Web development

 E-Learning Platform Development with Django and Docker

Kazakh British Technical University

Bitanov Assanali (23MD0392)

20.12.2024

# Executive summary

This project involved building the backend for an e-learning platform using Django Rest Framework (DRF) and deploying it with Docker. The goal was to create a system that allows users to manage courses, lessons, and quizzes through secure and scalable API endpoints. By combining Django's powerful tools with Docker's containerization capabilities, the project delivered a robust backend system suitable for modern e-learning needs.

Django models were used to define the structure of data, covering entities such as users, courses, lessons, and quizzes. These models ensured the data was well-organized and easy to retrieve, with clear relationships between the tables. Serializers in DRF were implemented to convert data between Python objects and JSON, enabling smooth communication between the API and external clients. Views built with DRF’s ModelViewSet handled all CRUD operations, simplifying the development process while maintaining flexibility. Permissions were added to enforce secure access and define what actions users could perform.

The API was made accessible and well-structured using URL routing, with endpoints designed to handle both current and future requirements. Unit tests were created to verify the functionality of the API, ensuring it behaved reliably under different scenarios and maintained data integrity.

For deployment, Docker was used to package the application, providing consistency across different environments. A Dockerfile and docker-compose.yml file were created to manage the backend and PostgreSQL database services, making the deployment process efficient and seamless.

Overall, this project demonstrated practical knowledge of developing RESTful APIs, managing data relationships, and deploying applications with Docker. It showcased the ability to build a scalable and secure backend system while preparing it for future enhancements or integrations.

Table of Contents

[Executive summary 2](#_Toc184736828)

[Introduction 4](#_Toc184736829)

[System Architecture 5](#_Toc184736830)

[Table Descriptions 6](#_Toc184736831)

[Intro to Containerization: Docker 11](#_Toc184736832)

[Dockerfile 12](#_Toc184736833)

[Docker-Compose 12](#_Toc184736834)

[Views 13](#_Toc184736835)

[Templates 14](#_Toc184736836)

[Django Rest Framework (DRF) 16](#_Toc184736837)

[Results 17](#_Toc184736838)

[Conclusion 21](#_Toc184736839)

[References 22](#_Toc184736840)

# Introduction

In this project, I built an e-learning platform backend using Django Rest Framework (DRF) and deployed it with Docker. The goal was to create a robust system where users can manage courses, lessons, and quizzes through API endpoints while ensuring the platform is scalable and easy to maintain.

First, I designed models in Django to structure the data for users, courses, lessons, and related entities. The models defined fields like titles, descriptions, categories, and relationships between different tables, making the data organized and easy to manage.

Next, I created serializers using DRF to handle the conversion between Python objects and JSON. This made it possible to exchange data between the backend and external clients in a standard format, ensuring the API was user-friendly and efficient.

I then implemented views using DRF’s ModelViewSet to handle CRUD operations for the main entities, such as users and courses. Permissions were added to control actions, ensuring that the API could be securely accessed and used.

To ensure all endpoints were accessible and logically organized, I set up URL routing with DRF’s DefaultRouter. This simplified the process of managing the API structure and ensured consistent versioning for future updates.

For deployment, I used Docker to containerize the application. I wrote a Dockerfile to define the setup for the backend and a docker-compose.yml file to orchestrate the services, including the web app and PostgreSQL database. This approach made the deployment process consistent and efficient across environments.

This project gave me practical experience in building RESTful APIs, managing complex data relationships, and deploying scalable backend systems with Docker. It also improved my understanding of how to design and implement secure and maintainable applications.

# System Architecture

The e-learning platform is built using a modular backend architecture. The backend is developed with Django and Django Rest Framework (DRF) to handle the business logic, database interactions, and API endpoints. Docker is used to containerize the system for consistent deployment and easier management. The structure of the project is shown in Figure 1.

