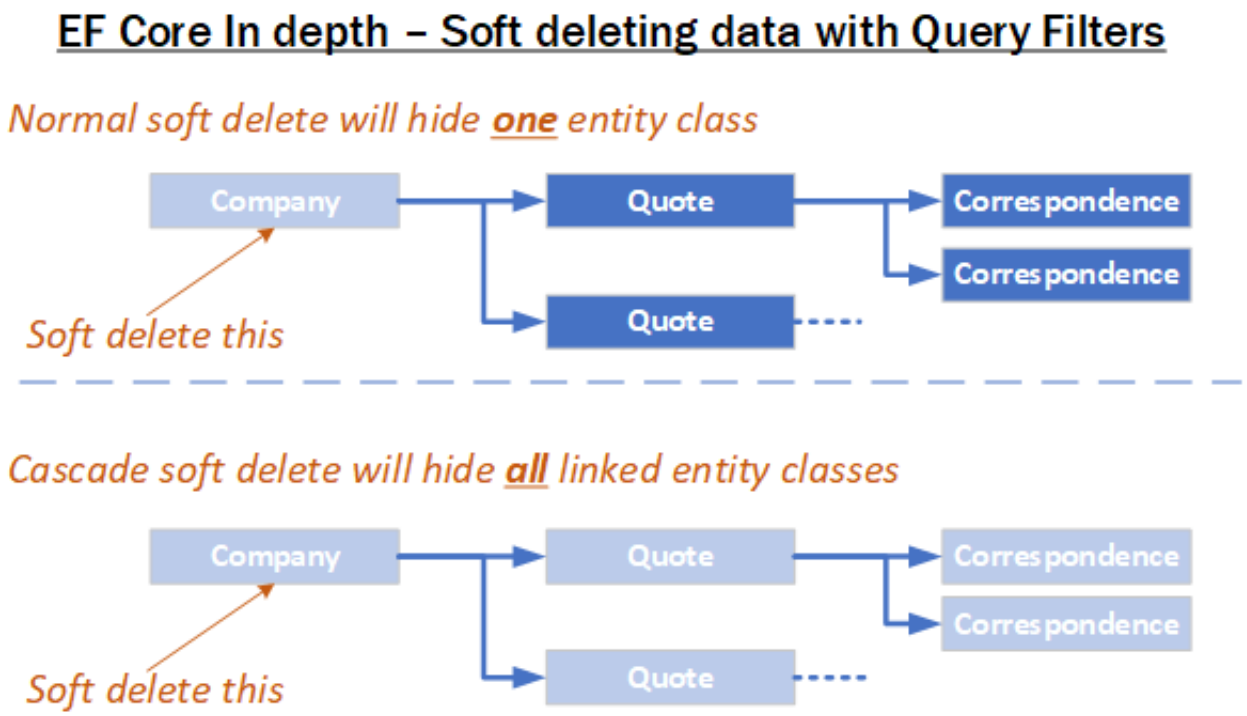


EF Core In depth – Soft deleting data with Global Query Filters

Last Updated: February 20, 2021 | Created: July 2, 2020

This article is about a way to seemingly delete data, but in fact EF Core hides it for you and you can get it back if you need to. This type of feature is known as *soft delete* and it has many good features, and the issues to be aware of too. In this article I use EF Core to implement the normal soft delete and a more powerful (and complex) cascade soft delete. Along the way I give you tips on how write reusable code to speed up your development of a soft delete solution.

I do assume you know EF Core, but I start with a look at using EF Core to make

sure we have the basics of deleting and soft deleting covered before looking at solutions.

This article is part of a "EF Core In depth" series. Here is the current list of articles in this series:

- EF Core In depth – what happens when EF Core reads from the database?
- EF Core In depth – what happens when EF Core writes to the database?
- EF Core In depth – Soft deleting data with Global Query Filters **(this article)**
- Introducing the EfCore.SoftDeleteServices library to automate soft deletes – **companion to this article**
- EF Core In depth – Tips and techniques for configuring EF Core

Other older articles in this series are

- Handling Entity Framework Core migrations: creating a migration – Part 1
- Handling Entity Framework Core migrations: applying a migration – Part 2
- NEW! Introducing the EfCore.SoftDeleteServices library to automate soft deletes

This "EF Core In depth" series is inspired by what I found while updating my book "Entity Framework Core in Action" to cover EF Core 5. I am also added a LOT of new content from my experiences of working with EF Core on client applications over the last 2½ years.

