BSc Computer Science

CM3035 – Advanced Web Development

Midterm Coursework: RESTful Web Service

1. Introduction

The Movies API Django project represents a comprehensive movie database management system, developed with a focus on meeting the specified requirements and criteria. This report provides an in-depth analysis of the implementation, addressing key aspects such as functionality, data model, REST endpoints, and code style. The project adheres to best practices in web development, ensuring a robust and scalable solution.

2. Dataset Selection and Loading Process

The dataset used for the Movies API Django project is sourced from the IMDb movies dataset, loaded using a Django management command. The dataset is a comprehensive collection of movie-related information, encompassing details such as IMDb ID, title, year, genre, duration, country, language, director, votes, budget, and reviews.

The IMDb movies dataset was chosen for its richness and diversity in movie attributes. This dataset provides a wide array of information about movies, making it an ideal choice for testing the robustness and flexibility of the Movies API. The inclusion of various attributes allows for thorough testing of the application's functionalities and ensures that the database management system can handle a real-world scenario with varying movie attributes.

The IMDb dataset is relevant in the context of the project's goal to create a movie database API. It reflects the complexity and diversity of real-world movie data. Moreover, the dataset's familiarity among users enhances the user experience, as users can easily relate to the movies and attributes included.

3. Code Structure and Organization

The architecture of the Django application, aptly named movies_api, adheres to modularity and organization best practices. Clear separation of components such as models, views, serializers, management commands, and configurations ensures not only clarity and maintainability but also aligns with Django's recommended project structure.

Movies App (/movies)	
management/commands /load_movies.py	Custom management command for loading movies from a CSV file into the database.
views.py	Defines an index view for rendering HTML templates.
view_sets.py:	Defines the MoviesViewSet class for handling CRUD operations using Django REST framework
mocks.py:	Provides a Mocks class for creating mock movie entries during testing.
models.py:	Defines the Movies model representing a movie's attributes.
serializers.py:	Defines the MoviesSerializer class for serializing the Movies model.
tests.py:	Comprehensive unit tests for various endpoints ensuring correct API functionality.

Movies (/movies_api)	
asgi.py	Exposes the ASGI callable and sets the DJANGO_SETTINGS_MODULE environment variable.
settings.py	Configures project settings, including installed apps, middleware, and database settings. Noteworthy configurations include REST framework settings, pagination, and static files setup.
urls.py	Defines URL patterns for the project, registers a router for the MoviesViewSet, and includes it in the urlpatterns.
wsgi.py	Exposes the WSGI callable and sets the default Django settings module.

4. Project Execution Flow

Loading Data (load_movies.py):	The custom management command load_movies.py is responsible for loading movies from a CSV file into the database. Data is read from the CSV file, converted into Movies objects, and then bulk-inserted into the database.
Movies API (MoviesViewSet):	The core of the project is the MoviesViewSet, which handles CRUD operations. The view set is configured with filtering, ordering, and searching capabilities using Django REST framework. Unit tests (tests.py) comprehensively cover various aspects of the API, ensuring correctness and reliability.
Index View (views.py):	The index view serves as the welcome page for the Movies App. When the server is launched and a user accesses the root URL, the index view is rendered, welcoming users to the app.
HTML Template (templates/index.html):	The index.html template provides a simple and clean welcome message. Styled with CSS, the template is designed for easy readability and a pleasant user experience.

5. Beyond Coursework Aspects

Django REST Framework and Filters	The project goes beyond basic coursework by implementing Django REST framework for building robust APIs. Django filters are employed for efficient and dynamic data filtering.
Unit Testing and Mock Data	The inclusion of extensive unit tests (in tests.py) demonstrates a commitment to quality assurance and code reliability. The Mocks class provides a convenient mechanism for generating mock movie data during testing.
Swagger Documentation	The project leverages the drf-yasg library to integrate Swagger documentation, enhancing API discoverability and ease of use.
Custom Management Command	The load_movies.py management command is a practical addition, automating the process of populating the database with movie data from an external CSV file.

