1) Introduction

In this advanced web development project, I designed and developed "ClassNet," an e-Learning app catering to the needs of both students and teachers. Depending on user type, ClassNet allows users to register, log in/out, change passwords, post status updates, chat, receive notifications, create courses and study materials, search for users, view and update profiles, enrol/unenrol in courses, provide feedback, and track progress.

These features were designed and developed with usability and scalability in mind, confirming that the “ClassNet” can grow to accommodate more users and courses over time. “ClassNet” implementation decision focuses on intuitive user interface, secure authentication, and real-time communication to enable seamless interaction between students and teachers.

2) User Account Management

A. Feature description and Logic

a) Users can Create Accounts (Registration)

Both students and teachers can create accounts. Upon visiting the registration page, they are prompted to input essential details such as name, email address, password, profile picture and role. All fields are validated (e.g., email format) to prevent errors. Passwords are hashed to securely store them in an irreversible format, preventing exposure of sensitive information. Upon user submitting the form successfully, details are saved to the user database, and they are logged in. After registration, users can view or update the profile.

b) Users can Log In and Log Out (Authentication)

A login page with form validation and session handling is implemented. Upon logging out, the session is destroyed, and the user is redirected to the login screen. Login verifies the provided credentials by the user against the stored data. Upon successful authentication, a session is created, allowing access to their respective dashboard. Also, logout is implemented to ensure end of the session.

B. Design & Implementation Decision

Design leverages Django’s authentication framework, utilizing its built-in tools and views for security and ease of integration. The project follows a standard Django structure.

A custom user model extends Django’s default User model, adding fields like username, password, email, user type, and profile picture. Custom forms handle registration, login, and logout, with validation using Django's ModelForm and AuthenticationForm. Views for these actions follow Django conventions and are linked to the respective templates. Each view is mapped to a URL pattern for registration, login, logout, password changes, and profile updates. Django's templating engine is used to render HTML forms and display error messages.

3) Teachers Functionalities

A. Feature description and Logic

a) Search for Students and Other Teachers

Teachers are provided with a search functionality that allows them to search for both students and other teachers stored in database based on details like name or email. A search API queries the database requiring at least three letters to ensure more accurate results and improved performance. The search bar filters the results and teachers can easily find the profile of any user in the system. Teacher can search specific students or other teachers that uses Django's filter () function on the Teacher and Student profile to retrieve matching records.

b) Add New Courses

Teachers can produce new courses by providing pertinent details such as course name, description, and any related files or resources. They have a "Create Course" form in the dashboard where they input course details, which are stored in the course and material database. An endpoint allows teachers to view and manage their courses through their dashboard. Teachers can add new courses through form and a foreign key relationship is created between teacher and the course model to link teacher with their respective courses.

c) Remove / Block Students

Teachers can remove or block students from their courses. This action prevents the student from accessing course materials and receiving further notifications regarding the course. Students cannot enrol again to the course until teacher revokes the block. Blocking a student is done by adding a flag in the StudentCourseEnrollment model designates whether a student is blocked from the course.

d) Add teaching material

Teachers are allowed to upload teaching materials (such as PDFs, presentations, and documents) to their course page. A file upload feature allows teachers to attach files. These materials are accessible to enrolled students through the course view page, allowing students to download and view relevant content.

B. Design and Implementation decision

The application leverages Django's robust backend framework to ensure seamless and intuitive experience for teachers. The design focus on simplicity, scalability and security. ClassNet follows Django's best practices with clear separation of concerns.

Pagination is used to break large sets of data into smaller, more manageable pages, that has improved both performance and user experience. For courses, pagination ensured that only a limited number of courses (e.g., 5 per page) are displayed at a time. This has reduced load times, made navigation easier and allowed users to explore content without overwhelming the page.

Course creation is managed by DRF (Django Rest Framework) that build APIs for user enrolment and chat message handling. Serializers are used to handle the input and output data for these operations. Django models store the necessary data, including course, user, and message details.

Model for Teacher, Student, Course, StudentCourseEnrollment and TeachingMaterial are used to implement these features. Search view and forms allows teachers to search for students and other teachers, add course view allows teachers to add new course, Remove or block student views allow teachers to remove or block students from their courses and add material view allows teachers to upload files and manage teaching material.

4) Student Functionalities

A. Feature description and Logic

a) Enrol on a course offered by teachers

A student dashboard lists available courses and an "Enroll" button kicks in enrolment process. The student's data is added to the course's student list, and a notification is sent to the teacher through a frontend notification system. A student is able to clearly see what courses are available, who is teaching them and key details like course description to make informed decisions about which course to enrol in.

b) View teaching material, mark them as complete to track progress

Students can view set of teaching materials for each course and track their progress by marking each material as completed. The progress bar will visually display the percentage of completed materials relative to total number of materials. Upon successful completion of material dashboard automatically updates "course progress" allowing students to dynamically track progress and stay motivated. This also helps teachers to monitor performance of the student. System will display a list of teaching materials for each course, student mark individual teaching material as completed by clicking a checkbox. The system will calculate the percentage of completed material to display a progress bar.

