Advanced Web Development - Midterm Report

Introduction

This report details the development of a RESTful API web service to access a comprehensive movie dataset sourced from TMDb (The Movie Database). The service is built using Django and the Django REST framework, facilitating efficient data management and robust endpoints for querying, adding, updating, and deleting movie information.

Dataset Discussion

The chosen dataset, sourced from TMDb (The Movie Database), comprises a collection of movie information. This dataset is particularly interesting due to its breadth and depth, encompassing various attributes such as:

- **Original Title:** The original name of the movie.
- **Genres:** A list of genres (e.g., Drama, Adventure, Science Fiction).
- **Popularity:** A numerical metric reflecting the movie's popularity.
- **Budget:** The movie's production budget.
- **Revenue:** The movie's box office revenue.
- **Cast:** A list of actors involved in the movie.
- **Homepage:** The official website URL of the movie.
- **Director:** The name of the movie's director.
- **Tagline:** A brief promotional tagline for the movie.
- **Keywords:** Relevant keywords associated with the movie.
- **Overview:** A summary of the movie's plot.
- **Runtime:** The movie's duration in minutes.
- **Production Companies:** The companies involved in producing the movie.
- **Release Date:** The date when the movie was released.

This dataset offers a rich foundation for exploring various aspects of movies, ranging from basic information retrieval to more complex analysis based on genre, cast, director, and financial performance.
It is particularly interesting due to its potential for both simple information retrieval (e.g., "Which movies were directed by Christopher Nolan?") and more complex analyses (e.g., "What is the average budget for science fiction movies released in the last decade?"). It opens doors for exploring trends, identifying patterns, and drawing insights into the film industry.

The movies dataset's unique value lies in its ability to support diverse queries:

- Simple Questions: Find all movies of a particular genre or directed by a specific director.
- Complex Analysis: Analyze genre trends, identify the most profitable directors, track the evolution of movie budgets over time.

Endpoints

GET /api/movies/:

- Example: /api/movies/
- Purpose: Retrieve a list of all movies (API endpoint).
- Interesting Aspects:
    - Filtering: Filter by genre, actor, director, year range, popularity range, budget range, revenue range, runtime range, or search query.
    - Sorting: Sort by popularity, title, release date, etc.
    - Pagination: Get paginated results with customizable page size.

GET /api/movies/<movie_id>/:

- Example: /api/movies/21/
- Purpose: Get details of a specific movie by its ID.
- Interesting Aspects:
    - Returns detailed information about a movie, including its genres, cast, and production companies.
    - Useful for displaying individual movie pages.

DELETE /api/movies/<movie_id>/:

- Example: /api/movies/31/
- Purpose: Delete a specific movie by its ID.
- Interesting Aspects:
    - Allows for removing movies from the database.
    - Requires appropriate authentication/authorization (not implemented in this project).
    - Redirects the user to the movie list, if the deletion was successful, if the deletion was unsuccessful it will show an error message.

GET /api/most_popular/?top_n=<number>:

- Example: /api/most_popular/?top_n=3
- Purpose: Get the top 'n' movies by popularity.
- Interesting Aspects:

    - Allows the user to easily identify the most popular movies in the dataset.
    - Can be used to create "Top 10 Movies" or similar lists.

GET /api/genre/?genre=<genre_name>:

- Example: /api/genre/?genre=Adventure
- Purpose: Filter movies by genre.
- Interesting Aspects:
    - Enables exploring movies within specific genres.
    - Can be used to create genre-specific movie lists or recommendations.

GET /api/actor/?actor=<actor_name>:

- Example: /api/actor/?actor=Tom%20Cruise
- Purpose: Filter movies by actors.
- Interesting Aspects:
    - Allows users to discover all movies an actor has been in.
    - Can be used to create actor-specific pages or filmographies.

Not API Endpoints:

GET /api/movie_list/:

- Purpose: Retrieve a list of all movies (not an API endpoint, used for the web interface).

GET /api/movie_detail/:

- Purpose: Retrieves the information of an individual movie (not an API endpoint, used for the web interface).

