seg3102 – Lab 9
Authentication & Authorization

The objective of this lab is to introduce Web security, particularly authentication and authorization. We first look at the basics of Spring Security by adding authentication to a Spring MVC application. We then add JSON Web Token (JWT) based authentication and authorization to a REST API and update an Angular Client Application that consumes the API, to use JWT.

1. Basics of Spring Security

We consider the Temperature Converter application developed as part of Lab2. The current implementation does not have security. Anyone can access through the application URL. We will add security so that only authenticated users are allowed to use the application temperature conversion functionality. The source code for this part of the lab is available at https://github.com/stephanesome/tempconverter-security.

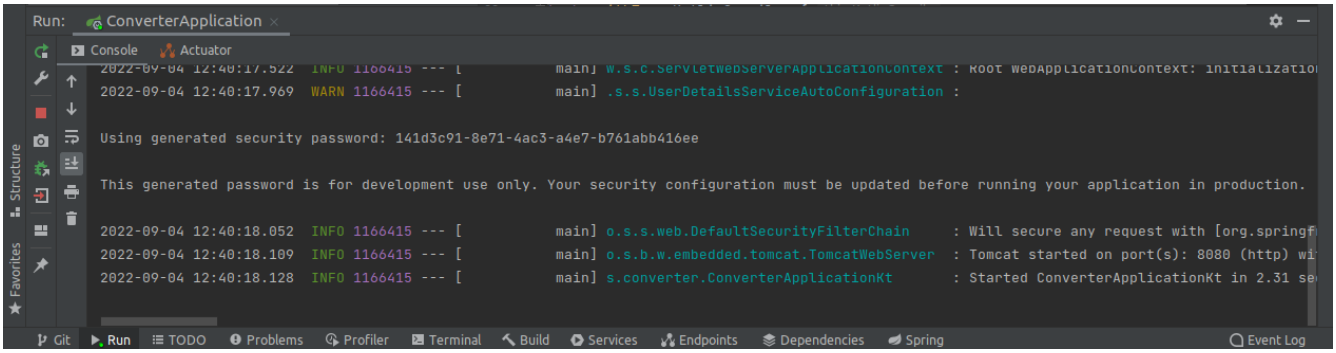
1.1. Spring Security setup

To enable Spring Security, edit build.gradle.kts and add the spring-boot-starter-security dependency.

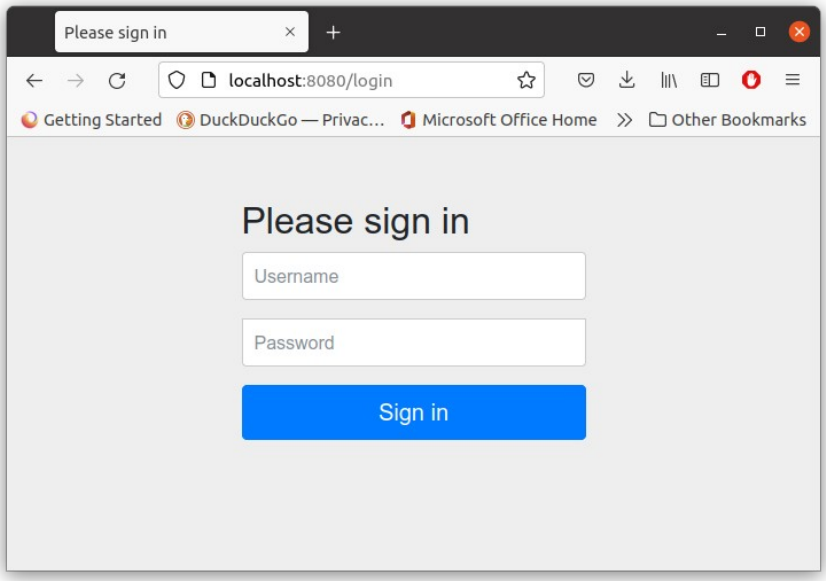
```
dependencies {
        ...
        implementation("org.springframework.boot:spring-boot-starter-security")
}
```

All the HTTP endpoints are automatically secured with just the addition of the dependency.

Build and run the application. Spring setup an application user with username '*user*' and a randomly generated password which is displayed with the application launch messages.



Take note of the password and navigate to http://localhost:8080. A default login page is presented asking for a username and password.



Authenticate with username '*user*' and the generated password.

1.2. In memory authentication

Spring supports a range of authentication approaches including In-Memory Authentication, JDBC Authentication, DAO Authentication, LDAP Authentication, Active Directory Authentication, ... In this section, we present In-Memory Authentication a simple username/password based authentication approach in which user credentials are stored in memory.

In order to customize the application security, we need to setup security configuration. This allows among others to specify our own user credentials as well as a custom login screen.

Security Configuration

The configuration of Spring Security is done by creating specific Beans.

Create a class WebSecurityConfig in package seg3x02.converter and edit as follow.

```
1. package seg3x02.converter
2.
3. import org.springframework.context.annotation.Bean
4. import org.springframework.context.annotation.Configuration
5. import org.springframework.security.config.annotation.web.builders.HttpSecurity
6. import org.springframework.security.config.annotation.web.builders.WebSecurity
7. import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity
8. import org.springframework.security.config.annotation.web.configuration.WebSecurityCustomizer
9. import org.springframework.security.core.userdetails.User
```

```
10. import org.springframework.security.core.userdetails.UserDetails
11. import org.springframework.security.core.userdetails.UserDetailsService
12. import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder
13. import org.springframework.security.provisioning.InMemoryUserDetailsManager
14. import org.springframework.security.web.SecurityFilterChain
15.
16. @Configuration
17. @EnableWebSecurity
18. class WebSecurityConfig {
19.     @Bean
20.     @Throws(Exception::class)
21.     fun securityFilterChain(http: HttpSecurity): SecurityFilterChain {
22.         http.
23.             authorizeRequests()
24.                 .anyRequest().authenticated()
25.             .and()
26.                 .formLogin()
27.                 .loginPage("/login")
28.                 .permitAll()
29.             .and()
30.                 .logout()
31.                 .permitAll()
32.         return http.build()
33.     }
34.
35.     @Bean
36.     fun userDetailsService(): UserDetailsService {
37.         val user: UserDetails = User.withUsername("appuser")
38.             .password(passwordEncoder().encode("userpassword"))
39.             .roles("USER")
40.             .build()
41.         return InMemoryUserDetailsManager(user)
42.     }
43.
44.     @Bean
45.     fun webSecurityCustomizer(): WebSecurityCustomizer {
46.         return WebSecurityCustomizer { web: WebSecurity ->
47.             web.ignoring()
48.                 .antMatchers("/resources/**", "/static/**", "/css/**", "/js/**", "/images/**","/vendor/**","/fonts/**") }
49.     }
50.
51.     @Bean
52.     fun passwordEncoder(): BCryptPasswordEncoder {
53.         return BCryptPasswordEncoder()
54.     }
55. }
```

We setup security in a configuration class decorated with @EnableWebSecurity. The configuration class creates a set of Beans:

- The SecurityFilterChain Bean (lines 21-33) configures HTTPSecurity to specify required the authorities for the application HTTP endpoints. We specify here that every request except login and logout must be authenticated. The configuration also specifies the login page.
- The UserDetailsService Bean (lines 36-42) specifies the application users credentials. These credentials are stored in an in-memory data store (line 41). Passwords are encoded using the BCryptPasswordEncoder encoder (lines 52-54).
- The WebSecurityCustomizer Bean (lines 44-49) provides a customization to ignore the static resources such as CSS, Images Javascripts.

Login Page

Create a HTML Template login.html in src/main/resources/templates and edit as follow.

