UNIVERSITY
OF LONDON

CM3035 - Advanced Web Development

Midterm Report

Contents

1. Introduction

This report will discuss about the development of a RESTful web service movie application created using Django and Django Rest Framework (DHF). The report covers the project's basic functionalities, data models, REST endpoints, data loader as well as the instruction for running the web application. The dataset includes information of movies that were produced in 2023, such as genres, directors, stars, ratings and more, allowing a demonstration of the DHF's functionalities. This report will cover how the functionalities of the web application meet the task requirements.

2. Dataset

Dataset Information:

The dataset retrieved from (https://www.kaggle.com/datasets/cabo1234/imdb-movie-release-2023-dataset) consists of approximately 8,900 entries of movies released in 2023 that were recorded in IMDB. Each row represents a unique movie, while the columns represent the following:

- **title:** Stores the name of each movie.
- **date:** Stores the release date of each movie.
- **run_time:** Records the duration of each movie in minutes.
- **genre:** Records one or more genres associated with each movie.
- **rating:** Stores the audience rating for each movie on a scale of 1.0 to 10.0.
- **introduction:** Stores the synopsis of each movie.
- **director:** List the name(s) of the director(s) that directed each movie.
- **stars:** List the name(s) of the actor(s) acted in each movie.
- **num_votes:** Records the number of audience's votes a movie received.

Dataset Discussion:

The dataset is interesting due to the wide range of genres it covers, such as Comedy, Horror, Drama, and more. Most movies consist of multiple genres, reflecting the trend of multi-genre films. This shows the evolution of the movies where multiple genres are blended to create an innovative film. Additionally, this dataset also includes movies from various regions like Vietnam, Pakistan and more, ensuring the dataset isn't bias to a specific region. Furthermore, some movies also feature someone playing as both director and star of a show, showcasing its talent. In addition, the ratings of each show in this dataset vary significantly as well, providing insight into audience preferences.

3. Models

Genre, Director, and Star Model:

The Genre, Director and Star model stores all the genres, directors and stars that appears in the dataset. They have name field with the CharField type that stores the name of their respective data. These field has unique set to true so there won't be any duplicated entries being stored.

Movie model:

```python
class Movie(models.Model):
    title = models.CharField(max_length=200)
    release_date = models.DateField(null=True, blank=True)
    run_time = models.IntegerField(null=True, blank=True)
    genre = models.ManyToManyField(Genre, related_name='movies')
    rating = models.DecimalField(max_digits=3, decimal_places=1, null=True, blank=True)
    introduction = models.TextField(null=True, blank=True)
    director = models.ManyToManyField(Director, related_name='movies')
    stars = models.ManyToManyField(Star, related_name='movies')

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

The Movie model is the main model, storing various information of each movie while linking genres, directors, and stars altogether.

- **title (CharField):** Stores the movie's title.
- **release_date (DateField):** Records the movie's release date
- **run_time: (IntegerField):** Records the movie's runtime in minutes
- **rating (Decimal Field):** Stores the movie's rating, allowing up to one decimal point and three maximum digits
- **introduction (TextField):** Records the description of the movie.

This model links to genres, directors, and stars through ManyToMany relationships, allowing a movie to have multiple genres, directors, and stars.

Some fields allow null value and blank input due to the missing data from the dataset. To prevent error produced by the blank inputs, default values will be set during the data loading process via the load_and_store.py.

4. Loading and Storing Data

The loadAndStoreData function is created in load_and_store.py to populate the database with the CSV data.

```python
# Open the CSV file and read the data
with open(csv_path, 'r',encoding='utf-8-sig') as file:
    reader = csv.DictReader(file)
```

The function opens the CSV file in read only mode with UTF-8 encoding to handle special characters and uses csv.DictReader to read the rows as dictionaries.

