JWT Authentication

Main Components:

@Service  
public class AuthFacadeService {  
 // Facade pattern: provides simplified interface for auth operations  
 // Handles:  
 // - User registration  
 // - User login  
 // - Error handling  
 // - Logging  
}  
  
@Service  
public class AuthService {  
 // Core authentication logic:  
 // - User registration business logic  
 // - User authentication verification  
 // - Password handling  
}  
  
public class JwtUtil {  
 // JWT token operations:  
 // - Token creation  
 // - Token validation  
 // - Token parsing  
}

Security Configuration:

@Configuration  
@EnableWebSecurity  
public class SecurityConfig {  
 // Configures security rules:  
 // - URL patterns to protect  
 // - Authentication manager  
 // - Password encoder  
 // - CORS settings  
 // - JWT filter registration  
   
 @Bean  
 public SecurityFilterChain filterChain(HttpSecurity http) {  
 return http  
 .csrf().disable()  
 .authorizeHttpRequests()  
 .requestMatchers("/auth/\*\*").permitAll()  
 .anyRequest().authenticated()  
 .and()  
 .sessionManagement()  
 .sessionCreationPolicy(SessionCreationPolicy.STATELESS)  
 .and()  
 .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class)  
 .build();  
 }  
}

Authentication Filter:

@Component  
public class JwtAuthenticationFilter extends OncePerRequestFilter {  
 private final JwtUtil jwtUtil;  
   
 @Override  
 protected void doFilterInternal(  
 HttpServletRequest request,  
 HttpServletResponse response,  
 FilterChain filterChain) {  
 // 1. Extract JWT from Authorization header  
 // 2. Validate token  
 // 3. Set authentication in SecurityContext  
 // 4. Continue filter chain  
 }  
}

Data Transfer:

public class UserDTO {  
 // Data Transfer Object  
 // Contains:  
 // - username  
 // - password  
 // Used for transferring data between layers  
}

Model:

public class User {  
 // Entity class  
 // Represents user in database  
 // Contains user details like:  
 // - username  
 // - password (hashed)  
 // - other user information  
}

Flow of Operations (Registration):

1. Controller receives UserDTO  
2. AuthFacadeService.registerUser called  
3. AuthService.register processes registration  
4. User saved to database  
5. Response returned with created user

Flow of Operations (Login):

1. Controller receives UserDTO  
2. AuthFacadeService.loginUser called  
3. AuthService.authenticate verifies credentials  
4. JwtUtil creates JWT token  
5. Response returned with token

Key Features:

Error handling with try-catch

Structured logging with @Slf4j

JWT token-based authentication

Facade pattern for simplified interface

Constructor injection for dependencies

Response wrapping with ResponseEntity

Complete Authentication Flow:

Login Flow:  
1. Client → /auth/login with credentials  
2. AuthFacadeService validates credentials  
3. JwtUtil creates token  
4. Token returned to client  
  
Protected Resource Access:  
1. Client → API with JWT in header  
2. JwtAuthenticationFilter intercepts  
3. Validates token using JwtUtil  
4. If valid:  
 - Sets SecurityContext  
 - Allows request to proceed  
5. If invalid:  
 - Rejects with 401

JWT Login:

Token Creation:  
1. User successfully authenticates  
2. JwtUtil creates signed token with:  
 - Username  
 - Expiration  
 - Secret key  
  
Token Validation:  
1. JwtAuthenticationFilter extracts token  
2. JwtUtil validates:  
 - Signature  
 - Expiration  
 - Token structure

Security Component Interaction:

Client → JWT Filter → Security Config → Protected Resources  
 ↓  
 JWT Validation  
 ↓  
 Security Context Setup

Request Flow:  
1. Client makes request  
 ↓  
2. JwtAuthenticationFilter intercepts FIRST  
 - Checks for "Authorization: Bearer <token>" header  
 - If token exists, validates it using JwtUtil  
 - If valid, creates Authentication object  
 ↓  
3. SecurityConfig rules applied  
 - Checks if URL requires authentication  
 - Public URLs (/auth/\*\*) bypass security  
 - Protected URLs require valid Authentication  
 ↓  
4. If authenticated, request reaches your Controller  
 Then AuthFacadeService, AuthService etc.

Public Endpoint (/auth/login):  
POST /auth/login  
→ Bypasses security (permitAll)  
→ Goes straight to Controller  
→ AuthFacadeService creates JWT  
← Returns token to client  
  
Protected Endpoint (/api/data):  
GET /api/data with JWT header  
→ JwtAuthenticationFilter validates token  
→ SecurityConfig checks authorization  
→ If valid, allows access to endpoint  
→ If invalid, returns 401 Unauthorized

Registration Flow (/auth/register):

1. Request hits DispatcherServlet  
2. Security Filters check URL → permitAll  
3. Controller receives UserDTO  
4. AuthFacadeService.registerUser called  
5. AuthService.register:  
 - Checks if username exists in database  
 - Hashes password  
 - Creates new User entity  
 - Saves to database using UserRepository  
6. Response: saved User object

Login Flow (/auth/login):

1. Request hits DispatcherServlet  
2. Security Filters check URL → permitAll  
3. Controller receives UserDTO  
4. AuthFacadeService.loginUser called  
5. AuthService.authenticate:  
 - Retrieves user from database by username  
 - Verifies hashed password  
 - If matches: JwtUtil creates token  
 - If not: throws exception  
6. Response: JWT token

Protected api call

1. Request hits DispatcherServlet  
2. JwtAuthenticationFilter:  
 - Extracts token  
 - Validates using JwtUtil  
 - Gets username from token  
 - Loads user from database using UserRepository  
 - Creates Authentication with user details  
3. SecurityConfig verifies authorization  
4. If authenticated:  
 → Controller processes request  
 → May perform additional database operations  
5. If not authenticated:  
 → Returns 401