Software Design Specifications

for

AN INTERACTIVE WEBSITE FOR MU (INTRANET)

Version: 1.0

Prepared by: Ideal Bits

Sai Sarvani.P (SE22UARI127)

Vivek Reddy.P (SE22UARI122)

Varshika.P (SE22UARI124)

Vignesh Anoop Naidu.N (SE22UARI108)

Shivani.N (SE22UARI109)

Document Information

Title: An Interactive Website for MU
Instructor Name: Prof.Avinash Arun Chauhan
Document Version Date: April 08,2025
Preparation Date: April 07,2025
Document Version No: 1.0
Prepared By: Sarvani, Varshika, Vivek, Shivani, Vignesh

Version History

Ver.No.	Ver.Date	Revised By	Description
1.0	April 08,2025	Sarvani, Varshika, Vivek, Shivani, Vignesh	Initial Release

Table of Contents

1. Introduction

This Software Design Specification (SDS) outlines the design of the *Intranet Portal for Mahindra University, developed* as a course project. The portal is a centralized platform that allows students and faculty to manage academic and administrative tasks.
The document explains how the system is structured, how its parts work together, and how it meets the requirements from the SRS. It includes design diagrams, data models, and interface details to guide development and testing.

1.1 Purpose

The purpose of this document is to describe how the intranet portal will be built. It helps:

- **Developers:** Understand the structure and flow of the system.
- **Testers:** Prepare test plans based on the design.
- **Evaluators:** Review the design approach.

It connects the requirements with actual development and serves as a reference throughout the project.

1.2 Scope

This Software Design Specification applies to the design and development of the Intranet Portal for Mahindra University. It defines the structure and behaviour of key modules such as:

- Login
- Meeting scheduling
- Course material access
- Results management
- Library check
- Fee payment
- Elective selection
- Event calendar

The document influences how the system is implemented, tested, and maintained, ensuring all components work together to meet the needs of students, faculty, and admin users through a role-based interface.

1.3 Scope Definitions, Acronyms, and Abbreviations

AI - Artificial Intelligence
API - Application Programming Interface
CRUD - Create, Read, Update, Delete
CSS - Cascading Style Sheets

DB - Database
HTML - HyperText Markup Language
HTTP - Hypertext Transfer Protocol
JWT - JSON Web Token
JS - JavaScript
MU - Mahindra University
REST API - Representational State Transfer API
SDD - Software Design Document
UI - User Interface

1.4 References

[1] Software Requirements Specification (SRS)
Primary requirements document outlining the system's functional and non-functional needs.
[2] MongoDB Official Documentation
https://www.mongodb.com/docs – For all DB schema-related configurations.
[3] JWT Guide
https://jwt.io/introduction – For token-based authentication mechanism.
[4] Node.js Documentation
https://nodejs.org/en/docs – Backend runtime environment documentation.
[5] Express.js Documentation
https://expressjs.com – Web application framework for Node.js.
[6] Mongoose Documentation
https://mongoosejs.com/docs – MongoDB object modeling for Node.js.
[7] Bootstrap Documentation
https://getbootstrap.com/docs – Frontend CSS framework.
[8] Mahindra University Brand Guidelines Internal resource for aligning portal UI with official branding. (Available via internal university portal)
More technical and domain-specific references may be added in Appendix A if required.

2. Use Case View

This section presents the major use cases derived from the system requirements. Each use case represents a key function of the Intranet Portal and involves interactions between users and the system. The use cases here cover the core functionalities of the portal and involve multiple system components, making them central to the software design.
These use cases are crucial for understanding how different roles—students and faculty—interact with the system through their dashboards.

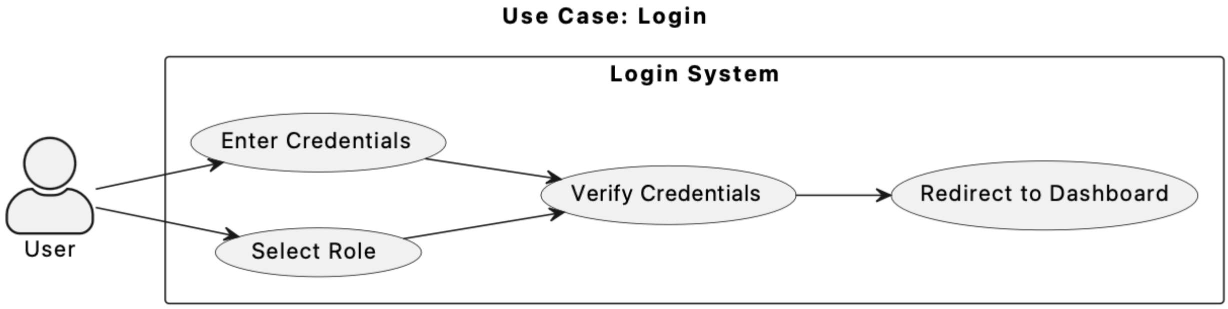
2.1 Use Case

Use Case: Login

Description:
Allows users to log into the portal by verifying their credentials and selected role.

Usage Steps:

1. User enters username and password.
2. Selects their role (Student / Faculty).
3. System verifies credentials and role.
4. Redirects to the respective dashboard.

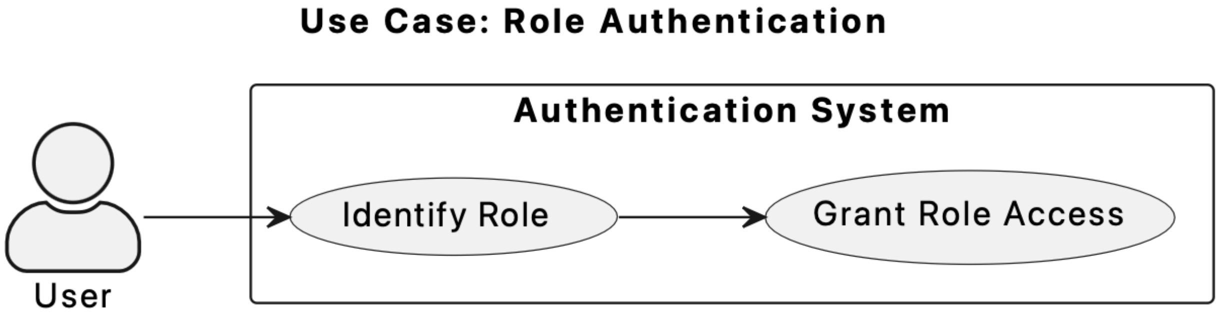


Use Case: Role Authentication

Description:
Determines and verifies the role of the user during login to load the appropriate dashboard and permissions.

Usage Steps:

1. After login, system identifies the selected role.
2. Grants access to role-specific features and views.

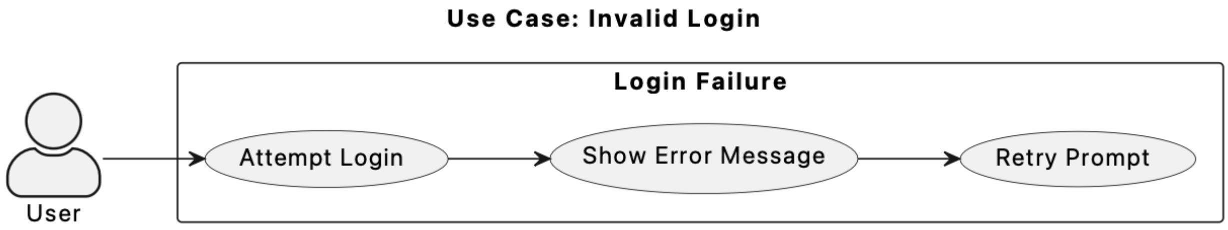


Use Case: Invalid Login

Description:
Handles failed login attempts due to incorrect credentials or role mismatch.

Usage Steps:

1. User attempts login with incorrect credentials.
2. System displays an error message.
3. Prompts user to try again.

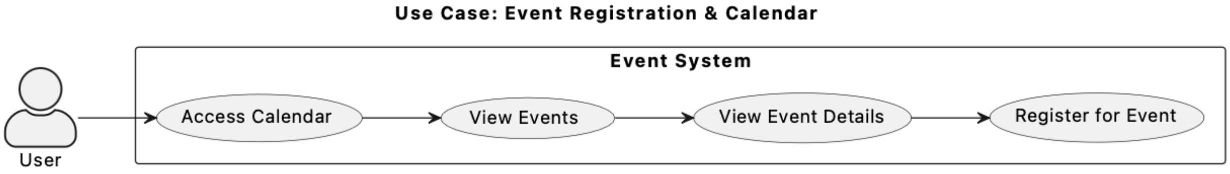


Use Case: Event Registration & Calendar

Description:
Displays university calendar with upcoming events. Allows users to view details and register.

Usage Steps:

1. User accesses calendar module.
2. Views upcoming events.
3. Clicks on an event to see details.
4. Registers if needed.

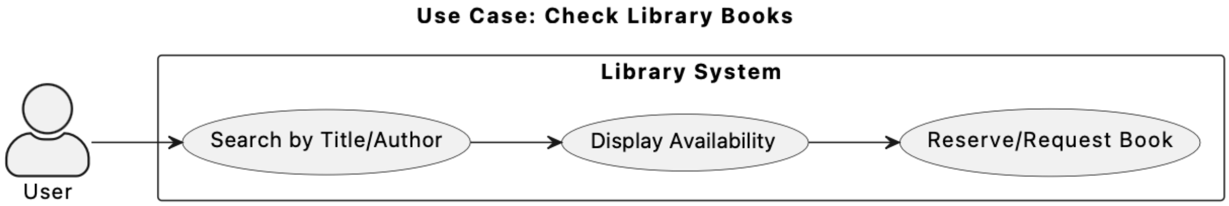


Use Case: Check Library Books

Description:
Allows users to search for books in the library and check their availability.

Usage Steps:

1. User enters a book title or author.
2. System searches the library database.
3. Displays availability status.
4. User can request or reserve book if available.

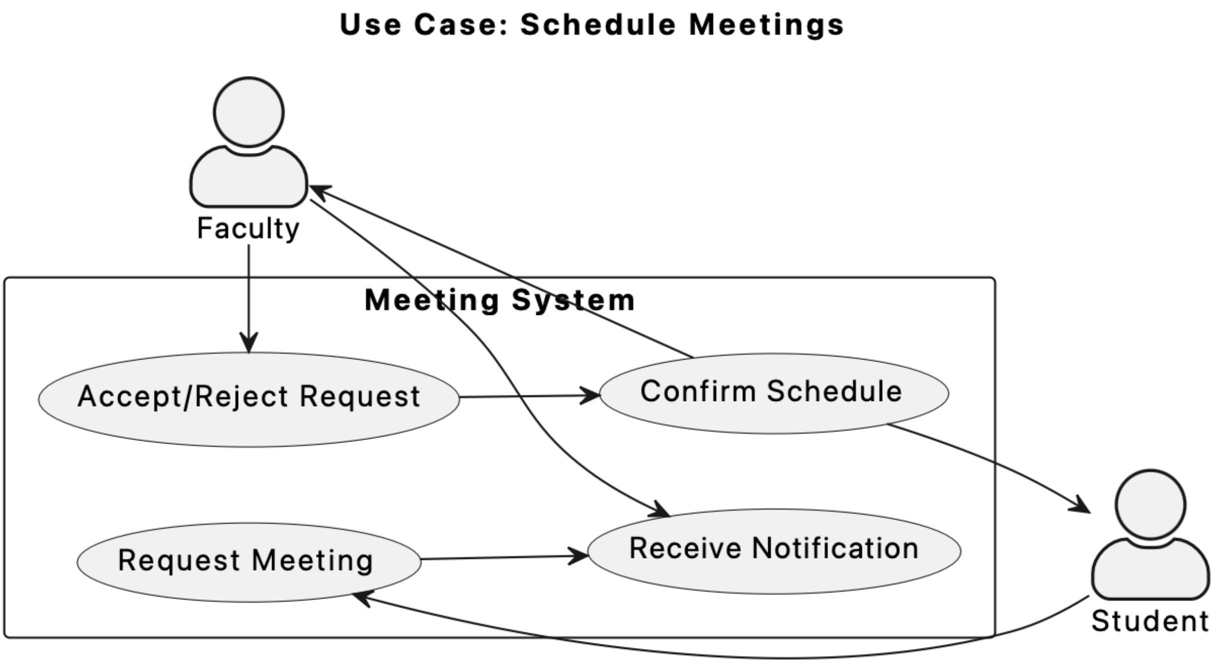


Use Case: Schedule Meetings

Description:
Students can request meetings with faculty. Faculty can accept or reject requests.

