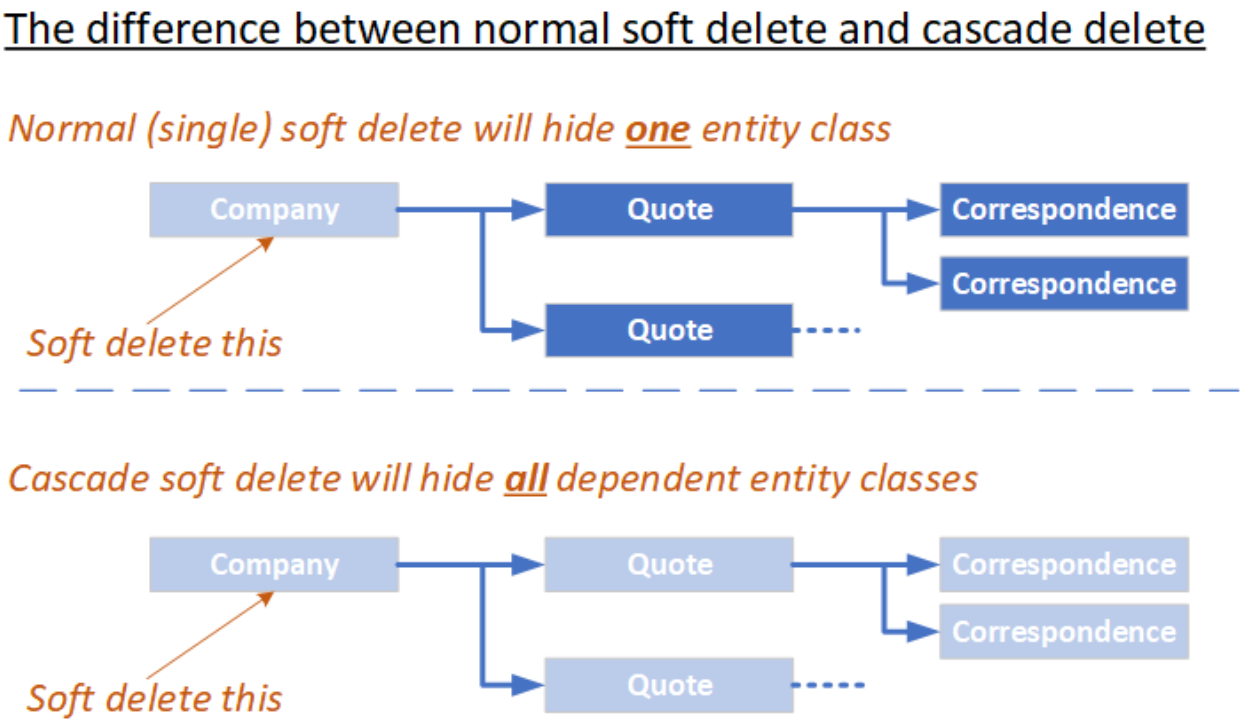


Introducing the EfCore.SoftDeleteServices library to automate soft deletes

Last Updated: January 13, 2021 | Created: January 12, 2021

Following on from my articles "EF Core In depth – Soft deleting data with Global Query Filters" I have built a library, EfCore.SoftDeleteServices (referred to as the *Soft Delete library* from now on), which provides services that automates the methods your need, that are: soft delete, find soft deleted items, un-soft delete, and hard delete a soft deleted item. In addition, this library provides solutions to various soft delete issues such as handling multiple query filter and mimicking the cascade delete feature that SQL database provide.

