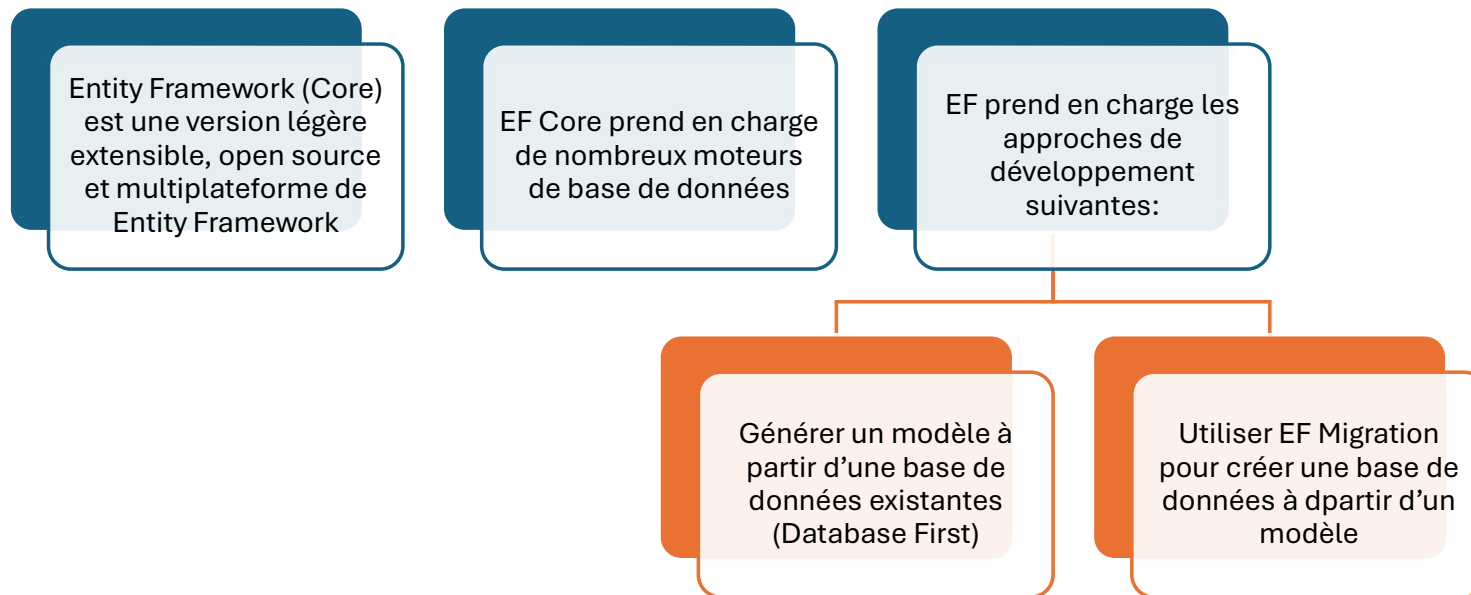


Entity Framework .NET

Entity Framework Core(EF Core)





Accès aux données avec Entity Framework .NET

- .NET propose deux niveaux d'abstractions pour l'accès aux bases de données relationnelles
 - Entity Framework / Entity Framework Core
 - ADO .NET (nécessite une bonne connaissance de SQL)
 - ADO .NET = ActiveX Data Objects for .NET
 - Pilote l'exécution de requêtes SQL en base de données
 - SQL est normalisé mais il existe des différences entre dialectes SQL des différents SGBDr
-



Accès aux données avec Entity Framework

- ORM (Object Relational Mapping)
 - Logiciel faisant correspondre des objets à des enregistrements et des tables en base de données
 - Python : Django ou SQLAlchemy
 - PHP: Doctrine
 - Java : Hibernate
 - C# .NET : Entity Framework
 - Un ORM propose
 - Une abstraction par rapport à la base de données et de son dialecte SQL
 - Peut gérer un cache mémoire pour limiter les requêtes en base
-



Monde objet // Monde bdd relationnelle

- Les types utilisés dans le monde objet n'ont pas tous un équivalent dans le monde relationnel
 - Un objet peut être constitué d'attributs de type collection
 - Il faut être vigilant aux problématiques d'héritage (pas de mécanisme équivalent dans une bdd relationnelle)
 - => solutions de mapping objet/relationnel tel qu'Entity Framework
-