```
for row in reader:
    # Get the movie data from the CSV file and set the default values if the data is not available
    release_date = row['date'] if row['date'] else None
    run_time = row['run_time'] if row['run_time'] else 0
    rating = row['rating'] if row['rating'] else None
    introduction = row['introduction'] if row['introduction'] else 'No Data'
```

For every movie entry, the function retrieves the data such as title, release date, runtime, rating and introduction. Default values are applied to missing data, such as None for release date, 0 for runtime, None for rating and “No Data” for introduction to prevent errors during API calls.

```
# Create a new movie object
movie, created = Movie.objects.get_or_create(
    # Get the title and remove whitespaces
    title = row['title'].strip(),
    # Set the default values for the movie object
    defaults = {'release_date': release_date, 'run_time': run_time, 'rating': rating, 'introduction': introduction}
)

# Get the genres and split them by comma
genres = row['genre'].split(',')
genre_objects = []
for genre in genres:
    # Get or create the genre object depending on whether it exists or not
    genre_object, _ = Genre.objects.get_or_create(name=genre.strip())
    # Add the genre object to the genre_objects
    genre_objects.append(genre_object)
# Set the genres for the movie
movie.genre.set(genre_objects)

# Get the directors and split them by comma
directors = row['director'].split(',')
director_objects = []
for director in directors:
    # Get or create the director object depending on whether it exists or not
    director_object, _ = Director.objects.get_or_create(name=director.strip())
    # Add the director object to the director_objects
    director_objects.append(director_object)
# Set the directors for the movie
movie.director.set(director_objects)

stars = row['stars'].split(',')
star_objects = []
for star in stars:
    # Get or create the star object depending on whether it exists or not
    star_object, _ = Star.objects.get_or_create(name=star.strip())
    # Add the star object to the star_objects
    star_objects.append(star_object)
# Set the stars for the movie
movie.stars.set(star_objects)

# Save the movie object
movie.save()
```

The function then utilises “get_or_create” method to avoid duplicate entries. A Movie object is either retrieved or created based on its title, with the remaining fields populated with the default values. The genre, director, and star objects are then processed by the similar method. Since a movie can have multiple genres, directors, and stars, these fields are split into lists using split method when encountering commas. A strip method is also called to remove all the whitespaces between each element in the lists.

For each movie, the function retrieves the existing genre, director, or star objects. If the object doesn’t exist in the database, it is created. This approach reduces the database queries, ensuring only unique entries are added, reducing the overall loading time. These objects are

then associated with the Movie object using the set method to handle the many-to-many relationships as per the model.

Error handlings were also implemented to raise an error when a file is not found, or when any exceptions are caught. Finally, a main function will call this loadAndStoreDate function and prints a success message once the data are imported.

5. Serializers

Genre Serializer, Director Serializer, and Star Serializer:

Genre, Director, and Star Serializer handles their respective genre, director and star objects. By having the "__all__" for the fields ensure that all the objects are included in the serialized output.

Movie Serializer:

```
class MovieSerializer(serializers.ModelSerializer):
    genre = GenreSerializer(many=True)
    director = DirectorSerializer(many=True)
    stars = StarSerializer(many=True)
    You, yesterday | 1 author (You)
    class Meta:
        model = Movie
        fields = ['id', 'title', 'release_date', 'run_time', 'rating', 'introduction', 'genre', 'director', 'stars']
```

The Movie Serializer is the main serializer that nests the Genre, Director, and Star Serializer altogether. The serializers have "many" sets to true because a movie can have more than one genre, director, and star. Instead of using __all__ for the fields like the other serializers, the Movie Serializer has field name explicitly defined to make sure the data is printed in an expected order.

When the Movie Serializer is called, it will return the output of all the previously mentioned serializer together with the movie output. This ensures that all the related objects are shown within the serialized output of a movie, allowing users to retrieve all relevant information of a movie in a single API call.

6. URL Routing

URL routing is important to link the user requests to the server logic to retrieve data. This web application uses two URL routing configurations to manage API endpoints.

Main URL Routes

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.renderHome),
    path('movies/', include('api.urls')),
]
```

- **“admin/”:** Paths to the admin site connects the user to Django’s built-in admin site, allowing user to manage models and database entries.
- **“”:** Renders a landing page for the application.
- **“movies/”:** Includes all the REST endpoint routes defined in the API app’s URL configuration.

