# Exercise: Integration Testing MongoDbCompass

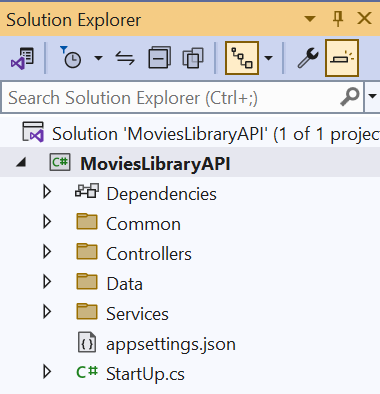
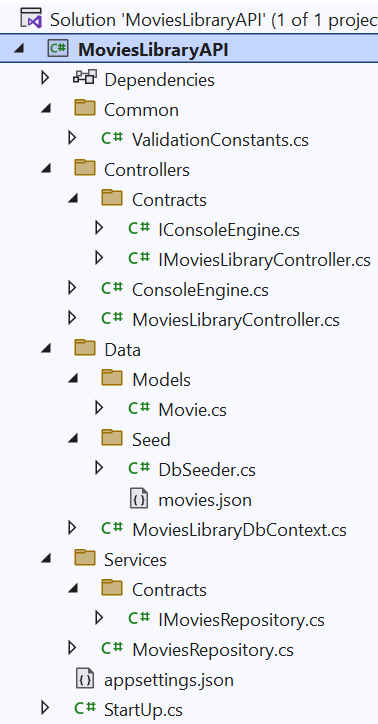
Exercise problems for the ["Back-End Technologies Basics"](https://softuni.bg/trainings/4398/back-end-technologies-basics-january-2024) Course [@ SoftUni](https://softuni.bg/).

## Introduction - MoviesLibraryAPI

* **MoviesLibraryAPI** is a console-based application built using ***.Net Core Framework*** that manages a collection of movies.
* It allows users to perform various operations, such as **creating**, **reading**, **updating**, **deleting** movies from database in **MongoDbCompass**.
* This application is designed to showcase **essential software development concepts and practices**.

## Architecture Overview

**MoviesLibraryAPI** follows a modular and organized architecture to ensure **maintainability** and **scalability**. The application is **divided into several key modules**, each **responsible for specific functionality**.

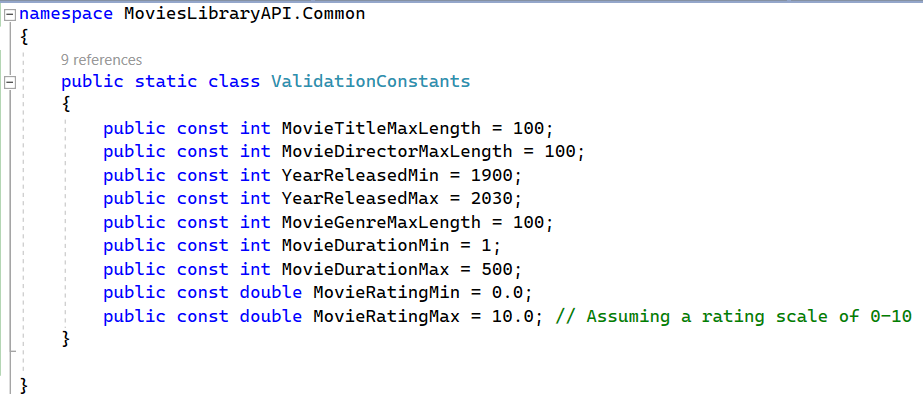
### Solution 'MoviesLibraryAPI'

This is the **main container for the entire application**, consisting of multiple projects.

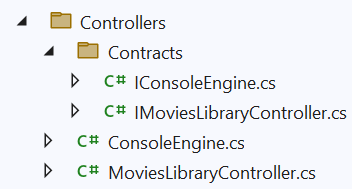
### MoviesLibraryAPI

This is the **primary project for the application**, likely serving as the **entry point for the API**.

### Common - The Common folder in the application contains shared constants that can be used across multiple layers of the application. In particular, the ValidationConstants class provides a centralized place to define the rules and constraints related to data validation for the domain models, particularly books in this context.