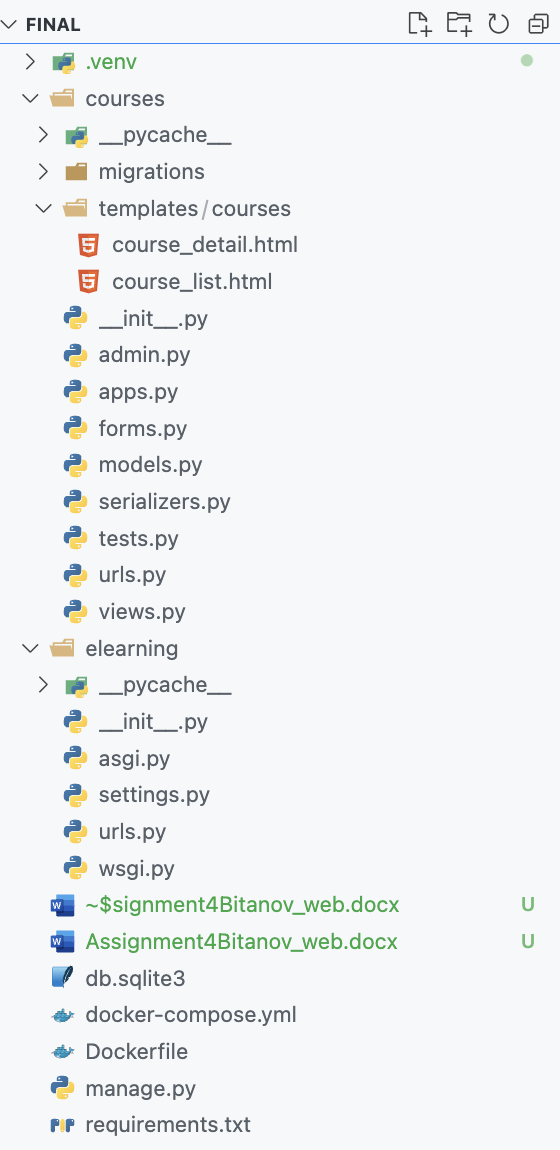


Figure 1. The structure of the project in VSCode

The Backend is written in Django. It’s perfect for almost every CRUD web application which can be created astonishingly fast [1].  The Django application is the platform's core, managing models such as Users, Courses, Lessons, and more. It also provides a web-based admin interface for managing data and uses DRF to expose API endpoints for potential integrations.

A PostgreSQL database stores all the platform’s data, including user accounts, course information, and progress tracking. With PostgreSQL, we can create, store databases, and work with data using SQL queries [2]. The database runs in its own Docker container to ensure data persistence and logical separation from the application.

The system is containerized into two main containers. Web - runs the Django backend. DB - hosts the PostgreSQL database. These containers are shown in Figure 2.

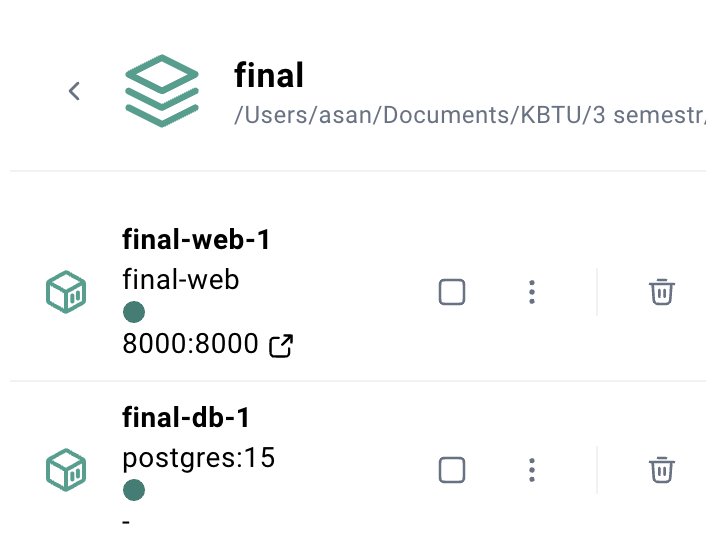


Figure 2. The running web and db containers in the Docker Desktop app

By focusing on the backend, the platform is designed to be extendable, ensuring scalability and adaptability for future features. Dockerization simplifies deployment, allowing the system to run reliably across different environments.

# Table Descriptions

The e-learning platform database is structured to efficiently store and manage information about users, courses, categories, lessons, enrollments, reviews, payments, quizzes, quiz questions, and user progress. In order to create these tables we create models.py and define the classes there. The user class is shown in Figure 3.

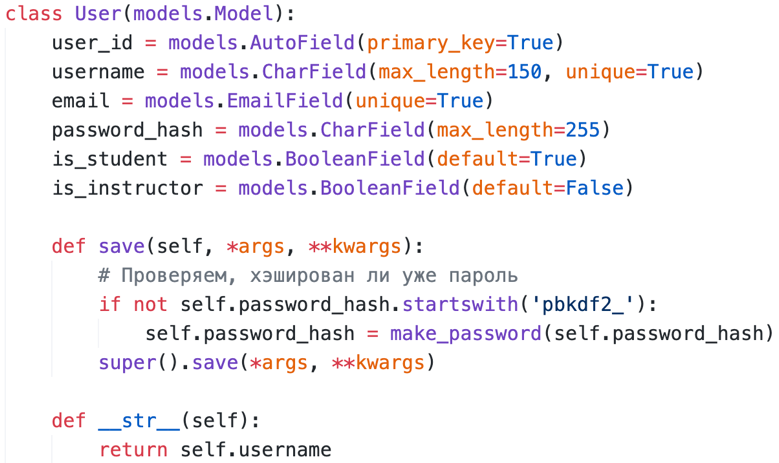


Figure 3. The User class

The purpose of this class is to store information about all users of the platform, including students and instructors. The user class contains these fields: user\_id is a unique identifier for each user, username is the username for the user,   
email is the email address for the user, password\_hash is the hashed version of the user's password for security, is\_student indicates if the user is a student, and is\_instructor indicates if the user is an instructor. Each user must have either is\_student or is\_instructor set to True.

The next class is the Category class. The purpose of this class is to define categories for grouping courses. The category class is shown in Figure 4.

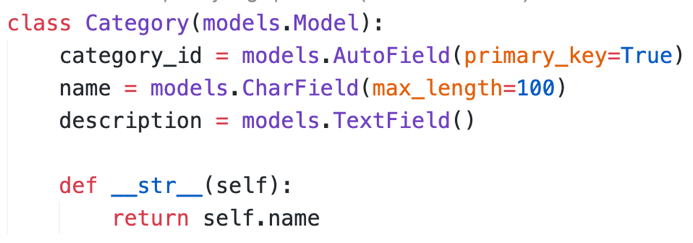


Figure 4. The Category class

The Categories class contains these fields: category\_id is a unique identifier for each category, name is the name of the category, and description provides additional information about the category.

