Part A

# Task 1: Roles (5%)

**Team Size:** 2 members **Methodology:** Agile (Scrum framework)

1. Santosh - Scrum Master

**Primary Responsibilities:**

* **Facilitate Scrum Events:** Organize and lead daily standups, sprint planning, sprint reviews, and retrospectives
* **Remove Impediments:** Identify and eliminate blockers that prevent the team from achieving sprint goals
* **Shield the Team:** Protect developers from external distractions and scope creep during sprints
* **Coach Agile Practices:** Guide the team in following Scrum principles and help improve processes
* **Stakeholder Communication:** Act as liaison between the development team and product owner/stakeholders
* **Metrics and Reporting:** Track sprint progress, velocity, and burndown charts
* **Process Improvement:** Continuously identify areas for team efficiency improvements

**Secondary Responsibilities**

* May also contribute to development work when not performing Scrum Master duties
* Handle project documentation and sprint artifacts
* Coordinate with external dependencies and resources

2. Sami - Developer/Team Member

**Primary Responsibilities:**

* **Feature Development:** Write, test, and implement code according to user stories and acceptance criteria
* **Code Quality:** Ensure code follows best practices, is well-documented, and maintainable
* **Sprint Commitment:** Estimate effort for user stories and commit to deliverables during sprint planning
* **Daily Participation:** Actively participate in daily standups, providing updates on progress and impediments
* **Testing:** Conduct unit testing, integration testing, and collaborate on user acceptance testing
* **Code Reviews:** Review code changes and provide constructive feedback
* **Technical Documentation:** Create and maintain technical documentation for developed features

**Collaborative Responsibilities:**

* Work closely with Santosh to break down user stories into manageable tasks
* Provide input during sprint planning and retrospectives
* Communicate technical challenges and dependencies early
* Participate in architecture and design decisions

**Shared Team Responsibilities**

Both team members collaborate on:

* **Sprint Planning:** Defining sprint goals and selecting user stories from the product backlog
* **Definition of Done:** Establishing and maintaining quality standards for completed work
* **Continuous Improvement:** Identifying process improvements during retrospectives
* **Stakeholder Engagement:** Participating in sprint reviews and demos
* **Risk Management:** Identifying and addressing project risks early

# Task 2: Project Implementation in Agile (12%)

## **Quick Setup**

Project: Employee Management System

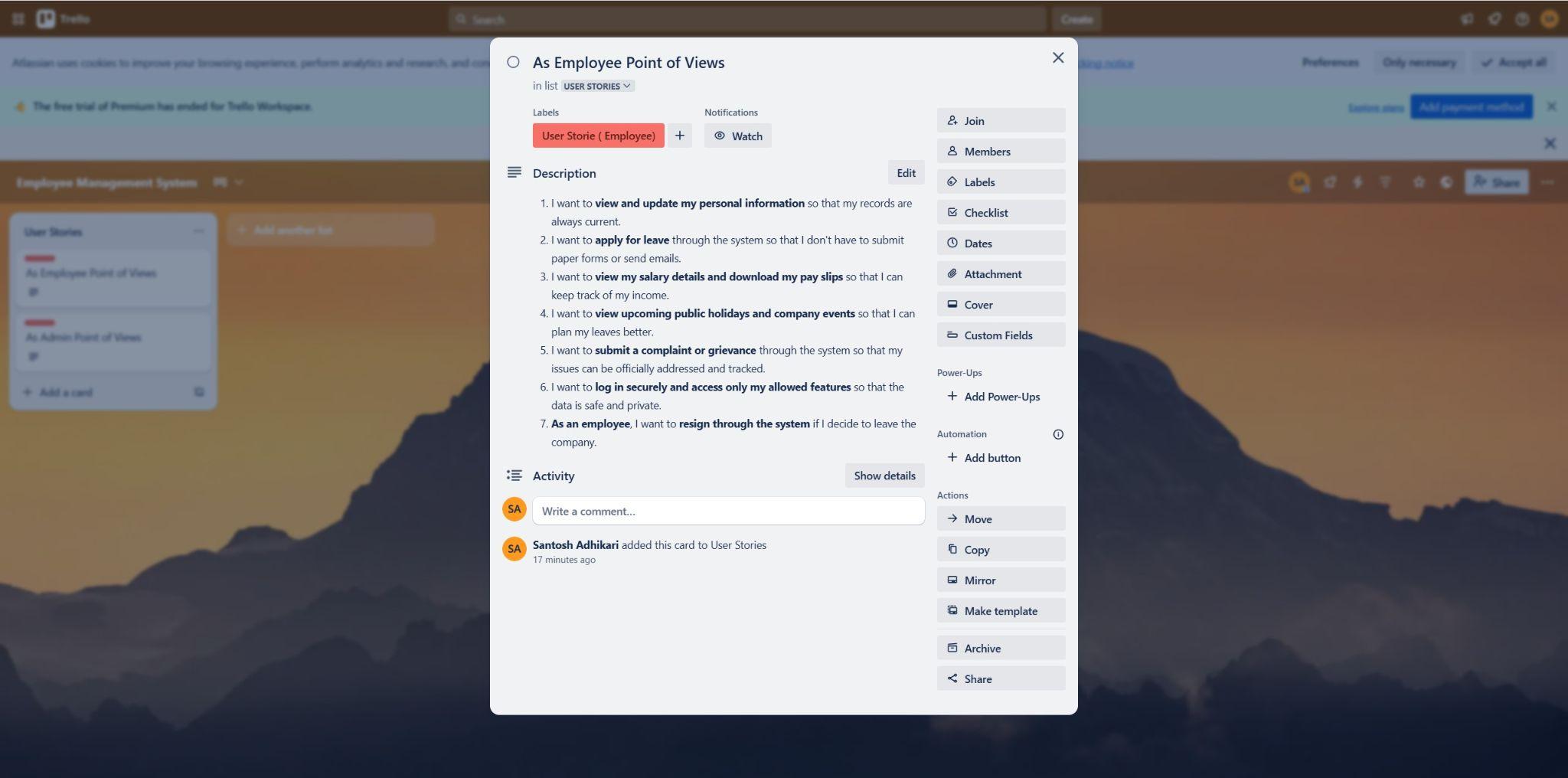
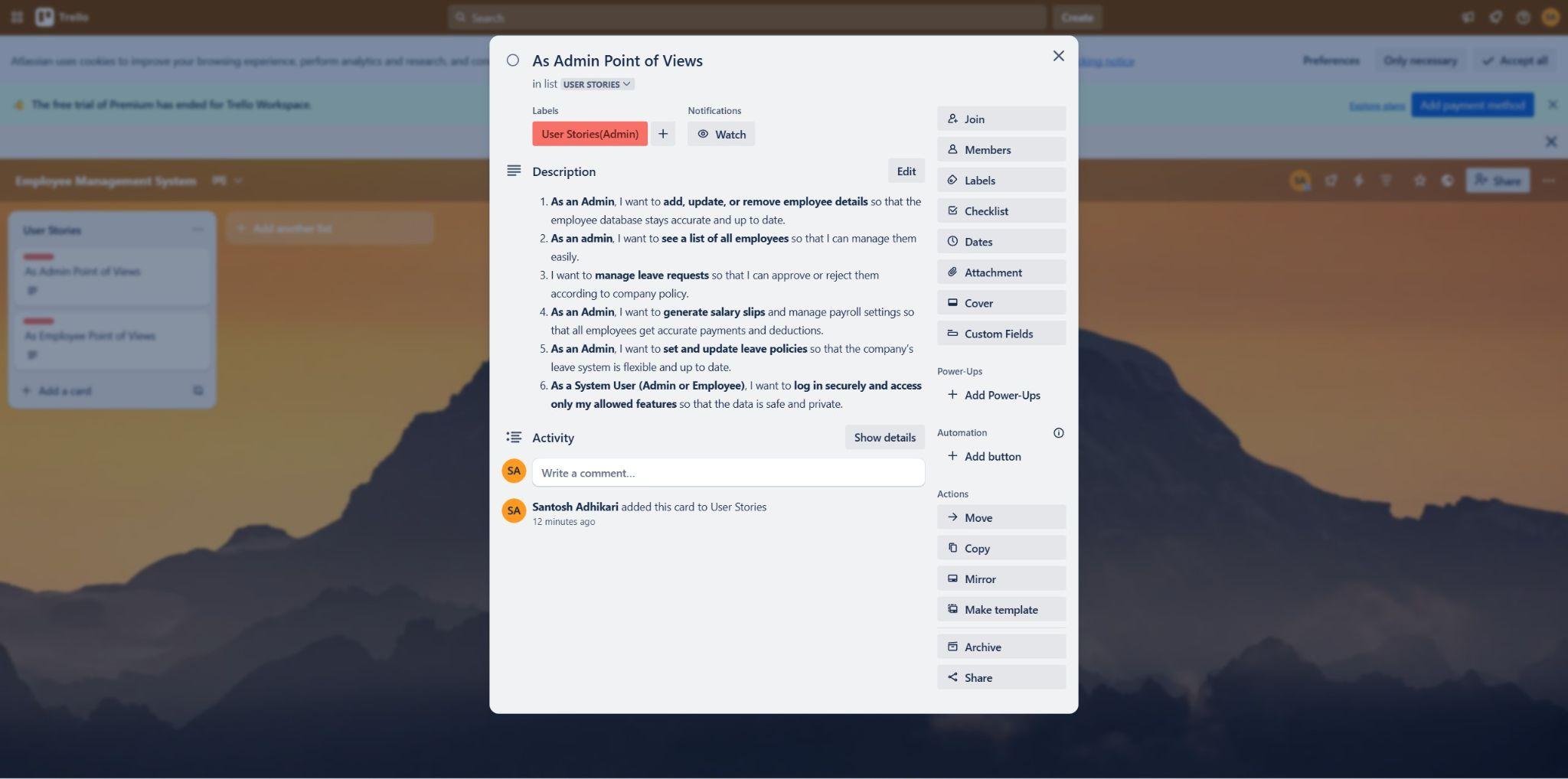
Tools: Teams + GitHub + Trello

Timeline: 3 weeks (5, 6, 7)

Our Employee Management System project breaks down 13 user stories (covering admin and employee features) into three weekly sprints, where each sprint delivers working functionality like secure login, leave management, or payroll features that we can actually demonstrate. We follow the Scrum framework with sprint planning meetings to assign tasks, daily scrums via MS Teams for quick progress updates, ongoing development with regular GitHub commits, and sprint reviews to show what we've accomplished each week. Our team collaborates using MS Teams for daily communication, GitHub for code management and tracking issues (like the dropdown color problems), and Trello boards to move user stories through development stages. We document everything we're already doing - our Teams conversations, GitHub commits, sprint meetings, and feature demonstrations - to prove we successfully used Agile methodology while building our EMS application.

**User Stories:**

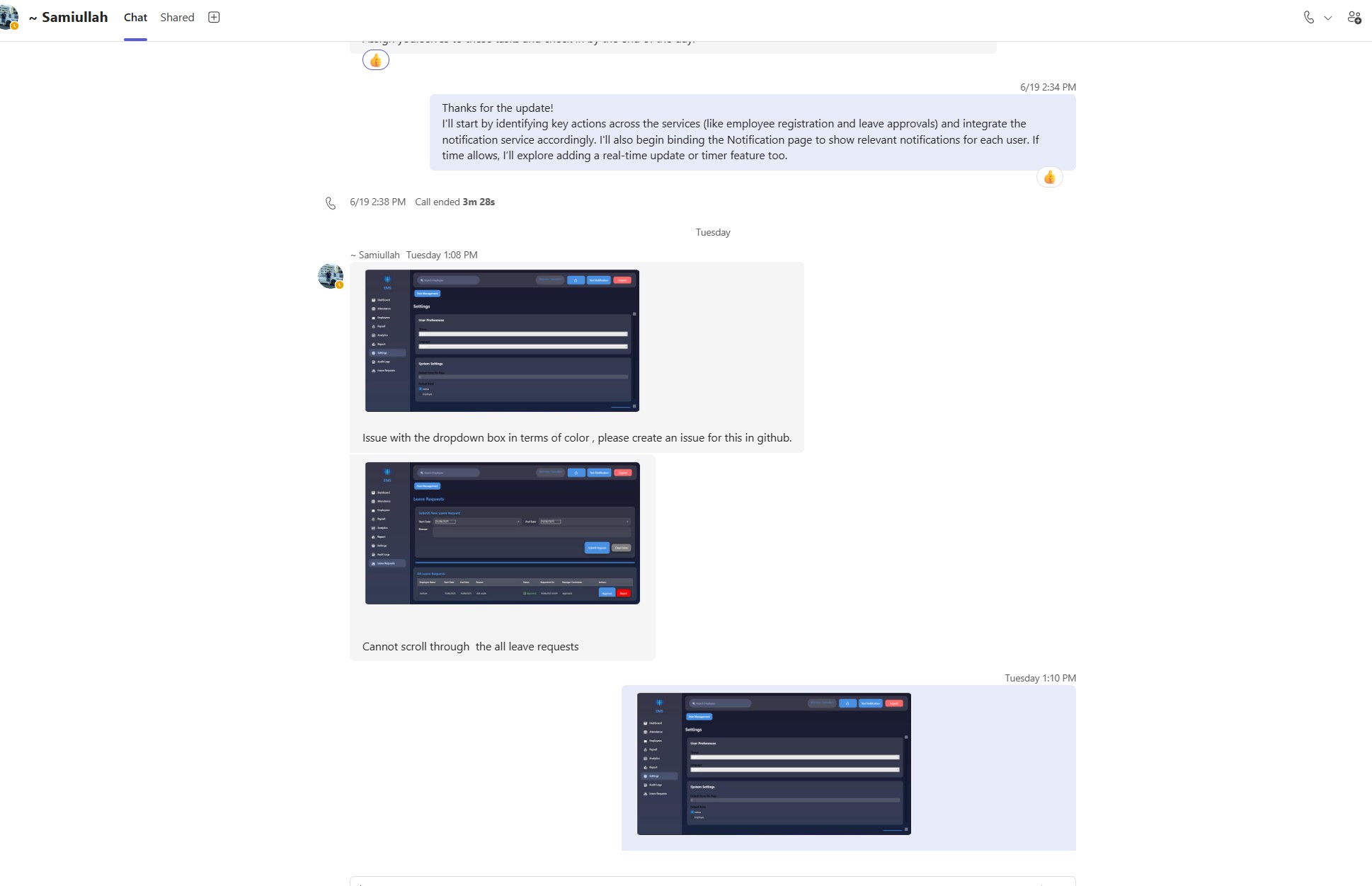
We collected user stories by discussing real-world use cases of an employee management system and identifying the key tasks needed by both admins and users. Our team (Santosh and Sami) communicated regularly to define and refine these requirements. To organize and track the progress, we used Trello, where we created cards for each user story. These cards were categorized into columns such as "To Do," "In Progress," and "Completed" to clearly show the development status of each feature. This helped us stay organized and follow the Agile workflow effectively.