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.      <meta charset="UTF-8">
5.      <title>Temperature Converter Login</title>
6.      <link rel="stylesheet" th:href="@{/css/style.css}" />
7.  </head>
8.  <body>
9.  <div th:if="${param.error}">
10.     <p>Invalid username and password, try again</p>
11. </div>
12. <div th:if="${param.logout}">
13.     <p>You have been successfully logged out</p>
14. </div>
15. <form th:action="@{/login}" method="post">
16.     <div><label> User Name : <input type="text" name="username"/> </label></div>
17.     <div><label> Password: <input type="password" name="password"/> </label></div>
18.     <div><input type="submit" value="Sign In"/></div>
19. </form>
20. </body>
21. </html>
```

The template presents a form for a username and password and post to /login (lines 15-19). We also display appropriate messages for unsuccessful login (line 10) as well as acknowledgement of logout (line 13) by checking the redirection parameter provided in these circumstances.

Login Controller

We register a view controller for the login view. This is a simpler approach. Alternatively, we could create a controller handler function.

Create a class LoginController in package seg3x02.converter and edit as follow.

```
1.  package seg3x02.converter
2.
```

```
import org.springframework.context.annotation.Configuration
import org.springframework.web.servlet.config.annotation.ViewControllerRegistry
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer

@Configuration
class LoginController: WebMvcConfigurer {
    override fun addViewControllers(registry: ViewControllerRegistry) {
        registry.addViewController("/login").setViewName("login")
    }
}
```

LoginController implements the interface WebMvcConfigurer and defines the callback addViewControllers to register the view *login* for the /login path.

Logout

We update the application main page to provide a logout option to the user. Edit src/main/resources/templates/hello.html as follow

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Temperature Converter</title>
    <link href="../static/css/style.css" rel="stylesheet"
          th:href="@{/css/style.css}" />
</head>
<body>
<div>
    <h1>Temperature Converter</h1>
    <form th:action="@{/logout}" method="post">
        <input type="submit" value="Sign Out"/>
    </form>
    <div th:switch="${error}">
        <p th:case="'FahrenheitFormatError'">
            Wrong value provided for Fahrenheit - must be a number
        </p>
        <p th:case="'CelsiusFormatError'">
            Wrong value provided for Celsius - must be a number
        </p>
        <p th:case="'OperationFormatError'">
            Wrong operation
        </p>
    </div>
    <form th:action="@{/convert}" method="get">
        <table>
            <tr>
                <td><label for="celsius">Celsius:</label></td>
                <td><input name="celsius"
                           id="celsius"
```

```
32.                     th:name="celsius" th:value="${celsius}"></td>
33.         </tr>
34.         <tr>
35.            <td><label for="fahrenheit">Fahrenheit:</label></td>
36.            <td><input name="fahrenheit"
37.                    id="fahrenheit"
38.                    th:name="fahrenheit" th:value="${fahrenheit}"></td>
39.         </tr>
40.         <tr>
41.            <td><button type="submit" th:name="operation" th:value="CtoF">
42.               Celsius to Fahrenheit</button></td>
43.            <td><button type="submit" th:name="operation" th:value="FtoC">
44.               Fahrenheit to Celsius</button></td>
45.         </tr>
46.      </table>
47.   </form>
48. </div>
49. </body>
50. </html>
```

We added a from with a submit button to POST to /logout (lines 12-14). This triggers the application to sign the user out and redirects to the login page.

2. JSON Web Token (JWT)

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. A Client requests a Token from a Server by checking its credentials (i.e. login id and password). If successful, the Server generates a Token that is returned to the Client. The Client then, send the Token with subsequent requests without having to provide their credentials again.

A JSON Web Token consists on a Header, Payload and Signature. The Header specifies the type of Token as well the hashing algorithm used. The Payload carries data to be exchanged. For instance, it may contain a user name and role. Finally, the Signature is a hash of the Header and Payload using the hashing algorithm specified in the header and a Secret only known to the Server. The hashing ensures a JWT can not be tampered with and changed on transit. The payload information is however only Base64 encoded. Encryption should be used when sensitive information is carried.

The source code for this part of the lab is available at https://github.com/stephanesome/apiJWTSecurity. The Angular Client application is in branch *client* and the source code for the SpringBoot Server application in branch *server*.

2.1. Securing Books API with JWT

We will update the Books API from Lab 7. We will add endpoints for sign-in and sign-up.

A user will sign-up by sending a POST request to URI `/auth/signup` with a username, password and optional role. The server will then keep the user's credentials to a database.

In order to authenticate, users will POST a request to URI `/auth/signin` with their username and password. The server will check the user credentials from the database and if successful, send a response with a JWT.

Any request to the URI `/books-api` will need a valid JWT. The token is sent in the Authorization header using the Bearer schema as follow:

Authorization: Bearer <token>

We distinguish two roles: USER and ADMIN. Both roles are allowed to make GET requests, but only users with an ADMIN role can make POST, PUT, PATCH, and DELETE requests to URI `/books-api`.

Configuration

We add dependencies for Spring Security and JJWT (https://github.com/jwtk/jjwt), a library for creating and verifying JSON Web Tokens (JWTs). Edit `build.gradle.kts` and add the following implementation dependencies:

```
dependencies {
        ...
        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")
        ...
}
```

Edit `src/main/resources/application.properties` as follow.

```
# Properties for MYSQL
spring.datasource.url=jdbc:mysql://localhost:6033/booksDb?serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=update
spring.sql.init.platform=mysql
# Properties for JWT
app.jwtSecret= QXBwbGljYXRpb25fU2VjcmV0X0tleV9YWFhfNTQyM19ZWVk4bWduYVpwWDJlZGl...
app.jwtExpirationMs= 86400000
```

We set a key string and an expiration delay for tokens. The key string must be Base64 encoded and long enough for the signature algorithm (https://github.com/jwtk/jjwt#signed-jwts). The expiration

delay is a time amount after which a Users will have to re-authenticate to acquire a new token. This is a security measure to mitigate the risk of malicious token reuse.

User Model

Create a package named `security` in `seg3x02.booksrestapi` and a sub-package named `credentials` in the `security` package. Create an Enum class named ERole and an entity class named User in `seg3x02.booksrestapi.security.credentials`. Modify the ERole class as follows.

```
package seg3x02.booksrestapi.security.credentials

enum class ERole {
    ROLE_USER,
    ROLE_ADMIN
}
```

The class ERole specifies the two roles for the users.

Edit class User as follow.

```
package seg3x02.booksrestapi.security.credentials

import javax.persistence.*
import javax.validation.constraints.NotBlank
import javax.validation.constraints.Size

@Entity
@Table(name = "users", uniqueConstraints = [UniqueConstraint(columnNames = ["username"])])
class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    var id: Long = 0

    @NotBlank
    @Size(max = 20)
    var username: String = ""

    @NotBlank
    @Size(max = 120)
    var password: String = ""

    @Enumerated(EnumType.STRING)
    var role: ERole = ERole.ROLE_USER

    constructor() {}
    constructor(username: String, password: String) {
        this.username = username
        this.password = password
```

```
29.     }
30. }
```

The User class is a JPA entity class to keep in the database. The @Table annotation (line 8) defines a table name and a constraint for a unique user name value in the database.

User Repository

Create UserRepository, a repository interface for the User entity in package `seg3x02.booksrestapi.repository`. Edit as follow.

```
1.  package seg3x02.booksrestapi.repository
2.
3.  import org.springframework.data.repository.CrudRepository
4.  import seg3x02.booksrestapi.security.credentials.User
5.  import java.util.*
6.
7.  interface UserRepository: CrudRepository<User, Long> {
8.      fun findByUsername(username: String): Optional<User>
9.
10.     fun existsByUsername(username: String): Boolean
11. }
```

We added two custom queries to the default CrudRepository queries: findByUsername to find a user based on a username, and existsByUsername to verify the existence of a user with a given username in the database. These queries are specified according to a naming convention used by Spring Data for automated generation.

