Defining One-to-Many Relationships

Julie Lerman

Most Trusted Authority on Entity Framework Core

@julielerman thedatafarm.com

Module Overview

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

Visualizing EF Core's Interpretation of Your Data Model

Reminder:
How EF Core Determines Mappings to DB

Conventions

Default assumptions

```
property name=column name
```

Override with Fluent Mappings

Apply in DbContext using Fluent API

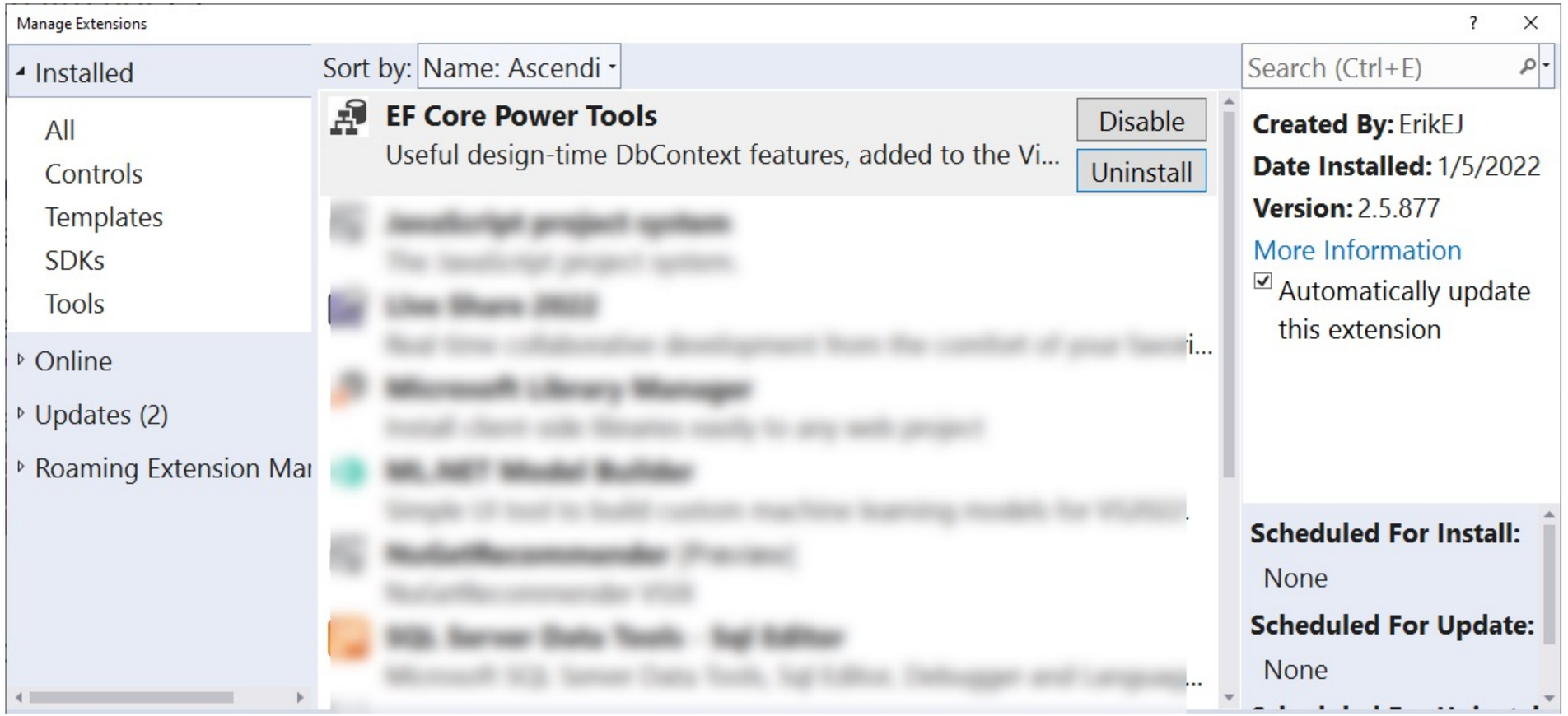
```
modelBuilder.Entity<Book>()
  .Property(b => b.Title)
  .HasColumnName("MainTitle");
```

