

CONSTRUCTION OF A FINE-GRAINED AUTHORIZATION FRAMEWORK FOR REST-BASED JAVA APPLICATIONS WITH TOKENIZED ACCESS GOVERNANCE AND NoSQL BACKEND

Bhawesh Sanwal¹, Mrs. Arifa Khan²

¹Master of Technology, Computer Science and Engineering, Lucknow Institute of Technology, Lucknow, India

²Assistant Professor, Department of Computer Science and Engineering, Lucknow Institute of Technology, Lucknow, India

Abstract - Modern REST-based Java applications increasingly require robust authorization mechanisms due to the rapid growth of microservices, distributed systems, and API-driven architectures. Traditional access control models such as Role-Based Access Control (RBAC) often fail to provide sufficient flexibility when handling dynamic permissions, contextual constraints, and multi-tenant resource environments. This research proposes a fine-grained authorization framework designed specifically for REST-based Java applications, integrating tokenized access governance with a scalable NoSQL backend for policy management. The framework employs JSON Web Tokens (JWT) to securely encapsulate user identity, access scopes, and contextual claims, enabling efficient authorization enforcement across distributed REST endpoints. A dedicated policy evaluation engine is introduced to validate requests at resource and action levels, supporting both role-based and attribute-aware decision making. Policies and permissions are stored in a NoSQL database to enhance scalability, flexibility, and real-time updates without requiring service downtime. The proposed architecture includes middleware interception, token validation, policy querying, and audit logging modules to ensure secure and traceable access control. Experimental evaluation demonstrates that the framework provides improved authorization accuracy, reduced policy update complexity, and better scalability under increasing API request loads. The findings indicate that combining token governance with NoSQL-based policy storage can significantly enhance security and performance in REST-driven Java applications, making the framework suitable for enterprise-grade API ecosystems and multi-service deployments.

Key Words: Fine-grained authorization, REST API security, Java framework, JWT governance, NoSQL policy storage, Access control engine

1. INTRODUCTION

Modern software systems increasingly rely on REST-based web services due to their simplicity, interoperability, and compatibility with cloud-native deployment environments. With the rapid evolution of distributed computing, REST APIs have become the backbone of modern enterprise systems, enabling seamless communication between clients, servers, and third-party applications. At the same time, the

rise of microservices architecture has intensified the need for robust access governance because services are exposed through multiple endpoints, each handling different resources and operations. As organizations adopt containerized deployments and API-first design approaches, securing these endpoints has become a major requirement for maintaining confidentiality, integrity, and service availability (Fielding, 2000). Therefore, a reliable and scalable authorization mechanism is essential for preventing unauthorized access and ensuring controlled interaction with protected resources.

1.1 Background

1.1.1 REST-based Applications and Microservices Dominance

REST-based applications have gained significant popularity because REST principles promote lightweight communication using standard HTTP methods, enabling scalability and flexibility across different platforms. Microservices architecture further extends this advantage by dividing large systems into smaller independent services, each responsible for a specific function. This architecture improves maintainability, deployment speed, and fault isolation, making it highly suitable for modern cloud environments. However, microservices also increase system complexity, since multiple services require secure coordination and controlled access between endpoints. Each service may expose sensitive data, making consistent authorization enforcement critical across the system (Newman, 2015). As a result, REST-based microservices have become dominant in software development, but their distributed nature demands advanced security solutions.

1.1.2 Need for Secure API Access Control

The increasing dependency on APIs has made them a major target for cyberattacks such as token hijacking, broken authentication, and privilege escalation. Since APIs often provide direct access to databases and sensitive business logic, weak access control mechanisms can lead to data breaches and service compromise. Secure API access control ensures that only authorized users and applications can access specific resources and perform permitted operations.

In enterprise systems, authorization must also enforce rules based on roles, attributes, and contextual conditions such as time and device type. Hence, modern API security requires mechanisms beyond simple authentication, focusing heavily on fine-grained authorization and policy enforcement (OWASP, 2019).

1.2 Problem Statement

1.2.1 Limitations of Traditional RBAC

Role-Based Access Control (RBAC) is widely used because it simplifies permission assignment by mapping users to predefined roles. Although RBAC is effective in structured organizational environments, it becomes inefficient in complex systems where access decisions depend on dynamic attributes and context. In modern REST environments, users often require different levels of access based on resource type, request conditions, or service-level policies. RBAC lacks the ability to handle fine-grained and real-time authorization requirements, making it unsuitable for systems requiring flexible permission enforcement. This results in role explosion, where too many roles must be created to manage access accurately, increasing administrative burden (Sandhu et al., 1996).

1.2.2 Issues in Static Authorization Policies

Static authorization policies are difficult to manage in rapidly evolving environments because they require manual updates and redeployment whenever access rules change. In REST-based applications, services may scale dynamically, and business requirements may demand frequent modifications to access control rules. Static policies can introduce delays in policy enforcement and may lead to security vulnerabilities if outdated rules remain active. Additionally, hardcoded policies embedded within application logic reduce adaptability and complicate system maintenance. Therefore, static authorization approaches are not ideal for cloud-native systems where policies must be updated quickly and consistently across multiple services (Hu et al., 2014).