Spring Security Setup

We setup security in a configuration class decorated with @EnableWebSecurity. The configuration class creates a set of Beans:

- AuthenticationFilter Bean to intercept HTTP requests in order to extract and check the JWT token,
- AuthenticationManager Bean that specifies the authentication manager,
- PasswordEncoder Bean specifies the encoding algorithm used to encrypt passwords before storing in the database,
- SecurityFilterChain Bean to specify required authorities and other security protections such as CORS.

UserDetails

User information are represented as instances of classes that implement interface UserDetails.

Create class UserDetailsImpl in package `seg3x02.booksrestapi.security` and edit as follow.

```
1.  package seg3x02.booksrestapi.security
2.
3.  import com.fasterxml.jackson.annotation.JsonIgnore
4.  import org.springframework.security.core.GrantedAuthority
5.  import org.springframework.security.core.authority.SimpleGrantedAuthority
6.  import org.springframework.security.core.userdetails.UserDetails
7.  import seg3x02.booksrestapi.security.credentials.User
8.
9.  class UserDetailsImpl(val id: Long, private val username: String,
10.                 @field:JsonIgnore private val password: String,
11.                 private val authorities: Collection<GrantedAuthority>) : UserDetails {
12.
13.     override fun getAuthorities(): Collection<GrantedAuthority> {
14.         return authorities
15.     }
16.
17.     override fun getPassword(): String {
18.         return password
19.     }
20.
21.     override fun getUsername(): String {
22.         return username
23.     }
24.
25.     override fun isAccountNonExpired(): Boolean {
26.         return true
27.     }
28.
29.     override fun isAccountNonLocked(): Boolean {
30.         return true
31.     }
32.
33.     override fun isCredentialsNonExpired(): Boolean {
34.         return true
35.     }
36.
37.     override fun isEnabled(): Boolean {
38.         return true
39.     }
40. }
41.
42. fun build(user: User): UserDetailsImpl {
43.     val authorities = ArrayList<GrantedAuthority>()
44.     authorities.add(SimpleGrantedAuthority(user.role.name))
45.     return UserDetailsImpl(
46.         user.id,
47.         user.username,
48.         user.password,
49.         authorities)
```

```
50. }
```

Class UserDetailsImpl is an implementation of interface UserDetails. We also provide a factory function to build a UserDetailsImpl instance from a User entity (lines 42-50). The factory creates a list with a GrantedAuthority set from the User's role. This set of GrantedAuthority determines the Authorization of the user.

UserDetailsService

The UserDetailsService provides a function loadUserByUsername that returns a UserDetails constructed from a user retrieved from the database.

Create class UserDetailsServiceImpl in package `seg3x02.booksrestapi.security` and edit as follow.

```
1.  package seg3x02.booksrestapi.security
2.
3.  import org.springframework.security.core.userdetails.UserDetails
4.  import org.springframework.security.core.userdetails.UserDetailsService
5.  import org.springframework.security.core.userdetails.UsernameNotFoundException
6.  import org.springframework.stereotype.Service
7.  import seg3x02.booksrestapi.repository.UserRepository
8.  import seg3x02.booksrestapi.security.credentials.User
9.  import javax.transaction.Transactional
10.
11. @Service
12. class UserDetailsServiceImpl(val userRepository: UserRepository): UserDetailsService {
13.
14.     @Transactional
15.     @Throws(UsernameNotFoundException::class)
16.     override fun loadUserByUsername(username: String): UserDetails {
17.         val user: User = userRepository.findByUsername(username)
18.                 .orElseThrow { UsernameNotFoundException("User username: $username not found") }
19.         return build(user)
20.     }
21. }
```

Class UserDetailsServiceImpl implements interface UserDetailsService. Method loadUserByUsername returns a UserDetails from a User retrieved from the database based on a username, or returns an error if such an User is not found (lines 17-19). The UserDetails is created from the User entity using the UserDetails factory function build (line 19).

Authentication Entry

Create class AuthenticationEntry in package `seg3x02.booksrestapi.security` and edit as follow.

```
1.  package seg3x02.booksrestapi.security
```

```
import org.springframework.security.core.AuthenticationException
import org.springframework.security.web.AuthenticationEntryPoint
import org.springframework.stereotype.Component
import javax.servlet.http.HttpServletRequest
import javax.servlet.http.HttpServletResponse

@Component
class AuthenticationEntry: AuthenticationEntryPoint {
    override fun commence(request: HttpServletRequest,
                          response: HttpServletResponse,
                          authException: AuthenticationException) {
        response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Error: Unauthorized")
    }
}
```

This class implements the AuthenticationEntryPoint interface. The server will trigger method commence to notify clients that authentication is required when an unauthenticated attempt is made to access a resource that needs authentication. Response code SC_UNAUTHORIZED (401) is returned.

JWT Utility Class

Create class JwtUtils in a sub-package named `jwt` of `seg3x02.booksrestapi.security`. Edit class JwtUtils as follow.

```
package seg3x02.booksrestapi.security.jwt

import io.jsonwebtoken.JwtException
import io.jsonwebtoken.Jwts
import io.jsonwebtoken.io.Decoders
import io.jsonwebtoken.security.Keys
import org.springframework.beans.factory.annotation.Value
import org.springframework.security.core.Authentication
import org.springframework.stereotype.Component
import seg3x02.booksrestapi.security.UserDetailsImpl
import java.util.*

@Component
class JwtUtils {
    @Value("\${app.jwtSecret}")
    private val jwtSecret: String = ""

    @Value("\${app.jwtExpirationMs}")
    private val jwtExpirationMs = 0

    fun generateJwtToken(authentication: Authentication): String {
        val userPrincipal = authentication.principal as UserDetailsImpl
        val keyBytes = Decoders.BASE64.decode(jwtSecret)
        val key = Keys.hmacShaKeyFor(keyBytes)
```

```
25.         return Jwts.builder()
26.             .setSubject(userPrincipal.username)
27.             .setIssuedAt(Date())
28.             .setExpiration(Date(Date().time + jwtExpirationMs))
29.             .signWith(key)
30.             .compact()
31.     }
32.
33.     fun getUserNameFromJwtToken(token: String): String {
34.         val keyBytes = Decoders.BASE64.decode(jwtSecret)
35.         val key = Keys.hmacShaKeyFor(keyBytes)
36.         return Jwts.parserBuilder().setSigningKey(key).build().parseClaimsJws(token).body.subject
37.     }
38.
39.     fun validateJwtToken(authToken: String): Boolean {
40.         val keyBytes = Decoders.BASE64.decode(jwtSecret)
41.         val key = Keys.hmacShaKeyFor(keyBytes)
42.         try {
43.             Jwts.parserBuilder().setSigningKey(key).build().parseClaimsJws(authToken)
44.             return true
45.         } catch (e: JwtException) {
46.             println(e.stackTrace)
47.         }
48.         return false
49.     }
50. }
```

This class provides utilities to make it easier to work with Tokens. Function generateJwtToken (lines 21-31) creates a JWT. The created token expiration is set using the expiration delay property (line 28), while the hashing key property is used for signing the token (line 29).

