Cours Inf3522 - Développement d'Applications JEE

Lab_5 : Sécurisation

Ce lab explique comment documenter, sécuriser et tester votre backend Spring Boot. Sécuriser votre backend est une partie cruciale du développement de code. Nous utiliserons l'application backend que nous avons créés dans les labs précédents.

Sécurisation du backend

Cette partie explique comment sécuriser votre backend Spring Boot. Sécuriser votre backend est une partie cruciale du développement de code. Il est essentiel pour protéger les données sensibles, se conformer aux réglementations et prévenir les accès non autorisés. Le backend gère souvent le processus d'authentification et d'autorisation des utilisateurs. Sécuriser correctement ces aspects garantit que seuls les utilisateurs autorisés peuvent accéder à l'application et effectuer des actions spécifiques. Nous utiliserons la même application que précédemment.

Dans cette partie, nous aborderons les sujets suivants :

- Comprendre Spring Security

- Sécuriser votre backend avec un JSON Web Token

- Sécurité basée sur les rôles

- Utiliser OAuth2 avec Spring Boot

Comprendre Spring Security

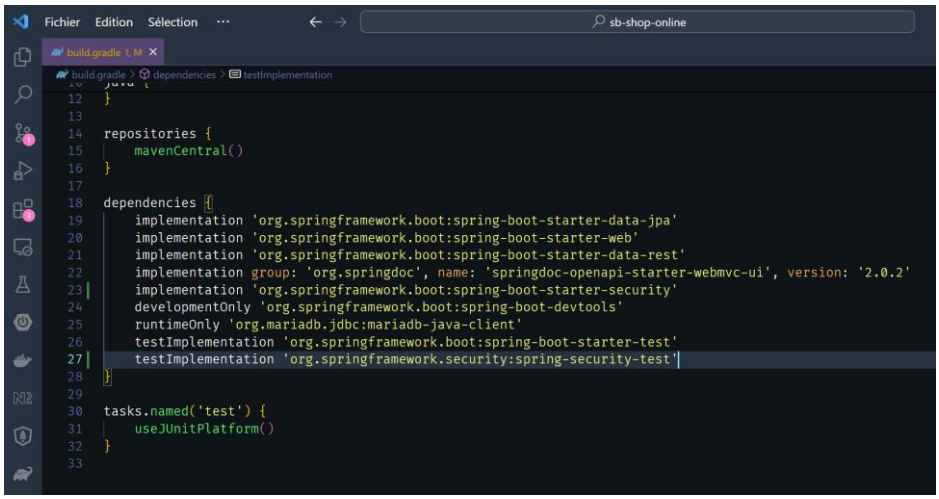
Spring Security (https://spring.io/projects/spring-security) fournit des services de sécurité pour les applications web Java. Le projet Spring Security a été lancé en 2003 et était auparavant appelé **Acegi Security System pour Spring**.

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.

