**JWT Token**

let's start **JWT implementation from scratch** for your full **DMS Spring Boot project**.

We’ll follow **step-by-step implementation** that works for:

* ✅ Swagger (already integrated)
* ✅ Login (to generate JWT)
* ✅ Securing all endpoints using JWT

**✅ Roadmap: Full JWT Implementation for DMS Project**

**🔐 Goal:**

Secure all endpoints except /authenticate using JWT token.

**✅ Phase 1: Setup JWT Authentication (Step-by-Step)**

**✅ Step 1: Create JWT Utility Class**

**JwtUtil.java** – this will generate and validate JWT tokens.

**✅ Step 2: Create a Custom UserDetailsServiceImpl**

Used by Spring Security to load user info from DB or hardcoded.

**✅ Step 3: Create JwtAuthenticationFilter.java**

Reads JWT from request header and authenticates the request.

**✅ Step 4: Create SecurityConfig.java**

Enables JWT filter and secures all endpoints.

**✅ Step 5: Create AuthenticationController.java**

Has the /authenticate API to return JWT token when user logs in.

**📁 Folder Structure (Suggested)**

com.dms

├── config

│ └── SecurityConfig.java

├── controller

│ └── AuthenticationController.java

├── filter

│ └── JwtAuthenticationFilter.java

├── service

│ └── UserDetailsServiceImpl.java

├── util

│ └── JwtUtil.java

**⏭️ Let’s Start with Step 1: JwtUtil.java**

Would you like me to generate and explain the JwtUtil.java class now?

Once that's done, we’ll move on to:  
→ UserDetailsServiceImpl.java  
→ JwtAuthenticationFilter.java  
→ SecurityConfig.java  
→ AuthenticationController.java

**Step 1: Create JwtUtil.java (JWT Utility Class)**

📁 **Location:**  
Put this in:

swift

CopyEdit

src/main/java/com/dms/util/JwtUtil.java

**📌 Purpose of JwtUtil**

This class will:

* 🔐 Generate JWT token when user logs in
* ✅ Validate token on every request
* 👤 Extract username from token

**✅ Full Working Code – JwtUtil.java**

java

CopyEdit

package com.dms.util;

import io.jsonwebtoken.Claims;

import io.jsonwebtoken.Jwts;

import io.jsonwebtoken.SignatureAlgorithm;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.stereotype.Component;

import java.util.Date;

import java.util.HashMap;

import java.util.Map;

import java.util.function.Function;

@Component

public class JwtUtil {

// 🔐 Secret key for signing the token (keep it private in real projects)

private final String SECRET\_KEY = "my\_secret\_key";

// ✅ Extract username from token

public String extractUsername(String token) {

return extractClaim(token, Claims::getSubject);

}

// ✅ Extract expiration

public Date extractExpiration(String token) {

return extractClaim(token, Claims::getExpiration);

}

// ✅ Extract any claim

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

final Claims claims = extractAllClaims(token);

return claimsResolver.apply(claims);

}

// ✅ Generate token

public String generateToken(UserDetails userDetails) {

Map<String, Object> claims = new HashMap<>(); // You can add custom claims here

return createToken(claims, userDetails.getUsername());

}

// ✅ Validate token

public boolean validateToken(String token, UserDetails userDetails) {

final String username = extractUsername(token);

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

}

// 🧠 Internal methods

private String createToken(Map<String, Object> claims, String subject) {

return Jwts.builder()

.setClaims(claims)

.setSubject(subject) // Username

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

.setExpiration(new Date(System.currentTimeMillis() + 1000 \* 60 \* 60 \* 10)) // 10 hrs validity

.signWith(SignatureAlgorithm.HS256, SECRET\_KEY)

.compact();

}

private Claims extractAllClaims(String token) {

return Jwts.parser()

.setSigningKey(SECRET\_KEY)

.parseClaimsJws(token)

.getBody();

}

private boolean isTokenExpired(String token) {

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

}

}

**✅ Summary:**

| **Method** | **Purpose** |
| --- | --- |
| generateToken() | Generates JWT using username |
| validateToken() | Checks if token is valid |
| extractUsername() | Pulls username from token |
| extractExpiration() | Gets expiration date |

**🔐 Note:**

In real applications, the secret key should be stored securely (e.g., environment variable or config server).

