NORTH SOUTH UNIVERSITY

School of Engineering & Physical Sciences

Project Name:

OpenEdu: A Comprehensive Open-Source Learning Platform

Submitted To:

Department of Electrical & Computer Engineering (ECE)
Course Instructor: AKM Iqtidar Newaz [IQN]
Course: Software Engineering
Course ID: CSE327
Section: 8
Semester: Spring 2025

Submitted By:

Group Member's Name	ID
Moontaha Rawshan	2221035642
Mohammad Ishzaz Asif Rafid	2221370642
Mujtaba Muhammad Abrar	2122265642

OpenEdu: A Comprehensive Open-Source Learning Platform

Abstract

This project report presents OpenEdu, a scalable and user-centric open-source learning platform designed to facilitate seamless content creation, course management, and academic collaboration. With role-based access control at its core, the system supports diverse users from students and faculty to administrators and master users enabling tailored functionalities for each. The platform is built on a modular Python-based architecture, implementing various software design patterns such as Singleton, Strategy, Factory, and Facade to ensure scalability, maintainability, and flexibility. Emphasis is placed on non-functional requirements like usability, security, data integrity, and interoperability with institutional systems. By combining thoughtful system architecture with dynamic role management, OpenEdu aims to redefine digital learning experiences in academic environments.

Introduction

The digital transformation of education has highlighted the need for robust, flexible, and accessible learning management systems. OpenEdu is a comprehensive open-source learning platform designed to meet the evolving requirements of modern educational institutions. Developed as part of the Software Engineering course (CSE327) at North South University, this project integrates user management, course and content handling, department structuring, and real-time request processing into a unified system.

Unlike traditional systems, OpenEdu employs advanced software engineering principles and design patterns to ensure system reliability and maintainability. With dynamic user roles—Regular User, Moderator, Admin, and Master—the platform efficiently manages permissions and responsibilities. This document outlines the functional and non-functional requirements, the system's core architecture, implemented design patterns, and class/sequence diagrams that collectively form the backbone of the OpenEdu ecosystem.

Functional Requirements

Proposed:

1. User Management:

- Users can be assigned roles (Regular User, Faculty, Moderator, Admin, Master).

- Users can request access to specific functionalities (e.g., content approval, appointment requests).

- Moderators and admins can manage approval workflows for user requests.

2. Course and Content Management:

- Courses can be created and assigned to faculties.

- Regular Users can upload lectures, videos, and notes.

- Students can access and interact with available course materials.

3. Request Handling System:

- Users can submit different types of requests (e.g., content approval, appointment requests).

- Moderators and administrators can manage and process these requests.

4. Role-Based Access Control:

- Regular Users can view course materials.

- Admins can oversee courses and faculty appointments.

- Masters (Super Admins) can manage high-level operations such as department structuring.

5. Department and Faculty Management:

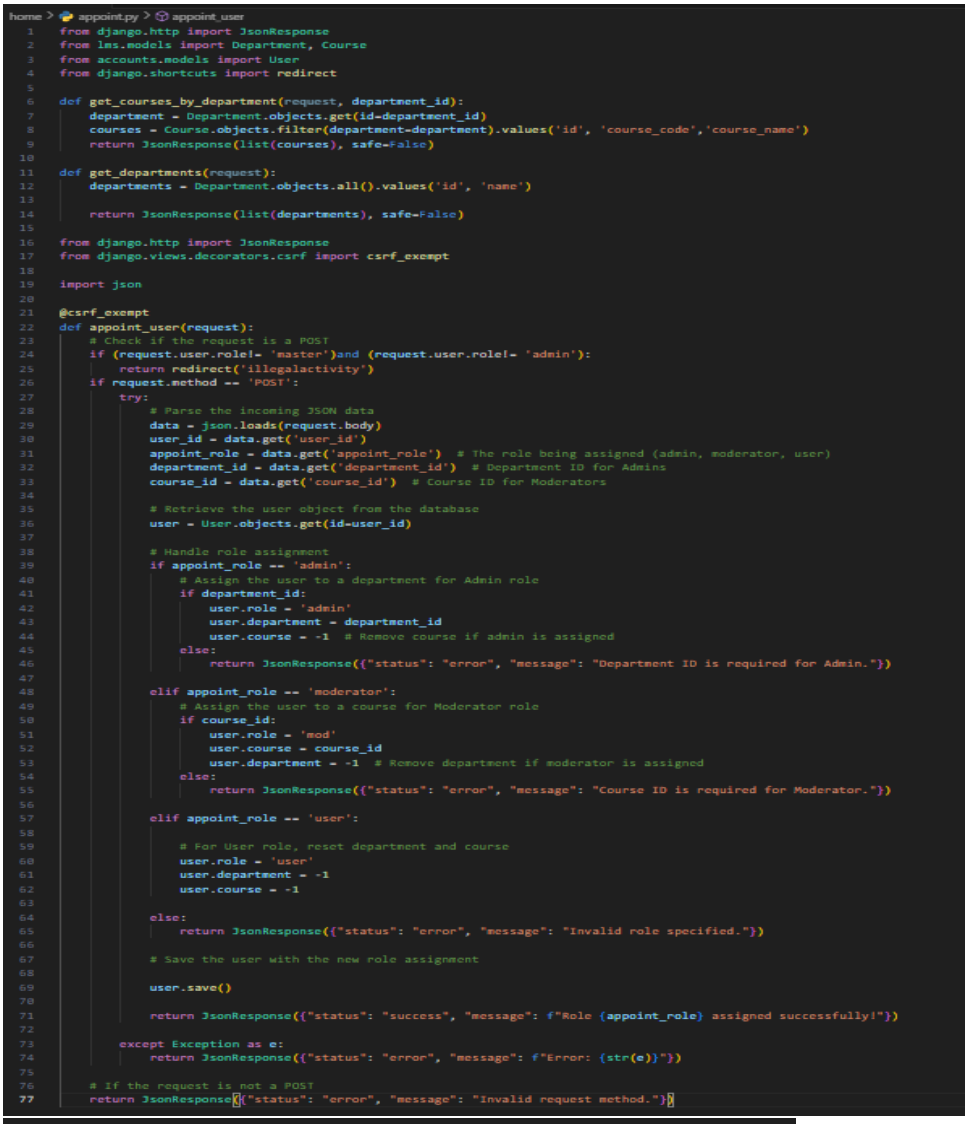
- Admins can organize courses, and faculties under a structured hierarchy.

- Masters can appoint administrators and manage department structures.

Completed:

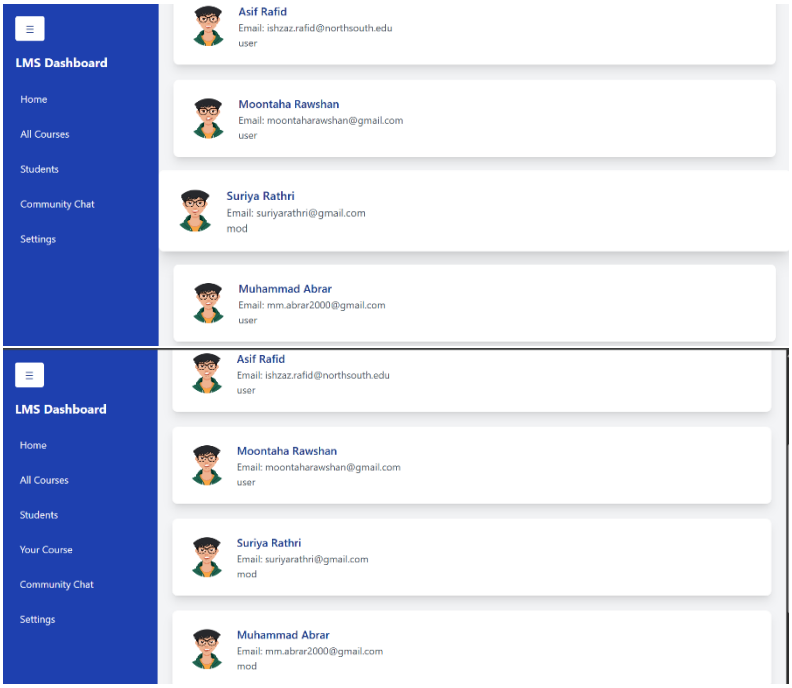
1. User Management:

- Users can be assigned roles (Regular User, Faculty, Moderator, Admin, Master).

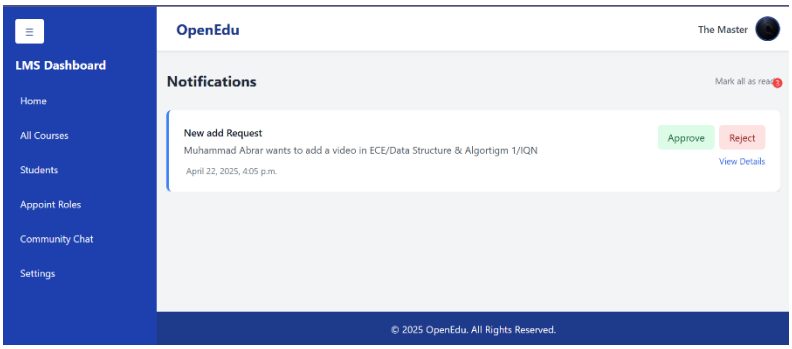


```python
from django.http import JsonResponse
from lms.models import Department, Course
from accounts.models import User
from django.shortcuts import redirect

def get_courses_by_department(request, department_id):
    department = Department.objects.get(id=department_id)
    courses = Course.objects.filter(department=department).values('id', 'course_code','course_name')
    return JsonResponse(list(courses), safe=False)

def get_departments(request):
    departments = Department.objects.all().values('id', 'name')

    return JsonResponse(list(departments), safe=False)

from django.http import JsonResponse
from django.views.decorators.csrf import csrf_exempt

import json

@csrf_exempt
def appoint_user(request):
    # Check if the request is a POST
    if (request.user.role!= 'master')and (request.user.role!= 'admin'):
        return redirect('illegalactivity')
    if request.method == 'POST':
        try:
            # Parse the incoming JSON data
            data = json.loads(request.body)
            user_id = data.get('user_id')
            appoint_role = data.get('appoint_role')  # The role being assigned (admin, moderator, user)
            department_id = data.get('department_id')  # Department ID for Admins
            course_id = data.get('course_id')  # Course ID for Moderators

            # Retrieve the user object from the database
            user = User.objects.get(id=user_id)

            # Handle role assignment
            if appoint_role == 'admin':
                # Assign the user to a department for Admin role
                if department_id:
                    user.role = 'admin'
                    user.department = department_id
                    user.course = -1  # Remove course if admin is assigned
                else:
                    return JsonResponse({"status": "error", "message": "Department ID is required for Admin."})

            elif appoint_role == 'moderator':
                # Assign the user to a course for Moderator role
                if course_id:
                    user.role = 'mod'
                    user.course = course_id
                    user.department = -1  # Remove department if moderator is assigned
                else:
                    return JsonResponse({"status": "error", "message": "Course ID is required for Moderator."})

            elif appoint_role == 'user':

                # For User role, reset department and course
                user.role = 'user'
                user.department = -1
                user.course = -1

            else:
                return JsonResponse({"status": "error", "message": "Invalid role specified."})

            # Save the user with the new role assignment

            user.save()

            return JsonResponse({"status": "success", "message": f"Role {appoint_role} assigned successfully!"})

        except Exception as e:
            return JsonResponse({"status": "error", "message": f"Error: {str(e)}"})

    # If the request is not a POST
    return JsonResponse({"status": "error", "message": "Invalid request method."})
```

```python
@login_required
def appoint(request):
    if (request.user.role!='admin' and request.user.role!='master'):
        return redirect('/errors/unauthorizedaccess')
    admins=User.objects.filter(role='admin')

    moderators=User.objects.filter(role='mod')
    rusers=User.objects.filter(role='user')
    isMaster=(request.user.role=='master')
    context={'name':request.user.username,'rusers':rusers,'admins':admins,'moderators':moderators,'isMaster':isMaster}

    return render(request,"pages/appoint.html",context)
```



- Users can request access to specific functionalities (e.g., content approval, appointment requests).



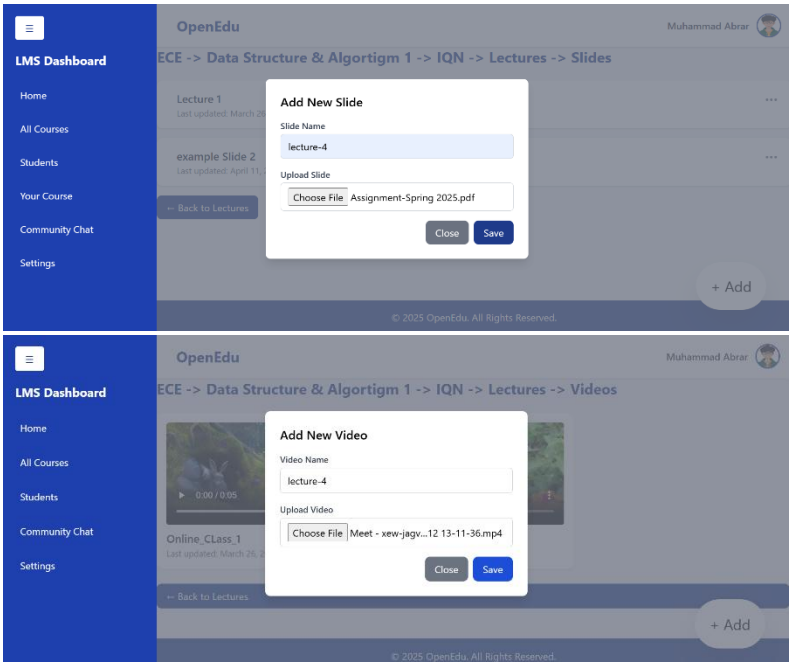
```python
@login_required
def approve_not(request, not_id):

    notification = get_object_or_404(Notification, id=not_id)
    if request.user not in notification.reciever.all():
        return redirect('illegalactivity')

    # Ensure the notification type is 'add', 'update', or 'delete'
    if notification.type in ['add', 'update', 'delete']:
        # Manually map the content_type to corresponding models
        content_map = {
            'slide': (Slide, temp_Slide),
            'video': (Video, temp_Video),
            'note': (Note, temp_Note)
        }

        # Check if the content type is valid and retrieve the models
        if notification.content_type in content_map:
            content_model, temp_content_model = content_map[notification.content_type]
        else:
            raise ValueError(f"Invalid content type '{notification.content_type}' provided.")  # Handle invalid type

        # Handle 'add' logic (no need to query for the object, just create it)
        if notification.type == 'add':
            temp_content = get_object_or_404(temp_content_model, id=notification.content_id)

            # Create the real content instance directly from the temporary content
            real_content = content_model.objects.create(
                name=temp_content.name,
                faculty=temp_content.faculty,
                content=temp_content.content,  # Using content from the temporary content
            )

            # Set the real_content_id in the notification
            notification.real_content_id = real_content.id
            notification.save()

            # Delete the temporary content instance after approval
            temp_content.delete()

        # Handle 'update' logic
        elif notification.type == 'update':
            content = get_object_or_404(content_model, id=notification.real_content_id)

            # Update the content fields using the name and content from the temporary content
            temp_content = get_object_or_404(temp_content_model, id=notification.content_id)  # Get associated temp content
            content.name = temp_content.name  # Update the real content's name
            content.content = temp_content.content  # Update the real content's content
            content.save()  # Save the updated real content to the database

            # Set the real_content_id to the updated content ID
            notification.real_content_id = content.id
            notification.save()

            # Delete the temporary content after updating
            temp_content.delete()

        # Handle 'delete' logic
        elif notification.type == 'delete':
            try:
                content = content_model.objects.get(id=notification.real_content_id)
                # Delete the real content instance
                content.delete()

                # Set the real_content_id to null (since it's deleted)
                Notification.objects.filter(Q(real_content_id=notification.real_content_id) & Q(content_type=notification.content_type)).delete()


            except content_model.DoesNotExist:
                # Handle case if the content to be deleted does not exist

                return redirect("notifications")

        # Create an approval notification for the user who requested it
        Not = Notification.objects.create(
            sender=request.user,

            message=f"Your {notification.type} request for {notification.content_type} has been approved by {request.user.first_name} {request.user.last_name} ({request.user.role})",
            type='Inf'
        )

        # Fix: Add the receiver properly
        reciever = [notification.sender]
        Not.reciever.add(*reciever)  # Use *reciever to unpack the list and add to the ManyToMany field


    notification.delete()

    # Redirect back to the notifications page
    return redirect("notifications")
```

2. Course and Content Management:

- Courses can be created and assigned to faculties.

```python
# Add Course
@login_required
def add_course(request, dept_id):
    if (request.user.role!='master')and(request.user.department!= dept_id):
        return redirect('illegalactivity')
    department=get_object_or_404(Department,id=dept_id)
    if request.method == 'POST':
        course_code = request.POST.get('courseCode')
        course_name = request.POST.get('courseName')
        course_desc = request.POST.get('courseDescription')
        course_image = request.FILES.get('courseImage')
        if(course_code and course_name):
            course=Course.objects.create(course_code=course_code, course_name=course_name,department=department)
            if course_desc:
                course.description=course_desc
            if course_image:
                course.image=course_image
        course.save()
        subject.notify(request, "Course has been added")
    return redirect('deptcourses',department.id)
```

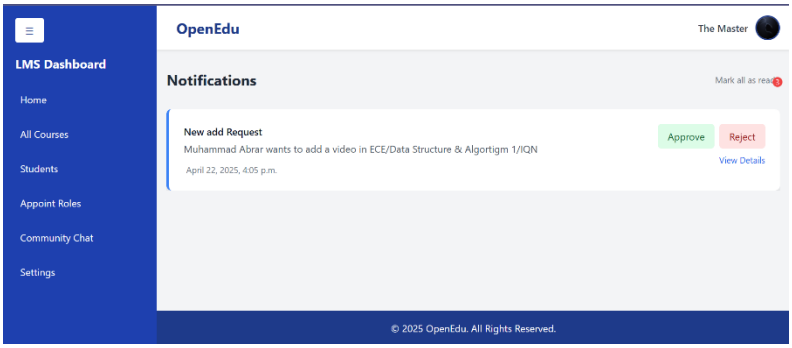
- Regular Users can upload lectures, videos, and notes.



```python
# Add Slide
@login_required
def add_slide(request, dept_id, course_id, fac_id):
    faculty = get_object_or_404(Faculty, id=fac_id)
    if request.method == 'POST':
        slide_name = request.POST.get('slideName')
        slide_content = request.FILES.get('slideContent')
        if (request.user.role == 'master') or (request.user.department == dept_id) or (request.user.course == course_id):
            SlideFactory.create_content(faculty, slide_name, slide_content)
            subject.notify(request, "Slide has been added")
        else:
            temp_slide=TemporarySlideFactory.create_temp_content(real_instance=None, faculty=faculty, name=slide_name, content_file=slide_content)
            notification = Notification.objects.create(
                message=f"{request.user.first_name} {request.user.last_name} wants to add a slide in {temp_slide.faculty.course.department.name}/{temp_slide.faculty.course.course_name}/{temp_slide.faculty.name}",
                sender=request.user,
                type="add",
                content_type="slide",
                content_id=temp_slide.id
            )
            receivers = User.objects.filter(Q(role='master') | Q(department=dept_id) | Q(course=course_id))
            print(receivers)
            notification.recievers.add(*receivers)
            notification.save()
            subject.notify(request, "Request sent")
    return redirect('lec_slides', dept_id, course_id, fac_id)
# Add Note
@login_required
def add_note(request, dept_id, course_id, fac_id):
    faculty = get_object_or_404(Faculty, id=fac_id)
    if request.method == 'POST':
        note_name = request.POST.get('noteName')
        note_content = request.FILES.get('noteContent')
        if (request.user.role == 'master') or (request.user.department == dept_id) or (request.user.course == course_id):
            NoteFactory.create_content( faculty, note_name, note_content)
            subject.notify(request, "Note has been added")
        else:
            temp_note=TemporaryNoteFactory.create_temp_content( real_instance=None, faculty=faculty, name=note_name, content_file=note_content)
            notification = Notification.objects.create(
                message=f"{request.user.first_name} {request.user.last_name} wants to add a note in {temp_note.faculty.course.department.name}/{temp_note.faculty.course.course_name}/{temp_note.faculty.name}",
                sender=request.user,
                type="add",
                content_type="note",
                content_id=temp_note.id
            )
            receivers = User.objects.filter(Q(role='master') | Q(department=dept_id) | Q(course=course_id))
            notification.recievers.add(*receivers)
            notification.save()
            subject.notify(request, "Request sent")
    return redirect('lec_notes', dept_id, course_id, fac_id)


# Add Video
@login_required
def add_video(request, dept_id, course_id, fac_id):
    faculty = get_object_or_404(Faculty, id=fac_id)
    if request.method == 'POST':
        video_name = request.POST.get('videoName')
        video_content = request.FILES.get('videoContent')
        if (request.user.role == 'master') or (request.user.department == dept_id) or (request.user.course == course_id):
            VideoFactory.create_content( faculty, video_name, video_content)
            subject.notify(request, "Video has been added")
        else:
            temp_video=TemporaryVideoFactory.create_temp_content( real_instance=None, faculty=faculty, name=video_name, content_file=video_content)
            notification = Notification.objects.create(
                message=f"{request.user.first_name} {request.user.last_name} wants to add a video in {temp_video.faculty.course.department.name}/{temp_video.faculty.course.course_name}/{temp_video.faculty.name}",
                sender=request.user,
                type="add",
                content_type="video",
                content_id=temp_video.id
            )
            receivers = User.objects.filter(Q(role='master') | Q(department=dept_id) | Q(course=course_id))
            notification.recievers.add(*receivers)
            notification.save()
            subject.notify(request, "Request sent")
    return redirect('lec_videos', dept_id, course_id, fac_id)
```

3. Request Handling System:

- Users can submit different types of requests (e.g., content approval, appointment requests).



