**Movie Collection Web Application - Requirements Document**

**Table of Contents**

1. Introduction

2. Functional Requirements

3. Non-Functional Requirements

4. API Specifications

    - User Registration and Authentication

    - Movie Retrieval

    - Collection Management

5. Data Models

6. Testing Requirements

7. Technology Stack

8. Assumptions

1. **Introduction**

The Movie Collection web application allows users to register, authenticate, create and manage their movie collections. Users can create, view, update, and delete collections, as well as add movies to these collections. The application will also fetch a list of available movies from a third-party API and provide statistics based on the user's collections.

1. **Functional Requirements**

User Registration and Authentication

1. Users must be able to register using a username and password.

2. Upon successful registration, users will receive a JWT token.

3. Users must authenticate using the JWT token for all actions except registration.

4. Only authenticated users can create, view, update, and delete collections.

Movie Retrieval

1. The application will fetch a list of movies from a third-party API.

2. The movies API should return paginated results.

Collection Management

1. Users must be able to create collections by providing a title, description, and a list of movies.

2. Users must be able to update existing collections and movies in it.

3. Users must be able to delete collections.

4. Users must be able to view their collections and the movies within them.

5. The response for viewing collections should include the top 3 favourite genres based on the movies in the user's collections.

1. **Non-Functional Requirements**

1. The application must be secure and protect user data.

2. The system must handle concurrent requests efficiently.

3. The API should be RESTful and follow standard conventions.

4. The application should log all significant actions and errors for auditing and debugging purposes.

5. The system should be scalable to handle a growing number of users and collections.

1. **API Specifications**

User Registration and Authentication

*Register User*:

URL: `POST /register/`

Request Payload:

{

"username": "<desired username>",

"password": "<desired password>"

}

Response:

{

"access\_token": "<JWT token>"

}

Movie Retrieval

*Get Movies*

URL:`GET /movies/`

Response:

{

"count": <total number of movies>,

"next": <link for next page, if present>,

"previous": <link for previous page>,

"data": [

{

"title": "<title of the movie>",

"description": "<a description of the movie>",

"genres": "<a comma-separated list of genres>",

"uuid": "<unique uuid for the movie>"

},

...

]

}

**Collection Management**

***Get All Collections***

URL: `GET /collection/`

Response:

{

"is\_success": true,

"data": {

"collections": [

{

"title": "<Title of my collection>",

"uuid": "<uuid of the collection name>",

"description": "<Description of the collection>"

},

...

],

"favourite\_genres": "<Top 3 favorite genres based on the movies in collections>"

}

}

***Create Collection***

URL:`POST /collection/`

Request Payload:

{

"title": "<Title of the collection>",

"description": "<Description of the collection>",

"movies": [

{

"title": "<title of the movie>",

"description": "<description of the movie>",

"genres": "<genres>",

"uuid": "<uuid>"

},

...

]

}

Response:

{

"collection\_uuid": "<uuid of the collection item>"

}

***Update Collection***

URL:`PUT /collection/<collection\_uuid>/`

Request Payload:

{

"title": "<Optional updated title>",

"description": "<Optional updated description>",

"movies": "<Optional movie list to be updated>"

}

Response:

{

"message": "Collection updated successfully"

}

***Get Collection Details***

URL: `GET /collection/<collection\_uuid>/`

Response:

{

"title": "<Title of the collection>",

"description": "<Description of the collection>",

"movies": [

{

"title": "<title of the movie>",

"description": "<description of the movie>",

"genres": "<genres>",

"uuid": "<uuid>"

},

...

]

}

***Delete Collection***

URL: `DELETE /collection/<collection\_uuid>/`

Response:

{

"message": "Collection deleted successfully"

}

1. **Data Models**

User:

Default User table (auth\_user)

Movie

Fields:

- `title` (String)

- `description` (Text)

- `genres` (String)

- `uuid` (UUID)

Collection

Fields:

- `title` (String)

- `description` (Text)

- `user` (ForeignKey to User)

- `movies` (ManyToMany to Movie)

- `uuid` (UUID)

1. **Testing Requirements**

1. Unit Tests: Ensure all individual components work as expected.

2. Integration Tests: Ensure that the components work together as expected.

3. Functional Tests: Test the application from the end-user perspective.

4. Load Tests: Ensure the application can handle concurrent requests efficiently.

Sample Test Cases

- Test user registration and authentication.

- Test creating, updating, viewing, and deleting collections (CRUD).

- Test fetching movies from the third-party API.

- Test the response structure for all endpoints.

- Test the request count accuracy for concurrent request using threading.

1. **Technology Stack**

- Backend: Django, Django REST Framework

- Authentication: JWT

- Database: SQLite (for development), PostgreSQL (for production)

- Testing: Django Test Framework, Factory Boy

- HTTP Client: `requests` library

1. **Assumptions**

- The third-party movie API is unreliable (Added retry mechanisms).

- Users will not register with the same username (If so, will show an error).

- The system will scale to handle an increasing number of users and collections.

1. **Future Enhancements**

* Use postrgres database for production server
* Run more test cases with improper and injective data
* Can write all the logs data to a log file for audit or issue solving

**Note: -**

I’ve also attached the post collection for testing purpose in the repository