KAZAKH KBTU BRITISH
TECHNICAL
UNIVERSITY

Final

E-Learning Platform Development with Django and Docker.

Prepared by Azimkhanov Bauyrzhan

Almaty, 22.12.2024

Executive summary

This project aimed to create an e-learning platform using Django and Docker. The main goal was to build a strong and flexible system for online learning. This project is important because it shows how to use modern web technologies to make education more accessible and interactive.

During the project, we used key Django features like models, views, and templates, and also applied Docker for containerization. The project included setting up the database, creating user and course management systems, and ensuring everything worked smoothly together.

Major findings from the project include successful integration of the Django framework and Docker for deployment. The e-learning platform now has a stable structure and can handle multiple users, courses, and interactions effectively.

We recommend further enhancements, such as improving the user interface and adding more interactive features. Overall, this project highlights the power of combining Django and Docker to build efficient and scalable web applications.

Table of contents

Executive summary 2
Table of contents 3
Introduction 4
System architecture 5
Table descriptions 6
Intro to containerization: Docker 7
Dockerfile 8
Docker-Compose 9
Docker networking and volumes 11
Django 12
Models 13
Views 14
Templates 16
Django Rest Framework (DRF) 17
Frontend integration 18
Challenges and solutions 19
Conclusion 20
References 21

Introduction

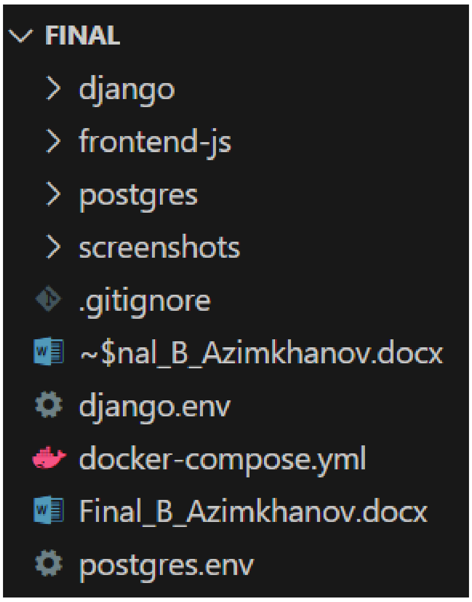
In the world of web development, Docker and Django are two very important technologies. Docker is a tool that helps developers create, deploy, and run applications in containers. Containers make it easier to manage software, because they package everything the application needs to run. Django, on the other hand, is a high-level web framework that encourages fast development and clean, pragmatic design. It helps developers build secure and maintainable websites.

The main goal of this project is to develop a solid e-learning platform. By using Docker and Django, we aim to create a scalable and efficient system for online education. The platform will allow users to enroll in courses, view lessons, take quizzes, and track their progress.

This project includes setting up a new Django project, creating the necessary database tables, and implementing the main features of the e-learning platform. We will also use Docker to containerize the application, making it easier to deploy and manage. However, this project does not cover advanced topics such as real-time interactions or machine learning integration.

System architecture

Initialize the project exactly as we did in all previous assignments on this subject.



```
FINAL
  django
  frontend-js
  postgres
  screenshots
  .gitignore
  ~$nal_B_Azimkhanov.docx
  django.env
  docker-compose.yml
  Final_B_Azimkhanov.docx
  postgres.env
```

Figure 1. Project structure.

The application consists of 3 parts: frontend, backend and database. JS, Python Django and PostgreSQL were used for these purposes respectively. The database was in its pure form without any intervention from our side. Almost all technologies from previous assessments were used in the backend: DRF, Swagger, CBV, static files, templates and others. For the frontend, pure JavaScript was used without any frameworks due to the complexity due to the lack of time to study VueJS.

The interaction of components and other things will be described in the relevant sections later. This project used a classic client-server architecture.

Table descriptions

The following table describes the data and fields of tables from the PostgreSQL database:

Table name	Description
Users	Stores user information. Fields include user_id, username, email, password_hash, is_student, is_instructor. Each user can be a student or an instructor.
Courses	Contains course details. Fields include course_id, title, description, price, category, created_at, instructor_id. Each course is created by an instructor and can belong to a category.
Enrollments	Tracks course enrollments. Fields include enrollment_id, user_id, course_id, enrollment_date, status. Users can enroll in multiple courses.
Lessons	Holds lesson content. Fields include lesson_id, course_id, title, content, video_url. Each lesson belongs to a specific course.
Reviews	Records user reviews for courses. Fields include review_id, course_id, user_id, rating, comment, created_at. Each review is linked to a course and a user.
Categories	Lists course categories. Fields include category_id, name, description. Each course can belong to one category.
Payments	Manages payment information. Fields include payment_id, user_id, amount, payment_date, status. Users can make payments for courses.
Quizzes	Contains quiz details. Fields include quiz_id, course_id, title, total_marks. Each quiz is linked to a course.
QuizQuestions	Stores quiz questions. Fields include question_id, quiz_id, question_text, option_a, option_b, option_c, option_d, correct_option. Each question belongs to a quiz.
UserProgress	Tracks user progress. Fields include progress_id, user_id, course_id, completed_lessons, quiz_scores. Each record shows a user's progress in a course.

