Configuration des Relations par Convention

Relations : Navigation Property

Les relations par convention en EF core sont définies grâce à **la propriété de navigation**.
Le fournisseur de base de données ne pourrait pas mapper ce type à des types primitifs définis dans la base de données.

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

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

```
    constraints: table =>
    {
        table.PrimaryKey("PK_Books", x => x.BookId);
        table.ForeignKey(
            name: "FK_Books_Authors_AuthorId",
            column: x => x.AuthorId,
            principalTable: "Authors",
            principalColumn: "AuthorId",
            onDelete: ReferentialAction.Restrict);
    });
```

Navigation Property de cet exemple est Books définie dans Author.
A ce stade, EF Core génère une propriété shadow pour la clé étrangère appelée AuthorId Mappé à une colonne AuthorId nullable

Relations : Inverse Navigation Property

Naviguer dans l'entité dépendante (propriété Author de l'entité Book)

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

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

Relations : One to Many

Fully Defined RelationShip

$ Les propriétés de navigation ainsi que la clé étrangère sont définis dans les modèles
$ Suppression activé est en cascade étant donnée que la clé étrangère est obligatoire

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

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

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

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

Relations : Many to Many

Cette relation est présentée par l'introduction de deux collections de part et d'autre des classes d'entités.

```
public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public Author Author { get; set; }
    public ICollection<Category> Categories { get; set; }
}
```

```
public class Category
{
    public int CategoryId { get; set; }
    public string CategoryName { get; set; }
    public ICollection<Book> Books { get; set; }
}
```

Relations : Many to Many

```
public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public Author Author { get; set; }
    public ICollection<BookCategory> BookCategories { get; set; }
}

public class Category
{
    public int CategoryId { get; set; }
    public string CategoryName { get; set; }
    public ICollection<BookCategory> BookCategories { get; set; }
}

public class BookCategory
{
    public int BookId { get; set; }
    public Book Book { get; set; }
    public int CategoryId { get; set; }
    public Category Category { get; set; }
}
```

Une relation 1 à plusieurs défini des deux côtés des entités nous amène vers une relation plusieurs à plusieurs.

Représentée via une table de jointure contenant les clés des deux côtés.

Relations : Many to Many

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<BookCategory>()
        .HasKey(bc => new { bc.BookId, bc.CategoryId });

    modelBuilder.Entity<BookCategory>()
        .HasOne(bc => bc.Book)
        .WithMany(b => b.BookCategories)
        .HasForeignKey(bc => bc.BookId);

    modelBuilder.Entity<BookCategory>()
        .HasOne(bc => bc.Category)
        .WithMany(c => c.BookCategories)
        .HasForeignKey(bc => bc.CategoryId);
}
```

Relations : One To One

La relation One to One est configurée en identifiant l'entité principale et l'entité dépendante via la clé étrangère

Condition : le nom de la clé étrangère suit les conventions EF Core.

```
public class Author
{
    public int AuthorId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public AuthorBiography Biography { get; set; }
}

public class AuthorBiography
{
    public int AuthorBiographyId { get; set; }
    public string Biography { get; set; }
    public DateTime DateOfBirth { get; set; }
    public string PlaceOfBirth { get; set; }
    public string Nationality { get; set; }
    public int AuthorRef { get; set; }
    public Author Author { get; set; }
}
```

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Author>()
        .HasOne(a => a.Biography)
        .WithOne(b => b.Author)
        .HasForeignKey<AuthorBiography>(b => b.AuthorRef);
}
```

Définit dans le cas où la clé étrangère ne suit pas les conventions

API FLUENT

API Fluent

EF Core Fluent API configure les aspects du modèle suivants:

- **Configuration du modèle**: Cette partie permet de configurer le schéma par défault de la base de données, ces fonctions et des annotations de données additionnelles.

- **Configuration de l'entité**: Configurer les relations (PrimaryKey, Index, TableName, one-to-one, one-to-many…)

- **Configuration des propriétés**: Mapping des colonnes, noms des colonnes, nullabilité, types de données, etc.

API Fluent : mapping types et propriétés

Schéma par défaut

Spécifier le schéma de base de données à utiliser pour toutes les tables, procédures stockées, etc.

Ce paramètre par défaut est substitué pour tous les objets —> utiliser explicitement un schéma différent.

```
modelBuilder.HasDefaultSchema("sales");
```

Clé primaire

Définir explicitement une propriété comme clé primaire, vous pouvez utiliser la méthode HasKey.

```
modelBuilder.Entity<OfficeAssignment>().HasKey(t => t.InstructorID);
```

API Fluent : mapping types et propriétés

Clé primaire composite

L'exemple suivant configure les propriétés DepartmentID et Name comme clé primaire composite du type Department.

```
modelBuilder.Entity<Department>().HasKey(t => new { t.DepartmentID, t.Name });
```

Désactiver l'identité pour les clés primaires numériques

L'exemple suivant définit la propriété DepartmentID sur System.ComponentModel. DataAnnotations. DatabaseGeneratedOption. None pour indiquer que la valeur ne sera pas générée par la base de données.

```
modelBuilder.Entity<Department>().Property(t => t.DepartmentID)
    .HasDatabaseGeneratedOption(DatabaseGeneratedOption.None);
```

API Fluent : mapping types et propriétés

Longueur Max d'une propriété

La propriété Name ne doit pas dépasser 50 caractères.