> *NOTE: The library is available on GitHub at https://github.com/JonPSmith/EfCore.SoftDeleteServices and on NuGet as EfCore.SoftDeleteServices.*
>
> *Also, Readers of my first article that starred the project should know I had a Git issue (my fault) and I renamed the first GitHub version to EfCore.SoftDeleteServices-Old and restarted the project. Please link to the new repo to make sure you are kept up to date.*

TL;DR – summary

- Soft delete is the term uses when you "hide" a row in your database instead of deleting a row. You can do implement this using EF Core Query Filter feature. See this link for more explanation.
- The EfCore.SoftDeleteServices solves a lot of issues when implementing a soft delete feature.
  - It provides code that can automatically configure the Query Filters for your soft delete entities.
  - It is very configurable, with you deciding where you what to put the soft delete value – in a bool property, a bit in a [Flags] enum, a shadow property, a Domain-Driven Design with methods.
  - This library can handle Query Filter contains other filters, like multi-tenant control. It makes sure the other filters are still applied, e.g. that means you are never in a situation where the multi-tenant filter isn't used. That's a very good security feature!
  - It has a cascade soft delete feature mimic what a normal (hard) delete does. This is useful in places when another part of your code accesses the dependent relationships of an entity that was soft deleted. That stops incorrect results – see this section.
  - It provides a method to register your configuration to DI and will automatically registers the right version of the Soft Delete service for you to use.
- The library has documentation and is available on NuGet at EfCore.SoftDeleteServices.

Setting the scene – what are the issues around implementing soft delete?

If you want to soft delete an *entity* instance (I use the term entity instance or *entity class* to refer to a class that has been mapped by EF Core to a database) by using EF Core's Query Filter feature, then you need to do three things:

1. Add a boolean property, say SoftDeleted, to your entity class.
2. Configure a Query Filter on that entity class using the SoftDeleted property
3. Build code to set/reset the SoftDeleted property
4. Build code to find the soft deleted entities using the IgnoreQueryFilters method

NOTE: I show these four stages in this section of my "EF Core In depth – Soft deleting data with Global Query Filters" article.

None of these steps are hard to do, but if we are really trying to mimic the way that the database deletes things, then you do need to be a bit more careful. I covered these issues, with some solutions in the previous article, but here is the list:

- If your Query Filter contains other filters, like multi-tenant control, then things get more complex (see this explanation).
- It's not a good idea to soft delete a one-to-one relationship, because EF Core will throw errors if you try to add a new version (see this explanation).
- The basic soft delete doesn't mimic what a normal (hard) delete does – a hard delete would, by default, delete any dependant rows too. This I solve by the cascade soft delete part of the Soft Delete library.

The EfCore.SoftDeleteServices library is designed to overcome all of these issues and gives you a few more options to. In the next section I will describe a simple example of using this service

An example of using the EfCore.SoftDeleteServices library

Let's start with the most used feature – soft deleting a single entity. The starting point is to create an interface and then adding that interface to the entity classes

which you want to soft delete. In this example I am going to add a boolean Soft-Deleted property to the Book entity class.

1. Using an interface to define what entities you want to soft delete

```
public interface ISingleSoftDelete
{
    bool SoftDeleted { get; set;  }
}
public class Book : ISingleSoftDelete
{
    public int Id { get; set; }

    public bool SoftDeleted { get; set; }
    //… rest of class left out
}
```

2. Setting up Query Filters to your entities

You need to add a Query Filter to every entity class. You can write the code for each entity class, which I show next, but you can automate added a Query Filter to every entity class, which I show after.

The manual setup goes in the OnModelCreating in your application's DbContext – see the code below.