Table 1. Table descriptions

Intro to containerization: Docker

Containerization is a method used to package software so that it can run smoothly in different environments. Docker is one of the most popular tools for containerization. It helps developers create, deploy, and manage applications in containers.

Containers are like lightweight virtual machines. They include everything needed to run an application, such as the code, libraries, and system tools. This makes it easier to move applications between different environments, like from a developer's laptop to a cloud server.

Using Docker has many benefits. First, it makes sure the application runs the same way everywhere. This reduces problems related to different environments. Second, Docker containers are easy to share and deploy. Developers can quickly set up and tear down environments, which speeds up the development process. Third, Docker helps with resource efficiency. Containers use less memory and storage compared to traditional virtual machines.

Overall, Docker simplifies the process of developing, testing, and deploying applications. It ensures consistency and efficiency, making it a valuable tool for modern software development.

Dockerfile

A Dockerfile is a text document that contains all the commands needed to assemble a Docker image. The Dockerfile for a Django application includes instructions to set up the environment and run the application.

In our Django Dockerfile, we start with the Python 3.11 image. We then update the package list and install the PostgreSQL client. After this, we set the working directory and copy the requirements.txt file, which lists all the Python dependencies. The pip install -r requirements.txt command installs these dependencies. Finally, we copy the application code, expose port 8000, and set the command to run the Django development server.

For the frontend, we use another Dockerfile. This one starts with the Nginx image. We copy the frontend code to the Nginx web server directory and expose port 80.

Using Dockerfiles helps to automate the building of application images. This makes deployment consistent and repeatable, ensuring that the application runs the same way on different machines.

```
FROM nginx:1.26.2-alpine3.20-perl
COPY . /usr/share/nginx/html
EXPOSE 80
```

Figure 2. frontent-js/Dockerfile

```
FROM python:3.11

RUN apt-get update \
    && apt-get install -y --no-install-recommends \
        postgresql-client \
    && rm -rf /var/lib/apt/lists/*
WORKDIR /usr/src/app
COPY requirements.txt ./

RUN pip install -r requirements.txt
COPY . .

EXPOSE 8000
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

Figure 3. django/Dockerfile

Docker-Compose

Docker Compose is a tool that helps manage applications with multiple containers. It allows developers to define and run multi-container Docker applications using a simple YAML file called docker-compose.yml.

In our project, the docker-compose.yml file is used to manage three main services: the frontend, the Django application, and the PostgreSQL database. Each service has its own section in the file, which includes details such as the build context, ports, environment variables, and dependencies.

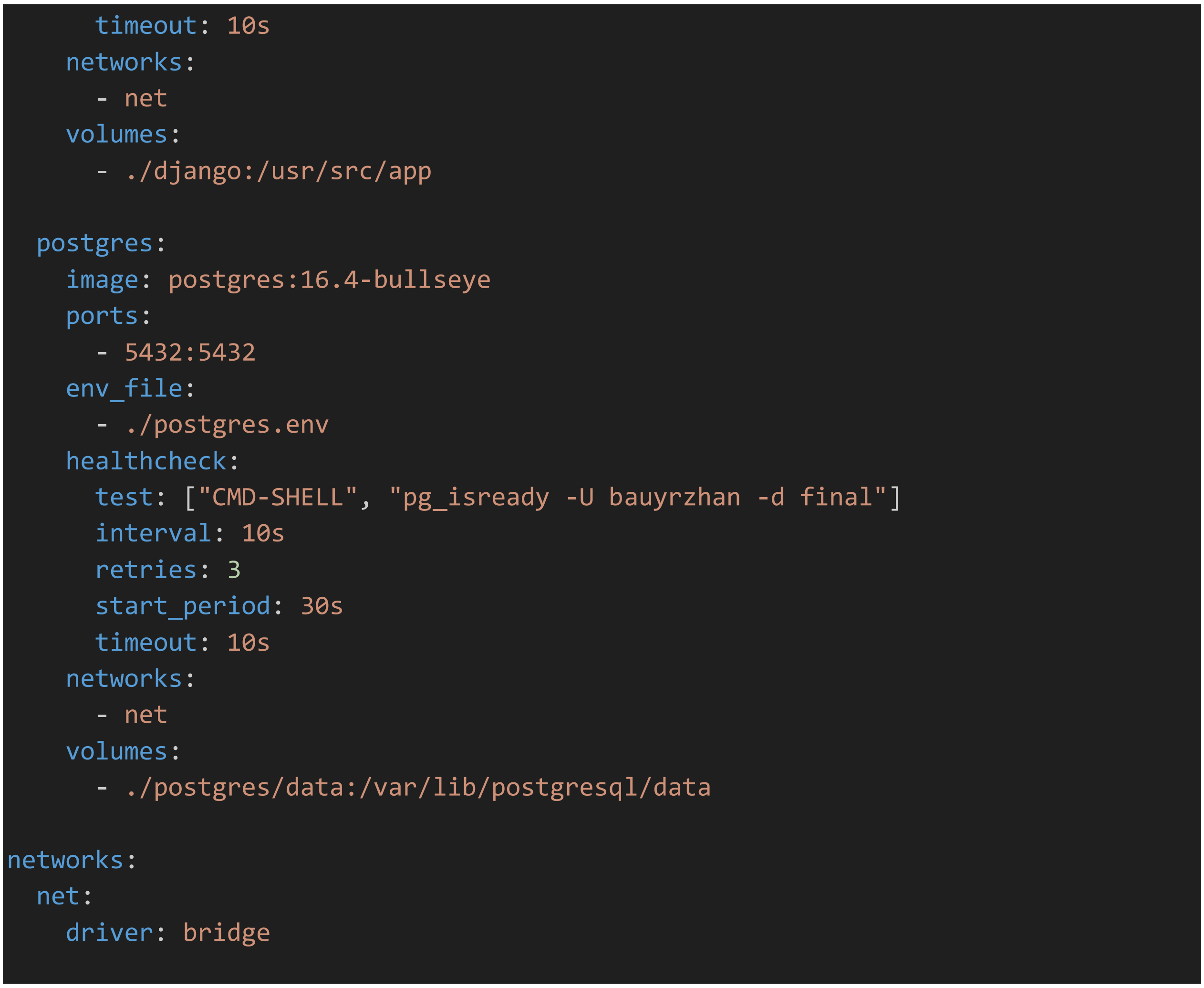
Using Docker Compose simplifies the process of starting and stopping these services together. Instead of running separate Docker commands for each container, we can use a single command to start all the services defined in the docker-compose.yml file. This ensures that all components are up and running smoothly and are properly connected to each other through defined networks and volumes.

Docker Compose also makes it easy to manage dependencies. For example, our frontend service depends on the Django service, and the Django service depends on the PostgreSQL database. These dependencies are clearly defined in the docker-compose.yml file, ensuring that the services start in the correct order.

Overall, Docker Compose streamlines the management of multi-container applications, making it more efficient and less error-prone.

