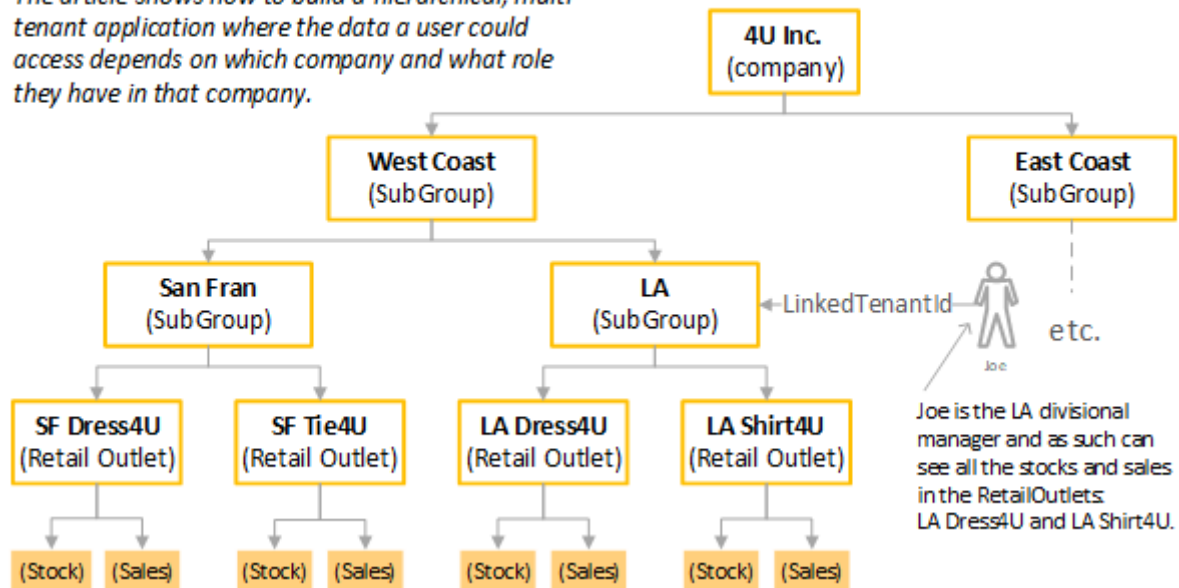


We use cookies to ensure that we give you the best experience on our website and monitoring traffic. If you continue to use this site we will assume that you are happy with it.

Ok

Building a robust and secure data authorization with EF Core

The article shows how to build a hierarchical, multi-tenant application where the data a user could access depends on which company and what role they have in that company.



Part 4: Building a robust and secure data authorization with EF Core

Last Updated: July 31, 2020 | Created: July 9, 2019

This article covers how to implement data authorization using Entity Framework Core (EF Core), that is only returning data from a database that the current user is allowed to access. The article focuses on how to implement data authorization in such a way that the filtering is very secure, i.e. the filter always works, and the architecture is robust, i.e. the design of the filtering doesn't allow developer to accidentally bypasses that filtering.

This article is part of a series and follows a [more general article on data authorization](#) I wrote six months ago. That first article introduced the idea of data authorization, but this article goes deeper and looks at one way to design a data authorization system that is secure and robust. It uses a new, improved example application, called [PermissionAccessControl2](#) (referred to as "version 2") and a series of new articles which cover other areas of improvement/change.

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

- [Part 7: Implementing the better ASP.NET Core authorization code in your app](#)

UPDATE: See my [NDC Oslo 2019](#) talk which covers these three articles.

Original articles:

- [A better way to handle authorization in ASP.NET Core](#) – original article.
- [Handling data authorization in ASP.NET Core and Entity Framework Core](#).