The next class is the Lesson class. The purpose of this class is to represent lessons within a course. The lesson class is shown in Figure 5.

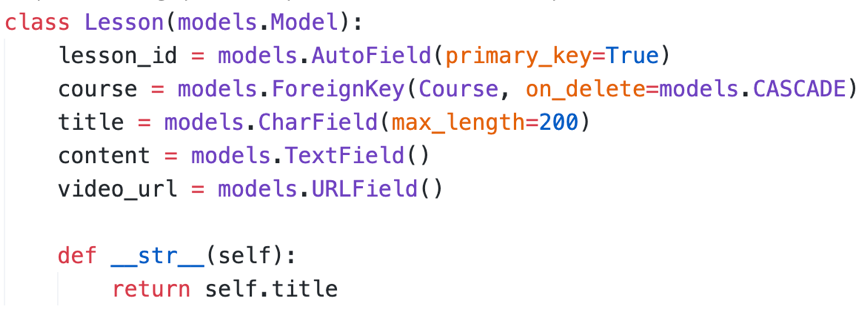


Figure 5. The Lesson class

The Lessons class contains these fields: lesson\_id is a unique identifier for each lesson, course links to the Courses class, title is the title of the lesson, content is the textual content of the lesson, and video\_url is the URL for the lesson's video.

The next class is the Course class. The purpose of this class is to store information about the courses offered on the platform. The course class is shown in Figure 6.

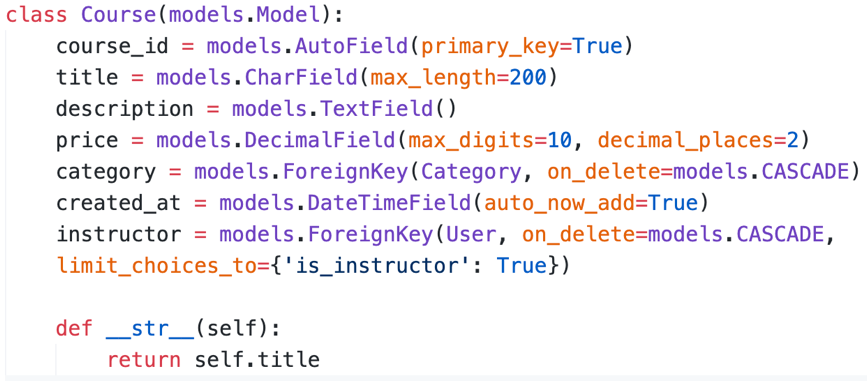


Figure 6. The Course class

The Courses class contains these fields: course\_id is a unique identifier for each course, title is the title of the course, description provides details about the course, price is the cost of the course, category links to the Categories class, created\_at is a timestamp indicating when the course was created, and instructor links to the Users class and must reference a user with is\_instructor=True.

The next class is the Enrollment class. The purpose of this class is to track which users are enrolled in which courses. The Enrollment class is shown in Figure 7.

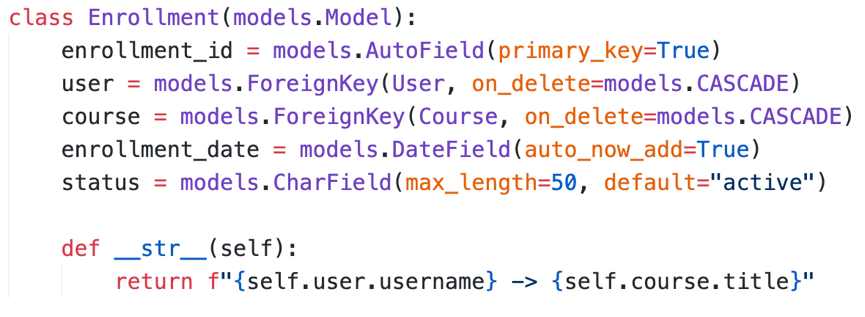


Figure 7. The Enrollment class

The Enrollments class contains these fields: enrollment\_id is a unique identifier for each enrollment, user links to the Users class, course links to the Courses class, enrollment\_date is the date of enrollment, and status indicates the enrollment's status.

The next class is the Review class. The purpose of this class is to store user reviews for courses. The Review class is shown in Figure 8.



Figure 8. The Review class

The Reviews class contains these fields: review\_id is a unique identifier for each review, course links to the Courses class, user links to the Users class, rating is the numeric rating given to the course, comment is the user's text review, and created\_at is a timestamp indicating when the review was created.

The next class is the Payment class. The purpose of this class is to track payments made by users for courses. The Payment class is shown in Figure 9.

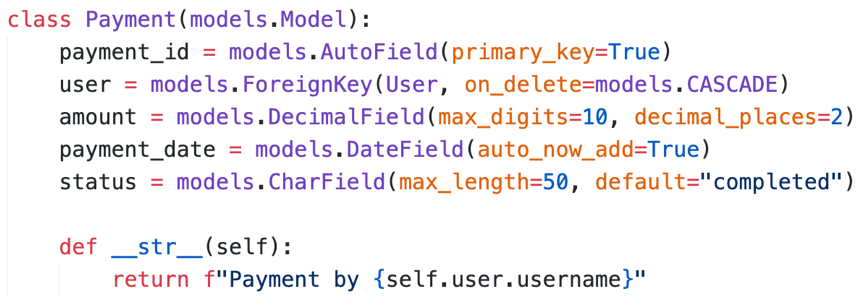


Figure 9. The Payment class

The Payments class contains these fields: payment\_id is a unique identifier for each payment, user links to the Users class, amount is the payment amount, payment\_date is the date the payment was made, and status indicates the payment's status.

