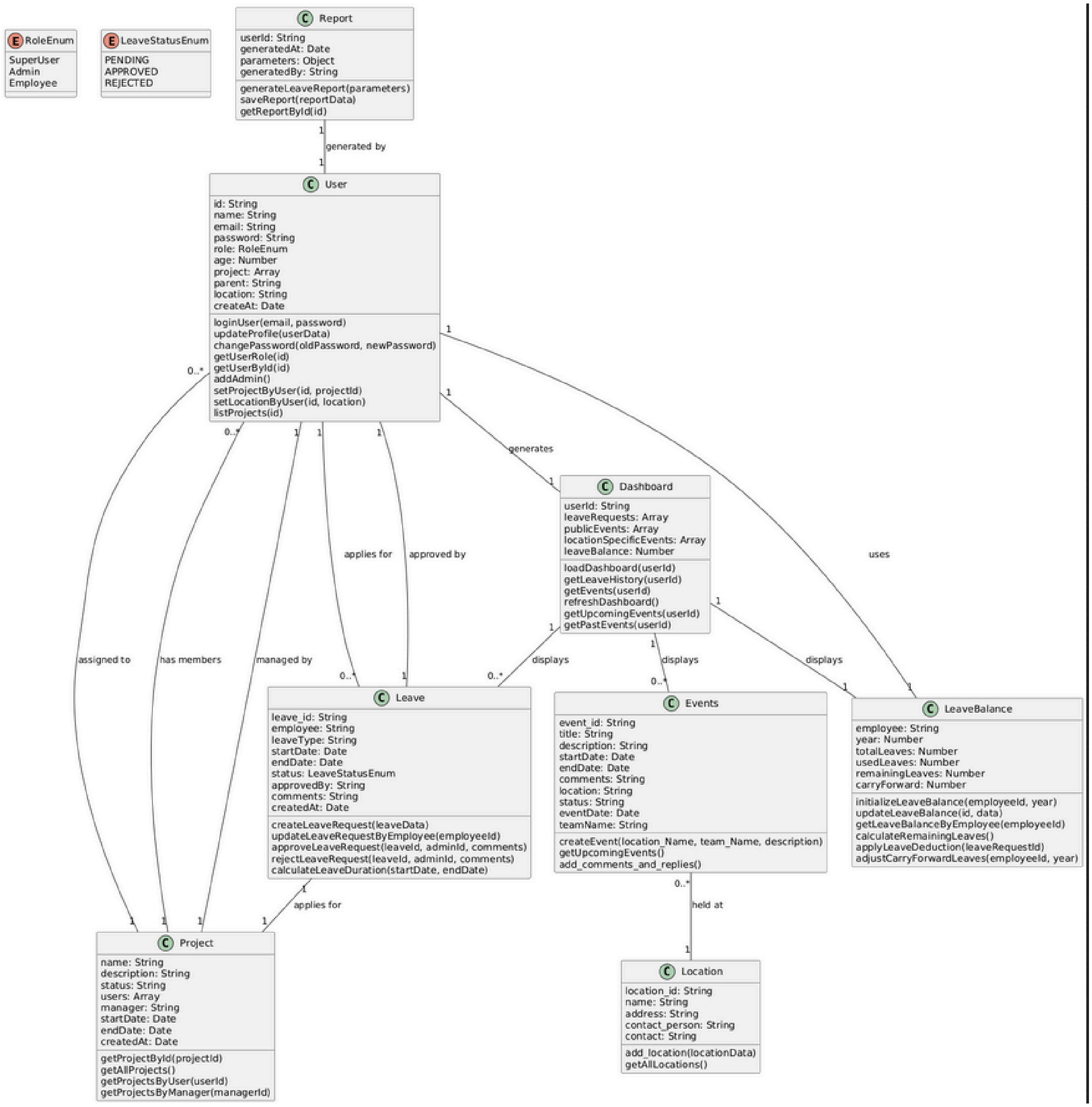
Design Document

Team – 5 – LEAVE MANAGEMENT AND EVENT MANAGEMENT SYSTEM

Team Members:

