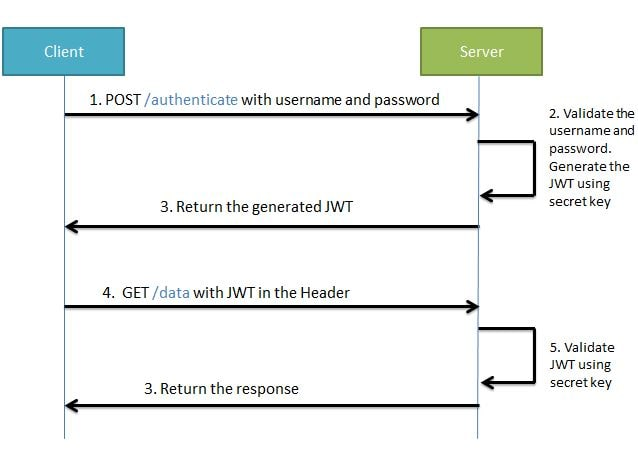
**JWT**

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.

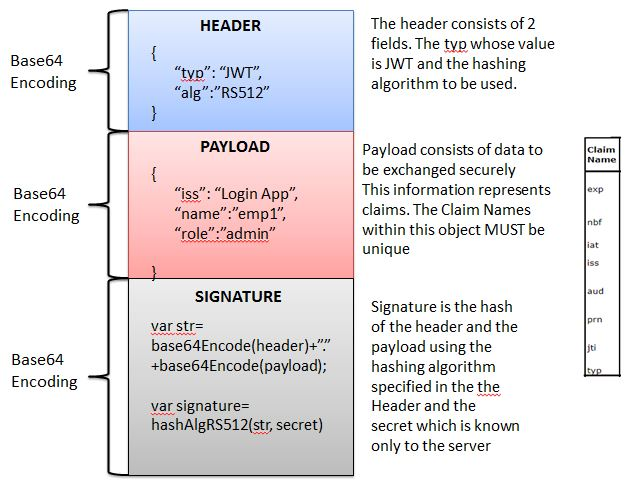


During the first request the client sends a POST request with username and password. Upon successful authentication the server generates the JWT sends this JWT to the client. This JWT can contain a payload of data. On all subsequent requests the client sends this JWT token in the header. Using this token 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



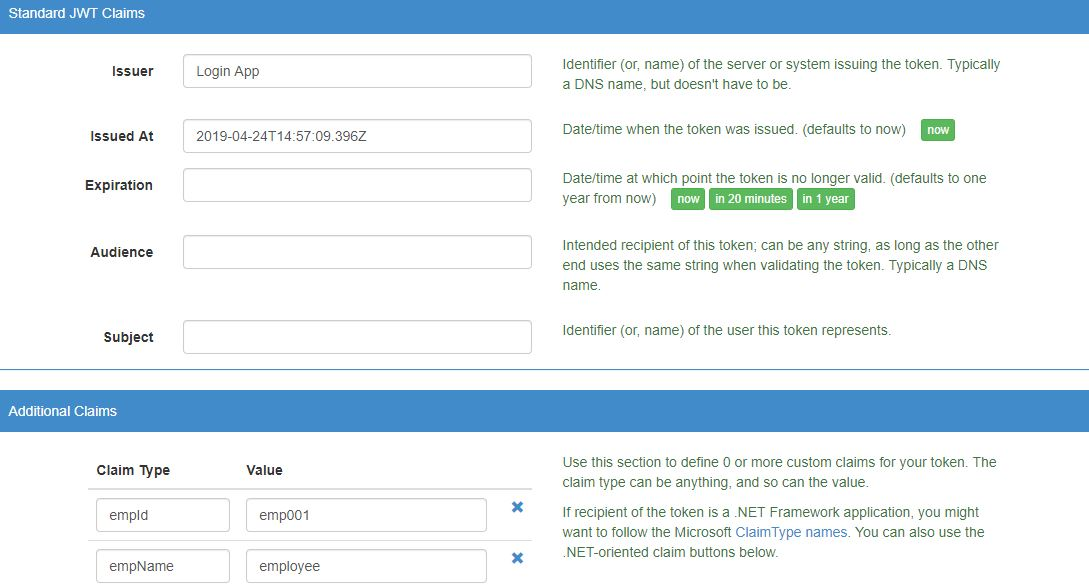


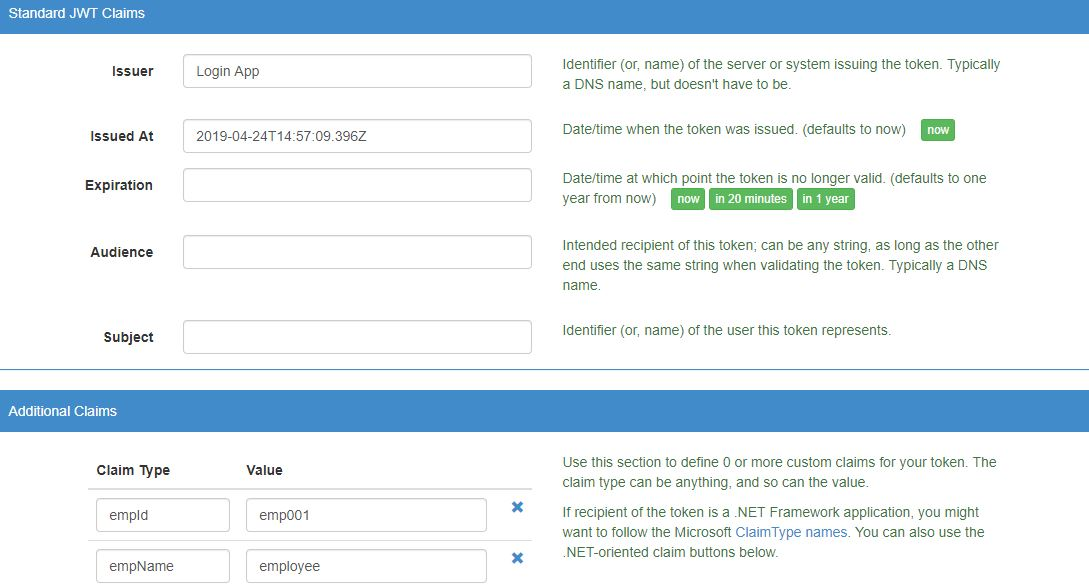
An important point to remember about JWT is that the information in the payload of the JWT is visible to everyone. So we should not pass any sensitive information like passwords in the payload. We can encrypt the payload data if we want to make it more secure. However we can be sure that no one can tamper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

