

**ArtNetwork**

**Distributed Systems project**

Student: Aleksandr Ivanov

Supervisor: Andres Käver

Student Code: 186093IADB

Tallinna Tehnikaülikool 2020

# Author’s declaration of originality

I hereby certify that I am the sole author of this thesis and this thesis has not been presented for examination or submitted for defence anywhere else. All used materials, references to the literature and work of others have been cited.

Author: Aleksandr Ivanov

18.02.2020

# Project Description

The goal of this project is to create an international social network for artists, where the community could communicate, share various information, especially their works, and, equally importantly, ensure their career growth.

At the moment, the world has a kind of social network for artists, for example: DeviantArt, Instagram, Tumblr, etc., but they have a sufficient number of shortcomings, and they cannot be called an international network for artists (except Instagram, but there is a claim to the functionality).

From my experience I can notice some things that are significant shortcomings. Here are a few of them: a single language interface, uncomfortable communication, dubious recommendation algorithms, distribution. In turn, correcting or improving these shortcomings, they can become a strategic advantage for my social network.

Particular attention should be paid to artists distribution. In the world there are a lot of artists of various colors, and a enough places where they stay. And this is a problem, because a talented but less popular artist, for example from Russia, may remain unpopular due to various restrictions (language barrier, different interests of the public), and since the popularity of the parameter that brings money is very likely that such talent will be lost.

The result of this project will be a social network where artists from around the world will be in one community

**This is not the final version of the document and will be supplemented as the project develops.**

# Soft-delete and soft-update methods analysis

Since it’s good practice in the modern business IT sector to use data audition also known as soft-delete and soft-update, to track any (or almost) changes in the state of the database and with the ability to use old data, will implement this technique in the project.

I figured out the 3 most suitable approach methods:

* additional column "master\_id"
* composite key with no child affection on master update
* composite key with full child affection

## An additional column "master\_id"

Using an additional column "master\_id" to create a link between different versions of the same record. When a record is created, its identifier is assigned to this field. With subsequent changes, copies of the record are created, to save the current value, the "master\_id" column remains unchanged.

Records have 'dat' column indicates date of deletion (if record is not deleted, it is null. “cat” column is date of record creation (not equal for example registration date or etc)

--create tables

CREATE TABLE Person (

    person\_id   INT             NOT NULL    PRIMARY KEY,

    name        VARCHAR(128)    NOT NULL,

    master\_id   INT             NOT NULL,

    cat         DATETIME2       NOT NULL,

    dat         DATETIME2       NULL

)

When deleting a record, the "dat" column is marked with the date of deletion. Dependent records are also marked with the deletion date.

--soft delete Alex (with marking depended records as deleted)

DECLARE @Alex\_id INT

SET @Alex\_id = 1

UPDATE Person SET dat = @Time2 WHERE person\_id = @Alex\_id

UPDATE Post SET dat = @Time2 WHERE person\_id = @Alex\_id

UPDATE Photo SET dat = @Time2 WHERE person\_id = @Alex\_id

When updating, a copy of the record is created, the current value is saved in it, and the modification time in the "dat" column

--soft update Valya (set name to Valentina) (childs not affected)

DECLARE @Valya\_id INT

SET @Valya\_id = 2

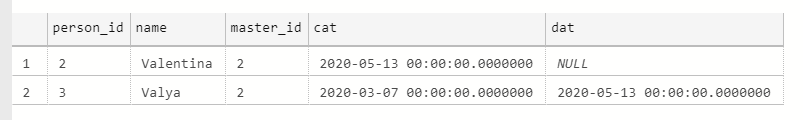
--create copy of current record

INSERT INTO Person(person\_id, name, master\_id, cat, dat)

SELECT 3, name, @Valya\_id, cat, @Time3 FROM Person WHERE person\_id = @Valya\_id

--modify curent record

UPDATE Person SET name = 'Valentina', cat = @Time3 WHERE person\_id = @Valya\_id



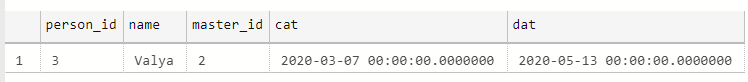
Viewing the history is carried out by a simple query with the required date, however, due to the fact that the dependent records are not updated with the master record, it may not be very obvious which version of the dependent record is valid at that moment.

--select database records that is valid at @Time2

SELECT \* from Person

WHERE   (dat > @Time2 or dat is null)

AND     cat <= @Time2



## Composite key

The second (and third) idea was to use a composite key from the id and dat columns as the primary key. Records have 'dat' column indicates date of deletion (if record is not deleted, it is **'3000-01-01' or other far-far date**). Column 'cat' is date of record creation (not equal for example registration date or etc).

