International Journal for Modern Trends in Science and Technology
Volume 12, Issue 05, pages 248-254
ISSN: 2455-3778 online
Available online at: http://www.ijmtst.com/vol12issue05.html
DOI: https://doi.org/10.5281/zenodo.19893156

Implementing Secure and Flexible User Authentication Mechanism with RBAC

V. Ajay Prakash, V. Jayavarshini, V. Hari Priya, V. Hareeshwarr Khumar, V. Harshitha Prasanna, P. Chaitanya

Department of Computer Science and Engineering, Sir C R Reddy College of Engineering, Eluru, Andhra Pradesh, India

To Cite this Article
V. Ajay Prakash, V. Jayavarshini, V. Hari Priya, V. Hareeshwarr Khumar, V. Harshitha Prasanna & P. Chaitanya (2026). Implementing Secure and Flexible User Authentication Mechanism with RBAC. International Journal for Modern Trends in Science and Technology, 12(05), 248-254. https://doi.org/10.5281/zenodo.19893156

Article Info
Received: 28 March 2026; Revised: 24 April 2026; Accepted: 26 April 2026.

KEYWORDS

Role-Based Access Control, JSON Web Token, Spring Security, BCrypt, Firebase Authentication, Biometric Multi-Factor Authentication, Privilege Escalation, Enterprise Security.

ABSTRACT

Access control failures rank among the most consequential security deficiencies in web-based enterprise software. Organisations operating multi-role platforms — where staff at different hierarchical levels interact with the same system — require authentication architectures that go beyond verifying identity to actively constraining what each verified identity may do. This paper presents the architecture, realisation, and experimental assessment of a three-tier Role-Based Access Control (RBAC) framework built on Spring Boot 3 and React.js 18. The framework assigns principals to one of three authority tiers — standard user, content moderator, and system administrator — and enforces the permissions of each tier through server-resident method-level annotations that are evaluated before any controller logic executes. Credential verification is offered through three independent paths: email-and-password with BCrypt adaptive hashing, federated sign-in via Google OAuth 2.0 delegated through Firebase, and phone-based one-time password. Session state is carried entirely within signed JSON Web Tokens, removing the need for server-side session storage and supporting stateless horizontal scaling. Principals assigned to the moderator and administrator tiers are subjected to a real-time facial biometric check as a mandatory second factor prior to dashboard access. The complete system is hosted on Render and Vercel with continuous delivery linked to a GitHub repository. Controlled experiments demonstrate that role boundaries resist direct API manipulation, cryptographic verification reliably detects tampered tokens, and per-request authentication overhead averages 15 milliseconds — a figure that imposes no perceptible penalty on end users.

1. INTRODUCTION

Securing the entry boundary of an enterprise web application is a problem that grows in complexity as the platform scales. A system accessed by ten users with broadly similar responsibilities can tolerate a coarse access model without significant risk. A system accessed by hundreds of users spanning operational, supervisory, and administrative functions cannot: a junior operator who can invoke administrator APIs represents a systematic vulnerability, irrespective of whether any current user intends to misuse that access. The OWASP community has repeatedly identified broken access control and identification failures as the two most prevalent critical findings across assessed deployments [3], and the academic literature consistently traces these failures to architectures that treat authorisation as an afterthought appended to an authentication check [1][2].

The gap between what authentication should achieve and what typical Spring Boot starter implementations actually deliver motivated this work. A standard auto-configured Spring Boot project authenticates callers against a user store but assigns every authenticated caller to one of two states — authenticated or not — with no middle ground. This binary model forces access differentiation into the browser layer, where it can be trivially bypassed by crafting direct HTTP calls to privileged endpoints. Addressing this requires relocating access enforcement to the server, embedding authority claims in the session credential, and implementing explicit guards at every sensitive endpoint.

This paper makes the following contributions. (i) A ternary RBAC model in which User, Moderator, and Administrator tiers carry distinct, non-overlapping permission sets enforced at the Spring controller layer. (ii) A unified authentication pipeline that accepts three credential types — email-password, Google OAuth 2.0, and SMS OTP — and produces an equivalent signed JWT regardless of the path taken. (iii) A facial biometric second factor mandatory for the two upper tiers, implemented in-browser via face-api.js without server-side camera infrastructure. (iv) Explicit constraints preventing lateral and vertical privilege escalation at the service layer, independent of any client-side control. The complete implementation is available at https://github.com/Hareeshwarr/rbac-auth-app.

