Created by: Bryan Loo Chun Wai

Advanced Web Development

Mid-Term Report

Introduction

This report provides an overview of the design and implementation of a Django-based RESTful web service for managing movies and their associated metadata. The project aligns with the requirements of the coursework brief and leverages the Django Rest Framework (DRF) to deliver CRUD functionalities, serialization, and data validation. Additionally, a bulk data loading script and comprehensive unit tests were developed to ensure application reliability.

Dataset Description

This dataset offers detailed information about a variety of TV shows and movies, including their genres, runtime, certification, ratings, stars, descriptions, votes, and directors. It encompasses a wide spectrum of genres, from action and adventure to drama and comedy, and contains approximately 129,890 entries. This makes it a valuable resource for movie enthusiasts, data analysts, and machine learning practitioners.

The dataset features popular titles such as The Witcher, Mission: Impossible - Dead Reckoning Part One, Sound of Freedom, Secret Invasion, Special Ops: Lioness, and They Cloned Tyrone. These entries provide an exciting range of content for exploration and analysis.

With its rich metadata, users on Kaggle can conduct a variety of analyses, such as examining the relationship between genres and ratings, identifying trends in certification types, and assessing the impact of directors on a movie's success.

Here is an overview of the key columns in the dataset:

- Movie: Lists the names or titles of TV shows and movies, with each row representing a unique title.
- Genre: Indicates the categories or types of content, often including multiple genres separated by commas (e.g., "Action, Adventure, Drama").
- Runtime: Specifies the duration of the content in minutes (e.g., "60 min" or "163 min").
- Certificate: Details the age rating or certification for the content, indicating the appropriate audience (e.g., "A," "UA," or "R").
- Rating: Provides the average viewer or critic score, typically ranging from 1 to 10.
- Stars: Lists the main actors or actresses involved in the production.
- Description: Offers a brief synopsis of the plot or storyline.
- Votes: Shows the total number of user ratings, reflecting audience engagement.
- Director: Names the directors responsible for the production.

In summary, this comprehensive dataset is a treasure trove for exploring various aspects of the entertainment industry. It allows data enthusiasts to investigate topics like genre trends, audience preferences, and the influence of directors and star actors on a title's success. Dive in and unleash your creativity as you explore this rich collection of entertainment data!

Dataset Size

The dataset includes approximately 129,892 records:

Columns include

movie, genre, runtime, certificate, rating, stars, description, votes, director

Data Preprocessing

1) **Loading Data**

- CSV files for movies, genres, stars, and directors were loaded into pandas DataFrames for further processing.
- Initial inspection of datasets was done using .head() to view the first few rows of each DataFrame.

2) **Preprocessing of Movies Dataset**

- Runtime Cleaning:
    - Removed the " min" suffix, commas, and whitespace.
    - Converted runtime values to integers.
    - Invalid or missing values were replaced with a default runtime of 0.
- Votes Cleaning:
    - Removed special characters like commas, dollar signs, and whitespace.
    - Converted values with suffixes (M for million, K for thousand) to integers.
    - Invalid or missing values were replaced with a default vote count of 0.
- Rating Handling:
    - Filled missing values in the rating column with 0.
    - Converted the rating column to a float type.
- Handling Missing Values:
    - Replaced all other missing values with the string "NULL".
- Removing Duplicates:
    - Dropped duplicate rows to ensure data integrity.

3) Preprocessing of Genres Dataset

- Genre Cleaning:
    - Stripped leading and trailing whitespace from genre values.
    - Replaced missing values in the genre column with "NULL".
- Removing Invalid Rows:
    - Dropped duplicate rows.
    - Removed rows where the genre column still contained "NULL" after cleaning.

4) Preprocessing of Stars Dataset

- Stars Cleaning:
    - Stripped leading and trailing whitespace from stars values.
    - Replaced missing values in the stars column with "NULL".
- Removing Invalid Rows:
    - Dropped duplicate rows.
    - Removed rows where the stars column still contained "NULL" after cleaning.

5) Preprocessing of Directors Dataset

- Director Cleaning:
    - Stripped leading and trailing whitespace from director values.
    - Replaced missing values in the director column with "NULL".
- Removing Invalid Rows:
    - Dropped duplicate rows.
    - Removed rows where the director column still contained "NULL" after cleaning.