B. Design and Implementation decision

The design of these features focuses on both efficiency and user-friendliness. The core functionality, student enrolment in courses, ensures clarity and ease of use for both students and teachers. The system is designed to be student-centric, allowing students to view their enrolled courses and unenrol if desired.

Django's ORM is used to store student-course relationships through a many-to-many model, ensuring scalability as the number of students and courses grows. The Student Course Enrolment model tracks enrolments, and efficient database queries are achieved by leveraging Django's ORM and filtering capabilities, minimizing load by retrieving only necessary data.

Models like Course, TeachingMaterial (representing content such as videos, PDFs, images), and TeachingMaterialCompletion (tracking student progress) implement this functionality. Django views handle logic for displaying courses, processing enrolments, showing materials, and calculating student progress based on completed materials. The views are simple, readable, and maintain a clear separation of concerns.

Templates display materials with checkboxes to mark completion, and a progress bar adjusts dynamically based on the completion percentage. This separation of concerns makes the code modular and maintainable. Only authenticated students can enrol, ensuring they are logged in before participating.

.

5) Student submit feedback for a course

A. Feature description and Logic

When a course is completed, students can submit feedback through a form about their experience. The feedback in the form of rating from 1(Excellent) to 5(Poor) and comments is stored in the database and can be retrieved later for analysis by teachers to improve course quality.

The feedback form is designed to be simple to fill out with rating from 1 to 5 and optional comment, only students enrolled in the course after completion should be able to submit the feedback to ensure integrity of feedback is valuable and reflective of their overall experience. After submission students are also able to edit the feedback to reflect updated thoughts.

B. Design and Implementation Decision

Models used to track the feedback provided by students are “Course” representing course offered in the system, “Course Feedback” tracks feedback for a course includes rating field to allow a rating between 1 to 5 and clean method ensures that the student can only submit feedback per course

Views submit_feedback renders a form that allows student to submit or edit feedback. If the student has already submitted feedback, the form will display existing feedback allowing them to update. After the form is submitted, the feedback is updated or created and student is redirected to course page.

Templates Submit_feedback.html renders form where students can rate the course and course_feedback_detail.html displays the list of all feedback for a course.

The views are separated into two distinct functions ne for submitting feedback and one for displaying feedback details to ensure modular approach and separation of concern for clear and manageable code.

6) Communication

A. Feature description and Logic

a) Real-time group chat

Teachers and students can use real-time chat feature for discussions related to coursework, queries and feedback. Feature allows them to send and receive messages instantly to the individuals in a group chats. When the teacher creates a course, a group chat room is directly created with the same name as course making it easy for both students and teachers to access and participate in course-related discussions. This ensures all relevant discussion happen in a context that is identifiable and accessible through centralised communication in a structured, course-based environment. The conversation is saved in the database along with associated user accounts to display the history of last 10 latest conversation.

b) Add status updates

Both students and teachers can add personal status updates to their home page. This feature allows them to share thoughts, announcements, or general status updates that fosters engagement and creates a sense of community. A "Status Update" section is available on the dashboard page that enhances communication, as both parties can stay informed about important news or individual updates helping to strengthen their connection and collaboration. These are stored in the database where teachers can view all updates and students can view their own.

B. Design and Implementation Decision

Real time messaging ensures that students and teachers can interact without delays, improving learning experience. Django Channels enables WebSocket connections allowing persistent real-time messaging between client-server and low-latency in communication.

Students who are enrolled in the course can join the WebSocket group, allowing them to participate in real-time communication. Messages sent by either the teacher or students are broadcast to all connected users in the group.

Django Channels, Redis, and WebSockets, make chat highly scalable and capable of handling concurrent users in real-time effectively. Redis is used as the message broker for Django Channels to handle concurrent users without blocking other processes.

Forms allow teachers to trigger the automatic creation of a corresponding group chat when course is created. Validators ensure the integrity of the data, ensuring the course name is unique and valid.

7) Notification

A. Feature description and Logic

a) Student enrol on a Course, Teacher is Notified

Whenever a student enrols in a course, the teacher is notified through the application's notification system using Django's signals. This helps teachers manage their courses effectively and track student engagement. A notification is triggered using a push notification within the app to alert the teacher about the new student upon successful enrolment. When a teacher views their notifications, they see a list of notification instances with a button or link next to each notification that allows them to mark it as read.

b) New material added to a Course, Student is Notified

Whenever a teacher adds new teaching materials or resources to a course, all enrolled students are notified about the update using Django's signals. This ensures students stay informed about new content and updates related to their courses. An event listener tracks file uploads or content changes. When new materials are added, an automatic notification is sent to all enrolled students, either by in-app push notification.

B. Design and Implementation Decision

The design revolves around the Student-Course-Teacher relationship, where a student can enrol in multiple courses, each taught by a teacher. This setup ensures automation and keeps both teachers and students up-to-date. Notifications are stored in the TeacherNotification and StudentNotification models, linked to the User model via a foreign key. The limit_choices_to = {'is_staff': true/false} argument ensures that only teachers or students can be selected for the respective notification. The message field holds the notification content, while the date_created field automatically records the timestamp.

