Adding Some More Practical Mappings to Your Application

Julie Lerman
Most Trusted Authority on Entity Framework Core

@JulieLerman   www.thedatafarm.com

Please use mappings, don’t change your entities to satisfy EF Core’s behavior

Focus is on making you aware of these capabilities, not a deep dive.

Overview

Understanding EF Core's response to project nullability

Some common conventions and mappings

A look at mapping with data annotations

Persisting enums with EF Core

Converting properties into database types when EF Core doesn't have a mapping

Bulk configurations

Mapping complex types and value objects

This module does not have exercise files.

Understanding How Project Nullability Affects EF Core's String Mappings

Reference Types Can Be Null

For example, a string property or variable can exist without being initialized.

This could lead to NullReferenceException

All New Projects Enable Nullable Reference Type

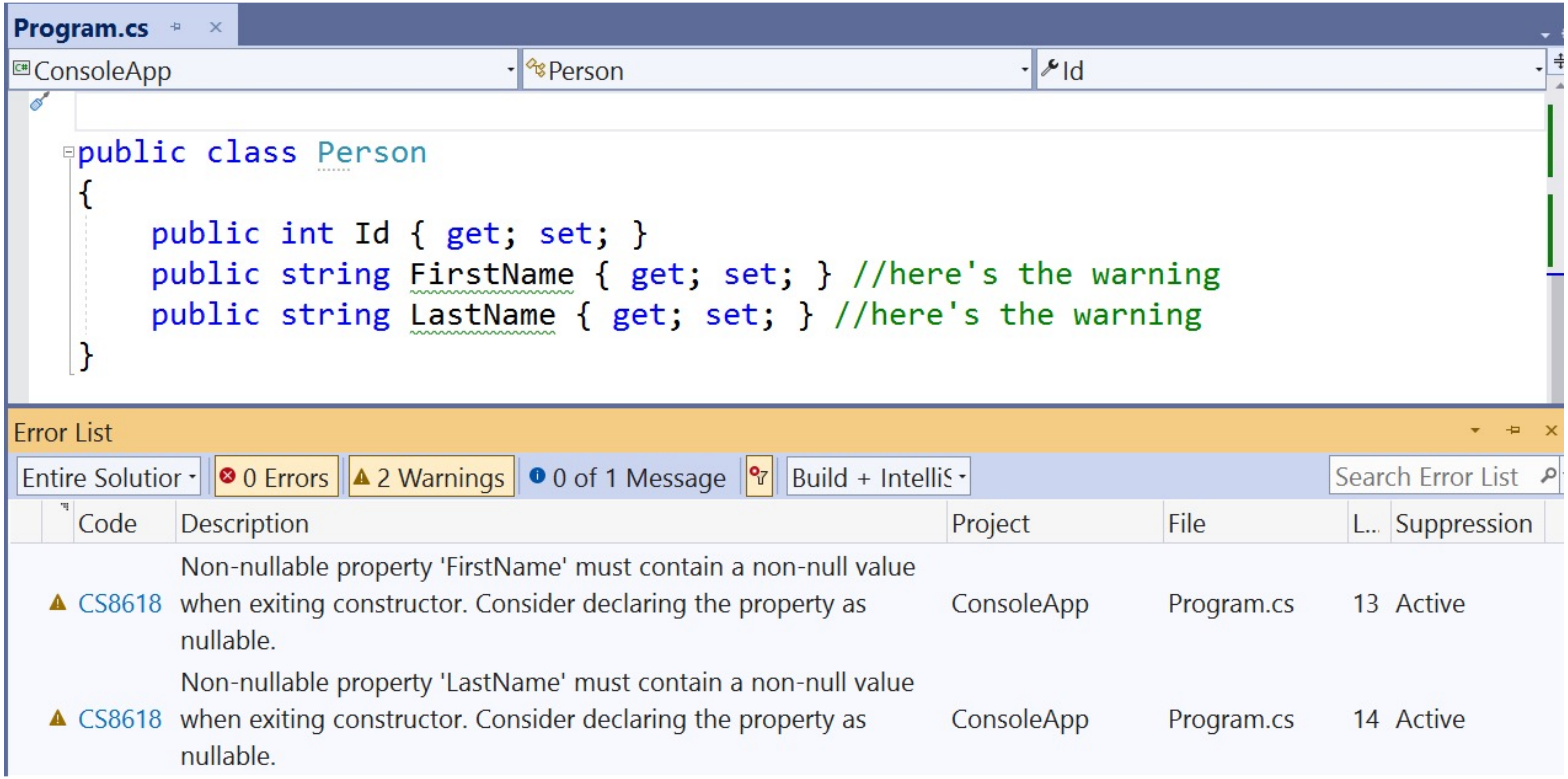
```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

The compiler will warn you where a null value can cause problems

Compiler Warns You of Nullable Properties



