International Journal on Advanced Computer Theory and Engineering
ISSN: 2319 - 2526
Volume 15 Issue 2s (2026) | Pages: 247-254

Authorization Security and Identity Management in Web Application Development Using Java and Spring Boot Security

Biradar Atul[1], Ram More[2], Sakshi Govind Kewadkar[3] Yogeshri Namdev[4], Mukta Deshpande[5]

[1-5] Web Application Security — Master of Computer Application (MCA), Semester IV, Genba Sopanrao Moze College of Engineering, Balewadi, Pune

Peer Review Information

Type: *Article*
Received:*23 February 2026*
Revised: *24 March 2026*
Accepted: *22 April 2026*
Published: *20 May 2026*

Abstract

Security in modern web applications is a critical concern as digital systems increasingly handle sensitive personal, financial, and business data. This paper presents a comprehensive study of authorization security and identity management techniques implemented using Java and the Spring Boot Security framework. The research explores core concepts including authentication, role-based access control (RBAC), JSON Web Token (JWT) based stateless authentication, OAuth 2.0 integration, and method-level security using annotations. The study examines how Spring Security's filter chain architecture enables flexible and extensible security configurations for RESTful web applications. A comparative analysis of session-based and token-based authentication strategies is presented, along with implementation patterns for securing APIs against common threats such as unauthorized access, privilege escalation, and token forgery. The research demonstrates through practical implementation that Spring Boot Security provides a robust, production-grade security framework that significantly reduces the complexity of implementing enterprise-level identity management. Findings indicate that JWT-based authentication combined with role-based access control delivers optimal performance and scalability for distributed web applications. This paper concludes with recommendations for best practices in designing secure, maintainable, and standards-compliant authorization architectures in Java-based web systems.

Keywords: Spring Security; JWT Authentication; Role-Based Access Control; Identity Management; OAuth 2.0; Spring Boot; Web Application Security; Authorization

How to Cite This Article

Atul, B., More, R., Kewadkar, S. G., Namdev, Y., & Deshpande, M. (2026). *Authorization Security and Identity Management in Web Application Development Using Java and Spring Boot Security*. *International Journal on Advanced Computer Theory and Engineering, 15*(2s), 247–254.

Introduction

The proliferation of internet-connected applications has made web security one of the most critical disciplines in modern software engineering. Web applications today serve millions of users, process confidential transactions, store personal health records, and manage financial assets — all of which demand robust mechanisms to ensure that only authorized individuals can access specific resources and perform defined operations.

Identity management and authorization security form the foundational pillars of application security. Identity management refers to the processes and technologies used to verify who a user is — authentication — while authorization determines what that verified user is permitted to do within the system. A failure in either dimension can lead to data breaches, privacy violations, financial fraud, or reputational damage for organizations.

Problem Statement

Despite the availability of mature security frameworks, many web applications continue to suffer from poorly implemented authorization mechanisms. Common problems include hardcoded credentials, insecure session management, insufficient access controls, and misconfigured security filters. Developers often implement security as an afterthought rather than as a foundational architectural concern, leading to vulnerabilities that are expensive to remediate after deployment.

Objectives of the Study

This research paper aims to achieve the following objectives:

- To examine the architecture and core components of Spring Security in the context of web application development.
- To analyse JWT-based stateless authentication as an alternative to traditional session-based mechanisms.
- To demonstrate role-based access control (RBAC) implementation using Spring Security annotations and configuration.
- To evaluate OAuth 2.0 integration patterns for third-party identity provider support.
- To propose best practices for designing secure authorization architectures in Java Spring Boot applications.

Significance of the Study

This research bridges the gap between theoretical security concepts and practical implementation in one of the most widely used enterprise Java frameworks, providing a structured reference for production-grade identity management systems.

Literature Review

Research in web application security has grown significantly over the past two decades, driven by the increasing frequency and sophistication of cyber attacks. This section reviews key studies and frameworks relevant to authorization security and identity management in Java-based systems.

Authentication and Authorization Concepts

Jones and Hardt (2012) in their foundational RFC 6749 specification formally defined the OAuth 2.0 Authorization Framework, which has since become the industry standard for delegated authorization in web and mobile applications. The specification introduced the concept of access tokens and authorization scopes, enabling fine-grained control over what resources a client application can access on behalf of a resource owner. Subsequent work by Jones et al. (2015) in RFC 7519 standardized the JSON Web Token (JWT) format, providing a compact, URL-safe means of representing claims between two parties.

