**JWT. Dưới đây là quy trình tổng quan:**

**1,Đăng nhập:**

-Khi người dùng đăng nhập vào ứng dụng, máy chủ sẽ tạo ra một token JWT và trả về cho trình duyệt của người dùng.

-Token này chứa thông tin về người dùng như ID hoặc tên, cũng như các thông tin khác như thời gian hết hạn và chữ ký kỹ thuật số để đảm bảo tính toàn vẹn của token.

**2,Lưu trữ token:**

-Trình duyệt của người dùng thường lưu trữ token này, thường trong local storage hoặc trong một cookie HTTP-only.

**3,Thực hiện thao tác:**

-Khi người dùng thực hiện một thao tác (ví dụ: truy cập một trang hoặc gửi một yêu cầu API), trình duyệt sẽ **gửi token JWT cùng với yêu cầu** đến máy chủ.

**4,Xác thực token:**

-Phía máy chủ nhận được yêu cầu, nó sẽ kiểm tra token được gửi kèm. Máy chủ sẽ kiểm tra chữ ký của token để đảm bảo rằng nó không bị thay đổi, và sau đó kiểm tra các thông tin trong token để xác định người dùng đã được xác thực và có quyền truy cập vào tài nguyên yêu cầu hay không.

**5,Phản hồi:**

-Nếu token được xác thực thành công, máy chủ sẽ phản hồi với thông tin yêu cầu của người dùng.

**6,Tiếp tục truy cập:**

-Nếu token vẫn còn hiệu lực, người dùng có thể tiếp tục sử dụng nó để truy cập các tài nguyên và dịch vụ mà họ có quyền truy cập mà không cần phải đăng nhập lại. Khi token hết hạn, người dùng sẽ cần phải đăng nhập lại để nhận được một token mới.

Tóm lại, JWT là một phương thức xác thực dựa trên token giúp cho việc quản lý phiên làm việc và xác thực trở nên dễ dàng hơn. Token JWT được tạo ra và trao đổi giữa máy chủ và trình duyệt của người dùng để xác thực và phân quyền người dùng một cách an toàn và hiệu quả.

Thêm dependencies:

<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-api</artifactId>  
 <version>0.11.5</version>  
</dependency>  
<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-impl</artifactId>  
 <version>0.11.5</version>  
</dependency>  
<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-jackson</artifactId>  
 <version>0.11.5</version>  
</dependency>

**Cài đặt Spring Security, tạo đối tượng UserDetails**

Cấu hình:

-Đối tượng user detail là 1 đối tượng theo chuẩn của java spring dùng để quản lí việc đặng nhập ,xác thực người dùng, vậy nên cái đối tượng UserEntity của ta sẽ phải config với đối tượng User của spring.

UserDetailService là 1 interface cho nên kết quả trả về của nó sẽ là 1 hàm.

@Bean  
public UserDetailsService userDetailsService() {  
 return email ->{

UserEntity existingUser=userRepository.findByUsername(email).orElseThrow(()-> new UsernameNotFoundException("Cant not find user " +email));  
 return existingUser;  
 };  
}

Và để lấy được thông tin của user thì ta sẽ tìm kiếm user đó dưới database với unique field (tức là filed dấy dùng để phân biệt với các bản ghi khác)

-Giải thích về việc sử dụng hàm lamda ở đây,UserDetailService là 1 interface và nó sẽ trả về 1 function.

ví dụ như 1 interface bất kì nha:

interface MyFunctionalInterface {  
 void doSomething(int a, int b);  
}

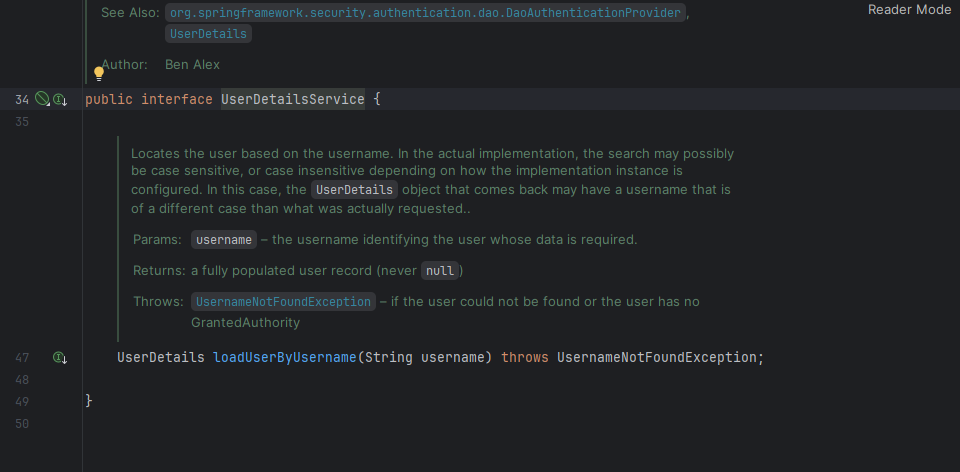
MyFunctionalInterface myLambda=(a, b) -> System.*out*.println("Sum: " + (a+b));

