Product Documentation

University Management System

Members

Core Members

Moe Thiha
Ye Man Aung
Nyein Min Htun
Htoo Aung Lin

Acknowledged Contributors

Thiri Han (UI/UX)
Swam Pyae Aung (UI/UX)

2024-2025 Academic Year
University of Computer Studies Yangon
B.C.Sc (KE) - Semester VI
CS-205 (Web Development with Python)

Table of Contents

VOLUME I
FOUNDATION & REQUIREMENTS

ABSTRACT

The **University Management System (UMS)** is a web-based platform designed to streamline academic and administrative processes within universities. It addresses the challenges of decentralized data handling and inefficient communication by providing a unified system for **Executives, Instructors, and Students**. The platform enables centralized control, efficient course and resource management, and enhanced student engagement through a user-friendly interface. Furthermore, UMS incorporates robust **security mechanisms** and is architected with **scalability in mind**, ensuring adaptability for future institutional growth and technological advancements.

INTRODUCTION

In modern educational institutions, managing academic, administrative, and student-related activities manually can be inefficient, time-consuming, and prone to errors. Universities often face challenges such as data redundancy, inconsistent records, delayed reporting, and difficulty in tracking student progress, course schedules, faculty assignments, and overall academic control. To address these challenges, a **University Management System (UMS)** is essential. This system provides a centralized platform to efficiently manage student information, faculty data, courses, assessments, results, and administrative tasks. By automating key processes and providing real-time access to accurate information, the system enhances operational efficiency, ensures data integrity, and supports informed decision-making.

OBJECTIVES

1. Automate and simplify administrative tasks for executives, including managing faculty, degrees, departments, courses, and overall academic statistics.
2. Provide university executives with efficient and centralized control over the overall academic structure of the university.
3. Provide instructors with tools to manage courses, assessments, and student interactions effectively.
4. Enable students to access course materials, results, and communication channels seamlessly.
5. Implement robust security measures to protect sensitive information and ensure compliance with data protection regulations.

REQUIREMENT SPECIFICATION AND ANALYSIS

1.1.1 User Identification

Designed for semester-based universities. There will be three specific users/actors

1. **Executives** who are the top-level executives of a university (e.g., Rector)
2. **Instructors** who teach courses to students and operate day-to-day academic processes in the university.
3. **Students** (fellow students attending the university)

For each actor, there will be a secure and trustworthy authentication system.

1.1.2 Authentication Process

The University Management System (UMS) categorizes users based on their authority within the university and the system itself. There are two main types of users:

1. **Standard Users** – Users with limited access and control over the system. This group includes:
   i. **Instructors**
   ii. **Students**
2. **Authority Users** – Users with almost full control over the system. This group includes **Executives.**
3. **Public Users** – Visitors who can access general information about the university, such as available courses, degrees, labs, and public notices, without authentication.

Registration and Access

Standard Users: To use the UMS as a standard user, registration must be completed with a specific user role (either student or instructor). Access is granted only after the registration is reviewed and approved by an executive.

Executives: There is no registration process for executives. A user can become an executive only if an existing executive adds them to the system.

Note: *Instructors may also be referred to as* ***admins****, as they perform various administrative tasks within the university.*

1.1.3 Functional Requirements

➢ As an Executive

1. Log in to the system securely
2. Add new executives to the system
3. Edit and update university information
4. Add and edit faculties, degrees, and departments
5. Assign and change heads of departments and faculties
6. Add courses and assign them to departments
7. Define the marking scheme for each course
8. Define syllabus structure for each degree by semester

9. Start new terms specifying semester timeline, exam, and result dates
10. Assign instructors to courses per term
11. View and manage previous academic terms
12. Post notices on the university noticeboard visible to all users
13. Edit noticeboard details
14. Approve or reject notice posts uploaded by instructors
15. Approve instructor and student registrations
16. Assign department and job positions to approved instructors
17. Approve student enrollments for related terms and batch. Monitor university operations through a professional dashboard
18. Manage and edit own profile and password securely
19. Graduate, drop, or suspend students
20. Retire, put on leave, or transfer instructors
21. Change the department of instructors
22. Delete the account and leave the system permanently
23. Get notification updates for everything

➢ As an Instructor

1. Register and submit details for admin approval
2. Log in after approval
3. Access previously stored course documents and refer them to students
4. Upload new course documents for currently taught courses
5. Create assessments with multiple questions and file attachments
6. Accept student answers in text or file formats accordingly
7. Allow automatic and manual grading of assessments
8. Schedule and organize meetings via the system.
9. Conduct videoconferencing sessions using meeting IDs.
10. Upload notices to the noticeboard (needs to be approved by an executive)
11. View notices uploaded on the noticeboard.

➢ As a Student

1. Register and wait for admin approval
2. Log in post-approval and manage personal profile with password changes requiring old password confirmation
3. View degree and course details comprehensively
4. Select a degree and enroll for a batch in a term
5. View documents for courses attended in each term
6. Submit answers to assessments within deadlines
7. View results per term
8. Join videoconferencing sessions scheduled by instructors for classes and meetings
9. Access the noticeboard and view the notices by executives and instructors.

10. Access a dedicated mailbox feature within the system to facilitate communication and community building
11. Create and submit posts in the mailbox to share information or any post
12. All posts submitted by students must be approved by a designated mailbox admin before they are visible to other students
13. Report inappropriate posts or comments to the mailbox admins for review and action
14. Receive notifications for every update.

1.1.4 Non-Functional Requirements

1. **Security:** Secure login with password protection and facial recognition. Encrypt data transmission.
2. **Performance:** Reduce page load times and responsive UI with AJAX to reduce latency.
3. **Scalability:** Ability to support increasing numbers of users across multiple terms and years.
4. **Usability:** Easy-to-understand interface and mobile responsiveness
5. **Maintainability:** Modular and well-documented code base for easy updates and debugging
6. **Privacy:** Compliance with data protection laws and confidentiality of user data
7. **Executive-specific requirements**
   - Dashboard performance with real-time data visualization
   - Secure approval workflows with role-based access control

Instructor-specific requirements

- Reliable document handling and storage with version control
- Accurate and timely calculation and communication of assessment results

Student-specific requirements

- Seamless access to course materials and videos without interruption
- Notification mechanisms to alert students about assessment deadlines and results.

System Overview

1.1.5 High Level Description

- The **University Management System** is a modular, web-based platform built with Django to digitize and streamline the academic and administrative processes of a university or college. It provides tailored interfaces and workflows for different user roles—**executives**, **instructors** (faculty), and **students**—enabling efficient management of courses, assessments, enrollments, notifications, internal communications, and institutional notices.
- The system is designed for scalability, maintainability, and security, with each major domain encapsulated in its own Django app. It supports both authenticated and public access, offering a public portal for general information and restricted dashboards for internal users.

1.1.6 System Scope

➢ Included:

1. User management for executives, instructors, and students.
2. Course, degree, department, and term management.
3. Batch and enrollment management.
4. Lab management and related APIs.
5. Assignment and quiz creation, submission, and grading.
6. Result calculation, visualization, and notification.
7. Ratings and reviews for courses and instructors.
8. Internal messaging/mailbox system for students.
9. Notification system for all user roles.
10. Noticeboard with approval workflow for institutional notices.
11. Public portal for general information and announcements.
12. Role-based dashboards and access control.
13. Modular, extensible architecture for future features.

➢ Excluded:

1. Financial transactions (e.g., tuition payments, payroll, invoicing).
2. Library management (book loans, cataloging).
3. Hostel/accommodation management.
4. Inventory or asset management.
5. Human resources (HR) and payroll processing.
6. Third-party integrations not related to academic management (e.g., payment gateways).
7. Advanced analytics or business intelligence beyond basic reporting and dashboards.

VOLUME II
DESIGN & ARCHITECTURE

This volume presents the design and architectural blueprint of the **University Management System (UMS)**. While Volume I defined the requirements and objectives, this section focuses on how the system is structured, how its components interact, and how the requirements are realized through design decisions.

The chapter covers:

Use Case Diagrams & Explanations – to illustrate how different actors (students, instructors, executives) interact with the system and what functionalities they can access.

Sequence Diagrams & Explanations – to show the step-by-step flow of interactions between system components during specific operations.

Entity–Relationship (ER) Diagrams – to model the system's data structure, describing entities, their attributes, and relationships across different modules.

Technology Stack – to present the tools, frameworks, and technologies chosen for implementation, along with justifications for their selection.

Together, these design artifacts provide a clear roadmap from conceptual requirements to technical implementation, ensuring the UMS is scalable, efficient, secure, and aligned with the university's operational needs.

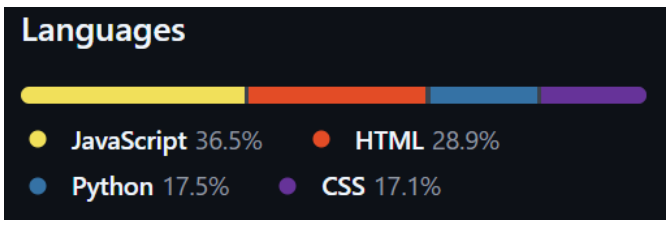
2.1 Technology Stack

For the frontend, we built responsive and mobile-friendly interfaces using **HTML, CSS, JavaScript, and Bootstrap**, which accelerated UI development through reusable components and utility classes. We enhanced interactivity with **AJAX**, enabling smooth real-time updates without full page reloads.

On the backend, we used **Django**, leveraging its **ORM**, templating, and admin interface for rapid and secure development. We added **custom middleware**, *"the bouncer"*, for layered security including authentication, brute force protection, and payload validation.

For storage, we deployed a **PostgreSQL** database on Render.com, ensuring data consistency across the team. **DBeaver** was used to simplify queries and database administration.

The following chart illustrates the distribution of programming languages used in our project, based on our **GitHub repository statistics**.



2.2 Use Case Diagrams and Explanations

2.2.1 Use Cases for Authentication from Different User Perspective

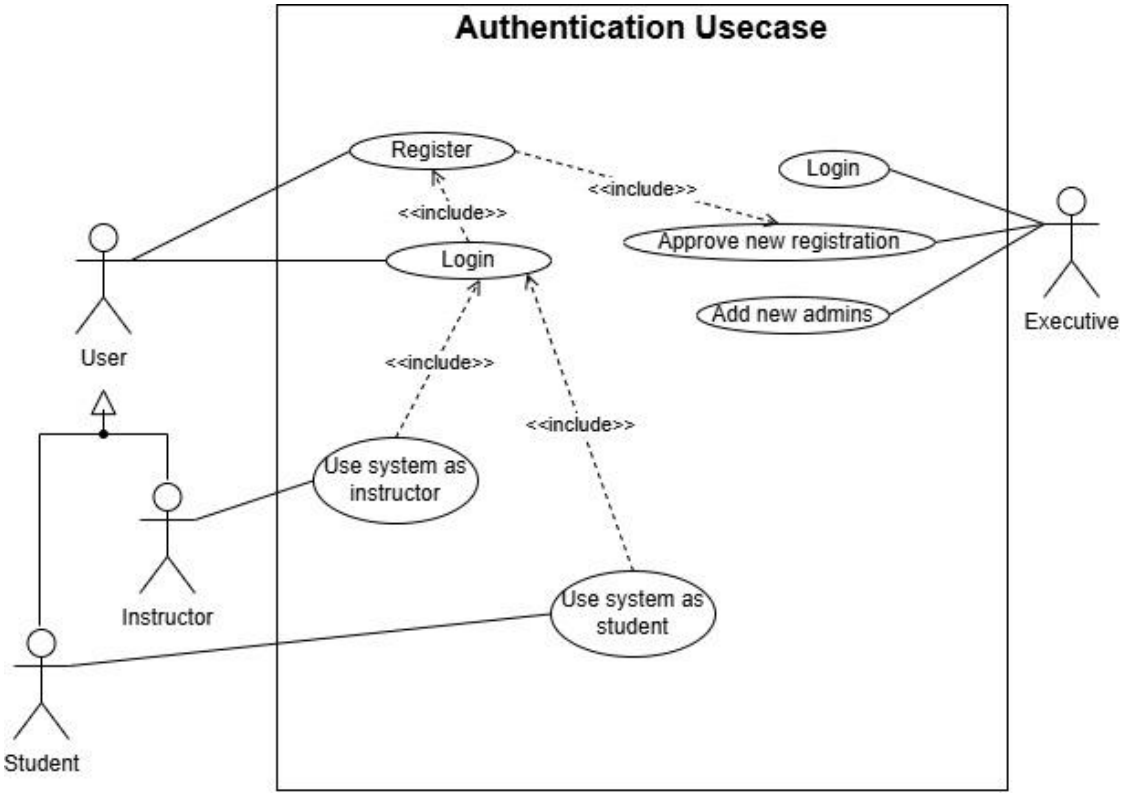


Fig. 2.2.1 Use Cases for Authentication from different user perspectives

2.2.2 Use Case Diagram for Executive(Admin)

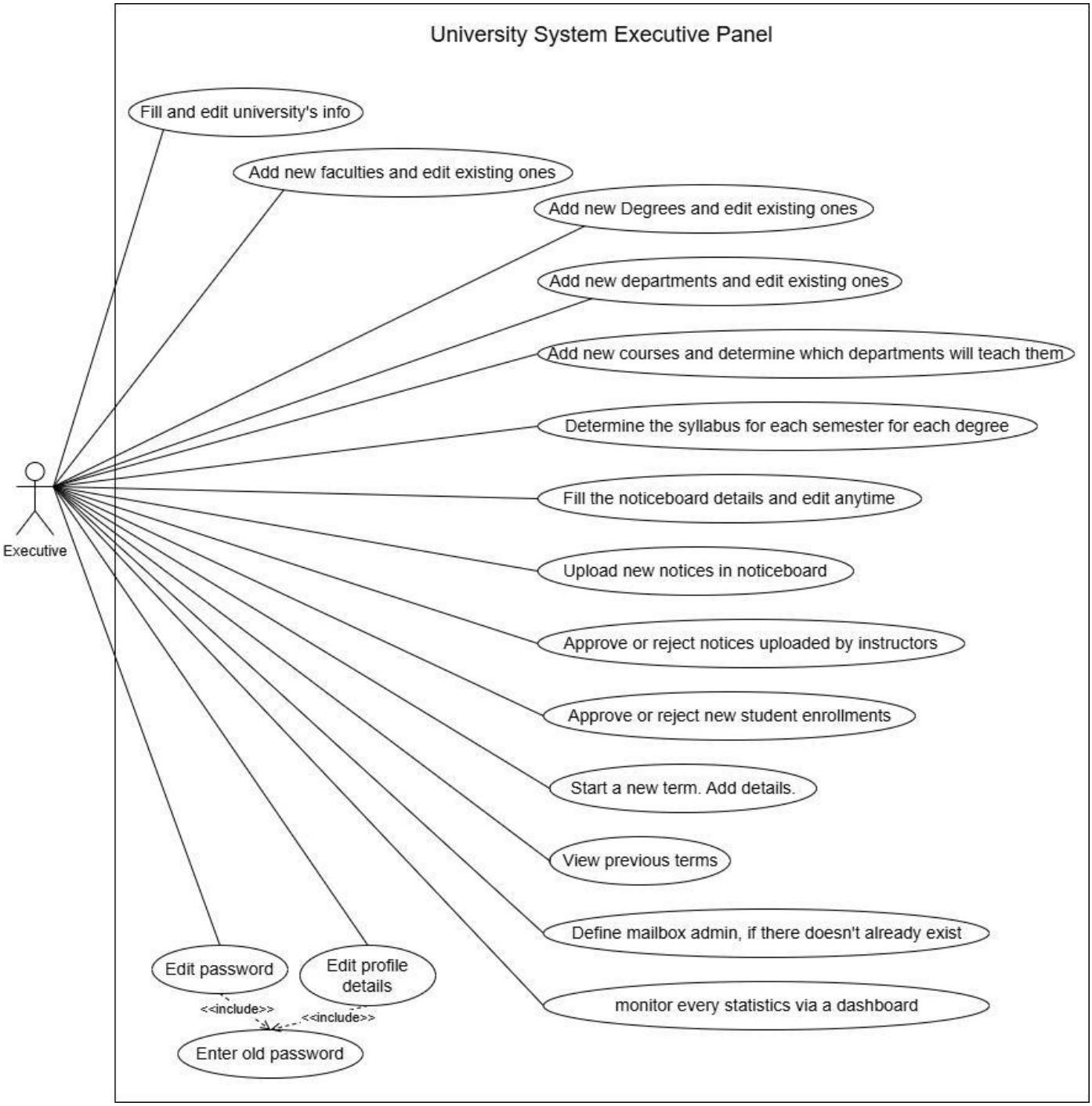


Fig. 2.2.2 Use Case Diagram for Executive (Admin)

2.2.3 Use Case Diagram for Instructor (Faculty)

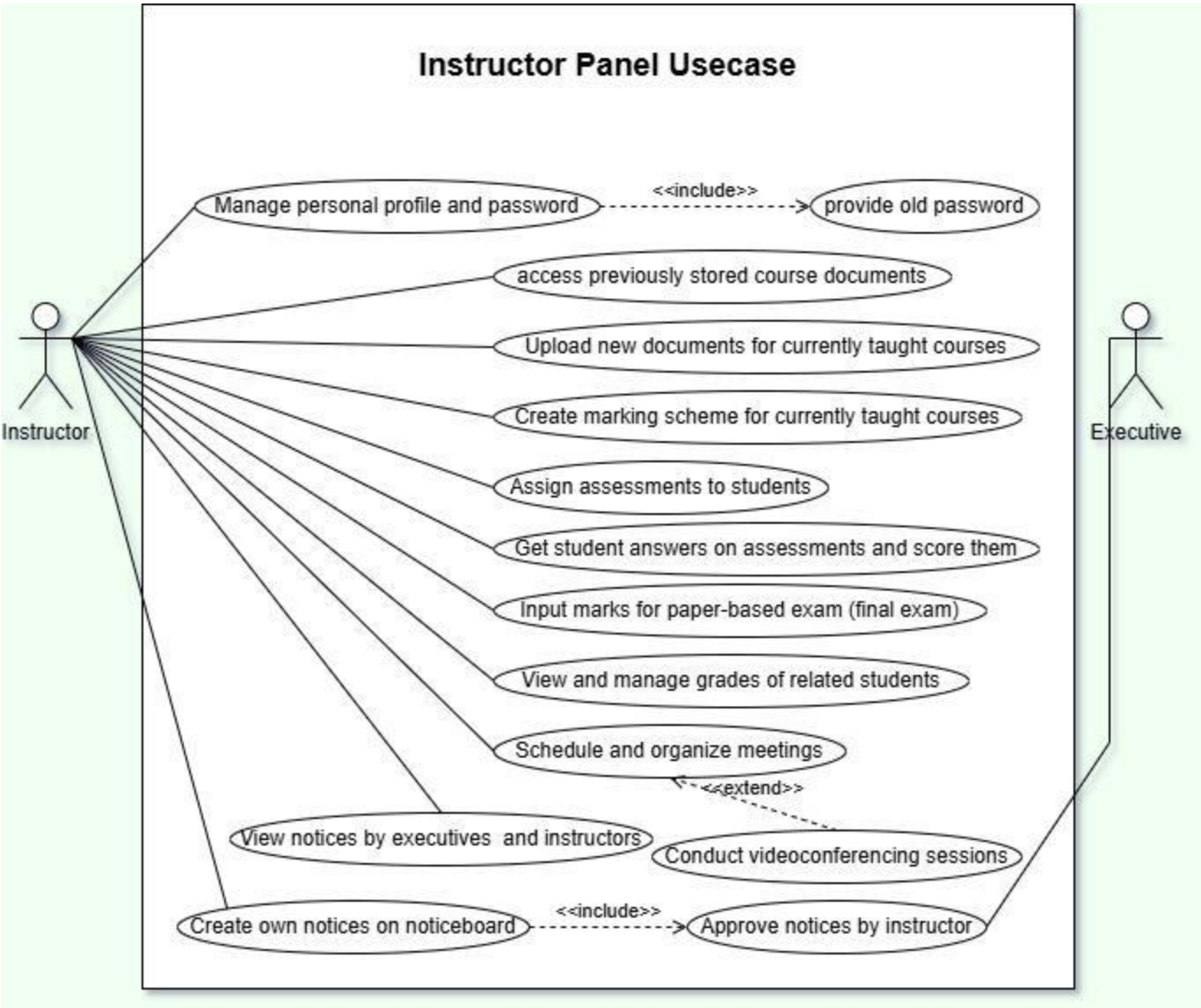


Fig. 2.2.3 Use Case Diagram for Instructor (Faculty)

