KAZAKH-BRITISH TECHNICAL UNIVERSITY

Final Project Report

E-Learning Platform

Serzhan Yerasyl

23MD0434

22.12.2024

Almaty,
2024

Executive Summary

This project focuses on developing an e-learning platform using Django for backend development and Docker for containerization. The platform provides features like user authentication, course management, and reviews for courses. By using Docker, launching the project is simplified, ensuring consistent environments across systems.

2. Table of Contents:

Introduction

I worked with Docker and Django - they're match for what I'm trying to build. Docker makes our lives easier by letting us package everything up neatly, while Django gives us all the tools we need to build a solid web app. It's an e-learning platform where students can take courses, track their progress, and available jwt auth. They can also leave reviews, which helps keep our content quality high.

System Architecture

The system architecture follows a modular approach. The backend is built with Django and uses the Django REST Framework for API management. Docker containers are used to host the backend, database, and web server. Components include separate containers for the application, PostgreSQL. These containers communicate through a Docker-managed network. This architecture ensures easy maintenance, and consistent environments across development

Table Descriptions

The database schema includes several tables:
Users: Manages user details, roles, and authentication.
Courses: Stores course information like title, description, and instructor.
Enrollments: Tracks which users are enrolled in specific courses.
Lessons: Contains content for each course.
Reviews: Stores user feedback and ratings for courses.
Payments: Records payment transactions.
Quizzes: Supports quizzes linked to specific courses. Each table is interconnected to maintain relational consistency and integrity.

Intro to Containerization: Docker

Containerization is the packaging of an application together with its dependencies, libraries and configuration files as a single entity into a single object, its main advantages are lightness and portability - that is, if it runs locally, then with a very high probability it will run on another machine. In order to install Docker, I downloaded Docker Desktop and enabled virtualization on the motherboard. In order to run the container, we use the command docker run image name or path to the Dockerfile, this command will check if such an image exists locally, if not, it will pull it from the repository and run it as a container.

Dockerfile

The Dockerfile I used for this application. FROM python:3.11 - this command will start a container with a Linux operating system and a Python interpreter installed in it WORKDIR /app set the working directory inside the container COPY requirements.txt . copy the requirements with a separate command to the directory so that each time the code changes, you don't have to copy and install it again RUN pip install --no-cache-dir -r requirements.txt install dependencies COPY . . copy the remaining files to the working directory EXPOSE 8000 specify which port the container will use

Docker-compose

```yaml
version: '3.8'

services:
  db:
      image: postgres
      container_name: pgdb
      environment:
          - POSTGRES_DB=postgres
          - POSTGRES_USER=postgres
          - POSTGRES_PASSWORD=postgres
      volumes:
          - postgres_data:/var/lib/postgresql/data/
      ports:
          - 5433:5432
      networks:
          - django_network

  web:
      build: .
      container_name: django
      command: python e_learning/manage.py runserver 0.0.0.0:8000
      volumes:
          - .:/app
      ports:
          - "8000:8000"
      depends_on:
          - db
      environment:
        DATABASE_URL: postgres://postgres:qwerty@db/postgres
        DB_NAME: postgres
        DB_USER: postgres
        DB_PASSWORD: qwerty
        DB_HOST: db
        DB_PORT: 5432
      networks:
      - django_network

volumes:
  postgres_data:

networks:
  django_network:
```

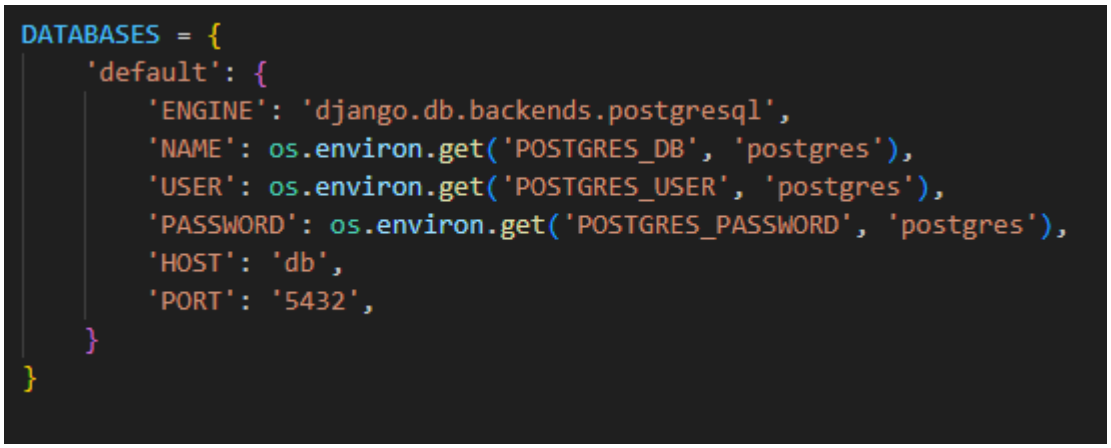
In docker-compose we have two services, db and django app. In the db service we use image postgres and set credentials for it, and set volumes for saving data to local machine for further new containers, because after removing container all data will disappear. Also we map ports from docker container to local machine and specify network where db will defined. In web service we use our Dockerfile as our image, also define volumes and ports and specify that it depends from db. Setting up env variables for connecting to db and define his network. And command that will start our app. I wrote a Dockerfile for Django, wrapped two services in docker compose, configured them to be in the same network, so that they were in the same domain, configured environment variables for the Django application so that it could pull secret data from environment variables inside the container. I configured docker volumes for the Django database and application so that the data in the container and on the local host were mapped to each other, this is done so that if the container is deleted, then when creating a new container, it will pull data from the local host files

