Microsoft

Entity Framework - II

SQL – Cascade Deletes

SQL – Cascade Deletes

Cascade deletes in SQL are a **referential integrity feature** that automatically removes related rows in child tables when a row in the parent table is deleted.

This behavior is controlled by the **ON DELETE CASCADE** option in a foreign key constraint.

Cascading deletes *can* impact performance if many child rows exist.

Cascade Deletes

```
CREATE TABLE Parent (
    ParentID INT PRIMARY KEY,
    Name NVARCHAR(50)
);

CREATE TABLE Child (
    ChildID INT PRIMARY KEY,
    ParentID INT,
    FOREIGN KEY (ParentID) REFERENCES Parent(ParentID)
         ON DELETE CASCADE
);

DELETE FROM Parent WHERE ParentID = 1;
```

EF Relationships

Relationships

```
public class Blog
{
    public string Name { get; set; }
    public virtual Uri SiteUri { get; set; }
}
public class Post
{
    public string Title { get; set; }
    public string Content { get; set; }
    public DateTime PublishedOn { get; set; }
    public bool Archived { get; set; }
}
```

Foreign Key Relationship

```
public class Post
{
    public string Title { get; set; }
    public string Content { get; set; }
    public DateOnly PublishedOn { get; set; }
    public bool Archived { get; set; }

    public Blog Blog { get; set; }
}
```

One-to-One Relationship

```
public class Blog  // Principal (parent)
{
    public int Id { get; set; }
    public BlogHeader? Header { get; set; } // Ref nav to dep
}

public class BlogHeader // Dependent (child)
{
    public int Id { get; set; }
    public int BlogId { get; set; } // Req foreign key property
    public Blog Blog { get; set; } = null!; // Req ref nav to principal
}
```

https://learn.microsoft.com/en-us/ef/core/modeling/relationships/one-to-one

= null!

This uses the **null-forgiving operator** (!) introduced in C# 8. It tells the compiler: *"I know this is non-nullable, but trust me—it will be initialized later."*

Without this, the compiler would warn: Non-nullable property 'Blog' must contain a non-null value when exiting constructor because the property is not assigned in the constructor.

In EF Core, navigation properties like Blog are often populated by the ORM after object creation, not in the constructor.
So developers use null! to suppress the warning and indicate: *"EF will set this later."*

One to Many Relationships

```
public class Blog
{
    public string Name { get; set; }
    public virtual Uri SiteUri { get; set; }

    public ICollection<Post> Posts { get; }
}
```

In EF Core, the Blog.Posts and Post.Blog properties are called "navigations".

Class → Relationships in RDBMS

```
CREATE TABLE [Blogs] (
    [Id] int NOT NULL IDENTITY,
    [Name] nvarchar(max) NULL,
    [SiteUri] nvarchar(max) NULL,
    CONSTRAINT [PK_Blogs] PRIMARY KEY ([Id]));
```

Class → Relationships in RDBMS

```
CREATE TABLE [Posts] (
    [Id] int NOT NULL IDENTITY,
    [Title] nvarchar(max) NULL,
    [Content] nvarchar(max) NULL,
    [PublishedOn] datetime2 NOT NULL,
    [Archived] bit NOT NULL,
    [BlogId] int NOT NULL,
    CONSTRAINT [PK_Posts] PRIMARY KEY ([Id]),
    CONSTRAINT [FK_Posts_Blogs_BlogId] FOREIGN KEY
 ([BlogId]) REFERENCES [Blogs] ([Id]) ON DELETE CASCADE);
```

Many-to-Many Relationship

```
CREATE TABLE "Posts" (
    "Id" INTEGER NOT NULL CONSTRAINT "PK_Posts" PRIMARY KEY AUTOINCREMENT);

CREATE TABLE "Tags" (
    "Id" INTEGER NOT NULL CONSTRAINT "PK_Tags" PRIMARY KEY AUTOINCREMENT);

CREATE TABLE "PostTag" (
    "PostsId" INTEGER NOT NULL,
    "TagsId" INTEGER NOT NULL,
    CONSTRAINT "PK_PostTag" PRIMARY KEY ("PostsId", "TagsId"),
    CONSTRAINT "FK_PostTag_Posts_PostsId" FOREIGN KEY ("PostsId")
  REFERENCES "Posts" ("Id") ON DELETE CASCADE,
    CONSTRAINT "FK_PostTag_Tags_TagsId" FOREIGN KEY ("TagsId") REFERENCES
  "Tags" ("Id") ON DELETE CASCADE);
```

https://learn.microsoft.com/en-us/ef/core/modeling/relationships/many-to-many

EF – Initial Implementation

```
public class Post
{
  public int Id { get; set; }
  public List<PostTag>
     PostTags { get; } = [];
}

public class Tag
{
   public int Id { get; set; }
   public List<PostTag>
      PostTags { get; } = [];
}
```

```
public class PostTag
{
public int PostsId { get; set; }
public int TagsId { get; set; }
public Post Post { get; set; } = nul
public Tag Tag { get; set; } = null!
}
```

EF – Many to Many Relationship

```
public class Post
{
    public int Id { get; set; }
    public List<Tag> Tags { get; } = [];
}

public class Tag
{
    public int Id { get; set; }
    public List<Post> Posts { get; } = [];
}
```

EF can manage the join entity transparently, without a .NET class defined for it, and without navigations for the two one-to-many relationships.

Indeed, EF model building conventions will, by default, map the Post and Tag types shown here to the three tables in the database schema at the top of this section. This mapping, without explicit use of the join type, is what is typically meant by the term "many-to-many".

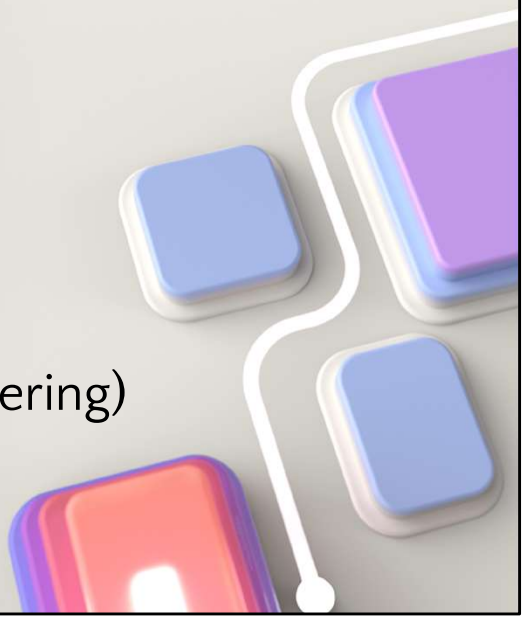
EF – Many to Many – Explicit Creation

An equivalent explicit configuration for the previous **Post←→ Tag** *relationship is shown below as a learning tool:*

```
protected override void
 OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Post>()
        .HasMany(e ⇒ e.Tags)
        .WithMany(e ⇒ e.Posts);
}
```

https://learn.microsoft.com/en-us/ef/core/modeling/relationships/many-to-many

Scaffolding (DB Reverse Engineering)



Reverse engineering is the process of scaffolding entity type classes and a DbContext class based on a database schema.

It can be performed using:

- The `Scaffold-DbContext` command of the EF Core Package Manager Console (PMC) tools (**VS Only**), or
- The `dotnet ef dbcontext scaffold` command of the .NET Command-line Interface (CLI) tools (**All Platforms**).

EF Scaffolding Command

```
dotnet ef dbcontext scaffold "Data
 Source=localhost;Database=AdventureWorksLT20
 22; User ID=sa;
 password=YourStrongPassword123;
 TrustServerCertificate=True"
 Microsoft.EntityFrameworkCore.SqlServer
```

EF Scaffolding – Specifying tables and views

- By default, all tables and views in the database schema are scaffolded into entity types.
  - You can limit which tables and views are scaffolded by specifying schemas and tables.
- The --schema (.NET CLI) argument specifies the schemas of tables and views for which entity types will be generated.
- The --table (.NET CLI) argument specified the tables and views for which entity types will be generated.
  - Tables or views in a specific schema can be included using the 'schema.table' or 'schema.view' format.

EF Scaffolding - Examples

```
dotnet ef dbcontext scaffold ... --table Artist --
 table Album
```

```
dotnet ef dbcontext scaffold ... --schema Customer
 --schema Contractor
```

```
dotnet ef dbcontext scaffold ... --table
 Customer.Purchases --table Contractor.Accounts --
 table Contractor.Contracts
```

https://learn.microsoft.com/en-us/ef/core/managing-schemas/scaffolding/?tabs=dotnet-core-cli

Managing DB Changes

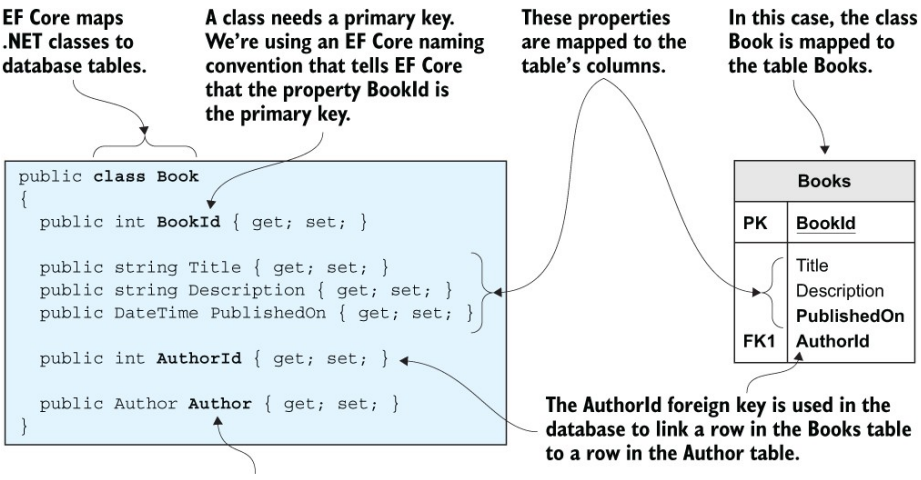
- Scaffold once only
  - Switch to using EF Core database migrations
  - Manually update the entity types and EF configuration when the database changes
- **Repeated scaffolding**
  - re-scaffold every time the database changes*
  - **Utilize Partial Classes!**
    - **All generated EF code is comprised of partial classes.**

*By default, the EF commands will not overwrite any existing code to protect against accidental code loss. The -Force (Visual Studio PMC) or --force (.NET CLI) argument can be used to force overwriting of existing files.

https://learn.microsoft.com/en-us/ef/core/managing-schemas/scaffolding/?tabs=dotnet-core-cli

Appendix

EF Core Mapping



The application's DbContext

You must have a class that inherits from the EF Core class DbContext. This class holds the information and configuration for accessing your database.

```
public class AppDbContext : DbContext
{
    private const string ConnectionString =
      @"Server=(localdb)\mssqllocaldb;
        Database=MyFirstEfCoreDb;
        Trusted_Connection=True";

    protected override void OnConfiguring(
        DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder
            .UseSqlServer(connectionString);
    }

    public DbSet<Book> Books { get; set; }
}
```

The database connection string holds information about the database:

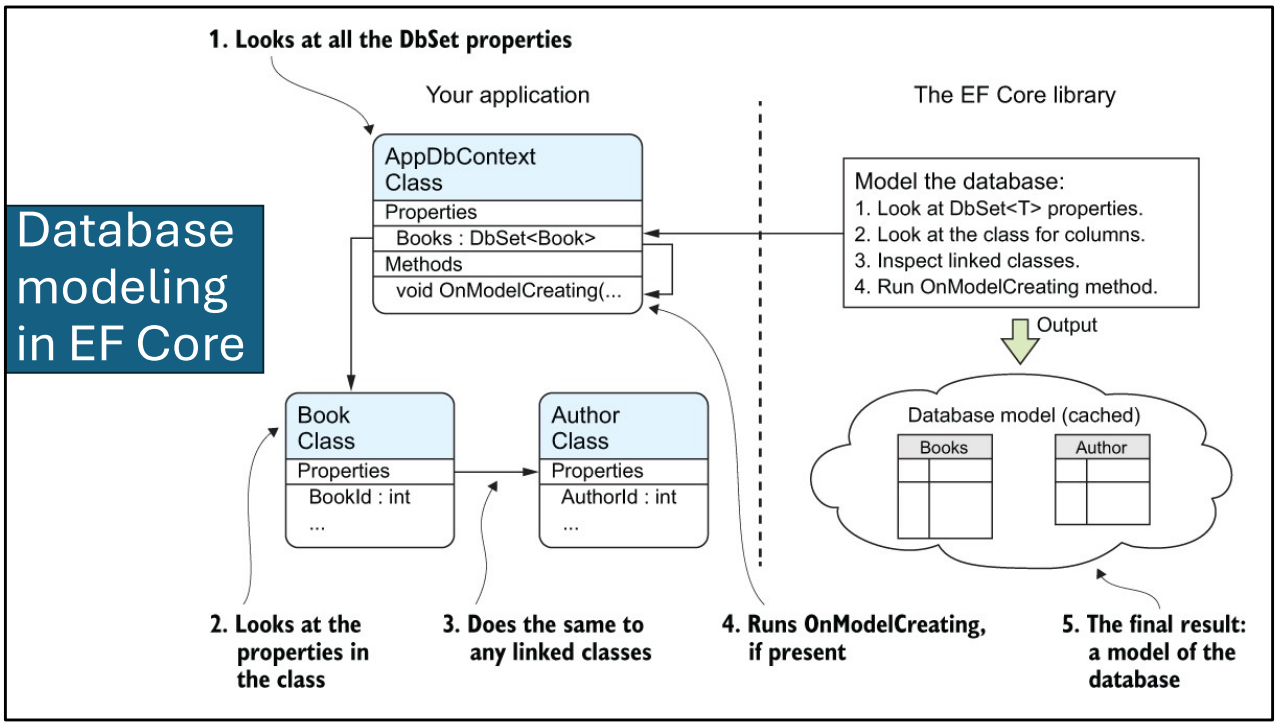
- **How to find the database server**
- **The name of the database**
- **Authorization to access the database**

In a console application, you configure EF Core's database options by overriding the OnConfiguring method. In this case, you tell it you're using an SQL Server database by using the UseSqlServer method.

By creating a property called Books of type DbSet<Book>, you tell EF Core that there's a database table named Books, and it has the columns and keys as found in the Book class.

Our database has a table called Author, but you purposely didn't create a property for that table. EF Core finds that table by finding a navigational property of type Author in the Book class.

Database modeling in EF Core



Reading data using EF Core

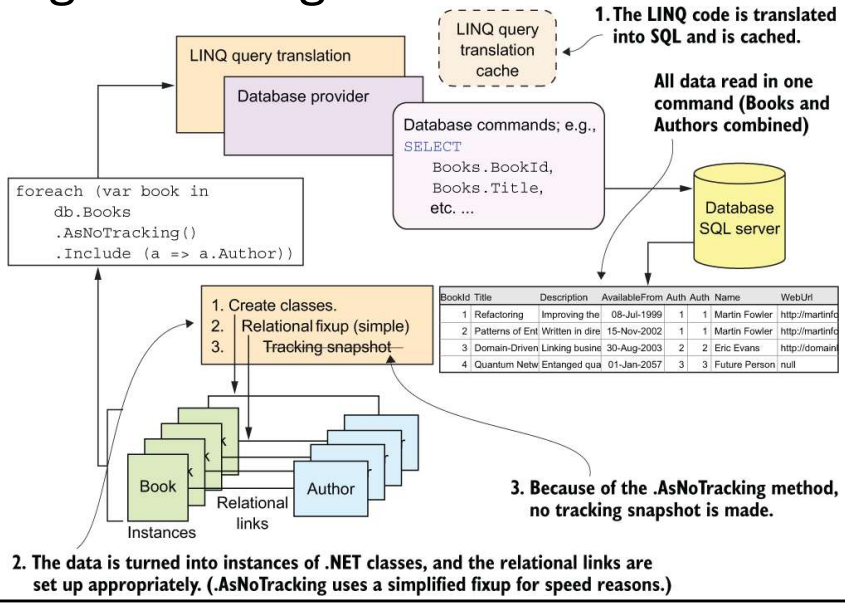
```
public static void ListAll()
{
    using (var db = new AppDbContext())          ❶
    {
        foreach (var book in
            db.Books.AsNoTracking()              ❷

            .Include(book => book.Author))       ❸
        {
            var webUrl = book.Author.WebUrl == null
                ? "- no web URL given -"
                : book.Author.WebUrl;
            Console.WriteLine(
                $"{book.Title} by {book.Author.Name}");
            Console.WriteLine("     " +
                "Published on " +
                $"{book.PublishedOn:dd-MMM-yyyy}" +
                $". {webUrl}");
        }
    }
}
```

Reading data using EF Core

```
SELECT [b].[BookId],
[b].[AuthorId],
[b].[Description],
[b].[PublishedOn],
[b].[Title],
[a].[AuthorId],
[a].[Name],
[a].[WebUrl]
FROM [Books] AS [b]
INNER JOIN [Author] AS [a] ON
[b].[AuthorId] = [a].[AuthorId]
```

Reading data using EF Core



Updating data using EF Core

```
public static void ChangeWebUrl()
{
    Console.Write("New Quantum Networking WebUrl > ");
    var newWebUrl = Console.ReadLine();                                  ❶

    using (var db = new AppDbContext())
    {
        var singleBook = db.Books
            .Include(book => book.Author)                                ❷
            .Single(book => book.Title == "Quantum Networking");         ❸

        singleBook.Author.WebUrl = newWebUrl;                            ❹
        db.SaveChanges();                                                ❺
        Console.WriteLine("... SavedChanges called.");
    }

    ListAll();                                                           ❻
}
```

Updating data using EF Core