Project Assignment Solution Outline
Paynet REST Security workshop Oct 2021

1 Registration system MVC app

The project folder for this app is `JWTRegistrationSystem`

To run this app properly, ensure that you have the following in place:

- A MySQL service with a database `workshopdb` already created
- A valid MySQL account that provides full read/write access to this database. The credentials for this account should be provided in the `spring.datasource.username` and `spring.datasource.password` properties in `application.properties`
- Set up an OAuth Client on Google API console as described in the lab manual on Spring Security
- Set up an OAuth App on GitHub as described in the lab manual on Spring Security
- Substituted the correct values for `client-id` and `client-secret` properties to configure the app to use the OAuth Client and OAuth app that you have just set up.

When the app is started, the startup logic implemented in StartupRunner automatically inserts 3 user accounts with hardcoded data into the `regusers` table in `workshopdb` (if they don't already exist). The data for user accounts is provided by RegUser, which uses the email field to uniquely identify each user account record in the `regusers` table. This field is the functional equivalent of the username that we would conventionally use in the Spring Security framework: so we no longer need an additional field for this.

1.1 Spring Security Configuration

In MainSecurityConfig, the DaoAuthenticationProvider uses the custom CustomUserDetailService to retrieve the relevant user account utilized in the Spring Security login workflow. Notice that when we override the loadUserByUsername method in CustomUserDetailService, we construct the UserDetails object by using the email field of the corresponding RegUser object as value of the username field (as explained previously).

The configure method configures authorization for access to the various URL paths. Two paths (`maintain` and `changedate`) are only accessible to accounts with ADMIN role. Customization is provided for standard login, OAuth login and logout. The passwords are encoded using the standard BCryptPasswordEncoder.

The customized login page also provides an option to login via Google and GitHub. The email of the user registered with that particular OAuth2 / OIDC provider (which will be available at the end of the OAuth2 workflow) is used to create a new user account in `regusers` (if there isn't already an existing one).

There is a CustomErrorController provided to return customized error messages (in the form of Thymeleaf templates) from `src/main/resources/templates/error` as a user-friendly response to standard 403 and 404 status code errors.

1.2 Forms and validation

As there is a lot of data acquired from the user, we create specific classes to model the forms (form backing objects) so that we can easily retrieve the form parameters and store them in the fields of the these objects. We use the Bean Validation API 2.0 to validate the content of all the form fields and return appropriate error messages when any of the input is not valid according to the constraints that we specify. This is achieved via the @Valid annotation in the signature of the handler methods in the main RegistrationController class that implement the form processing logic for the various forms that a user can submit while interacting with the app. We use the @ModelAttribute annotation to bind the form parameters to the corresponding fields in the form backing object, and express this binding through appropriate attribute parameters (`th:field`) in the form template itself.

1.3 Interacting with database table

We use RegUserRepository that provides 4 Spring Data JPA derived query methods to:

- locate a record by email
- delete a record by email
- update the REST role of a user
- update the account details of a user

The repository is autowired into a RegistrationService which in turn is autowired into the main RegistrationController, which processes requests and forms data and makes appropriate calls to RegistrationService to perform the required table operations.

1.4 JWT generation

The project root folder already contains the keystores with the necessary secret and private/public key pair needed to generate a JWS. The expiration date for the JWT is obtained from 3 properties stored in `jwt.properties` that specify the number of days, hours and minutes left until JWT expiration. These properties are used to compute the actual expiry date of the JWT at the point of its generation, and the admin has the ability to modify them and store the updated values back into `jwt.properties`

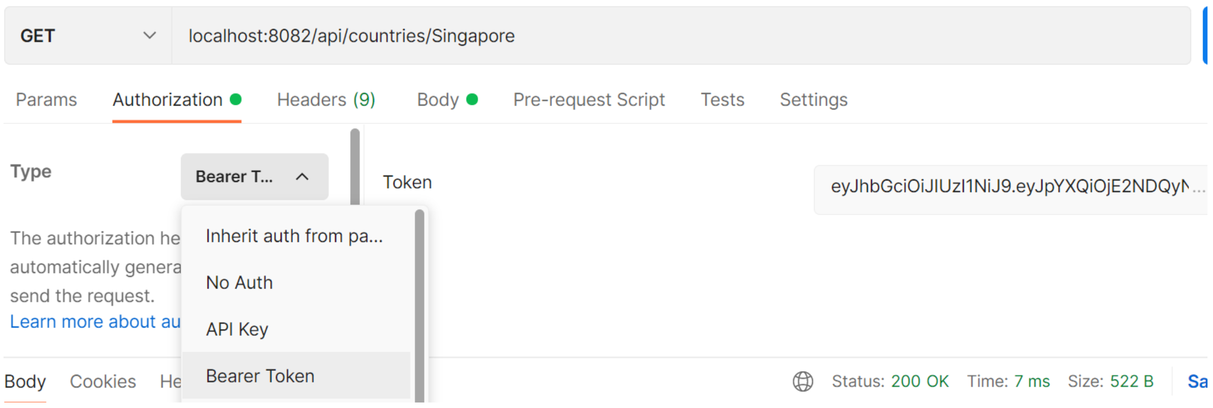
2 Secure REST service app

The project folder for this app is `JWTCountriesRESTService`

This is basically the same solution for the project on REST API, but now extended to support authentication / authorization via a JWT.

