****Member Management System****

**Version: 1.0**

**Date: 2025-09-11**

**Status: Draft**

# 1. Introduction

## 1.1 Purpose

**This document provides a detailed, technical design for the Member Management System as specified in the SRS v2.0. It describes the module decomposition, database schema, API specifications, and implementation strategies for meeting all functional and non-functional requirements. This document is intended for software developers, testers, and system architects.**

## 1.2 Scope

**This document covers the design of the backend services and database for the core member functionality:**

**Member Registration**

**Member Profile Management**

**Member Search**

**It explicitly excludes the design of the secondary verification system, user interface components, and any external organizational management systems.**

## 1.3 Definitions, Acronyms, and Abbreviations

**SRS: Software Requirements Specification**

**API: Application Programming Interface**

**REST: Representational State Transfer**

**JWT: JSON Web Token**

**HTTP: Hypertext Transfer Protocol**

**BCrypt: A password hashing function**

**PK: Primary Key**

**FK: Foreign Key**

**DAO: Data Access Object**

## 1.4 References

**SRS: "Member Management System - Version 2.0"**

**Architecture Diagram: [Link to System Architecture Confluence Page]**

**API Style Guide: [Link to Internal REST API Guidelines]**

# 2. Module Design

**The Member Management System is implemented as a single module. This section describes the key classes within this module that collectively fulfill the functional requirements.**

## 2.1 Class: MemberController

**Purpose: Handles incoming HTTP requests, routes them to the appropriate service methods, and returns HTTP responses. This class is the entry point for all REST API calls.**

**Key Methods:**

**createMember(memberDto dto): Processes registration requests.**

**Detailed Logic:**

**1. Validate input parameters (e.g., email format, password strength).**

**2. Hash the plaintext password using BCrypt (cost factor 12).**

**3. Map memberDto to Member entity object.**

**4. generate a unique Long as the member ID.**

**5. Call MemberDao.save(member).**

**6. Return an OpResult.**

**updateMember(memberDto dto): Processes profile update requests.**

**searchMembers(memberDto dto): Processes search requests.**

**Detailed Logic:**

**1. Authorize request: Check if `authenticatedUserId` has the "SEARCH\_MEMBERS" permission.**

**2. Validate pagination parameters (`pageNo`, `pageSize`).**

**3. Call MemberService to search totalRecords if pageNo is not null.**

**4. Call MemberService to search matched Member entities.**

**5. Map entities to Dtos.**

**6. Return OpResult with Dtos, result code etc.**

**Dependencies: MemberService**

## 2.2 Class: MemberService

**Purpose: Contains the core business logic for member operations, including registration, profile updates, and search. It orchestrates interactions between other classes.**

**Key Methods:**

save(Member member)**: Validates input, checks for duplicate login IDs, hashes passwords, and initiates member creation.**

update(Member member)**: Validates permissions and updates member profiles.**

search(MemberDto dto)**: Executes search logic and applies data masking.**

**Dependencies: MemberDao**

## 2.3 Class: MemberDao

**Purpose: Manages data persistence for member entities. Abstracts database interactions using the DAO pattern.**

**Key Methods:**

**No methods.**

**Dependencies:** IBaseDao**.**

## 2.4 Class: IBaseDao

public interface IBaseDao<T, ID extends Serializable> extends PagingAndSortingRepository<T, ID>, CrudRepository<T, ID>

