TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Enola Sander 179080IADB

DUNGEONS AND DRAGONS INVENTORY MANAGER

Scope of work in Distributed Systems project

Supervisor: Andres Käver
[Academic degree]

Table of contents

1 Introduction

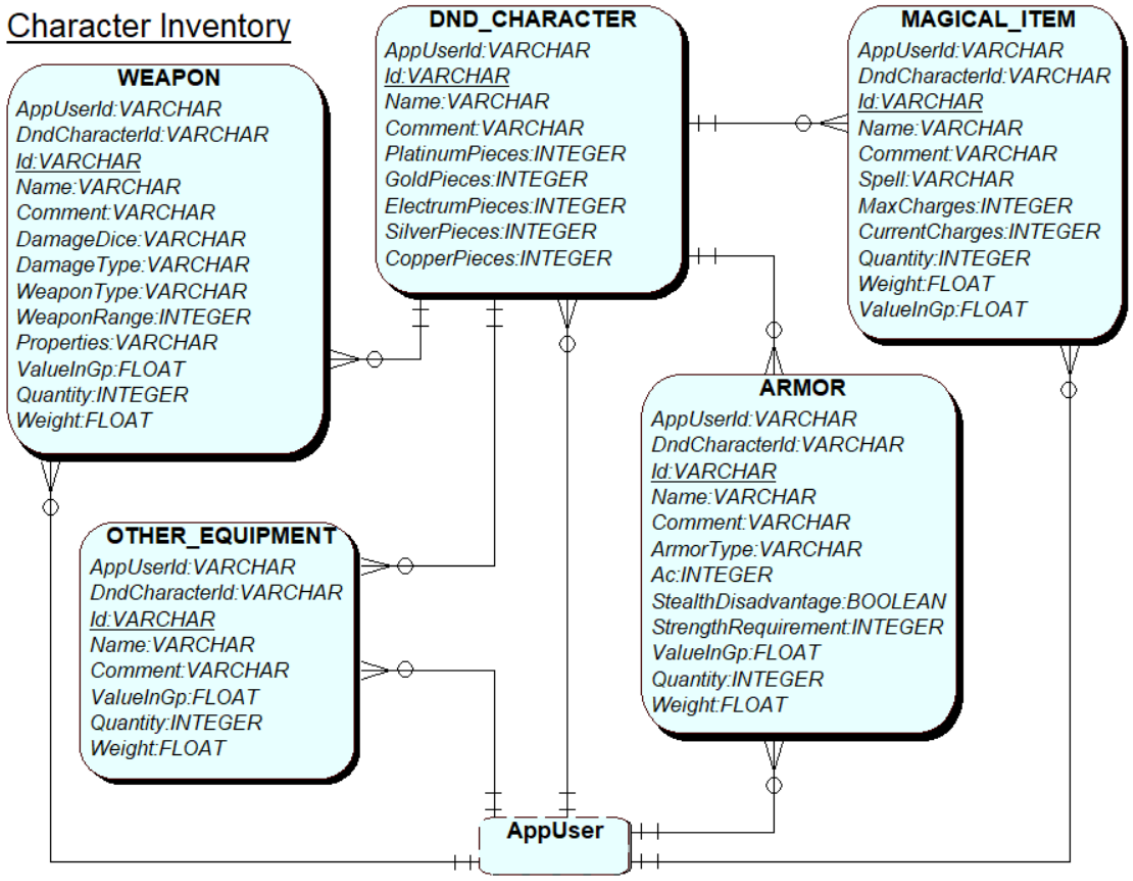
The goal of this project is to create an inventory manager application for a tabletop roleplaying game called Dungeons and Dragons (from now on DnD). In DnD the players go on different quests where they can battle mystical creatures for a few coins of gold. The world and the story are described to the players by a dungeon master.

According to most DnD players and dungeon masters, handling and managing a character's inventory can become a messy job. Usually an inventory is printed out on paper and managed from there during the game, but as time goes on the managing can get difficult.

The biggest downside to inventory management on paper is passing on inventory from one character to another due to a character's death. Bad handwriting and other such reasons can mean the loss of some notes that might become crucial at some point. That is why the author is also planning a far future feature in the application to pass on selected items from a character's inventory.

The scope of this project is to create a usable product for DnD inventory management. Dungeon masters with a lot of experience and other players will be consulted with regarding this work. The author will plan on testing the application in real life dungeons and dragons gaming sessions as it develops.

