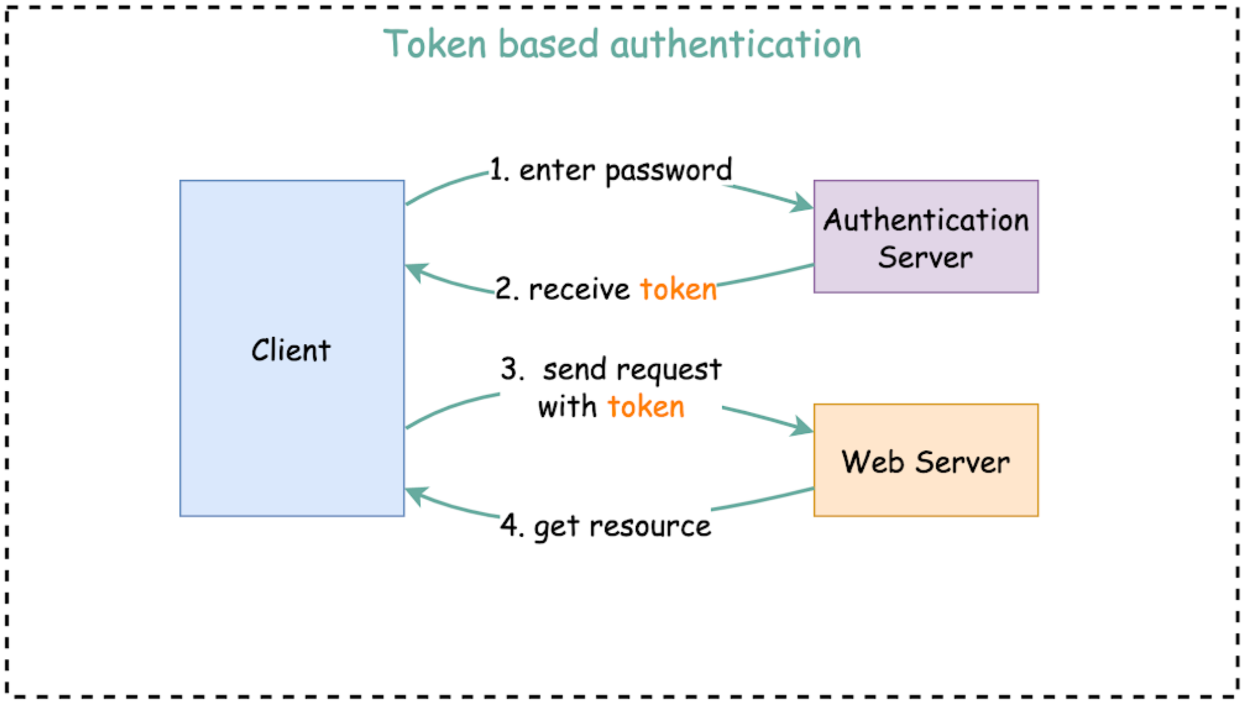
Implementing Token-Based Authentication with Spring Boot and Spring Security



Welcome to our guide on implementing token-based authentication with Spring Boot and Spring Security using JSON Web Tokens (JWT). In this tutorial, we'll walk you through the process of setting up a secure authentication system for your Spring Boot application, allowing users to register, log in, and access protected resources using JWT-based authentication.

1. Overview

This project serves as a starting point for understanding how to create a basic Spring Boot application that uses Spring Security as it's security protocol. It includes the necessary setup and dependencies to quickly get you up and running.

1.1. Key Components

In this guide, we'll cover the following key components:

- **Maven Dependencies**: We'll add the required dependencies to the `pom.xml` file, including Spring Boot starters for data JPA, security, and web, as well as PostgreSQL driver and JWT libraries.
- **Configuration**: We'll configure the `application.properties` to set up the database connection details and JWT secret key.
- **Entity Models**: We'll define the necessary entity models such as `AdminEntity`, `UserEntity`, and `UserType` enum to represent users and their roles.
- **Service Layer**: Our service layer will include services for user registration, authentication, and generating JWT tokens.
- **Spring Security Configuration**: We'll configure Spring Security to handle authentication and authorization using JWT tokens, including setting up authentication entry points, filters, and user details services.
- **REST Endpoints**: We'll define REST endpoints for user registration, login, and accessing protected resources, securing them with JWT-based authentication.
- **Testing**: Finally, we'll run the Spring Boot application and test the registration, login, and access to protected resources using cURL commands.

