**Spring Boot Security Jwt Authentication**

Now we will be securing our REST APIs with JWT(JSOn Web Token) authentication. We will be using spring boot maven based configuration to develop and secure our APIs with seperate API for signup and generate token. We will be extending OncePerRequestFilter class to define our custom authentication mechanism using JWT.The authentication mechanism can be applied to URLs as well as for methods. And at the end, we will be testing the implementation with google advanced REST client.

Table of Contents

* 1. What is JWT
* 2. Project Structure
* 3. JWT Authentication Mechanism
* 4. Spring Boot Security Config
* 5. Default Scripts
* 6. JWT Signup
* 7. Testing the Application

**What is JWT**

[JSON Web Token (JWT)](https://jwt.io/introduction/) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.a stateless authentication mechanism as the user state is never saved in server memory.A JWT token consists of 3 parts seperated with a dot(.) i.e. Header.payload.signature

Header has 2 parts type of token and hashing algorithm used.The JSON structure comprising these two keys are Base64Encoded.

{

"alg": "HS256",

"typ": "JWT"

}

Payload contains the claims.Primarily, there are three types of claims: reserved, public, and private claims. Reserved claims are predefined claims such as iss (issuer), exp (expiration time), sub (subject), aud (audience).In private claims, we can create some custom claims such as subject, role, and others.

{

"sub": "Alex123",

"scopes": [

{

"authority": "ROLE\_ADMIN"

}

],

"iss": " <http://smitakumar.blogspot.com/>",

"iat": 1508607322,

"exp": 1508625322

}

Signature ensures that the token is not changed on the way. For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

HMACSHA256(

base64UrlEncode(header) + "." +

base64UrlEncode(payload),

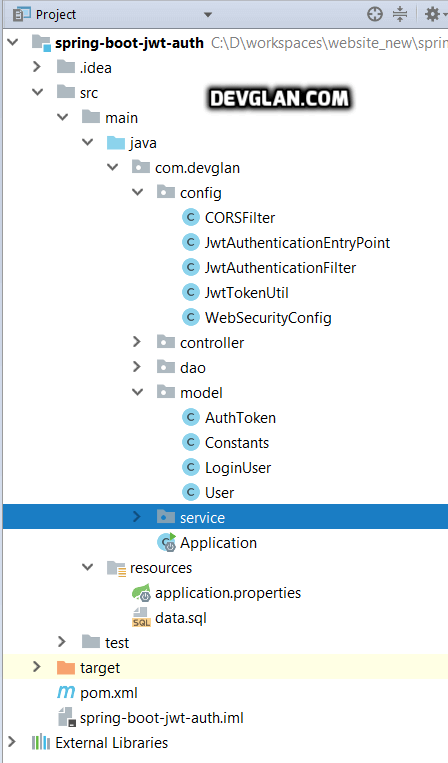
secret)

The server's protected routes will check for a valid JWT in the Authorization header, and if it's present, the user will be allowed to access protected resources.Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the Authorization header using the Bearer schema. The content of the header should look like the following:

Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJBbGV4MTIzIiwic2N.v9A80eU1VDo2Mm9UqN2FyEpyT79IUmhg

**Project Structure**

Following will be the final project structure that we will be building for our spring boot JWT authentication.

[](https://imgur.com/1hPaErj)

**JWT Authentication Mechanism**

Following class extends OncePerRequestFilter that ensures a single execution per request dispatch. This class checks for the authorization header and authenticates the JWT token and sets the authentication in the context.Doing so will protect our APIs from those requests which do not have any authorization token.The configuration about which resource to be protected and which not can be configured in WebSecurityConfig.java

**JwtAuthenticationFilter.java**

public class JwtAuthenticationFilter extends OncePerRequestFilter {

@Autowired

private UserDetailsService userDetailsService;

@Autowired

private JwtTokenUtil jwtTokenUtil;

@Override

protected void doFilterInternal(HttpServletRequest req, HttpServletResponse res, FilterChain chain) throws IOException, ServletException {

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

String username = null;

String authToken = null;

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

authToken = header.replace(TOKEN\_PREFIX,"");

try {

username = jwtTokenUtil.getUsernameFromToken(authToken);

} catch (IllegalArgumentException e) {

logger.error("an error occured during getting username from token", e);

} catch (ExpiredJwtException e) {

logger.warn("the token is expired and not valid anymore", e);

} catch(SignatureException e){

logger.error("Authentication Failed. Username or Password not valid.");

