KAZAKH-BRITISH TECHNICAL UNIVERSITY

Final Project

Web Application Development

E-Learning Platform Development with Django and Docker

Prepared by:
Seitbekov S.
Checked by:
Serek A.

Almaty, 21.12.2024

Table of Contents

Executive Summary

This report will give insight into the design, development, and deployment of the e-learning platform using Django as the back end with Docker for containerization. As designed, the system will allow users to sign up either as a student or instructor, enroll in courses, view lessons, undertake quizzes, and track their progress in learning. It aids instructors in course creation and management, lesson upload, monitoring of student performance. The platform supports different functionalities like user authentication, course reviews, payment, and progress tracking.

Integration of Docker has been done to maintain application execution consistency across environments. The Django Rest Framework exposes APIs that are set to be used by the frontend. A Vue.js frontend consumes these APIs, allowing a seamless and engaging user experience. Throughout the development, scalability, modularity, and readability of code structure have been emphasized. Deploying this platform using Docker renders the platform portable and efficient in scaling.

The following report describes the architecture, design choices, code implementation, challenges encountered, and the solutions adopted during the development of the platform.

Introduction

E-learning platforms are representative of a way in which education is both imparted and sought in the digital era. A properly developed e-learning system will make available tools, both to the students and instructors for structured and, therefore, efficient learning. Such systems need proper development based on sound technology that guarantees their smoothness of performance, scalability, and ease of use.

This project makes use of Django, a high-level Python web framework, to develop the e-learning platform backend. In this regard, Django eases development through the already implemented admin interface, ORM for database management, and support for RESTful APIs using the Django Rest Framework. Deployment and containerization will be done using Docker, so the application is packaged in a way that can be executed reliably in diverse environments without configuration problems.

This project will be able to provide a full e-learning system where students can enroll in courses, view lessons, take quizzes, and track their progress. Instructors can create courses, upload lessons, manage quizzes, and review student performance. It shall also integrate a payment system into the course enrollment process, allowing users to leave reviews.

It includes the back end, the front end, and deployment infrastructure. The system will be using Django at the back end, which will store the application data in PostgreSQL, while Vue.js will be providing a web user interface for the user to interface with at the front end. Docker containerizes the application to make deployment seamless across various environments: development, test, or production.

System Architecture

The e-learning platform is envisioned as a modular system with three main building blocks: the Django backend, the Vue.js frontend, and the PostgreSQL database. These, in turn, are to be used for intuitive, effective, and scalable environments for students and instructors. The

application uses Docker to containerize the application, ensuring that the system will be portable, consistent, and easy to deploy in different environments.

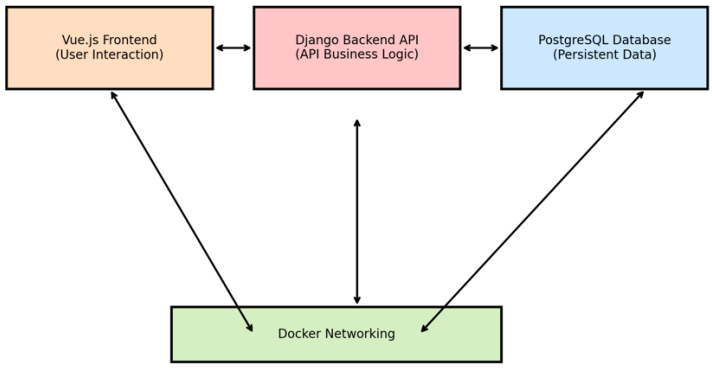


Figure 1. System Architecture

It contains a Django backend, which contains the core of the system: authentication, course and lesson management, quiz submission processing, and user progress tracking. This application makes use of Django Rest Framework (DRF) to expose a set of RESTful APIs. These APIs serve as a bridge between the frontend and the database for easy communication and data exchange. For example, it may be that when a user logs in, the frontend sends a request to the backend, which authenticates the user and provides a token for secure communication in subsequent requests.

The frontend part will be implemented: a Vue.js single-page application web front-end is dynamic, responsive, and interfaces directly to these APIs with the back end; it will be responsible for rendering course content, displaying the progress of users, and providing an interface where quizzes and reviews interactively take place. For example, a student wants to see the progress made in the course. The frontend retrieves that information from an API provided by the backend and displays it, along with visual elements such as progress bars.

PostgreSQL provides the backbone for data storage: securely managing information on users, courses, lessons, enrollments, and quiz results. The relational model makes PostgreSQL enforce data integrity and be efficiently queried. When a student accesses his enrolled courses, the database records the enrollment data of the student and pulls it out.

To make these work in harmony, both the Django backend and PostgreSQL database are containerized with Docker. This makes the application behave consistently during development and testing and into production. Docker Compose orchestrates these containers, essentially letting them talk to each other. For example, the Django back would connect to the PostgreSQL container via a network defined in Docker Compose; no need to define configurations manually.

Table Descriptions

In fact, this e-learning platform uses a well-designed relational database schema to store various entities, including users, courses, enrollments, lessons, reviews, payments, quizzes, and

progress tracking. Each of the tables in this database plays a certain role in assuring functionality and user experience within the system. Each of the following tables is described in detail with its fields and relationships and how they will be applied on the site.

```python
class User(AbstractUser):  Sanzhar
    is_student = models.BooleanField(
        default=False,
        help_text="Designates whether this user should be treated as a student.",
        verbose_name="student status",
    )
    is_instructor = models.BooleanField(
        default=False,
        help_text="Designates whether this user should be treated as an instructor.",
        verbose_name="instructor status",
    )

    def __str__(self) -> str:  Sanzhar
        return self.username

```

Figure 2. User Model.

The Users table contains information about all users on the platform, including both students and instructors. Each user is uniquely identified by a user_id serving as the primary key; other fields include username, email, and password_hash for secure user management, while is_student and is_instructor differentiate between the respective roles of the users. These are boolean fields that allow room for flexibility in creating role-based functionality. For example, an instructor-tagged user can create courses, while the student can enroll in and complete courses.

```python
class Course(models.Model):  Sanzhar
    title = models.CharField(
        max_length=255,
        unique=True,
        blank=False,
        null=False,
        db_index=True,
        verbose_name="Course Title",
    )
    description = models.TextField(
        blank=True,
        null=True,
        verbose_name="Description",
        help_text="Enter the course description.",
        default="",
        max_length=500,
    )
    price = models.DecimalField(
        max_digits=10,
        decimal_places=2,
        verbose_name="Price",
        help_text="Enter the course price.",
    )
    category = models.ForeignKey(
        Category,
        on_delete=models.SET_NULL,
        null=True,
        related_name="courses",
        verbose_name="Category",
        help_text="Select the course category.",
    )
```

Figure 3. Course Model.

The Courses table represents the courses put up on the site; each course has a unique identifier known as course_id. The Course title and description are metadata about the course, the price describes the fee for subscribing to that course. instructor_id is the foreign key that ties that course to its owner in the Users table. This relationship ensures that an instructor only manages courses that belong to them.

```python
class Enrollment(models.Model):  9 usages  Sanzhar
    STATUS_CHOICES = (
        ("active", "Active"),
        ("completed", "Completed"),
        ("cancelled", "Cancelled"),
    )
    user = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
        related_name="enrollments",
        verbose_name="User",
        help_text="Select the user.",
    )
    course = models.ForeignKey(
        Course,
        on_delete=models.CASCADE,
        related_name="enrollments",
        verbose_name="Course",
        help_text="Select the course.",
    )
    enrollment_date = models.DateTimeField(
        auto_now_add=True,
        verbose_name="Enrollment Date",
        help_text="The date and time this course was enrolled.",
    )
    status = models.CharField(
        max_length=20,
        choices=STATUS_CHOICES,
        default="active",
        verbose_name="Status",
        help_text="Select the enrollment status.",
    )
```

Figure 4. Enrollment Model.

The Enrollments table is designed to hold the many-to-many relationship between users and courses. This table keeps track of all the different users who are enrolled in one certain course. In this table, the primary key of the Enrollments table is enrollment_id; the user_id and course_id will represent composite relations. The Status describes active and canceled enrollments. This table will give assurance that this platform is going to be able to capture which course the student is enrolled into.

```python
class Lesson(models.Model):  9 usages  Sanzhar
    course = models.ForeignKey(
        Course,
        on_delete=models.CASCADE,
        related_name="lessons",
        verbose_name="Course",
        help_text="Select the course.",
    )
    title = models.CharField(
        max_length=255,
        unique=True,
        blank=False,
        null=False,
        db_index=True,
        verbose_name="Lesson Title",
    )
    content = models.TextField(
        verbose_name="Content",
        help_text="Enter the lesson content.",
        default="",
        blank=True,
        null=True,
    )
    video_url = models.URLField(
        blank=True,
        null=True,
        verbose_name="Video URL",
        help_text="Enter the video URL.",
    )
```

Figure 5. Lesson Model.

The Lessons table holds information regarding the lessons within each course. Each lesson is tied to a course through the course_id foreign key. The title and content fields contain the instructional material; the video_url field can hold a link to any additional video resources. This would make sense in a way that lessons are grouped under appropriate courses logically.

```python
class Review(models.Model):  8 usages  Sanzhar
    course = models.ForeignKey(
        Course,
        on_delete=models.CASCADE,
        related_name="reviews",
        verbose_name="Course",
        help_text="Select the course.",
    )
    user = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
        related_name="reviews",
        verbose_name="User",
        help_text="Select the user.",
    )
    rating = models.IntegerField(verbose_name="Rating", help_text="Enter the rating.")
    comment = models.TextField(
        verbose_name="Comment",
        help_text="Enter the comment.",
        default="",
        blank=True,
        null=True,
    )
    created_at = models.DateTimeField(
        auto_now_add=True,
        verbose_name="Created At",
        help_text="The date and time this review was created.",
    )

```

Figure 6. Running Migrations.

This allows the user to give their feedback for each course by the Reviews table. It connects every review with some user and some course, by foreign keys. A numeric rating field gives a score given by the user, while the comment is textual feedback. The present data aids in transparency and will enlighten the prospective students with the quality of courses being offered.

```python
class Category(models.Model):  Sanzhar
    name = models.CharField(
        max_length=100,
        unique=True,
        blank=False,
        null=False,
        db_index=True,
        verbose_name="Category Name",
        help_text="Enter the category name.",
    )
    description = models.TextField(
        blank=True,
        null=True,
        verbose_name="Description",   Sanzhar, 18.12.2024, 23:14 • added: Final Projec
        help_text="Enter the category description.",
        default="",
        max_length=500,
    )
```

Figure 7. Category Model.

The Categories table helps to classify courses into distinct categories and therefore allows users to quickly find courses of interest. Each category has a name and description. The category_id is used as a foreign key in the Courses table for consistent categorization.

```python
class Payment(models.Model):  7 usages  Sanzhar
    STATUS_CHOICES = (
        ("pending", "Pending"),
        ("completed", "Completed"),
        ("failed", "Failed"),
    )
    user = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
        related_name="payments",
        verbose_name="User",
        help_text="Select the user.",
    )
    amount = models.DecimalField(
        max_digits=10,
        decimal_places=2,
        verbose_name="Amount",
        help_text="Enter the payment amount.",
    )
    payment_date = models.DateTimeField(
        auto_now_add=True,
        verbose_name="Payment Date",
        help_text="The date and time this payment was made.",
    )
    status = models.CharField(
        max_length=20,
        choices=STATUS_CHOICES,
        default="pending",
        verbose_name="Status",
        help_text="Select the payment status.",
    )
```

Figure 8. Payment Model.

The Payments table is responsible for keeping track of the different transactions made by users for enrollments in courses. These fields assure traceability through the payment_id, user_id, and course_id. The amount field will contain the value of the transaction, and payment_date and status will denote when the payment was made and what its status is. This table is critical in managing the monetization of the platform.

```python
class Quiz(models.Model):  Sanzhar          Sanzhar, 18.12.2024, 23:14 • added: Final Project
    course = models.ForeignKey(
        Course,
        on_delete=models.CASCADE,
        related_name="quizzes",
        verbose_name="Course",
        help_text="Select the course.",
    )
    title = models.CharField(
        max_length=255,
        unique=True,
        blank=False,
        null=False,
        db_index=True,
        verbose_name="Quiz Title",
    )
    total_marks = models.IntegerField(
        verbose_name="Total Marks", help_text="Enter the total marks."
    )


class QuizQuestion(models.Model):  6 usages  Sanzhar
    quiz = models.ForeignKey(Quiz, on_delete=models.CASCADE, related_name="questions")
    question_text = models.TextField(
        verbose_name="Question Text", help_text="Enter the question text."
    )
    option_a = models.CharField(
        max_length=255, verbose_name="Option A", help_text="Enter option A."
    )
    option_b = models.CharField(
        max_length=255, verbose_name="Option B", help_text="Enter option B."
    )
    option_c = models.CharField(
        max_length=255, verbose_name="Option C", help_text="Enter option C."
    )
    option_d = models.CharField(
```

Figure 9. Quiz & QuizQuestion Models.

The Quizzes table stores the assessments assigned for courses, whereas the definition of the questions appearing in each quiz is stored in the QuizQuestions table. Each quiz corresponds to one course via the course_id foreign key. The maximum marks a student can achieve for the quiz are recorded under the total_marks field within the Quizzes table. The QuizQuestions table provides fields for the question and options text, and then the correct answer.

```python
class UserProgress(models.Model):  7 usages  Sanzhar  Sanzhar, 18.12.2024, 23:14
    user = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
        related_name="progress",
        verbose_name="User",
        help_text="Select the user.",
    )
    course = models.ForeignKey(
        Course,
        on_delete=models.CASCADE,
        related_name="progress",
        verbose_name="Course",
        help_text="Select the course.",
    )
    completed_lessons = models.JSONField(
        default=list,
        verbose_name="Completed Lessons",
        blank=True,
        null=True,
    )
    quiz_scores = models.JSONField(
        default=dict,
        verbose_name="Quiz Scores",
        help_text="Enter the quiz scores.",
        blank=True,
        null=True,
    )
```

Figure 10. UserProgess Model.

The UserProgress table monitors students' progress in their enrolled courses. The completed_lessons field is a JSON array storing lesson IDs completed by the user, and the quiz_scores field records quiz performance. This table facilitates tracking and visualizing individual learning journeys.

These database tables are indeed the backbone of the electronic learning platform, whereby careful design and determination have captured the essentials to keep every vital datum required to manage both users and courses, besides their interaction with each other. Relationships between tables establish data integrity that guarantees the consistency of this very learning process in being effective.

Intro to Containerization: Docker

Containerization revolutionized software deployment by creating an awesomely lightweight, portable, self-sufficient environment for applications to execute in. In this work, Docker was used for the encapsulation of the e-learning platform to guarantee predictability of its behavior during development, testing, and production. Docker containers bundle application code with its operating system, libraries, and dependencies, and eliminate the "it works on my machine" problem.

Unlike traditional virtual machines, Docker containers share an operating system kernel of the host. This means that Docker Containers start much faster and will not be as resource intensive. Further, this efficiency shall enable the developers to create isolated environments for different parts of the application such that none of them is interfering with another. In the above case, though

the back-end Django Application and the PostgreSQL database were running in different containers, the networking features of Docker allowed smooth communication between the two.

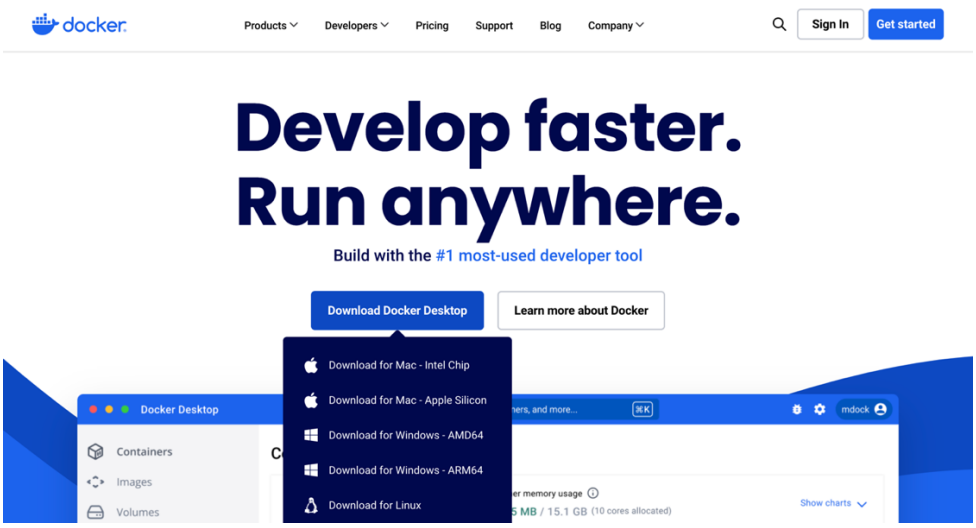


Figure 11. Docker Website.

First, for starting to work with Docker, it had been installed on a development machine. Having been downloaded onto an operating system, Docker was checked by execution of the following command:

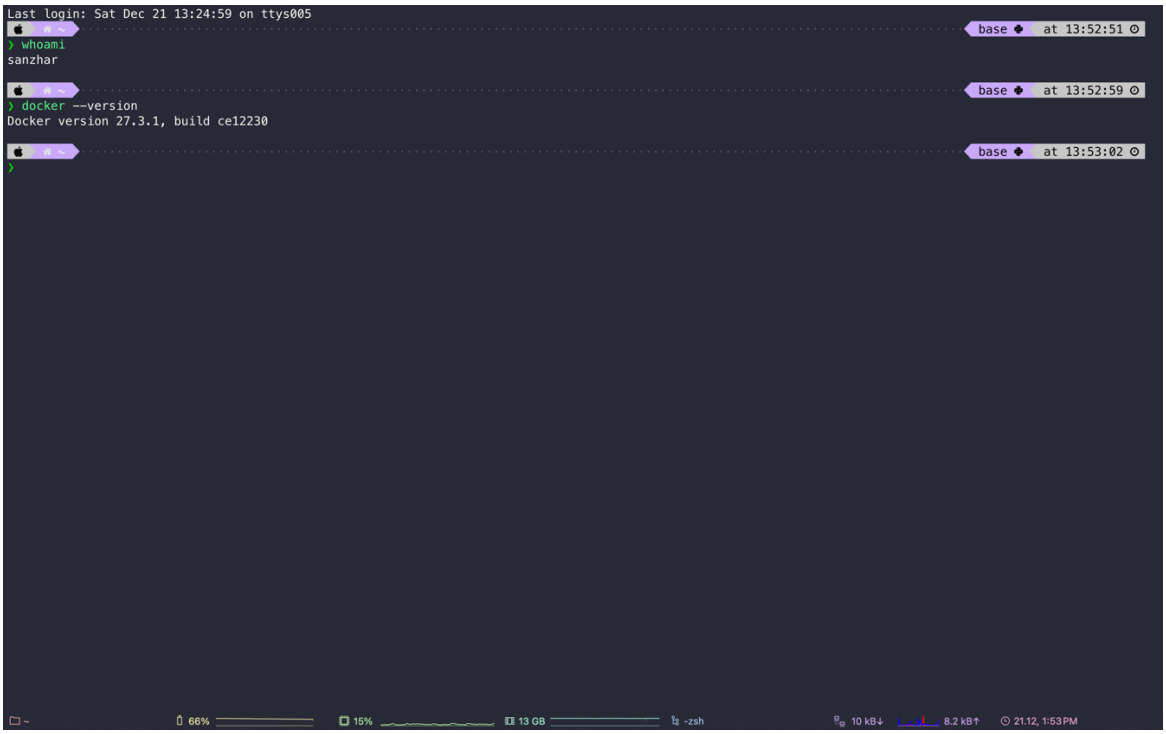


Figure 12. Docker Version.