Usage Steps:

1. Student selects faculty and preferred time.
2. Sends meeting request.
3. Faculty receives notification.
4. Faculty accepts or declines the request.
5. Confirmation is shown on both dashboards.



Use Case: Access Course Content

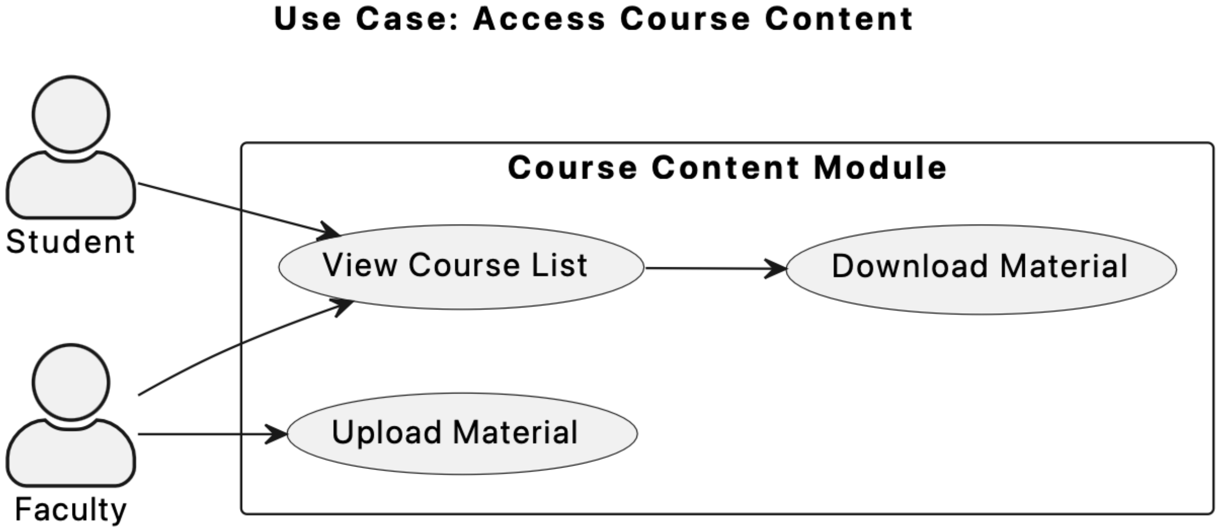
Description:
Enables students to view/download course materials and faculty to upload them.

Usage Steps (Student):

1. Select a course.
2. View list of materials (notes, slides, assignments).
3. Download as needed.

Usage Steps (Faculty):

1. Select a course.
2. Upload relevant files.
3. Materials become accessible to enrolled students.

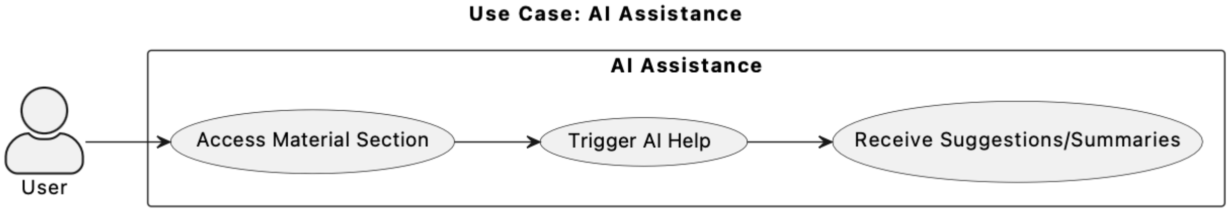


Use Case: AI Assistance for course content

Description:
Provides smart suggestions, summaries, or content help for both students and faculty.

Usage Steps:

1. User accesses a course material section.
2. Activates AI Assistance.
3. Receives suggestions, summaries, or improvements.



Use Case: Results

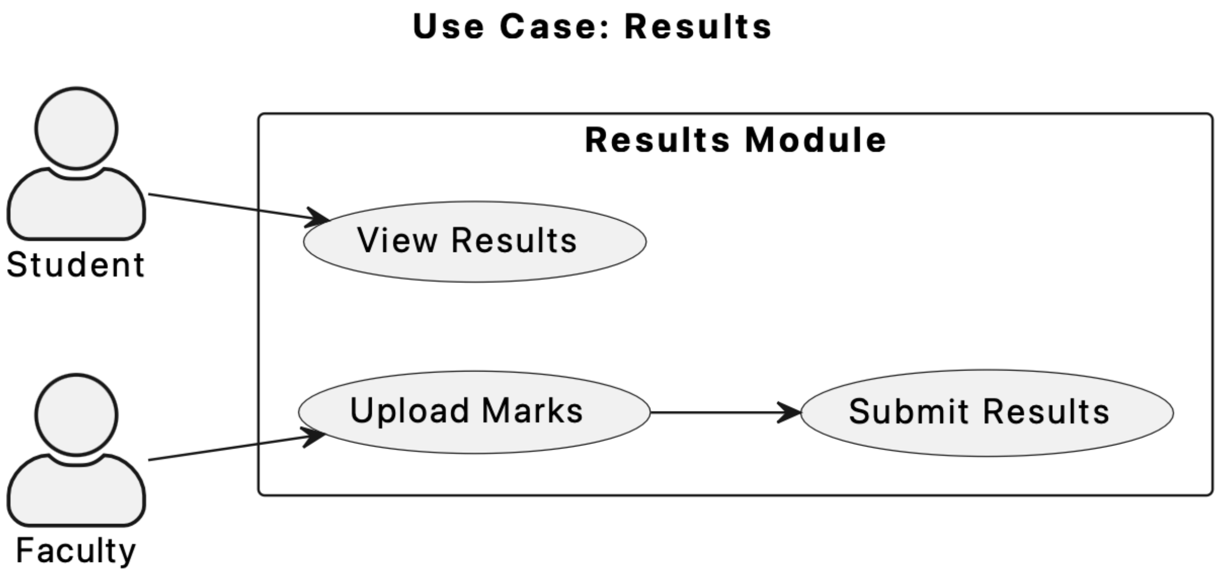
Description:
Faculty can upload results and students can view their scores.

Usage Steps (Faculty):

1. Select subject and batch.
2. Enter/upload student marks.
3. Submit to system.

Usage Steps (Student):

1. Navigate to results section.
2. View subject-wise marks.

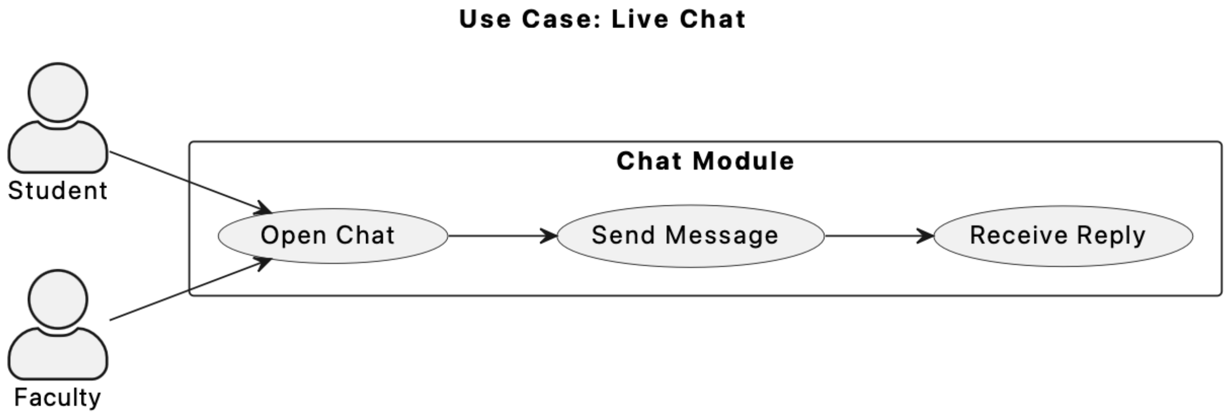


Use Case: Live Chat

Description:
Real-time chat module for students and faculty to communicate.

Usage Steps:

1. User opens chat section.
2. Selects contact (student/faculty).
3. Types and sends message.
4. Receives replies in real-time.

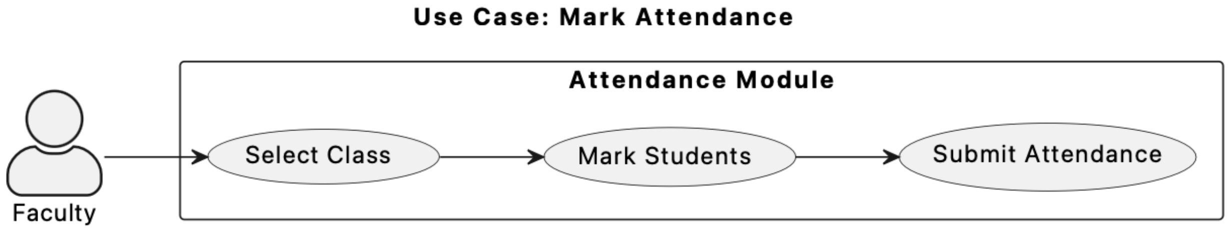


Use Case: Mark Attendance

Description:
Faculty can mark daily attendance for students in their classes.

Usage Steps:

1. Faculty selects class and date.
2. Views student list.
3. Marks present/absent for each student.
4. Submits record.



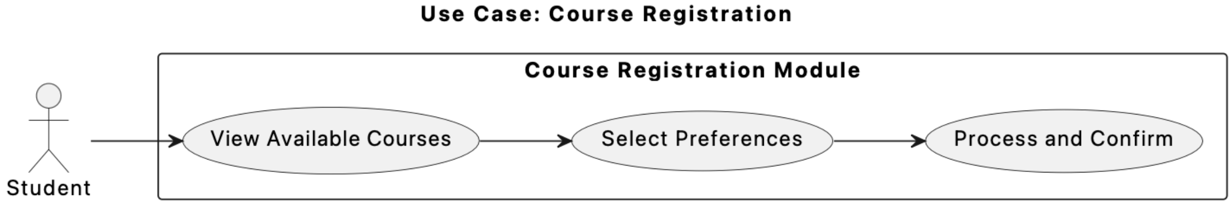
Use Case: Course Registration

Description:
Students can register for core or elective courses based on availability.

