**JWT – JSON Web Token**

JWT stands for JSON Web Token. It is a compact, URL-safe means of representing claims between two parties. This token format is often used for authentication and authorization purposes in web development and APIs.

JWTs consist of three parts, each encoded in Base64 and separated by dots ('.'). These parts are:

* Header: Contains metadata about the type of token and the signing algorithm used.

{

“alg”:” HS256”

“typ”: “JWT”

}

* Payload: Contains claims or statements about the user or entity, such as user ID, role, or other relevant information.

{

"sub": "vivek7@gmail.com",

"iat": 1704354718,

"exp": 1704356158

}

* Signature: Created by combining the encoded header, encoded payload, and a secret key using the specified algorithm. It ensures the integrity of the token

HMACSHA256(

base64UrlEncode(header) + "." +

base64UrlEncode(payload),

The sample project was done for JWT Authentication and Authorization

**Technologies**

* Spring Boot 3.0
* Spring Security
* JSON Web Tokens (JWT)
* Maven

**Dependencies used for the project:**

*<dependency>*

*<groupId>io.jsonwebtoken</groupId>*

*<artifactId>jjwt-jackson</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-api</artifactId>*

*<version>0.11.5</version>*

*</dependency>*

*<dependency>*

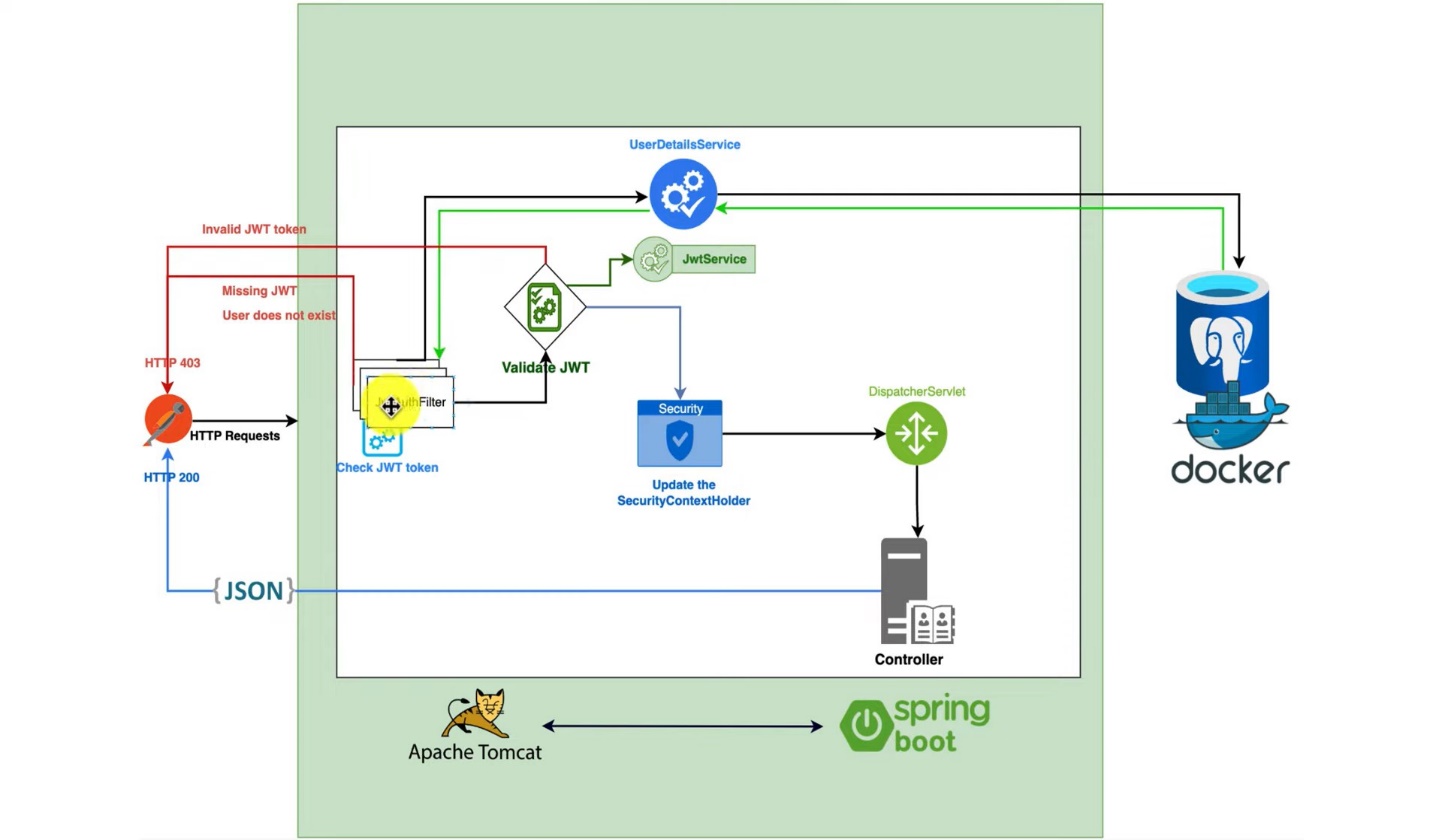
*<groupId>org.springframework.boot</groupId>*

