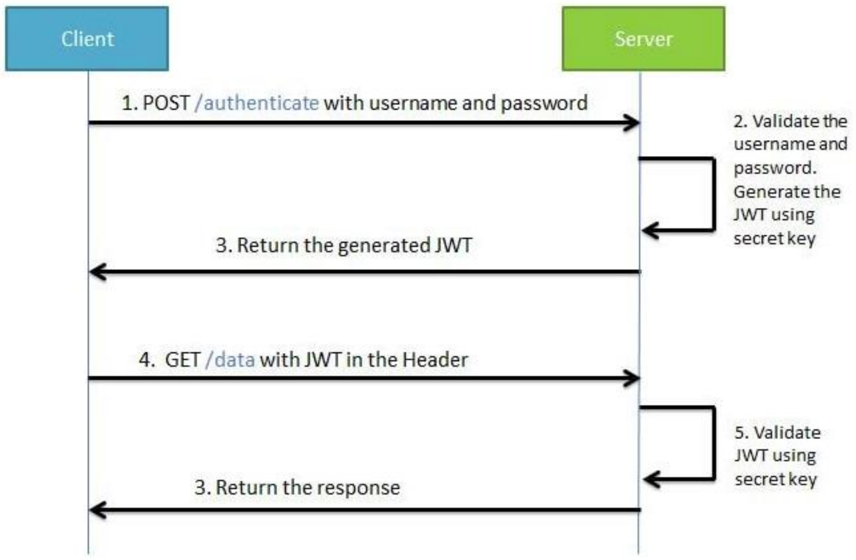
Spring Security with JWT

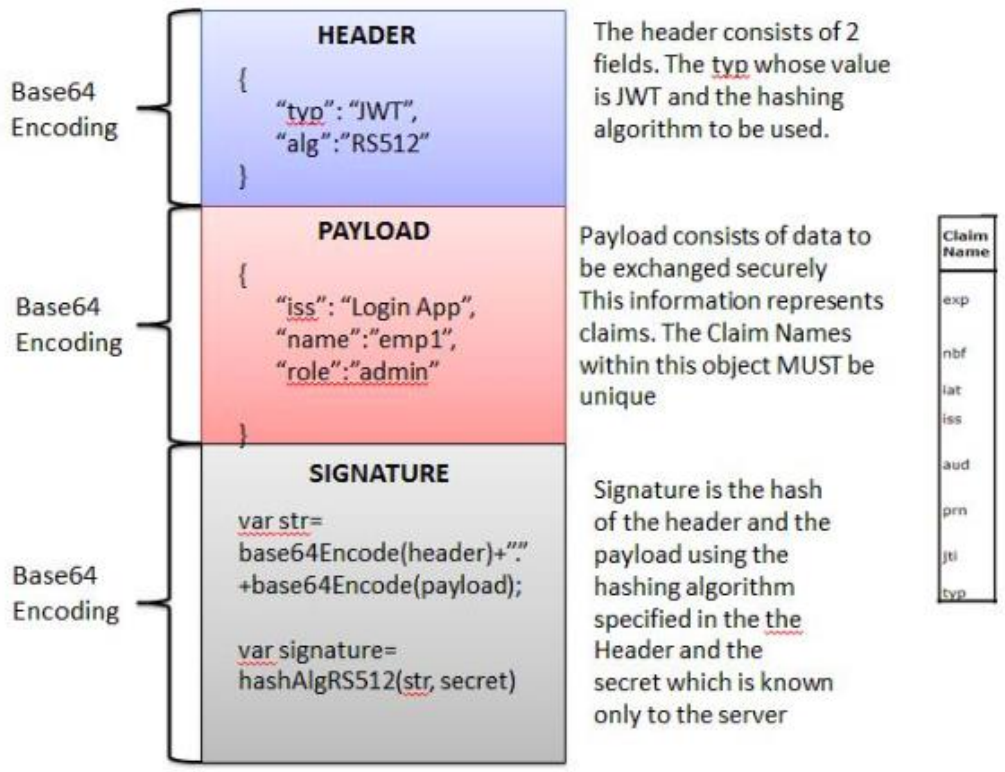
What is **JWT ?**

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.


Client
Server
1. POST /authenticate with username and password
2. Validate the username and password. Generate the JWT using secret key
3. Return the generated JWT
4. GET /data with JWT in the Header
5. Validate JWT using secret key
3. Return the response

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-


HEADER
{
"typ": "JWT",
"alg":"RS512"
}
PAYLOAD
{
"iss": "Login App",
"name":"emp1",
"role":"admin"
}
SIGNATURE
var str=
base64Encode(header)+"."
+base64Encode(payload);
var signature=
hashAlgRS512(str, secret)
Base64 Encoding
Base64 Encoding
Base64 Encoding
The header consists of 2 fields. The typ whose value is JWT and the hashing algorithm to be used.
Payload consists of data to be exchanged securely This information represents claims. The Claim Names within this object MUST be unique
Signature is the hash of the header and the payload using the hashing algorithm specified in the the Header and the secret which is known only to the server
Claim Name
exp
nbf
iat
iss
aud
prn
jti
typ

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.

