Context and introduction

Backend is separated in 2 separate solutions:

APIef – RESTful API created to save, edit and provide necessary data for mobile application. It is created using ASP.NET CORE API and Entity Framework.

Socket – Simple socket server that allows us to create real time application, but is implemented just as support for RESTful API. It does that by sending notifications to clients that let client know when to call API for needed data. sends notifications to clients that let them sends notifications to clients that let them

Description of app flow:

Most of App operations are made via API calls, starting from simple ones like login or registration of user ending at letting user receive and modify details of room in real time. App uses socket to send simple messages to clients in group like a New user joined room or picking phase started, you can get list of movies from API and start choosing movies you want to watch. Combining those two re real-timereallyfferent approaches for client-server we created real time app that uses RESTful API as its main source of data.

Technical Requirements:

-Backend uses http protocol to communicate with app.

-Response format of API is JSON

-API is RESTful so it uses GET, POST, PUT, PATCH and DELETE methods

-Authentication and Authorization is made through JWT token

-API uses: /api/genres, /api/moviecollections, /api/roles, /api/rooms, /api/tokens, /api/users as its main endpoints. They are specified in their own sections bellow in the document.

-API uses relational database MSSQL to store data

-As source of movie details backend uses TMDB API

SOCKET:

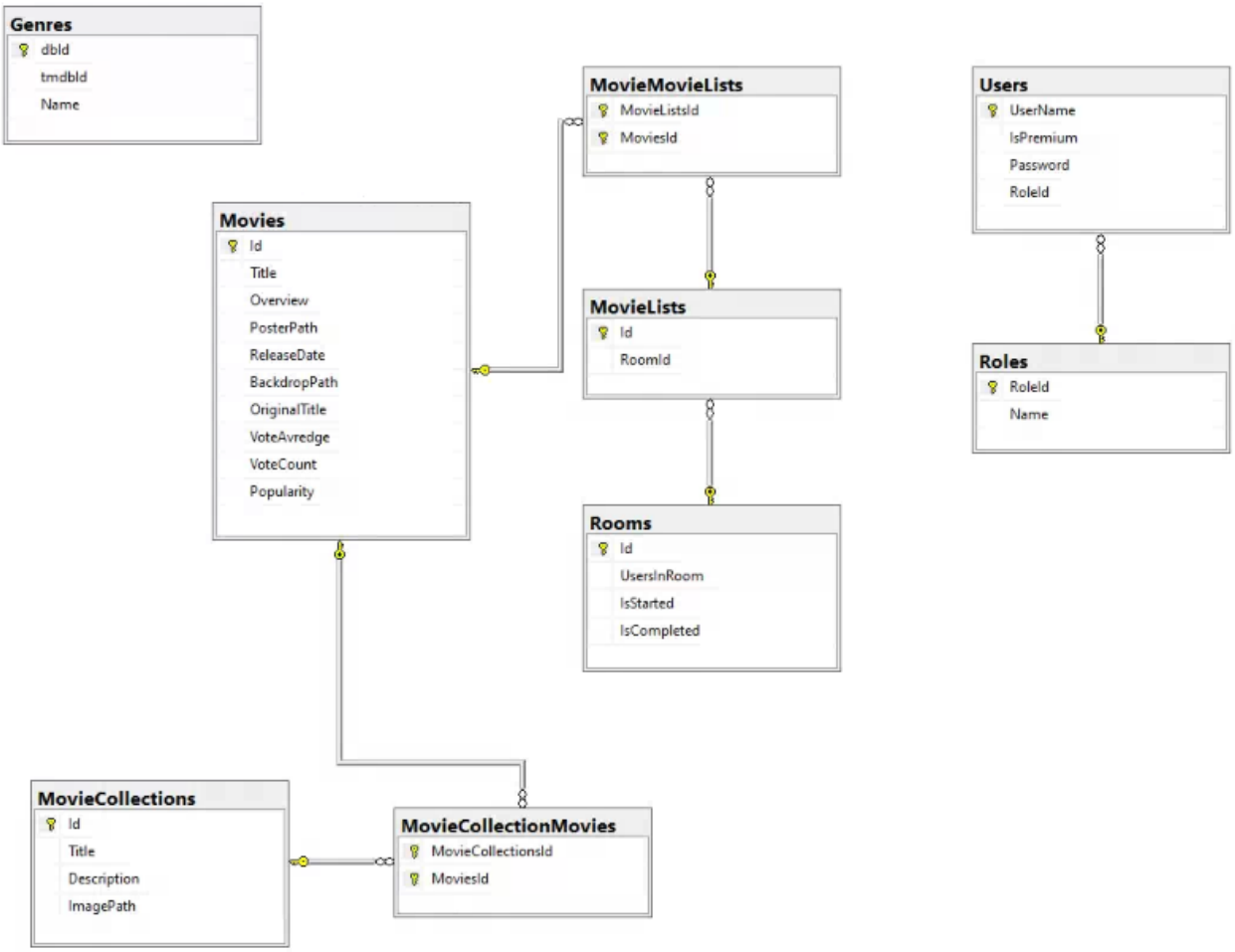
Socket is based on SignalR, it serves only one purpose: to notify clients that they can pull data from API.