```
public class Person
{
    public int Id { get; set; }
    public string FirstName { get; set; } //here's the warning
    public string LastName { get; set; } //here's the warning
}
```

	Code	Description	Project	File	L...	Suppression
⚠	CS8618	Non-nullable property 'FirstName' must contain a non-null value when exiting constructor. Consider declaring the property as nullable.	ConsoleApp	Program.cs	13	Active
⚠	CS8618	Non-nullable property 'LastName' must contain a non-null value when exiting constructor. Consider declaring the property as nullable.	ConsoleApp	Program.cs	14	Active

Without NRT vs. NRT Enabled

<Nullable>disable</Nullable>

Strings are nullable by default

Database column will be nullable

Alternate Mapping

Property( ).IsRequired maps DB columns as non-nullable

DB is only enforcer of the requirement. Provide business logic to protect data.

<Nullable>enable</Nullable>

String props have compiler warnings

Database column will be non-nullable

DB enforces the constraint

Alternate Mapping

`string?:` **Compiler will allow nulls and database column will be nullable**

More on EF Core and Nullability in the Pluralsight EF Core 6 Path

EF Core 6 Best Practices

Michael Perry

Learning Some Additional Common Conventions and Mappings

Some Common Conventions and Mappings

- **Column names match property name.**
  Change with `HasColumnName(“mybettername”)`

- **Column types and length are defined by db provider**
  e.g., SQL Server string default is nvarchar(max)
  Control database type: `HasColumnType(“varchar(500)”)`

- **Configure max length of strings and bytes without changing type:**
  `HasMaxLength(500)`

- **Precision/scale defaults to 18,2. Configure (in supported DBs) e.g.,**
  `.HasPrecision(14, 2)`

More Common Conventions and Mappings

- ✓ **Required & optional driven by .NET. Nullable types e.g. `int?` are mapped to database and honored by compiler. `IsRequired(true/false)` affect db but not compiler.**

- ✓ **Index is created on foreign keys. Use `HasIndex` to change or add more.**

- ✓ **All properties are mapped. Use Ignore to exclude it from database, queries and saves** `modelbuilder.Entity<e>().Ignore(e=>e.Property)`

- ✓ **All reachable entities are mapped.**
  Use `modelBuilder.Ignore<entity>` to exclude

```
modelbuilder.Entity<Author>().AutoInclude(a=>a.Books);
```

New to EF Core 6: AutoInclude Mapping

A rule to always include a navigation or collection property when querying

Even More EF Core Mapping Support

- Inheritance
- Backing Fields
- Concurrency tokens
- Composite keys
- DB value generation
- Splitting entities across tables

Using Data Annotations to Describe Mappings

Why I Favor Fluent API Configurations

Data Annotations provide only a small subset of mappings

Domain classes should not know about persistence

Mappings are not scattered across various classes. All in one place in DbContext.

Fluent API Mappings Override Data Annotations

Conventions	Override with Data Annotations	Override with Fluent Mappings
1	2	3

Some Commonly Used Data Annotations

[Key]

[ForeignKey]

[Required]

[MaxLength]

[KeyLess]

[Index]

Annotations are applied at runtime and ignored by the compiler.

Persisting Enums with EF Core

```
public enum BookGenre
   {
       ScienceFiction,
       Mystery,
       Memoir,
       YoungAdult,
       Adventure,
       HistoricalFiction,
       History
   }
```

Enums in Your Code

Be default, .NET enum types are ints
Int values will be assigned to each member, default 1, 2, 3, etc.

```
public class Book
{
  public int BookId { get; set; }
  public string Title { get; set; }
  public BookGenre Genre { get; set; }    🡐 Enum property
}


mybook.Genre= BookGenre.Memoir;           🡐 Setting the value in code


Database: Books.Genre=3                   🡐 How it's stored in the database
```

Enums in Your Code

EF Core will store the underlying member value into the database
EF Core will translate that value back into the enum when materialized
Also supports bitwise enums

Recommendation: Assign Member Values!

```
public enum BookGenre
   {
       ScienceFiction=1,
       Mystery=2,
       Memoir=3,
       YoungAdult=4,
       Adventure=5,
       HistoricalFiction=6,
       History=7
   }
```

With values assigned…

```
public enum BookGenre
{
  Adventure=5,
  HistoricalFiction=6,
  History=7
  Memoir=3,
  Mystery=2,
  ScienceFiction=1,
  YoungAdult=4,
}
```

You can modify or reorder the list without affecting the stored values

You can use value conversions to force the enums to be stored as text

Mapping “Unmappable” Property Types with Value Conversions

Why Do We Need Value Conversion?

EF Core can map to a pre-defined set of known database types that are common to RDBMS

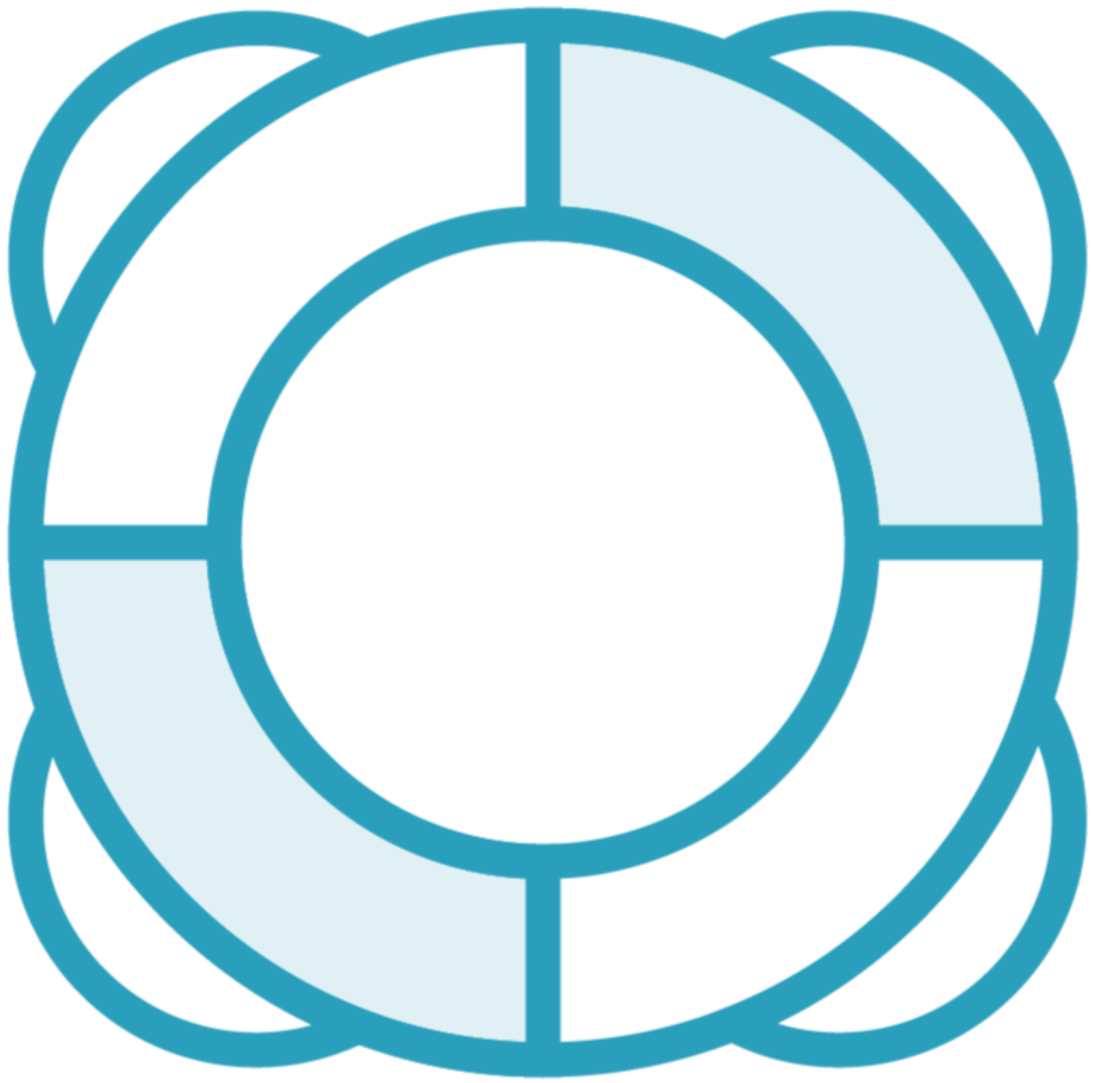
You can help it to map .NET types that don’t have a relevant database type

Storing Color Structs is Hard!

Each color is a property, not a value!

Before value conversions, this was an FAQ

Value Conversions to the Rescue!



Many built-in converters

Create your own custom converter

```
public class Book
{
  public int BookId { get; set; }
  public string Title { get; set; }
  public BookGenre Genre { get; set; }
}

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
   modelBuilder.Entity<Book>().Property(b=>b.Genre).HasConversion<string>();
}

Database: Books.Genre="Memoir"
```

HasConversion with Built-In Converters

Shortcuts will use the appropriate converter

This example will use the EnumtoStringConverter class

List of Built-In Value Converters

Value Conversion article

In EF Core Documentation

```
public class Cover
{
  public int CoverId { get; set; }
  public Color PrimaryColor { get; set; }
}

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  modelBuilder.Entity<Book>().Property(b=>b.PrimaryColor)
   .HasConversion(c=>c.ToString(),s=>Color.FromName(s));
}

Database: Books.PrimaryColor="Blue"
```

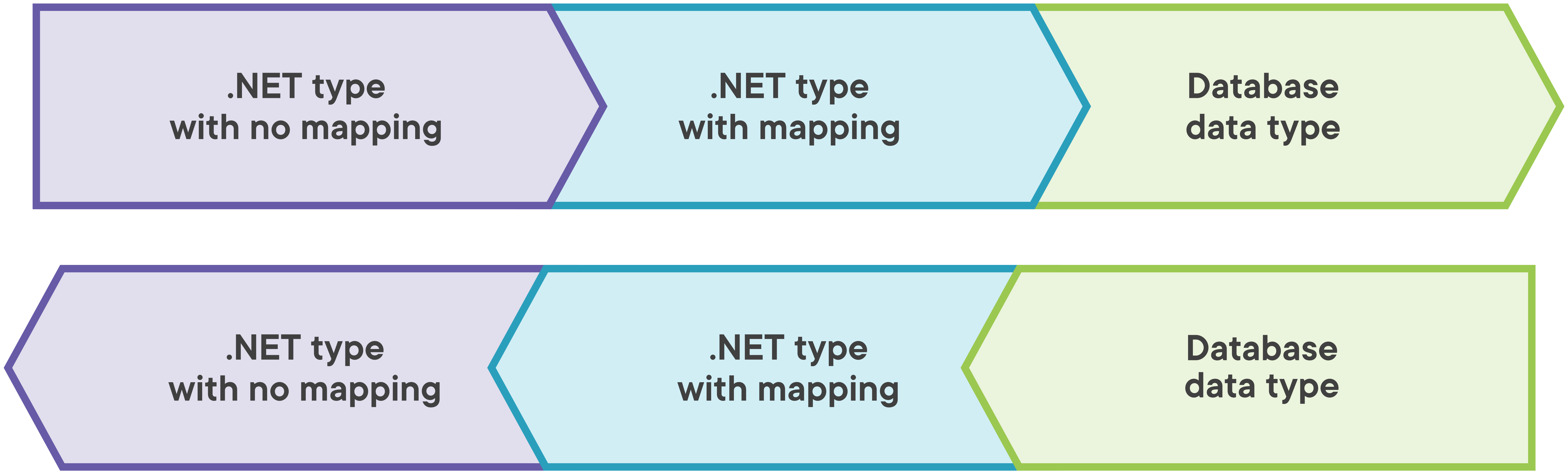
HasConversion with Your Own Conversion

Define a class that inherits from ValueConverter and use that as HasConversion parameter

Or use lambda methods for conversion on storing or retrieving as in this example

ValueConverter Converts Property

DB Provider Converts To/From Data Type



Applying Bulk Configurations and Conversions

Individual configurations for every string

```
modelBuilder.Entity<Author>().Property(a => a.FirstName).HasColumnType("varchar(100)");
modelBuilder.Entity<Author>().Property(a => a.LastName).HasColumnType("varchar(100)");
modelBuilder.Entity<Artist>().Property(a => a.FirstName).HasColumnType("varchar(100)");
modelBuilder.Entity<Artist>().Property(a => a.LastName).HasColumnType("varchar(100)");
```

Bulk configuration for all strings

```
protected override void ConfigureConventions(ModelConfigurationBuilder
  configurationBuilder)
{
    configurationBuilder.Properties<string>().HaveColumnType("varchar(100)");
}
```

Bulk Configuration with “Have” Methods

Use DbContext.ConfigureConventions virtual method
Apply configuration to configurationBuilder.Properties<T>
Override anomalies with individual configuration

Bulk Value Conversions

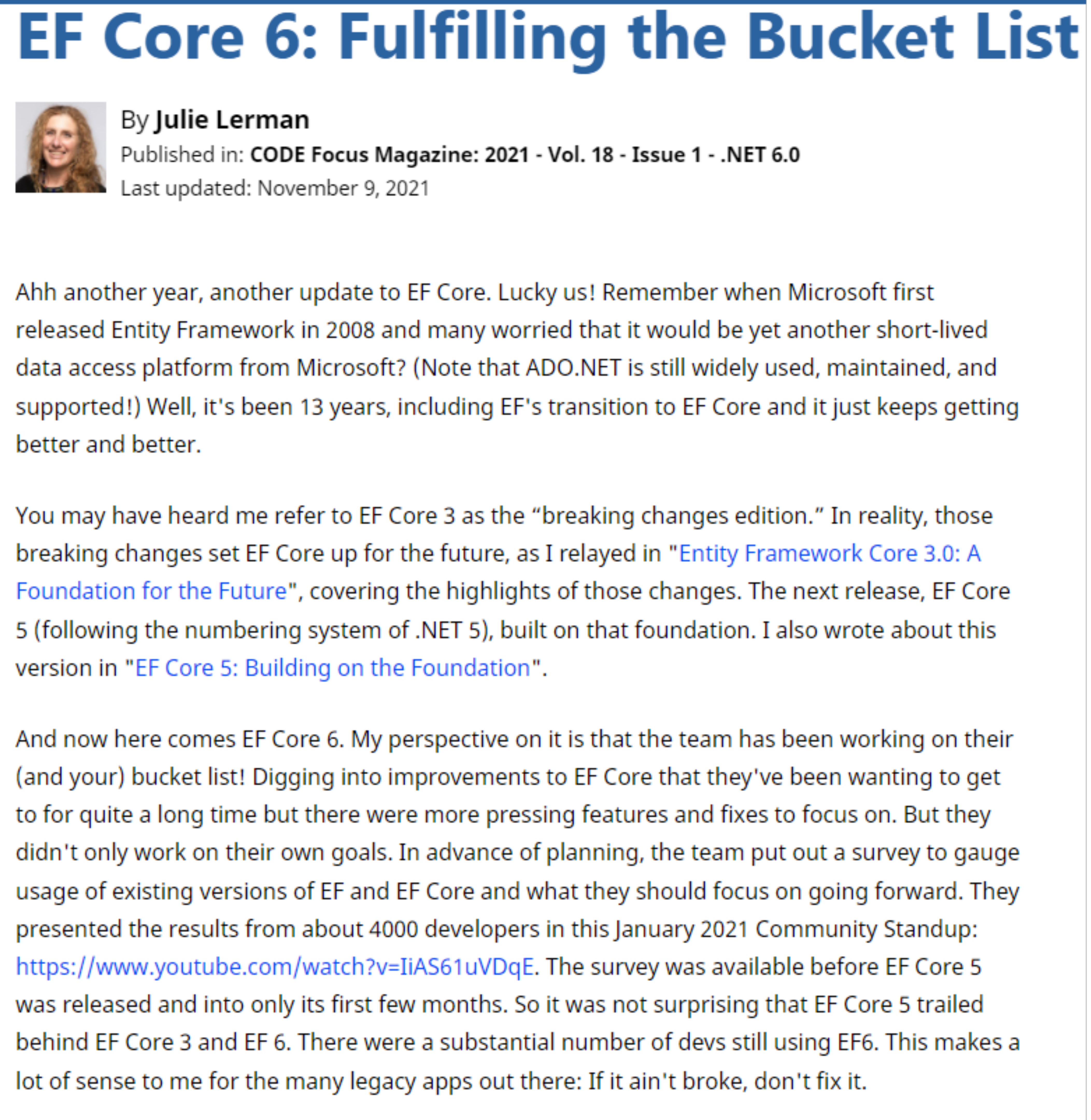
Using a built-in conversion

```
configurationBuilder.Properties<BookGenre>).HaveConversion<string>();
```

Custom Conversion Requires a Custom Class

```
configurationBuilder.Properties<Color>().HaveConversion(typeof(ColorToString));

public class ColorToString : ValueConverter<Color, string>
{
    public ColorToString() : base(ColorString, ColorStruct) { }
    private static Expression<Func<Color, string>>
          ColorString = v => new String(v.Name);
    private static Expression<Func<string, Color>>
          ColorStruct = v => Color.FromName(v);
}
```



EF Core 6: Fulfilling the Bucket List

By **Julie Lerman**
Published in: **CODE Focus Magazine: 2021 - Vol. 18 - Issue 1 - .NET 6.0**
Last updated: November 9, 2021

Ahh another year, another update to EF Core. Lucky us! Remember when Microsoft first released Entity Framework in 2008 and many worried that it would be yet another short-lived data access platform from Microsoft? (Note that ADO.NET is still widely used, maintained, and supported!) Well, it's been 13 years, including EF's transition to EF Core and it just keeps getting better and better.

