# Auftrag Registration, Login & JWT

|  |  |  |  |
| --- | --- | --- | --- |
| Richtzeit: 90‘ |  | Sozialform: Einzelarbeit |  |

## Einleitung

In diesem Auftrag wirst du sehr viel spezifischen und auch teils komplexen Code sehen. Du musst den Code in diesem Auftrag nicht explizit verstehen für den Ük. Jedoch ist das Grundprinzip von Authentication und Authorization sowie JWT wichtig.

## Package erstellen und Dependencies hinzufügen

Erstelle im Controller-Package (MVC) ein neues Package mit dem Namen «security». Unsere gesamte Authentication und Authorization Logik wird sich in diesem Package befinden.

Im build.gradle brauchen wir auch eine neue Dependency um JWT umzusetzen:

implementation 'com.auth0:java-jwt:4.2.2'

und im application.properties werden wir neu auch eine weitere Zeile Code benötigen um spezifische URLs entsperren und sperren zu können (wäre ja doof, wenn man für das Login bereits eingeloggt sein muss).

spring.mvc.pathmatch.matching-strategy = ANT\_PATH\_MATCHER

## Step-1 Url’s unzugänglich machen

Bis jetzt haben wir für unsere Testzwecke sämtliche URL’s entsperrt, damit wir uns nicht jedes Mal registrieren und einloggen müssen um einen Request zu testen. Dies wollen wir jetzt nicht mehr haben und wollen ein effektives Login einbauen. Dazu erstellen wir in unserem Package security eine neue Klasse namens «SecurityConstants.java».

Hier definieren wir Konstante, welche sich über die gesamte Applikation nicht verändern dürfen.

Für den Moment definieren wir welche URL’s wir dem User erlauben möchten ohne Login zu verwenden (zum Beispiel die sign-up URL):

public class SecurityConstants {

    public static final String SIGN\_UP\_URL = "/users/sign-up";

}

Diese URL können wir nun in unserem File SecurityConfiguration.java genauer beschreiben:

public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override

    protected void configure(HttpSecurity http) throws Exception {

        http.cors().and().csrf().disable().authorizeRequests()

                .antMatchers(HttpMethod.POST, SIGN\_UP\_URL).permitAll()

                .anyRequest().authenticated();

    }

}

Wir sehen nun, dass wir die http-Methode POST für die erwähnte URL zulassen. Sämtliche anderen URL’s müssen zwingend authentifizert werden.

Wir können dies im Postman testen, indem wir versuchen alle Genres auszugeben. Wir werden feststellen, dass wir nun den Code 403-Forbidden zurückerhalten. Also hat das funktioniert 😊

## Step-2 Passwörter schützen

Wenn du bereits ausprobiert hast, einen Benutzer zu erstellen, hast du wahrscheinlich gemerkt, dass die Passwörter unverschlüsselt in die Datenbank gelangen. Das ist ein gigantisches No-Go und müssen wir nun sofort ändern. In der Deep-Dive Präsentation werden wir im Detail darauf eingehen, wie wir das machen und warum wir BCrypt verwenden aber für diesen Auftrag, kannst du bereits ohne dieses Wissen die Logik umsetzen.

Zunächst müssen wir den Verschlüsselungsalgorithmus BCryptPasswordEncoder für die gesamte Applikation freischalten. Dazu werden wir im Package configurations eine neue Klasse namens ApplicationConfiguration.java anlegen:

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@Configuration

public class ApplicationConfiguration {

    @Bean

    public BCryptPasswordEncoder bCryptPasswordEncoder() {

        return new BCryptPasswordEncoder();

    }

}

So weit so gut, aber wir müssen nun BCrypt an den richtigen Orten verwenden. Spezifischer ausgedrückt im UserService.java.

Wir sehen, dass wir bisher in der update und signUp Methode den User einfach entgegennehmen und ohne Businesslogik an das Repository weiterleiten und in die Datenbank schreiben. Dies ändern wir nun.

Dazu instanziieren wir den BCryptPasswordEncoder und fügen ihn in den Konstruktor des UserServices hinzu. Danach können wir folgende Anpassungen treffen:

public void signUp(User user) {

        user.setPassword(bCryptPasswordEncoder.encode(user.getPassword()));

        userRepository.save(user);

    }

    public void update(User user) {

        user.setPassword(bCryptPasswordEncoder.encode(user.getPassword()));

        userRepository.save(user);

    }

Von nun an nehmen wir das User Objekt entgegen und bevor wir dieses in die Datenbank speichern, verschlüsseln wir das Passwort mit BCrypt.

Wir können dies auch bereits testen, indem wir einen Postman Request auf die URL /users/sign-up mit POST und einem validen json schicken.

## JWT

Gut, wir haben nun unsere Seiten gegen unbefugte Zugriffe geschützt und unsere Passwörter in der Datenbank sind nun verschlüsselt. Jetzt geht es darum JWT umzusetzen.

Wir benötigen nun dafür 2 neue Klassen, welche wir im security Package einfügen.

