Université de Strasbourg

UFAZ FRENCH-AZERBAIJANI UNIVERSITY

M1 DSAI

Distributed Algorithms

Project Work Report

Movie Cataloging Website Application: A RESTful Node.js Implementation

15 may 2025

Group 3 team members:

- ➢ Toghrul Mammadli
- ➢ Yusif Hajizada
- ➢ Mustafa Ahmadov

Introduction

The general objective of this project is to create an application using a REST architecture. Our domain of interest is Movies, consequently, we implemented RESTful architecture to develop Movie Cataloging Website Application using Node.js.

The main purpose of our web application is to simplify users, clients, to identify and relate different movies through actors, directors and genres in these movies, categorising them.

So, for implementing this web application project, our team reviewed all the theoretical and practical concepts described during the whole course digging in more depth of the concepts in course and at the end of reviewing pre development part we come to conclusion to use further technologies to implement this RESTful project: Nodejs, Swagger. PostgreSQL. Railway. Jest. Sequelize ORM and so on.

Overall source code of the project. implementation of abovelisted technologies codes and other details discussed furthermore in report - we shared in our github page [1].

Implementation

2.1 Brief description of the chosen technologies

Here are the main technologies implemented in these project work:

- **Node.js -** a JavaScript runtime environment for server-side development
- **Express.js -** a lightweight framework built on Node.js that simplifies web application and API development.
- **Swagger -** a set of open-source tools and specifications for designing, building, documenting, and consuming RESTful APIs.
- **PostgreSQL -** a powerful, open-source relational database system known for its reliability and extensibility.
- **Railway -** a cloud platform that simplifies deploying and managing web applications and databases.
- **Jest -** a JavaScript testing framework focused on simplicity and developer experience, commonly used for unit and integration testing.
- **Sequelize ORM -** a promise-based Node.js Object-Relational Mapper (ORM) that makes it easier to interact with SQL databases like PostgreSQL

2.2 Definition of endpoints

In this project, we used the following endpoints, methods to fetch a list of movies with support for pagination, version 1 and version 2, filtering by genre and director, sorting, searching by title/overview/year, and optionally including related actors and genres [2][3]:

Movies		
API endpoints for managing movies		
Methods	**Endpoints**	**Description**
GET	/api/v2/movies	Retrieve all movies
GET	/api/v2/movies/{id}	Get a movie by ID
POST	/api/v2/movies	Create a new movie
PUT	/api/v2/movies/{id}	Update a movie by ID
DELETE	/api/v2/movies/{id}	Delete a movie by ID

Genres		
API endpoints for managing genres		
Methods	**Endpoints**	**Description**

GET	/api/v2/genres	Retrieve all genres
POST	/api/v2/genres	Create a new genre
GET	/api/v2/genres/{id}	Get a genre by ID
PUT	/api/v2/genres/{id}	Update a genre by ID
DELETE	/api/v2/genres/{id}	Delete a genre by ID
GET	/api/v2/genres/{id}/movies	Get all movies for a specific genre

<table>
<tr><th colspan="3">Actors</th></tr>
<tr><td colspan="3">API endpoints for managing actors</td></tr>
<tr><th>Methods</th><th>Endpoints</th><th>Description</th></tr>
<tr><td>GET</td><td>/api/v2/actors</td><td>Retrieve all actors with pagination</td></tr>
<tr><td>POST</td><td>/api/v2/actors</td><td>Create a new actor</td></tr>
<tr><td>GET</td><td>/api/v2/actors/{id}</td><td>Get a actor by ID</td></tr>
<tr><td>PUT</td><td>/api/v2/actors/{id}</td><td>Update a actor by ID</td></tr>
<tr><td>DELETE</td><td>/api/v2/actors/{id}</td><td>Delete a actor by ID</td></tr>
<tr><td>GET</td><td>/api/v2/actors/{id}/movies</td><td>Get all movies acted for a specific actor</td></tr>
</table>