2.2.4 Use Case Diagram for Student

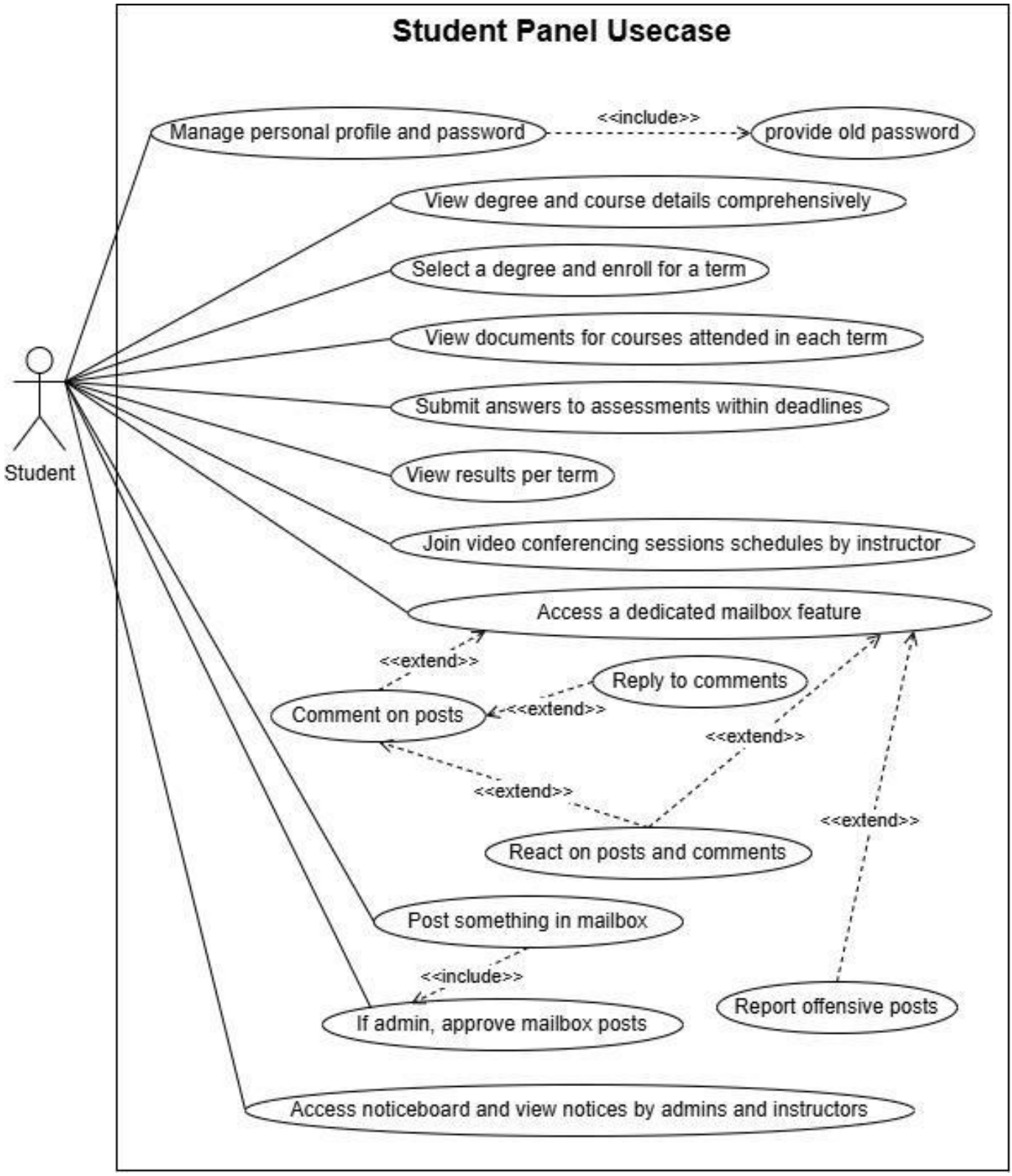


Fig. 2.2.4 Use Case Diagram for Student

2.3 System Sequence Overview

2.3.1 Executive Authentication Sequence

Initially, there will be a predefined executive/admin. This admin will be the **primary admin**. He can use the system after logging into the system. The detailed login process is described in the following sequence diagram. After logging in, the user can use the system as an executive/admin. He can add new admins as well. While adding, he will specify his details and set a **predefined one-time password**, after which the new admin can log in and change his password as desired.

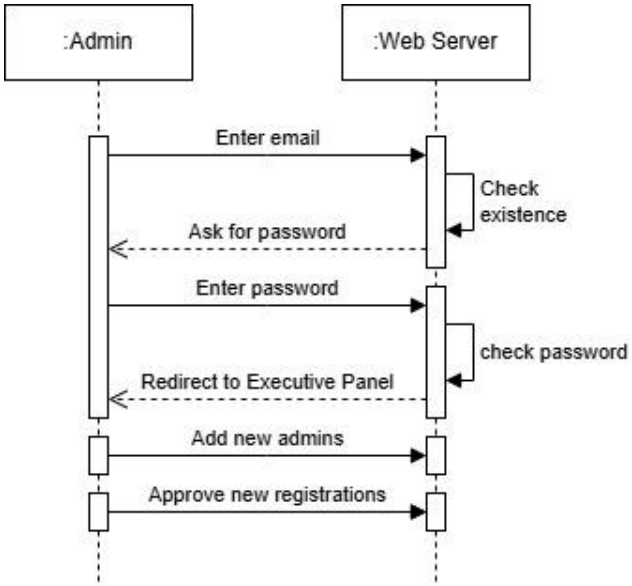


Fig. 2.2.4 Executive Authentication Sequence Diagram

2.3.2 Standard User Authentication Sequence

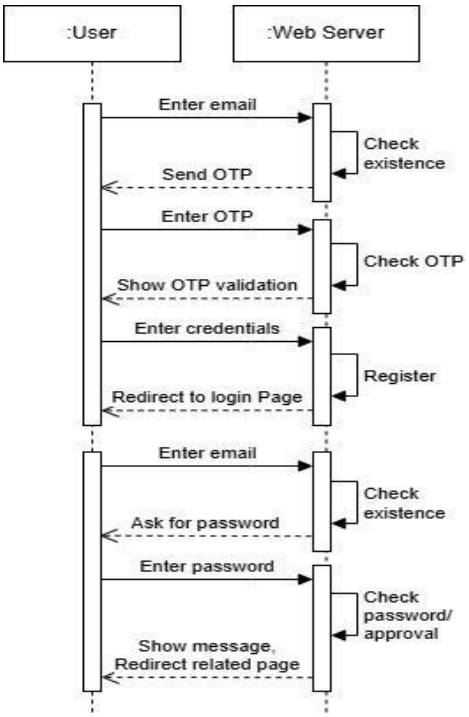


Fig. 2.2.4 Sequence Diagram for Authentication of Standard Users

2.3.4 Overall Academic Process

The academic process in the University Management System (UMS) involves multiple stakeholders—**executives**, **instructors**, and **students**—who must all be authenticated and logged in to access their respective functionalities. The process ensures structured academic operations, role-based responsibilities, and centralized approval mechanisms to maintain integrity and accuracy.

1. User Access and Approvals

Executives hold the **highest level of authority**. They approve registrations for students and instructors, manage course allocations, and oversee the entire academic structure.

Instructors register in the system but **require executive approval** before gaining access. While approving, the executive will assign a role (position in the department) and the department as well.

Students also **require executive approval** for registration and subsequent enrollment in each term.

2. University Structure

- **Faculties**: The university consists of multiple faculties, each headed by a Head of Faculty (HOF).
- **Departments**: Every faculty is divided into multiple departments, each managed by a **Head of Department (HOD).**
- **Course Allocation**: Each course belongs to a single department, and that department is solely responsible for managing it, regardless of the degree or semester it is included.

3. Degree Requirements and Academic Terms

Each degree program is divided into semesters, with a defined syllabus that specifies:

- Courses offered in the semester
- Credit hours for each course
- Academic terms and requirements

Each semester is structured around academic terms, which include:

- Start date of the term
- End date of the term
- Result announcement date

4. Academic Batches within a Term

A batch represents a ***specific group of students enrolled in a semester for a particular degree program***.

(e.g., *Bachelor of Science – Semester 1 (Batch A), Bachelor of Science – (Batch B), or similar batches from other degree programs)*

Multiple batches may exist in the same term, depending on the student enrollments and degree offerings.

5. Course Assignment to Instructors

For each batch, the **predefined courses** of the corresponding semester must be assigned to instructors. This responsibility lies **exclusively with the executive**. The assignment rule is strict:

- **Only instructors from the department that owns the course** can be assigned.
- Executives **cannot assign instructors from other departments** to courses that do not belong to them.
  This ensures that the subject expertise is maintained, and departmental authority over its courses is respected.

6. Instructor Capabilities

Once assigned courses by executives, instructors can manage academic delivery and evaluation. Their responsibilities include:

- Uploading and sharing course-related materials with enrolled students.
- Creating and managing assessments (assignments, quizzes, projects).
- Quizzes are automatically graded by the system.
- Assignments and projects require manual grading by the instructor.
- Scheduling and conducting online video conferencing sessions for teaching and interaction.

7. Student Enrollment and Learning Activities

Students must enroll each term, and the enrollment must be approved by an executive. During the enrollment process, if the batch includes multiple supporting, elective, or extracurricular subjects, the student must select only one option from each category. Once enrolled, students gain access to:

- The chosen subjects from the related batch
- Course materials and resources shared by instructors
- Assessments and quizzes
- Video conferencing sessions conducted by instructors

Students' academic performance is continuously tracked through the system and made available to both instructors and executives.

8. Instructor Review System

For every course in a batch, students can rate their instructor on a **5-point scale** and submit a **single written review**. Each student may review an instructor **only once per course per batch**.

All feedback is sent **privately to executives**, with **student identities hidden** to ensure privacy. Executives can view aggregated statistics like rating distributions and reviews to evaluate instructor performance.

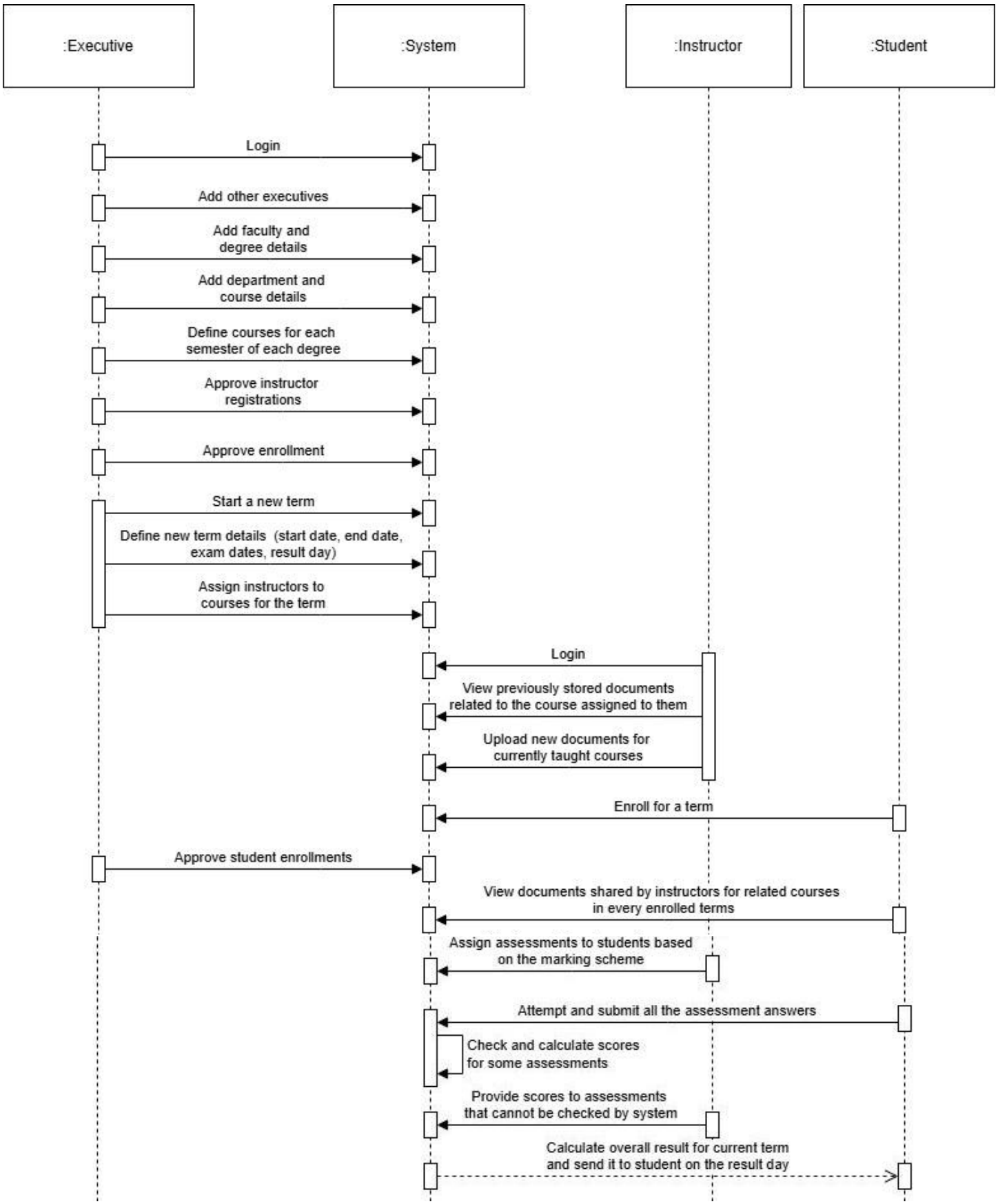
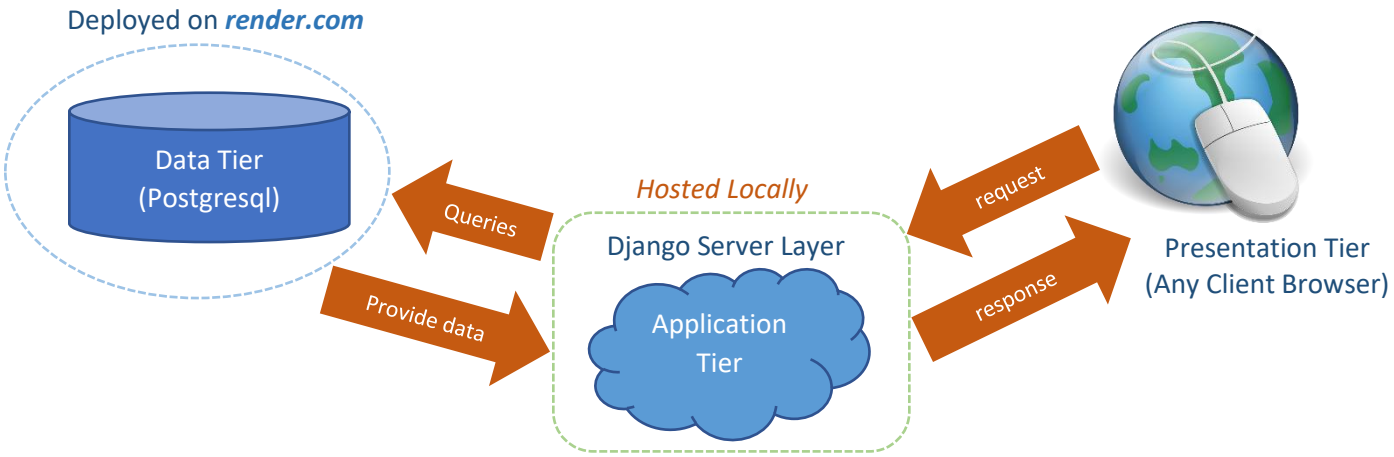


Fig. 2.2.4 Sequence Diagram for Overall Academic Process

2.4 Three Tier Architecture of the system



The Data Tier

The **Data Tier** forms the foundation of the **University Management System (UMS)**, responsible for **storing, managing, and securing** all institutional data. It defines the core **entities** that represent **academic, administrative, and communication processes**, along with their **relationships** and **constraints**. This tier also incorporates **triggers, rules, and integrity mechanisms** that ensure **accuracy, consistency, and reliability** of the data across modules. By organizing the **database** into well-structured entities and enforcing **business logic** at the data level, the Data Tier provides a stable backbone for the application and supports seamless interactions with higher tiers.

While the database of the UMS is designed for scalability and integrity, it also carries a significant level of complexity, containing a total of **31 entities**. Because representing all entities and their interconnections in a single ER diagram would make it difficult to interpret, the overall structure has been divided into **five focused modules**: the **User Module**, the **Academic Module**, the **Academic Operations Module**, the **Student Communication & Interaction Module**, and the **Noticeboard & University Profile Module**. This modular approach improves clarity, allows for better maintenance, and makes the logical design more understandable for both developers and stakeholder.