- **Ananthula Sai Ravichandra**
- **Krishiv Gupta**
- **Aditya Chandramouli**
- **Sai Aditya**
- **Chaitanya Thogata**

Design Model



Class 1: User

Class State	- **id:** string (unique) - **name:** Name of the user (String) - **email:** Email of the User (String) - **password:** Password of the User (String (encrypted))

	- **role:** Role of the User in the Organisation (Enum (SuperAdmin, Admin, User)) - **Age:** Age of the User (Number) - **Projects:** Projects the user is involved in (Array) - **Parent Role:** Heads of user (Array) - **Location:** Location of the User (String) - **createdAt:** User was created in the database at (Date)
Class Behavior	- **loginUser(email, password):** Authenticates a user - **updateProfile(userData):** Updates user profile - **changePassword(oldPassword, newPassword):** Changes user password - **getUserDetails(id):** Retrieves user details - **getUsersByRole(role):** Retrieves users by role - **updateLocation(newLocation):** Updates the user’s location - **isAdmin():** Checks if the user is an Admin. - **getUsersbyLocation(location):** Retrieves all users in a given location. - **listProjects(id):** Retrieves all projects of the user
Purpose	The User class is essential for managing identity, authentication, and access control within an organization. It enforces role-based permissions, tracks project involvement. Secure authentication, location tracking, and account management ensure compliance and operational security, making it an important component of organizational systems.

Class 2: Leave

Class State	- **leave_id:** String (unique identifier) - **employee:** Leave applied for by which employee. String (reference to User) - **leaveType:** Specifies the type of leave(sick,casual,paid). :String - **startDate**: Start date of the leave(Date) - **endDate**: End date of the leave(Date) - **reason:** Reason for taking leave(String) - **status:** Specifies the status of the leave ENUM(PENDING, APPROVED, REJECTED) - **approvedBy:** By whom was it approved by [String (reference to user)] - **comments:** The comments of the user who approves the leave. (String) - **createdAt:** When did the user apply for the leave(Date) - **updatedAt:** When was it updated by the user(Date)
Class Behavior	- **createLeaveRequest(leaveData):** Creates a new leave request - **getLeaveRequestsByEmployee(employeeId):** Retrieves leave requests for a specific employee - **approveLeaveRequest(id, adminId, comments):** Approves a leave request - **rejectLeaveRequest(id, adminId, comments):** Rejects a leave request - **calculateLeaveDuration(startDate, endDate):** Calculates the duration of leave
Purpose	Represents the leave applications filed by the users the system and handles all functions such as creating,accepting/rejecting,listing all the leave requests

Class 3: Leave_Balance

Class State	- **balance_id:** String (unique identifier) - **employee:** String (reference to User) - **year**: Number - **totalLeaves:** Total number of leaves the user is provided with in the given year.**(**Number) - **usedLeaves:** Leaves used by the user**. (**Number) - **remainingLeaves:** Leaves remaining for the user **.(**Number) - **carryForward:** number of leaves that will be carrioed on the next month/yr.(Number) - **updatedAt:** The date of updation **.(**Date)
Class Behavior	- **initializeLeaveBalance(employeeId, year):** Initializes leave balance for an employee - **updateLeaveBalance(id, data):** Updates leave balance - **getLeaveBalanceByEmployee(employeeId):** Retrieves leave balance for a specific employee - **calculateRemainingLeaves():** Calculates remaining leaves - **applyLeaveDeduction(leaveRequestId):** Deducts leaves when a request is approved - **adjustCarryForwardLeaves(employeeId, year):** Adjusts carry-forward leaves for the new year
Purpose	Represents the number of the leaves left in the system for any user and handles all operations such as initializing,updating,calculating the leaves

