**MyAuthencticationController**

@RestController  
public class MyAuthencticationController {  
  
 */\*\*  
 \* In older version it worked like this  
 \* But in new version of Spring boot its not backward compatible  
 \* so, we need to create its Bean otherwise it will not be Autowired  
 \*/* @Autowired  
 private AuthenticationManager authenticationManager;  
  
 @Autowired  
 private MyUserDetailsService userDetailsService;  
  
 @Autowired  
 private JwtUtil jwtUtil;  
  
 */\*\*  
 \* We can not call this /authenticate directly as  
 \* spring security will automatically secured it  
 \* and putting authentication in every endpoint  
 \*/*@PostMapping("/authenticate")  
 public ResponseEntity<?> authenticate(@RequestBody UserCredentials userCredentials) throws Exception {  
  
 String username = userCredentials.getUsername();  
 String password = userCredentials.getPassword();  
 */\*\*  
 \* 1. Authenticate  
 \*/* try {  
 authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(username, password));  
 } catch (BadCredentialsException ex) {  
 throw new Exception("Incorrect username/Password");  
 }  
  
 */\*\*  
 \* 2. UserDetails from DB or in-memory  
 \*/* UserDetails userDetails = userDetailsService.loadUserByUsername(username);  
  
 */\*\*  
 \* 3. Get Token  
 \*/* String token = jwtUtil.generateToken(userDetails);  
  
 return ResponseEntity.*ok*(new ResponseToken(token));  
 }  
}

Above is a rest controller that exposes “/authenticate” which is responsible for generating jwt token. As we can’t directly access as spring security secures all the rest-endpoint so we have to allow all the users to access this url to get JWT token. So, we have to Configure this to allow all.

@Data

@AllArgsConstructor  
public class UserCredentials {  
 private String username;  
 private String password;  
}

@Data  
@AllArgsConstructor  
public class ResponseToken {  
 private String token;  
}

@Configuration  
@EnableWebSecurity  
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {  
  
 @Autowired  
 private MyUserDetailsService myUserDetailsService;  
 *// Setting my UserDetailsService* @Override  
 protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.userDetailsService(myUserDetailsService);  
 }  
  
  *// This method is related to http rest-endpoints* @Override  
 protected void configure(HttpSecurity http) throws Exception {  
 // disables cross site request forgery  
 http.csrf().disable()  
 // authorizes requests for /authenticate and permit everybody to access it  
 .authorizeRequests().antMatchers("/authenticate").permitAll()  
 // any other request needs to be authenticated  
 .anyRequest().authenticated();  
 }  
  
  *// Authentication Manager* @Override  
 @Bean  
 public AuthenticationManager authenticationManagerBean() throws Exception {  
 return super.authenticationManagerBean();  
 }

*// In case you DataBase have encrypted password you have to use Bycrypt* @Bean  
 public PasswordEncoder passwordEncoderBean() {  
 return NoOpPasswordEncoder.*getInstance*();  
 }  
}

Above is a configuration class that has override configure(http) method under which we can configure endpoint that we want to expose to the users or not.

1. **http.csrf().disable()** : This disable the cross site request forgery
2. **authorizeRequests().antMatchers("/authenticate").permitAll()**:this authorizes /authenticate to be permited for everybody
3. **.anyRequest().authenticated():** Any other request other than “/authenicate” must be allowed only after being authenticated.

@Service  
public class MyUserDetailsService implements UserDetailsService {  
  
 @Override  
 public UserDetails loadUserByUsername(String s) throws UsernameNotFoundException {  
 return new User("foo", "foo", new ArrayList<>());  
 }  
}

Above, sets Username and password of one user to foo and foo respectively. So now when the user hits /authenicate with body:

{

"username": "foo",

"password": "foo"

}

A jwt token is generated.

Util class for doing certain necessary logics.

@Service  
public class JwtUtil {  
  
 private String SECRET\_KEY = "secret";  
  
 private Clock clock = DefaultClock.*INSTANCE*;  
 private long expiration = 604800;  
  
  
 public String getUsernameFromToken(String token) {  
 return getClaimFromToken(token, Claims::getSubject);  
 }  
  
 public <T> T getClaimFromToken(String token, Function<Claims, T> claimsResolver) {  
 final Claims claims = getAllClaimsFromToken(token);  
 return claimsResolver.apply(claims);  
 }  
  
 public Date getIssuedAtDateFromToken(String token) {  
 return getClaimFromToken(token, Claims::getIssuedAt);  
 }  
  
 public Date getExpirationDateFromToken(String token) {  
 return getClaimFromToken(token, Claims::getExpiration);  
 }  
  
