**Software Requirements Document (SRD)**

**1. Introduction**

This document outlines the Software Requirements Specification (SRS) for the Dashboard Application, focusing on key functionalities such as Leave Management System (LMS) and Pods. The goal is to provide a clear reference for autonomous coding and implementation of the required features.

**2. Overall Description**

The Dashboard Application serves as a centralized interface for employees and managers to access various features, including:

* Applying for and managing leaves
* Viewing and managing Pods
* Dashboard displaying multiple tiles summarizing key application highlights.

The DNA ecosystem consists of multiple microservices designed to streamline enterprise operations. Each service is developed using a modern technology stack:

* **Backend:** FastAPI (Python, Pydantic for validation, SQLAlchemy for database interaction)
* **Database:** PostgreSQL
* **Communication:** REST APIs with WebSockets for real-time interactions
* **Deployment:** Cloud-based, containerized architecture using Docker and Kubernetes

Each microservice follows a modular and scalable design, supporting CRUD operations, authentication, Role-Based Access Control (RBAC), and asynchronous operations.

**3. Customers**

The primary users of this system include:

* **General Users:** Employees utilizing LMS and PODs Features.
* **Managers:** Supervisory roles with permissions for approval workflows.

**4. Functionality**

Each microservice/application provides distinct capabilities:

**Dashboard**

* Displays multiple tiles representing key insights from different applications.
* Supports real-time data updates and drill-down interactions.
* Allows configuration of displayed widgets based on user roles.

API Endpoints

Fetch Dashboard Data

**Request:**

GET /api/dashboard/tiles

Headers: { Authorization: Bearer <token> }

**Response:**

{

  "tiles": [

    { "id": "1", "title": "Leave Summary", "content": "10 leaves remaining" },

    { "id": "2", "title": "Pod Members", "content": "3 active members" }

  ]

}

**LMS (Leave Management System)**

* **General User:**
  + Submit leave requests with category selection (e.g., paid leave, sick leave, etc.).
  + View granted and pending leave requests.
  + Track available leave balances.
* **Manager:**
  + Approve or reject leave requests with comments.
  + Access reports of team leave history.
* **APIs & Endpoints:**
  + POST /leave/apply - Apply for leave.
  + GET /leave/status - Retrieve leave status.
  + PATCH /leave/approve/{id} - Approve/reject leave (Manager only).

API Endpoints

Apply for Leave

**Request:**

POST /api/lms/leaves/apply

Headers: { Authorization: Bearer <token> }

Body:

{

  "start\_date": "2025-03-15",

  "end\_date": "2025-03-18",

  "reason": "Family event"

}

**Response:**

{

  "message": "Leave request submitted successfully",

  "status": "pending"

}

Approve Leave (Manager Only)

**Request:**

PATCH /api/lms/leaves/{leave\_id}/approve

Headers: { Authorization: Bearer <token> }

Body:

{

  "status": "approved"

}

**Response:**

{

  "message": "Leave request approved",

  "status": "approved"

}

**PODs (Project Oriented Development)**

* **Manager:**
  + Assign employees to specific pods.
* **Employee:**
  + View assigned pod.
  + Recommend colleagues for inclusion.
* **APIs & Endpoints:**
  + POST /pods/assign - Assign employee to pod.
  + GET /pods/members - Retrieve pod members.
  + POST /pods/recommend - Recommend employees for pods.

API Endpoints

Get Pod Details

**Request:**

GET /api/pods/{pod\_id}/details

Headers: { Authorization: Bearer <token> }

**Response:**

{

  "pod\_id": "56789",

  "pod\_name": "Innovation Team",

  "members": [

    { "id": "1", "name": "John Doe", "role": "Lead Developer" },

    { "id": "2", "name": "Jane Smith", "role": "UI/UX Designer" }

  ]

}

Recommend an Employee for a Pod

**Request:**

POST /api/pods/{pod\_id}/recommend

Headers: { Authorization: Bearer <token> }

Body:

{

  "recommended\_user\_id": "3"

}

**Response:**

{

  "message": "Recommendation sent successfully"

}

**4. Authentication & Authorization**

API Endpoints

User Login

**Request:**

POST /api/auth/login

Body:

{

  "email": [user@example.com](mailto:user@example.com),

  "password": "securepassword"

}

**Response:**

{

  "token": "jwt-token-here",

  "user": { "id": "1", "role": "manager" }

}

Fetch Current User Details

**Request:**

GET /api/auth/user

Headers: { Authorization: Bearer <token> }

**Response:**

{

  "id": "1",

  "name": "John Doe",

  "role": "manager"

}

**5. Platform**

The system will operate using the following technology stack:

* **Backend:** FastAPI (Pydantic, SQLAlchemy)
* **Database:** PostgreSQL
* **Deployment:** Cloud-based, utilizing Docker and Kubernetes for container orchestration.
* **Data Exchange:** REST APIs, WebSockets for real-time operations.
* **CI/CD:** Automated deployment pipelines using GitHub Actions / Jenkins.

**6. Development Responsibilities**

* **Project Managers:** Oversee project execution and ensure milestones are met.
* **Backend Developers:** Implement FastAPI-based microservices, business logic, and database interactions.
* **QA/Testers:** Conduct unit, integration, and end-to-end testing.
* **DevOps Engineers:** Maintain infrastructure, deployment pipelines, and monitoring tools.

**7. User Class and Characteristics**

·       Ensure RBAC,

o    Manager can access both manager and employee related APIs

o   While, user can only access user specific APIs .

**8. System Features and Requirements**

**Functional Requirements**

* Secure authentication and RBAC implementation.
* Asynchronous API calls for background operations.
* CRUD operations for core entities across all services.
* Dashboard with real-time insights and analytics.

**Non-Functional Requirements**

* **Scalability:** Support for high user concurrency and horizontal scaling.
* **Security:** End-to-end encryption, data validation, and API rate-limiting.
* **Performance:** API response times below 300ms.
* **Availability:** 99.9% uptime with automated failover mechanisms.
* **Logging & Monitoring:** Centralized logging with alert-based anomaly detection.

**9. Common Mistakes to Avoid**

* **Ambiguous Requirements:** Clearly define workflows and API contracts.
* **Overcomplicated Workflows:** Ensure ease of use and minimal user friction.
* **Ignoring Performance Optimization:** Optimize queries, indexing, and caching.
* **Lack of Security Measures:** Enforce RBAC, data encryption, and secure API access.
* **Inadequate Testing:** Implement comprehensive unit and integration testing.