ASP.NET Boilerplate

Articles Tutorials

Filter...

latest (v7.1)

Edit on GitHub

Introduction

It's common to use the **soft-delete** pattern which is used to not actually delete an entity from database but to only mark it as 'deleted'. If an entity is soft-deleted, it should not be accidentally retrieved into the application. To provide for that, we would have to add an SQL **where** condition like 'IsDeleted = false' in every query where we select entities. This is not only tedious, but is more importantly a forgettable task. To keep things DRY, there should be an automatic way to do this.

ASP.NET Boilerplate provides **data filters** that can be used to automatically filter queries based on some rules. There are some pre-defined filters, but you can also create your own.

Pre-Defined Filters

ISoftDelete

This soft-delete filter is used to automatically filter (extract from the results) deleted entities while querying the database. If an entity must be soft-deleted, it must implement the **ISoftDelete** interface which defines the **IsDeleted** property. Example:

Copy

```
public class Person : Entity, ISoftDelete
{
    public virtual string Name { get; set; }

    public virtual bool IsDeleted { get; set; }
}
```

A **Person** entity is not actually deleted from the database, instead, the **IsDeleted** property is set to true. This is done automatically by ASP.NET Boilerplate when you use the **IRepository.Delete** method (you can manually set IsDeleted to true, but the Delete method is the more natural and preferred way).

In some cases, soft-delete entities may be requested to be permanently deleted. In those cases, **IRepository.HardDelete** extension method can be used. This method is currently implemented for EntityFramework 6.x and Entity Framework Core.

When you get a list of People entities that implement ISoftDelete from the database, deleted people are not retrieved. Here is an example class that uses a person repository to get all people:

Copy

```
public class MyService
{
    private readonly IRepository<Person> _personRepository;

    public MyService(IRepository<Person> personRepository)
    {
        _personRepository = personRepository;
    }

    public List<Person> GetPeople()
    {
        return _personRepository.GetAllList();
    }
}
```

The GetPeople method only returns the Person entities where IsDeleted = false (not deleted). All repository methods and navigation properties properly work. We could add some other Where conditions, joins.. etc. It will automatically add IsDeleted = false condition properly to the generated SQL query.

When is ISoftDelete enabled?

The ISoftDelete filter is always enabled unless you explicitly disable it.

A side note: If you implement IDeletionAudited (which extends ISoftDelete) then the deletion time and user id that deleted it are also automatically set by ASP.NET Boilerplate.

IMustHaveTenant

If you are building multi-tenant applications and store all tenant data in single database, you definitely do not want a tenant accidentally seeing other tenants' data. You can implement **IMustHaveTenant** in that case. Example:

Copy

```
public class Product : Entity, IMustHaveTenant
{
    public int TenantId { get; set; }

    public string Name { get; set; }
}
```

IMustHaveTenant defines the **TenantId** property to distinguish between different tenant entities. ASP.NET Boilerplate uses the IAbpSession to get the current TenantId by default and automatically filters the query for the current tenant.

When is IMustHaveTenant enabled?

IMustHaveTenant is enabled by default.

If the current user is not logged in to the system or the current user is a **host** user (Host user is an upper-level user that can manage tenants and tenant data), ASP.NET Boilerplate automatically **disables** the IMustHaveTenant filter. Thus, all data of all tenants can be retrieved to the application. Notice that this is not about security, you should always authorize sensitive data!

IMayHaveTenant

If an entity class is shared by tenants **and** the host (that means an entity object may be owned by a tenant or the host), you can use the IMayHaveTenant filter. The **IMayHaveTenant** interface defines **TenantId** but it's **nullable**.

Copy

```
public class Role : Entity, IMayHaveTenant
{
    public int? TenantId { get; set; }

    public string RoleName { get; set; }
}
```

A **null** value means this is a **host** entity, a **non-null** value means this entity is owned by a **tenant** in which the Id is the TenantId. ASP.NET Boilerplate uses the IAbpSession to get the current TenantId by default. The IMayHaveTenant filter is not as common as IMustHaveTenant, but you may need it for **common entitiy types** used by both the host and tenants.

When is IMayHaveTenant enabled?

IMayHaveTenant is always enabled unless you explicitly disable it.

Disable Filters

You can disable a filter per unit of work by calling the **DisableFilter** method as shown below:

Copy

```
var people1 = _personRepository.GetAllList();

using (_unitOfWorkManager.Current.DisableFilter(AbpDataFilters.SoftDelete))
{
    var people2 = _personRepository.GetAllList();
}

var people3 = _personRepository.GetAllList();
```

The DisableFilter method gets one or more filter names as strings. AbpDataFilters.SoftDelete is a constant string that contains the name of the standard soft delete filter of ASP.NET Boilerplate.

people2 will also include deleted people while people1 and people3 will only return non-deleted people. With the **using** statement, you can disable a filter in a **scope**. If you don't use the using stamement, the filter will be disabled until the end of the current unit of work, or until you enable it again explicitly.

You can inject **IUnitOfWorkManager** and use it as in the example. Also, you can use the the **CurrentUnitOfWork** property as a shortcut if your class inherits some special base classes (like ApplicationService, AbpController, AbpApiController...).