Class 4: Events

Class State	- **event_Id**: Unique Identifier for event (unique) - **title**: Name of the event (string) - **description**: Detailed information about the event (string) - **startDate:** Date when event begins (Date) - **endDate**: Date when the event ends(Date) - **createdAt**: Date when the event was created(Date) - **comments**: Comments by other users on a particular event (string) - **location**: Location where the event will be held (string) - **status**: Current Status (upcoming, ongoing, completed, cancelled) (Enum) - **event_Date**: Date of the event (date) - **location_Name**: The branch at which the event is being organized(string) - **team_Name**: The teams which are/ would be part of the event
Class Behavior	- **createEvent**(location_Name, team_Name, description): Creates a new event - **getUpcomingEvents**()**:** Retrieves all upcoming events - **add_comments_and_replies**(): Adds a comment / reply (to existing comment)
Purpose	Represents events in the system and manages all event-related operations including creation, updates and cancellation of events (due to unforeseen reasons)

Class 5: Location

Class State	- **location_Id**: Unique Identifier for the location (unique) -

	name: Name of the location (string) - **address**: address where the center is situated (string) - **contact_person**: Name of the contact person (string) - **contact**: Phone number of the location (string)
Class Behavior	- **add_location**(locationData): Creates a new location - **getAllLocations**(): Retrieves all locations
Purpose	Manages organization locations that can be used for events, storing their details and providing functionality to check availability and manage location information.

Class 6: Dashboard

Class State	- **userId:** String - **leaveRequests:** All the leave requests requested by the User (Array) - **Public Events:** All the Events that open for entire Organization (Array) - **Location specific Events:** All those events which are open for certain location people (Array) - **leaveBalance:** Number of leaves left for the User (Number)
Class Behavior	- **loadDashboard(userId):** Loads the dashboard for a user - **getLeaveHistory(uesrId):** Retrieves the leave history for a user - **getEvents(userId):** Retrieves all the events open for the user - **refreshDashboard()**: Refreshes the dashboard data - **getUpcomingEvents(userId):** Retrieves all the upcoming events for a user - **getPastEvents(userId):** Retrieves all the past events that were open for the user

Purpose	The Dashboard class provides users with a simple and organized way to track their leave history, remaining leave balance, and upcoming events. It displays both company-wide and location-specific events, ensuring users never miss important updates. With real-time data updates, users can easily stay informed about their requests, past activities, and upcoming engagements, making workplace management smoother and more efficient.

Class 7: Report

Class State	- **userId:** String - **generatedAt:** Time of report generation (Date) - **parameters:** parameters of the report (Object of report parameters) - **generatedBy:** References to User who is generating it (String)
Class Behavior	- **generateLeaveReport(parameters):** Generates a leave report - **saveReport(reportData):** Saves a report - **getReportById(id):** Retrieves a report by ID
Purpose	The Report class helps users generate and access reports on leaves. It tracks when a report was created, who generated it, and what details it includes. Users can save reports for future reference and retrieve them as needed, making it easier to analyze leave records or event participation.