 private Claims getAllClaimsFromToken(String token) {  
 return Jwts.*parser*().setSigningKey(SECRET\_KEY).parseClaimsJws(token).getBody();  
 }  
  
 private Boolean isTokenExpired(String token) {  
 final Date expiration = getExpirationDateFromToken(token);  
 return expiration.before(clock.now());  
 }  
  
 private Boolean ignoreTokenExpiration(String token) {  
 // here you specify tokens, for that the expiration is ignored  
 return false;  
 }  
  
 public String generateToken(UserDetails userDetails) {  
  
 return Jwts.*builder*()  
 .setClaims(new HashMap<>())  
 .setSubject(userDetails.getUsername())  
 .setIssuedAt(new Date(System.*currentTimeMillis*()))  
 .setExpiration(new Date(System.*currentTimeMillis*() + 1000 \* 60 \* 60 \* 10))  
 .signWith(SignatureAlgorithm.*HS512*, SECRET\_KEY).compact();  
 }  
  
  
 public Boolean canTokenBeRefreshed(String token) {  
 return (!isTokenExpired(token) || ignoreTokenExpiration(token));  
 }  
  
 public String refreshToken(String token) {  
 final Date createdDate = clock.now();  
 final Date expirationDate = new Date(createdDate.getTime() + expiration \* 1000);  
  
 final Claims claims = getAllClaimsFromToken(token);  
 claims.setIssuedAt(createdDate);  
 claims.setExpiration(expirationDate);  
  
 return Jwts.*builder*().setClaims(claims).signWith(SignatureAlgorithm.*HS512*, SECRET\_KEY).compact();  
 }  
  
 public Boolean validateToken(String token, UserDetails userDetails) {  
 //JwtUserDetails user = (JwtUserDetails) userDetails;  
 final String username = getUsernameFromToken(token);  
 return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));  
 }  
  
 private Date calculateExpirationDate(Date createdDate) {  
 return new Date(createdDate.getTime() + expiration \* 1000);  
 }  
}

With the above when you hit “/authenticate”, you will get JWT token. If you want to access other rest end point you have to use this token with the header.

Format of Header

Key: Authorization

Value: Bearer <token>

But, with above code if we want to access the endoint we will get forbidden error:

{

    "timestamp": "2020-01-19T11:02:27.729+0000",

    "status": 403,

    "error": "Forbidden",

    "message": "Access Denied",

    "path": "/hello"

}

As we have the functionality of generating token but functionality of using this token to access other endpoints is yet to be written.

So we have to tell spring listen every request with header Bearer <token> and take out this <token> and get username out of it, verify it’s a valid JWT and put it into security context. This is achieved by using Filter. There are number of filters available so we have to exptend one of those Filters and implement the filter method ourselves.

@Component  
public class MyJwtFilter extends OncePerRequestFilter {  
 @Autowired  
 private MyUserDetailsService myUserDetailsService;  
  
 @Autowired  
 private JwtUtil jwtUtil;  
  
 @Override  
 protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {

String header = request.getHeader("Authorization");  
 String token = null;  
 String username = null;  
  
 if (header != null && header.startsWith("Bearer ")) {  
 token = header.substring(7);  
 username = jwtUtil.getUsernameFromToken(token);  
 }  
  
 if (username != null && SecurityContextHolder.*getContext*().getAuthentication() == null) {  
 UserDetails userDetails = myUserDetailsService.loadUserByUsername(username);  
  
 if (jwtUtil.validateToken(token, userDetails)) {  
 UsernamePasswordAuthenticationToken upt =

new UsernamePasswordAuthenticationToken(userDetails, null,userDetails.getAuthorities());  
   
 upt.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));  
 SecurityContextHolder.*getContext*().setAuthentication(usernamePasswordAuthenticationToken);  
 }  
 }  
  
 filterChain.doFilter(request, response);  
 }  
}

Now that we have this filter, we have to tell Spring to use this filter instead of its default filter i.e, UsernamePasswordAuthenticationFilter

**Modifying Configuration class**

@Override  
protected void configure(HttpSecurity http) throws Exception {  
 // disables cross site request forgery  
 http.csrf().disable()  
 // authorizes requests for /authenticate and permit everybody to access it  
 .authorizeRequests().antMatchers("/authenticate").permitAll()  
 // any other request needs to be authenticated  
 .anyRequest().authenticated()  
 .and()  
 // don't maintain sessions  
 .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.*STATELESS*);  
  
 // call MyJwtFilter Before default UsernamePasswordAuthenticationFilter  
 http.addFilterBefore(myJwtFilter, UsernamePasswordAuthenticationFilter.class);  
}