Code Documentation

Table of Contents

1. **Introduction**
    - Project Overview
    - System Architecture
2. **Backend Development**
    - Setup and Configuration
    - Models
    - Views
    - Serializers
    - URL Routing
    - Authentication and Permissions
    - Testing
3. **Frontend Development**
    - Component Architecture
    - State Management
    - Routing
    - Integration with Backend
4. **API Documentation**
    - Endpoint Descriptions
    - Request and Response Examples
5. **Deployment**
    - Server Configuration
    - Continuous Integration and Deployment

1. Introduction

Project Overview

The Course Recommendation System is designed to recommend educational courses to users based on their preferences and interaction history. The backend is built with Django and handles data management, API exposure, and recommendation logic, while the frontend is developed in React to provide a dynamic and responsive user interface.

2. Backend Development

Setup and Configuration

- **Dependencies**: Django, Django REST Framework, SQLite, OpenAI
- **Installation Steps**:

  Navigate to the backend directory:

  cd backend

  Install required Python dependencies:

  pip install -r requirements.txt

  Run migrations to set up the database:

  python manage.py migrate

  Start the Django server:

  python manage.py runserver

Models

CustomUser

- **Purpose:** The CustomUser model extends Django's built-in user model to provide a more flexible authentication system tailored to the specific needs of the Course Recommendation System.

- **Fields:**

    - email: Serves as the unique identifier for the user instead of the traditional username field. It's an EmailField with a uniqueness constraint.
    - first_name: Optional CharField to store the user's first name, can be left blank.
    - last_name: Optional CharField to store the user's last name, can be left blank.
    - is_active: A BooleanField indicating whether the user's account is considered active.
    - is_staff: A BooleanField that determines whether the user can access the Django admin. By default, this is set to False.
- **Role in System**: Used for all aspects of user management, including but not limited to authentication, session management, and permission handling.
- **Manager Class**: CustomUserManager, a custom model manager for creating users and superusers with email as the unique identifier.

CustomUserManager

- **Purpose:** Handles creation and management of user instances. This custom manager is particularly important for ensuring that email addresses are normalized and that the user creation process adheres to business rules.
- **Methods:**
    - create_user(email, password=None, **extra_fields): Standard method for creating a user. It requires an email and password and accepts additional fields. Validates the presence of an email.
    - create_superuser(email, password=None, **extra_fields): Used to create a superuser with administrative privileges. It sets is_staff and is_superuser flags to True.

Course

- **Description:** Contains details about the courses available for recommendation.
- **Attributes:**
    - id: Unique identifier for the course.
    - title: Title of the course.
    - description: A brief description of what the course covers.
- **Relationships:**
    - **Interactions**: Each course can be associated with many user interactions.
- **Role in System:** Courses are the central aspect of the system, serving as the primary entities that users interact with and receive recommendations for.

Recommendations

UserCourseInteraction

- **Purpose:** Records various types of interactions between users and courses to inform and optimize the recommendation algorithms.
- **Fields:**
  - user: A ForeignKey linking to the AUTH_USER_MODEL which identifies the user involved in the interaction.
  - course: A ForeignKey linking to the Course model, specifying which course the interaction is related to.
  - interaction_type: A CharField describing the nature of the interaction, such as 'viewed', 'liked', or 'enrolled'.
  - timestamp: A DateTimeField that records the exact time the interaction occurred, automatically set to the time of creation.
- **Role in System:** Critical for the recommendation engine, as these interactions serve as the primary data source for understanding user preferences and behavior.

OpenAIPrompt

- **Purpose:** Manages prompts used for generating dynamic content or queries within the system, potentially integrating with external AI services like OpenAI for content generation or interactive chat functionalities.
- **Fields:**
  - prompt: A TextField holding the prompt text used for interactions or content generation.
  - description: An optional TextField providing additional context or details about the prompt.
  - created_at: A DateTimeField capturing the time when the prompt was created, to track versions and updates.
- **Role in System:** Utilized primarily by backend services to generate dynamic responses or content based on user queries or system needs, enhancing user interaction and engagement.

