API Development and Testing Report

University of
South Wales
Prifysgol
De Cymru

Student Name: Ibtisam Rafiq

Student ID: 30135822

Module Title: Mobile Application Development

Module Code: CS4S763

Module Tutor: Alun King

Assessment Ttile: Develop and test a REST API

Contents

List of Figures

1 Introduction

The Movie API project aims to provide a structured and efficient system for managing movie-related data, including movies, directors, genres, and actors. The theme was chosen due to the widespread use of APIs in modern software development, particularly in media applications such as streaming services, film databases, and entertainment platforms. The development of this API allows seamless integration of movie-related data, enabling functionalities such as retrieving movie information, adding actors and directors, and managing genre classifications. The API follows RESTful principles, ensuring scalability and ease of use for various applications.

2 Technical Details

The Movie API is built using the following technology stack:

- **Backend Framework:** Node.js with Express.js for handling HTTP requests.
- **Authentication:** Uses JWT-based authentication to store user credentials using bcrypt for hashing, and tokens are issued upon successful login.
- **Database:** MongoDB, a NoSQL database, is used for storing movie-related data.
- **ORM:** Mongoose is utilized for schema validation and data management.
- **Testing Framework:** Jest and Supertest are used to ensure the reliability and correctness of the API.
- **Version Control:** GitHub, a version control platform, used to store the complete code.
- **Postman:** A platform for developers to design, build, test, and collaborate on APIs.

2.1 Project Structure

The project follows a modular structure, ensuring maintainability and scalability:

- **config/** - Contains functionality that connects the API to database and also stores JWT token and its expiry time.

- **middleware/** - Contains functions such as authentication and request validation, ensuring security and data integrity by processing requests before they reach controllers.
- **routes/** - Contains API route definitions for movies, directors, genres, actors, and movie-actor relations.
- **models/** - Defines Mongoose models for database schema representation.
- **controllers/** - Implements business logic for handling API requests.
- **tests/** - Includes Jest test cases to validate API endpoints.
- **server.js** - The main entry point to initialize the Express server and connect to the database.

2.2 Environment Configuration

Environment variables are used to store sensitive information:

- `MONGO_URI` - Stores the MongoDB connection string.
- `PORT` - Defines the port on which the server runs.
- `TEST_PORT` - Defines the test port on which the server performs testing.

2.3 API Framework and Middleware

The API uses middleware to handle requests efficiently:

- **express.json()** - Parses incoming JSON requests.
- **authMiddleware.js** - Checks for authentication of the user trying to access the application.
- **Mongoose Middleware** - Used for validation and pre-processing of database entries.

2.4 Use of Test Database

Used mongodb-memory-server to create an in-memory MongoDB database that resets after each test.

2.5 Entities

The following are the entities that are relevant to the project:

- **Movie**
- **Director**
- **Actor**
- **Genre**
- **MovieActor**

2.6 ER Diagram

ER Diagram to represent all the entities and how they are related to each other in the project is:

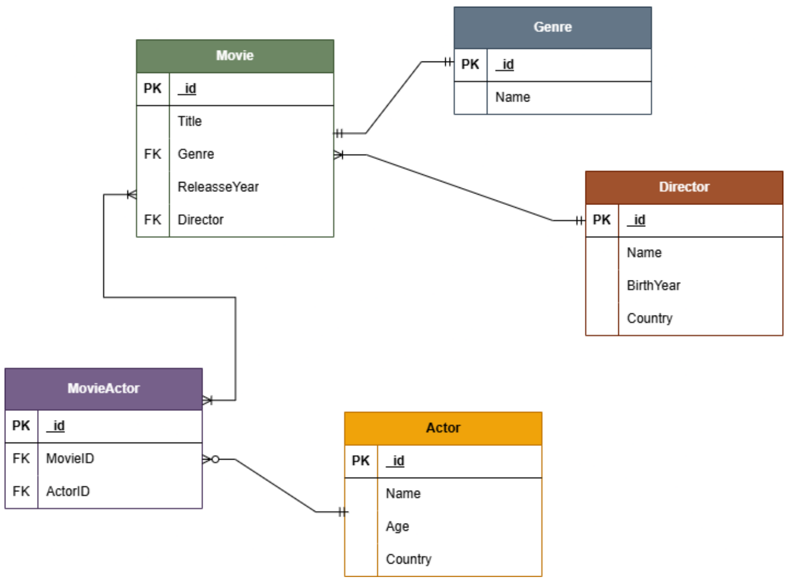


Figure 1: ER Diagram

2.7 Architecture Diagram

This structure ensures that the API remains modular, scalable, and easy to extend in the future. Architecture diagram for this project is:

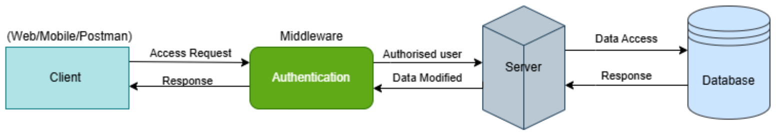


Figure 2: System Architecture of the API

2.8 Project Timeline

Table 1: Project Timeline

Days	**Task**	**Description**
1	Theme Selection	Choose a suitable theme for the API (e.g., Movie Database).
3	Database Design	Define entities, relationships, and create an ER diagram.
3	API Development	Implement CRUD operations and authentication.
2	Testing	Write unit tests for API endpoints.
3	Debugging	Fix errors, improve response handling, and optimize queries.
2	Documentation	Write API documentation and prepare the final report.

3 Justification of Theme Choice

The theme for this API was chosen to address the need for efficient management of movie-related data, including movies, genres, directors, and actors. This API provides a structured approach to retrieving, storing, and updating movie information, making it useful for applications such as movie databases, streaming services, and content management platforms.

4 Overview of API Endpoints

The API exposes several endpoints to facilitate interaction with movie-related data. Below is a brief overview:

4.1 Authentication

- **POST /api/register** - Registers a new user with valid username and password
- **POST /api/login** - Generates a token if valid credentials are entered

4.2 Directors

- **GET /api/directors** - Retrieve all directors
- **POST /api/directors** - Add a new director
- **GET /api/directors/{id}** - Retrieve a specific director
- **PUT /api/directors/{id}** - Update a director
- **DELETE /api/directors/{id}** - Remove a director

4.3 Genres

- **GET /api/genres** - Retrieve all genres
- **POST /api/genres** - Add a new genre
- **GET /api/genres/{id}** - Retrieve a specific genre
- **PUT /api/genres/{id}** - Update a genre
- **DELETE /api/genres/{id}** - Remove a genre

4.4 Movies

- **GET /api/movies** - Retrieve all movies
- **POST /api/movies** - Add a new movie
- **GET /api/movies/{id}** - Retrieve a specific movie
- **PUT /api/movies/{id}** - Update a movie
- **DELETE /api/movies/{id}** - Remove a movie

4.5 Actors

- **GET /api/actors** - Retrieve all actors
- **POST /api/actors** - Add a new actor
- **GET /api/actors/{id}** - Retrieve a specific actor
- **PUT /api/actors/{id}** - Update a actor
- **DELETE /api/actors/{id}** - Remove a actor

4.6 MovieActors

- **GET /api/movieActors** - Retrieve all movie actors
- **POST /api/movieActors** - Add a new movie actor
- **GET /api/movieActors/{id}** - Retrieve a specific movie actor
- **DELETE /api/movieActors/{id}** - Remove a movie actor

5 Overview of Tests

The API has been tested using Jest to ensure robustness and reliability. The testing suite includes:

5.1 Movie Tests

- Adding a movie with valid data
- Attempting to add a movie with missing required fields
- Fetching movies and validating response structure

5.2 Director Tests

- Adding a new director
- Adding a director with missing required fields
- Fetch all directors

5.3 Genre Tests

- Adding a new genre
- Preventing duplicate genre additions

5.4 Actor Tests

- Adding a new actor
- Fetching all actors
- Trying to add a duplicate error

5.5 Movie-Actors Relationship Tests

- Fetching all movie-actor relationships
- Ensuring duplicate assignments are prevented
- Trying to add a relationship with incomplete fields
- Adding same actor to same movie twice

6 Link to Code Repository

The complete source code for the API and test suite is available at:

GitHub Repository Link

7 Code

7.1 Schema

Listing 1: Database Schema

```
const mongoose = require('mongoose');

const movieSchema = new mongoose.Schema({
    title: { type: String, required: true },
    genre: { type: String, required: true },
    releaseYear: { type: Number, required: true }
});

module.exports = mongoose.model('Movie', movieSchema);
```

7.2 Controller Functions

Listing 2: CRUD Operations

```
const Movie = require('../models/Movie');

// Create a movie
exports.createMovie = async (req, res) => {
    try {
        const movie = new Movie(req.body);
        await movie.save();
        res.status(201).json(movie);
    } catch (error) {
        res.status(400).json({ error: error.message });
    }
};

// Read all movies
exports.getMovies = async (req, res) => {
    const movies = await Movie.find();
    res.json(movies);
};

// Update a movie
exports.updateMovie = async (req, res) => {
    try {
        const movie = await Movie.findByIdAndUpdate(req.params.id,
            req.body, { new: true });
        res.json(movie);
    } catch (error) {
        res.status(400).json({ error: error.message });
    }
};

// Delete a movie
exports.deleteMovie = async (req, res) => {
    await Movie.findByIdAndDelete(req.params.id);
    res.status(204).send();
};
```