* JWTAuthenticationFilter.java
* JWTAuthorizationFilter.java

Wir behandeln nun Code, welcher nicht selbstverständlich ist. Daher zeigen wir den Code in diesem Auftrag und erklären die einzelnen Methoden.

**Anpassungen in den SecurityConstants.java**

public class SecurityConstants {

    public static final String SIGN\_UP\_URL = "/users/sign-up";

    public static final String SECRET = "SecretKeyToGenJWTs";

    public static final long EXPIRATION\_TIME = 864\_000\_000; // 10 days

    public static final String TOKEN\_PREFIX = "Bearer ";

    public static final String HEADER\_STRING = "Authorization";

}

Wir deklarieren nun verschiedene Konstants für JWT. Hier finden wir beispielsweise das Secret des JWT, die Dauer der Gültigkeit unserer JWT und die Headers, die gesendet werden.

**JWTAuthenticationFilter.java**

public class JWTAuthenticationFilter extends UsernamePasswordAuthenticationFilter {

    private AuthenticationManager authenticationManager;

    public JWTAuthenticationFilter(AuthenticationManager authenticationManager) {

        this.authenticationManager = authenticationManager;

    }

    @Override

    public Authentication attemptAuthentication(HttpServletRequest req,

            HttpServletResponse res) throws AuthenticationException {

        try {

            User creds = new ObjectMapper()

                    .readValue(req.getInputStream(), User.class);

            return authenticationManager.authenticate(

                    new UsernamePasswordAuthenticationToken(

                            creds.getUsername(),

                            creds.getPassword(),

                            new ArrayList<>()));

        } catch (IOException e) {

            throw new RuntimeException(e);

        }

    }

    @Override

    protected void successfulAuthentication(HttpServletRequest req,

            HttpServletResponse res,

            FilterChain chain,

            Authentication auth) {

        String token = JWT.create()

                .withSubject(((org.springframework.security.core.userdetails.User) auth.getPrincipal()).getUsername())

                .withExpiresAt(new Date(System.currentTimeMillis() + EXPIRATION\_TIME))

                .sign(HMAC512(SECRET.getBytes()));

        res.addHeader(HEADER\_STRING, TOKEN\_PREFIX + token);

    }

}

Wir erben mit dieser Klasse von der Standard Spring Security Klasse. Mithilfe dieser können wir Benutzer authentisieren. Mit der attemptAuthentication Funktion entnehmen wir aus dem Request den Usernamen sowie das Passwort und versuchen uns anzumelden. Falls dies nicht funktioniert, werfen wir eine RuntimeException.

Die successfulAuthentication Methode wird nach Erfolg der attemptAuthentication Methode ausgeführt. Diese erstellt ein neues JWT. Wir sehen auch, dass wir die Variablen aus den SecurityConstants verwenden, um unser JWT zu erstellen.

**JWTAuthorizationFilter.java**

public class JWTAuthorizationFilter extends BasicAuthenticationFilter {

    public JWTAuthorizationFilter(AuthenticationManager authManager) {

        super(authManager);

    }

    @Override

    protected void doFilterInternal(HttpServletRequest req,

            HttpServletResponse res,

            FilterChain chain) throws IOException, ServletException {

        String header = req.getHeader(HEADER\_STRING);

        if (header == null || !header.startsWith(TOKEN\_PREFIX)) {

            chain.doFilter(req, res);

            return;

        }

        UsernamePasswordAuthenticationToken authentication = getAuthentication(req);

        SecurityContextHolder.getContext().setAuthentication(authentication);

        chain.doFilter(req, res);

    }

    private UsernamePasswordAuthenticationToken getAuthentication(HttpServletRequest request) {

        String token = request.getHeader(HEADER\_STRING);

        if (token != null) {

            // parse the token.

            String user = JWT.require(Algorithm.HMAC512(SECRET.getBytes()))

                    .build()

                    .verify(token.replace(TOKEN\_PREFIX, ""))

                    .getSubject();

            if (user != null) {

                return new UsernamePasswordAuthenticationToken(user, null, new ArrayList<>());

            }

            return null;

        }

        return null;

    }

}

Den Unterschied zwischen Authentication und Authorization wurde ja bereits während der Präsentation erklärt. Das Wichtigste hier ist die getAuthentication Methode. Diese Methode stellt dem User ein Token aus, falls dieser noch keines besitzt.

Okay! Kurze Verschnaufpause! Du hast nun die komplette Logik für JWT im Backend umgesetzt. Wir sind aber noch nicht fertig. Damit nun alles miteinander harmonisiert, müssen wir natürlich dem Backend bzw. der Konfiguration auch mitteilen, dass es von nun an unsere JWTFilter-Klassen benutzen soll. Dazu gehen wir zurück in die SecurityConfiguration.java Klasse.

