

# Defining and Using One-to-One Relationships



**Julie Lerman**

EF Core Expert and Software Coach

@julielerman | thedatafarm.com

# Module Overview



**Understand how EF Core sees 1:1**

**Add 1:1 between book and cover**

**Migrate and fix seed data in database**

**Explore querying**

**Traverse our bigger picture graph**

**Adding, updating, deleting**

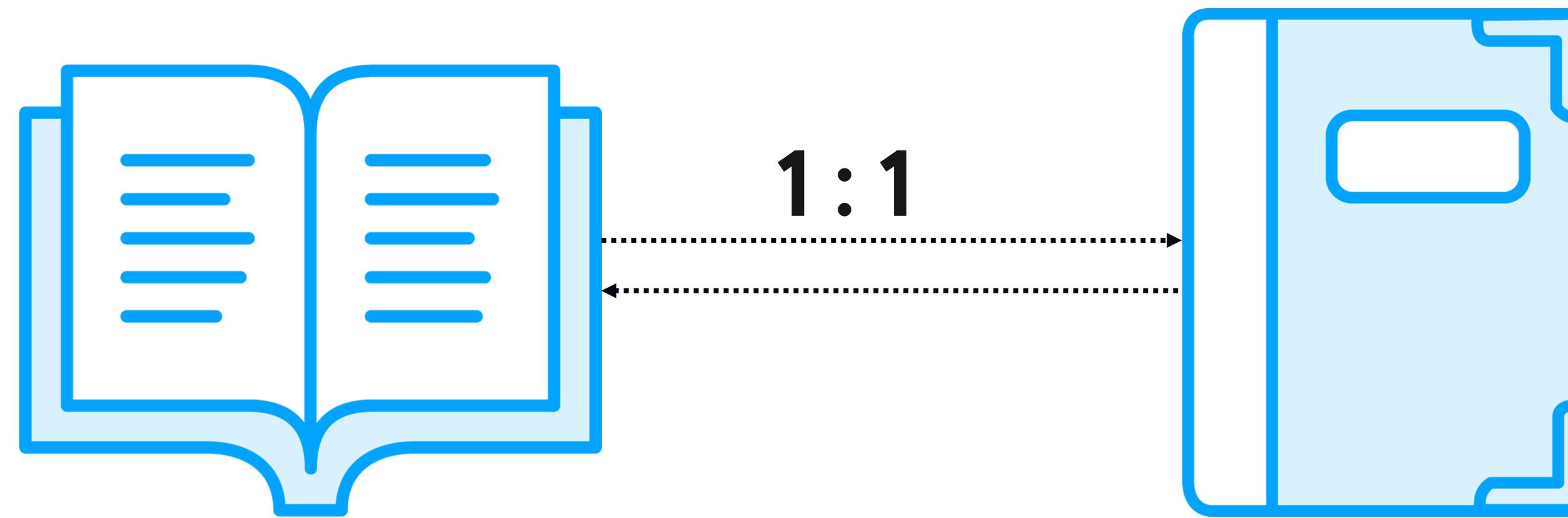
**Understand possible side effects of default FK constraints**