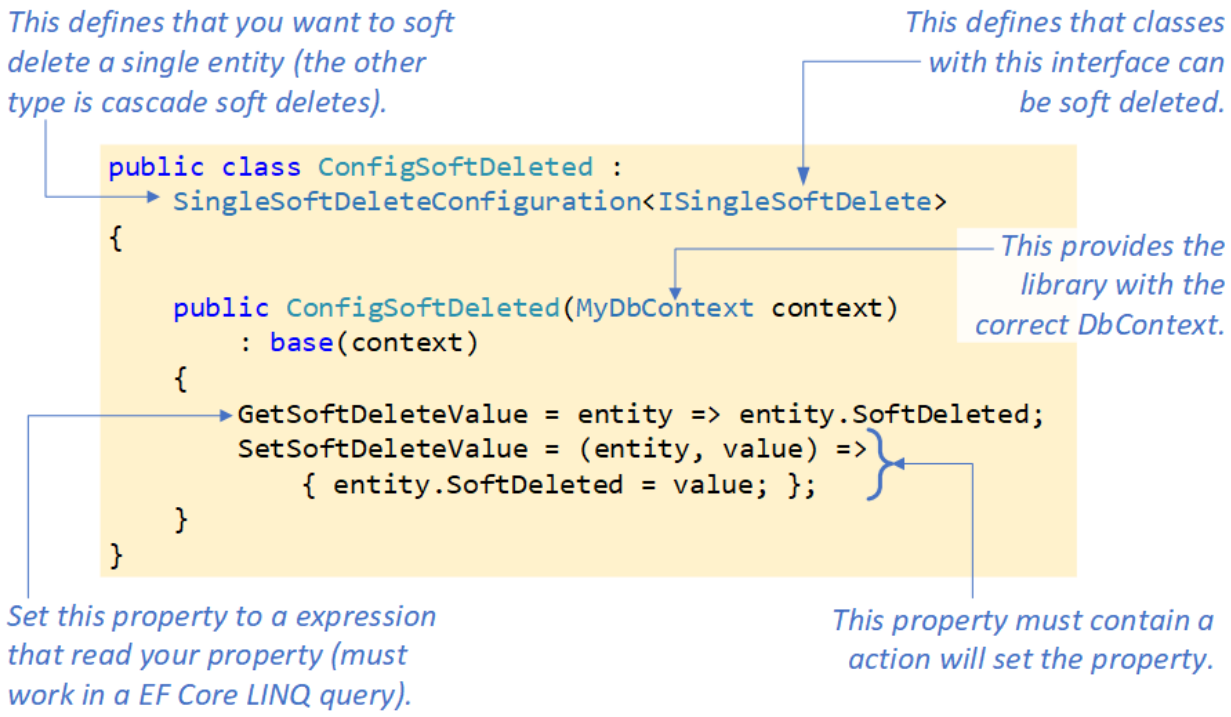
```
public class EfCoreContext : DbContext
{
    //Other code left out to focus on Soft delete

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

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

But as I said I recommend automating your query filters by running code inside of your OnModleCreating method that looks at all the entity classes and adds a Query Filter to every entity class that has your soft delete interface, in this example ISin-

gleSoftDelete. I have already described how to do this in this section of the article of soft deletion. You can also find some example code to do that in this directory of the Soft Delete's GitHub repo

3. Configuring the soft delete library to your requirements

You need to create a configuration class which will tell the Soft Delete library what to do when you call one of its methods. The class below provides the definition for your entity classes with your interface (in this case ISingleSoftDelete), and how to get/set the soft delete property, plus other things like whether you want the single or cascade soft delete service and gets access to your application's DbContext.




```
public class ConfigSoftDeleted :
    SingleSoftDeleteConfiguration<ISingleSoftDelete>
{

    public ConfigSoftDeleted(MyDbContext context)
        : base(context)
    {
        GetSoftDeleteValue = entity => entity.SoftDeleted;
        SetSoftDeleteValue = (entity, value) =>
            { entity.SoftDeleted = value; };
    }
}
```

Your configuration must inherit either the `SingleSoftDeleteConfiguration<TInterface>` or the `CascadeSoftDeleteConfiguration<TInterface>` class – which one you use will define what service/features it provides.

> *NOTE: While I show the SoftDeleted property as a boolean type you could make it part of say a flag Enum. The only rule is you can get and set the property using a true/false value.*

4. Setting up the Soft Delete services

To use the Soft Delete library, you need to get an instance of its service. You can create an instance manually (I use that in unit tests), but many applications now use dependency injection (DI), such as ASP.Net Core. The Soft Delete library provides an extension method called RegisterSoftDelServicesAndYourConfigurations, which will find and register all of your configuration classes and also registers the correct soft delete service for each configuration. The code below shows an example of calling this method inside ASP.Net Core's startup method.

