**Spring Security**

User Details: They simply store user information which is later encapsulated into Authentication objects. This allows non-security related user information (such as email addresses, telephone numbers etc) to be stored in a convenient location.

Step 0: All the spring security and all the jwt related dependencies.

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
</dependency>

<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-jackson</artifactId>  
 <version>0.11.1</version>  
 <scope>runtime</scope>  
</dependency>  
<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-api</artifactId>  
 <version>0.11.1</version>  
</dependency>  
  
<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-impl</artifactId>  
 <version>0.11.1</version>  
 <scope>runtime</scope>  
</dependency>

Step 1:

Create an api to create Jwt token:

@RequestMapping(value = "/authenticate", method = RequestMethod.*POST*)  
 public ResponseEntity<?> createAuthentication(@RequestBody AuthenticationRequest authenticationRequest) throws Exception  
 {  
 *log*.info("kcdjcdscsdcdsc");  
 *myUsername* =authenticationRequest.getUsername();  
   
 final UserDetails userDetails = userDetailsService  
 .loadUserByUsername(authenticationRequest.getUsername());  
  
 final String jwt = jwtTokenUtil.generateToken(userDetails);  
 return ResponseEntity.*ok*(new AutheticationResponse(jwt));  
  
  
 }

This api will take username and password in the form of AuthenticationRequest and will return jwt in form of AuthenticationResponse.

Step:2

Create jwt util class which will have all the utility methods to generate the jwt token

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();  
}

Step 3: Create one configuration file which will extend WebSecurityConfigureAdapter

Override both the configure methods and do the following modifications

@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 // *TODO Auto-generated method stub* auth.userDetailsService(userDetailsService);  
   
}  
  
@Override  
protected void configure(HttpSecurity http) throws Exception {  
 // *TODO Auto-generated method stub* http.csrf().disable()  
 .authorizeRequests()  
 .antMatchers("/admin").hasRole("ADMIN")  
 .antMatchers("/user").hasAnyRole("ADMIN","USER")  
 .antMatchers("/authenticate").permitAll()  
 .antMatchers("/registeruser").permitAll()  
 .anyRequest().authenticated()  
 .and().sessionManagement()  
 .sessionCreationPolicy(SessionCreationPolicy.*STATELESS*);  
   
 http.addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class);  
   
}

Here I have added authentication for “/admin” and “/user”

And other are open.

Done With Spring Security in spring Mvc using JWT.

**Creting api Gateway And adding jwt authentication to it**

Step4: add the following dependencies

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-webflux</artifactId>  
</dependency>

<dependency>  
 <groupId>jakarta.servlet</groupId>  
 <artifactId>jakarta.servlet-api</artifactId>  
 <version>6.0.0</version>  
</dependency>  
<dependency>  
 <groupId>org.eclipse.jetty</groupId>  
 <artifactId>jetty-server</artifactId>  
 <version>${jetty.version}</version>  
</dependency>  
  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-gateway</artifactId>  
</dependency>  
<!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-starter-hystrix -->  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-hystrix</artifactId>  
 <version>1.3.0.RELEASE</version>  
</dependency>  
  
  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
</dependency>

And the dependencies form step 0

Step 5:

Inside yml file add all ther services which you want to call through api gateway

cloud:  
 gateway:  
 routes:  
 - id: MOVIE-CATELOG  
 uri: lb://MOVIE-CATELOG  
 predicates:  
 - Path=/catelog/\*\*  
  
 - id: MOVIE-INFO-SERVICE  
 uri: lb://MOVIE-INFO-SERVICE  
 predicates:  
 - Path=/movies/\*\*  
  
 - id: RATING-INFO-SERVICE  
 uri: lb://RATING-INFO-SERVICE  
 predicates:  
 - Path=/ratings/\*\*

Step:6

Register the api gateway on eureka server:

eureka:  
 instance:  
 prefer-ip-address: true  
 client:  
 fetch-registry: true  
 register-with-eureka: true  
 service-url:  
 defaultZone: http://localhost:8761/eureka

Step 7:

Add @EnableEurekaClient on main application

@SpringBootApplication  
@EnableEurekaClient  
public class ApiGateWaySecurityApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(ApiGateWaySecurityApplication.class, args);  
 }  
  
}

Now the api gateway is ready

Step 8:As api gateway use webflux(Reactive) and We have done configuration for jwt in spring mvc way so when doing security in reactive **ignore the step 3** as mvc and webflux is not compatible together.