Authentication Filter

Create class AuthenticationFilter in package `seg3x02.booksrestapi.security`. Edit as follow.

```
1.  package seg3x02.booksrestapi.security
2.
3.  import org.springframework.security.authentication.UsernamePasswordAuthenticationToken
4.  import org.springframework.security.core.context.SecurityContextHolder
5.  import org.springframework.security.web.authentication.WebAuthenticationDetailsSource
6.  import org.springframework.util.StringUtils
7.  import org.springframework.web.filter.OncePerRequestFilter
8.  import seg3x02.booksrestapi.security.jwt.JwtUtils
9.  import javax.servlet.FilterChain
10. import javax.servlet.http.HttpServletRequest
11. import javax.servlet.http.HttpServletResponse
12.
13. class AuthenticationFilter(var jwtUtils: JwtUtils,
```

```
14.                         var userDetailsService: UserDetailsServiceImpl) : OncePerRequestFilter() {
15.     override fun doFilterInternal(request: HttpServletRequest, response: HttpServletResponse, filterChain:
    FilterChain) {
16.         try {
17.             val jwt = parseJwt(request)
18.             if (jwt != null && jwtUtils.validateJwtToken(jwt)) {
19.                 val username: String = jwtUtils.getUserNameFromJwtToken(jwt)
20.                 val userDetails = userDetailsService.loadUserByUsername(username)
21.                 val authentication = UsernamePasswordAuthenticationToken(
22.                         userDetails, null, userDetails?.authorities)
23.                 authentication.details = WebAuthenticationDetailsSource().buildDetails(request)
24.                 SecurityContextHolder.getContext().authentication = authentication
25.             }
26.         } catch (e: Exception) {}
27.         filterChain.doFilter(request, response)
28.     }
29.
30.     private fun parseJwt(request: HttpServletRequest): String? {
31.         val headerAuth = request.getHeader("Authorization")
32.         return if (StringUtils.hasText(headerAuth) && headerAuth.startsWith("Bearer ")) {
33.             headerAuth.substring(7, headerAuth.length)
34.         } else null
35.     }
36. }
```

A Client request goes through the registered Filters before reaching a Controller. Class AuthenticationFilter extends class OncePerRequestFilter which implements the Filter interface. Function doFilterInternal in a OncePerRequestFilter is guaranteed to be executed just once per request. It retrieves the JWT from the request header (function parseJwt) and if valid, gets the UserDetails with corresponding username using the UserDetailsService (line 20). The function then creates an Authentication token (lines 21-23) and stores it in the Security Context. Notice that the authentication is not successful if UserDetailsService can not load a UserDetails.

Spring Security Configuration

Create class ApiSecurityConfig in package `seg3x02.booksrestapi.security`. Edit as follow.

```
package seg3x02.booksrestapi.security

import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.http.HttpMethod
import org.springframework.security.authentication.AuthenticationManager
import
org.springframework.security.config.annotation.authentication.configuration.AuthenticationConfiguration
import org.springframework.security.config.annotation.web.builders.HttpSecurity
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity
```

```
10. import org.springframework.security.config.http.SessionCreationPolicy
11. import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder
12. import org.springframework.security.crypto.password.PasswordEncoder
13. import org.springframework.security.web.SecurityFilterChain
14. import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter
15. import seg3x02.booksrestapi.security.jwt.JwtUtils
16.
17.
18. @Configuration
19. @EnableWebSecurity
20. class ApiSecurityConfig(var userDetailsService: UserDetailsServiceImpl,
21.                         var unauthorizedHandler: AuthenticationEntry,
22.                         var jwtUtils: JwtUtils) {
23.     @Bean
24.     fun authenticationJwtTokenFilter(): AuthenticationFilter {
25.         return AuthenticationFilter(jwtUtils,userDetailsService)
26.     }
27.
28.     @Bean
29.     @Throws(Exception::class)
30.     fun authenticationManager(authenticationConfiguration: AuthenticationConfiguration):
    AuthenticationManager {
31.         return authenticationConfiguration.authenticationManager
32.     }
33.
34.     @Bean
35.     fun passwordEncoder(): PasswordEncoder {
36.         return BCryptPasswordEncoder()
37.     }
38.
39.     @Bean
40.     fun configure(http: HttpSecurity): SecurityFilterChain {
41.         http.cors().and().csrf().disable()
42.             .exceptionHandling().authenticationEntryPoint(unauthorizedHandler).and()
43.             .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS).and()
44.             .authorizeRequests().antMatchers("/auth/**").permitAll()
45.             .antMatchers(HttpMethod.GET,"/books-api/**").hasAnyRole("USER", "ADMIN")
46.             .antMatchers(HttpMethod.POST, "/books-api/**").hasRole("ADMIN")
47.             .antMatchers(HttpMethod.DELETE, "/books-api/**").hasRole("ADMIN")
48.             .antMatchers(HttpMethod.PUT, "/books-api/**").hasRole("ADMIN")
49.             .antMatchers(HttpMethod.PATCH, "/books-api/**").hasRole("ADMIN")
50.             .anyRequest().authenticated()
51.         http.addFilterBefore(authenticationJwtTokenFilter(),
52.             UsernamePasswordAuthenticationFilter::class.java)
53.         return http.build()
54.     }
55. }
```

Annotation @EnableWebSecurity (line 19) allows Spring to find the class and apply it to the global Web Security.

Function configure configures the HttpSecurity Bean. In line 41, we setup CORS and disable CSRF protection (this protection is not needed for a HTTP API). The AuthenticationEntry is set for exception handling on line 42 and specifies a session creation policy at line 43. That session creation policy states that no session is to be created or used by Spring Security. This is in accordance with the *statelessness* constraint of a REST API. Each and every request will needs to be re-authenticated.

We configure the authorization parameters by matching URI and HTTP Methods with antMatchers. Access to the authentication URI is permitted with no authentication (line 44). In line 45, we specify that GET requests are allowed for users with role USER and ADMIN to any path starting with URI `/books-api`. Lines 46 – 49 restricts every other HTTP Method to role ADMIN. We specify that any request must be authenticated in line 50.

Finally, on lines 51-53, we add our AuthenticationFilter as being processed before the default Spring UsernamePasswordAuthenticationFilter.

Authentication Controller

The authentication controller exposes endpoints for registration and connection. We first provide data classes for the payload of requests and responses.

Payload

Create a package `payload` in `seg3x02.booksrestapi.controller`. Create classes SignInData, SignUpData, MessageResponse and AuthResponse in package `seg3x02.booksrestapi.controller.payload`, and edit as follow.

```
1. package seg3x02.booksrestapi.controller.payload
2.
3. data class SignInData(val username: String, val password: String)
```

SignInData captures the username and password sent with a login request.

```
1. package seg3x02.booksrestapi.controller.payload
2.
3. data class SignUpData(val username: String, val password: String, val role: String?)
```

SignUpData captures the username, password and role with a registration request.

```
1. package seg3x02.booksrestapi.controller.payload
2.
3. class MessageResponse(var message: String)
```

MessageResponse specifies a returned message response to notify the result of the registration.

```
1. package seg3x02.booksrestapi.controller.payload
2.
3. class AuthResponse(val token: String, val id: Long, val username: String, val role: String)
```

AuthResponse specifies a response returned for a successful sign in. The response includes the generated JWT string.

