

# Defining One-to-Many Relationships



**Julie Lerman**

EF Core Expert and Software Coach

@julielerman | thedatafarm.com



# Module Overview



- Install a tool to visualize the data model
- How EF Core interprets various one-to-many setups
- Benefits of foreign key properties
- Modify the entity classes
- Seed related data
- Use migrations to evolve the database to match a modified model
- Using mappings to guide EF Core when needed



# Visualizing EF Core's Interpretation of Your Data Model



# How EF Core Determines Mappings to DB

Reminder

## Conventions

### Default assumptions

```
property name=column name
```

## Override with Fluent Mappings

### Apply in DbContext using Fluent API

```
modelBuilder.Entity<Book>()  
.Property(b => b.Title)  
.HasColumnName("MainTitle");
```

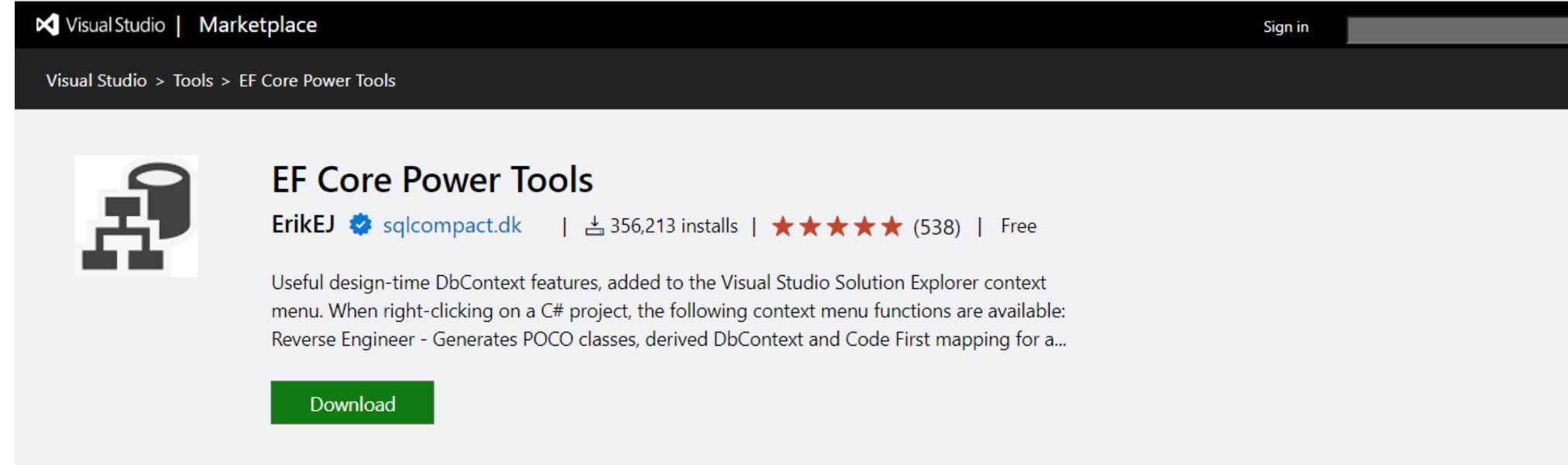
## Override with Data Annotations

### Apply in entity

```
[Column("MainTitle")]  
public string Title{get;set;}
```



# EF Core Power Tools (Visual Studio) Extension



The screenshot shows the Visual Studio Marketplace page for the "EF Core Power Tools" extension. At the top, there's a navigation bar with "VisualStudio | Marketplace" and a "Sign in" button. Below it, the breadcrumb navigation shows "Visual Studio > Tools > EF Core Power Tools". The main content area features a large icon of a database with three tables, followed by the title "EF Core Power Tools" and the developer name "ErikEJ". It also displays "sqlcompact.dk", "356,213 installs", a 5-star rating from 538 reviews, and the word "Free". A brief description follows, mentioning design-time DbContext features and context menu functions like Reverse Engineer and Migrations Tool. A prominent green "Download" button is at the bottom. Below the main content, there are tabs for "Overview" (which is selected) and "Rating & Review". The "Overview" section contains links to "Getting Started" and "Feedback", along with detailed descriptions of the extension's features and how it integrates with Visual Studio. To the right, there are sections for "Categories" (Tools, Data, Modeling, Scaffolding), "Tags" (Entity Framework, Scaffolding, SQL Server, sqlite), "Works with" (Visual Studio 2022 (amd64)), "Resources" (License, Copy ID), and "Project Details" (GitHub link, No Pull Requests). There is also a "Copy" button.

