MovieTracker Application - Jordi Schoetens R0983966

Project Overview

MovieTracker is a cross-platform movie tracker application built with .NET MAUI for the frontend and ASP.NET Core Web API for the backend. The application allows users to:

- View a list of movies
- Add new movies
- Edit existing movies
- Delete movies
- Persist data in a MySQL database

The application follows modern software architecture principles, including separation of concerns, dependency injection, and the MVVM (Model-View-ViewModel) pattern.

Architecture

The project follows a **client-server architecture**:

- **Frontend (Client):** .NET MAUI application that runs on multiple platforms (Android, iOS, Windows, macOS), Android for this project
- **Backend (Server):** ASP.NET Core Web API that provides RESTful endpoints
- **Database:** MySQL database for persistent data storage
- **Containerization:** Docker for easy deployment and management

```
┌─────────────────────┐
│   MAUI Frontend     │
│  (Cross-Platform)   │
└─────────────────────┘
          │
          │ HTTP/REST
          ▼
┌─────────────────────┐
│   ASP.NET Web API   │
│     (Backend)       │
└─────────────────────┘
          │
          │ Entity Framework
          ▼
┌─────────────────────┐
│   MySQL Database    │
└─────────────────────┘
```

Technologies Used

Backend

- **ASP.NET Core 9.0** - Web API framework
- **Entity Framework Core** - Database operations
- **MySQL** - Relational database

Frontend

- **.NET MAUI** - Multi-platform app UI framework
- **MVVM Pattern** - Model-View-ViewModel architecture
- **Dependency Injection** - Service management
- **HttpClient** - API communication

Containerization

- **Docker** - Application containerization platform
- **Docker Compose** - Multi-container orchestration and management

Project Structure

MovieTracker.WebAPI (Backend)

The backend project contains the following key components:

Controllers/

- `MovieTrackerAPIController.cs` - Defines the RESTful API endpoints for movie operations

DB/

- `MovieDBContext.cs` - Entity Framework database context that manages the connection to MySQL

Migrations/

- Migration files for database schema versioning and updates

Models/

- `Movie.cs` - Movie entity model representing the database table structure

Repository/

- `IMovieRepository.cs` - Interface defining the contract for data access operations
- `MovieRepository.cs` - Implementation of the repository pattern using Entity Framework Core

Configuration Files:

- `Program.cs` - Application entry point and service configuration
- `Dockerfile` - Docker container configuration
- `appsettings.json` - Application settings including database connection strings

MovieTracker (Frontend)

The frontend MAUI application is organized as follows:

Models/

- `Movie.cs` - Data transfer object (DTO) representing a movie

MovieService/

- `IMovieService.cs` - Service interface for API communication
- `MovieService.cs` - Implementation handling all HTTP requests to the Web API

ViewModels/

- `MainPageViewModel.cs` - Viewmodel for the main movie list page
- `AddMoviePageViewModel.cs` - Viewmodel for adding or editing movies

Views/

- `MainPage.xaml` - Main page user interface
- `MainPage.xaml.cs` - Main page code-behind
- `AddMoviePage.xaml` - Add/Edit page user interface
- `AddMoviePage.xaml.cs` - Add/Edit page code-behind

Configuration Files:

- `MauiProgram.cs` - Dependency injection and service registration
- `AppShell.xaml` - Application shell and navigation structure

Backend - Web API

Movie Model

The `Movie` class represents a movie entity in the database:

```
public class Movie
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Genre { get; set; }
    public DateOnly Year { get; set; }
    public string Rating { get; set; }
}
```

Repository Pattern

The application implements the **Repository Pattern** to abstract data access:

- **IMovieRepository:** Defines the contract for movie data operations
- **MovieRepository:** Implements the interface using Entity Framework Core

Database Context

MovieDBContext inherits from DbContext and manages the connection to the MySQL database:

```
builder.Services.AddDbContext<MovieDBContext>(options =>

options.UseMySql(builder.Configuration.GetConnectionString("DefaultConnection")
,
        new MySqlServerVersion(new Version(8, 0, 21))));
```

Automatic Migrations

The application automatically applies database migrations on startup:

```
using (var scope = app.Services.CreateScope())
{
    var dbContext = scope.ServiceProvider.GetRequiredService<MovieDBContext>();
    dbContext.Database.Migrate();
}
```

Frontend - MAUI Application

MVVM Pattern

The application follows the **MVVM (Model-View-ViewModel)** pattern:

- **Model:** Represents the data (Movie class)
- **View:** XAML pages that display the UI
- **ViewModel:** Contains presentation logic and communicates with services

Dependency Injection

Services and ViewModels are registered in MauiProgram.cs:

```
// Singleton services
builder.Services.AddSingleton<HttpClient>();
builder.Services.AddSingleton<IMovieService, MovieService>();

// Transient ViewModels and Pages
builder.Services.AddTransient<MainPageViewModel>();
builder.Services.AddTransient<AddMoviePageViewModel>();
builder.Services.AddTransient<MainPage>();
builder.Services.AddTransient<AddMoviePage>();
```

Service Lifetimes:

- **Singleton:** One instance for the entire application (HttpClient, MovieService)

- **Transient:** New instance every time it's requested (ViewModels, Pages)

Movie Service

`MovieService` handles all communication with the Web API:

- **LoadMoviesAsync()** - Fetches all movies from the API
- **AddMovieAsync()** - Adds a new movie
- **UpdateMovieAsync()** - Updates an existing movie
- **DeleteMovieAsync()** - Deletes a movie

The service uses `ObservableCollection<Movie>` to automatically update the UI when data changes.

ViewModels

MainPageViewModel

- Displays the list of movies
- Handles movie selection
- Provides commands for deleting and editing movies
- Uses `INotifyPropertyChanged` for data binding

AddMoviePageViewModel

- Handles adding new movies
- Handles editing existing movies
- Validates user input
- Communicates with MovieService to persist changes

Database

MySQL Configuration

Database Details:

- **Database Name:** movietracker
- **User:** api
- **Password:** api

Schema

The `Movies` table contains:

- `Id`
- `Title`
- `Genre`
- `Year`
- `Rating`

Entity Framework Migrations

Migrations are stored in `Migrations/` folder and are automatically applied on application startup.

To create a new migration:

```
dotnet ef migrations add MigrationName --project MovieTracker.WebAPI
```

Docker Configuration

docker-compose.yml

The application uses Docker Compose to manage two services:

1. **MySQL Database Container**
   - Image: `mysql:8.0`
   - Port: 3306
   - Volume: Persistent data storage
   - Health check: Ensures database is ready before API starts
2. **ASP.NET Web API Container**
   - Built from Dockerfile
   - Port: 5000 (host) → 8080 (container)
   - Depends on MySQL container

Network

Both containers are connected via a custom network (`movietracker-network`) for inter-container communication.

Starting the Application with Docker

```
docker-compose up --build
```

This command:

1. Builds the API container
2. Starts the MySQL container
3. Waits for MySQL to be healthy
4. Starts the API container
5. Applies database migrations
6. API becomes available at `http://localhost:5000`

API Endpoints

The API is available at `http://localhost:5000/api/MovieTrackerAPI`

GET /api/MovieTrackerAPI

Description: Get all movies
Response: 200 OK with array of movies

GET /api/MovieTrackerAPI/{id}

Description: Get a movie by ID
Parameters: `id`
Response: 200 OK with movie object

POST /api/MovieTrackerAPI

Description: Add a new movie
Request Body: Movie object (JSON)
Response: 201 Created with location header

PUT /api/MovieTrackerAPI/{id}

Description: Update an existing movie
Parameters: `id`
Request Body: Updated movie object (JSON)
Response: 204 No Content

DELETE /api/MovieTrackerAPI/{id}

Description: Delete a movie
Parameters: `id`
Response: 204 No Content

The screenshots inside the project folder demonstrate the API working with Postman.

Setup and Installation

Prerequisites

- .NET 9.0 SDK
- Docker Desktop
- Visual Studio Code and VS Code

Running the Backend With Docker

1. Navigate to the project root directory in VS Code
2. Run Docker Compose:

```
docker-compose up --build
```

3. The API will be available at `http://localhost:5000`

Running the Frontend

1. Open the solution in Visual Studio
2. Set `MovieTracker` as the startup project
3. Select target platform, Windows or Android emulator
4. Press Run project

Conclusion

This MovieTracker application demonstrates:

- Full-stack .NET development skills
- Modern architectural patterns (MVVM)
- Cross-platform mobile development with .NET MAUI
- RESTful API design and implementation
- Database management with Entity Framework Core
- Containerization with Docker
- Dependency injection and service-oriented architecture