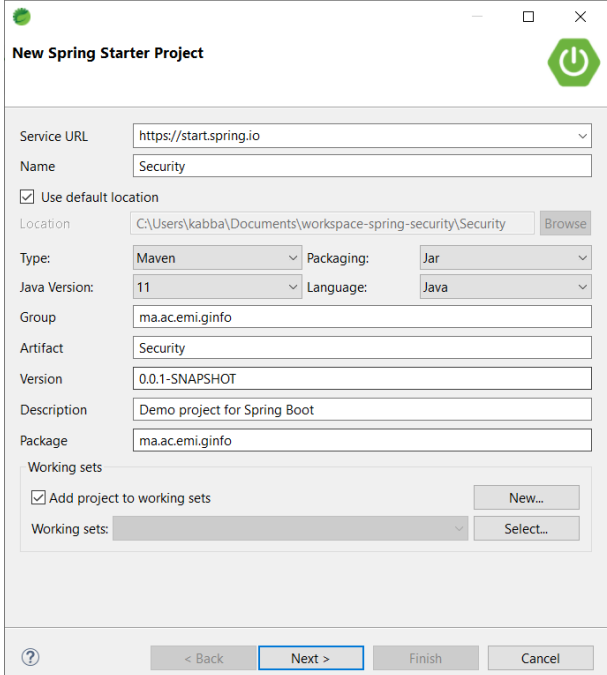


# Sécuriser une application Web

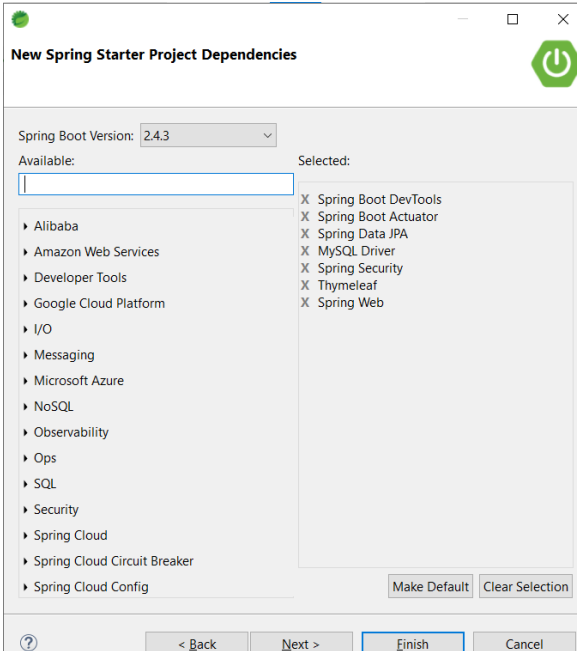
Ce tutorial vous guide tout au long du processus de création d'une application Web simple avec des ressources protégées par Spring Security.

## Création d'une application Spring Boot

Créer une application Spring Boot avec les options suivante :

<b>Nom du projet :</b> Security <b>Groupe :</b> ma.ac.emi.ginfo <b>Artifact :</b> Security <b>Package :</b> ma.ac.emi.ginfo	 <p>The dialog box 'New Spring Starter Project' contains the following fields and options:</p> <ul style="list-style-type: none"><li>Service URL: <a href="https://start.spring.io">https://start.spring.io</a></li><li>Name: Security</li><li><input checked="" type="checkbox"/> Use default location</li><li>Location: C:\Users\kabba\Documents\workspace-spring-security\Security</li><li>Type: Maven</li><li>Packaging: Jar</li><li>Java Version: 11</li><li>Language: Java</li><li>Group: ma.ac.emi.ginfo</li><li>Artifact: Security</li><li>Version: 0.0.1-SNAPSHOT</li><li>Description: Demo project for Spring Boot</li><li>Package: ma.ac.emi.ginfo</li><li>Working sets: <input checked="" type="checkbox"/> Add project to working sets</li><li>Buttons: &lt; Back, Next &gt;, Finish, Cancel</li></ul>
--	--