To run this app properly, ensure that you have the following in place:

- A MySQL service with a database `workshopdb` already created
- A valid MySQL account that provides full read/write access to this database. The credentials for this account should be provided in the `spring.datasource.username` and `spring.datasource.password` properties in `application.properties`
- If you are planning to run both Registration System app and the REST service app at the same time, make sure you start them up on different ports. You can configure the port that the embedded Tomcat server for a Spring Boot project starts up on via the `server.port` property in `application.properties`.
- Set the properties in `application.properties` to populate the MySQL table with initial data. Since the RapidAPI Countries REST API that this app is based on is no longer available publicly, you will need to initialize from a file. So for a first time run, set:
  ```
  initialize.mode=file
  json.file=allcountries.json
  ```
- Use Postman in the usual manner to send the various requests. Remember to include the JWT generated from the registration system app as the Bearer Token in all your outgoing requests.


GET localhost:8082/api/countries/Singapore

Type: Bearer T...

Token: eyJhbGciOiJIUzI1NiJ9.eyJpYXQiOjE2NDQyN...

The authorization he automatically genera send the request. Learn more about au

Status: 200 OK Time: 7 ms Size: 522 B

2.1 Spring Security Configuration

In MainSecurityConfig, the primary functionality is implemented in the configure method. Here we:

- Disable CSRF protection. CSRF protection is really only needed when the user is interacting with the app via a browser, which makes it vulnerable to requests send by malicious scripts that are loaded from other websites. In our app, we are only able to interact with the REST API either manually via the Postman client or programmatically using the RestTemplate client library. Therefore, CSRF protection is no longer necessary; and leaving it enabled by default may complicate our interactions when we are solely depending on the JWT to authenticate / authorize user requests.
- Disable session tracking via cookies. Since we are not interacting via a browser, we will no longer need to track the logged-in user via a session cookie. Instead, every single request is treated as a completely new one and authenticated / authorized based on the attached JWT in its Authorization: Token header.
- Set up authorization constraints on the `/api/countries` endpoint for various REST HTTP methods.

- Set up a custom handler to handle authorization exceptions and return appropriate error methods to the user

Notice that we do not need any configuration for password encoding or Authentication Provider since the actual authentication of an incoming request is achieved by validating the JWT in the Authorization: Bearer header. This is performed by the JWT Filter.

2.2 Spring Security Filter

JwtRequestFilter implements the filter functionality that intercepts the incoming request in order to extract the JWT from the Authorization: Token header and validate it accordingly. This filter will be placed at an appropriate location in the ordering of the filters within the SecurityFilterChain that forms the core of the Spring Security framework.

https://docs.spring.io/spring-security/reference/servlet/architecture.html#servlet-securityfilterchain
https://docs.spring.io/spring-security/reference/servlet/architecture.html#servlet-security-filters

By placing it before UsernamePasswordAuthenticationFilter in the ordering of the filters, we can implement our own custom authentication, which is to fundamentally extract the JWT from the request and validate it.

https://docs.spring.io/spring-security/reference/servlet/authentication/passwords/form.html#servlet-authentication-usernamepasswordauthenticationfilter

Depending on the result of this validation (success or specific type of validation failure), we will create an authenticated token with specific values for its role field.

For the case of a successful validation of the JWT, we will set the role of our authenticated token to the value extracted from the role claim in the JWT token. For validation failures, we will set the role of our authenticated token to specific custom values. These custom values (which are other than the valid values of BASIC or FULL) will result in an authorization exception, since the main security configuration will only authorize against these two roles. The custom AccessDenied handler will then be invoked.

2.3 Custom AccessDenied Handler

CustomAccessDeniedHandler will handle any authorization exceptions resulting from violations of the authorization constraints specified in MainSecurityConfig. Based on the custom role values provided in the authenticated token, it will decide on the appropriate error message to return to the user.

3 MVC Client for Secure REST service app

The project folder for this app is `JWTCountriesMVCClient`

This is basically the same solution for the project on REST API, but now extended to make REST APIs via RestTemplate library to the Secure REST service with a JWT token inserted in the Authorization header.

To run this app properly, ensure that you have the following in place:

- If you are running this app simultaneously with the Secure REST service app and the Registration System MVC app, you need to ensure that it is running on a separate port which you can set via `server.port` in application. Properties
- Ensure that the base URL for the Secure REST service app (`myrest.url`) is pointing to the correct port

3.1 Using RestTemplate library

The various calls to the RestTemplate API in order to make the outgoing HTTP requests to the Secure REST service app are achieved in MyRestService. We simply need to modify all calls to include a JWT token in the Authorization header.