```yaml
services:
  frontend-js:
    build: frontend-js/.
    ports:
      - 80:80
    depends_on:
      django:
        condition: service_healthy
        restart: true
    networks:
      - net
    volumes:
      - ./frontend-js:/usr/share/nginx/html

  django:
    build: django/.
    ports:
      - 8000:8000
    env_file:
      - ./django.env
    depends_on:
      postgres:
        condition: service_healthy
        restart: true
    healthcheck:
      test: ["CMD-SHELL", "curl -f http://0.0.0.0:8000/ || exit 1"]
      interval: 10s
      retries: 3
      start_period: 30s
```



```
      timeout: 10s
    networks:
      - net
    volumes:
      - ./django:/usr/src/app

  postgres:
    image: postgres:16.4-bullseye
    ports:
      - 5432:5432
    env_file:
      - ./postgres.env
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U bauyrzhan -d final"]
      interval: 10s
      retries: 3
      start_period: 30s
      timeout: 10s
    networks:
      - net
    volumes:
      - ./postgres/data:/var/lib/postgresql/data

networks:
  net:
    driver: bridge
```

Figure 4. docker-compose.yml

Docker networking and volumes

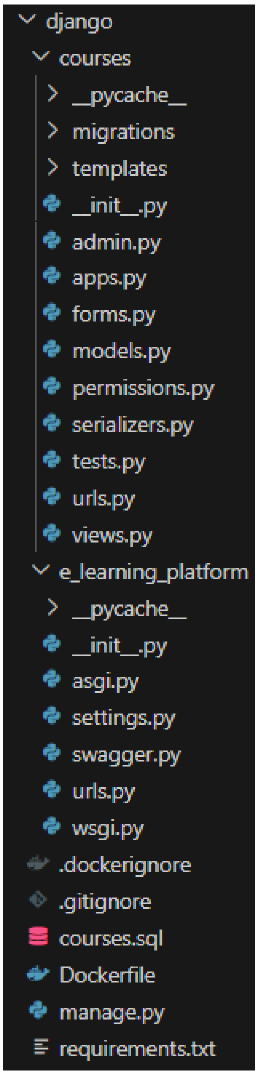
Docker networking and volume management are important for running applications in containers. Docker networking allows different containers to communicate with each other, while volumes help store data that needs to be saved even when containers are stopped or removed.

In Docker, there are different types of networks. The most common one is the bridge network, which is used by default. Bridge networks let containers on the same network communicate with each other easily. This is useful for our project, where the Django application, frontend, and PostgreSQL database need to work together.

Volumes are used to store data outside of the container's file system. This means that even if a container is deleted, the data in the volume is still safe. For our project, we use volumes to store database data so that it persists between container restarts. This is important for keeping user data and course information safe.

By using Docker networking and volumes, we can create a reliable and efficient environment for our e-learning platform. Networking ensures that different parts of the application can talk to each other, and volumes help keep important data secure and persistent.

Django