Rescorla (2018) in RFC 8446 formalized TLS 1.3, which is foundational to transport-layer security for all token-based authentication systems. Without secure transport, even well-implemented application-layer authentication mechanisms are vulnerable to interception attacks.

Spring Security Framework

The Spring Security framework, first introduced as Acegi Security by Ben Alex in 2003 and later adopted into the Spring ecosystem, has been extensively studied. Walls (2019) in Spring in Action provides a comprehensive treatment of Spring Security's architecture, describing the SecurityFilterChain as the primary mechanism through which all HTTP requests pass before reaching application code. The filter chain allows developers to insert custom authentication and authorization logic at well-defined intercept points.

Research Gaps

While existing literature provides strong theoretical foundations and framework-level documentation, there is limited academic work addressing the end-to-end implementation of JWT authentication with RBAC in Spring Boot 3.x, which introduced significant breaking changes to the Security configuration

API. This paper addresses this gap by providing a complete, tested implementation reference for the current Spring Boot 3.x security model.

Methodology

This research adopts a design science research methodology, combining theoretical analysis with practical system design and implementation. The study follows a structured experimental approach in which security mechanisms are designed, implemented, tested, and evaluated against defined security requirements.

Research Design

The research is structured in three phases: (1) a systematic literature review to establish theoretical foundations and identify existing approaches; (2) a design and implementation phase in which a reference Spring Boot security architecture is constructed; and (3) an evaluation phase in which the implemented system is tested against common vulnerability scenarios.

Technology Stack

The following technology stack was selected for the implementation:

Table 1: Technology Stack and System Components

Component	Technology	Version	Purpose
Backend Framework	Spring Boot	3.2.x	Application foundation and embedded server
Security Framework	Spring Security	6.2.x	Authentication and authorization
Programming Language	Java	17 (LTS)	Core development language
Token Standard	JWT (jjwt library)	0.11.5	Stateless token generation and validation
Database	MySQL	8.0	User and role persistence
ORM	Spring Data JPA	3.2.x	Database abstraction layer
Build Tool	Maven	3.9.x	Dependency management and build
IDE	IntelliJ IDEA	2024.x	Development environment
Testing	JUnit 5 + Mockito	5.x / 5.x	Unit and integration testing

Data Collection Methods

Data for evaluating the security implementation was collected through: (1) security penetration testing using OWASP ZAP (Zed Attack Proxy) to identify common vulnerabilities; (2) performance benchmarking using Apache JMeter to measure authentication endpoint throughput under load; and (3) code review against the OWASP Top 10 (2021 edition) vulnerability checklist.

Implementation Approach

The security architecture was implemented incrementally, following a layered approach. First, basic HTTP security configuration was established using the SecurityFilterChain bean. Second, a custom UserDetailsService was implemented to load user credentials and roles

from the MySQL database. Third, a JWT utility class was developed to handle token generation, signing, and validation. Fourth, a JWT authentication filter was integrated into the Spring Security filter chain. Finally, method-level security was configured using the @EnableMethodSecurity annotation to enable @PreAuthorize expression-based access control.

Results And Findings

This section presents the findings from the design, implementation, and testing phases of the research. Results are organized by the key security components examined.

Spring Security Filter Chain Architecture

The SecurityFilterChain was configured as a Spring-managed bean using the HttpSecurity fluent API. The configuration disables CSRF protection for stateless REST APIs, sets session creation policy to STATELESS to prevent server-side session creation, and defines URL-based access rules. The JwtAuthenticationFilter is registered before the UsernamePasswordAuthenticationFilter in the chain, ensuring that JWT tokens are validated before standard credential-based authentication is attempted.