### Controllers – The Controllers are the components in the application that handle incoming requests, perform operations by leveraging services and/or repositories, and provide responses. Within the MoviesLibraryAPI project, the controllers would interact with the underlying data models and business logic to carry out the CRUD operations.

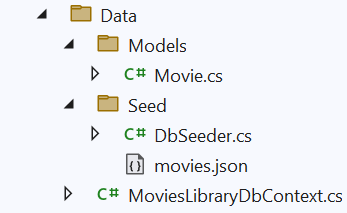


### Data - The Data folder is a critical component of the MoviesLibraryAPI, serving as the bridge between the application logic and the database. Here's an elaboration on the contents and purpose of this folder:

### Models: This subfolder keeps the data structures that represent the entities in the application. The Movie model within the Movie.cs file is a key part of this, defining the properties and behaviors of the movie entity that correspond to the records in the database.

* + - **Seed:** Within the Seed subfolder, you will find the **DbSeeder.cs** class. This class contains methods responsible for **populating the database with initial data**, which

is essential for both development and testing. The initial data set is provided in the **movies.json file**, which includes a collection of movies with predefined values. When the application starts or during the deployment process, the **DbSeeder.cs uses this JSON file to ensure the database is pre-filled with data**, which simulates a real-world scenario and provides a base for performing operations right away.



### Services - The Service Layer in the MoviesLibraryAPI plays a pivotal role in abstracting business logic and operations from the Controllers. It is a layer of the application that encapsulates the core functionality, acting as an intermediary for data transactions between the Controllers and the Data layer.

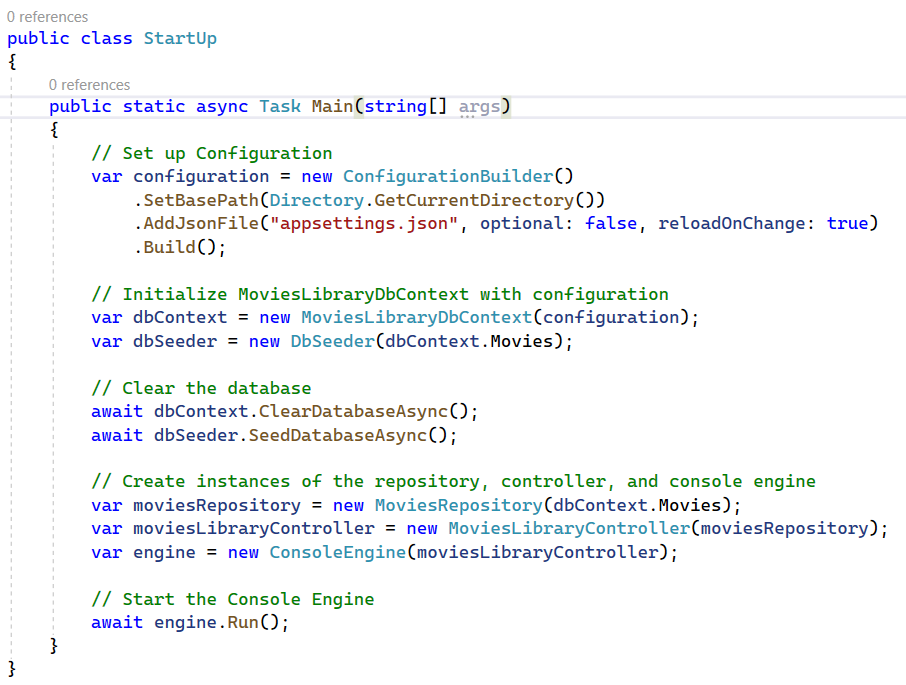
### Contracts - Within the Contracts folder, interfaces define the blueprint of services that the application will provide. The IMoviesRepository.cs file specifies the contract that any implementing class must fulfill to interact with movie data.

### MoviesRepository - The MoviesRepository.cs file provides a concrete implementation of the IMoviesRepository interface. This class contains the actual code that interacts with the DbContext to perform data operations. By following the repository pattern, this class abstracts the details of the data access logic from the rest of the applicati

### The separation of interfaces (Contracts) and their implementations makes the application more modular, easier to test, and maintain. It allows for the dependency inversion principle, one of the SOLID principles, which promotes the dependency on abstractions rather than concrete classes. This makes it easier to change the data source or the business logic without affecting the Controllers that consume these services.