2.4.1 The User Module

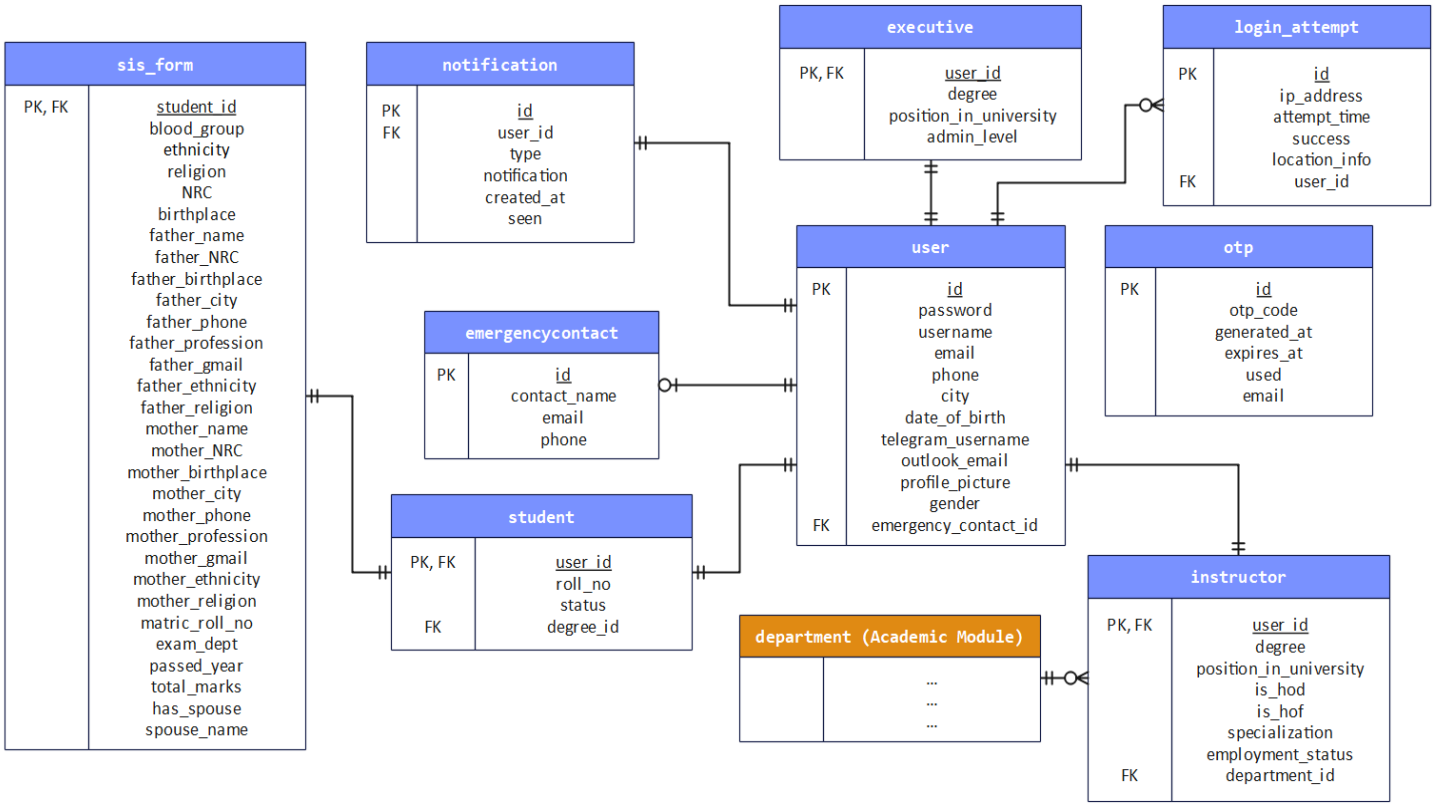


Fig. 2.4.1 Entity Relationship Model for User Module

This module outlines the user tier for a university management system, encapsulating the various entities and their interrelationships. The main entities include **Students, Instructors, Executives**, and users covering login attempts and notifications. Each student is associated with a **SIS** (Student Information System) form, capturing vital personal and family details. The **User** entity manages the authentication details, linking users to login attempts and **OTP** (One-Time Password) for secure access. An **Emergency Contact** entity allows students to record crucial contacts for emergencies, establishing a direct relation to the Student entity. **Notifications** ensure users receive alerts relevant to their respective roles within the system. This structured approach allows for robust data management and ensures seamless interaction across various facets of the university management system.

Notes:

1. ***Instructor*** *entity references* ***department*** *module which is defined in the next part,* ***Academic Module****.*
2. *The* ***notification*** *attribute in the* ***Notification*** *entity is implemented as a* ***JSONField****, allowing flexible storage of structured data.*

   Sample JSON structure

```
{
"text": "notification text",
"destination": "the destination where the notification will lead to when clicked"
}
```

2.4.2 The Academic Module

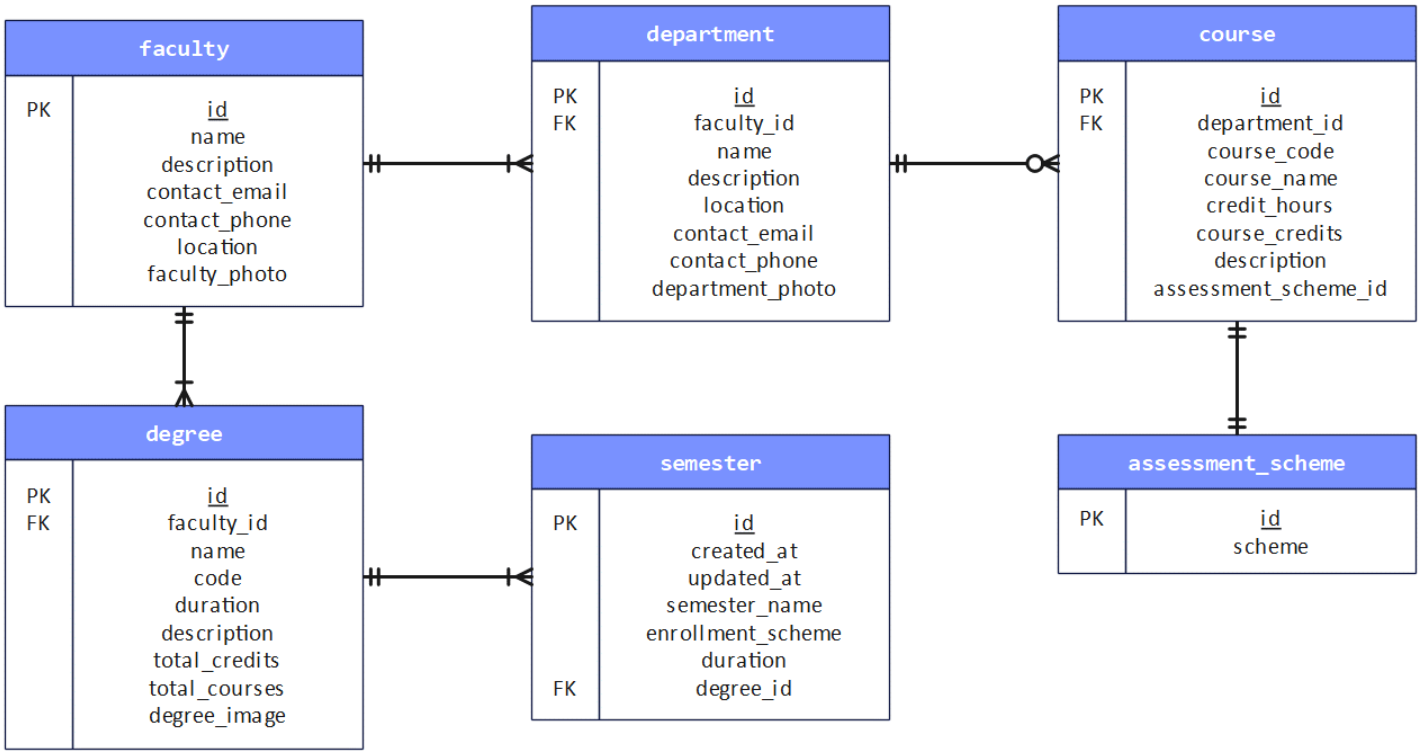


Fig. 2.4.2 Entity Relationship Model for Academic Module

This module represents an entity-relationship model for an academic database designed to manage various facets of a university's educational structure. The central entities include **faculty, department, course, degree, semester,** and **assessment scheme**, each with its unique attributes and relationships.

The **faculty** entity contains essential details about faculty members, such as their identification, name, description, and contact information. Each faculty member can oversee one or more **departments**, identified by a foreign key relationship. Similarly, each **department** is characterized by its name, description, and location and may also include associated faculty details.

The **course** entity represents the academic courses offered within departments, encapsulating attributes like course code, credit hours, and assessment schemes. Each course is linked to a department through a foreign key, indicating its organizational association. The **assessment_scheme** entity delineates the evaluation methods employed for courses, highlighting its role in academic assessment.

The **degree** entity is affiliated with specific faculties and outlines the various academic programs available, detailing aspects such as name, code, duration, and total credits required. Each degree can include multiple associated semesters, further defining the structure of academic timelines. The **semester** entity records critical information regarding semester timelines, including creation and update dates, name, and duration linked to the degrees.

The `scheme` attribute in the **AssessmentScheme** entity is defined as a **JSONField** to store flexible evaluation structures for courses. Instead of maintaining rigid columns for every type of assessment, the JSON structure maps each assessment component to its corresponding weight. For example:

```
{
        "Quiz": 5.0,
        "Lab Test": 10.0,
        "Tutorial": 10.0,
        "Assignment": 10.0,
        "Final(On Paper)": 60.0,
        "Class Participation": 5.0
}
```

This design makes it easier to support diverse grading policies across courses, as new components can be added or existing ones adjusted without modifying the database schema.

The `syllabus_structure` attribute in the **Semester** entity is also a **JSONField**, used to represent the list of courses offered in a semester along with their classification. Each element in the JSON array contains course metadata such as type, code, name, hours, and credits. For example:

```
[
  {
    "type": "Core",
    "course_code": "CS 109",
    "course_name": "Reinforcement Learning",
    "course_hours": 4,
    "course_credits": 4
  },
  etc.
]
```

This structure provides flexibility in representing semester curricula, supporting multiple categories (e.g., Core, Elective, Supporting) and allowing easy updates when course offerings change.

This comprehensive structure ensures proper organization of academic data, facilitating seamless management and efficient retrieval of information within the university system.

2.4.3 The Academic Operations Module

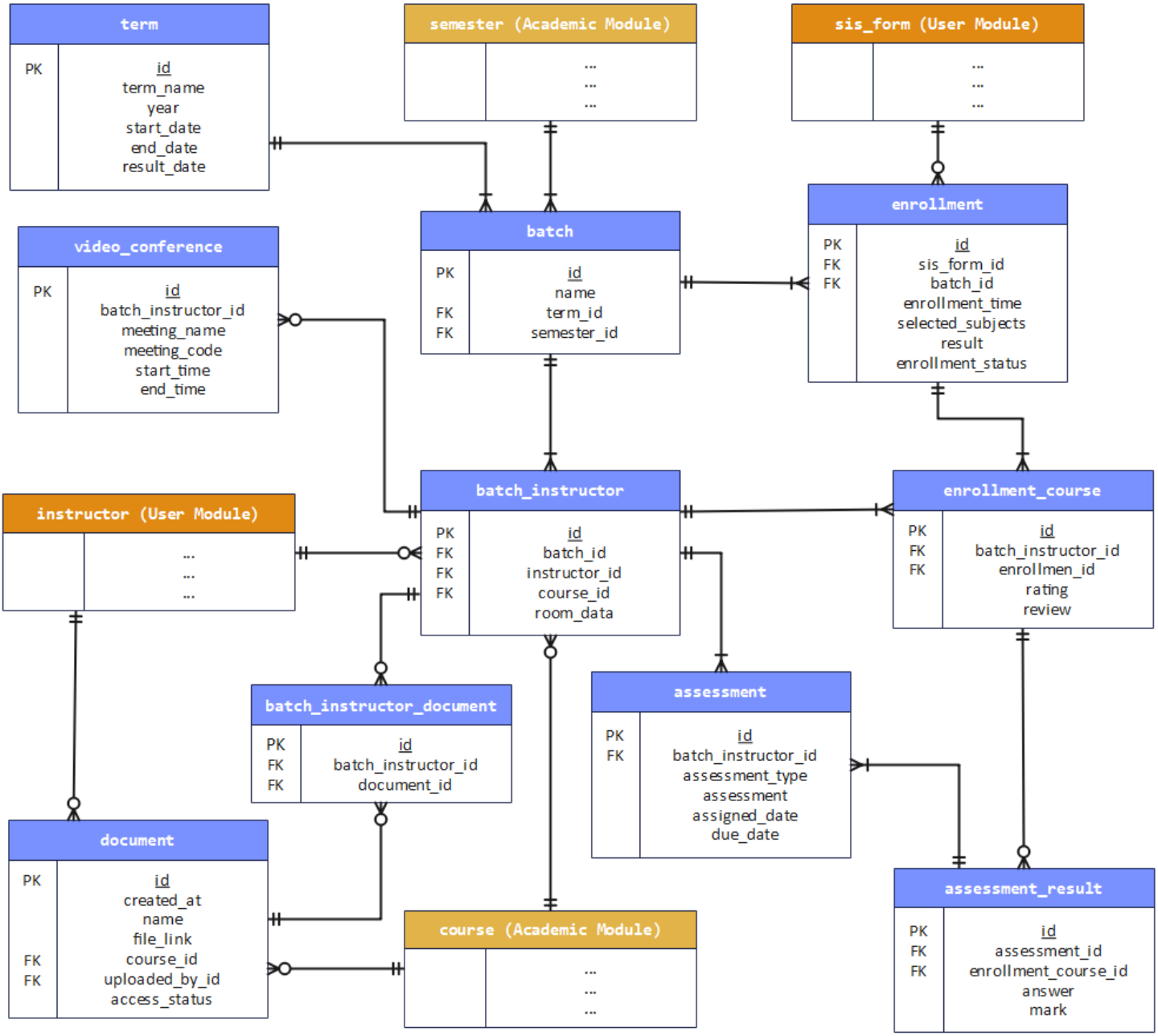


Fig. 2.4.3 Entity Relationship Model for Academic Operations Module

This module models the **core academic and user interactions** within the University Management System. At its heart are entities such as **batch, term, semester, and course**, which together define the academic structure and scheduling of the institution. Each **batch** is linked to a specific **term** and **semester**, and is managed by one or more **instructors** through the **batch_instructor** relationship. The **batch_instructor** entity also connects to **course**, indicating which instructor is responsible for which course within a batch, and is further associated with **document** resources and **video_conference** sessions for online learning support.

Student participation is managed through the **sis_form** (student information system form), which connects to **enrollment** records. Each **enrollment** ties a **student** to a **batch** and tracks their selected subjects, results, and enrollment status. The **enrollment_course** entity links a student's **enrollment** to specific batch-instructor-course combinations, allowing for individualized ratings and reviews. Academic progress is tracked through **assessment** and **assessment_result** entities, where assessments are assigned by instructors to batches and

results are recorded per student per course. The model also supports **document** management and sharing, as well as integration with video conferencing for remote instruction.

The `result` attribute in the **Enrollment** entity is implemented as a **JSONField** that stores detailed academic outcomes for each course a student has enrolled in. For every enrolled course, the JSON structure captures the course name, course code, credit hours, and the student's performance in terms of letter grade, grade score, and weighted grade points. This structure makes it possible to calculate cumulative results and GPA directly from the enrollment record. An example entry would look like:

```
{
  "course_name": "Database Systems",
  "course_code": "CS 201",
  "credits": 3,
  "letter_grade": "A",
  "grade_score": 4.0,
  "grade_point": 12.0
}
```

The `selected_subjects` attribute in the **Enrollment** entity is another **JSONField**, used to record the list of courses chosen by a student during registration. Instead of storing multiple rows for each selection, the JSON structure consolidates all chosen courses into a single attribute, represented by their IDs. An example structure is:

```
{
  "ids": [3, 4, 5, …]
}
```

The `room_data` attribute in the **BatchInstructor** entity is modeled as a **JSONField** to record classroom allocations and teaching frequency. Each key-value pair maps a course to the assigned room and the number of sessions per week in that room. For example:

```
{
  "room1": "F-102",
  "times1": 3,
  "room2": "Theatre B",
  "times2": 2
}
```

The `assessment` attribute in the **Assessment** entity and the answer attribute in the **AssessmentResult** entity are both **JSONFields** whose structure varies depending on the type of assessment. For instance, an online quiz assessment may store questions, choices, and correct answers, while a written exam may store descriptive question prompts. Correspondingly, the answer attribute in **AssessmentResult** holds the student's responses in a matching JSON format. This flexible design allows the system to support multiple assessment types (quizzes, assignments, lab tests, exams, etc.) within the same framework, ensuring adaptability to diverse evaluation methods.

Overall, this ER module provides a comprehensive and normalized structure for managing academic operations, instructor assignments, student enrollments, assessments, and resource sharing. It ensures that all relationships between students, instructors, courses, and administrative processes are clearly defined, supporting robust reporting, analytics, and workflow automation within the university system

2.4.4 Student Communication/Interaction Module

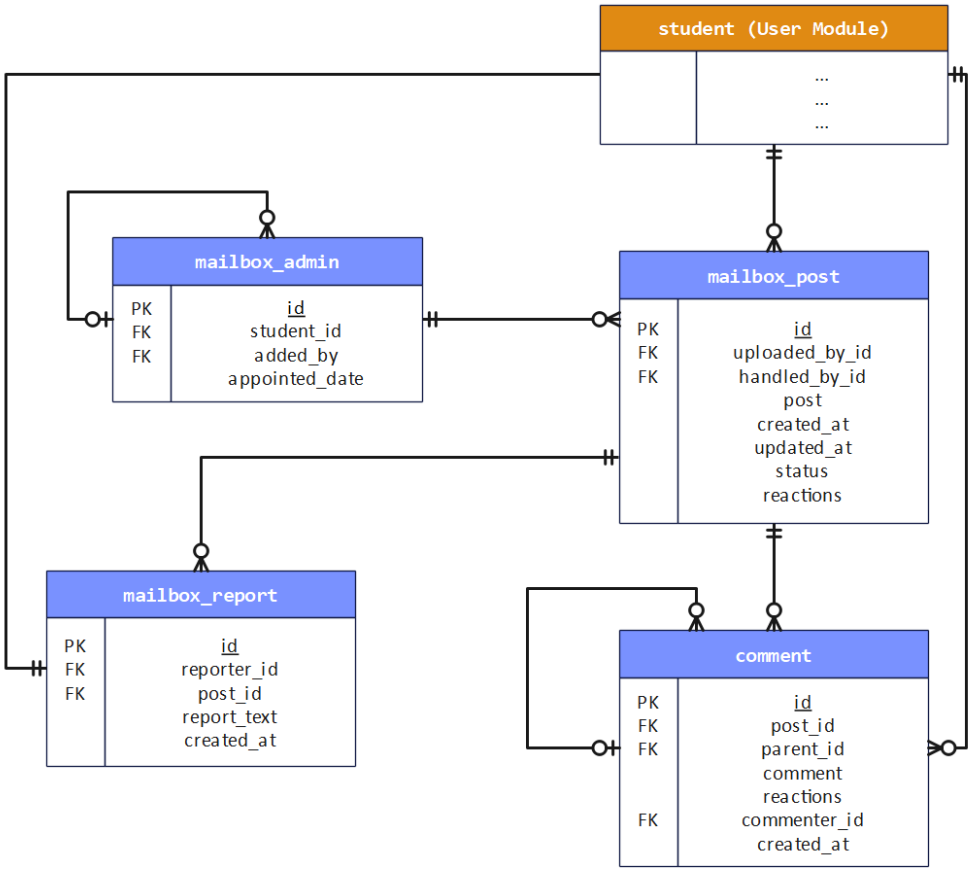


Fig. 2.4.4 Entity Relationship Model for Student Communication/Interaction Module

This module represents the relational structure of a **student communication/interaction management system**, where various entities are interconnected. Central to the design are the tables for **students**, **mailbox admins**, **posts**, **reports**, and **comments**. Each entity serves a specific purpose, with foreign keys (FK) indicating relationships between them. For instance, the **mailbox_admin** table associates admins with specific **students**, while **mailbox_post** connects posts to the students who created and handled them.

The **mailbox_report** table stores information about reports made by students regarding specific posts, including a reference to the reporter and detailed report text. This table highlights the importance of feedback within the system, facilitating an organized way to address any issues. The comment table further enriches interactions by allowing users to comment on posts, maintaining connections between comments and their respective posts through foreign keys.