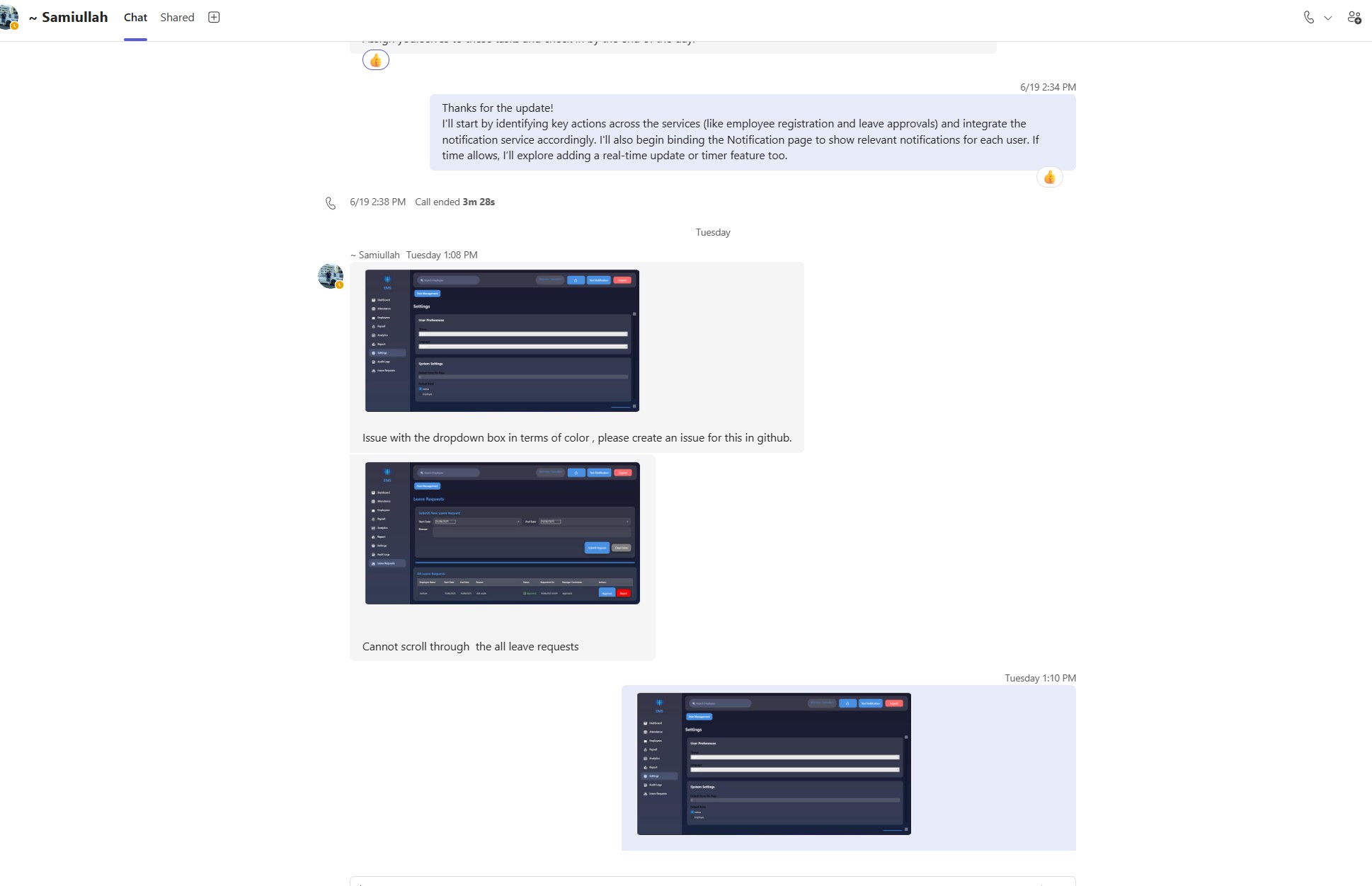


### **Evidence of Communication Between Team Members**

Throughout the project, effective communication was maintained between team members (Santosh and Sami) to ensure smooth collaboration and progress. We regularly discussed tasks, shared updates, and resolved issues through:

* **Microsoft Teams Video Calls** – We held regular meetings to plan features, assign responsibilities, and review progress.
* **Chat Messaging** – We used Teams chat to share quick updates, ask questions, and coordinate daily tasks.



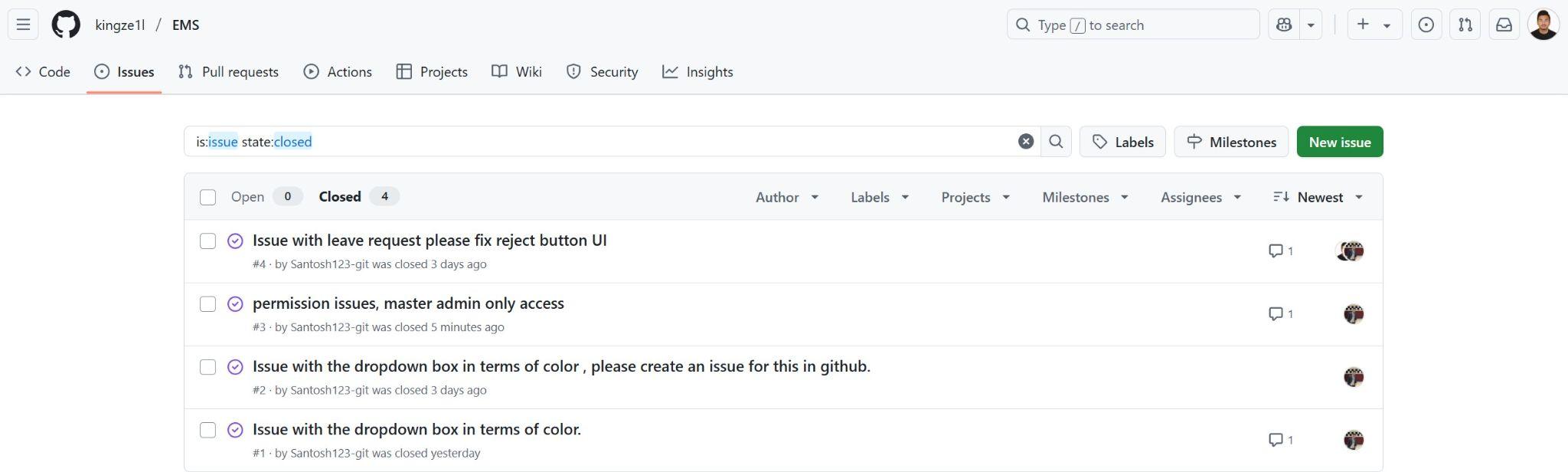


## **Evidence for Assigning issues in the github:**

In our project, we used **GitHub Issues** to track tasks, bugs, and improvements. Each issue was assigned to a specific team member based on their role and expertise. For example, if a UI bug was reported, it was assigned to the member responsible for the front-end.

After assigning, the responsible person worked on the issue and committed their changes with clear commit messages referencing the issue (e.g., "Fixes #5 – Login button alignment"). Once the task was completed and tested, the issue was marked as **closed** in GitHub.

This process helped us manage our workflow, divide responsibilities clearly, and maintain a proper history of work done throughout the project.



Some video screen shot for the video call meeting for the project:

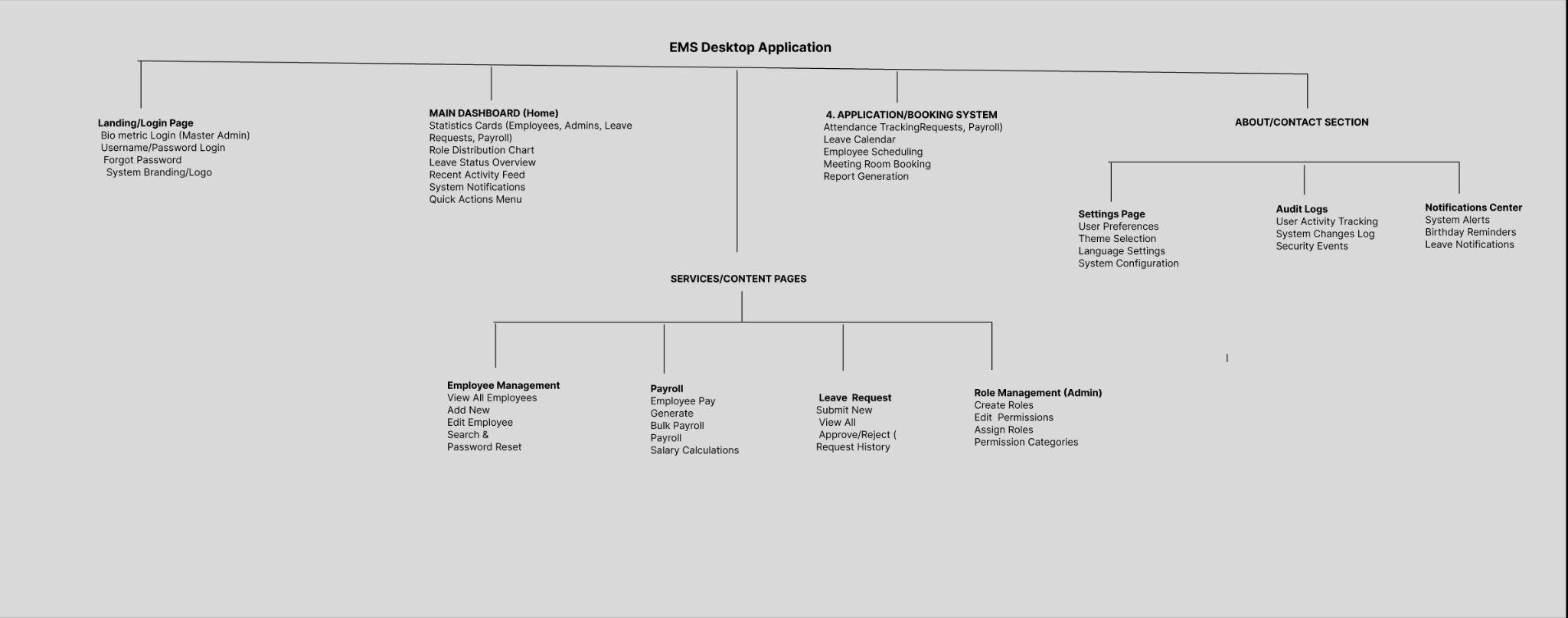
|  |  |
| --- | --- |

# Task 3: Design and Implementation of the Project (65%)

## Task 3.1 UI Design (15%)

UI Design (10%)

1.Sitemap:



2.Style Guide:

Topography

* Main Headings:Bold
* Body Text:Regular
* Button Text:: SemiBold

Colour Palette

* Primary Colors:#4A90E2 (RGB: 74, 144, 226)
* Background Gradient:#1a1a2e (RGB: 26, 26, 46)
* Functional Colors:#28A745 (RGB: 40, 167, 69)
* Glass/Transparency Effects:#20FFFFFF (White with 12% opacity)

Imagery,Buttons and Icons

* **Logo:** Minimalist people/team icon in white/blue
* **Background:** Animated gradient with floating geometric elements
* **Charts:** LiveCharts with blue color scheme
* **Illustrations:** Simple, flat design with consistent color palette

Icon Style:

* **Type:** Unicode emoji icons and Material Design icons
* **Size:** 16px (small), 20px (medium), 24px (large)
* **Color:** Inherits from parent or #4A90E2
* **Examples:**
  + Dashboard: 📊
  + Employees: 👥
  + Payroll: 💰
  + Settings: ⚙️
  + Calendar: 📅
  + Reports: 📈

Visual Effects:

**Drop Shadows:** Blur radius 10-30px, various opacities

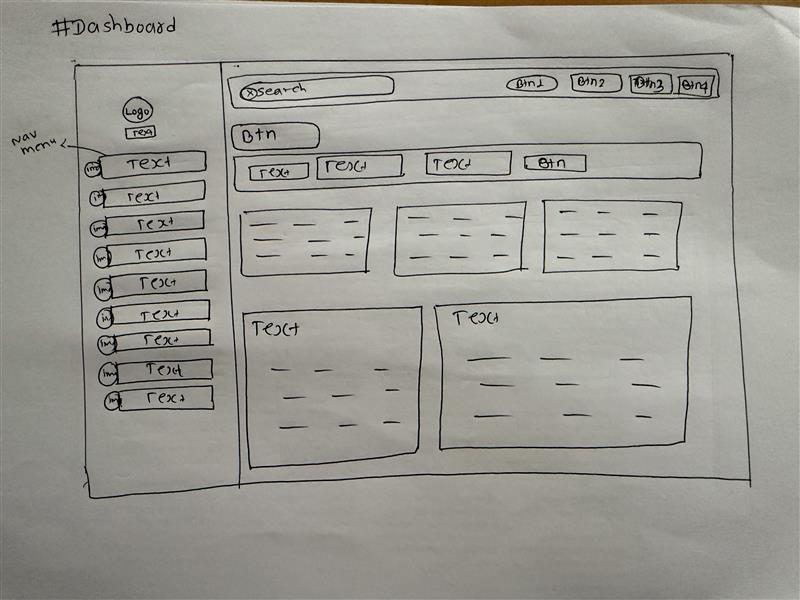
**Corner Radius:** 10-20px for cards, 12px for buttons

**Animations:** 0.3s duration with easing functions

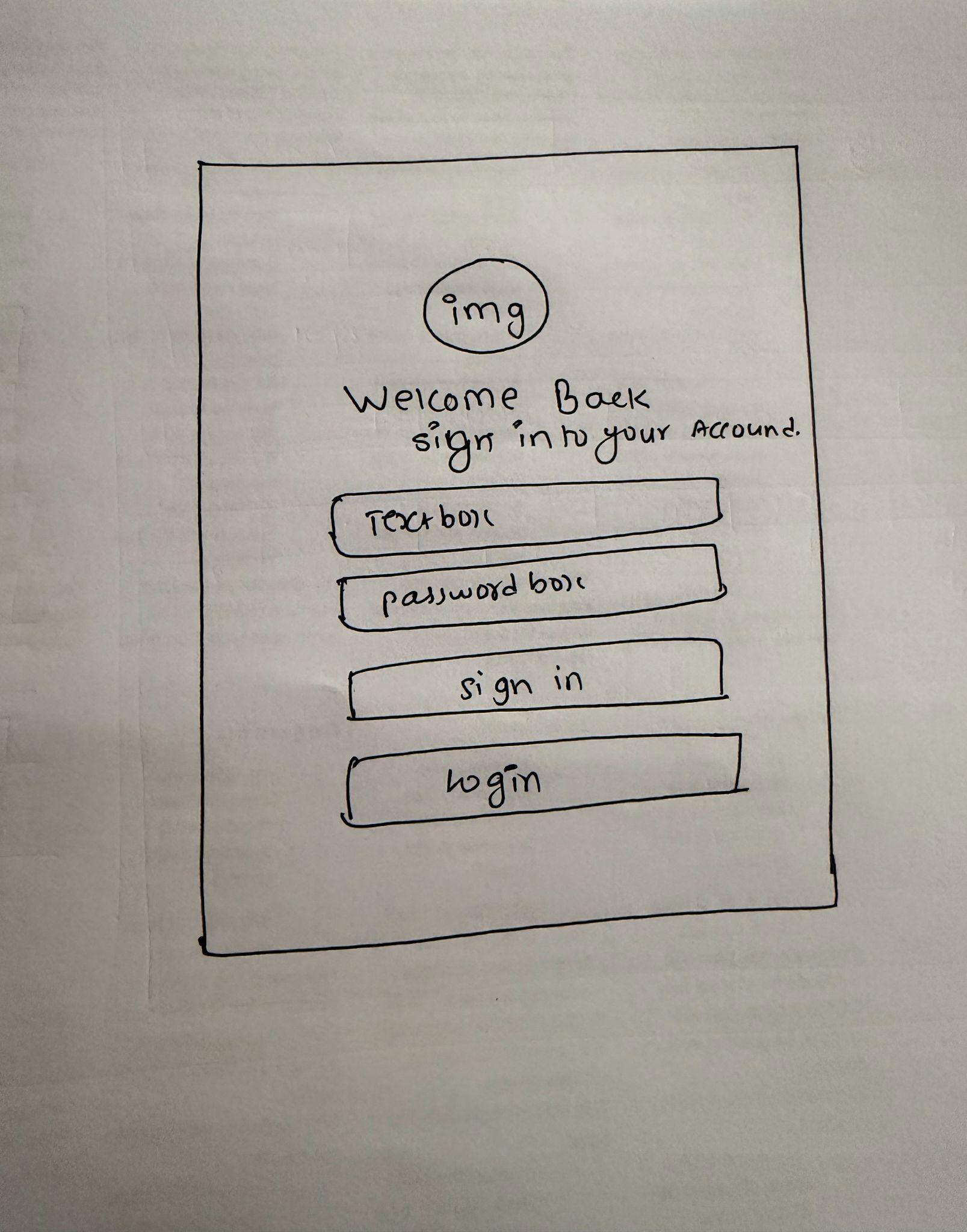
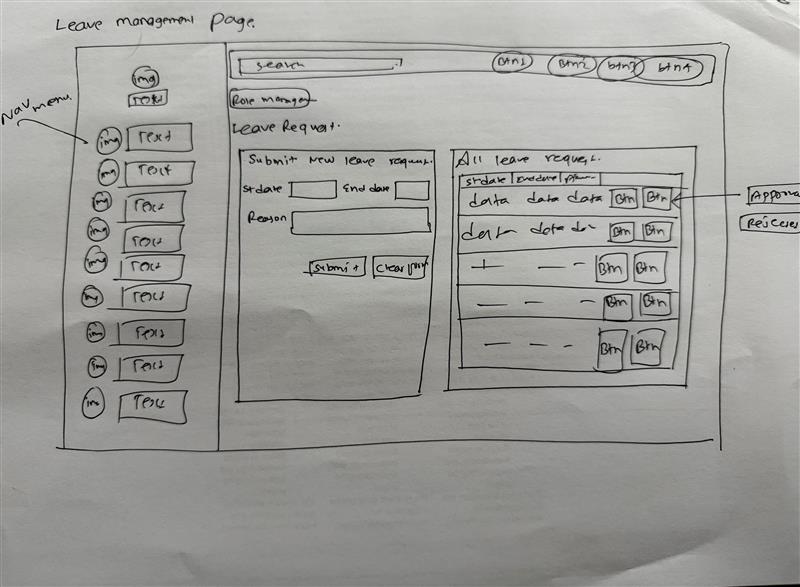
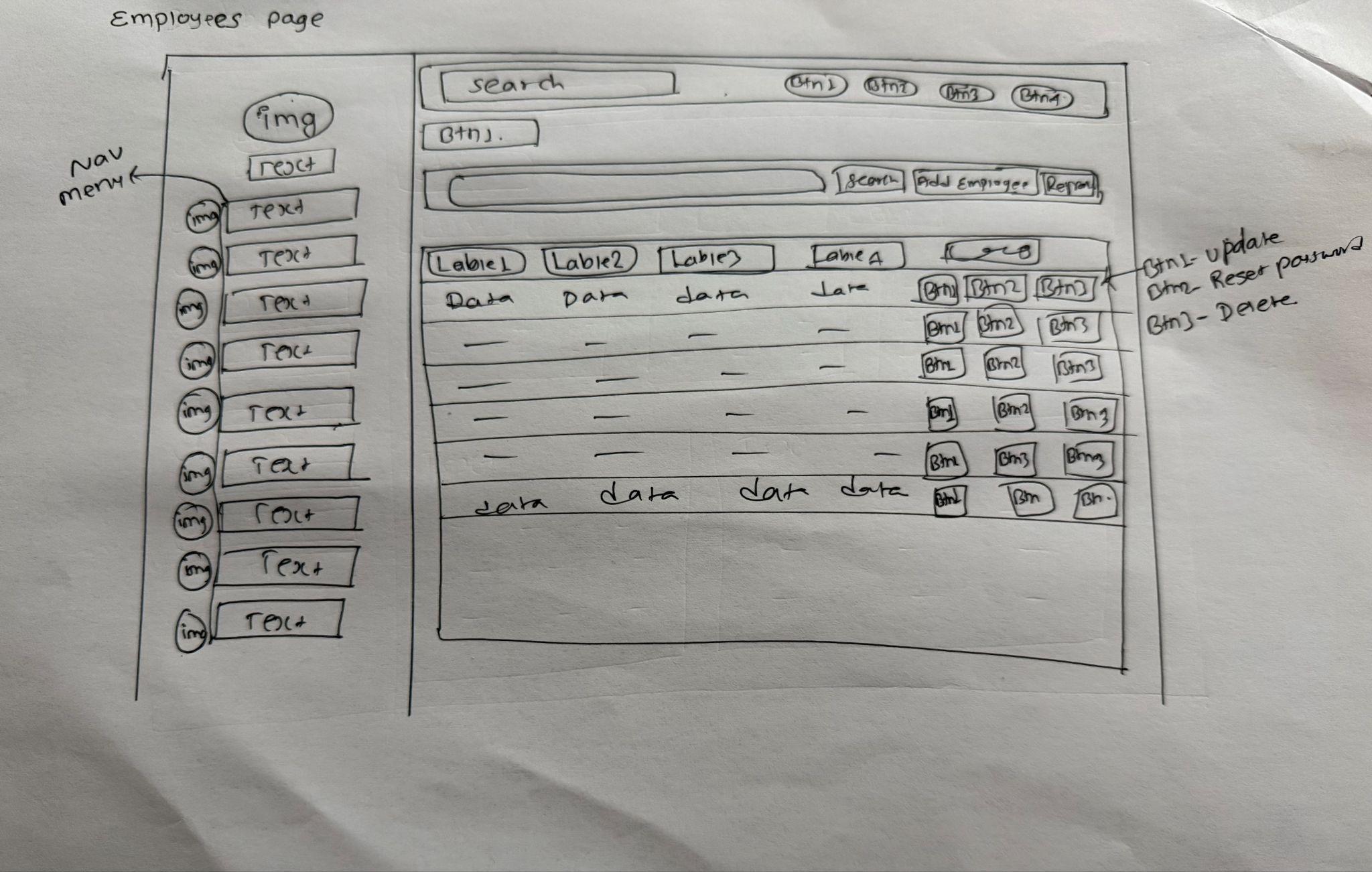
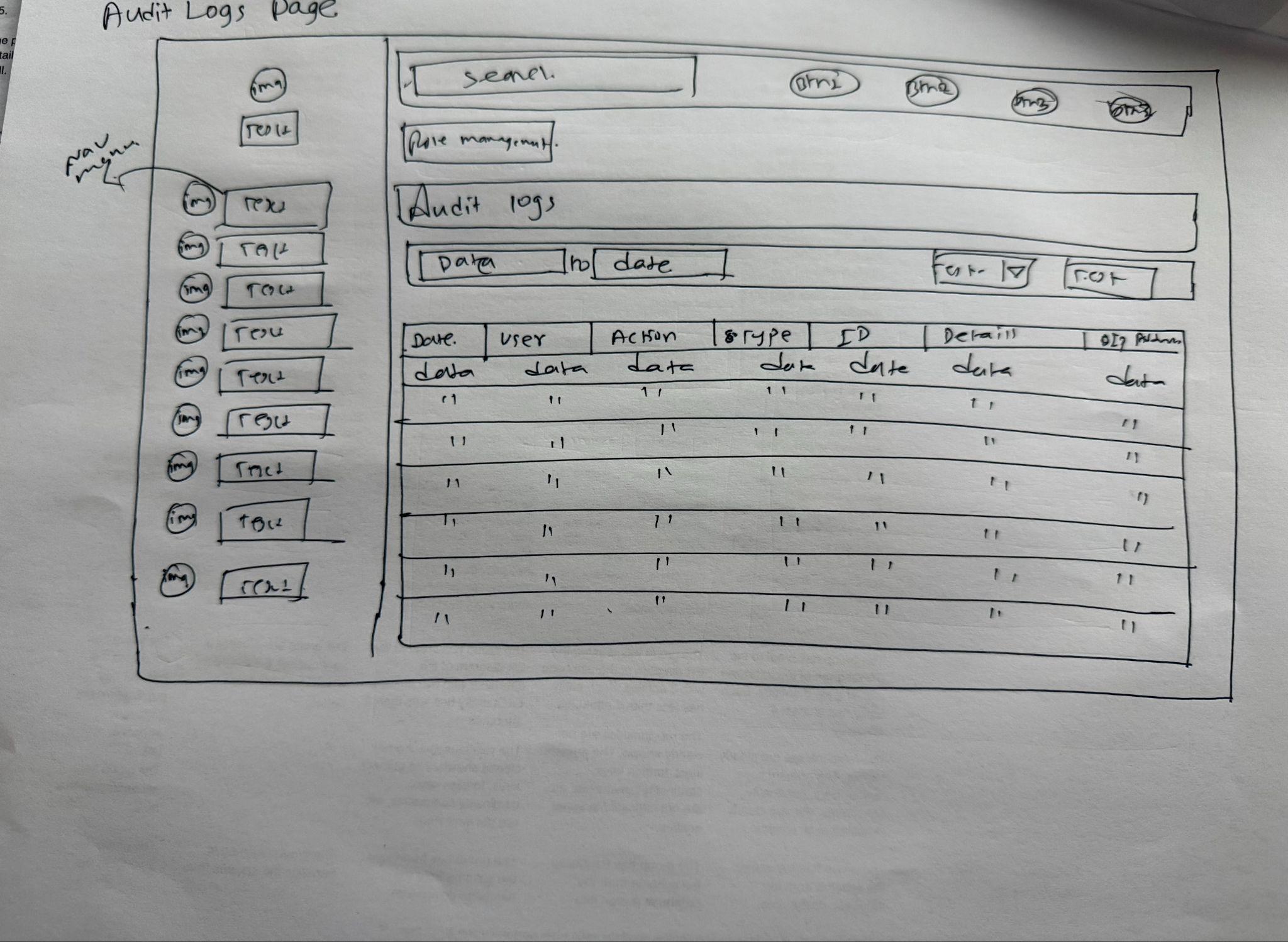
**Blur Effects:** Used for background elements and glass morphism