TL;DR; – summary

- This article provides a very detailed description of how I build a hierarchical, multi-tenant application where the data a user could access depends on which company and what role they have in that company.
- The example application is built using ASP.NET Core and EF Core. You can look at the code and run the application, which has demo data/users, by cloning [this GitHub repo](#).
- This article and its new (version 2) example application is a lot more complicated than the data authorization described in the [original \(Part 2\) data authorisation article](#). If you want to start with something, then read the original (Part 2) first.
- The key feature that makes it work are EF Core's [Query Filters](#), which provides a way to filter data in ALL EF Core database queries.
- I break the article into two parts:
 - Making it Secure, which covers how I implemented a hierarchical, multi-tenant application that filters data based on the user's given data access rules.
 - Making it robust, which is about designing an architecture that guides developers so that they can't accidentally bypass the security code that has been written.

Setting the scene – examples of data authorization

Pretty much every web application with a database will filter data – Amazon doesn't show you every product it has, but tried to show you things you might be interested in. But this type of filtering for the convenience of the user and is normally part of the application code.

Another type of database filtering is driven from security concerns – I refer to this this as *data authorization*. This isn't about filtering data for user convenience, but

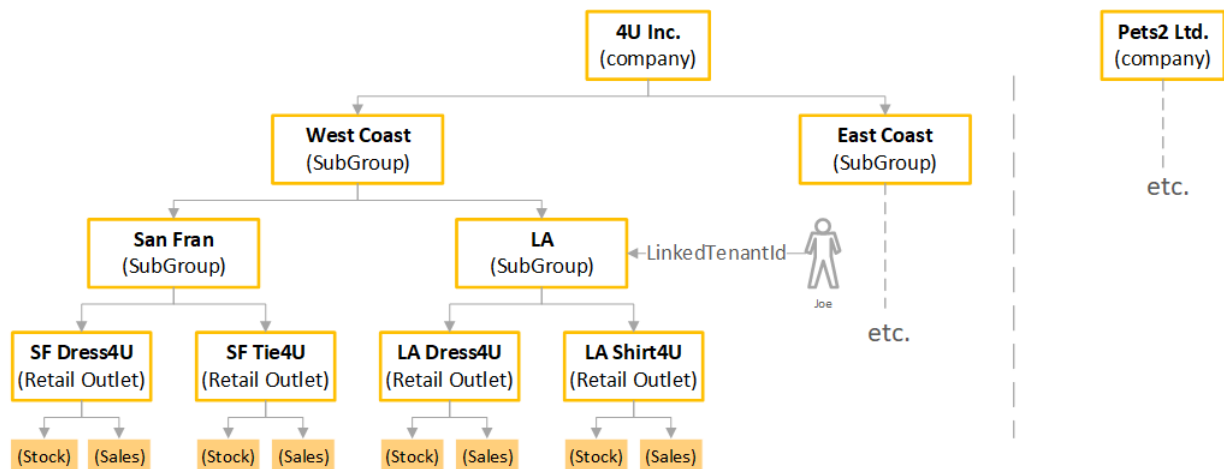
We use cookies to ensure that we give you the best experience on our website and monitoring traffic. If you continue to use this site we will assume that you are happy with it.

Ok

- multi-tenant systems, where one database is used to support multiple, separate user's data.
- Hierarchical systems, for instance a company with divisions where the CEO can see all the sales data, but each division can only see their own sales data.
- Collaboration systems like GitHub/BitBucket where you can invite people to work with you on a project etc.

NOTE: If you are new to this area then please see [this section of the original article](#) where I have a longer introduction to data protection, both what's it about and what the different part are.

My example is a both a multi-tenant and a hierarchical system, which is what one of my client's needed. The diagram below shows two companies 4U Inc. and Pets2 Ltd. with Joe, our user in charge of the LA division of 4U's outlets.



The rest of this article will deal with how to build an application which gives Joe access to the sales and stock situation in both LA outlets, but no access to any of the other divisions' data or other tenants like Pets2 Ltd.

In the next sections I look at the two aspects: a) making my data authorization design secure, i.e. the filter always works, and, b) its architecture is robust, i.e. the design of the filtering doesn't allow developer to accidentally bypasses that filtering.

