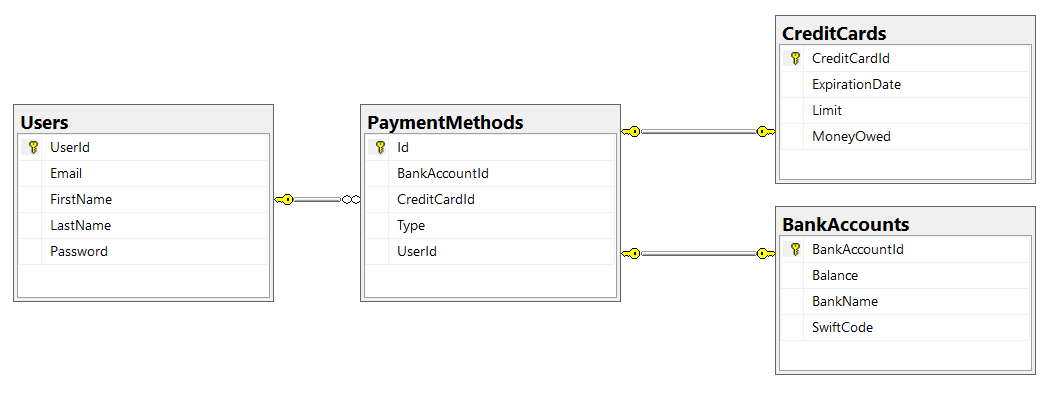
# Exercises: Advanced Relations

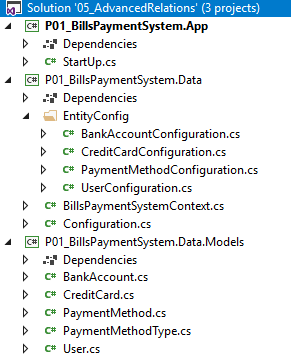
This document defines the **exercise assignments** for the ["Databases Advanced – EF Core" course @ Software University](https://softuni.bg/trainings/1741/databases-advanced-entity-framework-october-2017).

## 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.

## Vehicles

Your task is to create a database for **Vehicles Info System,** using the **Entity Framework Code First** approach. In the database, we should keep information about different kind of vehicles. Each vehicle has **manufacturer**, **model**, **price** and **max speed**. There are two main types of vehicles: **motor** and **non-motor vehicles**. There is only one type of non-motor vehicles – **bike**. Bike has **shifts count** and **color**. All motor vehicles have **number of engines**, **engine type** and **tank capacity**. There are several types of motor vehicles:

* **Car** – number of doors, information about having insurance
* **Train** – locomotive, number of carriages, list of carriages
* **Plane** – airline owner, color, passengers’ capacity
* **Ship** – nationality, captain name, size of ship crew
  + **Cargo Ship** – max load kilograms
  + **Cruise Ship** – passengers’ capacity

All carriages have passengers’ seats capacity. There are three types of carriages

* **Passenger** – standing passengers capacity
* **Restaurant** – tables count
* **Sleeping** – beds count

Locomotive has **model** and **power**. Each locomotive can pull one train and one train can be pulled only by one locomotive.

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

## Bank System

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 banking accounts. There are two types of bank accounts:

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

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 |

## \*\*\*Bank System GUI

Use the logic and database of the previous exercise and replace the console client with graphical user interface with technology of your choice.

## 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.

* **Book** - id, title (between 1 ... 50 symbols - inclusive), description (optional, up to 1000 symbols), edition type   
  (**Normal**, **Promo** or **Gold**), price, copies (number), release date (optional)
* **Author** - id, first name (optional) and last name
* **Category** - id, name

Assume everything **not market optional** is mandatory. The **classes** should describe with **properties** each of the **table columns**.

For the book edition type you could use enumeration:

Add constraints as described above. Do the same for the **Book** and **Category** models. Make sure you add **navigation properties** when there are relations.

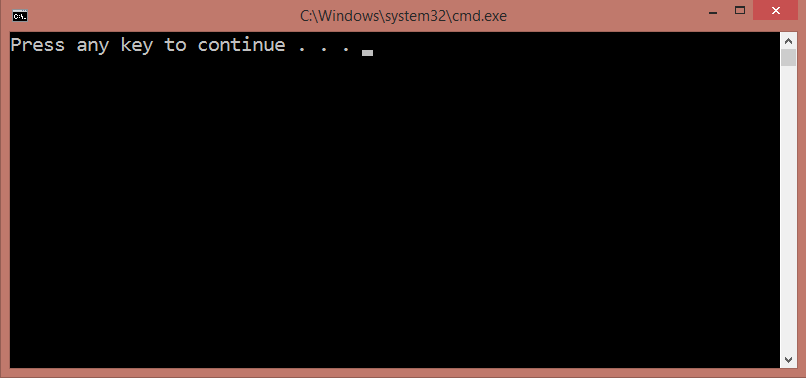
**Important** **Note**: Make **navigation properties** **virtual**, so Entity Framework can **override** them and apply **lazy loading**.

