Adding Some More Practical Mappings to Your Application

Julie Lerman

EF Core Expert and Software Coach

@julielerman | 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.

There won’t be a walk through, so this module only has an “after” solution.

Module Overview

Understanding EF Core's response to project nullability

Some common conventions and mappings

A look at mapping with data annotations

Storing (certain) properties as JSON

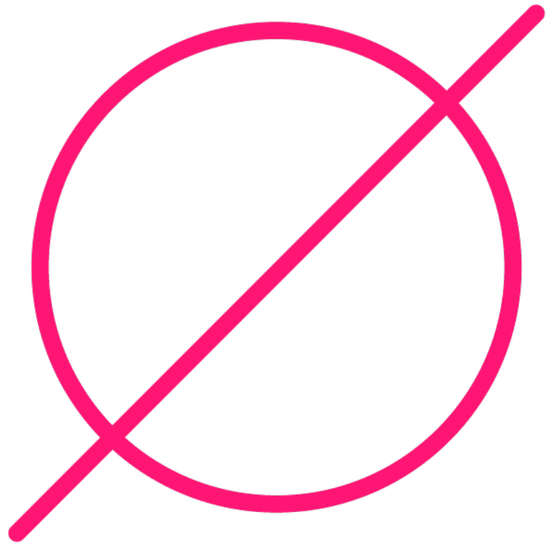
Converting properties into database types when EF Core doesn't have a mapping

Bulk configurations

Mapping complex types and value objects

Understanding How Project Nullability Affects EF Core's String Mappings

Reference Types Can Be Null



E.g., a string property or variable can exist without being initialized.

This could lead to NullReferenceException

All New Projects Enable Nullable Reference Type (NRT)

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

  <PropertyGroup>
    <TargetFramework>net8.0</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

Program.cs

ConsoleApp | Person | Id

```
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
}
```

Error List

Entire Solution | 0 Errors | 2 Warnings | 0 of 1 Message | Build + IntelliSense | Search Error List

	Code	Description	Project	File	Line	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

Strings are nullable by default

With <Nullable>disable</Nullable>

Database column will be nullable

```
migrationBuilder.CreateTable(
    name: "Authors",
    columns: table => new
    {
        AuthorId = table.Column<int>(type: "int", nullable: false)
            .Annotation("SqlServer:Identity", "1, 1"),
        FirstName = table.Column<string>(type: "nvarchar(100)", nullable: true),
        LastName = table.Column<string>(type: "nvarchar(100)", nullable: true)
    },
```

Alternate Mapping

`modelBuilder.Entity().Property( ).IsRequired`
forces DB columns as non-nullable

But then database is only enforcer of the requirement.
You must provide business logic to protect from errors.

With <Nullable>enable</Nullable>

Strings have compiler warnings

```
public class Author
{
    public int AuthorId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public List<Book> Books { get; set; }
        =new();
}
```

Database columns are non-nullable

```
FirstName = table.Column<string>
  (type: "nvarchar(100)", nullable: false),
LastName = table.Column<string>
   (type: "nvarchar(100)", nullable: false)
```

Other nullables have warnings, too

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

Force nullability via nullable types

```
public class Author
{
    public int AuthorId { get; set; }
    public string? FirstName { get; set; }
    public string? LastName { get; set; }
    public List<Book> Books { get; set; }
        =new();
}
```

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.

```
public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public DateOnly PublishDate { get; set; }  //Time of day is inconsequential!
    public decimal BasePrice { get; set; }
}
```

DateOnly & TimeOnly (SQL Server) Mapping is New to EF Core 8

Thanks to Erik Ejlskov Jensen for this important contribution to EF Core!
DateOnly maps to DATE, TimeOnly maps to TIME
Using DateTime when you don't need the time data adds complexity
DateOnly is quite often what you are looking for in your classes

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

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

Even More EF Core Mapping Support

Inheritance

Backing Fields

Concurrency tokens

And enums, too!

Composite keys

DB value generation

Splitting entities across tables

Storing Sub-Types and Primitive Collections as JSON

Much more useful than just shoving JSON in string properties

Map a Subtype as JSON in the Database

You'll need a mapping in OnModelCreating

```
public class ContactDetails
{
    public string EmailAddress { get; set; }
    public string Phone { get; set; }
}
public class Author
{
    ...
    public ContactDetails Contact { get; set; } = new();
}
```

	Author...	FirstNa...	LastName	Contact
	1	William	Shakespeare	{"EmailAddress":"will@upstart.crow","Phone":"555-555-1234"}

Map a Subtype as JSON in the Database

{JSON} Requires a mapping in OnModelCreating

{JSON} Better than just using JSON in string properties

{JSON} Allows querying, filtering, projecting on the type's properties

{JSON} Supports updates and SaveChanges

Primitive Collections Map to JSON

```
public class Author
{     ...
      public ICollection<string> Nicknames { get; set; }
}


var author = new Author
{
   FirstName = "William", LastName = "Shakespeare",
   Nicknames=new List<string> { "The Bard", "The Bard of Avon","Swan of Avon"  }
};
```

	A...	FirstNa...	LastNa...	Contact	Nicknames
▸	1	William	Shakes...	{"EmailAddress":"will@upstart.crow","Phone":"55...	
	2	William	Shakes...	{"EmailAddress":null,"Phone":null}	["The Bard","The Bard of Avon","Swan of Avon"]

Mapping “Unmappable” Property Types with Value Conversions

Why Do We Need Value Conversion?

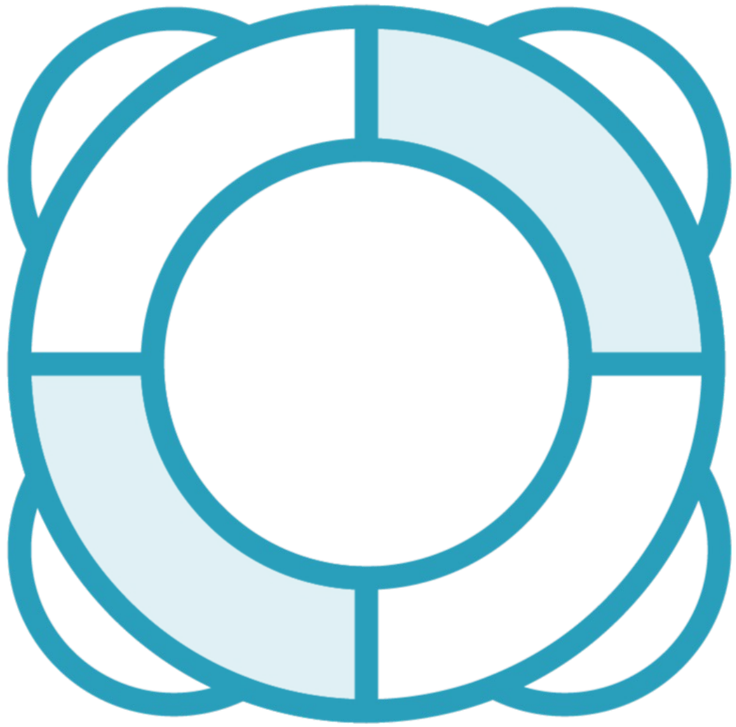
EF Core can map to a pre-defined set of known database types that are common to RDBMS

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"
```

BookGenre is an enum

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"
```

