Assignment: JWT-based authentication and authorization

Authentication is the process of verifying a user's credentials. It is also commonly named as "sign-in" or "login" in many systems. Authorization is about checking what the user can do in the system after authentication succeeds.

Token-based authentication and authorization is nowadays the standard for securing client-server (Figure 1) architectures where the Frontend is totally decoupled from the Backend. It is also commonly used in Microservices architectures (Figure 2) together with OAuth.

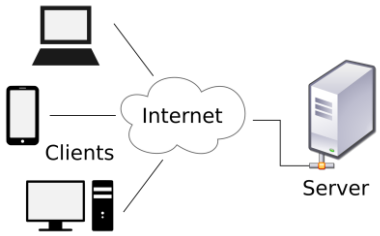

Figure 1: Client-server model illustration (From Wikipedia)

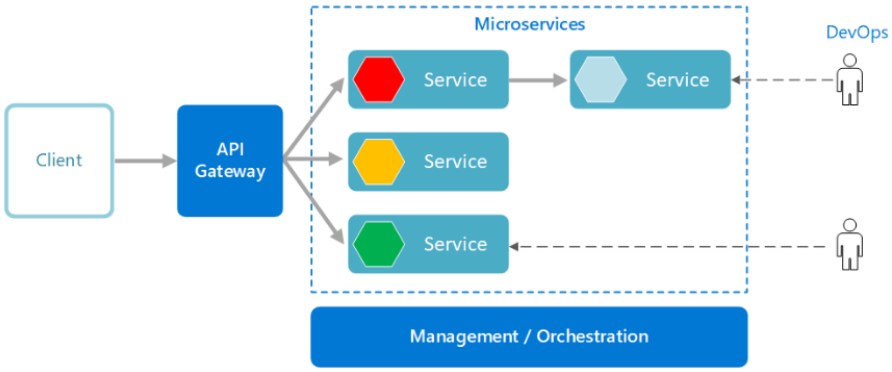

Figure 2: Microservices architecture Illustration (From docs.microsoft.com)

In this assignment you will learn how to setup your backend App to have basic token-based authentication and authorization in-place.
First make sure you have the starter app as you will gradually extend it to implement auth. functionalities using JSON Web Tokens (JWTs).

Difficulty: ☆☆☆☆

Learning objectives:

- I can encode/decode passwords using BCrypt
- I can generate signed JWT tokens with user claims and expiration time
- I know how to check the contents of a JWT
- I can validate a signed JWT
- I can secure endpoints based on authentication
- I can secure endpoints based on authorization
- I can retrieve the request's authenticated user
- I can test a secured controller

Step 1 - Password hashing: Implementing user registration

This assignment starts with adding the possibility to create an account by a student. The start-up project already contains the required code for the repository but is missing the actual code for the endpoint and service. In addition, the Spring Security has been included as a dependency in the project which allows Spring to take care of authentication and authorization.

What remains is for you to implement the following:

> **User story:** *As a* student, *I can* register for an account *so that* I can login to the app
>
> **Acceptance criteria:**
> - Registering for an account requires a unique username and a password
> - An appropriate message should be displayed when:
>   - the username is not unique
>   - Password and repeated password do not match
> - An account also requires the role to be set as *student*, the date and time of when it was created and that it is active
> - Passwords must be safely stored

Part of the acceptance criteria is to ensure that password storage is done safely (e.g. via hashing); currently, the BCrypt hashing function stands-out as the industry standard and Spring Security also supports it.

