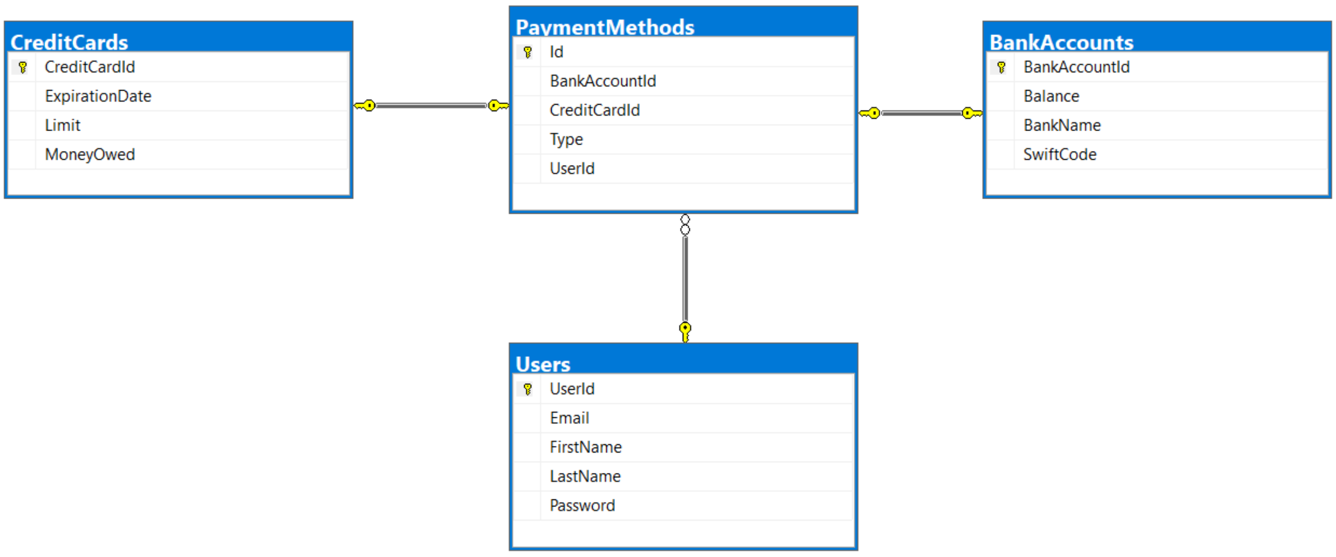
Exercises: Advanced Relations

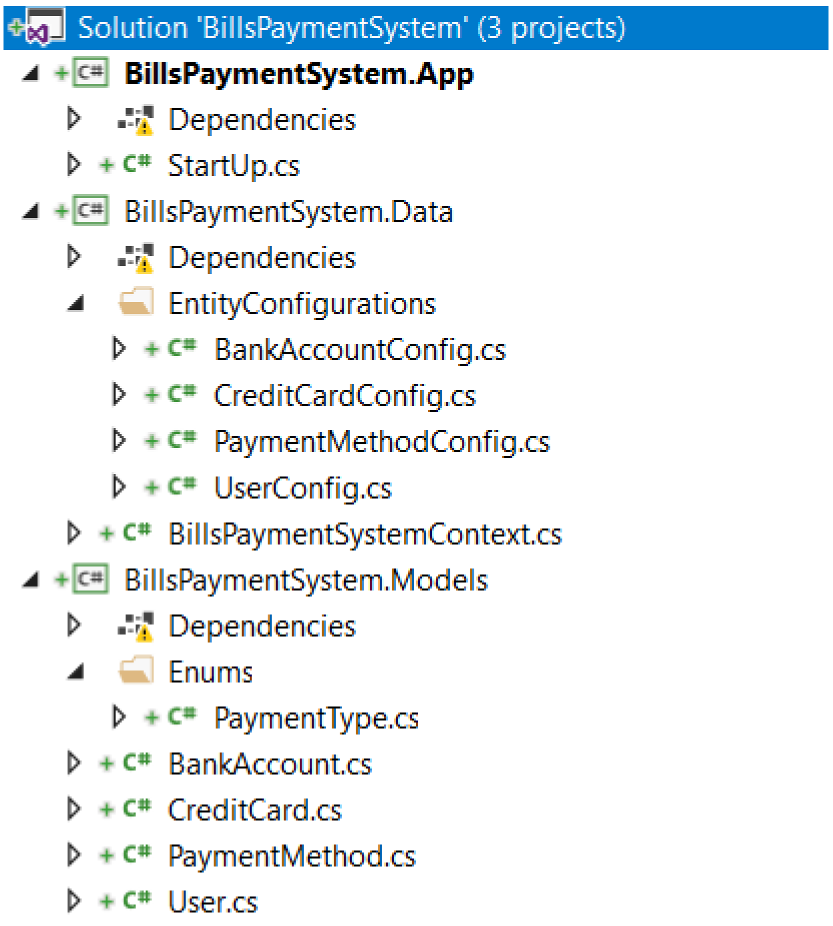
This document defines the **exercise assignments** for the "Databases Advanced – EF Core" course @ Software University.