6) Exporting Cleaned Data

- The cleaned datasets were saved into new CSV files:
    - cleaned_Movies.csv
    - cleaned_Genres.csv
    - cleaned_Stars.csv
    - cleaned_Directors.csv

Summary

This preprocessing pipeline ensured that the data was clean, consistent, and ready for analysis by:

- Standardizing formats (e.g., converting runtime and votes to integers).
- Handling missing values appropriately (filling or removing).
- Removing duplicate and invalid rows.
- Saving the cleaned datasets for further use.

Database Design

The relational database schema includes the following models:

1) **User**

- Fields: username, email, role, is_active, is_admin, etc.
- Role: Determines if a user is an admin or customer.

2) **Movie**

- Fields: movie, runtime, certificate, rating, description, votes.
- Represents the primary entity in the database.

3) **Genre**

- Fields: movie, genre.
- Foreign key relationship to the Movie model.

4) **Star**

- Fields: movie, star_name.
- Foreign key relationship to the Movie model.

5) **Director**

- Fields: movie, director_name.
- Foreign key relationship to the Movie model.

The schema ensures data integrity and supports one-to-many relationships (e.g., one movie can have multiple genres).

ER Diagram

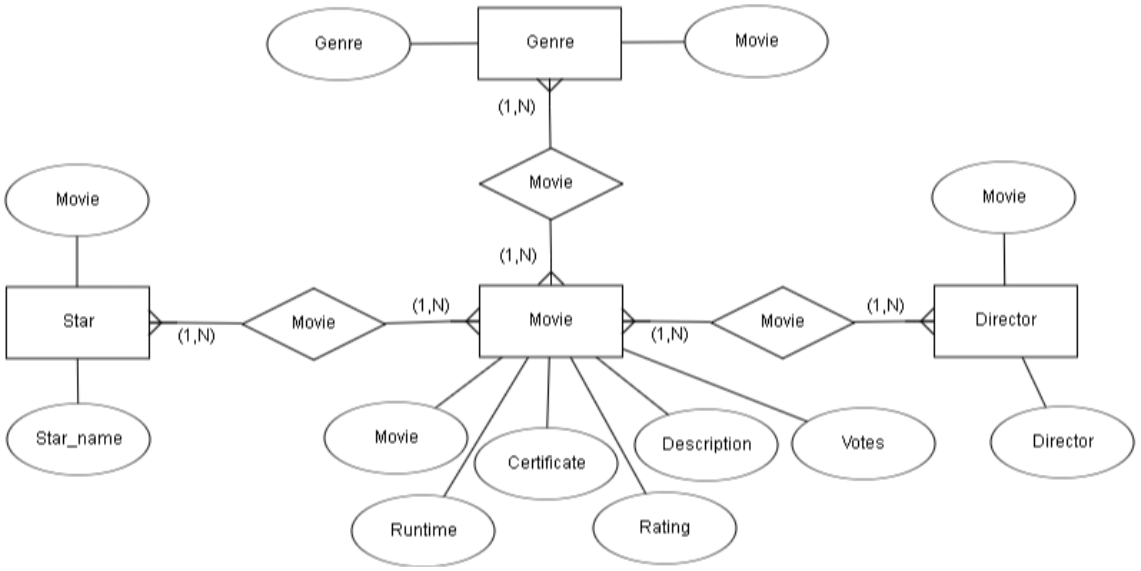

Table schemas

User table schema

Column Name	Data type	Constraints	Default value
Username	VARCHAR(255)	UNIQUE, NOT NULL	
Email	VARCHAR(255)	UNIQUE, NOT NULL	
Age	INT		30
Phone_number	VARCHAR(15)		
Address	TEXT		
Date_of_birth	DATE	NULL	
Gender	VARCHAR(20)		“Prefer_not_to_say”
Role	VARCHAR(10)		“customer”
Is_active	BOOLEAN		TRUE
Is_admin	BOOLEAN		FALSE

Movie table schema

Column name	Data type	Constraints
Movie	VARCHAR(255)	NOT NULL
Runtime	VARCHAR(50)	NOT NULL
Certificate	VARCHAR(50)	NOT NULL
Rating	FLOAT	
Description	TEXT	
Votes	INT	