Method	Endpoint	Purpose	Interesting Aspects
GET	**/api/movies/**	**Retrieve a list of all movies.**	**- Filtering (by genre, actor, director, year, popularity, budget, revenue, runtime). - Sorting (popularity, title, release date). - Pagination (customizable page size).**
GET	**/api/movies/<movie_id>/**	**Retrieve detailed information about a specific movie by its ID.**	**Returns comprehensive movie details, including nested information about genres, cast, and production companies.**
DELETE	**/api/movies/<movie_id>/**	**Delete a movie from the database.**	**Handles CSRF protection for security. Redirects to movie list on success or displays an error message on failure.**

GET	**/api/most_popular/?top_n=<number>**	**Retrieve the top n movies based on popularity.**	**Allows easy identification of trending movies and supports the creation of "Top 10" style lists.**
GET	**/api/genre/?genre=<genre_name>**	**Filter movies by genre.**	**Enables users to explore movies within specific genres or build genre-based recommendations.**
GET	**/api/actor/?actor=<actor_name>**	**Filter movies featuring a specific actor.**	**Allows users to find all movies associated with a particular actor, building a filmography.**
GET	**/api/movie_list/**	**Non-API Endpoint: Renders the movie list in a web page with pagination.**	**This endpoint is specifically designed to render the HTML template and display the movie list with pagination controls using Django's templating system.**
GET	**/api/movie_detail/<movie_id>/**	**Non-API Endpoint: Renders the detail view for a specific movie, including navigation to previous/next movies.**	**This endpoint leverages Django's ORM to retrieve the movie details and adjacent movies for navigation. It provides a rich display of information for a single movie and includes a delete button to remove the movie from the database, triggering a DELETE API request.**

Application Requirements

The application meets the core requirements:

Models & Migrations:

Well-defined Django models represent entities like Movie, Genre, Actor, etc. Migrations are used to create and manage the database schema.

```
  # models.py
from django.db import models

class Genre(models.Model):
   name = models.CharField(max_length=50, unique=True)

   def __str__(self):
       return self.name

class ProductionCompany(models.Model):
   name = models.CharField(max_length=100, unique=True)
```

```
    def __str__(self):
        return self.name

class Actor(models.Model):
    name = models.CharField(max_length=100)

    def __str__(self):
        return self.name

class Director(models.Model):
    name = models.CharField(max_length=100)

    def __str__(self):
        return self.name

class Movie(models.Model):
    original_title = models.CharField(max_length=200)
    genres = models.ManyToManyField(Genre)
    # Allowing null for missing values
    popularity = models.FloatField(null=True, blank=True)
    budget = models.PositiveIntegerField(null=True, blank=True)
    revenue = models.PositiveIntegerField(null=True, blank=True)
    cast = models.ManyToManyField(Actor)
    homepage = models.URLField(blank=True, null=True)
    director = models.ForeignKey(
        Director, on_delete=models.SET_NULL, null=True, blank=True)
    tagline = models.TextField(blank=True, null=True)
    keywords = models.TextField(blank=True, null=True)
    overview = models.TextField(blank=True, null=True)
    runtime = models.PositiveIntegerField(null=True, blank=True)
    production_companies = models.ManyToManyField(ProductionCompany)
    release_date = models.DateField(null=True, blank=True)

    def __str__(self):
        return self.original_title
```

Forms, Validators, and Serialization

While not explicitly required for the REST API endpoints, the project also considers the scenario of adding a front-end interface for data entry and updates. To handle this, Django's forms, validators, and serializers are being used.

A form (MovieForm) with validation is implemented for adding new movies through a simple UI. Serializers (MovieSerializer) handle the conversion between Python objects and JSON.

Forms:

MovieForm class, inherited from django.forms.ModelForm, is created to define the input fields for adding or updating movie information. This form will automatically generate the HTML form structure and handle basic validation (e.g., required fields, field types).