EF Core Power Tools

ErikEJ · sqlcompact.dk · 356,213 installs · ★★★★★ (538) · Free

Useful design-time DbContext features, added to the Visual Studio Solution Explorer context menu. When right-clicking on a C# project, the following context menu functions are available:

Reverse Engineer - Generates POCO classes, derived DbContext and Code First mapping for a...

Download

Overview Rating & Review

[Getting Started](#) | [Feedback](#)

Useful design-time DbContext features, added to the Visual Studio Solution Explorer context menu.

When right-clicking on an applicable C# project, the following context menu functions are available:

**Reverse Engineer** - Generates POCO classes, derived DbContext and mappings for an existing Azure SQL Database, SQL Server, SQLite, Postgres, MySQL, Firebird or Oracle database, a SQL Server Database project or a .dacpac file. Offers an advanced UI for selecting database objects, including views, stored procedures and functions, and preserves all options in a configuration file in your project.

**Migrations Tool** - Manage EF Core Migrations in the project, get migration status, add migration and update the database to keep your model and database in sync.

**Add DbContext Model Diagram** - Adds a DGML graph from your DbContext Model.

**View DbContext Model DDL SQL** - View the SQL CREATE script for the current Model

**View DbContext Model as DebugView** - View the current Model as DebugView text

Categories

Tools Data Modeling Scaffolding

Tags

Entity Framework Scaffolding SQL Server sqlite

Works with

Visual Studio 2022 (amd64)

Resources

[License](#) [Copy ID](#)

Project Details

[ErikEJ/EFCorePowerTools](#) [No Pull Requests](#)

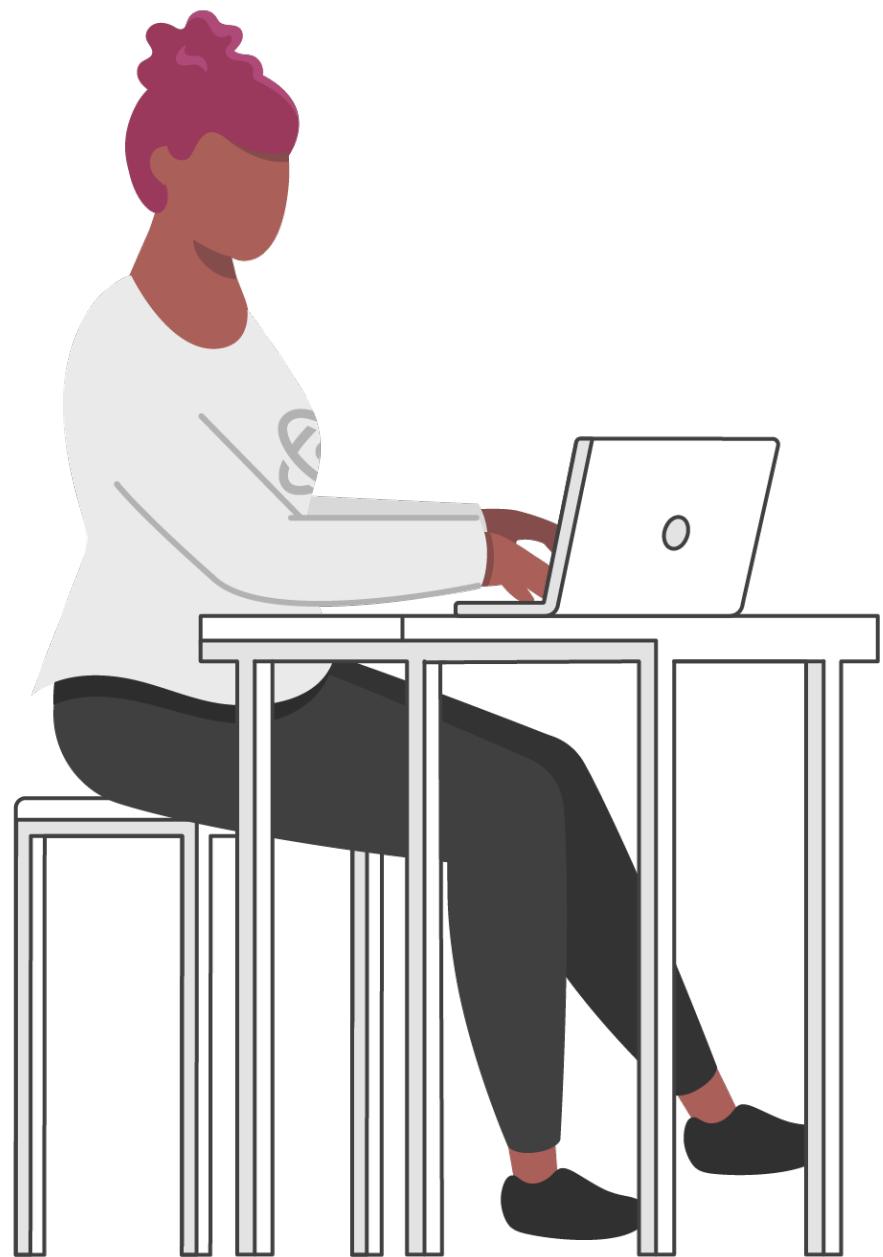
And open source at: [github.com/ErikEJ/EFCorePowerTools](https://github.com/ErikEJ/EFCorePowerTools)