Pour ce projet nous allons choisir les dépendances suivante : **DevTools, Actuator, JPA, MySQL, Security, Thymeleaf et Web.**



The dialog box 'New Spring Starter Project Dependencies' shows the following configuration:

- Spring Boot Version: 2.4.3
- Available: (empty list)
- Selected:
  - X Spring Boot DevTools
  - X Spring Boot Actuator
  - X Spring Data JPA
  - X MySQL Driver
  - X Spring Security
  - X Thymeleaf
  - X Spring Web
- Buttons: < Back, Next >, Finish, Cancel

- Essayer de démarrer l'application. Que vous avez obtenu ?
- Mettons tous les dépendances en commentaire dans le POM.XML, sauf le starter **Web**, de **Test** et de **Thymeleaf**.
- Redémarrer l'application. Que vous avez obtenu ?

## Créer une application Web non sécurisée

Avant de pouvoir appliquer la sécurité à une application Web, vous avez besoin d'une application Web à sécuriser. Cette section vous guide tout au long de la création d'une application Web simple. Ensuite, vous le sécuriserez avec Spring Security dans la section suivante.

L'application Web comprend deux vues simples : une page d'accueil et une page « Bonjour les EMISTs ». Les deux pages web sont des pages html enrichies et interprétable par le moteur de vue Thymeleaf.

- Créons, sous l'arborescence de Thymeleaf : **src/main/resources/templates/**, deux pages html nommé **hello.html** et **index.html** respectivement.
- Changer le titre de chaque page et ajouter dans chacune d'elle un élément h3 pour pouvoir les identifier.
- Redémarrer l'application. Que vous avez obtenu ?
- Ajouter le path **/hello** à votre url : <http://localhost:8080/hello>. Que vous avez obtenu ?
- Modifier la page index.html comme suit (pour le moment on va ignorer les apports de Thymeleaf) :



```

1 <!DOCTYPE html>
2 <html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org"
3     xmlns:sec="https://www.thymeleaf.org/thymeleaf-extras-springsecurity3">
4 <head>
5 <meta charset="ISO-8859-1">
6 <title>Page d'accueil</title>
7 </head>
8 <body>
9
10 <h3>Page d'accueil</h3>
11
12 <p><a th:href="@{/hello}">Cliquez ici</a> pour voir un message d'accueil.</p>
13
14 </body>
15 </html>

```

- Redémarrer l'application. Que vous avez obtenu ?
- Cliquer sur le lien hypertexte **Cliquez ici**. Que vous avez obtenu ?

L'application Web est basée sur **Spring MVC**. Par conséquent, nous devons ajouter des contrôleurs de vue pour supporter la requête hello.

- Créons un contrôleur MVC dans l'application comme indiquer dans l'image suivante :

```
hello.html  index.html  HomeControlleur.java  ✕
1 package ma.ac.emi.ginfo;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.web.bind.annotation.GetMapping;
5
6 @Controller
7 public class HomeControlleur {
8
9     @GetMapping("/hello")
10    public String bonjour() {
11        return "hello";
12    }
13 }
14
```

- Redémarrer l'application, cliquer sur le lien hypertexte **Cliquez ici**. Que vous avez obtenu ?

Maintenant que vous disposez d'une application Web non sécurisée, vous pouvez la sécuriser.

## Configurer Spring Security

Supposons que vous souhaitiez empêcher les utilisateurs non autorisés de consulter la page d'accueil à l'adresse /hello. Dans l'état actuel des choses, si les visiteurs cliquent sur le lien de la page d'accueil, ils voient le message d'accueil sans aucune restriction. Vous devez ajouter une barrière qui oblige le visiteur à se connecter avant de pouvoir visiter cette page.

