Movies API project

Table of Contents

1. Overview

The goal of this project is to create a REST API to manage movies. The API must be built with Python, and (recommended) any framework of your choice. The project must include a database to store the movies and reviews. The API must be able to:

- Create, update, delete, and get movies
- Get a list of movies
- Search for movies by keywords in the title or description
- For the list and search endpoints: filter by genre, date, and rating.

2. Guidelines for implementation

- **Language**: The API must be built with Python. You can use any framework of your choice (example: Flask, Django, FastAPI).
- **Database**: The project must include a database to store the movies and reviews. You can use any database of your choice (example: SQLite, PostgreSQL, MongoDB), the easiest to use is SQLite.
- **Environment**: The project must include a requirements.txt file listing the required packages. The project must be runnable in a virtual environment. The project must run in development mode, using the development server of the framework.
- **Git**: The project must be submitted as a git repository on Github, Gitlab, or any other platform.
- **Documentation**: The API must be documented using Swagger or any other API documentation tool. The project must include a README file that explains how to run the project.
- **Tests**: The project must include unit tests. The tests must be runnable with a single command (explained in README).

3. Detailed steps

3.1. Create a virtual environment

Create a virtual environment and list the installed packages in a requirements.txt file. The project must be runnable in this virtual environment. Explain in the README how to create the virtual environment and run the project.

3.2. Set up the API project

Set up the API project with the framework of your choice. The project must be runnable in development mode, using the development server of the framework. Explain in the README how to run the project.

The API must be documented in Swagger/OpenAPI format. The docs page must be accessible from the API root URL (example: http://localhost:5000/) or from any other page that you will specify in the README (example: http://localhost:5000/docs).

3.3. Create the database

Create a database to store the movies and reviews. You can use any database of your choice (example: SQLite, PostgreSQL, MongoDB), the easiest to use is SQLite. Explain in the README how to create the database and run the project.

3.4. Implement the API (CRUD endpoints)

A movie object can be represented by a JSON object with the following fields:

```
{
  "id": 1,
  "title": "La Grande Vadrouille",
  "description": "During World War II, two French civilians and a downed English Bomber Crew set
out from Paris to cross the demarcation line between Nazi-occupied Northern France and the
South. From there they will be able to escape to England. First, they must avoid German troops -
and the consequences of their own blunders.",
  "genres": [
    "Comedy",
    "War"
  ],
  "release_date": "1966-12-07",
  "vote_average": 7.7,
  "vote_count": 1123
}
```

All input and output data must be in JSON format.

Implement the following endpoints:

```
GET /movies/<int:id>
```

```
    Return the movie with the given id or a 404 error if not found.
```

```
POST /movies
    Create a new movie. The body of the request must be a JSON object representing the movie to
create. The id field must not be set in the request body. Return the created movie with its id,
or a 400 error with an explicit error message if the request body is invalid.
```

```
DELETE /movies/<int:id>
    Delete the movie with the given id. Return a 204 status code with an empty body.
    Return a 404 error if the movie is not found.
```

```
PUT /movies/<int:id>
    Update the movie with the given id. The body of the request must be a JSON object
representing the movie to update. The id field must not be set in the request body. Return the
updated movie, or a 400 error with an explicit error message if the request body is invalid.
    Return a 404 error if the movie is not found.
```

3.5. Fill the database with movies

Fill the database with at least 1000 movies. You can use this dataset: https://www.kaggle.com/rounakbanik/the-movies-dataset (file `movies_metadata.csv` contains all the necessary fields).

Explain in the README how to fill the database from the dataset. You can commit the dataset file in the git repository, and any script used to fill the database.

3.6. Implement the list and search endpoints

Implement the following endpoints:

```
GET /movies/
    Return a list of movies. The response must be a JSON object with the following fields:
    * `count`: the total number of movies returned from the request
    * `movies`: an array of movies (see the movie object above)

    If no filter is provided (see below), the endpoint must return all the movies in the
database.

    You are not expected to implement pagination, but you can do it as a bonus. In this case,
the response must include the following fields:
    * `page`: the current page number
    * `total_pages`: the total number of pages
    * `next_page`: the URL of the next page, or null if the current page is the last one
    * `previous_page`: the URL of the previous page, or null if the current page is the first
one
```

```
GET /movies/search/<term>
```

```
    Return a list of movies matching the given search term. The response must be a JSON object
with the following fields:
    * `count`: the total number of movies returned from the request
    * `movies`: an array of movies (see the movie object above)

    You're expected to implement the search against a single word in the title or description of
the movie (so `term` is a single word, not a sentence).
    The search must be case-insensitive and must match the exact search term, found in the title
or description of the movie.

    You are not expected to implement pagination, but you can do it as a bonus, like for the
`/movies/` endpoint.
```

3.7. Implement the filters

Implement the following filters for the `/movies/` and `/movies/search/<term>` endpoints:

- `genre`: filter by genre. The value of the filter is a single genre (example: `/movies/?genre=Comedy`). The filter must return movies that have at least one of the given genres.
- `before` and `after`: filter by date. The value of the filters are dates in the format `YYYY-MM-DD` (example: `/movies/?before=2000-01-01&after=1990-01-01`). The filter must return movies that have been released before/after the given date. The two filters can be used alone (example: `/movies/?before=2000-01-01`) or together (example: `/movies/?before=2000-01-01&after=1990-01-01`).
- `vote_average`: filter by vote average. The value of the filter is a float number between 0 and 10 (example: `/movies/?vote_average=7.2`). The filter must return movies that have a vote average greater than or equal to the given value.

The filters can be combined together (example: `/movies/?genre=Comedy&after=2000-01-01&vote_average=8`).

3.8. Tests

Write unit tests for the API endpoints. The tests must be runnable with a single command (explained in README).

For each endpoint, write at least one test for each of the following cases: * the endpoint returns the expected response * the endpoint returns the expected status code * the endpoint returns the expected response when the request is invalid (example: invalid JSON body, invalid query parameters, etc.) * the endpoint returns the expected response when the requested resource is not found (example: movie with the given id not found)

3.9. Bonus

- Implement pagination for the `/movies/` and `/movies/search/<term>` endpoints.
- In the search endpoint, implement the search against multiple words in the title or description of the movie (so `term` is a sentence, not a single word). Sort the results by relevance (the number of words matched in the title or description).
- In the search endpoint, allow inexact search. For example, if the search term is `spider`, the search must return movies with the title or description containing `spider`, `spiders`, `spiderman`, `spider-man`, etc. Typos can be considered or not, it's up to you (example: `spyder` can match `spider` or not).