Steps –

1. Add jwt dependency in POM.xml

```
<dependency>
          <groupId>io.jsonwebtoken</groupId>
          <artifactId>jjwt</artifactId>
          <version>0.9.1</version>
</dependency>
```

2. Create JWTUtil Class

   Create the JWTUtil class. As the name suggest this class will have utility methods corresponding to the JWT like create JWT, check if JWT is valid.

```java
package com.example.springsecurityDatabase.Model;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Service;

import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import java.util.Objects;
import java.util.concurrent.ConcurrentHashMap;
import java.util.function.Function;

@Service
public class JwtManager
{
    private static String SECRET_KEY = "secret";

    public String generateToken(UserDetails userDetails)
    {
        Map<String, Object> claims = new HashMap<>();
        return createToken(claims,userDetails.getUsername());
    }

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

        return Jwts.builder()
                .setClaims(claims)
                .setSubject(subject)
                .setIssuedAt(new Date(System.currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60 * 10 ))
                .signWith(SignatureAlgorithm.HS256 , SECRET_KEY)
                .compact();
    }
}
```

- **Secret Key** - In Understanding JWT Structure Tutorial we had created the signature for the JWT using HS512Algo(Header + Payload + Secret Key).
  The JWT has this signature present. If any user intercepts and tampers with the payload, then he cannot change the signature as he does not have the secret key. So the server which has the secret key will know that the JWT has been tampered with as the signature created by it using the secret key will not match the secret key in the JWT.
  Our secret key value will be "secret".
- **Expiration Time** - This the time for which we want the generated JWT to be valid for. This will be in milliseconds. Suppose we want the JWT to be valid for 5 hours, then we will specify this value as 18000000

3. Create the AuthenticationRequest class. The JSON input request provided by the user will be unmarshalled to Java Object using this class.

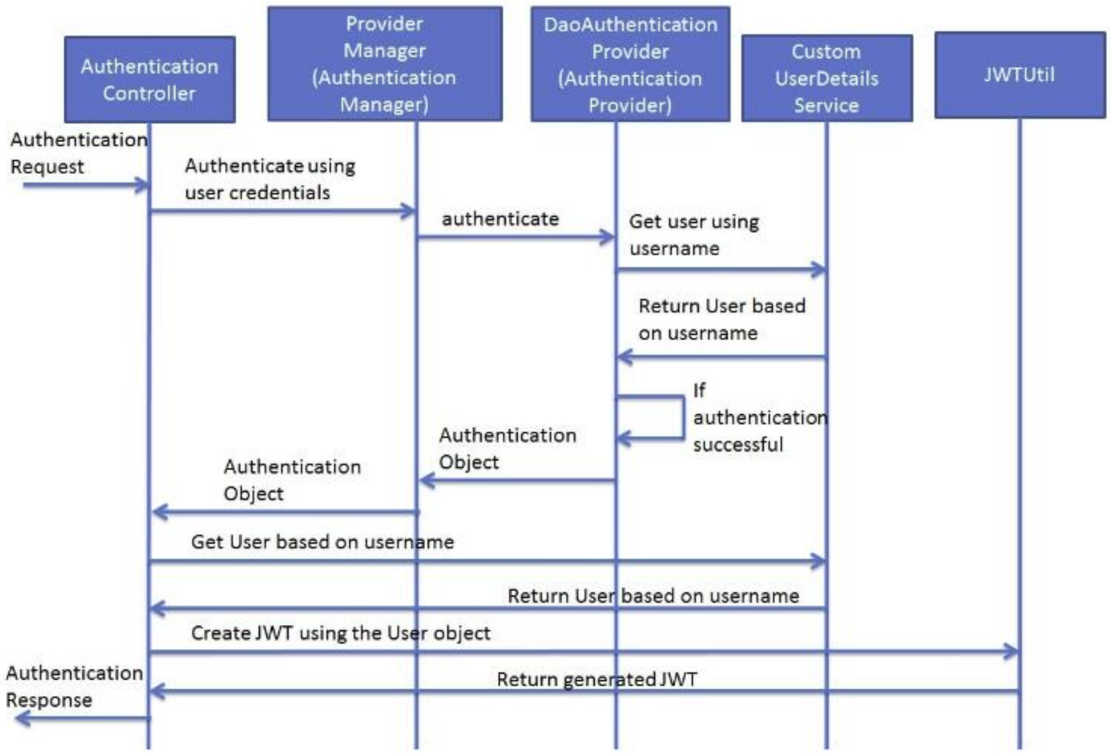
```java
package com.example.springsecurityDatabase.Model;

public class AuthenticationRequest {

    private String username;
    private String password;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

4. Create the AuthenticationResponse class which is a model class to return the token on successful authentication.

```java
package com.example.springsecurityDatabase.Model;

public class AuthenticationResponse {
    private String jwt;

    public AuthenticationResponse() {
    }
    public AuthenticationResponse(String jwt) {
        this.jwt = jwt;
    }

    public String getJwt() {
        return jwt;
    }

    public void setJwt(String jwt) {
        this.jwt = jwt;
    }
}
```

5. Next create a AuthenticationController class which will expose a POST REST API to take username and password from the user and if user is authenticated then return a JSON Web Token. For Authentication we will be making use of the AuthenticationManager which we have already configured in the SecurityConfiguration. This flow is quite similar to the previous Spring Boot Security Project where we has seen the Spring Boot Security Architecture and the Authentication Manager authenticates the incoming HTTP request. Once the authentication is successful we will be making a call to the generateToken method of the JwtUtil class which will create the token. This token will be returned back to the user.