> *NOTE: There is a GitHub repo at https://github.com/JonPSmith/EfCore.SoftDeleteServices that contains all the code used in this article.*

UPDATE – new library

I have just released the library EfCore.SoftDeleteServices which I talk about in this article. See the new article Introducing the EfCore.SoftDeleteServices library to automate soft deletes to read how this library helps you implement soft delete in your applications.

TL;DR – summary

- You can add a soft delete feature to your EF Core application using Global Query Filters (referred to as *Query Filters* for now on).
- The main benefits of using soft delete in your application are inadvertent deletes can be restored and history is preserved.
- There are three parts to adding the soft delete feature to your application
  - Add a new soft delete property to every entity class you want to soft delete.
  - Configure the Query Filters in your application's DbContext
  - You create code to set and reset the soft delete property.
- You can combine soft delete with other uses of Query Filters, like multi-tenant uses but you need to be more careful when you are looking for soft deleted entries.
- Don't soft delete a one-to-one entity class as it can cause problems.
- For entity classes that has relationships you need to consider what should happen to the dependant relationships when the top entity class is soft deleted.
- I introduce a way to implement a cascade soft delete that works for entities where you need its dependant relationships soft deleted too.

Setting the scene – why soft delete is such a good idea

When you hard delete (I use the term *hard delete* from now on, so its obvious what sort of delete I'm talking about), then it gone from your database. Also, hard deleting might also hard delete rows that rely on the row you just hard deleted (known as *dependant relationships*). And as the saying says "When it's gone, then it's gone" – no getting it back unless you have a backup.

But nowadays we are more used to "I deleted it, but I can get it back" – On windows it's in the recycle bin, if you deleted some text in an editor you can get it back with ctrl-Z, and so on. Soft delete is EF Core's version of windows recycle bin – the *entity class* (the term for classes mapped to the database via EF Core) is gone from normal usage, but you can get it back.

Two of my client's applications used soft delete extensively. Any "delete" the normal user did set the soft delete flag, but an admin user could reset the soft delete

flag to get the item back for the user. In fact, one of my clients used the terms "delete" for a soft delete and "destroy" for a hard delete. The other benefit of keeping soft-deleted data is history – you can see what changed in the past even its soft deleted. Most client's keeps soft deleted data in the database for some time and only backup/remove that data many months (years?) later.

You can implement the soft delete feature using EF Core *Query Filters*. Query Filters are also used for multi-tenant systems, where each tenant's data can only be accessed by users who belong to the same tenant. This means EF Core Query Filters are designed to be very secure when it comes to hiding things – in this case data that has been soft deleted.

I should also say there are some down sides of using soft delete. The main one is performance – an extra, hidden SQL WHERE clause is included in every query of entity classes using soft delete.

There is also a difference between how soft delete handles dependant relationships when compared with hard delete. By default, if you soft delete an entity class then its dependant relationships are NOT soft deleted, whereas a hard delete of an entity class would normally delete the dependant relationships. This means if I soft delete a Book entity class then the Book's Reviews will still be visible, which might be a problem in some cases. At the end of this article I show you how to handle that and talk about a prototype library that can do cascade soft deletes.

Adding soft delete to your EF Core application

In this section I'm going to go through each for the steps to add soft delete to your application

1. Add soft delete property to your entity classes that need soft delete
2. Add code to your DbContext to apply a query filter to these entity classes
3. How to set/reset Soft Delete

In the next sections I describe these stages in detail. I assume a typical EF Core class with normal read/write properties, but you can adapt it to other entity class styles, like Domain-Driven Design (DDD) styled entity classes.

The Reformed Programmer
I am a freelance .NET Core back-end developer

1. Adding soft delete property to your entity classes

For the standard soft delete implementation, you need a boolean flag to control soft delete. For instance, here is a Book entity with a SoftDeleted property highlighted.

```
public class Book : ISoftDelete
{
    public int BookId { get; set; }
    public string Title { get; set; }
    //… other properties left out to focus on Soft delete

    public bool SoftDeleted { get; set; }
}
```

You can tell by its name, SoftDeleted, that if it is true, then its soft deleted. This means when you create a new entity it is not soft deleted.

You other thing I added was an ISoftDelete interface to the Book class (line 1). This interface says the class must have a public SoftDeleted property which can be read and written to. This interface is going to make it much easier to configure the delete query filters in your DbContext.