UserRecommendation

- **Purpose:** Stores personalized recommendations generated for each user, ensuring that users can view a history of their recommendations.
- **Fields:**
  - user: A ForeignKey linking to the CustomUser model, identifying the user for whom the recommendation is generated.
  - recommendation: A TextField containing the textual representation of the recommendation.

        - created_at: A DateTimeField that logs when the recommendation was created, helping to maintain a timeline of user recommendations.
    - **Meta Options:**
        - ordering: Configured to display the most recent recommendations first (['-created_at']).
    - **Role in System:** Provides a user-specific record of recommendations, allowing for historical analysis and potentially enabling users to revisit past recommendations.

Views

CourseListView

- **Purpose**: Manages the display and administration of course listings. It allows for the creation of new courses and lists all available courses in the system.
- **Functionality**:
    - **GET Request**: Retrieves and displays a list of all courses stored in the database. This list can be filtered and paginated based on parameters provided by the client.
    - **POST Request**: Allows authenticated administrators to create a new course by submitting course details. Validation is performed to ensure data integrity.
- **Attributes**:
    - queryset: The Course objects to be displayed, filtered, and paginated.
    - serializer_class: Utilizes CourseSerializer to serialize course data for API responses.
    - permission_classes: Defines access permissions, restricting POST requests to users with administrative privileges.
- **Methods**:
    - get_queryset(): Can be overridden to apply additional filters dynamically based on the request parameters or user attributes.
- **URL Path**: Typically mapped to /api/courses/ to handle API requests concerning course listings and creation.

RecommendationView

- **Purpose**: Generates and serves personalized course recommendations to users based on their historical interactions and preferences.
- **Functionality**:

    - **GET Request**: When a request is received, the view interacts with the recommendation engine to fetch personalized course suggestions for the logged-in user.
- **Attributes**:
    - serializer_class: Utilizes a custom serializer, possibly RecommendationSerializer, designed to handle the specifics of recommendation data.
    - permission_classes: Ensures that only authenticated users can access their personalized recommendations.
- **Methods**:
    - get_queryset(): Instead of a static dataset, this method interacts with the recommendation engine to generate dynamic results based on the user's profile and past interactions.
- **URL Path**: Typically mapped to /api/recommendations/ to cater to requests related to fetching user-specific course recommendations.
- **Additional Components**:
    - **Recommendation Engine Integration**: This view acts as a conduit between the frontend and the complex algorithms running in the backend that analyze user data to produce recommendations.

Serializers

CourseSerializer

- **Purpose**: Handles serialization and deserialization of Course data.
- **Functionality**:
    - **Serialization**: Converts Course model instances into JSON format that can be sent to the client.
    - **Deserialization**: Validates and converts incoming JSON data from the client into Course model instances.
- **Fields**:
    - id: AutoField, read-only, serves as the unique identifier for courses.
    - title: CharField, required for course creation and updates.
    - description: TextField, provides a detailed description of the course.
- **Methods**:
    - create(validated_data): Overrides the default create method to add custom creation logic.
- **Validation**: Ensures that all required fields are present and correctly formatted before saving to the database.
- **Usage**: Utilized in CourseViewSet for handling POST requests and GET responses involving course data.

UserSerializer

- **Purpose**: Manages serialization and deserialization of User model data, particularly for user authentication and profile management.
- **Functionality**:
  - **Serialization**: Translates CustomUser model instances to JSON.
  - **Deserialization**: Processes and validates JSON data to create or update CustomUser model instances.
- **Fields**:
  - id: AutoField, read-only, unique identifier for users.
  - username: CharField, necessary for user identification.
  - email: EmailField, used for communication and recovery actions.
  - password: CharField, stored securely, never serialized back to the client.
- **Methods**:
  - validate_password(value): Custom validation logic to ensure strong passwords.
- **Usage**: Employed by the user management system for registration and updating user profiles.