This command outputs the installed Docker version, confirming successful installation. Following this, the "Hello World" container was executed to validate Docker's functionality:

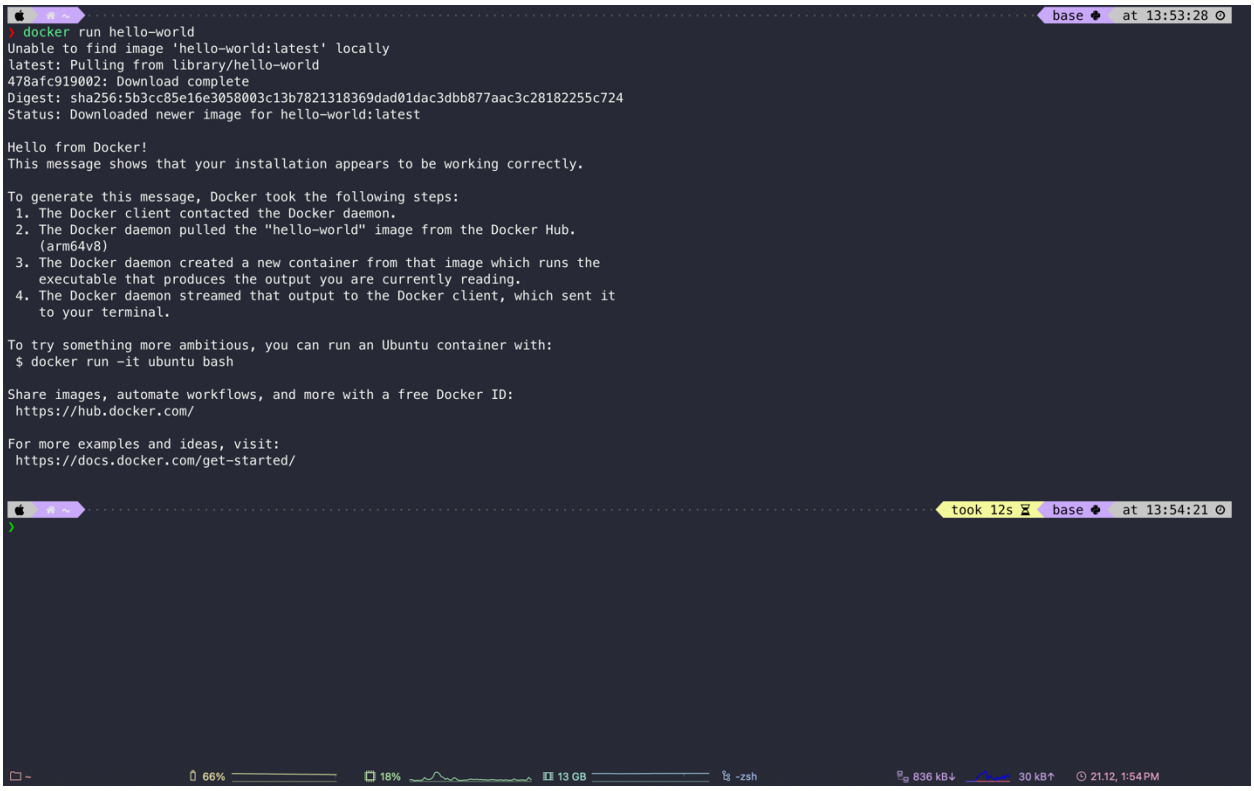


Figure 13. Docker Hello World.

This command pulls the “hello-world” image, starts a container, and then runs it. If everything in Docker is working right, the container outputs a friendly welcome message, through which it indicates that Docker can be used. When its validity was confirmed, containerization of the Django application using a Dockerfile would proceed.

Dockerfile

The Dockerfile is considered the backbone of containerization. A Dockerfile is a script that consists of instructions or commands to build an application in a Docker image. Multistage build was an approach used to create this lightweight and, at the same time, efficient container. This helps in differentiating the environments of a build versus runtime and only includes what is required for running the application inside the final image. Following is the Dockerfile used for the e-learning platform:

```dockerfile
FROM python:3.12-slim AS builder

ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

RUN apt-get update && apt-get install -y --no-install-recommends \
    build-essential \
    libpq-dev \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .
```

Figure 14. Docker Build Stage.

The builder stage uses the python:3.12-slim image, which has been selected for its small size. Various environment variables, like PYTHONDONTWRITEBYTECODE and PYTHONUNBUFFERED, are configured to optimize Python's behavior in a containerized environment. Basic build tools like libpq-dev are installed for building Python dependencies. Next is the configuration of the runtime environment itself:

```dockerfile
FROM python:3.12-slim

ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

RUN apt-get update && apt-get install -y --no-install-recommends \
    netcat-openbsd \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /app

COPY --from=builder /usr/local/lib/python3.12/site-packages /usr/local/lib/python3.12/site-packages
COPY --from=builder /usr/local/bin /usr/local/bin

COPY --from=builder /app /app

EXPOSE 8000

CMD ["gunicorn", "elearning.wsgi:application", "--bind", "0.0.0.0:8000"]
```

Figure 15. Docker Final Stage.

It is in this stage that the final image is built, removing all unnecessary build-time dependencies and leaving only the components that are needed for the application to serve. This helps in improving security and reducing the size of the image, hence efficient deployment.

Docker-Compose

Docker Compose is a powerful tool for developers to define and manage multi-container applications using one single YAML file. For the e-learning platform, Docker Compose orchestrates several services like Django backend, PostgreSQL database, and static/media file storage so that they interact well with each other and have consistent development environments.

The docker-compose.yaml defines services, networks, and volumes necessary for the application. These different services will represent different parts of the e-learning platform. The configuration explains the relationship between these services: there are three main sections- Services, Volumes, and Networks.

1. Services: Describes which containers shall be built and started, which means web (Django Backend) and db for a PostgreSQL database.
2. Volumes: The volumes are persistent storage for data, mainly database records and user-generated files.
3. Networks: This configures an isolated network, allowing secure communications between containers.

Here is the structure of the docker-compose.yaml file:

```yaml
services:
  web:
    build: .
    ports:
      - "8000:8000"
    depends_on:
      - db
    volumes:
      - static_data:/app/static
      - media_data:/app/media
    networks:
      - elearning_network
    env_file:
      - .env
  Sanzhar, 18.12.2024, 23:14 • added: Final Project
  db:
    image: postgres:17-alpine
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - elearning_network

volumes:
  static_data:
  media_data:
  postgres_data:

networks:
  elearning_network:
    driver: bridge
```

Figure 16. Docker Compose File.

Web service runs Django backend. The build directive here stipulates that the service uses the Dockerfile in the root of the project to build the container. Service exposes port 8000 so that the application will be accessible at http://localhost:8000.

The depends_on property ensures that the db service starts before the backend, preventing runtime errors due to the unavailability of the database. The container mounts two volumes: one for static files, static_data, and another for user-uploaded files, media_data. These volumes ensure persistence across container restarts.

An .env file is used to store sensitive data such as database credentials. This keeps secrets out of the docker-compose.yaml file and adds both security and flexibility.

```
POSTGRES_DB=elearning_db
POSTGRES_USER=elearning_user
POSTGRES_PASSWORD=elearning_password
POSTGRES_HOST=localhost
POSTGRES_PORT=5432
SECRET_KEY='django-insecure-1a0p4×3&(-e-q_6=i=yg73q4#7_8×8p)9u+x88@l1m)3vr)p$w'
DEBUG=True
```

Figure 17. Env File.

The db service uses a light image called postgres:14-alpine, which provides a slim and lightweight environment for running a database. The environment variables for the PostgreSQL database are taken from the.env file through the variables named POSTGRES_USER, POSTGRES_PASSWORD, and POSTGRES_DB, making it secure and flexible.

Volume will persist the data within postgres_data volume to avoid container restarts or redeployment and keep its former database data. The db service will also join the network by assigning the elearning_network for further allowing access on safe communication with the backend.

E-learning_network is specified as a bridge network. It means that the containers cannot be reached from the outside, but in return, the containers may talk to each other safely. The web service uses here the db service with a name db; in that respect, the configuration of how the container will interact or the container interaction becomes easy.

Some volumes used here for data persistent storage of critical information include:

1. postgres_data: for PostgreSQL database records.
2. static_data: Django static files, such as CSS and JavaScript
3. media_data: User uploaded files, like images and documents.

To build and start all services, the following command is used:

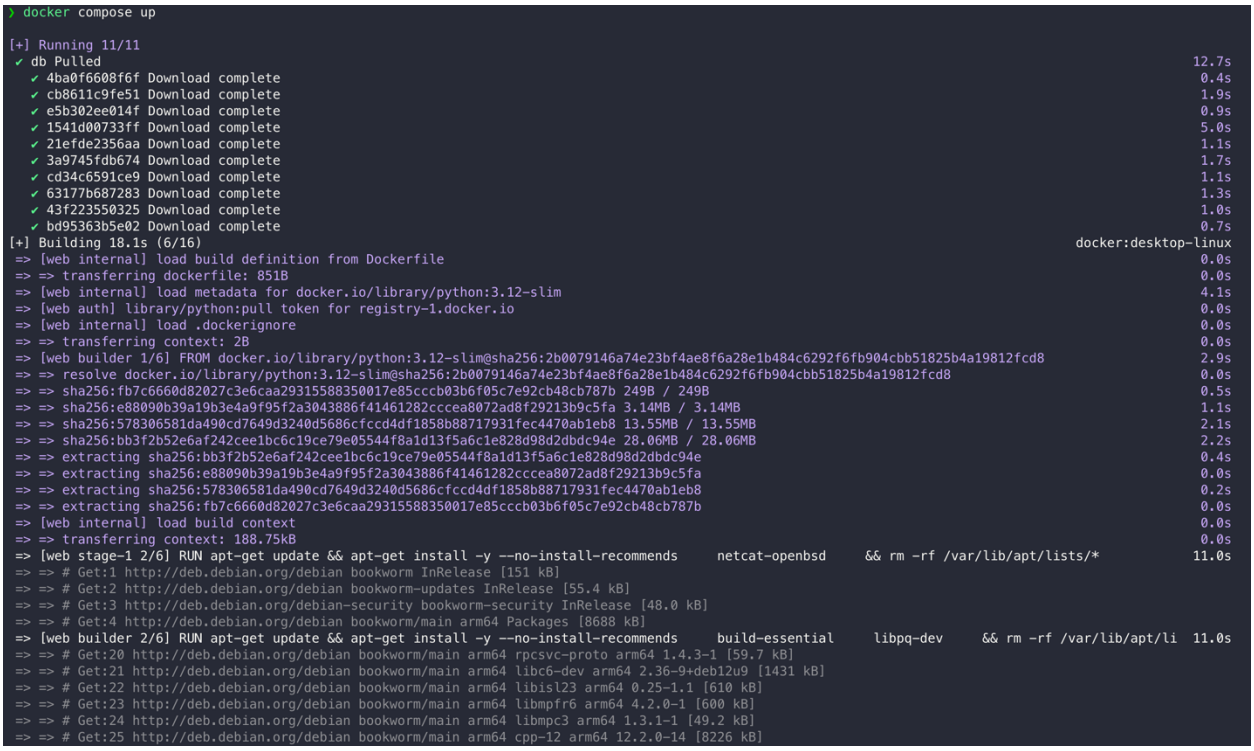


Figure 18. Docker Compose Build.

This command builds the Docker images and starts the containers, ensuring they are interconnected and running in the defined network. To stop the containers while retaining data, the following command is used:

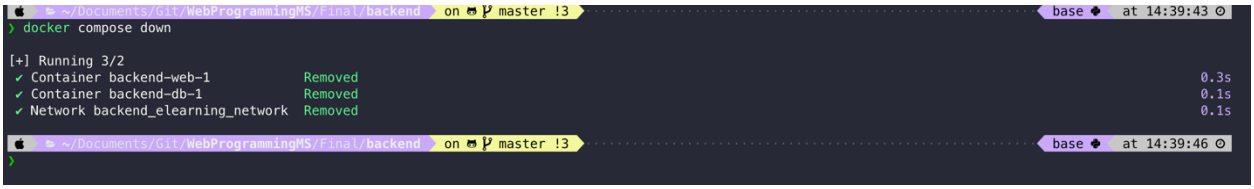


Figure 19. Turning Down Docker Containers.

By default, when Docker Compose starts the application, the Django backend (the web service) will connect to the PostgreSQL database on the default host, which is the name of the service: db over the elearning_network. Static and media files will be persisted in their respective volumes. For example, if a user uploads some course resource-say a PDF-that gets stored in the media_data volume. Similarly, every action taken with the database, the creation of a user, or the submission of a quiz, has persisted into the postgres_data volume.

I use Docker Compose to get much easier management of your e-learning platform. A sole configuration file for services dictates consistency in the environments-from development and testing right through to production. Light orchestration of many containers brings added productivity and reduced human-made error risks.

In general, Docker Compose plays an integral role in the deployment and operation of the e-learning platform by providing a capable and efficient framework for managing the multi-container architecture of this platform. It ensures the platform is scalable, portable, resilient, and the user experience is not hampered.

Docker Networking and Volumes

Communication among containers and durable storage of data is important for the functionality and reliability of a multi-container application like the e-learning platform. Docker networking provides the ability for container communication, while Docker volumes provide durable storage for data such as user-generated content, static files, and database records.

The Docker networking in this project was set up with the creation of a bridge network named elearning_network. This private network connects the services defined in the docker-compose.yaml file:

```
31
32  networks:
33    elearning_network:
34      driver: bridge
35
```

Figure 20. Docker Compose Network.

The Django web application, also called the web service, and the PostgreSQL database, referred to as the db service. The good thing with the bridge network is that, in general, containers can talk to each other securely by using their service names with no dependency on IP addresses that might change upon restarts or redeployment of containers.

The following command was executed just to ensure that indeed a network was created:

```
~/Documents/Git/WebProgrammingMS/Final/frontend on master ?1
❯ docker network ls

NETWORK ID     NAME                          DRIVER    SCOPE
b6db11437a92   backend_elearning_network     bridge    local
9a988e54a8c8   bridge                        bridge    local
696d2b28f62d   host                          host      local
b3ad457d7528   none                          null      local
```

Figure 21. List Of Docker Networks.

This lists all Docker networks on the system; thus, proving the existence of elearning_network. To see details about the network-connected containers and configuration settings, the docker network inspect command was used:

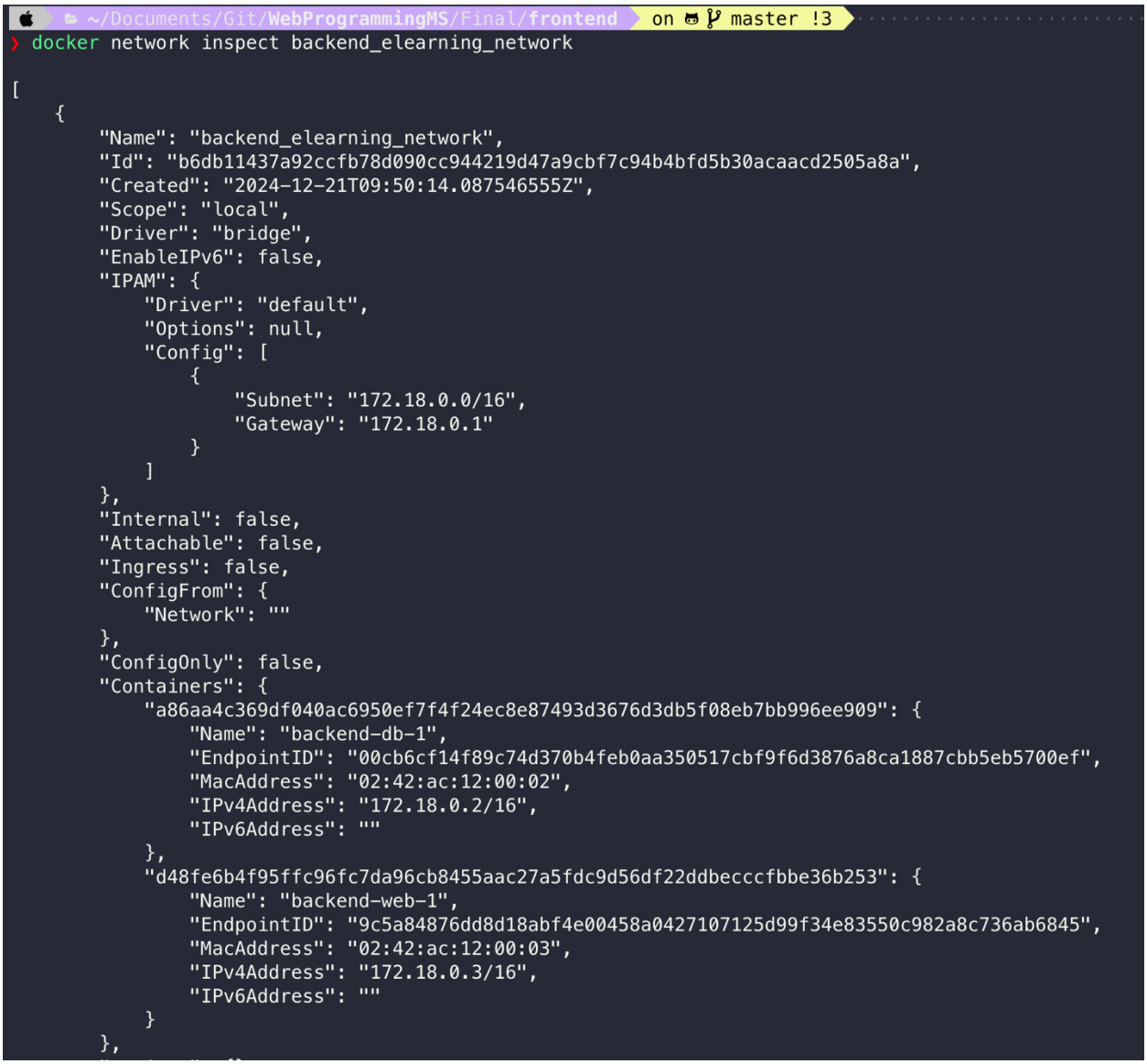


Figure 22. Docker Network Inspection.