--create tables

CREATE TABLE Person (

    person\_id   INT             NOT NULL,

    dat         DATETIME2       NOT NULL,

    name        VARCHAR(128)    NOT NULL,

    cat         DATETIME2       NOT NULL,

    PRIMARY KEY (person\_id, dat)

)

When a record is deleted, column "dat" is marked with the date it was deleted. This also affects dependent records: firstly, each composite foreign key record needs to be updated, and secondly, also be marked with the date of deletion.

--soft delete Alex (with marking depended records as deleted)

DECLARE @Alex\_id INT

DECLARE @Alex\_dat DATETIME2

SET @Alex\_id = 1

SET @Alex\_dat = '3000-01-01'

UPDATE Person SET dat = @Time2 WHERE person\_id = @Alex\_id AND dat = @Alex\_dat

UPDATE Post SET dat = @Time2, person\_dat = @Time2 WHERE person\_id = @Alex\_id AND person\_dat = @Alex\_dat

UPDATE Photo SET dat = @Time2, person\_dat = @Time2 WHERE person\_id = @Alex\_idAND person\_dat = @Alex\_dat

### Master update not affects child records

In this case, when updating master records, dependent records are not affected.

--soft update Valya (set name to Valentina) (childs not affected)

DECLARE @Valya\_id INT

DECLARE @Valya\_dat DATETIME2

SET @Valya\_id = 2

SET @Valya\_dat = '3000-01-01'

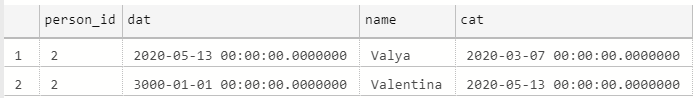
--create copy of current record

INSERT INTO Person(person\_id, name, cat, dat)

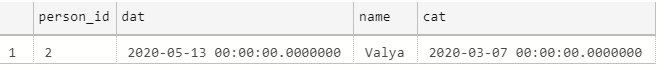
SELECT @Valya\_id, name, cat, @Time3 FROM Person WHERE person\_id = @Valya\_id AND dat = @Valya\_dat

--modify curent record

UPDATE Person SET name = 'Valentina', cat = @Time3 WHERE person\_id = @Valya\_id AND dat = @Valya\_dat



Viewing the history is the same as using “master\_id” column.



### Master update affects child records

This case is more complicated, because when updating a record, you need to update both the composite key of the recording records and create a copy of them (as when updating)

--soft update Valya (set name to Valentina)

DECLARE @Valya\_id INT

DECLARE @Valya\_dat DATETIME2

SET @Valya\_id = 2

SET @Valya\_dat = '3000-01-01'

--#1 soft update Valya's photo

SELECT 'Soft update Valya`s photo'

INSERT INTO Photo (photo\_id, person\_id, person\_dat, [value], cat, dat)

SELECT Ph.photo\_id, P.person\_id, @Time3, Ph.[value], Ph.cat, @Time3 FROM Person P, Photo Ph WHERE P.person\_id = @Valya\_id

AND P.dat = @Valya\_dat

AND Ph.person\_id = P.person\_id

AND Ph.person\_dat = P.dat

UPDATE Photo SET cat = @Time3 WHERE person\_id = @Valya\_id AND person\_dat = @Valya\_dat

--#2 soft update Valya's posts

SELECT 'Soft update Valya`s posts'

INSERT INTO Post (post\_id, person\_id, person\_dat, title, [description], cat, dat)

SELECT Po.post\_id, P.person\_id, @Time3, Po.title, Po.[description], Po.cat, @Time3 FROM Person P, Post Po WHERE P.person\_id = @Valya\_id

AND P.dat = @Valya\_dat

AND Po.person\_id = P.person\_id

AND Po.person\_dat = P.dat

UPDATE Post SET cat = @Time3 WHERE person\_id = @Valya\_id AND person\_dat = @Valya\_dat

SELECT 'Soft delete Valya'

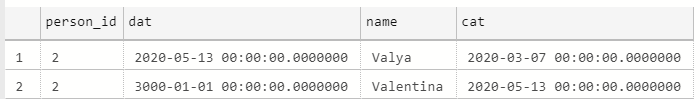
--create copy of current record

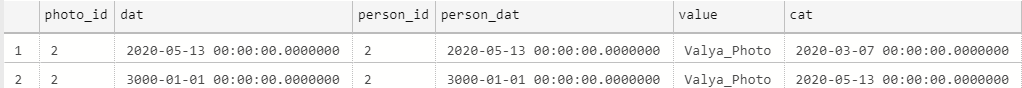
INSERT INTO Person(person\_id, name, cat, dat)