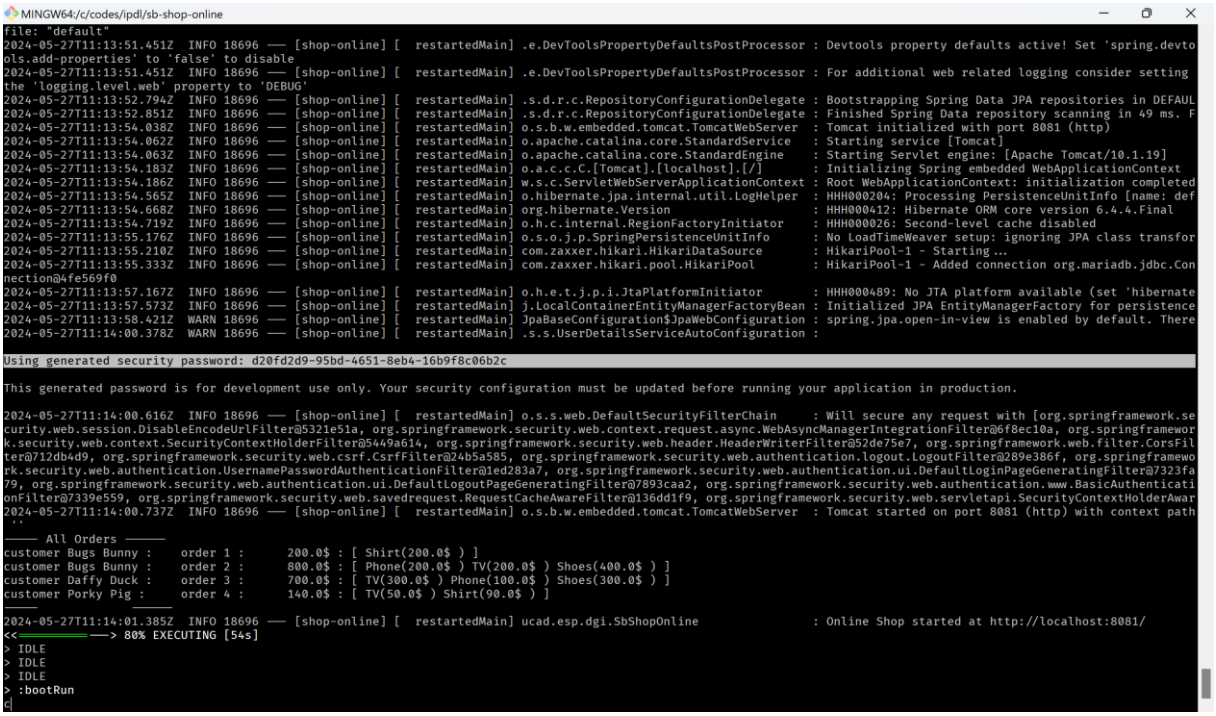
Récupérez le projet de démarrage depuis github : https://github.com/elbachir67/sb-car-5.git

Vous pouvez inclure Spring Security dans votre application en ajoutant les deux dépendances suivantes au fichier **build.gradle**. La première dépendance est pour l'application (Ligne 23) et la deuxième est pour les tests (Ligne 27) :



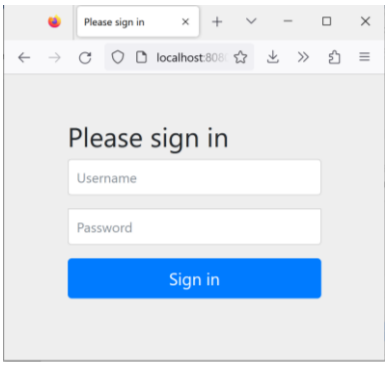
Stockage de l'utilisateur en mémoire

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 :



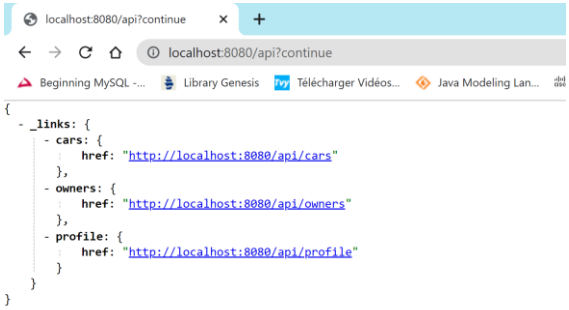
Si aucun mot de passe n'apparaît dans la console, essayez de redémarrer votre projet.

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 :



Configuration de la sécurité avec Spring Security

Pour configurer le comportement de Spring Security, nous devons ajouter une nouvelle classe de configuration pour Spring Security. Le fichier de configuration de sécurité permet de définir quelles URL ou motifs d'URL sont accessibles à quels rôles ou utilisateurs. Vous pouvez également définir le mécanisme d'authentification, le processus de connexion, la gestion des sessions, etc.

Créez une nouvelle classe appelée SecurityConfig dans le package racine de votre application (fst.dmi.cardatabase). Le code source suivant montre la structure de la classe de configuration de sécurité :

```
package fst.dmi.cardatabase;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

@Configuration
@EnableWebSecurity
public class SecurityConfig {
}
```

Les annotations @Configuration et @EnableWebSecurity désactivent la configuration de sécurité web par défaut, nous permettant ainsi de définir notre propre configuration dans cette classe.

