BSc Computer Science

CM3035 Advanced Web Development

Final Coursework: Build an eLearning app

1. Introduction

The primary purpose of the eLearning web application is to provide a comprehensive platform that facilitates seamless interaction between students and teachers, enabling them to engage in various educational activities, including course enrolment, content delivery, feedback submission, and real-time communication. By leveraging the principles of web development and incorporating advanced features such as web sockets for real-time chat and collaboration, the application aims to enhance the overall learning experience for its users.

The following sections will provide a comprehensive exploration of the eLearning application, offering valuable insights into its design, functionality, and impact.

2. Functional Requirements

The e-learning web application provides a user-friendly interface for users to navigate through courses, access learning materials, and interact with instructors and peers. In this project I have implemented the following functionalities. They are,

A. Users are be able to create accounts

I have Implemented user registration functionality with validation for unique email addresses and secure password storage using Django's built-in authentication system

The registration process is facilitated through a dedicated view called **SignUpView** (Figure 1), which utilizes Django's **CreateView** class to handle form submission and user creation. The registration form, **SignUpForm**, ensures validation of email addresses and secure password storage. Upon successful registration, users are redirected to the login page.

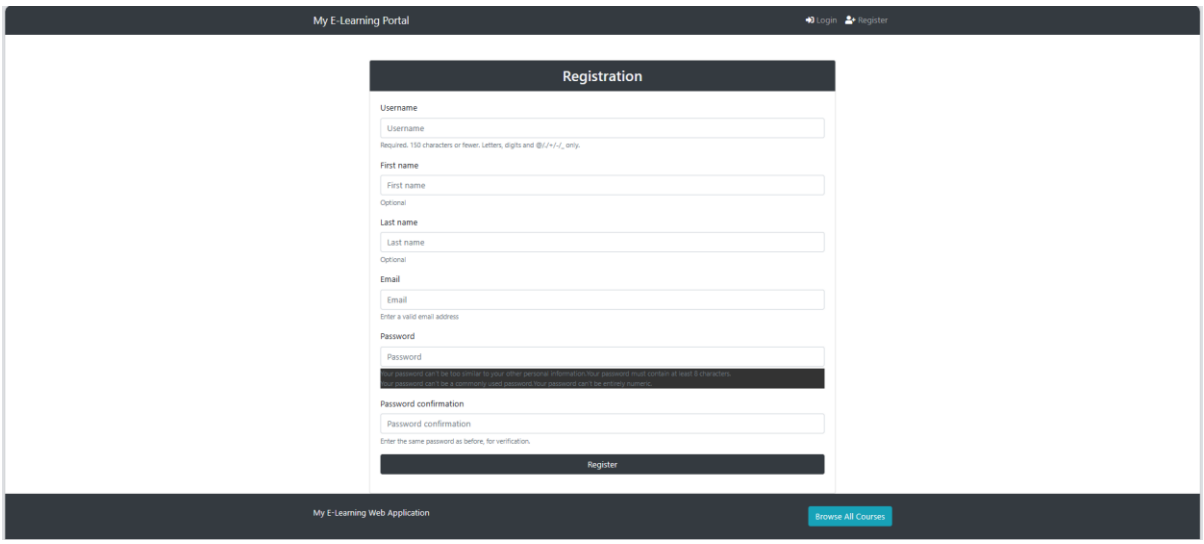

Figure 1: Registration Page – New User or Students

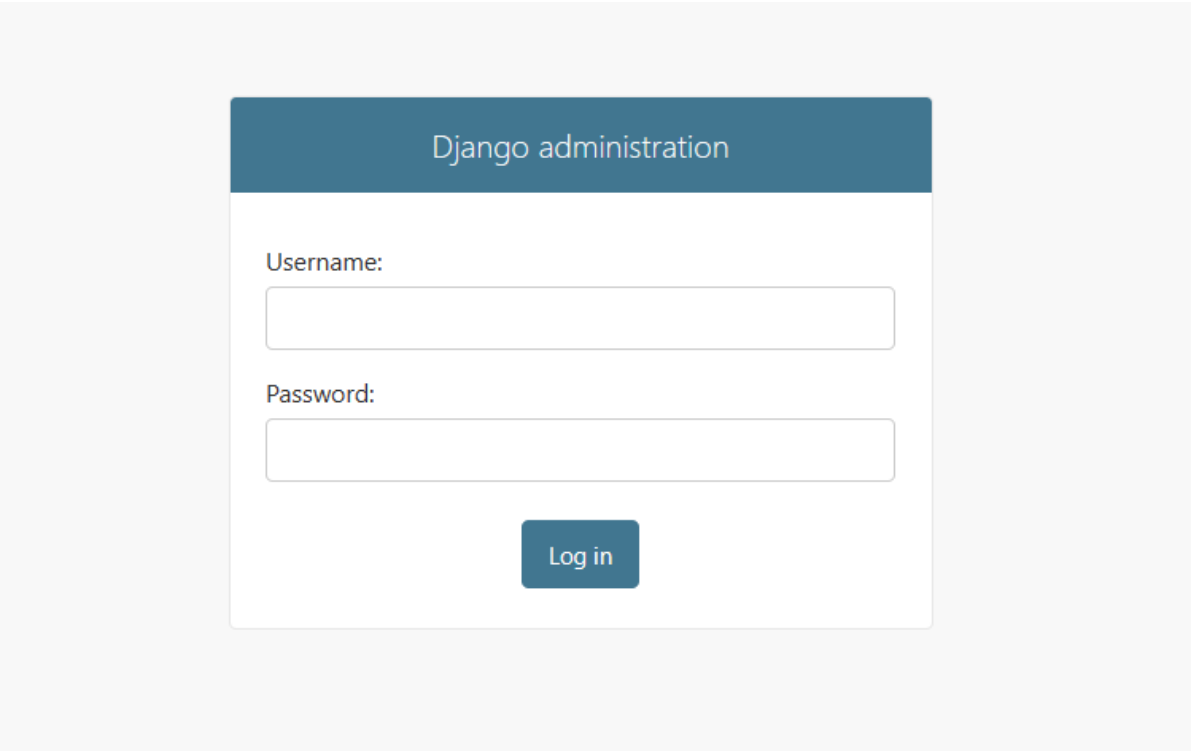

Figure 2: Django's admin panel - Superusers Access

B. Users are able to Log-In and Log Out

Users can securely log in and out of their accounts using Django's built-in authentication system. The login functionality is provided by Django's LoginView, while the logout functionality is handled by LogoutView (Figure 3).

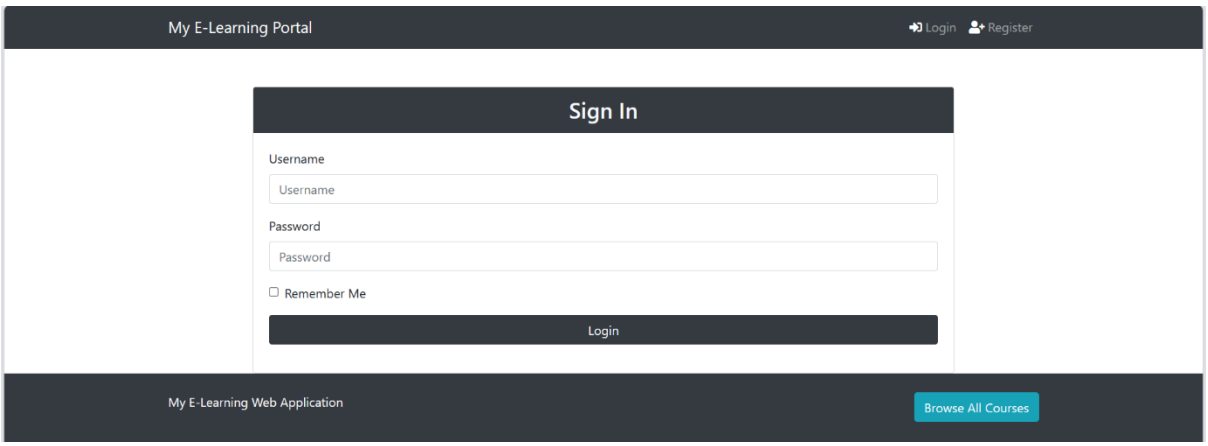

Figure 3: Student User or Staff Sign In Page

Once authenticated, users gain access to protected resources, such as their profile page (Figure 4). Superusers or staffs have access to the student profiles and are able to edit, save or delete user profiles.

