LAB 3 REPORT: LAYERED ARCHITECTURE IMPLEMENTATION

Project Name: Movie Booking Online System

Member Name: Nguyễn Đình Quyền - 23010485

Trần Hữu Kiên - 23010258

1. OBJECTIVES

The goal of this lab is to implement the **Movie Management** feature using a strict **Layered Architecture** in **Node.js**. The system is divided into three distinct layers (Presentation, Business Logic, Persistence) to ensure separation of concerns and maintainability.

2. PROJECT STRUCTURE

The project follows a standard layered structure, separating the API handling, business rules, and data access logic into different directories within `src/`.

Directory Tree:

- **`data/`**: Storage Layer (JSON files simulating the database).
- **`src/repositories/`**: Persistence Layer (Direct access to `data/`).
- **`src/services/`**: Business Logic Layer (Validations and logic).
- **`src/controllers/`**: Presentation Layer (HTTP Request/Response).
- **`server.js`**: Main entry point.

3. IMPLEMENTATION DETAILS

3.1. Data Model (Entity)

The **Movie** entity is stored as a JSON object in `data/movies.json`.

- **Structure:** `{ "id": number, "title": string, "genre": string, "price": number }`

3.2. Persistence Layer (Repository)

File: `src/repositories/movieRepo.js` Responsible for reading and writing directly to the `movies.json` file. It abstracts the file I/O operations from the rest of the application.

JavaScript
```
const fs = require('fs');
const path = require('path');
const dbPath = path.join(__dirname, '../../data/movies.json');

const getAllMovies = () => {
    try {
        if (!fs.existsSync(dbPath)) return [];
        const data = fs.readFileSync(dbPath, 'utf8');
        return JSON.parse(data);
    } catch (err) {
        return [];
    }
};

const addMovie = (movie) => {
    const movies = getAllMovies();
    movies.push(movie);
    fs.writeFileSync(dbPath, JSON.stringify(movies, null, 2));
    return movie;
};

module.exports = { getAllMovies, addMovie };
```

3.3. Business Logic Layer (Service)

File: `src/services/movieService.js` Handles business rules (e.g., validating that the movie title is not empty) before calling the Repository.

JavaScript
```
const movieRepo = require('../repositories/movieRepo');

const getMovieList = () => {
    return movieRepo.getAllMovies();
};

const createNewMovie = (title, genre, price) => {
    // Business Rule Validation
    if (!title || title.trim() === "") {
        throw new Error("Movie title cannot be empty.");
    }
    if (price <= 0) {
        throw new Error("Price must be a positive number.");
    }

    const newMovie = {
        id: Date.now(), // Auto-generate ID
        title,
        genre,
        price
    };

    return movieRepo.addMovie(newMovie);
};
```

```
module.exports = { getMovieList, createNewMovie };
```

3.4. Presentation Layer (Controller)

File: `src/controllers/movieController.js` Handles incoming HTTP requests, parses the body, calls the Service layer, and sends back the JSON response.

JavaScript
```
const movieService = require('../services/movieService');

const getMovies = (req, res) => {
    try {
        const movies = movieService.getMovieList();
        res.status(200).json(movies);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
};

const createMovie = (req, res) => {
    try {
        const { title, genre, price } = req.body;
        const savedMovie = movieService.createNewMovie(title, genre, price);
        res.status(201).json(savedMovie);
    } catch (error) {
        res.status(400).json({ error: error.message });
    }
};

module.exports = { getMovies, createMovie };
```

3.5. Application Entry Point

File: `server.js` Configures the Express server and links the routes.

JavaScript
```
const express = require('express');
const app = express();
const movieRoutes = require('./src/routes/movieRoutes'); // Assumes routes
map to controller

app.use(express.json());

// Link the routes
app.use('/api/movies', movieRoutes);

const PORT = 5000;
app.listen(PORT, () => {
    console.log(`Server running on port ${PORT}`);
});
```

4. EXECUTION AND TESTING

The system was tested using Postman to verify the flow from Controller -> Service -> Repository.

1. **Start Server:** `node server.js` -> *Output: Server running on port 5000*
2. **Test GET:** `GET http://localhost:5000/api/movies` -> *Returns list of movies.*
3. **Test POST (Valid):** `POST` with `{"title": "Dune 2", "price": 100}` -> *Returns 201 Created.*
4. **Test POST (Invalid):** `POST` with `{"title": "", "price": 100}` -> *Returns 400 Error.*