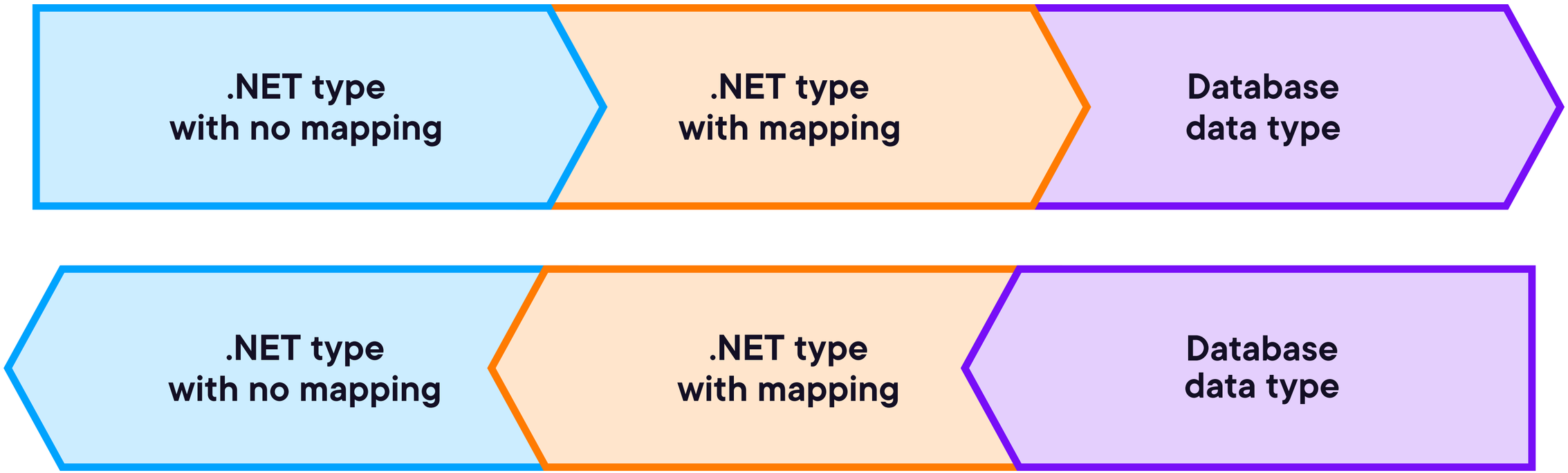
HasConversion with Your Own Conversion

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>();
```

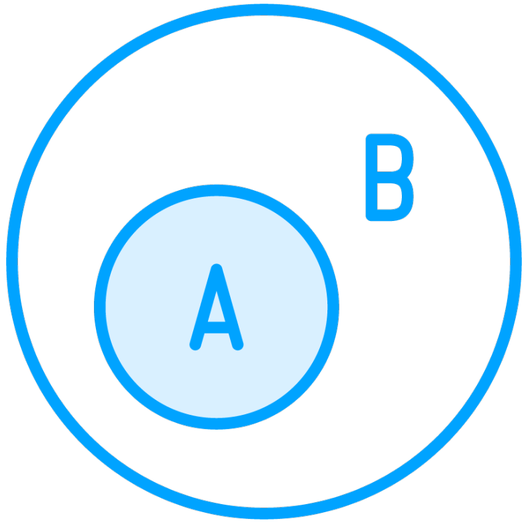
Custom Conversion Requires a Custom Class

```
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);
}
```

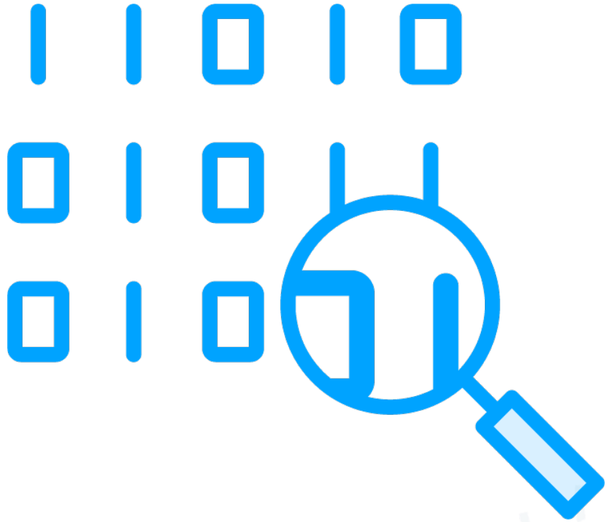
Using Data Annotations to Describe Mappings

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

1

Override with Data Annotations

2

Override with Fluent Mappings

3

Some Commonly Used Data Annotations

[Key]

[ForeignKey]

[Required]

[MaxLength]

[KeyLess]

[Index]

Annotations are applied at runtime and ignored by the compiler.

Note: I have removed this enum clip from the course in favor of the JSON clip. However, I'm leaving these slides in here for your benefit.

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

Sentinel value support can also help!

New to EF Core 8!

See “What’s New in EFCore 8” in docs

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

Mapping Complex Types and DDD 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.,

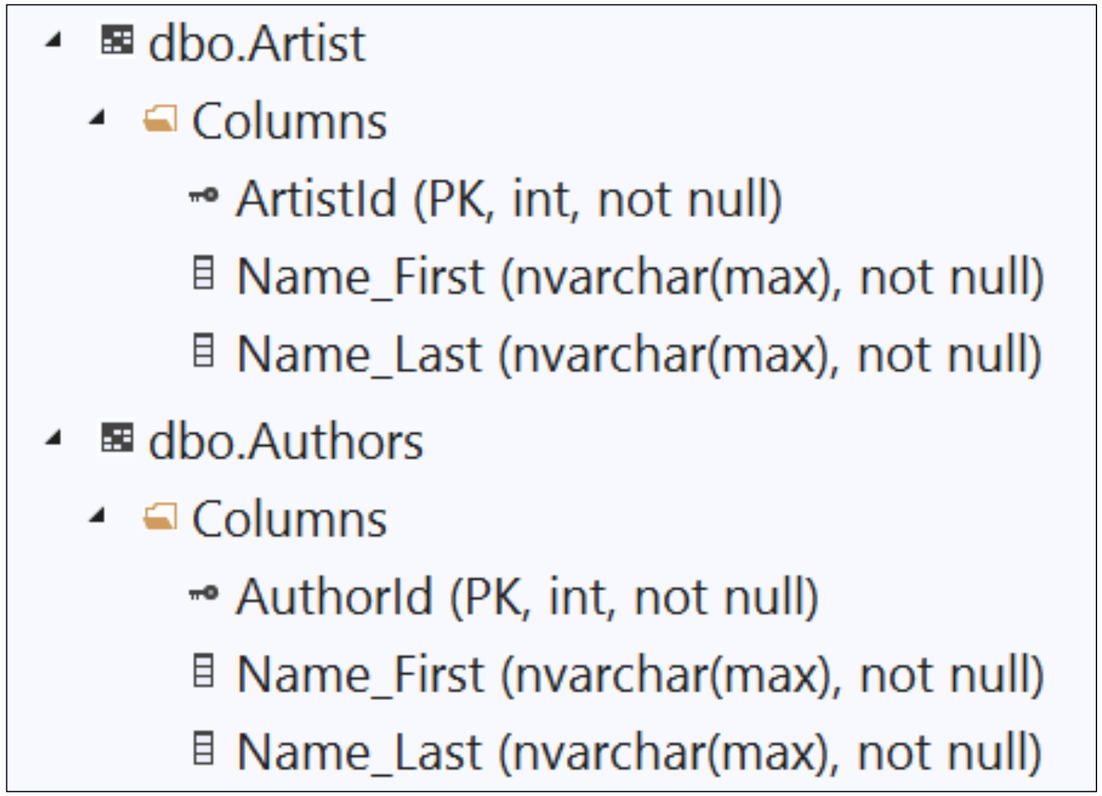
```
Author.Name.FirstName
Author.Name.Reverse
```

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

You Can Map These as a ComplexProperty (New to EF Core 8!)

Can replace most uses of owned entities
Convention will not recognize them
You have to map it for each “host”
The properties of the complex type are, by default, stored as columns in the host’s table
Remember it must have NO KEY property

Default ComplexProperty Columns in Database



Learn More About DDD on Pluralsight

Domain-Driven Design Fundamentals

Steve Smith and Julie Lerman

EF Core and Domain-Driven Design

Julie Lerman

You can map value objects with ComplexProperty, and its better than Owned Entity mappings

ComplexType Mapping is Superior to Owned Entities

Less “trickery” under the covers

Fewer side effects

More usage scenarios

Is it obvious that I love the ComplexProperty mapping?

Review

- Conventions map many .NET to DB types
- Configurations 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
- Type properties & primitive lists to JSON
- Value conversions for “unmappable” type
- Define bulk mappings and conversions
- ComplexProperty for complex types and value objects

Reminder:
This module only has an “after” solution.

Up Next:

Understanding EF Core's Database Connectivity

Resources

What's New in EF Core 8 docs:
learn.microsoft.com/ef/core/what-is-new/ef-core-8.0/whatsnew

EF Core Documentation on ComplexProperty Mapping:
learn.microsoft.com/ef/core/what-is-new/ef-core-8.0/whatsnew#value-objects-using-complex-types

EF Core 6 & Domain-Driven Design on Pluralsight bit.ly/EFCoreDDD
Watch for EF Core 8 update with ComplexProperty mapping

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

More Resources

EF Core Team Standup on Complex Types bit.ly/ComplexStandup

All EF Core Community Standups (YouTube playlist): bit.ly/EFCoreStandup