Title: “E-Learning Platform Development with Django and Docker”
Author’s Name: Abylay Aitbanov
Date of Submission: 22.12.2024
Course Details: Web app development (MSc)

Executive Summary

The goal of this project is to develop a robust e-learning platform using Django, containerized with Docker. This platform integrates fundamental Django concepts, including models, views, templates, and the Django Rest Framework (DRF), while leveraging Docker for containerization and deployment. The project aims to provide a scalable, secure, and user-friendly environment for managing courses, lessons, quizzes, and user progress.

The implementation focuses on creating a structured backend powered by Django, with APIs exposed via DRF to support seamless frontend interaction. Docker is utilized to containerize the application, ensuring consistent deployment across various environments and simplifying dependency management. The platform currently includes core features like user management, course enrollment, quiz handling, and user progress tracking, with plans for future enhancements.

Key outcomes of the project highlight the effective combination of Django and Docker for streamlined development and deployment. DRF ensures a flexible API architecture, while Docker facilitates portability and scalability. These technologies together establish a solid foundation for expanding the platform's capabilities.

My recommendations include transitioning to modern frontend frameworks like React or Vue.js for a richer user interface, integrating advanced features such as real-time chat or video streaming, and deploying the platform on cloud services like AWS or Azure for greater scalability. This project demonstrates the potential of integrating Django and Docker to build modern, efficient e-learning platforms.

Table of Contents

Introduction

Background

This project focuses on creating an e-learning platform using Django and Docker. Django is a Python-based web framework that simplifies the development of secure and scalable web applications. It provides built-in tools for managing databases, creating APIs, and handling user authentication. Docker is a platform for containerization, allowing applications to run consistently across different environments by packaging them with all their dependencies.

Project Goals

The goal of this project is to develop a robust e-learning platform using Django, containerized with Docker. The project will encompass fundamental Django concepts, including models, views, templates, and the Django Rest Framework (DRF), as well as Docker for containerization and deployment.

Scope

The scope of this project is to develop an e-learning platform using Django and Docker. The platform will enable users to register, log in, and manage their profiles, with different user roles such as students and instructors, each having distinct permissions. Instructors will be able to create and manage courses, including adding lessons, quizzes, and other learning materials. Students will be able to view and enroll in these courses, track their progress, and resume lessons or quizzes at any time.

The system will include a quiz and assessment module to evaluate students' understanding of course material, with automatic grading and progress tracking. The application will be fully containerized using Docker to ensure smooth deployment and consistency across various environments, including development, testing, and production.

System Architecture

The system architecture of the e-learning platform is built to ensure scalability, flexibility, and ease of deployment, leveraging the Django framework for backend development and Docker for containerization. The platform is structured around several key components that work together to provide a seamless experience for the users.

At the core of the architecture is the backend, developed using Django. Django handles all the business logic, including managing users, courses, lessons, quizzes, and progress tracking. It interacts with the database to store and retrieve data, ensuring that user interactions, such as enrolling in courses or submitting quiz answers, are processed correctly.

The database is a critical component of the system, storing all the necessary data for the platform. It includes user information, course details, lessons, quizzes, and progress tracking. For this platform, relational database like PostgreSQL is commonly used to manage the data in an organized and scalable manner.

Docker plays a central role in ensuring the system is easy to deploy and manage. The entire application, including the backend and database, is containerized using Docker. Docker containers isolate each component of the platform, ensuring that the platform runs consistently across different environments. Each component, such as the Django application and the database, runs in its own container, making it easy to develop, test, and deploy without worrying about conflicts between different software versions or system configurations.

Docker Compose is used to orchestrate and manage the interaction between these containers. It simplifies the process of running multiple containers together, allowing the backend and database containers to communicate seamlessly. By using Docker Compose, the platform can be set up quickly and efficiently, ensuring that all components are configured correctly and can work together as a unified system.

Overall, the system architecture is designed to be modular and scalable, ensuring that the platform can handle increased traffic and easily accommodate future feature enhancements. The use of Docker ensures that the platform is portable and can run consistently across different development, testing, and production environments.

Table Descriptions

The database schema of the e-learning platform consists of several tables that define the structure of the data for users, courses, lessons, quizzes, progress tracking, and other related components. Below are the descriptions of the primary tables in the schema, including their relationships and constraints.

User Table:

The User table stores information about users, including their credentials and role (student or instructor).

- **Fields**:
    - user_id: Primary key, auto-incremented integer.
    - username: A unique identifier for the user.
    - email: User’s email address (unique).
    - password_hash: Encrypted password for authentication.
    - is_student: Boolean field indicating if the user is a student.
    - is_instructor: Boolean field indicating if the user is an instructor.

- **Relationships**:
    - One-to-many relationship with Course (An instructor can create multiple courses).
    - One-to-many relationship with Enrollment (A user can enroll in multiple courses).
    - One-to-many relationship with Review (A user can leave reviews for courses).

Category Table:
The Category table stores information about different categories for courses (e.g., Programming, Data Science).

- **Fields**:
    - category_id: Primary key, auto-incremented integer.
    - name: Name of the category.
    - description: A text description of the category.
- **Relationships**:
    - One-to-many relationship with Course (A category can have multiple courses).

Course Table:
The Course table contains details about each course, including its title, description, price, and the instructor.

- **Fields**:
    - course_id: Primary key, auto-incremented integer.
    - title: The course title.
    - description: The course description.
    - price: The price of the course.
    - category: Foreign key to Category (A course belongs to one category).
    - created_at: Date and time when the course was created.
    - instructor: Foreign key to User (The instructor of the course).
- **Relationships**:
    - Many-to-one relationship with Category (A course belongs to one category).
    - Many-to-one relationship with User (Each course is taught by an instructor).
    - One-to-many relationship with Lesson (A course can have multiple lessons).
    - One-to-many relationship with Review (A course can have multiple reviews).
    - One-to-many relationship with Enrollment (A course can have multiple students enrolled).

Enrollment Table:
The Enrollment table tracks which users are enrolled in which courses and their enrollment status.

- **Fields**:
    - enrollment_id: Primary key, auto-incremented integer.
    - user: Foreign key to User (The user enrolled in the course).
    - course: Foreign key to Course (The course in which the user is enrolled).
    - enrollment_date: The date and time the user enrolled in the course.
    - status: The status of the enrollment (e.g., enrolled, completed).
- **Relationships**:
    - Many-to-one relationship with User (A user can enroll in many courses).
    - Many-to-one relationship with Course (A course can have many users enrolled).
- **Constraints**: The combination of user, course, and status must be unique.

Lesson Table:
The Lesson table stores individual lessons for each course, including lesson content and video URL.

- **Fields**:
    - lesson_id: Primary key, auto-incremented integer.
    - course: Foreign key to Course (Each lesson belongs to a course).
    - title: The title of the lesson.
    - content: The content of the lesson.
    - video_url: URL for the lesson video.
- **Relationships**:
    - Many-to-one relationship with Course (A lesson belongs to one course).

Review Table:
The Review table contains feedback provided by users for courses.

- **Fields**:
    - review_id: Primary key, auto-incremented integer.
    - course: Foreign key to Course (The course being reviewed).
    - user: Foreign key to User (The user who left the review).
    - rating: The rating given to the course (e.g., 1-5 stars).
    - comment: The text comment or feedback left by the user.

- created_at: The date and time the review was created.

- **Relationships**:
  - Many-to-one relationship with Course (A course can have many reviews).
  - Many-to-one relationship with User (A user can leave multiple reviews).

Payment Table:
The Payment table stores information about payments made by users for courses.

- **Fields**:
  - payment_id: Primary key, auto-incremented integer.
  - user: Foreign key to User (The user who made the payment).
  - course: Foreign key to Course (The course for which the payment was made).
  - amount: The amount paid.
  - payment_date: The date and time the payment was made.
  - status: The status of the payment (e.g., pending, completed).
- **Relationships**:
  - Many-to-one relationship with User (A user can make many payments).
  - Many-to-one relationship with Course (A course can have many payments).
- **Constraints**: The combination of user, course, and status must be unique.

Quiz Table:
The Quiz table contains quizzes associated with courses.

- **Fields**:
  - quiz_id: Primary key, auto-incremented integer.
  - course: Foreign key to Course (Each quiz is associated with a course).
  - title: The title of the quiz.
  - total_marks: The total number of marks for the quiz.
- **Relationships**:
  - Many-to-one relationship with Course (A quiz is associated with one course).
  - One-to-many relationship with QuizQuestion (A quiz contains multiple questions).

QuizQuestion Table:
The QuizQuestion table contains individual questions for quizzes.

- **Fields**:
  - question_id: Primary key, auto-incremented integer.

- quiz: Foreign key to Quiz (Each question belongs to a quiz).
- question_text: The text of the question.
- option_a, option_b, option_c, option_d: The answer options for the question.
- correct_option: The correct option (A, B, C, or D).

- **Relationships**:
  - Many-to-one relationship with Quiz (A question belongs to one quiz).

UserProgress Table:
The UserProgress table tracks a user's progress in each course, including completed lessons and quiz scores.

- **Fields**:
  - progress_id: Primary key, auto-incremented integer.
  - user: Foreign key to User (The user whose progress is tracked).
  - course: Foreign key to Course (The course for which progress is tracked).
  - completed_lessons: The number of lessons completed by the user.
  - quiz_scores: The user's quiz score.
  - progress_date: The date and time when the progress record was created.
- **Relationships**:
  - Many-to-one relationship with User (A user can have many progress records).
  - Many-to-one relationship with Course (A course can have many progress records).

UserQuizAnswer Table:
The UserQuizAnswer table stores the answers selected by users for each question in a quiz.

- **Fields**:
  - user: Foreign key to User (The user who answered the question).
  - quiz: Foreign key to Quiz (The quiz in which the question is asked).
  - question: Foreign key to QuizQuestion (The question being answered).
  - selected_option: The selected answer option (A, B, C, or D).
- **Relationships**:
  - Many-to-one relationship with User (A user can answer many questions).
  - Many-to-one relationship with Quiz (A quiz can have many answers).
  - Many-to-one relationship with QuizQuestion (A question can have many answers).

Each table in the schema is related to others through foreign key relationships, and the design ensures that the platform can handle operations like enrolling users, tracking progress, and associating lessons with quizzes. The use of foreign keys enforces referential integrity between the tables, ensuring that data remains consistent across the system. The primary keys ensure each record is uniquely identifiable, and constraints like unique fields for email and username prevent data duplication.

Intro to Containerization: Docker

Containerization is a modern approach to software development that allows applications to run in isolated environments called containers. Each container includes everything an application needs to function, such as the code, runtime, libraries, and system tools. This encapsulation provides several benefits.

One of the main advantages of containerization is portability. Since containers package all dependencies, applications can run consistently across different environments, from development to production, without worrying about configuration issues. This means that if an application works on one machine, it will work on any other machine that has Docker installed.