Usage Steps:

1. Student accesses course registration module.

2. Views list of available electives.
3. Selects preferences and submits.
4. Processes and confirms selections.

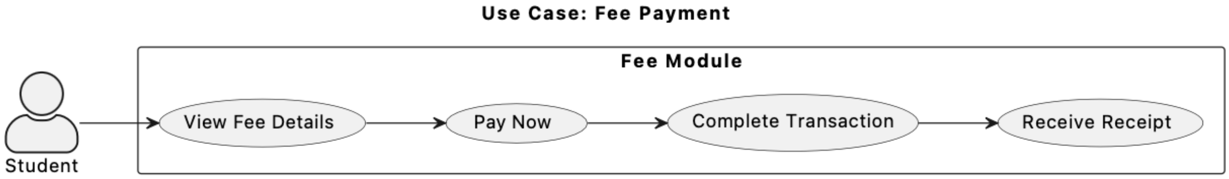


Use Case: Fee Payment

Description:
Allows students to check due fees and pay online.

Usage Steps:

1. Student opens fee module.
2. Views due amount and history.
3. Clicks "Pay Now."
4. Redirects to payment gateway.
5. Completes transaction and receives receipt.



3. Design Overview

The Intranet Portal for Mahindra University is built using a modular, role-based architecture powered by Node.js and Express.js on the backend, and MongoDB for data storage. The frontend is developed using HTML, CSS, JavaScript, with styling supported by Bootstrap.

The application supports two user roles—Student and Faculty—with access to specific features through their individual dashboards. Authentication is handled using JWT (JSON Web Tokens), ensuring secure role-based access and session management.

High-Level Modules

- **Authentication Module:** Handles user login, token generation, role validation, and secure access control.
- **Dashboard Module:** Displays personalized dashboards based on user roles.
- **Meeting Scheduler Module:** Students can request meetings with faculty; faculty can manage requests.
- **Course Material Module:** Faculty can upload lecture content; students can view and download materials.
- **Results Module:** Faculty can upload student results; students can access their individual grades.
- **Library Module:** Users can view available books, check status, and request books.
- **Fee Module:** Students can view due fees and pay through integrated UPI/payment links.
- **Course Registration Module:** Handles elective selection and submission of course preferences.
- **Events & Calendar Module:** Shows academic calendar and allows users to browse event details.
- **AI Assistance Module :** Provides intelligent support for viewing or uploading content.
- **Live Chat Module:** Enables real-time communication between students and faculty.

Data is stored in **MongoDB**, structured using collections specific to users, meetings, results, books, fees, etc. The backend APIs are **RESTful**, enabling smooth communication between client and server.

The system's architecture ensures **clean separation** between logic, data, and presentation layers, making it scalable, maintainable, and easy to extend with future enhancements.

3.1 Design Goals and Constraints

Design Goals

- **Role-Based Access Control:** Personalized access to features based on user role.
- **Modularity:** Clearly separated modules for easier development and testing.
- **Security:** Secure authentication using JWT and protected API routes.
- **Scalability:** Supports growing user base.
- **User Experience:** Intuitive dashboards and minimal-click access.

Constraints

- **MongoDB schema planning** needed for relationships.

- **JWT session handling** only; no refresh tokens.
- **Time-limited development** due to academic schedule.
- **Small team size**, so simplicity is prioritized.
- **Only open-source tools**.
- **No legacy code** or system integration required.

3.2 Design Assumptions

- Users will access the portal using modern web browsers.
- Each user has unique credentials.
- All backend communication is via RESTful JSON APIs.
- MongoDB will always be available during operations.
- JWT will handle session storage in local storage.
- User traffic will remain moderate.
- Deployment will be local or on university infrastructure.
- Frontend is responsive but not a dedicated mobile app.
- User roles are fixed for a session (no dynamic role switching).

3.3 Significant Design Packages

The Mahindra University Intranet Portal follows a well-structured **3-Tier Architecture**, designed to ensure maintainability, scalability, and separation of concerns. The system is decomposed into layered **packages/modules** with clear **responsibilities and dependencies** as outlined below.

A. Presentation Layer (Client Tier)

Packages:

- views/ – HTML files for each role (e.g., index.html, faculty.html, myprofile.html)
- assets/css/ – Styling assets (style.css, bootstrap.min.css)
- assets/js/ – Client-side scripts (main.js, dashboard.js, piechart.js)

Responsibilities:

- Handles user interactions and UI rendering.
- Sends authenticated API requests to backend.
- Updates DOM elements using user-specific data (e.g., name, role, tasks).

Dependencies:

- Depends on Application Layer (API endpoints).
- No dependency on Data Layer directly.

B. Application Layer (Node.js Backend Tier)

Packages:

- routes/ – Express routes (authRoutes.js, taskRoutes.js, profileRoutes.js)
- controllers/ – Business logic (authController.js, taskController.js)
- middleware/ – Authentication and authorization (authMiddleware.js)
- models/ – Mongoose schemas (User.js, Task.js, Schedule.js, UserProfile.js)
- utils/ – Utility functions (e.g., mailer.js, token.js)

Responsibilities:

- Manages authentication (JWT), authorization, and session handling.
- Performs validation, data manipulation, and API logic.
- Interacts with MongoDB via Mongoose models.

Dependencies:

- Uses express, mongoose, jsonwebtoken, bcrypt, cors.
- Depends on the models package for DB access.
- Used by Presentation Layer via HTTP requests.

Hierarchy:

- Routes → Controllers → Models
- Middleware intercepts requests before controllers.

C. Data Layer (MongoDB Tier)

Packages:

- MongoDB collections represented via Mongoose models:
  - User: Login and role data.
  - UserProfile: Personal, academic, and contact info.
  - Task: To-Do list per user.
  - Schedule: Daily schedule linked to user/faculty.
  - Feedback: Class reviews.
  - Event, Result, LibraryItem, etc. (Planned)

Responsibilities:

- Persistent storage of all user-related and domain-specific data.
- Each schema includes userId as a reference (for linking data).

Dependencies:

- Used only by the Application Layer.
- No upward dependency – pure data layer.

3. Data Layer (MongoDB)

Uses Mongoose schemas for collections:

- UserModel.js
- MeetingModel.js
- ResultModel.js
- MaterialModel.js
- BookModel.js
- AttendanceModel.js

3.4 Dependent External Interfaces

The following table lists the external interfaces this system relies on. These interfaces are all open-source or locally simulated, and are used to support key internal modules without requiring any paid services or third-party integration

External Interface	Used By Module	Purpose
MongoDB (Mongoose Driver)	All modules	Core database for all operations
JSON Web Token (JWT)	Auth, Middleware	Secure session token generation and
Socket.IO (optional)	Live Chat	Enables real-time messaging
Static UPI Simulation Link	Fee Portal	Simulates fee payment without real transactions
Local AI Script	AI Assistance	Provides suggestions and summaries using local logic

3.5 Implemented Application External Interfaces

The following table lists the public interfaces developed and exposed by the system for use within the application. These are RESTful APIs built using Express.js and are accessible to the frontend through HTTP requests.

Interface	Module	Functionality
/api/auth/login	Authentication	Login and JWT issuance
/api/auth/verifyToken	Middleware	Token validation for secure routes
/api/meetings	Meeting Scheduler	Meeting CRUD

/api/materials	Course Content	Upload/view materials
/api/results	Results	Faculty uploads, students view results

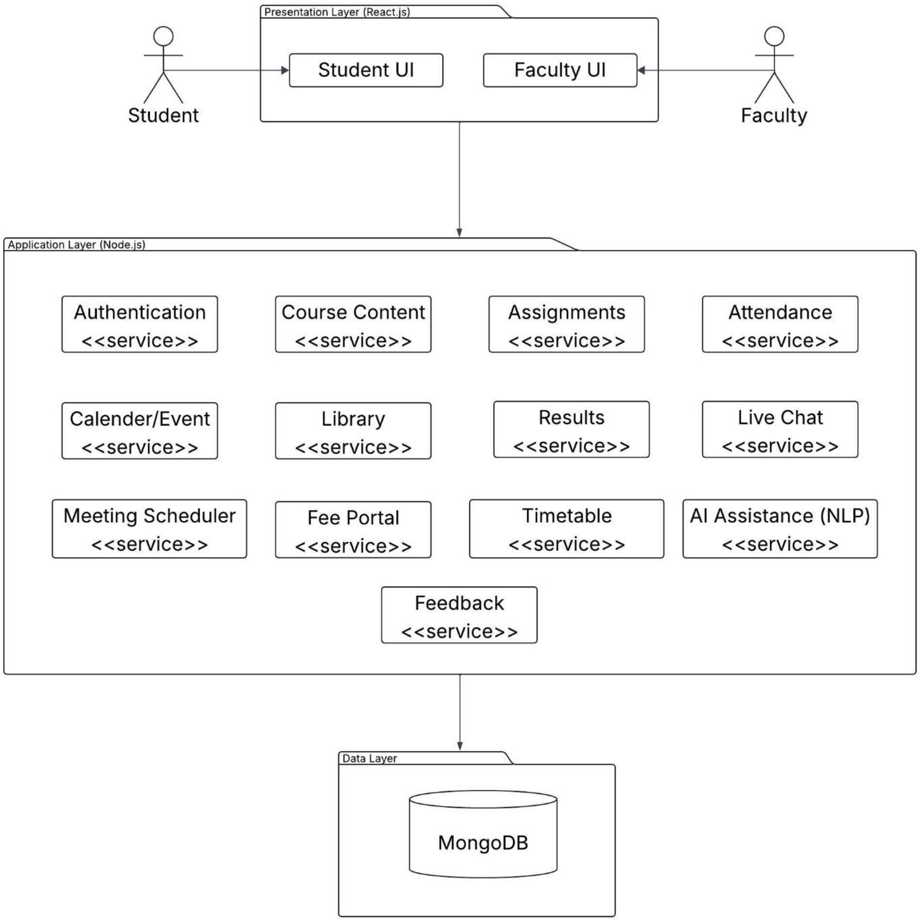
4. Logical View

This section describes the detailed design of the Intranet Portal through a layered view of the system. It begins with high-level interactions between modules and user roles and then drills down into how each module is structured internally to fulfill its responsibilities.

Layered Design Structure

The system follows a 3-tier architecture with clearly separated responsibilities:

3-Tier Architecture



Presentation Layer

• Handles the UI for Student and Faculty roles.
• Sends requests to the backend via REST APIs.
• Displays responses and manages client-side logic.

Application Layer

• Built using Node.js and Express.js.
• Contains business logic and route handlers.
• Manages security, validation, and module coordination.

Data Layer

• Powered by MongoDB using Mongoose.
• Stores all persistent data in schema-defined collections.
• Supports CRUD operations for each functional module.

Data Layer Interaction Between Modules and Use Cases

Each major use case (as listed in Section 2) is mapped to specific backend services. For example:

• Login & Role Authentication: Frontend calls /api/auth/login → Backend validates → JWT is returned → Dashboard loads.
• Meeting Scheduling: Student sends POST to /api/meetings → Meeting saved in DB → Faculty GETs meeting list → Accepts/rejects.
• Course Materials: Faculty uploads via /api/materials → Stored with reference → Student GETs material list per course.
• Live Chat: Uses Socket.IO → Establishes real-time WebSocket channel → Messages transmitted between users instantly.

These interactions reflect the logical orchestration of requests and data flow across layers.

Module-Level Internet Structure

Each module is composed of:
• Route file: Defines endpoint URLs and HTTP methods.
• Controller: Contains logic for request handling.
• Model (Mongoose): Represents the schema and interacts with MongoDB.
For example, the Meeting Scheduler module includes:
- meeting.routes.js
- meeting.controller.js
- MeetingModel.js

Class-Level Internet Collaboration

Classes (represented by Mongoose models) are referenced by controllers to perform actions.
For instance:
In the Results module, the ResultModel holds marks per subject per student.
The controller fetches or inserts records using methods like ResultModel.find() or new ResultModel().save().

Method-Level Decomposition (Example) Collaboration

To demonstrate detailed logic, the following is a simplified pseudocode for a controller method in the Meeting Scheduler module:

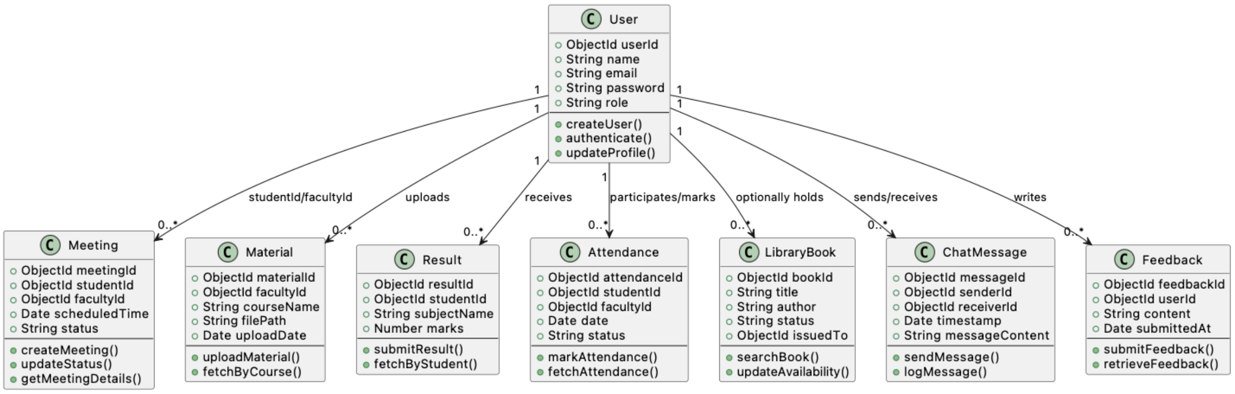
```
function createMeeting(req, res):
    Extract studentId, facultyId, date from req.body
    If input invalid:
        return error
    Check existing meetings at same time
  If clash found:
        return conflict message
    Save meeting with status = "pending"
    Return success response
```

This level of decomposition is followed for each module, where controller methods align directly with use case steps, maintain error handling, and interact with the database as needed.

This layered and modular breakdown ensures the system is easy to understand, test, and maintain, while each module cleanly collaborates with others to deliver a seamless user experience.

4.1 Design Model

This section presents the class-level decomposition of the system. Each module is represented by one or more key classes (or Mongoose schemas) that encapsulate specific functionality.



4.1.1 User Class

Responsibilities:

Manages user information and authentication data.

Relationships:

Associates with meeting, result, and course content records.

4.1.2 Meeting Class

Responsibilities:

Handles meeting request and scheduling information between a student and a faculty member.

Relationships:

References the User class for both student and faculty roles.

4.1.3 Material Class

Responsibilities:

Represents course material and content uploaded by faculty.

Relationships:

Connects to the User class via the facultyId attribute.

4.1.4 Result Class

Responsibilities:

Stores academic results for individual students.

Relationships:

Relates to the User class by linking each result to a student record.

4.1.5 Library Class

Responsibilities:

Encapsulates the details of library resources and their availability.

Relationships:
May reference a User when a book is issued.

4.1.6 Attendance Class

Responsibilities:
Tracks attendance data for sessions.
Relationships:
Links to the User class for both student and faculty.

4.1.7 LiveChat Class

Responsibilities:
Manages the log and transmission of real-time messages between users.
Relationships:
References the User class for sender and receiver details.

4.1.8 Feedback Class

Responsibilities:
Collects and stores feedback submitted by users.
Relationships:
Optionally relates to the User class if not submitted anonymously.

4.2 Use Case Realization

This section explains how each use case defined in Section 2 is realized within the system design. Each use case is broken down into:
• High-level flow between modules
• Class-level collaboration (how models, controllers, and routes work together)
• A sequence diagram**.**

Use Case: Login

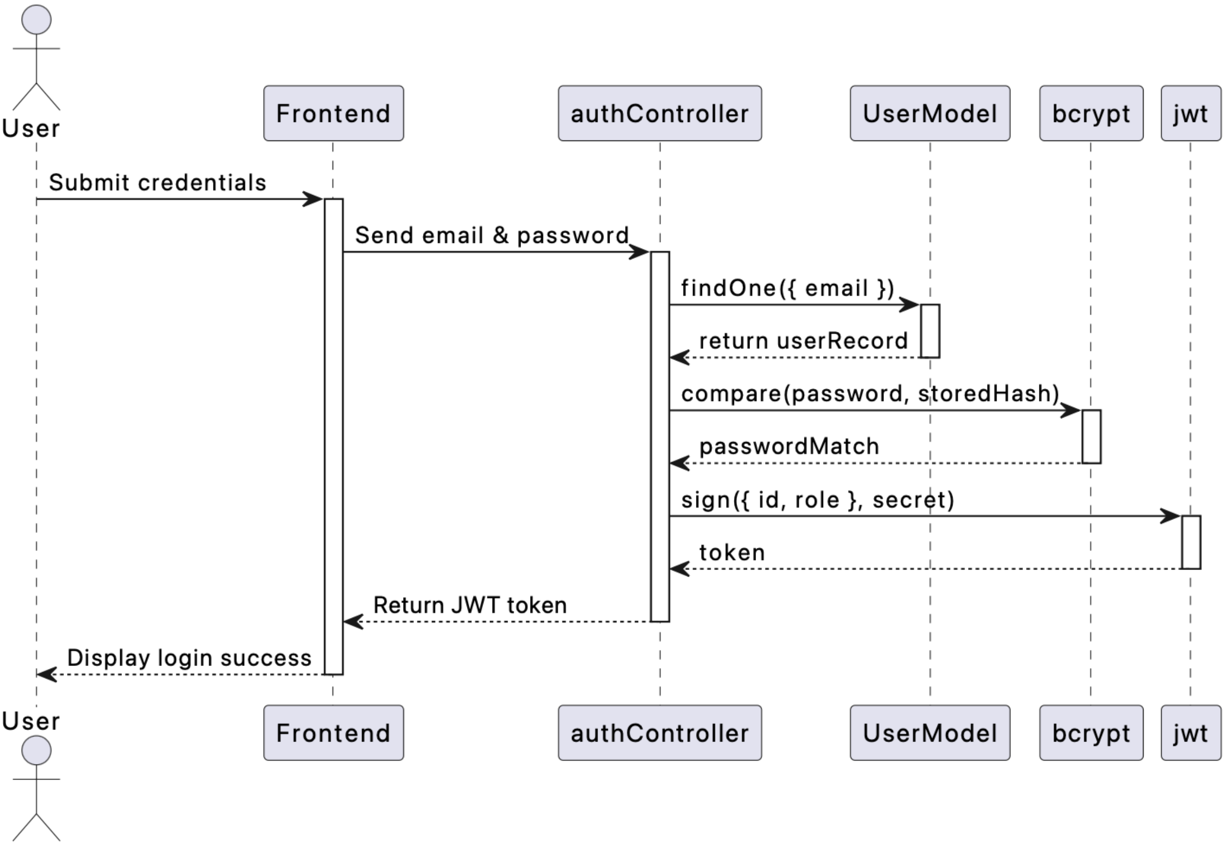
High-Level Flow:

- User submits login form with email, password, and role.
- auth.routes.js receives the POST request.
- auth.controller.js verifies user using UserModel.
- On success, JWT is generated and returned to the client.

Class-Level Collaboration:

- UserModel.findOne({ email })
- bcrypt.compare()
- jwt.sign()

Diagram:



Use Case: Role Authentication

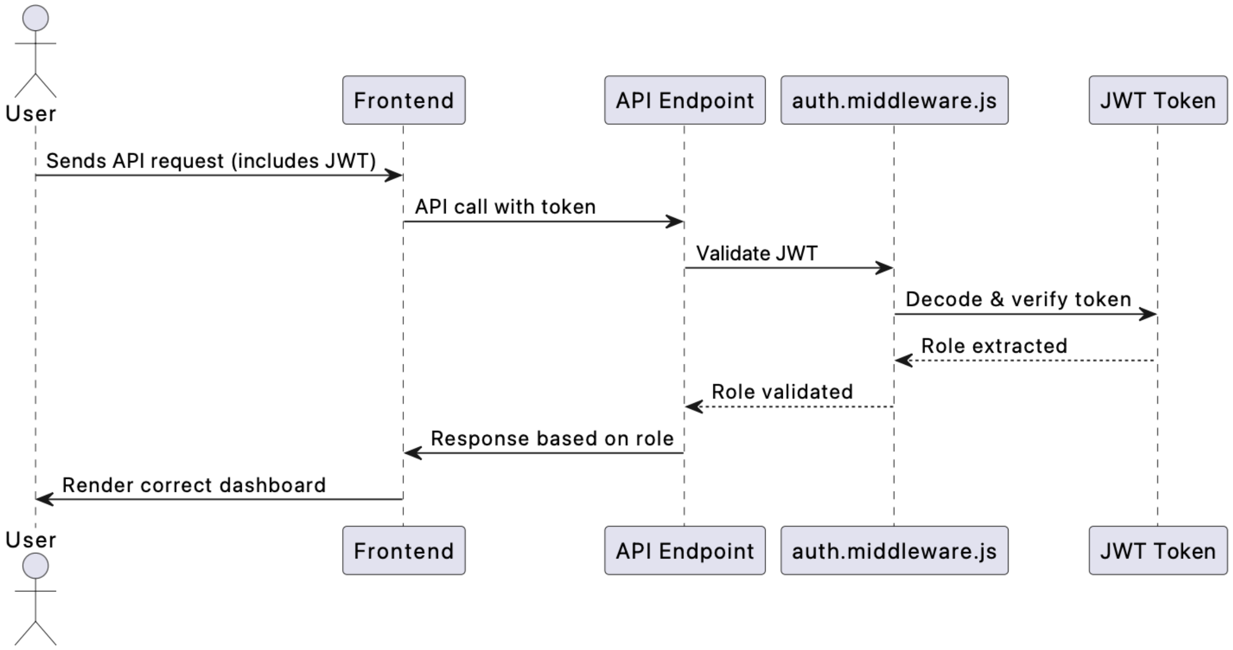
High-Level Flow:

- After login, token contains the user’s role.
- Frontend checks role to render correct dashboard.
- Backend verifies token on every API call via middleware.

Class-Level Collaboration:

- auth.middleware.js extracts and verifies role from JWT.

