

FUNCTIONAL DESIGN OF E-PAY Java SERVICE

August 2024

|  |  |
| --- | --- |
| **Document Control Sheet** | |
| **Current Version** | 0.1 |
| **Project Code** | NS\_SBI\_04\_08\_2024 |
| **Project Name** | SBI ePay Implementation |
| **Document Type** | ePay Java Service |
| **Author** | Ranu Jain |
| **First Reviewed By** | Deependra |
| **Final Reviewed By** |  |
| **Contact Person** | Ranu Jain |
| **Frequency of Document Review** | Next Revision of Standard Documents |
| **Document Creation Date** | 20-08-2024 |
| **Last Updated Date** |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision History** | | | |
| **Version** | **Date** | **Name** | **Comments** |
| 0.1 | 20-08-2024 | Ranu Jain | Draft Version |
|  |  |  |  |

[Document Purpose 2](#_Toc1023272218)

[Scope 3](#_Toc324412227)

[1. Authentication/Authorization Service 3](#_Toc385283557)

[2. Caching Service 3](#_Toc1845841800)

[3. Notification Service 3](#_Toc298486342)

[4. Encryption / Decryption Service 4](#_Toc589863422)

[5. Distributed Logging Service 4](#_Toc1169162415)

[Functional Specifications 4](#_Toc1563973837)

[1. FS\_JS 1 – Authentication/Authorization Service 5](#_Toc1605617064)

[1.1 - Objective 5](#_Toc523629254)

[1.3 - Non - Functional Specification 5](#_Toc1893265866)

# Document Purpose

This document refers to implementing Java Based Service for ePay2.0 portal for SBI ePay application which can be reused in all microservices.

# Scope

1. Authentication/Authorization Service

* **Authentication:** Verify user identities to ensure only legitimate access.
* **Authorization:** Determine and enforce user access rights based on roles.
* **Token Management:** Generate, verify, and manage tokens for secure functionality.
* **Role-Based Access Control:** Enforce access permissions based on user roles.
* **Whitelisting:** Restrict access to endpoints via IP or service-based whitelisting.
* **Token Revocation:** Implement logic to invalidate tokens, focusing on refresh tokens.
* **CORS Configuration:** Control allowed external domains for secure API interactions

1. Caching Service

* **In-Memory Data Storage:** Implement fast, in-memory caching using Redis for enhanced data retrieval speeds.
* **Data Expiry Management:** Use TTL (Time-To-Live) settings to manage cache entry lifecycles efficiently.
* **Cache Invalidation:** Develop strategies to invalidate or update cache entries when underlying data changes.
* **Distributed Caching:** Enable distributed caching across multiple Redis instances
* **High Availability:** Implement Redis replication and failover to ensure continuous service availability.
* **Performance Monitoring:** Monitor and optimize cache performance in minimal latency in read/write operations.

1. Notification Service

* **Email Notifications:** Implement functionality to send emails using customizable templates.
* **SMS Notifications:** Develop the capability to send SMS messages with predefined templates.
* **Template Creation:** Creating templates for both email and SMS.
* **Template Integration:** Ensure seamless integration of templates into the notification workflow for dynamic content delivery.
* **Delivery Tracking:** Implement mechanisms to track and log the status of sent notifications.
* **Error Handling:** Include robust error handling and retry logic for failed notifications.
* **Scalability:** Design the service to handle high volumes of notifications efficiently.

1. Encryption / Decryption Service

* **AES 256 with GCM Cipher:** Implement encryption and decryption using AES 256 with GCM mode for secure and authenticated data encryption.
* **SHA-256, SHA-512, and SHA-1024:** Provide hashing functionality using SHA-256, SHA-512, and SHA-1024 for data integrity and security.
* **Key Management:** Develop secure key generation, storage, and retrieval mechanisms for encryption and hashing processes.
* **Data Integrity:** Ensure data integrity with authenticated encryption and secure hashing algorithms.
* **Performance Optimization:** Optimize encryption, decryption, and hashing operations for minimal latency and resource efficiency.

