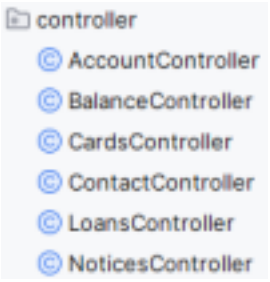
Atelier Spring Security

I. Partie 1 : Exploration du Spring Security

1. Cloner le projet SpringSecurity : https://github.com/badrhr/SpringSecurity.git Cette version de l'application explose l'ensemble des services sans aucune sécurité. L'objectif est de sécuriser les quatre services critiques avec un mécanisme d'authentification et de laisser les deux service **Cantact** et **Notices** publique sans aucune authentification.



2. Vérifier l'existence de la dépendance suivante dans le fichier pom.xml.

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

3. Exécuter l'application [http://localhost:8080/myCards]. Qui ce que vous remarquez ?

4. Entrer **user** comme login et le mot de passe utiliser le mot de passe générer dans la console de projet.

5. Pour changer le login et le mot de passe, modifier le fichier application.properties en ajoutant les deux lignes suivantes, et réexécuter l'application:

```
spring.security.user.name = xproce
spring.security.user.password = 12345
```

Pour satisfaire le comportement de la sécurité discuter dans la question 1, on doit tout d'abord explorer les deux mécanismes basic de Spring-Security :

a. Configuration pour refuser toutes les requêtes.

b. Configuration pour autoriser toutes les requêtes.

6. Créer un nouveau package "**config**", et créer la classe **ProjectSecurityConfig avec l'annotation @Configuration.**

```
@Configuration
public class ProjectSecurityConfig {… }
```

7. Dans la même classe, créer le bean suivant:

```
@Bean
SecurityFilterChain defaultSecurityFilterChain( HttpSecurity http) throws
Exception {  // Configuration to deny all the requests
 http.authorizeHttpRequests(requests ->
requests.anyRequest().denyAll())  //This line configures form-based
authentication
 .formLogin(Customizer.withDefaults())
 //This line configures HTTP Basic authentication
 .httpBasic(Customizer.withDefaults());
 return http.build();
}
```

8. Tester l'application.

9. De la même manière modifier le bean de la question 7, et tester l'application :

```
@Bean
SecurityFilterChain defaultSecurityFilterChain( HttpSecurity http) throws
  Exception { // Configuration to permit all the requests
   http.authorizeHttpRequests(requests ->
requests.anyRequest().permitAll())
.formLogin(Customizer.withDefaults())
 .httpBasic(Customizer.withDefaults());
 return http.build();
}
```

10. Dans notre cas (pratiquement dans tous des projets), on devra personnaliser cette configuration par défaut :

```
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws
Exception {  http.csrf((csrf) -> csrf.disable())
 .authorizeHttpRequests((requests)->requests

.requestMatchers("/myAccount","/myBalance","/myLoans","/myCards").authenticate
d()  .requestMatchers("/notices","/contact","/register").permitAll())
.formLogin(Customizer.withDefaults())
 .httpBasic(Customizer.withDefaults());
 return http.build();
}
```

11. La méthode la plus efficace consiste à définir un tableau des endpoints n'exigeant pas d'authentification, appelé AUTH_WHITELIST.


```
private static final String[] AUTH_WHITELIST = {
 "/notices",
 "/contact",
 "/register"
};
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http)
throws Exception {  http.authorizeRequests(authorizeRequests ->
 authorizeRequests
 .requestMatchers(AUTH_WHITELIST).permitAll()
 .anyRequest().authenticated()
 )
…
```

II. Partie 2 : Gestion des utilisateurs

La configuration précédente vis a vi les utilisateurs n'est pas pratique, car tous les intervenants vont être obligés à partager les mêmes informations d'authentification. La première solution est de créer un nombre de compte en mémoire pour les diffèrent intervenants, la deuxième méthode est d'utiliser la table **users** dans la base de données, et la troisième solution est d'utiliser une table personnalisée.

12. Créer la classe **UserSecurityConfig** dans le package config :

```
@Configuration
```

```
public class UserSecurityConfig { … }
```

13. Pour créer des comptes des utilisateurs dans la mémoire, créer le bean suivant qui utilise la méthode ***withDefaultPasswordEncoder*()** qui encode le mot de passe automatiquement:

```
@Bean
 public InMemoryUserDetailsManager userDetailsService() {
 /*Approach 1 where we use withDefaultPasswordEncoder() method while creating the
user details*/  UserDetails admin = User.withDefaultPasswordEncoder()
 .username("admin")
 .password("12345")
 .authorities("admin")
 .build();
 UserDetails user = User.withDefaultPasswordEncoder()
 .username("user")
 .password("12345")
 .authorities("read")
 .build();
 return new InMemoryUserDetailsManager(admin, user);
}
```

14. Tester l'application.

15. Remarquer que la méthode *withDefaultPasswordEncoder*() est deprecated.

16. Pour la question 2, modifiez le bean en remplaçant la méthode **withDefaultPasswordEncoder**() par **withUsername**(String username):

```
    @Bean
     public InMemoryUserDetailsManager userDetailsService() {/*Approach 2
    where we use  NoOpPasswordEncoder Bean while creating the user
    details*/
     UserDetails admin = User.withUsername("admin")
     .password("12345")
     .authorities("admin")
     .build();
     UserDetails user = User.withUsername("user")
     .password("12345")
     .authorities("read")
     .build();
     return new InMemoryUserDetailsManager(admin, user);
}
```