OpenAIPromptSerializer

- **Purpose**: Serializes interactions between users and courses to track and analyze user behavior for generating recommendation.
- **Functionality**:
  - **Serialization**: Converts interaction instances into JSON format for analytics.
  - **Deserialization**: Validates incoming data to log interactions in the system.
- **Fields**:
  - interaction_id: AutoField, read-only, the unique identifier for interactions.
  - user: ForeignKey, links to the User model.
  - course: ForeignKey, links to the Course model.
  - type: CharField, describes the type of interaction (e.g., viewed, completed).
  - timestamp: DateTimeField, records the time of the interaction.
- **Usage**: Used in the RecommendationView to log user actions and preferences.

URL Routing

User Authentication URLs (users/urls.py)

- **Djoser URLs**:
  - **Purpose**: Provides URL patterns for user authentication and account management.
  - **Routes**:
    - path('', include('djoser.urls')): Includes standard Djoser URL patterns for user management (registration, login, logout, password reset).
    - path('', include('djoser.urls.authtoken')): Includes Djoser URL patterns for token-based authentication.
    - path('auth/', include('djoser.urls.authtoken')): Specifically targets authentication actions to manage API token issuance, renewal, and invalidation.
  - **Methods Supported**: Varies by endpoint; typically GET and POST for retrieving user details and managing authentication tokens.

Recommendations URLs (recommendations/urls.py)

- **OpenAIPrompt URLs**:
  - **OpenAIPromptListCreateView**:
    - **Path**: prompts/
    - **View**: OpenAIPromptListCreateView.as_view()
    - **Purpose**: Handles the listing and creation of AI prompts.
    - **Methods**: GET (list prompts), POST (create a new prompt).
  - **OpenAIPromptDetailView**:
    - **Path**: prompts/<int:pk>/
    - **View**: OpenAIPromptDetailView.as_view()
    - **Purpose**: Provides detail view for a single prompt, including updating or deleting.
    - **Methods**: GET (retrieve details), PUT (update prompt), DELETE (remove prompt).
- **Recommendation URLs**:
  - **RecommendationView**:
    - **Path**: ``
    - **View**: RecommendationView.as_view()
    - **Purpose**: Fetches personalized course recommendations.
    - **Methods**: GET.
  - **RecommendationUpdate**:
    - **Path**: new
    - **View**: RecommendationUpdate.as_view()
    - **Purpose**: Updates or creates new recommendations.
    - **Methods**: POST.

Courses URLs (courses/urls.py)

- **CourseViewSet**:
  - **Router**: Uses Django Rest Framework's DefaultRouter to create routes for standard RESTful actions on the Course model.
  - **Endpoints Generated**:
    - **Root Path**: ``
    - **View**: CourseViewSet
    - **Purpose**: Manages all CRUD operations for courses.
    - **Methods Supported**:
      - GET (list all courses, retrieve details of a specific course)
      - POST (create a new course)
      - PUT/PATCH (update an existing course)
      - DELETE (remove a course)
  - **Automatic URL Routing**: The DefaultRouter automatically generates URLs for all CRUD operations, including listing all courses and detailed views for specific courses.

Authentication and Permissions

Authentication System

The Course Recommendation System employs token-based authentication to secure and manage user sessions. This section outlines how the system authenticates users and how it ensures secure API access.

- **Token-Based Authentication**:
  - **Mechanism**: When a user logs in, the system verifies their credentials (email and password) against the stored values in the database. Upon successful authentication, the system generates an authentication token using Djoser, which is then returned to the user.
  - **Token Usage**: This token must be included in the Authorization header of HTTP requests to access protected resources. The token is used to validate user sessions on subsequent requests without requiring re-authentication.
  - **Management**: Tokens are managed through Djoser's token management system, which provides endpoints for obtaining, renewing, and invalidating tokens.

Permissions

Permissions dictate what actions a user can perform within the system. Django's permissions framework, along with Django Rest Framework's (DRF) permissions classes, are utilized to manage access control based on user roles.