Override with Data Annotations

Apply in entity

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

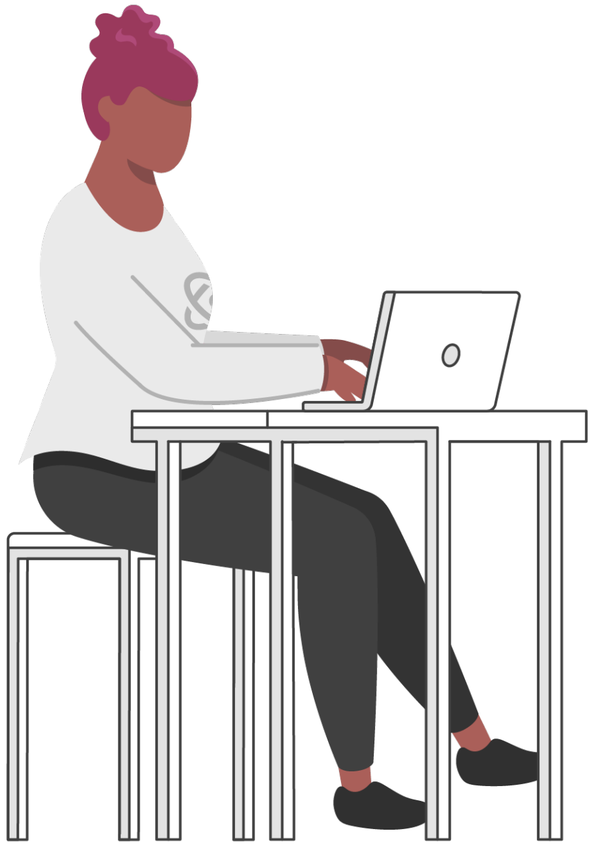
EF Core Power Tools VS Extension



And open source at: github.com/ErikEJ/EFCorePowerTools

Interpreting One-to-Many Relationships

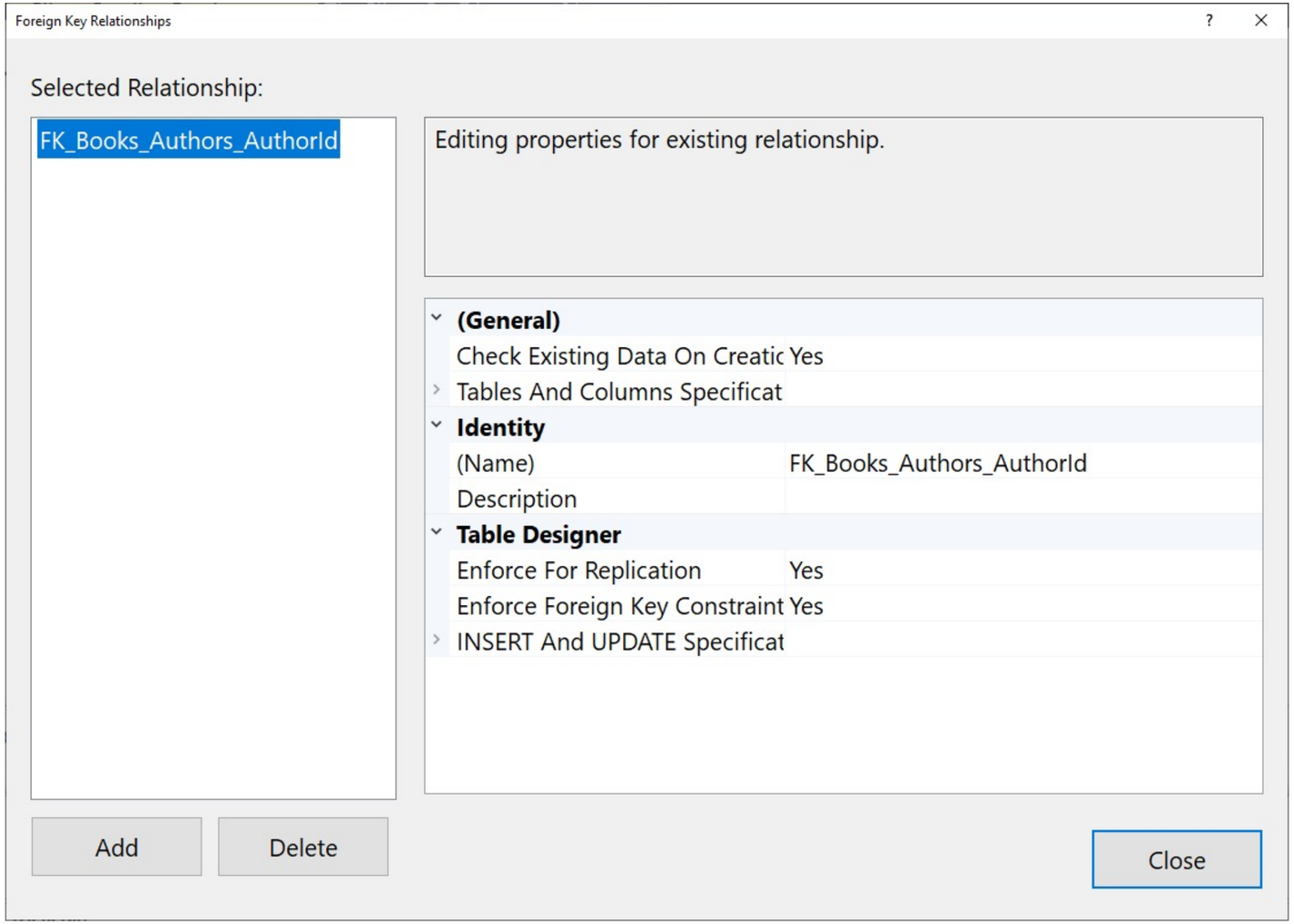
One Author Can Have Many Books



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 existing 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 a samurai object in memory to connect a quote**

```
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

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);
```

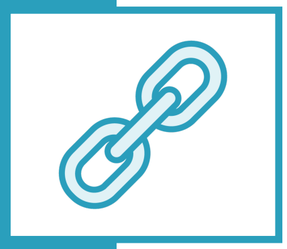
- Specify the FK property is nullable
- Map the FK property as not required
- When FK property doesn’t exist, map the inferred (“shadow”) property as not required