REST Endpoint URL Routes

```
urlpatterns = [
    path('all/', views.getAllMovies, name='get-all-movies'),
    path('genres/<str:genres>/', views.getMovieByGenre, name='get-movie-by-genres'),
    path('production/', views.getMovieByStarOrDirector, name='get-movie-by-star-or-director'),
    path('search/', views.searchMovies, name='search-movies'),
    path('delete/<int:movie_id>/', views.deleteMovie, name='delete-movie'),
    path('add/', views.addMovie, name='add-movie'),
]
```

- **“all/”:** Fetches all the movies, also allowing user to rearrange the movie order in the query parameter.
- **“genres/<str:genres>/”:** Filters movies based on the provided genres in the URL.
- **“productions/”:** Allows users to filter movies associated by certain stars or directors through query parameter.
- **“search/”:** Allows users to search for movies by the keywords in the movie titles.
- **“delete/<int:movie_id>/”:** Deletes a specific movie based on the ID of the movie specified in the URL.
- **“add/”:** Brings users to the form to input new movie entries into the database.

7. Pagination:

```
class DataPagination(PageNumberPagination):
    page_size = 10
    page_size_query_param = 'page_size'
```

Paginator is implemented using DRF’s build-in paginators, which split the results and display them in smaller sets. This allows the data to be loaded in the page faster and more efficiently. In my case, the pagination is set to display 10 results per page. When the results are returned, the API includes data showing the total number of pages, and links to the next and previous page. Each page will display 10 relevant results, and users can navigate through the results by clicking on the next or previous page links at the top of the JSON object. The paginator is applied to almost all the GET endpoints in *8. REST Endpoints*.

8. REST Endpoints

8.1 Return all movies in a specific order

```python
# GET request to return all movies in certain order
@api_view(['GET'])
def getAllMovies(request):
    # Get the order from the query parameters
    order = request.query_params.get('order')
    movies = Movie.objects.all()

    # Order the movies based on the query parameters
    if order == 'title_asc':
        movies = movies.order_by('title')
    elif order == 'title_desc':
        movies = movies.order_by('-title')
    elif order == 'release_asc':
        movies = movies.order_by('release_date')
    elif order == 'release_desc':
        movies = movies.order_by('-release_date')
    elif order == 'rating_asc':
        movies = movies.order_by('rating')
    elif order == 'rating_desc':
        movies = movies.order_by('-rating')
    elif order == 'runtime_asc':
        movies = movies.order_by('runtime')
    elif order == 'runtime_desc':
        movies = movies.order_by('-runtime')
    elif order == 'id_desc':
        movies = movies.order_by('-id')
    else:
        movies = movies.order_by('id')

    # Return an error if no movies are found
    if not movies.exists():
        return Response({'error': 'No movies found.'}, status=status.HTTP_404_NOT_FOUND)

    # Paginate the movies
    paginator = pagination.DataPagination()
    paginated_movies = paginator.paginate_queryset(movies, request)

    # Serialize the data and return the response
    serializer = MovieSerializer(paginated_movies, many=True)
    return paginator.get_paginated_response(serializer.data)
```

The getAllMovies function handle GET requests from “/all/” to get all movies in the database, sorting them in various orders such as title, release date, rating, runtime, or id. This feature is interesting as it allows users to customize the sorting order using the “?order=<order>” query parameter. The query parameter is processed through the if-else statement, making the result to be returned the way the users want them to be. Additionally, pagination has been implemented to the movie objects to split the results into multiple pages to prevent the browser from overloading, trying to load around 8,900 movies in a page. This endpoint has a medium complexity as it sorts the movies based on the query parameter.

