

# How to Implement JWT Authentication & Authorization using SQL



sahu-himanshu



# What is 'JWT'? {

JWT stands for JSON Web Token. It is an open standard (RFC 7519) used for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed.

}

# Steps to 'Implement' {

01 Write the Entity classes  
for User and Role

02 Setup SQL & Create Role/User Repo

03 Create Controller and Services  
for User and Roles

04 Create Security Configuration Files

}

# Steps to 'Implement' {

05 Implement UserDetailsService

06 Implement AuthenticationEntryPoint and  
AuthenticationFilter

07 Create JWT Classes: JWTResponse,  
JWTRequest, JWTHelper

}

```
1  User 'Class';
2
3      @Getter
4      @Setter
5      @AllArgsConstructor
6      @Entity
7      @Table(name = "users")
8      public class User implements UserDetails {
9          @Id
10         @Column(name = "id")
11         private String userId;
12         private String name;
13         private String email;
14         private String password;
15         @ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
16         @JoinTable(name = "employee_roles",
17             joinColumns = @JoinColumn(name = "employee_id", referencedColumnName = "id"),
18             inverseJoinColumns = @JoinColumn(name = "role_id", referencedColumnName =
19 "id")
20         )
21         @JsonManagedReference
22         private List<Role> roles;
23     }
```

```
1  Role 'Class';
```

```
2  
3  
4      @Getter  
5      @Setter  
6      @AllArgsConstructor  
7      @Entity  
8      @Table(name = "role")  
9      public class Role {  
10         @Id  
11         @Column(name = "id")  
12         private String roleId;  
13  
14         private String name;  
15  
16         @ManyToMany(mappedBy = "roles")  
17         @JsonBackReference  
18         private List<User> users;  
19     }
```

# User & RoleRepository 'Class';

# UserRepository.java

@Repository

```
public interface UserRepository extends JpaRepository<User, String> {  
    Optional<User> findByEmail(String email);  
}
```

# RoleRepository.java

@Repository

```
public interface RoleRepository extends JpaRepository<Role, String> {  
    Optional<Role> findById(String id);  
}
```

# AdminController 'Class';

```
1
2
3
4
5     @RestController
6     @RequestMapping("/admin")
7     public class AdminController {
8
9         @Autowired
10        private UserService userService;
11
12        @PostMapping("/create")
13        public String user(@RequestBody User user) {
14            return userService.createUser(user);
15        }
16    }
```



# AuthController 'Class';

```
1  @RestController
2  @RequestMapping("/auth")
3  public class AuthController {
4
5      @Autowired
6      private UserDetailsService userDetailsService;
7      @Autowired
8      private AuthenticationManager manager;
9      @Autowired
10     private JwtHelper helper;
11
12     private Logger logger =
13     LoggerFactory.getLogger(AuthController.class);
14
15     @PostMapping("/login")
16     public ResponseEntity<JwtResponse> login(@RequestBody JwtRequest
17     request) {
18
19         this.doAuthenticate(request.getEmail(),
20         request.getPassword());
21         UserDetails userDetails =
22         userDetailsService.loadUserByUsername(request.getEmail());
23         String token = this.helper.generateToken(userDetails);
24         JwtResponse response = JwtResponse.builder()
25             .jwtToken(token)
26             .username(userDetails.getUsername())
27             .build();
28         return new ResponseEntity<>(response, HttpStatus.OK);
29     }
30 }
```

```
private void doAuthenticate(String email, String password) {
    UsernamePasswordAuthenticationToken authentication = new
    UsernamePasswordAuthenticationToken(email, password);
    try {
        manager.authenticate(authentication);
    } catch (BadCredentialsException e) {
        throw new BadCredentialsException(" Invalid Username or
        Password !!");
    }
}

ExceptionHandler(BadCredentialsException.class)
public String exceptionHandler() {
    return "Credentials Invalid !!";
}
```

# UserController 'Class';

```
1
2
3     @RestController
4     @RequestMapping("/user")
5     public class UserController {
6
7         @Autowired
8         private UserService userService;
9
10        @RequestMapping("/getevents")
11        private String getEvents() {
12            return "Events";
13        }
14
15        @GetMapping("/all")
16        public List<User> user() {
17            return this.userService.getAllUsers();
18        }
19    }
```

