

# Cours Inf3522 - Développement d'Applications N tiers

## Lab\_5 : Sécurisation et tests du backend

Ce lab explique comment sécuriser et tester votre backend Spring Boot. Sécuriser votre backend est une partie cruciale du développement de code. Dans la partie de test de ce lab, nous créerons quelques tests unitaires en relation avec notre backend - cela facilitera la maintenance du code du backend. Nous utiliserons l'application de base de données que nous avons créée dans le lab précédent comme point de départ.

### Exercice 1

#### Comprendre Spring Security

Spring Security (<https://spring.io/projects/spring-security>) fournit des services de sécurité pour les applications web Java. Le projet Spring Security a été lancé en 2003 et était auparavant appelé **Acegi Security System** pour Spring.

Par défaut, Spring Security active les fonctionnalités suivantes :

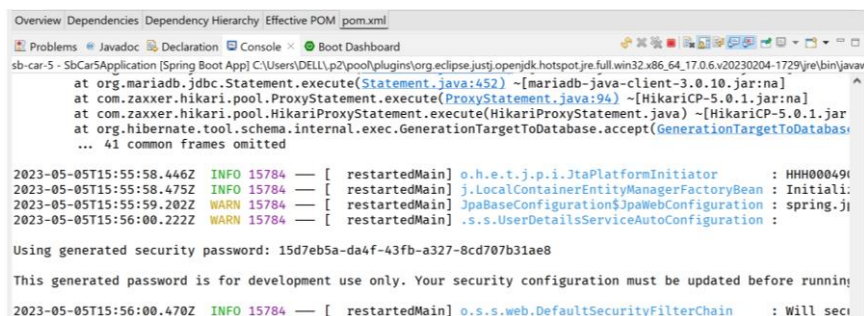
- Un **bean AuthenticationManager** avec un seul utilisateur en mémoire. Le nom d'utilisateur est « **user** » et le mot de passe est imprimé dans la sortie de la console.
- Des chemins ignorés pour les emplacements courants de ressources statiques, tels que **/css** et **/images**.
- Sécurité basique **HTTP** pour tous les autres points d'extrémité.
- Des événements de sécurité publiés sur l'interface **ApplicationEventPublisher** de Spring.
- Les fonctionnalités communes de bas niveau sont activées par défaut (**HTTP Strict Transport Security (HSTS)**, **cross-site scripting (XSS)**, **cross-site request forgery (CSRF)**, etc.).
- Une page de connexion générée automatiquement par défaut.

Vous pouvez inclure Spring Security dans votre application en ajoutant les dépendances suivantes au fichier **pom.xml**. La première dépendance est pour l'application et la deuxième est pour les tests :

```
49
50<
51    <groupId>org.springframework.boot</groupId>
52    <artifactId>spring-boot-starter-security</artifactId>
53  </dependency>
54<
55    <groupId>org.springframework.security</groupId>
56    <artifactId>spring-security-test</artifactId>
57    <scope>test</scope>
58  </dependency>
59</dependencies>
60</pre>
```

