Software Design Document (SDD)

Requirement: 3.1 - User Management

---

1. Introduction

1.1 Purpose

This document outlines the design and technical implementation of the **User Management** module within the SaaS Management Platform. This module will support user registration, authentication, role-based access control, and team management.

1.2 Scope

The User Management module is a foundational part of the platform, providing secure and structured access for various types of users (e.g., Admin, IT Personnel, Finance, Security Officer, Team Member). This document covers:

- User registration and login
- Role-based access control (RBAC)
- Team management functionality

---

2. Architecture and Design Approach

2.1 Overview

The User Management module will be developed using a **microservices architecture** with separate services for authentication and user management to ensure modularity and scalability. These services will interact via HTTP and REST APIs.

2.2 Technologies

- **Backend**: Java with Spring Boot for REST API development
- **Database**: PostgreSQL for persistent user data storage
- **Authentication**: JSON Web Tokens (JWT) for secure, stateless authentication
- **Authorization**: Spring Security for role-based access control
- **Frontend**: React (for the platform's UI)
- **CI/CD**: Docker for containerization, Jenkins for CI/CD pipelines

2.3 High-Level System Components

- **Authentication Service**: Handles user authentication and JWT generation.

- **User Service**: Manages user profiles, roles, and permissions.
- **Database (PostgreSQL)**: Stores user data, roles, and team structures.
- **Notification Service**: Sends email notifications for registration, password reset, etc.

---

3. Detailed Design

3.1 Database Design

The **User Management** module will store data in the following main tables:

- **Users Table**
  - `user_id` (Primary Key)
  - `email` (Unique)
  - `password_hash` (hashed password for secure storage)
  - `created_at`, `updated_at`
  - `status` (active, suspended, deleted)
  - `role_id` (Foreign Key to Roles table)
- **Roles Table**
  - `role_id` (Primary Key)
  - `role_name` (e.g., Admin, IT Personnel, Finance, etc.)
- **Teams Table**
  - `team_id` (Primary Key)
  - `team_name`
  - `created_by` (Foreign Key to Users table)
- **UserTeams Table** (many-to-many relationship between users and teams)
  - `user_id` (Foreign Key to Users table)
  - `team_id` (Foreign Key to Teams table)

3.2 API Design

3.2.1 Authentication Service APIs

- **POST /auth/register**: Registers a new user and sends a confirmation email.
  - **Request**: `{ "email": "user@example.com", "password": "password123", "role_id": "role_id" }`
  - **Response**: `{ "message": "User registered successfully. Please verify your email." }`
- **POST /auth/login**: Authenticates the user and returns a JWT.
  - **Request**: `{ "email": "user@example.com", "password": "password123" }`

    - **Response**: `{ "token": "jwt_token_here" }`
- **POST /auth/logout**: Invalidates the JWT (optional if using stateless tokens).

3.2.2 User Service APIs

- **GET /users/{user_id}**: Fetch user profile and role information.
    - **Response**: `{ "user_id": 1, "email": "user@example.com", "role": "Admin", "teams": [...] }`
- **PUT /users/{user_id}**: Update user profile and role.
    - **Request**: `{ "role_id": "new_role_id", "team_ids": [1, 2] }`
- **POST /teams**: Create a new team.
    - **Request**: `{ "team_name": "Team A", "created_by": "user_id" }`
    - **Response**: `{ "team_id": 1, "team_name": "Team A" }`
- **POST /teams/{team_id}/add_user**: Add a user to a team.
    - **Request**: `{ "user_id": "user_id" }`

3.3 Authentication and Authorization Flow

1. **User Registration**:
    - Users register by providing an email, password, and optional role. The `auth/register` API hashes the password and stores the user in the database with a default "pending verification" status.
2. **Email Verification**:
    - After registration, an email is sent with a verification link. Once verified, the user's status is updated to "active".
3. **Login and JWT Generation**:
    - Upon login, the `auth/login` API verifies the credentials and generates a JWT with claims that include user ID and role. The token is returned to the client for use in authenticated requests.
4. **Role-Based Access Control (RBAC)**:
    - **Spring Security** intercepts each request, verifies the JWT, and checks the user's role. Access to resources (like creating new users or assigning roles) is controlled based on the role.
5. **Session Management**:
    - JWT tokens are stateless, so there's no need to maintain sessions in the backend. Tokens are valid until they expire, and users are required to re-authenticate once the token expires.

3.4 UI/Frontend Design

The frontend will have the following components in the **User Management Module**:

- **Login Page**: Accepts user credentials and calls `auth/login`.
- **Registration Page**: Allows new users to register and calls `auth/register`.

- **User Dashboard**:
  - Displays a list of teams, roles, and profile information.
  - Admins have additional options for user and team management.
- **Team Management Interface**:
  - Allows admins to create new teams and add/remove users from teams.

Each component will leverage API calls to interact with the backend, and `React Context` or a similar state management solution (like Redux) will store the JWT and user role locally.

---

4. Security Considerations

4.1 Data Protection

- **Password Hashing**: Use `bcrypt` for hashing passwords to protect user credentials.
- **JWT Expiry and Refresh**: Set a reasonable expiry on JWT tokens (e.g., 1 hour) and implement token refresh to maintain security.

4.2 Role and Permission Checks

- **Backend Authorization**: Role checks will be enforced on the backend using Spring Security annotations (e.g., `@PreAuthorize("hasRole('ADMIN')")`).
- **Frontend Authorization**: Frontend components will be conditionally rendered based on the user's role to prevent unauthorized actions.

4.3 Email Verification

- Email verification is required for new users. The platform sends an email with a secure token-based link that updates the user's status to "active" upon verification.

---

5. Testing Strategy

5.1 Unit Testing

- Test individual components (e.g., registration, login, RBAC) to ensure they work independently.

5.2 Integration Testing

- Ensure the `auth` and `user` services communicate properly, focusing on the registration, login, and RBAC flows.

5.3 End-to-End (E2E) Testing

- Test the complete user journey (registration to login, role-based dashboard, team management).

5.4 Security Testing

- Verify that only authorized roles can access restricted endpoints.
- Conduct penetration testing on the login and user management APIs to ensure robustness.

---

6. Deployment Considerations

- **Database Migrations**: Use a tool like Flyway or Liquibase for PostgreSQL database migrations to handle changes to the schema over time.
- **CI/CD Pipeline**: Set up a Jenkins pipeline to automate tests and deploy updates to staging/production environments.
- **Containerization**: Use Docker to containerize the microservices, with separate containers for the auth service, user service, and PostgreSQL database.