CM3035 Advanced Web Development Midterm Coursework - MovieWebApi

1. Introduction

This coursework is based on a Django-based web application that exposes a RESTful API for working with a movie and series dataset derived from the https://www.kaggle.com/datasets/muqarrishzaib/tmdb-10000-movies-dataset website, and there is no copyright as a result it is free to use the TMDB (The Movie Database). The dataset contains rich movie metadata such as title, language, overview, popularity scores, vote counts, and release dates. These attributes are vital for building systems related to movie recommendation, discovery platforms, or analytical dashboards.

The primary motivation behind this coursework is to showcase the practical use of Django and Django REST Framework (DRF) for building APIs that are robust, scalable, and production-ready. By integrating pandas, the system also supports preprocessing and bulk ingestion of CSV files. Users can interact with the system via a set of RESTful endpoints to perform standard actions such as retrieving all records, filtering based on conditions (e.g., popularity or language), and inserting new data. The application also includes a user interface via the Django Admin, comprehensive unit testing, and follows best practices in web API development.

This report outlines how the MovieWebApi project satisfies technical and structural requirements outlined by the R1-R4 criteria: model design, REST implementation, routing, validation, and testing. Additionally, this report explains how the application is set up, deployed, and maintained.

2. Dataset and Purpose

The MovieWebApi project is centered around a CSV dataset titled "TMDB 10000 Movies Dataset.csv". This file contains metadata for over 10,000 movies, including fields like id, title, original_title, original_language, release_date, overview, popularity, vote_average, and vote_count.

These fields were selected for their relevance in filtering and analysis, making it possible to construct endpoints such as 'Top Rated', 'Most Popular', 'Recent Movies', and 'Movies by Language'.

The project uses a custom data loader script (load_data.py) to preprocess and populate the database from this CSV. This is essential for managing inconsistencies in date formats, missing values, or malformed records. The script uses pandas to parse and transform the dataset before loading it into the SQLite database via Django's ORM.

This dataset provides a versatile foundation for building both interactive and analytical applications. It supports functionality such as search and recommendation systems, language-specific catalogs, and temporal analysis of releases. The API endpoints are designed to reflect these possible use cases.

3. Development Environment

The project was developed in the following environment:

- Operating System: Windows and macOS
- Python Version: 3.11
- Django Version: 5.0.6
- Django REST Framework Version: 3.15.1
- Pandas Version: 2.2.2

The application is deployed on **PythonAnywhere**, a cloud-based platform ideal for hosting Django apps. **SQLite3** is used as the default database, providing a lightweight, file-based storage mechanism that works well in small- to medium-scale applications. Virtual environments are used for dependency isolation. All packages are installed using the requirements.txt file, ensuring consistency across development and production environments.

The environment is highly portable. It can be run locally, deployed to servers, or used in containerized environments such as Docker with minimal configuration changes.

4. Project Structure and Core Files

The project is organized into a Django project directory and an application called MovieWebApi. The core directories and files are as follows:

- manage.py: Entry point to Django's command-line tools.
- MovieWebApiProject/: Root Django configuration directory containing settings.py, urls.py, and wsgi.py.
- MovieWebApi/: Contains the actual application logic.
- Key files inside MovieWebApi include:
- models.py: Defines the DataSet model used to store movie records.
- views.py: Implements the API views using Django REST Framework.
- serializers.py: Converts model instances to JSON and handles validation.
- urls.py: Maps URL paths to views.

- admin.py: Registers models with the Django admin.
- tests.py: Contains test cases for the API endpoints.
- migrations/: Stores database migration files.
- scripts/load_data.py: A standalone script used to bulk import the dataset into the database.

The logical separation of files ensures maintainability, testability, and scalability. For instance, the views are strictly focused on business logic and data handling, while the serializers ensure input and output data conform to expected formats.

5. Functional Requirements (R1–R4) Overview

This section details how the MovieWebApi application satisfies the four key requirements outlined: R1 through R4. These include appropriate usage of Django features (models, serializers, forms, testing), sound data modeling, complete REST endpoint implementation, and bulk data loading using pandas.

R1: Core Application Functionality

a) Correct Use of Models and Migrations:

The application uses Django’s ORM to define a model named `DataSet`, which represents a movie entry. The model includes typed fields such as CharField, FloatField, DateField, and IntegerField. These provide automatic validation, form generation, and query optimization. Database migrations are created using `makemigrations` and applied using `migrate`, ensuring all schema changes are version-controlled.

b) Serializers and Validation:

Django REST Framework serializers are used to transform querysets into JSON responses and vice versa. The `MovieSerializer` class inherits from ModelSerializer and exposes all fields. It validates POST data and ensures correct formats before insertion.

Example:

```python

class MovieSerializer(serializers.ModelSerializer):

    class Meta:

        model = DataSet

        fields = '__all__'
```

\`\`\`

c) Use of Django REST Framework:

Views are implemented using DRF's APIView class. Each endpoint is structured to handle GET or POST requests. Responses use DRF's `Response` object to format and return data with HTTP status codes.

d) URL Routing:

URLs are declared in a clear, REST-compliant manner. They map endpoints like `/movies/`, `/movies/popular/`, and `/movies/add/` to their respective views.

e) Unit Testing:

Unit tests in `tests.py` verify the correct behavior of endpoints. These use Django's TestCase and DRF's APIClient to simulate requests and assert correctness. Each test checks both the HTTP status code and the structure/content of the returned data.

