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

Table of contents

1 Project Description 4
2 Soft-delete and soft-update methods analysis 5
  2.1 An additional column "master_id" 5
  2.2 Composite key 7
    2.2.1 Master update not affects child records 8
    2.2.2 Master update affects child records 9
  2.3 Making a decision 11
  2.4 Implementation 11
3 Diagram 12
4 Repository pattern 13
  4.1 Purpose 13
  4.2 Pluses 13
  4.3 Minuses 13
  4.4 Current implementation 14
5 UoW 15
  5.1 Purpose 15
  5.2 Current implementation: 15
6 Services/BLL 15
  6.1 Purpose/Pluses 15
  6.2 Minuses 15
  6.3 Current implementation 16
7 Project solution structure 17
  7.1 Shared, common codebase 17
    7.1.1 Contracts.DAL.Base 17
    7.1.2 Contracts.BLL.Base 17
    7.1.3 DAL.Base 17
    7.1.4 DAL.Base.EF 17
    7.1.5 BLL.Base.EF 17
  7.2 App specific codebase for our solution 18
    7.2.1 Domain 18
    7.2.2 Contracts.DAL.App 18

7.2.3 Contracts.BLL.App 18
7.2.4 DAL.App.EF 18
7.2.5 DAL.App.DTO.. 18
7.2.6 BLL.App... 18
7.2.7 BLL.App.DTO 18
7.2.8 Extensions.. 19
7.2.9 PublicApi.DTO... 19
7.2.10 Resources.. 19
7.2.11 WebApp.. 19
8 Internalization.. 20
9 Entry Points 21
9.1 User Managing 21
9.2 User Views 21
9.3 Administration Views.. 22
10 Images control 23
10.1 Managing miniature... 23
10.2 Storing 24
11 Client Application 25
11.1 Frameworks analysis 25
11.2 Summary.. 25
12 Deployment 26
13 Result... 27

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.

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)

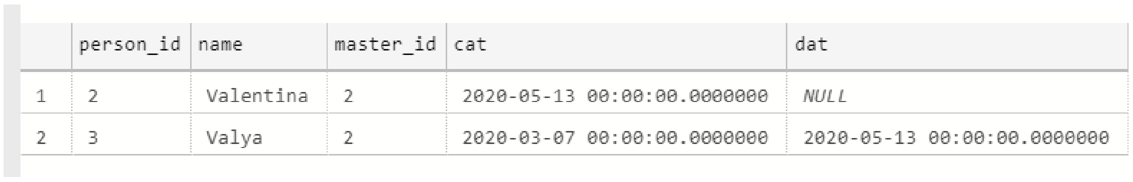
```
--create tables
CREATE TABLE Person (
    person_id   INT             NOT NULL    PRIMARY KEY,
    name        VARCHAR(128)    NOT NULL,
    master_id   INT             NOT NULL,
    cat         DATETIME2       NOT NULL,
    dat         DATETIME2       NULL
)
```

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
```

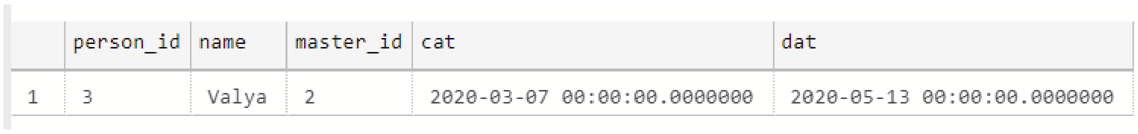


	person_id	name	master_id	cat	dat
1	2	Valentina	2	2020-05-13 00:00:00.0000000	*NULL*
2	3	Valya	2	2020-03-07 00:00:00.0000000	2020-05-13 00:00:00.0000000

Figure 1. Result of updating record using master_id column

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

Figure 2. Viewing history of record at @Time2

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)
)
```

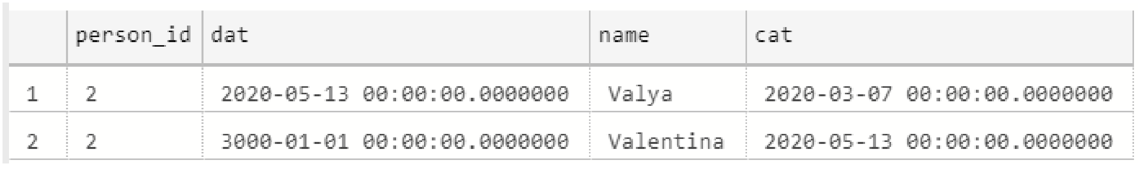
When a record is deleted, column "dat" is marked with the date it was deleted. This also affects dependent records: firstly, each composite foreign key record needs to be updated, and secondly, also be marked with the date of deletion.

```
--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.

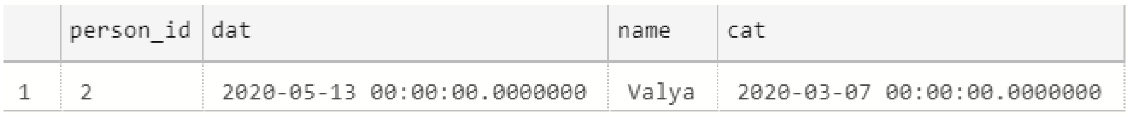
```
--soft update Valya (set name to Valentina) (childs not affected)
DECLARE @Valya_id INT
DECLARE @Valya_dat DATETIME2
SET @Valya_id = 2
SET @Valya_dat = '3000-01-01'

--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

Figure 3. Updating record using composite key

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



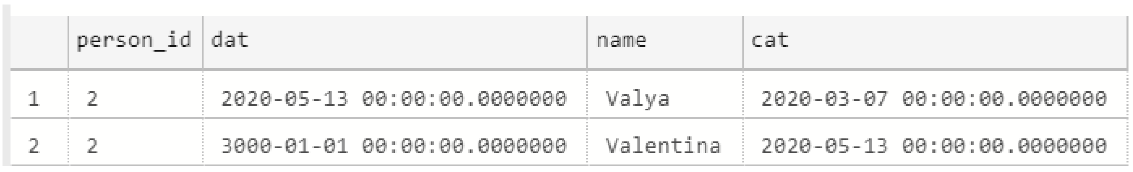
	person_id	dat	name	cat
1	2	2020-05-13 00:00:00.0000000	Valya	2020-03-07 00:00:00.0000000

Figure 4. Viewing history at @Time 2

2.2.2 Master update affects child records

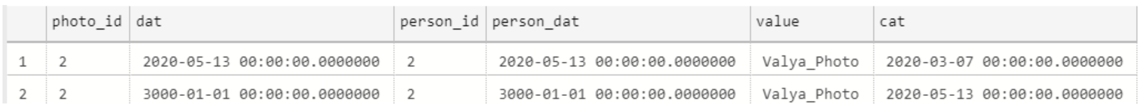
This case is more complicated, because when updating a record, you need to update both the composite key of the recording records and create a copy of them (as when updating)

```
--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
```



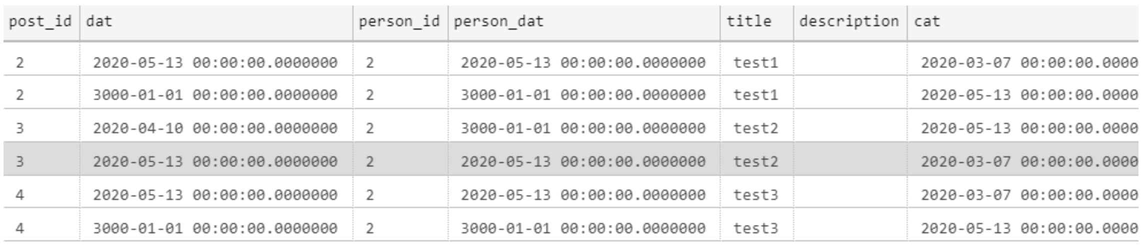
	person_id	dat	name	cat
1	2	2020-05-13 00:00:00.0000000	Valya	2020-03-07 00:00:00.0000000
2	2	3000-01-01 00:00:00.0000000	Valentina	2020-05-13 00:00:00.0000000

Figure 5. Updating record using composite key and modifying dependent entities



	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

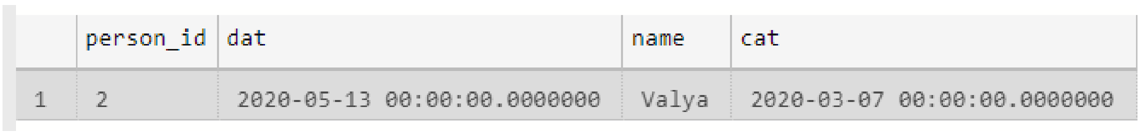
Figure 6. Updating dependent entity



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

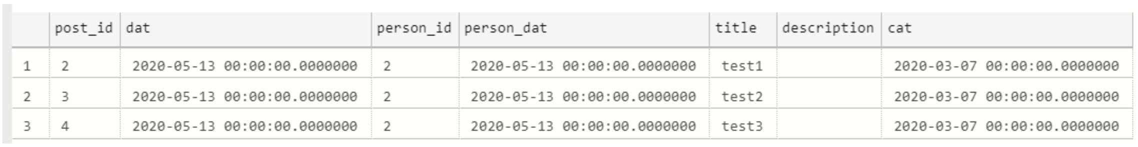
Figure 7. Viewing history

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



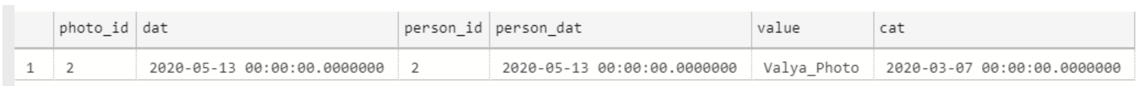
	person_id	dat	name	cat
1	2	2020-05-13 00:00:00.0000000	Valya	2020-03-07 00:00:00.0000000

Figure 8. Viewing person history



	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

Figure 9. Viewing post history



	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

Figure 10. Viewing photo history

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.

2.4 Implementation

Majority of records supports soft-update and soft-delete. Using interfaces ISoftDeleteEntity and ISoftUpdateEntity database context determine next step.

If entity is implementing ISoftUpdateEntity interface, when updating, created a copy of current record and saves in database with filled DeletedAt, DeletedBy and MasterId fields. Current record modifies and updates in database.

If entity is implementing ISoftDeleteEntity interface, when deleting, c record modifies with filled DeletedAt and DeletedBy fields and updates in database.

3 Diagram

[file in attachment]

4 Repository pattern

4.1 Purpose

It is not a very 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 UoW

5.1 Purpose

Unit of Work provides effective access to data, conflicts and transactions handling and atomic save.

Business logic will modify several entities and inform UoW that work is done

UoW makes an atomic save, business logic does not have to worry about implementation details

5.2 Current implementation:

BaseUnitOfWork provides factory methods to create/get instance of repository for services.

BaseDALMapper provides mapping between Domain and DAL layers.

6 Services/BLL

Like in repository pattern direct access is not good for scalability and maintainability.

To improve stability and pure design of our controllers (presentation layer), the BLL layer must be implemented.

6.1 Purpose/Pluses

Reduce data complexity and data quantity in between clients and rest service.

Clean controllers, reusable business logic.

6.2 Minuses

Is hard to implement first time, due to seemingly complexity.

6.3 Current implementation

BaseService class represents base implementation of service using generics, to avoid unnecessary implementations for each entity. **BaseService** implements **IBaseService** interface to support the Dependency Inversion principle.

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

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

BaseBLL provides factory methods to create/get instance of service for controllers.

BaseBLLMapper provides mapping between DAL and BLL layers.

7 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 Base projects are deployed to nuget system, as they can be used in other solutions in future.

7.1 Shared, common codebase

7.1.1 Contracts.DAL.Base

This project provides specs for domain metadata and PK in entities, and specs for base repository and DTO objects layer mapper between database and repositories.

7.1.2 Contracts.BLL.Base

This project provides specs for base service and DTO objects layer mapper between repositories and services.

7.1.3 DAL.Base

DAL.Base - abstract implementations of interfaces of domain entities and metadata.

7.1.4 DAL.Base.EF

DAL.Base.EF - abstract implementation of common base repository, unit of work and DTO mapper.

7.1.5 BLL.Base.EF

BLL.Base.EF - abstract implementation of common base service, bll and DTO mapper.

7.2 App specific codebase for our solution

7.2.1 Domain

This project provides communication between solution and database. It contains domain entities, which represent tables in db.

7.2.2 Contracts.DAL.App

This project provides interfaces for UoW and each repository.

7.2.3 Contracts.BLL.App

This project provides interfaces for BLL and each service.

7.2.4 DAL.App.EF

This project provides implementations of UoW, repositories, mappers and database access context (**ApplicationDbContext**) which provide communication, control and database migration system.

Class **DataInitialisers** provide database automatic drop, migrate, seed, which is configurable from appsettings

7.2.5 DAL.App.DTO

This project contains all DAL DTO objects to transfer data between DAL and BLL

7.2.6 BLL.App

This project provides implementations of BLL, services and mappers

7.2.7 BLL.App.DTO

This project contains all BLL DTO objects to transfer data between BLL and Controllers

7.2.8 Extensions

This project provides extension methods like current user id and JWT generation.

7.2.9 PublicApi.DTO

This project contains all PublicApi DTO objects to transfer data between REST controllers and clients.

All DTO are tied up with API version.

7.2.10 Resources

This project provides translation string for views, controllers etc.

7.2.11 WebApp

This is main project of solution. It provides API backend using REST controllers and ASP.NET Core MVC client using MVC controllers.

8 Internalization

Solution is pretended to be international and must support internalization. For this course scope app will support English, Estonian and Russian Languages.

For views use Resources project, which contains translations strings. For language selection, application has selector on top of page. Current culture stores in cookies and saves after reloading.

Also due to that every country has own time format, all fields contain time must be internalized. To approach this, views use JavaScript based scripts to change format of display. For UX project has “flatpickr” js-based library to convert default time pickup forms into usable forms.

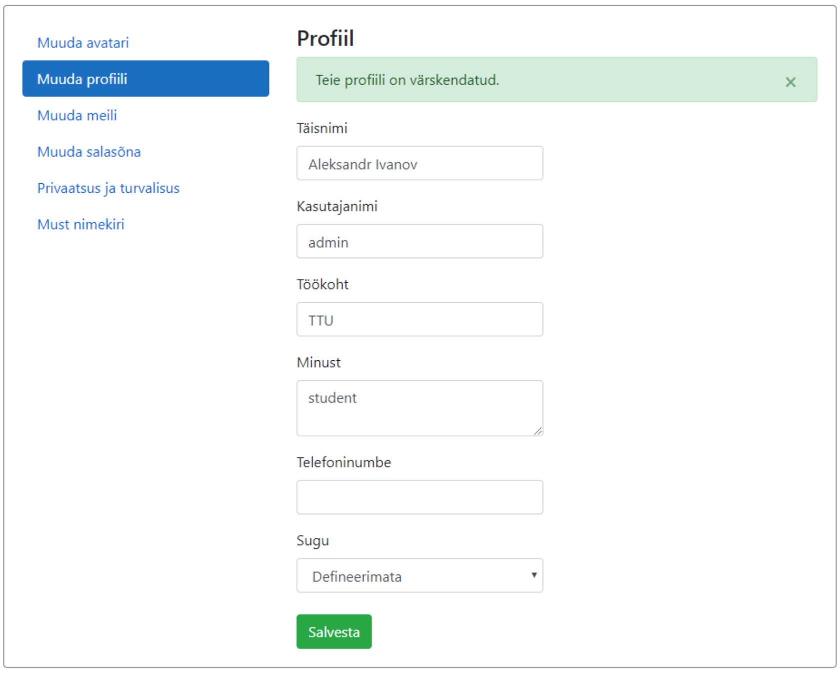
For some specific database entities, such as “ChatRoles”, “Gifts” and “Ranks” database must support translation entities. Using current application culture language string convert to suitable form. Default language for language strings is English. When record is created, language string creates using current culture. When updates, database create new language string for new language, or updates current.

REST controllers are also support internalization requests using special culture header in request.

9 Entry Points

Application have three areas: for managing user data, for user access and for administrating.

9.1 User Managing



Muuda avatari
Muuda profiili
Muuda meili
Muuda salasõna
Privaatsus ja turvalisus
Must nimekiri

Profiil

Teie profiili on värskendatud. ×

Täisnimi: Aleksandr Ivanov

Kasutajanimi: admin

Töökoht: TTU

Minust: student

Telefoninumbe:

Sugu: Defineerimata

Salvesta

Figure 11. Profile data managing page

9.2 User Views

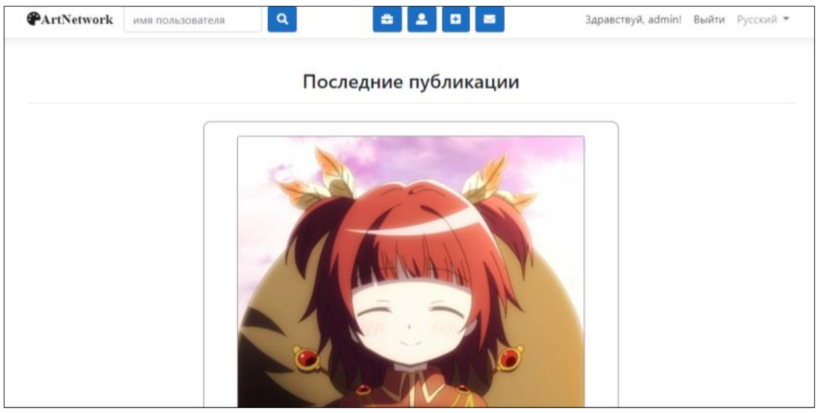


Figure 12. Application index page

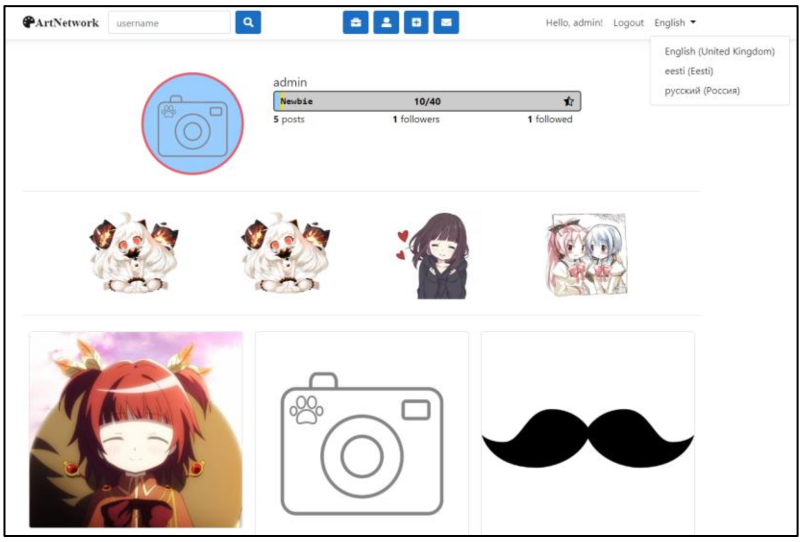


Figure 13. Profile “admin” user page

9.3 Administration Views

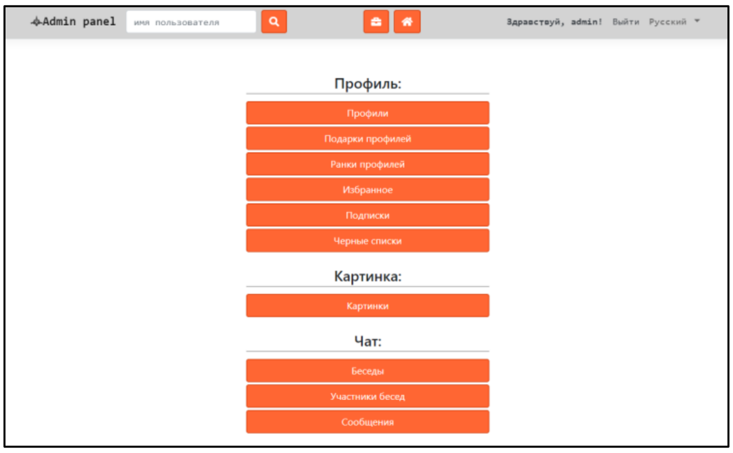
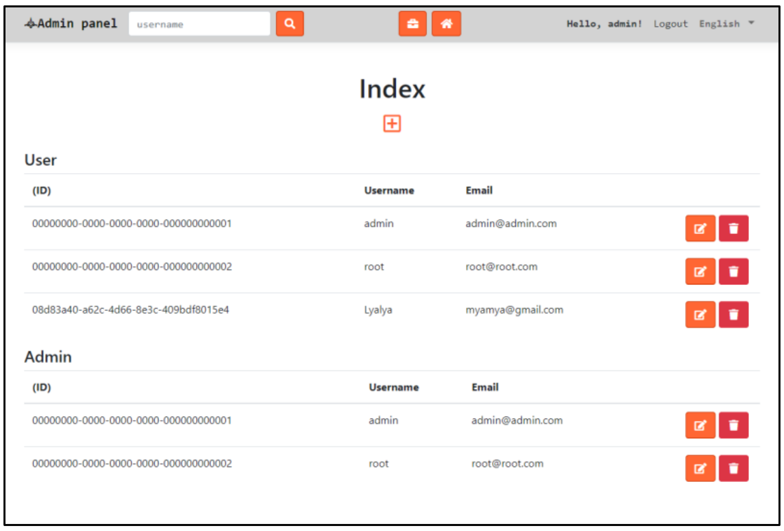


Figure 14. Administration panel



Index

User

(ID)	Username	Email
00000000-0000-0000-0000-000000000001	admin	admin@admin.com
00000000-0000-0000-0000-000000000002	root	root@root.com
08d83a40-a62c-4d66-8e3c-409bdf8015e4	Lyalya	myamya@gmail.com

Admin

(ID)	Username	Email
00000000-0000-0000-0000-000000000001	admin	admin@admin.com
00000000-0000-0000-0000-000000000002	root	root@root.com

Figure 15. Role managing index page

10 Images control

The main purpose of this application to be social network for artists, due to this, users must be able to submit posts with images, setup avatars and change images miniatures.

10.1 Managing miniature

Users must be able at least to crop images. For controlling image miniature, views load JavaScript based script, that create an image miniature outline.

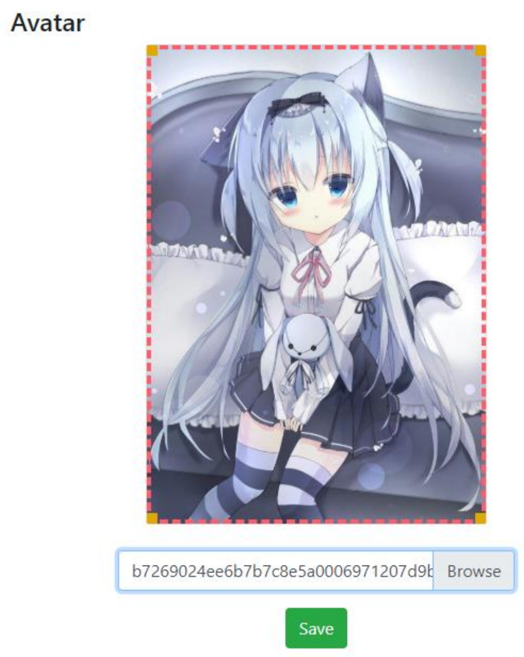
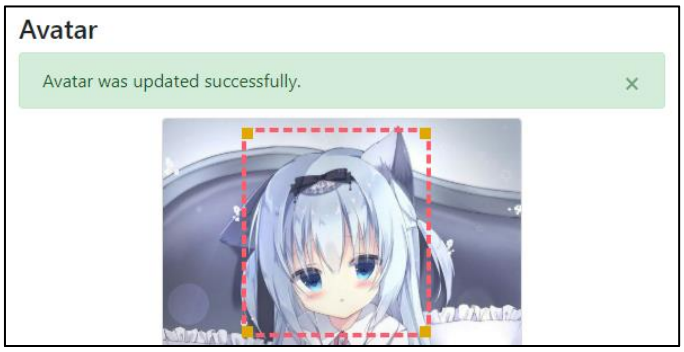


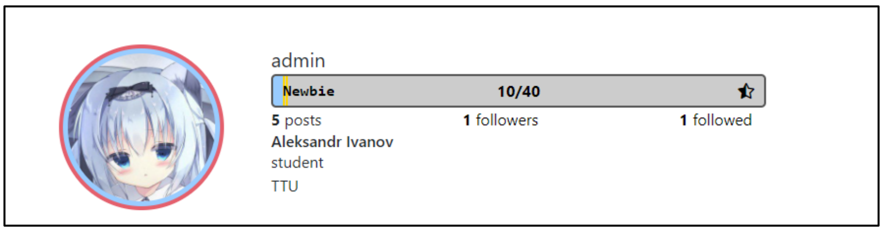
Figure 16. Avatar setup

On submitting image, it transfers to service by Image dto, which also contains IFormFile (if image was created or replaced). Then it validates and save two copies, original (for miniature setup later) and cropped copy (for display in posts or as avatar)


Avatar

Avatar was updated successfully. ×

Figure 17. Avatar was updated with miniature


admin

Newbie 10/40

5 posts 1 followers 1 followed

Aleksandr Ivanov

student

TTU

Figure 18. Profile “admin” user page after avatar setup

10.2 Storing

To store images was selected the easiest and free approach – to store locally. When image creates or updates with new image it stores in localstorage folder in application “wwwroot” directory.

Advantage of this method is simplicity, however if application is deployed using docker container, after restarting the system, all data created during container run is being destroyed.

11 Client Application

For client application was used JavaScript language, because it shared another course scope.

For client application was analyzed three main JavaScript frameworks:

- Aurelia
- Vue
- React

11.1 Frameworks analysis

Aurelia is easiest full framework, good to start with. Standards based, convention over configuration. Also supports Typescript, which is strongly typed and more useful that pure JavaScript.

Vue is also very easy, supports object and class base components. Views and code are shared in same files which is very useful. Supports Typescript, has powerful documentation.

React is more complex. Despite that it also supports class-based components, it is recommended to use functions instead. Needs time to become addictive.

11.2 Summary

For client realization I have chosen Vue, because it is more powerful that Aurelia, have better hooking system and good libraries like “Vuex” for state managing, “Vue-i18n” etc. and not so complex like React.

12 Deployment

There are several ways to do it:

- Full bare metal server
- Virtual server
- App hosting (azure) (deployable directly from Rider)
- Most standardized way – containers – Docker

The easiest and fastest way to deploy your application is docker image.

For hosting backend and client docker images is a good way to use Azure service, because it has many different options, webhooks and pricing plans, which is very important for student who just want to test application.

Backend image is built with premade Microsoft images.

mcr.microsoft.com/dotnet/core/sdk:latest - build
mcr.microsoft.com/dotnet/core/aspnet:latest - run

Backend image also includes "libgdiplus" library, which is needed to save images to localstorage.

Client image is build using "nginx" image

backend deployed to Azure - *http://artnetwork.azurewebsites.net/*

client deployed to Azure - *https://artnetwork-js.azurewebsites.net/*

13 Result

The goal of this project was to create an international social network for artists. However the result was quite a bit different. Building a fully functional social network is not so simple. It requires a lot of thinking and a lot more imlementation that my solution has.

My implementation became small social network, where artist (and not only) can publish their images, receive feedback from comment, follow and communicate other people in this small community. It also provides usable administration tool and internalisation.

Building such complex system is really hard for one person, not surprising that companies like Facebook have hundred of employes of different kind working around. However if we look behind we might be able to see something simmilar to my solution scope at first versions of that such complex systems today.