Another significant benefit is isolation. Each container operates independently of others, ensuring that applications do not interfere with one another. This isolation also enhances security, as it limits the potential impact of vulnerabilities in one application on the rest of the system.

Docker Installation

To get started with Docker, I followed these steps for installation:

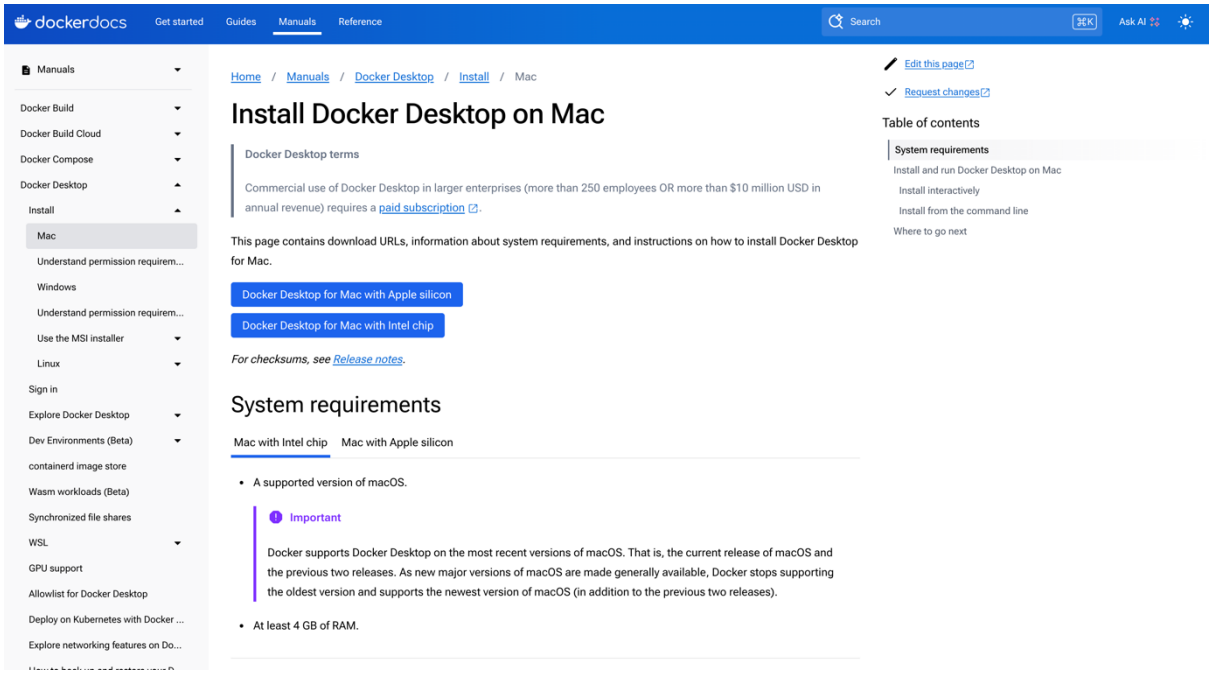


Figure 1: Installation of Docker

I visited the official Docker website and downloaded the Docker Desktop application (Figure 1).

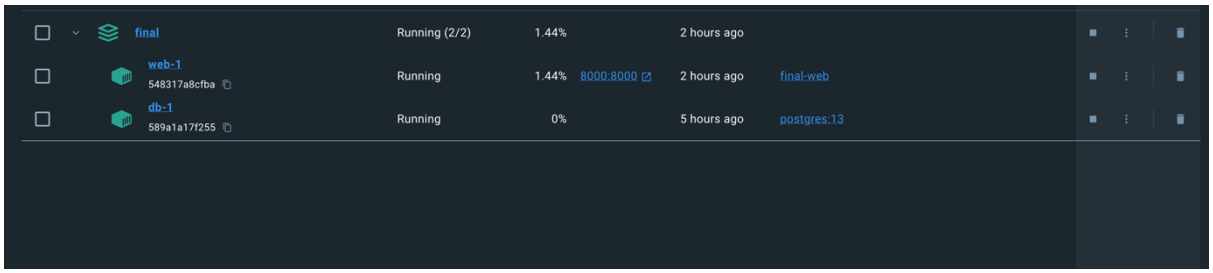


Figure 2: Docker Desktop

Once the installation was complete, I launched Docker Desktop (Figure 2). I had to wait for the application to initialize, which ensured that all necessary components were running.

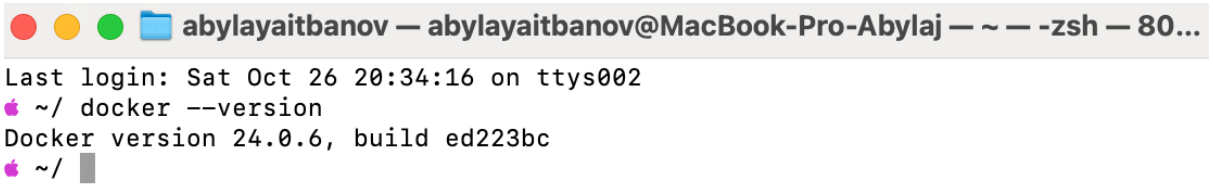


Figure 3: Docker version

To confirm that Docker was installed correctly, I opened a terminal and ran the command docker --version. This displayed the installed version of Docker (Figure 2).

Figure 4: Run a Simple Container

Then I tested the installation by running a simple container using the command: **docker run hello-world** (Figure 4). This command downloaded a test image and ran it, displaying a message confirming that Docker was installed and functioning properly.

Dockerfile

The Dockerfile is a crucial component in building a Docker image for the Django application. It contains a set of instructions that Docker uses to assemble the image, ensuring that the application runs in a consistent environment. The structure of the Dockerfile typically includes several key components.

Figure 5: Base Image

The Dockerfile starts with a base image (Figure 5) that provides the necessary environment. In this case, I used the slim version of the official Python image, which is lightweight and optimized for running Python applications.

Figure 6: Author Information

I included a label in the Dockerfile to indicate the author of the image (Figure 6). This is helpful for documentation purposes.

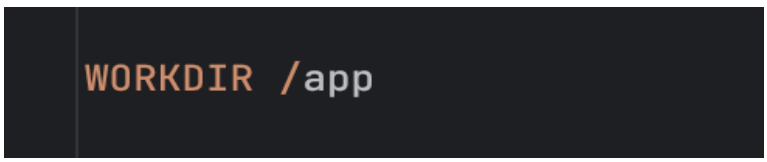


Figure 7: Working Directory

The WORKDIR instruction sets the working directory within the container to /app (Figure 7). This is where all subsequent commands will be executed.

Figure 8: Copying Requirements File

The COPY instruction copies the requirements.txt file from the local machine to the working directory in the container (Figure 8).

Figure 9: Installing Dependencies

After copying the requirements file, I used the RUN instruction to install the dependencies listed in requirements.txt. The --no-cache-dir option prevents caching of the installation, making the image smaller (Figure 9).

Figure 10: Copying Application Files

The next step is to copy (Figure 10) the entire application code into the container. This includes all the necessary files for the Django application.

Figure 11: Exposing Ports

To allow external access to the application, I specified the port on which the application will run. In this case, the application will be accessible on port 8000 (Figure 11).

Docker-Compose

Docker Compose simplifies the process of managing multi-container Docker applications. It allows me to define and run multiple services in a single file, making it easier to orchestrate the different components of my application. For this project, I created a docker-compose.yml file that specifies both the Django application and the PostgreSQL database.

Compose File Structure

The docker-compose.yml file is organized into several key sections that define the services, networks, and volumes for the application. Here's the structure of my docker-compose.yml file (Figure 12).

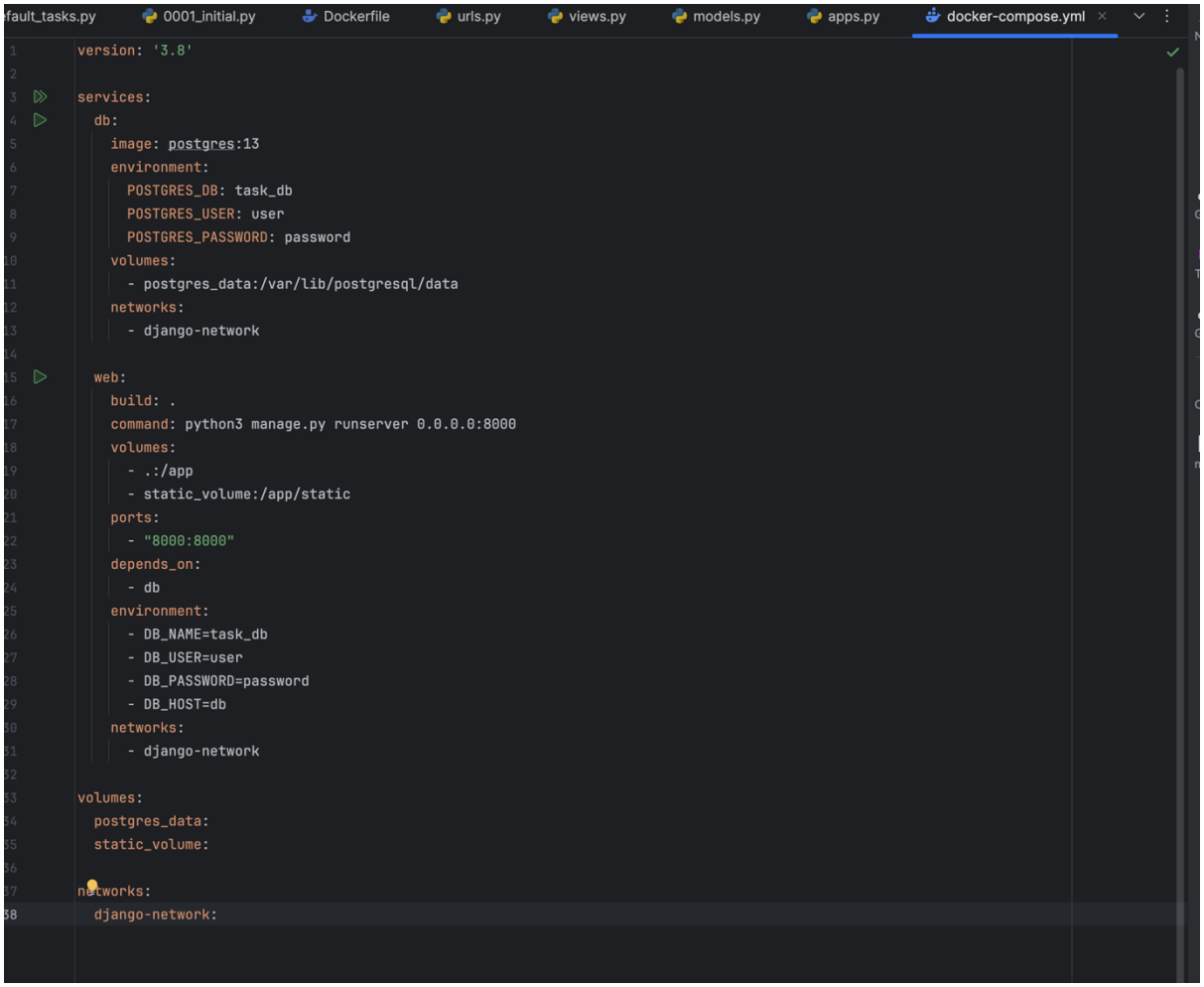


Figure 12: Docker-compose.yml file

Service Definitions

Database Service (db**)** - this service defines the PostgreSQL database that will be used by the Django application. The main components include:

Image: The image: postgres:13 line specifies that the PostgreSQL 13 image should be used from Docker Hub.