Dans la méthode filterChain(HttpSecurity http), que nous verrons plus tard en action, nous pouvons définir quels points d'accès de notre application sont sécurisés et lesquels ne le sont pas. Pour l'instant, nous n'avons pas besoin de cette méthode, car nous pouvons utiliser les paramètres par défaut qui sécurisent tous les points d'accès.

Nous pouvons également ajouter des utilisateurs en mémoire à notre application en utilisant InMemoryUserDetailsManager de Spring Security, qui implémente UserDetailsService. Cela nous permet d'implémenter une authentification utilisateur/mot de passe stockée en mémoire. Nous pouvons aussi utiliser PasswordEncoder pour encoder les mots de passe avec l'algorithme bcrypt.

Le code source suivant crée un utilisateur en mémoire avec un nom d'utilisateur "user", un mot de passe "password" et un rôle "USER" :

```
package fst.dmi.cardatabase;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;

@Configuration
@EnableWebSecurity
public class SecurityConfig {
  @Bean
  public InMemoryUserDetailsManager userDetailsService() {
    UserDetails user = User.builder().username("user").
    password(passwordEncoder().encode("password"))
    .roles("USER").build();
    return new InMemoryUserDetailsManager(user);
  }

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

Redémarrez maintenant l'application et vous pourrez tester l'authentification en utilisant l'utilisateur en mémoire.

L'utilisation d'utilisateurs en mémoire est acceptable en phase de développement, mais une application en production devrait stocker les utilisateurs dans une base de données.

Enregistrement des utilisateurs en base de données

Pour enregistrer les utilisateurs dans une base de données, nous devons créer une classe d'entité et un repository. Les mots de passe ne doivent jamais être stockés en clair dans la base de données. Si une base de données contenant des mots de passe est compromise, les attaquants pourraient récupérer les mots de passe directement en clair.

Spring Security propose plusieurs algorithmes de hachage, comme bcrypt, que nous pouvons utiliser pour chiffrer les mots de passe. Voici les étapes pour l'implémentation :

1. Créer la classe AppUser

Dans le package fst.dmi.cardatabase.domain, créez une nouvelle classe appelée AppUser.

```
package fst.dmi.cardatabase.domain;
import jakarta.persistence.*;

@Entity
public class AppUser {
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   @Column(nullable = false, updatable = false)
   private Long id;

   @Column(nullable = false, unique = true)
   private String username;

   @Column(nullable = false)
   private String password;

   @Column(nullable = false)
   private String role;

   // Constructeurs, getters et setters
}
```

2. Créer le repository AppUserRepository

Dans le package domain, créez une nouvelle interface appelée AppUserRepository.

```
package fst.dmi.cardatabase.domain;
import java.util.Optional;
import org.springframework.data.repository.CrudRepository;

public interface AppUserRepository extends CrudRepository<AppUser, Long> {
   Optional<AppUser> findByUsername(String username);
}
```

3. Implémenter UserDetailsService

Spring Security utilise cette interface pour l'authentification et l'autorisation des utilisateurs.

Créez un package service et ajoutez-y une classe UserDetailsServiceImpl.

```
package fst.dmi.cardatabase.service;

import java.util.Optional;
import org.springframework.security.core.userdetails.*;
import org.springframework.stereotype.Service;
import fst.dmi.cardatabase.domain.AppUser;
import fst.dmi.cardatabase.domain.AppUserRepository;

@Service
public class UserDetailsServiceImpl implements UserDetailsService {
  private final AppUserRepository repository;

  public UserDetailsServiceImpl(AppUserRepository repository) {
    this.repository = repository;
  }

  @Override
  public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException
{
    Optional<AppUser> user = repository.findByUsername(username);
    if (user.isPresent()) {
      AppUser currentUser = user.get();
      return User.withUsername(username)
          .password(currentUser.getPassword())
          .roles(currentUser.getRole())
          .build();
    } else {
      throw new UsernameNotFoundException("Utilisateur non trouvé.");
    }
  }
}
```

4. Modifier SecurityConfig pour utiliser la base de données

Supprimez userDetailsService() et ajoutez la méthode configureGlobal pour utiliser les utilisateurs de la base de données.

