TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Maria-Helen Vozdvizenski 185756IADB

ONLINE ART GALLERY

Scope of work in Distributed Systems project

Juhendaja: Andres Käver

Tallinn 2020

Table of Contents

1 Introduction

The aim of this project is to create an online art selling platform. Users will be able to buy and comment on paintings from multiple artists, whose paintings will be displayed on the site. It will be possible to log onto the site using different services, such as Facebook and Google accounts. The payments will be possible using PayPal or credit card information.

The motivation for this project came from authors personal interest in art, drawing and painting. This project will make buying art more convenient and in turn help lesser known artist generate a revenue.

The basic process behind the platform is as follows – an artist contacts us if they wish to sell their art on our site and delivers the painting to our warehouse. Then the art is put on the site for display, which means that users can comment on it freely. If the piece is not sold within a certain timeframe, the painting will be returned back to the artist and it is removed from the website. However if the painting is bought within that timeframe, then it will be shipped accordingly and the pay will be redirected to the artist. The site will collect 10-30% of the pay depending on the size and shipping costs. The painting will be kept on the site for display.

2 Diagrams

The following schema was made using QSEE SuperLite.

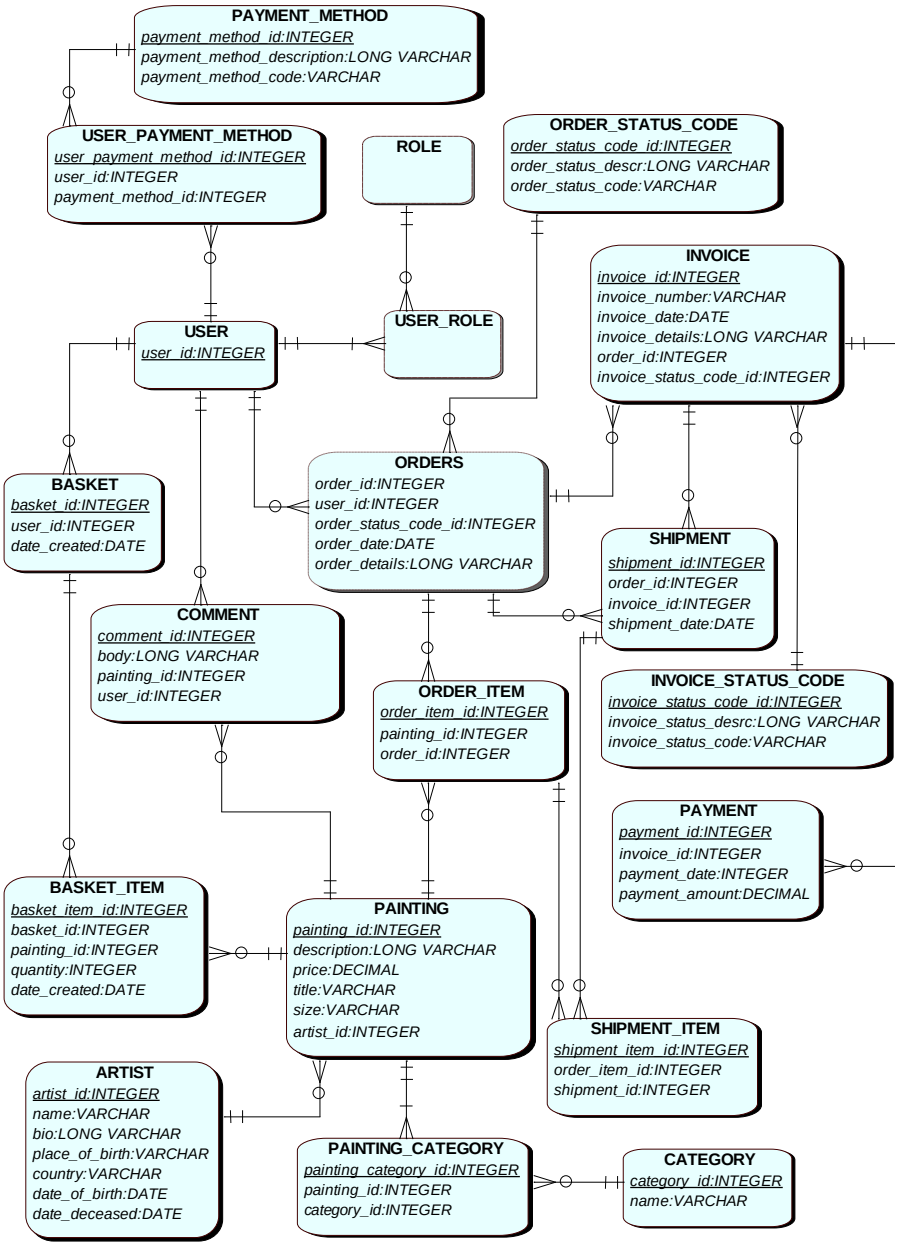

Diagram 1 - ERD

3 Database design

Both soft update and soft delete will be implemented in the database. In addition to the attributes that the entities already have, DeletedAt and CreatedAt columns will be added to them.

Soft delete means that a record is not deleted in a traditional sense – the current date at the time of our supposed deletion will be added to the DeletedAt column of a record, thus marking it as deleted. Soft update works using the same principle – the old record is not just overwritten with new data, instead it will be marked as deleted in the DeletedAt column and then a copy of that record with new data inside it shall be added to the database.

Using soft delete in the database reduces the risk of data loss, as the data is not actually deleted but only marked as such. Soft updating allows us to check the history of changes on a record. Thus, reverting back to the previous state of a record in case of need becomes easier.

Composite primary- and foreign keys shall be used in the database. A composite key consisting of an Id and a DeletedAt field will be created. Since none of the fields making up the primary key can be NULL, we will have to give the DeletedAt column a default value, which will indicate that the record has not been deleted yet. It can be any date far in the future, in this case it will be ’9999-12-12’.

3.1 Soft Update

3.1.1 Soft Update - Single Table

Soft updating in a single table is fairly simple as we do not have to worry about the relationships and how the change of a primary key affects them.

The process is as follows – the record that needs updating is first marked as deleted, by inserting the current date in the DeletedAt column. Then, the values from that record are copied, updated and inserted into the database, while the DeletedAt value will be set to ’9999-12-12’ which indicates that the current record is up to date.

```
DECLARE @OriginalId INT
SELECT @OriginalId = Id FROM Artist Where Name like 'Foo'

-- Set IDENTITY_INSERT to ON so that we can set the Id ourselves
SET IDENTITY_INSERT Artist ON

-- Update 1
UPDATE Artist SET DeletedAt = @Time3 WHERE Id = @OriginalId AND DeletedAt = @
TimeFuture
INSERT INTO Artist (Id, Name, PlaceOfBirth, Country, DateOfBirth, DateDecease
d, CreatedAt, DeletedAt) SELECT Id, 'Fido', PlaceOfBirth, Country, DateOfBirt
h, DateDeceased, CreatedAt, @TimeFuture FROM Artist where Id = @OriginalId

SET IDENTITY_INSERT Artist OFF
```

3.1.2 Soft Update - 1:m

Soft updating tables that are in one to many relationship requires more steps. Updating records on the child side of the relationship works the same way as with the single table case. However problems arise when a record on the master side of the relationship requires an update.

First of all, an error will be thrown out when the DeletedAt column in the master record is modified, as it is a part of the composite foreign key in the child record. Therefore referential integrity is broken. Implementing cascade update is needed to resolve this problem. It is implemented when creating a foreign key in the child table.

```
Alter TABLE Painting WITH CHECK ADD CONSTRAINT FK_Painting_Artist FOREIGN KEY
 (ArtistId, ArtistDeletedAt ) REFERENCES Artist (Id, DeletedAt) ON UPDATE CAS
CADE
```

Secondly, by using this approach the timeline will be lost. Therefore, the child record has to be updated manually using the soft update principles. Meaning that the DeletedAt column of the child record will be set to the current date and a new copy of that record shall be inserted into the database.

```
DECLARE @OriginalId2 INT
SELECT @OriginalId2 = Id FROM Artist Where Name like 'Foo'

-- Set IDENTITY_INSERT to ON so that we can set the Id ourselves
SET IDENTITY_INSERT Artist ON

-- Update the master
UPDATE Artist SET DeletedAt = @Time6 WHERE Id = @OriginalId2 AND DeletedAt =
@TimeFuture2
INSERT INTO Artist (Id, Name, PlaceOfBirth, Country, DateOfBirth, DateDecease
d, CreatedAt, DeletedAt) SELECT Id, 'Fido', PlaceOfBirth, Country, DateOfBirt
h, DateDeceased, CreatedAt, @TimeFuture2 FROM Artist where Id = @OriginalId2
AND DeletedAt = @Time6

SET IDENTITY_INSERT Artist OFF

SELECT 'Master table after soft update'
SELECT * FROM Artist

--Update the child record
--Set the child as deleted
SET IDENTITY_INSERT Painting ON

UPDATE Painting SET DeletedAt = @Time6 WHERE Id = @OriginalId2 AND DeletedAt
= @TimeFuture2
INSERT INTO Painting (Id, ArtistId, ArtistDeletedAt, [Description], Price, Ti
tle, [Size], DeletedAt, CreatedAt ) SELECT Id, ArtistId, @TimeFuture2, [Descr
iption], Price, Title, [Size], @TimeFuture2, CreatedAt FROM Painting where Ar
tistId = @OriginalId2 AND DeletedAt = @Time6

SET IDENTITY_INSERT Painting OFF
```

3.2 Soft Delete

3.2.1 Soft Delete – Single Table

As previously stated, soft delete does not actually remove the record from the database. Instead, the DeletedAt column is updated with the current date indicating that a record is deleted.

```
UPDATE Artist SET DeletedAt=@Time2 WHERE Name like '%a%'
```

3.2.2 Soft Delete – 1:m

Deleting the child record of the relationship works the same way as with a single table.

When deleting a master record, the child has to be deleted aswell. The DeletedAt field of the child record will be set to the same date as the DeletedAt field of the master record.

```
-- Deleting the master record
UPDATE Artist SET DeletedAt = @Time8 WHERE Id = @OriginalId4 AND DeletedAt =
'9999-12-12'

-- Deleting the child record
UPDATE Painting SET DeletedAt = @Time8 WHERE ArtistId = @OriginalId4 AND Arti
stDeletedAt = @Time8
```

3.3 Database design – 1:0-1

This kind of relationship proved to not work with soft update and soft delete. If a foreign key is in the master table then it cannot be unique. The reason is that a unique constraint cannot be NULL but a master record does not have to have a child record.

4 Repository design pattern

The Repository pattern is used for separating the data access logic from business logic. As the project grows bigger, using the repository pattern makes it more convenient for the developer to make changes regarding the data access, as it is all in one place instead of being scaterred around in different projects and files. Therefore, using the repository pattern could save the developer hours of tedious work, making the code easier to maintain.

In addition to that, using the repository pattern avoids duplication of code. There are certain actions that are performed on many data objects. These actions include:

- Getting all records
- Creating a new record
- Getting a record by its primary key
- Deleting a record etc.

Duplication can be avoided by creating a base repository which implements forementioned actions.

4.1 Testing

The Repository pattern allows easy testing of application with unit tests.

By defining and putting the repository interfaces in the domain model layer and using Dependency Injection in the controllers, mock repositories that return fake data instead of data from the database can be implemented. This decoupled approach allows the

creation and running of unit tests that focus the logic the application without requiring connectivity to the database.[1]

4.2 Data Access Objects

The concept of DAO is roughly the same – as with the repository design pattern it stores data and abstracts the access to it. Repository is a higher level concept dealing directly with business/domain objects, while DAO is more lower level, closer to the database/storage dealing only with data. For data-centric apps, a repository and DAO are interchangeable because the 'business' objects are simple data.[2]

4.3 Implementation

In the current project the code is separated into two major parts – current app specific and common shared base.

App specific codebase	Shared codebase
Contracts.DAL.App – specifications for app specific repositories. DAL.App.EF - implementation of app specific repositories done in EF.	Contracts.DAL.Base - specifications for domain metadata and PK in entities. Specifications for common base repository. DAL.Base - abstract implementations of interfaces for domain. DAL.Base.EF - implementation of common base repository done in EF.

[1] https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design#the-repository-pattern

[2] https://blog.sapiensworks.com/post/2012/11/01/Repository-vs-DAO.aspx