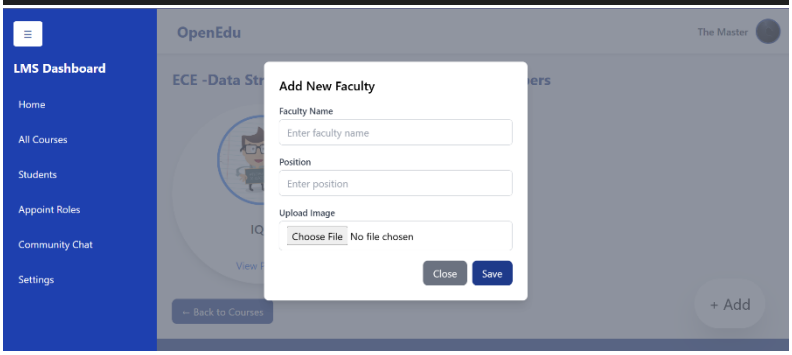
4. Role-Based Access Control:

- Regular Users can view course materials.


```python
#for slides inside a lecture
@login_required
def lec_slides(request,dept_id, course_id,fac_id):
    department = Department.objects.get(id=dept_id)
    course = Course.objects.get(id=course_id)
    faculty = Faculty.objects.get(id=fac_id)
    slides=Slide.objects.filter(faculty=faculty)
    context = {'department': department, 'faculty': faculty,'course':course,'slides':slides,'showSlideModal':True,'showUpdateSlideModal':True,'showA

    return render(request,"lms/slides.html",context)



# For videos inside a lecture
@login_required
def lec_videos(request, dept_id, course_id, fac_id):
    department = Department.objects.get(id=dept_id)
    course = Course.objects.get(id=course_id)
    faculty = Faculty.objects.get(id=fac_id)
    videos = Video.objects.filter(faculty=faculty)
    context = {'department': department, 'faculty': faculty, 'course': course, 'videos': videos,'showVideoModal':True,'showUpdateVideoModal':True,'s
    return render(request, "lms/videos.html", context)



# For notes inside a lecture
@login_required
def lec_notes(request, dept_id, course_id, fac_id):
    department = Department.objects.get(id=dept_id)
    course = Course.objects.get(id=course_id)
    faculty = Faculty.objects.get(id=fac_id)
    notes = Note.objects.filter(faculty=faculty)
    context = {'department': department, 'faculty': faculty, 'course': course, 'notes': notes,'showNoteModal':True,'showUpdateNoteModal':True,'showA
    return render(request, "lms/notes.html", context)
```

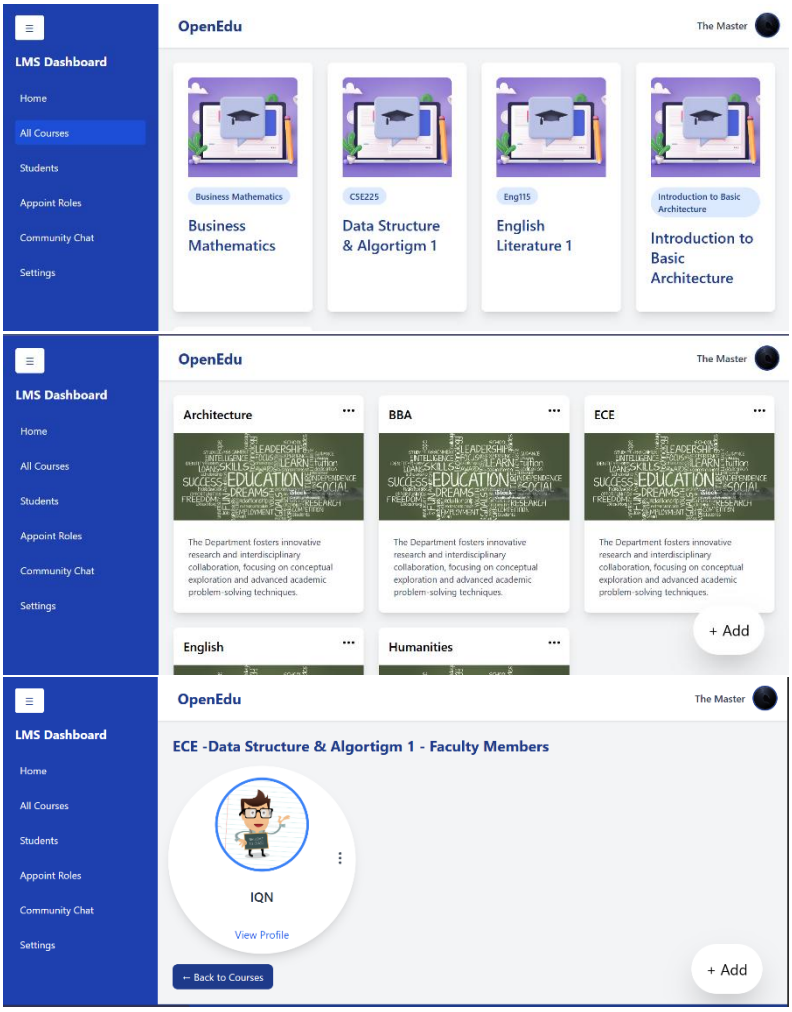
- Admins can add faculty appointments.

```python
# Add Faculty
@login_required
def add_fac(request, dept_id, course_id):
    if (request.user.role!='master')and(request.user.department!= dept_id)and(request.user.course!= course_id):
        return redirect('illegalactivity')
    course = get_object_or_404(Course, id=course_id)
    if request.method == 'POST':
        faculty_name = request.POST.get('facultyName')
        position = request.POST.get('position')
        image= request.FILES.get('facultyImage')
        if(faculty_name and position):
            faculty=Faculty.objects.create(name=faculty_name, position=position,course=course)
            if image:
                faculty.image=image
            faculty.save()
        subject.notify(request, "Faculty has been added")
    return redirect('course_facs',dept_id,course_id)
```



5. Department and Faculty Management:

- Admins can organize courses, and faculties under a structured hierarchy.



```python
from django.shortcuts import render,redirect
from django.contrib.auth.decorators import login_required
from .models import Department,Course,Faculty,Slide,Video,Note
from django.conf import settings
from .contentViewers import *


@login_required
# Create your views here.
def departments(request):
    departments=Department.objects.all().order_by('name')
    showDeptModal=(request.user.role=='master')
    showUpdateDeptModal=(request.user.role=='master')
    showAddButton=(request.user.role=='master')
    context={'name':request.user.username,'departments':departments,
             'showDeptModal':showDeptModal,'showUpdateDeptModal':showUpdateDeptModal,'showAddButton':showAddButton}

    return render(request,'lms/departments.html',context)

@login_required
def courses(request):
    courses = Course.objects.all().order_by('course_name')

    context={'name':request.user.username,'courses':courses}

    return render(request,"courses.html",context)

# Create your views here.

#for Courses inside a Department
@login_required
def deptcourses(request,id):
    department=Department.objects.get(id=id)
    courses = Course.objects.filter(department=department).order_by('course_name')
    showAddButton=(request.user.role=='master')or(request.user.department==department.id)
    showUpdateCourseModal=(request.user.role=='master')or(request.user.department==department.id)
    showCourseModal=(request.user.role=='master')or(request.user.department==department.id)
    context={'name':request.user.username,'courses':courses , 'department': department,
             'showCourseModal':showCourseModal,'showUpdateCourseModal':showUpdateCourseModal,'showAddButton':showAddButton}
    return render(request,'lms/deptcourses.html',context)



#for faculties inside a Course
@login_required
def course_facs(request, dept_id, course_id):
    department = Department.objects.get(id=dept_id)
    course = Course.objects.get(id=course_id)
    faculties = Faculty.objects.filter(course=course)
    showFacultyModal=(request.user.role=='master')or(request.user.department==department.id)or(request.user.course==course.id)
    showUpdateFacultyModal=(request.user.role=='master')or(request.user.department==department.id)or(request.user.course==course.id)
    showAddButton=(request.user.role=='master')or(request.user.department==department.id)or(request.user.course==course.id)
    context = {'department': department, 'faculties': faculties,'course':course,
               'showFacultyModal':showFacultyModal,'showUpdateFacultyModal':showUpdateFacultyModal,'showAddButton':showAddButton}
    return render(request, 'lms/faculty.html', context)



#for lectures inside a faculty
@login_required
def fac_lecs(request, dept_id, course_id,fac_id):
    department = Department.objects.get(id=dept_id)
    course = Course.objects.get(id=course_id)
    faculty = Faculty.objects.get(id=fac_id)
    context = {'department': department, 'faculty': faculty,'course':course,'MEDIA_URL':settings.MEDIA_URL}
    return render(request, 'lms/lectures.html', context)

#for slides inside a lecture
@login_required
def lec_slides(request,dept_id, course_id,fac_id):
    department = Department.objects.get(id=dept_id)
    course = Course.objects.get(id=course_id)
    faculty = Faculty.objects.get(id=fac_id)
    slides=Slide.objects.filter(faculty=faculty)
    context = {'department': department, 'faculty': faculty,'course':course,'slides':slides,'showSlideModal':True,'showUpdateSlideModal':True,'showAddButton':True}

    return render(request,"lms/slides.html",context)



# for videos inside a lecture
@login_required
def lec_videos(request, dept_id, course_id, fac_id):
    department = Department.objects.get(id=dept_id)
    course = Course.objects.get(id=course_id)
    faculty = Faculty.objects.get(id=fac_id)
    videos = Video.objects.filter(faculty=faculty)
    context = {'department': department, 'faculty': faculty, 'course': course, 'videos': videos,'showVideoModal':True,'showUpdateVideoModal':True,'showAddButton':True}
    return render(request, "lms/videos.html", context)



# for notes inside a lecture
@login_required
def lec_notes(request, dept_id, course_id, fac_id):
    department = Department.objects.get(id=dept_id)
    course = Course.objects.get(id=course_id)
    faculty = Faculty.objects.get(id=fac_id)
    notes = Note.objects.filter(faculty=faculty)
    context = {'department': department, 'faculty': faculty, 'course': course, 'notes': notes,'showNoteModal':True,'showUpdateNoteModal':True,'showAddButton':True}
    return render(request, "lms/notes.html", context)
```

Non-Functional

Proposed:

1. Product Requirements:

- Performance: The system must handle many concurrent users (e.g., students, faculty, and admins) without significant latency.

- Reliability: The system should have an acceptable failure rate, with a maximum downtime of 1 hour per month.

- Usability: The system should be intuitive and easy for all user roles (students, moderators, admins, and masters).

- Portability: The system should be accessible across different devices (desktop, mobile, tablet) and operating systems (Windows, macOS, Linux, iOS, Android).

2. Organizational Requirements:

- Process Standards: The system must adhere to the organization's policies and procedures for content management and user roles.

- Implementation Requirements: The system should be developed using Python programming.

- Delivery Requirements: The system and its documentation must be delivered within 16 April 2025.

3. External Requirements:

- Interoperability: The system should interact seamlessly with external systems (e.g., student information systems).

- Legislative Requirements: The system must comply with data protection laws to ensure user data privacy and security. Users can't upload any illegal material.

- Ethical Requirements: The system should ensure that all content and interactions are ethical and acceptable to users and the general public (e.g., no offensive or inappropriate content).

4. Security:

- The system must implement robust security measures to protect user data and prevent unauthorized access (e.g., encryption, role-based access control).

5. Maintainability:

- The system should be easy to maintain, with clear documentation and modular code structure to facilitate updates and bug fixes.

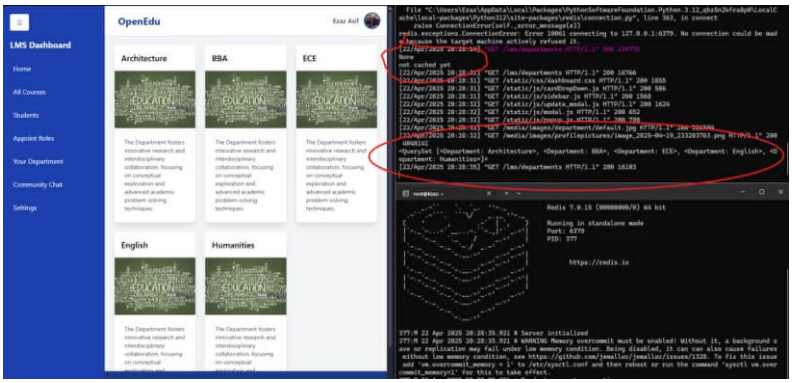
6. Data Integrity:

- The system must ensure data integrity, with mechanisms to prevent data corruption or loss (e.g., regular backups and transaction logs).

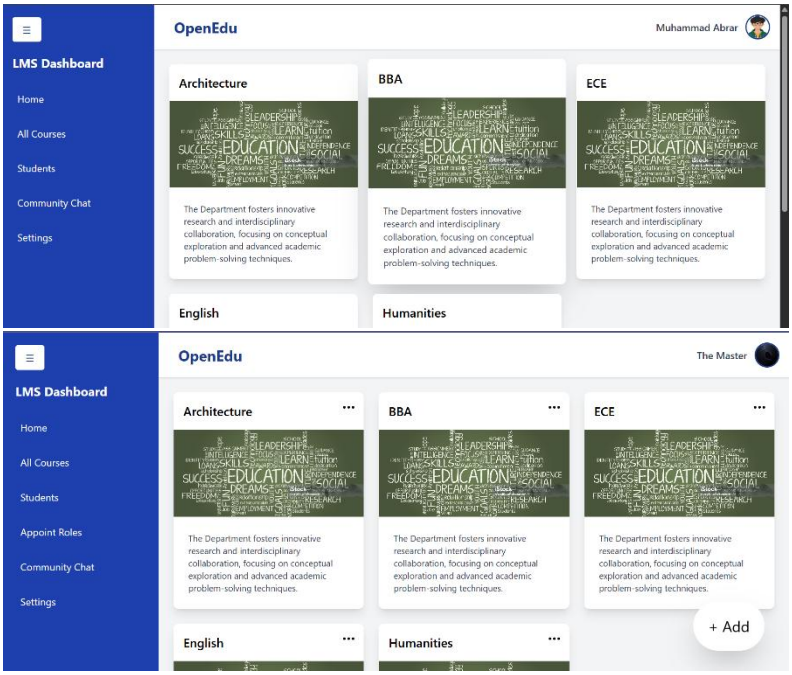
Completed:

1. Product Requirements:

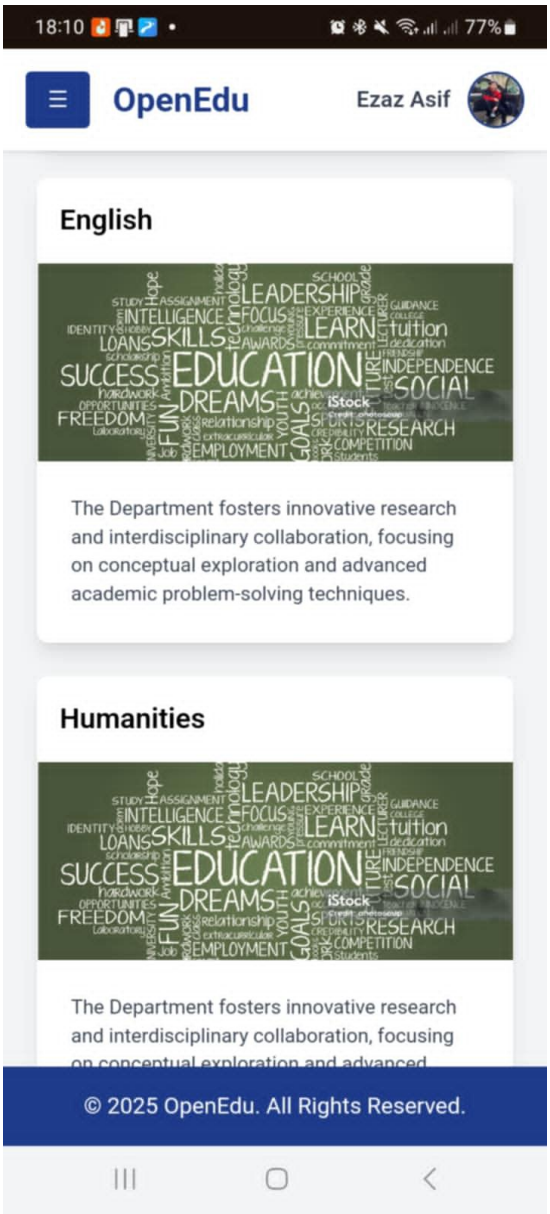
- Performance: The system must handle many concurrent users (e.g., students, faculty, and admins) without significant latency.



- Usability: The system should be intuitive and easy for all user roles (students, moderators, admins, and masters).

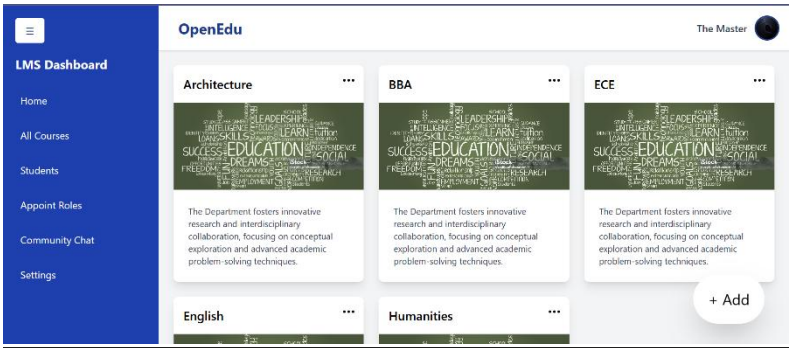


- Portability: The system should be accessible across different devices and operating systems(Windows, macOS, Linux, iOS, Android).



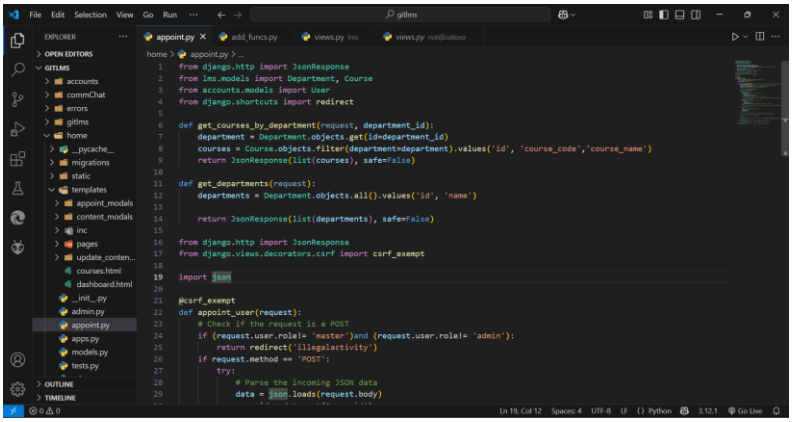
2. Organizational Requirements:

- Process Standards: The system must adhere to the organization's policies and procedures for content management and user roles.



