

Cours Inf3522 - Développement d'Applications N tiers

Lab 5 : Sécurisation

Ce lab explique comment sécuriser et tester votre backend Spring Boot. Sécuriser votre backend est une partie cruciale du développement de code. 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 :

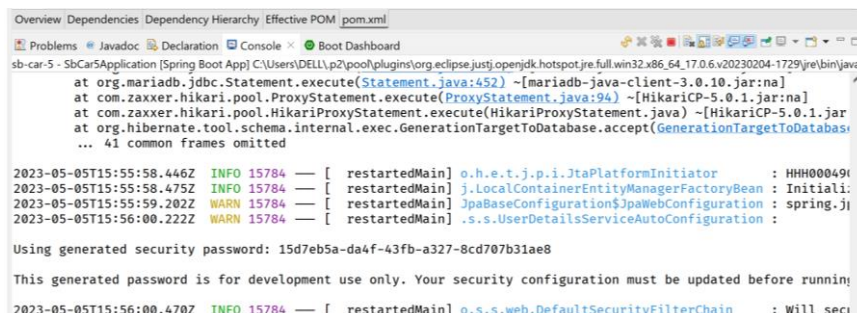
- Un **bean AuthenticationManager** avec un seul utilisateur en mémoire. Le nom d'utilisateur est « **user** » et le mot de passe est imprimé dans la sortie de la console.
- Des chemins ignorés pour les emplacements courants de ressources statiques, tels que **/css** et **/images**.
- Sécurité basique **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    <dependency>
52        <groupId>org.springframework.boot</groupId>
53        <artifactId>spring-boot-starter-security</artifactId>
54    </dependency>
55    <dependency>
56        <groupId>org.springframework.security</groupId>
57        <artifactId>spring-security-test</artifactId>
58        <scope>test</scope>
59    </dependency>
60</dependencies>
```

Stockage de l'utilisateur en mémoire

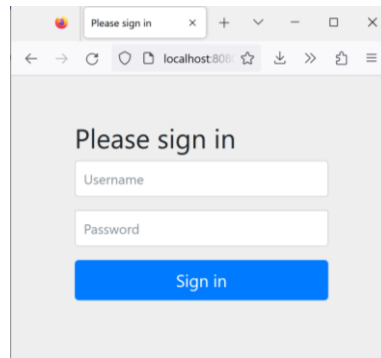
Lorsque vous démarrez votre application, vous pouvez voir dans la console que Spring Security a créé un utilisateur en mémoire avec un nom d'utilisateur « **user** ». Le mot de passe de l'utilisateur peut être vu dans la sortie de la console, comme illustré ici :



```
Overview | Dependencies | Dependency Hierarchy | Effective POM | pom.xml
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
```

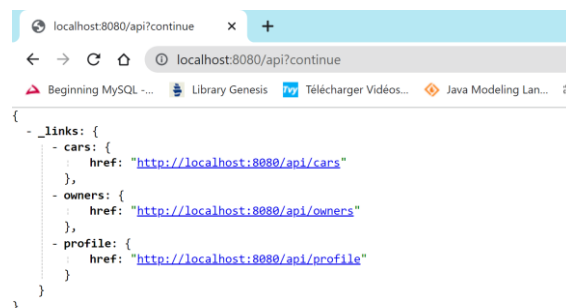
Si aucun mot de passe n'apparaît dans la console, essayez de redémarrer votre projet.

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 :

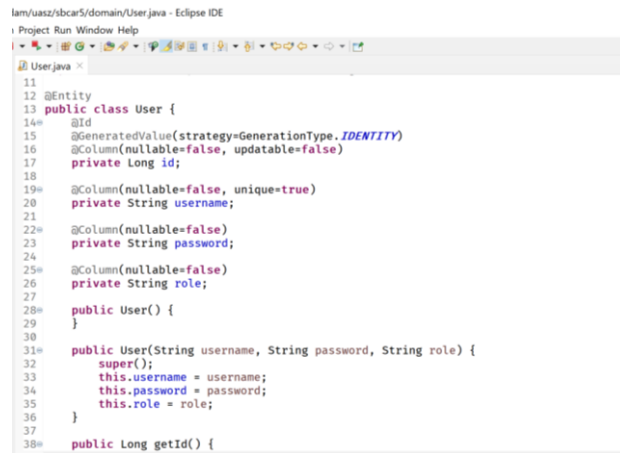


Stockage de l'utilisateur dans une Base de données

Pour enregistrer les utilisateurs dans la base de données, vous devez créer une classe d'entité **User** 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 :

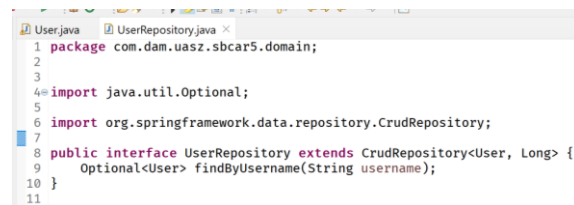


```

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 lab précédent, mais il existe une méthode de requête, **findByUsername**, dont nous aurons 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 :



