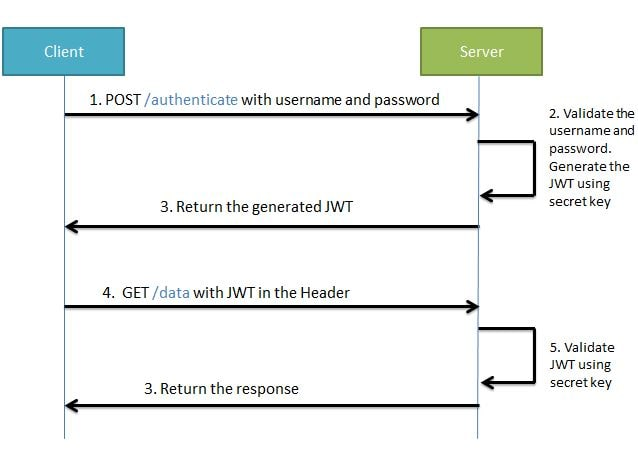
**JWT**

JWT stands for JSON Web Token. 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. The client will need to authenticate with the server using the credentials only once. During this time the server validates the credentials and returns the client a JSON Web Token(JWT). For all future requests the client can authenticate itself to the server using this JSON Web Token(JWT) and so does not need to send the credentials like username and password.

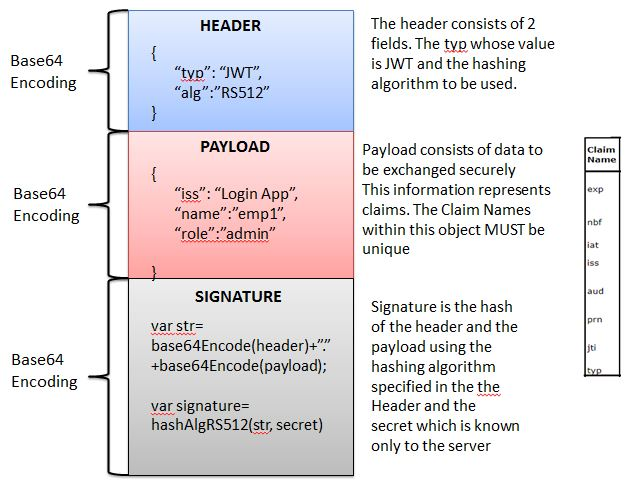


During the first request the client sends a POST request with username and password. Upon successful authentication the server generates the JWT sends this JWT to the client. This JWT can contain a payload of data. On all subsequent requests the client sends this JWT token in the header. Using this token the server authenticates the user. So we don't need the client to send the user name and password to the server during each request for authentication, but only once after which the server issues a JWT to the client. A JWT payload can contain things like user ID so that when the client again sends the JWT, you can be sure that it is issued by you, and you can see to whom it was issued.

