# Exercise: Introduction to .NET Core and EF Core

This document defines the **in-class exercises** assignments for the ["C# Web Development Basics" course @ Software University](https://softuni.bg/courses/csharp-web-development-basics). You can submit your code in the course page.

# Student System

## Create models

Your task is to create a database for the **Student System**, using the **Entity Framework Code First** approach. Separate the **models**, **data** and **client** into **different layers** (projects). Model the following tables:

* **Students**: id, name, phone number (**optional**), registration date, birthday (**optional**)
* **Courses**: id, name, description (**optional**), start date, end date, price
* **Resources**: id, name, type of resource (**video / presentation / document / other**), URL
* **Homework**: id, content, content-type (**application/pdf/zip**), submission date

Table relations:

* **Students** can be in **many course**s
* **Courses** can have **many students**
* **Courses** can have **many resources**
* **One course** can have **many homework submissions**
* **Homework submissions** have a **student**



Add **navigational properties** in all models to simplify navigation. Annotate the data models with the appropriate **attributes** and validations and **enable code first migrations**.

## Working with the Database

Write a console application that works with the EF data layer and performs some CRUD operations.

### Tasks

1. Lists **all students** and their **homework submissions**. Print only their **names** and for each homework - **content** and **content-type**.
2. ist **all courses** with their corresponding **resources**. Print the **course name** and **description** and everything for each **resource**. Order the courses by start date (ascending), then by end date (descending).
3. List **all courses** with **more than 5 resources**. Order them by **resources count** (descending), then by **start date** (descending). Print only the **course name** and the **resource count**.
4. \*List all **courses** which were active on a **given date** (choose the date depending on the data seeded to ensure there are results), and for each course count the **number of students enrolled**.   
   Print the **course name**, **start** and **end date**, **course duration** (difference between end and start date) and **number of students enrolled**. Order the results by the **number of students** enrolled (in descending order), then by **duration** (descending).
5. For each student, calculate the **number of courses** he/she has enrolled in, the **total price** of these courses and the **average price per course** for the student.  
   Print the **student name**, **number of courses**, **total price** and **average price**. Order the results by **total price** (descending), then by **number of courses** (descending) and then by the **student's name** (ascending).

## Resource Licenses

Resources should now have many **licenses**. A **license** should have an **Id** and **Name**.

Make these changes using Code First Migrations. Make sure no data is lost after the update.

### Tasks

1. List **all courses** with their corresponding **resources**. Print the **course name**, the **resource name** and the **names** of all the **licenses** that resource has (if any). Order the courses by **resources count** (**descending**), then by **course name** (**ascending**). **Order** **resources** by **licenses count** (**descending**) and then by **name** (**ascending**).
2. For **each student** print the **name**, the **count of courses** he or she is enrolled, the **total number of resources** for their courses and the **total number of licenses** those resources have. Order the results by **number of** **courses** (**descending**), then by **number of recourses** (**descending**), then by **name** (**ascending**).

# Social Network

Create a database for a new social network. You will have to store users and albums with pictures. You should implement appropriate relations between the entities.

## Table Users

Your task is to create table **Users** using the Code First approach. The table should contain the following fields:

1. **Id** – Primary Key (number in range [1, 231-1])
2. **Username** – Text with length between 4 and 30 symbols. Required.
3. **Password** – Required field. Text with length between 6 and 50 symbols. Should contain at least:
   1. 1 lowercase letter
   2. 1 uppercase letter
   3. 1 digit
   4. 1 special symbol (!, @, #, $, %, ^, &, \*, (, ), \_, +, <, >, ?)
4. **E-mail** – Required field. Text is in format **<user>@<host>** where:
   1. **<user>** is a sequence of letters and digits, where '**.**', '**-**' and '**\_**' can appear between them (they cannot appear at the beginning or at the end of the sequence).
   2. **<host>** is a sequence of at least two words, separated by dots '**.**' (dots cannot appear at the beginning or at the end of the sequence)
5. **ProfilePicture** – Image file with size maximum of 1MB
6. **RegisteredOn** – Date and time of user registration
7. **LastTimeLoggedIn** – Date and time of the last time the user logged in
8. **Age** – number in range [1, 120]
9. **IsDeleted** – Shows whether the user is deleted or not

## Friends

Let’s say that the **user can have many friends** that would be again other users (or in other words **many to many self-relationship**).