#### Stockage de l'utilisateur en mémoire

Lorsque vous démarrez votre application, vous pouvez voir dans la console que Spring Security a créé un utilisateur en mémoire avec un nom d'utilisateur « **user** ». Le mot de passe de l'utilisateur peut être vu dans la sortie de la console, comme illustré ici :



```
sb-car-5 - SbCar5Application [Spring Boot App] C:\Users\DELL\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64.17.0.6.v20230204-1729\jre\bin\javaw
at org.mariadb.jdbc.Statement.execute(Statement.java:452) ~[mariadb-java-client-3.0.10.jar:na]
at com.zaxxer.hikari.pool.ProxyStatement.execute(ProxyStatement.java:94) ~[HikariCP-5.0.1.jar:na]
at com.zaxxer.hikari.pool.HikariProxyStatement.execute(HikariProxyStatement.java) ~[HikariCP-5.0.1.jar:na]
at org.hibernate.tool.schema.internal.exec.GenerationTargetToDatabase.accept(GenerationTargetToDatabase:
... 41 common frames omitted

2023-05-05T15:55:58.446Z INFO 15784 [ restartedMain] o.h.e.t.j.p.i.JtaPlatformInitiator : HHH00049i
2023-05-05T15:55:58.475Z INFO 15784 [ restartedMain] j.LocalContainerEntityManagerFactoryBean : Initiali:
2023-05-05T15:55:59.202Z WARN 15784 [ restartedMain] JpaBaseConfiguration$JpaWebConfiguration : spring.ji
2023-05-05T15:56:00.222Z WARN 15784 [ restartedMain] .s.s.UserDetailsServiceAutoConfiguration :

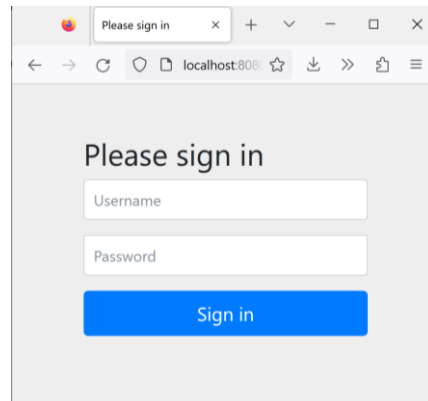
Using generated security password: 15d7eb5a-da4f-43fb-a327-8cd707b31ae8

This generated password is for development use only. Your security configuration must be updated before running

2023-05-05T15:56:00.470Z INFO 15784 [ restartedMain] o.s.s.web.DefaultSecurityFilterChain : Will secur
```

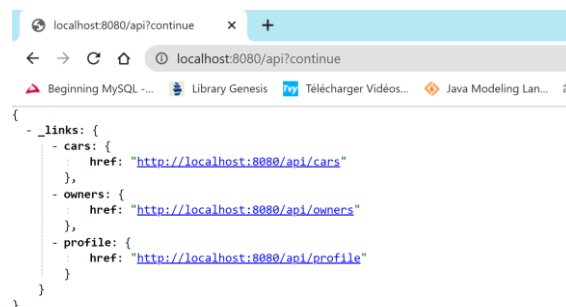
Si aucun mot de passe n'apparaît dans la console, essayez de redémarrer votre projet.

Maintenant, si vous effectuez une requête **GET** sur le point d'extrémité racine de votre interface de programmation d'application (API), vous verrez qu'il est désormais sécurisé. Ouvrez votre navigateur web et accédez à <http://localhost:8080/api>. Maintenant, vous verrez que vous êtes redirigé vers la page de connexion par défaut de Spring Security : <http://localhost:8080/login>, comme illustré dans la capture d'écran suivante :



Pour pouvoir effectuer une requête **GET** réussie, nous devons nous authentifier. Saisissez « user » dans le champ Nom d'utilisateur et copiez le mot de passe généré depuis la console dans le champ Mot de passe.

Avec l'authentification, nous pouvons voir que la réponse contient nos ressources API, comme illustré dans la capture d'écran suivante :

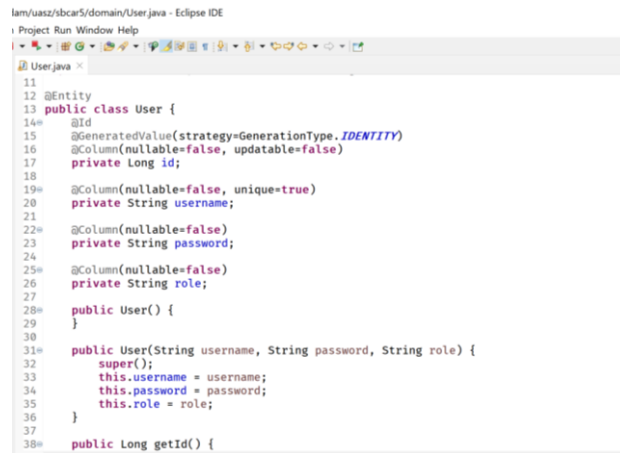


## Stockage de l'utilisateur dans une Base de données

Pour enregistrer les utilisateurs dans la base de données, vous devez créer une classe d'entité utilisateur et son repository.