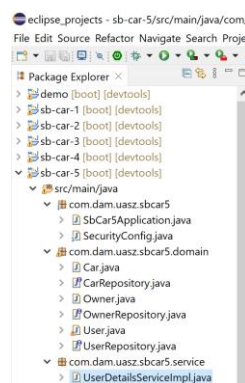
```

1 package com.dam.uaszbcar5.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 :



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.sbcar5.domain.User;
14 import com.dam.uas2.sbcar5.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.sbcar5;
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.sbcar5.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\$8cjz47bjbR4Mn8GMg9Izx.vyjhLXR/SKKMSZ9.mP9vpMuossKi8GW

\$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 en injectant le **UserRepository** : ligne 32 et 33, puis en créant 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$10$NM0n8ElarGg7zW01CxUdei7vWoPg91Lz2aYavh9.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 :

```

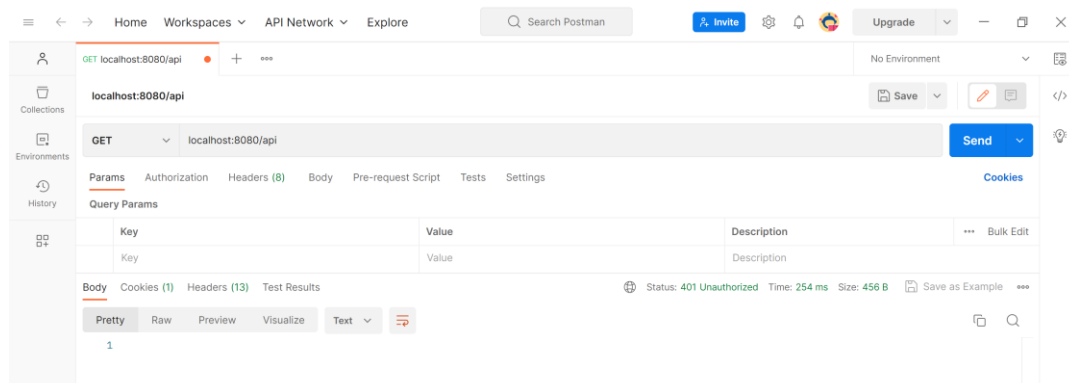
XAMPP for Windows - mysql -u root

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

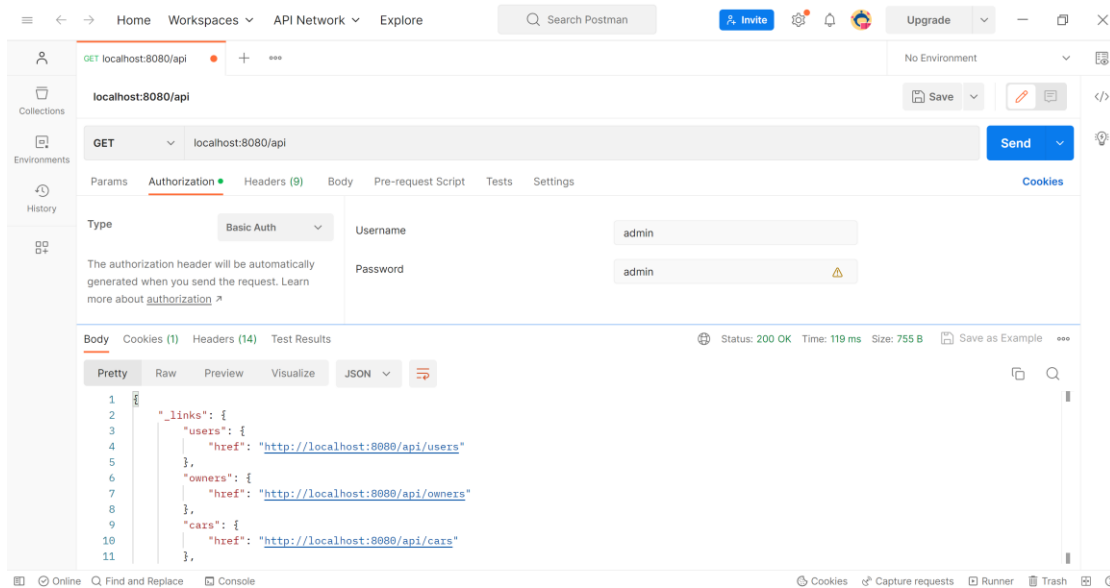
MariaDB [cardb]> select * from user ;
+-----+
| id | password | role | username |
+-----+
| 1  | $2a$10$NM0n8ElarGg7zW01CxUdei7vWoPg91Lz2aYavh9.f9q0e4bRadue | USER | user |
| 2  | $2a$10$8cjz47bjbR4Mn8GMg9IZx.vyjhLXR/SKKMSZ9.mP9vpMu0ssKi8GW | ADMIN | admin |
+-----+
2 rows in set (0.000 sec)

```

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**) :



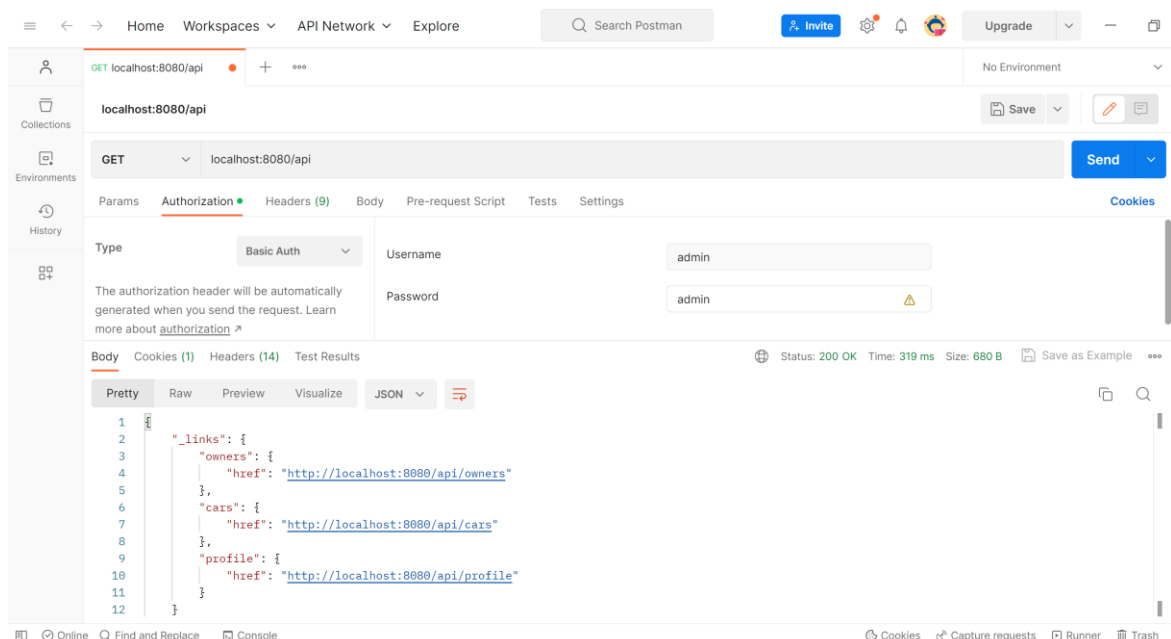
Maintenant, vous pouvez voir que nous obtenons les utilisateurs en appelant l'endpoint `/users` dans notre service web RESTful, et c'est quelque chose que nous voulons éviter. Comme mentionné précédemment, **Spring Data REST** génère par défaut un service web RESTful à partir de tous les référentiels 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 :