A. Building a secure data authorization system

I start with the most important part of the data authorization – making sure my approach is secure, that is, it will correctly filter out the data the user isn't allowed to see. The cornerstone of this design is EF Core's [Query Filters](#), but to use them we

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

- ization must have a property that holds a security key, which I call the DataKey.
2. **Setting the DataKey property/column:** The DataKey needs to be set to the right value whenever a new filtered class is added to the database.
 3. **Add the user's DataKey to their claims:** The user claims need to contain a security key that is matched in some way to the DataKey in the database.
 4. **Filter via the DataKey in EF Core:** The EF Core DbContext has the user's DataKey injected into it and it uses that key in EF Core Query Filters to only return the data that the user is allowed to see.

A1. Adding a DataKey to each filtered class

Every class/table that needs data authorization must have a property that holds a security key, which I call the DataKey.

In my example application there are multiple different classes that need to be filtered. I have a base IDataKey interface which defines the DataKey string property. All the classes to be filter inherit this interface. The DataKey definition looks like this

As you will see the DataKey is used a lot in this application.

A2. Setting the DataKey property/column

The DataKey needs to be set to the right value whenever a new filtered class is added to the database.

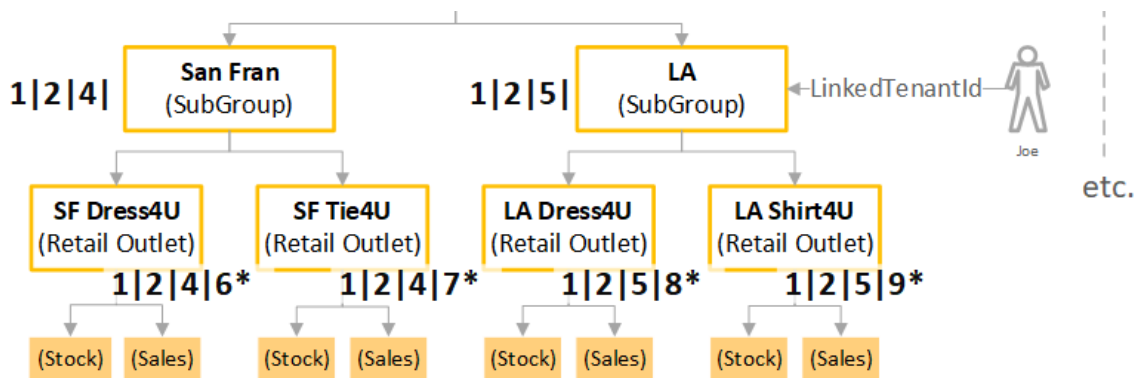
NOTE: This example is complex because of the hierarchical data design. If you want a simple example of setting a DataKey see the "personal data filtering" example in the original, Part 2 article.

Because of the hierarchical nature of my example the "right value" is bit complex. I have chosen to create a DataKey than is a combination of the primary keys of all the layers in the hierarchy. As you will see in the next subsection this allows the query filter to target different levels in the multi-tenant hierarchy.

The diagram below shows you the DataKeys (bold, containing numbers and |) for all the levels in the 4U Company hierarchy.

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



The hierarchy consists of three classes, Company, SubGroup and RetailOutlet, which all inherit from an abstract class called TenantBase. This allows me to create relationships between any of the class types that inherit from TenantBase (EF Core to treat these classes as a Table-Per-Hierarchy, TPH, and stores all the different types in one table).

But setting the DataKey on creation is difficult because the DataKey needs the primary key, which isn't set until its created in the database. My way around this is to use a transaction. Here is the method in the TennantBase class that is called by the Company, SubGroup or RetailOutlet static creation factory.

```

1 | protected static void AddTenantToDatabaseWithSaveChanges
2 |     (TenantBase newTenant, CompanyDbContext context)
3 |     {
4 |         // ... Various business rule checks let out
5 |
6 |         using (var transaction = context.Database.BeginTransaction())
7 |         {
8 |             //set up the backward link (if Parent isn't null)
9 |             newTenant.Parent?._children.Add(newTenant);
10 |             context.Add(newTenant); //also need to add it in case
11 |             // Add this to get primary key set
12 |             context.SaveChanges();
13 |
14 |             //Now we can set the DataKey
15 |             newTenant.SetDataKeyFromHierarchy();
16 |             context.SaveChanges();
17 |
18 |             transaction.Commit();
19 |         }
20 |     }

```

The Stock and Sales classes are easier to handle, as they use the user's DataKey. I override EF Core's SaveChanges/SaveChangesAsync to do this, using the code shown below.

```

1 | public override int SaveChanges(bool acceptAllChangesOnSuccess)
2 | {

```

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

10 | }

A3. Add the user's DataKey to their claims:

The user claims need to contain a security key that is matched in some way to the DataKey in the database.

