Web Applications 2

Lab 4: authentication and authorization

Learning Objectives

By successfully completing this lab, students will develop the following skills:

- Configuring a Spring project with Spring Security and JWT management
- Implementing Authentication and Authorization filters supporting Role-Based Access Control (RBAC)
- Implement a RESTful endpoint for storing user-related information
- Managing a couple of cooperating microservices, keeping a loose coupling among them

Description

The public transport agency provides a WebPortal that allows citizens to consult global transport information, like transit timetables and network connections, as well as to manage tickets and subscriptions purchases and other administrative tasks. The portal is implemented as a collection of interoperating web services, each specialized in a particular functional subdomain. One of these services is the TravelerService: this is in charge of managing the identity and personal data of the travelers registered to the WebPortal. The service is accessed via a REST API, which exposes various end-points, some accessible to everybody, some restricted to authenticated users and providing only information pertaining to the identity of the client, some accessible only to users having administrative privileges.
The TravelerService indirectly cooperates with the LoginService: the latter exposes a REST API consisting of some endpoints for registering a new user and validating their email address (those implemented in Lab 3) as well as a login endpoint, which accepts users credentials (username and password) and returns, when authentication is successful, a JWT temporarily representing the identity and role of the user. This token must be enclosed, in a HTTP header, in all the requests directed to the TravelerService in order to have them authorized and properly identify the subject issuing the request. In this way, only authenticated users can consult the information related to themselves and only those who have the suitable role will be able to issue the restricted set of administrative requests.
The two services can be deployed as independent processes on different ports of the same hosts.

A registered user will have to first contact the LoginService, posting their credentials to the login endpoint and obtaining a JWT valid for the next hour, in order to be able to later contact the TravelerService, specifying the JWT as a value of the Authorisation http header.
Upon reception of a new request, the TravelerService will validate the JWT (relying on a shared secret kept into the service properties, which should be the same used by the LoginService to emit the JWT) and, if successful, let the request be handled by the service business logic.
The following endpoints must be implemented by TravelerService:

- GET /my/profile → returns a JSON representation of the current user's profile (name, address, date_of_birth, telephone_number) as stored in the service DB.
- PUT /my/profile → accepts a JSON representation of the current user's profile and updates the corresponding record in the service DB.
- GET /my/tickets/ → returns a JSON list of all the tickets purchased by the current user. A ticket is represented as a JSON object consisting of the fields "sub" (the unique ticketID), "iat" (issuedAt, a timestamp), "exp" (expiry timestamp), "zid" (zoneID, the set of transport zones it gives access to), "jws" (the encoding of the previous information as a signed JWT) [Note that this JWT will be used for providing physical access to the train area and will be signed by a key that has nothing to do with the key used by the LoginService]
- POST /my/tickets → accepts a JSON payload like the following {cmd: "buy_tickets", quantity: 2, zones: "ABC"} and generates a corresponding number of tickets, issued now and valid for the next hour, for the indicated transport zones. The generated tickets are stored in the service DB and will be returned as a payload of the response. At the moment, a user may require an unlimited number of tickets.
- GET /admin/travelers → returns a JSON list of usernames for which there exists any information (either in terms of user profile or issued tickets). THIS ENDPOINT is only available for users having the Admin role.
- GET /admin/traveler/{userID}/profile → returns the profile corresponding to userID. THIS ENDPOINT is only available for users having the Admin role.
- GET /admin/traveler/{userID}/tickets → returns the tickets owned by userID. THIS ENDPOINT is only available for users having the Admin role.

The Login service should expose, beyond all the endpoints provider in the previous Lab, also the following one:

- POST /user/login → receives a JSON object containing username and password and returns, upon successful validation, a JWT containing fields "sub", "iat", "exp", "roles".

LoginService Steps

1. Open the lab3 repository and create a branch named "lab 4" and switch to it.

2. In order to know the authorization level of each user, we need a way to distinguish them. For this purpose create an enum class Role, having values CUSTOMER and ADMIN. Add Role to User entity. When a new user performs a registration request, a CUSTOMER role must be assigned to him.

3. User passwords must be encrypted. We can use BCryptPasswordEncoder Class, which handles encryption with salt, in order to have different hashes with the same passwords.
   (https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/bcrypt/BCryptPasswordEncoder.html)

4. Create the /user/login endpoint that receives a JSON object containing username and password. In case of success, returns a valid JWT for the next hour containing fields "sub", "iat", "exp", "roles".

TravelerService Steps

1. Create a repository and start a new Spring Boot project including Spring WebMVC, Spring Data JPA. Commit and push.

2. Services must be loosely coupled so that they can be developed, deployed and scaled independently. Each service must have its own database. To reduce the consumption of resources, instead of launching a new postgres container, create a new database in the existing one.

3. In order to store requested data, two Entity classes are needed: UserDetails and TicketPurchased. UserDetails will have a property of type List<TicketPurchased>.

4. To sign and validate a JWT, it is necessary to configure its secret key and expiration time. These will be loaded from the application.properties file via @Value(…) properties. Amongst the other properties to be configured in the above mentioned file, there is the name of the http header where the authorization token will be stored when a request is performed (the Authorization header) and the Bearer prefix string, that introduces the value of the header as a JWT. When the token is retrieved from the Authorization header, the Bearer prefix must be removed in order to validate it.
   Create the security package, and inside it create the JwtUtils bean class with two methods:
   a. validateJwt (authToken: String): Boolean: it validates the token received in a request, catching all exceptions and returning true only if the token is valid
   b. getDetailsJwt (authToken: String): UserDetailsDTO: it relies on 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 from the username and the roles. Validation will fail if the current date is not in the validity period of the token.

   In a Spring security application, the WebSecurityConfigurerAdapter class provides the default security configuration. We extend this class to customize and override the default security configuration: in the security package create the WebSecurityConfig class that extends WebSecurityConfigurerAdapter and label it with @Configuration. Inject the password encoder and the user service. Override method configure(http: HttpSecurity) to provide rules for protected resources. Later 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 a example you can see (https://github.com/osahner/kotlin-spring-boot-rest-jpa-jwt-starter)
   Create the “/my/tickets” endpoint accessible only by authenticated users with a CUSTOMER role. This endpoint retrieves a json object containing the historical list of the tickets purchased by the principal.

Also implement the POST method for the same URL, creating and persisting the requested number of tickets for the current principal.
Proceed with the other endpoints, taking care to respect the security constraints related to the requested role.

Submission rules

Download a copy of the zipped repository and upload it in the “Elaborati” section of “Portale della didattica”. Label your file “Lab4-Group<N>.zip” (one copy, only - not one per each team member).
Work is due by Friday May 6, 23.59 for odd numbered teams, and by Friday May 13, 23.59 for even numbered ones.