# Spring Boot Authentication System Development Journey

## Overview

This document chronicles the development of a Spring Boot-based authentication system for the “SmartMunimji” application, focusing on JWT-based security, customer management, and database integration. It details the transformation of the code from the initial setup to the final version, addressing errors encountered, explaining key concepts, and providing a workflow explanation with the significance of each component.

## Initial Setup

The project began with a requirement to create a secure Spring Boot application with the following specifications:

* **Database**: MySQL database named smartmunimji\_db with a customers table.
* **Credentials**: Username smartmunim, password smartmunimji.
* **Security**: JWT-based authentication with role-based access control (ROLE\_CUSTOMER, ROLE\_ADMIN).
* **Files**: Security-related classes (JwtUtil, JwtFilter, SecurityConfig), a customer entity, a controller, and a repository.

### Initial Code

The initial code included three security classes and a basic configuration:

1. **JwtUtil.java**:
   * Purpose: Generates and validates JWT tokens.
   * Issues: Incorrect import of List (org.hibernate.mapping.List instead of java.util.List), potential null pointer risks in token validation, and hardcoded JWT secret.
2. **JwtFilter.java**:
   * Purpose: Intercepts HTTP requests to validate JWT tokens.
   * Issues: Insecure Bearer token parsing (no space after “Bearer”), missing null checks.
3. **SecurityConfig.java**:
   * Purpose: Configures Spring Security for authentication and authorization.
   * Issues: Outdated AuthenticationManager configuration, unused UserDetailsService field causing warnings.

The customers table schema was provided, but no entity or repository was initially included. A controller was later requested.

## Transformation Journey

### Step 1: Fixing Initial Security Classes

The first iteration focused on correcting errors in the security classes.

#### Changes Made

* **JwtUtil**:
  + Corrected List import to java.util.List.
  + Added default values for @Value annotations (jwtExpiration, jwtSecret) to prevent null issues.
  + Improved token validation with try-catch to return null for invalid tokens.
  + Used Collectors.joining for role concatenation.
* **JwtFilter**:
  + Fixed Bearer token parsing by checking for "Bearer " (with space) and using substring(7).
  + Added null checks for authHeader.
* **SecurityConfig**:
  + Updated to use AuthenticationConfiguration for modern AuthenticationManager setup.
  + Configured stateless session management and role-based access control.

#### New Files

* **application.properties**:
  + Configured database connection (smartmunimji\_db, smartmunim/smartmunimji).
  + Set JWT properties (expiration.millis, secret).
* **CustomerController.java**:
  + Added a REST controller with /api/authenticate for login and endpoints for customer (/cust/profile) and admin (/admin/dashboard).

#### Errors Resolved

* **Incorrect List Import**: Fixed by using java.util.List, ensuring proper collection handling.
* **Null Pointer Risks**: Added default values and null checks.
* **Bearer Token Parsing**: Corrected substring logic to handle standard Bearer tokens.

### Step 2: Creating the Customer Entity

The next step was to create a JPA entity for the customers table based on the provided SQL schema.

#### Initial Entity

* **Customer.java**:
  + Mapped all fields (id, name, email, etc.) with JPA annotations.
  + Implemented UserDetails for Spring Security integration.
  + Used LocalDateTime for timestamps with @PrePersist and @PreUpdate.

#### Issues

* The initial entity used manual getters/setters, which was later updated to use Lombok as per user requirements.

#### Changes Made

* Updated to use specified imports and Lombok annotations (@NoArgsConstructor, @AllArgsConstructor, @Getter, @Setter).
* Added @JsonIgnore on sensitive fields (password) and UserDetails methods.
* Simplified role assignment with AuthorityUtils.createAuthorityList.

### Step 3: Resolving getId() Error

An error was reported: “The method getId() is undefined for the type Customer.”

#### Cause

* The JwtUtil class called getId() on Customer, but there was a concern that Lombok’s @Getter might not be generating the method due to misconfiguration or IDE issues.

#### Solution

* **Customer.java**:
  + Added an explicit getId() method alongside @Getter to ensure compatibility.
  + Verified Lombok setup instructions (dependency, IDE plugin, mvn clean install).
* **JwtUtil**:
  + Confirmed compatibility with the updated Customer entity.