ở trên là 1 interface với 1 function cho nên ta có thể sử dụng hàm lambda.

myLambda(a,b) chính là triển khai của doSomething()

-Hàm lambda được sử dụng để triển khai những interface chỉ chứa 1 function

-Quay lại với bài thì UserDetailService có 1 hàm là loadUserByUsername(String usename)



Vậy nên

return email ->{

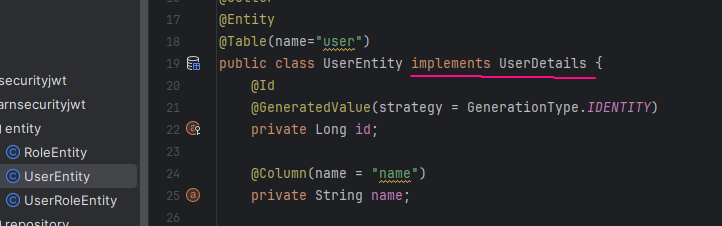
UserEntity existingUser=userRepository.findByUsername(email).orElseThrow(()-> new UsernameNotFoundException("Cant not find user " +email));  
 return existingUser;  
};

chính là loadUserByUsername(String usename) với email chính là tham số truyền vào .

@Bean  
public UserDetailsService userDetailsService() {  
 return email ->{  
 UserEntity existingUser=userRepository.findByUsername(email).orElseThrow(()-> new UsernameNotFoundException("Cant not find user " +email));  
 return existingUser;  
 };  
}

ở đây đáng ra là phải trả về 1 đối tượng UserDetails vì hàm lambda này chính là hàm loadUserByUsername(String usename) , mà ở đây lại trả về 1 Entity là bởi ta sẽ xử lí như sau:

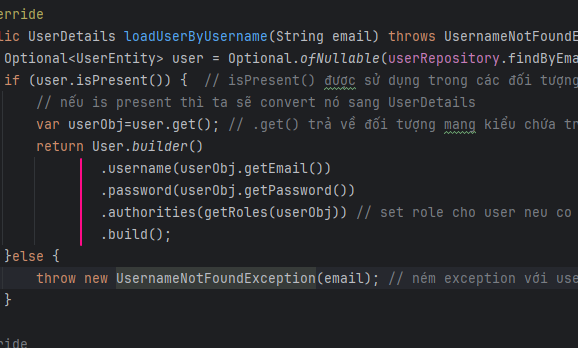
-cho UserEntity kế thừa từ UserDetails



Overide các phương thức:



Và việc mình xét các trường ở user Entity này cũng giống như việc mình sét cho các phương thức của UserDetail



Giờ nhìn lại khi mà set những trường cho User trên và sau đó loadUser…() trả về User đó giống với việc mà hàm lambda trả về UserEntity đã implement từ UserDetail và triển khai các phương thức từ UserDetail nó ngay trong UserEntity

Sau config xong class SecurityConfig:

@Configuration  
@EnableWebSecurity  
@RequiredArgsConstructor  
public class SecurityConfig {  
 @Autowired  
 private UserRepository userRepository;  
// creat user's detail object  
 @Bean  
 public UserDetailsService userDetailsService() { // lấy UserDetail  
 return email ->{  
 UserEntity existingUser=userRepository.findByUsername(email).orElseThrow(()-> new UsernameNotFoundException("Cant not find user " +email));  
 return existingUser;  
 };  
 }  
 @Bean  
 public PasswordEncoder passwordEncoder() { // mã hóa mật khẩu  
 return new BCryptPasswordEncoder();  
 }  
  
 @Bean  
 public AuthenticationProvider authenticationProvider() { // xác thực người dùng  
 DaoAuthenticationProvider daoAuthenticationProvider = new DaoAuthenticationProvider();  
 daoAuthenticationProvider.setUserDetailsService(userDetailsService());  
 daoAuthenticationProvider.setPasswordEncoder(passwordEncoder());  
 return daoAuthenticationProvider;  
 }  
  
 @Bean  
 public AuthenticationManager authenticationManager(AuthenticationConfiguration config) throws Exception {  
 return config.getAuthenticationManager();  
 }  
  
}

Tiếp đến ta sẽ tạo 1 Class là WebSecurityConfig để khi mà các request gửi tới thì cần phải có Role gì đó mới truy cập vào được.