Pour ce faire, configurez Spring Security dans l'application. Si Spring Security est activé, Spring Boot sécurise automatiquement tous les points de terminaison HTTP avec une authentification « de base ». Cependant, vous pouvez personnaliser davantage les paramètres de sécurité. La première chose à faire est d'ajouter les dépendances de Spring Security au POM.XML.

- Supprimer le commentaire sur les dépendances de **Spring Security, Security Test et Thymeleaf Extra**.
- Redémarrer l'application, que vous avez obtenu ?
- Pour pouvoir consulter le site, utiliser le compte « user » avec le mot de passe généré dans la console de l'exécution.

Pour que la page d'accueil soit consultable par tous les internautes, nous devons configurer Spring Security.

- Créer un sous package nommé « **ma.ac.emi.ginfo.security** »
- Créer une classe dans le package nommée « **WebSecurityConfig** »
- Modifier la classe ainsi :

```

hello.html index.html HomeContrôleur.java Security/pom.xml WebSecurityConfig.java
3 import org.springframework.context.annotation.Configuration;
4 import org.springframework.security.config.annotation.web.builders.HttpSecurity;
5 import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
6 import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
7
8 @Configuration
9 @EnableWebSecurity
10 public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
11
12     @Override
13     protected void configure(HttpSecurity http) throws Exception {
14         http
15             .authorizeRequests()
16             .antMatchers("/").permitAll();
17     }
18
19 }

```

La classe **WebSecurityConfig** est annotée **@EnableWebSecurity** pour activer la prise en charge de la sécurité Web de Spring Security et fournir de l'intégration à Spring MVC. Il étend la classe **WebSecurityConfigurerAdapter** et remplace également quelques-unes de ses méthodes pour définir certaines spécificités de la configuration de la sécurité Web.

La méthode **configure(HttpSecurity)** définit quels chemins d'URL doivent être sécurisés et lesquels ne le doivent pas. Plus précisément, les chemins « / » et « /login » seront configurés pour ne nécessiter aucune authentification. Tous les autres chemins doivent être authentifiés.

Lorsqu'un utilisateur se connecte avec succès, il est redirigé vers la page précédemment demandée qui nécessitait une authentification.

- **Redémarrer l'application**, Que vous avez obtenu ?
- Modifier la classe en apportant la modification suivante :

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
        .antMatchers("/").permitAll()
        .anyRequest().authenticated();
}

```

- **Redémarrer l'application**, Que vous avez obtenu ? quelles est le code d'erreur obtenu ? et que signifie-t-il ?

A ce stade, vous regrettez probablement la nécessité de redémarrer l'application à chaque changement. Alors, on va utiliser un plugin qui va faciliter la donne.

- Décommenter la dépendance **Devtools** dans le POM.XML
- Redémarrer l'application et consulter la **page d'accueil**.
- Modifier le fichier **index.html** comme suit :

```

10 <h3>Page d'accueil</h3>
11
12 <p><a th:href="@{/hello}">Cliquez ici</a> pour voir un message d'accueil.</p>
13 <p>une autre paragraphe </p>
14
15 </body>
16 </html>

```

- **Sans redémarrer l'application**, Recharger le navigateur. Que vous avez obtenu ?
- Modifier la méthode **configure** de la classe **WebSecurityConfig** en ajoutant les lignes suivante :

```

12  @Override
13  protected void configure(HttpSecurity http) throws Exception {
14      http
15          .authorizeRequests()
16              .antMatchers("/").permitAll()
17              .anyRequest().authenticated()
18              .and()
19              .formLogin()
20                  .loginPage("/login")
21                  .permitAll()
22                  .and()
23                  .logout()
24                      .permitAll();
25  }

```

- Sauvegarder et **observer la console**. Que vous avez remarqué ?
- **Sans redémarrer l'application**, recharger le navigateur. Tenter d'accéder à la page **hello.html**. Que vous avez obtenu ? quelles est le code d'erreur obtenu ? et que signifie-t-il ?

Vous devez maintenant créer la page de connexion. Il faut aussi un contrôleur de vue pour la vue login. Il vous suffit donc de créer la vue de connexion elle-même, et de prévoir la redirection de l'entrée « **/login** » vers cette page.

- Créer la page **login.html** dans le sous répertoire **templates**.
- Modifier la page **login.html** pour quelle corresponde à ceci :