R2: Data Modeling

The `DataSet` model is designed to be minimal yet expressive. It includes critical fields relevant for querying movie information. For example:

- `release_date` enables date-based filtering.
- `vote_average` and `vote_count` support quality-based ranking.
- `popularity` reflects trending status.

Each field in the model corresponds to a column in the dataset, ensuring a 1:1 mapping. The model adheres to normalization principles and is indexed implicitly by Django for fast retrieval. Additionally, the model is easy to extend. For future features, developers could add foreign key relationships (e.g., genres, tags) or computed fields (e.g., decade of release) without breaking the existing structure.

R3: REST Endpoints Implementation

The MovieWebApi application implements six REST endpoints, each serving a distinct purpose:

1. `/api/movies/`: Lists all movies in the database.

2. `/api/movies/popular/`: Returns the top 10 movies sorted by popularity.

3. `/api/movies/top-rated/`: Filters movies with a vote_average >= 8.0.

4. `/api/movies/language/?lang=en`: Returns movies matching the provided language code.

5. `/api/movies/recent/`: Filters movies released within the past 5 years.

6. `/api/movies/add/`: Accepts POST requests to add new movies.

Each endpoint uses appropriate HTTP methods (GET or POST) and validates input parameters. For instance, the language filter accepts `lang` as a query parameter, ensuring a clean and maintainable API.

Example of top-rated view:

```python
class TopRatedMoviesView(APIView):
    def get(self, request):
        movies = DataSet.objects.filter(vote_average__gte=8.0)
        serializer = MovieSerializer(movies, many=True)
        return Response(serializer.data)
```

These views rely on Django ORM queries, and serializers handle transformation and validation of the response payload.

R4: Bulk Data Loading

The bulk data loading mechanism is implemented using a script `load_data.py`, which reads the CSV file and writes its contents to the database. The script uses pandas to parse the dataset, normalize column names, and clean missing or invalid entries. It defines a helper function to parse inconsistent date formats. Invalid or malformed records are skipped.

Example of date parsing:

```python
def parse_date_with_fallback(date_str):
    try:
        return pd.to_datetime(date_str).date()
    except:
        return None
```

The script then iterates through rows and either inserts new entries or updates existing ones using `update_or_create`. This avoids duplication and ensures that repeated imports do not break data integrity. This is especially important for working with large datasets where performance, cleanliness, and fault-tolerance are critical.

6. Django Admin and Superuser Access

Django includes a built-in administrative interface that allows developers and site managers to view and modify database records. In MovieWebApi, the `DataSet` model is registered to the admin site through `admin.py`:

```python
from django.contrib import admin
from .models import DataSet
admin.site.register(DataSet)
```

This allows staff users to view all uploaded movies through a user-friendly table with sorting, searching, and pagination.

To access the admin interface:

1. Run `python manage.py createsuperuser`

2. Use the following credentials (as per project setup):

   - Username: admin

   - Password: admin123

3. Navigate to `http://127.0.0.1:8000/admin/` or the hosted domain `/admin/` path.

The admin interface is useful for debugging, managing content, and manually editing database entries in small datasets.

7. Running the Project and Unit Tests

To run the application locally:

1. Clone the repository and navigate into the project folder.

2. Create a virtual environment:

```bash
python -m venv venv
source venv/bin/activate  # or venv\Scripts\activate on Windows
```

3. Install dependencies:

```bash
pip install -r requirements.txt
```

4. Run database migrations:

```bash
python manage.py migrate
```

5. Create a superuser (if needed):

```bash
python manage.py createsuperuser
```

6. Load movie data:

```bash
python Scripts/load_data.py
```

7. Run the server:

```bash
python manage.py runserver
```

To run tests:

```bash
python manage.py test MovieWebApi
```

These tests validate endpoint functionality, including list retrieval, filtering, and movie insertion. Testing ensures the API functions as expected and supports future scalability.

8. Design Rationale and Code Organization

The application design follows Django best practices, ensuring logical separation of concerns:

- `models.py`: Data representation and schema.
- `views.py`: Business logic and request/response handling.
- `serializers.py`: Format transformation and input validation.
- `tests.py`: Automated endpoint verification.
- `load_data.py`: External dataset ingestion.

Views are kept lean and rely on serializers for validation, while queries are handled efficiently using Django ORM filters. API endpoints use REST conventions, and query parameters (e.g., `?lang=en`) are parsed directly from requests.

Additionally, Django's built-in features like middleware, URL routing, and settings configuration make the application scalable and secure. By following a modular approach, new features such as authentication, pagination, or third-party integration can be added without refactoring existing code.

9. Use Cases and Real-World Applications

This system can be extended into several production-ready applications:

- A public-facing movie catalog website with search and filtering.
- An API layer for a mobile app that helps users discover trending or top-rated content.

- An administrative dashboard for managing content in a private movie database.
- A recommendation engine integrated with user ratings and genres.

Future enhancements may include:

- User authentication and role-based access control.
- A frontend using React or Vue.js to consume the REST API.
- Advanced filters (by genre, actors, release decade, etc.).
- Designing the api webpage to make it more accessible and easier to use.

10. Conclusion

To conclude the MovieWebApi project is a fully functional Django REST API that provides an efficient and user-friendly interface to interact with a rich dataset of movies. It is a well-engineered Django REST application that demonstrates the use of models, serializers, URL routing, and testing to create a maintainable and scalable backend system.

Which fulfills all R1–R4 requirements:

- R1: Full use of Django models, forms (via serializers), DRF, and testing.
- R2: An expressive and normalized data model suited to the dataset.
- R3: A RESTful API with logical, usable endpoints.
- R4: A bulk-loading mechanism that is fault-tolerant and scalable.