```
django
  courses
    __pycache__
    migrations
    templates
    __init__.py
    admin.py
    apps.py
    forms.py
    models.py
    permissions.py
    serializers.py
    tests.py
    urls.py
    views.py
  e_learning_platform
    __pycache__
    __init__.py
    asgi.py
    settings.py
    swagger.py
    urls.py
    wsgi.py
  .dockerignore
  .gitignore
  courses.sql
  Dockerfile
  manage.py
  requirements.txt
```

Figure 5. Django project structure

As we did in previous assignment here we have also 1 project (e_learning_platform) and 1 application (courses).

Django is a high-level web framework for building web applications quickly and easily. It is written in Python and follows the model-template-views (MTV) architectural pattern. This means it helps developers structure their code in a way that separates the data model, user interface, and application logic.

One of the main benefits of Django is its simplicity. It comes with a lot of built-in features, such as user authentication, admin interface, and database management, which makes it easier to develop complex web applications. Django is also known for its strong security features, helping protect applications from common web threats.

Another significant advantage of Django is its scalability. It can handle a large number of users and high traffic, making it suitable for both small projects and large-scale applications. Additionally, Django is well-documented and has a large community of developers, which means there are plenty of resources and support available.

In web development, Django is valued for its ability to speed up the development process while maintaining high standards of code quality and security. It allows developers to focus on building features rather than spending time on repetitive tasks, making it a powerful tool for creating modern web applications.

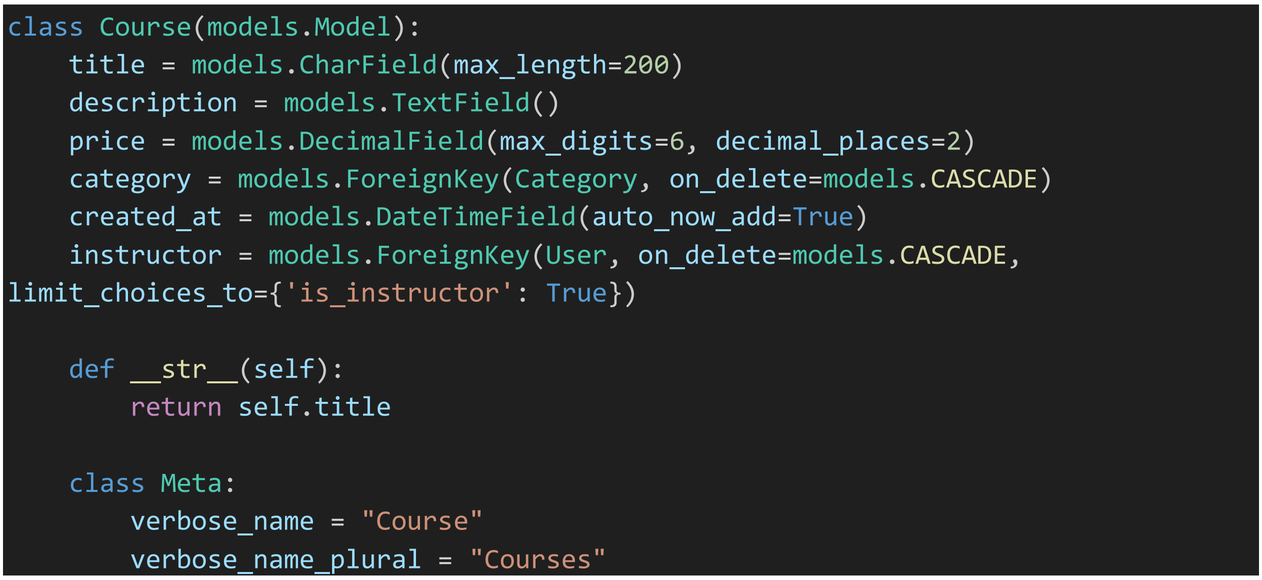
Models

In Django, models are used to define the structure of the database. They represent the data and the relationships between the data. Each model is a class that subclasses django.db.models.Model, and each attribute of the model represents a database field.

For our e-learning platform, several models were created to handle different types of data:

1. User: This model represents the users of the platform. It includes fields for user information such as username, email, and password. Additionally, it has boolean fields to indicate whether a user is a student or an instructor.
2. Category: This model is used to categorize the courses. It has fields for the name and description of the category.
3. Course: This model represents the courses available on the platform. It includes fields for the course title, description, price, and the category it belongs to. It also has a field to link the course to an instructor.
4. Enrollment: This model keeps track of which users are enrolled in which courses. It includes fields for the user, the course, and the enrollment date.
5. Lesson: This model represents the lessons within a course. It includes fields for the lesson title, content, and an optional video URL.
6. Review: This model allows users to leave reviews for courses. It includes fields for the course, the user, the rating, and the review comment.
7. Payment: This model handles the payment information. It includes fields for the user, the amount paid, and the payment date.
8. Quiz: This model represents quizzes associated with courses. It includes fields for the quiz title and total marks.
9. QuizQuestion: This model represents the questions within a quiz. It includes fields for the question text, multiple choice options, and the correct option.
10. UserProgress: This model tracks the progress of users in their courses. It includes fields for the user, the course, completed lessons, and quiz scores.

These models are essential for organizing and managing the data in the e-learning platform. They define the database schema and help ensure data consistency and integrity.