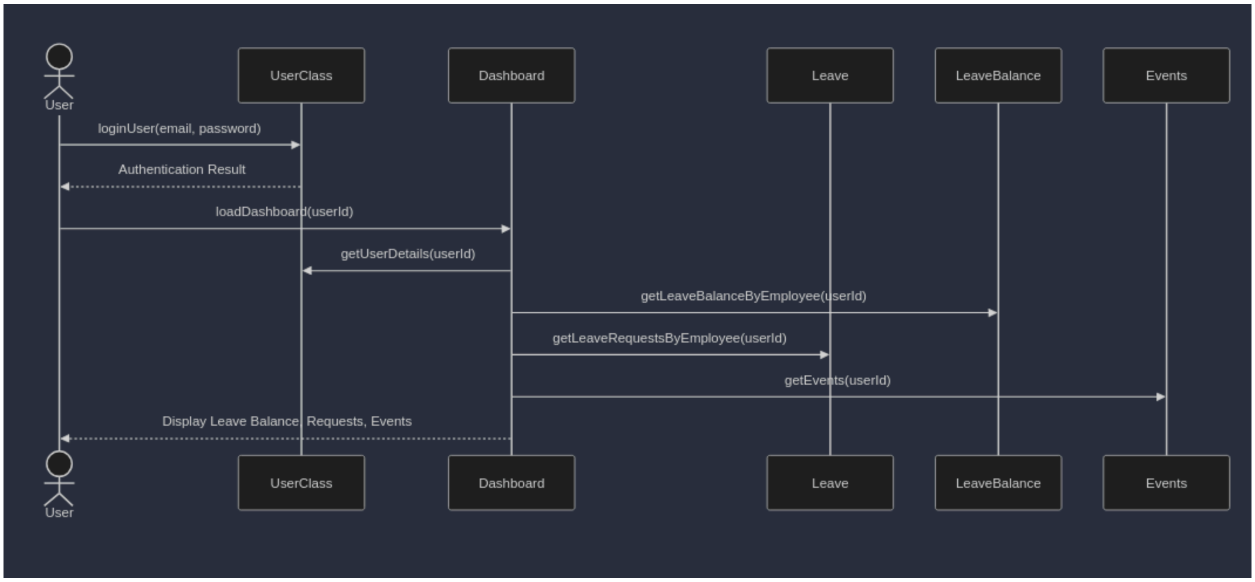
Class 8: Project

Class State	**- name (String)** – The name of the project. - **description (String)** – A brief summary of the project. - **status (String)** – The current status of the project (e.g., Active, Completed, On Hold). - **users (Array)** – List of users working on the project. -

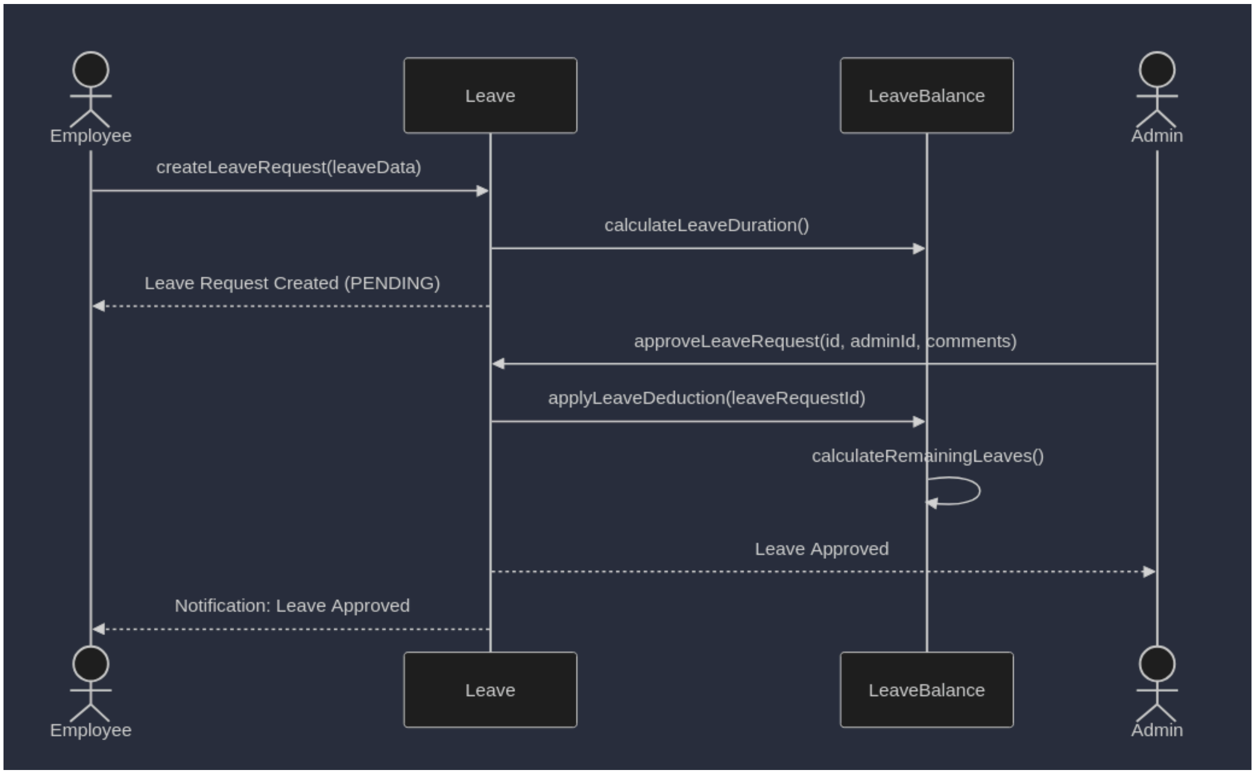
	manager (Array) – List of managers overseeing the project. - **startDate (Date)** – The date when the project starts. - **endDate (Date)** – The expected completion date. - **createdAt (Date)** – The date the project was created in the system.
Class Behavior	- **getProjectById(projectId)** – Retrieves project details by ID. - **getAllProjects()** – Retrieves all projects in the system. - **getProjectsByUser(userId)** – Retrieves all projects a user is part of. - **getProjectsByManager(managerId)** – Retrieves all projects managed by a specific manager.
Purpose	The Project class is responsible for viewing projects within an organization. It keeps track of project details, including its name, description, status, timeline, and team members. It helps in retrieving project-related data. This class ensures clear organization, easy tracking, and efficient team collaboration.