**✅ Step 2: Create UserDetailsServiceImpl.java**

📁 **Location:**

swift

CopyEdit

src/main/java/com/dms/securityInfo/UserDetailsServiceImpl.java

**📌 Why do we need this?**

Spring Security needs a way to:

* 🔍 Load user by username (during /authenticate)
* ✅ Compare user credentials and roles
* 🔐 Validate the JWT token later (using username)

This class implements:

java

CopyEdit

org.springframework.security.core.userdetails.UserDetailsService

It must override:

java

CopyEdit

loadUserByUsername(String username)

**✅ Full Code: UserDetailsServiceImpl.java**

We'll start with a **hardcoded user** (great for learning). Later you can connect it to DB via UserRepository.

java

CopyEdit

package com.dms.securityInfo;

import org.springframework.security.core.userdetails.User;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.core.userdetails.UserDetailsService;

import org.springframework.security.core.userdetails.UsernameNotFoundException;

import org.springframework.stereotype.Service;

import java.util.ArrayList;

@Service

public class UserDetailsServiceImpl implements UserDetailsService {

// You can inject UserRepository here if loading from DB

@Override

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

// ✅ Hardcoded user for learning/testing

if ("admin".equals(username)) {

return new User(

"admin",

"{noop}password", // ✅ No encoding, use {noop} for plain text

new ArrayList<>() // 👤 Empty roles/authorities for now

);

} else {

throw new UsernameNotFoundException("User not found: " + username);

}

}

}

**✅ What’s happening here:**

| **Part** | **Explanation** |
| --- | --- |
| @Service | Tells Spring this is a service bean |
| UserDetailsService | Spring interface to load user info |
| loadUserByUsername | Loads user (by username) at login or token validation |
| {noop} | Says the password is plain-text (you can use encoder later) |
| new ArrayList<>() | No roles added yet (for simplicity) |

**🧪 Test User Credentials**

* **Username:** admin
* **Password:** password

When you POST to /authenticate (we’ll build this soon), use the above credentials.

✅ **Step 2 Complete!**

**⏭️ Ready for Step 3?**

Next is 🔍 **Step 3: JwtAuthenticationFilter.java** — this class will:

* Read JWT from header
* Validate token
* Set Spring Security context

**Step 3: JwtAuthenticationFilter.java**

📁 **Location:**

swift

CopyEdit

src/main/java/com/dms/securityInfo/JwtAuthenticationFilter.java

**📌 Why do we need this?**

When a user makes a request to a secured endpoint:

* This **filter intercepts the request**
* Extracts the **JWT token from the Authorization header**
* Validates it using JwtUtil
* Sets the authenticated user in Spring Security’s context

Without this filter, Spring Security **won’t know** about the token.

**✅ Full Code: JwtAuthenticationFilter.java**

java

CopyEdit

package com.dms.securityInfo;

import jakarta.servlet.FilterChain;

import jakarta.servlet.ServletException;

import jakarta.servlet.http.HttpServletRequest;

import jakarta.servlet.http.HttpServletResponse;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

import org.springframework.security.core.context.SecurityContextHolder;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.core.userdetails.UserDetailsService;

import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;

import org.springframework.stereotype.Component;

import org.springframework.web.filter.OncePerRequestFilter;

import java.io.IOException;

@Component

public class JwtAuthenticationFilter extends OncePerRequestFilter {

@Autowired

private JwtUtil jwtUtil;

@Autowired

private UserDetailsService userDetailsService;

@Override

protected void doFilterInternal(HttpServletRequest request,

HttpServletResponse response,

FilterChain filterChain)

throws ServletException, IOException {

// 🔍 Step 1: Get token from header

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

String jwtToken = null;

String username = null;

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

jwtToken = authHeader.substring(7);

username = jwtUtil.extractUsername(jwtToken);

}

// 🔐 Step 2: Validate and set authentication

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

UserDetails userDetails = userDetailsService.loadUserByUsername(username);

if (jwtUtil.validateToken(jwtToken, userDetails)) {

UsernamePasswordAuthenticationToken authToken =

new UsernamePasswordAuthenticationToken(

userDetails,

null,

userDetails.getAuthorities()

);

authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

// ✅ Set the authentication in the context

SecurityContextHolder.getContext().setAuthentication(authToken);

}

}

// ⏭️ Move to next filter

filterChain.doFilter(request, response);

}

}

**🧠 What is OncePerRequestFilter?**

A Spring Security filter that runs **once per HTTP request** — perfect for JWT token checks.

**🔐 Responsibilities of this Filter**

| **Task** | **Description** |
| --- | --- |
| ✅ Extract JWT | From Authorization header |
| ✅ Extract username | Using JwtUtil.extractUsername() |
| ✅ Validate token | Against secret and expiry |
| ✅ Load user | From UserDetailsServiceImpl |
| ✅ Set Security context | So Spring knows who is logged in |

**✅ Step 3 Complete!**

Next step is the **final piece of configuration:**

**⏭️ Step 4: SecurityConfig.java – glue everything together**

**✅ Step 4: Create SecurityConfig.java**

📁 **Location:**

swift

CopyEdit

src/main/java/com/dms/securityInfo/SecurityConfig.java

**📌 Why do we need this?**

This class:

* Tells Spring Security what to **allow and protect**
* Configures **JWT filter**
* Allows /authenticate and Swagger access
* Secures all other endpoints

**✅ Full Code: SecurityConfig.java**

java

CopyEdit

package com.dms.securityInfo;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.config.annotation.authentication.configuration.AuthenticationConfiguration;

import org.springframework.security.config.annotation.method.configuration.EnableMethodSecurity;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.config.http.SessionCreationPolicy;

import org.springframework.security.web.SecurityFilterChain;

import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

@Configuration

@EnableWebSecurity

@EnableMethodSecurity

public class SecurityConfig {

private final JwtAuthenticationFilter jwtAuthenticationFilter;

public SecurityConfig(JwtAuthenticationFilter jwtAuthenticationFilter) {

this.jwtAuthenticationFilter = jwtAuthenticationFilter;

}

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http

.csrf().disable()

.authorizeHttpRequests(auth -> auth

.requestMatchers(

"/authenticate", // ✅ Public login endpoint

"/swagger-ui/\*\*", // ✅ Swagger UI

"/v3/api-docs/\*\*" // ✅ Swagger docs

).permitAll()

.anyRequest().authenticated() // 🔐 Secure all other endpoints

)

.sessionManagement(sess -> sess

.sessionCreationPolicy(SessionCreationPolicy.STATELESS) // 🔄 No session - JWT only

)

.addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.class); // 🛡 Add JWT filter

return http.build();

}

@Bean

public AuthenticationManager authenticationManager(AuthenticationConfiguration config) throws Exception {

return config.getAuthenticationManager();

}

}

**🧠 Key Annotations**

| **Annotation** | **Meaning** |
| --- | --- |
| @EnableWebSecurity | Enables Spring Security |
| @EnableMethodSecurity | Enables @PreAuthorize, etc. (optional but useful) |
| @Bean SecurityFilterChain | Replaces old WebSecurityConfigurerAdapter |
| @Bean AuthenticationManager | Needed for login (authenticate user) |

**🔐 Security Behavior After This:**

| **Endpoint** | **Access** |
| --- | --- |
| /authenticate | ✅ Public |
| /swagger-ui/\*\* | ✅ Public |
| All other APIs (like /documents/\*\*) | 🔐 Require JWT token in header |

✅ **Step 4 Complete!**

You're now fully secured — just need a way to **log in and get a token**.

**Step 5: AuthenticationController.java**

It will:

* Take username + password
* Authenticate the user
* Generate and return a JWT token

Ready to build the controller for login?

**Step 5: AuthenticationController.java**

📁 **Location:**

swift

CopyEdit

src/main/java/com/dms/securityInfo/AuthenticationController.java

**📌 Purpose:**

This controller:

* Accepts username + password
* Authenticates user using AuthenticationManager
* Generates JWT using JwtUtil
* Returns the token to the client

**✅ Full Code: AuthenticationController.java**

java

CopyEdit

package com.dms.securityInfo;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.ResponseEntity;

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.authentication.BadCredentialsException;

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping

public class AuthenticationController {

@Autowired

private AuthenticationManager authenticationManager;

@Autowired

private UserDetailsServiceImpl userDetailsService;

@Autowired

private JwtUtil jwtUtil;

@PostMapping("/authenticate")

public ResponseEntity<?> createAuthenticationToken(@RequestBody AuthRequest authRequest) {

try {

authenticationManager.authenticate(

new UsernamePasswordAuthenticationToken(

authRequest.getUsername(),

authRequest.getPassword()

)

);

} catch (BadCredentialsException e) {

return ResponseEntity.status(401).body("❌ Invalid username or password");

}

final UserDetails userDetails = userDetailsService.loadUserByUsername(authRequest.getUsername());

final String jwt = jwtUtil.generateToken(userDetails);

return ResponseEntity.ok(new AuthResponse(jwt));

}

}

**✅ 2 Supporting Classes: AuthRequest.java and AuthResponse.java**

**✅ Create AuthRequest.java**

java

CopyEdit

package com.dms.securityInfo;

public class AuthRequest {

private String username;

private String password;

// Getters and setters

public String getUsername() { return username; }

public void setUsername(String username) { this.username = username; }

public String getPassword() { return password; }

public void setPassword(String password) { this.password = password; }

}

**✅ Create AuthResponse.java**

package com.dms.securityInfo;

public class AuthResponse {

private String token;

public AuthResponse(String token) {

this.token = token;

}

public String getToken() {

return token;

}

}

**🧪 How to Test This**

**✅ Step 1: Run the Application**

**✅ Step 2: Send POST request to /authenticate with:**

json

CopyEdit

{

"username": "admin",

"password": "password"

}

**✅ Response:**

json

CopyEdit

{

"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."