```java
package com.example.springsecurityDatabase.Controller;

@RestController
@RequestMapping("/secure/auth")

public class AdminController {
    @Autowired
    AuthenticationManager authenticationManager;
    @Autowired
    private UserRepository userRepository;
    @Autowired
    private BCryptPasswordEncoder bCryptPasswordEncoder;
    @Autowired
    private CustomUserDetailService userDetailsService;
   @Autowired
    private JwtManager jwtManager;

    @PostMapping("/admin/login")
    public ResponseEntity login(@RequestBody LoginCred user) throws Exception {
        try {
            Authentication authentication = authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(user.getUsername(), user.getPassword())
            );
        }
        catch (BadCredentialsException ex)
        {      throw new Exception("Username or password invalid " + ex);
        }
        final UserDetails userDetails = userDetailsService
                .loadUserByUsername(user.getUsername());
        final  String jwt = jwtManager.generateToken(userDetails);
        return ResponseEntity.ok(new AuthenticationResponse(jwt));
    }
```


Authentication Controller
Provider Manager (Authentication Manager)
DaoAuthentication Provider (Authentication Provider)
Custom UserDetails Service
JWTUtil
Authentication Request
Authenticate using user credentials
authenticate
Get user using username
Return User based on username
If authentication successful
Authentication Object
Authentication Object
Get User based on username
Return User based on username
Create JWT using the User object
Return generated JWT
Authentication Response

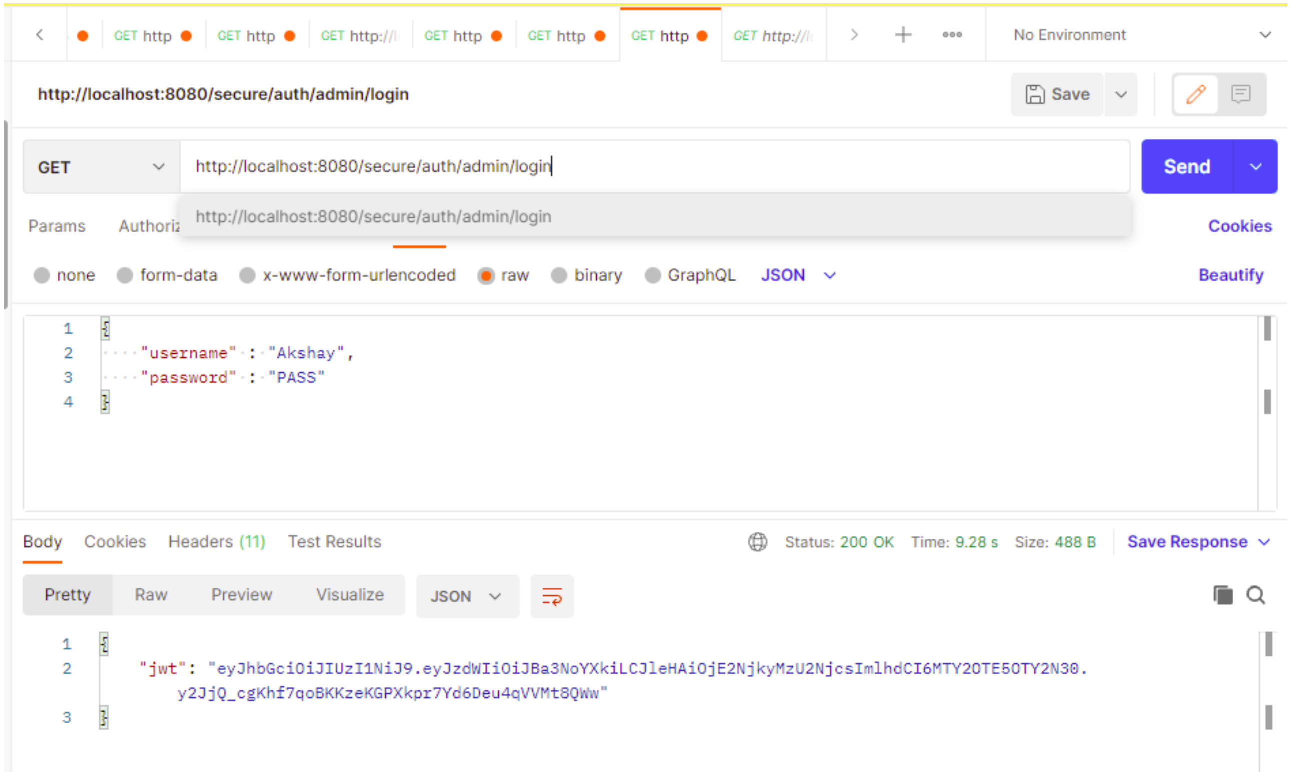
6. Finally in the Spring Security Configuration

- Allow the /admin/login url without any authentication i.e permitAll().
- We have already configured the Authentication Manager using the AuthenticationManagerBuilder. But Spring Security needs us to explicitly create the AuthenticationManager Bean.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.csrf().disable();
    http.authorizeRequests()
            .antMatchers("/rest/**")
            .authenticated()
            .anyRequest()
            .permitAll()
            .and().exceptionHandling()
            .authenticationEntryPoint(entryPointException)
           .and().sessionManagement()
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS);

    http.addFilterBefore(jwtRequestFilter
UsernamePasswordAuthenticationFilter.class);
}
```

```
@Override@Bean
public AuthenticationManager authenticationManagerBean () throws Exception
{
    return super.authenticationManagerBean();
}
```

7. Start the application and test the POST request /authenticate url-



http://localhost:8080/secure/auth/admin/login

GET http://localhost:8080/secure/auth/admin/login

```
{
    "username" : "Akshay",
    "password" : "PASS"
}
```

Status: 200 OK Time: 9.28 s Size: 488 B

```
{
    "jwt": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJBa3NoYXkiLCJleHAiOjE2NjkyMzU2NjcsImlhdCI6MTY2OTE5OTY2N30.
        y2JjQ_cgKhf7qoBKKzeKGPXkpr7Yd6Deu4qVVMt8QWw"
}
```

8. Validation of JWT Token

We will now be using the generated JWT to authorize user to perform operations. In previous tutorial we have seen that any incoming request is first intercepted by Filters which perform authentication and authorization. One example of such filter is the **BasicAuthenticationFilter** which is a type of **OncePerRequestFilter**. We will now be writing our own **CustomJWTAuthenticationFilter** which will also be type of **OncePerRequestFilter**.