Diagram:



Use Case: Invalid Login

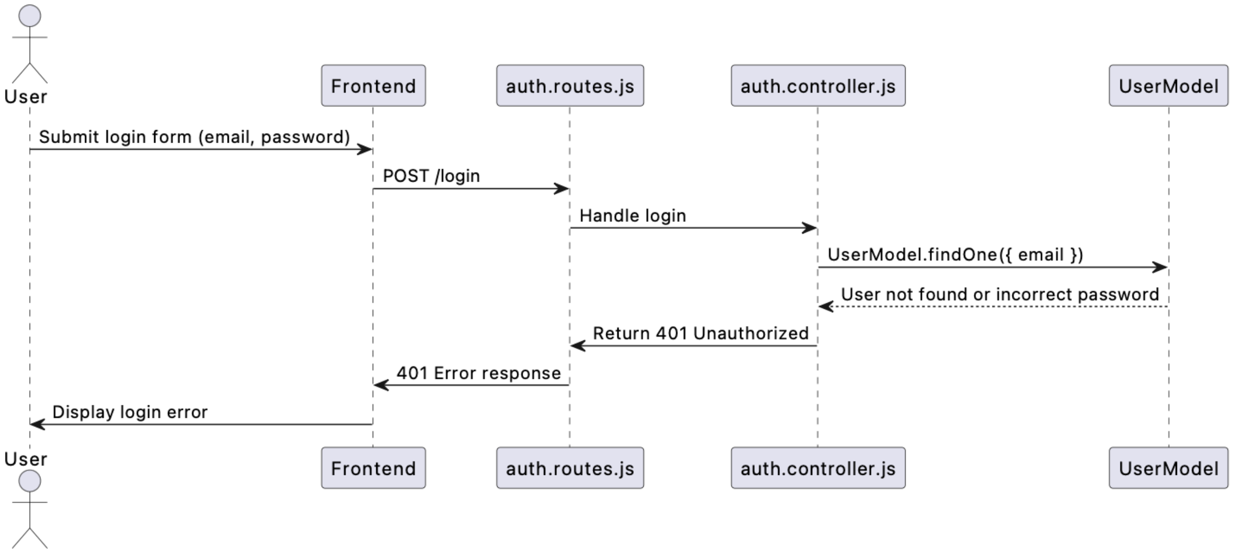
High-Level Flow:

- User submits wrong credentials.
- System returns error via auth.controller.js.

Class-Level Collaboration:

- Returns 401 Unauthorized from route handler if user not found or password mismatch.

Diagram:



Use Case: Event Registration & Calendar

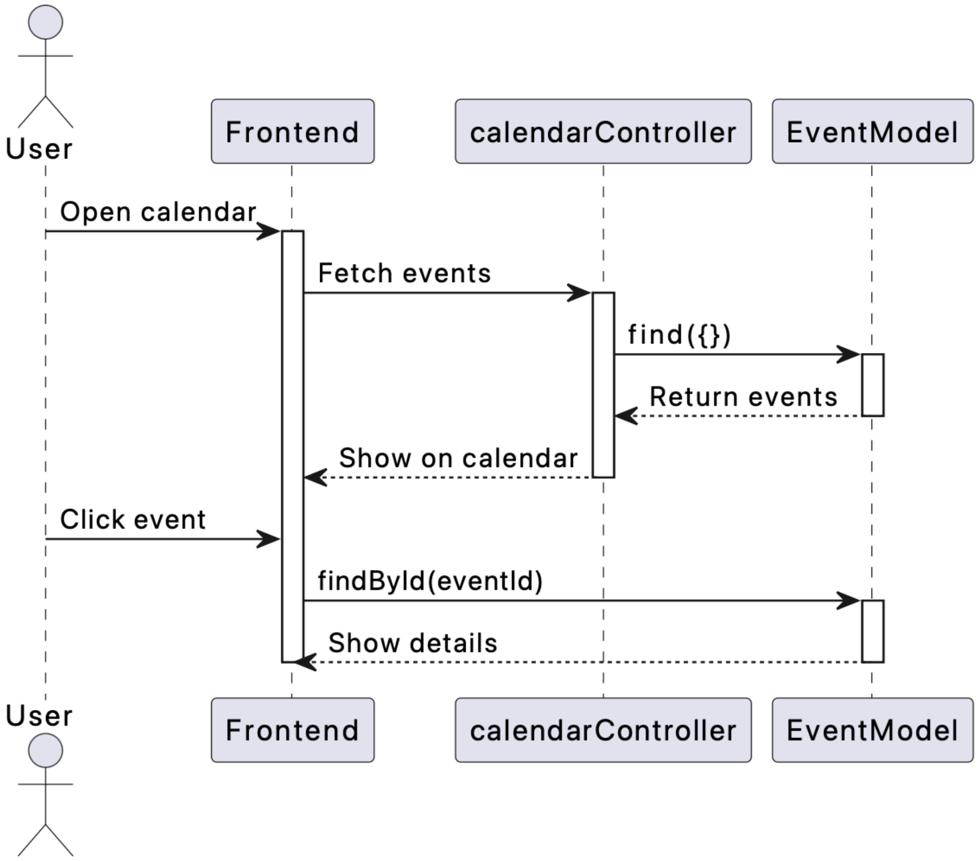
High-Level Flow:

- User navigates to calendar.
- Frontend sends GET request to /api/calendar.
- Server fetches event data from CalendarModel.

Class-Level Collaboration:

- calendar.routes.js → calendar.controller.js → CalendarModel.find({})

Diagram:



Use Case: Check Library Books

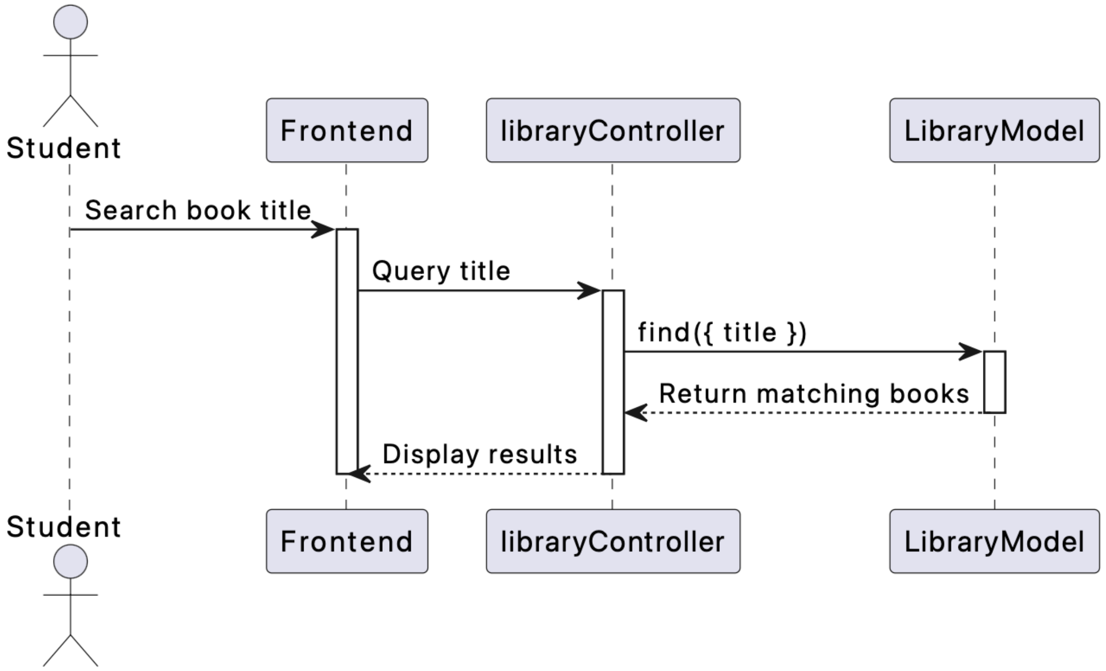
High-Level Flow:

- User inputs title/author in UI.
- Frontend calls /api/library?query=xyz.
- Backend searches LibraryModel.

Class-Level Collaboration:

- LibraryModel.find({ title: /xyz/i })

Diagram:



Use Case: Schedule Meetings

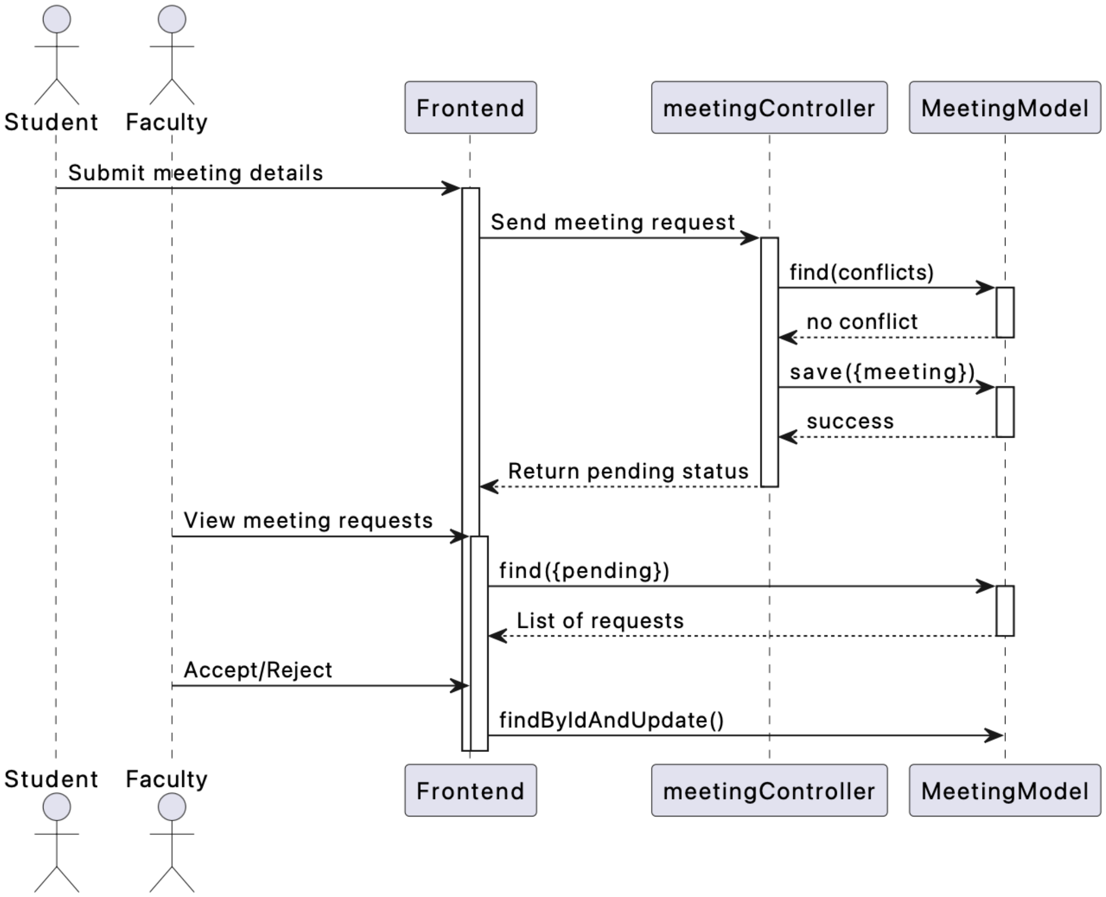
High-Level Flow:

- Student submits meeting form.
- /api/meetings POST handled by meeting.controller.js.
- Stored via MeetingModel.
- Faculty can GET, PATCH to accept/reject.

Class-Level Collaboration:

- MeetingModel.save(), MeetingModel.findByIdAndUpdate()

Diagram:



Use Case: Access Course Content

High-Level Flow (Student):

- Student selects course.
- Frontend sends GET request to /api/materials?course=xyz.

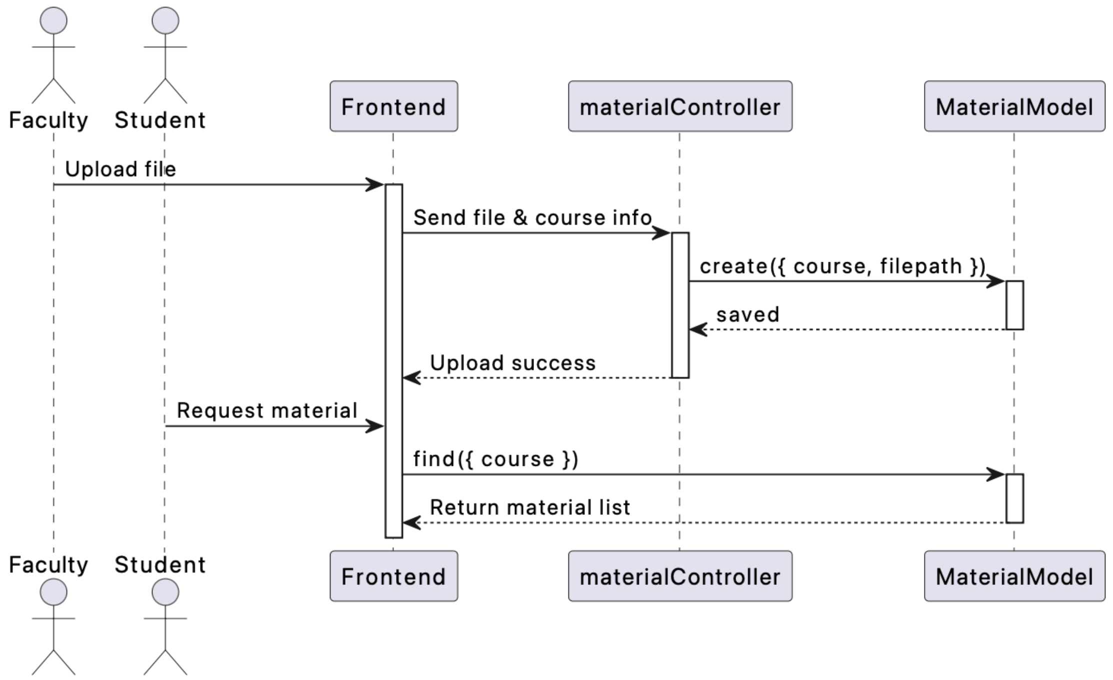
High-Level Flow (Faculty):

- Faculty selects course and uploads file.
- POST /api/materials saves metadata in MaterialModel.

Class-Level Collaboration:

- MaterialModel.find(), MaterialModel.create()

Diagram:



Use Case: AI Assistance

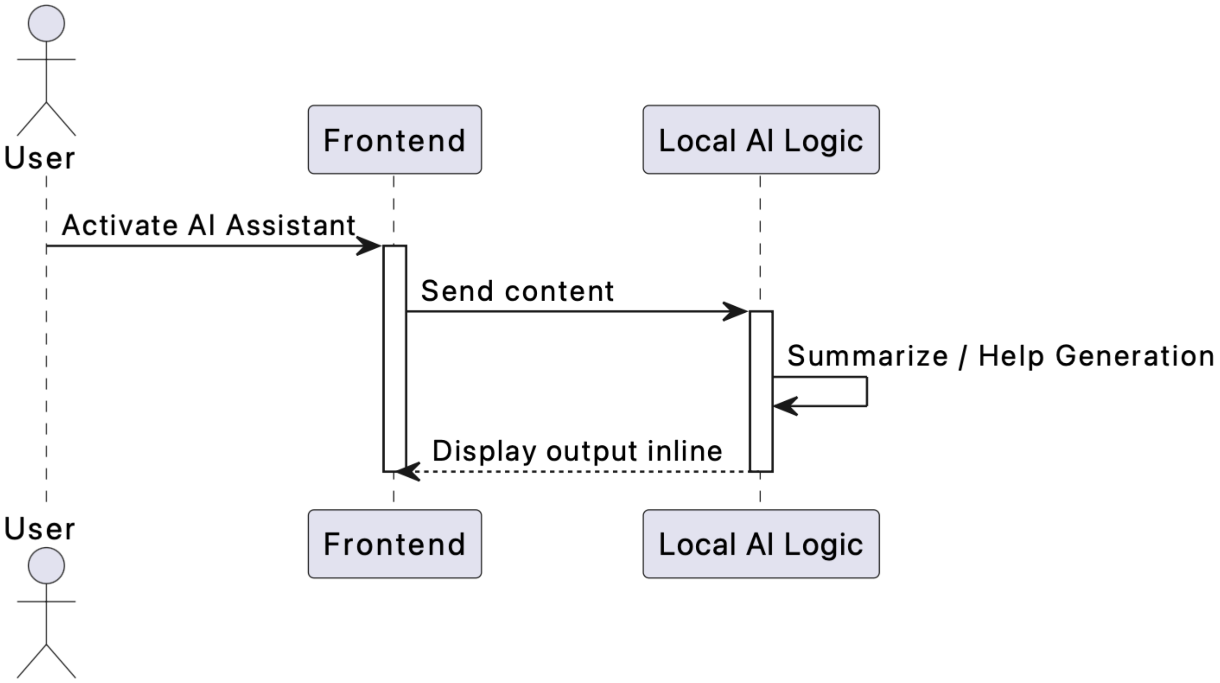
High-Level Flow:

- User clicks AI assist button.
- Frontend triggers local JS logic or calls /api/ai.
- Returns static or rule-based response.

Class-Level Collaboration:

- No external API used; response handled via predefined logic.

Diagram:



Use Case: Results

High-Level Flow (Faculty):

- Faculty submits marks via POST /api/results.

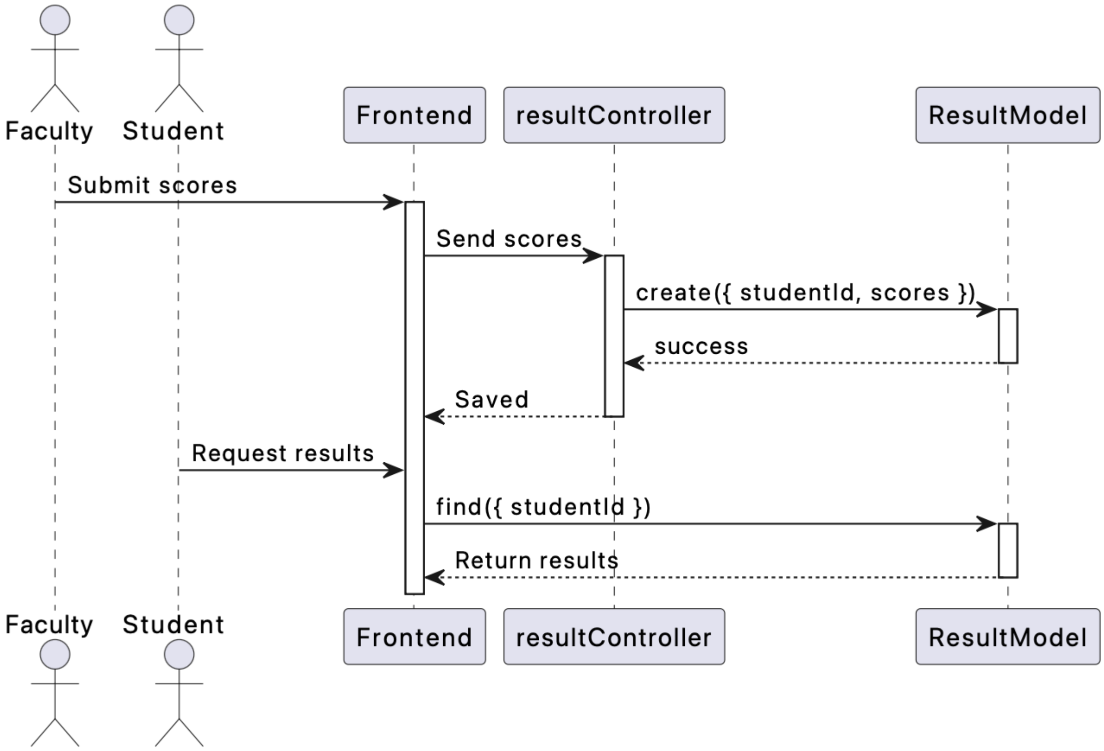
High-Level Flow (Student):

- Student sends GET /api/results.

Class-Level Collaboration:

- ResultModel.create()
- ResultModel.find({ studentId })

Diagram:



Use Case: Live Chat

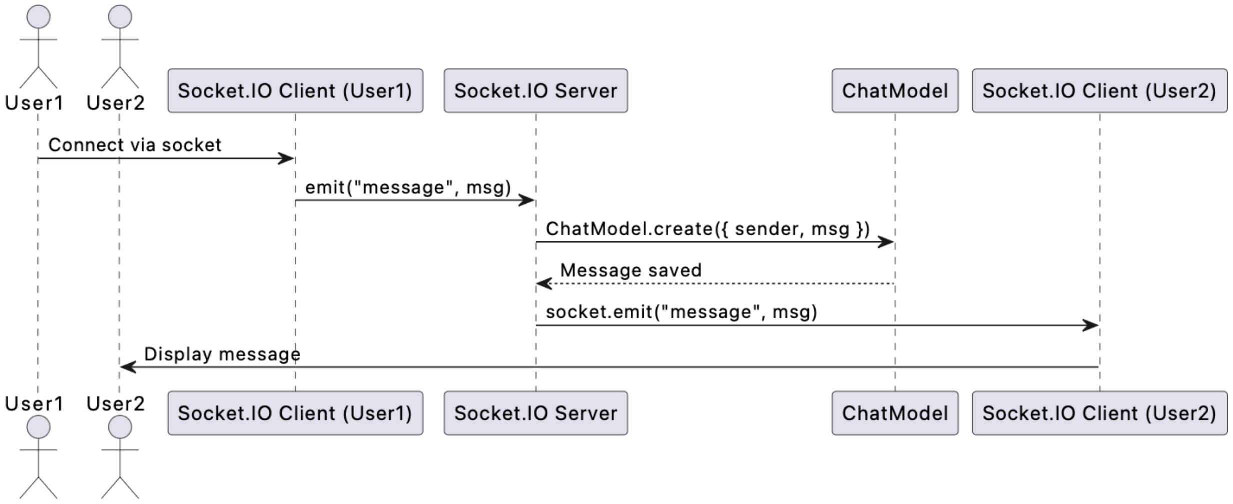
High-Level Flow:

- Socket.IO client connects.
- User sends message with socket.emit().
- Backend broadcasts via socket.on() to recipient.

Class-Level Collaboration:

- Optional: Store using ChatModel.create()

Diagram:



Use Case: Mark Attendance

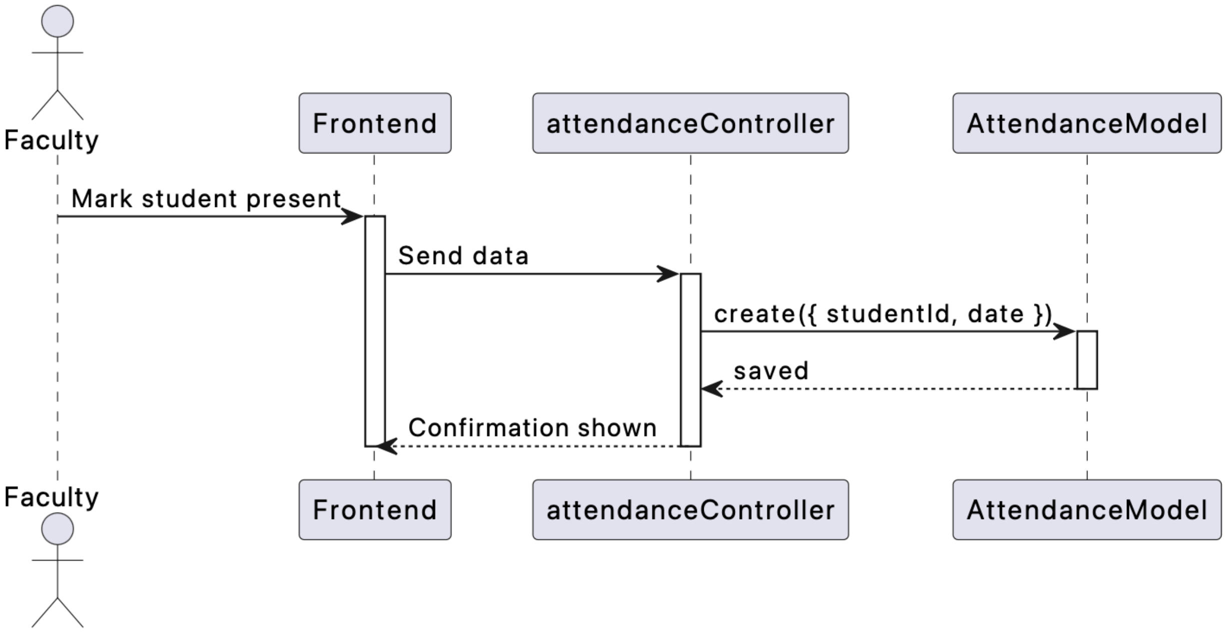
High-Level Flow:

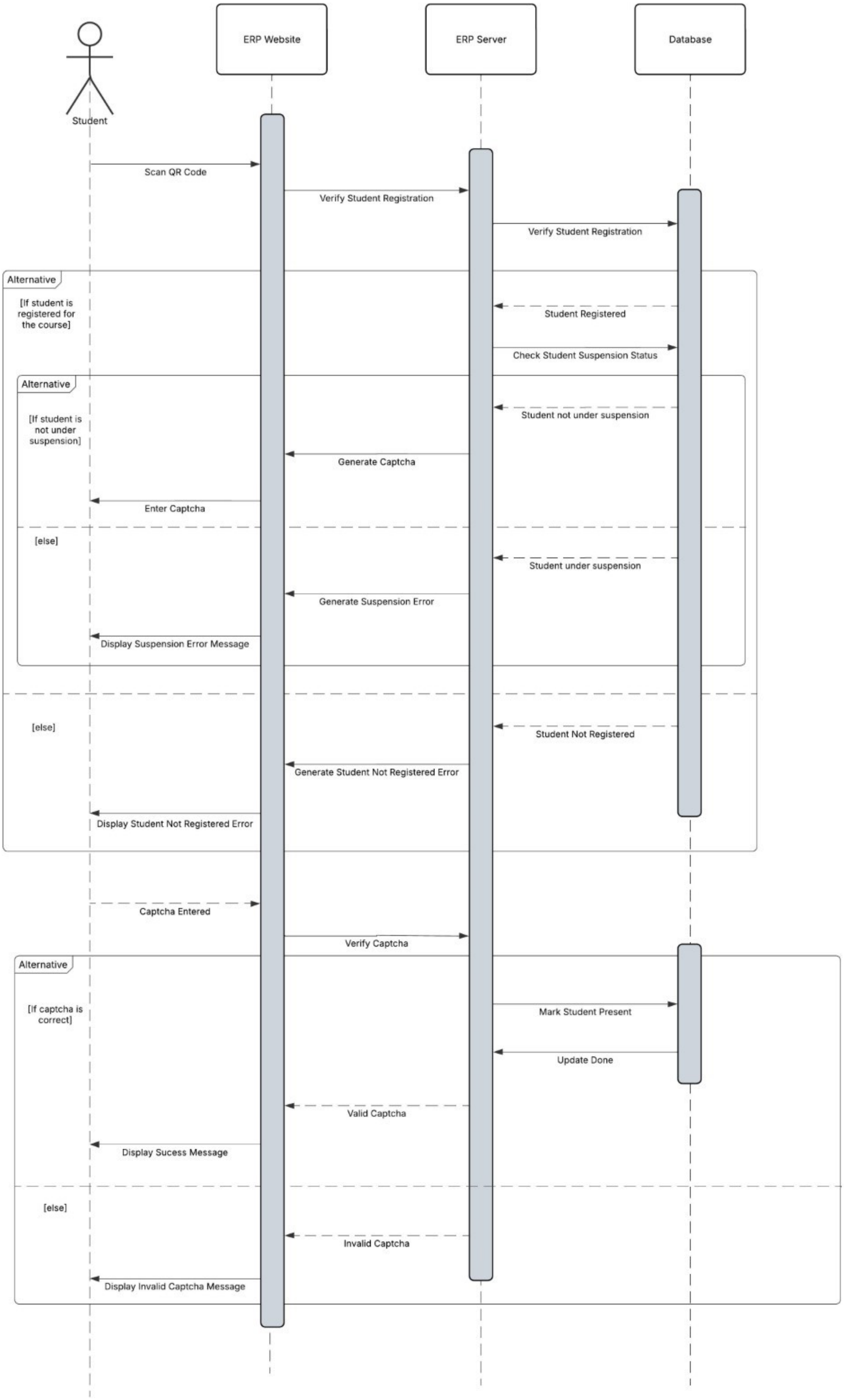
- Faculty selects student list and marks attendance.
- Data sent via POST /api/attendance.

Class-Level Collaboration:

- AttendanceModel.create()
- AttendanceModel.find({ studentId })

Diagram:




ERP Website
ERP Server
Database
Student
Scan QR Code
Verify Student Registration
Verify Student Registration
Alternative
[If student is registered for the course]
Student Registered
Check Student Suspension Status
Alternative
[If student is not under suspension]
Student not under suspension
Generate Captcha
Enter Captcha
[else]
Student under suspension
Generate Suspension Error
Display Suspension Error Message
[else]
Student Not Registered
Generate Student Not Registered Error
Display Student Not Registered Error
Captcha Entered
Verify Captcha
Alternative
[If captcha is correct]
Mark Student Present
Update Done
Valid Captcha
Display Sucess Message
[else]
Invalid Captcha
Display Invalid Captcha Message

Use Case: Course Registration

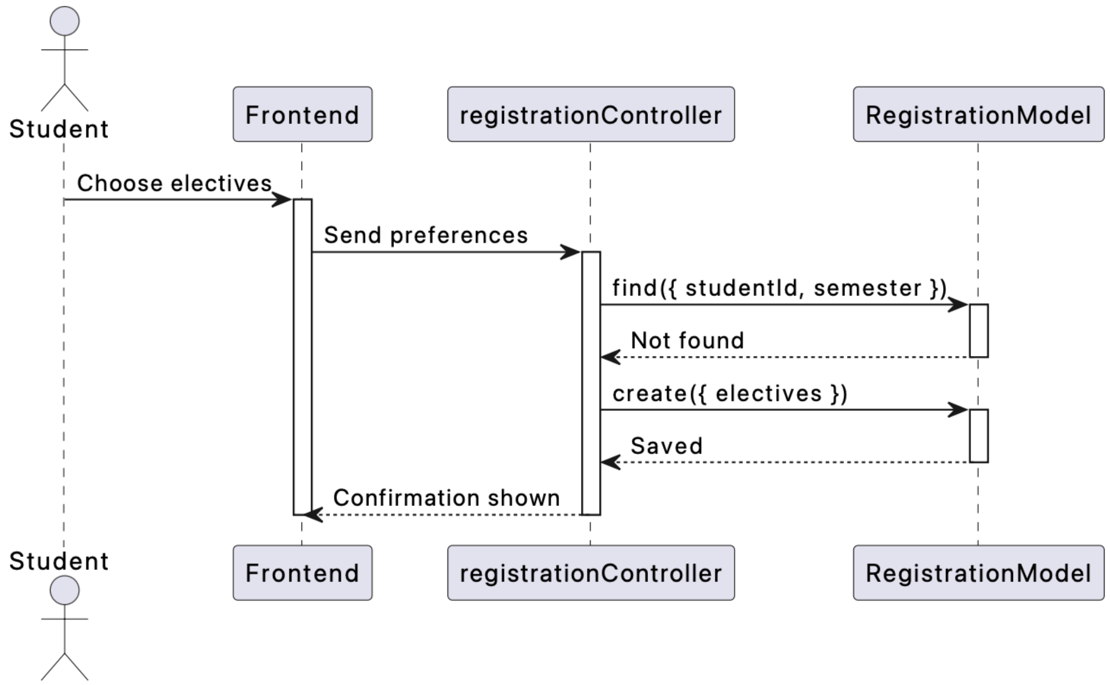
High-Level Flow:

- Student views list via GET /api/courses.
- Submits choices via POST /api/registrations.

Class-Level Collaboration:

- RegistrationModel.create()
- Validation for duplicate entries.

Diagram:



Use Case: Fee Payment

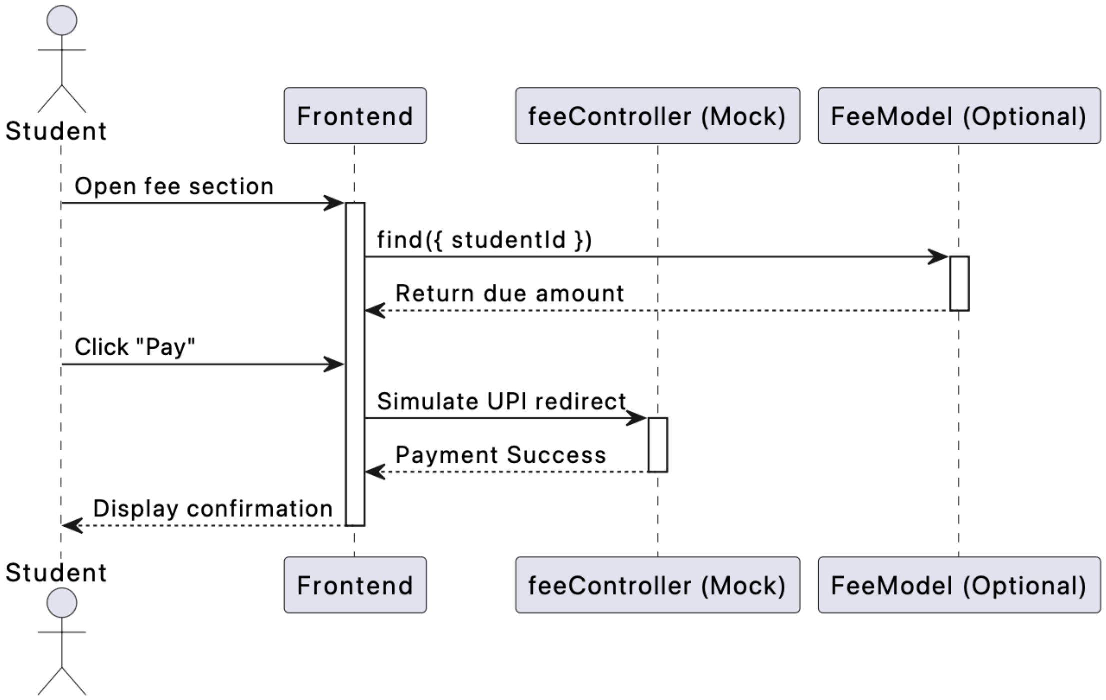
High-Level Flow:

- Student opens /api/fees.
- Fee info returned from FeeController.
- Simulated payment via redirect.

Class-Level Collaboration:

- No third-party gateway
- Static confirmation using dummy values

Diagram:



5. Data View

This section describes the persistent data storage perspective of the system. The Intranet Portal stores significant user and operational data using MongoDB, a document-based NoSQL database. Data persistence is essential for core modules such as login authentication, course content access, meeting scheduling, attendance, and academic results.