Structure of JWT

JWT has the following format -header.payload.signature



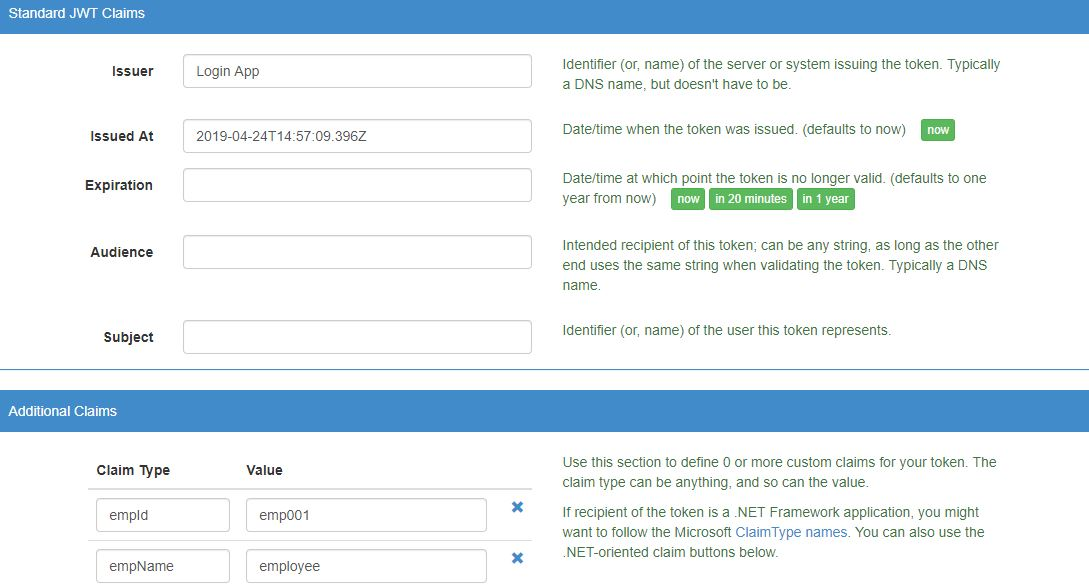


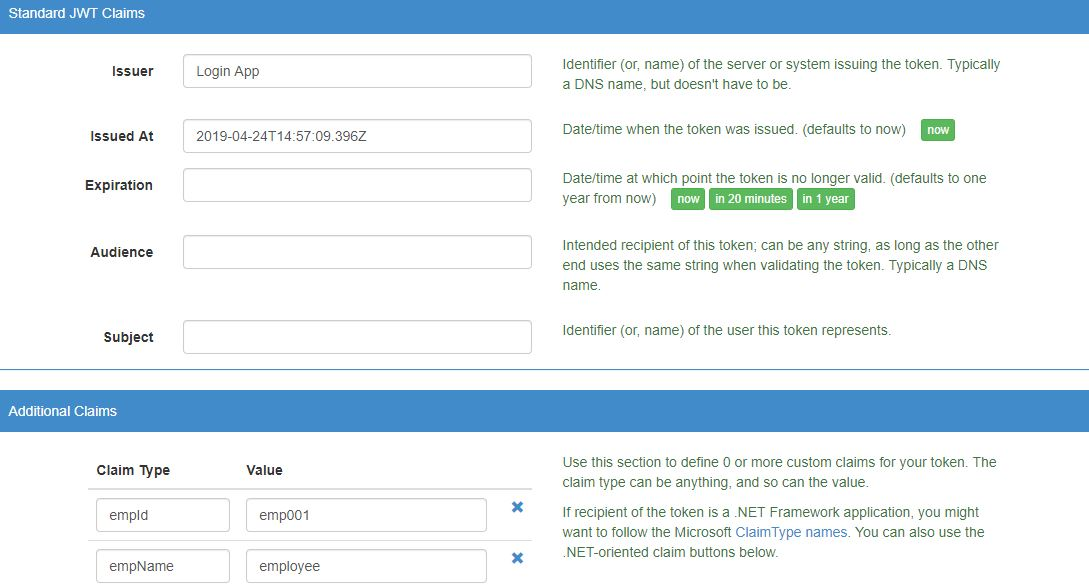
An important point to remember about JWT is that the information in the payload of the JWT is visible to everyone. So we should not pass any sensitive information like passwords in the payload. We can encrypt the payload data if we want to make it more secure. However we can be sure that no one can tamper and change the payload information. If this is done the server will recognize it.