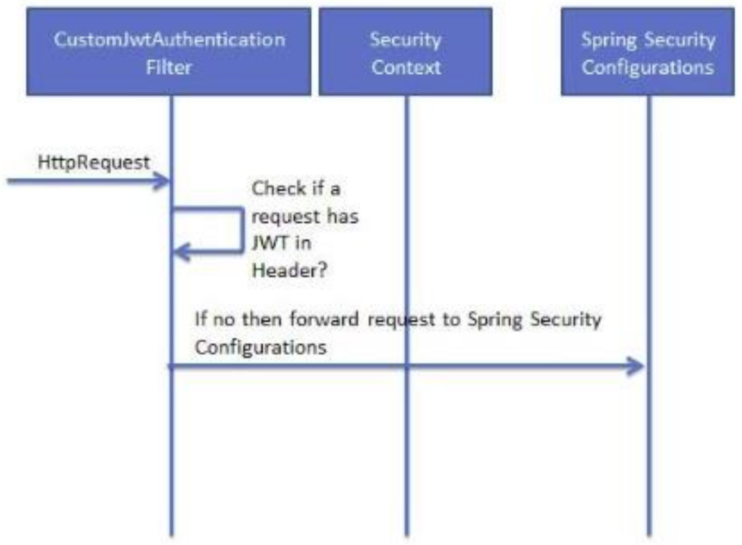
This **CustomJWTAuthenticationFilter** will intercept the incoming request and check if it contains a JSON Web Token(JWT). If JWT is present it will then call the validate method of the JWTUtil class to validate the token. If the validation is successful, then it will create a User Object using the JWT payload and store this object in the Security context, which indicates that the current user is authenticated.

9. We will first be adding the some more utility methods like validateToken to the JWTUtil class .

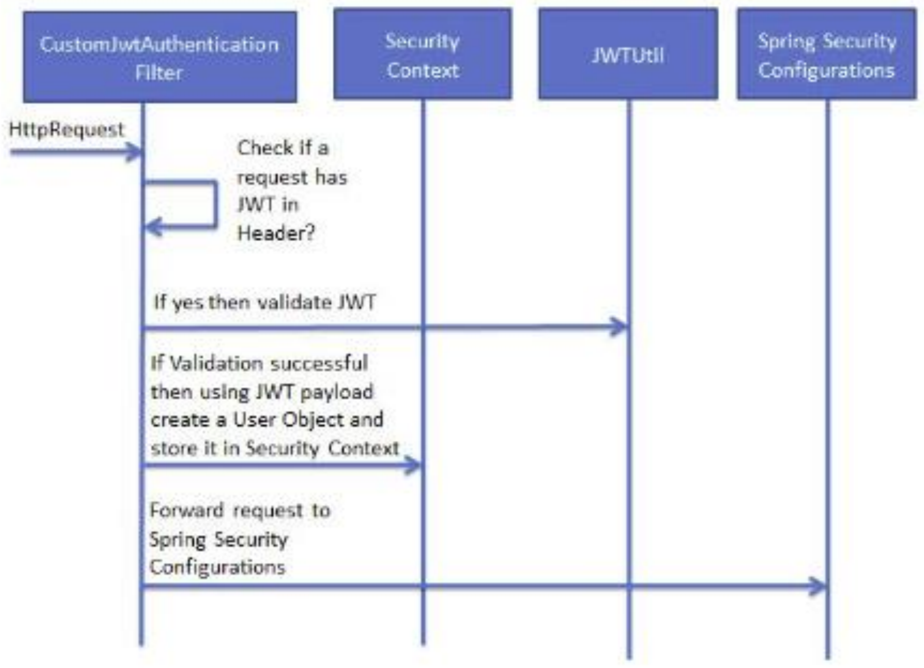
```
public String extractUsername(String token)
{
    return extractClaim(token , Claims::getSubject);
}

public Date extractExpiration(String token)
{
    return extractClaim(token , Claims::getExpiration);
}

public <T> T extractClaim(String token , Function<Claims,T> claimsResolver)
{
    final Claims claims = extractAllClaims(token);
    return claimsResolver.apply(claims);

}

private Claims extractAllClaims(String token)
{
    return
Jwts.parser().setSigningKey(SECRET_KEY).parseClaimsJws(token).getBody();
}

private boolean isTokenExpired(String token)
{
    return extractExpiration(token).before(new Date());
}
public boolean validateToken(String token,UserDetails userDetails)
{
    final String userName = extractUsername(token);
    return (userName.equals(userDetails.getUsername()) &&
!isTokenExpired(token));

}
```

