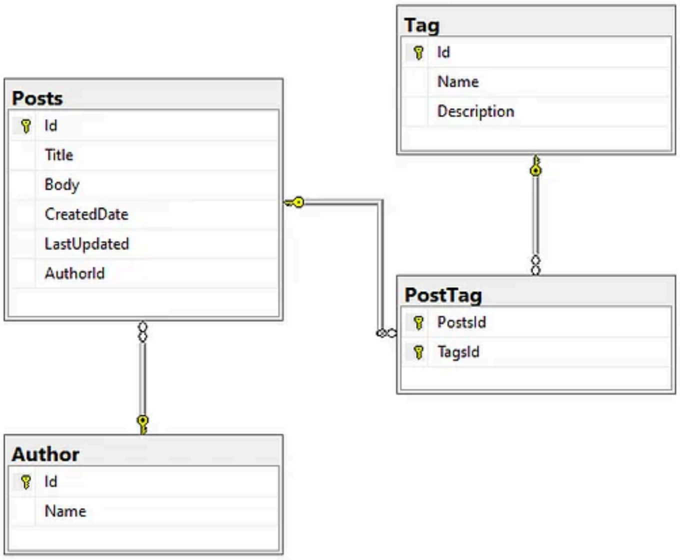
medium.com /c-sharp-programming/relationships-in-entity-framework-core-6bea347a51e4

Relationships in Entity Framework Core - .Net Programming - Medium

Osempu：17-21 minutes：2023/5/30



Hello guys, welcome to yet another exciting chapter of this series on developing a web API with ASP NET Core. Today we will move on to an advanced topic “Relationships on entity framework core” We will create more models that are a vital part of a blog, for example, authors and tags and you will learn how they relate to each other and how to configure those relationships using different approaches of entity framework core, also I will leave down a link to the last chapter of this series, it was about “Understanding the filter pipeline” which is something you should understand to leverage the power of action filters in your web apps, and having said this without a further ado let’s get into it.

Understanding The Filter Pipeline In ASP.NET Core

What are relationships in Entity Framework Core ❓

In Entity Framework Core, relationships refer to how different database tables or entities are connected or related to each other. Just like in real life, where people, objects, or concepts can have relationships with each other, databases also have relationships between tables.

Think of a relationship as a way to establish a connection between two entities in a database. These entities could represent real-world objects or concepts, such as customers and orders in an e-commerce application. By defining relationships, we can specify how these entities are associated and how they interact with each other.

Entity Framework Core allows us to define and manage different types of relationships between entities, which helps in organizing and retrieving data effectively. These relationships can be **one-to-one**, **one-to-many**, or **many-to-many**, depending on the nature of the data and the requirements of our application.

One-to-One Relationship □

- In a one-to-one relationship, it’s like having a special connection between two things. It means that each record in one table is associated with exactly one record in another table, and vice versa.
- Think of it like a person and their passport. Each person has only one passport, and that passport is unique to that person. So, we can say that the person and the passport have a one-to-one relationship.
- In database terms, this relationship is useful when we want to split information about an entity into separate tables to maintain cleanliness and avoid duplicating data.

```
public class Person
{
    public int PersonId { get; set; }
    public string Name { get; set; }
    public Passport Passport { get; set; }
}

public class Passport
{
    public int PassportId { get; set; }
    public string City { get; set; }
    public Person Person { get; set; }
}// Configuration
public class MyContext : DbContext
{
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Person>()
            .HasOne(p => p.Passport)
            .WithOne(a => a.Person)
            .HasForeignKey<Passport>(a => a.PassportId);
    }
}
```

One-to-Many Relationship 👨‍👩‍👧‍👦

- A one-to-many relationship is like a parent-child relationship, where one record in a table is associated with multiple records in another table.
- For example, think of a bookstore. Each book can have many reviews, but each review is tied to only one book. So, we can say that the relationship between books and reviews is one-to-many.
- In database terms, this relationship is useful when we need to represent situations where one entity can have multiple related entities.

```
public class Book
{
    public int BookId { get; set; }
    public string Name { get; set; }
    public ICollection<Review> Reviews { get; set; }
}

public class Review
{
    public int ReviewId { get; set; }
    public string Reviewer { get; set; }
    public Book Book { get; set; }
}// Configuration
public class MyContext : DbContext
{
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Review>()
            .HasOne(r => r.Book)
            .WithMany(b => b.Reviews)
            .HasForeignKey(r => r.BookId);
    }
}
```

Many-to-Many Relationship 👥

- A many-to-many relationship is like a group of friends. Each person can have many friends, and each friend can have many people they are friends with.
- For example, think of students and courses. A student can enroll in multiple courses, and each course can have multiple students. So, we can say that students and courses have a many-to-many relationship.
- In database terms, this relationship is implemented using an intermediary table, often called a junction or join table, to connect the entities.

```
public class Student
{
    public int StudentId { get; set; }
    public string Name { get; set; }
    public ICollection<Course> Courses { get; set; }
}

public class Course
{
    public int CourseId { get; set; }
    public string Name { get; set; }
    public ICollection<Student> Students { get; set; }
}// Configuration
public class MyContext : DbContext
{
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Student>()
            .HasMany(s => s.Courses)
            .WithMany(c => c.Students)
            .UsingEntity(j => j.ToTable("StudentCourse"));
    }
}
```

Configurations in Entity Framework Core ⚙️

Relationship configurations in Entity Framework Core are a way to define and control how different entities are related to each other in a database. These configurations help establish the rules and behavior of the relationships between entities, such as how they are mapped to database tables, how foreign key constraints are created, and how navigation properties are set up.

Above I listed all the relationships and showed you how to set them up using the fluent API configuration because it's my favorite way to do it as it keeps my models clean and free of data annotations but it doesn't have to be the same for you, maybe you do like using data annotations or setup the relationships by convention which by the way is a pretty powerful way to do it too so I will show you how to use these three type of configurations.

To configure these relationships, Entity Framework Core provides different ways to specify the rules. There are three common approaches to relationship configuration.

By Convention 🏪

Entity Framework Core has a set of conventions that can automatically configure relationships based on naming conventions and default behaviors. For example, if you have a navigation property named “Address” in the “Person” entity, Entity Framework Core will assume it’s a one-to-one relationship and configure it accordingly. This approach is convenient when your code follows the naming conventions expected by Entity Framework Core.

```
public class Person
{
    public int PersonId { get; set; }
    public string Name { get; set; }
    public Address Address { get; set; }
}

public class Address
{
    public int AddressId { get; set; }
    public string City { get; set; }
    public Person Person { get; set; }
}
```

Data Annotations 📆

Data annotations are attributes that you can apply to your entity classes and properties to specify relationship configurations. For example, you can use the **`[ForeignKey]`** attribute to specify the foreign key property or the **`[InverseProperty]`** attribute to specify the navigation property for a relationship. Data annotations are helpful when you want to explicitly define the relationships within your entity classes using attributes.

```
public class Person
{
    public int PersonId { get; set; }
    public string Name { get; set; }

    [ForeignKey("Address")]
    public int AddressId { get; set; }
    public Address Address { get; set; }
}public class Address
{
    [Key, ForeignKey("Person")]
    public int AddressId { get; set; }
    public string City { get; set; }
    public Person Person { get; set; }
}
```

Entity Framework Core provides several data annotations that you can use to configure entities and their relationships. These data annotations can be applied to entity classes and properties to specify various aspects of the configuration. Here are some commonly used data annotations for entity configuration:

1. **`[Key]`**: Specifies the primary key property of an entity. By default, Entity Framework Core assumes that a property named "Id" or "<EntityName>Id" is the primary key, but you can use this annotation to explicitly designate a property as the primary key.
2. **`[ForeignKey]`**: Specifies the foreign key property in a relationship. This annotation is used to define the foreign key column that relates to another entity.
3. **`[Required]`**: Indicates that a property is required and cannot be null. This annotation ensures that the corresponding column in the database is configured as non-nullable.
4. **`[MaxLength]`**: Sets the maximum length or size for a string or byte array property. It can be used to limit the length of a string column in the database.
5. **`[Column]`**: Specifies the name of the column in the database associated with a property. This annotation allows you to define custom column names.
6. **`[Table]`**: Specifies the table name for an entity in the database. It can be used to override the default table name generated by Entity Framework Core.
7. **`[NotMapped]`**: Excludes a property from being mapped to a column in the database. This annotation is useful when you have properties in your entity that don't have corresponding columns in the database.
8. **`[InverseProperty]`**: Specifies the inverse navigation property for a relationship. It is used when you have multiple navigation properties between two entities and need to specify the inverse relationship explicitly.
9. **`[Index]`**: Indicates that an index should be created on one or more columns in the database. This annotation can improve query performance by optimizing data retrieval.

Fluent API 🍃

Fluent API provides a more flexible and explicit way to configure relationships. With the Fluent API, you can use a set of builder methods to define the relationships between entities. This approach gives you fine-grained control over how the relationships are configured and allows you to specify additional options like cascading delete behavior, index creation, or table names.

```
public class Person
{
    public int PersonId { get; set; }
    public string Name { get; set; }
    public Address Address { get; set; }
}

public class Address
{
    public int AddressId { get; set; }
    public string City { get; set; }
    public Person Person { get; set; }
}// Configuration
public class MyContext : DbContext
{
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Person>()
            .HasOne(p => p.Address)
            .WithOne(a => a.Person)
            .HasForeignKey<Address>(a => a.AddressId);
    }
}
```

Whether you choose to configure relationships by convention, data annotations, or fluent API depends on your preference and specific requirements. Using any of these approaches, you can define the relationships between entities in a clear and understandable manner, ensuring that your database schema and entity model are aligned and properly represented.

Implementing Entities Relationships in Your Project 👷

Now let’s move on to apply the needed relationships in your web API or any kind of web project. We will use the same project we have been using for so long the “Blog API” which you can clone from GitHub using the link down below, make sure you get the code from the “ActionFilters” branch which contains the code up to this chapter.

GitHub — Osempu/BlogAPI at ActionFilters

Adding new Models 🆕

<aside> ⚠️ **Important Note:** I will not show you all the code changes the project suffered at this stage as two new models were added there was a need to create relationships, update old dtos, and add repositories and mapping profiles to say the least. But do not worry you will be able to get the code from GitHub at the end of this article to compare with your changes or to directly go and clone the branch.

</aside>

Let’s begin by adding two new needed models to create the relationships between them and enrich our project domain layer.

Author Model

Every Post will have an author and every author can have many posts as he wants so the model would end up looking like this, you will add an `ICollection<Post>` which will be the collection navigation property

```
public class Tag
{
                public int Id { get; set; }
    public string Name { get; set; }

    public ICollection<Post> Posts { get; set; }
}
```

Tag model

Every `Post` can have multiple tags and every `Tag` can be applied to any number of posts so this will be a many-to-many relationship. The model will contain some basic properties and again a collection navigation property `ICollection<Post>`.

```
public class Tag
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }

    public ICollection<Post> Posts { get; set; }
}
```

Updating the `Post` model

The post model suffered some changes as now we have two other models that interact and have a direct relationship with it so we added a navigation property for the `Author` of the post and a collection navigation property for the `Tags`.

```
public class Post
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Body { get; set; }
    public DateTime CreatedDate { get; set; }
    public DateTime LastUpdated { get; set; }

    public int AuthorId { get; set; }
    public Author Author { get; set; }    public ICollection<Tag> Tags { get; set; }
}
```

Important Note: Now that the `Author` of a post is an entity and not a single string this means that you will have to change the PostDto's that were created before now instead of specifying the name of the author to add it you will need to specify the id of that author to create a relationship between the post and the author.

`Post` Dtos updated

```
public record AddPostDTO(string Title, string Body, int AuthorId);

public record EditPostDTO(int Id, string Title, string Body, int AuthorId);
```

Configure the Entities Relationships 🛠

Now go to your `PostConfiguration` class and update it adding the relationship with the `Tag` and `Author` models, down below you can see how it will end looking using the EF Core Fluent API.

```
public class PostConfiguration : IEntityTypeConfiguration<Post>
  {
      public void Configure(EntityTypeBuilder<Post> builder)
      {
          builder.HasOne( p => p.Author)
              .WithMany( a => a.Posts)
              .HasForeignKey( p => p.AuthorId);

          builder.HasMany( p => p.Tags)
              .WithMany( t => t.Posts)
              .UsingEntity( j => j.ToTable("PostTag"));                                //Data seeding ** this
is not used for relationship configuration **
          builder.HasData(
              new Post {Id = 1, Author = new Author {Id = 1, Name = "Oscar Montenegro"}, Title = "My first
Post", Body = "Hello world, this is my first post"},
              new Post {Id = 2, Author = new Author {Id = 2, Name = "Another Author"}, Title = "My second Post",
Body = "Hello world, this is my second post"}
          );
      }
  }
```

Update your Database ⬆️

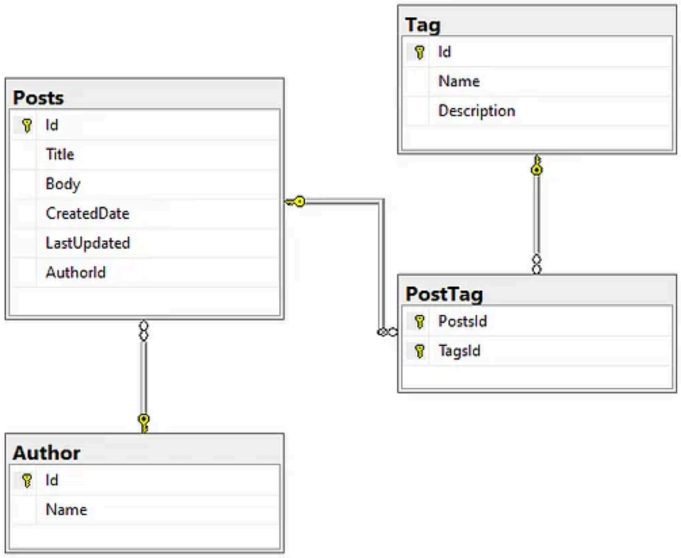
Now that you have all the changes in place the only thing that you need to do is to update your database. For this, I recommend you delete your old database to avoid having problems and to be able to update it seamlessly.

Now perform the next `dotnet ef` commands to add a new migration and update your database.

```
dotnet ef migrations add "Added new entities and applied relationships"

dotnet ef database update
```

Now if you go to your **SQL Server Management System** you should be able to see your database and if you get into the **Database Diagrams** folder it will ask you if you want to create a new diagram, say yes and select the tables you want to include and then you should have something like this.



On this DB diagram we can appreciate that indeed every post can have a single author and an author can have many posts (check the key and kind of infinite symbols). And for the Tag ⇒ Post relationship we can see that we have another intermediate table (junction table) that holds both `Post` and `Tag` id which makes the many-to-many relationship.

Adding functionality to your new Models 💪🏾

Now comes the hard part, you added the entities relationships and the new models as well but know you need to add their controllers, mapping profiles, Dtos, Repositories, and IRepositories to make sure that you can add, create, read, update,e and delete resources for both the `Tag` and `Author` entities.

As I said earlier I won’t be showing the code for all these updates but you can get them here from GitHub.

GitHub — Osempu/BlogAPI at EfCoreRelationships

Test your application 🧪

As always the moment has come to test your application, you could test your API without having the controllers or repositories as you can insert some records directly to **SQL Server** using the **SSMS (SQL Server Management System).** But I would not recommend you this as anyways you will need those controllers and repos in the future so why not code them now right? take it as an assignment.

Conclusion 🌆

Entity Framework Core is an amazing tool that makes dealing with different types of relationships in your database a breeze. Whether you’re working with one-to-one, one-to-many, or many-to-many relationships, Entity Framework Core has got you covered. It even supports self-referencing and required/optional relationships, giving you the flexibility to design your database schema just the way you want. With Entity Framework Core, you can create scalable and maintainable ASP.NET Core applications without breaking a sweat. So, let’s dive in and explore the power of Entity Framework Core’s relationship capabilities. Get ready to unleash your coding skills and enjoy the journey!

Support me on my Journey as a Writer and Content Creator! 🥳🎉

This is the time of the day when I take some time to thank all the amazing people that have supported me and followed this series all along with my other C# projects giving me such energy to continue. But if you are new to my blogs please consider supporting me on my Blog Unit Coding and also on my YouTube channel under the same name where I returned after a long hiatus and plan to continue with the streak, also you can find me on Twitter as @OscarOsempu. Thanks for your time in reading my article I hope you can get tons of value from it and that It solved all your doubts about this amazing topic. Thanks for everything, I'll see you soon in my next article!