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

1 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.

2 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

2.1 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_i
d

--modify curent record
UPDATE Person SET name = 'Valentina', cat = @Time3 WHERE person_id = @Valya_i
d
```

	person_id	name	master_id	cat	dat
1	2	Valentina	2	2020-05-13 00:00:00.0000000	*NULL*
2	3	Valya	2	2020-03-07 00:00:00.0000000	2020-05-13 00:00:00.0000000

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
```

	person_id	name	master_id	cat	dat
1	3	Valya	2	2020-03-07 00:00:00.0000000	2020-05-13 00:00:00.0000000

2.2 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_id
AND person_dat = @Alex_dat
```

2.2.1 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 A
ND dat = @Valya_dat

--modify curent record
UPDATE Person SET name = 'Valentina', cat = @Time3 WHERE person_id = @Valya_i
d AND dat = @Valya_dat
```

	person_id	dat	name	cat
1	2	2020-05-13 00:00:00.0000000	Valya	2020-03-07 00:00:00.0000000
2	2	3000-01-01 00:00:00.0000000	Valentina	2020-05-13 00:00:00.0000000

Viewing the history is the same as using “master_id” column.

	person_id	dat	name	cat
1	2	2020-05-13 00:00:00.0000000	Valya	2020-03-07 00:00:00.0000000

2.2.2 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 Pers
on 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 = @V
alya_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 = @Va
lya_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 A
ND dat = @Valya_dat