**Purpose: Abstracts database interactions using the DAO pattern.**

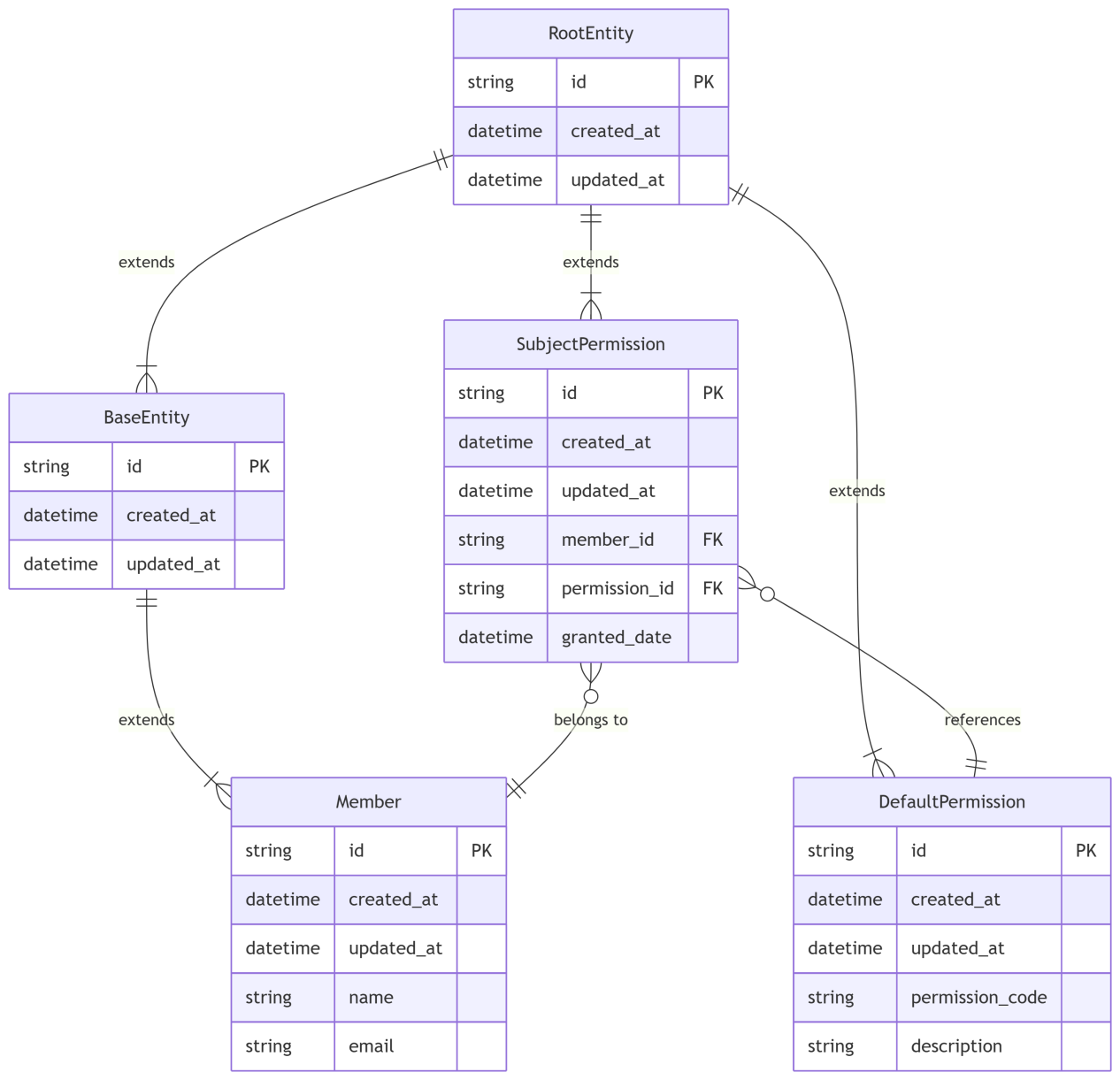
**Key Methods:**

**No methods.**

**Dependencies: JPA.**

# Data Design

## 3.1 The Entity Relationship Diagram (ERD)

****

## 3.2 Detailed Entity Definitions

**Class Member:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ****Field name**** | ****Type**** | ****Constraints**** | ****Description**** | ****Notation**** |
| **id** | **Long** | **Pk, updatable = false, nullable = false** | **Pk, generate from shardingsphere** | **Global unique** |
| **orgId** | **Long** | **fk** | **Fk, refer Organization** |  |
|  |  |  |  |  |
|  |  |  |  |  |

# 4. User Interface Design (UI)

**This section is out of scope for this backend-focused DDD. Please refer to the separate "MMS-UI-Wireframes" document.**

# 5. Interface/API Design

**HTTP Method: POST**

**Request Headers:**

**Content-Type: application/json or application/x-www-form-urlencoded**

**Request Body:**

**Dtos extends RootDto**

**Response Body :**

**OpResult**

# 6. Non-Functional Requirement Implementation

**SEC-1 (Password Hashing): Implemented using the BCrypt algorithm in the MemberService.createMember method. The work factor is set to 12.**

**SEC-2/3/4 (Access Control & Data Protection):**

**Implemented via a JWT-based authentication filter on all API endpoints.**

**Authorization checks are performed within service methods**

**Sensitive data is masked in DTOs before being returned in API responses.**

**PERF-1 (Auth Response <2s): Achieved through efficient BCrypt hashing (tuned for ~1s on production hardware) and indexed database lookups on login\_id.**

**PERF-2/3 (Pagination & Caching):**

**Pagination is implemented at the database query level using LIMIT and OFFSET.**

**A caching strategy will cache frequently accessed, public member profiles by their id with a TTL of 5 minutes.**

**PERF-4 (1B Members): The chosen database (Mariadb) can handle this scale with proper indexing, partitioning on created\_at or organization\_id, and archiving strategies.**

**DATA-1 (Unique Login ID): Enforced by the unique constraint on the members.login\_id column.**

**USE-1/2 (Clear Errors & Consistent Format): All errors are caught and transformed into a consistent JSON error response body.**

# 7. Deployment & Operational Considerations

**Build Instructions: The application is built using Maven. Execute mvn clean package to produce a deployable JAR file.**

**Deployment Instructions: The JAR file is deployed to a Docker container and orchestrated via Kubernetes. Deployment is managed by CI/CD pipelines in Jenkins.**

**Configuration: Key configuration is externalized via environment variables:**

**DB\_URL: JDBC connection string for the database.**

**JWT\_SECRET: Secret key for signing JWTs.**

**VERIFICATION\_SERVICE\_URL: Endpoint for the external verification service.**

**Monitoring: The application will expose metrics on /actuator/metrics (Spring Boot Actuator) and will be integrated with Prometheus/Grafana for monitoring performance and error rates.**