@Configuration  
@EnableMethodSecurity  
public class WebSecurityConfig {  
 @Bean  
 public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {  
 http  
 .csrf(AbstractHttpConfigurer ::disable)  
 .authorizeHttpRequests(requests->{  
 requests.requestMatchers("\*\*").permitAll();  
 });  
  
 return http.build();  
 }  
}

Ta sẽ cấu hình cho SecurityFilterChain để lọc những request theo role. Và tạm thời ta để quyền truy cập là all.

**LOGIN + Setup token +Trả jwt token key**

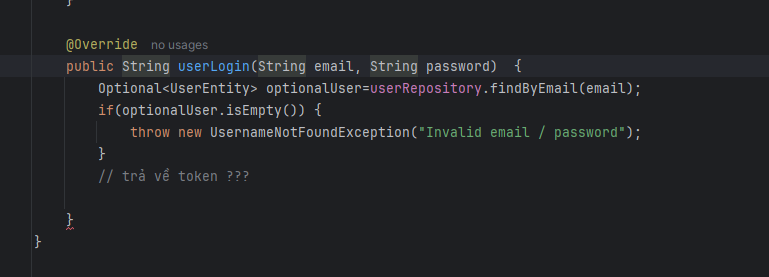
=>Vậy trước kia khi đăng nhập ta sử dụng form login để đăng nhập và xác thực người dùng nhưng trong dự án này ta sử dụng Jwt để xác thực người dùng thì ta sẽ tạo API endpoint để xử lý yêu cầu đăng nhập .

-Tạo một API endpoint nhận yêu cầu đăng nhập từ người dùng. Thông thường, yêu cầu này sẽ bao gồm tên người dùng và mật khẩu.

-Trong phần xử lý yêu cầu đăng nhập, bạn sẽ kiểm tra thông tin đăng nhập của người dùng, và nếu thông tin đúng, bạn sẽ tạo một JWT và trả về cho người dùng.

-Sau khi tạo JWT, bạn sẽ gửi nó trả lại cho người dùng, thường trong phản hồi của yêu cầu đăng nhập. Người dùng sẽ lưu trữ JWT và sử dụng nó cho các yêu cầu sau này.

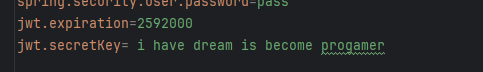
Xử lí bước đầu ở service:



-Tạo package jwt để tạo token:



Ta sẽ lưu 2 biến là expriation để lưu thời gian hiệu lực token, secrectKey(later) , file properties



**Luồng xử lý JWT**

**1.Đăng nhập (Authentication):**

* Người dùng gửi yêu cầu đăng nhập đến server với thông tin xác thực (ví dụ: username và password).
* Server kiểm tra thông tin xác thực. Nếu thông tin đúng, server tạo ra một JWT chứa các claims cần thiết (ví dụ: email, vai trò của người dùng).
* JWT này được ký với một secret key để đảm bảo tính toàn vẹn và tính xác thực của token.
* Server trả về JWT cho client.

**2.Gửi yêu cầu với JWT (Authorization):**

* Client lưu trữ JWT (thường là trong local storage hoặc cookies) và gửi JWT này trong header của các yêu cầu HTTP tiếp theo.
* Header của yêu cầu thường chứa JWT trong trường Authorization với định dạng Bearer <JWT>.

Ví dụ:

http

GET /protected/resource Authorization: **Bearer** eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

**3.Kiểm tra JWT tại server:**

* Khi server nhận được yêu cầu có chứa JWT, server sẽ kiểm tra tính hợp lệ của JWT.
* Server giải mã JWT và kiểm tra chữ ký (signature) để đảm bảo JWT không bị giả
* Nếu chữ ký hợp lệ, server sẽ trích xuất các claims từ payload của JWT.
* Server kiểm tra các thông tin trong claims (ví dụ: email, vai trò) để xác định quyền truy cập và tính hợp lệ của người dùng.

**4.Xử lý yêu cầu và trả về kết quả:**

* Nếu JWT hợp lệ và người dùng có quyền truy cập, server xử lý yêu cầu và trả về kết quả cho client.
* Nếu JWT không hợp lệ hoặc đã hết hạn, server trả về lỗi (ví dụ: 401 Unauthorized).
* @Component  
  public class JwTokenUtil {  
   @Value("${jwt.expiration}")  
   private int expiredTime; // save to an enviroment variable  
    
