers("/user/**").hasAuthority("USER")
// (3)
authorizeHttpRequests.requestMatchers("/staff/**").hasAnyAuthority("USER","ADMIN","
MANAGER") // (4)
            authorizeHttpRequests.anyRequest().authenticated() // (5)
        }
        return http.build()
    }
}
```

Méthode	Description
1 - **permitAll**	Pas de sécurisation sur les urls commençant par **/css**, **/assets**, **/login**.
2 - **hasRole**	Les utilisateurs ayant le rôle **ROLE_ADMIN** peuvent accéder aux urls commençant par **/admin**.
3 - **hasAuthority**	Les utilisateurs ayant l'authority **USER** peuvent accéder aux urls commençant par **/user**.

Méthode	Description
4 - **hasAnyAuthority**	Les utilisateurs ayant l'une des authorities **USER**, **ADMIN** ou **MANAGER** peuvent accéder aux urls commençant par **/staff**.
5 - **authenticated**	Pour toutes les autres URLs, il faut être authentifié.

defense in depth

Spring security permet de définir des authorisations sur les méthodes sollicitées, au delà des autorisations qu'il est possible de définir sur les URLs.

Activation

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity(
    prePostEnabled = true,
    securedEnabled = true,
    jsr250Enabled = true
)
class WebSecurityConfig {
    ...
}
```

Propriété	Défaut	Description
prePostEnabled	true	permet d'utiliser les annotations **@PreAuthorize** et **@PostAuthorize**.
securedEnabled	false	permet d'utiliser l'annotation **@Secured**.
jsr250Enabled	false	permet d'utiliser l'annotation **@RolesAllowed**.

@Secured

```
    @Secured("ROLE_ADMIN")
    fun adminOnly():String {
        return "admin"
    }
```

@RolesAllowed

```
    @RolesAllowed(value = [ "ROLE_USER", "ROLE_MANAGER"])
    fun userOrManager():String {
        return "userOrManager"
    }
```

@PreAuthorize

Permet d'agir avant l'exécution de la méthode :

- Avec usage des SpEL
- En testant éventuellement les paramètres passés.

```
@PreAuthorize("has_role('ROLE_VIEWER')")
fun getUsernameInUpperCase():String=getUserName().upperCase()
```

Comparaison du paramètre **username** avec le nom de l'utilisateur connecté :

```
@PreAuthorize("#username == authentication.principal.username")
fun getRoles(username:String):List<String> {
    ...
}
```

@PostAuthorize

Permet d'agir après l'exécution de la méthode :

- Avec usage des SpEL
- En testant éventuellement le retour.

```
@PostAuthorize("returnObject.username == authentication.principal.nickName")
fun loadUserDetail(username:String):CustomUser {
    return userRoleRepository.loadUserByUserName(username)
}
```

voir https://docs.spring.io/spring-security/reference/servlet/authorization/expression-based.html

Mise en place CSRF

Objectif : éviter les attaques CSRF

Activation

Activer CSRF dans le fichier **WebSecurityConfig** en enlevant la ligne :

```
http.csrf(AbstractHttpConfigurer::disable)
```

La protection csrf est activée par défaut dans **Spring security**

Intégration dans les vues

Dans **application.properties**, exposer les attributs de requête dans les vues :

```
# Expose Request Attributes
spring.mustache.servlet.expose-request-attributes=true
```

voir Spring Cross Site Request Forgery