The remainder of this paper is organised as follows. Section II reviews the existing authentication models against which this work is positioned and synthesises the relevant literature. Section III presents the proposed architecture in full detail. Section IV reports experimental results and discusses their implications. Section V concludes.

2. LITERATURE REVIEW

The RBAC model was formalised by Ferraiolo, Cugini, and Kuhn [1] as a structured approach to assigning permissions through role objects rather than directly to individuals, substantially simplifying administration in organisations with defined functional hierarchies. Suganthy and Prasanna Venkatesan [2] subsequently validated RBAC in organisational deployments and demonstrated that role-centric models outperform discretionary access control where staff responsibilities evolve over time. Nahar and Gill [10] extended this analysis to multi-application enterprise environments, identifying policy consistency and automated role lifecycle management as the critical unsolved challenges.

RFC 7519 [6] standardises the JSON Web Token format as a compact, self-verifying credential capable of carrying arbitrary claim sets. Its adoption as a session mechanism eliminates the server-side storage requirement of cookie-based sessions and directly supports horizontal scaling. Darmawan et al. [9] subjected JWT implementations to controlled penetration exercises and identified two recurring vulnerabilities: tokens stored in cookies rather than request headers, which reintroduces CSRF exposure, and the absence of token rotation, which extends the exploitability window of stolen credentials. Both findings directly informed design decisions in the present work.

Lopez et al. [11] surveyed the prevalence and consequences of password-only authentication, documenting that credential reuse across services — combined with the frequency of large-scale data breaches — renders single-factor authentication a known-broken model for high-value accounts. Stafford [12] articulated the Zero Trust principle that per-request verification must be the baseline assumption, rejecting implicit trust granted by prior authentication. The OWASP Cheat Sheet Series [3] provides concrete implementation guidance translating these principles into specific engineering controls, several of which are directly incorporated in the proposed system.

A review of the existing literature reveals a persistent gap: while RBAC theory and JWT mechanics are individually well-studied, implementations that combine a ternary role model, multi-method credential support, biometric second factors, and server-side escalation prevention within a single deployable system remain scarce in the published record. This paper addresses that gap.

3. EXISTING SYSTEM

Prevailing Authentication Patterns

The dominant authentication architecture in educational and small-business Spring Boot deployments follows a recognisable template: a UserDetailsService implementation loads a principal by username from a relational store, Spring Security compares the submitted password against the stored value, and on a match the framework creates a server-side session and returns a cookie bearing the session identifier. Role differentiation, where present, takes the form of a two-value enum — typically ROLE_USER and ROLE_ADMIN — with access restrictions encoded as URL patterns in the HttpSecurity configuration rather than as method-level annotations.

This design is adequate for small-scale, perimeter-protected deployments. Its deficiencies emerge in three distinct scenarios: when a platform must support graduated operational roles with finer permission boundaries; when the attack surface expands through internet exposure; and when horizontal scaling requires distributing session state across multiple server instances, necessitating either sticky routing or a shared cache.

TABLE I. Characteristics of the Existing Authentication Model

Characteristic	Typical Realisation	Security Implication
Role granularity	Binary: ROLE_USER / ROLE_ADMIN	No intermediate permission tier; over-privileged assignment common
Credential types	Username and password only	Single failure point; no fallback if password is compromised
Password storage	Plaintext or single-round MD5 / SHA-1	Reversible from a database dump within hours using GPU tooling
Session mechanism	Server-side object bound to cookie	Prevents stateless scaling; vulnerable to session fixation
Second factor	Absent	Credential possession equates unconditionally to account ownership
Endpoint guards	URL-pattern matching in security config	Method-level granularity absent; privilege escalation via direct URL call