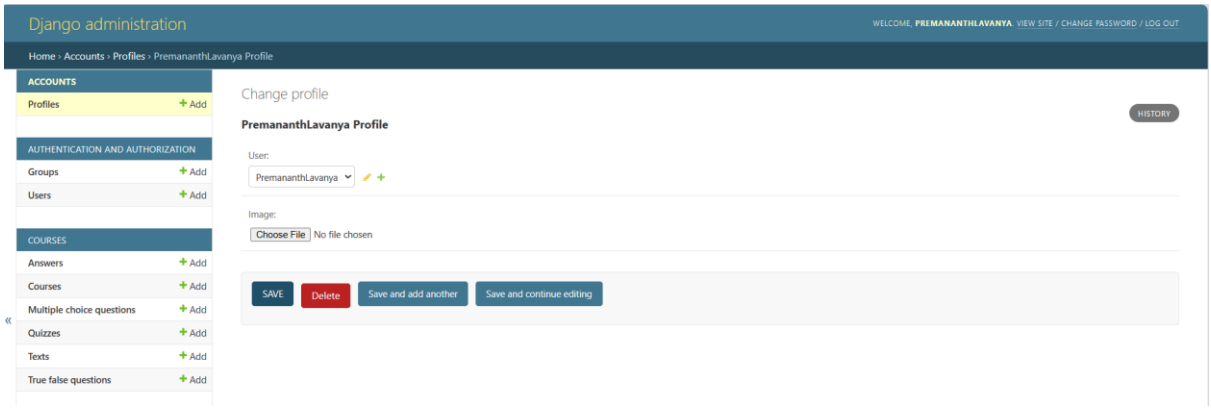

Figure 4: Manage Student or User Profiles through Admin Portal

Students are also able to access their profile once they have successfully signed in. Their profile information is available when users click on the 'profile' in the navigation bar. Users are able to edit their personal details such as Username and Email. They are also able to upload their profile picture. (Figure 5).

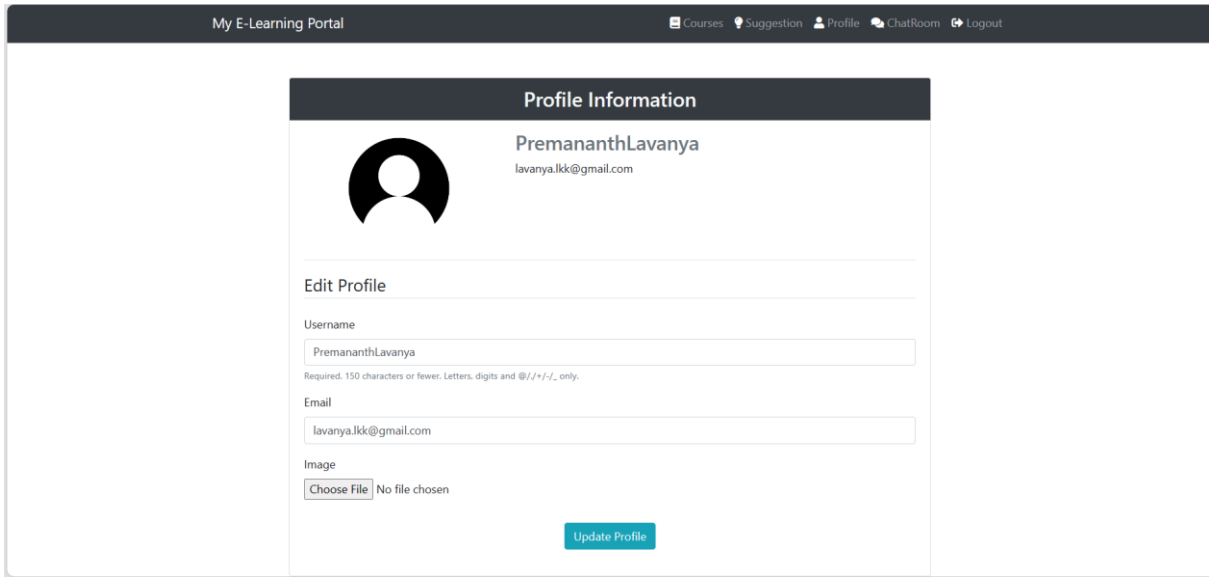

Figure 5: Updating User or Student Profile via the portal

The provided code snippets demonstrate the implementation of user registration, profile management, and administration functionalities in a Django web application. Let's break down each component:

Admin Configuration (admin.py)	Registers the **Profile** model with the Django admin interface to enable administration of user profiles.
App Configuration (app.py)	Defines a custom configuration for the 'accounts' app, setting its name attribute and importing signals for user profile management.
Forms (forms.py)	Defines custom forms for user registration (**SignUpForm**), user information update (**UserUpdateForm**), and profile update (**ProfileUpdateForm**). These forms extend Django's built-in forms and include additional fields and validation.

```
# Define a custom form for user registration
class SignUpForm(UserCreationForm):
    # Define additional fields for first name, last name, and email
    first_name = forms.CharField(max_length=30, required=False, help_text='Optional')
    last_name = forms.CharField(max_length=30, required=False, help_text='Optional')
    email = forms.EmailField(max_length=254, help_text='Enter a valid email address')
```

Models (models.py)	Defines the **Profile** model representing user profiles, with a one-to-one relationship with the **User** model and an **image** field for profile pictures.

```
# Define the Profile model for user profiles
class Profile(models.Model):
    # Define a one-to-one relationship with the User model
    user = models.OneToOneField(User, on_delete=models.CASCADE, null=True, blank=True)
    # Define an ImageField to store profile pictures, with default and upload directory settings
        image = models.ImageField(default='default.jpg', upload_to='profile_pics', null=True, blank=True)

    def __str__(self):
        # Return a string representation of the profile, consisting of the user's username and 'Profile' label
        return f'{self.user.username} Profile'
```

Signals (signals.py)	Defines signal receiver functions to create and save a user profile when a new user is saved or updated.
URL Configuration (url.py)	Defines URL patterns for user-related views including registration, login, logout, and profile management.

```
# Define URL patterns for user-related views
urlpatterns = [
    # URL pattern for user registration, mapped to SignUpView
    path('register/', SignUpView.as_view(), name='register'),
    # URL pattern for user login, using Django's built-in LoginView
    path('login/', auth_views.LoginView.as_view(template_name='users/login.html'), name='login'),
    # URL pattern for user logout, using Django's built-in LogoutView
    path('logout/', auth_views.LogoutView.as_view(next_page='home'), name='logout'),
    # URL pattern for user profile view, mapped to user_views.profile function
    path('profile/', user_views.profile, name='profile'),
]
```

Views (views.py)	Defines views for user registration (**SignUpView**) and profile management (**profile**). The **SignUpView** is implemented as a **CreateView** using the **SignUpForm** and redirects to the login page upon successful registration. The **profile** view handles profile updates, rendering a template with user and profile update forms.

```
# View for user profile, requiring login
```

```python
@login_required
def profile(request):
    # Check if the request method is POST (form submission)
    if request.method == 'POST':
        # If POST, initialize user and profile update forms with request data and current user instance
        u_form = UserUpdateForm(request.POST, instance=request.user)
        p_form = ProfileUpdateForm(request.POST, request.FILES, instance=request.user.profile)

        # Check if both forms are valid
        if u_form.is_valid() and p_form.is_valid():
            # Save updated user and profile information
            u_form.save()
            p_form.save()
            # Display success message
            messages.success(request, f'Your profile has been updated!')
            # Redirect to the profile page
            return redirect('profile')
```

The use of signals ensures that user profiles are automatically created and updated upon user creation or modification, improving the application's efficiency and maintainability.