```

hello.html  index.html  HomeController.java  Security/pom.xml  WebSecurityConfig.java  login.html
1  <!DOCTYPE html>
2  <html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org"
3      xmlns:sec="https://www.thymeleaf.org/thymeleaf-extras-springsecurity3">
4  <head>
5      <title>Page de login </title>
6  </head>
7  <body>
8      <div th:if="${param.error}">
9          Nom d'utilisateur et password sont invalides.
10     </div>
11     <div th:if="${param.logout}">
12         vous n'êtes plus connecté.
13     </div>
14     <form th:action="@{/login}" method="post">
15         <div><label> Nom d'utilisateur : <input type="text" name="username"/> </label></div>
16         <div><label> Password : <input type="password" name="password"/> </label></div>
17         <div><input type="submit" value="Se connecter"/></div>
18     </form>
19 </body>
20 </html>

```

- Recharger la page du navigateur. Que vous avez obtenu ? comment vous l'expliquer ?

Ce modèle Thymeleaf présente un formulaire qui capture un **nom d'utilisateur** et un **mot de passe** et les publie sur **/login**. Tel que configuré, Spring Security fournit un filtre qui intercepte cette demande et authentifie l'utilisateur. Si l'utilisateur ne parvient pas à s'authentifier, la page est redirigée vers **/login?error** et votre page affiche le message d'erreur approprié. Une fois la déconnexion réussie, votre demande est envoyée à **/login?logout** et votre page affiche le message de réussite approprié.

- Ajouter un autre point d'entre à notre contrôleur :

```

-
6  @Controller
7  public class HomeController {
8
9      @GetMapping("/hello")
10     public String bonjour() {
11         return "hello";
12     }
13
14     @GetMapping("/login")
15     public String authentication() {
16         return "login";
17     }
18 }
19
20

```

- Recharger la page du navigateur. Que vous avez obtenu ? comment vous l'expliquer ?
- Tenter de se déconnecter avec l'url suivant : <http://localhost:8080/hello?logout>. Etes-vous déconnecté ? comment vous expliquez le résultat obtenu ?

Vous devez maintenant fournir au visiteur un moyen d'afficher le nom d'utilisateur actuel et de se déconnecter.

- Pour ce faire, mettez à jour le **hello.html** pour dire bonjour à l'utilisateur actuel et contenez un formulaire de déconnection. Mettre à jour **hello.html** comme suit :

```

1 <!DOCTYPE html>
2 <html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org"
3   xmlns:sec="https://www.thymeleaf.org/thymeleaf-extras-springsecurity3">
4 <head>
5 <meta charset="ISO-8859-1">
6 <!-- Defines metadata information for the document -->
7 </head>
8 <body>
9     <h3 th:inline="text">Hello [[${#httpServletRequest.remoteUser}]]!</h3>
10    <form th:action="@{/logout}" method="post">
11        <input type="submit" value="Déconnexion" />
12    </form>
13 </body>
14 </html>

```

Nous affichons le **nom d'utilisateur** en utilisant l'intégration de Spring Security avec **HttpServletRequest#getRemoteUser()**. Le formulaire « **Déconnexion** » soumet une requête POST à **/logout**. Une fois la déconnexion réussie, il redirige l'utilisateur vers **/login?logout**.

Maintenant que nous avons sécurisé notre page **hello.html**, nous souhaitons d'avoir un login et mot de passe propre à nous.

Pour cela, nous devons ajouter de la configuration des utilisateurs à la classe **WebSecurityConfig**.

- Modifier la classe **WebSecurityConfig** comme suit :