Environment Variables: The environment section sets up the initial database configuration, including the database name (task_db), username (user), and password (password).

Volumes: The volumes section mounts a named volume (postgres_data) to /var/lib/postgresql/data in the container, ensuring that the database data persists across container restarts.

Networks: The networks section connects this service to the django-network, allowing communication between containers.

Web Service (web**)**: This service represents the Django application. Here are the main components of this service definition:

Build Context: The build: . line indicates that Docker should build the image using the Dockerfile located in the current directory.

Command: The command: python3 manage.py runserver 0.0.0.0:8000 line specifies the command to run the Django development server, listening on all network interfaces.

Volumes: The volumes section mounts the current directory (.) to the /app directory in the container, allowing live code updates. Additionally, it creates a volume for static files with static_volume:/app/static.

Ports: The ports section maps port 8000 of the container to port 8000 on the host machine, making the Django application accessible via http://localhost:8000.

Dependencies: The depends_on directive ensures that the db service starts before the web service.

Environment Variables: The environment section sets up connection details for the database, including the database name, user, password, and host.

Networks: Similar to the database service, this service is also connected to the django-network.

Volumes and Networks: The volumes section defines named volumes for persisting PostgreSQL data (postgres_data) and static files (static_volume). The networks section defines the django-network, enabling communication between the web and db services.

Docker Networking and Volumes

Docker simplifies the way containers communicate and manage data through networking and volumes. Here's how these features work in my project.

Networking

```
networks:
  django-network:
```

Figure 13: Docker networking

Docker networking (Figure 13) allows containers to talk to each other easily. In my application, I created a custom network called django-network. This setup has several benefits:

Isolation: Each container runs separately, which keeps them secure and organized. By using a specific network, I make sure only the services that need to communicate can do so.

Easy Communication: Instead of using complicated IP addresses, containers can connect using their service names. For example, the Django app (web service) can access the PostgreSQL database (db service) simply by using the name db. This makes the application more flexible and easier to manage.

Scalability: If I need to run multiple instances of the Django app, they can all share the same network. This allows them to communicate with the database without any issues.

Volumes

```
volumes:
  - .:/app
  - static_volume:/app/static
```

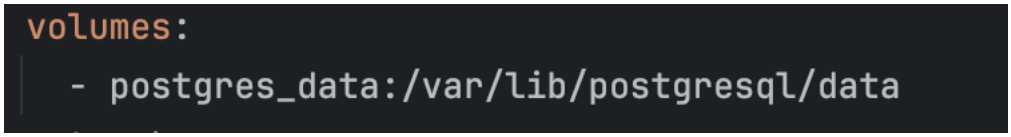


Figure 14: Docker volumes

Volumes are essential for keeping data safe and persistent. In my project, I used volumes to ensure that my PostgreSQL (Figure 14) database data doesn't get lost when containers are restarted or removed. Here's how I utilized volumes:

Data Persistence: The postgres_data volume is mounted to the PostgreSQL container at /var/lib/postgresql/data. This means that all the database data is stored in this volume, so it stays intact even if I stop or recreate the container.

Easy Backups: With volumes, I can easily back up my database data. I can create a backup without affecting the running container, making recovery simple if something goes wrong.

Static Files: I also used a volume called static_volume for the Django app's static files. This ensures that these files remain available even if the application container is rebuilt.

Django

Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. It was created to simplify the complexities of web development and make it easier for developers to build powerful, secure, and maintainable web applications.

Django follows the "batteries-included" philosophy, providing a comprehensive set of built-in tools and features that can handle most common web development tasks. This makes it an ideal choice for developers looking for an all-in-one solution that allows them to focus more on business logic rather than infrastructure and repetitive coding tasks.

For applications that require an API, Django is also well-suited for building RESTful APIs. With the Django Rest Framework (DRF), developers can quickly create and manage API endpoints, which is especially useful for applications that need to interact with web and mobile clients.

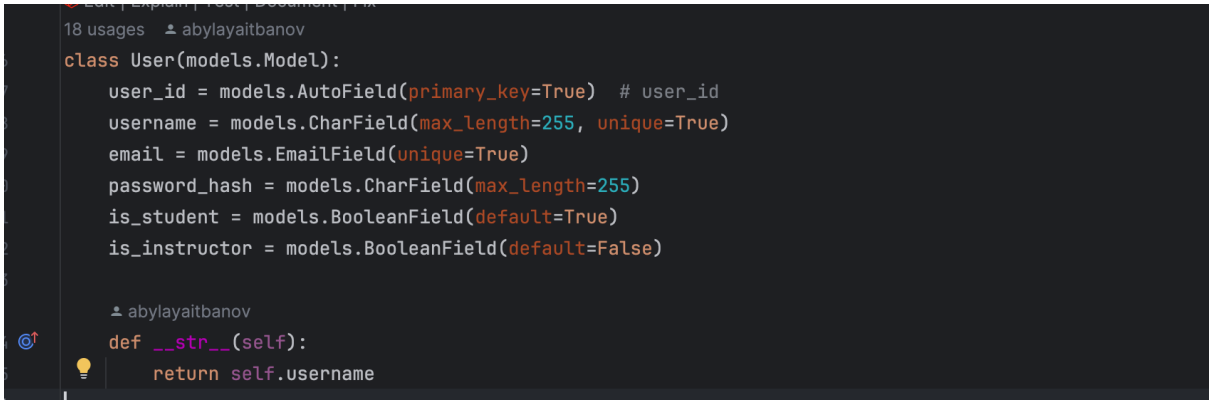
In terms of community support, Django boasts a large and active user base, as well as comprehensive documentation and resources. This ensures that developers can easily find solutions to problems, tutorials, and third-party packages to extend the framework's functionality.

In conclusion, Django is a powerful, flexible, and secure web framework that is well-suited for building a wide range of web applications. Its ability to handle complex, data-driven websites while providing tools to simplify development tasks makes it a preferred choice for many developers in the web development community.

Models

In Django, models are the heart of the application's data structure. They define the schema of the database, representing the data and its relationships. Each model corresponds to a table in the database, with each attribute of the model becoming a column in that table. Models provide a high-level abstraction over the database, allowing developers to interact with the data using Python code instead of raw SQL queries.

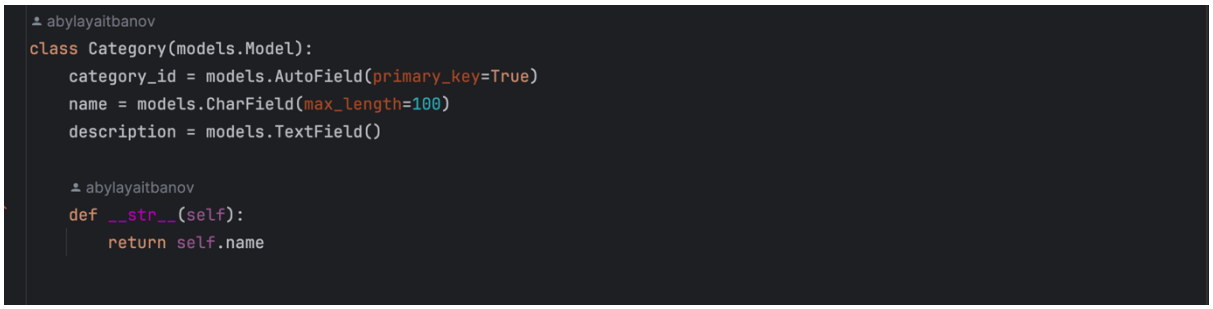
In the context of the e-learning platform, several models were created to structure and manage the various components of the platform. Below are the key models defined for the project:



```
18 usages  abylayaitbanov
class User(models.Model):
    user_id = models.AutoField(primary_key=True)  # user_id
    username = models.CharField(max_length=255, unique=True)
    email = models.EmailField(unique=True)
    password_hash = models.CharField(max_length=255)
    is_student = models.BooleanField(default=True)
    is_instructor = models.BooleanField(default=False)

    abylayaitbanov
    def __str__(self):
        return self.username
```

Figure 15: User model

This model represents the users of the platform. It includes fields such as username, email, and password_hash to store the user's credentials. Additionally, it has boolean fields like is_student and is_instructor to distinguish between students and instructors. The user_id serves as the primary key for identifying users uniquely in the system.



```
abylayaitbanov
class Category(models.Model):
    category_id = models.AutoField(primary_key=True)
    name = models.CharField(max_length=100)
    description = models.TextField()

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

Figure 16: Category model

Categories are used to group courses into different subjects or areas of study. This model has a name field to represent the category name and a description field to provide more details about the category. The category_id field serves as the primary key.

```
18 usages  abylayaitbanov
class Course(models.Model):
    course_id = models.AutoField(primary_key=True)
    title = models.CharField(max_length=255)
    description = models.TextField()
    price = models.DecimalField(max_digits=10, decimal_places=2)
    category = models.ForeignKey(Category, on_delete=models.SET_NULL, null=True)
    created_at = models.DateTimeField(auto_now_add=True)
    instructor = models.ForeignKey(User, on_delete=models.CASCADE, null=False)

    abylayaitbanov
    def __str__(self):
        return self.title
```

Figure 17: Course model

This model stores the courses offered on the platform. It contains fields like title, description, price, and created_at to store relevant course details. The model also has foreign key relationships with the Category and User models. Each course is linked to a category and an instructor (who is represented by a user). The course_id is the primary key.

Figure 18: Enrollment Model

The Enrollment model links users (students) to the courses they are enrolled in. It includes a status field to track the student's progress, whether they are currently enrolled or have completed the course. The model has foreign key relationships with the User and Course models. The enrollment_id is the primary key, and a unique_together constraint ensures that each user can only enroll in a course once with a given status.

Figure 19: Lesson Model

The Lesson model represents the individual lessons within each course. It contains a title, content, and a video_url to store lesson details. Each lesson is linked to a specific course via a foreign key relationship. The lesson_id is the primary key for each lesson.