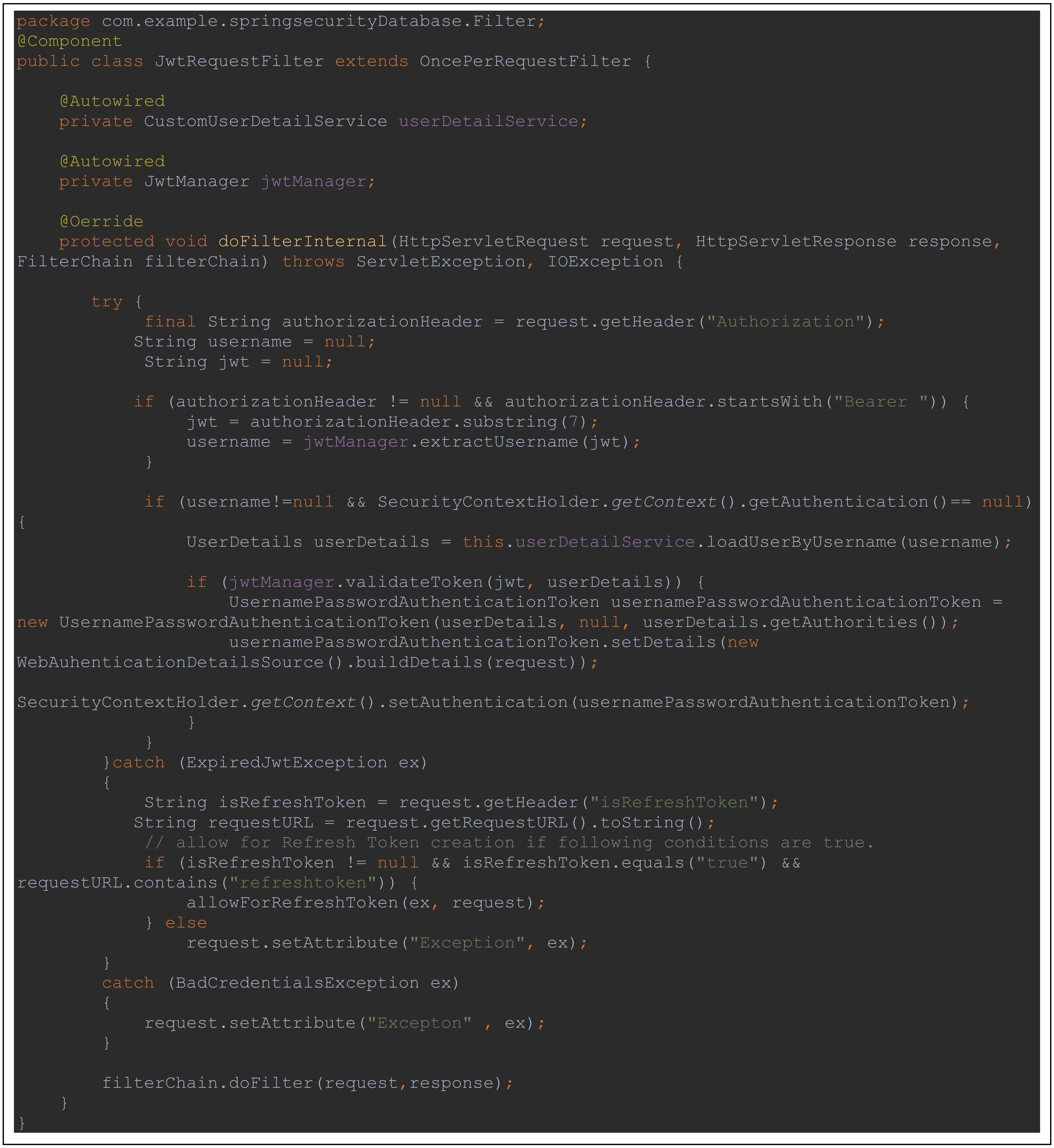
10. Create the **CustomJwtAuthenticationFilter** class will be type of **OncePerRequestFilter**. This class will intercept the requests and

- Check if header contains a JWT. If no then let the usual Spring Security Flow take place.


CustomJwtAuthentication Filter
Security Context
Spring Security Configurations
HttpRequest
Check if a request has JWT in Header?
If no then forward request to Spring Security Configurations

- If the header contains a JWT then it will validate the token. On successful validation it will add the User Object in the Spring Context to indicate that the user can be authorized to perform operation.


CustomJwtAuthentication Filter
Security Context
JWTUtil
Spring Security Configurations
HttpRequest
Check if a request has JWT in Header?
If yes then validate JWT
If Validation successful then using JWT payload create a User Object and store it in Security Context
Forward request to Spring Security Configurations



```
package com.example.springsecurityDatabase.Filter;
@Component
public class JwtRequestFilter extends OncePerRequestFilter {

    @Autowired
    private CustomUserDetailService userDetailService;

    @Autowired
    private JwtManager jwtManager;

    @Oerride
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
FilterChain filterChain) throws ServletException, IOException {

        try {
             final String authorizationHeader = request.getHeader("Authorization");
            String username = null;
             String jwt = null;

            if (authorizationHeader != null && authorizationHeader.startsWith("Bearer ")) {
                jwt = authorizationHeader.substring(7);
                username = jwtManager.extractUsername(jwt);
             }

             if (username!=null && SecurityContextHolder.getContext().getAuthentication()== null)
{
                UserDetails userDetails = this.userDetailService.loadUserByUsername(username);

                if (jwtManager.validateToken(jwt, userDetails)) {
                    UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken =
new UsernamePasswordAuthenticationToken(userDetails, null, userDetails.getAuthorities());
                    usernamePasswordAuthenticationToken.setDetails(new
WebAuhenticationDetailsSource().buildDetails(request));

SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticationToken);
                }
             }
        }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("Excepton" , ex);
        }

        filterChain.doFilter(request,response);
    }
}
```

11. Create JwtAuthenticationEntryPoint class. This class is used to return a 401 unauthorized error to clients that try to access a protected resource without proper authentication. It implements Spring Security AuthenticationEntryPoint interface. In this class we will be creating the HttpResponse which should be returned to the user in case of an exception.