The next class is the Quiz class. The purpose of this class is to represent quizzes associated with courses. The Quiz class is shown in Figure 10.

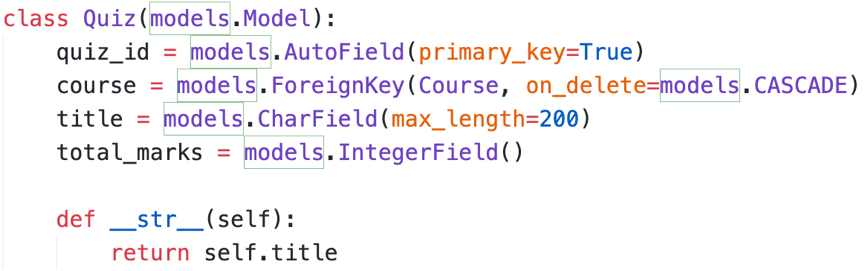


Figure 10. The Quiz class

The Quizzes class contains these fields: quiz\_id is a unique identifier for each quiz, course links to the Courses class, title is the title of the quiz, and total\_marks indicates the total marks for the quiz.

The next class is the QuizQuestion class. The purpose of this class is to store questions for quizzes. The QuizQuestion class is shown in Figure 11.

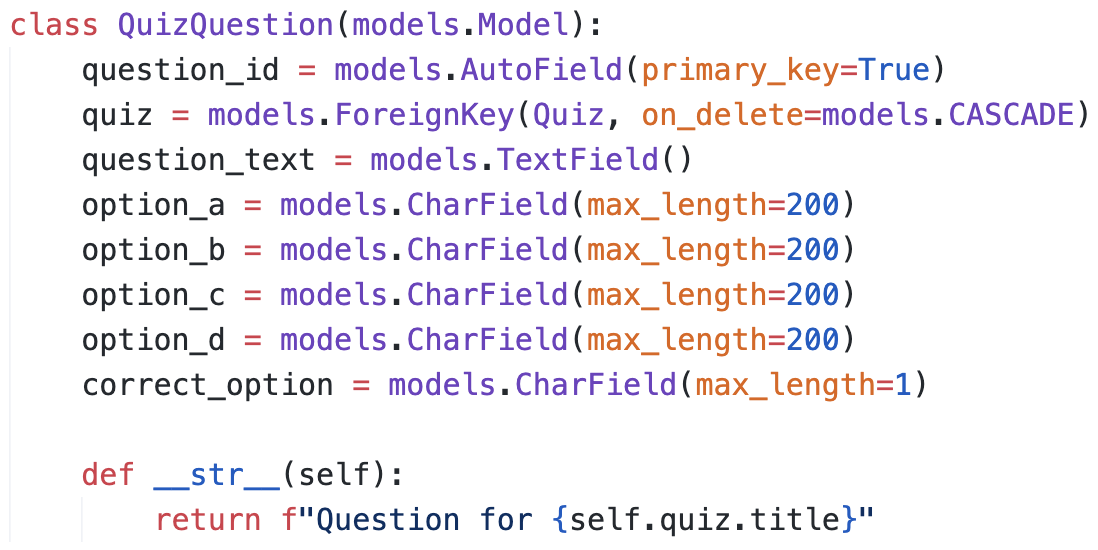


Figure 11. The QuizQuestion class

The Quiz Questions class contains these fields: question\_id is a unique identifier for each question, quiz links to the Quizzes class, question\_text is the text of the question, option\_a, option\_b, option\_c, and option\_d are the available options, and correct\_option indicates the correct answer option.

The next class is the User Progress class. The purpose of this class is to track user progress in courses. The User Progress class is shown in Figure 12.

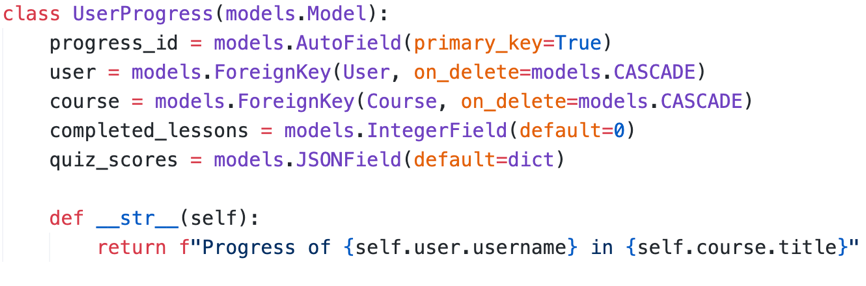


Figure 12. The User Progress class

The User Progress class contains these fields: progress\_id is a unique identifier for each progress record, user links to the Users class, course links to the Courses class, completed\_lessons indicates the number of lessons completed by the user, and quiz\_scores is a JSON field storing the user's quiz scores.