# **Understanding How EF Core Discovers One-to-One Relationships**



# Books Have Covers



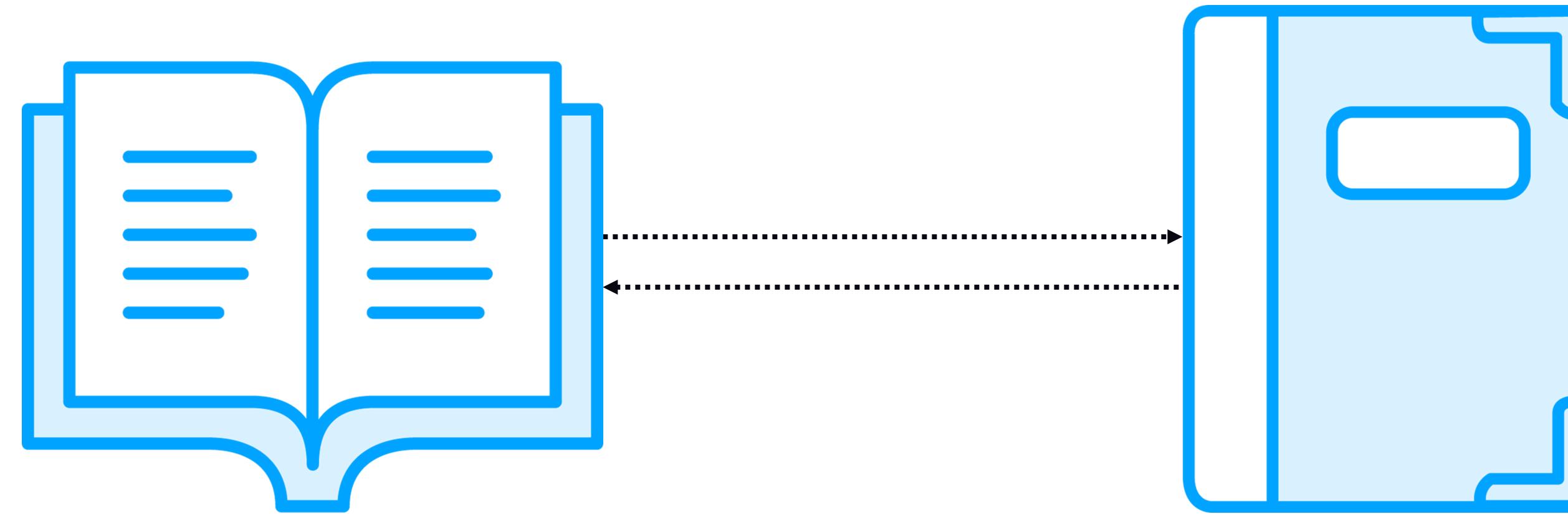
We Need to Bind Them



**DbContext must be able to  
identify a principal  
("parent") and a dependent  
("child")**



# Which is the Principal?



# Which is the Principal?



**For human resources, the desk is an attribute of the employee.**



**For the inventory keepers, the employee is an attribute of the desk.**



**EF Core can correctly identify some one-to-one mappings. But this has not always been the case.**



# Common Ways EF Core Identifies One-to-One

Navigations on both ends  
with FK in the dependent

Navigation on one end,  
FK on the other

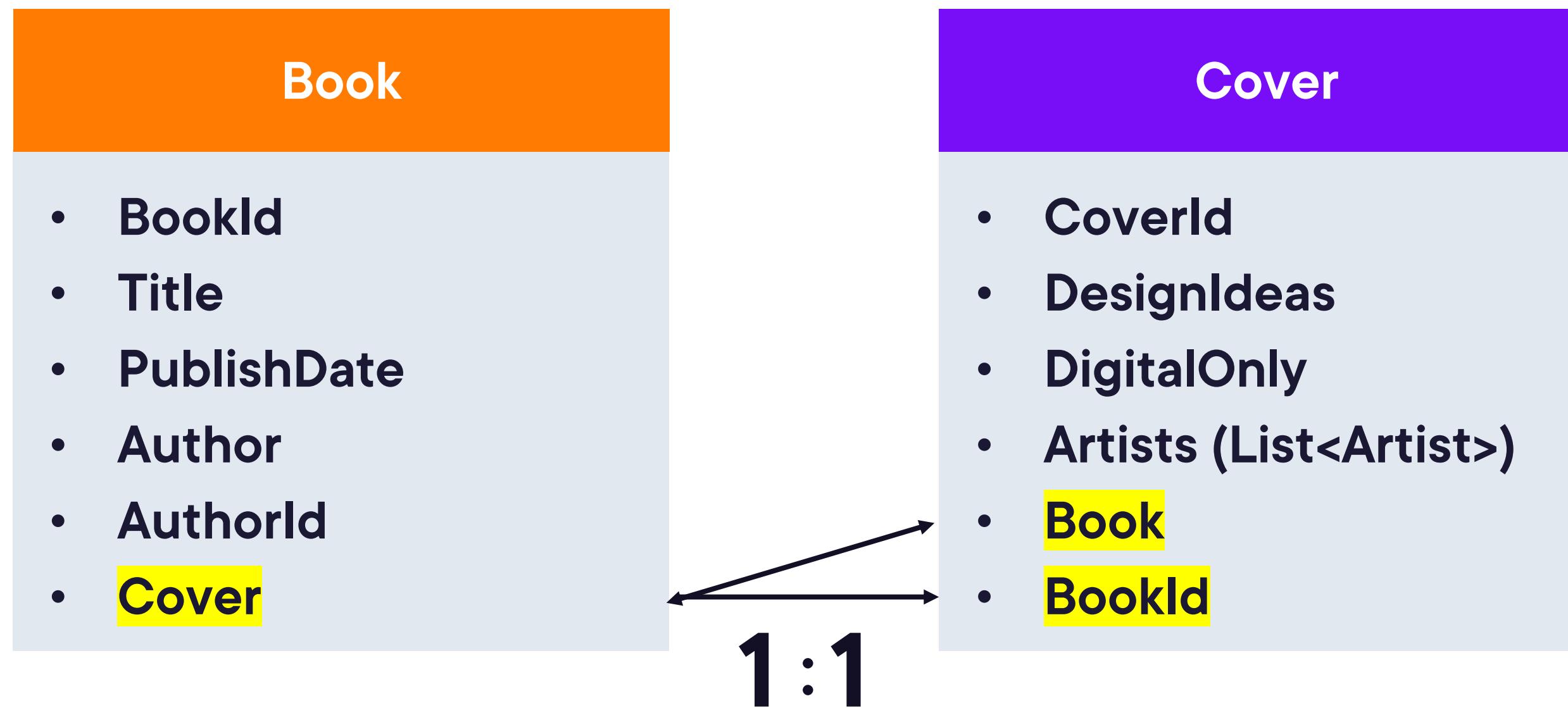
Navigations  
on both ends

EF Core will  
recognize one-to-one  
and identify the  
dependent

EF Core requires a  
mapping to define  
principal/dependent



# Tying Book and Cover Together



**Defining one-to-one with  
existing data can cause  
conflicts with DB  
constraints**



# Dependents Are Optional by Default

**By default, the cover is optional**

The database constraint will allow books to be inserted without a cover.

Your business logic allows a book's cover to be created at a later date

**Configure it to be required**

Map the cover property as `IsRequired`  
This causes the database constraint to require a cover

EF Core will not enforce the rule; the database will throw an error