Controller

Create a class AuthenticationController in package `seg3x02.booksrestapi.controller`. Edit as follow.

```
package seg3x02.booksrestapi.controller

import org.springframework.http.ResponseEntity
import org.springframework.security.authentication.AuthenticationManager
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken
import org.springframework.security.core.Authentication
import org.springframework.security.core.context.SecurityContextHolder
import org.springframework.security.crypto.password.PasswordEncoder
import org.springframework.web.bind.annotation.*
import seg3x02.booksrestapi.controller.payload.AuthResponse
import seg3x02.booksrestapi.controller.payload.MessageResponse
import seg3x02.booksrestapi.controller.payload.SignInData
import seg3x02.booksrestapi.controller.payload.SignUpData
import seg3x02.booksrestapi.repository.UserRepository
import seg3x02.booksrestapi.security.UserDetailsImpl
import seg3x02.booksrestapi.security.credentials.ERole
import seg3x02.booksrestapi.security.credentials.User
import seg3x02.booksrestapi.security.jwt.JwtUtils
import javax.validation.Valid

@RestController
@CrossOrigin(origins = ["http://localhost:4200"])
@RequestMapping("/auth")
class AuthenticationController(val authenticationManager: AuthenticationManager,
                               val userRepository: UserRepository,
                               val encoder: PasswordEncoder,
                               val jwtUtils: JwtUtils) {
    @PostMapping("/signin")
    fun authenticateUser(@RequestBody loginRequest: @Valid SignInData): ResponseEntity<*> {
        val authentication: Authentication = authenticationManager.authenticate(
                UsernamePasswordAuthenticationToken(loginRequest.username, loginRequest.password))
        SecurityContextHolder.getContext().authentication = authentication
        val jwt = jwtUtils.generateJwtToken(authentication)
        val userDetails = authentication.principal as UserDetailsImpl
        val role = userDetails.authorities.elementAtOrNull(0)
        return ResponseEntity.ok<Any>(AuthResponse(jwt,
                userDetails.id,
                userDetails.username,
                role!!.authority))
    }

```

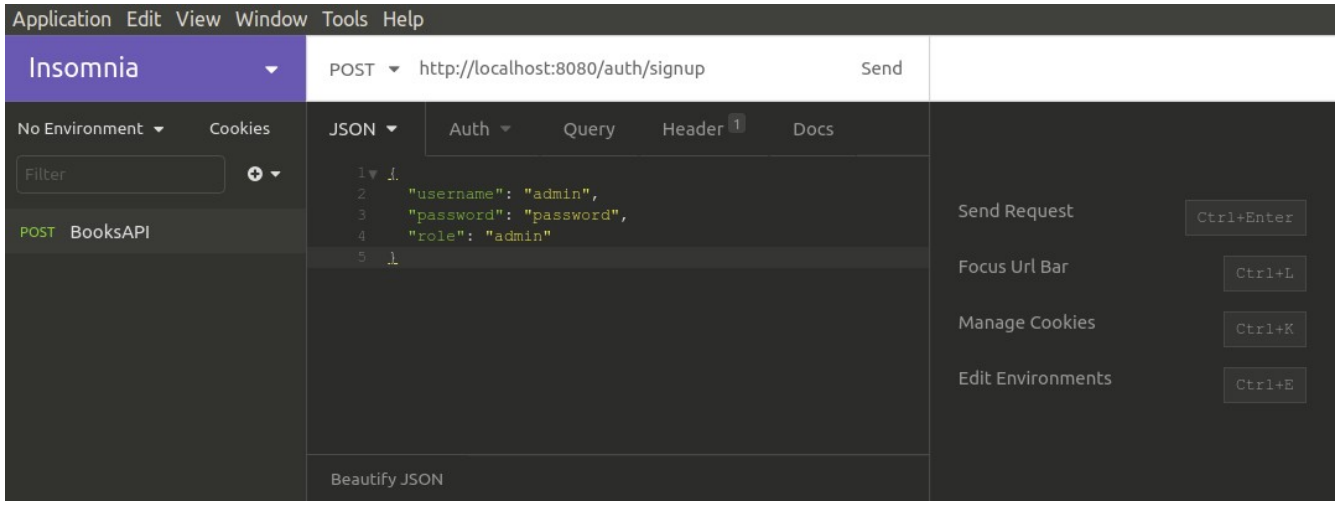
```
42.     @PostMapping("/signup")
43.     fun registerUser(@RequestBody signUpRequest: @Valid SignUpData): ResponseEntity<*> {
44.         if (userRepository.existsByUsername(signUpRequest.username)) {
45.             return ResponseEntity
46.                     .badRequest()
47.                     .body<Any>(MessageResponse("Error: Username is already taken!"))
48.         }
49.         val user = User(signUpRequest.username,
50.                 encoder.encode(signUpRequest.password))
51.         user.role = if ("admin" == signUpRequest.role) ERole.ROLE_ADMIN else ERole.ROLE_USER
52.         userRepository.save(user)
53.         return ResponseEntity.ok<Any>(MessageResponse("User registered successfully!"))
54.     }
55. }
```

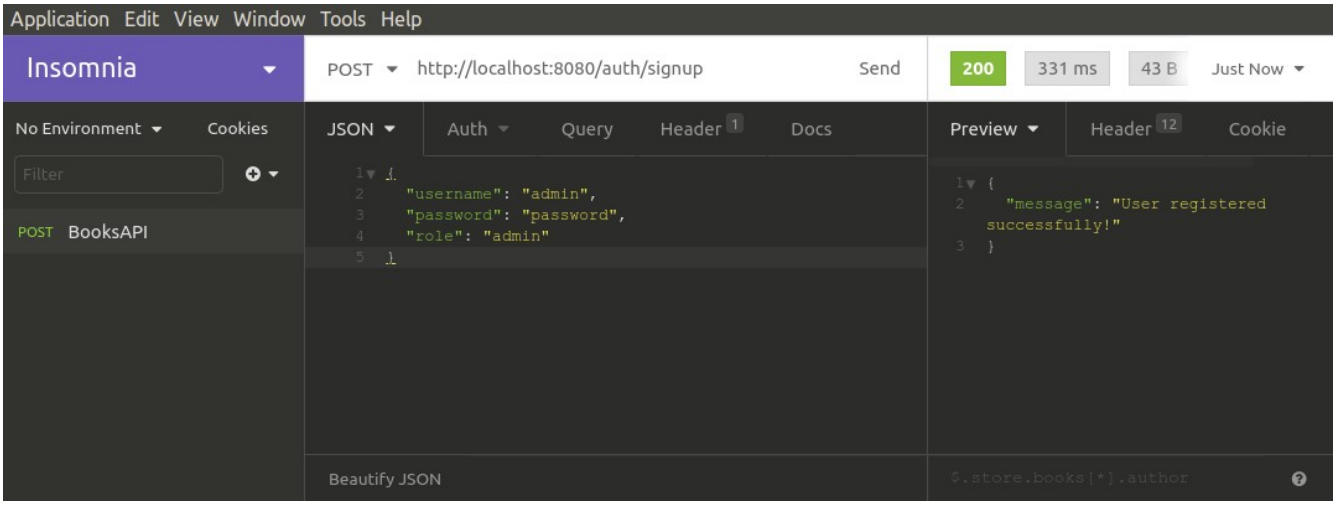
Function authenticateUser, the handler for *sign-in* uses the AuthenticationManager to authenticate the user based on the provided credentials. (lines 30-31). If the authentication is successful, a JWT is generated (line 33) and a response returned with the token as well as the user information (lines 34-39).

The user registration manager, the registerUser function, first checks for the existence of a user with the user name specified in the database (line 44). An error is returned as a response if a user already exists (lines 45 to 47). Otherwise, a new user is created. The user's role is defined as ADMIN if "admin" is provided in the request. Otherwise, it is defined as USER.
Note that this is not recommended in a deployed application. An administrator user should not be defined using an open API like here. It is better to use a protected approach where the database is directly updated with the credentials of the admins.