#### Errors Resolved

* **getId() Undefined**: Ensured getId() was available via explicit method and Lombok.
* **Lombok Issues**: Provided setup instructions to prevent annotation processing errors.

### Step 4: Resolving userDetailsService Warning

A warning was reported: “The value of the field SecurityConfig.userDetailsService is not used.”

#### Cause

* The userDetailsService was autowired but not explicitly used in SecurityConfig, as Spring Security implicitly used it for authentication.

#### Solution

* **SecurityConfig.java**:
  + Explicitly configured userDetailsService in the authenticationManager bean using authBuilder.userDetailsService(userDetailsService).
  + Updated to use AuthenticationManagerBuilder for clarity.
* **CustomerUserDetailsService.java**:
  + Created a UserDetailsService implementation to load Customer entities by email.
* **CustomerRepository.java**:
  + Added a JPA repository with findByEmail to support user loading.

#### Errors Resolved

* **Unused Field Warning**: Explicitly used userDetailsService in authenticationManager.
* **Missing UserDetailsService**: Provided a concrete implementation to load users from the database.

## Final Code Structure

The final application consists of the following files:

1. **JwtUtil.java**:
   * Generates and validates JWT tokens.
   * Uses Customer.getId() as the token subject.
   * Handles roles with comma-separated strings.
2. **JwtFilter.java**:
   * Intercepts requests to extract and validate JWT tokens.
   * Sets Authentication in SecurityContextHolder.
3. **SecurityConfig.java**:
   * Configures Spring Security with stateless sessions, role-based access, and JWT filtering.
   * Explicitly uses userDetailsService for authentication.
4. **Customer.java**:
   * JPA entity mapping the customers table.
   * Implements UserDetails with ROLE\_CUSTOMER.
5. **CustomerController.java**:
   * Handles authentication and role-specific endpoints.
6. **CustomerUserDetailsService.java**:
   * Loads Customer entities by email for authentication.
7. **CustomerRepository.java**:
   * Provides database access for Customer entities.
8. **application.properties**:
   * Configures database and JWT settings.

## Key Concepts and Their Significance

### 1. JWT Authentication

* **Concept**: JSON Web Tokens (JWT) are used for stateless authentication. A token contains claims (e.g., user ID, roles) signed with a secret key.
* **Components**:
  + **JwtUtil**: Generates tokens with user ID and roles, validates tokens to create Authentication objects.
  + **JwtFilter**: Extracts tokens from Authorization headers, validates them, and sets the security context.
* **Significance**: Enables secure, stateless authentication without server-side session storage, ideal for REST APIs.

### 2. Spring Security

* **Concept**: A framework for securing Spring applications with authentication and authorization.
* **Components**:
  + **SecurityConfig**: Defines security rules (e.g., permit /api/authenticate, restrict /api/cust/\*\* to ROLE\_CUSTOMER).
  + **UserDetailsService**: Loads user details for authentication.
* **Significance**: Provides robust security with role-based access control and integration with JWT.

### 3. JPA and Entity

* **Concept**: Java Persistence API (JPA) maps Java objects to database tables.
* **Components**:
  + **Customer**: Maps the customers table with fields like id, email, and timestamps.
  + **CustomerRepository**: Provides CRUD operations and custom queries (e.g., findByEmail).
* **Significance**: Simplifies database interactions with object-oriented programming.

### 4. Lombok

* **Concept**: A library that reduces boilerplate code with annotations like @Getter, @Setter.
* **Usage**: Used in Customer to generate getters/setters.
* **Significance**: Improves code readability and maintainability.

### 5. UserDetails and UserDetailsService

* **Concept**: UserDetails represents a user for Spring Security; UserDetailsService loads users.
* **Components**:
  + **Customer**: Implements UserDetails with email as username and ROLE\_CUSTOMER.
  + **CustomerUserDetailsService**: Loads Customer by email.
* **Significance**: Bridges JPA entities with Spring Security authentication.

## Workflow Explanation

### Authentication Flow

1. **User Login**:
   * Client sends POST request to /api/authenticate with email and password (via AuthRequest in CustomerController).
   * AuthenticationManager authenticates using CustomerUserDetailsService to load the Customer by email.
   * If valid, JwtUtil.createToken generates a JWT with the customer’s id and ROLE\_CUSTOMER.
2. **Token Validation**:
   * Client includes JWT in Authorization: Bearer <token> header for subsequent requests.
   * JwtFilter extracts the token, calls JwtUtil.validateToken to create an Authentication object, and sets it in SecurityContextHolder.
3. **Authorization**:
   * Spring Security checks the Authentication object’s roles against endpoint rules (e.g., /api/cust/\*\* requires ROLE\_CUSTOMER).
   * If authorized, the request proceeds to the controller (e.g., CustomerController.getProfile).