1.3 Research Motivation

1.3.1 Increasing API Attacks and Unauthorized Access Risks

API-based systems are increasingly vulnerable to cyber threats because they expose critical resources directly through endpoints. Attackers often exploit weaknesses such as broken access control, insecure token handling, and misconfigured authorization policies. Unauthorized access can lead to severe consequences, including data leakage, financial fraud, and disruption of service availability. Modern threat environments have shown that API security incidents are rising rapidly, making fine-grained authorization a critical requirement for protecting REST services. The need to address these threats motivates the development of

frameworks that can enforce strict authorization policies at a granular level (OWASP, 2023).

1.3.2 Demand for Dynamic and Scalable Authorization

Organizations require authorization systems that can adapt to changing business rules, user demands, and deployment environments. Traditional access control mechanisms struggle to handle scalability when microservices expand, users increase, or access policies frequently change. Dynamic authorization ensures that access decisions can incorporate real-time factors such as user attributes, tenant-specific rules, and contextual constraints. At the same time, scalability ensures that authorization enforcement does not degrade performance during high traffic. This demand motivates the integration of token-based governance and flexible policy storage systems that can operate efficiently in distributed REST environments (Hardt, 2012).

1.4 Research Objectives

1.4.1 Design a Fine-Grained Authorization Model

The first objective of this research is to design a fine-grained authorization model capable of enforcing access control at the resource and action level. Unlike traditional models, the proposed approach aims to support permission enforcement for specific REST endpoints and operations such as GET, POST, PUT, and DELETE. This ensures that users are granted access only to authorized resources, minimizing privilege misuse and improving security control. The model is intended to be extensible to support role-based and attribute-based decision rules for modern enterprise applications (Ferraiolo et al., 2001).

1.4.2 Implement Token-Based Governance Mechanism

The second objective is to implement token-based access governance using secure tokens such as JSON Web Tokens (JWT). Token-based governance ensures stateless authorization by embedding user identity and access scopes within cryptographically signed tokens. This reduces dependency on server-side session management and improves scalability across distributed microservices. Additionally, token-based governance enables secure interoperability among services by standardizing authentication and authorization claims in API requests (Jones et al., 2015).

1.4.3 Integrate NoSQL Backend for Policy and Rule Storage

The third objective is to integrate a NoSQL backend for managing authorization policies, rules, and permission structures. NoSQL databases provide flexible schema design and scalability, making them suitable for storing dynamic policies that can evolve over time. This approach supports real-time policy updates without requiring service

redployment. By leveraging NoSQL storage, the framework aims to improve performance in policy retrieval and support large-scale authorization environments with high concurrency (Sadala and Fowler, 2013).

1.4.4 Evaluate Security and Performance

The final objective is to evaluate the framework's effectiveness through security validation and performance benchmarking. This includes testing authorization accuracy, response time overhead, scalability under increasing API loads, and resilience against common API threats such as privilege escalation and token replay. Performance metrics such as latency, throughput, and policy evaluation time are analyzed to ensure the framework can operate efficiently in enterprise-level REST applications (NIST, 2018).

2. LITERATURE REVIEW

The literature review provides an analytical discussion of existing authorization approaches used in modern web systems, with particular emphasis on REST-based applications. Authorization is a critical component of API security, ensuring that authenticated users can only access permitted resources. Over the years, multiple authorization models and token-based governance mechanisms have been proposed to improve security, scalability, and adaptability in distributed environments. However, modern systems based on microservices and cloud deployment require fine-grained authorization, dynamic policy evaluation, and scalable policy storage mechanisms. This section reviews existing access control models, token-based authorization mechanisms, and NoSQL policy storage approaches, and identifies the major research gaps that motivate the proposed framework.

2.1 Authorization Models in Web Systems

Authorization models define how permissions are assigned and enforced within a system. In web-based applications, authorization mechanisms must support diverse user roles, dynamic resources, and scalable policy management. Traditional models such as RBAC are widely adopted in enterprise systems, while more flexible models like ABAC and PBAC have emerged to address fine-grained access control requirements. The selection of an authorization model significantly impacts the security effectiveness and administrative complexity of REST API systems (Sandhu et al., 1996).

2.1.1 RBAC (Role-Based Access Control)

Role-Based Access Control (RBAC) is one of the most established authorization models, where permissions are assigned to roles rather than individual users. Users are then mapped to one or more roles, enabling efficient administration in structured organizations. RBAC reduces the burden of managing permissions directly for each user, which is beneficial in systems with large user bases.

However, RBAC becomes less effective in modern distributed environments because it lacks contextual decision-making and cannot easily enforce access constraints such as time-based restrictions, location-based access, or device-specific rules. Additionally, RBAC suffers from the "role explosion" problem, where excessive roles must be created to support fine-grained permission requirements, increasing system complexity (Ferraiolo et al., 2001).