```python
class Review(models.Model):
    review_id = models.AutoField(primary_key=True)  # review_id
    course = models.ForeignKey(Course, on_delete=models.CASCADE)
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    rating = models.IntegerField()
    comment = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)

    abylayaitbanov
    def __str__(self):
        return f'{self.user.username} - {self.course.title}'
```

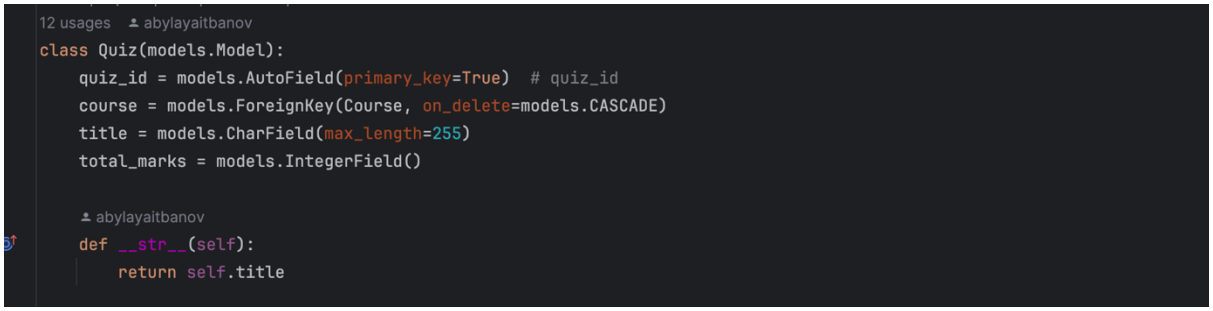
Figure 20: Review Model

This model allows users to leave reviews for courses. It includes fields such as rating, comment, and created_at to store feedback about the course. The Review model has foreign key relationships with both the User and Course models, linking the review to the user who wrote it and the course it pertains to. The review_id is the primary key.

```python
7 usages  abylayaitbanov
class Payment(models.Model):
    payment_id = models.AutoField(primary_key=True)
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    course = models.ForeignKey(Course, on_delete=models.CASCADE, null=True)
    amount = models.DecimalField(max_digits=10, decimal_places=2)
    payment_date = models.DateTimeField(auto_now_add=True)
    status = models.CharField(max_length=50, choices=[('pending', 'Pending'), ('completed', 'Completed')])

    abylayaitbanov
    class Meta:
        unique_together = ['user', 'course', 'status']
```

Figure 21: Payment Model

The Payment model tracks payments made by users for courses. It includes fields like amount, payment_date, and status to manage payment transactions. The model has foreign key relationships with the User and Course models to link payments to specific users and courses. The payment_id is the primary key, and the unique_together constraint ensures that each payment is uniquely identified by the combination of user, course, and payment status.



```python
12 usages  abylayaitbanov
class Quiz(models.Model):
    quiz_id = models.AutoField(primary_key=True)  # quiz_id
    course = models.ForeignKey(Course, on_delete=models.CASCADE)
    title = models.CharField(max_length=255)
    total_marks = models.IntegerField()

    abylayaitbanov
    def __str__(self):
        return self.title
```

Figure 22: Quiz Model

The Quiz model represents quizzes associated with courses. It contains fields like title and total_marks to describe the quiz, and it is linked to a course via a foreign key relationship. The quiz_id serves as the primary key.

```python
10 usages  abylayaitbanov
class QuizQuestion(models.Model):
    question_id = models.AutoField(primary_key=True)  # question_id
    quiz = models.ForeignKey(Quiz, on_delete=models.CASCADE)
    question_text = models.TextField()
    option_a = models.CharField(max_length=255)
    option_b = models.CharField(max_length=255)
    option_c = models.CharField(max_length=255)
    option_d = models.CharField(max_length=255)
    correct_option = models.CharField(max_length=1, choices=[('a', 'A'), ('b', 'B'), ('c', 'C'), ('d', 'D')])

    Edit | Explain | Test | Document | Fix
    abylayaitbanov
    def __str__(self):
        return (f'{self.quiz.title} - {self.question_text} | '
                f'A: {self.option_a} | B: {self.option_b} | '
                f'C: {self.option_c} | D: {self.option_d} ')
```

Figure 23: QuizQuestion Model

The QuizQuestion model stores individual questions for each quiz. It includes fields like question_text and multiple-choice options (option_a, option_b, option_c, option_d). It also includes a correct_option field to mark the correct answer. Each quiz question is related to a specific quiz via a foreign key relationship. The question_id is the primary key.

```python
class UserProgress(models.Model):
    progress_id = models.AutoField(primary_key=True)  # progress_id
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    course = models.ForeignKey(Course, on_delete=models.CASCADE)
    completed_lessons = models.IntegerField()
    quiz_scores = models.FloatField(default=0.0)
    progress_date = models.DateTimeField(default=timezone.now)

    abylayaitbanov
    def __str__(self):
        return f'{self.user.username} - {self.course.title}'
```

Figure 24: UserProgress Model

This model tracks the progress of students in each course. It includes fields for completed_lessons, quiz_scores, and a progress_date. Each entry is associated with a user and a course through foreign key relationships. The progress_id is the primary key.

```python
8 usages  abylayaitbanov
class UserQuizAnswer(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    quiz = models.ForeignKey(Quiz, on_delete=models.CASCADE)
    question = models.ForeignKey(QuizQuestion, on_delete=models.CASCADE)
    selected_option = models.CharField(max_length=1, choices=[('A', 'Option A'), ('B', 'Option B'), ('C', 'Option C'),
                                                              ('D', 'Option D')])

    Edit | Explain | Test | Document | Fix
    abylayaitbanov
    def __str__(self):
        # Add all options to the string representation
        return f"User {self.user.username} answer for {self.quiz.title} - {self.question.question_text} | " \
               f"A: {self.question.option_a} | B: {self.question.option_b} | " \
               f"C: {self.question.option_c} | D: {self.question.option_d} | " \
               f"Selected: {self.selected_option}"
```

Figure 25: UserQuizAnswer Model

The UserQuizAnswer model stores the answers given by users for each quiz question. It includes fields like selected_option to track which option the user selected for each question. It also maintains foreign key relationships with the User, Quiz, and QuizQuestion models. Each answer is uniquely identified by the combination of user, quiz, and question.

These models form the backbone of the e-learning platform, allowing for efficient management of users, courses, lessons, enrollments, payments, quizzes, and progress tracking. They are connected through foreign key relationships, ensuring data integrity and enabling the development of rich, data-driven features for the platform.

Views (Django Rest Framework (DRF))

Django Rest Framework (DRF) is a powerful and flexible toolkit for building Web APIs in Django. It simplifies the process of creating RESTful APIs by providing a set of pre-built components and tools for handling common tasks like serialization, authentication, and view management. DRF builds on top of Django and extends its functionality to support more robust and scalable API development.

The API architecture for the e-learning platform is designed to support a variety of interactions, from course browsing to user enrollment, payments, and quizzes. The API is built using Django Rest Framework to expose the platform's resources in a RESTful manner, making it easy to interact with through HTTP requests

Advantages of DRF

Ease of Use

DRF provides an easy-to-use interface for developers. With minimal setup, developers can expose their models as APIs and create endpoints for performing CRUD (Create, Read, Update, Delete) operations. This makes it a great choice for rapidly developing APIs.

Serialization

DRF comes with powerful serializers that allow complex data structures such as Django models, querysets, and custom data types to be easily converted into JSON, XML, or other formats suitable for web communication. This makes it simple to return structured data to clients, such as mobile apps or web applications.

```
class CourseSerializer(serializers.ModelSerializer):
    instructor = serializers.PrimaryKeyRelatedField(queryset=User.objects.filter(is_instructor=True), required=True)
    category = serializers.PrimaryKeyRelatedField(queryset=Category.objects.all(), required=True)

    abylayaitbanov
    class Meta:
        model = Course
        fields = ['course_id', 'title', 'description', 'price', 'category', 'created_at', 'instructor']

    Edit | Explain | Test | Document | Fix
    abylayaitbanov
    @staticmethod
    def validate_category(value):
        # Ensure category exists and is valid
        if not value:
            raise serializers.ValidationError("Category is required.")
        return value
```

Figure 26: CourseSerializer

Serializers provide built-in validation features, making it easy to ensure that the data adheres to certain rules before it is processed. For example:

The CourseSerializer (Figure 26) checks whether the category and instructor values are valid by referencing existing models. The validation ensures that: A valid category is provided (using validate_category).The instructor is a valid user with the is_instructor flag set to True (using validate_instructor).

```
2 usages   abylayaitbanov
class EnrollmentSerializer(serializers.ModelSerializer):
    user = serializers.PrimaryKeyRelatedField(queryset=User.objects.all(), required=True)
    course = serializers.PrimaryKeyRelatedField(queryset=Course.objects.all(), required=True)

    abylayaitbanov
    class Meta:
        model = Enrollment
        fields = ['enrollment_id', 'user', 'course', 'enrollment_date', 'status']

    Edit | Explain | Test | Document | Fix
    abylayaitbanov
    def validate(self, data):
        user = data.get('user')
        course = data.get('course')
        status = data.get('status')

        if Enrollment.objects.filter(user=user, course=course, status=status).exists():
            raise serializers.ValidationError("The user is already enrolled in this course.")