--modify curent record
UPDATE Person SET name = 'Valentina', cat = @Time3 WHERE person_id = @Valya_i
d AND dat = @Valya_dat
```

	person_id	dat	name	cat
1	2	2020-05-13 00:00:00.0000000	Valya	2020-03-07 00:00:00.0000000
2	2	3000-01-01 00:00:00.0000000	Valentina	2020-05-13 00:00:00.0000000

	photo_id	dat	person_id	person_dat	value	cat
1	2	2020-05-13 00:00:00.0000000	2	2020-05-13 00:00:00.0000000	Valya_Photo	2020-03-07 00:00:00.0000000
2	2	3000-01-01 00:00:00.0000000	2	3000-01-01 00:00:00.0000000	Valya_Photo	2020-05-13 00:00:00.0000000

post_id	dat	person_id	person_dat	title	description	cat
2	2020-05-13 00:00:00.0000000	2	2020-05-13 00:00:00.0000000	test1		2020-03-07 00:00:00.0000
2	3000-01-01 00:00:00.0000000	2	3000-01-01 00:00:00.0000000	test1		2020-05-13 00:00:00.0000
3	2020-04-10 00:00:00.0000000	2	3000-01-01 00:00:00.0000000	test2		2020-05-13 00:00:00.0000
3	2020-05-13 00:00:00.0000000	2	2020-05-13 00:00:00.0000000	test2		2020-03-07 00:00:00.0000
4	2020-05-13 00:00:00.0000000	2	2020-05-13 00:00:00.0000000	test3		2020-03-07 00:00:00.0000
4	3000-01-01 00:00:00.0000000	2	3000-01-01 00:00:00.0000000	test3		2020-05-13 00:00:00.0000

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

	person_id	dat	name	cat
1	2	2020-05-13 00:00:00.0000000	Valya	2020-03-07 00:00:00.0000000

	post_id	dat	person_id	person_dat	title	description	cat
1	2	2020-05-13 00:00:00.0000000	2	2020-05-13 00:00:00.0000000	test1		2020-03-07 00:00:00.0000000
2	3	2020-05-13 00:00:00.0000000	2	2020-05-13 00:00:00.0000000	test2		2020-03-07 00:00:00.0000000
3	4	2020-05-13 00:00:00.0000000	2	2020-05-13 00:00:00.0000000	test3		2020-03-07 00:00:00.0000000

	photo_id	dat	person_id	person_dat	value	cat
1	2	2020-05-13 00:00:00.0000000	2	2020-05-13 00:00:00.0000000	Valya_Photo	2020-03-07 00:00:00.0000000

2.3 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.

3 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.

3.1 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.

3.2 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

4 Repository pattern

4.1 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.

4.2 Pluses:

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

Removes duplicate data hard access code throughout the business layer.

4.3 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)

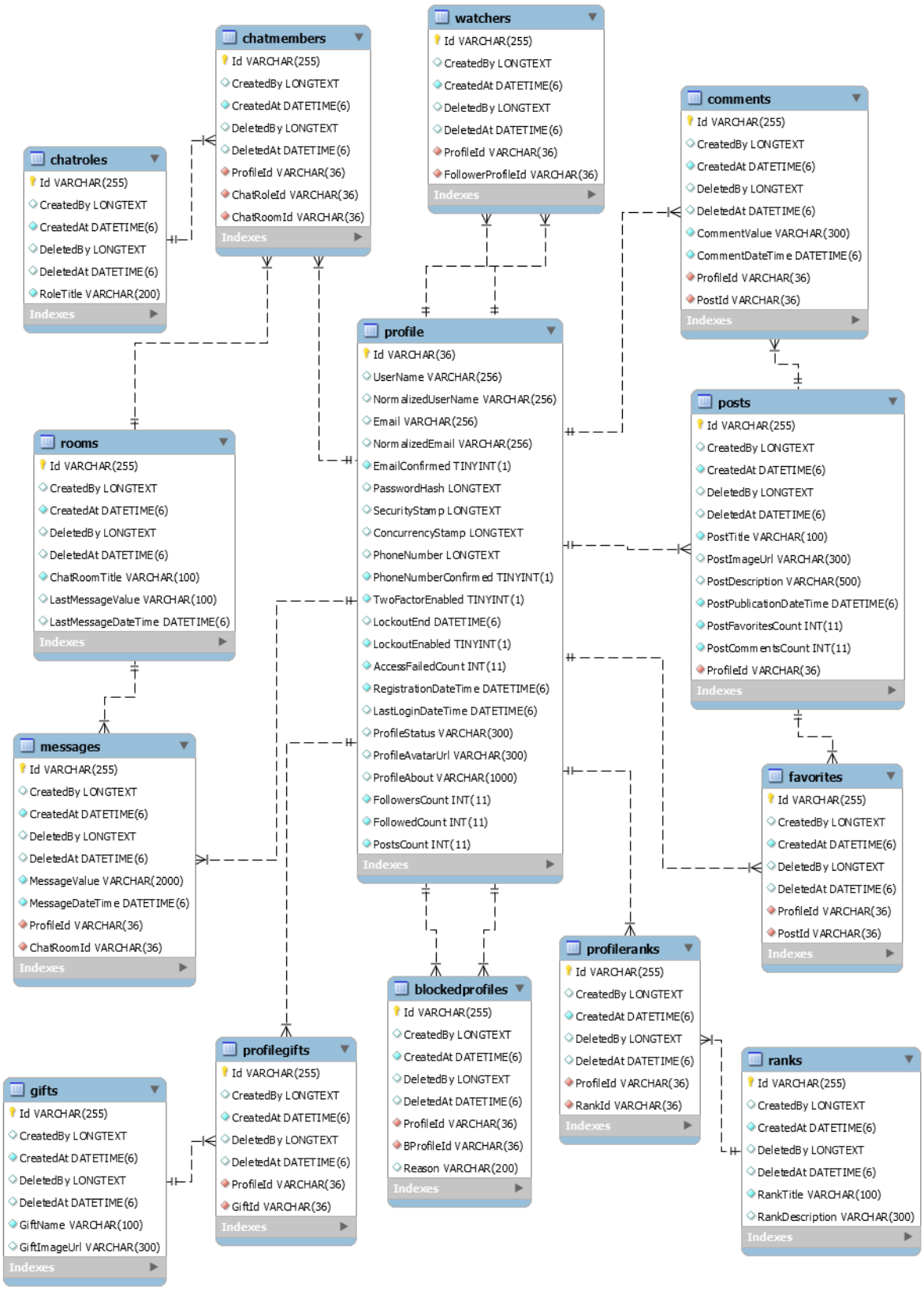
4.4 Current implementation:

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

5 Diagram


chatmembers
Id VARCHAR(255)
CreatedBy LONGTEXT
CreatedAt DATETIME(6)
DeletedBy LONGTEXT
DeletedAt DATETIME(6)
ProfileId VARCHAR(36)
ChatRoleId VARCHAR(36)
ChatRoomId VARCHAR(36)
Indexes
watchers
Id VARCHAR(255)
CreatedBy LONGTEXT
CreatedAt DATETIME(6)
DeletedBy LONGTEXT