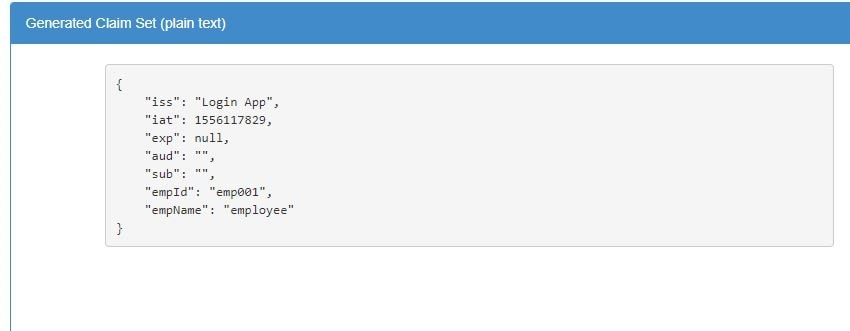
Creating a JWT Token

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

Specify the payload data as folows-





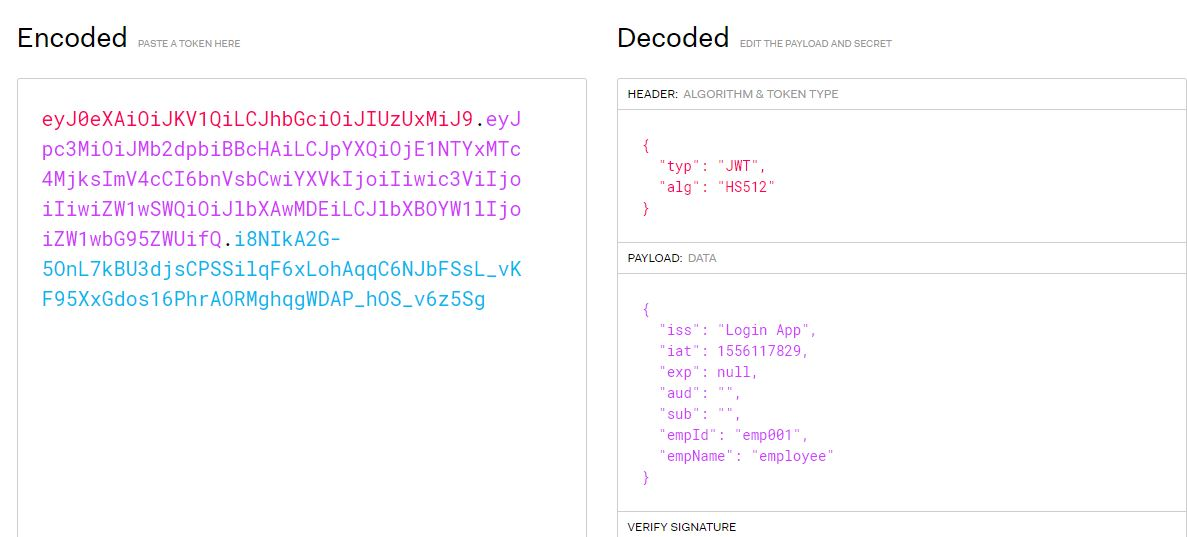


Sign the payload using the hashing algorithm-



Inspect the contents of the created token

We will be inspecting JWT token using JWT Online Decoder

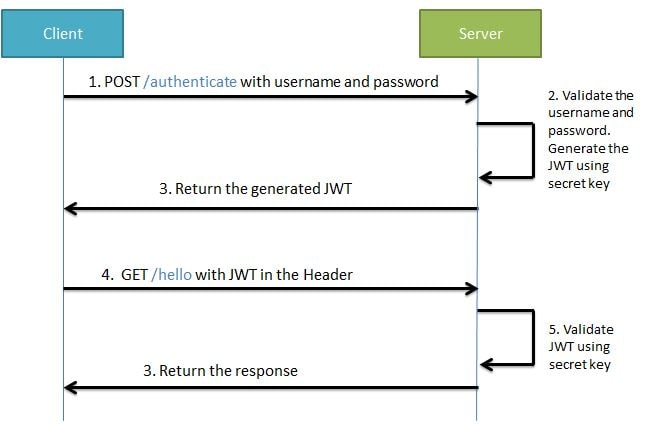


We will be making use of hard coded user values for User Authentication. In next tutorial we will be implementing **Spring Boot + JWT + MYSQL JPA** for storing and fetching user credentials. Any user will be able to consume this API only if it has a valid JSON Web Token(JWT). In a previous tutorial we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



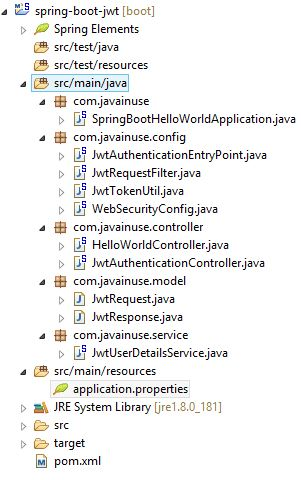
**Spring Security and JWT Configuration**

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)