1. 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** and an amount of **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:


Solution 'BillsPaymentSystem' (3 projects)
- BillsPaymentSystem.App
  - Dependencies
  - StartUp.cs
- BillsPaymentSystem.Data
  - Dependencies
  - EntityConfigurations
    - BankAccountConfig.cs
    - CreditCardConfig.cs
    - PaymentMethodConfig.cs
    - UserConfig.cs
  - BillsPaymentSystemContext.cs
- BillsPaymentSystem.Models
  - Dependencies
  - Enums
    - PaymentType.cs
  - BankAccount.cs
  - CreditCard.cs
  - PaymentMethod.cs
  - User.cs

Constraints

Your **namespaces** should be:

- **`BillsPaymentSystem.App`** – for your Startup class, if you have one
- **`BillsPaymentSystem.Data`** – for your DbContext
- **`BillsPaymentSystem.Models`** – for your models

Your **models** should be:

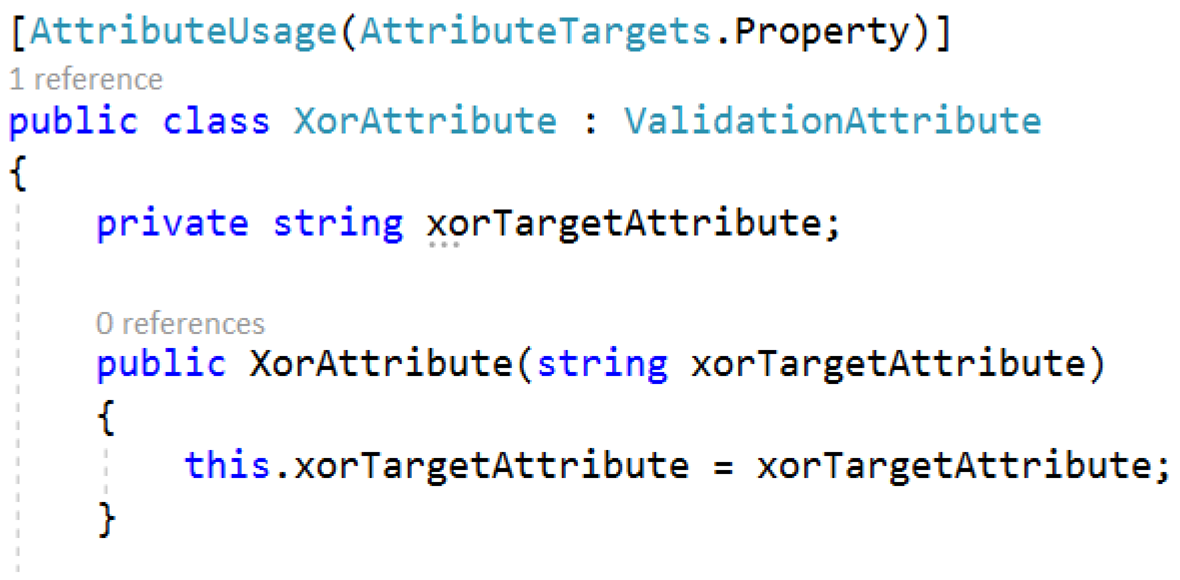
- **`User`**:
  - UserId
  - FirstName (up to 50 characters, unicode)
  - LastName (up to 50 characters, unicode)
  - Email (up to 80 characters, non-unicode)
  - Password (up to 25 characters, non-unicode)
- **`CreditCard`**:
  - CreditCardId
  - Limit
  - MoneyOwed
  - LimitLeft (calculated property, not included in the database)
  - ExpirationDate
- **`BankAccount`**:
  - BankAccountId
  - Balance

© Software University Foundation. This work is licensed under the CC-BY-NC-SA license.
Follow us:

    - 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 **custom attribute** constraint).