<table>
<tr><th colspan="3">Directors</th></tr>
<tr><td colspan="3">API endpoints for managing directors</td></tr>
<tr><th>Methods</th><th>Endpoints</th><th>Description</th></tr>
<tr><td>GET</td><td>/api/v2/directors</td><td>Retrieve all directors</td></tr>
<tr><td>POST</td><td>/api/v2/directors</td><td>Create a new director</td></tr>
<tr><td>GET</td><td>/api/v2/directors/{id}</td><td>Get a director by ID</td></tr>
<tr><td>PUT</td><td>/api/v2/directors/{id}</td><td>Update a director by ID</td></tr>
<tr><td>DELETE</td><td>/api/v2/directors/{id}</td><td>Delete a director by ID</td></tr>
<tr><td>GET</td><td>/api/v2/directors/{id}/movies</td><td>Get all movies directed for a specific director</td></tr>
</table>

2.3 Design decisions

We have designed our system with scalability and maintainability as top priorities, leveraging a clean and modular architecture. The codebase is well-organized into separate, purpose-specific directories, making it easy to manage and extend, We followed a clean **MVC-style** separation:

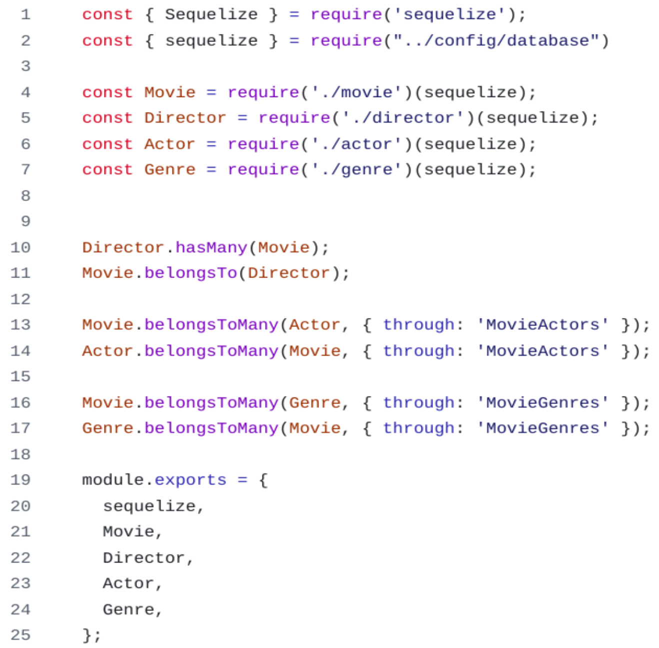
- models/ — contains all data models, sequelize definitions of database entities.
- controllers/ — handles business logic
- services/ — All business logic encapsulated here, e.g., filtering, sorting, and pagination logic
- middlewares/ — includes all custom middleware (e.g., for logging)
- routes/ — defines API routes
- tests/ — includes comprehensive test coverage using Jest

This separation of concerns ensures that each component of the application is isolated, reusable, and easy to scale. Adding new features, modifying existing logic, or onboarding new developers becomes significantly easier with this structure.

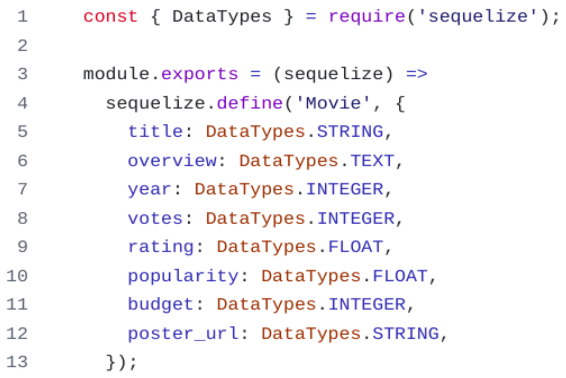
In addition, we have implemented API versioning, V1, V2, with each version residing in its own directory. This allows us to support pair versions concurrently while keeping the codebase clean and organized. We also use a custom logging middleware for clear, structured logging, which improves traceability and debugging.