Using Django's post_save signal, the system triggers actions after saving a StudentEnrollment or CourseMaterial object. When a new enrolment or material is created, the notify_teacher_on_enrollment or notify_student_on_change_of_material functions are activated, generating notifications and creating corresponding TeacherNotification or StudentNotification instances. This decouples the notification logic from the core application flow, allowing the system to focus on enrolment and material creation, reducing the risk of missing important updates.

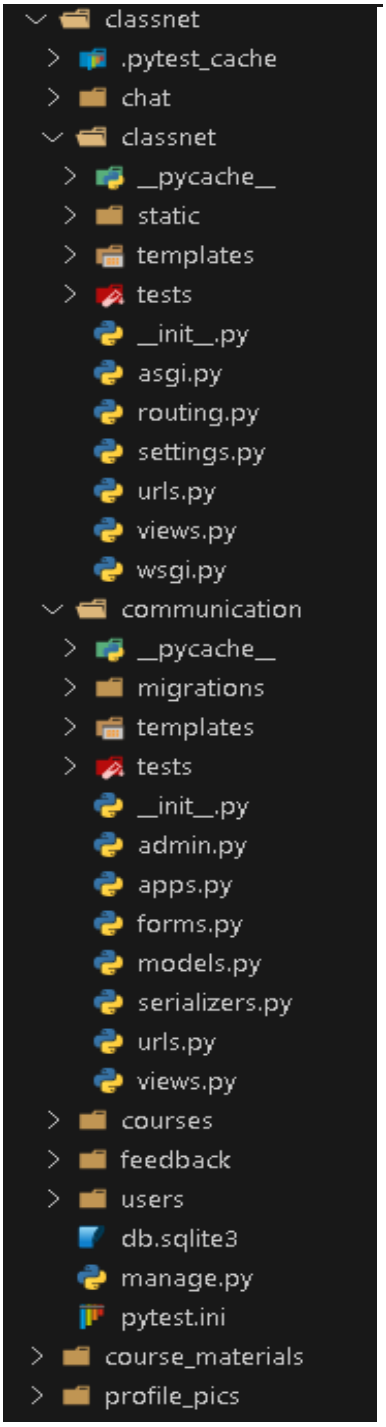
The CourseMaterial model stores information about materials uploaded by teachers, including the course (ForeignKey to the Course model), material type (e.g., file, video, link), and content. Django's routing, views, and models ensure secure access for authorized users, like teachers. A view to mark notifications as read accepts a notification ID, marking it as read and removing it from the view, enhancing the user experience.

.

8) Implementation Details

A. Directory and code structure

The application is organised in a single Git repository to allow centralised management of the entire project, making it easier to track changes. The project is divided into several Django apps, users manage authentication, registration, profiles and roles, chat handles real time messaging and chat rooms, communication manages notifications between users, courses handle course creation, enrolment and tracking student progress, feedback collects and processes user feedback including course evaluation and reviews. Each app is modular with its own model, views, templates, serializers, forms and URLs ensuring clean, maintainable code structure. Shared resources like static files are kept in common directory classnet.



a) Models and Migrations

ClassNet uses Django's ORM to model data, with models for User, Course, Enrolment, Feedback, and Message. Users, categorized as students or teachers, have distinct roles—teachers manage courses and materials, while students enrol, track progress, and provide feedback. Real-time communication is enabled through the Message model. Migrations manage the database schema, ensuring smooth creation and updates.

b) Forms, Validators, and Serialization

Forms were used to manage user input for registration, login, and course creation, with input validation ensuring proper formats for email, password strength, and unique course names. Django's built-in form validation methods enforced these rules. For API interactions, Django REST Framework's serializers were employed to serialize and deserialize data, enabling smooth data transfer between the client and server in JSON format. This approach ensured data integrity and streamlined the process of handling user input, making the system more robust and efficient.

c) Django REST Framework (DRF)

I implemented a RESTful API to allow users to access their data programmatically. DRF was used to create API endpoints for registering, logging in, enrolling in courses, submitting feedback, and sending messages. These endpoints use the HTTP methods (GET, POST, DELETE) to interact with the models.

d) URL Routing

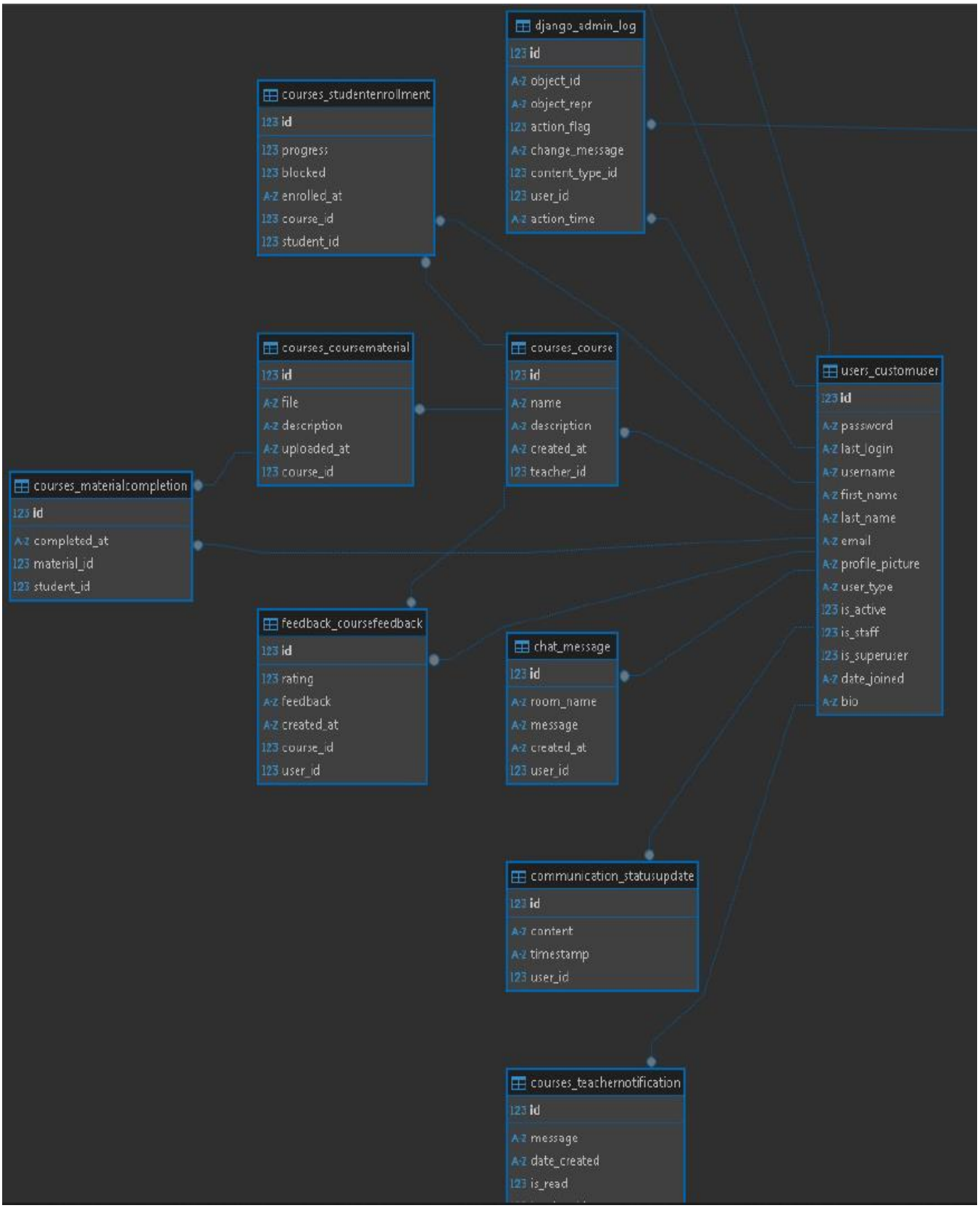
I defined clean and meaningful URL patterns using Django's urls.py. This includes routes for both web views (e.g., course homepages, user dashboards) and API endpoints (e.g., creating a course, enrolling in a course). DRF's routers were used to automatically generate RESTful routes.

e) Unit Testing

For this project, I used pytest and pytest-django to implement unit testing for various features, ensuring correct behavior and edge case handling. Tests were organized into separate modules based on functionality, making them easier to manage and scale, while pytest provided an efficient framework for maintaining application integrity through automated testing.

9) Models and data structure

A. Entity relationship diagram



Initially, I experimented with Django's built-in User model to gain hands-on experience with authentication and user management. This allowed me to explore features like registration, login, and user permissions. It was a great starting point to understand how Django handles user-related functionalities before customizing models for specific applications. to gain experience.

Users: This custom Django User model extends AbstractBaseUser and PermissionsMixin to provide advanced user management. It includes fields such as username, first_name, last_name, email, and profile_picture, along with a custom user_type (student/teacher). The model also allows for an optional bio field, giving users a chance to add personal information. The CustomUserManager class is used to create regular users and superusers, ensuring proper password hashing and field validation. Custom permissions, such as can_view_student_data, are integrated to control user access. This model supports authentication, role-based access, and personalization for user profiles in a Django application.

Courses: This Django model represents a comprehensive course management system. It includes models for Course, StudentEnrollment, CourseMaterial, MaterialCompletion, and notifications for both teachers and students. The Course model stores course details and links to a teacher. The StudentEnrollment model tracks students' enrollments and progress in courses. The CourseMaterial model manages resources for courses, and MaterialCompletion records when students finish materials. Notifications are stored for both teachers and students to ensure communication. Each model uses relationships like ForeignKey for linking users and courses, providing a robust structure for managing courses, materials, and user interactions.

Feedback: The CourseFeedback model allows users to provide feedback on courses. It includes a rating field with predefined choices (Excellent, Very Good, Good, Bad, Very Bad) and a text feedback field. It also stores the created_at timestamp. The model links to both Course and User models, ensuring proper relationships.