8.2 Return movies of certain genres

```python
# GET request to return movies by genre
@api_view(['GET'])
def getMovieByGenre(request, genres):
    if not genres:
        return Response({'error': 'No genre provided. Please provide a genre.'}, status=status.HTTP_400_BAD_REQUEST)

    # Split the genres by comma and capitalize the first letter of each genre
    genres = [genre.strip().capitalize() for genre in genres.split(',')]

    # Filter the movies by the genres and prevent duplicate movies
    movies = Movie.objects.filter(genre__name__in=genres).distinct()

    # Return an error if no movies are found for the genres
    if not movies.exists():
        return Response({'error': 'No movies found for the provided genres.'}, status=status.HTTP_404_NOT_FOUND)

    # Paginate the movies
    paginator = pagination.DataPagination()
    paginated_movies = paginator.paginate_queryset(movies.order_by('id'), request)

    # Serialize the data and return the response
    serializer = MovieSerializer(paginated_movies, many=True)
    return paginator.get_paginated_response(serializer.data)
```

The getMovieByGenre handles GET requests from the “/genres/<str:genres>/ endpoint, allowing users to get a list of movies filtered by one or more genres. This endpoint is interesting as it supports the input of multiple genres by entering them as “/genres/comedy, horror”. The genres are then split by commas to separate each genre using split method. The strip and capitalize method are then utilised to remove the whitespaces and capitalizing the first letter of each genre. After filtering the movies by the genres, a distinct method is utilised to prevent duplicate entries when a belongs to multiple genres. This has a higher complexity as it filters movies by multiple genres by the many-to-many relationship between movies and genres while ensuring distinct results.

8.3 Return movies with a certain star or director

```python
# GET request to return movies by director or star
@api_view(['GET'])
def getMovieByStarOrDirector(request):
    # Get the star or director from the query parameters
    star = request.query_params.get('star')
    director = request.query_params.get('director')

    # Return an error if no star and director are provided in the query parameters
    if not star and not director:
        return Response({'error': 'Please provide a star or director name.'}, status=status.HTTP_400_BAD_REQUEST)

    movies = Movie.objects.all()

    # Filter the movies by the star
    if star:
        movies = Movie.objects.filter(stars__name__iexact=star)
    # Filter the movies by the director
    if director:
        movies = Movie.objects.filter(director__name__iexact=director)

    # Return an error if no movies are found for the star or director
    if not movies.exists():
        return Response({'error': 'No movies found for the provided star or director.'}, status=status.HTTP_404_NOT_FOUND)

    # Paginate the movies
    paginator = pagination.DataPagination()
    paginated_movies = paginator.paginate_queryset(movies.order_by('id'), request)

    # Serialize the data and return the response
    serializer = MovieSerializer(paginated_movies, many=True)
    return paginator.get_paginated_response(serializer.data)
```

The getMovieByStarOrDirector function handles the GET requests to the “/production/”, allowing users to filter movies by a specific star or director. The query utilises the “__iexact” filter to perform case-insensitive matches, making it flexible for users who didn’t input exact name. This endpoint is interesting as it also supports filtering the movies by both star and director parameters. Error handling is implemented to check whether the parameters are

provided and handle cases where neither star nor director parameters is entered. This endpoint also has a higher complexity as it handles multiple filtering and utilising the many-to-many relationship between movies, stars, and directors.

8.4 Search for movies

```python
# GET request to search for movies by title
@api_view(['GET'])
def searchMovies(request):
    # Get the search query from the query parameters
    query = request.query_params.get('query')

    # Return an error if no search query is provided
    if not query:
        return Response({'error': 'Please provide a search query.'}, status=status.HTTP_400_BAD_REQUEST)

    # Filter the movies by the search query
    movies = Movie.objects.filter(title__icontains=query)

    # Return an error if no movies are found for the search query
    if not movies.exists():
        return Response({'error': 'No movies found for the provided search query.'}, status=status.HTTP_404_NOT_FOUND)

    # Paginate the movies
    paginator = pagination.DataPagination()
    paginated_movies = paginator.paginate_queryset(movies.order_by('id'), request)

    # Serialize the data and return the response
    serializer = MovieSerializer(paginated_movies, many=True)
    return paginator.get_paginated_response(serializer.data)
```

The searchMovie function handles GET requests to the “/search/” endpoint, allowing users to search for movies by keywords in their titles. By utilising the “__icontains” filter, the query performs case-insensitive searches, making it flexible to get appropriate matches. For instance, a search for “Inter” would return “Interstellar”. This enhances the web application’s usability by providing a easier way to find specific movies without knowing the exact titles. This endpoint has a lower complexity, it utilises the “__icontain” filter to provide efficient searching, and pagination has also been applied to manage the performance.

8.5 Delete a movie

```python
# DELETE request to delete a movie
@api_view(['DELETE'])
def deleteMovie(request, movie_id):
    try:
        # Get the movie by the id
        movie = Movie.objects.get(id=movie_id)
        # Delete the movie
        movie.delete()
        return Response("message: Movie deleted successfully", status=status.HTTP_204_NO_CONTENT)
    # Return an error if the movie does not exist
    except Movie.DoesNotExist:
        return Response({'error': 'Movie does not exist.'}, status=status.HTTP_404_NOT_FOUND)
```

The deleteMovie function handles the DELETE requests to the “/delete/<int:movie_id>/” endpoint, allowing users to delete a movie by its id. The query retrieves the movie object based on the specified id and deletes it if it’s found. Error handling is also implemented to ensures that users are informed when they tried to delete a movie that doesn’t exist. This has the lowest complexity out of all endpoints as this endpoint just query the database for the movie that matches the id and delete it.

8.6 Add a movie

```python
# POST request to add a new movie
@api_view(['POST'])
def addMovie(request):
    try:
        data = request.data

        # Process genres
        genre_objects = []
        for genre in data.get('genre', []):
            genre_object, _ = Genre.objects.get_or_create(id=genre['id'], defaults={'name': genre['name']})
            genre_objects.append(genre_object.id)

        # Process directors
        director_objects = []
        for director in data.get('director', []):
            director_object, _ = Director.objects.get_or_create(id=director['id'], defaults={'name': director['name']})
            director_objects.append(director_object.id)

        # Process stars
        star_objects = []
        for star in data.get('stars', []):
            star_object, _ = Star.objects.get_or_create(id=star['id'], defaults={'name': star['name']})
            star_objects.append(star_object.id)

        # Create the movie
        movie = Movie.objects.create(title=data['title'], release_date=data.get('release_date'), run_time=data.get('run_time'), rating=data.get('rating'), introduction=data.get('introduction', ''))
        # Add many-to-many relationships
        movie.genre.set(genre_objects)
        movie.director.set(director_objects)
        movie.stars.set(star_objects)

        # Serialize the created movie
        serializer = MovieSerializer(movie)
        return Response(serializer.data, status=status.HTTP_201_CREATED)

    except Exception as e:
        return Response(status=status.HTTP_400_BAD_REQUEST)
```

The addMovie function handles POST requests to the “/add/” endpoint, allowing users to add new movies to the database. This endpoint is interesting as it handles the many-to-many relationship using “get_or_create” to prevent duplicated entries for all the related objects. After objects are processed, the set method is utilised to establish relationships between the new movie and its genres, directors, and stars. If there is no error, the data is serialized and added to the database. Otherwise, it will throw an exception raising an error status. This endpoint has the highest complexity as it involves creating and retrieving multiple related objects and setting the many-to-many relationship.

9. Unit Testing

Various unit testing was performed to check if the web application is functioning as expected, ensuring the reliability of the application. Tests were implemented to validate the functionality of serializers and RESTful endpoints.