Entity Framework / Entity Framework Core

- .NET Framework historique ne fonctionne que sur Windows
 - .NET Core fonctionne sur tous les OS classiques
 - Entity Framework compatible avec .NET Framework
 - Entity Framework Core : compatible avec .NET Core
 - Réécriture/simplification de l'ancienne version
 - Rupture de compatibilité avec Entity Framework
 - Il faut donc installer la bonne version de l'ORM en fonction de la version .NET utilisée
-



Installation d'EF Core

- EF Core n'est pas présent nativement dans .NET Core
 - Besoin de télécharger l'extension en tant que paquet NuGet
 - NuGet est un gestionnaire de paquet sur lequel vous pouvez trouver de très nombreuses librairies complémentaires
 - L'ajout de paquet se fait en cliquant sur le projet et en sélectionnant « gérer les packages NuGet »
 - Sélectionner Microsoft.EntityFrameworkCore et l'installer
 - Idem avec Microsoft.EntityFrameworkCore.SqlServer
 - On peut vérifier après install que les dépendances sont bien présentes dans le projet
-

NuGet

Gestionnaire de
paquet NuGet: outil
utilisé dans Visual
Studio pour gérer les
dépendances d'un
projet
(bibliothèques,
packages externes)

Gestionnaire de
package officiel pour
projets .NET

Les packages
contiennent des bibli
dll, des fichiers de
configuration, des
scripts de code...

Utilisable en ligne de
commandes ou sous
sa forme d'interface



Installation d'EFCore

- On peut aussi installer deux autres paquets NuGet
 - Microsoft.EntityFrameworkCore.Design : propose des outils de conception intégrés à Visual Studio
 - Microsoft.EntityFrameworkCore.Tools : propose des outils complémentaires accessibles en mode ligne de commandes
-

EF Core - Utilisation

- Creation d'une classe Article

```
0 références
public class Article
{
    0 références
    public Article() : this("", "", 0)
    {
    }

    3 références
    public Article(string description, string brand, double price = 0)
    {
        this.Description = description;
        this.Brand = brand;
        this.Price = price;
    }

    0 références
    public int ArticleId { get; set; } = 0;
    1 référence
    public string Description { get; set; } = "";
    1 référence
    public string Brand { get; set; } = "";
    1 référence
    public double Price { get; set; } = 0;
}
```



DbContext et DbSet

Concepts fondamentaux de EF
Core permettant de manipuler une
bdd relationnelle en utilisation des
objets C#

DbContext: représente une
session avec la base de données.
Communique avec la base,
récupère les données, enregistre
les modifications et effectue des
transactions. C'est une passerelle
entre le code C# et la base de
données

Fonctionnalités de DbContext

- Gestion des entités mappées aux tables de la base de données
 - Gestion des changements: on fait persister dans la base de données avec la méthode `SaveChanges()`
 - Exécution des requêtes LINQ pour interagir avec la base
 - Configuration de la base de données
-
- `OnConfiguring` permet de spécifier la chaîne de connexion à la base de données
 - `OnModelCreating` est utilisée pour configurer le modèle comme les clés primaires et les liens entre les entités

DbSet

- Un DbSet est une collection d'entités dans la base de données. Equivalent d'une table dans la base de données.
- Utilisés pour effectuer des opérations sur un type d'entité particulier.
- Un DbSet est une représentation d'une table en C#
- On se sert du DbContext pour manipuler les DbSet
- Un DbSet fournit les méthodes Add(), Remove(), Find(), Where() pour effectuer des opérations sur une table