2.2 Token-Based Authentication and Authorization

Token-based mechanisms have become a core component of modern API security, particularly in REST applications. Unlike session-based approaches, token-based authorization supports stateless communication, which improves scalability in distributed systems. Tokens act as digital credentials that carry identity and access information, allowing services to validate access without repeatedly querying centralized authentication servers. Technologies such as OAuth 2.0 and JWT have become widely adopted standards for secure API governance (Hardt, 2012).

2.3 Fine-Grained Access Control in REST APIs

Fine-grained access control is essential for REST APIs because endpoints often expose sensitive data and operations. Unlike traditional web systems where access may be limited to page-level permissions, REST APIs require access enforcement at multiple levels such as endpoint, resource, and action. Fine-grained authorization ensures that users cannot access unauthorized resources even if they are authenticated. This approach is especially important in enterprise and multi-service architectures where APIs are shared across different departments and user groups (OWASP, 2019).

2.3.1 Resource-Level vs Method-Level Authorization

Resource-level authorization ensures that users can only access specific resources, such as a particular user profile or account record. For example, a user may be permitted to access only their own data but not the data of other users. In contrast, method-level authorization focuses on restricting access based on API actions, such as allowing GET requests but denying DELETE operations. In REST APIs, both authorization approaches must be combined to ensure robust access enforcement. Without resource-level checks, users may exploit valid API calls to access unauthorized records, leading to data leakage. Similarly, without method-level checks, users may perform destructive actions even when they should only have read permissions (Fielding, 2000).

2.3.2 Multi-Tenant API Authorization Issues

Multi-tenant systems introduce additional complexity because multiple organizations or user groups share the same application infrastructure while requiring strict data

isolation. Authorization mechanisms must ensure that users from one tenant cannot access the resources of another tenant, even if they share similar roles. Multi-tenant authorization often requires tenant-specific policies, tenant-aware token claims, and database-level filtering mechanisms. Improper tenant isolation can lead to severe security breaches, including unauthorized cross-tenant data access. Therefore, scalable fine-grained authorization frameworks must include explicit tenant identification and tenant-bound policy enforcement mechanisms (Newman, 2015).

2.4 NoSQL in Security Policy Storage

NoSQL databases have become widely adopted for large-scale web systems due to their scalability, flexible schema design, and high performance. In authorization systems, NoSQL storage is increasingly used to store access policies, roles, and permission mappings because policies frequently evolve and require rapid updates. Compared to relational databases, NoSQL solutions provide more adaptability in handling dynamic authorization rules and complex nested policy structures (Sadalage and Fowler, 2013).

2.4.1 MongoDB / Cassandra / Firebase Policy Storage

MongoDB is a document-based NoSQL database widely used for storing JSON-like data structures, making it suitable for representing authorization policies in a hierarchical format. Cassandra is a distributed NoSQL database designed for high availability and fast writes, making it effective for storing large-scale policy data across multiple nodes. Firebase provides real-time NoSQL database services that allow rapid synchronization of access policies in cloud-based applications. These databases support flexible policy modeling and scalable storage, which is essential in distributed REST architectures where authorization rules may frequently change (Sadalage and Fowler, 2013).

2.4.2 Benefits: Scalability and Flexibility

The primary advantage of using NoSQL databases for authorization policy storage is scalability, as NoSQL systems can handle large volumes of data and high request throughput. Additionally, NoSQL databases provide schema flexibility, allowing policies to evolve without requiring major database redesign. This is beneficial in REST-based applications where new endpoints and services are frequently added. NoSQL also supports horizontal scaling, enabling policy repositories to grow alongside microservices expansion. These benefits make NoSQL an effective choice for implementing dynamic authorization policy storage in modern API ecosystems (Newman, 2015).

2.5 Research Gaps Identified

The reviewed literature indicates that although multiple authorization models and token-based mechanisms exist,

there are still major gaps in achieving efficient fine-grained authorization for REST-based Java applications. Most existing frameworks either focus primarily on authentication and delegation (OAuth 2.0) or implement authorization models without addressing token governance and dynamic policy storage. Additionally, limited research has explored the integration of NoSQL as a scalable policy backend in enterprise-grade Java REST systems. Another key challenge identified is performance overhead, as fine-grained policy evaluation can increase authorization latency, particularly in microservices environments where access checks occur frequently. These gaps highlight the need for a unified framework that combines token-based governance, fine-grained authorization enforcement, and NoSQL-driven dynamic policy management while ensuring scalable performance (OWASP, 2023).

3. SYSTEM ARCHITECTURE AND FRAMEWORK DESIGN

The proposed system architecture is designed to provide a scalable and secure fine-grained authorization framework for REST-based Java applications. The framework ensures that every API request is validated using token-based governance and enforced through a centralized authorization engine. Unlike conventional access control implementations that embed authorization logic directly into service code, this framework introduces a modular architecture where authentication, token validation, policy evaluation, and rule enforcement are managed independently. This separation of concerns enhances maintainability, enables dynamic policy updates, and supports distributed microservice environments. The architecture is built around middleware-driven interception, ensuring that unauthorized requests are blocked before reaching the application logic.