Les mots de passe ne doivent pas être enregistrés en texte brut dans la base de données. Spring Security propose plusieurs algorithmes de hachage, tels que bcrypt, que vous pouvez utiliser pour hacher les mots de passe. Les étapes suivantes vous montrent comment implémenter cela :

Créez une nouvelle classe appelée User dans le package domain. Ajoutez l'annotation @Entity à la classe User. Ajoutez les champs de classe identifiant (ID), nom d'utilisateur, mot de passe et rôle. Enfin, ajoutez les constructeurs, les getters et les setters. Nous définirons tous les champs comme étant nullable, et spécifierons que le nom d'utilisateur doit être unique en utilisant l'annotation @Column. Consultez le code source User.java suivant pour les champs et les constructeurs :

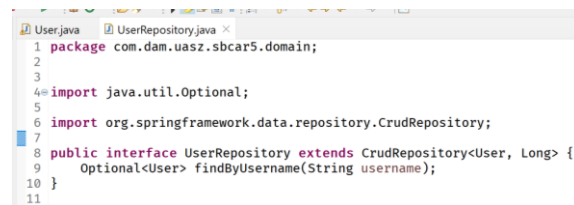


```

11
12 @Entity
13 public class User {
14     @Id
15     @GeneratedValue(strategy=GenerationType.IDENTITY)
16     @Column(nullable=false, updatable=false)
17     private Long id;
18
19     @Column(nullable=false, unique=true)
20     private String username;
21
22     @Column(nullable=false)
23     private String password;
24
25     @Column(nullable=false)
26     private String role;
27
28     public User() {
29     }
30
31     public User(String username, String password, String role) {
32         super();
33         this.username = username;
34         this.password = password;
35         this.role = role;
36     }
37
38     public Long getId() {

```

Créer l'interface UserRepository dans le package domain. Le code source de la classe du référentiel (repository) est similaire à ce que nous avons réalisé dans le chapitre précédent, mais il existe une méthode de requête, `findByUsername`, dont nous avons besoin pour les étapes suivantes. Cette méthode est utilisée pour rechercher un utilisateur dans la base de données lors du processus d'authentification. La méthode renvoie un `Optional` pour éviter une exception de valeur nulle. Consultez le code source UserRepository suivant :



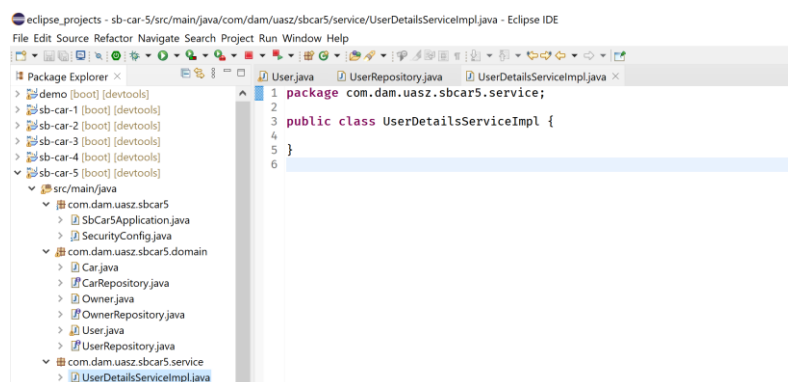
```

1 package com.dam.uasz.sbcar5.domain;
2
3
4 import java.util.Optional;
5
6 import org.springframework.data.repository.CrudRepository;
7
8 public interface UserRepository extends CrudRepository<User, Long> {
9     Optional<User> findByUsername(String username);
10 }
11

```

Ensuite, nous allons créer une classe qui implémente l'interface UserDetailsService fournie par Spring Security. Spring Security l'utilise pour l'authentification et l'autorisation des utilisateurs. Créez un nouveau package service.