Overall, this design fosters a comprehensive structure for managing student interactions, oversight by admins, and the reporting of content within the mailbox system, ensuring efficient communication and administrative control.

2.4.5 Noticeboard and University Profile Module

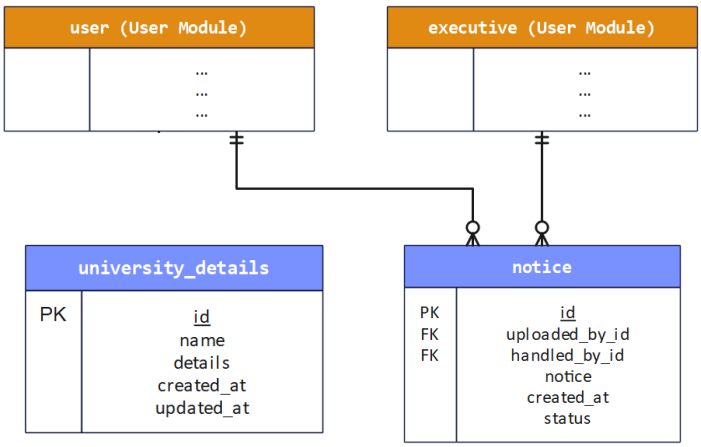


Fig. 2.4.4 Entity Relationship Model for Noticeboard and University Profile Module

This module integrates two critical entities: **university_details** and notice, each serving a distinct role in institutional communication and information management. The **university_details** table stores fundamental data about the university—such as its name, general description, and relevant timestamps—providing a central reference for institutional identity. The notice table, on the other hand, functions as the core of the university's internal noticeboard system. It records each notice's content, the **user** who uploaded it, the **executive** who processed or approved it, creation timestamps, and current status.

The module relies on two primary user roles—**User Module** **(**user**)** and **Executive Module** **(**executive**)**—to manage notice creation and approval workflows. These roles are linked to notices through foreign key relationships, ensuring that each notice has clear accountability for who created and who reviewed or approved it. This design supports a transparent and traceable notice management process, reinforcing institutional communication and governance.

2.5 Transactions and Triggers

In this system, all transaction management is handled at the server layer using Django's built-in capabilities. With **transaction.atomic()**, critical operations are guaranteed to either succeed fully or roll back entirely, ensuring data integrity and preventing corruption. Centralizing this logic at the application level provides flexibility, easier debugging, and consistency across the platform without relying on database-specific triggers.

Triggers are also managed at the server layer through Django's signal framework. Instead of database triggers, signals listen for key events—such as user logins, student registrations, or new term creation—and automatically generate relevant notifications. This approach keeps triggers closely aligned with business logic, while allowing them to be easily extended as system needs evolve.

A wide range of notification events are supported: students and instructors are alerted about new notices, assessments, and reference materials; admins are notified of instructor requests; and students receive enrollment updates in real time. By unifying both transactions and triggers in the application layer, the system achieves robustness, maintainability, and flexibility while ensuring users remain consistently informed.

VOLUME III

DEVELOPMENT & IMPLEMENTATION

(Tutorial Volume For Future Collaborators)

3.1 Visual Overview

The University Management System is a modular Django platform that digitalizes various university operations. It provides **personalized dashboards** for students, faculty, and executives, manages the **academic structure** and **assessment processes**, and ensures smooth **communication** through a mailbox and notification system, all while centralizing **university configurations** and enforcing robust **access control** via custom middleware.

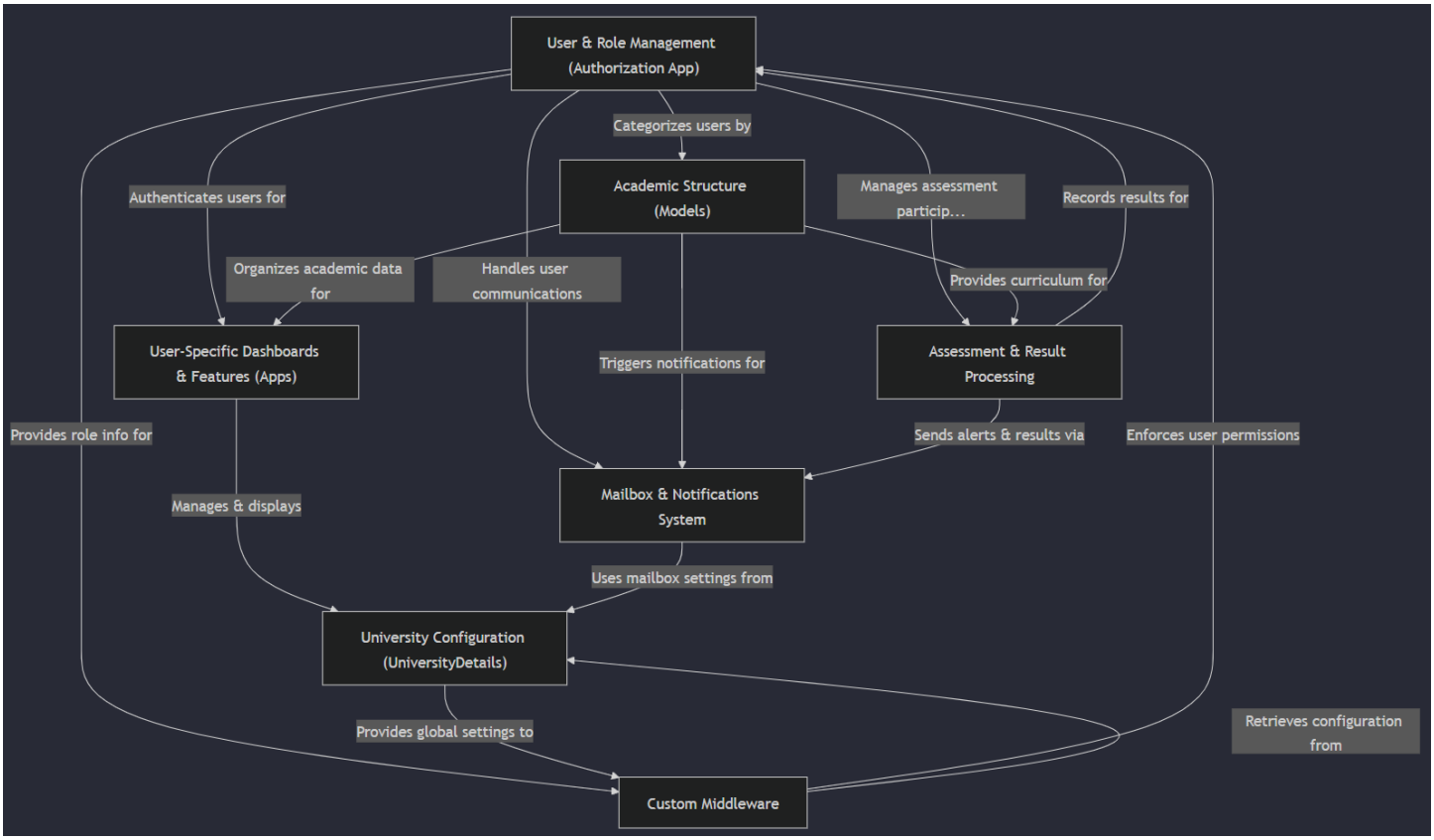


Fig. 2.2.4 Visual Overview of UMS

3.2 University Configuration (University Details)

Over time, university information naturally changes. The mission statement may be revised, new partnerships may be established, or the official contact email may be updated. But should such updates require calling a developer, editing code, and redeploying the entire system? Certainly not — that would be unnecessary overhead for simple content updates.

This is where the **University Configuration**, powered by the **UniversityDetails model**, comes in. Think of it as the university's **central settings panel**, a dynamic whiteboard where important institutional details are stored and managed. Instead of embedding this information directly into the website's code, it is maintained in a structured format that executives (or other authorized executives) can easily update without touching any code.

Goal of this Part

By the end, you will understand how the university's core information (such as its name, logo, and mission) is stored in the system and how it can be updated seamlessly.

3.2.1 What is University Configuration?

At its core, **UniversityDetails** is a flexible data record designed to hold high-level information about the entire institution. It consists of two key parts:

1. `name` Field (Identifier)
    a. A short, unique label that identifies the category of information
    b. Examples

Key Name	Detail Type
university_info	General details
labs	Labs, their details and related projects
photos	University public galleries
certificates	Certificates got by the university

    Each of these categories is represented by a separate UniversityDetails record.
2. `details` Field (The Data Store)
    a. A JSONField that holds the actual information in a structured dictionary-like format.
    b. The flexibility allows one record to store diverse information such as the university's full name, address, website, mission, vision, and more (all neatly organized)

How it Solves the Problem

Let's look at the most common use case: an executive wants to update the university's official name, contact details, or logo.

1. Retrieve Current Information

- When someone visits the university's public website, the system automatically fetches the latest **UniversityDetails** record.
- For example, in public/views.py, the system reads this data and displays it dynamically on the website.

2. Update Information

- Instead of editing code, the executive simply updates the relevant fields (e.g., name, logo, email) through the system's admin panel or configuration page.
- The changes are saved in the database and immediately reflected across the website.

This approach ensures that critical university information remains accurate, flexible, and easily updatable, without the need for redeployment or developer intervention.

```
def show_public_page(request):
    # Find the UniversityDetails record named 'university_info'
    university_info = UniversityDetails.objects.filter(
        name='university_info',
    ).first().details # Get the 'details' (JSON) part

    # ... other data fetching ...

    data = {
        'university_info': university_info,
        # ... other data ...
    }
    return render(request, 'htmls/home.html', context=data)
```

- `UniversityDetails.objects.filter(name='university_info')`: This line asks the database: “Give me the `UniversityDetails` record that has its `name` field set to 'university_info'.”
- `.first()`: We expect only one 'university_info' record, so we grab the first one.
- `.details`: We then extract the actual `JSONField` content, which will be a Python dictionary containing all the university's specific data (name, address, etc.).
- This dictionary is then passed to the website's template to display the information.

3.2.2 Executives Updating the Information

Now, let's look at how an executive would change these details using an admin interface.

An executive would go to a special page, let's say `/executives/uni_info/`, which displays a form with all the current university details. They would type in the new name, update the address, perhaps upload a new logo, and then click “**Save**”.
Here's a simplified look at what happens in the ***executives/views.py*** when they save:

```
def uni_info_edit(request):
    if request.method != “POST”:
        return JsonResponse({“success”: False})

    # 1. Get the data from the form
    data_from_form = {
        “name”: request.POST.get(“name”, ““),
        “address”: request.POST.get(“address”, ““),
        # ... other fields like email, website, mission, vision ...
    }

    # 2. Find the 'university_info' record, or create it if it doesn't exist
    university_info_obj, created = UniversityDetails.objects.get_or_create(name='university_info')

    # 3. Handle logo upload (if a new file was provided)
    if profile_file := request.FILES.get(“profile”): # 'profile' is the logo file input name
```

```
        # Delete old logo if it exists (good practice!)
        old_profile_path = university_info_obj.details.get("profile_url")
        if old_profile_path:
            default_storage.delete(old_profile_path)
        # Save the new logo file and get its path
        logo_path = default_storage.save(profile_file.name, profile_file)
        data_from_form['profile_url'] = logo_path # Store the path in our details
    else:
        # No new logo, keep the old one
        data_from_form['profile_url'] = university_info_obj.details.get("profile_url")

    # 4. Update the 'details' JSONField with the new data
    university_info_obj.details = data_from_form
    university_info_obj.save() # Save the changes to the database

    return JsonResponse({'success': True})
```

- ***UniversityDetails.objects.get_or_create(name='university_info')***: This is a very handy function. It tries to *get* the record with ***name='university_info'***. If it doesn't find one, it *create*s it automatically.
- ***request.POST.get(...)***: This collects all the text and values entered into the form fields.
- ***request.FILES.get("profile")***: This gets the actual image file that was uploaded.
- ***default_storage.save(...)***: This is a Django way to save files (like images) to your project's media folder. It returns the path to the saved file.
- ***university_info_obj.details = data_from_form***: This is the crucial step! We replace the entire *details* dictionary with our newly updated ***data_from_form*** dictionary.
- ***university_info_obj.save()***: This commits the changes to the database.

Now, the next time anyone visits the public page, they will see the updated university name, address, and new logo! This entire process didn't require a developer to write any new code or redeploy the application.

3.2.3 Under the Hood: A Step-by-Step Look

The core idea is that we only have a few UniversityDetails records (one for general info, one for labs, one for photos, etc.), and inside each record's details field, we store all the specific configuration in a flexible JSON format.

```
class UniversityDetails(models.Model):
    name = models.CharField(max_length=100, default='')
    details = models.JSONField(default=dict)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
```

- *`name = models.CharField(...)`*: This is a standard text field for our unique identifier (e.g., 'university_info').
- *`details = models.JSONField(default=dict)`*: This is the star of the show. It allows us to store a Python dictionary (which gets converted to JSON in the database) directly. This means we don't need to create separate database columns for every single piece of university information. If we need to add a “fax number” later, we just update the details `dictionary`; no database table changes are needed! This makes it incredibly flexible.

3.2.4 Other Examples of *`UniversityDetails`* in Action

The same pattern is used for other dynamic information:

- **Labs:** The system stores information about different research labs. Each lab's details (name, description, projects, members, photos) are stored within the details JSON of a `UniversityDetails` record where `name='labs'`. You can see this in `executives/lab_api.py and executives/additional_business_logics/data_formatter.py` (e.g., `get_labs_obj`, `get_lab_details_data`).
- **Partnerships & Certificates:** Similar to labs, these are stored in UniversityDetails records with `name='partnerships'` and `name='certificates'`, respectively. Their specific data is inside the details JSON.
- **Photos:** Public photos and their captions are managed in a `UniversityDetails` record named 'photos'.

This single, flexible model allows the system to manage a wide range of configuration data without needing a new database table for every single category. It's a highly maintainable and administrator-friendly approach.

In this part, we've explored the `UniversityDetails` model, understanding its role as the central hub for dynamic, high-level university information. We saw how it uses a simple `name` field to identify different categories of information and a flexible `JSONField (details)` to store the actual data. This design allows administrators to easily update public-facing information without requiring any code changes, making the system much more agile.
Next, we'll dive into the core building blocks of the university's academic structure, learning about how Faculties, Departments, Degrees, and Courses are defined and organized.

3.3 Academic Structure (Models)

A university is much more than just its details; it's a place where students learn, and instructors teach specific subjects organized into programs.

How do we represent this educational framework? How does a student know what degree they are pursuing, what courses are available, and when they are offered? This is where the **Academic Structure** comes in.
Our Goal for this Chapter: By the end, you'll understand how the entire academic landscape of the university — from broad colleges to individual subjects and academic terms — is defined and organized within the UMS system.

3.3.1 What is Academic Structure?

Think of the academic structure as the blueprint of the university's educational offerings. It defines all the building blocks that make up a student's journey, from picking a major to attending classes in a specific semester.

The UMS project organizes this through several interconnected **models**, which are like specialized data tables in our database, each holding a piece of the academic puzzle.
Let's break down these key academic building blocks:

Model Name	Analogy	Description
Faculty	**Faculties** (e.g., Faculty of Computer Science)	Broad academic units usually covering several related departments.
Department	**Departments under faculties** (e.g., Department of Artificial Intelligence)	More focused units within a Faculty, managing specific areas of study.
Degree	**Study Programs** (e.g., Bachelor of Science Knowledge Engineering)	Defines what a student studies, including its duration and overall credits.
Course	**Individual Subjects** (e.g., Introduction to Python)	Represents a single subject with a code, name, credits, and marking scheme.
Semester	**Parts of a Degree** (e.g., Semester 1 of a BSc)	Organizes Courses within a Degree, often outlining which courses are taken in a specific period.
Term	**Academic Year Period** (e.g., Fall 2024)	Defines a broader academic period with start, end, and result dates, used across all degrees.
Batch	**Group of Students within a Degree and Term** (e.g., *BSc Knowledge Engineering – Semester 1, Batch A*)	Represents a specific cohort of students enrolled in a degree program for a given semester and term. Each batch is linked to predefined courses of that semester, and executives assign the courses to instructors from the appropriate departments.

These models work together to create a clear, hierarchical structure for the university's education system.

3.3.2 Central Use Case: Setting Up a New Academic Program

Imagine the university wants to introduce a brand-new academic program, such as a "*Bachelor of Data Science.*" An executive (an admin user) needs to add this program, specify which Faculty it belongs to, list its departments, define the courses, and structure them into semesters.

Here's how our academic models help solve this:

- **Define the** Faculty**:** If a new Faculty is needed (*e.g., "Faculty of Applied Sciences"*), it's created first.
- **Define** Department**s:** Create the *"Department of Artificial Intelligence"* within the *"Faculty of Computer Science."*
- **Define the** Degree**:** Create the *"Bachelor of Science (Knowledge Engineering)"* and link it to the *"Faculty of Computer Science."*
- **Define** Course**s:** Add courses like "*Artificial Intelligence (A modern approach)*", "*Machine Learning Basics,*" etc., linking each to the "*Department of Artificial Intelligence*"

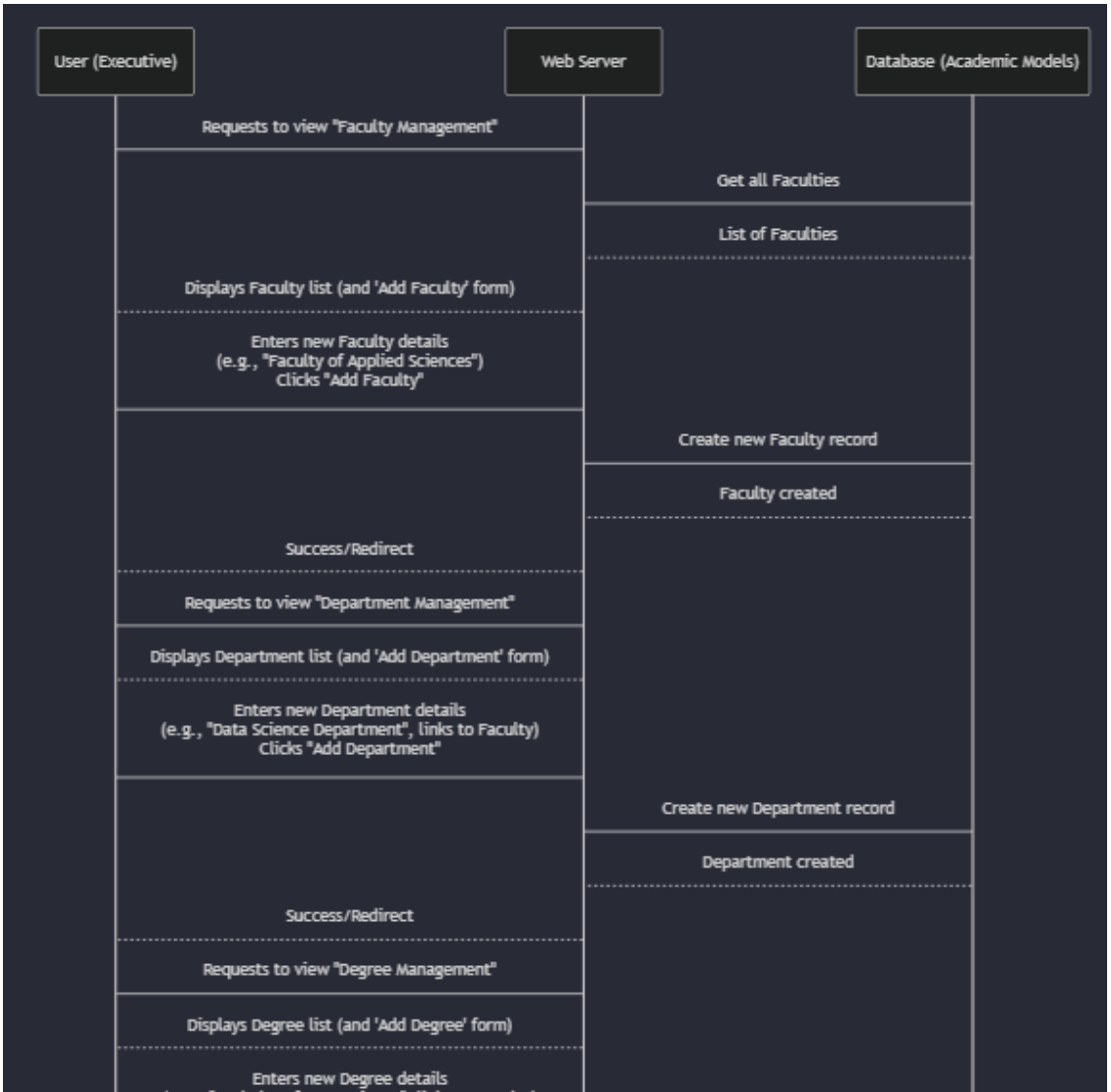
- **Define** Semesters**:** For the “Bachelor of Data Science,” create “Semester 1,” “Semester 2,” and so on, listing which Courses belong to each Semester.
- **Define** Terms**:** Executives **also define Terms** like “*Spring 2025*” or “*Fall 2025*” to mark specific academic periods. These terms are then used to organize Batches of students.
- **Define** Batches: Executives create Batches within a specific Term and Semester (e.g., *BSc Knowledge Engineering – Semester 1, Batch A*). Each Batch represents a group of students enrolled for that period, and executives assign the semester’s predefined courses to instructors from the respective departments.