For our database, we chose PostgreSQL because our data has strong relational requirements (e.g., films and their actors). PostgreSQL provides robust support for relational data, allowing us to model complex relationships accurately and efficiently. So we implemented relation design for the interconnectedness of movies with actors, genres, and directors, we modeled many-to-many and one-to-many relationships explicitly using Sequelize associations and through tables. that is the reason why we used PostgreSQL instead of MongoDB

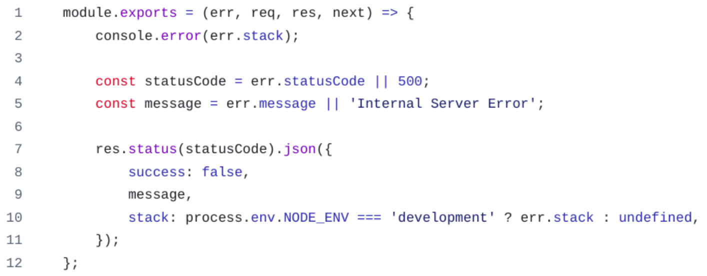
Models Indexing, Movie Model and Error Handler [4][5][6]:



```
const { Sequelize } = require('sequelize');
const { sequelize } = require("../config/database")

const Movie = require('./movie')(sequelize);
const Director = require('./director')(sequelize);
const Actor = require('./actor')(sequelize);
const Genre = require('./genre')(sequelize);


Director.hasMany(Movie);
Movie.belongsTo(Director);

Movie.belongsToMany(Actor, { through: 'MovieActors' });
Actor.belongsToMany(Movie, { through: 'MovieActors' });

Movie.belongsToMany(Genre, { through: 'MovieGenres' });
Genre.belongsToMany(Movie, { through: 'MovieGenres' });

module.exports = {
  sequelize,
  Movie,
  Director,
  Actor,
  Genre,
};
```



```
const { DataTypes } = require('sequelize');

module.exports = (sequelize) =>
  sequelize.define('Movie', {
    title: DataTypes.STRING,
    overview: DataTypes.TEXT,
    year: DataTypes.INTEGER,
    votes: DataTypes.INTEGER,
    rating: DataTypes.FLOAT,
    popularity: DataTypes.FLOAT,
    budget: DataTypes.INTEGER,
    poster_url: DataTypes.STRING,
  });
```

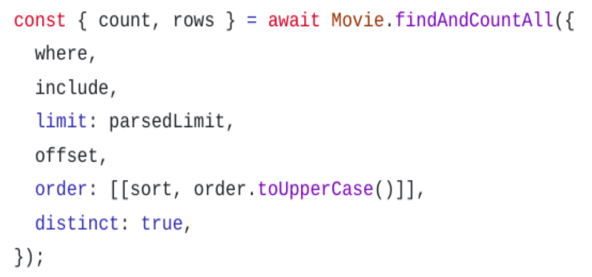


```
module.exports = (err, req, res, next) => {
    console.error(err.stack);

    const statusCode = err.statusCode || 500;
    const message = err.message || 'Internal Server Error';

    res.status(statusCode).json({
        success: false,
        message,
        stack: process.env.NODE_ENV === 'development' ? err.stack : undefined,
    });
};
```

2.4 Implemented methods

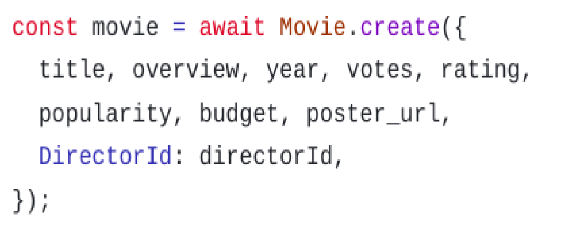
In this project, we used and wrote a lot of methods, here are most common and main of them [7];

Method to Get All Movies and to Create a Movie



```
const { count, rows } = await Movie.findAndCountAll({
  where,
  include,
  limit: parsedLimit,
  offset,
  order: [[sort, order.toUpperCase()]],
  distinct: true,
});
```