EF Core - Utilisation

- Il faut ensuite définir un DbContext, une classe qui sera le point d'entrée pour manipuler les entités
 - Exposition des différents DbSet (collections d'entités)
 - Elle lie l'ORM au moteur de base de données utilisé
 - Dans un premier temps, on va utiliser la base de données SQLite (pas besoin de serveur)

```
4 références
internal class AppDbContext : DbContext
{
    0 références
    public DbSet<Article> Articles { get; set; }
    3 références
    public string DbPath { get; }

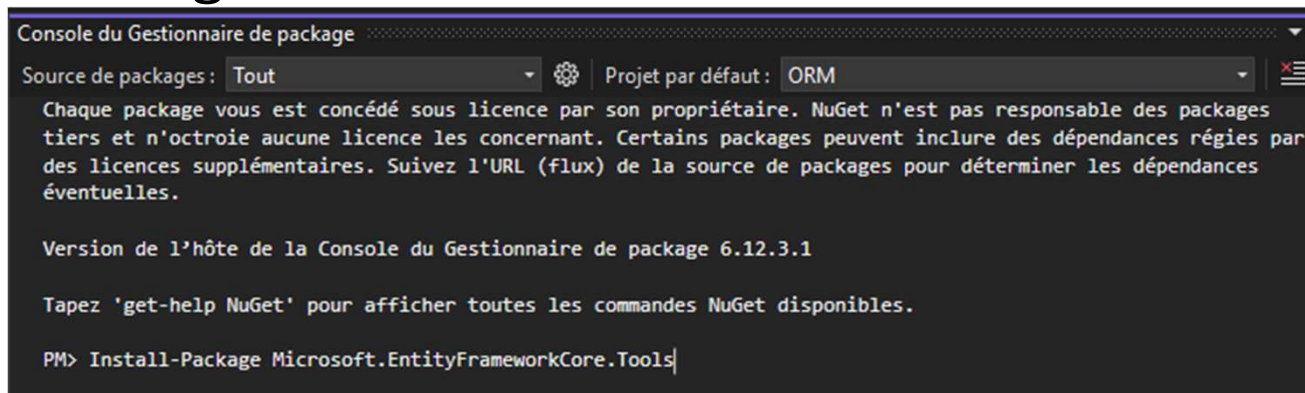
    1 référence
    public AppDbContext()
    {
        var folder = Environment.SpecialFolder.Desktop;
        var path = Environment.GetFolderPath(folder);
        DbPath = System.IO.Path.Join(path, "hello.db");

        Console.WriteLine(DbPath);
    }

    0 références
    protected override void OnConfiguring(DbContextOptionsBuilder options)
    {
        options.UseSqlite($"Data Source={DbPath}");
    }
}
```

EF Core - Utilisation

- Pour générer la base de données avec la console du « Package Manager » de VS :



The screenshot shows the 'Console du Gestionnaire de package' (Package Manager Console) in Visual Studio. At the top, there are two dropdown menus: 'Source de packages' set to 'Tout' and 'Projet par défaut' set to 'ORM'. Below these, there is a block of text in French explaining the license and dependencies of packages. Further down, it shows the version '6.12.3.1' and a prompt to type 'get-help NuGet' for more commands. The command prompt shows 'PM> Install-Package Microsoft.EntityFrameworkCore.Tools'.

```
Console du Gestionnaire de package
Source de packages : Tout
Projet par défaut : ORM

Chaque package vous est concédé sous licence par son propriétaire. NuGet n'est pas responsable des packages tiers et n'octroie aucune licence les concernant. Certains packages peuvent inclure des dépendances régies par des licences supplémentaires. Suivez l'URL (flux) de la source de packages pour déterminer les dépendances éventuelles.

Version de l'hôte de la Console du Gestionnaire de package 6.12.3.1

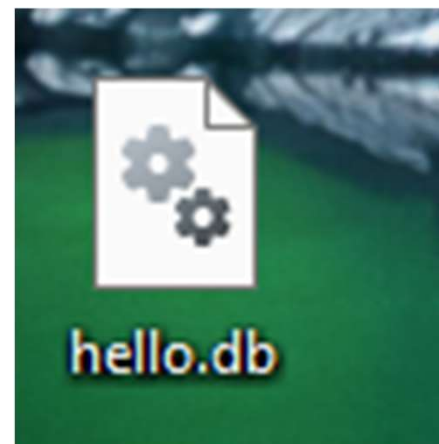
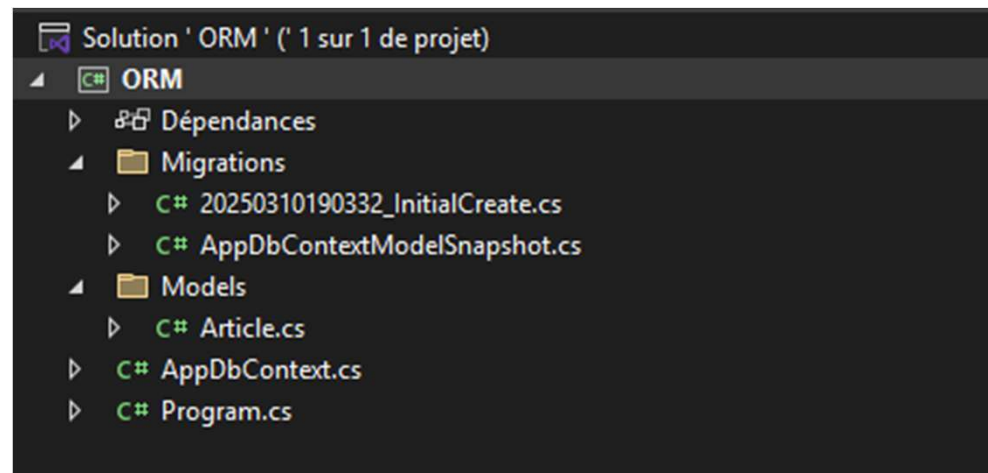
Tapez 'get-help NuGet' pour afficher toutes les commandes NuGet disponibles.

PM> Install-Package Microsoft.EntityFrameworkCore.Tools
```

```
Temps écoulé : 00:00:06.3069527
PM> Add-Migration InitialCreate
```

```
PM> Update-Database
```