Sécuriser votre backend avec un jeton Web JSON (JWT)

Dans la section précédente, nous avons couvert comment utiliser l'authentification de base avec un service web RESTful. Cette méthode ne peut pas être utilisée lorsque nous développerons notre propre frontend avec **React**, nous allons donc utiliser l'authentification **JWT** à la place. Un **JWT** est une façon compacte de mettre en œuvre l'authentification dans les applications web modernes. Un JWT est vraiment petit en taille et peut donc être envoyé dans l'URL (Uniform Resource Locator), dans le paramètre POST ou à l'intérieur de l'en-tête. Il contient également toutes les informations nécessaires concernant l'utilisateur.

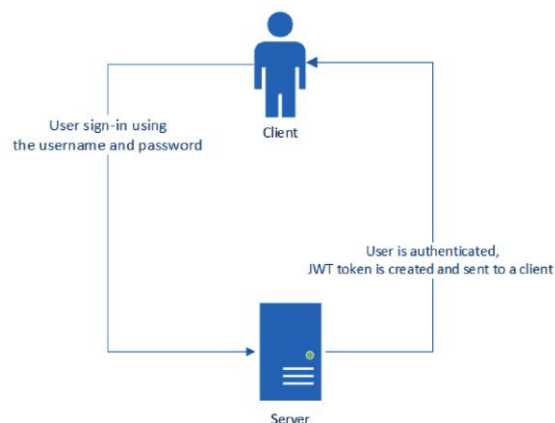
Un JWT contient trois parties différentes, séparées par des points : **xxxxx.yyyyy.zzzzz**. Ces parties sont réparties comme suit :

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

Voici un exemple de JWT :

```
eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJKb2UiZD.ipevRNuRP6Hf1G8cFKnmUPtypruR  
C4fc1DWtoLL62SYI
```

Le diagramme suivant montre une représentation simplifiée du processus d'authentification JWT :



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

Nous utiliserons la bibliothèque Java **jjwt** (<https://github.com/jwtk/jjwt>), qui est la bibliothèque JWT pour Java et Android. Par conséquent, nous devons ajouter la dépendance suivante au fichier **pom.xml**. La bibliothèque **jjwt** est utilisée pour créer et analyser les JWT :