```
public void ConfigureServices(IServiceCollection services)
{
    //other setup code left out
    var softLogs = services
       .RegisterSoftDelServicesAndYourConfigurations(
           Assembly.GetAssembly(typeof(ConfigSoftDeleted))
        );
}
```

This will scan the assembly which has the ConfigSoftDeleted in and register all the configuration classes it finds there, and also registers the correct versions of the single or cascade services. In this example you would have three services configured

- ConfigSoftDeleted as SingleSoftDeleteConfiguration<ISingleSoftDelete>
- SingleSoftDeleteService<ISingleSoftDelete>
- SingleSoftDeleteServiceAsync<ISingleSoftDelete>

A few features here:

- You can provide multiple assemblies to scan.
- If you don't provide any assemblies, then it scans the assembly that called it
- The method outputs a series of logs (see var softLogs in the code) when it finds/registers services. This can be useful for debugging if your soft delete methods don't work. The listing below shows the output for my use of this library in my Book App.

```
No assemblies provided so only scanning the calling assembly
Starting scanning assembly BookApp.UI for your soft delete c
Registered your configuration class ConfigSoftDelete as Sing
SoftDeleteServices registered as SingleSoftDeleteService<ISo
SoftDeleteServicesAsync registered as SingleSoftDeleteServic
```

5. Calling the soft delete library's methods

Having registered your configuration(s) you are now ready to use the soft delete methods. In this example I have taken an example from a ASP.NET Core application I build for my book. This code allows users to a) soft delete a Book, b) find all the soft deleted Books, and c) undelete a Book.

NOTE: You can access the ASP.NET Core's Admin Controller via this link. You can also run this application by cloning the https://github.com/JonPSmith/EfCoreinAction-SecondEdition GitHub repo and selecting branch Part3.

```
public async Task<IActionResult> SoftDelete(int id, [FromS
    SingleSoftDeleteServiceAsync<ISingleSoftDelete> service
{
    var status = await service.SetSoftDeleteViaKeysAsync<Bo

    return View("BookUpdated", new BookUpdatedDto(
        status.IsValid ? status.Message : status.GetAllErro
        _backToDisplayController));
}

public async Task<IActionResult> ListSoftDeleted([FromServ
    SingleSoftDeleteServiceAsync<ISingleSoftDelete> service
{
    var softDeletedBooks = await service.GetSoftDeletedEnti
        .Select(x => new SimpleBookList{
              BookId = x.BookId,
              LastUpdatedUtc = x.LastUpdatedUtc,
              Title = x.Title})
        .ToListAsync();

    return View(softDeletedBooks);
}

public async Task<IActionResult> UnSoftDelete(int id, [Fro
     SingleSoftDeleteServiceAsync<ISingleSoftDelete> servic
{
    var status = await service.ResetSoftDeleteViaKeysAsync

    return View("BookUpdated", new BookUpdatedDto(
        status.IsValid ? status.Message : status.GetAllErro
        _backToDisplayController));
}
```

The other feature I left out was the HardDeleteViaKeys method, which would hard delete (i.e., calls the EF Core Remove method) the found entity instance, but only if it had already been soft deleted.

> *NOTE: As well as the ...ViaKeys methods there are the same methods that work on an entity instance.*

Soft Delete library easily implement this example, but coding this yourself isn't hard. So, let's look at two, more complex examples that brings out some extra feature in the Soft Delete library. They are:

- Handling Query Filters with multiple filter parts
- Using cascade soft delete to 'hide' related information

Handling Query Filters with multiple filter parts

The EF Core documentation in Query Filters gives two main usages for Query Filters: soft delete and multi-tenancy filtering. One of my client's application needed BOTH of these at the same time, which is doable but was a bit complex. While the soft delete filter is very important it's also critical that at the multi-tenant part isn't forgotten when using IgnoreQueryFilters method to access the soft deleted entities.

One of the reasons for building this library was to handle applications where you want soft delete and multi-tenancy filtering. And the solution only needs you to add one line to the Soft Delete configuration – see lines 12 and 13 in the code below.