EFCore - Utilisation



1

```
0 références
internal class Program
{
    0 références
    static void Main(string[] args)
    {
        using var db = new AppDbContext();

        var article1 = new Article("art1", "brand1", 100);
        var article2 = new Article("art2", "brand2", 50);
        db.Add(article1);
        db.Add(article2);
        db.SaveChanges();
        Console.WriteLine(article1);
        Console.WriteLine(article2);

        //var articles = new List<Article>() {
        //    new Article("art1", "brand1", 100),
        //    new Article("art2", "brand2", 50),
        //};
        //db.AddRange(articles);
        //db.SaveChanges();
    }
}
```

```
Fichier Modifier Affichage
#
N

P>-->@tablessqlite_sequencesqlite_sequence@CREATE TABLE sqlite_sequence(name,seq)@
@#tableArticlesArticles@CREATE TABLE "Articles" (
  "ArticleId" INTEGER NOT NULL CONSTRAINT "PK_Articles" PRIMARY KEY AUTOINCREMENT,
  "Description" TEXT NOT NULL,
  "Brand" TEXT NOT NULL,
  "Price" REAL NOT NULL
)@
@778,Otable_EFMigrationsHistory_EFMigrationsHistory@CREATE TABLE "_EFMigrationsHistory" (
  "MigrationId" TEXT NOT NULL CONSTRAINT "PK___EFMigrationsHistory" PRIMARY KEY,
  "ProductVersion" TEXT NOT NULL
)@
@778 indexsqlite_autoindex___EFMigrationsHistory_1_EFMigrationsHistory@ @ @A@ @11@,-table_EFMigrationsLock_EFMigrationsLock@CREATE
TABLE "_EFMigrationsLock" (
  "Id" INTEGER NOT NULL CONSTRAINT "PK___EFMigrationsLock" PRIMARY KEY,
  "Timestamp" TEXT NOT NULL
)
@ @

$ 02025-03-16 20:19:49.2612895+00:00
```

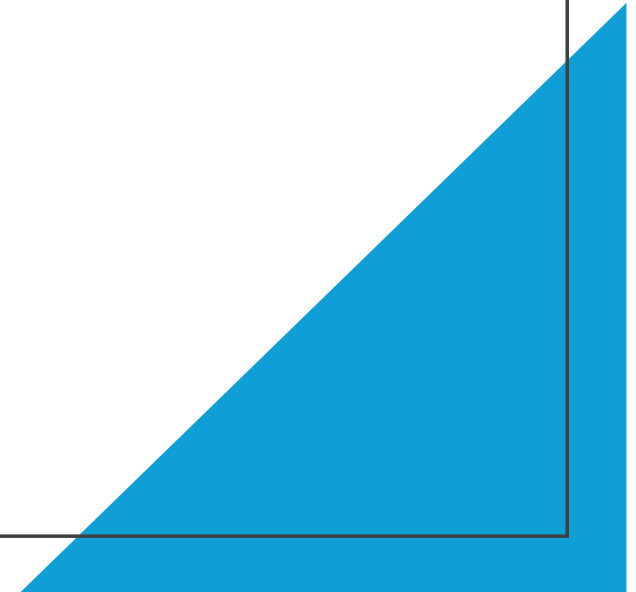
```
// retrouver une entité par sa clé primaire  
var article = db.Articles.Single(art => art.ArticleId == 2);  
  
// on peut retrouver plusieurs entités qui correspondent à un critère donnée  
var articles = db.Articles.Where(art => art.Price < 150000);  
foreach (var art in articles)  
    Console.WriteLine(art);|
```