Summary

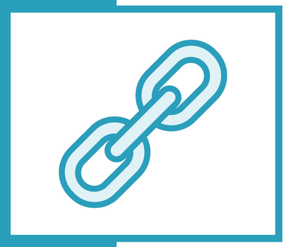
- Many ways to describe one-to-many that conventions will recognize
- You can add mappings if those patterns don't align with your desired logic
- Foreign keys are your friends
- An unconventional FK name is a common "gotcha" which you can fix in mappings
- Watch out for required principals!
- Use the EF Core Power Tools to verify how EF Core will interpret the data model

Up Next:
Logging EF Core Activity and SQL

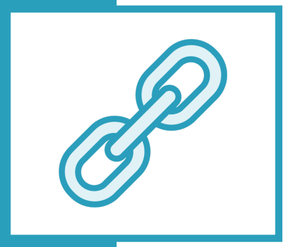
Resources



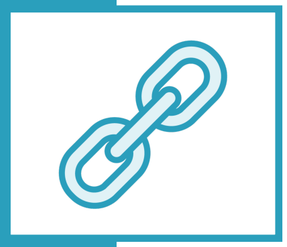
Entity Framework Core on GitHub: github.com/dotnet/efcore



EF Core Documentation: docs.microsoft.com/ef/core



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



EF Core 6.0: Fulfilling the Bucket List: codemag.com/Article/20100412