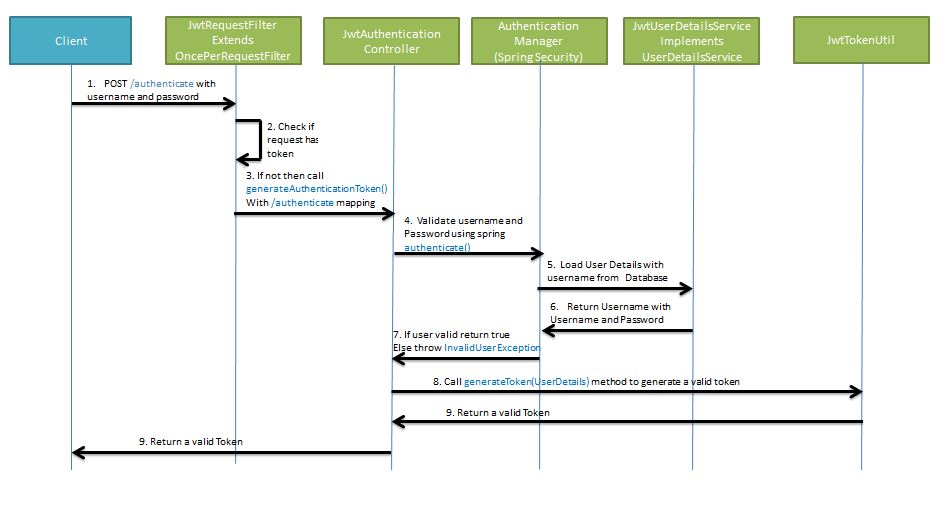
Maven Project will be as follows-

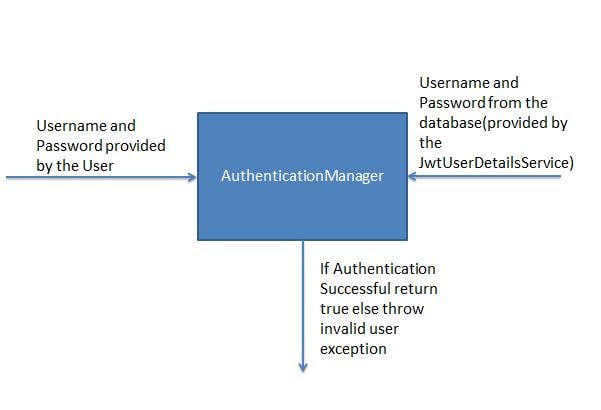


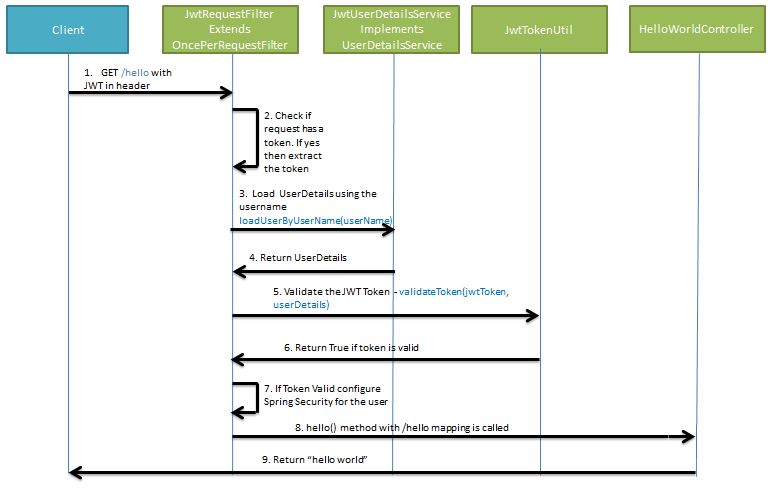
Create a Controller class for exposing a GET REST API-

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

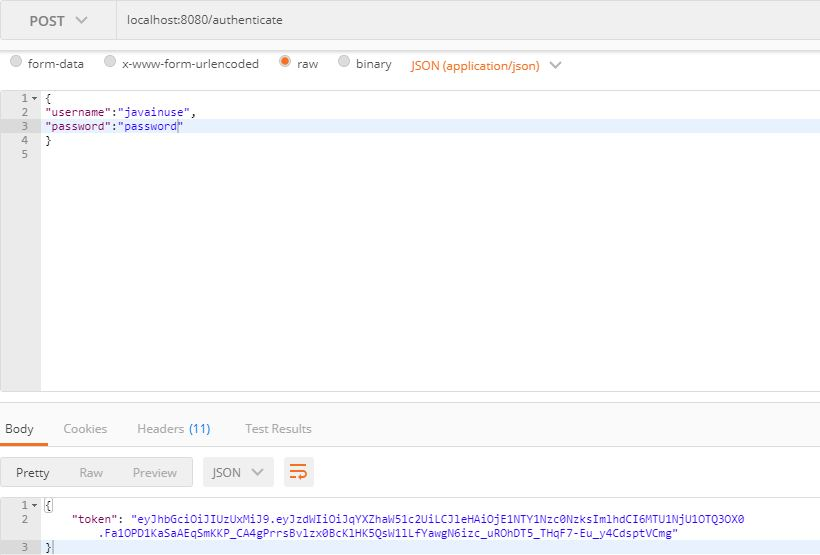
httpSecurity.addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class);

}

Start the Spring Boot Application

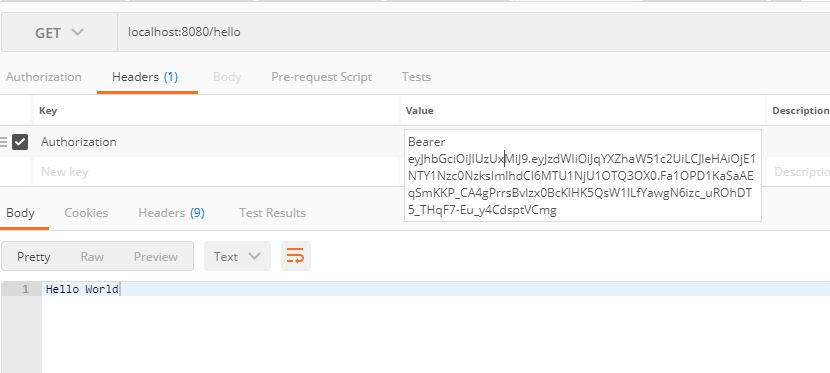
**Generate a JSON Web Token -**

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**

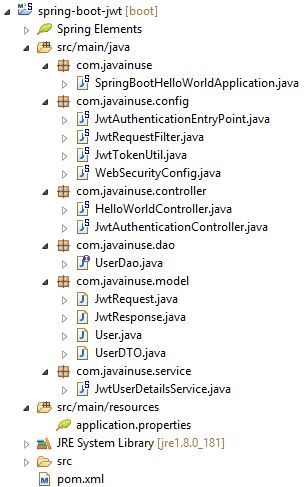
- Try accessing the url localhost:8080/hello using the above generated token in the header as follows



Lets Begin?