### appsettings.json - This JSON file is where you store configuration data that the application can read at runtime. It typically includes settings like connection strings for databases, logging configuration, application parameters, and external service API keys. It is essential for the principle of separation of concerns, allowing the application's runtime behavior to be adjusted without code changes, by merely altering the settings file.

### StartUp.cs



### The StartUp.cs file serves as the entry point for the MoviesLibraryAPI. This is where the application starts execution. The Main method in this file is responsible for setting up the application, initializing services, and starting the main workflow.

* The **ConfigurationBuilder** is used to set up the application configuration. It reads from **appsettings.json**, which likely contains application settings such as the **connection string to the database.**
* An instance of **DbContext** is created using the configuration information. This context **will be used to interact with the database.**
* **Database Seeding** The application proceeds to potentially **clear the database and re-seed it with initial data**. This is done via the ClearDatabaseAsync and SeedDatabaseAsync methods, respectively. This ensures that the application has a predictable set of data to work with, which is crucial for development and testing environments..

### Repository and Controller Creation

### With the database context in place, the application creates an instance of MoviesRepository, which implements the logic defined by IMoviesRepository for interacting with the database..

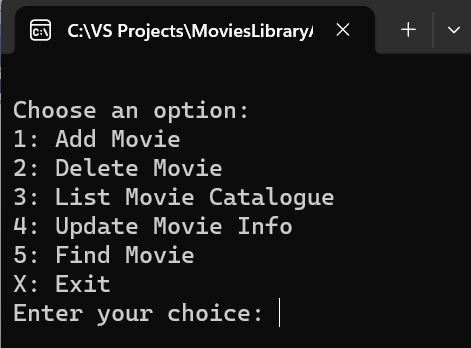
### Following that, it creates an instance of MoviesLibraryController, which likely handles incoming requests, and injects the repository instance into it. This setup follows the dependency injection principle, which facilitates loose coupling and easier testing.

* + - Finally, an **instance of a ConsoleEngine is created**, passing in the MoviesLibraryController. This ConsoleEngine is **responsible for running a loop to accept input from the console**, performing actions based on **commands entered by the user**. The **Run method of the engine is awaited**, which suggests **it contains an asynchronous loop or set of operations** that continue **until the application is closed or an exit command is issued**.

## Overview the Console UI

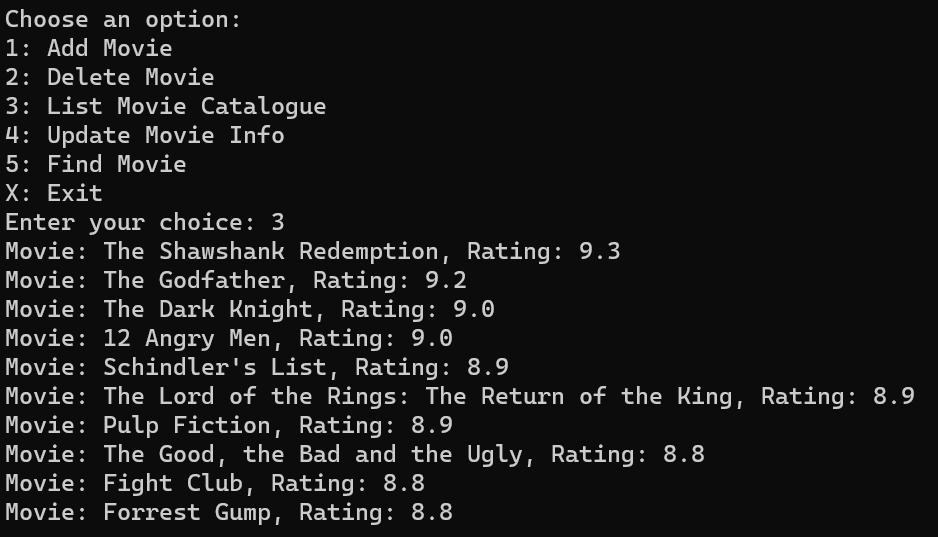
### User-Friendly Menu

* When the application is executed, it presents a menu with options that the user can select from, such as adding a movie, deleting a movie, listing all movies, updating movie details, or searching for a movie by title.