```
39  
60  
61 <dependency>  
62 <groupId>io.jsonwebtoken</groupId>  
63 <artifactId>jjwt-api</artifactId>  
64 <version>0.11.2</version>  
65 </dependency>  
66 <dependency>  
67 <groupId>io.jsonwebtoken</groupId>  
68 <artifactId>jjwt-impl</artifactId>  
69 <version>0.11.2</version>  
70 <scope>runtime</scope>  
71 </dependency>  
72 <dependency>  
73 <groupId>io.jsonwebtoken</groupId>  
74 <artifactId>jjwt-jackson</artifactId>  
75 <version>0.11.2</version>  
76 <scope>runtime</scope>  
77 </dependency>  
</dependencies>
```


Les étapes suivantes démontrent comment activer l'authentification JWT dans notre backend.

Partie 1

Nous commencerons par la fonctionnalité de connexion:

1. Tout d'abord, nous créerons une classe qui génère et vérifie un JWT signé. Créez une nouvelle classe appelée **JwtService** dans le package **com.dam.uaszbcar5.service**.
Au début de la classe, nous définirons quelques constantes : **EXPIRATIONTIME** définit le temps d'expiration du jeton en millisecondes (ms), **PREFIX** définit le préfixe du jeton, et le schéma **Bearer** est généralement utilisé. Une clé secrète est créée à l'aide de la méthode **secretKeyFor** de la bibliothèque **jjwt**, et cela peut être utilisé pour la démonstration. Dans un environnement de production, vous devriez lire votre clé secrète à partir de la configuration de l'application. La méthode **getToken** génère et retourne le jeton. La méthode **getAuthUser** obtient le jeton de l'en-tête **Authorization** de la réponse.
Ensuite, nous utilisons la méthode **parserBuilder** fournie par la bibliothèque **jjwt** pour créer une instance de **JwtParserBuilder**. La méthode **setSigningKey** est utilisée pour spécifier une clé secrète pour la vérification du jeton. Enfin, nous utilisons la méthode **getSubject** pour obtenir le nom d'utilisateur. Le code source complet de **JwtService** peut être consulté ici:

```
sb-car-5/pom.xml  JwtService.java
1 package com.dam.uaszbcar5.service;
2 import io.jsonwebtoken.Jwts;
3 import io.jsonwebtoken.SignatureAlgorithm;
4 import io.jsonwebtoken.security.Keys;
5 import java.security.Key;
6 import org.springframework.http.HttpHeaders;
7 import org.springframework.stereotype.Component;
8 import jakarta.servlet.http.HttpServletRequest;
9 import java.util.Date;
10 @Component
11 public class JwtService {
12     static final long EXPIRATIONTIME = 86400000; // 1 day in ms
13     static final String PREFIX = "Bearer";
14     // Generate secret key. Only for the demonstration
15     // You should read it from the application configuration
16     static final Key key = Keys.secretKeyFor(SignatureAlgorithm.HS256);
17     // Generate JWT token
18     public String getToken(String username) {
19         String token = Jwts.builder()
20             .setSubject(username)
21             .setExpiration(new Date(System.currentTimeMillis() + EXPIRATIONTIME))
22             .signWith(key)
23             .compact();
24         return token;
25     }
26
27     // Get a token from request Authorization header,
28     // parse a token and get username
29     public String getAuthUser(HttpServletRequest request) {
30         String token = request.getHeader(HttpHeaders.AUTHORIZATION);
31
32         if (token != null) {
33             String user = Jwts.parserBuilder()
34                 .setSigningKey(key)
35                 .build()
36                 .parseClaimsJws(token.replace(PREFIX, ""))
37                 .getBody()
38                 .getSubject();
39
40             if (user != null)
41                 return user;
42         }
43
44         return null;
45     }
46 }
```

2. Ensuite, nous ajouterons une nouvelle classe Plain Old Java Object (**POJO**) simple pour stocker les informations d'identification pour l'authentification. Créez une nouvelle classe appelée **AccountCredentials** dans le package **com.dam.uaszbcar5.domain**. La classe a deux champs : **username** et **password**. Voici le code source de la classe. Cette classe n'a pas l'annotation **@Entity** car nous n'avons pas besoin de sauvegarder les informations d'identification dans la base de données :


```

sb-car-5/pom.xml  JwtService.java  AccountCredentials.java  LoginController.java
1 package com.dam.uaszbcar5.domain;
2
3 public class AccountCredentials {
4     private String username;
5     private String password;
6
7     public String getUsername() {
8         return username;
9     }
10
11    public void setUsername(String username) {
12        this.username = username;
13    }
14
15    public String getPassword() {
16        return password;
17    }
18
19    public void setPassword(String password) {
20        this.password = password;
21    }
22 }

```

- Maintenant, nous allons implémenter la classe du contrôleur pour la connexion. La connexion se fait en appelant l'endpoint **/login** en utilisant la méthode **POST** et en envoyant le nom d'utilisateur et le mot de passe dans le corps de la requête. Créez une classe appelée **LoginController** dans le package **com.dam.uaszbcar5.web**. Nous devons injecter une instance de **JwtService** dans la classe du contrôleur car elle est utilisée pour générer un JWT signé en cas de succès de la connexion.
- Ensuite, nous allons implémenter la méthode **getToken** qui gère la fonctionnalité de connexion, comme suit :