- **General Approach**:
  - Permissions are implemented at the view level in DRF. Each view can specify a permission_classes attribute that lists the permission classes that requests must pass to access the view.
  - **Example**: For CRUD operations on courses, permissions might restrict creation and deletion to users with admin privileges while allowing all authenticated users to read course details.
- **Custom Permissions**:
  - **Admin Permissions**: Admin users have unrestricted access to all endpoints. They can create, modify, and delete user accounts and courses, and access administrative features.
  - **Standard User Permissions**: Standard users can browse courses, view recommendations, and modify their profiles. They cannot access administrative functions or view other users' sensitive data.
  - **Anonymous Permissions**: Unauthenticated users are typically only allowed to view publicly accessible course content and must authenticate to interact with most parts of the system.
- **Implementation in Django**:
  - **PermissionsMixin**: The CustomUser model uses Django's PermissionsMixin, which provides fields and methods related to handling permissions and groups.
  - **Permission Classes in Views**: For example, a view handling course creation might use IsAdminUser from DRF to ensure only users marked as is_staff can create courses.

3. Frontend Development

Environment Setup

Setting up the development environment for the frontend involves installing Node.js, setting up React, and configuring additional necessary libraries and tools.

Dependencies Installation:

- **React Router** for routing: npm install react-router-dom
- **Axios** for HTTP requests: npm install axios
- **Redux** and **React Redux** for state management (if used): npm install redux react-redux
- **Material-UI** for UI components: npm install @mui/material @emotion/react @emotion/styled

Component Architecture

The React components are structured to maintain scalability and manageability:

- **Directory Structure**:
  - src/
    - components/: Contains all reusable UI components.
    - views/: Components representing entire views or pages.
    - layout/: Components that constitute parts of the application layout.
    - hooks/: Custom React hooks.
    - utils/: Utility functions and helpers.
    - services/: Services for interacting with backend APIs.
- **Component Types**:
  - **Stateful Components**: Manage state and business logic.
  - **Presentational Components**: Focus on UI rendering.
- **Interaction Patterns**: Components communicate via props for downward data flow and callbacks for events, ensuring separation of concerns and better maintainability.

State Management

- **Approach**: The system uses Redux for managing global state across the application.
    - **Store**: The central store holds the application state, accessible from anywhere in the component tree.
    - **Actions**: Defined to describe state changes.
    - **Reducers**: Specify how the state changes in response to actions.
- **Alternatives**:
    - **Context API**: Used for smaller applications or specific areas within the app where Redux might be overkill.

Routing

- **React Router**: Manages navigation between different components without reloading the browser.
    - Setup in src/index.js or src/App.js:

Integration with Backend

- **Axios** is used to make HTTP requests to Django's RESTful APIs.

4. API Documentation

Endpoint Descriptions

1. User Authentication Endpoints (via Djoser)

URL: /auth/token/login/

- **Method**: POST
- **Parameters**: None
- **Request Body**:

{"email": "user@example.com",`"password": "password123"}

- **Response**:

  {

  "auth_token": "9fbt3b...3nr39"

  }

Error Codes:

- **400 Bad Request:** Missing fields or wrong data format
- **401 Unauthorized:** Incorrect username or password

URL: /auth/token/logout/

- **Method**: POST
- **Parameters**: None
- **Request Body**: Requires token in the Authorization header
- **Response**: HTTP 204 No Content on successful logout
- **Error Codes**:
  - 401 Unauthorized: Missing or invalid token

2. Course Management Endpoints

URL: /courses/

- **Method**: GET
- **Parameters**: Optional query parameters for filtering and pagination (e.g., ?page=2)
- **Request Body**: None
- **Response**:

  [

  {

  "course_id": 1,

  "title": "Machine Learning",

```
    "description": "Learn Machine Learning!",
  },
  ...
]
```


Error Codes:

- **200 OK:** Successfully retrieved
- **500 Internal Server Error:** Server issues

URL: /courses/

- **Method**: POST
- **Parameters**: None
- **Request Body**:


```
{
  "title": "New Course",
  "description": "Detailed course description.",
  "category": "New Category"
}
```


- **Response**:


```
{
  "course_id": 101,
  "title": "New Course",
  "description": "Detailed course description.",
  "category": "New Category"
}
```

Error Codes:

- 201 Created: Successfully created
- 400 Bad Request: Missing or invalid fields

3. Recommendation Endpoints

URL: /recommendations/

- **Method**: GET
- **Parameters**: None
- **Request Body**: None
- **Response**:

```
[
  "Python for Data Science",
  "Advanced JavaScript Techniques"
]
```