```

17 @Override
18 protected void configure(HttpSecurity http) throws Exception {
19     http
20         .authorizeRequests()
21             .antMatchers("/").permitAll()
22             .anyRequest().authenticated()
23             .and()
24         .formLogin()
25             .loginPage("/login")
26             .permitAll()
27             .and()
28         .logout()
29             .permitAll();
30 }
31
32 @Bean
33 @Override
34 public UserDetailsService userDetailsService() {
35     UserDetails user =
36         User.withDefaultPasswordEncoder()
37             .username("user")
38             .password("password")
39             .roles("USER")
40             .build();
41
42     return new InMemoryUserDetailsManager(user);
43 }
44
45 }

```

La méthode `userDetailsService()` configure un store d'utilisateurs en mémoire avec un seul utilisateur. Cet utilisateur reçoit un nom d'utilisateur « **user** », un mot de passe « **password** » et un rôle de « **USER** ».

## Exploration

- Mettez le contenu de la méthode `configure(HttpSecurity http)` de la classe `WebSecurityConfig` en commentaire.
- Ajoutez le texte suivant au corps de la fonction :

```
http.authorizeRequests().anyRequest().authenticated().and().formLogin().and().httpBasic();
```

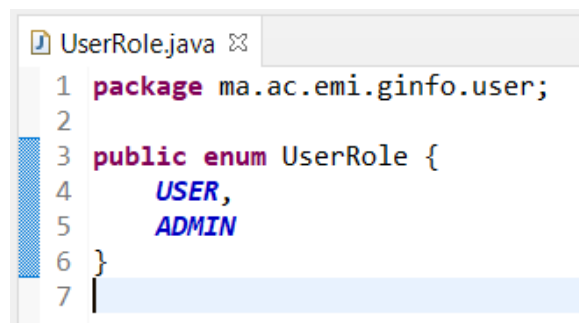
- Analysez la situation.
- Interceptez la requête et commentez le résultat.
- Consulter l'url suivant : <https://docs.spring.io/spring-security/site/docs/5.4.5/api/>
- Lire quelques méthodes de la classe `WebSecurityConfigurerAdapter`.

## Configurer une base de données pour l'authentification

- Remettre le contenu de la méthode `configure(HttpSecurity http)` de la classe `WebSecurityConfig` à sa version initiale.
- Décommenter les autres dépendances du fichier `POM.XML`
- Créer une base de données MySQL nommé : **ProjetPI**
- Modifier le fichier `application.properties` comme suit :

```
1 server.error.include-message = always
2 server.error.include-binding-errors: always
3
4 spring.datasource.username = kabbaaj
5 spring.datasource.password = toor@1234
6 spring.datasource.url = jdbc:mysql://localhost:3306/ProjetPI
7 spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
8 spring.jpa.hibernate.ddl-auto = Update
9 spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL8Dialect
10 spring.jpa.properties.hibernate.format_sql: true
11 spring.jpa.show-sql: true
```

- Créer le sous-package « **ma.ac.emi.ginfo.user** ».
- Créer une **énumération** nommée : **UserRole** (la liste des rôles supportée dans l'application).



```
1 package ma.ac.emi.ginfo.user;
2
3 public enum UserRole {
4     USER,
5     ADMIN
6 }
7
```

- Créer la classe **User** et faire l'implémenter l'interface **UserDetails**. Cette classe fournit les informations de base d'un utilisateur.