My general approach as detailed in the [original data authorization article](#) is to have claim in the user's identity that is used to filter data on. For personal data this can be the user's Id (typically a string containing a GUID), or for a straight-forward multi-tenant it would be some form of tenant key stored in the user's information. For this you might have the following class in your extra authorization data:

```

1 | public class UserDataAccessKey
2 | {
3 |     public UserDataAccessKey(string userId, string accessKey)
4 |     {
5 |         UserId = userId ?? throw new ArgumentNullException(nameof(UserId));
6 |         AccessKey = accessKey;
7 |     }
8 |
9 |     [Key]
10 |    [Required(AllowEmptyStrings = false)]
11 |    [MaxLength(ExtraAuthConstants.UserIdSize)]
12 |    public string UserId { get; private set; }
13 |
14 |    [MaxLength(DataAuthConstants.AccessKeySize)]
15 |    public string AccessKey { get; private set; }
16 | }

```

In this hierarchical and multi-tenant example it gets a bit more complex, mainly because the hierarchy could change, e.g. a company might start with simple hierarchy of company to shops, but as it grows it might move a shop to sub-divisions like the west-coast. This means the DataKey can change dynamically.

For that reason, I link to the actual Tenant class that holds the DataKey that the user should use. This means that we can look up the current DataKey of the Tenant when the user logs in.

NOTE: In [Part 5](#) I talk about how to dynamically update the user's DataKey claim of any logged in user if the hierarchy they are in changes.

```

1 | public class UserDataHierarchical
2 | {
3 |     public UserDataHierarchical(string userId, TenantBase tenant)

```

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

```
11     }
12
13     public int LinkedTenantId { get; private set; }
14
15     [ForeignKey(nameof(LinkedTenantId))]
16     public TenantBase LinkedTenant { get; private set; }
17
18     [Key]
19     [Required(AllowEmptyStrings = false)]
20     [MaxLength(ExtraAuthConstants.UserIdSize)]
21     public string UserId { get; private set; }
22
23     [MaxLength(DataAuthConstants.AccessKeySize)]
24     public string AccessKey { get; private set; }
25 }
```

This dynamic DataKey example might be an extreme case but seeing how I handled this might help you when you come across something that is more complex than a simple value.

At login you can add the feature and data claims to the user's claims using the code I showed in Part 3, where I add to the user's claims via a UserClaimsPrincipalFactory, as described in [this section of the Part 3 article](#).

The diagram below shows how my factory method would lookup the UserDataHierarchical entry using the User's Id and then adds the current DataKey of the linked Tenant.

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

....

EF Core's Query Filters are a new feature added in EF Core 2.0 and they are fantastic for this job. You define a query filter in the `OnModelCreating` configuration method inside your `DbContext` and it will filter ALL queries, that comprises of LINQ queries, using `Find` method, included navigation properties and it even adds extra filter SQL to EF Core's `FromSql` method (`FromSqlRaw` or `FromSqlInterpolated` in EF Core 3+). This makes Query Filters a very secure way to filter data.