3.Lo-Fi Wireframes/Prototype:

1.DashBoard Page

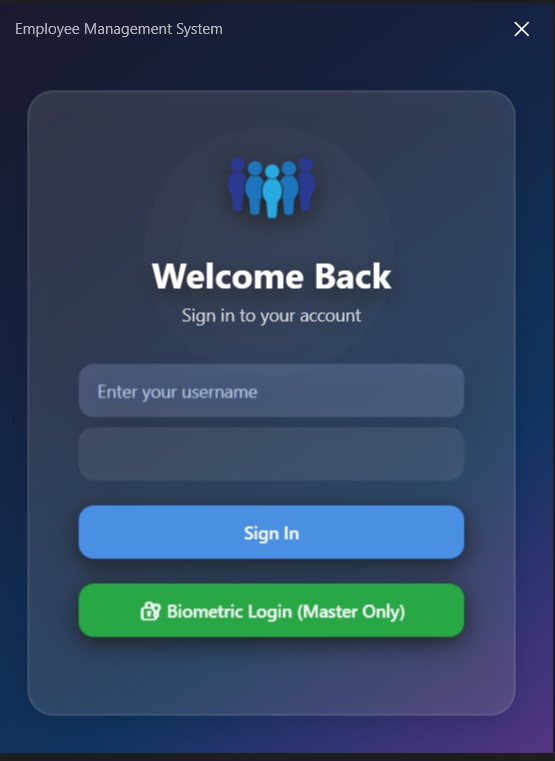


2.Audit Logs Page



Hi-Fi Prototype /Desktop Interface Development (5%)