2. Configuring the soft delete query filters in your DbContext

You must tell EF Core which entity classes needs a query filter and provide a query which will be true if you want it to be seen. You can do this manually using the following code in your DbContext – see highlighted line in the following listing.

```
public class EfCoreContext : DbContext
{
    public EfCoreContext(DbContextOptions<EfCoreContext> op
        : base(options)
    {}

    //Other code left out to focus on Soft delete

    protected override OnModelCreating(ModelBuilder modelB
    {
        //Other configuration left out to focus on Soft del

        modelBuilder.Entity<Book>().HasQueryFilter(p => !p.
    }
}
```

That's fine but let me show you a way to automate adding query filters. This uses

1. The modelBuilder.Model.GetEntityTypes() feature available in the OnModelCreating method
2. A little bit of generic magic to create the correct query filter

Here are two part:

1. Automating the configuring of the soft delete query filters

The OnModelCreating method in your DbContext is where you can configure EF Core via what are known as Fluent API configuration commands – you saw that in

We use cookies to ensure that we give you the best experience on our website and monitoring traffic. If you continue to use this site we will assume that you are happy with it.

Ok

in turn. You will see a test to see if the entity class implements the ISoftDelete interface and if it does it calls a extension method I created to configure a query filter with the correct soft delete filter.

```
protected override void OnModelCreating(ModelBuilder mode
{
    //other manual configurations left out

    foreach (var entityType in modelBuilder.Model.GetEntit
    {
        //other automated configurations left out
        if (typeof(ISoftDelete).IsAssignableFrom(entityTyp
        {
            entityType.AddSoftDeleteQueryFilter();
        }
    }
}
```

2. Creating the AddSoftDeleteQueryFilter extension method

There are many configurations you can apply directly to the type that the GetEntityTypes method returns but setting up the Query Filter needs a bit more work. That's because LINQ query in the Query Filter needs the type of the entity class to create the correct LINQ expression. For this I created a small extension class that can dynamically create the correct LINQ expression to configure the Query Filter

and adds a index to that property.

```
public static class SoftDeleteQueryExtension
{
    public static void AddSoftDeleteQueryFilter(
        this IMutableEntityType entityData)
    {
        var methodToCall = typeof(SoftDeleteQueryExtension
            .GetMethod(nameof(GetSoftDeleteFilter),
                BindingFlags.NonPublic | BindingFlags.Stati
            .MakeGenericMethod(entityData.ClrType);
        var filter = methodToCall.Invoke(null, new object
        entityData.SetQueryFilter((LambdaExpression)filter)
        entityData.AddIndex(entityData.
             FindProperty(nameof(ISoftDelete.SoftDeleted)))
    }

    private static LambdaExpression GetSoftDeleteFilter<TE
        where TEntity : class, ISoftDelete
    {
        Expression<Func<TEntity, bool>> filter = x => !x.So
        return filter;
    }
}
```

I really like this because I a) saves me time, and b) I can't forget to configure a query filter.

3. How to set/reset Soft Delete

Setting the SoftDeleted property to true is easy – the user picks an entry and clicks "Soft Delete", which send back the entities primary key. Then your code to implement that is.

```
var entity = context.Books.Single(x => x.BookId == id);
entity.SoftDeleted = true;
context.SaveChanges();
```

Resetting the SoftDeleted property is a little bit more complex. First you most likely want to show the user a list of JUST the soft deleted entities – think of it as showing the trash can/recycle bin for an individual entity class type, e.g. Book. To do this need to add the IgnoreQueryFilters method to your query which means you will get ALL the entities, ones that aren't soft deleted and ones that are, but you then pick out the ones where the SoftDeleted property is true.

```
var softDelEntities = _context.Books.IgnoreQueryFilters()
    .Where(x => x.SoftDeleted)
```

```
    .ToList();
```

And when you get a request to reset the SoftDeleted property this typically contains the entity classes primary key. To load this entry you need include the IgnoreQueryFilters method in your query to get the entity class you want to reset.

```
var entity = context.Books.IgnoreQueryFilters()
     .Single(x => x.BookId == id);
entity.SoftDeleted = false;
context.SaveChanges();
```

Things to be aware of if you use Soft delete

First, I should say that Query Filters are very secure, by that I mean if the query filter returns false then that specific entity/row won't be returned in a query, a Find, an Include of a relationship etc. You can get around it by using direct SQL, but other than that EF Core is going to hide things that you soft delete.

But there are a couple of things you do need to be aware of.