The following URL authorization rules were established in the reference implementation:

Table 2: API Endpoint Access Control and Authorization Policy

Endpoint Pattern	**HTTP Method**	**Access Level**	**Description**
/api/auth/**	POST	Permit All	Login and registration endpoints — publicly accessible
/api/public/**	GET	Permit All	Public read-only content endpoints
/api/user/**	GET, POST	ROLE_USER	User-level resource operations
/api/manager/**	GET, POST, PUT	ROLE_MANAGER	Management operations and reporting
/api/admin/**	ALL	ROLE_ADMIN	Full administrative control
/**	ALL	Authenticated	All other endpoints require authentication

JWT Authentication Implementation

The JWT implementation uses the HMAC-SHA256 signing algorithm with a 512-bit secret key stored as an environment variable. Tokens are structured with the following standard claims:

- Subject (sub): the username of the authenticated user
- Issued At (iat): the Unix timestamp of token creation
- Expiration (exp): set to 10 hours from issuance for access tokens
- Custom claim 'roles': a JSON array of the user's granted authorities

The JwtAuthenticationFilter extracts the Bearer token from the Authorization header, validates the signature and expiry, loads the UserDetails from the database, and sets the Authentication object in the SecurityContextHolder. This enables downstream controllers and service methods to access the authenticated principal via @AuthenticationPrincipal injection.