The output has been included to show some of the critical information: driver type-for example, bridge-connected container services, and IP Address Ranges-so one is sure that the network works as it should. Such a network will, for example, allow easy communication between backend and database when the Django application queries the database about course content or saves user progress.

Docker volumes ensure that data have persisted, which is quite important for components like PostgreSQL database and static/media files of Django. Without it, every stop, removal, or redeployment of a container would result in lost data.

Volumes for the project were defined in docker-compose.yaml to the following:

1. postgres_data: This volume persists data for PostgreSQL databases. It ensures that records of the database, such as information about users, courses, and enrollment statuses, persist even after container restarts.

2. static_data: This volume is used by Django to store static files such as CSS and JavaScript, which are required at the frontend for proper appearance and functionality.
3. media_data: This is meant for user-uploaded content, such as profile pictures and course resources like PDFs and videos.

Here's how volumes were defined inside docker-compose.yaml:

```
27    volumes:
28      static_data:
29      media_data:
30      postgres_data:
31
```

Figure 23. Docker Compose Volumes.

These volumes are mounted into the respective services in the docker-compose.yaml configuration. For example, the web service mounts static_data and media_data, while the db service mounts postgres_data.

This means that when a user uploads, say, a profile picture, it goes into the media_data volume. In case the web container is replaced, the uploaded file would still be accessible. Likewise, static files collected using Django's collectstatic command persisted to the static_data volume and, hence, available across container restarts.

```
8       volumes:
9         - static_data:/app/static
10        - media_data:/app/media
11      networks:
12        - elearning_network
```

Figure 24. Docker Volumes In Web Application.

Persistence of the records on postgres_data is very important because it holds user accounts, course enrollment, and quiz scores persistently. For instance, in the case of a student who has completed his quiz, the score that he got will be recorded in the PostgreSQL database through the volume postgres_data to make sure that even if the db container goes for a restart, the score remains.

In verifying the volume and whether the volume is working fine, a set of commands used would be as follows:

```
~/Documents/Git/WebProgrammingMS/Final/frontend on master !3
❯ docker volume ls
DRIVER    VOLUME NAME
local     backend_media_data
local     backend_postgres_data
local     backend_static_data
```

Figure 25. List Of Docker Volumes.

These commands will confirm that volumes have been correctly created and are linked to the right containers.

```
~/Documents/Git/WebProgrammingMS/Final/frontend on master !3
❯ docker volume inspect backend_postgres_data

[
    {
        "CreatedAt": "2024-12-21T09:35:54Z",
        "Driver": "local",
        "Labels": {
            "com.docker.compose.project": "backend",
            "com.docker.compose.version": "2.30.3",
            "com.docker.compose.volume": "postgres_data"
        },
        "Mountpoint": "/var/lib/docker/volumes/backend_postgres_data/_data",
        "Name": "backend_postgres_data",
        "Options": null,
        "Scope": "local"
    }
]
```

Figure 26. Docker Volume Inspect.

Similarly, Docker networking-related commands like docker network inspect confirm that all services in elearning_network can talk securely.

Networking in Docker will be used to ensure proper service communication in the e-learning platform, while volumes will be utilized for durable storage of critical data. It allows the backend, database, and other services to work cohesively as one unit; volumes provide the persistence needed for maintaining user and application data across container lifecycles. This is crucial for developing a robust, scalable, and production-ready e-learning platform.

Django

Django is a high-level Python Web Framework that enables rapid development, clean, and pragmatic design. It favors reusability and maintainability; thus, ideal for the said e-learning platform. Django follows one of the popular architectural patterns, MVT-Model-View-Template; in which application logic, user interface, and data handling vary in layers, assuring maintainability and extensibility while this platform is grown.

The core functionality is managed in Django for the e-learning platform-for instance, user authentication, course management, and lesson handling. This framework takes away some of the difficulties of development with pre-made components: an admin interface, ORM, and middleware.

Below is the command used to create the Django project:

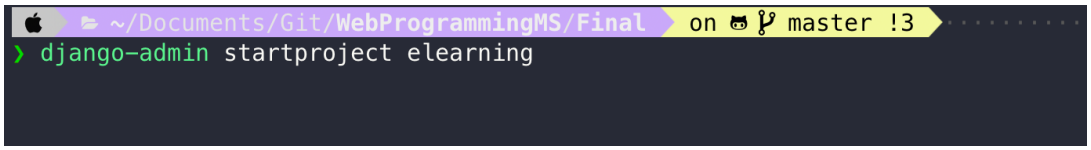


Figure 27. Starting Django Project.

This initializes a new Django project with a default directory structure. The main components include the settings.py file for configuration, urls.py for routing, and manage.py for administrative tasks.

Once the project is set up, apps are created to encapsulate specific functionalities. For instance, separate apps were created for managing courses, users, lessons, and reviews. The command to create an app is as follows:

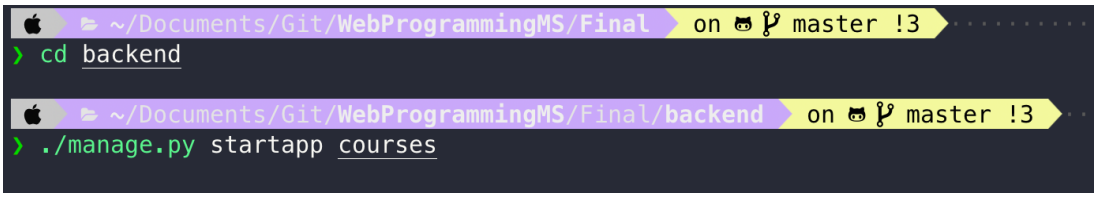


Figure 28. Starting Django Application.

Each app is registered in the INSTALLED_APPS section of settings.py, ensuring Django recognizes and integrates the app into the project.

```
39  INSTALLED_APPS = [
40      "django.contrib.admin",
41      "django.contrib.auth",
42      "django.contrib.contenttypes",
43      "django.contrib.sessions",
44      "django.contrib.messages",
45      "django.contrib.staticfiles",
46      "rest_framework",
47      "corsheaders",
48      "users",
49      "courses",
50      "enrollments",
51      "lessons",
52      "reviews",
53      "categories",
54      "payments",
55      "quizzes",
56      "progress",
57      "drf_spectacular",
58  ]
```

Figure 29. Installed Apps for Django Project.

Django’s robust features, such as the built-in authentication system and ORM, reduce development time and ensure security. For example, user credentials are hashed by default, providing an extra layer of protection

Models

Models are one of the core things that Django provides. They help a developer define the structure of the database in Python code. Each model corresponds with one table in the database, and Django's ORM insulates the developer from needing to know how to create that schema in the database.

In this e-learning platform, models were used to design the main entities of the system that were created, including those of a User, Course, Enrollments, and Lesson. Next is the Course model for defining the structures of the courses in this system. Each field on this model corresponds to each column that will be in the database.

```
class Course(models.Model):  Sanzhar        Sanzhar, 18.12.2024, 23:14 • added: Final Project
    title = models.CharField(
        max_length=255,
        unique=True,
        blank=False,
        null=False,
        db_index=True,
        verbose_name="Course Title",
    )
    description = models.TextField(
        blank=True,
        null=True,
        verbose_name="Description",
        help_text="Enter the course description.",
        default="",
        max_length=500,
    )
    price = models.DecimalField(
        max_digits=10,
        decimal_places=2,
        verbose_name="Price",
        help_text="Enter the course price.",
    )
    category = models.ForeignKey(
        Category,
        on_delete=models.SET_NULL,
        null=True,
        related_name="courses",
        verbose_name="Category",
        help_text="Select the course category.",
    )
    created_at = models.DateTimeField(
        auto_now_add=True,
```

Figure 30. Course Model.

1. title - CharField with a maximum of 255 characters and a unique=True constraint, so no two courses would have the same title.
2. description - TextField for detailed course descriptions.
3. price - DecimalField for storing course fees with precision defined by max_digits and decimal_places.
4. category - ForeignKey to a category to link courses to a category for course categorization.
5. instructor is another ForeignKey, this time relating the course to its instructor represented by the Django-provided User model.
6. created_at is a DateTimeField that auto-sets when a course is created.

Relationships between models are essential for handling the interdependent data of the platform. For example, one Course may have many Lessons, but each Lesson will belong to only one Course. This is modeled using Django's foreign key relationships. Below is the Lesson model, which describes the structure of a lesson:

```python
class Lesson(models.Model):  9 usages  Sanzhar
    course = models.ForeignKey(
        Course,
        on_delete=models.CASCADE,
        related_name="lessons",
        verbose_name="Course",
        help_text="Select the course.",
    )
    title = models.CharField(
        max_length=255,
        unique=True,
        blank=False,
        null=False,
        db_index=True,
        verbose_name="Lesson Title",
    )
    content = models.TextField(
        verbose_name="Content",
        help_text="Enter the lesson content.",
        default="",
        blank=True,
        null=True,
    )
    video_url = models.URLField(
        blank=True,
        null=True,
        verbose_name="Video URL",
        help_text="Enter the video URL.",
    )

```

Figure 31. Lesson Model.

This model features:

1. The course field: Many-to-one relationship with the Course model, where each lesson should be a part of exactly one course.

2. The video_url field: Optional; a lesson might not have a video attached to it - expressed with blank=True, null=True.

The Enrollments table monitors the many-to-many relationships between the user and the course. It logs a user being enrolled in a course, where the enrollment_id serves as the primary key while the user_id and course_id establish a composite relationship. Status defines whether the enrollment is active or canceled. This table guarantees that the system can capture the courses for which a student has enrolled.

```
class Enrollment(models.Model):  9 usages  Sanzhar
    STATUS_CHOICES = (
        ("active", "Active"),
        ("completed", "Completed"),
        ("cancelled", "Cancelled"),
    )
    user = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
        related_name="enrollments",
        verbose_name="User",
        help_text="Select the user.",
    )
    course = models.ForeignKey(
        Course,
        on_delete=models.CASCADE,
        related_name="enrollments",
        verbose_name="Course",
        help_text="Select the course.",
    )
    enrollment_date = models.DateTimeField(
        auto_now_add=True,
        verbose_name="Enrollment Date",
        help_text="The date and time this course was enrolled.",
    )
    status = models.CharField(
        max_length=20,
        choices=STATUS_CHOICES,
        default="active",
        verbose_name="Status",
        help_text="Select the enrollment status.",
    )
```

Figure 32. Enrollment Model.

The Reviews table enables users to give feedback on the courses. Every review is related to a user and a course by foreign keys. The rating field, being numeric, stands for the user's score, while the comment field stands for textual feedback. It keeps transparency going and helps incoming students get a view about the course quality.

```python
class Review(models.Model):  8 usages  Sanzhar
    course = models.ForeignKey(
        Course,
        on_delete=models.CASCADE,
        related_name="reviews",
        verbose_name="Course",
        help_text="Select the course.",
    )
    user = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
        related_name="reviews",
        verbose_name="User",
        help_text="Select the user.",
    )
    rating = models.IntegerField(verbose_name="Rating", help_text="Enter the rating.")
    comment = models.TextField(
        verbose_name="Comment",
        help_text="Enter the comment.",
        default="",
        blank=True,
        null=True,
    )
    created_at = models.DateTimeField(
        auto_now_add=True,
        verbose_name="Created At",
        help_text="The date and time this review was created.",
    )
```

Figure 33. Review Model.

The Categories table separates courses into distinct categories, making the process of finding the course one is interested in much easier. Each category is represented with a name and description. The category_id becomes a foreign key in the Courses table to maintain consistency across categorizations.

```python
class Category(models.Model):  Sanzhar
    name = models.CharField(
        max_length=100,
        unique=True,
        blank=False,
        null=False,
        db_index=True,
        verbose_name="Category Name",
        help_text="Enter the category name.",
    )
    description = models.TextField(
        blank=True,
        null=True,
        verbose_name="Description",    Sanzhar, 18.12.2024, 23:14 • added: Final Project
        help_text="Enter the category description.",
        default="",
        max_length=500,
    )
```

Figure 34. Category Model.

The Payments table records the different transactions of users enrolling in certain courses. It contains a payment_id, user_id, and course_id to ensure traceability. The Amount field shows the amount that was transacted, whereas Payment_date and Status indicate the time and result of that payment. This table will be important in managing monetization on the platform.

Figure 35. Payment Model.

The Quizzes table will store assessments related to courses, and the QuizQuestions table defines the questions within each quiz. Each quiz is then associated with a course using the course_id foreign key. The total_marks field in the Quizzes table stores the maximum possible score for the quiz. The QuizQuestions table defines fields for question text, multiple-choice options, and the correct answer.

Figure 36. Quiz Model.

Once models are defined, they are translated into database schema using Django's migration system. The process involves two commands:

1. Makemigration generates migration files

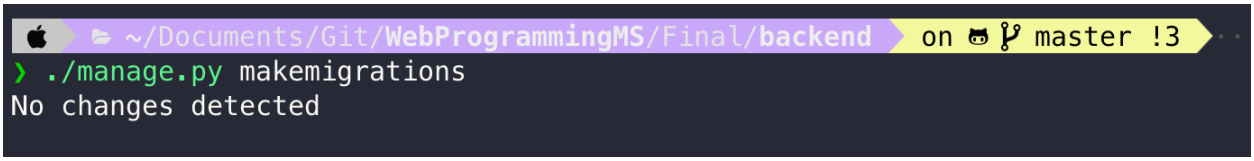


Figure 37. Django Makemigraions Command.

2. Migrate applies the migration to database

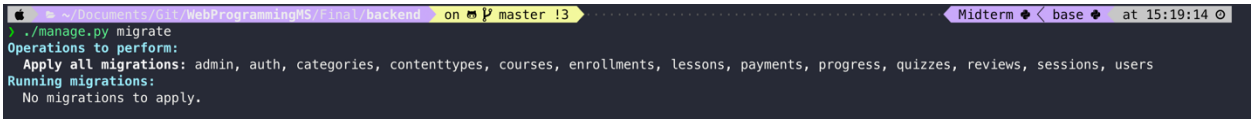


Figure 38. Django Migrate Command.

This ensures the database structure aligns with the models defined in Python code.

Views

In Django, views sit between the models and the templates; they, in a way, contain the business logic of the application. It receives HTTP requests, acts on it through models by either pulling data out of databases or by modifying them, and sends the HTTP response back. For views, Django has supported both FBVs (Function-Based-Views) and CBVs.

FBVs are simple and grant the developer fine-grained control over how requests should be handled. They are appropriate for simple tasks or whenever there is a need to execute custom logic. Here's an example of an FBV that returns all the courses available within the database:

```
def course_list(request):  new *
    courses = Course.objects.all()
    return render(request, template_name: 'courses/course_list.html', context: {'courses': courses})
```

Figure 39. FBV Course List

In this example:

1. The request object carries metadata about the HTTP request.
2. Course.objects.all() retrieves all records in the Course model.
3. The render function takes the template-courses/course_list.html and the data-{'courses': courses} and renders an HTML response.

This structure makes FBVs suitable to use in simple use cases like listing, updating or deleting records.

Class-based views are just another more modular and re-usable way of managing requests. They capture commonality, such as listing or creating or updating objects and make it easy to change that behavior by overriding class methods. Here is a CBV equivalent to the course detail view described above.

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

Figure 40. CBV CourseDetailView.

Here are the important aspects of this CBV:

1. The model attribute says what model the view is intended to act upon (in this case Course).
2. The attribute template_name above indicates that it will render an HTML file called course_detail.html.
3. The context_object_name 'course' defines the name of the variable that the template will use to access the course instance.

CBVs are better for maintainability. Because a single CBV can cover filtering, sorting and pagination with minimal customization.

A real-world Django application generally uses a mix of FBVs and CBVs to balance between simplicity and extensibility. For example, the common tasks of displaying detail or listing might be implemented using CBVs, with FBVs managing highly customized logic. Below is an example of an FBV that manages course creation and shows how user input is processed:

```
def create_course(request):  new *
    if request.method == 'POST':
        form = CourseForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('course_list')
    else:
        form = CourseForm()
    return render(request, template_name: 'courses/create_course.html', context: {'form': form})
```

Figure 41. Create Course View.

This view:

1. Checks whether the request method is POST.
2. If true, it instantiates the form with the data and performs the validation using form.is_valid().
3. If the form is valid, the save() method commits the new Course record to the database.
4. Then it redirects to course_list page.
5. If the request method is not POST it displays an empty form for user to fill.

Each view must be mapped to an URL pattern in the file urls.py. Example:

```python
from django.urls import path
from .views import course_list, CourseDetailView, create_course

urlpatterns = [
    path( route: '', course_list, name='course_list'),
    path( route: 'course/<int:pk>/', CourseDetailView.as_view(), name='course_detail'),
    path( route: 'course/create/', create_course, name='create_course'),
]
```

Figure 42. Course URLs.

1. the following path(“”, course_list, name=”course_list”) maps the root url to the course_list FBV;
2. the routing in the code below will match each course's ID to a detail view: path(‘course/<int:pk>/’, CourseDetailView.as_view(), name=”course_detail”),
3. path (‘course/create/’, create_course, name=”create_course”) - linking to the form for adding a new course.