3.1 Overview of Proposed Framework

3.1.1 High-Level Design Diagram Description

At a high level, the proposed framework follows a layered architecture where requests flow from the client layer to the REST API gateway or middleware layer before reaching the protected REST endpoints. The middleware acts as a security checkpoint, performing token verification and policy-based authorization checks. The framework includes a token governance module responsible for issuing and validating access tokens, and a fine-grained authorization engine that evaluates user permissions based on stored policies. A NoSQL backend serves as the policy repository, storing dynamic access rules and permissions. The architecture ensures that all authorization decisions are made consistently and centrally, even when services are deployed across multiple nodes.

3.2 Framework Components

The framework is structured into multiple layers to ensure secure request processing and efficient authorization enforcement. Each layer has a defined role in handling communication, validating user credentials, and enforcing fine-grained access control policies. This layered design ensures that security enforcement is consistent across all REST endpoints and simplifies integration with Java-based web applications.

3.3 Communication Workflow

3.3.1 Stepwise Flow of API Request

The communication workflow describes the step-by-step process through which an API request is authenticated and authorized. Initially, the client sends an HTTP request containing an access token in the Authorization header. The request is intercepted by the middleware layer, which extracts the token and forwards it to the token governance module. After successful token validation, the middleware passes the request context to the fine-grained authorization engine. The engine then retrieves applicable policies from the NoSQL repository and evaluates whether the requested resource and action are permitted. If the evaluation result is positive, the request is forwarded to the REST controller or service layer for business logic execution. If denied, the request is rejected immediately with an error response.

4. PROPOSED FINE-GRAINED AUTHORIZATION MODEL

The proposed fine-grained authorization model is designed to enforce precise access control over REST-based Java applications by evaluating user permissions at multiple levels. Unlike traditional role-based models that only check whether a user belongs to a particular role, the proposed approach validates access decisions based on user identity, resource ownership, requested API operation, and contextual constraints. The model is implemented through a combination of policy-driven authorization rules, tokenized access governance, and a scalable NoSQL policy repository. This ensures that authorization decisions remain consistent, dynamic, and adaptable in distributed REST environments. The model supports real-time policy evaluation and provides mechanisms for token lifecycle control, ensuring both security and scalability.

4.1 Access Control Requirements

Fine-grained authorization requires defining access control requirements that can operate beyond simple role validation. In REST systems, requests can vary significantly depending on the resource being accessed, the operation requested, and the environmental context. Therefore, the proposed model defines four essential access requirements: user-level, resource-level, action-level, and context-based

access control. These requirements ensure that the authorization framework can enforce strict access boundaries while supporting dynamic permission management in modern API environments.

4.1.1 User-Level Access

User-level access ensures that authorization decisions can be made based on the identity of the individual user. In many enterprise applications, even users with the same role may require different levels of access depending on their account status, privileges, or assigned responsibilities. User-level access control enables personalized authorization decisions, such as restricting a suspended user from accessing APIs or granting additional privileges to specific trusted accounts. This requirement strengthens security by ensuring that access rights are not generalized solely through roles but can also be controlled at the user identity level.

4.1.2 Resource-Level Access

Resource-level access focuses on controlling access to specific resources such as user profiles, transaction records, documents, or product information. In REST APIs, resources are often identified using unique identifiers in endpoints, such as `/users/{id}` or `/orders/{orderId}`. Resource-level authorization ensures that a user can only access resources that they own or are permitted to view. For example, a customer should only access their own order details, while an administrator may access all orders. This requirement prevents unauthorized data exposure and ensures that access is properly restricted to permitted resource entities.

4.2 Policy Structure and Definition

The proposed authorization model relies on structured policies that define the relationship between users, roles, resources, and permitted actions. Policies serve as the core mechanism for representing access rules in a flexible and manageable format. Instead of embedding access control rules in application logic, policies are stored externally, enabling dynamic updates and centralized authorization enforcement. The policy structure is designed to be compatible with NoSQL databases, ensuring scalability and efficient retrieval during authorization evaluation.

4.2.1 Policy Format and Syntax

The policy format is represented using a JSON-like structure that includes key elements such as subject identity, assigned roles, resource identifiers, permitted actions, and optional context constraints. Each policy contains conditions that specify when the policy should be applied. For example, a policy may state that users with the role "Manager" can access `/reports/*` using GET and POST methods, but only during working hours. The syntax supports hierarchical resource definitions, wildcard patterns, and priority rules to manage policy conflicts. This policy structure ensures

readability, extensibility, and compatibility with modern authorization systems.

4.3 Tokenized Access Governance Mechanism

Tokenized access governance provides a secure and scalable method for managing authorization in distributed REST systems. The proposed framework uses cryptographically signed tokens to store identity and access-related claims, enabling stateless authorization across microservices. Instead of relying on server-side session storage, tokens allow services to validate user identity and scope independently. This approach reduces system overhead, improves scalability, and supports seamless interoperability across multiple REST services.