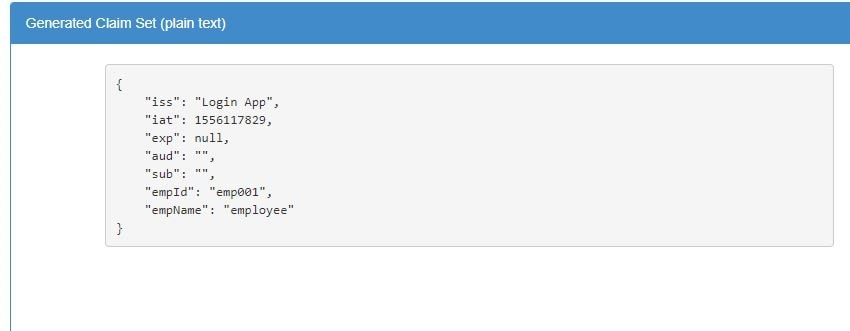
Creating a JWT Token

We will be creating a JWT token using JWT Online Token Generator

Specify the payload data as folows-





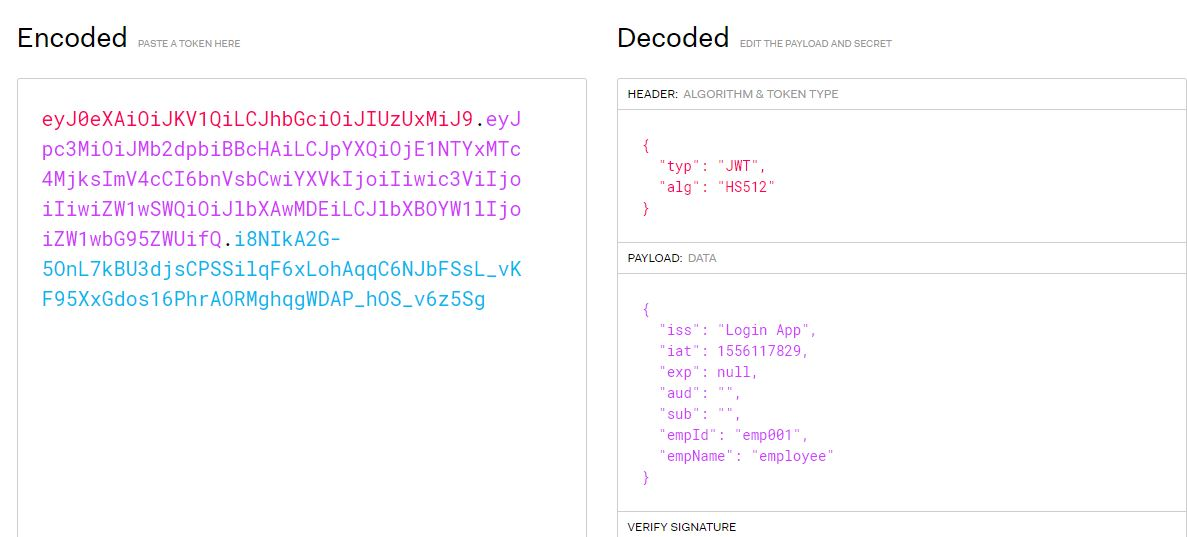


Sign the payload using the hashing algorithm-



Inspect the contents of the created token

We will be inspecting JWT token using JWT Online Decoder

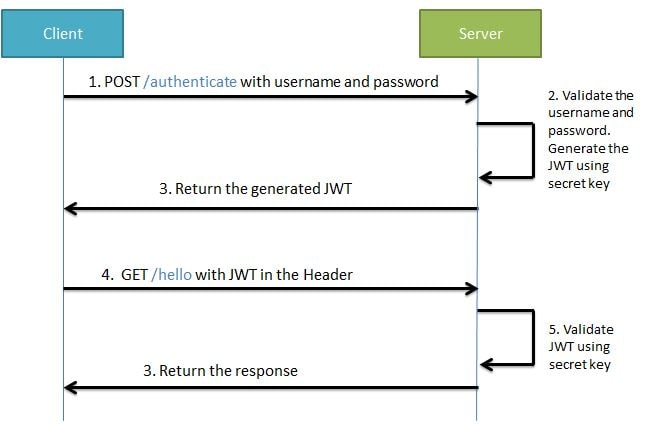


We will be making use of hard coded user values for User Authentication. In next tutorial we will be implementing **Spring Boot + JWT + MYSQL JPA** for storing and fetching user credentials. Any user will be able to consume this API only if it has a valid JSON Web Token(JWT). In a previous tutorial we have seen what is JWT, when and how to use it.

Lets Begin?