Proper routing allows for much better naming of links and easy site-wide navigation. The Django view returns HTTP responses in one of several ways:

1. HTML Responses: rendering templates with context (provided by the view)
2. JSON Responses: API endpoint returning data as JSON.
3. Redirect Responses: Examples include, redirecting after a user has successfully submitted a form

A Simple JSON Response for an API View. This view pulls all courses and converts them to a JSON-friendly format.

Views are the backbone in Django's request-response mechanism, enabling the platform to process user input and serve dynamic content. This e-learning platform combines the use of FBVs and CBVs in efficiently handling this set of diversities for the maintenance of a seamless user experience, keeping the code simple and reusable. Also, the use of URL routing and rendering mechanisms seals up the structure and scalability of the platform.

Templates

Templates in Django are an important ingredient that bridges the gap between the backend logic and the user interface. They are used to render HTML pages dynamically by embedding data retrieved from views into predefined HTML structures. The syntax of Django templates is straightforward; moreover, it natively supports variable interpolation, filters, and template tags, making them both powerful and easy to use.

Django's template system has got separation of presentation and logic. Templates take dynamic values and render them at placeholders while loading. The mentioned feature makes the presentation completely independent of back-side logic, which enables one not to worry much while the developers are still changing many things in their code. For instance, simple view of course list could have the following template:

```
<!DOCTYPE html>
<html>
<head>
    <title>Courses</title>
</head>
<body>
    <h1>Available Courses</h1>
    <ul>
        {% for course in courses %}
            <li>
                <a href="{% url 'course_detail' course.id %}">{{ course.title }}</a>
                - {{ course.price }} USD
            </li>
        {% endfor %}
    </ul>
</body>
</html>
```

Figure 43. Course List Template.

This template utilizes Django's template language to loop through a list of courses and render them dynamically. The for loop will iterate over each course, while the url template tag will create URLs for individual course detail pages.

Of all the features, Django templates have something like inheritance. It enables to define a base template containing the shared elements: header and footer, extending the base template in specific pages of it; and minimizing this redundancy and keeping consistency within your application.

The base template could have the following form for this e-learning platform:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{% block title %}E-Learning Platform{% endblock %}</title>
</head>
<body>
<header>
    <nav>
        <ul>
            <li><a href="{% url 'home' %}">Home</a></li>
            <li><a href="{% url 'courses' %}">Courses</a></li>
            <li><a href="{% url 'profile' %}">Profile</a></li>
        </ul>
    </nav>
</header>
<main>
    {% block content %}{% endblock %}
</main>
<footer>
    <p>&copy;E-Learning Platform</p>
</footer>
</body>
</html>
```

Figure 44. Home Page Template.

From this, child templates just extend this base template to define their unique content:

```
{% extends "courses/base.html" %}

{% block title %}Course List{% endblock %}

{% block content %}
    <h1>Course List</h1>
    <ul>
        {% for course in courses %}
            <li>{{ course.title }} - {{ course.price }} USD</li>
        {% endfor %}
    </ul>
{% endblock %}

```

Figure 45. Base Template.

This is the way of ensuring that the base structure remains the same but gives room for individual pages.

Django provides several built-in template filters and tags to work with data in your templates more effectively. For example:

1. Filters: Modify the appearance of data. Here, the title filter capitalizes the first letter of each word in the course title.

```
            <li>
                <p>Course Title: {{ course.title|title }}</p>

                <a href="{% url 'course_detail' course.id %}">{{ course.title }}</a>
                - {{ course.price }} USD
            </li>
```

Figure 46. Template Filters.

2. Tags: Add logic to templates, such as conditional rendering.

```
{% if user.is_authenticated %}
    <p>Welcome, {{ user.username }}!</p>
{% else %}
    <a href="{% url 'login' %}">Login</a>
{% endif %}
```

Figure 47. Template Tags.

Templates often have static content - CSS, JavaScript, images which extend the user interface. Django provides a simple way to include these in your template with the static template tag. For example:

```
{% load static %}
<!DOCTYPE html>
<html>
<link rel="stylesheet" href="{% static "style.css" %}">   You, 5 minutes ago • Uncommitted changes

<head>
```

Figure 48. Template Styles using CSS.

Static content for this project is included under a static directory off the project root. These are collected using Django's collectstatic and deployed.

The way templates get data is by the context provided from the views. Here is a simple example of a view that passes a list of courses to the template:

```
def course_list(request):  2 usages  new *
    courses = Course.objects.all()
    return render(request,  template_name: 'courses/course_list.html',  context: {'courses': courses})
```

Figure 49. FBV with Template.

Above, the following operations are performed:

1. The Course.objects.all() query fetches all courses from the database.
2. This data is injected by the render function into the template at courses/course_list.html.

This template uses placeholders {{ courses }} for dynamic presentation of this data.

Forms are an integral part of any application. The Django template handles forms using the {{ form }} context variable. For example, this is how the course creation form might look:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Create Course</title>
    <form method="post" action="{% url 'create_course' %}">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Create Course</button>
    </form>
</head>
<body>

</body>
</html>
```

Figure 50. Template Form for Course Creation.

The {{ form.as_p }} renders the form fields wrapped in <p> tags. The {% csrf_token %} ensures security against cross-site request forgery attacks.

The Django template system is powerful and flexible for creating dynamic, user-friendly web pages. Inheritance, filters, tags, and integration of static files are some of the features that

make the templates of this e-learning platform consistent in providing an engaging user interface. Separation of logic from presentation enhances maintainability and allows iterative development with minimal disruption to the overall design.

Django Rest Framework (DRF)

Django Rest Framework is an essential toolkit while building Web APIs in Django. This allows one to quickly build APIs by providing a whole lot of tools and abstractions for serialization, URL routing, handling views, authentication, and permissions. In this platform, e-learning exposes data through RESTful endpoints using DRF. It helps the back end to communicate very smoothly with the front-end. These APIs will be responsible, in turn, for user authentication management, information about courses, lesson contents, quiz interactions, and the progress of users using the platform.

Summary in principle, DRF is used to serialize complex data - such as Django model instances - into JSON format for interchange between servers and Web applications. For example, the serializer for the model Course will convert data from its database representation into JSON and present it to the client-side application in this fashion. The following is how the class CourseSerializer may be implemented. It maps the fields of the Course model to their JSON counterparts and ensures consistency and proper structure in the data.

```
class CourseSerializer(serializers.ModelSerializer):  3 usages  Sanzhar *
    instructor = UserSerializer(read_only=True)
    instructor_id = serializers.PrimaryKeyRelatedField(
        write_only=True,
        source="instructor",
        queryset=Course._meta.get_field("instructor").related_model.objects.filter(
            is_instructor=True
        ),
    )
    category = CategorySerializer(read_only=True)
    category_id = serializers.PrimaryKeyRelatedField(
        write_only=True, queryset=Category.objects.all(), source="category"
    )

```

Figure 51. Model CourseSerializer.

With the CourseSerializer defined, the next step is to create API views to handle HTTP requests. Fortunately, DRF simplifies this with ModelViewSet that provides default implementations for common CRUD operations. An example could be a CourseViewSet that manages courses using ModelViewSet whose endpoint, say /api/courses/, would list all courses while /api/courses/<id>/ would retrieve specific course details:

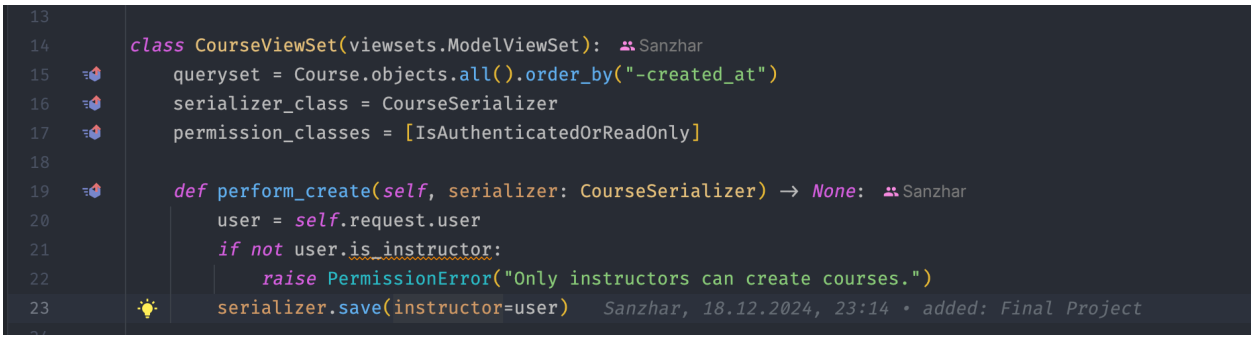


Figure 52. CourseViewSet.

To make these views accessible via URLs, DRF employs routers to map viewsets to RESTful endpoints. By registering the CourseViewSet with a router, the platform automatically generates the required URL patterns, reducing the need for manual configurations. The following example shows how this is achieved.

Figure 53. Router URLs.

Apart from routing, DRF also has strong mechanisms for securing the API with authentication and permissions. For example, while students can only see the courses, instructors can create and manage them. Again, these controls are enforced through DRF's IsAuthenticated and IsAdminUser permissions. The following shows how such permissions are applied to CourseViewSet:

Figure 54. DRF Permission Custiomization.

Another feature of DRF that adds to the user experience is pagination. This limits the number of records returned within a single response. This can be particularly useful for endpoints such as /api/courses/ where there may be many courses on the platform. First, the page size is controlled using a paginator REST_FRAMDEWORK settings:

```
REST_FRAMEWORK = {
    "DEFAULT_AUTHENTICATION_CLASSES": (
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ),
    "DEFAULT_PERMISSION_CLASSES": [
        "rest_framework.permissions.IsAuthenticatedOrReadOnly",
    ],
    "DEFAULT_SCHEMA_CLASS": "drf_spectacular.openapi.AutoSchema",
    "DEFAULT_PAGINATION_CLASS": "rest_framework.pagination.PageNumberPagination",
    "PAGE_SIZE": 10,
}
```

Figure 55. DRF Pagination.

One of the critical endpoints developed for the platform is the API for progress tracking, which will allow students to view their progress within a course. The API draws data from the UserProgress model, which tracks completed lessons and quiz scores. The following code defines the UserProgressSerializer and the corresponding view to expose progress data.

```
class UserProgressSerializer(serializers.ModelSerializer):  2 usages  Sanzhar
    course_title = serializers.CharField(source='course.title', read_only=True)
    lessons_completed = serializers.SerializerMethodField()
    total_lessons = serializers.SerializerMethodField()
    quizzes_completed = serializers.SerializerMethodField()
    total_quizzes = serializers.SerializerMethodField()

    class Meta:  Sanzhar
        model = UserProgress
        fields = [
            'id',
            'course_title',
            'lessons_completed',
            'total_lessons',
            'quizzes_completed',
            'total_quizzes',
            'completed_lessons',
        ]

    def get_lessons_completed(self, obj):  Sanzhar
        return len(obj.completed_lessons) if obj.completed_lessons else 0
```

Figure 56. Model UserProgressSerializer.

The UserProgressView ensures that only authenticated users can access progress data. It filters the data based on the logged-in user and the specified course, providing a tailored response.

```python
class UserProgressViewSet(viewsets.ModelViewSet):  Sanzhar          Sanzhar, 18.12.
    queryset = UserProgress.objects.all()
    serializer_class = UserProgressSerializer
    permission_classes = [permissions.IsAuthenticated]

    def get_queryset(self):  Sanzhar
        user = self.request.user
        course_id = self.request.query_params.get('course_id')
        queryset = UserProgress.objects.filter(user=user)
        if course_id:
            queryset = queryset.filter(course_id=course_id)
        return queryset

    def perform_create(self, serializer):  Sanzhar
        if "user" not in serializer.validated_data:
            serializer.save(user=self.request.user)
        else:
            serializer.save()
```

Figure 57. UserProgressViewSet.

Another significant feature of DRF is the browsable API-a web interface to explore and test APIs. This feature is very useful during development and debugging because it gives real-time feedback about API functionality.

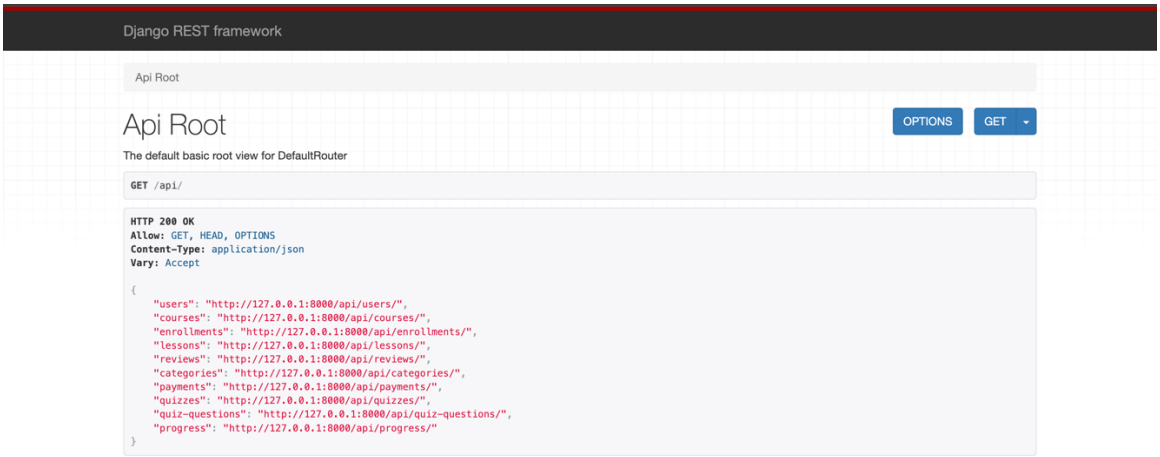


Figure 58. DRF Browsable API.

In the end, DRF acts as the backbone of the API architecture for the e-learning platform. It fills the gap between the backend and frontend by ensuring smooth data exchange and user interactions. By leveraging features in DRF, the platform achieves scalability, security, and maintainability, laying a strong foundation for future enhancements.

Frontend Integration

On the client side, this e-learning system is built on top of Vue.js; it is a modern JavaScript Framework that is usually used to address the development of interactive user interfaces for the web. Because the Django back-end is integrated into the Vue.js front-end with RESTful APIs exposed by Django Rest Framework-Django Rest Framework. This section describes the integration of the frontend, basically how the Vue.js application will be consuming the APIs and the communication with the backend.

The frontend is meant to be client-side, which will render dynamic content, handle user interactions, and send requests back to the backend for data retrieval or modifications. Integration with the frontend starts with setup in Vue.js: setting up the project structure. Here, the code structure is modular; by components, it means isolated parts of the user's interface, such as course listing, user profile, and progress tracking. To each component, there is an exact correspondence to some exact API endpoint exposed by the backend.

For example, the component CourseList is supposed to list the courses available. To be able to do this it would make a GET on the /api/courses/ endpoint. The active component CourseList would be set up as follows:

```vue
<template> Show component usages
  <Navbar />
  <div class="p-8">
    <h1 class="text-3xl font-bold mb-4">All Courses</h1>
    <div class="grid grid-cols-1 md:grid-cols-3 gap-4">
      <CourseCard v-for="course in courses" :key="course.id" :course="course" />
    </div>
  </div>
</template>
  Sanzhar, 18.12.2024, 23:14 • added: Final Project
<script lang="ts">
import ...

export default defineComponent( options: { Show usages  Sanzhar
  name: 'CourseList',
  components: { Navbar, CourseCard },
  setup() {
    const courses = ref<Course[]>( value: []);

    onMounted( hook: () => {
      axios.get( url: 'http://localhost:8000/api/courses/')
        .then((res) => (courses.value = res.data.results))
        .catch(console.error);
    });

    return { courses };
  },
});
</script>
```

Figure 59. CourseList Vue Component.

In this, the mounted life cycle hook fetches course data when the component becomes active. Axios handles HTTP requests and provides easy communication between the client and server sides of the code.

The second most critical approach involves how authentication would be handled. On the frontend side, it uses token-based authentication; the API will only give access to any of its routes if they have a valid token for access. The request also does a POST to /api/auth/token/ for the purposes of logging in. On successful authentication, the access token is returned by the backend and is locally stored in the browser's local storage by the frontend. Deeper in the application, this is used in the Authorization header for subsequent requests.

Following is a sample implementation of the login functionality inside of the Login component:

```
<script lang="ts">
export default defineComponent( options: {  Show usages  Sanzhar
  name: "Login",
  components: { Navbar },
  setup() {
    const username = ref( value: "");
    const password = ref( value: "");

    function loginUser() {  Show usages  Sanzhar
      axios
        .post( url: "http://localhost:8000/api/auth/token/",  data: {
          username: username.value,
          password: password.value,
        })
        .then((res) => {
          const token = res.data.access;
          setAuthToken(token);

          axios
            .get( url: "http://localhost:8000/api/users/me/",  config: {
              headers: { Authorization: `Bearer ${token}` },
            })
            .then((uRes) => {
              const user = uRes.data;
              localStorage.setItem("is_instructor", user.is_instructor.toString());
              localStorage.setItem("username", user.username);
              localStorage.setItem("user_id", user.id.toString());
              window.location.href = "/";
            })
            .catch(() => alert("Failed to fetch user data"));
        })
        .catch(() => alert("Invalid credentials"));
    }

    return { username, password, loginUser };
  },
```

Figure 60. Login Vue Component.

It then sets up an Axios interceptor on the frontend to automatically include this token in all outgoing requests for consistent authentication throughout the application. This way, it is not necessary to attach the token manually to every request.

```
171     methods: {
172       attachToken() {
173         axios.interceptors.request.use( onFulfilled: (config) => {
174           const token = localStorage.getItem( key: "token");
175           if (token) config.headers["Authorization"] = `Bearer ${token}`;
176           return config;
177         });
178       },   Sanzhar, 18.12.2024, 23:14 • added: Final Project
```

Figure 61. Attach Token in Config.

Another critical feature on the platform is progress tracking. The ProgressTracker component pulls data about progress from /api/progress/ and displays it to the user. The following code demonstrates how progress data is retrieved and displayed:

```
61  <script>
62  import Navbar from "../components/Navbar.vue";
63  import axios from "axios";
64
65  export default {  Show usages   Sanzhar *
66    components: {Navbar},
67    data() {
68      return {
69        progress: [],
70      };
71    },
72    mounted() {
73      this.fetchProgress();
74    },
75    methods: {
76      fetchProgress() {
77        axios
78          .get( url: "http://localhost:8000/api/progress/")
79          .then((res) => {
80            if (res.data.results) {
81              this.progress = res.data.results;
82            } else {
83              this.progress = [];
84            }
85          })
86          .catch((err) => {
87            console.error("Failed to load progress:", err);
88            this.progress = [];
89          });
90      },
91      getProgressPercentage(completed, total) {
92        if (!total || total === 0) return 0;
93        return Math.round( x: (completed / total) * 100);
94      },
95    },
```

Figure 62. Progress Bar Vue Component.