        return data
```

Figure 27: EnrollmentSerializer

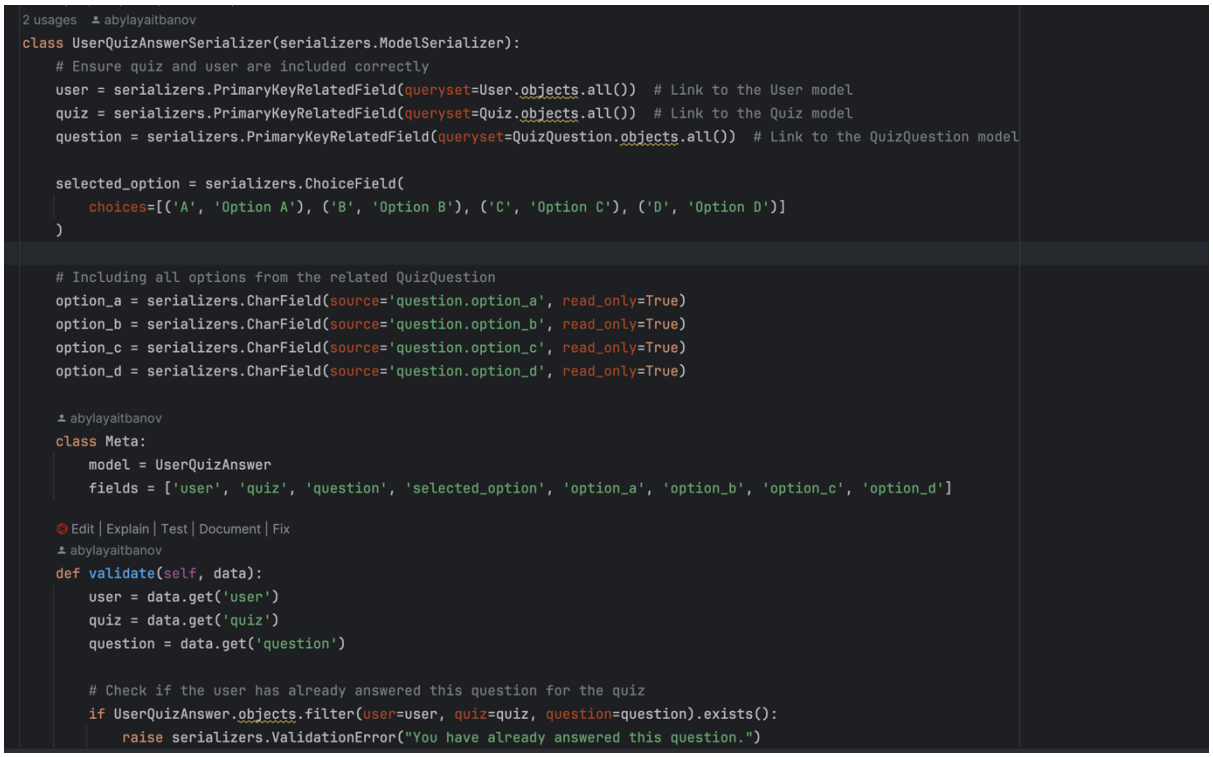
The EnrollmentSerializer ensures that a user is not enrolled in the same course twice by checking for an existing enrollment with the same course and status.

```
2 usages  abylayaitbanov
class PaymentSerializer(serializers.ModelSerializer):
    user = serializers.PrimaryKeyRelatedField(queryset=User.objects.all(), required=True)
    course = serializers.PrimaryKeyRelatedField(queryset=Course.objects.all(), required=True)

    abylayaitbanov
    class Meta:
        model = Payment
        fields = ['payment_id', 'user', 'course', 'amount', 'payment_date', 'status']

    Edit | Explain | Test | Document | Fix
    abylayaitbanov
    def validate(self, data):
        user = data.get('user')
        course = data.get('course')
        status = data.get('status')

        if Payment.objects.filter(user=user, course=course, status=status).exists():
            raise serializers.ValidationError("You have already made a payment for this course.")

        return data
```

Figure 28: PaymentSerializer

The PaymentSerializer validates that a user has not already made a payment for the same course, preventing duplicate payments.

Serializers allow you to define how the data should be represented or handled on a field-by-field basis



```
2 usages  abylayaitbanov
class UserQuizAnswerSerializer(serializers.ModelSerializer):
    # Ensure quiz and user are included correctly
    user = serializers.PrimaryKeyRelatedField(queryset=User.objects.all())  # Link to the User model
    quiz = serializers.PrimaryKeyRelatedField(queryset=Quiz.objects.all())  # Link to the Quiz model
    question = serializers.PrimaryKeyRelatedField(queryset=QuizQuestion.objects.all())  # Link to the QuizQuestion model

    selected_option = serializers.ChoiceField(
        choices=[('A', 'Option A'), ('B', 'Option B'), ('C', 'Option C'), ('D', 'Option D')]
    )

    # Including all options from the related QuizQuestion
    option_a = serializers.CharField(source='question.option_a', read_only=True)
    option_b = serializers.CharField(source='question.option_b', read_only=True)
    option_c = serializers.CharField(source='question.option_c', read_only=True)
    option_d = serializers.CharField(source='question.option_d', read_only=True)

    abylayaitbanov
    class Meta:
        model = UserQuizAnswer
        fields = ['user', 'quiz', 'question', 'selected_option', 'option_a', 'option_b', 'option_c', 'option_d']

    Edit | Explain | Test | Document | Fix
    abylayaitbanov
    def validate(self, data):
        user = data.get('user')
        quiz = data.get('quiz')
        question = data.get('question')

        # Check if the user has already answered this question for the quiz
        if UserQuizAnswer.objects.filter(user=user, quiz=quiz, question=question).exists():
            raise serializers.ValidationError("You have already answered this question.")
```

Figure 29: UserQuizAnswerSerializer

The UserQuizAnswerSerializer includes additional fields such as option_a, option_b, option_c, and option_d, sourced from the related QuizQuestion model. These fields allow

you to display the options alongside the user's selected answer without storing these options separately in the UserQuizAnswer model.

Views and Viewsets

DRF viewsets are used to define the behavior for each resource, such as listing courses, viewing course details, enrolling in a course, and submitting quizzes. The viewsets handle requests and manage interactions with the underlying models, performing actions like retrieving data, creating records, or updating information.

```
2 usages  abylayaitbanov
class UserViewSet(viewsets.ModelViewSet):
    queryset = User.objects.all()
    serializer_class = UserSerializer


2 usages  abylayaitbanov
class CategoryViewSet(viewsets.ModelViewSet):
    queryset = Category.objects.all()
    serializer_class = CategorySerializer


2 usages  abylayaitbanov
class CourseViewSet(viewsets.ModelViewSet):
    queryset = Course.objects.all()
    serializer_class = CourseSerializer
```

Figure 30: UserViewSet, CategoryViewSet, CourseViewSet

UserViewSet, **CategoryViewSet**, **CourseViewSet**, automatically provide behavior for handling requests like:

- GET /users/: List all users.
- POST /users/: Create a new user.
- GET /users/{id}/: Retrieve a specific user.
- PUT /users/{id}/: Update a user.
- DELETE /users/{id}/: Delete a user.

This is made possible by inheriting from viewsets.ModelViewSet, where:

- queryset defines the data that can be retrieved from the database.
- serializer_class links the viewset to the appropriate serializer, which handles the validation and transformation of data between models and JSON.

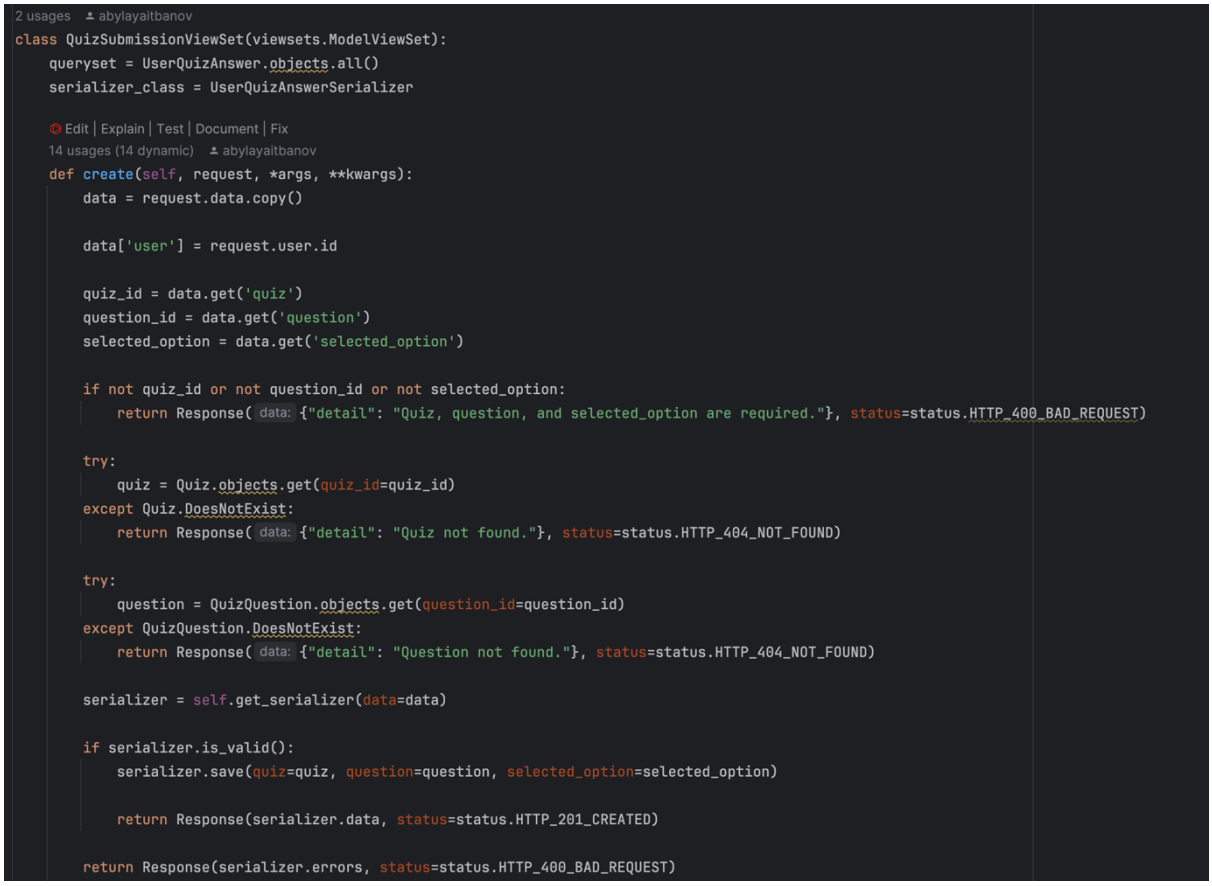
While ModelViewSet automates the CRUD operations, sometimes you need to customize the behavior. The create() method allows you to define custom behavior for the creation of new records.

```
2 usages  abylayaitbanov
class UserProgressViewSet(viewsets.ModelViewSet):
    queryset = UserProgress.objects.all()
    serializer_class = UserProgressSerializer

    Edit | Explain | Test | Document | Fix
    14 usages (14 dynamic)  abylayaitbanov
    def create(self, request, *args, **kwargs):
        data = request.data.copy()

        data['user'] = request.user.id

        course_id = request.data.get('course')
        if not course_id:
            return Response( data: {"detail": "Course is required."}, status=status.HTTP_400_BAD_REQUEST)

        serializer = self.get_serializer(data=data)

        if serializer.is_valid():
            serializer.save()  # Save the UserProgress instance
            return Response(serializer.data, status=status.HTTP_201_CREATED)

        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Figure 31: UserProgressViewSet

UserProgressViewSet:

The create() method customizes how user progress is recorded. Here, it ensures that the user field is automatically populated with the logged-in user (request.user.id).It also checks if a course is provided in the request data. If not, it returns a 400 Bad Request response, ensuring that no progress is recorded without a course.