Validators:

Custom validators have been added to the MovieForm to enforce specific data constraints. For instance, a validator is added to check if the release date is not in the future.

```
# Movie list add movie form
from django import forms
from django.core.exceptions import ValidationError
from django.utils import timezone
from .models import Movie, Genre, Actor, ProductionCompany, Director
import datetime

class MovieForm(forms.ModelForm):
   genres = forms.ModelMultipleChoiceField(queryset=Genre.objects.all())
   cast = forms.ModelMultipleChoiceField(queryset=Actor.objects.all())
   production_companies = forms.ModelMultipleChoiceField(
       queryset=ProductionCompany.objects.all())
   director = forms.ModelChoiceField(queryset=Director.objects.all())

   class Meta:
       model = Movie
       fields = ['original_title', 'genres', 'popularity', 'budget', 'revenue',
'cast', 'homepage',
                 'director', 'tagline', 'keywords', 'overview', 'runtime',
'production_companies', 'release_date']

   def clean_release_date(self):
       release_date = self.cleaned_data.get('release_date')
       if release_date and release_date > datetime.date.today():
           raise ValidationError('Release date cannot be in the future.')
       return release_date
```

```
<!DOCTYPE html>
<html>
```

```
<head>
  <title>Add Movie</title>
</head>
<body>
  <h1>Add New Movie</h1>

  <form method="POST">
    {% csrf_token %} {{ form.as_p }}

    <button type="submit">Add Movie</button>
  </form>
</body>
</html>
```

Serialization:

Django's serialization framework is used to convert movie objects into JSON format when sending data to the front-end interface. Conversely, when receiving data from the front-end, the serializer is used to validate and convert the JSON data back into Python objects.

Django REST Framework:

The project effectively uses Django REST framework to build the API, including serializers, viewsets, and generic views.

URL Routing:

Clean and organized URL patterns (movieNight/urls.py, movieList/urls.py) facilitate easy navigation and endpoint access.

Unit Testing:

To ensure the correctness and robustness of the REST API, comprehensive unit tests have been developed. These tests cover various scenarios, including:

- Success Scenarios: Tests verify that valid requests to each endpoint return the expected data format and values.
- Error Handling: Tests simulate error conditions (e.g., requesting a movie that doesn't exist) and check that the API responds with appropriate error codes and messages.
- Edge Cases: Tests for handling empty result sets, invalid input parameters, and large datasets are included.

The unit tests are written using Django's built-in testing framework and leverage the APIClient from Django REST framework to simulate API requests. Test data is created for each test case, and the responses from the API are evaluated to ensure they match the expected output.

```
def test_movie_detail_endpoint_exists(self):
        """Test that the movie_detail endpoint exists for the first movie."""
        movie_id = self.movies[0].id
        response = self.client.get(f"/api/movie_detail/{movie_id}/")
        self.assertEqual(
            response.status_code, 200
        )  # Check for a successful response

    def test_get_movie_list(self):
        """
        Test retrieving a list of movies using the movie_list view.
        """
        response = self.client.get("/api/movie_list/")
        self.assertEqual(response.status_code, 200)
        # Check correct template
        self.assertTemplateUsed(
            response, "movieList/movie_list.html"
        )

    def test_get_movie_detail_status_code(self):
        """
        Test getting details of a movie with a valid ID, and check if the correct
status is returned
        """
        movie_id = self.movies[0].id  # Choose the first movie from the test data
        response = self.client.get(f"/api/movie_detail/{movie_id}/")
        self.assertEqual(
            response.status_code, 200
        )  # Check for a successful response

    # Testing pagination
    def test_pagination(self):
        # Order by popularity descending
        response = self.client.get("/api/movies/")
        self.assertEqual(response.status_code, status.HTTP_200_OK)
```

```
......
----------------------------------------------------------------------
Ran 20 tests in 3.393s

OK
Destroying test database for alias 'default'...
```

Bulk Data Loading:

A custom management command (load_data) efficiently loads movie data from the CSV file into the database.

Command: *python manage.py load_data*

Application of Techniques

The project employs good practices and key techniques from Django and Django REST framework to create a robust and efficient API. The following sections elaborate on these techniques.

The project employs several key techniques:

- ## ModelViewSet:

  Simplifies the creation of API endpoints for the Movie model, providing CRUD (create, read, update, delete) operations out of the box.

- ## Generic Views:

  ListAPIView and RetrieveAPIView are used for specific endpoints (e.g., top-rated movies, filtering), making the code more focused and reusable.

- ## Django Admin Customization:

  The Django admin interface is enhanced with inline editing for related models and custom CSS for improved aesthetics.