}

} else {

logger.warn("couldn't find bearer string, will ignore the header");

}

if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {

UserDetails userDetails = userDetailsService.loadUserByUsername(username);

if (jwtTokenUtil.validateToken(authToken, userDetails)) {

UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthenticationToken(userDetails, null, Arrays.asList(new SimpleGrantedAuthority("ROLE\_ADMIN")));

authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(req));

logger.info("authenticated user " + username + ", setting security context");

SecurityContextHolder.getContext().setAuthentication(authentication);

}

}

chain.doFilter(req, res);

}

}

Following is the util class to generate the auth token as well as to extract username from the token.Here is the configuration that we want url such as /token/\* and /signup/\* to be publicly available and rest of the urls to be restricted from public access.

**JwtTokenUtil.java**

@Component

public class JwtTokenUtil implements Serializable {

public String getUsernameFromToken(String token) {

return getClaimFromToken(token, Claims::getSubject);

}

public Date getExpirationDateFromToken(String token) {

return getClaimFromToken(token, Claims::getExpiration);

}

public T getClaimFromToken(String token, Function claimsResolver) {

final Claims claims = getAllClaimsFromToken(token);

return claimsResolver.apply(claims);

}

private Claims getAllClaimsFromToken(String token) {

return Jwts.parser()

.setSigningKey(SIGNING\_KEY)

.parseClaimsJws(token)

.getBody();

}

private Boolean isTokenExpired(String token) {

final Date expiration = getExpirationDateFromToken(token);

return expiration.before(new Date());

}

public String generateToken(User user) {

return doGenerateToken(user.getUsername());

}

private String doGenerateToken(String subject) {

Claims claims = Jwts.claims().setSubject(subject);

claims.put("scopes", Arrays.asList(new SimpleGrantedAuthority("ROLE\_ADMIN")));

return Jwts.builder()

.setClaims(claims)

.setIssuer("<http://devglan.com>")

.setIssuedAt(new Date(System.currentTimeMillis()))

.setExpiration(new Date(System.currentTimeMillis() + ACCESS\_TOKEN\_VALIDITY\_SECONDS\*1000))

.signWith(SignatureAlgorithm.HS256, SIGNING\_KEY)

.compact();

}

public Boolean validateToken(String token, UserDetails userDetails) {

final String username = getUsernameFromToken(token);

return (

username.equals(userDetails.getUsername())

&& !isTokenExpired(token));

}

}

Following are the constants that we have used in above implementation.

**Constants.java**

public class Constants {

public static final long ACCESS\_TOKEN\_VALIDITY\_SECONDS = 5\*60\*60;

public static final String SIGNING\_KEY = "devglan123r";

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

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

}

**Spring Boot Security Config**

Now let us define our usual spring boot security configurations.We have userDetailsService injected to fetch user credentials from database.

Here the annotation @EnableGlobalMethodSecurity enables method level security and you can annotate your method with annotations such as @Secured to provide role based authentication at method level.

**WebSecurityConfig.java**

@Configuration

@EnableWebSecurity

@EnableGlobalMethodSecurity(prePostEnabled = true)

public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

@Resource(name = "userService")

private UserDetailsService userDetailsService;

@Autowired

private JwtAuthenticationEntryPoint unauthorizedHandler;

@Override

@Bean

public AuthenticationManager authenticationManagerBean() throws Exception {

return super.authenticationManagerBean();

}

@Autowired

public void globalUserDetails(AuthenticationManagerBuilder auth) throws Exception {

auth.userDetailsService(userDetailsService)

.passwordEncoder(encoder());

}

@Bean

public JwtAuthenticationFilter authenticationTokenFilterBean() throws Exception {

return new JwtAuthenticationFilter();

}

@Override

protected void configure(HttpSecurity http) throws Exception {

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

authorizeRequests()

.antMatchers("/token/\*").permitAll()

.anyRequest().authenticated()

.and()

.exceptionHandling().authenticationEntryPoint(unauthorizedHandler).and()

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

http

.addFilterBefore(authenticationTokenFilterBean(), UsernamePasswordAuthenticationFilter.class);

}

@Bean

public BCryptPasswordEncoder encoder(){

return new BCryptPasswordEncoder();

}

}

Following is the controller that is exposed to create token on user behalf and if you noticed in WebSecurityConfig.java we have configured this url to have no authentication so that user can generate JWT token with valid credentials.

**AuthenticationController.java**

@RestController

@RequestMapping("/token")

public class AuthenticationController {

@Autowired

private AuthenticationManager authenticationManager;

@Autowired

private JwtTokenUtil jwtTokenUtil;

@Autowired

private UserService userService;

@RequestMapping(value = "/generate-token", method = RequestMethod.POST)

public ResponseEntity register(@RequestBody LoginUser loginUser) throws AuthenticationException {

final Authentication authentication = authenticationManager.authenticate(

new UsernamePasswordAuthenticationToken(

loginUser.getUsername(),

loginUser.getPassword()

)

);

SecurityContextHolder.getContext().setAuthentication(authentication);

final User user = userService.findOne(loginUser.getUsername());

final String token = jwtTokenUtil.generateToken(user);

return ResponseEntity.ok(new AuthToken(token));

}

}

We have very simple REST Apis exposed for testing purpose.Following is the implementation.

@RestController

public class UserController {

@Autowired

private UserService userService;

@RequestMapping(value="/user", method = RequestMethod.GET)

public List listUser(){

return userService.findAll();

}

@RequestMapping(value = "/user/{id}", method = RequestMethod.GET)

public User getOne(@PathVariable(value = "id") Long id){

return userService.findById(id);

}

}

Following is our entity class.

**User.java**

@Entity @Table(name = "jwt\_user")

public class User {

@Id

@GeneratedValue(strategy= GenerationType.AUTO)

private long id;

@Column

private String username;

@Column

@JsonIgnore

private String password;

@Column

private long salary;

@Column

private int age;

**Default Scripts**

Following are the insert statements that are inserted when application starts.

**INSERT** **INTO** jwt\_user (id, firstname, lastname, username, password, salary, age) **VALUES** (1, 'Smita', 'Kumar', 'smita', '$2a$04$WeT1SvJaGjmvQj34QG8VgO9qdXecKOYKEDZtCPeeIBSTxxEhazNla', 9999, 40);

**INSERT** **INTO** jwt\_user (id, firstname, lastname, username, password, salary, age) **VALUES** (2, 'Alex','Knr', 'alex123','$2a$04$4vwa/ugGbBVDvbWaKUVZBuJbjyQyj6tqntjSmG8q.hi97.xSdhj/2', 3456, 33);

**INSERT** **INTO** jwt\_user (id, firstname, lastname, username, password, salary, age) **VALUES** (3, 'Tom', 'Asr', 'tom234', '$2a$04$QED4arFwM1AtQWkR3JkQx.hXxeAk/G45NiRd3Q4ElgZdzGYCYKZOW', 7823, 23);

**INSERT** **INTO** jwt\_user (id, firstname, lastname, username, password, salary, age) **VALUES** (4, 'Adam', 'Psr', 'adam', '$2a$04$WeT1SvJaGjmvQj34QG8VgO9qdXecKOYKEDZtCPeeIBSTxxEhazNla', 4234, 45);

**JWT User Signup**

As we have default scripts to pre-populate data in the database for our testing purpose but we can also have an API exposed for user registration. Using this API user can register and use the same username and password to generate token. For this we added following method in our controller class.

@RequestMapping(value="/signup", method = RequestMethod.POST)

public User saveUser(@RequestBody UserDto user){

return userService.save(user);

}

Once, this is added, we need to remove restrictions on this url for public access. To do so, add below line in our spring boot security config.

.antMatchers("/token/\*", "/signup").permitAll()

Now to create user, we have simple implementation in our UserServiceImpl.java to save user record in database.One thing to note here is the password encryption using bcrypt encoder. We have autowired the same encoder that we have defined as a bean in WebSecurityConfig.java

@Autowired

private BCryptPasswordEncoder bcryptEncoder;

@Override

public User save(UserDto user) {

User newUser = new User();

newUser.setUsername(user.getUsername());

newUser.setPassword(bcryptEncoder.encode(user.getPassword()));

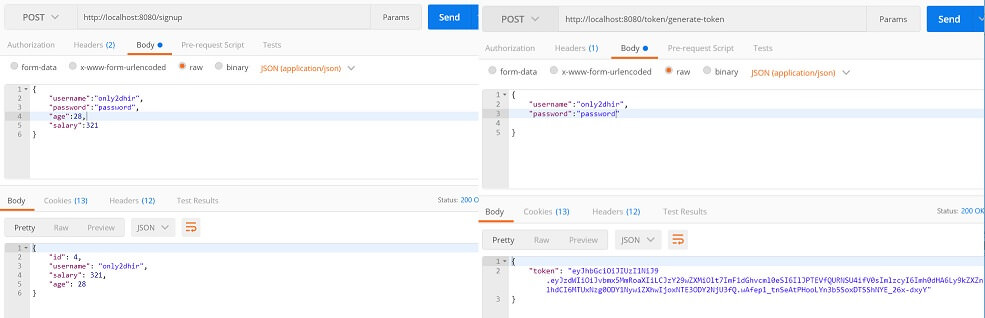
newUser.setAge(user.getAge());

newUser.setSalary(user.getSalary());

return userDao.save(newUser);

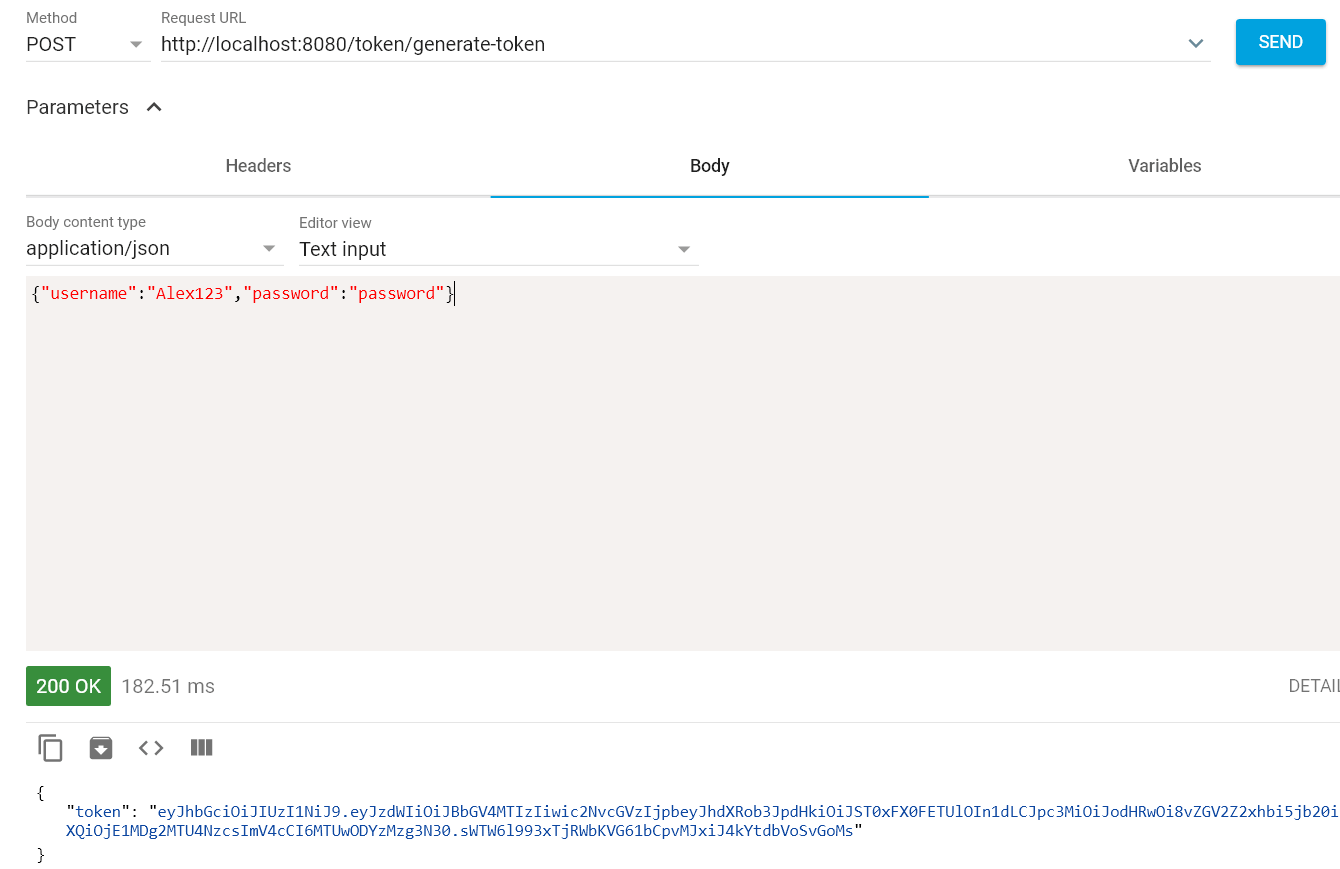
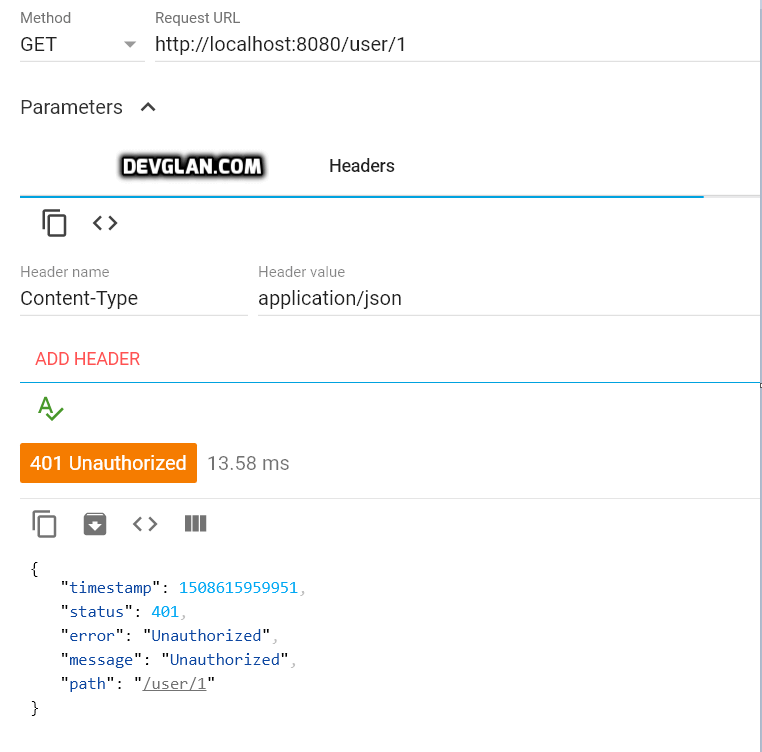
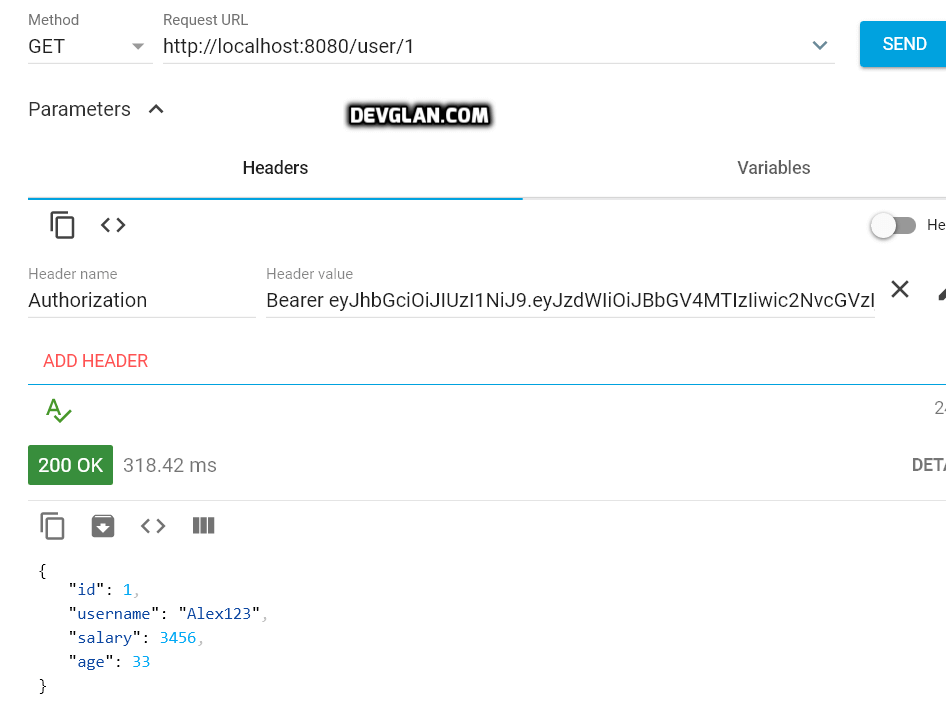
}

This will expose following urls for user signup process to generate JWT token.

[](https://imgur.com/DvYKCAE)

**Testing the Application**

We will be using Advanced REST Client to test the spring boot jwt authentication.

**Generate AuthToken** [](https://imgur.com/UzPtuXV)**Accessing Resource Without Token** [](https://imgur.com/wFfuMnB)**Accessing Resource With Token** [](https://imgur.com/IloIRvq)