Class Diagrams:

API Class Diagram:

In diagram above we can see that app models are clearly devided. There is group of models asociated with User: User, Role, UserCredentials and ChangePassword. And there is also other group asociated with Movie: Movie, Genre, MovieList, MovieCollection and Room.

Database Diagram:



Database diagram isn't that different to Class diagram, biggest difference is in junction tables that allow many-to-many relationship.

Model descriptions:

User group:

-**Role:** Role class is self explanatory, it defines what role user has, is it regular user, premium user or admin

-**User:** User model is a class that stores user data, his role, Username and Password

-**UserCredentials:** UserCredentials model serves as simple model that makes operations like login much easier. It contains just Username and Password and thanks to that we can use it for verification

-ChangePassword: Name of this model might not be the prettiest but its self explanatory, this model is created just for changing user password. It contains user's login, oldpassword, and newpassword.

Movie group:

-Genre: Genre model as its name states is class containing information about genre: its TMDBID that we use to receive movies from tmdbAPI and Genre name that we can show to user.

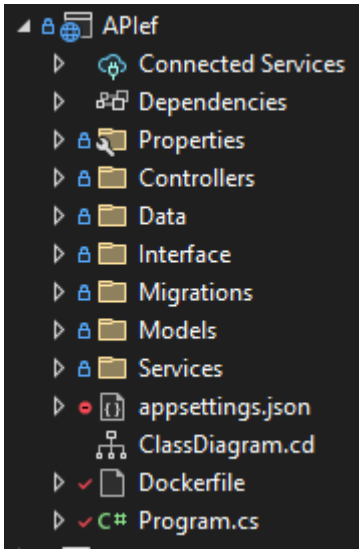
-Movie: Movie is class that contains all information about movie that we are able to receive from external api

-MovieList: As the name sugests it is list of movies

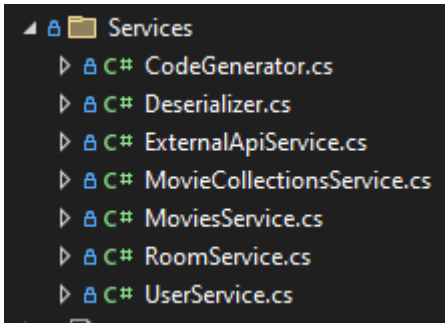
-MovieCollection: This class is used to store collections of movies that administrators/moderators an craete. It contains name, description, imagepath that we can display in app and list of movies.

-Room: This is class that is the mainlar of application. It stores list of MovieLists, stores bool values: IsCompleted and IsStarted that are used to determine if users can start picking movies or if final list of movies is ready and also it stores intiger value: usersinroom that the backends to check if room can be marked as finished or not yet.

Folder structure



Folder structure is quite simple and as close to standards as i could get, Controllers folder as name sugests stores all controllers, Data folder stores DbContext, Interface folder stores interfaces for repositories, Migrations folder is default folder generated by entity framework and stores project's migrations, Models as name sugests stores models and Services stores all services that app uses and repositories for managing data in db.



Libraries used for backend:

-EntityFramework: Well-known and widely used library for connecting code with database, we chose it since it doesn't have any competitor on the market.

-Newtonsoft.Json: Popular for json serialization, we use it alternately with built-in serializer cause both of them have quite different syntax and in some cases it was just easier to find a solutionhat uses one of those two, and in some cases one of those just fit better.

-SignalR: Microsoft's open-source made for real time client-server communication. It was made for ASP.NET which made it perfect for our use case

Endpoints:

-api/genres:

-GET: Gets list of genres. Can be used by Regular user.

-POST: Post new genre and save in db. Can be used only by admin.

-api/moviecollections:

-GET: Gets list of moviecollections, Can be used by Regular user.

-GET{id}: Gets moviecollection by id, Can be used by Regular user.

-POST: Gets title and description from query, and list of movie ids (string) and creates moviecollection out of this data and saves it in db. Can be used only by admin.