   @Value("${jwt.secretKey}")  
   private String secretKey;  
    
   public String generateToken(UserEntity userEntity) throws Exception{  
   // đối tượng User được đưa vào sercurity thì có các properties gọi chung là các claims  
   Map<String, Object> claims = new HashMap<>();  
   claims.put("email", userEntity.getEmail());  
   // khi tạo token trả ve String có thể bị Exception  
   try{  
   String token= Jwts.*builder*()  
   .setClaims(claims)  
   .setSubject(userEntity.getEmail())  
   .setExpiration(new Date(System.*currentTimeMillis*() + expiredTime\*1000L)) // thoi han  
   .signWith(getSignInKey(), SignatureAlgorithm.*HS256*) // signWith là kí , nghĩa là khi sinh ra 1 token thì chúng ta sẽ có 1 câu hỏi bảo mật được gọi là secrectKey,  
   .compact(); // chính sau này dịch cái token để lấy các claims thông qua nó  
   return token;  
   }catch (Exception e){  
   throw new InvalidParameterException("Cannot generate jwt token,error:"+e.getMessage());  
   }  
    
   }  
    
   private Key getSignInKey() {  
   //chuyển đổi secretKey thành một mảng byte  
   byte[] keyBytes = secretKey.getBytes(StandardCharsets.*UTF\_8*);  
   //Dòng này nhận mảng byte và tạo ra một khóa HMAC SHA-256 từ nó.  
   return Keys.*hmacShaKeyFor*(keyBytes);  
   }  
    
   // extracAll claim  
   private Claims extractAllclaims(String token ) {  
   return Jwts.*parserBuilder*()  
   .setSigningKey(getSignInKey())  
   .build()  
   .parseClaimsJws(token)  
   .getBody();  
   }  
    
   // Lấy một claim từ extracAll claim  
   public <T> T extractClaim(String token, Function<Claims,T> claimsResolve) { // lấy claim tương ứng chỉ cần truyền nó vào claimsResolve  
   final Claims claims = extractAllclaims(token);  
   return claimsResolve.apply(claims);  
   }  
    
   // check expriation token  
   public boolean isTokenExpired(String token) {  
   Date expiration = extractClaim(token,Claims::getExpiration);  
   return expiration.before(new Date());  
   }  
    
   // lay thong tin email của user tu claim thong qua token  
   public String extractEmail(String token) {  
   return extractClaim(token,Claims::getSubject);  
   }  
    
   // check thong tin token tu request client co hop le voi thong tin duoi DB cua UserDetails ? va con hieu luc hay khong ?  
   public boolean validateToken(String token, UserDetails userDetails) {  
   String email = extractEmail(token);  
   return email.equals(userDetails.getUsername()) && !isTokenExpired(token);  
   }  
    
  }

**+expiredTime:** Được chú thích **với @Value("${jwt.expiration}")**, thuộc tính này lấy giá trị từ tệp (**application.properties**) và xác định thời gian hết hạn của token tính bằng giây.

**+secretKey:** Được chú thích với **@Value("${jwt.secretKey}"),** thuộc tính này lấy giá trị từ tệp (**application.properties**) và chứa khóa bí mật được sử dụng để **ký và giải mã** token.

**-generateToken(UserEntity userEntity){} :**Phương thức này tạo ra một JWT dựa trên thông tin của UserEntity.

* claims: Là một Map chứa các thông tin (claims) của người dùng, trong trường hợp này là email của người dùng.
* token: Sử dụng thư viện jjwt để xây dựng token với các claims, chủ thể (subject), thời gian hết hạn và ký bằng khóa bí mật.

public String generateToken(UserEntity userEntity) {  
 Map<String, Object> claims = new HashMap<>();  
 claims.put("email", userEntity.getEmail());  
 try {  
 String token = Jwts.*builder*()  
 .setClaims(claims)  
 .setSubject(userEntity.getEmail())  
 .setExpiration(new Date(System.*currentTimeMillis*() + expiredTime \* 1000L))  
 .signWith(getSignInKey(),SignatureAlgorithm.*HS256*)  
 .compact();  
 return token;  
 } catch (Exception e) {  
 System.*err*.println("Cannot generate jwt token, error: " + e.getMessage());  
 return null;  
 }  
}

**getSignInKey():** Phương thức này chuyển đổi secrectKey thành một mảng byte, và sau đó tạo đối tượng Key từ mảng byte đó để sử dụng trong quá trình tạo và xác thực các chuỗi token JWT.

private Key getSignInKey() {  
// chuyển đổi secretKey thành một mảng byte  
 byte[] keyBytes = secretKey.getBytes(StandardCharsets.*UTF\_8*);  
 //Dòng này nhận mảng byte và tạo ra một khóa HMAC SHA-256 từ nó.  
 return Keys.*hmacShaKeyFor*(keyBytes);  
 }

**extractAllClaims(String token):** Phương thức này giải mã token và trả về tất cả các claims chứa trong token.

private Claims extractAllclaims(String token) {  
 return Jwts.*parserBuilder*()  
 .setSigningKey(getSignInKey())  
 .build()  
 .parseClaimsJws(token)  
 .getBody();  
}

**extractClaim(String token, Function<Claims, T> claimsResolver):** Phương thức này cho phép trích xuất một claim cụ thể từ token bằng cách sử dụng một hàm Function để xử lý claims.