Hier ergänzen wir das File wie folgt:

@Configuration

@EnableWebSecurity

public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    private UserDetailsServiceImpl userDetailsService;

    private BCryptPasswordEncoder bCryptPasswordEncoder;

    public SecurityConfiguration(UserDetailsServiceImpl userDetailsService,

            BCryptPasswordEncoder bCryptPasswordEncoder) {

        this.userDetailsService = userDetailsService;

        this.bCryptPasswordEncoder = bCryptPasswordEncoder;

    }

    @Override

    protected void configure(HttpSecurity http) throws Exception {

        http.cors().and().csrf().disable().authorizeRequests()

                .antMatchers(HttpMethod.POST, SIGN\_UP\_URL).permitAll()

                .anyRequest().authenticated()

                .and()

                .addFilter(new JWTAuthenticationFilter(authenticationManager()))

                .addFilter(new JWTAuthorizationFilter(authenticationManager()))

                // this disables session creation on Spring Security

                .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);

    }

    @Override

    public void configure(AuthenticationManagerBuilder auth) throws Exception {

        auth.userDetailsService(userDetailsService).passwordEncoder(bCryptPasswordEncoder);

    }

    @Bean

    CorsConfigurationSource corsConfigurationSource() {

        final UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();

        source.registerCorsConfiguration("/\*\*", new CorsConfiguration().applyPermitDefaultValues());

        return source;

    }

}

In der configure Methode haben wir nun Spring Security mitgeteilt, dass wir unsere JWTFilter für Authentication und Authorization im ganzen Projekt umsetzen möchten. Zusätzlich benötigen wir nun unsere «spezielle» Methode um den User zu callen. Die Methode ganz unten «corsConfigurationSource» ist nicht zwingend notwendig, aber gut, wenn wir sie bereits im File haben. Wir schalten somit frei, dass unser allfälliges Frontend später keine Probleme beim Verbinden bekommt.

Damit die «spezielle» Methode für den User-Call funktioniert, müssen wir nun noch zusätzlich die Klasse UserDetailsImpl.java im Service Package erstellen:

@Service

public class UserDetailsServiceImpl implements UserDetailsService {

    private final UserRepository userRepository;

    public UserDetailsServiceImpl(UserRepository userRepository) {

        this.userRepository = userRepository;

    }

    @Override

    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        User user = userRepository.findByUsername(username);

        if (user == null) {

            throw new UsernameNotFoundException(username);

        }

        return new org.springframework.security.core.userdetails.User(user.getUsername(), user.getPassword(),

                emptyList());

    }

}

Weil Spring Boot bzw. Spring Security nur ihren User akzeptiert für Authentication und Authorization machen wir mit dieser Methode aus unserem User Objekt ein Spring Security User Objekt.

Und zu guter letzt, erstellen wir noch die benötigte Methode im UserRepository:

public interface UserRepository extends CrudRepository<User, Integer> {

    User findByUsername(@Param("username") String username);

}

**UND TA-DA** wir haben es geschafft. Keine Sorge, dies war zum Teil sehr High-Level Code und wenn du nicht 100% drauskommst, ist das normal. Weiter unten gibt es für die wichtigsten Punkte, welche du verstehen solltest, noch ein kleines Verständnisquiz.

Teste aber nun noch ob dein Backend funktioniert! Du solltest dich mit dem /sign-up registrieren können und mit /login anmelden. Verifiziere, ob du in den Headers bei der Antwort den Header «Authorization» siehst und ein JWT-Token sich darin befindet.

Falls du einen Fehler bekommst, «doFilterInternal….» dann musst du mit Gradle dein Projekt kurz neu builden.

## Verständnisquiz

Die Struktur eines JWT besteht aus 3 spezifischen Elementen. Diese sind (**Reihenfolge wichtig!**):

|  |  |  |
| --- | --- | --- |
|  |  |  |

Welche Informationen befinden sich in den jeweiligen Elementen eines JWT? Nenne mind. 2 Stichpunkte pro Element:

|  |  |  |
| --- | --- | --- |
|  |  |  |

Der Name JWT bedeutet ausgeschrieben:

Wie wird anhand eines JWT überprüft ob der User berechtigt ist, um eine Aktion auszuführen?

Wann wird dem User ein JWT ausgestellt? Begründe deine Antwort möglichst spezifisch:

Wer / Was kümmert sich bei Spring Boot um die Authentifizierung?

«Wenn ich meinem JWT keine Lebenszeit bzw. unendlich lange Lebenszeit gebe, muss ich dieses nie erneuern. Dadurch passieren weniger Fehler». Bewerte diese Aussage, würdest du dieser zustimmen?

## Kontrolle

Wenn du diesen Auftrag richtig erledigt hast, solltest du nun die folgenden Methoden via Postman auf deiner Applikation ausführen können:

POST <http://localhost:8080/users/sign-up> Benötigt ein JSON mit username und password

POST <http://localhost:8080/login> Benötigt ein JSON mit username und password

GET <http://localhost:8080/genres> Benötigt das JWT Token vom Login im Header

GET <http://localhost:8080/movies> Benötigt das JWT Token vom Login im Header