This structured approach ensures that all academic information is organized logically and can be easily managed and retrieved.

3.3.3 Under the Hood: Building the Academic Structure

Let's see a simplified flow of how an executive might set up a new academic program like a “*Bachelor of Science (Knowledge Engineering)”* using these models.

The following diagram shows that the executive interacts with different “management” pages, and each interaction results in the creation or update of specific academic model records in the database.



3.3.4 Diving into the Code: Academic Models

Let's look at the actual model definitions in authorization/models.py. These define the blueprint for how each piece of academic data is stored.

1. Faculty

The **Faculty** model represents the largest academic divisions, like a college.

```
class Faculty(BaseModel):

    id = models.BigAutoField(auto_created=True, primary_key=True, serialize=False, default=1)

    name = models.CharField(max_length=100, default='')

    description = models.TextField(default='')

    contact_email = models.EmailField(default='')

    # ... other fields like head_of_faculty, faculty_photo ...
```

- **name**: The name of the faculty (e.g., "Faculty of Engineering").
- **description**: A longer text explaining what the faculty is about.
- **id**: A unique number that identifies each faculty.

2. *Department*

A **Department** belongs to a Faculty and represents a more specialized area of study.

```
# authorization/models.py (simplified Department model)

class Department(BaseModel):

    name = models.CharField(max_length=100, default='')

    faculty = models.ForeignKey(Faculty, on_delete=models.CASCADE) # Links to a Faculty

    description = models.TextField(default='')

    # ... other fields like location, head_of_department, department_photo ...
```

- **name**: The name of the department (e.g., "Computer Science Department").
- **faculty**: This is a crucial link! It connects the department to its parent **Faculty** using a **ForeignKey. on_delete=models.CASCADE** means if the parent Faculty is deleted, all its **Departments** are also deleted.

Example: Adding a Department

Here's how an executive could add a department in **executives/components/department_manager.py**:

```
def department_add(request):

    data = request.POST # Or json.loads(request.body)

    dept = Department()

    dept.name = data.get("name", "")

    dept.faculty = Faculty.objects.get(pk=data.get("faculty_id")) # Get the Faculty object

    dept.description = data.get("description", "")

    # ... set other fields like contact_email, photo ...

    dept.save()

    return JsonResponse({"success": True})
```

- This code receives department details from a form.
- It finds the **Faculty** based on the **faculty_id** provided.
- It then creates and saves a new **Department** record, linking it to the **Faculty**.

3. *Degree*

The **Degree** model defines a specific academic program a student can pursue.

```
# authorization/models.py (simplified Degree model)

class Degree(BaseModel):

    name = models.CharField(max_length=100, default='')

    code = models.CharField(max_length=20, default='')

    faculty = models.ForeignKey(Faculty, on_delete=models.CASCADE) # Links to a Faculty

    duration = models.PositiveIntegerField(default=4) # duration in years

    description = models.TextField(default='')

    degree_type = models.CharField(max_length=1, choices=DegreeType.choices, default=DegreeType.BACHELORS)

    # ... other fields like total_credits, total_courses, total_hours, degree_image ...
```

- **name**: The full name of the degree (e.g., "*Bachelor of Science in Knowledge Engineering*").
- **code**: A short code for the degree (*e.g., "B.C.Sc (KE)"*).
- **faculty**: Links the degree to the **Faculty** that offers it.
- **duration**: How many years the degree program typically takes.

Example: Adding a Degree

An executive can add a degree using **executives/components/degree_manager.py**:

```
def add_degree_api(request):

    data = request.POST # Or json.loads(request.body)

    faculty = Faculty.objects.get(id=data['faculty']) # Find the associated Faculty


    degree = Degree(

        name=data['name'],

        code=data['code'],

        faculty=faculty, # Link the Degree to the Faculty

        description=data.get('description', ''),

        duration=data.get('duration', 4),

        # ... other fields ...

    )

    degree.save()


    # Semesters are often created right after the degree

    semesters_data = json.loads(data.get('semesters', '[]'))
```

```
    for sem in semesters_data:
        Semester.objects.create(
            semester_name=sem['semester_name'],
            degree=degree, # Link the Semester to the newly created Degree
            duration_weeks=sem.get('duration_weeks', 16),
            syllabus_structure=sem.get('syllabus_structure', []), # A JSONField for course details
        )
    return JsonResponse({'success': True, 'degree_id': degree.id})
```

- This code creates a **Degree** record and then iterates through the provided **semesters_data** to create multiple **Semester** records, linking each back to the degree.

4. *Course*

A **Course** is an individual subject offered by a **Department**.

```
# authorization/models.py (simplified Course model)
class Course(BaseModel):
    course_code = models.CharField(max_length=20, unique=True, default='')
    course_name = models.CharField(max_length=100, default='')
    department = models.ForeignKey(Department, on_delete=models.CASCADE) # Links to a Department
    course_hours = models.PositiveIntegerField(default=30)
    course_credits = models.PositiveSmallIntegerField(default=3)
    description = models.TextField(default='')
    marking_scheme = models.JSONField(default=dict) # How assessments are weighted
```

- **course_code**: A unique identifier (e.g., "CS101").
- **course_name**: The full name (e.g., "Introduction to Programming").
- **department**: Links the course to the Department that teaches it.
- **marking_scheme**: A JSONField that stores how different parts of a course (like quizzes, assignments, finals) contribute to the final grade. This is very flexible!

Example: Adding a Course

An executive can add a course via **executives/components/course_manager.py**:

```
def add_course(request):
    data = request.POST
    department = Department.objects.get(pk=int(data['department'])) # Find the Department

    new_course = Course(
        department=department, # Link the Course to the Department
        course_code=data['code'],
        course_hours=int(data['hours']),
        course_name=data['name'],
        course_credits=int(data['credits']),
```

```
        description=data['description'],
        marking_scheme={'Quiz': 20, 'Assignment': 30, 'Final': 50}, # Example marking scheme
    )
    new_course.save()
    return redirect('/executives/show_course_management')
```

- This snippet shows creating a Course and associating it with a **Department**. It also demonstrates how **marking_scheme** (a **JSONField**) can store flexible data.

5. Semester

Semester models break down a **Degree** into manageable periods, usually containing a list of courses.

```
# authorization/models.py (simplified Semester model)
class Semester(BaseModel):
    semester_name = models.CharField(max_length=100, default='')
    degree = models.ForeignKey(Degree, on_delete=models.CASCADE) # Links to a Degree
    duration_weeks = models.PositiveSmallIntegerField(default=4, null=False)
    syllabus_structure = models.JSONField(default=dict, null=False)
    # Example syllabus_structure:
    # [
    #   {
    #     "course_code": "CS101",
    #     "course_name": "Intro to CS",
    #     "course_credits": 3,
    #     "course_hours": 45,
    #     "type": "Core"
    #   },
    #   ...
    # ]
```

- **semester_name**: (e.g., "Semester 1").
- **degree**: Links this semester to the Degree it belongs to.
- **syllabus_structure**: Another powerful **JSONField**! This field can store a list of dictionaries, where each dictionary describes a course taken in this semester, including its code, name, credits, and type. This makes the syllabus highly flexible and easy to update without changing database columns.

6. Term

Term models define the overarching academic calendar periods.

```
# authorization/models.py (simplified Term model)
class Term(BaseModel):
    year = models.PositiveIntegerField(default=timezone.now().year, null=True)
```

```
    term_name =models.CharField(max_length=100, default='', null=True)
    start_date = models.DateField(default=timezone.now, null=True)
    end_date = models.DateField(default=timezone.now, null=True)
    result_date = models.DateField(default=timezone.now, null=True)


    def save(self, *args, **kwargs):
        with transaction.atomic():
            super().save(*args, **kwargs)
            # When a Term is saved with valid dates, it sends notifications
            # to all active students about the new term!
            if self.start_date and self.end_date and self.result_date:
                # ... (code to create notifications for students) ...
                pass # Simplified for brevity
```

- **year**, **term_name**: Define the specific term (e.g., 2024-25 June Term).
- **start_date**, **end_date**, **result_date**: Key dates for the academic period.
- **save()** method: Notice the special save method here! It contains logic to automatically send notifications to students when a new term is created or updated, informing them about the upcoming academic period. This is an example of a **model's method triggering business logic**.

Example: Creating a Term

Creating a term is handled in **executives/components/term_manager.py**:

```
# executives/components/term_manager.py (simplified create_term)
def create_term(request):
    if request.method == 'POST':
        data = json.loads(request.body)
        term = Term.objects.create(
            term_name=data.get('name', ''),
            start_date=data.get('startDate') or None,
            end_date=data.get('endDate') or None,
            year=data.get('year') or 0,
            result_date=data.get('resultDate') or None
        )
        # The save() method of the Term model will automatically send notifications.
        return JsonResponse({'success': True, 'id': term.id})
    return HttpResponseBadRequest('Invalid method')
```

- This creates a **Term** record. When **Term.objects.create()** is called, it eventually calls the **save()** method of the **Term** model, which then triggers the notification logic.

3.4 User and Role Management (Authorization App)

In the previous part, we built the blueprint of our university's courses, degrees, and departments.

We know *what* is taught and *how* it's organized. But who are the people involved in this university system? Who learns? Who teaches? Who manages everything?

This is where **User & Role Management** comes in, powered by the core authorization app. Think of it as the university's central ID and access control system. It defines who everyone is (students, faculty, executives) and what they're allowed to do. Just like a physical university has different types of ID cards that grant access to different buildings or rooms, our digital system needs to know your identity and your role to give you the right access.

By the end of this part, you'll understand how different people (users) are identified in the system, how they are assigned specific roles (like Student, Instructor, or Admin), and how the system makes sure they only access the parts they're allowed to see.

3.4.1 What Problem Does User & Role Management Solve?

Imagine a university without proper ID cards. A student might accidentally walk into a high-security executive meeting, or an instructor might not be able to access the course materials they need to teach. Chaos!

Our digital system faces the same challenge. We need to:

1. **Identify everyone:** Give each person a unique identity in the system.
2. **Assign roles:** Categorize people based on what they do (student, instructor, admin).
3. **Control access:** Ensure a student can't see an executive's dashboard, and an instructor can't change a student's grades arbitrarily.

Central Use Case: A new student registers, logs in, and sees their student dashboard. Later, an instructor logs in and sees their instructor dashboard. An executive logs in and sees their administrative tools.

3.4.2 Key Concepts: Users, Roles, and the Bouncer

The authorization app uses a few core ideas to manage people:

1. **The** User **Model: Your Master ID Card**
    - This is the most basic identity for *anyone* in the system.
    - Think of it as the master ID card that everyone gets. It holds fundamental information like your email, full name, and password.
    - The **User** model is built upon Django's powerful **AbstractUser**, which provides a secure foundation for user accounts.

```
class User(AbstractUser, BaseModel): # BaseModel adds created_at, updated_at
    email = models.EmailField(unique=True, db_index=True, default='')
    full_name = models.CharField(max_length=100, default='')
    phone = models.CharField(max_length=20, default='')
    profile_picture = models.ImageField(upload_to='profile_pictures/', default='default_profile.png')
    # ... other fields like city, date_of_birth, telegram_username, gender ...

    # Special fields inherited from AbstractUser for roles
    is_active = models.BooleanField(default=True)
    is_staff = models.BooleanField(default=False) # True for Instructors and Admins
    is_superuser = models.BooleanField(default=False) # True for top-level Admins
    # ... and more for permissions

    # Fix clashing reverse accessors (don't worry about this for now, it's advanced)
    groups = models.ManyToManyField(
        'auth.Group', related_name='custom_user_set', blank=True, # ...
    )
    user_permissions = models.ManyToManyField(
        'auth.Permission', related_name='custom_user_set', blank=True, # ...
    )
```

```
22.
23.     def save(self, *args, **kwargs):
24.         if not self.full_name:
25.             self.full_name = f"{self.first_name} {self.last_name}".strip()
26.         super().save(*args, **kwargs)
```

- **email**: Every user needs a unique email to log in.
- **full_name**: The user's complete name.
- **profile_picture**: Stores their profile photo.
- **is_staff**: This is a built-in Django field. If True, the user can access the Django admin site. We use it to identify Instructors and Admins.
- **is_superuser**: If True, the user has *all* permissions, like the university President.

2. **Role Models (**Student**,** Instructor**,** Admin**): Specialized Access Cards**
   - ➢ While **User** gives you a basic identity, these models give you a *specific role* and extra details unique to that role.
   - ➢ Each of these models has a special link to the User model called a **OneToOneField**. This means one User can only be one **Student**, one **Instructor**, or one **Admin** at a time.

Here's a quick look at these role models:

Role Model	Linked to User	Extra Information (Examples)	What it represents
Student	Yes (OneToOneField)	roll_no and status	A learner pursuing an academic program.
Instructor	Yes (OneToOneField)	department, specialization, employment_status	A teacher assigned to courses and batches.
Admin	Yes (OneToOneField)	degree, position_in_university, admin_level	An executive or staff member managing the university.

3.4.3 *UserRoleMiddleware*: The Bouncer at the Door

This is a special piece of code that runs before every page request. Its job is to check: "*Who is trying to access this page, and do they have permission?*"
If you're not logged in, it sends you to the login page.If you're trying to access an */executives/* page but you're only a student, it sends you to an ***access_denied*** page.

It uses a temporary storage called ***cache*** to remember your role so it doesn't have to look it up in the database every single time, making the system faster!

3.4.4 How it Solves the Problem: Registration and Login

Let's walk through our central use case: a new student registers and logs in, then a different user logs in with an instructor role, and finally, an executive logs in.

1. Registering a New Student (or Instructor)

When a new person signs up, they first provide their email. The system verifies it with an **OTP** (One-Time Password) and then asks for their full details and role.

```
def verify_mail(request):
    data = json.loads(request.body)
```

```
    email = data.get('email')

    # 1. Check if email already exists
    if User.objects.filter(email=email).exists():
        return JsonResponse({'success': False, 'message': 'Email already has an account'})

    # 2. Generate and store OTP
    otp_code = generate_otp()
    expiration_time = timezone.now() + timedelta(minutes=5)
    otp_record = OTP.objects.create(email=email, otp_code=otp_code, expires_at=expiration_time)

    # 3. Send OTP via email (simplified)
    send_mail('Email Verification OTP', f'Your OTP is: {otp_code}', settings.EMAIL_HOST_USER, [email])

    return JsonResponse({'success': True, 'message': 'OTP sent'})
```

- The ***verify_mail*** function receives an email address.
- It checks if an ***User*** already exists with that email.
- It generates a random **OTP**, stores it in the ***OTP*** model (a temporary record with an expiration time), and sends it to the user's email.

After verifying the **OTP** (***verify_otp*** function, not shown but similar to ***verify_mail***), the user provides their profile details and indicates if they are registering as a student or instructor.

```
save_student(request): # This view handles both student & instructor registration
    # 1. Get user details from form (e.g., full_name, email, password, profile_picture)
    email = request.POST.get('email', '')
    password = request.POST.get('password', '')
    profile_picture_file = request.FILES.get('profile_picture')
    # ... and other fields

    # 2. Save profile picture (simplified)
    fs = FileSystemStorage(location=settings.MEDIA_ROOT)
    profile_picture_path = fs.save(f'profile_pictures/{profile_picture_file.name}', profile_picture_file)

    # 3. Create the base User object
    user = User.objects.create_user(
        username=email, # Django often uses username for login, we use email
        email=email,
        password=password,
        full_name=request.POST.get('full_name', ''),
```

```
        profile_picture=profile_picture_path,
        # ... set other User fields ...
    )

    # 4. Create the specific role object (Student or Instructor)
    role = request.POST.get('role', 'student')
    if role.lower() == 'instructor':
        user.is_staff = True # Instructors can access some staff features
        user.save()
        Instructor.objects.create(
            user=user,
            specialization=request.POST.get('specialization'),
            employment_status=EmploymentStatus.UNAPPROVED # Needs admin approval
            # ... set other Instructor fields ...
        )
        return JsonResponse({'success': True, 'message': 'Instructor registered'})
    else: # Default to student
        Student.objects.create(
            user=user,
            student_number=generate_student_number(), # A unique ID for students
            status=StudentStatus.UNAPPROVED # Needs admin approval
            # ... set other Student fields ...
        )
        return JsonResponse({'success': True, 'message': 'Student registered'})
```

- This simplified ***save_student*** function first creates a base ***User*** record with the provided information.
- Then, depending on the *role* chosen, it creates either a ***Student*** or an ***Instructor*** record, linking it to the newly created ***User***.
- Notice how ***is_staff*** is set to ***True*** for instructors, granting them higher access.
- New students and instructors start with ***UNAPPROVED*** status, meaning an executive needs to approve them before they can fully use the system. This is a crucial security step!