public <T> T extractClaim(String token, Function<Claims,T> claimsResolve) { // lấy claim tương ứng chỉ cần truyền nó vào claimsResolve  
 final Claims claims = extractAllclaims(token);  
 return claimsResolve.apply(claims);  
}

**isTokenExpired(String token):** Phương thức này kiểm tra xem token đã hết hạn hay chưa bằng cách trích xuất thời gian hết hạn từ token và so sánh với thời gian hiện tại.

public boolean isTokenExpired(String token) {  
 Date expiration = extractClaim(token,Claims::getExpiration);  
 return expiration.before(new Date());  
}

**extractEmail(String token)** :Trích xuất email của người dùng từ token.

// lay thong tin email của user tu claim thong qua token  
public String extractEmail(String token) {  
 return extractClaim(token,Claims::getSubject);  
}

**validateToken():** Kiểm tra tính hợp lệ của token bằng cách so sánh email trong token với email của người dùng trong DB được trích xuất từ UserDetails. Kiểm tra token chưa hết hạn.

// check thong tin token tu request client co hop le voi thong tin duoi DB cua UserDetails ? va con hieu luc hay khong ?  
public boolean validateToken(String token, UserDetails userDetails) {  
 String email = extractEmail(token);  
 return email.equals(userDetails.getUsername()) && !isTokenExpired(token);  
}

*Tóm tắt Lớp JwTokenUtil cung cấp các phương thức sau:*

* generateToken(UserEntity userEntity): Tạo ra một JWT cho người dùng.
* getSignInKey():Chuyển đổi secrectKey thành đối tượng Key.
* extractAllClaims(String token): Giải mã và trích xuất tất cả các claims từ token.
* extractClaim(String token, Function<Claims, T> claimsResolver): Trích xuất một claim cụ thể từ token.
* isTokenExpired(String token): Kiểm tra xem token đã hết hạn hay chưa.
* extractEmail(String token) :Trích xuất email của người dùng từ token.
* validateToken(): Kiểm tra tính hợp lệ của token bằng cách so sánh email trong token với email của người dùng trong DB được trích xuất từ UserDetails. Kiểm tra token chưa hết hạn.

=>Các phương thức này giúp quản lý và xác thực JWT trong ứng dụng của bạn, đảm bảo an ninh và tính toàn vẹn của thông tin người dùng.

Đây là luồng code chính để tạo ra token.

public String generateToken(UserEntity userEntity) throws Exception{  
 // đối tượng User được đưa vào sercurity thì có các properties gọi chung là các claims  
 Map<String, Object> claims = new HashMap<>();  
 claims.put("email", userEntity.getEmail());  
 // khi tạo token trả ve String có thể bị Exception  
 try{  
 String token= Jwts.*builder*()  
 .setClaims(claims)  
 .setSubject(userEntity.getEmail())  
 .setExpiration(new Date(System.*currentTimeMillis*() + expiredTime\*1000L)) // thoi han  
 .signWith(getSignInKey(), SignatureAlgorithm.*HS256*) // signWith là kí , nghĩa là khi sinh ra 1 token thì chúng ta sẽ có 1 câu hỏi bảo mật được gọi là secrectKey,  
 .compact(); // chính sau này dịch cái token để lấy các claims thông qua nó  
 return token;  
 }catch (Exception e){  
 throw new InvalidParameterException("Cannot generate jwt token,error:"+e.getMessage());  
 }  
  
}

Cái gì quan trọng thì cứ nhắc lại:

**Payload (claims):** Payload của token chứa các thông tin về người dùng hoặc các thông tin khác bạn muốn chuyển đi cùng với token. Trong đoạn mã của bạn, payload được đặt thông qua phương thức .**setClaims(claims)** . Trong payload này, thông tin người dùng được đặt là email (userEntity.getEmail()) và chủ đề ( setSubject() ), cùng với thời gian hết hạn ( setExpiration() ).

**Signature (Chữ ký):** Chuỗi token được ký bằng cách sử dụng secretKey (được tạo ra bởi **getSignInKey()**) và thuật toán chữ ký (SignatureAlgorithm.HS256 thông tin này được hiển thị tại header của token). Điều này đảm bảo rằng token không bị sửa đổi trong quá trình truyền tải và có thể được xác thực bởi máy chủ.

**Chuỗi token:** Cuối cùng, tất cả các thành phần trên (Header, Payload và Signature) được kết hợp lại và mã hóa thành một chuỗi token duy nhất. Chuỗi token này có thể được gửi đến máy chủ để xác thực và xác định người dùng.

