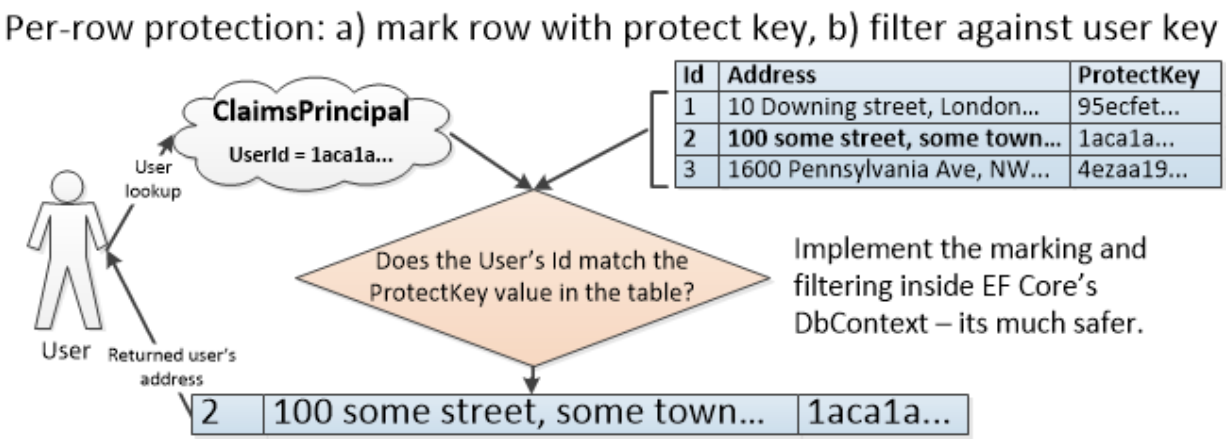


Part 2: Handling data authorization in ASP.NET Core and Entity Framework Core

Last Updated: July 31, 2020 | Created: January 14, 2019

In the first part of this series I looked at *authorization* in ASP.NET Core, but I only focused on controlling what pages/features the logged in user can access. In this article I'm going to cover how to control what *data* a user can see and change. By *data* I mean dynamic information that is stored in a database, such as your home address, what orders you made on an e-commerce site etc. Lots of web sites need to protect their, and your, data – it's a hot topic now and you really don't want to get it wrong.

- Part 1: A better way to handle authorization in ASP.NET Core.
- Part 2: Handling data authorization ASP.NET Core and Entity Framework Core **(this article)**.

This article focuses on controlling access to data in an ASP.NET Core application using Entity Framework Core (EF Core) to access the database. I have extended the example ASP.NET Core application, I built for the first article and added a new ASP.NET Core MVC web app called DataAuthWebApp which covers data authorization instead of the feature authentication I have already described in Part 1. This is an open-source (MIT licence) you can look at to see how it works underneath.

improved example application.

- Part 3: A better way to handle authorization – six months on.
- Part 4: Building robust and secure data authorization with EF Core.
- Part 5: A better way to handle authorization – refreshing user's claims
- Part 6: Adding user impersonation to an ASP.NET Core web application.
- Part 7: Implementing the "better ASP.NET Core authorization" code in your app

The Part 4 article is worth reading after this article, as it goes much deeper into the design of the data authorisation parts.

NOTE: you can Clone the GitHub repo and run locally – it uses in-memory databases so it will run anywhere and I seed it with test data. The application was written using ASP.NET Core 2.1 and EF Core 2.1: parts of the ASP.NET Identity is changing, but the overall concept will work with any version of ASP.NET Core.

This is a long article, so here are links to the major parts:

TL;DR; – summary

- You can control what data a user sees by adding [Authorize(…] attributes to your controller actions or configuring razor pages, but it's an all-or-nothing control – see the Part1 article for how to do this.
- If you want to filter the rows in a table, say only returning data associated to the current logged-in user, the you need a *per-row data protection* system – see this article on how to do that.
- The best way to implement per-row data protection is to put all the code inside EF Core's DbContext. That way the protection is on be default to every class/table that you want protected.
- The process to add per-row protection to an entity class (table) is
  - Mark every new row added to a table with a *protect key*, and…

more complex hierarchical (e.g. manager > staff) system example.