*<artifactId>spring-boot-starter-security</artifactId>*

*</dependency>*

**JWT Authentication Flow**

The JWT (JSON Web Token) authentication flow follows a secure and stateless process for user identification in web applications. When a user registers or logs in, the server generates a JWT containing essential user information and signs it with a secret key. This token is issued to the client, usually stored in a secure HTTP-only cookie or local storage. For subsequent requests, the client presents the JWT in the Authorization header. The server verifies the token's signature, extracts user details, performs authorization checks based on roles, and ensures the token's validity by checking expiration time. The stateless nature of JWTs eliminates the need for server-side session storage, providing an efficient and scalable solution for secure user authentication and access control in modern web applications and APIs.



**JwtAuthenticationFilter Class**

This Java class represents a custom filter for JWT authentication in an HTTP request-response cycle. Its purpose is to extract and validate JWT tokens from the "Authorization" header, subsequently establishing the user's authentication context. The filter optionally allows specific paths to bypass authentication. It verifies the presence and format of the "Authorization" header, extracts the JWT token, and performs user authentication logic only if necessary conditions are met. User details are loaded, and the JWT token is validated using designated services. Upon successful validation, an authentication token is created, and the authentication context is set up. The filter then allows the request to continue through the remaining filters in the chain, making it suitable for securing API endpoints with JWT-based authentication.

*@Override*

*protected void doFilterInternal(*

*@NonNull HttpServletRequest request,*

*@NonNull HttpServletResponse response,*

*@NonNull FilterChain filterChain*

*) throws ServletException, IOException {*

*// if (request.getServletPath().contains("/api/v1/auth")) {*

*// filterChain.doFilter(request, response);*

*// return;*

*// }*

*final String authHeader = request.getHeader("Authorization");*

*final String jwt;*

*final String userEmail;*

*if (authHeader == null ||!authHeader.startsWith("Bearer ")) {*

*filterChain.doFilter(request, response);*

*return;*

*}*

*jwt = authHeader.substring(7);*

*userEmail = jwtService.extractUsername(jwt);*

*if (userEmail != null && SecurityContextHolder.getContext().getAuthentication() == null) {*

*UserDetails userDetails = this.userDetailsService.loadUserByUsername(userEmail);*

*if (jwtService.isTokenValid(jwt, userDetails) ) {*

*UsernamePasswordAuthenticationToken authToken = new UsernamePasswordAuthenticationToken(*

*userDetails,*

*null,*

*userDetails.getAuthorities()*

*);*

*authToken.setDetails(*

*new WebAuthenticationDetailsSource().buildDetails(request)*

*);*

*SecurityContextHolder.getContext().setAuthentication(authToken);*

*}*

*}*

*filterChain.doFilter(request, response);*

*}*

*}*

**JwtService Class**

The presented Java class functions as a JWT Service Class, playing a crucial role in the JWT authentication process within the application. It is responsible for token generation, validation, and extraction of user credentials from the database during authentication. Leveraging the Java JWT (JJWT) library, the class includes methods to extract usernames, generate JWT tokens for UserDetails, validate token authenticity, and handle token expiration. Notably, the SECRET\_KEY is employed for token signing, and its confidentiality is paramount. Methods such as generateToken, isTokenValid, and extractUsername contribute to the comprehensive JWT authentication logic within the application. This service class enhances security by ensuring the validity and integrity of JWT tokens, thereby facilitating secure user authentication.