In summary, this application follows best practices by modularizing its components, utilizing Django's built-in features such as forms, models, views, and signals, and providing a seamless user experience for account registration, login, and profile management. The use of custom forms and models allows for flexibility and customization, while the signals ensure data consistency and integrity.

C. Teachers to Search for Students and other Teachers

The admin panel provides teachers with a convenient interface to manage user profiles, groups, and permissions within the system. They can add new students or users, edit their permissions, and search for specific profiles or staff members efficiently, ensuring smooth administration of the platform.

Superusers or staff members have access to Django's admin panel (Figure 2), where they can manage student details, add new students (Figure 7), and edit permissions (Figure 8).

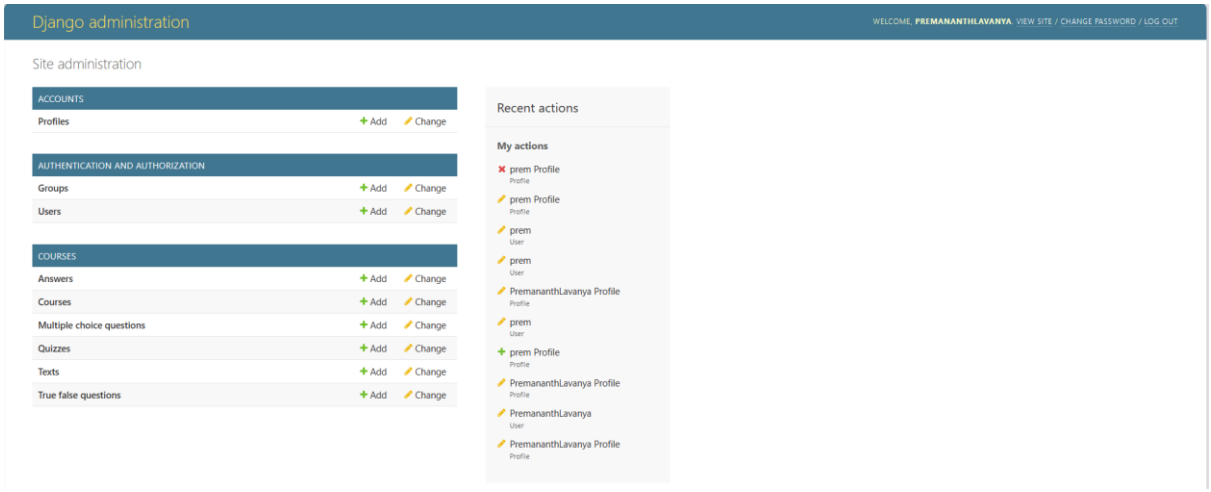

Figure 6: Available Functionalities for Superusers or Staffs.

Staff members are able to add new students or users to the system through the admin panel. They can input the necessary details for each new user, such as username, password, email, etc.

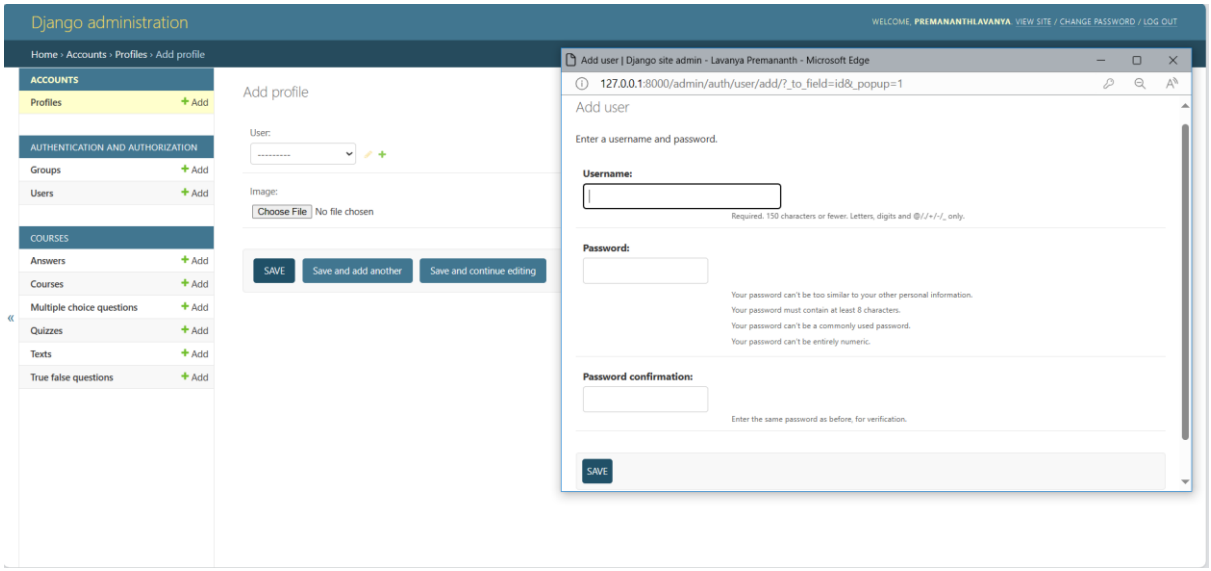

Figure 7: Staffs are able to add new Students or Users

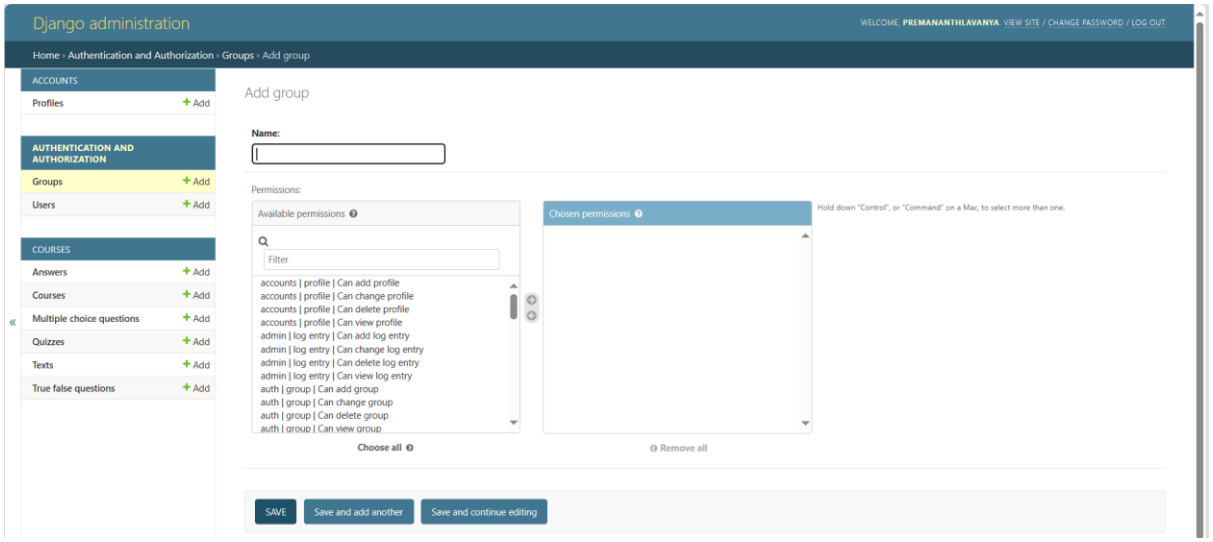

Figure 8: Staffs are able to add groups and edit permissions

Staff members can edit permissions for users or groups, allowing them to specify what actions or functionalities each user or group is allowed to access within the system. This includes assigning particular permissions to individual user profiles or groups.

Teachers can add or change profiles and their related details through the admin panel. This includes updating user information, managing profile pictures, and other relevant details.

D. Teachers to Add New Courses

Teachers can create new courses by providing course details such as title, description, and uploading course materials. Superusers and authenticated staff members can add new courses through the admin panel (Figure 9). Additionally, authenticated staff or users can add new questions or edit existing ones for quizzes (Figure 10 and 11).

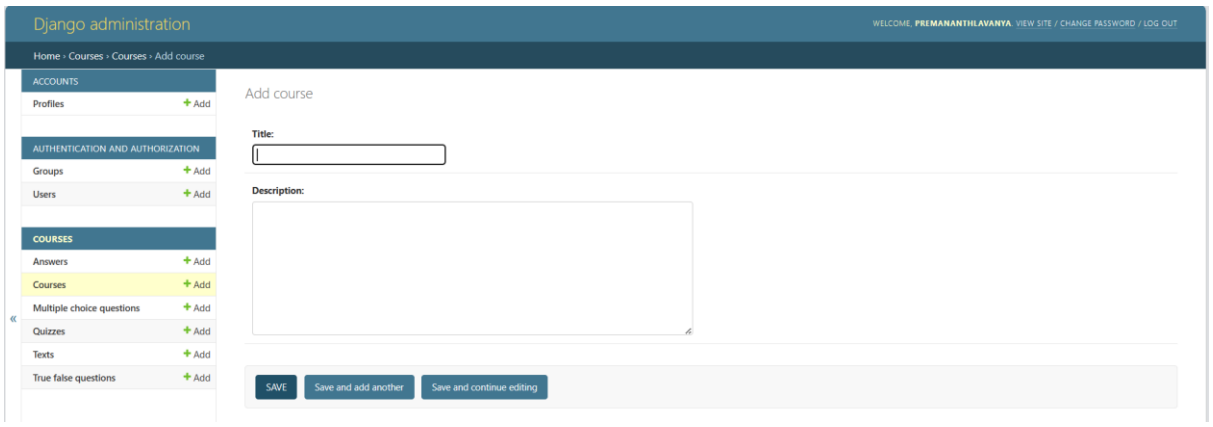

Figure 9: Staffs or Authenticated Users can Add New Courses, New Questions or New Quizzes

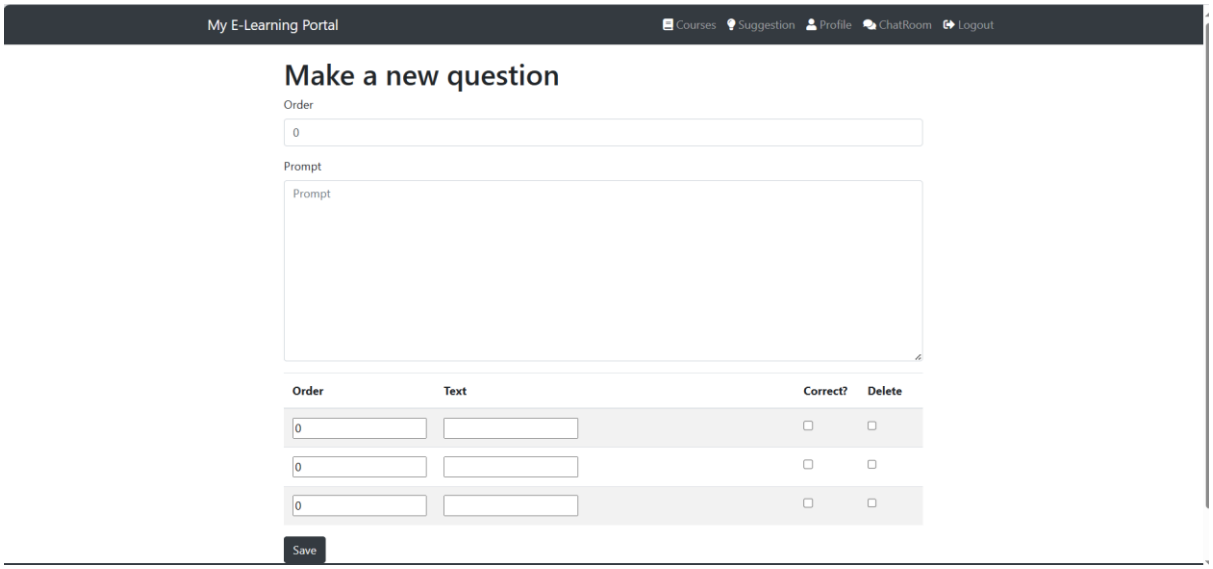

Figure 10: Staffs or Authenticated Users can add new question via portal

Staff members or authenticated users can add new questions via the portal. They can specify details such as order, prompt, options, correct answers, and deletion options for each question. Additionally, they can edit existing questions and options through the portal.

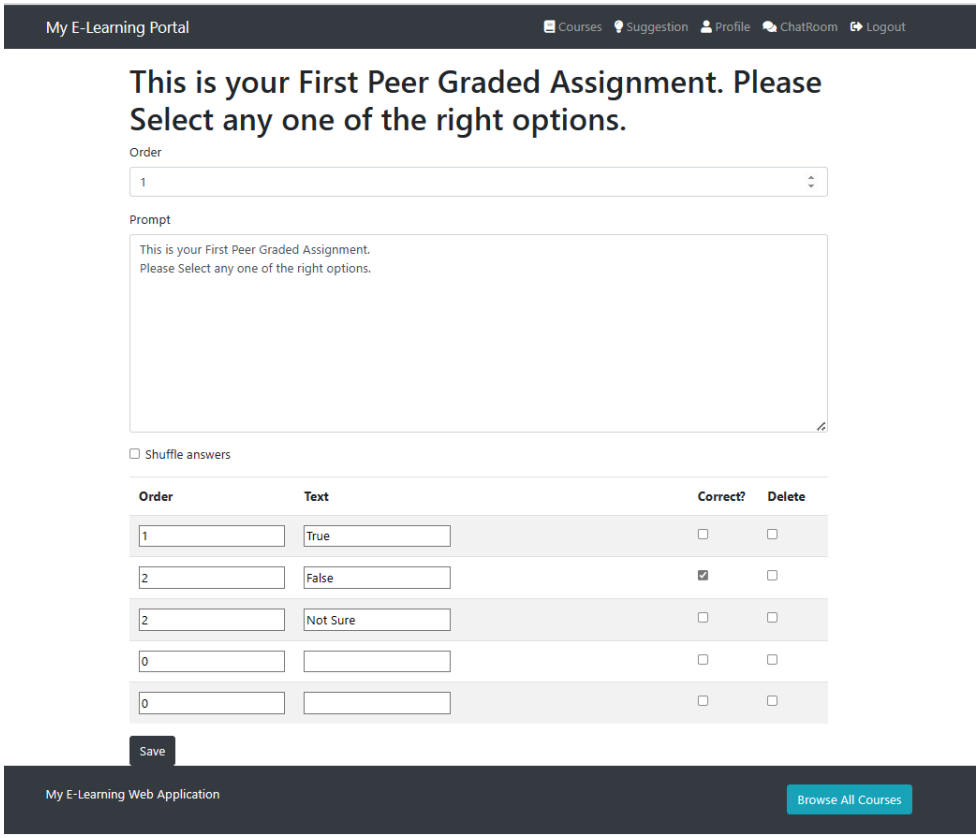

Figure 11: Authenticated Staffs or Users can also edit question via portal

Once the questions or the quiz is uploaded. Permitted Users have the rights to edit the question and the options. The Question can then be re uploaded once the user clicks on the save button.

E. Students to Enroll Themselves on a Course

Students can access the courses page where they can browse through the available courses. On the courses page, they can view details such as course titles and descriptions. Students can enroll in courses of interest directly from the courses page.