```python
class Course(models.Model):
    title = models.CharField(max_length=200)
    description = models.TextField()
    price = models.DecimalField(max_digits=6, decimal_places=2)
    category = models.ForeignKey(Category, on_delete=models.CASCADE)
    created_at = models.DateTimeField(auto_now_add=True)
    instructor = models.ForeignKey(User, on_delete=models.CASCADE,
limit_choices_to={'is_instructor': True})

    def __str__(self):
        return self.title

    class Meta:
        verbose_name = "Course"
        verbose_name_plural = "Courses"
```

Figure 6. /courses/models.py fragment

You can see all large files in GitHub repository. Find link in references section.

Views

In Django, views are an important part of the framework. They handle user requests and return the appropriate responses. Views connect the models and templates, making the web application work smoothly.

There are different types of views in Django. Class-based views (CBVs) and function-based views (FBVs) are the most common. Function-based views are simple functions that take a web request and return a web response. Class-based views, on the other hand, provide more structure and reuse by using Python classes.

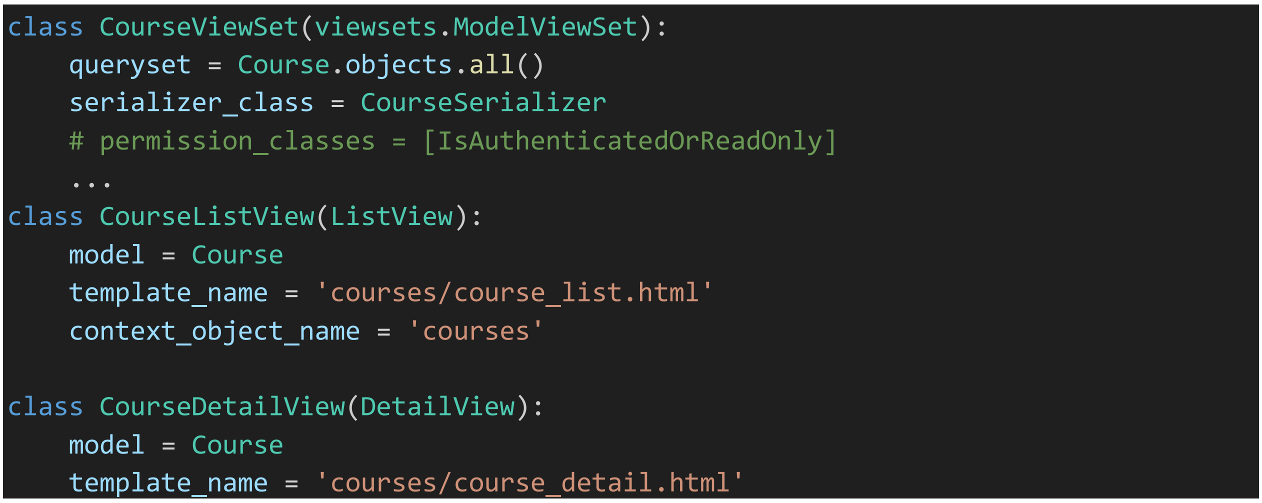
For our e-learning platform, we used class-based views and viewsets. The views handle various tasks such as displaying a list of items, showing detailed information about a single item, creating new items, updating existing items, and deleting items.

We implemented viewsets using Django Rest Framework (DRF) to manage the API endpoints. Viewsets simplify the process of creating API views by combining the logic for handling multiple HTTP methods (GET, POST, PUT, DELETE). Each viewset is linked to a specific model and serializer, ensuring that the data is processed correctly.

Here's a brief overview of the viewsets implemented in our project:

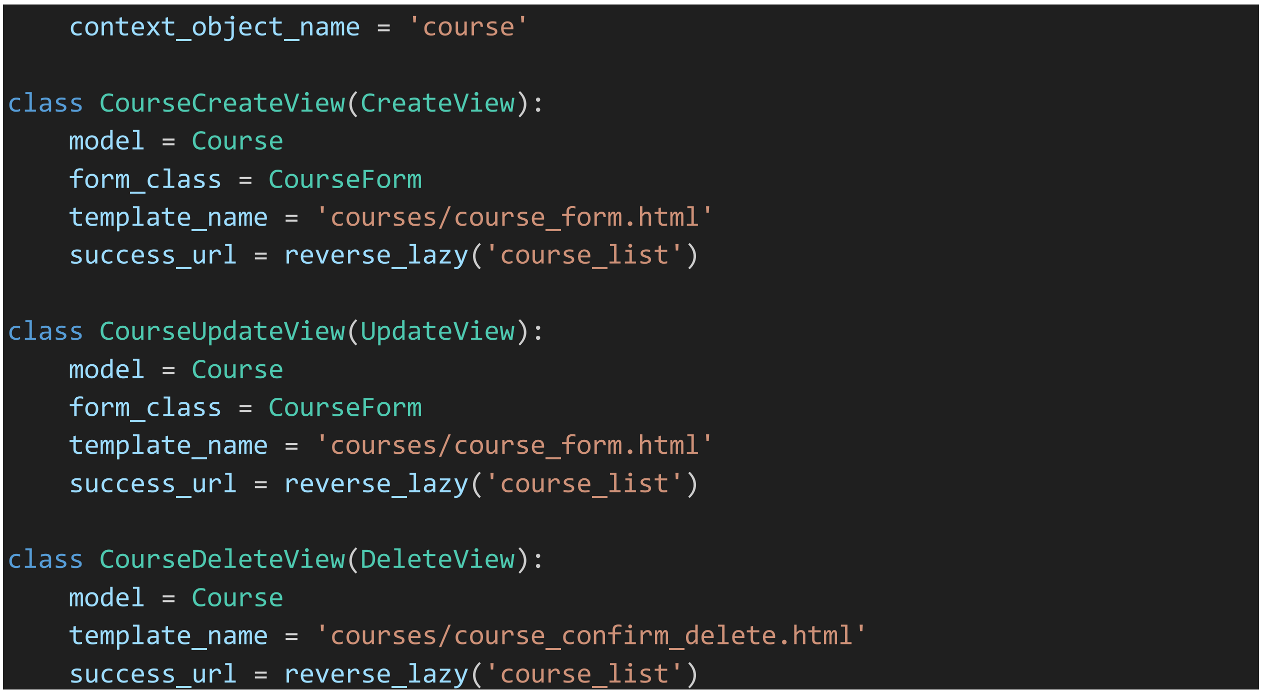
1. UserViewSet: Manages user data, such as listing users and handling user creation and updates.
2. CategoryViewSet: Handles course categories, including listing and managing categories.
3. CourseViewSet: Manages courses, allowing users to view, create, update, and delete courses.
4. EnrollmentViewSet: Tracks enrollments, letting users enroll in courses and manage their enrollments.
5. LessonViewSet: Manages lessons within courses, including viewing and updating lesson content.
6. ReviewViewSet: Handles course reviews, allowing users to read and write reviews.
7. PaymentViewSet: Manages payment information, tracking payments made by users.
8. QuizViewSet: Handles quizzes associated with courses, including creating and managing quizzes.
9. QuizQuestionViewSet: Manages quiz questions, allowing users to view and manage quiz content.
10. UserProgressViewSet: Tracks user progress in courses, including completed lessons and quiz scores.

These views ensure that the web application functions correctly and provides a seamless experience for the users.