For better understanding we will be developing the project in stages

* Develop a Spring Boot Application to expose a Simple REST GET API with mapping /hello.
* Configure Spring Security for JWT. Expose REST POST API with mapping /authenticate using which User will get a valid JSON Web Token. And then allow the user access to the api /hello only if it has a valid token



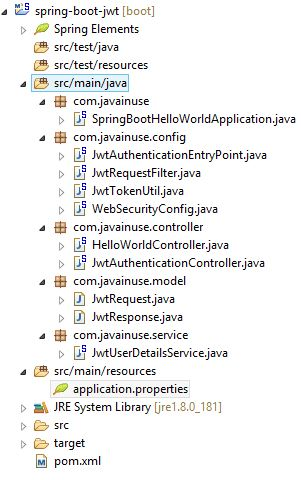
**Spring Security and JWT Configuration**

We will be configuring Spring Security and JWT for performing 2 operations-

**Generating JWT** - Expose a POST API with mapping /authenticate. On passing correct username and password it will generate a JSON Web Token(JWT)

**Validating JWT** - If user tries to access GET API with mapping /hello. It will allow access only if request has a valid JSON Web Token(JWT)

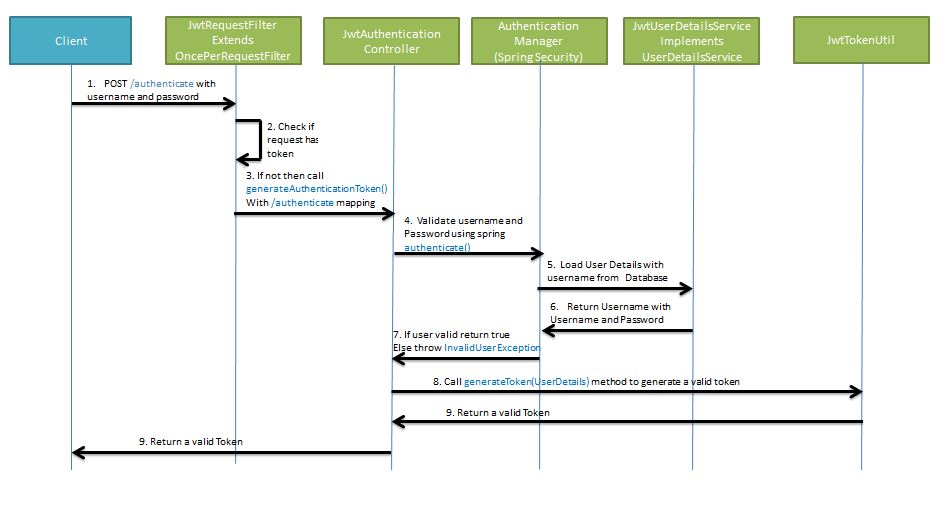
Maven Project will be as follows-

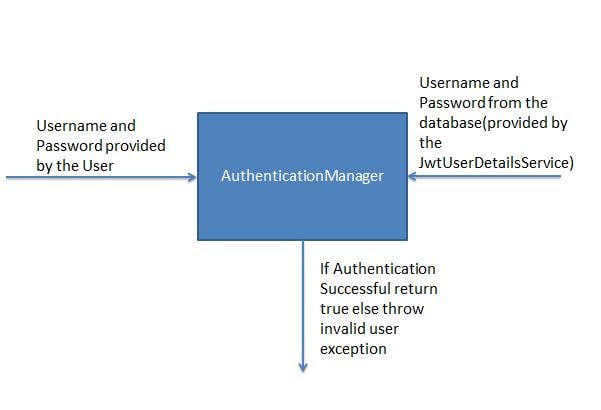


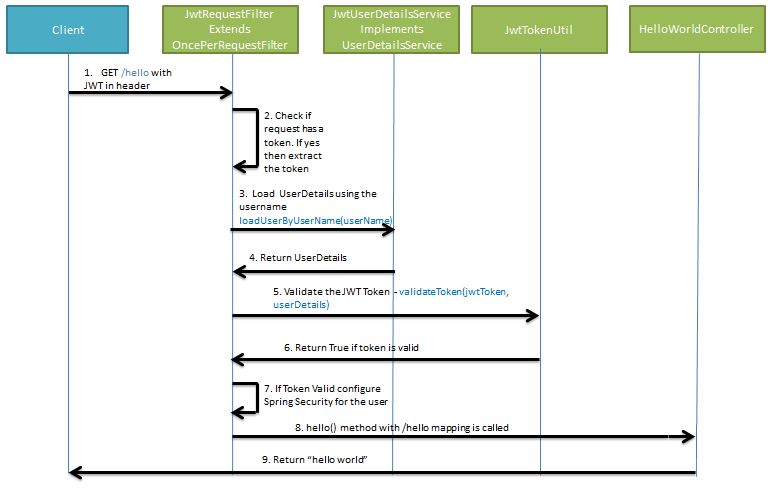
Create a Controller class for exposing a GET REST API-

The sequence flow for these operations will be as follows-

**Generating JWT**







Add the Spring Security and JWT dependencies

**POM.xml**

|  |
| --- |
| <dependency> |
| <groupId>org.springframework.boot</groupId> |
| <artifactId>spring-boot-starter-security</artifactId> |
| </dependency> |
| <dependency> |
| <groupId>io.jsonwebtoken</groupId> |
| <artifactId>jjwt</artifactId> |
| <version>0.9.1</version> |
| </dependency> |

Define the **application.properties**. As see in previous JWT tutorial, we specify the secret key using which we will be using for hashing algorithm. The secret key is combined with the header and the payload to create a unique hash. We are only able to verify this hash if you have the secret key.

**jwt.secret=javainuse**

**JwtTokenUtil**

The JwtTokenUtil is responsible for performing JWT operations like creation and validation.It makes use of the io.jsonwebtoken.Jwts for achieving this.

//Bring the value to properties file

**@Value("${jwt.secret}")**

**private String secret;**

**JwtTokenUtil** implements Serializable

@Component

public class JwtTokenUtil implements Serializable

{

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

@Value("${jwt.secret}")

private String secret;

//**retrieve username from jwt token**

public String getUsernameFromToken(String token) {

return getClaimFromToken(token, Claims::getSubject);

}

//**retrieve expiration date from jwt token**

public Date getExpirationDateFromToken(String token) {