EFCore - Utilisation

```
C:\Users\antho\Desktop\hello.db  
ORM.Models.Article  
ORM.Models.Article  
art1  
art2
```

```
//modification d'une entité existante en base  
var toto = db.Articles.Single(art => art.ArticleId == 2);  
toto.Price++;  
db.SaveChanges();  
  
//supprimer une entité de la base de données  
db.Remove(article);  
db.SaveChanges();
```

EF Core - Utilisation





SGBDr– clés primaires

Une clé primaire est un attribut ou un ensemble d'attributs qui permet d'identifier de manière unique un enregistrement dans une table

Elle garantit l'unicité et l'intégrité des données

SGBDr – clé primaire

Unicité : chaque valeur de clé primaire doit être unique dans une table

Non-nullité: une clé primaire ne peut pas contenir de valeur NULL

Immuabilité: une clé primaire ne doit pas être modifiée fréquemment, car elle sert de référence à d'autres tables

Indexation automatique: la plupart des SGBD créent automatiquement un index sur la clé primaire pour améliorer les performances des requêtes

Clés primaires

Une clé primaire peut être simple :
une seule colonne

Elle peut aussi être composite:
plusieurs colonnes sont combinées
pour former une clé unique

En SQL, on utilise l'instruction
PRIMARY KEY

Parallèle avec Entity Framework

Dans Entity Framework, on va utiliser un attribut [Key] pour spécifier la clé primaire d'une entité dans un modèle de données.

Cet attribut permet de définir explicitement quel champ ou quelle propriété représente l'identifiant unique d'une entité

Par défaut, EF suppose d'une propriété nommée Id est la clé primaire.

Si ce n'est pas le cas ou si on veut définir une autre clé primaire personnalisée, on doit utiliser [Key]

```

public class Livre
{
    [Key]
    0 références
    public int LivreId { get; set; }

    0 références
    public string Titre { get; set; }

    // Clé étrangère
    0 références
    public int AuteurId { get; set; }

    // Navigation vers l'auteur
    0 références
    public Auteur Auteur { get; set; }
}

```

```

public class Auteur
{
    [Key]
    0 références
    public int AuteurId { get; set; }

    0 références
    public string Nom { get; set; }

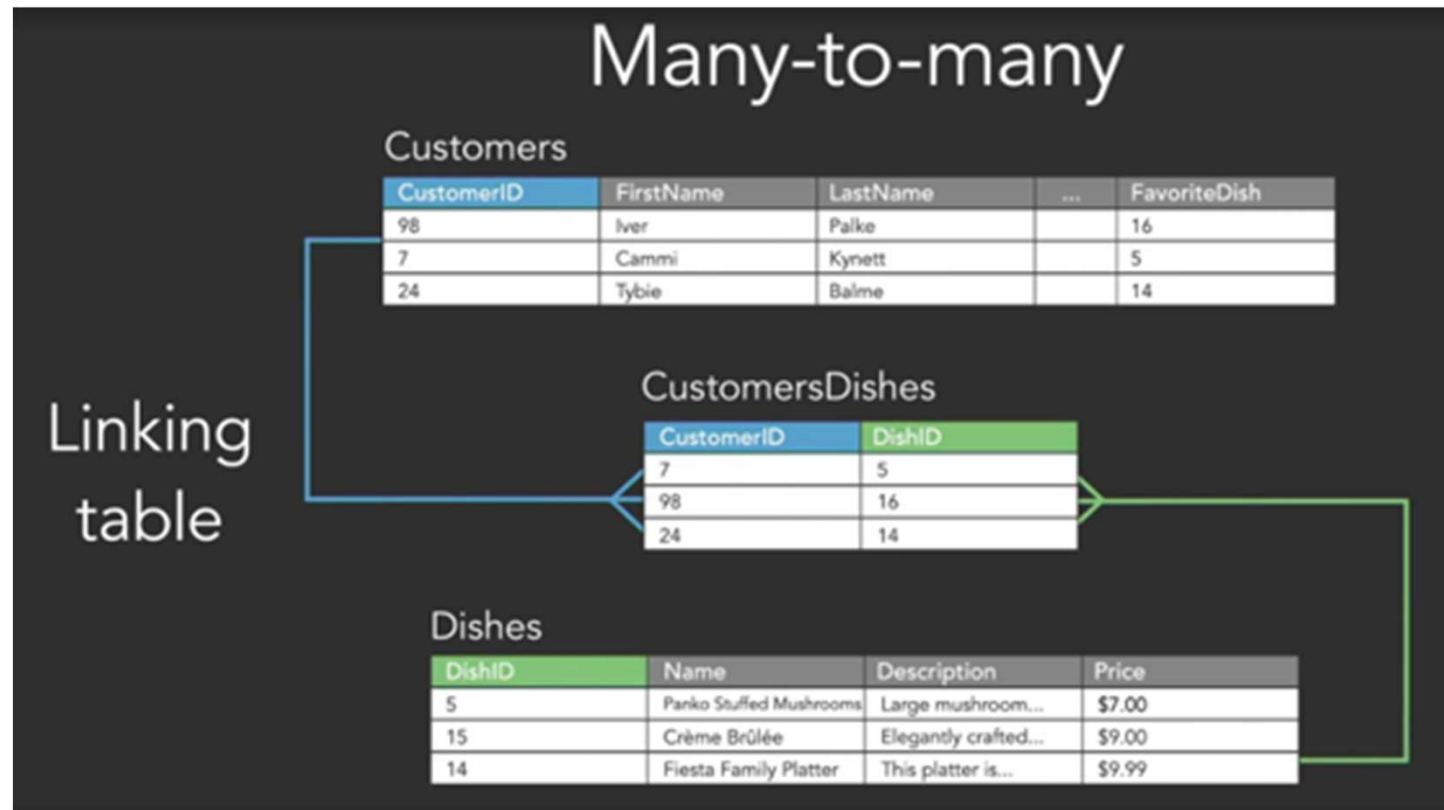
    // Navigation vers la liste des livres
    0 références
    public List<Livre> Livres { get; set; } = new();
}

```

Relation 1-to-Many (Un à plusieurs)

- Une relation un à plusieurs signifie qu'un enregistrement d'une table peut être lié à plusieurs enregistrements d'une autre table.
- Un auteur peut avoir écrit plusieurs livres. On suppose ici qu'un livre n'a qu'un auteur
- EF Core va générer automatiquement la table de jointure. Il est également possible de la surcharger en utilisant la méthode `OnModelCreating`

Relation Many-to-Many



```

1 référence
public class Post
{
    0 références
    public int Id { get; set; }
    0 références
    public List<PostTag> PostTags { get; } = [];
}

1 référence
public class Tag
{
    0 références
    public int Id { get; set; }
    0 références
    public List<PostTag> PostTags { get; } = [];
}

2 références
public class PostTag
{
    0 références
    public int PostsId { get; set; }
    0 références
    public int TagsId { get; set; }
    0 références
    public Post Post { get; set; } = null!;
    0 références
    public Tag Tag { get; set; } = null!;
}

```

```

1 référence
public class Post
{
    0 références
    public int Id { get; set; }
    0 références
    public List<Tag> Tags { get; } = [];
}

1 référence
public class Tag
{
    0 références
    public int Id { get; set; }
    0 références
    public List<Post> Posts { get; } = [];
}

