Java Advanced – lesson 7

Wim Wiltenburg

(willem.wiltenburg@inholland.nl)

User management with database

So far we have discussed two ways of securing our application with Spring Security:

- API key – Stateless, every request needs to be authenticated
- Users with InMemory authentication

The reality is that most user management occurs with other mechanisms, like LDAP, ActiveDirectory, and many times with users that are persisted in a database.

To accomplish this we need the UserDetailsService

User

With InMemory security we needed a user with a password, and a role, something like USER or ADMIN. Now we need to create this in the database.

Therefore we create an entity class that holds strings for username and password.

We also need to store the role or roles that the user has. We can make this a separate entity with a @OneToOne or @OneToMany relationship. But we can also create it in the user objects itself by defining an Enum or a List of Enum.

The question of creating a separate object depends on whether the number of roles are fixed or expected to change regularly.

User repository

Users are stored in the database and can be retrieved, saved and updated through the database. For this we need the repository, and that repository needs to have at least one method:

```
User findByUsername(String username);
```

This query is defined by Hibernate; it only needs to be declared.

Important: if the username is not found in the database it can cause a NullPointerException later in the process… When called from a service, please check this, and if no user is found the proper exception to throw is UsernameNotFoundException.

Password encryption

One thing we don't want is to have the password in our database in plain text. That's why we need encyption before storing it.

We use Bcrypt for this. It comes with Spring Security, and we encrypt our password before saving it to the database, e.g. in the constructor.

```
this.password = new BCryptPasswordEncoder().encode(password);
```

We never decrypt the password. When logging in we encrypt the password the user entered again, and compare it to what's stored in the database.

UserDetails

In order for the authenticate manager to communicate with the user object, it is wrapped within an object that has methods getting the username, password, and which roles and privileges the user has. In the example code this object is called SecurityUserDetails and extends the UserDetails class

The method getAuthorities() gives a Collection of roles, and privileges (e.g. READ_ACCESS) which we can use in the Security Configuration to authorize endpoints.

So, we create a class that extends UserDetails, and in the constructor we define the user that it concerns.

There is a Spring Security class User that extends UserDetails, so one could say: why bother, but the question is: would you want that user to be an entity… I would choose no.

The UserDetailService

The UserDetailService is the service that retrieves the userdetails. The most important method of the service is loadUserByUsername. It's used by the authentication manager to retrieve the user details. We discussed the need to check if the username exists or else throw an exception:

```
@Override
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
  User user = Optional.ofNullable(userRepository.findByUsername(username))
     .orElseThrow(() -> new UsernameNotFoundException("user not found"));

  return new SecurityUserDetails(user);
}
```

UserDetailsManager

If you want more functionality from the service like creating a user, changing passwords, etc. you can create a service object by extending not UserDetailsService, but by extending UserDetailsManager.

This is still a service, but it allows for the additional functionality desciberd above.

The security configuration

The easiest part of this process is creating the authentication manager to handle users from the database. The following example is enough:

```
private final UserDetailsServiceImpl userDetailsService;

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

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

The authentication manager uses the service to retrieve UserDetails, and checks basic user characteristics: username, password, locked or not

Integration with a login page

The security configuration also allows for a user defined login page, and redirect if authentication is successful. Example:

```
protected void configure(HttpSecurity http) throws Exception {
  http.csrf().disable();
  http.authorizeRequests()
     .antMatchers(HttpMethod.GET, "/guitars/**").permitAll()
     .antMatchers(HttpMethod.POST, "/guitars/**").hasAuthority("ADMIN")
     .antMatchers("/**").permitAll()
     .anyRequest().authenticated()
     .and()
     .formLogin()
     .loginPage("/login.html")
     .loginProcessingUrl("/perform_login") // is the POST action in your login form
     .defaultSuccessUrl("/create-guitar.html", true) // is where the application redirects to after succesful login
     .permitAll()
     .and()
     .logout()
     .logoutUrl("/perform_logout")  // is the endpoint you can use for logging out
     .deleteCookies("JSESSIONID") // delete session
     .permitAll();
    }
```

Questions

Last Slide