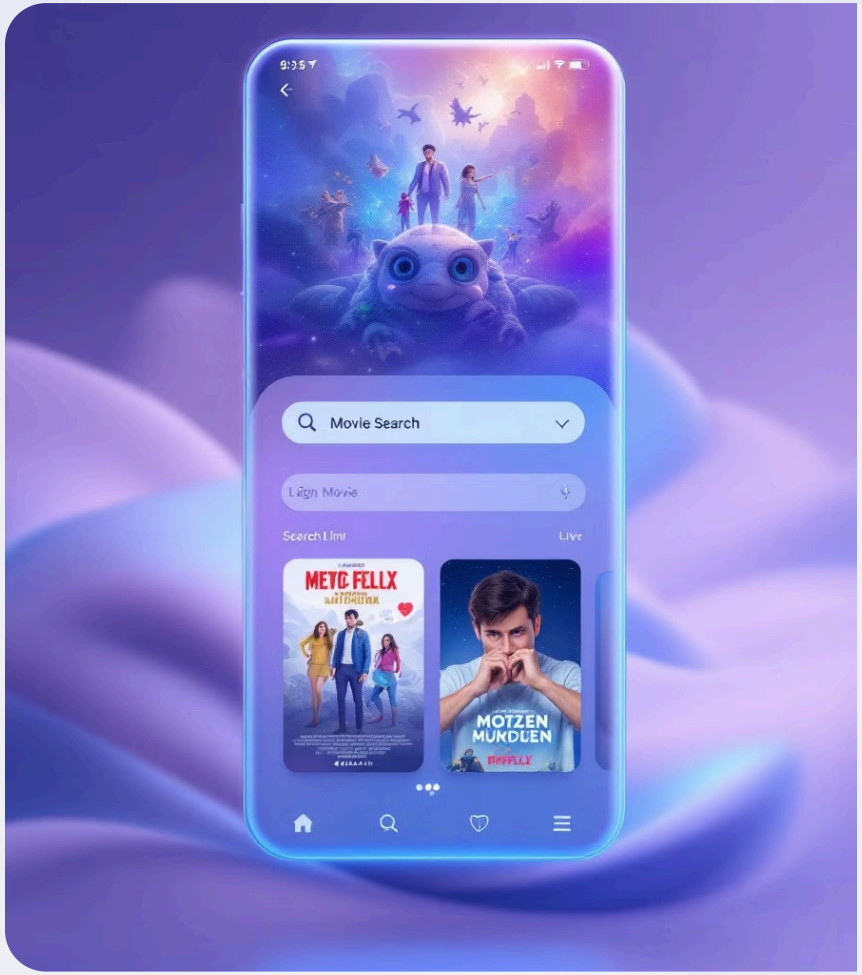


Report on the Movie Streaming API Implementation

This report explores the features implemented, the technical challenges faced, and the reasons why certain planned features were not realized within the project timeline.