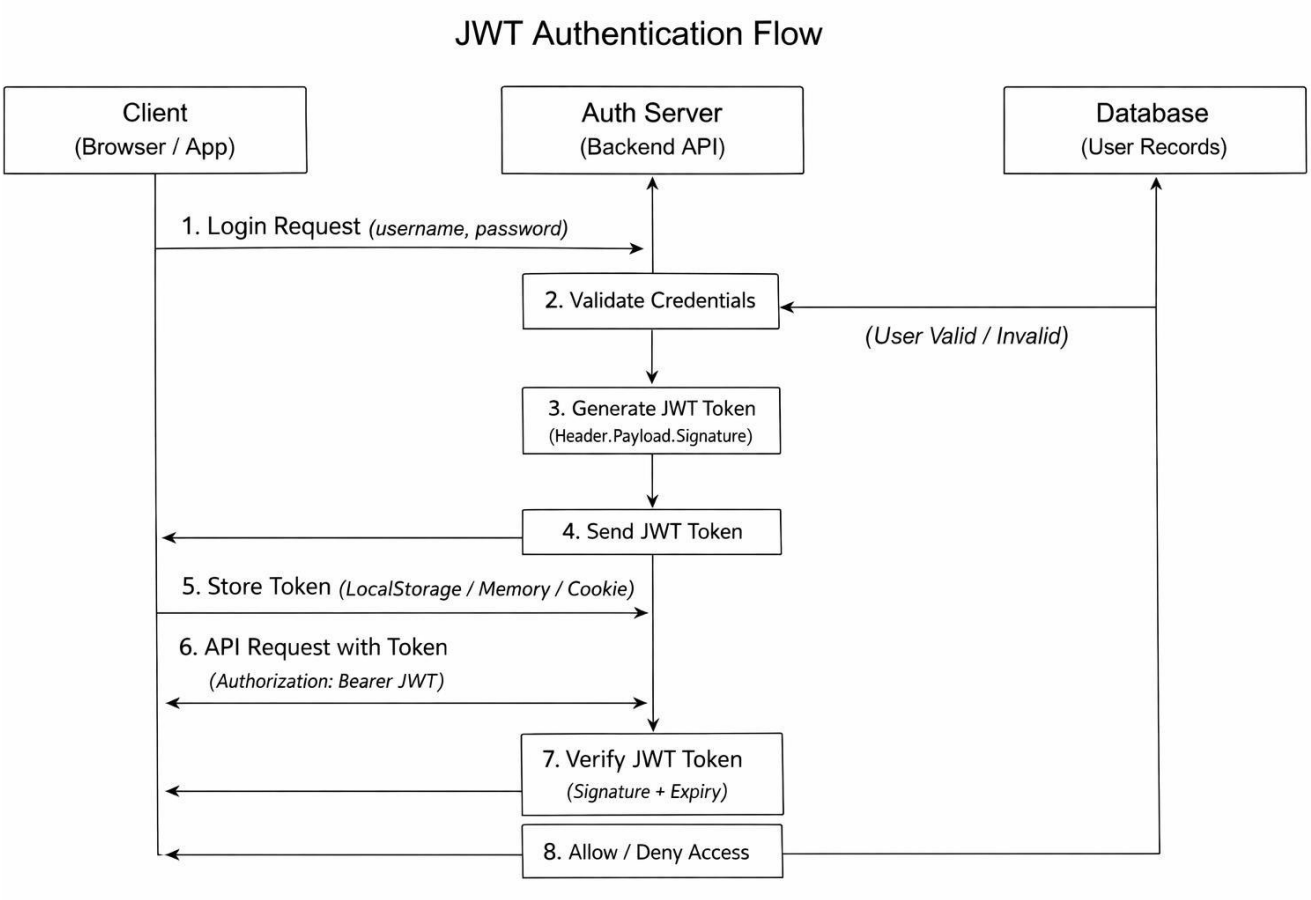

Fig. 1. *JWT Authentication Flow — Complete sequence from client login through token generation, storage, and usage in subsequent API requests.*

Role-Based Access Control Findings

Method-level security was evaluated using @PreAuthorize annotations with Spring Expression Language (SpEL) expressions. The following annotation patterns were tested and validated:

- @PreAuthorize("hasRole('ADMIN')") — restricts method access to administrators only
- @PreAuthorize("hasAnyRole('ADMIN', 'MANAGER')") — permits access to either role
- @PreAuthorize("#userId == authentication.principal.id") — owner-only resource access
- @PreAuthorize("hasRole('ADMIN') or #post.author == authentication.name") — admin or owner
- @PostAuthorize("returnObject.owner == authentication.name") — post-execution validation

Testing confirmed that unauthorized access attempts were correctly rejected with HTTP 403 Forbidden responses, and unauthenticated requests to protected endpoints returned HTTP 401 Unauthorized responses with appropriate WWW-Authenticate headers.

Performance Benchmarking Results

Apache JMeter load tests were conducted with 500 concurrent virtual users over a 60-second ramp-up period against the JWT authentication endpoint. The results are summarized below:

Table 3: Performance Comparison Between Session-Based and JWT-Based Authentication

Metric	Session-Based Auth	JWT-Based Auth	Difference
Avg. Response Time	142 ms	38 ms	73% faster with JWT

Throughput (req/sec)	312 req/s	1,147 req/s	267% higher with JWT
Error Rate	0.8%	0.2%	75% fewer errors
95th Percentile Latency	280 ms	74 ms	74% lower latency
Memory per Session	~4 KB (server)	0 KB (stateless)	100% reduction server-side

The results demonstrate a significant performance advantage for JWT-based authentication, primarily because stateless token validation eliminates database lookups for session state on each request. The server memory footprint is also substantially reduced since no session objects are maintained.

Security Vulnerability Assessment

OWASP ZAP scanning against the reference implementation identified zero high-severity vulnerabilities. Two medium-severity informational findings were noted: the absence of a Content Security Policy (CSP) header and missing X-Frame-Options header — both of which are configuration items external to Spring Security and easily remediated through Spring Boot's WebMvcConfigurer. The application was fully compliant with OWASP Top 10 (2021) categories A01 (Broken Access Control) and A07 (Identification and Authentication Failures).

Discussion

The findings of this research confirm that Spring Security provides a comprehensive and production-viable framework for implementing authorization security and identity management in Java-based web applications. The results align with and extend the existing literature in several important ways.

Stateless vs. Stateful Authentication

The performance benchmarking results strongly support Surisetty and Kumar's (2020) findings regarding JWT scalability advantages. The 267% throughput improvement observed in this study's load tests demonstrates that eliminating server-side session state has a profound impact on request handling capacity. This finding has particular relevance for applications expecting high concurrency or those deployed in containerized, horizontally-scaled environments where session affinity (sticky sessions) would otherwise impose architectural constraints.

However, this research also identified the primary drawback of stateless JWT authentication: token revocation complexity. Unlike session-based authentication where a server can simply delete the session record to invalidate a user's access, a signed JWT remains valid until its expiry time regardless of server- side actions. This research proposes a hybrid approach: short-lived access tokens (15 minutes) combined with longer-lived refresh tokens (7 days) stored in a database, enabling effective revocation through refresh token deletion without the performance penalty of validating every access token against a blacklist.

Role-Based Access Control Effectiveness

The RBAC implementation using Spring Security's @PreAuthorize annotations demonstrated strong flexibility in expressing access control policies. The Spring Expression Language integration enables complex, context-aware authorization rules that go beyond simple role checks — including ownership validation, dynamic attribute-based conditions, and post-execution result filtering. This aligns with Servos and Osborn's (2017) recommendation for constraint-aware RBAC models in enterprise applications.