Tôi đăng nhập với data như sau:

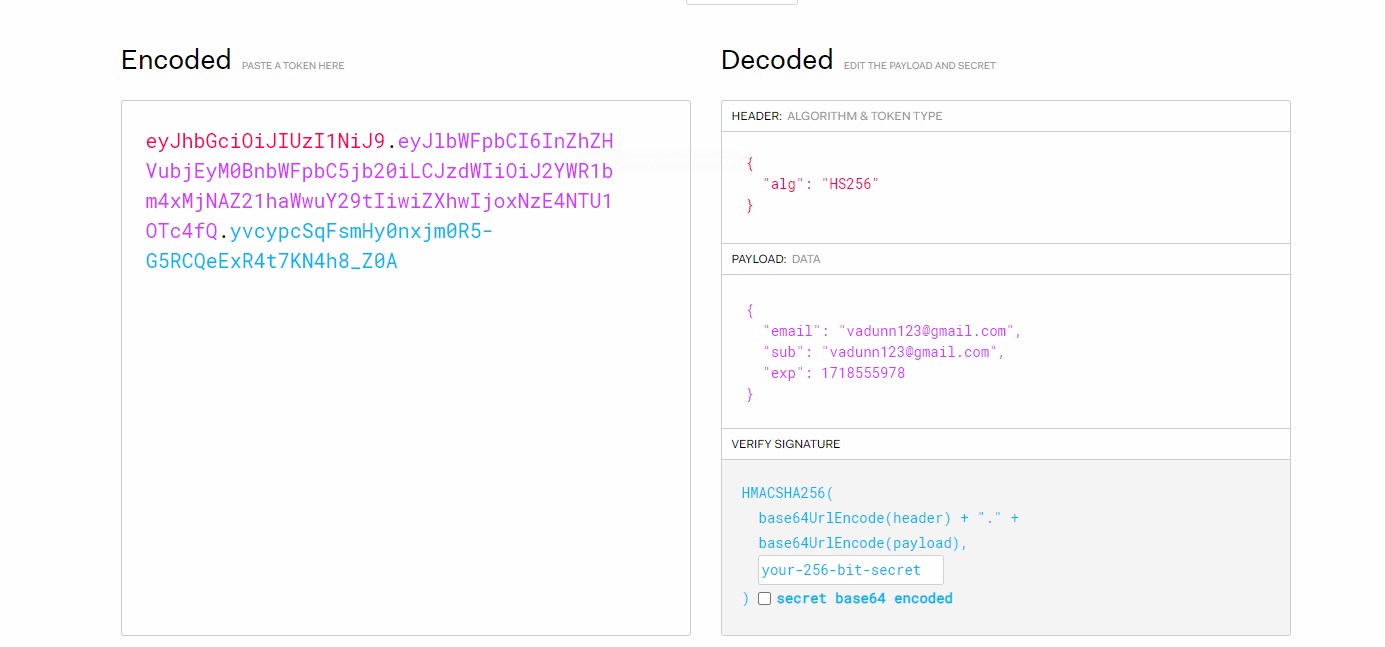
{

  "email":"vadunn123@gmail.com",

    "password":"123"

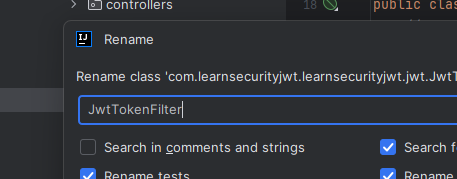
}

Token sinh ra: eyJhbGciOiJIUzI1NiJ9.eyJlbWFpbCI6InZhZHVubjEyM0BnbWFpbC5jb20iLCJzdWIiOiJ2YWR1bm4xMjNAZ21haWwuY29tIiwiZXhwIjoxNzE4NTU1OTc4fQ.yvcypcSqFsmHy0nxjm0R5-G5RCQeExR4t7KN4h8\_Z0A



**Lọc Request với doFilterlnternal**

Khởi tạo:



@Component  
public class JwtTokenFilter extends OncePerRequestFilter {  
 @Autowired  
 private UserDetailsService userDetailsService;  
 @Autowired  
 private JwTokenUtil jwTokenUtil;  
  
 @Override  
 protected void doFilterInternal(@NonNull HttpServletRequest request,@NonNull HttpServletResponse response,@NonNull FilterChain filterChain) throws ServletException, IOException {  
 try{  
 if (isBypassToken((request))){   
 filterChain.doFilter(request,response;

return;  
 }  
 final String auHeader=request.getHeader("Authorization");   
 if(auHeader == null || !auHeader.startsWith("Bearer ")){  
 response.sendError(HttpServletResponse.*SC\_UNAUTHORIZED*,"Unauthorized");  
 return;  
 }  
 final String token = auHeader.substring(7);

final String email=jwTokenUtil.extractEmail(token);

if(email!=null && SecurityContextHolder.*getContext*().getAuthentication()== null){ UserEntity userDetails= (UserEntity) userDetailsService.loadUserByUsername(email);