Implement MYSQL JPA for storing and fetching user details

The starting code for this tutorial will be the **Spring Boot + JWT Hello World** Example we had implemented previously. Currently using JwtUserDetailsService we are validating the user. We are doing this using hard coded values for username and password. Now we will be using Spring Data JPA to validate user credentials by fetching username and password from the mysql db. The maven project will be as follows-



**Inserting a user**

Define the database properties as follows-

**application.properties**

**jwt.secret=vinodinuse**

spring.datasource.url=jdbc:mysql://localhost/bootdb?createDatabaseIfNotExist=true&autoReconnect=true&useSSL=false

spring.datasource.username=root

spring.datasource.password=root

spring.datasource.platform=mysql

spring.jpa.hibernate.ddl-auto=create-drop

In a previous tutorial we had implemented Spring Boot + Spring Data JPA Hello World Example. Create the Entity class as follows. It will be used while performing database operations-

@Entity

@Table(name = "user")

public class **DAOUser** {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private long id;

@Column

private String username;

@Column

@JsonIgnore

private String password;

//setters and getters }

Define the **UserDTO** model class as follows. It is responsible for getting values from user and passing it to the DAO layer for inserting in database.

public class **UserDTO** {

private String username;

private String password;

//setters and getters

}

Next we define the UserDao which is an interface that extends the Spring Framework class CrudRepository. CrudRepository class is a generics and takes the following two parameters as arguments- What type of Object will this repository be working with- In our case DAOUser and Id will be what type of object- Integer(since id defined in the UserDao class is Integer) Thats the only configuration required for the repository class. The required operation of inserting user details in DB will now be handled. Define the DAO class as follows.

@Repository

public interface **UserDao** extends CrudRepository<DAOUser, Integer> { }

Allow the url /register to be allowed without applying spring security-

@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", "/register").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);

}

}

In the **JwtUserDetailsService**, autowire the UserDao bean and the BcryptEncoder bean. Also define the saveUser function for inserting user details-

public class **JwtUserDetailsService** implements UserDetailsService {

@Autowired

private UserDao userDao;

@Autowired

private PasswordEncoder bcryptEncoder;

@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);

}

}

public UserDao save(UserDTO user) {

DAOUser newUser = new DAOUser();

newUser.setUsername(user.getUsername());

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

return userDao.save(newUser); }

Finally modify the Controller class for adding a POST request for adding user details to database.

@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)); }

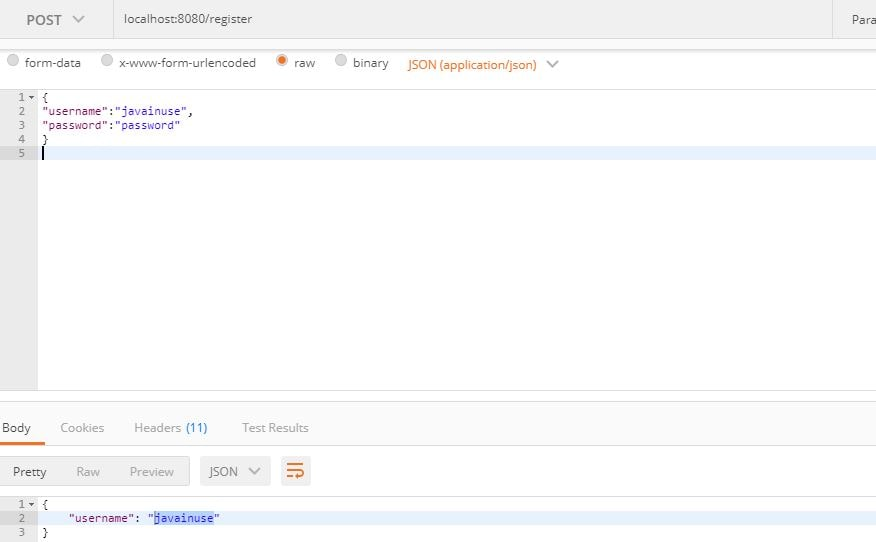
@RequestMapping(value = "/register", method = RequestMethod.POST)

public ResponseEntity<?> saveUser(@RequestBody UserDTO user) throws Exception { return ResponseEntity.ok(userDetailsService.save(user)); }

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) } } }

Start the Spring Boot Application- Register a new user by creating a post request to url /register and the body having username and password



Make use of Database credentials for authentication

In the UserDao interface add a method findByUsername(String username)

@Repository

public interface **UserDao** extends CrudRepository<DAOUser, Integer> {

UserDao findByUsername(String username); }

In the loadUserByUsername method, we fetch the user records from the database instead of using hardcoded value.

@Service

public class **JwtUserDetailsService** implements UserDetailsService {

@Autowired

private UserDao userDao;

@Autowired

private PasswordEncoder bcryptEncoder;

@Override

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

DAOUser user = userDao.findByUsername(username);

if (user == null) { throw new UsernameNotFoundException("User not found with username: " + username); }

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

new ArrayList<>()); }

public User save(UserDTO user) {

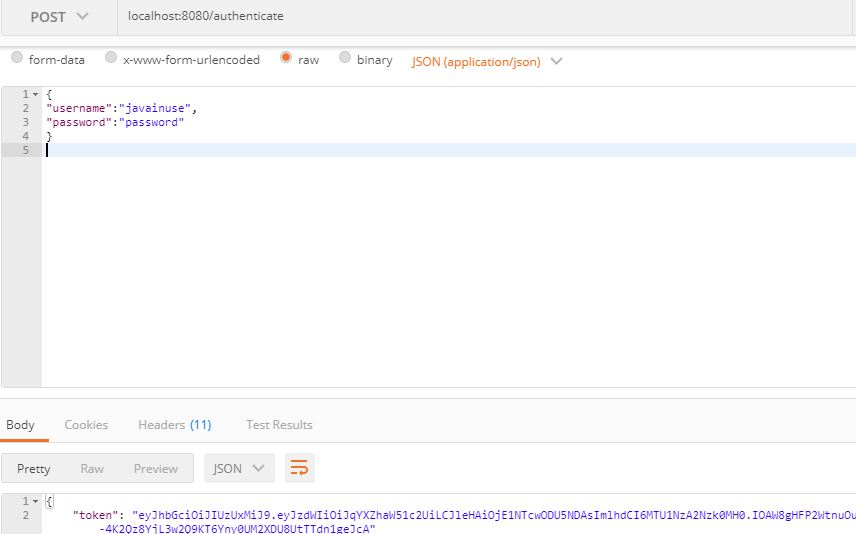
DAOUser newUser = new DAOUser();

newUser.setUsername(user.getUsername());