Docker Networking and Volumes

Docker networks allow containers to interact if they are on the same network, this is like a definition area for them. Within the same network, they access each other by the service name described in the docker compose file, and not via an IP address Docker volumes are needed so that the data inside the container is not lost after they are completed or deleted. To do this, a folder is created on the local machine that is associated with the container folder, after deleting the container, all data inside the container is lost, but they are on the local machine, as soon as a new container is launched, the data from the local directory will be displayed in the container directory.

Django

Django is a Python web framework that simplifies backend development. It offers built-in features like ORM, authentication, and an admin panel. In this project, Django manages course data, user authentication, and API endpoints. Its modular structure ensures that the application is easy to maintain.
To create a django project I entered django-admin startproject myproject the skeleton of the project is created. Then python manage.py startapp tasks to create an application inside the project, then this application was added to INSTALLED_APPS so that the project recognizes this application. In order to connect to the database that we raised via docker, you need to change the credentials in the variable database



Models

The structure and functionality of the database in our Django application is defined in models. Django models allow us to represent the structure of our data, easily manage it via an ORM, and provide the flexibility to make changes as needed. Migrations in Django allow us to apply changes to the database schema based on our models, keeping the database structure in sync with the code. Once we have described the structure in the models, we write python manage.py makemigrations to have Django generate migration files and apply them to the database with python manage.py migrate

```python
from django.db import models
from django.contrib.auth.models import AbstractUser
from django.utils.timezone import now

class User(AbstractUser):
    is_student = models.BooleanField(default=False)
    is_instructor = models.BooleanField(default=False)

class Category(models.Model):
    category_id = models.AutoField(primary_key=True)
    name = models.CharField(max_length=255)
    description = models.TextField()

    def __str__(self):
        return self.name

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.CASCADE)
    created_at = models.DateTimeField(default=now)
    instructor = models.ForeignKey(User, on_delete=models.CASCADE, limit_choices_to={'is_instructor': True})

    def __str__(self):
        return self.title

class Enrollment(models.Model):
    enrollment_id = models.AutoField(primary_key=True)
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    course = models.ForeignKey(Course, on_delete=models.CASCADE)
    enrollment_date = models.DateTimeField(default=now)
    status = models.CharField(max_length=50, choices=[('active', 'Active'), ('completed', 'Completed')], default='active')

class Lesson(models.Model):
    lesson_id = models.AutoField(primary_key=True)
    course = models.ForeignKey(Course, on_delete=models.CASCADE)
    title = models.CharField(max_length=255)
    content = models.TextField()
    video_url = models.URLField(blank=True, null=True)
```

Views

CourseListView class is responsible for displaying a list of all courses. In the get method, a database query is made to retrieve all Course model objects using Course.objects.all(). Then, the courses.html HTML template is loaded and data is passed to it via the context, which contains the list of courses. The template is rendered using loader.get_template and template.render, and the result is returned as an HTTP response.