*private static final String SECRET\_KEY= "tYAZvJOzK/CIsyyegXpZWzsTR2JuHD6+Y5bwBSHKLE1n82G4KAJWIpHIe+JWa7qZ\n";*

*public String extractUsername(String token) {*

*return extractClaim(token, Claims::getSubject);*

*}*

*public <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {*

*final Claims claims = extractAllClaims(token);*

*return claimsResolver.apply(claims)}*

*public String generateToken(UserDetails userDetails) {*

*return generateToken(new HashMap<>(), userDetails);*

*}*

*public String generateToken(*

*Map<String, Object> extraClaims,*

*UserDetails userDetails*

*) {*

*return Jwts*

*.builder()*

*.setClaims(extraClaims)*

*.setSubject(userDetails.getUsername())*

*.setIssuedAt(new Date(System.currentTimeMillis()))*

*.setExpiration(new Date(System.currentTimeMillis()+1000\*60\*24))*

*.signWith(getSignInKey(),SignatureAlgorithm.HS256)*

*.compact();*

*}*

*private String buildToken(*

*Map<String, Object> extraClaims,*

*UserDetails userDetails,*

*long expiration*

*) {*

*return Jwts*

*.builder()*

*.setClaims(extraClaims)*

*.setSubject(userDetails.getUsername())*

*.setIssuedAt(new Date(System.currentTimeMillis()))*

*.setExpiration(new Date(System.currentTimeMillis() + expiration))*

*.signWith(getSignInKey(), SignatureAlgorithm.HS256)*

*.compact();*

*}*

*public boolean isTokenValid(String token, UserDetails userDetails) {*

*final String username = extractUsername(token);*

*return (username.equals(userDetails.getUsername())) && !isTokenExpired(token);*

*}*

*private boolean isTokenExpired(String token) {*

*return extractExpiration(token).before(new Date());*

*}*

*private Date extractExpiration(String token) {*

*return extractClaim(token, Claims::getExpiration);*

*}*

*private Claims extractAllClaims(String token) {*

*return Jwts*

*.parserBuilder()*

*.setSigningKey(getSignInKey())*

*.build()*

*.parseClaimsJws(token)*

*.getBody();*

*}*

*private Key getSignInKey() {*

*byte[] keyBytes = Decoders.BASE64.decode(SECRET\_KEY);*

*return Keys.hmacShaKeyFor(keyBytes);*

*}*

The application employs three key classes—RegisterRequest, AuthenticationRequest, and AuthenticationResponse—to facilitate user registration and authentication processes. The RegisterRequest class encapsulates the necessary information for user registration, while the AuthenticationRequest class is designed for handling user authentication requests. The AuthenticationResponse class is utilized to collect and structure the responses generated during the authentication process. The central logic orchestrating user registration and authentication is encapsulated in the AuthenticationService class. This class serves as the core component responsible for coordinating the interactions between these classes, ensuring a seamless and secure user authentication experience. The modular design of these classes promotes code organization and maintainability, facilitating effective user management within the application. Developers can refer to the AuthenticationService class for a comprehensive understanding of the application's user registration and authentication mechanisms.

**AuthenticationService Class**

The authenticate method within the AuthenticationService class orchestrates user authentication by utilizing the AuthenticationManager. It validates the user's credentials provided in an AuthenticationRequest by creating a UsernamePasswordAuthenticationToken. After successful authentication, the method retrieves the user from the repository based on the email, generates a JWT token using the JwtService, and constructs an AuthenticationResponse encapsulating the token. On the other hand, the register method facilitates user registration. It constructs a new user entity from the information in a RegisterRequest, encrypts the password using a PasswordEncoder, and persists the user to the repository. Subsequently, it generates a JWT token using the JwtService and constructs an AuthenticationResponse containing the token and user information. These methods collectively provide a comprehensive solution for user authentication and registration within the application, encapsulating core logic for secure user management. Developers can seamlessly integrate these methods to enhance the application's user authentication and registration capabilities.