2. Prerequisites

Before we dive into the implementation, make sure you have the necessary prerequisites installed on your system:

- **Java Development Kit (JDK)**: Ensure you have JDK version 17 or higher installed.
- **Maven**: You'll need Maven as a build tool for managing dependencies and building the project.
- **PostgreSQL Database**: Make sure you have PostgreSQL installed and running, as we'll use it to store user data.

3. Maven Dependencies

> We'll start by adding the required dependencies to your `pom.xml` file. These dependencies include Spring Boot starters for data JPA, security, and web, as well as PostgreSQL driver and JWT libraries.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
```

```xml
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
  </dependency>

  <dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
  </dependency>
  <dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
  </dependency>
```

4. Configure Properties

> Next, we'll configure the `application.properties`, including the database connection details and JWT secret key.

```
spring.datasource.url=jdbc:postgresql://localhost:5432/springsecurity
spring.datasource.username=<DB_USERNAME>
spring.datasource.password=<DB_USER_PASSWORD>
spring.sql.init.mode=always
spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.open-in-view=false

server.port=8081

jwt.secret= authenticationsecretrandomstringwithmorethan256bits
jwt.expirationMs= 86400000
```

5. Models

> We'll define the necessary entity models for our application, including `AdminEntity`, `UserEntity`, and `UserType` enum.

5.1. AdminEntity

```
@Getter
@Setter
@NoArgsConstructor
@Entity
@Table(name = "admin")
public class AdminEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String username;

    @JsonIgnore
    private String password;
}
```

5.2. UserEntity

```
@Getter
@Setter
@NoArgsConstructor
@Entity
@Table(name = "users")
public class UserEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "user_id")
    private Long id;

    private String username;

    @JsonIgnore
    private String password;
}
```

5.3. UserType

```
public enum UserType {

    ADMIN("ADMIN"), USER("USER");

    private final String type;

    UserType(String string) {
        type = string;
    }

    @Override
    public String toString() {
        return type;
    }
}
```

6. Service Layer

> Our service layer will include services for user registration, authentication, and generating JWT tokens.

6.1. UserServiceImpl

```
@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private UserRepo userRepo;

    @Autowired
    private PasswordEncoder passwordEncoder;

    @Autowired
    private JwtGenerator jwtGenerator;

    @Override
    public UserEntity findById(Long id) {
        return userRepo.findById(id)
                .orElseThrow(() -> new NotFoundException("User not found"));
    }

    @Override
    public UserEntity findByUsername(String username) {
        return userRepo.findByUsername(username)
                .orElseThrow(() -> new NotFoundException("User not found"));
    }

    @Override
    public boolean existsByUsername(String username) {
        return userRepo.existsByUsername(username);
    }

    @Override
    public String regUser(UserAuthDto userAuthDto) {
        if (existsByUsername(userAuthDto.getUsername())) {
            throw new BadRequestException("Username is already registered
!!");
        }
        UserEntity userEntity = new UserEntity();
        userEntity.setUsername(userAuthDto.getUsername());
        userEntity.setPassword(passwordEncoder.encode(userAuthDto.getPassword
()));
        userRepo.save(userEntity);
        return "User Register successfull !!";
    }

    @Override
    public UserLoginResponseDto loginUser(UserAuthDto userAuthDto) {
        String token = jwtGenerator.generateToken(userAuthDto.getUsername(),
UserType.USER.toString());
        UserEntity user = findByUsername(userAuthDto.getUsername());

        UserLoginResponseDto responseDto = new UserLoginResponseDto();
        responseDto.setToken(token);
        responseDto.setUser(user);
        return responseDto;
    }
}
```

7. Configure Spring Security

> We'll configure Spring Security to handle authentication and authorization using JWT tokens. This includes setting up authentication entry points, authentication filters, and user details services.

7.1. SecurityConfig

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Autowired
    private JwtAuthEntryPoint jwtAuthEntryPoint;

    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Excepti
on {
        http
        .csrf(AbstractHttpConfigurer::disable)
        .exceptionHandling(exceptionHandling -> exceptionHandling
                .authenticationEntryPoint(jwtAuthEntryPoint)
                )
        .sessionManagement(sessionManagement -> sessionManagement
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                )
        .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/api/public/admin/**").hasAuthority(UserTyp
e.ADMIN.toString())
                .requestMatchers("/api/public/user/**").hasAuthority(UserTyp
e.USER.toString())
                .requestMatchers("/auth/**").permitAll()
                .anyRequest().authenticated()
                )
        .addFilterBefore(jwtAuthenticationFilter(), UsernamePasswordAuthentic
ationFilter.class);
        return http.build();
    }

    @Bean
    AuthenticationManager authenticationManager(AuthenticationConfiguration a
uthenticationConfiguration) throws Exception {
        return authenticationConfiguration.getAuthenticationManager();
    }

    @Bean
    PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    JwtAuthenticationFilter jwtAuthenticationFilter() {
        return new JwtAuthenticationFilter();
    }
}
```