You may have heard me refer to EF Core 3 as the "breaking changes edition." In reality, those breaking changes set EF Core up for the future, as I relayed in "Entity Framework Core 3.0: A Foundation for the Future", covering the highlights of those changes. The next release, EF Core 5 (following the numbering system of .NET 5), built on that foundation. I also wrote about this version in "EF Core 5: Building on the Foundation".

And now here comes EF Core 6. My perspective on it is that the team has been working on their (and your) bucket list! Digging into improvements to EF Core that they've been wanting to get to for quite a long time but there were more pressing features and fixes to focus on. But they didn't only work on their own goals. In advance of planning, the team put out a survey to gauge usage of existing versions of EF and EF Core and what they should focus on going forward. They presented the results from about 4000 developers in this January 2021 Community Standup: https://www.youtube.com/watch?v=IiAS61uVDqE. The survey was available before EF Core 5 was released and into only its first few months. So it was not surprising that EF Core 5 trailed behind EF Core 3 and EF 6. There were a substantial number of devs still using EF6. This makes a lot of sense to me for the many legacy apps out there: If it ain't broke, don't fix it.

More on bulk configuration and other advanced features introduced in EF Core 6

codemag.com/Article/2111072

Mapping Complex Types and Value Objects

Not every property is a scalar type or a relationship!

Before