*This is a business rule to be applied in your business logic.*



# Principals Are Required By Default

A cover can't exist  
without a book

Your business logic is responsible  
for following  
this rule



# **Updating the Model and Database with the New Relationship**



If you are following along in code, please *DO NOT* add this migration. I will be removing it shortly!



We're stuck with this  
because it's a demo.  
In a real app you would  
probably introduce the new  
relationship as optional.



# A Cavalcade of Cascade Deletes



Author ..... Books ..... Covers ..... Relationships to Artist



You can edit migrations files  
that have not yet been  
applied to the database.



# Order of Operations for the Migration

- 1 Adds new BookId column
- 2 Updates the BookId column values
- 3 Applies Index
- 4 Adds foreign key constraint



**Remember that this change  
to existing data was easy  
because there was minimal  
data.**



# Querying One-to-One Relationships



# Same Query Patterns for One-to-One

## Eager Loading

**Include related objects in query**

## Query Projections

**Define the shape of query results**

## Lazy Loading

**On-the-fly retrieval of data  
related to objects in memory**

## Explicit Loading

**Explicitly request related data for  
objects in memory**



```
_context.Books.Include(b => b.Cover)  
.ToList();
```

◀ Get all books with their covers even if there is no cover

```
_context.Books.Include(b => b.Cover)  
.Where(b=>b.Cover!=null)  
.ToList();
```

