Assignment
Create a Web API and document it.

Film API

Create an Entity Framework Code First workflow and an ASP.NET Core Web API in C#. Follow the guidelines given below, feel free to expand on the functionality. It must meet the minimum requirements prescribed.

1) Set up the development environment.

Make sure you have installed at least the following tools:

- Visual Studio 2022 with .NET 6.
- SQL Server Management Studio

2) Optional: Database diagrams

You can use SSMS to draw relationship diagrams of your database, you can include these in your repository.

3) Use Entity Framework Code First to create a database with the following minimum requirements (See Appendix A for details):

a) Create models and DbContext to cater for the specifications in Appendix A.
b) Proper configuration of datatypes is to be done using data attributes.
c) Comments on each of the classes showing where navigation properties are and aspects of DbContext.
d) Connection String should not be hard coded into DbContext.

4) Create a Web API in ASP.NET Core with the following minimum requirements (See Appendix B for details):

a) Create controllers and DTOs according to the specifications in Appendix B.
b) Encapsulate DbContext into Services for each domain entity. Movie, Character, and Franchise.
c) Swagger/Open API documentation using annotations.
d) *Summary (///) tags* for each method you write, explaining what the method does, any exceptions it can throw, and what data it returns (if applicable). You do not need to write summary tags for overloaded methods.

5) Submit

a) Create a GitLab repository containing all your code.
b) You can include the generated class diagram in this repository if you have made one.
c) The repository must be private, add your lecturer as a Maintainer.
d) Submit only the link to your GitLab repository (not the “clone with SSH”).
e) Only one person from each group needs to submit but add the names of both group members in the submission.

Appendix A: EF Core Code First

1) Introduction and overview

You and a friend have been tasked with creating a datastore and interface to store and manipulate movie characters. It may be expanded over time to include other digital media, but for now, stick to movies.

The application should be constructed in ASP.NET Core and comprise of a database made in SQL Server through EF Core with a RESTful API to allow users to manipulate the data. The database will store information about **characters**, **movies** they appear in, and the **franchises** these movies belong to. This should be able to be expanded on.

NOTE: This is an overall description of the system to provide some context.

2) Business rules

The following sub-sections described how the above-mentioned entities interact and what rules they are governed by.

Characters and movies

One **movie** contains many **characters**, and a **character** can play in multiple **movies**.

Movies and franchises

One **movie** belongs to one **franchise**, but a **franchise** can contain many **movies**.

For example: The Marvel Cinematic Universe contains 23 movies, and the Lord of the Rings franchise contains both the good trilogy and that other Hobbit one.

3) Data requirements

The following subsections detail the minimum required information to be stored for each entity, you can add more fields if you would like. The foreign keys are omitted – this is up to you.

Character

- Autoincremented Id
- Full name
- Alias (if applicable)
- Gender
- Picture (URL to photo – do not store an image)

Movie

- Autoincremented Id
- Movie title
- Genre (just a simple string of comma separated genres, there is no genre modelling required as a base)
- Release year
- Director (just a string name, no director modelling required as a base)
- Picture (URL to a movie poster)
- Trailer (YouTube link most likely)

Franchise

- Autoincremented Id
- Name
- Description

Note: The nullability and length limitations are up to you to decide what makes sense given the context. Every string should not be NVARCHAR(MAX).

4) Seeding

You are to create some dummy data using seeded data. You are to add at least 3 movies, with some characters and franchises.

Appendix B: Web API using ASP.NET Core.

1) Introduction and overview

This section and subsections detail the requirements of the endpoints of the system on a high level. The naming of these endpoints and the controller they are contained in is up to you to decide and forms part of the convention mark.

2) API requirements

Generic CRUD

Full CRUD is expected for **Movies**, **Characters**, and **Franchises**. You can do proper deletes, just ensure related data is not deleted – the foreign keys can be set to null. This means that Movies can have no Characters, and Franchises can have no Movies. This will be done in the services.

NOTE: This is done with setting nullability of the relationships, you can use seeded data to test out if related data is being deleted or its working as intended.

Updating related data

In addition to generic "update entity" methods, there should be dedicated endpoints to:

- Updating **characters** in a **movie**.
    - This can take in an integer array of character Id's in the body, and a Movie Id in the path.
- Updating **movies** in a **franchise.**
    - This can take in an integer array of movie Id's in the body, and an Franchise Id in the path.

NOTE: This can be done by updating related data, i.e. Movie.Characters.

Reporting

At a high level, your application should provide the following reports in addition to the basic reads for each entity:

- Get all the **movies** in a **franchise**.
- Get all the **characters** in a **movie**.
- Get all the **characters** in a **franchise**.

NOTE: You can easily scaffold out controllers with ASP.NET Core to save time when doing generic CRUD. Ensure the controllers are named appropriately and the actions within are logically grouped, also remember to move your DbContext calls to services. *Hint*: Anything to do with a franchise, i.e., movies in franchise or characters in franchise should appear in the FranchiseController. Same goes for the other controllers.

3) DTOs with AutoMapper

You are to make data transfer objects (DTOs) for all the model classes clients will interact with. There should be DTOs to encapsulate all requests – i.e. Adding new entities, updating existing,

getting existing entities. You are to use AutoMapper to create mappings between your domain models (entities) and these DTOs.

NOTE: Remember what a DTOs role is; to decouple your client from your domain and to prevent over-posting and exposing internal schemas. Use this to guide how your DTOs will look. The DTO mapping must happen in the controller –return the domain object from the services then map it in the controller.

4) Documentation with Swagger

You will need to create proper documentation using Swagger / Open API. This implementation required extra attributes to be added to controller methods. Once everything is functional, you should look to incorporate this documentation.

NOTE: We can use ///Summary tags with some libraries to help generate Swagger Documentation.

5) Services and/or Repositories

The use of **Services/Repositories** can really help clean up your controllers when there is a lot of DbContext manipulation that needs to be done. You should use at least Services, Repositories are optional. Ensure these Services/Repositories you make can be injected (registered in builder.Services).

6) Deployment (Optional)

You can create a CI pipeline to build your application as a Docker artifact. You can use this base template in your gitlab-ci.yml file.