```

1 package ma.ac.emi.ginfo.user;
2
3 import org.springframework.security.core.GrantedAuthority;
4
5 @Entity
6 @Table(
7     name="USER",
8     uniqueConstraints=
9         @UniqueConstraint(columnNames={"email"})
10 )
11
12 public class User implements UserDetails {
13
14     private static final long serialVersionUID = 2593823355758855652L;
15
16     @SequenceGenerator(
17         name = "user_sequence",
18         sequenceName = "user_sequence",
19         allocationSize = 1
20     )
21     @Id
22     @GeneratedValue(
23         strategy = GenerationType.SEQUENCE,
24         generator = "user_sequence"
25     )
26     private Long id;
27     @NotNull
28     private String firstName;
29     @NotNull
30     private String lastName;
31     @NotNull
32     private String email;
33     @NotNull
34     private String password;
35     @Enumerated(EnumType.STRING)
36     private UserRole userRole;
37     private Boolean locked = false;
38     private Boolean enabled = false;
39
40     public User() {
41         super();
42     }
43
44     public User(String firstName,
45                 String lastName,
46                 String email,
47                 String password,
48                 UserRole userRole) {
49         this.firstName = firstName;
50         this.lastName = lastName;
51         this.email = email;
52         this.password = password;
53         this.userRole = userRole;
54     }
55
56     @Override
57     public Collection<? extends GrantedAuthority> getAuthorities() {
58         SimpleGrantedAuthority authority =
59             new SimpleGrantedAuthority(userRole.name());
60         return Collections.singletonList(authority);
61     }
62
63     @Override
64     public String getPassword() {
65         return password;
66     }
67
68     @Override
69     public String getUsername() {
70         return email;
71     }
72
73 }

```

- Générer les **getters** et les **setters** et **toString**.
- Générer les méthodes **equals** et **hashCode** afin qu'elle utilise l'attribut **email**.
- Créer l'interface **UserRepository** qui étend l'interface **JpaRepository**. C'est une classe de service. Cette classe est responsable sur le mapping objet relationnelle. Autrement dit, c'est elle enregistre un objet dans la page et inversement remplir une liste d'objet depuis les enregistrements relatifs à une requête SQL



```

UserRepository.java
1 package ma.ac.emi.ginfo.user;
2
3 import org.springframework.data.jpa.repository.JpaRepository;
4
5 @Repository
6 @Transactional(readOnly = true)
7 public interface UserRepository
8     extends JpaRepository<User, Long> {
9
10     Optional<User> findByEmail(String email);
11
12     @Transactional
13     @Modifying
14     @Query("UPDATE User a " +
15           "SET a.enabled = TRUE WHERE a.email = ?1")
16     int enableUser(String email);
17 }
18
19
20

```

- Créer la classe **UserService** qui implémente l'interface **UserDetailsService**. Cette classe à la fois elle offre de service de haut niveau par rapport à la classe UserRepository et elle est utilisé dans tout le framework Spring Security en tant que classe DAO des utilisateurs.

```

User.java  UserRepository.java  UserRole.java  UserService.java
1 package ma.ac.emi.ginfo.user;
2
3 import org.springframework.security.core.userdetails.UserDetails;
4
5 @Service
6 public class UserService implements UserDetailsService {
7
8     private final static String USER_NOT_FOUND_MSG =
9         "user with email %s not found";
10
11     private final UserRepository userRepository;
12     private final BCryptPasswordEncoder bCryptPasswordEncoder;
13
14     public UserService(UserRepository userRepository, BCryptPasswordEncoder bCryptPasswordEncoder) {
15         super();
16         this.userRepository = userRepository;
17         this.bCryptPasswordEncoder = bCryptPasswordEncoder;
18     }
19
20     @Override
21     public UserDetails loadUserByUsername(String email)
22         throws UsernameNotFoundException {
23         return userRepository.findByEmail(email)
24             .orElseThrow(() ->
25                 new UsernameNotFoundException(
26                     String.format(USER_NOT_FOUND_MSG, email)));
27     }
28
29     public void signUpUser(User user) {
30         boolean userExists = userRepository
31             .findByEmail(user.getEmail())
32             .isPresent();
33
34         if (userExists) {
35             throw new IllegalStateException("email already taken");
36         }
37
38         String encodedPassword = bCryptPasswordEncoder.encode(user.getPassword());
39         user.setPassword(encodedPassword);
40         userRepository.save(user);
41     }
42
43     public int enableUser(String email) {
44         return userRepository.enableUser(email);
45     }
46 }
47
48

```

- Modifier la classe **WebSecurityConfig** afin d'utiliser une base de données pour récupérer les informations sur les comptes utilisateurs au lieu d'un sauvegarde en mémoire. La classe **WebSecurityConfig** sera comme suit :