7.2. CustomUserDetailsService

```
@Service
public class CustomUserDetailsService implements UserDetailsService {

    @Autowired
    private AdminService adminService;

    @Autowired
    private UserService userService;

    private UserType userType;

    public UserType getUserType() {
        return userType;
    }

    public void setUserType(UserType userType) {
        this.userType = userType;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNot
FoundException {
        if (userType == UserType.ADMIN) {
            AdminEntity adminEntity = adminService.findByUsername(username);
            SimpleGrantedAuthority adminAuthority = new SimpleGrantedAuthorit
y(UserType.ADMIN.toString());
```

```
            Collection<GrantedAuthority> authorities = new ArrayList<>();
            authorities.add(adminAuthority);
            return new User(adminEntity.getUsername(), adminEntity.getPasswor
d(), authorities);

        } else if (userType == UserType.USER) {
            UserEntity user = userService.findByUsername(username);
            SimpleGrantedAuthority userAuthority = new SimpleGrantedAuthority
(UserType.USER.toString());
            Collection<GrantedAuthority> authorities = new ArrayList<>();
            authorities.add(userAuthority);
            return new User(user.getUsername(), user.getPassword(), authoriti
es);
        }
        return null;
    }
}
```

7.3. JwtAuthenticationFilter

```
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    @Autowired
    private JwtGenerator jwtGenerator;

    @Autowired
    private CustomUserDetailsService customUserDetailsService;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletRe
sponse response, FilterChain filterChain)
            throws ServletException, IOException {

        String token = getJWTfromRequest(request);
        if(token != null && jwtGenerator.validateToken(token)) {
            String username = jwtGenerator.getUserNameFromJWT(token);
            String userType = jwtGenerator.getUserTypeFromJWT(token);
            customUserDetailsService.setUserType(UserType.valueOf(userType));
            UserDetails userDetails = customUserDetailsService.loadUserByUser
name(username);
            UsernamePasswordAuthenticationToken authenticationToken = new Use
rnamePasswordAuthenticationToken(
                    userDetails, null, userDetails.getAuthorities());
            authenticationToken.setDetails(new WebAuthenticationDetailsSource
().buildDetails(request));
            SecurityContextHolder.getContext().setAuthentication(authenticati
onToken);
        }
        filterChain.doFilter(request, response);
    }

    private String getJWTfromRequest(HttpServletRequest request) {
        String bearerToken = request.getHeader("Authorization");
        if(bearerToken!=null &&  bearerToken.startsWith("Bearer ")) {
            return bearerToken.substring(7);
        } else {
            return null;
        }
    }
}
```

7.4. JwtGenerator

```
@Component
public class JwtGenerator {

    @Value("${jwt.secret}")
    private String jwtSecret;

    @Value("${jwt.expirationMs}")
    private int jwtExpirationMs;

    private static final Logger logger = LoggerFactory.getLogger(JwtGenerato
r.class);

    public String generateToken(String userName, String userType) {
        Date currentDate = new Date();
        Date expiryDate = new Date(currentDate.getTime()+ jwtExpirationMs);

        String token = Jwts.builder()
                .setSubject(userName)
                .setIssuedAt(currentDate)
                .setExpiration(expiryDate)
                .signWith(getSignKey(), SignatureAlgorithm.HS256)
                .claim("usertype", userType)
                .compact();
        return token;
    }
    public String getUserNameFromJWT(String token) {
        Claims claims = Jwts.parserBuilder()
                .setSigningKey(getSignKey())
                .build()
                .parseClaimsJws(token)
                .getBody();

        return claims.getSubject();
    }

    public String getUserTypeFromJWT(String token) {
        Claims claims = Jwts.parserBuilder()
                .setSigningKey(getSignKey())
                .build()
                .parseClaimsJws(token)
                .getBody();

        return claims.get("usertype").toString();
    }

    public boolean validateToken(String token) {
        try {
            Jwts.parserBuilder().setSigningKey(getSignKey()).build().parseCla
imsJws(token);
            return true;
        } catch (MalformedJwtException e) {
            logger.error("Invalid JWT token: {}", e.getMessage());
        } catch (ExpiredJwtException e) {
            logger.error("JWT token is expired: {}", e.getMessage());
        } catch (UnsupportedJwtException e) {
            logger.error("JWT token is unsupported: {}", e.getMessage());
        } catch (IllegalArgumentException e) {
            logger.error("JWT claims string is empty: {}", e.getMessage());
        } catch (SignatureException e) {
            logger.error("Invalid JWT token: {}", e.getMessage());
        }

        return false;
    }

    private Key getSignKey() {
        byte[] keyBytes= Decoders.BASE64.decode(jwtSecret);
        return Keys.hmacShaKeyFor(keyBytes);
    }
}
```

7.5. JwtAuthEntryPoint

```
@Component
public class JwtAuthEntryPoint implements AuthenticationEntryPoint {

    private static final Logger logger = LoggerFactory.getLogger(JwtAuthEntry
Point.class);
    private static final String UNAUTHORIZED_USER = "Unauthorized User";

    @Override
    public void commence(HttpServletRequest request, HttpServletResponse resp
onse,
            AuthenticationException authException) throws IOException, Servle
tException {

        logger.error("Unauthorized error: {}", authException.getMessage());

        response.setContentType(MediaType.APPLICATION_JSON_VALUE);
        response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);

        ErrorResponse errorResponse = new ErrorResponse(UNAUTHORIZED_USER, au
thException.getMessage(), HttpServletResponse.SC_UNAUTHORIZED);

        final ObjectMapper mapper = new ObjectMapper();
        mapper.writeValue(response.getOutputStream(), errorResponse);
    }
}
```

8. The Controller

> We'll define REST endpoints for user registration, login, and accessing protected resources. These endpoints will be secured using JWT-based authentication.

8.1. AuthController

```
@RestController
@RequestMapping("/auth/user")
public class AuthController {
    @Autowired
    private UserService userService;

    @Autowired
    private AuthenticationService authenticationService;

    @PostMapping("/reg")
    @ResponseStatus(HttpStatus.CREATED)
    public String userRegister(@RequestBody UserAuthDto userAuthDto) {
        return userService.regUser(userAuthDto);
    }

    @PostMapping("/login")
    @ResponseStatus(HttpStatus.CREATED)
    public UserLoginResponseDto userLogin(@RequestBody UserAuthDto userAuthDt
o) {
        authenticationService.authenticate(UserType.USER, userAuthDto.getUser
name(), userAuthDto.getPassword());
        return userService.loginUser(userAuthDto);
    }
}
```

8.2. BasicRestUserAPI

```
@RestController
@RequestMapping("/api/public/user")
public class BasicRestUserAPI {

    @GetMapping("/")
    public ResponseEntity<String> userHome(Authentication authentication) {
        return new ResponseEntity<String>("Ok", HttpStatus.OK);
    }
}
```

9. Run The Application

> Finally, we'll run the Spring Boot application using Maven and test the registration, login, and access to protected resources using cURL commands.

```
mvn clean spring-boot:run
```

10. Usage

10.1. User Registration

```
curl -X POST -H "Content-Type: application/json" -d '{"username" : "user","pa
ssword":"123456"}' http://localhost:8081/auth/user/reg -w "\n"
```

10.2. User Login

```
curl -X POST -H "Content-Type: application/json" -d '{"username" : "user","pa
ssword":"123456"}' http://localhost:8081/auth/user/login -w "\n"
```

10.3. Public User Api

> After Login with User Creds, you will get a bearer token use that token below to access that API.

```
curl -H "Authorization: Bearer <TOKEN>" http://localhost:8081/api/public/use
r/ -w "\n"
```

> If you get a `200` status code with a response body `Ok`, your authentication microservice is succesfully working.

By following this guide, you'll be able to implement a robust token-based authentication system for your Spring Boot application, providing secure access to your resources while ensuring scalability and flexibility. Whether you're building a simple web application or a complex enterprise system, JWT-based authentication with Spring Security is a powerful solution for managing user authentication and access control.

Here is the Github Repo for this Article!