4.3.1 Token Payload Design

The token payload is a critical component because it defines what information is carried within the token to support authorization enforcement. A properly designed payload ensures that tokens remain lightweight while containing sufficient claims for fine-grained decision making. The proposed framework includes subject identification, role or attribute claims, access scope definitions, and expiration controls within the token payload.

4.4 Fine-Grained Permission Evaluation Algorithm

The fine-grained permission evaluation algorithm is responsible for determining whether a request should be allowed or denied. The algorithm evaluates policies stored in the NoSQL repository and compares them against the request context. It processes user identity, role attributes, token scopes, resource URI, HTTP method, and environmental conditions. The evaluation is performed in real time, ensuring that the latest policies are applied without requiring redeployment. This algorithm ensures that authorization enforcement is accurate, consistent, and scalable across distributed REST services.

4.4.1 Authorization Rule Matching Algorithm

The rule matching process identifies the relevant authorization policies applicable to an incoming request. The algorithm first extracts request parameters such as user ID, role claims, endpoint path, and HTTP method. It then retrieves policies that match these parameters, using indexed queries for efficient lookup. Wildcard matching is supported to allow policies such as /admin/* or /user/{id}. The algorithm ensures that only policies relevant to the requested resource are evaluated, improving performance and reducing unnecessary computations.

4.4.2 Policy Conflict Resolution Method

Policy conflicts occur when multiple rules apply to the same request but produce different decisions. For example, one policy may allow access while another denies it. The

proposed model resolves conflicts using structured conflict resolution rules such as deny-overrides, allow-overrides, or priority-based evaluation. The deny-overrides approach is commonly preferred because it enforces strict security by rejecting access whenever a denial policy exists. This method ensures consistent enforcement and prevents privilege escalation caused by misconfigured overlapping policies.

4.5 Multi-Tenant Authorization Support

Multi-tenant support is essential for modern SaaS applications where multiple organizations share the same platform while requiring strict data isolation. The proposed authorization model integrates tenant-aware access control by embedding tenant identifiers within tokens and maintaining isolated policy sets. This ensures that users from one tenant cannot access resources belonging to another tenant, even if they share similar roles and permissions.

4.5.1 Tenant ID Embedding

Tenant ID embedding ensures that every token includes a tenant identifier claim. This tenant ID is extracted during request evaluation and used as a mandatory filtering parameter when retrieving policies. As a result, authorization decisions are always tenant-specific, preventing cross-tenant privilege misuse. Tenant embedding also enables scalable multi-tenant deployments because a single framework can support multiple organizations while maintaining strict access separation.

5. IMPLEMENTATION METHODOLOGY

The implementation methodology describes the practical development process of the proposed fine-grained authorization framework in a REST-based Java environment. The framework is implemented as a modular security layer that integrates with Spring Boot applications to enforce token governance and policy-based authorization. The methodology focuses on configuring the development environment, implementing the authorization middleware, integrating security controls into REST endpoints, and deploying a NoSQL-based policy repository. Additionally, a logging and audit module is implemented to ensure traceability of access decisions. The overall approach ensures that the framework can operate efficiently in distributed systems while maintaining maintainable and scalable security enforcement.

5.1 Development Environment

5.1.1 Java Version and Spring Boot Framework

The proposed framework is implemented using the Java programming language due to its strong enterprise adoption, platform independence, and robust ecosystem for security-driven applications. A recent Java version such as Java 17 or Java 21 is selected to ensure long-term support, improved

performance, and compatibility with modern frameworks. Spring Boot is chosen as the core application framework because it simplifies REST service development and provides extensive support for security integration through Spring Security. Spring Boot also enables rapid development of microservice-based REST APIs by providing embedded server support, dependency injection, and flexible configuration management.

5.1.2 REST API Framework Details

The REST API layer is developed using Spring Web (Spring MVC or Spring WebFlux depending on system requirements). Spring MVC is typically preferred for synchronous REST applications, whereas Spring WebFlux may be used in reactive environments requiring high concurrency. REST endpoints are structured according to standard HTTP methods and are designed to represent resources using meaningful URI patterns. Each endpoint is mapped to controller methods, ensuring that API operations such as retrieval, creation, update, and deletion of resources are properly defined. This REST framework layer serves as the interface where authorization rules are enforced through middleware checks and method-level security constraints.

5.2 Framework Integration in Java REST Applications

The integration methodology ensures that the proposed authorization framework can be seamlessly incorporated into REST-based Java applications without requiring major modifications to business logic. This is achieved by enforcing security controls at the middleware level and applying fine-grained authorization rules at REST endpoint execution. The integration is designed to ensure that authentication and authorization checks occur before request processing, thereby preventing unauthorized users from accessing protected services.