For the version 2 example here is a look the `CompanyDbContext` class with the query filters set up by the `OnModelCreating` method towards the end of the code.

```

1  public class CompanyDbContext : DbContext
2  {
3      internal readonly string DataKey;
4
5      public DbSet<TenantBase> Tenants { get; set; }
6      public DbSet<ShopStock> ShopStocks { get; set; }
7      public DbSet<ShopSale> ShopSales { get; set; }
8
9      public CompanyDbContext(DbContextOptions<CompanyDbContext>
10         DbContextOptions, IClaimsProvider claimsProvider)
11         : base(options)
12     {
13         DataKey = claimsProvider.DataKey;
14     }
15
16     //... override of SaveChanges/SaveChangesAsync left out
17
18     protected override void OnModelCreating(ModelBuilder modelBuilder)
19     {
20         //... other configurations left out
21
22         AddHierarchicalQueryFilter(modelBuilder.Entity<TenantBase>());
23         AddHierarchicalQueryFilter(modelBuilder.Entity<ShopStock>());
24         AddHierarchicalQueryFilter(modelBuilder.Entity<ShopSale>());
25     }
26
27     private static void AddHierarchicalQueryFilter<T>(EntityTypeBuilder<T> builder)
28         where T : class, IDataKey
29     {
30         builder.HasQueryFilter(x => x.DataKey.StartsWith(DataKey));
31         builder.HasIndex(x => x.DataKey);
32     }
33 }

```

As you can see the Query Filter uses the `StartsWith` method to compare the user's `DataKey` and the `DataKeys` of the tenants and their Sales/Stock data. This means if Joe has the `DataKey` of `1|2|5` then he can see the Stock/Sales data for the shops "LA Dress4U" and "LA Shirt4U" – see diagram below.

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

NOTE: You can try this by cloning the PermissionAccessControl2 repo and running it locally (by default it uses an in-memory to make it easy to run). Pick different users to see the different data/features you can access.

B. Building a robust data authorization architecture

We could stop here because we have covered all the code needed to secure the data. But I consider data authorization as a high-risk part of my system, so I want to make it “secure by design”, i.e. it shouldn’t be possible for a developer to accidentally write something that bypasses the filtering. Here are the things I have done to make my code robust and guide another developer on how to do things.

1. **Use Domain-Driven design database classes.** Some of the code, especially creating the Company, SubGroup and RetailOutlet DataKeys, are complicated. I use DDD-styled classes which provides only one way to create or update the various DataKeys and relationships.
2. **filter-Only DbContext:** I build a specific DbContext to contain all the classes/tables that need to be filtered.
3. **Unit test to check you haven’t missed anything:** I build unit tests that ensure the classes in the filter-Only DbContext have a query filter.
4. **checks:** With an evolving application some features are left for later. I often add small fail-safe checks in the original design to make sure any new features follow the original design approach.

B1. Use Domain-Driven design (DDD) database classes

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

One, instead you can call to get certain jobs done.

The effect is I can “lock down” how something is done and make sure everyone uses the correct methods. Below the TenantBase abstract class which all the tenant classes inherit from showing the `MoveToNewParent` method that moves a tenant to another parent, for instance moving a `RetailOutlet` to a different `SubGroup`.

```

1  public abstract class TenantBase : IDataKey
2  {
3      private HashSet<TenantBase> _children;
4
5      [Key]
6      public int TenantItemId { get; private set; }
7      [MaxLength(DataAuthConstants.HierarchicalKeySize)]
8      public string DataKey { get; private set; }
9      public string Name { get; private set; }
10
11     // -----
12     // Relationships
13
14     public int? ParentItemId { get; private set; }
15     [ForeignKey(nameof(ParentItemId))]
16     public TenantBase Parent { get; private set; }
17     public IEnumerable<TenantBase> Children => _children?.ToList()
18
19     //-----
20     // public methods
21
22     public void MoveToNewParent(TenantBase newParent, DbContext context)
23     {
24         void SetKeyExistingHierarchy(TenantBase existingTenant)
25         {
26             existingTenant.SetDataKeyFromHierarchy();
27             if (existingTenant.Children == null)
28                 context.Entry(existingTenant).Collection(x => x.Children)
29                     .Load();
30             if (!existingTenant._children.Any())
31                 return;
32             foreach (var tenant in existingTenant._children)
33             {
34                 SetKeyExistingHierarchy(tenant);
35             }
36         }
37
38         //... various validation checks removed
39
40         Parent._children?.Remove(this);
41         Parent = newParent;
42         //Now change the data key for all the hierarchy from this
43         SetKeyExistingHierarchy(this);
44     }
45     //... other methods/constructors left out
46 }

```

The things to note are:

Privacy - Terms

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

changed by methods inside the TenantBase class.