```
public class ConfigSoftDeleteWithUserId :
    SingleSoftDeleteConfiguration<ISingleSoftDelete>
{
    public ConfigSoftDeleteWithUserId(
        SingleSoftDelDbContext context)
        : base(context)
    {
        GetSoftDeleteValue = entity =>
            entity.SoftDeleted;
        SetSoftDeleteValue = (entity, value) =>
            entity.SoftDeleted = value;
        OtherFilters.Add(typeof(IUserId), entity =>
            ((IUserId)entity).UserId == context.UserId);
    }
}
```

The OtherFilters.Add method allows you to define one or more extra filter parts,

and when it filters for the GetSoftDeletedEntries method, or the Reset/HardDelete methods it makes sure these 'Other Filters' are applied (if needed).

To test this approach, I use my standard example of an application that sells books, where I want to soft delete a Book entity class (which has no multi-tenant part), and the Order entity class which has a multi-tenant part, so orders can only be seen by the user. This means the filter for finding each of the soft deleted entities are different.

Find soft deleted Book entity type, which doesn't have the IUserId interface

```
context.Books.IgnoreQueryFilters.Where(b => b.SoftDeleted)
```

Find soft deleted on a Order entity type, which has the IUserId interface

```
context.Orders.IgnoreQueryFilters.Where(o => o.SoftDeleted && o.UserId = =
context.UserId)
```

This is automatically done inside the Soft Delete library by dynamically building an expression tree. So the complex part is done inside the library and all you need to do is cut/paste the filter part and call the OtherFilters.Add method inside the configuration class.

Using cascade soft delete to also soft delete dependent relationships

One (small) problem with the single soft delete is it doesn't work the same way as a normal (hard) delete. A hard delete would delete the entity in the database, and normally the database would also delete any relationship that can't exist without that first entity (called a *dependent relationships*). For instance, if you hard deleted a Book entity that had some Reviews, then the database's constraints would *cascade delete* all its Reviews linked to that Book. It does this to keep the *referential integrity* of the database, otherwise the foreign key in the Review table would be incorrect.