Créez une nouvelle classe appelée UserDetailsServiceImpl dans le package service que nous venons de créer. Maintenant, la structure de votre projet devrait ressembler à ceci :



```

1 package com.dam.uasz.sbcar5.service;
2
3 public class UserDetailsServiceImpl {
4
5 }
6

```

Nous devons injecter la classe UserRepository dans la classe UserDetailsServiceImpl car cela est nécessaire pour récupérer l'utilisateur depuis la base de données lorsque Spring Security gère l'authentification. La méthode `findByUsername` que nous avons implémentée précédemment renvoie un `Optional`, nous pouvons donc utiliser la méthode `isPresent()` pour vérifier si l'utilisateur existe. Si l'utilisateur n'existe pas, nous lançons une exception `UsernameNotFoundException`. La méthode `loadByUsername` retourne l'objet `UserDetails`, qui est requis pour l'authentification. Nous

utilisons la classe `UserBuilder` de Spring Security pour construire l'utilisateur pour l'authentification. Voici le code source de `UserDetailsServiceImpl.java` :

```
12
13 import com.dam.uas2.sbcars5.domain.User;
14 import com.dam.uas2.sbcars5.domain.UserRepository;
15
16
17 @Service
18 public class UserDetailsServiceImpl implements UserDetailsService {
19     @Autowired
20     private UserRepository repository;
21
22     @Override
23     public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
24         Optional<User> user = repository.findByUsername(username);
25
26         UserBuilder builder = null;
27         if (user.isPresent()) {
28             User currentUser = user.get();
29             builder = org.springframework.security.core.userdetails.User.withUsername(username);
30             builder.password(currentUser.getPassword());
31             builder.roles(currentUser.getRole());
32         } else {
33             throw new UsernameNotFoundException("User not found.");
34         }
35
36         return builder.build();
37     }
38 }
```

Dans notre classe de configuration de sécurité, nous devons spécifier que Spring Security doit utiliser les utilisateurs de la base de données plutôt que des utilisateurs en mémoire. Ajoutez une nouvelle méthode `configureGlobal` pour activer les utilisateurs depuis la base de données. Nous ne devons jamais enregistrer les mots de passe en texte brut dans la base de données. Par conséquent, nous allons définir un algorithme de hachage des mots de passe dans la méthode `configureGlobal`. Dans cet exemple, nous utilisons l'algorithme `bcrypt`. Cela peut être facilement mis en œuvre avec la classe `BCryptPasswordEncoder` de Spring Security, qui encode un mot de passe haché dans le processus d'authentification. Voici le code source de `SecurityConfig.java`. Désormais, le mot de passe doit être haché à l'aide de `BCrypt` avant d'être enregistré dans la base de données :

```
1 package com.dam.uas2.sbcars5;
2
3 import org.springframework.context.annotation.Configuration;
4 import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
5 import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
6
7 import com.dam.uas2.sbcars5.service.UserDetailsServiceImpl;
8
9 import org.springframework.beans.factory.annotation.Autowired;
10
11 @Configuration
12 public class SecurityConfig {
13
14     @Autowired
15     private UserDetailsServiceImpl userDetailsService;
16
17     @Autowired
18     public void configureGlobal(AuthenticationManagerBuilder auth)
19         throws Exception {
20         auth.userDetailsService(userDetailsService)
21             .passwordEncoder(new BCryptPasswordEncoder());
22     }
23
24 }
```

**NB** : *BCrypt est une fonction de hachage puissante conçue par Niels Provos et David Mazières. Voici un exemple de hachage BCrypt généré à partir de la chaîne "admin" :*

*\$2a\$10\$8cjz47bjbR4Mn8GMggIZx.vyjhLXR/SKKMSZg.mPgvpmuossKi8GW*

*\$2a* représente la version de l'algorithme et *\$10* représente la force de l'algorithme. La force par défaut de la classe `BcryptPasswordEncoder` de Spring Security est de 10. *BCrypt* génère un sel aléatoire lors du hachage, ce qui fait que le résultat haché est toujours différent.