Watch out for mixing soft delete with other Query Filter usages

Query Filters are great for soft delete, but Query Filters are even better for controlling access to groups of data. For instance, say you wanted to build a web application that to provide a service, like payroll, to lots of companies. In that case you need make sure that company "A" couldn't see company "B" data, and vis versa. This type of system is called a *multi-tenant application*, and Query Filters are a perfect fit for this.

> *NOTE: See my article Part 2: Handling data authorization in ASP.NET Core and Entity Framework Core for using query filters to control access to data.*

The problem is you are only allowed one query filter per entity type, so if you want to use soft delete with a multi-tenant system then you must combine both parts to form the query filter – here is an example of what the query filter might look like

```
modelBuilder.Entity<MyEntity>()
    .HasQueryFilter(x => !x.SoftDeleted
                         && x.TenantId == currentTenantId);
```

That work fine, but when you use the IgnoreQueryFilters method, say to reset a soft deleted flag, then it ignores the whole query filter, including the multi-tenant part. So, if you're not careful you could show multi-tenant data too!

The answer is to build yourself an application-specific IgnoreSoftDeleteFilter method something like this.

```
public static IQueryable<TEntity> IgnoreSoftDeleteFilter<TE
    this IQueryable<TEntity> baseQuery, string currentTenan
    where TEntity : class, ITenantId
{
    return baseQuery.IgnoreQueryFilters()
        .Where(x => x.TenantId == currentTenantId)
}
```

This ignores all the filters and then add back the multi-tenant part of the filter. That will make it much easier to safely handle showing/resetting soft deleted entities

Don't soft delete a one-to-one relationship

I was called in to help on a very interesting client system that used soft delete on every entity class. My client had found that you really shouldn't soft delete a one-to-one relationship. The problem he found was if you soft delete a one-to-one relationship and try to add a replacement one-to-one entity, then it fails. That's because a one-to-one relationship has a unique foreign key and that is already set by the soft deleted entity so, at the database level, you just can't provide another one-to-one relationship because there is one already.

One-to-one relationships are rare, so it might not be a problem in your system. But if you really need to soft delete a one-to-one relationship, then I suggest turn it into a one-to-many relationship where you make sure only one of the entities has a soft delete turned off, which I cover in the next problem area.

Handling multiple versions where some are soft deleted

There are business cases where you might create an entity, then soft delete it, and then create a new version. For example, say you were creating invoice for order 1234, then you are told the order has been stopped, so you soft delete it (that way you keep the history). Then later someone else (who doesn't know about the soft deleted version) is told to create an invoice for 1234. Now you have two versions of the invoice 1234. For something like an invoice that could cause a problem business-wise, especially if someone reset the soft deleted version.

You have a few ways to handle this:

- Add a LastUpdated property of type DateTime to your invoice entity class and the latest, not soft-deleted, entry is the one to use.
- Each new entry has a version number, so in our case the first invoice wold be 1234-1 and the section would be 1234-2. Then, like the LastUpdated version, the invoice with the highest version number, and is not soft deleted, is the one to use.
- Make sure there is only one not soft-deleted version by using a *unique filtered index*. This works by creating a unique index for all entries that aren't soft deleted, which means you would get an exception if you tried to reset a soft-deleted invoice but there was an existing non-soft deleted invoice already there. But at the same time, you could have lots of soft-deleted version for your history. Microsoft SQL Server RDBMS, PostgreSQL RDBMS, SQLite RDBMS have this feature (PostgreSQL and SQLite call it *partial indexes*) and I am told you can something like this in MySQL too. The code below is the SQL Server version of a filtered unique index.

```
CREATE UNIQUE INDEX UniqueInvoiceNotSoftDeleted
ON [Invoices] (InvoiceNumber)
WHERE SoftDeleted = 0
```

> *NOTE: For handling the exception that would happen with the unique index issue see my article called "Entity Framework Core – validating data and catching SQL errors" which shows you how to convert a SQL exception into a user-friendly error string.*

What about relationships?

Up to now we have been looking at soft deleting/resetting a single entity, but EF Core is all about relationships. So, what should I do about any relationships linked to the entity class that you just soft deleted? To help us, lets look at two different relationships that have different business needs.

Relationship example 1 – A Book with its Reviews

In my book "Entity Framework Core in Action" I build a super-simple book selling web site with books, author, reviews etc. And in that application, I can soft delete a Book. It turns out that once I delete the Book then there really isn't another way to get to the Reviews. So, in this case I don't have to worry about the Reviews of a soft deleted book.