```python
class CourseViewSet(viewsets.ModelViewSet):
    queryset = Course.objects.all()
    serializer_class = CourseSerializer
    # permission_classes = [IsAuthenticatedOrReadOnly]
    ...
class CourseListView(ListView):
    model = Course
    template_name = 'courses/course_list.html'
    context_object_name = 'courses'

class CourseDetailView(DetailView):
    model = Course
    template_name = 'courses/course_detail.html'
```



```python
    context_object_name = 'course'

class CourseCreateView(CreateView):
    model = Course
    form_class = CourseForm
    template_name = 'courses/course_form.html'
    success_url = reverse_lazy('course_list')

class CourseUpdateView(UpdateView):
    model = Course
    form_class = CourseForm
    template_name = 'courses/course_form.html'
    success_url = reverse_lazy('course_list')

class CourseDeleteView(DeleteView):
    model = Course
    template_name = 'courses/course_confirm_delete.html'
    success_url = reverse_lazy('course_list')
```

Figure 7. /courses/views.py fragment

You can see all large files in GitHub repository. Find link in references section.

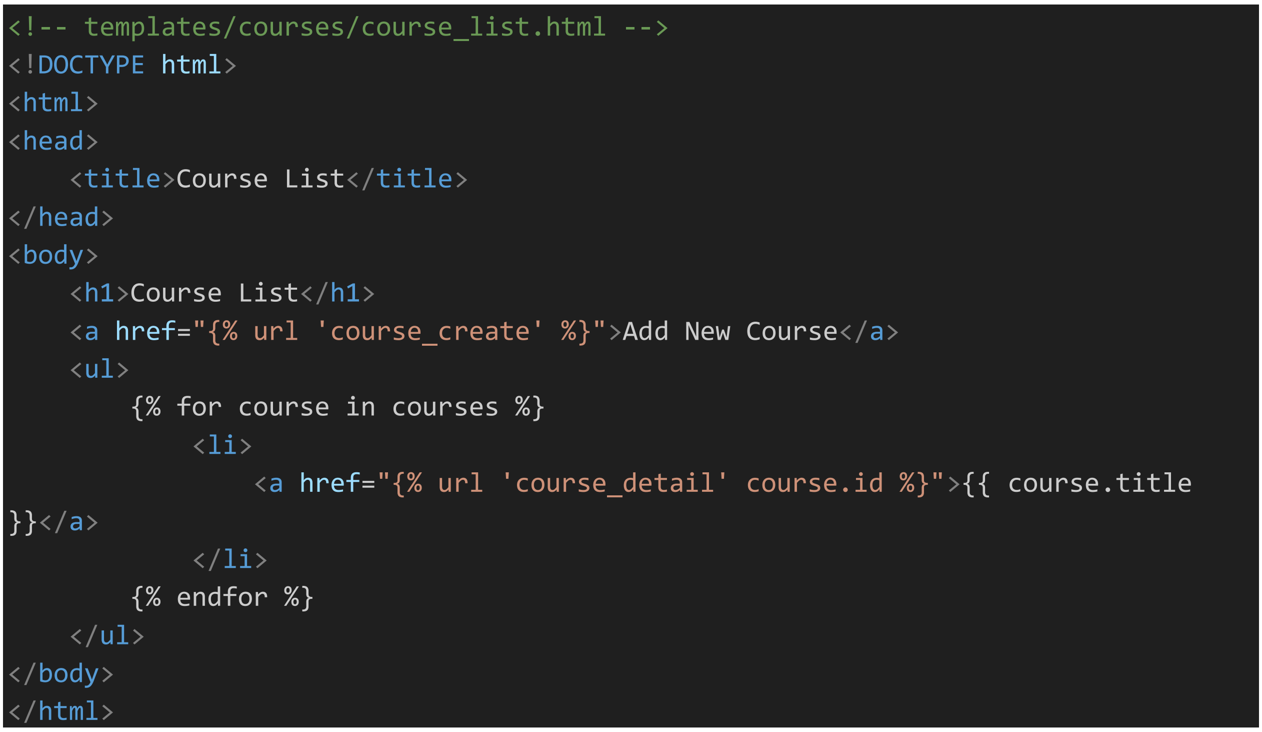
Templates

In Django, templates play a crucial role in rendering dynamic HTML content. Templates are used to create the HTML structure of web pages and can include placeholders for dynamic data. This allows developers to generate HTML pages that display different data without needing to write separate HTML files for each piece of data.

The Django template language provides various tools to include dynamic content in templates. For example, it allows using variables, loops, and conditional statements. This makes it easier to customize the appearance of web pages based on the data being displayed.

In our e-learning platform, templates are used to display course information, user interactions, and other dynamic content. For example, the course_list.html template is used to show a list of courses. It includes a loop that iterates through each course and displays its title. This way, whenever new courses are added, the template automatically updates to show the new courses without needing to change the HTML code.

Here is a sample template for the course list:



```
<!-- templates/courses/course_list.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Course List</title>
</head>
<body>
    <h1>Course List</h1>
    <a href="{% url 'course_create' %}">Add New Course</a>
    <ul>
        {% for course in courses %}
            <li>
                <a href="{% url 'course_detail' course.id %}">{{ course.title
}}</a>
            </li>