Chat: The Message model stores chat messages within specific rooms. It includes a room_name to identify the chat room, a message field for the content, and a user field linking to the User model. The created at field automatically stores the timestamp when the message is created.

Communication: The StatusUpdate model stores user status updates with user, content, and timestamp fields. It is ordered by timestamp, showing the most recent updates first.

10) Data access control

In developing an eLearning app classnet, ensuring proper control over data visibility and modifiability was a top priority. Since the platform stores data for many different users, including students, teachers, and courses, it was essential to ensure that each user could only access their own data or data they were authorized to view. For instance, student should be able to see their own progress, grades, and enrolled courses but not access other students' data. Similarly, teachers should only have access to the students and course materials they manage.

To achieve this, I created two serializers: UserSerializer for detailed user information and UserSerializerRestricted for situations requiring limited access. The restricted version only displayed basic details, such as the user's name and course progress, without exposing sensitive data like grades or enrolment in other courses.

However, these serializers were not designed to inherit from one another, which led to some redundancy. Ideally, UserSerializer should extend UserSerializerRestricted, allowing for a more layered structure. This would reduce code duplication, streamline the management of different access levels, and make it easier to enforce privacy and security. By structuring the serializers this way, we can better control what data users can view, ensuring a secure and user-specific experience.

11) REST API and other endpoints and data structure

Endpoint	HTTP Methods	Description
/	GET	Renders home page of the application
/admin/	GET	Admin panel for managing the app
/users/	**Varies**	**Includers uer-related routes**
/register/	GET, POST	User registration page to create a new account
/profile/	GET, POST	User profile page to view and edit own profile
/profile/<user_id>/	GET	View another user's profile by user_id
/login/	GET, POST	Login page to authenticate users
/logout/	POST	Logout the current user
/change_password/	GET, POST	Change password page for current user
/student/	GET	Student home page for student specific views
/teacher/	GET	Teacher home page for teacher specific views
/search/	GET	Search for users or profiles
/api/users	GET, POST	API endpoint to list users or create a new user
/api/users/<username>	GET,POST	API endpoint to retrieve or update a specific user by username
/courses/	**Varies**	**Includes course related routes**
/create/	GET POST	Create a new course
/course/<course_id>/	GET	View details of a specific course by course_id
/available_courses	GET	List available courses
/enroll/course_id	POST	Enroll in a specific course by course_id
/unenroll/course_id	POST	Unenroll from a specific course by course_id
/course/<course_id>/add-material	POST	Add new material to a specific course by course_id
/mark_material/<material_id>/completed/	POST	Mark a material as completed by material_id
/course/delete/<course_id>/	DELETE	Delete a specific course by course_id
/teacher/block/<course_id>/<student_id>/	POST	Block a student from course by course_id and student_id
/teacher/unblock/<course_id>/<student_id>/	POST	Unblock a student from course by course_id and student_id
/teacher/remove/<course_id>/<student_id>/	DELETE	Remove a student from course by course_id & student_id
/notifications/mark_as_read_teacher_notifications/<notification_id>/	POST	Mark student notification as read by notification_id
/notifications/mark_as_read_student_notifications/<notification_id>/	POST	Mark teacher notification as read by notification_id

/api/courses/	GET POST	List courses or create a new course via the API.
/api/enrollments/	GET POST	List enrollments or create a new enrolment via the API.
/api/materials/	GET POST	List course materials or create a new material via the API.
/api/material-completions/	GET POST	List material completions or create a new material completion via the API.
/api/material-completions/	GET POST	List teacher notifications or create a new teacher notification via the API.
/api/student-notifications/	GET POST	List teacher notifications or create a new teacher notification via the API.
/communication/	**Varies**	**Includes communication related routes**
/status_update	POST	Adds new status update to the system
api/status_updates/	GET	Retrieves a list of all status updates
/chat/	**Varies**	**Includes chat related routes**
/room_name	GET	Display content related to a specific room where room_name is dynamic paramater
/api/messages	GET	Retrieves a list of messages
/feedback/	**Varies**	**Includes feedback related routes**
/courses/<course_id>/feedback/	GET POST	View or submit feedback for a specific course (course_id).
/api/feedback/	GET POST	Retrieve or create course feedback via API

12) Development Environment

The ClassNet is built around Django, a powerful all-in-one web framework written in Python. Django has advanced set of tools for designing backend APIs including routing, models, ORM, database auto migrations and more. It also has built in templating system for frontend functionality. I used Django Rest Framework (DRF) for REST API endpoints.

Chat functionality was implemented using Django Channels, supporting protocols beyond HTTP via ASGI. The chat is powered by Daphne, an ASGI web server for WebSocket connections, with Redis acting as an in-memory message broker for async tasks and queues. Redis manages the Celery task queue, enabling long-running tasks like sending notifications and saving data asynchronously, ensuring the chat app remains responsive.

For the front-end, I used Django's views and templating to create a powerful and flexible user interface, separating business logic from presentation. Function-based views provided simplicity and clarity, while template inheritance allowed for a consistent layout across pages like student and teacher dashboards.