# Interpreting One-to-Many Relationships

# One Author Can Have Many Books

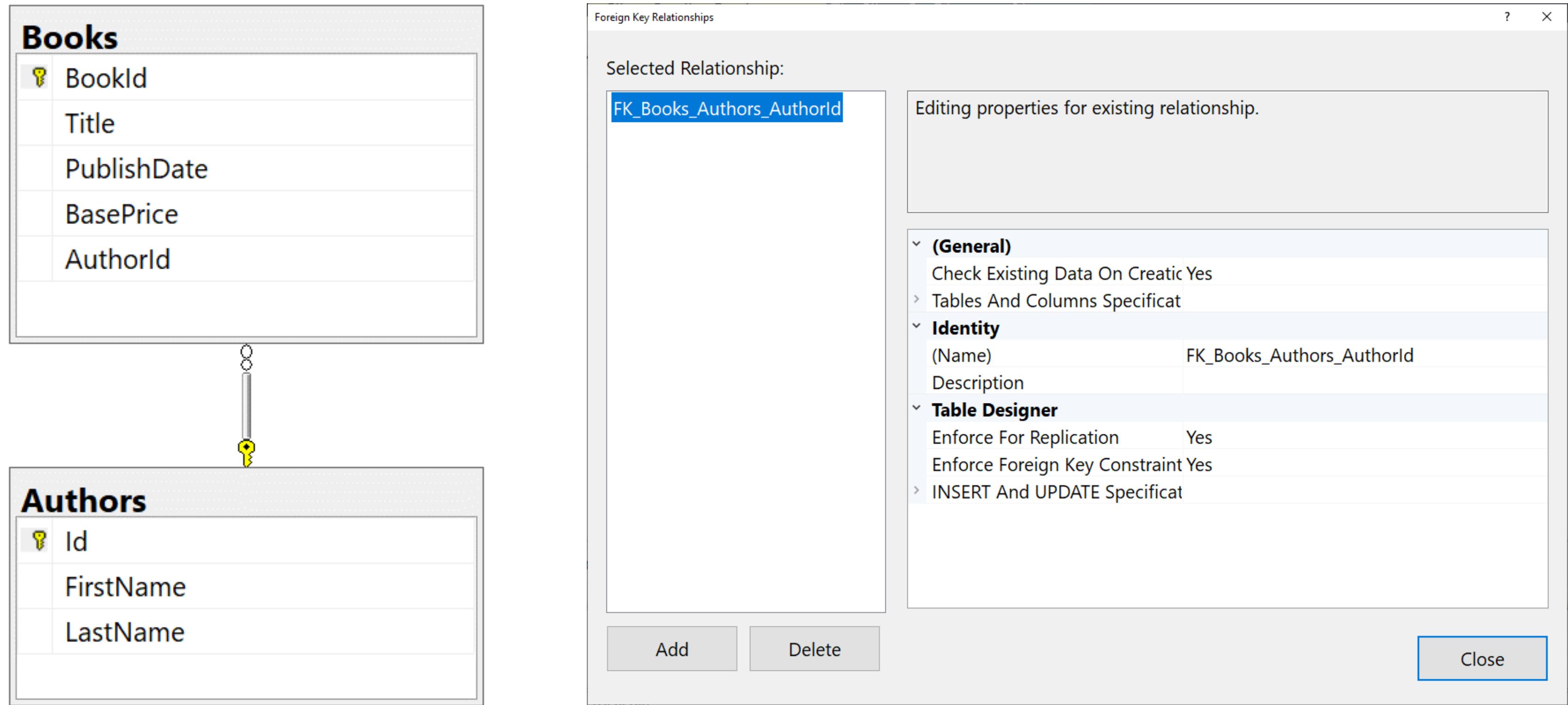


# Convention Over Configuration

**Default behavior that can be overridden using configurations.**



# One-to-Many in the Database



Views from SQL Server Management Studio



# Shadow Properties

**Properties that exist in the data model but not the entity class. These can be inferred by EF Core or you can explicitly define them in the DbContext.**



# **Relationship Terminology**

**Parent / Child**

**Principal / Dependent**



# Reference from parent to child is sufficient

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

```
public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
}
```

◀ List<Child> in Parent

◀ This Child has no references back to Parent  
◀ Foreign Key (e.g, AuthorId) will be inferred  
in database

# Child has navigation prop pointing to Parent

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

```
public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public Author Author { get; set; }
}
```

◀ List<Child> in Parent

- ◀ This child has a reference back to parent aka “Navigation Property”
- ◀ It already understands the one-to-many because of the parent’s List<Child>
- ◀ The reference back to parent aka “Navigation Property” is a bonus
- ◀ AuthorId FK will be inferred in the database

# FK is recognized because of reference from parent

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

```
public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public int AuthorId { get; set; }
}
```

◀ List<Child> in Parent