But to make things interesting in chapter 5, which is about using EF Core with ASP.NET Core, I added a background task that counts the number of reviews. Here is the code I wrote to count the Reviews

```
var numReviews = await context.Set<Review>().CountAsync();
```

This, of course, gave the same count irrespective of whether the Book is soft deleted, which is different to what happens if I hard deleted the Book (because that would also delete the book's Review). I cover how to get around this problem later.

Relationship example 2 – A Company with its Quotes

In this example I have many companies that I sell to and each Company has set of Quotes we sent to that company. This is the same one-to-many relationship that the Book/Reviews has, but in this case, we have a list of companies and AND a separate list of Quotes. So, if I soft delete a Company then all the Quotes attached to that company should be soft deleted too.

I have come up with three useful solutions to both soft delete relationships examples I have just described.

Solution 1 – do nothing because it doesn't matter

Sometimes it doesn't matter that you soft deleted something, and its relationships are still available. Until I added the background task that counts Reviews my application worked fine if I soft deleted a book.

Solution 2 – Use the Aggregates/Root approach

The solution to the background task Reviews count I used was to apply a Domain-Driven Design (DDD) approach called Aggregate. This says a that you get grouping of entities that work together, in this case the Book, Review, and the BookAuthor linking table to the Author. In a group like this there is a *Root* entity, in this case the Book.

What Eric Evans, who is the person that define DDD, says is you should always access the aggregates via the Root aggregate. There are lots of DDD reasons for saying that, but in this case, it also solves our soft delete issue, as if I only get the Reviews through the Book then when it is soft deleted then the Reviews count is gone. So, the code below is the replacement to go in background task Reviews count

```
var numReviews = await context.Books
                    .SelectMany(x => x.Reviews).CountAsync();
```

You could also do a version of the review count query to list the Quotes via the Company, but there is another option – mimicking the way that database handles cascade deletes, which I cover next.

Solution 3 – mimicking the way that cascade deletes works

Databases have a delete setting called CASCADE, and EF Core has two DeleteBehaviours, Cascade and ClientCascade. These behaviours causes the hard delete of a row to also hard delete any rows that rely on that row. For instance, in my book-selling application the Book is what is called the *principal entity* and the Review, and the BookAuthor linking table are *dependant entities* because they rely on the Book's Primary key. So, if you hard delete a Book then all the Review, and BookAuthor rows link to that Book row are deleted too. And if those dependant entities had their own dependants, then they would be deleted too – the delete cascades down all the dependant entities.

So, if we duplicate that cascade delete down the dependant entities but setting the SoftDeleted property to true, then it would soft delete all the dependant too. That works, but it gets a bit more complex when you want to reset the soft delete. Read the next section for what you really need to do.

Building solution 3 – Cascade SoftDeleteService

I decided I wanted to write a service that would provide a cascade soft delete solution. Once I started to really build this, I found all sorts of interesting things to that I had to solve because we when we reset the soft delete we want the related entities to come it back to their original soft deleted state. I turns out that I bit more complex, so let's first explore this problem I found with an example.