2. Logging In

Once registered and approved, users can log in.

```
# authorization/views.py (simplified login function)
def login(request):
    data = json.loads(request.body)
    email = data.get('email')
    password = data.get('password')
```

```
    # 1. Authenticate user credentials
    user = auth.authenticate(request, username=email, password=password) # Checks email and password

    if user is not None:
        # 2. Check if account is approved/active
        if not user.is_active: # Base User is not active
            return JsonResponse({'success': False, 'error': 'Account is inactive'})

        # Additional checks for Student/Instructor status
        if not user.is_staff: # If not staff, it's likely a student
            try:
                student = Student.objects.get(user=user)
                if student.status == StudentStatus.UNAPPROVED:
                    return JsonResponse({'success': False, 'error': 'Student account pending approval.'})
            except Student.DoesNotExist:
                pass # Not a student, proceed

        if user.is_staff and not user.is_superuser: # If staff but not superuser, it's likely an instructor
or regular admin
            try:
                instructor = Instructor.objects.get(user=user)
                if instructor.employment_status == EmploymentStatus.UNAPPROVED:
                    return JsonResponse({'success': False, 'error': 'Instructor account pending approval.'})
            except Instructor.DoesNotExist:
                pass # Not an instructor, proceed (could be a regular admin)

        # 3. Log the user into Django's session system
        auth.login(request, user)

        # 4. Determine and store the user's role for quick access
        user_role = ""
        if Student.objects.filter(user=user).exists():
            user_role = "student"
        elif Instructor.objects.filter(user=user).exists():
            user_role = "instructor"
        elif Admin.objects.filter(user=user).exists():
            user_role = "executive" # For our system, Admin means executive
```

```
        # Store role in cache (for fast lookup by middleware) and cookie
        cache.set(f"user_role:{user.email}", user_role, timeout=60*60*24*30*2) # Cache for 2 months
        response = JsonResponse({'success': True, 'role': user_role})
        response.set_cookie('my_user', user.email, max_age=60*24*60*60) # Cookie stores email
        return response
    else:
        return JsonResponse({'success': False, 'error': 'Invalid credentials'})
```

- This ***login*** function first tries to ***authenticate*** the user using their email and password against the ***User*** model.
- If authentication succeeds, it performs additional checks based on the *Student* or *Instructor* status (e.g., ***UNAPPROVED***).
- Then, *auth.login(request, user)* establishes a session, marking the user as logged in.
- Crucially, it determines the user's specific role (student, instructor, or executive) by checking if a related *Student*, *Instructor*, or *Admin* object exists for that *User*.
- This ***user_role*** is then stored in a *cache* and a browser *cookie* (***my_user***) for quick retrieval later by the ***UserRoleMiddleware***.

3. Role-Based Redirection with UserRoleMiddleware

After a user logs in, the ***UserRoleMiddleware*** (our bouncer) springs into action for every subsequent page request.

```
class UserRoleMiddleware:
    # List of paths anyone can access without being logged in
    PUBLIC_PATHS = [
        '/auth/login/', # Login page itself
        '/public/',     # Public home page
        '/public/acces_denied/', # Access denied page
        # ... many more public paths ...
    ]

    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        path = request.path # The URL the user is trying to visit

        # 1. Skip checks for public paths
        for p in self.PUBLIC_PATHS:
            if path.startswith(p):
                return self.get_response(request) # Let them pass immediately
```

```
        # 2. Get user's role from cache (set during login)
        user_email = request.COOKIES.get('my_user') # Get email from cookie
        role = cache.get(f"user_role:{user_email}") # Get role from cache using email

        # 3. If no role (not logged in), redirect to login page
        if not role:
            return redirect('/auth/login/')

        # 4. Check specific role permissions for protected paths
        if request.path.startswith('/executives/') and role != 'executive':
            return redirect('/public/access_denied/') # Not an executive? No executive pages!

        # ... similar checks for /faculty/ and /students/ paths ...

        # 5. If all checks pass, let the request continue to the view
        request.user_role = role # Make the role available in the view
        response = self.get_response(request)
        return response
```

- The ***UserRoleMiddleware*** first checks if the requested *path* is one of the ***PUBLIC_PATHS***. If so, it allows access immediately.
- If not a public path, it tries to get the user's ***email*** from a cookie and then retrieves their *role* from the ***cache***.
- If no *role* is found, the user is not logged in, so they are ***redirected*** to the login page.
- Finally, it checks if the *role* matches the *path*. For example, if the path starts with */executives/* but the user's ***role*** is not ***executive***, they are redirected to an ***access_denied*** page.
- This ensures that only users with the correct role can access specific parts of the system, just like having the right ID card to open a specific door.

3.5 User-Specific Dashboards & Features (Apps)

In the previous part, we learned how our university system identifies who everyone is (their User ID) and what their job is (their Student, Instructor, or Admin "role"). We even saw how a special "bouncer" (**UserRoleMiddleware**) helps control who can go where.
But what happens once you're inside? Once the bouncer lets you through, you don't want to see a confusing mess of tools and information meant for someone else! Imagine walking into the student lounge but finding executive meeting notes everywhere, or a professor trying to teach a class but only seeing student assignment submission forms. That would be chaotic!

This is where **User-Specific Dashboards & Features**, organized into separate "apps," come in. Think of these apps as custom-designed control panels. Each main type of user (students, faculty, and executives) gets their own personalized control panel, filled only with the tools and information relevant to their role.

Our Goal for this Chapter: By the end, you'll understand why the system has separate "apps" for students, faculty, and executives, and how each app provides a streamlined, customized experience tailored to that user's specific needs.

3.5.1 What Problem Do User-Specific Dashboards Solve?

The main problem is complexity and relevance. A university system needs to do *a lot* of different things: manage courses, grade assignments, approve students, update university info, send messages, and so on. If all these features were mixed into one giant screen, it would be overwhelming and confusing for everyone.

Central Use Case:

- A **student** logs in and immediately sees their enrolled courses, upcoming assignments, and grades.
- A **faculty** member logs in and sees a list of the courses they are teaching, links to create quizzes, and a way to view student submissions.
- An **executive** logs in and sees a summary of unapproved student applications, overall university statistics, and tools to manage departments or labs.

Each user needs a dedicated workspace. This is achieved by creating separate "applications" within our larger **ums** project.

3.5.2 Key Concepts: Apps as Custom Control Panels

In Django (the web framework ums is built on), a "project" is usually made up of several smaller, self-contained "apps." This is perfect for our university system! We have:

1. **The** students **App:**
    - This is the student's personal portal.
    - It contains everything a student needs: academic dashboards, tools for taking quizzes, submitting assignments, checking results, and communicating with others via a mailbox.
    - It's like a student's personal "My Campus" section.
2. **The** faculty **App:**
    - This is the faculty member's (instructor's) teaching workstation.
    - It offers tools to manage courses, create assignments and quizzes, upload teaching materials, and review student submissions.
    - It's their "Teaching Hub."
3. **The** executives **App:**
    - This is the administrative powerhouse for university staff.
    - It provides dashboards for oversight, management tools for academic structures (faculties, departments, degrees), student and instructor approvals, and even complex tasks like result processing and lab management.
    - It's their "University Command Center."

This modular design means that each user type has a tailored experience, with relevant features grouped together, making the system much easier to use and manage.

3.5.3 How it Solves the Problem: Personalized Access

Let's see how our system directs users to their specific dashboards after they log in, building on what we learned in Chapter 3: User & Role Management (Authorization App).

1. Login and Role Identification

When a user logs in, the system uses their email and password to verify their identity. Then, it quickly figures out their role (Student, Instructor, or Admin/Executive) using the User and linked role models. This role is stored temporarily in a fast-access cache and a browser cookie.

2. The UserRoleMiddleware (Our Bouncer) Steps In

For ***every*** page request after login, the **UserRoleMiddleware** checks the user's role. It acts as a router, sending users to the correct "app" or dashboard based on who they are.

- If you're a student, it points you to the **students** app (e.g., /students/academic/).
- If you're an instructor, it points you to the **faculty** app (e.g., /faculty/dashboard/).
- If you're an executive, it points you to the **executives** app (e.g., /executives/dashboard/).

This happens automatically behind the scenes!

3. Displaying the Right Dashboard

Once the **UserRoleMiddleware** directs the user to their app's home page, that app's views.py file takes over. It fetches the specific data needed for *that* user's dashboard and displays it.
Let's look at simplified examples from each app:

Executive Dashboard **(**executives/views.py**)** Executes see administrative summaries, like how many students or instructors are waiting for approval.

```
# executives/views.py (simplified)
def executive_home(request):
    # Fetch data specific to an executive's needs
    unapproved_student_count = Student.objects.filter(
        status=StudentStatus.UNAPPROVED
    ).count()
    unapproved_instructor_count = Instructor.objects.filter(
        employment_status=EmploymentStatus.UNAPPROVED
    ).count()

    data = {
        'unapproved_student_count': unapproved_student_count,
        'unapproved_instructor_count': unapproved_instructor_count,
    }
    return render(request, 'executives/home.html', context=data)
```

- This code snippet fetches counts of unapproved students and instructors – data highly relevant to an executive.
- It then passes this data to an HTML template designed for the executive dashboard.

Faculty Dashboard **(**faculty/views.py**)** Faculty members see information related to the courses they teach.

```
def faculty_dashboard(request):
    # Find the logged-in instructor's profile
    instructor = Instructor.objects.filter(
        user__email = request.COOKIES.get('my_user'),
    ).first()

    if not instructor:
        # If somehow not an instructor, deny access
        return render(request, 'htmls/access_denied.html')

    # Fetch all courses and batches assigned to this instructor
    batch_instructors = BatchInstructor.objects.filter(
        instructor=instructor
    ).select_related('batch__term', 'course') # Also get related batch/course info

    return render(request, 'faculty/faculty_home.html', {'batch_instructors': batch_instructors})
```

- This code gets the **Instructor** object for the logged-in user.
- It then retrieves all **BatchInstructor** records associated with them, which link the instructor to specific courses they are teaching in specific terms. This is highly relevant information for a faculty member.

Student dashboard contains all the enrolled terms of that related student. All the terms are grouped by the enrollment status (**PENDING**, **APPROVED**, **REJECTED**)

To summarize, after a successful login, the **UserRoleMiddleware** acts as the traffic controller, routing requests to the appropriate application's dashboard based on the user's role. If a user tries to access a dashboard not meant for their role, they are redirected to an Access Denied page.
This system ensures security and a clean, relevant user experience for everyone within the university.

3.6 Assessment and Result Processing

In the previous part, we saw how different users (students, faculty, executives) get their own specialized control panels. Now, let's dive into one of the most critical aspects of any university: **Assessment & Result Processing**. This is how students are tested, how their performance is measured, and how their final grades are calculated and delivered.
Imagine a student named Alice. She attends her classes, completes assignments, takes quizzes, and sits for her final exams. How does the university system keep track of all her grades? How does it know how much each quiz or exam contributes to her final course mark? And how does it generate her official report card at the end of the term?

By the end of this part, you'll understand how assessments are defined, how student scores are recorded, and how the system calculates and delivers final results, just like a university's examination board.

3.6.1 What Problem Does Assessment & Result Processing Solve?

Managing grades manually for hundreds or thousands of students across many courses and different types of assessments would be a nightmare! This system automates the entire grading and reporting pipeline, ensuring fairness, accuracy, and timely delivery of results.

Central Use Case: An instructor creates an assignment for their course. Students submit their work. The instructor grades it. At the end of the term, the university processes all grades to generate final report cards, which are then sent to students.

3.6.2 Key Concepts: The Grading Blueprint

The **ums** system uses several interconnected pieces to handle assessments and results:

Concept	Analogy	Description
Assessment	**The Test Itself**	Defines a specific quiz, assignment, or exam, including its structure, questions, and due dates.
AssessmentScheme	**Course Grading Blueprint**	A recipe for how different Assessment types (quizzes, assignments, finals) contribute to a course's final grade.
AssessmentResult	**Your Scorecard Entry**	Stores what a specific student scored on a particular Assessment.

Result Components	**The Examination Board**	A set of internal tools that take all the raw AssessmentResults, apply the AssessmentScheme, calculate final grades, generate official reports, and send them out.

3.6.3 How It Solves the Problem: From Assignment to Report Card

Let's follow our central use case step-by-step:

1. Instructor Creates an Assessment (Assessment)

An instructor logs into their faculty app, and wants to create a new quiz. They provide details like the **quiz title, questions, total marks, and the due date**. This information is stored in an Assessment object.
Here's how an instructor might save a new quiz using **`faculty/components/quiz_manager.py`**:

```
def create_quiz(request):
    quiz_data = json.loads(request.POST.get('quiz')) # Get quiz details
    start_time = request.POST.get('start_time')
    end_time = request.POST.get('end_time')

    # Convert date strings to timezone-aware datetime objects
    assigned_date = timezone.make_aware(datetime.fromisoformat(start_time))
    due_date = timezone.make_aware(datetime.fromisoformat(end_time))

    # Find the existing Assessment object (it's usually pre-created by the system)
    assessment = Assessment.objects.get(id=int(request.POST.get('assessment_id')))

    # Update its details
    assessment.assessment = quiz_data # Store quiz questions, options in JSONField
    assessment.assigned_date = assigned_date
    assessment.due_date = due_date
    assessment.save() # This triggers a notification to students!

    return JsonResponse({'success': True, 'message': 'Quiz saved!'})
```

- The **create_quiz** function receives the quiz details from a form.
- It finds an existing Assessment record (which would have been automatically created when the course was set up for the instructor – we'll see this soon!).
- It updates the assessment field (a **JSONField**) with all the quiz questions and options, and sets the **assigned_date** and **due_date**.
- When **assessment.save()** is called, the **Assessment** model's special save method (we'll look at this later) automatically sends notifications to all relevant students, informing them about the new quiz!

2. Student Submits an Assessment (AssessmentResult)

Alice sees the notification for the new quiz, takes it, and submits her answers. The system then creates an **AssessmentResult** record for Mg Mg, storing her answers and initially setting her mark to zero.
Here's a conceptual idea of how a student's submission might be saved (the actual code would be in the students app and is not provided, but it would interact with the **AssessmentResult** model):

```
def submit_quiz_answers(request, assessment_id):
    assessment = Assessment.objects.get(id=assessment_id)
    student = Student.objects.get(user=request.user) # Get current student

    student_answers = request.POST.get('answers') # Answers from the quiz form

    # Create an AssessmentResult for this student and assessment
    AssessmentResult.objects.create(
        assessment=assessment,
        student=student,
```

```
        answer=json.loads(student_answers), # Store answers in JSONField
        mark=0 # Instructor will update this later
    )
    return JsonResponse({'success': True, 'message': 'Quiz submitted!'})
```

- When a student submits, a new **AssessmentResult** is created, linking to the specific **Assessment** and the Student.
- The student's answer is stored in a **JSONField** for detailed record-keeping.
- The mark is initially 0; the instructor will fill this in.

3. Instructor Grades an Assessment (AssessmentResult)

The instructor reviews Alice's submission and gives her a score. They update the **AssessmentResult** record for Alice with her earned mark.

This is handled by **faculty/components/assignment_manager.py** (the same logic applies for quizzes):

```
def update_assessment_mark(request, assessment_result_id, mark):
    try:
        # Find the specific AssessmentResult for Alice
        assessment_result = AssessmentResult.objects.get(pk=assessment_result_id)

        # Update her mark
        assessment_result.mark = mark
        assessment_result.save() # This triggers a notification to the student!

        return JsonResponse({'success': True, 'message': 'Mark updated!'})
    except AssessmentResult.DoesNotExist:
        return JsonResponse({'success': False, 'error': 'Result not found.'})
```

- The function takes the **assessment_result_id** (a unique ID for Mg Mg's specific submission) and the mark.
- It updates the mark field of the **AssessmentResult**.
- Crucially, when **assessment_result.save()** is called, the **AssessmentResult** model's special save method (which we'll explore) automatically sends a notification to Alice, telling her that her grades are out!

4. Executives Process Results (Result Components)

At the end of the term, an executive triggers the "examination board" (**result_components**) to process all student grades for a given Term.

This process involves:

- **Calculating Course Scores:** For each student and each course, the system looks at all AssessmentResults and applies the AssessmentScheme (the grading blueprint) to calculate a final score for the course. For example, if quizzes are 20%, assignments 30%, and finals 50%, it weights the scores accordingly.
- **Generating Report Cards:** It uses the calculated scores to determine letter grades (A+, B, C, etc.) and GPA. It then generates a visual report card (an image file) for each student.
- **Sending Results:** The generated report cards are attached to emails and sent to students, and a notification is also pushed to their in-app mailbox.

Here's how the **executives/result_components/result_email_sender.py** orchestrates this:

```
def send_result_to_all_students(term_id, executive_user):
    term = Term.objects.get(id=term_id)
    enrollments = Enrollment.objects.filter(batch__term=term)

    for enrollment in enrollments:
        student = enrollment.sis_form.student
        user = student.user

        # 1. Calculate all course results for this student
        enrollment_courses = enrollment.enrollmentcourse_set.all() # Get all courses student took
        final_results_data = get_result_for_all_courses(enrollment_courses)
```

```
        # 2. Generate an image report card
        image_path = create_result_image_dynamic(
            courses=final_results_data,
            student_name=user.full_name,
            # ... other student/course details ...
        )

        # 3. Store result data and image path in the Enrollment record
        enrollment.result = {'data': final_results_data, 'image': image_path}
        enrollment.save()

        # 4. Send email with attached report card
        email_message = EmailMessage(
            subject=f"Exam Result - {term.term_name}",
            body=f"Dear {user.full_name}, ...",
            to=[user.email],
        )
        email_message.attach_file(os.path.join(settings.MEDIA_ROOT, image_path))
        email_message.send()

        # 5. Send in-app notification
        Notification.objects.create(
            user=user,
            notification={"text": f"Exam results for {term.term_name} are out!"},
        )
    return JsonResponse({'success': True, 'message': 'Results sent!'})
```

- This function is called by an executive for a specific **Term**.
- It loops through all **Enrollments** for that term.
- For each student's enrollment, it calls **get_result_for_all_courses** (from **result_calculator.py**) to calculate their final grades, and then **create_result_image_dynamic** (from **result_image_generator.py**) to make a visual report card.
- It saves these results to the Enrollment record.
- Finally, it sends an email with the attached image and creates an in-app **Notification**.

5. Student Views Their Result Sheet (students app)

Alice receives the email and also sees an in-app notification. She can also visit her students app dashboard to view her past result sheets.

Here's how the **students/components/result_sheet_manager.py** would display these results:

```
def show_result_sheets(request, student_id):
    # Find all approved enrollments for the student
    enrollments = Enrollment.objects.filter(
        sis_form__student__user__id=student_id,
        enrollment_status=EnrollmentStatus.APPROVED,
    ).select_related('batch__term', 'batch__semester__degree')

    results = []
    for enrollment in enrollments:
        if enrollment.result and enrollment.result.get('image'):
            results.append({
                'term': enrollment.batch.term.term_name,
                'degree': enrollment.batch.semester.degree.name,
                'semester': enrollment.batch.semester.semester_name,
                'image_url': f"/media/{enrollment.result.get('image')}" # Path to the report card image
            })

    # Pass the list of report cards to the student's template
    data = {'results': json.dumps(results)}
    return render(request, 'students/academic/result_sheets.html', context=data)
```

- This function retrieves all Enrollment records for the logged-in student.
- It extracts the **image_url** for the generated report card (stored in the **enrollment.result JSONField**).
- The student's dashboard then displays these report card images.

3.6.4 Diving into the Code: Core Models and Logic

Let's look at the model definitions in **authorization/models.py** and the **result_components** to understand how this all works.

1. AssessmentScheme

This model is the grading blueprint for a course. It's automatically created when an instructor is assigned a course for a batch.

```
# authorization/models.py (simplified AssessmentScheme model)
class AssessmentScheme(BaseModel):
    batch_instructor = models.OneToOneField(BatchInstructor, on_delete=models.CASCADE)
    scheme = models.JSONField(default=dict)
```

- **batch_instructor**: This links the grading scheme directly to a specific course being taught by a specific instructor in a specific Batch (a unique offering of a course).
- **scheme**: This is a **JSONField** (our flexible "magic box"). It stores a dictionary where keys are assessment types (like 'Quiz', 'Assignment', 'Final') and values are their percentage weights *(e.g., {'Quiz': 20, 'Assignment': 30, 'Final': 50})*. This makes the grading structure very flexible – you can easily change percentages without changing the database structure!