        {% endfor %}
    </ul>
</body>
</html>
```

Figure 8. models.py

In this template, the *{% for course in courses %}* loop dynamically generates a list of courses. The *{% url 'course_detail' course.id %}* tag creates a link to the detailed view of each course.

Overall, templates in Django make it easy to create dynamic and interactive web pages by separating the HTML structure from the data that is displayed.

Django Rest Framework (DRF)

The Django Rest Framework (DRF) is a powerful toolkit for building Web APIs. It simplifies the process of creating RESTful APIs, making it easier for developers to build and manage these APIs.

One of the main advantages of DRF is its flexibility. It allows developers to create customizable and reusable API components. DRF also provides built-in authentication and permissions, making it secure and easy to manage user access.

DRF supports a wide range of content types, including JSON, XML, and more. This ensures that the APIs can communicate effectively with different systems. Additionally, DRF comes with a browsable API, which means that developers can interact with the API through a web browser, making it easier to test and debug.

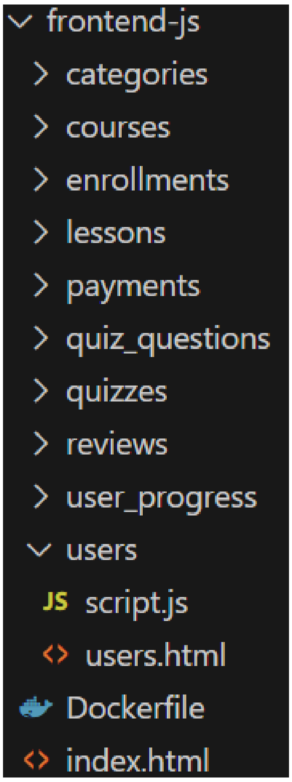
In our e-learning platform project, the API architecture developed using DRF includes several viewsets and serializers. Each viewset handles the CRUD (Create, Read, Update, Delete) operations for a specific model, while serializers convert model instances to JSON format and vice versa.

The main components of our API architecture are:

1. User API: Manages user data, allowing for user registration, authentication, and profile management.
2. Course API: Handles course information, enabling the creation, updating, and retrieval of course details.
3. Enrollment API: Tracks user enrollments in courses, allowing users to enroll and view their course status.
4. Lesson API: Manages course lessons, providing endpoints to access and update lesson content.
5. Review API: Allows users to leave and view reviews for courses.
6. Payment API: Handles payment transactions, enabling users to make and view payments for courses.
7. Quiz API: Manages quizzes, including creating and retrieving quiz details.
8. QuizQuestion API: Manages quiz questions, allowing users to view and answer questions.
9. UserProgress API: Tracks user progress in courses, providing endpoints to view completed lessons and quiz scores.

Overall, DRF makes it easy to develop and maintain a robust and secure API for our e-learning platform, enhancing the overall user experience.

Frontend integration