```
package fst.dmi.cardatabase;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Bean;
import
org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBu
ilder;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import fst.dmi.cardatabase.service.UserDetailsServiceImpl;
import org.springframework.security.crypto.password.PasswordEncoder;
```

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
  private final UserDetailsServiceImpl userDetailsService;

  public SecurityConfig(UserDetailsServiceImpl userDetailsService) {
    this.userDetailsService = userDetailsService;
  }

  public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
    auth.userDetailsService(userDetailsService)
      .passwordEncoder(new BCryptPasswordEncoder());
  }

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

Désormais, le mot de passe doit être haché à l'aide de bcrypt avant d'être enregistré dans la base de données.

5. Enfin, nous pouvons enregistrer quelques utilisateurs de test dans la base de données en utilisant l'interface CommandLineRunner. Ouvrez le fichier CardatabaseApplication.java et injectez AppUserRepository dans la classe principale :

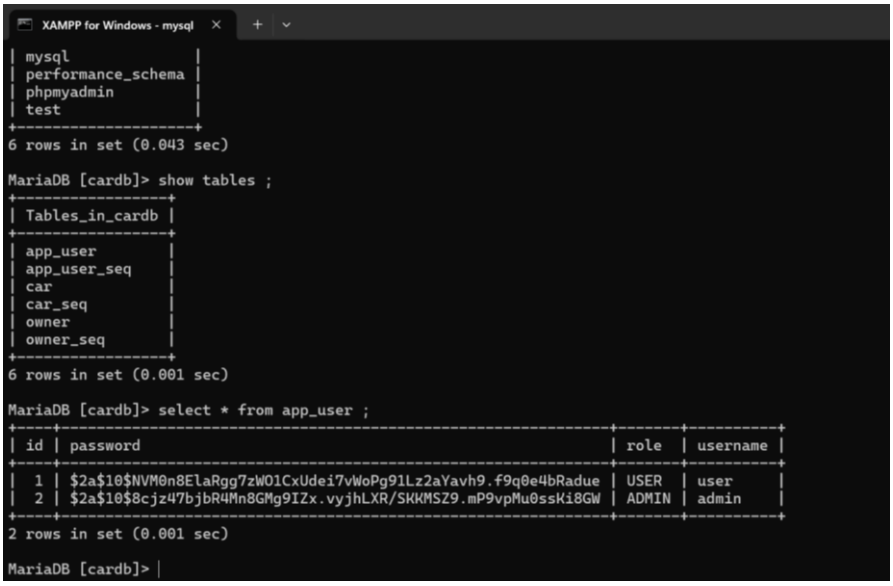
```
private final CarRepository repository;
private final OwnerRepository orepository;
private final AppUserRepository urepository;

public CardatabaseApplication(CarRepository repository,
                OwnerRepository orepository,
                AppUserRepository urepository) {
  this.repository = repository;
  this.orepository = orepository;
  this.urepository = urepository;
}
```

6. Enregistrons deux utilisateurs dans la base de données avec des mots de passe hachés avec bcrypt. Vous pouvez trouver des calculateurs ou générateurs bcrypt sur Internet. Ces générateurs vous permettent de saisir un mot de passe en clair, et ils produiront le hachage bcrypt correspondant :

```
@Override
public void run(String... args) throws Exception {
  // Ajouter des objets Owner et les enregistrer dans la base de données
  Owner owner1 = new Owner("John", "Johnson");
```

```
  Owner owner2 = new Owner("Mary", "Robinson");
  orepository.saveAll(Arrays.asList(owner1, owner2));

  repository.save(new Car("Ford", "Mustang", "Rouge", "ADF-1121",
            2023, 59000, owner1));
  repository.save(new Car("Nissan", "Leaf", "Blanc", "SSJ-3002",
            2020, 29000, owner2));
  repository.save(new Car("Toyota", "Prius", "Argent", "KKO-0212",
            2022, 39000, owner2));