return getClaimFromToken(token, Claims::getExpiration);

}

public <T> T getClaimFromToken(String token, Function<Claims, T> claimsResolver) {

final Claims claims = getAllClaimsFromToken(token);

return claimsResolver.apply(claims);

}

//for retrieveing any information from token we will need the secret key

private Claims getAllClaimsFromToken(String token) {

return Jwts.parser().setSigningKey(secret).parseClaimsJws(token).getBody();

}

//check if the token has expired

private Boolean isTokenExpired(String token) {

final Date expiration = getExpirationDateFromToken(token);

return expiration.before(new Date());

}

//generate token for user

public String generateToken(UserDetails userDetails) {

Map<String, Object> claims = new HashMap<>();

return doGenerateToken(claims, userDetails.getUsername());

}

//while creating the token -

//1. Define claims of the token, like Issuer, Expiration, Subject, and the ID

//2. Sign the JWT using the HS512 algorithm and secret key.

//3. According to JWS Compact Serialization(<https://tools.ietf.org/html/>

//draft-ietf-jose-json-web-signature-41#section-3.1)

// compaction of the JWT to a URL-safe string

private String doGenerateToken(Map<String, Object> claims, String subject) {

return Jwts.builder().setClaims(claims).setSubject(subject).setIssuedAt(new Date(System.currentTimeMillis())).setExpiration(new Date(System.currentTimeMillis() + JWT\_TOKEN\_VALIDITY \* 1000))

.signWith(SignatureAlgorithm.HS512, secret).compact();

}

//validate token

public Boolean validateToken(String token, UserDetails userDetails) {

final String username = getUsernameFromToken(token);

return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));

}

**JWTUserDetailsService**

JWTUserDetailsService implements the Spring Security UserDetailsService interface. It overrides the loadUserByUsername for fetching user details from the database using the username. The Spring Security Authentication Manager calls this method for getting the user details from the database when authenticating the user details provided by the user. Here we are getting the user details from a hardcoded User List. In the next tutorial we will be adding the DAO implementation for fetching User Details from the Database. Also the password for a user is stored in encrypted format using BCrypt. Previously we have seen Spring Boot Security - Password Encoding Using Bcrypt. Here using the Online Bcrypt Generator you can generate the Bcrypt for a password.

@Service

public class JwtUserDetailsService implements UserDetailsService {

@Override

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

if ("javainuse".equals(username)) {

return new User("javainuse", "$2a$10$slYQmyNdGzTn7ZLBXBChFOC9f6kFjAqPhccnP6DxlWXx2lPk1C3G6",

new ArrayList<>());

} else {

throw new UsernameNotFoundException("User not found with username: " + username); } } }

**JwtAuthenticationController**

Expose a POST API /authenticate using the JwtAuthenticationController. The POST API gets username and password in the body- Using Spring Authentication Manager we authenticate the username and password.If the credentials are valid, a JWT token is created using the JWTTokenUtil and provided to the client.

@RestController

@CrossOrigin

public class **JwtAuthenticationController** {

@Autowired

private AuthenticationManager authenticationManager;

@Autowired

private JwtTokenUtil jwtTokenUtil;

@Autowired

private JwtUserDetailsService userDetailsService;

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

public ResponseEntity<?> createAuthenticationToken(@RequestBody JwtRequest authenticationRequest) throws Exception {

authenticate(authenticationRequest.getUsername(), authenticationRequest.getPassword());

final UserDetails userDetails = userDetailsService

.loadUserByUsername(authenticationRequest.getUsername());

final String token = jwtTokenUtil.generateToken(userDetails);

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

}

private void authenticate(String username, String password) throws Exception { try {

authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(username, password));

} catch (DisabledException e) {

throw new Exception("USER\_DISABLED", e);

} catch (BadCredentialsException e) {

throw new Exception("INVALID\_CREDENTIALS", e); }

**JwtRequest**

This class is required for storing the username and password we recieve from the client.

public class **JwtRequest** implements Serializable {

private String username;

private String password;

//default constructors

//All args constructor

//setters and getters

}

**JwtResponse**

This is class is required for creating a response containing the JWT to be returned to the user.

public class **JwtResponse** implements Serializable {

private static final long serialVersionUID = -8091879091924046844L;

private final String jwttoken;

public JwtResponse(String jwttoken) {this.jwttoken = jwttoken; }

public String getToken() {return this.jwttoken; }

**JwtRequestFilter**

The JwtRequestFilter extends the Spring Web Filter OncePerRequestFilter class. For any incoming request this Filter class gets executed. It checks if the request has a valid JWT token. If it has a valid JWT Token then it sets the Authentication in the context, to specify that the current user is authenticated.

@Component

public class **JwtRequestFilter** extends OncePerRequestFilter {

@Autowired

private JwtUserDetailsService jwtUserDetailsService;

@Autowired

private JwtTokenUtil jwtTokenUtil;

@Override

protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain)

throws ServletException, IOException {

final String requestTokenHeader = request.getHeader("Authorization");

String username = null;

String jwtToken = null;

// JWT Token is in the form "Bearer token". Remove Bearer word and get

// only the Token

if (requestTokenHeader != null && requestTokenHeader.startsWith("Bearer ")) {

jwtToken = requestTokenHeader.substring(7);

try {

username = jwtTokenUtil.getUsernameFromToken(jwtToken);

} catch (IllegalArgumentException e) {

System.out.println("Unable to get JWT Token");

} catch (ExpiredJwtException e) {

System.out.println("JWT Token has expired");

}

} else {

logger.warn("JWT Token does not begin with Bearer String");

}

// Once we get the token validate it.

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

UserDetails userDetails = this.jwtUserDetailsService.loadUserByUsername(username);

// if token is valid configure Spring Security to manually set

// authentication

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

UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken = new UsernamePasswordAuthenticationToken(

userDetails, null, userDetails.getAuthorities());

usernamePasswordAuthenticationToken .setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

// After setting the Authentication in the context, we specify

// that the current user is authenticated. So it passes the

// Spring Security Configurations successfully.

SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticationToken); } }

chain.doFilter(request, response); }

**JwtAuthenticationEntryPoint**

This class will extend Spring's AuthenticationEntryPoint class and override its method commence. It rejects every unauthenticated request and send error code 401

@Component

public class **JwtAuthenticationEntryPoint** implements AuthenticationEntryPoint, Serializable {

private static final long serialVersionUID = -7858869558953243875L;

@Override

public void commence(HttpServletRequest request, HttpServletResponse response,

AuthenticationException authException) throws IOException {

response.sendError(HttpServletResponse.SC\_UNAUTHORIZED, "Unauthorized");

} }

**WebSecurityConfig**

This class extends the WebSecurityConfigurerAdapter is a convenience class that allows customization to both WebSecurity and HttpSecurity.

@Configuration

@EnableWebSecurity

@EnableGlobalMethodSecurity(prePostEnabled = true)

public class **WebSecurityConfig** extends WebSecurityConfigurerAdapter {

@Autowired

private JwtAuthenticationEntryPoint jwtAuthenticationEntryPoint;

@Autowired

private UserDetailsService jwtUserDetailsService;

@Autowired

private JwtRequestFilter jwtRequestFilter;

@Autowired

public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {

// configure AuthenticationManager so that it knows from where to load

// user for matching credentials

// Use BCryptPasswordEncoder

auth.userDetailsService(jwtUserDetailsService).passwordEncoder(passwordEncoder());

}

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

@Bean

@Override

public AuthenticationManager authenticationManagerBean() throws Exception {

return super.authenticationManagerBean();

}

@Override

protected void configure(HttpSecurity httpSecurity) throws Exception {

// We don't need CSRF for this example

httpSecurity.csrf().disable()

// dont authenticate this particular request

.authorizeRequests().antMatchers("/authenticate").permitAll().

// all other requests need to be authenticated

anyRequest().authenticated().and().

// make sure we use stateless session; session won't be used to

// store user's state.

exceptionHandling().authenticationEntryPoint(jwtAuthenticationEntryPoint).and().sessionManagement()

.sessionCreationPolicy(SessionCreationPolicy.STATELESS);

// Add a filter to validate the tokens with every request

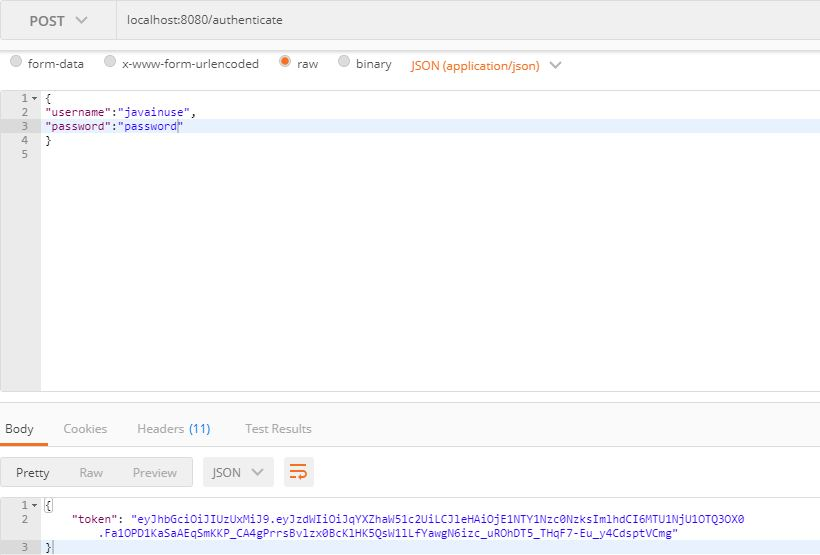
httpSecurity.addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class);

}

Start the Spring Boot Application

**Generate a JSON Web Token -**

Create a POST request with url localhost:8080/authenticate. Body should have valid username and password. In our case username is javainuse and password is password.



**Validate the JSON Web Token**

- Try accessing the url localhost:8080/hello using the above generated token in the header as follows