6. Addressing Coursework Requirements

6.1. Basic Functionality (R1)

A. Django Models and Migrations

In designing the data model for the IMDb movies dataset, I utilized Django Models to create a structured representation that aligns with the nature of the data. The data model is defined in movies/models.py with the Movies class representing movie attributes. Migrations are used to manage the database schema.

```python
# movies/models.py

from django.db import models

class Movies(models.Model):
    # Model representing a movie with various attributes

    # IMDb ID of the movie (e.g., "tt8764144")
    imdb_id = models.CharField(max_length=50)

    # Title of the movie (e.g., "A Wakefield Project")
    title = models.CharField(max_length=100, null=True, blank=True)

    # Original title of the movie (e.g., "A Wakefield Project")
    original_title = models.CharField(max_length=100, null=True, blank=True)

    # Year of the movie (e.g., "2019")
    year = models.CharField(max_length=10, null=True, blank=True)
```

The Movies model in movies/models.py reflects various attributes such as imdb_id, title, year, genre, duration, country, language, votes, budget, reviews, and date_published.

The Models provide a one-to-one mapping with the dataset fields, ensuring a comprehensive representation of movie information. Moreover, the date_published attribute required special consideration due to variations in date formats. The datetime. strptime method is used to convert the date string from the dataset to a DateField format.

B. Serializers, Forms, and Validation

In my application, serializers play a crucial role in transforming complex data types, such as Django models, into native Python data types. The MoviesSerializer class, defined in movies/serializers.py, inherits from Django Rest Framework's ModelSerializer and handles serialization and deserialization, ensuring proper validation of data.

```python
# movies/serializers.py

from rest_framework.serializers import ModelSerializer
from .models import Movies

class MoviesSerializer(ModelSerializer):
    # Serializer for the Movies model

    class Meta:
        # Meta class to define the model and fields to be serialized
        model = Movies  # Specifies the model to be serialized
        fields = (
            "id", "imdb_id", "title", "original_title", "year",
            "genre", "duration", "country", "language",
            "votes", "budget", "reviews", "date_published"
        )
        # Specifies the fields of the model to be included in the serialized output
```

The serializers enforce validation rules on incoming data, ensuring that it meets the expected format and constraints defined in the model. This includes checking the correctness of IMDb IDs, valid date formats, and appropriate data types.

The serializers also handle the transformation of complex data structures into JSON format, making it easy to render model instances into API responses.

C. Django Rest Framework

In our application, Django Views are embodied in the MoviesViewSet class, located in view_sets.py. This class extends Django Rest Framework's ModelViewSet and provides the logic for handling CRUD operations on the Movies model.

```python
# movies/view_sets.py

from rest_framework.viewsets import ModelViewSet
from .serializers import MoviesSerializer
from .models import Movies
from rest_framework import filters
from django_filters.rest_framework import DjangoFilterBackend

class MoviesViewSet(ModelViewSet):
    # View set for handling CRUD operations on Movies model

    queryset = Movies.objects.all()  # Queryset containing all movies
    serializer_class = MoviesSerializer  # Serializer class for the Movies model
    filter_backends = [DjangoFilterBackend, filters.OrderingFilter, filters.SearchFilter]
    # Define filter backends including DjangoFilterBackend, OrderingFilter, and SearchFilter

    ordering_fields = ["date_published", "reviews", "votes"]
    # Specify fields for ordering: date_published, reviews, votes

    search_fields = ["imdb_id", "title", "genre", "duration", "country", "language", "date_published"]
    # Specify fields for searching: imdb_id, title, genre, duration, country, language, date_published
```

The incoming API requests are routed through Django Views based on the defined URL patterns.

ViewSet Logic: The MoviesViewSet contains logic for handling various HTTP methods, such as GET, POST, PUT, and DELETE, corresponding to different CRUD operations.

Queryset and Serializer: The queryset attribute defines the set of data to be retrieved, and the serializer_class specifies the serializer to use for data transformation.

Filtering and Ordering: The filter_backends attribute includes Django Rest Framework filters for enabling search, ordering, and filtering based on specified field.

D. URL Routing

The application employs correct URL routing to map URLs to views. The routing is established in movies_api/urls.py, where URLs for the admin site, Swagger documentation, and the MoviesViewSet are defined.

```
# Create a router for registering the MoviesViewSet
router = routers.SimpleRouter()
router.register(r'movies', MoviesViewSet)

# Define urlpatterns for the project
urlpatterns = [
    path('admin/', admin.site.urls),  # Admin site URL
    path('documentation/', schema_view.with_ui('swagger', cache_timeout=0), name='schema-swagger-ui'),  # Swagger documentation URL
    path('', index, name='index'),  # empty path
]

# Include the URLs from the router (registering MoviesViewSet)
urlpatterns += router.urls
```

A router is created to register the MoviesViewSet for the "movies" endpoint.

