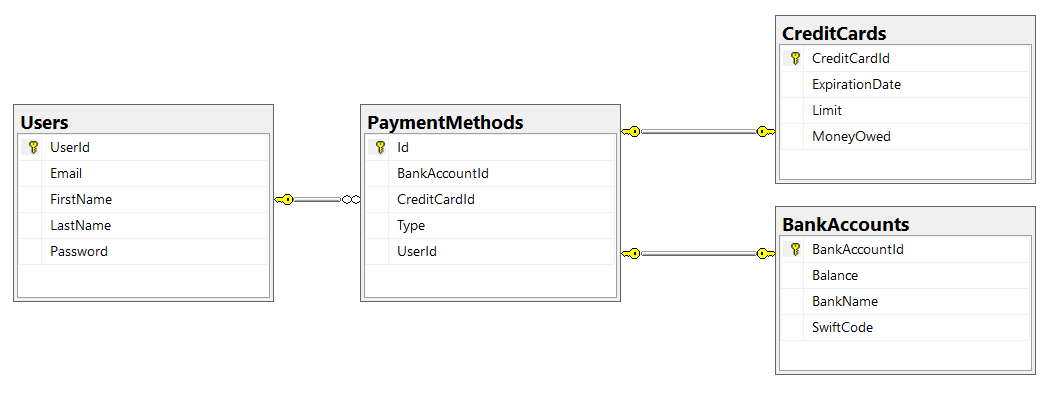
# Exercises: Advanced Relations

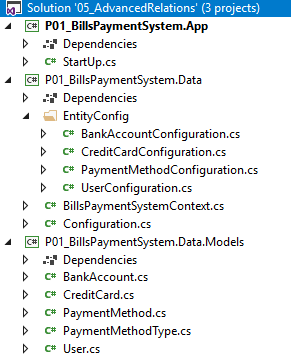
This document defines the **exercise assignments** for the ["Databases Advanced – EF Core" course @ Software University](https://softuni.bg/trainings/1741/databases-advanced-entity-framework-october-2017).

## Bills Payment System

Your task is to create a database for **Bills Payment System**, using the **Code First** approach. In the database, we should keep information about the **users** (**first name, last name, email, password, payment methods**). Every **payment method** should have an **id**, an **owner**, a **type** and a **credit** **card** or a **bank** **account** connected to it. There are **two types** of billing details – **credit card** and **bank account**. The credit card has **expiration date**, a **limit** andanamountof **money owed**. The **bank** **account** has a **balance**, a **bank name** and a **SWIFT** **code**.



Create the configuration of each model in a new class, implementing the IEntityTypeConfiguration interface. Your solution should look similar to this:



### Constraints

Your **namespaces** should be:

* P01\_BillsPaymentSystem – for your Startup class, if you have one
* P01\_BillsPaymentSystem.Data – for your DbContext
* P01\_BillsPaymentSystem.Data.Models – for your models

Your **models** should be:

* BillsPaymentSystemContext – your DbContext
* User:
  + UserId
  + FirstName (up to 50 characters, unicode)
  + LastName (up to 50 characters, unicode)
  + Email (up to 80 characters, non-unicode)
  + Password (up to 25 characters, non-unicode)
* CreditCard:
  + CreditCardId
  + Limit
  + MoneyOwed
  + LimitLeft (calculated property, not included in the database)
  + ExpirationDate
* BankAccount:
  + BankAccountId
  + Balance
  + BankName (up to 50 characters, unicode)
  + SWIFT Code (up to 20 characters, non-unicode)
* PaymentMethod:
  + Id - PK
  + Type – enum (BankAccount, CreditCard)
  + UserId
  + BankAccountId
  + CreditCardId

**Everything** is required! Only **PaymentMethod**’s **BankAccountId** and **CreditCardId** should be **nullable**,and you should make sure that always **one** of them **is** **null** and the **other** **one** is **not** (add a **CHECK** constraint).

