# Homework: Entity Framework

This document defines the homework assignments from the ["Database Applications" Course @ Software University](https://softuni.bg/trainings/21/Database-Applications-Mar-2015). Please submit as homework a single zip / rar / 7z archive holding the solutions (source code) of all below described problems.

## News Database (Code First)

Using Entity Framework Code First, create a database "NewsDB" to hold a table "News". News should hold only a single text field – the **news content**. **Seed** the database with a news item holding some text.

## Concurrent Updates (Console App)

Write a **console app** that edits the first news item in the DB. It should **detect and** **handle any concurrent updates**.

1. At startup, the app should **load from the DB the news text** and print it on the console.
2. After that, the app should **enter a new value** for the news text from the console.
3. After entering a new value, the app should **try to save it to the DB**.
   * In case of **success** (no conflicting updates), the app should say that the changes were saved and should finish its work.
   * In case of **concurrent update conflict**, the app should display an **error message**, should display the new (changed) text from the DB and should go to **Step 2**.

Run simultaneously two instances of your app to simulate **two concurrent users**. Make an update conflict in the database and handle it. The console of the first and the second user might look as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **First User** | |  | **Second User** | |
| Application started.  Text from DB: *EF 7 Beta To Be Released in May 2016*.  Enter the corrected text: | | Application started.  Text from DB: *EF 7 Beta To Be Released in May 2016*.  Enter the corrected text: | |
| User input: | *EF 7 To Be Delayed* |
| Changes successfully saved in the DB. | |
|  | | User input: | *EF 7 Beta To Be Released in June 2016* |
| **Conflict!** Text from DB: *EF 7 Beta To Be Delayed*. Enter the corrected text: | |
| User input: | *EF 7 Beta To Be Delayed Or Released in June 2016?* |
| Changes successfully saved in the DB. | |

## \* Concurrent Updates (GUI App)

Create a graphical user interface (GUI) for the previous application. It should allow editing the first news item in the DB and should **detect and** **handle any concurrent updates**. You may use Windows Forms, Windows Presentation Foundation (WPF) or any other GUI toolkit for C#.

The application screens are shown on the UI prototype below:

|  |  |
| --- | --- |
| Main screen:   * **[Edit]** 🡪 opens the first news item from the DB in the editor screen. * **[Exit]** button 🡪 exits from the application.   Editor screen:   * **[Save]** 🡪 saves the modified text from the editor screen to the database. If the same record in the database was changed in the meantime (concurrent update), display an error message. * **[Cancel]** button 🡪 cancels the changes made in the editor (forgets unsaved changes).   Success screen:   * **[OK]** button 🡪 returns to the main screen.   Error screen:   * **[OK]** button 🡪 returns to the main screen. |  |

## ATM Database

Suppose you are creating a simple engine for an **ATM machine**. Create a new database "**ATM**" in SQL Server to hold the accounts of the cardholders and the balance (money) for each account. Add a new table CardAccounts with the following fields:

* Id – int
* CardNumber – char(10)
* CardPIN – char(4)
* CardCash – money

Add a few sample records in the table. Submit as solution the SQL script for your database.

## Transactional ATM Withdrawal

Using **transactional logic** in Entity Framework write a method that **withdraws money** (for example $200) from given account. The withdrawal is successful when the following steps are completed successfully:

1. Check if the given CardPIN and CardNumber are valid. Throw an exception if not.
2. Check if the amount on the account (CardCash) is bigger than the requested sum (in our example $200). Throw an exception if not.
3. The ATM machine pays the required sum (e.g. $200) and stores in the table CardAccounts the new amount (CardCash = CardCash - 200).

Put the above steps in **explicit transaction** that is started before the first step and is committed after the last step. Think why the isolation level needs to be set to "**repeatable read**".

## ATM Transactions History

Extend the project from the previous exercise and add a new table TransactionHistory with fields (Id, CardNumber, TransactionDate, Amount) holding information about all money withdrawals on all accounts.

Modify the withdrawal logic so that it **preserves history** in the new table after each successful money withdrawal.

## \* Unit Tests for the ATM

Design and implement **unit tests** for the ATM withdrawal operation with 100% code coverage. Your tests should cover all major cases and should assert for each case that the values in the CardAccounts and TransactionHistory tables are correct.