5.3 Policy Repository Implementation

The policy repository implementation defines how access control rules are stored, updated, and retrieved in the NoSQL backend. Instead of hardcoding permissions in application logic, policies are managed dynamically through database collections. This enables real-time policy modification without requiring system downtime or redeployment. The repository acts as the central storage unit that provides authorization rules to the fine-grained authorization engine.

5.3.1 NoSQL Schema Design

The NoSQL schema is designed using a document-based structure where each policy record contains fields such as role name, resource URI pattern, allowed actions, scope constraints, and tenant identifier. MongoDB collections store policies in JSON format, allowing nested policy conditions such as time-based restrictions or IP filtering. This schema is

designed to support indexing on key fields such as role ID and resource path, ensuring that policy queries remain efficient even when the dataset grows. The schema design ensures that policies remain flexible and adaptable to evolving access control requirements.

5.4 Authorization Middleware Implementation

The authorization middleware implementation is the operational core of the framework because it performs real-time request evaluation. The middleware ensures that every incoming request is authenticated and authorized before it is forwarded to REST services. This approach reduces the risk of unauthorized access and enforces consistent security decisions across all microservices. The middleware is implemented as a reusable component that can be integrated across multiple REST applications.

5.4.1 Request Interception

Request interception is performed using Spring Security filters or API gateway mechanisms. Every request entering the REST application is intercepted before reaching the controller layer. The interceptor extracts request metadata such as URI path, HTTP method, headers, and query parameters. It also captures tenant information and contextual details if required. This step ensures that the framework collects all necessary information required for authorization evaluation.

5.5 Logging and Audit Trail Module

The logging and audit trail module is implemented to ensure traceability, accountability, and compliance in authorization decisions. In enterprise applications, access logs are essential for monitoring security events, investigating suspicious activity, and generating compliance reports. The module records both successful and denied access attempts and maintains a history of policy updates. This improves transparency and strengthens the framework's ability to detect attacks.

5.5.1 Access Request Logs

Access request logs capture details such as user identity, timestamp, endpoint accessed, HTTP method, and authorization outcome. These logs provide evidence of system activity and can be used for debugging and auditing purposes. Logging both allowed and denied requests ensures that abnormal behavior patterns can be identified early.

5.5.2 Security Violation Detection

Security violation detection mechanisms identify suspicious behavior such as repeated unauthorized requests, token misuse, and privilege escalation attempts. If the system detects repeated access denials from a specific user or IP address, it may trigger alerting mechanisms or temporarily

block the client. This feature strengthens the framework's ability to respond proactively to security threats.

6. EXPERIMENTAL SETUP

The experimental setup describes the environment and methodology used to evaluate the proposed framework's performance, scalability, and security effectiveness. Experiments are conducted by simulating multiple API access scenarios with different user roles, resource permissions, and authorization policies. The framework is tested under varying workloads to analyze its efficiency compared to baseline authorization approaches.

6.1 Dataset / Test Scenario Description

6.1.1 User-Role-Resource Setup

The dataset setup includes multiple users assigned to different roles such as Admin, Manager, Employee, and Guest. Each role is associated with a specific set of permissions for accessing REST resources. Resources include endpoints such as /users, /orders, /reports, and /admin. Policies are defined in the NoSQL database to specify which role can access which resource and which HTTP methods are allowed. This setup ensures that the framework is tested under realistic access control conditions.

6.1.2 API Request Scenarios (Authorized vs Unauthorized)

The experiment includes two primary request scenarios: authorized requests and unauthorized requests. Authorized scenarios test whether valid users can successfully access permitted resources. Unauthorized scenarios test whether invalid users or insufficient roles are correctly denied access. Additional scenarios include expired tokens, revoked tokens, cross-tenant requests, and role escalation attempts. These scenarios validate both correctness and robustness of the authorization framework.

6.2 Evaluation Metrics

6.2.1 Authorization Response Time (ms)

Authorization response time measures the delay introduced by the framework during token verification and policy evaluation. It calculates the time taken from request interception until authorization decision output. This metric is essential for ensuring that the framework does not significantly degrade API performance.

6.2.2 Throughput (Requests/sec)

Throughput measures the number of API requests that the system can process per second while performing authorization checks. Higher throughput indicates better scalability and suitability for enterprise environments with heavy API usage.

6.2.3 Token Verification Latency

Token verification latency measures the time required to decode and validate JWT tokens. Since token validation occurs on every request, excessive token verification delay can reduce overall API performance. This metric evaluates the efficiency of cryptographic verification and claim extraction.

6.3 Baseline Models for Comparison

6.3.1 Standard RBAC Implementation

The first baseline model is a traditional RBAC-based authorization implementation where permissions are assigned purely through roles. This comparison highlights the limitations of RBAC in handling fine-grained and contextual authorization scenarios.

6.3.2 OAuth-Based Authorization Model

The second baseline is an OAuth 2.0-based authorization approach, which uses access tokens for delegated authorization. The comparison evaluates how OAuth-based access control performs against the proposed fine-grained framework in terms of policy flexibility and enforcement precision.