Le cas contraire implique une exception DbEntityValidationException.

Si Code First crée une base de données à partir de ce modèle —> la longueur maximale de la colonne Name sur 50 caractères.

```
modelBuilder.Entity<Department>().Property(t => t.Name).HasMaxLength(50);
```

Propriété Obligatoire

La propriété Name est requise.

Si vous ne spécifiez pas le nom, vous obtiendrez une exception DbEntityValidationException.

Si Code First crée une base de données à partir de ce modèle, la colonne utilisée pour stocker cette propriété est généralement non nullable.

```
modelBuilder.Entity<Department>().Property(t => t.Name).IsRequired();
```

API Fluent : mapping types et propriétés

Ne pas mapper une propriété CLR à une colonne dans la base de données

Comment spécifier qu'une propriété sur un type CLR n'est pas mappée à une colonne de la base de données.

```
modelBuilder.Entity<Department>().Ignore(t => t.Budget);
```

Mappage d'une propriété CLR à une colonne spécifique dans la base de données

L'exemple suivant mappe la propriété CLR Name à la colonne de base de données DepartmentName.

```
modelBuilder.Entity<Department>()
    .Property(t => t.Name)
    .HasColumnName("DepartmentName");
```

API Fluent : mapping types et propriétés

Renommer d'une clé étrangère qui n'est pas définie dans le modèle

Si vous choisissez de ne pas définir de clé étrangère sur un type CLR, mais que vous souhaitez spécifier le nom qu'il doit avoir dans la base de données, procédez comme suit :

```
modelBuilder.Entity<Course>()
    .HasRequired(c => c.Department)
    .WithMany(t => t.Courses)
    .Map(m => m.MapKey("ChangedDepartmentID"));
```

Mappage d'une propriété CLR à une colonne spécifique dans la base de données

L'exemple suivant mappe la propriété CLR Name à la colonne de base de données DepartmentName.

```
modelBuilder.Entity<Department>()
    .Property(t => t.Name)
    .HasColumnName("DepartmentName");
```

API Fluent : mapping types et propriétés

Configuration du type de données d'une colonne de base de données

La méthode HasColumnType permet de mapper à différentes représentations du même type de base. L'utilisation de cette méthode ne vous permet pas d'effectuer une conversion des données au moment de l'exécution.

```
modelBuilder.Entity<Department>()
    .Property(p => p.Name)
    .HasColumnType("varchar");
```

Configuration des propriétés sur un type complexe

```
modelBuilder.ComplexType<Details>()
    .Property(t => t.Location)
    .HasMaxLength(20);
```

API Fluent : mapping types et propriétés

Mappage d’un type d’entité à une table spécifique dans la base de données

Toutes les propriétés du service sont mappées aux colonnes d’une table appelée t_ Department.

```
modelBuilder.Entity<Department>()
    .ToTable("t_Department");
```

```
modelBuilder.Entity<Department>()
    .ToTable("t_Department", "school");
```

API Fluent : Configuration relations

Configuration d'une relation obligatoire à facultative (un-à–zéro-ou-un)

OfficeAssignment a la propriété InstructorID qui est une clé primaire et une clé étrangère, car le nom de la propriété ne suit pas la convention. La méthode HasKey est utilisée pour configurer la clé primaire.

```
// Configure the primary key for the OfficeAssignment
modelBuilder.Entity<OfficeAssignment>()
    .HasKey(t => t.InstructorID);
```

```
// Map one-to-zero or one relationship
modelBuilder.Entity<OfficeAssignment>()
    .HasRequired(t => t.Instructor)
    .WithOptional(t => t.OfficeAssignment);
```

API Fluent : Configuration relations

Configuration d'une relation où les deux terminaisons sont obligatoires (un-à-un)

- Entity Framework peut déduire le type dépendant et le type principal dans une relation.
- Toutefois, lorsque les deux terminaisons de la relation sont obligatoires ou que les deux côtés sont facultatifs, Entity Framework ne peut pas identifier le type principal et le type dépendant.
- Lorsque les deux terminaisons de la relation sont obligatoires, utilisez WithRequiredPrincipal ou WithRequiredDependent après la méthode HasRequired.
- Lorsque les deux terminaisons de la relation sont facultatives, utilisez WithOptionalPrincipal ou WithOptionalDependent après la méthode HasOptional.