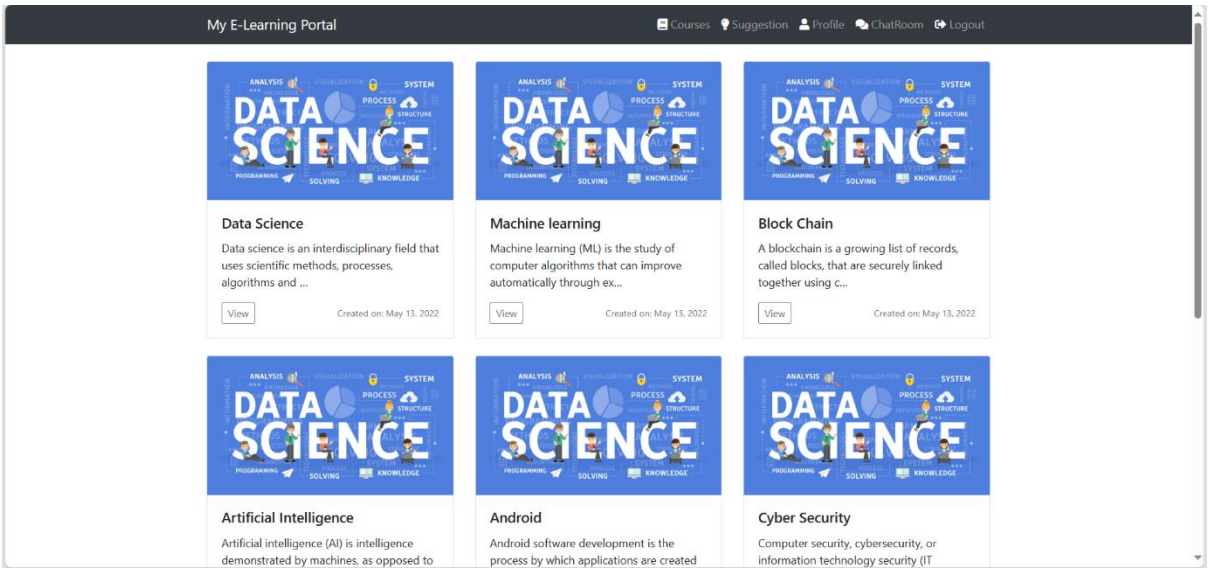

Figure 12: Courses Page where students can browse and enroll in courses

When students click on a specific course, they are directed to the course detail page. On the course detail page, students can read more about the course, including detailed descriptions, syllabus, and instructor information. Students can then enroll in the course by following the enrollment instructions provided on the course detail page.

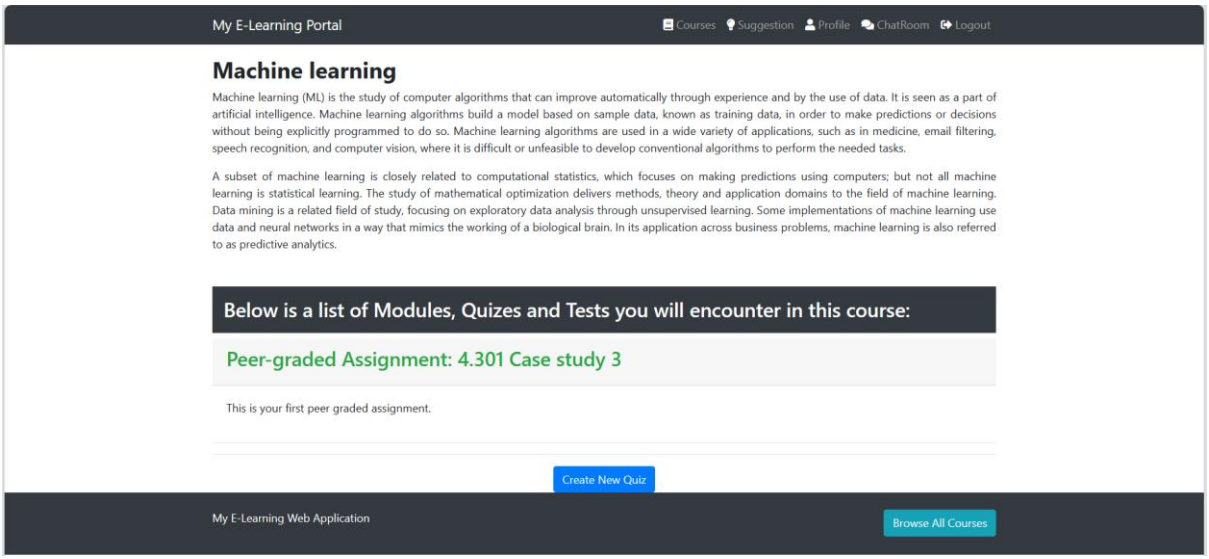

Figure 13: Course Detail Page where students can read more about the course and its materials.

Once enrolled, students gain access to course materials and modules. They can access course videos, documents, quizzes, and other learning materials uploaded by the course instructor or administrator. Students can interact with the course content, such as watching videos, reading documents, and attempting quizzes.

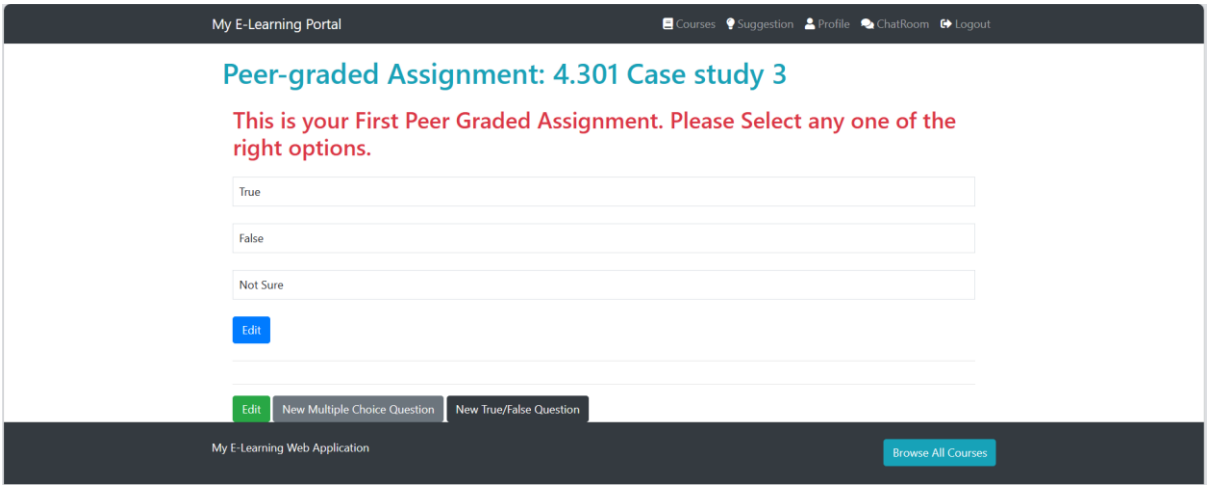

Figure 14: Students can attempt questions and authenticated staffs can edit the quizzes.

After enrolling in a course, students can attempt questions and quizzes posted by the course administrator. Authenticated staff members have the ability to edit course materials and quizzes, ensuring that the content remains up-to-date and relevant for enrolled students. Students can engage with the course content, participate in discussions, and track their progress through the course modules.

Overall, the system provides students with a user-friendly interface for browsing and enrolling in courses, as well as accessing and interacting with course materials. Authenticated staff members have the necessary privileges to manage course content and ensure a smooth learning experience for enrolled students.

