Spring Boot Security - Authentication with JPA

Create a Spring boot application\(^1\) with the following dependencies:

- Spring Web;
- Spring Data JPA SQL;
- PostgreSQL Driver SQL;
- Spring Security

After selecting the aforementioned dependencies, you need to press enter and you’ll be asked where you want to save your project. After choosing the location you’ll see a message asking you if you want to open the project.

**Note:** if you are starting a new application, Spring Data JPA SQL requires you to define your data source by adding the following snippet to the application.properties, which can be found at:

```
src/main/resources
    application.properties
```

You need also to specify “your_database_name”, “username”, and “password” for your database

```
# Database Settings
spring.datasource.url=jdbc:postgresql://localhost:5432/your_database_name
spring.datasource.username= username
spring.datasource.password= password

# The SQL dialect makes Hibernate generate better SQL for the chosen database
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect

# Hibernate ddl auto (create, create-drop, validate, update)
# Allows for auto creation of tables
spring.jpa.hibernate.ddl-auto = update

# Is used to escape all database identifiers, meaning that we don’t need to manually escape the table or column names:
spring.jpa.properties.hibernate.globally_quoted_identifiers=true
```

Create a simple controller (e.g., SecController.java) in your main package

```
main/java/..../..
    SecController.java
```

\(^1\) If you need to refresh your memory on how you can create a Spring Boot application, please refer to the material of Lecture 7
A metamodel of the classes and interfaces involved in the Authentication and Authorization through JPA

```
//nameController.java

@RestController
public class secController {

    @GetMapping("/")
    public String publicAPI(){
        return "Hi, this is a public API";
    }

    @GetMapping("/auth")
    public String authenticatedAPI(){
        return "Hi, you are authenticated";
    }

    @GetMapping("/user")
    public String userAPI(){
        return "Hi, you are a user/admin";
    }

    @GetMapping("/admin")
    public String adminAPI(){
        return "Hi, you are an admin";
    }
}
```
Create user table

We need to create a table in the Postgres database to store the users and their credentials, which can be done by creating a user class, as follows:

```java
//User.java

import javax.persistence.*;

@Entity
@Table(name = "user")
public class User {

@Id
@GeneratedValue(strategy = GenerationType.AUTO)
@Column(name = "user_id")
private int id;

@Column(name = "name")
private String name;

@Column(name = "password")
private String password;

@Column(name = "roles")
private String roles;

public User() {
}

public User(User user) {
    this.id = user.getId();
    this.name = user.getName();
    this.password = user.getPassword();
    this.roles = user.getRoles();
}

public int getId() {
    return this.id;
}

public void setId(int id) {
    this.id = id;
}

public String getName() {
    return this.name;
}

public void setName(String name) {
```
Remember that we set “spring.jpa.hibernate.ddl-auto = update” in the application.properties, which allows for auto-creation of tables.

Note: the table will be created automatically, but you need to insert your users, passwords, and roles manually. An example of the users used in this scenario is shown in the figure below.

<table>
<thead>
<tr>
<th>user_id</th>
<th>name</th>
<th>password</th>
<th>roles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>admin</td>
<td>admin</td>
<td>ADMIN</td>
</tr>
<tr>
<td>2</td>
<td>user</td>
<td>user</td>
<td>USER</td>
</tr>
</tbody>
</table>

### Implement UserDetails

Next, we need to create a class that implements the UserDetails interface as required by Spring Security. So create the MyUserDetails class with the following code:

```java
// MyUserDetails.java

public class MyUserDetails extends User implements UserDetails {

    private User user;

    public MyUserDetails(final User user) {
        this.user = user;
    }

    // Getters and setters
}
```
Implement UserDetailsService

For Spring Security authentication using JPA and Hibernate, we need to implement the UserDetailsService interface by the following class:

```java
    this.user = user;

    @Override
    public List<? extends GrantedAuthority> getAuthorities() {
        SimpleGrantedAuthority authority = new SimpleGrantedAuthority(user.getRoles());
        return Arrays.asList(authority);
    }

    @Override
    public String getPassword() {
        return user.getPassword();
    }

    @Override
    public String getUsername() {
        return user.getName();
    }

    //hard-coding these attributes
    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }
```
This class uses an implementation of UserRepository, which will be created and injected by Spring Data JPA. And we override the loadUserByUsername() method to authenticate the users.

**Codify the UserRepository class**

Create the UserRepository interface with the following code

```java
// UserRepository.java

public interface UserRepository extends JpaRepository<User, Integer> {
    Optional<User> findByName(String username);
}
```

**Configure authentication provider and HTTP security (authorization)**

We connect all the pieces by creating a Java file for security configuration (e.g., SecurityConfiguration.java) that extends the WebSecurityConfigurerAdapter in your main package

```
main/java/.../...
    SecurityConfiguration.java
```
Add the following code, which overrides the AuthenticationManagerBuilder and registers two different users into the authentication service. Additionally, this class must be annotated with the @EnableWebSecurity and @Configuration annotations

```java
// SecurityConfiguration.java

@EnableWebSecurity
@Configuration
public class SecurityConfiguration extends WebSecurityConfigurerAdapter{

@Autowired
private UserDetailsService userDetailsService;

@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception{
    auth.userDetailsService(userDetailsService)
        .passwordEncoder(getPasswordEncoder());
}

@Bean(name = "passwordEncoder")
public PasswordEncoder getPasswordEncoder() {
    return NoOpPasswordEncoder.getInstance();
}

@Override
protected void configure(HttpSecurity http) throws Exception{
    http.cors()
        .and()
        .csrf().disable()
        .authorizeRequests()
        .antMatchers("/admin").hasAuthority("ADMIN")
        .antMatchers("/user").hasAnyAuthority("ADMIN", "USER")
        .antMatchers("/auth").authenticated()
        .antMatchers("/").permitAll()
        .and().formLogin().permitAll();
}
```