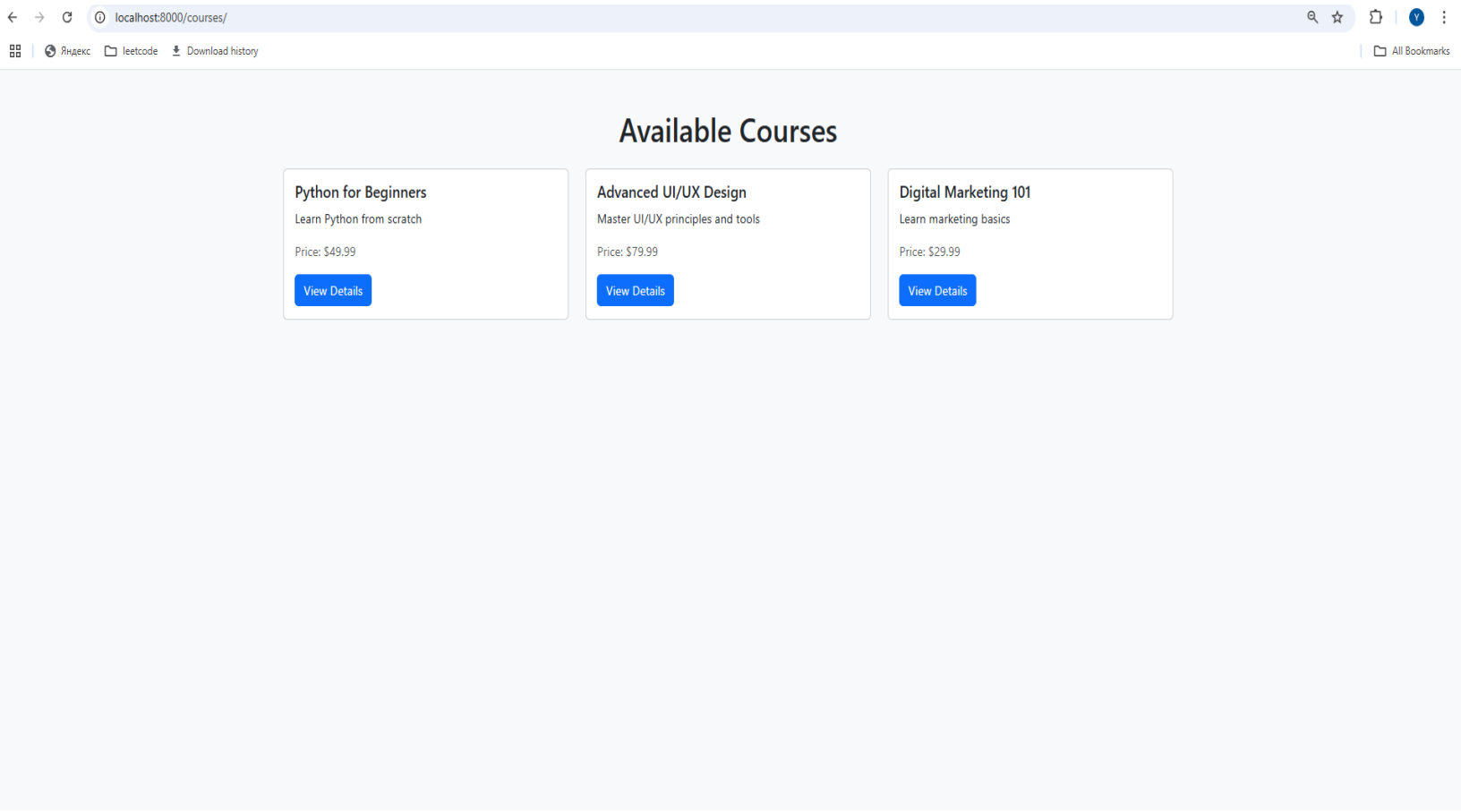
CourseDetailView class is designed to display details of a specific course. In the get method, a course is retrieved from the database using method get_object by field Id. If a course with the specified ID is not found, a code 404 will return. Then, the course_detail.html template is loaded and the course information is passed. The template is rendered and returned as an HTTP response.

```python
class CourseListView(View):
    def get(self, request):
        courses = Course.objects.all()
        template = loader.get_template('courses.html')
        context = {'courses': courses}
        return HttpResponse(template.render(context, request))

class CourseDetailView(View):
    def get(self, request, course_id):
        course = get_object_or_404(Course, pk=course_id)
        template = loader.get_template('course_detail.html')
        context = {'course': course}
        return HttpResponse(template.render(context, request))
```