7.3 Routes

Listing 3: API Routes

```
const express = require("express");
const { createMovie, getMovies, getMovieById, updateMovie,
   deleteMovie } = require("../controllers/movieController");
```

```
const authMiddleware = require("../middleware/authMiddleware");

const router = express.Router();

router.post("/", authMiddleware, createMovie);
router.get("/", authMiddleware, getMovies);
router.get("/:id", authMiddleware, getMovieById);
router.put("/:id", authMiddleware, updateMovie);
router.delete("/:id", authMiddleware, deleteMovie);

module.exports = router;
```

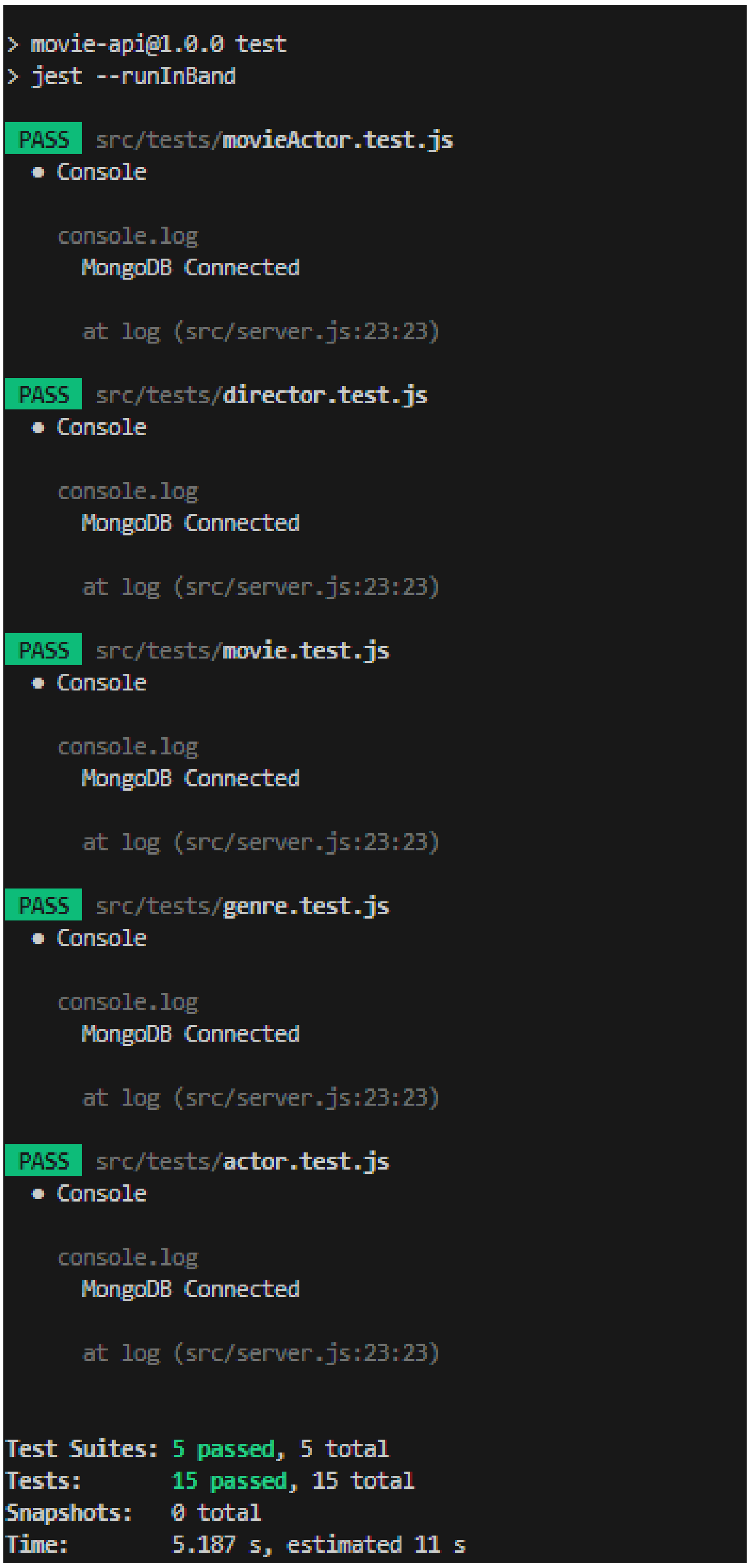
7.4 Tests

Listing 4: Movie Tests

```
const request = require(’supertest’);
const app = require(’../app’);

describe(’Movies␣API’, () => {
    it(’should␣create␣a␣new␣movie’, async () => {
        const res = await request(app).post(’/api/movies’).send({
            title: "Inception",
            genre: "Sci-Fi",
            releaseYear: 2010
        });

        expect(res.statusCode).toBe(201);
        expect(res.body.title).toBe("Inception");
    });

    it(’should␣retrieve␣all␣movies’, async () => {
        const res = await request(app).get(’/api/movies’);
        expect(res.statusCode).toBe(200);
    });

    it(’should␣update␣a␣movie’, async () => {
        const movieId = "60c72b2f5f1b2c001c8e4d5e";
        const res = await request(app).put(‘/api/movies/${movieId
           }‘).send({
            title: "Interstellar"
        });

        expect(res.statusCode).toBe(200);
        expect(res.body.title).toBe("Interstellar");
    });

    it(’should␣delete␣a␣movie’, async () => {
        const movieId = "60c72b2f5f1b2c001c8e4d5e";
        const res = await request(app).delete(`/api/movies/${movieId}`);
        expect(res.statusCode).toBe(204);
    });
});
```

Listing 5: Actors Tests

```
describe("Actors␣API", () => {
    it("should␣fetch␣all␣actors␣(empty␣at␣first)", async () => {
        const res = await request(app).get("/api/actors");
        expect(res.statusCode).toBe(200);
        expect(res.body).toEqual([]);
    });

    it("should␣fail␣to␣add␣an␣actor␣without␣required␣fields", async () => {
        const res = await request(app).post("/api/actors").send({});

        expect(res.statusCode).toBe(400);
        expect(res.body.error).toContain("Actor␣validation␣failed");
    });

    it("should␣not␣add␣a␣duplicate␣actor", async () => {
        await request(app).post("/api/actors").send({
            name: "Leonardo␣DiCaprio",
            birthdate: "1974-11-11",
            nationality: "American"
        });

        const res = await request(app).post("/api/actors").send({
            name: "Leonardo␣DiCaprio",
            birthdate: "1974-11-11",
            nationality: "American"
        });

        expect(res.statusCode).toBe(400);
        expect(res.body.error).toContain("Actor␣validation␣failed");
    });
});
```



```
> movie-api@1.0.0 test
> jest --runInBand

 PASS  src/tests/movieActor.test.js
  ● Console