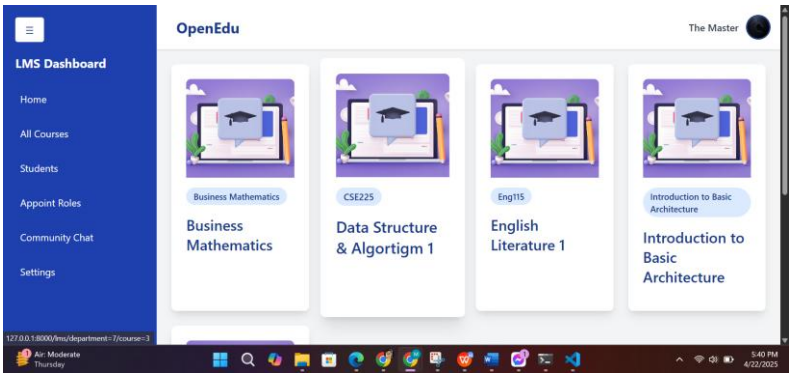

```python
# Add Course
@login_required
def add_course(request, dept_id):
    if (request.user.role!='master')and(request.user.department!= dept_id):
        return redirect('illegalactivity')
    department=get_object_or_404(Department,id=dept_id)
    if request.method == 'POST':
        course_code = request.POST.get('courseCode')
        course_name = request.POST.get('courseName')
        course_desc = request.POST.get('courseDescription')
        course_image = request.FILES.get('courseImage')
        if(course_code and course_name):
            course=Course.objects.create(course_code=course_code, course_name=course_name,department=department)
            if course_desc:
                course.description=course_desc
            if course_image:
                course.image=course_image
        course.save()
        subject.notify(request, "Course has been added")
    return redirect('deptcourses',department.id)
```

- Implementation Requirements: The system should be developed using Python programming.



3. External Requirements:

- Interoperability: The system should interact seamlessly with external systems (e.g., student information systems).



- Legislative Requirements: The system must comply with data protection laws to ensure user data privacy and security. Users can't upload any illegal material.

```
errors > views.py > illegalactivity
1  from django.shortcuts import render
2
3  # Create your views here.
4  def unauthorizedaccess(request):
5      return render(request,"unauthorizedaccess.html")
6
7  def illegalactivity(request):
8      return render(request,"illegalactivity.html")
```

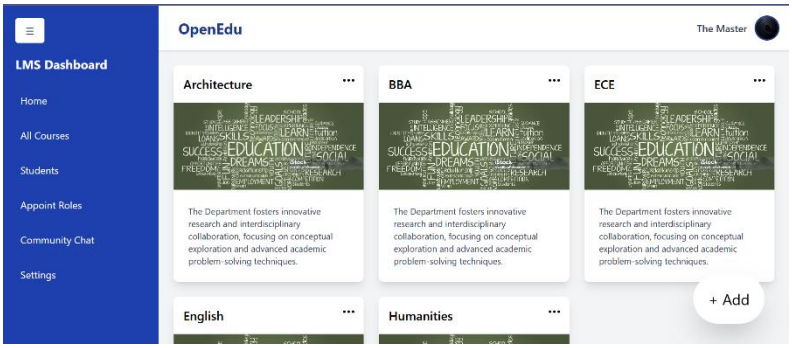
- Ethical Requirements: The system should ensure that all content and interactions are ethical and acceptable to users and the general public (e.g., no offensive or inappropriate content).

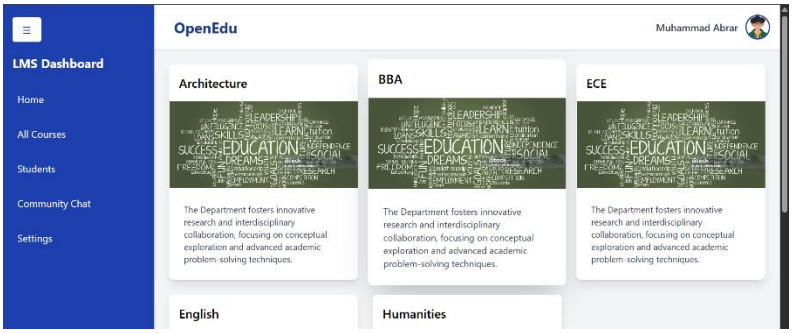
4. Security:

- The system must implement robust security measures to protect user data and prevent unauthorized access (e.g., encryption, role-based access control).


```
errors > views.py > illegalactivity
1  from django.shortcuts import render
2
3  # Create your views here.
4  def unauthorizedaccess(request):
5      return render(request,"unauthorizedaccess.html")
6
7  def illegalactivity(request):
8      return render(request,"illegalactivity.html")
```

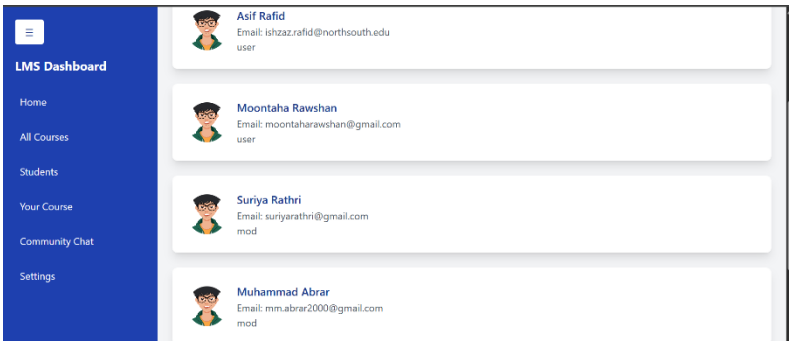
```python
# Add Department
@login_required
def add_dept(request):
    if (request.user.role!='master'):
        return redirect('illegalactivity')
    if request.method == 'POST':
        dept_name = request.POST.get('departmentName')
        dept_desc = request.POST.get('departmentDescription')
        dept_image = request.FILES.get('departmentImage')
        if(dept_name  ):
            department=Department.objects.create( name=dept_name   )
            if(dept_image):
                department.image=dept_image
            if( dept_desc):
                department.description=dept_desc
        department.save()
        subject.notify(request, "Department has been added")
        return redirect('departments')

    return redirect('departments')

# Add Course
@login_required
def add_course(request, dept_id):
    if (request.user.role!='master')and(request.user.department!= dept_id):
        return redirect('illegalactivity')
    department=get_object_or_404(Department,id=dept_id)
    if request.method == 'POST':
        course_code = request.POST.get('courseCode')
        course_name = request.POST.get('courseName')
        course_desc = request.POST.get('courseDescription')
        course_image = request.FILES.get('courseImage')
        if(course_code and course_name):
            course=Course.objects.create(course_code=course_code, course_name=course_name,department=department)
            if course_desc:
                course.description=course_desc
            if course_image:
                course.image=course_image
        course.save()
        subject.notify(request, "Course has been added")
    return redirect('deptcourses',department.id)



# Add Faculty
@login_required
def add_fac(request, dept_id, course_id):
    if (request.user.role!='master')and(request.user.department!= dept_id)and(request.user.course!= course_id):
        return redirect('illegalactivity')
    course = get_object_or_404(Course, id=course_id)
    if request.method == 'POST':
        faculty_name = request.POST.get('facultyName')
        position = request.POST.get('position')
        image= request.FILES.get('facultyImage')
        if(faculty_name and position):
            faculty=Faculty.objects.create(name=faculty_name, position=position,course=course)
            if image:
                faculty.image=image
            faculty.save()
        subject.notify(request, "Faculty has been added")
    return redirect('course_facs',dept_id,course_id)
```

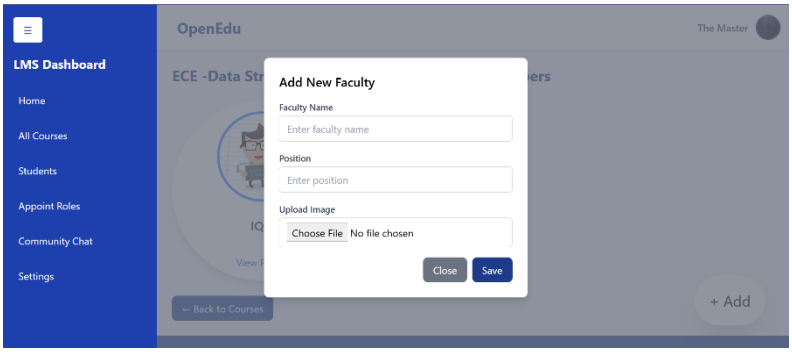




5. Maintainability:

- The system should be easy to maintain, with clear documentation and modular code structure to facilitate updates and bug fixes.

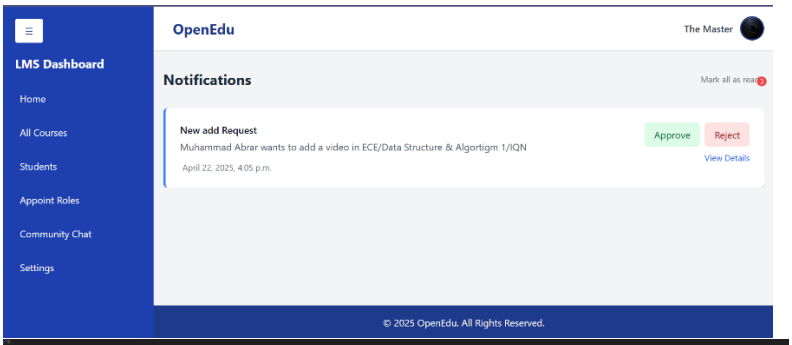




home > appoint.py > appoint_user

```python
from django.http import JsonResponse
from lms.models import Department, Course
from accounts.models import User
from django.shortcuts import redirect

def get_courses_by_department(request, department_id):
    department = Department.objects.get(id=department_id)
    courses = Course.objects.filter(department=department).values('id', 'course_code','course_name')
    return JsonResponse(list(courses), safe=False)

def get_departments(request):
    departments = Department.objects.all().values('id', 'name')

    return JsonResponse(list(departments), safe=False)

from django.http import JsonResponse
from django.views.decorators.csrf import csrf_exempt

import json

@csrf_exempt
def appoint_user(request):
    # Check if the request is a POST
    if (request.user.role!= 'master')and (request.user.role!= 'admin'):
        return redirect('illegalactivity')
    if request.method == 'POST':
        try:
            # Parse the incoming JSON data
            data = json.loads(request.body)
            user_id = data.get('user_id')
            appoint_role = data.get('appoint_role')  # The role being assigned (admin, moderator, user)
            department_id = data.get('department_id')  # Department ID for Admins
            course_id = data.get('course_id')  # Course ID for Moderators

            # Retrieve the user object from the database
            user = User.objects.get(id=user_id)

            # Handle role assignment
            if appoint_role == 'admin':
                # Assign the user to a department for Admin role
                if department_id:
                    user.role = 'admin'
                    user.department = department_id
                    user.course = -1  # Remove course if admin is assigned
                else:
                    return JsonResponse({"status": "error", "message": "Department ID is required for Admin."})

            elif appoint_role == 'moderator':
                # Assign the user to a course for Moderator role
                if course_id:
                    user.role = 'mod'
                    user.course = course_id
                    user.department = -1  # Remove department if moderator is assigned
                else:
                    return JsonResponse({"status": "error", "message": "Course ID is required for Moderator."})

            elif appoint_role == 'user':

                # For User role, reset department and course
                user.role = 'user'
                user.department = -1
                user.course = -1

            else:
                return JsonResponse({"status": "error", "message": "Invalid role specified."})

            # Save the user with the new role assignment

            user.save()

            return JsonResponse({"status": "success", "message": f"Role {appoint_role} assigned successfully!"})

        except Exception as e:
            return JsonResponse({"status": "error", "message": f"Error: {str(e)}"})

    # If the request is not a POST
    return JsonResponse({"status": "error", "message": "Invalid request method."})
```

```python
# Add Faculty
@login_required
def add_fac(request, dept_id, course_id):
    if (request.user.role!='master')and(request.user.department!= dept_id)and(request.user.course!= course_id):
        return redirect('illegalactivity')
    course = get_object_or_404(Course, id=course_id)
    if request.method == 'POST':
        faculty_name = request.POST.get('facultyName')
        position = request.POST.get('position')
        image= request.FILES.get('facultyImage')
        if(faculty_name and position):
            faculty=Faculty.objects.create(name=faculty_name, position=position,course=course)
            if image:
                faculty.image=image
            faculty.save()
        subject.notify(request, "Faculty has been added")
    return redirect('course_facs',dept_id,course_id)
```




```python
@login_required
def approve_not(request, not_id):

    notification = get_object_or_404(Notification, id=not_id)
    if request.user not in notification.recievers.all():
        return redirect('illegalactivity')

    # Ensure the notification type is 'add', 'update', or 'delete'
    if notification.type in ['add', 'update', 'delete']:
        # Manually map the content type to corresponding models
        content_map = {
            'slide': (Slide, temp_Slide),
            'video': (Video, temp_Video),
            'note': (Note, temp_Note)
        }

        # Check if the content type is valid and retrieve the models
        if notification.content_type in content_map:
            content_model, temp_content_model = content_map[notification.content_type]
        else:
            raise ValueError(f"Invalid content type '{notification.content_type}' provided.")  # Handle invalid type

        # Handle 'add' logic (no need to query for the object, just create it)
        if notification.type == 'add':
            temp_content = get_object_or_404(temp_content_model, id=notification.content_id)

            # Create the real content instance directly from the temporary content
            real_content = content_model.objects.create(
                name=temp_content.name,
                faculty=temp_content.faculty,
                content=temp_content.content,  # Using content from the temporary content
            )

            # Set the real_content_id in the notification
            notification.real_content_id = real_content.id
            notification.save()

            # Delete the temporary content instance after approval
            temp_content.delete()

        # Handle 'update' logic
        elif notification.type == 'update':
            content = get_object_or_404(content_model, id=notification.real_content_id)

            # Update the content fields using the name and content from the temporary content
            temp_content = get_object_or_404(temp_content_model, id=notification.content_id)  # Get associated temp content
            content.name = temp_content.name  # Update the real content's name
            content.content = temp_content.content  # Update the real content's content
            content.save()  # Save the updated real content to the database

            # Set the real_content_id to the updated content ID
            notification.real_content_id = content.id
            notification.save()

            # Delete the temporary content after updating
            temp_content.delete()

        # Handle 'delete' logic
        elif notification.type == 'delete':
            try:
                content = content_model.objects.get(id=notification.real_content_id)
                # Delete the real content instance
                content.delete()

                # Set the real_content_id to null (since it's deleted)
                Notification.objects.filter(Q(real_content_id=notification.real_content_id) & Q(content_type=notification.content_type)).delete()


            except content_model.DoesNotExist:
                # Handle case if the content to be deleted does not exist

                return redirect("notifications")

        # Create an approval notification for the user who requested it
        Not = Notification.objects.create(
            sender=request.user,

            message=f"Your {notification.type} request for {notification.content_type} has been approved by {request.user.first_name} {request.user.last_name} ({request.user.role})",
            type='Inf'
        )

        # Fix: Add the receiver properly
        reciever = [notification.sender]
        Not.recievers.add(*reciever)  # Use *reciever to unpack the list and add to the ManyToMany field


    notification.delete()

    # Redirect back to the notifications page
    return redirect("notifications")
```

6. Data Integrity:

- The system must ensure data integrity, with mechanisms to prevent data corruption or loss (e.g., regular backups and transaction logs).

Using both regular and redis database to ensure there is less lose on database or prevent database crashing causing data loss.

Design Patterns Implementation in Code

1. Singleton Pattern:

In the first screenshot, **the UserSingleton** class is implementing the Singleton Design Pattern.

-**Purpose:** This pattern ensures that a class has only one instance and provides a global point of access to that instance.

Implementation:

- The `_instance` variable is used to store the unique instance of the class.
- The `get_instance` method is marked as a `@classmethod` and checks whether the instance is `None`. If it is, it creates the instance by fetching the user from the database. If an instance already exists, it returns that existing instance.
- The pattern ensures that only one instance of the `UserSingleton` is created and accessed throughout the application.

```python
# singleton.py
from .models import User

class UserSingleton:
    _instance = None

    @classmethod
    def get_instance(cls, user_id=None):
        if cls._instance is None and user_id:
            cls._instance = User.objects.get(id=user_id)
        return cls._instance
```

2. Strategy Pattern:

In the second screenshot, the Strategy Design Pattern is implemented to manage the upload strategies for different content types.

Purpose: The Strategy pattern defines a family of algorithms (upload strategies) and allows them to be interchangeable.

Implementation:

- The `UploadStrategy` is an abstract class that defines the method `get_upload_to`, which is used to determine the file upload behavior.
- The classes `SlideUploadStrategy`, `VideoUploadStrategy`, and `NoteUploadStrategy` are concrete implementations of the `UploadStrategy` class, each specifying a different upload path based on the content type.
- The `get_upload_to` method uses the `os.path.join()` method to dynamically create the appropriate file path based on the instance's properties.

- The `RegistrationStrategy` is an abstract class with a method `register`, which is implemented by the concrete strategies `RegularUserRegistration` and `AdminUserRegistration`.
- The `RegularUserRegistration` class implements the `register` method to handle the logic of registering a regular user.
- The `AdminUserRegistration` class implements the `register` method for admin user registration. It also sets an `is_admin` flag to `True` before saving the user.
- The pattern abstracts the creation process of different types of users, allowing for flexible changes to user registration logic in the future.

```python
import os
from abc import ABC, abstractmethod

class UploadStrategy(ABC):
    """
    Abstract Strategy class that defines the upload behavior for file fields.
    """
    @abstractmethod
    def get_upload_to(self, instance, filename):
        pass


# Strategy for Slide model (Base)
class SlideUploadStrategy(UploadStrategy):
    def get_upload_to(self, instance, filename):
        return os.path.join(
            'contents',
            instance.faculty.course.department.name,
            instance.faculty.course.course_name,
            instance.faculty.name,
            'slides',  # Folder for regular slides
            filename
        )

# Strategy for Video model (Base)
class VideoUploadStrategy(UploadStrategy):
    def get_upload_to(self, instance, filename):
        return os.path.join(
            'contents',
            instance.faculty.course.department.name,
            instance.faculty.course.course_name,
            instance.faculty.name,
            'videos',  # Folder for regular videos
            filename
        )

# Strategy for Note model (Base)
class NoteUploadStrategy(UploadStrategy):
    def get_upload_to(self, instance, filename):
        return os.path.join(
            'contents',
            instance.faculty.course.department.name,
            instance.faculty.course.course_name,
            instance.faculty.name,
            'notes',  # Folder for regular notes
            filename
        )
```

```python
class TempSlideUploadStrategy(UploadStrategy):
    def get_upload_to(self, instance, filename):
        if hasattr(instance, 'real') and instance.real:
            try:
                # Construct the file upload path using the 'real' instance
                return os.path.join(
                    'contents',
                    instance.real.faculty.course.department.name,
                    instance.real.faculty.course.course_name,
                    instance.real.faculty.name,
                    'temp_slides',  # Temporary folder for slides
                    filename
                )
            except AttributeError:
                raise ValueError("Real instance is not linked properly for temp_Slide.")
        else:
            # If 'real' is not linked, use a default upload path
            return os.path.join(
                'contents',
                'temp',
                'slides',  # Default folder for slides
                filename
            )


# Strategy for Temporary Video model
class TempVideoUploadStrategy(UploadStrategy):
    def get_upload_to(self, instance, filename):
        if hasattr(instance, 'real') and instance.real:
            return os.path.join(
                'contents',
                instance.real.faculty.course.department.name,
                instance.real.faculty.course.course_name,
                instance.real.faculty.name,
                'temp_videos',  # Temporary folder for videos
                filename
            )
        else:
            # Default path when 'real' is not linked
            return os.path.join(
                'contents',
                'temp',
                'videos',  # Default folder for videos
                filename
            )
```

```python
# Strategy for Temporary Note model
class TempNoteUploadStrategy(UploadStrategy):
    def get_upload_to(self, instance, filename):
        if hasattr(instance, 'real') and instance.real:
            return os.path.join(
                'contents',
                instance.real.faculty.course.department.name,
                instance.real.faculty.course.course_name,
                instance.real.faculty.name,
                'temp_notes',  # Temporary folder for notes
                filename
            )
        else:
            # Default path when 'real' is not linked
            return os.path.join(
                'contents',
                'temp',
                'notes',  # Default folder for notes
                filename
            )
```

```python
from django.contrib.auth.hashers import make_password
from .models import User

def create_user(email, password, first_name, last_name,username):
    user = User(email=email, first_name=first_name, last_name=last_name,username=username)
    user.set_password(password)
    user.save()
    return user

class RegistrationStrategy:
    def register(self, data):
        raise NotImplementedError

class RegularUserRegistration(RegistrationStrategy):
    def register(self, data):
        # Regular user registration logic
        return create_user(email=data['email'], password=data['passw'], first_name=data['fname'], last_name=data['lname'],username=data[

class AdminUserRegistration(RegistrationStrategy):
    def register(self, data):
        # Admin registration logic
        user = create_user(email=data['email'], password=data['passw'], first_name=data['fname'], last_name=data['lname'],username=data[
        user.is_admin = True
        user.save()
        return user
```

3. Observer Pattern:

In the third screenshot, the Observer Design Pattern is used to notify observers about a change in the state.