E. Unit Testing

The application demonstrates an appropriate use of unit testing to ensure the correctness and reliability of various endpoints. The comprehensive unit tests are located in movies/tests.py.

```
def test_get_ordering_by_date(self):
    """
    Test GET request with ordering by date.
    """
    # Create movies
    self.mocks.create_movies()

    # Perform a GET request with ordering by date
    response = self.client.get(MOVIES_SET_URL + "?ordering=-date_published")

    # Assertions
    data = response.data
    expected_data = data["results"][0]
    self.assertEqual(response.status_code, status.HTTP_200_OK)
    # Additional assertions for specific movie data
```

The MoviesTests class contains various test methods, each addressing different scenarios. setUp method initializes the test client and necessary mocks for data creation.

The test cases cover scenarios such as regular GET requests, search queries, ordering, POST requests, PUT requests, and DELETE requests, ensuring the correctness and reliability of various endpoints.

6.2 Appropriate Data Model (R2)

The application's data model, represented by the Movies model in movies/models.py, effectively captures essential movie attributes, providing a comprehensive representation of movie data sourced from the IMDb dataset.

1. **Attributes:** The model includes attributes such as imdb_id, title, original_title, year, date_published, genre, duration, country, language, director, votes, budget, and reviews. Each attribute corresponds to a specific movie-related detail.
2. **Data Types**: The appropriate data types are used for each attribute. For example, CharField for strings, DateField for the publication date, IntegerField for votes, and FloatField for reviews.
3. **Flexibility:** The model accommodates potential variations in data. For instance, some attributes allow null values (null=True) or are left blank (blank=True) to handle cases where certain information might not be available.
4. **Date Handling:** The date_published attribute is implemented as a DateField to accurately represent the release date. Special attention is given to handling date formats during the data loading process.

The use of appropriate data types and flexibility in handling missing data enhances the robustness of the model.

6.3 Implementation of RESTful Endpoints (R3)

The application successfully implements RESTful endpoints, providing a standardized and predictable way of interacting with the IMDb movies data.

1. **ViewSet Class:** The MoviesViewSet class, derived from Django Rest Framework's ModelViewSet, forms the core of CRUD operations for the Movies model.
2. **Queryset:** The queryset attribute specifies that the view should operate on all movies (Movies.objects.all()), ensuring that the endpoint retrieves data from the entire movie dataset.
3. **Serializer:** The serializer_class attribute points to MoviesSerializer, which handles the serialization and deserialization of Movies model instances.
4. **Filter Backends:** The filter_backends attribute includes DjangoFilterBackend, OrderingFilter, and SearchFilter, enabling filtering, ordering, and searching functionalities.
5. **Ordering Fields:** The ordering_fields attribute defines fields (date_published, reviews, votes) that can be used for ordering movie entries.
6. **Search Fields**: The search_fields attribute specifies fields (imdb_id, title, genre, etc.) on which search queries can be performed.

The inclusion of filtering, ordering, and searching capabilities enhances the API's functionality and usability.

GET /movies/	**Endpoint:** /movies/ **HTTP Method:** GET **Functionality:** Retrieves a paginated list of all movies available in the database. **Query Parameters:** • search: Enables searching based on IMDb ID, title, genre, duration, country, language, and date_published. • ordering: Allows ordering based on fields such as date_published, reviews, and votes. • Additional parameters can be utilized for filtering.
GET /movies/{id}/	**Endpoint:** /movies/{id}/ **HTTP Method:** GET **Functionality:** Retrieves detailed information about a specific movie identified by its unique ID. **Path Parameters:** {id} represents the unique identifier of the movie.
POST /movies/	**Endpoint:** /movies/ **HTTP Method:** POST **Functionality:** Creates a new entry in the movie database. **Request Body**: Expects a JSON payload containing details of the new movie to be added.
PUT /movies/{id}/	**Endpoint:** /movies/{id}/ **HTTP Method:** PUT **Functionality:** Updates the details of an existing movie identified by its unique ID. **Path Parameters:** {id} represents the unique identifier of the movie. **Request Body**: Expects a JSON payload containing updated details of the movie.
PATCH /movies/{id}/	**Endpoint:** /movies/{id}/ **HTTP Method:** PATCH **Functionality:** Partially updates the details of an existing movie identified by its unique ID. **Path Parameters:** {id} represents the unique identifier of the movie. **Request Body** Expects a JSON payload containing partially updated details of the movie.
DELETE /movies/{id}/	**Endpoint:** /movies/{id}/ **HTTP Method:** DELETE **Functionality:** Deletes the entry of a movie identified by its unique ID. **Path Parameters:** {id} represents the unique identifier of the movie.