}

**✅ Final Swagger Testing**

1. Open http://localhost:8080/swagger-ui/index.html
2. Click **Authorize 🔒**
3. Paste:

nginx

CopyEdit

Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

1. Now test any **secured endpoint** — token will be sent in header

✅ **Step 5 Complete! Your JWT setup is DONE and TESTABLE!**  
Would you like me to:

* Help connect this to DB instead of hardcoded users?
* Write a one-line summary of each class for interview prep?
* Start logging, Swagger improvements, or role-based access?

You're doing amazing — let’s keep going!

**JWT Authorization in DMS: Summary**

🔐 **What it does:**

* Protects all API endpoints by default.
* Only allows access to authenticated users who provide a **valid JWT token**.

**📥 If you send the token (in Authorization: Bearer <token>):**

* ✅ You can access secured endpoints (GET, POST, PUT, DELETE).
* ✅ You are authenticated and authorized as a valid user.

**🚫 If you don't send the token or send an invalid one:**

* ❌ The system sets the user as **anonymous**.
* ❌ You receive **403 Forbidden** or **401 Unauthorized**.
* ❌ You **cannot** access or manipulate any secured data.

**✅ Real-time Example from DMS:**

* 🔓 POST /authenticate → Public (no token needed)
* 🔐 GET /documents/{id} → Requires token
* 🔐 POST /documents → Requires token

**Exception Handling Overview – DMS Project**

🔷 1. Centralized Exception Handling

* Created a global exception handler using @ControllerAdvice.
* Captures all exceptions across the application in a structured and user-friendly format.

java

CopyEdit

@RestControllerAdvice

public class GlobalExceptionHandler {

...

}

🔷 2. Custom Exception: ResourceNotFoundException

* Used to handle missing resources like Documents by ID.
* Thrown from DocumentServiceImpl when a document is not found.

java

CopyEdit

public class ResourceNotFoundException extends RuntimeException {

public ResourceNotFoundException(String message) {

super(message);

}

}

Usage:

java

CopyEdit

Document document = documentRepository.findById(id)

.orElseThrow(() -> new ResourceNotFoundException("Document not found with id: " + id));

🔷 3. Validation Error Handling (@Valid)

* Validates incoming @RequestBody in controller using Bean Validation annotations.
* Uses @Valid in controller:

java

CopyEdit

@PostMapping

public ResponseEntity<Document> createDocument(@Valid @RequestBody Document document) {

...

}

* In GlobalExceptionHandler, catches and transforms validation errors:

java

CopyEdit

@ExceptionHandler(MethodArgumentNotValidException.class)

public ResponseEntity<List<FieldErrorResponse>> handleValidationErrors(MethodArgumentNotValidException ex) {

List<FieldErrorResponse> errors = ex.getBindingResult()

.getFieldErrors()

.stream()

.map(err -> new FieldErrorResponse(err.getField(), err.getDefaultMessage()))

.collect(Collectors.toList());

return new ResponseEntity<>(errors, HttpStatus.BAD\_REQUEST);

}

🔷 4. Validation Constraints on Model

Added constraints in Document class for cleaner input validation:

java

CopyEdit

@NotBlank(message = "File path is required")

private String filePath;

@Min(value = 10, message = "File size must be greater than 10")

private long fileSize;

@NotBlank(message = "Title is required")

private String title;

🔷 5. Validation Error Output (JSON)

Sample output when invalid data is submitted:

json

CopyEdit

[

{

"field": "filePath",

"message": "File path is required"

},

{

"field": "fileSize",

"message": "File size must be greater than 10"

}

]

✅ Benefits in Interviews:

* Shows you know clean exception handling.
* Demonstrates use of Spring best practices.
* Helps build robust and user-friendly APIs.
* Shows ability to write custom reusable exceptions.

**What’s Done So Far in Exception Handling:**

| Feature | Status |
| --- | --- |
|  |  |
| Global Exception Handling | ✅ Implemented |
| Custom ResourceNotFoundException | ✅ Done |
| Structured Validation Errors | ✅ Done |
| Integrated with Swagger & Postman | ✅ Working |

You’ve successfully completed:

✔️ JWT Authentication  
✔️ Role-Based Authorization  
✔️ Model enhancement (with validation)  
✔️ Custom error handling  
✔️ Secure CRUD APIs

Here are a few things we can implement now:

1. **Role-Based Authorization** (Admin/User roles with access control)
2. **Audit Logging** (Track who created/updated what & when)
3. **File Upload Support** (Multipart + file storage with metadata)
4. **Pagination & Filtering** for GET endpoints
5. **Soft Delete / Archive feature** instead of hard delete
6. **Dockerize the App** (make it deployable)
7. **Unit & Integration Tests** (with JUnit & Mockito)
8. **Email Notifications** on document upload/update