by Ebad Salehi

Introduction

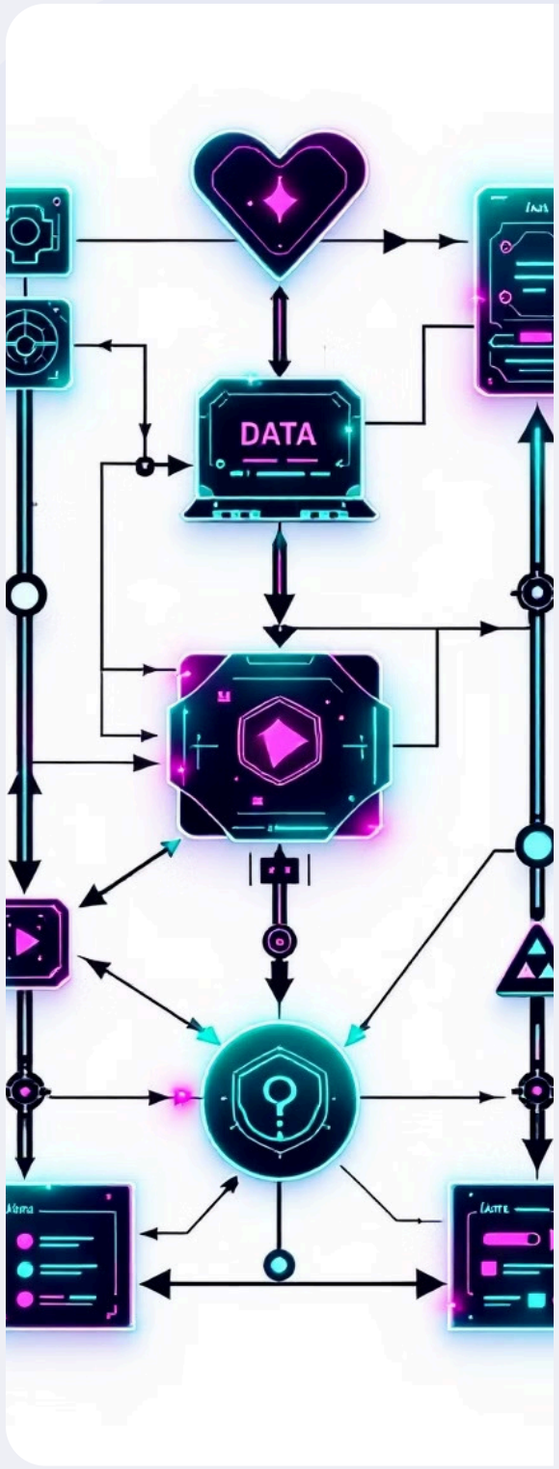
The Movie Streaming API is a Node.js-based application designed to provide comprehensive movie-related data by integrating the **Streaming Availability API** and the **OMDB API**. This project showcases the ability to handle movie searches, fetch detailed movie information, and manage movie posters without relying on external frameworks like Express.js. Instead, the API leverages native Node.js modules, emphasizing minimalism and control over the HTTP server operations.

Implemented Features

- Comprehensive endpoint functionality, including searching for movies, retrieving movie details, and managing poster images.
- Robust error handling to enhance API reliability and user experience.
- Input validation to ensure the integrity of user-provided data.

Limitations

- Caching frequently requested API responses for improved efficiency was not implemented due to time constraints.
- Advanced features like pagination for search results and support for more complex queries were deferred to future iterations.



Technical Description of the Application

Architecture Overview

The Movie Streaming API adheres to a modular design to simplify development and maintenance. The project structure is as follows:

server.js

Serves as the entry point for the application, initializing the server and delegating requests to appropriate handlers.

movieService.js

Contains the core logic for communicating with external APIs, parsing responses, and assembling the required data.

.env

Centralized storage of sensitive information like API keys and configuration parameters for easy management and enhanced security.

posters/

A dedicated directory for managing locally stored poster images, with dynamic creation and cleanup mechanisms.

Implemented Features

1

Movie Search by Title

The `/movies/search/{title}` endpoint fetches movies matching the given title. It processes user input to construct API calls, ensuring valid requests and delivering accurate responses.

2

Detailed Movie Data by IMDb ID

The `/movies/data/{imdbId}` endpoint provides comprehensive information about a specific movie, including its synopsis, release year, and available streaming platforms among much more details.

3

Poster Retrieval and Management

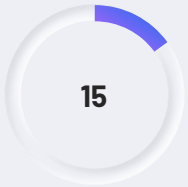
- *Fetching Posters:* The `/posters/{imdbId}` endpoint retrieves a movie's poster. If unavailable locally, it fetches the image from the API and caches it for future requests.
- Uploading Posters: The `/posters/add/{imdbId}` endpoint allows users to upload custom posters. Images are stored locally and linked to their corresponding IMDb ID by their filename.

4

Error Handling

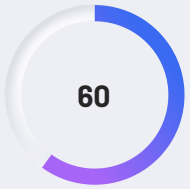
- Invalid input is addressed with meaningful error messages.
- API communication errors are managed with retries and fallbacks to maintain functionality.
- Detailed feedback is provided for unsupported operations, ensuring transparency to the user.

Technical Challenges



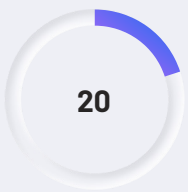
Native Node.js Limitations

- Routing and request parsing were manually implemented since external libraries like Express.js were disallowed. This increased development complexity but provided a deeper understanding of HTTP fundamentals.
- Parsing JSON payloads and handling query strings required custom logic, adding to the implementation effort.