API Fluent : Configuration relations

```
// Configure the primary key for the OfficeAssignment
modelBuilder.Entity<OfficeAssignment>()
    .HasKey(t => t.InstructorID);

modelBuilder.Entity<Instructor>()
    .HasRequired(t => t.OfficeAssignment)
    .WithRequiredPrincipal(t => t.Instructor);
```

Configuration d'une relation plusieurs à plusieurs

Dans l'exemple suivant, les conventions Code First par défaut sont utilisées pour créer une table de jointure. Par conséquent, la table CourseInstructor est créée avec des colonnes Course_CourseID et Instructor_InstructorID.

```
modelBuilder.Entity<Course>()
    .HasMany(t => t.Instructors)
    .WithMany(t => t.Courses)
```

API Fluent : Configuration relations

- Si vous souhaitez spécifier le nom de la table de jointure et les noms des colonnes de la table, vous devez effectuer une configuration supplémentaire à l'aide de la méthode Map.
- Le code suivant génère la table CourseInstructor avec les colonnes CourseID et InstructorID.

```
modelBuilder.Entity<Course>()
    .HasMany(t => t.Instructors)
    .WithMany(t => t.Courses)
    .Map(m =>
    {
        m.ToTable("CourseInstructor");
        m.MapLeftKey("CourseID");
        m.MapRightKey("InstructorID");
    });
```

API Fluent : Configuration relations

Configuration d'une relation avec une propriété de navigation

- Une relation unidirectionnelle existe lorsqu'une propriété de navigation est définie sur une seule terminaison de la relation, et non sur les deux.
- Par convention, Code First interprète toujours une relation unidirectionnelle comme une relation une-à-plusieurs.
- Par exemple, une relation un-à-un entre Instructor et OfficeAssignment, où vous disposez d'une propriété de navigation uniquement sur le type Instructor, doit utiliser l'API Fluent pour configurer cette relation.

```
// Configure the primary Key for the OfficeAssignment
modelBuilder.Entity<OfficeAssignment>()
    .HasKey(t => t.InstructorID);

modelBuilder.Entity<Instructor>()
    .HasRequired(t => t.OfficeAssignment)
    .WithRequiredPrincipal();
```

https://learn.microsoft.com/fr-fr/ef/ef6/modeling/code-first/fluent/relationships

API Fluent : Configuration relations

Activation de la suppression en cascade

- Si une clé étrangère sur l'entité dépendante ne peut pas accepter la valeur Null, Code First définit la suppression en cascade sur la relation.
- Si une clé étrangère sur l'entité dépendante ne peut pas accepter la valeur Null, Code First ne définit pas de suppression en cascade sur la relation, et lorsque le principal est supprimé, la clé étrangère est définie sur Null.

```
modelBuilder.Entity<Course>()
    .HasRequired(t => t.Department)
    .WithMany(t => t.Courses)
    .HasForeignKey(d => d.DepartmentID)
    .WillCascadeOnDelete(false);
```

https://learn.microsoft.com/fr-fr/ef/ef6/modeling/code-first/fluent/relationships

API Fluent : mapping des relations

- On commence par l'instance EntityTypeConfiguration
- On utilise la méthode HasRequired, HasOptional ou HasMany pour spécifier le type de relation que cette entité utilise.
- Les méthodes HasRequired et HasOptional prennent une expression lambda qui représente une propriété de navigation de référence.
- La méthode HasMany prend une expression lambda qui représente une propriété de navigation de collection.

API Fluent : mapping des relations

- On configure une propriété de navigation inversé à l'aide des méthodes WithRequired, WithOptional et WithMany.

- On configurer les propriétés de clé étrangère à l'aide de la méthode HasForeignKey. Cette méthode accepte une expression lambda qui représente la propriété à utiliser comme clé étrangère.

Configuration d'une relation obligatoire à facultative (un à zéro ou un) : exemple

```
// Configure the primary key for the OfficeAssignment
modelBuilder.Entity<OfficeAssignment>()
    .HasKey(t => t.InstructorID);

// Map one-to-zero or one relationship
modelBuilder.Entity<OfficeAssignment>()
    .HasRequired(t => t.Instructor)
    .WithOptional(t => t.OfficeAssignment);
```

Clé facultative

Clé obligatoire

```
// Configure the primary key for the
OfficeAssignment
modelBuilder.Entity<OfficeAssignment>()
    .HasKey(t => t.InstructorID);

modelBuilder.Entity<Instructor>()
    .HasRequired(t => t.OfficeAssignment)
    .WithRequiredPrincipal(t => t.Instructor);
```

Configuration d’une relation plusieurs à plusieurs

```
modelBuilder.Entity<Course>()
    .HasMany(t => t.Instructors)
    .WithMany(t => t.Courses)
```

Conventions utilisées pour créer la table de jointure

Spécifier le nom de la table de jointure

```
modelBuilder.Entity<Course>()
    .HasMany(t => t.Instructors)
    .WithMany(t => t.Courses)
    .Map(m =>
    {
        m.ToTable("CourseInstructor");
        m.MapLeftKey("CourseID");
        m.MapRightKey("InstructorID");
    });
```