◀ AuthorId is recognized as foreign key for two reasons:

- 1) the known relationship from parent
- 2) follows FK naming convention (type + Id)

# Child has navigation prop pointing to Parent & an FK

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

```
public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public Author Author { get; set; }
    public int AuthorId {get;set;}
}
```

◀ List<Child> in Parent

◀ One to many is known because of List<Child>  
◀ Navigation and FK are bonus

# No detectable relationship

```
public class Author
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public int AuthorId {get;set;}
}
```

◀ Parent has no knowledge of children

◀ This AuthorId property is just a random integer



You can achieve so  
much with  
**mappings**

Here's a peek at an example:  
mapping a relationship that is  
completely unconventional.



# Configuring a One-to-Many

When convention can't discover it because there are no references

PubContext.cs

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Author>()
        .HasMany<Book>()
        .WithOne();
}
```



# Benefitting from Foreign Key Properties



**Foreign Key properties  
are your friends.**

**And sometimes, navigations  
are just in the way.**



# Tying a Book to an Author the Easy Way

mybook.AuthorId=23





# When/Why Ditch Navigation to the Parent?



# Considering the Parent Navigation Property

Coming from  
earlier versions?



EF Core is now smarter about missing  
navigation data, but not in all scenarios.

Tip: Default to no  
navigation property



Only add it if your biz logic requires it!





# Why have FK property?