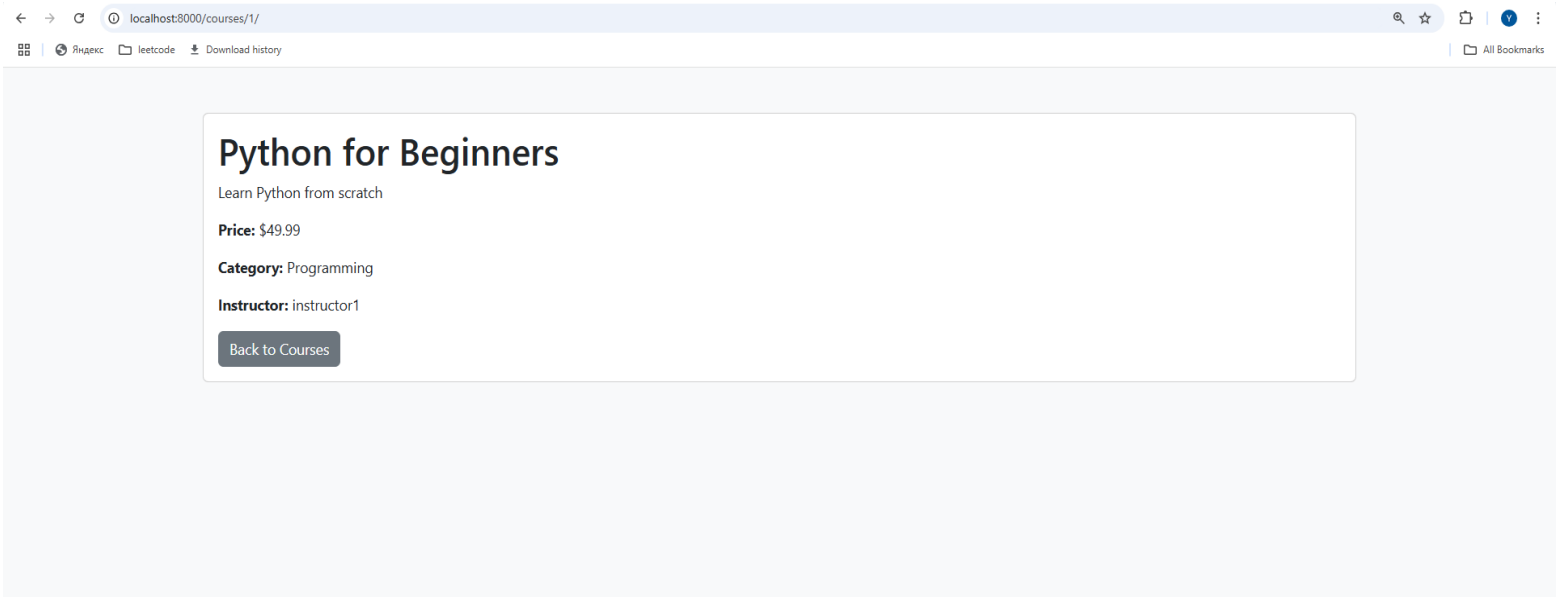
Templates

This HTML template displays a list of available courses using Bootstrap for a modern and responsive design. The header includes styles and metadata for mobile optimization. The main content includes a header and a grid of course cards, where each card shows the course name, a short description (truncated to 100 characters), price, and a button to navigate to the details page. The template uses Django's {% for course in courses %} loop to dynamically generate cards from the passed list of courses, ensuring usability and scalability.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Courses</title>
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/bootstrap.min.css">
</head>
<body class="bg-light">
    <div class="container mt-5">
        <h1 class="text-center mb-4">Available Courses</h1>
        <div class="row">
            {% for course in courses %}
                <div class="col-md-4 mb-4">
                    <div class="card">
                        <div class="card-body">
                            <h5 class="card-title">{{ course.title }}</h5>
                            <p class="card-text">{{ course.description|truncatechars:100 }}</p>
                            <p class="text-muted">Price: ${{ course.price }}</p>
                            <a href="/courses/{{ course.course_id }}/" class="btn btn-primary">View Details</a>
                        </div>
                    </div>
                </div>
            {% endfor %}
        </div>
    </div>
</body>
</html>
```


localhost:8000/courses/
Яндекс
leetcode
Download history
All Bookmarks
Available Courses
Python for Beginners
Learn Python from scratch
Price: $49.99
View Details
Advanced UI/UX Design
Master UI/UX principles and tools
Price: $79.99
View Details
Digital Marketing 101
Learn marketing basics
Price: $29.99
View Details

All courses



Separately course

Django-Rest-Framework

Django REST Framework is a tool for creating apis in Django. It allows developers to easily create apis through which different applications iteracts with it to get data.
Instead of manually writing a lot of code, DRF provides convenient tools such as: serializers, viewSets, routers etc.

```
REST_FRAMEWORK = {
    'DEFAULT_SCHEMA_CLASS': 'drf_spectacular.openapi.AutoSchema',
   'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ),
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticatedOrReadOnly',
    ),

}
```

This code configures Django REST Framework to work with the API, adding automatic documentation generation via drf-spectacular (OpenAPI/Swagger), authentication using JWT tokens (rest_framework_simplejwt), and permission-based data access. The settings allow all users to view data (GET requests), but changes (POST, PUT, DELETE) are only available to authorized users. This ensures the security of the API and simplifies working with the documentation.