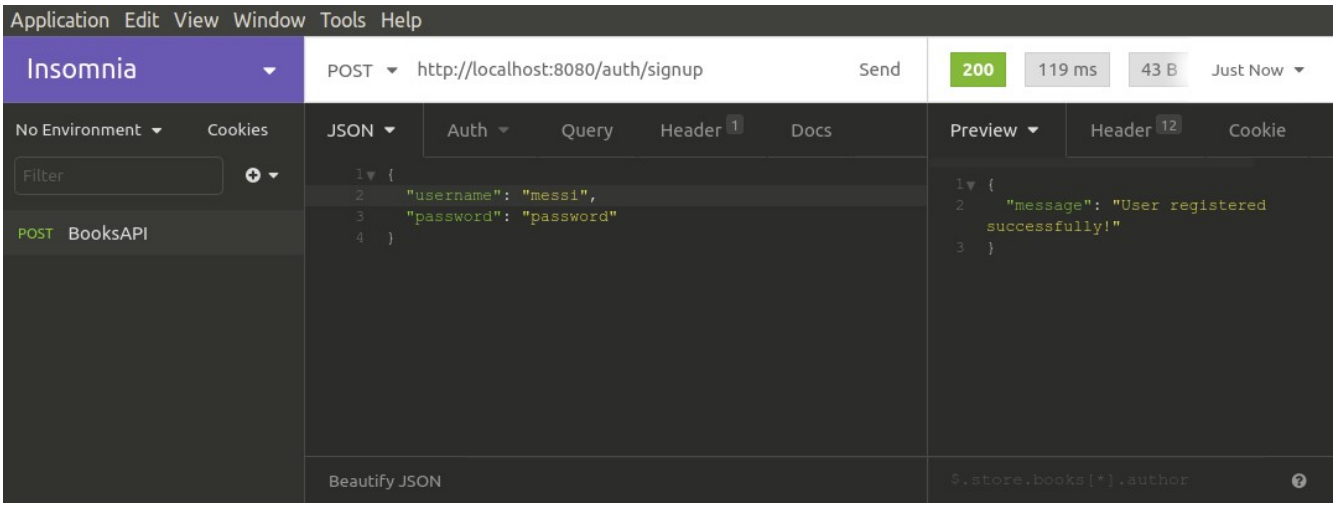
Building and Running

Build and run the application. Register an admin.

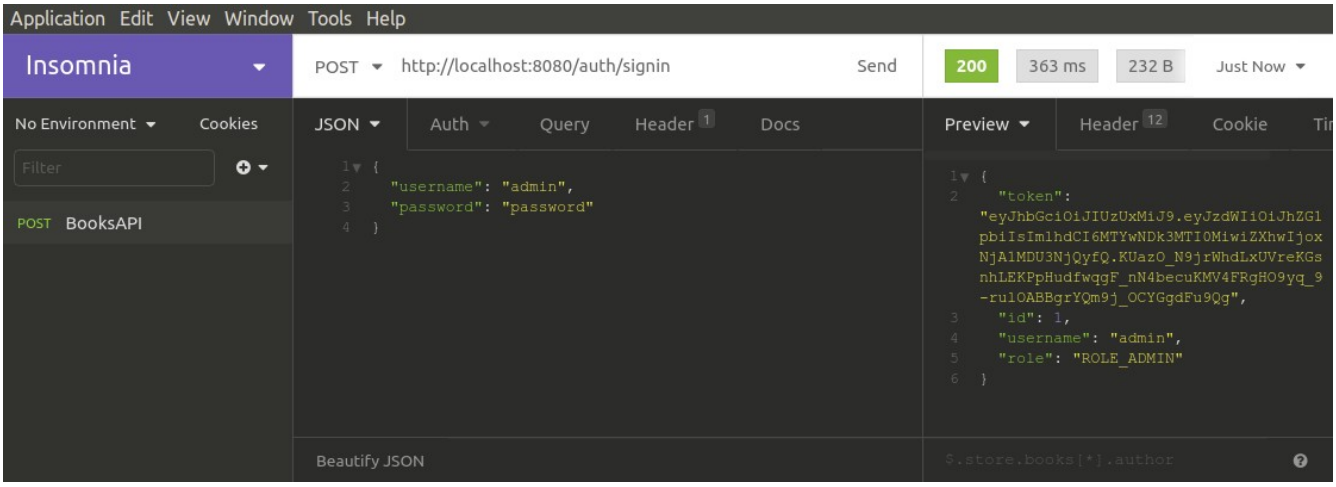




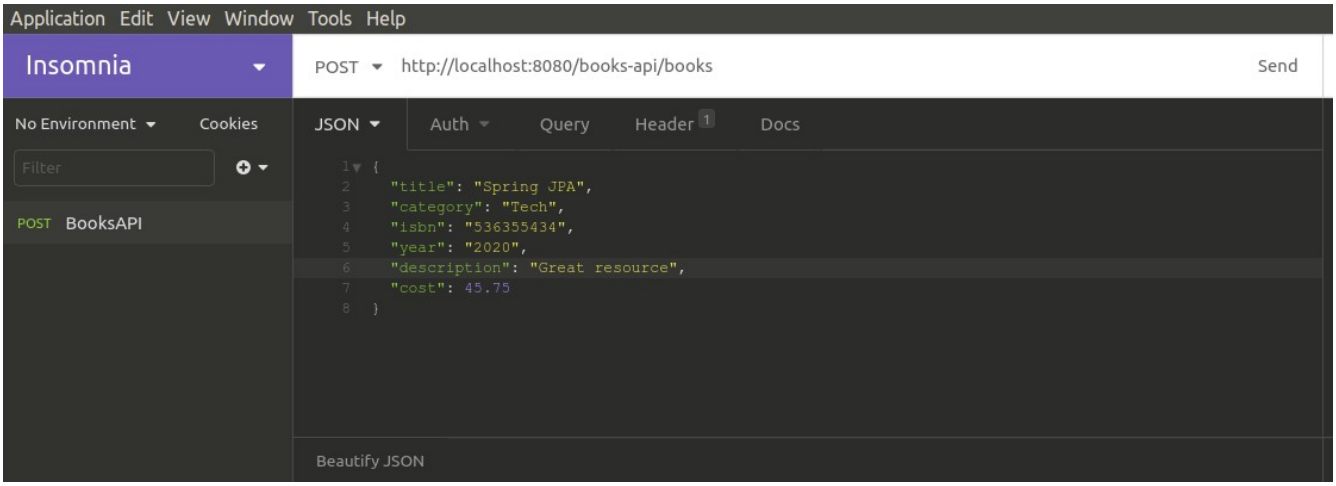
Register a user.

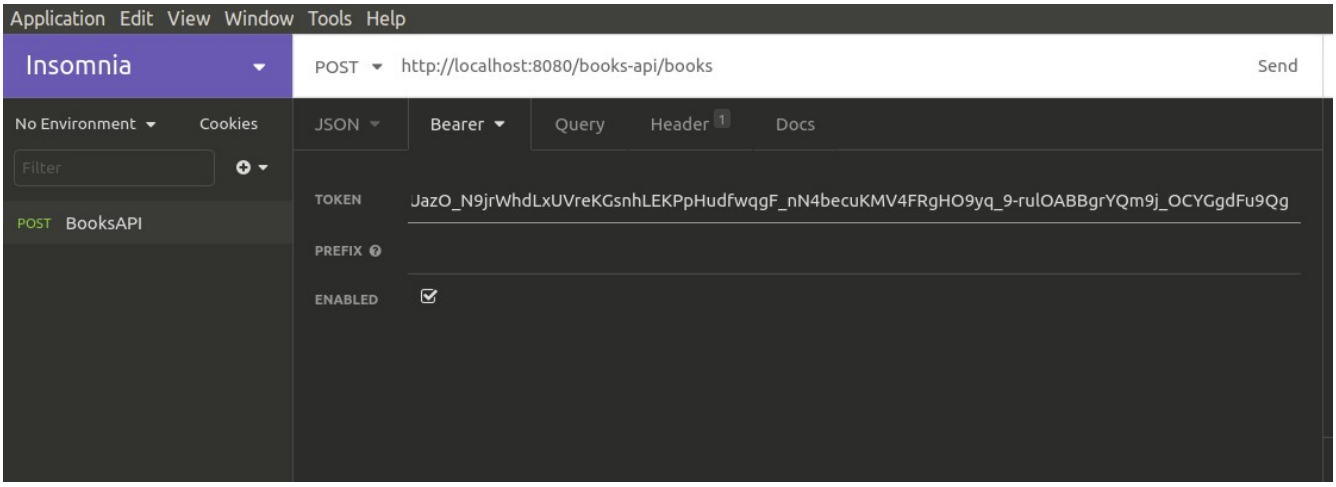


Sign-in



Perform a request to the `books-api`. You will need to include the token from the sign-in in the Authentication header of the request as *Bearer Token*.