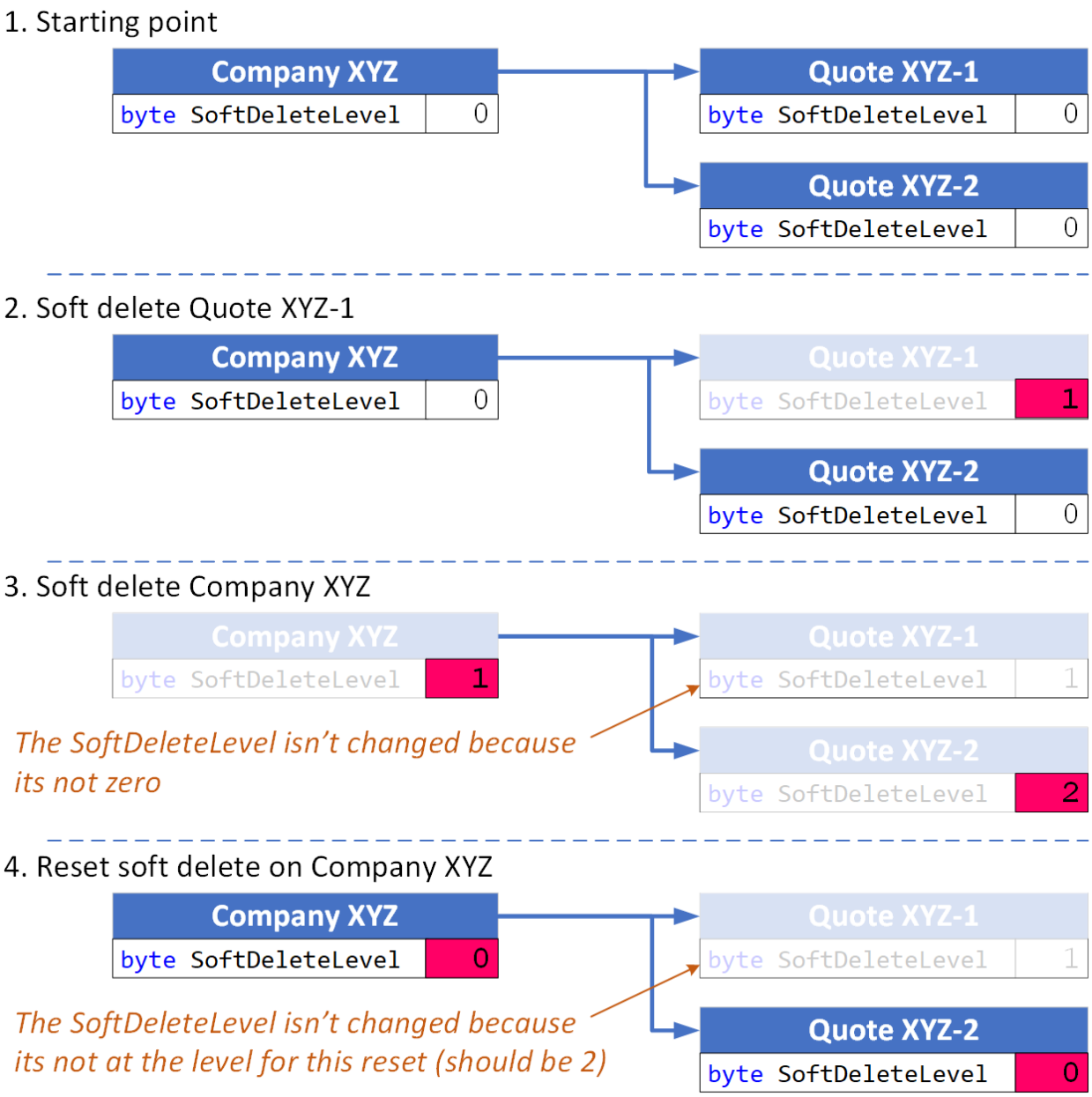
Going back to our Company/Quotes example let's see what happens if we do cascade the setting of the SoftDeleted boolean down from the Company to the Quotes (hint – it doesn't work in some scenarios). The starting point is we have a company called XYZ, which has two quotes XYZ-1 and XYZ-2. Then:

What	Company	Quotes
Starting	XYZ	XYZ-1 XYZ-2
Soft delete the quote XYZ-1	XYZ	XYZ-2
Soft delete Company XZ	– none –	– none –
Reset the soft delete on the company XYZ	XYZ	**XYZ-1 (wrong!)** XYZ-2

What has happened here is when I reset Company XYZ it also resets ALL the Quotes, and that's not what the original state was. It turns out we need a byte, not a boolean so that we can know what to reset and what to keep still soft deleted.

What we need to do is have a soft delete *level*, where the level tells you how far down was this soft delete setting set. Using this we can work out whether we should reset the soft delete or not. This gets pretty complex, so I have a figure that shows how this works. Light coloured rectangle represent entities that are soft

deleted, with the change from the last step shown in red.



So, you can handle cascade soft deletes/resets and it works really well. There are lots of little rules you cover in the code, like you can't start a reset of an entity if its SoftDeleteLevel isn't 1, because a higher-level entity soft deleted it, and I have tried to build in checks/workarounds to the issues.

I think this cascade soft delete approach is useful and I have built some prototype code to do this, but it's going to take quite a bit more work to turn it into a NuGet library that can work with any system (here is my current list of things to do).

The soft delete library out now. Here are the some useful links

- See EfCore.SoftDeleteServices on NuGet for the library
- The "Introducing the EfCore.SoftDeleteServices library to automate soft deletes" article for how to use this library.
- The documentation can be found in the repo wiki.
- The code can be found at https://github.com/JonPSmith/EfCore.SoftDeleteServices.

Conclusion

Well we have well and truly looked at soft delete and what it can (and cannot) do. As I said at the beginning, I have used soft delete on two of my client's systems and it makes so much sense to me. The main benefits are inadvertent deletes can be restored and history is preserved. The main downside is the soft delete filter might slow queries down but adding an index on the soft deleted property will help.