# Intro to Containerization: Docker

Containerization is a modern software deployment method that involves bundling an application along with its dependencies, libraries, and configuration files into a single unit called a container. A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. The list of containers is shown in Figure 13.

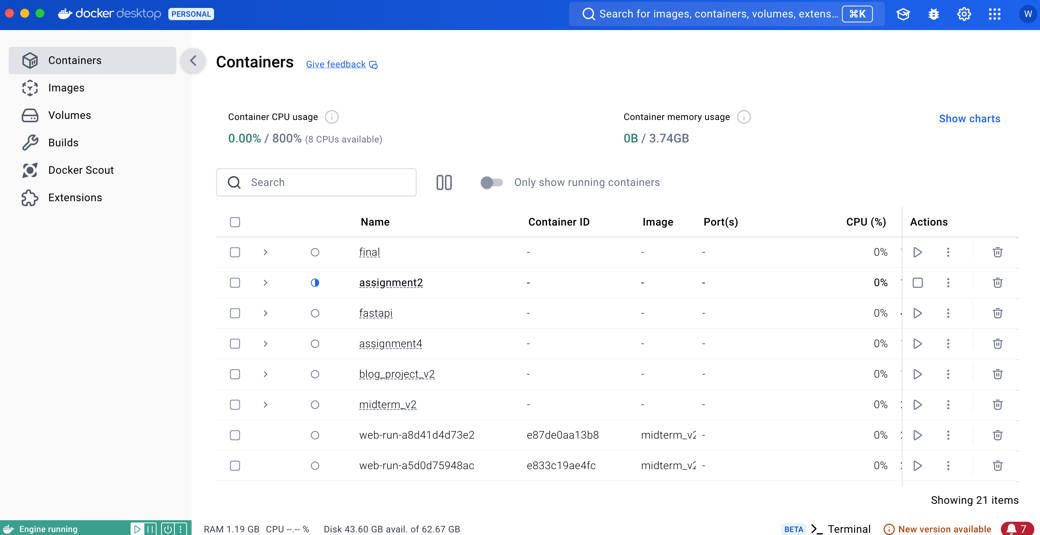


Figure 13. The list of my containers in Docker Desktop

A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings [3]. It has several benefits, such as portability, efficiency, scalability, isolation, faster deployment, version control, and improved collaboration.

# Dockerfile

Docker builds images by reading the instructions from a Dockerfile. A Dockerfile is a text file containing instructions for building your source code. The Dockerfile instruction syntax is defined by the specification reference in the Dockerfile reference [4]. The Dockerfile of the project is shown in Figure 14.



Figure 14. The Dockerfile of the project

The Dockerfile starts with FROM python:3.10-slim, which uses a lightweight Python 3.10 image as the base for a minimal environment. The WORKDIR /app command sets the working directory in the container to /app, where application files will be stored. The COPY requirements.txt . command adds the requirements file to the container, and RUN pip install --no-cache-dir -r requirements.txt installs all Python dependencies. The COPY . . command copies all application files from the host into the container. The EXPOSE 8000 command specifies that the application will run on port 8000, making it accessible for external communication. Finally, the CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"] command starts the Django development server and binds it to all network interfaces.

# Docker-Compose

Docker Compose is a tool for defining and running multi-container applications. It is the key to unlocking a streamlined and efficient development and deployment experience. Compose simplifies the control of your entire application stack, making it easy to manage services, networks, and volumes in a single, comprehensible YAML configuration file [5]. The Docker Compose of the project is shown in Figure 15.



Figure 15. The Docker Compose of the project

Docker Compose makes it easier to manage applications with multiple containers by using a single docker-compose.yml file. In this project, it runs two services: web for the Django application and db for the PostgreSQL database. The web service is built from the Dockerfile, runs on port 8000, and mounts the project directory to allow updates. It also uses environment variables like DEBUG=True and waits for the db service to be ready. The db service uses a PostgreSQL 15 image with settings for the username, password, and database name. It also saves data in a volume called postgres\_data to keep it even if the container is restarted. Docker Compose simplifies starting, stopping, and connecting these services.

# Views

In Django, views play a crucial role in handling incoming requests and generating responses. They serve as the bridge between the URL patterns and the underlying logic that processes the data before rendering it to the user [6]. There are two main types of views used in this project: class-based views and ViewSets. The views.py is shown in Figure 16.