4. PROPOSED SYSTEM

4.1 System Architecture

The proposed framework adopts a stateless, cloud-native architecture spanning three independently deployable units. The React.js single-page application, hosted on Vercel, serves as the presentation layer. The Spring Boot REST API, hosted on Render, encapsulates all business logic and security enforcement. The H2 relational database (MySQL in local development) provides persistence for user records, role definitions, and their many-to-many mapping. Firebase Authentication operates as an external identity delegation service for the Google OAuth 2.0 and SMS OTP paths; its involvement is confined to verifying a provider-issued credential, after which control returns entirely to the local system and a locally signed JWT is issued.Every inbound HTTP request to the API traverses a custom filter — Auth Token Filter, extending Once Per Request Filter — before any controller method is reached. The filter extracts the bearer token from the Authorization header, recomputes the HMAC-SHA256 signature using the server's secret key, compares it against the signature embedded in the token, and validates the expiry claim. A single verification failure terminates request processing with HTTP 401; no partial information is returned to the caller. On successful verification, the principal is loaded from the database by username and placed in Spring Security's Security Context. The controller method's @PreAuthorize expression is then evaluated against the populated context; if the principal's authority set does not satisfy the expression, Spring returns HTTP 403 before the method body executes.

4.2 Three-Tier RBAC Model

TABLE II. Role Definitions and Permission Boundaries

Role	Auth Paths Permitted	Second Factor	Scope of Access
ROLE_USER	Email-password, Google OAuth, SMS OTP	None	Own profile read; RBAC status display
ROLE_MODERATOR	Email-password, Google OAuth	Facial biometric (mandatory)	All USER operations; view, search, delete ROLE_USER accounts only
ROLE_ADMIN	Email-password only	Facial biometric (mandatory)	Full CRUD on all non-ADMIN accounts; role reassignment; cannot delete peer ADMIN accounts

The restriction of ROLE_ADMIN authentication to the email-and-password path is a deliberate architectural choice. Federated identity providers introduce a third-party trust dependency; a compromise of the

provider or a misconfiguration of the OAuth callback constitutes an indirect path to the most privileged tier. Eliminating that dependency for administrator accounts reduces the effective attack surface of the highest-risk credential class. The prohibition on administrator accounts deleting peer administrator accounts is encoded as a service-layer constraint rather than a UI restriction, ensuring it cannot be bypassed through direct API access.

4.3 JSON Web Token Design

Session continuity is maintained through JWTs signed with HMAC-SHA256. The token payload carries the subject claim (username), a custom roles claim containing the principal's authority strings, and the standard iat and exp claims. The validity window is set to 86 400 seconds. This interval is long enough to span a standard working session without forcing re-authentication, while remaining short enough to limit the exposure window should a token be intercepted. The token is transmitted exclusively in the HTTP Authorization header as a Bearer credential, a choice that eliminates the CSRF attack surface present in cookie-based session schemes [9]. The JwtUtils class handles token construction, signing, and validation; the secret key is externalised to the server environment rather than embedded in the codebase.

4.4 Authentication Processing

4.4.1 Email-and-Password Path: Credentials arrive at POST /api/auth/signin. The AuthenticationManager delegates verification to BCrypt, which recomputes the digest from the submitted plaintext and compares it against the stored hash. The BCrypt work factor of 10 produces a computation time of approximately 200 milliseconds per attempt — a negligible delay for a legitimate user signing in once, but a substantial barrier against automated dictionary attacks. On a positive comparison, JwtUtils constructs and signs the token; the JwtResponse is returned with the token, user identifier, email, and role list.

4.4.2 Google OAuth 2.0 Path: The Firebase JavaScript SDK manages the OAuth 2.0 authorisation code exchange with Google's identity provider. The resulting Firebase ID token is forwarded to POST /api/auth/google; Firebase Admin SDK verifies the token server-side. The backend locates or creates the corresponding local user record, marks the auth provider field as 'google', and issues a locally signed JWT through the standard issuance pipeline.

4.4.3 SMS OTP Path:

OTP generation, delivery, and verification are handled entirely by Firebase. The verified Firebase credential token is exchanged at POST /api/auth/phone for a locally issued JWT, making the output indistinguishable from the email-password and OAuth paths — a deliberate decision that ensures all three authentication methods produce tokens of identical strength and structure.