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

return userDao.save(newUser); } }

Generate a new Token by creating a post request to url /authenticate and the body having username and password

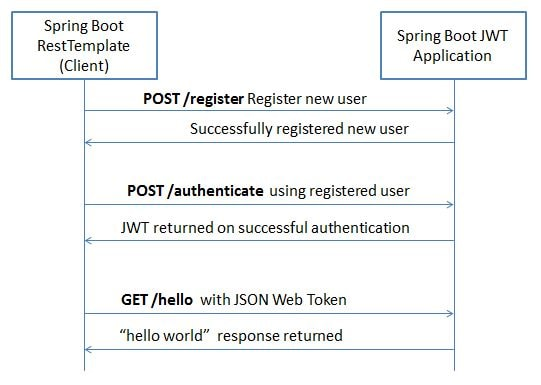


Create the domain class named User as follows. This class will be used for User Authentication.

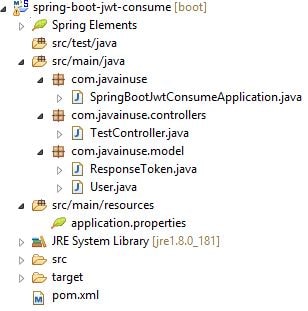
**Spring Boot RestTemplate + JWT Authentication Example**

In a previous tutorial we had implemented Spring Boot + MYSQL + JWT Authentication Example

Previously we had consumed the exposed service using external client like Postman. But we may also need to call this JWT authenticated service from some other microservice. In this tutorial we will be consuming the JWT authenticated exposed service programmatically using RestTemplate. Using the Spring Boot RestTemplate as the client we will be performing the following operations-



Let’s Begin



public class **User** {

private String username;

private String password;

//setters and getters

}

Create the domain class named RegistrationUser as follows. This class will be used for User Registration. This class extends the User class and has additional field role.

public class **RegistrationUser** extends User {

private String role;

public String getRole() { return role; }

public void setRole(String role) { this.role = role; } }

Create the **ResponseToken** class. This will hold the JWT response returned on successful authentication.

public class **ResponseToken** {

private String token;

//setters and getters }

Define the Bootstrap class with the @SpringBootApplication annotation. Also in this class we will be creating the bean of type RestTemplate.

@SpringBootApplication

public class **SpringBootJwtConsumeApplication** {

public static void main(String[] args) {

SpringApplication.run(SpringBootJwtConsumeApplication.class, args);

}

@Bean

public RestTemplate getRestTemplate() {

return new RestTemplate();

}

}

Finally create the TestController class. In this class we will be autowiring the RestTemplate bean we had created previously. Using the RestTemplate we will

* Register new User
* Authenticate the registered User to get JWT
* Using JWT make a call to the hello world service

@RestController

public class **TestController** {

@Autowired

RestTemplate restTemplate;

private static final String REGISTRATION\_URL = "http://localhost:8080/register";

private static final String AUTHENTICATION\_URL = "http://localhost:8080/authenticate";

private static final String HELLO\_URL = "http://localhost:8080/helloadmin";

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

public String getResponse() throws JsonProcessingException {

String response = null;

// create user registration object

RegistrationUser registrationUser = getRegistrationUser();

// convert the user registration object to JSON

String registrationBody = getBody(registrationUser);

// create headers specifying that it is JSON request

HttpHeaders registrationHeaders = getHeaders();

HttpEntity<String> registrationEntity = new HttpEntity<String>(registrationBody, registrationHeaders);

try {

// Register User

ResponseEntity<String> registrationResponse = restTemplate.exchange(REGISTRATION\_URL, HttpMethod.POST,

registrationEntity, String.class);

// if the registration is successful

if (registrationResponse.getStatusCode().equals(HttpStatus.OK)) {

// create user authentication object

User authenticationUser = getAuthenticationUser();

// convert the user authentication object to JSON

String authenticationBody = getBody(authenticationUser);

// create headers specifying that it is JSON request

HttpHeaders authenticationHeaders = getHeaders();

HttpEntity<String> authenticationEntity = new HttpEntity<String> (authenticationBody,authenticationHeaders);

// Authenticate User and get JWT

ResponseEntity<ResponseToken> authenticationResponse = restTemplate.exchange(AUTHENTICATION\_URL,

HttpMethod.POST, authenticationEntity, ResponseToken.class);

// if the authentication is successful

if (authenticationResponse.getStatusCode().equals(HttpStatus.OK)) {

String token = "Bearer " + authenticationResponse.getBody().getToken();

ttpHeaders headers = getHeaders();

headers.set("Authorization", token);

HttpEntity<String> jwtEntity = new HttpEntity<String>(headers);

// Use Token to get Response

ResponseEntity<String> helloResponse = restTemplate.exchange(HELLO\_URL, HttpMethod.GET, jwtEntity, String.class);

if (helloResponse.getStatusCode().equals(HttpStatus.OK)) {

response = helloResponse.getBody(); } } }

} catch (Exception ex) { System.out.println(ex); }

return response; }

private RegistrationUser getRegistrationUser() {

RegistrationUser user = new RegistrationUser();

user.setUsername("javainuse");

user.setPassword("javainuse");

user.setRole("ROLE\_ADMIN");

return user; }

private User getAuthenticationUser() {

User user = new User();

user.setUsername("javainuse");

user.setPassword("javainuse");

return user; }

private HttpHeaders getHeaders() {

HttpHeaders headers = new HttpHeaders();

headers.set("Content-Type", MediaType.APPLICATION\_JSON\_VALUE);

headers.set("Accept", MediaType.APPLICATION\_JSON\_VALUE);

return headers; }

private String getBody(final User user) throws JsonProcessingException {

return new ObjectMapper().writeValueAsString(user); } }

Start the Spring Boot + JWT + MySQL Project we had developed previously.