Views

This code creates two ViewSets to work with the Category and Course model APIs using the Django REST Framework. Both ViewSets inherit from ModelViewSet, which automatically adds the ability to perform CRUD (create, read, update, delete) operations. The full list of data to work with is specified via queryset, and the data conversion between the JSON format and model objects is performed using the specified serializers (CategorySerializer and CourseSerializer). Access to data is restricted using the IsAuthenticatedOrReadOnly permission class unauthorized users can only view the data, and changes are available only to authorized users. This simplifies the creation of secure and functional APIs.

```
class CategoryViewSet(viewsets.ModelViewSet):
    queryset = models.Category.objects.all()
    serializer_class = serializers.CategorySerializer
    permission_classes = [IsAuthenticatedOrReadOnly]

class CourseViewSet(viewsets.ModelViewSet):
    queryset = models.Course.objects.all()
    serializer_class = serializers.CourseSerializer
    permission_classes = [IsAuthenticatedOrReadOnly]
```

Routing

Urls are configured with DRF's DefaultRouter to manage endpoint routing automatically. The API Provides endpoints for post operations (list, create, retrieve, update, delete). /api/v1/courses/: Handles CRUD operations for coruses. Additionally, API documentation is available via Swagger and ReDoc at /api/schema/swagger-ui/ and /api/schema/redoc/.

```
router = DefaultRouter()
router.register(r'users', viewSets.UserViewSet)
router.register(r'categories', viewSets.CategoryViewSet)
router.register(r'courses', viewSets.CourseViewSet)
router.register(r'enrollments', viewSets.EnrollmentViewSet)
router.register(r'lessons', viewSets.LessonViewSet)
router.register(r'reviews', viewSets.ReviewViewSet)
router.register(r'payments', viewSets.PaymentViewSet)
router.register(r'quizzes', viewSets.QuizViewSet)
router.register(r'quiz-questions', viewSets.QuizQuestionViewSet)
router.register(r'user-progress', viewSets.UserProgressViewSet)
```

This code creates a router to automatically manage API endpoints using the DefaultRouter. The router registers ten view sets. This automatically generates standard RESTful routes such as GET /categories/ to get a list of category, POST /categories/ to create a new post, GET /categories/{id}/ to get a specific post, and similar routes for comments. This approach simplifies URL configuration and makes the API structured and easy to use.

Authentication and Permissions

Token-based authentication is implemented using rest_framework_simplejwt, ensuring secure access via JWT tokens. Anonymous users can view posts and comments, while authenticated users can create, update, or delete them. A custom permission class, IsAuthorOrReadOnly, restricts modifications to posts or comments to their respective authors, ensuring user-specific access control.

```
REST_FRAMEWORK = {
    'DEFAULT_SCHEMA_CLASS': 'drf_spectacular.openapi.AutoSchema',
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ),
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticatedOrReadOnly',
    ),

}
```

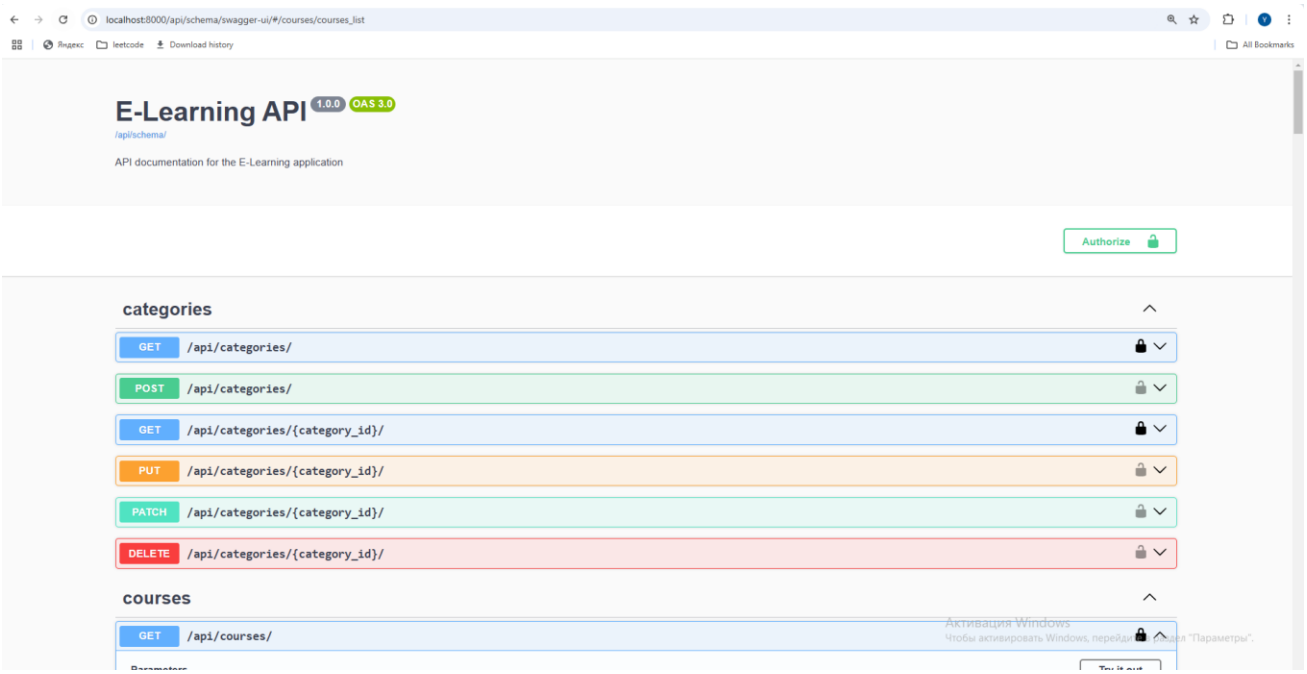
Default auth classs specifies the authentication method used in the API. JWT (JSON Web Token) authentication provided by the rest_framework_simplejwt library is enabled here. This allows you to secure API endpoints by requiring a token in the request header to access protected resources.

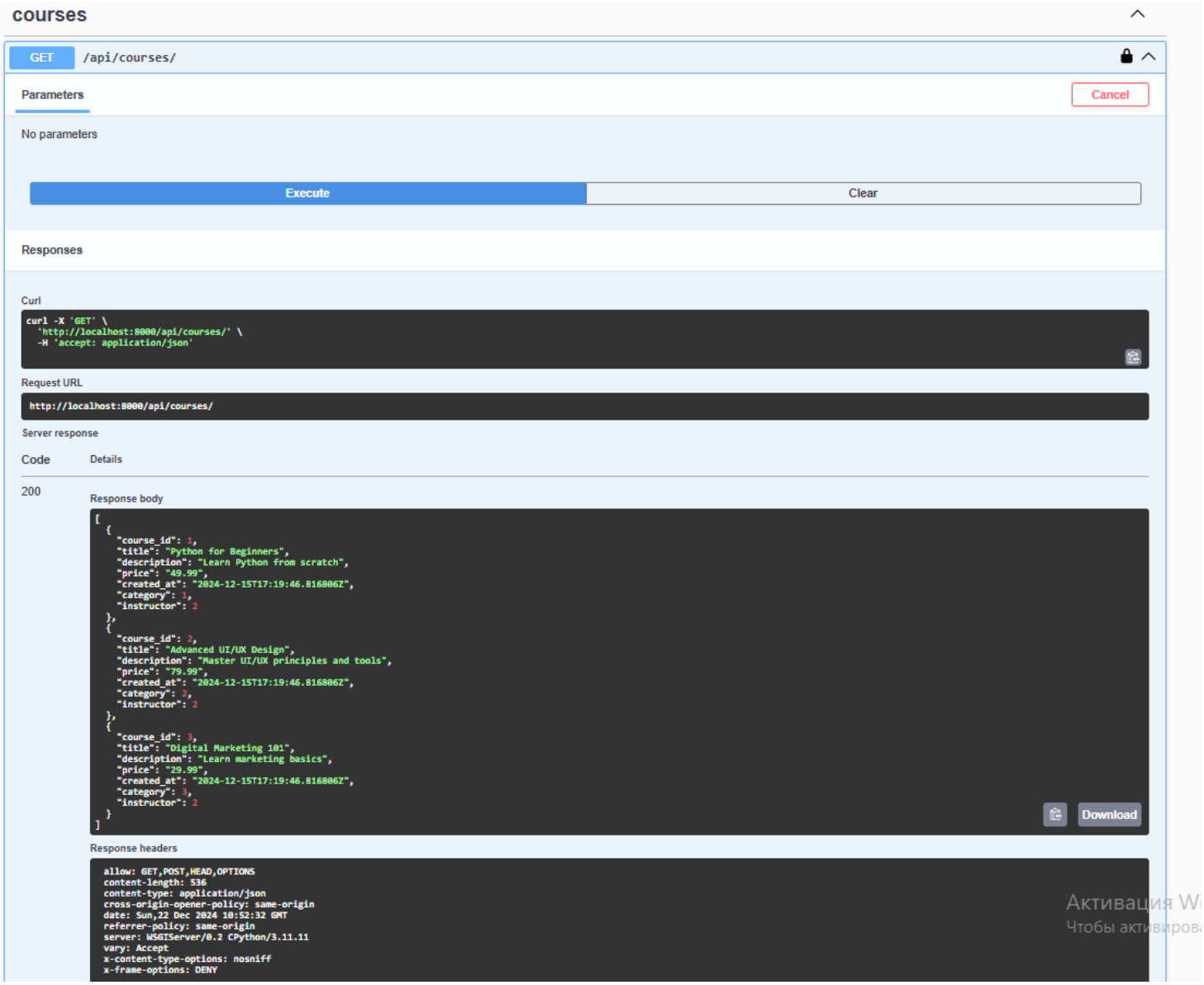
```
class IsAuthorOrReadOnly(BasePermission):