```

JwtService.java  UserDetailsServiceImpl.java  AccountCredentials.java  LoginController.java
1 package com.dam.uaszbcar5.web;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.http.HttpHeaders;
5 import org.springframework.http.ResponseEntity;
6 import org.springframework.security.authentication.AuthenticationManager;
7 import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
8 import org.springframework.security.core.Authentication;
9 import org.springframework.web.bind.annotation.PostMapping;
10 import org.springframework.web.bind.annotation.RequestBody;
11 import org.springframework.web.bind.annotation.RestController;
12
13 import com.dam.uaszbcar5.domain.AccountCredentials;
14 import com.dam.uaszbcar5.service.JwtService;
15
16
17 @RestController
18 public class LoginController {
19     @Autowired
20     private JwtService jwtService;
21
22     @Autowired
23     AuthenticationManager authenticationManager;
24
25     @PostMapping("/login")
26     public ResponseEntity<> getToken(@RequestBody AccountCredentials credentials) {
27         UsernamePasswordAuthenticationToken creds =
28             new UsernamePasswordAuthenticationToken(
29                 credentials.getUsername(),
30                 credentials.getPassword());
31
32         Authentication auth = authenticationManager.authenticate(creds);
33
34         // Generate token
35         String jwt = jwtService.getToken(auth.getName());
36
37         // Build response with the generated token
38         return ResponseEntity.ok()
39             .header(HttpHeaders.AUTHORIZATION, "Bearer " + jwt)
40             .header(HttpHeaders.ACCESS_CONTROL_EXPOSE_HEADERS, "Authorization")
41             .build();
42     }
43 }
44

```

- Nous avons également injecté **AuthenticationManager** dans la classe **LoginController**, donc nous devons ajouter le code suivant à la classe **SecurityConfig**.

```

43 @Bean
44 public AuthenticationManager authenticationManager(AuthenticationConfiguration authenticationConfiguration)
45     throws Exception {
46     return authenticationConfiguration.getAuthenticationManager();
47 }
48
49

```

- Nous devons configurer la fonctionnalité de Spring Security. La méthode **configureSecurity** de Spring Security définit quels chemins sont sécurisés et lesquels ne le sont pas. Ajoutez la méthode **configureSecurity** suivante à la classe **SecurityConfig**. Là, nous définissons que la requête **POST** vers l'endpoint **/login** est autorisée sans authentification, tandis que les requêtes vers tous les autres endpoints nécessitent une authentification. Nous définirons

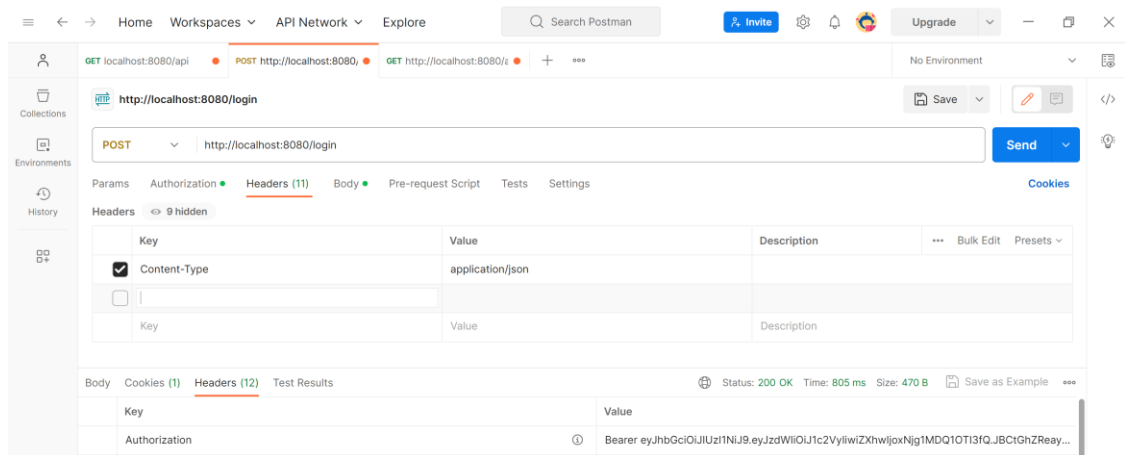
également que Spring Security ne créera jamais de session, nous pouvons donc également désactiver **CSRF (Cross-Site Request Forgery)** ¹.

```

25
26 @Bean
27 SecurityFilterChain configureSecurity(HttpSecurity http) throws Exception {
28     http.csrf().disable()
29     .sessionManagement()
30     .sessionCreationPolicy(SessionCreationPolicy.STATELESS).and()
31     .authorizeHttpRequests().requestMatchers(HttpMethod.POST, "/login").permitAll()
32     .anyRequest().authenticated();
33     return http.build();
34 }
35

```

7. Enfin, nous sommes prêts à tester notre fonctionnalité de connexion. Ouvrez Postman et effectuez une requête **POST** vers l'URL **http://localhost:8080/login**. Définissez un utilisateur valide dans le corps de la requête, par exemple `{"username":"user", "password":"user"}`, et définissez l'en-tête **Content-Type** sur **application/json**. Maintenant, vous devriez voir un en-tête **Authorization** dans la réponse qui contient le JWT signé, comme celui montré dans la capture d'écran suivante :



Vous pouvez également tester la connexion en utilisant le mauvais mot de passe et constater que la réponse ne contient pas l'en-tête d'autorisation.

Partie 2

Nous avons maintenant finalisé l'étape de connexion et nous passerons à l'authentification dans le reste des requêtes entrantes.

Dans le processus d'authentification, nous utilisons des filtres qui nous permettent d'effectuer certaines opérations avant qu'une requête n'arrive au contrôleur ou avant qu'une réponse ne soit envoyée à un client. Les étapes suivantes illustrent le reste du processus d'authentification :

1. Nous utiliserons une classe de filtre pour authentifier toutes les autres requêtes entrantes. Créez une nouvelle classe appelée **AuthenticationFilter** dans le package principal. La classe **AuthenticationFilter** étend l'interface **OncePerRequestFilter** de Spring Security, qui fournit une méthode **doFilterInternal** où nous implémentons notre authentification. Nous devons injecter une instance **JwtService** dans la classe de filtre, car cela est nécessaire pour vérifier un jeton provenant de la requête. Le code est illustré dans l'extrait suivant :

¹ <https://developer.mozilla.org/fr/docs/Glossary/CSRF>

```

19
20 @Component
21 public class AuthenticationFilter extends OncePerRequestFilter {
22     @Autowired
23     private JwtService jwtService;
24
25     @Override
26     protected void doFilterInternal(HttpServletRequest request,
27                                     HttpServletResponse response,
28                                     FilterChain filterChain)
29         throws ServletException, IOException {
30
31         // Get token from Authorization header
32         String jwt = request.getHeader(HttpHeaders.AUTHORIZATION);
33
34         if (jwt != null) {
35             // Verify token and get user
36             String user = jwtService.getAuthUser(request);
37
38             // Authenticate
39             Authentication authentication =
40                 new UsernamePasswordAuthenticationToken(user, null, java.util.Collections.emptyList());
41             SecurityContextHolder.getContext().setAuthentication(authentication);
42
43             filterChain.doFilter(request, response);
44         }
45     }
46 }

```

2. Ensuite, nous devons ajouter notre classe de filtre à la configuration de Spring Security. Ouvrez la classe **SecurityConfig** et injectez la classe **AuthenticationFilter** que nous venons de mettre en œuvre, comme suit :

```

21
22 @Autowired
23 private UserDetailsServiceImpl userDetailsService;
24
25 @Autowired
26 private AuthenticationFilter authenticationFilter;
27

```

3. Ensuite modifier le code de la méthode configure de la classe SecurityConfig.java en ajoutant les lignes 40, 41 et 42 :

```

28
29 @Autowired
30 private AuthenticationFilter authenticationFilter;
31
32
33 @Bean
34 SecurityFilterChain configureSecurity(HttpSecurity http) throws Exception {
35     http.csrf().disable()
36         .sessionManagement()
37         .sessionCreationPolicy(SessionCreationPolicy.STATELESS).and()
38         .authorizeHttpRequests().requestMatchers(HttpMethod.POST, "/login").permitAll()
39         .anyRequest().authenticated()
40         .and()
41         .addFilterBefore(authenticationFilter, UsernamePasswordAuthenticationFilter.class)
42         .httpBasic(withDefaults());
43     return http.build();
44 }
45

```

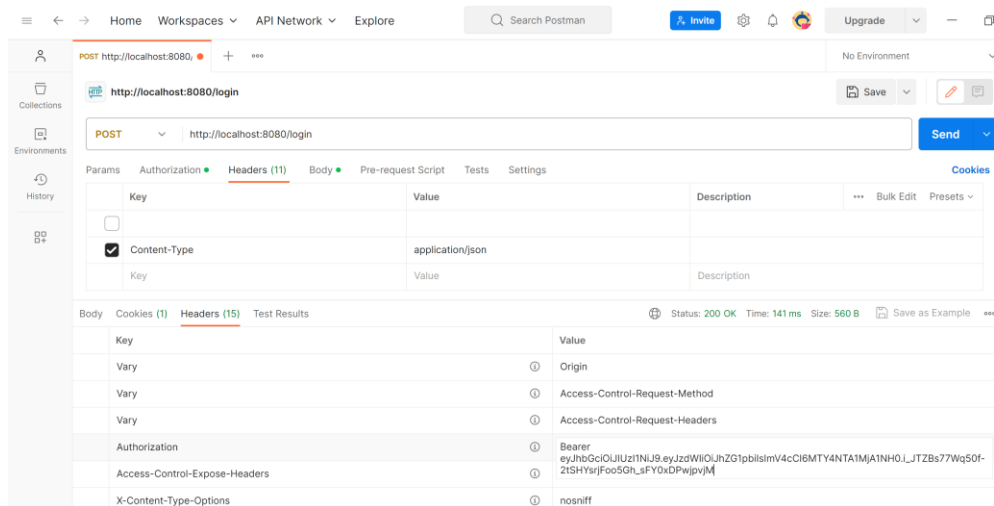
La classe **UsernamePasswordAuthenticationFilter** est importée ci-dessous :