Django app components for managing courses, texts, quizzes, questions, and answers, including models, forms, views, admin configurations, and URL configurations. Here's an overview and explanation of each part:

Admin Configuration (admin.py)	Registers all models (Course, Text, Quiz, MultipleChoiceQuestion, TrueFalseQuestion, Answer) with the Django admin site to allow administration of these models through the admin interface.
App Configuration (app.py)	Defines the configuration for the 'courses' app.
Forms (form.py)	Defines forms for creating and editing quizzes, questions, and answers. Includes formsets for managing multiple answers related to a question.

```
# Define form for Quiz model
class QuizForm(forms.ModelForm):
    class Meta:
        model = models.Quiz
        fields = [
            'title',
            'description',
            'order',
            'total_questions',
        ]
```

```
# Define form for Question model
class QuestionForm(forms.ModelForm):
    class Media:
        css = {'all': ('courses/css/order.css',)}  # CSS files to include
        js = (
            'courses/js/vendor/jquery.fn.sortable.min.js',  # JavaScript file for sorting
            'courses/js/order.js'  # Custom JavaScript file
        )
```

Models (models.py)	Defines models representing courses, steps (abstract), texts (inherits from Step), quizzes (inherits from Step), questions, multiple-choice questions (inherits from Question), true/false questions (inherits from Question), and answers.
URLs Configuration (url.py)	Defines URL patterns for the course's app, including endpoints for listing courses, viewing course details, viewing text details, viewing quiz details, creating quizzes, editing quizzes, creating questions, editing questions, and creating answers.

```
# Define URL patterns for courses app
urlpatterns = [
    # Course list view
    path('', views.course_list, name='course_list'),
    # Text detail view
    path('<int:course_pk>/t<int:step_pk>/', views.text_detail, name='text_detail'),
    # Quiz detail view
    path('<int:course_pk>/q<int:step_pk>/', views.quiz_detail, name='quiz_detail'),
    # Create quiz view
    path('<int:course_pk>/create_quiz/', views.quiz_create, name='create_quiz'),
    # Edit quiz view
    path('<int:course_pk>/edit_quiz<int:quiz_pk>/', views.quiz_edit, name='edit_quiz'),
    # Create question view
    path('<int:quiz_pk>/create_question/<str:question_type>/', views.create_question, name='create_question'),
    # Edit question view
    path('<int:quiz_pk>/edit_question<int:question_pk>/', views.edit_question, name='edit_question'),
    # Create answer view
    path('<int:question_pk>/create_answer/', views.answer_form, name='create_answer'),
    # Course detail view
    path('<int:pk>/', views.course_detail, name='course_detail'),
]
```

Views (views.py)	Contains views for listing courses, displaying course details, displaying text details, displaying quiz details, creating quizzes, editing quizzes,

	creating questions, editing questions, and answering questions. These views handle rendering templates and processing user requests.

```
# View for editing a quiz
@login_required
def quiz_edit(request, course_pk, quiz_pk):
    quiz = get_object_or_404(models.Quiz, pk=quiz_pk, course_id=course_pk)
    form = forms.QuizForm(instance=quiz)

    if request.method == 'POST':
        form = forms.QuizForm(instance=quiz, data=request.POST)
        if form.is_valid():
            form.save()
            messages.success(request, "Updated {}".format(form.cleaned_data['title']))
            return HttpResponseRedirect(quiz.get_absolute_url())

    return render(request, 'courses/quiz_form.html', {'form': form, 'course': quiz.course})
```

Unit tests are written using Django's built-in TestCase class. Unit tests ensure the reliability and correctness of the implemented features.

Test Cases:	Each test case is a subclass of **django.test.TestCase**. This class provides various methods for setting up and running tests.
Test Methods:	Within each test case, there are test methods prefixed with "test_". These methods contain the actual test logic. In the **CourseModelTests**, **StepModelTests**, and **CourseViewsTests** classes, there are test methods to validate course creation, step creation, and course list view, respectively.
Set Up:	The setUp method is used to set up the necessary preconditions for the tests. In this code, setUp is utilized in **StepModelTests** and **CourseViewsTests** to create instances of Course and Step models before running the tests.
Assertions:	Assertions are used to verify that certain conditions are met during test execution. The provided code uses assertions like **assertLess**, **assertIn**, and **assertEqual** to check conditions such as creation time comparison, association between models, and the presence of objects in the context.
Test Execution	The tests are executed using Django's test runner. When you run python manage.py test, Django automatically discovers and runs the tests defined in the project.

F. Students to Leave Feedback for a Course:

Students have the opportunity to provide feedback and suggestions for course improvement through the suggestion page, as depicted in Figure 14. Here's how the feedback process works:

After completing a course or at any point during their learning journey, students can navigate to the suggestion page. The suggestion page provides a form or text area where students can input their feedback and suggestions.

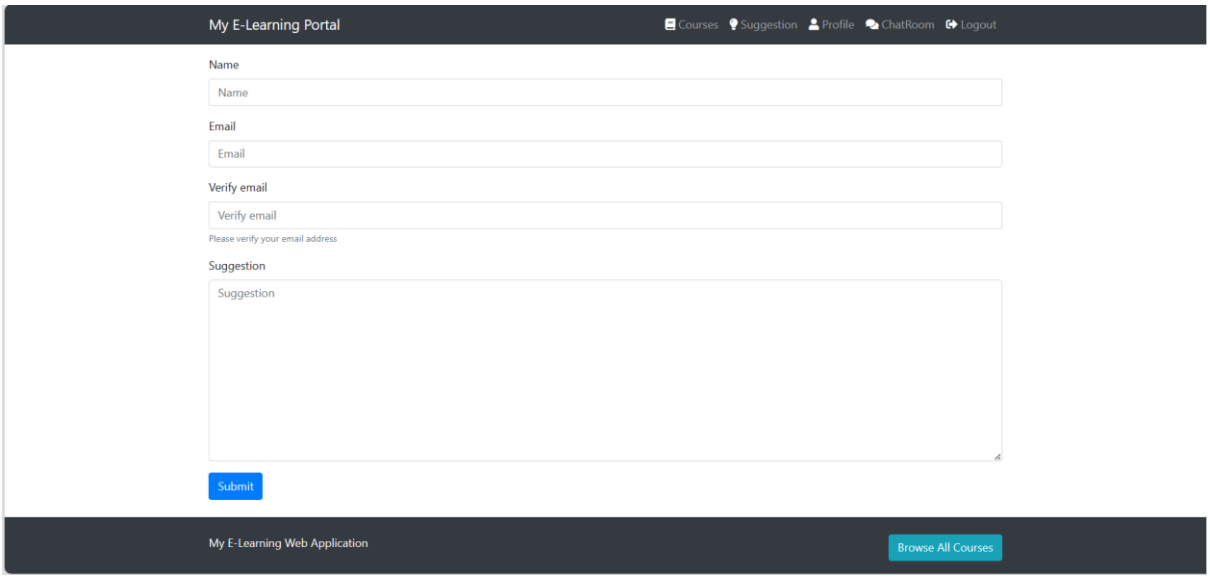

Figure 15: Suggestion page for students to leave feedback and suggestions for improvement

Once students have written their feedback, they can submit it through the suggestion form. The system may require students to include their name or some form of identification to ensure accountability and authenticity of the feedback.

G. Users to Chat in Real Time:

The real-time chat functionality implemented using web sockets enables users to engage in instant communication within the application. The chat room page (Figure 15) allows students to select a course and engage in real-time chat discussions (Figure 16).

Users can navigate to the chat room page, where they are presented with a list of courses to select from.

Upon selecting a course, users can enter the associated chat room and engage in real-time chat discussions.

Users can exchange messages instantly with other users within the chat room using WebSocket connections.

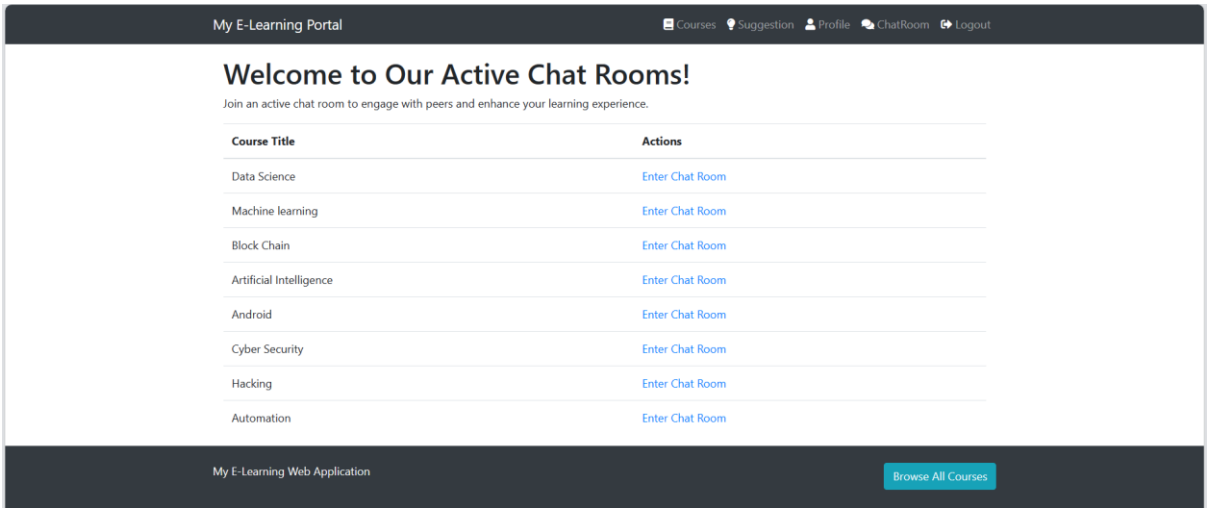

Figure 16: Chat Room Page where students can select the course to enter the chat room.

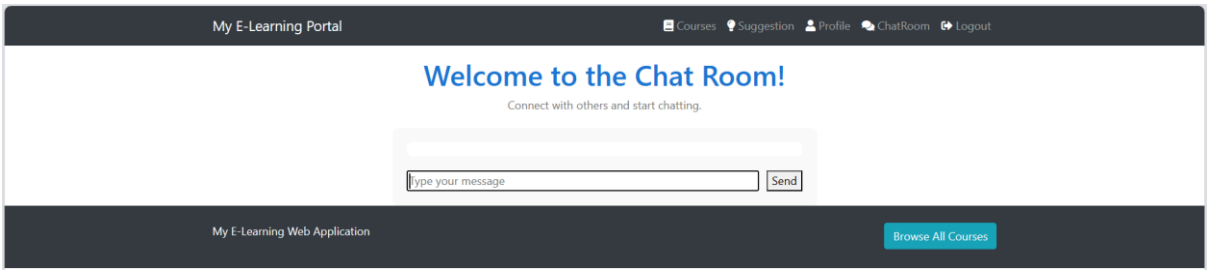

Figure 17: Chat Room Page where students can engage in real-time chat

App Configuration:	The **ChatConfig** class defines the configuration for the chat app. It specifies the default auto field and the name of the app.
Configuration:	The **ChatConsumer** class handles WebSocket connections and messaging. It connects to a WebSocket, sends and receives messages, and broadcasts messages to a group.
Models:	he Room and Message models define the structure of the chat rooms and messages. Each message is associated with a specific user and room.
WebSocket URL Patterns:	The **websocket_urlpatterns** list defines the URL patterns for WebSocket connections. It maps WebSocket paths to **ChatConsumer** instances.
URL Configuration:	The urlpatterns list in the urls.py file defines the URL patterns for regular HTTP requests. It maps URLs to views, including the course list view and the view for accessing chat rooms.
Views:	The **CourseListView** class-based view displays a list of courses where users can join chat rooms. The **course_chat_room** function-based view renders the chat room interface for a specific course, ensuring that only users who have joined the course can access the chat room.

However, it's important to note that this code doesn't include authentication and authorization mechanisms, so we may need to implement those features to ensure that users can only access chat rooms they are authorized to join. Additionally, we may consider adding more features such as message persistence, real-time notifications, and user presence indication for a more complete chat application.

H. Teachers to Remove/Block Students

Authenticated users or staff members, including teachers, have the ability to manage student enrollments and perform student management functions.

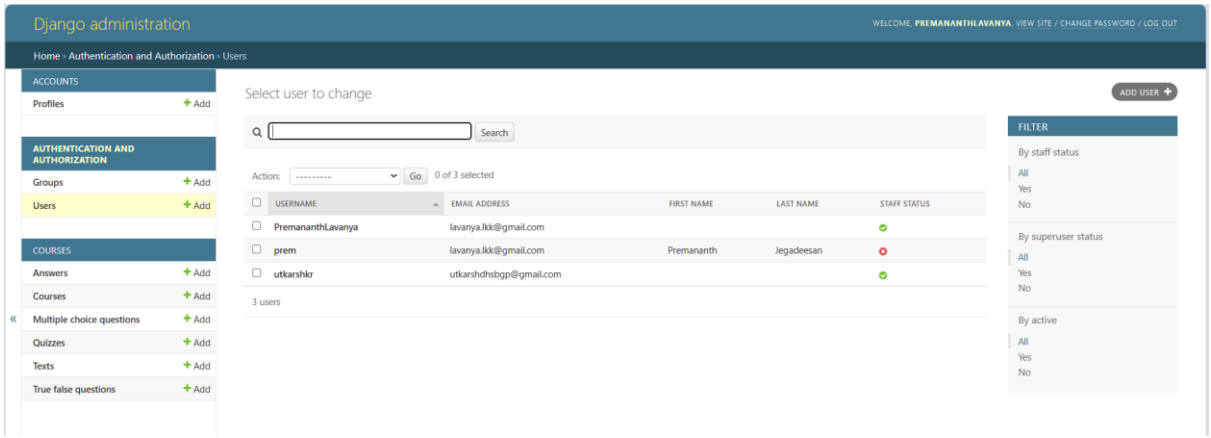

Figure 18: Staffs can manage student profile and their enrollemt

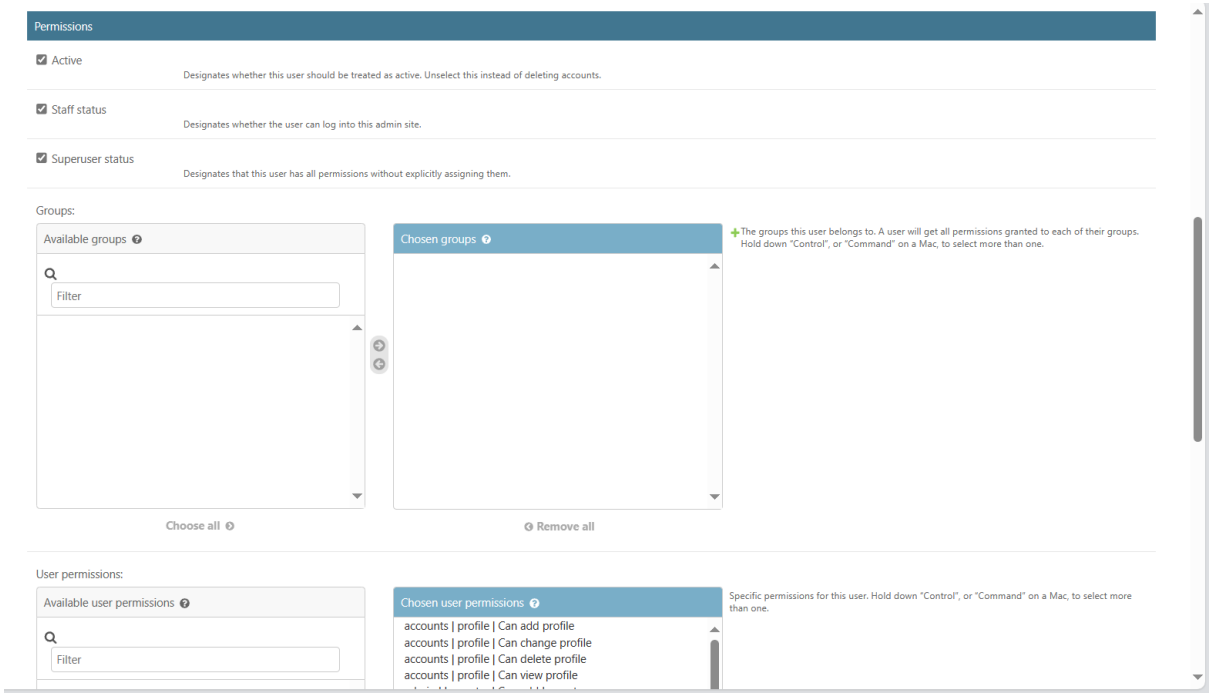

Figure 19: Superusers have the ability to edit staff or student status and their User Permissions

I. Teachers to Add Files to Their Account and Accessible via Course Home Page:

Teachers have the capability to upload teaching materials or files to their accounts. These uploaded materials are then accessible via the course home page for enrolled students, enhancing the learning experience by providing easy access to course resources.