Genre table schema

Column name	Data type	Constraints	Default value
Movie		FOREIGN KEY	References Movie(id)
Genre	VARCHAR(50)	NOT NULL	

Star table schema

Column name	Data type	Constraints	Default value
Movie		FOREIGN KEY	References Movie(id)
Star_name	VARCHAR(255)	NOT NULL	

Director table schema

Column name	Data type	Constraints	Default value
Movie		FOREIGN KEY	References Movie(id)
Director_name	VARCHAR(255)	NOT NULL	

Normalization

Normalization is a database design process that organizes data to reduce redundancy and improve data integrity. This process involves dividing a database into multiple related tables, ensuring each table adheres to specific normal forms (NF). Below is the step-by-step normalization process applied to the given example.

Step 1: Initial Table

Initial table structure:

- Columns: movie, genre, runtime, certificate, rating, stars, description, votes, director
- Issues:
    - Repeating Groups: A movie may have multiple genres, stars, and possibly directors, causing redundancy and making the table not in 1NF.

Step 2: First Normal Form (1NF)

To achieve 1NF:

1. Eliminate repeating groups: Each cell contains atomic (indivisible) values, and no column has multiple values (e.g., a list of genres or stars).
2. Create unique rows: Ensure each row has a unique identifier (e.g., movie).

Transformed Table in 1NF: Each combination of movie, genre, star, and director is represented in a separate row, leading to significant redundancy.

Step 3: Second Normal Form (2NF)

To achieve 2NF:

1. Ensure 1NF: The table must already be in 1NF.
2. Eliminate partial dependencies: Move columns that depend only on part of the primary key (not the entire composite key) to separate tables.

In the initial table:

- runtime, certificate, rating, description, and votes depend solely on movie (not on genre, star, or director).
- Separate these columns into a Movie table.

Transformed Tables in 2NF:

1. Movie Table: movie, runtime, certificate, rating, description, votes
2. Genre Table: movie, genre
3. Star Table: movie, star
4. Director Table: movie, director

This eliminates partial dependencies by associating attributes with the correct unique identifiers.

Step 4: Third Normal Form (3NF)

To achieve 3NF:

1. Ensure 2NF: The tables must already be in 2NF.
2. Eliminate transitive dependencies: No non-key column should depend on another non-key column.

In the given example:

- No transitive dependencies exist (e.g., certificate does not depend on runtime).

Final 3NF Tables:

1. Movie Table: movie, runtime, certificate, rating, description, votes
2. Genre Table: movie, genre
3. Star Table: movie, star_name
4. Director Table: movie, director_name

Summary of Normalization

1. 1NF: Removed repeating groups; ensured atomicity and unique rows.
2. 2NF: Eliminated partial dependencies by splitting attributes based on their direct dependency on the primary key.
3. 3NF: Removed transitive dependencies to ensure no non-key attribute is dependent on another non-key attribute.

These transformations ensure that the database:

- Minimizes redundancy
- Improves data integrity
- Simplifies data management and querying

RESTful Endpoints

Six endpoints were implemented:

1. Admin Dashboard

- URL: /admin-dashboard/
- Method: GET
- Description: Displays an admin interface for managing movies.

2. Customer Dashboard

- URL: /customer-dashboard/
- Method: GET
- Description: Provides a personalized dashboard for customers to browse and view movies.

3. Top-Rated Movies

- URL: /top-movies/
- Method: GET
- Description: Lists the top 5 movies sorted by rating.

4. Movie List with Genre Filter

- URL: /view-movies/
- Method: GET
- Description: Displays movies filtered by genre, with pagination support.

5. Update Movie

- URL: /update/<int:movie_id>/
- Method: PUT
- Description: Updates movie details.

6. Delete Movie

- URL: /delete/<int:movie_id>/
- Method: DELETE
- Description: Deletes a movie record.

Bulk Data Loading Script

A Python script was created to load data from CSV files into the database. The script:

1. Reads cleaned CSV files for movies, genres, stars, and directors.
2. Creates corresponding database entries using Django ORM.
3. Ensures foreign key relationships are maintained.

The script includes error handling to manage missing or inconsistent data.