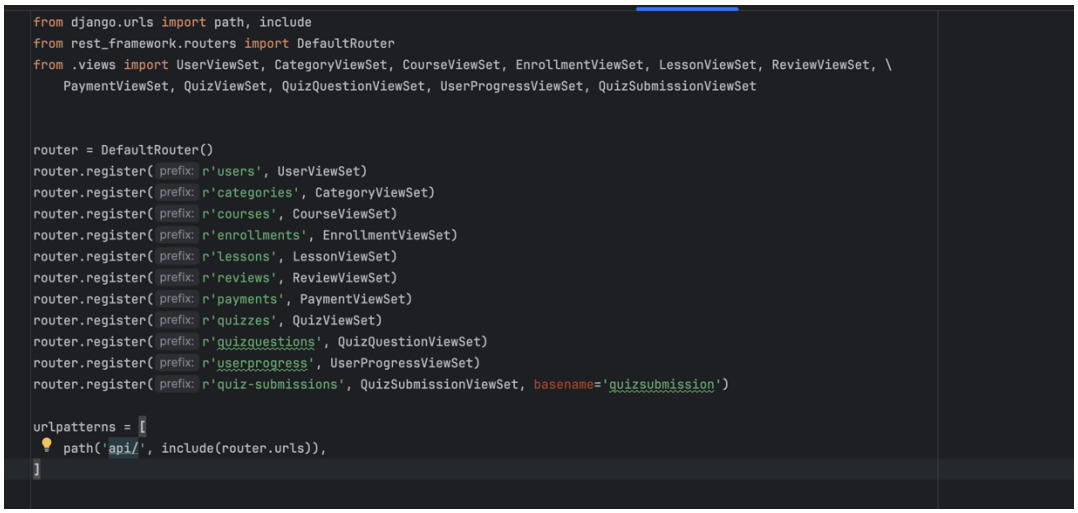
```
2 usages  abylayaitbanov
class QuizSubmissionViewSet(viewsets.ModelViewSet):
    queryset = UserQuizAnswer.objects.all()
    serializer_class = UserQuizAnswerSerializer

    Edit | Explain | Test | Document | Fix
    14 usages (14 dynamic)  abylayaitbanov
    def create(self, request, *args, **kwargs):
        data = request.data.copy()

        data['user'] = request.user.id

        quiz_id = data.get('quiz')
        question_id = data.get('question')
        selected_option = data.get('selected_option')

        if not quiz_id or not question_id or not selected_option:
            return Response( data: {"detail": "Quiz, question, and selected_option are required."}, status=status.HTTP_400_BAD_REQUEST)

        try:
            quiz = Quiz.objects.get(quiz_id=quiz_id)
        except Quiz.DoesNotExist:
            return Response( data: {"detail": "Quiz not found."}, status=status.HTTP_404_NOT_FOUND)

        try:
            question = QuizQuestion.objects.get(question_id=question_id)
        except QuizQuestion.DoesNotExist:
            return Response( data: {"detail": "Question not found."}, status=status.HTTP_404_NOT_FOUND)

        serializer = self.get_serializer(data=data)

        if serializer.is_valid():
            serializer.save(quiz=quiz, question=question, selected_option=selected_option)

            return Response(serializer.data, status=status.HTTP_201_CREATED)

        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Figure 32: QuizSubmissionViewSet

QuizSubmissionViewSet:

The create() method checks if the required fields (quiz, question, selected_option) are present.It validates that the specified quiz and question exist in the database. If they don't, it returns a 404 Not Found error. It then serializes and saves the answer, linking the submitted data with the corresponding quiz and question.

URLs and Routers



```
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from .views import UserViewSet, CategoryViewSet, CourseViewSet, EnrollmentViewSet, LessonViewSet, ReviewViewSet, \
    PaymentViewSet, QuizViewSet, QuizQuestionViewSet, UserProgressViewSet, QuizSubmissionViewSet


router = DefaultRouter()
router.register( prefix: r'users', UserViewSet)
router.register( prefix: r'categories', CategoryViewSet)
router.register( prefix: r'courses', CourseViewSet)
router.register( prefix: r'enrollments', EnrollmentViewSet)
router.register( prefix: r'lessons', LessonViewSet)
router.register( prefix: r'reviews', ReviewViewSet)
router.register( prefix: r'payments', PaymentViewSet)
router.register( prefix: r'quizzes', QuizViewSet)
router.register( prefix: r'quizquestions', QuizQuestionViewSet)
router.register( prefix: r'userprogress', UserProgressViewSet)
router.register( prefix: r'quiz-submissions', QuizSubmissionViewSet, basename='quizsubmission')

urlpatterns = [
    path('api/', include(router.urls)),
]
```

Figure 33: DefaultRouter

DRF uses routers to automatically generate URL mappings for the viewsets. This simplifies URL management and reduces the need for manually defining each route. The DefaultRouter class is used to automatically generate endpoints like /courses/, /courses/{id}/, and /enrollments/. By using routers, the API can easily scale as new endpoints are added.

This approaches make building REST APIs with Django much more manageable and maintainable, allowing developers to focus on business logic while DRF handles the repetitive aspects of API creation.

Frontend Integration

When integrating Vue.js with a Django backend through APIs, the process involves making HTTP requests from the Vue.js frontend to the Django backend, which is powered by Django Rest Framework (DRF).

In Django, Django Rest Framework (DRF) is used to create RESTful APIs. These APIs expose endpoints for CRUD operations (Create, Read, Update, Delete), such as:

- GET /courses/: Retrieve a list of courses.
- POST /courses/: Create a new course.
- PUT /courses/{id}/: Update an existing course.
- DELETE /courses/{id}/: Delete a course.

These APIs are handled by viewsets or API views in Django. To make HTTP requests to the Django API, you can use **Axios**, a popular library for making HTTP requests. If frontend and backend are hosted on different domains or ports (e.g., Django on http://127.0.0.1:8000 and Vue.js on http://localhost:8080), then it needs to handle **CORS**. Install and configure django-cors-headers in your Django project to allow the frontend to make requests to the backend.

Challenges and Solutions

```python
class CourseSerializer(serializers.ModelSerializer):
    instructor = serializers.PrimaryKeyRelatedField(queryset=User.objects.filter(is_instructor=True), required=True)
    category = serializers.PrimaryKeyRelatedField(queryset=Category.objects.all(), required=True)

    abylayaitbanov
    class Meta:
        model = Course
        fields = ['course_id', 'title', 'description', 'price', 'category', 'created_at', 'instructor']

    Edit | Explain | Test | Document | Fix
    abylayaitbanov
    @staticmethod
    def validate_category(value):
        # Ensure category exists and is valid
        if not value:
            raise serializers.ValidationError("Category is required.")
        return value