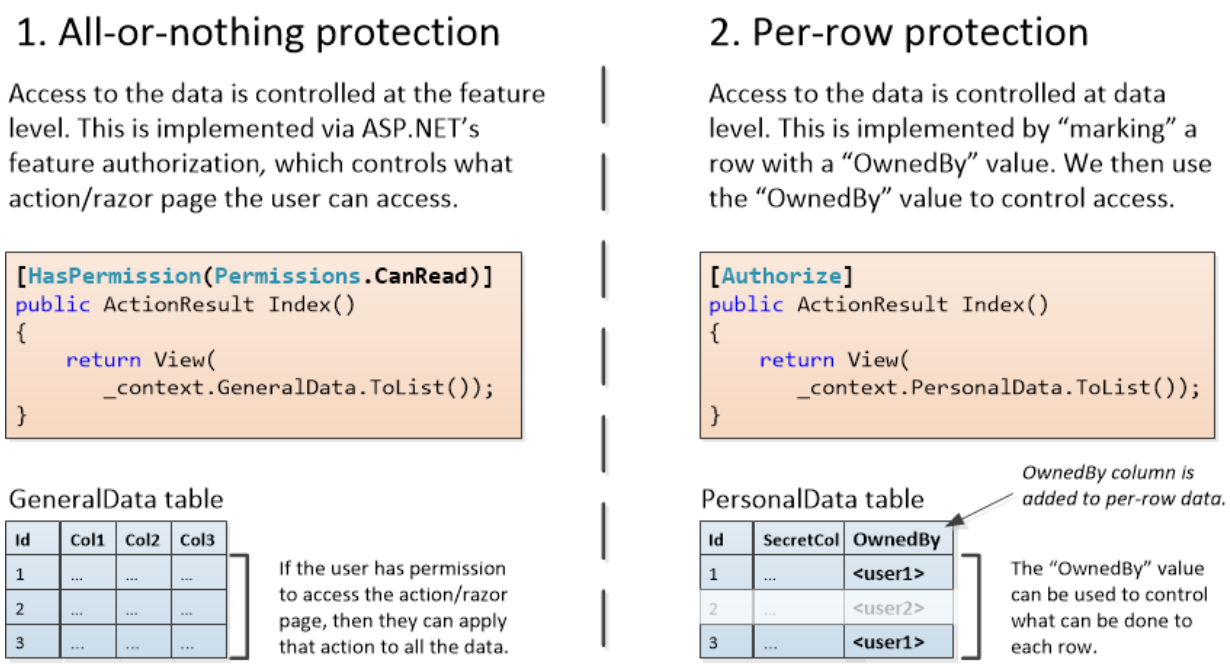
- All the code in this article is available as an open-source repo on GitHub – see PermissionAccessControl repo.

Setting the Scene – the different ways for protecting data

An ASP.NET Core web application typically has two parts:

- Static parts that is fixed by the code, e.g. how a page looks or the type of Web API it provides. Typically, you need to update your code and publish it to change the static parts.
- Dynamic data that can change, e.g. data held in a database or data obtained from external services.

In this article I'm going to look how you can protect data in a database. When we say we want to *protect* data we are talking about controlling who can see, or change, data stored in the database. If going to discuss two forms of data protection: all-or-nothing and per-row protection – see diagram below



The all-or-nothing data protection is where the user can either access all of the data or can't access it at all. In ASP.NET applications this is done by placing authorization controls on the controller actions or Razor Pages. This does allow separate control

types need per-row protection. Here are some examples:

- Protection of personal data, e.g. e-commerce systems which hold personal data for its users.
- Applications with groups of users that need to be isolated from each other. As a developer I'm sure you know GitHub, which is one (extreme) example of such a system. Generally, these types of systems are referred to as *multi-tenant* systems.
- Applications where have hierarchical data needs, e.g. where a Manager-to-staff relationship exists and say staff can access one set of data, but the Manager can see all the data their staff can access. I add this as it covers the implementation of filtering data in different ways depending on the type of user, e.g. a manager or a staff user.

The rest of this article is about applications that need per-row protection and how you might implement that with EF Core.

The two elements of per-row protection

In its core per-row protection needs two things:

1. A *protect key* in each row used to define who can access that data.
2. The user needs to provide a corresponding *key* than can be compared with *protect key* in the row.