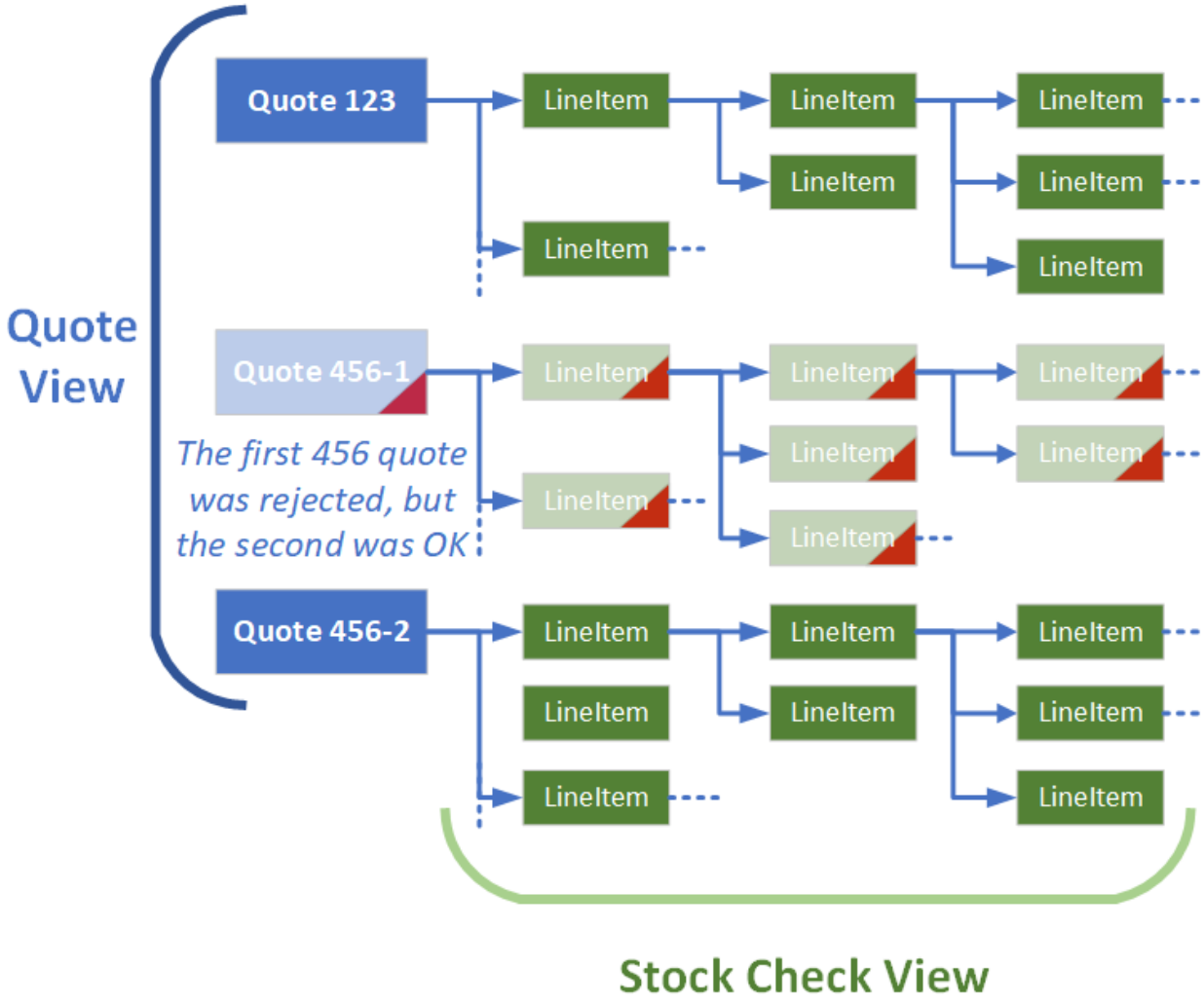
Most of the time the fact that a soft delete doesn't also soft delete the dependent re-

lationships doesn't matter. For instance, not soft deleting the Reviews when you soft delete a Book most likely doesn't matter as no one can see the Reviews because the Book isn't visible. But sometimes it does matter, which is why I looked at what I would have to do to mimic the databases cascade deletes but using soft deletes. It turns out to be much more complex than I thought, but the soft delete library contains my implementation of cascade soft deleting.

NOTE: In the previous article I go through the various options you have when soft deleting an entity with dependant relationships – see this link.

While the single soft delete is useful everywhere, the cascade soft delete approach is only useful in specific situations. One that I came across was a company that did complex bespoke buildings. The process required created detailed quotes for a job which uses a hierarchical structure (shown as the "Quote View" in the diagram). Some quotes were accepted, and some were rejected, but they needed to keep the rejected quotes as a history of the project.

At the same time, they wanted to know if their warehouse had enough stock to build the quotes have sent out (shown as the "Stock Check View" in the diagram). Quote 456-1 was rejected which means it was cascade soft deleted, which soft deletes all the LineItems for Quote 456-1 as well. This means that when the Stock Check code is run it wouldn't see the LineItems from Quote 456-1 so the Stock Check gives the correct value of the valid Quotes.



Using cascade soft delete makes the code for the Stock Check View much simpler, because the cascade soft delete of a quote also soft deletes its LineItems. The code below creates a Dictionary whose Key are the ProductSku, with the Value being how many are needed.