2.2. Angular Client

We are going to update the Bookstore application from Lab7 for access to the secured API. We assume, only users authenticated as ADMIN can add books and users must be authenticated as ADMIN or USER to retrieve books. All other routes are accessible to unauthenticated users.

Token Service

We will create a service to manage user and token information in a central place. Generate a TokenService in folder `authentication` (ng generate service authentication/token). Edit `src/app/authentication/token.service.ts` as follow.

```
1. import { Injectable } from '@angular/core';
2.
```

```
3.  export const TOKEN = 'token';
4.  export const USER_NAME = 'username';
5.  export const USER_ROLE = 'role';
6.
7.  @Injectable({
8.    providedIn: 'root'
9.  })
10. export class TokenService {
11.
12.   constructor() { }
13.
14.   signOut(): void {
15.     window.sessionStorage.clear();
16.   }
17.
18.   public saveToken(token: string): void {
19.     window.sessionStorage.removeItem(TOKEN);
20.     window.sessionStorage.setItem(TOKEN, token);
21.   }
22.
23.   public getToken(): string {
24.     return <string>sessionStorage.getItem(TOKEN);
25.   }
26.
27.   public saveUserName(username: string): void {
28.     window.sessionStorage.removeItem(USER_NAME);
29.     window.sessionStorage.setItem(USER_NAME, username);
30.   }
31.
32.   public saveUserRole(role: string): void {
33.     window.sessionStorage.removeItem(USER_ROLE);
34.     window.sessionStorage.setItem(USER_ROLE, role);
35.   }
36.
37.   public getUser(): string {
38.     return <string>sessionStorage.getItem(USER_NAME);
39.   }
40.
41.   public getRole(): string {
42.     return <string>sessionStorage.getItem(USER_ROLE);
43.   }
44. }
```

The service provides function to store and retrieve the authentication information to/from the Session Storage.

Authentication Service

Refactor move `logger-in-guard.ts`, `logger-in-guard.spec.ts`, `authentication.service.ts` and `authentication.service.spec.ts` to folder `src/app/authentication`. This just for better organization of the code.

Edit class AuthenticationService as follow.

```
import { Injectable } from '@angular/core';
import {Router} from '@angular/router';
import {Observable} from 'rxjs';
import {HttpClient} from '@angular/common/http';
import {TokenService} from './token.service';

const Url = 'http://localhost:8080/auth/';

@Injectable({
  providedIn: 'root'
})
export class AuthenticationService {
  constructor(private router: Router,
              private http: HttpClient,
              private tokenService: TokenService) {}

  login(username: string, password: string): Observable<any> {
    return this.http.post(Url + 'signin', {
      username,
      password
    });
  }

  logout(): void {
    this.tokenService.signOut();
  }

  getUser(): string {
    return this.tokenService.getUser();
  }

  isLoggedIn(): boolean {
    return this.getUser() !== null;
  }

  isAdmin(): boolean {
    return this.tokenService.getRole() === 'ROLE_ADMIN';
  }

  register(username: string, password: string): Observable<any> {
    return this.http.post(Url + 'signup', {
```

```
      username,
      password
    });
  }
}
```

The service interacts with the server for sign-in and (function login) and sign-up (function register).

HTTP Interceptor

We use an interceptor to add the JWT to requests for the books-api. An interceptor is used to inspect and transform HTTP requests from an application to a server or responses from the server to the client.

Create a service AuthenticationInterceptor (ng generate service authentication/authentication-interceptor). Edit class AuthenticationInterceptorService as follow.

```
1.  import { Injectable } from '@angular/core';
2.  import {HTTP_INTERCEPTORS, HttpEvent, HttpHandler, HttpInterceptor, HttpRequest} from
    '@angular/common/http';
3.  import {TokenService} from './token.service';
4.  import {Observable} from 'rxjs';
5.
6.  @Injectable()
7.  export class AuthenticationInterceptorService implements HttpInterceptor {
8.
9.    constructor(private tokenService: TokenService) { }
10.
11.   intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
12.     let authReq = req;
13.     const token = this.tokenService.getToken();
14.     if (token != null) {
15.       authReq = req.clone({ headers: req.headers.set('Authorization', 'Bearer ' + token) });
16.     }
17.     return next.handle(authReq);
18.   }
19. }
20.
21. export const authInterceptorProviders = [
22.   { provide: HTTP_INTERCEPTORS, useClass: AuthenticationInterceptorService, multi: true }
23. ];
```

The service's intercept function intercepts all outgoing HTTP requests, retrieves the token using the token service, and if it is not null, adds it as a Bearer authorization header to the request (lines 13 to 16). The interceptor then transmits the request to the next object (line 17).

An interceptor should be provided in the same injector or a parent of the injector that provides the HttpClient module. Line 21-23 associates the AuthenticationInterceptorService to the HTTP_INTERCEPTORS injection token. We must now edit the App Module and add authInterceptorProviders to the providers array. Modify `src/app/app.module.ts` as follow.

```
import {authInterceptorProviders} from './authentication/authentication-interceptor.service';

@NgModule({
  ...
  providers: [authInterceptorProviders],
  ...
})
export class AppModule { }
```

SignUp Component

Create a component for sign-up (ng generate component signup). Edit the component class `src/app/signup/signup.component.ts` as follow.

```
import { Component, OnInit } from '@angular/core';
import {AbstractControl, FormBuilder, ValidationErrors, Validators} from "@angular/forms";
import {AuthenticationService} from "../authentication/authentication.service";

function passwordMatcher(pwGrp: AbstractControl): ValidationErrors | null {
  const passwd = pwGrp.get('password');
  const confpasswd = pwGrp.get('confirmPassword');
  return passwd!.value === confpasswd!.value ? null : {mismatch: true};
}

@Component({
  selector: 'app-signup',
  templateUrl: './signup.component.html',
  styleUrls: ['./signup.component.css']
})
export class SignupComponent implements OnInit {
  message: string = '';
  signupForm = this.builder.group({
    username: ['', Validators.required],
    pwGroup: this.builder.group({
      password: ['', Validators.required],
      confirmPassword: ['', Validators.required]
    }, { validators: [passwordMatcher] })
  });

  get username(): AbstractControl {return <AbstractControl>this.signupForm.get('username'); }
  get password(): AbstractControl {return
<AbstractControl>this.signupForm.get('pwGroup')!.get('password'); }
  get confirmPassword(): AbstractControl {return
<AbstractControl>this.signupForm.get('pwGroup')!.get('confirmPassword'); }
  get pwGroup(): AbstractControl {return <AbstractControl>this.signupForm.get('pwGroup'); }

  constructor(private builder: FormBuilder,
              private authService: AuthenticationService) { }

  ngOnInit(): void {
```

```
35.   }
36.
37.   register(): void {
38.     this.authService.register(this.username.value, this.password.value).subscribe(
39.       data => {
40.         this.message = data.message;
41.         setTimeout(() => {
42.           this.message = '';
43.           this.signupForm.reset();
44.         }, 3000);
45.       },
46.       error => {
47.         this.message = 'Registration ' + error.error.message;
48.         setTimeout(() => {
49.           this.message = '';
50.           this.signupForm.reset();
51.         }, 3000);
52.       }
53.     );
54.   }
55. }
```

The component sets up a reactive form for the registration information (lines 18-24). We use an embedded form group to capture a password and a confirmation of that password. A Validator attached to the form group checks that both match (lines 5-9).