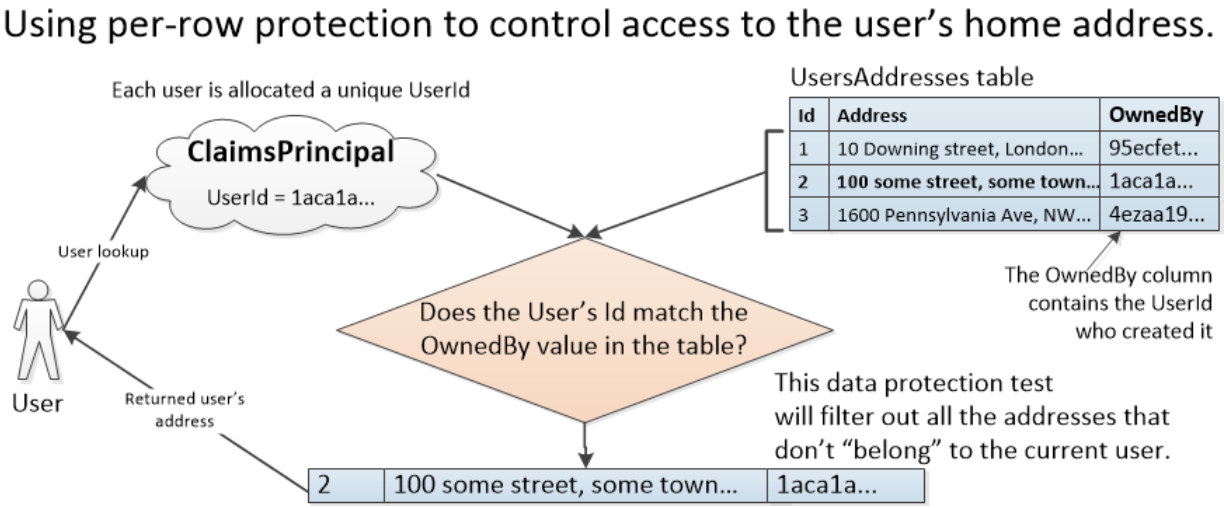
The *protect key* I refer to in the list above about could be a unique value related to an individual user, a group of users, or some form of access key. This *protect key* is sometimes referred to as the user Id or the tenant Id, and in this article I also use the name "OwnedBy".

Typically, the "OwnedBy" *protect key* is held in the information about the currently logged-in user. In ASP.NET Core this is called the ClaimsPrincipal, and it contains a series of Claims (see this article for an introduction on Claims and ClaimsPrincipal). Below is a screenshot of the basic claims of a logged-in user called Staff@g1.com.

User "Staff@g1.com"

- http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier: 1aca1a46-3956-49dd-95ea-8581598786b9
- http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name: Staff@g1.com
- AspNet.Identity.SecurityStamp: HY2O6KA4RCTAHZUQSOAPUDVMRW6VIYJL

As you can see the User has a nameidentifier claim, which a unique string for that user, often referred to as the *UserId*. In the diagram below I use the UserId as the protect key to select only the addresses in the UsersAddresses table that are link this this user. At the same time the UsersAddresses table has a corresponding protect key held in the "OwnedBy" property/column.



This diagram shows the filter part of the per-row protection system: we get a protect key from the ClaimsPrincipal and use it to filter any read of the protected data. The key will vary, and the filter expression will vary, but the approach will be the same in all cases.

The other part is how we add the protect key to an entity class/database table (what I call "marking an entity"), but that I will explain in the implementation section.

Example 1: Implementation of per-row protection on personal data

I am now going to implement the two elements of the protect key, which are:

protection code inside EF Core's DbContext. This simplifies your own CRUD/business code, and more importantly it makes sure the per-row protection doesn't get forgotten in the rush to get your code finished.

What I describe below is the same for all per-row protection, but I use the example of protecting data linked to a user. I go into detail of each step in this process, but in later examples I assume you have read this section for the general approach.

1. Marking data with the per-row protect key

The first step is to "mark" each row with the correct protect key, which I refer to at the "OwnedBy" property, i.e. that row is "owned" by the given key. I want to automatically mark any new protected table rows with the correct protect key deep inside the application's DbContext. I do this by creating an interface, IOwnedBy, (see below) which I apply to every entity class that needs row-level protection. That interface allows me to detect inside SaveChanges any newly created data that needs the OwnedBy" property filled in before saving to the database.

Let's look at parts of the code that do this, starting with the IOwnedBy interface.

1a. IOwned interface

```
public interface IOwnedBy
{
    //This holds the UserId of the person who created it
    string OwnedBy { get; }
    //This the method to set it
    void SetOwnedBy(string protectKey);
}
```

We also create a class called OwnedByBase that implements that interface and inherits that interface, which makes it easier to add the protection code to any entity classes (i.e. classes that are mapped to the database by EF Core). Here is a link to the PersonalData class in the example code that shows this in action.

1b. Override the SaveChanges and calling MarkCreatedItemAsOwnedBy

The next part is override the SaveChanges/SaveChangesAsync in the application's D Context. Here is SaveChanges version (Async version the same, but with async), so I

```
3  {
4      this.MarkCreatedItemAsOwnedBy(_userId);
5      return base.SaveChanges(acceptAllChangesOnSuccess);
6  }
```

Note the _userId on line 4. That is the UserId taken from the nameidentifier Claim – I show how you get that in the next section.