2. Assessment

This defines a single test, assignment, or quiz.

```
# authorization/models.py (simplified Assessment model)
class Assessment(BaseModel):
    assessment_scheme = models.ForeignKey(AssessmentScheme, on_delete=models.CASCADE, null=True, blank=True)
    assessment_type = models.TextField(max_length=1, choices=AssessmentType.choices,
default=AssessmentType.QUIZ)
    assessment = models.JSONField(default=dict) # The actual questions/details
    assigned_date = models.DateTimeField(auto_now_add=False, default=None, null=True, blank=True)
    due_date = models.DateTimeField(auto_now_add=False, default=None, null=True, blank=True)

    def save(self, *args, **kwargs):
        with transaction.atomic():
            super().save(*args, **kwargs)
            # ... (simplified for brevity - notification logic follows) ...

            # If this assessment is for specific students (e.g., re-take for some)
            if self.assessment.get('students'):
                # Send notifications to specific students
                # ... (code to create notifications for selected students) ...
                pass
            else:
                # If for all students in the course, notify the instructor who created it
                # ... (code to create notification for instructor) ...
                pass
```

- **assessment_scheme**: Links this assessment to the overall grading blueprint for the course.
- **assessment_type**: Specifies the type (e.g., 'Q' for Quiz, 'A' for Assignment).
  The **AssessmentType** choices help categorize.
- **assessment**: Another **JSONField**! This is where the actual content of the quiz (questions, options, correct answers) or assignment (description, rubric) is stored. This allows for highly flexible assessment structures.
- **assigned_date, due_date**: Important dates for the assessment.
- **save()** method: This method contains business logic. After an **Assessment** is saved (or updated), it automatically sends notifications either to all students assigned to it (if specific students are targeted) or to the instructor (if it's a general assessment creation). This ensures timely communication.

3. AssessmentResult

This model stores a student's performance on a single assessment.

```python
# authorization/models.py (simplified AssessmentResult model)
class AssessmentResult(BaseModel):
    assessment = models.ForeignKey(Assessment, on_delete=models.CASCADE)
    student = models.ForeignKey(Student, on_delete=models.CASCADE, null=True, blank=True)
    answer = models.JSONField(default=dict) # Student's submitted answers
    mark = models.PositiveIntegerField(default=0)

    def save(self, *args, **kwargs):
        with transaction.atomic():
            old_mark = None # To check if mark changed
            if self.pk:
                try:
                    old_instance = AssessmentResult.objects.get(pk=self.pk)
                    old_mark = old_instance.mark
                except AssessmentResult.DoesNotExist:
                    pass

            super().save(*args, **kwargs) # Save current state

            # ... (simplified for brevity - notification logic follows) ...

            # Notify student upon submission
            if self.assessment.assessment_type in (AssessmentType.QUIZ, AssessmentType.ASSIGNMENT):
                # ... (code to create notification for student about submission) ...
                pass

            # Notify student when mark changes (e.g., instructor has graded)
            if old_mark is not None and old_mark != self.mark:
                # ... (code to create notification for student about new grade) ...
                pass
```

- **assessment**: Links this result to the specific Assessment (e.g., "Math Quiz 1").
- **student**: Links this result to the specific Student (e.g., Alice).
- **answer**: A JSONField holding the student's actual submitted answers for the assessment.
- **mark**: The score the student received.
- **save()** method: Similar to Assessment, this model also has important business logic in its save() method. It sends a notification to the student upon their submission, and *another* notification if their mark changes (meaning the instructor has graded it!).

4. The "Examination Board": Result Components

These are not models but Python scripts that contain the logic to calculate and distribute results.

- **executives/result_components/result_calculator.py**:
  - Contains functions like **get_result_for_a_course**(**enrollment_course**) which uses the **AssessmentScheme** and **AssessmentResults** to calculate a student's final score for a single course.
  - **get_credits_and_score(total_marks)** which converts raw scores into letter grades and grade points.
  - **get_result_for_all_courses(enrollment_courses)** which processes all courses for a student in a term.
- **executives/result_components/result_image_generator.py**:

    - Contains **create_result_image_dynamic(...)**. This function takes the processed results for a student and visually lays them out on an image, generating a professional-looking report card. This image is then saved to the system's media storage and its path is returned.
- **executives/result_components/result_email_sender.py**:
    - Orchestrates the entire result distribution. It calls **result_calculator** and **result_image_generator**, then attaches the generated image to an email and sends it to the student, also creating an in-app notification.

These components work together seamlessly to automate what would otherwise be a massive manual effort, making the university's result processing efficient and reliable.

3.7 Mailbox and Notifications System

In the previous part, we explored how the university manages grades, from setting up assignments to delivering final report cards. But beyond formal academics, a university is a vibrant community that needs to communicate effectively. Students need to know about new quizzes, faculty might post important announcements, and the system itself needs to alert users to important events.
Imagine your university is a bustling campus, and sometimes you need to share a quick announcement, discuss a topic, or get an urgent alert. How do you do that in a digital system? You don't want to rely on messy group chats or forgotten emails!

This is where the **Mailbox & Notifications System** comes in. Think of it as the university's internal communication hub – a central place for announcements, discussions, and personalized alerts, much like a school messenger service combined with a campus bulletin board.

By the end of this part, you'll understand how users can post messages and engage in discussions, and how the system automatically sends timely alerts about crucial events directly to their personalized inbox.

3.7.1What Problem Does the Mailbox & Notifications System Solve?

Without a dedicated communication system, important messages can get lost, students might miss critical deadlines, and general announcements could go unnoticed. This system aims to:

1. **Facilitate internal communication:** Provide a platform for users to share information and engage in discussions.
2. **Ensure timely alerts:** Deliver automated, personalized notifications for important events.
3. **Create a central hub:** Offer a single, organized place for all university-related communications.

Central Use Case:

- A **student** wants to ask a question to the wider student body or make an announcement. They post it on the virtual "bulletin board."
- Other students see the post and **comment** on it, creating a discussion.
- Separately, when an **instructor grades a student's quiz**, the system automatically sends a **notification** to that student about their new grade.
- When a **new academic term is announced**, all active students receive a notification.

3.7.2 Key Concepts: Your Digital Communication Tools

The **ums** project uses three main models to build this communication hub:

Model Name	Analogy	Description
MailboxPost	**Campus Bulletin Board**	Where users (students) can share public announcements or messages, often requiring moderation.
Comment	**Discussion Thread**	Allows users to respond and discuss under a MailboxPost or even reply to other Comments.
MailboxAdmin	**Bulletin Board Moderator**	Special student users who can approve, reject, or disqualify MailboxPosts to ensure appropriate content.
Notification	**Personalized School Messenger Alerts**	Ensures users receive timely, personalized alerts about important events (grades, deadlines, approvals).

These models work together to create a dynamic and informative communication environment.

3.7.3 How It Solves the Problem: From Post to Personalized Alert

Let's walk through our central use case step-by-step:

1. Student Creates a Mailbox Post (MailboxPost)

A student, let's call her Thidar, wants to share an interesting event on campus. She goes to the "Mailbox" section of her students app and creates a new post. This post is initially in a **PENDING** state, waiting for moderation.

Here's a simplified look at how a student might create a new post (from **students/mailbox/views.py**):

```
def create_post_api(request):
    data = json.loads(request.body)
    post_text = data.get('post_text', '')
    is_anonymous = data.get('is_anonymous', False)

    # Get the Student object for the logged-in user
    student = Student.objects.get(user=request.user)

    if not post_text:
        return JsonResponse({'success': False, 'error': 'Post text cannot be empty'}, status=400)

    # Create a new MailboxPost
    MailboxPost.objects.create(
        uploaded_by=student, # Link to the student who posted
        post={'text': post_text}, # Store the actual post content in JSON
        is_anonymous=is_anonymous,
        status='P' # PENDING status by default, needs admin approval
    )
    # At this point, the system would typically send a notification to Mailbox Admins
    # to let them know there's a new post to review. (Logic omitted for brevity).

    return JsonResponse({'success': True, 'message': 'Post created and pending approval!'})
```

- This code snippet takes the post content (post_text) and is_anonymous flag.
- It links the post to the Student who uploaded it.
- It sets the status to 'P' (Pending), meaning it won't be visible to everyone until approved.

2. Mailbox Admin Approves the Post (MailboxPost)

A **MailboxAdmin** (a student with special moderation privileges) reviews Sarah's post. If it meets guidelines, they approve it. This changes the post's status and makes it visible to other students. Crucially, the system notifies Sarah about her post's new status.

Let's look at the **MailboxPost** model's **save()** method for this:

```
# authorization/models.py (simplified MailboxPost save method)
class MailboxPost(BaseModel):
    # ... other fields ...
    status = models.CharField(max_length=1, choices=MailboxPostStatus.choices,
default=MailboxPostStatus.PENDING)

    def save(self, *args, **kwargs):
        old_status = None
        if self.pk: # If the post already exists (not a new one)
            old_status = MailboxPost.objects.get(pk=self.pk).status

        super().save(*args, **kwargs) # Save the post with the new status

        if old_status != self.status: # Check if status actually changed
            truncated_text = Truncator(self.post.get('text', '')).chars(50, truncate='...')

            if self.status == MailboxPostStatus.APPROVED:
                # Notify the original poster that their post has been approved
                Notification.objects.create(
                    user=self.uploaded_by.user,
                    type=NotificationType.STUDENT_MAILBOX,
                    notification={"text": f"Your post '{truncated_text}' has been
approved!"},
                    # ... other notification details ...
                )
            elif self.status == MailboxPostStatus.REJECTED:
                # Notify the original poster that their post has been rejected
                Notification.objects.create(
                    user=self.uploaded_by.user,
                    type=NotificationType.STUDENT_MAILBOX,
                    notification={"text": f"Your post '{truncated_text}' has been
rejected."},
                    # ... other notification details ...
                )
            # ... similar logic for DISQUALIFIED ...
```

- The **save()** method automatically checks if the status of the **MailboxPost** has changed.
- If the status changes to **APPROVED** (or **REJECTED**, **DISQUALIFIED**), it creates a Notification for the **uploaded_by** Student (via their linked User), informing them of the update.

3. Student Comments on a Mailbox Post (Comment)

Once Sarah's post is approved, another student, David, sees it and wants to comment.

Here's how a **Comment** might be created (conceptual view, as specific views.py code not provided):

```
def add_comment_api(request, post_id):
    data = json.loads(request.body)
    comment_text = data.get('comment_text', '')
    parent_comment_id, is_anonymous= data.get('parent_comment_id'), data.get('is_anonymous', False)

    post = MailboxPost.objects.get(pk=post_id)
    commented_by_user = request.user

    new_comment = Comment.objects.create(
        post=post,
        comment=comment_text,
        commented_by=commented_by_user,
        is_anonymous=is_anonymous,
        parent=Comment.objects.get(pk=parent_comment_id) if parent_comment_id else None
    )

    # Crucial: Manually trigger notification logic since Comment model doesn't override save()
    if post.uploaded_by.user != commented_by_user: # Don't notify if user comments on their own post
        notify_new_comment_on_post(
            user=post.uploaded_by.user,
            post_id=post.pk,
            commenter_name=commented_by_user.full_name,
            post_text=post.post.get('text', ''),
            is_anonymous=is_anonymous
        )

    return JsonResponse({'success': True, 'comment_id': new_comment.id})
```

- This code creates a new Comment record, linking it to the **MailboxPost** and the User who commented.
- It also allows for nested comments (parent field) for threaded discussions.
- **Important:** Since Comment doesn't have a custom save() method, a separate function (**notify_new_comment_on_post** from **students/notifications.py**) is explicitly called to generate a notification for the original post creator.

4. System Sends a General Notification (e.g., New Grades or Term)

Beyond mailbox interactions, the system generates automatic notifications for other important events. We've seen examples of this in previous chapters:

New Grades: When an instructor updates a student's mark in an AssessmentResult, the AssessmentResult model's save() method automatically creates a Notification for that student.

```
# authorization/models.py (simplified AssessmentResult save method)
class AssessmentResult(BaseModel):
    # ... fields ...
    mark = models.PositiveIntegerField(default=0)

    def save(self, *args, **kwargs):
        old_mark = None
        if self.pk: old_mark = AssessmentResult.objects.get(pk=self.pk).mark
        super().save(*args, **kwargs)

        # If mark changed, send notification
        if old_mark is not None and old_mark != self.mark:
            Notification.objects.create(
                user=self.student.user,
```

```
                type=NotificationType.STUDENT,
                notification={
                    "text": f"Your instructor provided grades for
{self.assessment.assessment.get('title')}. Check it out!",
                    "destination":
f"/students/academics/batch_instructor/{self.assessment.assessment_scheme.batch_instructor.pk}"
                }
            )
```

This ensures students are immediately informed when their work is graded.

New Term Announcement: When an executive creates a new **Term**, the Term model's **save()** method automatically creates Notifications for all active students.

```
# authorization/models.py (simplified Term save method)
class Term(BaseModel):
    # ... fields ...
    start_date = models.DateField(default=timezone.now, null=True)
    # ... other fields ...

    def save(self, *args, **kwargs):
        super().save(*args, **kwargs)
        if self.start_date and self.end_date and self.result_date:
            # ... fetch active students ...
            for user in active_users: # Simplified loop
                Notification.objects.create(
                    user=user,
                    type=NotificationType.STUDENT,
                    notification={
                        "text": f"A new term is starting on {self.start_date}. Consider enrollment!",
                        "destination": f"/students/academics/home"
                    }
                )
```

This ensures all students are aware of important academic calendar updates.

These notifications are then displayed in the user's dedicated "notifications" section, providing a personalized stream of important alerts.

3.7.4 Diving into the Code: Core Models

Let's look at the actual model definitions in **authorization/models.py** and some logic in **`students/notifications.py`**.

1. MailboxPost

This is the core model for public messages on the bulletin board.

```
# authorization/models.py (simplified MailboxPost model)
class MailboxPost(BaseModel):
    post = models.JSONField(default=None) # The content, potentially with media links
    uploaded_by = models.ForeignKey(Student, on_delete=models.SET_NULL, null=True, default=None)
    approved_by = models.ForeignKey(MailboxAdmin, on_delete=models.SET_NULL, null=True, default=None)
    status = models.CharField(max_length=1, choices=MailboxPostStatus.choices,
default=MailboxPostStatus.PENDING)
    is_anonymous = models.BooleanField(default=False)

    def save(self, *args, **kwargs):
        # ... (logic to handle status change notifications, as shown above) ...
```

```
        # This part ensures the original poster is notified when their post is approved/rejected.
        super().save(*args, **kwargs)
```

- **post**: A **JSONField** where the actual message text and any associated media (like image URLs) are stored. This makes the post content flexible.
- **uploaded_by**: Links the post to the Student who created it.
- **approved_by**: Links to the MailboxAdmin who handled the post.
- **status**: Crucial for moderation, determines if the post is **PENDING**, **APPROVED**, **REJECTED**, or **DISQUALIFIED**.
- **save()** method: Contains the business logic to send notifications back to the **uploaded_by** student when the status of their post changes.

2. Comment

This model allows for discussions under posts or as replies to other comments.

```
# authorization/models.py (simplified Comment model)
class Comment(BaseModel):
    post = models.ForeignKey(MailboxPost, on_delete=models.CASCADE)
    parent = models.ForeignKey('self', on_delete=models.CASCADE, null=True, blank=True, default=None,
related_name='replies')
    comment = models.TextField(default='') # The actual comment text
    reactions = models.JSONField(default=dict
    commented_by = models.ForeignKey(User, on_delete=models.CASCADE)
    is_anonymous = models.BooleanField(default=False)
```

- **post**: Links the comment to the **MailboxPost** it belongs to.
- **parent**: This enables threaded conversations! If a comment is a reply to another comment, this links to the parent **Comment**.
- **comment**: The text content of the comment.
- **reactions**: A **JSONField** to store emoji reactions (like 👍 or ❤️) and their counts, making comments interactive.
- **commented_by**: Links to the User who made the comment.
- **Note:** The Comment model itself *doesn't* have a custom save() method to send notifications directly. Instead, the logic to notify the original poster (or other commenters) is handled by helper functions in students/notifications.py when a comment is created or updated in the students app's views.

3. MailboxAdmin

These are special students who act as moderators.

```
# authorization/models.py (simplified MailboxAdmin model)
class MailboxAdmin(BaseModel):
    student = models.ForeignKey(Student, on_delete=models.CASCADE)
    added_by = models.ForeignKey('self', on_delete=models.SET_NULL, null=True, default=None,
related_name='added_admins')
    appointed_date = models.DateField(default=timezone.now)
```

- **student**: Links this administrative role to a specific **Student** in the system.
- **added_by**: Allows tracking which **MailboxAdmin** appointed another. This is for internal management of moderators.

4. Notification

This is the central model for all alerts, whether from the mailbox or other parts of the system.

```
# authorization/models.py (simplified Notification model)
class Notification(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    type = models.CharField(max_length=1, choices=NotificationType.choices, default=None)
    notification = models.JSONField(default=dict) # The actual message content
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    seen = models.BooleanField(default=False)
```

- **user**: Links the notification to the specific **User** who should receive it.
- **type**: Categorizes the notification (e.g., **STUDENT**, **EXECUTIVE**, **INSTRUCTOR**, **STUDENT_MAILBOX**). This helps filter and display notifications correctly.
- **notification**: A powerful **JSONField** that stores the actual message. This allows for rich, flexible content (*e.g., {"text": "Your grades are out!", "destination": "/students/grades/"}*).
- **seen**: A simple flag to track if the user has viewed the notification, allowing for "new notification" indicators.

3.7.5 Notification Logic Helpers (students/notifications.py)

As mentioned, some notification logic (especially for complex scenarios like multiple people reacting to a post, or multiple comments) is handled in dedicated helper functions.

```
def notify_new_comment_on_post(user, post_id, commenter_name, post_text, is_anonymous=False):
    # This function is called by the view when a comment is created.
    # It intelligently groups notifications. For example, if 3 people comment
    # on your post, it might update an existing notification to say:
    # "Commenter1 and 2 others commented on your post..."
    # ... (complex logic to find/update existing notifications or create new ones) ...
    if Notification.objects.filter(user=user, type=NotificationType.STUDENT_MAILBOX,
notification__contains=f'"post": {post_id}', seen=False).exists():
        # Update existing notification
        pass
    else:
        # Create new notification
        Notification.objects.create(
            user=user,
            type=NotificationType.STUDENT_MAILBOX,
            notification=json.dumps({
                "is_new_comment_on_post": True,
                "post": post_id,
                "comment_count": 1,
                "commenters": [] if is_anonymous else [commenter_name],
                "text": f"{commenter_name} commented on your post '{post_text[:30]}...'",
            }),
            # ... timestamps ...
        )
```

These helper functions ensure that notifications are smart (e.g., they group multiple reactions or comments into a single, updating notification) and don't spam the user. They are explicitly called from the views or other business logic, rather than being part of the **Comment** model's **save()** method.