DeletedAt DATETIME(6)
ProfileId VARCHAR(36)
FollowerProfileId VARCHAR(36)
Indexes
comments
Id VARCHAR(255)
CreatedBy LONGTEXT
CreatedAt DATETIME(6)
DeletedBy LONGTEXT
DeletedAt DATETIME(6)
CommentValue VARCHAR(300)
CommentDateTime DATETIME(6)
ProfileId VARCHAR(36)
PostId VARCHAR(36)
Indexes
chatroles
Id VARCHAR(255)
CreatedBy LONGTEXT
CreatedAt DATETIME(6)
DeletedBy LONGTEXT
DeletedAt DATETIME(6)
RoleTitle VARCHAR(200)
Indexes
profile
Id VARCHAR(36)
UserName VARCHAR(256)
NormalizedUserName VARCHAR(256)
Email VARCHAR(256)
NormalizedEmail VARCHAR(256)
EmailConfirmed TINYINT(1)
PasswordHash LONGTEXT
SecurityStamp LONGTEXT
ConcurrencyStamp LONGTEXT
PhoneNumber LONGTEXT
PhoneNumberConfirmed TINYINT(1)
TwoFactorEnabled TINYINT(1)
LockoutEnd DATETIME(6)
LockoutEnabled TINYINT(1)
AccessFailedCount INT(11)
RegistrationDateTime DATETIME(6)
LastLoginDateTime DATETIME(6)
ProfileStatus VARCHAR(300)
ProfileAvatarUrl VARCHAR(300)
ProfileAbout VARCHAR(1000)
FollowersCount INT(11)
FollowedCount INT(11)
PostsCount INT(11)
Indexes
posts
Id VARCHAR(255)
CreatedBy LONGTEXT
CreatedAt DATETIME(6)
DeletedBy LONGTEXT
DeletedAt DATETIME(6)
PostTitle VARCHAR(100)
PostImageUrl VARCHAR(300)
PostDescription VARCHAR(500)
PostPublicationDateTime DATETIME(6)
PostFavoritesCount INT(11)
PostCommentsCount INT(11)
ProfileId VARCHAR(36)
Indexes
rooms
Id VARCHAR(255)
CreatedBy LONGTEXT
CreatedAt DATETIME(6)
DeletedBy LONGTEXT
DeletedAt DATETIME(6)
ChatRoomTitle VARCHAR(100)
LastMessageValue VARCHAR(100)
LastMessageDateTime DATETIME(6)
Indexes
messages
Id VARCHAR(255)
CreatedBy LONGTEXT
CreatedAt DATETIME(6)
DeletedBy LONGTEXT
DeletedAt DATETIME(6)
MessageValue VARCHAR(2000)
MessageDateTime DATETIME(6)
ProfileId VARCHAR(36)
ChatRoomId VARCHAR(36)
Indexes
favorites
Id VARCHAR(255)
CreatedBy LONGTEXT
CreatedAt DATETIME(6)
DeletedBy LONGTEXT
DeletedAt DATETIME(6)
ProfileId VARCHAR(36)
PostId VARCHAR(36)
Indexes
profileranks
Id VARCHAR(255)
CreatedBy LONGTEXT
CreatedAt DATETIME(6)
DeletedBy LONGTEXT
DeletedAt DATETIME(6)
ProfileId VARCHAR(36)
RankId VARCHAR(36)
Indexes
blockedprofiles
Id VARCHAR(255)
CreatedBy LONGTEXT
CreatedAt DATETIME(6)
DeletedBy LONGTEXT
DeletedAt DATETIME(6)
ProfileId VARCHAR(36)
BProfileId VARCHAR(36)
Reason VARCHAR(200)
Indexes
profilegifts
Id VARCHAR(255)
CreatedBy LONGTEXT
CreatedAt DATETIME(6)
DeletedBy LONGTEXT
DeletedAt DATETIME(6)
ProfileId VARCHAR(36)
GiftId VARCHAR(36)
Indexes
ranks
Id VARCHAR(255)
CreatedBy LONGTEXT
CreatedAt DATETIME(6)
DeletedBy LONGTEXT
DeletedAt DATETIME(6)
RankTitle VARCHAR(100)
RankDescription VARCHAR(300)
Indexes
gifts
Id VARCHAR(255)
CreatedBy LONGTEXT
CreatedAt DATETIME(6)
DeletedBy LONGTEXT
DeletedAt DATETIME(6)
GiftName VARCHAR(100)
GiftImageUrl VARCHAR(300)
Indexes