Bulk data loading implementation

```
import os
import sys
import django
import csv

# Add your Django project's base directory to the Python path
sys.path.append('C:/AdvancedWebDevelopment Mid Term
Project/midTermProj_AWD/project/')

# Set up Django environment
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'project.settings')
django.setup()
```

```
# Import models from the movies app
from movies.models import Movie, Genre, Star, Director

# File paths for the cleaned CSV files
script_dir = os.path.dirname(os.path.abspath(__file__))
movies_file = os.path.join(script_dir, 'cleaned_Movies.csv')
genres_file = os.path.join(script_dir, 'cleaned_Genres.csv')
stars_file = os.path.join(script_dir, 'cleaned_Stars.csv')
directors_file = os.path.join(script_dir, 'cleaned_Directors.csv')

# Clear existing records from the database
Movie.objects.all().delete()
Genre.objects.all().delete()
Star.objects.all().delete()
Director.objects.all().delete()

# Populate the Movies table
with open(movies_file, 'r', encoding='utf-8') as csv_file:
  csv_reader = csv.DictReader(csv_file)
  movies = {}  # Dictionary to store movies by title
  for row in csv_reader:
    movie = Movie.objects.create(
      movie=row['movie'],
      runtime=row['runtime'],
      certificate=row.get('certificate', ''),  # Handle optional fields
      rating=float(row['rating']),
      description=row.get('description', ''),
      votes=int(row['votes'])
    )
    movies[row['movie']] = movie  # Store the movie instance for reference

# Populate the Genres table
with open(genres_file, 'r', encoding='utf-8') as csv_file:
  csv_reader = csv.DictReader(csv_file)
  for row in csv_reader:
    movie_title = row.get('movie')
    genre_list = row['genre'].split(', ')  # Split genres if multiple
    if movie_title in movies:
      movie = movies[movie_title]
      for genre in genre_list:
        Genre.objects.create(movie=movie, genre=genre)
    else:
      print(f"Movie with title '{movie_title}' not found in Movies table. Skipping genre.")

# Populate the Stars table
with open(stars_file, 'r', encoding='utf-8') as csv_file:
  csv_reader = csv.DictReader(csv_file)
  for row in csv_reader:
    movie_title = row.get('movie')
    star_name = row.get('stars')
    if movie_title in movies:
```

```
        movie = movies[movie_title]
        Star.objects.create(movie=movie, star_name=star_name)
      else:
        print(f"Movie with title '{movie_title}' not found in Movies table. Skipping star.")

# Populate the Directors table
with open(directors_file, 'r', encoding='utf-8') as csv_file:
  csv_reader = csv.DictReader(csv_file)
  for row in csv_reader:
    movie_title = row.get('movie')
    director_name = row.get('director')
    if movie_title in movies:
      movie = movies[movie_title]
      Director.objects.create(movie=movie, director_name=director_name)
    else:
      print(f"Movie with title '{movie_title}' not found in Movies table. Skipping director.")

# Print completion message
print("Database population complete.")
```

Testing

Comprehensive unit tests were written for:

1. Models: Ensuring data integrity and relationship constraints.
2. Endpoints: Validating responses for valid and invalid inputs.
3. Data Loading Script: Verifying successful insertion of data into the database.