Django signals were employed for automatic notifications, triggered by events such as student enrolment or new course material creation. Signals enable decoupled communication between system components, allowing notifications for teachers and students without modifying views or business logic.

Operating System: Windows 10 / Ubuntu 24.04 on AWS

- Python Version: Python 3.13.0
- Django Version: 5.1.6
- Django REST Framework Version: 3.15.2
- Django Channels Version: 4.2.0
- Database: db. sqllite3: 2.6.0
- Packages: details are in requirements.txt file
    - celery==5.4.0
    - channels==4.2.0
    - cryptography==44.0.1
    - daphne==4.1.2
    - Django==5.1.6
    - django-bootstrap4==24.4
    - djangorestframework==3.15.2
    - redis==5.2.1
    - uritemplate==4.1.1
    - urllib3==2.3.0
    - websockets==14.2
    - pytest-django

13) Version control

Even though I didn't release any versions yet, I used GitHub to keep a history of changes to the codebase. Since I was the only one developing the application, I didn't create separate branches and merge requests and instead directly committed changes to the main branch. A log of my changes may be obtained by running git log in terminal.

14) How to Run ClassNet

A. Local Server

To run the application on local development environment, we need to separately run Django application and redis server.

Ensure python is installed and working. To check please run the following command in terminal:

- python –version

It is also recommended to use python virtual environment to install Django dependencies. The details are located at https://docs.python.org/3/library/venv.html.

Redis server installation:

- On Linux, redis is installed from the package manager:
- sudo apt update
- sudo apt install redis-server
- sudo systemctl start redis
- Check if Redis is running:
  - sudo systemctl status redis
- Verify if redis is still running
  - redis-cli

Django application Installation and Setup Instructions

- Unzip the project classnet
- Navigate to classnet
- Install Django dependencies:
  - pip install -r requirements.txt
- Run the development server:
  - python manage.py runserver

Logging into Site: Home page can be accessed through: http://localhost:8000/

Teacher and Student Login Credentials

- **Teacher**: teacher, password: password123
- **Student**: student, password: password123

==========

Note: Database is already setup in sqlitedb3 – Hence don't have to run below commands:

Setting up database and creating superusers

- Set up the database:
  - Python manage.py makemigrations
  - python manage.py migrate
- Create a superuser:
  - python manage.py createsuperuser
  - Follow the prompts to create the admin user.

B. AWS cloud

Classnet Django application is deployed to an AWS EC2 instance.

All the necessary Django dependencies are installed and dataset is migrated.

This classnet's home page can be accessed using following url:

- http://35.179.227.81:8000/

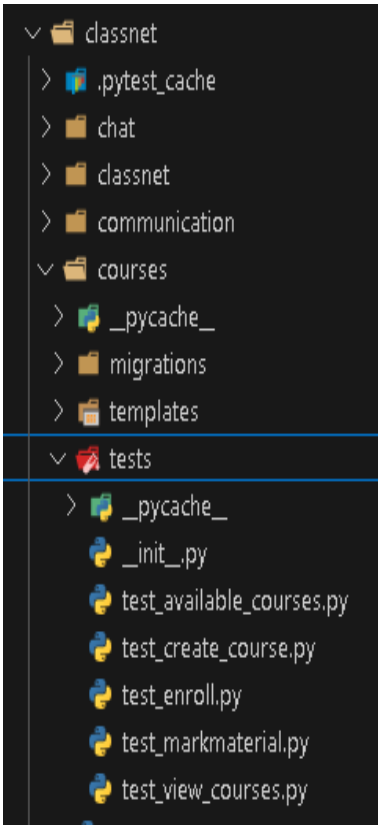
Application admin page can be accessed using:

- http://35.179.227.81:8000/admin/
- Username: admin
- Password: password

15) Unit testing

For this project, I used pytest and pytest-django to implement unit testing for all the features. Each feature such as users, courses, enrolment, chat, notifications, and profiles, was tested to ensure correct behaviour and edge case handling. Tests were organized into separate modules based on functionality, making them easier to manage and scale. pytest and pytest-django provided an efficient framework for maintaining application integrity through automated testing. By writing focused, isolated tests for each feature, I could catch bugs early, improve code quality, and maintain a reliable ClassNet. The use of pytest also streamlined the testing process with features like simple assertions, automatic test discovery, and a powerful plugin system, making it an excellent choice for Django-based applications.