```
var requiredProducts = context.Set<LineItem>().ToList()
    .GroupBy(x => x.ProductSku, y => y.NumProduct)
    .ToDictionary(x => x.Key, y => y.Sum());
```

Running this code against the three quotes in the diagram means that only the dark green LineItems – the cascade soft deleted LineItems (shown in light green and a red triangle in it) aren't included, which is what is needed in this case.

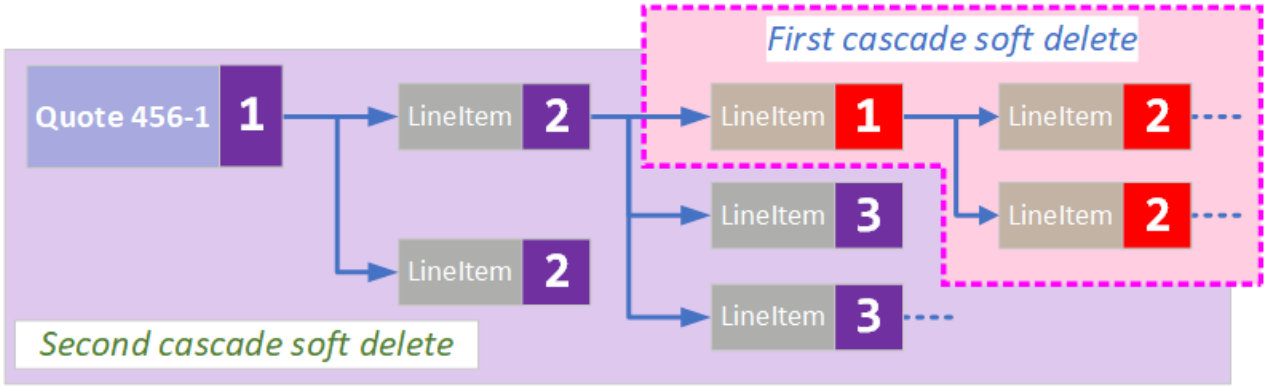
Solving the problem of cascade soft un-delete

There is one main difference between soft delete and the database delete – you can get back the soft delete data! That's what we want, but it does cause a problem

when using cascade soft delete/un-delete in that you might have already cascade soft deleted some relationships deeper down the in relationships. When you cascade soft un-delete you want the previous cascade soft delete of the deeper relationships to stay as they were.

The solution is to use a delete level number instead of a boolean. I cover this in more detail in the first article and I recommend you read this part of the article, but here is a single diagram that shows how the Soft Delete library's cascade soft un-delete can return a entity and its dependent relationships back to the point where the last cascade soft was applied – in this case the LineItems in the red area will still be soft deleted.

A cascade soft un-delete will return the entity and its relationships to state it was before the last cascade soft delete was applied.



Using the cascade soft delete methods

Using the cascade soft delete versions of the library requires you to:

1. Set up an interface which adds a property of type byte to take the delete level number.
2. Inherit the CascadeSoftDeleteConfiguration<TInterface> class in your configuration class. The code below shows you an example.

```
public class ConfigCascadeDelete :
    CascadeSoftDeleteConfiguration<ICascadeSoftDelete>
{

    public ConfigCascadeDelete(
        CascadeSoftDelDbContext context)
        : base(context)
```

```
        {
            GetSoftDeleteValue = entity =>
                entity.SoftDeleteLevel;
            SetSoftDeleteValue = (entity, value) =>
                entity.SoftDeleteLevel = value;
        }
    }
```

There are the same methods as the single soft delete methods, but they contain the word “Cascade”, for instance SetSoftDeleteViaKeys becomes SetCascadeSoftDelete-ViaKeys and so on. All the same features are there, such as handler multiple filters.

Conclusion

I have now released the EfCore.SoftDeleteServices and the library is available on GitHub at https://github.com/JonPSmith/EfCore.SoftDeleteServices and on NuGet as EfCore.SoftDeleteServices. It was quite a bit of work, but I’m pleased with the final library. I have already put it to work in my BookApp.UI example application.

My experience on working on client projects says that soft delete is a “must have” feature. Mainly because users sometime delete something they didn’t mean to do. Often the soft delete is shown to users as “delete” even though it’s a soft delete with only the admin having the ability to un-soft delete or hard delete.

Let me know how you get on with the library!

Happy coding.

Jon P Smith Entity Framework

0

Article Rating

Subscribe Login

guest

Be the First to Comment!

0 COMMENTS

Proudly powered by WordPress