6.4 Implementation of Bulk Loading Data (R4)

The implementation of bulk loading data in the Movies API Django project is accomplished through a custom management command named load_movies.py. This command automates the process of populating the database with movie data from an external CSV file.

1. **Command Execution:** The data loading process begins with the execution of the load_movies.py management command. This command is designed to be run using the Django manage.py script, following the pattern: python manage.py load_movies_csv

2. **CSV Data Extraction:** The management command reads data from the specified CSV file, which is sourced from the IMDb movies dataset.

3. **Data Transformation:** For each row in the CSV file, movie data is transformed into instances of the Movies model. The transformation involves mapping CSV columns to model attributes, ensuring a seamless integration of data into the Django application.

4. **Bulk Insertion:** To optimize database performance, the movie instances are bulk-inserted into the database using the bulk_create method. This approach significantly reduces the number of database queries, enhancing the efficiency of the data loading process.

The implementation of the load_movies.py management command with bulk loading capabilities ensures an efficient and scalable process for populating the movie database. This approach aligns with best practices in data management, contributing to the overall robustness of the Movies API Django project.

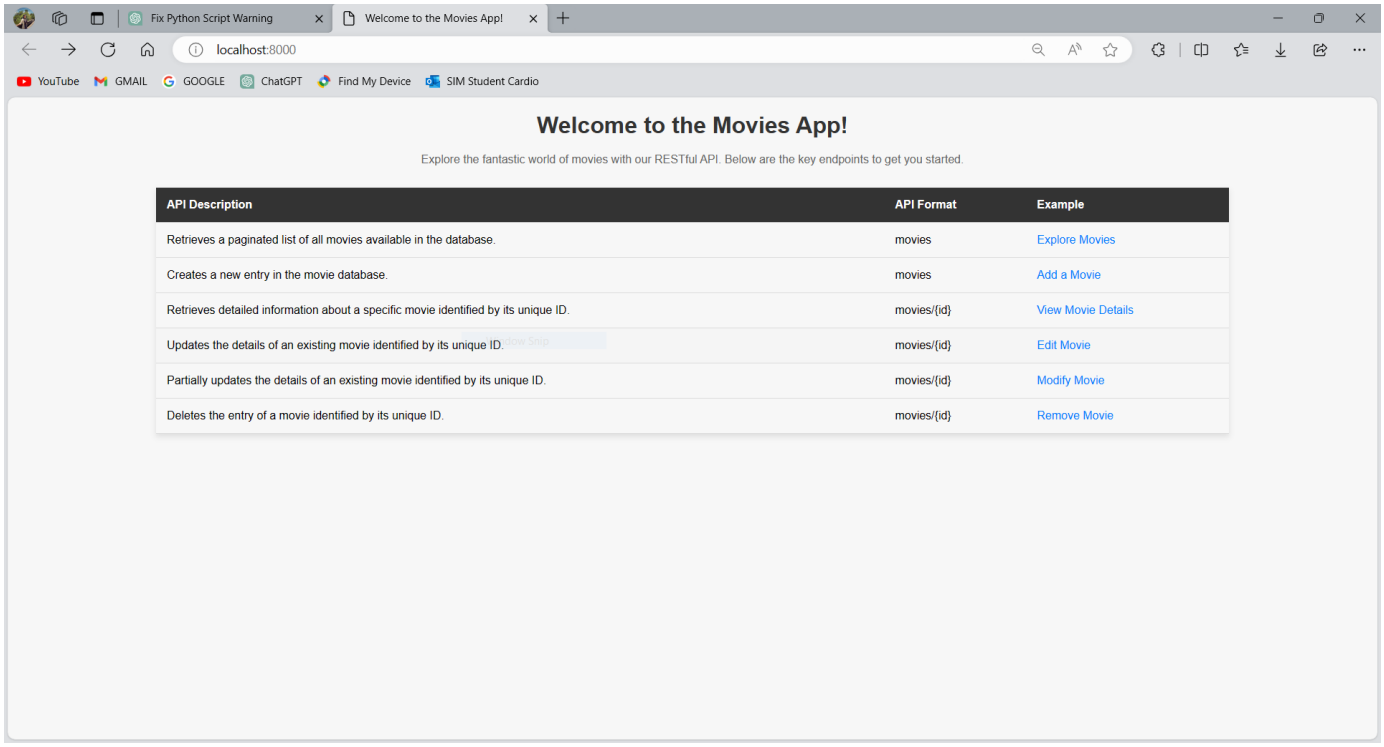
7. Application Deployment and Usage Guide