Make the necessary changes using Code First Migrations. Make sure no data is lost after the update.

### Tasks

1. List **all users** with the count of their friends. Order them by **friends count** (**descending**) and then by **name** (**ascending**). Print the **name**, the **number of friends** and the **status** of each user. If the user is **not deleted**, their **status** is **Active**, **otherwise** their status is **Inactive**.
2. List all **active** users (**not** deleted ones) with **more than 5 friends**. Order users by **registration date** (**ascending**) and by **friends count** (**descending**). Print the **name**, the **number of friends** and the **period** (in **days**) the user has been part of the network (the **difference** between **current date** and the **registration** **date**).

## Albums

Previously 1 user could upload only 1 picture (just their profile picture). Now each user can create **personal albums**. Each album has **name**, **background color** and **information** whether is **public** or **not**. Each **picture** has **title**, **caption** and **path** on the file system. **An album** can contain **many pictures** and **one picture** can be present in **many albums**. **Each user** can have **many albums** but **an album** can have **only one** owner **user**.

Make the necessary changes using Code First Migrations. Make sure no data is lost after the update.

### Tasks

1. List **all albums** with the **name** of their **owner** and the **count** of their **pictures**. Print album **title**, owner **name** and pictures **count**. Order them by the **number of pictures** (**descending**) and then by the **name of the owner** (**ascending**).
2. List the **pictures**, that are **included** in **more than 2 albums**. Print the picture **title**, the **names of the albums** and the **names of the owners** of the albums. Order pictures by **number of albums** (**descending**) and then by the **title** (**ascending**).
3. List **all albums** of a given user id with information about the pictures in each album. Print the name of the user. If the album is **public**, print its **name** and the **titles** of the pictures in it with the **path** for each picture. If the album is **not public**, print **only** the **name** of the album and the **message** "Private content!". Order the albums by **name** (**ascending**).

## Tags

Imagine how much cooler would be if the user can put tags on each album so they can be easily organized (such as, #NewYear2016, #HolidaySummer, #NoMakeup etc…). A tag is just simply a string without any spaces. Each album can have as many tags the user wants and each tag can be placed on unlimited number of albums.

Make a [Tag] attribute that would validate if the given string is valid tag. A valid tag is a string starting with pound sign (#), do not contain any spaces in it and is no more than 20 symbols long.

Write a static class TagTransofrmer that would have a single public static method inside Transform(string tag). That method would convert given tag to a valid one (remove all spaces, put pound sign at first position if it is not present and reduce the length of the tag if it is more than 20 symbols).

Write a program that receives as an input tags and insert them into the database. Use the [Tag] attribute and TagTransformer class to make sure only valid attributes are inserted in the database.

### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| #summer | #summer was added to database |
| myCat | #myCat was added to database |
| #no make up | #nomakeup was added to database |
| #aaaaaaaaaaaaaaaaaaXCutThisEnd | #aaaaaaaaaaaaaaaaaaX was added to database |
| me and my bff doing selfie | #meandmybffdoingself was added to database |

### Tasks

1. List **all albums** with a given tag. Print the **title** of each album and the **name** of its **owner**. Order albums by **number of tags** (**descending**) and then by **name** (**ascending**).
2. List **all users** that have **albums** with **more than 3 tags** with the **albums** that **have more than 3 tags**. Print the **name** of the user, the **title** of the album and list the **tags** for each album. Order the users by **number** **of** **albums** with more than 3 tags (**descending**), then by **number of tags**(**descending**) and then by **name** (**ascending**).

==========================================================================================

## Shared Albums

Currently an album can have just one owner lets modify it so the user can share its albums with other users. To do that just change the type of the relationship between user and album from one-to-many to more appropriate one.

Make the necessary changes using Code First Migrations. Make sure no data is lost after the update.

### Tasks

1. List **all users** with the **users** they have **shared** albums with. Print the **name** of the user, the **names** of their **friends** and the **names** of **all albums** shared with that friend. Order users by **name** (**ascending**).
2. List **all albums**, shared with **more than 2 people**. Print the **name** of the album, the **number of people** and the information whether the album is **public or not**. Order the albums by **number of people**(**descending**) and by **name** (**ascending**).
3. List **all the albums**, shared with a **user** with a **given name**. Print the **names** of the albums and the **count of** the **pictures** in each album. Order the albums by the **number of pictures** (**descending**) and then by **name** (**ascending**).

## User Roles

Right now, if some user share album with a friend for example, their friend has total control over the shared album. That means they can add or delete photos without the permission of the initial owner of the album. To restrict that we can set role for each user for given album. The roles should be:

* **Owner** - can modify the album
* **Viewer** - can only see the pictures in that album but cannot add or delete any

Make the necessary changes using Code First Migrations. Make sure no data is lost after the update.

### Tasks

1. List **all albums** with their **users.** Print the **name** of the album and the **name** of each user. For each user print information whether they are **owner** or **viewer** of the album. Order the albums by **name** of the **owner** (**ascending**) and then by the **count of the viewers** (**descending**).
2. Find a **user** with **a given name**. Print the **number of albums** they are **owners** of and the **number of albums** they are **viewers of**.
3. List **all the users** that are viewers of **at least 1 album**. Print the **name of the user** and the **number of public albums** they **can** view.

# Football Betting

## Database Models

Your task is to create a database for the **Football Bookmaker System**, using the **Entity Framework Code First** approach. Model the following tables:

* **Teams** – Id, Name, Logo, 3 letter Initials (JUV, LIV, ARS…), Primary Kit Color, Secondary Kit Color, Town, Budget
* **Colors** – Id, Name
* **Towns** – Id, Name, Country
* **Countries** – Id (3 letters – for example BUL, USA, GER, FRA, ITA…), Name, Continent
* **Continents** – Id, Name
* **Players** – Id, Name, Squad Number, Team, Position, Is Currently Injured
* **Position** – Id (2 letters – GK, DF, MF, FW…), position description (for example – goal keeper, defender…)
* **PlayerStatistics** – Game, Player, Scored Goals, Player Assists, Played Minutes During Game, (PK = Game + Player)
* **Games** – Id, Home Team, Away Team, Home Goals, Away Goals, Date and Time of Game, Home team Win bet rate, Away Team Win Bet Rate, Draw Game Bet Rate, Round, Competition)
* **Rounds** – Id, Name (for example Groups, League, 1/8 Final, 1/4 Final, Semi-Final, Final…)
* **Competitions** – Id, Name, Type (local, national, international)
* **CompetitionTypes** –Id, Name
* **BetGame** – Game, Bet, Result Prediction (PK = Game + Bet)
* **Bets** – Id, Bet Money, Date and Time of Bet, User
* **ResultPrediction** – Id, Prediction (possible values - Home Team Win, Draw Game, Away Team Win)
* **Users** – Id, Username, Password, Email, Full Name, Balance

## Table relationships

* Team has one primary kit color and one secondary kit color
* Team resident in one town
* Each town can host several teams
* Town can be placed in one country and a country can have many towns
* Country can be placed in several continents and a continent can have many countries
* Player can play for one team and one team can have many players that play for it
* Player can play at one position and one position can be played by many players
* Player can play in many games and in each game, many players take part
* Additionally, for each player for given game is kept statistics such as scored goals, goal assists and minutes played during given game
* A game can be played in one round and in one round many games can be played
* A game can be played in one competition and in one competition many games can be played
* On a game, many bets can be placed and one bet can be on several games
* Each bet for given game must have prediction result
* A bet can be placed by only one user and one user can place many bets

Add **navigational properties** in all models to simplify navigation. Annotate the data models with the appropriate **attributes** and validations and **enable code first migrations**.

Separate the **models**, **data** and **client** into **different layers** (projects).

### Hint - Database Schema



# Bank System

## Bank System Database

Your task is to create a database for **Bank System,** using the **Entity Framework Code First** approach. In the database, we should keep information about users and their banking accounts.

* **User** – username (**required**), password (**required**), email address (**required**)

There are two types of bank accounts:

* **Saving account** – account number, balance, interest rate
* **Checking account** – account number, balance, fee

**Each** user can have **many** Savings accounts and **many** Checking accounts.

The **operations** that can be performed with those accounts are:

* **Savings account** – deposit money, withdraw money, add interest
* **Checking account** – deposit money, withdraw money, deduct fee

## Bank System Console Client

Extend the database from the previous exercise to support keeping information about **users**. A user has **username, password, email** and can have **many bank accounts**. Design a console application that uses that database and support the following commands:

Commands that can be executed when there is **no currently logged in user**:

* Register <username> <password> <email> - That command add new user to the database in case username, password and email are valid. Otherwise print appropriate message informing why the user cannot be registered. The requirements for valid parameters are:
  + **Username** – can contain only letters [a-Z] and numbers. Cannot start with number. Cannot be less than 3 symbols long
  + **Password** – must contain at least 1 lowercase letter, 1 uppercase letter and 1 digit. Also, must be more than 6 symbols long
  + **Email** – must be in format **<user>@<host>** where:
    - **<user>** is a sequence of letters and digits, where '**.**', '**-**' and '**\_**' can appear between them.
    - **<host>** is a sequence of at least two words, separated by dots '**.**'. Each word is sequence of letters and can have hyphens '**-**' between the letters.
* Login <username> <password> - That command set the current logged in user if exists. Otherwise print appropriate message.

Commands that can be executed when there is **currently logged in user**:

* Logout – log out the user from the system. If there is no logged in user print appropriate message.
* Add SavingAccount <initial balance> <interest rate> - add saving account to the currently logged in user. Also, set the account number to random combination of 10 uppercase letters and digits. For example: “PX234ADG56”, “90M09JKE73”, etc.
* Add CheckingAccount <initial balance> <fee> - add checking account to the currently logged in user. Also, set the account number to random combination of 10 uppercase letters and digits.
* ListAccounts – prints a list of overall information for all accounts of currently logged in user in format:

Saving Accounts:

--{Account Number} {Current Balance}  
Checking Accounts:  
--{Account Number} {Current Balance}

Order them **by account number ascending**.

* Deposit <Account number> <money> - adds money to the account with given number
* Withdraw <Account number> <money> - subtracts money from the account with given number
* DeductFee <Account number> - deduct the fee from the balance of the account with given number
* AddInterest <Account number> - add interest to the balance of the account with given number

After each command **print appropriate message** telling whether the command was successfully executed or not. If it is not print appropriate message telling what was the error. Use all of the **best practices** in programming and **suitable** **design patterns**.

### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| Register vl Tsepesh89 vlad@rom.ro  Register vlad123 tspesh vlad@rom.ro  Register vlad123 Tsepesh89 -v-@-rom.ro  Register vlad123 Tsepesh89 vlad@rom.ro  Logout  Login vlad321 Tsepesh89  Login vlad123 smallPussyCat  Login vlad123 Tsepesh89  Add SavingsAccount 1000 0.2  Add CheckingAccount 100 4.20  Deposit A8234JDG9M 10.42  Withdraw A8234JDG9M 5  Deposit PO8FHH34GM 200  Withdraw PO8FHH34GM 45.2  AddInterest A8234JDG9M  DeductFee PO8FHH34GM  ListAccounts  Logout | Incorrect username  Incorrect password  Incorrect email  vlad123 was registered in the system  Cannot log out. No user was logged in.  Incorrect username / password  Incorrect username / password  Succesfully logged in vlad123  Succesfully added account with number A8234JDG9M  Succesfully added account with number PO8FHH34GM  Account A8234JDG9M has balance of 1010.42  Account A8234JDG9M has balance of 1005.42  Account PO8FHH34GM has balance of 300.00  Account PO8FHH34GM has balance of 254.80  Added interest to A8234JDG9M. Current balance: 1206.50  Deducted fee of PO8FHH34GM. Current balance: 250.60  Accounts for user vlad123  Saving Accounts:  --A8234JDG9M 1206.50  Checking Accounts:  --PO8FHH34GM 250.60  User vlad123 successfully logged out |