    def has_object_permission(self, request, view, obj):
        if request.method in SAFE_METHODS:
            return True

        return obj.author == request.user
```

The IsAuthorOrReadOnly class represents a custom permission for controlling access to objects in the API. has_object_permission this method determines whether the user has permission to perform an action on a specific object. SAFE_METHODS contains read-only methods (GET, HEAD, OPTIONS). If the request method is safe, the method returns True, allowing access to any user.

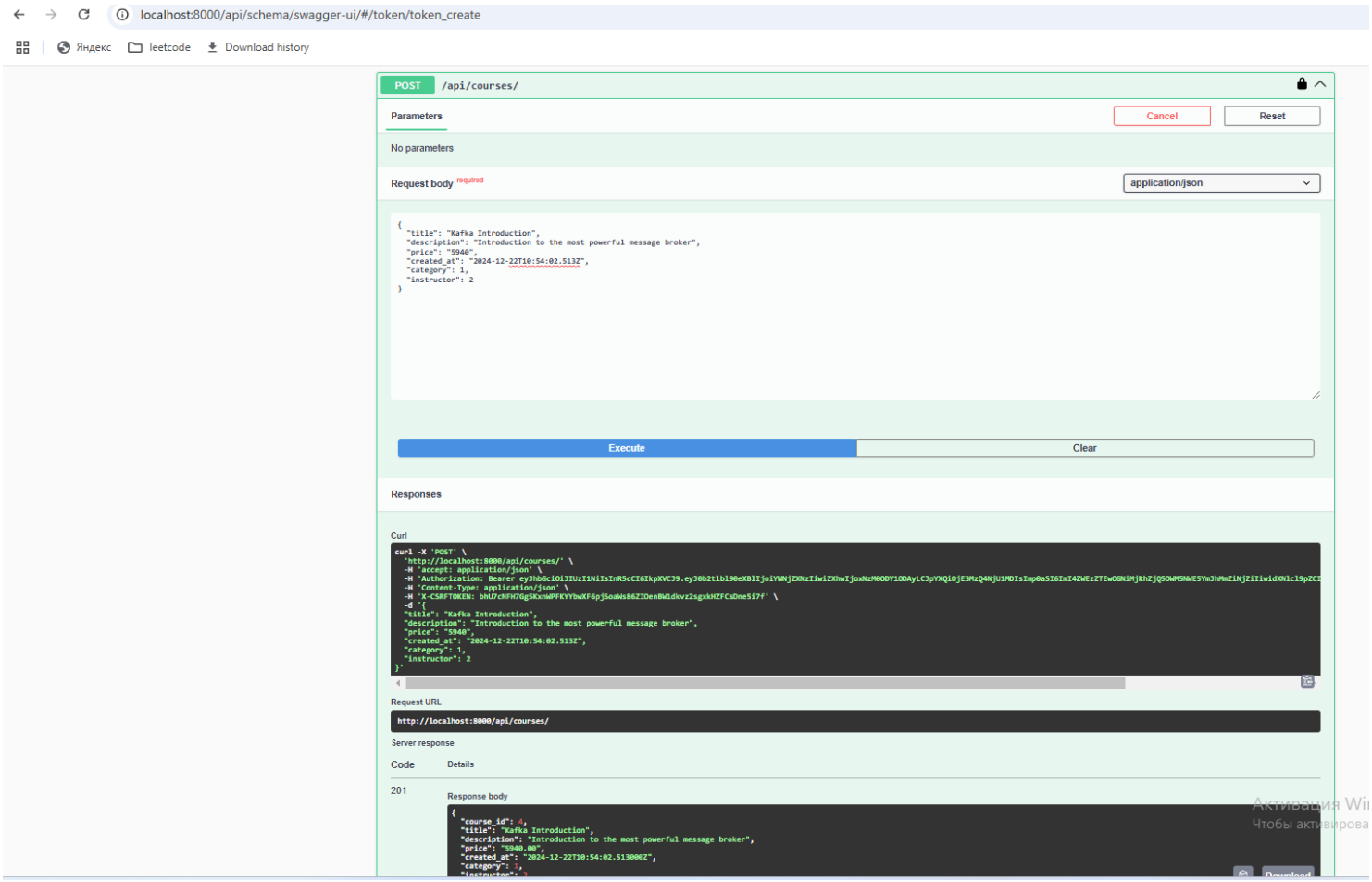
Swagger Api Documentation

API documentation is generated using drf-spectacular, providing interactive Swagger and ReDoc interfaces. The documentation includes detailed descriptions of all endpoints, request/response