This component showcases real-time capabilities of the platform where data flows seamlessly from the front-end to the back end. Progress data is dynamic, depending on what actions are taken by the user in terms of lesson and quiz completion.

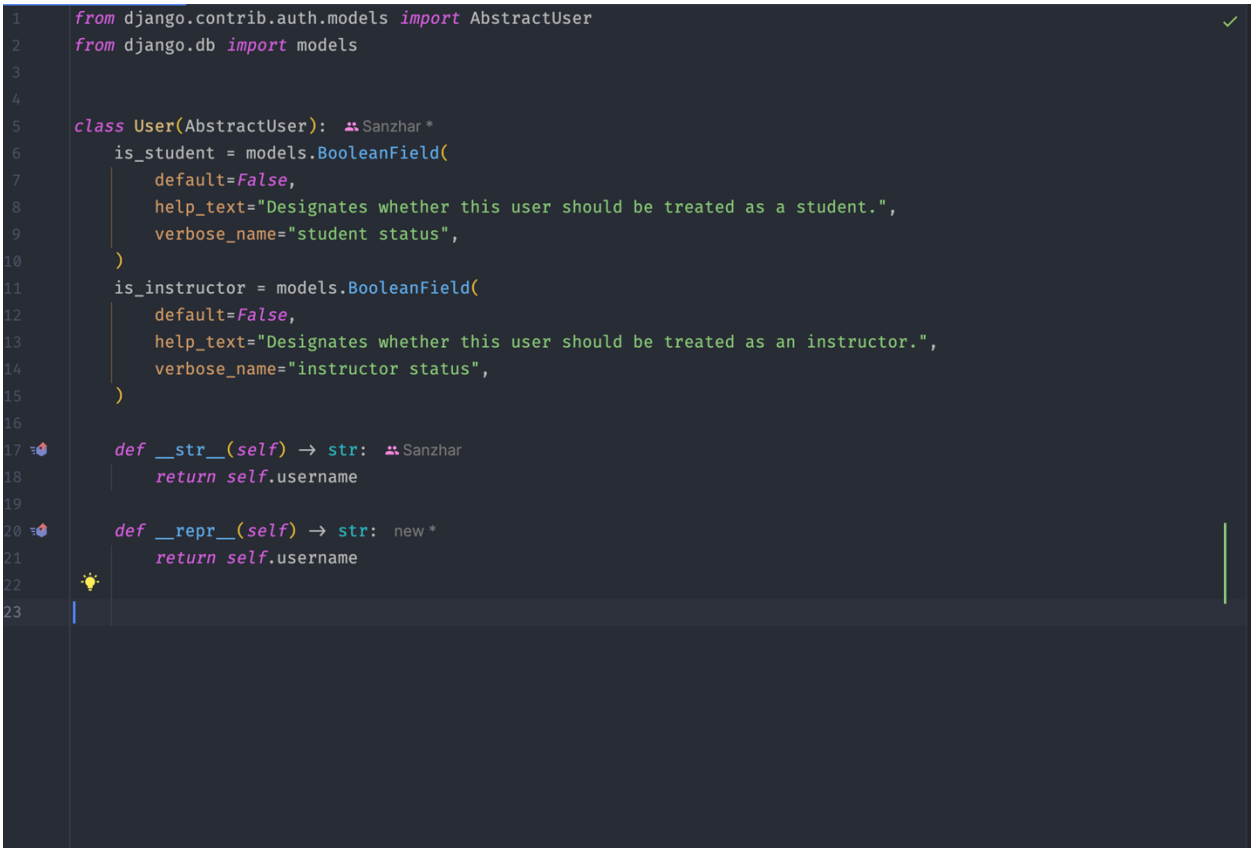
Frontend integration plays a major role in connecting the user interface of the platform with its functional backend. The platform is responsive and user-friendly, powered by Vue.js and

RESTful APIs. This integration makes sure that all components work harmoniously to provide users with a seamless e-learning journey.

Challenges and Solutions

This e-learning platform had many challenges to be overcome during the development process, be it at the backend, frontend, or database layers. These have been addressed through effective strategy and the features of Django, Vue.js, and Docker.

The most challenging part, of course, was establishing a secure and scalable authentication system that would provide user roles, including but not limited to students and instructors. Instructors should be able to manage courses and quizzes while allowing students to access enrolled courses.



```python
from django.contrib.auth.models import AbstractUser
from django.db import models


class User(AbstractUser):  Sanzhar *
    is_student = models.BooleanField(
        default=False,
        help_text="Designates whether this user should be treated as a student.",
        verbose_name="student status",
    )
    is_instructor = models.BooleanField(
        default=False,
        help_text="Designates whether this user should be treated as an instructor.",
        verbose_name="instructor status",
    )

    def __str__(self) -> str:  Sanzhar
        return self.username

    def __repr__(self) -> str:  new *
        return self.username
```

Figure 63. User Inheritance.

Solution was token-based authentication carried out using Django Rest Framework. In addition, there was an extension of the default DRF's User model to include extra fields that would differentiate between instructor and student.

The heavy relationships between courses, lessons, quizzes, and users' progress needed careful designing of the database. Completed lessons, quiz scores, and reviews needed to be tracked down on the platform while maintaining data integrity and scalability.

```python
class UserProgress(models.Model):  7 usages  Sanzhar        Sanzhar, 18.12.2024, 23:14 • added: Final Project
    user = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
        related_name="progress",
        verbose_name="User",
        help_text="Select the user.",
    )
    course = models.ForeignKey(
        Course,
        on_delete=models.CASCADE,
        related_name="progress",
        verbose_name="Course",
        help_text="Select the course.",
    )
    completed_lessons = models.JSONField(
        default=list,
        verbose_name="Completed Lessons",
        blank=True,
        null=True,
    )
    quiz_scores = models.JSONField(
        default=dict,
        verbose_name="Quiz Scores",
        help_text="Enter the quiz scores.",
        blank=True,
        null=True,
    )
```

Figure 64. UserProgress with JSONField.

To handle this problem, the ORM in Django was used to clearly relate the models. For instance, the UserProgress model used JSON fields to store completed lessons and quiz scores dynamically.

Fetching and showing lessons and progress dynamically were somewhat tricky because API calls are asynchronous. The user needed real-time updates for a smooth learning experience.

```
  mounted() {
    this.fetchProgress();
  },
  methods: {
    fetchProgress() {
      axios
        .get( url: "http://localhost:8000/api/progress/")
        .then((res) => {
          if (res.data.results) {
            this.progress = res.data.results;
          } else {
            this.progress = [];
          }   Sanzhar, 18.12.2024, 23:14 • added: Final Project
        })
        .catch((err) => {
          console.error("Failed to load progress:", err);
          this.progress = [];
        });
    },
    getProgressPercentage(completed, total) {
      if (!total || total === 0) return 0;
      return Math.round( x: (completed / total) * 100);
    },
  },
};
</script>
```

Figure 65. Fetching User Progress.

The solution was to implement the reactivity system by Vue.js. For example, according to user progress on lessons and quizzes, the progress would be updated on the same page.

Conclusion

The e-learning platform is the embodiment of all modern web development technologies in one entity to develop an interactive, secure, scalable system. Using Django as the backend, Vue.js for the user interface, and PostgreSQL for robust data management, the system effectively closes the gap between instructors and students. The containerization of Docker further assures deployment across diverse environments without any hindrance.

Key takeaways from the project are a fully implemented modular architecture with clarity in the role of its components; the Django Rest Framework allowed to quickly build a strong API level that would allow for comfortable communication between the backend and the frontend. Furthermore, this allowed the usage of Vue.js to create a truly dynamic and user-friendly interface thanks to which responses to all actions taken by users happened instantaneously.

These were some of the challenges involved in the project: implementing flexible data models that can accommodate various user interactions, handling role-based authentication, and efficiently handling dynamic content. These are overcome with good development practices that include performance optimization with caching, security with role-based permissions, and responsive design principles for an engaging user experience.

While the platform does a decent job for its intended goals, it allows for added avenues where enhancement can be made in tracking user progress with more in-depth analytics, AI recommendations for course suggestions based on personal interest, and live interaction capabilities via video conferencing that will go a step further in enhancing learning.

In the end, this e-learning platform is just a simple example of what can be achieved using Python, Django, Vue.js, and Docker in web development, but it also covers the scalability of the system, being user-oriented, and flexible for future improvements. The project ushers in the transformative role of technology in making education accessible and engaging for people all over the world.

References

1. Django Documentation - https://docs.djangoproject.com
2. Docker Official Documentation - https://docs.docker.com
3. Django REST Framework Documentation - https://www.django-rest-framework.org
4. drf-spectacular Documentation - https://drf-spectacular.readthedocs.io
5. Vue-js Documentation - https://vuejs.org/
6. Tailwind CSS Documentation - https://tailwindcss.com/
7. Free API for Mock Data - https://mockaroo.com/
8. MDN Web Docs - https://developer.mozilla.org/
9. Vite Documentation - https://vitejs.dev/
10. Docker Compose Documentation - https://docs.docker.com/compose/

Appendices

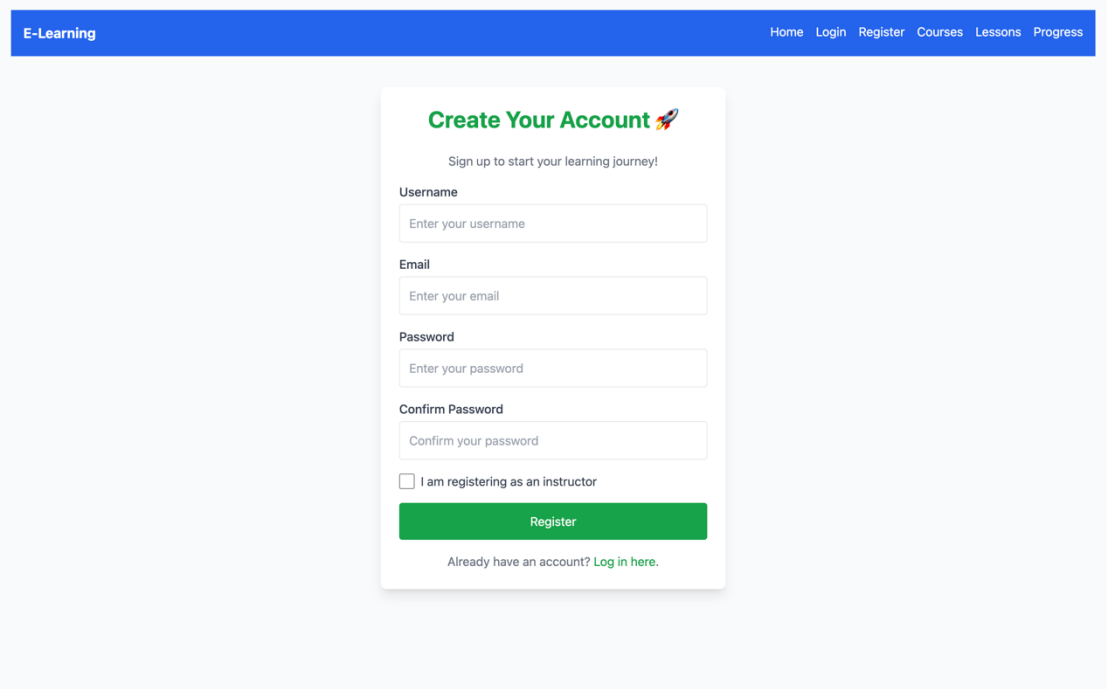


Figure 66. Register Page.

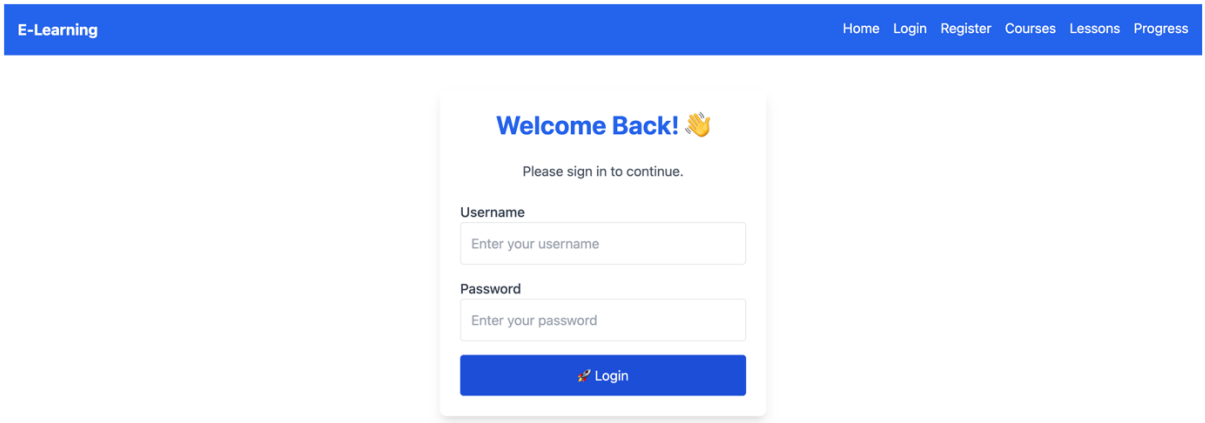


Figure 67. Login Page.

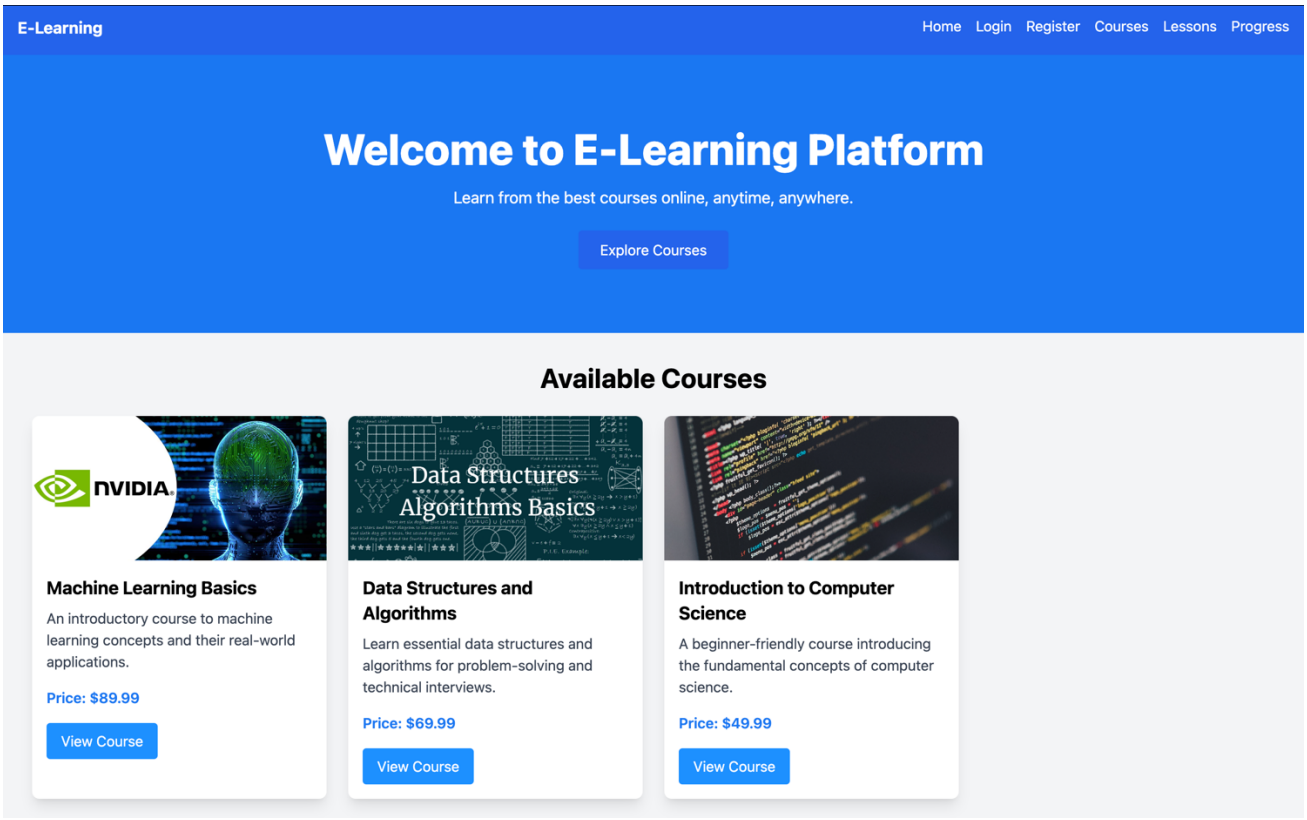


Figure 68. Home Page.

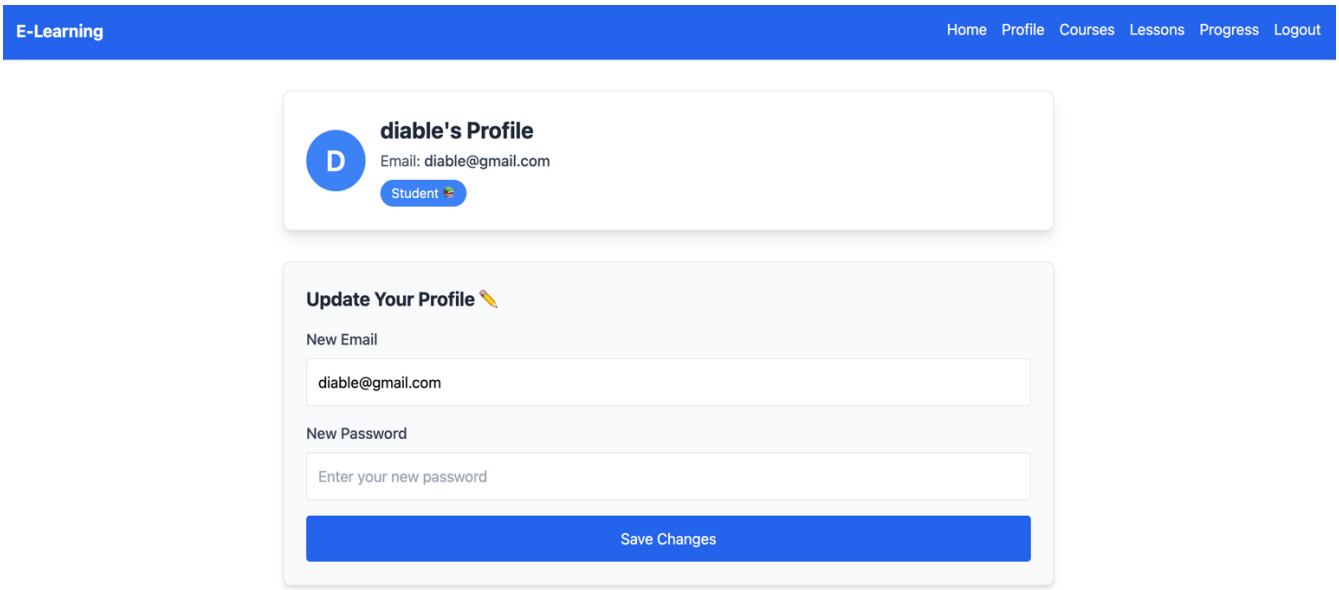


Figure 69. Student Profile Page

.