About the using Statement

If a filter is enabled when you call the DisableFilter method with a using statement, the filter is disabled. It is then automatically re-enabled after the using statement. If the filter was already disabled before the using statement, DisableFilter does nothing and the filter remains disabled even after the using statement.

About Multi-Tenancy

You can disable tenancy filters to query all tenant data. Note that this only works for a single-database approach. If you have separated databases for each tenant, disabling the filter does not help to query all the data of all tenants since they are in different databases, or even on different servers. See the Multi-Tenancy document for more information.

Disable Filters Globally

If you need to, you can disable pre-defined filters globally. For example, to disable the soft-delete filter globally, add this code to the PreInitialize method of your module:

Copy

```
Configuration.UnitOfWork.OverrideFilter(AbpDataFilters.SoftDelete, false);
```

Enable Filters

You can enable a filter in a unit of work using the **EnableFilter** method, which is similar to (and opposite of) DisableFilter. EnableFilter also returns a disposable to be used in a **using** statement to automatically re-disable the filter if needed.

Setting Filter Parameters

A filter can be **parametric**. The IMustHaveTenant filter is an example of these types of filters since the current tenant's Id is determined at runtime. For such filters, we can change the filter value if needed. Example:

Copy

```
CurrentUnitOfWork.SetFilterParameter("PersonFilter", "personId", 42);
```

Here's an example on how to set the tenantId value for the IMayHaveTenant filter:

Copy

```
CurrentUnitOfWork.SetFilterParameter(AbpDataFilters.MayHaveTenant, AbpDataFilters.Paramet
```

The SetFilterParameter method also returns an IDisposable. So, we can use it in a **using** statement to automatically **restore the old value** after the using statement.

SetTenantId Method

While you can use the SetFilterParameter method to change the filter value for the MayHaveTenant and MustHaveTenant filters, there is a better way to change the tenant filter: **SetTenantId()**. SetTenantId changes the parameter value for both filters, and also works for single database and database per tenant approaches. **It is highly recommended that you use SetTenantId** to change tenancy filter parameter values. See the Multi-Tenancy document for more information.

ORM Integrations

Data filtering for pre-defined filters works for NHibernate, Entity Framework 6.x and Entity Framework Core. Currently, you can only define custom filters for Entity Framework 6.x.

Entity Framework Core

For Entity Framework Core, automatic data filtering is implemented using the **EntityFrameworkCore Global Qurty Filters**.

To create a custom filter for Entity Framework Core and integrate it into ASP.NET Boilerplate, see **Add Custom Data Filters with EntityFrameworkCore**.

Entity Framework

For Entity Framework integration, automatic data filtering is implemented using the **EntityFramework.DynamicFilters** library.

To create a custom filter for Entity Framework and integrate it into ASP.NET Boilerplate, first we need to define an interface that will be implemented by entities which use this filter. Assume that we want to automatically filter entities by PersonId. Example interface:

Copy

```
public interface IHasPerson
{
    int PersonId { get; set; }
}
```

We can then implement this interface for the needed entities. Example entity:

Copy

```
public class Phone : Entity, IHasPerson
{
    [ForeignKey("PersonId")]
    public virtual Person Person { get; set; }
    public virtual int PersonId { get; set; }

    public virtual string Number { get; set; }
}
```

We use it's rules to define the filter. In our **DbContext** class, we override **OnModelCreating** and define a filter as shown below:

Copy

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    modelBuilder.Filter("PersonFilter", (IHasPerson entity, int personId) => entity.Perso
}
```

"PersonFilter" is the unique name of the filter here. The second parameter defines the filter interface and personId filter parameter (not needed if the filter is not parametric). The last parameter is the default value of the personId.

Finally, we must register this filter to ASP.NET Boilerplate's unit of work system in the PreInitialize method of our module:

Copy

```
Configuration.UnitOfWork.RegisterFilter("PersonFilter", false);
```

The first parameter is the same unique name we defined before. The second parameter indicates whether this filter is enabled or disabled by default. After declaring such a parametric filter, we can use it by supplying it's value at runtime.

Copy

```
using (CurrentUnitOfWork.EnableFilter("PersonFilter"))
{
    using(CurrentUnitOfWork.SetFilterParameter("PersonFilter", "personId", 42))
    {
        var phones = _phoneRepository.GetAllList();
        //...
    }
}
```

We could get the personId from some source instead of it being statically coded. The example above was for parametric filters. A filter can have zero or more parameters. If it has no parameter, you don't need to set the filter parameter value. Also, if it's enabled by default, there is no need to enable it manually (you can always disable this).

Documentation for EntityFramework.DynamicFilters

For more information on dynamic data filters, see the documentation on EF's github page: https://github.com/jcachat/EntityFramework.DynamicFilters

Custom filters can be created for security, active/passive entities and so on.

Other ORMs

For NHibernate, data filtering is implemented in the repository level. This means it only filters when you query over repositories.

Note: If you directly query via custom SQL, you have to handle the filtering yourself.