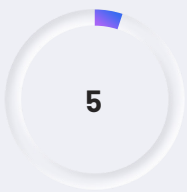
API Communication

- Inconsistent Data: The Streaming Availability API occasionally returned varied data structures, necessitating additional validation layers to ensure compatibility with the OMDB API.
- Rate Limiting: External APIs imposed strict request quotas, prompting careful usage and error-handling mechanisms to avoid service interruptions.



Poster Management

- Handling multipart form-data uploads using native Node.js modules proved challenging. Custom parsers were written to process file uploads and save them locally.
- Ensuring the consistency of poster filenames and preventing overwrites added another layer of complexity.



Security Considerations

- Sensitive data like API keys were managed through `.env` files to prevent accidental exposure.
- Input validation routines were implemented to guard against potential vulnerabilities such as injection attacks.

Future Improvements

Caching Mechanism

Implementing a robust caching layer can significantly reduce redundant API calls to external services like OMDB and Streaming Availability. This will not only improve response times for users but also alleviate the load on these external APIs, potentially reducing associated costs.

Pagination

Adding pagination support, especially for search results and large datasets, is crucial for enhancing the user experience. This prevents overwhelming users with large volumes of data and improves the performance of the application, particularly on slower connections.

User Authentication

Introducing token-based authentication is essential for controlling access to sensitive API endpoints. This will allow for the implementation of user-specific features like personalized watchlists and recommendations while also enhancing the overall security of the application.

Monitoring and Logging

Deploying a comprehensive logging system to capture request details, API response times, and potential errors is vital for ongoing maintenance and performance optimization. This data can provide insights into user behaviour, identify bottlenecks, and help debug issues effectively.

Appendix 1: Installation Guide

1

Clone the Repository

`git clone https://github.com/Ebad-S/Movie_Streaming_API.git`

2

Install Dependencies

`npm install`

3

Configure Environment Variables

Create a `.env` file by renaming `SAMPLE.env` in the root directory and include the following:

`OMDB_API_KEY=your_omdb_key`

`STREAMING_API_KEY=your_rapidapi_key`

`STREAMING_API_HOST=streaming-availability.p.rapidapi.com`

`PORT=3000`

4

Start the Server

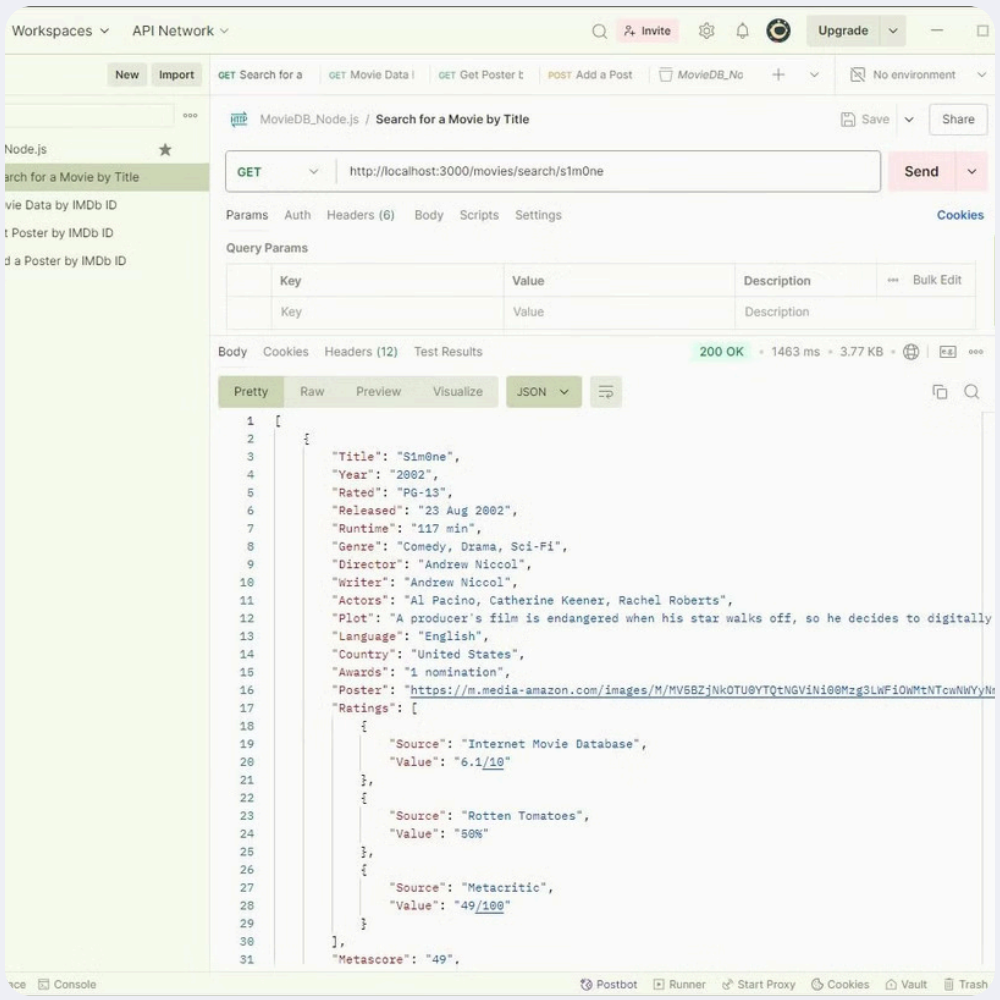
`node server.js`

5

Test API Endpoints

Use **Postman** or a similar tool to send requests to `http://localhost:3000.`

Appendix 2: Example API Requests



Search for a Movie by Title

Method: GET

URL:

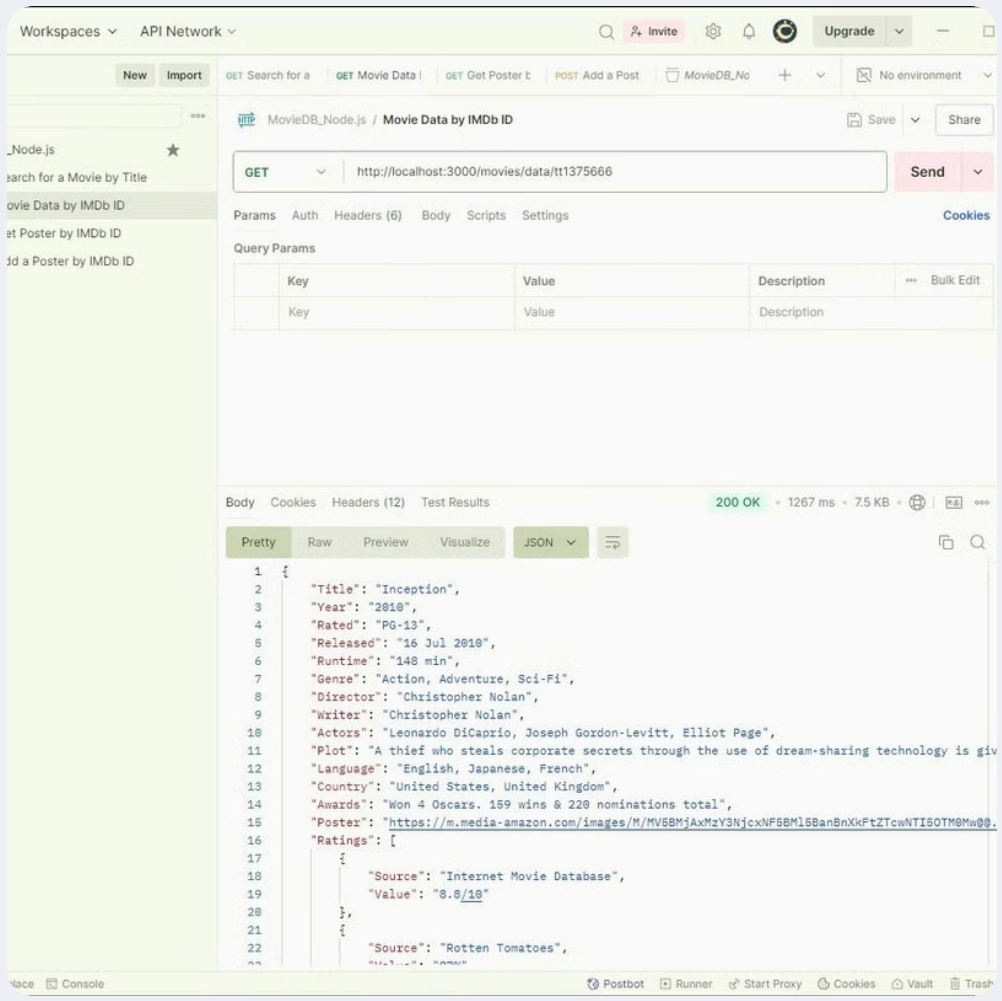
http://localhost:3000/movies/search/{movietitle}

Example:

http://localhost:3000/movies/search/s1m0ne

Response:

JSON object containing movie search results.



Get Movie Data by IMDb ID

Method: GET

URL:

http://localhost:3000/movies/data/{IMDB_ID}

Example:

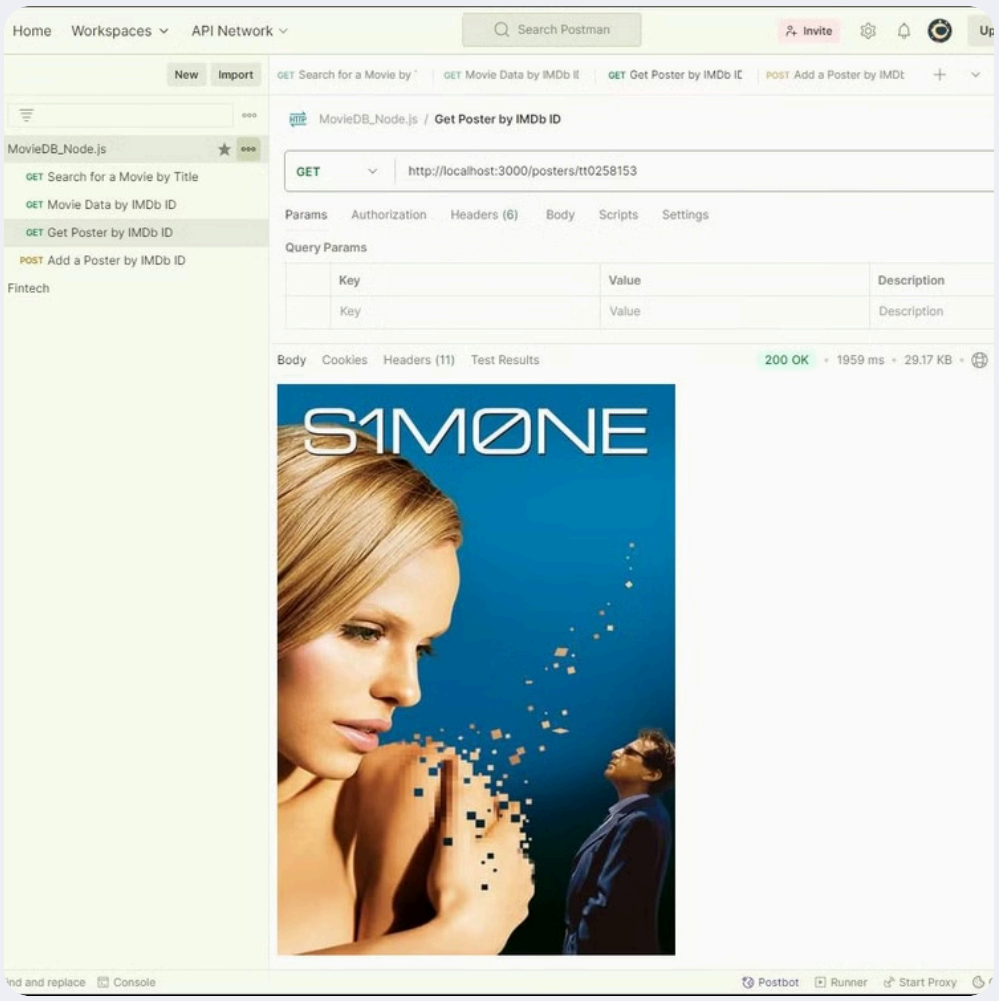
http://localhost:3000/movies/data/tt1375666