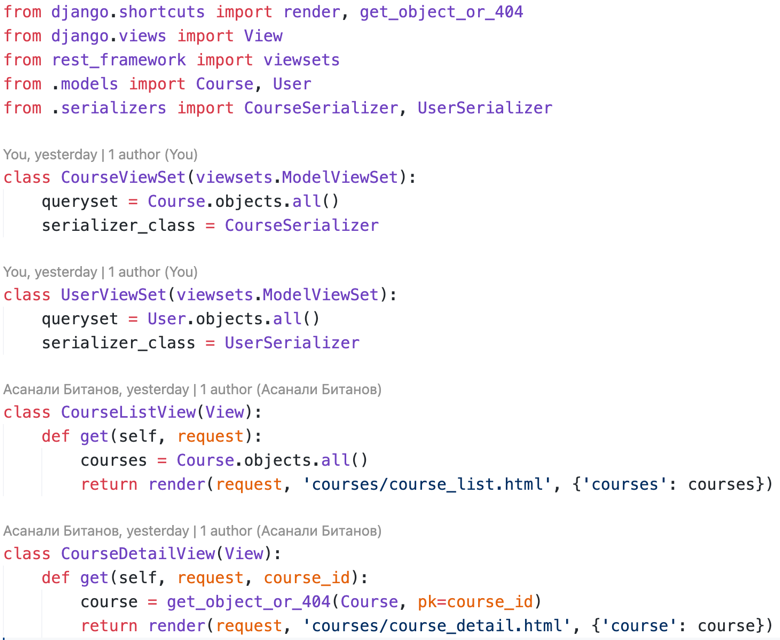


Figure 15. The views.py

Class-Based Views (CBVs) provide reusable and modular logic for handling HTTP requests. In this project, CourseListView displays a list of all courses by retrieving them from the database and rendering the course\_list.html template, while CourseDetailView fetches details of a specific course using its course\_id and renders the course\_detail.html template. ViewSets, used with Django Rest Framework (DRF), simplify creating API endpoints. The CourseViewSet handles CRUD operations (Create, Read, Update, Delete) for the Course model, and the UserViewSet does the same for the User model, enabling interaction through the API.

# Templates

Templates are the third and most important part of Django’s MVT Structure. A template in Django is basically written in HTML, CSS, and Javascript in a .html file. Django framework efficiently handles and generates dynamic HTML web pages that are visible to the end-user [7]. The course\_list.html template displays a list of available courses. The course\_list.html is shown in Figure 16.



Figure 16. The course\_list.html

It uses a {% for %} loop to iterate through the courses passed from the CourseListView and dynamically generates links to their details, including the title and price of each course.

The course\_detail.html template shows detailed information about a specific course, including its description, price, category, and creation date, using variables like {{ course.title }} and {{ course.description }}. The course\_detail.html is shown in Figure 17.



Figure 17. The course\_list.html

Using Django templates, the application can efficiently generate dynamic pages while separating HTML structure and backend logic.

# Django Rest Framework (DRF)

Django REST Framework (DRF) is a widely-used, full-featured API framework designed for building RESTful APIs with Django. At its core, DRF integrates with Django's core features - models, views, and URLs-- making it simple and seamless to create a RESTful API [8]. Serializers are used to convert complex data types, such as Django model instances, into Python data types that can be easily rendered into JSON, XML, or other content types [9]. The serializers.py is shown in Figure 18.

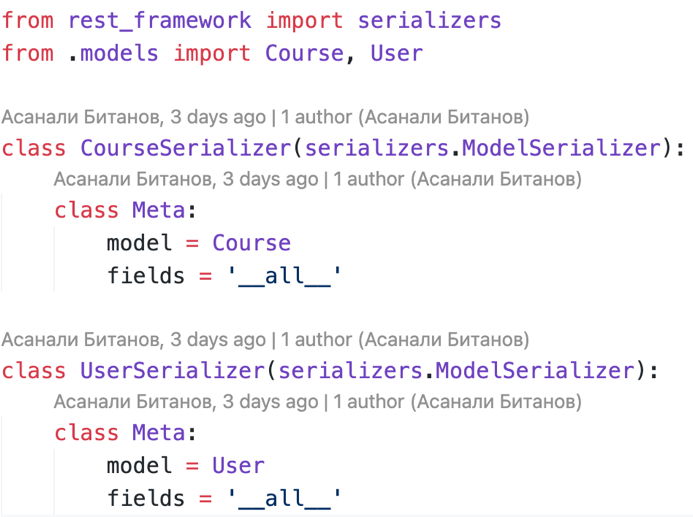


Figure 18. The serializers.py file

This file defines CourseSerializer and UserSerializer, which convert model instances into JSON format and validate incoming data for API operations. The fields = '\_\_all\_\_' ensures all model fields are included in the API responses and requests.

The views.py file defines the logic for how the API interacts with the Course and User models. The file is shown in Figure 19.

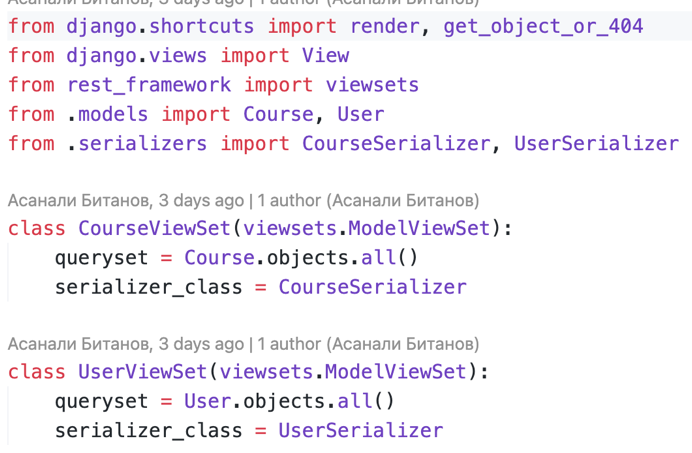


Figure 19. The views.py file

The viewsets handle CRUD operations like Create, Read, Update, Delete for their models. Each viewset takes the appropriate data (queryset = Course.objects.all() for courses or queryset = User.objects.all() for users) and uses the proper serializer to convert data into JSON format for API responses.

The urls.py file is essential for connecting API requests to the appropriate views. The file is shown in Figure 20.