```

1 package ma.ac.emi.ginfo.security;
2
3 import org.springframework.context.annotation.Bean;
4
5
6
7
8
9
10
11
12
13
14 @Configuration
15 @EnableWebSecurity
16 public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
17
18     public final BCryptPasswordEncoder bCryptPasswordEncoder;
19
20     public final UserService userService;
21
22     public WebSecurityConfig(BCryptPasswordEncoder bCryptPasswordEncoder, UserService userService) {
23         super();
24         this.bCryptPasswordEncoder = bCryptPasswordEncoder;
25         this.userService = userService;
26     }
27
28     @Override
29     protected void configure(HttpSecurity http) throws Exception {
30         http
31             .authorizeRequests()
32                 .antMatchers("/").permitAll()
33                 .anyRequest().authenticated()
34                 .and()
35             .formLogin()
36                 .loginPage("/login")
37                 .permitAll()
38                 .and()
39             .logout()
40                 .permitAll();
41         // http.authorizeRequests().anyRequest().authenticated().and().formLogin().and().httpBasic();
42     }
43
44     @Override
45     protected void configure(AuthenticationManagerBuilder auth) throws Exception {
46         auth.authenticationProvider(daoAuthenticationProvider());
47     }
48
49     @Bean
50     public DaoAuthenticationProvider daoAuthenticationProvider() {
51         DaoAuthenticationProvider provider =
52             new DaoAuthenticationProvider();
53         provider.setPasswordEncoder(bCryptPasswordEncoder);
54         provider.setUserDetailsService(userService);
55         return provider;
56     }
57
58 }
59
60 @Configuration
61 class PasswordEncoder {
62
63     @Bean
64     public BCryptPasswordEncoder bCryptPasswordEncoder() {
65         return new BCryptPasswordEncoder();
66     }
67 }
68

```

- Enfin, enregistrons quelques comptes dans la base de données. Pour cela, on va modifier la classe **SecurityApplication**.

```
User.java  UserRepository.java  UserRole.java  UserService.java  WebSecurityConfig.java  *SecurityApplication.java
3* import org.springframework.beans.factory.annotation.Autowired;
14
15 @SpringBootApplication
16 public class SecurityApplication {
17
18     @Autowired
19     UserService userService;
20
21     public static void main(String[] args) {
22         SpringApplication.run(SecurityApplication.class, args);
23     }
24
25     @Bean
26     CommandLineRunner commandLineRunner(ApplicationContext sc) {
27         return args -> {
28             User u = new User("Mohammed Issam", "KABBAJ", "kabbaj@emi.ac.ma", "toor@1234", UserRole.ADMIN);
29             u.setEnabled(true);
30             u.setLocked(false);
31             userService.signUpUser(u);
32             System.err.println(u);
33             u = new User("Mohammed", "KABBAJ", "kabbaj.m@gmail.com", "toor@1234", UserRole.USER);
34             u.setEnabled(true);
35             u.setLocked(false);
36             userService.signUpUser(u);
37             System.err.println(u);
38             u = new User("Issam", "KABBAJ", "kabbajemi@gmail.com", "toor@1234", UserRole.ADMIN);
39             u.setEnabled(false);
40             u.setLocked(false);
41             userService.signUpUser(u);
42             System.err.println(u);
43             u = new User("Issam mohammed", "KABBAJ", "m.kabbaj@gmail.Com", "toor@1234", UserRole.ADMIN);
44             u.setEnabled(true);
45             u.setLocked(true);
46             userService.signUpUser(u);
47             System.err.println(u);
48         };
49     }
50 }
```