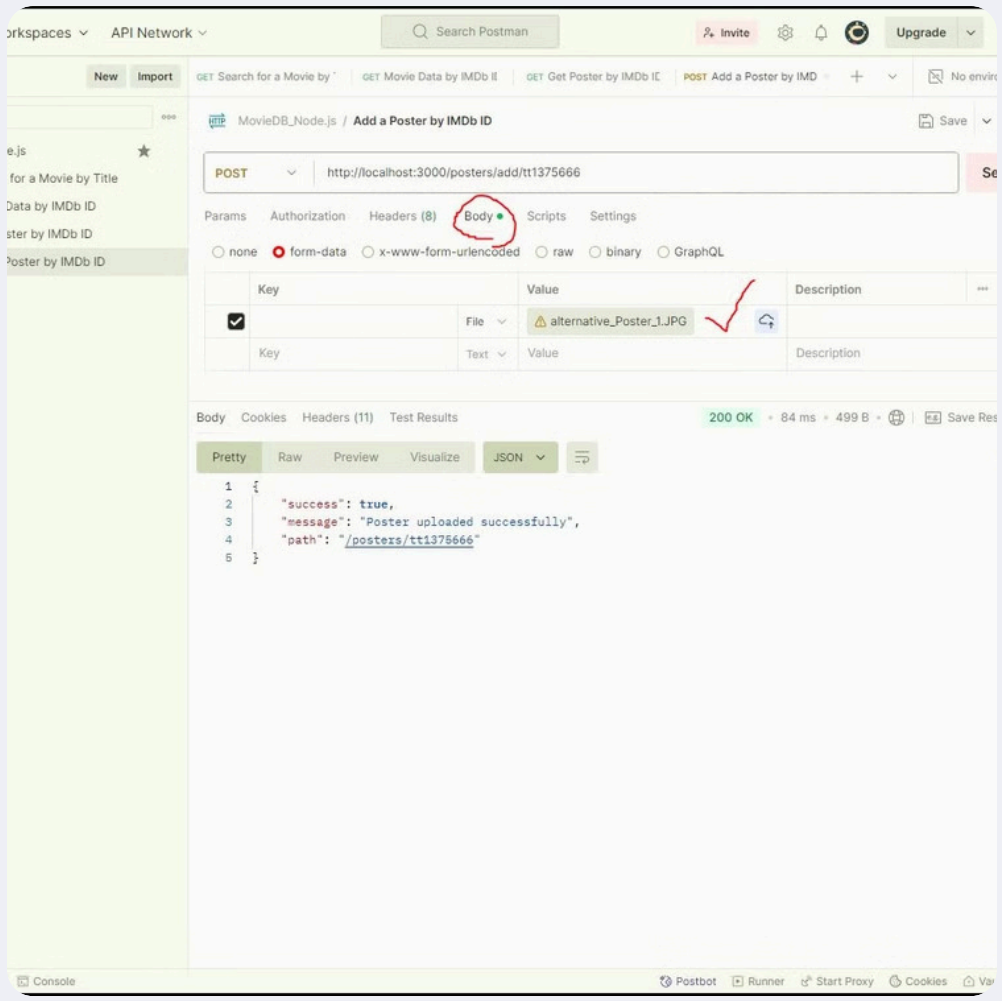
Response:

JSON object with detailed movie information.

```
< {< MiinAy(VP=net)}
    < cuncuss(=echunge) >
{: rauvs, sfor cmk;
    < Miri lemsot,;
```

Appendix 2: Example API Requests





Retrieve a Poster by IMDb ID

Method: GET

URL:

http://localhost:3000/posters/{IMDB_ID}

Example:

http://localhost:3000/posters/tt0258153

Response:

JPEG image of the movie poster.

Add an Alternative Poster by IMDB ID

Method: POST

URL:

http://localhost:3000/posters/add/{IMDB_ID}

How to:

in the Body section use Form-data containing the image file.

References

- Dotenv. (n.d.). Dotenv: Load environment variables from .env files. Version 16.4.5. Retrieved November 14, 2024, from **npmjs.org**
- Streaming Availability. (n.d.). Streaming Availability: API client for fetching streaming options. Version 4.4.0. Retrieved November 14, 2024, from **npmjs.org**
- OMDb API. (n.d.). OMDb API - The Open Movie Database. **https://www.omdbapi.com/**
- OpenAI. (2024). ChatGPT language model. Retrieved from **openai.com/chatgpt**
- All featured images used in this report were generated by ai.