1 **UserService** 'Class';

```
2
3
4 @Service
5 public class UserService {
6     @Autowired
7     private UserRepository userRepository;
8     @Autowired
9     private RoleRepository roleRepository;
10    @Autowired
11    private PasswordEncoder passwordEncoder;
12    public List<User> getAllUsers() {
13        return userRepository.findAll();
14    }
15}
```

```
public String createUser(User user) {
    user.setUserId(UUID.randomUUID().toString());

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

    List<Role> roles = user.getRoles();
    for (Role role: roles) {
        Role r = new Role();

        r.setRoleId(UUID.randomUUID().toString());
        r.setName(role.getName());
        roleRepository.save(r);
    }
    userRepository.save(user);
    return user.getUserId();
}
```

1 **AppConfig** 'Class';  
2  
3  
4  
5

```
6 @Configuration  
7 public class AppConfig {  
8     @Bean  
9     public PasswordEncoder passwordEncoder() {  
10         return new BCryptPasswordEncoder();  
11     }  
12 }  
13  
14
```

# SecurityConfig 'Class';

```

1  @Configuration
2  public class SecurityConfig {
3
4      @Autowired
5      private JWTAuthentcationEntryPoint point;
6
7      @Autowired
8      private JwtAuthenticationFilter filter;
9
10     @Autowired
11     private CustomUserDetailsService userDetailsService;
12
13     @Autowired
14     private PasswordEncoder passwordEncoder;
15
16     @Bean
17     public SecurityFilterChain securityFilterChain(HttpSecurity
18     http) throws Exception {
19         http.csrf(csrf → csrf.disable())
20             .authorizeHttpRequests(auth → auth
21                 .requestMatchers("/auth/login").permitAll()
22                 .requestMatchers("/admin/create").permitAll()
23                 .requestMatchers("/user/all").hasAuthority("USER")
24                     .anyRequest().hasAnyAuthority("ADMIN",
25                         "USER"))
26             .exceptionHandling(ex →
27                 ex.authenticationEntryPoint(point))
28             .sessionManagement(session →
29                 session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
30         http.addFilterBefore(filter,
31             UsernamePasswordAuthenticationFilter.class);
32         return http.build();
33     }

```

```

34     @Bean
35     public DaoAuthenticationProvider daoAuthenticationProvider() {
36         DaoAuthenticationProvider daoAuthenticationProvider = new
37         DaoAuthenticationProvider();
38
39         daoAuthenticationProvider.setUserDetailsService(userDetailsService);
40
41         daoAuthenticationProvider.setPasswordEncoder(passwordEncoder);
42         return daoAuthenticationProvider;
43     }
44
45     @Bean
46     public AuthenticationManager
47     authenticationManager(AuthenticationConfiguration configuration)
48     throws Exception {
49         return configuration.getAuthenticationManager();
50     }

```

# UserDetailsService 'Class';

```
1
2
3
4     @Service
5     public class CustomUserDetailsService implements UserDetailsService {
6
7         @Autowired
8         private UserRepository userRepository;
9
10        @Override
11        public UserDetails loadUserByUsername(String username) throws
12        UsernameNotFoundException {
13            User user = userRepository.findByEmail(username).orElseThrow(() -> new
14            RuntimeException("User Not found!"));
15            return user;
16        }
17    }
```

```
1  JWTAuthenticationEntryPoint 'Class';
```

```
2  
3  
4  
5  @Component  
6  public class JWTAuthenticationEntryPoint implements AuthenticationEntryPoint {  
7      @Override  
8      public void commence(HttpServletRequest request, HttpServletResponse response,  
9          AuthenticationException authException) throws IOException, ServletException {  
10         response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);  
11         PrintWriter printWriter = response.getWriter();  
12         printWriter.println("Access Denied!!! " + authException.getMessage());  
13     }  
14 }
```