I know from my experiences that soft delete works really well in business applications. I also know that cascade soft deletes would have helped in one of my client's systems which had some hierarchical parts – deleting a higher level would then marked all child parts as soft deleted too which would make things faster when querying the data.

The EfCore.SoftDeleteSevices is out now. I had problems with the Git setup (my fault, not GitHub's fault) so renamed the old repo to -Old and created a new EfCore.SoftDeleteServices repo. That means I lost all the stars people had added to say they wanted the library. Thanks to to starred the old repo you for your support.

Happy coding.

Jon P Smith .NET Core, Entity Framework

5

Article Rating

Subscribe Login

guest

Join the discussion

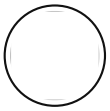
16 COMMENTS Newest

Tom 4 months ago

Thanks for such a thorough article on the subject of soft deletes. I'm currently implementing soft deletes for existing code that includes owned entities (I hope I'm getting the terminology correct). For that reason, the child entity can't have its code in OnModelCreating and it is instead part of the parent entity's configuration. Unfortunately, at that point HasQueryFilter is not available, so I'm not sure how to enable a global query filter for the child entity.

I've looked around and haven't found a good solution for this situation. I'm wondering if it's something that you've considered and what solution you might have come up with.

Thanks!



Author

Jon P Smith 4 months ago

Reply to Tom

Hi Tom,

As you said the Owned Type is added to its parent entity class – in database terms the parent class and its owned types are all in one table. That means you can't soft delete on owned type inside a parent – it just shares the parent entity SoftDeleted flag.

If you do want to soft delete a Owned Type you would need to move the data from your Owned Type into another entity class/table (or just stopped telling EF Core that the class was an Owned Type), with a one-to-one relationship from the parent to this new entity class. Then you can add the SoftDeleted flag to the new entity class.

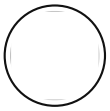
I hope that helps.

0 Reply

Felix Rabinovich 5 months ago

I may be taking soft delete concept where it wasn't intended to – but I am trying to apply query filter for *inactivated* entity. Using your relationship example 2, say, the company went out of business, doesn't pay – whatever, we are not sending quotes to them any more. Any place I show a choice of companies, it is filtered out. However, for historical quotes – I *do* want to see related data. Is it possible? Or this important but minor use case prevents me from using query filter at all?
I post a more detailed question on SO:
https://stackoverflow.com/questions/65453220/global-query-filter-breaks-navigation-entity

0 Reply



Author

Jon P Smith 5 months ago

Reply to *Felix Rabinovich*

Hi Felix,

Firstly using query filters is a perfect solution for your *inactive* user feature. I know this article is about soft delete, but you can, and should, use query filters for situations where you want to 'hide, but not delete' an class/row in your EF Core/database.

So, the issue how to access the classes/rows that are hidden, and its relationships. Here are two options:

1. Add the `IgnoreQueryFilters` method in your query. That will make all the classes/rows available.
2. If you want to only get an relationship, like historical quotes, that isn't using a query filter, then you could simply query the historical quotes directly. Taking some code from your stackoverflow question you could use the code below to see all Courses grouped by each student

```
context.Courses.GroupBy(c => c.StudentName);
//Or groupby StudentId
```

I hope that helps.

Last edited 5 months ago by Jon P Smith

+ 1 − Reply

Felix Rabinovich 5 months ago

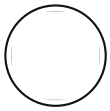
Reply to *Jon P Smith*

Thank you. The second option excludes inactive records; so I *still* need to use `IgnoreQueryFilters`, which is OK.

The need to use `IgnoreQueryFilters` in the first option makes the choice of Query Filters less obvious. Yes, I *always* want to exclude inactive Courses when I provide a list of courses to the user. But similarly, I *always* want to show course information when I show information about the student. So, the question is what is a bigger hassle – explicitly specify `&& ctx.Active` or `IgnoreQueryFilters()`

Good food for thought. Thank you. Needless to say that real application domain is much more complex; but this is a very good MCVE

0 Reply



Author

Jon P Smith 5 months ago

Reply to Felix Rabinovich

You might want to look at my repo EfCore.SoftDeleteServices which is designed to handle multiple query filters on the same entity. I did that because I had to do this on a client's app where we were using query filters for separating tenants AND using soft delete.

This library has code to allow you to create a query that turns off just one of the query filters, for instance for my client app turn off the soft delete filter while keeping the tenant filter (actually it turns off all the query filters but then recreates the other filters into the query).