- Lines 22 to the end: This has the code to change the relationship and then immediately runs a recursive method to change all the DataKeys of the other tenants underneath this one.

B2. Build a filter-Only DbContext

One possible problem could occur if a non-filtered relationship was present, say a link back to some authorization code (there is such a relationship linking a tenant to a UserDataHierarchical class). If that happened a developer could write code that could expose data that the user shouldn't see – for instance accessing the UserDataHierarchical class could expose the user's Id which I wish to keep secret.

My solution was to create a separate DbContext for the multi-tenant classes, with a different (but overlapping) DbContext for the extra authorization classes (see the diagram below as to what this looks like). The effect is to make a multi-tenant DbContext which any contains the filtered multi-tenant data. For a developer makes it clear what classes you can access when using multi-tenant DbContext.

NOTE: having multiple DbContexts with a shared table can make database migrations a bit more complicated. Have a look at my article [“Handling Entity Framework Core database migrations in production”](#) for different ways to handle migrations.

B3. Using unit tests to check you haven't missed anything

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

```

1  [Fact]
2  public void CheckQueryFiltersAreAppliedToEntityClassesOk()
3  {
4      //SETUP
5      var options = SqliteInMemory.CreateOptions<CompanyDbContext>();
6      using (var context = new CompanyDbContext(options,
7          new FakeGetClaimsProvider("accessKey")))
8      {
9          var entities = context.Model.GetEntityTypes().ToList();
10
11         //ATTEMPT
12         var queryFilterErrs = entities.CheckEntitiesHasAQueryFilter();
13
14         //VERIFY
15         queryFilterErrs.Any().ShouldBeFalse(string.Join('\n',
16             queryFilterErrs));
17     }
18     public static IEnumerable<string> CheckEntitiesHasAQueryFilter(
19         this IEnumerable<EntityType> entityTypes)
20     {
21         foreach (var entityType in entityTypes)
22         {
23             if (entityType.QueryFilter == null &&
24                 entityType.BaseType == null && //not a TPH subclass
25                 entityType.ClrType
26                     .GetCustomAttribute<OwnedAttribute>() == null &&
27                 entityType.ClrType
28                     .GetCustomAttribute<NoQueryFilterNeeded>() == null)
29             {
30                 yield return $"The entity class {entityType.Name} does not have a query filter";
31             }
32         }
33     }

```

B4. Adding fail-safe checks

You need to be careful of breaking the Yagni (You Aren't Gonna Need It) rule, but a few fail-safe checks on security stuff makes me sleep better at night. Here are the two small things I did in this example which will cause an exception if the DataKey isn't set properly.

Firstly, I added a [Required] attribute to the DataKey property (see below) which tells the database that the DataKey cannot be null. This means if my code fails to set a DataKey then the database will return a constraint error.

```

1  [Required] //This means SQL will throw an error if we don't find a DataKey
2  [MaxLength(DataAuthConstants.HierarchicalKeySize)]
3  public string DataKey { get; private set; }

```

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

divisional managers can create items in a retail outlet. The divisional managers don't have the correct DataKey, but a new developer might miss that and you could "lose" data.

My answer is to add a safely-check to the retail outlet's DataKey. A retail outlet has a slightly different DataKey format – it ends with a * instead of a |. That means I can check a retail outlet format DataKey is used in the SetShopLevelDataKey and throw an exception if it's not in the right format. Here is my code that catches this possible problem.

```

1 public void SetShopLevelDataKey(string key)
2 {
3     if (key != null && !key.EndsWith("*"))
4         //The shop key must end in "*" (or be null and set else
5         throw new ApplicationException(
6             "You tried to set a shop-level DataKey but your k
7
8     DataKey = key;
9 }
```

This is a very small thing, but because I know that change is likely to come and I might not be around it could save someone a lot of head scratching working out why data doesn't end up in the right place.

Conclusion

Well done for getting to the end of this long article. I could have made the article much shorter if I only dealt with the parts on how to implement data authorization, but I wanted to talk about how handling security issues should affect the way you build your application (what I refer to as have a "robust architecture").