## 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.

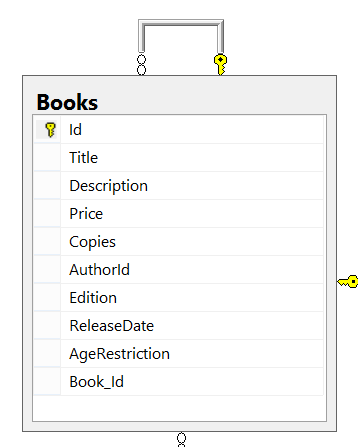
Go to the **Book** class and add a **navigational property RelatedBooks**(collection of releted books). Make sure you **instantiate** it in the constructor.

Start the program again.



No errors. The migration has most likely been successful and we should see a many-to-many self-reference in the database schema.

However, that's not the case - the migration is successful but a book can have only 1 related book.

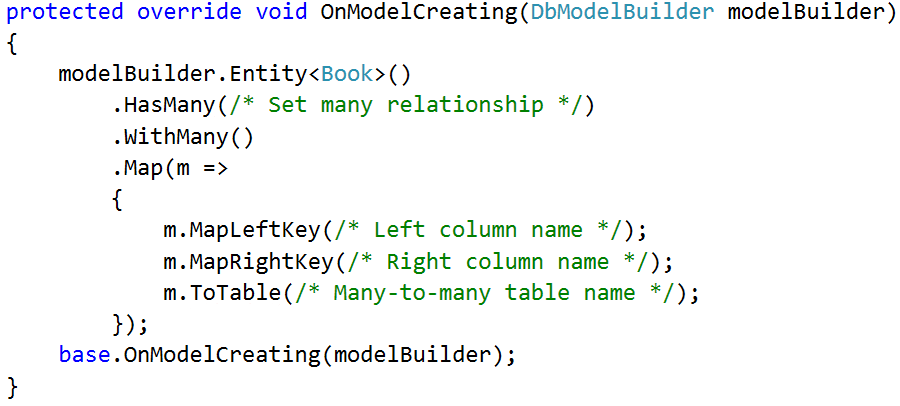


This is one of those cases where Entity Framework does not correctly update the DB schema after changes have been made to the model.

In such events, we need to manually configure the relationship with the so-called **Model Builder**. Go to the **BookShopContext** class and override the **OnModelCreating()** method.

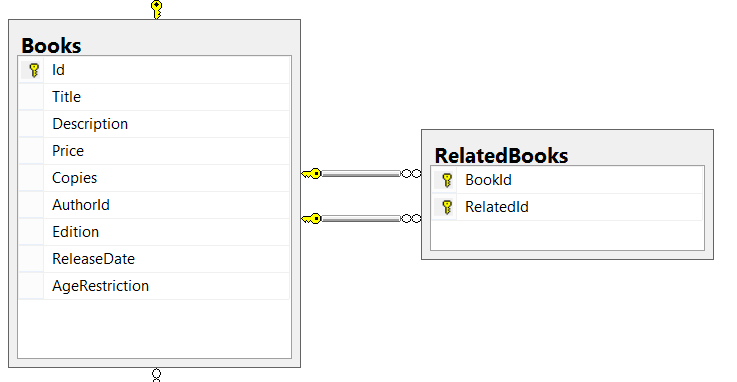
This method is called only once when entity framework starts for the first time. It receives a **DbModelBuilder** as argument. We will use that model builder to configure the desired relationship.

The Model Builder (also **fluent API**) will allow us to tell EF that a **Book** has **many books** (the related books) and those books have many as well. Then we will map that relationship to a **junction** (**many-to-many**) table where we manually set the **left key** (book id) and **right key** (related book id). Finally, we **name the table**.



At the end, we call the **base** implementation of the method (it contains native EF configurations, so it's best we call it at the end).

Restart the program and if no exception is thrown, the new DB schema should now look as follows:



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

And finally, let's test the **RelatedBooks** functionality it. **Query 3 books** from the database and set them as **related**.

|  |  |
| --- | --- |
| **Sample Code** | **Sample Output** |
| var books = context.Books  .Take(3)  .ToList();  books[0].RelatedBooks.Add(books[1]);  books[1].RelatedBooks.Add(books[0]);  books[0].RelatedBooks.Add(books[2]);  books[2].RelatedBooks.Add(books[0]);  context.SaveChanges();  // **TODO:**  // **Query the first three books to get their names**  // **and their related book names**  foreach (var book in booksFromQuery)  {  Console.WriteLine("--{0}", book.Title);  foreach (var relatedBook in book.RelatedBooks)  {  Console.WriteLine(relatedBook);  }  } | --Absalom  A che punto A" la notte  After Many a Summer Dies the Swan  --A che punto A" la notte  Absalom  --After Many a Summer Dies the Swan  Absalom |