    Edit | Explain | Test | Document | Fix
    abylayaitbanov
    @staticmethod
    def validate_instructor(value):
        # Ensure the user is actually an instructor
        if not value.is_instructor:
            raise serializers.ValidationError("The user must be an instructor.")
        return value
```

Figure 34: CourseSerializer

One of the main challenges was ensuring that duplicate courses were not created in the system. Users should not be able to create multiple courses with the same title, as it could lead to confusion and a poor user experience. To resolve this, a custom validation method was added to the CourseSerializer to check for existing courses with the same title. If a course with the same title already exists, a ValidationError is raised, preventing the creation of a duplicate course. Another challenge was to ensure that students could not pay for or enroll in the same course multiple times without appropriate handling of data integrity, also custom validation logic was added in both the PaymentSerializer and EnrollmentSerializer to prevent students from enrolling in or paying for the same course more than once.

Keeping track of users' progress in courses, such as completed lessons and quiz scores, was complex due to multiple data points being involved (lessons, quizzes, and course completion status). A custom UserProgressSerializer was developed to manage and validate progress, ensuring that users could not make multiple entries for the same course or incomplete progress. It also included the logic for tracking completed lessons and quiz scores.

Conclusion

In conclusion, i'd like to say i successfully created an e-learning platform with features such as course enrollment, payment processing, quizzes, and user progress tracking. Key highlights include robust data validation to prevent duplicate courses and payments, and a clean integration between the backend (Django) and frontend (Vue.js). The project showed the importance of handling user roles properly, ensuring data integrity, and providing a smooth experience for both students and instructors.

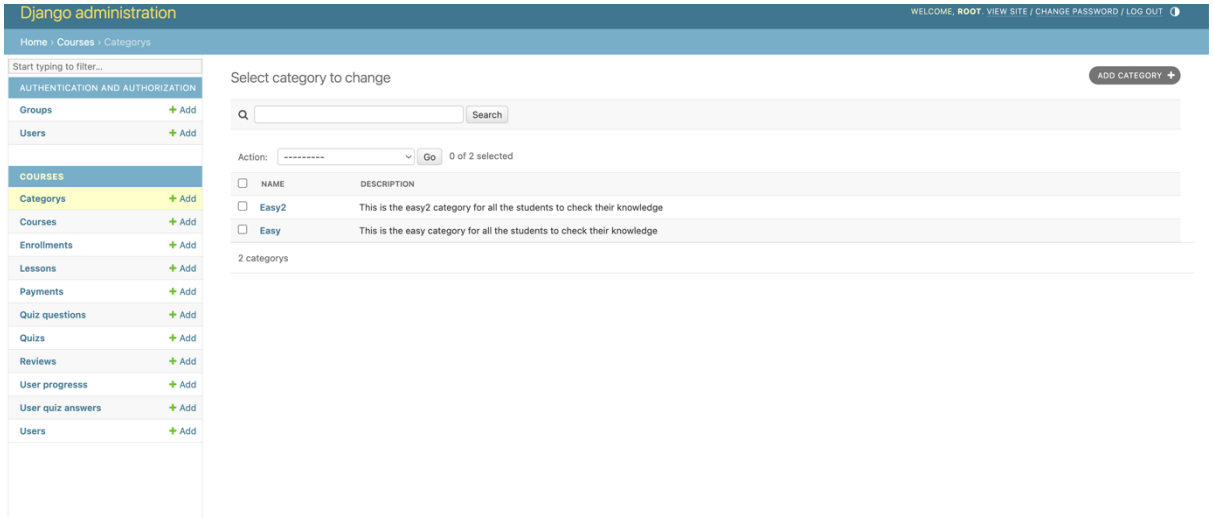
The main lessons learned were the value of clear data validation and user access control, as well as the importance of keeping the backend and frontend in sync. For future

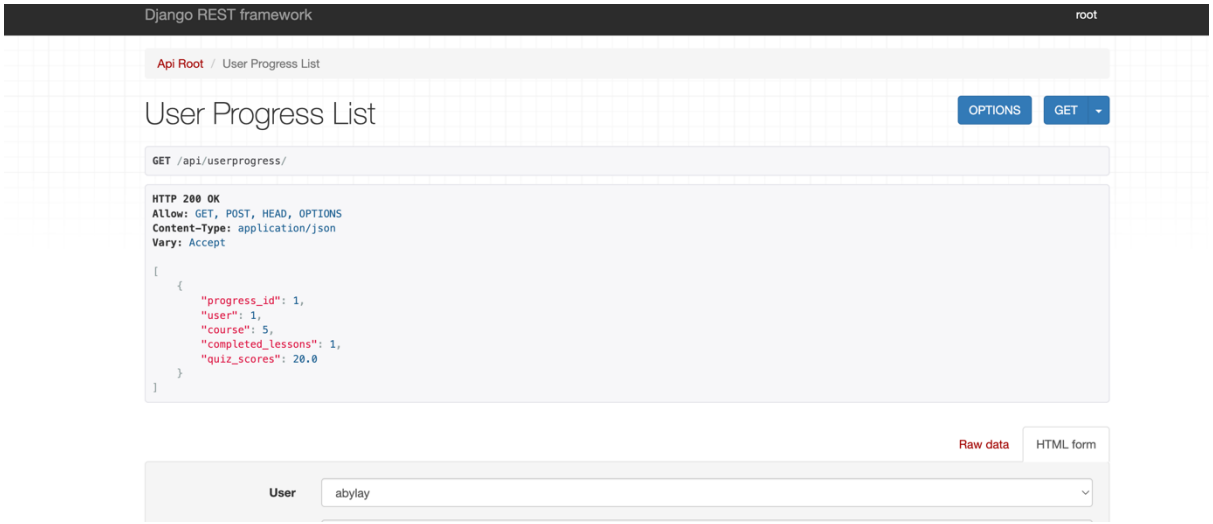
improvements, adding real-time features, course recommendations, and mobile support could enhance the platform further.

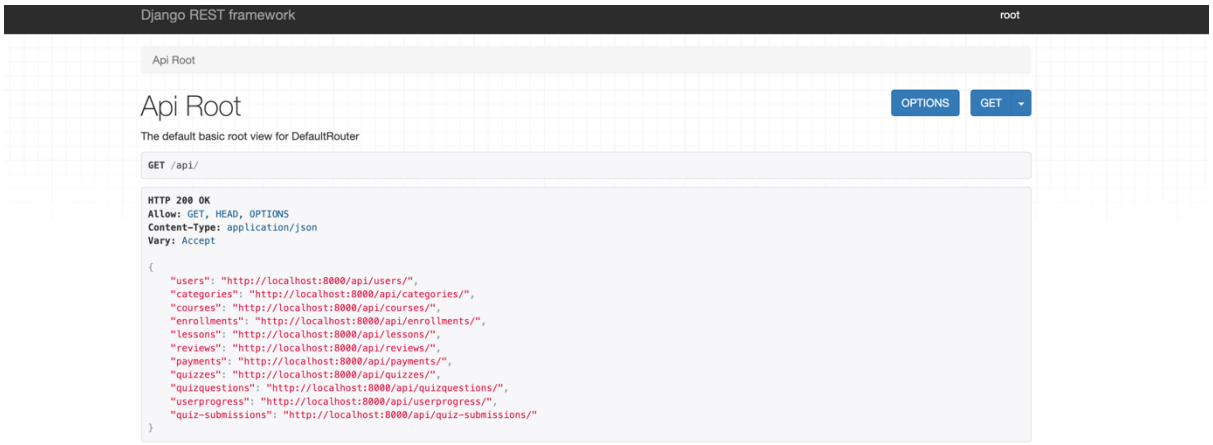
References

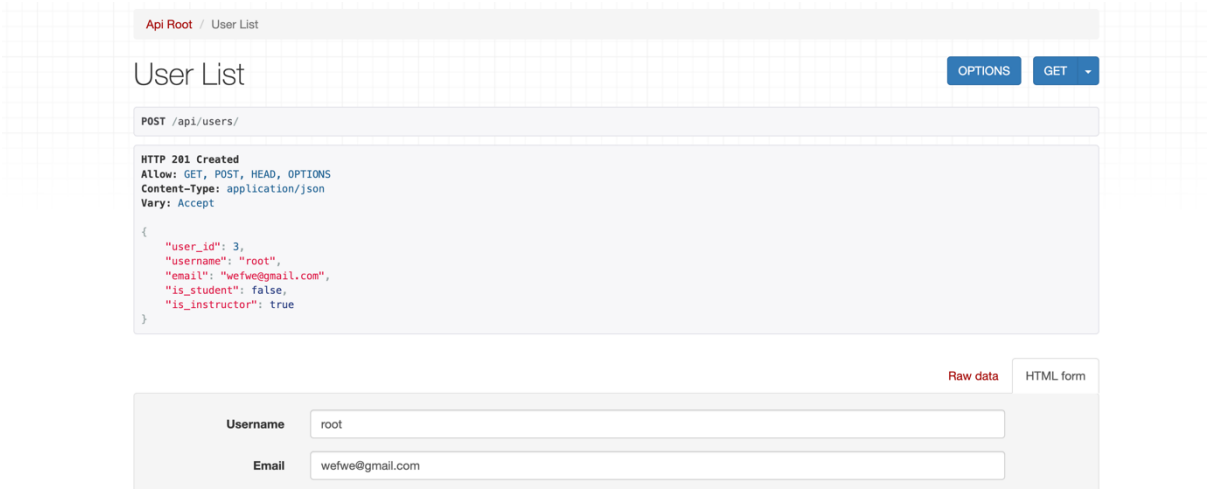
1. Django Software Foundation. (n.d.). *Django documentation*. Retrieved from https://docs.djangoproject.com/en/stable/
2. Docker, Inc. (n.d.). *Docker documentation*. Retrieved from https://docs.docker.com/
3. The PostgreSQL Global Development Group. (n.d.). *PostgreSQL documentation*. Retrieved from https://www.postgresql.org/docs/
4. Docker Hub. (n.d.). *PostgreSQL*. Retrieved from https://hub.docker.com/_/postgres
5. Docker, Inc. (2021). *Networking in Docker*. Retrieved from https://docs.docker.com/network/
6. GitHub. (n.d.). *AA19BD Repository*. Retrieved from https://github.com/AA19BD
7. Django REST Framework Documentation: https://www.django-rest-framework.org/
8. DRF SimpleJWT Documentation: https://django-rest-framework-simplejwt.readthedocs.io/
9. DRF Yasg Documentation: https://drf-yasg.readthedocs.io/en/stable/\

Appendices




Django REST framework
root
Api Root / User Progress List
User Progress List
OPTIONS
GET
GET /api/userprogress/
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
[
{
"progress_id": 1,
"user": 1,
"course": 5,
"completed_lessons": 1,
"quiz_scores": 20.0
}
]
Raw data
HTML form
User
abylay


Django REST framework
root
Api Root
Api Root
OPTIONS
GET
The default basic root view for DefaultRouter
GET /api/
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
{
"users": "http://localhost:8000/api/users/",
"categories": "http://localhost:8000/api/categories/",
"courses": "http://localhost:8000/api/courses/",
"enrollments": "http://localhost:8000/api/enrollments/",
"lessons": "http://localhost:8000/api/lessons/",
"reviews": "http://localhost:8000/api/reviews/",
"payments": "http://localhost:8000/api/payments/",
"quizzes": "http://localhost:8000/api/quizzes/",
"quizquestions": "http://localhost:8000/api/quizquestions/",
"userprogress": "http://localhost:8000/api/userprogress/",
"quiz-submissions": "http://localhost:8000/api/quiz-submissions/"
}


Api Root / User List
User List
OPTIONS
GET
POST /api/users/
HTTP 201 Created
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
{
"user_id": 3,
"username": "root",
"email": "wefwe@gmail.com",
"is_student": false,
"is_instructor": true
}
Raw data
HTML form
Username
root
Email
wefwe@gmail.com

Quiz Submission List

OPTIONS GET

```
GET /api/quiz-submissions/
```

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
```


```
[
    {
        "user": 1,
        "quiz": 2,
        "question": 1,
        "selected_option": "A",
        "option_a": "4",
        "option_b": "3",
        "option_c": "1",
        "option_d": "Who knows?"
    },
    {
        "user": 1,
        "quiz": 2,
        "question": 1,
        "selected_option": "A",
        "option_a": "4",
        "option_b": "3",
        "option_c": "1",
        "option_d": "Who knows?"
    },
    {
        "user": 1,
        "quiz": 2,
        "question": 2,
        "selected_option": "B",
        "option_a": "4",
        "option_b": "3",
        "option_c": "1",
        "option_d": "Who knows?"
    }
]
```