-POST(test/CreateTestCollection): Endpoint created just to make testing easier, it gets list of movies from api and saves test collection in database. Can be used only by admin.

-api/roles:

-GET: Gets list of all roles. Can be used only by admin.

-POST: Takes name of new role from request body and creates new role in db. Can be used only by admin.

-DELETE{id}: Deletes role from db. Can be used only by admin.

-api/rooms:

-GET{id}: Gets room by id. Can be used by Regular User.

-POST{?option?movie?ammount?collectionID}: Also it can take nullable List<int> genreList. Option is string that can take 2 values, “discover” or “collection” it defines which way of generating starter movielist you prefer. If you chooose “collection” you also are required to fill collectionId, than room will be created and starter list will simply be taken from collection with the same id as you provided. If you choose discover, you need to fill: movie, ammount and genreList. Movie is bool that defines if you want to look for movie or tv series, ammount is intiger that defines how many movies you want in your starter list, genreList as its name sugests is list of genre ids, starter movie list will be generated based on those parameters. This call to api will return room in form of JSON and can be used by Regular user.

-POST{id}: This call also takes list of movies from body, if roomId is correct and room is already started it takes list of movies and saves it in room. If amount of lists sent by users is equal to users in room it changes room status to completed and sends message to user “Picking Phase Completed”. Can be used by Regular user.

-PATCH{id?option}: This call is meant for adding or deleting user from room. Option parameter is a string that can be equal to either “add” or “remove”. It adds or removes 1 from usersinroom. Can be used by Regular user.

-Patch{id}: This call changes isStarted property of room to true. Can be used by Regular user.

-Delete{id}: It deletes room. Can be used by Regular user.

-api/tokens:

POST: This call can be also called login. It takes user credentials, checks if they are correct and if they are it sends JWT token in form of JSON.

-api/users:

GET: Gets list of all users. Can be used only by admin.

GET{userName}: Gets user by his username(string). Can be used by Regular user.

POST: This call can be called register. It simply takes user credentials, checks if user with same username already exists and if not adds new user to db. Can be used by anyone.

DELETE{userName}: This call deletes user from db. It can be used by admin.

PATCH{userName}: This call also takes object change password, it checks if credentials are matching and if yes it changes password of user.

POST(/admin): This call can be performed only by admin and it creates new admin account in db.

TESTS:

User:

User1Register: this test checks if registering new user works.

LoginWrongPass: this test checks that if you will get negative status code after typing wrong password

User2Login: this test checks if you can succesfully login with correct credentials.

User3ChangePassword: this test checks if you can change password.

User4IsPasswordChanged: this test gets user from db and checks if password was changed.

CheckIfUserGetterDosentReturnNull: check if getting all users works.

DeleteUserDoesNotExist: check if deleting nonexistant user returns error.

User6DeleteUser: checks if you can delete user.

Genres:

CheckIfGenreGetterHasData: checks if genre getter returns data.

CheckIfGenreGetterDosentReturnNull: checks if genre getter doesnt return null.

CanRegularUserCreateGenre: checks if regular user gets forbidden response from api after trying to create genre.

Roles:

CheckIfRoleGetterDosentReturnNull: check if role getter doesnt return null.

CheckIfRoleGetterHasData: check if role getter has data.

MovieCollections:

PostMovieCollectionRegular: try if you can post movie collection as regullar user and check if response code was forbidden.

Col1PostMovieCollectionAdmin: check if admin can post movie collection

Col2GetMovieCollections: check if collection getter works.

Col3GetMovieCollectionById: check if you can get movie collection by id

Col5DeleteMovieCollection: check if you can delete collection.

Col7GetMovieCollectionByIdAfterDelete: check if deleted collection is no more aviable to get

Rooms:

TryToCreateRoomUnauthorized: check if you can create room without being logged in

TryToCreateRoomDiscoverWithoutNecessaryData: try if you can create room using discover option without providing non nullable data

TryToCreateRoomCollectionWithoutCecesarryData: check if you can create room using collection option without providing non nullable data

Col4CollectionRoomFlowTest: check if you can create, get freshly created room by id, add user to room, start room, post movie lists to room, check if room was marked as completed after receiving movie lists, delete room and check if it was deleted for room created with collection option. I had to make all of those separate tests into one because room id is randomly generated and is unique, and because tests cannot change value of global variable (at least i

didnt know any non hacky way of doing that) i had to do whole room flow in one test to have access to its id.

RRDiscoverRoomFlowTest: same as the above but for room created with discover option.