Figure 20. The urls.py file

Using DRF's DefaultRouter, we automate the creation of RESTful API endpoints for the viewsets. This ensures that endpoints like /api/courses/ and /api/users/ are automatically set up with routes for list, detail, create, update, and delete operations.

# Results

To illustrate the results of the project I used Django Admin panel and the API routes. Both are running by the Docker containers. All of the tables are presented in the panel. The User table is shown in Figure 21.

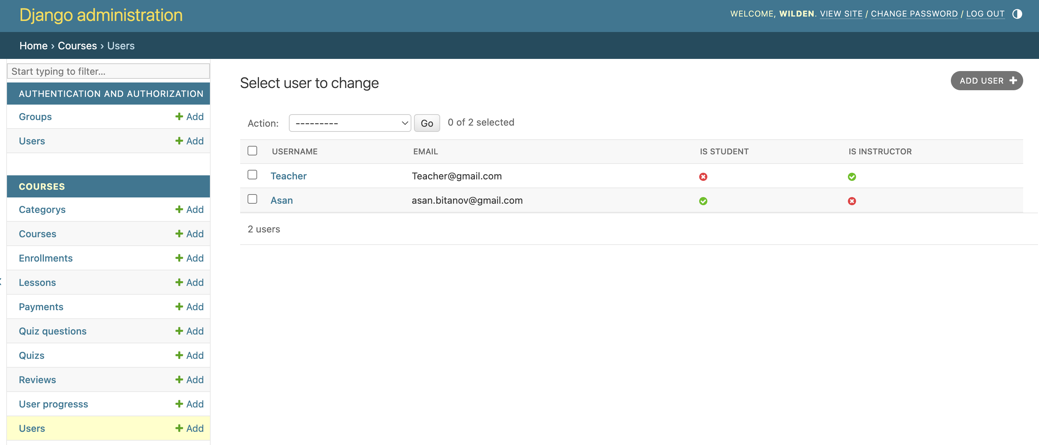


Figure 20. The User table page in the Django panel

Here we can see two users: Asan and Teacher. On this page, we can see all the data about the users. To create a user, we can fill out the form by clicking the ADD USER button. The filled form for the Asan user is shown in Figure 21.

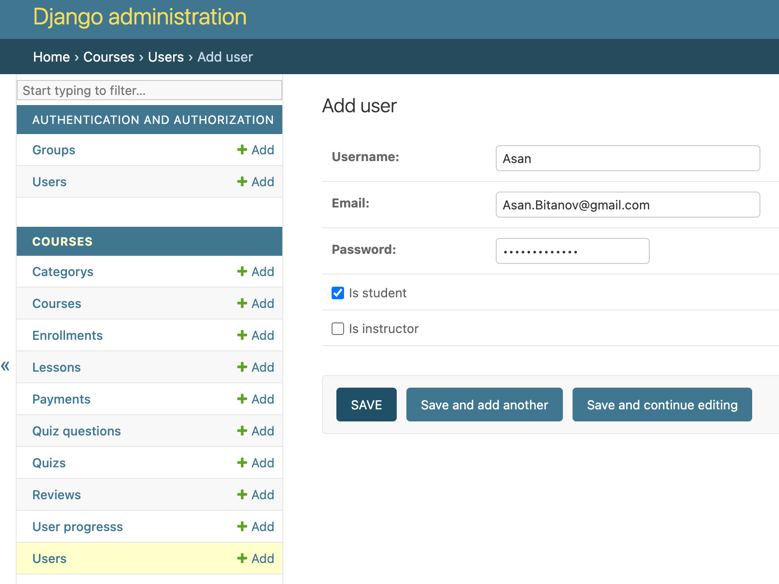


Figure 21. The Add user page

Next, the Course table. This table represents data about the courses on the e-learning platform. Each course is a part of a corresponding category. The Django course is shown in Figure 22.

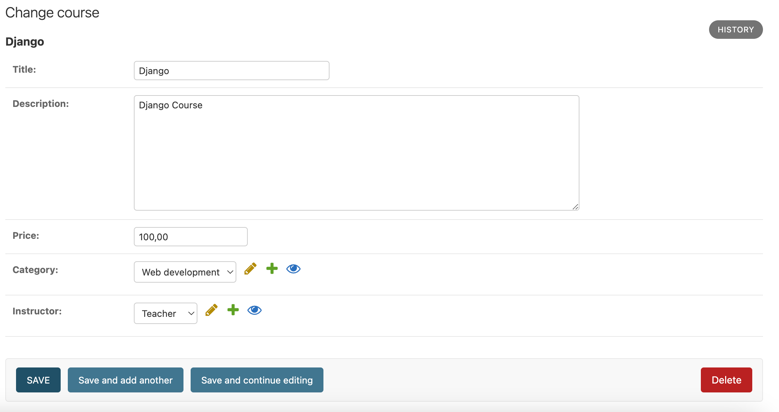


Figure 22. The Django Course on the Admin panel

The table has title, description, price, category, and instructor column. As shown in Figure 22, the course is a part of the Web Development category. The data about this category is in the Category table. Also, the instructor of this course is the Teacher user. The category and instructor columns are related to the corresponding tables.

Next, the Lesson table. This table represents data about the lessons on the e-learning platform. Each lesson is a part of a corresponding course. The Introduction lesson of the Django course is shown in Figure 23.