5. Distributed Logging Service

* **Centralized Log Collection:** Implement a service that collects logs from distributed applications and services, aggregating them in a centralized location.
* **Log Aggregation and Processing:** Utilize tools like Logstash to aggregate, filter, and transform logs before sending them to Elasticsearch.
* **Elasticsearch Integration:** Store logs in Elasticsearch, enabling efficient querying, indexing, and analysis of large volumes of log data.
* **Kibana Visualization:** Configure Kibana to visualize and analyze log data,
* **Distributed Logging Support:** Ensure the service supports logging from distributed systems and microservices, with capabilities to handle large-scale, multi-node environments.
* **Log Retention and Management:** Implement policies for log retention, archiving, and deletion, ensuring compliance with storage and data protection regulations.
* **Error Handling and Alerting:** Develop mechanisms to detect and alert logging errors, anomalies, or patterns that indicate potential issues or security threats.
* **Scalability and Performance:** Design the service to handle high throughput and large volumes of log data, ensuring scalability and performance optimization across distributed environments.

# Functional Specifications

## FS\_JS 1 – Authentication/Authorization Service

### 1.1 - Objective

* Authenticate users securely using Java Spring Security Based solutions.
* Manage role-based access control for fine-grained permissions.
* Handle token generation, validation, and expiration for session management.
* Implement IP and service whitelisting to restrict access to specific endpoints.
* Support token revocation to invalidate compromised or expired tokens.
* Enforce CORS policies to control allowed external domains for secure API interactions.
* Ensure scalability and integration within the Java application ecosystem.

1.2 - Functional Specification

**1. Authentication**

* + **Sign In:** Implement authentication service using Java Spring Security supporting multiple ways like username/password combinations, Merchant and PG Keys.
  + **OAuth2 Integration:** Enable OAuth2 authentication via Java libraries like Spring Security OAuth2, allowing users to authenticate through third-party providers.

**2. Authorization**

* + **Role-Based Access Control:** Define and enforce RBAC using Java annotations or configuration files, ensuring users have access only to resources matching their roles.
  + **Permission Hierarchies:** Implement hierarchical permissions using Java-based role management, allowing fine-grained control over resource access.
  + **Resource-Based Access Control:** Restrict access to specific resources within your Java application based on user roles and resource identifiers.

**3. Token Management**

* + **Token Generation:** Use Java libraries like java-jwt or Spring Security to generate and manage JWT tokens, supporting both access and refresh tokens.
  + **Token Validation:** Implement token validation logic within Java, ensuring tokens are verified for authenticity, expiration, and integrity.
  + **Token Expiry and Renewal:** Handle token expiration and renewal processes using Java's date/time utilities and token refresh mechanisms.

**4. Role-Based Access Control (RBAC)**

* + **Role Definition:** Define user roles using Java-based configuration, integrating roles into your security framework.
  + **Dynamic Role Assignment:** Dynamically assign roles in Java based on user attributes, leveraging Spring Security or custom logic.
  + **API Access Control:** Enforce role-based restrictions on Java-based REST APIs, using annotations like @PreAuthorize and @Secured.

**5. Whitelisting**

* + **IP Whitelisting:** Implement IP whitelisting at the service level using Java filters or interceptors, restricting access based on IP address.
  + **Service Whitelisting:** Control access to specific services or endpoints within the Java application, allowing only whitelisted services to interact with sensitive APIs.
  + **Whitelist Management:** Provide an API or admin interface in Java to manage and update the whitelist dynamically.

**6. Token Revocation**

* + **Refresh Token Revocation:** Implement refresh token revocation logic using Java, ensuring tokens can be invalidated and are no longer usable.
  + **Immediate Effect:** Ensure revoked tokens are immediately blocked by maintaining a blacklist in memory or in a database using Java's concurrency utilities.
  + **Revocation List Management:** Manage and persist a list of revoked tokens within your Java application, using a database or in-memory store like Redis.

**7. CORS Configuration**

* + **CORS Policy Definition:** Define CORS policies within your Java application using Spring Security or JAX-RS filters, specifying allowed origins, methods, and headers.
  + **Preflight Requests:** Handle CORS preflight requests using Java-based filters, ensuring they are processed correctly according to the defined CORS policy.
  + **Dynamic CORS Rules:** Allow for dynamic updates to CORS rules in Java by configuring properties or using a configuration management system.

**8. Logging and Monitoring**

* + **Authentication Logs:** Log all authentication attempts using Java's logging framework (e.g., Logback, SLF4J), including both successful and failed login attempts.
  + **Authorization Logs:** Track and log authorization decisions within your Java application, ensuring you can audit who accessed what resources and when.
  + **Monitoring Integration:** Integrate with monitoring tools like Prometheus or ELK Stack using Java agents or APIs to track the performance and security of the Authentication/Authorization service.

This functional specification outlines how to implement the key features of the Authentication/Authorization service using Java, ensuring a secure and scalable application.

1.3 - Non - Functional Specification

TBD