Assignment
Create a Web API and database with Spring.

Movie Characters API

Create a PostgreSQL database using Hibernate and expose it through a Web API. 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:

- IntelliJ with Java 17
- PostgreSQL with PgAdmin

2) Optional: Database diagrams

You can draw relationship diagrams of your database and include these in your GitLab repository.

3) Use Hibernate to create a database with the following minimum requirements (See Appendix A for details):

a) Create models and Repositories to cater for the specifications in Appendix A.
b) Create representations of domain object as to not overexpose your database data, this is detailed in Appendix B. You can make these changes at a later stage in the assignment (i.e. When you want to test the endpoints).

4) Create a Web API in Spring Web with the following minimum requirements (See Appendix B for details):

a) Create controllers according to specifications in Appendix B.
b) Swagger/Open API documentation.

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 either public, or I am added as a Maintainer (@NicholasLennox).
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: Hibernate

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 Spring Web and comprise of a database made in PostgreSQL through Hibernate 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 is 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 Spring Web.

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.

Do not worry about adding a new resource and a new related resource (adding a new movie with a list of new characters), these related resources should already exist, and a reference (single id or list of id's – whichever is applicable) should be used, this is for simplification with cascades. E.g.

```
{
    "fullName": "Ben Solo",
    "alias": "Kylo Ren",
    ...
    "movies": [
        {
            "id": 1
        },
        {
            "id":3
        }
    ]
}
```

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.setCharacters**

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: Ensure the controllers are named appropriately and the methods within are logically grouped. *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) Domain representation

You should not show related data as domain objects, it should be changed with a @JsonGetter to represent a URI to GET that resource. Consult the online notes for a refresher.

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 – if you are using a service or repository, return the domain object then map it in the controller.

4) Documentation with Swagger

You will need to create proper documentation using Swagger / Open API. A good article about this can be found here. This is something you should look to add once all the functionality is complete.

5) OPTIONAL: Services and DTOs

The use of **Services** can really help clean up your controllers when there is a lot of extra logic manipulation that needs to be done. Ensure these Services you make can be injected (tagged as @Service).

The same goes for DTOs, if you want to add them in and map from domain objects over, feel free to. This allowed more flexibility with the information exposed and allows you to leave out the @JsonGetter's in the models.