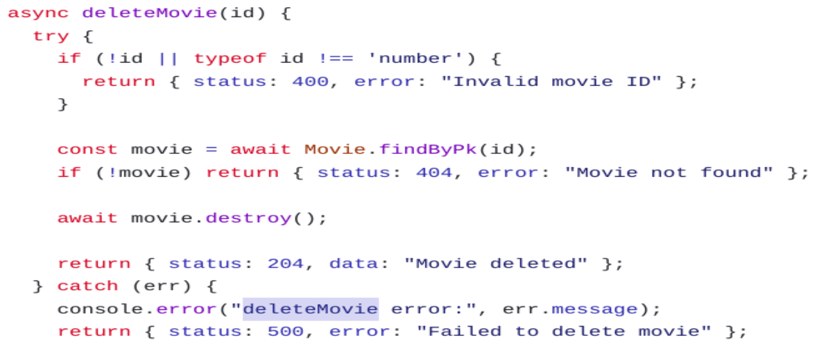
The distinct: true ensures correct count with include. Include dynamically adapts to actor/genre filters. Prevents SQL injection through Sequelize parameter binding.



```
const movie = await Movie.create({
  title, overview, year, votes, rating,
  popularity, budget, poster_url,
  DirectorId: directorId,
});
```

Efficiently uses Sequelize association methods to handle M:N relationships.Adds robustness by rejecting invalid data early.

Delete Movie

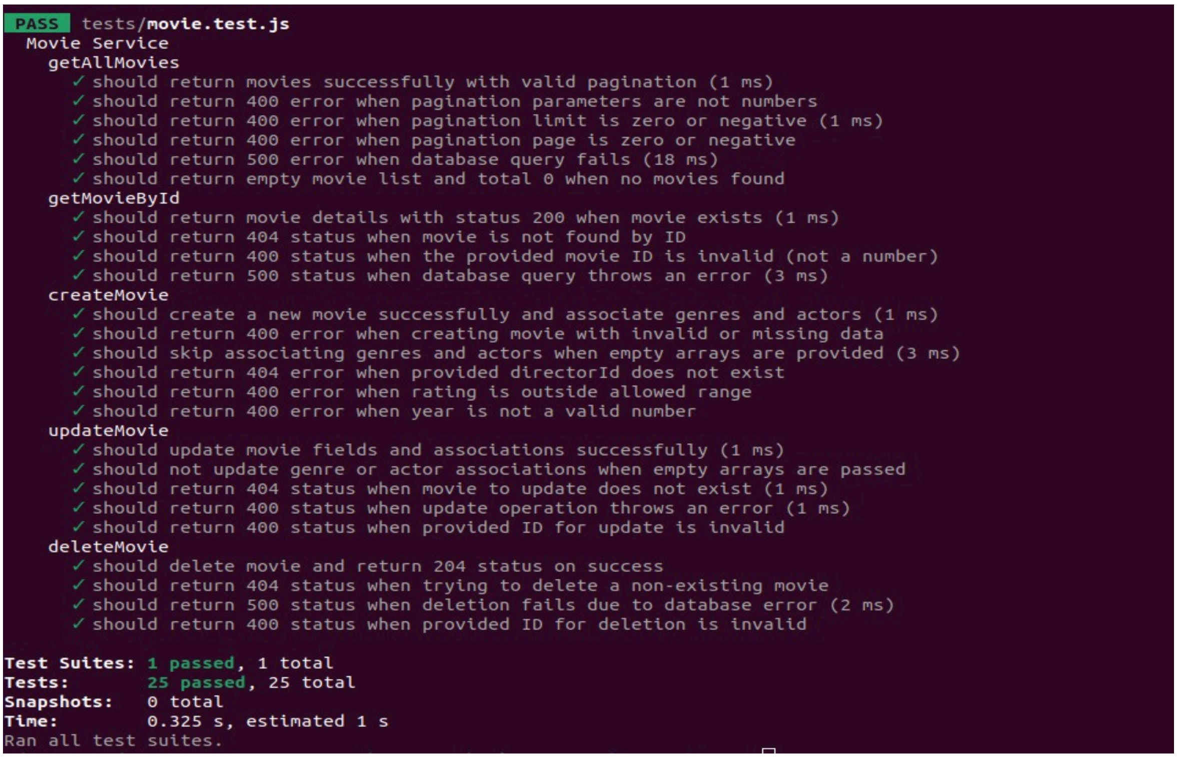


```
async deleteMovie(id) {
  try {
    if (!id || typeof id !== 'number') {
      return { status: 400, error: "Invalid movie ID" };
    }

    const movie = await Movie.findByPk(id);
    if (!movie) return { status: 404, error: "Movie not found" };

    await movie.destroy();

    return { status: 204, data: "Movie deleted" };
  } catch (err) {
    console.error("deleteMovie error:", err.message);
    return { status: 500, error: "Failed to delete movie" };
```