Tests implementation

```
from django.test import TestCase, Client
from django.urls import reverse
from movies.models import Movie, Genre, Star, Director
from django.contrib.auth.models import User

class UserTests(TestCase):

  def setUp(self):
    # Create test users
    self.admin_user = User.objects.create_user(
      username='adminTest',
      email='adminTest@example.com',
      password='Pass',
      is_staff=True
    )
    self.customer_user = User.objects.create_user(
      username='customerTest',
      email='customerTest@example.com',
```

```
            password='Pass'
        )
        self.client = Client()

    def test_admin_dashboard_access(self):
        self.client.login(username='adminTest', password='Pass')
        response = self.client.get(reverse('admin_dashboard'))
        self.assertEqual(response.status_code, 200)

    def test_customer_dashboard_access(self):
        self.client.login(username='customerTest', password='Pass')
        response = self.client.get(reverse('customer_dashboard'))
        self.assertEqual(response.status_code, 200)

    def test_unauthenticated_access(self):
        response = self.client.get(reverse('admin_dashboard'))
        self.assertEqual(response.status_code, 302)
        self.assertRedirects(response, '/login/')

class MovieTests(TestCase):

    def setUp(self):
        # Create sample movie and related entities
        self.movie = Movie.objects.create(
            movie="Test Movie",
            runtime="120",
            certificate="PG-13",
            rating=8.5,
            description="A test movie description.",
            votes=10000
        )
        self.genre = Genre.objects.create(movie=self.movie, genre="Action")
        self.star = Star.objects.create(movie=self.movie, star_name="Test Star")
        self.director = Director.objects.create(movie=self.movie, director_name="Test Director")
        self.client = Client()

    def test_movie_detail_view(self):
        response = self.client.get(reverse('movie_detail', args=[self.movie.id]))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "Test Movie")
        self.assertContains(response, "Action")
        self.assertContains(response, "Test Star")
        self.assertContains(response, "Test Director")

    def test_movie_list_view(self):
        response = self.client.get(reverse('movie_list'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "Test Movie")

    def test_create_movie_view(self):
        self.client.login(username='adminTest', password='Pass')
```

```
        response = self.client.post(reverse('create_movie'), {
            'movie': "New Movie",
            'runtime': "150",
            'certificate': "R",
            'rating': 7.0,
            'description': "A new test movie description.",
            'votes': 5000
        })
        self.assertEqual(response.status_code, 302)
        self.assertTrue(Movie.objects.filter(movie="New Movie").exists())

    def test_update_movie_view(self):
        self.client.login(username='adminTest', password='Pass')
        response = self.client.post(reverse('update_movie', args=[self.movie.id]), {
            'movie': "Updated Movie",
            'runtime': "140",
            'certificate': "PG",
            'rating': 9.0,
            'description': "An updated test movie description.",
            'votes': 15000
        })
        self.assertEqual(response.status_code, 302)
        updated_movie = Movie.objects.get(id=self.movie.id)
        self.assertEqual(updated_movie.movie, "Updated Movie")

    def test_delete_movie_view(self):
        self.client.login(username='adminTest', password='Pass')
        response = self.client.post(reverse('delete_movie', args=[self.movie.id]))
        self.assertEqual(response.status_code, 302)
        self.assertFalse(Movie.objects.filter(id=self.movie.id).exists())

class AuthenticationTests(TestCase):

    def setUp(self):
        self.user = User.objects.create_user(
            username='testuser123',
            email='testuser123@example.com',
            password='Pass'
        )
        self.client = Client()

    def test_registration_view(self):
        response = self.client.post(reverse('register'), {
            'username': "newuser111",
            'email': "newuser111@example.com",
            'password': "Pass",
            'confirm_password': "Pass"
        })
        self.assertEqual(response.status_code, 302)
        self.assertTrue(User.objects.filter(username="newuser111").exists())
```

```
  def test_login_view(self):
    response = self.client.post(reverse('login'), {
      'username': "testuser123",
      'password': "Pass"
    })
    self.assertEqual(response.status_code, 302)
    self.assertEqual(response.url, reverse('customer_dashboard'))

  def test_logout_view(self):
    self.client.login(username='testuser123', password='Pass')
    response = self.client.get(reverse('logout'))
    self.assertEqual(response.status_code, 302)
    self.assertRedirects(response, '/login/')
```

Sample test cases include:

- Verifying correct serialization of movie data.
- Testing filters for genre-based movie listings.
- Ensuring proper handling of invalid update or delete requests.

Deployment

The application was deployed locally and tested using SQLite3 as the database backend. Future deployment options include hosting on AWS or Heroku for external accessibility.

Challenges and Solutions

1. Handling Large Datasets:
    - Used efficient batch processing during bulk data loading.
2. Endpoint Design:
    - Ensured clarity and consistency by adhering to RESTful conventions.
3. Testing:
    - Leveraged Django's testing framework to automate test execution.

Conclusion

This project successfully implements a Django-based RESTful web service for managing movies. The system adheres to the coursework requirements by offering robust CRUD operations, efficient data loading, and comprehensive testing. Future improvements include deploying the application on a cloud platform and adding advanced features like user-specific recommendations.

References

TV & Movie Metadata with Genres and Ratings (2023) -

https://www.kaggle.com/datasets/gayu14/tv-and-movie-metadata-with-genres-and-ratings-imbd

To load data:

Python populate_movies.py

To Run the code

Python manage.py runserver