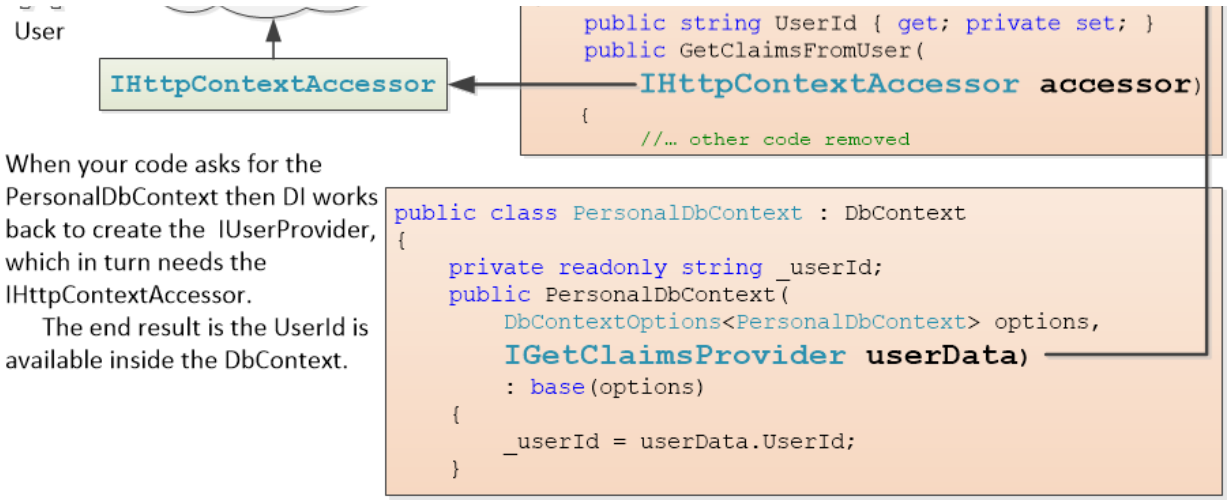
The MarkCreatedItemAsOwnedBy extension method uses EF Core's ChangeTracker to find the newly created entities and then checks if the entity has the IOwnedBy interface. The code looks like this:

```
1  public static void MarkCreatedItemAsOwnedBy(
2      this DbContext context, string userId)
3  {
4      foreach (var entityEntry in context.ChangeTracker.Entries
5          .Where(e => e.State == EntityState.Added))
6      {
7          if (entityEntry.Entity is IOwnedBy entityToMark)
8          {
9              entityToMark.SetOwnedBy(userId);
10         }
11     }
12 }
```

The effect of all that is that the OwnedBy property is filled in with the current user's UserId on any class that has the IOwnedBy interface.

1c. Getting the UserId into the application's DbContext

Ok, the first two parts are fairly basic, but getting the UserId into the DbContext is a bit more complex. In ASP.NET Core its done by a series of dependency injection (DI) steps. This can be a bit hard to understand so here is a diagram of how it works.



Here is the method extract the UserId from the current user's Claims.

```
public class GetClaimsFromUser : IGetClaimsProvider
{
    public GetClaimsFromUser(IHttpContextAccessor accessor)
    {
        UserId = accessor.HttpContext?
            .User.Claims.SingleOrDefault(x =>
                x.Type == ClaimTypes.NameIdentifier)?.Value;
    }

    public string UserId { get; private set; }
}
```

There are lots of null handling because a) there will be no HttpContext on start, and b) the NameIdentifier Claim won't be there if no one is logged in. This class is registered in DI in the ConfigureServices method inside the Startup class. That means when the application's DbContext is created the UserIdFromClaims class is injected.

NOTE: In this case I am injecting the UserId, which is found in the nameidentifier Claim. But you will see later that the same process will allow us to access any Claim in the User's data.

Here is constructor of the application's DbContext showing the normal options parameter and the added IGetClaimsProvider data.

```
public class PersonalDbContext : DbContext
{
    private readonly string _userId;

    public PersonalDbContext(
        DbContextOptions< PersonalDbContext> options,
        IGetClaimsProvider userData)
        : base(options)
    {
```

2. Filtering per-row protected data

Now that I have the per-row protected data marked with a protect key (in this case the UserId) I need to apply a filter to only allow users with the correct protect key to see that data. You could write your own code to restrict access to certain data, but with EF Core there is a much better approach – Global Query Filters (shortened to Query Filters in this article).

You can apply Query Filters to any entity class used by EF Core and they pre-filter every read of that entity class. Here is the code that sets this up for the entity class PersonalData in the application's DbContext.

```
protected override void OnModelCreating
    (ModelBuilder modelBuilder)
{
    modelBuilder.Entity<PersonalData>()
        .HasQueryFilter(x => x.OwnedBy == _userId);
}
```

In the UserId example that means a user can only retrieve the PersonalData entity classes that have their UserId, and that filtering is done inside the application's Db-Context so you can't forget.

NOTE: There is a way to remove any Query Filter from an entity class by adding the IgnoreQueryFilters method to any query. Obviously, you need to be careful where to do this as it removed your security, but it's pretty clear what you are doing.

Example 2: Implementing a multi-tenant system

The first example was dealing with personal data, which is useful but it's a simple problem which could have written into your CRUD or business logic code. But when it comes more complex per-row protection problems, like multi-tenant systems, then all that code I just showed you is a much better solution to hand-coding data protection.

As an example of a multi-tenant system I'm going to produce an application to manage the stock in a shop. To cover the cost of the development and hosting I want to

2. An admin person needs to assign a user to a specific shop.
3. I need to add the shop's protect key (called ShopKey) as a claim in the logged-in user's ClaimPrincipal.
4. I need to add the per-row protection to all the entity classes/tables that are used by a shop.

Let's look at each of these stages in turn.

1. Creating a unique protect key for each shop

When a new shop wants to join our inventory control application I need to add a new row to the Shops table. This might be done automatically via some signup system or via an admin person. In my example I use the primary key of the new Shop entity class as the protect key.

2. Assigning a user to a specific shop

Someone (or some service) needs to manage the users that can access a shop. One good approach is to make the person who signed up for the service an admin person, and they can register new users to their shop.

There isn't anything built in to ASP.NET Core's authorization system to do this, so we need to add an extra class/table to match UserId's to the shop key. Here is my MultiTenantUser class that holds the data protection information for each shop user.

```
1  public class MultiTenantUser : IShopKey
2  {
3      [Required]
4      [MaxLength(40)]
5      [Key]
6      public string UserId { get; set; }
7      public int ShopKey { get; private set; }
8
9      //other code left out for clarity
10 }
```

This is very simple: I use the UserId provided by ASP.NET Core as the primary key and then set the ShopKey to the admin person's ShopKey.

Principal as the protect key. In the multi-tenant example the big difference is I need to add a externally held protect key to the ClaimsPrincipal.

NOTE: In my implementation I use cookie-base authentication, but I could have used JWT tokens, social logins etc. The approach I share will work with any of these, but the implementation of how you add a new claim to the ClaimsPrincipal will be different for each authentication type.

In part 1, a better way to handle authorization in ASP.NET Core, I already showed how to add extra Claims to the User – in that article it was a Permissions Claim to help with feature authorization. In this article use the same approach as Part 1 to add the ShopKey to the user's Claims. That is, I tapped into an event in the Authorization Cookie called 'OnValidatePrincipal' (here is a link to the lines in the example application startup class that sets that up). This calls the code below:

```
public class DataCookieValidate
{
    private readonly DbContextOptions<MultiTenantDbContext>
         _multiTenantOptions;

    public DataCookieValidate(DbContextOptions<MultiTenantDbC
        multiTenantOptions)
    {
        _multiTenantOptions = multiTenantOptions;
    }

    public async Task ValidateAsync(CookieValidatePrincipalCo
    {
        if (context.Principal.Claims.Any(x =>
            x.Type == GetClaimsFromUser.ShopKeyClaimName))
            return;

        //No ShopKey in the claims, so we need to add it.
        //This is only happens once after the user has logged
        var claims = new List<Claim>();
        claims.AddRange(context.Principal.Claims);

        //now we lookup the user to find what shop they are l
        using (var multiContext = new MultiTenantDbContext(
              _multiTenantOptions, new DummyClaimsFromUser()))
        {
            var userId = context.Principal.Claims.Single(x =>
                 x.Type == ClaimTypes.NameIdentifier).Value;
            var mTUser = multiContext.MultiTenantUsers
                 .IgnoreQueryFilters()
                 .SingleOrDefault(x => x.UserId == userId);
            if (mTUser == null)
                throw new InvalidOperationException($"error.");
            claims.Add(new
                Claim(GetClaimsFromUser.ShopKeyClaimName,
                      mTUser.ShopKey.ToString()));
        }
```

NOTE: In Part1 I give you a detailed description of the stages in the ValidateAsync method, but for authorization. However the steps are very similar so see "How do I turn the Roles into a Permissions Claim?" for a step-by-step breakdown of how the code works.

This code is called for every HTTP request, but the first line quickly returns on all but the first request after a login. On first login the ShopKey claim isn't in the ClaimsPrincipal so it has to look up the user in the MultiTenantUsers table to add that key to the ClaimsPrincipal. The last line of the code updates the ASP.NET Core authentication cookie content so we don't have to do this again, which means it's very quick once it has run once.

4. Add the per-row protection to all the entity classes linked to a shop

This is the same process as I described in the first example of personal data, so I'm going to whiz through the code.

First, I need an interface that every protected shop entity class has to have. In this example it's called IShopKey, and here is it applied to the StockInfo entity class, with lines 7 to 11 implementing the IShopKey requirements.

```
1  public class StockInfo : IShopKey
2  {
3      public int StockInfoId { get; set; }
4      public string Name { get; set; }
5      public int NumInStock { get; set; }
6
7      public int ShopKey { get; private set; }
8      public void SetShopKey(int shopKey)
9      {
10         ShopKey = shopKey;
11     }
12
13     //-----------------------------------------
14     //relationships
15
16     [ForeignKey(nameof(ShopKey))]
17     public Shop AtShop { get; set; }
18 }
```

Then I need to extract the ShopKey claim so I can access it in the Shop application's DbContext. Here is the variant of the GetClaimsProvider that does that.

```
            var shopKeyString = accessor.HttpContext?
                .User.Claims.SingleOrDefault(x =>
                      x.Type == ShopKeyClaimName)?.Value;
            if (shopKeyString != null)
            {
                int.TryParse(shopKeyString, out var shopKey);
                ShopKey = shopKey;
            }
        }
    }
```

In this case I convert the claim value, which is a string, back into an integer as that is more efficient in the database. The query filters in the DbContext's OnModelCreating method look like this.

```
protected override void OnModelCreating(ModelBuilder modelBuil
{
    modelBuilder.Entity<MultiTenantUser>()
        .HasQueryFilter(x => x.ShopKey == ShopKey);
    modelBuilder.Entity<Shop>()
        .HasQueryFilter(x => x.ShopKey == ShopKey);
    modelBuilder.Entity<StockInfo>()
        .HasQueryFilter(x => x.ShopKey == ShopKey);
}
```

NOTE: You need to think hard about the query inside the Query Filters, as they are called on every database read of that emtity. Make sure they are as optimised as possible and add indexes to the protect key column in the database for in each protected entity class.

The DbContext parts can be found at:

- MultiTenantDbContext class with the injection of the ShopKey and the overridden SaveChanges
- MarkCreatedItemWithShopKey extension method which adds the ShopKey to any newly created Shop-related entity class (row).

Example 3 (advanced): Handling hierarchical accesses

UPDATE: I rushed this section and I wasn't very happy with it. In the new article Part 4, Building a robust and secure data authorization with EF Core, I build a proper implementation of a hierarchical system – I recommend you go and read that instead.

I have extended my multi-tenant example such that shops can have district managers. This means:

- A user linked to a shop, say Dress4U, can only access the information about that shop.
- But district managers can access the group of shops, say Dress4U, Shirt4U and Tie4U, can see information for all the shops they manage.

This requires us to add four new parts to the existing multi-tenant shop inventory application. They are:

1. The MultiTenantUser class must change to tell us if the user is a shop worker or a district manager.
2. When a user logs in I need to see if they are a district manager, and if they are I need to add a DistrictManagerId claim to the ClaimsPrincipal.
3. The Shop and StockInfo need a reference to an (optional) district manager.
4. The query filters now need to change depending on whether the current user is a shop-only user or a district manager.

I'm not going to show you the code for all the steps in detail as this article is already very long. What I want to focus on is the Query Filters.

In ASP.NET Core if you ask for an instance of the application's DbContext, then a new one is created per HTTP request (a scoped lifetime). But, for performance reasons EF Core builds the configuration on first use and caches it. This means you can't change Query Filters using an If-then-else command – instead you can use the ?: operators in the Query Filter's LINQ expression.

So, for the multi-tenant application with district manager support the OnModelCreating method looks like this.

```
protected override void OnModelCreating(
    ModelBuilder modelBuilder)
{
    //Altered query filter to handle hierarchical access
    modelBuilder.Entity<Shop>().HasQueryFilter(
        x => DistrictManagerId == null
            ? x.ShopKey == ShopKey
            : x.DistrictManagerId == DistrictManagerId);
    modelBuilder.Entity<StockInfo>().HasQueryFilter(
        x => DistrictManagerId == null
            ? x.ShopKey == ShopKey
```

and for the ShopKey. Have a look in the DataLayer for the multi-tenant entity classes and the multi-tenant DbContext.

Conclusion

Another long article, but I have given you a lot of details on how to implement a per-row data protection system. As I said at the beginning feature authorisation (covered in Part1) is much more common than per-row data protection. But if you need per-row data protection then this article shows how you can implement it.

I started with an example of using the user's unique id to control access to personal data. That also introduced all the parts needed for per-row data protection. I then went onto two more complex per-row protection usages: a multi-tenant system and a hierarchical system. The hierarchical system is complex, but one of my clients needed that (and more!), so it is used.

Security is a very important topic, and one that can be tough to implement with no security loopholes. That is why my implementation places all of the per-row data protection is inside the application's DbContext. That reduces duplication of code and makes sure per-row data protection is on by default. Personally, I also add unit tests that check I haven't forgotten to add the query filters and any protection interfaces to the entity classes – you can never be too careful on security.

Happy coding.

PS. Don't forget to sign up to Jerrie Pelser's, ASP.NET Weekly newsletter if you are interested in ASP.NET or Entity Framework. It is the most important newsletter I get.

Jon P Smith .NET Core, ASP.NET Core, Entity Framework

14 COMMENTS Newest

Mark 4 months ago

Dear Jon,

My apologies in advance if I missed the answer to the following:

As per your example, most instances of my DbContext originates from controller actions. This means that `IHttpContextAccessor` is available when DbContext instance is created.

However some parts of my application will get an instance of my DbContext while not originating from a controller action. For example, certain controller actions are allowed to create a work item in a BackgroundService queue which on processing the work item requires an instance of my DbContext.

Would there be an obvious *robust* design to support having the keys for changes in entities, when the DbContext instance is created outside the controller action?

Hope my question makes sense!

Regards,
Mark

Jon P Smith 4 months ago

Reply to Mark

want background tasks to access data that is protected via a DataKey query filter you need to:

1. Provide the background with a datakey to use
2. The background task has to create its own DbContext, passing in the datakey via the class that implements the IGetClaimsProvider interface (which just has a DataKey property).

+ 0 − Reply

Jon P Smith 5 months ago

I noticed that all the comments had been deleted when I tried to answer Gapwe's question. I have reconstructed Gapwe's question from the email I was sent – see below.

Thanks for your answer in the comments of this article's part one with the link. I implemented the same system in my project using GlobalQueryFilter with two Interfaces (ISoftDelete and IAuditedEntity). The second one marks every row with data such as CreatedBy, CreationDate, ModifiedBy & ModificationDate. (very similar to IOwnedBase in your article).

I needed to deactivate the filter for some requests and if anyone get the same requirements, for information, .IgnoreQueryFilter disable all the filters in my case. I use a UserSession object that is populated (DI) on every request with the userId and added two booleans (DisableSoftDeleteFilter and DisableAuditedEntityFilter). Adding a if-else clause when the QueryFilter is created allows me to deactivate one part or the other. (If that can avoid somebody the same struggling I experienced :p)

I still have two issues when using the GenericService library with the per-row protection.
This obviously makes a request from a user trying to access data he didn't created return an empty list instead of Unauthorized. After thinking about it, I find it even more protected this way as a malicous user can't even know if there's data at the other end of the request.

```
class FooDtoConfig : PerDtoConfig
{
public override Action> AlterReadMapping
{
get
{
return cfg => cfg
.ForMember(x => x.CreatedById, x => x.MapFrom(item =>
EF.Property(item, "CreatedById")))
.ForMember(x => x.CreatedOn, x => x.MapFrom(item =>
EF.Property(item, "CreatedOn")))

.ForMember(x => x.ModifiedById, x => x.MapFrom(item =>
EF.Property(item, "ModifiedById")))
.ForMember(x => x.ModifiedOn, x => x.MapFrom(item =>
EF.Property(item, "ModifiedOn")));
}
}
}
```

Do you have any suggestion about how to make this generic for all the DTO that will need to be populated with those fields? (DRY). I thought about marking them with an Interface but still can't see where I should make the call.

Second issue : Whenever a user creates a new row, I need to verify that the user is actually the owner of the root entity. I'm thinking the best place to do that would be when the mapping occures and not in the controller. I noticed there's a AlterSaveMapping that might be the key to what I'm looking for. Would you have an example of overriding the AlterSaveMapping?

Again, thanks so much for your articles and your time. Usually, I find what I need by looking in a lot of different sources but I could find a lot of solutions and new ways to improve my code in one place with your work.

+ 0 − Reply

Jon P Smith 5 months ago

`FooDtoConfig` class inherit from a generic abstract class which contains a method containing you four `ForMember` calls. The generic abstract class with have a signature of `ClassName` and you will then need to call the method in the `AlterReadMapping` override.

Second issue: You say:

> *Whenever a user creates a new row, I need to verify that the user is actually the owner of the root entity.*

This sentence could be interpreted in a few ways. I *think* you are talking about adding a relationship (DDD name aggregate) to a entity (DDD name root) and what to check that the root entity is also owned by the user. I also assume you want the create to fail if the root entity isn't owned by the user.

Because you need to read the root entity to check it before you do the create you are going to have to write your own EF Core code to do this (you can't use GeneriServices) The steps are a) read the root in, b) check the userId against the root owner, c) if OK then go ahead with create, otherwise return error.

PS. If you write your own EF Core code you can access GenericService's AutoMapper mappings. See this comment on one of the issues where I show you how to do that – see https://github.com/JonPSmith/EfCore.GenericServices/issues/10#issuecomment-427636892

+ 0 − Reply

Gapwe 5 months ago

Reply to Jon P Smith

I've dealt with the first issue with an abstract class. Working great !

For the second issue (Checking the root entity), for now, I'm going with the solution of only checking if the user can get the parent entity and returning NotFound() if the parent is null :

```
[FromServices]ICrudServicesAsync service)
{
//Try to access the parent entity
var parent = await service.ReadSingleAsync(x =>
x.FooId == item.FooId);
if (parent == null)
return new NotFoundResult();
else {
var result = await
service.CreateAndSaveAsync(item);
return service.Response(this, "GetSingleBar",
new { id = result?.BarId }, result);
}
}
```

For more complex cases, I'm thinking on moving the whole checking logic to another layer.
It doesn't resolve the case of checking that the user is the owner of the Root entity but I think I'm not using the DDD architecture fully here. Having a lot of levels of aggregates in many domains, I find it really difficult to deal with accessing the root entity (and keeping everything clear) when adding a entity low in the hierarchy.

```
Foo -> Bar -> Baz -> Qux
```

(May be and probably, I am missing something somewhere but I'll get to it some day with experience. For now, it's still unclear).

Having a check for the parent on every POST sounds enough to me for now. In addition, If a user managed to add an entity without being the owner of the parent, it won't be accessible neither to him nor the actual owner of the parent because of the globalqueryfilters.

+ 0 − Reply

Jon P Smith 1 year ago

Reply to Gapwe

looks OK. However I realised you could put a test inside a method in the root entity (Foo in your case) and test that the aggregate entity (Bar in your case) is linked to the root. That might be better (sorry, should have thought about that in the first place).

I have an example in my GenericServices example DDD database where I remove a Review (aggregate) from a Book (root). In this case I check the Review is linked via a foreign key to the Book that the user selected – see https://github.com/JonPSmith/EfCore.GenericServices/blob/master/DataLayer/EfClasses/Book.cs#L123-L145 (see lines 135 to 139).

In that case I threw an exception because the error suggests the system made a mistake (or you were hacked), but in your case you could have the method return bool which is true if all went well. If it was false you could return NotFound.

I hope that helps.

+ 0 — Reply

Jon P Smith 5 months ago

Sorry, but it seems that all the comments have been deleted. I don't know how or why it happened.

+ 0 — Reply

Konstantin Fukszon 5 months ago

Hi Jon,

Thanks a lot for the article. However, I still have one question. In your first example you are creating an OwnedBy field and add Query filters. However, these would protect only Get methods, i.e. it will prevent returning the items, that do not belong to the user. But what would be the best way to protect an Update method? I checked and using your approach a user still can update

Konstantin Fukszon 5 months ago

Reply to Konstantin Fukszon

I am currently checking if item exists and while checking this, the items are filtered by the current user, so if the user does not own the item, i return NotFound(). However, it would be great to know the best practices. Thanks

0 Reply

Jon P Smith 1 year ago

Reply to Konstantin Fukszon

I wrote an article called “How to write good, testable ASP.NET Core Web API code quickly” link here and I had very good feedback from people that returning NoFound (204) was the right answer – see this section of my article https://www.thereformedprogrammer.net/how-to-write-good-testable-asp-net-core-web-api-code-quickly/#nocontent-http-204-when-the-requested-data-lookup-returned-null

0 Reply

Jon P Smith 5 months ago

Reply to Konstantin Fukszon

If you are using standard EF Core you need to read the entry before you can update it. Because the Query Filters will stop you from reading a row, then you can’t update it.

PS. I have built an better, version 2 example application and written a new article on multi-tenant. It more complicated but worth reading as I skimped of a few thigs in this article, like using separate DbContext to ensure there is no leakage – see https://www.thereformedprogrammer.net/part-4-building-a-robust-and-secure-data-authorization-with-ef-core/

0 Reply

problem.

I am using Asp.net core 2.2 and I would like to implement data level authorization for my application. The requirement is many-many relationships between user and task.

Only user/users who is/are assigned to the task are authorized to view/edit the tasks based on the given role (HR, Manager ...etc,.) I would like to ask for your help related to this.

My current design for database:
+ User (extended the Identity user)
– Id
– Username
– Email
– First Name
– Last Name
...

+ Task
– Id
– Name
– Description
– Create Date
...

+ UserTask
– UserId
– TaskId

My main concern is how can I restrict the access for the users. Please advice. Thanks in advance.

0 Reply

Jon P Smith 5 months ago

Reply to REACH

Hi Reach,

This is a many-to-many problem, with many users linking to different Tasks. There are a number of ways

query filter would be something like this:

```
protected override void
OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity().HasQueryFilter(x =>
          x.UserTasks.SingleOrDefault(y =>
y.UserId ==  && y.TaskId == x.TaskId) != null );
}
```

I haven't run that code so it might not be right, but it should give you an idea to persue.

+ 0 − Reply

REACH 1 year ago

Reply to Jon P Smith

Update- I got this error

An unhandled exception occurred while processing the request.
InvalidOperationException: Reference equality is not defined for the types 'Microsoft.EntityFrameworkCore.Storage.ValueBuffer' and 'System.Object'.
System.Linq.Expressions.Expression.ReferenceNotEqual(Expression left, Expression right)

While trying to display the list of tasks on the web page.

public IActionResult Index()
{
return View(_context.Task.ToList());
}

+ 0 − Reply