Error Codes:

- 200 OK: Successfully retrieved recommendations
- 404 Not Found: No recommendations available

URL: /recommendations/new

- **Method**: POST
- **Parameters**: None
- **Request Body**: None (assumes data based on user's history)
- **Response**:

```
{
  "status": "success",
  "message": "New recommendations generated"
}
```

Error Codes:

- 200 OK: Recommendations successfully updated
- 500 Internal Server Error: Error during processing

5. Deployment

Server Configuration

Deploying the Course Recommendation System involves setting up a server environment that can robustly support both the Django backend and the React frontend. Here are the details on configuring the server:

- **Operating System**: Ubuntu 20.04 LTS or any compatible Linux distribution.
- **Web Server**: Nginx is recommended for serving the static files of the React application and as a reverse proxy for the Django application.
- **Application Server**: Gunicorn (a Python WSGI HTTP server) for running the Django application.
- **Database**: SQLite/PostgreSQL for data management. Ensure that the database is configured with adequate user permissions and secured access.
- **Python Environment**:
  - Use virtual environments (created via virtualenv) for Python dependency management.
  - Install all Python dependencies specified in the requirements.txt file.
- **Node.js Environment**:
  - Use Node.js to manage and build the React application.
  - Ensure all node packages are installed using npm install or yarn.

Docker Configuration

Using Docker, the application components can be containerized. This ensures that each part of the application runs in an isolated environment with its dependencies.

Components:

Django Backend Dockerfile:

```
FROM python:3.8-slim

ENV PYTHONDONTWRITEBYTECODE 1

ENV PYTHONUNBUFFERED 1

WORKDIR /app
```

```
RUN apt-get update && apt-get install -y --no-install-recommends \
    gcc \
    libc6-dev \
    && rm -rf /var/lib/apt/lists/*
COPY requirements.txt /app/
RUN pip install --upgrade pip && pip install -r requirements.txt
COPY . /app/  EXPOSE 8000
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

React Frontend Dockerfile:

```
FROM node:14-alpine
WORKDIR /app
ENV PATH /app/node_modules/.bin:$PATH
# Install app dependencies
COPY package.json ./
COPY package-lock.json ./
RUN npm install
COPY . ./
# Start the app
CMD ["npm", "start"]
```

Docker Compose File:

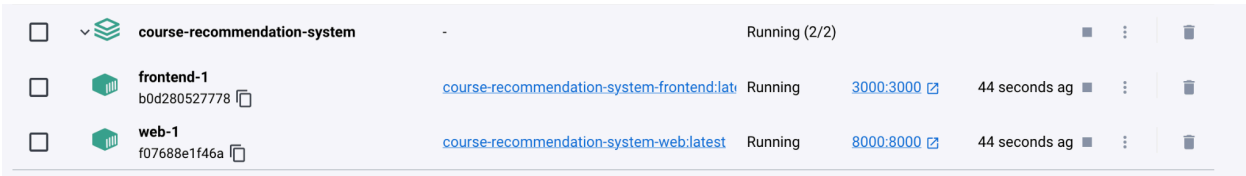
```
version: "3.8"
services:
  web:
    build:
      context: ./backend
      volumes: - ./backend:/app
      ports: - "8000:8000"
  frontend:
    build: context: ./frontend
    ports: - "3000:3000"
```

Build your containers

```
docker-compose build
```

Start your application

```
docker-compose up
```



Security Configurations:

- Implement SSL/TLS to secure HTTP requests using Let's Encrypt for certificate management.
- Configure firewalls to restrict access to unnecessary ports and ensure only HTTPS traffic is allowed through ports 443 (and 80 for redirection).

Continuous Integration and Deployment (CI/CD)

Setting up a CI/CD pipeline automates the process of testing, building, and deploying the application, which increases development productivity and reduces the likelihood of errors during deployment.

- **CI/CD Tools**:
    - **GitHub Actions**: For projects hosted on GitHub, it provides mechanisms to automate the workflow for software builds, tests, and deployments.