I have to say that the star feature in my data authorization approach is EF Core's Query Filters. These Query Filters cover ALL possible EF Core based queries with no exceptions. The Query Filters are the cornerstone of data authorization approach which I then add a few more features to manage user's DataKeys and do clever things to handle the hierarchical features my client needed.

While you most likely don't need all the features I included in this example it does give you a look at how far you can push EF Core to get what you want. If you need a nice, simple data authorization example, then please look at the Part 2 article "Han-

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

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


0

Article Rating












 Subscribe ▼

Login




Join the discussion


B I U        



19 COMMENTS

  Newest ▼



Mark  3 months ago

Hi Jon,

Based on your articles we are implementing (a derived) design. One of the restrictions we have is that we use Microsoft Identity Platform (Azure AD) for Authentication and the providing of the Identity. This means we don't issue our own tokens, and I am not able to include the DataKey as a claim (for roles we rely on Application Roles that are available on Azure AD).

As an alternative I am considering to implement `Microsoft.Extensions.Caching.Memory.IMemoryCache` in my `DbContext` with a dictionary between `UserId` and `DataKey`. I would prevent it from getting stale by doing a check in my

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

comments on my intended implementation?

Regards,
Mark

+ 0 – ➔ Reply



Mark ⌚ 3 months ago

🗨 Reply to Mark

(I did try to research this option, but most questions and articles are about caching the DbContext itself, which is not what I want. I also read about 2nd level caching, but that seems overkill for this one dictionary and I don't want to rely on a third party library for this seemingly simple cache and core design of my data security.)

+ 0 – ➔ Reply



Author

Jon P Smith ⌚ 3 months ago

🗨 Reply to Mark

Hi Mark,

I have good news for you in that I worked with a client that uses Azure AD. They had found a way to intercept the Azure AD OpenID Connect and we could add our own claims – see the link below.
<https://joonasw.net/view/adding-custom-claims-aspnet-core-2>

✎ Last edited 3 months ago by Jon P Smith

+ 1 – ➔ Reply



Mark ⌚ 3 months ago

🗨 Reply to Jon P Smith

Thanks Jon, I had seen that article (perhaps even in one of your posts (:). However the issue is that I am using Javascript SPA as the client, and the tokens are issued to this application. I did discuss (in quite a bit of details) with Microsoft Azure support how I could use the OpenID Connect

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

WHERE THE TOKENS WERE ISSUED TO THE ASP.NET
application?

 Last edited 3 months ago by Mark

+ 0 –  Reply



Author

Jon P Smith  3 months ago

|  Reply to Mark

Ah, that's a pity.

I would assume ASP.NET Core's authorize system will kick in so you could catch the login via the `UserClaimsPrincipalFactory` and add the `DataKey` there (see this section in part 3 of this article).

+ 0 –  Reply



Sander  4 months ago

Hi Jon, thanks for sharing this great series of articles.

What if Joe suddenly needs access to 1|2|4|7 as well? So in fact, in two separated parts of the hierarchy? Then we need to store a list of `DataKeys` in the user claim? How would you check in the `QueryFilter` that Joe has access?


thanks!

+ 0 –  Reply



Author

Jon P Smith  4 months ago

|  Reply to Sander

Hi Sanders,

Its hard to get a hierarchical key combined with a multiple keys that is efficient. If the data keys aren't hierarchical then you can make a comma delimited string containing all the data keys, e.g. "1,23,456,543," and use `DataKey.Contains("$",{context.DataKey},")` in the Query Filter.

That's the best I can offer. If you work out how to do it then do write about it and send me a link.

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

This is what i was looking for for last 1h reading your articles and trying to understand that complex scenarios. Why you don't simply use existing column in entity/table instead of adding custom datakey – this would remove one step from tutorial. I still try to figure out how to assign “key” for filtering for the lowest level of the hierarchy i.e. retail outlets – why we can't simply define retail outlet names in scope (claims) of user and filter table on column with outlet names? How to do this?