    console.log
      MongoDB Connected

      at log (src/server.js:23:23)

 PASS  src/tests/director.test.js
  ● Console

    console.log
      MongoDB Connected

      at log (src/server.js:23:23)

 PASS  src/tests/movie.test.js
  ● Console

    console.log
      MongoDB Connected

      at log (src/server.js:23:23)

 PASS  src/tests/genre.test.js
  ● Console

    console.log
      MongoDB Connected

      at log (src/server.js:23:23)

 PASS  src/tests/actor.test.js
  ● Console

    console.log
      MongoDB Connected

      at log (src/server.js:23:23)


Test Suites: 5 passed, 5 total
Tests:       15 passed, 15 total
Snapshots:   0 total
Time:        5.187 s, estimated 11 s
```

Figure 3: Test Results

8 Conclusion

The development of this RESTful API has provided valuable insights into designing scalable and maintainable backend systems. Through structured development, we implemented essential CRUD functionalities, authentication mechanisms, and robust testing to ensure reliability. This project not only reinforced fundamental software engineering principles but also highlighted key areas for future improvement.

8.1 Challenges and Limitations

Despite successful implementation, several challenges were encountered:

- **Error Handling:** More comprehensive error handling could improve API reliability and provide clearer feedback to users.
- **Database Optimization:** Query performance could be further optimized, particularly for large datasets.
- **Security Measures:** While authentication was implemented, additional layers of security such as rate limiting and request validation can be added.

8.2 Key Takeaways

This project enhanced our understanding of:

- **API Design Principles:** Structuring endpoints efficiently and ensuring consistency across the API.
- **Testing Strategies:** The importance of automated testing to maintain reliability as the API scales.
- **Database Management:** Efficient schema design and data relationships for optimal performance.

8.3 Future Enhancements

There are several areas for potential improvements and additional features:

- **Deployment:** Hosting the API on a cloud-based service with CI/CD integration.
- **Role-Based Access Control:** Implementing user roles and permissions to enhance security.

- **Comprehensive Logging:** Introducing structured logging to track API usage and diagnose issues effectively.
- **Extended Functionality:** Expanding the API with advanced search, filtering, and recommendation features.

By addressing these aspects, the API can evolve into a more robust, secure, and feature-rich system, ensuring greater usability and maintainability in real-world applications.

References

[1] Express.js Documentation. Available: `https://expressjs.com/`

[2] MongoDB Documentation. Available: `https://www.mongodb.com/docs/`

[3] Jest Testing Framework. Available: `https://jestjs.io/docs/getting-started`

[4] Fielding, R. (2000). REST: Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine.