TALLIN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

PIZZA ORDERING WEB APPLICATION

Author: Robert Tõnisson
Supervisor: Andres Käver

Tallinn 2020

Table of contents

1. Project description

This project is going to be ASP .NET web application for pizza delivery app. The reasoning for selecting this topic is to make a better application than the existing pizza delivery apps. In the end of this project, there is going to be functional application, that has the good features from currently available pizza restaurants as well as newly implemented features.

The reason for selecting this topic is to compare the results with other students, who chose the same project. This gives an understanding of my level in development, that I can use for choosing an approach for further learning. Also, this is an interesting topic to me, because I love pizza and I think that learning while doing something that I love is the best way to approach it.

This app can be used by different pizza restaurants, like Peetri Pizza, New York Pizza, Pizza Kiosk etc. It allows clients to choose whatever pizza they like from the menu. They have full control over pizza they want to receive. That means, they can add additional toppings, remove toppings that they don’t like. They can also choose what type of crust they prefer, what size should the pizza be.

Store allows to order drinks with their pizzas and even more products if the restaurant pleases to do so. That being said, the order must contain pizza and this application will secure that it is so to make it easier for the restaurant.

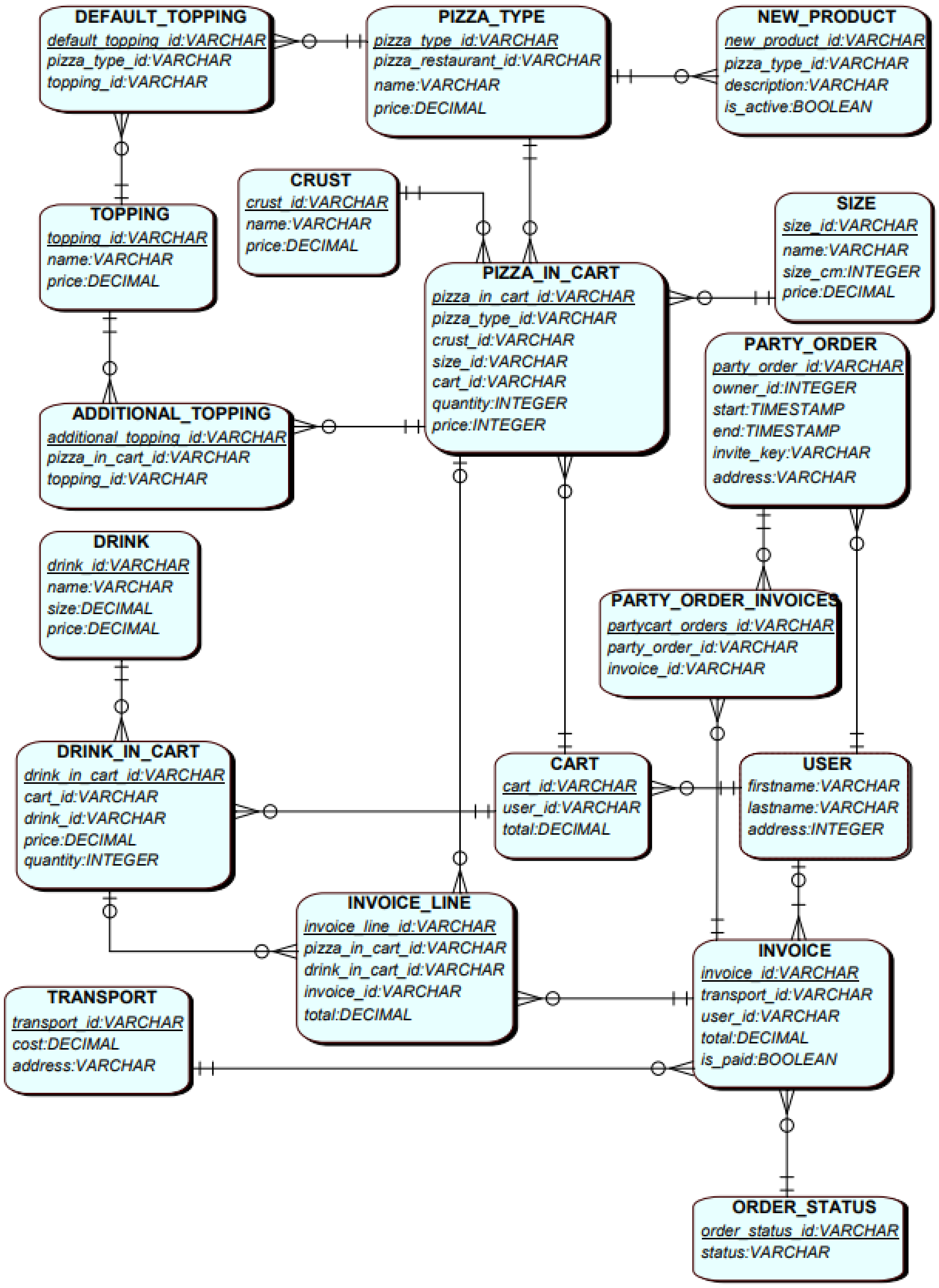
The User Interface for this web application will be very friendly for the user. Everything for creating a pizza is in one place, with simple “add and remove by clicking”. The total cost is always visible. It’s easy to remove products before final confirmation and it even recommends.

Every past order is saved for the user, so it’s easy to „one click order“ he previous ones or even add or remove something and then order it.

The application will have full security, with user roles, authentications, confirmations etc.

There are a lot more to this project, that was not covered in this short description. Also, this has to mentioned that it is still in the beginning stages of the development, so many features will most likely be added as the development continues.

2. ERD-schema


DEFAULT_TOPPING
default_topping_id:VARCHAR
pizza_type_id:VARCHAR
topping_id:VARCHAR
PIZZA_TYPE
pizza_type_id:VARCHAR
pizza_restaurant_id:VARCHAR
name:VARCHAR
price:DECIMAL
NEW_PRODUCT
new_product_id:VARCHAR
pizza_type_id:VARCHAR
description:VARCHAR
is_active:BOOLEAN
CRUST
crust_id:VARCHAR
name:VARCHAR
price:DECIMAL
TOPPING
topping_id:VARCHAR
name:VARCHAR
price:DECIMAL
SIZE
size_id:VARCHAR
name:VARCHAR
size_cm:INTEGER
price:DECIMAL
PIZZA_IN_CART
pizza_in_cart_id:VARCHAR
pizza_type_id:VARCHAR
crust_id:VARCHAR
size_id:VARCHAR
cart_id:VARCHAR
quantity:INTEGER
price:INTEGER
PARTY_ORDER
party_order_id:VARCHAR
owner_id:INTEGER
start:TIMESTAMP
end:TIMESTAMP
invite_key:VARCHAR
address:VARCHAR
ADDITIONAL_TOPPING
additional_topping_id:VARCHAR
pizza_in_cart_id:VARCHAR
topping_id:VARCHAR
DRINK
drink_id:VARCHAR
name:VARCHAR
size:DECIMAL
price:DECIMAL
PARTY_ORDER_INVOICES
partycart_orders_id:VARCHAR
party_order_id:VARCHAR
invoice_id:VARCHAR
DRINK_IN_CART
drink_in_cart_id:VARCHAR
cart_id:VARCHAR
drink_id:VARCHAR
price:DECIMAL
quantity:INTEGER
CART
cart_id:VARCHAR
user_id:VARCHAR
total:DECIMAL
USER
firstname:VARCHAR
lastname:VARCHAR
address:INTEGER
INVOICE_LINE
invoice_line_id:VARCHAR
pizza_in_cart_id:VARCHAR
drink_in_cart_id:VARCHAR
invoice_id:VARCHAR
total:DECIMAL
INVOICE
invoice_id:VARCHAR
transport_id:VARCHAR
user_id:VARCHAR
total:DECIMAL
is_paid:BOOLEAN
TRANSPORT
transport_id:VARCHAR
cost:DECIMAL
address:VARCHAR
ORDER_STATUS
order_status_id:VARCHAR
status:VARCHAR

3. Soft delete and update in SQL

Nowadays it's important for businesses to keep all most of their data history. That means that database schemas should implement a solution for not actually deleting any records but marking them as "deleted" in database. Also, when updating record, we need to add another copy with new values, rather than modifying the existing one. This kind of deletion is called soft delete/update and we need to find the best way to implement this in our pizza app's database.

3.1 The problems

The problem arises with relationships in database schemas. Normally, it would be easy to just cascade delete a record, meaning that if we delete something from parent table, we also must delete every child table that uses it and the children of the child table and so on. When we need to implement soft delete/update, the record in parent table will stay there and is just marked as deleted. That means cascade update will not work, because they still point back to the record, which is now marked as deleted, which means the system will not have any children to update. Same thing happens when deleting. We would have to delete the parent record manually in each step to the lowest of the children.

There is another problem – how can we keep track of the history if we mark a record as deleted and add another one. Normally that would mean that the primary key(id) will change within the new record. That means those two records will not have any connection between each other.

3.2 The solution – composite key

If we want to keep the connection between record that are deleted to the current one, we can not use ID as primary key. As we know, the record, which is currently in use, must be marked as not deleted. If we use DATETIME to mark something as deleted, we can have the primary key as a composite key with fields ID, and DeletedAt. With that, ID can be used multiple times, as DeletedAt will always differ. The used record DeletedAt must be marked as a date in the far future (9999.01.01 for example), because a field in a primary key cannot be nullable.

With this method, updating our data will be easy, since we will not have to update our children tables. For that, the child table also must have composite key, with the same fields as our parent table primary key, as a foreign key. To update record in parent table, we need to create a copy with of the original and set value to DeletedAt field. Then we can change the original one. The database will not give an exception, because the foreign key will point to the correct record. We can easily find the history of the parent records from a child by using the id field in foreign key.

Deleting records from parent table will not become much easier. Automatic cascade delete wouldn’t work, because there won’t be any deleting. We can use cascade update, but that would only change the foreign key part of the child record and we would still have to mark the record itself as deleted. Same with the children of the children.

The example tables with query results to prove this solution can be found in appendices (appendix 1).

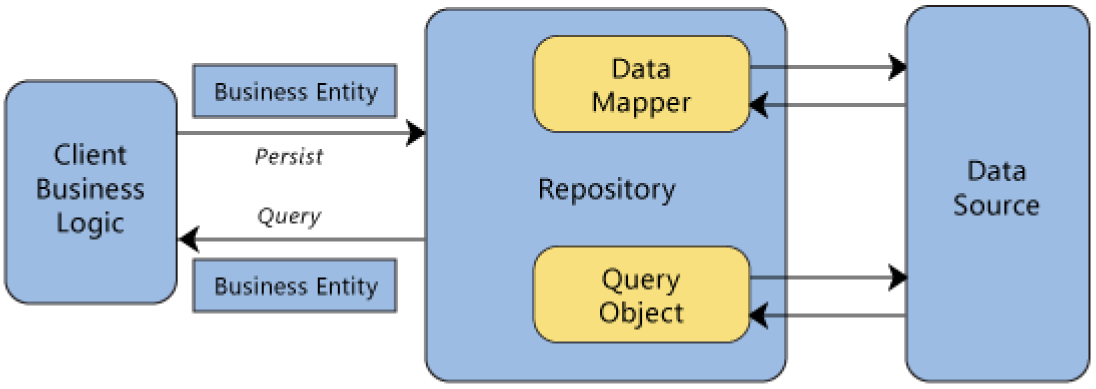
4. Repository pattern

In our solution we are going to use repository pattern for abstracting data access. It was chosen over something like Data Object Access Layer because it reduces code duplication, separates data form business logic and makes it easier to adapt changes for company.

4.1 What Is Repository Pattern?

A Repository goes between the domain and data mapping layers (like Entity Framework). It makes it possible to take a set of records from the database, and then have those records to work like an in-memory domain object collection. You can also update or delete records within those data sets, and the code written in the repository will carry out the necessary operations behind the scenes. (Codes with Shadman, Shadman Kudchikar, 2019)

Repository pattern is a way to implement data access by encapsulating the set of objects persisted in a data store and the operations performed over them, providing a more object-oriented view of the persistence layer. It also makes it possible to have a clean separation and one-way dependency between the domain and data mapping layers. Repository pattern is mostly used where we need to modify the data before passing to the next stage. Here's a graph that illustrates the idea:



(Codes with Shadman, Shadman Kudchikar, 2019)

4.2 Why Repository Pattern?

Code duplication: You can reduce the amount of code by separating company business logic with application specific logic. Company can use same logic within multiple applications and add application specific logic within the application layer itself. Using repositories is like putting something between your application and your data, so it doesn't matter what data storage method you use. Your app just wants the data and does not want to know how it was stored or where it comes from. This makes making changes easier because we need to implement change to only one place not for every area in the application that would normally save items to your tables.

Increase testability: Repository systems are good for testing. One reason being that you can use Dependency Injection. Basically, you create an interface for your repository, and you reference the interface for it when you are making the object. Then you can later make a fake object which implements that interface. You can then bind the proper type to that interface. Boom you've just taken a dependence out of the equation and replaced it with something testable. (Codes with Shadman, Shadman Kudchikar, 2019)

Easily swapped out with various data storages without changing the API: For example, in one instance, you may need to retrieve data from the database, in other cases you may need to retrieve something like xml-s or something else. Regardless, the idea behind the repository pattern is that whatever sits behind it doesn't matter so long as the API it provides works for the layer of the application calling into it. (Codes with Shadman, Shadman Kudchikar, 2019)

5. Making the project

5.1 Backend

After making a new solution in for web application in ASP, I started with making the necessary domain objects, that we use to generate tables to the MS-SQL server. These domain objects are also used to generate basic controllers for these tables. The project uses identity (users and roles) provided by the framework. After the generation for the base of the project, I started implementing repository patterns for every domain object. The reasoning was provided in the previous chapter.

After that I started implementing unit of work (UOW) in my solution. That helps to move the business logic away from the controllers by using single transaction, provided by the UOW, that makes all the necessary updates in the database and saves them. With all that provided, I started working with my controllers (both web and API controllers). I implemented JWT (JSON web token) based authentication in my project to secure my web-api controllers. There is also resource protection, so the users can only access the data that they allowed to.

In the middle of the project, I moved the projects and repositories into three layers. Domain layer – the bottom of the layers that gets the data directly from the database and passed it to the next layer. Data access layer (DAL) – the middle layer that maps the necessary data from the domain to DAL.DTO (data transfer object) or other way around and starts working with them by making queries, providing methods to update the database etc. The third layer is business logic layer (BLL). As the name suggests, this is where all the business logic is implemented in. It maps the DAL.DTO to BLL.DTO or other way around, it works with the data so only the required data gets sent back to the DAL to make changes in the database or provides controllers with the data.

At the end of the project, I pushed all the base projects into Nuget. These projects are not specific to the application and can be reused in other solutions. After that I started finished my ASP.NET application for the client and pushed the backend Docker into Azure.

5.2 Front-end

During the project, I also made a front-end client application. With the implementation of JWT we can authenticate the users from front-end and secure our api controllers. It also means we can finally work on our application. Our front-end is written TypeScript using Aurelia client framework for web. I chose Aurelia because it’s easy to learn and use, while providing all the necessary features. The focus is the communication between front-end and back-end, making sure that the correct data reaches API controllers, and the received data gets stored. To store data globally, we use app-state. It’s used to put data into JS localStorage. For example, it there is stored JWT tokens, user information, culture information etc. Page specific data is stored into its local variables. There is domain for our front-end as well so we can cast incoming data to objects from json. For design, we mainly use Bootstrap along with some pure HTML and CSS.

Front-end is also generated into Docker file and pushed into the Azure web.

5.3 ASP.NET web application

This is the same application that I have built in aurelia, but instead of using api controllers, I use Razor Views and controllers in C#. It has i18n support for cultures and languages and I’m using resource files for translation.

6. Project summary

In the end, almost everything that I had planned was done. Of course, it does not mean that my application is perfect. I learned so much during this course, but it made me realize how little I know about the world of distributed programming. There is so much potential for my app to grow and now all I must do is learn even more, improve my skills and make my next projects, or even this one, better.

7. References

https://codewithshadman.com/repository-pattern-csharp/#references

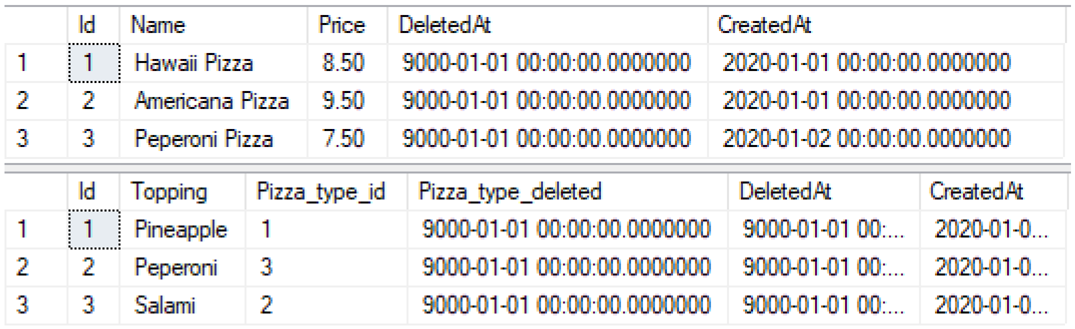
https://makingloops.com/why-should-you-use-the-repository-pattern/

8. Appendices

8.1 Soft delete/update example tables and queries

```
-- Script for testing soft delete/update between 1:m tables
-- Create basic table structure for one to many relationship
CREATE TABLE Pizza_Type (
       Id      INT                 NOT NULL    IDENTITY,
       Name    VARCHAR(128)        NOT NULL,
       Price   Decimal(5,2)        NOT NULL,
)

CREATE TABLE Default_Topping (
       Id                  INT                 NOT NULL IDENTITY ,
       Topping             Varchar(128)        NOT NULL,
       Pizza_type_id       INT                 NOT NULL,
       Pizza_type_deleted  DATETIME2           DEFAULT '9000-01-01' NOT NULL
)

-- Add metadata for soft delete
ALTER TABLE Pizza_Type      ADD DeletedAt DATETIME2 DEFAULT '9000-01-01' NOT NULL
ALTER TABLE Pizza_Type      ADD CreatedAt DATETIME2 NOT NULL
ALTER TABLE Default_Topping ADD DeletedAt DATETIME2 DEFAULT '9000-01-01' NOT NULL
ALTER TABLE Default_Topping ADD CreatedAt DATETIME2 NOT NULL

-- add index on metadata and ID
CREATE INDEX DeletedAt_idx  ON Pizza_Type       ( DeletedAt );
CREATE INDEX CreatedAt_idx  ON Pizza_Type       ( CreatedAt );
CREATE INDEX DeletedAt_idx  ON Default_Topping  ( DeletedAt );
CREATE INDEX CreatedAt_idx  ON Default_Topping  ( CreatedAt );
CREATE INDEX ID_idx         ON Pizza_Type       ( Id );
CREATE INDEX ID_idx         ON Default_Topping  ( Id );

--add COMPOSITE FKs and PKs
ALTER TABLE Pizza_Type ADD CONSTRAINT PK_Pizza_Type PRIMARY KEY (Id, DeletedAt)
ALTER TABLE Default_Topping ADD CONSTRAINT PK_Default_Topping
       PRIMARY KEY (Id, DeletedAt)
ALTER TABLE Default_Topping ADD CONSTRAINT FK_Pizza_Type
       FOREIGN KEY (Pizza_type_id, Pizza_type_deleted)
       REFERENCES Pizza_Type (Id, DeletedAt) ON DELETE CASCADE ON UPDATE CASCADE

DECLARE @Time1 DATETIME2
SELECT @Time1 = '2020-01-01'

-- add data to the table
INSERT INTO Pizza_Type (Name, Price, CreatedAt)
       VALUES ('Hawaii Pizza', 8.50, @Time1)
INSERT INTO Pizza_Type (Name, Price, CreatedAt)
       VALUES ('Americana Pizza', 9.50, @Time1)
SELECT @Time1 = '2020-01-02'
INSERT INTO Pizza_Type (Name, Price, CreatedAt)
       VALUES ('Peperoni Pizza', 7.50, @Time1)

SELECT @Time1 = '2020-01-03'
INSERT INTO Default_Topping (Topping, CreatedAt, Pizza_type_id, Pizza_type_deleted)
       VALUES ('Pineapple', @Time1, 1, '9000-01-01')
INSERT INTO Default_Topping (Topping, CreatedAt, Pizza_type_id, Pizza_type_deleted)
       VALUES ('Peperoni', @Time1, 3, '9000-01-01')
SELECT @Time1 = '2020-01-05'
INSERT INTO Default_Topping (Topping, CreatedAt, Pizza_type_id, Pizza_type_deleted)
       VALUES ('Salami', @Time1, 2, '9000-01-01')
```

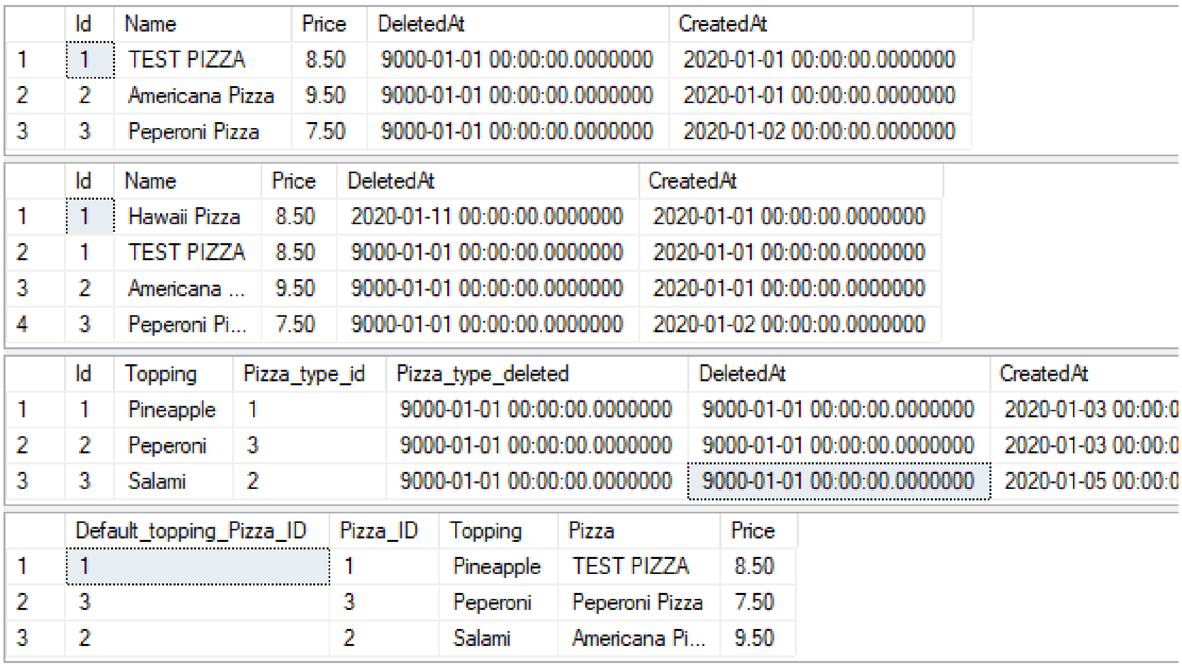
```sql
-- Select all data before updating/deleting
SELECT * FROM Pizza_Type
SELECT * FROM Default_Topping
```



	Id	Name	Price	DeletedAt	CreatedAt
1	1	Hawaii Pizza	8.50	9000-01-01 00:00:00.0000000	2020-01-01 00:00:00.0000000
2	2	Americana Pizza	9.50	9000-01-01 00:00:00.0000000	2020-01-01 00:00:00.0000000
3	3	Peperoni Pizza	7.50	9000-01-01 00:00:00.0000000	2020-01-02 00:00:00.0000000

	Id	Topping	Pizza_type_id	Pizza_type_deleted	DeletedAt	CreatedAt
1	1	Pineapple	1	9000-01-01 00:00:00.0000000	9000-01-01 00:...	2020-01-0...
2	2	Peperoni	3	9000-01-01 00:00:00.0000000	9000-01-01 00:...	2020-01-0...
3	3	Salami	2	9000-01-01 00:00:00.0000000	9000-01-01 00:...	2020-01-0...

```sql
-- soft update
DECLARE @Time2 DATETIME2
DECLARE @id INT
SELECT @Time2 = '2020-01-11'
       --get the id for the record/s to upadte
SELECT @id = Id FROM Pizza_Type Where Name = 'Hawaii Pizza'

       --make a copy of the record as a deleted record
SET IDENTITY_INSERT Pizza_Type ON
insert into Pizza_Type(id, DeletedAt, Name, Price, CreatedAt)
       select @id, @Time2, Name, Price, CreatedAt
       from Pizza_Type
       where Id = @id
SET IDENTITY_INSERT Pizza_Type OFF

       --change the original record values
UPDATE Pizza_Type SET Name='TEST PIZZA' WHERE Id=@id AND DeletedAt >
CURRENT_TIMESTAMP

-- All not updated Pizza_Type records
SELECT * FROM Pizza_Type WHERE DeletedAt > CURRENT_TIMESTAMP
-- All Pizza_Type records including pre-updated
SELECT * FROM Pizza_Type
-- All not updated Default_Topping records
SELECT * FROM Default_Topping WHERE DeletedAt > CURRENT_TIMESTAMP
-- Check if updated children point to correct parent
SELECT t.Pizza_type_id as 'Default_topping_Pizza_ID', p.id as 'Pizza_ID',
       Topping as 'Topping', p.Name as 'Pizza', p.Price as 'Price'
       FROM Default_Topping t
       JOIN Pizza_Type p ON t.Pizza_type_id = p.Id AND t.Pizza_type_deleted =
       p.DeletedAt
       AND p.DeletedAt > CURRENT_TIMESTAMP AND t.DeletedAt > CURRENT_TIMESTAMP
```



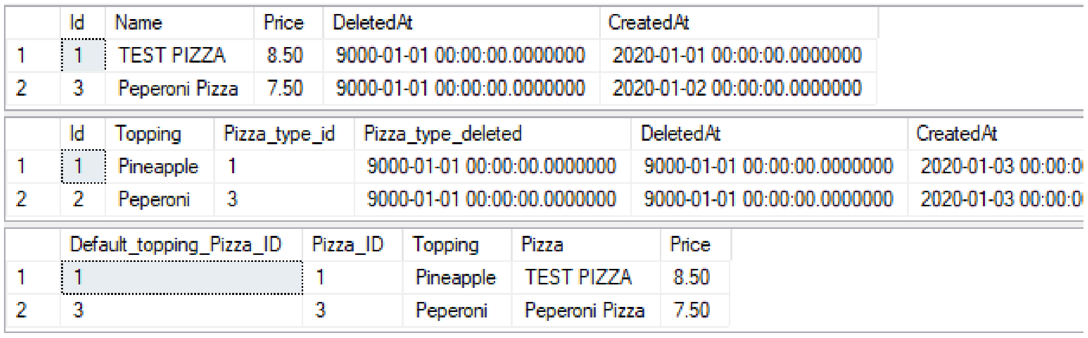
	Id	Name	Price	DeletedAt	CreatedAt
1	1	TEST PIZZA	8.50	9000-01-01 00:00:00.0000000	2020-01-01 00:00:00.0000000
2	2	Americana Pizza	9.50	9000-01-01 00:00:00.0000000	2020-01-01 00:00:00.0000000
3	3	Peperoni Pizza	7.50	9000-01-01 00:00:00.0000000	2020-01-02 00:00:00.0000000

	Id	Name	Price	DeletedAt	CreatedAt
1	1	Hawaii Pizza	8.50	2020-01-11 00:00:00.0000000	2020-01-01 00:00:00.0000000
2	1	TEST PIZZA	8.50	9000-01-01 00:00:00.0000000	2020-01-01 00:00:00.0000000
3	2	Americana ...	9.50	9000-01-01 00:00:00.0000000	2020-01-01 00:00:00.0000000
4	3	Peperoni Pi...	7.50	9000-01-01 00:00:00.0000000	2020-01-02 00:00:00.0000000

	Id	Topping	Pizza_type_id	Pizza_type_deleted	DeletedAt	CreatedAt
1	1	Pineapple	1	9000-01-01 00:00:00.0000000	9000-01-01 00:00:00.0000000	2020-01-03 00:00:0
2	2	Peperoni	3	9000-01-01 00:00:00.0000000	9000-01-01 00:00:00.0000000	2020-01-03 00:00:0
3	3	Salami	2	9000-01-01 00:00:00.0000000	9000-01-01 00:00:00.0000000	2020-01-05 00:00:0

	Default_topping_Pizza_ID	Pizza_ID	Topping	Pizza	Price
1	1	1	Pineapple	TEST PIZZA	8.50
2	3	3	Peperoni	Peperoni Pizza	7.50
3	2	2	Salami	Americana Pi...	9.50

```
-- soft delete
SELECT @Time1 = '2020-01-15'
SELECT @id = Id FROM Pizza_Type Where Name = 'Americana Pizza'
       --delete the record by setting value to DeletedAt
UPDATE Pizza_Type SET DeletedAt=@Time1 WHERE Id=@id AND DeletedAt >
CURRENT_TIMESTAMP
       --also need to delete the record from child table by setting value to
DeletedAt
UPDATE Default_Topping SET DeletedAt=@Time1 WHERE Pizza_type_id=@id AND DeletedAt >
CURRENT_TIMESTAMP
-- All not deleted Pizza_Type records
SELECT * FROM Pizza_Type WHERE DeletedAt > CURRENT_TIMESTAMP
-- All not deleted Default_Topping records
SELECT * FROM Default_Topping WHERE DeletedAt > CURRENT_TIMESTAMP
-- Check the remaining relationships between children and parents
SELECT t.Pizza_type_id as 'Default_topping_Pizza_ID', p.id as 'Pizza_ID', t.Topping
as 'Topping', p.Name as 'Pizza', p.Price as 'Price' FROM Default_Topping t
       JOIN Pizza_Type p ON t.Pizza_type_id = p.Id AND t.Pizza_type_deleted =
p.DeletedAt
       AND p.DeletedAt > CURRENT_TIMESTAMP AND t.DeletedAt > CURRENT_TIMESTAMP
```

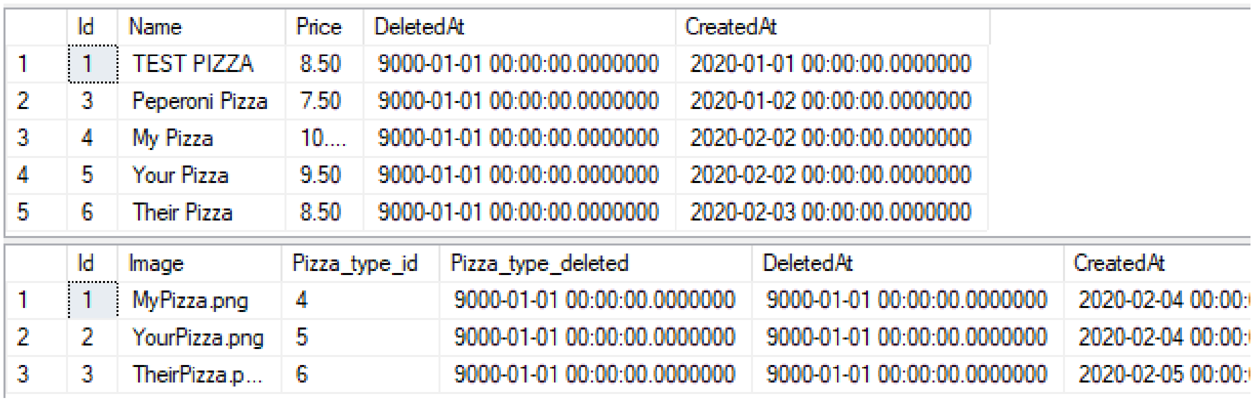


	Id	Name	Price	DeletedAt	CreatedAt
1	1	TEST PIZZA	8.50	9000-01-01 00:00:00.0000000	2020-01-01 00:00:00.0000000
2	3	Peperoni Pizza	7.50	9000-01-01 00:00:00.0000000	2020-01-02 00:00:00.0000000

	Id	Topping	Pizza_type_id	Pizza_type_deleted	DeletedAt	CreatedAt
1	1	Pineapple	1	9000-01-01 00:00:00.0000000	9000-01-01 00:00:00.0000000	2020-01-03 00:00:0
2	2	Peperoni	3	9000-01-01 00:00:00.0000000	9000-01-01 00:00:00.0000000	2020-01-03 00:00:0

	Default_topping_Pizza_ID	Pizza_ID	Topping	Pizza	Price
1	1	1	Pineapple	TEST PIZZA	8.50
2	3	3	Peperoni	Peperoni Pizza	7.50

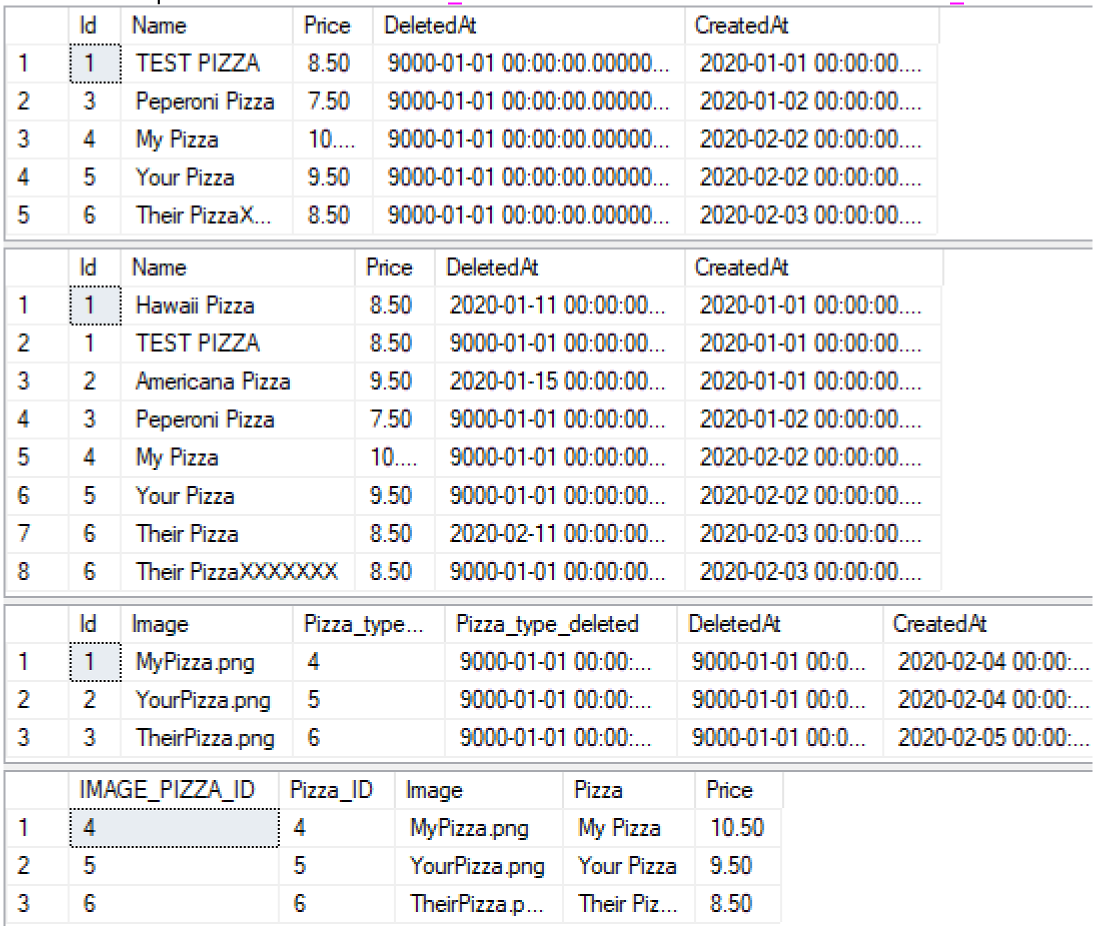
```sql
-- Script for testing soft delete/update between 1:0-1 tables
-- Create a table for 1 to 0-1 relationship
CREATE TABLE Pizza_Image (
       Id                   INT           NOT NULL IDENTITY,
       Image                VARCHAR(128)  NOT NULL,
       -- UNIQUE to make it 1:0-1
       Pizza_type_id        INT           NOT NULL UNIQUE,
       Pizza_type_deleted   DATETIME2     DEFAULT '9000-01-01' NOT NULL
)
-- Add metadata for soft delete
ALTER TABLE Pizza_Image           ADD DeletedAt DATETIME2 DEFAULT '9000-01-01' NOT
NULL
ALTER TABLE Pizza_Image           ADD CreatedAt DATETIME2
               NOT NULL
-- add index on metadata and ID
CREATE INDEX DeletedAt_idx ON Pizza_Image       ( DeletedAt );
CREATE INDEX CreatedAt_idx ON Pizza_Image       ( CreatedAt );

--add COMPOSITE FK and PK
ALTER TABLE Pizza_Image     ADD CONSTRAINT PK_Pizza_Image
       PRIMARY KEY (Id, DeletedAt)
ALTER TABLE Pizza_Image     ADD CONSTRAINT FK_IMAGE_Pizza_Type
       FOREIGN KEY (Pizza_type_id, Pizza_type_deleted)
       REFERENCES Pizza_Type (Id, DeletedAt) ON DELETE CASCADE ON UPDATE CASCADE

-- add data to the table
SELECT @Time1 = '2020-02-02'

INSERT INTO Pizza_Type (Name, Price, CreatedAt) VALUES ('My Pizza', 10.50, @Time1)
INSERT INTO Pizza_Type (Name, Price, CreatedAt) VALUES ('Your Pizza', 9.50, @Time1)
SELECT @Time1 = '2020-02-03'
INSERT INTO Pizza_Type (Name, Price, CreatedAt) VALUES ('Their Pizza', 8.50, @Time1)

SELECT @Time1 = '2020-02-04'
INSERT INTO Pizza_Image (Image, CreatedAt, Pizza_type_id, Pizza_type_deleted)
       VALUES ('MyPizza.png', @Time1, 4, '9000-01-01')
INSERT INTO Pizza_Image (Image, CreatedAt, Pizza_type_id, Pizza_type_deleted)
       VALUES ('YourPizza.png', @Time1, 5, '9000-01-01')
SELECT @Time1 = '2020-02-05'
INSERT INTO Pizza_Image (Image, CreatedAt, Pizza_type_id, Pizza_type_deleted)
       VALUES ('TheirPizza.png', @Time1, 6, '9000-01-01')

-- Select all data before updating/deleting
SELECT * FROM Pizza_Type WHERE DeletedAt > CURRENT_TIMESTAMP
SELECT * FROM Pizza_Image WHERE DeletedAt > CURRENT_TIMESTAMP
```



	Id	Name	Price	DeletedAt	CreatedAt
1	1	TEST PIZZA	8.50	9000-01-01 00:00:00.0000000	2020-01-01 00:00:00.0000000
2	3	Peperoni Pizza	7.50	9000-01-01 00:00:00.0000000	2020-01-02 00:00:00.0000000
3	4	My Pizza	10....	9000-01-01 00:00:00.0000000	2020-02-02 00:00:00.0000000
4	5	Your Pizza	9.50	9000-01-01 00:00:00.0000000	2020-02-02 00:00:00.0000000
5	6	Their Pizza	8.50	9000-01-01 00:00:00.0000000	2020-02-03 00:00:00.0000000

	Id	Image	Pizza_type_id	Pizza_type_deleted	DeletedAt	CreatedAt
1	1	MyPizza.png	4	9000-01-01 00:00:00.0000000	9000-01-01 00:00:00.0000000	2020-02-04 00:00:
2	2	YourPizza.png	5	9000-01-01 00:00:00.0000000	9000-01-01 00:00:00.0000000	2020-02-04 00:00:
3	3	TheirPizza.p...	6	9000-01-01 00:00:00.0000000	9000-01-01 00:00:00.0000000	2020-02-05 00:00:

```
-- soft update
SELECT @Time2 = '2020-02-11'
       --get the id for the record/s to upadte
SELECT @id = Id FROM Pizza_Type Where Name = 'Their Pizza'

       --make a copy of the record as a deleted record
SET IDENTITY_INSERT Pizza_Type ON
insert into Pizza_Type(id, DeletedAt, Name, Price, CreatedAt)
       select @id, @Time2, Name, Price, CreatedAt
       from Pizza_Type
       where Id = @id
SET IDENTITY_INSERT Pizza_Type OFF

       --change the original record values
UPDATE Pizza_Type SET Name='Their PizzaXXXXXXX' WHERE Id=@id AND DeletedAt >
CURRENT_TIMESTAMP

-- All not updated Pizza_Type records
SELECT * FROM Pizza_Type WHERE DeletedAt > CURRENT_TIMESTAMP
-- All Pizza_Type records including pre-update records
SELECT * FROM Pizza_Type
-- All not updated Pizza_Image records
SELECT * FROM Pizza_Image WHERE DeletedAt > CURRENT_TIMESTAMP
-- Check if updated children point to correct parent
SELECT t.Pizza_type_id as 'IMAGE_PIZZA_ID', p.id as 'Pizza_ID', Image as 'Image',
p.Name as 'Pizza', p.Price as 'Price' FROM Pizza_Image t
       JOIN Pizza_Type p ON t.Pizza_type_id = p.Id AND t.Pizza_type_deleted =
p.DeletedAt
       AND p.DeletedAt > CURRENT_TIMESTAMP AND t.DeletedAt > CURRENT_TIMESTAMP
```



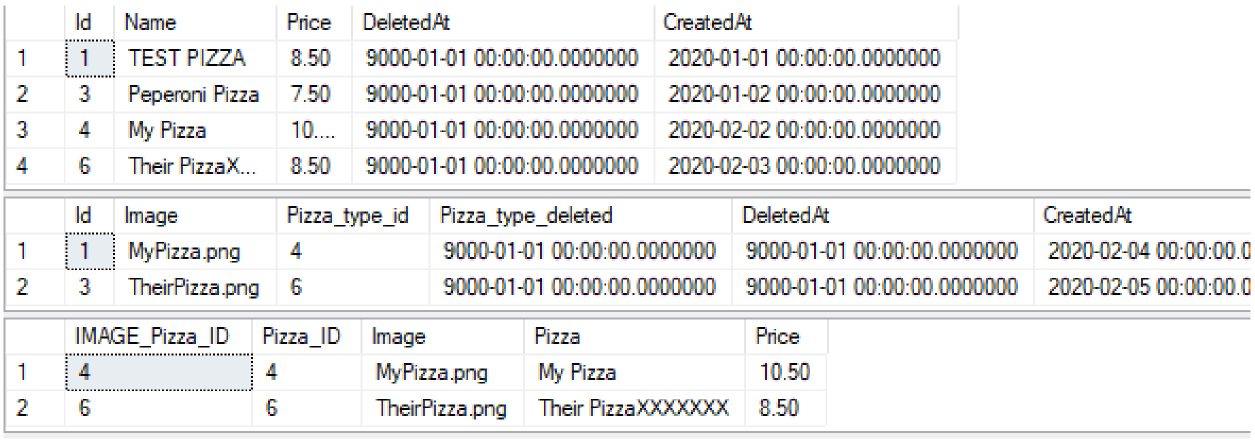
	Id	Name	Price	DeletedAt	CreatedAt
1	1	TEST PIZZA	8.50	9000-01-01 00:00:00.00000...	2020-01-01 00:00:00....
2	3	Peperoni Pizza	7.50	9000-01-01 00:00:00.00000...	2020-01-02 00:00:00....
3	4	My Pizza	10....	9000-01-01 00:00:00.00000...	2020-02-02 00:00:00....
4	5	Your Pizza	9.50	9000-01-01 00:00:00.00000...	2020-02-02 00:00:00....
5	6	Their PizzaX...	8.50	9000-01-01 00:00:00.00000...	2020-02-03 00:00:00....

	Id	Name	Price	DeletedAt	CreatedAt
1	1	Hawaii Pizza	8.50	2020-01-11 00:00:00...	2020-01-01 00:00:00....
2	1	TEST PIZZA	8.50	9000-01-01 00:00:00...	2020-01-01 00:00:00....
3	2	Americana Pizza	9.50	2020-01-15 00:00:00...	2020-01-01 00:00:00....
4	3	Peperoni Pizza	7.50	9000-01-01 00:00:00...	2020-01-02 00:00:00....
5	4	My Pizza	10....	9000-01-01 00:00:00...	2020-02-02 00:00:00....
6	5	Your Pizza	9.50	9000-01-01 00:00:00...	2020-02-02 00:00:00....
7	6	Their Pizza	8.50	2020-02-11 00:00:00...	2020-02-03 00:00:00....
8	6	Their PizzaXXXXXXX	8.50	9000-01-01 00:00:00...	2020-02-03 00:00:00....

	Id	Image	Pizza_type...	Pizza_type_deleted	DeletedAt	CreatedAt
1	1	MyPizza.png	4	9000-01-01 00:00:...	9000-01-01 00:0...	2020-02-04 00:00:...
2	2	YourPizza.png	5	9000-01-01 00:00:...	9000-01-01 00:0...	2020-02-04 00:00:...
3	3	TheirPizza.png	6	9000-01-01 00:00:...	9000-01-01 00:0...	2020-02-05 00:00:...

	IMAGE_PIZZA_ID	Pizza_ID	Image	Pizza	Price
1	4	4	MyPizza.png	My Pizza	10.50
2	5	5	YourPizza.png	Your Pizza	9.50
3	6	6	TheirPizza.p...	Their Piz...	8.50

```sql
-- soft delete
SELECT @Time1 = '2020-02-15'
SELECT @id = Id FROM Pizza_Type Where Name = 'Your Pizza'
       --delete the record by setting value to DeletedAt
UPDATE Pizza_Type SET DeletedAt=@Time1 WHERE Id=@id AND DeletedAt >
CURRENT_TIMESTAMP
       --also need to delete the record from child table by setting value to
DeletedAt
UPDATE Pizza_Image SET DeletedAt=@Time1 WHERE Pizza_type_id=@id AND DeletedAt >
CURRENT_TIMESTAMP


-- All not deleted Pizza_Type records
SELECT * FROM Pizza_Type WHERE DeletedAt > CURRENT_TIMESTAMP
-- All not deleted Pizza_Image records
SELECT * FROM Pizza_Image WHERE DeletedAt > CURRENT_TIMESTAMP
-- Check the remaining relationships between children and parents
SELECT t.Pizza_type_id as 'IMAGE_Pizza_ID', p.id as 'Pizza_ID', Image as 'Image',
p.Name as 'Pizza', p.Price as 'Price' FROM Pizza_Image t
       JOIN Pizza_Type p ON t.Pizza_type_id = p.Id AND t.Pizza_type_deleted =
p.DeletedAt
       AND p.DeletedAt > CURRENT_TIMESTAMP AND t.DeletedAt > CURRENT_TIMESTAMP
```



	Id	Name	Price	DeletedAt	CreatedAt
1	1	TEST PIZZA	8.50	9000-01-01 00:00:00.0000000	2020-01-01 00:00:00.0000000
2	3	Peperoni Pizza	7.50	9000-01-01 00:00:00.0000000	2020-01-02 00:00:00.0000000
3	4	My Pizza	10....	9000-01-01 00:00:00.0000000	2020-02-02 00:00:00.0000000
4	6	Their PizzaX...	8.50	9000-01-01 00:00:00.0000000	2020-02-03 00:00:00.0000000

	Id	Image	Pizza_type_id	Pizza_type_deleted	DeletedAt	CreatedAt
1	1	MyPizza.png	4	9000-01-01 00:00:00.0000000	9000-01-01 00:00:00.0000000	2020-02-04 00:00:00.0
2	3	TheirPizza.png	6	9000-01-01 00:00:00.0000000	9000-01-01 00:00:00.0000000	2020-02-05 00:00:00.0

	IMAGE_Pizza_ID	Pizza_ID	Image	Pizza	Price
1	4	4	MyPizza.png	My Pizza	10.50
2	6	6	TheirPizza.png	Their PizzaXXXXXXX	8.50