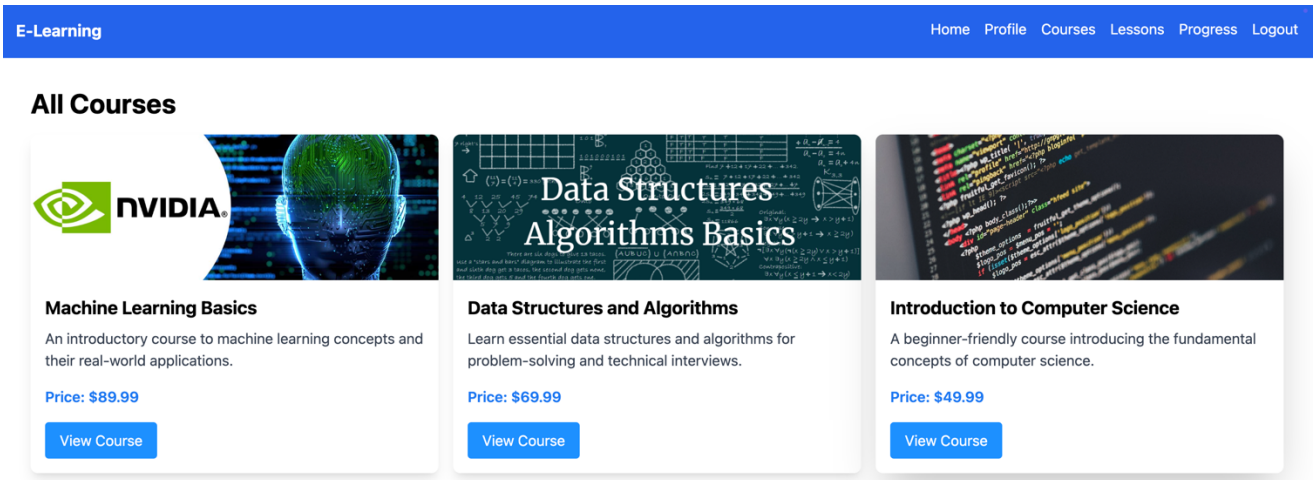


Figure 70. Course List Page.

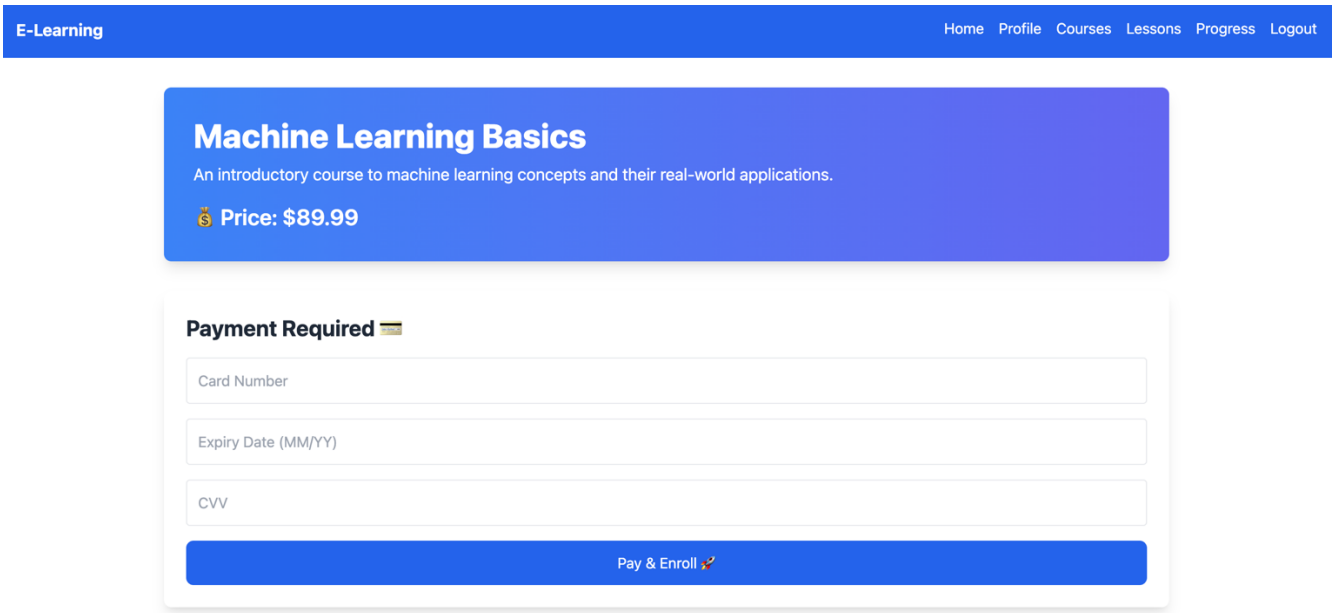


Figure 71. Course Payment Form.

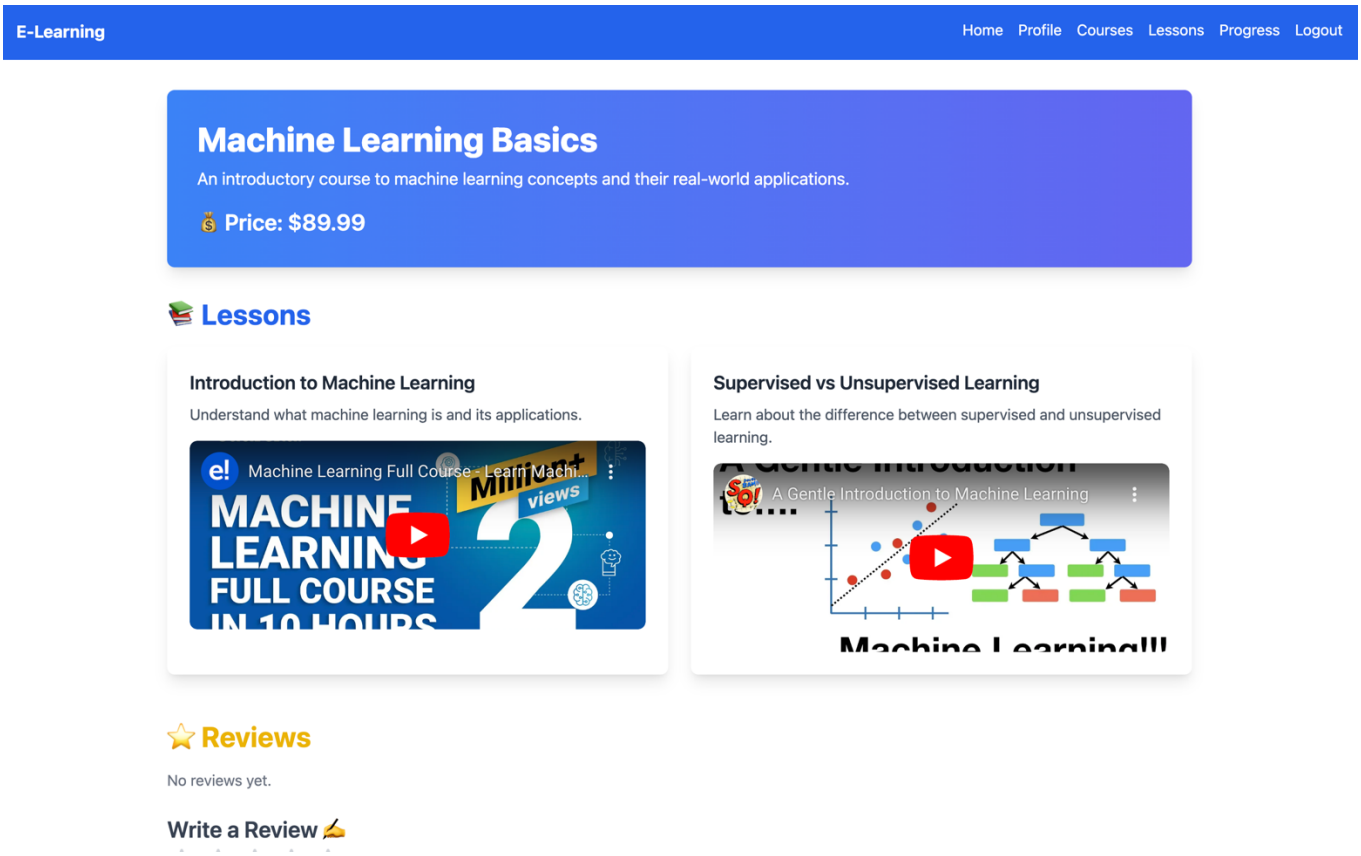


Figure 72. Course Detail Page after Payment.

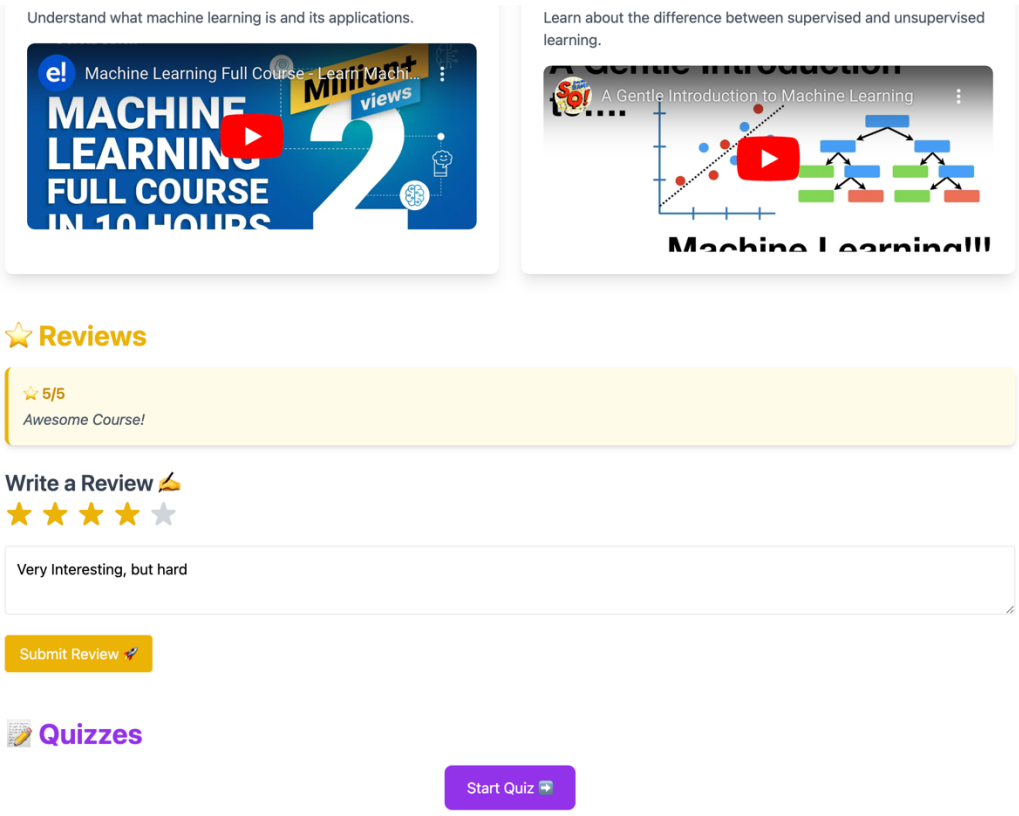


Figure 73. Course Review List and Review Adding. Also Quizzes Button

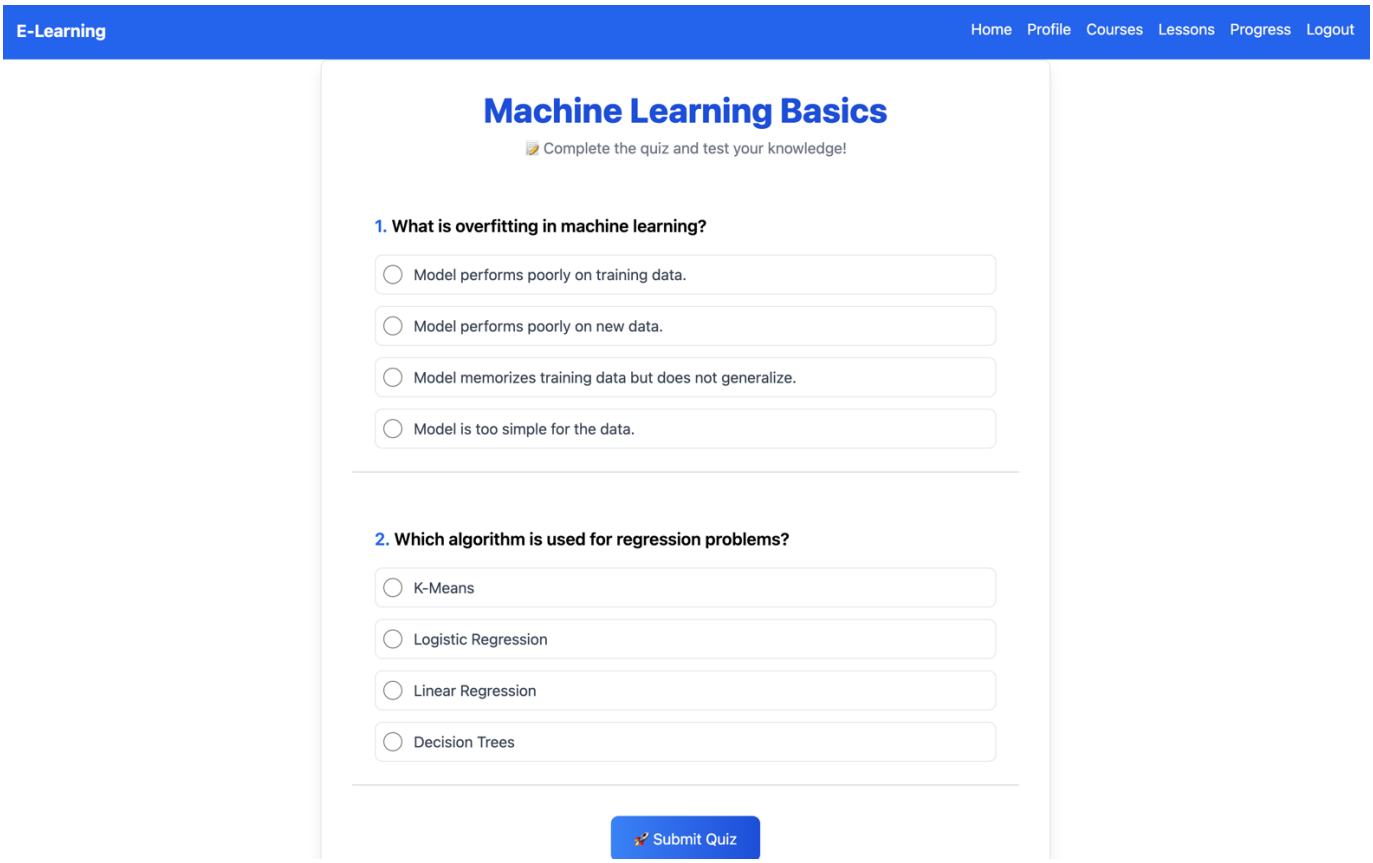


Figure 74. Quiz Page.

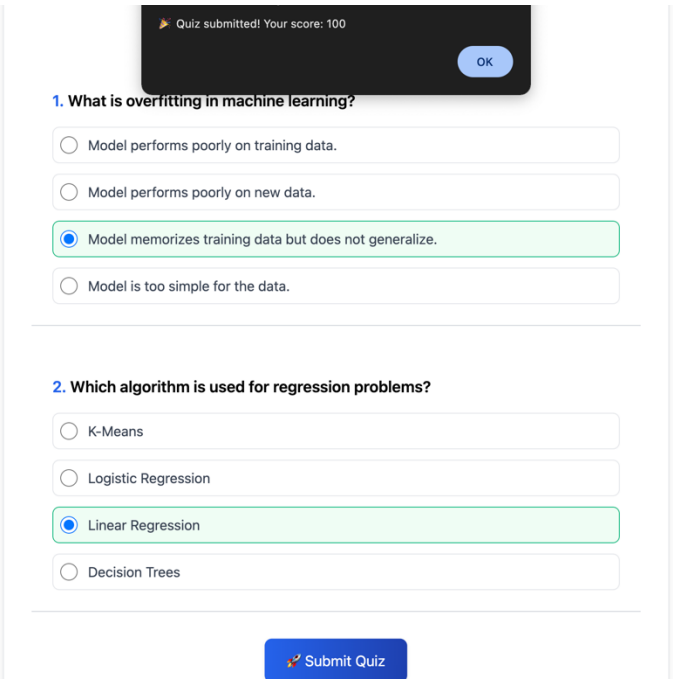


Figure 75. Quiz Submission with Results.

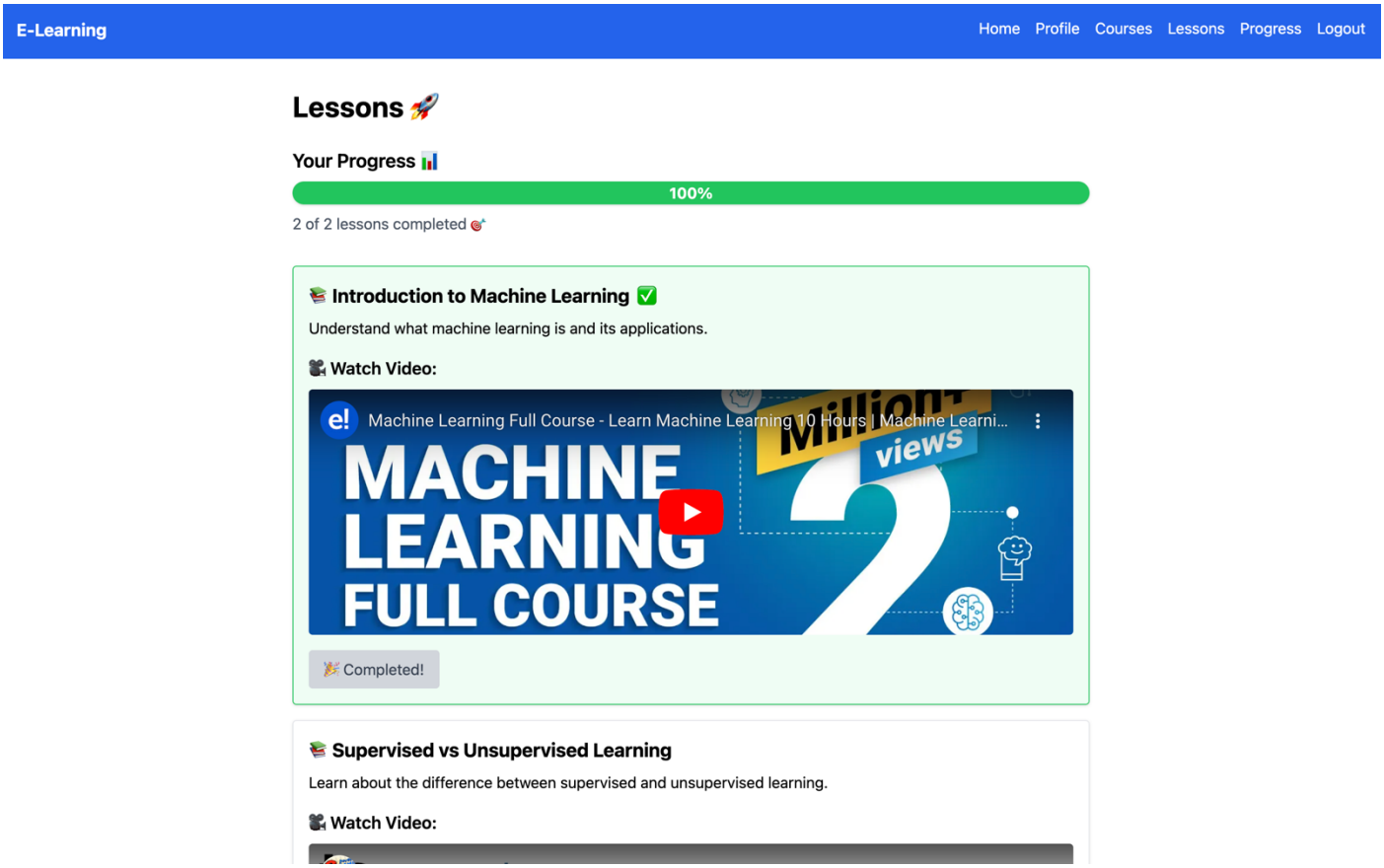


Figure 76. Lesson Page.

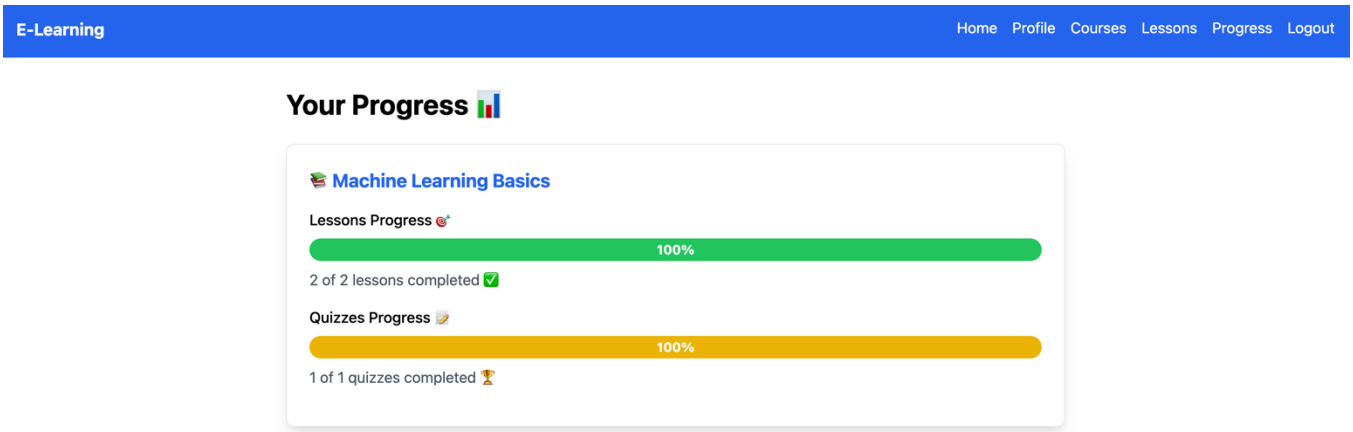


Figure 77. User Progress Page.

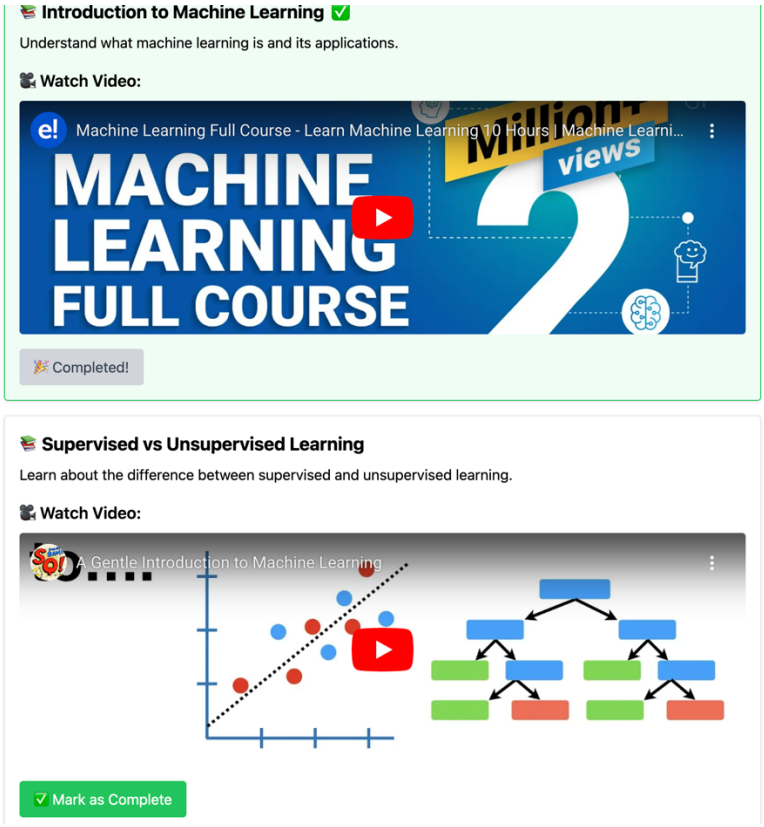


Figure 78. Lessons List & Marking Lesson as Complete.

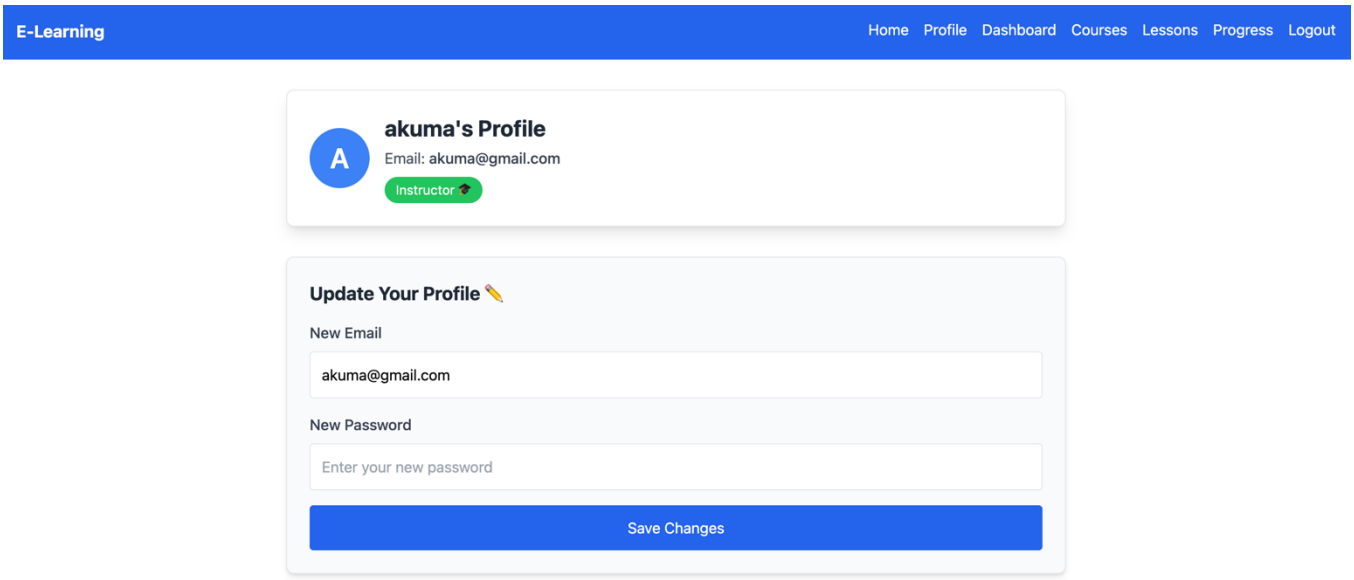


Figure 79. Instructor Profile.

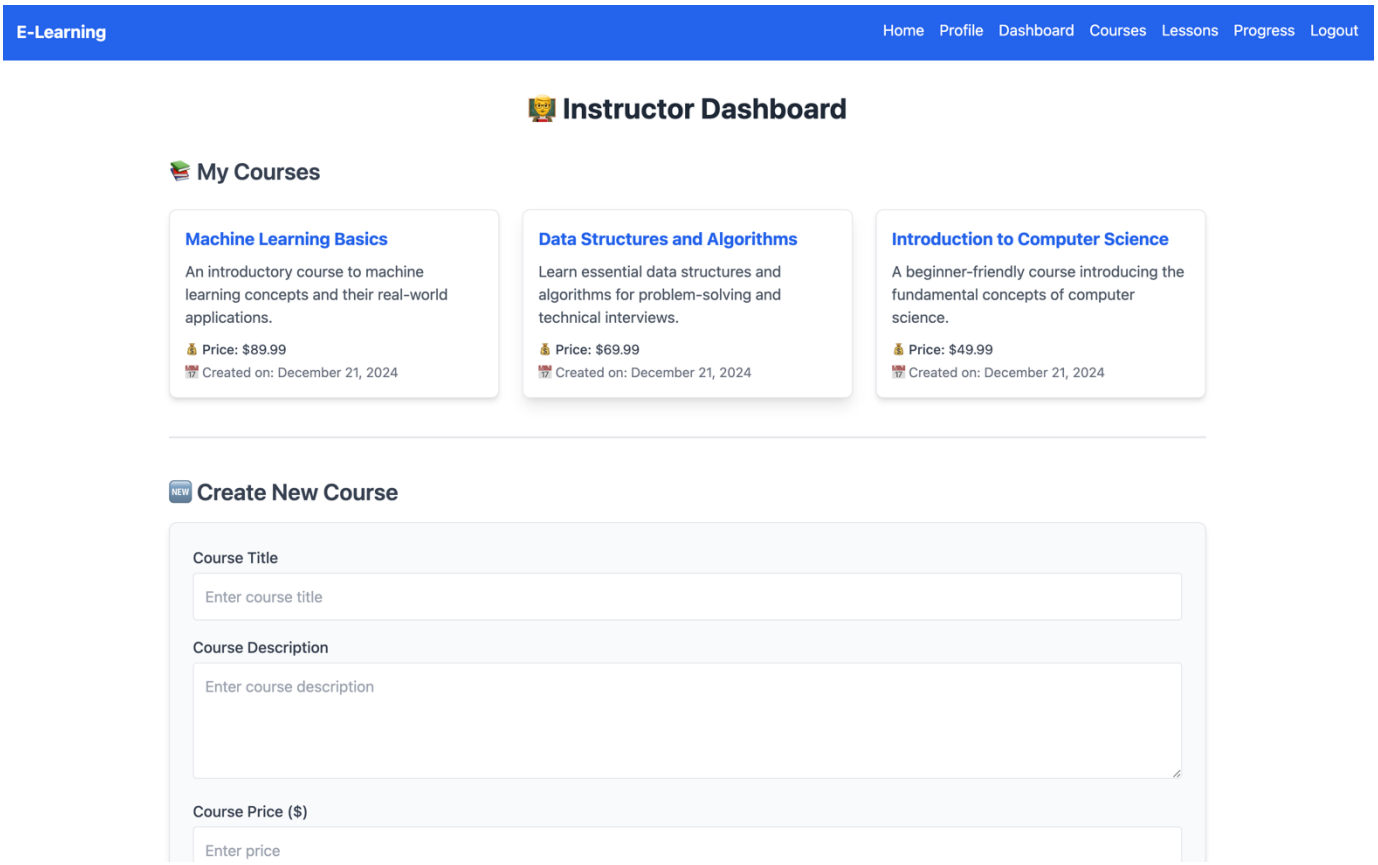


Figure 80. Instructor Dashboard Page.

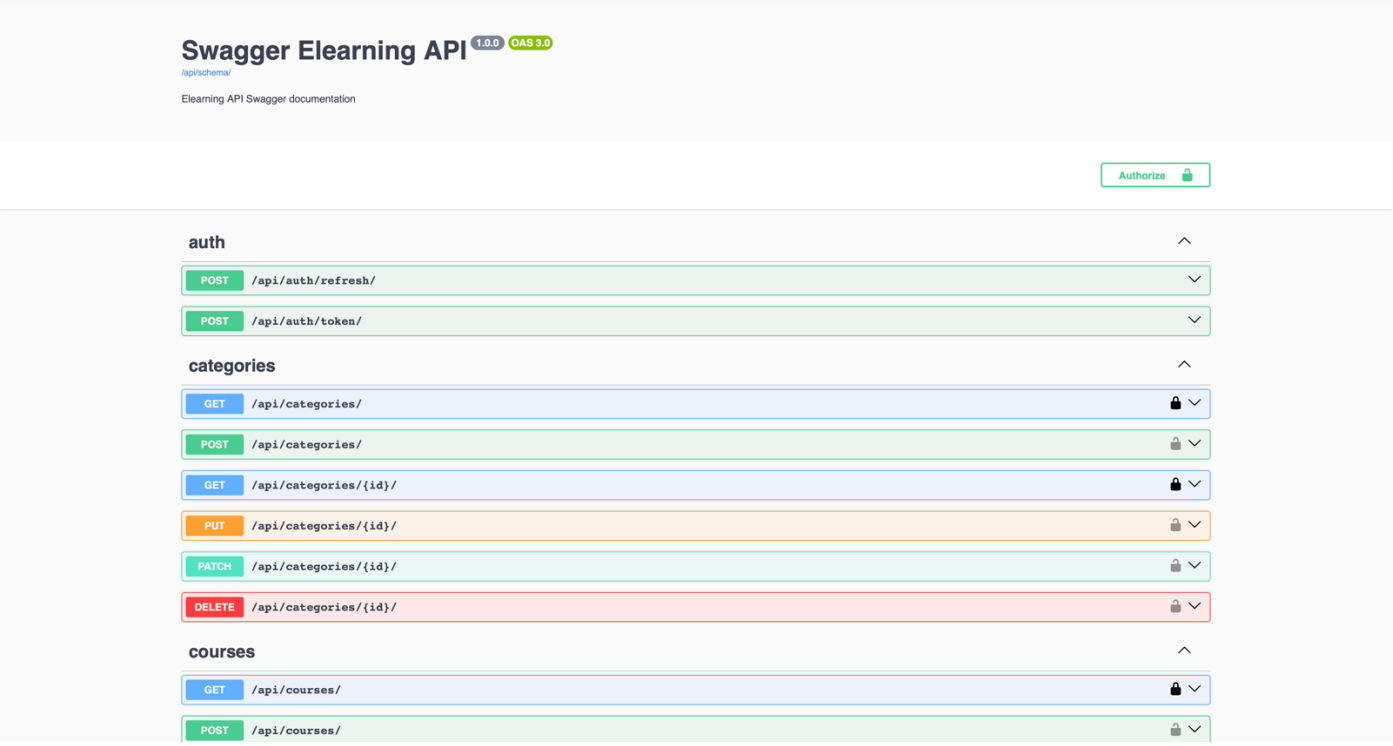


Figure 81. Swagger API Documentation.