```
frontend-js
  categories
  courses
  enrollments
  lessons
  payments
  quiz_questions
  quizzes
  reviews
  user_progress
  users
    script.js
    users.html
  Dockerfile
  index.html
```

Figure 9. Frontend structure

In this project, the frontend is written in pure JavaScript without using any frameworks. It interacts with the Django backend through APIs to fetch and manage data.

The main file, index.html, serves as the entry point. It lists different models like users and courses, allowing users to view their data. Each model has its own directory structure on the frontend side. For example, the /frontend-js/users/users.html file displays a list of all users retrieved from the backend. The /frontend-js/users/script.js file contains JavaScript code to interact with the backend API.

The JavaScript code uses GET requests to fetch data from the backend and display it on the frontend. It also uses DELETE requests to remove data from the backend when needed. This structure is replicated for other models, such as courses and enrollments, ensuring a consistent approach across the application.

By using JavaScript to handle API requests, the frontend can dynamically update the content displayed to users. This makes the application more interactive and responsive, providing a better user experience.

Overall, the integration between the frontend and the Django backend is achieved through well-defined API endpoints and JavaScript code, allowing seamless communication and data management.

To display the frontend, a docker container with Nginx was used. It is used as a reverse proxy to interact with the backend. It runs on port 80.

Challenges and solutions

The biggest problem was writing the code for the frontend. Due to lack of time, it was not possible to study and apply VueJS from the requirements for this work. However, instead of the above framework, "pure" JS was used. It also coped well with its not very complex tasks of displaying content and interacting with the backend.

Conclusion

This project successfully developed an e-learning platform using Django and Docker. The key findings highlight the effective use of these technologies to create a robust, scalable, and efficient system. The platform allows users to enroll in courses, view lessons, take quizzes, and track their progress.

Through the project, several important lessons were learned. One major lesson is the importance of containerization with Docker, which makes the deployment process easier and more consistent. Another lesson is the power of Django in managing complex web applications, providing essential features out of the box and maintaining high security standards.

Potential future improvements include enhancing the user interface to make it more user-friendly and visually appealing. Adding more interactive features, such as live chat support and real-time notifications, could further improve the user experience. Additionally, integrating advanced analytics could help track user engagement and performance more effectively.

Overall, this project demonstrates the potential of combining Django and Docker to build modern web applications, setting a strong foundation for further development and enhancement.

References

Links to Django tutorials from official documentation:

1. https://docs.djangoproject.com/en/5.1/intro/tutorial01/
2. https://docs.djangoproject.com/en/5.1/intro/tutorial02/
3. https://docs.djangoproject.com/en/5.1/intro/tutorial03/
4. https://docs.djangoproject.com/en/5.1/intro/tutorial04/
5. https://docs.djangoproject.com/en/5.1/intro/tutorial02/

Link to official Docker documentation:

- https://docs.docker.com/

Link to official Docker image registry where from I took base images for compose:

- https://hub.docker.com/

Links to some topics in StackOverflow and other resources:

1. https://stackoverflow.com/questions/13164048/text-box-input-height
2. https://stackoverflow.com/questions/3681627/how-to-update-fields-in-a-model-without-creating-a-new-record-in-django
3. https://stackoverflow.com/questions/42614172/how-to-redirect-from-a-view-to-another-view-in-django
4. https://stackoverflow.com/questions/3805958/how-to-delete-a-record-in-django-models
5. https://stackoverflow.com/questions/14719883/django-can-not-get-a-time-function-timezone-datetime-to-work-properly-gett
6. https://stackoverflow.com/questions/3289601/null-object-in-python
7. https://stackoverflow.com/questions/11336548/how-to-get-post-request-values-in-django
8. https://stackoverflow.com/questions/11714721/how-to-set-the-space-between-lines-in-a-div-without-setting-line-height
9. https://stackoverflow.com/questions/3430432/django-update-table
10. https://stackoverflow.com/questions/15128705/how-to-insert-a-row-of-data-to-a-table-using-djangos-orm
11. https://stackoverflow.com/questions/67155473/how-can-i-get-values-from-a-view-in-django-every-time-i-access-the-home-page
12. https://stackoverflow.com/questions/3500859/django-request-get
13. https://stackoverflow.com/questions/50346326/programmingerror-relation-django-session-does-not-exist
14. https://stackoverflow.com/questions/18713086/virtualenv-wont-activate-on-windows
15. https://www.geeksforgeeks.org/swagger-integration-with-python-django/
16. https://stackoverflow.com/questions/51182823/django-rest-framework-nested-serializers
17. https://www.django-rest-framework.org/api-guide/permissions/
18. https://www.django-rest-framework.org/api-guide/versioning/
19. https://www.django-rest-framework.org/api-guide/viewsets/

Link to my GitHub repository (screenshots, logs and SQL structure included):

- https://github.com/BauyrzhanAzimkhanov/Web-application-development-MSc