formats, and authentication requirements (e.g., Bearer token usage). It automatically reflects changes in the codebase, ensuring accuracy. Components such as schema definitions, parameter details, and security schemes make it easier for developers to understand and interact with the API.

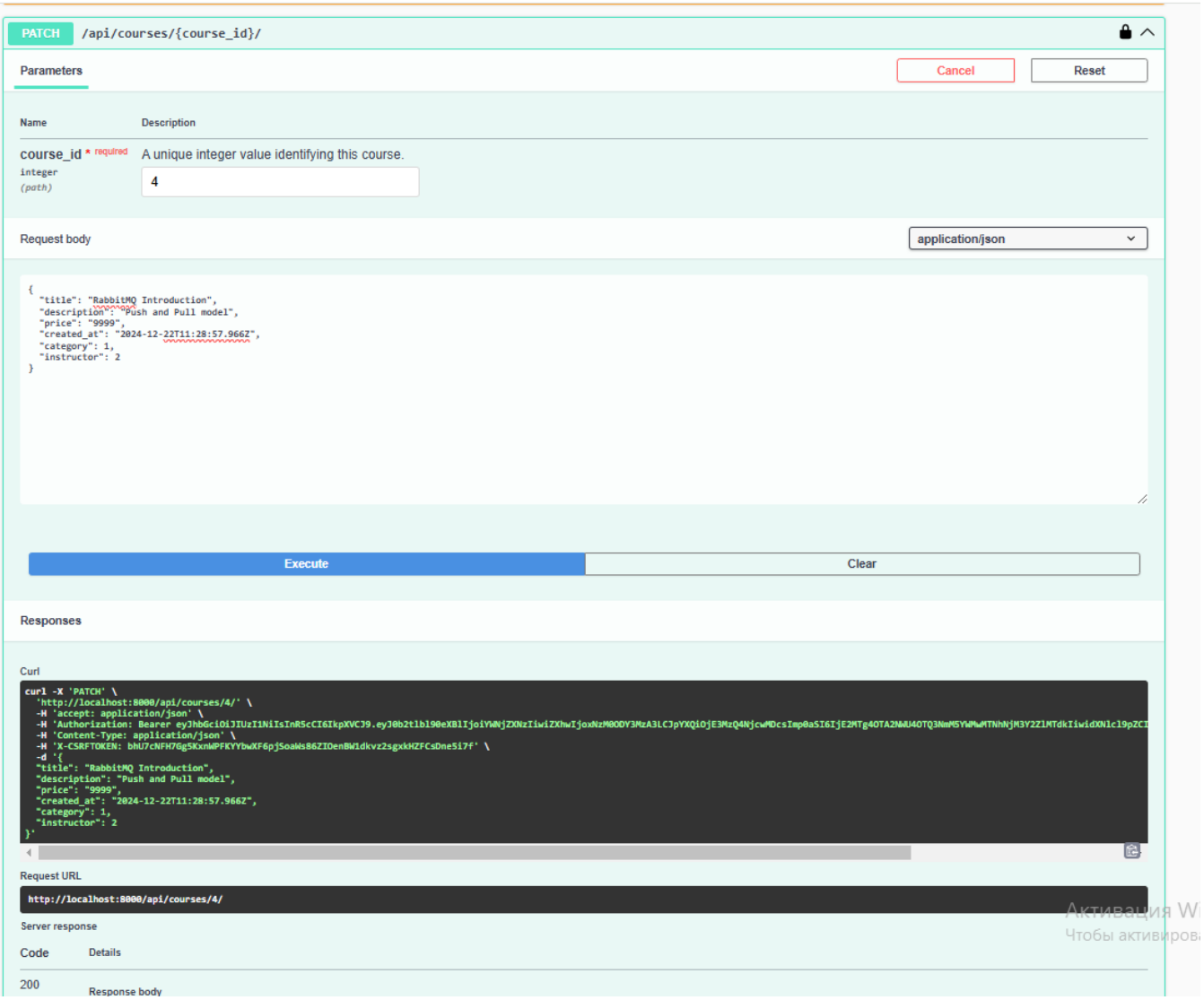




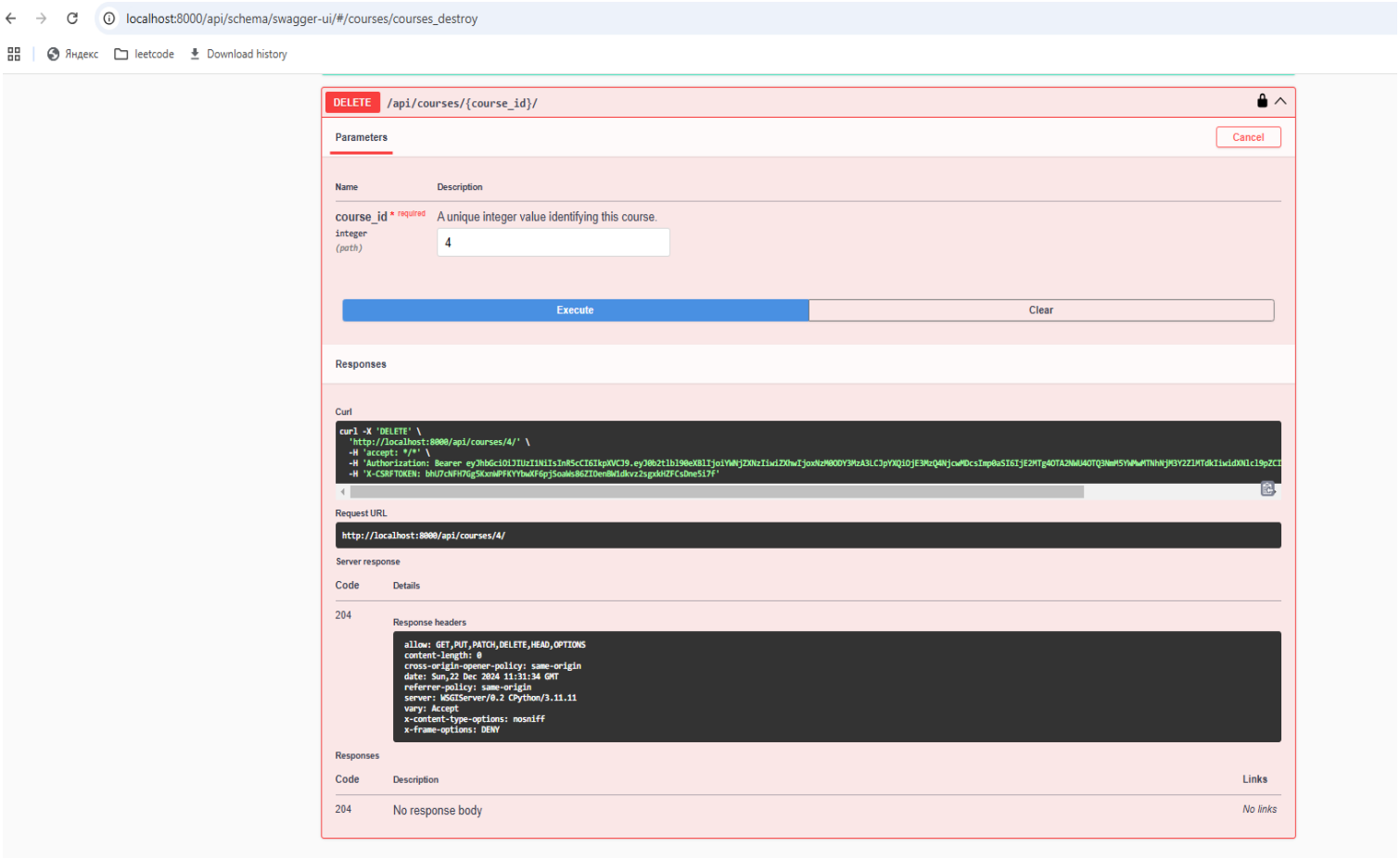
Get all courses



Course creating



Patch course



Delete Course

Conclusion

This project demonstrates how Django and Docker can be combined to create a e-learning platform. Key takeaways include the benefits of containerization, modular architecture, and API-driven design. JWT
authentication and custom permissions safeguards sensitive operations. Future improvements could include real-time notifications.