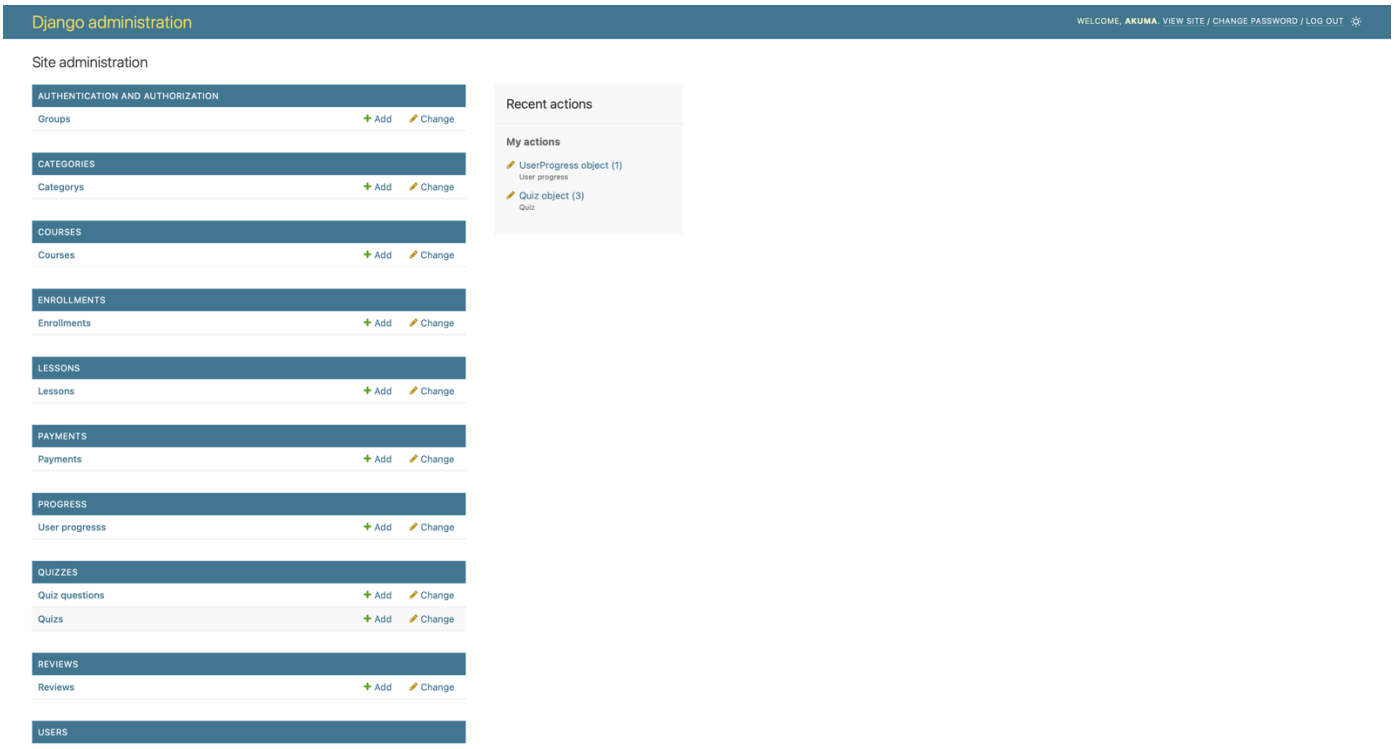


Figure 82. Django Admin Page.

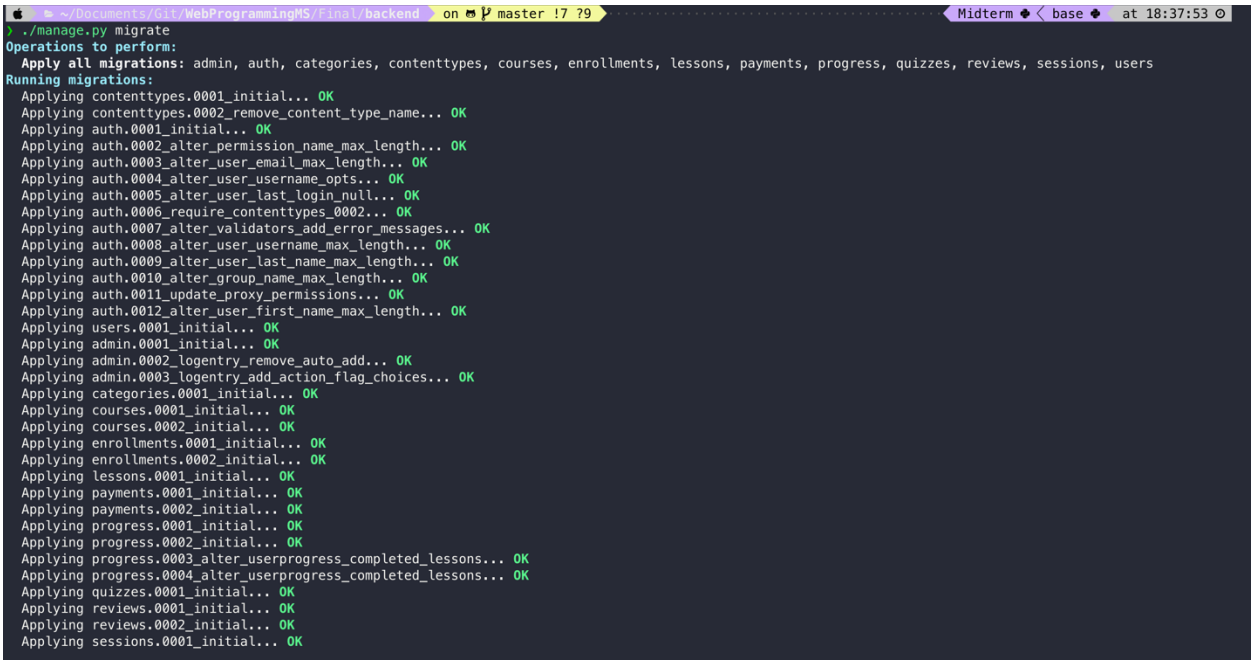


Figure 83. Django Applied Migrations.

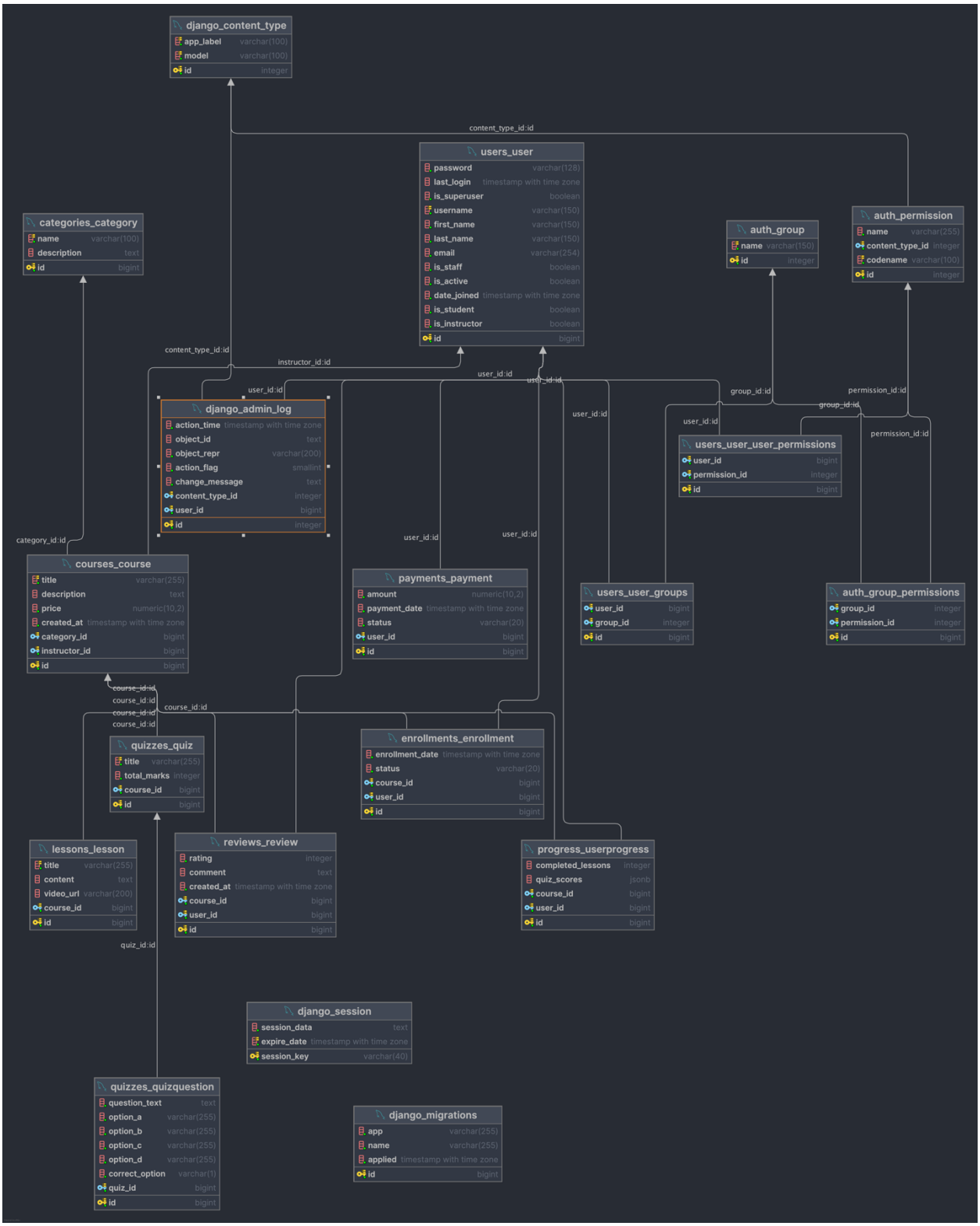


Figure 84. Database Diagram.

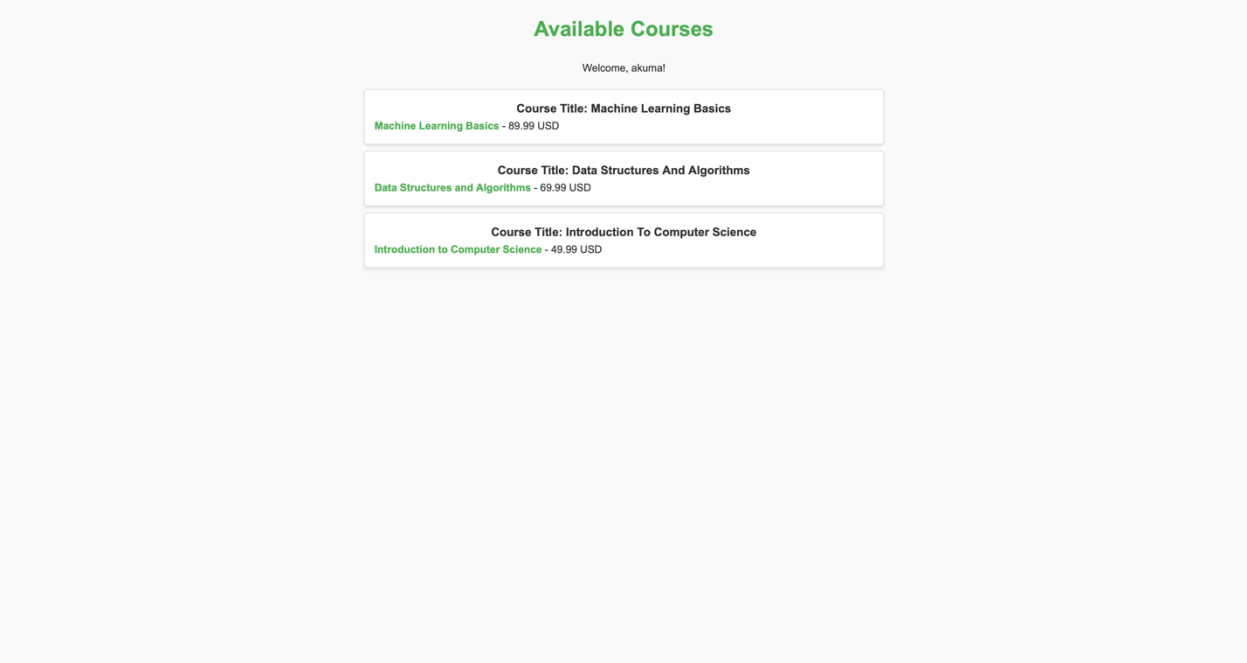


Figure 85. Course List Page with Django Template without Vue.

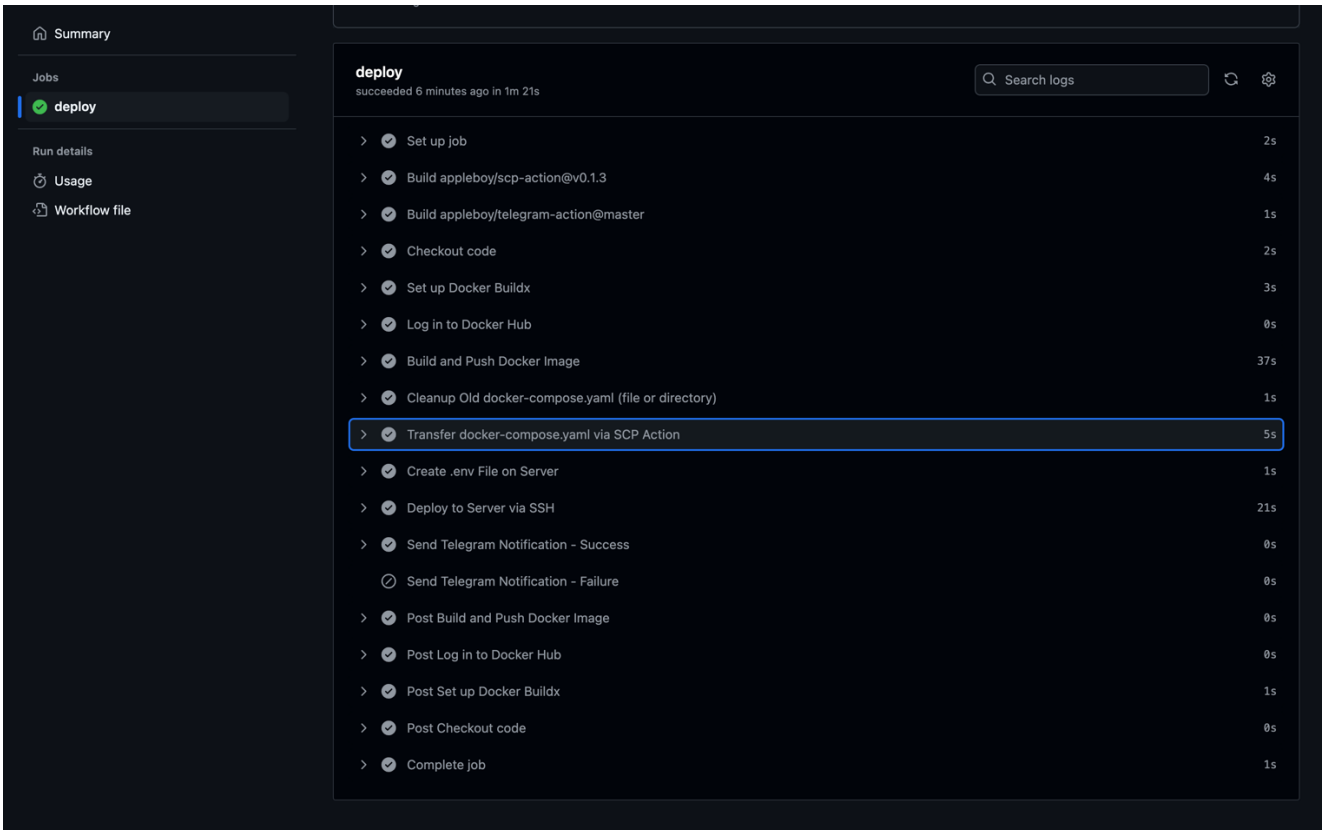


Figure 86. CI/CD Pipeline.

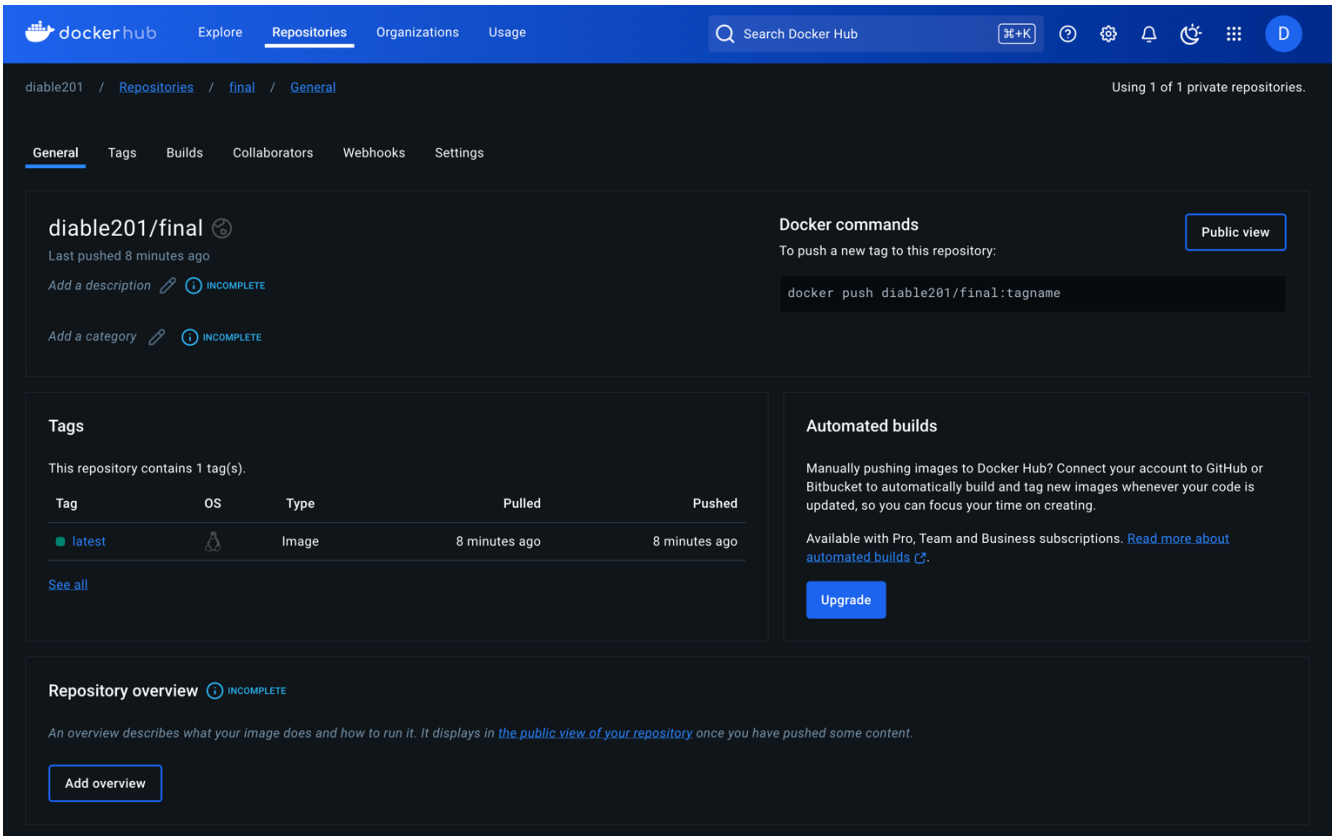


Figure 87. Docker Hub Image.