Web Application 2 2020-2021

Exercise 3: Wallet Service Security: Authentication and Authorization

In order to use the services offered by the Wallet Service, the user must to be registered on the e-commerce platform. In this way, only authenticated users can associate a wallet, perform transactions, or consult the information related to them. Moreover, there are users who may have less restrictive authorizations than others, being able to perform – for example – administrative operations. Only the owner of a wallet can spend money contained inside. Both the owners and the administrators can check the balance and retrieve the transaction list of a given wallet.

We will proceed according the following plan:

1. We first create a separate branch of the previous code, in order to extend it.
2. In the new branch, we add support for modelling users: this means introducing a table to store their details, the corresponding entity, a repository and a service for managing them
3. We create some endpoints for registering new users, validating their email addresses, support updating their password, checking their credentials (login)
4. We add the Spring Security module, configuring it in order to properly allow contacting public endpoints, restrict the access to the secured ones, insert a validation of incoming credentials provided as a JWT.

Step 1

If your project is not yet under VCS, enable VCS and create an initial commit. Create a project branch named "lab3" and switch to it.

Step 2

1. Create an enum class Rolename, having values CUSTOMER and ADMIN.
2. Create the User entity with username (unique, indexed, nonempty, and validated), password, email (unique and validated), isEnabled (set to false by default, which indicates whether the User is enabled to perform operations in the Wallet Service), and roles. The latter will be a string obtained as the concatenation of Rolename values. Add three methods for retrieving the set of Rolenames (by parsing the roles property), adding and removing a Rolename (and update the roles property, accordingly).
3. Make an index on username: add @Table(indexes = [Index(name = "index", columnList = "username", unique = true)]) in front of the User entity. The database index is a data structure that improves the speed of data retrieval operations on a table at the cost of additional writes and storage space.
4. The Customer entity will have a One-to-One relationship with the User object.
5. Create the corresponding Repository, adding a method findByUsername(username: String): User?
6. Write and run some tests checking that you can safely store, retrieve and update a User entity as well as manipulate its roles without creating duplicates or breaking anything.
7. Commit your project.
8. Create class UserDetailsDTO, implementing the UserDetails interface: this class will expose information contained in the User entity to other parts of the system, as well as implement the required methods. Also add a mapping function to convert a User into a UserDetailsDTO
9. Create service class UserDetailsServiceImpl implementing the UserDetailsService interface. Add methods for creating a new user given its properties, add and remove a role from a user with a given

username, enable or disable a user given its username. Method *loadUserByUsername(username: String) : UserDetailsDTO*, will throw an *UsernameNotFound* exception if the user cannot be found.

10. Write and run some more tests checking that everything is ok.
11. Commit your project.

Step 3

Now we need to implement the APIs so that the user can register on the e-commerce service and log in, to take advantage of the features offered. When the user performs the registration request, an email must be sent to the provided address with a confirmation link. The end-points to be created are:

- /auth/register [POST]
- /auth/signin [POST]
- /auth/registrationConfirm?token=<token> [GET]

12. Registration: The request should contain username, valid email, name, surname, address, password and confirmPassword (they should match). Use the UserDetailsServiceImpl service to create a new user, assigning the CUSTOMER rolename, and setting the enable flag to false.
13. Write and run some more tests checking that you can properly create a new user, getting the proper role and flag.
14. Commit your project.
15. Registration confirmation: during the registration process, we have to send a confirmation email to the registered user. We'll use the *JavaMail* sender interface for this purpose. Add spring-boot-starter-mail to your build.gradle.kts dependencies
16. Create a DUMMY gmail account NOT LINKED TO YOUR main account one (one per group is enough), in order to be able to share it with your colleagues without leaking personal information. Add to the application.properties file the necessary settings:

```
spring.mail.host=smtp.gmail.com
spring.mail.port=587
spring.mail.username=YOUR_MAIL
spring.mail.password=YOUR_PASSWORD
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
spring.mail.properties.mail.debug=true
```