Ensuite modifier la classe principale en ajoutant les utilisateurs. Pour ce faire, il faut injecter le `UserRepository` : ligne 32 et 33, puis créer deux instances d'utilisateur lignes : 55 -58.

```

29= @Autowired
30 private OwnerRepository orepository;
31
32= @Autowired
33 private UserRepository urepository;
34
35= public static void main(String[] args) {
36     SpringApplication.run(SbCar5Application.class, args);
37 }
38
39= @Override
40 public void run(String... args) throws Exception {
41
42     Owner owner1 = new Owner("John", "Johnson");
43     Owner owner2 = new Owner("Mary", "Robinson");
44     orepository.saveAll(Arrays.asList(owner1, owner2));
45
46     Car car1 = new Car("Ford", "Mustang", "Red", "ADF-1121", 2021, 59000);
47     Car car2 = new Car("Nissan", "Leaf", "White", "SSJ-3002", 2019, 29000);
48     Car car3 = new Car("Toyota", "Prius", "Silver", "KKO-0212", 2020, 39000);
49     repository.saveAll(Arrays.asList(car1, car2, car3));
50
51     for (Car car : repository.findAll()) {
52         logger.info(car.getBrand() + " " + car.getModel());
53     }
54
55     urepository.save(new User("user",
56         "$2a$10$NVm0n8ElarGg7zW01CxUdei7vWoPg91Lz2aYavh9.f9q0e4bRadue", "USER"));
57     urepository.save(new User("admin",
58         "$2a$10$8cjz47bjbR4Mn8GMg9IZx.vyjhLXR/SKKMSZ9.mP9vpMu0ssKi8GW", "ADMIN"));
59 }
60 }