Sequence Diagrams

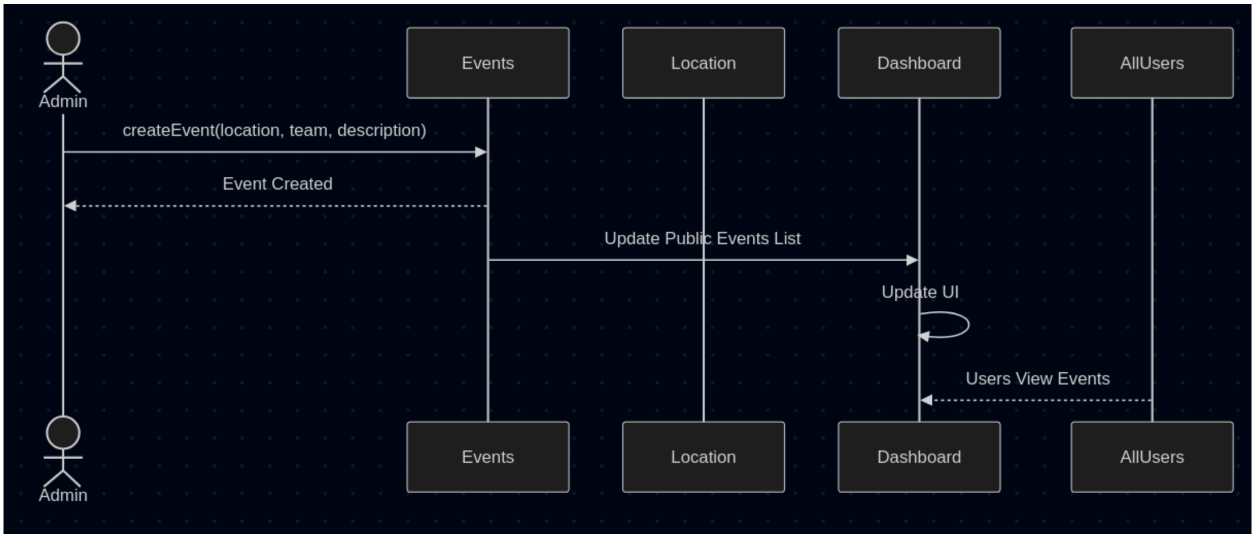
1. User Login & Dashboard Loading



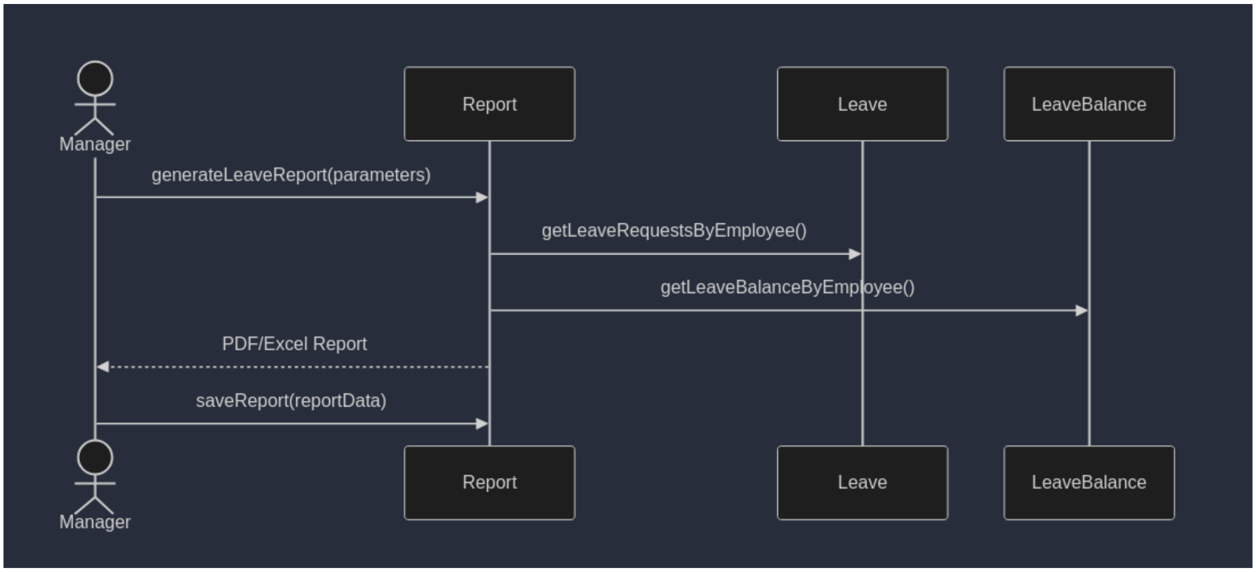
2. Leave Request Lifecycle


Employee
Leave
LeaveBalance
Admin
createLeaveRequest(leaveData)
calculateLeaveDuration()
Leave Request Created (PENDING)
approveLeaveRequest(id, adminId, comments)
applyLeaveDeduction(leaveRequestId)
calculateRemainingLeaves()
Leave Approved
Notification: Leave Approved
Employee
Leave
LeaveBalance
Admin

3. Event Management Flow



4. Report Generation



Design Rationale

Rejected Proposals

1. WordPress CMS Implementation

Alternative Considered:

Initially, the client suggested using WordPress as the CMS for dashboard development, as indicated in our early discussions.

Pros:

- Familiar interface for content management
- Quick setup with pre-built themes and plugins
- Lower initial development complexity
- Extensive documentation and community support

Cons:

- Limited customization for specialized features like leave management calculations
- Challenges in implementing complex hierarchical structures
- Performance issues with extensive customizations
- Security concerns with plugins and updates
- Difficulty implementing custom user roles and permissions

Final Decision:

After thorough analysis and discussions with the client, we decided against using WordPress in favor of a custom MERN stack implementation. While WordPress would have offered faster initial development, the custom nature of Eklavya's requirements—particularly the dynamic organizational hierarchy, specialized leave management system, and complex information sharing rules—would have required extensive customization that might ultimately compromise WordPress's performance and maintainability.

2. User Self-Registration System