17. Implement, in your main class, a *@Bean getMailSender()* method that returns the *JavaMailSender* object, setting to with the configured parameters. *JavaMailSender* is an interface that extends *MailSender* interface for JavaMail. A default implementation of it is provided by Spring, but it must be configured with the relevant settings provided in the application.properties file (use @Value properties to grab them)
18. Implement the *MailService* that will wire the JavaMailSender bean. Implement the *sendMessage (toMail, subject, mailBody)* method. Inside this method, build a *SimpleMailMessage* object setting the necessary parameters. Use the *send* method of JavaMailSender to send it.
19. Write and run some tests to check that everything is ok
20. Commit your project
21. Implement the Entity *EmailVerificationToken*, with the expiryDate timestamp, the string token (a random UUID converted to string), and the username to which it refers to. Implement the *EmailVerificationTokenRepository,* exposing a method that let you locate an entity given the token.
22. Implement the *NotificationService*, with a method that creates an EmailVerificationToken and persists it.
23. Inject in the UserDetailsServiceImpl the references to the NotificationService and MailService
24. Update the registerUser() method in this class, to support the registration flow: when the user is registered, an email containing a link to the confirmation endpoint should be sent

25. Create the endpoint */auth/registrationConfirm?token=<token>:* when this is invoked, retrieve from repository the *EmailVerificationToken* by the token String. Verify if it is not expired and enable the corresponding username. If the token is not found or it is expired, throw an exception.
26. Implement an Automatic Task that cleans the expired *EmailVerificationTokens* from db.
27. Write and run some tests that check that everything is ok. Commit your project.

Step 4

Now, it is necessary to add the security layer to the application. For these purposes we'll use *Spring Security* module. The mechanism chosen for authorizing access to web APIs is JSON Web Token.

28. Add the following dependencies and settings:

```
implementation("io.jsonwebtoken:jjwt-api:0.11.2")
runtimeOnly("io.jsonwebtoken:jjwt-impl:0.11.2")
runtimeOnly("io.jsonwebtoken:jjwt-jackson:0.11.2")
implementation("org.springframework.boot:spring-boot-starter-security")
```

```
application.jwt.jwtSecret = <YOUR_LONG_SECRET>
application.jwt.jwtExpirationMs = <EXPIRATION_IN_MILLISECONDS>
application.jwt.jwtHeader = Authorization
application.jwt.jwtHeaderStart = Bearer
```

29. In the main file, add a @Bean passwordEncoder(): PasswordEncoder method returning a delegating password encoder. This will be used throughout the application to encode passwords.
30. Create the *security* package in your application.
31. To sign and validate a JWT, it is necessary to configure the its secret key and expiration time, in ms. These will be loaded from the *application.properties* file via @Value(…) properties. We can set also the name of the http header in which the token must be stored when a request is performed (the *Authorization* header) and the *Bearer* prefix String. In this way, when we'll retrieve the token from the *Authorization* header, to validate it, we'll take the part after the *Bearer* prefix.
32. In the security package, create the *JwtUtils* bean class with three methods:
    - *fun generateJwtToken (authentication: Authentication): String*: use Jwts builder to generate a JWT from an Authentication object and to sign it using the *jwtSecret*. The authentication principal will be an instance of UserDetailsDTO. Apart the mandatory information, the claims section will only contain the user roles
    - *fun validateJwtToken (authToken: String): Boolean*: validate the token received in a request, catching all exceptions and returning true only if the token is valid
    - *fun getDetailsFromJwtToken (authToken: String): UserDetailsDTO:* use the Jwts parser to retrieve the username and roles from the token. This method will be used by the authentication filter to extract the relevant data after the JWT has been validated. The extracted information will not contain any details apart the username and the roles.
33. Write and run some tests, checking that a JWT is properly created when submitting a UserDetailsDTO. Use the online service at https://jwt.io/ to verify that the generated JWT actually encodes the expected values or to create a JWT to validate and extract data from.
34. Commit your project.