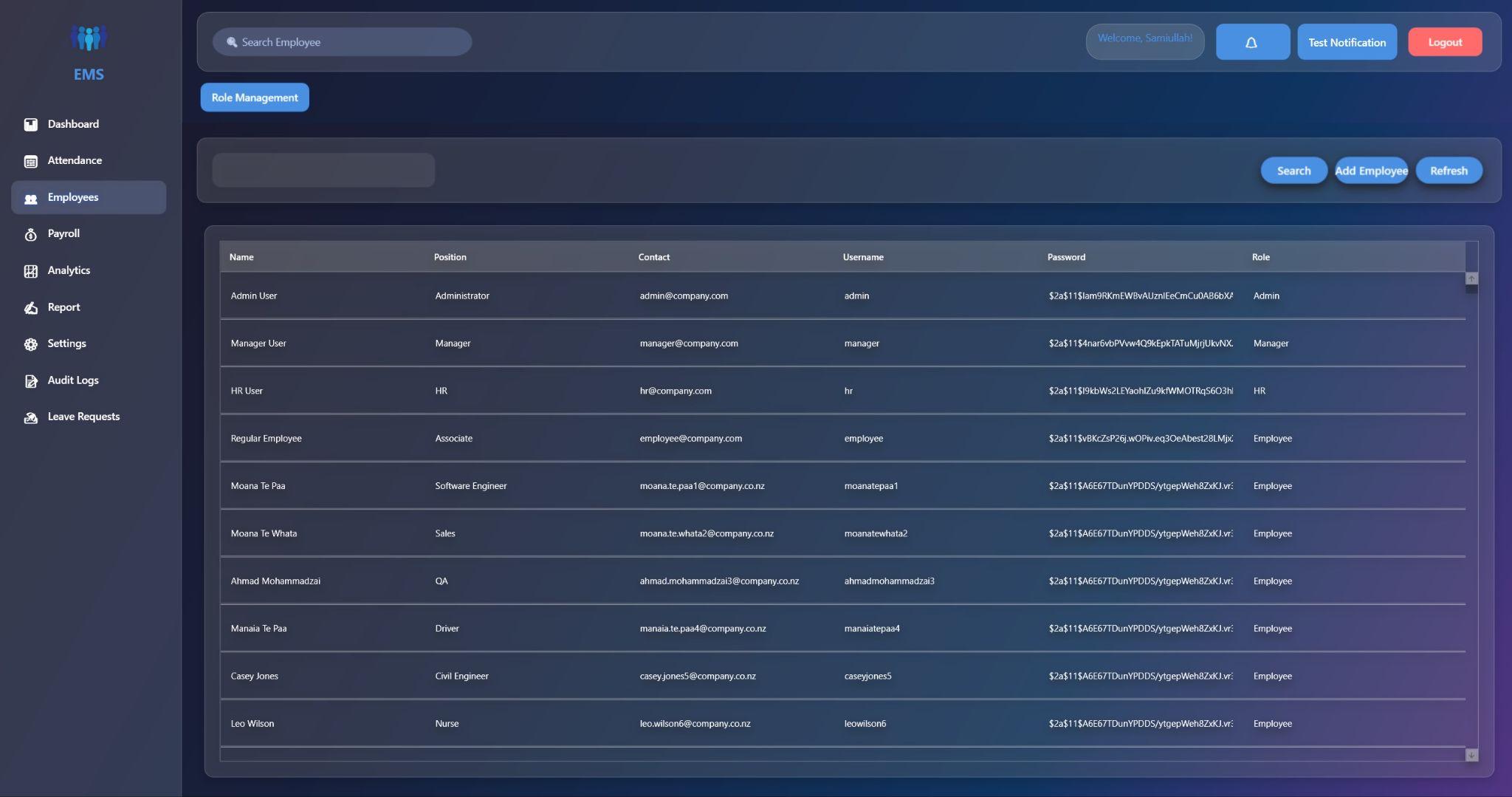
1. Login Page



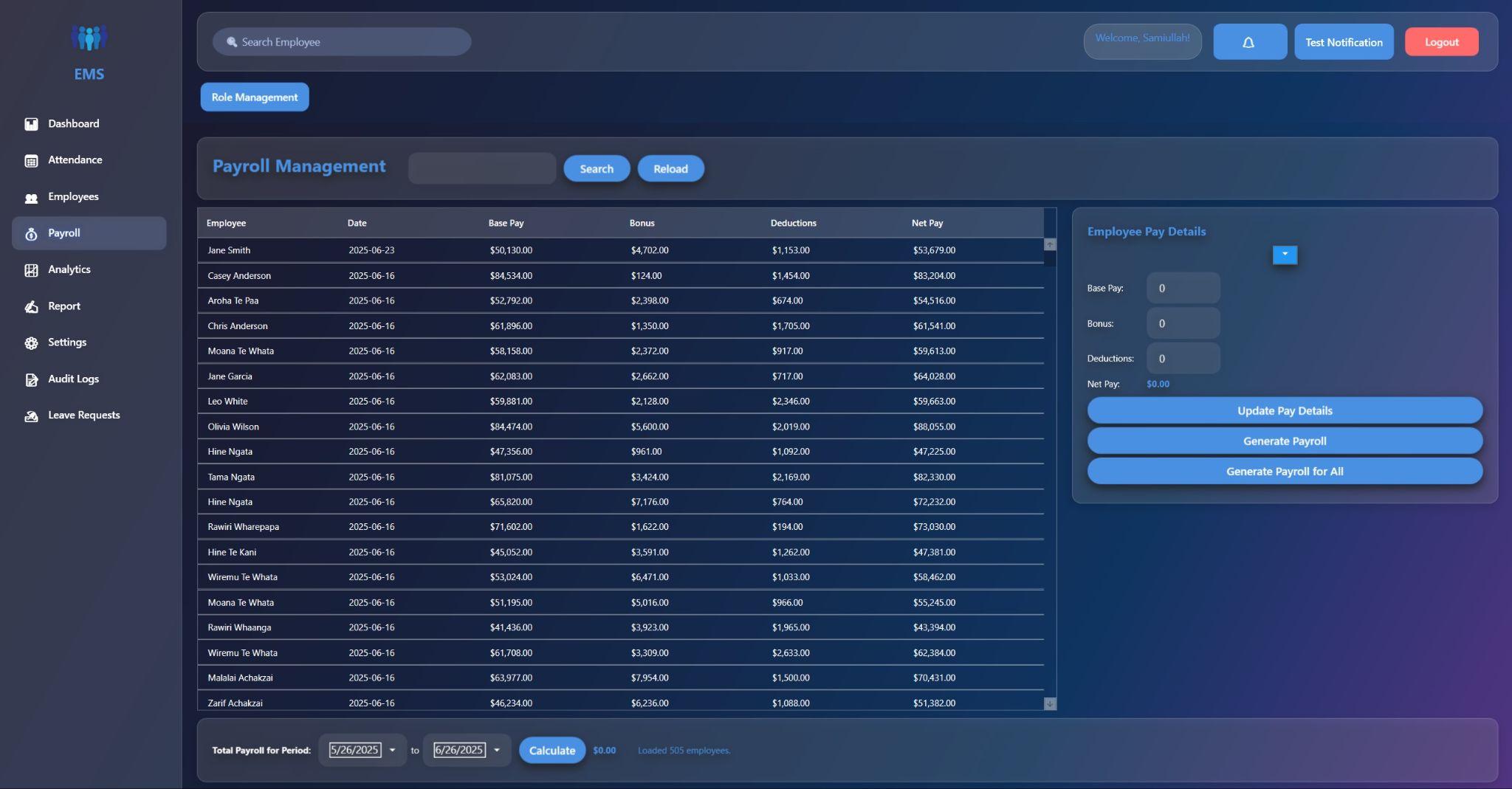
1. DashBoard



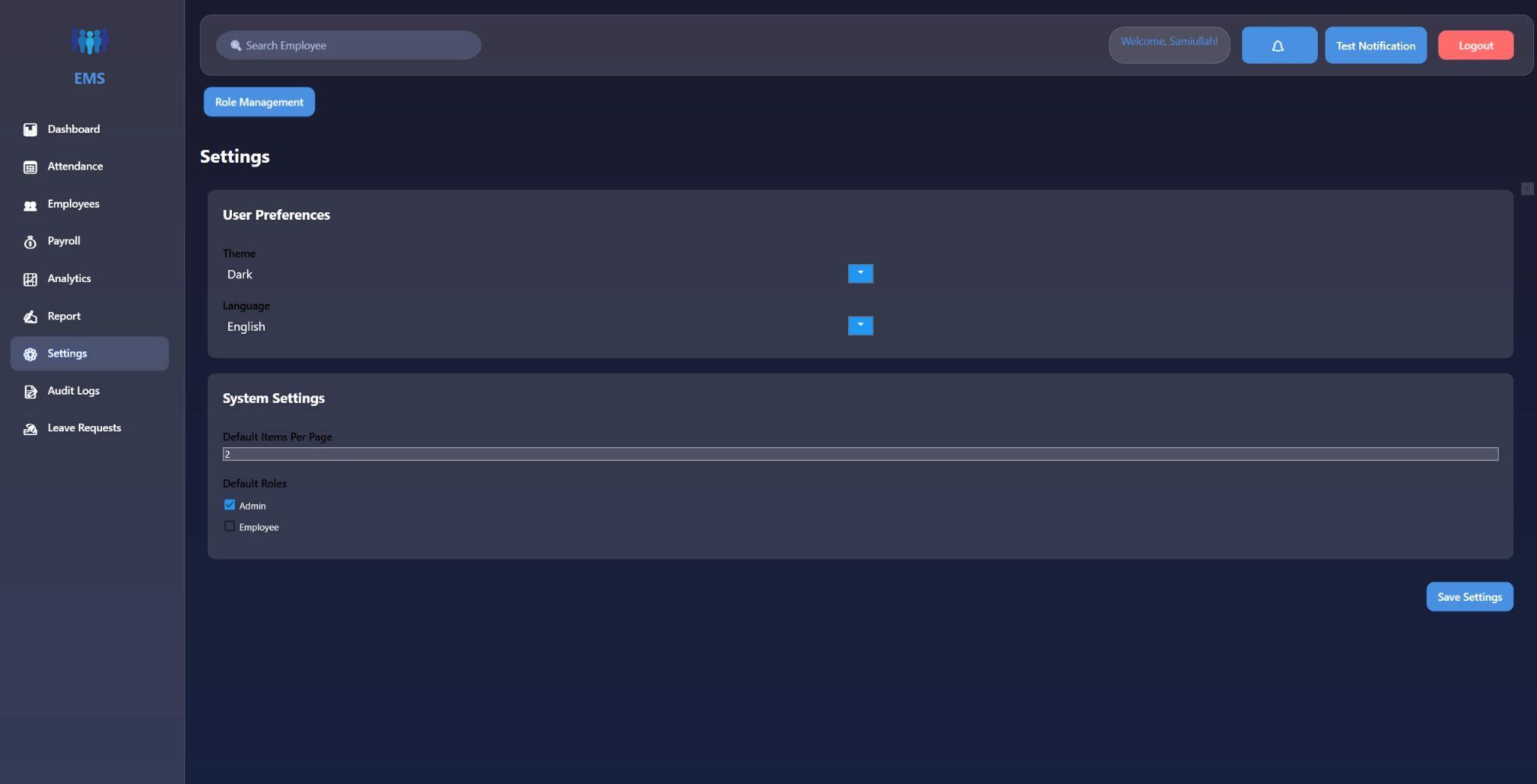
1. Employee Management



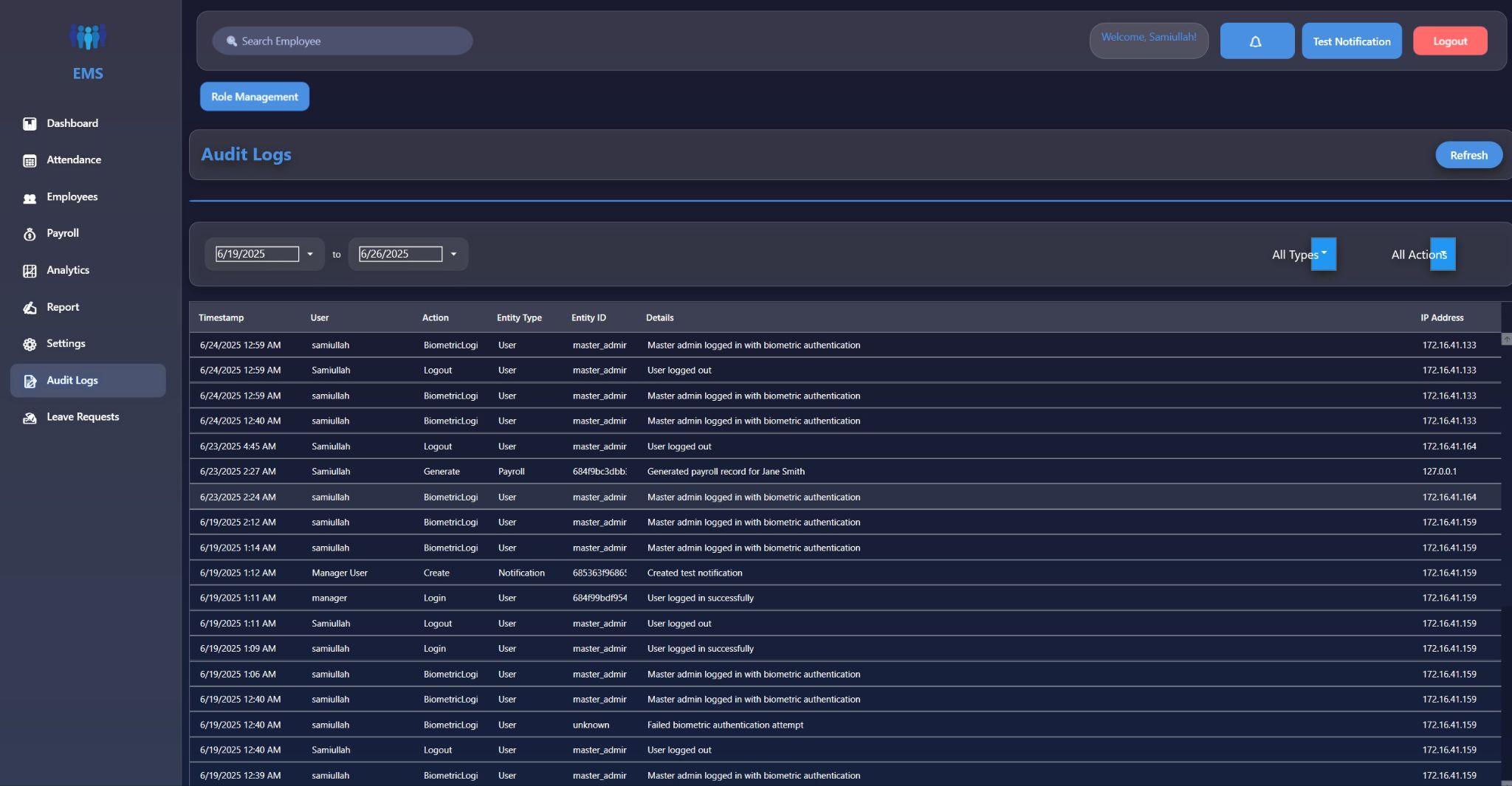
1. Payroll Management



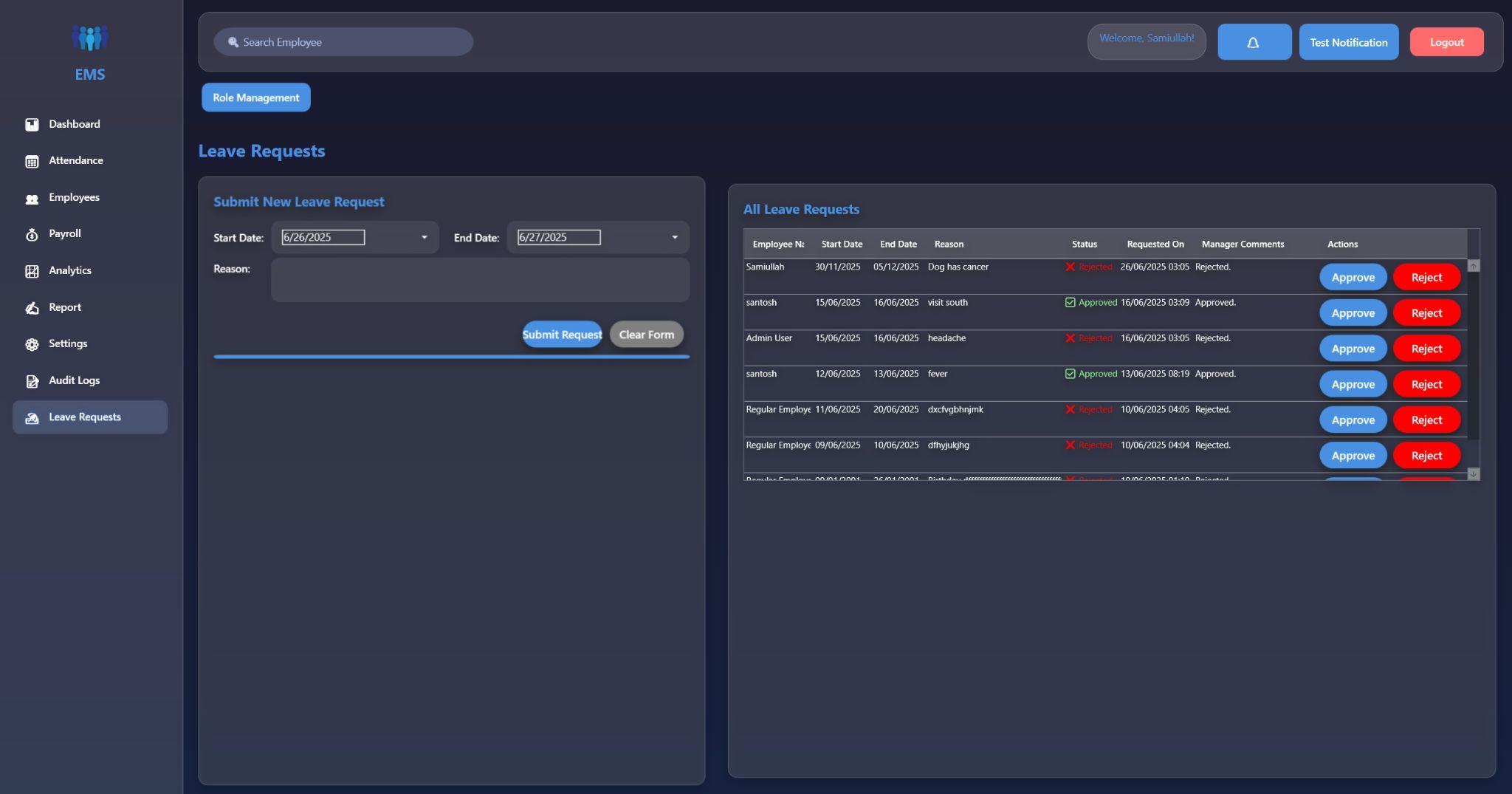
1. Settings



1. Audit Logs



1. Leave Request



# Task 3.2 Software Development (25%)

https://github.com/kingze1l/EMS.git

### Software Design (10%)

### Understanding the Requirements

Four Key Requirements Identified:

1. **Employee Management Requirement**
   * The system must allow administrators to add, edit, view, and delete employee records including personal information, job positions, contact details, and role assignments with proper authentication and authorization.
2. **Payroll Processing Requirement**
   * The system must calculate and manage employee payroll including base pay, bonuses, deductions, and net pay calculations with the ability to generate individual and bulk payroll records with audit trails.
3. **Leave Request Management Requirement**
   * The system must provide functionality for employees to submit leave requests and for managers/HR to approve or reject these requests with proper workflow management and notification system.
4. **Role-Based Access Control Requirement**
   * The system must implement a comprehensive permission system where different user roles (Admin, Manager, HR, Employee) have specific access rights to different features and data within the application.

### Identify the use of OOP concepts

## Abstraction

* **Interfaces (IEmployeeService, IPayrollService, etc.)**

**public interface IEmployeeService**

**{**

**Task<IEnumerable<Employee>> GetAllEmployeesAsync(UserRole currentUserRole);**

**Task<bool> AddEmployeeAsync(Employee employee);**

**}**

The project uses interfaces to define contracts for services without exposing implementation details. For example, IEmployeeService declares methods like GetAllEmployeesAsync() without revealing how data is retrieved from MongoDB.

* **ViewModelBase Abstract Class**

public class ViewModelBase : INotifyPropertyChanged

{

public event PropertyChangedEventHandler? PropertyChanged;

protected virtual void OnPropertyChanged([CallerMemberName] string? propertyName = null)

{

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

}

}

The ViewModelBase class provides common functionality like OnPropertyChanged() for all ViewModels. This abstracts the implementation of INotifyPropertyChanged so derived classes only focus on their specific logic.

## Encapsulation

* Employee Model Properties

public class Employee

{

public string Id { get; set; } = string.Empty;

public required string Name { get; set; }

public required string Position { get; set; }

public required string Password { get; set; }

public required UserRole UserRole { get; set; }

}

The Employee class encapsulates data like Name, Position, and Password with public getters/setters that could include validation logic. Private backing fields protect the internal state while properties provide controlled access to the data.

* Service Layer Data Protection

public class EmployeeService : IEmployeeService

{

private readonly IMongoCollection<Employee> \_employees;

private Employee SanitizeEmployeeData(Employee employee)

{

// Private method - encapsulated logic

return new Employee { /\* limited data \*/ };

}

}

Services like EmployeeService encapsulate database operations and business rules within private methods. External classes can only access employees through public methods like GetAllEmployeesAsync(), hiding the MongoDB connection details.

## Inheritance

* ViewModel Hierarchy

public class DashboardViewModel : ViewModelBase

{

private int \_totalEmployees;

public int TotalEmployees

{

get => \_totalEmployees;

set => SetProperty(ref \_totalEmployees, value); // Inherited functionality

}

}

All ViewModels (DashboardViewModel, EmployeeViewModel, etc.) inherit from ViewModelBase to share common functionality. This eliminates code duplication for property change notifications and provides a consistent base structure.

* Command Class Inheritance

public class RelayCommand : ICommand

{

public event EventHandler? CanExecuteChanged;

public bool CanExecute(object? parameter) { /\* implementation \*/ }

public void Execute(object? parameter) { /\* implementation \*/ }

}

public class AsyncRelayCommand : ICommand

{

// Inherits ICommand contract but implements async behavior

}

RelayCommand and AsyncRelayCommand both implement the ICommand interface, inheriting the command pattern structure. Each class specializes the base command behavior for synchronous and asynchronous operations respectively.

## Polymorphism

* Service Interface Implementations

// In DI Container setup

services.AddSingleton<IEmployeeService, EmployeeService>();

// Usage - polymorphic behavior

IEmployeeService employeeService = new EmployeeService();

// Could be MockEmployeeService for testing

Different service implementations can be used interchangeably through their interfaces, like IEmployeeService employeeService = new EmployeeService(). This allows for easy testing with mock services or switching between different data sources without changing client code.

* Command Pattern Polymorphism

public class EmployeeViewModel : ViewModelBase

{

public ICommand AddEmployeeCommand { get; }

public ICommand RefreshCommand { get; }

public EmployeeViewModel()