### Function Details and Significance

#### JwtUtil

* **createToken(Authentication)**:
  + **Function**: Generates a JWT with user ID as subject, roles as a claim, and signs it with jwtSecret.
  + **Significance**: Creates a secure token for client authentication.
* **validateToken(String)**:
  + **Function**: Parses JWT, verifies signature, and creates an Authentication object with user ID and roles.
  + **Significance**: Ensures only valid tokens grant access.

#### JwtFilter

* **doFilterInternal(HttpServletRequest, HttpServletResponse, FilterChain)**:
  + **Function**: Extracts JWT from headers, validates it, and sets Authentication in the security context.
  + **Significance**: Enables per-request authentication without session storage.

#### SecurityConfig

* **passwordEncoder()**:
  + **Function**: Provides BCryptPasswordEncoder for password hashing.
  + **Significance**: Secures stored passwords.
* **authenticationManager(AuthenticationConfiguration)**:
  + **Function**: Configures authentication with UserDetailsService and PasswordEncoder.
  + **Significance**: Enables user authentication via database.
* **securityFilterChain(HttpSecurity)**:
  + **Function**: Defines security rules, disables CSRF, and adds JwtFilter.
  + **Significance**: Enforces access control and stateless security.

#### Customer

* **getUsername(), getPassword(), getAuthorities()**:
  + **Function**: Implement UserDetails methods for Spring Security.
  + **Significance**: Integrates customer data with authentication.
* **onCreate(), onUpdate()**:
  + **Function**: Set timestamps on entity persistence.
  + **Significance**: Tracks creation and update times.

#### CustomerUserDetailsService

* **loadUserByUsername(String)**:
  + **Function**: Loads Customer by email from the database.
  + **Significance**: Enables database-driven authentication.

#### CustomerRepository

* **findByEmail(String)**:
  + **Function**: Queries customers table by email.
  + **Significance**: Supports user lookup for authentication.

#### CustomerController

* **authenticate(AuthRequest)**:
  + **Function**: Authenticates users and returns a JWT.
  + **Significance**: Provides the login endpoint.
* **getProfile(), getAdminDashboard()**:
  + **Function**: Role-specific endpoints for customers and admins.
  + **Significance**: Demonstrates role-based access.

## Error Resolution Summary

1. **Incorrect List Import in JwtUtil**:
   * Fixed by using java.util.List.
   * Lesson: Always verify import statements to avoid conflicts.
2. **getId() Undefined in Customer**:
   * Added explicit getId() and verified Lombok setup.
   * Lesson: Ensure Lombok is configured or use explicit methods for critical fields.
3. **userDetailsService Unused Warning**:
   * Explicitly used in authenticationManager.
   * Lesson: Explicit configuration clarifies intent and avoids warnings.
4. **Bearer Token Parsing**:
   * Corrected substring logic in JwtFilter.
   * Lesson: Follow HTTP header standards for token extraction.

## Final Workflow

1. **Setup**: Database (smartmunimji\_db) is initialized with the customers table. Application starts with configurations in application.properties.
2. **User Registration** (assumed external): A customer is created in the database with a hashed password.
3. **Login**: User sends credentials to /api/authenticate. CustomerUserDetailsService loads the user, AuthenticationManager verifies, and JwtUtil generates a token.
4. **Protected Access**: User sends requests with JWT. JwtFilter validates the token, and Spring Security enforces role-based access.
5. **Response**: Authorized requests reach controllers, returning appropriate responses.

## Conclusion

The SmartMunimji authentication system evolved from a basic setup with errors to a robust, secure application. By addressing import issues, Lombok configuration, unused field warnings, and token parsing errors, the code became production-ready. Each component (JwtUtil, JwtFilter, SecurityConfig, Customer, etc.) plays a critical role in ensuring secure, stateless authentication with role-based access control, integrated with a MySQL database. This journey highlights the importance of iterative development, error resolution, and clear configuration in building secure Spring Boot applications.