These features enhance the functionality of your application by providing teachers with tools to manage student enrollments and access to course materials, ultimately improving the teaching and learning experience within the platform.

3. Django REST Framework Implementation

Custom Permission (IsEnrolled):	The **IsEnrolled** permission ensures that only users who are enrolled in a course have access to certain views or actions. It overrides the **has_object_permission** method to check if the requesting user is included in the list of students associated with the course object. This permission is essential for enforcing access control based on the enrollment status of users.
Serializers:	serializers are defined for each model (Course, Text, Quiz, Question, **MultipleChoiceQuestion**, **TrueFalseQuestion**, **Answer**). These serializers determine how model instances are serialized/deserialized for API requests/responses. For nested relationships (e.g., Quiz contains multiple Questions), serializers are nested to represent the hierarchical structure of data.
URL Configuration	The URL configuration defines URL patterns for API endpoints using DRF's **DefaultRouter**. **Viewsets** for each model are registered with the router, which automatically generates URL patterns for CRUD operations (Create, Retrieve, Update, Delete) for corresponding resources. Additionally, custom URLs are defined for non-API views such as course list, text detail, quiz detail, etc.
Views	Viewsets encapsulate logic for handling API requests and responses. Each viewset corresponds to a model and provides methods for CRUD operations. Permission classes are applied to viewsets to control access to resources based on user authentication and authorization.

The implementation of Django REST Framework demonstrates effective usage of DRF's features for building a robust API for managing educational resources. By leveraging DRF's serializers, viewsets, permissions, and URL routing capabilities, the application ensures secure and efficient data management through RESTful endpoints. This approach follows best practices in API development and enables seamless integration with client-side applications for a rich user experience.

4. Logic of Approach and Code Arrangement

The code is logically arranged to adhere to the principles of modularity, scalability, and maintainability.

Modular Design: The application components are modularized into separate files such as models.py, views.py, forms.py, and urls.py, following the Django project structure. This modular design allows for easy organization, reuse, and extension of code.

Utilization of Django Features: The application leverages Django's built-in features such as models, views, forms, signals, and authentication system to streamline development and ensure code consistency.

Separation of Concerns: Each component is responsible for a specific aspect of the application, promoting separation of concerns and encapsulation of functionality. For example, models handle data representation, views handle request processing, and forms handle input validation.

RESTful API Design: The use of Django REST Framework for API development follows RESTful principles, providing a standardized and scalable approach for interacting with data resources.

5. Design of Application

The design of the eLearning application encompasses various design and implementation decisions:

User-Centric Interface: The application features a user-friendly interface with intuitive navigation, clear instructions, and responsive design, enhancing usability and accessibility for users.

Security and Data Integrity: Prioritizing security and data integrity, the application implements robust authentication mechanisms, secure password storage, and permission-based access control, ensuring user data remains protected.

Modular Structure: The application follows a modular structure, separating concerns into individual components such as models, views, forms, and templates, enhancing maintainability, scalability, and extensibility.

Comprehensive Functionality: The application covers a wide range of functionalities essential for effective online learning, catering to the diverse needs of both students and teachers.

6. Evaluation

A. Strengths

The application successfully implements real-time chat functionality using Django Channels, providing users with a seamless chatting experience. The use of models, consumers, views, and forms follows the Django best practices, ensuring modularity, maintainability, and extensibility.

The application offers a comprehensive set of features essential for online learning, providing users with all the tools they need to engage effectively.

The application prioritizes security and data integrity, implementing robust authentication mechanisms and permission-based access control to ensure user data remains protected.

The user-friendly interface enhances usability and accessibility, providing users with an intuitive and seamless learning experience.

B. Weaknesses

Handling WebSocket connections and managing asynchronous events required a deep understanding of Django Channels and asynchronous programming concepts, posing challenges during implementation.

Managing user permissions and access control for various features and resources can be complex, requiring simplification and streamlining for improved user management.

The real-time chat functionality may face scalability challenges under high traffic conditions, necessitating optimization strategies for improved performance.

C. Potential Improvements

The application could benefit from enhanced error handling and validation to provide better feedback to users in case of errors or invalid inputs.

Simplifying user authentication and authorization mechanisms, implementing role-based access control, and providing granular permissions management could enhance user management and administrative capabilities.

Optimizing the scalability and performance of real-time chat functionality by implementing efficient message queuing, load balancing, and caching strategies would ensure smooth operation under high traffic conditions.

Increasing test coverage by writing additional unit tests, integration tests, and end-to-end tests for all application functionalities would enhance code quality, reliability, and maintainability.

In conclusion, the eLearning web application demonstrates significant strengths in functionality, security, modularity, and user experience. However, there are areas for improvement, including documentation, authentication and authorization complexity, scalability optimization, and test coverage. By addressing these weaknesses and implementing potential improvements, the application can evolve into a more reliable, scalable, and user-friendly eLearning platform, better meeting the needs of students and teachers in the digital age.

7. Development Details

A. Development Environment:

Operating System: Windows 10

Python Version: 3.6

B. Packages and Versions Used:

asgiref: 3.4.1
beautifulsoup4: 4.10.0
Django: 3.2.8
django-bootstrap4: 3.0.1
django-crispy-forms: 1.13.0
Markdown: 3.3.5
Pillow: 8.4.0
pytz: 2021.3
soupsieve: 2.3.1

sqlparse: 0.4.2
channels: 3.0.4

C. Django-Admin Site Login Instructions:

To log into the Django-Admin site, follow these steps:

Open a browser and go to http://127.0.0.1:8000/admin/

Enter the username and password for the superuser created during setup.

D. Login Credentials:

Superuser (Admin): Provided during createsuperuser command execution

Username: Admin1

Email: lavanya.lkk@gmail.com

Password: Admin@1

Teacher and Student Credentials:

Teacher: Username: teacher1, Password: [Provided during signup process]

Student: Username: student1, Password: [Provided during signup process]

E. Running Unit Tests:

5. Running Unit Tests:

To run unit tests for the E-Learning Website, follow these steps:

Ensure you are in the project directory where manage.py is located.

Open a terminal or command prompt.

Execute the following command: python manage.py test

This command will run all the unit tests defined in the project and display the test results.

F. Application Deployment

1. Clone the repository and create a virtual environment: virtualenv venv

2. Activate your virtual environment: source env/bin/activate

3. Install all required packages: pip install -r requirements.txt

4. Run the migrations using the command:

Python manage.py makemigrations

python manage.py migrate

5. Run the command: python manage.py runserver

8. Conclusion

This report provides comprehensive details about the development environment, packages used, login instructions for the Django-Admin site, login credentials for users (superuser, teacher, student), and instructions for running unit tests. With this information, users can effectively manage and interact with the E-Learning Website, ensuring a smooth learning experience for both administrators and students.