9.1 Serializer Test Case

```python
class SerializersTestCases(APITestCase):
    # Set up the movie data
    def setUp(self):
        self.genre = Genre.objects.create(name='Scifi')
        self.director = Director.objects.create(name='Christopher Nolan')
        self.star = Star.objects.create(name='Matthew McConaughey')
        self.movie = Movie.objects.create(title='Interstellar', release_date='2023-11-26', run_time=169, rating=8.7, introduction='When Earth becomes uninhabitable in the future, a farmer and ex-NASA pilot, Joseph Cooper, i

        # Add the genre, director, and star to the movie
        self.movie.genre.add(self.genre)
        self.movie.director.add(self.director)
        self.movie.stars.add(self.star)

    # Test the Genre serializer
    def testGenreSerializer(self):
        serializer = GenreSerializer(self.genre)
        self.assertEqual(serializer.data, {'id': 1, 'name': 'Scifi'})

    # Test the Director serializer
    def testDirectorSerializer(self):
        serializer = DirectorSerializer(self.director)
        self.assertEqual(serializer.data, {'id': 1, 'name': 'Christopher Nolan'})

    # Test the Star serializer
    def testStarSerializer(self):
        serializer = StarSerializer(self.star)
        self.assertEqual(serializer.data, {'id': 1, 'name': 'Matthew McConaughey'})

    # Test the Movie serializer
    def testMovieSerializer(self):
        data = {
            'id': 1,
            'title': 'Interstellar',
            'release_date': '2023-11-26',
            'run_time': 169,
            'rating': '8.7',
            'introduction': 'When Earth becomes uninhabitable in the future, a farmer and ex-NASA pilot, Joseph Cooper, is tasked to pilot a spacecraft, along with a team of researchers, to find a new planet for humans.',
            'genre': [{'id': 1, 'name': 'Scifi'}],
            'director': [{'id': 1, 'name': 'Christopher Nolan'}],
            'stars': [{'id': 1, 'name': 'Matthew McConaughey'}]
        }
        serializer = MovieSerializer(self.movie)
        self.assertEqual(serializer.data, data)
```

The setUp method creates a temporary dataset with some placeholder data. The testGenreSerializer, testDirectorSerializer, and testStarSerializer methods verify that their respective serializer serialize the objects into the correct dictionary format with their id and name field. The testMovieSerialzer method tests the nested MovieSerializer, which includes all the previous serialized objects. It ensures that all fields and nested fields are correctly serialized into a JSON format.

9.2 Views Test Case

setUp:

```
def setUp(self):
    # Set up the movie data
    self.genre = Genre.objects.create(name='Scifi')
    self.director = Director.objects.create(name='Christopher Nolan')
    self.star = Star.objects.create(name='Matthew McConaughey')
    self.movie = Movie.objects.create(title='Interstellar', release_date='2023-11-26', run_time=169, rating=8.7, introduction='When Earth becomes uninhabitable in the future, a farmer and ex-NASA pilot, Joseph Cooper, is taske

    # Add the genre, director, and star to the movie
    self.movie.genre.add(self.genre)
    self.movie.director.add(self.director)
    self.movie.stars.add(self.star)
```

The setUp method creates a temporary dataset with some placeholder data.

testGetAllMovies:

```
# Test getAllMovies() view function
def testGetAllMovies(self):
    response = self.client.get('/movies/all/')
    self.assertEqual(response.status_code, status.HTTP_200_OK)
    self.assertEqual(len(response.data['results']) > 0, True)
```

This test ensures that “/movies/all” endpoint loads successfully with at least one result.

testGetMoviesByGenre

```
# Test getMovieByGenre() view function
def testGetMovieByGenre(self):
    response = self.client.get('/movies/genres/Scifi/')
    self.assertEqual(response.status_code, status.HTTP_200_OK)
    self.assertEqual(response.data['results'][0]['title'], 'Interstellar')
```

The test verifies that “/movies/genres/<str:genres>/” endpoint filters movies based on their genre. For example, “/movies/genres/scifi/” returns Interstellar that belongs to the SciFi genre.

testGetMovieByStarOrDirector:

```
# Test getMovieByStarOrDirector() view function
def testGetMovieByStarOrDirector(self):
    # Test view function by star
    response = self.client.get('/movies/production/?star=Matthew McConaughey')
    self.assertEqual(response.status_code, status.HTTP_200_OK)
    self.assertEqual(response.data['results'][0]['title'], 'Interstellar')
    # Test view function by director
    response = self.client.get('/movies/production/?director=Christopher Nolan')
    self.assertEqual(response.status_code, status.HTTP_200_OK)
    self.assertEqual(response.data['results'][0]['title'], 'Interstellar')
```