3.7 Custom Middleware

In the previous part, we learned how students, faculty, and executives communicate effectively within the university system. Now, let's explore a powerful concept that works behind the scenes, almost like a silent guardian or a helpful assistant, for every interaction in our system: **Custom Middleware.**

Imagine your university campus has various checkpoints and services that every person (or "request" in software terms) passes through. Before you enter a building, security checks your ID. As you walk through a hallway, there might be signs reminding you of rules. Before you leave, a helpful bot might offer you a campus map. Middleware does exactly this for our ums system.

By the end of this chapter, you'll understand what middleware is, why it's so important for applying global rules and behaviors, and how our ums project uses custom middleware for security, performance, and user experience enhancements.

3.7.1 What Problem Does Custom Middleware Solve?

Think about common tasks that apply to almost *every* request or web page:

- Checking if a user is logged in.
- Making sure a student doesn't access an executive's page.
- Preventing someone from sending too many requests to our server (like spamming).
- Making sure image links automatically show a default picture if the original is broken.
- Rejecting requests that try to upload extremely large files.

If we had to write this code in *every single view function* (the code that builds a web page), it would be incredibly repetitive, hard to manage, and prone to errors.
This is where middleware shines! Middleware acts like a series of checkpoints that every incoming request (and outgoing response) passes through. It allows us to apply global logic efficiently without cluttering our main page-building code.
Central Use Case: A student tries to visit a page meant only for executives. Our system needs to quickly detect this and redirect them to an "**Access Denied**" page, all while also ensuring the student doesn't send too many requests in a short time.

3.7.2 Key Concepts: Your System's Gatekeepers and Helpers

In **Django** (the framework **ums** is built on), **middleware** is a framework of hooks that allows you to globally alter Django's request and response processing.
The ums project uses several **important custom middlewares**, found in authorization/middleware.py:

These middlewares are like different specialized services that process every request, adding security, resilience, and convenience to the entire system.

Middleware Name	Analogy	Description
UserRoleMiddleware	**The Main Bouncer**	Checks if you're logged in and if your role (Chapter 3: User & Role Management (Authorization App)) is allowed to access the requested page. It's the primary access control gate.
ImgFallbackMiddleware	**The Automatic Photo Fixer**	Automatically injects a tiny piece of code into every web page that makes sure broken image links (<img> tags) show a default placeholder image instead of an ugly broken icon.
RateLimitMiddleware	**The Traffic Controller**	Counts how many requests come from a specific user's computer (IP address) within a short time. If too many, it temporarily blocks them to prevent abuse (like spamming or hacking attempts).
PayloadSizeLimitMiddleware	**The Baggage Checker**	Examines the size of data (like file uploads) in incoming requests. If a request tries to send too much data (e.g., a huge image file), it rejects it to protect the server from being overwhelmed.

3.7.3 How it Solves the Problem: Seamless Global Handling

Let's revisit our central use case: a student tries to visit an executive page, and we also want to protect against too many requests.

1. Checking Roles with UserRoleMiddleware

As we saw in the previous parts, the **UserRoleMiddleware** is crucial. For every incoming request, it first checks if the user is logged in. If not, it sends them to the login page. Then, if they are logged in, it quickly checks their assigned role (**student, instructor, or executive**) against the URL they are trying to visit.
If a student tries to access **/executives/dashboard/**, this middleware instantly detects the mismatch and redirects them to **/public/access_denied/.** This logic is applied *before* the request even reaches the executive's dashboard code, saving resources and enforcing security.

```
class UserRoleMiddleware:
    # ... (init and PUBLIC_PATHS omitted for brevity) ...

    def __call__(self, request):
        path = request.path
        user_email = request.COOKIES.get('my_user') # Get user's email from browser cookie
        role = cache.get(f"user_role:{user_email}") # Get role from cache

        # If not logged in and not a public path, redirect to login
        if not role and not any(path.startswith(p) for p in self.PUBLIC_PATHS):
            return redirect('/auth/login/')

        # If trying to access executive path but not an executive, redirect
        if request.path.startswith('/executives/') and role != 'executive':
            return redirect('/public/access_denied/')

        # Store role in request for views to use (optional but helpful)
```

```
        request.user_role = role

        # Let the request continue to the next middleware or view
        response = self.get_response(request)
        return response
```

- **This snippet shows the core logic**: it fetches the user's role and, if it doesn't match the requested path (e.g., **/executives/ path for a student**), it redirects them.
- **The self.get_response(request) line is critical**: it passes the request down the chain of **middlewares** and eventually to the view that generates the page.

2. Preventing Brute Force Attacks with RateLimitMiddleware

To prevent someone from overwhelming our server with too many requests, the **RateLimitMiddleware** steps in. It tracks each unique user's IP address.

If a user sends more than, say, 60 requests within a 60-second window, this middleware automatically blocks any further requests from that IP for a short period (e.g., 5 minutes), returning a "**Too Many Requests**" error. This protects our server from denial-of-service attacks or excessive automated scraping.

```
class RateLimitMiddleware(MiddlewareMixin):
    RATE_LIMIT = 60         # Max requests in a window
    WINDOW = 60             # Window duration in seconds
    BLOCK_TIME = 300        # How long to block in seconds

    def process_request(self, request):
        ip = self.get_ip(request) # Get user's IP address

        block_key = f"block:{ip}"
        if cache.get(block_key): # Check if already blocked
            return JsonResponse({"error": "Too many requests. You are blocked."}, status=429)

        req_key = f"req:{ip}"
        req_data = cache.get(req_key, {"count": 0, "start": time.time()})

        # Logic to count requests within the window
        if (time.time() - req_data["start"]) < self.WINDOW:
            req_data["count"] += 1
        else: # Reset window if expired
            req_data = {"count": 1, "start": time.time()}

        cache.set(req_key, req_data, timeout=self.WINDOW)

        if req_data["count"] > self.RATE_LIMIT: # If limit exceeded, block
            cache.set(block_key, True, timeout=self.BLOCK_TIME)
            return JsonResponse({"error": "Too many requests. You are blocked."}, status=429)

        return None # Let request continue if not rate-limited

    def get_ip(self, request):
        # Helper to get IP, even if behind a proxy
        x_forwarded_for = request.META.get("HTTP_X_FORWARDED_FOR")
        if x_forwarded_for: return x_forwarded_for.split(",")[0].strip()
        return request.META.get("REMOTE_ADDR")
```

- The **process_request** method is called for every incoming request.
- It uses Django's cache to store the request count for each ip address.

- If the count exceeds **RATE_LIMIT** within the **WINDOW**, it **JsonResponses** with an error and blocks the **IP** for **BLOCK_TIME**.
- return None means the middleware allows the request to pass to the next middleware or view.

3. Enhancing User Experience with ImgFallbackMiddleware

After a page is generated and before it's sent back to the user's browser, the **ImgFallbackMiddleware** can modify the response. It injects a small **JavaScript** code snippet into every HTML page. This script automatically replaces any broken **<img>** tags with a default placeholder image, making the website look better even if some images fail to load.

```
class ImgFallbackMiddleware:
    # ... (init omitted for brevity) ...

    def __call__(self, request):
        response = self.get_response(request) # Get the response from views/other middlewares

        # Only inject script into HTML responses
        if isinstance(response, TemplateResponse) or (isinstance(response, HttpResponse) and
response.get('Content-Type', '').startswith('text/html')):
            # This part runs after the page content is generated
            self.inject_script(response, request)
        return response

    def inject_script(self, response, request):
        # Find where to inject in the <head> of the HTML
        head_index = response.content.lower().find(b'<head>')
        if head_index != -1:
            insert_index = head_index + len(b'<head>')
            script = b"""
            <script>
            document.addEventListener("DOMContentLoaded", function() {
                document.querySelectorAll("img").forEach(img => {
                    img.addEventListener("error", function handler() {
                        this.removeEventListener("error", handler);
                        this.src = "https://img.freepik.com/free-vector/blue-circle-with-white-user_78370-
4707.jpg"; // Default image URL
                    });
                });
            });
            </script>
            """
            # Inject the script into the response content
            response.content = (response.content[:insert_index] + script +
                                response.content[insert_index:])
            response['Content-Length'] = len(response.content) # Update content length

        return response
```

- The ***__call__*** method passes the request, gets a response, and then, if it's an HTML response, calls **inject_script**.
- **inject_script** modifies the **response.content (the HTML)** to add a **<script>** tag that handles image errors. This is a "response middleware."

4. Protecting Against Large Payloads with PayloadSizeLimitMiddleware

Finally, to guard against very large file uploads or data submissions that could consume too many server resources, the **PayloadSizeLimitMiddleware** checks the size of incoming **POST**, **PUT**, or **PATCH** requests.
If a request's **CONTENT_LENGTH** (the size of its data) exceeds a predefined limit (e.g., 50 MB), this middleware rejects it immediately with a "**Payload Too Large**" error.

```
class PayloadSizeLimitMiddleware(MiddlewareMixin):
    MAX_PAYLOAD_SIZE = 50 * 1024 * 1024  # 50 MB in bytes

    def process_request(self, request):
        if request.method in ("POST", "PUT", "PATCH"):
            content_length = request.META.get("CONTENT_LENGTH")
            if content_length:
                try:
                    content_length = int(content_length)
                except (ValueError, TypeError):
                    return JsonResponse({"error": "Invalid Content-Length."}, status=400)
                if content_length > self.MAX_PAYLOAD_SIZE:
                    return JsonResponse(
                        {"error": f"Payload too large. Max {self.MAX_PAYLOAD_SIZE / (1024*1024)} MB."},
                        status=413 # HTTP 413 Payload Too Large
                    )
        return None # Let request continue if size is okay
```

- The **process_request** method is called for relevant request types (**POST**, **PUT**, **PATCH**).
- It checks the **CONTENT_LENGTH** header.
- If the length exceeds **MAX_PAYLOAD_SIZE**, it returns a **JsonResponse** error.
- return None indicates the request is allowed to proceed.

3.7.4 Diving into the Code: Enabling Middleware

To make these custom **middlewares** work, they need to be listed in your Django project's **settings.py** file, within the **MIDDLEWARE** list. The order of middlewares is crucial because they execute in the order they are listed for incoming requests, and in reverse order for outgoing responses.
Here's how they are configured in **course_management/settings.py**:

```
# course_management/settings.py (simplified)
# ... other settings ...

INSTALLED_APPS = [
    'django.contrib.admin',
    # ... other default Django apps ...
    'authorization', # Our authorization app where middlewares are defined
    # ... other project apps ...
]

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
```

```
    # My custom middlewares (order is important!)
    'authorization.middleware.UserRoleMiddleware',
    'authorization.middleware.ImgFallbackMiddleware',
    'authorization.middleware.RateLimitMiddleware',
    'authorization.middleware.PayloadSizeLimitMiddleware',
]

# ... rest of settings ...
```

- **INSTALLED_APPS**: The authorization app must be listed here because our custom middlewares are defined within it.
- **MIDDLEWARE**: This is the list where we add our custom middleware classes, using their full Python path (*e.g., 'authorization.middleware.UserRoleMiddleware'*).
- Order Matters:
    - **UserRoleMiddleware** is placed early to handle authentication and redirection quickly.
    - **ImgFallbackMiddleware** needs to operate on the final HTML content, so it's often placed after middlewares that modify the response's content.
    - **RateLimitMiddleware** and **PayloadSizeLimitMiddleware** are placed early to quickly reject invalid/abusive requests before they consume more server resources.

By carefully ordering these middlewares, our ums system ensures that every request is securely handled, protected from abuse, and provides a polished user experience.

VOLUME IV
PROJECT MANAGEMENT

4.1 SDLC Model Used

For this project, we did **not strictly follow any traditional Software Development Life Cycle (SDLC)** model such as Agile, Waterfall, or Spiral. Instead, our workflow was flexible and adaptive, carried out in segments. We can call it a "**hybrid**" approach. To some extent, it had agile-like characteristics, but it was not completely agile. Whenever changes were required, we incorporated them **on the go without being bound to a rigid framework**. To track progress, we used a **Kanban Board** to some extent, which allowed us to visualize tasks and ensure that development continued to move forward.

i. This approach turned out to be highly effective for our project because:
ii. It allowed us to adapt quickly to new requirements.
iii. We were not constrained by a fixed methodology.
iv. The team was able to respond to unexpected issues dynamically.

If we had adhered strictly to one framework, implementing new requirements or changes would have been more complicated and less efficient. Our flexible, **"hybrid"** approach fits well with the evolving nature of this project.

4.2 Time Taken & Task Breakdown

The initial **idea** was **conceived in May 2025**. During this period, brainstorming was carried out, and potential members were approached. The actual extensive project phase officially began **in the first week of July 2025**.

Development continued steadily, and the project was completed by the **third week of September 2025**. In total, the system was developed in approximately two months.

We did not strictly divide tasks by time phases (such as completing UI/UX in the first month and backend in the second). Instead, frontend and backend development were carried out in parallel. Frontend developers focused on UI/UX while backend developers implemented functionality. This parallel approach helped us achieve faster speed and better synchronization between the two layers of the project.

4.3 Team Members & Responsibilities

4.3.1 Core Team Members

1. Moe Thiha (Semester VI – KE)

Role: Leader, Idea Owner, Project Manager, Main Backend Developer, Partial UI Designer

Responsibilities:

- Provided the **original product idea** and defined the project vision.
- Guided and supported all members throughout the project lifecycle.
- Designed and managed the **entire database system**, covering ER modeling, normalization, query design, ORM integration, optimization, deployment, and regular backups.
- Managed **backend development primarily** and ensured seamless system integration with the data tier.

- Contributed to **UI/UX design decisions** and ensured consistency across the system.
- Acted as the **main pillar of collaboration**, leading meetings, facilitating requirement analysis, and coordinating documentation.
- Oversaw and managed the entire **GitHub collaboration workflow**, ensuring smooth version control and proper team contributions.

2. Ye Man Aung (Semester VI – KE)

Role: UI/UX Developer, Partial Backend Developer

Responsibilities:

- Designed and developed the system's **UI/UX**, ensuring an intuitive and user-friendly interface.
- Implemented specific **backend functionalities**, including modules such as **SISForm**.
- Assisted with **sample data population** for testing and demonstration purposes.

3. Nyein Min Htun (Semester VI – KE)

Role: UI/UX Developer, Data Population Specialist

Responsibilities:

- Focused on **UI/UX development**, contributing to the design and usability of the system.
- Took primary responsibility for **entering and managing sample data** within the system, assisted by other members as needed.
- Led the **manual testing process**, verifying system functionality, identifying issues, and ensuring overall quality.

4. Htoo Aung Lin (Semester VI – KE)

Role: UI/UX Developer, Backend Developer

Responsibilities:

- Developed **frontend components** to enhance system usability and visual consistency.
- Implemented key **backend functionalities**, ensuring proper integration with the overall system.
- Took full ownership of the **Noticeboard Module**, handling both its **frontend and backend development**.

4.3.2 Acknowledged Contributors

1. Thiri Han (Semester VI – SE)

Role: Initial UI Lead

Responsibilities:

- Served as the **UI Lead** during the first month of development.
- Designed the **frontend color scheme** and established the overall **design framework** for the project.

- Although unable to continue due to a busy schedule, she laid an **important design foundation** that guided subsequent UI/UX development.

2. Swam Pyae Aung (Semester IV – KE)

Role: Frontend Contributor
Responsibilities:

- Contributed during the **first month of development**, focusing on frontend work.
- Developed **key frontend components** that later became essential to the system's interface and functionality.
- Although contributions were limited in duration, they played a **valuable role in establishing the foundation** of the project's frontend.

4.4 Difficulties Faced During Development

Throughout development, we encountered several challenges, both technical and non-technical:

4.4.1 Technical Challenges

1. **Merge Conflicts**
   - Since frontend and backend were developed in parallel, frequent merge conflicts occurred.
   - Initially, resolving these consumed significant time, but the team later developed strategies to minimize them.
2. **Database Consistency Issues**
   - At the beginning, each member used a **local MySQL database**, which caused data inconsistencies.
   - This was resolved by deploying a **centralized shared database**, ensuring uniformity.
3. **Free Trial Hosting Limitations**
   - The database was hosted on **Render (free trial)**, which expired after one month.
   - Data occasionally disappeared unexpectedly, requiring frequent **manual backups** to prevent loss.
   -
4. **Large Sample Data Requirement**
   - The system required significant amounts of data.
   - Data collection involved surveying students, but many were initially hesitant due to **cybersecurity concerns.**
   - The team had to build trust before successfully collecting sufficient data.

4.4.2 Non-Technical Challenges

1. **Time Management for the Leader**
   - The project leader, also serving as the main backend developer, was heavily occupied with integration and coding.
   - This reduced the time available for **project management**, occasionally causing communication gaps and tracking delays.

2. **Internet Connectivity Issues**
    - Unstable internet in the team's local area slowed **collaboration, synchronization, and deployment** efforts.
3. **Parallel Development Coordination**
    - Developing frontend and backend simultaneously led to coordination challenges.
    - Aligning **interfaces and feature requirements** required extra effort and communication.

4.5 Limitations

1. **Batch Size Restrictions** – Each batch can only be assigned **one instructor and one classroom**, making it difficult to manage large groups.
2. **Interdisciplinary Courses** – The system does not support **joint instruction from multiple departments**, which is often required for interdisciplinary subjects.
3. **Re-Examination Feature** – If a student fails an exam, they must **retake the entire subject** in the next enrollment; no supplementary or re-exam option is available.
4. **Instructor Changes During Term** – If an instructor **leaves, transfers, or retires** in the middle of an academic term, the system currently **cannot handle reassignment or continuity**, creating gaps in class management.
5. **Time Consumption during Result Sending** – If an executive sends results to a term, the system will send results to all the students enrolled in each batch in that term. But this consumes a lot of time and resource. Also, if there is any error like system error or internet dysconnectivity, the system will stop the result sending there. This is not efficient for a university with a large population of students.
6. **Language Support** – The platform supports only English and lacks multilingual features.
7. **Mobile Accessibility** – No dedicated **mobile application** is available, limiting usability on mobile devices. One can use it on a phone via any browser.

4.6 Further Implementation

To enhance the system's performance and extend its capabilities, the following improvements are planned:

- **Query Optimization** – Improve efficiency and speed of database queries for smoother performance with larger datasets.
- **Flexible Batch and Instructor Assignment** – Enable assignment of multiple instructors and dynamic classroom allocation, improving management of large or interdisciplinary classes.
- **Timetable Generation** – Implement **an automated timetable generation system** that schedules courses, instructors, and classrooms while respecting constraints such as batch size, instructor availability, and room capacity.
- **Facial Recognition for Attendance** – Integrate class participation tracking through **facial recognition technology** for automated attendance and engagement analysis.

CONCLUSION

In conclusion, the University Management System developed in this project serves as a comprehensive and efficient solution for managing the complex operations of a modern educational institution. By integrating modules for student information, faculty management, course scheduling, and examination handling, the system significantly reduces manual workload, minimizes errors, and enhances overall data accuracy. The implementation of relational databases ensures the integrity and security of sensitive information, while the user-friendly interface allows administrators, faculty, and students to interact seamlessly with the system. Additionally, the system's scalability and modular design provide the flexibility to incorporate future enhancements, such as automated timetable generation, advanced analytics, and mobile accessibility. Overall, this project not only demonstrates the practical application of software engineering principles, database management, and web development but also contributes to streamlining academic processes, improving communication, and supporting informed decision-making within the university. The University Management System thus stands as a valuable tool that aligns with the goals of modern educational institutions, promoting efficiency, transparency, and sustainable growth.