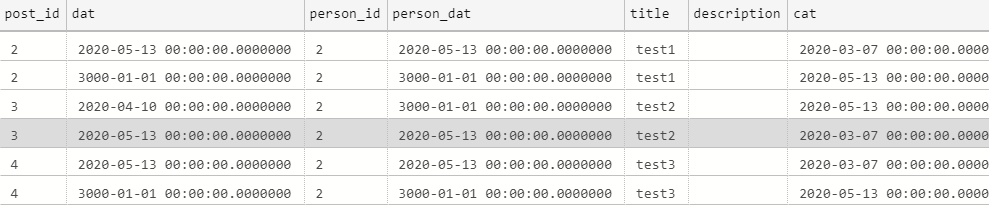
SELECT @Valya\_id, name, cat, @Time3 FROM Person WHERE person\_id = @Valya\_id AND dat = @Valya\_dat

--modify curent record

UPDATE Person SET name = 'Valentina', cat = @Time3 WHERE person\_id = @Valya\_id AND dat = @Valya\_dat

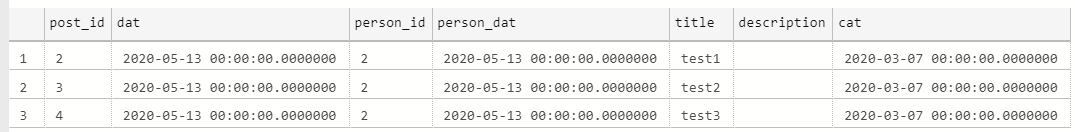


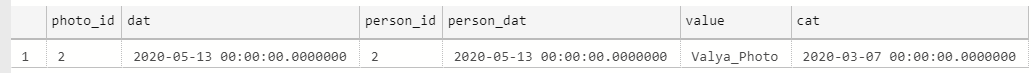




Viewing the history is almost the same as using “master\_id” column. However, in this case, a full copy of the records is created, which allows you to see a complete picture of the state of a given database area at a certain moment







## Making a decision

Each method has both advantages and disadvantages. An additional column "master\_id" is a simple approach, but it does not allow you to see the full picture of the event.

Using a composite key is more difficult than an additional column, since the dependent foreign key entries are also composite, and if you change, you can mistakenly change the key change, which will break the dependency and give rise to the so-called “false data anomaly”. In case children are affected when updating the master record, a complete picture of the state of the region is saved in the database.

However, for this project, I prefer to use an additional column, since this approach is simple, and unlike a composite key, there is no risk of breaking the connection. Also, using a composite key with affecting all the dependent records during the update creates a huge load on the database, so long as the users of my social network can have millions of messages, constantly update the status of profiles, create and edit posts, this load will be critical and will ruin my database and storage.

# Project solution structure

To avoid writing the same code again in the next solution. It is needed to separate our code into 2 major parts - current app specific and common shared base.

## Shared, common codebase:

Contracts.DAL.Base - specs for domain metadata and PK in entities. Specs for common base repository.

DAL.Base - abstract implementations of interfaces for domain.

DAL.Base.EF - implementation of common base repository done in EF.

## App specific codebase for our solution:

Domain - Domain objects - what is our business about

Contracts.DAL.App - specs for repositories

DAL.App.EF - implementation of repositories

# Repository pattern

## Purpose:

It is not a good idea to access the database logic directly in the business logic like it used in **DAO** (Data Access Object) pattern. Tight coupling of the database logic in the business logic make applications tough to test and extend further.

Repository Pattern separates the data access logic and maps it to the entities in the business logic. It works with the domain entities and performs data access logic.

## Pluses:

Using DAO pattern is hard to Business logic to unit test.

Removes duplicate data hard access code throughout the business layer.

## Minuses:

Sometimes not nessesary to just capsulate objects in empty implementation or if project is to simple or project goal it hardly specific, using DAO pattern will only spent your time (and money)

## Current implementation:

**BaseRepo** class represents base implementation of repository using generics, to avoid unnecessary implementations in each database entity repository. **BaseRepo** implements **IBaseRepo** interface to support the Dependency Inversion principle.

Each database entity name has formal view **<entity>Repo** and is derived from **BaseRepo** using entity as generics type. Entity repository also implements **I<entity>Repo** interface to support the Dependency Inversion principle.

**I<entity>Repo** interface implements **IBaseRepo** interface using entity as generics type to support the Dependency Inversion principle

# Diagram