Purpose: The Observer pattern is used to allow a subject (the object being observed) to notify its observers about any state changes without knowing who or what those observers are.

Implementation:

- The `Observer` class defines an `update` method, which is implemented by concrete observers like `EmailObserver`, `SMSObserver`, and `MessageObserver`.
- The `Subject` class maintains a list of observers and notifies them whenever there's a change in its state (such as sending a message).
- The `notify` method in the `Subject` class calls the `update` method of each observer to send notifications like emails, SMS, or messages.

```python
from django.contrib import messages

class Observer:
    def update(self, request, message):
        pass


class EmailObserver(Observer):
    def update(self, request, message):
        # Here, you'd add your email notification logic.
        print(f"Email sent with message: {message}")


class SMSObserver(Observer):
    def update(self, request, message):
        # Here, you'd add your SMS notification logic.
        print(f"SMS sent with message: {message}")


class MessageObserver(Observer):
    def update(self, request, message):
        messages.success(request,message)
        print(f"Message: {message}")


class Subject:
    def __init__(self):
        # Initialize the observers list and attach the default MessageObserver
        self._observers = []
        self.attach(MessageObserver())  # Attach MessageObserver by default

    def attach(self, observer: Observer):
        """Attach an observer to the subject."""
        self._observers.append(observer)

    def detach(self, observer: Observer):
        """Remove an observer from the subject."""
        self._observers.remove(observer)

    def notify(self, request, message):
        """Notify all observers about the change."""
        for observer in self._observers:
            observer.update(request, message)
```

4. Factory Pattern:

In the last screenshot, the Factory Design Pattern is used to create different types of content instances.

Purpose: The Factory pattern is used to create objects without specifying the exact class of object that will be created.

Implementation:

- The `SlideFactory` and `VideoFactory` classes are responsible for creating instances of the `Slide` and `Video` models, respectively.
- These factory classes handle the creation logic and use the `SlideUploadStrategy` and `VideoUploadStrategy` to manage file upload behavior for each content type.
- The `create_slide` and `create_video` methods are static methods that encapsulate the object creation process, ensuring that all necessary steps (like setting upload strategies) are handled before the object is returned.

Product Interface (SessionDisplay)->Defines the contract (get_display_data()) that all session displays must implement.
Concrete Products (DetailedSessionDisplay)->Implements specific display formats (e.g., shows device, OS, IP).
Factory Class (SessionDisplayFactory)->Acts as a centralized "assembly line" to create the right display object based on input (type parameter).

```python
class SlideFactory:

    @staticmethod
    def create_slide(faculty, name, content_file):

        # Create the Slide model instance
        slide_instance = Slide(
            name=name,
            faculty=faculty,
            content=content_file,
        )

        # Set the upload strategy
        upload_strategy = SlideUploadStrategy()
        slide_instance.content.upload_to = upload_strategy.get_upload_to

        # Save the Slide instance
        slide_instance.save()
        return slide_instance


class VideoFactory:

    @staticmethod
    def create_video(faculty, name, content_file):

        # Create the Video model instance
        video_instance = Video(
            name=name,
            faculty=faculty,
            content=content_file,
        )

        # Set the upload strategy
        upload_strategy = VideoUploadStrategy()
        video_instance.content.upload_to = upload_strategy.get_upload_to

        # Save the Video instance
        video_instance.save()
        return video_instance
```

```python
class NoteFactory:

    @staticmethod
    def create_note(faculty, name, content_file):

        # Create the Note model instance
        note_instance = Note(
            name=name,
            faculty=faculty,
            content=content_file,
        )

        # Set the upload strategy
        upload_strategy = NoteUploadStrategy()
        note_instance.content.upload_to = upload_strategy.get_upload_to

        # Save the Note instance
        note_instance.save()
        return note_instance


class TemporarySlideFactory:

    @staticmethod
    def create_temp_slide(real_instance, faculty, name, content_file):

        # Create the temp_Slide model instance
        temp_slide_instance = temp_Slide(
            name=name,
            real=real_instance,  # Link to the real Slide instance
            faculty=faculty,
            content=content_file,
        )

        # Set the upload strategy
        upload_strategy = TempSlideUploadStrategy()
        temp_slide_instance.content.upload_to = upload_strategy.get_upload_to

        # Save the temp_Slide instance
        temp_slide_instance.save()
        return temp_slide_instance
```

```python
class TemporaryVideoFactory:

    @staticmethod
    def create_temp_video(real_instance, faculty, name, content_file):

        # Create the temp_Video model instance
        temp_video_instance = temp_Video(
            name=name,
            real=real_instance,  # Link to the real Video instance
            faculty=faculty,
            content=content_file,
        )

        # Set the upload strategy
        upload_strategy = TempVideoUploadStrategy()
        temp_video_instance.content.upload_to = upload_strategy.get_upload_to

        # Save the temp_Video instance
        temp_video_instance.save()
        return temp_video_instance


class TemporaryNoteFactory:


    @staticmethod
    def create_temp_note(real_instance, faculty, name, content_file):

        # Create the temp_Note model instance
        temp_note_instance = temp_Note(
            name=name,
            real=real_instance,  # Link to the real Note instance
            faculty=faculty,
            content=content_file,
        )

        # Set the upload strategy
        upload_strategy = TempNoteUploadStrategy()
        temp_note_instance.content.upload_to = upload_strategy.get_upload_to

        # Save the temp_Note instance
        temp_note_instance.save()
        return temp_note_instance
```

```python
from django.utils import timezone
import user_agents
class SessionDisplay:
    def __init__(self, session, request):
        self.session = session
        self.request = request

    def get_display_data(self):
        raise NotImplementedError

class DetailedSessionDisplay(SessionDisplay):
    def get_display_data(self):
        ua_string = self.request.META.get('HTTP_USER_AGENT', '')
        user_agent = user_agents.parse(ua_string)
        return {
            'id': self.session.session_key,
            'device': f"{user_agent.get_device()} {user_agent.get_os()}",
            'location': self._get_ip_location(),
            'last_activity': self.session.expire_date - timezone.timedelta(seconds=3600),
            'current': (self.session.session_key == self.request.session.session_key)
        }

    def _get_ip_location(self):
        ip = self.request.META.get('HTTP_X_FORWARDED_FOR') or self.request.META.get('REMOTE_ADDR')
        return 'Localhost' if ip == '127.0.0.1' else f"IP: {ip.split(',')[0]}"

class SessionDisplayFactory:
    @staticmethod
    def create_display(type, session, request):
        if type == 'detailed':
            return DetailedSessionDisplay(session, request)
        # Add other display types as needed
        raise ValueError(f"Unknown display type: {type}")
```

5.Proxy Design Pattern:

In our code, the **Cache Proxy** pattern is implemented through the QueryCacheProxy class. Here's how the pattern fits:

 Purpose: The **QueryCacheProxy** class serves as a proxy that caches the results of database queries. When you request data (e.g., departments, courses, faculty), it first checks if the data is available in the cache. If the data is cached, it returns the cached data, avoiding the need to fetch the data again from the database. If the data is not cached, it queries the database, stores the result in the cache, and then returns it.

- **Usage**: This caching mechanism significantly reduces the number of database calls, improving the performance of your application. Instead of querying the database every time a request is made, the system only performs the query when necessary (i.e., when the data is not found in the cache).

Implementation:

 Cache Check: Before making any expensive database query, the proxy checks if the requested data (departments, courses, etc.) is already in the cache.

 Data Fetching: If the data is not found in the cache, the proxy fetches it from the database.

 Cache Storage: Once the data is fetched from the database, it is stored in the cache for future use. The cache time (timeout) is set to 15 minutes, ensuring the data is refreshed periodically.

```python
class QueryCacheProxy:
    def __init__(self, user):
        self.user = user

    @login_required
    def get_departments(self):

        departments_cache_key = 'all_departments'
        departments = cache.get(departments_cache_key)
        print(departments)
        if not departments:
            print("not cached yet")

            departments = Department.objects.all().order_by('name')
            cache.set(departments_cache_key, departments, timeout=15)  # Cache for 15 minutes

        return departments
    @login_required
    def get_deptCourses(self,dept_id):
        department=Department.objects.get(id=dept_id)

        coursess_cache_key = f'department?{department.id}'
        courses = cache.get(coursess_cache_key)
        print(courses)
        if not courses:
            print("not cached yet")
            # If departments are not cached, fetch from DB and cache them
            courses = Course.objects.filter(department=department).order_by('course_name')
            cache.set(coursess_cache_key, courses, timeout=15)  # Cache for 15 minutes

        return courses,department
    @login_required
    def get_courseFacs(self,dept_id,course_id):
        department=Department.objects.get(id=dept_id)
        course=Course.objects.get(id=course_id)

        faculties_cache_key = f'department?{department.id}/course?{course.id}'
        faculties = cache.get(faculties_cache_key)
        print(faculties)
        if not faculties:
            print("not cached yet")

            faculties = Faculty.objects.filter(course=course).order_by('name')
            cache.set(faculties_cache_key, faculties, timeout=15)  # Cache for 15 minutes

        return faculties,department,course
```

```python
class QueryCacheProxy:
    @login_required
    def get_LecSlides(self,dept_id,course_id,fac_id):
        department=Department.objects.get(id=dept_id)
        course=Course.objects.get(id=course_id)
        faculty=Faculty.objects.get(id=fac_id)
        Slides_cache_key = f'department?{department.id}/course?{course.id}/faculty?{faculty.id}/Lectures/Slides'
        slides = cache.get(Slides_cache_key)
        print(slides)
        if not slides:
            print("not cached yet")
            # If departments are not cached, fetch from DB and cache them
            slides = Slide.objects.filter(faculty=faculty).order_by('-id')
            cache.set(Slides_cache_key, slides, timeout=10)  # Cache for 15 minutes
        return slides,department,course,faculty

    @login_required
    def get_LecVideos(self,dept_id,course_id,fac_id):
        department=Department.objects.get(id=dept_id)
        course=Course.objects.get(id=course_id)
        faculty=Faculty.objects.get(id=fac_id)
        Videos_cache_key = f'department?{department.id}/course?{course.id}/faculty?{faculty.id}/Lectures/Videos'
        videos = cache.get(Videos_cache_key)
        print(videos)
        if not videos:
            print("not cached yet")

            videos = Video.objects.filter(faculty=faculty).order_by('-id')
            cache.set(Videos_cache_key, videos, timeout=10)  # Cache for 15 minutes

        return videos,department,course,faculty


    @login_required
    def get_LecNotes(self,dept_id,course_id,fac_id):
        department=Department.objects.get(id=dept_id)
        course=Course.objects.get(id=course_id)
        faculty=Faculty.objects.get(id=fac_id)
        Notes_cache_key = f'department?{department.id}/course?{course.id}/faculty?{faculty.id}/Lectures/Notes'
        notes = cache.get(Notes_cache_key)
        print(notes)
        if not notes:
            print("not cached yet")

            notes = Note.objects.filter(faculty=faculty).order_by('-id')
            cache.set(Notes_cache_key, notes, timeout=10)

        return notes,department,course,faculty
```

6. Decorator Design Pattern:

Purpose:
The **Decorator Design Pattern** is a structural pattern that allows behavior to be added to an object dynamically, without altering the object's structure. It involves wrapping an object (or function) with another function that enhances or modifies its behavior.

Usage:
Decorators are commonly used in Python to modify or enhance the behavior of functions or methods. The decorator pattern allows you to add new functionality to an object or function at runtime, which is especially useful for logging, authentication, caching, or access control without modifying the core logic.

Implementation:

In your code, the log_activity decorator is implemented to log user activity whenever a function is called. Here's how the **Decorator Pattern** fits:

- **Purpose**: The log_activity decorator logs the activity of the user, specifically tracking what action they are performing (e.g., calling a view or executing a method). It adds this behavior to the function it's decorating without modifying the core function itself.
- **Usage**: This decorator can be applied to any function or method to log user activities. For example, if we have different views handling various requests, you can decorate each view function with @log_activity to track which user is performing what action.

```python
# decorators.py
def log_activity(func):
    def wrapper(request, *args, **kwargs):
        print(f"User activity logged for {request.user.username} performing {func.__name__}")
        return func(request, *args, **kwargs)
    return wrapper
```

7. Adapter Design Pattern

Implementation:

We implemented the **Adapter Design Pattern** as an intermediary between the **Factory** and the **Strategy** patterns during the content upload process (specifically for Slides, Videos, and Notes). The Adapter plays a crucial role in connecting the content creation logic (from the Factory) with the appropriate content upload behavior (defined in the Strategy pattern).

Why Adapter Was Needed:

Each content type (Slide, Video, Note) and its status (temporary or permanent) required different uploading strategies. Rather than handling these decisions directly in the Factory — which would violate the Single Responsibility Principle — I introduced an Adapter that **maps the content type and status to the correct strategy**.

How the Adapter Works:

1. **Factories** like SlideFactory, VideoFactory, and their temporary counterparts call the **ContentUploadAdapter**, passing in:
    - The content type (e.g., "slide", "video", "note"),
    - Whether the content is temporary (is_temp=True) or permanent.
2. Inside the ContentUploadAdapter, based on these parameters, the correct **Strategy** is selected
3. The selected **upload strategy** then defines the get_upload_to() method — essentially telling Django where to store the uploaded file (permanent folder or temporary).
4. The Factory receives the strategy from the adapter and sets the file path.

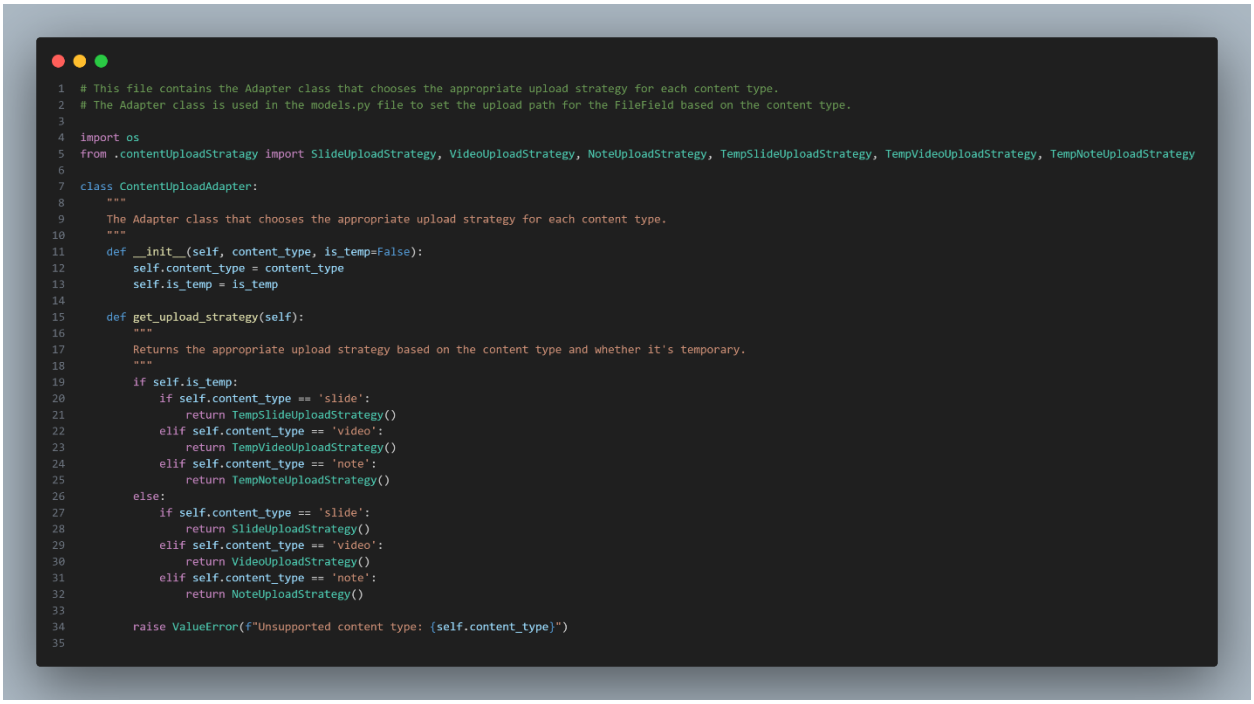
Key Benefits of This Adapter Design:

- The Factory class **doesn't need to know about upload logic**.
- The Adapter acts as a **translator**, determining the appropriate strategy dynamically.
- The Strategy pattern ensures **upload logic is encapsulated** and easily extendable.
- This design supports **clean separation of concerns**, making the system modular, scalable, and easy to maintain.

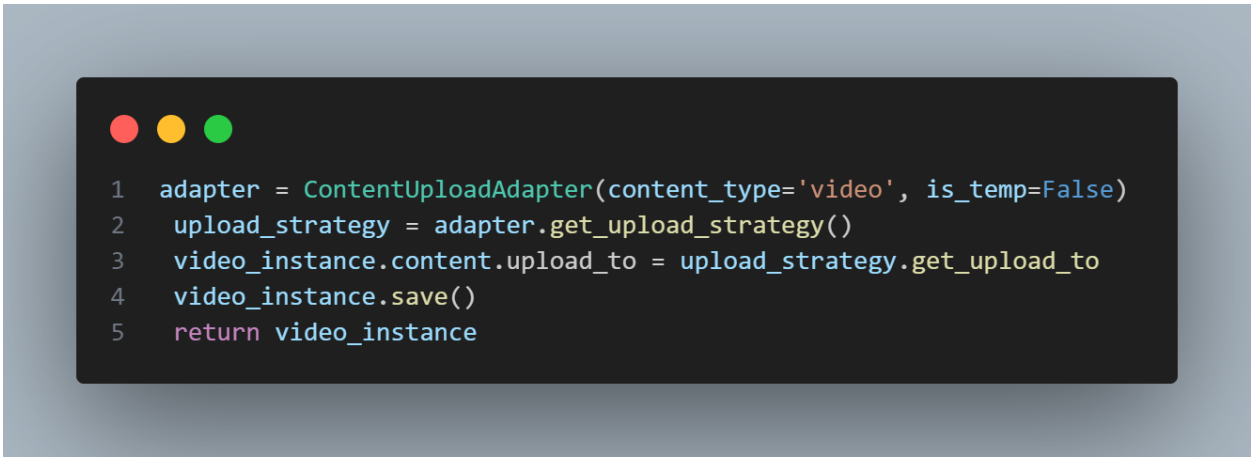
Flow Summary:

Factory → Adapter → Strategy
Content object is created → Adapter selects upload strategy → Strategy defines upload path

Implementation Screenshots:



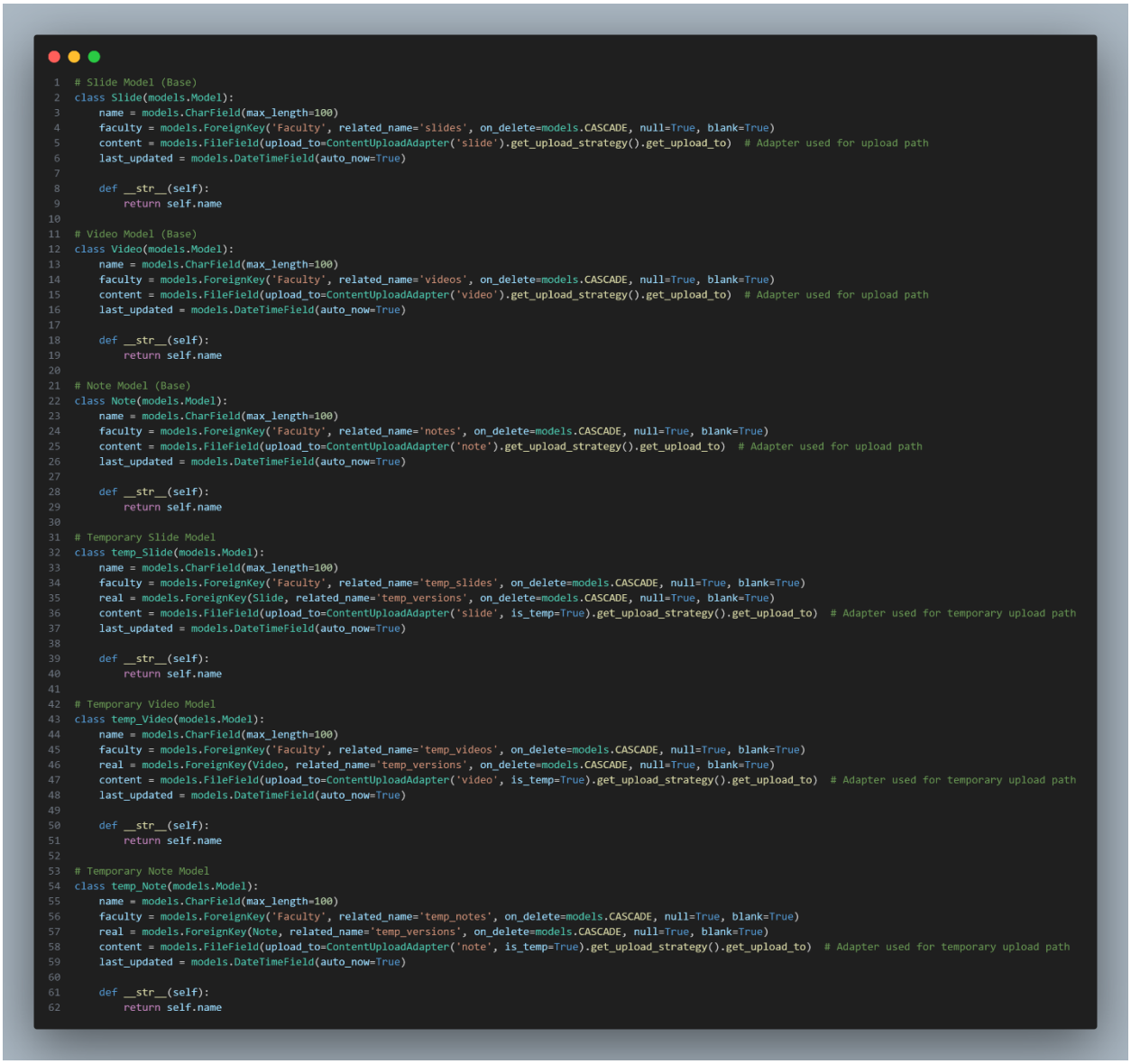
```python
# This file contains the Adapter class that chooses the appropriate upload strategy for each content type.
# The Adapter class is used in the models.py file to set the upload path for the FileField based on the content type.

import os
from .contentUploadStratagy import SlideUploadStrategy, VideoUploadStrategy, NoteUploadStrategy, TempSlideUploadStrategy, TempVideoUploadStrategy, TempNoteUploadStrategy

class ContentUploadAdapter:
    """
    The Adapter class that chooses the appropriate upload strategy for each content type.
    """
    def __init__(self, content_type, is_temp=False):
        self.content_type = content_type
        self.is_temp = is_temp

    def get_upload_strategy(self):
        """
        Returns the appropriate upload strategy based on the content type and whether it's temporary.
        """
        if self.is_temp:
            if self.content_type == 'slide':
                return TempSlideUploadStrategy()
            elif self.content_type == 'video':
                return TempVideoUploadStrategy()
            elif self.content_type == 'note':
                return TempNoteUploadStrategy()
        else:
            if self.content_type == 'slide':
                return SlideUploadStrategy()
            elif self.content_type == 'video':
                return VideoUploadStrategy()
            elif self.content_type == 'note':
                return NoteUploadStrategy()

        raise ValueError(f"Unsupported content type: {self.content_type}")

```



```python
adapter = ContentUploadAdapter(content_type='video', is_temp=False)
 upload_strategy = adapter.get_upload_strategy()
 video_instance.content.upload_to = upload_strategy.get_upload_to
 video_instance.save()
 return video_instance
```

```
adapter = ContentUploadAdapter(content_type='slide', is_temp=False)
upload_strategy = adapter.get_upload_strategy()
slide_instance.content.upload_to = upload_strategy.get_upload_to
slide_instance.save()
return slide_instance
```

```
adapter = ContentUploadAdapter(content_type='video', is_temp=True)
upload_strategy = adapter.get_upload_strategy()
temp_video_instance.content.upload_to = upload_strategy.get_upload_to
temp_video_instance.save()
return temp_video_instance
```



```python
# Slide Model (Base)
class Slide(models.Model):
    name = models.CharField(max_length=100)
    faculty = models.ForeignKey('Faculty', related_name='slides', on_delete=models.CASCADE, null=True, blank=True)
    content = models.FileField(upload_to=ContentUploadAdapter('slide').get_upload_strategy().get_upload_to)  # Adapter used for upload path
    last_updated = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.name

# Video Model (Base)
class Video(models.Model):
    name = models.CharField(max_length=100)
    faculty = models.ForeignKey('Faculty', related_name='videos', on_delete=models.CASCADE, null=True, blank=True)
    content = models.FileField(upload_to=ContentUploadAdapter('video').get_upload_strategy().get_upload_to)  # Adapter used for upload path
    last_updated = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.name

# Note Model (Base)
class Note(models.Model):
    name = models.CharField(max_length=100)
    faculty = models.ForeignKey('Faculty', related_name='notes', on_delete=models.CASCADE, null=True, blank=True)
    content = models.FileField(upload_to=ContentUploadAdapter('note').get_upload_strategy().get_upload_to)  # Adapter used for upload path
    last_updated = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.name

# Temporary Slide Model
class temp_Slide(models.Model):
    name = models.CharField(max_length=100)
    faculty = models.ForeignKey('Faculty', related_name='temp_slides', on_delete=models.CASCADE, null=True, blank=True)
    real = models.ForeignKey(Slide, related_name='temp_versions', on_delete=models.CASCADE, null=True, blank=True)
    content = models.FileField(upload_to=ContentUploadAdapter('slide', is_temp=True).get_upload_strategy().get_upload_to)  # Adapter used for temporary upload path
    last_updated = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.name

# Temporary Video Model
class temp_Video(models.Model):
    name = models.CharField(max_length=100)
    faculty = models.ForeignKey('Faculty', related_name='temp_videos', on_delete=models.CASCADE, null=True, blank=True)
    real = models.ForeignKey(Video, related_name='temp_versions', on_delete=models.CASCADE, null=True, blank=True)
    content = models.FileField(upload_to=ContentUploadAdapter('video', is_temp=True).get_upload_strategy().get_upload_to)  # Adapter used for temporary upload path
    last_updated = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.name

# Temporary Note Model
class temp_Note(models.Model):
    name = models.CharField(max_length=100)
    faculty = models.ForeignKey('Faculty', related_name='temp_notes', on_delete=models.CASCADE, null=True, blank=True)
    real = models.ForeignKey(Note, related_name='temp_versions', on_delete=models.CASCADE, null=True, blank=True)
    content = models.FileField(upload_to=ContentUploadAdapter('note', is_temp=True).get_upload_strategy().get_upload_to)  # Adapter used for temporary upload path
    last_updated = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.name
```

8. Facade Design Pattern

Implementation:

To manage the **complex registration process** in the system, I implemented the **Facade Design Pattern** through a dedicated class called RegistrationFacade. This pattern was crucial because the application supports **multiple user types** — including **Regular Users, Moderators, Admins, and the Master** — each with potentially distinct registration logic.

Why Facade Was Needed:

The registration process involves several steps such as:

- Validating form data,
- Selecting the appropriate user type,
- Applying the correct registration logic (Strategy),
- Saving the user instance,
- Sending confirmation messages or triggers.

Rather than exposing all of this complexity inside the signup() view (or spreading logic across multiple parts of the system), the **Facade centralizes** this entire process into a **single point of access**. This ensures the view remains **clean, simple, and readable**, while the **complex underlying logic is encapsulated** within the Facade class.

How the Facade Works:

1. In the signup() view, when a user submits a registration form, an instance of the **RegistrationFacade** is created.
2. The register(data) method is then called on the facade, which:
    - **Parses user-provided data**,
    - **Identifies the user type** (e.g., "regular", "admin"),
    - Internally selects the appropriate **Strategy class** (e.g., RegularUserRegistration, AdminUserRegistration) to handle the logic,
    - **Creates and saves** the new user,
    - Returns the final user object or registration status.
3. The calling view only sees a **simple method call**, while the Facade manages the delegation of responsibilities to the right classes.

Key Benefits of This Facade Implementation:

- Provides a **simple interface** (register()) for a **multi-step, complex process**.
- Maintains **separation of concerns** — view handles routing and response, while the facade handles logic.

- Easily extendable: new user types (e.g., Supervisor) can be added with minimal changes to the view.
- Reduces **duplicate logic** and simplifies debugging and maintenance.

User Hierarchy Managed via Facade:

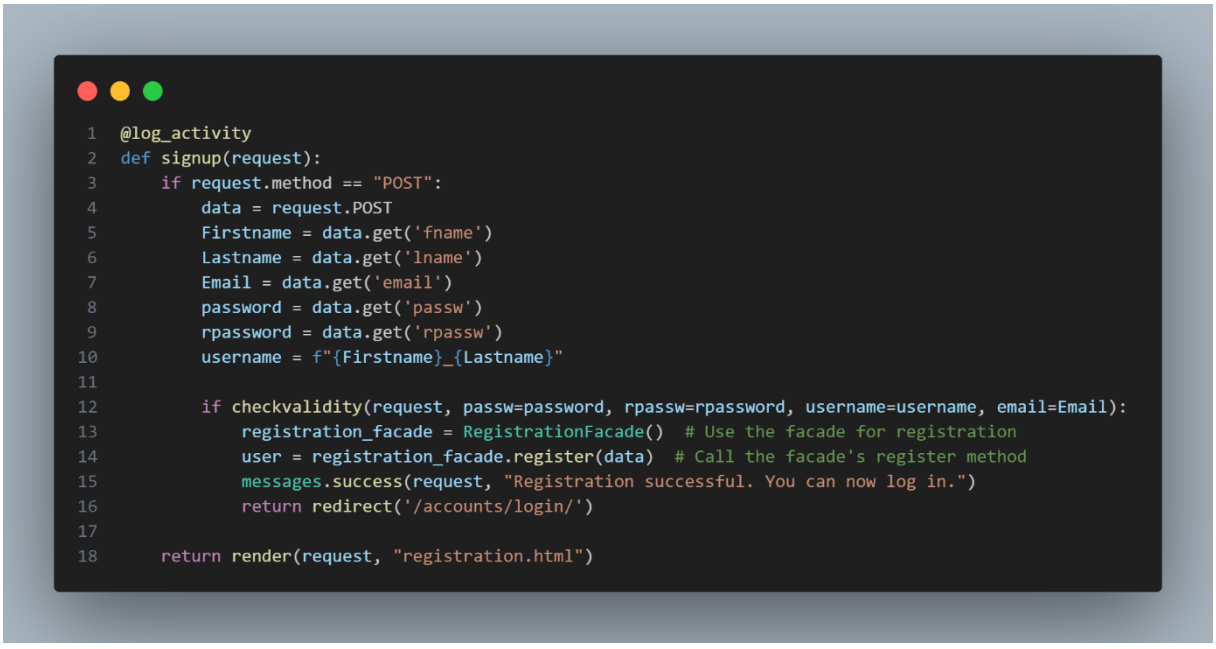
- **Master User**
- **Admin**
- **Moderator**
- **Regular User**

Each role may have a different registration strategy. The Facade **hides these role-specific details** and provides a consistent interface for the entire hierarchy.

Flow Summary:

Signup View → Facade → Registration Strategy → User Created

Implementation Screenshots:

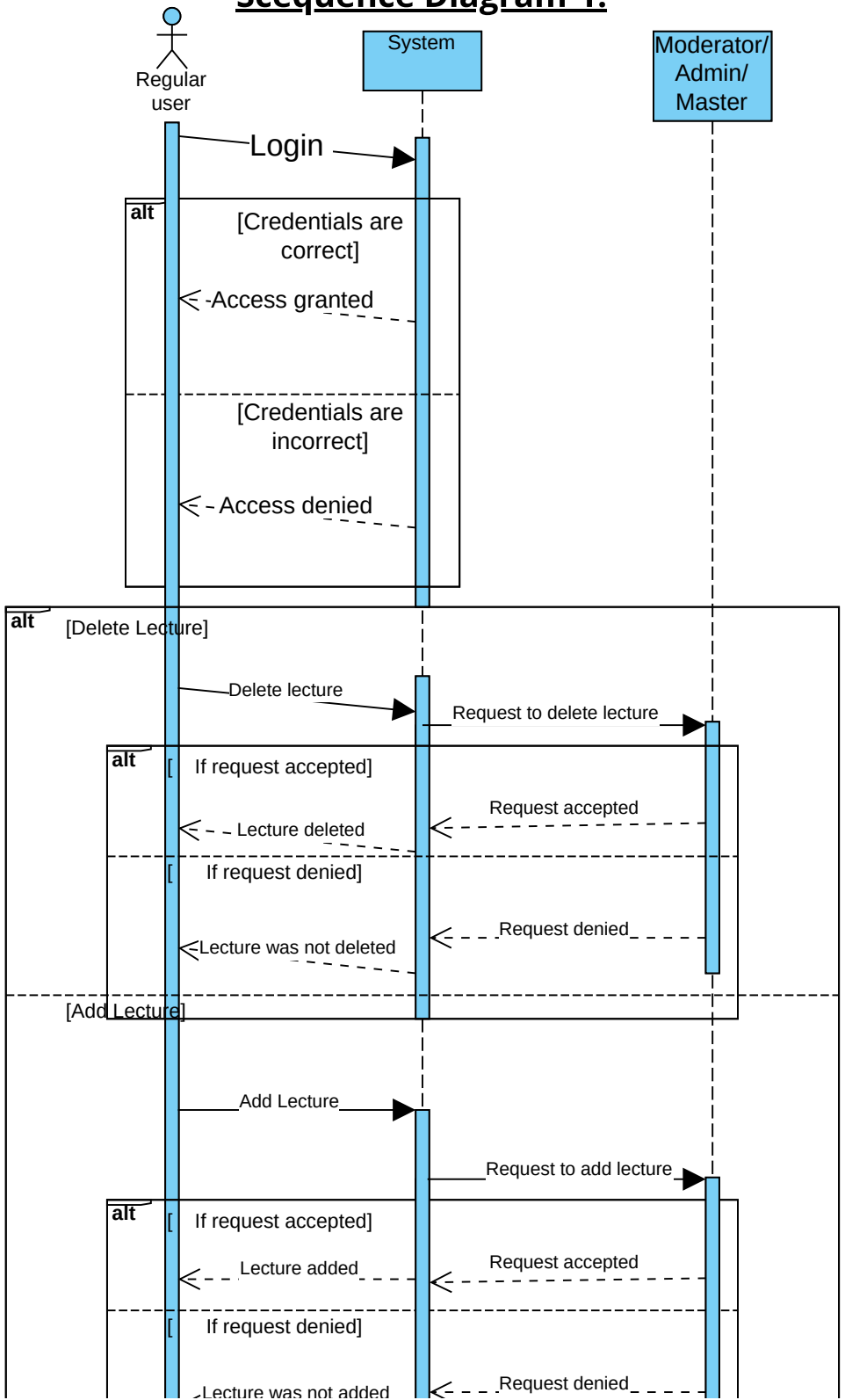


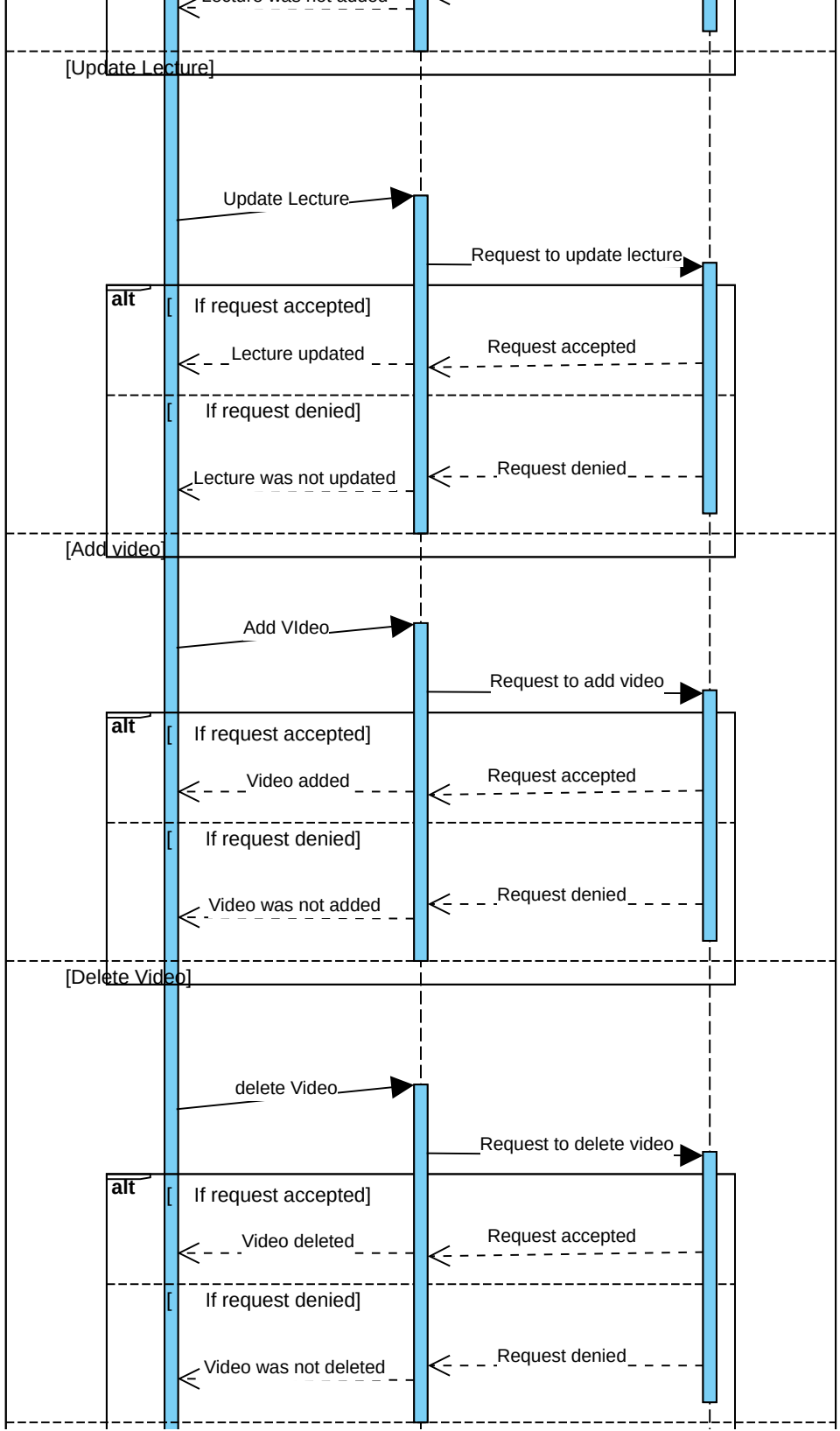
```python
@log_activity
def signup(request):
    if request.method == "POST":
        data = request.POST
        Firstname = data.get('fname')
        Lastname = data.get('lname')
        Email = data.get('email')
        password = data.get('passw')
        rpassword = data.get('rpassw')
        username = f"{Firstname}_{Lastname}"

        if checkvalidity(request, passw=password, rpassw=rpassword, username=username, email=Email):
            registration_facade = RegistrationFacade()  # Use the facade for registration
            user = registration_facade.register(data)  # Call the facade's register method
            messages.success(request, "Registration successful. You can now log in.")
            return redirect('/accounts/login/')

    return render(request, "registration.html")
```

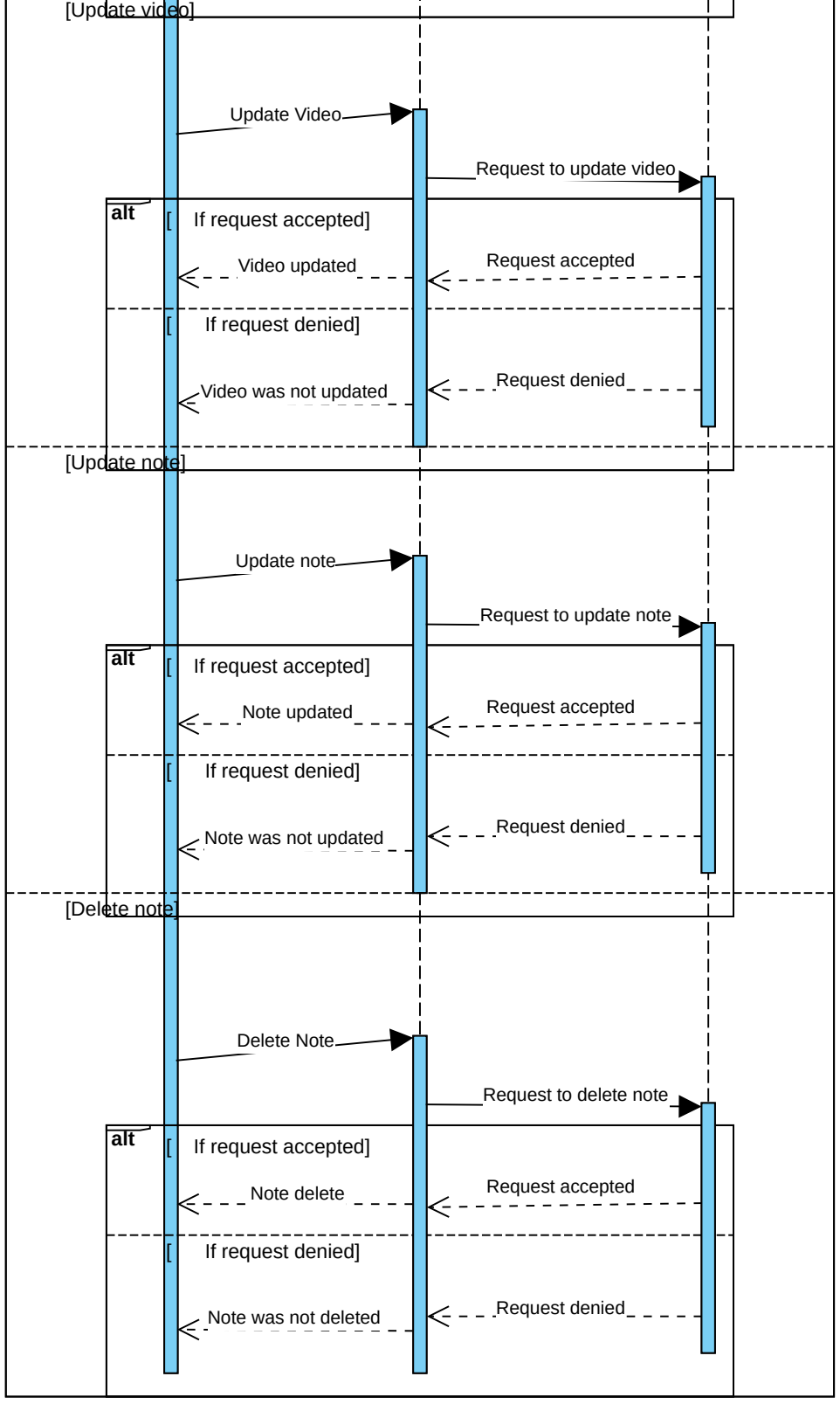
```python
# facade.py
from .strategies import RegularUserRegistration, AdminUserRegistration

class RegistrationFacade:
    def __init__(self):
        # Initialize strategies for regular users and admins
        self.regular_registration = RegularUserRegistration()
        self.admin_registration = AdminUserRegistration()

    def register(self, data, user_type="regular"):
        """
        A simplified interface for user registration.

        Parameters:
            data (dict): User data (email, passw, fname, lname)
            user_type (str): Type of user, can be "regular" or "admin"

        Returns:
            User object if registration is successful
        """
        if user_type == "admin":
            return self.admin_registration.register(data)
        return self.regular_registration.register(data)

```

```python
from django.contrib.auth.hashers import make_password
from .models import User

def create_user(email, password, first_name, last_name,username):
    user = User(email=email, first_name=first_name, last_name=last_name,username=username)
    user.set_password(password)
    user.save()
    return user

class RegistrationStrategy:
    def register(self, data):
        raise NotImplementedError

class RegularUserRegistration(RegistrationStrategy):
    def register(self, data):
        # Regular user registration logic
        return create_user(email=data['email'], password=data['passw'], first_name=data['fname'], last_name=data['lname'],username=data['email'])

class AdminUserRegistration(RegistrationStrategy):
    def register(self, data):
        # Admin registration logic
        user = create_user(email=data['email'], password=data['passw'], first_name=data['fname'], last_name=data['lname'],username=data['email'])
        user.is_admin = True
        user.save()
        return user
```

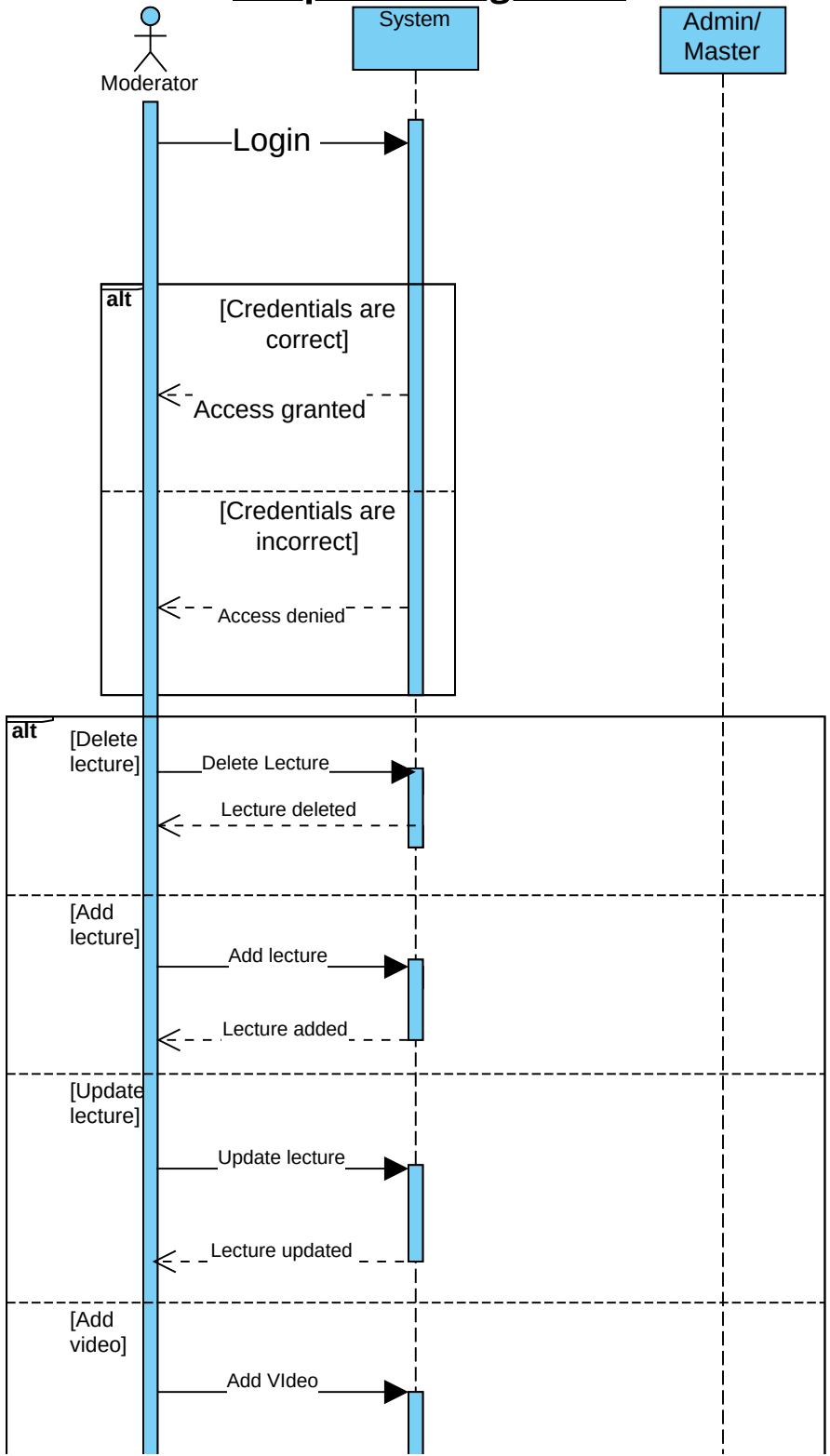
Scequence Diagram-1:

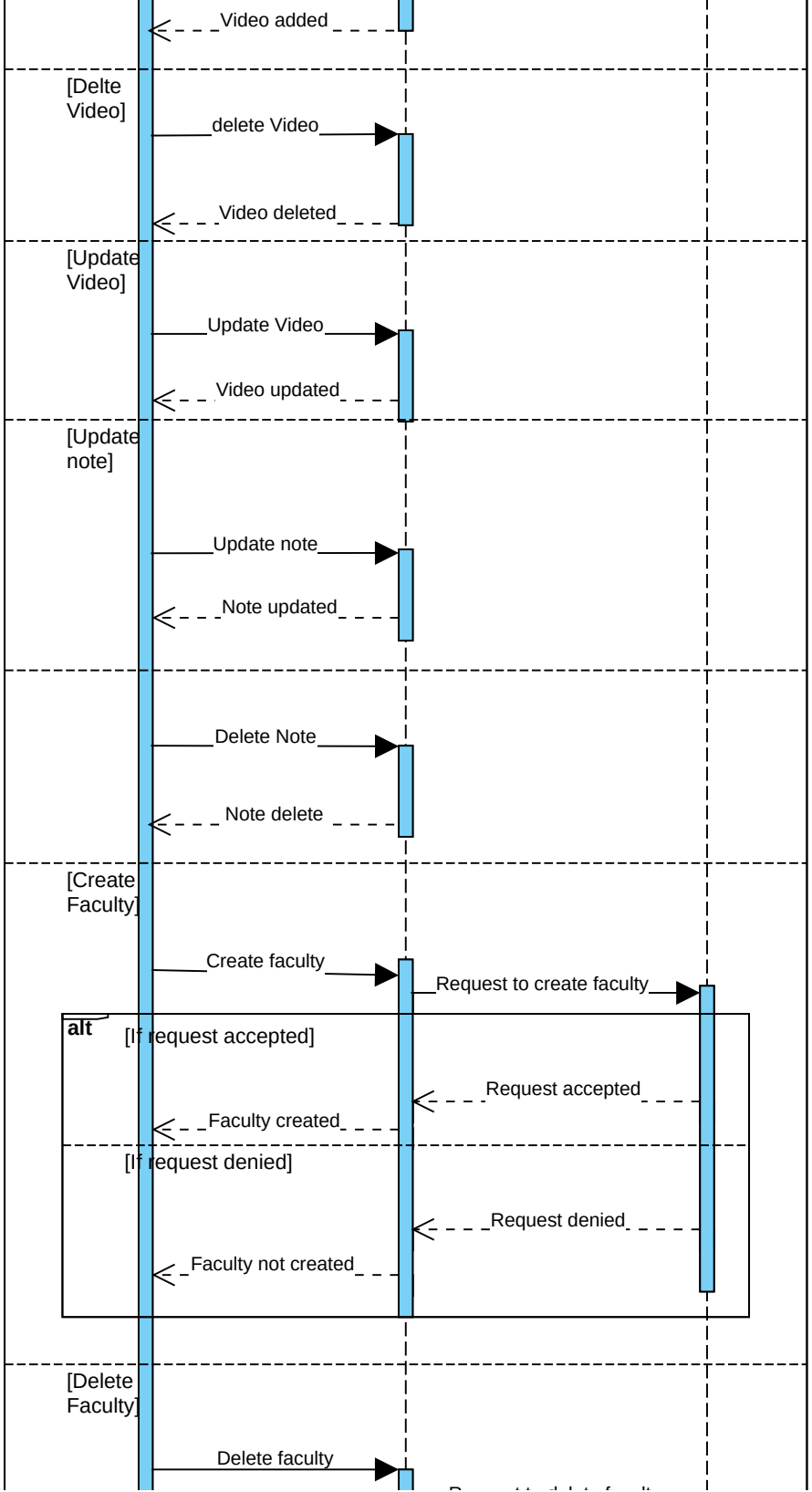


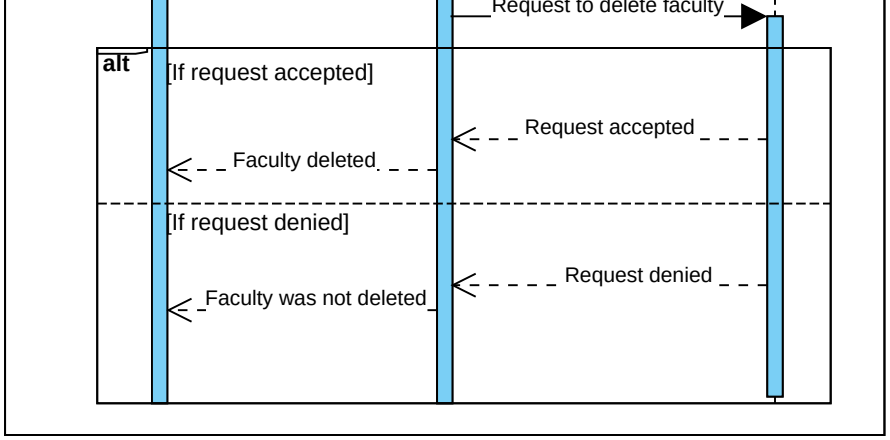




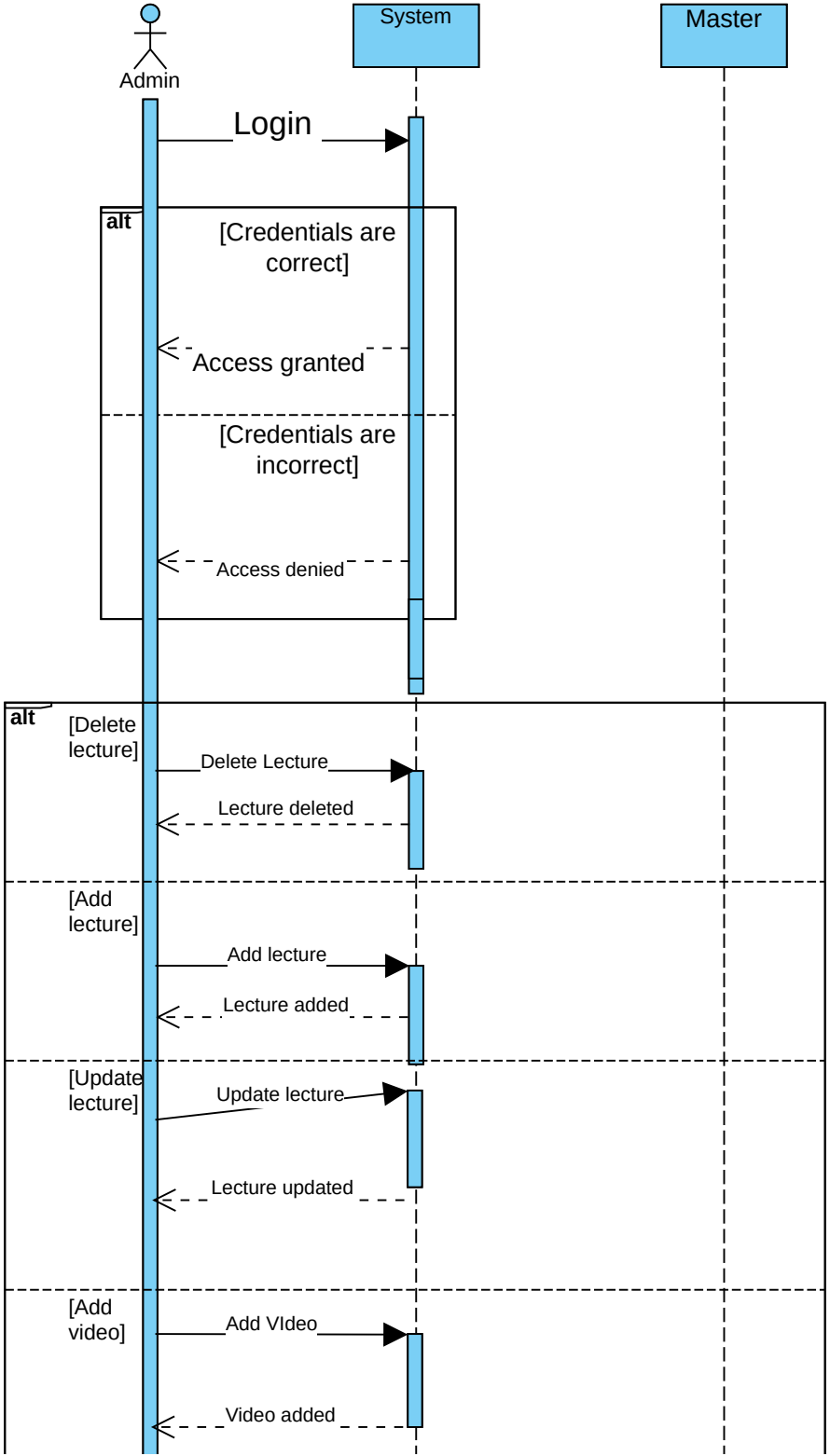
Scequence Diagram-2:

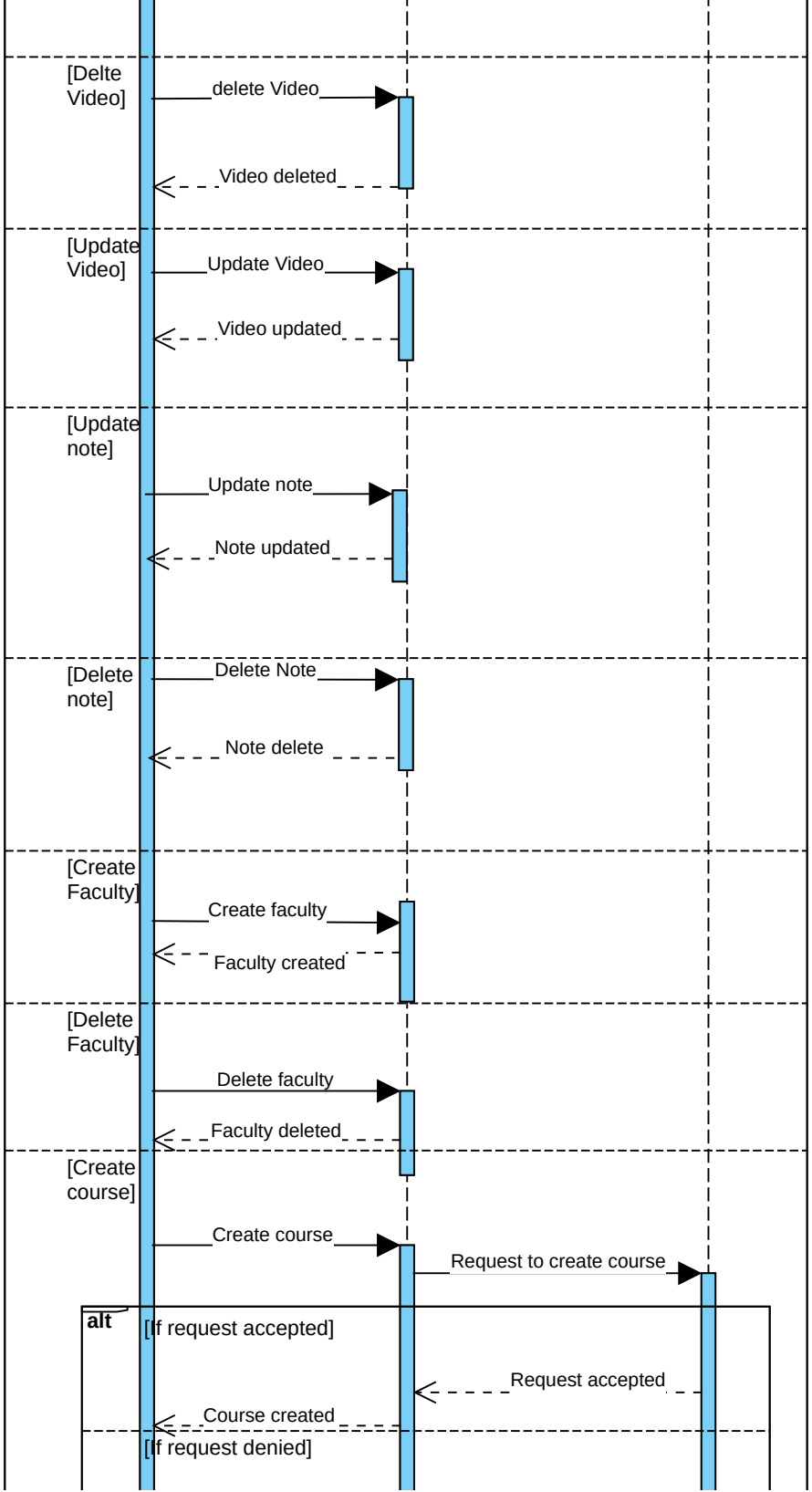


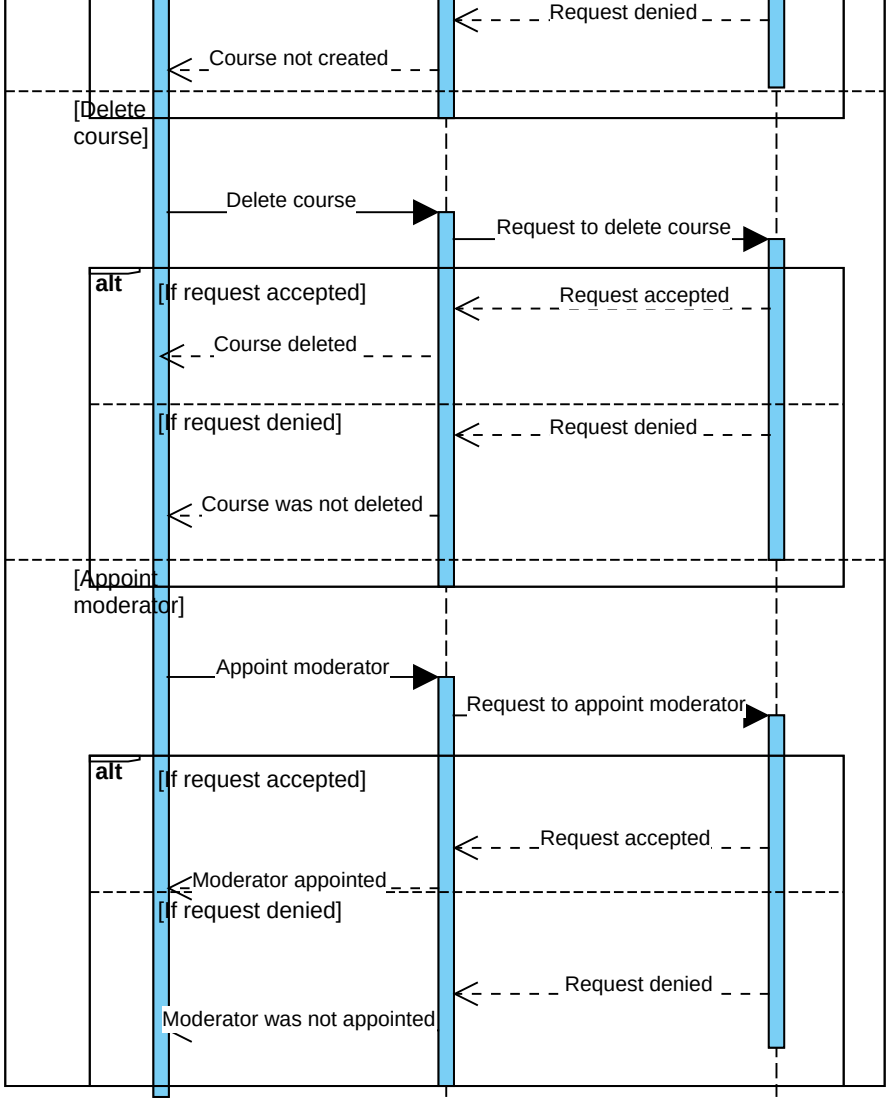




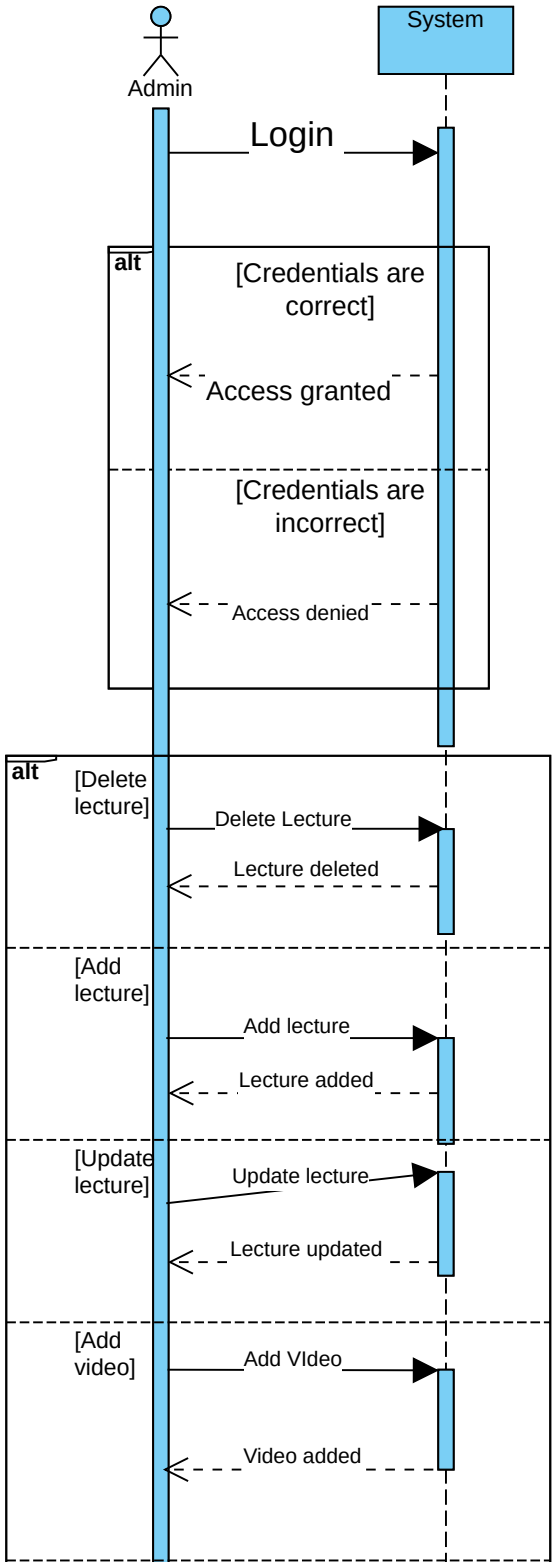
Scequence Diagram-3:

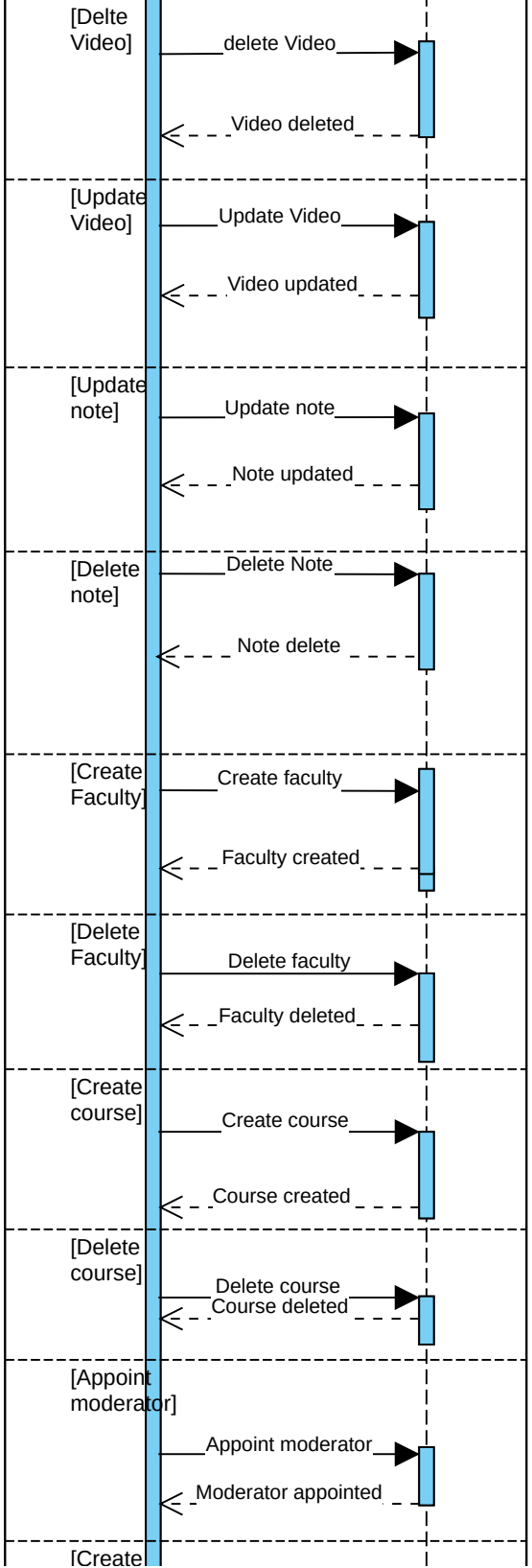


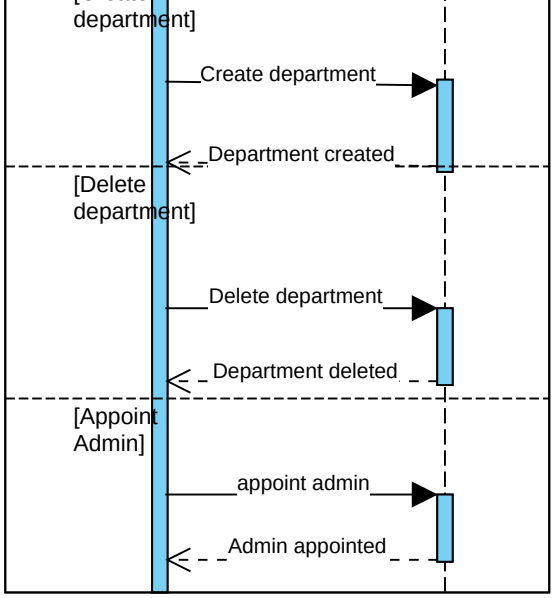




Scequence Diagram-4:







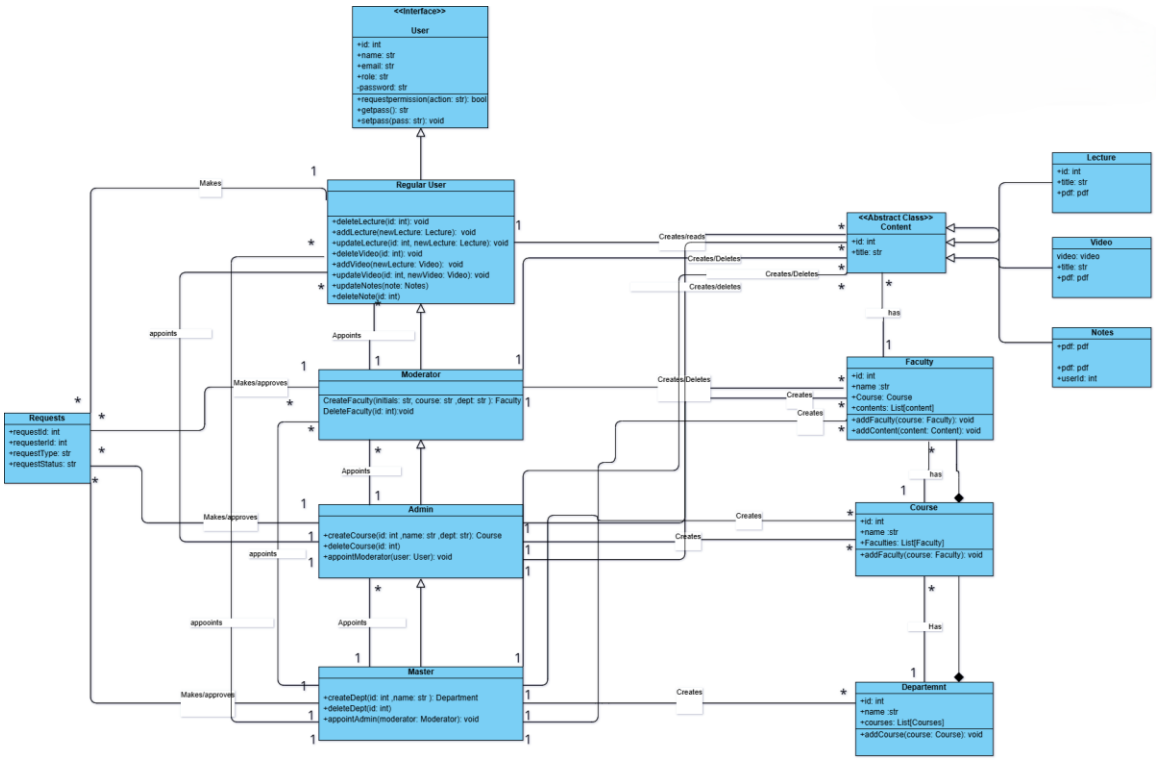
Class and Sequence Diagram Implementation Overview

Class and Sequence Diagram Overview

The **Class Diagram** and **Sequence Diagram** provide a visual representation of the system's structure and the flow of operations, respectively. Initially, the class diagram depicted separate classes for each user role: **Regular User**, **Moderator**, **Admin**, and **Master**. Each of these roles had its own set of attributes and methods tailored to their respective responsibilities:

- **Regular User**: Allowed users to manage lectures and notes, with functionality focused on content creation and updates.
- **Moderator**: Handled administrative tasks like managing faculty members and had additional permissions compared to Regular Users.
- **Admin**: Had broader responsibilities, including managing courses, faculties, and appointing or modifying user roles (Moderators and Admins).
- **Master**: The highest authority, responsible for creating or deleting departments, appointing Admins, and performing high-level administrative actions.

While this structure worked, it resulted in redundancy and increased complexity due to similar functionalities across roles. To simplify the system and enhance scalability, these four separate classes were replaced by a single **User** class. This class includes a **role** attribute, which dynamically determines the user's role and the permissions they can access.



Key Features of the Updated Class Structure:

- **Role-based Functionality**: Rather than managing four distinct classes, a single User class manages all user roles by checking the role attribute. This attribute determines whether the user is a **Regular User**, **Moderator**, **Admin**, or **Master**, and thus what actions they can perform in the system.
    - **Regular Users**: Can manage lectures and notes.
    - **Moderators**: Have permission to manage faculty.
    - **Admins**: Can create and manage courses and appoint other users.
    - **Masters**: Perform high-level tasks, such as managing departments and appointing Admins.
- **Simplification**: Consolidating the roles into one class reduces the need for separate classes, making the system easier to maintain. Instead of adding or modifying classes for each role, changes can be made by updating role-based logic in the User class.
- **Scalability**: The flexible, role-based approach enhances the scalability of the system. New roles can be added or modified easily by updating the role definitions and adjusting permissions.

Sequence Diagram Overview:

The **Sequence Diagram** outlines the dynamic interactions between the **Admin user** and the system. It captures the sequence of events triggered when the Admin logs in and performs various actions. Here's a breakdown of the key interactions:

1. **Login Process**:
    - **Admin attempts to log in** by submitting credentials.
    - If **credentials are correct**, access is granted, allowing the Admin to proceed with the operations.
    - If **credentials are incorrect**, access is denied, preventing further actions.
2. **Admin Operations**: Once logged in, the **Admin** can perform various tasks:
    - **Delete Lecture**: The Admin can delete an existing lecture.
    - **Add Lecture**: The Admin can add a new lecture to the system.
    - **Update Lecture**: The Admin can update lecture details.
3. **Video Management**: The Admin can manage videos associated with lectures:
    - **Add Video**: Admin can add videos to lectures.
    - **Delete Video**: Admin can remove videos from lectures.
    - **Update Video**: Admin can update the video content.
4. **Note Management**: Admin can also manage notes:
    - **Update Note**: Admin can update notes related to lectures.
    - **Delete Note**: Admin can delete notes from the system.
5. **Faculty Management**: Admin has control over faculty management:
    - **Create Faculty**: Admin can create new faculty members.
    - **Delete Faculty**: Admin can delete faculty members.

6. **Course Management**: Admin can manage courses:
    - **Create Course**: Admin can create new courses.
    - **Delete Course**: Admin can remove existing courses.
7. **Appointing Moderator**: Admin can appoint moderators to manage specific areas:
    - **Appoint Moderator**: Admin can assign moderators to various tasks or responsibilities.
8. **Department Management**: Admin has control over departments:
    - **Create Department**: Admin can create new academic departments.
    - **Delete Department**: Admin can delete departments from the system.
9. **Appointing Admin**: Admin can appoint other admins:
    - **Appoint Admin**: Admin has the authority to appoint other Admins to the system.

Implementations

Key changes in the class structure include:
User Class (Abstract Class): The 'User' class now contains common attributes like 'id', 'name', 'email', and 'role'. **Role-based Operations**: Instead of separate classes for each user type, a single 'User' class uses the 'role' attribute to determine the available functionality for that user (e.g., Regular User vs. Admin).

```python
class User(AbstractUser):
    contact = models.CharField(max_length=15, blank=True, null=True)
    profilepicture = models.ImageField(upload_to='images/profilepictures', blank=True, null=True,default='images/profilepictures/rafid1.jpg')
    role=models.TextField(default='user')
    course=models.IntegerField(default=-1)
    department=models.IntegerField(default=-1)
    groups = models.ManyToManyField(
        'auth.Group',
        related_name='custom_user_set',
        blank=True
    )
    user_permissions = models.ManyToManyField(
        'auth.Permission',
        related_name='custom_user_permissions_set',
        blank=True
    )

    def __str__(self):
        return self.username  # Adjusted to remove raw password display
```

5. Lectures, Videos, Notes, Faculty, Courses, and Departments Classes

5.1 Lecture Class

The **Lecture Class** represents academic lectures in the system. It contains the following attributes:
id: A unique identifier for the lecture.
title: The title of the lecture.
pdf: A link to the PDF file that contains the lecture's content.

```python
# Slide Model (Base)
class Slide(models.Model):
    name = models.CharField(max_length=100)
    faculty = models.ForeignKey('Faculty', related_name='slides', on_delete=models.CASCADE, null=True, blank=True)
    content = models.FileField(upload_to=ContentUploadAdapter('slide').get_upload_strategy().get_upload_to)  # Adapter used for upload path
    last_updated = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.name
```

5.2 Video Class

The **Video Class** represents videos attached to lectures. It contains the following attributes:

id: A unique identifier for the video.

title: The title of the video.

pdf: A link to the PDF file associated with the video.

```python
# Video Model (Base)
class Video(models.Model):
    name = models.CharField(max_length=100)
    faculty = models.ForeignKey('Faculty', related_name='videos', on_delete=models.CASCADE, null=True, blank=True)
    content = models.FileField(upload_to=ContentUploadAdapter('video').get_upload_strategy().get_upload_to)  # Adapter used for upload path
    last_updated = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.name
```

Like the **Lecture Class**, the **Video Class** provides multimedia content related to the lecture, allowing for richer learning experiences.

5.3 Notes Class

The **Notes Class** stores notes associated with each lecture. It contains the following attributes:

id: A unique identifier for the notes.

pdf: A link to the notes file (PDF format).

userId: Links the notes to the user who created or modified them.

```python
# Note Model (Base)
class Note(models.Model):
    name = models.CharField(max_length=100)
    faculty = models.ForeignKey('Faculty', related_name='notes', on_delete=models.CASCADE, null=True, blank=True)
    content = models.FileField(upload_to=ContentUploadAdapter('note').get_upload_strategy().get_upload_to)  # Adapter used for upload path
    last_updated = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.name
```

The **Notes Class** provides the opportunity to store important lecture notes, and it is tied to the user who manages the notes.

5.4 Faculty Class

The **Faculty Class** represents the faculty members teaching or associated with courses. It has the following attributes:

id: A unique identifier for the faculty member.

name: The name of the faculty member.

courses: A list of courses the faculty member is associated with.

The **Faculty Class** is linked to the **Course Class**, as faculty members teach courses and contribute content to lectures.

```
class Faculty(models.Model):
    name = models.CharField(max_length=100)
    course = models.ForeignKey('Course', related_name='Faculty', on_delete=models.CASCADE, null=True, blank=True)
    image = models.ImageField(upload_to='images/faculty/', null=True,default='images/faculty/default.jpg')
    position = models.CharField(max_length=100)

    # Add other relevant fields

    def __str__(self):
        return self.name
```

5.5 Course Class

The **Course Class** represents the courses in the system. It contains the following attributes:
id: A unique identifier for the course.
name: The name of the course.
faculty List: A list of faculty members teaching the course.

The **Course Class** manages courses and contains methods to add or remove faculty members from a course.

```
class Course(models.Model):
    id = models.AutoField(primary_key=True)  # Automatically generates a primary key
    course_code = models.CharField(max_length=6, unique=True)  # Unique course code
    course_name = models.CharField(max_length=50)  # Course name
    course_description = models.TextField(
        default="This course provides a comprehensive overview of essential topics, methodologies, and concepts. Further details will be provided."
    )  # Description of the course
    department = models.ForeignKey('Department', related_name='courses', on_delete=models.CASCADE)  # Foreign key to Department

    image = models.ImageField(upload_to='images/course/', null=True, default='images/course/default.jpg')  # Course image

    def __str__(self):
        return self.course_name
```

5.6 Department Class

The **Department Class** represents academic departments within the institution. Each department contains the following attributes:
id: A unique identifier for the department.
name: The name of the department.
courses: A list of courses managed by the department.

The **Department Class** is responsible for managing the courses under each department and linking them to the faculty and courses within the system.

```
class Department(models.Model):
    id = models.AutoField(primary_key=True)  # Automatically generates a primary key
    name = models.CharField(max_length=255, unique=True)  # Unique name field
    image = models.ImageField(upload_to='images/department/', null=True,default='images/department/default.jpg')
    description=models.TextField(max_length=255,default="The Department fosters innovative research and interdisciplinary collaboration, focusing on conceptual exploration and advan
    def __str__(self):
        return self.name
```

6. Relationships and Associations

6.1 User Interactions

All users (Regular, Moderator, Admin, Master) interact with core entities such as Lectures, **Videos, Notes, Courses, Faculty,** and **Departments**. Each user type has different permissions, dictated by the role attribute, which restricts or grants access to specific functionalities.

```python
@login_required
# Create your views here.
def departments(request):
    departments=Department.objects.all().order_by('name')
    showDeptModal=(request.user.role=='master')
    showUpdateDeptModal=(request.user.role=='master')
    showAddButton=(request.user.role=='master')
    context={'name':request.user.username,'departments':departments,
             'showDeptModal':showDeptModal,'showUpdateDeptModal':showUpdateDeptModal,'showAddButton':showAddButton}

    return render(request,"lms/departments.html",context)

#for Courses inside a Department
@login_required
def deptcourses(request,id):
    department=Department.objects.get(id=id)
    courses = Course.objects.filter(department=department).order_by('course_name')
    showAddButton=(request.user.role=='master')or(request.user.department==department.id)
    showUpdateCourseModal=(request.user.role=='master')or(request.user.department==department.id)
    showCourseModal=(request.user.role=='master')or(request.user.department==department.id)
    context={'name':request.user.username,'courses':courses , 'department': department,
             'showCourseModal':showCourseModal,'showUpdateCourseModal':showUpdateCourseModal,'showAddButton':showAddButton}
    return render(request,'lms/deptcourses.html',context)



#for faculties inside a Course
@login_required
def course_facs(request, dept_id, course_id):
    department = Department.objects.get(id=dept_id)
    course = Course.objects.get(id=course_id)
    faculties = Faculty.objects.filter(course=course)
    showFacultyModal=(request.user.role=='master')or(request.user.department==department.id)or(request.user.course==course.id)
    showUpdateFacultyModal=(request.user.role=='master')or(request.user.department==department.id)or(request.user.course==course.id)
    showAddButton=(request.user.role=='master')or(request.user.department==department.id)or(request.user.course==course.id)
    context = {'department': department, 'faculties': faculties,'course':course,
               'showFacultyModal':showFacultyModal,'showUpdateFacultyModal':showUpdateFacultyModal,'showAddButton':showAddButton}
    return render(request, 'lms/faculty.html', context)


#for slides inside a lecture
@login_required
def lec_slides(request,dept_id, course_id,fac_id):
    department = Department.objects.get(id=dept_id)
    course = Course.objects.get(id=course_id)
    faculty = Faculty.objects.get(id=fac_id)
    slides=Slide.objects.filter(faculty=faculty)
    context = {'department': department, 'faculty': faculty,'course':course,'slides':slides,'showSlideModal':True,'showUpdateSlideModal':True,'showAddButton':True}

    return render(request,"lms/slides.html",context)



# For videos inside a lecture
@login_required
def lec_videos(request, dept_id, course_id, fac_id):
    department = Department.objects.get(id=dept_id)
    course = Course.objects.get(id=course_id)
    faculty = Faculty.objects.get(id=fac_id)
    videos = Video.objects.filter(faculty=faculty)
    context = {'department': department, 'faculty': faculty, 'course': course, 'videos': videos,'showVideoModal':True,'showUpdateVideoModal':True,'showAddButton':True}
    return render(request, "lms/videos.html", context)



# For notes inside a lecture
@login_required
def lec_notes(request, dept_id, course_id, fac_id):
    department = Department.objects.get(id=dept_id)
    course = Course.objects.get(id=course_id)
    faculty = Faculty.objects.get(id=fac_id)
    notes = Note.objects.filter(faculty=faculty)
    context = {'department': department, 'faculty': faculty, 'course': course, 'notes': notes,'showNoteModal':True,'showUpdateNoteModal':True,'showAddButton':True}
    return render(request, "lms/notes.html", context)
```

6.2 Appointing Relationships

The **Appointing Relationships** between different entities are key for understanding how the system evolves through user-level interactions. For example:

- Users can appoint faculty, moderators, admins, and departments.
- The **Appoints** relationship helps the system model who can assign roles or actions to which entities (e.g., Admin appointing Moderators or Faculty members).

```python
@csrf_exempt
def appoint_user(request):

    if (request.user.role!= 'master')and (request.user.role!= 'admin'):
        return redirect('illegalactivity')
    if request.method == 'POST':
        try:
            data = json.loads(request.body)
            user_id = data.get('user_id')
            appoint_role = data.get('appoint_role')  # The role being assigned (admin, moderator, user)
            department_id = data.get('department_id')  # Department ID for Admins
            course_id = data.get('course_id')  # Course ID for Moderators
            user = User.objects.get(id=user_id)
            if appoint_role == 'admin':
                # Assign the user to a department for Admin role
                if department_id:
                    user.role = 'admin'
                    user.department = department_id
                    user.course = -1  # Remove course if admin is assigned
                else:
                    return JsonResponse({"status": "error", "message": "Department ID is required for Admin."})
            elif appoint_role == 'moderator':
                # Assign the user to a course for Moderator role
                if course_id:
                    user.role = 'mod'
                    user.course = course_id
                    user.department = -1  # Remove department if moderator is assigned
                else:
                    return JsonResponse({"status": "error", "message": "Course ID is required for Moderator."})
            elif appoint_role == 'user':
                user.role = 'user'
                user.department = -1
                user.course = -1

            else:
                return JsonResponse({"status": "error", "message": "Invalid role specified."})
            user.save()

            return JsonResponse({"status": "success", "message": f"Role {appoint_role} assigned successfully!"})

        except Exception as e:
            return JsonResponse({"status": "error", "message": f"Error: {str(e)}"})

    # If the request is not a POST
    return JsonResponse({"status": "error", "message": "Invalid request method."})
```

6.3 Creation and Deletion

The system allows entities to be created or deleted, such as **Courses**, **Faculties**, and **Departments** ,**Contents** through the respective classes (Admin, Master, Moderator). These operations are managed based on the user roles, ensuring that only users with appropriate permissions can create or delete important system entities.

```python
@login_required
def add_dept(request):
    if (request.user.role!='master'):
        return redirect('illegalactivity')
    if request.method == 'POST':
        dept_name = request.POST.get('departmentName')
        dept_desc = request.POST.get('departmentDescription')
        dept_image = request.FILES.get('departmentImage')
        if(dept_name  ):
            department=Department.objects.create( name=dept_name   )
            if(dept_image):
                department.image=dept_image
            if( dept_desc):
                department.description=dept_desc
        department.save()
        subject.notify(request, "Department has been added")
        return redirect('departments')

    return redirect('departments')

# Add Course
@login_required
def add_course(request, dept_id):
    if (request.user.role!='master')and(request.user.department!= dept_id):
        return redirect('illegalactivity')
    department=get_object_or_404(Department,id=dept_id)
    if request.method == 'POST':
        course_code = request.POST.get('courseCode')
        course_name = request.POST.get('courseName')
        course_desc = request.POST.get('courseDescription')
        course_image = request.FILES.get('courseImage')
        if(course_code and course_name):
            course=Course.objects.create(course_code=course_code, course_name=course_name,department=department)
            if course_desc:
                course.description=course_desc
            if course_image:
                course.image=course_image
        course.save()
        subject.notify(request, "Course has been added")
    return redirect('deptcourses',department.id)
```

```python
# Add Faculty
@login_required
def add_fac(request, dept_id, course_id):
    if (request.user.role!='master')and(request.user.department!= dept_id)and(request.user.course!= course_id):
        return redirect('illegalactivity')
    course = get_object_or_404(Course, id=course_id)
    if request.method == 'POST':
        faculty_name = request.POST.get('facultyName')
        position = request.POST.get('position')
        image= request.FILES.get('facultyImage')
        if(faculty_name and position):
            faculty=Faculty.objects.create(name=faculty_name, position=position,course=course)
            if image:
                faculty.image=image
            faculty.save()
        subject.notify(request, "Faculty has been added")
    return redirect('course_facs',dept_id,course_id)


# Add Slide
@login_required
def add_slide(request, dept_id, course_id, fac_id):
    faculty = get_object_or_404(Faculty, id=fac_id)
    if request.method == 'POST':
        slide_name = request.POST.get('slideName')
        slide_content = request.FILES.get('slideContent')
        if (request.user.role == 'master') or (request.user.department == dept_id) or (request.user.course == course_id):
            SlideFactory.create_slide(faculty, slide_name, slide_content)
            subject.notify(request, "Slide has been added")
        else:
            temp_slide=TemporarySlideFactory.create_temp_slide(real_instance=None, faculty=faculty, name=slide_name, content_file=slide_content)
            notification = Notification.objects.create(
                message=f"{request.user.first_name} {request.user.last_name} wants to add a slide in {temp_slide.faculty.course.department.name}/{temp_slide.faculty.course.course_nam
                sender=request.user,
                type="add",
                content_type="slide",
                content_id=temp_slide.id
            )
            receivers = User.objects.filter(Q(role='master') | Q(department=dept_id) | Q(course=course_id))
            print(receivers)
            notification.recievers.add(*receivers)
            notification.save()
            subject.notify(request, "Request sent")
    return redirect('lec_slides', dept_id, course_id, fac_id)
```

```python
def add_slide(request, dept_id, course_id, fac_id):
    faculty = get_object_or_404(Faculty, id=fac_id)
    if request.method == 'POST':
        slide_name = request.POST.get('slideName')
        slide_content = request.FILES.get('slideContent')
        if (request.user.role == 'master') or (request.user.department == dept_id) or (request.user.course == course_id):
            SlideFactory.create_slide(faculty, slide_name, slide_content)
            subject.notify(request, "Slide has been added")
        else:
            temp_slide=TemporarySlideFactory.create_temp_slide(real_instance=None, faculty=faculty, name=slide_name, content_file=slide_content)
            notification = Notification.objects.create(
                message=f"{request.user.first_name} {request.user.last_name} wants to add a slide in {temp_slide.faculty.course.department.name}/{temp_slide.faculty.course.course_nam
                sender=request.user,
                type="add",
                content_type="slide",
                content_id=temp_slide.id
            )
            receivers = User.objects.filter(Q(role='master') | Q(department=dept_id) | Q(course=course_id))
            print(receivers)
            notification.recievers.add(*receivers)
            notification.save()
            subject.notify(request, "Request sent")
    return redirect('lec_slides', dept_id, course_id, fac_id)
# Add Note
@login_required
def add_note(request, dept_id, course_id, fac_id):
    faculty = get_object_or_404(Faculty, id=fac_id)
    if request.method == 'POST':
        note_name = request.POST.get('noteName')
        note_content = request.FILES.get('noteContent')
        if (request.user.role == 'master') or (request.user.department == dept_id) or (request.user.course == course_id):
            NoteFactory.create_note( faculty, note_name, note_content)
            subject.notify(request, "Note has been added")
        else:
            temp_note=TemporaryNoteFactory.create_temp_note( real_instance=None, faculty=faculty, name=note_name, content_file=note_content)
            notification = Notification.objects.create(
                message=f"{request.user.first_name} {request.user.last_name} wants to add a note in {temp_note.faculty.course.department.name}/{temp_note.faculty.course.course_name}/
                sender=request.user,
                type="add",
                content_type="note",
                content_id=temp_note.id
            )
            receivers = User.objects.filter(Q(role='master') | Q(department=dept_id) | Q(course=course_id))
            notification.recievers.add(*receivers)
            notification.save()
            subject.notify(request, "Request sent")
    return redirect('lec_notes', dept_id, course_id, fac_id)

 # Add Video
 @login_required
 def add_video(request, dept_id, course_id, fac_id):
     faculty = get_object_or_404(Faculty, id=fac_id)
     if request.method == 'POST':
         video_name = request.POST.get('videoName')
         video_content = request.FILES.get('videoContent')
         if (request.user.role == 'master') or (request.user.department == dept_id) or (request.user.course == course_id):
             VideoFactory.create_video( faculty, video_name, video_content)
             subject.notify(request, "Video has been added")
         else:
             temp_video=TemporaryVideoFactory.create_temp_video( real_instance=None, faculty=faculty, name=video_name, content_file=video_content)
             notification = Notification.objects.create(
                 message=f"{request.user.first_name} {request.user.last_name} wants to add a video in {temp_video.faculty.course.department.name}/{temp_video.faculty.course.course_nam
                 sender=request.user,
                 type="add",
                 content_type="video",
                 content_id=temp_video.id
             )
             receivers = User.objects.filter(Q(role='master') | Q(department=dept_id) | Q(course=course_id))
             notification.recievers.add(*receivers)
             notification.save()
             subject.notify(request, "Request sent")
     return redirect('lec_videos', dept_id, course_id, fac_id)
```

```python
@login_required
def delete_dept(request,dept_id):
    if (request.user.role!='master')&(request.user.department!=dept_id):
        return redirect('illegalactivity')
    if request.user.role=='master':
        department = Department.objects.get(id=dept_id)
        department.delete()
        subject.notify(request, "Department has been deleted")
    else:
        subject.notify(request, "Request Sent")

    return redirect('departments')

@login_required
def delete_course(request,dept_id,course_id):
    if (request.user.role!='master')and(request.user.department!= dept_id)and(request.user.course!= course_id):
        return redirect('illegalactivity')
    department=get_object_or_404(Department,id=dept_id)
    course = get_object_or_404(Course, id=course_id)
    if((request.user.role=='master')or(request.user.department== dept_id)):
        course.delete()
        subject.notify(request, "Course has been deleted")
    else:
        subject.notify(request, "Request Sent")
    return redirect('deptcourses',department.id)



@login_required
def delete_fac(request,dept_id,course_id,fac_id):
    if (request.user.role!='master')and(request.user.department!= dept_id)and(request.user.course!= course_id):
        return redirect('illegalactivity')
    faculty=get_object_or_404(Faculty,id=fac_id)
    faculty.delete()
    subject.notify(request, "Faculty has been deleted")
    return redirect('course_facs',dept_id,course_id)
```

```python
# Delete Slide
@login_required
def delete_slide(request, dept_id, course_id, fac_id, slide_id):
    slide = get_object_or_404(Slide, id=slide_id)
    if (request.user.role == 'master') or (request.user.department == dept_id) or (request.user.course == course_id):
        Notification.objects.filter(Q(real_content_id=slide.id)&Q(content_type="slide")).delete()
        slide.delete()
        subject.notify(request, "Slide has been deleted")
    else:

        notification = Notification.objects.create(
            message=f"{request.user.first_name} {request.user.last_name} wants to delete a slide in {slide.faculty.course.department.name}/{slide.faculty.course.course_name}/{slide.f
            sender=request.user,
            type="delete",
            content_type="slide",
            real_content_id=slide.id
        )
        receivers = User.objects.filter(Q(role='master') | Q(department=dept_id) | Q(course=course_id))
        notification.recievers.add(*receivers)
        notification.save()
        subject.notify(request, "Request sent")

    return redirect('lec_slides', dept_id, course_id, fac_id)
# Delete Note
@login_required
def delete_note(request, dept_id, course_id, fac_id, note_id):
    note = get_object_or_404(Note, id=note_id)
    if (request.user.role == 'master') or (request.user.department == dept_id) or (request.user.course == course_id):
        Notification.objects.filter(Q(real_content_id=note.id)&Q(content_type="note")).delete()
        note.delete()
        subject.notify(request, "Note has been deleted")
    else:

        notification = Notification.objects.create(
            message=f"{request.user.first_name} {request.user.last_name} wants to delete a note in {note.faculty.course.department.name}/{note.faculty.course.course_name}/{note.facul
            sender=request.user,
            type="delete",
            content_type="note",
            real_content_id=note.id
        )
        receivers = User.objects.filter(Q(role='master') | Q(department=dept_id) | Q(course=course_id))
        notification.recievers.add(*receivers)
        notification.save()
        subject.notify(request, "Request sent")
    return redirect('lec_notes', dept_id, course_id, fac_id)
```

```python
# Delete Video
@login_required
def delete_video(request, dept_id, course_id, fac_id, video_id):
    video = get_object_or_404(Video, id=video_id)
    if (request.user.role == 'master') or (request.user.department == dept_id) or (request.user.course == course_id):
        Notification.objects.filter(Q(real_content_id=video.id)&Q(content_type="video")).delete()
        video.delete()
        subject.notify(request, "Video has been deleted")
    else:

        notification = Notification.objects.create(
            message=f"{request.user.first_name} {request.user.last_name} wants to delete a video in {video.faculty.course.department.name}/{video.faculty.course.course_name}/{video.f
            sender=request.user,
            type="delete",
            content_type="video",
            real_content_id=video.id
        )
        receivers = User.objects.filter(Q(role='master') | Q(department=dept_id) | Q(course=course_id))
        notification.recievers.add(*receivers)
        notification.save()
        subject.notify(request, "Request sent")
    return redirect('lec_videos', dept_id, course_id, fac_id)
```

Summary

By introducing the role attribute in the User Class, the system enables flexible and scalable user permission management, eliminating the need for separate classes for each role. This role-based approach allows users to perform role-specific actions such as managing lectures, videos, and courses.

The system models important relationships between entities such as Lectures, Videos, Notes, Courses, Faculty, and Departments. It also clearly defines appointment relationships and supports the creation and deletion of entities based on user permissions. This refactoring significantly reduces the complexity of the system, enhances its scalability, and makes it easier to maintain and expand in the future.

Conclusion

OpenEdu exemplifies how open-source technology and thoughtful software design can be combined to build a scalable and maintainable educational platform. By leveraging a unified user model, the platform minimizes redundancy and enhances scalability through role-based logic. Design patterns such as Singleton, Strategy, Factory, and Facade reinforce code reusability and clean architecture.

The platform addresses both functional and non-functional requirements critical to modern learning environments—ranging from content creation and user management to system security and data integrity. With its flexible, modular structure, OpenEdu is poised to support future academic needs and foster collaboration among students, faculty, and administrators. Ultimately, this project represents a step toward democratizing digital education through an accessible, extensible, and ethical platform.

Github Link: [GitHub - EzazAsif/OpenEdu-A-Comprehensive-Open-Source-Learning-Platform]