```
public class Author
{
  public int AuthorId {get; set;}
  public string FirstName { get; set;}
  public string LastName { get; set; }
  public List<Book> Books { get; set; }
  public Full=>$"{FirstName} {LastName}";


public class Artist
{
  public int ArtistId {get; set;}
  public string FirstName { get; set;}
  public string LastName { get; set; }
  public List<Cover> Covers {get; set; }
  public Full=>$"{FirstName} {LastName}";
}
```

- **Author has first name and last name and composed property, full name**
- **Artist has first name and last name and composed property, full name**

After

```
public class PersonName
{
  public string First { get; set;}
  public string Last { get; set; }
  public string Full=>$"{First} {Last}";
  public string Reverse=>
      $"{Last}, {First}“;
}

public class Author
{
  public int AuthorId {get; set;}
  public PersonName Name { get; set;}
  public List<Book> Books { get; set; }
}

public class Artist
{
  public int ArtistId {get; set;}
  public PersonName Name { get; set;}
  public List<Cover> Covers {get; set; }
}
```

◂ **New type that encapsulates the repeated members and logic**

No key property!

◂ **Author & Artist both use the new type**

Access e.g.,

`Author.Name.FirstName`

`Author.Name.Reverse`

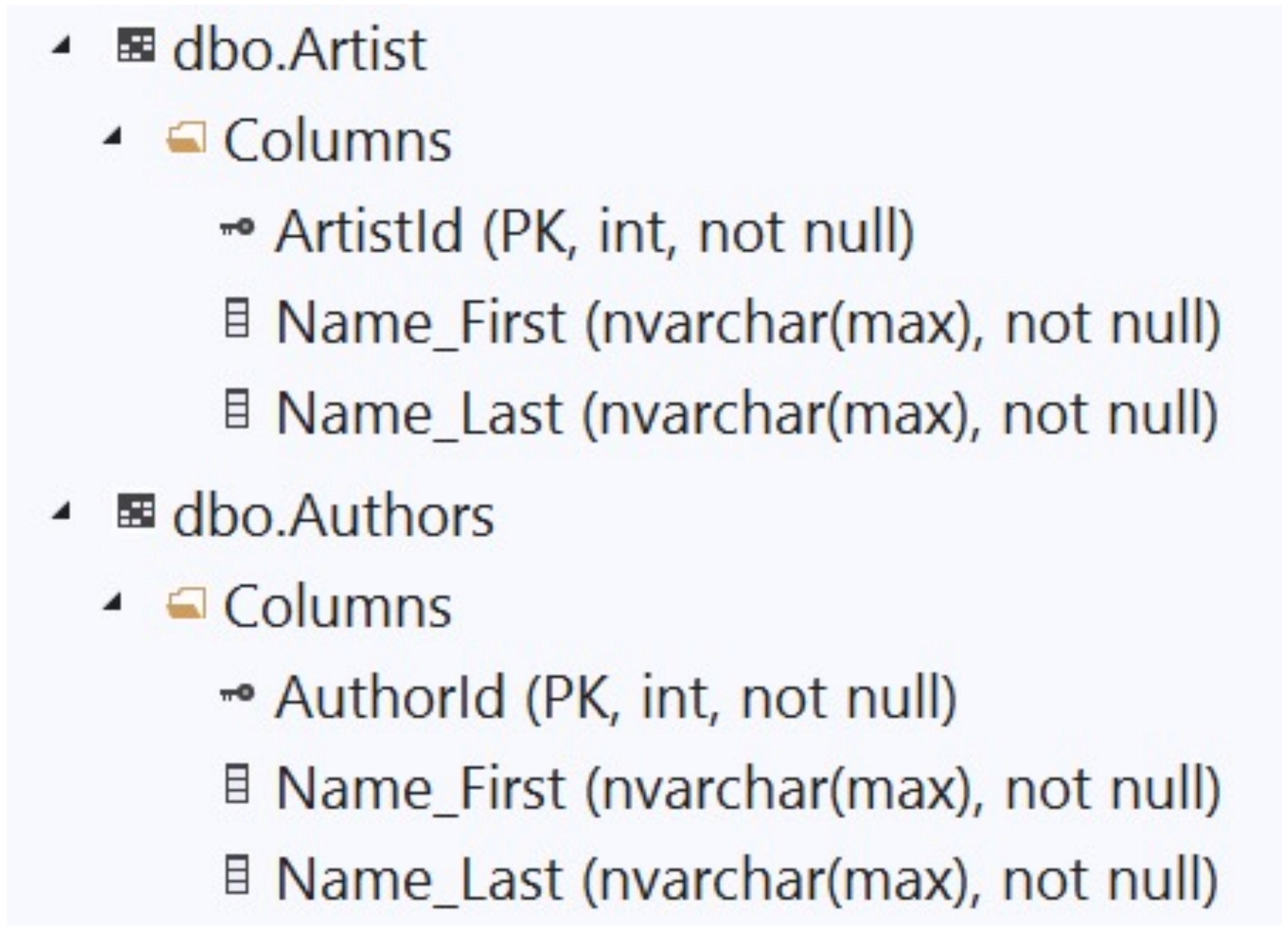
```
modelBuilder.Entity<Author>().OwnsOne(a => a.Name);

modelBuilder.Entity<Artist>().OwnsOne(a => a.Name);
```

You Can Map These as “Owned Entities”