if(jwTokenUtil.validateToken(token, userDetails)){   
 UsernamePasswordAuthenticationToken authenticationToken=new UsernamePasswordAuthenticationToken(  
 userDetails,  
 null,  
 userDetails.getAuthorities());  
 authenticationToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));  
 SecurityContextHolder.*getContext*().setAuthentication(authenticationToken) }  
 }  
 filterChain.doFilter(request,response);  
 }catch (Exception e){  
 response.sendError(HttpServletResponse.*SC\_UNAUTHORIZED*,"Unauthorized");  
 }  
 }  
  
 private boolean isBypassToken(@NonNull HttpServletRequest request){  
  
 final List<Pair<String,String>> bypassTokens= Arrays.*asList*(  
 //Request khong can token  
 Pair.*of*("/api/home","GET"),  
 Pair.*of*("/user/login","POST"),  
 Pair.*of*("/user/register","POST")  
 );  
 for(Pair<String,String> bypassToken: bypassTokens){ if(request.getServletPath().contains((bypassToken.getFirst())) && request.getMethod().equals(bypassToken.getSecond())){  
 return true;  
 }  
 }  
 return false;  
 }  
}

-Đoạn mã trên là một phần của một bộ lọc (filter) trong Spring Framework được sử dụng để kiểm tra và xử lý các yêu cầu HTTP (HTTP requests) trước khi chúng đến các bộ điều khiển (controllers).

-Cụ thể hơn, đây là một bộ lọc kiểm tra token JWT (JSON Web Token) để xác định xem yêu cầu có được phép truy cập vào hệ thống hay không. Nếu yêu cầu không cần token (bypass), bộ lọc sẽ cho phép yêu cầu đó đi qua mà không cần kiểm tra token. Sau đây là giải thích chi tiết từng phần của đoạn mã:

**1. JwtTokenFilter class**

public class JwtTokenFilter extends OncePerRequestFilter {  
 // ...  
}

-JwtTokenFilter kế thừa từ OncePerRequestFilter, một lớp cơ sở trong Spring để đảm bảo bộ lọc chỉ được gọi một lần mỗi yêu cầu (request), mọi request HTTP đến server sẽ được xử lý qua filter này trước khi đến các endpoint.

**2.doFilterInternal method**

@Override  
 protected void doFilterInternal(@NonNull HttpServletRequest request,@NonNull HttpServletResponse response,@NonNull FilterChain filterChain) throws ServletException, IOException {  
 try{  
 if (isBypassToken((request))){   
 filterChain.doFilter(request,response;

return;  
 }  
 final String auHeader=request.getHeader("Authorization");   
 if(auHeader!= null || !auHeader.startsWith("Bearer ")){  
 response.sendError(HttpServletResponse.*SC\_UNAUTHORIZED*,"Unauthorized");  
 return;  
 }  
 final String token = auHeader.substring(7);

final String email=jwTokenUtil.extractEmail(token);

if(email!=null && SecurityContextHolder.*getContext*().getAuthentication()== null){ UserEntity userDetails= (UserEntity) userDetailsService.loadUserByUsername(email);

if(jwTokenUtil.validateToken(token, userDetails)){   
 UsernamePasswordAuthenticationToken authenticationToken=new UsernamePasswordAuthenticationToken(  
 userDetails,  
 null,  
 userDetails.getAuthorities());  
 authenticationToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));  
 SecurityContextHolder.*getContext*().setAuthentication(authenticationToken) }  
 }  
 filterChain.doFilter(request,response);  
 }catch (Exception e){  
 response.sendError(HttpServletResponse.*SC\_UNAUTHORIZED*,"Unauthorized");  
 }  
 }

-Phương thức doFilterInternal() được ghi đè từ **OncePerRequestFilter** và chứa logic kiểm tra để xác định xem yêu cầu có cần token hay không.

@Override cho biết phương thức này ghi đè phương thức cùng tên từ lớp cha. if (isBypassToken(request)) kiểm tra xem yêu cầu hiện tại có nằm trong danh sách yêu cầu không cần token (bypass) hay không. Nếu điều kiện trên đúng, filterChain.doFilter(request, response) sẽ cho phép yêu cầu tiếp tục đi qua các bộ lọc và đến các bộ điều khiển mà không cần kiểm tra token.ngược lại sẽ phải check token và xác thực người dùng

**2.1 Lấy và kiểm tra header Authorization**

**Khi client muốn request vào tài nguyên bảo vệ, nó sẽ gửi JWT trong header của HTTP request:**

final String auHeader=request.getHeader("Authorization"); // Lấy authorization (bearer…token)  
// check xem request có gửi token được lưu trong Authorization nằm trong header của HTTP request ?  
if(auHeader== null || !auHeader.startsWith("Bearer ")){  
 response.sendError(HttpServletResponse.*SC\_UNAUTHORIZED*,"Unauthorized");  
 return;  
}

* request.getHeader("Authorization"): Lấy giá trị của header Authorization từ phần header HTTPrequest.
* auHeader.startsWith("Bearer "): Kiểm tra xem header này có bắt đầu bằng "Bearer " hay không.
* auHeader.substring(7): Loại bỏ phần "Bearer " để lấy token thực sự.