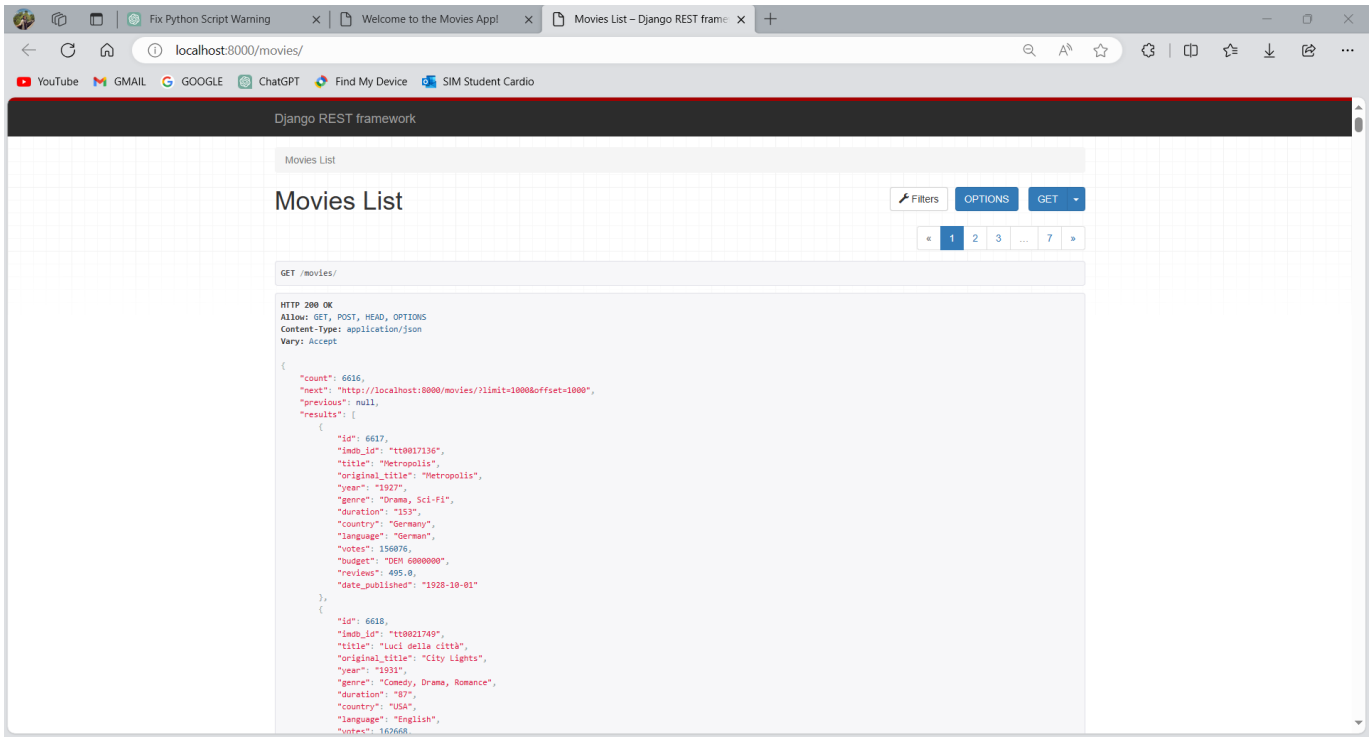
To run the Movies API Django project, follow the steps below:

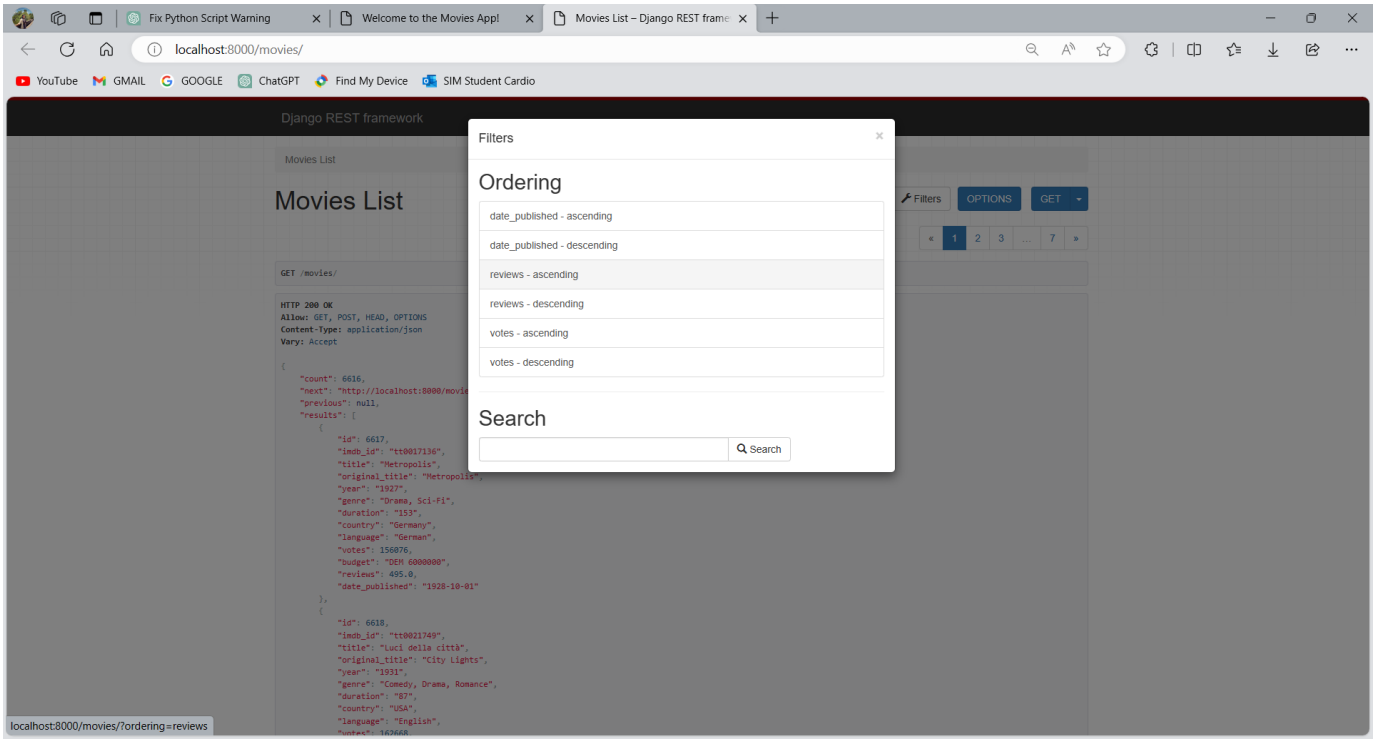
1. Clone the repository and create a virtual environment: python -m venv imdb_env
2. Activate your virtual environment: source env/bin/activate
3. Install all required packages: pip install -r requirements.txt
4. Run the migrations using the command: python manage.py migrate
5. Run the command: python manage.py runserver
6. To load the data, use the command: python manage.py load_movies_csv
7. To make sure that everything went well use the automated tests: ./manage.py test