Start the Spring Boot Project we have developed in this tutorial. If we now go to <http://localhost:8081/getResponse>



Spring Boot Security Example - Refresh Expired JSON Web Token

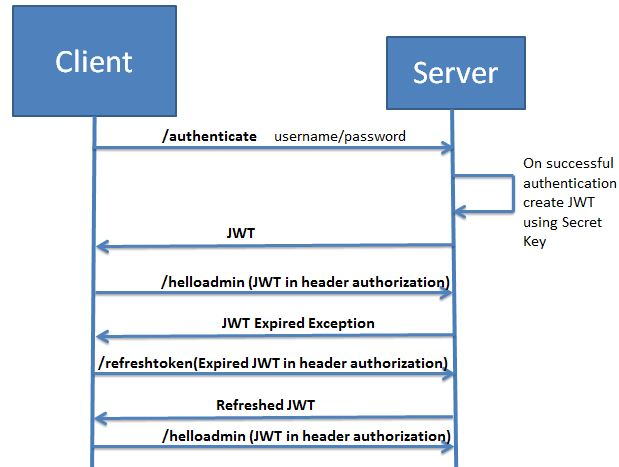
In previous tutorial we had implemented Spring Boot + JWT Example. We had also covered the topic of JWT Expiration. We had implemented the solution such that if the JWT has expired then the user gets JWTExpiredException.

Suppose our requirement is such that if the token has expired, still the user should be allowed to access the system if the token is valid. That is the token should be refreshed or a new valid token should be provided.

We will be working on a solution where if the user he receives JWT expired exception, then he can call another API with the expired token. A new token will then provided to the user which he can use for future interactions. Previously we had implemented an example for programmatically consuming the JWT secure API using Spring RestTemplate. We will be testing this refresh Token generation API both using Postman as well as the Spring RestTemplate.

Create and return new JWT token on Expiration

We will be modifying the Spring Boot + JWT + MySql example to implement Refresh JWT.



In application properties specify expiration time for the refresh token to be created. We use a seperate value for Refresh Token as we may want to specify different value to refresh token that the original JWT.

Also the expirationDateInMs we have specified as 0 because we want to test the expiration scenario.

**application.properties**

jwt.secret = javainuse

jwt.expirationDateInMs=0

jwt.refreshExpirationDateInMs=9000000

spring.datasource.url=jdbc:mysql://localhost/bootjwt?createDatabaseIfNotExist=true&autoReconnect=true&useSSL=false

spring.datasource.username=root

spring.datasource.password=root

spring.jpa.hibernate.ddl-auto=create-drop

In the JwtUtil class create a method named doGenerateRefreshToken to create the refresh token.

@Service

public class **JwtUtil** {

private String secret;

private int jwtExpirationInMs;

private int refreshExpirationDateInMs;

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

public void setSecret(String secret) { this.secret = secret; }

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

public void setJwtExpirationInMs(int jwtExpirationInMs) {

this.jwtExpirationInMs = jwtExpirationInMs; }

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

public void setRefreshExpirationDateInMs(int refreshExpirationDateInMs) {

this.refreshExpirationDateInMs = refreshExpirationDateInMs; }

public String generateToken(UserDetails userDetails) {

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

Collection<? extends GrantedAuthority> roles = userDetails.getAuthorities();

if (roles.contains(new SimpleGrantedAuthority("ROLE\_ADMIN"))) {

claims.put("isAdmin", true); }

if (roles.contains(new SimpleGrantedAuthority("ROLE\_USER"))) {

claims.put("isUser", true); }

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

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() + jwtExpirationInMs))

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

public String doGenerateRefreshToken(Map<String, Object> claims, String subject) {

return Jwts.builder().setClaims(claims).setSubject(subject).setIssuedAt(new Date(System.currentTimeMillis()))

.setExpiration(new Date(System.currentTimeMillis() + refreshExpirationDateInMs))

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

public boolean validateToken(String authToken) {

try { Jws<Claims> claims = Jwts.parser().setSigningKey(secret).parseClaimsJws(authToken);

return true;

} catch (SignatureException | MalformedJwtException | UnsupportedJwtException | IllegalArgumentException ex) {

throw new BadCredentialsException("INVALID\_CREDENTIALS", ex);

} catch (ExpiredJwtException ex) { throw ex; } }

public String getUsernameFromToken(String token) {

Claims claims = Jwts.parser().setSigningKey(secret).parseClaimsJws(token).getBody();

return claims.getSubject(); }

public List<SimpleGrantedAuthority> getRolesFromToken(String token) {

Claims claims = Jwts.parser().setSigningKey(secret).parseClaimsJws(token).getBody();

List<SimpleGrantedAuthority> roles = null;

Boolean isAdmin = claims.get("isAdmin", Boolean.class);

Boolean isUser = claims.get("isUser", Boolean.class);

if (isAdmin != null && isAdmin) {

roles = Arrays.asList(new SimpleGrantedAuthority("ROLE\_ADMIN"));

}

if (isUser != null && isAdmin) {

roles = Arrays.asList(new SimpleGrantedAuthority("ROLE\_USER"));

} return roles; }

Next we will be making changes in CustomJwtAuthenticationFilter class. If during JWT validation we get JWT Expiration Exception then we check -

If the HttpRequest header has the isRefreshToken set to true

If the HttpRequest url is refreshtoken. We do not want any other url to be allowed if the JWT has expired

If both the above conditions are true then we extract the claims from the ExpiredJwtException and store them as an attribute in the HttpRequest. These claims will be later used for Refresh JWT creation. Also we set the Security context by creating a UsernamePasswordAuthenticationToken with null values.