35. In the security package, create the WebSecurityConfig class extending WebSecurityConfigurerAdapter and label it with @Configuration. Inject the password encoder, the UserDetailsService.

36. Override method configure(http: HttpSecurity) to provide rules for managing CORS, CSRF, sessions, and define rules for protected resources.
37. Implement the HttpSecurity configuration and protect all resources that not match the "/auth/**" path. The *auth* path will be used for the entry points that not need authentication.
38. Later we will add a JwtAuthenticationTokenFilter to the filter chain in order to handle JWT authentication and set the authenticated user (aka the Principal) in the SecurityContext. For this reason, disable the default Spring session management behaviour, which relies on HTTP sessions to store access information.
39. Implement an *AuthenticationEntryPoint* bean and override its *commence()* method: this will be invoked anytime an AuthenticationException is thrown. In these situations, we want to customize the default behaviour and send an UNAUTHORIZED response error. Set your bean as the authentication entry point Exception handler adding the following line to the configure(http: HttpSecurity) method of WebSecurityConfig.

```
http.exceptionHandling().authenticationEntryPoint(authenticationEntryPoint)
```

The Authentication object encapsulates the Principal being authenticated, or the authenticated principal after authentication. This will consist of an instance of the UserDetailsDTO class

40. In the WebSecurityConfig class, also override the configure(auth: AuthenticationManagerBuilder) method. Set the injected UserDetailService and the passwordEncoder via the following code.

```
override fun configure (auth: AuthenticationManagerBuilder) {
   auth
        .userDetailsService(userDetailsService)
        .passwordEncoder(passwordEncoder())

}
```

41. Now we add to filter chain an Authentication Filter to validate a request. Create the *JwtAuthenticationTokenFilter* bean class that extends the *OncePerRequestFilter* abstract class that aims to guarantee a single execution per request dispatch, on any servlet container. It provides a *doFilterInternal(…)* method that must be overridden. Inside the *doFilterInternal* method:
    - retrieve the JWT from the Authorization header (removing the Bearer prefix)
    - if the request has JWT, validate it and extract the user details contained there via the methods offered by the JwtUtils injected instance
    - Create a UsernamePasswordAuthenticationToken object, setting the username and granted authorities fetched from the JWT, while leaving the password to null.
    - Add extra details coming from the request
    - Set the Authentication object in the SecurityContext

```
val authentication = UsernamePasswordAuthenticationToken(
    userDetails,
    null,
    userDetails.authorities
)
authentication.details = WebAuthenticationDetailsSource()
                            .buildDetails(request)
SecurityContextHolder.getContext().authentication = authentication
```

42. Set the implemented filter in HttpSecurity configure method of the WebSecurityConfig, adding it before *UsernamePasswordAuthenticationFilter*.

```
http.addFilterBefore(authenticationJwtTokenFilter(),
UsernamePasswordAuthenticationFilter::class.java)
```

43. Check that you can safely access the /auth/signin method obtaining a valid JWT
44. Try accessing a secured end-point both submitting and non submitting the JWT and check the correct behaviour.
45. Commit your project.

Requests must be not only authenticated, but authorized, too. In order to do that, we will use the AOP annotations provided by Spring Security.

46. Registered users can be enabled/disabled by administrator. Use Spring Security authorization semantics to secure the service layer, restricting only to the ADMIN role the right to invoke the enableUser(…) method of class UserDetailsServiceImpl.
47. It is necessary to enable global Method Security using the annotation @EnableGlobalMethodSecurity(prePostEnabled = true) in your program entry point.
48. Moreover, only the owners of a wallet should be allowed to perform transactions. So, add the necessary checks.
49. Write and run some tests to verify that everything is ok.
50. Commit your project.