  // Récupérer toutes les voitures et les afficher dans la console
  for (Car car : repository.findAll()) {
    logger.info(car.getBrand() + " " + car.getModel());
  }

  // Nom d'utilisateur : user, mot de passe : user
  urepository.save(new AppUser("user",
    "$2a$10$NVMon8ElaRgg7zWO1CxUdei7vWoPg91Lz2aYavh9.f9q0e4bRadue", "USER"));

  // Nom d'utilisateur : admin, mot de passe : admin
  urepository.save(new AppUser("admin",
    "$2a$10$8cjz47bjbR4Mn8GMg9IZx.vyjhLXR/SKKMSZ9.mP9vpMu0ssKi8GW", "ADMIN"));
}
```

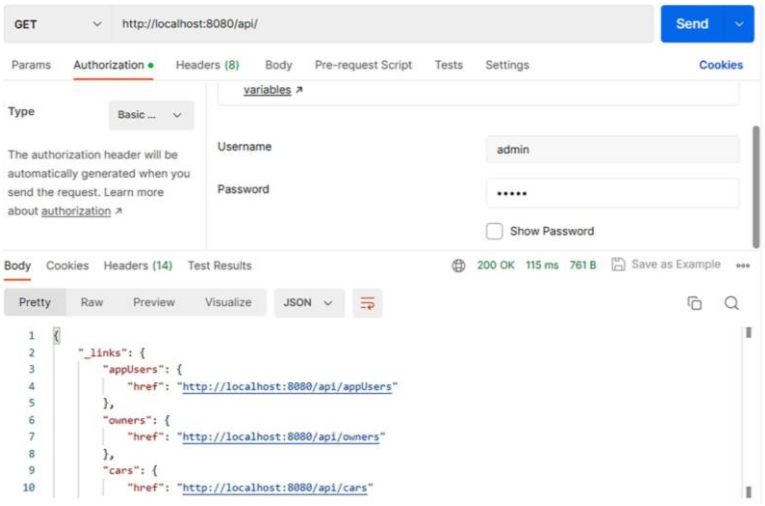
7. Après l'exécution de votre application, vous verrez qu'une table app_user a été créée dans la base de données et que deux enregistrements d'utilisateurs sont sauvegardés avec des mots de passe hachés, comme illustré dans la capture d'écran suivante :



Maintenant, vous devez redémarrer l'application et vous obtiendrez une erreur **401 Unauthorized** si vous essayez d'envoyer une requête **GET** à l'URL http://localhost:8080/api sans authentification. Vous devez vous authentifier pour pouvoir envoyer une requête avec succès.

La différence, par rapport à l'exemple précédent, est que nous utilisons désormais les utilisateurs de la base de données pour l'authentification.

8. Vous pouvez vous connecter en envoyant une requête **GET** au point de terminaison /api via un navigateur, ou utiliser **Postman** avec l'authentification de base, comme illustré dans la capture d'écran suivante :



Actuellement, nous pouvons récupérer la liste des utilisateurs en appelant le point de terminaison /api/appUsers dans notre service web RESTful, ce que nous voulons éviter.

9. Comme mentionné au **lab 4, Spring Data REST** génère par défaut un service web RESTful à partir de tous les **repositories publics**. Pour empêcher cela, nous pouvons utiliser l'attribut exported de l'annotation @RepositoryRestResource et le définir à false. Ainsi, le repository suivant ne sera **pas exposé** comme une ressource REST :

```
package fst.dmi.cardatabase.domain;

import java.util.Optional;
import org.springframework.data.repository.CrudRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

@RepositoryRestResource(exported = false)
public interface AppUserRepository extends CrudRepository<AppUser, Long> {
  Optional<AppUser> findByUsername(String username);
}
```

Maintenant, si vous redémarrez l'application et envoyez une requête **GET** au point de terminaison /api, vous verrez que l'endpoint /appUsers n'est plus accessible.

Ensuite, nous allons commencer à implémenter l'authentification en utilisant un **JSON Web Token (JWT)**.

Sécuriser votre backend avec un JSON Web Token