Step 9:

Craeate **RouterValidator** class and inside that class mentioned list of open apiEndpoints where you don’t want to do authentication

@Component  
public class RouterValidator {  
  
 public static final List<String> *openApiEndpoints* = List.*of*(  
 "/authenticate",  
 "/registeruser"  
  
 );  
  
 public Predicate<ServerHttpRequest> isSecured =  
 request -> *openApiEndpoints* .stream()  
 .noneMatch(uri -> request.getURI().getPath().contains(uri));  
  
}

Step 10:

Create on configuration class which will have @EnableWebFluxSecurity annotation

@EnableWebFluxSecurity  
public class ReactiveWebfluxSecurityConfig {  
  
 @Bean  
 public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {  
 http  
 .csrf(csrf -> csrf.disable());  
 return http.build();  
 }  
}

Step11:

Create AuthenticationFilter class and add following code

@RefreshScope  
@Component  
@Slf4j  
public class AuthenticationFilter implements GatewayFilter {  
  
 @Autowired  
 private RouterValidator routerValidator;  
 @Autowired  
 private JwtUtil jwtUtil;  
  
 @Override  
 public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {  
 ServerHttpRequest request = exchange.getRequest();  
  
 if (routerValidator.isSecured.test(request)) {  
 if (this.isAuthMissing(request))  
 return this.onError(exchange, "Authorization header is missing in request", HttpStatus.*UNAUTHORIZED*);  
  
 final String token = this.getAuthHeader(request);  
 *log*.info(token+"..........");  
  
// if (jwtUtil.isInvalid(token))  
// return this.onError(exchange, "Authorization header is invalid", HttpStatus.UNAUTHORIZED);  
 *log*.info("before populating");  
 this.populateRequestWithHeaders(exchange, token);  
 *log*.info("after populating");  
 }  
 return chain.filter(exchange);  
 }  
  
  
 /\*PRIVATE\*/  
  
 private Mono<Void> onError(ServerWebExchange exchange, String err, HttpStatus httpStatus) {  
 ServerHttpResponse response = exchange.getResponse();  
 response.setStatusCode(httpStatus);  
 return response.setComplete();  
 }  
  
 private String getAuthHeader(ServerHttpRequest request) {  
 return request.getHeaders().getOrEmpty("Authorization").get(0);  
 }  
  
 private boolean isAuthMissing(ServerHttpRequest request) {  
 return !request.getHeaders().containsKey("Authorization");  
 }  
  
 private void populateRequestWithHeaders(ServerWebExchange exchange, String token) {  
 *log*.info("inside method.........");  
// log.info("Token"+ token);  
 String jwtToken = token.split(" ")[1].trim();  
 *log*.info("Token "+ jwtToken);  
 Claims claims = jwtUtil.extractAllClaims(jwtToken);  
 System.*out*.println("//////////////////////");  
  
 *log*.info("token found in request!");  
 *log*.info(claims+"..........");  
 System.*out*.println(claims);  
 exchange.getRequest().mutate()  
 .header("id", String.*valueOf*(claims.get("id")))  
 .header("role", String.*valueOf*(claims.get("role")))  
 .build();  
 }  
}

Step 12:

Create gatewayConfig class and add **authentication filter** on all the microservices

@Configuration  
@EnableHystrix  
public class GatewayConfig {  
  
 @Autowired  
 AuthenticationFilter filter;  
  
 @Bean  
 public RouteLocator routes(RouteLocatorBuilder builder) {  
 return builder.routes()  
 .route("MOVIE-CATELOG", r -> r.path("/catelog/\*\*")  
 .filters(f -> f.filter(filter))  
 .uri("lb://MOVIE-CATELOG"))  
  
 .route("MOVIE-INFO-SERVICE", r -> r.path("/movies/\*\*")  
 .filters(f -> f.filter(filter))  
 .uri("lb://MOVIE-INFO-SERVICE"))  
 .route("RATING-INFO-SERVICE", r -> r.path("/ratings/\*\*")  
 .filters(f -> f.filter(filter))  
 .uri("lb://RATING-INFO-SERVICE"))  
  
 .build();  
 }  
  
}