Convention will not recognize them
You have to map it for each “owner”
The properties of the owned type are, by default, stored as columns in the owner table
Remember it must have NO KEY property

Default Mapping of Owned Entities



dbo.Artist
- Columns
  - ArtistId (PK, int, not null)
  - Name_First (nvarchar(max), not null)
  - Name_Last (nvarchar(max), not null)

dbo.Authors
- Columns
  - AuthorId (PK, int, not null)
  - Name_First (nvarchar(max), not null)
  - Name_Last (nvarchar(max), not null)

Learn More About DDD on Pluralsight

Domain-Driven Design Fundamentals

Steve Smith and Julie Lerman

If you are designing via Domain-Driven Design, owned entities can be used to map value objects

Review

Mappings provide a rich means of overriding those defaults

EF Core respects project NRT setting

A subset of data annotations are recognized by the model builder

Enums have been supported for long time

Map “unmappable” types with value conversions

Define bulk mappings and conversions

Owned entities let you map complex types and value objects

Reminder:
This module does not have exercise files.

Up Next: Understanding EF Core’s Database Connectivity

Resources

Entity Framework Core on GitHub: github.com/dotnet/efcore

EF Core Documentation: docs.microsoft.com/ef

EF Core 6: Fulfilling the Bucket List:
codemag.com/Article/2111072

Domain-Driven Design Fundamentals on Pluralsight
bit.ly/DDDPluralsight