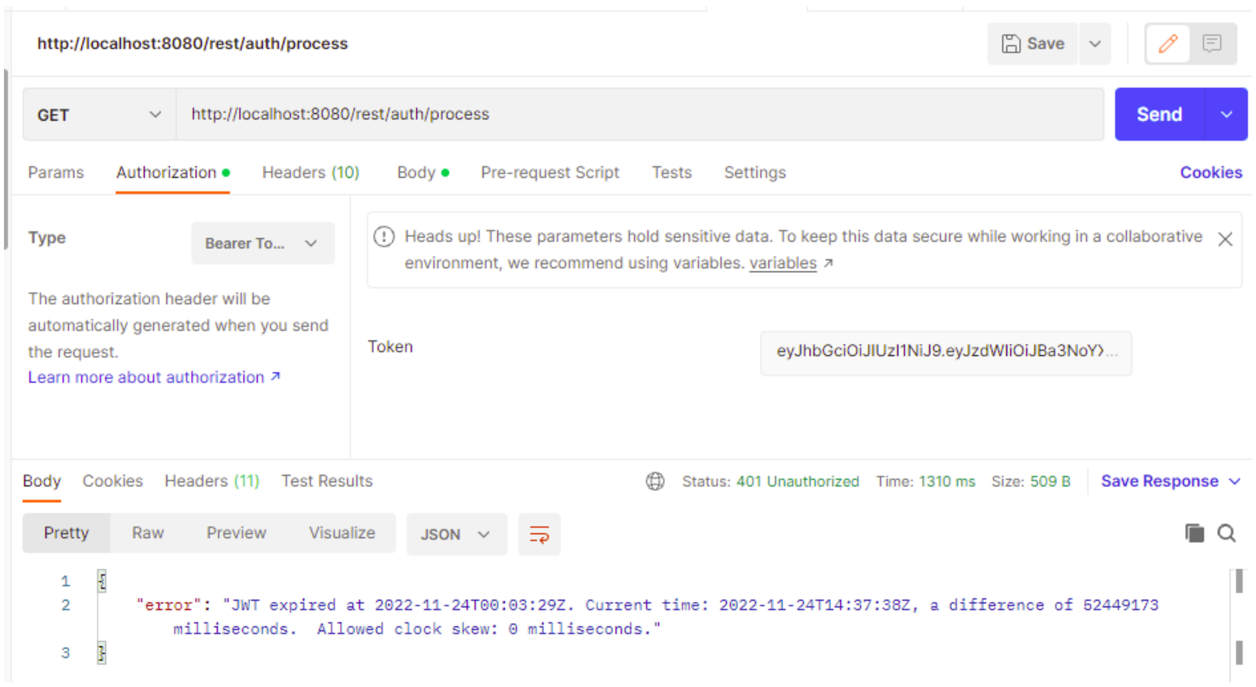
```java
package com.example.springsecurityDatabase.EntryPoint;

@Component
public class EntryPointException implements AuthenticationEntryPoint {
    @Override
    public void commence(HttpServletRequest request, HttpServletResponse response,
AuthenticationException authException) throws IOException, ServletException {


        Exception exception = (Exception) request.getAttribute("Exception");


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

        String msg;


        if (exception != null)
        {
            if(exception.getCause()!=null)
            {
                msg = exception.getCause().toString()+" "+exception.getMessage();
            }
            else
            {
                msg=exception.getMessage();
            }

            byte[] body = new
ObjectMapper().writeValueAsBytes(Collections.singletonMap("error" , msg));
            response.getOutputStream().write(body);
        }
        else
        {
            if(authException.getCause()!=null)
            {
                msg = authException.getCause().toString()+" "+authException.getMessage();
            }
            else
            {
                msg=authException.getMessage();
            }

            byte[] body = new
ObjectMapper().writeValueAsBytes(Collections.singletonMap("error" , msg));
            response.getOutputStream().write(body);
        }
    }
}
```

12. Modify Security Configuration-

- Remove HttpBasic Security, as we will be using JWT for authorization.
- Configure Http Security to make use of the CustomJwtAuthenticationFilter and the JwtAuthenticationEntryPoint.

```
@Autowired
EntryPointException entryPointException;

@Resource
CustomAuthenticationProvider customAuthenticationProvider;


@Override
protected void configure(HttpSecurity http) throws Exception {
    http.csrf().disable();
    http.authorizeRequests()
            .antMatchers("/rest/**")
            .authenticated()
            .anyRequest()
            .permitAll()
   // If Any exception occurs call this
            .and().exceptionHandling().authenticationEntryPoint(entryPointException)
   // make sure we use stateless session; session won't be used to
   // store user's state.
           .and().sessionManagement()
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS);

   // Add Filter to validate token with every request

    http.addFilterBefore(jwtRequestFilter , UsernamePasswordAuthenticationFilter.class);
}
```

13. If we now run the application and use an expired JWT we get the following custom exception –



http://localhost:8080/rest/auth/process

GET http://localhost:8080/rest/auth/process

Type: Bearer To...