 Last edited 3 months ago by Kml

+ 0 –  Reply



Author

Jon P Smith  3 months ago

 Reply to Kml

Hi Kml,

I'm not sure I fully understand what you are asking, but here is my answers.

Why you don't simply use existing column in entity/table instead of adding custom datakey

This is about performance. Its easier to look up a user and get the datakey when they log in. By putting the datakey in the user's Claims it is saved by the ASP.NET Core authorise system in either a cookie or token. That means you don't have to an extra lookup for every database access, and it also makes it easier to the EF Core Query Filter to control what the user can see.

I still try to figure out how to assign “key” for filtering for the lowest level of the hierarchy

This is a bit complicated because this example is about building a hierarchical system. That

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

manager with a data key 1|2| can access a shop with a key of 1|2|3 and shops with a key of 1|2|99 (and 1|2|99|4 etc. down the hierarchy).

If you don't need a hierarchical system, then it's a lot simpler. The part 2 article in this series covers a simple system with a key per person.

I hope that helps.

 Last edited 3 months ago by Jon P Smith

+ 0 -  Reply



Jon P Smith  5 months ago

Hi Jeremy @disqus_ii92PNQPep

For some reason your comment was lost by Disqus, or you deleted it.

Thanks for sending me a message that some on my articles were “broken”. You were right, WordPress did an update that messed up my code. It took me the whole morning to get that fixed! Quite annoying.


I'm glad the articles have been useful to you. I agree that it would be nice to split the Roles/Permissions and DataKey but this series has already taken a great deal of time to code and write. I don't regret that, but I have to minimise the effort. I also try to keep the various part in separate projects or folders so you can delete the bulk of the code and then see what breaks in the common code.

All the best with you project

+ 0 -  Reply



Jeremy Navarro  5 months ago

 Reply to Jon P Smith

Not sure what happened to my comment, but thank you for replying, and fixing the article! Also, thanks for the idea about pulling the projects out of your complete solution. I might give that a go to help me

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



Jon P Smith 5 months ago

[Reply to Jeremy Navarro](#)

Hi Jeremy,

Glad that comment worked! If you want to ask any more questions then please do (but I might be a bit slow at replying with the weekend coming up).

+ 0 - [Reply](#)



Pelle 5 months ago

Why not just send the userid to sql and only allow executions of stored procedures, then have the mapping in database. You won't be able to hack it even...

+ 0 - [Reply](#)



Jon P Smith 5 months ago

[Reply to Pelle](#)

That's certainly an option and you can do that. What I was describing was how you could implement a multi-tenant system using EF Core. Overall I think EF Core solution is a good one. Sure, any software system can be hacked but I don't think EF Core is less secure than a SQL solution.

+ 0 - [Reply](#)



Pelle 8 months ago

[Reply to Jon P Smith](#)

If you allow sql-login other rights than execute then you have a higher risk.

Sure it is secure enough but it is less secure than a SQL-solution, especially if you run the application as a serviceaccount and have integrated trust in connectionstring then you have maximum security. 😊

But still, then it wouldn't be EF Core multi-tenant. I really liked your system, even if I had a question

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



Great series of articles.

Quick question though, will you explain in a little more detail what you mean when you say “lose” data in the fail-safe check section?

+ 0 — ➔ Reply



Jon P Smith ⌚ 5 months ago

↻ Reply to Brian Wied

I have added an answer next to your comment, but Disqus is not sending notifications (pain!). I hope sending this from the Disqus web site will get to you!

+ 0 — ➔ Reply



Jon P Smith ⌚ 5 months ago

↻ Reply to Brian Wied

The way I designed this example the divisional manager didn't have the right key to create new items in a RetailOutlet. If I didn't do anything the divisional manager could add a item and it would look like it worked, but it wouldn't have the correct key – so the item would be there, but “lost”.

I don't like thinks that fails silently, so I came up with a format for the DataKey such that if the user's DataKey didn't end with a “*” then it would throw an exception.

Does that make sense?

+ 0 — ➔ Reply



Brian Wied ⌚ 5 months ago

↻ Reply to Jon P Smith

Perfect sense. Thank you!

+ 0 — ➔ Reply

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