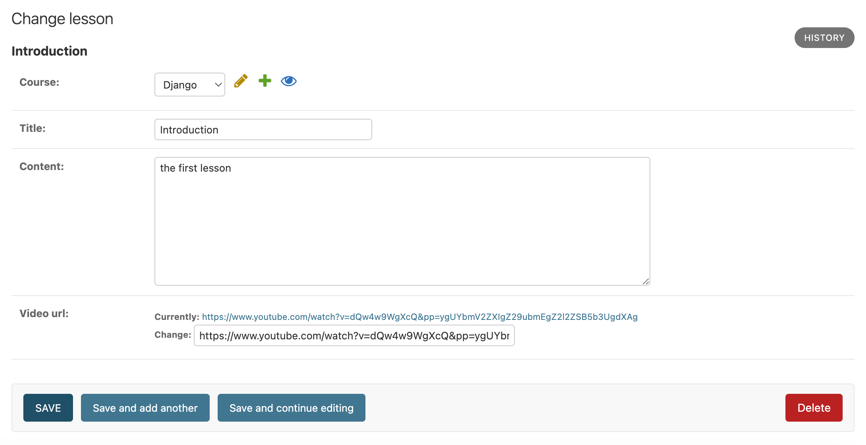


Figure 23. The Introduction lesson of the Django course

The table has course, title, content, and viedeo\_url columns. The video\_url column contains the URL of the lesson.

Next, the Quiz questions table. The table represents data about the quiz questions. Each question is a part of a quiz. The Quiz questions of the Django quiz are shown in Figure 24.

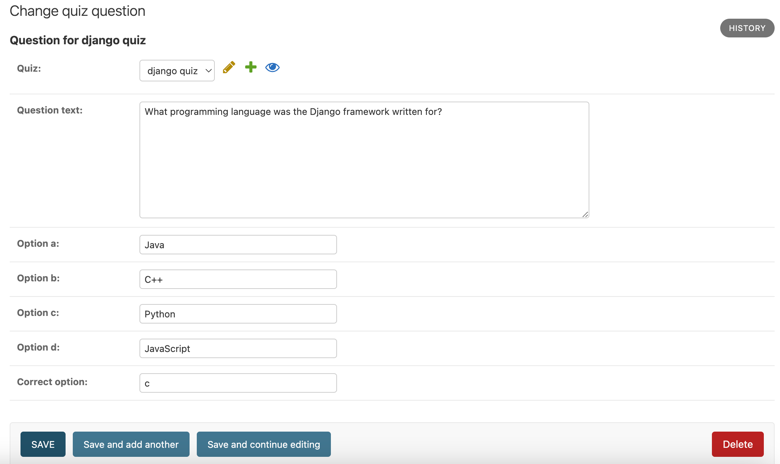


Figure 24. The Django quiz questions

The table has quiz, questions\_text, options, and correct\_option columns. The correct\_option takes a char(1), which takes the correct letter.

The Django Admin panel makes it easy to manage data in all tables, and the API routes allow working with the data programmatically. The tables, like Users and Quiz Questions, are well-organized with clear relationships, making the platform reliable and scalable. This shows that the database and backend were implemented successfully, achieving the project's goals.

# Conclusion

In this project, I created the backend for an e-learning platform using Django Rest Framework (DRF) and deployed it with Docker. The system allows users to manage courses, lessons, and quizzes through API endpoints. The main focus was to build a secure, scalable, and easy-to-maintain backend system.

I used Django models to define the data for users, courses, lessons, and quizzes. These models helped keep the data organized and easy to manage, with clear connections between the tables. Serializers were used to convert data between Python objects and JSON, so the API could communicate effectively with clients. Views, built with DRF’s ModelViewSet, handled all operations for adding, updating, and retrieving data. I also added permissions to control what actions users could perform, which made the API more secure.

I set up URL routing to make sure the API endpoints were easy to use and well-organized. This structure also prepared the system for future updates without breaking older features. I tested the API with unit tests to confirm it worked correctly and could handle different scenarios reliably.

Finally, Docker was used to containerize the application. I wrote a Dockerfile and a docker-compose.yml file to manage the backend and the PostgreSQL database together. This made deploying the app simple and consistent in any environment.

This project gave me valuable experience in building APIs, managing data relationships, and deploying apps with Docker. It was a great opportunity to improve my skills and learn practical tools for backend development.

# References

1. Django Framework Review - Pros and Cons [netguru] <https://www.netguru.com/blog/django-pros-and-cons>
2. PostgreSQL DBMS: why you should choose it for working with data and how to install it [Yandex practicum] <https://practicum.yandex.ru/blog/chto-takoe-subd-postgresql/>
3. Use containers to Build, Share and Run your applications [Docker] <https://www.docker.com/resources/what-container/>
4. Dockerfile overview [dockerdocs] <https://docs.docker.com/build/concepts/dockerfile/>
5. Docker Compose [dockerdocs] <https://docs.docker.com/compose/>
6. A Comprehensive Guide to Django Views: Understanding Types and Use Cases [medium] <https://medium.com/@satyarepala/a-comprehensive-guide-to-django-views-understanding-types-and-use-cases-4a02a078ced>
7. Django Templates [geeksforgeeks] <https://www.geeksforgeeks.org/django-templates/>
8. Django REST Framework Basics [testdriven] <https://testdriven.io/blog/drf-basics/>