*public AuthenticationResponse register (RegisterRequest request) {*

*var user = User.builder()*

*.firstname(request.getFirstname())*

*.lastname(request.getLastname())*

*. email(request.getEmail())*

*. password (passwordEncoder.encode(request.getPassword()))*

*. role (Role.USER)*

*. build ();*

*repository.save(user);*

*var jwtToken=jwtService.generateToken(user);*

*return AuthenticationResponse.builder()*

*. token(jwtToken)*

*.user(user)*

*. build();*

*public AuthenticationResponse authenticate (AuthenticationRequest request) {*

*authenticationManager.authenticate(*

*new UsernamePasswordAuthenticationToken(*

*request.getEmail(),*

*request.getPassword()*

*)*

*);*

*var user = repository.findByEmail(request.getEmail())*

*.orElseThrow();*

*var jwtToken = jwtService.generateToken(user);*

*return AuthenticationResponse.builder(). token(jwtToken)*

*. build();*

*}*

**Role Based JWT Authentication**

Role-based JWT authentication is a security mechanism that leverages JSON Web Tokens (JWTs) to manage user authentication and authorization based on their roles within an application. In this authentication approach, each user is assigned one or more roles, and a JWT token is used to verify their identity and access permissions. Here's an overview of the key components and concepts involved in role based JWT authentication:

Roles:

Roles represent specific permissions or responsibilities assigned to a user within the application. Common roles include "USER," "ADMIN," or other custom roles tailored to the application's needs.

JWT Token Structure:

JWTs typically consist of three parts: a header, a payload, and a signature. The payload includes claims such as the user's roles. In a role based JWT authentication system, a user's roles would be included in the JWT payload.

Authentication Process:

When a user logs in or makes a request that requires authentication, the server verifies the user's credentials and generates a JWT containing relevant information, including the user's roles. The user receives this JWT.

Token Issuance and Validation:

The server issues JWTs upon successful authentication. Subsequent requests from the user include this JWT in the Authorization header. The server validates the token's signature to ensure its integrity and extracts user information, including roles, to determine access permissions.

Authorization Checks:

Upon receiving a request, the server checks the roles embedded in the JWT to determine if the user has the required permissions for the requested operation. Access is granted or denied based on the user's roles.

Middleware or Filter Implementation:

In the server-side code, middleware or filters are often employed to intercept incoming requests, extract and validate JWTs, and enforce role-based access control.

The application facilitates user registration through the register API, generating a JWT token upon successful registration. This token is then used for authentication via the authenticate API. Subsequently, the token is employed to access secured endpoints in the admin controller, ensuring role-based authorization. The generated JWT token contains user details and roles, allowing for secure and controlled access to admin functionalities. This streamlined process enhances application security and facilitates seamless integration for developers. The mechanism establishes a reliable and efficient workflow for user management and authorization within the system.

The provided code defines the security configuration for an application using Spring Security. It establishes a SecurityFilterChain bean, configuring various aspects of the HTTP security settings. Cross-Site Request Forgery (CSRF) protection is disabled to accommodate stateless authentication using JWT tokens. Authorization rules are specified for different URL patterns, allowing unrestricted access to the registration API ("/api/v1/auth/register") and restricting access to endpoints under "/api/v1/admin/" and "/api/v1/user/" based on user roles. The configuration enforces a stateless session management policy and incorporates a custom authentication provider for user authentication. Additionally, a custom JWT authentication filter is added to the security chain before the standard UsernamePasswordAuthenticationFilter. Overall, this configuration ensures secure access control and authentication for different API endpoints within the application. Developers can extend and customize these settings to suit the specific security requirements of their application.

*@Bean*

*public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {*

*http.*

*csrf()*

*. disable ()*

*.authorizeHttpRequests()*

*.requestMatchers("/api/v1/auth/register")*

*.permitAll()*

*. requestMatchers("/api/v1/admin/\*\*"). hasAuthority(Role.ADMIN.name())*

*. requestMatchers("/api/v1/user/\*\*"). hasAuthority(Role.USER.name())*

*.anyRequest()*

*. authenticated ()*

*.and ()*

*.sessionManagement()*

*.sessionCreationPolicy(SessionCreationPolicy.STATELESS)*

*.and ()*

*.authenticationProvider(authenticationProvider)*

*.addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class)*

*return http.build();*

*}*