- ## Filtering and Pagination:

  The django-filter library enables filtering movies by various criteria, and pagination ensures efficient display of large datasets.

```python
class MovieList(generics.ListAPIView):
```

```
    queryset = Movie.objects.all()
    serializer_class = MovieSerializer
    filter_backends = [filters.DjangoFilterBackend]

    filterset_class = MovieFilter

    pagination_class = PageNumberPagination

def movie_list(request):
    # Get all movies
    all_movies = Movie.objects.all()

    # Set up pagination
    # Show 50 movies per page
    paginator = Paginator(all_movies, 50)
    # Get page number from request
    page_number = request.GET.get("page", 1)
    # Get the page of movies
    page_obj = paginator.get_page(page_number)

    return render(request, "movieList/movie_list.html", {"movies": page_obj})
```

Delete Functionality:

The project utilizes Django REST Framework's built-in capabilities of the ModelViewSet to provide a DELETE endpoint for removing movies from the database. This is implemented in the MovieViewSet without requiring additional custom views.

Delete functionality - Frontend application:

The frontend implementation of the delete functionality is handled by a JavaScript function called deleteMovie. This function is attached to the "Delete" button on the movie detail page. When a user clicks this button:

1. **Confirmation:** A confirmation dialog is presented to the user to ensure they intend to delete the movie.
2. **API Request:** If confirmed, the function sends a DELETE request to the /api/movies/<movie_id>/ endpoint. This request includes a CSRF token in the headers for security.
3. **Handling Response:**
    a. **Success:** If the deletion is successful (status code 204), the user is redirected to the movie list page (/api/movie_list/).
    b. **Failure:** If the deletion fails, an error message is displayed to the user.

```
<script>
    async function deleteMovie(movieId) {
      if (confirm("Are you sure you want to delete this movie?")) {
        const response = await fetch(`/api/movies/${movieId}/`, {
          method: "DELETE",
          headers: {
            "Content-Type": "application/json",
            "X-CSRFToken": csrftoken,
          },
        });

        if (response.ok) {
          window.location.href = "{% url 'movie_list' %}";
        } else {
          alert("Failed to delete movie.");
        }
      }
    }
  </script>
```

CSRF Protection:

The project incorporates Cross-Site Request Forgery (CSRF) protection by including CSRF tokens in the delete requests made from the frontend. This is essential for security in web applications.

Enhanced Django Admin Interface

In addition to the core functionality of the REST API, the project also provides a user-friendly Django admin interface for managing movie data.

This interface is enhanced with the following features:

Inline Admin:

To streamline data management, inline admin functionality is implemented. This allows for the efficient editing of related objects directly within the parent object's admin page. Specifically, when editing a movie in the admin interface, the user can now also modify the associated genres, actors, and production companies without having to navigate to separate admin pages.

This improves the overall user experience and simplifies the process of maintaining the movie database.

Customization (CSS):

The Django admin interface has been customized to improve its visual appeal and user experience. A custom CSS file is added to the project (movieList/static/css/admin.css) to override the default styles. This customization includes changes like modifying the header background color, adding a custom logo, and adjusting the font styles. These modifications enhance the aesthetic appeal of the admin interface and create a more personalized look and feel.

Swagger/OpenAPI Integration (C7)

To enhance the developer experience and improve the API's usability, Swagger/OpenAPI integration has been implemented using the drf_yasg library. This provides interactive documentation that allows developers to:

Explore Endpoints: View a list of available API endpoints with their parameters and descriptions.
Test Requests: Send sample requests directly from the documentation to see the API response.
Generate Client Code: Easily generate client code in various languages to interact with the API.

By integrating Swagger/OpenAPI, the API becomes more self-documenting and easier for others to consume, contributing to its overall maintainability and success.

Critical Evaluation

The project successfully delivers a RESTful API for interacting with movie data. The code is well-structured, following best practices. The test suite is robust, covering various scenarios and edge cases.

The project could be further enhanced by:

- Implementing more advanced features like user authentication and authorization to restrict access to certain endpoints or actions (e.g., deleting movies).
- Adding more sophisticated filtering and search options (e.g., full-text search, faceted search).
- Implementing a frontend interface using a JavaScript framework like React or Vue.js to provide a more interactive user experience.

Run Information

This section contains information including:

Operating system: macOS Sonoma 14.5
Python version: 3.8.3

Dependencies and version

How to install the dependencies: *pip install -r requirements.txt*
Django version:

Django Admin Credentials:

Username: marieid
Password: *Degmont7?*