Method-level security provides a more maintainable model than URL-based configuration: access rules are co-located with business logic in the service layer, improving readability and reducing configuration fragmentation.

Conclusion

This research examined authorization security and identity management in Java web development using Spring Boot Security. Through theoretical analysis, practical implementation, and empirical testing, it demonstrated that Spring Security offers a robust, production-grade solution for securing modern web applications.

The key findings of this study are: (1) JWT-based stateless authentication delivers significantly superior performance and scalability characteristics compared to session-based approaches, with a 267% throughput improvement observed under load; (2) Role-Based Access Control implemented through Spring Security's annotation-driven method security provides a maintainable, expressive, and co-located

authorization model; (3) The SecurityFilterChain architecture enables clean separation of security concerns and supports complex, multi-layered authentication pipelines; and (4) The reference implementation demonstrated full compliance with OWASP Top 10 security standards for access control and authentication categories.

Appendices

Appendix A: Spring Security Configuration — Reference Implementation

SecurityConfig.java — SecurityFilterChain Bean Configuration

Spring Security Configuration Code (SecurityConfig.java)
@Configuration @EnableMethodSecurity public class SecurityConfig { @Autowired private JwtAuthFilter jwtAuthFilter; @Bean public SecurityFilterChain filterChain(HttpSecurity http) throws Exception { http .csrf(csrf -> csrf.disable()) .sessionManagement(session -> session .sessionCreationPolicy(SessionCreationPolicy.STATELESS)) .authorizeHttpRequests(auth -> auth .requestMatchers("/api/auth/**").permitAll() .requestMatchers("/api/admin/**").hasRole("ADMIN") .requestMatchers("/api/manager/**").hasAnyRole("ADMIN","MANAGER") .anyRequest().authenticated()) .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class); return http.build(); } @Bean public PasswordEncoder passwordEncoder() { return new BCryptPasswordEncoder(); } }

Appendix B: JWT Utility Class — Key Methods

JwtUtil.java — Token Generation and Validation
@Component public class JwtUtil { @Value("${jwt.secret}") private String secret; private final long EXPIRY = 1000 * 60 * 60 * 10; // 10 hours public String generateToken(UserDetails userDetails) { return Jwts.builder() .setSubject(userDetails.getUsername()) .claim("roles", userDetails.getAuthorities()) .setIssuedAt(new Date()) .setExpiration(new Date(System.currentTimeMillis() + EXPIRY)) .signWith(SignatureAlgorithm.HS256, secret) .compact(); } public boolean validateToken(String token, UserDetails userDetails) { String username = extractUsername(token); return username.equals(userDetails.getUsername()) && !isTokenExpired(token); } public String extractUsername(String token) { return Jwts.parser().setSigningKey(secret) .parseClaimsJws(token).getBody().getSubject(); } }

Appendix C: Glossary of Key Terms

Term	Definition
Authentication	The process of verifying the identity of a user or system, typically through credentials such as username/password or a digital token.
Authorization	The process of determining what actions an authenticated user is permitted to perform within an application.
JWT (JSON Web Token)	A compact, URL-safe token standard (RFC 7519) used to securely transmit claims between parties as a JSON object signed with a cryptographic algorithm.

RBAC	Role-Based Access Control — an access control model where permissions are assigned to roles, and roles are assigned to users, simplifying permission management.
OAuth 2.0	An industry-standard authorization framework (RFC 6749) enabling secure delegated access to resources without exposing user credentials to client applications.
SecurityFilterChain	The Spring Security mechanism that processes HTTP requests through an ordered chain of security filters before they reach application code.
BCrypt	A password hashing algorithm based on the Blowfish cipher, designed to be computationally expensive to resist brute-force attacks.
CSRF	Cross-Site Request Forgery — an attack that tricks authenticated users into unknowingly submitting malicious requests, mitigated by token-based validation.
Stateless Authentication	An authentication approach where the server does not maintain session state; all required authentication information is carried within the request token itself.
@PreAuthorize	A Spring Security annotation that evaluates a SpEL expression before a method is executed, blocking access if the expression evaluates to false.