```
public class Book
{
    public int Id {get;set;}
    public string Title {get;set;}
    public Author Author{get;set;}
}
```

```
public class Book
{
    public int Id {get;set;}
    public string Title {get;set;}
    public Author Author{get;set;}
    public int AuthorId {get;set;}
}
```

```
public class Book
{
    public int Id {get;set;}
    public string Title {get;set;}
    public int AuthorId {get;set;}
}
```

◀ Only a navigation property to Author.  
You must have an author in memory to  
connect a book.

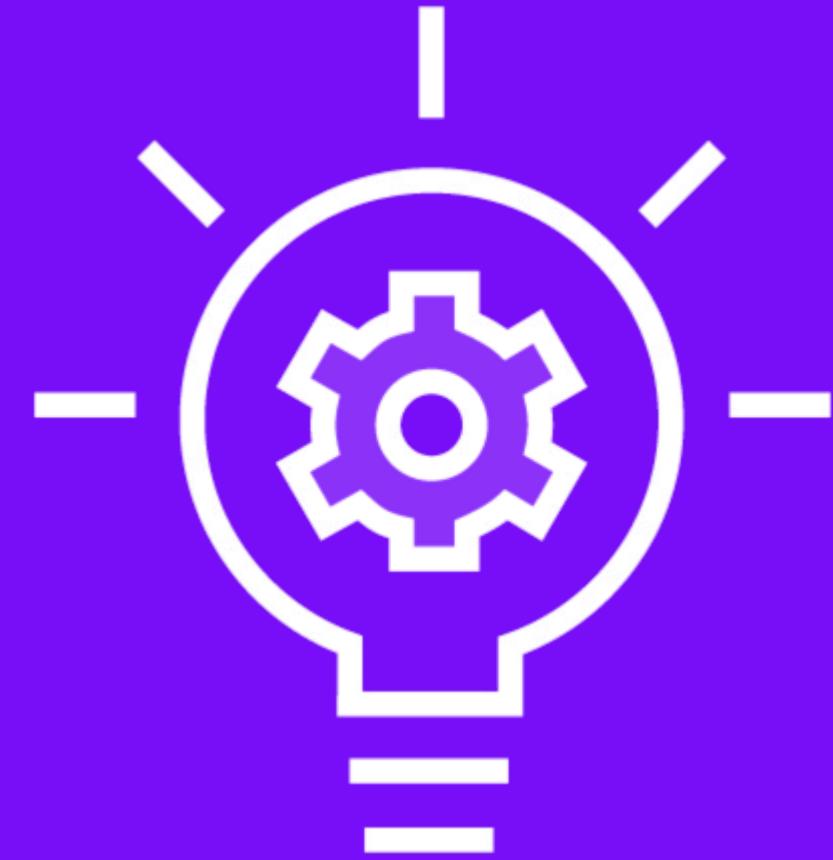
```
author.Books.Add(abook)
abook.Author=someauthor
```

◀ With a foreign key property, you don't  
need an Author object.

```
book.AuthorId=1
```

◀ ...and you can even eliminate the  
navigation property if your logic doesn't  
need it.





# Foreign Key Properties & HasData Seeding

HasData requires explicit primary and foreign key values to be set. With AuthorId now in Book, you can seed book data as well.



```
modelBuilder.Entity<Author>().HasData(new Author {Id=1, FirstName="Julie", ...});  
modelBuilder.Entity<Book>().HasData(  
    new Book {BookId=1, AuthorId=1, Title="Programming Entity Framework"});
```

## Seeding Related Data

Provide property values including keys from and foreign keys from “Parent”  
HasData will get interpreted into migrations  
Inserts will get interpreted into SQL  
Data will get inserted when migrations are executed



# My Author & Book Key Properties Use Different Naming Conventions!

That's not a recommended coding practice.  
Let's fix it.



# Changes to My Model



**Added AuthorId foreign key property to book**



**Added seed data for books via HasData in PubContext**



**Modified the Author's key property from *Id* to *AuthorId***



# Mapping Unconventional Foreign Keys



# Following along with code?

I've reverted experiments  
from previous clips!



# **EF Core should not drive how you design your business logic**



# Configuring a Non-Conventional Foreign Key

FK is tied to a relationship, so you must first describe that relationship

PubContext.cs

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Author>()
        .HasMany(a => a.Books)
        .WithOne(b => b.Author)
        .HasForeignKey(b => b.AuthorFK);
}
```

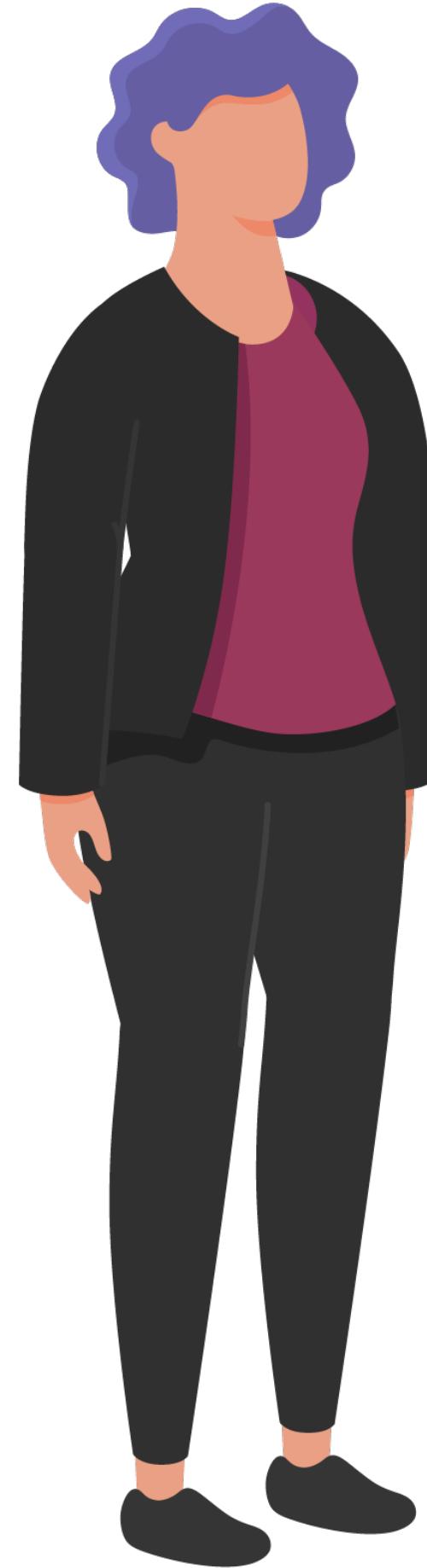


# **Understanding Nullability and Required vs. Optional Principals**



**By default, every  
dependent must have a  
principal, but  
EF Core does not enforce  
this**





**Ooh! We should publish a book on all the  
funny things our pets do!**

**Now we'll have to find an author.**



# Allowing Optional Parent

```
public class Book
{
    ...
    public int? AuthorId { get; set; }
}
```

```
modelBuilder.Entity<Author>()
    .HasMany(a => a.Books)
    .WithOne(b => b.Author)
    .HasForeignKey(b => b.AuthorId)
    .IsRequired(false);
```

```
modelBuilder.Entity<Author>()
    .HasMany(a => a.Books)
    .WithOne(b => b.Author)
    .HasForeignKey("AuthorId")
    .IsRequired(false);
```

◀ Specify the FK property is nullable

◀ Map the FK property as not required

◀ When FK property doesn't exist, map the inferred (“shadow”) property as not required

## Summary



**Many ways to describe one-to-many that conventions will recognize**

**You can add mappings if those patterns don't align with your desired logic**

**Foreign keys are your friends**

**An unconventional FK name is a common “gotcha” which you can fix in mappings**

**Watch out for required principals!**

**Use the EF Core Power Tools to verify how EF Core will interpret the data model**



**Up Next:**

# **Logging EF Core Activity and SQL**

---



# Resources

Entity Framework Core on GitHub: [github.com/dotnet/efcore](https://github.com/dotnet/efcore)

EF Core Documentation: [docs.microsoft.com/ef/core](https://docs.microsoft.com/ef/core)

EF Core Power Tools Extension (model visualizer, scaffold and more):  
<https://github.com/ErikEJ/EFCorePowerTools>