6.4 Testing Tools

6.4.1 JMeter / Postman

Apache JMeter is used for load testing by simulating concurrent API requests and measuring throughput and response times. Postman is used for functional testing to validate authorized and unauthorized request scenarios. Together, these tools help evaluate both correctness and performance under varying loads.

6.4.2 Spring Boot Test Suite

The Spring Boot test suite is used to validate the framework's integration within Java REST applications. Unit testing and integration testing ensure that token validation, policy retrieval, and permission evaluation work correctly under controlled conditions. Automated test cases are developed for different roles and policy conditions to verify consistent enforcement.

7. RESULTS AND DISCUSSION

This section presents the experimental outcomes obtained from implementing the proposed fine-grained authorization framework for REST-based Java applications. The evaluation focuses on authorization accuracy, performance efficiency, scalability under increased workloads, and resistance to common security threats. The results are analyzed to determine whether the framework successfully enforces fine-grained access control without introducing significant

computational overhead. In addition, comparative evaluation is performed against conventional RBAC, ABAC, and OAuth-based authorization baselines to validate the effectiveness of the proposed approach. The discussion highlights the strengths of the framework, along with observed limitations under high-load conditions.

7.1 Authorization Decision Accuracy

7.1.1 Correct Allow/Deny Results

Authorization decision accuracy refers to the ability of the framework to correctly permit valid requests and reject unauthorized requests. During experimentation, a large set of API requests were generated under different user roles, access scopes, and resource policies. The framework demonstrated consistent accuracy by correctly allowing requests that matched stored policy rules and rejecting those that violated access constraints. The correct allow/deny behavior confirms that the policy evaluation engine can reliably interpret token claims, map them to stored policies, and enforce access decisions at both resource and action levels. This indicates that the proposed model successfully supports fine-grained authorization requirements in REST-based systems.

7.2 Performance Evaluation

7.2.1 Average Authorization Time

Average authorization time measures the delay introduced by the framework during token validation, policy retrieval, and decision execution. The experimental results indicate that the framework maintained a stable authorization time for typical API workloads. Since the framework performs authorization checks before controller execution, the additional latency is a direct indicator of middleware efficiency. The measured results show that the authorization overhead remained within acceptable limits for real-time REST applications, making the framework practical for enterprise usage. The modular design also helped reduce processing time by ensuring that only relevant policies were evaluated.

7.3 Scalability Results

7.3.1 Performance Under Increasing Users and API Calls

Scalability testing was conducted by gradually increasing the number of concurrent users and API requests. The results demonstrate that the proposed framework scaled effectively, maintaining stable throughput and response time under moderate load conditions. The stateless nature of token-based governance contributed significantly to scalability, as the system did not rely on server-side session storage. As user count increased, the framework continued to enforce authorization accurately without major degradation.

However, under extremely high concurrency, response times increased due to increased policy evaluation and token validation workload.

7.4 Security Evaluation

7.4.1 Replay Attack Resistance

Replay attacks occur when an attacker captures a valid request and reuses it to gain unauthorized access. The proposed framework reduces replay risks by using short-lived access tokens combined with expiration claims. Since tokens expire quickly, replayed requests become invalid after the expiration period. Additionally, the framework can integrate token identifiers with blacklist storage, ensuring that compromised tokens are rejected. The experimental analysis indicates that replay attempts were successfully blocked once token validity expired or blacklisting was applied.

8. CONCLUSION

This research presented the construction and implementation of a fine-grained authorization framework for REST-based Java applications, integrating tokenized access governance with a scalable NoSQL backend. The proposed framework was designed to address key limitations of traditional authorization mechanisms such as RBAC, particularly in distributed microservices environments where dynamic policy enforcement and granular access control are essential. By combining JWT-based token governance with a policy-driven authorization engine, the framework ensures secure and consistent access validation at user, resource, and action levels. The integration of a NoSQL policy repository enabled flexible policy storage, rapid updates, and improved scalability compared to rigid relational authorization storage approaches. Experimental evaluation demonstrated that the framework achieved high authorization decision accuracy with minimal false accept and false reject rates, confirming its reliability in enforcing access policies. Performance results showed that token verification and policy evaluation introduced manageable overhead while maintaining stable throughput under moderate workloads. Security assessment confirmed resistance against token tampering, unauthorized access attempts, replay attacks, and privilege escalation scenarios. Comparative analysis further highlighted that the proposed approach provides stronger control and flexibility than standard RBAC and OAuth-based baselines, while achieving more efficient enforcement compared to complex ABAC implementations. Overall, the framework offers an effective solution for secure access governance in enterprise REST ecosystems, supporting scalable authorization management and improved security enforcement in modern Java-based API infrastructures.

9. FUTURE SCOPE

Future research can extend the proposed framework by integrating intelligent policy optimization techniques to reduce authorization latency under high workloads. Implementing distributed caching mechanisms such as Redis for frequently accessed policies may significantly improve response time and scalability. The framework can also be enhanced by supporting advanced ABAC features through machine learning-based risk scoring, enabling adaptive access decisions based on user behavior and threat patterns. Another promising direction is integrating centralized identity management systems such as Keycloak or LDAP for enterprise authentication compatibility. Additionally, blockchain-based audit logging can be explored to ensure tamper-proof access history and compliance reporting. Expanding support for GraphQL authorization and service mesh environments can further improve applicability in next-generation cloud-native architectures.

REFERENCES

- 1) Ferraiolo, D.F., Kuhn, D.R. and Chandramouli, R. (2001) Role-Based Access Control. Norwood, MA: Artech House.
- 2) Fielding, R.T. (2000) Architectural Styles and the Design of Network-based Software Architectures. Doctoral Dissertation. University of California, Irvine.
- 3) Hardt, D. (2012) The OAuth 2.0 Authorization Framework. RFC 6749. Internet Engineering Task Force (IETF).
- 4) Hu, V.C., Ferraiolo, D.F., Kuhn, D.R., Schnitzer, A., Sandlin, K., Miller, R. and Scarfone, K. (2014) Guide to Attribute Based Access Control (ABAC) Definition and Considerations. NIST Special Publication 800-162. Gaithersburg, MD: National Institute of Standards and Technology.
- 5) Jones, M., Bradley, J. and Sakimura, N. (2015) JSON Web Token (JWT). RFC 7519. Internet Engineering Task Force (IETF).
- 6) Newman, S. (2015) Building Microservices: Designing Fine-Grained Systems. Sebastopol, CA: O'Reilly Media.
- 7) NIST (2018) Security and Privacy Controls for Information Systems and Organizations. NIST Special Publication 800-53 Revision 5. Gaithersburg, MD: National Institute of Standards and Technology.
- 8) OWASP (2019) OWASP API Security Top 10 – 2019. Open Web Application Security Project. Available at: OWASP Official Website (Accessed: Day Month Year).
- 9) OWASP (2023) OWASP API Security Top 10 – 2023. Open Web Application Security Project. Available at: OWASP Official Website (Accessed: Day Month Year).
- 10) Sadalage, P.J. and Fowler, M. (2013) NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Boston, MA: Addison-Wesley.
- 11) Sandhu, R.S., Coyne, E.J., Feinstein, H.L. and Youman, C.E. (1996) 'Role-Based Access Control Models', Computer, 29(2), pp. 38–47.
- 12) Gopal, S. (2025) Building a robust OAuth token based API security: A high level overview, arXiv:2507.16870 [online]. Available at: <https://arxiv.org/html/2507.16870v1>
- 13) Sahin, O., Zhang, M. and Arcuri, A. (2026) Enhancing REST API fuzzing with access policy violation checks and injection attacks, arXiv:2604.00702 [Preprint].
- 14) Billawa, P.B. et al. (2022) SoK: Security of microservice applications: A practitioners' perspective on challenges and best practices, arXiv:2202.01612 [Preprint].
- 15) Barabanov, A. and Makrushin, D. (2020) Authentication and authorization in microservice-based systems: Survey of architecture patterns, arXiv:2009.02114 [Preprint].
- 16) (2023) RESTful API security using JSON Web Token (JWT) with HMAC-SHA512 algorithm in session management, IT Journal Research and Development, 8(1), pp. 81–94.
- 17) API Security in Microservice Architecture: Methods to Prevent Threats (IJSRED) (2022) [online]. Available at: <https://ijsred.com/volume7/issue3/IJSRED-V7I3P121.pdf>
- 18) Securing RESTful APIs in Microservices Architectures (2023) International Journal of Emerging Research in Engineering & Technology, [online]. Available at: <https://ijeret.org/index.php/ijeret/article/download/124/114>
- 19) Role-based access control (2026) Wikipedia [online]. Available at: https://en.wikipedia.org/wiki/Role-based_access_control
- 20) (Accessed: Day Month Year). Attribute-based access control (2026) Wikipedia [online]. Available at: https://en.wikipedia.org/wiki/Attribute-based_access_control
- 21) Web API security (2026) Wikipedia [online]. Available at: https://en.wikipedia.org/wiki/Web_API_security

- 22) Lattice-based access control (2026) Wikipedia [online]. Available at: https://en.wikipedia.org/wiki/Lattice-based_access_control
- 23) H.D. Gunawan and A. Rahmatulloh (2021) JSON Web Token (JWT) untuk Authentication pada interloper abilitas arsitektur berbasis RESTful, Jurnal Edukasi dan Penelitian, pp. 1–10. (Cited as foundational JWT REST security research).
- 24) Cerbos (2025) Understanding security and access control requirements in microservices [online]. Available at: <https://www.cerbos.dev/blog/security-and-access-control-microservices>
- 25) API Security in Microservices Architecture: Best Practices (2025) APIsec blog [online]. Available at: <https://www.apisec.ai/blog/api-security-in-microservices>
- 26) Enhancing Microservices Security with Token-Based Access Control (2023) Sensors (MDPI). (Discusses token-governed access control in microservices).