```

Après avoir exécuté votre application, vous verrez qu'il y a maintenant une table utilisateur dans la base de données et que deux enregistrements d'utilisateurs sont enregistrés avec des mots de passe hachés, comme illustré dans la capture d'écran suivante :

```

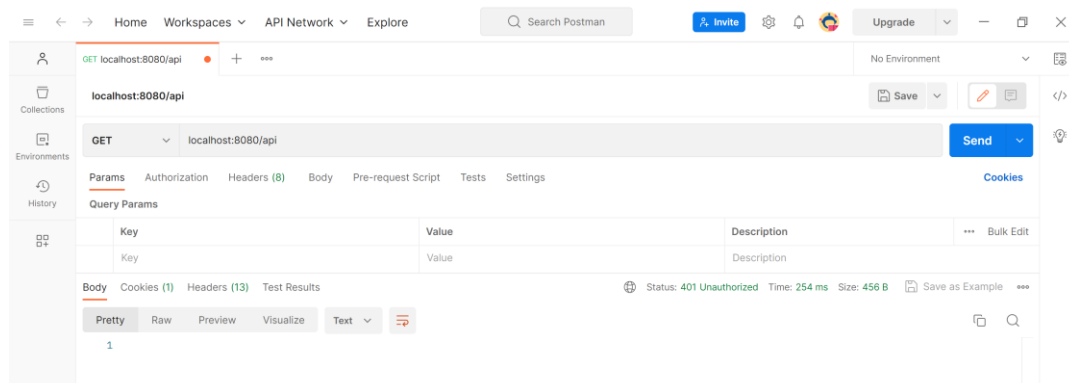
XAMPP for Windows - mysql -u root

MariaDB [cardb]> show tables ;
+-----+
| Tables_in_cardb |
+-----+
| car              |
| car_seq         |
| owner           |
| owner_seq       |
| user            |
+-----+
5 rows in set (0.001 sec)

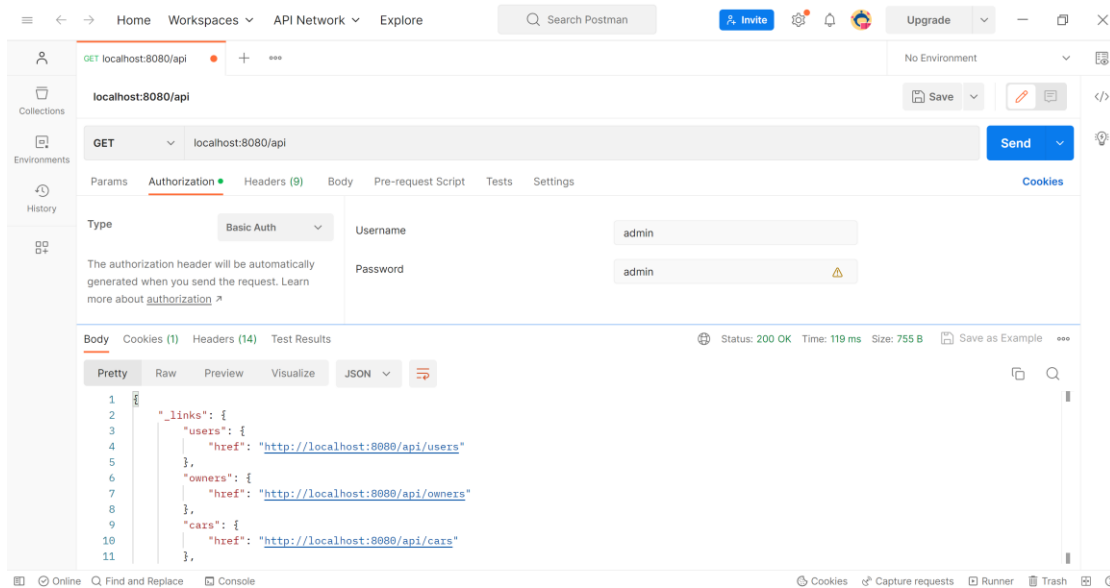
MariaDB [cardb]> select * from user ;
+-----+
| id | password | role | username |
+-----+
| 1  | $2a$10$NVm0n8ElarGg7zW01CxUdei7vWoPg91Lz2aYavh9.f9q0e4bRadue | USER | user |
| 2  | $2a$10$8cjz47bjbR4Mn8GMg9IZx.vyjhLXR/SKKMSZ9.mP9vpMu0ssKi8GW | ADMIN | admin |
+-----+
2 rows in set (0.000 sec)

```

Maintenant, si vous essayez d'envoyer une requête GET à l'adresse <http://localhost:8080/api> sans authentification, vous recevrez une erreur 401 Unauthorized. Vous devez vous authentifier pour pouvoir envoyer une requête réussie. La différence par rapport à l'exemple précédent est que nous utilisons les utilisateurs de la base de données pour l'authentification.



Vous pouvez voir une requête GET vers l'endpoint /api utilisant l'utilisateur admin dans la capture d'écran suivante. Vous pouvez également utiliser Postman et l'authentification de base (basic authentication) :



Maintenant, vous pouvez voir que nous obtenons les utilisateurs en appelant l'endpoint /users dans notre service web RESTful, et c'est quelque chose que nous voulons éviter. Comme mentionné précédemment, Spring Data REST génère par défaut un service web RESTful à partir de tous les référentiels publics. Nous pouvons utiliser le paramètre `exported` de l'annotation `@RepositoryRestResource` et le définir sur `false`, de cette manière le référentiel suivant ne sera pas exposé en tant que ressource REST :

```

1 package com.dam.usz.sbcars5.domain;
2
3
4 import java.util.Optional;
5
6 import org.springframework.data.repository.CrudRepository;
7 import org.springframework.data.rest.core.annotation.RepositoryRestResource;
8
9 @RepositoryRestResource(exported = false)
10 public interface UserRepository extends CrudRepository<User, Long> {
11     Optional<User> findByUsername(String username);
12 }

```

Maintenant, si vous effectuez une requête GET vers l'endpoint /api, vous verrez que l'endpoint /users n'est plus visible, comme illustré dans la capture d'écran suivante :