2 EDR Schema


Character Inventory
WEAPON
AppUserId:VARCHAR
DndCharacterId:VARCHAR
Id:VARCHAR
Name:VARCHAR
Comment:VARCHAR
DamageDice:VARCHAR
DamageType:VARCHAR
WeaponType:VARCHAR
WeaponRange:INTEGER
Properties:VARCHAR
ValueInGp:FLOAT
Quantity:INTEGER
Weight:FLOAT
DND_CHARACTER
AppUserId:VARCHAR
Id:VARCHAR
Name:VARCHAR
Comment:VARCHAR
PlatinumPieces:INTEGER
GoldPieces:INTEGER
ElectrumPieces:INTEGER
SilverPieces:INTEGER
CopperPieces:INTEGER
MAGICAL_ITEM
AppUserId:VARCHAR
DndCharacterId:VARCHAR
Id:VARCHAR
Name:VARCHAR
Comment:VARCHAR
Spell:VARCHAR
MaxCharges:INTEGER
CurrentCharges:INTEGER
Quantity:INTEGER
Weight:FLOAT
ValueInGp:FLOAT
ARMOR
AppUserId:VARCHAR
DndCharacterId:VARCHAR
Id:VARCHAR
Name:VARCHAR
Comment:VARCHAR
ArmorType:VARCHAR
Ac:INTEGER
StealthDisadvantage:BOOLEAN
StrengthRequirement:INTEGER
ValueInGp:FLOAT
Quantity:INTEGER
Weight:FLOAT
OTHER_EQUIPMENT
AppUserId:VARCHAR
DndCharacterId:VARCHAR
Id:VARCHAR
Name:VARCHAR
Comment:VARCHAR
ValueInGp:FLOAT
Quantity:INTEGER
Weight:FLOAT
AppUser

3 Application architecture

The application consists of several well bordered and clear layers that come together to create one working and usable application.

The main layers are the database, the back-end, and the front-end. Both the back-end and the front-end will be deployed using docker and MS Azure.

3.1 Database

The author has decided to use a Microsoft SQL server as a database. The access to the server was given to the author by their supervisor.

The main reason the author took advantage of this opportunity is because the supervisor was using the same database in their examples. The author found that it would be easier to solve problems if any occurred because the supervisor most likely already got rid of them.

3.2 Base packages

Several base projects that were used by one or more of the layers were published to nuget.org and then downloaded from there for usage.

The projects were put to a different solution and published from there. All the published projects had dependencies only from other nuget packages or from already published projects.

The publishing cleaned up the main solution folder which made the development easier. The author can use the published packages in other future solutions as well.

3.3 Domain

The domain layer defined the structure of the database and so it was redesigned quite a lot. The domain layer consists only of specific classes to define the database.

3.4 Data Access Layer

The data access layer (DAL) is responsible for connecting to the database and mapping the domain classes to actual database tables. It is also responsible for enabling the program to access the database.

The purpose of the DAL is to keep database queries separate from the rest of the program.

The DAL has so-called repositories which the author has used for designing and creating the main objects that will be mapped and passed on to the upper layer of the application that will eventually reach the front-end.

The author has mutated the main ‘character’ object to suit their needs better. Since the author knows that they will be using APIs, the author has made the decision to make the ‘domain’ character into two different ‘DAL’ objects to better suit the APIs.

The purpose of the first object is for a ‘get a specific character’ API that will return a character based on the ID of the character. The returned object shall have all the lists of all the items that the specific character has. It also has extra properties for a sum of the characters money, a sum for the total worth of the characters items and weight.

The purpose of the second object is for ‘get all characters’ API. The result of that will be a compact array of all the user’s characters. The character object in that array does not have all the details of the first object. Instead it shows the amount of all the items and a total sum of the characters money.

3.5 Business Logic Layer

The purpose of the business logic layer (BLL) is to enforce the required business rules to the data that DAL has allowed through.

The author has chosen to keep this layer to bare minimum because the author has not set down any business rules.

3.6 APIs

The APIs are the last layer for the back-end and they enable communication between back-end and front-end.

The author has created two APIs for data retrieval: ‘get all characters’ and ‘get specific character’. The output of these APIs was defined in the DAL layer.

The APIs also allow for data modifications. The author has created separate APIs for adding, updating, and deleting items.

3.7 Front-end

The user interface was made as a separate project and it communicates with the back-end using REST APIs.

The author has chosen to create the user interface in ReactJs because they already have quite a lot of experience with it.

4 Soft update

Soft update is a method of handling data changes (deleting, updating, and adding data) in a way where nothing is ever truly deleted or modified. It is required when handling data concerning legal matters, e.g. financial data.

The most problematic part that the author finds is maintaining the relationships between tables when updating data: avoiding cascading where possible and long search chains.

Integrating soft updating into databases also gives a lot of advantages when it comes to reading data. For example, all the changes (data creation, deletion and updating) are recorded and it is possible to check the state of the database at a certain point in time.

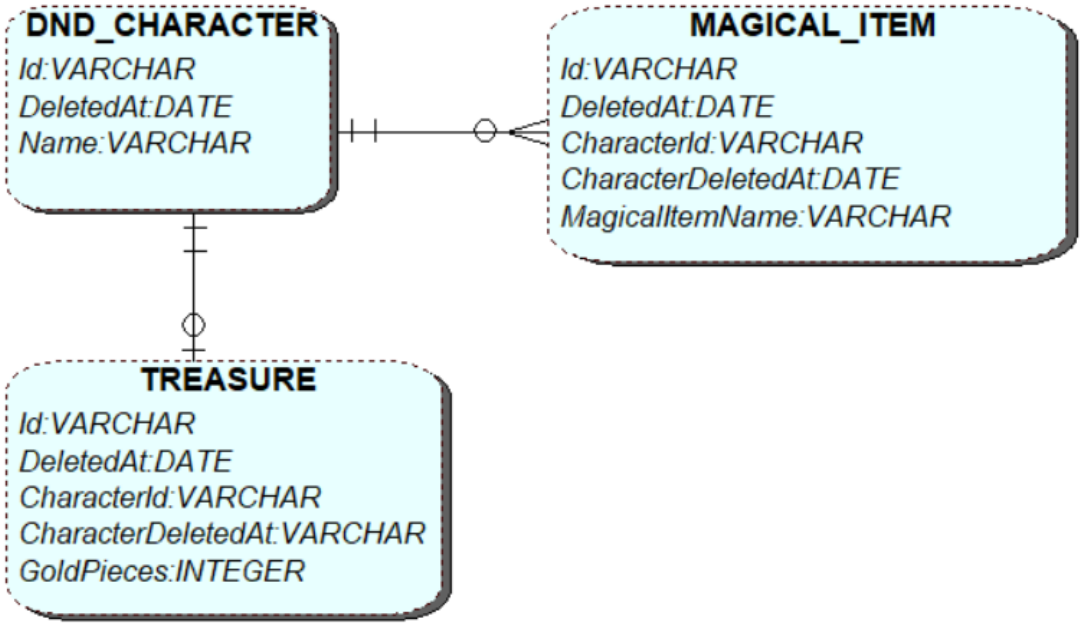
4.1 Soft update methods

There are many ways of implementing soft updating to databases. It also requires time and a lot of thinking about the logic. Most of the methods require extra column or columns of metadata to each table to keep track of the changes:

- a column that stores the timestamp of the time the entry was made (“created at”),
- a column that stores the timestamp of the time the entry was deleted (“deleted at”),
- a column that stores the original entry primary key or the previous *step* (“old ID”).

Another solution would be to unite the unique ID and the “deleted at” column to be the new composite primary key.

For the examples, the author will be using a simplified version of their database. The database depicts a situation, where the characters’ treasure is in a separate table and with a one-to-one relationship with the character table. All the primary keys will be composite keys made from “Id” and “DeletedAt” columns. All the foreign keys will be the characters’ “Id” and “DeletedAt”. The author has chosen not to use the “created at” column in the following examples.




```
-- The master table
CREATE TABLE DndCharacter (
    Id VARCHAR(36) NOT NULL,
    DeletedAt DATETIME NOT NULL,
    Name VARCHAR(128) NOT NULL,
    CONSTRAINT CharacterPK PRIMARY KEY (Id, DeletedAt)
)

-- The one to one relationship child table
CREATE TABLE Treasure (
    Id VARCHAR(36) NOT NULL,
    DeletedAt DATETIME NOT NULL,
    CharacterId VARCHAR(36),
    CharacterDeletedAt DATETIME,
    GoldPieces INT NOT NULL,
    CONSTRAINT TreasurePK PRIMARY KEY (Id, DeletedAt),
    CONSTRAINT CharactersTreasureFK FOREIGN KEY (CharacterId, CharacterDeletedAt) REFERENCES DndCharacter(Id, DeletedAt)
)

-- The one to many relationship child table
CREATE TABLE MagicalItem (
    Id VARCHAR(36) NOT NULL,
    DeletedAt DATETIME NOT NULL,
    CharacterId VARCHAR(36),
    CharacterDeletedAt DATETIME,
    MagicalItemName VARCHAR(128) NOT NULL,
    CONSTRAINT MagicalItemPK PRIMARY KEY (Id, DeletedAt),
    CONSTRAINT CharactersMagicalItemFK FOREIGN KEY (CharacterId, CharacterDeletedAt) REFERENCES DndCharacter(Id, DeletedAt)
)
```

Since “deleted at” column values cannot be null because it is part of the composite primary keys, the latest possible datetime that the server can handle is used. In this case it is 31th of December, 9999, at 23:59:59.99 o’clock. The maximum datetime is saved in a variable so that it can be used multiple times.

```
-- Declare the latest possible datetime in server for not deleted entities
DECLARE @MaxDateTime DATETIME = CAST('9999-12-31 23:59:59.99' AS DATETIME)
```

4.1.1 Single table

Updating a table with no relations to other tables is as simple as it can get because there is no worry with maintaining the referential integrity.

4.1.1.1 Creating

In this example, three entries will be inserted into the characters' table. For the unique ID, the built-in method of "NEWID()" is used and the previously declared maximum date is used.

```
-- Initial data, characters table
INSERT INTO DndCharacter (Id, DeletedAt, Name) VALUES (NEWID(), @MaxDateTime, 'Orc McOrcson')
INSERT INTO DndCharacter (Id, DeletedAt, Name) VALUES (NEWID(), @MaxDateTime, 'Fighty McChampion')
INSERT INTO DndCharacter (Id, DeletedAt, Name) VALUES (NEWID(), @MaxDateTime, 'Aadu Tihevarvas')
```

	Id	DeletedAt	Name
1	741DA228-5DFC-45ED-B657-C9AB3C928397	9999-12-31 23:59:59.990	Orc McOrcson
2	879EA8DE-F7C0-4FCB-90B6-3F4550387EE6	9999-12-31 23:59:59.990	Aadu Tihevarvas
3	A59E4C1D-A29B-4AEC-A74B-F2B6EF48FFEF	9999-12-31 23:59:59.990	Fighty McChampion

4.1.1.2 Deleting

If the entry is to be deleted, then the column "deleted at" would be updated to be the needed timestamp. In this example the character "Aadu Tihevarvas" will be deleted. For readability, the ID of the entry is found and saved to a variable, then the entry's "deleted at" column is updated to current time. For that, the built-in method "GETDATE()" is used.

```
-- Deleting an entry from the master table (no children)
DECLARE @Id VARCHAR(36)
SELECT @Id = Id FROM DndCharacter WHERE Name like 'Aadu Tihevarvas'
UPDATE DndCharacter SET DeletedAt = GETDATE() WHERE Id = @Id
```

	Id	DeletedAt	Name
1	741DA228-5DFC-45ED-B657-C9AB3C928397	9999-12-31 23:59:59.990	Orc McOrcson
2	879EA8DE-F7C0-4FCB-90B6-3F4550387EE6	2020-03-17 14:48:11.260	Aadu Tihevarvas
3	A59E4C1D-A29B-4AEC-A74B-F2B6EF48FFEF	9999-12-31 23:59:59.990	Fighty McChampion

4.1.1.3 Updating

If an entry is to be updated, a copy of said entry is made with the needed updates and the old entry is marked as deleted. In this example the author saves the ID of the

updated entry to a variable, marks the entry as deleted and then makes a copy with the updates. It is done in that particular order to avoid primary key conflicts.

```
-- Updating an entry from the master table (no children)
--     Get the Id
DECLARE @IdToBeUpdated VARCHAR(36)
SELECT @IdToBeUpdated = Id FROM DndCharacter WHERE Name like 'Fighty McChampion'

--     Mark the entry as deleted
UPDATE DndCharacter SET DeletedAt = GETDATE() WHERE Id = @IdToBeUpdated

--     Make a copy with the update
INSERT INTO DndCharacter (Id, DeletedAt, Name) VALUES (@IdToBeUpdated, @MaxDateTime, 'Human Fighty McChampion')
```

	Id	DeletedAt	Name
1	741DA228-5DFC-45ED-B657-C9AB3C928397	9999-12-31 23:59:59.990	Orc McOrcson
2	879EA8DE-F7C0-4FCB-90B6-3F4550387EE6	2020-03-17 14:48:11.260	Aadu Tihevarvas
3	A59E4C1D-A29B-4AEC-A74B-F2B6EF48FFEF	2020-03-17 15:06:19.420	Fighty McChampion

	Id	DeletedAt	Name
1	741DA228-5DFC-45ED-B657-C9AB3C928397	9999-12-31 23:59:59.990	Orc McOrcson
2	879EA8DE-F7C0-4FCB-90B6-3F4550387EE6	2020-03-17 14:48:11.260	Aadu Tihevarvas
3	A59E4C1D-A29B-4AEC-A74B-F2B6EF48FFEF	2020-03-17 15:06:19.420	Fighty McChampion
4	A59E4C1D-A29B-4AEC-A74B-F2B6EF48FFEF	9999-12-31 23:59:59.990	Human Fighty McChampion

4.1.2 One to one

In a one-to-one relationship one entry can have one child or no children, and a child must have a parent.

4.1.2.1 Creating

Adding data to the master table is exactly like adding data in a single table. Adding to the child table requires the primary key of the master table entry. For readability, the author has saved the characters primary key components to variables.

```
-- Adding stuff to Mr Orc McOrcson
DECLARE @CharacterId VARCHAR(36)
DECLARE @CharacterDeletedAt DATETIME
SELECT @CharacterId = Id FROM DndCharacter WHERE Name like 'Orc McOrcson';
SELECT @CharacterDeletedAt = DeletedAt FROM DndCharacter WHERE Name like 'Orc McOrcson';

-- Initial data, Treasure table
INSERT INTO Treasure (Id, DeletedAt, CharacterId, CharacterDeletedAt, GoldPieces) VALUES (NEWID(), @MaxDateTime, @CharacterId, @CharacterDeletedAt, 666)
```

	Id	DeletedAt	Name
1	741DA228-5DFC-45ED-B657-C9AB…	9999-12-31 23:59:59.990	Orc McOrcson
2	879EA8DE-F7C0-4FCB-90B6-3F45…	2020-03-17 14:48:11.260	Aadu Tihevarvas
3	A59E4C1D-A29B-4AEC-A74B-F2B6…	2020-03-17 15:06:19.420	Fighty McChampion
4	A59E4C1D-A29B-4AEC-A74B-F2B6…	9999-12-31 23:59:59.990	Human Fighty McChampion

	Id	DeletedAt	CharacterId	CharacterDeletedAt	GoldPieces
1	8B6D2EEA-69E7-4729-8757-0954…	9999-12-31 23:59:59.990	741DA228-5DFC-45ED-B657-C9AB…	9999-12-31 23:59:59.990	666

4.1.2.2 Deleting

There are two choices when it comes to deleting:

- deleting the child,
- deleting the master.

When deleting the child, the “deleted at” column will be updated to current datetime and that will be it.

When deleting the master, a question arises: if a master entity is deleted, should the child/children of this entity also be deleted (cascade delete)? Most commonly the answer is yes but this is a matter of preference and need.

In the next example, the author has chosen the use of a cascade delete. Since the deletion requires the change of the primary key, the foreign key of the child table also requires an update. The author has added an extra entry to the tables (character “Kitty McCat” with 13 gold pieces in the treasure table).

	Id	DeletedAt	Name
1	6CA610C1-1400-498F-9568-EE17…	9999-12-31 23:59:59.990	Orc McOrcson
2	8B776997-58D6-4650-AF7F-2063…	2020-03-18 11:31:47.587	Aadu Tihevarvas
3	9B14606B-5213-4752-95DC-F4E1…	9999-12-31 23:59:59.990	Kitty McCat
4	C7CEEE3B-3C18-4BE1-9F0C-037A…	2020-03-18 11:31:47.597	Fighty McChampion
5	C7CEEE3B-3C18-4BE1-9F0C-037A…	9999-12-31 23:59:59.990	Human Fighty McChampion

	Id	DeletedAt	CharacterId	CharacterDeletedAt	GoldPieces
1	AA074136-DF68-4576-B620-1812…	9999-12-31 23:59:59.990	6CA610C1-1400-498F-9568-EE17…	9999-12-31 23:59:59.990	666
2	C5070A18-98B0-4F3D-8A6F-A297…	9999-12-31 23:59:59.990	9B14606B-5213-4752-95DC-F4E1…	9999-12-31 23:59:59.990	13

It was at this point that the author realized an error in the database. There was no way of updating the key connecting the two tables. To fix that, the foreign key constraint must be updated to cascade in the case of update. First the old constraint must be dropped, and only then can the cascading key be entered.

```
-- Update Treasures' table by adding updating cascade
ALTER TABLE Treasure DROP CONSTRAINT CharactersTreasureFK
ALTER TABLE TREASURE ADD CONSTRAINT CharactersTreasureFK FOREIGN KEY (CharacterId, CharacterDeletedAt) REFERENCES DndCharacter(Id, DeletedAt) ON UPDATE CASCADE
```

To commence the soft delete of the master, the masters ID and the current datetime is saved to variables. Saving the datetime to a variable is made in order to avoid possible time lag between updates.

```
-- Soft deleting one-to-one (master entity)
--     Get Id of the master entity
DECLARE @IdMasterToBeDeleted VARCHAR(36)
SELECT @IdMasterToBeDeleted = Id FROM DndCharacter WHERE Name like 'Kitty McCat'

--     Save current time to variable
DECLARE @CurrentTime DATETIME
SELECT @CurrentTime = GETDATE()

--     Marking the master entity as deleted
UPDATE DndCharacter SET DeletedAt = @CurrentTime WHERE Id = @IdMasterToBeDeleted

--     Update the child's foreign key, mark it as deleted
UPDATE Treasure SET DeletedAt = @CurrentTime WHERE CharacterId = @IdMasterToBeDeleted
```

	Id	DeletedAt	Name
1	6CA610C1-1400-498F-9568-EE17…	9999-12-31 23:59:59.990	Orc McOrcson
2	8B776997-58D6-4650-AF7F-2063…	2020-03-18 11:31:47.587	Aadu Tihevarvas
3	9B14606B-5213-4752-95DC-F4E1…	2020-03-18 12:34:10.817	Kitty McCat
4	C7CEEE3B-3C18-4BE1-9F0C-037A…	2020-03-18 11:31:47.597	Fighty McChampion
5	C7CEEE3B-3C18-4BE1-9F0C-037A…	9999-12-31 23:59:59.990	Human Fighty McChampion

	Id	DeletedAt	CharacterId	CharacterDeletedAt	GoldPieces
1	AA074136-DF68-4576-B620-1812…	9999-12-31 23:59:59.990	6CA610C1-1400-498F-9568-EE17…	9999-12-31 23:59:59.990	666
2	C5070A18-98B0-4F3D-8A6F-A297…	2020-03-18 12:34:10.817	9B14606B-5213-4752-95DC-F4E1…	2020-03-18 12:34:10.817	13

Kitty McCat has been successfully deleted along with his treasure.

4.1.2.3 Updating

Just as deleting, there are two choices when it comes to update:

- update the child,
- update the master.

Updating in one-to-one relationships is easier than deleting because there is no need for cascade update. Since the cascade update was added to the foreign key, the updates can be added just like in a single table.

```
-- Soft updating one-to-one (master entity)
--     Declare the latest possible datetime in server for not deleted entities
DECLARE @MaxDateTime DATETIME = CAST('9999-12-31 23:59:59.99' AS DATETIME)

--     Get the Id
DECLARE @IdMasterToBeUpdated VARCHAR(36)
SELECT @IdMasterToBeUpdated = Id FROM DndCharacter WHERE Name like 'Orc McOrcson'

--     Mark the entry as deleted
UPDATE DndCharacter SET DeletedAt = GETDATE() WHERE Id = @IdMasterToBeUpdated

SELECT 'ENTRY UPDATE ("Orc McOrcson"): Entry is marked as deleted'
SELECT * FROM DndCharacter

--     Make a copy with the update
INSERT INTO DndCharacter (Id, DeletedAt, Name) VALUES (@IdMasterToBeUpdated, @MaxDateTime, 'Orc McSkullbasher')
```

	Id	DeletedAt	Name
1	6CA610C1-1400-498F-9568-EE17…	2020-03-18 15:27:43.213	Orc McOrcson
2	6CA610C1-1400-498F-9568-EE17…	9999-12-31 23:59:59.990	Orc McSkullbasher
3	8B776997-58D6-4650-AF7F-2063…	2020-03-18 11:31:47.587	Aadu Tihevarvas
4	9B14606B-5213-4752-95DC-F4E1…	2020-03-18 12:34:10.817	Kitty McCat
5	C7CEEE3B-3C18-4BE1-9F0C-037A…	2020-03-18 11:31:47.597	Fighty McChampion
6	C7CEEE3B-3C18-4BE1-9F0C-037A…	9999-12-31 23:59:59.990	Human Fighty McChampion

4.1.3 One to many

In a one-to-many relationship a master entity can have one, many or no children and a child entity must have a single parent.

4.1.3.1 Creating

Adding data to tables is no different than adding to single or one-to-one tables.

```
-- Adding stuff to Mr Orc
DECLARE @CharacterId VARCHAR(36)
DECLARE @CharacterDeletedAt DATETIME
SELECT @CharacterId = Id FROM DndCharacter WHERE Name like 'Orc McSkullbasher';
SELECT @CharacterDeletedAt = DeletedAt FROM DndCharacter WHERE Name like 'Orc McSkullbasher';

-- Initial data, Magical items' table
INSERT INTO MagicalItem (Id, DeletedAt, CharacterId, CharacterDeletedAt, MagicalItemName) VALUES (NEWID(), @MaxDateTime, @CharacterId, @CharacterDeletedAt, 'Magic missile shooting talking skull')
INSERT INTO MagicalItem (Id, DeletedAt, CharacterId, CharacterDeletedAt, MagicalItemName) VALUES (NEWID(), @MaxDateTime, @CharacterId, @CharacterDeletedAt, 'Furry wand')
```

	Id	DeletedAt	Name
1	6CA610C1-1400-498F-9568-EE17…	2020-03-18 15:27:43.213	Orc McOrcson
2	6CA610C1-1400-498F-9568-EE17…	9999-12-31 23:59:59.990	Orc McSkullbasher
3	8B776997-58D6-4650-AF7F-2063…	2020-03-18 11:31:47.587	Aadu Tihevarvas
4	9B14606B-5213-4752-95DC-F4E1…	2020-03-18 12:34:10.817	Kitty McCat
5	C7CEEE3B-3C18-4BE1-9F0C-037A…	2020-03-18 11:31:47.597	Fighty McChampion
6	C7CEEE3B-3C18-4BE1-9F0C-037A…	9999-12-31 23:59:59.990	Human Fighty McChampion

	Id	DeletedAt	CharacterId	CharacterDeletedAt	MagicalItemName
1	7F7E1613-CE25-4308-8912-B8FC…	9999-12-31 23:59:59.990	6CA610C1-1400-498F-9568-EE17…	2020-03-18 15:27:43.213	Magic missile shooting talking skull
2	DCFDC719-5A7F-4859-ABEE-FA6E…	9999-12-31 23:59:59.990	6CA610C1-1400-498F-9568-EE17…	2020-03-18 15:27:43.213	Furry wand

4.1.3.2 Deleting

Deleting from a table with a one-to-many relationship has similar problems with one-to-one table: cascade delete. Magical items' table's foreign key must be updated the same way treasure table was.

```
-- Update Magical items' table by adding updating cascade
ALTER TABLE MagicalItem DROP CONSTRAINT CharactersMagicalItemFK
ALTER TABLE MagicalItem ADD CONSTRAINT CharactersMagicalItemFK FOREIGN KEY (CharacterId, CharacterDeletedAt) REFERENCES DndCharacter(Id, DeletedAt) ON UPDATE CASCADE
```

Since the database has an updated version of the "Orc" character, an extra condition must be added to the character update statement. No additional children deletion statements must be added because all the children have the needed conditions.

```
-- Soft deleting one-to-many (master entity)
DECLARE @MaxDateTime DATETIME = CAST('9999-12-31 23:59:59.99' AS DATETIME)

--      Get Id of the master entity
DECLARE @IdManyMasterToBeDeleted VARCHAR(36)
SELECT @IdManyMasterToBeDeleted = Id FROM DndCharacter WHERE Name like 'Orc McSkullbasher'

--      Save current time to variable
DECLARE @CurrentTime DATETIME
SELECT @CurrentTime = GETDATE()

--      Marking the master entity as deleted
UPDATE DndCharacter SET DeletedAt = @CurrentTime WHERE Id = @IdManyMasterToBeDeleted AND DeletedAt = @MaxDateTime

--      Update the children's foreign keys, mark the children as deleted
UPDATE MagicalItem SET DeletedAt = @CurrentTime WHERE CharacterId = @IdManyMasterToBeDeleted
```

	Id	DeletedAt	Name
1	6CA610C1-1400-498F-9568-EE17…	2020-03-18 15:27:43.213	Orc McOrcson
2	6CA610C1-1400-498F-9568-EE17…	2020-03-18 15:52:48.897	Orc McSkullbasher
3	8B776997-58D6-4650-AF7F-2063…	2020-03-18 11:31:47.587	Aadu Tihevarvas
4	9B14606B-5213-4752-95DC-F4E1…	2020-03-18 12:34:10.817	Kitty McCat
5	C7CEEE3B-3C18-4BE1-9F0C-037A…	2020-03-18 11:31:47.597	Fighty McChampion
6	C7CEEE3B-3C18-4BE1-9F0C-037A…	9999-12-31 23:59:59.990	Human Fighty McChampion

	Id	DeletedAt	CharacterId	CharacterDeletedAt	MagicalItemName
1	7F7E1613-CE25-4308-8912-B8FC…	2020-03-18 15:52:48.897	6CA610C1-1400-498F-9568-EE17…	2020-03-18 15:52:48.897	Magic missile shooting talki…
2	DCFDC719-5A7F-4859-ABEE-FA6E…	2020-03-18 15:52:48.897	6CA610C1-1400-498F-9568-EE17…	2020-03-18 15:52:48.897	Furry wand

4.1.3.3 Updating

Updating the one-to-many master entity is exactly like in the previous update examples.

4.2 Author’s opinion

Regarding the demonstrated examples and author’s experience, the best way to do soft updates is to have composite primary and foreign keys and to have cascade update enabled in foreign keys. Also, it would be smart to have an extra column “Old Id” in the tables, which points to the original entry. In the examples there was no such column because there was no need for it and readability might had been compromised.

5 Repositories

The problem with big projects and solutions is that the code seems to repeat itself. Accessing data of different entities, same properties in different classes etc. When a change is required, the code needs to be changed in a lot of different places. Not only is the code repetitive, it's difficult to test the code and add new parts.

The main purpose of repository pattern is to separate data access logic and business logic. The repository pattern makes use of interfaces, inheritance, and dependency injections which reduces duplicate code drastically. Implementing a repository pattern is a matter of taste and need.

The repository pattern makes development easier. If a developer wanted to know about a repository, they only need to look at the interface of the repository where all the properties are declared.

The author of this project has chosen to use the repository pattern. The author has put all the unique IDs and comments into a universal interface. The author has also decided to put one extra interface between the universal interface and the class entities in case there is a need to add more universal interfaces. If there is no need for it in the end, the extra interface will be removed in the final version.

5.1 Comparing repository pattern with DAO

Data access object (DAO) pattern is different from repositories but they share the same purpose: separation of different application parts that can and should develop independently. While DAO is an abstraction of data persistence, repository pattern is an abstraction of a collection of objects. DAO is closer to the database and repository is closer to domain.

When it comes to choosing between DAO and repository patterns, it is a matter of taste and need. DAO is more flexible and generic, and repository is more specific and restrictive to a certain type of object.

6 Summary

The purpose of this project was to make a usable Dungeons and Dragons inventory management application with a clean a separated architecture. The author, in their opinion, has managed to succeed in doing so.

Due to the conditions that were out of the authors hands, real life testing has been delayed. The author had to accept just themself and another local human who has no experience with role playing tabletop games as the only test subjects.

Regarding the initial application plan, the author could not finish the inventory sharing feature but intends to add it if more people start using the application.

Overall the author is pleased with the outcome and will use their creation in future Dungeons and Dragons games.