

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 créée 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

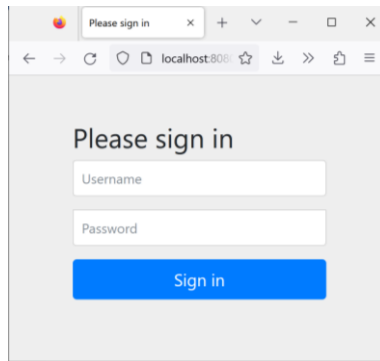
Spring Security (<https://spring.io/projects/spring-security>) fournit des services de sécurité pour les applications web Java. Le projet Spring Security a été lancé en 2003 et était auparavant appelé **Acegi Security System pour Spring**.

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.

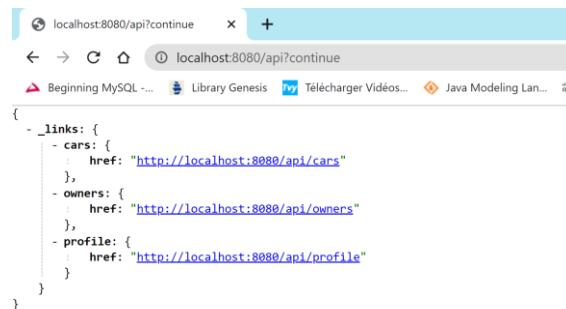
Récupérez le projet de démarrage depuis github : <https://github.com/elbachir67/sb-car-5.git>

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) :



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.AuthenticationManagerBuilder;
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));

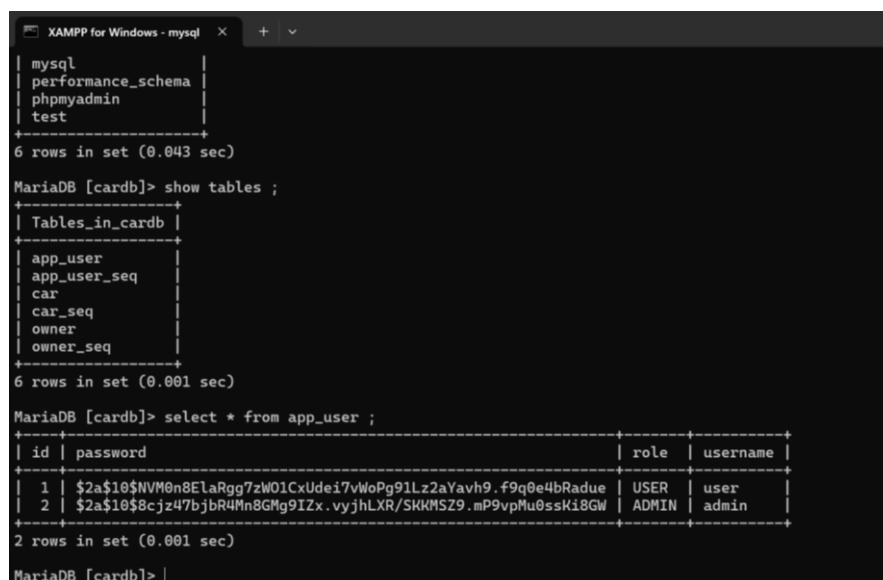
// Récupérer toutes les voitures et les afficher dans la console
for (Car car : repository.findAll()) {
    logger.info(car.getBrand() + " " + car.getModel());
}

// 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 :



```

XAMPP for Windows - mysql
mysql
+-----+
| mysql |
| performance_schema |
| phpmyadmin |
| test |
+-----+
6 rows in set (0.043 sec)

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

MariaDB [cardb]> select * from app_user ;
+----+-----+-----+-----+
| id | password | role | username |
+----+-----+-----+-----+
| 1 | $2a$10$NVMon8ElarGg7zWO1CxUdei7vWoPg91Lz2aYavh9.f9q0e4bRadue | USER | user |
| 2 | $2a$10$8cjz47bjbR4Mn8GMg9IZx.vyjhLXR/SKKMSZ9.mP9vpMu0ssKi8GW | ADMIN | admin |
+----+-----+-----+-----+
2 rows in set (0.001 sec)

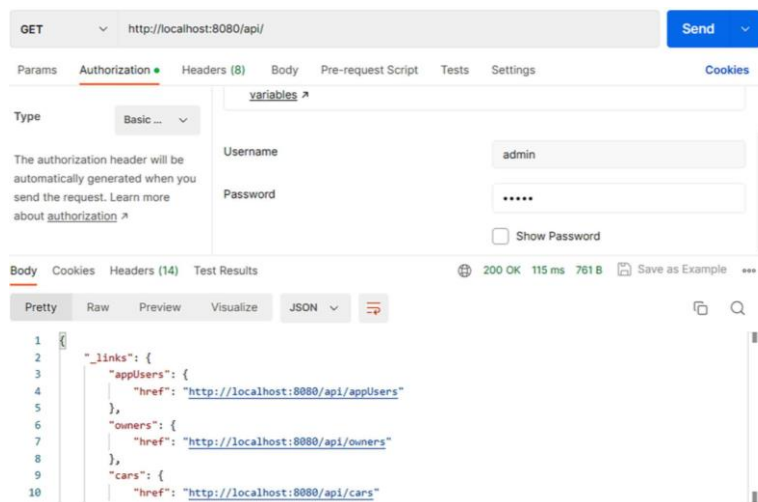
MariaDB [cardb]>

```

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

Dans la section précédente, nous avons abordé l'utilisation de l'authentification de base avec un service web RESTful.

L'authentification de base ne permet pas de gérer les jetons ni les sessions. Lorsqu'un utilisateur se connecte, ses identifiants sont envoyés avec chaque requête, ce qui peut entraîner des défis de gestion de session ainsi que des risques de sécurité. Cette méthode n'est pas adaptée lorsque nous développons notre propre frontend avec React. C'est pourquoi nous allons utiliser l'authentification par JSON Web Token (JWT) à la place (<https://jwt.io/>). Cela vous donnera également une idée plus détaillée de la configuration de Spring Security.

Les JWT sont couramment utilisés dans les API RESTful à des fins d'authentification et d'autorisation. Ils constituent une solution compacte pour l'authentification dans les applications web modernes. Un JWT est de petite taille, ce qui lui permet d'être envoyé dans l'URL, dans un paramètre POST ou dans l'en-tête d'une requête. Il contient aussi toutes les informations nécessaires sur l'utilisateur, telles que son nom d'utilisateur et son rôle.

Un JWT est composé de trois parties distinctes, séparées par des points : **xxxxx.yyyyy.zzzzz**. Ces parties sont définies comme suit :

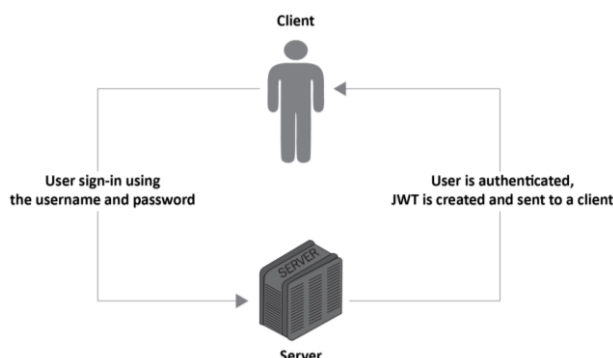
- **La première partie (xxxxx)** est l'en-tête, qui définit le type de jeton ainsi que l'algorithme de hachage utilisé.
- **La deuxième partie (yyyyy)** est la charge utile (*payload*), qui contient généralement les informations de l'utilisateur dans le cadre de l'authentification.
- **La troisième partie (zzzzz)** est la signature, utilisée pour vérifier que le jeton n'a pas été modifié en cours de route.

Une autre option pour sécuriser votre service web RESTful est **OAuth 2** (<https://oauth.net/2/>). OAuth2 est la norme industrielle pour l'autorisation et peut être facilement intégré aux applications Spring Boot. Une section ultérieure du lab vous donnera une introduction à son utilisation dans vos applications.

Voici un exemple de JWT :

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWliOiIxMjMoNTY3ODtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTEzMjM5MDIyfQ.
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

Le diagramme suivant illustre de manière simplifiée le processus d'authentification utilisant un JWT.



Après une authentification réussie, les requêtes envoyées par le client doivent toujours contenir le JWT reçu lors de l'authentification.

Nous allons utiliser **jjwt** (<https://github.com/jwt/jjwt>), qui est la bibliothèque JWT pour Java et Android permettant de créer et d'analyser les JWT.

Par conséquent, nous devons ajouter les dépendances suivantes au fichier **build.gradle** (Lignes 23, 24):

```
18 dependencies {
19     implementation 'org.springframework.boot:spring-boot-starter-web'
20     implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
21     implementation 'org.springframework.boot:spring-boot-starter-data-rest'
22     implementation 'org.springframework.boot:spring-boot-starter-security'
23     implementation 'io.jsonwebtoken:jjwt-api:0.11.5'
24     runtimeOnly 'io.jsonwebtoken:jjwt-impl:0.11.5', 'io.jsonwebtoken:jjwt-jackson:0.11.5'
25     implementation group: 'org.springdoc', name: 'springdoc-openapi-starter-webmvc-ui', version: '2.0.2'
26     developmentOnly 'org.springframework.boot:spring-boot-devtools'
27     testImplementation 'org.springframework.boot:spring-boot-starter-test'
28     runtimeOnly 'org.mariadb.jdbc:mariadb-java-client'
29     testImplementation 'org.springframework.security:spring-security-test'
30 }
```

Activation de l'authentification JWT dans notre backend

Sécurisation de la connexion

Nous allons commencer par implémenter la fonctionnalité de connexion :

1. Création d'une classe pour générer et vérifier un JWT signé

Créez une nouvelle classe appelée **JwtService** dans le package `fst.dmi.cardatabase.service`.

Au début de la classe, nous définirons quelques constantes :

`EXPIRATIONTIME` définit la durée de validité du jeton en millisecondes.

`PREFIX` définit le préfixe du jeton, et le schéma "Bearer" est généralement utilisé.

Un JWT est envoyé dans l'en-tête Authorization, qui a la structure suivante :

Authorization: Bearer <token>

Voici le code source de la classe **JwtService** :

```
package fst.dmi.cardatabase.service;
import org.springframework.stereotype.Component;

@Component
public class JwtService {
    static final long EXPIRATIONTIME = 86400000; // 1 jour en ms (réduire en
    production)
    static final String PREFIX = "Bearer";
}
```

N'oubliez pas de rafraîchir le projet Gradle après avoir mis à jour les dépendances.

2. Génération et validation d'un JWT

- Nous allons utiliser la bibliothèque **jjwt** pour créer une clé secrète avec la méthode `secretKeyFor`.
- **En production, la clé secrète doit être lue depuis la configuration de l'application.**
- La méthode `getToken` génère et retourne le JWT.
- La méthode `getAuthUser` récupère le token depuis l'en-tête `Authorization`, le vérifie et extrait le nom d'utilisateur.

// Code complet de la classe Jwt Service.java

```
package fst.dmi.cardatabase.service;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import io.jsonwebtoken.security.Keys;
import java.security.Key;
import org.springframework.http.HttpHeaders;
import org.springframework.stereotype.Component;
import jakarta.servlet.http.HttpServletRequest;
import java.util.Date;

@Component
public class JwtService {
    static final long EXPIRATIONTIME = 86400000; // 1 jour en ms
    static final String PREFIX = "Bearer";

    // Générer une clé secrète (uniquement pour démonstration)
    static final Key key = Keys.secretKeyFor(SignatureAlgorithm.HS256);

    // Générer un JWT signé
    public String getToken(String username) {
        return Jwts.builder()
            .setSubject(username)
            .setExpiration(new Date(System.currentTimeMillis() + EXPIRATIONTIME))
            .signWith(key)
            .compact();
    }

    // Récupérer un token depuis l'en-tête Authorization et extraire l'utilisateur
    public String getAuthUser(HttpServletRequest request) {
        String token = request.getHeader(HttpHeaders.AUTHORIZATION);
        if (token != null) {
            String user = Jwts.parserBuilder()
                .setSigningKey(key)
                .build()
                .parseClaimsJws(token.replace(PREFIX, ""))
                .getBody()
                .getSubject();
        }
    }
}
```

```

        if (user != null)
            return user;
    }
    return null;
}
}

```

3. Création d'un enregistrement pour stocker les identifiants d'authentification

Java 14 a introduit les **records**, qui permettent de créer des classes de données sans écrire de code répétitif.

Créez un **record** nommé `AccountCredentials` dans le package `fst.dmi.cardatabase.domain` :

```

package fst.dmi.cardatabase.domain;
public record AccountCredentials(String username, String password) {}

```

4. Implémentation du contrôleur pour gérer la connexion

La connexion se fait via une requête **POST** à l'endpoint `/login`, avec le nom d'utilisateur et le mot de passe dans le corps de la requête.

Créez la classe `LoginController` dans le package `fst.dmi.cardatabase.web` :

```

package fst.dmi.cardatabase.web;
import org.springframework.http.HttpHeaders;
import org.springframework.http.ResponseEntity;
import org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.authentication.UsernamePasswordAuthenticationTo
ken;
import org.springframework.security.core.Authentication;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RestController;
import com.packt.cardatabase.domain.AccountCredentials;
import com.packt.cardatabase.service.JwtService;

```

```

@RestController
public class LoginController {
    private final JwtService jwtService;
    private final AuthenticationManager authenticationManager;

    public LoginController(JwtService jwtService, AuthenticationManager
authenticationManager) {
        this.jwtService = jwtService;
        this.authenticationManager = authenticationManager;
    }

```

```

    @PostMapping("/login")

```

```

    public ResponseEntity<?> getToken(@RequestBody AccountCredentials
credentials) {
        UsernamePasswordAuthenticationToken creds = new
UsernamePasswordAuthenticationToken(
            credentials.username(), credentials.password());

        Authentication auth = authenticationManager.authenticate(creds);

        // Générer le JWT
        String jwt = jwtService.getToken(auth.getName());

        // Construire la réponse avec le JWT dans l'en-tête Authorization
        return ResponseEntity.ok()
            .header(HttpHeaders.AUTHORIZATION, "Bearer " + jwt)
            .header(HttpHeaders.ACCESS_CONTROL_EXPOSE_HEADERS,
"Authorization")
            .build();
    }
}

```

5. Ajout du gestionnaire d'authentification dans la configuration de sécurité

Remplacez le code suivant par le code de la classe SecurityConfig :

```

package fst.dmi.cardatabase;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.config.annotation.authentication.configuration.Auth
enticationConfiguration;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecu
rity;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import fst.dmi.cardatabase.service.UserDetailsServiceImpl;

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

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

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

```

```

    }

    @Bean
    public AuthenticationManager
    authenticationManager(AuthenticationConfiguration authConfig) throws Exception
    {
        return authConfig.getAuthenticationManager();
    }
}

```

6. Configuration de Spring Security avec SecurityFilterChain

Ajoutez la méthode `filterChain` dans `SecurityConfig` pour sécuriser les endpoints :

```

@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.csrf(csrf -> csrf.disable())
        .sessionManagement(session ->
            session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .authorizeHttpRequests(auth -> auth
            .requestMatchers(HttpMethod.POST, "/login").permitAll()
            .anyRequest().authenticated());
    return http.build();
}

```

Vous pourrez avoir besoin des imports ci-dessous :

```

import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.http.HttpMethod;
import org.springframework.context.annotation.Bean;

```

7. Test de la connexion avec Postman

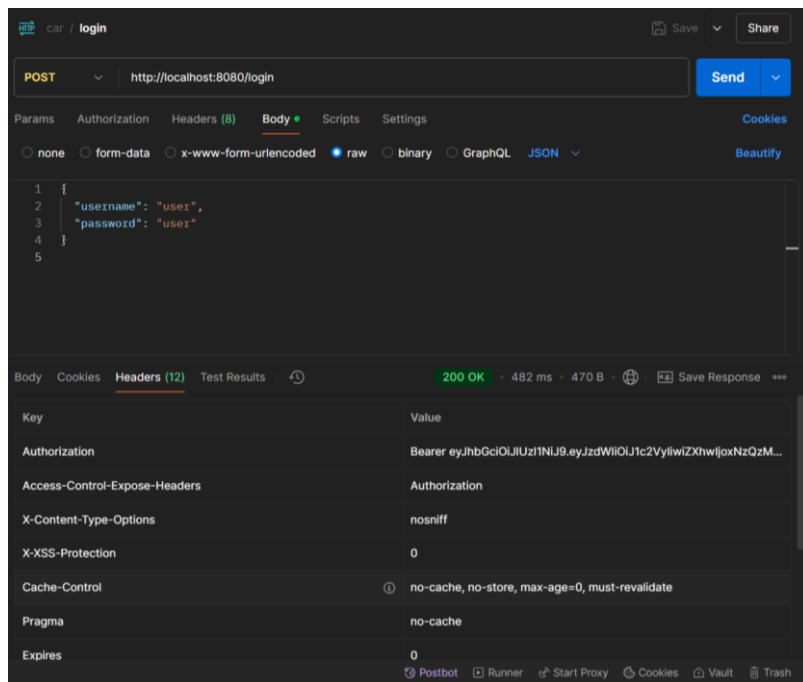
Envoyez une requête **POST** à `http://localhost:8080/login` avec les identifiants :

```

{
  "username": "user",
  "password": "user"
}

```

Le **token JWT** doit être présent dans l'en-tête **Authorization** de la réponse (commençant par **Bearer** ...).