### Interaction Through Commands

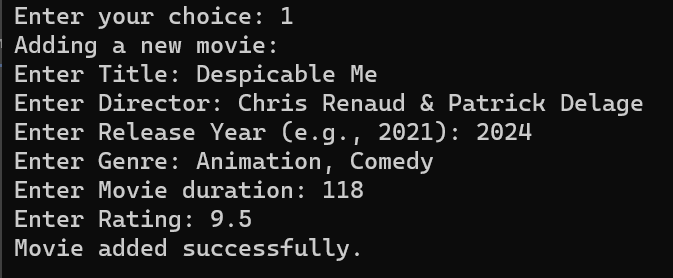
* **Users interact with the application** by entering commands or data into the console. The application processes these inputs and performs the corresponding actions.
* **Example:** choosing **option 3** in the Console UI initiates the process of **listing all movies** currently stored in the database. If the database is empty upon the application's initial launch, the seeding process populates it with a predefined set of movies from the **movies.json** seed file:

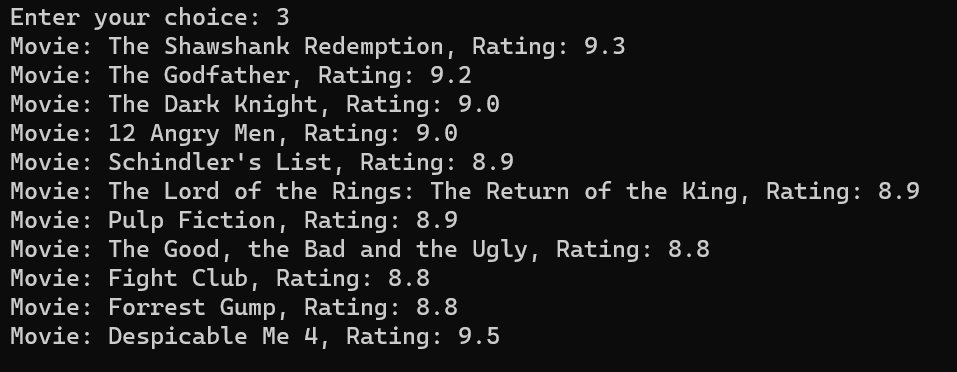


* **Fetching Data:** The application, through **the MoviesLibraryController**, calls the **GetAllMoviesAsync method** on the **MoviesRepository**. This **method is responsible for retrieving all movie records from the database**.
* **Process:** This retrieval is performed **asynchronously**, ensuring that the **application remains responsive**. Once the data is fetched, the **method iterates over the collection of Movie objects and prints their details** to the console

### Adding a New Movies

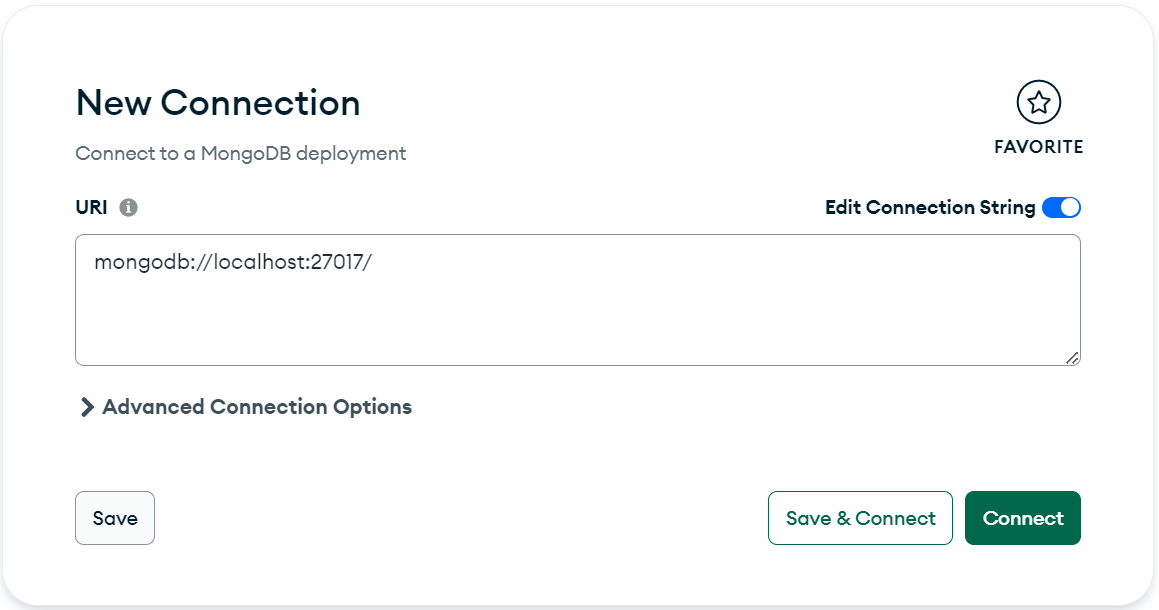
* **User Input:** When option 1 is selected, the **application prompts the user to enter details for a new movie**, such as title, director, release year, genre, movie duration - minutes, and rating.
* **Validation:** Input **data is validated** according to the rules **defined in the business layer** and the **model's data annotations**.
* **Persistence:** Valid data is **passed to MoviesLibraryController.AddAsync(newMovie)**, which **adds the movie   
  to the database**.