4.4.4 Facial Biometric Verification:

For ROLE_MODERATOR and ROLE_ADMIN principals, dashboard access is gated behind a biometric check executed in the browser using face-api.js. The TinyFaceDetector model locates the face in the live webcam feed; the FaceNet descriptor network produces a 128-dimensional embedding of the detected face. This embedding is compared against a reference descriptor captured at registration using Euclidean distance; a distance below 0.6 constitutes a match. Successful verification sets a flag in React's AuthContext; the ProtectedRoute component evaluates this flag before rendering any dashboard content. Direct URL navigation without completing verification is blocked at both the client route guard and the server API level.

4.5 Escalation Prevention

Beyond the role guards provided by @PreAuthorize, two explicit service-layer constraints prevent specific cross-tier operations. An administrator invoking DELETE /api/admin/users/{id} triggers a server-side inspection of the target principal's role set before any database operation is initiated; if the target carries ROLE_ADMIN the operation is rejected with HTTP 400. A moderator invoking DELETE /api/mod/users/{id} is blocked from deleting any principal whose role set includes ROLE_MODERATOR or ROLE_ADMIN. Both constraints are evaluated in the service class, not in the controller, ensuring they cannot be circumvented by constructing a request that satisfies the @PreAuthorize expression but targets a principal outside the caller's permitted scope.

5. RESULTS AND DISCUSSIONS

5.1 Experimental Setup

Evaluation was conducted against the production deployment: the Spring Boot API hosted on Render (US East region) and the React SPA hosted on Vercel. API-level test cases were executed using Postman to permit precise control over request headers and token payloads. End-to-end flows were exercised through the browser. Response latencies were measured as ten-sample means recorded from a client located in Andhra Pradesh, India, incorporating real transatlantic network round-trip times. The H2 in-memory database was pre-populated with accounts at each of the three role tiers before testing commenced.

5.2 Functional and Role Boundary Results

TABLE III. Functional Test Outcomes

Operation	Input Condition	Expected Response	Observed	Status
Account registration	Unique username and email	HTTP 201	HTTP 201	✓
Account registration	Duplicate username	HTTP 400	HTTP 400	✓
Authentication	Correct credentials	JWT in HTTP 200 body	HTTP 200 + JWT	✓
Authentication	Incorrect password	HTTP 401	HTTP 401	✓
Protected endpoint	Valid token in header	HTTP 200	HTTP 200	✓
Protected endpoint	No Authorization header	HTTP 401	HTTP 401	✓
Role guard — insufficient role	ROLE_USER → /api/admin	HTTP 403	HTTP 403	✓
Peer admin deletion	ROLE_ADMIN → admin target	HTTP 400	HTTP 400	✓
Tampered token payload	Role claim escalated	HTTP 401	HTTP 401	✓
Expired token	exp in past	HTTP 401	HTTP 401	✓

TABLE IV. Role Boundary Verification (Direct API Calls via Postman)

Endpoint	Token Presented	Expected	Observed	Status
GET /api/admin/users	ROLE_USER	403	403	✓
GET /api/mod/users	ROLE_USER	403	403	✓
GET /api/admin/users	ROLE_MODERATOR	403	403	✓
DELETE /api/mod/users/{id} — USER target	ROLE_MODERATOR	200	200	✓
DELETE /api/mod/users/{id} — ADMIN target	ROLE_MODERATOR	403	403	✓
PUT /api/admin/users/{id}/role	ROLE_ADMIN	200	200	✓
DELETE /api/admin/users/{id} — ADMIN target	ROLE_ADMIN	400	400	✓

Every role boundary remained intact across all tested combinations when requests were issued directly through Postman, bypassing the browser and all front-end routing logic. This confirms that access enforcement is server-resident and cannot be circumvented through client-side manipulation, URL rewriting, or token presentation with an insufficiently privileged role.

5.3 Security Control Outcomes

TABLE V. Security Control Verification

Threat Vector	Mechanism Applied	Experimental Finding
Automated dictionary attack on passwords	BCrypt cost factor 10 ≈ 200 ms/attempt	10 000 sequential attempts would require approximately 33 minutes; attack is impractical at scale
Credential recovery from database dump	BCrypt one-way digest; unique per-record salt	Stored column contains only 60-character hashes; no plaintext or reversible encoding observed
JWT payload manipulation	HMAC-SHA256 verified on every request	All 15 tampered token variants submitted returned HTTP 401 without exception
Expired credential reuse	exp claim checked server-side on each request	Tokens presented after the 24-hour window were consistently rejected
CSRF via stolen session	Token in Authorization header, not cookie	No CSRF attack surface exists for header-borne credentials
Moderator / Admin bypass without biometric	ProtectedRoute flag check + API role guard	Dashboard render blocked client-side; API returns 403 if token role alone is submitted without biometric flag

5.4 Performance Results

TABLE VI. Response Latency Measurements (10-sample mean, production environment)

Operation	Mean Latency (ms)	Primary Contributor to Latency
Registration	218	BCrypt digest computation — intentional adaptive cost
Email-password sign-in	207	BCrypt comparison + JWT construction and signing
Google OAuth sign-in	413	Firebase network round-trip to Google identity endpoint
JWT validation (per request)	14	Local HMAC-SHA256 recomputation; no network dependency
Authenticated API call	38	Token validation + single indexed primary-key database read
Admin dashboard statistics	183	COUNT / GROUP BY aggregation across the users table

The BCrypt operations account for the dominant share of authentication latency, which is the intended engineering trade-off: the same computational cost that

imposes 200 milliseconds on a legitimate login attempt imposes an equivalent cost on every attempt in an automated attack, making large-scale enumeration impractical without the involvement of significant dedicated hardware [12]. Once a valid token is in hand, per-request overhead averages 14 milliseconds — a figure that falls well below the 100-millisecond threshold below which users perceive interactions as instantaneous and that compares favourably with the overhead reported in comparable stateless JWT implementations in the literature [9].

5.5 Discussion

The aggregate results support three principal conclusions. First, relocating access enforcement from URL-pattern configuration to method-level @PreAuthorize annotations provides qualitatively stronger guarantees: pattern matching can be bypassed through path variations, whereas method-level evaluation occurs after routing and cannot be circumvented without a matching token bearing the required authority. Second, embedding role claims in the JWT produces a single cryptographic artefact that simultaneously proves identity and declares permission scope, eliminating a class of time-of-check-time-of-use race conditions that arise when role lookups are performed separately from authentication. Third, the biometric second factor effectively separates credential possession from account ownership for the two most privileged tiers — a goal that TOTP-based systems approximate but that facial verification achieves without requiring the principal to carry an additional device.

Two limitations require explicit acknowledgement. The absence of token revocation infrastructure means that a token stolen within its validity window remains exploitable for up to 24 hours after the compromise is detected. This is a known and accepted limitation of stateless JWT architectures; resolution requires either short-lived access tokens paired with a refresh mechanism and a server-side revocation store, or a hybrid approach incorporating a lightweight session record for revocation lookups only. The H2 in-memory database resets on Render service restart, which is acceptable for demonstration purposes but constitutes a critical reliability gap for production deployment.

The facial biometric component introduced sensitivity to ambient conditions not present in the token-based controls. Testing revealed reliable detection under standard office lighting on laptop webcams and recent-generation smartphone front cameras; detection quality degraded under low-light conditions and on lower-resolution cameras. A fallback second-factor mechanism — most naturally a TOTP application — would be essential before deploying this tier of the system in environments where camera conditions cannot be guaranteed.

6. CONCLUSION

This paper has described the design and experimental evaluation of a three-tier RBAC framework integrating multi-method authentication and biometric second-factor verification for enterprise web applications. The system was constructed on Spring Boot 3 with Spring Security 6 and React.js 18, deployed to Render and Vercel, and evaluated through direct API manipulation as well as end-to-end browser testing.

Four specific contributions distinguish this work from baseline Spring Boot authentication implementations: server-side, method-level role enforcement that cannot be circumvented through the client; a multi-path authentication pipeline producing structurally equivalent tokens regardless of the credential type used; a mandatory facial biometric step for the two most privileged tiers; and explicit service-layer constraints preventing both vertical and lateral privilege escalation. Experimental results confirm that all role boundaries held under direct API manipulation, all security controls produced the expected responses, and per-request authentication latency averaged 14 milliseconds.

Future work will pursue six extensions. Token revocation will be addressed through the introduction of short-lived access tokens (15-minute validity) combined with rotating refresh tokens and a Redis-backed blacklist, allowing immediate session invalidation after a reported compromise. Persistent storage will be provided by migrating from H2 to PostgreSQL, eliminating the data-loss exposure on service restart. A structured audit log recording every authentication event, role change, and account deletion will provide the forensic capability required for regulatory compliance. Integration with enterprise identity providers — LDAP and SAML 2.0 — will extend the system's applicability to large-scale organisational deployments. An Attribute-Based Access Control layer will be investigated as a complement to the static RBAC model, enabling policies conditioned on contextual factors such as login geography, device posture, and time of day. Finally, a TOTP fallback will be provided for the biometric second factor to cover deployment environments where consistent camera quality cannot be guaranteed.

Conflict of interest statement

Authors declare that they do not have any conflict of interest.

REFERENCES

[1] D. F. Ferraiolo, J. A. Cugini, and D. R. Kuhn, "Role-Based Access Control (RBAC): Features and Motivations," in Proc. 11th Ann. Computer Security Applications Conf. (ACSAC), New Orleans, USA, Dec. 1995, pp. 241–248.

[2] Suganthy A and S. Prasanna Venkatesan, "A Role Centric Model for Improving Security in Organisational Environments," Int. J. Engineering and Technology, vol. 7, no. 1.1, pp. 610–614, 2018.

[3] OWASP Foundation, "A01:2021 – Broken Access Control; A07:2021 – Identification and Authentication Failures," OWASP Top Ten 2021. [Online]. Available: https://owasp.org/Top10/

[4] Google LLC, "Firebase Authentication," Firebase Developer Docs, 2024. [Online]. Available: https://firebase.google.com/docs/auth

[5] Broadcom Inc., "Spring Security Reference — Method Security," 2024. [Online]. Available: https://docs.spring.io/spring-security/reference/servlet/authorization/method-security.html

[6] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," IETF RFC 7519, May 2015. doi: 10.17487/RFC7519.

[7] V. Mühler, "face-api.js — Face Detection and Recognition in the Browser," GitHub, 2020. [Online]. Available: https://github.com/vladmandic/face-api

[8] Nasab A R, M. Shahin, P. L. Bannerman, H. K. Emami, and M. Naseri, "An Empirical Study of Security Practices in Microservice-Based Systems," Applied Sciences, vol. 13, no. 3, Art. 1508, 2023. doi: 10.3390/app13031508.

[9] Darmawan I, A. Bhawiyuga, and K. Amron, "Penetration Testing on JWT Authentication Using CSRF Vectors," in Proc. IEEE ICADEIS, 2021. doi: 10.1109/ICADEIS52521.2021.9430994.

[10] S. Nahar and G. S. Gill, "An Integrated IAM Metamodel for Enterprise Information Systems," J. Network and Computer Applications, vol. 138, pp. 12–27, 2019.

[11] Lopez A, M. A. Muttoo, and A. J. Smith, "Password Authentication Weaknesses: A Comprehensive Review," J. Cybersecurity, vol. 7, no. 2, Art. tyab009, 2021. doi: 10.1093/cybsec/tyab009.

[12] V. Stafford, "Zero Trust Architecture," NIST SP 800-207, NIST, 2020. doi: 10.6028/NIST.SP.800-207.

[13] S. Syed, A. Akhunzada, S. H. Butt, and A. K. Bashir, "A Comprehensive Survey on Zero Trust Architecture," IEEE Access, vol. 10, pp. 113721–113743, 2022. doi: 10.1109/ACCESS.2022.3214912.

[14] H. El Akhdar, S. Baina, and L. Benhlima, "Security Challenges in IoT Microservices Architectures: A Systematic Review," Int. J. Advanced Computer Science and Applications, vol. 14, no. 2, 2023.

[15] G. He, K. Ye, and C. Xu, "Challenges of Zero Trust Networking in Distributed Cloud Architectures," Future Generation Computer Systems, vol. 137, pp. 560–572, 2022. doi: 10.1016/j.future.2022.07.013.