Ensures clean removal from database with proper HTTP status codes.

Test

Here below are the tests implemented in the main part of the project: the movie section [8].



```
PASS  tests/movie.test.js
  Movie Service
    getAllMovies
      ✓ should return movies successfully with valid pagination (1 ms)
      ✓ should return 400 error when pagination parameters are not numbers
      ✓ should return 400 error when pagination limit is zero or negative (1 ms)
      ✓ should return 400 error when pagination page is zero or negative
      ✓ should return 500 error when database query fails (18 ms)
      ✓ should return empty movie list and total 0 when no movies found
    getMovieById
      ✓ should return movie details with status 200 when movie exists (1 ms)
      ✓ should return 404 status when movie is not found by ID
      ✓ should return 400 status when the provided movie ID is invalid (not a number)
      ✓ should return 500 status when database query throws an error (3 ms)
    createMovie
      ✓ should create a new movie successfully and associate genres and actors (1 ms)
      ✓ should return 400 error when creating movie with invalid or missing data
      ✓ should skip associating genres and actors when empty arrays are provided (3 ms)
      ✓ should return 404 error when provided directorId does not exist
      ✓ should return 400 error when rating is outside allowed range
      ✓ should return 400 error when year is not a valid number
    updateMovie
      ✓ should update movie fields and associations successfully (1 ms)
      ✓ should not update genre or actor associations when empty arrays are passed
      ✓ should return 404 status when movie to update does not exist (1 ms)
      ✓ should return 400 status when update operation throws an error (1 ms)
      ✓ should return 400 status when provided ID for update is invalid
    deleteMovie
      ✓ should delete movie and return 204 status on success
      ✓ should return 404 status when trying to delete a non-existing movie
      ✓ should return 500 status when deletion fails due to database error (2 ms)
      ✓ should return 400 status when provided ID for deletion is invalid

Test Suites: 1 passed, 1 total
Tests:       25 passed, 25 total
Snapshots:   0 total
Time:        0.325 s, estimated 1 s
Ran all test suites.
```

Conclusion

In conclusion, this project allowed us to apply the concepts of distributed programming by developing a scalable and well-structured RESTful movie cataloging application using Node.js, Express, PostgreSQL, and Sequelize. We designed a modular architecture with clear separation of concerns, enabling easy maintenance and extensibility. By implementing features such as filtering, sorting, pagination, and relational mapping between movies, actors, directors, and genres, we demonstrated a practical understanding of REST principles and relational database design. Swagger documentation and Jest testing further ensured code reliability and clarity. Overall, this project enhanced our technical skills and provided hands-on experience in building a full-stack distributed system.

References

1. https://github.com/TogrulMemmedli/distalgoproject/tree/main
2. https://distalgoproject.up.railway.app/api-docs/v1/
3. https://distalgoproject.up.railway.app/api-docs/v2/
4. https://github.com/TogrulMemmedli/distalgoproject/blob/main/models/index.js
5. https://github.com/TogrulMemmedli/distalgoproject/blob/main/models/movie.js
6. https://github.com/TogrulMemmedli/distalgoproject/blob/main/middlewares/errorHandler.js
7. https://github.com/TogrulMemmedli/distalgoproject/blob/main/services/v2/movieService.js
8. https://github.com/TogrulMemmedli/distalgoproject/blob/main/tests/movie.test.js