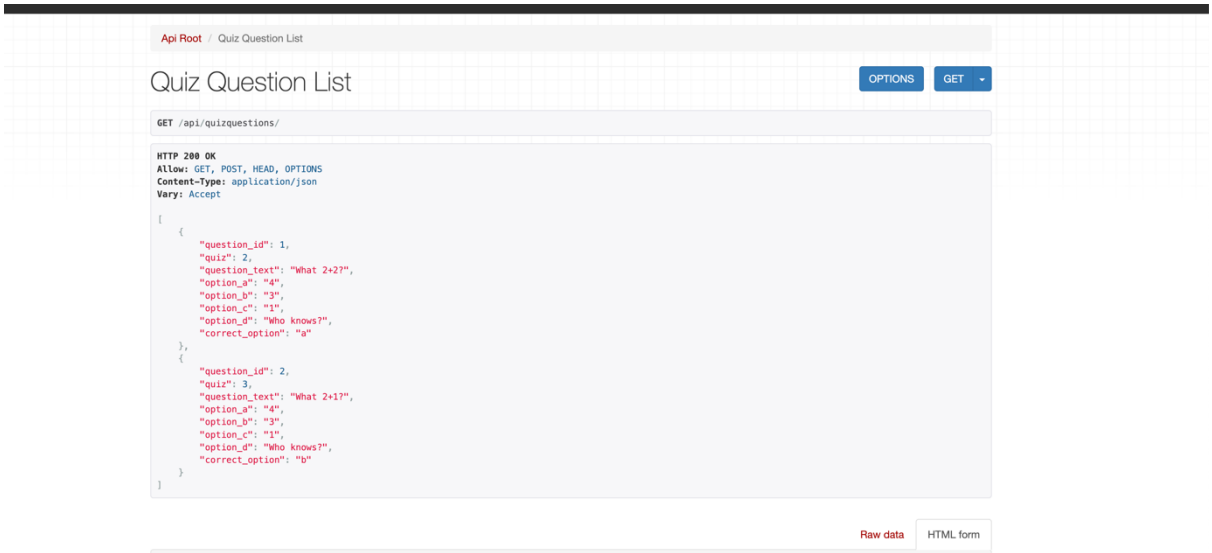

Enrollment List

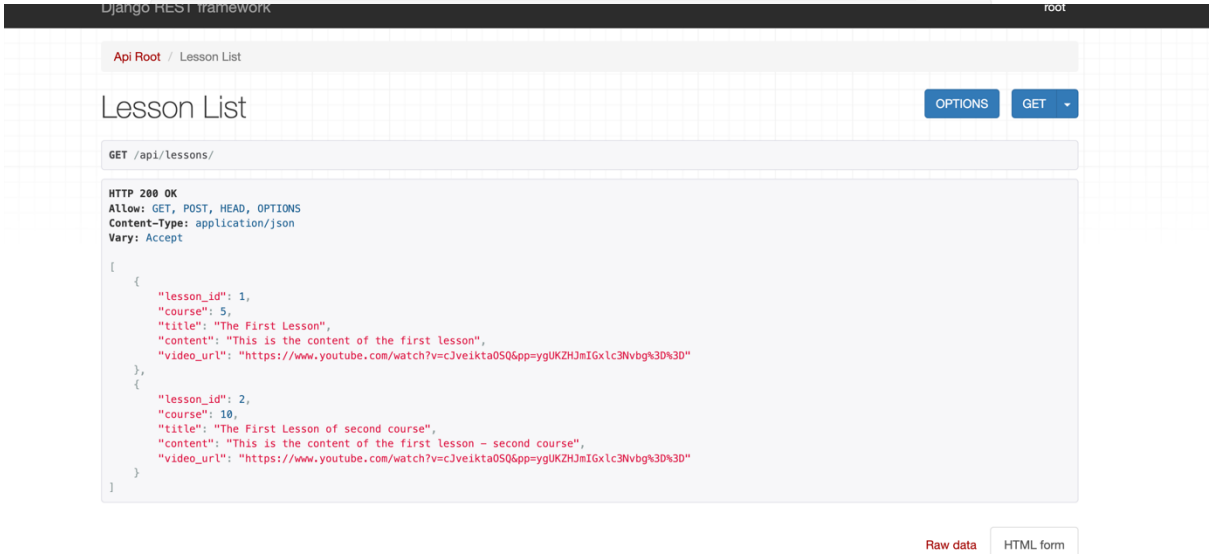
OPTIONS GET

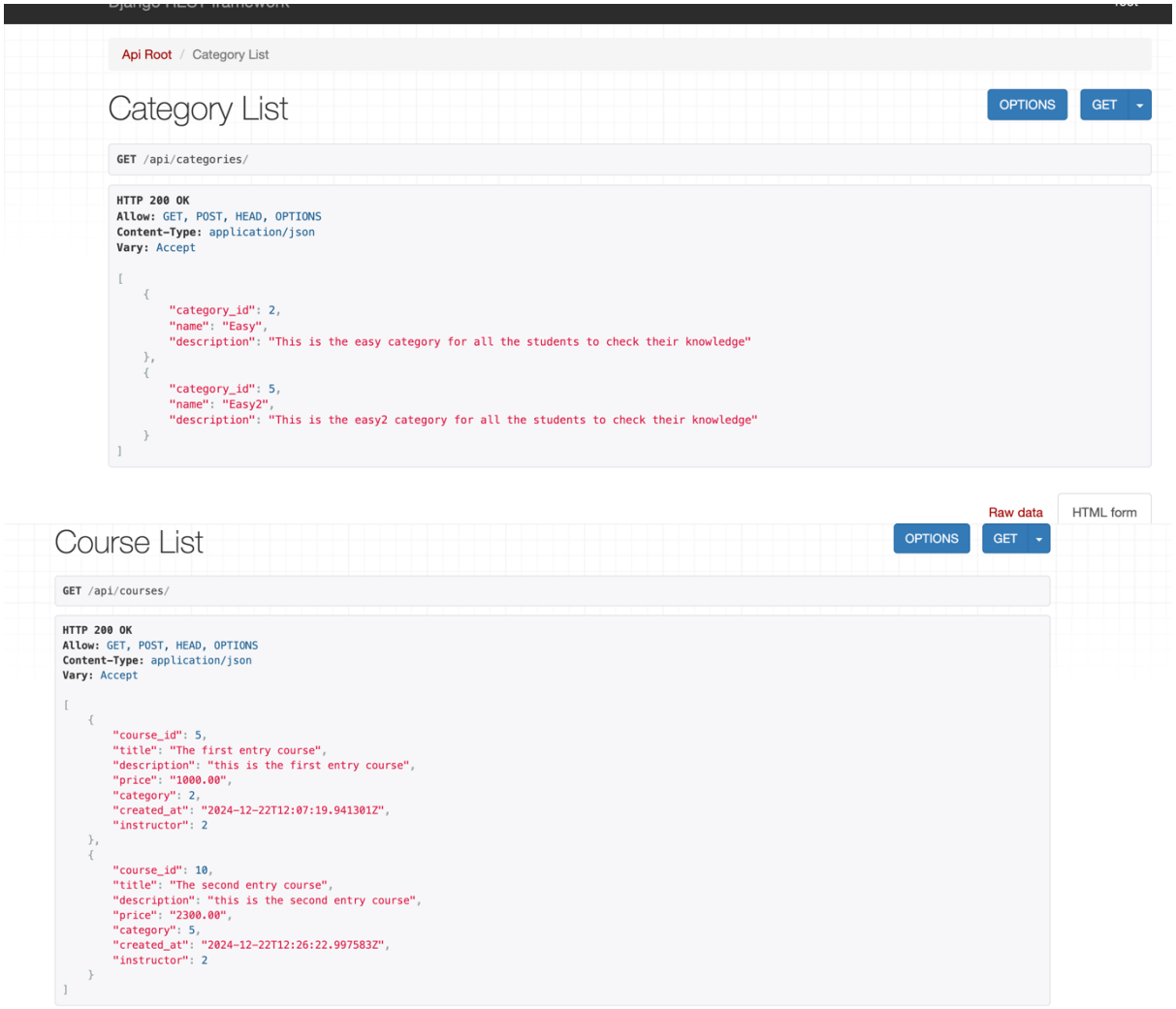
```
GET /api/enrollments/
```

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
```


```
[
    {
        "enrollment_id": 4,
        "user": 1,
        "course": 5,
        "enrollment_date": "2024-12-22T12:50:42.956153Z",
        "status": "enrolled"
    },
    {
        "enrollment_id": 5,
        "user": 1,
        "course": 5,
        "enrollment_date": "2024-12-22T12:50:51.700254Z",
        "status": "completed"
    },
    {
        "enrollment_id": 6,
        "user": 1,
        "course": 10,
        "enrollment_date": "2024-12-22T12:51:01.731034Z",
        "status": "enrolled"
    },
    {
        "enrollment_id": 7,
        "user": 2,
        "course": 5,
        "enrollment_date": "2024-12-22T12:51:10.422186Z",
        "status": "enrolled"
    },
    {
        "enrollment_id": 8,
        "user": 2,
        "course": 5,
        "enrollment_date": "2024-12-22T12:51:16.036448Z",
        "status": "completed"
    }
]
```


Api Root / Quiz Question List
Quiz Question List
OPTIONS
GET
GET /api/quizquestions/
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
[
{
"question_id": 1,
"quiz": 2,
"question_text": "What 2+2?",
"option_a": "4",
"option_b": "3",
"option_c": "1",
"option_d": "Who knows?",
"correct_option": "a"
},
{
"question_id": 2,
"quiz": 3,
"question_text": "What 2+1?",
"option_a": "4",
"option_b": "3",
"option_c": "1",
"option_d": "Who knows?",
"correct_option": "b"
}
]
Raw data
HTML form


Django REST framework
root
Api Root / Lesson List
Lesson List
OPTIONS
GET
GET /api/lessons/
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
[
{
"lesson_id": 1,
"course": 5,
"title": "The First Lesson",
"content": "This is the content of the first lesson",
"video_url": "https://www.youtube.com/watch?v=cJveikta0SQ&pp=ygUKZHJmIGxlc3Nvbg%3D%3D"
},
{
"lesson_id": 2,
"course": 10,
"title": "The First Lesson of second course",
"content": "This is the content of the first lesson - second course",
"video_url": "https://www.youtube.com/watch?v=cJveikta0SQ&pp=ygUKZHJmIGxlc3Nvbg%3D%3D"
}
]
Raw data
HTML form


Api Root / Category List
Category List
OPTIONS
GET
GET /api/categories/
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
[
{
"category_id": 2,
"name": "Easy",
"description": "This is the easy category for all the students to check their knowledge"
},
{
"category_id": 5,
"name": "Easy2",
"description": "This is the easy2 category for all the students to check their knowledge"
}
]
Raw data
HTML form
Course List
OPTIONS
GET
GET /api/courses/
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
[
{
"course_id": 5,
"title": "The first entry course",
"description": "this is the first entry course",
"price": "1000.00",
"category": 2,
"created_at": "2024-12-22T12:07:19.941301Z",
"instructor": 2
},
{
"course_id": 10,
"title": "The second entry course",
"description": "this is the second entry course",
"price": "2300.00",
"category": 5,
"created_at": "2024-12-22T12:26:22.997583Z",
"instructor": 2
}
]

```
swagger: '2.0'
info:
  title: E-Learning Platform Development API
  description: API for E-Learning Platform Development
  contact:
    email: contact@myblog.local
  version: v1
host: localhost:8000
schemes:
- http
basePath: /api
consumes:
- application/json
produces:
- application/json
securityDefinitions:
  Basic:
    type: basic
security:
- Basic: []
paths:
  /categories/:
    get:
      operationId: categories_list
      description: ''
      parameters: []
      responses:
        '200':
          description: ''
          schema:
            type: array
            items:
              $ref: '#/definitions/Category'
      tags:
      - categories
    post:
      operationId: categories_create
      description: ''
      parameters:
      - name: data
        in: body
        required: true
        schema:
          $ref: '#/definitions/Category'
      responses:
        '201':
          description: ''
          schema:
            $ref: '#/definitions/Category'
      tags:
      - categories
    parameters: []
  /categories/{category_id}/:
    get:
      operationId: categories_read
      description: ''
      parameters: []
      responses:
        '200':
          description: ''
          schema:
            $ref: '#/definitions/Category'
      tags:
      - categories
    put:
      operationId: categories_update
      description: ''
      parameters:
      - name: data
```

```python
from django.contrib import admin
from django.urls import path, include
from rest_framework import permissions
from drf_yasg.views import get_schema_view
from drf_yasg import openapi

schema_view = get_schema_view(
    openapi.Info(
        title="E-Learning Platform Development API",
        default_version='v1',
        description="API for E-Learning Platform Development",
        contact=openapi.Contact(email="contact@myblog.local"),
    ),
    public=True,
    permission_classes=[permissions.AllowAny,],
)

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('courses.urls')),
    path('swagger/', schema_view.as_view(), name='swagger-docs'),
]

```