The authorization header will be automatically generated when you send the request. Learn more about authorization ↗

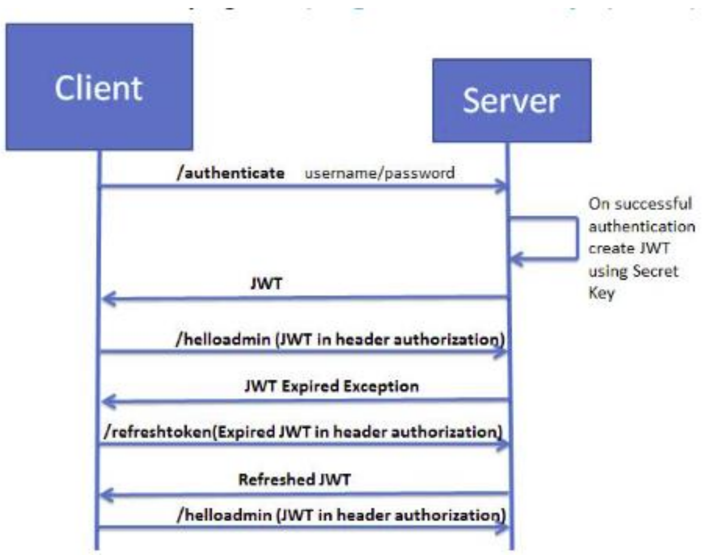
Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. variables ↗

Token: eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJBa3NoY...

Status: 401 Unauthorized Time: 1310 ms Size: 509 B

```
{
    "error": "JWT expired at 2022-11-24T00:03:29Z. Current time: 2022-11-24T14:37:38Z, a difference of 52449173
        milliseconds.  Allowed clock skew: 0 milliseconds."
}
```

14. REFRESH TOKENS
    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.


Client
Server
/authenticate username/password
On successful authentication create JWT using Secret Key
JWT
/helloadmin (JWT in header authorization)
JWT Expired Exception
/refreshtoken(Expired JWT in header authorization)
Refreshed JWT
/helloadmin (JWT in header authorization)

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

```
public static String doGenerateRefreshTokens(Map<String, Object> claims, String subject)
{
    return Jwts.builder()
            .setClaims(claims)
            .setSubject(subject)
            .setIssuedAt(new Date(System.currentTimeMillis()))
            .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60 * 10 ))
            .signWith(SignatureAlgorithm.HS256 , SECRET_KEY)
            .compact();
}
```

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

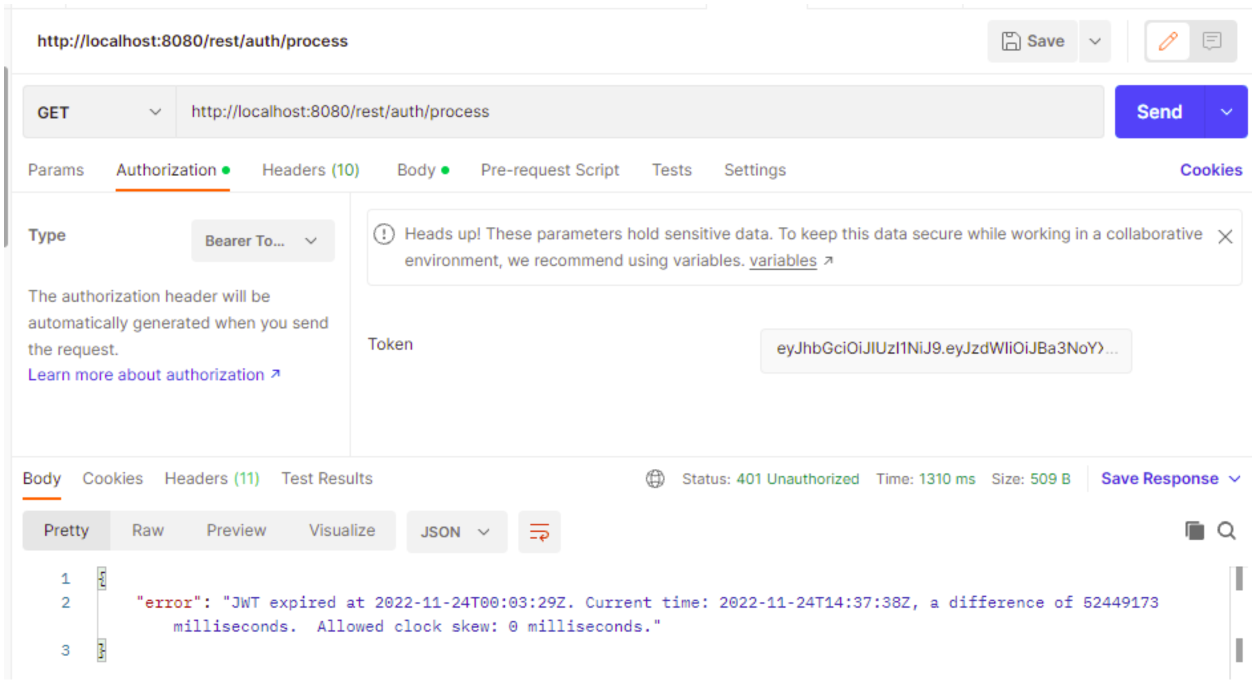
- If the HttpRequest header has the isRefreshToken set to true
- If the HttpRequest url is refreshtoken. We do not want any other url to be allowed if the JWT has expired
- If both the above conditions are true then we extract the claims from the ExpiredJwtException and store them as an attribute in 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.