Create a new class **XorAttribute** which has attribute ussage and its targets are properties.



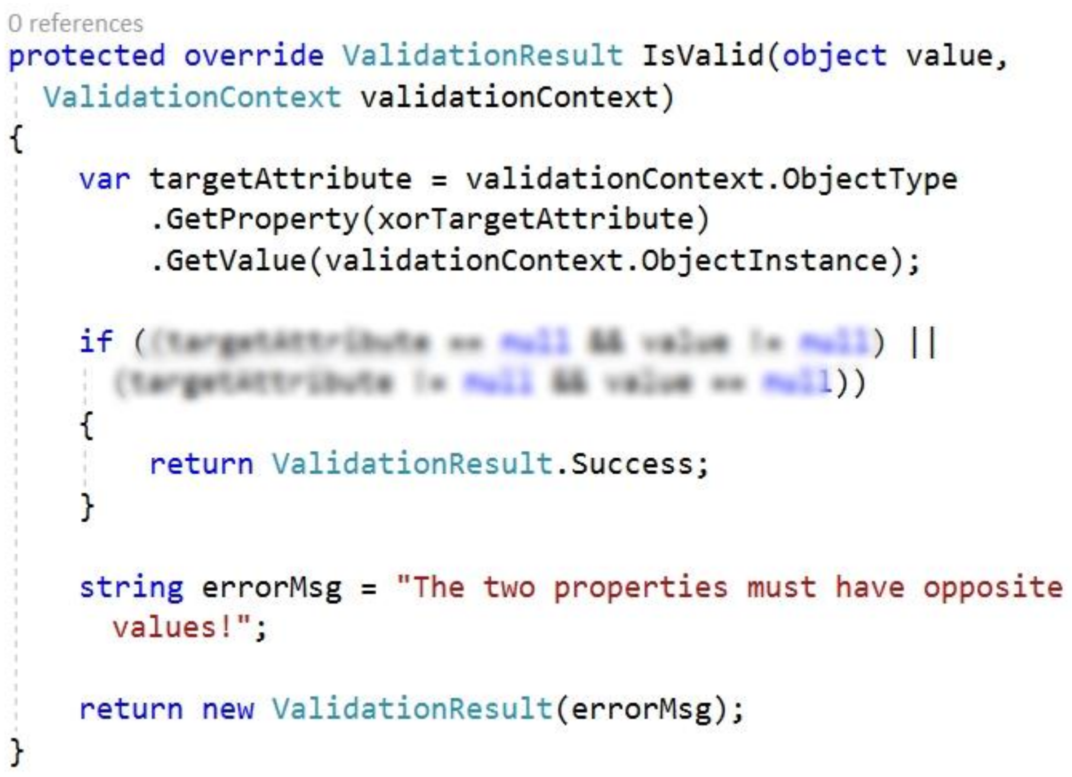
```
[AttributeUsage(AttributeTargets.Property)]
1 reference
public class XorAttribute : ValidationAttribute
{
    private string xorTargetAttribute;

    0 references
    public XorAttribute(string xorTargetAttribute)
    {
        this.xorTargetAttribute = xorTargetAttribute;
    }
```

To be an attribute the class must inherit **ValidationAttribute** class and override its **IsValid** method and in this method you should check that always **one** of them **is null** and the **other one** is **not** . If the condition is valid you shout return success validation result otherwise return an error message **"`The two properties must have opposite values!`"**

© Software University Foundation. This work is licensed under the CC-BY-NC-SA license.

Follow us:



```
0 references
protected override ValidationResult IsValid(object value,
  ValidationContext validationContext)
{
    var targetAttribute = validationContext.ObjectType
        .GetProperty(xorTargetAttribute)
        .GetValue(validationContext.ObjectInstance);

    if ([redacted]) ||
      ([redacted]))
    {
        return ValidationResult.Success;
    }

    string errorMsg = "The two properties must have opposite
      values!";

    return new ValidationResult(errorMsg);
}
```

2. Seed Some Data

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

3. 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.

© Software University Foundation. This work is licensed under the CC-BY-NC-SA license.

Follow us:

4. Pay Bills

Add **Withdraw**(int userId, decimal amount) and **Deposit**(int userId, decimal amount) methods. 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.

© Software University Foundation. This work is licensed under the CC-BY-NC-SA license.

Follow us: