TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Karl Oskar Anderson 185611IADB

CREATING A FOOD ORDERING ENVIRONMENT

Course work

Supervisor: Andres Käver

Tallinn 2019

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Karl Oskar Anderson 185611IADB

TOIDU TELLIMISE KESKKONNA LOOMINE

Kursuse iseseisev töö

Juhendaja: Andres Käver

Tallinn 2019

Author’s declaration of originality

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

Author: Karl Oskar Anderson

22.02.2020

Table of contents

List of figures

1 Introduction

The author noticed the shortcomings of the food ordering environments currently on the market and thus decided to develop his own vision.

The author believes that the design of the project symbolizes a snow-capped mountain top of his university journey. The author respects his own desire to embark upon this wide-eyed journey to emerge in the end feeling tired, slightly crushed, less naive perhaps, but ultimately beaming with accomplishment. Provided he does not find his end on the way that is.

The author prepares a project that he sells to companies operating within the bounds of a particular country, who are primarily engaged in preparing end-user customizable foods like pizza and serving them to customers. All food servicing companies will hereinafter be referred to collectively as restaurants. The project is of interest to companies that are satisfied with this solution and the end customers are private individuals who want to order a seal. In essence the author has a fictional client, business owner X, who wants to create a web service solution for their company Y in order for the company to operate online.

2 Project

Let there be a website www.ZY.ee where company administrators can create their own user accounts. Company administrators have rights to manage descriptions of their restaurants, add food to menus, change menus and change food prices.

Restaurants have a menu; the same menu may be used in different restaurants. When ordering food, the user simply has to choose the restaurant of his choice. Food is divided into categories: pizzas, drinks, desserts etc. Users should not be confused, all foods should be listed on a single page – meaning category is not recursive. One product can only be under one category. The pizzas on the menu are not definitive; the pizzas have a variety of sizes, bases and additional components that the customer specifies on purchase.

The food prices are determined by individual restaurants. The same food may be differently priced in different restaurants. All pizzas have a base price and basic components. The components of the pizza can be modified and added up to a certain limit. It must be possible to add one component to the same pizza several times. Removing the basic components will not reduce the price of the pizza. Basic components can be removed to some extent and new components can be added step by step. The components have a price, but the price of the basic components is included in the basic price of the pizza. Removing the basic pizza components will not reduce the price of the pizza, but you can add more components in place of the removed component. If the added component is more expensive than the removed base component, the difference will have to be paid by the customer; if lower, the customer will still have to pay the basic pizza price.

Meals do not have prices in multiple currencies, but there can be several prices for one meal at a given time. Users belong to customer groups; prices vary for different customer groups. If the user belongs to more than one customer group, the lowest price is assigned to the user. Price will be displayed in gross, net prices will be calculated to receipt by tax.

The customer can add food to their cart. The customer can choose the type of food handover: carryout/dine-in or by courier delivery. When ordering by courier, the customer must specify the delivery address. The address is saved under the user's account so that the food delivered to the same location will not need to be refilled next time. The invoice is issued not to the user but to a person, nevertheless the buying user will pay. The person who receives the food may not be the owner of the user account who paid the invoice. An invoice consists of invoice

lines. The invoice lines are not related to the price of food, otherwise the existing invoices will change as the price of the food changes (unless a significantly more advanced solution is used). Invoice rows cannot be changed- they are permanent. Calculating fees or physical limitations of the courier is beyond the scope of this project, we expect the pizzas to arrive free, anywhere and instantaneously.

The invoice can be shared between friends of the user. This is a useful feature for events, where one person is responsible for providing food for a group and compensation is expected. A new share item is created in order to share the invoice. Shared foods come from invoice lines. Manually selecting single invoice rows would not be user friendly, so sharing meals will be generated based on the invoice. Users can share foods with their friends at a percentage and the system calculates how much money that friend owns them. Unallocated foods are the property of the user.

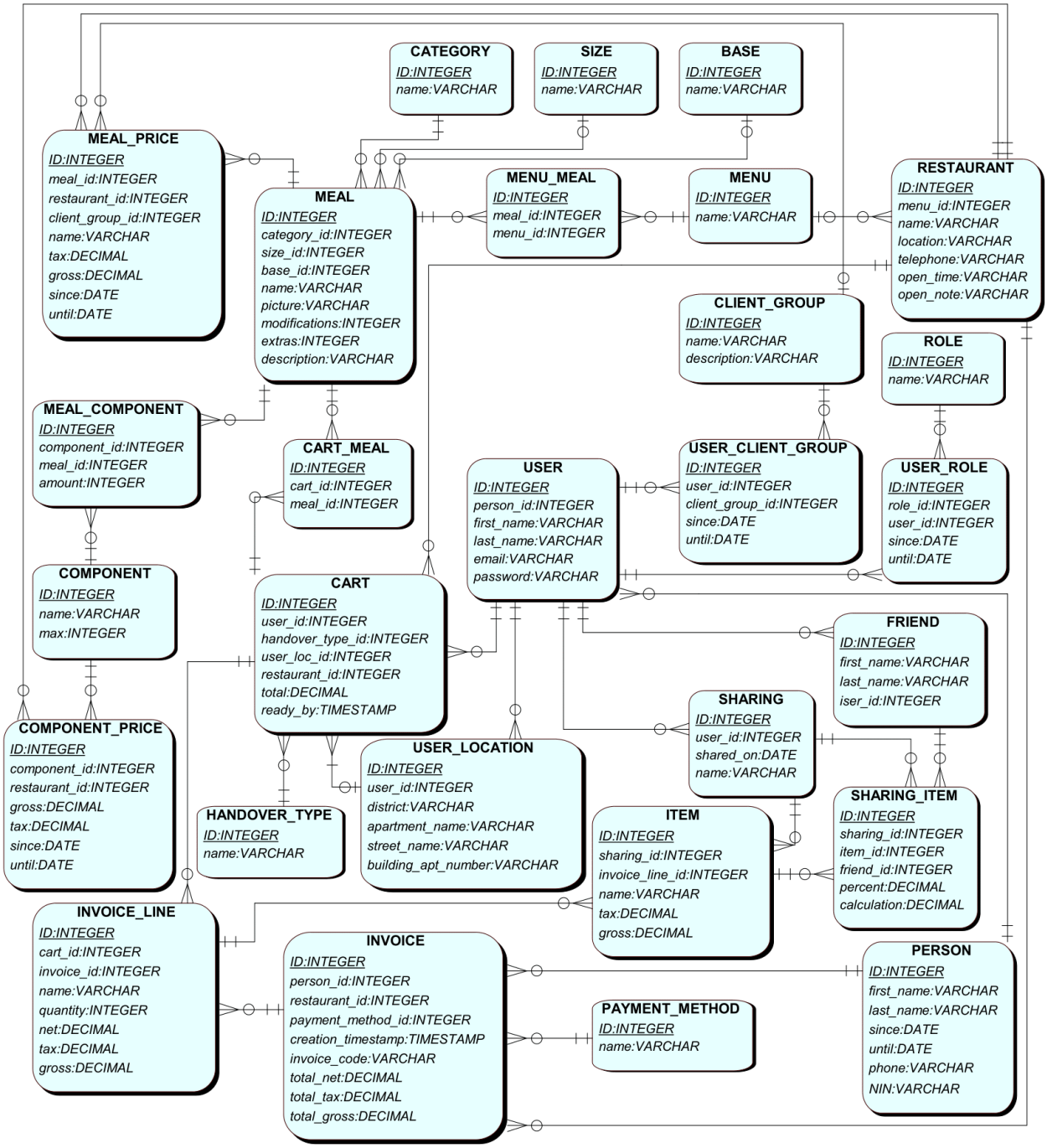

Figure 1: ERD – diagraph of the food ordering environment

3 Database *soft delete* and *soft update*

Author used Azure Data Studio to test different solutions on implementing *soft delete* and *soft update* on the project database. *Soft delete* in essence means that data is never deleted from a database, but rather marked as expired and SQL queries will have to be adjusted to filter these results out. *Soft update* means that database entries cannot be directly updated, but a new entry will be made, its exact relationship to other data will largely depend on the implementation.

In general *soft delete* is a bad idea, with few exceptions. Implementing them as a matter of course, regardless of the actual needs of their application is foolish. Their use as a data loss is trivial as databases should be backed up as archives regularly and data loss prevention by natural causes is neglected by 2-step delete-update process. They increase database size and complexity significantly. Performance decreases, but using an additional table to separate storing deleted data and active data somewhat mitigates this along with partitioning tricks. On a development standpoint omitted WHERE clauses will result in chaos. There is no agreed upon solution on implementation and a great number of workaround have to be made It only has a niche use in preventing invalid deletes and keeping track of every data change that is required when dealing with financial data.

Over one table soft update is demonstrated by creating a Person table. Person table has a compound primary key of Id and DeletedAt. Updating entries is done by keeping the “old” entry active and creating a new entry that is actually expired. Updating- a new entry will be created that will be identical to the old entry besides the DeletedAt field. The old entry is updated directly changing its payload and its CreatedAt value that will be identical to the old entries DeletedAt value. This approach will mean that no foreign reference integrity will be broken. Also Id will act uniquely in terms of separate objects meaning the Id will not change for the same object, only the DeletedAt field. This property can be exploited to sort objects by DeletedAt field descendingly and while it might seem clever to select the top 1 provided that it is not expired or none if it is expired, unfortunately objects can be set to be created in the future, stopping this approach from working.

	Id	PersonName	CreatedAt	DeletedAt
1	1	Niilo	2020-01-01 00:00:00.0000000	2999-01-01 00:00:00.0000000
2	2	Raimo	2020-01-01 00:00:00.0000000	2020-01-02 00:00:00.0000000
3	2	Raimo	2020-01-02 00:00:00.0000000	2020-01-03 00:00:00.0000000
4	2	Raimo	2020-01-03 00:00:00.0000000	2999-01-01 00:00:00.0000000

1:0-1 connection is demonstrated by a Person table and a User table. Users might be connected to a person, but there is also the possibility that User is just a shared account among admins and is therefore not connected to a person. A person is not necessarily a user, but a person is required when a user makes a invoice on the behalf of someone else. In that case the User will have Person reference as null. For this examples letter T is added to avoid SQL Server keywords and have no semantic meaning.

	ID	PersonID	PersonDeletedAt	CreatedAt	DeletedAt	Email	passwordT
1	1	2	2999-01-01 00:00:00.0000000	2020-01-03 00:00:00.0000000	2020-01-04 00:00:00.0000000	raimo@hot.ee	maasikas
2	1	2	2999-01-01 00:00:00.0000000	2020-01-04 00:00:00.0000000	2999-01-01 00:00:00.0000000	raimo@gmail.ee	maasikas
3	2	*NULL*	*NULL*	2020-01-03 00:00:00.0000000	2999-01-01 00:00:00.0000000	admin@gmail.com	admin

NB! This example is purely demonstrative; such connections don't exist in the scope of this project. In this project these tables share a connection of User-1---0, m-Person, where a Person is automatically created on the creation of a User with a true Boolean this_is_me. Further persons can be created by the user for marking receivers for goods. Even if the user is the receiver invoice is still given to the person of that user as per the words of database expert Priit Raspel: "User is technical property, but invoice goes to the person that is connected to the user."

1:m connection is demonstrated by tables MealPrice and Meal. One meal can have multiple prices over time and even multiple values at the same time. There need to be a opportunity for one meal to have multiple prices depending on the UserGroup the purchaser belonged to and exact restaurant to allow this system to be used by restaurant chains. A price change should be doable beforehand so no one has to change hundreds of meal prices at midnight.

	ID	MealID	MealDeletedAt	Gross	CreatedAt	DeletedAt	ID	CreatedAt	DeletedAt	nameT
1	1	1	2999-01-01 …	6,50	2020-01-02 …	2020-01-03 …	1	2020-01-01 …	2999-01-01 …	Romana
2	2	2	2999-01-01 …	5,50	2020-01-02 …	2020-01-03 …	2	2020-01-01 …	2999-01-01 …	Americana
3	3	3	2999-01-01 …	5,90	2020-01-02 …	2020-01-03 …	3	2020-01-01 …	2999-01-01 …	Topolino
4	3	3	2999-01-01 …	5,60	2020-01-03 …	2999-01-01 …	3	2020-01-01 …	2999-01-01 …	Topolino

4 Repository pattern

Repository patterns are a concept from DDD (Domain Driven Design). They are used in enterprise applications for various reasons like increasing testability; easing data storage swapping and avoiding duplicate code by provides an abstraction data layer between domain and data mapping layers. Repository acts like an in-memory domain object collection. CRUD operations with data from this collection is done through a series of straightforward methods, making database specific concerns related to how the objects's state is saved and later retrieved irrelevant. [1]

Although repositori pattern and DAO (Data Access Object) have both the function of abstracting data persistence and hiding SQL queries of dubious beauty. A DAO is closer to the underlying storage, it's really data centric. It returns data as an object state. A repository is on a higher level. It does everything a DAO does, but also deals with domain objects. A repository returns data as a business object. Sometimes DAO and repository are the same thing. [2]

In the scope of this project a repository will be used, as it's implementation is less time consuming. The project is estimated to have 26 domain objects, that includes identity objects, therefore avoiding tedious work and code duplication. In the abstraction galore that is this project there are class libraries Contacts.DAL.App, Contracts.DAL.Base, DAL.App.EF, DAL.Base and DAL.Base.EF. In Dal.Base.EF there is a generic class EFBaseRepository and it will continue to be there unless it is refactored again. This class is responsible for CRUD (create, read, update, delete) operations for all classes in DAL.App.EF/Repositories. The following code snippet demonstrates how generic EFBaseRepository can be implemented by every domain object just by specifying its class.

```
public class CartRepository : EFBaseRepository<Cart, AppDbContext>,
ICartRepository
{
    public CartRepository(AppDbContext dbContext) : base(dbContext)
    {
    }
}
```

This implementation pattern means that each domain class interface contains little actual code, but can for specificc reason shall the need arise.

References

[1] Code with Shadman:

https://codewithshadman.com/repository-pattern-csharp/ [Accessed 26 03 2020]

[2] Sapiens Works:

https://blog.sapiensworks.com/post/2012/11/01/Repository-vs-DAO.aspx [Accessed 26 03 2020]

Extras

```
IF db_id('kaande.kaande_softdelete') IS NOT NULL BEGIN
    USE master
    DROP DATABASE "kaande.kaande_softdelete"
END
GO

CREATE DATABASE "kaande.kaande_softdelete"
GO

USE "kaande.kaande_softdelete"
GO

-- Single table
CREATE TABLE Person (
    Id      INT             NOT NULL,
    PersonName    VARCHAR(128)    NOT NULL,
    CreatedAt DATETIME2     NOT NULL,
    DeletedAt DATETIME2     NOT NULL,
    PRIMARY KEY (Id, DeletedAt)
)

DECLARE @Time1 DATETIME2
SELECT @Time1 = '2020-01-01'
DECLARE @Time0 DATETIME2
SELECT @Time0 = '2999-01-01'
INSERT INTO Person (Id, PersonName, CreatedAt, DeletedAt) VALUES (1, 'Niilo', @Time1, @Time
0)
INSERT INTO Person (Id, PersonName, CreatedAt, DeletedAt) VALUES (2, 'Raimo', @Time1, @Time
0)


DECLARE @Time2 DATETIME2
SELECT @Time2 = '2020-01-02'

UPDATE Person SET CreatedAt = @Time2 where Id = 2 and DeletedAt = @Time0;
INSERT INTO Person (Id, PersonName, CreatedAt, DeletedAt) VALUES (2, 'Raimo', @Time1, @Time
2);

DECLARE @Time3 DATETIME2
SELECT @Time3 = '2020-01-03'

UPDATE Person SET CreatedAt = @Time3 where Id = 2 and DeletedAt = @Time0;
INSERT INTO Person (Id, PersonName, CreatedAt, DeletedAt) VALUES (2, 'Raimo', @Time2, @Time
3);

SELECT 'Initial data'
SELECT * FROM Person

SELECT * FROM Person where Id = 2 ORDER BY DeletedAt;

-- 1:0-1
CREATE TABLE UserT (
    ID      INT             NOT NULL,
```

```
    PersonID INT            NULL,
    PersonDeletedAt DATETIME2   NULL,
    CreatedAt DATETIME2     NOT NULL,
    DeletedAt DATETIME2     NOT NULL,
    Email VARCHAR(64)       NOT NULL,
    passwordT VARCHAR(64)   NOT NULL,
    FOREIGN KEY (PersonID, PersonDeletedAt) REFERENCES Person(Id, DeletedAt),
    PRIMARY KEY (ID, DeletedAt)
)

DECLARE @Time0 DATETIME2
SELECT @Time0 = '2999-01-01'
DECLARE @Time1 DATETIME2
SELECT @Time1 = '2020-01-01'
DECLARE @Time3 DATETIME2
SELECT @Time3 = '2020-01-03'
DECLARE @Time4 DATETIME2
SELECT @Time4 = '2020-01-04'
INSERT INTO UserT (ID, PersonID, PersonDeletedAt, CreatedAt, DeletedAt, Email, PasswordT) V
ALUES (1, 2, @Time0, @Time3, @Time0, 'raimo@hot.ee', 'maasikas')
INSERT INTO UserT (ID, PersonID, PersonDeletedAt, CreatedAt, DeletedAt, Email, PasswordT) V
ALUES (2, NULL, NULL, @Time3, @Time0, 'admin@gmail.com', 'admin')

SELECT 'Initial data'
SELECT * FROM UserT


UPDATE UserT SET CreatedAt = @Time4, Email= 'raimo@gmail.ee' where Id = 1 and DeletedAt = @
Time0;
INSERT INTO UserT (ID, PersonID, PersonDeletedAt, CreatedAt, DeletedAt, Email, PasswordT) V
ALUES (1, 2, @Time0, @Time3, @Time4, 'raimo@hot.ee', 'maasikas');

DECLARE @Time3 DATETIME2
SELECT @Time3 = '2020-01-03'
SELECT UserT.ID, UserT.CreatedAt, UserT.DeletedAt, UserT.Email, UserT.passwordT, Person.Id,
 Person.CreatedAt,
Person.DeletedAt, Person.PersonName FROM UserT INNER JOIN Person on UserT.PersonID = Person
.Id AND
UserT.PersonDeletedAt = Person.DeletedAt WHERE UserT.CreatedAt > @Time3;


-- 1:m
CREATE TABLE Meal (
    ID      INT             NOT NULL,
    CreatedAt DATETIME2     NOT NULL,
    DeletedAt DATETIME2     NOT NULL,
    nameT VARCHAR(64)       NOT NULL,
    PRIMARY KEY (ID, DeletedAt)
)

CREATE TABLE Meal_Price (
    ID      INT             NOT NULL,
    MealID INT              NULL,
    MealDeletedAt DATETIME2     NOT NULL,
    Gross DECIMAL(18,2)           NULL,
    CreatedAt DATETIME2     NOT NULL,
    DeletedAt DATETIME2     NOT NULL,
    FOREIGN KEY (MealID, MealDeletedAt) REFERENCES Meal(ID, DeletedAt),
```

```
    PRIMARY KEY (ID, DeletedAt)
)

DECLARE @Time0 DATETIME2
SELECT @Time0 = '2999-01-01'
DECLARE @Time1 DATETIME2
SELECT @Time1 = '2020-01-01'
DECLARE @Time2 DATETIME2
SELECT @Time2 = '2020-01-02'
DECLARE @Time3 DATETIME2
SELECT @Time3 = '2020-01-03'
INSERT INTO Meal (ID, CreatedAt, DeletedAt, nameT) VALUES (1, @Time1, @Time0, 'Romana')
INSERT INTO Meal (ID, CreatedAt, DeletedAt, nameT) VALUES (2, @Time1, @Time0, 'Americana')
INSERT INTO Meal (ID, CreatedAt, DeletedAt, nameT) VALUES (3, @Time1, @Time0, 'Topolino')

INSERT INTO Meal_Price (ID, MealId, MealDeletedAt, Gross, CreatedAt, DeletedAt) VALUES (1,
1, @Time0, 6.50, @Time2, @Time3)
INSERT INTO Meal_Price (ID, MealId, MealDeletedAt, Gross, CreatedAt, DeletedAt) VALUES (2,
2, @Time0, 5.50, @Time2, @Time3)
INSERT INTO Meal_Price (ID, MealId, MealDeletedAt, Gross, CreatedAt, DeletedAt) VALUES (3,
3, @Time0, 5.90, @Time2, @Time0)

select * from Meal_Price
select top 1 * from Meal_Price where ID = 1 ORDER by CreatedAt desc

UPDATE Meal_Price SET CreatedAt = @Time3, Gross = 5.60 where Id = 3 and DeletedAt = @Time0;
INSERT INTO Meal_Price (ID, MealId, MealDeletedAt, Gross, CreatedAt, DeletedAt) VALUES (3,
3, @Time0, 5.90, @Time2, @Time3);


SELECT UserT.ID, UserT.CreatedAt, UserT.DeletedAt, UserT.Email, UserT.passwordT, Person.Id,
 Person.CreatedAt,
Person.DeletedAt, Person.PersonName FROM UserT INNER JOIN Person on UserT.PersonID = Person
.Id AND
UserT.PersonDeletedAt = Person.DeletedAt WHERE UserT.CreatedAt > @Time3;

DECLARE @Time1 DATETIME2
SELECT @Time1 = '2020-01-01'
select * from Meal_Price INNER JOIN Meal on Meal_Price.MealID = Meal.ID AND
Meal_Price.MealDeletedAt = Meal.DeletedAt WHERE Meal_Price.CreatedAt > @Time1;
```