@Component

public class CustomJwtAuthenticationFilter extends OncePerRequestFilter {

@Autowired

private JwtUtil jwtTokenUtil;

@Override

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

throws ServletException, IOException {

try {

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

// get only the Token

String jwtToken = extractJwtFromRequest(request);

if (StringUtils.hasText(jwtToken) && jwtTokenUtil.validateToken(jwtToken)) {

UserDetails userDetails = new User(jwtTokenUtil.getUsernameFromToken(jwtToken), "", jwtTokenUtil.getRolesFromToken(jwtToken));

UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken = new UsernamePasswordAuthenticationToken(userDetails, null, userDetails.getAuthorities());

// 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);

} else {

System.out.println("Cannot set the Security Context");

}

} catch (ExpiredJwtException ex) {

String isRefreshToken = request.getHeader("isRefreshToken");

String requestURL = request.getRequestURL().toString();

// allow for Refresh Token creation if following conditions are true.

if (isRefreshToken != null && isRefreshToken.equals("true") && requestURL.contains("refreshtoken")) {

allowForRefreshToken(ex, request); } else

request.setAttribute("exception", ex);

} catch (BadCredentialsException ex) {

request.setAttribute("exception", ex);

} catch (Exception ex) { System.out.println(ex); }

chain.doFilter(request, response); }

private void allowForRefreshToken(ExpiredJwtException ex, HttpServletRequest request) {

// create a UsernamePasswordAuthenticationToken with null values.

UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken = new UsernamePasswordAuthenticationToken( null, null, null);

// 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);

// Set the claims so that in controller we will be using it to create

// new JWT

request.setAttribute("claims", ex.getClaims()); }

private String extractJwtFromRequest(HttpServletRequest request) {

String bearerToken = request.getHeader("Authorization");

if (StringUtils.hasText(bearerToken) && bearerToken.startsWith("Bearer ")) {

return bearerToken.substring(7, bearerToken.length());

} return null; } } }

Finally in the controller class expose the GET API for creating the refresh token -

@RestController

public class **AuthenticationController** {

@Autowired

private AuthenticationManager authenticationManager;

@Autowired

private CustomUserDetailsService userDetailsService;

@Autowired

private JwtUtil jwtUtil;

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

public ResponseEntity<?> createAuthenticationToken(@RequestBody AuthenticationRequest authenticationRequest)

throws Exception {

try {

authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(

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

} catch (DisabledException e) {

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

} catch (BadCredentialsException e) {

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

}

UserDetails userdetails = userDetailsService.loadUserByUsername(authenticationRequest.getUsername());

String token = jwtUtil.generateToken(userdetails);

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

}

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

public ResponseEntity<?> saveUser(@RequestBody UserDTO user) throws Exception {

return ResponseEntity.ok(userDetailsService.save(user));

}

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

public ResponseEntity<?> refreshtoken(HttpServletRequest request) throws Exception {

// From the HttpRequest get the claims

DefaultClaims claims = (io.jsonwebtoken.impl.DefaultClaims) request.getAttribute("claims");

Map<String, Object> expectedMap = getMapFromIoJsonwebtokenClaims(claims);

String token = jwtUtil.doGenerateRefreshToken(expectedMap, expectedMap.get("sub").toString());

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

}

public Map<String, Object> getMapFromIoJsonwebtokenClaims(DefaultClaims claims) {

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

for (Entry<String, Object> entry : claims.entrySet()) {

expectedMap.put(entry.getKey(), entry.getValue());

}

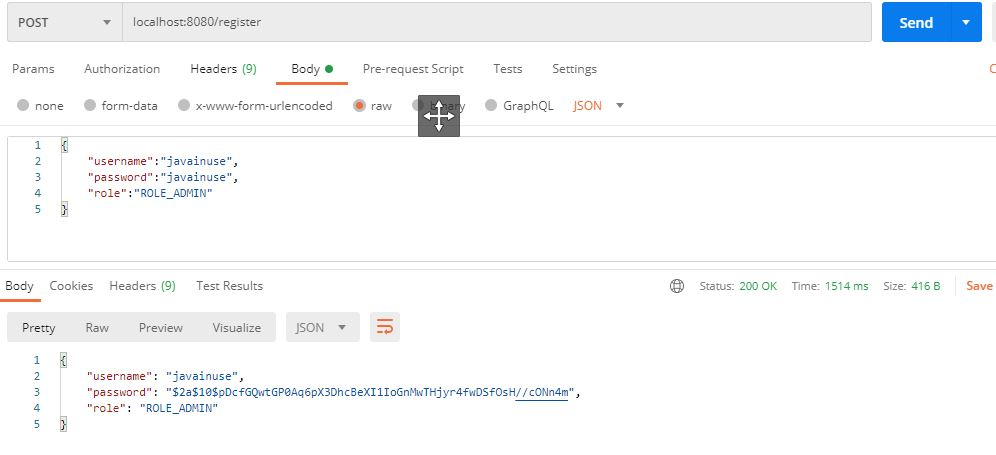
return expectedMap;

}

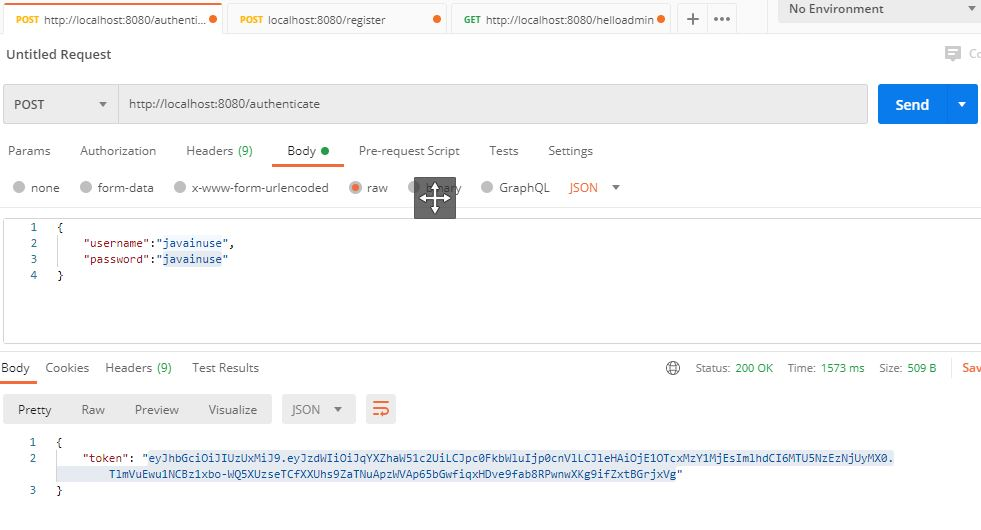
}

We are done with the changes.

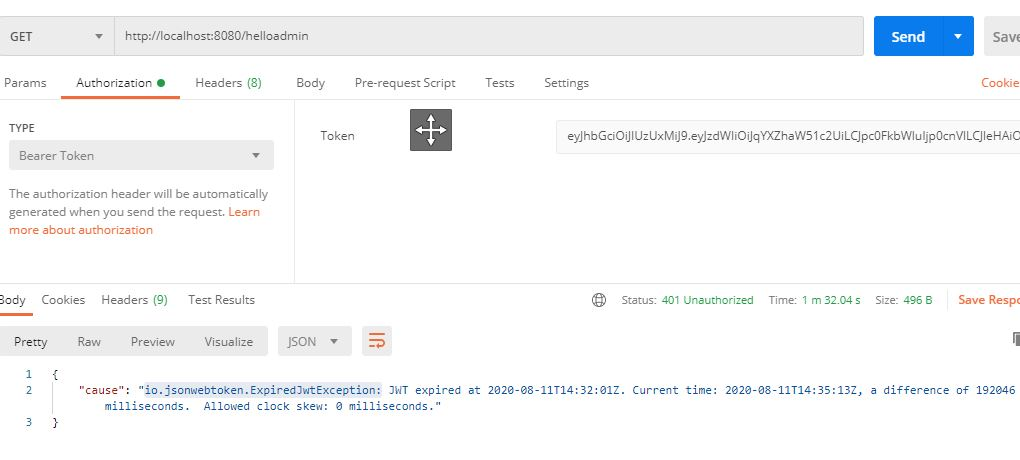
Register new User - localhost:8080/register



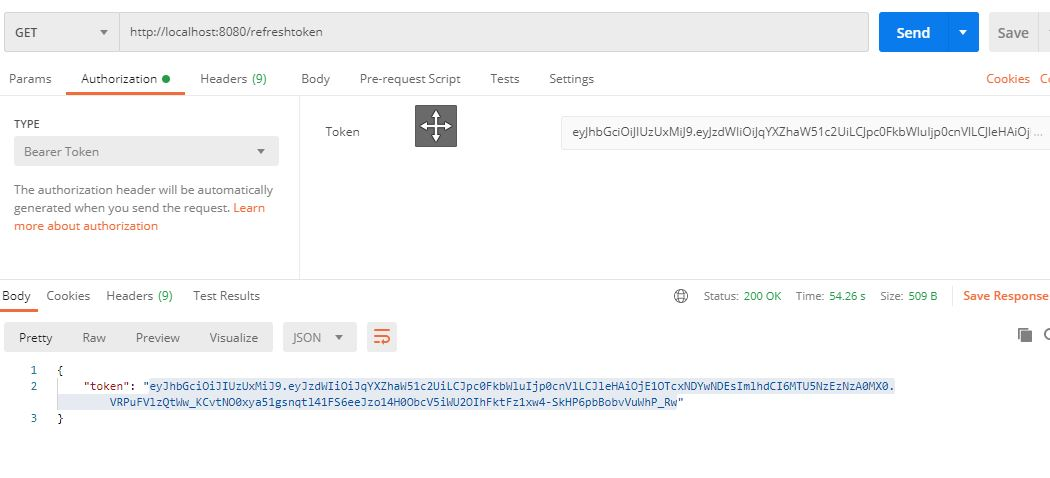
Authenticate User and get JWT - localhost:8080/authenticate



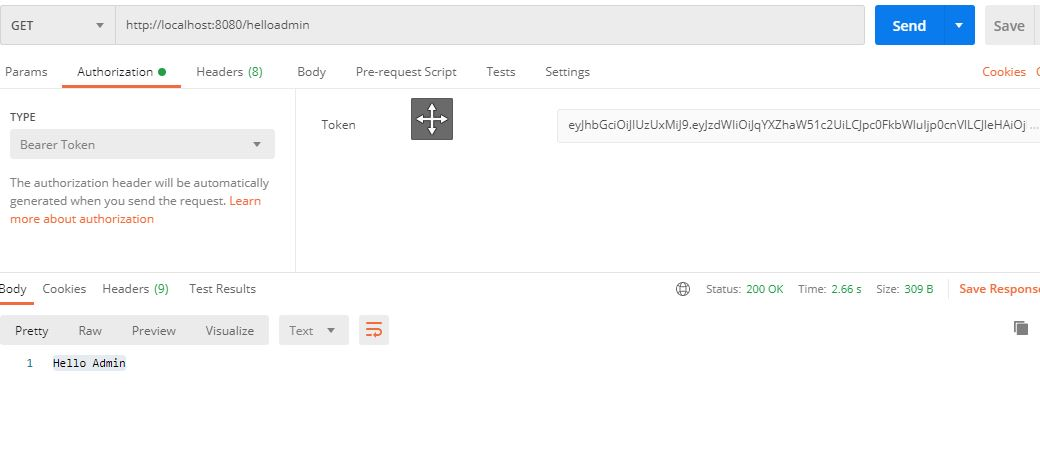
Try accessing the helloadmin url - localhost:8080/helloadmin with JWT. We get Expired JWT Exception



Generate Refresh Token - localhost:8080/refreshtoken



Use the new Token to successfully get data from helloadmin - localhost:8080/helloadmin



Test Refresh Token with Spring Boot RestTemplate

Previously we have implemented Spring Boot RestTemplate + JWT example. We will be modifying the code to test the refresh token scenario.

Modify the TestController class. If we get the Expired JWT Exception, we will be creating a new refresh JWT and using it to get the data.