Make sure that **every** **record** in the **PaymentMethods** table has a unique combination of **UserId**, **BankAccountId** and **CreditCardId**!

## Seed Some Data

Make a **Seed**() method to seed some data into the **BillsPaymentSystem** database.

## User Details

Create a **console** **app** that retrieves from the database a **user** and all of his **payment** **methods** by a given **user id**, and prints them on the console in the format:

|  |
| --- |
| User: Guy Gilbert  Bank Accounts:  -- ID: 1  --- Balance: 2000.00  --- Bank: Unicredit Bulbank  --- SWIFT: UNCRBGSF  -- ID: 2  --- Balance: 1000.00  --- Bank: First Investment Bank  --- SWIFT: FINVBGSF  Credit Cards:  -- ID: 1  --- Limit: 800.00  --- Money Owed: 100.00  --- Limit Left:: 700.00  --- Expiration Date: 2020/03 |

First, list the user’s **bank** **accounts** and then – his **credit** **cards**. If **no** such **user** exist, print "User with id {**userId**} not found!" instead.

## Pay Bills

Add **Withdraw**() and **Deposit**() methods to the **BankAccount** and **CreditCard** classes, and make sure they are the only way you can change the **Balance**, **MoneyOwed** and **Limit** properties. Then create a **PayBills**(int userId, decimal amount) method that uses all of a user’s payment methods to pay his bills. Start with his **bank** **accounts**, ordered by id, and then his **credit** **cards**, ordered by **id**. If the user doesn’t have enough money available, don’t withdraw anything and print "Insufficient funds!" to the console.

## Users

Create table Users. Users should have: **UserId**, **Username, Password, FirstName, LastName, Email.**

## Friends

Now in that and the next several tasks we are going to extend the database where we created the table User from the previous exercise. 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.

## Albums

Each user is capable of creating **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.

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

## Tag Attribute

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 |

## 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.

## \*User Roles

Right now, if some user share album with a friend for example. His friend has total control over his/her album. That means he 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.

## Bills Payment System

Your task is to create a database for **Bills Payment System**, using the **Entity Framework Code First** approach. In the database, we should keep information about the **users** who are using that system (**first name, last name, email, password, billing details**). Every **billing detail** have **number** and **owner**. Also, there are **two types** of billing details **credit card** and **bank account**. The credit card has **card type, expiration month, expiration year**. And the bank account has **bank name** and **SWIFT** **code**.

**Solve the task**. Use the following approach to make model of the classes and the database tables.

* Table per Hierarchy

Add **navigational properties** in all models to simplify navigation.

## Create Database for BookShopSystem using Code First

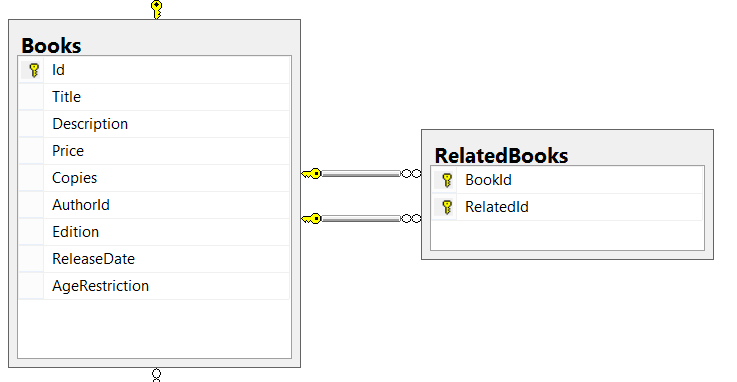
## 

A book shop keeps **books**. A book can have one **author** and many **categories**. Let's create a class for each of the main tables.

## RelatedBooks

Let's say at one point we decide that **books** should have **related books** - i.e. a book has many related books and each related book has related books as well.

Go to the **Book** class and add a **navigational property** **RelatedBooks**(collection of releted books). Make sure you **instantiate** it in the constructor.



Keep in mind that your column and table names may differ from the ones in the above picture.