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

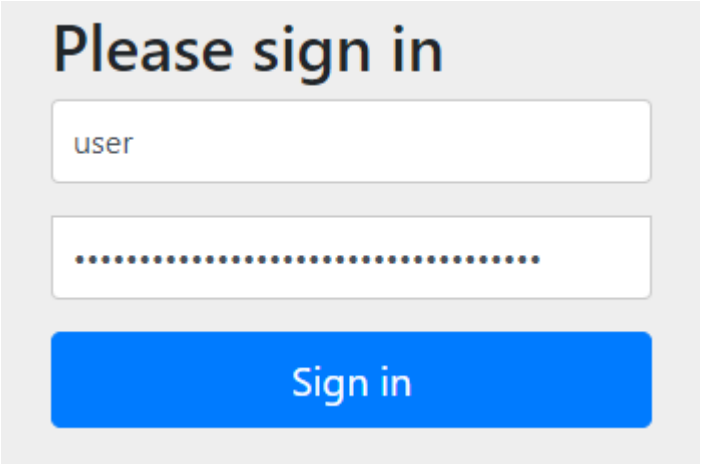
Ajouter les dépendances suivantes à **pom.xml** :

```
        <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
public class WebSecurityConfiguration {

    @Bean
    public SecurityFilterChain configure(HttpSecurity http) throws Exception {
        http.csrf(AbstractHttpConfigurer::disable).authorizeHttpRequests(
                        (req) -> req.requestMatchers(
                                        AntPathRequestMatcher.antMatcher("/"),
AntPathRequestMatcher.antMatcher("/css/**"),
                                        AntPathRequestMatcher.antMatcher("/js/**"),
AntPathRequestMatcher.antMatcher("/img/**"),
AntPathRequestMatcher.antMatcher("/register/**")
                                    )
                                    .permitAll()
                                    .anyRequest()
                                    .authenticated()
                ).formLogin();
        return http.build();
    }
}
```

Authentification depuis une BDD

Entity

Créer ou reprendre votre entity **User** pour gérer les utilisateurs et leur rôle :

Elle devra implémenter l'interface **UserDetails** :

```
public class User implements UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(length = 50, unique = true)
    private String login;

    @Column(length = 255)
    private String password;

    @Column(length = 255)
    private String email;

    @ManyToOne
    private Role role;
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        List<GrantedAuthority> authorities = new ArrayList<>();
        if (role != null) {
            authorities.add(new SimpleGrantedAuthority(role.getName()));
        }
        return authorities;
    }

    @Override
    public String getUsername() {
        return login;
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
```

```
    }
}
```

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

```
public interface UserRepository extends CrudRepository<User, Long> {
    Optional<User> findByLoginOrEmail(String login, String email);
}
```

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** :

```
@Service
public class UserService implements UserDetailsService {

    @Autowired
    private UserRepository uRepo;

    @Autowired
    private RoleRepository rRepo;

    @Autowired
    private PasswordEncoder pEncoder;

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        Optional<User> optUser = uRepo.findByLoginOrEmail(username, username);
        return optUser.orElseThrow(() -> new UsernameNotFoundException("Utilisateur
inconnu"));
    }

    public void encodePassword(User user) {
        user.setPassword(pEncoder.encode(user.getPassword()));
    }

    public User createUser(String login, String password) {
        User user = new User();
        user.setLogin(login);
        user.setPassword(password);
        encodePassword(user);
        user.setRole(getRole("USER"));
        return uRepo.save(user);
    }
}
```

Password encoder

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

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfiguration {

    @Bean
    public SecurityFilterChain configure(HttpSecurity http) throws Exception {
        ...
        return http.build();
    }

    @Primary
    @Bean
    public UserDetailsService getUserDetailsService() {
        return new UserService();
    }

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

Le **Bean** UserDetailsService est marqué comme **@Primary** pour être le premier choisi par Spring lors de l'injection.

Déclaration du service

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfiguration {
    ...
    @Bean
    public DaoAuthenticationProvider authenticationProvider(UserDetailsService
userService) {
        DaoAuthenticationProvider auth = new DaoAuthenticationProvider();
        auth.setUserDetailsService(userService);
        auth.setPasswordEncoder(getPasswordEncoder());
        return auth;
    }
}
```

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="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
@EnableMethodSecurity(
        prePostEnabled = true, proxyTargetClass = true
)
```

```
public class WebSecurityConfiguration {

    @Bean
    public SecurityFilterChain configure(HttpSecurity http) throws Exception {
        http.
                ...
                .formLogin(
                        (form) -> form.loginPage("/login").defaultSuccessUrl("/",
true).permitAll()
                );
        return http.build();
    }
```

Connexion à la console H2

Pour autoriser l'accès à la console h2 :

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

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfiguration {

    @Bean
    public SecurityFilterChain configure(HttpSecurity http) throws Exception {
        http.csrf(AbstractHttpConfigurer::disable).authorizeHttpRequests(   // (1)
                        (req) -> req.requestMatchers(
                                        AntPathRequestMatcher.antMatcher("/h2-
console/**"),   // (2)
                                        ...
                                )
                                .permitAll()
                                .anyRequest()
                                .authenticated()
                ).headers(
                        (headers) ->
headers.frameOptions(HeadersConfigurer.FrameOptionsConfig::sameOrigin)   // (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.

Hiérarchie des authorities

Il est possible de définir les authorities de manière hiérarchique :

```
    @Bean
    static RoleHierarchy roleHierarchy() {
        RoleHierarchyImpl hierarchy = new RoleHierarchyImpl();
        hierarchy.setHierarchy("ROLE_ADMIN > ROLE_USER");
        return hierarchy;
    }


    @Bean
    static MethodSecurityExpressionHandler methodSecurityExpressionHandler() {
        DefaultMethodSecurityExpressionHandler expressionHandler = new
DefaultMethodSecurityExpressionHandler();
        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**.
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
```

Utilisation dans un formulaire :

```
<form>
...
    <input type="hidden"
           name="{{_csrf.parameterName}}"
           value="{{_csrf.token}}"/>
...
</form>
```

voir Spring Cross Site Request Forgery