Unit tests are run by using following command on local environment: pytest



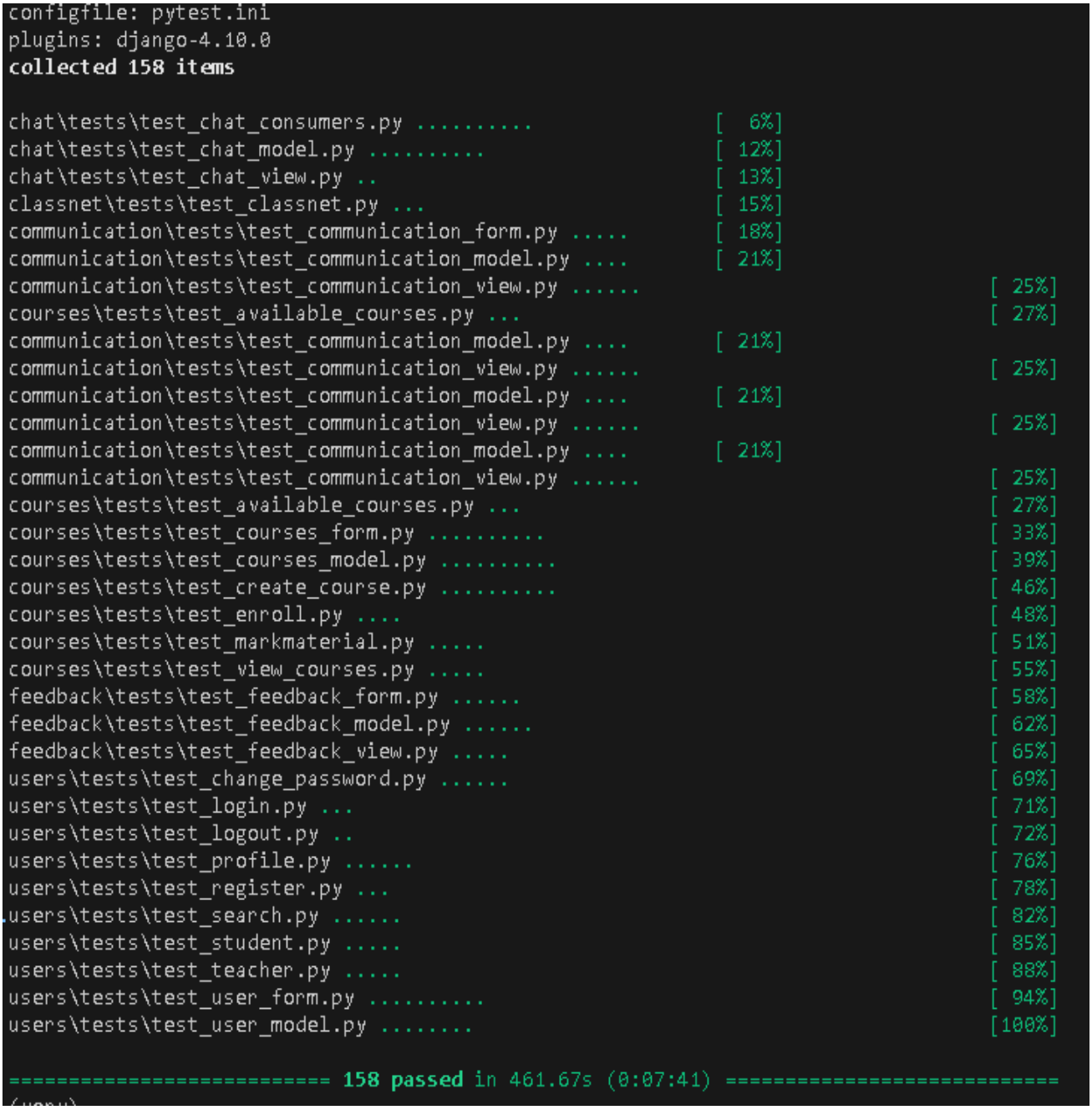
Tests in Django are typically structured within directory called tests inside each app directory. This directory contains test files and each test file begin with test_ to ensure Django recognise it as a test method.

The setup method is used to create and initialize the data or objects needed for the test case. It's called before each test method is executed.

The test method send request, check status code and assert that the correct data is present and verify behaviour.

```
class EnrollInCourseTestCase(TestCase):

    def setUp(self):...

    def test_enroll_in_course_authenticated_student(self):...

    def test_enroll_in_course_non_authenticated_user(self):...

    def test_enroll_in_course_already_enrolled_student(self):...

    def test_enroll_in_course_non_student_user(self):...
```

A. PyTest results:



16) Critical Evaluation

A. What Worked Well

a) User Authentication:

Django's permissions system, along with decorators like @login_required and class-based permissions in DRF, provided me with fine-grained control over what users could see or do within the app. By utilizing these features, I was able to easily restrict access to certain views and resources based on user roles and authentication status. For example, using @login_required ensured that only authenticated users could access specific pages, while class-based permissions in DRF allowed me to define custom permissions for API endpoints. This made it simple for me to enforce security rules and ensure users could only perform actions they were authorized to.

b) Real-time Chat:

Using Django Channels and WebSockets, I enabled smooth, real-time messaging between users. The integration of course creation with automatic group chat creation ensured a seamless connection between the course and its respective chat room. Redis and Channels allowed the system to scale efficiently, handling large user bases and multiple courses. By storing messages in a persistent database, I ensured that users could review past conversations whenever needed.

c) Notifications:

Django signals made it easy for me to notify users of events like course enrolment or new materials, automating the notification process. When teachers uploaded new materials, the system automatically notified all students. By decoupling the notification logic from the main application flow, I kept the codebase clean and maintainable. Real-time notifications through Django Channels or other mechanisms provided an interactive experience, ensuring that teachers were instantly informed of new enrolments without any additional manual effort.

d) Responsive and user-friendly UI:

I found that Django templates were highly flexible, allowing me to dynamically generate pages with complex logic. This feature enabled me to manage content and structure efficiently without sacrificing performance. By using Django's built-in template tags and filters, I could display data conditionally, loop through collections, and render content seamlessly.