**2.2. Trích xuất email từ token và đồng thời check xem đã được xác thực chưa**

final String email=jwTokenUtil.extractEmail(token); // lay email tu trong payload ra  
if(email!=null && SecurityContextHolder.*getContext*().getAuthentication()== null){

jwTokenUtil.extractEmail(token): Sử dụng JwTokenUtil để trích xuất email từ token.

SecurityContextHolder.getContext().getAuthentication() == null: Kiểm tra xem đã có đối tượng Authentication trong SecurityContext chưa (nghĩa là người dùng đã được xác thực chưa).

**2.3. Tải thông tin người dùng và xác thực token với hệ thống**

**-Chú ý ở đây ta sẽ Sử dụng UserEntity vì ta đã cho nó implements từ UserDetails**

UserEntity userDetails= (UserEntity) userDetailsService.loadUserByUsername(email); //lấy thông tin UserEntity (đã được implements từ UserDetail) dưới hệ thống thông qua loadUserByUsername(email), để xác thực với user trong token

if(jwTokenUtil.validateToken(token, userDetails)){ // check thong tin token va thong tin UserDetail ? check hieu luc token ?  
 UsernamePasswordAuthenticationToken authenticationToken=new UsernamePasswordAuthenticationToken(userDetails,  
 null,  
 userDetails.getAuthorities()); // authenticate voi he thong cua spring  
 authenticationToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));  
 SecurityContextHolder.*getContext*().setAuthentication(authenticationToken);  
 }  
}  
filterChain.doFilter(request,response);

userDetailsService.loadUserByUsername(email): Lấy thông tin người dùng từ email bằng cách sử dụng UserDetailsService.

jwTokenUtil.validateToken(token, userDetails): Kiểm tra tính hợp lệ và hiệu lực của token bằng cách so sánh token và thông tin người dùng.

UsernamePasswordAuthenticationToken: Tạo đối tượng Authentication mới với thông tin người dùng và quyền hạn của họ.

WebAuthenticationDetailsSource().buildDetails(request)): Đặt thông tin bổ sung về request (như địa chỉ IP) vào Authentication.

SecurityContextHolder.getContext().setAuthentication(authenticationToken): Đặt Authentication vào SecurityContext để Spring Security có thể quản lý và có thể biết rằng người dùng đã được xác thực và có các quyền gì.

**3. isBypassToken method**

private boolean isBypassToken(@NonNull HttpServletRequest request){  
 final List<Pair<String, String>> bypassTokens = Arrays.*asList*(  
 // Requests không cần token  
 Pair.*of*("/api/home", "GET"),  
 Pair.*of*("/user/login", "POST"),  
 Pair.*of*("/user/register", "POST")  
 );  
 for(Pair<String, String> bypassToken : bypassTokens) { // duyệt  
 if(request.getServletPath().contains(bypassToken.getFirst()) && request.getMethod().equals(bypassToken.getSecond())) {  
 return true;  
 }  
 }  
 return false;  
}

-isBypassToken là một phương thức để kiểm tra xem yêu cầu có nằm trong danh sách yêu cầu không cần token hay không.

-final List<Pair<String, String>> bypassTokens tạo một danh sách các cặp (path, method) của các yêu cầu không cần token.

* Pair.of("/api/home", "GET"): Yêu cầu GET đến đường dẫn /api/home.
* Pair.of("/user/login", "POST"): Yêu cầu POST đến đường dẫn /user/login.
* Pair.of("/user/register", "POST"): Yêu cầu POST đến đường dẫn /user/register.

-Vòng lặp for duyệt qua danh sách bypassTokens. Nếu request.getServletPath().contains(bypassToken.getFirst()) và request.getMethod().equals(bypassToken.getSecond()) đều đúng, phương thức trả về true, nghĩa là yêu cầu này được bypass.

-Nếu không có cặp nào khớp, phương thức trả về false, nghĩa là yêu cầu này cần token để được xử lý.

**-Tổng kết:**

Bộ lọc này cho phép các yêu cầu đến một số endpoint cụ thể (/api/home, /user/login, /user/register) với các phương thức HTTP tương ứng (GET hoặc POST) được bypass mà không cần token JWT.

Các yêu cầu khác ngoài danh sách này sẽ phải qua kiểm tra token JWT trước khi được phép truy cập vào hệ thống.

Lấy authorization (bearer…token)