MongoDB collections are mapped directly to system modules using Mongoose schemas. Each collection maintains structured yet flexible data records that enable scalable performance and schema validation.

5.1 Domain Model

The domain model defines the main entities (domain objects) in the system and illustrates how they are logically related to each other. These entities represent the real-world components of Mahindra University's intranet environment, such as users, meetings, materials, and results. Each entity captures business-relevant attributes and is linked to others where applicable.

Entity Descriptions and Relationships

Entity	Description	Relationships

User	Represents students and faculty. Stores role, name, email, and authentication data.	- 1:N with Meeting (as requester/receiver) • 1:N with Result, Material, Attendance,
Meeting	Represents scheduled or requested meetings between students and	- Linked to User through studentId and facultyId
Material	Course files uploaded by faculty (PDFs, slides, notes, etc.).	- 1:N with User (faculty uploads)
Result	Academic results stored per student per	- 1:N with User (students)
Attendance	Records student attendance by date and	- 1:N with User (faculty marks for students)
Library Book	Represents books in the library and their	- Optional link to User when issued
Feedback	Text feedback submitted by students about faculty or	- Optionally linked to User
ChatMessage	Real-time messages between users.	- 1:N between User (sender and receiver)
CalendarEvent	Academic calendar events, talks, or	- Independent, global to all users
CourseRegistration	Stores elective choices or selected courses per	- 1:N with User (students register for

Key Relationship Types:

One-to-Many (1:N):

A faculty can upload many materials

A student can have many results

A user can request multiple meetings

Associations:

Feedback may or may not be linked to a user (anonymous)

Library books may be linked to a user only when issued

5.2 Data Model

The persistent data model represents the technical storage structure of the application. Each functional module corresponds to a MongoDB collection that holds documents representing records for users, meetings, course materials, and more.

Primary Collections - MongoDB Collection Overview

Collection Name	Purpose
users	Stores student and faculty credentials, roles, and metadata
meetings	Holds meeting requests and statuses
materials	Tracks uploaded course content with
results	Stores subject-wise marks for students
attendance	Records student attendance by class and
library	Stores book data and availability status
feedback	Collects user-submitted feedback
chat (optional)	Stores live chat messages (if chat history is persisted)
calendarEvents	Academic and event schedule data
registrations	Stores course or elective selections

Each collection is defined and validated through a Mongoose schema, which specifies fields, types, defaults, and relationships using ObjectId references.

5.2.1 Data Dictionary

The following tables define each field within the major collection, describing its purpose and data type.

Users Collection

Field	Type	Description
_id	ObjectId	Unique user ID
name	String	Full name
email	String	Unique email address
password	String	Hashed password
role	String	"student" or "faculty"

Meetings Collection

Field	Type	Description
_id	ObjectId	Unique meeting ID
studentId	ObjectId	Refers to the requesting student
facultyId	ObjectId	Refers to the invited faculty member
scheduledTime	Date	Requested date and
status	String	"pending", "accepted", "rejected"

Materials Collection

Field	Type	Description
_id	ObjectId	Unique material ID
facultyId	ObjectId	Uploader (faculty)
courseName	String	Course or subject
filePath	String	Path or link to
uploadDate	Date	Timestamp of upload

Results Collection

Field	Type	Description
_id	ObjectId	Unique result ID
studentId	ObjectId	Refers to student receiving marks
subjectName	String	Name of the subject
marks	Number	Score/grade

Attendance Collection

Field	Type	Description
_id	ObjectId	Unique attendance ID
studentId	ObjectId	Student being marked
facultyId	ObjectId	Faculty recording
date	Date	Session date
status	String	"present" or "absent"

Library Collection

Field	Type	Description
_id	ObjectId	Unique book ID
title	String	Book title
author	String	Book author
status	String	"available" or "issued"
issuedTo	ObjectId	(Optional) user ID if

Feedback Collection

Field	Type	Description
_id	ObjectId	Unique feedback ID
userId	ObjectId	(Optional) sender's ID
content	String	Feedback text
submittedAt	Date	Timestamp of

CalendarEvents Collection

Field	Type	Description
_id	ObjectId	Unique event ID
title	String	Event name
description	String	Event details
date	Date	Date of the event

Registrations Collection

Field	Type	Description
_id	ObjectId	Unique registration ID
studentId	ObjectId	Refers to the student
courseChoices	[String]	List of elective/course preferences
submittedAt	Date	Timestamp of

6. Exception Handling

This section describes how the system handles exceptions and errors during runtime. It covers types of exceptions defined across the application, when they may occur, how they are logged or reported, and what follow-up actions are expected by the system or user.

1 Types of Exceptions and Their Triggers

Type	Context	Trigger
AuthenticationError	Login/Auth Middleware	Invalid or missing token
ValidationError	All modules	Bad input fields
DatabaseError	DB Operations	Query/connection failures
NotFoundError	Results/Meetings	Record doesn’t exist
ConflictError	Registrations	Duplicate record
SocketError (opt)	Chat	WebSocket fail
FileUploadError	Uploads	Oversize/wrong file type

2 Handling Mechanism

Centralized Error Middleware

All routes use next(err) to pass exceptions to a central error handler in Express.
Errors are returned with appropriate HTTP status code and a JSON message.

```
app.use((err, req, res, next) => {
   console.error(err.stack);
   res.status(err.statusCode || 500).json({ message: err.message });
});
```

Errors like AuthError, NotFoundError, ConflictError can be defined as custom JS classes and thrown using throw new AuthError("Invalid token").

3 Logging and Feedback

Console Logging:

All critical errors (e.g., failed DB operations, internal server issues) are logged to the console during development.

Client Feedback:

User-friendly error messages are shown on the frontend (e.g., “Incorrect email or password” or “Meeting already exists at this time”).

Optional Enhancements (Future Scope):

• Add file-based or external error logging (e.g., Winston, Sentry)

- Store logs in a logs/ directory or logging service

4 Logging and Follow-Up Actions

Scenario	System Action
Invalid Login	Reject request, display login error message
Duplicate Meeting	Reject booking and prompt for rescheduling
Missing Fields in Forms	Block submission and highlight required fields
File Upload Failed	Prompt user to retry with valid file
Record Not Found	Show “not found” message or fallback content
Database Connection Fails	Return generic error, advise user to retry later

7. Configurable Parameters

This section describes the configuration parameters used in the application that are externalized for easier deployment and modification. These parameters are typically stored in a .env file and accessed using the process.env object in the Node.js environment.

Name	Purpose	Dynamic?
PORT	App listening port	Yes
MONGO_URI	MongoDB connection string	No
JWT_SECRET	Token signing key	No
NODE_ENV	Environment mode (dev/prod)	Yes
UPLOAD_LIMIT	Max file size for uploads	Yes
FRONTEND_URL	CORS settings	Yes
SOCKET_IO_ORIGIN	Security config for WebSocket	Yes

8. Quality of Service

This section provides an overview of the system's non-functional qualities, which ensure it operates reliably, securely, and efficiently under real-world conditions. Key aspects include application availability, system security, performance optimization, and production-level monitoring and control.

The following subsections describe how each of these quality dimensions has been addressed in the design and implementation of the Intranet Portal.

8.1 Availability

The Intranet Portal is designed to provide high availability at all times, ensuring students and faculty can access course materials, results, meeting schedules, and other academic resources whenever needed — including outside of traditional working hours.

Design Aspects Supporting Availability:

• Stateless architecture using JWT: Enables continuous uptime without reliance on session-based server memory, supporting horizontal scaling.

• Module independence: Each functional component (e.g., materials, meetings, results) is isolated, ensuring one module's failure doesn't impact others.

• No third-party service dependencies: All core features (login, uploads, viewing) are handled internally, minimizing external failure risk.

Design Aspects That May Impact Availability:

• Database operations: Maintenance tasks like backups, indexing, or bulk imports must be scheduled to avoid downtime.

• Heavy file uploads: Large content uploads (e.g., lecture slides) may cause short-lived slowdowns if not optimized.

The system is intended to be accessible 24/7, supporting self-paced learning and flexibility for users across different schedules.

8.2 Security and Authorization

Security and role-based access are essential for ensuring that only authorized users can access academic resources, submit information, or modify data. The system enforces strict separation between student and faculty privileges in line with business requirements.

Authorization Features and Implementation:

• JWT-based access control: All authenticated users are issued a JSON Web Token (JWT) on login. This token is required to access any protected route.

• Role-specific routing: The user's role (student or faculty) is embedded in the JWT and verified on each request. Different routes or features are exposed based on role:

- Students can only view content, register, and request meetings.
- Faculty can upload materials, enter results, and mark attendance.

• Route-level protection: Middleware validates the JWT token on protected endpoints and denies access if the token is missing, expired, or invalid.

Additional Security Measures:

• Password hashing: All passwords are securely hashed using bcrypt before storage in the database.

• No sensitive data exposure: Tokens do not store raw passwords or sensitive database identifiers.

• Input validation: Server-side validation prevents injection attacks or malformed requests from reaching the database.

User Access Setup and Management:

• Initial user data (students and faculty) can be added via a seeded script or admin console if needed.

• While there is no Admin UI in this version, the system can be extended to support role assignment, password resets, or account management interfaces if required in future.

8.3 Load and Performance Implications

The system is expected to serve the Mahindra University community, which includes a limited number of students and faculty. While the expected load is moderate, the design ensures that performance remains stable even under peak usage, such as during exam result releases, bulk file uploads, or course registration windows.

Design Considerations for Performance

• Stateless REST API Design: Allows for lightweight and independent request handling. The server does not retain session data between requests, improving throughput.

• Indexed MongoDB Collections: Fields like email, studentId, courseName, and scheduledTime are indexed to support fast lookups and avoid collection scans.

• Asynchronous Non-blocking I/O: Node.js handles concurrent requests efficiently without thread blocking, ideal for API response times under academic load.

• File Upload Handling: Material uploads are streamed and processed with size limits to avoid memory overflow.

Load and Growth Projections

Component	Expected Load
Login Requests	Moderate spikes at class times and assessment periods
Meeting Requests	10–20 per faculty per week; peaks near exam weeks
Results Upload	Batch updates at end of semester by faculty
Course Materials	Average 2–5 uploads per subject; accessed daily by students
Chat Messages (if stored)	High frequency during working hours; low message size
Attendance Marking	Daily operation by faculty per class group
Library Queries	Light usage; mostly read-heavy

Support for Load and Performance Testing

• Test Plan Preparation: Each module can be stress tested using tools like Postman (collections), Apache JMeter, or k6.
• Simulated Peak Testing: Login bursts and batch file uploads can be simulated to observe performance thresholds.
• Database Growth: Projected to remain lightweight due to academic-cycle-based data. Archival or cleanup strategies may be introduced for older records.

8.4 Monitoring and Control

The application includes basic monitoring and control features suitable for a university-level system. These mechanisms help track system behavior, display useful messages to users, and support future maintainability.

Controllable Processes

Process	Description
Token Validation Middleware	Checks if the user is logged in and authorized before allowing access to routes.

Error Handling Middleware	Catches and handles errors across all modules; returns clear messages to the frontend.
Meeting Request Handling	Automatically updates meeting statuses based on faculty actions.
Material Upload Handling	Processes uploaded files and stores metadata without blocking other operations.

Basic Monitoring Features

What It Tracks	How It Helps
Login Success or Failure	Helps identify if a user is having trouble logging in.
Meeting Requests per User	Can be used to understand user activity and usage trends.
Upload Confirmation Messages	Lets users know if file uploads or data submissions were successful.
Console Logs (Development)	Developers can see what’s happening internally, especially during testing.