This test ensures the “/movies/production/” endpoint for filtering movies by a specific star or director. Two queries were ran to test the functionality for both query parameters. For instance, querying by the star “Matthew McConaughey” or director “Christopher Nolan” correctly returns the movie “Interstellar”.

testSearchMovies

```python
# Test searchMovies() view function
def testSearchMovies(self):
    response = self.client.get('/movies/search/?query=interstellar')
    self.assertEqual(response.status_code, status.HTTP_200_OK)
    self.assertEqual(response.data['results'][0]['title'], 'Interstellar')
```

This test verifies that “/movies/search/” endpoint returns the correct movie by searching for them with keywords. A query for “Interstellar” returns the movie titled “Interstellar”

testDeleteMovie

```python
# Test deleteMovie() view function
def testDeleteMovie(self):
    response = self.client.delete('/movies/delete/1/')
    self.assertEqual(response.status_code, status.HTTP_204_NO_CONTENT)
    self.assertFalse(Movie.objects.filter(id=self.movie.id).exists())
```

This test ensures the “/movies/delete/<int:movie_id> endpoint deletes the specified movie. After “Interstellar” with the id 1 is deleted, the test confirms the movie no longer exists via the 204 No Content Response.

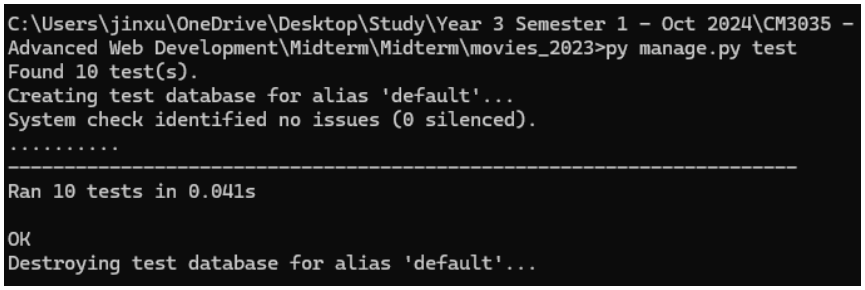
testAddMovie

```python
# Test addMovie() view function
def testAddMovie(self):
    data = {
        'title': 'Top Gun: Maverick',
        'release_date': '2023-11-26',
        'run_time': 131,
        'rating': '8.2',
        'introduction': 'Thirty years of service leads Maverick to train a group of elite TOPGUN graduates to prepare for a high-profile mission while Maverick battles his past demons.',
        'genre': [
            {
                'id': 2,
                'name': 'Action'
            }
        ],
        'director': [
            {
                'id': 2,
                'name': 'Joseph Kosinski'
            }
        ],
        'stars': [
            {
                'id': 2,
                'name': 'Tom Cruise'
            }
        ]
    }

    response = self.client.post('/movies/add/', data, format='json')
    self.assertEqual(response.status_code, status.HTTP_201_CREATED)
    self.assertTrue(Movie.objects.filter(title='Top Gun: Maverick').exists())
    self.assertTrue(Genre.objects.filter(name='Action').exists())
    self.assertTrue(Director.objects.filter(name='Joseph Kosinski').exists())
    self.assertTrue(Star.objects.filter(name='Tom Cruise').exists())
```

This test verifies that the “/movies/add” endpoint adds new movies to the database by sending a POST request with the valid movie data. The test confirms the objects of the new movie exists in the database.

Unit Test Result:



10. Appendix

10.1 Packages and Versions

Packages	Version
Django	5.2.1
Django Rest Framework	3.15.2
Python	3.13.0

10.2 Development Environment

Operating System	Version
Windows 11 Home	24H2

10.3 Instruction for Logging into Django Admin Site

Field	Details
URL	/admin/
Username	admin
Password	password

10.4 Running Unit Tests

Ensure that you are in the same path as manage.py, then run:

python manage.py test

10.5 Location of the Data Loading Script

Data loading script is named load_and_store.py and is stored at the same directory as the manage.py. To run the data loading script, run:

python load_and_store.py