The template system's integration with Django models made it easy to access and display data from the database. This flexibility, combined with Django's robust view and URL routing system, gave me a solid foundation to build interactive, data-driven applications that responded efficiently to user input and requests.

e) Security features:

Django provided secure password hashing by default, ensuring that user passwords were stored safely and protected from unauthorized access. It also securely handled session data, which enhanced the app's overall security. Additionally, Django prevented CSRF attacks by using a unique token for each user session, verifying it with each data-modifying request. If the token was missing or incorrect, Django denied the request, preventing unauthorized actions. I found these built-in security features to be highly effective in safeguarding user data and maintaining the integrity of the application.

B. Areas for Improvement

a) Scalability:

As user numbers grow, I'd optimize database queries and introduce caching mechanisms like Redis or Memcached to reduce redundant calls and improve response times. Additionally, I'd implement indexing and query optimization to minimize database load, ensuring the application remains performant and scalable with increased traffic.

To further enhance scalability, I would focus on optimizing database queries. Using techniques like indexing and query optimization could minimize database load and ensure faster data retrieval.

Additionally, employing query techniques like select related and prefetch related in Django would reduce the number of database queries and optimize data fetching, improving the app's performance.

b) Real-time Notifications:

To improve notification efficiency, I considered using Celery for batch processing or queuing notifications. This would reduce server burden by sending notifications in batches and only to students who haven't been notified yet, preventing duplicates and ensuring the system scales without overloading resources, especially for large courses.

In addition to making notifications more immediate, I also recognized that the message content could be more specific. For example, rather than sending a generic notification about new materials, I could include a direct link to the resource. This would make it easier for users to access the material directly from the notification, improving their overall experience and efficiency.

To improve user experience, I considered allowing users to customize notification preferences, enabling them to mute notifications for specific courses or materials like quizzes and assignments, reducing notification fatigue and increasing satisfaction.

c) Chat

I would enhance the chat functionality to allow for group chats and file-sharing. The initial design did not focus file-sharing functionality, which is often an essential part of course-based communication. Integrating file upload and download capabilities would be a great improvement.

While basic access control for teachers and students was implemented, adding granular role management, like moderating or muting users, would improve the system. Enhancements in message storage, file sharing, and role management would further elevate user experience, building on the successful real-time communication feature for courses.

d) Frontend

Currently, I used Django templates for the frontend, but in the future, I plan to integrate React.js within a Next.js application to improve frontend structure by encapsulating UI markup and logic in components using JSX. I'll use TypeScript for strong typing, minimizing errors and enhancing maintainability. This will improve scalability and make the codebase more organized and efficient as the project grows.

I also plan to improve performance by optimizing the app's data fetching process. Instead of loading all data at once, I would implement lazy loading with React.js, ensuring that only necessary content is loaded initially. This would improve page load times and overall user experience, especially for large applications.

e) Security of APIs

I would enhance the security of APIs by implementing rate limiting to prevent abuse and DDoS attacks. Additionally, I plan to ensure that all API endpoints are protected with proper authentication mechanisms, like OAuth or JWT, to verify user identities. I also intend to use HTTPS for all communications to encrypt sensitive data. Further, I would implement strict input validation and sanitize user inputs to prevent SQL injection and other security vulnerabilities. Lastly, I will integrate API logging and monitoring to detect and respond to potential security breaches more effectively.

f) Implementation of Docker for Streamlining Development and Deployment

This approach will help streamline the development, testing, and deployment processes by containerizing the Django app. Docker ensures consistency across different environments, simplifies dependency management, and improves scalability. It also makes deployment more efficient, enabling smoother transitions between development, staging, and production environments.

g) Amazon EC2

To enhance my EC2 setup, I plan to integrate Elastic Load Balancer (ELB) for improved traffic distribution across multiple instances. This will ensure better availability, fault tolerance, and scalability, preventing any single instance from being overloaded and allowing the application to handle higher traffic volumes efficiently.

I plan to integrate Amazon API Gateway for efficient API management, allowing secure creation, deployment, and monitoring of APIs. It will help throttle requests, manage traffic, and provide a secure backend interface.

I'll improve EC2 security by configuring security groups and NACLs for traffic control, implementing IAM roles for granular permissions, and using VPC to isolate the network and manage secure resource communication.

These improvements would make the EC2 setup more resilient, scalable, and secure.

17) Resources

1. Django,
2. Django Rest Framework
3. Django Channels
4. Redis
5. Next.js
6. React.js
7. TypeScript
8. Python 3
9. Python virtual environment
10. Redis quickstart
11. Simple JWT
12. Reactstrap
13. Bootstrap
14. Git
15. Github
16. Signals
17. Celery
18. Docker
19. AWS -EC2