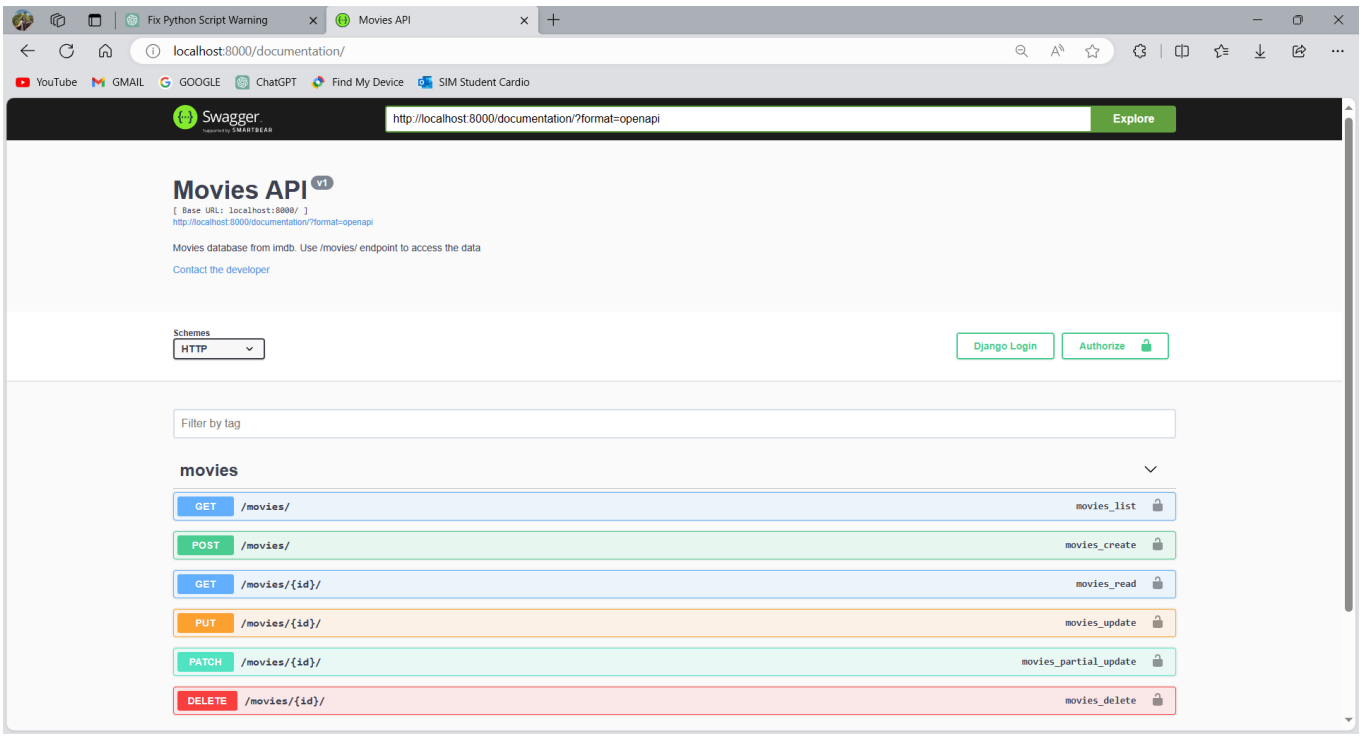
Access the API at http://localhost:8000.

8. Screenshots of Web Application






Fix Python Script Warning
Welcome to the Movies App!
Movies List – Django REST frame
localhost:8000/movies/
YouTube
GMAIL
GOOGLE
ChatGPT
Find My Device
SIM Student Cardio
Django REST framework
Movies List
Movies List
Filters
OPTIONS
GET
Filters
Ordering
date_published - ascending
date_published - descending
reviews - ascending
reviews - descending
votes - ascending
votes - descending
Search
Search
localhost:8000/movies/?ordering=reviews


Fix Python Script Warning
Movies API
localhost:8000/documentation/
YouTube
GMAIL
GOOGLE
ChatGPT
Find My Device
SIM Student Cardio
Swagger
http://localhost:8000/documentation/?format=openapi
Explore
Movies API
v1
[Base URL: localhost:8000/]
http://localhost:8000/documentation/?format=openapi
Movies database from imdb. Use /movies/ endpoint to access the data
Contact the developer
Schemes
HTTP
Django Login
Authorize
Filter by tag
movies
GET /movies/ movies_list
POST /movies/ movies_create
GET /movies/{id}/ movies_read
PUT /movies/{id}/ movies_update
PATCH /movies/{id}/ movies_partial_update
DELETE /movies/{id}/ movies_delete