While this support is 'out of the box', you will still need to explicitly configure it. You can do this by creating the following class inside the project:

```
package fontys.sem3.school.configuration.security;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

@Configuration
public class PasswordEncoderConfig {

    @Bean
    public PasswordEncoder createBCryptPasswordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

Now, whenever your app needs a *PasswordEncoder*, Spring will know it needs to inject the *BCryptPasswordEncoder* instance created in the method above; this is because it marked as a *Configuration* and *Bean*, respectively.

With this in place, you can start implementing the endpoint of this user story and test it via Postman and unit tests.

Step 2 - Authentication: Implementing login with JWT support

The next step is to extend the app with an authentication feature by implementing the following user story:

> **User story:** *As a* user, *I can* login to the app *so that* I make use of the app
>
> **Acceptance criteria:**
> - Login done by supplying a valid username and password of an active account
> - An appropriate message should be displayed when login is not possible
> - The stateless nature of REST must be maintained

Based on semester 2, you probably have been working with sessions/authentication cookies to make your web application aware of user sessions (e.g. after login a session id in a cookie is returned). This principle should not be applied for a RESTful API as it would make it stateful while REST should be completely *stateless*.

To maintain stateless you can, for example, implementing support for signed JSON Web Tokens (JWT) as *access tokens*. This will require the login feature to return an access token which then will be included with each subsequent request.
A response after successful login could return something like this:

```
{
    "accessToken": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiIxMjM0QGZvbnR5cy5ubCIsImlhdCI6MTY1MTQ0MDg5MCwiZXhwIjoxNjUxNDQyNjkwLCJzdHVkZW50SWQiOjEsInJvbGVzIjpbIlNUVURFTlQiXX0.HqrN9R8ZDpl1D-ZVtux9yBhry1ifm7OiW-RsC3vQb0A"
}
```

`"accessToken"` is a signed JWT and its value is not readable because it is base64 encoded. *base64* is a binary-to-text encoding which doesn't really add to the security of the token. It instead makes it very convenient to send the access token in a *Authorization* HTTP header in the upcoming requests.

Note that the value of the token can easily be inspected by using jwt.io.(see Figure 4). Notice that the token has three parts: header, payload and signature. The signature cannot be verified by jwt.io because the *secret* needed is only known in the backend (and in this case has more than 256bits).

The purpose of this *signature* is to verify that the payload has not been tempered with. You can find the *secret* at *src/main/resources/application.properties*:

```
jwt.secret=E91E158E4C6656F68B1B5D1C316766DE98D2AD6EF3BFB44F78E9CFCDF5
```

The start-up project makes use of the HS256 (HMAC + SHA-256) algorithm and it is highly recommended to use strong and lengthy secrets keys, like the one above. It is also possible to sign JWTs with a public/private key-pair algorithm like RS256, which would increase the security by a lot.

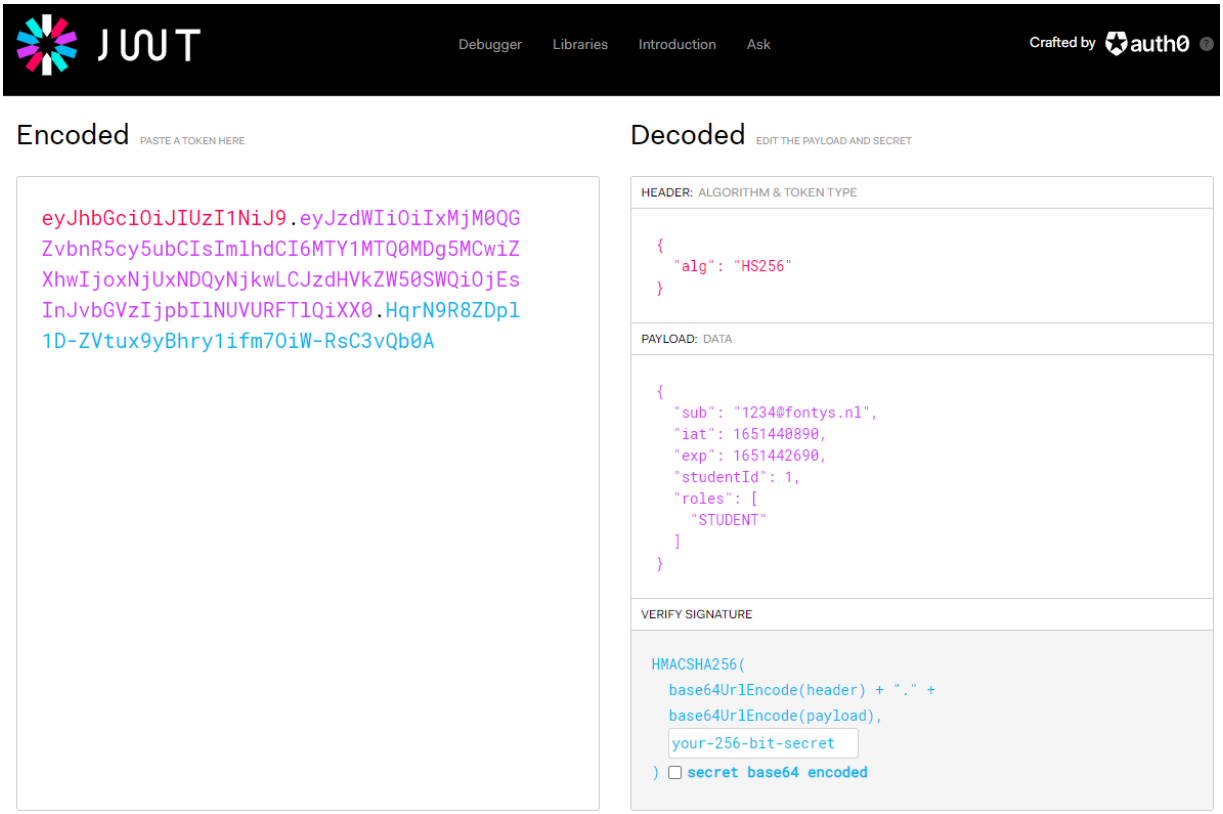

Figure 3: Decoded JWT

When looking at the start-up project answer for yourself the following questions:

a) Can you identify which class is generating the access token?
b) How is property “jwt.secret” injected into this class?
c) What is the expiration time of the token? Would you be able to change it?
d) How are claims added to the JWT? Would you be able to add your own if needed?

When considering all this, you can start implementing the user story as depicted in Figure 4*’s* activity diagram; make sure you test everything with Postman, unit tests and jwt.io.

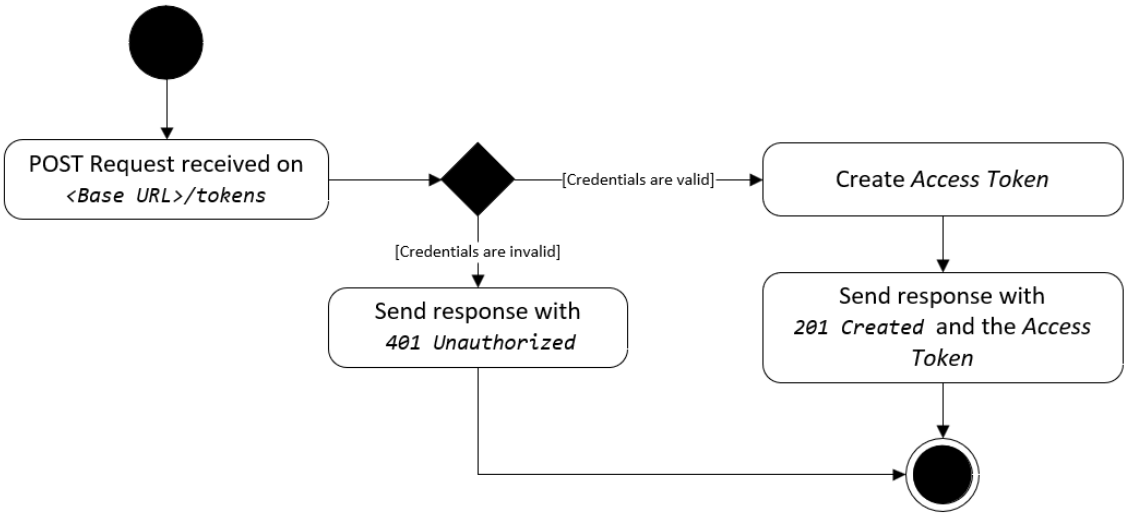

Figure 4: Login workflow

Step 3 - Authorization: Adding required configuration

With the login in place, you will now start extending the app by introducing a secured endpoint by implementing the following user story:

> **User story:** *As a* user, *I can* see an overview of all the active users *so that* I can later select a user to do further interaction with.
>
> **Acceptance criteria:**
> - Only show the username and role in the overview
> - Can only be accessed by authenticated users

The project already has a few security configurations in-place:

- *WebSecurityConfig*: disables CSRF (enabled by default for Spring MVC apps), disables the login form (our app is only about rest services after all), and sets the session management to stateless (no session at all).
  It also configures all endpoints to be private (i.e. requires authentication) instead of having all of them public. The reason being that we want to follow a whitelisting principle instead of blacklisting.
- *AuthenticationRequestFilter*: this filter will handle HTTP requests before they land in the controllers. It will look for a JWT token inside an authentication bearer HTTP header. If one is found then it will be verified, decoded and the authenticated user will be configured into Spring Security context.

This is how a request with a bearer token looks like in curl:

```
curl --location --request GET 'http://localhost:8080/students/1' --
header 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiIxMjM0QGZvbnR5cy5ubCIs
ImlhdCI6MTY1MTQ0MDg5MCwiZXhwIjoxNjUxNDQyNjkwLCJzdHVkZW50SWQiOjEsInJvbGVzIjpbIlNUVUR
FTlQiXX0.HqrN9R8ZDpl1D-ZVtux9yBhry1ifm7OiW-RsC3vQb0A'
```

Start implementing the user story and then can confirm that the newly created endpoint is secured (e.g. with Postman). You can do this by sending a request to the endpoint and confirm that an unauthorized response has been send back.

To gain access to this endpoint you will first have to get an access token by logging in, but the problem is that this endpoint is also protected!

Your next step is to modify the security configuration to make the login feature public. In addition, what other feature(s) should you make public?

When you have configured everything correctly, you can now get the aforementioned access token and include it in the header of the request to get all users; TIP: Have a look at the curl example to see how it should be included in the header.

Step 4 - Authorization: Protecting an endpoint based on role or specific user

Often an app requires finer grained authorization than just whether a user is authenticated. For example, the following user story:

> **User story:** *As a* user, *I can* delete an account *so that* that user cannot make use of the app anymore.
>
> **Acceptance criteria:**
> - An account should be deactivated instead of being removed from the app
> - Only an admin or the owner of that account can delete an account

To be able to implement this feature you need to know how to determine 'who' the authenticated user is by looking at its role and unique identifier (i.e. the user id). First, try to answer for yourself the following questions:

a) Can you rely on the *id* supplied in a DELETE request to be the same as the authenticated user?
b) How does the app 'know' a request has an authenticated user?
c) What is *UserDetails* in *AuthenticationRequestFilter.setupSpringSecurityContext(AccessToken)*

With the way the project has been set-up, it is quite easy to determine the role of an authenticated user. The only thing you have to do is make use of the @RolesAllowed annotation to configure which roles can accessible an endpoint.

Getting the actual authenticated user requires you to inject the access token. In the class *RequestAuthenticatedUserProvider* (inside *configuration/security/auth*) the following method provides the *AccessToken* of the logged-in user, if any:

```
@Bean
@Scope(value = WebApplicationContext.SCOPE_REQUEST, proxyMode =
ScopedProxyMode.TARGET_CLASS)
public AccessToken getAuthenticatedUserInRequest() {
    …
}
```

Notice that the returned token has scope REQUEST. This will allow the injection of the *AccessToken* into other classes and its value will be evaluated at runtime for each http request. This is only possible by injecting a proxy into the target class (parameter *proxyMode* above); harness the power IoC!

Now you only have to include the required code at the correct place to gain access to the access token of the authenticated user. TIP: how did you have your services injected into your controller?

When considering all this, you can start implementing the user story as depicted in Figure 5Figure 4*'s* activity diagram; make sure you test everything with Postman and unit tests.

Note that you will have to include the admin role yourself.

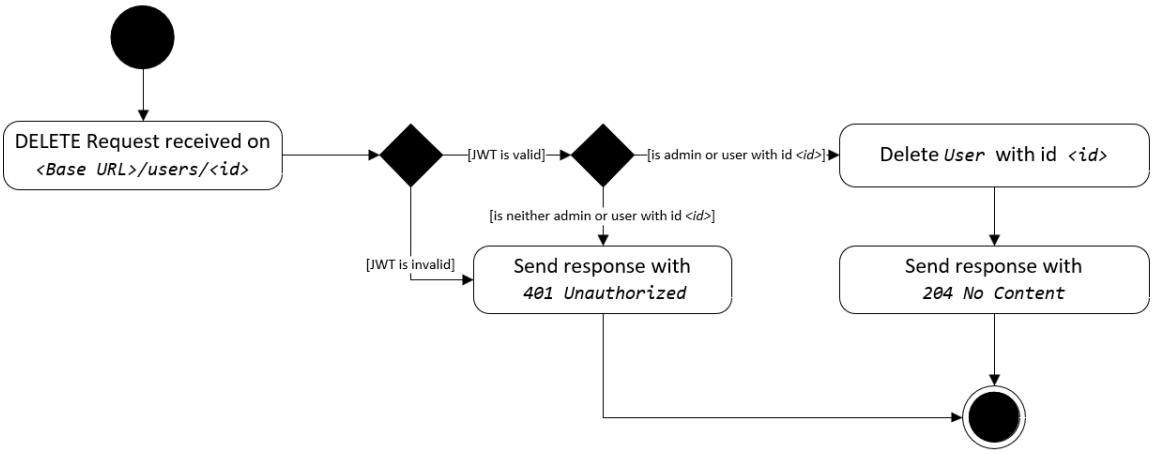

Figure 5: Delete user workflow

Final step: Follow-up questions and frontend integration

Hope you enjoyed the ride so far! Evaluate if this kind of authentication/authorization suits your project's requirements and if it is the case, time to apply it there!

Realize that the *jwt.secret* in *application.properties* should be unique per application (and environment). Make sure you create your own secret there!

Also make sure you have the following dependencies in your project's *build.gradle* file:

```
implementation 'org.springframework.boot:spring-boot-starter-security'
implementation 'io.jsonwebtoken:jjwt-api:0.11.5'
runtimeOnly 'io.jsonwebtoken:jjwt-impl:0.11.5'
runtimeOnly 'io.jsonwebtoken:jjwt-jackson:0.11.5'
testImplementation 'org.springframework.boot:spring-boot-starter-test'
testImplementation 'org.springframework.security:spring-security-test'
```

In addition, please consider the following questions:

- How do you test your secured controller? This will require that all the security configurations are properly bootstrapped by Spring. The easiest way of doing it is by starting a full-fledged integration test and then mocking the controller's business layer and authenticated user.
  You should also have a look at the annotations *@SpringBootTest* and *@AutoConfigureMockMvc* for you unit test class
- How can access tokens be securely stored in the frontend? What are the pros, cons and security risks of each approach (for instance: local storage, in-memory and cookies)?
- After authentication, would something like an axios interceptor be useful for sending the access token in all subsequent requests?
- What is a refresh token and how can it be helpful?

Discuss your answers with your teachers. Supporting refresh tokens in your solution is a plus (only work on it if you have the time), but understanding the concept is expected from you.

Note:

- sending tokens via cookies is NOT an acceptable solution because of CSRF attacks.
- local storage and in-memory approaches are ok for this course, but make sure you know for which kind of attack they are susceptible.