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

    try {
        final String authorizationHeader = request.getHeader("Authorization");

        String username = null;
        String jwt = null;


        if (authorizationHeader != null && authorizationHeader.startsWith("Bearer ")) {
            jwt = authorizationHeader.substring(7);
            username = jwtManager.extractUsername(jwt);
        }

        if (username != null && SecurityContextHolder.getContext().getAuthentication() ==
null) {
            UserDetails userDetails = this.userDetailService.loadUserByUsername(username);

            if (jwtManager.validateToken(jwt, userDetails)) {
                UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken = new
UsernamePasswordAuthenticationToken(userDetails, null, userDetails.getAuthorities());
                usernamePasswordAuthenticationToken.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));

SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticationToken);
            }
        }
    }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("Excepton" , ex);
    }

    filterChain.doFilter(request,response);
}
```

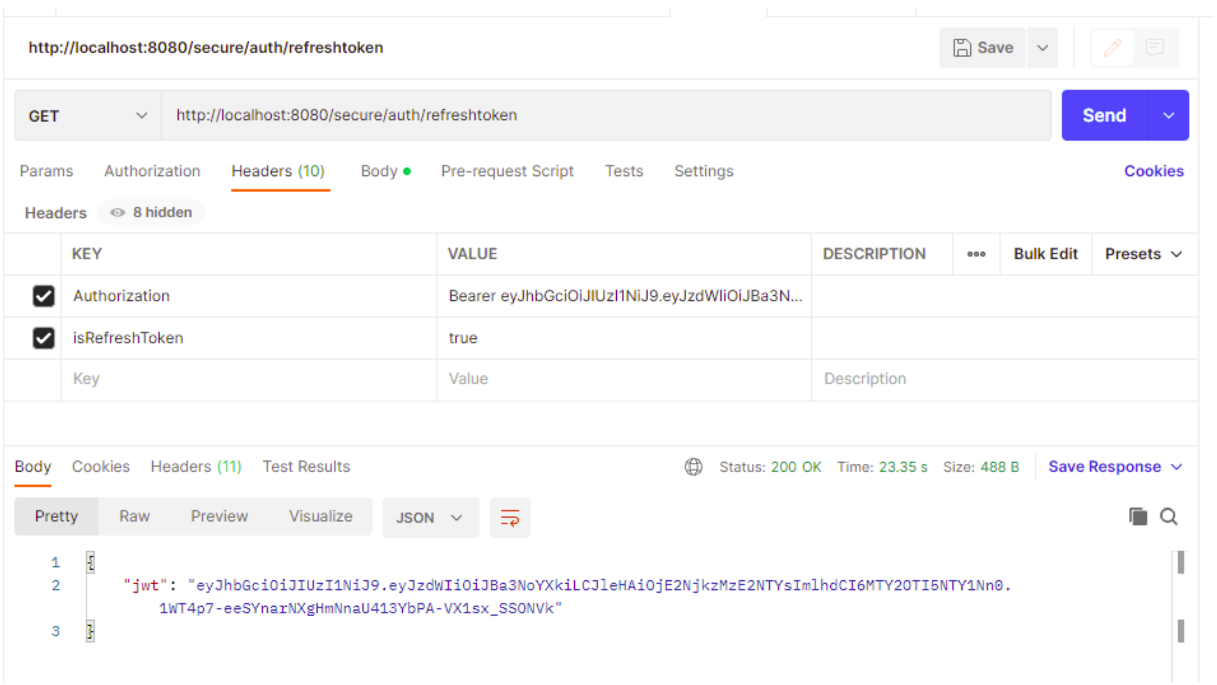
```
private void allowForRefreshToken(ExpiredJwtException ex, HttpServletRequest request) {

    Set<Integer> ss = new HashSet<>();

    // 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());
}
```

17. Finally in the controller class expose the GET API for creating the refresh token –

```
@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 = JwtManager.doGenerateRefreshTokens(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 (Map.Entry<String, Object> entry : claims.entrySet()) {
        expectedMap.put(entry.getKey(), entry.getValue());
    }
    return expectedMap;
}
```


http://localhost:8080/rest/auth/process
Save
GET
Send
Params
Authorization
Headers (10)
Body
Pre-request Script
Tests
Settings
Cookies
Type
Bearer To...
The authorization header will be automatically generated when you send the request.
Learn more about authorization ↗
Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. variables ↗
Token
eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJBa3NoY...
Body
Cookies
Headers (11)
Test Results
Status: 401 Unauthorized
Time: 1310 ms
Size: 509 B
Save Response
Pretty
Raw
Preview
Visualize
JSON

```
{
    "error": "JWT expired at 2022-11-24T00:03:29Z. Current time: 2022-11-24T14:37:38Z, a difference of 52449173
        milliseconds.  Allowed clock skew: 0 milliseconds."
}
```



http://localhost:8080/secure/auth/refreshtoken

GET http://localhost:8080/secure/auth/refreshtoken

	KEY	VALUE	DESCRIPTION
✓	Authorization	Bearer eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJBa3N...	
✓	isRefreshToken	true	
	Key	Value	Description

Status: 200 OK Time: 23.35 s Size: 488 B

```
{
    "jwt": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJBa3NoYXkiLCJleHAiOjE2NjkzMzE2NTYsImlhdCI6MTY2OTI5NTY1Nn0.
      1WT4p7-eeSYnarNXgHmNnaU413YbPA-VX1sx_SSONVk"
}
```