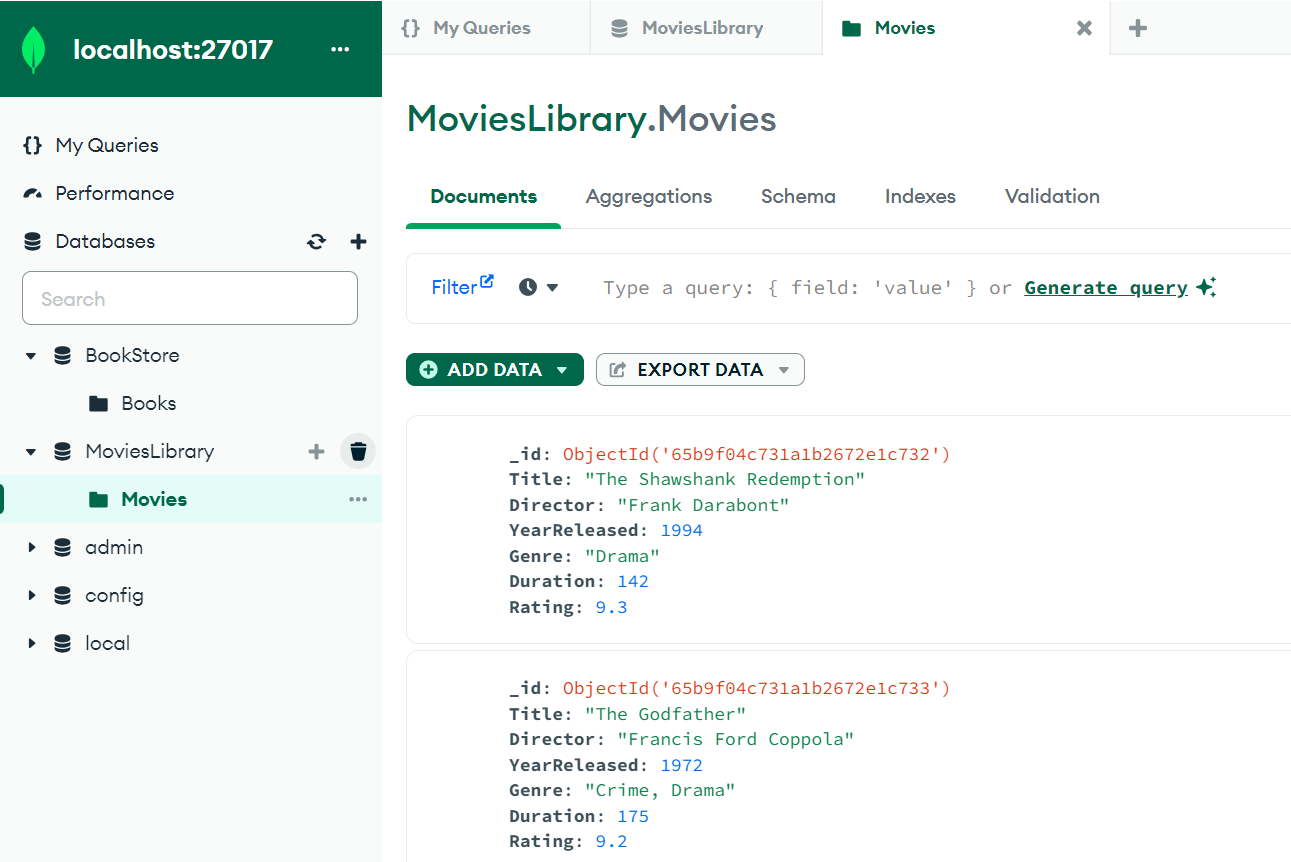


### Before Adding a Movie

* **Open Mongo Database:** Launch MongoDB Compassand connect to your MongoDb instance.

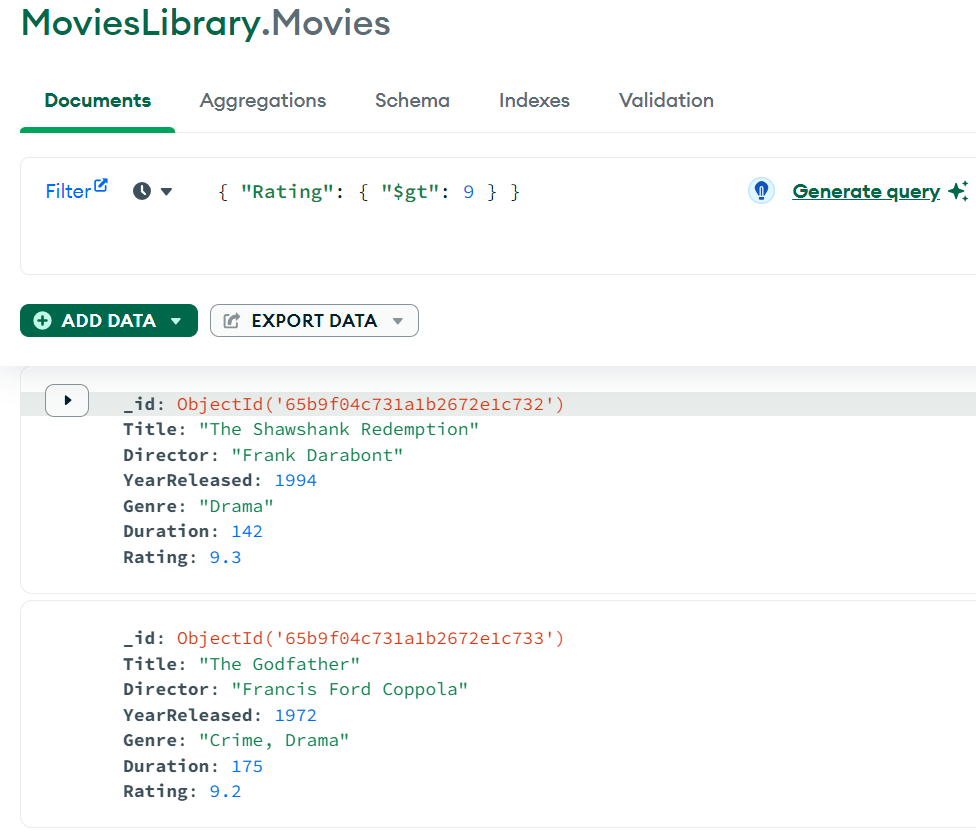
****

* **Navigate to the Database:** In Compass, once connected, you will see a list of databases on your MongoDB server. Click on the database you are using for your application (the name would correspond to what you've set in your **appsettings.json**).
* **Select the Collection:** Within your database, find the collection that contains your movie documents. In MongoDB, collections are equivalent to tables in SQL databases.