{

AddEmployeeCommand = new AsyncRelayCommand(AddEmployee); // Async

RefreshCommand = new RelayCommand(Refresh); // Sync

}

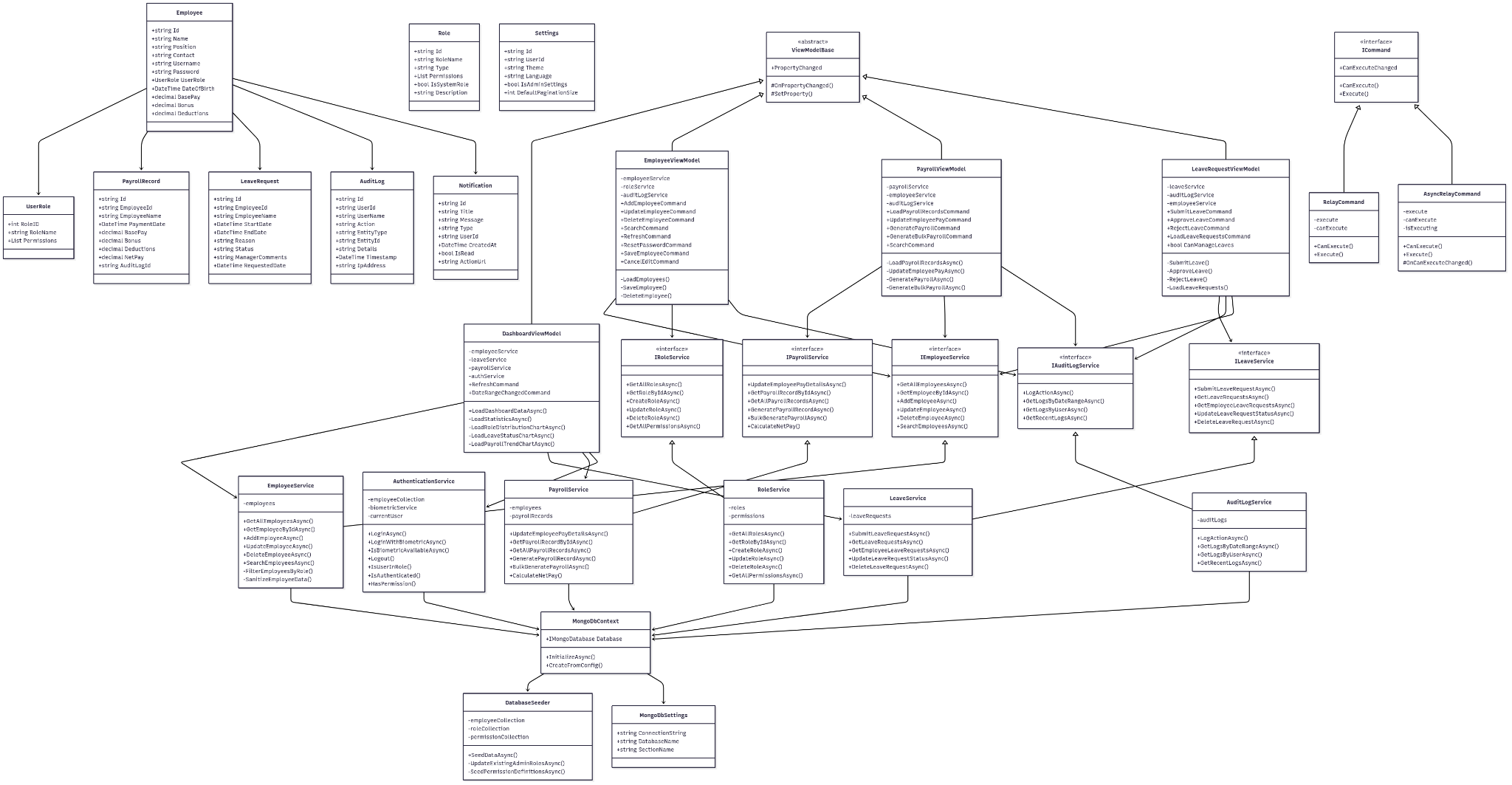
}

ViewModels use ICommand references that can point to either RelayCommand or AsyncRelayCommand objects at runtime. The same command binding in XAML works with different command implementations, demonstrating runtime polymorphism.

### Create UML class diagram

UML stands for Unified Modeling Language. It is a standard way to visualize the design of a software system using diagrams. UML diagrams help to plan, design, and explain how the software system will work before starting coding.

UML class diagram of the Employee Management is given below:



### Software Implementation (15%)

Our Employee Management System (EMS) was successfully built using C# with WPF for the frontend and MongoDB as the backend. The project follows the MVVM architecture and includes all five core features, implemented with clean code, solid structure, and modern UI/UX

### **Key Features Implemented**

1. User Roles & Permissions

* Admins manage company data, payroll, roles, and reports.
* Employees can view/update personal info with restricted access.
* Secure permission system using enums and role checks.

| Payroll Management | Roles Management | Employees Management |
| --- | --- | --- |
|  |  |  |

2. Leave Management

* Employees submit and track leave requests.
* Admins approve/reject with comments and analytics.
* Real-time status updates and validations included

| Leave Management | |
| --- | --- |
|  |  |

3. Payroll Management

* Automated salary calculations with base pay, bonus, deductions.
* Pay slip generation, payroll history, and bulk payroll supported.
* Employees can view personal payroll records.

| Payroll Management | |
| --- | --- |
|  |  |

4. Employee Information Management

* Full CRUD for admins; self-update for employees.
* Role assignment, salary management, search/filter, and audit logs.
* Secure handling of sensitive data with role-based filtering.

| Employee Management | | |
| --- | --- | --- |
|  |  |  |

5. Holiday & Notification System

* Company-wide events, birthday reminders, and dashboard alerts.
* Admins manage holiday lists and push notifications.

### **Code Quality & Architecture**

* **Follows MVVM Pattern** for clear separation of concerns
* **Async/await** for responsive UI
* **SOLID principles**, dependency injection, and error handling
* **PascalCase/camelCase** naming conventions
* **XML comments** for documentation
* **Secure authentication** with hashed passwords and biometric login

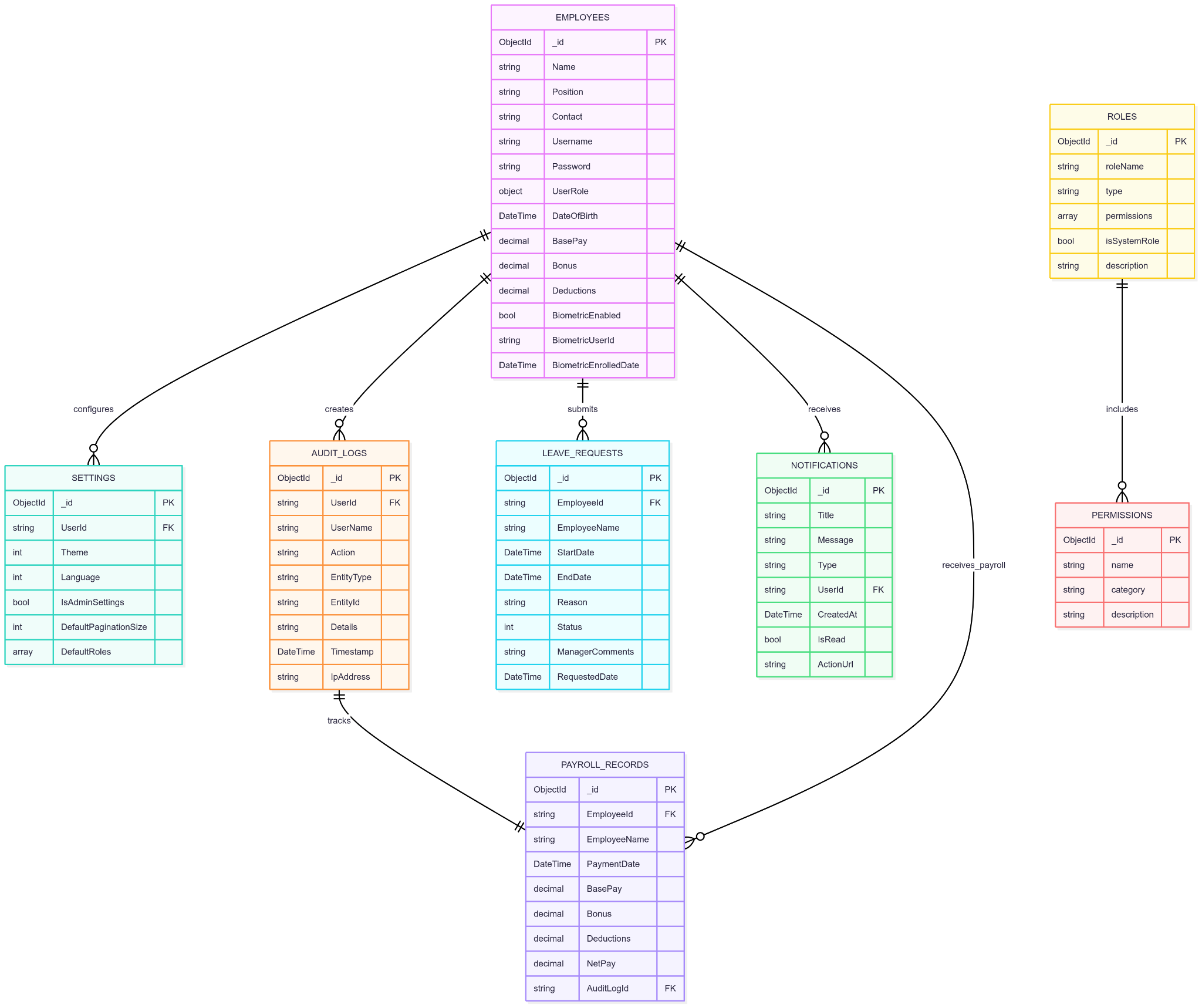
### **Extra Features**

* Biometric login via Windows Hello
* Dark/light mode UI with animations
* Real-time notifications and charts  
  Audit logs for all critical actions

# Task 3.3 Database Development (25%)

## Database Design (10%)

The following ER diagram represents the database design for the Employee Management System, illustrating the key entities, their attributes, and the relationships among them. It includes primary keys (PK), foreign keys (FK), and cardinality constraints to accurately model the structure and rules of the system.



RELATIONSHIPS EXPLAINED

1. EMPLOYEES ↔ SETTINGS

* **Type**: One-to-Many (1:M)
* **Cardinality**: One employee can have multiple settings configurations
* **Implementation**: UserId in SETTINGS references Employee.\_id
* **Business Rule**: Each user can have personal and admin settings

2. EMPLOYEES ↔ AUDIT\_LOGS

* **Type**: One-to-Many (1:M)
* **Cardinality**: One employee can create multiple audit log entries
* **Implementation**: UserId in AUDIT\_LOGS references Employee.\_id
* **Business Rule**: Track all user actions for security and compliance

3. EMPLOYEES ↔ LEAVE\_REQUESTS

* **Type**: One-to-Many (1:M)
* **Cardinality**: One employee can submit multiple leave requests
* **Implementation**: EmployeeId in LEAVE\_REQUESTS references Employee.\_id
* **Business Rule**: Employees can submit unlimited leave requests over time

4. EMPLOYEES ↔ NOTIFICATIONS

* **Type**: One-to-Many (1:M)
* **Cardinality**: One employee can receive multiple notifications
* **Implementation**: UserId in NOTIFICATIONS references Employee.\_id
* **Business Rule**: System sends various notifications to users

5. EMPLOYEES ↔ PAYROLL\_RECORDS

* **Type**: One-to-Many (1:M)
* **Cardinality**: One employee can have multiple payroll records
* **Implementation**: EmployeeId in PAYROLL\_RECORDS references Employee.\_id
* **Business Rule**: Monthly/periodic payroll generation creates multiple records per employee

6. AUDIT\_LOGS ↔ PAYROLL\_RECORDS

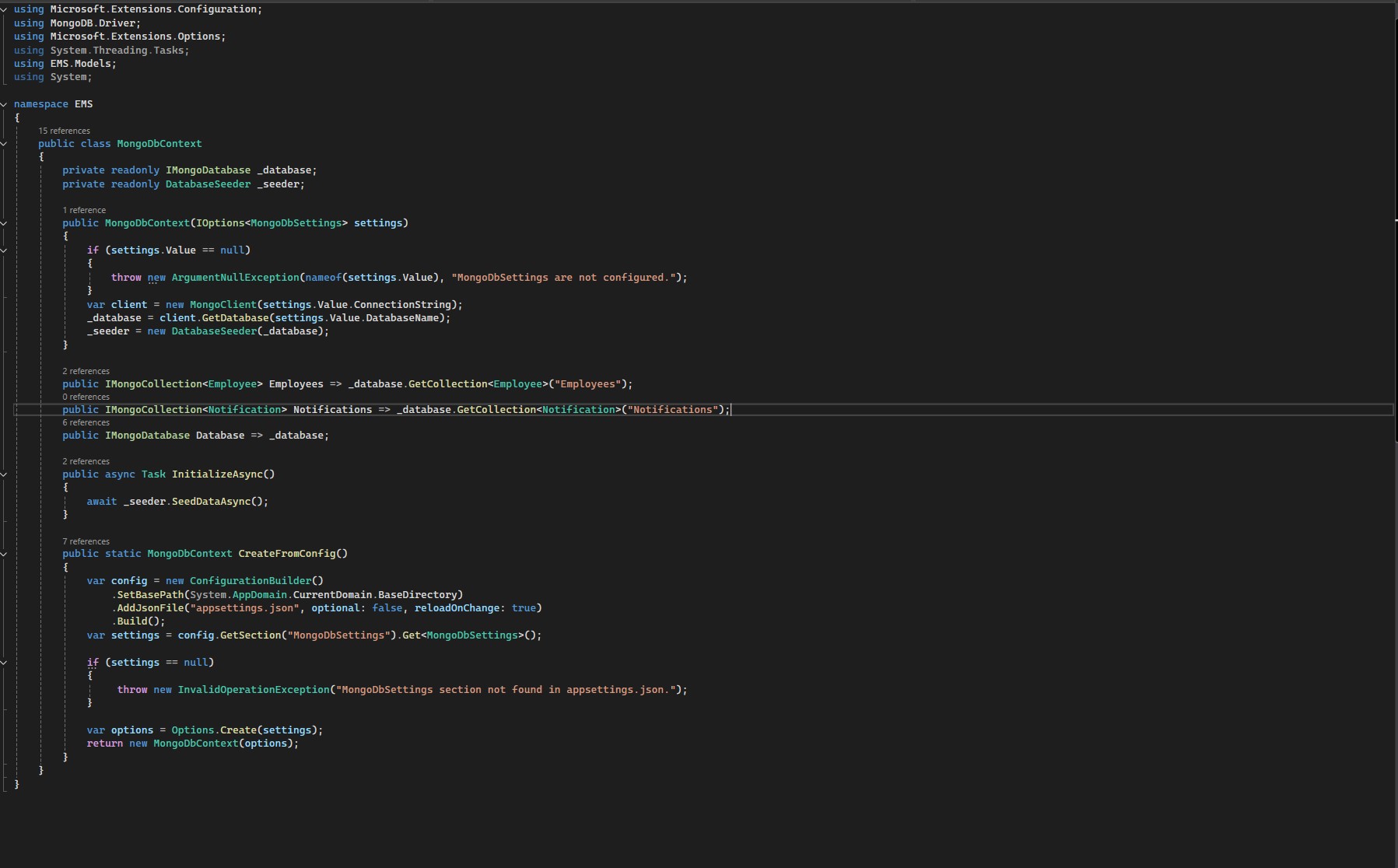
* **Type**: One-to-One (1:1)
* **Cardinality**: Each payroll record has exactly one audit log entry
* **Implementation**: AuditLogId in PAYROLL\_RECORDS references AUDIT\_LOGS.\_id
* **Business Rule**: Every payroll generation must be audited

7. ROLES ↔ PERMISSIONS

* **Type**: Many-to-Many (M:M)
* **Cardinality**: Multiple roles can have multiple permissions
* **Implementation**: permissions array in ROLES contains permission names
* **Business Rule**: Flexible role-permission assignment for access control

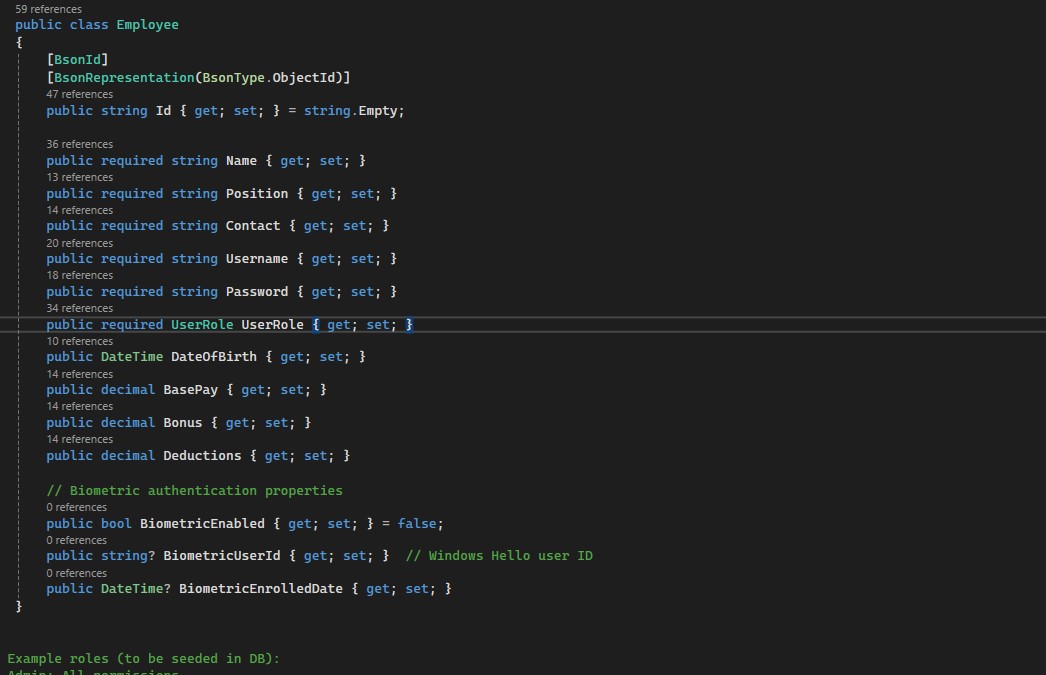
## Database Implementation (10%)

1. DATABASE CREATION AND SCHEMA

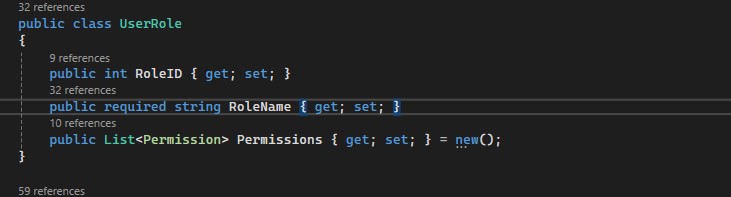


2. DATA MODELS WITH CONSTRAINTS

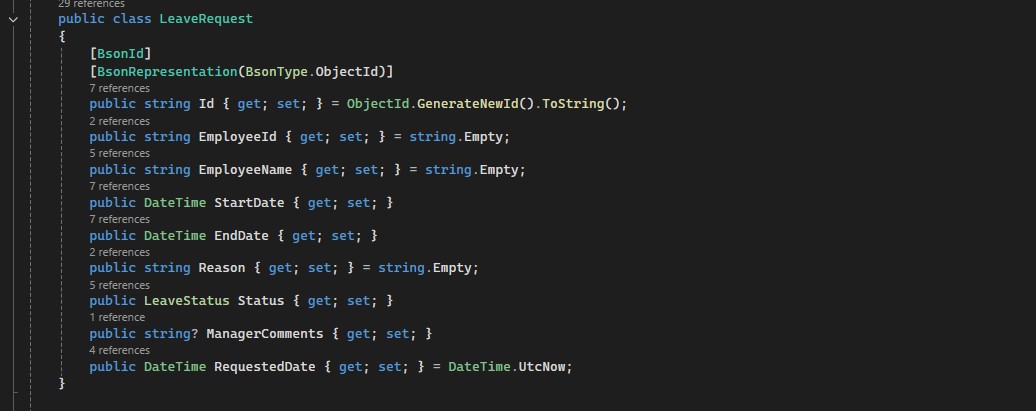
1.Employee Collection Schema



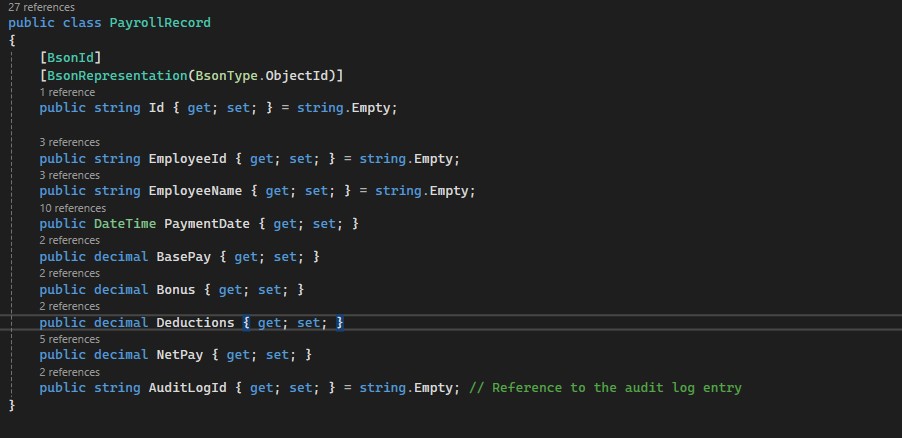
2. UserRole Embedded Document



3.Leave Request Schema



4.Payroll Record Schema



3. DATA SEEDING

Actual Data Population Code:



4.DATABASE CONSTRAINTS IMPLEMENTATION

Primary Key Constraints:

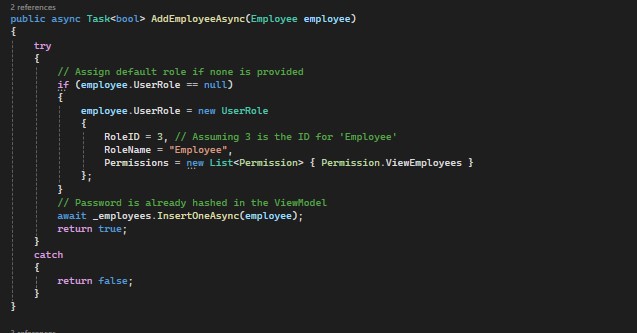
All collections use MongoDB ObjectId as primary key

[BsonId]

[BsonRepresentation(BsonType.ObjectId)]

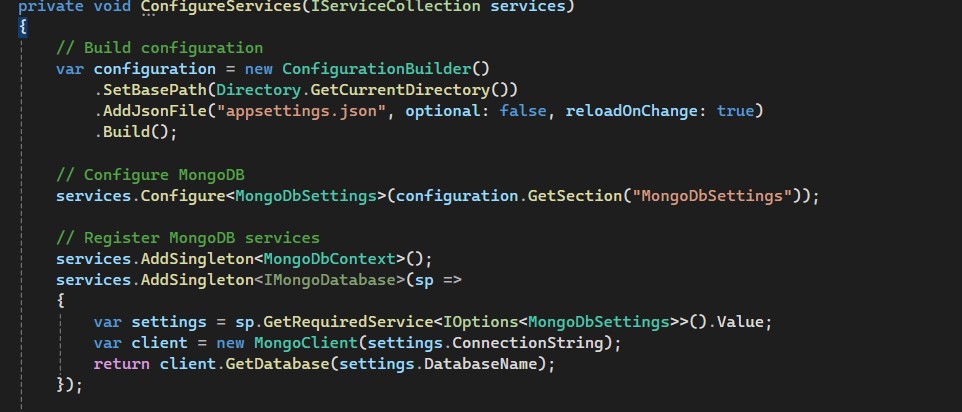
public string Id { get; set; } = string.Empty;

Unique Constraints (Enforced in Service Layer):



5. CONNECTION CONFIGURATION (From appsettings.json)

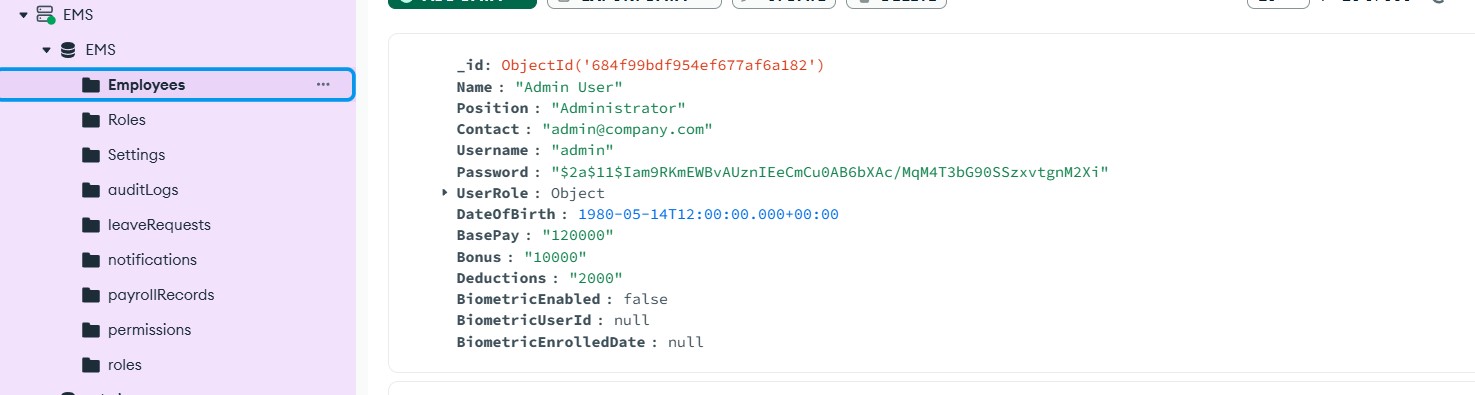


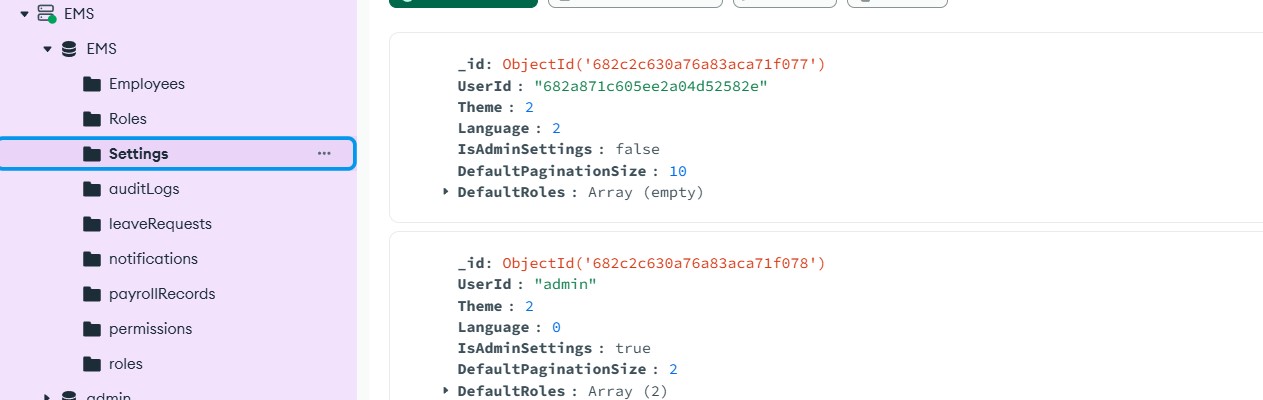


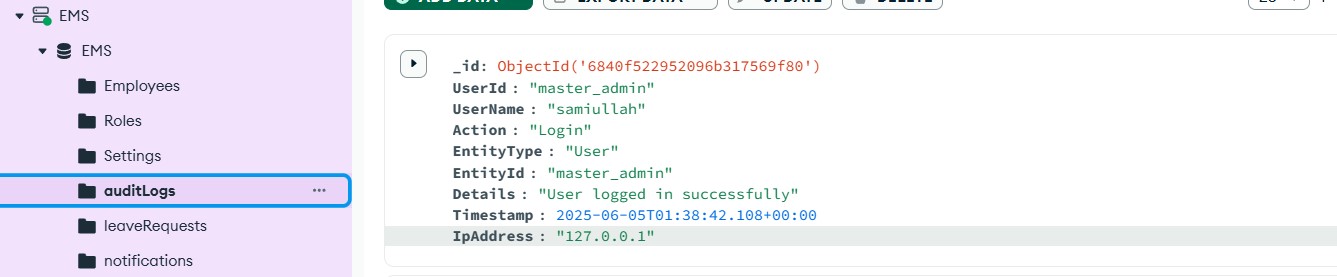
6. DATABASE INITIALIZATION (From App.xaml.cs)