Function register is the form submission manager. It uses the authentication service to send a connection request and retrieve the response message (lines 37 to 54).

Edit the HTML Template (`src/app/signup/signup.component.html`) as follow.

```
1.  <div class="container">
2.    <div class="alert alert-danger" role="alert" *ngIf="message">
3.      {{ message }}
4.    </div>
5.    <form [formGroup]="signupForm" (ngSubmit)="register()">
6.      <div class="form-group">
7.        <label for="username">Username:</label>
8.        <input type="text" class="form-control" id="username"  formControlName="username">
9.        <div [hidden]="username.pristine || username.valid"
10.           class="alert alert-danger">
11.         Username is required.
12.       </div>
13.     </div>
14.     <div formGroupName="pwGroup">
15.       <div class="form-group">
16.         <label for="password">Password:</label>
17.         <input type="password" class="form-control" id="password" required
18.                                formControlName="password">
19.         <div [hidden]="password.pristine || password.valid"
```

```
            class="alert alert-danger">
          A Password is required.
        </div>
      </div>
      <div class="form-group">
        <label for="pwconfirm">Confirm Password:</label>
        <input type="password" class="form-control" id="pwconfirm" required
                                formControlName="confirmPassword">
        <div [hidden]="confirmPassword.pristine || confirmPassword.valid"
            class="alert alert-danger">
          A Confirmation Password is required.
        </div>
      </div>
      <div [hidden]="(password.pristine || confirmPassword.pristine) || pwGroup.valid"
          class="alert alert-danger">
        The passwords do not match.
      </div>
    </div>
    <button type="submit" class="btn btn-success" [disabled]="signupForm.invalid">Register</button>
  </form>
</div>
```

Login Component

Update the Login Component Class (`src/app/login/login.component.ts`) as follow.

```
import { Component, OnInit } from '@angular/core';
import {AuthenticationService} from '../authentication/authentication.service';
import {Router} from '@angular/router';
import {TokenService} from '../authentication/token.service';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
export class LoginComponent {
  username = '';
  password = '';
  message!: string;
  loggedIn = false;

  constructor(private router: Router,
              private loginService: AuthenticationService,
              private tokenService: TokenService) { }

  get isLoggedIn(): boolean {
    return this.loginService.isLoggedIn();
  }

```

```
  get loggedUser(): string {
    return this.loginService.getUser();
  }

  checkLogin(): void {
    this.message = '';
    this.loginService.login(this.username, this.password).subscribe(
      data => {
        this.tokenService.saveToken(data.token);
        this.tokenService.saveUserName(data.username);
        this.tokenService.saveUserRole(data.role);
        this.loggedIn = true;
      },
      err => {
        this.loggedIn = false;
        this.message = 'Invalid Login ' + err.error.message;
        setTimeout(() => {
          this.message = '';
        }, 3000);
      }
    );
  }

  logout(): boolean {
    this.loginService.logout();
    return true;
  }
}
```

Logged-In Guard

Modify the LoggedIn Guard (`src/app/authentication/logged-in.guard.ts`) as follow.

```
import { Injectable } from '@angular/core';
import {CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, UrlTree, Router} from
'@angular/router';
import { Observable } from 'rxjs';
import {AuthenticationService} from './authentication.service';

@Injectable({
  providedIn: 'root'
})
export class LoggedInGuard implements CanActivate {
  constructor(private authService: AuthenticationService, private router: Router) {}

  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot):
      Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
```

```
    return this.authService.isLoggedIn();
  }
}
```

Admin Guard

We are adding an HTTP Route Guard to ensure that only users authenticated with the ADMIN role can navigate route *admin*.

Create a guard (ng generate guard authentication/admin) with Can Activate and edit `src/app/authentication/admin.guard.ts` as follow.

```
import { Injectable } from '@angular/core';
import {CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, UrlTree} from '@angular/router';
import { Observable } from 'rxjs';
import {AuthenticationService} from './authentication.service';

@Injectable({
  providedIn: 'root'
})
export class AdminGuard implements CanActivate {
  constructor(private authService: AuthenticationService) {}
  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot):
            Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
    return this.authService.isAdmin();
  }
}
```

Router Module

Update the AppRouting Module (`src/app/app-routing.module.ts`) as follow.

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import {HomeComponent} from './home/home.component';
import {AboutComponent} from './about/about.component';
import {ContactComponent} from './contact/contact.component';
import {BooksComponent} from './books/books.component';
import {BookComponent} from './books/book/book.component';
import {LoginComponent} from './login/login.component';
import {AdminComponent} from './admin/admin.component';
import {LoggedInGuard} from './authentication/logged-in.guard';
import {AdminGuard} from './authentication/admin.guard';
import {SignupComponent} from './signup/signup.component';

const booksRoutes: Routes = [
  {path: ':id', component: BookComponent}
];

```

```
18. const routes: Routes = [
19.   {path: 'home', component: HomeComponent},
20.   {path: 'about', component: AboutComponent},
21.   {path: 'contact', component: ContactComponent},
22.   { path: 'login', component: LoginComponent },
23.   {
24.     path: 'admin',
25.     component: AdminComponent,
26.     canActivate: [ AdminGuard ]
27.   },
28.   {path: 'books', component: BooksComponent,
29.     canActivate: [ LoggedInGuard ],
30.     children: booksRoutes
31.   },
32.   { path: 'signup', component: SignupComponent },
33.   {path: '', redirectTo: 'home', pathMatch: 'full'},
34.   {path: '**', component: HomeComponent}
35. ];
36.
37.
38. @NgModule({
39.   imports: [RouterModule.forRoot(routes)],
40.   exports: [RouterModule]
41. })
42. export class AppRoutingModule { }
```

We have updated the *admin* and *books* route guards and added a *signup* route for the Signup component.

App Component

Update the App Component Class (`src/app/app.component.ts`) as follow.

```
1.  import { Component } from '@angular/core';
2.  import {AuthenticationService} from './authentication/authentication.service';
3.
4.  @Component({
5.    selector: 'app-root',
6.    templateUrl: './app.component.html',
7.    styleUrls: ['./app.component.css']
8.  })
9.  export class AppComponent {
10.   title = 'book-store';
11.   get login_label(): string {
12.     return this.authService.isLoggedIn() ? 'Logout' : 'Login';
13.   }
14.   constructor(private authService: AuthenticationService) {}
15. }
```

We added function getLogin (lines 10-12) to return a string value depending of the user's login status. The string will serve as label for a route link in the template.

Update the App Component HTML Template (`src/app/app.component.html`) as follow.

```
<div class="container">
  <div class="nav-link">
    <a [routerLink]="['/home']"> Home </a>
    <a [routerLink]="['/about']"> About Us </a>
    <a [routerLink]="['/contact']"> Contact Us </a>
    <a [routerLink]="['/books']"> Books </a>
    <a [routerLink]="['/login']"> {{login_label}} </a>
    <a [routerLink]="['/admin']"> Admin </a>
    <a [routerLink]="['/signup']"> Signup </a>
  </div>
</div>

<div class="container-fluid" >
  <div class="row">
    <div class="col">
      <img src="../assets/gobooks.jpg" height="150" width="175"/>
    </div>
    <div class="col-6">
      <router-outlet></router-outlet>
    </div>
    <div class="col">
      <img src="../assets/1books2-med.jpg" height="150" width="175"/>
    </div>
  </div>
</div>
```

3. Exercise

The following exercise is the deliverable for the lab. Complete and check-in the code to Github Classroom before the deadline. Only this exercise will be evaluated.

Add In Memory authentication/authorization to your REST Calculator Web Service API developed for Lab 6. Configure at least two users: *user1* with password *pass1* and *user2* with password *pass2* for the application.