# JWTAuthenticationFilter 'Class';

```

1  @Component
2  public class JwtAuthenticationFilter extends OncePerRequestFilter {
3
4      private Logger logger =
5      LoggerFactory.getLogger(OncePerRequestFilter.class);
6      @Autowired
7      private JwtHelper jwtHelper;
8      @Autowired
9      private UserDetailsService userDetailsService;
10     @Override
11     protected void doFilterInternal(HttpServletRequest request,
12     HttpServletResponse response, FilterChain filterChain) throws ServletException,
13     IOException {
14         String requestHeader = request.getHeader("Authorization");
15         logger.info(" Header : {}", requestHeader);
16         String username = null;
17         String token = null;
18         if (requestHeader != null && requestHeader.startsWith("Bearer")) {
19             token = requestHeader.substring(7);
20             try {
21                 username = this.jwtHelper.getUsernameFromToken(token);
22             } catch (IllegalArgumentException e) {
23                 logger.info("Illegal Argument while fetching the username !!");
24                 e.printStackTrace();
25             } catch (ExpiredJwtException e) {
26                 logger.info("Given jwt token is expired !!");
27                 e.printStackTrace();
28             } catch (MalformedJwtException e) {
29                 logger.info("Some changed has done in token !! Invalid Token");
30                 e.printStackTrace();
31             } catch (Exception e) {
32                 e.printStackTrace();
33             }
34         } else {
35             logger.info("Invalid Header Value !! ");
36         }
37     }
38 }

```

```

1  if (username != null && SecurityContextHolder.getContext().getAuthentication()
2  == null) {
3
4      UserDetails userDetails =
5      this.userDetailsService.loadUserByUsername(username);
6      Boolean validateToken = this.jwtHelper.validateToken(token,
7      userDetails);
8      if (validateToken) {
9
10         UsernamePasswordAuthenticationToken authentication = new
11         UsernamePasswordAuthenticationToken(userDetails, null,
12         userDetails.getAuthorities());
13         authentication.setDetails(new
14         WebAuthenticationDetailsSource().buildDetails(request));
15
16         SecurityContextHolder.getContext().setAuthentication(authentication);
17
18     } else {
19         logger.info("Validation fails !!");
20     }
21 }
22 filterChain.doFilter(request, response);
23 }

```



```
1  JWTResponse 'Class';
```

```
2  
3  
4  
5  
6  @Getter  
7  @Setter  
8  @AllArgsConstructor  
9  @Builder  
10 public class JwtResponse {  
11  
12     private String jwtToken;  
13     private String username;  
14 }
```

```
1  JWTRequest 'Class';
```

```
2  
3  
4  
5  @Getter  
6  @Setter  
7  @NoArgsConstructor  
8  @AllArgsConstructor  
9  @Builder  
10 @ToString  
11 public class JwtRequest {  
12     private String email;  
13     private String password;  
14 }
```

## JWTHelper 'Class';

```

1  @Component
2  public class JwtHelper {
3      public static final long JWT_TOKEN_VALIDITY = 5 * 60 * 60;
4      private String secret =
5      "afafasfafafasfasfasfafacasdasfasxASFACASDFACASDFASFASFDASFDASDAAD
6      SCSDFADCVS GCFVADXCcadwavfsfarvf";
7
8      public String getUsernameFromToken(String token) {
9          return getClaimFromToken(token, Claims::getSubject);
10     }
11     public Date getExpirationDateFromToken(String token) {
12         return getClaimFromToken(token, Claims::getExpiration);
13     }
14     public <T> T getClaimFromToken(String token, Function<Claims,
15     T> claimsResolver) {
16         final Claims claims = getAllClaimsFromToken(token);
17         return claimsResolver.apply(claims);
18     }
19     private Claims getAllClaimsFromToken(String token) {
20         return
21         Jwts.parser().setSigningKey(secret).parseClaimsJws(token).getBody()
22     ;
23     }
24     private Boolean isTokenExpired(String token) {
25         final Date expiration = getExpirationDateFromToken(token);
26         return expiration.before(new Date());
27     }

```

```

28     public String generateToken(UserDetails userDetails) {
29         Map<String, Object> claims = new HashMap<>();
30         return doGenerateToken(claims, userDetails.getUsername());
31     }
32     private String doGenerateToken(Map<String, Object> claims, String subject) {
33
34         return Jwts.builder().setClaims(claims).setSubject(subject).setIssuedAt(new
35         Date(System.currentTimeMillis()))
36         .setExpiration(new Date(System.currentTimeMillis() +
37         JWT_TOKEN_VALIDITY * 1000))
38         .signWith(SignatureAlgorithm.HS512, secret).compact();
39     }
40     public Boolean validateToken(String token, UserDetails userDetails) {
41         final String username = getUsernameFromToken(token);
42         return (username.equals(userDetails.getUsername()) &&
43         !isTokenExpired(token));
44     }
45 }

```

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14

```
Thanks {  
Guys;  
|  
}  

```