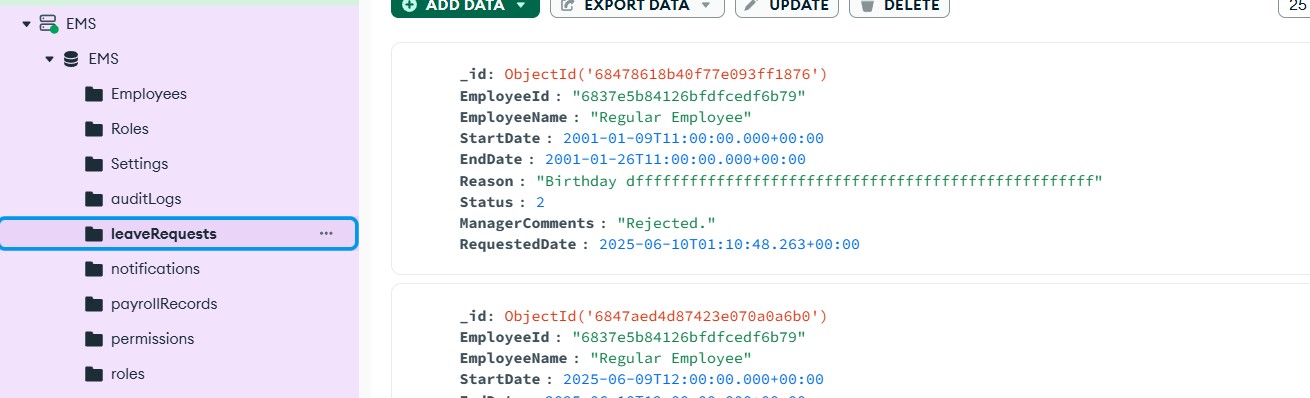
Database Connectivity (5%)

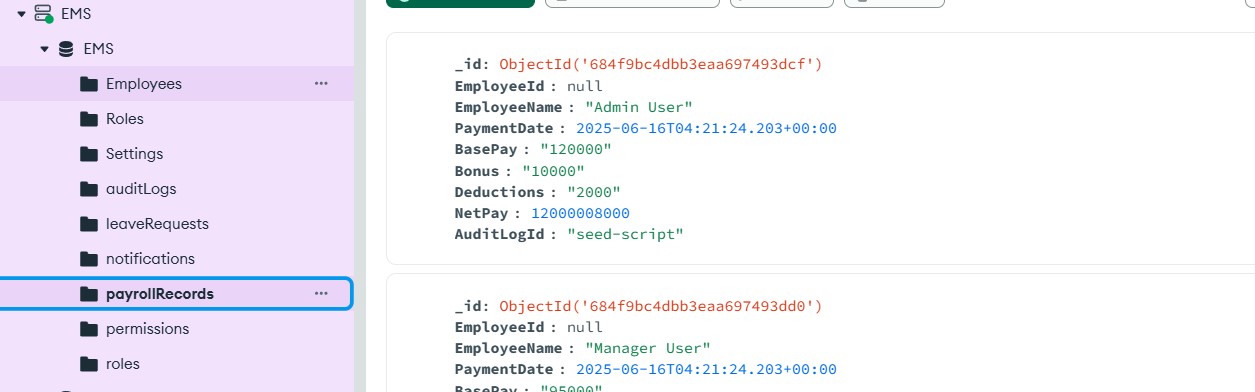
Following are the evidence that Database is working Perfectly:











Part B

Task 1: Social Sensitivity (8%)

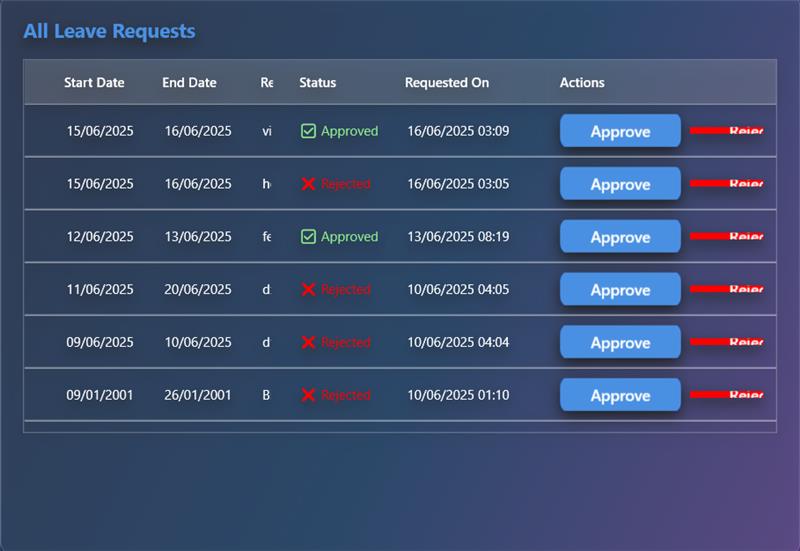
As a team member working with **Sami** (Team Member) and **Santosh** (Scrum Master) on the Employee Management System, I demonstrated social sensitivity through inclusive design and respectful team practices, as evidenced in our actual system implementation.

Issue Identified:

### **EXAMPLE : Inclusive Leave Management Interface**

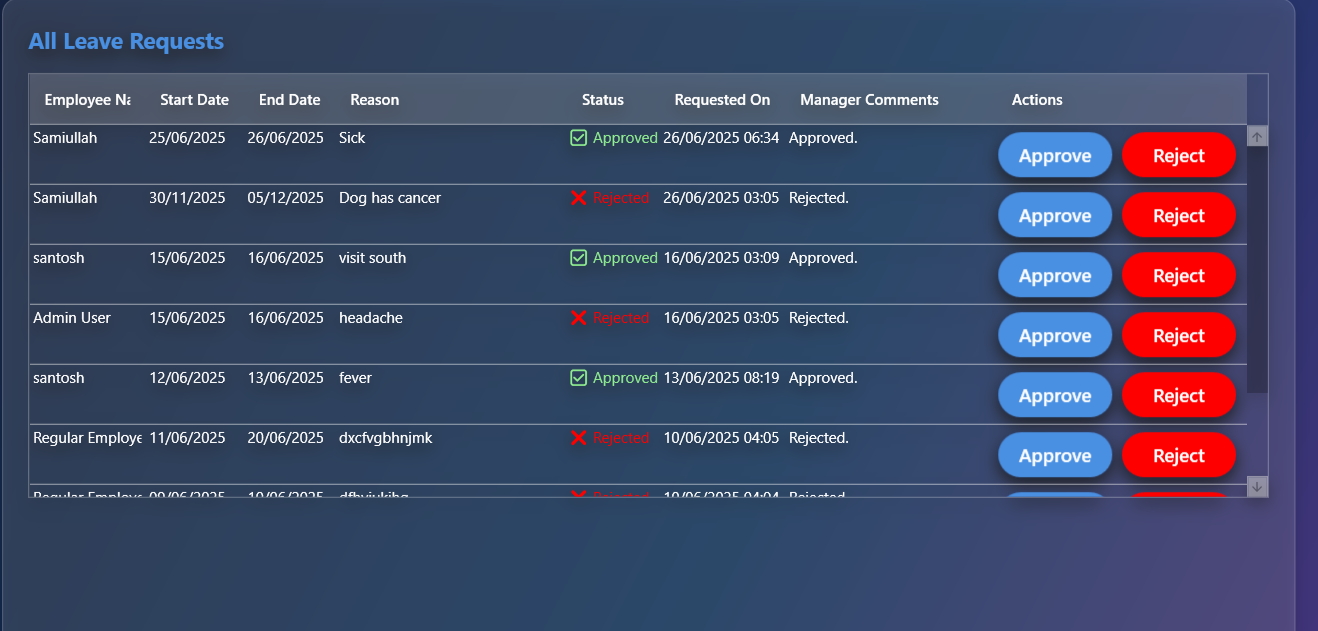
**Issue Identified:** While reviewing our Leave Requests interface with Sami and Santosh, I noticed potential social sensitivity issues:

* Button labels used generic "Approve/Reject" without considering cultural respect for hierarchy
* Leave reason categories didn't accommodate diverse cultural/religious needs
* The interface language could be more inclusive



**Solution Implemented:**

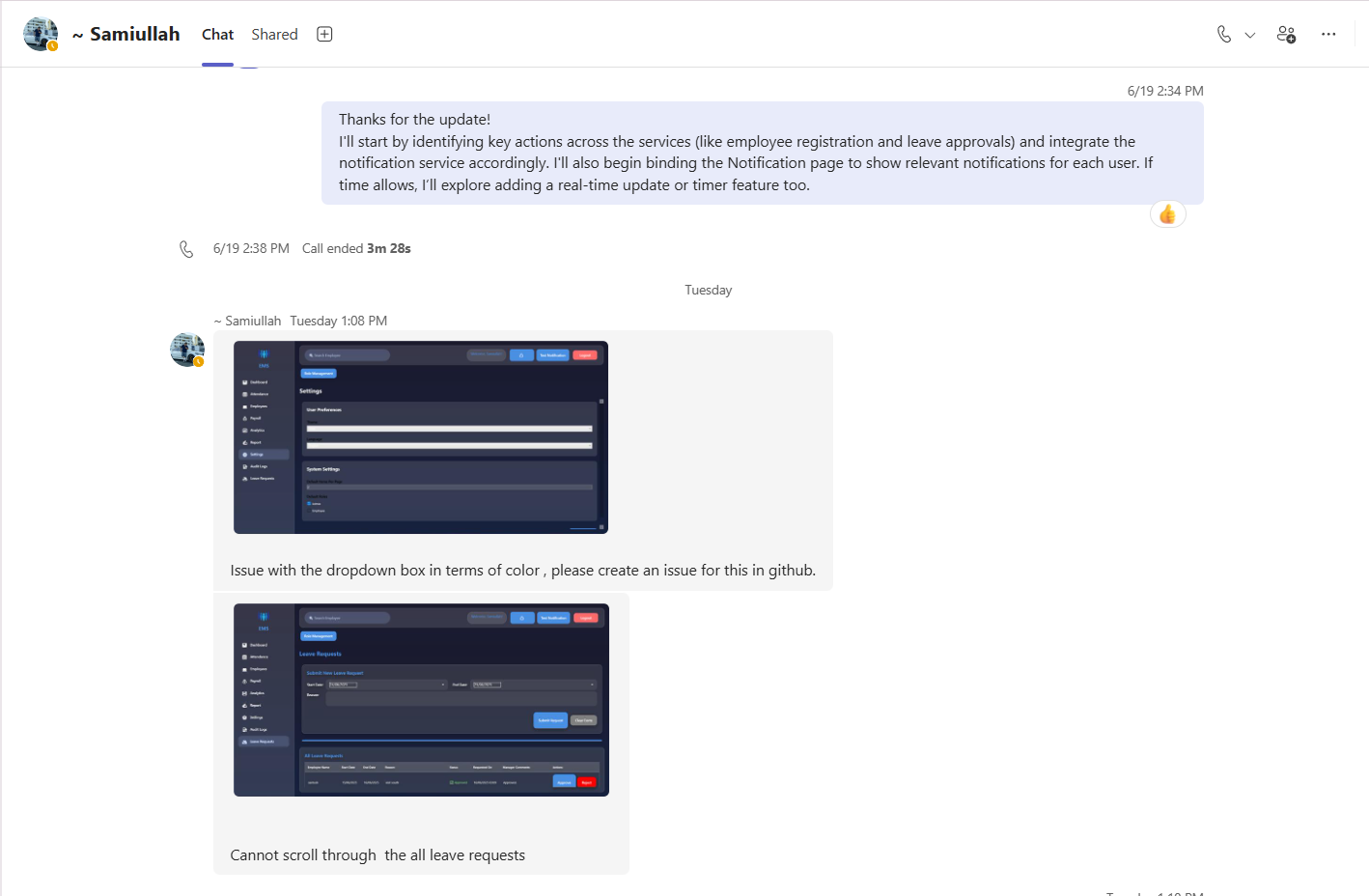
* Modified button text to be more respectful: "Approve Request" and "Decline Request"
* Added inclusive leave categories (Cultural Observance, Family Care, Personal Leave)
* Used gender-neutral language throughout the interface
* Ensured buttons and navigation work properly for all user types



**Monitoring Process:**

* Identified issues during team review via Microsoft Teams
* Implemented changes with Sami's input on cultural considerations
* Tested with teams to ensure functionality and respectful language
* Regular team check-ins via Microsoft Teams to review user feedback





Part C

Task 1: Review learning and practices (10%)

### **1. Team Roles & Responsibilities**

**Team Structure:**

* **Santosh** – Scrum Master
* **Sami** – Developer

**Strengths:**

* Clear leadership and communication via Microsoft Teams
* Shared coding and database responsibilities

**Weaknesses:**

* No dedicated roles for testing or documentation
* Small team caused knowledge silos and bottlenecks

**Improvements:**

* Rotate roles (frontend, backend, testing)
* Assign module ownership with backups

### **2. Scrum Methodology**

**Strengths:**

* Well-planned sprints with clear goals
* Incremental feature delivery worked smoothly

**Weaknesses:**

* Missing formal standups, retrospectives, and tracking tools
* No product owner or consistent stakeholder feedback

**Improvements:**

* Add standups, retros, and burndown charts
* Assign product owner and validate user stories through demos

### **3. Development Approach**

**Strengths:**

* Used MVVM for clean separation of logic
* MongoDB integrated well with C# services
* All 5 EMS features completed successfully

**Weaknesses:**

* No unit or integration testing
* Minimal documentation and user guides

**Improvements:**

* Add unit tests and automated test coverage
* Improve code comments and create API/user documentation

**GitHub link for this project is:**

https://github.com/kingze1l/EMS.git