```

Relation Many-to-Many

Relation Many-to-Many

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

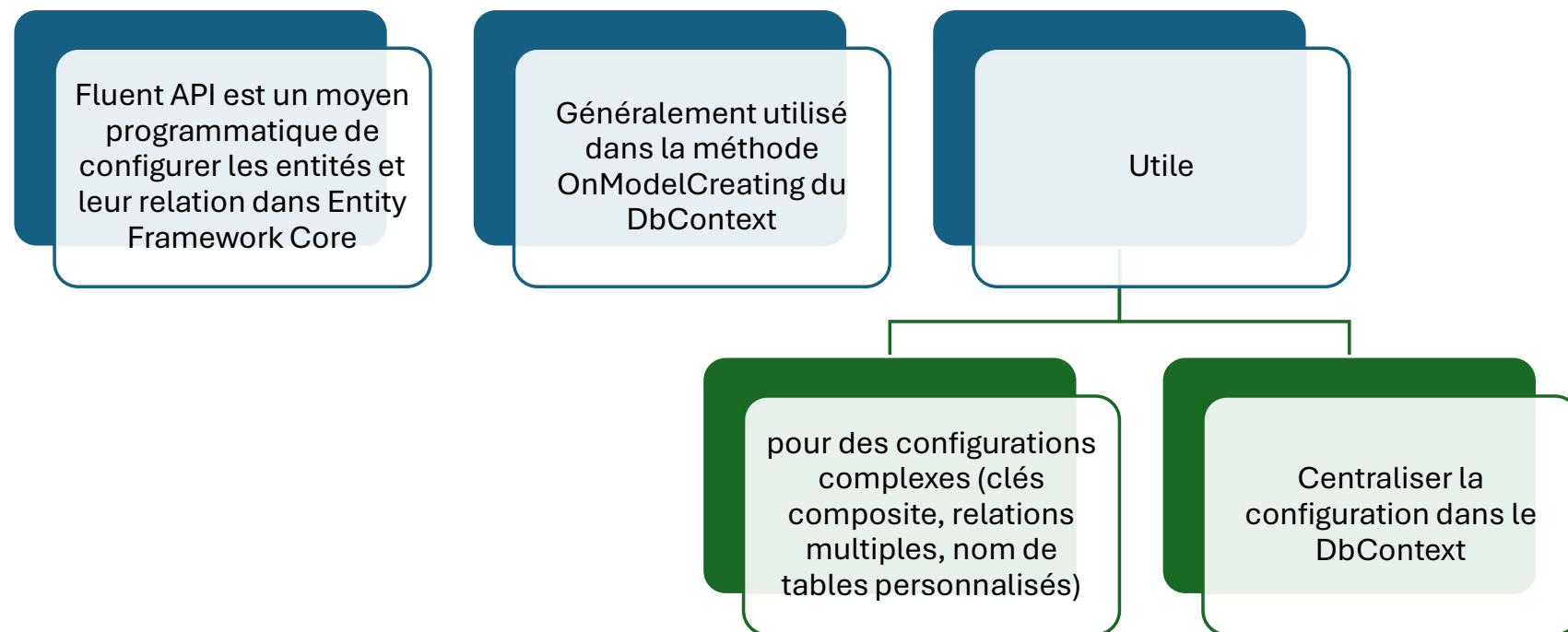
```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Post>()
        .HasMany(e => e.Tags)
        .WithMany(e => e.Posts)
        .UsingEntity(
            "PostTag",
            l => l.HasOne(typeof(Tag)).WithMany().HasForeignKey("TagsId").HasPrincipalKey(nameof(Tag.Id)),
            r => r.HasOne(typeof(Post)).WithMany().HasForeignKey("PostsId").HasPrincipalKey(nameof(Post.Id)),
            j => j.HasKey("PostsId", "TagsId"));
}
```

```
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);
```

Relation Many- To-Many



Fluent API



Fluent API

- C'est également une alternative ou un complément aux annotations type [Required] (nous verrons plus tard..)

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {  
    modelBuilder.Entity<Product>()  
        .Property(p => p.Name)  
        .IsRequired()  
        .HasMaxLength(100);  
  
    modelBuilder.Entity<Product>()  
        .Property(p => p.Price)  
        .HasColumnType("decimal(10,2)");  
}
```

```
public class Product {  
    public int Id { get; set; }  
    public string Name { get; set; }  
    public decimal Price { get; set; }  
}
```

Fluent API

Plus flexible que
les annotations

Configurations
avancées
possibles (clés
composites,
types SQL, etc...)

Code centralisé
dans
OnModelCreating

Souvent utilisé de
manière
complémentaire
aux Data
Annotations

Fluent API - TP

- Créer un projet Console
- Créer les entités Auteur et Livre.
- Dans cet exercice, un livre a un seul auteur, un auteur a écrit plusieurs livres
 - auteur : id, name, livres
 - Livre: id, titre, prix, date de publication, auteur
- Créer un LibraryContext qui dérive de DbContext et valoriser le OnModelCreating en définissant
 - Les clés
 - Name de auteur est obligatoire avec une taille max de 100
 - Titre du livre obligatoire, taille max 200
 - Le prix est en decimal (10,2)