```

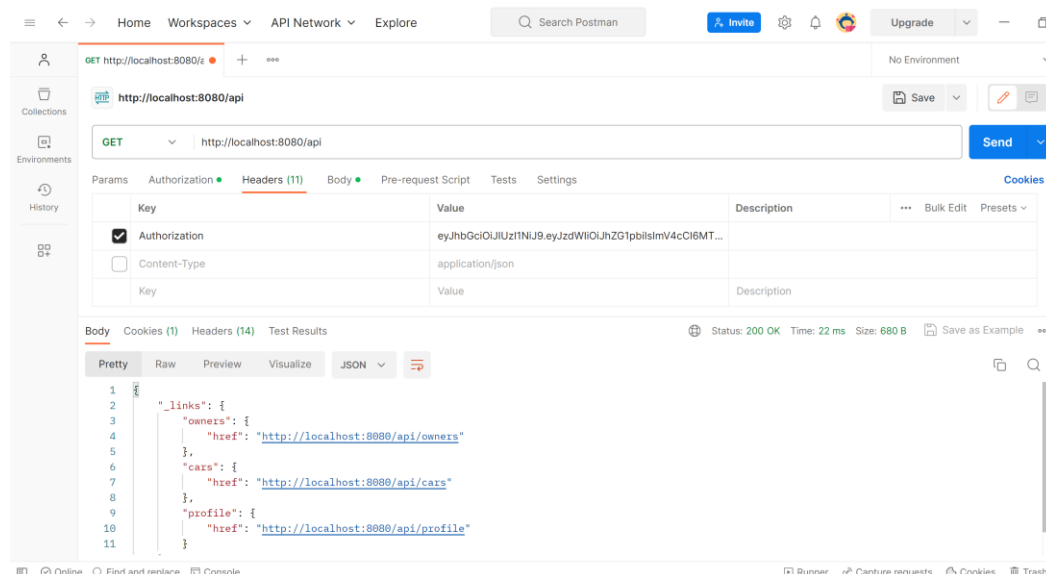
import
org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

```

Maintenant, nous sommes prêts à tester l'ensemble du flux de travail. Après avoir exécuté l'application, nous pouvons d'abord nous connecter en appelant l'endpoint **/login** avec la méthode **POST**, et en cas de connexion réussie, nous recevrons un JWT dans l'en-tête **Authorization**. N'oubliez pas d'ajouter un utilisateur valide dans le corps de la requête et de définir l'en-tête **Content-Type** sur **application/json**. Les captures d'écran suivantes illustrent le processus.



4. Suite à une connexion réussie, nous pouvons appeler les autres points de terminaison du service RESTful en envoyant le **JWT reçu** lors de la connexion dans l'en-tête **Authorization**. Copiez le jeton à partir de la réponse de connexion (sans le préfixe Bearer) et ajoutez l'en-tête **Authorization** avec le jeton dans la colonne VALUE. Référez-vous à l'exemple dans la capture d'écran suivante :



5. Nous devrions également gérer les exceptions lors de l'authentification. Maintenant, si vous essayez de vous connecter en utilisant le mauvais mot de passe, vous obtenez un statut 403 Forbidden sans plus d'explications. Spring Security fournit une interface **AuthenticationEntryPoint** qui peut être utilisée pour gérer les exceptions. Créez une nouvelle classe nommée **AuthEntryPoint** dans le package principal qui implémente **AuthenticationEntryPoint**.

Nous implémentons la méthode **commence** qui reçoit une exception en tant que paramètre. En cas d'exception, nous définissons le statut de la réponse sur **401 Unauthorized** et écrivons un message d'exception dans le corps de la réponse. Le code est illustré dans l'extrait suivant :

```

1 package com.dam.uas2.sbcar5;
2
3 import java.io.IOException;
4 import java.io.PrintWriter;
5
6 import jakarta.servlet.ServletException;
7 import jakarta.servlet.http.HttpServletRequest;
8 import jakarta.servlet.http.HttpServletResponse;
9
10 import org.springframework.http.MediaType;
11 import org.springframework.security.core.AuthenticationException;
12 import org.springframework.security.web.AuthenticationEntryPoint;
13 import org.springframework.stereotype.Component;
14
15 @Component
16 public class AuthEntryPoint implements AuthenticationEntryPoint {
17     @Override
18     public void commence(HttpServletRequest request, HttpServletResponse response,
19         AuthenticationException authException) throws IOException, ServletException {
20         response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
21         response.setContentType(MediaType.APPLICATION_JSON_VALUE);
22         PrintWriter writer = response.getWriter();
23         writer.println("Error: " + authException.getMessage());
24     }
25 }
26
27

```

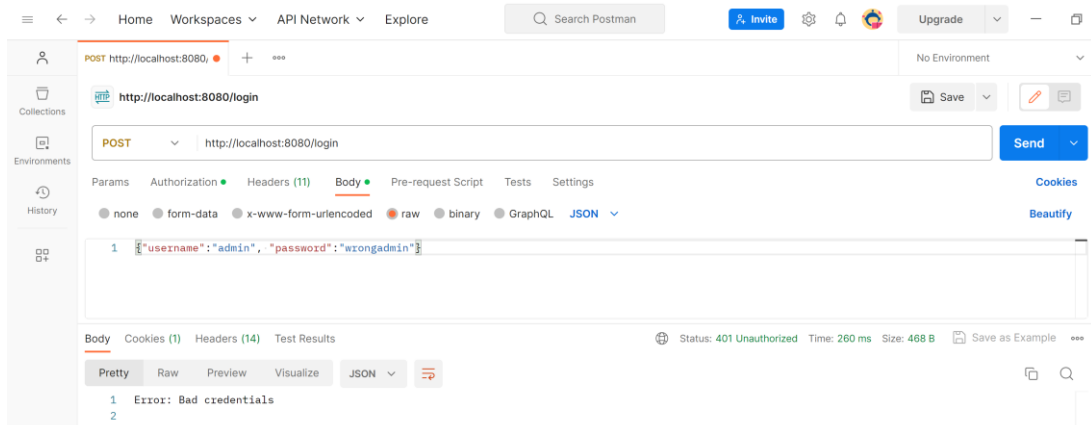
6. Ensuite, nous devons configurer Spring Security pour la gestion des exceptions. Injectez notre classe **AuthEntryPoint** (ligne 32 - 33), dans la classe **SecurityConfig**, et modifier la méthode configure en ajoutant les lignes 44 - 46:

```

31
32 @Autowired
33 private AuthEntryPoint exceptionHandler;
34
35
36 @Bean
37 SecurityFilterChain configureSecurity(HttpSecurity http) throws Exception {
38     http.csrf().disable()
39         .sessionManagement()
40         .sessionCreationPolicy(SessionCreationPolicy.STATELESS).and()
41         .authorizeHttpRequests().requestMatchers(HttpMethod.POST, "/login").permitAll()
42         .anyRequest().authenticated()
43         .and()
44         .exceptionHandling()
45         .authenticationEntryPoint(exceptionHandler)
46         .and()
47         .addFilterBefore(authenticationFilter, UsernamePasswordAuthenticationFilter.class)
48         .httpBasic(withDefaults());
49     return http.build();
50 }
51

```

7. Maintenant, si vous envoyez une requête **POST** de connexion avec des identifiants incorrects, vous obtiendrez un statut **401 Unauthorized** dans la réponse et un message d'erreur dans le corps, comme indiqué dans la capture d'écran suivante :



Nous ajouterons également un filtre de partage des ressources en **cross-origin (CORS)**² à notre classe de configuration de sécurité. Cela est nécessaire pour le frontend, qui envoie des requêtes à partir d'une autre origine. Le filtre CORS intercepte les requêtes et, s'il les identifie comme étant cross-origin, il ajoute les en-têtes appropriés à la requête. Pour cela, nous utiliserons l'interface **CorsConfigurationSource** de Spring Security. Dans cet exemple, nous autoriserons toutes les méthodes et en-têtes HTTP des origines. Vous pouvez définir une liste d'origines, de méthodes et d'en-têtes autorisés ici si vous avez besoin d'une définition plus précise.

² <https://developer.mozilla.org/fr/docs/Web/HTTP/CORS>

8. Ajouter la méthode ci-dessous à la classe **SecurityConfig** (Ligne 57 – 69) pour permettre le filtre CORS :

```
56
57 @Bean
58 CorsConfigurationSource corsConfigurationSource() {
59     UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
60     CorsConfiguration config = new CorsConfiguration();
61     config.setAllowedOrigins(Arrays.asList("*"));
62     config.setAllowedMethods(Arrays.asList("*"));
63     config.setAllowedHeaders(Arrays.asList("*"));
64     config.setAllowCredentials(false);
65     config.applyPermitDefaultValues();
66
67     source.registerCorsConfiguration("/*", config);
68     return source;
69 }
70
```

9. Si vous souhaitez définir explicitement les origines, vous pouvez le faire comme suit :

```
// localhost:3000 is allowed
config.setAllowedOrigins(Arrays.asList("http://localhost:3000"));
```

Nous devons également ajouter la fonction **cors()** à la méthode configure, en ajoutant la ligne

44 :

```
40
41 @Bean
42 SecurityFilterChain configureSecurity(HttpSecurity http) throws Exception {
43     http.csrf().disable()
44         .cors().and()
45         .sessionManagement()
46         .sessionCreationPolicy(SessionCreationPolicy.STATELESS).and()
47         .authorizeHttpRequests().requestMatchers(HttpMethod.POST, "/login").permitAll()
48         .anyRequest().authenticated()
49         .and()
50         .exceptionHandling()
51         .authenticationEntryPoint(exceptionHandler)
52         .and()
53         .addFilterBefore(authenticationFilter, UsernamePasswordAuthenticationFilter.class)
54         .httpBasic(withDefaults());
55     return http.build();
56 }
```

Maintenant, toutes les fonctionnalités requises ont été implémentées dans notre backend. Vous pouvez refaire les tests précédents pour vous assurer que le backend est fonctionnel.

Résumé

Dans ce lab, nous nous sommes concentrés sur la sécurisation. La sécurisation a été effectuée avec Spring Security. Le frontend sera développé avec React dans les prochains labs ; par conséquent, nous avons mis en œuvre l'authentification JWT, qui est une méthode d'authentification légère adaptée à nos besoins.