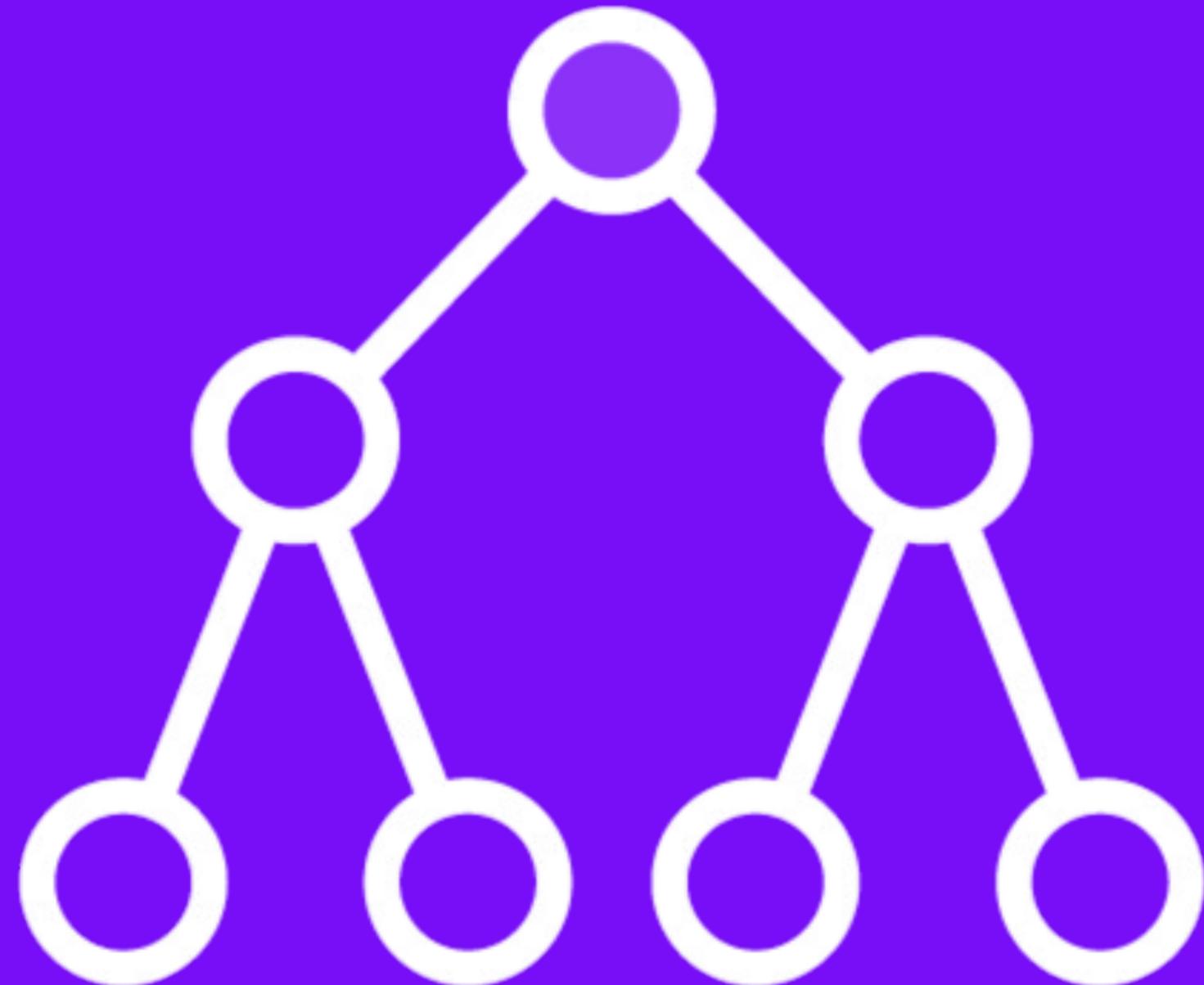
◀ Get books that have a cover (Cover != null)  
Include the cover.

```
_context.Books.Where(b=>b.Cover==null)  
.ToList();
```

◀ Get books who don't have a cover yet.  
(Cover == null)

```
_context.Books.Where(b=>b.Cover!=null)  
.Select(b=>new {b.Title,  
                 b.Cover.DesignIdeas })  
.ToList();
```

◀ Project an anonymous type of Title and DesignIdeas for books who have a cover  
(Cover != Null)  
*Notice that the Select is after the Where.  
Where returns books, select doesn't.*



## Multi-Level Query

Using multiple includes and  
*ThenInclude* to query more deeply  
into a graph



```
_context.Covers.Include(c=>c.Book)
    .ThenInclude(b=>b.Author)
    .ToList();
```

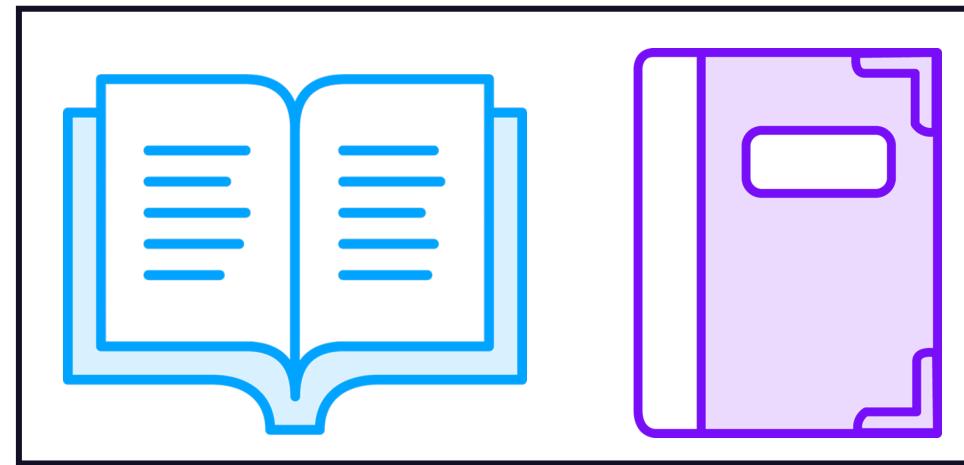
**You Can Always Query from the Dependent**

# Combining Objects in One-to-One Relationships

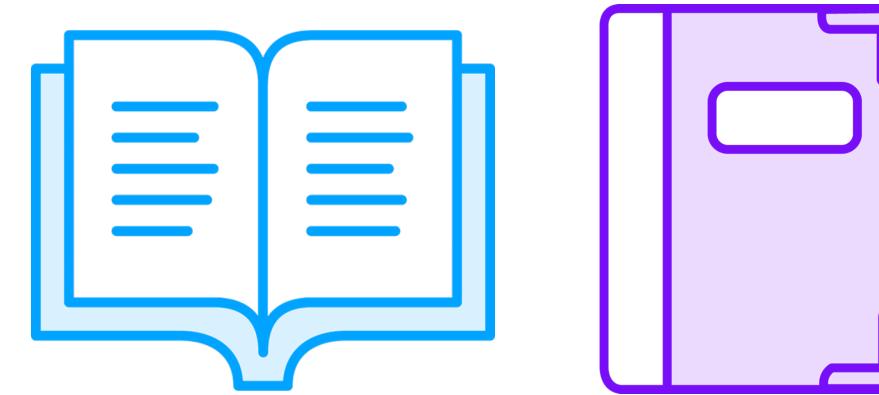


# What Is Your Path to Adding One-to-One?

**NEW**

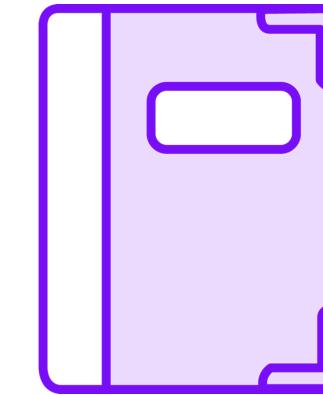


Add new book and  
cover together



Add cover to existing  
book that's in memory

BookId=1



Add cover to existing  
book that is in DB but  
not in memory





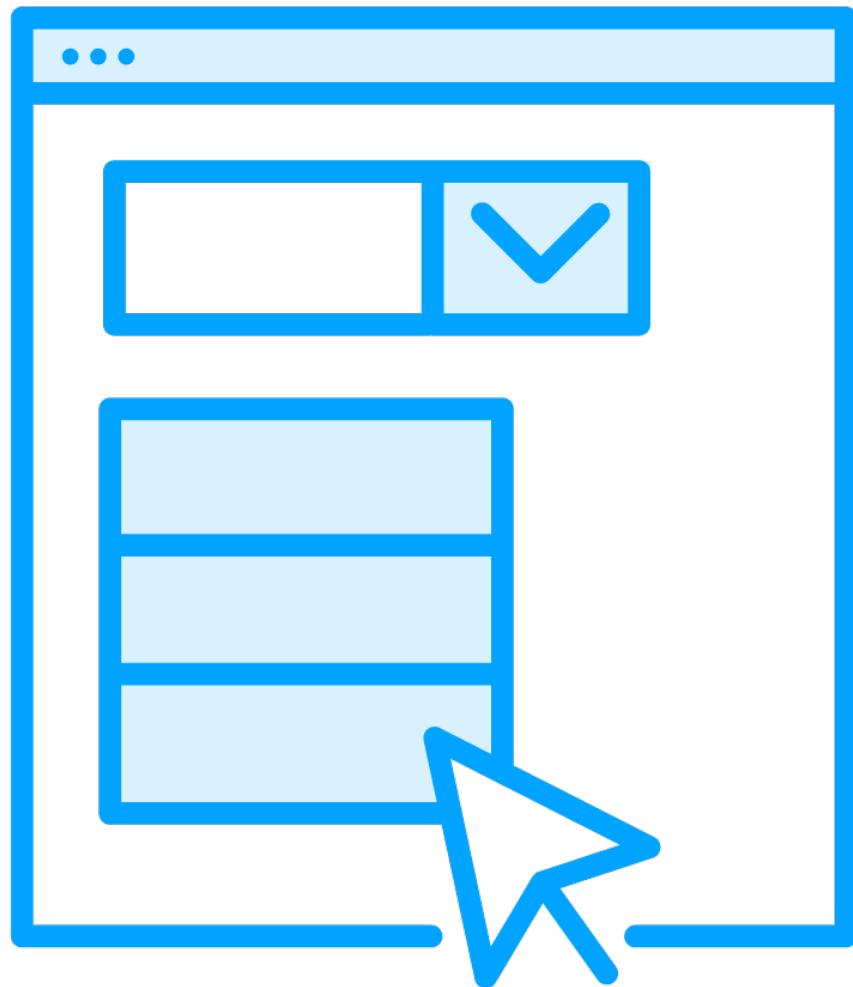
# **Unique FK Constraint**

Watch out for accidentally deleting dependents or causing an error in the database.



**Write code that is smart  
enough to protect you from  
expected behaviors**

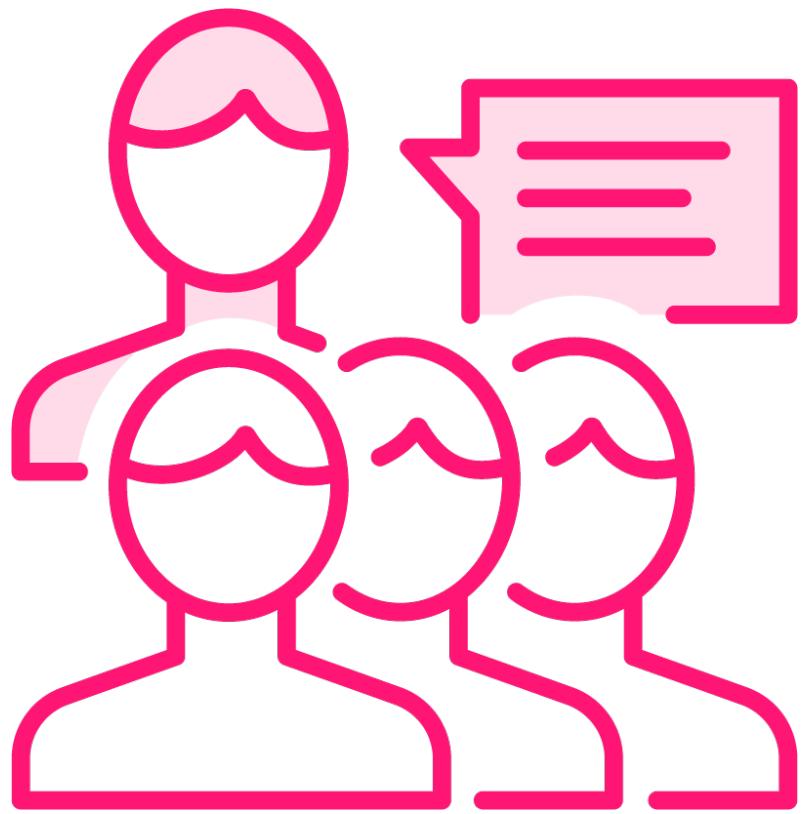




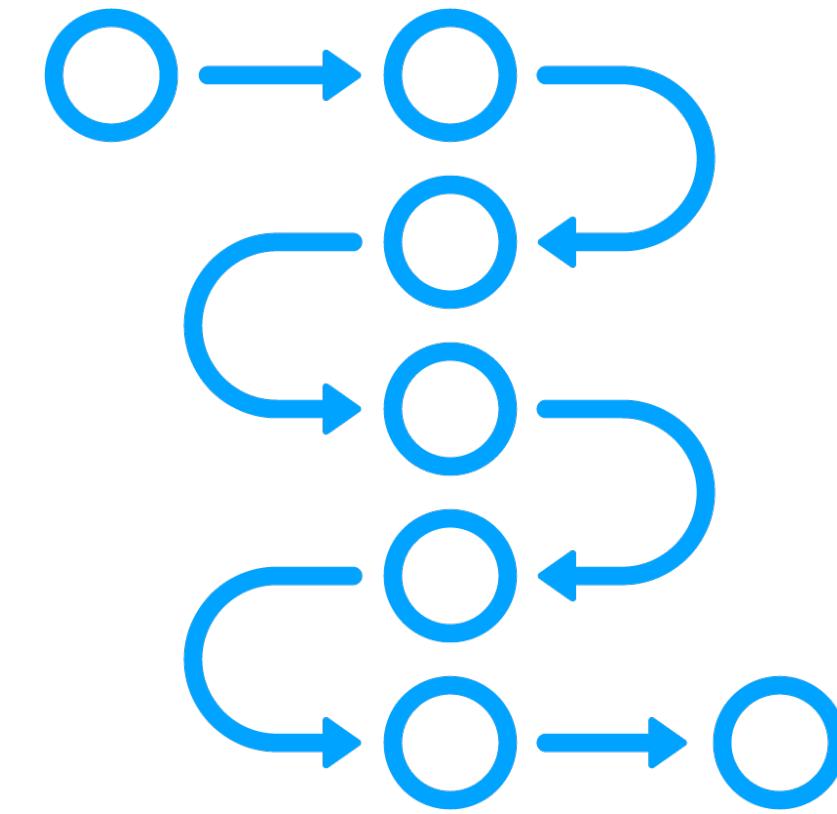
**Inform the user of the conflict  
Present them with options to resolve it**



# You're Here to Learn About EF Core



**Architectural Decisions for Application**

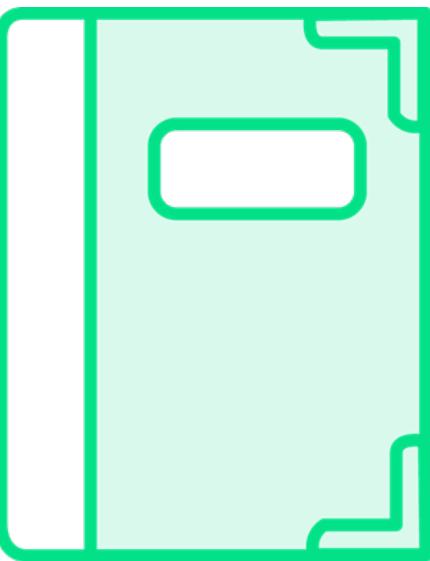


**Understand EF Core's Behavior**

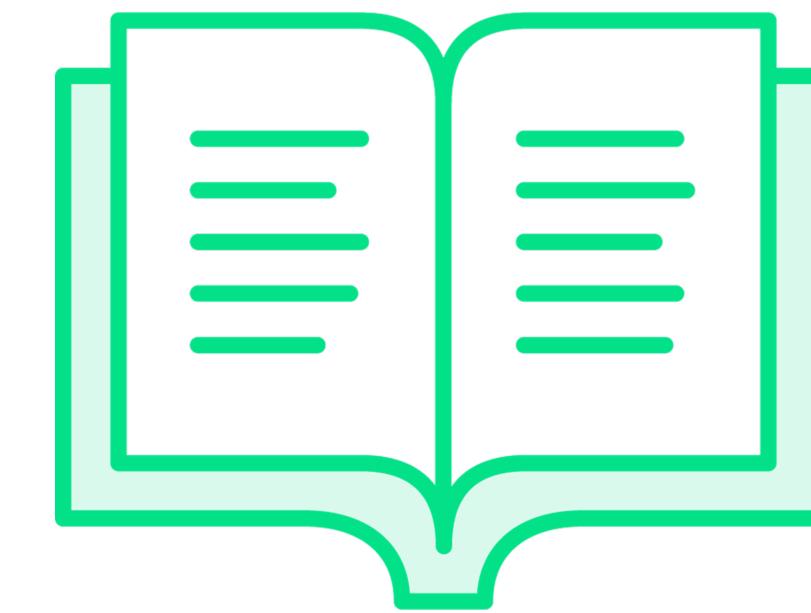


# Replacing or Removing One-to-One Relationships

# Reassigning a Cover to a Different Book



Cover originally  
assigned to a blue book



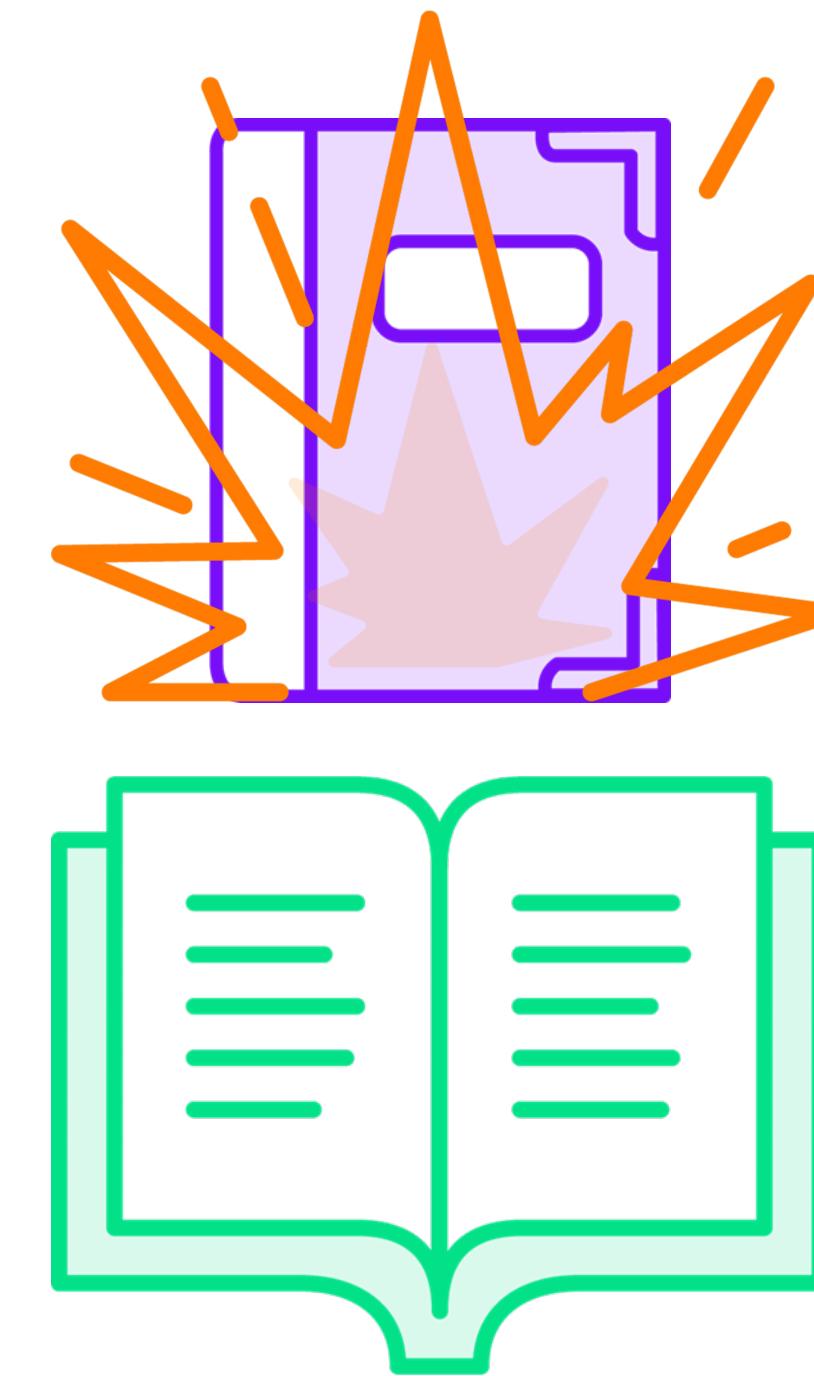
Reassigning it  
to a green book



# Foreign Key Conflicts!



Cover originally  
assigned to a blue book



Green book already has  
a purple cover



```
greenCover.BookId=3
```

```
greenCover.Book=greenBookObject
```

```
greenCover=blueBook.Cover;  
greenBook.Cover= greenCover ;
```

---

```
UPDATE covers SET bookid=3  
WHERE coverid=5
```

◀ Change the foreign key value

◀ Change the navigation property in the dependent object

◀ Change the navigation property of the principal

◀ Resulting SQL on SaveChanges for all three variations

# Foreign Key Conflict Outcomes

```
greenCover=blueBook.Cover;  
greenBookWithPurpleCover.Cover=  
    greenCover ;
```

```
greenCover.BookId=3;
```

- ◀ Blue Book with Green Cover is tracked, Green Book with Purple Cover is tracked
  - ◀ EF Core will DELETE Purple Cover and change green cover's BookId
- 
- ◀ Change tracker is only aware of Green Cover In the database, purple cover has BookId=3
  - ◀ EF Core sends an Update command to change the cover's bookId to 3
  - ◀ Database returns an error: two covers cannot have the same bookId

# Removing One-to-One Deletes the Dependent!

```
_context.Covers.Remove(greenCover);
```

- ◀ Just delete the row in your code
- ◀ EF Core sends a DELETE to the database for that cover row

```
bookWithCoverGraph.Cover=null;
```

- ◀ Change tracker is aware of book and cover
- ◀ EF Core sends a DELETE to the database for that cover row

**There are even more paths  
to affecting the  
relationships**



# Some Factors that Affect Relationship Behavior



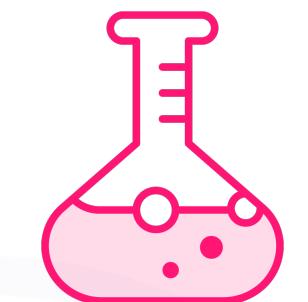
**Is the dependent required?**



**Is the child object in memory?**



**Are parent or child object being tracked?**



**Test the behaviors to learn cause and effect!**



## Review



**EF Core needs to be able to ID principal and child**

**Simplest (our path) is navigation props in both ends with FK in dependent**

**Default: child is optional, parent is required, cascade delete**

**Same query patterns as other relationships**

**Watch out for FK constraints when joining or changing related ends**

**Know the behaviors and design your app to protect you from side effects**



**Up Next:**

# **Working with Views and Stored Procedures and Raw SQL**

---



# Resources

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

EF Core Relationship Documentation [bit.ly/EFCoreRelation](https://bit.ly/EFCoreRelation)

EF Core Power Tools on GitHub [github.com/ErikEJ/EFCorePowerTools/wiki](https://github.com/ErikEJ/EFCorePowerTools/wiki)

C# Conditional Operator Docs [bit.ly/3s4nSql](https://bit.ly/3s4nSql)