Vous pouvez également tester la connexion avec un mot de passe incorrect pour vérifier qu'aucun token n'est retourné.

Sécurisation des autres requêtes

Nous avons maintenant finalisé l'étape de connexion et allons passer à l'authentification des autres requêtes entrantes. Dans ce processus, nous utilisons des filtres qui nous permettent d'exécuter certaines opérations avant qu'une requête n'atteigne le contrôleur ou avant qu'une réponse ne soit envoyée au client. Les étapes suivantes démontrent le reste du processus d'authentification :

1. Création d'un filtre pour authentifier les requêtes entrantes

Nous allons utiliser une classe de filtre pour authentifier toutes les requêtes entrantes. Créez une nouvelle classe appelée `AuthenticationFilter` dans le package racine. Cette classe étend l'interface `OncePerRequestFilter` de Spring Security, qui fournit la méthode `doFilterInternal` dans laquelle nous implémentons notre logique d'authentification.

Nous devons injecter une instance de `JwtService` dans la classe du filtre, car elle est nécessaire pour vérifier un jeton provenant de la requête. `SecurityContextHolder` est utilisé pour stocker les détails de l'utilisateur authentifié. Voici le code correspondant :

```

package fst.dmi.cardatabase;
import org.springframework.http.HttpHeaders;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;
import fst.dmi.cardatabase.service.JwtService;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;

```



```

import jakarta.servlet.http.HttpServletResponse;

@Component
public class AuthenticationFilter extends OncePerRequestFilter {
    private final JwtService jwtService;

    public AuthenticationFilter(JwtService jwtService) {
        this.jwtService = jwtService;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain)
        throws ServletException, java.io.IOException {
        // Récupérer le token depuis l'en-tête Authorization
        String jws = request.getHeader(HttpHeaders.AUTHORIZATION);
        if (jws != null) {
            // Vérifier le token et obtenir l'utilisateur
            String user = jwtService.getAuthUser(request);
            // Authentifier l'utilisateur
            Authentication authentication =
                new UsernamePasswordAuthenticationToken(user, null,
java.util.Collections.emptyList());
            SecurityContextHolder.getContext().setAuthentication(authentication);
        }
        filterChain.doFilter(request, response);
    }
}

```

2. Ajout du filtre à la configuration de Spring Security

Nous devons maintenant ajouter notre classe de filtre à la configuration de Spring Security. Ouvrez la classe SecurityConfig et injectez la classe AuthenticationFilter que nous venons d'implémenter, comme indiqué dans le code suivant (éviter la duplication de code, attention à la copie-colle) :

```

private final UserDetailsServiceImpl userDetailsService;
private final AuthenticationFilter authenticationFilter;

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

```

3. Modification de la méthode filterChain dans SecurityConfig

Modifiez la méthode `filterChain` dans la classe `SecurityConfig` et ajoutez les lignes de code suivantes.

```
// Importer la classe suivante
```

```
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;
```

```
// Modifier la méthode filterChain
```

@Bean

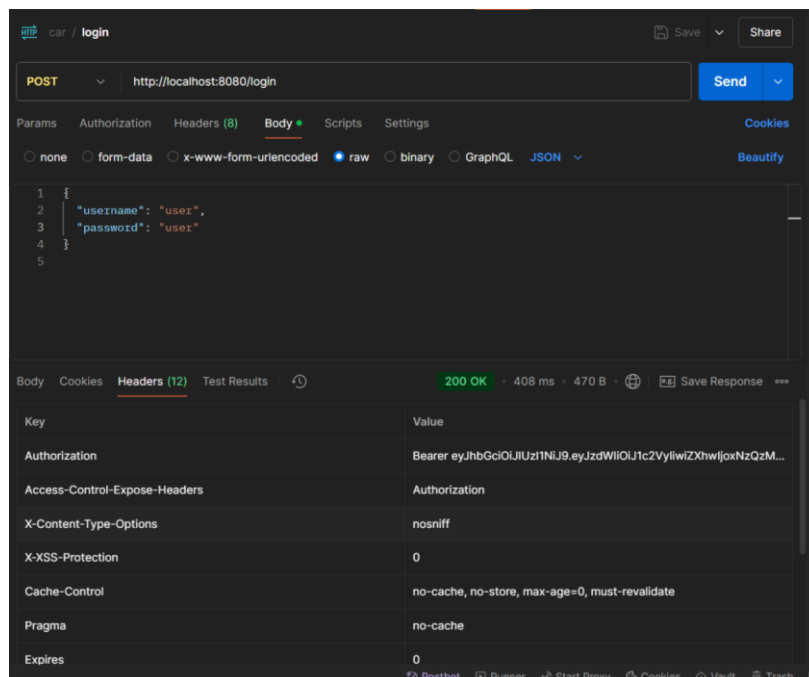
```
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.csrf(csrf -> csrf.disable())
        .sessionManagement(sessionManagement -> sessionManagement
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .authorizeHttpRequests(authorizeHttpRequests -> authorizeHttpRequests
            .requestMatchers(HttpMethod.POST, "/login").permitAll()
            .anyRequest().authenticated())
        .addFilterBefore(authenticationFilter, UsernamePasswordAuthenticationFilter.class);

    return http.build();
}
```