17. Pour simuler un scénario sans encodage de mot de passe en vue des tests, injectons un bean personnalisé :

```
// NoOpPasswordEncoder is not recommended for production usage. Use only for non-prod. @return
        PasswordEncoder @Bean
        public PasswordEncoder passwordEncoder() {
         return NoOpPasswordEncoder.getInstance();
        }
```

Cette solution a résolu le premier problème, mais elle nous limite toujours à un nombre restreint de comptes. Pour une plus grande flexibilité, l'utilisation d'une base de données s'avère être la solution idéale. Elle nous permet d'enregistrer un nombre illimité de comptes.

18. Dans un premier temps, créer une base de données MySQL (xproce). Ensuite, configurons le fichier application.properties avec les paramètres de connexion suivants :

```
spring.datasource.url=jdbc:mysql://localhost:3306/xproce?createDatabaseIfNot
Exist=true spring.jpa.hibernate.ddl-auto=update
spring.datasource.username=root
spring.datasource.password=******
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

19. Créer les deux tables (users et authorities) dans la base de données :

```
CREATE TABLE `users` (
`id` INT NOT NULL AUTO_INCREMENT,
`username` VARCHAR(45) NOT NULL,
`password` VARCHAR(45) NOT NULL,
`enabled` INT NOT NULL,
PRIMARY KEY (`id`));

CREATE TABLE `authorities` (
 `id` int NOT NULL AUTO_INCREMENT,
 `username` varchar(45) NOT NULL,
 `authority` varchar(45) NOT NULL,
 PRIMARY KEY (`id`));

INSERT IGNORE INTO `users` VALUES (NULL, 'xproce', '123456', '1');
INSERT IGNORE INTO `authorities` VALUES (NULL, 'xproce', 'write');
```

20. Supprimer ou commenter l'annotation @Configuration dans classe UserSecurityConfig (Question 12).
21. Créer une nouvelle classe qui va contenir les différents beans:

```
public class ProjectSecurityConfigMYSQL {
@Bean
public UserDetailsService userDetailsService(DataSource dataSource) {
return new JdbcUserDetailsManager(dataSource);
}
@Bean
public PasswordEncoder passwordEncoder() {
return NoOpPasswordEncoder.getInstance();
}
}
```

JdbcUserDetailsManager est un outil puissant qui permet de gérer l'authentification des utilisateurs en s'appuyant sur une base de données. En fournissant un DataSource pointant vers les tables users et authorities, vous configurez Spring Security pour charger les informations des utilisateurs directement depuis la base de données.
La structure des tables : Bien que les noms de colonnes puissent varier, il est généralement recommandé d'avoir au moins les colonnes suivantes :
La table users: username (unique), password, enabled.
La table authorities: username (clé étrangère vers la table users), authority (nom du rôle).

22. Vérifier les différentes dépendances : MySQL, JPA, ….

23. Tester l'application [http://localhost:8080/myCards].

Personnalisé les paramètres d'authentication

24. Créer la table *customer* dans la même base de données *xproce*, et insérer un enregistrement qui va servir comme cas de test :

```
CREATE TABLE `customer` (
 `id` int NOT NULL AUTO_INCREMENT,
 `email` varchar(45) NOT NULL,
 `pwd` varchar(200) NOT NULL,
 `role` varchar(45) NOT NULL,
 PRIMARY KEY (`id`)
);
INSERT INTO `customer` (`email`, `pwd`, `role`) VALUES ('xproce@example.com', '54321',
'admin');
```

25. Dans le package config, créer la classe BankUserDetails avec l'annotation @Service qui implémente l'interface UserDetailsService.

```
@Service
public class BankUserDetails implements UserDetailsService {…}
```

26. Créer l'entité Customer représentant la table `customer` de la question 6. 27. Créer le repository CustomerRepository, et ajouter la signature de la méthode *findByEmail* :

```
public interface CustomerRepository extends
JpaRepository<Customer, Integer> {  public List<Customer>
findByEmail(String email);
}
```

28.Le code final de la classe est :

```
@Service
public class BankUserDetails implements UserDetailsService {
 @Autowired
 private CustomerRepository customerRepository;
 @Override
 public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {  String login;
 String password;
 List<GrantedAuthority> authorities = new ArrayList<>();
 List<Customer> customers = customerRepository.findByEmail(username);
 if (customers.size() == 0) {
 throw new UsernameNotFoundException(username + " Does Not Exist
… " );  } else {
 login = customers.get(0).getEmail();
 password = customers.get(0).getPwd();
 authorities.add(new
SimpleGrantedAuthority(customers.get(0).getRole()));  }
 return new User(login, password, authorities);
 }
 @Bean
 public PasswordEncoder passwordEncoder() {
 return NoOpPasswordEncoder.getInstance();
 }
}
```

29. Tester l'application [http://localhost:8080/myCards].