Alternative Considered:

We initially considered implementing a self-registration system where employees could create their own accounts on the dashboard.

Pros:

- Reduced administrative overhead
- Faster onboarding process for new employees

Cons:

- Potential security risks with unverified users
- Difficulty in maintaining organizational hierarchy integrity

Final Decision:

As discussed in our February 19th meeting, we opted for an admin-controlled user management system instead. The admin will manually add users to the system with appropriate roles, ensuring proper control over who has access to the system and what permissions they have. This approach aligns better with Eklavya's organizational structure and security requirements.

3. MySQL Database Implementation

Alternative Considered:

We initially evaluated using MySQL as our database solution for its reliability and structured approach.

Pros:

- Established relational database with proven reliability
- Strong data integrity through relationships and constraints
- Familiarity among team members

Cons:

- Less flexibility for evolving schemas and dynamic data structures
- Complexity in handling hierarchical data structures
- Potential scalability challenges for document-heavy applications

Final Decision:

As discussed in our February meetings, we opted for MongoDB over MySQL. MongoDB's document-oriented structure allows for greater flexibility in handling the dynamic hierarchical data required for Eklavya's organizational structure. The non rigid nature of MongoDB makes it easier to adapt to evolving requirements without extensive database migrations. Additionally, MongoDB works seamlessly with the rest of our MERN stack, providing better integration and development efficiency.