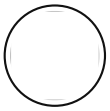
Unfortunately the library isn't finished, and there isn't an documentation. I plan to release the library soon, but finishing my book is a higher priority at the moment.

+ 0 − Reply

yavuz 6 months ago

Wow! thank you for your sharing and also thank you for EfCore.SoftDeleteServices repo.

+ 0 − Reply



Author

Jon P Smith 6 months ago

Reply to *yavuz*

Hi yavuz,

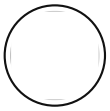
Glad you liked the article and the EfCore.SoftDeleteServices repo.I do plan to release EfCore.SoftDeleteServices as a NuGet package but I'm working really hard to finish my book Entity Framework Cor in Action, as EF Core 5 is now out.

0 Reply

Peter Szel 9 months ago

This must be the best article on soft deletes on the internet 🙂 Thank you!

0 Reply



Author

Jon P Smith 9 months ago

Reply to *Peter Szel*

Glad you like it. Soft delete is really useful but when you use if for real then lots of little things pop up. I have worked on two client systems with soft delete and I learnt a lot. Writing real world applications is SO interesting.

I have written a library called EfCore.SoftDeleteServices and learnt some things, like how to handle query filters that have multiple filter parts, for instance query filter with soft delete AND a filter by UserId. For that you need to ignore the soft delete part but put back the UserId filter part to make sure you don't undelete someone else's items!

PS. The EfCore.SoftDeleteServices isn't out yet as it uses some EF Core 5 features. Once its out I'll write another article.

0 Reply

Tominator 10 months ago

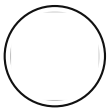
The problem with this is when you write queries with multiple joins (or includes) over soft-deleteable entities. In that case when EF Core writes the actual SQL, for every join it writes a subquery that returns not deleted entries (and those subqueries select ALL the fields of that entity!!!). But if you cascade the soft-deletion to all children of a entity, then that is a complete waste of time.

Does anybody know how to work around this? To only check on ther “FROM entity” of a query?

More info here.

Last edited 10 months ago by Tominator

0 Reply



Author

Jon P Smith 9 months ago

Reply to Tominator

I have given you an answer to your stack overflow question. I hope that helps.

0 Reply

AlexandreJobin 10 months ago

Great post Jon! What I have done on my side is to let the programmer hard delete the entity that is marked as ISoftDelete and override the SaveChanges to convert the hard delete to soft delete. That way, there's no way that the programmer can make a mistake and hard delete the entity. I think the code come from the https://abp.io/

```
public override Task SaveChangesAsync(bool
acceptAllChangesOnSuccess, CancellationToken
cancellationToken = default)
{
foreach (var entry in
ChangeTracker.Entries().ToList())
{
CancelDeletionForSoftDelete(entry);
}

return
base.SaveChangesAsync(acceptAllChangesOnSuccess,
cancellationToken);
}

protected virtual void
CancelDeletionForSoftDelete(EntityEntry entry)
{
if (!(entry.Entity is ISoftDelete softDeleteEntity))
{
return;
}
entry.Reload();
entry.State = EntityState.Modified;
softDeleteEntity.IsDeleted = true;
}
```

+ 0 − Reply

Jon P Smith 10 months ago

Reply to *AlexandreJobin*

Hi AlexandreJobin,

Thanks for your comment.

I see where you are going but I'm not sure that would work in all cases. For instance, if you use the EF Core delete behaviour of the thing you deleted is set to ClientCascadeDelete then all the dependants will also marked as Deleted too. That means that the dependant classes will also be soft deleted, which isn't what you expected to happen with a single SoftDelete version.

0 Reply

AlexandreJobin 10 months ago

Reply to *Jon P Smith*

I don't understand why you say that this is not what you expected to happen to the dependant classes when you soft delete the parent. If the Company and Quote entities are marked with ISoftDelete and they are configured with ClientCascadeDelete, then you want that dependant classes to be soft deleted right?

0 Reply

Jon P Smith 10 months ago

Reply to AlexandreJobin

In the normal, single soft delete you normally soft delete one row. If you soft delete other dependent rows then how do you get them back? Resetting the original row's soft delete flag won't get the other dependent rows back.

Also I discuss a cascade soft delete approach in this article. When I started to implement this I found some issues around returning an entity with relationships back to the same state. There are quite a few rules you need to implement to cascade soft delete.

Your approach works in many cases and it might be find in your application but there are some (small?) issues that can catch you out.

0 Reply

Proudly powered by WordPress