4. Test du workflow complet

Nous sommes maintenant prêts à tester l'ensemble du processus. Après avoir lancé l'application, commencez par effectuer une connexion en appelant l'endpoint `/login` via la méthode `POST`. En cas de succès, un `JWT` sera renvoyé dans l'en-tête `Authorization` de la réponse.

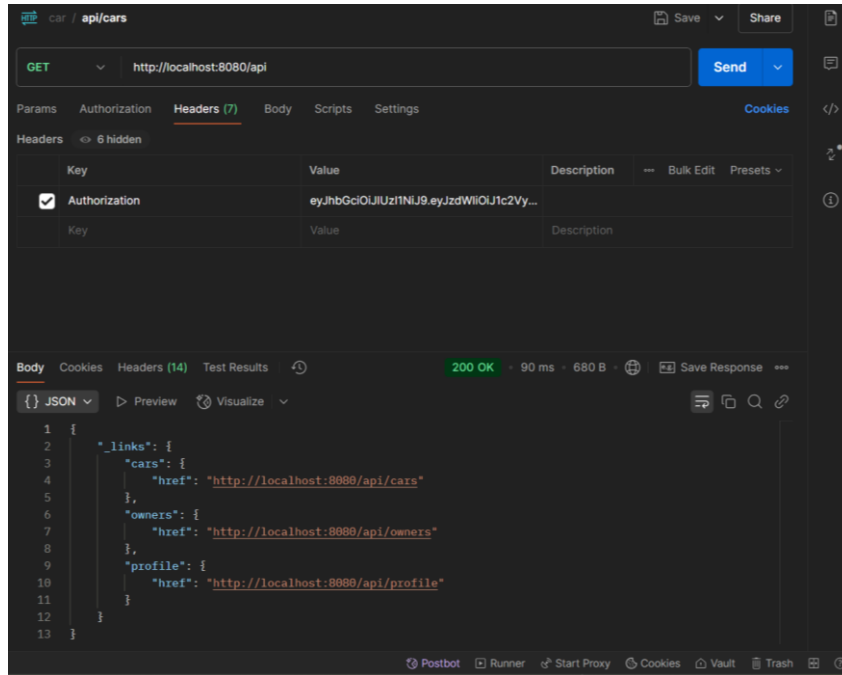
Assurez-vous d'ajouter un utilisateur valide dans le corps de la requête et de définir l'en-tête Content-Type sur application/json si ce n'est pas fait automatiquement par Postman.



5. Utilisation du token pour authentifier d'autres requêtes

Une fois connecté, vous pouvez appeler d'autres endpoints REST en envoyant le JWT reçu dans l'en-tête Authorization. Copiez le token depuis la réponse de connexion (sans le préfixe Bearer) et ajoutez-le dans l'en-tête Authorization des requêtes suivantes.

Par exemple, une requête GET vers /cars avec le token dans l'en-tête Authorization permettra d'accéder aux ressources protégées.



Avec ces étapes, nous avons sécurisé l'ensemble des requêtes en utilisant JWT pour l'authentification.

NB : Chaque fois que l'application est redémarrée, vous devez vous authentifier à nouveau, car un nouveau JWT est généré.

Le JWT n'est pas valide indéfiniment, car une date d'expiration lui a été attribuée. Dans notre cas, nous avons défini une durée d'expiration longue à des fins de démonstration. En production, cette durée devrait de préférence être de quelques minutes, en fonction du cas d'utilisation.

Gestion des exceptions

(A venir)