4. Rigid Location Hierarchy Structure

Alternative Considered:

We initially considered implementing a fixed hierarchical structure for locations and departments.

Pros:

- Simplified implementation
- Clearer navigation structure
- Less complex database schema

Cons:

- Lack of adaptability to organizational changes
- Difficulty in accommodating new locations or restructuring
- Potential redesign requirements for future changes

Final Decision:

Based on discussions from our February 12th meeting, we decided to implement a dynamic, flexible hierarchy system. This allows the admin to add new locations as needed and designate parent locations, accommodating the growing and changing nature of Eklavya Foundation's structure. The HQ in Bhopal remains the top level, with three main independent locations under it, and the system allows for expansion without requiring code changes.

5. Infinite Depth of Comments and Replies for Events

Alternative Considered:

We proposed implementing a flexible, **infinite-depth commenting system** for events, where replies could have nested replies indefinitely—similar to Reddit or GitHub discussions.

Pros:

- Facilitates rich, threaded conversations
- Allows for more organized and contextual discussions
- Provides better clarity in collaborative discussions or feedback

Cons:

- Increased implementation complexity, especially in frontend rendering and backend data modeling
- Potential performance issues with deeply nested comments
- Complicates UI/UX for users not familiar with such structures
- Higher cognitive load for users reading or tracking deeply nested threads

Final Decision:

After receiving feedback from the client, we decided to simplify the system by **restricting nesting to one level**. Each event can have multiple **top-level comments**, and each comment can have multiple **replies—**but replies **cannot have replies of their own**. This design ensures a cleaner interface, simpler backend data structure, and better usability for the end-users, aligning with the client's preference for simplicity and ease of use.

Accepted Proposals

1. MERN Stack Implementation over WordPress

Decision Context:

During our January 25th meeting, we evaluated the possibility of developing the project using the MERN stack instead of WordPress CMS as initially suggested by the client.

Benefits:

- Greater flexibility for custom features like leave management
- Better scalability for dynamic organizational structures
- Complete control over the codebase for customization
- More modern user interface capabilities

2. Admin-Controlled User Management over Self-Registration

Decision Context:

During our February 19th meeting, we evaluated different user onboarding approaches, including self-registration and admin-controlled user management. While self-registration would have streamlined the onboarding process, it raised security and organizational integrity concerns.

Benefits:

- Ensures only verified employees gain access to the system, preventing unauthorized users.
- Avoids issues where users incorrectly assign themselves to departments or roles.
- Reduces the risk of fraudulent or unintended access.
- Admins can assign the correct roles upon user creation, preventing permission misuse.
- Allows the admin to add users in a structured manner, ensuring consistency in user data.

3. MongoDB over MySQL

Decision Context:

After evaluating both options and discussing database requirements in our February meetings, we chose MongoDB over MySQL.

Benefits:

- Flexible schema design that can evolve with organizational changes
- Better handling of hierarchical and document-based data
- Easier integration with the Node.js/Express back-end

4. Hierarchical Information Sharing System

Decision Context:

Based on discussions from multiple meetings, particularly on March 9th, we established a sophisticated approach to information sharing based on location and program hierarchies.

Benefits:

- Targeted communication based on relevant locations and programs
- Ability to share events across all relevant locations when needed
- Control over information visibility based on user roles and locations

5. Two-Level Comment-Reply System for Events

Decision Context:

Following our discussions with the client, we accepted the proposal to implement a **two-level** commenting structure for event discussions.

Benefits:

- Keeps the UI simple and user-friendly
- Prevents clutter and complexity in both front-end rendering and back-end data modeling
- Sufficient for most event-based discussions without over engineering
- Easier for users to follow and participate in conversations

- Improves maintainability and performance of the application

This decision ensured that **each event can have multiple comments**, and **each comment can have multiple replies**, but **replies cannot have further replies**, maintaining clarity while supporting essential interactions.