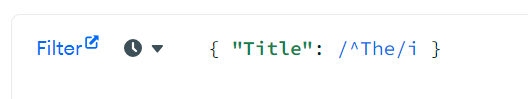


### Examples: Write and Execute Query

* **Open the Filter Bar:** At the top of the interface, there is a filter bar where you can **enter your query criteria**.
* **Enter the $gt Query:** Suppose you want to find **all movies in your database with a viewer rating higher than 9**. You would enter the following query into the filter bar:



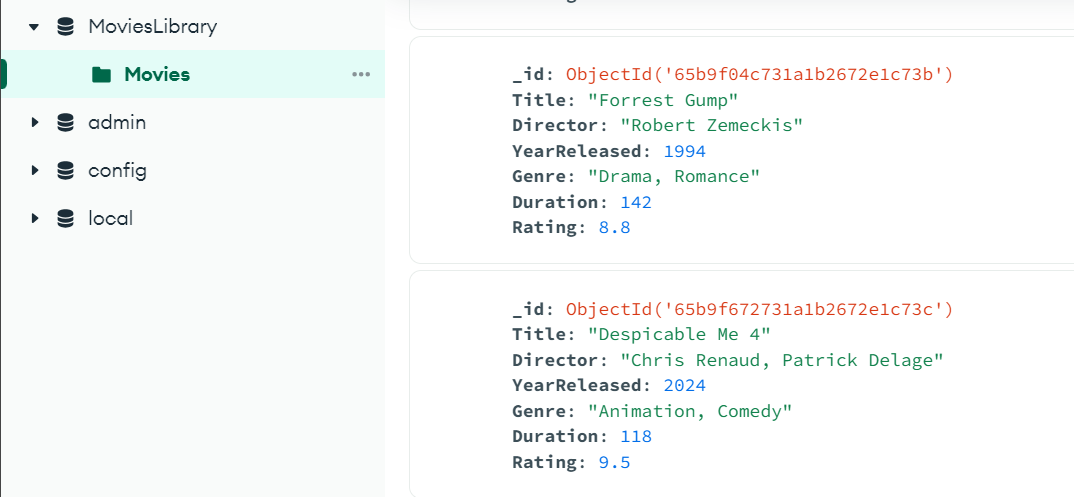
* **In this Query:** 
  + **"Rating"** is the name of the field you want to check.
  + **{ "$gt": 9 } is the condition,** with **$gt** being the operator and **9 the value you want the "Rating" to be greater than.**
* **Utilizing Regular Expressions for Pattern Matching**
  + For a pattern match, such as **finding movies with titles beginning with "The",** the correct query with regular expressions would be:



* + This uses the BSON regular expression syntax which is directly embedded in the query.

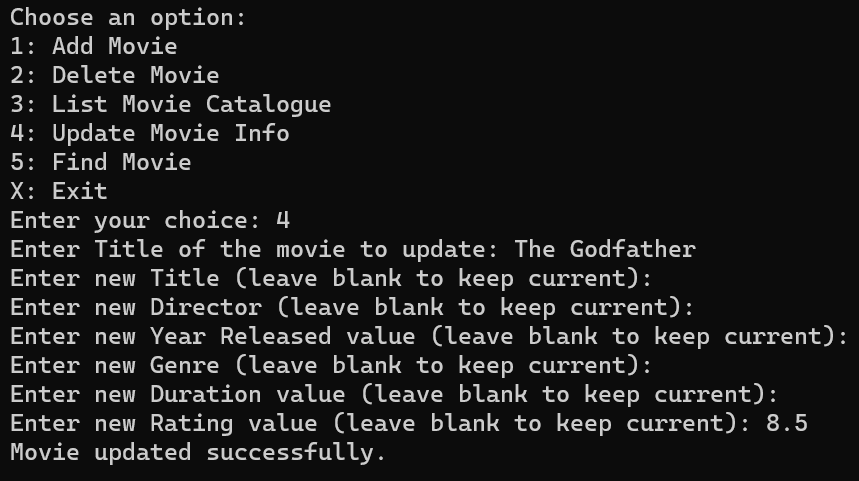
### After Adding a Movie

* **Add a Movie Through Your Application:** Use the Console UI of your application to add a new movie. Follow the prompts in the application to enter the details of the new movie and submit it.
* **Refresh the Data in Compass**
* **The new movie should appear in the Movies collection in the MoviesLibrary database.**

****

### Update a Movie:

* **Task:** Use the Console UI to update an existing movies's details. This might involve changing the title, author, year published, genre, pages, or price.

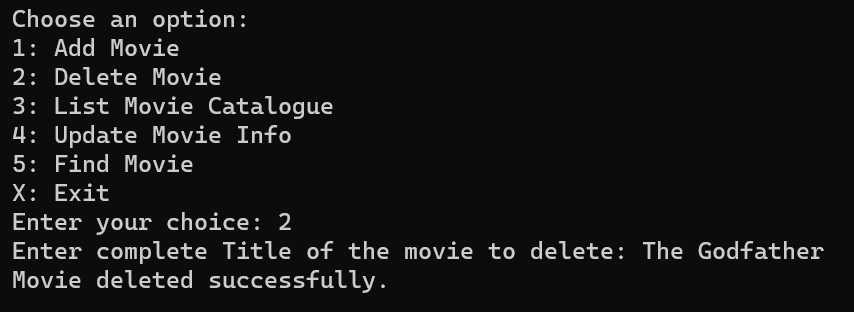




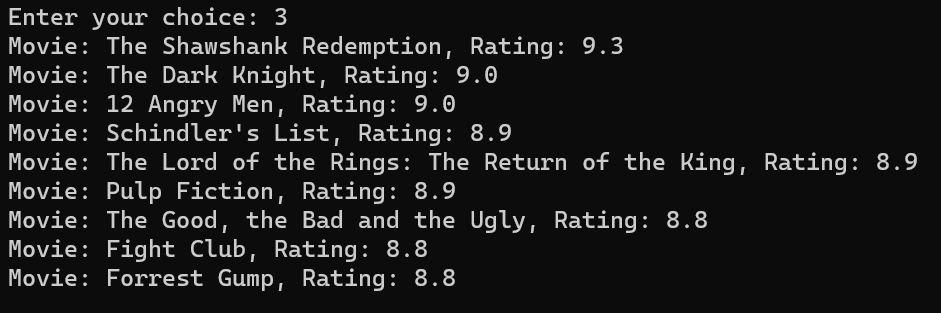
* **Process:**
  + Select the option to update a movie in the Console UI.
  + Enter the Title of the movie you wish to update.
  + Follow the prompts to enter new values for various attributes.
  + If you wish any of the values to remain unchanged, just hit Enter and proceed with the next value.
* **Database Check:**
  + Refrech the database in **MongoDbCompass** and check the Movies collection.
  + **Verify that the changes made** in the Console UI are **accurately reflected in the database.**

### Delete a Movie:

* **Task:** Use the Console UI to delete a movie from the database using its Title.
* **Process:**
  + Choose the **delete movie option** in the Console UI.
  + Enter the **Title** of the movie to delete.



* **Console UI check:**
  + Select the option ti list all movies
  + Observe the movie displayed and their details.
  + Ensure that the movie deleted is not existing in the list displayed.



* **Database Check:**
  + Refrech the database in **MongoDbCompass** and check the Movies collection.
  + **Verify that the changes made** in the Console UI are **accurately reflected in the database.**

## Integration Tests: Create and Implement the Integration Test Project

* As part of your journey through the software academy, you've reached a critical milestone where you'll enhance your skills in quality assurance and learn the **importance of testing in the software development lifecycle**. Specifically, you will focus on **Integration Testing using NUnit and xUnit** – two of the most popular testing frameworks in the **.NET** ecosystem.
* Your task is to **implement the test methods in the provided test projects**. These methods are placeholders that represent the **various scenarios** you will test within the application. For example, in **BookManagerFixture.cs**, you might write tests to ensure that the book management functionality is working as expected.

### NUnit Testing Project (MoviesLibraryAPI.IntegrationTests.NUnit)

### A project configured with the NUnit framework, ready for you to write integration tests for the LibroConsoleAPI.

### xUnit Testing Project (MoviesLibraryAPI.IntegrationTests.xUnit)

### An alternative project using the xUnit framework, which you can also use to implement integration tests.

### Implementing Tests

### Both projects are already configured with the necessary packages and references to work seamlessly with the MoviesLibraryAPI:

### NUnit or xUnit for defining and running tests.

### Additional Notes

* + **Test Isolation:** The test should be isolated, meaning it shouldn't be affected by or affect other tests.
  + **Asynchronous Code:** Integration tests often involve asynchronous operations, especially when dealing with database operations. Using await ensures that these asynchronous operations complete before assertions are made.
  + **Realistic Test Data:** While the data used in the test is arbitrary, it should resemble realistic data that the application might handle.

### Cover All the Logic / Methods in the MoviesLibraryController Class